The Python Library Reference

发布 *3.6.15*

Guido van Rossum
and the Python development team

九月 05, 2021

Python Software Foundation
Email: docs@python.org

Contents

reference-index 描述了 Python 语言的具体语法和语义，这份库参考则介绍了与 Python 一同发行的标准库。它还描述了通常包含在 Python 发行版中的一些可选组件。

Python 标准库非常庞大，所提供的组件涉及范围十分广泛，正如以下内容目录所显示的。这个库包含了多个内置模块 (以 C 编写)，Python 程序员必须依靠它们来实现系统级功能，例如文件 I/O，此外还有大量以 Python 编写的模块，提供了日常编程中许多问题的标准解决方案。其中有些模块经过专门设计，通过将特定平台功能抽象化为平台中立的 API 来鼓励和加强 Python 程序的可移植性。

Windows 版本的 Python 安装程序通常包含整个标准库，往往还包含许多额外组件。对于类 Unix 操作系统，Python 通常会分成一系列的软件包，因此可能需要使用操作系统所提供的包管理工具来获取部分或全部可选组件。

在这个标准库以外还存在成千上万并且不断增加的其他组件 (从单独的程序、模块、软件包直到完整的应用开发框架)，访问 Python 包索引 即可获取这些第三方包。

CHAPTER 1

概述

“Python 库”中包含了几种不同的组件。

它包含通常被视为语言“核心”中的一部分的数据类型，例如数字和列表。对于这些类型，Python 语言核心定义了文字的形式，并对它们的语义设置了一些约束，但没有完全定义语义。（另一方面，语言核心确实定义了语法属性，如操作符的拼写和优先级。）

这个库也包含了内置函数和异常—不需要 `import` 语句就可以在所有 Python 代码中使用的对象。有一些是由语言核心定义的，但是许多对于核心语义不是必需的，并且仅在这里描述。

不过这个库主要是由一系列的模块组成。这些模块集可以不同方式分类。有些模块是用 C 编写并内置于 Python 解释器中；另一些模块则是用 Python 编写并以源码形式导入。有些模块提供专用于 Python 的接口，例如打印栈追踪信息；有些模块提供专用于特定操作系统的接口，例如操作特定的硬件；另一些模块则提供针对特定应用领域的接口，例如万维网。有些模块在所有更新和移植版本的 Python 中可用；另一些模块仅在底层系统支持或要求时可用；还有些模块则仅当编译和安装 Python 时选择了特定配置选项时才可用。

本手册以“从内到外”的顺序组织：首先描述内置函数、数据类型和异常，最后是根据相关性进行分组的各种模块。

这意味着如果你从头开始阅读本手册，并在感到厌烦时跳到下一章，你仍能对 Python 库的可用模块和所支持的应用领域有个大致了解。当然，你并非*必须*如同读小说一样从头读到尾—你也可以先浏览内容目录 (在手册开头)，或在索引 (在手册末尾) 中查找某个特定函数、模块或条目。最后，如果你喜欢随意学习某个主题，你可以选择一个随机页码 (参见 *random* 模块) 并读上一两小节。无论你想以怎样的顺序阅读本手册，还是建议先从内置函数 这一章开始，因为本手册的其余内容都需要你熟悉其中的基本概念。

让我们开始吧！

CHAPTER 2

内置函数

Python 解释器内置了很多函数和类型，您可以在任何时候使用它们。以下按字母表顺序列出它们。

		内置函数		
abs()	*dict()*	*help()*	*min()*	*setattr()*
all()	*dir()*	*hex()*	*next()*	*slice()*
any()	*divmod()*	*id()*	*object()*	*sorted()*
ascii()	*enumerate()*	*input()*	*oct()*	*staticmethod()*
bin()	*eval()*	*int()*	*open()*	*str()*
bool()	*exec()*	*isinstance()*	*ord()*	*sum()*
bytearray()	*filter()*	*issubclass()*	*pow()*	*super()*
bytes()	*float()*	*iter()*	*print()*	*tuple()*
callable()	*format()*	*len()*	*property()*	*type()*
chr()	*frozenset()*	*list()*	*range()*	*vars()*
classmethod()	*getattr()*	*locals()*	*repr()*	*zip()*
compile()	*globals()*	*map()*	*reversed()*	*__import__()*
complex()	*hasattr()*	*max()*	*round()*	
delattr()	*hash()*	*memoryview()*	*set()*	

abs(*x*)
返回一个数的绝对值。实参可以是整数或浮点数。如果实参是一个复数，返回它的模。

all(*iterable*)
如果 *iterable* 的所有元素为真（或迭代器为空），返回 `True` 。等价于:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any(*iterable*)
如果 *iterable* 的任一元素为真则返回 `True`。如果迭代器为空，返回 `False`。等价于:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

ascii(*object*)

与*repr()* 类似，返回一个包含对象的可打印表示形式的字符串，但是使用 \x、\u 和 \U 对*repr()* 返回的字符串中非 ASCII 编码的字符进行转义。生成的字符串和 Python 2 的*repr()* 返回的结果相似。

bin(*x*)

将一个整数转变为一个前缀为“0b”的二进制字符串。结果是一个合法的 Python 表达式。如果 *x* 不是 Python 的*int* 对象，那它需要定义 __index__() 方法返回一个整数。一些例子：

```
>>> bin(3)
'0b11'
>>> bin(-10)
'-0b1010'
```

如果不一定需要前缀“0b”，还可以使用如下的方法。

```
>>> format(14, '#b'), format(14, 'b')
('0b1110', '1110')
>>> f'{14:#b}', f'{14:b}'
('0b1110', '1110')
```

另见*format()* 获取更多信息。

class bool([*x*])

返回一个布尔值，True 或者 False。*x* 使用标准的*真值测试过程* 来转换。如果 *x* 是假的或者被省略，返回 False；其他情况返回 True。*bool* 类是*int* 的子类（参见*数字类型—int, float, complex*）。其他类不能继承自它。它只有 False 和 True 两个实例（参见*布尔值*）。

class bytearray([*source*[, *encoding*[, *errors*]]])

返回一个新的 bytes 数组。*bytearray* 类是一个可变序列，包含范围为 0 <= x < 256 的整数。它有可变序列大部分常见的方法，见*可变序列类型* 的描述；同时有*bytes* 类型的大部分方法，参见*bytes 和 bytearray 操作*。

可选形参 *source* 可以用不同的方式来初始化数组：

- 如果是一个 *string*，您必须提供 *encoding* 参数（*errors* 参数仍是可选的）；*bytearray()* 会使用*str.encode()* 方法来将 string 转变成 bytes。
- 如果是一个 *integer*，会初始化大小为该数字的数组，并使用 null 字节填充。
- 如果是一个符合 *buffer* 接口的对象，该对象的只读 buffer 会用来初始化字节数组。
- 如果是一个 *iterable* 可迭代对象，它的元素的范围必须是 0 <= x < 256 的整数，它会被用作数组的初始内容。

如果没有实参，则创建大小为 0 的数组。

另见*二进制序列类型—bytes, bytearray, memoryview* 和*bytearray 对象*。

class bytes([*source*[, *encoding*[, *errors*]]])

返回一个新的“bytes”对象，是一个不可变序列，包含范围为 0 <= x < 256 的整数。*bytes* 是*bytearray* 的不可变版本 - 它有其中不改变序列的方法和相同的索引、切片操作。

因此，构造函数的实参和*bytearray()* 相同。

字节对象还可以用字面值创建，参见 strings。

另见二进制序列类型—*bytes, bytearray, memoryview*，*bytes* 对象 和*bytes* 和 *bytearray* 操作。

callable(*object*)

Return *True* if the *object* argument appears callable, *False* if not. If this returns true, it is still possible that a call fails, but if it is false, calling *object* will never succeed. Note that classes are callable (calling a class returns a new instance); instances are callable if their class has a `__call__()` method.

3.2 新版功能: 这个函数一开始在 Python 3.0 被移除了，但在 Python 3.2 被重新加入。

chr(*i*)

返回 Unicode 码位为整数 *i* 的字符的字符串格式。例如，`chr(97)` 返回字符串 `'a'`，`chr(8364)` 返回字符串 `'€'`。这是*ord()* 的逆函数。

实参的合法范围是 0 到 1,114,111（16 进制表示是 0x10FFFF）。如果 *i* 超过这个范围，会触发*ValueError* 异常。

@classmethod

把一个方法封装成类方法。

一个类方法把类自己作为第一个实参，就像一个实例方法把实例自己作为第一个实参。请用以下习惯来声明类方法:

```
class C:
    @classmethod
    def f(cls, arg1, arg2, ...): ...
```

The `@classmethod` form is a function *decorator* –see the description of function definitions in function for details.

It can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see *staticmethod()* in this section.

For more information on class methods, consult the documentation on the standard type hierarchy in types.

compile(*source*, *filename*, *mode*, *flags=0*, *dont_inherit=False*, *optimize=-1*)

将 *source* 编译成代码或 AST 对象。代码对象可以被*exec()* 或*eval()* 执行。*source* 可以是常规的字符串、字节字符串，或者 AST 对象。参见*ast* 模块的文档了解如何使用 AST 对象。

filename 实参需要是代码读取的文件名；如果代码不需要从文件中读取，可以传入一些可辨识的值（经常会使用 `'<string>'`）。

mode 实参指定了编译代码必须用的模式。如果 *source* 是语句序列，可以是 `'exec'`；如果是单一表达式，可以是 `'eval'`；如果是单个交互式语句，可以是 `'single'`。（在最后一种情况下，如果表达式执行结果不是 `None` 将会被打印出来。）

可选参数 *flags* 和 *dont_inherit* 控制在编译 *source* 时要用到哪个 future 语句。如果两者都未提供（或都为零）则会使用调用*compile()* 的代码中有效的 future 语句来编译代码。如果给出了 *flags* 参数但没有 *dont_inherit* (或是为零) 则 *flags* 参数所指定的以及那些无论如何都有效的 future 语句会被使用。如果 *dont_inherit* 为一个非零整数，则只使用 *flags* 参数–在调用外围有效的 future 语句将被忽略。

Future 语句使用比特位来指定，多个语句可以通过按位或来指定。具体特性的比特位可以通过*__future__* 模块中的 `_Feature` 类的实例的 `compiler_flag` 属性来获得。

optimize 实参指定编译器的优化级别；默认值 `-1` 选择与解释器的 `-O` 选项相同的优化级别。显式级别为 `0`（没有优化；`__debug__` 为真）、`1`（断言被删除，`__debug__` 为假）或 `2`（文档字符串也被删除）。

如果编译的源码不合法，此函数会触发*SyntaxError* 异常；如果源码包含 null 字节，则会触发*ValueError* 异常。

如果您想分析 Python 代码的 AST 表示，请参阅*ast.parse()*。

注解: 在 'single' 或 'eval' 模式编译多行代码字符串时，输入必须以至少一个换行符结尾。这使*code* 模块更容易检测语句的完整性。

警告: 在将足够大或者足够复杂的字符串编译成 AST 对象时，Python 解释器有可以因为 Python AST 编译器的栈深度限制而崩溃。

在 3.2 版更改: 允许使用 Windows 和 Mac 的换行符。在 'exec' 模式不再需要以换行符结尾。增加了 *optimize* 形参。

在 3.5 版更改: 之前 *source* 中包含 null 字节的话会触发*TypeError* 异常。

class complex([*real*[, *imag*]])

返回值为 *real* + *imag**1j 的复数，或将字符串或数字转换为复数。如果第一个形参是字符串，则它被解释为一个复数，并且函数调用时必须没有第二个形参。第二个形参不能是字符串。每个实参都可以是任意的数值类型（包括复数）。如果省略了 *imag*，则默认值为零，构造函数会像*int* 和*float* 一样进行数值转换。如果两个实参都省略，则返回 0j。

注解: 当从字符串转换时，字符串在 + 或 - 的周围必须不能有空格。例如 complex('1+2j') 是合法的，但 complex('1 + 2j') 会触发*ValueError* 异常。

数字类型—*int, float, complex* 描述了复数类型。

在 3.6 版更改: 您可以使用下划线将代码文字中的数字进行分组。

delattr(*object*, *name*)

setattr() 相关的函数。实参是一个对象和一个字符串。该字符串必须是对象的某个属性。如果对象允许，该函数将删除指定的属性。例如 delattr(x, 'foobar') 等价于 del x.foobar 。

class dict(***kwarg*)
class dict(*mapping*, ***kwarg*)
class dict(*iterable*, ***kwarg*)

创建一个新的字典。*dict* 对象是一个字典类。参见*dict* 和映射类型—*dict* 了解这个类。

其他容器类型，请参见内置的*list*、*set* 和*tuple* 类，以及*collections* 模块。

dir([*object*])

如果没有实参，则返回当前本地作用域中的名称列表。如果有实参，它会尝试返回该对象的有效属性列表。

如果对象有一个名为 __dir__() 的方法，那么该方法将被调用，并且必须返回一个属性列表。这允许实现自定义 __getattr__() 或 __getattribute__() 函数的对象能够自定义*dir()* 来报告它们的属性。

如果对象不提供 __dir__()，这个函数会尝试从对象已定义的*__dict__* 属性和类型对象收集信息。结果列表并不总是完整的，如果对象有自定义 __getattr__()，那结果可能不准确。

默认的*dir()* 机制对不同类型的对象行为不同，它会试图返回最相关而不是最全的信息：

- 如果对象是模块对象，则列表包含模块的属性名称。
- 如果对象是类型或类对象，则列表包含它们的属性名称，并且递归查找所有基类的属性。
- 否则，列表包含对象的属性名称，它的类属性名称，并且递归查找它的类的所有基类的属性。

返回的列表按字母表排序。例如：

```
>>> import struct
>>> dir()   # show the names in the module namespace
['__builtins__', '__name__', 'struct']
>>> dir(struct)   # show the names in the struct module
['Struct', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__initializing__', '__loader__', '__name__', '__package__',
 '_clearcache', 'calcsize', 'error', 'pack', 'pack_into',
 'unpack', 'unpack_from']
>>> class Shape:
...     def __dir__(self):
...         return ['area', 'perimeter', 'location']
>>> s = Shape()
>>> dir(s)
['area', 'location', 'perimeter']
```

注解： 因为*dir()* 主要是为了便于在交互式时使用，所以它会试图返回人们感兴趣的名字集合，而不是试图保证结果的严格性或一致性，它具体的行为也可能在不同版本之间改变。例如，当实参是一个类时，metaclass 的属性不包含在结果列表中。

divmod(*a*, *b*)

它将两个（非复数）数字作为实参，并在执行整数除法时返回一对商和余数。对于混合操作数类型，适用双目算术运算符的规则。对于整数，结果和 `(a // b, a % b)` 一致。对于浮点数，结果是 `(q, a % b)`，*q* 通常是 `math.floor(a / b)` 但可能会比 1 小。在任何情况下，`q * b + a % b` 和 *a* 基本相等；如果 `a % b` 非零，它的符号和 *b* 一样，并且 `0 <= abs(a % b) < abs(b)`。

enumerate(*iterable*, *start=0*)

返回一个枚举对象。*iterable* 必须是一个序列，或*iterator*，或其他支持迭代的对象。*enumerate()* 返回的迭代器的*__next__()* 方法返回一个元组，里面包含一个计数值（从 *start* 开始，默认为 0）和通过迭代 *iterable* 获得的值。

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

等价于:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

eval(*expression*, *globals=None*, *locals=None*)

实参是一个字符串，以及可选的 globals 和 locals。*globals* 实参必须是一个字典。*locals* 可以是任何映射对象。

expression 参数会作为一个 Python 表达式（从技术上说是一个条件列表）被解析并求值，使用 *globals* 和 *locals* 字典作为全局和局部命名空间。如果 *globals* 字典存在且不包含以 `__builtins__` 为键的值，则会在解析 *expression* 之前插入以此为键的对内置模块*builtins* 的字典的引用。这意味着 *expression* 通常具有对标准*builtins* 模块的完全访问权限且受限的环境会被传播。如果省略 *locals* 字典则其默认值为 *globals* 字典。如果两个字典同时省略，表达式会在*eval()* 被调用的环境中执行。返回值为表达式求值的结果。语法错误将作为异常被报告。例如:

```
>>> x = 1
>>> eval('x+1')
2
```

这个函数也可以用来执行任何代码对象（如*compile()* 创建的）。这种情况下，参数是代码对象，而不是字符串。如果编译该对象时的 *mode* 实参是 `'exec'` 那么*eval()* 返回值为 `None` 。

提示：*exec()* 函数支持动态执行语句。*globals()* 和*locals()* 函数各自返回当前的全局和本地字典，因此您可以将它们传递给*eval()* 或*exec()* 来使用。

另外可以参阅*ast.literal_eval()*，该函数可以安全执行仅包含文字的表达式字符串。

exec(*object*[, *globals*[, *locals*]])

这个函数支持动态执行 Python 代码。*object* 必须是字符串或者代码对象。如果是字符串，那么该字符串将被解析为一系列 Python 语句并执行（除非发生语法错误）。[1] 如果是代码对象，它将被直接执行。在任何情况下，被执行的代码都需要和文件输入一样是有效的（见参考手册中关于文件输入的章节）。请注意即使在传递给*exec()* 函数的代码的上下文中，`return` 和 `yield` 语句也不能在函数定义之外使用。该函数返回值是 `None` 。

无论哪种情况，如果省略了可选参数，代码将在当前范围内执行。如果提供了 *globals* 参数，就必须是字典类型，而且会被用作全局和本地变量。如果同时提供了 *globals* 和 *locals* 参数，它们分别被用作全局和本地变量。如果提供了 *locals* 参数，则它可以是任何映射型的对象。请记住在模块层级，全局和本地变量是相同的字典。如果 exec 有两个不同的 *globals* 和 *locals* 对象，代码就像嵌入在类定义中一样执行。

如果 *globals* 字典不包含 `__builtins__` 键值，则将为该键插入对内建*builtins* 模块字典的引用。因此，在将执行的代码传递给*exec()* 之前，可以通过将自己的 `__builtins__` 字典插入到 *globals* 中来控制可以使用哪些内置代码。

注解： 内置*globals()* 和*locals()* 函数各自返回当前的全局和本地字典，因此可以将它们传递给*exec()* 的第二个和第三个实参。

注解： 默认情况下，*locals* 的行为如下面*locals()* 函数描述的一样：不要试图改变默认的 *locals* 字典。如果您想在*exec()* 函数返回时知道代码对 *locals* 的变动，请明确地传递 *locals* 字典。

---

filter(*function*, *iterable*)

用 *iterable* 中函数 *function* 返回真的那些元素，构建一个新的迭代器。*iterable* 可以是一个序列，一个支持迭代的容器，或一个迭代器。如果 *function* 是 `None` ，则会假设它是一个身份函数，即 *iterable* 中所有返回假的元素会被移除。

请注意，`filter(function, iterable)` 相当于一个生成器表达式，当 function 不是 `None` 的时候为 `(item for item in iterable if function(item))`；function 是 `None` 的时候为 `(item for item in iterable if item)` 。

请参阅*itertools.filterfalse()* 了解，只有 *function* 返回 false 时才选取 *iterable* 中元素的补充函数。

class float([*x*])

返回从数字或字符串 *x* 生成的浮点数。

如果实参是字符串，则它必须是包含十进制数字的字符串，字符串前面可以有符号，之前也可以有空格。可选的符号有 `'+'` 和 `'-'` ； `'+'` 对创建的值没有影响。实参也可以是 NaN（非数字）、正负无穷大的字符串。确切地说，除去首尾的空格后，输入必须遵循以下语法：

[1] 解析器只接受 Unix 风格的行结束符。如果您从文件中读取代码，请确保用换行符转换模式转换 Windows 或 Mac 风格的换行符。

```
sign            ::=  "+" | "-"
infinity        ::=  "Infinity" | "inf"
nan             ::=  "nan"
numeric_value   ::=  floatnumber | infinity | nan
numeric_string  ::=  [sign] numeric_value
```

这里，floatnumber 是 Python 浮点数的字符串形式，详见 floating。字母大小写都可以，例如，“inf”、“Inf”、“INFINITY”、“iNfINity” 都可以表示正无穷大。

另一方面，如果实参是整数或浮点数，则返回具有相同值（在 Python 浮点精度范围内）的浮点数。如果实参在 Python 浮点精度范围外，则会触发*OverflowError*。

对于一般的 Python 对象 x，float(x) 指派给 x.__float__()。

如果没有实参，则返回 0.0。

示例:

```
>>> float('+1.23')
1.23
>>> float('   -12345\n')
-12345.0
>>> float('1e-003')
0.001
>>> float('+1E6')
1000000.0
>>> float('-Infinity')
-inf
```

数字类型—int, float, complex 描述了浮点类型。

在 3.6 版更改: 您可以使用下划线将代码文字中的数字进行分组。

format(*value*[, *format_spec*])
将 *value* 转换为 *format_spec* 控制的“格式化”表示。*format_spec* 的解释取决于 *value* 实参的类型，但是大多数内置类型使用标准格式化语法：*格式规格迷你语言*。

默认的 *format_spec* 是一个空字符串，它通常给出与调用*str(value)* 相同的结果。

调用 format(value, format_spec) 会转换成 type(value).__format__(value, format_spec)，所以实例字典中的 __format__() 方法将不会调用。如果方法搜索回退到*object* 类但 *format_spec* 不为空，或者如果 *format_spec* 或返回值不是字符串，则会触发*TypeError* 异常。

在 3.4 版更改: 当 *format_spec* 不是空字符串时，object().__format__(format_spec) 会触发*TypeError*。

class frozenset([*iterable*])
返回一个新的*frozenset* 对象，它包含可选参数 *iterable* 中的元素。frozenset 是一个内置的类。有关此类的文档，请参阅*frozenset* 和*集合类型—set, frozenset*。

请参阅内建的*set*、*list*、*tuple* 和*dict* 类，以及*collections* 模块来了解其它的容器。

getattr(*object*, *name*[, *default*])
返回对象命名属性的值。*name* 必须是字符串。如果该字符串是对象的属性之一，则返回该属性的值。例如，getattr(x, 'foobar') 等同于 x.foobar。如果指定的属性不存在，且提供了 *default* 值，则返回它，否则触发*AttributeError*。

globals()
返回表示当前全局符号表的字典。这总是当前模块的字典（在函数或方法中，不是调用它的模块，而是定义它的模块）。

hasattr(*object*, *name*)
该实参是一个对象和一个字符串。如果字符串是对象的属性之一的名称，则返回 True，否则返回 False。（此功能是通过调用 getattr(object, name) 看是否有*AttributeError* 异常来实现的。）

hash(*object*)
返回该对象的哈希值（如果它有的话）。哈希值是整数。它们在字典查找元素时用来快速比较字典的键。相同大小的数字变量有相同的哈希值（即使它们类型不同，如 1 和 1.0）。

注解： 如果对象实现了自己的 __hash__() 方法，请注意，*hash()* 根据机器的字长来截断返回值。另请参阅 __hash__()。

---

help([*object*])
启动内置的帮助系统（此函数主要在交互式中使用）。如果没有实参，解释器控制台里会启动交互式帮助系统。如果实参是一个字符串，则在模块、函数、类、方法、关键字或文档主题中搜索该字符串，并在控制台上打印帮助信息。如果实参是其他任意对象，则会生成该对象的帮助页。

该函数通过*site* 模块加入到内置命名空间。

在 3.4 版更改: *pydoc* 和*inspect* 的变更使得可调用对象的签名信息更加全面和一致。

hex(*x*)
将整数转换为以“0x”为前缀的小写十六进制字符串。如果 *x* 不是 Python *int* 对象，则必须定义返回整数的 __index__() 方法。一些例子：

```
>>> hex(255)
'0xff'
>>> hex(-42)
'-0x2a'
```

如果要将整数转换为大写或小写的十六进制字符串，并可选择有无“0x”前缀，则可以使用如下方法：

```
>>> '%#x' % 255, '%x' % 255, '%X' % 255
('0xff', 'ff', 'FF')
>>> format(255, '#x'), format(255, 'x'), format(255, 'X')
('0xff', 'ff', 'FF')
>>> f'{255:#x}', f'{255:x}', f'{255:X}'
('0xff', 'ff', 'FF')
```

另见*format()* 获取更多信息。

另请参阅*int()* 将十六进制字符串转换为以 16 为基数的整数。

注解： 如果要获取浮点数的十六进制字符串形式，请使用*float.hex()* 方法。

---

id(*object*)
返回对象的“标识值”。该值是一个整数，在此对象的生命周期中保证是唯一且恒定的。两个生命期不重叠的对象可能具有相同的*id()* 值。

CPython implementation detail: This is the address of the object in memory.

input([*prompt*])
如果存在 *prompt* 实参，则将其写入标准输出，末尾不带换行符。接下来，该函数从输入中读取一行，将其转换为字符串（除了末尾的换行符）并返回。当读取到 EOF 时，则触发*EOFError*。例如:

```
>>> s = input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

如果加载了*readline*模块，*input()* 将使用它来提供复杂的行编辑和历史记录功能。

class int(*x=0*)
class int(*x*, *base=10*)

返回一个使用数字或字符串 *x* 生成的整数对象，或者没有实参的时候返回 `0`。如果 *x* 定义了 `__int__()`，`int(x)` 返回 `x.__int__()`。如果 *x* 定义了 `__trunc__()`，它返回 `x.__trunc__()`。对于浮点数，它向零舍入。

如果 *x* 不是数字，或者有 *base* 参数，*x* 必须是字符串、*bytes*、表示进制为 *base* 的整数字面值的*bytearray* 实例。该文字前可以有 `+` 或 `-`（中间不能有空格），前后可以有空格。一个进制为 n 的数字包含 0 到 n-1 的数，其中 `a` 到 `z`（或 `A` 到 `Z`）表示 10 到 35。默认的 *base* 为 10，允许的进制有 0、2-36。2、8、16 进制的数字可以在代码中用 `0b`/`0B`、`0o`/`0O`、`0x`/`0X` 前缀来表示。进制为 0 将安照代码的字面量来精确解释，最后的结果会是 2、8、10、16 进制中的一个。所以 `int('010', 0)` 是非法的，但 `int('010')` 和 `int('010', 8)` 是合法的。

整数类型定义请参阅数字类型—*int, float, complex*。

在 3.4 版更改: 如果 *base* 不是*int* 的实例，但 *base* 对象有 `base.__index__` 方法，则会调用该方法来获取进制数。以前的版本使用 `base.__int__` 而不是 `base.__index__`。

在 3.6 版更改: 您可以使用下划线将代码文字中的数字进行分组。

isinstance(*object*, *classinfo*)

Return true if the *object* argument is an instance of the *classinfo* argument, or of a (direct, indirect or *virtual*) subclass thereof. If *object* is not an object of the given type, the function always returns false. If *classinfo* is a tuple of type objects (or recursively, other such tuples), return true if *object* is an instance of any of the types. If *classinfo* is not a type or tuple of types and such tuples, a *TypeError* exception is raised.

issubclass(*class*, *classinfo*)

Return true if *class* is a subclass (direct, indirect or *virtual*) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects, in which case every entry in *classinfo* will be checked. In any other case, a *TypeError* exception is raised.

iter(*object*[, *sentinel*])

返回一个*iterator* 对象。根据是否存在第二个实参，第一个实参的解释是非常不同的。如果没有第二个实参，*object* 必须是支持迭代协议（有 `__iter__()` 方法）的集合对象，或必须支持序列协议（有 `__getitem__()` 方法，且数字参数从 `0` 开始）。如果它不支持这些协议，会触发*TypeError*。如果有第二个实参 *sentinel*，那么 *object* 必须是可调用的对象。这种情况下生成的迭代器，每次迭代调用它的*__next__()* 方法时都会不带实参地调用 *object*；如果返回的结果是 *sentinel* 则触发*StopIteration*，否则返回调用结果。

另请参阅迭代器类型。

One useful application of the second form of *iter()* is to read lines of a file until a certain line is reached. The following example reads a file until the *readline()* method returns an empty string:

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

len(*s*)

返回对象的长度（元素个数）。实参可以是序列（如 string、bytes、tuple、list 或 range 等）或集合（如 dictionary、set 或 frozen set 等）。

class list([*iterable*])
虽然被称为函数，*list* 实际上是一种可变序列类型，详情请参阅列表 和序列类型—list, tuple, range。

locals()
Update and return a dictionary representing the current local symbol table. Free variables are returned by *locals()* when it is called in function blocks, but not in class blocks.

注解： 不要更改此字典的内容；更改不会影响解释器使用的局部变量或自由变量的值。

map(*function*, *iterable*, ...)
返回一个将 *function* 应用于 *iterable* 中每一项并输出其结果的迭代器。如果传入了额外的 *iterable* 参数，*function* 必须接受相同个数的实参并被应用于从所有可迭代对象中并行获取的项。当有多个可迭代对象时，最短的可迭代对象耗尽则整个迭代就将结束。对于函数的输入已经是参数元组的情况，请参阅*itertools.starmap()*。

max(*iterable*, *[, *key*, *default*])
max(*arg1*, *arg2*, **args*[, *key*])
返回可迭代对象中最大的元素，或者返回两个及以上实参中最大的。

如果只提供了一个位置参数，它必须是非空*iterable*，返回可迭代对象中最大的元素；如果提供了两个及以上的位置参数，则返回最大的位置参数。

有两个可选只能用关键字的实参。*key* 实参指定排序函数用的参数，如传给*list.sort()* 的。*default* 实参是当可迭代对象为空时返回的值。如果可迭代对象为空，并且没有给 *default*，则会触发*ValueError*。

如果有多个最大元素，则此函数将返回第一个找到的。这和其他稳定排序工具如 `sorted(iterable, key=keyfunc, reverse=True)[0]` 和 `heapq.nlargest(1, iterable, key=keyfunc)` 保持一致。

3.4 新版功能: keyword-only 实参 *default* 。

memoryview(*obj*)
返回由给定实参创建的“内存视图”对象。有关详细信息，请参阅内存视图。

min(*iterable*, *[, *key*, *default*])
min(*arg1*, *arg2*, **args*[, *key*])
返回可迭代对象中最小的元素，或者返回两个及以上实参中最小的。

如果只提供了一个位置参数，它必须是*iterable*，返回可迭代对象中最小的元素；如果提供了两个及以上的位置参数，则返回最小的位置参数。

有两个可选只能用关键字的实参。*key* 实参指定排序函数用的参数，如传给*list.sort()* 的。*default* 实参是当可迭代对象为空时返回的值。如果可迭代对象为空，并且没有给 *default*，则会触发*ValueError*。

如果有多个最小元素，则此函数将返回第一个找到的。这和其他稳定排序工具如 `sorted(iterable, key=keyfunc)[0]` 和 `heapq.nsmallest(1, iterable, key=keyfunc)` 保持一致。

3.4 新版功能: keyword-only 实参 *default* 。

next(*iterator*[, *default*])
通过调用 *iterator* 的*__next__()* 方法获取下一个元素。如果迭代器耗尽，则返回给定的 *default*，如果没有默认值则触发*StopIteration*。

class object
返回一个没有特征的新对象。*object* 是所有类的基类。它具有所有 Python 类实例的通用方法。这个函数不接受任何实参。

注解： 由于*object* 没有*__dict__*，因此无法将任意属性赋给*object* 的实例。

oct(*x*)

将一个整数转变为一个前缀为“0o”的八进制字符串。结果是一个合法的 Python 表达式。如果 *x* 不是 Python 的*int* 对象，那它需要定义 `__index__()` 方法返回一个整数。一些例子：

```
>>> oct(8)
'0o10'
>>> oct(-56)
'-0o70'
```

如果要将整数转换为八进制字符串，并可选择有无“0o”前缀，则可以使用如下方法：

```
>>> '%#o' % 10, '%o' % 10
('0o12', '12')
>>> format(10, '#o'), format(10, 'o')
('0o12', '12')
>>> f'{10:#o}', f'{10:o}'
('0o12', '12')
```

另见*format()* 获取更多信息。

open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)

打开 *file* 并返回对应的*file object*。如果该文件不能打开，则触发*OSError*。

file 是一个*path-like object*，表示将要打开的文件的路径（绝对路径或者当前工作目录的相对路径），也可以是要被封装的整数类型文件描述符。（如果是文件描述符，它会随着返回的 I/O 对象关闭而关闭，除非 *closefd* 被设为 `False` 。）

mode 是一个可选字符串，用于指定打开文件的模式。默认值是 `'r'` ，这意味着它以文本模式打开并读取。其他常见模式有：写入 `'w'`（截断已经存在的文件）；排它性创建 `'x'` ；追加写 `'a'`（在一些 Unix 系统上，无论当前的文件指针在什么位置，所有写入都会追加到文件末尾）。在文本模式，如果 *encoding* 没有指定，则根据平台来决定使用的编码：使用 `locale.getpreferredencoding(False)` 来获取本地编码。（要读取和写入原始字节，请使用二进制模式并不要指定 *encoding*。）可用的模式有：

字符	含义
`'r'`	读取（默认）
`'w'`	写入，并先截断文件
`'x'`	排它性创建，如果文件已存在则失败
`'a'`	写入，如果文件存在则在末尾追加
`'b'`	二进制模式
`'t'`	文本模式（默认）
`'+'`	更新磁盘文件（读取并写入）
`'U'`	*universal newlines* mode (deprecated)

默认的模式是 `'r'` （打开并读取文本，同 `'rt'` ）。对于二进制写入，`'w+b'` 模式打开并把文件截断成 0 字节；`'r+b'` 则不会截断。

正如在概述 中提到的，Python 区分二进制和文本 I/O。以二进制模式打开的文件（包括 *mode* 参数中的 `'b'` ）返回的内容为 `bytes`对象，不进行任何解码。在文本模式下（默认情况下，或者在 *mode* 参数中包含 `'t'`）时，文件内容返回为*str*，首先使用指定的 *encoding*（如果给定）或者使用平台默认的的字节编码解码。

注解： Python 不依赖于底层操作系统的文本文件概念; 所有处理都由 Python 本身完成，因此与平台无关。

---

buffering 是一个可选的整数，用于设置缓冲策略。传递 0 以切换缓冲关闭（仅允许在二进制模式下），1 选择行缓冲（仅在文本模式下可用），并且 >1 的整数以指示固定大小的块缓冲区的大小（以字节为单位）。如果没有给出 *buffering* 参数，则默认缓冲策略的工作方式如下:

- 二进制文件以固定大小的块进行缓冲；使用启发式方法选择缓冲区的大小，尝试确定底层设备的“块大小”或使用*io.DEFAULT_BUFFER_SIZE*。在许多系统上，缓冲区的长度通常为 4096 或 8192 字节。
- “交互式”文本文件（*isatty()* 返回 `True` 的文件）使用行缓冲。其他文本文件使用上述策略用于二进制文件。

encoding 是用于解码或编码文件的编码的名称。这应该只在文本模式下使用。默认编码是依赖于平台的（不管*locale.getpreferredencoding()* 返回何值），但可以使用任何 Python 支持的*text encoding* 。有关支持的编码列表，请参阅*codecs* 模块。

errors 是一个可选的字符串参数，用于指定如何处理编码和解码错误 - 这不能在二进制模式下使用。可以使用各种标准错误处理程序（列在*错误处理方案* ），但是使用*codecs.register_error()* 注册的任何错误处理名称也是有效的。标准名称包括:

- 如果存在编码错误，`'strict'` 会引发*ValueError* 异常。默认值 `None` 具有相同的效果。
- `'ignore'` 忽略错误。请注意，忽略编码错误可能会导致数据丢失。
- `'replace'` 会将替换标记（例如 `'?'` ）插入有错误数据的地方。
- `'surrogateescape'` 将表示任何不正确的字节作为 Unicode 专用区中的代码点，范围从 U+DC80 到 U+DCFF。当在写入数据时使用 `surrogateescape` 错误处理程序时，这些私有代码点将被转回到相同的字节中。这对于处理未知编码的文件很有用。
- 只有在写入文件时才支持 `'xmlcharrefreplace'`。编码不支持的字符将替换为相应的 XML 字符引用 `&#nnn;`。
- `'backslashreplace'` 用 Python 的反向转义序列替换格式错误的数据。
- `'namereplace'` （也只在编写时支持）用 `\N{...}` 转义序列替换不支持的字符。

newline 控制*universal newlines* 模式如何生效（它仅适用于文本模式）。它可以是 `None`，`''`，`'\n'`，`'\r'` 和 `'\r\n'`。它的工作原理:

- 从流中读取输入时，如果 *newline* 为 `None`，则启用通用换行模式。输入中的行可以以 `'\n'`，`'\r'` 或 `'\r\n'` 结尾，这些行被翻译成 `'\n'` 在返回呼叫者之前。如果它是 `''`，则启用通用换行模式，但行结尾将返回给调用者未翻译。如果它具有任何其他合法值，则输入行仅由给定字符串终止，并且行结尾将返回给未调用的调用者。
- 将输出写入流时，如果 *newline* 为 `None`，则写入的任何 `'\n'` 字符都将转换为系统默认行分隔符*os.linesep*。如果 *newline* 是 `''` 或 `'\n'`，则不进行翻译。如果 *newline* 是任何其他合法值，则写入的任何 `'\n'` 字符将被转换为给定的字符串。

如果 *closefd* 是 `False` 并且给出了文件描述符而不是文件名，那么当文件关闭时，底层文件描述符将保持打开状态。如果给出文件名则 *closefd* 必须为 `True` （默认值），否则将引发错误。

可以通过传递可调用的 *opener* 来使用自定义开启器。然后通过使用参数（*file*，*flags* ）调用 *opener* 获得文件对象的基础文件描述符。*opener* 必须返回一个打开的文件描述符（使用*os.open* as *opener* 时与传递 `None` 的效果相同）。

新创建的文件是*不可继承的*。

下面的示例使用*os.open()* 函数的*dir_fd* 的形参，从给定的目录中用相对路径打开文件:

```
>>> import os
>>> dir_fd = os.open('somedir', os.O_RDONLY)
>>> def opener(path, flags):
...     return os.open(path, flags, dir_fd=dir_fd)
```

(下页继续)

(续上页)

```
...
>>> with open('spamspam.txt', 'w', opener=opener) as f:
...     print('This will be written to somedir/spamspam.txt', file=f)
...
>>> os.close(dir_fd)  # don't leak a file descriptor
```

open() 函数所返回的*file object* 类型取决于所用模式。当使用*open()* 以文本模式 (`'w'`, `'r'`, `'wt'`, `'rt'` 等) 打开文件时，它将返回*io.TextIOBase* (特别是*io.TextIOWrapper*) 的一个子类。当使用缓冲以二进制模式打开文件时，返回的类是*io.BufferedIOBase* 的一个子类。具体的类会有多种：在只读的二进制模式下，它将返回*io.BufferedReader*；在写入二进制和追加二进制模式下，它将返回*io.BufferedWriter*，而在读/写模式下，它将返回*io.BufferedRandom*。当禁用缓冲时，则会返回原始流，即*io.RawIOBase* 的一个子类*io.FileIO*。

另请参阅文件操作模块，例如*fileinput*、*io* (声明了*open()*)、*os*、*os.path*、*tempfile* 和*shutil*。

在 3.3 版更改:

- 增加了 *opener* 形参。
- 增加了 `'x'` 模式。
- 过去触发的*IOError*，现在是*OSError* 的别名。
- 如果文件已存在但使用了排它性创建模式（`'x'`），现在会触发*FileExistsError*。

在 3.4 版更改:

- 文件现在禁止继承。

Deprecated since version 3.4, will be removed in version 4.0: `'U'` 模式。

在 3.5 版更改:

- 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。
- 增加了 `'namereplace'` 错误处理接口。

在 3.6 版更改:

- 增加对实现了*os.PathLike* 对象的支持。
- 在 Windows 上，打开一个控制台缓冲区将返回*io.RawIOBase* 的子类，而不是*io.FileIO*。

ord(*c*)

对表示单个 Unicode 字符的字符串，返回代表它 Unicode 码点的整数。例如 `ord('a')` 返回整数 `97`，`ord('€')` （欧元符号）返回 `8364` 。这是*chr()* 的逆函数。

pow(*x*, *y*[, *z*])

返回 *x* 的 *y* 次幂；如果 *z* 存在，则对 *z* 取余（比直接 `pow(x, y) % z` 计算更高效）。两个参数形式的 `pow(x, y)` 等价于幂运算符：`x**y`。

参数必须为数值类型。对于混用的操作数类型，则适用二元算术运算符的类型强制转换规则。对于*int* 操作数，结果具有与操作数相同的类型（转换后），除非第二个参数为负值；在这种情况下，所有参数将被转换为浮点数并输出浮点数结果。例如，`10**2` 返回 `100`，但 `10**-2` 返回 `0.01`。如果第二个参数为负值，则第三个参数必须省略。如果存在 *z*，则 *x* 和 *y* 必须为整数类型，且 *y* 必须为非负数。

print(**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*, *flush=False*)

将 *objects* 打印到 *file* 指定的文本流，以 *sep* 分隔并在末尾加上 *end*。*sep*, *end*, *file* 和 *flush* 如果存在，它们必须以关键字参数的形式给出。

所有非关键字参数都会被转换为字符串，就像是执行了*str()* 一样，并会被写入到流，以 *sep* 且在末尾加上 *end*。*sep* 和 *end* 都必须为字符串；它们也可以为 None，这意味着使用默认值。如果没有给出 *objects*，则*print()* 将只写入 *end*。

file 参数必须是一个具有 write(string) 方法的对象；如果参数不存在或为 None，则将使用*sys.stdout*。由于要打印的参数会被转换为文本字符串，因此*print()* 不能用于二进制模式的文件对象。对于这些对象，应改用 file.write(...)。

输出是否被缓存通常决定于 *file*，但如果 *flush* 关键字参数为真值，流会被强制刷新。

在 3.3 版更改: 增加了 *flush* 关键字参数。

class property(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)

返回 property 属性。

fget 是获取属性值的函数。*fset* 是用于设置属性值的函数。*fdel* 是用于删除属性值的函数。并且 *doc* 为属性对象创建文档字符串。

一个典型的用法是定义一个托管属性 x:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

如果 *c* 是 *C* 的实例，c.x 将调用 getter，c.x = value 将调用 setter，del c.x 将调用 deleter。

如果给出，*doc* 将成为该 property 属性的文档字符串。否则该 property 将拷贝 *fget* 的文档字符串（如果存在）。这令使用*property()* 作为*decorator* 来创建只读的特征属性可以很容易地实现:

```
class Parrot:
    def __init__(self):
        self._voltage = 100000

    @property
    def voltage(self):
        """Get the current voltage."""
        return self._voltage
```

以上 @property 装饰器会将 voltage() 方法转化为一个具有相同名称的只读属性的“getter”，并将 *voltage* 的文档字符串设置为“Get the current voltage.”

特征属性对象具有 getter, setter 以及 deleter 方法，它们可用作装饰器来创建该特征属性的副本，并将相应的访问函数设为所装饰的函数。这最好是用一个例子来解释:

```
class C:
    def __init__(self):
        self._x = None

    @property
```

(下页继续)

(续上页)

```
    def x(self):
        """I'm the 'x' property."""
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

上述代码与第一个例子完全等价。注意一定要给附加函数与原始的特征属性相同的名称 (在本例中为 `x`。)

返回的特征属性对象同样具有与构造器参数相对应的属性 `fget`, `fset` 和 `fdel`。

在 3.5 版更改: 特征属性对象的文档字符串现在是可写的。

range(*stop*)
range(*start*, *stop*[, *step*])
虽然被称为函数，但*range* 实际上是一个不可变的序列类型，参见在*range 对象* 与*序列类型—list, tuple, range* 中的文档说明。

repr(*object*)
返回包含一个对象的可打印表示形式的字符串。对于许多类型来说，该函数会尝试返回的字符串将会与该对象被传递给*eval()* 时所生成的对象具有相同的值，在其他情况下表示形式会是一个括在尖括号中的字符串，其中包含对象类型的名称与通常包括对象名称和地址的附加信息。类可以通过定义 `__repr__()` 方法来控制此函数为它的实例所返回的内容。

reversed(*seq*)
返回一个反向的*iterator*。*seq* 必须是一个具有 `__reversed__()` 方法的对象或者是支持该序列协议 (具有从 `0` 开始的整数类型参数的 `__len__()` 方法和 `__getitem__()` 方法)。

round(*number*[, *ndigits*])
返回 *number* 舍入到小数点后 *ndigits* 位精度的值。如果 *ndigits* 被省略或为 `None`，则返回最接近输入值的整数。

对于支持*round()* 的内置类型，值会被舍入到最接近的 10 的负 *ndigits* 次幂的倍数；如果与两个倍数的距离相等，则选择偶数 (因此，`round(0.5)` 和 `round(-0.5)` 均为 `0` 而 `round(1.5)` 为 `2`)。任何整数值都可作为有效的 *ndigits* (正数、零或负数)。如果 *ndigits* 被省略或为 `None` 则返回值将为整数。否则返回值与 *number* 的类型相同。

对于一般的 Python 对象 `number`, `round` 将委托给 `number.__round__`。

注解: 对浮点数执行*round()* 的行为可能会令人惊讶：例如，`round(2.675, 2)` 将给出 `2.67` 而不是期望的 `2.68`。这不算是程序错误：这一结果是由于大多数十进制小数实际上都不能以浮点数精确地表示。请参阅 tut-fp-issues 了解更多信息。

---

class set([*iterable*])
返回一个新的*set* 对象，可以选择带有从 *iterable* 获取的元素。`set` 是一个内置类型。请查看*set* 和*集合类型—set, frozenset* 获取关于这个类的文档。

有关其他容器请参看内置的*frozenset*, *list*, *tuple* 和*dict* 类，以及*collections* 模块。

setattr(*object*, *name*, *value*)
此函数与*getattr()* 两相对应。其参数为一个对象、一个字符串和一个任意值。字符串指定一个现有属

性或者新增属性。函数会将值赋给该属性，只要对象允许这种操作。例如，`setattr(x, 'foobar', 123)` 等价于 `x.foobar = 123`。

class slice(*stop*)
class slice(*start*, *stop*[, *step*])
返回一个表示由 `range(start, stop, step)` 所指定索引集的*slice* 对象。其中 *start* 和 *step* 参数默认为 `None`。切片对象具有仅会返回对应参数值（或其默认值）的只读数据属性 `start`, `stop` 和 `step`。它们没有其他的显式功能；不过它们会被 NumPy 以及其他第三方扩展所使用。切片对象也会在使用扩展索引语法时被生成。例如: `a[start:stop:step]` 或 `a[start:stop, i]`。请参阅*itertools.islice()* 了解返回迭代器的一种替代版本。

sorted(*iterable*, *, *key=None*, *reverse=False*)
根据 *iterable* 中的项返回一个新的已排序列表。

具有两个可选参数，它们都必须指定为关键字参数。

key 指定带有单个参数的函数，用于从 *iterable* 的每个元素中提取用于比较的键 (例如 `key=str.lower`)。默认值为 `None` (直接比较元素)。

reverse 为一个布尔值。如果设为 `True`，则每个列表元素将按反向顺序比较进行排序。

使用*functools.cmp_to_key()* 可将老式的 *cmp* 函数转换为 *key* 函数。

内置的*sorted()* 确保是稳定的。如果一个排序确保不会改变比较结果相等的元素的相对顺序就称其为稳定的—这有利于进行多重排序（例如先按部门、再按薪级排序）。

有关排序示例和简要排序教程，请参阅 sortinghowto 。

@staticmethod
将方法转换为静态方法。

静态方法不会接收隐式的第一个参数。要声明一个静态方法，请使用此语法

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

The `@staticmethod` form is a function *decorator* –see the description of function definitions in function for details.

It can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class.

Python 中的静态方法与 Java 或 C++ 中的静态方法类似。另请参阅*classmethod()* ，用于创建备用类构造函数的变体。

像所有装饰器一样，也可以像常规函数一样调用 `staticmethod` ，并对其结果执行某些操作。比如某些情况下需要从类主体引用函数并且您希望避免自动转换为实例方法。对于这些情况，请使用此语法:

```
class C:
    builtin_open = staticmethod(open)
```

For more information on static methods, consult the documentation on the standard type hierarchy in types.

class str(*object=''*)
class str(*object=b''*, *encoding='utf-8'*, *errors='strict'*)
返回一个*str* 版本的 *object* 。有关详细信息，请参阅*str()* 。

`str` 是内置字符串*class* 。更多关于字符串的信息查看文本序列类型—*str*。

sum(*iterable*[, *start*])
从 *start* 开始自左向右对 *iterable* 中的项求和并返回总计值。*start* 默认为 `0`。*iterable* 的项通常为数字，开始值则不允许为字符串。

对某些用例来说，存在*sum()* 的更好替代。拼接字符串序列的更好更快方式是调用 ''.join(sequence)。要以扩展精度对浮点值求和，请参阅*math.fsum()*。要拼接一系列可迭代对象，请考虑使用*itertools.chain()*。

super([*type*[, *object-or-type*]])

返回一个代理对象，它会将方法调用委托给 *type* 指定的父类或兄弟类。这对于访问已在类中被重载的继承方法很有用。搜索顺序与*getattr()* 所使用的相同，只是 *type* 指定的类型本身会被跳过。

type 的*__mro__* 属性列出了*getattr()* 和*super()* 所使用的方法解析顺序。该属性是动态的，可以在继承层级结构更新的时候任意改变。

如果省略第二个参数，则返回的超类对象是未绑定的。如果第二个参数为一个对象，则 isinstance(obj, type) 必须为真值。如果第二个参数为一个类型，则 issubclass(type2, type) 必须为真值（这适用于类方法）。

super 有两个典型用例。在具有单继承的类层级结构中，*super* 可用来引用父类而不必显式地指定它们的名称，从而令代码更易维护。这种用法与其他编程语言中 *super* 的用法非常相似。

第二个用例是在动态执行环境中支持协作多重继承。此用例为 Python 所独有，在静态编译语言或仅支持单继承的语言中是不存在的。这使得实现“菱形图”成为可能，在这时会有多个基类实现相同的方法。好的设计强制要求这种方法在每个情况下具有相同的调用签名（因为调用顺序是在运行时确定的，也因为该顺序要适应类层级结构的更改，还因为该顺序可能包含在运行时之前未知的兄弟类）。

对于以上两个用例，典型的超类调用看起来是这样的:

```
class C(B):
    def method(self, arg):
        super().method(arg)    # This does the same thing as:
                               # super(C, self).method(arg)
```

请注意*super()* 是作为显式加点属性查找的绑定过程的一部分来实现的，例如 super().__getitem__(name)。它做到这一点是通过实现自己的 __getattribute__() 方法，这样就能以可预测的顺序搜索类，并且支持协作多重继承。对应地，*super()* 在像 super()[name] 这样使用语句或操作符进行隐式查找时则未被定义。

还要注意的是，除了零个参数的形式以外，*super()* 并不限于在方法内部使用。两个参数的形式明确指定参数并进行相应的引用。零个参数的形式仅适用于类定义内部，因为编译器需要填入必要的细节以正确地检索到被定义的类，还需要让普通方法访问当前实例。

对于有关如何使用*super()* 来如何设计协作类的实用建议，请参阅 使用 super() 的指南。

tuple([*iterable*])

虽然被称为函数，但*tuple* 实际上是一个不可变的序列类型，参见在*元组* 与*序列类型—list, tuple, range* 中的文档说明。

class type(*object*)

class type(*name*, *bases*, *dict*)

传入一个参数时，返回 *object* 的类型。返回值是一个 type 对象，通常与*object.__class__* 所返回的对象相同。

推荐使用*isinstance()* 内置函数来检测对象的类型，因为它会考虑子类的情况。

传入三个参数时，返回一个新的 type 对象。这在本质上是 class 语句的一种动态形式。*name* 字符串即类名并且会成为*__name__* 属性；*bases* 元组列出基类并且会成为*__bases__* 属性；而 *dict* 字典为包含类主体定义的命名空间并且会被复制到一个标准字典成为*__dict__* 属性。例如，下面两条语句会创建相同的*type* 对象:

```
>>> class X:
...     a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

另请参阅类型对象。

在 3.6 版更改: *type* 的子类如果未重载 type.__new__，将不再能使用一个参数的形式来获取对象的类型。

vars([*object*])
返回模块、类、实例或任何其它具有*__dict__* 属性的对象的*__dict__* 属性。

模块和实例这样的对象具有可更新的*__dict__* 属性；但是，其它对象的*__dict__* 属性可能会设为限制写入（例如，类会使用*types.MappingProxyType* 来防止直接更新字典）。

不带参数时，*vars()* 的行为类似*locals()*。请注意，locals 字典仅对于读取起作用，因为对 locals 字典的更新会被忽略。

zip(**iterables*)
创建一个聚合了来自每个可迭代对象中的元素的迭代器。

返回一个元组的迭代器，其中的第 *i* 个元组包含来自每个参数序列或可迭代对象的第 *i* 个元素。当所输入可迭代对象中最短的一个被耗尽时，迭代器将停止迭代。当只有一个可迭代对象参数时，它将返回一个单元组的迭代器。不带参数时，它将返回一个空迭代器。相当于:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

函数会保证可迭代对象按从左至右的顺序被求值。使得可以通过 zip(*[iter(s)]*n) 这样的惯用形式将一系列数据聚类为长度为 n 的分组。这将重复 同样的迭代器 n 次，以便每个输出的元组具有第 n 次调用该迭代器的结果。它的作用效果就是将输入拆分为长度为 n 的数据块。

当你不用关心较长可迭代对象末尾不匹配的值时，则*zip()* 只须使用长度不相等的输入即可。如果那些值很重要，则应改用*itertools.zip_longest()*。

zip() 与 * 运算符相结合可以用来拆解一个列表:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zip(x, y))
>>> x == list(x2) and y == list(y2)
True
```

__import__(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)

注解: 与*importlib.import_module()* 不同，这是一个日常 Python 编程中不需要用到的高级函数。

This function is invoked by the `import` statement. It can be replaced (by importing the *builtins* module and assigning to `builtins.__import__`) in order to change semantics of the `import` statement, but doing so is **strongly** discouraged as it is usually simpler to use import hooks (see **PEP 302**) to attain the same goals and does not cause issues with code which assumes the default import implementation is in use. Direct use of *__import__()* is also discouraged in favor of *importlib.import_module()*.

该函数会导入 *name* 模块，有可能使用给定的 *globals* 和 *locals* 来确定如何在包的上下文中解读名称。*fromlist* 给出了应该从由 *name* 指定的模块导入对象或子模块的名称。标准实现完全不使用其 *locals* 参数，而仅使用 *globals* 参数来确定 `import` 语句的包上下文。

level 指定是使用绝对还是相对导入。`0` (默认值) 意味着仅执行绝对导入。*level* 为正数值表示相对于模块调用*__import__()* 的目录，将要搜索的父目录层数 (详情参见 **PEP 328**)。

当 *name* 变量的形式为 `package.module` 时，通常将会返回最高层级的包（第一个点号之前的名称），而 不是以 *name* 命名的模块。但是，当给出了非空的 *fromlist* 参数时，则将返回以 *name* 命名的模块。

例如，语句 `import spam` 的结果将为与以下代码作用相同的字节码:

```
spam = __import__('spam', globals(), locals(), [], 0)
```

语句 `import spam.ham` 的结果将为以下调用:

```
spam = __import__('spam.ham', globals(), locals(), [], 0)
```

请注意在这里*__import__()* 是如何返回顶层模块的，因为这是通过 `import` 语句被绑定到特定名称的对象。

另一方面，语句 `from spam.ham import eggs, sausage as saus` 的结果将为

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], 0)
eggs = _temp.eggs
saus = _temp.sausage
```

在这里，`spam.ham` 模块会由*__import__()* 返回。要导入的对象将从此对象中提取并赋值给它们对应的名称。

如果您只想按名称导入模块（可能在包中），请使用*importlib.import_module()*

在 3.3 版更改: Negative values for *level* are no longer supported (which also changes the default value to 0).

备注

CHAPTER 3

内置常量

有少数的常量存在于内置命名空间中。它们是：

False
bool 类型的假值。给 `False` 赋值是非法的并会引发*SyntaxError*。

True
bool 类型的真值。给 `True` 赋值是非法的并会引发*SyntaxError*。

None
`NoneType` 类型的唯一值。`None` 经常用于表示缺少值，当因为默认参数未传递给函数时。给 `None` 赋值是非法的并会引发*SyntaxError*。

NotImplemented
二进制特殊方法应返回的特殊值（例如，`__eq__()`、`__lt__()`、`__add __()`、`__rsub__()` 等）表示操作没有针对其他类型实现；为了相同的目的，可以通过就地二进制特殊方法（例如，`__imul __()`、`__ rightnd__()` 等）返回。它的逻辑值为真。

注解： 当二进制（或就地）方法返回 "NotImplemented"时，解释器将尝试对另一种类型（或其他一些回滚操作，取决于运算符）的反射操作。如果所有尝试都返回 "NotImplemented"，则解释器将引发适当的异常。错误返回的 "NotImplemented"将导致误导性错误消息或返回到 Python 代码中的 "NotImplemented"值。

参见实现算术运算 为例。

注解： `NotImplementedError` 和 `NotImplemented` 不可互换，即使它们有相似的名称和用途。有关何时使用它的详细信息，请参阅*NotImplementedError*。

Ellipsis
与省略号文字字面 "`...`" 相同。特殊值主要与用户定义的容器数据类型的扩展切片语法结合使用。

__debug__
如果 Python 没有以 `-O` 选项启动，则此常量为真值。另请参见 `assert` 语句。

注解: 变量名*None*，*False*，*True* 和 __debug__ 无法重新赋值（赋值给它们，即使是属性名，将引发*SyntaxError*），所以它们可以被认为是“真正的”常数。

3.1 由 site 模块添加的常量

site 模块（在启动期间自动导入，除非给出 -S 命令行选项）将几个常量添加到内置命名空间。它们对交互式解释器 shell 很有用，并且不应在程序中使用。

quit(*code=None*)
exit(*code=None*)
当打印此对象时，会打印出一条消息，例如“Use quit() or Ctrl-D (i.e. EOF) to exit”，当调用此对象时，将使用指定的退出代码来引发*SystemExit*。

copyright
credits
打印或调用的对象分别打印版权或作者的文本。

license
当打印此对象时，会打印出一条消息“Type license() to see the full license text”，当调用此对象时，将以分页形式显示完整的许可证文本（每次显示一屏）。

CHAPTER 4

内置类型

以下部分描述了解释器中内置的标准类型。

主要内置类型有数字、序列、映射、类、实例和异常。

有些多项集类是可变的。它们用于添加、移除或重排其成员的方法将原地执行，并不返回特定的项，绝对不会返回多项集实例自身而是返回 None。

有些操作受多种对象类型的支持；特别地，实际上所有对象都可以被比较、检测逻辑值，以及转换为字符串（使用*repr()* 函数或略有差异的*str()* 函数）。后一个函数是在对象由*print()* 函数输出时被隐式地调用的。

4.1 逻辑值检测

任何对象都可以进行逻辑值的检测，以便在 if 或 while 作为条件或是作为下文所述布尔运算的操作数来使用。

一个对象在默认情况下均被视为真值，除非当该对象被调用时其所属类定义了 __bool__() 方法且返回 False 或是定义了 __len__() 方法且返回零。[1] 下面基本完整地列出了会被视为假值的内置对象:

- 被定义为假值的常量: None 和 False。
- 任何数值类型的零: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
- 空的序列和多项集: '', (), [], {}, set(), range(0)

产生布尔值结果的运算和内置函数总是返回 0 或 False 作为假值，1 或 True 作为真值，除非另行说明。（重要例外：布尔运算 or 和 and 总是返回其中一个操作数。）

[1] 有关这些特殊方法的额外信息可参看 Python 参考指南 (customization)。

4.2 Boolean Operations —and, or, not

这些属于布尔运算，按优先级升序排列:

运算	结果:	注释
`x or y`	if *x* is false, then *y*, else *x*	(1)
`x and y`	if *x* is false, then *x*, else *y*	(2)
`not x`	if *x* is false, then `True`, else `False`	(3)

注释:

(1) 这是个短路运算符，因此只有在第一个参数为假值时才会对第二个参数求值。

(2) 这是个短路运算符，因此只有在第一个参数为真值时才会对第二个参数求值。

(3) `not` 的优先级比非布尔运算符低，因此 `not a == b` 会被解读为 `not (a == b)` 而 `a == not b` 会引发语法错误。

4.3 比较运算

在 Python 中有八种比较运算符。它们的优先级相同（比布尔运算的优先级高）。比较运算可以任意串连；例如，`x < y <= z` 等价于 `x < y and y <= z`，前者的不同之处在于 *y* 只被求值一次（但在两种情况下当 `x < y` 结果为假值时 *z* 都不会被求值）。

此表格汇总了比较运算:

运算	含义
`<`	严格小于
`<=`	小于或等于
`>`	严格大于
`>=`	大于或等于
`==`	等于
`!=`	不等于
`is`	对象标识
`is not`	否定的对象标识

除了不同数字类型以外，不同类型的对象比较时绝对不会相等。而且，某些类型（例如函数对象）仅支持简化比较形式，即任何两个该种类型的对象必定不相等。<, <=, > 和 >= 运算符在以下情况中将引发 *TypeError* 异常：当比较复数与另一个内置数字类型时，当两个对象具有无法被比较的不同类型时，或在未定义次序的其他情况时。

具有不同标识的类的实例比较结果通常为不相等，除非类定义了 `__eq__()` 方法。

一个类实例不能与相同类或的其他实例或其他类型的对象进行排序，除非该类定义了足够多的方法，包括 `__lt__()`, `__le__()`, `__gt__()` 以及 `__ge__()` (而如果你想实现常规意义上的比较操作，通常只要有 `__lt__()` 和 `__eq__()` 就可以了)。

`is` 和 `is not` 运算符无法自定义；并且它们可以被应用于任意两个对象而不会引发异常。

还有两种具有相同语法优先级的运算 `in` 和 `not in`，它们被*iterable* 或实现了 `__contains__()` 方法的类型所支持。

4.4 数字类型—int, float, complex

存在三种不同的数字类型: *整数*, *浮点数*和 *复数*。此外，布尔值属于整数的子类型。整数具有无限的精度。浮点数通常使用 C 中的 double 来实现；有关你的程序运行所在机器上浮点数的精度和内部表示法可在*sys.float_info* 中查看。复数包含实部和虚部，分别以一个浮点数表示。要从一个复数 *z* 中提取这两个部分，可使用 z.real 和 z.imag。(标准库包含附加的数字类型，如表示有理数的*fractions* 以及以用户定制精度表示浮点数的*decimal*。)

数字是由数字字面值或内置函数与运算符的结果来创建的。不带修饰的整数字面值（包括十六进制、八进制和二进制数）会生成整数。包含小数点或幂运算符的数字字面值会生成浮点数。在数字字面值末尾加上 'j' 或 'J' 会生成虚数（实部为零的复数），你可以将其与整数或浮点数相加来得到具有实部和虚部的复数。

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where integer is narrower than floating point, which is narrower than complex. Comparisons between numbers of mixed type use the same rule.[2] The constructors *int()*, *float()*, and *complex()* can be used to produce numbers of a specific type.

All numeric types (except complex) support the following operations, sorted by ascending priority (all numeric operations have a higher priority than comparison operations):

运算	结果:	注释	完整文档
x + y	*x* 和 *y* 的和		
x - y	*x* 和 *y* 的差		
x * y	*x* 和 *y* 的乘积		
x / y	*x* 和 *y* 的商		
x // y	*x* 和 *y* 的商数	(1)	
x % y	x / y 的余数	(2)	
-x	*x* 取反		
+x	*x* 不变		
abs(x)	*x* 的绝对值或大小		*abs()*
int(x)	将 *x* 转换为整数	(3)(6)	*int()*
float(x)	将 *x* 转换为浮点数	(4)(6)	*float()*
complex(re, im)	一个带有实部 *re* 和虚部 *im* 的复数。*im* 默认为 0。	(6)	*complex()*
c.conjugate()	复数 *c* 的共轭		
divmod(x, y)	(x // y, x % y)	(2)	*divmod()*
pow(x, y)	*x* 的 *y* 次幂	(5)	*pow()*
x ** y	*x* 的 *y* 次幂	(5)	

注释:

(1) 也称为整数除法。结果值是一个整数，但结果的类型不一定是 int。运算结果总是向负无穷的方向舍入: 1//2 为 0, (-1)//2 为 -1, 1//(-2) 为 -1 而 (-1)//(-2) 为 0。

(2) 不可用于复数。而应在适当条件下使用*abs()* 转换为浮点数。

(3) 从浮点数转换为整数会被舍入或是像在 C 语言中一样被截断；请参阅*math.floor()* 和*math.ceil()* 函数查看转换的完整定义。

(4) float 也接受字符串 "nan" 和附带可选前缀 "+" 或 "-" 的 "inf" 分别表示非数字 (NaN) 以及正或负无穷。

(5) Python 将 pow(0, 0) 和 0 ** 0 定义为 1，这是编程语言的普遍做法。

(6) 接受的数字字面值包括数码 0 到 9 或任何等效的 Unicode 字符（具有 Nd 特征属性的代码点）。

See http://www.unicode.org/Public/9.0.0/ucd/extracted/DerivedNumericType.txt for a complete list of code points with the Nd property.

[2] 作为结果，列表 [1, 2] 与 [1.0, 2.0] 是相等的，元组的情况也类似。

所有*numbers.Real* 类型 (*int* 和*float*) 还包括下列运算:

运算	结果:
math.trunc(x)	*x* 截断为*Integral*
round(x[, n])	*x* 舍入到 *n* 位小数，半数值会舍入到偶数。如果省略 *n*，则默认为 0。
math.floor(x)	<= *x* 的最大*Integral*
math.ceil(x)	>= *x* 的最小*Integral*

有关更多的数字运算请参阅*math* 和*cmath* 模块。

4.4.1 整数类型的按位运算

按位运算只对整数有意义。计算按位运算的结果，就相当于使用无穷多个二进制符号位对二的补码执行操作。

二进制按位运算的优先级全都低于数字运算，但又高于比较运算；一元运算 ~ 具有与其他一元算术运算 (+ and −) 相同的优先级。

此表格是以优先级升序排序的按位运算列表:

运算	结果:	注释
`x | y`	*x* 和 *y* 按位 或	(4)
`x ^ y`	*x* 和 *y* 按位 异或	(4)
`x & y`	*x* 和 *y* 按位 与	(4)
`x << n`	*x* 左移 *n* 位	(1)(2)
`x >> n`	*x* 右移 *n* 位	(1)(3)
`~x`	*x* 逐位取反	

注释:

(1) 负的移位数是非法的，会导致引发*ValueError*。

(2) A left shift by *n* bits is equivalent to multiplication by `pow(2, n)` without overflow check.

(3) A right shift by *n* bits is equivalent to division by `pow(2, n)` without overflow check.

(4) 使用带有至少一个额外符号扩展位的有限个二进制补码表示（有效位宽度为 `1 + max(x.bit_length(), y.bit_length())` 或以上）执行这些计算就足以获得相当于有无数个符号位时的同样结果。

4.4.2 整数类型的附加方法

int 类型实现了*numbers.Integral* *abstract base class*。此外，它还提供了其他几个方法:

`int.`**`bit_length`**`()`

返回以二进制表示一个整数所需要的位数，不包括符号位和前面的零:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

更准确地说，如果 x 非零，则 x.bit_length() 是使得 2**(k-1) <= abs(x) < 2**k 的唯一正整数 k。同样地，当 abs(x) 小到足以具有正确的舍入对数时，则 k = 1 + int(log(abs(x), 2))。如果 x 为零，则 x.bit_length() 返回 0。

等价于:

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) --> '-0b100101'
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

3.1 新版功能.

int.**to_bytes**(*length*, *byteorder*, *, *signed=False*)

返回表示一个整数的字节数组。

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

整数会使用 *length* 个字节来表示。如果整数不能用给定的字节数来表示则会引发*OverflowError*。

byteorder 参数确定用于表示整数的字节顺序。如果 *byteorder* 为 "big"，则最高位字节放在字节数组的开头。如果 *byteorder* 为 "little"，则最高位字节放在字节数组的末尾。要请求主机系统上的原生字节顺序，请使用*sys.byteorder* 作为字节顺序值。

signed 参数确定是否使用二的补码来表示整数。如果 *signed* 为 False 并且给出的是负整数，则会引发*OverflowError*。*signed* 的默认值为 False。

3.2 新版功能.

classmethod int.**from_bytes**(*bytes*, *byteorder*, *, *signed=False*)

返回由给定字节数组所表示的整数。

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

bytes 参数必须为一个*bytes-like object* 或是生成字节的可迭代对象。

byteorder 参数确定用于表示整数的字节顺序。如果 *byteorder* 为 "big"，则最高位字节放在字节数组的开头。如果 *byteorder* 为 "little"，则最高位字节放在字节数组的末尾。要请求主机系统上的原生字节顺序，请使用*sys.byteorder* 作为字节顺序值。

signed 参数指明是否使用二的补码来表示整数。

3.2 新版功能.

4.4.3 浮点类型的附加方法

float 类型实现了*numbers.Real abstract base class*。float 还具有以下附加方法。

`float.`**`as_integer_ratio`**`()`
返回一对整数，其比率正好等于原浮点数并且分母为正数。无穷大会引发*OverflowError* 而 NaN 则会引发*ValueError*。

`float.`**`is_integer`**`()`
如果 float 实例可用有限位整数表示则返回 `True`，否则返回 `False`:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

两个方法均支持与十六进制数字符串之间的转换。由于 Python 浮点数在内部存储为二进制数，因此浮点数与十进制数字符串之间的转换往往会导致微小的舍入错误。而十六进制数字符串却允许精确地表示和描述浮点数。这在进行调试和数值工作时非常有用。

`float.`**`hex`**`()`
以十六进制字符串的形式返回一个浮点数表示。对于有限浮点数，这种表示法将总是包含前导的 `0x` 和尾随的 `p` 加指数。

`classmethod` `float.`**`fromhex`**`(`*s*`)`
返回以十六进制字符串 *s* 表示的浮点数的类方法。字符串 *s* 可以带有前导和尾随的空格。

请注意*float.hex()* 是实例方法，而*float.fromhex()* 是类方法。

十六进制字符串采用的形式为:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

可选的 `sign` 可以是 `+` 或 `-`，`integer` 和 `fraction` 是十六进制数码组成的字符串，`exponent` 是带有可选前导符的十进制整数。大小写没有影响，在 integer 或 fraction 中必须至少有一个十六进制数码。此语法类似于 C99 标准的 6.4.4.2 小节中所描述的语法，也是 Java 1.5 以上所使用的语法。特别地，*float.hex()* 的输出可以用作 C 或 Java 代码中的十六进制浮点数字面值，而由 C 的 `%a` 格式字符或 Java 的 `Double.toHexString` 所生成的十六进制数字符串由为*float.fromhex()* 所接受。

请注意 exponent 是十进制数而非十六进制数，它给出要与系数相乘的 2 的幂次。例如，十六进制数字符串 `0x3.a7p10` 表示浮点数 `(3 + 10./16 + 7./16**2) * 2.0**10` 即 `3740.0`:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

对 `3740.0` 应用反向转换会得到另一个代表相同数值的十六进制数字符串:

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

4.4.4 数字类型的哈希运算

对于可能为不同类型的数字 x 和 y，要求 x == y 时必定 hash(x) == hash(y) (详情参见 __hash__() 方法的文档)。为了便于在各种数字类型 (包括 *int*, *float*, *decimal.Decimal* 和 *fractions.Fraction*) 上实现并保证效率，Python 对数字类型的哈希运算是基于为任意有理数定义统一的数学函数，因此该运算对 *int* 和 *fractions.Fraction* 的全部实例，以及 *float* 和 *decimal.Decimal* 的全部有限实例均可用。从本质上说，此函数是通过以一个固定质数 P 进行 P 降模给出的。P 的值在 Python 中可以 *sys.hash_info* 的 modulus 属性的形式被访问。

CPython implementation detail: 目前所用的质数设定，在 C long 为 32 位的机器上 P = 2**31 - 1 而在 C long 为 64 位的机器上 P = 2**61 - 1。

详细规则如下所述:

- 如果 x = m / n 是一个非负的有理数且 n 不可被 P 整除，则定义 hash(x) 为 m * invmod(n, P) % P，其中 invmod(n, P) 是对 n 模 P 取反。
- 如果 x = m / n 是一个非负的有理数且 n 可被 P 整除（但 m 不能）则 n 不能对 P 降模，以上规则不适用；在此情况下则定义 hash(x) 为常数值 sys.hash_info.inf。
- 如果 x = m / n 是一个负的有理数则定义 hash(x) 为 -hash(-x)。如果结果哈希值为 -1 则将其替换为 -2。
- 特定值 sys.hash_info.inf, -sys.hash_info.inf 和 sys.hash_info.nan 被用作正无穷、负无穷和空值（所分别对应的）哈希值。(所有可哈希的空值都具有相同的哈希值。)
- 对于一个 *complex* 值 z，会通过计算 hash(z.real) + sys.hash_info.imag * hash(z.imag) 将实部和虚部的哈希值结合起来，并进行降模 2**sys.hash_info.width 以使其处于 range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1)) 范围之内。同样地，如果结果为 -1 则将其替换为 -2。

为了阐明上述规则，这里有一些等价于内置哈希算法的 Python 代码示例，可用于计算有理数、*float* 或 *complex* 的哈希值:

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P.  (Unnecessary if m and n already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_value = -hash_value
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

(下页继续)

(续上页)

```
def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return sys.hash_info.nan
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.imag)
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

4.5 迭代器类型

Python 支持在容器中进行迭代的概念。这是通过使用两个单独方法来实现的；它们被用于允许用户自定义类对迭代的支持。将在下文中详细描述的序列总是支持迭代方法。

容器对象要提供迭代支持，必须定义一个方法:

container.**__iter__**()
: 返回一个迭代器对象。该对象需要支持下文所述的迭代器协议。如果容器支持不同的迭代类型，则可以提供额外的方法来专门地请求不同迭代类型的迭代器。(支持多种迭代形式的对象的例子有同时支持广度优先和深度优先遍历的树结构。) 此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iter` 槽位。

迭代器对象自身需要支持以下两个方法，它们共同组成了 *迭代器协议*:

iterator.**__iter__**()
: 返回迭代器对象本身。这是同时允许容器和迭代器配合 `for` 和 `in` 语句使用所必须的。此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iter` 槽位。

iterator.**__next__**()
: 从容器中返回下一项。如果已经没有项可返回，则会引发 *StopIteration* 异常。此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iternext` 槽位。

Python 定义了几种迭代器对象以支持对一般和特定序列类型、字典和其他更特别的形式进行迭代。除了迭代器协议的实现，特定类型的其他性质对迭代操作来说都不重要。

一旦迭代器的 *__next__()* 方法引发了 *StopIteration*，它必须一直对后续调用引发同样的异常。不遵循此行为特性的实现将无法正常使用。

4.5.1 生成器类型

Python 的*generator* 提供了一种实现迭代器协议的便捷方式。如果容器对象 `__iter__()` 方法被实现为一个生成器，它将自动返回一个迭代器对象（从技术上说是一个生成器对象），该对象提供 `__iter__()` 和 `__next__()` 方法。有关生成器的更多信息可以参阅 yield 表达式的文档。

4.6 序列类型—list, tuple, range

有三种基本序列类型：list, tuple 和 range 对象。为处理二进制数据 和文本字符串 而特别定制的附加序列类型会在专门的小节中描述。

4.6.1 通用序列操作

大多数序列类型，包括可变类型和不可变类型都支持下表中的操作。*collections.abc.Sequence* ABC 被提供用来更容易地在自定义序列类型上正确地实现这些操作。

此表按优先级升序列出了序列操作。在表格中，*s* 和 *t* 是具有相同类型的序列，*n*, *i*, *j* 和 *k* 是整数而 *x* 是任何满足 *s* 所规定的类型和值限制的任意对象。

`in` 和 `not in` 操作具有与比较操作相同的优先级。+ (拼接) 和 * (重复) 操作具有与对应数值运算相同的优先级。[3]

运算	结果：	注释
`x in s`	如果 *s* 中的某项等于 *x* 则结果为 `True`，否则为 `False`	(1)
`x not in s`	如果 *s* 中的某项等于 *x* 则结果为 `False`，否则为 `True`	(1)
`s + t`	*s* 与 *t* 相拼接	(6)(7)
`s * n` 或 `n * s`	相当于 *s* 与自身进行 *n* 次拼接	(2)(7)
`s[i]`	*s* 的第 *i* 项，起始为 0	(3)
`s[i:j]`	*s* 从 *i* 到 *j* 的切片	(3)(4)
`s[i:j:k]`	*s* 从 *i* 到 *j* 步长为 *k* 的切片	(3)(5)
`len(s)`	*s* 的长度	
`min(s)`	*s* 的最小项	
`max(s)`	*s* 的最大项	
`s.index(x[, i[, j]])`	*x* 在 *s* 中首次出现项的索引号（索引号在 *i* 或其后且在 *j* 之前）	(8)
`s.count(x)`	*x* 在 *s* 中出现的总次数	

相同类型的序列也支持比较。特别地，tuple 和 list 的比较是通过比较对应元素的字典顺序。这意味着想要比较结果相等，则每个元素比较结果都必须相等，并且两个序列长度必须相同。（完整细节请参阅语言参考的 comparisons 部分。）

注释:

(1) 虽然 `in` 和 `not in` 操作在通常情况下仅被用于简单的成员检测，某些专门化序列 (例如*str*, *bytes* 和*bytearray*) 也使用它们进行子序列检测:

```
>>> "gg" in "eggs"
True
```

(2) 小于 `0` 的 *n* 值会被当作 `0` 来处理 (生成一个与 *s* 同类型的空序列)。请注意序列 *s* 中的项并不会被拷贝；它们会被多次引用。这一点经常会令 Python 编程新手感到困扰；例如:

[3] 它们必须如此，因为解析器无法区分这些操作数的类型。

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

具体的原因在于 `[[]]` 是一个包含了一个空列表的单元素列表，所以 `[[]] * 3` 结果中的三个元素都是对这一个空列表的引用。修改 `lists` 中的任何一个元素实际上都是对这一个空列表的修改。你可以用以下方式创建以不同列表为元素的列表:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

进一步的解释可以在 FAQ 条目 faq-multidimensional-list 中查看。

(3) 如果 *i* 或 *j* 为负值，则索引顺序是相对于序列 *s* 的末尾: 索引号会被替换为 `len(s) + i` 或 `len(s) + j`。但要注意 `-0` 仍然为 `0`。

(4) *s* 从 *i* 到 *j* 的切片被定义为所有满足 `i <= k < j` 的索引号 *k* 的项组成的序列。如果 *i* 或 *j* 大于 `len(s)`，则使用 `len(s)`。如果 *i* 被省略或为 `None`，则使用 `0`。如果 *j* 被省略或为 `None`，则使用 `len(s)`。如果 *i* 大于等于 *j*，则切片为空。

(5) *s* 从 *i* 到 *j* 步长为 *k* 的切片被定义为所有满足 `0 <= n < (j-i)/k` 的索引号 `x = i + n*k` 的项组成的序列。换句话说，索引号为 `i`, `i+k`, `i+2*k`, `i+3*k`，以此类推，当达到 *j* 时停止 (但一定不包括 *j*)。当 *k* 为正值时，*i* 和 *j* 会被减至不大于 `len(s)`。当 *k* 为负值时，*i* 和 *j* 会被减至不大于 `len(s) - 1`。如果 *i* 或 *j* 被省略或为 `None`，它们会成为“终止”值 (是哪一端的终止值则取决于 *k* 的符号)。请注意，*k* 不可为零。如果 *k* 为 `None`，则当作 `1` 处理。

(6) 拼接不可变序列总是会生成新的对象。这意味着通过重复拼接来构建序列的运行时开销将会基于序列总长度的乘方。想要获得线性的运行时开销，你必须改用下列替代方案之一:

- 如果拼接*str*对象,你可以构建一个列表并在最后使用*str.join()*或是写入一个*io.StringIO*实例并在结束时获取它的值
- 如果拼接*bytes*对象，你可以类似地使用*bytes.join()*或*io.BytesIO*，或者你也可以使用*bytearray*对象进行原地拼接。*bytearray*对象是可变的，并且具有高效的重分配机制
- 如果拼接*tuple*对象，请改为扩展*list*类
- 对于其它类型，请查看相应的文档

(7) 某些序列类型 (例如*range*) 仅支持遵循特定模式的项序列，因此并不支持序列拼接或重复。

(8) 当 *x* 在 *s* 中找不到时 `index` 会引发*ValueError*。不是所有实现都支持传入额外参数 *i* 和 *j*。这两个参数允许高效地搜索序列的子序列。传入这两个额外参数大致相当于使用 `s[i:j].index(x)`，但是不会复制任何数据，并且返回的索引是相对于序列的开头而非切片的开头。

4.6.2 不可变序列类型

不可变序列类型普遍实现而可变序列类型未实现的唯一操作就是对*hash()* 内置函数的支持。

这种支持允许不可变类型，例如*tuple* 实例被用作*dict* 键，以及存储在*set* 和*frozenset* 实例中。

尝试对包含有不可哈希值的不可变序列进行哈希运算将会导致*TypeError*。

4.6.3 可变序列类型

以下表格中的操作是在可变序列类型上定义的。*collections.abc.MutableSequence* ABC 被提供用来更容易地在自定义序列类型上正确实现这些操作。

表格中的 *s* 是可变序列类型的实例，*t* 是任意可迭代对象，而 *x* 是符合对 *s* 所规定类型与值限制的任何对象 (例如，*bytearray* 仅接受满足 `0 <= x <= 255` 值限制的整数)。

运算	结果:	注释
`s[i] = x`	将 *s* 的第 *i* 项替换为 *x*	
`s[i:j] = t`	将 *s* 从 *i* 到 *j* 的切片替换为可迭代对象 *t* 的内容	
`del s[i:j]`	等同于 `s[i:j] = []`	
`s[i:j:k] = t`	将 `s[i:j:k]` 的元素替换为 *t* 的元素	(1)
`del s[i:j:k]`	从列表中移除 `s[i:j:k]` 的元素	
`s.append(x)`	将 *x* 添加到序列的末尾 (等同于 `s[len(s):len(s)] = [x]`)	
`s.clear()`	从 *s* 中移除所有项 (等同于 `del s[:]`)	(5)
`s.copy()`	创建 *s* 的浅拷贝 (等同于 `s[:]`)	(5)
`s.extend(t)` 或 `s += t`	用 *t* 的内容扩展 *s* (基本上等同于 `s[len(s):len(s)] = t`)	
`s *= n`	使用 *s* 的内容重复 *n* 次来对其进行更新	(6)
`s.insert(i, x)`	在由 *i* 给出的索引位置将 *x* 插入 *s* (等同于 `s[i:i] = [x]`)	
`s.pop([i])`	提取在 *i* 位置上的项，并将其从 *s* 中移除	(2)
`s.remove(x)`	remove the first item from *s* where `s[i] == x`	(3)
`s.reverse()`	就地将列表中的元素逆序。	(4)

注释:

(1) *t* 必须与它所替换的切片具有相同的长度。

(2) 可选参数 *i* 默认为 `-1`，因此在默认情况下会移除并返回最后一项。

(3) 当在 *s* 中找不到 *x* 时 `remove` 操作会引发*ValueError*。

(4) 当反转大尺寸序列时 `reverse()` 方法会原地修改该序列以保证空间经济性。为提醒用户此操作是通过间接影响进行的，它并不会返回反转后的序列。

(5) 包括 `clear()` 和 `copy()` 是为了与不支持切片操作的可变容器 (例如*dict* 和*set*) 的接口保持一致

3.3 新版功能: `clear()` 和 `copy()` 方法。

(6) *n* 值为一个整数，或是一个实现了 `__index__()` 的对象。*n* 值为零或负数将清空序列。序列中的项不会被拷贝；它们会被多次引用，正如通用序列操作 中有关 `s * n` 的说明。

4.6.4 列表

列表是可变序列，通常用于存放同类项目的集合（其中精确的相似程度将根据应用而变化）。

class list([*iterable*])

可以用多种方式构建列表：

- 使用一对方括号来表示空列表: `[]`
- 使用方括号，其中的项以逗号分隔: `[a]`, `[a, b, c]`
- 使用列表推导式: `[x for x in iterable]`
- 使用类型的构造器: `list()` 或 `list(iterable)`

构造器将构造一个列表，其中的项与 *iterable* 中的项具有相同的的值与顺序。*iterable* 可以是序列、支持迭代的容器或其它可迭代对象。如果 *iterable* 已经是一个列表，将创建并返回其副本，类似于 `iterable[:]`。例如，`list('abc')` 返回 `['a', 'b', 'c']` 而 `list( (1, 2, 3) )` 返回 `[1, 2, 3]`。如果没有给出参数，构造器将创建一个空列表 `[]`。

其它许多操作也会产生列表，包括*sorted()* 内置函数。

列表实现了所有一般 和可变 序列的操作。列表还额外提供了以下方法：

sort(*, *key*=*None*, *reverse*=*False*)

此方法会对列表进行原地排序，只使用 < 来进行各项间比较。异常不会被屏蔽——如果有任何比较操作失败，整个排序操作将失败（而列表可能会处于被部分修改的状态）。

sort() 接受两个仅限以关键字形式传入的参数 (仅限关键字参数):

key 指定带有一个参数的函数，用于从每个列表元素中提取比较键 (例如 `key=str.lower`)。对应于列表中每一项的键会被计算一次，然后在整个排序过程中使用。默认值 `None` 表示直接对列表项排序而不计算一个单独的键值。

可以使用*functools.cmp_to_key()* 将 2.x 风格的 *cmp* 函数转换为 *key* 函数。

reverse 为一个布尔值。如果设为 `True`，则每个列表元素将按反向顺序比较进行排序。

当顺序大尺寸序列时此方法会原地修改该序列以保证空间经济性。为提醒用户此操作是通过间接影响进行的，它并不会返回排序后的序列（请使用*sorted()* 显示地请求一个新的已排序列表实例）。

sort() 方法确保是稳定的。如果一个排序确保不会改变比较结果相等的元素的相对顺序就称其为稳定的—这有利于进行多重排序（例如先按部门、再接薪级排序）。

CPython implementation detail: 在一个列表被排序期间，尝试改变甚至进行检测也会造成未定义的影响。Python 的 C 实现会在排序期间将列表显示为空，如果发现列表在排序期间被改变将会引发*ValueError*。

4.6.5 元组

元组是不可变序列，通常用于储存异构数据的多项集（例如由*enumerate()* 内置函数所产生的二元组）。元组也被用于需要同构数据的不可变序列的情况（例如允许存储到*set* 或*dict* 的实例）。

class tuple([*iterable*])

可以用多种方式构建元组：

- 使用一对圆括号来表示空元组: `()`
- 使用一个后缀的逗号来表示单元组: `a,` 或 `(a,)`
- 使用以逗号分隔的多个项: `a, b, c` or `(a, b, c)`

- 使用内置的*tuple()*: `tuple()` 或 `tuple(iterable)`

构造器将构造一个元组，其中的项与 *iterable* 中的项具有相同的值与顺序。*iterable* 可以是序列、支持迭代的容器或其他可迭代对象。如果 *iterable* 已经是一个元组，会不加改变地将其返回。例如，`tuple('abc')` 返回 `('a', 'b', 'c')` 而 `tuple( [1, 2, 3] )` 返回 `(1, 2, 3)`。如果没有给出参数，构造器将创建一个空元组 `()`。

请注意决定生成元组的其实是逗号而不是圆括号。圆括号只是可选的，生成空元组或需要避免语法歧义的情况除外。例如，`f(a, b, c)` 是在调用函数时附带三个参数，而 `f((a, b, c))` 则是在调用函数时附带一个三元组。

元组实现了所有一般 序列的操作。

对于通过名称访问相比通过索引访问更清晰的异构数据多项集，*collections.namedtuple()* 可能是比简单元组对象更为合适的选择。

4.6.6 range 对象

range 类型表示不可变的数字序列，通常用于在 `for` 循环中循环指定的次数。

class range(*stop*)
class range(*start*, *stop*[, *step*])

range 构造器的参数必须为整数（可以是内置的*int* 或任何实现了 `__index__` 特殊方法的对象）。如果省略 *step* 参数，其默认值为 `1`。如果省略 *start* 参数，其默认值为 `0`，如果 *step* 为零则会引发*ValueError*。

如果 *step* 为正值，确定 range `r` 内容的公式为 `r[i] = start + step*i` 其中 `i >= 0` 且 `r[i] < stop`。

如果 *step* 为负值，确定 range 内容的公式仍然为 `r[i] = start + step*i`，但限制条件改为 `i >= 0` 且 `r[i] > stop`.

如果 `r[0]` 不符合值的限制条件，则该 range 对象为空。range 对象确实支持负索引，但是会将其解读为从正索引所确定的序列的末尾开始索引。

元素绝对值大于*sys.maxsize* 的 range 对象是被允许的，但某些特性 (例如*len()*) 可能引发*OverflowError*。

一些 range 对象的例子:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

range 对象实现了一般 序列的所有操作，但拼接和重复除外（这是由于 range 对象只能表示符合严格模式的序列，而重复和拼接通常都会违反这样的模式）。

start
start 形参的值 (如果该形参未提供则为 `0`)

stop
stop 形参的值

step
step 形参的值 (如果该形参未提供则为 1)

range 类型相比常规*list* 或*tuple* 的优势在于一个*range* 对象总是占用固定数量的（较小）内存，不论其所表示的范围有多大（因为它只保存了 start, stop 和 step 值，并会根据需要计算具体单项或子范围的值）。

range 对象实现了*collections.abc.Sequence* ABC，提供如包含检测、元素索引查找、切片等特性，并支持负索引 (参见*序列类型—list, tuple, range*):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

使用 == 和 != 检测 range 对象是否相等是将其作为序列来比较。也就是说，如果两个 range 对象表示相同的值序列就认为它们是相等的。（请注意比较结果相等的两个 range 对象可能会具有不同的*start*, *stop* 和*step* 属性，例如 `range(0) == range(2, 1, 3)` 而 `range(0, 3, 2) == range(0, 4, 2)`。）

在 3.2 版更改: 实现 Sequence ABC。支持切片和负数索引。使用*int* 对象在固定时间内进行成员检测，而不是逐一迭代所有项。

在 3.3 版更改: 定义‘==’和‘!=’以根据 range 对象所定义的值序列来进行比较（而不是根据对象的标识）。

3.3 新版功能: *start*, *stop* 和*step* 属性。

参见:

- linspace recipe 演示了如何实现一个延迟求值版本的适合浮点数应用的 range 对象。

4.7 文本序列类型—str

在 Python 中处理文本数据是使用*str* 对象，也称为 *字符串*。字符串是由 Unicode 码位构成的不可变*序列*。字符串字面值有多种不同的写法：

- 单引号: `'允许包含有" 双" 引号'`
- 双引号: `"允许包含有 ' 单' 引号"`。
- 三重引号: `'''三重单引号'''`, `"""三重双引号"""`

使用三重引号的字符串可以跨越多行——其中所有的空白字符都将包含在该字符串字面值中。

作为单一表达式组成部分，之间只由空格分隔的多个字符串字面值会被隐式地转换为单个字符串字面值。也就是说，`("spam " "eggs") == "spam eggs"`。

请参阅 strings 有解有关不同字符串字面值的更多信息，包括所支持的转义序列，以及使用 r（“raw”）前缀来禁用大多数转义序列的处理。

字符串也可以通过使用 *str* 构造器从其他对象创建。

由于不存在单独的“字符”类型，对字符串做索引操作将产生一个长度为 1 的字符串。也就是说，对于一个非空字符串 *s*, `s[0] == s[0:1]`。

不存在可变的字符串类型，但是 *str.join()* 或 *io.StringIO* 可以被被用来根据多个片段高效率地构建字符串。

在 3.3 版更改: 为了与 Python 2 系列的向下兼容，再次允许字符串字面值使用 u 前缀。它对字符串字面值的含义没有影响，并且不能与 r 前缀同时出现。

class str(*object=''*)
class str(*object=b''*, *encoding='utf-8'*, *errors='strict'*)

返回 *object* 的字符串 版本。如果未提供 *object* 则返回空字符串。在其他情况下 `str()` 的行为取决于 *encoding* 或 *errors* 是否有给出，具体见下。

如果 *encoding* 或 *errors* 均未给出，`str(object)` 返回 `object.__str__()`，这是 *object* 的“非正式”或格式良好的字符串表示。对于字符串对象，这是该字符串本身。如果 *object* 没有 `__str__()` 方法，则 *str()* 将回退为返回 *repr(object)*。

如果 *encoding* 或 *errors* 至少给出其中之一，则 *object* 应该是一个 *bytes-like object* (例如 *bytes* 或 *bytearray*)。在此情况下，如果 *object* 是一个 *bytes* (或 *bytearray*) 对象，则 `str(bytes, encoding, errors)` 等价于 *bytes.decode(encoding, errors)*。否则的话，会在调用 *bytes.decode()* 之前获取缓冲区对象下层的 bytes 对象。请参阅二进制序列类型—*bytes, bytearray, memoryview* 与 bufferobjects 了解有关缓冲区对象的信息。

将一个 *bytes* 对象传入 *str()* 而不给出 *encoding* 或 *errors* 参数的操作属于第一种情况，将返回非正式的字符串表示（另请参阅 Python 的 -b 命令行选项）。例如:

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

有关 str 类及其方法的更多信息，请参阅下面的文本序列类型—*str* 和字符串的方法 小节。要输出格式化字符串，请参阅 f-strings 和格式字符串语法 小节。此外还可以参阅文本处理服务 小节。

4.7.1 字符串的方法

字符串实现了所有一般 序列的操作，还额外提供了以下列出的一些附加方法。

字符串还支持两种字符串格式化样式，一种提供了很大程度的灵活性和可定制性 (参阅 *str.format()*, 格式字符串语法 和自定义字符串格式化) 而另一种是基于 C printf 样式的格式化，它可处理的类型范围较窄，并且更难以正确使用，但对于它可处理的情况往往会更为快速 (*printf* 风格的字符串格式化)。

标准库的文本处理服务 部分涵盖了许多其他模块，提供各种文本相关工具（例如包含于 *re* 模块中的正则表达式支持）。

str.**capitalize**()
返回原字符串的副本，其首个字符大写，其余为小写。

str.**casefold**()
返回原字符串消除大小写的副本。消除大小写的字符串可用于忽略大小写的匹配。

消除大小写类似于转为小写，但是更加彻底一些，因为它会移除字符串中的所有大小写变化形式。例如，德语小写字母 'ß' 相当于 "ss"。由于它已经是小写了，*lower()* 不会对 'ß' 做任何改变；而 *casefold()* 则会将其转换为 "ss"。

消除大小写算法的描述请参见 Unicode 标准的 3.13 节。

3.3 新版功能.

str.**center**(*width*[, *fillchar*])
返回长度为 *width* 的字符串，原字符串在其正中。使用指定的 *fillchar* 填充两边的空位（默认使用 ASCII 空格符）。如果 *width* 小于等于 `len(s)` 则返回原字符串的副本。

str.**count**(*sub*[, *start*[, *end*]])
返回子字符串 *sub* 在 [*start*, *end*] 范围内非重叠出现的次数。可选参数 *start* 与 *end* 会被解读为切片表示法。

str.**encode**(*encoding="utf-8"*, *errors="strict"*)
返回原字符串编码为字节串对象的版本。默认编码为 `'utf-8'`。可以给出 *errors* 来设置不同的错误处理方案。*errors* 的默认值为 `'strict'`，表示编码错误会引发*UnicodeError*。其他可用的值为 `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` 以及任何其他通过*codecs.register_error()* 注册的值，请参阅错误处理方案 小节。要查看可用的编码列表，请参阅标准编码 小节。

在 3.1 版更改: 加入了对关键字参数的支持。

str.**endswith**(*suffix*[, *start*[, *end*]])
如果字符串以指定的 *suffix* 结束返回 `True`，否则返回 `False`。*suffix* 也可以为由多个供查找的后缀构成的元组。如果有可选项 *start*，将从所指定位置开始检查。如果有可选项 *end*，将在所指定位置停止比较。

str.**expandtabs**(*tabsize=8*)
返回字符串的副本，其中所有的制表符会由一个或多个空格替换，具体取决于当前列位置和给定的制表符宽度。每 *tabsize* 个字符设为一个制表位（默认值 8 时设定的制表位在列 0, 8, 16 依次类推）。要展开字符串，当前列将被设为零并逐一检查字符串中的每个字符。如果字符为制表符 (`\t`)，则会在结果中插入一个或多个空格符，直到当前列等于下一个制表位。（制表符本身不会被复制。）如果字符为换行符 (`\n`) 或回车符 (`\r`)，它会被复制并将当前列重设为零。任何其他字符会被不加修改地复制并将当前列加一，不论该字符在被打印时会如何显示。

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

str.**find**(*sub*[, *start*[, *end*]])
返回子字符串 *sub* 在 `s[start:end]` 切片内被找到的最小索引。可选参数 *start* 与 *end* 会被解读为切片表示法。如果 *sub* 未被找到则返回 `-1`。

注解: *find()* 方法应该只在你需要知道 *sub* 所在位置时使用。要检查 *sub* 是否为子字符串，请使用 `in` 操作符:

```
>>> 'Py' in 'Python'
True
```

---

str.**format**(**args*, ***kwargs*)
执行字符串格式化操作。调用此方法的字符串可以包含字符串字面值或者以花括号 `{}` 括起来的替换域。每个替换域可以包含一个位置参数的数字索引，或者一个关键字参数的名称。返回的字符串副本中每个替换域都会被替换为对应参数的字符串值。

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

请参阅格式字符串语法 了解有关可以在格式字符串中指定的各种格式选项的说明。

注解: 当使用 n 类型 (例如: `'{:n}'.format(1234)`) 来格式化数字 (*int*, *float*, *complex*, *decimal.Decimal* 及其子类) 的时候，该函数会临时性地将 LC_CTYPE 区域设置为 LC_NUMERIC 区域以解码 `localeconv()` 的 `decimal_point` 和 `thousands_sep` 字段，如果它们是非 ASCII 字符或长度超过 1 字节的话，并且 LC_NUMERIC 区域会与 LC_CTYPE 区域不一致。这个临时更改会影响其他线程。

在 3.6.5 版更改: 当使用 n 类型格式化数字时，该函数在某些情况下会临时性地将 LC_CTYPE 区域设置为 LC_NUMERIC 区域。

str.**format_map**(*mapping*)

类似于 `str.format(**mapping)`，不同之处在于 `mapping` 会被直接使用而不是复制到一个*dict*。适宜使用此方法的一个例子是当 `mapping` 为 dict 的子类的情况：

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

3.2 新版功能.

str.**index**(*sub*[, *start*[, *end*]])

类似于*find()*，但在找不到子类时会引发*ValueError*。

str.**isalnum**()

Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise. A character `c` is alphanumeric if one of the following returns `True`: `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, or `c.isnumeric()`.

str.**isalpha**()

Return true if all characters in the string are alphabetic and there is at least one character, false otherwise. Alphabetic characters are those characters defined in the Unicode character database as "Letter", i.e., those with general category property being one of "Lm", "Lt", "Lu", "Ll", or "Lo". Note that this is different from the "Alphabetic" property defined in the Unicode Standard.

str.**isdecimal**()

Return true if all characters in the string are decimal characters and there is at least one character, false otherwise. Decimal characters are those that can be used to form numbers in base 10, e.g. U+0660, ARABIC-INDIC DIGIT ZERO. Formally a decimal character is a character in the Unicode General Category "Nd".

str.**isdigit**()

Return true if all characters in the string are digits and there is at least one character, false otherwise. Digits include decimal characters and digits that need special handling, such as the compatibility superscript digits. This covers digits which cannot be used to form numbers in base 10, like the Kharosthi numbers. Formally, a digit is a character that has the property value Numeric_Type=Digit or Numeric_Type=Decimal.

str.**isidentifier**()

Return true if the string is a valid identifier according to the language definition, section identifiers.

请使用*keyword.iskeyword()* 来检测保留标识符，例如 `def` 和 `class`。

str.**islower**()

Return true if all cased characters[4] in the string are lowercase and there is at least one cased character, false otherwise.

[4] 区分大小写的字符是指所属一般类别属性为 "Lu" (Letter, uppercase), "Ll" (Letter, lowercase) 或 "Lt" (Letter, titlecase) 之一的字符。

str.**isnumeric**()
Return true if all characters in the string are numeric characters, and there is at least one character, false otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. U+2155, VULGAR FRACTION ONE FIFTH. Formally, numeric characters are those with the property value Numeric_Type=Digit, Numeric_Type=Decimal or Numeric_Type=Numeric.

str.**isprintable**()
Return true if all characters in the string are printable or the string is empty, false otherwise. Nonprintable characters are those characters defined in the Unicode character database as "Other" or "Separator", excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when *repr()* is invoked on a string. It has no bearing on the handling of strings written to *sys.stdout* or *sys.stderr*.)

str.**isspace**()
Return true if there are only whitespace characters in the string and there is at least one character, false otherwise. Whitespace characters are those characters defined in the Unicode character database as "Other" or "Separator" and those with bidirectional property being one of "WS", "B", or "S".

str.**istitle**()
Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

str.**isupper**()
Return true if all cased characters[4] in the string are uppercase and there is at least one cased character, false otherwise.

str.**join**(*iterable*)
返回一个由 *iterable* 中的字符串拼接而成的字符串。如果 *iterable* 中存在任何非字符串值包括*bytes* 对象则会引发*TypeError*。调用该方法的字符串将作为元素之间的分隔。

str.**ljust**(*width*[, *fillchar*])
返回长度为 *width* 的字符串，原字符串在其中靠左对齐。使用指定的 *fillchar* 填充空位 (默认使用 ASCII 空格符)。如果 *width* 小于等于 `len(s)` 则返回原字符串的副本。

str.**lower**()
返回原字符串的副本，其所有区分大小写的字符[4] 均转换为小写。

所用转换小写算法的描述请参见 Unicode 标准的 3.13 节。

str.**lstrip**([*chars*])
返回原字符串的副本，移除其中的前导字符。*chars* 参数为指定要移除字符的字符串。如果省略或为 `None`，则 *chars* 参数默认移除空白符。实际上 *chars* 参数并非指定单个前缀；而是会移除参数值的所有组合:

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

static str.**maketrans**(*x*[, *y*[, *z*]])
此静态方法返回一个可供*str.translate()* 使用的转换对照表。

如果只有一个参数，则它必须是一个将 Unicode 码位序号（整数）或字符（长度为 1 的字符串）映射到 Unicode 码位序号、（任意长度的）字符串或 `None` 的字典。字符键将会被转换为码位序号。

如果有两个参数，则它们必须是两个长度相等的字符串，并且在结果字典中，x 中每个字符将被映射到 y 中相同位置的字符。如果有第三个参数，它必须是一个字符串，其中的字符将在结果中被映射到 `None`。

str.**partition**(*sep*)
在 *sep* 首次出现的位置拆分字符串，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含字符本身以及两个空字符串。

str.**replace**(*old*, *new*[, *count*])
返回字符串的副本，其中出现的所有子字符串 *old* 都将被替换为 *new*。如果给出了可选参数 *count*，则只替换前 *count* 次出现。

str.**rfind**(*sub*[, *start*[, *end*]])
返回子字符串 *sub* 在字符串内被找到的最大（最右）索引，这样 *sub* 将包含在 `s[start:end]` 当中。可选参数 *start* 与 *end* 会被解读为切片表示法。如果未找到则返回 `-1`。

str.**rindex**(*sub*[, *start*[, *end*]])
类似于*rfind()*，但在子字符串 *sub* 未找到时会引发*ValueError*。

str.**rjust**(*width*[, *fillchar*])
返回长度为 *width* 的字符串，原字符串在其中靠右对齐。使用指定的 *fillchar* 填充空位 (默认使用 ASCII 空格符)。如果 *width* 小于等于 `len(s)` 则返回原字符串的副本。

str.**rpartition**(*sep*)
在 *sep* 最后一次出现的位置拆分字符串，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含两个空字符串以及字符串本身。

str.**rsplit**(*sep=None*, *maxsplit=-1*)
返回一个由字符串内单词组成的列表，使用 *sep* 作为分隔字符串。如果给出了 *maxsplit*，则最多进行 *maxsplit* 次拆分，从 *最右边*开始。如果 *sep* 未指定或为 `None`，任何空白字符串都会被作为分隔符。除了从右边开始拆分，*rsplit()* 的其他行为都类似于下文所述的*split()*。

str.**rstrip**([*chars*])
返回原字符串的副本，移除其中的末尾字符。*chars* 参数为指定要移除字符的字符串。如果省略或为 `None`，则 *chars* 参数默认移除空白符。实际上 *chars* 参数并非指定单个后缀；而是会移除参数值的所有组合:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

str.**split**(*sep=None*, *maxsplit=-1*)
返回一个由字符串内单词组成的列表，使用 *sep* 作为分隔字符串。如果给出了 *maxsplit*，则最多进行 *maxsplit* 次拆分（因此，列表最多会有 `maxsplit+1` 个元素）。如果 *maxsplit* 未指定或为 `-1`，则不限制拆分次数（进行所有可能的拆分）。

如果给出了 *sep*，则连续的分隔符不会被组合在一起而是被视为分隔空字符串 (例如 `'1,,2'.split(',')` 将返回 `['1', '', '2']`)。*sep* 参数可能由多个字符组成 (例如 `'1<>2<>3'.split('<>')` 将返回 `['1', '2', '3']`)。使用指定的分隔符拆分空字符串将返回 `['']`。

例如

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

如果 *sep* 未指定或为 `None`，则会应用另一种拆分算法：连续的空格会被视为单个分隔符，其结果将不包含开头或末尾的空字符串，如果字符串包含前缀或后缀空格的话。因此，使用 `None` 拆分空字符串或仅包含空格的字符串将返回 `[]`。

例如

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

str.**splitlines**([*keepends*])

返回由原字符串中各行组成的列表，在行边界的位置拆分。结果列表中不包含行边界，除非给出了 *keepends* 且为真值。

此方法会以下列行边界进行拆分。特别地，行边界是*universal newlines* 的一个超集。

表示符	描述
\n	换行
\r	回车
\r\n	回车 + 换行
\v 或 \x0b	行制表符
\f 或 \x0c	换表单
\x1c	文件分隔符
\x1d	组分隔符
\x1e	记录分隔符
\x85	下一行 (C1 控制码)
\u2028	行分隔符
\u2029	段分隔符

在 3.2 版更改: \v 和 \f 被添加到行边界列表

例如

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

不同于*split()*，当给出了分隔字符串 *sep* 时，对于空字符串此方法将返回一个空列表，而末尾的换行不会令结果中增加额外的行:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

作为比较，split('\n') 的结果为:

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

str.**startswith**(*prefix*[, *start*[, *end*]])

如果字符串以指定的 *prefix* 开始则返回 True，否则返回 False。*prefix* 也可以为由多个供查找的前缀构成的元组。如果有可选项 *start*，将从所指定位置开始检查。如果有可选项 *end*，将在所指定位置停止比较。

str.**strip**([*chars*])
返回原字符串的副本，移除其中的前导和末尾字符。*chars* 参数为指定要移除字符的字符串。如果省略或为 None，则 *chars* 参数默认移除空白符。实际上 *chars* 参数并非指定单个前缀或后缀；而是会移除参数值的所有组合:

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

最外侧的前导和末尾 *chars* 参数值将从字符串中移除。开头端的字符的移除将在遇到一个未包含于 *chars* 所指定字符集的字符时停止。类似的操作也将在结尾端发生。例如:

```
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

str.**swapcase**()
返回原字符串的副本，其中大写字符转换为小写，反之亦然。请注意 s.swapcase().swapcase() == s 并不一定为真值。

str.**title**()
返回原字符串的标题版本，其中每个单词第一个字母为大写，其余字母为小写。

例如

```
>>> 'Hello world'.title()
'Hello World'
```

该算法使用一种简单的与语言无关的定义，将连续的字母组合视为单词。该定义在多数情况下都很有效，但它也意味着代表缩写形式与所有格的撇号也会成为单词边界，这可能导致不希望的结果:

```
>>> "they're bill's friends from the UK".title()
"They're Bill'S Friends From The Uk"
```

可以使用正则表达式来构建针对撇号的特别处理:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

str.**translate**(*table*)
返回原字符串的副本，其中每个字符按给定的转换表进行映射。转换表必须是一个使用 __getitem__() 来实现索引操作的对象，通常为*mapping* 或*sequence*。当以 Unicode 码位序号（整数）为索引时，转换表对象可以做以下任何一种操作：返回 Unicode 序号或字符串，将字符映射为一个或多个字符；返回 None，将字符从结果字符串中删除；或引发*LookupError* 异常，将字符映射为其自身。

你可以使用*str.maketrans()* 基于不同格式的字符到字符映射来创建一个转换映射表。

另请参阅*codecs* 模块以了解定制字符映射的更灵活方式。

str.**upper**()
返回原字符串的副本，其中所有区分大小写的字符[4] 均转换为大写。请注意如果 s 包含不区分大小写的字符或者如果结果字符的 Unicode 类别不是 “Lu” (Letter, uppercase) 而是 “Lt” (Letter, titlecase) 则 s.upper().isupper() 有可能为 False。

所用转换大写算法的描述请参见 Unicode 标准的 3.13 节。

str.**zfill**(*width*)
返回原字符串的副本，在左边填充 ASCII '0' 数码使其长度变为 *width*。正负值前缀 ('+'/'-') 的处理方式是在正负符号之后填充而非在之前。如果 *width* 小于等于 len(s) 则返回原字符串的副本。

例如

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

4.7.2 printf 风格的字符串格式化

注解： The formatting operations described here exhibit a variety of quirks that lead to a number of common errors (such as failing to display tuples and dictionaries correctly). Using the newer formatted string literals or the *str.format()* interface helps avoid these errors. These alternatives also provide more powerful, flexible and extensible approaches to formatting text.

字符串具有一种特殊的内置操作：使用 % (取模) 运算符。这也被称为字符串的 *格式化* 或 *插值* 运算符。对于 format % values (其中 *format* 为一个字符串)，在 *format* 中的 % 转换标记符将被替换为零个或多个 *values* 条目。其效果类似于在 C 语言中使用 sprintf()。

如果 *format* 要求一个单独参数，则 *values* 可以为一个非元组对象。[5] 否则的话，*values* 必须或者是一个包含项数与格式字符串中指定的转换符项数相同的元组，或者是一个单独映射对象（例如字典）。

转换标记符包含两个或更多字符并具有以下组成，且必须遵循此处规定的顺序：

1. '%' 字符，用于标记转换符的起始。
2. 映射键（可选），由加圆括号的字符序列组成 (例如 (somename))。
3. 转换旗标（可选），用于影响某些转换类型的结果。
4. 最小字段宽度（可选）。如果指定为 '*' (星号)，则实际宽度会从 *values* 元组的下一元素中读取，要转换的对象则为最小字段宽度和可选的精度之后的元素。
5. 精度（可选），以在 '.' (点号) 之后加精度值的形式给出。如果指定为 '*' (星号)，则实际精度会从 *values* 元组的下一元素中读取，要转换的对象则为精度之后的元素。
6. 长度修饰符（可选）。
7. 转换类型。

当右边的参数为一个字典（或其他映射类型）时，字符串中的格式 必须包含加圆括号的映射键，对应 '%' 字符之后字典中的每一项。映射键将从映射中选取要格式化的值。例如：

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

[5] 要格式化单独一个元组，那么你应当提供一个单例元组，其唯一的元素就是要被格式化的元组。

在此情况下格式中不能出现 * 标记符（因其需要一个序列类的参数列表）。
转换旗标为：

标志	含义
'#'	值的转换将使用“替代形式”（具体定义见下文）。
'0'	转换将为数字值填充零字符。
'-'	转换值将靠左对齐（如果同时给出 '0' 转换，则会覆盖后者）。
' '	(空格) 符号位转换产生的正数（或空字符串）前将留出一个空格。
'+'	符号字符 ('+' 或 '-') 将显示于转换结果的开头（会覆盖“空格”旗标）。

可以给出长度修饰符 (h, l 或 L)，但会被忽略，因为对 Python 来说没有必要–所以 %ld 等价于 %d。
转换类型为：

转换符	含义	注释
'd'	有符号十进制整数。	
'i'	有符号十进制整数。	
'o'	有符号八进制数。	(1)
'u'	过时类型–等价于 'd'。	(6)
'x'	有符号十六进制数（小写）。	(2)
'X'	有符号十六进制数（大写）。	(2)
'e'	浮点指数格式（小写）。	(3)
'E'	浮点指数格式（大写）。	(3)
'f'	浮点十进制格式。	(3)
'F'	浮点十进制格式。	(3)
'g'	浮点格式。如果指数小于 -4 或不小于精度则使用小写指数格式，否则使用十进制格式。	(4)
'G'	浮点格式。如果指数小于 -4 或不小于精度则使用大写指数格式，否则使用十进制格式。	(4)
'c'	单个字符（接受整数或单个字符的字符串）。	
'r'	字符串（使用*repr()* 转换任何 Python 对象）。	(5)
's'	字符串（使用*str()* 转换任何 Python 对象）。	(5)
'a'	字符串（使用*ascii()* 转换任何 Python 对象）。	(5)
'%'	不转换参数，在结果中输出一个 '%' 字符。	

注释:

(1) 此替代形式会在第一个数码之前插入标示八进制数的前缀 ('0o')。

(2) 此替代形式会在第一个数码之前插入 '0x' 或 '0X' 前缀（取决于是使用 'x' 还是 'X' 格式）。

(3) 此替代形式总是会在结果中包含一个小数点，即使其后并没有数码。

小数点后的数码位数由精度决定，默认为 6。

(4) 此替代形式总是会在结果中包含一个小数点，末尾各位的零不会如其他情况下那样被移除。

小数点前后的有效数码位数由精度决定，默认为 6。

(5) 如果精度为 N，输出将截短为 N 个字符。

(6) 参见 **PEP 237**。

由于 Python 字符串显式指明长度，%s 转换不会将 '\0' 视为字符串的结束。

在 3.1 版更改: 绝对值超过 1e50 的 %f 转换不会再被替换为 %g 转换。

4.8 二进制序列类型—bytes, bytearray, memoryview

操作二进制数据的核心内置类型是*bytes*和*bytearray*。它们由*memoryview*提供支持，该对象使用 缓冲区协议来访问其他二进制对象所在内存，不需要创建对象的副本。

*array*模块支持高效地存储基本数据类型，例如 32 位整数和 IEEE754 双精度浮点值。

4.8.1 bytes 对象

bytes 对象是由单个字节构成的不可变序列。由于许多主要二进制协议都基于 ASCII 文本编码，因此 bytes 对象提供了一些仅在处理 ASCII 兼容数据时可用，并且在许多特性上与字符串对象紧密相关的方法。

class bytes([*source*[, *encoding*[, *errors*]]])

首先，表示 bytes 字面值的语法与字符串字面值的大致相同，只是添加了一个 `b` 前缀：

- 单引号: `b' 同样允许嵌入" 双" 引号'`。
- 双引号: `b" 同样允许嵌入 ' 单' 引号"`。
- 三重引号: `b''' 三重单引号''',b""" 三重双引号"""`

bytes 字面值中只允许 ASCII 字符（无论源代码声明的编码为何）。任何超出 127 的二进制值必须使用相应的转义序列形式加入 bytes 字面值。

像字符串字面值一样，bytes 字面值也可以使用 `r` 前缀来禁用转义序列处理。请参阅 strings 了解有关各种 bytes 字面值形式的详情，包括所支持的转义序列。

虽然 bytes 字面值和表示法是基于 ASCII 文本的，但 bytes 对象的行为实际上更像是不可变的整数序列，序列中的每个值的大小被限制为 `0 <= x < 256`（如果违反此限制将引发*ValueError*）。这种限制是有意设计用以强调以下事实，虽然许多二进制格式都包含基于 ASCII 的元素，可以通过某些面向文本的算法进行有用的操作，但情况对于任意二进制数据来说通常却并非如此（盲目地将文本处理算法应用于不兼容 ASCII 的二进制数据格式往往将导致数据损坏）。

除了字面值形式，bytes 对象还可以通过其他几种方式来创建：

- 指定长度的以零值填充的 bytes 对象: `bytes(10)`
- 通过由整数组成的可迭代对象: `bytes(range(20))`
- 通过缓冲区协议复制现有的二进制数据: `bytes(obj)`

另请参阅*bytes* 内置类型。

由于两个十六进制数码精确对应一个字节，因此十六进制数是描述二进制数据的常用格式。相应地，bytes 类型具有从此种格式读取数据的附加类方法：

classmethod fromhex(*string*)

此*bytes*类方法返回一个解码给定字符串的 bytes 对象。字符串必须由表示每个字节的两个十六进制数码构成，其中的 ASCII 空白符会被忽略。

```
>>> bytes.fromhex('2Ef0 F1f2  ')
b'.\xf0\xf1\xf2'
```

存在一个反向转换函数，可以将 bytes 对象转换为对应的十六进制表示。

hex()

返回一个字符串对象，该对象包含实例中每个字节的两个十六进制数字。

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

3.5 新版功能.

由于 bytes 对象是由整数构成的序列（类似于元组），因此对于一个 bytes 对象 *b*，`b[0]` 将为一个整数，而 `b[0:1]` 将为一个长度为 1 的 bytes 对象。（这与文本字符串不同，索引和切片所产生的将都是一个长度为 1 的字符串）。

bytes 对象的表示使用字面值格式 (`b'...'`)，因为它通常都要比像 `bytes([46, 46, 46])` 这样的格式更好用。你总是可以使用 `list(b)` 将 bytes 对象转换为一个由整数构成的列表。

注解： 针对 Python 2.x 用户的说明：在 Python 2.x 系列中，允许 8 位字符串（2.x 所提供的最接近内置二进制数据类型的对象）与 Unicode 字符串进行各种隐式转换。这是为了实现向下兼容的变通做法，以适应 Python 最初只支持 8 位文本而 Unicode 文本是后来才被加入这一事实。在 Python 3.x 中，这些隐式转换已被取消——8 位二进制数据与 Unicode 文本间的转换必须显式地进行，bytes 与字符串对象的比较结果将总是不相等。

4.8.2 bytearray 对象

bytearray 对象是*bytes* 对象的可变对应物。

class bytearray([*source*[, *encoding*[, *errors*]]])

bytearray 对象没有专属的字面值语法，它们总是通过调用构造器来创建：

- 创建一个空实例: `bytearray()`
- 创建一个指定长度的以零值填充的实例: `bytearray(10)`
- 通过由整数组成的可迭代对象: `bytearray(range(20))`
- 通过缓冲区协议复制现有的二进制数据: `bytearray(b'Hi!')`

由于 bytearray 对象是可变的，该对象除了*bytes 和 bytearray 操作* 中所描述的 bytes 和 bytearray 共有操作之外，还支持可变 序列操作。

另请参见*bytearray* 内置类型。

由于两个十六进制数码精确对应一个字节，因此十六进制数是描述二进制数据的常用格式。相应地，bytearray 类型具有从此种格式读取数据的附加类方法：

classmethod fromhex(*string*)

bytearray 类方法返回一个解码给定字符串的 bytearray 对象。字符串必须由表示每个字节的两个十六进制数码构成，其中的 ASCII 空白符会被忽略。

```
>>> bytearray.fromhex('2Ef0 F1f2  ')
bytearray(b'.\xf0\xf1\xf2')
```

存在一个反向转换函数，可以将 bytearray 对象转换为对应的十六进制表示。

hex()

返回一个字符串对象，该对象包含实例中每个字节的两个十六进制数字。

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

3.5 新版功能.

由于 bytearray 对象是由整数构成的序列（类似于列表），因此对于一个 bytearray 对象 *b*，`b[0]` 将为一个整数，而 `b[0:1]` 将为一个长度为 1 的 bytearray 对象。（这与文本字符串不同，索引和切片所产生的将都是一个长度为 1 的字符串）。

bytearray 对象的表示使用 bytes 对象字面值格式 (`bytearray(b'...')`)，因为它通常都要比 `bytearray([46, 46, 46])` 这样的格式更好用。你总是可以使用 `list(b)` 将 bytearray 对象转换为一个由整数构成的列表。

4.8.3 bytes 和 bytearray 操作

bytes 和 bytearray 对象都支持通用 序列操作。它们不仅能与相同类型的操作数，也能与任何*bytes-like object* 进行互操作。由于这样的灵活性，它们可以在操作中自由地混合而不会导致错误。但是，操作结果的返回值类型可能取决于操作数的顺序。

注解： bytes 和 bytearray 对象的方法不接受字符串作为其参数，就像字符串的方法不接受 bytes 对象作为其参数一样。例如，你必须使用以下写法:

```
a = "abc"
b = a.replace("a", "f")
```

和:

```
a = b"abc"
b = a.replace(b"a", b"f")
```

某些 bytes 和 bytearray 操作假定使用兼容 ASCII 的二进制格式，因此在处理任意二进数数据时应当避免使用。这些限制会在下文中说明。

注解： 使用这些基于 ASCII 的操作来处理未以基于 ASCII 的格式存储的二进制数据可能会导致数据损坏。

---

bytes 和 bytearray 对象的下列方法可以用于任意二进制数据。

`bytes.`**`count`**(*sub*[, *start*[, *end*]])
`bytearray.`**`count`**(*sub*[, *start*[, *end*]])
返回子序列 *sub* 在 [*start*, *end*] 范围内非重叠出现的次数。可选参数 *start* 与 *end* 会被解读为切片表示法。

要搜索的子序列可以是任意*bytes-like object* 或是 0 至 255 范围内的整数。

在 3.3 版更改: 也接受 0 至 255 范围内的整数作为子序列。

`bytes.`**`decode`**(*encoding="utf-8"*, *errors="strict"*)
`bytearray.`**`decode`**(*encoding="utf-8"*, *errors="strict"*)
返回从给定 bytes 解码出来的字符串。默认编码为 `'utf-8'`。可以给出 *errors* 来设置不同的错误处理方案。*errors* 的默认值为 `'strict'`，表示编码错误会引发*`UnicodeError`*。其他可用的值为 `'ignore'`，`'replace'` 以及任何其他通过*`codecs.register_error()`* 注册的名称，请参阅错误处理方案 小节。要查看可用的编码列表，请参阅标准编码 小节。

注解： 将 *encoding* 参数传给*`str`* 允许直接解码任何*bytes-like object*，无须创建临时的 bytes 或 bytearray 对象。

在 3.1 版更改: 加入了对关键字参数的支持。

`bytes.`**`endswith`**(*suffix*[, *start*[, *end*]])
`bytearray.`**`endswith`**(*suffix*[, *start*[, *end*]])
如果二进制数据以指定的 *suffix* 结束则返回 `True`，否则返回 `False`。*suffix* 也可以为由多个供查找的后缀构成的元组。如果有可选项 *start*，将从所指定位置开始检查。如果有可选项 *end*，将在所指定位置停止比较。

要搜索的后缀可以是任意*bytes-like object*。

bytes.**find**(*sub*[, *start*[, *end*]])
bytearray.**find**(*sub*[, *start*[, *end*]])
返回子序列 *sub* 在数据中被找到的最小索引，*sub* 包含于切片 `s[start:end]` 之内。可选参数 *start* 与 *end* 会被解读为切片表示法。如果 *sub* 未被找到则返回 `-1`。

要搜索的子序列可以是任意*bytes-like object* 或是 0 至 255 范围内的整数。

注解： *find()* 方法应该只在你需要知道 *sub* 所在位置时使用。要检查 *sub* 是否为子串，请使用 `in` 操作符:

```
>>> b'Py' in b'Python'
True
```

在 3.3 版更改: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**index**(*sub*[, *start*[, *end*]])
bytearray.**index**(*sub*[, *start*[, *end*]])
类似于*find()*，但在找不到子序列时会引发*ValueError*。

要搜索的子序列可以是任意*bytes-like object* 或是 0 至 255 范围内的整数。

在 3.3 版更改: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**join**(*iterable*)
bytearray.**join**(*iterable*)
返回一个由 *iterable* 中的二进制数据序列拼接而成的 bytes 或 bytearray 对象。如果 *iterable* 中存在任何非*字节类对象* 包括存在*str* 对象值则会引发*TypeError*。提供该方法的 bytes 或 bytearray 对象的内容将作为元素之间的分隔。

static bytes.**maketrans**(*from*, *to*)
static bytearray.**maketrans**(*from*, *to*)
此静态方法返回一个可用于*bytes.translate()* 的转换对照表，它将把 *from* 中的每个字符映射为 *to* 中相同位置上的字符；*from* 与 *to* 必须都是*字节类对象* 并且具有相同的长度。

3.1 新版功能.

bytes.**partition**(*sep*)
bytearray.**partition**(*sep*)
在 *sep* 首次出现的位置拆分序列，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身或其 bytearray 副本，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含原序列以及两个空的 bytes 或 bytearray 对象。

要搜索的分隔符可以是任意*bytes-like object*。

bytes.**replace**(*old*, *new*[, *count*])
bytearray.**replace**(*old*, *new*[, *count*])
返回序列的副本，其中出现的所有子序列 *old* 都将被替换为 *new*。如果给出了可选参数 *count*，则只替换前 *count* 次出现。

要搜索的子序列及其替换序列可以是任意*bytes-like object*。

注解： 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

---

bytes.**rfind**(*sub*[, *start*[, *end*]])

bytearray.**rfind**(*sub*[, *start*[, *end*]])
返回子序列 *sub* 在序列内被找到的最大（最右）索引，这样 *sub* 将包含在 `s[start:end]` 当中。可选参数 *start* 与 *end* 会被解读为切片表示法。如果未找到则返回 `-1`。

要搜索的子序列可以是任意*bytes-like object* 或是 0 至 255 范围内的整数。

在 3.3 版更改: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**rindex**(*sub*[, *start*[, *end*]])
bytearray.**rindex**(*sub*[, *start*[, *end*]])
类似于`rfind()`，但在子序列 *sub* 未找到时会引发`ValueError`。

要搜索的子序列可以是任意*bytes-like object* 或是 0 至 255 范围内的整数。

在 3.3 版更改: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**rpartition**(*sep*)
bytearray.**rpartition**(*sep*)
Split the sequence at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself or its bytearray copy, and the part after the separator. If the separator is not found, return a 3-tuple containing a copy of the original sequence, followed by two empty bytes or bytearray objects.

要搜索的分隔符可以是任意*bytes-like object*。

bytes.**startswith**(*prefix*[, *start*[, *end*]])
bytearray.**startswith**(*prefix*[, *start*[, *end*]])
如果二进制数据以指定的 *prefix* 开头则返回 `True`，否则返回 `False`。*prefix* 也可以为由多个供查找的前缀构成的元组。如果有可选项 *start*，将从所指定位置开始检查。如果有可选项 *end*，将在所指定位置停止比较。

要搜索的前缀可以是任意*bytes-like object*。

bytes.**translate**(*table*, *delete=b''*)
bytearray.**translate**(*table*, *delete=b''*)
返回原 bytes 或 bytearray 对象的副本，移除其中所有在可选参数 *delete* 中出现的 bytes，其余 bytes 将通过给定的转换表进行映射，该转换表必须是长度为 256 的 bytes 对象。

你可以使用`bytes.maketrans()` 方法来创建转换表。

对于仅需移除字符的转换，请将 *table* 参数设为 `None`:

```
>>> b'read this short text'.translate(None, b'aeiou')
b'rd ths shrt txt'
```

在 3.6 版更改: 现在支持将 *delete* 作为关键字参数。

以下 bytes 和 bytearray 对象的方法的默认行为会假定使用兼容 ASCII 的二进制格式，但通过传入适当的参数仍然可用于任意二进制数据。请注意本小节中所有的 bytearray 方法都 *不是* 原地执行操作，而是会产生新的对象。

bytes.**center**(*width*[, *fillbyte*])
bytearray.**center**(*width*[, *fillbyte*])
返回原对象的副本，在长度为 *width* 的序列内居中，使用指定的 *fillbyte* 填充两边的空位（默认使用 ASCII 空格符）。对于`bytes` 对象，如果 *width* 小于等于 `len(s)` 则返回原序列的副本。

注解: 此方法的 bytearray 版本 *并非* 原地操作——它总是产生一个新对象，即便没有做任何改变。

---

bytes.**ljust**(*width*[, *fillbyte*])

bytearray.**ljust**(*width*[, *fillbyte*])
返回原对象的副本，在长度为 *width* 的序列中靠左对齐。使用指定的 *fillbyte* 填充空位（默认使用 ASCII 空格符）。对于*bytes* 对象，如果 *width* 小于等于 `len(s)` 则返回原序列的副本。

> **注解：** 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**lstrip**([*chars*])
bytearray.**lstrip**([*chars*])
返回原序列的副本，移除指定的前导字节。*chars* 参数为指定要移除字节值集合的二进制序列——这个名称表明此方法通常是用于 ASCII 字符。如果省略或为 `None`，则 *chars* 参数默认移除 ASCII 空白符。*chars* 参数并非指定单个前缀；而是会移除参数值的所有组合:

```
>>> b'   spacious   '.lstrip()
b'spacious   '
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

要移除的字节值二进制序列可以是任意*bytes-like object*。

> **注解：** 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**rjust**(*width*[, *fillbyte*])
bytearray.**rjust**(*width*[, *fillbyte*])
返回原对象的副本，在长度为 *width* 的序列中靠右对齐。使用指定的 *fillbyte* 填充空位（默认使用 ASCII 空格符）。对于*bytes* 对象，如果 *width* 小于等于 `len(s)` 则返回原序列的副本。

> **注解：** 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**rsplit**(*sep=None*, *maxsplit=-1*)
bytearray.**rsplit**(*sep=None*, *maxsplit=-1*)
将二进制序列拆分为相同类型的子序列，使用 *sep* 作为分隔符。如果给出了 *maxsplit*，则最多进行 *maxsplit* 次拆分，从 最右边开始。如果 *sep* 未指定或为 `None`，任何只包含 ASCII 空白符的子序列都会被作为分隔符。除了从右边开始拆分，*rsplit()* 的其他行为都类似于下文所述的*split()*。

bytes.**rstrip**([*chars*])
bytearray.**rstrip**([*chars*])
返回原序列的副本，移除指定的末尾字节。*chars* 参数为指定要移除字节值集合的二进制序列——这个名称表明此方法通常是用于 ASCII 字符。如果省略或为 `None`，则 *chars* 参数默认移除 ASCII 空白符。*chars* 参数并非指定单个后缀；而是会移除参数值的所有组合:

```
>>> b'   spacious   '.rstrip()
b'   spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

要移除的字节值二进制序列可以是任意*bytes-like object*。

> **注解：** 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**split**(*sep=None*, *maxsplit=-1*)
bytearray.**split**(*sep=None*, *maxsplit=-1*)
将二进制序列拆分为相同类型的子序列，使用 *sep* 作为分隔符。如果给出了 *maxsplit* 且非负值，则最多

进行 *maxsplit* 次拆分（因此，列表最多会有 `maxsplit+1` 个元素）。如果 *maxsplit* 未指定或为 `-1`，则不限制拆分次数（进行所有可能的拆分）。

如果给出了 *sep*，则连续的分隔符不会被组合在一起而是被视为分隔空子序列 (例如 `b'1,,2'.split(b',')` 将返回 `[b'1', b'', b'2']`)。*sep* 参数可能为一个多字节序列 (例如 `b'1<>2<>3'.split(b'<>')` 将返回 `[b'1', b'2', b'3']`)。使用指定的分隔符拆分空序列将返回 `[b'']` 或 `[bytearray(b'')]`，具体取决于被拆分对象的类型。*sep* 参数可以是任意*bytes-like object*。

例如

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
```

如果 *sep* 未指定或为 `None`，则会应用另一种拆分算法：连续的 ASCII 空白符会被视为单个分隔符，其结果将不包含序列开头或末尾的空白符。因此，在不指定分隔符的情况下对空序列或仅包含 ASCII 空白符的序列进行拆分将返回 `[]`。

例如

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'   1   2   3   '.split()
[b'1', b'2', b'3']
```

bytes.**strip**([*chars*])
bytearray.**strip**([*chars*])

返回原序列的副本，移除指定的开头和末尾字节。*chars* 参数为指定要移除字节值集合的二进制序列——这个名称表明此方法通常是用于 ASCII 字符。如果省略或为 `None`，则 *chars* 参数默认移除 ASCII 空白符。*chars* 参数并非指定单个前缀或后缀；而是会移除参数值的所有组合:

```
>>> b'   spacious   '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

要移除的字节值二进制序列可以是任意*bytes-like object*。

注解： 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

以下 bytes 和 bytearray 对象的方法会假定使用兼容 ASCII 的二进制格式，不应当被应用于任意二进制数据。请注意本小节中所有的 bytearray 方法都 不是原地执行操作，而是会产生新的对象。

bytes.**capitalize**()
bytearray.**capitalize**()

返回原序列的副本，其中每个字节将都将被解读为一个 ASCII 字符，并且第一个字节的字符大写而其余的小写。非 ASCII 字节值将保持原样不变。

注解： 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**expandtabs**(*tabsize=8*)

bytearray.**expandtabs**(*tabsize=8*)

返回序列的副本，其中所有的 ASCII 制表符会由一个或多个 ASCII 空格替换，具体取决于当前列位置和给定的制表符宽度。每 *tabsize* 个字节设为一个制表位（默认值 8 时设定的制表位在列 0, 8, 16 依次类推）。要展开序列，当前列位置将被设为零并逐一检查序列中的每个字节。如果字节为 ASCII 制表符 (`b'\t'`)，则并在结果中插入一个或多个空格符，直到当前列等于下一个制表位。（制表符本身不会被复制。）如果当前字节为 ASCII 换行符 (`b'\n'`) 或回车符 (`b'\r'`)，它会被复制并将当前列重设为零。任何其他字节会被不加修改地复制并将当前列加一，不论该字节值在被打印时会如何显示:

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012     0123    01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012 0123    01234'
```

注解: 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**isalnum**()
bytearray.**isalnum**()

Return true if all bytes in the sequence are alphabetical ASCII characters or ASCII decimal digits and the sequence is not empty, false otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

例如

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

bytes.**isalpha**()
bytearray.**isalpha**()

Return true if all bytes in the sequence are alphabetic ASCII characters and the sequence is not empty, false otherwise. Alphabetic ASCII characters are those byte values in the sequence `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'`.

例如

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

bytes.**isdigit**()
bytearray.**isdigit**()

Return true if all bytes in the sequence are ASCII decimal digits and the sequence is not empty, false otherwise. ASCII decimal digits are those byte values in the sequence `b'0123456789'`.

例如

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

bytes.**islower**()

bytearray.**islower**()
Return true if there is at least one lowercase ASCII character in the sequence and no uppercase ASCII characters, false otherwise.

例如

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

小写 ASCII 字符就是字节值包含在序列 b'abcdefghijklmnopqrstuvwxyz' 中的字符。大写 ASCII 字符就是字节值包含在序列 b'ABCDEFGHIJKLMNOPQRSTUVWXYZ' 中的字符。

bytes.**isspace**()
bytearray.**isspace**()
Return true if all bytes in the sequence are ASCII whitespace and the sequence is not empty, false otherwise. ASCII whitespace characters are those byte values in the sequence b' \t\n\r\x0b\f' (space, tab, newline, carriage return, vertical tab, form feed).

bytes.**istitle**()
bytearray.**istitle**()
Return true if the sequence is ASCII titlecase and the sequence is not empty, false otherwise. See *bytes.title()* for more details on the definition of "titlecase".

例如

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

bytes.**isupper**()
bytearray.**isupper**()
Return true if there is at least one uppercase alphabetic ASCII character in the sequence and no lowercase ASCII characters, false otherwise.

例如

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

小写 ASCII 字符就是字节值包含在序列 b'abcdefghijklmnopqrstuvwxyz' 中的字符。大写 ASCII 字符就是字节值包含在序列 b'ABCDEFGHIJKLMNOPQRSTUVWXYZ' 中的字符。

bytes.**lower**()
bytearray.**lower**()
返回原序列的副本，其所有大写 ASCII 字符均转换为对应的小写形式。

例如

```
>>> b'Hello World'.lower()
b'hello world'
```

小写 ASCII 字符就是字节值包含在序列 b'abcdefghijklmnopqrstuvwxyz' 中的字符。大写 ASCII 字符就是字节值包含在序列 b'ABCDEFGHIJKLMNOPQRSTUVWXYZ' 中的字符。

注解： 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**splitlines**(*keepends=False*)
bytearray.**splitlines**(*keepends=False*)
返回由原二进制序列中各行组成的列表，在 ASCII 行边界符的位置拆分。此方法使用*universal newlines*方式来分行。结果列表中不包含换行符，除非给出了 *keepends* 且为真值。

例如

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

不同于*split()*，当给出了分隔符 *sep* 时，对于空字符串此方法将返回一个空列表，而末尾的换行不会令结果中增加额外的行:

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

bytes.**swapcase**()
bytearray.**swapcase**()
返回原序列的副本，其所有小写 ASCII 字符均转换为对应的大写形式，反之亦反。

例如

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

不同于*str.swapcase()*，在些二进制版本下 `bin.swapcase().swapcase() == bin` 总是成立。大小写转换在 ASCII 中是对称的，即使其对于任意 Unicode 码位来说并不总是成立。

注解： 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**title**()
bytearray.**title**()
返回原二进制序列的标题版本，其中每个单词以一个大写 ASCII 字符为开头，其余字母为小写。不区别大小写的字节值将保持原样不变。

例如

```
>>> b'Hello world'.title()
b'Hello World'
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。所有其他字节值都不区分大小写。

该算法使用一种简单的与语言无关的定义，将连续的字母组合视为单词。该定义在多数情况下都很有效，但它也意味着代表缩写形式与所有格的撇号也会成为单词边界，这可能导致不希望的结果:

```
>>> b"they're bill's friends from the UK".title()
b"They're Bill'S Friends From The Uk"
```

可以使用正则表达式来构建针对撇号的特别处理:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1].upper() +
...                              mo.group(0)[1:].lower(),
...                   s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

注解: 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**upper**()
bytearray.**upper**()

返回原序列的副本，其所有小写 ASCII 字符均转换为对应的大写形式。

例如

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

注解: 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

bytes.**zfill**(*width*)
bytearray.**zfill**(*width*)

返回原序列的副本，在左边填充 `b'0'` 数码使序列长度为 *width*。正负值前缀 (`b'+'`/`b'-'`) 的处理方式是在正负符号 之后 填充而非在之前。对于 *bytes* 对象，如果 *width* 小于等于 `len(seq)` 则返回原序列。

例如

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

注解: 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

4.8.4 printf 风格的字节串格式化

注解： 此处介绍的格式化操作具有多种怪异特性，可能导致许多常见错误（例如无法正确显示元组和字典）。如果要打印的值可能为元组或字典，请将其放入一个元组中。

字节串对象 (bytes/bytearray) 具有一种特殊的内置操作：使用 % (取模) 运算符。这也被称为字节串的 *格式化* 或 *插值* 运算符。对于 `format % values` (其中 *format* 为一个字节串对象)，在 *format* 中的 % 转换标记符将被替换为零个或多个 *values* 条目。其效果类似于在 C 语言中使用 `sprintf()`。

如果 *format* 要求一个单独参数，则 *values* 可以为一个非元组对象。[5] 否则的话，*values* 必须或是是一个包含项数与格式字节串对象中指定的转换符项数相同的元组，或者是一个单独的映射对象（例如元组）。

转换标记符包含两个或更多字符并具有以下组成，且必须遵循此处规定的顺序：

1. `'%'` 字符，用于标记转换符的起始。
2. 映射键（可选），由加圆括号的字符序列组成 (例如 `(somename)`)。
3. 转换旗标（可选），用于影响某些转换类型的结果。
4. 最小字段宽度（可选）。如果指定为 `'*'` (星号)，则实际宽度会从 *values* 元组的下一元素中读取，要转换的对象则为最小字段宽度和可选的精度之后的元素。
5. 精度（可选），以在 `'.'` (点号) 之后加精度值的形式给出。如果指定为 `'*'` (星号)，则实际精度会从 *values* 元组的下一元素中读取，要转换的对象则为精度之后的元素。
6. 长度修饰符（可选）。
7. 转换类型。

当右边的参数为一个字典（或其他映射类型）时，字节串对象中的格式 *必须* 包含加圆括号的映射键，对应 `'%'` 字符之后字典中的每一项。映射键将从映射中选取要格式化的值。例如：

```
>>> print(b'%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

在此情况下格式中不能出现 * 标记符（因其需要一个序列类的参数列表）。

转换旗标为：

标志	含义
`'#'`	值的转换将使用“替代形式”（具体定义见下文）。
`'0'`	转换将为数字值填充零字符。
`'-'`	转换值将靠左对齐（如果同时给出 `'0'` 转换，则会覆盖后者）。
`' '`	(空格) 符号位转换产生的正数（或空字符串）前将留出一个空格。
`'+'`	符号字符 (`'+'` 或 `'-'`) 将显示于转换结果的开头（会覆盖“空格”旗标）。

可以给出长度修饰符 (h, l 或 L)，但会被忽略，因为对 Python 来说没有必要–所以 `%ld` 等价于 `%d`。

转换类型为：

转换符	含义	注释
'd'	有符号十进制整数。	
'i'	有符号十进制整数。	
'o'	有符号八进制数。	(1)
'u'	过时类型–等价于 'd'。	(8)
'x'	有符号十六进制数（小写）。	(2)
'X'	有符号十六进制数（大写）。	(2)
'e'	浮点指数格式（小写）。	(3)
'E'	浮点指数格式（大写）。	(3)
'f'	浮点十进制格式。	(3)
'F'	浮点十进制格式。	(3)
'g'	浮点格式。如果指数小于 -4 或不小于精度则使用小写指数格式，否则使用十进制格式。	(4)
'G'	浮点格式。如果指数小于 -4 或不小于精度则使用大写指数格式，否则使用十进制格式。	(4)
'c'	单个字节（接受整数或单个字节对象）。	
'b'	字节串（任何遵循 缓冲区协议或是具有 __bytes__() 的对象）。	(5)
's'	's' 是 'b' 的一个别名，只应当在基于 Python2/3 的代码中使用。	(6)
'a'	字节串(使用 repr(obj).encode('ascii','backslashreplace) 转换任何 Python 对象)。	(5)
'r'	'r' 是 'a' 的一个别名，只应当在基于 Python2/3 的代码中使用。	(7)
'%'	不转换参数，在结果中输出一个 '%' 字符。	

注释:

(1) 此替代形式会在第一个数码之前插入标示八进制数的前缀 ('0o')。

(2) 此替代形式会在第一个数码之前插入 '0x' 或 '0X' 前缀（取决于是使用 'x' 还是 'X' 格式）。

(3) 此替代形式总是会在结果中包含一个小数点，即使其后并没有数码。

小数点后的数码位数由精度决定，默认为 6。

(4) 此替代形式总是会在结果中包含一个小数点，末尾各位的零不会如其他情况下那样被移除。

小数点前后的有效数码位数由精度决定，默认为 6。

(5) 如果精度为 N，输出将截短为 N 个字符。

(6) b'%s' 已弃用，但在 3.x 系列中将不会被移除。

(7) b'%r' 已弃用，但在 3.x 系列中将不会被移除。

(8) 参见 **PEP 237**。

注解： 此方法的 bytearray 版本 并非原地操作——它总是产生一个新对象，即便没有做任何改变。

参见:

PEP 461 - 为 bytes 和 bytearray 添加% 格式化

3.5 新版功能.

4.8.5 内存视图

memoryview 对象允许 Python 代码访问一个对象的内部数据，只要该对象支持 缓冲区协议而无需进行拷贝。

class memoryview(*obj*)

创建一个引用 *obj* 的*memoryview*。*obj* 必须支持缓冲区协议。支持缓冲区协议的内置对象包括*bytes*和*bytearray*。

memoryview 具有 *元素* 的概念，即由原始对象 *obj* 所处理的基本内存单元。对于许多简单类型例如*bytes* 和*bytearray* 来说，一个元素就是一个字节，但是其他的类型例如*array.array* 可能有更大的元素。

`len(view)` 与*tolist* 的长度相等。如果 `view.ndim = 0`，则其长度为 1。如果 `view.ndim = 1`，则其长度等于 view 中元素的数量。对于更高的维度，其长度等于表示 view 的嵌套列表的长度。*itemsize* 属性可向你给出单个元素所占的字节数。

memoryview 支持通过切片和索引访问其元素。一维切片的结果将是一个子视图:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

如果*format* 是一个来自于*struct* 模块的原生格式说明符，则也支持使用整数或由整数构成的元组进行索引，并返回具有正确类型的单个 *元素*。一维内存视图可以使用一个整数或由一个整数构成的元组进行索引。多维内存视图可以使用由恰好 *ndim* 个整数构成的元素进行索引，*ndim* 即其维度。零维内存视图可以使用空元组进行索引。

这里是一个使用非字节格式的例子:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

如果下层对象是可写的，则内存视图支持一维切片赋值。改变大小则不被允许:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
```

(下页继续)

(续上页)

```
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different structures
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

由带有格式符号‘B’，‘b’或‘c’的可哈希（只读）类型构成的一维内存视图同样是可哈希的。哈希定义为 `hash(m) == hash(m.tobytes())`：

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

在 3.3 版更改: 一维内存视图现在可以被切片。带有格式符号‘B’，‘b’或‘c’的一维内存视图现在是可哈希的。

在 3.4 版更改: 内存视图现在会自动注册为*collections.abc.Sequence*

在 3.5 版更改: 内存视图现在可使用整数元组进行索引。

memoryview 具有以下一些方法：

__eq__(*exporter*)

memoryview 与 **PEP 3118** 中的导出器这两者如果形状相同，并且如果当使用*struct* 语法解读操作数的相应格式代码时所有对应值都相同，则它们就是等价的。

对于*tolist()* 当前所支持的*struct* 格式字符串子集，如果 `v.tolist() == w.tolist()` 则 `v` 和 `w` 相等：

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

如果两边的格式字符串都不被*struct* 模块所支持，则两对象比较结果总是不相等（即使格式字符串和缓冲区内容相同）：

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
```

(下页继续)

(续上页)

```
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

请注意，与浮点数的情况一样，对于内存视图对象来说，`v is w` 也 并不意味着 `v == w`。

在 3.3 版更改: 之前的版本比较原始内存时会忽略条目的格式与逻辑数组结构。

tobytes()
将缓冲区中的数据作为字节串返回。这相当于在内存视图上调用*bytes* 构造器。

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

对于非连续数组，结果等于平面化表示的列表，其中所有元素都转换为字节串。*tobytes()* 支持所有格式字符串，不符合*struct* 模块语法的那些也包括在内。

hex()
返回一个字符串对象，其中分别以两个十六进制数码表示缓冲区里的每个字节。

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

3.5 新版功能.

tolist()
将缓冲区内的数据以一个元素列表的形式返回。

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

在 3.3 版更改: *tolist()* 现在支持*struct* 模块语法中的所有单字符原生格式以及多维表示形式。

release()
释放由内存视图对象所公开的底层缓冲区。许多对象在被视图所获取时都会采取特殊动作（例如，*bytearray* 将会暂时禁止调整大小）；因此，调用 release() 可以方便地尽早去除这些限制（并释放任何多余的资源）。

在此方法被调用后，任何对视图的进一步操作将引发*ValueError*（*release()* 本身除外，它可以被多次调用）:

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
```

(下页继续)

(续上页)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

使用 `with` 语句，可以通过上下文管理协议达到类似的效果:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

3.2 新版功能.

cast (*format*[, *shape*])

将内存视图转化为新的格式或形状。*shape* 默认为 `[byte_length//new_itemsize]`，这意味着结果视图将是一维的。返回值是一个新的内存视图，但缓冲区本身不会被复制。支持的转化有 1D -> C-*contiguous* 和 C-contiguous -> 1D。

目标格式仅限于 *struct* 语法中的单一元素原生格式。其中一种格式必须为字节格式（‘B’，‘b’或‘c’）。结果的字节长度必须与原始长度相同。

将 1D/long 转换为 1D/unsigned bytes:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

将 1D/unsigned bytes 转换为 1D/char:

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
```

(下页继续)

(续上页)

```
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

将 1D/bytes 转换为 3D/ints 再转换为 1D/signed char:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

Cast 1D/unsigned char to 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

3.3 新版功能.

在 3.5 版更改: 当转换为字节视图时，源格式将不再受限。

还存在一些可用的只读属性:

obj

内存视图的下层对象:

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

3.3 新版功能.

nbytes

nbytes == product(shape) * itemsize == len(m.tobytes())。这是数组在连续表示时将会占用的空间总字节数。它不一定等于 len(m):

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

多维数组:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0, 16.5]]
>>> len(y)
3
>>> y.nbytes
96
```

3.3 新版功能.

readonly

一个表明内存是否只读的布尔值。

format

一个字符串，包含视图中每个元素的格式（表示为*struct* 模块样式）。内存视图可以从具有任意格式字符串的导出器创建，但某些方法 (例如*tolist()*) 仅限于原生的单元素格式。

在 3.3 版更改: 格式 'B' 现在会按照 struct 模块语法来处理。这意味着 memoryview(b'abc')[0] == b'abc'[0] == 97。

itemsize

memoryview 中每个元素以字节表示的大小:

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

一个整数，表示内存所代表的多维数组具有多少个维度。

shape
一个整数元组，通过*ndim* 的长度值给出内存所代表的 N 维数组的形状。

在 3.3 版更改: 当 ndim = 0 时值为空元组而不再为 None。

strides
一个整数元组，通过*ndim* 的长度给出以字节表示的大小，以便访问数组中每个维度上的每个元素。

在 3.3 版更改: 当 ndim = 0 时值为空元组而不再为 None。

suboffsets
供 PIL 风格的数组内部使用。该值仅作为参考信息。

c_contiguous
一个表明内存是否为 C-*contiguous* 的布尔值。

3.3 新版功能.

f_contiguous
一个表明内存是否为 Fortran *contiguous* 的布尔值。

3.3 新版功能.

contiguous
一个表明内存是否为*contiguous* 的布尔值。

3.3 新版功能.

4.9 集合类型—set, frozenset

set 对象是由具有唯一性的*hashable* 对象所组成的无序多项集。常见的用途包括成员检测、从序列中去除重复项以及数学中的集合类计算，例如交集、并集、差集与对称差集等等。(关于其他容器对象请参看*dict*, *list* 与*tuple* 等内置类，以及*collections* 模块。)

与其他多项集一样，集合也支持 x in set, len(set) 和 for x in set。作为一种无序的多项集，集合并不记录元素位置或插入顺序。相应地，集合不支持索引、切片或其他序列类的操作。

目前有两种内置集合类型，*set* 和*frozenset*。*set* 类型是可变的—其内容可以使用 add() 和 remove() 这样的方法来改变。由于是可变类型，它没有哈希值，且不能被用作字典的键或其他集合的元素。*frozenset* 类型是不可变并且为*hashable* —其内容在被创建后不能再改变；因此它可以被用作字典的键或其他集合的元素。

除了可以使用*set* 构造器，非空的 set (不是 frozenset) 还可以通过将以逗号分隔的元素列表包含于花括号之内来创建，例如: {'jack', 'sjoerd'}。

两个类的构造器具有相同的作用方式:

class set([*iterable*])
class frozenset([*iterable*])
返回一个新的 set 或 frozenset 对象，其元素来自于 *iterable*。集合的元素必须为*hashable*。要表示由集合对象构成的集合，所有的内层集合必须为*frozenset* 对象。如果未指定 *iterable*，则将返回一个新的空集合。

set 和*frozenset* 的实例提供以下操作:

len(s)
返回集合 *s* 中的元素数量（即 *s* 的基数)。

x in s
检测 *x* 是否为 *s* 中的成员。

x not in s
检测 *x* 是否非 *s* 中的成员。

isdisjoint(*other*)
如果集合中没有与 *other* 共有的元素则返回 `True`。当且仅当两个集合的交集为空集合时，两者为不相交集合。

issubset(*other*)
set <= other
检测是否集合中的每个元素都在 *other* 之中。

set < other
检测集合是否为 *other* 的真子集，即 `set <= other and set != other`。

issuperset(*other*)
set >= other
检测是否 *other* 中的每个元素都在集合之中。

set > other
检测集合是否为 *other* 的真超集，即 `set >= other and set != other`。

union(**others*)
set | other | ...
返回一个新集合，其中包含来自原集合以及 others 指定的所有集合中的元素。

intersection(**others*)
set & other & ...
返回一个新集合，其中包含原集合以及 others 指定的所有集合中共有的元素。

difference(**others*)
set - other - ...
返回一个新集合，其中包含原集合中在 others 指定的其他集合中不存在的元素。

symmetric_difference(*other*)
set ^ other
返回一个新集合，其中的元素或属于原集合或属于 *other* 指定的其他集合，但不能同时属于两者。

copy()
Return a new set with a shallow copy of *s*.

请注意，非运算符版本的*union()*，*intersection()*，*difference()*，以及*symmetric_difference()*，*issubset()* 和*issuperset()* 方法会接受任意可迭代对象作为参数。相比之下，它们所对应的运算符版本则要求其参数为集合。这就排除了容易出错的构造形式例如 `set('abc') & 'cbs'`，而推荐可读性更强的 `set('abc').intersection('cbs')`。

set 和*frozenset* 均支持集合与集合的比较。两个集合当且仅当每个集合中的每个元素均包含于另一个集合之内（即各为对方的子集）时则相等。一个集合当且仅当其为另一个集合的真子集（即为后者的子集但两者不相等）时则小于另一个集合。一个集合当且仅当其为另一个集合的真超集（即为后者的超集但两者不相等）时则大于另一个集合。

set 的实例与*frozenset* 的实例之间基于它们的成员进行比较。例如 `set('abc') == frozenset('abc')` 返回 `True`，`set('abc') in set([frozenset('abc')])` 也一样。

子集与相等比较并不能推广为完全排序函数。例如，任意两个非空且不相交的集合不相等且互不为对方的子集，因此以下 所有比较均返回 `False`: `a<b`, `a==b`, or `a>b`。

由于集合仅定义了部分排序（子集关系），因此由集合构成的列表*list.sort()* 方法的输出并无定义。

集合的元素，与字典的键类似，必须为*hashable*。

混合了*set* 实例与*frozenset* 的二进制位运算将返回与第一个操作数相同的类型。例如: `frozenset('ab') | set('bc')` 将返回*frozenset* 的实例。

下表列出了可用于*set* 而不能用于不可变的*frozenset* 实例的操作：

update(*others*)
set |= other | ...
更新集合，添加来自 others 中的所有元素。

intersection_update(*others*)
set &= other & ...
更新集合，只保留其中在所有 others 中也存在的元素。

difference_update(*others*)
set -= other | ...
更新集合，移除其中也存在于 others 中的元素。

symmetric_difference_update(*other*)
set ^= other
更新集合，只保留存在于集合的一方而非共同存在的元素。

add(*elem*)
将元素 *elem* 添加到集合中。

remove(*elem*)
从集合中移除元素 *elem*。如果 *elem* 不存在于集合中则会引发*KeyError*。

discard(*elem*)
如果元素 *elem* 存在于集合中则将其移除。

pop()
从集合中移除并返回任意一个元素。如果集合为空则会引发*KeyError*。

clear()
从集合中移除所有元素。

请注意，非运算符版本的*update()*，*intersection_update()*，*difference_update()* 和*symmetric_difference_update()* 方法将接受任意可迭代对象作为参数。

请注意，`__contains__()`，*remove()* 和*discard()* 方法的 *elem* 参数可能是一个 set。为支持对一个等价的 frozenset 进行搜索，会根据 *elem* 临时创建一个该类型对象。

4.10 映射类型—dict

mapping 对象会将*hashable* 值映射到任意对象。映射属于可变对象。目前仅有一种标准映射类型 字典。（关于其他容器对象请参看*list*，*set* 与*tuple* 等内置类，以及*collections* 模块。）

字典的键 几乎可以是任何值。非*hashable* 的值，即包含列表、字典或其他可变类型的值（此类对象基于值而非对象标识进行比较）不可用作键。数字类型用作键时遵循数字比较的一般规则：如果两个数值相等 (例如 `1` 和 `1.0`) 则两者可以被用来索引同一字典条目。（但是请注意，由于计算机对于浮点数存储的只是近似值，因此将其用作字典键是不明智的。）

字典可以通过将以逗号分隔的 键：值 对列表包含于花括号之内来创建，例如：`{'jack': 4098, 'sjoerd': 4127}` 或 `{4098: 'jack', 4127: 'sjoerd'}`，也可以通过*dict* 构造器来创建。

class dict(***kwarg*)
class dict(*mapping*, ***kwarg*)
class dict(*iterable*, ***kwarg*)
返回一个新的字典，基于可选的位置参数和可能为空的关键字参数集来初始化。

如果没有给出位置参数，将创建一个空字典。如果给出一个位置参数并且其属于映射对象，将创建一个具有与映射对象相同键值对的字典。否则的话，位置参数必须为一个*iterable* 对象。该可迭代对象中的每一项本身必须为一个刚好包含两个元素的可迭代对象。每一项中的第一个对象将成为新字典的一

个键，第二个对象将成为其对应的值。如果一个键出现一次以上，该键的最后一个值将成为其在新字典中对应的值。

如果给出了关键字参数，则关键字参数及其值会被加入到基于位置参数创建的字典。如果要加入的键已存在，来自关键字参数的值将替代来自位置参数的值。

作为演示，以下示例返回的字典均等于 `{"one": 1, "two": 2, "three": 3}`:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

像第一个例子那样提供关键字参数的方式只能使用有效的 Python 标识符作为键。其他方式则可使用任何有效的键。

这些是字典所支持的操作（因而自定义的映射类型也应当支持）:

len(d)
返回字典 *d* 中的项数。

d[key]
返回 *d* 中以 *key* 为键的项。如果映射中不存在 *key* 则会引发*KeyError*。

如果字典的子类定义了方法 `__missing__()` 并且 *key* 不存在，则 `d[key]` 操作将调用该方法并附带键 *key* 作为参数。`d[key]` 随后将返回或引发 `__missing__(key)` 调用所返回或引发的任何对象或异常。没有其他操作或方法会发起调用 `__missing__()`。如果未定义 `__missing__()`，则会引发*KeyError*。`__missing__()` 必须是一个方法；它不能是一个实例变量:

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

上面的例子显示了*collections.Counter* 实现的部分代码。还有另一个不同的 `__missing__` 方法是由*collections.defaultdict* 所使用的。

d[key] = value
将 `d[key]` 设为 *value*。

del d[key]
将 `d[key]` 从 *d* 中移除。如果映射中不存在 *key* 则会引发*KeyError*。

key in d
如果 *d* 中存在键 *key* 则返回 `True`，否则返回 `False`。

key not in d
等价于 `not key in d`。

iter(d)
返回以字典的键为元素的迭代器。这是 `iter(d.keys())` 的快捷方式。

clear()
移除字典中的所有元素。

copy()
返回原字典的浅拷贝。

classmethod fromkeys(*seq*[, *value*])
Create a new dictionary with keys from *seq* and values set to *value*.

fromkeys() 属于类方法，会返回一个新字典。*value* 默认为 None。

get(*key*[, *default*])
如果 *key* 存在于字典中则返回 *key* 的值，否则返回 *default*。如果 *default* 未给出则默认为 None，因而此方法绝不会引发*KeyError*。

items()
返回由字典项 ((键, 值) 对) 组成的一个新视图。参见视图对象文档。

keys()
返回由字典键组成的一个新视图。参见视图对象文档。

pop(*key*[, *default*])
如果 *key* 存在于字典中则将其移除并返回其值，否则返回 *default*。如果 *default* 未给出且 *key* 不存在于字典中，则会引发*KeyError*。

popitem()
Remove and return an arbitrary (key, value) pair from the dictionary.

popitem() 适用于对字典进行消耗性的迭代，这在集合算法中经常被使用。如果字典为空，调用*popitem()* 将引发*KeyError*。

setdefault(*key*[, *default*])
如果字典存在键 *key*，返回它的值。如果不存在，插入值为 *default* 的键 *key*，并返回 *default*。*default* 默认为 None。

update([*other*])
使用来自 *other* 的键/值对更新字典，覆盖原有的键。返回 None。

update() 接受另一个字典对象，或者一个包含键/值对（以长度为二的元组或其他可迭代对象表示）的可迭代对象。如果给出了关键字参数，则会以其所指定的键/值对更新字典：d.update(red=1, blue=2)。

values()
返回由字典值组成的一个新视图。参见视图对象文档。

Dictionaries compare equal if and only if they have the same (key, value) pairs. Order comparisons ('<', '<=', '>=', '>') raise *TypeError*.

参见:

types.MappingProxyType 可被用来创建一个*dict* 的只读视图。

4.10.1 字典视图对象

由*dict.keys()*, *dict.values()* 和*dict.items()* 所返回的对象是 视图对象。该对象提供字典条目的一个动态视图，这意味着当字典改变时，视图也会相应改变。

字典视图可以被迭代以产生与其对应的数据，并支持成员检测：

len(dictview)
返回字典中的条目数。

iter(dictview)
返回字典中的键、值或项（以（键, 值）为元素的元组表示）的迭代器。

Keys and values are iterated over in an arbitrary order which is non-random, varies across Python implementations, and depends on the dictionary's history of insertions and deletions. If keys, values and items views are iterated over with no intervening modifications to the dictionary, the order of items will directly correspond. This allows the creation of `(value, key)` pairs using *zip()*: `pairs = zip(d.values(), d.keys())`. Another way to create the same list is `pairs = [(v, k) for (k, v) in d.items()]`.

在添加或删除字典中的条目期间对视图进行迭代可能引发*RuntimeError* 或者无法完全迭代所有条目。

x in dictview

如果 *x* 是对应字典中存在的键、值或项（在最后一种情况下 *x* 应为一个（键, 值）元组）则返回 `True`。

键视图类似于集合，因为其条目不重复且可哈希。如果所有值都是可哈希的，即（键, 值）对也是不重复且可哈希的，那么条目视图也会类似于集合。（值视图则不被视为类似于集合，因其条目通常都是有重复的。）对于类似于集合的视图，为抽象基类*collections.abc.Set* 所定义的全部操作都是有效的 (例如 ==, < 或 ^)。

一个使用字典视图的示例:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order
>>> list(keys)
['eggs', 'bacon', 'sausage', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['spam', 'bacon']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

4.11 上下文管理器类型

Python 的 `with` 语句支持通过上下文管理器所定义的运行时上下文这一概念。此对象的实现使用了一对专门方法，允许用户自定义类来定义运行时上下文，在语句体被执行前进入该上下文，并在语句执行完毕时退出该上下文：

`contextmanager.__enter__()`
Enter the runtime context and return either this object or another object related to the runtime context. The value returned by this method is bound to the identifier in the `as` clause of `with` statements using this context manager.

一个返回其自身的上下文管理器的例子是*file object*。文件对象会从 __enter__() 返回其自身，以允许*open()* 被用作 `with` 语句中的上下文表达式。

An example of a context manager that returns a related object is the one returned by *decimal.localcontext()*. These managers set the active decimal context to a copy of the original decimal context and then return the copy. This allows changes to be made to the current decimal context in the body of the `with` statement without affecting code outside the `with` statement.

`contextmanager.__exit__`(*exc_type*, *exc_val*, *exc_tb*)
退出运行时上下文并返回一个布尔值旗标来表明所发生的任何异常是否应当被屏蔽。如果在执行 `with` 语句的语句体期间发生了异常，则参数会包含异常的类型、值以及回溯信息。在其他情况下三个参数均为 `None`。

Returning a true value from this method will cause the `with` statement to suppress the exception and continue execution with the statement immediately following the `with` statement. Otherwise the exception continues propagating after this method has finished executing. Exceptions that occur during execution of this method will replace any exception that occurred in the body of the `with` statement.

传入的异常绝对不应当被显式地重新引发——相反地，此方法应当返回一个假值以表明方法已成功完成并且不希望屏蔽被引发的异常。这允许上下文管理代码方便地检测*__exit__()* 方法是否确实已失败。

Python 定义了一些上下文管理器来支持简易的线程同步、文件或其他对象的快速关闭，以及更方便地操作活动的十进制算术上下文。除了实现上下文管理协议以外，不同类型不会被特殊处理。请参阅*contextlib* 模块查看相关的示例。

Python 的*generator* 和*contextlib.contextmanager* 装饰器提供了实现这些协议的便捷方式。如果使用*contextlib.contextmanager* 装饰器来装饰一个生成器函数，它将返回一个实现了必要的 `__enter__()` and `__exit__()` 方法的上下文管理器，而不再是由未经装饰的生成器函数所产生的迭代器。

请注意，Python/C API 中 Python 对象的类型结构中并没有针对这些方法的专门槽位。想要定义这些方法的扩展类型必须将它们作为普通的 Python 可访问方法来提供。与设置运行时上下文的开销相比，单个类字典查找的开销可以忽略不计。

4.12 其他内置类型

解释器支持一些其他种类的对象。这些对象大都仅支持一两种操作。

4.12.1 模块

模块唯一的特殊操作是属性访问: `m.name`，这里 *m* 为一个模块而 *name* 访问定义在 *m* 的符号表中的一个名称。模块属性可以被赋值。(请注意 `import` 语句严格来说也是对模块对象的一种操作；`import foo` 不要求存在一个名为 *foo* 的模块对象，而是要求存在一个对于名为 *foo* 的模块的 (永久性) 定义。)

每个模块都有一个特殊属性*__dict__*。这是包含模块的符号表的字典。修改此字典将实际改变模块的符号表，但是无法直接对*__dict__* 赋值 (你可以写 `m.__dict__['a'] = 1`，这会将 `m.a` 定义为 `1`，但是你不能写 `m.__dict__ = {}`)。不建议直接修改*__dict__*。

内置于解释器中的模块会写成这样: `<module 'sys' (built-in)>`。如果是从一个文件加载，则会写成 `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`。

4.12.2 类与类实例

关于这些类型请参阅 objects 和 class。

4.12.3 函数

函数对象是通过函数定义创建的。对函数对象的唯一操作是调用它: `func(argument-list)`。

实际上存在两种不同的函数对象：内置函数和用户自定义函数。两者支持同样的操作（调用函数），但实现方式不同，因此对象类型也不同。

更多信息请参阅 function。

4.12.4 方法

方法是使用属性表示法来调用的函数。存在两种形式：内置方法（例如列表的 `append()` 方法）和类实例方法。内置方法由支持它们的类型来描述。

如果你通过一个实例来访问方法（即定义在类命名空间内的函数），你会得到一个特殊对象: *绑定方法* (或称 *实例方法*) 对象。当被调用时，它会将 `self` 参数添加到参数列表。绑定方法具有两个特殊的只读属性: `m.__self__` 操作该方法的对象，而 `m.__func__` 是实现该方法的函数。调用 `m(arg-1, arg-2, ..., arg-n)` 完全等价于调用 `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`。

与函数对象类似，绑定方法对象也支持获取任意属性。但是，由于方法属性实际上保存于下层的函数对象中 (`meth.__func__`)，因此不允许设置绑定方法的方法属性。尝试设置方法的属性将会导致引发*AttributeError*。想要设置方法属性，你必须在下层的函数对象中显式地对其进行设置:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

更多信息请参阅 types。

4.12.5 代码对象

代码对象被具体实现用来表示“伪编译”的可执行 Python 代码，例如一个函数体。它们不同于函数对象，因为它们不包含对其全局执行环境的引用。代码对象由内置的*compile()* 函数返回，并可通过从函数对象的__code__ 属性从中提取。另请参阅*code* 模块。

可以通过将代码对象（而非源码字符串）传给*exec()* 或*eval()* 内置函数来执行或求值。

更多信息请参阅 types。

4.12.6 类型对象

类型对象表示各种对象类型。对象的类型可通过内置函数*type()* 来获取。类型没有特殊的操作。标准库模块*types* 定义了所有标准内置类型的名称。

类型以这样的写法来表示: `<class 'int'>`。

4.12.7 空对象

此对象会由不显式地返回值的函数所返回。它不支持任何特殊的操作。空对象只有一种值 `None` (这是个内置名称)。`type(None)()` 会生成同一个单例。

该对象的写法为 `None`。

4.12.8 省略符对象

此对象常被用于切片 (参见 slicings)。它不支持任何特殊的操作。省略符对象只有一种值*Ellipsis* (这是个内置名称)。`type(Ellipsis)()` 会生成*Ellipsis* 单例。

该对象的写法为 `Ellipsis` 或 `...`。

4.12.9 未实现对象

此对象会被作为比较和二元运算被应用于它们所不支持的类型时的返回值。请参阅 comparisons 了解更多信息。未实现对象只有一种值 `NotImplemented`。`type(NotImplemented)()` 会生成这个单例。

该对象的写法为 `NotImplemented`。

4.12.10 布尔值

布尔值是两个常量对象 `False` 和 `True`。它们被用来表示逻辑上的真假（不过其他值也可被当作真值或假值）。在数字类的上下文中（例如被用作算术运算符的参数时），它们的行为分别类似于整数 0 和 1。内置函数*bool()* 可被用来将任意值转换为布尔值，只要该值可被解析为一个逻辑值（参见之前的逻辑值检测 部分）。

该对象的写法分别为 `False` 和 `True`。

4.12.11 内部对象

有关此对象的信息请参阅 types。其中描述了栈帧对象、回溯对象以及切片对象等等。

4.13 特殊属性

语言实现为部分对象类型添加了一些特殊的只读属性，它们具有各自的作用。其中一些并不会被*dir()* 内置函数所列出。

object.**__dict__**
一个字典或其他类型的映射对象，用于存储对象的（可写）属性。

instance.**__class__**
类实例所属的类。

class.**__bases__**
由类对象的基类所组成的元组。

definition.**__name__**
类、函数、方法、描述器或生成器实例的名称。

definition.**__qualname__**
类、函数、方法、描述器或生成器实例的*qualified name*。

3.3 新版功能.

class.**__mro__**
此属性是由类组成的元组，在方法解析期间会基于它来查找基类。

class.**mro**()
此方法可被一个元类来重载，以为其实例定制方法解析顺序。它会在类实例化时被调用，其结果存储于*__mro__* 之中。

class.**__subclasses__**()
每个类会保存由对其直接子类的弱引用组成的列表。此方法将返回一个由仍然存在的所有此类引用组成的列表。例如:

```
>>> int.__subclasses__()
[<class 'bool'>]
```

备注

CHAPTER 5

内置异常

在 Python 中，所有异常必须为一个派生自 *BaseException* 的类的实例。在带有提及一个特定类的 except 子句的 try 语句中，该子句也会处理任何派生自该类的异常类（但不处理 它所派生出的异常类）。通过子类化创建的两个不相关异常类永远是不等效的，既使它们具有相同的名称。

下面列出的内置异常可通过解释器或内置函数来生成。除非另有说明，它们都会具有一个提示导致错误详细原因的“关联值”。这可以是一个字符串或由多个信息项（例如一个错误码和一个解释错误的字符串）组成的元组。关联值通常会作为参数被传递给异常类的构造器。

用户代码可以引发内置异常。这可被用于测试异常处理程序或报告错误条件，“就像”在解释器引发了相同异常的情况时一样；但是请注意，没有任何机制能防止用户代码引发不适当的错误。

内置异常类可以被子类化以定义新的异常；鼓励程序员从 *Exception* 类或它的某个子类而不是从 *BaseException* 来派生新的异常。关于定义异常的更多信息可以在 Python 教程的 tut-userexceptions 部分查看。

当在 except 或 finally 子句中引发（或重新引发）异常时，__context__ 会被自动设为所捕获的最后一个异常；如果新的异常未被处理，则最终显示的回溯信息将包括原始的异常和最后的异常。

当引发一个新的异常（而不是简单地使用 raise 来重新引发当前在处理的异常）时，隐式的异常上下文可以通过使用带有 raise 的 from 来补充一个显式的原因:

```
raise new_exc from original_exc
```

跟在 from 之后的表达式必须为一个异常或 None。它将在所引发的异常上被设置为 __cause__。设置 __cause__ 还会隐式地将 __suppress_context__ 属性设为 True，这样使用 raise new_exc from None 可以有效地将旧异常替换为新异常来显示其目的 (例如将 *KeyError* 转换为 *AttributeError*)，同时让旧异常在 __context__ 中保持可用状态以便在调试时进行内省。

除了异常本身的回溯以外，默认的回溯还会显示这些串连的异常。__cause__ 中的显式串连异常如果存在将总是显示。__context__ 中的隐式串连异常仅在 __cause__ 为 *None* 并且 __suppress_context__ 为假值时显示。

不论在哪种情况下，异常本身总会在任何串连异常之后显示，以便回溯的最后一行总是显示所引发的最后一个异常。

5.1 基类

下列异常主要被用作其他异常的基类。

exception BaseException
所有内置异常的基类。它不应该被用户自定义类直接继承 (这种情况请使用*Exception*)。如果在此类的实例上调用*str()*，则会返回实例的参数表示，或者当没有参数时返回空字符串。

args
传给异常构造器的参数元组。某些内置异常 (例如*OSError*) 接受特定数量的参数并赋予此元组中的元素特殊的含义，而其他异常通常只接受一个给出错误信息的单独字符串。

with_traceback(*tb*)
此方法将 *tb* 设为异常的新回溯信息并返回该异常对象。它通常以如下的形式在异常处理程序中使用:

```
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

exception Exception
所有内置的非系统退出类异常都派生自此类。所有用户自定义异常也应当派生自此类。

exception ArithmeticError
此基类用于派生针对各种算术类错误而引发的内置异常: *OverflowError*, *ZeroDivisionError*, *FloatingPointError*。

exception BufferError
当与 缓冲区相关的操作无法执行时将被引发。

exception LookupError
此基类用于派生当映射或序列所使用的键或索引无效时引发的异常: *IndexError*, *KeyError*。这可以通过*codecs.lookup()* 来直接引发。

5.2 具体异常

以下异常属于经常被引发的异常。

exception AssertionError
当 `assert` 语句失败时将被引发。

exception AttributeError
当属性引用 (参见 attribute-references) 或赋值失败时将被引发。(当一个对象根本不支持属性引用或属性赋值时则将引发*TypeError*。)

exception EOFError
当*input()* 函数未读取任何数据即达到文件结束条件 (EOF) 时将被引发。(另外，`io.IOBase.read()` 和*io.IOBase.readline()* 方法在遇到 EOF 则将返回一个空字符串。)

exception FloatingPointError
Raised when a floating point operation fails. This exception is always defined, but can only be raised when Python is configured with the `--with-fpectl` option, or the `WANT_SIGFPE_HANDLER` symbol is defined in the `pyconfig.h` file.

exception GeneratorExit
当一个*generator*或*coroutine*被关闭时将被引发；参见 generator.close() 和 coroutine.close()。它直接继承自*BaseException*而不是*Exception*，因为从技术上来说它并不是一个错误。

exception ImportError
当 import 语句尝试加载模块遇到麻烦时将被引发。并且当 from ... import 中的“from list”存在无法找到的名称时也会被引发。

name 与 path 属性可通过对构造器使用仅关键字参数来设定。设定后它们将分别表示被尝试导入的模块名称与触发异常的任意文件所在路径。

在 3.3 版更改: 添加了 name 与 path 属性。

exception ModuleNotFoundError
*ImportError*的子类，当一个模块无法被定位时将由 import 引发。当在*sys.modules*中找到 None 时也会被引发。

3.6 新版功能.

exception IndexError
当序列抽取超出范围时将被引发。（切片索引会被静默截短到允许的范围；如果指定索引不是整数则*TypeError*会被引发。）

exception KeyError
当在现有键集合中找不到指定的映射（字典）键时将被引发。

exception KeyboardInterrupt
当用户按下中断键 (通常为 Control-C 或 Delete) 时将被引发。在执行期间，会定期检测中断信号。该异常继承自*BaseException*以确保不会被处理*Exception*的代码意外捕获，这样可以避免退出解释器。

exception MemoryError
当一个操作耗尽内存但情况仍可（通过删除一些对象）进行挽救时将被引发。关联的值是一个字符串，指明是哪种（内部）操作耗尽了内存。请注意由于底层的内存管理架构（C 的 malloc() 函数），解释器也许并不总是能够从这种情况下完全恢复；但它毕竟可以引发一个异常，这样就能打印出栈回溯信息，以便找出导致问题的失控程序。

exception NameError
当某个局部或全局名称未找到时将被引发。此异常仅用于非限定名称。关联的值是一条错误信息，其中包含未找到的名称。

exception NotImplementedError
此异常派生自*RuntimeError*。在用户自定义的基类中，抽象方法应当在其要求所派生类重载该方法，或是在其要求所开发的类提示具体实现尚待添加时引发此异常。

注解: 它不应当用来表示一个运算符或方法根本不能被支持–在此情况下应当让特定运算符 / 方法保持未定义，或者在子类中将其设为*None*。

注解: NotImplementedError 和 NotImplemented 不可互换，即使它们有相似的名称和用途。请参阅*NotImplemented*了解有关何时使用它们的详细说明。

---

exception OSError([*arg*])
exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])
此异常在一个系统函数返回系统相关的错误时将被引发，此类错误包括 I/O 操作失败例如“文件未找到”或“磁盘已满”等（不包括非法参数类型或其他偶然性错误）。

构造器的第二种形式可设置如下所述的相应属性。如果未指定这些属性则默认为*None*。为了能向下兼容，如果传入了三个参数，则*args*属性将仅包含由前两个构造器参数组成的 2 元组。

构造器实际返回的往往是*OSError*的某个子类，如下文*OS exceptions*中所描述的。具体的子类取决于最终的*errno*值。此行为仅在直接或通过别名来构造*OSError*时发生，并且在子类化时不会被继承。

errno
来自于 C 变量 errno 的数字错误码。

winerror
在 Windows 下，此参数将给出原生的 Windows 错误码。而*errno*属性将是该原生错误码在 POSIX 平台下的近似转换形式。

在 Windows 下，如果 *winerror* 构造器参数是一个整数，则*errno*属性会根据 Windows 错误码来确定，而 *errno* 参数会被忽略。在其他平台上，*winerror* 参数会被忽略，并且*winerror*属性将不存在。

strerror
操作系统所提供的相应错误信息。它在 POSIX 平台中由 C 函数 perror() 来格式化，在 Windows 中则是由 FormatMessage()。

filename
filename2
对于与文件系统路径有关 (例如*open()* 或*os.unlink()*) 的异常，*filename* 是传给函数的文件名。对于涉及两个文件系统路径的函数 (例如*os.rename()*)，*filename2* 将是传给函数的第二个文件名。

在 3.3 版更改: *EnvironmentError*, *IOError*, *WindowsError*, *socket.error*, *select.error* 与 mmap.error 已被合并到*OSError*，构造器可能返回其中一个子类。

在 3.4 版更改: *filename* 属性现在将是传给函数的原始文件名，而不是经过编码或基于文件系统编码进行解码之后的名称。此外还添加了 *filename2* 构造器参数和属性。

exception OverflowError
当算术运算的结果大到无法表示时将被引发。这对整数来说不可能发生（宁可引发*MemoryError*也不会放弃尝试）。但是出于历史原因，有时也会在整数超出要求范围的情况下引发 OverflowError。因为在 C 中缺少对浮点异常处理的标准化，大多数浮点运算都不会做检查。

exception RecursionError
此异常派生自*RuntimeError*。它会在解释器检测发现超过最大递归深度 (参见*sys.getrecursionlimit()*) 时被引发。

3.5 新版功能: 在此之前将只引发*RuntimeError*。

exception ReferenceError
此异常将在使用*weakref.proxy()*函数所创建的弱引用来访问该引用的某个已被作为垃圾回收的属性时被引发。有关弱引用的更多信息请参阅*weakref*模块。

exception RuntimeError
当检测到一个不归属于任何其他类别的错误时将被引发。关联的值是一个指明究竟发生了什么问题的字符串。

exception StopIteration
由内置函数*next()*和*iterator*的*__next__()*方法所引发，用来表示该迭代器不能产生下一项。

该异常对象只有一个属性 value，它在构造该异常时作为参数给出，默认值为*None*。

当一个*generator*或*coroutine*函数返回时，将引发一个新的*StopIteration*实例，函数返回的值将被用作异常构造器的 value 形参。

If a generator function defined in the presence of a from __future__ import generator_stop directive raises *StopIteration*, it will be converted into a *RuntimeError* (retaining the *StopIteration* as

the new exception's cause).

在 3.3 版更改: 添加了 value 属性及其被生成器函数用作返回值的功能。

在 3.5 版更改: Introduced the RuntimeError transformation.

exception StopAsyncIteration
必须由一个*asynchronous iterator* 对象的 __anext__() 方法来引发以停止迭代操作。

3.5 新版功能.

exception SyntaxError
当解析器遇到语法错误时将被引发。这可以发生在 import 语句，对内置函数*exec()* 或*eval()* 的调用，或者读取原始脚本或标准输入（也包括交互模式）的时候。

该类的实例包含有属性 filename, lineno, offset 和 text 用于方便地访问相应的详细信息。异常实例的*str()* 仅返回消息文本。

exception IndentationError
与不正确的缩进相关的语法错误的基类。这是*SyntaxError* 的一个子类。

exception TabError
当缩进包含对制表符和空格符不一致的使用时将被引发。这是*IndentationError* 的一个子类。

exception SystemError
当解释器发现内部错误，但情况看起来尚未严重到要放弃所有希望时将被引发。关联的值是一个指明发生了什么问题的字符串（表示为低层级的符号）。

你应当将此问题报告给你所用 Python 解释器的作者或维护人员。请确认报告 Python 解释器的版本号 (sys.version; 它也会在交互式 Python 会话开始时被打印出来)，具体的错误消息（异常所关联的值）以及可能触发该错误的程序源码。

exception SystemExit
此异常由*sys.exit()* 函数引发。它继承自*BaseException* 而不是*Exception* 以确保不会被处理*Exception* 的代码意外捕获。这允许此异常正确地向上传播并导致解释器退出。如果它未被处理，则 Python 解释器就将退出；不会打印任何栈回溯信息。构造器接受的可选参数与传递给*sys.exit()* 的相同。如果该值为一个整数，则它指明系统退出状态码（会传递给 C 的 exit() 函数）；如果该值为 None，则退出状态码为零；如果该值为其他类型（例如字符串），则会打印对象的值并将退出状态码设为一。

对*sys.exit()* 的调用会被转换为一个异常以便能执行清理处理程序 (try 语句的 finally 子句)，并且使得调试器可以执行一段脚本而不必冒失去控制的风险。如果绝对确实地需要立即退出（例如在调用*os.fork()* 之后的子进程中）则可使用*os._exit()*.

code
传给构造器的退出状态码或错误信息（默认为 None。）

exception TypeError
当一个操作或函数被应用于类型不适当的对象时将被引发。关联的值是一个字符串，给出有关类型不匹配的详情。

此异常可以由用户代码引发，以表明尝试对某个对象进行的操作不受支持也不应当受支持。如果某个对象应当支持给定的操作但尚未提供相应的实现，所要引发的适当异常应为*NotImplementedError*。

传入参数的类型错误 (例如在要求*int* 时却传入了*list*) 应当导致*TypeError*，但传入参数的值错误 (例如传入要求范围之外的数值) 则应当导致*ValueError*。

exception UnboundLocalError
当在函数或方法中对某个局部变量进行引用，但该变量并未绑定任何值时将被引发。此异常是*NameError* 的一个子类。

exception UnicodeError
当发生与 Unicode 相关的编码或解码错误时将被引发。此异常是*ValueError* 的一个子类。

UnicodeError 具有一些描述编码或解码错误的属性。例如 `err.object[err.start:err.end]` 会给出导致编解码器失败的特定无效输入。

encoding
引发错误的编码名称。

reason
描述特定编解码器错误的字符串。

object
编解码器试图要编码或解码的对象。

start
object 中无效数据的开始位置索引。

end
object 中无效数据的末尾位置索引（不含）。

exception UnicodeEncodeError
当在编码过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

exception UnicodeDecodeError
当在解码过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

exception UnicodeTranslateError
在转写过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

exception ValueError
当操作或函数接收到具有正确类型但值不适合的参数，并且情况不能用更精确的异常例如*IndexError* 来描述时将被引发。

exception ZeroDivisionError
当除法或取余运算的第二个参数为零时将被引发。关联的值是一个字符串，指明操作数和运算的类型。

下列异常被保留以与之前的版本相兼容；从 Python 3.3 开始，它们都是*OSError* 的别名。

exception EnvironmentError

exception IOError

exception WindowsError
限在 Windows 中可用。

5.2.1 OS 异常

下列异常均为*OSError* 的子类，它们将根据系统错误代码被引发。

exception BlockingIOError
当一个操作会被某个设置为非阻塞操作的对象（例如套接字）所阻塞时将被引发。对应于 `errno` `EAGAIN`, `EALREADY`, `EWOULDBLOCK` 和 `EINPROGRESS`。

除了*OSError* 已有的属性，*BlockingIOError* 还有一个额外属性：

characters_written
一个整数，表示在被阻塞前已写入到流的字符数。当使用来自*io* 模块的带缓冲 I/O 类时此属性可用。

exception ChildProcessError
当一个子进程上的操作失败时将被引发。对应于 `errno` `ECHILD`。

exception ConnectionError
与连接相关问题的基类。

其子类有*BrokenPipeError*，*ConnectionAbortedError*，*ConnectionRefusedError* 和*ConnectionResetError*。

exception BrokenPipeError
ConnectionError 的子类，当试图写入另一端已被关闭的管道，或是试图写入已关闭写入的套接字时将被引发。对应于 errno EPIPE 和 ESHUTDOWN。

exception ConnectionAbortedError
ConnectionError 的子类，当连接尝试被对端中止时将被引发。对应于 errno ECONNABORTED。

exception ConnectionRefusedError
ConnectionError 的子类，当连接尝试被对端拒绝时将被引发。对应于 errno ECONNREFUSED。

exception ConnectionResetError
ConnectionError 的子类，当连接被对端重置时将被引发。对应于 errno ECONNRESET。

exception FileExistsError
当试图创建一个已存在的文件或目录时将被引发。对应于 errno EEXIST。

exception FileNotFoundError
当所请求的文件或目录不存在时将被引发。对应于 errno ENOENT。

exception InterruptedError
当系统调用被输入信号中断时将被引发。对应于 errno *EINTR*。

在 3.5 版更改: 当系统调用被某个信号中断时，Python 现在会重试系统调用，除非该信号的处理程序引发了其它异常 (原理参见 **PEP 475**) 而不是引发*InterruptedError*。

exception IsADirectoryError
当请求对一个目录执行文件操作 (例如*os.remove()*) 将被引发。对应于 errno EISDIR。

exception NotADirectoryError
当请求对一个非目录对象执行目录操作 (例如*os.listdir()*) 时将被引发。对应于 errno ENOTDIR。

exception PermissionError
当在没有足够操作权限的情况下试图执行某个操作时将被引发——例如缺少文件系统权限。对应于 errno EACCES 和 EPERM。

exception ProcessLookupError
当给定的进程不存在时将被引发。对应于 errno ESRCH。

exception TimeoutError
当一个系统函数发生系统级超时的情况下将被引发。对应于 errno ETIMEDOUT。

3.3 新版功能: 添加了以上所有*OSError* 的子类。

参见:

PEP 3151 - 重写 OS 和 IO 异常的层次结构

5.3 警告

The following exceptions are used as warning categories; see the *warnings* module for more information.

exception Warning
警告类别的基类。

exception UserWarning
用户代码所产生警告的基类。

exception DeprecationWarning
Base class for warnings about deprecated features.

exception PendingDeprecationWarning
Base class for warnings about features which will be deprecated in the future.

exception SyntaxWarning
与模糊的语法相关的警告的基类。

exception RuntimeWarning
与模糊的运行时行为相关的警告的基类。

exception FutureWarning
Base class for warnings about constructs that will change semantically in the future.

exception ImportWarning
与在模块导入中可能的错误相关的警告的基类。

exception UnicodeWarning
与 Unicode 相关的警告的基类。

exception BytesWarning
与*bytes* 和*bytearray* 相关的警告的基类。

exception ResourceWarning
Base class for warnings related to resource usage.

3.2 新版功能.

5.4 异常层次结构

内置异常的类层级结构如下：

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
```

(下页继续)

(续上页)

```
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

CHAPTER 6

文本处理服务

本章介绍的模块提供了广泛的字符串操作和其他文本处理服务。

在二进制数据服务 之下描述的*codecs* 模块也与文本处理高度相关。此外也请参阅 Python 内置字符串类型的文档文本序列类型—*str*。

6.1 string —常见的字符串操作

源代码： Lib/string.py

参见：

文本序列类型—*str*

字符串的方法

6.1.1 字符串常量

此模块中定义的常量为：

string.**ascii_letters**
下文所述*ascii_lowercase* 和*ascii_uppercase* 常量的拼连。该值不依赖于语言区域。

string.**ascii_lowercase**
小写字母 'abcdefghijklmnopqrstuvwxyz'。该值不依赖于语言区域，不会发生改变。

string.**ascii_uppercase**
大写字母 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'。该值不依赖于语言区域，不会发生改变。

string.**digits**
字符串 '0123456789'。

string.**hexdigits**
字符串 '0123456789abcdefABCDEF'。

string.**octdigits**
字符串 '01234567'。

string.**punctuation**
由在 C 语言区域中被视为标点符号的 ASCII 字符组成的字符串。

string.**printable**
由被视为可打印符号的 ASCII 字符组成的字符串。这是*digits*, *ascii_letters*, *punctuation* 和*whitespace* 的总和。

string.**whitespace**
由被视为空白符号的 ASCII 字符组成的字符串。其中包括空格、制表、换行、回车、进纸和纵向制表符。

6.1.2 自定义字符串格式化

内置的字符串类提供了通过使用 **PEP 3101** 所描述的*format()* 方法进行复杂变量替换和值格式化的能力。*string* 模块中的*Formatter* 类允许你使用与内置*format()* 方法相同的实现来创建并定制你自己的字符串格式化行为。

class string.**Formatter**
Formatter 类包含下列公有方法：

format(*format_string*, **args*, ***kwargs*)
首要的 API 方法。它接受一个格式字符串和任意一组位置和关键字参数。它只是一个调用*vformat()* 的包装器。

3.5 版后已移除: Passing a format string as keyword argument *format_string* has been deprecated.

vformat(*format_string*, *args*, *kwargs*)
此函数执行实际的格式化操作。它被公开为一个单独的函数，用于需要传入一个预定义字母作为参数，而不是使用 *args 和 **kwargs 语法将字典解包为多个单独参数并重打包的情况。*vformat()* 完成将格式字符串分解为字符数据和替换字段的工作。它会调用下文所述的几种不同方法。

此外，*Formatter* 还定义了一些旨在被子类替换的方法：

parse(*format_string*)
循环遍历 format_string 并返回一个由可迭代对象组成的元组 (*literal_text*, *field_name*, *format_spec*, *conversion*)。它会被*vformat()* 用来将字符串分解为文本字面值或替换字段。

元组中的值在概念上表示一段字面文本加上一个替换字段。如果没有字面文本（如果连续出现两个替换字段就会发生这种情况），则 *literal_text* 将是一个长度为零的字符串。如果没有替换字段，则 *field_name*, *format_spec* 和 *conversion* 的值将为 None。

get_field(*field_name*, *args*, *kwargs*)
给定 *field_name* 作为*parse()* (见上文) 的返回值，将其转换为要格式化的对象。返回一个元组 (obj, used_key)。默认版本接受在 **PEP 3101** 所定义形式的字符串，例如 “0[name]” 或 “label.title”。*args* 和 *kwargs* 与传给*vformat()* 的一样。返回值 *used_key* 与*get_value()* 的 *key* 形参具有相同的含义。

get_value(*key*, *args*, *kwargs*)
提取给定的字段值。*key* 参数将为整数或字符串。如果是整数，它表示 *args* 中位置参数的索引；如果是字符串，它表示 *kwargs* 中的关键字参数名。

args 形参会被设为*vformat()* 的位置参数列表，而 *kwargs* 形参会被设为由关键字参数组成的字典。

对于复合字段名称，仅会为字段名称的第一个组件调用这些函数；后续组件会通过普通属性和索引操作来进行处理。

因此举例来说，字段表达式‘0.name’将导致调用*get_value()*时附带 *key* 参数值 0。在*get_value()*通过调用内置的*getattr()*函数返回后将会查找 name 属性。

如果索引或关键字引用了一个不存在的项，则将引发*IndexError*或*KeyError*。

check_unused_args(*used_args*, *args*, *kwargs*)
在必要时实现对未使用参数进行检测。此函数的参数是是格式字符串中实际引用的所有参数键的集合（整数表示位置参数，字符串表示名称参数），以及被传给 vformat 的 *args* 和 *kwargs* 的引用。未使用参数的集合可以根据这些形参计算出来。如果检测失败则*check_unused_args()*应会引发一个异常。

format_field(*value*, *format_spec*)
*format_field()*会简单地调用内置全局函数*format()*。提供该方法是为了让子类能够重载它。

convert_field(*value*, *conversion*)
使用给定的转换类型（来自*parse()*方法所返回的元组）来转换（由*get_field()*所返回的）值。默认版本支持‘s’(str), ‘r’(repr) 和‘a’(ascii) 等转换类型。

6.1.3 格式字符串语法

*str.format()*方法和*Formatter*类共享相同的格式字符串语法（虽然对于*Formatter*来说，其子类可以定义它们自己的格式字符串语法）。具体语法与 格式化字符串字面值相似，但也存在区别。

格式字符串包含有以花括号 `{}` 括起来的 “替换字段”。不在花括号之内的内容被视为字面文本，会不加修改地复制到输出中。如果你需要在字面文本中包含花括号字符，可以通过重复来转义: `{{` and `}}`。

替换字段的语法如下:

```
replacement_field  ::=  "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name         ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name           ::=  [identifier | digit+]
attribute_name     ::=  identifier
element_index      ::=  digit+ | index_string
index_string       ::=  <any source character except "]"> +
conversion         ::=  "r" | "s" | "a"
format_spec        ::=  <described in the next section>
```

用不太正式的术语来描述，替换字段开头可以用一个 *field_name* 指定要对值进行格式化并取代替换字符被插入到输出结果的对象。*field_name* 之后有可选的 *conversion* 字段，它是一个感叹号 `'!'` 加一个 *format_spec*，并以一个冒号 `':'` 打头。这些指明了替换值的非默认格式。

另请参阅格式规格迷你语言 一节。

field_name 本身以一个数字或关键字 *arg_name* 打头。如果为数字，则它指向一个位置参数，而如果为关键字，则它指向一个命名关键字参数。如果格式字符串中的数字 arg_names 为 0, 1, 2, …的序列，它们可以全部省略（而非部分省略），数字 0, 1, 2, …将会按顺序自动插入。由于 *arg_name* 不使用引号分隔，因此无法在格式字符串中指定任意的字典键 (例如字符串 `'10'` 或 `':-]'`)。*arg_name* 之后可以带上任意数量的索引或属性表达式。`'.name'` 形式的表达式会使用*getattr()*选择命名属性，而 `'[index]'` 形式的表达式会使用 `__getitem__()` 执行索引查找。

在 3.1 版更改: 位置参数说明符对于*str.format()*可以省略，因此 `'{} {}'.format(a, b)` 等价于 `'{0} {1}'.format(a, b)`。

在 3.4 版更改: 位置参数说明符对于*Formatter* 可以省略。

一些简单的格式字符串示例

```
"First, thou shalt count to {0}"  # References first positional argument
"Bring me a {}"                   # Implicitly references the first positional␣
↪argument
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument 'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"   # First element of keyword argument 'players'.
```

使用 *conversion* 字段在格式化之前进行类型强制转换。通常，格式化值的工作由值本身的 `__format__()` 方法来完成。但是，在某些情况下最好强制将类型格式化为一个字符串，覆盖其本身的格式化定义。通过在调用 `__format__()` 之前将值转换为字符串，可以绕过正常的格式化逻辑。

目前支持的转换旗标有三种: `'!s'` 会对值调用*str()*，`'!r'` 调用*repr()* 而 `'!a'` 则调用*ascii()*。

几个例子:

```
"Harold's a clever {0!s}"         # Calls str() on the argument first
"Bring out the holy {name!r}"     # Calls repr() on the argument first
"More {!a}"                       # Calls ascii() on the argument first
```

format_spec 字段包含值应如何呈现的规格描述，例如字段宽度、对齐、填充、小数精度等细节信息。每种值类型可以定义自己的“格式化迷你语言”或对 *format_spec* 的解读方式。

大多数内置类型都支持同样的格式化迷你语言，具体描述见下一节。

format_spec 字段还可以在其内部包含嵌套的替换字段。这些嵌套的替换字段可能包括字段名称、转换旗标和格式规格描述，但是不再允许更深层的嵌套。format_spec 内部的替换字段会在解读 *format_spec* 字符串之前先被解读。这将允许动态地指定特定值的格式。

请参阅格式示例 一节查看相关示例。

格式规格迷你语言

“格式规格”在格式字符串所包含的替换字段内部使用，用于定义单个值应如何呈现 (参见格式字符串语法 和 f-strings)。它们也可以被直接传给内置的*format()* 函数。每种可格式化的类型都可以自行定义如何对格式规格进行解读。

大多数内置类型都为格式规格实现了下列选项，不过某些格式化选项只被数值类型所支持。

A general convention is that an empty format string (`""`) produces the same result as if you had called *str()* on the value. A non-empty format string typically modifies the result.

标准格式说明符的一般形式如下:

```
format_spec     ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill            ::=  <any character>
align           ::=  "<" | ">" | "=" | "^"
sign            ::=  "+" | "-" | " "
width           ::=  digit+
grouping_option ::=  "_" | ","
precision       ::=  digit+
type            ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s
```

如果指定了一个有效的 *align* 值，则可以在该值前面加一个 *fill* 字符，它可以为任意字符，如果省略则默认为

空格符。在 格式化字符串字面值或在使用*str.format()* 方法时是无法使用花括号字面值 (“{” or “}”) 作为 *fill* 字符的。但是，通过嵌套替换字段插入花括号则是可以的。这个限制不会影响*format()* 函数。

各种对齐选项的含义如下：

选项	含义
'<'	强制字段在可用空间内左对齐（这是大多数对象的默认值）。
'>'	强制字段在可用空间内右对齐（这是数字的默认值）。
'='	强制将填充放置在符号（如果有）之后但在数字之前。这用于以“+000000120”形式打印字段。此对齐选项仅对数字类型有效。当’0’紧接在字段宽度之前时，它成为默认值。
'^'	强制字段在可用空间内居中。

请注意，除非定义了最小字段宽度，否则字段宽度将始终与填充它的数据大小相同，因此在这种情况下，对齐选项没有意义。

sign 选项仅对数字类型有效，可以是以下之一：

选项	含义
'+'	表示标志应该用于正数和负数。
'-'	表示标志应仅用于负数（这是默认行为）。
space	表示应在正数上使用前导空格，在负数上使用减号。

'#' 选项可以让“替代形式”被用于转换。替代形式可针对不同类型分别定义。此选项仅对整数、浮点、复数和 Decimal 类型有效。对于整数类型，当使用二进制、八进制或十六进制输出时，此选项会为输出值添加相应的 '0b','0o' 或 '0x' 前缀。对于浮点数、复数和 Decimal 类型，替代形式会使得转换结果总是包含小数点符号，即使其不带小数。通常只有在带有小数的情况下，此类转换的结果中才会出现小数点符号。此外，对于 'g' 和 'G' 转换，末尾的零不会从结果中被移除。

',' 选项表示使用逗号作为千位分隔符。对于感应区域设置的分隔符，请改用 'n' 整数表示类型。

在 3.1 版更改: 添加了 ',' 选项 (另请参阅 **PEP 378**)。

'_' 选项表示对浮点表示类型和整数表示类型 'd' 使用下划线作为千位分隔符。对于整数表示类型 'b', 'o','x' 和 'X'，将为每 4 个数位插入一个下划线。对于其他表示类型指定此选项则将导致错误。

在 3.6 版更改: 添加了 '_' 选项 (另请参阅 **PEP 515**)。

width is a decimal integer defining the minimum field width. If not specified, then the field width will be determined by the content.

当未显式给出对齐方式时，在 *width* 字段前加一个零 ('0') 字段将为数字类型启用感知正负号的零填充。这相当于设置 *fill* 字符为 '0' 且 *alignment* 类型为 '='。

precision 是一个十进制数字，表示对于以 'f' and 'F' 格式化的浮点数值要在小数点后显示多少个数位，或者对于以 'g' 或 'G' 格式化的浮点数值要在小数点前后共显示多少个数位。对于非数字类型，该字段表示最大字段大小——换句话说就是要使用多少个来自字段内容的字符。对于整数值则不允许使用 *precision*。

最后，*type* 确定了数据应如何呈现。

可用的字符串表示类型是：

类型	含义
's'	字符串格式。这是字符串的默认类型，可以省略。
None	和 's' 一样。

可用的整数表示类型是：

类型	含义
'b'	二进制格式。输出以 2 为基数的数字。
'c'	字符。在打印之前将整数转换为相应的 unicode 字符。
'd'	十进制整数。输出以 10 为基数的数字。
'o'	八进制格式。输出以 8 为基数的数字。
'x'	十六进制格式。输出以 16 为基数的数字，使用小写字母表示 9 以上的数码。
'X'	十六进制格式。输出以 16 为基数的数字，使用大写字母表示 9 以上的数码。
'n'	数字。这与 'd' 相似，不同之处在于它会使用当前区域设置来插入适当的数字分隔字符。
None	和 'd' 相同。

在上述的表示类型之外，整数还可以通过下列的浮点表示类型来格式化 (除了 'n' 和 None)。当这样做时，会在格式化之前使用 *float()* 将整数转换为浮点数。

浮点数和小数值可用的表示类型有：

类型	含义
'e'	指数表示。以使用字母‘e’来标示指数的科学计数法打印数字。默认的精度为 6。
'E'	指数表示。与 'e' 相似，不同之处在于它使用大写字母‘E’作为分隔字符。
'f'	定点表示。将数字显示为一个定点数。默认的精确度为 6。
'F'	定点表示。与 'f' 相似，但会将 nan 转为 NAN 并将 inf 转为 INF。
'g'	常规格式。对于给定的精度 p >= 1，这会将数值舍入到 p 位有效数字，再将结果以定点格式或科学计数法进行格式化，具体取决于其值的大小。 The precise rules are as follows: suppose that the result formatted with presentation type 'e' and precision p-1 would have exponent exp. Then if -4 <= exp < p, the number is formatted with presentation type 'f' and precision p-1-exp. Otherwise, the number is formatted with presentation type 'e' and precision p-1. In both cases insignificant trailing zeros are removed from the significand, and the decimal point is also removed if there are no remaining digits following it. 正负无穷，正负零和 nan 会分别被格式化为 inf, -inf, 0, -0 和 nan，无论精度如何设定。 精度 0 会被视为等同于精度 1。默认精度为 6。
'G'	常规格式。类似于 'g'，不同之处在于当数值非常大时会切换为 'E'。无穷与 NaN 也会表示为大写形式。
'n'	数字。这与 'g' 相似，不同之处在于它会使用当前区域设置来插入适当的数字分隔字符。
'%'	百分比。将数字乘以 100 并显示为定点 ('f') 格式，后面带一个百分号。
None	类似于 'g'，不同之处在于当使用定点表示法时，小数点后将至少显示一位。默认精度与表示给定值所需的精度一样。整体效果为与其他格式修饰符所调整的 *str()* 输出保持一致。

格式示例

本节包含*str.format()* 语法的示例以及与旧式 `%` 格式化的比较。

该语法在大多数情况下与旧式的 `%` 格式化类似，只是增加了 `{}` 和 `:` 来取代 `%`。例如，, `'%03.2f'` 可以被改写为 `'{:03.2f}'`。

新的格式语法还支持新增的不同选项，将在以下示例中说明。

按位置访问参数:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

按名称访问参数:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.
↪81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

访问参数的属性:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.
↪0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

访问参数的项:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

替代 `%s` 和 `%r`:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

对齐文本以及指定宽度:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

替代 `%+f`, `%-f` 和 `% f` 以及指定正负号:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

替代 `%x` 和 `%o` 以及转换基于不同进位制的值:

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

使用逗号作为千位分隔符:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

表示为百分数:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

使用特定类型的专属格式化:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

嵌套参数以及更复杂的示例:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
```

(下页继续)

(续上页)

```
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
    5     5     5   101
    6     6     6   110
    7     7     7   111
    8     8    10  1000
    9     9    11  1001
   10     A    12  1010
   11     B    13  1011
```

6.1.4 模板字符串

模板字符串提供了由 **PEP 292** 所描述的更简便的字符串替换方式。模板字符串的一个主要用例是文本国际化 (i18n)，因为在此场景下，更简单的语法和功能使得文本翻译过程比使用 Python 的其他内置字符串格式化工具更为方便。作为基于模板字符串构建以实现 i18n 的库的一个示例，请参看 flufl.i18n 包。

模板字符串支持基于 `$` 的替换，使用以下规则：

- `$$` 为转义符号；它会被替换为单个的 `$`。
- `$identifier` 为替换占位符，它会匹配一个名为 `"identifier"` 的映射键。在默认情况下，`"identifier"` 限制为任意 ASCII 字母数字（包括下划线）组成的字符串，不区分大小写，以下划线或 ASCII 字母开头。在 `$` 字符之后的第一个非标识符字符将表明占位符的终结。
- `${identifier}` 等价于 `$identifier`。当占位符之后紧跟着有效的但又不是占位符一部分的标识符字符时需要使用，例如 `"${noun}ification"`。

在字符串的其他位置出现 `$` 将导致引发*ValueError*。

string 模块提供了实现这些规则的*Template* 类。*Template* 有下列方法：

class string.**Template**(*template*)

该构造器接受一个参数作为模板字符串。

substitute(*mapping*, ***kwds*)

执行模板替换，返回一个新字符串。*mapping* 为任意字典类对象，其中的键将匹配模板中的占位符。或者你也可以提供一组关键字参数，其中的关键字即对应占位符。当同时给出 *mapping* 和 *kwds* 并且存在重复时，则以 *kwds* 中的占位符为优先。

safe_substitute(*mapping*, ***kwds*)

类似于*substitute()*，不同之处是如果有占位符未在 *mapping* 和 *kwds* 中找到，不是引发*KeyError* 异常，而是将原始占位符不加修改地显示在结果字符串中。另一个与*substitute()* 的差异是任何在其他情况下出现的 `$` 将简单地返回 `$` 而不是引发*ValueError*。

此方法被认为“安全”，因为虽然仍有可能发生其他异常，但它总是尝试返回可用的字符串而不是引发一个异常。从另一方面来说，*safe_substitute()* 也可能根本算不上安全，因为它将静默地忽略错误格式的模板，例如包含多余的分隔符、不成对的花括号或不是合法 Python 标识符的占位符等等。

Template 的实例还提供一个公有数据属性：

template
这是作为构造器的 *template* 参数被传入的对象。一般来说，你不应该修改它，但并不强制要求只读访问。

以下是一个如何使用模版的示例：

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

进阶用法：你可以派生 *Template* 的子类来自定义占位符语法、分隔符，或用于解析模板字符串的整个正则表达式。为此目的，你可以重载这些类属性：

- *delimiter* –This is the literal string describing a placeholder introducing delimiter. The default value is `$`. Note that this should *not* be a regular expression, as the implementation will call *re.escape()* on this string as needed.
- *idpattern* –This is the regular expression describing the pattern for non-braced placeholders (the braces will be added automatically as appropriate). The default value is the regular expression `(?-i:[_a-zA-Z][_a-zA-Z0-9]*)`.

  注解： Since default *flags* is `re.IGNORECASE`, pattern `[a-z]` can match with some non-ASCII characters. That's why we use local `-i` flag here.

  While *flags* is kept to `re.IGNORECASE` for backward compatibility, you can override it to `0` or `re.IGNORECASE | re.ASCII` when subclassing.

- *flags* –将在编译用于识别替换内容的正则表达式被应用的正则表达式旗标。默认值为 `re.IGNORECASE`。请注意 `re.VERBOSE` 总是会被加为旗标，因此自定义的 *idpattern* 必须遵循详细正则表达式的约定。

  3.2 新版功能.

作为另一种选项，你可以通过重载类属性 *pattern* 来提供整个正则表达式模式。如果你这样做，该值必须为一个具有四个命名捕获组的正则表达式对象。这些捕获组对应于上面已经给出的规则，以及无效占位符的规则：

- *escaped* –这个组匹配转义序列，在默认模式中即 `$$`。
- *named* –这个组匹配不带花括号的占位符名称；它不应当包含捕获组中的分隔符。
- *braced* –这个组匹配带有花括号的占位符名称；它不应当包含捕获组中的分隔符或者花括号。
- *invalid* –这个组匹配任何其他分隔符模式（通常为单个分隔符），并且它应当出现在正则表达式的末尾。

6.1.5 辅助函数

string.**capwords**(*s*, *sep=None*)
使用*str.split()* 将参数拆分为单词，使用*str.capitalize()* 将单词转为大写形式，使用*str.join()* 将大写的单词进行拼接。如果可选的第二个参数 *sep* 被省略或为 None，则连续的空白字符会被替换为单个空格符并且开头和末尾的空白字符会被移除，否则 *sep* 会被用来拆分和拼接单词。

6.2 re —正则表达式操作

源代码: Lib/re.py

这个模块提供了与 Perl 语言类似的正则表达式匹配操作。

模式和被搜索的字符串既可以是 Unicode 字符串 (*str*)，也可以是 8 位字节串 (*bytes*)。但是，Unicode 字符串与 8 位字节串不能混用：也就是说，你不能用一个字节串模式去匹配 Unicode 字符串，反之亦然；类似地，当进行替换操作时，替换字符串的类型也必须与所用的模式和搜索字符串的类型一致。

正则表达式使用反斜杠（'\'）来表示特殊形式，或者把特殊字符转义成普通字符。而反斜杠在普通的 Python 字符串里也有相同的作用，所以就产生了冲突。比如说，要匹配一个字面上的反斜杠，正则表达式模式不得不写成 '\\\\'，因为正则表达式里匹配一个反斜杠必须是 \\，而每个反斜杠在普通的 Python 字符串里都要写成 \\。

解决办法是对于正则表达式样式使用 Python 的原始字符串表示法；在带有 'r' 前缀的字符串字面值中，反斜杠不必做任何特殊处理。因此 r"\n" 表示包含 '\' 和 'n' 两个字符的字符串，而 "\n" 则表示只包含一个换行符的字符串。样式在 Python 代码中通常都会使用这种原始字符串表示法来表示。

绝大部分正则表达式操作都提供为模块函数和方法，在编译正则表达式. 这些函数是一个捷径，不需要先编译一个正则对象，但是损失了一些优化参数。

参见:

第三方模块 regex , 提供了与标准库*re* 模块兼容的 API 接口, 同时还提供了额外的功能和更全面的 Unicode 支持。

6.2.1 正则表达式语法

一个正则表达式（或 RE）指定了一集与之匹配的字符串；模块内的函数可以让你检查某个字符串是否跟给定的正则表达式匹配（或者一个正则表达式是否匹配到一个字符串，这两种说法含义相同）。

正则表达式可以拼接；如果 *A* 和 *B* 都是正则表达式，那么 *AB* 也是正则表达式。通常，如果字符串 *p* 匹配 *A* 并且另一个字符串 *q* 匹配 *B*, 那么 *pq* 可以匹配 AB。除非 *A* 或者 *B* 包含低优先级操作，*A* 和 *B* 存在边界条件；或者命名组引用。所以，复杂表达式可以很容易的从这里描述的简单源语表达式构建。了解更多正则表达式理论和实现，参考 the Friedl book [Frie09]，或者其他编译器构建的书籍。

以下是正则表达式格式的简要说明。更详细的信息和演示，参考 regex-howto。

正则表达式可以包含普通或者特殊字符。绝大部分普通字符，比如 'A', 'a', 或者 '0'，都是最简单的正则表达式。它们就匹配自身。你可以拼接普通字符，所以 last 匹配字符串 'last'. （在这一节的其他部分，我们将用 this special style 这种方式表示正则表达式，通常不带引号，要匹配的字符串用 'in single quotes'，单引号形式。）

有些字符，比如 '|' 或者 '('，属于特殊字符。特殊字符既可以表示它的普通含义，也可以影响它旁边的正则表达式的解释。

重复修饰符 (*, +, ?, {m,n}, 等) 不能直接嵌套。这样避免了非贪婪后缀 ? 修饰符，和其他实现中的修饰符产生的多义性。要应用一个内层重复嵌套，可以使用括号。比如，表达式 (?:a{6})* 匹配 6 个 'a' 字符重复任意次数。

特殊字符是：

. (点) 在默认模式，匹配除了换行的任意字符。如果指定了标签*DOTALL*，它将匹配包括换行符的任意字符。

^ (插入符号) 匹配字符串的开头，并且在*MULTILINE* 模式也匹配换行后的首个符号。

$ 匹配字符串尾或者在字符串尾的换行符的前一个字符，在*MULTILINE* 模式下也会匹配换行符之前的文本。foo 匹配 'foo' 和 'foobar'，但正则表达式 foo$ 只匹配 'foo'。更有趣的是，在 'foo1\nfoo2\n' 中搜索 foo.$，通常匹配 'foo2'，但在*MULTILINE* 模式下可以匹配到 'foo1'；在 'foo\n' 中搜索 $ 会找到两个（空的）匹配：一个在换行符之前，一个在字符串的末尾。

* 对它前面的正则式匹配 0 到任意次重复，尽量多的匹配字符串。ab* 会匹配 'a'，'ab'，或者 'a' 后面跟随任意个 'b'。

+ 对它前面的正则式匹配 1 到任意次重复。ab+ 会匹配 'a' 后面跟随 1 个以上到任意个 'b'，它不会匹配 'a'。

? 对它前面的正则式匹配 0 到 1 次重复。ab? 会匹配 'a' 或者 'ab'。

?, +?, ?? '', '+'，和 '?' 修饰符都是 贪婪的；它们在字符串进行尽可能多的匹配。有时候并不需要这种行为。如果正则式 <.*> 希望找到 '<a> b <c>'，它将会匹配整个字符串，而不仅是 '<a>'。在修饰符之后添加 ? 将使样式以 非贪婪 *方式或者:dfn:*最小方式进行匹配；尽量 少的字符将会被匹配。使用正则式 <.*?> 将会仅仅匹配 '<a>'。

"{m}" 对其之前的正则式指定匹配 *m* 个重复；少于 *m* 的话就会导致匹配失败。比如，a{6} 将匹配 6 个 'a'，但是不能是 5 个。

"{m, n}" 对正则式进行 *m* 到 *n* 次匹配，在 *m* 和 *n* 之间取尽量多。比如，a{3,5} 将匹配 3 到 5 个 'a'。忽略 *m* 意为指定下界为 0，忽略 *n* 指定上界为无限次。比如 a{4,}b 将匹配 'aaaab' 或者 1000 个 'a' 尾随一个 'b'，但不能匹配 'aaab'。逗号不能省略，否则无法辨别修饰符应该忽略哪个边界。

{m,n}? 前一个修饰符的非贪婪模式，只匹配尽量少的字符次数。比如，对于 'aaaaaa'，a{3,5} 匹配 5 个 'a'，而 a{3,5}? 只匹配 3 个 'a'。

\ 转义特殊字符（允许你匹配 '*', '?', 或者此类其他），或者表示一个特殊序列；特殊序列之后进行讨论。

如果你没有使用原始字符串（r'raw'）来表达样式，要牢记 Python 也使用反斜杠作为转义序列；如果转义序列不被 Python 的分析器识别，反斜杠和字符才能出现在字符串中。如果 Python 可以识别这个序列，那么反斜杠就应该重复两次。这将导致理解障碍，所以高度推荐，就算是最简单的表达式，也要使用原始字符串。

[] 用于表示一个字符集合。在一个集合中：

- 字符可以单独列出，比如 [amk] 匹配 'a'，'m'，或者 'k'。
- 可以表示字符范围，通过用 '-' 将两个字符连起来。比如 [a-z] 将匹配任何小写 ASCII 字符，[0-5][0-9] 将匹配从 00 到 59 的两位数字，[0-9A-Fa-f] 将匹配任何十六进制数位。如果 - 进行了转义（比如 [a\-z]）或者它的位置在首位或者末尾（如 [-a] 或 [a-]），它就只表示普通字符 '-'。
- 特殊字符在集合中，失去它的特殊含义。比如 [(+*)] 只会匹配这几个文法字符 '(', '+', '*', or ')'。
- 字符类如 \w 或者 \S (如下定义) 在集合内可以接受，它们可以匹配的字符由*ASCII* 或者*LOCALE* 模式决定。
- 不在集合范围内的字符可以通过 取反来进行匹配。如果集合首字符是 '^'，所有 不在集合内的字符将会被匹配，比如 [^5] 将匹配所有字符，除了 '5'，[^^] 将匹配所有字符，除了 '^'. ^ 如果不在集合首位，就没有特殊含义。

- 在集合内要匹配一个字符 `']'`，有两种方法，要么就在它之前加上反斜杠，要么就把它放到集合首位。比如，`[()[\]{}]` 和 `[]()[{}]` 都可以匹配括号。

`|` `A|B`，*A* 和 *B* 可以是任意正则表达式，创建一个正则表达式，匹配 *A* 或者 *B*. 任意个正则表达式可以用 `'|'` 连接。它也可以在组合（见下列）内使用。扫描目标字符串时，`'|'` 分隔开的正则样式从左到右进行匹配。当一个样式完全匹配时，这个分支就被接受。意思就是，一旦 *A* 匹配成功，*B* 就不再进行匹配，即便它能产生一个更好的匹配。或者说，`'|'` 操作符绝不贪婪。如果要匹配 `'|'` 字符，使用 `\|`，或者把它包含在字符集里，比如 `[|]`.

`(...)`（组合），匹配括号内的任意正则表达式，并标识出组合的开始和结尾。匹配完成后，组合的内容可以被获取，并可以在之后用 `\number` 转义序列进行再次匹配，之后进行详细说明。要匹配字符 `'('` 或者 `')'`，用 `\(` 或 `\)`，或者把它们包含在字符集合里: `[(]`, `[)]`.

`(?…)` 这是个扩展标记法（一个 `'?'` 跟随 `'('` 并无含义）。`'?'` 后面的第一个字符决定了这个构建采用什么样的语法。这种扩展通常并不创建新的组合；`(?P<name>...)` 是唯一的例外。以下是目前支持的扩展。

`(?aiLmsux)` (`'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'` 中的一个或多个) 这个组合匹配一个空字符串；这些字符对正则表达式设置以下标记 *re.A* (只匹配 ASCII 字符), *re.I* (忽略大小写), *re.L* (语言依赖), *re.M* (多行模式), *re.S* (点 dot 匹配全部字符), `re.U` (Unicode 匹配), and *re.X* (冗长模式)。(这些标记在模块内容 中描述) 如果你想将这些标记包含在正则表达式中，这个方法就很有用，免去了在 *re.compile()* 中传递 *flag* 参数。标记应该在表达式字符串首位表示。

`(?:…)` 正则括号的非捕获版本。匹配在括号内的任何正则表达式，但该分组所匹配的子字符串 不能在执行匹配后被获取或是之后在模式中被引用。

`(?imsx-imsx:...)` (Zero or more letters from the set `'i'`, `'m'`, `'s'`, `'x'`, optionally followed by `'-'` followed by one or more letters from the same set.) The letters set or removes the corresponding flags: *re.I* (ignore case), *re.M* (multi-line), *re.S* (dot matches all), and *re.X* (verbose), for the part of the expression. (The flags are described in 模块内容.)

3.6 新版功能.

`(?P<name>…)`（命名组合）类似正则组合，但是匹配到的子串组在外部是通过定义的 *name* 来获取的。组合名必须是有效的 Python 标识符，并且每个组合名只能用一个正则表达式定义，只能定义一次。一个符号组合同样是一个数字组合，就像这个组合没有被命名一样。

命名组合可以在三种上下文中引用。如果样式是 `(?P<quote>['"]).*?(?P=quote)` （也就是说，匹配单引号或者双引号括起来的字符串):

引用组合“quote”的上下文	引用方法
在正则式自身内	• `(?P=quote)` (如示) • `\1`
处理匹配对象 *m*	• `m.group('quote')` • `m.end('quote')` (等)
传递到 `re.sub()` 里的 *repl* 参数中	• `\g<quote>` • `\g<1>` • `\1`

`(?P=name)` 反向引用一个命名组合；它匹配前面那个叫 *name* 的命名组中匹配到的串同样的字串。

`(?#…)` 注释；里面的内容会被忽略。

(?=…) 匹配 …的内容，但是并不消费样式的内容。这个叫做 *lookahead assertion*。比如，`Isaac (?=Asimov)` 匹配 `'Isaac '` 只有在后面是 `'Asimov'` 的时候。

(?!…) 匹配 …不符合的情况。这个叫 *negative lookahead assertion*（前视取反）。比如说，`Isaac (?!Asimov)` 只有后面 不是 `'Asimov'` 的时候才匹配 `'Isaac '` 。

(?<=…) 匹配字符串的当前位置，它的前面匹配 …的内容到当前位置。这叫:dfn:*positive lookbehind assertion*（正向后视断定）。`(?<=abc)def` 会在 `'abcdef'` 中找到一个匹配，因为后视会往后看 3 个字符并检查是否包含匹配的样式。包含的匹配样式必须是定长的，意思就是 `abc` 或 `a|b` 是允许的，但是 `a*` 和 `a{3,4}` 不可以。注意以 positive lookbehind assertions 开始的样式，如 `(?<=abc)def` ，并不是从 a 开始搜索，而是从 d 往回看的。你可能更加愿意使用*`search()`* 函数，而不是*`match()`* 函数：

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

这个例子搜索一个跟随在连字符后的单词：

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

在 3.5 版更改: 添加定长组合引用的支持。

(?<!…) 匹配当前位置之前不是 …的样式。这个叫:dfn:*negative lookbehind assertion* （后视断定取非）。类似正向后视断定，包含的样式匹配必须是定长的。由 negative lookbehind assertion 开始的样式可以从字符串搜索开始的位置进行匹配。

(?(id/name)yes-pattern|no-pattern) 如果给定的 *id* 或 *name* 存在，将会尝试匹配 `yes-pattern` ，否则就尝试匹配 `no-pattern`，`no-pattern` 可选，也可以被忽略。比如，`(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` 是一个 email 样式匹配，将匹配 `'<user@host.com>'` 或 `'user@host.com'` ，但不会匹配 `'<user@host.com'` ，也不会匹配 `'user@host.com>'`。

由 `'\'` 和一个字符组成的特殊序列在以下列出。如果普通字符不是 ASCII 数位或者 ASCII 字母，那么正则样式将匹配第二个字符。比如，`\$` 匹配字符 `'$'`.

\number 匹配数字代表的组合。每个括号是一个组合，组合从 1 开始编号。比如 `(.+) \1` 匹配 `'the the'` 或者 `'55 55'`，但不会匹配 `'thethe'` (注意组合后面的空格)。这个特殊序列只能用于匹配前面 99 个组合。如果 *number* 的第一个数位是 0，或者 *number* 是三个八进制数，它将不会被看作是一个组合，而是八进制的数字值。在 `'['` 和 `']'` 字符集合内，任何数字转义都被看作是字符。

\A 只匹配字符串开始。

\b 匹配空字符串，但只在单词开始或结尾的位置。一个单词被定义为一个单词字符的序列。注意，通常 `\b` 定义为 `\w` 和 `\W` 字符之间，或者 `\w` 和字符串开始/结尾的边界，意思就是 `r'\bfoo\b'` 匹配 `'foo'`, `'foo.'`, `'(foo)'`, `'bar foo baz'` 但不匹配 `'foobar'` 或者 `'foo3'`。

默认情况下，Unicode 字母和数字是在 Unicode 样式中使用的，但是可以用*`ASCII`* 标记来更改。如果*`LOCALE`* 标记被设置的话，词的边界是由当前语言区域设置决定的，`\b` 表示退格字符，以便与 Python 字符串文本兼容。

\B 匹配空字符串，但 不能在词的开头或者结尾。意思就是 `r'py\B'` 匹配 `'python'`, `'py3'`, `'py2'`，但不匹配 `'py'`, `'py.'`, 或者 `'py!'`. `\B` 是 `\b` 的取非，所以 Unicode 样式的词语是由 Unicode 字母，数字或下划线构成的，虽然可以用*`ASCII`* 标志来改变。如果使用了*`LOCALE`* 标志，则词的边界由当前语言区域设置。

\d

对于 Unicode (str) 样式： Matches any Unicode decimal digit (that is, any character in Unicode character category [Nd]). This includes `[0-9]`, and also many other digit characters. If the *`ASCII`* flag is used only

[0-9] is matched (but the flag affects the entire regular expression, so in such cases using an explicit [0-9] may be a better choice).

对于 8 位 (bytes) 样式： 匹配任何十进制数，就是 [0-9]。

\D Matches any character which is not a decimal digit. This is the opposite of \d. If the *ASCII* flag is used this becomes the equivalent of [^0-9] (but the flag affects the entire regular expression, so in such cases using an explicit [^0-9] may be a better choice).

\s

对于 Unicode (str) 样式： Matches Unicode whitespace characters (which includes [\t\n\r\f\v], and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages). If the *ASCII* flag is used, only [\t\n\r\f\v] is matched (but the flag affects the entire regular expression, so in such cases using an explicit [\t\n\r\f\v] may be a better choice).

对于 8 位 (bytes) 样式： 匹配 ASCII 中的空白字符，就是 [\t\n\r\f\v] 。

\S Matches any character which is not a whitespace character. This is the opposite of \s. If the *ASCII* flag is used this becomes the equivalent of [^ \t\n\r\f\v] (but the flag affects the entire regular expression, so in such cases using an explicit [^ \t\n\r\f\v] may be a better choice).

\w

对于 Unicode (str) 样式： Matches Unicode word characters; this includes most characters that can be part of a word in any language, as well as numbers and the underscore. If the *ASCII* flag is used, only [a-zA-Z0-9_] is matched (but the flag affects the entire regular expression, so in such cases using an explicit [a-zA-Z0-9_] may be a better choice).

对于 8 位 (bytes) 样式： 匹配 ASCII 字符中的数字和字母和下划线，就是 [a-zA-Z0-9_] 。如果设置了*LOCALE* 标记，就匹配当前语言区域的数字和字母和下划线。

\W Matches any character which is not a word character. This is the opposite of \w. If the *ASCII* flag is used this becomes the equivalent of [^a-zA-Z0-9_] (but the flag affects the entire regular expression, so in such cases using an explicit [^a-zA-Z0-9_] may be a better choice). If the *LOCALE* flag is used, matches characters considered alphanumeric in the current locale and the underscore.

\Z 只匹配字符串尾。

绝大部分 Python 的标准转义字符也被正则表达式分析器支持。：

```
\a      \b      \f      \n
\r      \t      \u      \U
\v      \x      \\
```

（注意 \b 被用于表示词语的边界，它只在字符集合内表示退格，比如 [\b] 。）

'\u' and '\U' escape sequences are only recognized in Unicode patterns. In bytes patterns they are errors.

八进制转义包含为一个有限形式。如果首位数字是 0，或者有三个八进制数位，那么就认为它是八进制转义。其他的情况，就看作是组引用。对于字符串文本，八进制转义最多有三个数位长。

在 3.3 版更改: 增加了 '\u' 和 '\U' 转义序列。

在 3.6 版更改: 由 '\' 和一个 ASCII 字符组成的未知转义会被看成错误。

6.2.2 模块内容

模块定义了几个函数，常量，和一个例外。有些函数是编译后的正则表达式方法的简化版本（少了一些特性）。绝大部分重要的应用，总是会先将正则表达式编译，之后在进行操作。

在 3.6 版更改: 标志常量现在是 RegexFlag 类的实例，这个类是*enum.IntFlag* 的子类。

re.**compile**(*pattern*, *flags=0*)

Compile a regular expression pattern into a *regular expression object*, which can be used for matching using its *match()*, *search()* and other methods, described below.

这个表达式的行为可以通过指定 标记的值来改变。值可以是以下任意变量，可以通过位的 OR 操作来结合（| 操作符）。

序列

```
prog = re.compile(pattern)
result = prog.match(string)
```

等价于

```
result = re.match(pattern, string)
```

如果需要多次使用这个正则表达式的话，使用*re.compile()* 和保存这个正则对象以便复用，可以让程序更加高效。

注解: 通过*re.compile()* 编译后的样式，和模块级的函数会被缓存，所以少数的正则表达式使用无需考虑编译的问题。

---

re.**A**

re.**ASCII**

让 \w, \W, \b, \B, \d, \D, \s 和 \S 只匹配 ASCII，而不是 Unicode。这只对 Unicode 样式有效，会被 byte 样式忽略。相当于前面语法中的内联标志 (?a) 。

注意，为了保持向后兼容，re.U 标记依然存在（还有他的同义 re.UNICODE 和嵌入形式 (?u)），但是这些在 Python 3 是冗余的，因为默认字符串已经是 Unicode 了（并且 Unicode 匹配不允许 byte 出现）。

re.**DEBUG**

显示编译时的 debug 信息，没有内联标记。

re.**I**

re.**IGNORECASE**

进行忽略大小写匹配；表达式如 [A-Z] 也会匹配小写字符。Unicode 匹配（比如 Ü 匹配 ü）同样有用，除非设置了*re.ASCII* 标记来禁用非 ASCII 匹配。当前语言区域不会改变这个标记，除非设置了*re.LOCALE* 标记。这个相当于内联标记 (?i) 。

Note that when the Unicode patterns [a-z] or [A-Z] are used in combination with the *IGNORECASE* flag, they will match the 52 ASCII letters and 4 additional non-ASCII letters: 'İ' (U+0130, Latin capital letter I with dot above), 'ı' (U+0131, Latin small letter dotless i), 'ſ' (U+017F, Latin small letter long s) and 'K' (U+212A, Kelvin sign). If the *ASCII* flag is used, only letters 'a' to 'z' and 'A' to 'Z' are matched (but the flag affects the entire regular expression, so in such cases using an explicit (?-i:[a-zA-Z]) may be a better choice).

re.**L**

re.**LOCALE**

由当前语言区域决定 \w, \W, \b, \B 和大小写敏感匹配。这个标记只能对 byte 样式有效。这个标记不推荐使用，因为语言区域机制很不可靠，它一次只能处理一个 "习惯"，而且只对 8 位字节有效。Unicode 匹配在 Python 3 里默认启用，并可以处理不同语言。这个对应内联标记 (?L) 。

在 3.6 版更改: *re.LOCALE* 只能用于 byte 样式，而且不能和*re.ASCII* 一起用。

re.**M**
re.**MULTILINE**
设置以后，样式字符 '^' 匹配字符串的开始，和每一行的开始（换行符后面紧跟的符号）；样式字符 '$' 匹配字符串尾，和每一行的结尾（换行符前面那个符号）。默认情况下，' ^' 匹配字符串头，'$' 匹配字符串尾。对应内联标记（?m）。

re.**S**
re.**DOTALL**
让 '.' 特殊字符匹配任何字符，包括换行符；如果没有这个标记，'.' 就匹配 除了换行符的其他任意字符。对应内联标记（?s）。

re.**X**
re.**VERBOSE**
这个标记允许你编写更具可读性更友好的正则表达式。通过分段和添加注释。空白符号会被忽略，除非在一个字符集合当中或者由反斜杠转义，或者在 *?, (?: or (?P<…> 分组之内。当一个行内有 # 不在字符集和转义序列，那么它之后的所有字符都是注释。

意思就是下面两个正则表达式等价地匹配一个十进制数字：

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

对应内联标记（?x）。

re.**search**(*pattern*, *string*, *flags=0*)
扫描整个 字符串找到匹配样式的第一个位置，并返回一个相应的匹配对象。如果没有匹配，就返回一个 None；注意这和找到一个零长度匹配是不同的。

re.**match**(*pattern*, *string*, *flags=0*)
如果 *string* 开始的 0 或者多个字符匹配到了正则表达式样式，就返回一个相应的匹配对象 。如果没有匹配，就返回 None；注意它跟零长度匹配是不同的。

注意即便是*MULTILINE* 多行模式，*re.match()* 也只匹配字符串的开始位置，而不匹配每行开始。

如果你想定位 *string* 的任何位置，使用*search()* 来替代（也可参考*search() vs. match()* ）

re.**fullmatch**(*pattern*, *string*, *flags=0*)
如果整个 *string* 匹配到正则表达式样式，就返回一个相应的匹配对象 。否则就返回一个 None；注意这跟零长度匹配是不同的。

3.4 新版功能.

re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)
用 *pattern* 分开 *string* 。如果在 *pattern* 中捕获到括号，那么所有的组里的文字也会包含在列表里。如果 *maxsplit* 非零，最多进行 *maxsplit* 次分隔，剩下的字符全部返回到列表的最后一个元素。

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

如果分隔符里有捕获组合，并且匹配到字符串的开始，那么结果将会以一个空字符串开始。对于结尾也是一样

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

这样的话，分隔组将会出现在结果列表中同样的位置。

注解： *split()* doesn't currently split a string on an empty pattern match. For example:

```
>>> re.split('x*', 'axbc')
['a', 'bc']
```

Even though `'x*'` also matches 0 'x' before 'a', between 'b' and 'c', and after 'c', currently these matches are ignored. The correct behavior (i.e. splitting on empty matches too and returning `['', 'a', 'b', 'c', '']`) will be implemented in future versions of Python, but since this is a backward incompatible change, a *FutureWarning* will be raised in the meanwhile.

Patterns that can only match empty strings currently never split the string. Since this doesn't match the expected behavior, a *ValueError* will be raised starting from Python 3.5:

```
>>> re.split("^$", "foo\n\nbar\n", flags=re.M)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
ValueError: split() requires a non-empty pattern match.
```

在 3.1 版更改: 增加了可选标记参数。

在 3.5 版更改: Splitting on a pattern that could match an empty string now raises a warning. Patterns that can only match empty strings are now rejected.

`re.`**`findall`**(*pattern*, *string*, *flags=0*)

对 *string* 返回一个不重复的 *pattern* 的匹配列表，*string* 从左到右进行扫描，匹配按找到的顺序返回。如果样式里存在一到多个组，就返回一个组合列表；就是一个元组的列表（如果样式里有超过一个组合的话）。空匹配也会包含在结果里。

注解： Due to the limitation of the current implementation the character following an empty match is not included in a next match, so `findall(r'^|\w+', 'two words')` returns `['', 'wo', 'words']` (note missed "t"). This is changed in Python 3.7.

`re.`**`finditer`**(*pattern*, *string*, *flags=0*)

Return an *iterator* yielding *match objects* over all non-overlapping matches for the RE *pattern* in *string*. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result. See also the note about *findall()*.

`re.`**`sub`**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

Return the string obtained by replacing the leftmost non-overlapping occurrences of *pattern* in *string* by the replacement *repl*. If the pattern isn't found, *string* is returned unchanged. *repl* can be a string or a function; if it is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes such as `\&` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by group 6 in the pattern. For example:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...         r'static PyObject*\npy_\1(void)\n{',
...         'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

如果 *repl* 是一个函数，那它会对每个非重复的 *pattern* 的情况调用。这个函数只能有一个匹配对象 参数，并返回一个替换后的字符串。比如

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

样式可以是一个字符串或者一个样式对象 。

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. If omitted or zero, all occurrences will be replaced. Empty matches for the pattern are replaced only when not adjacent to a previous match, so `sub('x*', '-', 'abc')` returns `'-a-b-c-'`.

在字符串类型的 *repl* 参数里，如上所述的转义和向后引用中，`\g<name>` 会使用命名组合 `name`，（在 `(?P<name>…)` 语法中定义）`\g<number>` 会使用数字组；`\g<2>` 就是 `\2`，但它避免了二义性，如 `\g<2>0`。`\20` 就会被解释为组 20，而不是组 2 后面跟随一个字符 `'0'`。向后引用 `\g<0>` 把 *pattern* 作为一整个组进行引用。

在 3.1 版更改: 增加了可选标记参数。

在 3.5 版更改: 不匹配的组合替换为空字符串。

在 3.6 版更改: *pattern* 中的未知转义（由 `'\'` 和一个 ASCII 字符组成）被视为错误。

Deprecated since version 3.5, will be removed in version 3.7: Unknown escapes in *repl* consisting of `'\'` and an ASCII letter now raise a deprecation warning and will be forbidden in Python 3.7.

`re.`**`subn`**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

行为与*sub()* 相同，但是返回一个元组（字符串，替换次数）.

在 3.1 版更改: 增加了可选标记参数。

在 3.5 版更改: 不匹配的组合替换为空字符串。

`re.`**`escape`**(*pattern*)

Escape all the characters in *pattern* except ASCII letters, numbers and `'_'`. This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. For example:

```
>>> print(re.escape('python.exe'))
python\.exe

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789\!\#\$\%\&\'\*\+\-\.\^_\`\|\~\:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
\/|\-|\+|\*\*|\*
```

This functions must not be used for the replacement string in *sub()* and *subn()*, only backslashes should be escaped. For example:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

在 3.3 版更改: '_' 不再被转义。

re.**purge**()
清除正则表达式缓存。

exception re.**error**(*msg*, *pattern=None*, *pos=None*)
raise 一个例外。当传递到函数的字符串不是一个有效正则表达式的时候（比如，包含一个不匹配的括号）或者其他错误在编译时或匹配时产生。如果字符串不包含样式匹配，是不会被视为错误的。错误实例有以下附加属性:

msg
未格式化的错误消息。

pattern
正则表达式样式。

pos
编译失败的 *pattern* 的位置索引（可以是 None ）。

lineno
对应 *pos* (可以是 None) 的行号。

colno
对应 *pos* (可以是 None) 的列号。

在 3.5 版更改: 添加了附加属性。

6.2.3 正则表达式对象（正则对象）

编译后的正则表达式对象支持以下方法和属性:

regex.**search**(*string*[, *pos*[, *endpos*]])
扫描整个 *string* 寻找第一个匹配的位置，并返回一个相应的匹配对象。如果没有匹配，就返回 None ；注意它和零长度匹配是不同的。

可选的第二个参数 *pos* 给出了字符串中开始搜索的位置索引；默认为 0，它不完全等价于字符串切片；'^' 样式字符匹配字符串真正的开头，和换行符后面的第一个字符，但不会匹配索引规定开始的位置。

可选参数 *endpos* 限定了字符串搜索的结束；它假定字符串长度到 *endpos* ，所以只有从 pos 到 endpos - 1 的字符会被匹配。如果 *endpos* 小于 *pos*，就不会有匹配产生；另外，如果 *rx* 是一个编译后的正则对象，rx.search(string, 0, 50) 等价于 rx.search(string[:50], 0)。

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<_sre.SRE_Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

regex.**match**(*string*[, *pos*[, *endpos*]])
如果 *string* 的 开始位置能够找到这个正则样式的任意个匹配，就返回一个相应的匹配对象。如果不匹配，就返回 None ；注意它与零长度匹配是不同的。

The optional *pos* and *endpos* parameters have the same meaning as for the *search()* method.

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<_sre.SRE_Match object; span=(1, 2), match='o'>
```

If you want to locate a match anywhere in *string*, use *search()* instead (see also *search() vs. match()*).

regex.**fullmatch**(*string*[, *pos*[, *endpos*]])
如果整个 *string* 匹配这个正则表达式，就返回一个相应的匹配对象。否则就返回 None；注意跟零长度匹配是不同的。

The optional *pos* and *endpos* parameters have the same meaning as for the *search()* method.

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.fullmatch("ogre")     # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<_sre.SRE_Match object; span=(1, 3), match='og'>
```

3.4 新版功能.

regex.**split**(*string*, *maxsplit=0*)
等价于*split()* 函数，使用了编译后的样式。

regex.**findall**(*string*[, *pos*[, *endpos*]])
类似函数*findall()*，使用了编译后样式，但也可以接收可选参数 *pos* 和 *endpos*，限制搜索范围，就像*search()*。

regex.**finditer**(*string*[, *pos*[, *endpos*]])
类似函数 finiter()，使用了编译后样式，但也可以接收可选参数 *pos* 和 *endpos*，限制搜索范围，就像*search()*。

regex.**sub**(*repl*, *string*, *count=0*)
等价于*sub()* 函数，使用了编译后的样式。

regex.**subn**(*repl*, *string*, *count=0*)
等价于*subn()* 函数，使用了编译后的样式。

regex.**flags**
正则匹配标记。这是可以传递给*compile()* 的参数，任何 (?…) 内联标记，隐性标记比如 UNICODE 的结合。

regex.**groups**
捕获组合的数量。

regex.**groupindex**
映射由 (?P<id>) 定义的命名符号组合和数字组合的字典。如果没有符号组，那字典就是空的。

regex.**pattern**
The pattern string from which the RE object was compiled.

6.2.4 匹配对象

Match objects always have a boolean value of True. Since *match()* and *search()* return None when there is no match, you can test whether there was a match with a simple if statement:

```
match = re.search(pattern, string)
if match:
    process(match)
```

匹配对象支持以下方法和属性：

match.**expand**(*template*)
Return the string obtained by doing backslash substitution on the template string *template*, as done by the *sub()* method. Escapes such as \n are converted to the appropriate characters, and numeric backreferences (\1, \2) and named backreferences (\g<1>, \g<name>) are replaced by the contents of the corresponding group.

在 3.5 版更改: 不匹配的组合替换为空字符串。

match.**group**([*group1*, ...])

返回一个或者多个匹配的子组。如果只有一个参数，结果就是一个字符串，如果有多个参数，结果就是一个元组（每个参数对应一个项），如果没有参数，组 1 默认到 0（整个匹配都被返回）。如果一个组 N 参数值为 0，相应的返回值就是整个匹配字符串；如果它是一个范围 [1..99]，结果就是相应的括号组字符串。如果一个组号是负数，或者大于样式中定义的组数，一个*IndexError* 索引错误就 raise。如果一个组包含在样式的一部分，并被匹配多次，就返回最后一个匹配。:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

如果正则表达式使用了 (?P<name>…) 语法，*groupN* 参数就也可能是命名组合的名字。如果一个字符串参数在样式中未定义为组合名，一个*IndexError* 就 raise。

一个相对复杂的例子

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

命名组合同样可以通过索引值引用

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

如果一个组匹配成功多次，就只返回最后一个匹配

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

match.**__getitem__**(*g*)

这个等价于 m.group(g)。这允许更方便的引用一个匹配

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

3.6 新版功能.

match.**groups**(*default=None*)

返回一个元组，包含所有匹配的子组，在样式中出现的从 1 到任意多的组合。*default* 参数用于不参与匹配的情况，默认为 None。

例如

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

如果我们使小数点可选，那么不是所有的组都会参与到匹配当中。这些组合默认会返回一个 None，除非指定了 *default* 参数。

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

match.**groupdict**(*default=None*)
返回一个字典，包含了所有的 命名子组。key 就是组名。*default* 参数用于不参与匹配的组合；默认为 None。例如

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

match.**start**([*group*])
match.**end**([*group*])
返回 *group* 匹配到的字串的开始和结束标号。*group* 默认为 0（意思是整个匹配的子串）。如果 *group* 存在，但未产生匹配，就返回 -1。对于一个匹配对象 *m*，和一个未参与匹配的组 *g*，组 *g* (等价于 m.group(g)) 产生的匹配是

```
m.string[m.start(g):m.end(g)]
```

注意 m.start(group) 将会等于 m.end(group)，如果 *group* 匹配一个空字符串的话。比如，在 m = re.search('b(c?)', 'cba') 之后，m.start(0) 为 1, m.end(0) 为 2, m.start(1) 和 m.end(1) 都是 2, m.start(2) raise 一个 *IndexError* 例外。

这个例子会从 email 地址中移除掉 *remove_this*

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

match.**span**([*group*])
对于一个匹配 *m*，返回一个二元组 (m.start(group), m.end(group))。注意如果 *group* 没有在这个匹配中，就返回 (-1, -1)。*group* 默认为 0，就是整个匹配。

match.**pos**
The value of *pos* which was passed to the *search()* or *match()* method of a *regex object*. This is the index into the string at which the RE engine started looking for a match.

match.**endpos**
The value of *endpos* which was passed to the *search()* or *match()* method of a *regex object*. This is the index into the string beyond which the RE engine will not go.

match.**lastindex**
捕获组的最后一个匹配的整数索引值，或者 None 如果没有匹配产生的话。比如，对于字符串 'ab'，表达式 (a)b, ((a)(b)), 和 ((ab)) 将得到 lastindex == 1，而 (a)(b) 会得到 lastindex == 2。

match.**lastgroup**
最后一个匹配的命名组名字，或者 None 如果没有产生匹配的话。

match.**re**
The *regular expression object* whose *match()* or *search()* method produced this match instance.

match.**string**
The string passed to *match()* or *search()*.

6.2.5 正则表达式例子

检查对子

在这个例子里，我们使用以下辅助函数来更好的显示匹配对象：

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

假设你在写一个扑克程序，一个玩家的一手牌为五个字符的串，每个字符表示一张牌，”a”就是 A，“k”K，“q”Q，“j”J，“t”为 10，“2”到“9”表示 2 到 9。

要看给定的字符串是否有效，我们可以按照以下步骤

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

最后一手牌，"727ak"，包含了一个对子，或者两张同样数值的牌。要用正则表达式匹配它，应该使用向后引用如下

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
>>> displaymatch(pair.match("718ak"))     # No pairs.
>>> displaymatch(pair.match("354aa"))     # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

To find out what card the pair consists of, one could use the *group()* method of the match object in the following manner:

```
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'
```

(下页继续)

(续上页)

```
>>> pair.match("354aa").group(1)
'a'
```

模拟 scanf()

Python 目前没有一个类似 c 函数 `scanf()` 的替代品。正则表达式通常比 `scanf()` 格式字符串要更强大一些，但也带来更多复杂性。下面的表格提供了 `scanf()` 格式符和正则表达式大致相同的映射。

scanf() 格式符	正则表达式
%c	.
%5c	.{5}
%d	[-+]?\d+
%e, %E, %f, %g	[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
%i	[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)
%o	[-+]?[0-7]+
%s	\S+
%u	\d+
%x, %X	[-+]?(0[xX])?[\dA-Fa-f]+

从文件名和数字提取字符串

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

你可以使用 `scanf()` 格式化

```
%s - %d errors, %d warnings
```

等价的正则表达式是：

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python 提供了两种不同的操作：基于*re.match()* 检查字符串开头，或者*re.search()* 检查字符串的任意位置（默认 Perl 中的行为）。

例如

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<_sre.SRE_Match object; span=(2, 3), match='c'>
```

在*search()* 中，可以用 `'^'` 作为开始来限制匹配到字符串的首位

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<_sre.SRE_Match object; span=(0, 1), match='a'>
```

注意*MULTILINE* 多行模式中函数*match()* 只匹配字符串的开始，但使用*search()* 和以 `'^'` 开始的正则表达式会匹配每行的开始

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE)  # No match
>>> re.search('^X', 'A\nB\nX', re.MULTILINE)  # Match
<_sre.SRE_Match object; span=(4, 5), match='X'>
```

建立一个电话本

split() 将字符串用参数传递的样式分隔开。这个方法对于转换文本数据到易读而且容易修改的数据结构，是很有用的，如下面的例子证明。

首先，这里是输入。通常是一个文件，这里我们用三引号字符串语法

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

条目用一个或者多个换行符分开。现在我们将字符串转换为一个列表，每个非空行都有一个条目:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

最终，将每个条目分割为一个由名字、姓氏、电话号码和地址组成的列表。我们为*split()* 使用了 maxsplit 形参，因为地址中包含有被我们作为分割模式的空格符:

```
>>> [re.split(":? ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

:? 样式匹配姓后面的冒号，因此它不出现在结果列表中。如果 maxsplit 设置为 4 ，我们还可以从地址中获取到房间号:

```
>>> [re.split(":? ", entry, 4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

文字整理

sub() 替换字符串中出现的样式的每一个实例。这个例子证明了使用*sub()* 来整理文字，或者随机化每个字符的位置，除了首位和末尾字符

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

找到所有副词

findall() 匹配样式 所有的出现，不仅是像*search()* 中的第一个匹配。比如，如果一个作者希望找到文字中的所有副词，他可能会按照以下方法用*findall()*

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly", text)
['carefully', 'quickly']
```

找到所有副词和位置

如果需要匹配样式的更多信息，*finditer()* 可以起到作用，它提供了匹配对象 作为返回值，而不是字符串。继续上面的例子，如果一个作者希望找到所有副词和它的位置，可以按照下面方法使用*finditer()*

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

原始字符记法

原始字符串记法 (`r"text"`) 保持正则表达式正常。否则，每个正则式里的反斜杠 (`'\'`) 都必须前缀一个反斜杠来转义。比如，下面两行代码功能就是完全一致的

```
>>> re.match(r"\W(.)\1\W", " ff ")
<_sre.SRE_Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<_sre.SRE_Match object; span=(0, 4), match=' ff '>
```

当需要匹配一个字符反斜杠，它必须在正则表达式中转义。在原始字符串记法，就是 `r"\\"`。否则就必须用 `"\\\\"`，来表示同样的意思

```
>>> re.match(r"\\", r"\\")
<_sre.SRE_Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<_sre.SRE_Match object; span=(0, 1), match='\\'>
```

写一个词法分析器

一个 词法器或词法分析器 分析字符串，并分类成目录组。这是写一个编译器或解释器的第一步。

文字目录是由正则表达式指定的。这个技术是通过将这些样式合并为一个主正则式，并且循环匹配来实现的

```
import collections
import re

Token = collections.namedtuple('Token', ['type', 'value', 'line', 'column'])

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

这个词法器产生以下输出

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
```

(下页继续)

(续上页)

```
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

6.3 difflib —计算差异的辅助工具

源代码: Lib/difflib.py

此模块提供用于比较序列的类和函数。例如，它可以用于比较文件，并可以产生各种格式的不同信息，包括 HTML 和上下文以及统一格式的差异点。有关目录和文件的比较，请参见*filecmp* 模块。

class difflib.**SequenceMatcher**

这是一个灵活的类，可用于比较任何类型的序列对，只要序列元素为*hashable* 对象。其基本算法要早于由 Ratcliff 和 Obershelp 于 1980 年代末期发表并以“格式塔模式匹配”的夸张名称命名的算法，并且更加有趣一些。其思路是找到不包含“垃圾”元素的最长连续匹配子序列；所谓“垃圾”元素是指其在某种意义上没有价值，例如空白行或空白符。(处理垃圾元素是对 Ratcliff 和 Obershelp 算法的一个扩展。)然后同样的思路将递归地应用于匹配序列的左右序列片段。这并不能产生最小编辑序列，但确实能产生在人们看来“正确”的匹配。

耗时: 基本 Ratcliff-Obershelp 算法在最坏情况下为立方时间而在一般情况下为平方时间。*SequenceMatcher* 在最坏情况下为平方时间而在一般情况下的行为受到序列中有多少相同元素这一因素的微妙影响；在最佳情况下则为线性时间。

自动垃圾启发式计算: *SequenceMatcher* 支持使用启发式计算来自动将特定序列项视为垃圾。这种启发式计算会统计每个单独项在序列中出现的次数。如果某一项（在第一项之后）的重复次数超过序列长度的 1% 并且序列长度至少有 200 项，该项会被标记为“热门”并被视为序列匹配中的垃圾。这种启发式计算可以通过在创建*SequenceMatcher* 时将 autojunk 参数设为 False 来关闭。

3.2 新版功能: *autojunk* 形参。

class difflib.**Differ**

这个类的作用是比较由文本行组成的序列，并产生可供人阅读的差异或增量信息。Differ 统一使用*SequenceMatcher* 来完成行序列的比较以及相似（接近匹配）行内部字符序列的比较。

Differ 增量的每一行均以双字母代码打头：

双字母代码	含义
'- '	行为序列 1 所独有
'+ '	行为序列 2 所独有
' '	行在两序列中相同
'? '	行不存在于任一输入序列

以‘?’打头的行尝试将视线引至行以外而不存在于任一输入序列的差异。如果序列包含制表符则这些行可能会令人感到迷惑。

class difflib.**HtmlDiff**

这个类可用于创建 HTML 表格（或包含表格的完整 HTML 文件）以并排地逐行显示文本比较，行间与行外的更改将突出显示。此表格可以基于完全或上下文差异模式来生成。

这个类的构造函数：

__init__(*tabsize=8*, *wrapcolumn=None*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

初始化*HtmlDiff* 的实例。

tabsize 是一个可选关键字参数，指定制表位的间隔，默认值为 8。

wrapcolumn 是一个可选关键字参数，指定行文本自动打断并换行的列位置，默认值为 None 表示不自动换行。

linejunk 和 *charjunk* 均是可选关键字参数，会传入*ndiff()* (被*HtmlDiff* 用来生成并排显示的 HTML 差异)。请参阅*ndiff()* 文档了解参数默认值及其说明。

下列是公开的方法

make_file(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*, ***, *charset='utf-8'*)

比较 *fromlines* 和 *tolines* (字符串列表) 并返回一个字符串，表示一个完整 HTML 文件，其中包含各行差异的表格，行间与行外的更改将突出显示。

fromdesc 和 *todesc* 均是可选关键字参数，指定来源/目标文件的列标题字符串（默认均为空白字符串）。

context 和 *numlines* 均是可选关键字参数。当只要显示上下文差异时就将 *context* 设为 True，否则默认值 False 为显示完整文件。*numlines* 默认为 5。当 *context* 为 True 时 *numlines* 将控制围绕突出显示差异部分的上下文行数。当 *context* 为 False 时 *numlines* 将控制在使用“next”超链接时突出显示差异部分之前所显示的行数（设为零则会导致“next”超链接将下一个突出显示差异部分放在浏览器顶端，不添加任何前导上下文）。

在 3.5 版更改: 增加了 *charset* 关键字参数。HTML 文档的默认字符集从 'ISO-8859-1' 更改为 'utf-8'。

make_table(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*)

比较 *fromlines* 和 *tolines* (字符串列表) 并返回一个字符串，表示一个包含各行差异的完整 HTML 表格，行间与行外的更改将突出显示。

此方法的参数与*make_file()* 方法的相同。

Tools/scripts/diff.py 是这个类的命令行前端，其中包含一个很好的使用示例。

difflib.**context_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

比较 *a* 和 *b* (字符串列表)；返回上下文差异格式的增量信息 (一个产生增量行的*generator*)。

所谓上下文差异是一种只显示有更改的行再加几个上下文行的紧凑形式。更改被显示为之前/之后的样式。上下文行数由 *n* 设定，默认为三行。

默认情况下，差异控制行（以 *** or --- 表示）是通过末尾换行符来创建的。这样做的好处是从*io.IOBase.readlines()* 创建的输入将得到适用于*io.IOBase.writelines()* 的差异信息，因为输入和输出都带有末尾换行符。

对于没有末尾换行符的输入，应将 *lineterm* 参数设为 `""`，这样输出内容将统一不带换行符。

上下文差异格式通常带有一个记录文件名和修改时间的标头。这些信息的部分或全部可以使用字符串 *fromfile*, *tofile*, *fromfiledate* 和 *tofiledate* 来指定。修改时间通常以 ISO 8601 格式表示。如果未指定，这些字符串默认为空。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py', tofile=
↪'after.py'))
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

请参阅*difflib 的命令行接口* 获取更详细的示例。

`difflib.`**`get_close_matches`**(*word*, *possibilities*, *n=3*, *cutoff=0.6*)

返回由最佳“近似”匹配构成的列表。*word* 为一个指定目标近似匹配的序列（通常为字符串），*possibilities* 为一个由用于匹配 *word* 的序列构成的列表（通常为字符串列表）。

可选参数 *n* (默认为 3) 指定最多返回多少个近似匹配；*n* 必须大于 0.

可选参数 *cutoff* (默认为 `0.6`) 是一个 [0, 1] 范围内的浮点数。与 *word* 相似度得分未达到该值的候选匹配将被忽略。

候选匹配中（不超过 *n* 个）的最佳匹配将以列表形式返回，按相似度得分排序，最相似的排在最前面。

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

`difflib.`**`ndiff`**(*a*, *b*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

比较 *a* 和 *b* (字符串列表)；返回*Differ* 形式的增量信息 (一个产生增量行的*generator*)。

可选关键字形参 *linejunk* 和 *charjunk* 均为过滤函数 (或为 `None`)：

linejunk: 此函数接受单个字符串参数，如果其为垃圾字符串则返回真值，否则返回假值。默认为 `None`。此外还有一个模块层级的函数*IS_LINE_JUNK()*，它会过滤掉没有可见字符的行，除非该行添加了至多一个井号符 (`'#'`)–但是下层的*SequenceMatcher* 类会动态分析哪些行的重复频繁到足以形成噪音，这通常会比使用此函数的效果更好。

charjunk: 此函数接受一个字符（长度为 1 的字符串)，如果其为垃圾字符则返回真值，否则返回假值。默认为模块层级的函数*IS_CHARACTER_JUNK()*，它会过滤掉空白字符（空格符或制表符；但包含换行符可不是个好主意!)。

Tools/scripts/ndiff.py 是这个函数的命令行前端。

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.**restore**(*sequence*, *which*)

返回两个序列中产生增量的那一个。

给出一个由*Differ.compare()* 或*ndiff()* 产生的 序列，提取出来自文件 1 或 2 (*which* 形参) 的行，去除行前缀。

示例:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end="")
one
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
tree
emu
```

difflib.**unified_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)

比较 *a* 和 *b* (字符串列表)；返回统一差异格式的增量信息 (一个产生增量行的*generator*)。

所以统一差异是一种只显示有更改的行再加几个上下文行的紧凑形式。更改被显示为内联的样式（而不是分开的之前/之后文本块)。上下文行数由 *n* 设定，默认为三行。

默认情况下，差异控制行 (以 ---, +++ 或 @@ 表示) 是通过末尾换行符来创建的。这样做的好处是从*io.IOBase.readlines()* 创建的输入将得到适用于*io.IOBase.writelines()* 的差异信息，因为输入和输出都带有末尾换行符。

对于没有末尾换行符的输入，应将 *lineterm* 参数设为 `""`，这样输出内容将统一不带换行符。

上下文差异格式通常带有一个记录文件名和修改时间的标头。这些信息的部分或全部可以使用字符串 *fromfile*, *tofile*, *fromfiledate* 和 *tofiledate* 来指定。修改时间通常以 ISO 8601 格式表示。如果未指定，这些字符串默认为空。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile=
↪'after.py'))
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
```

(下页继续)

(续上页)

```
-eggs
-ham
+python
+eggy
+hamster
 guido
```

请参阅*difflib 的命令行接口* 获取更详细的示例。

difflib.**diff_bytes**(*dfunc*, *a*, *b*, *fromfile=b''*, *tofile=b''*, *fromfiledate=b''*, *tofiledate=b''*, *n=3*, *lineterm=b'\n'*)

使用 *dfunc* 比较 *a* 和 *b* (字节串对象列表)；产生以 *dfunc* 所返回格式表示的差异行列表（也是字节串）。*dfunc* 必须是可调用对象，通常为`unified_diff()` 或`context_diff()`。

允许你比较编码未知或不一致的数据。除 *n* 之外的所有输入都必须为字节串对象而非字符串。作用方式为无损地将所有输入 (除 *n* 之外) 转换为字符串，并调用 `dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`。*dfunc* 的输出会被随即转换回字节串，这样你所得到的增量行将具有与 *a* 和 *b* 相同的未知/不一致编码。

3.5 新版功能.

difflib.**IS_LINE_JUNK**(*line*)

Return true for ignorable lines. The line *line* is ignorable if *line* is blank or contains a single `'#'`, otherwise it is not ignorable. Used as a default for parameter *linejunk* in `ndiff()` in older versions.

difflib.**IS_CHARACTER_JUNK**(*ch*)

Return true for ignorable characters. The character *ch* is ignorable if *ch* is a space or tab, otherwise it is not ignorable. Used as a default for parameter *charjunk* in `ndiff()`.

参见:

模式匹配：格式塔方法 John W. Ratcliff 和 D. E. Metzener 对于一种类似算法的讨论。此文于 1988 年 7 月发表于 Dr. Dobb's Journal。

6.3.1 SequenceMatcher 对象

`SequenceMatcher` 类具有这样的构造器：

class difflib.**SequenceMatcher**(*isjunk=None*, *a=''*, *b=''*, *autojunk=True*)

Optional argument *isjunk* must be `None` (the default) or a one-argument function that takes a sequence element and returns true if and only if the element is "junk" and should be ignored. Passing `None` for *isjunk* is equivalent to passing `lambda x: 0`; in other words, no elements are ignored. For example, pass:

```
lambda x: x in " \t"
```

如果你以字符序列的形式对行进行比较，并且不希望区分空格符或硬制表符。

可选参数 *a* 和 *b* 为要比较的序列；两者默认为空字符串。两个序列的元素都必须为*hashable*。

可选参数 *autojunk* 可用于启用自动垃圾启发式计算。

3.2 新版功能: *autojunk* 形参。

SequenceMatcher 对象接受三个数据属性: *bjunk* 是 *b* 当中 *isjunk* 为 `True` 的元素集合；*bpopular* 是被启发式计算（如果其未被禁用）视为热门候选的非垃圾元素集合；*b2j* 是将 *b* 当中剩余元素映射到一个它们出现位置列表的字典。所有三个数据属性将在 *b* 通过`set_seqs()` 或`set_seq2()` 重置时被重置。

3.2 新版功能: *bjunk* 和 *bpopular* 属性。

`SequenceMatcher` 对象具有以下方法：

set_seqs(*a*, *b*)
设置要比较的两个序列。

SequenceMatcher 计算并缓存有关第二个序列的详细信息，这样如果你想要将一个序列与多个序列进行比较，可使用*set_seq2()* 一次性地设置该常用序列并重复地对每个其他序列各调用一次*set_seq1()*。

set_seq1(*a*)
设置要比较的第一个序列。要比较的第二个序列不会改变。

set_seq2(*b*)
设置要比较的第二个序列。要比较的第一个序列不会改变。

find_longest_match(*alo*, *ahi*, *blo*, *bhi*)
找出 a[alo:ahi] 和 b[blo:bhi] 中的最长匹配块。

如果 *isjunk* 被省略或为 None，*find_longest_match()* 将返回 (i, j, k) 使得 a[i:i+k] 等于 b[j:j+k]，其中 alo <= i <= i+k <= ahi 并且 blo <= j <= j+k <= bhi。对于所有满足这些条件的 (i', j', k')，如果 i == i', j <= j' 也被满足，则附加条件 k >= k', i <= i'。换句话说，对于所有最长匹配块，返回在 *a* 当中最先出现的一个，而对于在 *a* 当中最先出现的所有最长匹配块，则返回在 *b* 当中最先出现的一个。

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

如果提供了 *isjunk*，将按上述规则确定第一个最长匹配块，但额外附加不允许块内出现垃圾元素的限制。然后将通过（仅）匹配两边的垃圾元素来尽可能地扩展该块。这样结果块绝对不会匹配垃圾元素，除非同样的垃圾元素正好与有意义的匹配相邻。

这是与之前相同的例子，但是将空格符视为垃圾。这将防止 ' abcd' 直接与第二个序列末尾的 ' abcd' 相匹配。而只可以匹配 'abcd'，并且是匹配第二个序列最左边的 'abcd'：

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

如果未找到匹配块，此方法将返回 (alo, blo, 0)。

此方法将返回一个*named tuple* Match(a, b, size)。

get_matching_blocks()
返回描述非重叠匹配子序列的三元组列表。每个三元组的形式为 (i, j, n)，其含义为 a[i:i+n] == b[j:j+n]。这些三元组按 *i* 和 *j* 单调递增排列。

最后一个三元组用于占位，其值为 (len(a), len(b), 0)。它是唯一 n == 0 的三元组。如果 (i, j, n) 和 (i', j', n') 是在列表中相邻的三元组，且后者不是列表中的最后一个三元组，则 i+n < i' 或 j+n < j'；换句话说，相邻的三元组总是描述非相邻的相等块。

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()
返回描述如何将 *a* 变为 *b* 的 5 元组列表，每个元组的形式为 (tag, i1, i2, j1, j2)。在第一个元组中 i1 == j1 == 0，而在其余的元组中 *i1* 等于前一个元组的 *i2*，并且 *j1* 也等于前一个元组的 *j2*。

tag 值为字符串，其含义如下：

值	含义
'replace'	a[i1:i2] 应由 b[j1:j2] 替换。
'delete'	a[i1:i2] 应被删除。请注意在此情况下 j1 == j2。
'insert'	b[j1:j2] 应插入到 a[i1:i1]。请注意在此情况下 i1 == i2。
'equal'	a[i1:i2] == b[j1:j2] (两个子序列相同)。

例如

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]     'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]     'cd' --> 'cd'
insert    a[6:6] --> b[5:6]       '' --> 'f'
```

get_grouped_opcodes(*n=3*)

返回一个带有最多 *n* 行上下文的分组的*generator*。

从*get_opcodes()* 所返回的组开始，此方法会拆分出较小的更改簇并消除没有更改的间隔区域。

这些分组以与*get_opcodes()* 相同的格式返回。

ratio()

返回一个取值范围 [0, 1] 的浮点数作为序列相似性度量。

其中 T 是两个序列中元素的总数量，M 是匹配的数量，即 2.0*M / T。请注意如果两个序列完全相同则该值为 1.0，如果两者完全不同则为 0.0。

如果*get_matching_blocks()* 或*get_opcodes()* 尚未被调用则此方法运算消耗较大，在此情况下你可能需要先调用*quick_ratio()* 或*real_quick_ratio()* 来获取一个上界。

quick_ratio()

相对快速地返回一个*ratio()* 的上界。

real_quick_ratio()

非常快速地返回一个*ratio()* 的上界。

这三个返回匹配部分占字符总数的比率的方法可能由于不同的近似级别而给出不一样的结果，但是 quick_ratio() 和 real_quick_ratio() 总是会至少与 ratio() 一样大:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.3.2 SequenceMatcher 的示例

以下示例比较两个字符串，并将空格视为“垃圾”：

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

ratio() 返回一个 [0, 1] 范围内的整数作为两个序列相似性的度量。根据经验，ratio() 值超过 0.6 就意味着两个序列是近似匹配的：

```
>>> print(round(s.ratio(), 3))
0.866
```

如果你只对两个序列相匹配的位置感兴趣，则 get_matching_blocks() 就很方便：

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

请注意 get_matching_blocks() 返回的最后一个元组总是只用于占位的 (len(a), len(b), 0)，这也是元组末尾元素（匹配的元素数量）为 0 的唯一情况。

如果你想要知道如何将第一个序列转成第二个序列，可以使用 get_opcodes()：

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

参见：

- 此模块中的*get_close_matches()* 函数显示了如何基于*SequenceMatcher* 构建简单的代码来执行有用的功能。
- 使用*SequenceMatcher* 构建小型应用的 简易版本控制方案。

6.3.3 Differ 对象

请注意*Differ* 所生成的增量并不保证是 **最小**差异。相反，最小差异往往是违反直觉的，因为它们会同步任何可能的地方，有时甚至意外产生相距 100 页的匹配。将同步点限制为连续匹配保留了一些局部性概念，这偶尔会带来产生更长差异的代价。

Differ 类具有这样的构造器：

class difflib.**Differ**(*linejunk=None*, *charjunk=None*)
可选关键字形参 *linejunk* 和 *charjunk* 均为过滤函数 (或为 None)：

linejunk: 接受单个字符串作为参数的函数，如果其为垃圾字符串则返回真值。默认值为 None，意味着没有任何行会被视为垃圾行。

charjunk: 接受单个字符（长度为 1 的字符串）作为参数的函数，如果其为垃圾字符则返回真值。默认值为 None，意味着没有任何字符会被视为垃圾字符。

这些垃圾过滤函数可加快查找差异的匹配速度，并且不会导致任何差异行或字符被忽略。请阅读*find_longest_match()* 方法的 *isjunk* 形参的描述了解详情。

Differ 对象是通过一个单独方法来使用（生成增量）的：

compare(*a*, *b*)
比较两个由行组成的序列，并生成增量（一个由行组成的序列）。

每个序列必须包含一个以换行符结尾的单行字符串。这样的序列可以通过文件类对象的*readlines()* 方法来获取。所生成的增量同样由以换行符结尾的字符串构成，可以通过文件类对象的*writelines()* 方法原样打印出来。

6.3.4 Differ 示例

此示例比较两段文本。首先我们设置文本为以换行符结尾的单行字符串构成的序列（这样的序列也可以通过文件类对象的 `readlines()` 方法来获取）：

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(keepends=True)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
...   3.   Simple is better than complex.
...   4. Complicated is better than complex.
...   5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

接下来我们实例化一个 Differ 对象：

```
>>> d = Differ()
```

请注意在实例化*Differ* 对象时我们可以传入函数来过滤掉“垃圾”行和字符。详情参见*Differ()* 构造器说明。

最后，我们比较两个序列：

```
>>> result = list(d.compare(text1, text2))
```

`result` 是一个字符串列表，让我们将其美化打印出来：

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
 '+   4. Complicated is better than complex.\n',
 '?           ++++ ^                      ^\n',
 '+   5. Flat is better than nested.\n']
```

作为单独的多行字符串显示出来则是这样：

```
>>> import sys
>>> sys.stdout.writelines(result)
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?            ^                     ---- ^
+   4. Complicated is better than complex.
?           ++++ ^                      ^
+   5. Flat is better than nested.
```

6.3.5 difflib 的命令行接口

这个实例演示了如何使用 difflib 来创建一个类似于 `diff` 的工具。它同样包含在 Python 源码发布包中，文件名为 `Tools/scripts/diff.py`。

```
#!/usr/bin/env python3
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                               timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
    options = parser.parse_args()

    n = options.lines
```

(下页继续)

(续上页)

```
    fromfile = options.fromfile
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
    with open(fromfile) as ff:
        fromlines = ff.readlines()
    with open(tofile) as tf:
        tolines = tf.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate,␣
↪todate, n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
        diff = difflib.HtmlDiff().make_file(fromlines,tolines,fromfile,tofile,
↪context=options.c,numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate,␣
↪todate, n=n)

    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

6.4 textwrap —文本自动换行与填充

源代码: Lib/textwrap.py

---

textwrap 模块提供了一些快捷函数，以及可以完成所有工作的类*TextWrapper*。如果你只是要对一两个文本字符串进行自动换行或填充，快捷函数应该就够用了；否则的话，你应该使用*TextWrapper* 的实例来提高效率。

textwrap.**wrap**(*text*, *width=70*, ***kwargs*)
对 *text* (字符串) 中的单独段落自动换行以使每行长度最多为 *width* 个字符。返回由输出行组成的列表，行尾不带换行符。

可选的关键字参数对应于*TextWrapper* 的实例属性，具体文档见下。*width* 默认为 `70`。

请参阅*TextWrapper.wrap()* 方法了解有关*wrap()* 行为的详细信息。

textwrap.**fill**(*text*, *width=70*, ***kwargs*)
对 *text* 中的单独段落自动换行，并返回一个包含被自动换行段落的单独字符串。*fill()* 是以下语句的快捷方式

```
"\n".join(wrap(text, ...))
```

特别要说明的是，*fill()* 接受与*wrap()* 完全相同的关键字参数。

textwrap.**shorten**(*text*, *width*, ***kwargs*)
折叠并截短给定的 *text* 以符合给定的 *width*。

首先将折叠 *text* 中的空格（所有连续空格替换为单个空格）。如果结果能适合 *width* 则将其返回。否则将丢弃足够数量的末尾单词以使得剩余单词加 `placeholder` 能适合 `width`:

```
>>> textwrap.shorten("Hello  world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello  world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

可选的关键字参数对应于 *TextWrapper* 的实际属性，具体见下文。请注意文本在被传入 *TextWrapper* 的 *fill()* 函数之前会被折叠，因此改变 *tabsize*, *expand_tabs*, *drop_whitespace* 和 *replace_whitespace* 的值将没有任何效果。

3.4 新版功能.

`textwrap.`**`dedent`**(*text*)

移除 *text* 中每一行的任何相同前缀空白符。

这可以用来清除三重引号字符串行左侧空格，而仍然在源码中显示为缩进格式。

请注意制表符和空格符都被视为是空白符，但它们并不相等：以下两行 `"  hello"` 和 `"\thello"` 不会被视为具有相同的前缀空白符。

例如

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print(repr(s))          # prints '    hello\n      world\n    '
    print(repr(dedent(s)))  # prints 'hello\n  world\n'
```

`textwrap.`**`indent`**(*text*, *prefix*, *predicate=None*)

将 *prefix* 添加到 *text* 中选定行的开头。

通过调用 `text.splitlines(True)` 来对行进行拆分。

默认情况下，*prefix* 会被添加到所有不是只由空白符（包括任何行结束符）组成的行。

例如

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, '  ')
'  hello\n\n \n  world'
```

可选的 *predicate* 参数可用来控制哪些行要缩进。例如，可以很容易地为空行或只有空白符的行添加 *prefix*:

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

3.3 新版功能.

wrap(), *fill()* 和*shorten()* 的作用方式为创建一个*TextWrapper* 实例并在其上调用单个方法。该实例不会被重用，因此对于要使用*wrap()* 和/或*fill()* 来处理许多文本字符串的应用来说，创建你自己的*TextWrapper* 对象可能会更有效率。

文本最好在空白符位置自动换行，包括带连字符单词的连字符之后；长单词仅在必要时会被拆分，除非*TextWrapper.break_long_words* 被设为假值。

class textwrap.**TextWrapper**(***kwargs*)

TextWrapper 构造器接受多个可选的关键字参数。每个关键字参数对应一个实例属性，比如说

```
wrapper = TextWrapper(initial_indent="* ")
```

就相当于

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

你可以多次重用相同的*TextWrapper* 对象，并且你也可以在使用期间通过直接向实例属性赋值来修改它的任何选项。

TextWrapper 的实例属性（以及构造器的关键字参数）如下所示：

width

(默认: `70`) 自动换行的最大行长度。只要输入文本中没有长于*width* 的单个单词，*TextWrapper* 就能保证没有长于*width* 个字符的输出行。

expand_tabs

(默认: `True`) 如果为真值，则 *text* 中所有的制表符将使用 *text* 的 `expandtabs()` 方法扩展为空格符。

tabsize

(默认: `8`) 如果*expand_tabs* 为真值，则 *text* 中所有的制表符将扩展为零个或多个空格，具体取决于当前列位置和给定的制表宽度。

3.3 新版功能.

replace_whitespace

(default: `True`) 如果为真值，在制表符扩展之后、自动换行之前，*wrap()* 方法将把每个空白字符都替换为单个空格。会被替换的空白字符如下：制表，换行，垂直制表，进纸和回车 (`'\t\n\v\f\r'`)。

注解： 如果*expand_tabs* 为假值且*replace_whitespace* 为真值，每个制表符将被替换为单个空格，这与制表符扩展是 不一样的。

注解： 如果*replace_whitespace* 为假值，在一行的中间有可能出现换行符并导致怪异的输出。因此，文本应当（使用*str.splitlines()* 或类似方法）拆分为段落并分别进行自动换行。

---

drop_whitespace

(默认: `True`) 如果为真值，每一行开头和末尾的空白字符（在包装之后、缩进之前）会被丢弃。但是段落开头的空白字符如果后面不带任何非空白字符则不会被丢弃。如果被丢弃的空白字符占据了一个整行，则该整行将被丢弃。

initial_indent

(默认: `''`) 将被添加到被自动换行输出内容的第一行的字符串。其长度会被计入第一行的长度。空字符串不会被缩进。

subsequent_indent
(default: `''`) 将被添加到被自动换行输出内容除第一行外的所有行的字符串。其长度会被计入除行一行外的所有行的长度。

fix_sentence_endings
(默认: `False`) 如果为真值，*TextWrapper* 将尝试检测句子结尾并确保句子间总是以恰好两个空格符分隔。对于使用等宽字体的文本来说通常都需要这样。但是，句子检测算法并不完美：它假定句子结尾是一个小写字母加字符 `'.'`, `'!'` 或 `'?'` 中的一个，并可能带有字符 `'"'` 或 `"'"`，最后以一个空格结束。此算法的问题之一是它无法区分以下文本中的 “Dr.”

```
[...] Dr. Frankenstein's monster [...]
```

和以下文本中的 “Spot.”

```
[...] See Spot. See Spot run [...]
```

fix_sentence_endings 默认为假值。

Since the sentence detection algorithm relies on `string.lowercase` for the definition of “lowercase letter,” and a convention of using two spaces after a period to separate sentences on the same line, it is specific to English-language texts.

break_long_words
(默认: `True`) 如果为真值，则长度超过*width* 的单词将被分开以保证行的长度不会超过*width*。如果为假值，超长单词不会被分开，因而某些行的长度可能会超过*width*。(超长单词将被单独作为一行，以尽量减少超出*width* 的情况。)

break_on_hyphens
(默认: `True`) 如果为真值，将根据英语的惯例首选在空白符和复合词的连字符之后自动换行。如果为假值，则只有空白符会被视为合适的潜在断行位置，但如果你确实不希望出现分开的单词则你必须将*break_long_words* 设为假值。之前版本的默认行为总是允许分开带有连字符的单词。

max_lines
(默认: `None`) 如果不为 `None`，则输出内容将最多包含 *max_lines* 行，并使 *placeholder* 出现在输出内容的末尾。

3.4 新版功能.

placeholder
(默认: `' [...]'`) 该文本将在输出文本被截短时出现在文本末尾。

3.4 新版功能.

TextWrapper 还提供了一些公有方法，类似于模块层级的便捷函数:

wrap(*text*)
对 *text* (字符串) 中的单独段落自动换行以使每行长度最多为*width* 个字符。所有自动换行选项均获取自*TextWrapper* 实例的实例属性。返回由输出行组成的列表，行尾不带换行符。如果自动换行输出结果没有任何内容，则返回空列表。

fill(*text*)
对 *text* 中的单独段落自动换行并返回包含被自动换行段落的单独字符串。

6.5 unicodedata —Unicode 数据库

This module provides access to the Unicode Character Database (UCD) which defines character properties for all Unicode characters. The data contained in this database is compiled from the UCD version 9.0.0.

The module uses the same names and symbols as defined by Unicode Standard Annex #44, "Unicode Character Database". It defines the following functions:

unicodedata.**lookup**(*name*)
按名称查找字符。如果找到具有给定名称的字符，则返回相应的字符。如果没有找到，则*KeyError*被引发。

在 3.3 版更改: 已添加对名称别名[1] 和命名序列[2] 的支持。

unicodedata.**name**(*chr*[, *default*])
返回分配给字符 *chr* 的名称作为字符串。如果没有定义名称，则返回 *default* ，如果没有给出，则*ValueError*被引发。

unicodedata.**decimal**(*chr*[, *default*])
返回分配给字符 *chr* 的十进制值作为整数。如果没有定义这样的值，则返回 *default* ，如果没有给出，则*ValueError*被引发。

unicodedata.**digit**(*chr*[, *default*])
返回分配给字符 *chr* 的数字值作为整数。如果没有定义这样的值，则返回 *default* ，如果没有给出，则*ValueError*被引发。

unicodedata.**numeric**(*chr*[, *default*])
返回分配给字符 *chr* 的数值作为浮点数。如果没有定义这样的值，则返回 *default* ，如果没有给出，则*ValueError*被引发。

unicodedata.**category**(*chr*)
返回分配给字符 *chr* 的常规类别为字符串。

unicodedata.**bidirectional**(*chr*)
返回分配给字符 *chr* 的双向类作为字符串。如果未定义此类值，则返回空字符串。

unicodedata.**combining**(*chr*)
返回分配给字符 *chr* 的规范组合类作为整数。如果没有定义组合类，则返回 `0` 。

unicodedata.**east_asian_width**(*chr*)
返回分配给字符 *chr* 的东亚宽度作为字符串。

unicodedata.**mirrored**(*chr*)
返回分配给字符 *chr* 的镜像属性为整数。如果字符在双向文本中被识别为"镜像"字符，则返回 `1` ，否则返回 `0` 。

unicodedata.**decomposition**(*chr*)
返回分配给字符 *chr* 的字符分解映射作为字符串。如果未定义此类映射，则返回空字符串。

unicodedata.**normalize**(*form*, *unistr*)
返回 Unicode 字符串 *unistr* 的正常形式 *form* 。*form* 的有效值为'NFC'、'NFKC'、'NFD'和'NFKD'。

Unicode 标准基于规范等价和兼容性等效的定义定义了 Unicode 字符串的各种规范化形式。在 Unicode 中，可以以各种方式表示多个字符。例如，字符 U+00C7（带有 CEDILLA 的 LATIN CAPITAL LETTER C）也可以表示为序列 U+0043（LATIN CAPITAL LETTER C）U+0327（COMBINING CEDILLA）。

[1] http://www.unicode.org/Public/9.0.0/ucd/NameAliases.txt
[2] http://www.unicode.org/Public/9.0.0/ucd/NamedSequences.txt

对于每个字符，有两种正规形式：正规形式 C 和正规形式 D 。正规形式 D（NFD）也称为规范分解，并将每个字符转换为其分解形式。正规形式 C（NFC）首先应用规范分解，然后再次组合预组合字符。

除了这两种形式之外，还有两种基于兼容性等效的其他常规形式。在 Unicode 中，支持某些字符，这些字符通常与其他字符统一。例如，U+2160（ROMAN NUMERAL ONE）与 U+0049（LATIN CAPITAL LETTER I）完全相同。但是，Unicode 支持它与现有字符集（例如 gb2312 ）的兼容性。

正规形式 KD（NFKD）将应用兼容性分解，即用其等价项替换所有兼容性字符。正规形式 KC（NFKC）首先应用兼容性分解，然后是规范组合。

即使两个 unicode 字符串被规范化并且人类读者看起来相同，如果一个具有组合字符而另一个没有，则它们可能无法相等。

此外，该模块暴露了以下常量：

unicodedata.**unidata_version**
: 此模块中使用的 Unicode 数据库的版本。

unicodedata.**ucd_3_2_0**
: 这是一个与整个模块具有相同方法的对象，但对于需要此特定版本的 Unicode 数据库（如 IDNA ）的应用程序，则使用 Unicode 数据库版本 3.2 。

示例：

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional('\u0660') # 'A'rabic, 'N'umber
'AN'
```

备注

6.6 stringprep —因特网字符串预备

源代码: Lib/stringprep.py

在标识因特网上的事物（例如主机名），经常需要比较这些标识是否（相等）。这种比较的具体执行可能会取决于应用域的不同，例如是否要区分大小写等等。有时也可能需要限制允许的标识为仅由“可打印”字符组成。

RFC 3454 定义了在因特网协议中 Unicode 字符串的“预备”过程。在将字符串连线传输之前，它们会先使用预备过程进行处理，之后它们将具有特定的标准形式。该 RFC 定义了一系列表格，它们可以被组合为选项配置。每个配置必须定义所使用的表格，stringprep 过程的其他可选项也是配置的组成部分。stringprep 配置的一个例子是 nameprep，它被用于国际化域名。

模块*stringprep* 仅公开了来自 RFC 3454 的表格。由于这些如果表格如果表示为字典或列表将会非常庞大，该模块在内部使用 Unicode 字符数据库。该模块本身的源代码是使用 mkstringprep.py 工具生成的。

As a result, these tables are exposed as functions, not as data structures. There are two kinds of tables in the RFC: sets and mappings. For a set, *stringprep* provides the "characteristic function", i.e. a function that returns true if the parameter is part of the set. For mappings, it provides the mapping function: given the key, it returns the associated value. Below is a list of all functions available in the module.

stringprep.**in_table_a1**(*code*)
确定 *code* 是否属于 tableA.1 (Unicode 3.2 中的未分配码位)。

stringprep.**in_table_b1**(*code*)
确定 *code* 是否属于 tableB.1 (通常映射为空值)。

stringprep.**map_table_b2**(*code*)
返回 *code* 依据 tableB.2 (配合 NFKC 使用的大小写转换映射) 所映射的值。

stringprep.**map_table_b3**(*code*)
返回 *code* 依据 tableB.3 (不附带正规化的大小写折叠映射) 所映射的值。

stringprep.**in_table_c11**(*code*)
确定 *code* 是否属于 tableC.1.1 (ASCII 空白字符)。

stringprep.**in_table_c12**(*code*)
确定 *code* 是否属于 tableC.1.2 (非 ASCII 空白字符)。

stringprep.**in_table_c11_c12**(*code*)
确定 *code* 是否属于 tableC.1 (空白字符，C.1.1 和 C.1.2 的并集)。

stringprep.**in_table_c21**(*code*)
确定 *code* 是否属于 tableC.2.1 (ASCII 控制字符)。

stringprep.**in_table_c22**(*code*)
确定 *code* 是否属于 tableC.2.2 (非 ASCII 控制字符)。

stringprep.**in_table_c21_c22**(*code*)
确定 *code* 是否属于 tableC.2 (控制字符，C.2.1 和 C.2.2 的并集)。

stringprep.**in_table_c3**(*code*)
确定 *code* 是否属于 tableC.3 (私有使用)。

stringprep.**in_table_c4**(*code*)
确定 *code* 是否属于 tableC.4 (非字符码位)。

stringprep.**in_table_c5**(*code*)
确定 *code* 是否属于 tableC.5 (替代码)。

stringprep.**in_table_c6**(*code*)
确定 *code* 是否属于 tableC.6 (不适用于纯文本)。

stringprep.**in_table_c7**(*code*)
确定 *code* 是否属于 tableC.7 (不适用于规范表示)。

stringprep.**in_table_c8**(*code*)
确定 *code* 是否属于 tableC.8 (改变显示属性或已弃用)。

stringprep.**in_table_c9**(*code*)
确定 *code* 是否属于 tableC.9 (标记字符)。

stringprep.**in_table_d1**(*code*)
确定 *code* 是否属于 tableD.1 (带有双向属性"R"或"AL"的字符)。

stringprep.**in_table_d2**(*code*)
确定 *code* 是否属于 tableD.2 (带有双向属性"L"的字符)。

6.7 readline —GNU readline 接口

readline 模块定义了许多方便从 Python 解释器完成和读取/写入历史文件的函数。此模块可以直接使用，或通过支持在交互提示符下完成 Python 标识符的*rlcompleter* 模块使用。使用此模块进行的设置会同时影响解释器的交互提示符以及内置*input()* 函数提供的提示符。

注解： The underlying Readline library API may be implemented by the `libedit` library instead of GNU readline. On MacOS X the *readline* module detects which library is being used at run time.

`libedit` 所用的配置文件与 GNU readline 的不同。如果你要在程序中载入配置字符串你可以在 `readline.__doc__` 中检测文本“libedit”来区分 GNU readline 和 libedit。

Readline keybindings may be configured via an initialization file, typically `.inputrc` in your home directory. See Readline Init File in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.

6.7.1 初始化文件

下列函数与初始化文件和用户配置有关：

`readline.`**`parse_and_bind`**`(`*string*`)`
执行在 *string* 参数中提供的初始化行。此函数会调用底层库中的 `rl_parse_and_bind()`。

`readline.`**`read_init_file`**`(`[*filename*]`)`
执行一个 readline 初始化文件。默认文件名为最近所使用的文件名。此函数会调用底层库中的 `rl_read_init_file()`。

6.7.2 行缓冲区

下列函数会在行缓冲区上操作。

`readline.`**`get_line_buffer`**`()`
返回行缓冲区的当前内容 (底层库中的 `rl_line_buffer`)。

`readline.`**`insert_text`**`(`*string*`)`
将文本插入行缓冲区的当前游标位置。该函数会调用底层库中的 `rl_insert_text()`，但会忽略其返回值。

`readline.`**`redisplay`**`()`
改变屏幕的显示以反映行缓冲区的当前内容。该函数会调用底层库中的 `rl_redisplay()`。

6.7.3 历史文件

下列函数会在历史文件上操作：

`readline.`**`read_history_file`**`(`[*filename*]`)`
载入一个 readline 历史文件，并将其添加到历史列表。默认文件名为 `~/.history`。此函数会调用底层库中的 `read_history()`。

`readline.`**`write_history_file`**`(`[*filename*]`)`
将历史列表保存为 readline 历史文件，覆盖任何现有文件。默认文件名为 `~/.history`。此函数会调用底层库中的 `write_history()`。

readline.**append_history_file**(*nelements*[, *filename*])
将历史列表的最后 *nelements* 项添加到历史文件。默认文件名为 ~/.history。文件必须已存在。此函数会调用底层库中的 append_history() 。此函数仅当 Python 编译包带有支持此功能的库版本时才会存在。

3.5 新版功能.

readline.**get_history_length**()
readline.**set_history_length**(*length*)
设置或返回需要保存到历史文件的行数。*write_history_file()* 函数会通过调用底层库中的 history_truncate_file() 以使用该值来截取历史文件。负值意味着不限制历史文件的大小。

6.7.4 历史列表

以下函数会在全局历史列表上操作：

readline.**clear_history**()
清除当前历史。此函数会调用底层库的 clear_history() 。此 Python 函数仅当 Python 编译包带有支持此功能的库版本时才会存在。

readline.**get_current_history_length**()
返回历史列表的当前项数。(此函数不同于*get_history_length()*，后者是返回将被写入历史文件的最大行数。)

readline.**get_history_item**(*index*)
返回序号为 *index* 的历史条目的当前内容。条目序号从一开始。此函数会调用底层库中的 history_get() 。

readline.**remove_history_item**(*pos*)
从历史列表中移除指定位置上的历史条目。条目位置从零开始。此函数会调用底层库中的 remove_history() 。

readline.**replace_history_item**(*pos*, *line*)
将指定位置上的历史条目替换为 *line*。条目位置从零开始。此函数会调用底层库中的 replace_history_entry() 。

readline.**add_history**(*line*)
将 *line* 添加到历史缓冲区，相当于是最近输入的一行。此函数会调用底层库中的 add_history() 。

readline.**set_auto_history**(*enabled*)
启用或禁用当通过 readline 读取输入时自动调用 add_history() 。*enabled* 参数应为一个布尔值，当其为真值时启用自动历史，当其为假值时禁用自动历史。

3.6 新版功能.

CPython implementation detail: Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.7.5 启动钩子

readline.**set_startup_hook**([*function*])
设置或移除底层库的 rl_startup_hook 回调所发起调用的函数。如果指定了 *function*，它将被用作新的钩子函数；如果省略或为 None，任何已安装的函数将被移除。钩子函数将在 readline 打印第一个提示信息之前不带参数地被调用。

readline.**set_pre_input_hook**([*function*])
设置或移除底层库的 rl_pre_input_hook 回调所发起调用的函数。如果指定了 *function*，它将被用作新的钩子函数；如果省略或为 None，任何已安装的函数将被移除。钩子函数将在打印第一个提示信息之后、readline 开始读取输入字符之前不带参数地被调用。此函数仅当 Python 编译包带有支持此功能的库版本时才会存在。

6.7.6 Completion

以下函数与自定义单词补全函数的实现有关。这通常使用 Tab 键进行操作，能够提示并自动补全正在输入的单词。默认情况下，Readline 设置为由*rlcompleter* 来补全交互模式解释器的 Python 标识符。如果*readline* 模块要配合自定义的补全函数来使用，则需要设置不同的单词分隔符。

readline.**set_completer**([*function*])
设置或移除补全函数。如果指定了 *function*，它将被用作新的补全函数；如果省略或为 None，任何已安装的补全函数将被移除。补全函数的调用形式为 function(text, state)，其中 *state* 为 0, 1, 2, …, 直至其返回一个非字符串值。它应当返回下一个以 *text* 开头的候选补全内容。

已安装的补全函数将由传递给底层库中 rl_completion_matches() 的 *entry_func* 回调函数来发起调用。*text* 字符串来自于底层库中 rl_attempted_completion_function 回调函数的第一个形参。

readline.**get_completer**()
获取补全函数，如果没有设置补全函数则返回 None。

readline.**get_completion_type**()
获取正在尝试的补全类型。此函数会将底层库中的 rl_completion_type 变量作为一个整数返回。

readline.**get_begidx**()
readline.**get_endidx**()
获取补全域的开始和结束序号。这些序号就是传给底层库中 rl_attempted_completion_function 回调函数的 *start* 和 *end* 参数。

readline.**set_completer_delims**(*string*)
readline.**get_completer_delims**()
设置或获取补全的单词分隔符。此分隔符确定了要考虑补全的单词的开始和结束位置（补全域）。这些函数会访问底层库的 rl_completer_word_break_characters 变量。

readline.**set_completion_display_matches_hook**([*function*])
设置或移除补全显示函数。如果指定了 *function*，它将被用作新的补全显示函数；如果省略或为 None，任何已安装的补全显示函数将被移除。此函数会设置或清除底层库的 rl_completion_display_matches_hook 回调函数。补全显示函数会在每次需要显示匹配项时以 function(substitution, [matches], longest_match_length) 的形式被调用。

6.7.7 示例

以下示例演示了如何使用*readline*模块的历史读取或写入函数来自动加载和保存用户主目录下名为 .python_history 的历史文件。以下代码通常应当在交互会话期间从用户的 PYTHONSTARTUP 文件自动执行。

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

此代码实际上会在 Python 运行于 交互模式时自动运行 (参见*Readline（类库）配置*)。

以下示例实现了同样的目标，但是通过只添加新历史的方式来支持并发的交互会话。

```
import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

以下示例扩展了*code.InteractiveConsole*类以支持历史保存/恢复。

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
```

(下页继续)

(续上页)

```
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.8 rlcompleter —GNU readline 的补全函数

源代码: Lib/rlcompleter.py

---

rlcompeleter 通过补全有效的 Python 标识符和关键字定义了一个适用于*readline* 模块的补全函数。

当此模块在具有可用的*readline* 模块的 Unix 平台被导入, 一个 Completer 实例将被自动创建并且它的 complete() 方法将设置为*readline* 的补全器.

示例:

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__         readline.insert_text(      readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

rlcompleter 模块是为了使用 Python 的 交互模式而设计的。除非 Python 是通过 -S 选项运行, 这个模块总是自动地被导入且配置 (参见*Readline（类库）配置*)。

在没有*readline* 的平台, 此模块定义的 Completer 类仍然可以用于自定义行为.

6.8.1 Completer 对象

Completer 对象具有以下方法:

Completer.**complete**(*text*, *state*)

为 *text* 返回第 *state* 项补全。

如果指定的 *text* 不包含句点字符 ('.'), 它将根据当前*__main__*, *builtins* 和保留关键字（定义于*keyword* 模块）所定义的名称进行补全。

如果为带有句点的名称执行调用，它将尝试尽量求值直到最后一部分为止而不产生附带影响（函数不会被求值，但它可以生成对 __getattr__() 的调用），并通过*dir()* 函数来匹配剩余部分。在对表达式求值期间引发的任何异常都会被捕获、静默处理并返回*None*。

CHAPTER 7

二进制数据服务

本章介绍的模块提供了一些操作二进制数据的基本服务操作。有关二进制数据的其他操作，特别是与文件格式和网络协议有关的操作，将在相关章节中介绍。

下面描述的一些库文本处理服务 也可以使用 ASCII 兼容的二进制格式（例如*re*）或所有二进制数据（例如*difflib*）。

另外，请参阅 Python 的内置二进制数据类型的文档二进制序列类型—*bytes, bytearray, memoryview* 。

7.1 struct —将字节串解读为打包的二进制数据

源代码： Lib/struct.py

此模块可以执行 Python 值和以 Python *bytes* 对象表示的 C 结构之间的转换。这可以被用来处理存储在文件中或是从网络连接等其他来源获取的二进制数据。它使用格式字符串 作为 C 结构布局的精简描述以及与 Python 值的双向转换。

注解： 默认情况下，打包给定 C 结构的结果会包含填充字节以使得所涉及的 C 类型保持正确的对齐；类似地，对齐在解包时也会被纳入考虑。选择此种行为的目的是使得被打包结构的字节能与相应 C 结构在内存中的布局完全一致。要处理平台独立的数据格式或省略隐式的填充字节，请使用 `standard` 大小和对齐而不是 `native` 大小和对齐：详情参见字节顺序，大小和对齐方式。

某些*struct* 的函数（以及*Struct* 的方法）接受一个 *buffer* 参数。这将指向实现了 bufferobjects 并提供只读或是可读写缓冲的对象。用于此目的的最常见类型为*bytes* 和*bytearray*，但许多其他可被视为字节数组的类型也实现了缓冲协议，因此它们无需额外从*bytes* 对象复制即可被读取或填充。

7.1.1 函数和异常

此模块定义了下列异常和函数：

exception struct.**error**
会在多种场合下被引发的异常；其参数为一个描述错误信息的字符串。

struct.**pack**(*fmt*, *v1*, *v2*, ...)
Return a bytes object containing the values *v1*, *v2*, …packed according to the format string *fmt*. The arguments must match the values required by the format exactly.

struct.**pack_into**(*fmt*, *buffer*, *offset*, *v1*, *v2*, ...)
Pack the values *v1*, *v2*, …according to the format string *fmt* and write the packed bytes into the writable buffer *buffer* starting at position *offset*. Note that *offset* is a required argument.

struct.**unpack**(*fmt*, *buffer*)
Unpack from the buffer *buffer* (presumably packed by `pack(fmt, ...)`) according to the format string *fmt*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes must match the size required by the format, as reflected by *calcsize()*.

struct.**unpack_from**(*fmt*, *buffer*, *offset=0*)
Unpack from *buffer* starting at position *offset*, according to the format string *fmt*. The result is a tuple even if it contains exactly one item. The buffer's size in bytes, minus *offset*, must be at least the size required by the format, as reflected by *calcsize()*.

struct.**iter_unpack**(*fmt*, *buffer*)
Iteratively unpack from the buffer *buffer* according to the format string *fmt*. This function returns an iterator which will read equally-sized chunks from the buffer until all its contents have been consumed. The buffer's size in bytes must be a multiple of the size required by the format, as reflected by *calcsize()*.

每次迭代将产生一个如格式字符串所指定的元组。

3.4 新版功能.

struct.**calcsize**(*fmt*)
Return the size of the struct (and hence of the bytes object produced by `pack(fmt, ...)`) corresponding to the format string *fmt*.

7.1.2 格式字符串

格式字符串是用来在打包和解包数据时指定预期布局的机制。它们使用指定被打包/解包数据类型的格式字符 进行构建。此外，还有一些特殊字符用来控制字节顺序，大小和对齐方式。

字节顺序，大小和对齐方式

默认情况下，C 类型以机器的本机格式和字节顺序表示，并在必要时通过跳过填充字节进行正确对齐（根据 C 编译器使用的规则）。

或者，根据下表，格式字符串的第一个字符可用于指示打包数据的字节顺序，大小和对齐方式：

字符	字节顺序	大小	对齐方式
@	按原字节	按原字节	按原字节
=	按原字节	标准	无
<	小端	标准	无
>	大端	标准	无
!	网络（= 大端）	标准	无

如果第一个字符不是其中之一，则假定为 '@'。

本机字节顺序可能为大端或是小端，取决于主机系统的不同。例如，Intel x86 和 AMD64 (x86-64) 是小端的；Motorola 68000 和 PowerPC G5 是大端的；ARM 和 Intel Itanium 具有可切换的字节顺序（双端）。请使用 sys.byteorder 来检查你的系统字节顺序。

本机大小和对齐方式是使用 C 编译器的 sizeof 表达式来确定的。这总是会与本机字节顺序相绑定。

标准大小仅取决于格式字符；请参阅格式字符 部分中的表格。

请注意 '@' 和 '=' 之间的区别：两个都使用本机字节顺序，但后者的大小和对齐方式是标准化的。

格式 '!' 适合给那些宣称他们记不得网络字节顺序是大端还是小端的可怜人使用。

没有什么方式能指定非本机字节顺序（强制字节对调）；请正确选择使用 '<' 或 '>'。

注释:

(1) 填充只会在连续结构成员之间自动添加。填充不会添加到已编码结构的开头和末尾。

(2) 当使用非本机大小和对齐方式即‘<’，‘>’，‘=’, and ‘!’时不会添加任何填充。

(3) 要将结构的末尾对齐到符合特定类型的对齐要求，请以该类型代码加重复计数的零作为格式结束。参见例子。

格式字符

格式字符具有以下含义；C 和 Python 值之间的按其指定类型的转换应当是相当明显的。‘标准大小’列是指当使用标准大小时以字节表示的已打包值大小；也就是当格式字符串以 '<', '>', '!' 或 '=' 之一开头的情况。当使用本机大小时，已打包值的大小取决于具体的平台。

格式	C 类型	Python 类型	标准大小	注释
x	填充字节	无		
c	char	长度为 1 的字节串	1	
b	signed char	整数	1	(1),(3)
B	unsigned char	整数	1	(3)
?	_Bool	bool	1	(1)
h	short	整数	2	(3)
H	unsigned short	整数	2	(3)
i	int	整数	4	(3)
I	unsigned int	整数	4	(3)
l	long	整数	4	(3)
L	unsigned long	整数	4	(3)
q	long long	整数	8	(2), (3)
Q	unsigned long long	整数	8	(2), (3)
n	ssize_t	整数		(4)
N	size_t	整数		(4)
e	(7)	浮点数	2	(5)
f	float	浮点数	4	(5)
d	double	浮点数	8	(5)
s	char[]	字节串		
p	char[]	字节串		
P	void *	整数		(6)

在 3.3 版更改: 增加了对 'n' 和 'N' 格式的支持

在 3.6 版更改: 添加了对 'e' 格式的支持。

注释:

(1) '?' 转换码对应于 C99 定义的 _Bool 类型。如果此类型不可用，则使用 char 来模拟。在标准模式下，它总是以一个字节表示。

(2) The 'q' and 'Q' conversion codes are available in native mode only if the platform C compiler supports C long long, or, on Windows, __int64. They are always available in standard modes.

(3) 当尝试使用任何整数转换码打包一个非整数时，如果该非整数具有 __index__() 方法，则会在打包之前调用该方法将参数转换为一个整数。

在 3.2 版更改: 为非整数使用 __index__() 方法是 3.2 版的新增特性。

(4) 'n' 和 'N' 转换码仅对本机大小可用（选择为默认或使用 '@' 字节顺序字符）。对于标准大小，你可以使用适合你的应用的任何其他整数格式。

(5) 对于 'f', 'd' 和 'e' 转换码，打包表示形式将使用 IEEE 754 binary32, binary64 或 binary16 格式 (分别对应于 'f', 'd' 或 'e')，无论平台使用何种浮点格式。

(6) 'P' 格式字符仅对本机字节顺序可用（选择为默认或使用 '@' 字节顺序字符）。字节顺序字符 '=' 选择使用基于主机系统的小端或大端排序。struct 模块不会将其解读为本机排序，因此 'P' 格式将不可用。

(7) IEEE 754 binary16 “半精度”类型是在 IEEE 754 标准 的 2008 修订版中引入的。它包含一个符号位，5 个指数位和 11 个精度位（明确存储 10 位），可以完全精确地表示大致范围在 6.1e-05 和 6.5e+04 之间的数字。此类型并不被 C 编译器广泛支持：在一台典型的机器上，可以使用 unsigned short 进行存储，但不会被用于数学运算。请参阅维基百科页面 half-precision floating-point format 了解详情。

格式字符之前可以带有整数重复计数。例如，格式字符串 '4h' 的含义与 'hhhh' 完全相同。

格式之间的空白字符会被忽略；但是计数及其格式字符中不可有空白字符。

对于 's' 格式字符，计数会被解析为字节的长度，而不是像其他格式字符那样的重复计数；例如，'10s' 表示一个 10 字节的字节串，而 '10c' 表示 10 个字符。如果未给出计数，则默认值为 1。对于打包操作，字节串会被适当地截断或填充空字节以符合要求。对于解包操作，结果字节对象总是恰好具有指定数量的字节。作为特殊情况，'0s' 表示一个空字符串（而 '0c' 表示 0 个字符）。

当使用某一种整数格式 ('b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q') 打包值 x 时，如果 x 在该格式的有效范围之外则将引发 *struct.error*。

在 3.1 版更改: 在 3.0 中，某些包装了超范围值的整数格式会引发 *DeprecationWarning* 而不是 *struct.error*。

'p' 格式字符用于编码“Pascal 字符串”，即存储在由计数指定的 固定长度字节中的可变长度短字符串。所存储的第一个字节为字符串长度或 255 中的较小值。之后是字符串对应的字节。如果传入 *pack()* 的字符串过长（超过计数值减 1），则只有字符串前 count-1 个字节会被存储。如果字符串短于 count-1，则会填充空字节以使得恰好使用了 count 个字节。请注意对于 *unpack()*，'p' 格式字符会消耗 count 个字节，但返回的字符串永远不会包含超过 255 个字节。

对于 '?' 格式字符，返回值为 *True* 或 *False*。在打包时将会使用参数对象的逻辑值。以本机或标准 bool 类型表示的 0 或 1 将被打包，任何非零值在解包时将为 True。

例子

注解: 所有示例都假定使用一台大端机器的本机字节顺序、大小和对齐方式。

打包/解包三个整数的基础示例:

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', b'\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
>>> calcsize('hhl')
8
```

解包的字段可通过将它们赋值给变量或将结果包装为一个具名元组来命名:

```
>>> record = b'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond   ', serialnum=4658, school=264, gradelevel=8)
```

格式字符的顺序可能对大小产生影响，因为满足对齐要求所需的填充是不同的:

```
>>> pack('ci', b'*', 0x12131415)
b'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, b'*')
b'\x12\x13\x14\x15*'
>>> calcsize('ci')
8
>>> calcsize('ic')
5
```

以下格式 `'llh0l'` 指定在末尾有两个填充字节，假定 long 类型按 4 个字节的边界对齐:

```
>>> pack('llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

这仅当本机大小和对齐方式生效时才会起作用；标准大小和对齐方式并不会强制进行任何对齐。

参见:

模块*array* 被打包为二进制存储的同质数据。

模块*xdrlib* 打包和解包 XDR 数据。

7.1.3 类

struct 模块还定义了以下类型：

class struct.**Struct**(*format*)
返回一个新的 Struct 对象，它会根据格式字符串 *format* 来写入和读取二进制数据。一次性地创建 Struct 对象并调用其方法相比使用同样的格式调用*struct* 函数更为高效，因为这样格式字符串只需被编译一次。

已编译的 Struct 对象支持以下方法和属性：

pack(*v1*, *v2*, ...)
等价于*pack()* 函数，使用了已编译的格式。(len(result) 将等于*size*。)

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)
等价于*pack_into()* 函数，使用了已编译的格式。

unpack(*buffer*)
等价于*unpack()* 函数，使用了已编译的格式。缓冲区的字节大小必须等于*size*。

unpack_from(*buffer*, *offset=0*)
等价于*unpack_from()* 函数，使用了已编译的格式。缓冲区的字节大小减去 *offset* 必须至少为*size*。

iter_unpack(*buffer*)
等价于*iter_unpack()* 函数，使用了已编译的格式。缓冲区的大小必须为*size* 的整数倍。

3.4 新版功能.

format
用于构造此 Struct 对象的格式字符串。

size
计算出对应于*format* 的结构大小（亦即*pack()* 方法所产生的字节串对象的大小）。

7.2 codecs —编解码器注册和相关基类

源代码： Lib/codecs.py

这个模块定义了标准 Python 编解码器（编码器和解码器）的基类，并提供接口用来访问内部的 Python 编解码器注册表，该注册表负责管理编解码器和错误处理的查找过程。大多数标准编解码器都属于文本编码，它们可将文本编码为字节串，但也提供了一些编解码器可将文本编码为文本，以及字节串编码为字节串。自定义编解码器可以在任意类型间进行编码和解码，但某些模块特性仅适用于文本编码 或将数据编码为 字节串 的编解码器。

该模块定义了以下用于使用任何编解码器进行编码和解码的函数:

codecs.**encode**(*obj*, *encoding='utf-8'*, *errors='strict'*)
使用为 *encoding* 注册的编解码器对 *obj* 进行编码。

可以给定 *Errors* 以设置所需要的错误处理方案。默认的错误处理方案 'strict' 表示编码错误将引发*ValueError* (或更特定编解码器相关的子类，例如*UnicodeEncodeError*)。请参阅编解码器基类了解有关编解码器错误处理的更多信息。

codecs.**decode**(*obj*, *encoding='utf-8'*, *errors='strict'*)
使用为 *encoding* 注册的编解码器对 *obj* 进行解码。

可以给定 *Errors* 以设置所需要的错误处理方案。默认的错误处理方案 `'strict'` 表示编码错误将引发*ValueError*(或更特定编解码器相关的子类，例如*UnicodeDecodeError*)。请参阅*编解码器基类*了解有关编解码器错误处理的更多信息。

每种编解码器的完整细节也可以直接查找获取：

codecs.**lookup**(*encoding*)
在 Python 编解码器注册表中查找编解码器信息，并返回一个*CodecInfo* 对象，其定义见下文。

首先将会在注册表缓存中查找编码，如果未找到，则会扫描注册的搜索函数列表。如果没有找到*CodecInfo* 对象，则将引发*LookupError*。否则，*CodecInfo* 对象将被存入缓存并返回给调用者。

class codecs.**CodecInfo**(*encode*, *decode*, *streamreader=None*, *streamwriter=None*, *incrementalencoder=None*, *incrementaldecoder=None*, *name=None*)
查找编解码器注册表所得到的编解码器细节信息。构造器参数将保存为同名的属性：

name
编码名称

encode
decode
无状态的编码和解码函数。它们必须是具有与 Codec 的*encode()* 和*decode()* 方法相同接口的函数或方法 (参见*Codec 接口*)。这些函数或方法应当工作于无状态的模式。

incrementalencoder
incrementaldecoder
增量式的编码器和解码器类或工厂函数。这些函数必须分别提供由基类*IncrementalEncoder* 和*IncrementalDecoder* 所定义的接口。增量式编解码器可以保持状态。

streamwriter
streamreader
流式写入器和读取器类或工厂函数。这些函数必须分别提供由基类*StreamWriter* 和*StreamReader* 所定义的接口。流式编解码器可以保持状态。

为了简化对各种编解码器组件的访问，本模块提供了以下附加函数，它们使用*lookup()* 来执行编解码器查找：

codecs.**getencoder**(*encoding*)
查找给定编码的编解码器并返回其编码器函数。

在编码无法找到时将引发*LookupError*。

codecs.**getdecoder**(*encoding*)
查找给定编码的编解码器并返回其解码器函数。

在编码无法找到时将引发*LookupError*。

codecs.**getincrementalencoder**(*encoding*)
查找给定编码的编解码器并返回其增量式编码器类或工厂函数。

在编码无法找到或编解码器不支持增量式编码器时将引发*LookupError*。

codecs.**getincrementaldecoder**(*encoding*)
查找给定编码的编解码器并返回其增量式解码器类或工厂函数。

在编码无法找到或编解码器不支持增量式解码器时将引发*LookupError*。

codecs.**getreader**(*encoding*)
查找给定编码的编解码器并返回其*StreamReader* 类或工厂函数。

在编码无法找到时将引发*LookupError*。

codecs.**getwriter**(*encoding*)
查找给定编码的编解码器并返回其*StreamWriter* 类或工厂函数。

在编码无法找到时将引发*LookupError*。

自定义编解码器的启用是通过注册适当的编解码器搜索函数：

codecs.**register**(*search_function*)
注册一个编解码器搜索函数。搜索函数预期接收一个参数，即全部以小写字母表示的编码名称，并返回一个*CodecInfo* 对象。在搜索函数无法找到给定编码的情况下，它应当返回 None。

注解： 搜索函数的注册目前是不可逆的，这在某些情况下可能导致问题，例如单元测试或模块重载等。

虽然内置的*open()* 和相关联的*io* 模块是操作已编码文本文件的推荐方式，但本模块也提供了额外的工具函数和类，允许在操作二进制文件时使用更多各类的编解码器：

codecs.**open**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *buffering=1*)
使用给定的 *mode* 打开已编码的文件并返回一个*StreamReaderWriter* 的实例，提供透明的编码/解码。默认的文件模式为 'r'，表示以读取模式打开文件。

注解： 下层的已编码文件总是以二进制模式打开。在读取和写入时不会自动执行 '\n' 的转换。*mode* 参数可以是内置*open()* 函数所接受的任意二进制模式；'b' 会被自动添加。

---

encoding 指定文件所要使用的编码格式。允许任何编码为字节串或从字节串解码的编码格式，而文件方法所支持的数据类型则取决于所使用的编解码器。

可以指定 *errors* 来定义错误处理方案。默认值 'strict' 表示在出现编码错误时引发*ValueError*。

buffering 的含义与内置*open()* 函数中的相同。默认为行缓冲。

codecs.**EncodedFile**(*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)
返回一个*StreamRecoder* 实例，它提供了 *file* 的透明转码包装版本。当包装版本被关闭时原始文件也会被关闭。

写入已包装文件的数据会根据给定的 *data_encoding* 解码，然后以使用 *file_encoding* 的字节形式写入原始文件。从原始文件读取的字节串将根据 *file_encoding* 解码，其结果将使用 *data_encoding* 进行编码。

如果 *file_encoding* 未给定，则默认为 *data_encoding*。

可以指定 *errors* 来定义错误处理方案。默认值 'strict' 表示在出现编码错误时引发*ValueError*。

codecs.**iterencode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)
使用增量式编码器通过迭代来编码由 *iterator* 所提供的输入。此函数属于*generator*。*errors* 参数（以及任何其他关键字参数）会被传递给增量式编码器。

此函数要求编解码器接受*str* 对象形式的文本进行编码。因此它不支持字节到字节的编码器，例如 base64_codec。

codecs.**iterdecode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)
使用增量式解码器通过迭代来解码由 *iterator* 所提供的输入。此函数属于*generator*。*errors* 参数（以及任何其他关键字参数）会被传递给增量式解码器。

此函数要求编解码器接受*bytes* 对象进行解码。因此它不支持文本到文本的编码器，例如 rot_13，但是 rot_13 可以通过同样效果的*iterencode()* 来使用。

本模块还提供了以下常量，适用于读取和写入依赖于平台的文件：

codecs.**BOM**
codecs.**BOM_BE**
codecs.**BOM_LE**

codecs.**BOM_UTF8**
codecs.**BOM_UTF16**
codecs.**BOM_UTF16_BE**
codecs.**BOM_UTF16_LE**
codecs.**BOM_UTF32**
codecs.**BOM_UTF32_BE**
codecs.**BOM_UTF32_LE**

这些常量定义了多种字节序列，即一些编码格式的 Unicode 字节顺序标记（BOM）。它们在 UTF-16 和 UTF-32 数据流中被用以指明所使用的字节顺序，并在 UTF-8 中被用作 Unicode 签名。*BOM_UTF16* 是*BOM_UTF16_BE* 或*BOM_UTF16_LE*，具体取决于平台的本机字节顺序，*BOM* 是*BOM_UTF16* 的别名，*BOM_LE* 是*BOM_UTF16_LE* 的别名，*BOM_BE* 是*BOM_UTF16_BE* 的别名。其他序列则表示 UTF-8 和 UTF-32 编码格式中的 BOM。

7.2.1 编解码器基类

codecs 模块定义了一系列基类用来定义配合编解码器对象进行工作的接口，并且也可用作定制编解码器实现的基础。

每种编解码器必须定义四个接口以便用作 Python 中的编解码器：无状态编码器、无状态解码器、流读取器和流写入器。流读取器和写入器通常会重用无状态编码器/解码器来实现文件协议。编解码器作者还需要定义编解码器将如何处理编码和解码错误。

错误处理方案

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the *errors* string argument. The following string values are defined and implemented by all standard Python codecs:

值	含义
`'strict'`	Raise *UnicodeError* (or a subclass); this is the default. Implemented in *strict_errors()*.
`'ignore'`	Ignore the malformed data and continue without further notice. Implemented in *ignore_errors()*.

以下错误处理方案仅适用于文本编码:

值	含义
`'replace'`	Replace with a suitable replacement marker; Python will use the official `U+FFFD` REPLACEMENT CHARACTER for the built-in codecs on decoding, and '?' on encoding. Implemented in *replace_errors()*.
`'xmlcharrefreplace'`	Replace with the appropriate XML character reference (only for encoding). Implemented in *xmlcharrefreplace_errors()*.
`'backslashreplace'`	使用带反斜杠的转义序列进行替换。在*backslashreplace_errors()* 中实现。
`'namereplace'`	Replace with `\N{...}` escape sequences (only for encoding). Implemented in *namereplace_errors()*.
`'surrogateescape'`	On decoding, replace byte with individual surrogate code ranging from `U+DC80` to `U+DCFF`. This code will then be turned back into the same byte when the `'surrogateescape'` error handler is used when encoding the data. (See **PEP 383** for more.)

此外，以下错误处理方案被专门用于指定的编解码器:

值	编解码器	含义
`'surrogatepass'`	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Allow encoding and decoding of surrogate codes. These codecs normally treat the presence of surrogates as an error.

3.1 新版功能: 'surrogateescape' 和 'surrogatepass' 错误处理方案。

在 3.4 版更改: 'surrogatepass' 错误处理方案现在适用于 utf-16* 和 utf-32* 编解码器。

3.5 新版功能: 'namereplace' 错误处理方案。

在 3.5 版更改: 'backslashreplace' 错误处理方案现在适用于解码和转换。

允许的值集合可以通过注册新命名的错误处理方案来扩展:

codecs.**register_error**(*name*, *error_handler*)
在名称 *name* 之下注册错误处理函数 *error_handler*。当 *name* 被指定为错误形参时，*error_handler* 参数所指定的对象将在编码和解码期间发生错误的情况下被调用，

For encoding, *error_handler* will be called with a *UnicodeEncodeError* instance, which contains information about the location of the error. The error handler must either raise this or a different exception, or return a tuple with a replacement for the unencodable part of the input and a position where encoding should continue. The replacement may be either *str* or *bytes*. If the replacement is bytes, the encoder will simply copy them into the output buffer. If the replacement is a string, the encoder will encode the replacement. Encoding continues on original input at the specified position. Negative position values will be treated as being relative to the end of the input string. If the resulting position is out of bound an *IndexError* will be raised.

解码和转换的做法很相似，不同之处在于将把*UnicodeDecodeError* 或*UnicodeTranslateError* 传给处理程序，并且来自错误处理程序的替换对象将被直接放入输出。

之前注册的错误处理方案（包括标准错误处理方案）可通过名称进行查找:

codecs.**lookup_error**(*name*)
返回之前在名称 *name* 之下注册的错误处理方案。

在处理方案无法找到时将引发*LookupError*。

以下标准错误处理方案也可通过模块层级函数的方式来使用:

codecs.**strict_errors**(*exception*)
实现 'strict' 错误处理方案：每个编码或解码错误都会引发*UnicodeError*。

codecs.**replace_errors**(*exception*)
实现 'replace' 错误处理方案 (仅用于文本编码)：编码错误替换为 '?' (并由编解码器编码)，解码错误替换为 '\ufffd' (Unicode 替换字符)。

codecs.**ignore_errors**(*exception*)
实现 'ignore' 错误处理方案：忽略错误格式的数据并且不加进一步通知就继续执行。

codecs.**xmlcharrefreplace_errors**(*exception*)
实现 'xmlcharrefreplace' 错误处理方案 (仅用于文本编码 的编码过程)：不可编码的字符将以适当的 XML 字符引用进行替换。

codecs.**backslashreplace_errors**(*exception*)
实现 'backslashreplace' 错误处理方案 (仅用于文本编码)：错误格式的数据将以带反斜杠的转义序列进行替换。

codecs.**namereplace_errors**(*exception*)
实现 'namereplace' 错误处理方案 (仅用于文本编码 的编码过程)：不可编码的字符将以 \N{...} 转义序列进行替换。

3.5 新版功能.

无状态的编码和解码

基本 Codec 类定义了这些方法，同时还定义了无状态编码器和解码器的函数接口：

Codec.**encode**(*input*[, *errors*])
编码 *input* 对象并返回一个元组 (输出对象, 消耗长度)。例如，*text encoding* 会使用特定的字符集编码格式 (例如 cp1252 或 iso-8859-1) 将字符串转换为字节串对象。

errors 参数定义了要应用的错误处理方案。默认为 'strict' 处理方案。

此方法不一定会在 Codec 实例中保存状态。可使用必须保存状态的*StreamWriter* 作为编解码器以便高效地进行编码。

编码器必须能够处理零长度的输入并在此情况下返回输出对象类型的空对象。

Codec.**decode**(*input*[, *errors*])
Decodes the object *input* and returns a tuple (output object, length consumed). For instance, for a *text encoding*, decoding converts a bytes object encoded using a particular character set encoding to a string object.

对于文本编码格式和字节到字节编解码器，*input* 必须为一个字节串对象或提供了只读缓冲区接口的对象–例如，缓冲区对象和映射到内存的文件。

errors 参数定义了要应用的错误处理方案。默认为 'strict' 处理方案。

此方法不一定会在 Codec 实例中保存状态。可使用必须保存状态的*StreamReader* 作为编解码器以便高效地进行解码。

解码器必须能够处理零长度的输入并在此情况下返回输出对象类型的空对象。

增量式的编码和解码

IncrementalEncoder 和*IncrementalDecoder* 类提供了增量式编码和解码的基本接口。对输入的编码/解码不是通过对无状态编码器/解码器的一次调用，而是通过对增量式编码器/解码器的*encode()*/*decode()* 方法的多次调用。增量式编码器/解码器会在方法调用期间跟踪编码/解码过程。

调用*encode()*/*decode()* 方法后的全部输出相当于将所有通过无状态编码器/解码器进行编码/解码的单个输入连接在一起所得到的输出。

IncrementalEncoder 对象

IncrementalEncoder 类用来对一个输入进行分步编码。它定义了以下方法，每个增量式编码器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**IncrementalEncoder**(*errors='strict'*)
IncrementalEncoder 实例的构造器。

所有增量式编码器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

IncrementalEncoder 可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。可用的值请参阅*错误处理方案*。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*IncrementalEncoder* 对象的生命期内在不同的错误处理策略之间进行切换。

encode(*object*[, *final*])
编码 *object* (会将编码器的当前状态纳入考虑) 并返回已编码的结果对象。如果这是对*encode()* 的最终调用则 *final* 必须为真值（默认为假值）。

reset()
将编码器重置为初始状态。输出将被丢弃：调用 `.encode(object, final=True)`，在必要时传入一个空字节串或字符串，重置编码器并得到输出。

getstate()
Return the current state of the encoder which must be an integer. The implementation should make sure that `0` is the most common state. (States that are more complicated than integers can be converted into an integer by marshaling/pickling the state and encoding the bytes of the resulting string into an integer).

setstate(*state*)
将编码器的状态设为 *state*。*state* 必须为*getstate()* 所返回的一个编码器状态。

IncrementalDecoder 对象

IncrementalDecoder 类用来对一个输入进行分步解码。它定义了以下方法，每个增量式解码器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**IncrementalDecoder**(*errors='strict'*)
IncrementalDecoder 实例的构造器。

所有增量式解码器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

IncrementalDecoder 可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。可用的值请参阅*错误处理方案*。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*IncrementalDecoder* 对象的生命期内在不同的错误处理策略之间进行切换。

decode(*object*[, *final*])
解码 *object* (会将解码器的当前状态纳入考虑) 并返回已解码的结果对象。如果这是对*decode()* 的最终调用则 *final* 必须为真值（默认为假值）。如果 *final* 为真值则解码器必须对输入进行完全解码并且必须刷新所有缓冲区。如果这无法做到（例如由于在输入结束时字节串序列不完整）则它必须像在无状态的情况下那样初始化错误处理（这可能引发一个异常）。

reset()
将解码器重置为初始状态。

getstate()
返回解码器的当前状态。这必须为一个二元组，第一项必须是包含尚未解码的输入的缓冲区。第二项必须为一个整数，可以表示附加状态信息。（实现应当确保 `0` 是最常见的附加状态信息。）如果此附加状态信息为 `0` 则必须可以将解码器设为没有已缓冲输入并且以 `0` 作为附加状态信息，以便将先前已缓冲的输入馈送到解码器使其返回到先前的状态而不产生任何输出。（比整数更复杂的附加状态信息可以通过编组/选择状态信息并将结果字符串的字节数据编码为整数来转换为一个整数值。）

setstate(*state*)
Set the state of the encoder to *state*. *state* must be a decoder state returned by *getstate()*.

流式的编码和解码

StreamWriter 和*StreamReader* 类提供了一些泛用工作接口，可被用来非常方便地实现新的编码格式子模块。请参阅 `encodings.utf_8` 中的示例了解如何做到这一点。

StreamWriter 对象

StreamWriter 类是 `Codec` 的子类，它定义了以下方法，每个流式写入器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**StreamWriter**(*stream*, *errors='strict'*)

StreamWriter 实例的构造器。

所有流式写入器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

stream 参数必须为一个基于特定编解码器打开用于写入文本或二进制数据的文件类对象。

StreamWriter 可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。请参阅*错误处理方案* 了解下层的流式编解码器可支持的标准错误处理方案。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*StreamWriter* 对象的生命期内在不同的错误处理策略之间进行切换。

write(*object*)

将编码后的对象内容写入到流。

writelines(*list*)

将拼接后的字符串列表写入到流（可能通过重用*write()* 方法）。标准的字节到字节编解码器不支持此方法。

reset()

刷新并重置用于保持状态的编解码器缓冲区。

调用此方法应当确保在干净的状态下放入输出数据，以允许直接添加新的干净数据而无须重新扫描整个流来恢复状态。

除了上述的方法，*StreamWriter* 还必须继承来自下层流的所有其他方法和属性。

StreamReader 对象

StreamReader 类是 `Codec` 的子类，它定义了以下方法，每个流式读取器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**StreamReader**(*stream*, *errors='strict'*)

StreamReader 实例的构造器。

所有流式读取器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

stream 参数必须为一个基于特定编解码器打开用于读取文本或二进制数据的文件类对象。

StreamReader 可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。请参阅*错误处理方案* 了解下层的流式编解码器可支持的标准错误处理方案。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*StreamReader* 对象的生命期内在不同的错误处理策略之间进行切换。

errors 参数所允许的值集合可以使用*register_error()* 来扩展。

read([*size*[, *chars*[, *firstline*]]])
解码来自流的数据并返回结果对象。

chars 参数指明要返回的解码后码位或字节数量。*read()* 方法绝不会返回超出请求数量的数据，但如果可用数量不足，它可能返回少于请求数量的数据。

The *size* argument indicates the approximate maximum number of encoded bytes or code points to read for decoding. The decoder can modify this setting as appropriate. The default value -1 indicates to read and decode as much as possible. This parameter is intended to prevent having to decode huge files in one step.

firstline 旗标指明如果在后续行发生解码错误，则仅返回第一行就足够了。

此方法应当使用“贪婪”读取策略，这意味着它应当在编码格式定义和给定大小所允许的情况下尽可能多地读取数据，例如，如果在流上存在可选的编码结束或状态标记，这些内容也应当被读取。

readline([*size*[, *keepends*]])
从输入流读取一行并返回解码后的数据。

如果给定了 *size*，则将其作为 size 参数传递给流的*read()* 方法。

如果 *keepends* 为假值，则行结束符将从返回的行中去除。

readlines([*sizehint*[, *keepends*]])
从输入流读取所有行并将其作为一个行列表返回。

Line-endings are implemented using the codec’s decoder method and are included in the list entries if *keepends* is true.

如果给定了 *sizehint*，则将其作为 *size* 参数传递给流的*read()* 方法。

reset()
重置用于保持状态的编解码器缓冲区。

Note that no stream repositioning should take place. This method is primarily intended to be able to recover from decoding errors.

除了上述的方法，*StreamReader* 还必须继承来自下层流的所有其他方法和属性。

StreamReaderWriter 对象

StreamReaderWriter 是一个方便的类，允许对同时工作于读取和写入模式的流进行包装。

其设计使得开发者可以使用*lookup()* 函数所返回的工厂函数来构造实例。

class codecs.**StreamReaderWriter**(*stream*, *Reader*, *Writer*, *errors='strict'*)
创建一个*StreamReaderWriter* 实例。*stream* 必须为一个文件类对象。*Reader* 和 *Writer* 必须为分别提供了*StreamReader* 和*StreamWriter* 接口的工厂函数或类。错误处理通过与流式读取器和写入器所定义的相同方式来完成。

StreamReaderWriter 实例定义了*StreamReader* 和*StreamWriter* 类的组合接口。它们还继承了来自下层流的所有其他方法和属性。

StreamRecoder 对象

StreamRecoder 将数据从一种编码格式转换为另一种，这对于处理不同编码环境的情况有时会很有用。

其设计使得开发者可以使用 *lookup()* 函数所返回的工厂函数来构造实例。

class codecs.**StreamRecoder**(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors='strict'*)

创建一个实现了双向转换的 *StreamRecoder* 实例: *encode* 和 *decode* 工作于前端—对代码可见的数据调用 `read()` 和 `write()`，而 *Reader* 和 *Writer* 工作于后端—*stream* 中的数据。

You can use these objects to do transparent transcodings from e.g. Latin-1 to UTF-8 and back.

stream 参数必须为一个文件类对象。

encode 和 *decode* 参数必须遵循 `Codec` 接口。*Reader* 和 *Writer* 必须为分别提供了 *StreamReader* 和 *StreamWriter* 接口对象的工厂函数或类。

错误处理通过与流式读取器和写入器所定义的相同方式来完成。

StreamRecoder 实例定义了 *StreamReader* 和 *StreamWriter* 类的组合接口。它们还继承了来自下层流的所有其他方法和属性。

7.2.2 编码格式与 Unicode

Strings are stored internally as sequences of code points in range `0x0–0x10FFFF`. (See **PEP 393** for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as *encoding*, and recreating the string from the sequence of bytes is known as *decoding*. There are a variety of different text serialisation codecs, which are collectivity referred to as *text encodings*.

最简单的文本编码格式 (称为 `'latin-1'` 或 `'iso-8859-1'`) 将码位 0–255 映射为字节值 `0x0–0xff`，这意味着包含 `U+00FF` 以上码位的字符串对象无法使用此编解码器进行编码。这样做将引发 *UnicodeEncodeError*，其形式类似下面这样 (不过详细的错误信息可能会有所不同)：`UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256)`。

还有另外一组编码格式（所谓的字符映射编码）会选择全部 Unicode 码位的不同子集并设定如何将这些码位映射为字节值 `0x0–0xff`。要查看这是如何实现的，只需简单地打开相应源码例如 `encodings/cp1252.py` (这是一个主要在 Windows 上使用的编码格式)。其中会有一个包含 256 个字符的字符串常量，指明每个字符所映射的字节值。

所有这些编码格式只能对 Unicode 所定义的 1114112 个码位中的 256 个进行编码。一种能够存储每个 Unicode 码位的简单而直接的办法就是将每个码位存储为四个连续的字节。存在两种不同的可能性：以大端序存储或以小端序存储。这两种编码格式分别被称为 `UTF-32-BE` 和 `UTF-32-LE`。它们的缺点可以举例说明：如果你在一台小端序的机器上使用 `UTF-32-BE` 则你将必须在编码和解码时翻转字节。`UTF-32` 避免了这个问题：字节的排列将总是使用自然顺序。当这些字节被具有不同字节顺序的 CPU 读取时，则必须进行字节翻转。为了能够检测 `UTF-16` 或 `UTF-32` 字节序列的大小端序，可以使用所谓的 BOM (“字节顺序标记”)。这对应于 Unicode 字符 `U+FEFF`。此字符可添加到每个 `UTF-16` 或 `UTF-32` 字节序列的开头。此字符的字节翻转版本 (`0xFFFE`) 是一个不可出现于 Unicode 文本中的非法字符。因此当发现一个 `UTF-16` 或 `UTF-32` 字节序列的首个字符是 `U+FFFE` 时，就必须在解码时进行字节翻转。不幸的是字符 `U+FEFF` 还有第二个含义 `ZERO WIDTH NO-BREAK SPACE`: 即宽度为零并且不允许用来拆分单词的字符。它可以被用来为语言分析算法提供提示。在 Unicode 4.0 中用 `U+FEFF` 表示 `ZERO WIDTH NO-BREAK SPACE` 已被弃用 (改用 `U+2060` (`WORD JOINER`) 负责此任务)。然而 Unicode 软件仍然必须能够处理 `U+FEFF` 的两个含义：作为 BOM 它被用来确定已编码字节的存储布局，并在字节序列被解码为字符串后将其去除；作为 `ZERO WIDTH NO-BREAK SPACE` 它是一个普通字符，将像其他字符一样被解码。

还有另一种编码格式能够对所有的 Unicode 字符进行编码：UTF-8。UTF-8 是一种 8 位编码，这意味着在 UTF-8 中没有字节顺序问题。UTF-8 字节序列中的每个字节由两部分组成：标志位（最重要的位）和内容位。

标志位是由零至四个值为 1 的二进制位加一个值为 0 的二进制位构成的序列。Unicode 字符会按以下形式进行编码（其中 x 为内容位，当拼接为一体时将给出对应的 Unicode 字符）：

范围	编码
U-00000000 …U-0000007F	0xxxxxxx
U-00000080 …U-000007FF	110xxxxx 10xxxxxx
U-00000800 …U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
U-00010000 …U-0010FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Unicode 字符最不重要的一个位就是最右侧的二进制位 x。

由于 UTF-8 是一种 8 位编码格式，因此 BOM 是不必要的，并且已编码字符串中的任何 U+FEFF 字符（即使是作为第一个字符）都会被视为是 ZERO WIDTH NO-BREAK SPACE。

在没有外部信息的情况下，就不可能毫无疑义地确定一个字符串使用了何种编码格式。每种字符映射编码格式都可以解码任意的随机字节序列。然而对 UTF-8 来说这却是不可能的，因为 UTF-8 字节序列具有不允许任意字节序列的特别结构。为了提升 UTF-8 编码检测的可靠性，Microsoft 发明了一种 UTF-8 变体形式 (Python 2.5 称之为 "utf-8-sig") 专门用于其 Notepad 程序：在任何 Unicode 字符在被写入文件之前，会先写入一个 UTF-8 编码的 BOM (它看起来是这样一个字节序列: 0xef, 0xbb, 0xbf)。由于任何字符映射编码后的文件都不大可能以这些字节值开头（例如它们会映射为

LATIN SMALL LETTER I WITH DIAERESIS
RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
INVERTED QUESTION MARK

in iso-8859-1), this increases the probability that a utf-8-sig encoding can be correctly guessed from the byte sequence. So here the BOM is not used to be able to determine the byte order used for generating the byte sequence, but as a signature that helps in guessing the encoding. On encoding the utf-8-sig codec will write 0xef, 0xbb, 0xbf as the first three bytes to the file. On decoding utf-8-sig will skip those three bytes if they appear as the first three bytes in the file. In UTF-8, the use of the BOM is discouraged and should generally be avoided.

7.2.3 标准编码

Python 自带了许多内置的编解码器，它们的实现或者是通过 C 函数，或者是通过映射表。以下表格是按名称排序的编解码器列表，并提供了一些常见别名以及编码格式通常针对的语言。别名和语言列表都不是详尽无遗的。请注意仅有大小写区别或使用连字符替代下划线的拼写形式也都是有效的别名；因此，'utf-8' 是 'utf_8' 编解码器的有效别名。

CPython implementation detail: Some common encodings can bypass the codecs lookup machinery to improve performance. These optimization opportunities are only recognized by CPython for a limited set of (case insensitive) aliases: utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (Windows only), ascii, us-ascii, utf-16, utf16, utf-32, utf32, and the same using underscores instead of dashes. Using alternative aliases for these encodings may result in slower execution.

在 3.6 版更改: 可识别针对 us-ascii 的优化机会。

许多字符集都支持相同的语言。它们在个别字符（例如是否支持 EURO SIGN 等）以及给字符所分配的码位方面存在差异。特别是对于欧洲语言来说，通常存在以下几种变体：

- 某个 ISO 8859 编码集
- 某个 Microsoft Windows 编码页，通常是派生自某个 8859 编码集，但会用附加的图形字符来替换控制字符。
- 某个 IBM EBCDIC 编码页
- 某个 IBM PC 编码页，通常会兼容 ASCII

编码	别名	语言
ascii	646, us-ascii	英语
big5	big5-tw, csbig5	繁体中文
big5hkscs	big5-hkscs, hkscs	繁体中文
cp037	IBM037, IBM039	英语
cp273	273, IBM273, csIBM273	德语 3.4 新版功能.
cp424	EBCDIC-CP-HE, IBM424	希伯来语
cp437	437, IBM437	英语
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	西欧
cp720		阿拉伯语
cp737		希腊语
cp775	IBM775	波罗的海语言
cp850	850, IBM850	西欧
cp852	852, IBM852	中欧和东欧
cp855	855, IBM855	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
cp856		希伯来语
cp857	857, IBM857	土耳其语
cp858	858, IBM858	西欧
cp860	860, IBM860	葡萄牙语
cp861	861, CP-IS, IBM861	冰岛语
cp862	862, IBM862	希伯来语
cp863	863, IBM863	加拿大语
cp864	IBM864	阿拉伯语
cp865	865, IBM865	丹麦语/挪威语
cp866	866, IBM866	俄语
cp869	869, CP-GR, IBM869	希腊语
cp874		泰语
cp875		希腊语
cp932	932, ms932, mskanji, ms-kanji	日语
cp949	949, ms949, uhc	韩语
cp950	950, ms950	繁体中文
cp1006		乌尔都语
cp1026	ibm1026	土耳其语
cp1125	1125, ibm1125, cp866u, ruscii	乌克兰语 3.4 新版功能.
cp1140	ibm1140	西欧
cp1250	windows-1250	中欧和东欧
cp1251	windows-1251	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
cp1252	windows-1252	西欧
cp1253	windows-1253	希腊语
cp1254	windows-1254	土耳其语
cp1255	windows-1255	希伯来语
cp1256	windows-1256	阿拉伯语
cp1257	windows-1257	波罗的海语言
cp1258	windows-1258	越南语

下页继续

表 1 – 续上页

编码	别名	语言
cp65001		仅 Windows: Windows UTF-8 (CP_UTF8) 3.3 新版功能.
euc_jp	eucjp, ujis, u-jis	日语
euc_jis_2004	jisx0213, eucjis2004	日语
euc_jisx0213	eucjisx0213	日语
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	韩语
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	简体中文
gbk	936, cp936, ms936	统一汉语
gb18030	gb18030-2000	统一汉语
hz	hzgb, hz-gb, hz-gb-2312	简体中文
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	日语
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	日语
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	日语，韩语，简体中文，西欧，希腊语
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	日语
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	日语
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	日语
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	韩语
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	West Europe
iso8859_2	iso-8859-2, latin2, L2	中欧和东欧
iso8859_3	iso-8859-3, latin3, L3	世界语，马耳他语
iso8859_4	iso-8859-4, latin4, L4	波罗的海语言
iso8859_5	iso-8859-5, cyrillic	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
iso8859_6	iso-8859-6, arabic	阿拉伯语
iso8859_7	iso-8859-7, greek, greek8	希腊语
iso8859_8	iso-8859-8, hebrew	希伯来语
iso8859_9	iso-8859-9, latin5, L5	土耳其语
iso8859_10	iso-8859-10, latin6, L6	北欧语言
iso8859_11	iso-8859-11, thai	泰语
iso8859_13	iso-8859-13, latin7, L7	波罗的海语言
iso8859_14	iso-8859-14, latin8, L8	凯尔特语
iso8859_15	iso-8859-15, latin9, L9	西欧
iso8859_16	iso-8859-16, latin10, L10	东南欧
johab	cp1361, ms1361	韩语
koi8_r		俄语
koi8_t		塔吉克 3.5 新版功能.
koi8_u		乌克兰语
kz1048	kz_1048, strk1048_2002, rk1048	哈萨克语 3.5 新版功能.

下页继续

表 1 – 续上页

编码	别名	语言
mac_cyrillic	maccyrillic	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
mac_greek	macgreek	希腊语
mac_iceland	maciceland	冰岛语
mac_latin2	maclatin2, maccentraleurope	中欧和东欧
mac_roman	macroman, macintosh	西欧
mac_turkish	macturkish	土耳其语
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	哈萨克语
shift_jis	csshiftjis, shiftjis, sjis, s_jis	日语
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	日语
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	日语
utf_32	U32, utf32	所有语言
utf_32_be	UTF-32BE	所有语言
utf_32_le	UTF-32LE	所有语言
utf_16	U16, utf16	所有语言
utf_16_be	UTF-16BE	所有语言
utf_16_le	UTF-16LE	所有语言
utf_7	U7, unicode-1-1-utf-7	所有语言
utf_8	U8, UTF, utf8	所有语言
utf_8_sig		所有语言

在 3.4 版更改: utf-16* and utf-32* 编码器将不再允许编码代理码位 (U+D800–U+DFFF)。utf-32* 解码器将不再解码与代理码位相对应的字节序列。

7.2.4 Python 专属的编码格式

A number of predefined codecs are specific to Python, so their codec names have no meaning outside Python. These are listed in the tables below based on the expected input and output types (note that while text encodings are the most common use case for codecs, the underlying codec infrastructure supports arbitrary data transforms rather than just text encodings). For asymmetric codecs, the stated purpose describes the encoding direction.

文字编码

以下编解码器提供了`str`到`bytes`的编码和*bytes-like object*到`str`的解码，类似于 Unicode 文本编码。

编码	别名	Purpose
idna		Implements **RFC 3490**, see also *encodings.idna*. Only `errors='strict'` is supported.
mbcs	ansi, dbcs	Windows only: Encode operand according to the ANSI codepage (CP_ACP)
oem		Windows only: Encode operand according to the OEM codepage (CP_OEMCP) 3.6 新版功能.
palmos		Encoding of PalmOS 3.5
punycode		Implements **RFC 3492**. Stateful codecs are not supported.
raw_unicode_escape		Latin-1 编码格式附带对其他码位以 `\uXXXX` 和 `\UXXXXXXXX` 进行编码。现有反斜杠不会以任何方式转义。它被用于 Python 的 pickle 协议。
undefined		所有转换都将引发异常，甚至对空字符串也不例外。错误处理方案会被忽略。
unicode_escape		Encoding suitable as the contents of a Unicode literal in ASCII-encoded Python source code, except that quotes are not escaped. Decodes from Latin-1 source code. Beware that Python source code actually uses UTF-8 by default.
unicode_internal		返回操作数的内部表示。不支持有状态的编解码器。 3.3 版后已移除: 此表示已被 **PEP 393** 所废弃。

二进制转换

The following codecs provide binary transforms: *bytes-like object* to `bytes` mappings. They are not supported by `bytes.decode()` (which only produces `str` output).

编码	别名	Purpose	编码器/解码器
base64_codec[1]	base64, base_64	Convert operand to multiline MIME base64 (the result always includes a trailing `'\n'`) 在 3.4 版更改: 接受任意*bytes-like object* 作为输入用于编码和解码	*base64.encodebytes()* / *base64.decodebytes()*
bz2_codec	bz2	Compress the operand using bz2	*bz2.compress()* / *bz2.decompress()*
hex_codec	hex	Convert operand to hexadecimal representation, with two digits per byte	*binascii.b2a_hex()* / *binascii.a2b_hex()*
quopri_codec	quopri, quotedprintable, quoted_printable	Convert operand to MIME quoted printable	*quopri.encode()* 且 `quotetabs=True` / *quopri.decode()*
uu_codec	uu	Convert the operand using uuencode	*uu.encode()* / *uu.decode()*
zlib_codec	zip, zlib	Compress the operand using gzip	*zlib.compress()* / *zlib.decompress()*

3.2 新版功能: 恢复二进制转换。

在 3.4 版更改: 恢复二进制转换的别名。

文字转换

The following codec provides a text transform: a *str* to *str* mapping. It is not supported by *str.encode()* (which only produces *bytes* output).

编码	别名	Purpose
rot_13	rot13	Returns the Caesar-cypher encryption of the operand

3.2 新版功能: 恢复 `rot_13` 文本转换。

在 3.4 版更改: 恢复 `rot13` 别名。

7.2.5 encodings.idna —应用程序中的国际化域名

此模块实现了 **RFC 3490** (应用程序中的国际化域名) 和 **RFC 3492** (Nameprep: 用于国际化域名 (IDN) 的 Stringprep 配置文件)。它是在 `punycode` 编码格式和*stringprep* 的基础上构建的。

这些 RFC 共同定义了一个在域名中支持非 ASCII 字符的协议。一个包含非 ASCII 字符的域名 (例如 `www.Alliancefrançaise.nu`) 会被转换为兼容 ASCII 的编码格式 (简称 ACE，例如 `www.xn--alliancefranaise-npb.nu`)。随后此域名的 ACE 形式可以用于所有由于特定协议而不允许使用任意字符的场合，例如 DNS 查询，HTTP *Host* 字段等等。此转换是在应用中进行的；如有可能将对用户可见：应用应当透明地将 Unicode 域名标签转换为线上的 IDNA，并在 ACE 标签被呈现给用户之前将其转换回 Unicode。

Python supports this conversion in several ways: the `idna` codec performs conversion between Unicode and ACE, separating an input string into labels based on the separator characters defined in **section 3.1 of RFC 3490** and converting

[1] 除了字节类对象，`'base64_codec'` 也接受仅包含 ASCII 的*str* 实例用于解码

each label to ACE as required, and conversely separating an input byte string into labels based on the . separator and converting any ACE labels found into unicode. Furthermore, the *socket* module transparently converts Unicode host names to ACE, so that applications need not be concerned about converting host names themselves when they pass them to the socket module. On top of that, modules that have host names as function parameters, such as *http.client* and *ftplib*, accept Unicode host names (*http.client* then also transparently sends an IDNA hostname in the *Host* field if it sends that field at all).

When receiving host names from the wire (such as in reverse name lookup), no automatic conversion to Unicode is performed: Applications wishing to present such host names to the user should decode them to Unicode.

encodings.idna 模块还实现了 nameprep 过程，该过程会对主机名执行特定的规范化操作，以实现国际域名的大小写不敏感特性与合并相似的字符。如果有需要可以直接使用 nameprep 函数。

encodings.idna.**nameprep**(*label*)
返回 *label* 经过名称处理操作的版本。该实现目前基于查询字符串，因此 AllowUnassigned 为真值。

encodings.idna.**ToASCII**(*label*)
将标签转换为 ASCII，规则定义见 **RFC 3490**。UseSTD3ASCIIRules 预设为假值。

encodings.idna.**ToUnicode**(*label*)
将标签转换为 Unicode，规则定义见 **RFC 3490**。

7.2.6 encodings.mbcs —Windows ANSI 代码页

Encode operand according to the ANSI codepage (CP_ACP).

Availability: Windows only.

在 3.3 版更改: 支持任何错误处理

在 3.2 版更改: 在 3.2 版之前，*errors* 参数会被忽略；总是会使用 'replace' 进行编码，并使用 'ignore' 进行解码。

7.2.7 encodings.utf_8_sig —带 BOM 签名的 UTF-8 编解码器

This module implements a variant of the UTF-8 codec: On encoding a UTF-8 encoded BOM will be prepended to the UTF-8 encoded bytes. For the stateful encoder this is only done once (on the first write to the byte stream). For decoding an optional UTF-8 encoded BOM at the start of the data will be skipped.

CHAPTER 8

数据类型

The modules described in this chapter provide a variety of specialized data types such as dates and times, fixed-type arrays, heap queues, synchronized queues, and sets.

Python also provides some built-in data types, in particular, *dict*, *list*, *set* and *frozenset*, and *tuple*. The *str* class is used to hold Unicode strings, and the *bytes* class is used to hold binary data.

本章包含以下模块的文档:

8.1 datetime —基本的日期和时间类型

源代码: Lib/datetime.py

datetime 模块提供了可以通过多种方式操作日期和时间的类。在支持日期时间数学运算的同时，实现的关注点更着重于如何能够更有效地解析其属性用于格式化输出和数据操作。相关功能可以参阅*time* 和*calendar* 模块。

有两种日期和时间的对象：“简单型“和”感知型“。

感知型对象有着用足以支持一些应用层面算法和国家层面时间调整的信息，例如时区和夏令时，来让自己和其他的感知型对象区别开来。感知型对象是用来表达不对解释器开放的特定时间信息[1]。

简单型对象没包含足够多的信息来明确定位与之相关的 date/time 对象。简单型对象所代表的是世界标准时间 (UTC)、当地时间或者是其它时区的时间完全取决于程序，就像一个数字是代表的是米、英里或者质量完全取决于程序一样。简单型对象以忽略了一些现实情况的为代价使得它容易理解和使用。

对于需要感知型对象的应用，*datetime* 对象和*time* 对象有一个可选的时区信息属性 `tzinfo` ，这个属性可以设置给*tzinfo* 类的子类实例。这些*tzinfo* 对象捕获关于相对于世界标准时间（UTC）偏移、时区名字和夏令时是否有效等信息。需要注意的是，只有一个具体的*tzinfo* 类，即由*datetime* 模块提供的*timezone* 类。*timezone* 类可以代表相对于世界标准时间（UTC）固定偏移的简单时区，比如世界标准时间（UTC）自己或者北美东部时间或者东部夏令时。支持时区的详细程度取决于应用。世界各地的时间调

[1] 就是说如果我们忽略相对论效应的话。

整规则相比理性更加政治性，经常会变更。也没有一个基于世界标准时间（UTC）的标准套件适合用于所有应用。

The *datetime* module exports the following constants:

datetime.**MINYEAR**
date 或者*datetime* 对象允许的最小年份。常量*MINYEAR* 是 1 。

datetime.**MAXYEAR**
date 或*datetime* 对象允许最大的年份。常量*MAXYEAR* 是 9999 。

参见:

模块*calendar* 日历相关函数

模块*time* 时间的访问和转换

8.1.1 有效的类型

class datetime.**date**
一个理想化的简单型日期，它假设当今的公历在过去和未来永远有效。属性: *year*, *month*, and *day*。

class datetime.**time**
一个理想化的时间，它独立于任何特定的日期，假设每天一共有 24*60*60 秒（这里没有”闰秒”的概念）。属性：*hour*, *minute*, *second*, *microsecond*, 和*tzinfo*。

class datetime.**datetime**
日期和时间的结合。属性：*year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.

class datetime.**timedelta**
表示两个*date* 对象或者*time* 对象, 或者*datetime* 对象之间的时间间隔，精确到微秒。

class datetime.**tzinfo**
一个描述时区信息的抽象基类。用于给*datetime* 类和*time* 类提供自定义的时间调整概念（例如，负责时区或者夏令时）。

class datetime.**timezone**
一个实现了*tzinfo* 抽象基类的子类，用于表示相对于世界标准时间（UTC）的偏移量。

3.2 新版功能.

这些类型的对象都是不可变的。

date 类型的对象都是简单型的。

An object of type *time* or *datetime* may be naive or aware. A *datetime* object *d* is aware if `d.tzinfo` is not `None` and `d.tzinfo.utcoffset(d)` does not return `None`. If `d.tzinfo` is `None`, or if `d.tzinfo` is not `None` but `d.tzinfo.utcoffset(d)` returns `None`, *d* is naive. A *time* object *t* is aware if `t.tzinfo` is not `None` and `t.tzinfo.utcoffset(None)` does not return `None`. Otherwise, *t* is naive.

简单型和感知型之间的差别不适用于*timedelta* 对象。

子类关系

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

8.1.2 timedelta 类对象

timedelta 对象表示两个 date 或者 time 的时间间隔。

class datetime.**timedelta**(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

所有的参数都是可选的并且默认为 0。这些参数可以是整数或者浮点数，也可以是正数或者负数。

只有 *days*,*seconds* 和 *microseconds* 会存储在内部，即 python 内部以 *days*,*seconds* 和 *microseconds* 三个单位作为存储的基本单位。参数单位转换规则如下：

- 1 毫秒会转换成 1000 微秒。
- 1 分钟会转换成 60 秒。
- 1 小时会转换成 3600 秒。
- 1 星期会转换成 7 天。

days, seconds, microseconds 本身也是标准化的，以保证表达方式的唯一性，例：

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (一天的秒数)
- `-999999999 <= days <= 999999999`

在有任何浮点型参数或者微秒为小数的情况下，所有小数均会按照前面的换算规则叠加到下一级，并使用 round-half-to-even 的方法对微秒进行取舍。没有浮点型参数情况下，转换的过程就是精确的（没有信息丢失）。

如果标准化后的 days 数值超过了指定范围，将会抛出*OverflowError* 异常。

需要注意的是，负数被标准化后的结果会让你大吃一惊。例如，

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

类属性：

timedelta.**min**
The most negative *timedelta* object, `timedelta(-999999999)`.

timedelta.**max**
The most positive *timedelta* object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

timedelta.**resolution**
两个不相等的*timedelta* 类对象最小的间隔为 `timedelta(microseconds=1)`。

需要注意的是，因为标准化的缘故，`timedelta.max` > `-timedelta.min`，`-timedelta.max` 不可以表示一个*timedelta* 类对象。

实例属性（只读）：

属性	值
days	-999999999 至 999999999，含 999999999
seconds	0 至 86399，包含 86399
microseconds	0 至 999999，包含 999999

支持的运算：

运算	结果:
`t1 = t2 + t3`	*t2* 和 *t3* 的和。运算后 *t1-t2 == t3* and *t1-t3 == t2* 必为真值。(1)
`t1 = t2 - t3`	*t2* 减 *t3* 的差。运算后 *t1 == t2 - t3* and *t2 == t1 + t3* 必为真值。(1)(6)
`t1 = t2 * i or t1 = i * t2`	乘以一个整数。运算后假如 `i != 0` 则 *t1* // i == *t2* 必为真值。
	In general, *t1* * i == *t1* * (i-1) + *t1* is true. (1)
`t1 = t2 * f or t1 = f * t2`	乘以一个浮点数，结果会被舍入到 timedelta 最接近的整数倍。精度使用四舍五偶入奇不入规则。
`f = t2 / t3`	Division (3) of *t2* by *t3*. Returns a *float* object.
`t1 = t2 / f or t1 = t2 / i`	除以一个浮点数或整数。结果会被舍入到 timedelta 最接近的整数倍。精度使用四舍五偶入奇不入规则。
`t1 = t2 // i` or `t1 = t2 // t3`	取整除，余数部分 (如果有的话) 将被丢弃。在第二种情况下，返回一个整数。(3)
`t1 = t2 % t3`	余数为一个*timedelta* 对象。(3)
`q, r = divmod(t1, t2)`	通过: `q = t1 // t2` (3) and `r = t1 % t2` 计算出商和余数。q 是一个整数，r 是一个*timedelta* 对象。
`+t1`	返回一个相同数值的*timedelta* 对象。
`-t1`	等价于*timedelta*(*-t1.days*, *-t1.seconds*, *-t1.microseconds*), 和 *t1** -1. (1)(4)
`abs(t)`	当 `t.days >= 0``时等于 +\ *t*, 当 ``t.days < 0` 时 *-t* 。(2)
`str(t)`	返回一个形如 `[D day[s], ][H]H:MM:SS[.UUUUUU]` 的字符串，当 `t` 为负数的时候，D 也为负数。(5)
`repr(t)`	Returns a string in the form `datetime.timedelta(D[, S[, U]])`, where D is negative for negative `t`. (5)

注释:

(1) 精确但可能会溢出。

(2) 精确且不会溢出。

(3) 除以 0 将会抛出异常*ZeroDivisionError* 。

(4) *-timedelta.max* 不是一个*timedelta* 类对象。

(5) String representations of *timedelta* objects are normalized similarly to their internal representation. This leads to somewhat unusual results for negative timedeltas. For example:

```
>>> timedelta(hours=-5)
datetime.timedelta(-1, 68400)
>>> print(_)
-1 day, 19:00:00
```

(6) 表达式 `t2 - t3` 通常与 `t2 + (-t3)` 是等价的，除非 t3 等于 `timedelta.max`; 在这种情况下前者会返回结果，而后者则会溢出。

除了上面列举的操作以外*timedelta* 对象还支持与*date* 和*datetime* 对象进行特定的相加和相减运算（见下文）。

在 3.2 版更改: 现在已支持*timedelta* 对象与另一个*timedelta* 对象相整除或相除，包括求余运算和*divmod()* 函数。现在也支持*timedelta* 对象被*float* 对象除或乘。

timedelta 对象与*timedelta* 对象比较的支持是通过将表示较小时间差的 timedelta 对象视为较小值。为了防止将混合类型比较回退为基于对象地址的默认比较，当*timedelta* 对象与不同类型的对象比较时，将会引发*TypeError*，除非比较运算符是 == 或 !=。在后一种情况下将分别返回*False* 或*True*。

timedelta 对象是*hashable* 类型的 (可以作为字典关键字), 支持高效获取, 在布尔上下文中，*timedelta* 对象大多数情况下都被视为真，仅在不等于 `timedelta(0)` 时。

实例方法：

timedelta.**total_seconds**()
Return the total number of seconds contained in the duration. Equivalent to `td / timedelta(seconds=1)`.

需要注意的是，时间间隔较大时，这个方法的结果中的微秒将会失真（大多数平台上大于 270 年视为一个较大的时间间隔）。

3.2 新版功能.

用法示例：

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)  # adds up to 365 days
>>> year.total_seconds()
31536000.0
>>> year == another_year
True
>>> ten_years = 10 * year
>>> ten_years, ten_years.days // 365
(datetime.timedelta(3650), 10)
>>> nine_years = ten_years - year
>>> nine_years, nine_years.days // 365
(datetime.timedelta(3285), 9)
>>> three_years = nine_years // 3;
>>> three_years, three_years.days // 365
(datetime.timedelta(1095), 3)
>>> abs(three_years - ten_years) == 2 * three_years + year
True
```

8.1.3 date 对象

date 对象代表一个理想化历法中的日期（年、月和日），即当今的格列高利历向前后两个方向无限延伸。公元 1 年 1 月 1 日是第 1 日，公元 1 年 1 月 2 日是第 2 日，依此类推。这与 Dershowitz 与 Reingold 所著 Calendrical Calculations 中“预期格列高利”历法的定义一致，它是适用于该书中所有运算的基础历法。请参阅该书了解在预期格列高利历序列与许多其他历法系统之间进行转换的算法。

class datetime.**date**(*year*, *month*, *day*)
All arguments are required. Arguments may be integers, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= 日期 <= 给定年月对应的天数`

如果参数不在这些范围内，则抛出*ValueError*异常。

其它构造器，所有的类方法：

classmethod date.**today**()
返回当地的当前日期。与“date.fromtimestamp(time.time())“等价。

classmethod date.**fromtimestamp**(*timestamp*)
返回对应于 POSIX 时间戳例如*time.time()* 返回值的本地日期。这可能引发*OverflowError*，如果时间戳数值超出所在平台 C `localtime()` 函数的支持范围的话，并会在 `localtime()` 出错时引发*OSError*。通常该数值会被限制在 1970 年至 2038 年之间。请注意在时间戳概念包含闰秒的非 POSIX 系统上，闰秒会被*fromtimestamp()* 所忽略。

在 3.3 版更改：引发*OverflowError* 而不是*ValueError*，如果时间戳数值超出所在平台 C `localtime()` 函数的支持范围的话，并会在 `localtime()` 出错时引发*OSError* 而不是*ValueError*。

classmethod `date.`**fromordinal**(*ordinal*)
返回对应于预期格列高利历序号的日期，其中公元 1 年 1 月 1 日的序号为 1。除非 `1 <= 序号 <= date.max.toordinal()` 否则会引发*ValueError*。对于任意日期 *d*，`date.fromordinal(d.toordinal()) == d`。

类属性：

`date.`**min**
最小的日期 `date(MINYEAR, 1, 1)`。

`date.`**max**
最大的日期，`date(MAXYEAR, 12, 31)`。

`date.`**resolution**
两个日期对象的最小间隔，`timedelta(days=1)`。

实例属性（只读）：

`date.`**year**
在*MINYEAR* 和*MAXYEAR* 之间，包含边界。

`date.`**month**
1 至 12（含）

`date.`**day**
返回 1 到指定年月的天数间的数字。

支持的运算：

运算	结果：
`date2 = date1 + timedelta`	*date2* 等于从 *date1* 减去 `timedelta.days` 天。(1)
`date2 = date1 - timedelta`	计算 *date2* 的值使得 `date2 + timedelta == date1`。(2)
`timedelta = date1 - date2`	(3)
`date1 < date2`	如果 *date1* 的时间在 *date2* 之前则认为 *date1* 小于 *date2* 。(4)

注释：

(1) 如果 `timedelta.days > 0` 则 *date2* 在时间线上前进，如果 `timedelta.days < 0` 则后退。操作完成后 `date2 - date1 == timedelta.days`。`timedelta.seconds` 和 `timedelta.microseconds` 会被忽略。如果 `date2.year` 将小于*MINYEAR* 或大于*MAXYEAR* 则会引发*OverflowError*。.

(2) `timedelta.seconds` 和 `timedelta.microseconds` 会被忽略。

(3) 精确且不会溢出。操作完成后 timedelta.seconds 和 timedelta.microseconds 均为 0，并且 date2 + timedelta == date1。

(4) 换句话说，当且仅当 `date1.toordinal() < date2.toordinal()` 时 `date1 < date2`。日期比较会引发*TypeError*，如果比较目标不为*date* 对象的话。不过也可能会返回 `NotImplemented`，如果比较目标具有 `timetuple()` 属性的话。这个钩子给予其他日期对象类型实现混合类型比较的机会。否则，当*date* 对象与不同类型的对象比较时将会引发*TypeError*，除非 == 或 != 比较。后两种情况将分别返回*False* 或*True*。

日期可以作为字典的关键字。在布尔上下文中，所有的*date* 对象都视为真。

实例方法：

date.**replace**(*year=self.year*, *month=self.month*, *day=self.day*)
返回一个具有同样值的日期，除非通过任何关键字参数给出了某些形参的新值。例如，如果 `d == date(2002, 12, 31)`，则 `d.replace(day=26) == date(2002, 12, 26)`。

date.**timetuple**()
返回一个*time.struct_time*，即与*time.localtime()* 的返回类型相同。hours, minutes 和 seconds 值为 0, 且 DST 标志值为 -1。`d.timetuple()` 等价于 `time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))`，其中 `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` 是日期在当前年份中的序号，起始序号 `1` 表示 1 月 1 日。

date.**toordinal**()
返回日期的预期格列高利历序号，其中公元 1 年 1 月 1 日的序号为 1。对于任意*date* 对象 *d*，`date.fromordinal(d.toordinal()) == d`。

date.**weekday**()
返回一个整数代表星期几，星期一为 0，星期天为 6。例如，`date(2002, 12, 4).weekday() == 2`，表示的是星期三。参阅*isoweekday()*。

date.**isoweekday**()
返回一个整数代表星期几，星期一为 1，星期天为 7。例如：`date(2002, 12, 4).isoweekday() == 3`, 表示星期三。参见*weekday()*, *isocalendar()*。

date.**isocalendar**()
返回一个三元元组，(ISO year, ISO week number, ISO weekday) 。

ISO 日历是一个被广泛使用的公历。可以从 https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm 上查看更完整的说明。

ISO 年由 52 或 53 个完整星期构成，每个星期开始于星期一结束于星期日。一个 ISO 年的第一个星期就是（格列高利）历法的的一年中第一个包含星期四的星期。这被称为 1 号星期，其中星期四所在的 ISO 年与其所在的格列高利年相同。

例如，2004 年的第一天是一个星期四，因此 ISO 2004 年的第一个星期开始于 2003 年 12 月 29 日星期一，结束于 2004 年 1 月 4 日星期日，因此 `date(2003, 12, 29).isocalendar() == (2004, 1, 1)` and `date(2004, 1, 4).isocalendar() == (2004, 1, 7)`。

date.**isoformat**()
返回一个 ISO 8601 格式的字符串，‘YYYY-MM-DD’。例如 `date(2002, 12, 4).isoformat() == '2002-12-04'`。

date.**__str__**()
对于日期对象 *d*, `str(d)` 等价于 `d.isoformat()` 。

date.**ctime**()
返回一个代表日期的字符串，例如 `date(2002, 12, 4).ctime() == 'Wed Dec 4 00:00:00 2002'`。在原生 C `ctime()` 函数 (*time.ctime()* 会发起调用该函数，但*date.ctime()* 则不会) 遵循 C 标准的平台上，`d.ctime()` 等价于 `time.ctime(time.mktime(d.timetuple()))`。

date.**strftime**(*format*)
返回一个由显式格式字符串所指明的代表日期的字符串。表示时、分或秒的格式代码值将为 0。要获取格式指令的完整列表请参阅*strftime() 和 strptime() 的行为*。

date.**__format__**(*format*)
与*date.strftime()* 相同。此方法使得为*date* 对象指定以 格式化字符串字面值表示的格式化字符串以及使用*str.format()* 进行格式化成为可能。要获取格式指令的完整列表，请参阅*strftime() 和 strptime() 的行为*。

计算距离特定事件天数的例子:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

使用*date*的例子:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'
```

8.1.4 datetime 对象

datetime 对象是一个包含了来自*date* 对象和*time* 对象所有信息的单一对象。与*date* 对象一样，*datetime* 假定当今的格列高利历向前后两个方向无限延伸；与 time 对象一样，*datetime* 假定每一天恰好有 3600*24 秒。

构造器：

class datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, ***, *fold=0*)

The year, month and day arguments are required. *tzinfo* may be `None`, or an instance of a *tzinfo* subclass. The remaining arguments may be integers, in the following ranges:

- `MINYEAR <= year <= MAXYEAR`,
- `1 <= month <= 12`,
- `1 <= day <=` 指定年月的天数,
- `0 <= hour < 24`,
- `0 <= minute < 60`,
- `0 <= second < 60`,
- `0 <= microsecond < 1000000`,
- `fold in [0, 1]`.

如果参数不在这些范围内，则抛出*ValueError* 异常。

3.6 新版功能: 增加了 `fold` 参数。

其它构造器，所有的类方法：

classmethod datetime.**today**()

返回当前的本地 datetime，*tzinfo* 值为 `None`。这等价于 `datetime.fromtimestamp(time.time())`。另请参阅*now()*, *fromtimestamp()*。

classmethod datetime.**now**(*tz=None*)

返回当前的本地 date 和 time。如果可选参数 *tz* 为 `None` 或未指定，这就类似于*today()*，但该方法会在可能的情况下提供比通过*time.time()* 时间戳所获时间值更高的精度（例如，在提供了 C `gettimeofday()` 函数的平台上就可能做到）。

如果 *tz* 不为 `None`，它必须是*tzinfo* 的子类的一个实例，并且当前日期和时间将转换为 *tz* 时区的日期和时间。在这种情况下结果等价于 `tz.fromutc(datetime.utcnow().replace(tzinfo=tz))`。另请参阅*today()*, *utcnow()*。

classmethod datetime.**utcnow**()

返回当前 UTC 日期和时间，其中*tzinfo* 为 `None`。这类似于*now()*，但返回的当前 UTC 日期和时间是一个本机*datetime* 对象。自动感知的当前 UTC 日期时间可通过调用 `datetime.now(timezone.utc)` 来获得。另请参阅*now()*。

classmethod datetime.**fromtimestamp**(*timestamp*, *tz=None*)

返回对应于 POSIX 时间戳例如*time.time()* 的返回值的本地日期和时间。如果可选参数 *tz* 为 `None` 或未指定，时间戳会被转换为所在平台的本地日期和时间，返回的*datetime* 对象将为天真型。

如果 *tz* 不为 `None`，它必须是*tzinfo* 子类的一个实例，并且时间戳将被转换到 *tz* 指定的时区。在这种情况下结果等价于 `tz.fromutc(datetime.utcfromtimestamp(timestamp).replace(tzinfo=tz))`。

fromtimestamp() 可能引发*OverflowError*，如果时间戳数值超出所在平台 C `localtime()` 或 `gmtime()` 函数的支持范围的话，并会在 `localtime()` 或 `gmtime()` 出错时引发*OSError*。通常该数值会被限制在 1970 年至 2038 年之间。请注意在时间戳概念包含闰秒的非 POSIX 系统上，闰秒会

被*fromtimestamp()* 所忽略，结果可能导致两个相差一秒的时间戳产生相同的*datetime* 对象。另请参阅*utcfromtimestamp()*。

在 3.3 版更改: 引发*OverflowError* 而不是*ValueError*，如果时间戳数值超出所在平台 C localtime() 或 gmtime() 函数的支持范围的话。并会在 localtime() 或 gmtime() 出错时引发*OSError* 而不是*ValueError*。

在 3.6 版更改: *fromtimestamp()* 可能返回*fold* 值设为 1 的实例。

classmethod datetime.**utcfromtimestamp**(*timestamp*)

返回对应于 POSIX 时间戳的 UTC *datetime*，其中*tzinfo* 为 None。这可能引发*OverflowError*，如果时间戳数值超出所在平台 C gmtime() 函数的支持范围的话，并会在 gmtime() 出错时引发*OSError*。通常该数值并会限制在 1970 年至 2038 年之间。

要得到一个感知型*datetime* 对象，应调用*fromtimestamp()*:

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

在 POSIX 兼容的平台上，它等价于以下表达式:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

不同之处在于后一种形式总是支持完整年份范围：从*MINYEAR* 到*MAXYEAR* 的开区间。

在 3.3 版更改: 引发*OverflowError* 而不是*ValueError*，如果时间戳数值超出所在平台 C gmtime() 函数的支持范围的话。并会在 gmtime() 出错时引发*OSError* 而不是*ValueError*。

classmethod datetime.**fromordinal**(*ordinal*)

返回对应于预期格列高利历序号的*datetime*，其中公元 1 年 1 月 1 日的序号为 1。除非 1 <= ordinal <= datetime.max.toordinal() 否则会引发*ValueError*。结果的 hour, minute, second 和 microsecond 值均为 0，并且*tzinfo* 值为 None。

classmethod datetime.**combine**(*date*, *time*, *tzinfo=self.tzinfo*)

返回一个新的*datetime* 对象，对象的日期数值等于给定的*date* 对象的数值，时间数值等于给定的*time* 对象的数值。如果提供 *tzinfo* 参数，其值会被用来设置结果的*tzinfo* 属性，否则将会使用 *time* 参数的*tzinfo* 属性。

对于任意*datetime* 对象 *d*，d == datetime.combine(d.date(), d.time(), d.tzinfo)。如果 date 是一个*datetime* 对象，它的时间数值和*tzinfo* 属性会被忽略。

在 3.6 版更改: 增加了 *tzinfo* 参数。

classmethod datetime.**strptime**(*date_string*, *format*)

返回根据 *format* 解析与 *date_string* 相对应的*datetime*，这等价于 datetime(*(time.strptime(date_string, format)[0:6]))，如果 date_string 和 format 无法被*time.strptime()* 解析或它返回一个不是时间元组的值，则将引发*ValueError*。要获取格式化指令的完整列表请参阅*strftime() 和 strptime() 的行为*。

类属性:

datetime.**min**

最早的可表示*datetime*，datetime(MINYEAR, 1, 1, tzinfo=None)。

datetime.**max**

最晚的可表示*datetime*，datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)。

datetime.**resolution**

两个不相等的*datetime* 对象之间可能的最小间隔，timedelta(microseconds=1)。

实例属性（只读）:

datetime.**year**
在*MINYEAR* 和*MAXYEAR* 之间，包含边界。

datetime.**month**
1 至 12（含）

datetime.**day**
返回 1 到指定年月的天数间的数字。

datetime.**hour**
取值范围是 `range(24)`。

datetime.**minute**
取值范围是 `range(60)`。

datetime.**second**
取值范围是 `range(60)`。

datetime.**microsecond**
取值范围是 `range(1000000)`。

datetime.**tzinfo**
作为 *tzinfo* 参数被传给*datetime* 构造器的对象，如果没有传入值则为 `None`。

datetime.**fold**
取值范围是 `[0, 1]`。用于在重复的时间段中消除边界时间歧义。（当夏令时结束时回调时钟或由于政治原因导致当前时区的 UTC 时差减少就会出现重复的时间段。）取值 0 (1) 表示两个时刻早于（晚于）所代表的同一边界时间。

3.6 新版功能.

支持的运算：

运算	结果：
`datetime2 = datetime1 + timedelta`	(1)
`datetime2 = datetime1 - timedelta`	(2)
`timedelta = datetime1 - datetime2`	(3)
`datetime1 < datetime2`	比较*datetime* 与*datetime*。(4)

(1) datetime2 是从中去掉的一段 timedelta 的结果，如果 `timedelta.days` > 0 则是在时间线上前进，如果 `timedelta.days` < 0 则后退。结果具有与输入的 datetime 相同的*tzinfo* 属性，并且操作完成后 datetime2 - datetime1 == timedelta。如果 datetime2.year 将小于*MINYEAR* 或大于*MAXYEAR* 则会引发*OverflowError*。请注意即使输入的是一个感知型对象，该方法也不会进行时区调整。

(2) 计算 datetime2 使得 datetime2 + timedelta == datetime1。与相加操作一样，结果具有与输入的 datetime 相同的*tzinfo* 属性，即使输入的是一个感知型对象，该方法也不会进行时区调整。

(3) 从一个*datetime* 减去一个*datetime* 仅对两个操作数均为简单型或均为感知型时有定义。如果一个是感知型而另一个是简单型，则会引发*TypeError*。

如果两个操作数都是简单型，或都是感知型且具有相同的*tzinfo* 属性，*tzinfo* 属性会被忽略，结果是一个使得 `datetime2 + t == datetime1` 的*timedelta* 对象 *t*。在此情况下不会进行时区调整。

如果两个操作数都是感知型且具有不同的*tzinfo* 属性，`a-b` 操作的行为就如同 *a* 和 *b* 被首先转换为简单型 UTC 日期时间。结果将是 `(a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())` 除非具体实现绝对不溢出。

(4) 当 *datetime1* 的时间在 *datetime2* 之前则认为 *datetime1* 小于 *datetime2*。

如果比较的一方是简单型而另一方是感知型，则如果尝试进行顺序比较将引发*TypeError*。对于相等比较，简单型实例将永远不等于感知型实例。

如果两个比较方都是感知型，且具有相同的*tzinfo*属性，相同的*tzinfo*属性会被忽略并对基本日期时间值进行比较。如果两个比较方都是感知型且具有不同的*tzinfo*属性，两个比较方将首先通过减去它们的 UTC 差值（使用 `self.utcoffset()` 获取）来进行调整。

在 3.3 版更改: 简单型和感知型*datetime*实例之间的相等比较不会引发*TypeError*。

注解： 为了防止比较操作回退为默认的对象地址比较方案，datetime 比较通常会引发*TypeError*，如果比较目标不同样为*datetime*对象的话。不过也可能会返回 `NotImplemented` 如果比较目标具有 `timetuple()` 属性的话。这个钩子给予其他日期对象类型实现混合类型比较的机会。否则，当*datetime*对象与不同类型的对象比较时将会引发*TypeError*，除非 == 或 != 比较。后两种情况将分别返回*False*或*True*。

---

*datetime*对象可以用作字典的键。在布尔运算时，所有*datetime*对象都被视为真值。

实例方法:

`datetime.`**`date`**`()`
返回具有同样 year, month 和 day 值的*date*对象。

`datetime.`**`time`**`()`
返回具有同样 hour, minute, second, microsecond 和 fold 值的*time*对象。*tzinfo*为 `None`。另请参见方法*timetz()*。

在 3.6 版更改: fold 值会被复制给返回的*time*对象。

`datetime.`**`timetz`**`()`
返回具有同样 hour, minute, second, microsecond, fold 和 tzinfo 属性值的*time*对象。另请参见方法*time()*。

在 3.6 版更改: fold 值会被复制给返回的*time*对象。

`datetime.`**`replace`**`(`*year=self.year*, *month=self.month*, *day=self.day*, *hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, * *fold=0*`)`
返回一个具有同样属性值的 datetime，除非通过任何关键字参数指定了某些属性值。请注意可以通过指定 `tzinfo=None` 从一个感知型 datetime 创建一个简单型 datetime 而不必转换日期和时间值。

3.6 新版功能: 增加了 `fold` 参数。

`datetime.`**`astimezone`**`(`*tz=None*`)`
返回一个具有新的*tzinfo*属性 *tz* 的*datetime*对象，并会调整日期和时间数据使得结果对应的 UTC 时间与 *self* 相同，但为 *tz* 时区的本地时间。

tz 如果给出则必须是一个*tzinfo*子类的实例，并且其*utcoffset()*和*dst()*方法不可返回 `None`。如果 *self* 为简单型，它会被假定为基于系统时区表示的时间。

如果调用时不传入参数（或传入 `tz=None`) 则将假定目标时区为系统的本地时区。转换后 datetime 实例的 `.tzinfo` 属性将被设为一个*timezone*实例，时区名称和时差值将从系统获取。

如果 `self.tzinfo` 为 *tz*，`self.astimezone(tz)` 等于 *self*: 不会对日期或时间数据进行调整。否则结果为 *tz* 时区的本地时间，代表的 UTC 时间与 *self* 相同：在 `astz = dt.astimezone(tz)` 之后，`astz - astz.utcoffset()` 将具有与 `dt - dt.utcoffset()` 相同的日期和时间数据。

如果你只想附加一个时区对象 *tz* 给一个 datetime 对象 *dt* 而不调整日期和时间数据，请使用 `dt.replace(tzinfo=tz)`。如果你只想从一个感知型 datetime 对象 *dt* 移除时区对象则不转换日期和时间数据，请使用 `dt.replace(tzinfo=None)`。

请注意默认的*tzinfo.fromutc()*方法在*tzinfo*的子类中可以被重载，从而影响*astimezone()*的返回结果。如果忽略出错的情况，*astimezone()*的行为就类似于:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

在 3.3 版更改: *tz* 现在可以被省略。

在 3.6 版更改: *astimezone()* 方法可以由简单型实例调用，这将假定其表示本地时间。

datetime.**utcoffset**()
If *tzinfo* is None, returns None, else returns self.tzinfo.utcoffset(self), and raises an exception if the latter doesn't return None, or a *timedelta* object representing a whole number of minutes with magnitude less than one day.

datetime.**dst**()
If *tzinfo* is None, returns None, else returns self.tzinfo.dst(self), and raises an exception if the latter doesn't return None, or a *timedelta* object representing a whole number of minutes with magnitude less than one day.

datetime.**tzname**()
如果*tzinfo* 为 None，则返回 None，否则返回 self.tzinfo.tzname(self)，如果后者不返回 None 或者一个字符串对象则将引发异常。

datetime.**timetuple**()
返回一个*time.struct_time*，即与*time.localtime()* 的返回类型相同。d.timetuple() 等价于 time.struct_time((d.year, d.month, d.day, d.hour, d.minute, d.second, d.weekday(), yday, dst))，其中 yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 是日期在当前年份中的序号，起始序号 1 表示 1 月 1 日。结果的 tm_isdst 旗标的设定会依据*dst()* 方法：如果*tzinfo* 为 None 或*dst()* 返回 None，则 tm_isdst 将设为 -1；否则如果*dst()* 返回一个非零值，则 tm_isdst 将设为 1；否则 tm_isdst 将设为 0。

datetime.**utctimetuple**()
如果*datetime* 实例 *d* 为简单型，这类似于 d.timetuple()，不同之处为 tm_isdst 会强设为 0，无论 d.dst() 返回什么结果。DST 对于 UTC 时间永远无效。

如果 *d* 为感知型，*d* 会通过减去 d.utcoffset() 被标准化为 UTC 时间，并返回标准化时间对应的*time.struct_time*。tm_isdst 会强设为 0。请注意如果 *d*.year 为 MINYEAR 或 MAXYEAR 并且 UTC 调整超出一年的边界则可能引发*OverflowError*。

datetime.**toordinal**()
返回日期的预期格列高利历序号。与 self.date().toordinal() 相同。

datetime.**timestamp**()
返回对应于*datetime* 实例的 POSIX 时间戳。返回值是与*time.time()* 类似的*float*。

简单型*datetime* 实例会假定为代表本地时间，并且此方法依赖于平台的 C mktime() 函数来执行转换。由于在许多平台上*datetime* 支持的值范围比 mktime() 更广，对于极其遥远的过去或未来时间此方法可能引发*OverflowError*。

对于感知型*datetime* 实例，返回值的计算方式为:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

3.3 新版功能.

在 3.6 版更改: *timestamp()* 方法使用*fold* 属性来消除重复间隔中的时间歧义。

注解： 没有一个方法能直接从简单型*datetime*实例获取 POSIX 时间戳来代表 UTC 时间。如果你的应用使用此惯例方式并且你的系统时区不是设为 UTC，你可以通过提供 `tzinfo=timezone.utc` 来获取 POSIX 时间戳:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

或者通过直接计算时间戳:

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

`datetime.`**`weekday`**`()`
返回一个整数代表星期几，星期一为 0，星期天为 6。相当于 `self.date().weekday()`。另请参阅*isoweekday()*。

`datetime.`**`isoweekday`**`()`
返回一个整数代表星期几，星期一为 1，星期天为 7。相当于 `self.date().isoweekday()`。另请参阅*weekday()*, *isocalendar()*。

`datetime.`**`isocalendar`**`()`
返回一个 3 元组 (ISO 年份, ISO 周序号, ISO 周日期)。相当于 `self.date().isocalendar()`。

`datetime.`**`isoformat`**`(`*sep='T'*, *timespec='auto'*`)`
Return a string representing the date and time in ISO 8601 format, YYYY-MM-DDTHH:MM:SS.mmmmmm or, if *microsecond* is 0, YYYY-MM-DDTHH:MM:SS

If *utcoffset()* does not return `None`, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: YYYY-MM-DDTHH:MM:SS.mmmmmm+HH:MM or, if *microsecond* is 0 YYYY-MM-DDTHH:MM:SS+HH:MM

可选参数 *sep* (默认为 `'T'`) 为单个分隔字符，会被放在结果的日期和时间两部分之间。例如

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     def utcoffset(self, dt): return timedelta(minutes=-399)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
```

可选参数 *timespec* 要包含的额外时间组件值 (默认为 `'auto'`)。它可以是以下值之一:

- `'auto'`: 如果*microsecond* 为 0 则与 `'seconds'` 相同，否则与 `'microseconds'` 相同。
- `'hours'`: 以两个数码的 HH 格式包含*hour*。
- `'minutes'`: 以 HH:MM 格式包含*hour* 和*minute*。
- `'seconds'`: 以 HH:MM:SS 格式包含*hour*, *minute* 和*second*。
- `'milliseconds'`: 包含完整时间，但将秒值的小数部分截断至微秒。格式为 HH:MM:SS.sss。
- `'microseconds'`: Include full time in HH:MM:SS.mmmmmm format.

注解： 排除掉的时间部分将被截断，而不是被舍入。

对于无效的 *timespec* 参数将引发*ValueError*。

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

3.6 新版功能: 增加了 *timespec* 参数。

datetime.**__str__**()
对于*datetime* 实例 *d*，`str(d)` 等价于 `d.isoformat(' ')`。

datetime.**ctime**()
返回一个代表日期和时间的字符串，例如 `datetime(2002, 12, 4, 20, 30, 40).ctime() == 'Wed Dec 4 20:30:40 2002'`。在原生 C `ctime()` 函数 (*time.ctime()* 会发起调用该函数，但*datetime.ctime()* 则不会) 遵循 C 标准的平台上，`d.ctime()` 等价于 `time.ctime(time.mktime(d.timetuple()))`。

datetime.**strftime**(*format*)
返回一个由显式格式字符串所指明的代表日期和时间的字符串，要获取格式指令的完整列表请参阅*strftime() 和 strptime() 的行为*。

datetime.**__format__**(*format*)
与*datetime.strftime()* 相同。此方法使得为*datetime* 对象指定以 格式化字符串字面值表示的格式化字符串以及使用*str.format()* 进行格式化成为可能。要获取格式指令的完整列表，请参阅*strftime() 和 strptime() 的行为*。

使用 datetime 对象的例子:

```
>>> from datetime import datetime, date, time
>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)
>>> # Using datetime.now() or datetime.utcnow()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
>>> datetime.utcnow()
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060)
>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)
>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None
```

(下页继续)

(续上页)

```
>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO year
47      # ISO week
2       # ISO weekday
>>> # Formatting datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day",
↪"month", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

使用 datetime 并附带 tzinfo:

```
>>> from datetime import timedelta, datetime, tzinfo
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1) + self.dst(dt)
...     def dst(self, dt):
...         # DST starts last Sunday in March
...         d = datetime(dt.year, 4, 1)   # ends last Sunday in October
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self,dt):
...          return "GMT +1"
...
>>> class GMT2(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=2) + self.dst(dt)
...     def dst(self, dt):
...         d = datetime(dt.year, 4, 1)
...         self.dston = d - timedelta(days=d.weekday() + 1)
...         d = datetime(dt.year, 11, 1)
...         self.dstoff = d - timedelta(days=d.weekday() + 1)
...         if self.dston <=  dt.replace(tzinfo=None) < self.dstoff:
...             return timedelta(hours=1)
...         else:
...             return timedelta(0)
...     def tzname(self,dt):
...         return "GMT +2"
...
>>> gmt1 = GMT1()
>>> # Daylight Saving Time
>>> dt1 = datetime(2006, 11, 21, 16, 30, tzinfo=gmt1)
>>> dt1.dst()
datetime.timedelta(0)
>>> dt1.utcoffset()
datetime.timedelta(0, 3600)
```

(下页继续)

(续上页)

```
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=gmt1)
>>> dt2.dst()
datetime.timedelta(0, 3600)
>>> dt2.utcoffset()
datetime.timedelta(0, 7200)
>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(GMT2())
>>> dt3
datetime.datetime(2006, 6, 14, 14, 0, tzinfo=<GMT2 object at 0x...>)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=<GMT1 object at 0x...>)
>>> dt2.utctimetuple() == dt3.utctimetuple()
True
```

8.1.5 time 对象

一个 time 对象代表某个日期内的（本地）时间，它独立于任何特定日期，并可通过*tzinfo*对象来调整。

class datetime.**time**(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

All arguments are optional. *tzinfo* may be None, or an instance of a *tzinfo* subclass. The remaining arguments may be integers, in the following ranges:

- 0 <= hour < 24,
- 0 <= minute < 60,
- 0 <= second < 60,
- 0 <= microsecond < 1000000,
- fold in [0, 1].

如果给出一个此范围以外的参数，则会引发*ValueError*。所有参数值默认为 0，除了 *tzinfo* 默认为*None*。

类属性：

time.**min**

早最的可表示*time*, time(0, 0, 0, 0)。

time.**max**

最晚的可表示*time*, time(23, 59, 59, 999999)。

time.**resolution**

两个不相等的*time* 对象之间可能的最小间隔，timedelta(microseconds=1)，但是请注意*time* 对象并不支持算术运算。

实例属性（只读）：

time.**hour**

取值范围是 range(24)。

time.**minute**

取值范围是 range(60)。

time.**second**

取值范围是 range(60)。

time.**microsecond**

取值范围是 range(1000000)。

time.**tzinfo**
作为 tzinfo 参数被传给*time* 构造器的对象，如果没有传入值则为 None。

time.**fold**
取值范围是 [0, 1]。用于在重复的时间段中消除边界时间歧义。（当夏令时结束时回调时钟或由于政治原因导致当前时区的 UTC 时差减少就会出现重复的时间段。）取值 0 (1) 表示两个时刻早于（晚于）所代表的同一边界时间。

3.6 新版功能.

支持的运算：

- 比较*time* 和另一个*time*，当 *a* 的时间在 *b* 之前时，则认为 *a* 小于 *b*。如果比较的一方是简单型而另一方是感知型，则如果尝试进行顺序比较将引发*TypeError*。对于相等比较，简单型实例将永远不等于感知型实例。

  如果两个比较方都是感知型，且具有相同的*tzinfo* 属性，相同的*tzinfo* 属性会被忽略并对基本时间值进行比较。如果两个比较方都是感知型且具有不同的*tzinfo* 属性，两个比较方将首先通过减去它们的 UTC 差值（使用 self.utcoffset() 获取）来进行调整。为了防止将混合类型比较回退为基于对象地址的默认比较，当*time* 对象与不同类型的对象比较时，将会引发*TypeError*，除非比较运算符是 == 或 !=。在后一种情况下将分别返回*False* 或*True*。

  在 3.3 版更改: 简单型和感知型*time* 实例之前的相等比较不会引发*TypeError*。

- 哈希，以便用作字典的键
- 高效的封存

在布尔运算时，*time* 对象总是被视为真值。

在 3.5 版更改: 在 Python 3.5 之前，如果一个*time* 对象代表 UTC 午夜零时则会被视为假值。此行为被认为容易引发困惑和错误，因此从 Python 3.5 起已被去除。详情参见 bpo-13936。

实例方法：

time.**replace**(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, * *fold=0*)
返回一个具有同样属性值的*time*，除非通过任何关键字参数指定了某些属性值。请注意可以通过指定 tzinfo=None 从一个感知型*time* 创建一个简单型*time* 而不必转换时间值。

3.6 新版功能: 增加了 fold 参数。

time.**isoformat**(*timespec='auto'*)
Return a string representing the time in ISO 8601 format, HH:MM:SS.mmmmmm or, if *microsecond* is 0, HH:MM:SS If *utcoffset()* does not return None, a 6-character string is appended, giving the UTC offset in (signed) hours and minutes: HH:MM:SS.mmmmmm+HH:MM or, if self.microsecond is 0, HH:MM:SS+HH:MM

可选参数 *timespec* 要包含的额外时间组件值 (默认为 'auto')。它可以是以下值之一：

- 'auto': 如果*microsecond* 为 0 则与 'seconds' 相同，否则与 'microseconds' 相同。
- 'hours': 以两个数码的 HH 格式包含*hour*。
- 'minutes': 以 HH:MM 格式包含*hour* 和*minute*。
- 'seconds': 以 HH:MM:SS 格式包含*hour*, *minute* 和*second*。
- 'milliseconds': 包含完整时间，但将秒值的小数部分截断至微秒。格式为 HH:MM:SS.sss。
- 'microseconds': Include full time in HH:MM:SS.mmmmmm format.

注解: 排除掉的时间部分将被截断，而不是被舍入。

对于无效的 *timespec* 参数将引发 *ValueError*。

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec=
↪'minutes')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

3.6 新版功能: 增加了 *timespec* 参数。

time.**__str__**()
对于时间对象 *t*, `str(t)` 等价于 `t.isoformat()`。

time.**strftime**(*format*)
返回一个由显式格式字符串所指明的代表时间的字符串。要获取格式指令的完整列表请参阅*strftime() 和 strptime() 的行为*。

time.**__format__**(*format*)
与*time.strftime()* 相同。此方法使得为*time* 对象指定以 格式化字符串字面值表示的格式化字符串以及使用*str.format()* 进行格式化成为可能。要获取格式指令的完整列表，请参阅*strftime() 和 strptime() 的行为*。

time.**utcoffset**()
If *tzinfo* is None, returns None, else returns `self.tzinfo.utcoffset(None)`, and raises an exception if the latter doesn't return None or a *timedelta* object representing a whole number of minutes with magnitude less than one day.

time.**dst**()
If *tzinfo* is None, returns None, else returns `self.tzinfo.dst(None)`, and raises an exception if the latter doesn't return None, or a *timedelta* object representing a whole number of minutes with magnitude less than one day.

time.**tzname**()
如果*tzinfo* 为 None，则返回 None，否则返回 `self.tzinfo.tzname(None)`，如果后者不返回 None 或者一个字符串对象则将引发异常。

示例:

```
>>> from datetime import time, tzinfo, timedelta
>>> class GMT1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "Europe/Prague"
...
>>> t = time(12, 10, 30, tzinfo=GMT1())
>>> t
datetime.time(12, 10, 30, tzinfo=<GMT1 object at 0x...>)
>>> gmt = GMT1()
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
```

(下页继续)

(续上页)

```
>>> t.tzname()
'Europe/Prague'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 Europe/Prague'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.6 tzinfo 对象

class datetime.**tzinfo**

这是一个抽象基类，即这个类不可直接被实例化。你必须从该类派生一个实体子类，并且（至少）提供你使用*datetime* 需要的标准*tzinfo* 方法的实现。*datetime* 模块提供了*tzinfo* 的一个简单实体子类，*timezone*，它能以与 UTC 的固定差值来表示不同的时区，例如 UTC 本身或北美的 EST 和 EDT。

tzinfo 的（某个实体子类）的实例可以被传给*datetime* 和*time* 对象的构造器。这些对象会将它们的属性视为对应于本地时间，并且*tzinfo* 对象支持展示本地时间与 UTC 的差值、时区名称以及 DST 差值的方法，都是与传给它们的日期或时间对象的相对值。

对于封存操作的特殊要求：一个*tzinfo* 子类必须具有可不带参数调用的 `__init__()` 方法，否则它虽然可以被封存，但可能无法再次解封。这是个技术性要求，在未来可能会被取消。

一个*tzinfo* 的实体子类可能需要实现以下方法。具体需要实现的方法取决于感知型*datetime* 对象如何使用它。如果有疑问，可以简单地全都实现。

tzinfo.**utcoffset**(*dt*)

Return offset of local time from UTC, in minutes east of UTC. If local time is west of UTC, this should be negative. Note that this is intended to be the total offset from UTC; for example, if a *tzinfo* object represents both time zone and DST adjustments, *utcoffset()* should return their sum. If the UTC offset isn't known, return `None`. Else the value returned must be a *timedelta* object specifying a whole number of minutes in the range -1439 to 1439 inclusive (1440 = 24*60; the magnitude of the offset must be less than one day). Most implementations of *utcoffset()* will probably look like one of these two:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

如果*utcoffset()* 返回值不为 `None`，则*dst()* 也不应返回 `None`。

默认的*utcoffset()* 实现会引发*NotImplementedError*。

tzinfo.**dst**(*dt*)

Return the daylight saving time (DST) adjustment, in minutes east of UTC, or `None` if DST information isn't known. Return `timedelta(0)` if DST is not in effect. If DST is in effect, return the offset as a *timedelta* object (see *utcoffset()* for details). Note that DST offset, if applicable, has already been added to the UTC offset returned by *utcoffset()*, so there's no need to consult *dst()* unless you're interested in obtaining DST info separately. For example, *datetime.timetuple()* calls its *tzinfo* attribute's *dst()* method to determine how the `tm_isdst` flag should be set, and *tzinfo.fromutc()* calls *dst()* to account for DST changes when crossing time zones.

一个可以同时处理标准时和夏令时的*tzinfo* 子类的实例 *tz* 必须在此情形中保持一致：

```
tz.utcoffset(dt) - tz.dst(dt)
```

必须为具有同样的*tzinfo* 子类实例 `dt.tzinfo == tz` 的每个*datetime* 对象 *dt* 返回同样的结果，此表达式会产生时区的“标准时差”，它不应取决于具体日期或时间，只取决于地理位置。*datetime.astimezone()* 的实现依赖此方法，但无法检测违反规则的情况；确保符合规则是程序员的责任。如

果一个*tzinfo* 子类不能保证这一点，也许应该重载*tzinfo.fromutc()* 的默认实现以便在任何情况下与 astimezone() 配合正常。

大多数*dst()* 的实现可能会如以下两者之一:

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

或者

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.  Then

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

默认的*dst()* 实现会引发*NotImplementedError*。

tzinfo.**tzname**(*dt*)

将对应于*datetime* 对象 *dt* 的时区名称作为字符串返回。*datetime* 模块没有定义任何字符串名称相关内容，也不要求名称有任何特定含义。例如，" GMT"，"UTC"，"-500"，"-5:00"，"EDT"，"US/Eastern"，"America/New York" 都是有效的返回值。如果字符串名称未知则返回 None。请注意这是一个方法而不是一个固定的字符串，这主要是因为某些*tzinfo* 子类可能需要根据所传入的特定 *dt* 值返回不同的名称，特别是当*tzinfo* 类要负责处理夏令时的时候。

默认的*tzname()* 实现会引发*NotImplementedError*。

这些方法会被*datetime* 或*time* 对象调用，用来对应它们的同名方法。*datetime* 对象会将自身作为传入参数，而*time* 对象会将 None 作为传入参数。这样*tzinfo* 子类的方法应当准备好接受 *dt* 参数值为 None 或是*datetime* 类的实例。

当传入 None 时，应当由类的设计者来决定最佳回应方式。例如，返回 None 适用于希望该类提示时间对象不参与*tzinfo* 协议处理。让 utcoffset(None) 返回标准 UTC 时差也许会更有用处，如果没有其他用于发现标准时差的约定。

当传入一个*datetime* 对象来回应*datetime* 方法时，dt.tzinfo 与 *self* 是同一对象。*tzinfo* 方法可以依赖这一点，除非用户代码直接调用了*tzinfo* 方法。此行为的目的是使得*tzinfo* 方法将 *dt* 解读为本地时间，而不需要担心其他时区的相关对象。

还有一个额外的*tzinfo* 方法，某个子类可能会希望重载它:

tzinfo.**fromutc**(*dt*)

此方法会由默认的*datetime.astimezone()* 实现来调用。当被调用时，dt.tzinfo 为 *self*，并且 *dt* 的日期和时间数据会被视为代表 UTC 时间。*fromutc()* 的目标是调整日期和时间数据，返回一个等价的 datetime 来表示 *self* 的本地时间。

大多数*tzinfo* 子类应该能够毫无问题地继承默认的*fromutc()* 实现。它的健壮性足以处理固定差值的时区以及同时负责标准时和夏令时的时区，对于后者甚至还能处理 DST 转换时间在各个年份有变化的情况。一个默认*fromutc()* 实现可能无法在所有情况下正确处理的例子是（与 UTC 的）标准差值取决于所经过的特定日期和时间，这种情况可能由于政治原因而出现。默认的 astimezone() 和*fromutc()* 实现可能无法生成你希望的结果，如果这个结果恰好是跨越了标准差值发生改变的时刻当中的某个小时值的话。

忽略针对错误情况的代码，默认*fromutc()* 实现的行为方式如下:

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

Example *tzinfo* classes:

```
from datetime import tzinfo, timedelta, datetime, timezone

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
```

(下页继续)

(续上页)

```
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
```

(下页继续)

(续上页)

```
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname

    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO
```

(下页继续)

(续上页)

```
    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time


Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

请注意同时负责标准时和夏令时的*tzinfo*子类在每年两次的 DST 转换点上会出现不可避免的微妙问题。具体而言，考虑美国东部时区 (UTC -0500)，它的 EDT 从三月的第二个星期天 1:59 (EST) 之后一分钟开始，并在十一月的第一个星期天 1:59 (EDT) 之后一分钟结束:

```
  UTC    3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST   22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
  EDT   23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start   22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end   23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

当 DST 开始时（见“start”行），本地时钟从 1:59 跳到 3:00。形式为 2:MM 的时间值在那一天是没有意义的，因此在 DST 开始的那一天 `astimezone(Eastern)` 不会输出包含 `hour == 2` 的结果。例如，在 2016 年春季时钟向前调整时，我们得到

```
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

当 DST 结束时（见“end”行），会有更糟糕的潜在问题：本地时间值中有一个小时是不可能没有歧义的：夏令时的最后一小时。即以美国东部时间表示当天夏令时结束时的形式为 5:MM UTC 的时间。本地时间从 1:59 (夏令时) 再次跳回到 1:00 (标准时)。形式为 1:MM 的本地时间就是有歧义的。此时 `astimezone()` 是通过将两个相邻的 UTC 小时映射到两个相同的本地小时来模仿本地时钟的行为。在这个美国东部时间的示例中，形式为 5:MM 和 6:MM 的 UTC 时间在转换为美国东部时间时都将被映射到 1:MM，但前一个时间会将*fold*

属性设为 0 而后一个时间会将其设为 1。例如，在 2016 年秋季时钟往回调整时，我们得到

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

请注意不同的*datetime*实例仅通过*fold*属性值来加以区分，它们在比较时会被视为相等。

不允许时间显示存在歧义的应用需要显式地检查*fold*属性的值，或者避免使用混合式的*tzinfo*子类；当使用*timezone*或者任何其他固定差值的*tzinfo*子类例如仅表示 EST（固定差值 -5 小时）或仅表示 EDT（固定差值 -4 小时）的类时是不会有歧义的。

参见:

dateutil.tz 该标准库具有*timezone*类用来将相对于 UTC 和*timezone.utc*的任意固定差值处理为 UTC 时区实例。

dateutil.tz 库将 *IANA* 时区数据库 (又名 Olson 数据库) 引入 Python 并推荐使用。

IANA 时区数据库 该时区数据库 (通常称为 tz, tzdata 或 zoneinfo) 包含大量代码和数据用来表示全球许多有代表性的地点的本地时间的历史信息。它会定期进行更新以反映各政治实体对时区边界、UTC 差值和夏令时规则的更改。

8.1.7 timezone 对象

*timezone*类是*tzinfo*的一个子类，它的每个实例代表一个由与 UTC 的固定差值所定义的时区。请注意该类的对象不可被用于代表某些特殊地点的时区信息，例如在一年的不同日期使用不同差值，或是在历史上对民用时间进行过调整的地点。

class datetime.**timezone**(*offset*, *name=None*)

The *offset* argument must be specified as a *timedelta* object representing the difference between the local time and UTC. It must be strictly between `-timedelta(hours=24)` and `timedelta(hours=24)` and represent a whole number of minutes, otherwise *ValueError* is raised.

name 参数是可选的。如果指定则必须为一个字符串，它将被用作*datetime.tzname()*方法的返回值。

3.2 新版功能.

timezone.**utcoffset**(*dt*)

返回在构造*timezone*实例时指定的固定值。*dt* 参数会被忽略。返回值是一个*timedelta*实例，其值等于本地时间与 UTC 之间的差值。

timezone.**tzname**(*dt*)

返回在构造*timezone*实例时指定的固定值。如果没有为构造器提供 *name*，则 `tzname(dt)` 所返回的名称将根据 `offset` 值按以下规则生成。如果 *offset* 为 `timedelta(0)`，则名称为 “UTC”，否则为字符串 ‘UTC±HH:MM’，其中 ± 为 `offset` 值的正负，HH 和 MM 分别为表示 `offset.hours` 和 `offset.minutes` 的两个数码。

在 3.6 版更改: 由 `offset=timedelta(0)` 生成的名称现在改为简单的 ‘UTC’ 而不再是 ‘UTC+00:00’。

timezone.**dst**(*dt*)

总是返回 `None`。

timezone.**fromutc**(*dt*)
返回 dt + offset。*dt* 参数必须为一个感知型*datetime* 实例，其中 tzinfo 值设为 self。

类属性：

timezone.**utc**
UTC 时区，timezone(timedelta(0))。

8.1.8 strftime() 和 strptime() 的行为

date, *datetime* 和*time* 对象都支持 strftime(format) 方法，可用来创建一个由指定格式字符串所控制的表示时间的字符串。总体而言，d.strftime(fmt) 类似于*time* 模块的 time.strftime(fmt, d.timetuple()) 但是并非所有对象都支持 timetuple() 方法。

相反地，*datetime.strptime()* 类方法可根据一个表示时间的字符串和对应的格式字符串创建来一个*datetime* 对象。datetime.strptime(date_string, format) 等价于 datetime(*(time.strptime(date_string, format)[0:6]))，差别在于当 format 包含小于秒的部分或者时区差值信息的时候，这些信息被 datetime.strptime 所支持但会被 time.strptime 所丢弃。

对于*time* 对象，年、月、日的格式代码不应被使用，因为 time 对象没有这些值。如果它们被使用，年份将被替换为 1900 而月和日将被替换为 1。

对于*date* 对象，时、分、秒和微秒的格式代码不应被使用，因为*date* 对象没有这些值。如果它们被使用，它们都将被替换为 0。

对完整格式代码集的支持在不同平台上有所差异，因为 Python 要调用所在平台 C 库的 strftime() 函数，而不同平台的差异是很常见的。要查看你所用平台所支持的完整格式代码集，请参阅 *strftime(3)* 文档。

以下列表显示了 C 标准（1989 版）所要求的全部格式代码，它们在带有标准 C 实现的所有平台上均可用。请注意 1999 版 C 标准又添加了额外的格式代码。

指令	含义	示例	注释
%a	当地工作日的缩写。	Sun, Mon, …, Sat (美国); So, Mo, …, Sa (德国)	(1)
%A	当地工作日的全名。	Sunday, Monday, …, Saturday (美国); Sonntag, Montag, …, Samstag (德国)	(1)
%w	以十进制数显示的工作日，其中 0 表示星期日，6 表示星期六。	0, 1, …, 6	
%d	补零后，以十进制数显示的月份中的一天。	01, 02, …, 31	
%b	当地月份的缩写。	Jan, Feb, …, Dec (美国); Jan, Feb, …, Dez (德国)	(1)
%B	当地月份的全名。	January, February, …, December (美国); Januar, Februar, …, Dezember (德国)	(1)
%m	补零后，以十进制数显示的月份。	01, 02, …, 12	
%y	补零后，以十进制数表示的，不带世纪的年份。	00, 01, …, 99	
%Y	十进制数表示的带世纪的年份。	0001, 0002, …, 2013, 2014, …, 9998, 9999	(2)
%H	以补零后的十进制数表示的小时（24 小时制）。	00, 01, …, 23	
%I	以补零后的十进制数表示的小时（12 小时制）。	01, 02, …, 12	
%p	本地化的 AM 或 PM 。	AM, PM (美国); am, pm (德国)	(1), (3)
%M	补零后，以十进制数显示的分钟。	00, 01, …, 59	
%S	补零后，以十进制数显示的秒。	00, 01, …, 59	(4)
%f	以十进制数表示的微秒，在左侧补零。	000000, 000001, …, 999999	(5)
%z	UTC offset in the form +HHMM or -HHMM (empty string if the object is naive).	(empty), +0000, -0400, +1030	(6)
%Z	时区名称（如果对象为简单型则为空字符串）。	(空), UTC, EST, CST	
%j	以补零后的十进制数表示的一年中的日序号。	001, 002, …, 366	
%U	以补零后的十进制数	00, 01, …, 53	(7)

有一些并非 C89 标准所要求的额外指令是为使用方便而加入的。这些形参都与 ISO 8601 日期值相对应。它们可能不是在所有平台上都可通过 strftime() 方法使用。ISO 8601 年份和 ISO 8601 星期指令并不能与上面的年份和星期序号指令相互替代。调用 strptime() 时传入不完整或有歧义的 ISO 8601 指令将引发*ValueError*。

指令	含义	示例	注释
%G	带有世纪的 ISO 8601 年份，表示包含大部分 ISO 星期 (%V) 的年份。	0001, 0002, …, 2013, 2014, …, 9998, 9999	(8)
%u	以十进制数显示的 ISO 8601 星期中的日序号，其中 1 表示星期一。	1, 2, …, 7	
%V	以十进制数显示的 ISO 8601 星期，以星期一作为每周的第一天。第 01 周为包含 1 月 4 日的星期。	01, 02, …, 53	(8)

3.6 新版功能: 增加了 %G, %u 和 %V。

注释:

(1) 由于此格式依赖于当前区域设置，因此对具体输出值应当保持谨慎预期。字段顺序会发生改变（例如 “month/day/year” 与 “day/month/year”），并且输出可能包含使用区域设置所指定的默认编码格式的 Unicode 字符（例如如果当前区域为 ja_JP，则默认编码格式可能为 eucJP, SJIS 或 utf-8 中的一个；使用*locale.getlocale()* 可确定当前区域设置的编码格式）。

(2) strptime() 方法能够解析整个 [1, 9999] 范围内的年份，但 < 1000 的年份必须加零填充为 4 位数字宽度。

在 3.2 版更改: 在之前的版本中，strftime() 方法只限于 >= 1900 的年份。

在 3.3 版更改: 在版本 3.2 中，strftime() 方法只限于 years >= 1000。

(3) 当与 strptime() 方法一起使用时，如果使用 %I 指令来解析小时，%p 指令只影响输出小时字段。

(4) 与*time* 模块不同的是，*datetime* 模块不支持闰秒。

(5) 当与 strptime() 方法一起使用时，%f 指令可接受一至六个数码及左边的零填充。%f 是对 C 标准中格式字符集的扩展（但单独在 datetime 对象中实现，因此它总是可用）。

(6) 对于简单型对象，%z and %Z 格式代码会被替换为空字符串。

对于一个觉悟型对象而言:

%z utcoffset() is transformed into a 5-character string of the form +HHMM or -HHMM, where HH is a 2-digit string giving the number of UTC offset hours, and MM is a 2-digit string giving the number of UTC offset minutes. For example, if utcoffset() returns timedelta(hours=-3, minutes=-30), %z is replaced with the string '-0330'.

%Z 如果 tzname() 返回 None，%Z 会被替换为一个空字符串。在其他情况下 %Z 会被替换为返回值，这必须为一个字符串。

在 3.2 版更改: 当提供 %z 指令给 strptime() 方法时，将产生一个感知型*datetime* 对象。结果的 tzinfo 将被设为一个*timezone* 实例。

(7) 当与 strptime() 方法一起使用时，%U 和 %W 仅用于指定星期几和日历年份 (%Y) 的计算。

(8) 类似于 %U 和 %W，%V 仅用于在 strptime() 格式字符串中指定星期几和 ISO 年份 (%G) 的计算。还要注意 %G 和 %Y 是不可交换的。

备注

8.2 calendar —日历相关函数

源代码： Lib/calendar.py

这个模块让你可以输出像 Unix **cal** 那样的日历，它还提供了其它与日历相关的实用函数。默认情况下，这些日历把星期一当作一周的第一天，星期天为一周的最后一天（按照欧洲惯例）。可以使用 *setfirstweekday()* 方法设置一周的第一天为星期天 (6) 或者其它任意一天。使用整数作为指定日期的参数。更多相关的函数，参见 *datetime* 和 *time* 模块。

Most of these functions and classes rely on the *datetime* module which uses an idealized calendar, the current Gregorian calendar extended in both directions. This matches the definition of the "proleptic Gregorian" calendar in Dershowitz and Reingold's book "Calendrical Calculations", where it's the base calendar for all computations.

class calendar.**Calendar**(*firstweekday=0*)

创建一个 *Calendar* 对象。*firstweekday* 是一个整数，用于指定一周的第一天。`0` 是星期一（默认值），`6` 是星期天。

Calendar 对象提供了一些可被用于准备日历数据格式化的方法。这个类本身不执行任何格式化操作。这部分任务应由子类来完成。

Calendar 类的实例有下列方法：

iterweekdays()

返回一个迭代器，迭代器的内容为一星期的数字。迭代器的第一个值与 *firstweekday* 属性的值一至。

itermonthdates(*year*, *month*)

返回一个迭代器，迭代器的内容为 *year* 年 *month* 月 (1-12) 的日期。这个迭代器返回当月的所有日期 (*datetime.date* 对象)，日期包含了本月头尾用于组成完整一周的日期。

itermonthdays2(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*. Days returned will be tuples consisting of a day number and a week day number.

itermonthdays(*year*, *month*)

Return an iterator for the month *month* in the year *year* similar to *itermonthdates()*. Days returned will simply be day numbers.

monthdatescalendar(*year*, *month*)

返回一个表示指定年月的周列表。周列表由七个 *datetime.date* 对象组成。

monthdays2calendar(*year*, *month*)

返回一个表示指定年月的周列表。周列表由七个代表日期的数字和代表周几的数字组成的二元元组。

monthdayscalendar(*year*, *month*)

返回一个表示指定年月的周列表。周列表由七个代表日期的数字组成。

yeardatescalendar(*year*, *width=3*)

返回可以用来格式化的指定年月的数据。返回的值是一个列表，列表是月份组成的行。每一行包含了最多 *width* 个月 (默认为 3)。每个月包含了 4 到 6 周，每周包含 1–7 天。每一天使用 *datetime.date* 对象。

yeardays2calendar(*year*, *width=3*)

返回可以用来模式化的指定年月的数据 (与 *yeardatescalendar()* 类似)。周列表的元素是由表示日期的数字和表示星期几的数字组成的元组。不在这个月的日子为 0。

yeardayscalendar(*year*, *width=3*)
返回可以用来模式化的指定年月的数据 (与*yeardatescalendar()* 类似)。周列表的元素是表示日期的数字。不在这个月的日子为 0。

class calendar.**TextCalendar**(*firstweekday=0*)
可以使用这个类生成纯文本日历。

TextCalendar 实例有以下方法：

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)
返回一个多行字符串来表示指定年月的日历。*w* 为日期的宽度，但始终保持日期居中。*l* 指定了每星期占用的行数。以上这些还依赖于构造器或者*setfirstweekday()* 方法指定的周的第一天是哪一天。

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)
与*formatmonth()* 方法一样，返回一个月的日历。

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
返回一个多行字符串，这个字符串为一个 *m* 列日历。可选参数 *w*, *l*, 和 *c* 分别表示日期列数，周的行数，和月之间的间隔。同样，以上这些还依赖于构造器或者*setfirstweekday()* 指定哪一天为一周的第一天。日历的第一年由平台依赖于使用的平台。

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
与*formatyear()* 方法一样，返回一整年的日历。

class calendar.**HTMLCalendar**(*firstweekday=0*)
可以使用这个类生成 HTML 日历。

HTMLCalendar instances have the following methods:

formatmonth(*theyear*, *themonth*, *withyear=True*)
返回一个 HTML 表格作为指定年月的日历。*withyear* 为真，则年份将会包含在表头，否则只显示月份。

formatyear(*theyear*, *width=3*)
返回一个 HTML 表格作为指定年份的日历。*width* (默认为 3) 用于规定每一行显示月份的数量。

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)
返回一个完整的 HTML 页面作为指定年份的日历。*width**(默认为 3) 用于规定每一行显示的月份数量。**css* 为层叠样式表的名字。如果不使用任何层叠样式表，可以使用*None* 。*encoding* 为输出页面的编码 (默认为系统的默认编码)。

class calendar.**LocaleTextCalendar**(*firstweekday=0*, *locale=None*)
这个子类*TextCalendar* 可以在构造函数中传递一个语言环境名称，并且返回月份和星期几的名称在特定语言环境中。如果此语言环境包含编码，则包含月份和工作日名称的所有字符串将作为 unicode 返回。

class calendar.**LocaleHTMLCalendar**(*firstweekday=0*, *locale=None*)
This subclass of *HTMLCalendar* can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

注解： 这两个类的 `formatweekday()` 和 `formatmonthname()` 方法临时更改 dang 当前区域至给定 *locale* 。由于当前的区域设置是进程范围的设置，因此它们不是线程安全的。

对于简单的文本日历，这个模块提供了以下方法。

calendar.**setfirstweekday**(*weekday*)
设置每一周的开始 (`0` 表示星期一，`6` 表示星期天)。calendar 还提供了 `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY` 和 `SUNDAY` 几个常量方便使用。例如，设置每周的第一天为星期天

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.**firstweekday**()
返回当前设置的每星期的第一天的数值。

calendar.**isleap**(*year*)
如果 *year* 是闰年则返回 *True*，否则返回 *False*。

calendar.**leapdays**(*y1*, *y2*)
返回在范围 *y1* 至 *y2* （包含 y1 和 y2 ）之间的闰年的年数，其中 *y1* 和 *y2* 是年份。

此函数适用于跨越一个世纪变化的范围。

calendar.**weekday**(*year*, *month*, *day*)
返回一周中的某一天（0 是周一）以年（1970–…），月（1–12），日（1–31）的格式。

calendar.**weekheader**(*n*)
返回一个包含星期几的缩写名的头。*n* 指定星期几缩写的字符宽度。

calendar.**monthrange**(*year*, *month*)
返回指定年份的指定月份第一天是星期几和这个月的天数。

calendar.**monthcalendar**(*year*, *month*)
返回表示一个月的日历的矩阵。每一行代表一周；此月份外的日子由零表示。每周从周一开始，除非使用 *setfirstweekday()* 改变设置。

calendar.**prmonth**(*theyear*, *themonth*, *w=0*, *l=0*)
打印由 *month()* 返回的一个月的日历。

calendar.**month**(*theyear*, *themonth*, *w=0*, *l=0*)
使用 *TextCalendar* 类的 formatmonth() 以多行字符串形式返回月份日历。

calendar.**prcal**(*year*, *w=0*, *l=0*, *c=6*, *m=3*)
打印由 *calendar()* 返回的整年的日历。

calendar.**calendar**(*year*, *w=2*, *l=1*, *c=6*, *m=3*)
使用 *TextCalendar* 类的 formatyear() 返回整年的三列的日历以多行字符串的形式。

calendar.**timegm**(*tuple*)
一个不相关但很好用的函数，它接受一个时间元组例如 *time* 模块中的 *gmtime()* 函数的返回并返回相应的 Unix 时间戳，假定 1970 年开始计数，POSIX 编码。实际上，*time.gmtime()* 和 *timegm()* 是彼此相反的。

calendar 模块导出以下数据属性：

calendar.**day_name**
在当前的语言环境下表示星期几。

calendar.**day_abbr**
在当前语言环境下表示星期几缩写的数组。

calendar.**month_name**
在当前语言环境下表示一年中月份的数组。这遵循一月的月号为 1 的通常惯例，所以它的长度为 13 且 month_name[0] 是空字符串。

calendar.**month_abbr**
在当前语言环境下表示月份简写的数组。这遵循一月的月号为 1 的通常惯例，所以它的长度为 13 且 month_abbr[0] 是空字符串。

参见：

模块 *datetime* 为日期和时间提供与 *time* 模块相似功能的面向对象接口。

模块*time* 底层时间相关函数。

8.3 collections —容器数据类型

Source code: Lib/collections/__init__.py

这个模块实现了特定目标的容器，以提供 Python 标准内建容器*dict*，*list*，*set*，和*tuple*的替代选择。

namedtuple()	创建命名元组子类的工厂函数
deque	类似列表 (list) 的容器，实现了在两端快速添加 (append) 和弹出 (pop)
ChainMap	类似字典 (dict) 的容器类，将多个映射集合到一个视图里面
Counter	字典的子类，提供了可哈希对象的计数功能
OrderedDict	字典的子类，保存了他们被添加的顺序
defaultdict	字典的子类，提供了一个工厂函数，为字典查询提供一个默认值
UserDict	封装了字典对象，简化了字典子类化
UserList	封装了列表对象，简化了列表子类化
UserString	封装了字符串对象，简化了字符串子类化

在 3.3 版更改: Moved *容器抽象基类* to the *collections.abc* module. For backwards compatibility, they continue to be visible in this module as well.

8.3.1 ChainMap 对象

3.3 新版功能.

一个*ChainMap*类是为了将多个映射快速的链接到一起，这样它们就可以作为一个单元处理。它通常比创建一个新字典和多次调用*update()*要快很多。

这个类可以用于模拟嵌套作用域，并且在模版化的时候比较有用。

class collections.**ChainMap**(**maps*)

一个*ChainMap*将多个字典或者其他映射组合在一起，创建一个单独的可更新的视图。如果没有 *maps* 被指定，就提供一个默认的空字典，这样一个新链至少有一个映射。

底层映射被存储在一个列表中。这个列表是公开的，可以通过 *maps* 属性存取和更新。没有其他的状态。

搜索查询底层映射，直到一个键被找到。不同的是，写，更新和删除只操作第一个映射。

一个*ChainMap*通过引用合并底层映射。所以，如果一个底层映射更新了，这些更改会反映到*ChainMap*。

支持所有常用字典方法。另外还有一个 *maps* 属性 (attribute)，一个创建子上下文的方法 (method)，一个存取它们首个映射的属性 (property):

maps

一个可以更新的映射列表。这个列表是按照第一次搜索到最后一次搜索的顺序组织的。它是仅有的存储状态，可以被修改。列表最少包含一个映射。

new_child(*m=None*)

返回一个新的*ChainMap*类，包含了一个新映射 (map)，后面跟随当前实例的全部映射 (map)。如果 m 被指定，它就成为不同新的实例，就是在所有映射前加上 m，如果没有指定，就加上一个空字典，这样的话一个 d.new_child() 调用等价于 ChainMap({}, *d.maps)。这个方法用于创建子上下文，不改变任何父映射的值。

在 3.4 版更改: 添加了 m 可选参数。

parents
属性返回一个新的*ChainMap* 包含所有的当前实例的映射，除了第一个。这样可以在搜索的时候跳过第一个映射。使用的场景类似在*nested scopes* 嵌套作用域中使用 nonlocal 关键词。用例也可以类比内建函数*super()* 。一个 d.parents 的引用等价于 ChainMap(*d.maps[1:]) 。

参见:

- MultiContext class 在 Enthought CodeTools package 有支持写映射的选项。
- Django’s Context class for templating is a read-only chain of mappings. It also features pushing and popping of contexts similar to the *new_child()* method and the *parents()* property.
- Nested Contexts recipe 提供了是否对第一个映射或其他映射进行写和其他修改的选项。
- 一个 极简的只读版 Chainmap.

ChainMap 例子和方法

这一节提供了多个使用链映射的案例。

模拟 Python 内部 lookup 链的例子

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

让用户指定的命令行参数优先于环境变量，优先于默认值的例子

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k:v for k, v in vars(namespace).items() if v}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

用*ChainMap* 类模拟嵌套上下文的例子

```
c = ChainMap()          # Create root context
d = c.new_child()       # Create nested child context
e = c.new_child()       # Child of c, independent from d
e.maps[0]               # Current context dictionary -- like Python's locals()
e.maps[-1]              # Root context -- like Python's globals()
e.parents               # Enclosing context chain -- like Python's nonlocals

d['x']                  # Get first key in the chain of contexts
d['x'] = 1              # Set value in current context
del d['x']              # Delete from current context
list(d)                 # All nested values
k in d                  # Check all nested values
len(d)                  # Number of nested values
d.items()               # All nested items
dict(d)                 # Flatten into a regular dictionary
```

ChainMap 类只更新链中的第一个映射，但 lookup 会搜索整个链。然而，如果需要深度写和删除，也可以很容易的通过定义一个子类来实现它

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'         # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
>>> d                            # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.3.2 Counter 对象

一个计数器工具提供快速和方便的计数。比如

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.**Counter**([*iterable-or-mapping*])

A *Counter* is a *dict* subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The *Counter* class is similar to bags or multisets in other languages.

元素从一个 *iterable* 被计数或从其他的 *mapping* (or counter) 初始化：

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Counter 对象有一个字典接口，如果引用的键没有任何记录，就返回一个 0，而不是弹出一个*KeyError*：

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

设置一个计数为 0 不会从计数器中移去一个元素。使用 `del` 来删除它:

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

3.1 新版功能.

计数器对象除了字典方法以外，还提供了三个其他的方法：

elements()
返回一个迭代器，每个元素重复计数的个数。元素顺序是任意的。如果一个元素的计数小于 1，*elements()* 就会忽略它。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])
Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or `None`, *most_common()* returns *all* elements in the counter. Elements with equal counts are ordered arbitrarily:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('r', 2), ('b', 2)]
```

subtract([*iterable-or-mapping*])
从 迭代对象或 映射对象减去元素。像*dict.update()* 但是是减去，而不是替换。输入和输出都可以是 0 或者负数。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

3.2 新版功能.

通常字典方法都可用于*Counter* 对象，除了有两个方法工作方式与字典并不相同。

fromkeys(*iterable*)
这个类方法没有在*Counter* 中实现。

update([*iterable-or-mapping*])
从 迭代对象计数元素或者从另一个 映射对象 (或计数器) 添加。像*dict.update()* 但是是加上，而不是替换。另外，迭代对象应该是序列元素，而不是一个 `(key, value)` 对。

Counter 对象的常用案例

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
```

(下页继续)

(续上页)

```
dict(c)                          # convert to a regular dictionary
c.items()                        # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))     # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]        # n least common elements
+c                               # remove zero and negative counts
```

提供了几个数学操作，可以结合*Counter*对象，以生产 multisets (计数器中大于 0 的元素)。加和减，结合计数器，通过加上或者减去元素的相应计数。交集和并集返回相应计数的最小或最大值。每种操作都可以接受带符号的计数，但是输出会忽略掉结果为零或者小于零的计数。

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                        # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                        # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                        # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                        # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

单目加和减（一元操作符）意思是从空计数器加或者减去。

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

3.3 新版功能: 添加了对一元加，一元减和位置集合操作的支持。

注解： 计数器主要是为了表达运行的正的计数而设计；但是，小心不要预先排除负数或者其他类型。为了帮助这些用例，这一节记录了最小范围和类型限制。

- *Counter* 类是一个字典的子类，不限制键和值。值用于表示计数，但你实际上 可以存储任何其他值。
- The `most_common()` method requires only that the values be orderable.
- For in-place operations such as `c[key] += 1`, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for `update()` and `subtract()` which allow negative and zero values for both inputs and outputs.
- Multiset 多集合方法只为正值的使用情况设计。输入可以是负数或者 0，但只输出计数为正的值。没有类型限制，但值类型需要支持加，减和比较操作。
- The `elements()` method requires integer counts. It ignores zero and negative counts.

参见：

- Bag class 在 Smalltalk。
- Wikipedia 链接 Multisets.
- C++ multisets 教程和例子。
- 数学操作和多集合用例，参考 *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19* 。

- To enumerate all distinct multisets of a given size over a given set of elements, see *itertools.combinations_with_replacement()*:

  map(Counter, combinations_with_replacement(‘ABC’, 2)) –> AA AB AC BB BC CC

8.3.3 deque 对象

class collections.**deque**([*iterable*[, *maxlen*]])

返回一个新的双向队列对象，从左到右初始化 (用方法*append()*)，从 *iterable* （迭代对象) 数据创建。如果 *iterable* 没有指定，新队列为空。

Deque 队列是由栈或者 queue 队列生成的（发音是“deck”,” double-ended queue”的简称)。Deque 支持线程安全，内存高效添加 (append) 和弹出 (pop)，从两端都可以，两个方向的大概开销都是 O(1) 复杂度。

虽然*list* 对象也支持类似操作，不过这里优化了定长操作和 `pop(0)` 和 `insert(0, v)` 的开销。它们引起 O(n) 内存移动的操作，改变底层数据表达的大小和位置。

如果 *maxlen* 没有指定或者是 `None` ，deques 可以增长到任意长度。否则，deque 就限定到指定最大长度。一旦限定长度的 deque 满了，当新项加入时，同样数量的项就从另一端弹出。限定长度 deque 提供类似 Unix filter `tail` 的功能。它们同样可以用与追踪最近的交换和其他数据池活动。

双向队列 (deque) 对象支持以下方法：

append(*x*)
添加 *x* 到右端。

appendleft(*x*)
添加 *x* 到左端。

clear()
移除所有元素，使其长度为 0.

copy()
创建一份浅拷贝。

3.5 新版功能.

count(*x*)
计算 deque 中元素等于 *x* 的个数。

3.2 新版功能.

extend(*iterable*)
扩展 deque 的右侧，通过添加 iterable 参数中的元素。

extendleft(*iterable*)
扩展 deque 的左侧，通过添加 iterable 参数中的元素。注意，左添加时，在结果中 iterable 参数中的顺序将被反过来添加。

index(*x*[, *start*[, *stop*]])
返回 *x* 在 deque 中的位置（在索引 *start* 之后，索引 *stop* 之前)。返回第一个匹配项，如果未找到则引发*ValueError*。

3.5 新版功能.

insert(*i*, *x*)
在位置 *i* 插入 *x* 。

如果插入会导致一个限长 deque 超出长度 *maxlen* 的话，就引发一个*IndexError*。

3.5 新版功能.

pop()
移去并且返回一个元素，deque 最右侧的那一个。如果没有元素的话，就引发一个*IndexError*。

popleft()
移去并且返回一个元素，deque 最左侧的那一个。如果没有元素的话，就引发*IndexError*。

remove(*value*)
移除找到的第一个 *value*。如果没有的话就引发*ValueError*。

reverse()
将 deque 逆序排列。返回 None 。

3.2 新版功能.

rotate(*n=1*)
向右循环移动 *n* 步。如果 *n* 是负数，就向左循环。

如果 deque 不是空的，向右循环移动一步就等价于 d.appendleft(d.pop()) ，向左循环一步就等价于 d.append(d.popleft()) 。

Deque 对象同样提供了一个只读属性:

maxlen
Deque 的最大尺寸，如果没有限定的话就是 None 。

3.1 新版功能.

除了以上，deque 还支持迭代，清洗，len(d), reversed(d), copy.copy(d), copy.deepcopy(d)，成员测试 in 操作符，和下标引用 d[-1] 。索引存取在两端的复杂度是 O(1)，在中间的复杂度比 O(n) 略低。要快速存取，使用 list 来替代。

Deque 从版本 3.5 开始支持 __add__(), __mul__(), 和 __imul__() 。

示例:

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
'i'

>>> list(reversed(d))                # list the contents of a deque in reverse
['i', 'h', 'g']
```

(下页继续)

(续上页)

```
>>> 'h' in d                            # search the deque
True
>>> d.extend('jkl')                     # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                         # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                        # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))                  # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                           # empty the deque
>>> d.pop()                             # cannot pop from an empty deque
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')                 # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

deque 用法

这一节展示了 deque 的多种用法。

限长 deque 提供了类似 Unix `tail` 过滤功能

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

另一个用法是维护一个近期添加元素的序列，通过从右边添加和从左边弹出

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

The `rotate()` method provides a way to implement *deque* slicing and deletion. For example, a pure Python implementation of `del d[n]` relies on the `rotate()` method to position elements to be popped:

```
def delete_nth(d, n):
    d.rotate(-n)
```

(下页继续)

(续上页)

```
    d.popleft()
    d.rotate(n)
```

To implement *deque* slicing, use a similar approach applying `rotate()` to bring a target element to the left side of the deque. Remove old entries with `popleft()`, add new entries with `extend()`, and then reverse the rotation. With minor variations on that approach, it is easy to implement Forth style stack manipulations such as `dup`, `drop`, `swap`, `over`, `pick`, `rot`, and `roll`.

8.3.4 defaultdict 对象

class collections.**defaultdict**([*default_factory*[, ...]])

返回一个新的类似字典的对象。*defaultdict* 是内置*dict* 类的子类。它重载了一个方法并添加了一个可写的实例变量。其余的功能与*dict* 类相同，此处不再重复说明。

第一个参数*default_factory* 提供了一个初始值。它默认为 None 。所有的其他参数都等同与*dict* 构建器中的参数对待，包括关键词参数。

defaultdict 对象除了支持*dict* 的操作，还支持下面的方法作为扩展:

__missing__(*key*)

如果*default_factory* 属性为 None，则调用本方法会抛出*KeyError* 异常，附带参数 *key*。

如果*default_factory* 不为 None ，它就会会被调用，不带参数，为 *key* 提供一个默认值，这个值和 *key* 作为一个对被插入到字典中，并返回。

如果调用*default_factory* 时抛出了异常，这个异常会原封不动地向外层传递。

在无法找到所需键值时，本方法会被*dict* 中的 __getitem__() 方法调用。无论本方法返回了值还是抛出了异常，都会被 __getitem__() 传递。

注意*__missing__()* 不会被 __getitem__() 以外的其他方法调用。意思就是 get() 会向正常的 dict 那样返回 None ，而不是使用*default_factory* 。

defaultdict 支持以下实例变量:

default_factory

这个属性被*__missing__()* 方法使用；它从构建器的第一个参数初始化，如果提供了的话，否则就是 None 。

defaultdict 例子

Using *list* as the `default_factory`, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

When each key is encountered for the first time, it is not already in the mapping; so an entry is automatically created using the `default_factory` function which returns an empty *list*. The `list.append()` operation then attaches the value to the new list. When keys are encountered again, the look-up proceeds normally (returning the list for that key) and the `list.append()` operation adds another value to the list. This technique is simpler and faster than an equivalent technique using *dict.setdefault()*:

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Setting the `default_factory` to *int* makes the *defaultdict* useful for counting (like a bag or multiset in other languages):

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

When a letter is first encountered, it is missing from the mapping, so the `default_factory` function calls *int()* to supply a default count of zero. The increment operation then builds up the count for each letter.

函数*int()* 总是返回 0，是常数函数的特殊情况。一个更快和灵活的方法是使用 lambda 函数，可以提供任何常量值 (不只是 0):

```
>>> def constant_factory(value):
...     return lambda: value
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

Setting the `default_factory` to *set* makes the *defaultdict* useful for building a dictionary of sets:

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.3.5 namedtuple() 命名元组的工厂函数

命名元组赋予每个位置一个含义，提供可读性和自文档性。它们可以用于任何普通元组，并添加了通过名字获取值的能力，通过索引值也是可以的。

`collections.`**`namedtuple`**(*typename*, *field_names*, *, *verbose=False*, *rename=False*, *module=None*)

返回一个新的元组子类，名为 *typename* 。这个新的子类用于创建类元组的对象，可以通过域名来获取属性值，同样也可以通过索引和迭代获取值。子类实例同样有文档字符串（类名和域名）另外一个有用的 `__repr__()` 方法，以 `name=value` 格式列明了元组内容。

field_names 是一个像 `['x', 'y']` 一样的字符串序列。另外 *field_names* 可以是一个纯字符串，用空白或逗号分隔开元素名，比如 `'x y'` 或者 `'x, y'` 。

任何有效的 Python 标识符都可以作为域名，除了下划线开头的那些。有效标识符由字母，数字，下划线组成，但首字母不能是数字或下划线，另外不能是关键词*keyword* 比如 *class*, *for*, *return*, *global*, *pass*, 或 *raise* 。

如果 *rename* 为真，无效域名会自动转换成位置名。比如 `['abc', 'def', 'ghi', 'abc']` 转换成 `['abc', '_1', 'ghi', '_3']`，消除关键词 `def` 和重复域名 `abc`。

If *verbose* is true, the class definition is printed after it is built. This option is outdated; instead, it is simpler to print the `_source` attribute.

如果 *module* 值有定义，命名元组的 `__module__` 属性值就被设置。

命名元组实例没有字典，所以它们要更轻量，并且占用更小内存。

在 3.1 版更改: 添加了对 *rename* 的支持。

在 3.6 版更改: *verbose* 和 *rename* 参数成为*仅限关键字参数*.

在 3.6 版更改: 添加了 *module* 参数。

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

命名元组尤其有用于赋值*csv* *sqlite3* 模块返回的元组

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade
↪')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

除了继承元组的方法，命名元组还支持三个额外的方法和两个属性。为了防止域名冲突，方法和属性以下划线开始。

classmethod somenamedtuple.**_make**(*iterable*)
类方法从存在的序列或迭代实例创建一个新实例。

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple.**_asdict**()
Return a new *OrderedDict* which maps field names to their corresponding values:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
OrderedDict([('x', 11), ('y', 22)])
```

在 3.1 版更改: 返回一个*OrderedDict* 而不是*dict* 。

somenamedtuple.**_replace**(***kwargs*)
返回一个新的命名元组实例，并将指定域替换为新的值

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum],␣
↪timestamp=time.now())
```

somenamedtuple.**_source**
A string with the pure Python source code used to create the named tuple class. The source makes the named tuple self-documenting. It can be printed, executed using *exec()*, or saved to a file and imported.

3.3 新版功能.

somenamedtuple.**_fields**
字符串元组列出了域名。用于提醒和从现有元组创建一个新的命名元组类型。

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

要获取这个名字域的值，使用*getattr()* 函数:

```
>>> getattr(p, 'x')
11
```

To convert a dictionary to a named tuple, use the double-star-operator (as described in tut-unpacking-arguments):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

因为一个命名元组是一个正常的 Python 类，它可以很容易的通过子类更改功能。这里是如何添加一个计算域和定宽输出打印格式:

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.
↪hypot)

>>> for p in Point(3, 4), Point(14, 5/7):
```

(下页继续)

(续上页)

```
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

上面的子类设置 __slots__ 为一个空元组。通过阻止创建实例字典保持了较低的内存开销。

Subclassing is not useful for adding new, stored fields. Instead, simply create a new named tuple type from the _fields attribute:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

文档字符串可以自定义，通过直接赋值给 __doc__ 属性:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

在 3.5 版更改: 文档字符串属性变成可写。

Default values can be implemented by using _replace() to customize a prototype instance:

```
>>> Account = namedtuple('Account', 'owner balance transaction_count')
>>> default_account = Account('<owner name>', 0.0, 0)
>>> johns_account = default_account._replace(owner='John')
>>> janes_account = default_account._replace(owner='Jane')
```

参见:

- Recipe for named tuple abstract base class with a metaclass mix-in by Jan Kaliszewski. Besides providing an *abstract base class* for named tuples, it also supports an alternate *metaclass*-based constructor that is convenient for use cases where named tuples are being subclassed.
- 对于以字典为底层的可变域名，参考*types.SimpleNamespace()* 。
- See *typing.NamedTuple()* for a way to add type hints for named tuples.

8.3.6 OrderedDict 对象

Ordered dictionaries are just like regular dictionaries but they remember the order that items were inserted. When iterating over an ordered dictionary, the items are returned in the order their keys were first added.

class collections.**OrderedDict**([*items*])

Return an instance of a dict subclass, supporting the usual *dict* methods. An *OrderedDict* is a dict that remembers the order that keys were first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

3.1 新版功能.

popitem(*last=True*)

有序字典的*popitem()* 方法移除并返回一个 (key, value) 键值对。如果 *last* 值为真，则按 LIFO (last-in, first-out) 后进先出的顺序返回键值对，否则就按 FIFO (first-in, first-out) 先进先出的顺序返回键值对。

move_to_end(*key*, *last=True*)

将现有 *key* 移动到有序字典的任一端。如果 *last* 为真值（默认）则将元素移至末尾；如果 *last* 为假值则将元素移至开头。如果 *key* 不存在则会触发*KeyError*:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

3.2 新版功能.

相对于通常的映射方法，有序字典还另外提供了逆序迭代的支持，通过*reversed()*。

OrderedDict 之间的相等测试是顺序敏感的，实现为 `list(od1.items())==list(od2.items())`。*OrderedDict* 对象和其他的*Mapping* 的相等测试，是顺序敏感的字典测试。这允许*OrderedDict* 替换为任何字典可以使用的场所。

在 3.5 版更改: *OrderedDict* 的项 (item)，键 (key) 和值 (value) 视图 现在支持逆序迭代，通过*reversed()*。

在 3.6 版更改: **PEP 468** 赞成将关键词参数的顺序保留，通过传递给*OrderedDict* 构造器和它的 `update()` 方法。

OrderedDict 例子和用法

Since an ordered dictionary remembers its insertion order, it can be used in conjunction with sorting to make a sorted dictionary:

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple': 4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

The new sorted dictionaries maintain their sort order when entries are deleted. But when new keys are added, the keys are appended to the end and the sort is not maintained.

It is also straight-forward to create an ordered dictionary variant that remembers the order the keys were *last* inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        if key in self:
            del self[key]
        OrderedDict.__setitem__(self, key, value)
```

An ordered dictionary can be combined with the *Counter* class so that the counter remembers the order elements are first encountered:

```
class OrderedCounter(Counter, OrderedDict):
    'Counter that remembers the order elements are first encountered'

    def __repr__(self):
        return '%s(%r)' % (self.__class__.__name__, OrderedDict(self))

    def __reduce__(self):
        return self.__class__, (OrderedDict(self),)
```

8.3.7 UserDict 对象

UserDict 类是用作字典对象的外包装。对这个类的需求已部分由直接创建*dict* 的子类的功能所替代；不过，这个类处理起来更容易，因为底层的字典可以作为属性来访问。

class collections.**UserDict**([*initialdata*])

模拟一个字典类。这个实例的内容保存为一个正常字典，可以通过*UserDict* 实例的*data* 属性存取。如果提供了 *initialdata* 值，*data* 就被初始化为它的内容；注意一个 *initialdata* 的引用不会被保留作为其他用途。

UserDict 实例提供了以下属性作为扩展方法和操作的支持:

data

一个真实的字典，用于保存*UserDict* 类的内容。

8.3.8 UserList 对象

这个类封装了列表对象。它是一个有用的基础类，对于你想自定义的类似列表的类，可以继承和覆盖现有的方法，也可以添加新的方法。这样我们可以对列表添加新的行为。

对这个类的需求已部分由直接创建*list* 的子类的功能所替代；不过，这个类处理起来更容易，因为底层的列表可以作为属性来访问。

class collections.**UserList**([*list*])

模拟一个列表。这个实例的内容被保存为一个正常列表，通过*UserList* 的*data* 属性存取。实例内容被初始化为一个 *list* 的 copy，默认为 `[]` 空列表。*list* 可以是迭代对象，比如一个 Python 列表，或者一个*UserList* 对象。

UserList 提供了以下属性作为可变序列的方法和操作的扩展:

data

一个*list* 对象用于存储*UserList* 的内容。

子类化的要求: *UserList* 的子类需要提供一个构造器，可以无参数调用，或者一个参数调用。返回一个新序列的列表操作需要创建一个实现类的实例。它假定了构造器可以以一个参数进行调用，这个参数是一个序列对象，作为数据源。

如果一个分离的类不希望依照这个需求，所有的特殊方法就必须重写；请参照源代码进行修改。

8.3.9 UserString 对象

UserString 类是用作字符串对象的外包装。对这个类的需求已部分由直接创建*str* 的子类的功能所替代；不过，这个类处理起来更容易，因为底层的字符串可以作为属性来访问。

class collections.**UserString**([*sequence*])

Class that simulates a string or a Unicode string object. The instance's content is kept in a regular string object, which is accessible via the data attribute of *UserString* instances. The instance's contents are initially set to a copy of *sequence*. The *sequence* can be an instance of *bytes*, *str*, *UserString* (or a subclass) or an arbitrary sequence which can be converted into a string using the built-in *str()* function.

在 3.5 版更改: 新方法 __getnewargs__, __rmod__, casefold, format_map, isprintable, 和 maketrans。

8.4 collections.abc —容器的抽象基类

3.3 新版功能: 该模块曾是*collections* 模块的组成部分。

源代码： Lib/_collections_abc.py

该模块定义了一些抽象基类，它们可用于判断一个具体类是否具有某一特定的接口；例如，这个类是否可哈希，或其是否为映射类。

8.4.1 容器抽象基类

这个容器模块提供了以下*ABCs*:

抽象基类	继承自	抽象方法	Mixin 方法
Container		__contains__	
Hashable		__hash__	
Iterable		__iter__	
Iterator	*Iterable*	__next__	__iter__
Reversible	*Iterable*	__reversed__	
Generator	*Iterator*	send, throw	close, __iter__, __next__
Sized		__len__	
Callable		__call__	
Collection	*Sized*, *Iterable*, *Container*	__contains__, __iter__, __len__	
Sequence	*Reversible*, *Collection*	__getitem__, __len__	__contains__, __iter__, __reversed__, index, and count
MutableSequence	*Sequence*	__getitem__, __setitem__, __delitem__, __len__, insert	继承自*Sequence* 的方法，以及 append, reverse, extend, pop, remove，和 __iadd__
ByteString	*Sequence*	__getitem__, __len__	继承自*Sequence* 的方法
Set	*Collection*	__contains__, __iter__, __len__	__le__, __lt__, __eq__, __ne__, __gt__, __ge__, __and__, __or__, __sub__, __xor__, and isdisjoint
MutableSet	*Set*	__contains__, __iter__, __len__, add, discard	继承自*Set* 的方法以及 clear, pop, remove, __ior__, __iand__, __ixor__，和 __isub__
Mapping	*Collection*	__getitem__, __iter__, __len__	__contains__, keys, items, values, get, __eq__, and __ne__
MutableMapping	*Mapping*	__getitem__, __setitem__, __delitem__, __iter__, __len__	继承自*Mapping* 的方法以及 pop, popitem, clear, update，和 setdefault
MappingView	*Sized*		__len__
ItemsView	*MappingView*, *Set*		__contains__, __iter__
KeysView	*MappingView*, *Set*		__contains__, __iter__
ValuesView	*MappingView*		__contains__, __iter__
Awaitable		__await__	
Coroutine	*Awaitable*	send, throw	close
AsyncIterable		__aiter__	
AsyncIterator	*AsyncIterable*	__anext__	__aiter__
AsyncGenerator	*AsyncIterator*	asend, athrow	aclose, __aiter__, __anext__

class collections.abc.**Container**
class collections.abc.**Hashable**
class collections.abc.**Sized**
class collections.abc.**Callable**
分别提供了 __contains__(), __hash__(), __len__() 和 __call__() 方法的抽象基类。

class collections.abc.**Iterable**
提供了 __iter__() 方法的抽象基类。

使用 isinstance(obj, Iterable) 可以检测一个类是否已经注册到了*Iterable* 或者实现了 __iter__() 函数，但是无法检测这个类是否能够使用 __getitem__() 方法进行迭代。检测一个对象是否是*iterable* 的唯一可信赖的方法是调用 iter(obj)。

class collections.abc.**Collection**
集合了 Sized 和 Iterable 类的抽象基类。

3.6 新版功能.

class collections.abc.**Iterator**
提供了*__iter__()* 和*__next__()* 方法的抽象基类。参见*iterator* 的定义。

class collections.abc.**Reversible**
为可迭代类提供了 __reversed__() 方法的抽象基类。

3.6 新版功能.

class collections.abc.**Generator**
生成器类，实现了 **PEP 342** 中定义的协议，继承并扩展了迭代器，提供了 send(), throw() 和 close() 方法。参见*generator* 的定义。

3.5 新版功能.

class collections.abc.**Sequence**
class collections.abc.**MutableSequence**
class collections.abc.**ByteString**
只读且可变的序列*sequences* 的抽象基类。

实现笔记：一些混入（Maxin）方法比如 __iter__(), __reversed__() 和 index() 会重复调用底层的 __getitem__() 方法。因此，如果实现的 __getitem__() 是常数级访问速度，那么相应的混入方法会有一个线性的表现；然而，如果底层方法是线性实现（例如链表），那么混入方法将会是平方级的表现，这也许就需要被重构了。

在 3.5 版更改: index() 方法支持 *stop* 和 *start* 参数。

class collections.abc.**Set**
class collections.abc.**MutableSet**
只读且可变的集合的抽象基类。

class collections.abc.**Mapping**
class collections.abc.**MutableMapping**
只读且可变的映射*mappings* 的抽象基类。

class collections.abc.**MappingView**
class collections.abc.**ItemsView**
class collections.abc.**KeysView**
class collections.abc.**ValuesView**
映射及其键和值的视图*views* 的抽象基类。

class collections.abc.**Awaitable**
为可等待对象*awaitable* 提供的类，可以被用于 await 表达式中。习惯上必须实现 __await__() 方法。

协程对象*Coroutine* 和*Coroutine* 抽象基类的实例都是这个抽象基类的实例。

注解： 在 CPython 里，基于生成器的协程（使用*types.coroutine()* 或*asyncio.coroutine()* 包装的生成器）都是 可等待对象，即使他们不含有 __await__() 方法。使用 isinstance(gencoro, Awaitable) 来检测他们会返回 False。要使用*inspect.isawaitable()* 来检测他们。

3.5 新版功能.

class collections.abc.**Coroutine**
用于协程兼容类的抽象基类。实现了如下定义在 coroutine-objects: 里的方法：send()，throw() 和 close()。通常的实现里还需要实现 __await__() 方法。所有的*Coroutine* 实例都必须是*Awaitable* 实例。参见*coroutine* 的定义。

注解： 在 CPython 里，基于生成器的协程（使用*types.coroutine()* 或*asyncio.coroutine()* 包装的生成器）都是 可等待对象，即使他们不含有 __await__() 方法。使用 isinstance(gencoro, Coroutine) 来检测他们会返回 False。要使用*inspect.isawaitable()* 来检测他们。

3.5 新版功能.

class collections.abc.**AsyncIterable**
提供了 __aiter__ 方法的抽象基类。参见*asynchronous iterable* 的定义。

3.5 新版功能.

class collections.abc.**AsyncIterator**
提供了 __aiter__ 和 __anext__ 方法的抽象基类。参见*asynchronous iterator* 的定义。

3.5 新版功能.

class collections.abc.**AsyncGenerator**
为异步生成器类提供的抽象基类，这些类实现了定义在 **PEP 525** 和 **PEP 492** 里的协议。

3.6 新版功能.

这些抽象基类让我们可以确定类和示例拥有某些特定的函数，例如：

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

有些抽象基类也可以用作混入类（mixin），这可以更容易地开发支持容器 API 的类。例如，要写一个支持完整*Set* API 的类，只需要提供下面这三个方法：__contains__(), __iter__() 和 __len__()。抽象基类会补充上其余的方法，比如 __and__() 和 isdisjoint()：

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

当把*Set* 和*MutableSet* 用作混入类时需注意：

(1) 由于某些集合操作会创建新集合，默认的混入方法需要一种从可迭代对象里创建新实例的方法。假如其类构造函数签名形如 `ClassName(iterable)`，则其会调用一个内部的类方法 `_from_iterable()`，其中调用了 `cls(iterable)` 来生成一个新集合。如果这个*Set* 混入类在类中被使用，但其构造函数的签名却是不同的形式，那么你就需要重载 `_from_iterable()` 方法，将其编写成一个类方法，并且它能够从可迭代对象参数中构造一个新实例。

(2) 重载比较符时时（想必是为了速度，因为其语义都是固定的），只需要重定义 `__le__()` 和 `__ge__()` 函数，然后其他的操作会自动跟进。

(3) 混入集合类*Set* 提供了一个 `_hash()` 方法为集合计算哈希值，然而，`__hash__()` 函数却没有被定义，因为并不是所有集合都是可哈希并且不可变的。为了使用混入类为集合添加哈希能力，可以同时继承*Set()* 和*Hashable()* 类，然后定义 `__hash__ = Set._hash`。

参见：

- OrderedSet recipe 是基于*MutableSet* 构建的一个示例。
- 对于抽象基类，参见*abc* 模块和 **PEP 3119**。

8.5 heapq —堆队列算法

源码： Lib/heapq.py

这个模块提供了堆队列算法的实现，也称为优先队列算法。

堆是一个二叉树，它的每个父节点的值都只会小于或等于所有孩子节点（的值）。它使用了数组来实现：从零开始计数，对于所有的 *k*，都有 `heap[k] <= heap[2*k+1]` 和 `heap[k] <= heap[2*k+2]`。为了便于比较，不存在的元素被认为是无限大。堆最有趣的特性在于最小的元素总是在根结点：`heap[0]`。

这个 API 与教材中堆算法的实现不太一样，在于两方面：(a) 我们使用了基于零开始的索引。这使得节点和其孩子节点之间的索引关系不太直观，但是由于 Python 使用了从零开始的索引，所以这样做更加合适。(b) 我们的 pop 方法返回了最小的元素，而不是最大的（这在教材中叫做“最小堆”；而“最大堆”在课本中更加常见，因为它更加适用于原地排序）。

基于这两方面，把堆看作原生的 Python list 也没什么奇怪的：`heap[0]` 表示最小的元素，同时 `heap.sort()` 维护了堆的不变性！

要创建一个堆，可以使用 list 来初始化为 `[]`，或者你可以通过一个函数*heapify()*，来把一个 list 转换成堆。

定义了以下函数：

`heapq.`**`heappush`**(*heap*, *item*)
将 *item* 的值加入 *heap* 中，保持堆的不变性。

`heapq.`**`heappop`**(*heap*)
弹出并返回 *heap* 的最小的元素，保持堆的不变性。如果堆为空，抛出*IndexError*。使用 `heap[0]`，可以只访问最小的元素而不弹出它。

`heapq.`**`heappushpop`**(*heap*, *item*)
将 *item* 放入堆中，然后弹出并返回 *heap* 的最小元素。该组合操作比先调用*heappush()* 再调用*heappop()* 运行起来更有效率。

`heapq.`**`heapify`**(*x*)
将 list *x* 转换成堆，原地，线性时间内。

heapq.**heapreplace**(*heap*, *item*)
弹出并返回 *heap* 中最小的一项，同时推入新的 *item*。堆的大小不变。如果堆为空则引发*IndexError*。

这个单步骤操作比*heappop()* 加*heappush()* 更高效，并且在使用固定大小的堆时更为适宜。pop/push 组合总是会从堆中返回一个元素并将其替换为 *item*。

返回的值可能会比添加的 *item* 更大。如果不希望如此，可考虑改用*heappushpop()*。它的 push/pop 组合会返回两个值中较小的一个，将较大的值留在堆中。

该模块还提供了三个基于堆的通用功能函数。

heapq.**merge**(**iterables*, *key=None*, *reverse=False*)
将多个已排序的输入合并为一个已排序的输出（例如，合并来自多个日志文件的带时间戳的条目）。返回已排序值的*iterator*。

类似于 `sorted(itertools.chain(*iterables))` 但返回一个可迭代对象，不会一次性地将数据全部放入内存，并假定每个输入流都是已排序的（从小到大）。

具有两个可选参数，它们都必须指定为关键字参数。

key 指定带有单个参数的*key function*，用于从每个输入元素中提取比较键。默认值为 `None` (直接比较元素)。

reverse is a boolean value. If set to `True`, then the input elements are merged as if each comparison were reversed.

在 3.5 版更改: 添加了可选的 *key* 和 *reverse* 形参。

heapq.**nlargest**(*n*, *iterable*, *key=None*)
从 *iterable* 所定义的数据集中返回前 *n* 个最大的元素。如果提供了 *key* 则其应指定一个单参数的函数，用于从 that is used to extract a comparison key from each element in *iterable* 的每个元素中提取比较键 (例如 `key=str.lower`)。等价于: `sorted(iterable, key=key, reverse=True)[:n]`。

heapq.**nsmallest**(*n*, *iterable*, *key=None*)
从 *iterable* 所定义的数据集中返回前 *n* 个最小元素组成的列表。如果提供了 *key* 则其应指定一个单参数的函数，用于从 *iterable* 的每个元素中提取比较键 (例如 `key=str.lower`)。等价于: `sorted(iterable, key=key)[:n]`。

后两个函数在 *n* 值较小时性能最好。对于更大的值，使用*sorted()* 函数会更有效率。此外，当 `n==1` 时，使用内置的*min()* 和*max()* 函数会更有效率。如果需要重复使用这些函数，请考虑将可迭代对象转为真正的堆。

8.5.1 基本示例

堆排序 可以通过将所有值推入堆中然后每次弹出一个最小值项来实现。

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

这类似于 `sorted(iterable)`，但与*sorted()* 不同的是这个实现是不稳定的。

堆元素可以为元组。这适用于将比较值（例如任务优先级）与跟踪的主记录进行赋值的场合:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.5.2 优先队列实现说明

优先队列 是堆的常用场合，并且它的实现包含了多个挑战：

- 排序稳定性：你该如何令相同优先级的两个任务按它们最初被加入时的顺序返回？
- 如果优先级相同且任务没有默认比较顺序，则 (priority, task) 对的元组比较将会中断。
- 如果任务优先级发生改变，你该如何将其移至堆中的新位置？
- 或者如果一个挂起的任务需要被删除，你该如何找到它并将其移出队列？

针对前两项挑战的一种解决方案是将条目保存为包含优先级、条目计数和任务对象 3 个元素的列表。条目计数可用来打破平局，这样具有相同优先级的任务将按它们的添加顺序返回。并且由于没有哪两个条目计数是相同的，元组比较将永远不会直接比较两个任务。

其余的挑战主要包括找到挂起的任务并修改其优先级或将其完全移除。找到一个任务可使用一个指向队列中条目的字典来实现。

移除条目或改变其优先级的操作实现起来更为困难，因为它会破坏堆结构不变量。因此，一种可能的解决方案是将条目标记为已移除，再添加一个改变了优先级的新条目：

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

8.5.3 理论

堆是通过数组来实现的，其中的元素从 0 开始计数，对于所有的 *k* 都有 `a[k] <= a[2*k+1]` 且 `a[k] <= a[2*k+2]`。为了便于比较，不存在的元素被视为无穷大。堆最有趣的特性在于 `a[0]` 总是其中最小的元素。

上面的特殊不变量是用来作为一场锦标赛的高效内存表示。下面的数字是 *k* 而不是 `a[k]`:

```
                                0

             1                                  2

      3              4                   5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

在上面的树中，每个 *k* 单元都位于 `2*k+1` 和 `2*k+2` 之上。体育运动中我们经常见到二元锦标赛模式，每个胜者单元都位于另两个单元之上，并且我们可以沿着树形图向下追溯胜者所遇到的所有对手。但是，在许多采用这种锦标赛模式的计算机应用程序中，我们并不需要追溯胜者的历史。为了获得更高的内存利用效率，当一个胜者晋级时，我们会用较低层级的另一条目来替代它，因此规则变为一个单元和它之下的两个单元包含三个不同条目，上方单元“胜过”了两个下方单元。

如果此堆的不变量始终受到保护，则序号 0 显然是最后的赢家。删除它并找出“下一个”赢家的最简单算法方式是家某个输家（让我们假定是上图中的 30 号单元）移至 0 号位置，然后将这个新的 0 号沿树下行，不断进行值的交换，直到不变量重新建立。这显然会是树中条目总数的对数。通过迭代所有条目，你将得到一个 O(n log n) 复杂度的排序。

此排序有一个很好的特性就是你可以在排序进行期间高效地插入新条目，前提是插入的条目不比你最近取出的 0 号元素“更好”。这在模拟上下文时特别有用，在这种情况下树保存的是所有传入事件，“胜出”条件是最小调度时间。当一个事件将其他事件排入执行计划时，它们的调试时间向未来方向延长，这样它们可方便地入堆。因此，堆结构很适宜用来实现调度器，我的 MIDI 音序器就是用的这个:-)。

用于实现调度器的各种结构都得到了充分的研究，堆是非常适宜的一种，因为它们的速度相当快，并且几乎是恒定的，最坏的情况与平均情况没有太大差别。虽然还存在其他总体而言更高效的实现方式，但其最坏的情况却可能非常糟糕。

堆在大磁盘排序中也非常有用。你应该已经了解大规模排序会有多个“运行轮次”（即预排序的序列，其大小通常与 CPU 内存容量相关），随后这些轮次会进入合并通道，轮次合并的组织往往非常巧妙[1]。非常重要的一点是初始排序应产生尽可能长的运行轮次。锦标赛模式是达成此目标的好办法。如果你使用全部有用内存来进行锦标赛，替换和安排恰好适合当前运行轮次的条目，你将可以对于随机输入生成两倍于内存大小的运行轮次，对于模糊排序的输入还会有更好的效果。

另外，如果你输出磁盘上的第 0 个条目并获得一个可能不适合当前锦标赛的输入（因为其值要“胜过”上一个输出值），它无法被放入堆中，因此堆的尺寸将缩小。被释放的内存可以被巧妙地立即重用以逐步构建第二个堆，其增长速度与第一个堆的缩减速度正好相同。当第一个堆完全消失时，你可以切换新堆并启动新的运行轮次。这样做既聪明又高效!

总之，堆是值得了解的有用内存结构。我在一些应用中用到了它们，并且认为保留一个‘heap’模块是很有意义的。:-)

[1] 当前时代的磁盘平衡算法与其说是巧妙，不如说是麻烦，这是由磁盘的寻址能力导致的结果。在无法寻址的设备例如大型磁带机上，情况则相当不同，开发者必须非常聪明地（极为提前地）确保每次磁带转动都尽可能地高效（就是说能够最好地加入到合并“进程”中）。有些磁带甚至能够反向读取，这也被用来避免倒带的耗时。请相信我，真正优秀的磁带机排序看起来是极其壮观的，排序一向都是一门伟大的艺术！:-)

备注

8.6 bisect —数组二分查找算法

源代码： Lib/bisect.py

这个模块对有序列表提供了支持，使得他们可以在插入新数据仍然保持有序。对于长列表，如果其包含元素的比较操作十分昂贵的话，这可以是对更常见方法的改进。这个模块叫做*bisect*因为其使用了基本的二分（bisection）算法。源代码也可以作为很棒的算法示例（边界判断也做好啦!）

定义了以下函数：

bisect.**bisect_left**(*a*, *x*, *lo=0*, *hi=len(a)*)
: 在 *a* 中找到 *x* 合适的插入点以维持有序。参数 *lo* 和 *hi* 可以被用于确定需要考虑的子集；默认情况下整个列表都会被使用。如果 *x* 已经在 *a* 里存在，那么插入点会在已存在元素之前（也就是左边）。如果 *a* 是列表（list）的话，返回值是可以被放在 `list.insert()` 的第一个参数的。

  返回的插入点 *i* 可以将数组 *a* 分成两部分。左侧是 `all(val < x for val in a[lo:i])`，右侧是 `all(val >= x for val in a[i:hi])`。

bisect.**bisect_right**(*a*, *x*, *lo=0*, *hi=len(a)*)
bisect.**bisect**(*a*, *x*, *lo=0*, *hi=len(a)*)
: 类似于*bisect_left()*，但是返回的插入点是 *a* 中已存在元素 *x* 的右侧。

  返回的插入点 *i* 可以将数组 *a* 分成两部分。左侧是 `all(val <= x for val in a[lo:i])`，右侧是 `all(val > x for val in a[i:hi])` for the right side。

bisect.**insort_left**(*a*, *x*, *lo=0*, *hi=len(a)*)
: 将 *x* 插入到一个有序序列 *a* 里，并维持其有序。如果 *a* 有序的话，这相当于 `a.insert(bisect.bisect_left(a, x, lo, hi), x)`。要注意搜索是 O(log n) 的，插入却是 O(n) 的。

bisect.**insort_right**(*a*, *x*, *lo=0*, *hi=len(a)*)
bisect.**insort**(*a*, *x*, *lo=0*, *hi=len(a)*)
: 类似于*insort_left()*，但是把 *x* 插入到 *a* 中已存在元素 *x* 的右侧。

参见：

SortedCollection recipe 使用 bisect 构造了一个功能完整的集合类，提供了直接的搜索方法和对用于搜索的 key 方法的支持。所有用于搜索的键都是预先计算的，以避免在搜索时对 key 方法的不必要调用。

8.6.1 搜索有序列表

上面的*bisect()*函数对于找到插入点是有用的，但在一般的搜索任务中可能会有点尴尬。下面 5 个函数展示了如何将其转变成有序列表中的标准查找函数

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
```

(下页继续)

(续上页)

```
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.6.2 其他示例

函数*bisect()*还可以用于数字表查询。这个例子是使用*bisect()*从一个给定的考试成绩集合里，通过一个有序数字表，查出其对应的字母等级：90 分及以上是‘A’，80 到 89 是‘B’，以此类推

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

与*sorted()*函数不同，对于*bisect()*函数来说，*key* 或者 *reversed* 参数并没有什么意义。因为这会导致设计效率低下（连续调用 bisect 函数时，是不会“记住”过去查找过的键的）。

正相反，最好去搜索预先计算好的键列表，来查找相关记录的索引。

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]         # precomputed list of keys
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.7 array —高效的数值数组

此模块定义了一种对象类型，可以紧凑地表示基本类型值的数组：字符、整数、浮点数等。数组属于序列类型，其行为与列表非常相似，不同之处在于其中存储的对象类型是受限的。类型在对象创建时使用单个字符的 *类型码* 来指定。已定义的类型码如下：

类型码	C 类型	Python 类型	以字节表示的最小尺寸	注释
'b'	signed char	int	1	
'B'	unsigned char	int	1	
'u'	Py_UNICODE	Unicode 字符	2	(1)
'h'	signed short	int	2	
'H'	unsigned short	int	2	
'i'	signed int	int	2	
'I'	unsigned int	int	2	
'l'	signed long	int	4	
'L'	unsigned long	int	4	
'q'	signed long long	int	8	(2)
'Q'	unsigned long long	int	8	(2)
'f'	float	float	4	
'd'	double	float	8	

注释:

(1) `'u'` 类型码对应于 Python 中已过时的 unicode 字符 (`Py_UNICODE` 即 `wchar_t`)。根据系统平台的不同，它可能是 16 位或 32 位。

`'u'` 将与其它的 `Py_UNICODE` API 一起被移除。

Deprecated since version 3.3, will be removed in version 4.0.

(2) The `'q'` and `'Q'` type codes are available only if the platform C compiler used to build Python supports C `long long`, or, on Windows, `__int64`.

3.3 新版功能.

值的实际表示会由机器的架构决定（严格地说是由 C 实现决定）。实际大小可通过 `itemsize` 属性来获取。

这个模块定义了以下类型：

class array.**array**(*typecode*[, *initializer*])

一个包含由 *typecode* 限制类型的条目的新数组，并由可选的 *initializer* 值进行初始化，该值必须为一个列表、*bytes-like object* 或包含正确类型元素的可迭代对象。

如果给定一个列表或字符串，该 initializer 会被传给新数组的 *fromlist()*，*frombytes()* 或 *fromunicode()* 方法（见下文）以将初始条目添加到数组中。否则会将可迭代对象作为 initializer 传给 *extend()* 方法。

array.**typecodes**

一个包含所有可用类型码的字符串。

数组对象支持普通的序列操作如索引、切片、拼接和重复等。当使用切片赋值时，所赋的值必须为具有相同类型码的数组对象；所有其他情况都将引发 *TypeError*。数组对象也实现了缓冲区接口，可以用于所有支持字节类对象 的场合。

以下数据项和方法也受到支持：

array.**typecode**
用于创建数组的类型码字符。

array.**itemsize**
在内部表示中一个数组项的字节长度。

array.**append**(*x*)
添加一个值为 *x* 的新项到数组末尾。

array.**buffer_info**()
返回一个元组 (address, length) 以给出用于存放数组内容的缓冲区元素的当前内存地址和长度。以字节表示的内存缓冲区大小可通过 array.buffer_info()[1] * array.itemsize 来计算。这在使用需要内存地址的低层级（因此不够安全）I/O 接口时会很有用，例如某些 ioctl() 操作。只要数组存在并且没有应用改变长度的操作，返回数值就是有效的。

注解: 当在 C 或 C++ 编写的代码中使用数组对象时（这是有效使用此类信息的唯一方式），使用数组对象所支持的缓冲区接口更为适宜。此方法仅保留用作向下兼容，应避免在新代码中使用。缓冲区接口的文档参见 bufferobjects。

---

array.**byteswap**()
“字节对调”所有数组项。此方法只支持大小为 1, 2, 4 或 8 字节的值；对于其他值类型将引发*RuntimeError*。它适用于从不同字节序机器所生成的文件中读取数据的情况。

array.**count**(*x*)
返回 *x* 在数组中的出现次数。

array.**extend**(*iterable*)
将来自 *iterable* 的项添加到数组末尾。如果 *iterable* 是另一个数组，它必须具有 完全相同的类型码；否则将引发*TypeError*。如果 *iterable* 不是一个数组，则它必须为可迭代对象并且其元素必须为可添加到数组的适当类型。

array.**frombytes**(*s*)
添加来自字符串的项，将字符串解读为机器值的数组（相当于使用*fromfile()* 方法从文件中读取数据）。

3.2 新版功能: *fromstring()* 重命名为*frombytes()* 以使其含义更清晰。

array.**fromfile**(*f*, *n*)
从*file object* *f* 中读取 *n* 项（解读为机器值）并将它们添加到数组末尾。如果可读取数据少于 *n* 项则将引发*EOFError*，但有效的项仍然会被插入数组。*f* 必须为一个真实的内置文件对象；不支持带有 read() 方法的其它对象。

array.**fromlist**(*list*)
添加来自 list 的项。这等价于 for x in list: a.append(x)，区别在于如果发生类型错误，数组将不会被改变。

array.**fromstring**()
已弃用的*frombytes()* 的别名。

array.**fromunicode**(*s*)
使用来自给定 Unicode 字符串的数组扩展数组。数组必须是类型为 'u' 的数组；否则将引发*ValueError*。请使用 array.frombytes(unicodestring.encode(enc)) 来将 Unicode 数据添加到其他类型的数组。

array.**index**(*x*)
返回最小的 *i* 使得 *i* 为 *x* 在数组中首次出现的序号。

array.**insert**(*i*, *x*)
将值 *x* 作为新项插入数组的 *i* 位置之前。负值将被视为相对于数组末尾的位置。

array.**pop**([*i*])
从数组中移除序号为 *i* 的项并将其返回。可选参数值默认为 -1，因此默认将移除并返回末尾项。

array.**remove**(*x*)
从数组中移除首次出现的 *x*。

array.**reverse**()
反转数组中各项的顺序。

array.**tobytes**()
将数组转换为一个机器值数组并返回其字节表示（即相当与通过*tofile()* 方法写入到文件的字节序列。）

3.2 新版功能: *tostring()* 被重命名为*tobytes()* 以使其含义更清晰。

array.**tofile**(*f*)
将所有项（作为机器值）写入到*file object* *f*。

array.**tolist**()
将数组转换为包含相同项的普通列表。

array.**tostring**()
tobytes() 的已弃用别名。

array.**tounicode**()
将数组转换为一个 Unicode 字符串。数组必须是类型为 'u' 的数组；否则将引发*ValueError*。请使用 array.tobytes().decode(enc) 来从其他类型的数组生成 Unicode 字符串。

当一个数组对象被打印或转换为字符串时，它会表示为 array(typecode, initializer)。如果数组为空则 *initializer* 会被省略，否则如果 *typecode* 为 'u' 则它是一个字符串，否则它是一个数字列表。使用*eval()* 保证能将字符串转换回具有相同类型和值的数组，只要*array* 类已通过 from array import array 被引入。例如:

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

参见:

模块*struct* 打包和解包异构二进制数据。

模块*xdrlib* 打包和解包用于某些远程过程调用系统的 External Data Representation (XDR) 数据。

The Numerical Python Documentation Numeric Python 扩展 (NumPy) 定义了另一种数组类型；请访问 http://www.numpy.org/ 了解有关 Numerical Python 的更多信息。

8.8 weakref —弱引用

源码: Lib/weakref.py

---

weakref 模块允许 Python 程序员创建对象的 *weak references* 。

在下文中，术语 *referent* 表示由弱引用引用的对象。

对对象的弱引用不能保证对象存活：当对像的引用只剩弱引用时，*garbage collection* 可以销毁引用并将其内存重用于其他内容。但是，在实际销毁对象之前，即使没有强引用，弱引用也一直能返回该对象。

弱引用的主要用途是实现保存大对象的高速缓存或映射，但又不希望大对象仅仅因为它出现在高速缓存或映射中而保持存活。

例如，如果您有许多大型二进制图像对象，则可能希望将名称与每个对象关联起来。如果您使用 Python 字典将名称映射到图像，或将图像映射到名称，则图像对象将保持活动状态，因为它们在字典中显示为值或键。*weakref* 模块提供的*WeakKeyDictionary* 和*WeakValueDictionary* 类可以替代 Python 字典，使用弱引用来构造映射，这些映射不会仅仅因为它们出现在映射对象中而使对象保持存活。例如，如果一个图像对象是*WeakValueDictionary* 中的值，那么当对该图像对象的剩余引用是弱映射对象所持有的弱引用时，垃圾回收可以回收该对象并将其在弱映射对象中相应的条目删除。

WeakKeyDictionary 和*WeakValueDictionary* 在它们的实现中使用弱引用，在弱引用上设置回调函数，当键或值被垃圾回收回收时通知弱字典。*WeakSet* 实现了*set* 接口，但像*WeakKeyDictionary* 一样，只持有其元素的弱引用。

finalize 提供了注册一个对象被垃圾收集时要调用的清理函数的方式。这比在原始弱引用上设置回调函数更简单，因为模块会自动确保对象被回收前终结器一直保持存活。

这些弱容器类型之一或者*finalize* 就是大多数程序所需要的 - 通常不需要直接创建自己的弱引用。*weakref* 模块暴露了低级机制，以便于高级用途。

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some *file objects*, *generator*s, type objects, sockets, arrays, deques, regular expression pattern objects, and code objects.

在 3.2 版更改: 添加了对 thread.lock，threading.Lock 和代码对象的支持。

几个内建类型如*list* 和*dict* 不直接支持弱引用，但可以通过子类化添加支持:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

Other built-in types such as *tuple* and *int* do not support weak references even when subclassed (This is an implementation detail and may be different across various Python implementations.).

Extension types can easily be made to support weak references; see weakref-support.

class weakref.**ref**(*object*[, *callback*])

返回对 对象的弱引用。如果原始对象仍然存活，则可以通过调用引用对象来检索原始对象；如果引用的原始对象不再存在，则调用引用对象将得到*None* 。如果提供了 回调而且值不是*None* ，并且返回的弱引用对象仍然存活，则在对象即将终结时将调用回调; 弱引用对象将作为回调的唯一参数传递；指示物将不再可用。

许多弱引用也允许针对相同对象来构建。为每个弱引用注册的回调将按从最近注册的回调到最早注册的回调的顺序被调用。

回调所引发的异常将记录于标准错误输出，但无法被传播；它们会按与对象的 __del__() 方法所引发的异常相同的方式被处理。

如果 *object* 可哈希，则弱引用也为*hashable*。即使在 *object* 被删除之后它们仍将保持其哈希值。如果*hash()* 在 *object* 被删除之后才首次被调用，则该调用将引发*TypeError*。

弱引用支持相等检测，但不支持排序比较。如果被引用对象仍然存在，两个引用具有与它们的被引用对象一致的相等关系（无论 *callback* 是否相同）。如果删除了任一被引用对象，则仅在两个引用对象为同一对象时两者才相等。

这是一个可子类化的类型而非一个工厂函数。

__callback__

这个只读属性会返回当前关联到弱引用的回调。如果回调不存在或弱引用的被引用对象已不存在，则此属性的值为 None。

在 3.4 版更改: 添加了 __callback__ 属性。

weakref.**proxy**(*object*[, *callback*])
返回 *object* 的一个使用弱引用的代理。此函数支持在大多数上下文中使用代理，而不要求显式地对所使用的弱引用对象解除引用。返回的对象类型将为 ProxyType 或 CallableProxyType，具体取决于 *object* 是否可调用。Proxy 对象不是*hashable* 对象，无论被引用对象是否可哈希；这可避免与它们的基本可变性质相关的多种问题，并可防止它们被用作字典键。*callback* 与 ref() 函数的同名形参含义相同。

weakref.**getweakrefcount**(*object*)
返回指向 *object* 的弱引用和代理的数量。

weakref.**getweakrefs**(*object*)
返回由指向 *object* 的所有弱引用和代理构成的列表。

class weakref.**WeakKeyDictionary**([*dict*])
弱引用键的映射类。当不再有对键的强引用时字典中的条目将被丢弃。这可被用来将额外数据关联到一个应用中其他部分所拥有的对象而无需在那些对象中添加属性。这对于重载了属性访问的对象来说特别有用。

注解: Caution: Because a WeakKeyDictionary is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a WeakKeyDictionary because actions performed by the program during iteration may cause items in the dictionary to vanish "by magic" (as a side effect of garbage collection).

WeakKeyDictionary 对象具有一个额外方法可以直接公开内部引用。这些引用不保证在它们被使用时仍然保持“存活”，因此这些引用的调用结果需要在使用前进行检测。此方法可用于避免创建会导致垃圾回收器将保留键超出实际需要时长的引用。

WeakKeyDictionary.**keyrefs**()
返回包含对键的弱引用的可迭代对象。

class weakref.**WeakValueDictionary**([*dict*])
弱引用值的映射类。当不再有对键的强引用时字典中的条目将被丢弃。

注解: Caution: Because a WeakValueDictionary is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a WeakValueDictionary because actions performed by the program during iteration may cause items in the dictionary to vanish "by magic" (as a side effect of garbage collection).

WeakValueDictionary 对象具有一个额外方法，此方法存在与WeakKeyDictionary 对象的 keyrefs() 方法相同的问题。

WeakValueDictionary.**valuerefs**()
返回包含对值的弱引用的可迭代对象。

class weakref.**WeakSet**([*elements*])
保持对其元素弱引用的集合类。当不再有对某个元素的强引用时元素将被丢弃。

class weakref.**WeakMethod**(*method*)
一个模拟对绑定方法（即在类中定义并在实例中查找的方法）进行弱引用的自定义 ref 子类。由于绑定方法是临时性的，标准弱引用无法保持它。WeakMethod 包含特别代码用来重新创建绑定方法，直到对象或初始函数被销毁:

```
>>> class C:
...     def method(self):
...         print("method called!")
```

(下页继续)

(续上页)

```
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

3.4 新版功能.

class weakref.**finalize**(*obj*, *func*, **args*, ***kwargs*)

返回一个可调用的终结器对象，该对象将在 *obj* 作为垃圾回收时被调用。与普通的弱引用不同，终结器将总是存活，直到引用对象被回收，这极大地简化了生存期管理。

终结器总是被视为 *存活* 直到它被调用（显式调用或在垃圾回收时隐式调用），调用之后它将 *死亡*。调用存活的终结器将返回 `func(*arg, **kwargs)` 的求值结果，而调用死亡的终结器将返回*None*。

在垃圾收集期间由终结器回调所引发异常将显示于标准错误输出，但无法被传播。它们会按与对象的 `__del__()` 方法或弱引用的回调所引发异常相同的方式被处理。

当程序退出时，剩余的存活终结器会被调用，除非它们的*atexit* 属性已被设为假值。它们会按与创建时相反的顺序被调用。

终结器在*interpreter shutdown* 的后期绝不会发起调用其回调函数，此时模块全局变量很可能已被替换为*None*。

__call__()

如果 *self* 为存活状态则将其标记为已死亡，并返回调用 `func(*args, **kwargs)` 的结果。如果 *self* 已死亡则返回*None*。

detach()

如果 *self* 为存活状态则将其标记为已死亡，并返回元组 `(obj, func, args, kwargs)`。如果 *self* 已死亡则返*None*。

peek()

如果 *self* 为存活状态则返回元组 `(obj, func, args, kwargs)`。如果 *self* 已死亡则返回*None*。

alive

如果终结器为存活状态则该特征属性为真值，否则为假值。

atexit

一个可写的布尔型特征属性，默认为真值。当程序退出时，它会调用所有*atexit* 为真值的剩余存活终结器。它们会按与创建时相反的顺序被调用。

注解: 很重要的一点是确保 *func*, *args* 和 *kwargs* 不拥有任何对 *obj* 的引用，无论是直接的或是间接的，否则的话 *obj* 将永远不会被作为垃圾回收。特别地，*func* 不应当是 *obj* 的一个绑定方法。

3.4 新版功能.

weakref.**ReferenceType**

弱引用对象的类型对象。

weakref.**ProxyType**
不可调用的对象的代理的类型对象。

weakref.**CallableProxyType**
可调用对象的代理的类型对象。

weakref.**ProxyTypes**
包含所有代理的类型对象的序列。这可以用于更方便地检测一个对象是否是代理，而不必依赖于两种代理对象的名称。

exception weakref.**ReferenceError**
Exception raised when a proxy object is used but the underlying object has been collected. This is the same as the standard *ReferenceError* exception.

参见:

PEP 205 - 弱引用 此特性的提议和理由，包括早期实现的链接和其他语言中类似特性的相关信息。

8.8.1 弱引用对象

弱引用对象没有*ref.__callback__* 以外的方法和属性。一个弱引用对象如果存在，就允许通过调用它来获取引用：

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

如果引用已不存在，则调用引用对象将返回*None*:

```
>>> del o, o2
>>> print(r())
None
```

检测一个弱引用对象是否仍然存在应该使用表达式 `ref() is not None`。通常，需要使用引用对象的应用代码应当遵循这样的模式:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

使用单独的“存活”测试会在多线程应用中制造竞争条件；其他线程可能导致某个弱引用在该弱引用被调用前就失效；上述的写法在多线程应用和单线程应用中都是安全的。

特别版本的*ref* 对象可以通过子类化来创建。在*WeakValueDictionary* 的实现中就使用了这种方式来减少映射中每个条目的内存开销。这对于将附加信息关联到引用的情况最为适用，但也可以被用于在调用中插入额外处理来提取引用。

这个例子演示了如何将*ref* 的一个子类用于存储有关对象的附加信息并在引用被访问时影响其所返回的值:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, **annotations):
        super(ExtendedRef, self).__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super(ExtendedRef, self).__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.8.2 示例

这个简单的例子演示了一个应用如何使用对象 ID 来提取之前出现过的对象。然后对象的 ID 可以在其它数据结构中使用，而无须强制对象保持存活，但处于存活状态的对象也仍然可以通过 ID 来提取。

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.8.3 终结器对象

使用*finalize*的主要好处在于它能更简便地注册回调函数，而无须保留所返回的终结器对象。例如

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

终结器也可以被直接调用。但是终结器最多只能对回调函数发起一次调用。

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because finalizer dead
>>> del obj                 # callback not called because finalizer dead
```

你可以使用*detach()* 方法来注销一个终结器。该方法将销毁终结器并返回其被创建时传给构造器的参数。

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<__main__.Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

除非你将*atexit* 属性设为*False*，否则终结器在程序退出时如果仍然存活就将被调用。例如

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.8.4 比较终结器与 __del__() 方法

假设我们想创建一个类，用它的实例来代表临时目录。当以下事件中的某一个发生时，这个目录应当与其内容一起被删除：

- 对象被作为垃圾回收，
- 对象的 remove() 方法被调用，或
- 程序退出。

我们可以尝试使用 __del__() 方法来实现这个类，如下所示：

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
```

(下页继续)

(续上页)

```
        return self.name is None

    def __del__(self):
        self.remove()
```

从 Python 3.4 开始，`__del__()` 方法不会再阻止循环引用被作为垃圾回收，并且模块全局变量在*interpreter shutdown* 期间不会被强制设为`None`。因此这段代码在 CPython 上应该会正常运行而不会出现任何问题。

然而，`__del__()` 方法的处理会严重地受到具体实现的影响，因为它依赖于解释器垃圾回收实现方式的内部细节。

更健壮的替代方式可以是定义一个终结器，只引用它所需要的特定函数和对象，而不是获取对整个对象状态的访问权:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

像这样定义后，我们的终结器将只接受一个对其完成正确清理目录任务所需细节的引用。如果对象一直未被作为垃圾回收，终结器仍会在退出时被调用。

基于弱引用的终结器还具有另一项优势，就是它们可被用来为定义由第三方控制的类注册终结器，例如当一个模块被卸载时运行特定代码:

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the function body
weakref.finalize(sys.modules[__name__], unloading_module)
```

注解: 如果当程序退出时你恰好在守护线程中创建终结器对象，则有可能该终结器不会在退出时被调用。但是，在一个守护线程中*atexit.register()*，`try: ... finally: ...` 和 `with: ...` 同样不能保证执行清理。

8.9 types —动态类型创建和内置类型名称

源代码: Lib/types.py

This module defines utility function to assist in dynamic creation of new types.

它还为某些对象类型定义了名称，这些名称由标准 Python 解释器所使用，但并不像内置的*int* 或*str* 那样对外公开。

最后，它还额外提供了一些类型相关但重要程度不足以作为内置对象的工具类和函数。

8.9.1 动态类型创建

types.**new_class**(*name*, *bases=()*, *kwds=None*, *exec_body=None*)
使用适当的元类动态地创建一个类对象。

前三个参数是组成类定义头的部件：类名称，基类 (有序排列)，关键字参数 (例如 metaclass)。

exec_body 参数是一个回调函数，用于填充新创建类的命名空间。它应当接受类命名空间作为其唯一的参数并使用类内容直接更新命名空间。如果未提供回调函数，则它就等效于传入 lambda ns: ns。

3.3 新版功能.

types.**prepare_class**(*name*, *bases=()*, *kwds=None*)
计算适当的元类并创建类命名空间。

参数是组成类定义头的部件：类名称，基类 (有序排列) 以及关键字参数 (例如 metaclass)。

返回值是一个 3 元组: metaclass, namespace, kwds

metaclass 是适当的元类，*namespace* 是预备好的类命名空间而 *kwds* 是所传入 *kwds* 参数移除每个 'metaclass' 条目后的已更新副本。如果未传入 *kwds* 参数，这将为一个空字典。

3.3 新版功能.

在 3.6 版更改: The default value for the namespace element of the returned tuple has changed. Now an insertion-order-preserving mapping is used when the metaclass does not have a __prepare__ method,

参见:

metaclasses 这些函数所支持的类创建过程的完整细节

PEP 3115 - Python 3000 中的元类 引入 __prepare__ 命名空间钩子

8.9.2 标准解释器类型

此模块为许多类型提供了实现 Python 解释器所要求的名称。它刻意地避免了包含某些仅在处理过程中偶然出现的类型，例如 listiterator 类型。

此种名称的典型应用如*isinstance()* 或*issubclass()* 检测。

以下类型有相应的标准名称定义：

types.**FunctionType**
types.**LambdaType**
用户自定义函数以及由 lambda 表达式所创建函数的类型。

types.**GeneratorType**
generator 迭代器对象的类型，由生成器函数创建。

types.**CoroutineType**
coroutine 对象的类型，由 async def 函数创建。

3.5 新版功能.

types.**AsyncGeneratorType**
asynchronous generator 迭代器对象的类型，由异步生成器函数创建。

3.6 新版功能.

types.**CodeType**
代码对象的类型，例如*compile()* 的返回值。

types.**MethodType**
用户自定义类实例方法的类型。

types.**BuiltinFunctionType**
types.**BuiltinMethodType**
内置函数例如*len()* 或*sys.exit()* 以及内置类方法的类型。(这里所说的“内置”是指“以 C 语言编写”。)

class types.**ModuleType**(*name*, *doc=None*)
模块 的类型。构造器接受待创建模块的名称及其作为可选项*docstring*。

注解: 如果你希望设置各种由导入控制的属性，请使用*importlib.util.module_from_spec()* 来创建一个新模块。

__doc__
模块的*docstring*。默认为 None。

__loader__
用于加载模块的*loader*。默认为 None。

在 3.4 版更改: 默认为 None。之前该属性为可选项。

__name__
模块的名字

__package__
一个模块所属的*package*。如果模块为最高层级的（即不是任何特定包的组成部分）则该属性应设为 ''，否则它应设为特定包的名称 (如果模块本身也是一个包则名称可以为 __name__)。默认为 None。

在 3.4 版更改: 默认为 None。之前该属性为可选项。

types.**TracebackType**
回溯对象的类型，例如 sys.exc_info()[2] 中的对象。

types.**FrameType**
帧对象的类型，例如 tb.tb_frame 中的对象，其中 tb 是一个回溯对象。

types.**GetSetDescriptorType**
使用 PyGetSetDef 在扩展模块中定义的对象的类型，例如 FrameType.f_locals 或 array.array.typecode。此类型被用作对象属性的描述器；它的目的与*property* 类型相同，但专门针对在扩展模块中定义的类。

types.**MemberDescriptorType**
使用 PyMemberDef 在扩展模块中定义的对象的类型，例如 datetime.timedelta.days。此类型被用作使用标准转换函数的简单 C 数据成员的描述器；它的目的与*property* 类型相同，但专门针对在扩展模块中定义的类。

CPython implementation detail: 在 Python 的其它实现中，此类型可能与 GetSetDescriptorType 完全相同。

class types.**MappingProxyType**(*mapping*)
一个映射的只读代理。它提供了对映射条目的动态视图，这意味着当映射发生改变时，视图会反映这些改变。

3.3 新版功能.

key in proxy
如果下层的映射中存在键 *key* 则返回 True，否则返回 False。

proxy[key]
返回下层的映射中以 *key* 为键的项。如果下层的映射中不存在键 *key* 则引发*KeyError*。

iter(proxy)
返回由下层映射的键为元素的迭代器。这是 iter(proxy.keys()) 的快捷方式。

len(proxy)
返回下层映射中的项数。

copy()
返回下层映射的浅拷贝。

get(*key*[, *default*])
如果 *key* 存在于下层映射中则返回 *key* 的值，否则返回 *default*。如果 *default* 未给出则默认为 None，因而此方法绝不会引发 *KeyError*。

items()
返回由下层映射项 ((键, 值) 对) 组成的一个新视图。

keys()
返回由下层映射的键组成的一个新视图。

values()
返回由下层映射的值组成的一个新视图。

8.9.3 附加工具类和函数

class types.**SimpleNamespace**
一个简单的 *object* 子类，提供了访问其命名空间的属性，以及一个有意义的 repr。

不同于 *object*，对于 SimpleNamespace 你可以添加和移除属性。如果一个 SimpleNamespace 对象使用关键字参数进行初始化，这些参数会被直接加入下层命名空间。

此类型大致等价于以下代码:

```
class SimpleNamespace:
    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        keys = sorted(self.__dict__)
        items = ("{}={!r}".format(k, self.__dict__[k]) for k in keys)
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        return self.__dict__ == other.__dict__
```

SimpleNamespace 可被用于替代 class NS: pass。但是，对于结构化记录类型则应改用 *namedtuple()*。

3.3 新版功能.

types.**DynamicClassAttribute**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)
在类上访问 __getattr__ 的路由属性。

这是一个描述器，用于定义通过实例与通过类访问时具有不同行为的属性。当实例访问时保持正常行为，但当类访问属性时将被路由至类的 __getattr__ 方法；这是通过引发 AttributeError 来完成的。

这样就允许有在实例上激活的特征属性，同时又有在类上的同名虚拟属性（一个这样的例子是 Enum）。

3.4 新版功能.

8.9.4 协程工具函数

types.**coroutine**(*gen_func*)
此函数可将*generator* 函数转换为返回基于生成器的协程的*coroutine function*。基于生成器的协程仍然属于*generator iterator*，但同时又可被视为*coroutine* 对象兼*awaitable*。不过，它没有必要实现 __await__()方法。

如果 *gen_func* 是一个生成器函数，它将被原地修改。

如果 *gen_func* 不是一个生成器函数，则它会被包装。如果它返回一个*collections.abc.Generator* 的实例，该实例将被包装在一个 *awaitable* 代理对象中。所有其他对象类型将被原样返回。

3.5 新版功能.

8.10 copy —浅层 (shallow) 和深层 (deep) 复制操作

源代码: Lib/copy.py

Python 中赋值语句不复制对象，而是在目标和对象之间创建绑定 (bindings) 关系。对于自身可变或者包含可变项的集合对象，开发者有时会需要生成其副本用于改变操作，进而避免改变原对象。本模块提供了通用的浅层复制和深层复制操作（如下所述）。

接口摘要：

copy.**copy**(*x*)
返回 *x* 的浅层复制。

copy.**deepcopy**(*x*[, *memo*])
返回 *x* 的深层复制。

exception copy.**error**
针对模块特定错误引发。

浅层复制和深层复制之间的区别仅与复合对象 (即包含其他对象的对象，如列表或类的实例) 相关:

- 一个 *浅层复制*会构造一个新的复合对象，然后（在可能的范围内）将原对象中找到的 *引用*插入其中。
- 一个 *深层复制*会构造一个新的复合对象，然后递归地将原始对象中所找到的对象的 *副本*插入。

深度复制操作通常存在两个问题，而浅层复制操作并不存在这些问题：

- 递归对象 (直接或间接包含对自身引用的复合对象) 可能会导致递归循环。
- 由于深层复制会复制所有内容，因此可能会过多复制（例如本应该在副本之间共享的数据）。

The *deepcopy()* function avoids these problems by:

- 保留在当前复制过程中已复制的对象的“备忘录”（memo）字典；以及
- 允许用户定义的类重载复制操作或复制的组件集合。

该模块不复制模块、方法、栈追踪（stack trace）、栈帧（stack frame）、文件、套接字、窗口、数组以及任何类似的类型。它通过不改变地返回原始对象来（浅层或深层地）“复制”函数和类；这与*pickle* 模块处理这类问题的方式是相似的。

制作字典的浅层复制可以使用*dict.copy()* 方法，而制作列表的浅层复制可以通过赋值整个列表的切片完成，例如，copied_list = original_list[:]。

类可以使用与控制序列化（pickling）操作相同的接口来控制复制操作，关于这些方法的描述信息请参考*pickle* 模块。实际上，*copy* 模块使用的正是从*copyreg* 模块中注册的 pickle 函数。

想要给一个类定义它自己的拷贝操作实现，可以通过定义特殊方法 `__copy__()` 和 `__deepcopy__()`。调用前者以实现浅层拷贝操作，该方法不用传入额外参数。调用后者以实现深层拷贝操作；它应传入一个参数即 memo 字典。如果 `__deepcopy__()` 实现需要创建一个组件的深层拷贝，它应当调用*deepcopy()* 函数并以该组件作为第一个参数，而将 memo 字典作为第二个参数。

参见:

模块*pickle* 讨论了支持对象状态检索和恢复的特殊方法。

8.11 pprint —数据美化输出

源代码: Lib/pprint.py

---

pprint 模块提供了“美化打印”任意 Python 数据结构的功能，这种美化形式可用作对解释器的输入。如果经格式化的结构包含非基本 Python 类型的对象，则其美化形式可能无法被加载。包含文件、套接字或类对象，以及许多其他不能用 Python 字面值来表示的对象都有可能导致这样的结果。

格式化后的形式会在可能的情况下以单行来表示对象，并在无法在允许宽度内容纳对象的情况下将其分为多行。如果你需要调整宽度限制则应显式地构造*PrettyPrinter* 对象。

Dictionaries are sorted by key before the display is computed. 字典在被计算前通过键值来排序。

pprint 模块定义了一个类:

class pprint.**PrettyPrinter**(*indent=1*, *width=80*, *depth=None*, *stream=None*, ***, *compact=False*)

构造一个*PrettyPrinter* 实例。此构造器接受几个关键字形参。使用 *stream* 关键字可设置输出流；在流对象上使用的唯一方法是文件协议的 `write()` 方法。如果未指定此关键字，则*PrettyPrinter* 会选择 `sys.stdout`。每个递归层级的缩进量由 *indent* 指定；默认值为一。其他值可导致输出看起来有些怪异，但可使得嵌套结构更易区分。可被打印的层级数量由 *depth* 控制；如果数据结构的层级被打印得过深，其所包含的下一层级会被替换为 `...`。在默认情况下，对被格式化对象的层级深度没有限制。希望的输出宽度可使用 *width* 形参来限制；默认值为 80 个字符。如果一个结构无法在限定宽度内被格式化，则将做到尽可能接近。如果 *compact* 为假值（默认）则长序列的每一项将被格式化为单独的行。如果 *compact* 为真值，则格式化将在 *width* 可容纳的情况下把尽可能多的项放入每个输出行。

在 3.4 版更改: 加入 *compact* 参数。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))
```

(下页继续)

(续上页)

```
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

pprint 模块还提供了一些快捷函数：

pprint.**pformat**(*object*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*)
将 *object* 格式化表示作为字符串返回。*indent*, *width*, *depth* 和 *compact* 将作为格式化形参被传入*PrettyPrinter* 构造器。

在 3.4 版更改: 加入 *compact* 参数。

pprint.**pprint**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, ***, *compact=False*)
在 *stream* 上打印 *object* 的格式化表示，并附带一个换行符。如果 *stream* 为 `None`，则使用 `sys.stdout`。这可以替换*print()* 函数在交互式解释器中使用以查看值（你甚至可以执行重新赋值 `print = pprint.pprint` 以在特定作用域中使用）。*indent*, *width*, *depth* 和 *compact* 将作为格式化形参被传给*PrettyPrinter* 构造器。

在 3.4 版更改: 加入 *compact* 参数。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

pprint.**isreadable**(*object*)
Determine if the formatted representation of *object* is "readable," or can be used to reconstruct the value using *eval()*. This always returns `False` for recursive objects.

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)
确定 *object* 是否需要递归表示。

此外还定义了一个支持函数：

pprint.**saferepr**(*object*)
返回 *object* 的字符串表示，并为递归数据结构提供保护。如果 *object* 的表示形式公开了一个递归条目，该递归引用会被表示为 `<Recursion on typename with id=number>`。该表示因而不会进行其它格式化。

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.11.1 PrettyPrinter 对象

PrettyPrinter 的实例具有下列方法：

PrettyPrinter.**pformat**(*object*)
　返回 *object* 格式化表示。这会将传给*PrettyPrinter* 构造器的选项纳入考虑。

PrettyPrinter.**pprint**(*object*)
　在所配置的流上打印 *object* 的格式化表示，并附加一个换行符。

下列方法提供了与同名函数相对应的实现。在实例上使用这些方法效率会更高一些，因为不需要创建新的*PrettyPrinter* 对象。

PrettyPrinter.**isreadable**(*object*)
　确定对象的格式化表示是否“可读”，或者是否可使用*eval()* 重建对象值。请注意此方法对于递归对象将返回 `False`。如果设置了*PrettyPrinter* 的 *depth* 形参并且对象深度超出允许范围，此方法将返回 `False`。

PrettyPrinter.**isrecursive**(*object*)
　确定对象是否需要递归表示。

此方法作为一个钩子提供，允许子类修改将对象转换为字符串的方式。默认实现使用*saferepr()* 实现的内部方式。

PrettyPrinter.**format**(*object*, *context*, *maxlevels*, *level*)
　返回三个值：字符串形式的 *object* 已格式化版本，指明结果是否可读的旗标，以及指明是否检测到递归的旗标。第一个参数是要表示的对象。第二个是以对象*id()* 为键的字典，这些对象是当前表示上下文的一部分（影响 *object* 表示的直接和间接容器）；如果需要呈现一个已经在 *context* 中表示的对象，则第三个返回值应当为 `True`。对*format()* 方法的递归调用应当将容器的附加条目添加到此字典中。第三个参数 *maxlevels* 给出了对递归的请求限制；如果没有请求限制则其值将为 `0`。此参数应当不加修改地传给递归调用。第四个参数 *level* 给出于当前层级；传给递归调用的参数值应当小于当前调用的值。

8.11.2 示例

为了演示*pprint()* 函数及其形参的几种用法，让我们从 PyPI 获取关于某个项目的信息：

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/json') as resp:
...     project_info = json.load(resp)['info']
```

pprint() 以其基本形式显示了整个对象：

```
>>> pprint.pprint(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
```

(下页继续)

(续上页)

```
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

结果可以被限制到特定的 *depth* (更深层的内容将使用省略号):

```
>>> pprint.pprint(project_info, depth=1)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
```

(下页继续)

(续上页)

```
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

此外，还可以设置建议的最大字符 *width*。如果一个对象无法被拆分，则将超出指定宽度:

```
>>> pprint.pprint(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the '
                'project.\n'
                '\n'
                'The file should use UTF-8 encoding and be '
                'written using ReStructured Text. It\n'
                'will be used to generate the project '
                'webpage on PyPI, and should be written '
                'for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would '
                'include an overview of the project, '
                'basic\n'
                'usage examples, etc. Generally, including '
                'the project changelog in here is not\n'
                'a good idea, although a simple "What\'s '
```

(下页继续)

(续上页)

```
                'New" section for the most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

8.12 reprlib —另一种 repr() 实现

源代码: Lib/reprlib.py

---

reprlib 模块提供了一种对象表示的产生方式，它会对结果字符串的大小进行限制。该方式被用于 Python 调试器，也适用于某些其他场景。

此模块提供了一个类、一个实例和一个函数：

class reprlib.**Repr**
该类提供了格式化服务适用于实现与内置*repr()* 相似的方法；其中附加了针对不同对象类型的大小限制，以避免生成超长的表示。

reprlib.**aRepr**
这是*Repr* 的一个实例，用于提供如下所述的*repr()* 函数。改变此对象的属性将会影响*repr()* 和 Python 调试器所使用的大小限制。

reprlib.**repr**(*obj*)
这是 aRepr 的*repr()* 方法。它会返回与同名内置函数所返回字符串相似的字符串，区别在于附带了对多数类型的大小限制。

在大小限制工具以外，此模块还提供了一个装饰器，用于检测对 __repr__() 的递归调用并改用一个占位符来替换。

@reprlib.**recursive_repr**(*fillvalue="..."*)
用于为 __repr__() 方法检测同一线程内部递归调用的装饰器。如果执行了递归调用，则会返回 *fillvalue*，否则执行正常的 __repr__() 调用。例如：

```
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
```

(下页继续)

(续上页)

```
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

3.2 新版功能.

8.12.1 Repr 对象

Repr 实例对象包含一些属性可以用于为不同对象类型的表示提供大小限制，还包含一些方法可以格式化特定的对象类型。

Repr.**maxlevel**
: 创建递归表示形式的深度限制。默认为 6。

Repr.**maxdict**
Repr.**maxlist**
Repr.**maxtuple**
Repr.**maxset**
Repr.**maxfrozenset**
Repr.**maxdeque**
Repr.**maxarray**
: 表示命名对象类型的条目数量限制。对于*maxdict* 的默认值为 4，对于*maxarray* 为 5，对于其他则为 6。

Repr.**maxlong**
: 表示整数的最大字符数量。数码会从中间被丢弃。默认值为 40。

Repr.**maxstring**
: 表示字符串的字符数量限制。请注意字符源会使用字符串的“正常”表示形式：如果表示中需要用到转义序列，在缩短表示时它们可能会被破坏。默认值为 30。

Repr.**maxother**
: 此限制用于控制在*Repr* 对象上没有特定的格式化方法可用的对象类型的大小。它会以类似*maxstring* 的方式被应用。默认值为 20。

Repr.**repr**(*obj*)
: 内置*repr()* 的等价形式，它使用实例专属的格式化。

Repr.**repr1**(*obj*, *level*)
: 供*repr()* 使用的递归实现。此方法使用 *obj* 的类型来确定要调用哪个格式化方法，并传入 *obj* 和 *level*。类型专属的方法应当调用*repr1()* 来执行递归格式化，在递归调用中使用 `level - 1` 作为 *level* 的值。

Repr.**repr_TYPE**(*obj*, *level*)
: 特定类型的格式化方法会被实现为基于类型名称来命名的方法。在方法名称中，**TYPE** 会被替换为 `'_'.join(type(obj).__name__.split())`。对这些方法的分派会由*repr1()* 来处理。需要对值进行递归格式化的类型专属方法应当调用 `self.repr1(subobj, level - 1)`。

8.12.2 子类化 Repr 对象

通过*Repr.repr1()* 使用动态分派允许*Repr* 的子类添加对额外内置对象类型的支持，或是修改对已支持类型的处理。这个例子演示了如何添加对文件对象的特殊支持:

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))          # prints '<stdin>'
```

8.13 enum —枚举类型支持

3.4 新版功能.

源代码： Lib/enum.py

枚举是一组符号名称（枚举成员）的集合，枚举成员应该是唯一的、不可变的。在枚举中，可以对成员进行恒等比较，并且枚举本身是可迭代的。

8.13.1 模块内容

此模块定义了四个枚举类，它们可被用来定义名称和值的不重复集合: *Enum*, *IntEnum*, *Flag* 和*IntFlag*。此外还定义了一个装饰器*unique()* 和一个辅助类*auto*。

class enum.**Enum**
用于创建枚举型常数的基类。请参阅*Functional API* 小节了解另一种替代性的构建语法。

class enum.**IntEnum**
用于创建同时也是*int* 的子类的枚举型常数的基类。

class enum.**IntFlag**
此基类用于创建可使用按位运算符进行组合而不会丢失其*IntFlag* 成员资格的枚举常量。*IntFlag* 成员同样也是*int* 的子类。

class enum.**Flag**
此基类用于创建枚举常量可使用按位运算符进行组合而不会丢失其*Flag* 成员资格的枚举常量。

enum.**unique**()
此 Enum 类装饰器可确保只将一个名称绑定到任意一个值。

class enum.**auto**
Instances are replaced with an appropriate value for Enum members.

3.6 新版功能: `Flag`, `IntFlag`, `auto`

8.13.2 创建一个 Enum

枚举是使用 `class` 语法来创建的，这使得它们易于读写。另一种替代创建方法的描述见*Functional API*。要定义一个枚举，可以对*Enum* 进行如下的子类化:

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...
```

注解： Enum 的成员值

成员值可以为任意类型: *int*, *str* 等等。如果具体的值不重要，你可以使用*auto* 实例，将为你选择适当的值。但如果你混用*auto* 与其他值则需要小心谨慎。

注解： 命名法

- 类 `Color` 是一个 *enumeration* (或称 *enum*)
- 属性 `Color.RED`, `Color.GREEN` 等等是 *枚举成员* (或称 *enum 成员*) 并且被用作常量。
- 枚举成员具有 *名称* 和 *值* (例如 `Color.RED` 的名称为 `RED`，`Color.BLUE` 的值为 `3` 等等)

注解： 虽然我们使用 `class` 语法来创建 Enum，但 Enum 并不是普通的 Python 类。更多细节请参阅*How are Enums different?*。

枚举成员具有适合人类阅读的表示形式:

```
>>> print(Color.RED)
Color.RED
```

…而它们的 `repr` 包含更多信息:

```
>>> print(repr(Color.RED))
<Color.RED: 1>
```

一个枚举成员的 *type* 就是它所从属的枚举:

```
>>> type(Color.RED)
<enum 'Color'>
>>> isinstance(Color.GREEN, Color)
True
>>>
```

Enum 的成员还有一个包含其条目名称的特征属性:

```
>>> print(Color.RED.name)
RED
```

枚举支持按照定义顺序进行迭代:

```
>>> class Shake(Enum):
...     VANILLA = 7
...     CHOCOLATE = 4
...     COOKIES = 9
...     MINT = 3
...
>>> for shake in Shake:
...     print(shake)
...
Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT
```

枚举成员是可哈希的，因此它们可在字典和集合中可用:

```
>>> apples = {}
>>> apples[Color.RED] = 'red delicious'
>>> apples[Color.GREEN] = 'granny smith'
>>> apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
True
```

8.13.3 对枚举成员及其属性的程序化访问

有时对枚举中的成员进行程序化访问是很有用的（例如在某些场合不能使用 `Color.RED` 因为在编程时并不知道要指定的确切颜色）。`Enum` 允许这样的访问:

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

如果你希望通过 *name* 来访问枚举成员，可使用条目访问:

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

如果你有一个枚举成员并且需要它的 `name` 或 `value`:

```
>>> member = Color.RED
>>> member.name
'RED'
>>> member.value
1
```

8.13.4 复制枚举成员和值

不允许有同名的枚举成员:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'SQUARE'
```

但是，允许两个枚举成员有相同的值。假定两个成员 A 和 B 有相同的值（且 A 先被定义），则 B 就是 A 的一个别名。按值查找 A 和 B 的值将返回 A。按名称查找 B 也将返回 A:

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

注解: 试图创建具有与某个已定义的属性（另一个成员或方法等）相同名称的成员或者试图创建具有相同名称的属性也是不允许的。

8.13.5 确保唯一的枚举值

默认情况下，枚举允许有多个名称作为某个相同值的别名。如果不想要这样的行为，可以使用以下装饰器来确保每个值在枚举中只被使用一次:

`@enum.unique`

专用于枚举的 `class` 装饰器。它会搜索一个枚举的 `__members__` 并收集所找到的任何别名；只要找到任何别名就会引发*`ValueError`* 并附带相关细节信息:

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

8.13.6 使用自动设定的值

如果确切的值不重要，你可以使用*auto*:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

值将由 _generate_next_value_() 来选择，该函数可以被重载:

```
>>> class AutoName(Enum):
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>, <Ordinal.EAST: 'EAST'>, <Ordinal.
↪WEST: 'WEST'>]
```

注解: 默认 _generate_next_value_() 方法的目标是提供所给出的最后一个*int* 所在序列的下一个*int*，但这种行为方式属于实现细节并且可能发生改变。

8.13.7 迭代

对枚举成员的迭代不会给出别名:

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
```

特殊属性 __members__ 是一个将名称映射到成员的有序字典。它包含枚举中定义的所有名称，包括别名:

```
>>> for name, member in Shape.__members__.items():
...     name, member
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

__members__ 属性可被用于对枚举成员进行详细的程序化访问。例如，找出所有别名:

```
>>> [name for name, member in Shape.__members__.items() if member.name != name]
['ALIAS_FOR_SQUARE']
```

8.13.8 比较运算

枚举成员是按标识号进行比较的:

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

枚举值之间的排序比较 不被支持。Enum 成员不属于整数 (另请参阅下文的*IntEnum*):

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color' and 'Color'
```

相等比较的定义如下:

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

与非枚举值的比较将总是不相等（同样地，*IntEnum* 被显式设计成不同的行为，参见下文）：

```
>>> Color.BLUE == 2
False
```

8.13.9 允许的枚举成员和属性

以上示例使用整数作为枚举值。使用整数相当简洁方便（并由*Functional API* 默认提供），但并不强制要求使用。在大部分用例中，开发者都关心枚举的实际值是什么。但如果值 确实重要，则枚举可以使用任意的值。

枚举属于 Python 的类，并可具有普通方法和特殊方法。如果我们有这样一个枚举:

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
...
...     def describe(self):
...         # self is the member here
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
```

那么:

```
>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
>>> Mood.HAPPY.describe()
('HAPPY', 3)
>>> str(Mood.FUNKY)
'my custom str! 1'
```

The rules for what is allowed are as follows: names that start and end with a single underscore are reserved by enum and cannot be used; all other attributes defined within an enumeration will become members of this enumeration, with the exception of special methods (`__str__()`, `__add__()`, etc.) and descriptors (methods are also descriptors).

注意：如果你的枚举定义了 `__new__()` 和/或 `__init__()` 那么指定给枚举成员的任何值都会被传入这些方法。请参阅示例*Planet*。

8.13.10 Restricted subclassing of enumerations

Subclassing an enumeration is allowed only if the enumeration does not define any members. So this is forbidden:

```
>>> class MoreColor(Color):
...     PINK = 17
...
Traceback (most recent call last):
...
TypeError: Cannot extend enumerations
```

但是允许这样的写法:

```
>>> class Foo(Enum):
...     def some_behavior(self):
...         pass
...
>>> class Bar(Foo):
...     HAPPY = 1
...     SAD = 2
...
```

允许子类化定义了成员的枚举将会导致违反类型与实例的某些重要的不可变规则。在另一方面，允许在一组枚举之间共享某些通用行为也是有意义的。(请参阅示例*OrderedEnum* 。)

8.13.11 封存

枚举可以被封存与解封:

```
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
True
```

封存的常规限制同样适用：可封存枚举必须在模块的最高层级中定义，因为解封操作要求它们可以从该模块导入。

注解： 使用 pickle 协议版本 4 可以方便地封存嵌套在其他类中的枚举。

通过在枚举类中定义*`__reduce_ex__()`* 可以对 Enum 成员的封存/解封方式进行修改。

8.13.12 可用 API

Enum 类属于可调用对象，它提供了以下可用的 API:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> Animal.ANT.value
1
>>> list(Animal)
[<Animal.ANT: 1>, <Animal.BEE: 2>, <Animal.CAT: 3>, <Animal.DOG: 4>]
```

该 API 的主义类似于*namedtuple*。调用*Enum* 的第一个参数是枚举的名称。

第二个参数是枚举成员名称的 *来源*。它可以是一个用空格分隔的名称字符串、名称序列、键/值对 2 元组的序列，或者名称到值的映射（例如字典）。最后两种选项使得可以为枚举任意赋值；其他选项会自动以从 1 开始递增的整数赋值（使用 `start` 形参可指定不同的起始值）。返回值是一个派生自*Enum* 的新类。换句话说，以上对 `Animal` 的赋值就等价于:

```
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
...
```

默认以 `1` 而以 `0` 作为起始数值的原因在于 `0` 的布尔值为 `False`，但所有枚举成员都应被求值为 `True`。

封存通过功能性 API 创建的枚举可能会有点麻烦，因为要使用帧堆栈的实现细节来尝试并找出枚举是在哪个模块中创建的（例如，当你使用不同模块的工具函数时可能会失败，在 IronPython 或 Jython 上也可能会没有效果）。解决办法是显式地指定模块名称，如下所示:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module=__name__)
```

警告: 如果未提供 `module`，且 Enum 无法确定是哪个模块，新的 Enum 成员将不可被解封；为了让错误尽量靠近源头，封存将被禁用。

新的 pickle 协议版本 4 在某些情况下同样依赖于*__qualname__* 被设为特定位置以便 pickle 能够找到相应的类。例如，类是否存在于全局作用域的 SomeData 类中:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', qualname='SomeData.Animal')
```

完整的签名为:

```
Enum(value='NewEnumName', names=<...>, *, module='...', qualname='...', type=<mixed-
↪in class>, start=1)
```

值 将被新 Enum 类将记录为其名称的数据。

names Enum 的成员。这可以是一个空格或逗号分隔的字符串 (起始值将为 1，除非另行指定):

```
'RED GREEN BLUE' | 'RED,GREEN,BLUE' | 'RED, GREEN, BLUE'
```

或是一个名称的迭代器:

```
['RED', 'GREEN', 'BLUE']
```

或是一个 (名称, 值) 对的迭代器:

```
[('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 6)]
```

或是一个映射:

```
{'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 42}
```

module 新 Enum 类所在模块的名称。

qualname 新 Enum 类在模块中的具体位置。

type 要加入新 Enum 类的类型。

start 当只传入名称时要使用的起始数值。

在 3.5 版更改: 增加了 *start* 形参。

8.13.13 派生的枚举

IntEnum

所提供的第一个变种*Enum* 同时也是*int* 的一个子类。*IntEnum* 的成员可与整数进行比较；通过扩展，不同类型的整数枚举也可以相互进行比较:

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Request(IntEnum):
...     POST = 1
...     GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

不过，它们仍然不可与标准*Enum* 枚举进行比较:

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

IntEnum 值在其他方面的行为都如你预期的一样类似于整数:

```
>>> int(Shape.CIRCLE)
1
>>> ['a', 'b', 'c'][Shape.CIRCLE]
'b'
>>> [i for i in range(Shape.SQUARE)]
[0, 1]
```

IntFlag

所提供的下一个*Enum*的变种*IntFlag*同样是基于*int*的，不同之处在于*IntFlag*成员可使用按位运算符 (&, |, ^, ~) 进行合并且结果仍然为*IntFlag*成员。如果，正如名称所表明的，*IntFlag*成员同时也是*int*的子类，并能在任何使用*int*的场合被使用。*IntFlag*成员进行除按位运算以外的其他运算都将导致失去*IntFlag*成员资格。

3.6 新版功能.

示例*IntFlag*类:

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
>>> Perm.R | Perm.W
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
```

对于组合同样可以进行命名:

```
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm.-8: -8>
```

*IntFlag*和*Enum*的另一个重要区别在于如果没有设置任何旗标（值为 0），则其布尔值为*False*:

```
>>> Perm.R & Perm.X
<Perm.0: 0>
>>> bool(Perm.R & Perm.X)
False
```

由于*IntFlag*成员同时也是*int*的子类，因此它们可以相互组合:

```
>>> Perm.X | 8
<Perm.8|X: 9>
```

标志

最后一个变种是*Flag*。与*IntFlag*类似，*Flag*成员可使用按位运算符 (&, |, ^, ~) 进行组合，与*IntFlag*不同的是它们不可与任何其它*Flag*枚举或*int*进行组合或比较。虽然可以直接指定值，但推荐使用*auto*作为值以便让*Flag*选择适当的值。

3.6 新版功能.

与*IntFlag*类似，如果*Flag*成员的某种组合导致没有设置任何旗标，则其布尔值为*False*:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color.0: 0>
>>> bool(Color.RED & Color.GREEN)
False
```

单个旗标的值应当为二的乘方 (1, 2, 4, 8, …)，旗标的组合则无此限制:

```
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
>>> Color.WHITE
<Color.WHITE: 7>
```

对“no flags set”条件指定一个名称并不会改变其布尔值:

```
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

注解： 对于大多数新代码，强烈推荐使用*Enum*和*Flag*，因为*IntEnum*和*IntFlag*打破了枚举的某些语义约定（例如可以同整数进行比较，并因而导致此行为被传递给其他无关的枚举）。*IntEnum*和*IntFlag*的使用应当仅限于*Enum*和*Flag*无法使用的场合；例如，当使用枚举替代整数常量时，或是与其他系统进行交互操作时。

其他事项

虽然*IntEnum*是*enum*模块的一部分，但要独立实现也应该相当容易:

```
class IntEnum(int, Enum):
    pass
```

这里演示了如何定义类似的派生枚举；例如一个混合了*str*而不是*int*的 StrEnum。

几条规则:

1. 当子类化*Enum*时，在基类序列中的混合类型必须出现于*Enum*本身之前，如以上*IntEnum*的例子所示。
2. 虽然*Enum*可以拥有任意类型的成员，不过一旦你混合了附加类型，则所有成员必须为相应类型的值，如在上面的例子中即为*int*。此限制不适用于仅添加方法而未指定另一数据类型如*int*或*str*的混合类。
3. 当混合了另一数据类型时，value 属性会 不同于枚举成员自身，但它们仍保持等价且比较结果也相等。
4. %-style formatting: *%s* 和 *%r* 会分别调用*Enum*类的 __str__() 和 __repr__()；其他代码 (例如表示 IntEnum 的 *%i* 或 *%h*) 会将枚举成员视为对应的混合类型。
5. 格式化字符串字面值, *str.format()* 和*format()* 将使用混合类型的 __format__()。如果需要*Enum*类的*str()* 或*repr()*，请使用 *!s* 或 *!r* 格式代码。

8.13.14 有趣的示例

虽然*Enum*, *IntEnum*, *IntFlag* 和*Flag* 预期可覆盖大多数应用场景，但它们无法覆盖全部。这里有一些不同类型枚举的方案，它们可以被直接使用，或是作为自行创建的参考示例。

省略值

在许多应用场景中人们都不关心枚举的实际值是什么。有几个方式可以定义此种类型的简单枚举:

- 使用*auto*的实例作为值
- 使用*object*的实例作为值
- 使用描述性的字符串作为值
- 使用元组作为值并用自定义的 __new__() 以一个*int*值来替代该元组

使用以上任何一种方法均可向用户指明值并不重要，并且使人能够添加、移除或重排序成员而不必改变其余成员的数值。

无论你选择何种方法，你都应当提供一个*repr()*并且它也需要隐藏（不重要的）值:

```
>>> class NoValue(Enum):
...     def __repr__(self):
...         return '<%s.%s>' % (self.__class__.__name__, self.name)
...
```

使用 auto

使用*auto* 看起来是这样:

```
>>> class Color(NoValue):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN>
```

使用 object

使用*object* 的形式如下:

```
>>> class Color(NoValue):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN>
```

使用描述性字符串

使用字符串作为值的形式如下:

```
>>> class Color(NoValue):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
'go'
```

使用自定义的 __new__()

使用自动编号 __new__() 的形式如下:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
```

(下页继续)

(续上页)

```
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
2
```

注解： 如果定义了 __new__() 则它会在创建 Enum 成员期间被使用；随后它将被 Enum 的 __new__() 所替换，该方法会在类创建后被用来查找现有成员。

OrderedEnum

一个有序枚举，它不是基于*IntEnum*，因此保持了正常的*Enum*不变特性（例如不可与其他枚举进行比较）：

```
>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value < other.value
...         return NotImplemented
...
>>> class Grade(OrderedEnum):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True
```

DuplicateFreeEnum

如果发现重复的成员名称则将引发错误而不是创建别名:

```
>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                 "aliases not allowed in DuplicateFreeEnum:  %r --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum:  'GRENE' --> 'GREEN'
```

注解: 这个例子适用于子类化 Enum 来添加或改变禁用别名以及其他行为。如果需要的改变只是禁用别名，也可以选择使用*unique()* 装饰器。

Planet

如果定义了 `__new__()` 或 `__init__()` 则枚举成员的值将被传给这些方法:

```
>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS   = (4.869e+24, 6.0518e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     MARS    = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27,   7.1492e7)
...     SATURN  = (5.688e+26, 6.0268e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
...     def __init__(self, mass, radius):
...         self.mass = mass       # in kilograms
...         self.radius = radius   # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant  (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
```

8.13.15 各种枚举有何区别？

枚举具有自定义的元类，它会影响所派生枚举类及其实例（成员）的各个方面。

枚举类

The `EnumMeta` metaclass is responsible for providing the `__contains__()`, `__dir__()`, `__iter__()` and other methods that allow one to do things with an *Enum* class that fail on a typical class, such as *list(Color)* or *some_var in Color*. `EnumMeta` is responsible for ensuring that various other methods on the final *Enum* class are correct (such as `__new__()`, *__getnewargs__()*, `__str__()` and `__repr__()`).

枚举成员（即实例）

有关枚举成员最有趣的特点是它们都是单例对象。`EnumMeta` 会在创建*Enum* 类本身时将它们全部创建完成，然后准备好一个自定义的 `__new__()`，通过只返回现有的成员实例来确保不会再实例化新的对象。

细节要点

支持 __dunder__ 名称

`__members__` 是一个 `OrderedDict`，由 `member_name:member` 条目组成。它只在类上可用。

如果指定了 `__new__()`，它必须创建并返回枚举成员；相应地设定成员的 `_value_` 也是一个很好的主意。一旦所有成员都创建完成它就不会再被使用。

支持的 _sunder_ 名称

- `_name_` –成员的名称
- `_value_` –成员的值；可以在 `__new__` 中设置 / 修改
- `_missing_` –当未发现某个值时所使用的查找函数；可被重载
- `_order_` –用于 Python 2/3 代码以确保成员顺序一致（类属性，在类创建期间会被移除）
- `_generate_next_value_` –用于*Functional API* 并通过*auto* 为枚举成员获取适当的值；可被重载

3.6 新版功能: `_missing_`, `_order_`, `_generate_next_value_`

用来帮助 Python 2 / Python 3 代码保持同步提供 `_order_` 属性。它将与枚举的实际顺序进行对照检查，如果两者不匹配则会引发错误:

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_
```

注解: 在 Python 2 代码中 `_order_` 属性是必须的，因为定义顺序在被记录之前就会丢失。

Enum 成员类型

Enum 成员是其*Enum* 类的实例，一般通过 EnumClass.member 的形式来访问。在特定情况下它们也可通过 EnumClass.member.member 的形式来访问，但你绝对不应这样做，因为查找可能失败，或者更糟糕地返回你所查找的*Enum* 成员以外的对象（这也是成员应使用全大写名称的另一个好理由）：

```
>>> class FieldTypes(Enum):
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
<FieldTypes.size: 2>
>>> FieldTypes.size.value
2
```

在 3.5 版更改.

Enum 类和成员的布尔值

混合了非*Enum* 类型（例如*int*, *str* 等）的*Enum* 成员会按所混合类型的规则被求值；在其他情况下，所有成员都将被求值为*True*。要使你的自定义 Enum 的布尔值取决于成员的值，请在你的类中添加以下代码:

```
def __bool__(self):
    return bool(self.value)
```

Enum 类总是会被求值为*True*。

带有方法的 Enum 类

如果你为你的*Enum* 子类添加了额外的方法，如同上述的*Planet* 类一样，这些方法将在对成员执行*dir()* 时显示出来，但对类执行时则不会显示:

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '__
↪class__', '__doc__', '__members__', '__module__']
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value']
```

组合 Flag 的成员

如果 Flag 成员的某种组合未被命名，则*repr()* 将包含所有已命名的旗标和值中所有已命名的旗标组合:

```
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
...     CYAN = GREEN | BLUE
...
>>> Color(3)  # named combination
```

(下页继续)

(续上页)

```
<Color.YELLOW: 3>
>>> Color(7)       # not named combination
<Color.CYAN|MAGENTA|BLUE|YELLOW|GREEN|RED: 7>
```

CHAPTER 9

数字和数学模块

本章介绍的模块提供与数字和数学相关的函数和数据类型。*numbers* 模块定义了数字类型的抽象层次结构。*math* 和*cmath* 模块包含浮点数和复数的各种数学函数。*decimal* 模块支持使用任意精度算术的十进制数的精确表示。

本章包含以下模块的文档:

9.1 numbers —数字的抽象基类

源代码: Lib/numbers.py

numbers 模块 (**PEP 3141**) 定义了数字抽象基类 的层次结构，其中逐级定义了更多操作。此模块中所定义的类型都不可被实例化。

class numbers.**Number**
数字的层次结构的基础。如果你只想确认参数 *x* 是不是数字而不关心其类型，则使用 `isinstance(x, Number)`。

9.1.1 数字的层次

class numbers.**Complex**
内置在类型*complex* 里的子类描述了复数和它的运算操作。这些操作有：转化至*complex* 和*bool*，*real*、*imag*、+、-、*、/、*abs()*、*conjugate()*、== 和 !=。所有的异常，- 和 != ，都是抽象的。

real
抽象的。得到该数字的实数部分。

imag
抽象的。得到该数字的虚数部分。

abstractmethod conjugate()
抽象的。返回共轭复数。例如 `(1+3j).conjugate() == (1-3j)`。

class numbers.**Real**
相对于*Complex*，*Real* 加入了只有实数才能进行的操作。

简单的说，它们是：转化至*float*，*math.trunc()*、*round()*、*math.floor()*、*math.ceil()*、*divmod()*、//、%、<、<=、>、和 >=。

实数同样默认支持*complex()*、*real*、*imag* 和*conjugate()*。

class numbers.**Rational**
子类型*Real* 并加入*numerator* 和*denominator* 两种属性，这两种属性应该属于最低的级别。加入后，这默认支持*float()*。

numerator
抽象的。

denominator
抽象的。

class numbers.**Integral**
子类型*Rational* 加上转化至*int*。默认支持*float()*、*numerator* 和*denominator*。在 ** 中加入抽象方法和比特字符串的操作：<<、>>、&、^、|、~。

9.1.2 类型接口注释。

实现者需要注意使相等的数字相等并拥有同样的值。当这两个数使用不同的扩展模块时，这其中的差异是很微妙的。例如，用*fractions.Fraction* 实现*hash()* 如下:

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

加入更多数字的 ABC

当然，这里有更多支持数字的 ABC，如果不加入这些，就将缺少层次感。你可以用如下方法在*Complex* 和*Real* 中加入 MyFoo:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

实现算术运算

我们希望实现计算，因此，混合模式操作要么调用一个作者知道参数类型的实现，要么转变成为最接近的内置类型并对这个执行操作。对于子类*Integral*，这意味着 __add__() 和 __radd__() 必须用如下方式定义：

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

Complex 有 5 种不同的混合类型的操作。我将上面提到的所有代码作为“模板”称作 MyIntegral 和 OtherTypeIKnowAbout。a 是*Complex* 的子类型 A 的实例 (a : A <: Complex)，同时 b : B <: Complex。我将要计算 a + b:

1. 如果 A 被定义成一个承认 b 的 __add__()，一切都没有问题。
2. 如果 A 转回成“模板”失败，它将返回一个属于 __add__() 的值，我们需要避免 B 定义了一个更加智能的 __radd__()，因此模板需要返回一个属于 __add__() 的*NotImplemented*。(或者 A 可能完全不实现 __add__()。)
3. 接着看 B 的 __radd__()。如果它承认 a，一切都没有问题。
4. 如果没有成功回退到模板，就没有更多的方法可以去尝试，因此这里将使用默认的实现。
5. 如果 B <: A，Python 在 A.__add__ 之前尝试 B.__radd__。这是可行的，是通过对 A 的认识实现的，因此这可以在交给*Complex* 处理之前处理这些实例。

如果 A <: Complex 和 B <: Real 没有共享任何资源，那么适当的共享操作涉及内置的*complex*，并且分别获得 __radd__()，因此 a+b == b+a。

由于对任何一直类型的大部分操作是十分相似的，可以定义一个帮助函数，即一个生成后续或相反的实例的生成器。例如，使用*fractions.Fraction* 如下：

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
```

(下页继续)

(续上页)

```
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 math —数学函数

This module is always available. It provides access to the mathematical functions defined by the C standard.

这些函数不适用于复数；如果你需要计算复数，请使用*cmath*模块中的同名函数。将支持计算复数的函数区分开的目的，来自于大多数开发者并不愿意像数学家一样需要学习复数的概念。得到一个异常而不是一个复数结果使得开发者能够更早地监测到传递给这些函数的参数中包含复数，进而调查其产生的原因。

该模块提供了以下函数。除非另有明确说明，否则所有返回值均为浮点数。

9.2.1 数论与表示函数

math.**ceil**(*x*)
: 返回 *x* 的上限，即大于或者等于 *x* 的最小整数。如果 *x* 不是一个浮点数，则委托 `x.__ceil__()`，返回一个*Integral*类的值。

math.**copysign**(*x*, *y*)
: 返回一个基于 *x* 的绝对值和 *y* 的符号的浮点数。在支持带符号零的平台上，`copysign(1.0, -0.0)` 返回 *-1.0*.

math.**fabs**(*x*)
返回 *x* 的绝对值。

math.**factorial**(*x*)
Return *x* factorial. Raises *ValueError* if *x* is not integral or is negative.

math.**floor**(*x*)
返回 *x* 的向下取整，小于或等于 *x* 的最大整数。如果 *x* 不是浮点数，则委托 `x.__floor__()`，它应返回*Integral* 值。

math.**fmod**(*x*, *y*)
返回 `fmod(x, y)`，由平台 C 库定义。请注意，Python 表达式 `x % y` 可能不会返回相同的结果。C 标准的目的是 `fmod(x, y)` 完全（数学上；到无限精度）等于 `x - n*y` 对于某个整数 *n*，使得结果具有与 *x* 相同的符号和小于 `abs(y)` 的幅度。Python 的 `x % y` 返回带有 *y* 符号的结果，并且可能不能完全计算浮点参数。例如，`fmod(-1e-100, 1e100)` 是 `-1e-100`，但 Python 的 `-1e-100 % 1e100` 的结果是 `1e100-1e-100`，它不能完全表示为浮点数，并且取整为令人惊讶的 `1e100`。出于这个原因，函数*fmod()* 在使用浮点数时通常是首选，而 Python 的 `x % y` 在使用整数时是首选。

math.**frexp**(*x*)
以 `(m, e)` 对的形式返回 *x* 的尾数和指数。*m* 是一个浮点数，*e* 是一个整数，正好是 `x == m * 2**e`。如果 *x* 为零，则返回 `(0.0, 0)`，否则返回 `0.5 <= abs(m) < 1`。这用于以可移植方式“分离”浮点数的内部表示。

math.**fsum**(*iterable*)
返回迭代中的精确浮点值。通过跟踪多个中间部分和来避免精度损失:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

该算法的准确性取决于 IEEE-754 算术保证和舍入模式为半偶的典型情况。在某些非 Windows 版本中，底层 C 库使用扩展精度添加，并且有时可能会使中间和加倍，导致它在最低有效位中关闭。

有关待进一步讨论和两种替代方法，参见 ASPN cookbook recipes for accurate floating point summation。

math.**gcd**(*a*, *b*)
返回整数 *a* 和 *b* 的最大公约数。如果 *a* 或 *b* 之一非零，则 `gcd(a, b)` 的值是能同时整除 *a* 和 *b* 的最大正整数。`gcd(0, 0)` 返回 `0`。

3.5 新版功能.

math.**isclose**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)
若 *a* 和 *b* 的值比较接近则返回 `True`，否则返回 `False`。

根据给定的绝对和相对容差确定两个值是否被认为是接近的。

rel_tol 是相对容差——它是 *a* 和 *b* 之间允许的最大差值，相对于 *a* 或 *b* 的较大绝对值。例如，要设置 5% 的容差，请传递 `rel_tol=0.05`。默认容差为 `1e-09`，确保两个值在大约 9 位十进制数字内相同。*rel_tol* 必须大于零。

abs_tol 是最小绝对容差——对于接近零的比较很有用。*abs_tol* 必须至少为零。

如果没有错误发生，结果将是: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`。

IEEE 754 特殊值 `NaN`，`inf` 和 `-inf` 将根据 IEEE 规则处理。具体来说，`NaN` 不被认为接近任何其他值，包括 `NaN`。`inf` 和 `-inf` 只被认为接近自己。

3.5 新版功能.

参见:

PEP 485 ——用于测试近似相等的函数

math.**isfinite**(*x*)
如果 *x* 既不是无穷大也不是 NaN，则返回 `True`，否则返回 `False`。（注意 `0.0` 被认为是有限的。）

3.2 新版功能.

math.**isinf**(*x*)
如果 *x* 是正或负无穷大，则返回 `True`，否则返回 `False`。

math.**isnan**(*x*)
如果 *x* 是 NaN（不是数字），则返回 `True`，否则返回 `False`。

math.**ldexp**(*x*, *i*)
返回 `x * (2**i)`。这基本上是函数*`frexp()`* 的反函数。

math.**modf**(*x*)
返回 *x* 的小数和整数部分。两个结果都带有 *x* 的符号并且是浮点数。

math.**trunc**(*x*)
返回*`Real`* 值 *x* 截断为*`Integral`*（通常是整数）。委托给 `x.__trunc__()`。

注意*`frexp()`* 和*`modf()`* 具有与它们的 C 等价函数不同的调用/返回模式：它们采用单个参数并返回一对值，而不是通过‘输出形参’返回它们的第二个返回参数（Python 中没有这样的东西）。

对于*`ceil()`*，*`floor()`* 和*`modf()`* 函数，请注意 所有足够大的浮点数都是精确整数。Python 浮点数通常不超过 53 位的精度（与平台 C double 类型相同），在这种情况下，任何浮点 *x* 与 `abs(x) >= 2**52` 必然没有小数位。

9.2.2 幂函数与对数函数

math.**exp**(*x*)
Return `e**x`.

math.**expm1**(*x*)
Return `e**x - 1`. For small floats *x*, the subtraction in `exp(x) - 1` can result in a significant loss of precision; the *`expm1()`* function provides a way to compute this quantity to full precision:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

3.2 新版功能.

math.**log**(*x*[, *base*])
使用一个参数，返回 *x* 的自然对数（底为 *e*）。

使用两个参数，返回给定的 *base* 的对数 *x*，计算为 `log(x)/log(base)`。

math.**log1p**(*x*)
返回 *1+x* 的自然对数（以 *e* 为底）。以对于接近零的 *x* 精确的方式计算结果。

math.**log2**(*x*)
返回 *x* 以 2 为底的对数。这通常比 `log(x, 2)` 更准确。

3.3 新版功能.

参见:

`int.bit_length()` 返回表示二进制整数所需的位数，不包括符号和前导零。

math.**log10**(*x*)
返回 *x* 底为 10 的对数。这通常比 `log(x, 10)` 更准确。

math.**pow**(*x*, *y*)
将返回 `x` 的 `y` 次幂。特殊情况尽可能遵循 C99 标准的附录’F’。特别是，`pow(1.0, x)` 和 `pow(x, 0.0)` 总是返回 `1.0`，即使 `x` 是零或 NaN。如果 `x` 和 `y` 都是有限的，`x` 是负数，`y` 不是整数那么 `pow(x, y)` 是未定义的，并且引发*ValueError*。

与内置的 `**` 运算符不同，*math.pow()* 将其参数转换为*float* 类型。使用 `**` 或内置的*pow()* 函数来计算精确的整数幂。

math.**sqrt**(*x*)
返回 *x* 的平方根。

9.2.3 三角函数

math.**acos**(*x*)
以弧度为单位返回 *x* 的反余弦值。

math.**asin**(*x*)
以弧度为单位返回 *x* 的反正弦值。

math.**atan**(*x*)
以弧度为单位返回 *x* 的反正切值。

math.**atan2**(*y*, *x*)
以弧度为单位返回 `atan(y / x)`。结果是在 `-pi` 和 `pi` 之间。从原点到点 `(x, y)` 的平面矢量使该角度与正 X 轴成正比。*atan2()* 的点的两个输入的符号都是已知的，因此它可以计算角度的正确象限。例如，`atan(1)` 和 `atan2(1, 1)` 都是 `pi/4`，但 `atan2(-1, -1)` 是 `-3*pi/4`。

math.**cos**(*x*)
返回 *x* 弧度的余弦值。

math.**hypot**(*x*, *y*)
返回欧几里德范数，`sqrt(x*x + y*y)`。这是从原点到点 `(x, y)` 的向量长度。

math.**sin**(*x*)
返回 *x* 弧度的正弦值。

math.**tan**(*x*)
返回 *x* 弧度的正切值。

9.2.4 角度转换

math.**degrees**(*x*)
将角度 *x* 从弧度转换为度数。

math.**radians**(*x*)
将角度 *x* 从度数转换为弧度。

9.2.5 双曲函数

双曲函数 是基于双曲线而非圆来对三角函数进行模拟。

math.**acosh**(*x*)
返回 *x* 的反双曲余弦值。

math.**asinh**(*x*)
返回 *x* 的反双曲正弦值。

math.**atanh**(*x*)
返回 *x* 的反双曲正切值。

math.**cosh**(*x*)
返回 *x* 的双曲余弦值。

math.**sinh**(*x*)
返回 *x* 的双曲正弦值。

math.**tanh**(*x*)
返回 *x* 的双曲正切值。

9.2.6 特殊函数

math.**erf**(*x*)
返回 *x* 处的 error function 。

erf() 函数可用于计算传统的统计函数，如 累积标准正态分布

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

3.2 新版功能.

math.**erfc**(*x*)
返回 *x* 处的互补误差函数。互补错误函数 定义为 `1.0 - erf(x)`。它用于 *x* 的大值，从其中减去一个会导致 有效位数损失。

3.2 新版功能.

math.**gamma**(*x*)
返回 *x* 处的 伽马函数 值。

3.2 新版功能.

math.**lgamma**(*x*)
返回 Gamma 函数在 *x* 绝对值的自然对数。

3.2 新版功能.

9.2.7 常量

math.**pi**
The mathematical constant π = 3.141592…, to available precision.

math.**e**
The mathematical constant e = 2.718281…, to available precision.

math.**tau**
The mathematical constant τ = 6.283185…, to available precision. Tau is a circle constant equal to 2π, the ratio of a circle's circumference to its radius. To learn more about Tau, check out Vi Hart's video Pi is (still) Wrong, and start celebrating Tau day by eating twice as much pie!

3.6 新版功能.

math.**inf**
浮点正无穷大。(对于负无穷大，使用 `-math.inf` 。) 相当于 "float('inf')" 的输出。

3.5 新版功能.

math.**nan**
浮点 "非数字" (NaN) 值。相当于 `float('nan')` 的输出。

3.5 新版功能.

CPython implementation detail: *math* 模块主要包含围绕平台 C 数学库函数的简单包装器。特殊情况下的行为在适当情况下遵循 C99 标准的附录 F。当前的实现将引发*ValueError* 用于无效操作，如 `sqrt(-1.0)` 或 `log(0.0)` (其中 C99 附件 F 建议发出无效操作信号或被零除)，和*OverflowError* 用于溢出的结果 (例如，`exp(1000.0)`)。除非一个或多个输入参数是 NaN，否则不会从上述任何函数返回 NaN；在这种情况下，大多数函数将返回一个 NaN，但是 (再次遵循 C99 附件 F) 这个规则有一些例外，例如 `pow(float('nan'), 0.0)` 或 `hypot(float('nan'), float('inf'))` 。

请注意，Python 不会将显式 NaN 与静默 NaN 区分开来，并且显式 NaN 的行为仍未明确。典型的行为是将所有 NaN 视为静默的。

参见:

***cmath* 模块** 这里很多函数的复数版本。

9.3 cmath ——关于复数的数学函数

This module is always available. It provides access to mathematical functions for complex numbers. The functions in this module accept integers, floating-point numbers or complex numbers as arguments. They will also accept any Python object that has either a `__complex__()` or a `__float__()` method: these methods are used to convert the object to a complex or floating-point number, respectively, and the function is then applied to the result of the conversion.

注解: 在具有对于有符号零的硬件和系统级支持的平台上，涉及支割线的函数在支割线的 两侧都是连续的: 零的符号可用来区别支割线的一侧和另一侧。在不支持有符号零的平台上，连续性的规则见下文。

9.3.1 到极坐标和从极坐标的转换

使用 矩形坐标或 笛卡尔坐标在内部存储 Python 复数 z。这完全取决于它的 实部 z.real 和 虚部 z.imag。换句话说:

```
z == z.real + z.imag*1j
```

极坐标提供了另一种复数的表示方法。在极坐标中，一个复数 *z* 由模量 *r* 和相位角 *phi* 来定义。模量 *r* 是从 *z* 到坐标原点的距离，而相位角 *phi* 是以弧度为单位的，逆时针的，从正 X 轴到连接原点和 *z* 的线段间夹角的角度。

下面的函数可用于原生直角坐标与极坐标的相互转换。

cmath.**phase**(*x*)
Return the phase of *x* (also known as the *argument* of *x*), as a float. `phase(x)` is equivalent to `math.atan2(x.imag, x.real)`. The result lies in the range [-π, π], and the branch cut for this operation lies along the negative real axis, continuous from above. On systems with support for signed zeros (which includes most systems in current use), this means that the sign of the result is the same as the sign of `x.imag`, even when `x.imag` is zero:

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

注解: 一个复数 *x* 的模数（绝对值）可以通过内置函数*abs()* 计算。没有单独的*cmath* 模块函数用于这个操作。

cmath.**polar**(*x*)
在极坐标中返回 *x* 的表达方式。返回一个数对 `(r, phi)`，*r* 是 *x* 的模数，*phi* 是 *x* 的相位角。`polar(x)` 相当于 `(abs(x), phase(x))`。

cmath.**rect**(*r*, *phi*)
通过极坐标的 *r* 和 *phi* 返回复数 *x*。相当于 `r * (math.cos(phi) + math.sin(phi)*1j)`。

9.3.2 幂函数与对数函数

cmath.**exp**(*x*)
Return the exponential value `e**x`.

cmath.**log**(*x*[, *base*])
返回给定 *base* 的 *x* 的对数。如果没有给定 *base*，返回 *x* 的自然对数。从 0 到 -∞ 存在一条支割线，沿负实轴之上连续。

cmath.**log10**(*x*)
返回底数为 10 的 *x* 的对数。它具有与*log()* 相同的支割线。

cmath.**sqrt**(*x*)
返回 *x* 的平方根。它具有与*log()* 相同的支割线。

9.3.3 三角函数

cmath.**acos**(*x*)
返回 *x* 的反余弦。这里有两条支割线：一条沿着实轴从 1 向右延伸到 ∞，从下面连续延伸。另外一条沿着实轴从 -1 向左延伸到 -∞，从上面连续延伸。

cmath.**asin**(*x*)
返回 *x* 的反正弦。它与*acos()* 有相同的支割线。

cmath.**atan**(*x*)
返回 *x* 的反正切。它具有两条支割线：一条沿着虚轴从 `1j` 延伸到 ∞`j`，向右持续延伸。另一条是沿着虚轴从 `-1j` 延伸到 -∞`j` ，向左持续延伸。

cmath.**cos**(*x*)
返回 *x* 的余弦。

cmath.**sin**(*x*)
返回 *x* 的正弦。

cmath.**tan**(*x*)
返回 *x* 的正切。

9.3.4 双曲函数

cmath.**acosh**(*x*)
返回 *x* 的反双曲余弦。它有一条支割线沿着实轴从 1 到 -∞ 向左延伸，从上方持续延伸。

cmath.**asinh**(*x*)
返回 *x* 的反双曲正弦。它有两条支割线：一条沿着虚轴从 `1j` 向右持续延伸到 ∞`j`。另一条是沿着虚轴从 `-1j` 向左持续延伸到 -∞`j`。

cmath.**atanh**(*x*)
返回 *x* 的反双曲正切。它有两条支割线：一条是沿着实轴从 `1` 延展到 ∞，从下面持续延展。另一条是沿着实轴从 `-1` 延展到 -∞，从上面持续延展。

cmath.**cosh**(*x*)
返回 *x* 的双曲余弦值。

cmath.**sinh**(*x*)
返回 *x* 的双曲正弦值。

cmath.**tanh**(*x*)
返回 *x* 的双曲正切值。

9.3.5 分类函数

cmath.**isfinite**(*x*)
如果 *x* 的实部和虚部都是有限的，则返回 `True`，否则返回 `False`。

3.2 新版功能.

cmath.**isinf**(*x*)
如果 *x* 的实部或者虚部是无穷大的，则返回 `True`，否则返回 `False`。

cmath.**isnan**(*x*)
如果 *x* 的实部或者虚部是 NaN，则返回 `True` ，否则返回 `False`。

cmath.**isclose**(*a*, *b*, *, *rel_tol=1e-09*, *abs_tol=0.0*)
若 *a* 和 *b* 的值比较接近则返回 `True`，否则返回 `False`。

根据给定的绝对和相对容差确定两个值是否被认为是接近的。

rel_tol 是相对容差——它是 *a* 和 *b* 之间允许的最大差值，相对于 *a* 或 *b* 的较大绝对值。例如，要设置5%的容差，请传递 `rel_tol=0.05` 。默认容差为 `1e-09`，确保两个值在大约 9 位十进制数字内相同。*rel_tol* 必须大于零。

abs_tol 是最小绝对容差——对于接近零的比较很有用。*abs_tol* 必须至少为零。

如果没有错误发生，结果将是: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)` 。

IEEE 754 特殊值 `NaN` ， `inf` 和 `-inf` 将根据 IEEE 规则处理。具体来说，`NaN` 不被认为接近任何其他值，包括 `NaN` 。`inf` 和 `-inf` 只被认为接近自己。

3.5 新版功能.

参见:

PEP 485 ——用于测试近似相等的函数

9.3.6 常量

cmath.**pi**
数学常数 π ，作为一个浮点数。

cmath.**e**
数学常数 e ，作为一个浮点数。

cmath.**tau**
数学常数 τ ，作为一个浮点数。

3.6 新版功能.

cmath.**inf**
浮点正无穷大。相当于 `float('inf')`。

3.6 新版功能.

cmath.**infj**
具有零实部和正无穷虚部的复数。相当于 `complex(0.0, float('inf'))`。

3.6 新版功能.

cmath.**nan**
浮点“非数字”（NaN）值。相当于 `float('nan')`。

3.6 新版功能.

cmath.**nanj**
具有零实部和 NaN 虚部的复数。相当于 `complex(0.0, float('nan'))`。

3.6 新版功能.

请注意，函数的选择与模块*math* 中的函数选择相似，但不完全相同。拥有两个模块的原因是因为有些用户对复数不感兴趣，甚至根本不知道它们是什么。它们宁愿 `math.sqrt(-1)` 引发异常，也不想返回一个复数。另请注意，被*cmath* 定义的函数始终会返回一个复数，尽管答案可以表示为一个实数（在这种情况下，复数的虚数部分为零）。

关于支割线的注释：它们是沿着给定函数无法连续的曲线。它们是许多复变函数的必要特征。假设您需要使用复变函数进行计算，您将会了解支割线的概念。请参阅几乎所有关于复变函数的（不太基本）的书来获得启发。对于如何正确地基于数值目的来选择支割线的相关信息，一个良好的参考如下：

参见：

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothing’s sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165–211.

9.4 decimal —十进制定点和浮点运算

源码： Lib/decimal.py

---

decimal 模块为快速正确舍入的十进制浮点运算提供支持。它提供了*float* 数据类型以外的几个优点：

- Decimal 类型的“设计是基于考虑人类习惯的浮点数模型，并且因此具有以下最高指导原则——计算机必须提供与人们在学校所学习的算术相一致的算术。”——摘自 decimal 算术规范描述。
- Decimal 数字的表示是精确的。相比之下，`1.1` 和 `2.2` 这样则是不精确的二进制浮点数表示。最终用户通常不希望 `1.1 + 2.2` 的结果会如采用二进制浮点数时那样显示为 `3.3000000000000003`。
- 精确性会延续到算术类操作中。对于 decimal 浮点数，`0.1 + 0.1 + 0.1 - 0.3` 会精确地等于零。而对于二进制浮点数，结果则为 `5.5511151231257827e-017` 。虽然接近于零，但其中的误差将妨碍可靠的相等性检验，并且误差还会不断累积。因此，decimal 更适合具有严格相等不变性要求的会计类应用。
- decimal 模块包含了有效位的概念，使得 `1.30 + 1.20` 是 `2.50` 。保留尾随零以表示有效位。这是货币类应用的习惯表示法。对于乘法，“教科书”方式使用被乘数中的所有数位。例如，`1.3 * 1.2` 给出 `1.56` 而 `1.30 * 1.20` 给出 `1.5600` 。
- 与基于硬件的二进制浮点数不同，decimal 模块具有用户可更改的精度（默认为 28 位），可以与给定问题所需的一样大：

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- 二进制和 decimal 浮点数都是根据已发布的标准实现的。虽然内置浮点类型只公开其功能的一小部分，但 decimal 模块公开了标准的所有必需部分。在需要时，程序员可以完全控制舍入和信号处理。这包括通过使用异常来阻止任何不精确操作来强制执行精确算术的选项。
- decimal 模块旨在支持“无偏差，精确无舍入的十进制算术（有时称为定点数算术）和有舍入的浮点数算术”。——摘自 decimal 算术规范说明。

该模块的设计以三个概念为中心：decimal 数值，算术上下文和信号。

decimal 数值是不可变对象。它由符号，系数和指数位组成。为了保持有效位，系数位不会截去末尾零。decimal 数值也包括特殊值例如 `Infinity` ，`-Infinity` 和 `NaN` 。该标准还区分 `-0` 和 `+0` 。

算术的上下文是指定精度、舍入规则、指数限制、指示操作结果的标志以及确定符号是否被视为异常的陷阱启用器的环境。舍入选项包括*ROUND_CEILING* 、*ROUND_DOWN* 、*ROUND_FLOOR* 、*ROUND_HALF_DOWN*, *ROUND_HALF_EVEN* 、*ROUND_HALF_UP* 、*ROUND_UP* 以及*ROUND_05UP*.

信号是在计算过程中出现的异常条件组。根据应用程序的需要，信号可能会被忽略，被视为信息，或被视为异常。十进制模块中的信号有：*Clamped*、*InvalidOperation*、*DivisionByZero*、*Inexact*、*Rounded*、*Subnormal*、*Overflow*、*Underflow* 以及*FloatOperation*。

对于每个信号，都有一个标志和一个陷阱启动器。遇到信号时，其标志设置为 1，然后，如果陷阱启用器设置为 1，则引发异常。标志是粘性的，因此用户需要在监控计算之前重置它们。

参见:

- IBM 的通用十进制算术规范描述 The General Decimal Arithmetic Specification。

9.4.1 快速入门教程

通常使用 decimal 的方式是先导入该模块，通过*getcontext()* 查看当前上下文，并在必要时为精度、舍入或启用的陷阱设置新值:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

可以基于整数、字符串、浮点数或元组构造 Decimal 实例。基于整数或浮点数构造将执行该整数或浮点值的精确转换。Decimal 数字包括特殊值例如 `NaN` 表示“非数字”，正的和负的 `Infinity` 和 `-0`

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.4142135623730950488016887242')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

如果*FloatOperation* 信号被捕获，构造函数中的小数和浮点数的意外混合或排序比较会引发异常

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
```

(下页继续)

(续上页)

```
>>> Decimal('3.5') == 3.5
True
```

3.3 新版功能.

新 Decimal 的重要性仅由输入的位数决定。上下文精度和舍入仅在算术运算期间发挥作用。

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

如果超出了 C 版本的内部限制，则构造一个 decimal 将引发*InvalidOperation*

```
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

在 3.3 版更改.

Decimal 数字能很好地与 Python 的其余部分交互。以下是一个小小的 decimal 浮点数飞行马戏团：

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

Decimal 也可以使用一些数学函数：

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.4142135623730950488016887242')
>>> Decimal(1).exp()
Decimal('2.7182818284590452353602874713')
>>> Decimal('10').ln()
Decimal('2.3025850929940456840179914547')
>>> Decimal('10').log10()
Decimal('1')
```

quantize() 方法将数字四舍五入为固定指数。此方法对于将结果舍入到固定的位置的货币应用程序非常有用：

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

如上所示，*getcontext()* 函数访问当前上下文并允许更改设置。这种方法满足大多数应用程序的需求。

对于更高级的工作，使用 Context() 构造函数创建备用上下文可能很有用。要使用备用活动，请使用*setcontext()* 函数。

根据标准，*decimal* 模块提供了两个现成的标准上下文*BasicContext* 和*ExtendedContext* 。前者对调试特别有用，因为许多陷阱都已启用：

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

上下文还具有用于监视计算期间遇到的异常情况的信号标志。标志保持设置直到明确清除，因此最好通过使用 clear_flags() 方法清除每组受监控计算之前的标志。：

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

flags 条目显示对 Pi 的有理逼近被舍入（超出上下文精度的数字被抛弃）并且结果是不精确的（一些丢弃的数字不为零）。

使用上下文的 traps 字段中的字典设置单个陷阱：

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

大多数程序仅在程序开始时调整当前上下文一次。并且，在许多应用程序中，数据在循环内单个强制转换为*Decimal* 。通过创建上下文集和小数，程序的大部分操作数据与其他 Python 数字类型没有区别。

9.4.2 Decimal 对象

class decimal.**Decimal**(*value="0"*, *context=None*)

根据 *value* 构造一个新的*Decimal* 对象。

value 可以是整数，字符串，元组，*float* ，或另一个*Decimal* 对象。如果没有给出 *value*，则返回 Decimal('0')。如果 *value* 是一个字符串，它应该在前导和尾随空格字符以及下划线被删除之后符合十进制数字字符串语法：

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

当上面出现 digit 时也允许其他十进制数码。其中包括来自各种其他语言系统的十进制数码（例如阿拉伯-印地语和天城文的数码）以及全宽数码 '\uff10' 到 '\uff19'。

如果 *value* 是一个*tuple* ，它应该有三个组件，一个符号（0 表示正数或 1 表示负数），一个数字的*tuple* 和整数指数。例如，Decimal((0, (1, 4, 1, 4), -3)) 返回 Decimal('1.414')。

如果 *value* 是*float* ，则二进制浮点值无损地转换为其精确的十进制等效值。此转换通常需要 53 位或更多位数的精度。例如，Decimal(float('1.1')) 转换为 "Decimal('1.1000000000000000888178419700125232338905334472656 25')"。

context 精度不会影响存储的位数。这完全由 *value* 中的位数决定。例如，Decimal('3.00000') 记录所有五个零，即使上下文精度只有三。

context 参数的目的是确定 *value* 是格式错误的字符串时该怎么做。如果上下文陷阱*InvalidOperation*，则引发异常；否则，构造函数返回一个新的 Decimal，其值为 NaN。

构造完成后，*Decimal* 对象是不可变的。

在 3.2 版更改: 现在允许构造函数的参数为*float* 实例。

在 3.3 版更改: *float* 参数在设置*FloatOperation* 陷阱时引发异常。默认情况下，陷阱已关闭。

在 3.6 版更改: 允许下划线进行分组，就像代码中的整数和浮点文字一样。

十进制浮点对象与其他内置数值类型共享许多属性，例如*float* 和*int* 。所有常用的数学运算和特殊方法都适用。同样，十进制对象可以复制、pickle、打印、用作字典键、用作集合元素、比较、排序和强制转换为另一种类型（例如*float* 或*int* ）。

算术对十进制对象和算术对整数和浮点数有一些小的差别。当余数运算符 % 应用于 Decimal 对象时，结果的符号是 被除数的符号，而不是除数的符号:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

整数除法运算符 // 的行为类似，返回真商的整数部分（截断为零）而不是它的向下取整，以便保留通常的标识 `x == (x // y) * y + x % y`:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

% 和 // 运算符实现了 remainder 和 divide-integer 操作（分别），如规范中所述。

十进制对象通常不能与浮点数或*fractions.Fraction* 实例在算术运算中结合使用：例如，尝试将*Decimal* 加到*float* ，将引发*TypeError*。但是，可以使用 Python 的比较运算符来比较*Decimal* 实例 x 和另一个数字 y 。这样可以避免在对不同类型的数字进行相等比较时混淆结果。

在 3.2 版更改: 现在完全支持*Decimal* 实例和其他数字类型之间的混合类型比较。

除了标准的数字属性，十进制浮点对象还有许多专门的方法：

adjusted()
在移出系数最右边的数字之后返回调整后的指数，直到只剩下前导数字：`Decimal('321e+5').adjusted()` 返回 7 。用于确定最高有效位相对于小数点的位置。

as_integer_ratio()
返回一对 `(n, d)` 整数，表示给定的*Decimal* 实例作为分数、最简形式项并带有正分母:

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

转换是精确的。在 Infinity 上引发 OverflowError ，在 NaN 上引起 ValueError 。

3.6 新版功能.

as_tuple()
返回一个*named tuple* 表示的数字：`DecimalTuple(sign, digits, exponent)`。

canonical()
返回参数的规范编码。目前，一个*Decimal* 实例的编码始终是规范的，因此该操作返回其参数不变。

compare(*other*, *context=None*)
比较两个 Decimal 实例的值。*compare()* 返回一个 Decimal 实例，如果任一操作数是 NaN ，那么结果是 NaN

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal(*other*, *context=None*)
除了所有 NaN 信号之外，此操作与*compare()* 方法相同。也就是说，如果两个操作数都不是信令 NaN，那么任何静默的 NaN 操作数都被视为信令 NaN。

compare_total(*other*, *context=None*)
使用它们的抽象表示而不是它们的数值来比较两个操作数。类似于*compare()* 方法，但结果给出了一个总排序*Decimal* 实例。两个*Decimal* 实例具有相同的数值但不同的表示形式在此排序中比较不相等：

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

静默和发出信号的 NaN 也包括在总排序中。这个函数的结果是 `Decimal('0')` 如果两个操作数具有相同的表示，或是 `Decimal('-1')` 如果第一个操作数的总顺序低于第二个操作数，或是 `Decimal('1')` 如果第一个操作数在总顺序中高于第二个操作数。有关总排序的详细信息，请参阅规范。

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

compare_total_mag(*other*, *context=None*)
比较两个操作数使用它们的抽象表示而不是它们的值，如*compare_total()*，但忽略每个操作数的符号。`x.compare_total_mag(y)` 相当于 `x.copy_abs().compare_total(y.copy_abs())`。

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

conjugate()
只返回 self，这种方法只符合 Decimal 规范。

copy_abs()
返回参数的绝对值。此操作不受上下文影响并且是静默的：没有更改标志且不执行舍入。

copy_negate()
回到参数的否定。此操作不受上下文影响并且是静默的：没有标志更改且不执行舍入。

copy_sign(*other*, *context=None*)
返回第一个操作数的副本，其符号设置为与第二个操作数的符号相同。例如：

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

exp(*context=None*)
返回给定数字的（自然）指数函数 “e**x“的值。结果使用*ROUND_HALF_EVEN* 舍入模式正确舍入。

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

from_float(*f*)
将浮点数转换为十进制数的类方法。

注意，*Decimal.from_float(0.1)* 与 *Decimal(‘0.1’)* 不同。由于 0.1 在二进制浮点中不能精确表示，因此该值存储为最接近的可表示值，即 *0x1.999999999999ap-4* 。十进制的等效值是 ‘0.1000000000000000055511151231257827021181583404541015625‘。

注解： 从 Python 3.2 开始，*Decimal* 实例也可以直接从*float* 构造。

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

3.1 新版功能.

fma(*other*, *third*, *context=None*)
混合乘法加法。返回 self*other+third ，中间乘积 self*other 没有四舍五入。

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()
如果参数是规范的，则为返回*True*，否则为*False* 。目前，*Decimal* 实例总是规范的，所以这个操作总是返回*True* 。

is_finite()
如果参数是一个有限的数，则返回为*True* ； 如果参数为无穷大或 NaN ，则返回为*False*。

is_infinite()
如果参数为正负无穷大，则返回为*True* ，否则为*False* 。

is_nan()
如果参数为 NaN （无论是否静默），则返回为*True* ，否则为*False* 。

is_normal(*context=None*)
如果参数是一个有限正规数，返回*True*，如果参数是 0、次正规数、无穷大或是 NaN，返回*False*。

is_qnan()
如果参数为静默 NaN，返回*True*，否则返回*False*。

is_signed()
如果参数带有负号，则返回为*True*，否则返回*False*。注意，0 和 NaN 都可带有符号。

is_snan()
如果参数为显式 NaN，则返回*True*，否则返回*False*。

is_subnormal(*context=None*)
如果参数为次正规数，则返回*True*，否则返回*False*。

is_zero()
如果参数是 0（正负皆可），则返回*True*，否则返回*False*。

ln(*context=None*)
返回操作数的自然对数（以 e 为底）。结果是使用*ROUND_HALF_EVEN* 舍入模式正确四舍五入的。

log10(*context=None*)
返回操作数的以十为底的对数。结果是使用*ROUND_HALF_EVEN* 舍入模式正确四舍五入的。

logb(*context=None*)
对于一个非零数，返回其运算数的调整后指数作为一个*Decimal* 实例。如果运算数为零将返回 `Decimal('-Infinity')` 并且产生 the *DivisionByZero* 标志。如果运算数是无限大则返回 `Decimal('Infinity')` 。

logical_and(*other*, *context=None*)
logical_and() 是需要两个 逻辑运算数的逻辑运算（参考逻辑操作数 ）。结果是按位输出的两运算数的“和”。

logical_invert(*context=None*)
logical_invert() 是一个逻辑运算。结果是按位的倒转的运算数。

logical_or(*other*, *context=None*)
logical_or() 是需要两个 *logical operands* 的逻辑运算（请参阅逻辑操作数 ）。结果是两个运算数的按位的 `or` 。

logical_xor(*other*, *context=None*)
logical_xor() 是需要两个 逻辑运算数的逻辑运算（参考逻辑操作数 ）。结果是按位输出的两运算数的异或运算。

max(*other*, *context=None*)
像 `max(self, other)` 一样，除了在返回之前应用上下文舍入规则并且用信号通知或忽略 `NaN` 值（取决于上下文以及它们是发信号还是安静）。

max_mag(*other*, *context=None*)
与*max()* 方法相似，但是操作数使用绝对值进行比较。

min(*other*, *context=None*)
像 `min(self, other)` 一样，除了在返回之前应用上下文舍入规则并且用信号通知或忽略 `NaN` 值（取决于上下文以及它们是发信号还是安静）。

min_mag(*other*, *context=None*)
与*min()* 方法相似，但是操作数使用绝对值进行比较。

next_minus(*context=None*)
返回小于给定操作数的上下文中可表示的最大数字（或者当前线程的上下文中的可表示的最大数字如果没有给定上下文）。

next_plus(*context=None*)
返回大于给定操作数的上下文中可表示的最小数字（或者当前线程的上下文中的可表示的最小数字如果没有给定上下文）。

next_toward(*other*, *context=None*)
如果两运算数不相等，返回在第二个操作数的方向上最接近第一个操作数的数。如果两操作数数值上相等，返回将符号设置为与第二个运算数相同的第一个运算数的拷贝。

normalize(*context=None*)
通过去除尾随的零并将所有结果等于 `Decimal('0')` 的转化为 `Decimal('0e0')` 来标准化数字。用于为等效类的属性生成规范值。比如，`Decimal('32.100')` 和 `Decimal('0.321000e+2')` 都被标准化为相同的值 `Decimal('32.1')`。

number_class(*context=None*)
返回一个字符串描述运算数的 *class* 。返回值是以下十个字符串中的一个。

- `"-Infinity"`，指示操作数为负无穷大。
- `"-Normal"`，指示该操作数是负正常数字。
- `"-Subnormal"`，指示该操作数是负的次正规数。
- `"-Zero"`，指示该操作数是负零。
- `"-Zero"`，指示该操作数是正零。
- `"+Subnormal"`，指示该操作数是正的次正规数。
- `"+Normal"`，指示该操作数是正的正规数。
- `"+Infinity"`，指示该运算数是正无穷。

- "NaN"，指示该运算数是沉寂的 NaN （非数字）。
- "sNaN"，指示该运算数是信号 NaN 。

quantize(*exp*, *rounding=None*, *context=None*)

返回的值等于四舍五入的第一个运算数并且具有第二个操作数的指数。

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

与其他运算不同，如果量化运算后的系数长度大于精度，那么会发出一个*InvalidOperation*信号。这保证了除非有一个错误情况，量化指数恒等于右手运算数的指数。

与其他运算不同，量化永不信号下溢，即使结果不正常且不精确。

如果第二个运算数的指数大于第一个运算数的指数那或许需要四舍五入。在这种情况下，舍入模式由给定 rounding 参数决定，其余的由给定 context 参数决定；如果参数都未给定，使用当前线程上下文的舍入模式。

每当结果的指数大于 Emax 或小于 Etiny 就会返回错误。

radix()

返回 Decimal(10)，即*Decimal*类进行所有算术运算所用的数制（基数）。这是为保持与规范描述的兼容性而加入的。

remainder_near(*other*, *context=None*)

返回 *self* 除以 *other* 的余数。这与 self % other 的区别在于所选择的余数要使其绝对值最小化。更准确地说，返回值为 self - n * other 其中 n 是最接近 self / other 的实际值的整数，并且如果两个整数与实际值的差相等则会选择其中的偶数。

如果结果为零则其符号将为 *self* 的符号。

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)

返回对第一个操作数的数码按第二个操作数所指定的数量进行轮转的结果。第二个操作数必须为 -precision 至 precision 精度范围内的整数。第二个操作数的绝对值给出要轮转的位数。如果第二个操作数为正值则向左轮转；否则向右轮转。如有必要第一个操作数的系数会在左侧填充零以达到 precision 所指定的长度。第一个操作数的符号和指数保持不变。

same_quantum(*other*, *context=None*)

检测自身与 other 是否具有相同的指数或是否均为 NaN。

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

scaleb(*other*, *context=None*)

返回第一个操作数使用第二个操作数对指数进行调整的结果。等价于返回第一个操作数乘以 10**other 的结果。第二个操作数必须为整数。

shift(*other*, *context=None*)

返回第一个操作数的数码按第二个操作数所指定的数量进行移位的结果。第二个操作数必须为 -precision 至 precision 范围内的整数。第二个操作数的绝对值给出要移动的位数。如果第二个操作数为正值则向左移位；否则向右移位。移入系数的数码为零。第一个操作数的符号和指数保持不变。

sqrt(*context=None*)
返回参数的平方根精确到完整精度。

to_eng_string(*context=None*)
转换为字符串，如果需要指数则会使用工程标注法。

工程标注法的指数是 3 的倍数。这会在十进制位的左边保留至多 3 个数码，并可能要求添加一至两个末尾零。

例如，此方法会将 `Decimal('123E+1')` 转换为 `Decimal('1.23E+3')`。

to_integral(*rounding=None*, *context=None*)
与*to_integral_value()* 方法相同。保留 `to_integral` 名称是为了与旧版本兼容。

to_integral_exact(*rounding=None*, *context=None*)
舍入到最接近的整数，发出信号*Inexact* 或者如果发生舍入则相应地发出信号*Rounded*。如果给出 `rounding` 形参则由其确定舍入模式，否则由给定的 `context` 来确定。如果没有给定任何形参则会使用当前上下文的舍入模式。

to_integral_value(*rounding=None*, *context=None*)
舍入到最接近的整数而不发出*Inexact* 或*Rounded* 信号。如果给出 *rounding* 则会应用其所指定的舍入模式；否则使用所提供的 *context* 或当前上下文的舍入方法。

逻辑操作数

`logical_and()`, `logical_invert()`, `logical_or()` 和 `logical_xor()` 方法期望其参数为 *逻辑操作数*。*逻辑操作数*是指数位与符号位均为零的*Decimal* 实例，并且其数字位均为 `0` 或 `1`。

9.4.3 Context 对象

上下文是算术运算所在的环境。它们管理精度、设置舍入规则、确定将哪些信号视为异常，并限制指数的范围。

每个线程都有自己的当前上下文，可使用*getcontext()* 和*setcontext()* 函数来读取或修改：

`decimal`.**getcontext**()
返回活动线程的当前上下文。

`decimal`.**setcontext**(*c*)
将活动线程的当前上下文设为 *c*。

你也可以使用 `with` 语句和*localcontext()* 函数来临时改变活动上下文。

`decimal`.**localcontext**(*ctx=None*)
返回一个上下文管理器，它将在进入 with 语句时将活动线程的当前上下文设为 *ctx* 的一个副本并在退出 with 语句时恢复之前的上下文。如果未指定上下文，则会使用当前上下文的一个副本。

例如，以下代码会将当前 decimal 精度设为 42 位，执行一个运算，然后自动恢复之前的上下文：

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision
```

新的上下文也可使用下述的*Context* 构造器来创建。此外，模块还提供了三种预设的上下文：

class decimal.**BasicContext**
这是由通用十进制算术规范描述所定义的标准上下文。精度设为九。舍入设为*ROUND_HALF_UP*。清除所有旗标。启用所有陷阱（视为异常），但*Inexact*, *Rounded* 和*Subnormal* 除外。

由于启用了许多陷阱，此上下文适用于进行调试。

class decimal.**ExtendedContext**
这是由通用十进制算术规范描述所定义的标准上下文。精度设为九。舍入设为*ROUND_HALF_EVEN*。清除所有旗标。不启用任何陷阱（因此在计算期间不会引发异常）。

由于禁用了陷阱，此上下文适用于希望结果值为 NaN 或 Infinity 而不是引发异常的应用。这允许应用在出现当其他情况下会中止程序的条件时仍能完成运行。

class decimal.**DefaultContext**
此上下文被*Context* 构造器用作新上下文的原型。改变一个字段（例如精度）的效果将是改变*Context* 构造器所创建的新上下文的默认值。

此上下文最适用于多线程环境。在线程开始前改变一个字段具有设置全系统默认值的效果。不推荐在线程开始后改变字段，因为这会要求线程同步避免竞争条件。

在单线程环境中，最好完全不使用此上下文。而是简单地电显式创建上下文，具体如下所述。

默认值为 prec=28, rounding=*ROUND_HALF_EVEN*，并为*Overflow*, *InvalidOperation* 和*DivisionByZero* 启用陷阱。

在已提供的三种上下文之外，还可以使用*Context* 构造器创建新的上下文。

class decimal.**Context**(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*)
创建一个新上下文。如果某个字段未指定或为*None*，则从*DefaultContext* 拷贝默认值。如果 *flags* 字段未指定或为*None*，则清空所有旗标。

prec 为一个 [1, *MAX_PREC*] 范围内的整数，用于设置该上下文中算术运算的精度。

rounding 选项应为*Rounding Modes* 小节中列出的常量之一。

traps 和 *flags* 字段列出要设置的任何信号。通常，新上下文应当只设置 traps 而让 flags 为空。

Emin 和 *Emax* 字段给定指数所允许的外部上限。*Emin* 必须在 [*MIN_EMIN*, 0] 范围内，*Emax* 在 [0, *MAX_EMAX*] 范围内。

capitals 字段为 0 或 1 (默认值)。如果设为 1，指数将附带打印大写的 E；其他情况则将使用小写的 e: Decimal('6.02e+23')。

clamp 字段为 0 (默认值) 或 1。如果设为 1，则*Decimal* 实例的指数 e 的表示范围在此上下文中将严格限制为 Emin - prec + 1 <= e <= Emax - prec + 1。如果 *clamp* 为 0 则将适用较弱的条件: *Decimal* 实例调整后的指数最大值为 Emax。当 *clamp* 为 1 时，一个较大的普通数值将在可能的情况下减小其指数并为其系统添加相应数量的零，以便符合指数值限制；这可以保持数字值但会丢失有效末尾零的信息。例如:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

clamp 值为 1 时即允许与在 IEEE 754 中描述的固定宽度十进制交换格式相兼容。

Context 类定义了几种通用方法以及大量直接在给定上下文中进行算术运算的方法。此外，对于上述的每种*Decimal* 方法（不包括 adjusted() 和 as_tuple() 方法）都有一个相应的*Context* 方法。例如，对于一个*Context* 的实例 C 和*Decimal* 的实例 x，C.exp(x) 就等价于 x.exp(context=C)。每个*Context* 方法都接受一个 Python 整数（即*int* 的实例）在任何接受 Decimal 的实例的地方使用。

clear_flags()
将所有旗标重置为 0。

clear_traps()
将所有陷阱重置为零 0。

3.3 新版功能.

copy()
返回上下文的一个副本。

copy_decimal(*num*)
返回 Decimal 实例 num 的一个副本。

create_decimal(*num*)
基于 *num* 创建一个新 Decimal 实例但使用 *self* 作为上下文。与*Decimal* 构造器不同，该上下文的精度、舍入方法、旗标和陷阱会被应用于转换过程。

此方法很有用处，因为常量往往被给予高于应用所需的精度。另一个好处在于立即执行舍入可以消除超出当前精度的数位所导致的意外效果。在下面的示例中，使用未舍入的输入意味着在总和中添加零会改变结果：

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

此方法实现了 IBM 规格描述中的转换为数字操作。如果参数为字符串，则不允许有开头或末尾的空格或下划线。

create_decimal_from_float(*f*)
基于浮点数 *f* 创建一个新的 Decimal 实例，但会使用 *self* 作为上下文来执行舍入。与*Decimal.from_float()* 类方法不同，上下文的精度、舍入方法、旗标和陷阱会应用到转换中。

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

3.1 新版功能.

Etiny()
返回一个等于 `Emin - prec + 1` 的值即次正规化结果中的最小指数值。当发生向下溢出时，指数会设为*Etiny*。

Etop()
返回一个等于 `Emax - prec + 1` 的值。

使用 decimal 的通常方式是创建*Decimal* 实例然后对其应用算术运算，这些运算发生在活动线程的当前上下文中。一种替代方式则是使用上下文的方法在特定上下文中进行计算。这些方法类似于*Decimal* 类的方法，在此仅简单地重新列出。

abs(*x*)
返回 *x* 的绝对值。

add(*x*, *y*)
返回 *x* 与 *y* 的和。

canonical(*x*)
返回相同的 Decimal 对象 *x*。

compare(*x*, *y*)
对 *x* 与 *y* 进行数值比较。

compare_signal(*x*, *y*)
对两个操作数进行数值比较。

compare_total(*x*, *y*)
对两个操作数使用其抽象表示进行比较。

compare_total_mag(*x*, *y*)
对两个操作数使用其抽象表示进行比较，忽略符号。

copy_abs(*x*)
返回 *x* 的副本，符号设为 0。

copy_negate(*x*)
返回 *x* 的副本，符号取反。

copy_sign(*x*, *y*)
从 *y* 拷贝符号至 *x*。

divide(*x*, *y*)
返回 *x* 除以 *y* 的结果。

divide_int(*x*, *y*)
返回 *x* 除以 *y* 的结果，截短为整数。

divmod(*x*, *y*)
两个数字相除并返回结果的整数部分。

exp(*x*)
返回 *e* ** *x*。

fma(*x*, *y*, *z*)
返回 *x* 乘以 *y* 再加 *z* 的结果。

is_canonical(*x*)
如果 *x* 是规范的则返回 `True`；否则返回 `False`。

is_finite(*x*)
如果 *x* 为有限数则返回 “True“；否则返回 `False`。

is_infinite(*x*)
如果 *x* 是无限的则返回 `True`；否则返回 `False`。

is_nan(*x*)
如果 *x* 是 qNaN 或 sNaN 则返回 `True`；否则返回 `False`。

is_normal(*x*)
如果 *x* 是正规数则返回 `True`；否则返回 `False`。

is_qnan(*x*)
如果 *x* 是静默 NaN 则返回 `True`；否则返回 `False`。

is_signed(*x*)
x 是负数则返回 `True`；否则返回 `False`。

is_snan(*x*)
如果 *x* 是显式 NaN 则返回 `True`；否则返回 `False`。

is_subnormal(*x*)
如果 *x* 是次标准数则返回 `True`；否则返回 `False`。

is_zero(*x*)
如果 *x* 为零则返回 `True`；否则返回 `False`。

ln(*x*)
返回 *x* 的自然对数（以 e 为底）。

log10(*x*)
返回 *x* 的以 10 为底的对数。

logb(*x*)
返回操作数的 MSD 等级的指数。

logical_and(*x*, *y*)
在操作数的每个数码间应用逻辑运算 *and*。

logical_invert(*x*)
反转 *x* 中的所有数位。

logical_or(*x*, *y*)
在操作数的每个数位间应用逻辑运算 *or*。

logical_xor(*x*, *y*)
在操作数的每个数位间应用逻辑运算 *xor*。

max(*x*, *y*)
对两个值执行数字比较并返回其中的最大值。

max_mag(*x*, *y*)
对两个值执行忽略正负号的数字比较。

min(*x*, *y*)
对两个值执行数字比较并返回其中的最小值。

min_mag(*x*, *y*)
对两个值执行忽略正负号的数字比较。

minus(*x*)
对应于 Python 中的单目取负运算符执行取负操作。

multiply(*x*, *y*)
返回 *x* 和 *y* 的积。

next_minus(*x*)
返回小于 *x* 的最大数字表示形式。

next_plus(*x*)
返回大于 *x* 的最小数字表示形式。

next_toward(*x*, *y*)
返回 *x* 趋向于 *y* 的最接近的数字。

normalize(*x*)
将 *x* 改写为最简形式。

number_class(*x*)
返回 *x* 的类的表示。

plus(*x*)
对应于 Python 中的单目前缀取正运算符执行取正操作。此操作将应用上下文精度和舍入，因此它不是标识运算。

power(*x*, *y*, *modulo=None*)
返回 x 的 y 次方，如果给出了模数 modulo 则取其余数。

如为两个参数则计算 x**y。如果 x 为负值则 y 必须为整数。除非 y 为整数且结果为有限值并可在‘precision’位内精确表示否则结果将是不精确的。上下文的舍入模式将被使用。结果在 Python 版中总是会被正确地舍入。

在 3.3 版更改: C 模块计算*power()* 时会使用已正确舍入的*exp()* 和*ln()* 函数。结果是经过良好定义的，但仅限于“几乎总是正确地舍入”。

带有三个参数时，计算 (x**y) % modulo。对于三个参数的形式，参数将会应用以下限制：

- 三个参数必须都是整数
- y 必须是非负数
- x 或 y 至少有一个不为零
- modulo 必须不为零且至多有‘precision’位

来自 Context.power(x, y, modulo) 的结果值等于使用无限精度计算 (x**y) % modulo 所得到的值，但其计算过程更高效。结果的指数为零，无论 x, y 和 modulo 的指数是多少。结果值总是完全精确的。

quantize(*x*, *y*)
返回的值等于 *x* (舍入后)，并且指数为 *y*。

radix()
恰好返回 10，因为这是 Decimal 对象:)

remainder(*x*, *y*)
返回整除所得到的余数。

结果的符号，如果不为零，则与原始除数的符号相同。

remainder_near(*x*, *y*)
返回 x - y * n，其中 *n* 为最接近 x / y 实际值的整数（如结果为 0 则其符号将与 *x* 的符号相同）。

rotate(*x*, *y*)
返回 *x* 翻转 *y* 次的副本。

same_quantum(*x*, *y*)
如果两个操作数具有相同的指数则返回 True。

scaleb(*x*, *y*)
返回第一个操作数对第二个值添加其指数后的结果。

shift(*x*, *y*)
返回 *x* 变换 *y* 次的副本。

sqrt(*x*)
非负数基于上下文精度的平方根。

subtract(*x*, *y*)
返回 *x* 和 *y* 的差。

to_eng_string(*x*)
转换为字符串，如果需要指数则会使用工程标注法。

工程标注法的指数是 3 的倍数。这会在十进制位的左边保留至多 3 个数码，并可能要求添加一至两个末尾零。

to_integral_exact(*x*)
舍入到一个整数。

to_sci_string(*x*)
使用科学计数法将一个数字转换为字符串。

9.4.4 常量

本节中的常量仅与 C 模块相关。它们也被包含在纯 Python 版本以保持兼容性。

	32 位	64 位
decimal.**MAX_PREC**	425000000	999999999999999999
decimal.**MAX_EMAX**	425000000	999999999999999999
decimal.**MIN_EMIN**	-425000000	-999999999999999999
decimal.**MIN_ETINY**	-849999999	-1999999999999999997

decimal.**HAVE_THREADS**
The default value is `True`. If Python is compiled without threads, the C version automatically disables the expensive thread local context machinery. In this case, the value is `False`.

9.4.5 舍入模式

decimal.**ROUND_CEILING**
舍入方向 `Infinity`。

decimal.**ROUND_DOWN**
舍入方向为零。

decimal.**ROUND_FLOOR**
舍入方向为 `-Infinity`。

decimal.**ROUND_HALF_DOWN**
舍入到最接近的数，同样接近则舍入方向为零。

decimal.**ROUND_HALF_EVEN**
舍入到最接近的数，同样接近则舍入到最接近的偶数。

decimal.**ROUND_HALF_UP**
舍入到最接近的数，同样接近则舍入到零的反方向。

decimal.**ROUND_UP**
舍入到零的反方向。

decimal.**ROUND_05UP**
如果最后一位朝零的方向舍入后为 0 或 5 则舍入到零的反方向；否则舍入方向为零。

9.4.6 信号

信号代表在计算期间引发的条件。每个信号对应于一个上下文旗标和一个上下文陷阱启用器。

上下文旗标将在遇到特定条件时被设定。在完成计算之后，将为了获得信息而检测旗标（例如确定计算是否精确）。在检测旗标后，请确保在开始下一次计算之前清除所有旗标。

如果为信号设定了上下文的陷阱启用器，则条件会导致特定的 Python 异常被引发。举例来说，如果设定了*DivisionByZero* 陷阱，则当遇到此条件时就将引发*DivisionByZero* 异常。

class decimal.**Clamped**
修改一个指数以符合表示限制。

通常，限位将在一个指数超出上下文的 Emin 和 Emax 限制时发生。在可能的情况下，会通过给系数添加零来将指数缩减至符合限制。

class decimal.**DecimalException**
其他信号的基类，并且也是*ArithmeticError* 的一个子类。

class decimal.**DivisionByZero**
非无限数被零除的信号。

可在除法、取余队法或对一个数求负数次幂时发生。如果此信号未被陷阱捕获，则返回 Infinity 或 -Infinity 并且由对计算的输入来确定正负符号。

class decimal.**Inexact**
表明发生了舍入且结果是不精确的。

有非零数位在舍入期间被丢弃的信号。舍入结果将被返回。此信号旗标或陷阱被用于检测结果不精确的情况。

class decimal.**InvalidOperation**
执行了一个无效的操作。

表明请求了一个无意义的操作。如未被陷阱捕获则返回 NaN。可能的原因包括:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.**Overflow**
数值的溢出。

表明在发生舍入之后的指数大于 Emax。如果未被陷阱捕获，则结果将取决于舍入模式，或者向下舍入为最大的可表示有限数，或者向上舍入为 Infinity。无论哪种情况，都将引发*Inexact* 和*Rounded* 信号。

class decimal.**Rounded**
发生了舍入，但或许并没有信息丢失。

一旦舍入丢弃了数位就会发出此信号；即使被丢弃的数位是零 (例如将 5.00 舍入为 5.0)。如果未被陷阱捕获，则不经修改地返回结果。此信号用于检测有效位数的丢弃。

class decimal.**Subnormal**
在舍入之前指数低于 Emin。

当操作结果是次标准数（即指数过小）时就会发出此信号。如果未被陷阱捕获，则不经修改过返回结果。

class decimal.**Underflow**
数字向下溢出导致结果舍入到零。

当一个次标准数结果通过舍入转为零时就会发出此信号。同时还将引发*Inexact* 和*Subnormal* 信号。

class decimal.**FloatOperation**
为 float 和 Decimal 的混合启用更严格的语义。

如果信号未被捕获（默认），则在*Decimal* 构造器、*create_decimal()* 和所有比较运算中允许 float 和 Decimal 的混合。转换和比较都是完全精确的。发生的任何混合运算都将通过在上下文旗标中设置*FloatOperation* 来静默地记录。通过*from_float()* 或*create_decimal_from_float()* 进行显式转换则不会设置旗标。

在其他情况下（即信号被捕获），则只静默执行相等性比较和显式转换。所有其他混合运算都将引发*FloatOperation*。

以下表格总结了信号的层级结构:

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 浮点数说明

通过提升精度来缓解舍入误差

使用十进制浮点数可以消除十进制表示错误（即能够完全精确地表示 `0.1` 这样的数）；然而，某些运算在非零数位超出给定的精度时仍然可能导致舍入错误。

舍入错误的影响可能因接近相互抵销的加减运算被放大从而导致损失有效位。Knuth 提供了两个指导性示例，其中出现了精度不足的浮点算术舍入，导致加法的交换律和分配律被打破:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

decimal 模块则可以通过充分地扩展精度来避免有效位的丢失：

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

特殊的值

decimal 模块的数字系统提供了一些特殊的值，包括 `NaN`, `sNaN`, `-Infinity`, `Infinity` 以及两种零值 `+0` 和 `-0`。

无穷大可以使用 `Decimal('Infinity')` 来构建。它们也可以在不捕获*DivisionByZero* 信号捕获时通过除以零来产生。类似地，当不捕获*Overflow* 信号时，也可以通过舍入到超出最大可表示数字限制的方式产生无穷大的结果。

无穷大是有符号的（仿射）并可用于算术运算，它们会被当作极其巨大的不确定数字来处理。例如，无穷大加一个常量结果也将为无穷大。

某些不存在有效结果的运算将会返回 `NaN`，或者如果捕获了*InvalidOperation* 信号则会引发一个异常。例如，`0/0` 会返回 `NaN` 表示结果“不是一个数字”。这样的 `NaN` 是静默产生的，并且在产生之后参与其它计算时总是会得到 `NaN` 的结果。这种行为对于偶而缺少输入的各类计算都很有用处—它允许在将特定结果标记为无效的同时让计算继续运行。

另一种变体形式是 `sNaN`，它在每次运算后会发出信号而不是保持静默。当对于无效结果需要中断计算进行特别处理时，这是一个很有用的返回值。

Python 中比较运算符的行为在涉及 `NaN` 时可能会令人有点惊讶。相等性检测在操作数中有静默型或信号型 `NaN` 时总是会返回*False* (即使是执行 `Decimal('NaN')==Decimal('NaN')`)，而不等性检测总是会返回*True*。当尝试使用 <, <=, > 或 >= 运算符中的任何一个来比较两个 Decimal 值时，如果运算数中有 `NaN` 则将引发*InvalidOperation* 信号，如果此信号未被捕获则将返回*False*。请注意通用十进制算术规范并未规定直接比较行为；这些涉及 `NaN` 的比较规则来自于 IEEE 854 标准 (见第 5.7 节表 3)。要确保严格符合标准，请改用 `compare()` 和 `compare-signal()` 方法。

有符号零值可以由向下溢出的运算产生。它们保留符号是为了让运算结果能以更高的精度传递。由于它们的大小为零，正零和负零会被视为相等，且它们的符号具有信息。

在这两个不相同但却相等的有符号零之外，还存在几种零的不同表示形式，它们的精度不同但值也都相等。这需要一些时间来逐渐适应。对于习惯了标准浮点表示形式的眼睛来说，以下运算返回等于零的值并不是显而易见的：

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 使用线程

getcontext() 函数会为每个线程访问不同的*Context* 对象。具有单独线程上下文意味着线程可以修改上下文 (例如 `getcontext().prec=10`) 而不影响其他线程。

类似的*setcontext()* 会为当前上下文的目标自动赋值。

如果在调用*setcontext()* 之前调用了*getcontext()*，则*getcontext()* 将自动创建一个新的上下文在当前线程中使用。

新的上下文拷贝自一个名为 *DefaultContext* 的原型上下文。要控制默认值以便每个线程在应用运行期间都使用相同的值，可以直接修改 *DefaultContext* 对象。这应当在任何线程启动 之前完成以使得调用*getcontext()* 的线程之间不会产生竞争条件。例如:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9 例程

以下是一些用作工具函数的例程，它们演示了使用*Decimal* 类的各种方式:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg:optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'
```

(下页继续)

(续上页)

```
    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
    3.141592653589793238462643383

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.

    >>> print(exp(Decimal(1)))
    2.718281828459045235360287471
    >>> print(exp(Decimal(2)))
    7.389056098930650227230427461
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (7.38905609893+0j)
```

(下页继续)

(续上页)

```
    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.87758256189037271611628158260
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.47942553860420300027328793521
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
```

(下页继续)

(续上页)

```
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10 Decimal FAQ

Q. 总是输入 `decimal.Decimal('1234.5')` 是否过于笨拙。在使用交互解释器时有没有最小化输入量的方式?

A. 有些用户会将构造器简写为一个字母:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. 在带有两个十进制位的定点数应用中，有些输入值具有许多位，需要被舍入。另一些数则不应具有多余位，需要验证有效性。这种情况应该用什么方法?

A. 用 `quantize()` 方法舍入到固定数量的十进制位。如果设置了 *Inexact* 陷阱，它也适用于验证有效性:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

Q. 当我使用两个有效位的输入时，我要如何在一个应用中保持有效位不变?

A. 某些运算例如与整数相加、相减和相乘将会自动保留固定的小数位数。其他运算，例如相除和非整数相乘则将会改变小数位数，需要再加上 `quantize()` 处理步骤:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
Decimal('325.62')
```

(下页继续)

(续上页)

```
>>> (b / a).quantize(TWOPLACES)      # And quantize division
Decimal('0.03')
```

在开发定点数应用时，更方便的做法是定义处理 quantize() 步骤的函数：

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                           # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. 表示同一个值有许多方式。数字 200, 200.000, 2E2 和 02E+4 的值都相同但有精度不同。是否有办法将它们转换为一个可识别的规范值?

A. normalize() 方法可将所有相同的值映射为统一表示形式：

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. 有些十进制值总是被打印为指数表示形式。是否有办法得到一个非指数表示形式?

A. 对于某些值来说，指数表示形式是表示系数中有效位的唯一办法。例如，将 5.0E+3 表示为 5000 可以让值保持恒定，但是无法显示原本的两位有效数字。

如果一个应用不必关心追踪有效位，则可以很容易地移除指数和末尾的零，丢弃有效位但让值保持不变：

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. 是否有办法将一个普通浮点数转换为*Decimal*?

A. 是的，任何二进制浮点数都可以精确地表示为 Decimal 值，但精确的转换可能需要比直觉设想更高的精度：

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. 在一个复杂的计算中，我怎样才能保证不会得到由精度不足和舍入异常所导致的虚假结果。

A. 使用 decimal 模块可以很容易地检测结果。最好的做法是使用更高的精度和不同的舍入模式重新进行计算。明显不同的结果表明存在精度不足、舍入模式问题、不符合条件的输入或是结果不稳定的算法。

Q. 我发现上下文精度的应用只针对运算结果而不针对输入。在混合使用不同精度的值时有什么需要注意的吗?

A. 是的。原则上所有值都会被视为精确值，在这些值上进行的算术运算也是如此。只有结果会被舍入。对于输入来说其好处是“所输入即所得”。而其缺点则是如果你忘记了输入没有被舍入，结果看起来可能会很奇怪：

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

解决办法是提高精度或使用单目加法运算对输入执行强制舍入：

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

此外，还可以使用*Context.create_decimal()* 方法在创建输入时执行舍入：

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

9.5 fractions —分数

源代码 Lib/fractions.py

---

fractions 模块支持分数运算。

分数实例可以由一对整数，一个分数，或者一个字符串构建而成。

class fractions.**Fraction**(*numerator=0*, *denominator=1*)
class fractions.**Fraction**(*other_fraction*)
class fractions.**Fraction**(*float*)
class fractions.**Fraction**(*decimal*)
class fractions.**Fraction**(*string*)

第一个版本要求 *numerator* 和 *denominator* 是*numbers.Rational* 的实例，并返回一个新的*Fraction* 实例，其值为 numerator/denominator。如果 *denominator* 为 0 将会引发*ZeroDivisionError*。第二个版本要求 *other_fraction* 是*numbers.Rational* 的实例，并返回一个*Fraction* 实例且与传入值相等。下两个版本接受*float* 或*decimal.Decimal* 的实例，并返回一个*Fraction* 实例且与传入值完全相等。请注意由于二进制浮点数通常存在的问题 (参见 tut-fp-issues)，Fraction(1.1) 的参数并不会精确等于 11/10，因此 Fraction(1.1) 也 不会返回用户所期望的 Fraction(11, 10)。(请参阅下文中*limit_denominator()* 方法的文档。) 构造器的最后一个版本接受一个字符串或 unicode 实例。此实例的通常形式为:

```
[sign] numerator ['/' denominator]
```

其中的可选项 sign 可以为 '+' 或 '-' 并且 numerator 和 denominator (如果存在) 是十进制数码的字符串。此外，*float* 构造器所接受的任何表示一个有限值的字符串也都为*Fraction* 构造器所接受。不论哪种形式的输入字符串也都可以带有前缀和/或后缀的空格符。这里是一些示例:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
```

(下页继续)

(续上页)

```
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

Fraction 类继承自抽象基类*numbers.Rational*，并实现了该类的所有方法和操作。*Fraction* 实例是可哈希的，并应当被视为不可变对象。此外，*Fraction* 还具有以下属性和方法：

在 3.2 版更改: *Fraction* 构造器现在接受*float* 和*decimal.Decimal* 实例。

numerator
最简分数形式的分子。

denominator
最简分数形式的分母。

from_float(*flt*)
此类方法可构造一个*Fraction* 来表示 *flt* 的精确值，该参数必须是一个*float*。请注意 `Fraction.from_float(0.3)` 的值并不等于 `Fraction(3, 10)`。

注解： 从 Python 3.2 开始，在构造*Fraction* 实例时可以直接使用*float*。

---

from_decimal(*dec*)
此类方法可构造一个*Fraction* 来表示 *dec* 的精确值，该参数必须是一个*decimal.Decimal* 实例。

注解： 从 Python 3.2 开始，在构造*Fraction* 实例时可以直接使用*decimal.Decimal* 实例。

---

limit_denominator(*max_denominator=1000000*)
找到并返回一个*Fraction* 使得其值最接近 `self` 并且分母不大于 max_denominator。此方法适用于找出给定浮点数的有理数近似值：

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

或是用来恢复被表示为一个浮点数的有理数：

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
```

(下页继续)

(续上页)

```
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()
返回最大的*int* <= self。此方法也可通过*math.floor()* 函数来访问：

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()
返回最小的*int* >= self。此方法也可通过*math.ceil()* 函数来访问。

__round__()
__round__(*ndigits*)
第一个版本返回一个*int* 使得其值最接近 self，位值为二分之一时只对偶数舍入。第二个版本会将 self 舍入到最接近 Fraction(1, 10**ndigits) 的倍数（如果 ndigits 为负值则为逻辑运算），位值为二分之一时同样只对偶数舍入。此方法也可通过*round()* 函数来访问。

fractions.**gcd**(*a*, *b*)
返回整数 *a* 和 *b* 的最大公约数。如果 *a* 或 *b* 之一非零，则 gcd(a, b) 的绝对值是能同时整除 *a* 和 *b* 的最大整数。若 *b* 非零，则 gcd(a,b) 与 *b* 同号；否则返回值与 *a* 同号。gcd(0, 0) 返回 0。

3.5 版后已移除: 由*math.gcd()* 取代.

参见：

***numbers* 模块** 构成数字塔的所有抽象基类。

9.6 random —生成伪随机数

源码： Lib/random.py

该模块实现了各种分布的伪随机数生成器。

对于整数，从范围中有统一的选择。对于序列，存在随机元素的统一选择、用于生成列表的随机排列的函数、以及用于随机抽样而无需替换的函数。

在实数轴上，有计算均匀、正态（高斯）、对数正态、负指数、伽马和贝塔分布的函数。为了生成角度分布，可以使用 von Mises 分布。

几乎所有模块函数都依赖于基本函数*random()* ，它在半开放区间 [0.0,1.0) 内均匀生成随机浮点数。Python 使用 Mersenne Twister 作为核心生成器。它产生 53 位精度浮点数，周期为 2**19937-1 ，其在 C 中的底层实现既快又线程安全。Mersenne Twister 是现存最广泛测试的随机数发生器之一。但是，因为完全确定性，它不适用于所有目的，并且完全不适合加密目的。

这个模块提供的函数实际上是 random.Random 类的隐藏实例的绑定方法。你可以实例化自己的 Random 类实例以获取不共享状态的生成器。

如果你想使用自己设计的不同基础生成器，类 Random 也可以作为子类：在这种情况下，重载 random() 、seed() 、getstate() 以及 setstate() 方法。可选地，新生成器可以提供 getrandbits() 方法——这允许*randrange()* 在任意大的范围内产生选择。

random 模块还提供*SystemRandom* 类，它使用系统函数*os.urandom()* 从操作系统提供的源生成随机数。

警告： 不应将此模块的伪随机生成器用于安全目的。有关安全性或加密用途，请参阅*secrets* 模块。

参见：

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.

Complementary-Multiply-with-Carry recipe 用于兼容的替代随机数发生器，具有长周期和相对简单的更新操作。

9.6.1 簿记功能

random.**seed**(*a=None*, *version=2*)
初始化随机数生成器。

如果 *a* 被省略或为 None，则使用当前系统时间。如果操作系统提供随机源，则使用它们而不是系统时间（有关可用性的详细信息，请参阅*os.urandom()* 函数）。

如果 *a* 是 int 类型，则直接使用。

对于版本 2（默认的），*str*、*bytes* 或*bytearray* 对象转换为*int* 并使用它的所有位。

对于版本 1（用于从旧版本的 Python 再现随机序列），用于*str* 和*bytes* 的算法生成更窄的种子范围。

在 3.2 版更改: 已移至版本 2 方案，该方案使用字符串种子中的所有位。

random.**getstate**()
返回捕获生成器当前内部状态的对象。这个对象可以传递给*setstate()* 来恢复状态。

random.**setstate**(*state*)
state 应该是从之前调用*getstate()* 获得的，并且*setstate()* 将生成器的内部状态恢复到*getstate()* 被调用时的状态。

random.**getrandbits**(*k*)
返回带有 *k* 位随机的 Python 整数。此方法随 MersenneTwister 生成器一起提供，其他一些生成器也可以将其作为 API 的可选部分提供。如果可用，*getrandbits()* 启用*randrange()* 来处理任意大范围。

9.6.2 整数用函数

random.**randrange**(*stop*)
random.**randrange**(*start*, *stop*[, *step*])
从 range(start, stop, step) 返回一个随机选择的元素。这相当于 choice(range(start, stop, step))，但实际上并没有构建一个 range 对象。

位置参数模式匹配*range()*。不应使用关键字参数，因为该函数可能以意外的方式使用它们。

在 3.2 版更改: *randrange()* 在生成均匀分布的值方面更为复杂。以前它使用了像 "int(random()*n)"这样的形式，它可以产生稍微不均匀的分布。

random.**randint**(*a*, *b*)
返回随机整数 *N* 满足 a <= N <= b。相当于 randrange(a, b+1)。

9.6.3 序列用函数

random.**choice**(*seq*)
从非空序列 *seq* 返回一个随机元素。如果 *seq* 为空，则引发 *IndexError*。

random.**choices**(*population*, *weights=None*, ***, *cum_weights=None*, *k=1*)
从 *population* 中选择替换，返回大小为 *k* 的元素列表。如果 *population* 为空，则引发 *IndexError*。

如果指定了 *weight* 序列，则根据相对权重进行选择。或者，如果给出 *cum_weights* 序列，则根据累积权重（可能使用 *itertools.accumulate()* 计算）进行选择。例如，相对权重 "[10, 5, 30, 5]" 相当于累积权重 "[10, 15, 45, 50]"。在内部，相对权重在进行选择之前会转换为累积权重，因此提供累积权重可以节省工作量。

如果既未指定 *weight* 也未指定 *cum_weights*，则以相等的概率进行选择。如果提供了权重序列，则它必须与 *population* 序列的长度相同。一个 *TypeError* 指定了 *weights* 和 *cum_weights*。

weights 或 *cum_weights* 可以使用任何与 *random()* 返回的 *float* 值互操作的数值类型（包括整数，浮点数和分数但不包括十进制小数）。

3.6 新版功能.

random.**shuffle**(*x*[, *random*])
将序列 *x* 随机打乱位置。

可选参数 *random* 是一个 0 参数函数，在 [0.0, 1.0) 中返回随机浮点数；默认情况下，这是函数 *random()*。

要改变一个不可变的序列并返回一个新的打乱列表，请使用 "sample(x, k=len(x))"。

请注意，即使对于小的 `len(x)`，*x* 的排列总数也可以快速增长，大于大多数随机数生成器的周期。这意味着长序列的大多数排列永远不会产生。例如，长度为 2080 的序列是可以在 Mersenne Twister 随机数生成器的周期内拟合的最大序列。

random.**sample**(*population*, *k*)
返回从总体序列或集合中选择的唯一元素的 *k* 长度列表。用于无重复的随机抽样。

返回包含来自总体的元素的新列表，同时保持原始总体不变。结果列表按选择顺序排列，因此所有子切片也将是有效的随机样本。这允许抽奖获奖者（样本）被划分为大奖和第二名获胜者（子切片）。

总体成员不必是 *hashable* 或 unique。如果总体包含重复，则每次出现都是样本中可能的选择。

要从一系列整数中选择样本，请使用 *range()* 对象作为参数。对于从大量人群中采样，这种方法特别快速且节省空间：`sample(range(10000000), k=60)`。

如果样本大小大于总体大小，则引发 *ValueError*。

9.6.4 实值分布

以下函数生成特定的实值分布。如常用数学实践中所使用的那样，函数参数以分布方程中的相应变量命名；大多数这些方程都可以在任何统计学教材中找到。

random.**random**()
返回 [0.0, 1.0) 范围内的下一个随机浮点数。

random.**uniform**(*a*, *b*)
返回一个随机浮点数 *N*，当 `a <= b` 时 `a <= N <= b`，当 `b < a` 时 `b <= N <= a`。

取决于等式 `a + (b-a) * random()` 中的浮点舍入，终点 `b` 可以包括或不包括在该范围内。

random.**triangular**(*low*, *high*, *mode*)
返回一个随机浮点数 *N*，使得 `low <= N <= high` 并在这些边界之间使用指定的 *mode*。*low* 和 *high* 边界默认为零和一。*mode* 参数默认为边界之间的中点，给出对称分布。

random.**betavariate**(*alpha*, *beta*)
Beta 分布。参数的条件是 alpha > 0 和 beta > 0。返回值的范围介于 0 和 1 之间。

random.**expovariate**(*lambd*)
指数分布。*lambd* 是 1.0 除以所需的平均值，它应该是非零的。（该参数本应命名为 "lambda"，但这是 Python 中的保留字。）如果 *lambd* 为正，则返回值的范围为 0 到正无穷大；如果 *lambd* 为负，则返回值从负无穷大到 0。

random.**gammavariate**(*alpha*, *beta*)
Gamma 分布。（不是 gamma 函数！）参数的条件是 alpha > 0 和 beta > 0。

概率分布函数是:

```
          x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu*, *sigma*)
高斯分布。*mu* 是平均值，*sigma* 是标准差。这比下面定义的*normalvariate()* 函数略快。

random.**lognormvariate**(*mu*, *sigma*)
对数正态分布。如果你采用这个分布的自然对数，你将得到一个正态分布，平均值为 *mu* 和标准差为 *sigma* 。*mu* 可以是任何值，*sigma* 必须大于零。

random.**normalvariate**(*mu*, *sigma*)
正态分布。*mu* 是平均值，*sigma* 是标准差。

random.**vonmisesvariate**(*mu*, *kappa*)
冯 · 米塞斯分布。*mu* 是平均角度，以弧度表示，介于 0 和 2**pi* 之间，*kappa* 是浓度参数，必须大于或等于零。如果 *kappa* 等于零，则该分布在 0 到 2**pi* 的范围内减小到均匀的随机角度。

random.**paretovariate**(*alpha*)
帕累托分布。*alpha* 是形状参数。

random.**weibullvariate**(*alpha*, *beta*)
威布尔分布。*alpha* 是比例参数，*beta* 是形状参数。

9.6.5 替代生成器

class random.**SystemRandom**([*seed*])
使用*os.urandom()* 函数的类，用从操作系统提供的源生成随机数。这并非适用于所有系统。也不依赖于软件状态，序列不可重现。因此，*seed()* 方法没有效果而被忽略。*getstate()* 和*setstate()* 方法如果被调用则引发*NotImplementedError*。

9.6.6 关于再现性的说明

有时能够重现伪随机数生成器给出的序列是有用的。通过重新使用种子值，只要多个线程没有运行，相同的序列就可以在两次不同运行之间重现。

大多数随机模块的算法和种子函数都会在 Python 版本中发生变化，但保证两个方面不会改变:

- 如果添加了新的播种方法，则将提供向后兼容的播种机。
- 当兼容的播种机被赋予相同的种子时，生成器的 random() 方法将继续产生相同的序列。

9.6.7 例子和配方

基本示例:

```
>>> random()                            # Random float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                  # Random float:  2.5 <= x < 10.0
3.1800146073117523

>>> expovariate(1 / 5)                  # Interval between arrivals averaging 5␣
↪seconds
5.148957571865031

>>> randrange(10)                       # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)                # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])     # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                       # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)   # Four samples without replacement
[40, 10, 50, 30]
```

模拟:

```
>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck of 52 playing cards
>>> # and determine the proportion of cards with a ten-value
>>> # (a ten, jack, queen, or king).
>>> deck = collections.Counter(tens=16, low_cards=36)
>>> seen = sample(list(deck.elements()), k=20)
>>> seen.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> trial = lambda: choices('HT', cum_weights=(0.60, 1.00), k=7).count('H') >= 5
>>> sum(trial() for i in range(10000)) / 10000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> trial = lambda : 2500 <= sorted(choices(range(10000), k=5))[2]  < 7500
>>> sum(trial() for i in range(10000)) / 10000
0.7958
```

statistical bootstrapping 使用重采样和替换来估计大小为五的样本的均值的置信区间的示例:

```
# http://statistics.about.com/od/Applications/a/Example-Of-Bootstrapping.htm
from statistics import mean
from random import choices

data = 1, 2, 4, 4, 10
means = sorted(mean(choices(data, k=5)) for i in range(20))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[1]:.1f} to {means[-2]:.1f}')
```

使用 重新采样排列测试 来确定统计学显著性或者使用 p-值 来观察药物与安慰剂的作用之间差异的示例:

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

模拟单个服务器队列中的到达时间和服务交付:

```
from random import expovariate, gauss
from statistics import mean, median, stdev

average_arrival_interval = 5.6
average_service_time = 5.0
stdev_service_time = 0.5

num_waiting = 0
arrivals = []
starts = []
arrival = service_end = 0.0
for i in range(20000):
    if arrival <= service_end:
        num_waiting += 1
        arrival += expovariate(1.0 / average_arrival_interval)
        arrivals.append(arrival)
    else:
        num_waiting -= 1
        service_start = service_end if num_waiting else arrival
        service_time = gauss(average_service_time, stdev_service_time)
        service_end = service_start + service_time
        starts.append(service_start)
```

(下页继续)

(续上页)

```
waits = [start - arrival for arrival, start in zip(arrivals, starts)]
print(f'Mean wait: {mean(waits):.1f}.  Stdev wait: {stdev(waits):.1f}.')
print(f'Median wait: {median(waits):.1f}.  Max wait: {max(waits):.1f}.')
```

参见:

Statistics for Hackers Jake Vanderplas 撰写的视频教程，使用一些基本概念进行统计分析，包括模拟、抽样、改组和交叉验证。

Economics Simulation Peter Norvig 编写的市场模拟，显示了该模块提供的许多工具和分布的有效使用（高斯、均匀、样本、beta 变量、选择、三角和随机范围等）。

A Concrete Introduction to Probability (using Python) Peter Norvig 撰写的教程，涵盖了概率论基础知识，如何编写模拟，以及如何使用 Python 进行数据分析。

9.7 statistics —数学统计函数

3.4 新版功能.

源代码: Lib/statistics.py

该模块提供了用于计算数字 (`Real` 值) 数据的数理统计量的函数。

注解: 除非明确注释，这些函数支持*int*, *float*, *decimal.Decimal* 和*fractions.Fraction*。当前不支持同其他类型（不论是否在数字塔中）的行为。混合类型也是未定义且取决于具体实现的。如果你输入由混合类型组成的数据，你应该能够使用*map()* 来确保得到一致的结果，例如 `map(float, input_data)`。

9.7.1 平均值以及对中心位置的评估

这些函数用于计算一个总体或样本的平均值或者典型值。

mean()	数据的算术平均数（“平均数”）。
harmonic_mean()	数据的调和均值
median()	数据的中位数（中间值）
median_low()	数据的低中位数
median_high()	数据的高中位数
median_grouped()	分组数据的中位数，即第 50 个百分点。
mode()	Mode (most common value) of discrete data.

9.7.2 对分散程度的评估

这些函数用于计算总体或样本与典型值或平均值的偏离程度。

pstdev()	数据的总体标准差
pvariance()	数据的总体方差
stdev()	数据的样本标准差
variance()	数据的样本方差

9.7.3 函数细节

注释：这些函数不需要对提供给它们的数据进行排序。但是，为了方便阅读，大多数例子展示的是已排序的序列。

statistics.**mean**(*data*)

返回 *data* 的样本算术平均数，数据可是是一个序列或迭代器。

算术平均数是数据之和与数据点个数的商。通常称作“平均数”，尽管它指示诸多数学平均数之一。它是数据中心位置的度量。

若 *data* 为空，将会引发*StatisticsError*。

一些用法示例：

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

注解： The mean is strongly affected by outliers and is not a robust estimator for central location: the mean is not necessarily a typical example of the data points. For more robust, although less efficient, measures of central location, see *median()* and *mode()*. (In this case, “efficient” refers to statistical efficiency rather than computational efficiency.)

The sample mean gives an unbiased estimate of the true population mean, which means that, taken on average over all the possible samples, `mean(sample)` converges on the true mean of the entire population. If *data* represents the entire population rather than a sample, then `mean(data)` is equivalent to calculating the true population mean μ.

---

statistics.**harmonic_mean**(*data*)

返回 *data* 的调和均值，数据可以是序列或实数值的迭代器。

调和均值, 也叫次相反均值，所有数据的倒数的算术平均数*mean()* 的倒数。比如说，数据 *a* ， *b* ， *c* 的调和均值等于 `3/(1/a + 1/b + 1/c)` 。

The harmonic mean is a type of average, a measure of the central location of the data. It is often appropriate when averaging quantities which are rates or ratios, for example speeds. For example:

假设一名投资者在三家公司各购买了等价值的股票，以 2.5，3 ，10 的 P/E (投资/回报) 率。投资者投资组合的平均市盈率是多少?

```
>>> harmonic_mean([2.5, 3, 10])  # For an equal investment portfolio.
3.6
```

Using the arithmetic mean would give an average of about 5.167, which is too high.

如果 *data* 为空或者任何一个元素的值小于零，会引发*StatisticsError* 。

3.6 新版功能.

statistics.**median**(*data*)

使用常见的“取中间两数平均值”方法，返回数字数据的中位数（中间值）。如果 *data* 为空，则引发*StatisticsError* 。*data* 可以是序列或迭代器。

The median is a robust measure of central location, and is less affected by the presence of outliers in your data. When the number of data points is odd, the middle data point is returned:

```
>>> median([1, 3, 5])
3
```

当数据点的总数为偶数时，中位数将通过对两个中间值求平均进行插值得出：

```
>>> median([1, 3, 5, 7])
4.0
```

这适用于当你的数据是离散的，并且你不介意中位数不是实际数据点的情况。

If your data is ordinal (supports order operations) but not numeric (doesn’t support addition), you should use *median_low()* or *median_high()* instead.

参见:

median_low(), *median_high()*, *median_grouped()*

statistics.**median_low**(*data*)

Return the low median of numeric data. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterator.

低中位数一定是数据集的成员。当数据点总数为奇数时，将返回中间值。当其为偶数时，将返回两个中间值中较小的那个。

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

当你的数据是离散的，并且你希望中位数是一个实际数据点而非插值结果时可以使用低中位数。

statistics.**median_high**(*data*)

Return the high median of data. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterator.

高中位数一定是数据集的成员。当数据点总数为奇数时，将返回中间值。当其为偶数时，将返回两个中间值中较大的那个。

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

当你的数据是离散的，并且你希望中位数是一个实际数据点而非插值结果时可以使用高中位数。

statistics.**median_grouped**(*data*, *interval=1*)

Return the median of grouped continuous data, calculated as the 50th percentile, using interpolation. If *data* is empty, *StatisticsError* is raised. *data* can be a sequence or iterator.

```
>>> median_grouped([52, 52, 53, 54])
52.5
```

在下面的示例中，数据已经过舍入，这样每个值都代表数据分类的中间点，例如 1 是 0.5–1.5 分类的中间点，2 是 1.5–2.5 分类的中间点，3 是 2.5–3.5 的中间点等待。根据给定的数据，中间值应落在 3.5–4.5 分类之内，并可使用插值法来进行估算：

```
>>> median_grouped([1, 2, 2, 3, 4, 4, 4, 4, 4, 5])
3.7
```

可选参数 *interval* 表示分类间隔，默认值为 1。改变分类间隔自然会改变插件结果：

```
>>> median_grouped([1, 3, 3, 5, 7], interval=1)
3.25
>>> median_grouped([1, 3, 3, 5, 7], interval=2)
3.5
```

此函数不会检查数据点之间是否至少相隔 *interval* 的距离。

CPython implementation detail: 在某些情况下，*median_grouped()* 可以会将数据点强制转换为浮点数。此行为在未来有可能会发生改变。

参见:

- “Statistics for the Behavioral Sciences”, Frederick J Gravetter and Larry B Wallnau (8th Edition).
- Calculating the median.
- Gnome Gnumeric 电子表格中的 SSMEDIAN 函数，包括 这篇讨论。

statistics.**mode**(*data*)

Return the most common data point from discrete or nominal *data*. The mode (when it exists) is the most typical value, and is a robust measure of central location.

If *data* is empty, or if there is not exactly one most common value, *StatisticsError* is raised.

`mode` assumes discrete data, and returns a single value. This is the standard treatment of the mode as commonly taught in schools:

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

The mode is unique in that it is the only statistic which also applies to nominal (non-numeric) data:

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

statistics.**pstdev**(*data*, *mu=None*)

返回总体标准差（总体方差的平方根）。请参阅*pvariance()* 了解参数和其他细节。

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.**pvariance**(*data*, *mu=None*)

Return the population variance of *data*, a non-empty iterable of real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *mu* is given, it should be the mean of *data*. If it is missing or `None` (the default), the mean is automatically calculated.

使用此函数可根据所有数值来计算方差。要根据一个样本来估算方差，通常*variance()* 函数是更好的选择。

如果 *data* 为空则会引发*StatisticsError*。

示例：

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

如果你已经计算过数据的平均值，你可以将其作为可选的第二个参数 *mu* 传入以避免重复计算：

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

This function does not attempt to verify that you have passed the actual mean as *mu*. Using arbitrary values for *mu* may lead to invalid or impossible results.

同样也支持使用 Decimal 和 Fraction 值：

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

注解： 当调用时附带完整的总体数据时，这将给出总体方差 σ^2。而当调用时只附带一个样本时，这将给出偏置样本方差 s^2，也被称为带有 N 个自由度的方差。

If you somehow know the true population mean μ, you may use this function to calculate the variance of a sample, giving the known population mean as the second argument. Provided the data points are representative (e.g. independent and identically distributed), the result will be an unbiased estimate of the population variance.

---

statistics.**stdev**(*data*, *xbar=None*)

返回样本标准差（样本方差的平方根）。请参阅*variance()* 了解参数和其他细节。

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.**variance**(*data*, *xbar=None*)

返回包含至少两个实数值的可迭代对象 *data* 的样本方差。方差或称相对于均值的二阶矩，是对数据变化幅度（延展度或分散度）的度量。方差值较大表明数据的散布范围较大；方差值较小表明它紧密聚集于均值附近。

如果给出了可选的第二个参数 *xbar*，它应当是 *data* 的均值。如果该参数省略或为 `None` (默认值)，则会自动进行均值的计算。

当你的数据是总体数据的样本时请使用此函数。要根据整个总体数据来计算方差，请参见*pvariance()*。

如果 *data* 包含的值少于两个则会引发*StatisticsError*。

示例：

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

如果你已经计算过数据的平均值，你可以将其作为可选的第二个参数 *xbar* 传入以避免重复计算：

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

此函数不会试图检查你所传入的 *xbar* 是否为真实的平均值。使用任意值作为 *xbar* 可能导致无效或不可能的结果。

同样也支持使用 Decimal 和 Fraction 值：

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

注解： 这是附带贝塞尔校正的样本方差 s^2，也称为具有 N-1 自由度的方差。假设数据点具有代表性（即为独立且均匀的分布），则结果应当是对总体方差的无偏估计。

如果你通过某种方式知道了真实的总体平均值 μ 则应当调用*pvariance()* 函数并将该值作为 *mu* 形参传入以得到一个样本的方差。

9.7.4 异常

只定义了一个异常：

exception statistics.**StatisticsError**

ValueError 的子类，表示统计相关的异常。

CHAPTER 10

函数式编程模块

本章里描述的模块提供了函数和类，以支持函数式编程风格和在可调用对象上的通用操作。

本章包含以下模块的文档：

10.1 itertools —为高效循环而创建迭代器的函数

本模块实现一系列*iterator*，这些迭代器受到 APL，Haskell 和 SML 的启发。为了适用于 Python，它们都被重新写过。

本模块标准化了一个快速、高效利用内存的核心工具集，这些工具本身或组合都很有用。它们一起形成了“迭代器代数”，这使得在纯 Python 中有可能创建简洁又高效的专用工具。

例如，SML 有一个制表工具：`tabulate(f)`，它可产生一个序列 `f(0), f(1), ...`。在 Python 中可以组合*`map()`* 和*`count()`* 实现：`map(f, count())`。

这些内置工具同时也能很好地与*`operator`* 模块中的高效函数配合使用。例如，我们可以将两个向量的点积映射到乘法运算符：`sum(map(operator.mul, vector1, vector2))`。

无穷迭代器：

迭代器	实参	结果	示例
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, …plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, …重复无限次或 n 次	`repeat(10, 3) --> 10 10 10`

根据最短输入序列长度停止的迭代器：

迭代器	实参	结果	示例
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, ···	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
chain()	p, q, ···	p0, p1, ···plast, q0, q1, ···	`chain('ABC', 'DEF') --> A B C D E F`
chain.from_iterable	iterable – 可迭代对象	p0, p1, ···plast, q0, q1, ···	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ···	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
dropwhile()	pred, seq	seq[n], seq[n+1], ···从 pred 首次真值测试失败开始	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
filterfalse()	pred, seq	seq 中 pred(x) 为假值的元素，x 是 seq 中的元素。	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
groupby()	iterable[, key]	根据 key(v) 值分组的迭代器	
islice()	seq, [start,] stop [, step]	seq[start:stop:step] 中的元素	`islice('ABCDEFG', 2, None) --> C D E F G`
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ···	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
takewhile()	pred, seq	seq[0], seq[1], ···, 直到 pred 真值测试失败	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
tee()	it, n	it1, it2, ···itn 将一个迭代器拆分为 n 个迭代器	
zip_longest()	p, q, ···	(p[0], q[0]), (p[1], q[1]), ···	`zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-`

排列组合迭代器：

迭代器	实参	结果
product()	p, q, ···[repeat=1]	笛卡尔积，相当于嵌套的 for 循环
permutations()	p[, r]	长度 r 元组，所有可能的排列，无重复元素
combinations()	p, r	长度 r 元组，有序，无重复元素
combinations_with_replacement()	p, r	长度 r 元组，有序，元素可重复
`product('ABCD', repeat=2)`		`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`		`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`		`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`		`AA AB AC AD BB BC BD CC CD DD`

10.1.1 Itertool 函数

下列模块函数均创建并返回迭代器。有些迭代器不限制输出流长度，所以它们只应在能截断输出流的函数或循环中使用。

itertools.**accumulate**(*iterable*[, *func*])

创建一个迭代器，返回累加和或其他二元函数的累加结果（通过可选参数 *func* 指定）。如果提供了 *func*，它应是 2 个参数的函数。输入 *iterable* 元素类型应是 *func* 能支持的任意类型。（例如，对于默认的加法操作，元素可以是任一支持加法的类型，包括*Decimal* 或*Fraction*）。如果可迭代对象的输入为空，输出也为空。

大致相当于：

```
def accumulate(iterable, func=operator.add):
    'Return running totals'
    # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
    # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
    it = iter(iterable)
    try:
        total = next(it)
    except StopIteration:
        return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

func 参数有几种用法。它可以被设为*min()* 最终得到一个最小值，或者设为*max()* 最终得到一个最大值，或设为*operator.mul()* 最终得到一个乘积。摊销表可通过累加利息和支付款项得到。给 iterable 设置初始值并只将参数 *func* 设为累加总数可以对一阶 递归关系 建模。

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))     # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))              # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]

# Amortize a 5% loan of 1000 with 4 annual payments of 90
>>> cashflows = [1000, -90, -90, -90, -90]
>>> list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]

# Chaotic recurrence relation https://en.wikipedia.org/wiki/Logistic_map
>>> logistic_map = lambda x, _:  r * x * (1 - x)
>>> r = 3.8
>>> x0 = 0.4
>>> inputs = repeat(x0, 36)     # only the initial value is used
>>> [format(x, '.2f') for x in accumulate(inputs, logistic_map)]
['0.40', '0.91', '0.30', '0.81', '0.60', '0.92', '0.29', '0.79', '0.63',
 '0.88', '0.39', '0.90', '0.33', '0.84', '0.52', '0.95', '0.18', '0.57',
 '0.93', '0.25', '0.71', '0.79', '0.63', '0.88', '0.39', '0.91', '0.32',
 '0.83', '0.54', '0.95', '0.20', '0.60', '0.91', '0.30', '0.80', '0.60']
```

参考一个类似函数*functools.reduce()*，它只返回一个最终累积值。

3.2 新版功能.

在 3.3 版更改: 增加可选参数 *func*。

itertools.**chain**(**iterables*)
创建一个迭代器，它首先返回第一个可迭代对象中所有元素，接着返回下一个可迭代对象中所有元素，直到耗尽所有可迭代对象中的元素。可将多个序列处理为单个序列。大致相当于：

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

classmethod chain.**from_iterable**(*iterable*)
构建类似*chain()* 迭代器的另一个选择。从一个单独的可迭代参数中得到链式输入，该参数是延迟计算的。大致相当于：

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

itertools.**combinations**(*iterable*, *r*)
返回由输入 *iterable* 中元素组成长度为 *r* 的子序列。

组合按照字典序返回。所以如果输入 *iterable* 是有序的，生成的组合元组也是有序的。

即使元素的值相同，不同位置的元素也被认为是不同的。如果元素各自不同，那么每个组合中没有重复元素。

大致相当于：

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

combinations() 的代码可被改写为*permutations()* 过滤后的子序列，（相对于元素在输入中的位置）元素不是有序的。

```
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

当 0 <= r <= n 时，返回项的个数是 n! / r! / (n-r)!；当 r > n 时，返回项个数为 0。

itertools.**combinations_with_replacement**(*iterable*, *r*)

返回由输入 *iterable* 中元素组成的长度为 *r* 的子序列，允许每个元素可重复出现。

组合按照字典序返回。所以如果输入 *iterable* 是有序的，生成的组合元组也是有序的。

不同位置的元素是不同的，即使它们的值相同。因此如果输入中的元素都是不同的话，返回的组合中元素也都会不同。

大致相当于：

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

combinations_with_replacement() 的代码可被改写为 production() 过滤后的子序列，（相对于元素在输入中的位置）元素不是有序的。

```
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

当 n > 0 时，返回项个数为 (n+r-1)! / r! / (n-1)!.

3.1 新版功能.

itertools.**compress**(*data*, *selectors*)

创建一个迭代器，它返回 *data* 中经 *selectors* 真值测试为 True 的元素。迭代器在两者较短的长度处停止。大致相当于：

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in zip(data, selectors) if s)
```

3.1 新版功能.

itertools.**count**(*start=0*, *step=1*)

创建一个迭代器，它从 *start* 值开始，返回均匀间隔的值。常用于*map()* 中的实参来生成连续的数据点。此外，还用于*zip()* 来添加序列号。大致相当于：

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
```

(下页继续)

(续上页)

```
    n = start
    while True:
        yield n
        n += step
```

当对浮点数计数时，替换为乘法代码有时精度会更好，例如：`(start + step * i for i in count())` 。

在 3.1 版更改: 增加参数 *step* ，允许非整型。

itertools.**cycle**(*iterable*)

创建一个迭代器，返回 *iterable* 中所有元素并保存一个副本。当取完 *iterable* 中所有元素，返回副本中的所有元素。无限重复。大致相当于：

```
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
              yield element
```

注意，该函数可能需要相当大的辅助空间（取决于 *iterable* 的长度）。

itertools.**dropwhile**(*predicate*, *iterable*)

创建一个迭代器，如果 *predicate* 为 true，迭代器丢弃这些元素，然后返回其他元素。注意，迭代器在 *predicate* 首次为 false 之前不会产生任何输出，所以可能需要一定长度的启动时间。大致相当于：

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

itertools.**filterfalse**(*predicate*, *iterable*)

创建一个迭代器，只返回 *iterable* 中 *predicate* 为 `False` 的元素。如果 *predicate* 是 `None`，返回真值测试为 false 的元素。大致相当于：

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**groupby**(*iterable*, *key=None*)

创建一个迭代器，返回 *iterable* 中连续的键和组。*key* 是一个计算元素键值函数。如果未指定或为 `None`，*key* 缺省为恒等函数（identity function），返回元素不变。一般来说，*iterable* 需用同一个键值函数预先排序。

groupby() 操作类似于 Unix 中的 `uniq`。当每次 *key* 函数产生的键值改变时，迭代器会分组或生成一个新组（这就是为什么通常需要使用同一个键值函数先对数据进行排序）。这种行为与 SQL 的 GROUP BY 操作不同，SQL 的操作会忽略输入的顺序将相同键值的元素分在同组中。

返回的组本身也是一个迭代器，它与*groupby()* 共享底层的可迭代对象。因为源是共享的，当*groupby()* 对象向后迭代时，前一个组将消失。因此如果稍后还需要返回结果，可保存为列表：

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

groupby() 大致相当于：

```
class groupby:
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def __next__(self):
        while self.currkey == self.tgtkey:
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
        self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey))
    def _grouper(self, tgtkey):
        while self.currkey == tgtkey:
            yield self.currvalue
            try:
                self.currvalue = next(self.it)
            except StopIteration:
                return
            self.currkey = self.keyfunc(self.currvalue)
```

itertools.**islice**(*iterable*, *stop*)
itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

创建一个迭代器，返回从 *iterable* 里选中的元素。如果 *start* 不是 0，跳过 *iterable* 中的元素，直到到达 *start* 这个位置。之后迭代器连续返回元素，除非 *step* 设置的值很高导致被跳过。如果 *stop* 为 `None`，迭代器耗光为止；否则，在指定的位置停止。与普通的切片不同，*islice()* 不支持将 *start*，*stop*，或 *step* 设为负值。可用来从内部数据结构被压平的数据中提取相关字段（例如一个多行报告，它的名称字段出现在每三行上）。大致相当于：

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    s = slice(*args)
```

(下页继续)

(续上页)

```
    start, stop, step = s.start or 0, s.stop or sys.maxsize, s.step or 1
    it = iter(range(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in zip(range(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
        # Consume to *stop*.
        for i, element in zip(range(i + 1, stop), iterable):
            pass
```

如果 *start* 为 None，迭代从 0 开始。如果 *step* 为 None ，步长缺省为 1。

itertools.**permutations**(*iterable*, *r=None*)

连续返回由 *iterable* 元素生成长度为 *r* 的排列。

如果 *r* 未指定或为 None ， *r* 默认设置为 *iterable* 的长度，这种情况下，生成所有全长排列。

排列依字典序发出。因此，如果 *iterable* 是已排序的，排列元组将有序地产出。

即使元素的值相同，不同位置的元素也被认为是不同的。如果元素值都不同，每个排列中的元素值不会重复。

大致相当于：

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

permutations() 的代码也可被改写为*product()* 的子序列，只要将含有重复元素（来自输入中同

一位置的）的项排除。

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

当 `0 <= r <= n`，返回项个数为 `n! / (n-r)!`；当 `r > n`，返回项个数为 0。

itertools.**product**(**iterables*, *repeat=1*)

可迭代对象输入的笛卡儿积。

大致相当于生成器表达式中的嵌套循环。例如，`product(A, B)` 和 `((x,y) for x in A for y in B)` 返回结果一样。

嵌套循环像里程表那样循环变动，每次迭代时将最右侧的元素向后迭代。这种模式形成了一种字典序，因此如果输入的可迭代对象是已排序的，笛卡尔积元组依次序发出。

要计算可迭代对象自身的笛卡尔积，将可选参数 *repeat* 设定为要重复的次数。例如，`product(A, repeat=4)` 和 `product(A, A, A, A)` 是一样的。

该函数大致相当于下面的代码，只不过实际实现方案不会在内存中创建中间结果。

```
def product(*args, repeat=1):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

itertools.**repeat**(*object*[, *times*])

创建一个迭代器，不断重复 *object* 。除非设定参数 *times*，否则将无限重复。可用于*map()* 函数中的参数，被调用函数可得到一个不变参数。也可用于*zip()* 的参数以在元组记录中创建一个不变的部分。

大致相当于：

```
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

repeat 最常见的用途就是在 *map* 或 *zip* 提供一个常量流：

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools.**starmap**(*function*, *iterable*)

创建一个迭代器，使用从可迭代对象中获取的参数来计算该函数。当参数对应的形参已从一个单独可迭代对象组合为元组时（数据已被“预组对”）可用此函数代替*map()*。*map()* 与*starmap()* 之间的区别可以类比 `function(a,b)` 与 `function(*c)` 的区别。大致相当于：

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

itertools.**takewhile**(*predicate*, *iterable*)

创建一个迭代器，只要 predicate 为真就从可迭代对象中返回元素。大致相当于:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

itertools.**tee**(*iterable*, *n=2*)

从一个可迭代对象中返回 *n* 个独立的迭代器。

下面的 Python 代码能帮助解释 *tee* 做了什么（尽管实际的实现更复杂，而且仅使用了一个底层的 FIFO 队列）。

大致相当于:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                try:
                    newval = next(it)   # fetch a new value and
                except StopIteration:
                    return
                for d in deques:        # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

一旦*tee()* 实施了一次分裂，原有的 *iterable* 不应再被使用；否则 tee 对象无法得知 *iterable* 可能已向后迭代。

该迭代工具可能需要相当大的辅助存储空间（这取决于要保存多少临时数据）。通常，如果一个迭代器在另一个迭代器开始之前就要使用大部份或全部数据，使用*list()* 会比*tee()* 更快。

itertools.**zip_longest**(**iterables*, *fillvalue=None*)

创建一个迭代器，从每个可迭代对象中收集元素。如果可迭代对象的长度未对齐，将根据 *fillvalue* 填充缺失值。迭代持续到耗光最长的可迭代对象。大致相当于:

```
class ZipExhausted(Exception):
    pass

def zip_longest(*args, **kwds):
    # zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    fillvalue = kwds.get('fillvalue')
    counter = len(args) - 1
    def sentinel():
        nonlocal counter
```

(下页继续)

(续上页)

```
        if not counter:
            raise ZipExhausted
        counter -= 1
        yield fillvalue
    fillers = repeat(fillvalue)
    iterators = [chain(it, sentinel(), fillers) for it in args]
    try:
        while iterators:
            yield tuple(map(next, iterators))
    except ZipExhausted:
        pass
```

如果其中一个可迭代对象有无限长度，*zip_longest()* 函数应封装在限制调用次数的场景中（例如*islice()* 或*takewhile()*）。除非指定，*fillvalue* 默认为 None 。

10.1.2 Itertools 食谱

本节将展示如何使用现有的 itertools 作为基础构件来创建扩展的工具集。

扩展的工具提供了与底层工具集相同的高性能。保持了超棒的内存利用率，因为一次只处理一个元素，而不是将整个可迭代对象加载到内存。代码量保持得很小，以函数式风格将这些工具连接在一起，有助于消除临时变量。速度依然很快，因为倾向于使用“矢量化”构件来取代解释器开销大的 for 循环和*generator* 。

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def prepend(value, iterator):
    "Prepend a single value in front of an iterator"
    # prepend(1, [2, 3, 4]) -> 1 2 3 4
    return chain([value], iterator)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def tail(n, iterable):
    "Return an iterator over the last n items"
    # tail(3, 'ABCDEFG') --> E F G
    return iter(collections.deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)
```

(下页继续)

(续上页)

```
def all_equal(iterable):
    "Returns True if all the elements are equal to each other"
    g = groupby(iterable)
    return next(g, True) and not next(g, False)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(map(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(map(operator.mul, vec1, vec2))

def flatten(listOfLists):
    "Flatten one level of nesting"
    return chain.from_iterable(listOfLists)

def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return zip_longest(*args, fillvalue=fillvalue)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    num_active = len(iterables)
    nexts = cycle(iter(it).__next__ for it in iterables)
    while num_active:
        try:
            for next in nexts:
```

(下页继续)

(续上页)

```
                yield next()
        except StopIteration:
            # Remove the iterator we just exhausted from the cycle.
            num_active -= 1
            nexts = cycle(islice(nexts, num_active))

def partition(pred, iterable):
    'Use a predicate to partition entries into false entries and true entries'
    # partition(is_odd, range(10)) --> 0 2 4 6 8   and  1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(pred, t1), filter(pred, t2)

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return map(next, map(itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised.

    Converts a call-until-exception interface to an iterator interface.
    Like builtins.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:
        iter_except(functools.partial(heappop, h), IndexError)    # priority queue␣
↪iterator
        iter_except(d.popitem, KeyError)                          # non-blocking dict␣
↪iterator
        iter_except(d.popleft, IndexError)                        # non-blocking deque␣
↪iterator
        iter_except(q.get_nowait, Queue.Empty)                    # loop over a␣
↪producer Queue
```

(下页继续)

(续上页)

```
        iter_except(s.pop, KeyError)                                # non-blocking set
↪iterator

    """
    try:
        if first is not None:
            yield first()            # For database APIs needing an initial cast to
↪db.first()
        while True:
            yield func()
    except exception:
        pass

def first_true(iterable, default=False, pred=None):
    """Returns the first true value in the iterable.

    If no true value is found, returns *default*

    If *pred* is not None, returns the first item
    for which pred(item) is true.

    """
    # first_true([a,b,c], x) --> a or b or c or x
    # first_true([a,b], x, f) --> a if f(a) else b if f(b) else x
    return next(filter(pred, iterable), default)

def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(random.choice(pool) for pool in pools)

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.randrange(n) for i in range(r))
    return tuple(pool[i] for i in indices)

def nth_combination(iterable, r, index):
    'Equivalent to list(combinations(iterable, r))[index]'
    pool = tuple(iterable)
    n = len(pool)
    if r < 0 or r > n:
```

(下页继续)

(续上页)

```
        raise ValueError
    c = 1
    k = min(r, n-r)
    for i in range(1, k+1):
        c = c * (n - k + i) // i
    if index < 0:
        index += c
    if index < 0 or index >= c:
        raise IndexError
    result = []
    while r:
        c, n, r = c*r//n, n-1, r-1
        while index >= c:
            index -= c
            c, n = c*(n-r)//n, n-1
        result.append(pool[-1-n])
    return tuple(result)
```

注意，通过将全局查找替换为局部变量的缺省值，上述配方中有很多可以这样优化。例如，*dotproduct* 配方可以这样写：

```
def dotproduct(vec1, vec2, sum=sum, map=map, mul=operator.mul):
    return sum(map(mul, vec1, vec2))
```

10.2 functools —高阶函数和可调用对象上的操作

源代码: Lib/functools.py

---

functools 模块应用于高阶函数，即——参数或（和）返回值为其他函数的函数。通常来说，此模块的功能适用于所有可调用对象。

functools 模块定义了以下函数:

functools.**cmp_to_key**(*func*)

将 (旧式的) 比较函数转换为新式的*key function*。在类似于*sorted()*，*min()*，*max()*，*heapq.nlargest()*，*heapq.nsmallest()*，*itertools.groupby()* 等函数的 *key* 参数中使用。此函数主要用作将 Python 2 程序转换至新版的转换工具，以保持对比较函数的兼容。

比较函数意为一个可调用对象，该对象接受两个参数并比较它们，结果为小于则返回一个负数，相等则返回零，大于则返回一个正数。key function 则是一个接受一个参数，并返回另一个用以排序的值的可调用对象。

示例:

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order
```

有关排序示例和简要排序教程，请参阅 sortinghowto 。

3.2 新版功能.

@functools.**lru_cache**(*maxsize=128*, *typed=False*)

一个为函数提供缓存功能的装饰器，缓存 *maxsize* 组传入参数，在下次以相同参数调用时直接返回上一次的结果。用以节约高开销或 I/O 函数的调用时间。

由于使用了字典存储缓存，所以该函数的固定参数和关键字参数必须是可哈希的。

如果 *maxsize* 设置为 `None`，LRU 功能将被禁用且缓存数量无上限。*maxsize* 设置为 2 的幂时可获得最佳性能。

如果 *typed* 设置为 true，不同类型的函数参数将被分别缓存。例如，`f(3)` 和 `f(3.0)` 将被视为不同而分别缓存。

为了衡量缓存的有效性以便调整 *maxsize* 形参，被装饰的函数带有一个 `cache_info()` 函数。当调用 `cache_info()` 函数时，返回一个具名元组，包含命中次数 *hits*，未命中次数 *misses*，最大缓存数量 *maxsize* 和当前缓存大小 *currsize*。在多线程环境中，命中数与未命中数是不完全准确的。

该装饰器也提供了一个用于清理/使缓存失效的函数 `cache_clear()`。

原始的未经装饰的函数可以通过 `__wrapped__` 属性访问。它可以用于检查、绕过缓存，或使用不同的缓存再次装饰原始函数。

“最久未使用算法”（LRU）缓存 在“最近的调用是即将到来的调用的最佳预测因子”时性能最好（比如，新闻服务器上最受欢迎的文章倾向于每天更改）。“缓存大小限制”参数保证缓存不会在长时间运行的进程比如说网站服务器上无限制的增加自身的大小。

一般来说，LRU 缓存只在当你想要重用之前计算的结果时使用。因此，用它缓存具有副作用的函数、需要在每次调用时创建不同、易变的对象的函数或者诸如 time () 或 random () 之类的不纯函数是没有意义的。

静态 Web 内容的 LRU 缓存示例:

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'http://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

以下是使用缓存通过 动态规划 计算 斐波那契数列 的例子。

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

3.2 新版功能.

在 3.3 版更改: 添加 *typed* 选项。

@functools.**total_ordering**

给定一个声明一个或多个全比较排序方法的类，这个类装饰器实现剩余的方法。这减轻了指定所有可能的全比较操作的工作。

此类必须包含以下方法之一：__lt__() 、__le__()、__gt__() 或 __ge__()。另外，此类必须支持 __eq__() 方法。

例如

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

注解： 虽然此装饰器使得创建具有良好行为的完全有序类型变得非常容易，但它 确实是以执行速度更缓慢和派生比较方法的堆栈回溯更复杂为代价的。如果性能基准测试表明这是特定应用的瓶颈所在，则改为实现全部六个富比较方法应该会轻松提升速度。

3.2 新版功能.

在 3.4 版更改: 现在已支持从未识别类型的下层比较函数返回 NotImplemented 异常。

functools.**partial**(*func*, **args*, ***keywords*)

返回一个新的部分对象，当被调用时其行为类似于 *func* 附带位置参数 *args* 和关键字参数 *keywords* 被调用。如果为调用提供了更多的参数，它们会被附加到 *args*。如果提供了额外的关键字参数，它们会扩展并重载 *keywords*。大致等价于:

```
def partial(func, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = keywords.copy()
        newkeywords.update(fkeywords)
        return func(*args, *fargs, **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

partial() 会被“冻结了”一部分函数参数和/或关键字的部分函数应用所使用，从而得到一个具有简化签名的新对象。例如，*partial()* 可用来创建一个行为类似于 *int()* 函数的可调用对象，其中 *base* 参数默认为二:

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

class functools.**partialmethod**(*func*, **args*, ***keywords*)

返回一个新的*partialmethod* 描述器，其行为类似*partial* 但它被设计用作方法定义而非直接用作可调用对象。

func 必须是一个*descriptor* 或可调用对象（同属两者的对象例如普通函数会被当作描述器来处理）。

当 *func* 是一个描述器（例如普通 Python 函数，*classmethod()*，*staticmethod()*，abstractmethod() 或其他*partialmethod* 的实例）时，对 __get__ 的调用会被委托给底层的描述器，并会返回一个适当的部分对象 作为结果。

当 *func* 是一个非描述器类可调用对象时，则会动态创建一个适当的绑定方法。当用作方法时其行为类似普通 Python 函数：将会插入 *self* 参数作为第一个位置参数，其位置甚至会处于提供给*partialmethod* 构造器的 *args* 和 *keywords* 之前。

示例:

```
>>> class Cell(object):
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

3.4 新版功能.

functools.**reduce**(*function*, *iterable*[, *initializer*])

将两个参数的 *function* 从左至右累积地应用到 *sequence* 的条目，以便将该序列缩减为单一值。例如，reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) 是计算 ((((1+2)+3)+4)+5) 的值。左边的参数 *x* 是累积值而右边的参数 *y* 则是来自 *sequence* 的更新值。如果存在可选项 *initializer*，它会被放在参与计算的序列的条目之前，并在序列对象为空时作为默认值。如果没有给出 *initializer* 并且 *sequence* 仅包含一个条目，则将返回第一项。

大致相当于:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

@functools.**singledispatch**

将一个函数转换为单分派 *generic function*。

要定义一个泛型函数，应使用 @singledispatch 装饰器进行装饰。请注意分派是作用于第一个参数的类型，要相应地创建你的函数:

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

To add overloaded implementations to the function, use the `register()` attribute of the generic function. It is a decorator, taking a type parameter and decorating a function implementing the operation for that type:

```
>>> @fun.register(int)
... def _(arg, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register(list)
... def _(arg, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

要启用注册 lambda 和现有函数，可以使用函数形式的 `register()` 属性:

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

`register()` 属性将返回启用了装饰器堆栈、封存的未装饰函数，并会为每个变量单独创建单元测试:

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

在调用时，泛型函数会根据第一个参数的类型进行分派:

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
```

(下页继续)

(续上页)

```
Nothing.
>>> fun(1.23)
0.615
```

在没有用于特定类型的已注册实现的情况下，则会使用其方法解析顺序来查找更通用的实现。以 `@singledispatch` 装饰的原始函数将为最基本的 `object` 类型进行注册，这意味着它将在找不到更好的实现时被使用。

要检查泛型函数将为给定类型选择哪个实现，请使用 `dispatch()` 属性:

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

要访问所有忆注册实现，请使用只读的 `registry` 属性:

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
           <class 'decimal.Decimal'>, <class 'list'>,
           <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

3.4 新版功能.

`functools.`**`update_wrapper`**(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

更新一个 *wrapper* 函数以使其类似于 *wrapped* 函数。可选参数为指明原函数的哪些属性要直接被赋值给 wrapper 函数的匹配属性的元组，并且这些 wrapper 函数的属性将使用原函数的对应属性来更新。这些参数的默认值是模块级常量 `WRAPPER_ASSIGNMENTS` (它将被赋值给 wrapper 函数的 `__module__`, `__name__`, `__qualname__`, `__annotations__` 和 `__doc__` 即文档字符串) 以及 `WRAPPER_UPDATES` (它将更新 wrapper 函数的 `__dict__` 即实例字典)。

为了允许出于内省和其他目的访问原始函数（例如绕过*`lru_cache()`* 之类的缓存装饰器），此函数会自动为 wrapper 添加一个指向被包装函数的 `__wrapped__` 属性。

此函数的主要目的是在*decorator* 函数中用来包装被装饰的函数并返回包装器。如果包装器函数未被更新，则被返回函数的元数据将反映包装器定义而不是原始函数定义，这通常没有什么用处。

`update_wrapper()` 可以与函数之外的可调用对象一同使用。在 *assigned* 或 *updated* 中命名的任何属性如果不存在于被包装对象则会被忽略（即该函数将不会尝试在包装器函数上设置它们）。如果包装器函数自身缺少在 *updated* 中命名的任何属性则仍将引发*`AttributeError`*。

3.2 新版功能: 自动添加 `__wrapped__` 属性。

3.2 新版功能: 默认拷贝 `__annotations__` 属性。

在 3.2 版更改: 不存在的属性将不再触发*`AttributeError`*。

在 3.4 版更改: `__wrapped__` 属性现在总是指向被包装的函数，即使该函数定义了 `__wrapped__` 属性。(参见 bpo-17482)

`@functools.`**`wraps`**(*wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

这是一个便捷函数，用于在定义包装器函数时发起调用*`update_wrapper()`* 作为函数装饰器。它等价于 `partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)`。例如:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

如果不使用这个装饰器工厂函数，则 example 函数的名称将变为 `'wrapper'`，并且 `example()` 原本的文档字符串将会丢失。

10.2.1 partial 对象

partial 对象是由*partial()* 创建的可调用对象。它们具有三个只读属性：

partial.**func**
一个可调用对象或函数。对*partial* 对象的调用将被转发给*func* 并附带新的参数和关键字。

partial.**args**
最左边的位置参数将放置在提供给*partial* 对象调用的位置参数之前。

partial.**keywords**
当调用*partial* 对象时将要提供的关键字参数。

partial 对象与 function 对象的类似之处在于它们都是可调用、可弱引用的对象并可拥有属性。但两者也存在一些重要的区别。例如前者不会自动创建*__name__* 和 __doc__ 属性。而且，在类中定义的*partial* 对象的行为类似于静态方法，并且不会在实例属性查找期间转换为绑定方法。

10.3 operator —标准运算符替代函数

源代码: Lib/operator.py

---

operator 模块提供了一套与 Python 的内置运算符对应的高效率函数。例如，`operator.add(x, y)` 与表达式 `x+y` 相同。许多函数名与特殊方法名相同，只是没有双下划线。为了向后兼容性，也保留了许多包含双下划线的函数。为了表述清楚，建议使用没有双下划线的函数。

函数包含的种类有：对象的比较运算、逻辑运算、数学运算以及序列运算。

对象比较函数适用于所有的对象，函数名根据它们对应的比较运算符命名。

operator.**lt**(*a*, *b*)
operator.**le**(*a*, *b*)

operator.**eq**(*a*, *b*)
operator.**ne**(*a*, *b*)
operator.**ge**(*a*, *b*)
operator.**gt**(*a*, *b*)
operator.**__lt__**(*a*, *b*)
operator.**__le__**(*a*, *b*)
operator.**__eq__**(*a*, *b*)
operator.**__ne__**(*a*, *b*)
operator.**__ge__**(*a*, *b*)
operator.**__gt__**(*a*, *b*)
在 *a* 和 *b* 之间进行全比较。具体的，`lt(a, b)` 与 `a < b` 相同，`le(a, b)` 与 `a <= b` 相同，`eq(a, b)` 与 `a == b` 相同，`ne(a, b)` 与 `a != b` 相同，`gt(a, b)` 与 `a > b` 相同，`ge(a, b)``与 ``a >= b` 相同。注意这些函数可以返回任何值，无论它是否可当作布尔值。关于全比较的更多信息请参考 comparisons 。

逻辑运算通常也适用于所有对象，并且支持真值检测、标识检测和布尔运算：

operator.**not_**(*obj*)
operator.**__not__**(*obj*)
返回 `not` *obj* 的结果。（请注意对象实例并没有*__not__()* 方法；只有解释器核心可定义此操作。结果会受 `__bool__()` 和 `__len__()` 方法影响。）

operator.**truth**(*obj*)
如果 *obj* 为真值则返回*True*，否则返回*False*。这等价于使用*bool* 构造器。

operator.**is_**(*a*, *b*)
返回 `a is b`. 测试对象标识。

operator.**is_not**(*a*, *b*)
返回 `a is not b`. 测试对象标识。

数学和按位运算的种类是最多的：

operator.**abs**(*obj*)
operator.**__abs__**(*obj*)
返回 *obj* 的绝对值。

operator.**add**(*a*, *b*)
operator.**__add__**(*a*, *b*)
对于数字 *a* 和 *b*，返回 `a + b`。

operator.**and_**(*a*, *b*)
operator.**__and__**(*a*, *b*)
返回 *x* 和 *y* 按位与

operator.**floordiv**(*a*, *b*)
operator.**__floordiv__**(*a*, *b*)
返回 `a // b`.

operator.**index**(*a*)
operator.**__index__**(*a*)
返回 *a* 转换为整数的结果。等价于 `a.__index__()`。

operator.**inv**(*obj*)
operator.**invert**(*obj*)
operator.**__inv__**(*obj*)
operator.**__invert__**(*obj*)
返回数字 *obj* 按位取反的结果。这等价于 `~obj`。

operator.**lshift**(*a*, *b*)

operator.**__lshift__**(*a*, *b*)
返回 *a* 左移 *b* 位的结果。

operator.**mod**(*a*, *b*)
operator.**__mod__**(*a*, *b*)
返回 `a % b`.

operator.**mul**(*a*, *b*)
operator.**__mul__**(*a*, *b*)
对于数字 *a* 和 *b*，返回 `a * b`。

operator.**matmul**(*a*, *b*)
operator.**__matmul__**(*a*, *b*)
返回 `a @ b`.

3.5 新版功能.

operator.**neg**(*obj*)
operator.**__neg__**(*obj*)
返回 *obj* 的负值 (`-obj`)。

operator.**or_**(*a*, *b*)
operator.**__or__**(*a*, *b*)
返回 *a* 和 *b* 按位或的结果。

operator.**pos**(*obj*)
operator.**__pos__**(*obj*)
返回 *obj* 取正的结果 (`+obj`)。

operator.**pow**(*a*, *b*)
operator.**__pow__**(*a*, *b*)
对于数字 *a* 和 *b*，返回 `a ** b`。

operator.**rshift**(*a*, *b*)
operator.**__rshift__**(*a*, *b*)
返回 *a* 右移 *b* 位的结果。

operator.**sub**(*a*, *b*)
operator.**__sub__**(*a*, *b*)
返回 `a - b`.

operator.**truediv**(*a*, *b*)
operator.**__truediv__**(*a*, *b*)
返回 `a / b` 例如 2/3 将等于.66 而不是 0。这也被称为“真”除法。

operator.**xor**(*a*, *b*)
operator.**__xor__**(*a*, *b*)
返回 *a* 和 *b* 按位异或的结果。

适用于序列的操作（其中一些也适用于映射）包括:

operator.**concat**(*a*, *b*)
operator.**__concat__**(*a*, *b*)
对于序列 *a* 和 *b*，返回 `a + b`。

operator.**contains**(*a*, *b*)
operator.**__contains__**(*a*, *b*)
返回 `b in a` 检测的结果。请注意操作数是反序的。

operator.**countOf**(*a*, *b*)
返回 *b* 在 *a* 中的出现次数。

operator.**delitem**(*a*, *b*)
operator.**__delitem__**(*a*, *b*)
移除 *a* 中索引号为 *b* 的值。

operator.**getitem**(*a*, *b*)
operator.**__getitem__**(*a*, *b*)
返回 *a* 中索引为 *b* 的值。

operator.**indexOf**(*a*, *b*)
返回 *b* 在 *a* 中首次出现所在的索引号。

operator.**setitem**(*a*, *b*, *c*)
operator.**__setitem__**(*a*, *b*, *c*)
将 *a* 中索引号为 *b* 的值设为 *c*。

operator.**length_hint**(*obj*, *default=0*)
返回对象 *o* 的估计长度。首先尝试返回其实际长度，再使用 `object.__length_hint__()` 得出估计值，最后返回默认值。

3.4 新版功能.

operator 模块还定义了一些用于常规属性和条目查找的工具。这些工具适合用来编写快速字段提取器作为*map()*, *sorted()*, *itertools.groupby()* 或其他需要相应函数参数的函数的参数。

operator.**attrgetter**(*attr*)
operator.**attrgetter**(**attrs*)
返回一个可从操作数中获取 *attr* 的可调用对象。如果请求了一个以上的属性，则返回一个属性元组。属性名称还可包含点号。例如：

- 在 `f = attrgetter('name')` 之后，调用 `f(b)` 将返回 `b.name`。
- 在 `f = attrgetter('name', 'date')` 之后，调用 `f(b)` 将返回 `(b.name, b.date)`。
- 在 `f = attrgetter('name.first', 'name.last')` 之后，调用 `f(b)` 将返回 `(b.name.first, b.name.last)`。

等价于：

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.**itemgetter**(*item*)
operator.**itemgetter**(**items*)
返回一个使用操作数的*__getitem__()* 方法从操作数中获取 *item* 的可调用对象。如果指定了多个条目，则返回一个查找值的元组。例如：

- 在 `f = itemgetter(2)` 之后，调用 `f(r)` 将返回 `r[2]`。

- 在 `g = itemgetter(2, 5, 3)` 之后，调用 `g(r)` 将返回 `(r[2], r[5], r[3])`。

等价于:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

传入的条目可以为操作数的*__getitem__()* 所接受的任何类型。字典接受任意可哈希的值。列表、元组和字符串接受 index 或 slice 对象:

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1,3,5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2,None))('ABCDEFG')
'CDEFG'
```

使用*itemgetter()* 从元组的记录中提取特定字段的例子:

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*[, *args...*])

返回一个在操作数上调用 *name* 方法的可调用对象。如果给出额外的参数和/或关键字参数，它们也将被传给该方法。例如:

- 在 `f = methodcaller('name')` 之后，调用 `f(b)` 将返回 `b.name()`。
- 在 `f = methodcaller('name', 'foo', bar=1)` 之后，调用 `f(b)` 将返回 `b.name('foo', bar=1)`。

等价于:

```
def methodcaller(name, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 将运算符映射到函数

以下表格显示了抽象运算是如何对应于 Python 语法中的运算符和*operator* 模块中的函数的。

运算	语法	函数
加法	`a + b`	`add(a, b)`
字符串拼接	`seq1 + seq2`	`concat(seq1, seq2)`
包含测试	`obj in seq`	`contains(seq, obj)`
除法	`a / b`	`truediv(a, b)`
除法	`a // b`	`floordiv(a, b)`
按位与	`a & b`	`and_(a, b)`
按位异或	`a ^ b`	`xor(a, b)`
按位取反	`~ a`	`invert(a)`
按位或	`a | b`	`or_(a, b)`
取幂	`a ** b`	`pow(a, b)`
一致	`a is b`	`is_(a, b)`
一致	`a is not b`	`is_not(a, b)`
索引赋值	`obj[k] = v`	`setitem(obj, k, v)`
索引删除	`del obj[k]`	`delitem(obj, k)`
索引取值	`obj[k]`	`getitem(obj, k)`
左移	`a << b`	`lshift(a, b)`
取模	`a % b`	`mod(a, b)`
乘法	`a * b`	`mul(a, b)`
矩阵乘法	`a @ b`	`matmul(a, b)`
否定（算术）	`- a`	`neg(a)`
否定（逻辑）	`not a`	`not_(a)`
正数	`+ a`	`pos(a)`
右移	`a >> b`	`rshift(a, b)`
切片赋值	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
切片删除	`del seq[i:j]`	`delitem(seq, slice(i, j))`
切片取值	`seq[i:j]`	`getitem(seq, slice(i, j))`
字符串格式化	`s % obj`	`mod(s, obj)`
减法	`a - b`	`sub(a, b)`
真值测试	`obj`	`truth(obj)`
比较	`a < b`	`lt(a, b)`
比较	`a <= b`	`le(a, b)`
相等	`a == b`	`eq(a, b)`
不等	`a != b`	`ne(a, b)`
比较	`a >= b`	`ge(a, b)`
比较	`a > b`	`gt(a, b)`

10.3.2 Inplace Operators

许多运算都有“原地”版本。以下列出的是提供对原地运算符相比通常语法更底层访问的函数，例如*statement* `x += y` 相当于 `x = operator.iadd(x, y)`。换一种方式来讲就是 `z = operator.iadd(x, y)` 等价于语句块 `z = x; z += y`。

在这些例子中，请注意当调用一个原地方法时，运算和赋值是分成两个步骤来执行的。下面列出的原地函数只执行第一步即调用原地方法。第二步赋值则不加处理。

对于不可变的目标例如字符串、数字和元组，更新的值会被计算，但不会被再被赋值给输入变量：

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

For mutable targets such as lists and dictionaries, the inplace method will perform the update, so no subsequent assignment is necessary:

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

operator.**iadd**(*a*, *b*)
operator.**__iadd__**(*a*, *b*)
a = iadd(a, b) 等价于 a += b。

operator.**iand**(*a*, *b*)
operator.**__iand__**(*a*, *b*)
a = iand(a, b) 等价于 a &= b。

operator.**iconcat**(*a*, *b*)
operator.**__iconcat__**(*a*, *b*)
a = iconcat(a, b) 等价于 a += b 其中 *a* 和 *b* 为序列。

operator.**ifloordiv**(*a*, *b*)
operator.**__ifloordiv__**(*a*, *b*)
a = ifloordiv(a, b) 等价于 a //= b.

operator.**ilshift**(*a*, *b*)
operator.**__ilshift__**(*a*, *b*)
a = ilshift(a, b) 等价于 a <<= b。

operator.**imod**(*a*, *b*)
operator.**__imod__**(*a*, *b*)
a = imod(a, b) 等价于 a %= b。

operator.**imul**(*a*, *b*)
operator.**__imul__**(*a*, *b*)
a = imul(a, b) 等价于 a *= b。

operator.**imatmul**(*a*, *b*)
operator.**__imatmul__**(*a*, *b*)
a = imatmul(a, b) 等价于 a @= b。

3.5 新版功能.

operator.**ior**(*a*, *b*)
operator.**__ior__**(*a*, *b*)
a = ior(a, b) 等价于 a |= b。

operator.**ipow**(*a*, *b*)
operator.**__ipow__**(*a*, *b*)
a = ipow(a, b) 等价于 a **= b。

operator.**irshift**(*a*, *b*)
operator.**__irshift__**(*a*, *b*)
a = irshift(a, b) 等价于 a >>= b。

operator.**isub**(*a*, *b*)
operator.**__isub__**(*a*, *b*)
    a = isub(a, b) 等价于 a -= b。

operator.**itruediv**(*a*, *b*)
operator.**__itruediv__**(*a*, *b*)
    a = itruediv(a, b) 等价于 a /= b。

operator.**ixor**(*a*, *b*)
operator.**__ixor__**(*a*, *b*)
    a = ixor(a, b) 等价于 a ^= b。

CHAPTER 11

文件和目录访问

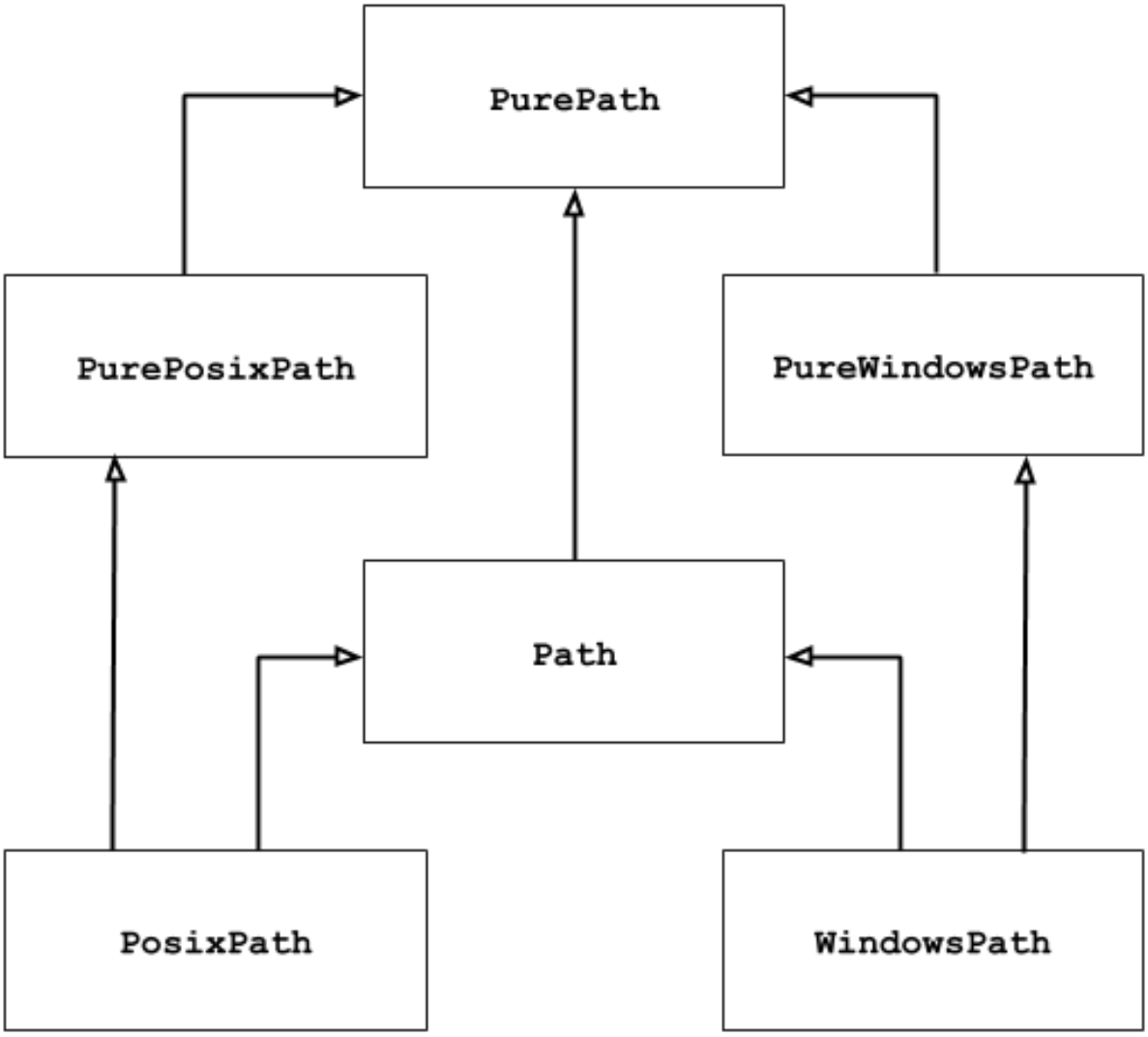
本章中描述的模块处理磁盘文件和目录。例如，有一些模块用于读取文件的属性，以可移植的方式操作路径以及创建临时文件。本章的完整模块列表如下：

11.1 pathlib —面向对象的文件系统路径

3.4 新版功能.

源代码 Lib/pathlib.py

该模块提供表示文件系统路径的类，其语义适用于不同的操作系统。路径类被分为提供纯计算操作而没有 I/O 的纯路径，以及从纯路径继承而来但提供 I/O 操作的具体路径。



如果你以前从未使用过此模块或者不确定在项目中使用哪一个类是正确的，则*Path* 总是你需要的。它在运行代码的平台上实例化为一个具体路径。

在一些用例中纯路径很有用，例如：

1. 如果你想要在 Unix 设备上操作 Windows 路径（或者相反）。你不应在 Unix 上实例化一个*WindowsPath*，但是你可以实例化*PureWindowsPath*。
2. 你只想操作路径但不想实际访问操作系统。在这种情况下，实例化一个纯路径是有用的，因为它们没有任何访问操作系统的操作。

参见：

PEP 428：pathlib 模块–面向对象的的文件系统路径。

参见：

对于底层的路径字符串操作，你也可以使用*os.path* 模块。

11.1.1 基础使用

导入主类:

```
>>> from pathlib import Path
```

列出子目录:

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

列出当前目录树下的所有 Python 源代码文件:

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

在目录树中移动:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

查询路径的属性:

```
>>> q.exists()
True
>>> q.is_dir()
False
```

打开一个文件:

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 纯路径

纯路径对象提供了不实际访问文件系统的路径处理操作。有三种方式来访问这些类，也是不同的风格:

class pathlib.**PurePath**(**pathsegments*)

一个通用的类，代表当前系统的路径风格（实例化为*PurePosixPath*或者*PureWindowsPath*）:

```
>>> PurePath('setup.py')      # Running on a Unix machine
PurePosixPath('setup.py')
```

每一个 *pathsegments* 的元素可能是一个代表路径片段的字符串，一个返回字符串的实现了*os.PathLike*接口的对象，或者另一个路径对象:

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

当 *pathsegments* 为空的时候，假定为当前目录:

```
>>> PurePath()
PurePosixPath('.')
```

当给出一些绝对路径，最后一位将被当作锚（模仿*os.path.join()* 的行为）：

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

但是，在 Windows 路径中，改变本地根目录并不会丢弃之前盘符的设置:

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

假斜线和单独的点都会被消除，但是双点（'..'）不会，以防改变符号链接的含义。

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

（如果你想让 `PurePosixPath('foo/../bar')` 等同于 `PurePosixPath('bar')`，那么 you are too young, too simple, sometimes naive! 如果 `foo` 是一个指向其他其他目录的符号链接，那就出毛病啦。）

纯路径对象实现了*os.PathLike* 接口，允许它们在任何接受此接口的地方使用。

在 3.6 版更改: 添加了*os.PathLike* 接口支持。

class pathlib.**PurePosixPath**(**pathsegments*)

一个*PurePath* 的子类，路径风格不同于 Windows 文件系统:

```
>>> PurePosixPath('/etc')
PurePosixPath('/etc')
```

pathsegments 参数的指定和*PurePath* 相同。

class pathlib.**PureWindowsPath**(**pathsegments*)

PurePath 的一个子类，路径风格为 Windows 文件系统路径:

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
```

pathsegments 参数的指定和*PurePath* 相同。

无论你正运行什么系统，你都可以实例化这些类，因为它们提供的操作不做任何系统调用。

通用性质

路径是不可变并可哈希的。相同风格的路径可以排序与比较。这些性质尊重对应风格的大小写转换语义:

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

不同风格的路径比较得到不等的结果并且无法被排序:

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and 'PurePosixPath
↪'
```

运算符

斜杠 / 操作符有助于创建子路径，就像*os.path.join()* 一样:

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
```

文件对象可用于任何接受*os.PathLike* 接口实现的地方。

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

路径的字符串表示法为它自己原始的文件系统路径（以原生形式，例如在 Windows 下使用反斜杠）。你可以传递给任何需要字符串形式路径的函数。

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

类似地，在路径上调用*bytes* 将原始文件系统路径作为字节对象给出，就像被*os.fsencode()* 编码一样:

```
>>> bytes(p)
b'/etc'
```

注解： 只推荐在 Unix 下调用*bytes*。在 Windows，unicode 形式是文件系统路径的规范表示法。

访问个别部分

为了访问路径独立的部分（组件），使用以下特征属性：

PurePath.**parts**
一个元组，可以访问路径的多个组件：

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

（注意盘符和本地根目录是如何重组的）

方法和特征属性

纯路径提供以下方法和特征属性：

PurePath.**drive**
一个表示驱动器盘符或命名的字符串，如果存在：

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC 分享也被认作驱动器：

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

PurePath.**root**
一个表示（本地或全局）根的字符串，如果存在：

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

UNC 分享一样拥有根：

```
>>> PureWindowsPath('//host/share').root
'\\'
```

PurePath.**anchor**
驱动器和根的联合:

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

PurePath.**parents**
An immutable sequence providing access to the logical ancestors of the path:

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

PurePath.**parent**
此路径的逻辑父路径:

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

你不能超过一个 anchor 或空路径:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

注解: 这是一个单纯的词法操作，因此有以下行为:

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

如果你想要向上移动任意文件系统路径，推荐先使用*Path.resolve()* 来解析符号链接以及消除 `".."` 组件。

PurePath.**name**
一个表示最后路径组件的字符串，排除了驱动器与根目录，如果存在的话:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

UNC 驱动器名不被考虑:

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

PurePath.**suffix**
最后一个组件的文件扩展名，如果存在:

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

PurePath.**suffixes**
路径的文件扩展名列表:

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

PurePath.**stem**
最后一个路径组件，除去后缀:

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.**as_posix**()
返回使用正斜杠（/）的路径字符串:

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.**as_uri**()
将路径表示为 file URL。如果并非绝对路径，抛出*ValueError*。

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = PureWindowsPath('c:/Windows')
```

(下页继续)

(续上页)

```
>>> p.as_uri()
'file:///c:/Windows'
```

PurePath.**is_absolute**()

返回此路径是否为绝对路径。如果路径同时拥有驱动器符与根路径（如果风格允许）则将被认作绝对路径。

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.**is_reserved**()

在*PureWindowsPath*，如果路径是被 Windows 保留的则返回 `True`，否则 `False`。在*PurePosixPath*，总是返回 `False`。

```
>>> PureWindowsPath('nul').is_reserved()
True
>>> PurePosixPath('nul').is_reserved()
False
```

当保留路径上的文件系统被调用，则可能出现玄学失败或者意料之外的效应。

PurePath.**joinpath**(**other*)

调用此方法等同于将每个 *other* 参数中的项目连接在一起:

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

PurePath.**match**(*pattern*)

将此路径与提供的通配符风格的模式匹配。如果匹配成功则返回 `True`，否则返回 `False`。

如果 *pattern* 是相对的，则路径可以是相对路径或绝对路径，并且匹配是从右侧完成的:

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

如果 *pattern* 是绝对的，则路径必须是绝对的，并且路径必须完全匹配:

```
>>> PurePath('/a.py').match('/*.py')
True
>>> PurePath('a/b.py').match('/*.py')
False
```

As with other methods, case-sensitivity is observed:

```
>>> PureWindowsPath('b.py').match('*.PY')
True
```

PurePath.**relative_to**(**other*)
计算此路径相对 *other* 表示路径的版本。如果不可计算，则抛出 ValueError:

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 694, in relative_to
    .format(str(self), str(formatted)))
ValueError: '/etc/passwd' does not start with '/usr'
```

PurePath.**with_name**(*name*)
返回一个新的路径并修改*name*。如果原本路径没有 name，ValueError 被抛出:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

PurePath.**with_suffix**(*suffix*)
返回一个新的路径并修改*suffix*。如果原本的路径没有后缀，新的 *suffix* 则被追加以代替。如果 *suffix* 是空字符串，则原本的后缀被移除:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

11.1.3 具体路径

具体路径是纯路径的子类。除了后者提供的操作之外，它们还提供了对路径对象进行系统调用的方法。有三种方法可以实例化具体路径:

class pathlib.**Path**(**pathsegments*)
一个*PurePath* 的子类，此类以当前系统的路径风格表示路径（实例化为*PosixPath* 或*WindowsPath*）：

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments 参数的指定和*PurePath* 相同。

class pathlib.**PosixPath**(**pathsegments*)
一个*Path* 和*PurePosixPath* 的子类，此类表示一个非 Windows 文件系统的具体路径:

```
>>> PosixPath('/etc')
PosixPath('/etc')
```

pathsegments 参数的指定和*PurePath* 相同。

class pathlib.**WindowsPath**(**pathsegments*)
Path 和*PureWindowsPath* 的子类，从类表示一个 Windows 文件系统的具体路径:

```
>>> WindowsPath('c:/Program Files/')
WindowsPath('c:/Program Files')
```

pathsegments 参数的指定和*PurePath* 相同。

你只能实例化与当前系统风格相同的类（允许系统调用作用于不兼容的路径风格可能在应用程序中导致缺陷或失败）：

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
NotImplementedError: cannot instantiate 'WindowsPath' on your system
```

方法

除纯路径方法外，具体路径还提供以下方法。如果系统调用失败（例如因为路径不存在），其中许多方法都会引发*OSError*:

classmethod Path.**cwd**()
返回一个新的表示当前目录的路径对象（和*os.getcwd()* 返回的相同）：

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

classmethod Path.**home**()
返回一个表示当前用户家目录的新路径对象（和*os.path.expanduser()* 构造含 ~ 路径返回的相同）：

```
>>> Path.home()
PosixPath('/home/antoine')
```

3.5 新版功能.

Path.**stat**()
Return information about this path (similarly to *os.stat()*). The result is looked up at each call to this method.

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

Path.**chmod**(*mode*)
改变文件的模式和权限，和*os.chmod()* 一样：

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

Path.**exists**()
此路径是否指向一个已存在的文件或目录：

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

注解： 如果路径指向一个符号链接，*exists()* 返回此符号链接是否指向存在的文件或目录。

Path.**expanduser**()
返回展开了包含 ~ 和 ~user 的构造，就和*os.path.expanduser()* 一样：

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

3.5 新版功能.

Path.**glob**(*pattern*)
Glob the given *pattern* in the directory represented by this path, yielding all matching files (of any kind):

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
```

(下页继续)

(续上页)

```
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
```

“**”模式表示“此目录以及所有子目录，递归”。换句话说，它启用递归通配:

```
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

注解: 在一个较大的目录树中使用“**”模式可能会消耗非常多的时间。

Path.**group**()
返回拥有此文件的用户组。如果文件的 GID 无法在系统数据库中找到，将抛出*KeyError* 。

Path.**is_dir**()
如果路径指向一个目录（或者一个指向目录的符号链接）则返回 True，如果指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**is_file**()
如果路径指向一个正常的文件（或者一个指向正常文件的符号链接）则返回 True，如果指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**is_symlink**()
如果路径指向符号链接则返回 True，否则 False。

如果路径不存在也返回 False；其他错误（例如权限错误）被传播。

Path.**is_socket**()
如果路径指向一个 Unix socket 文件（或者指向 Unix socket 文件的符号链接）则返回 True，如果指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**is_fifo**()
如果路径指向一个先进先出存储（或者指向先进先出存储的符号链接）则返回 True ，指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**is_block_device**()
如果文件指向一个块设备（或者指向块设备的符号链接）则返回 True，指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**is_char_device**()
如果路径指向一个字符设备（或指向字符设备的符号链接）则返回 True，指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**iterdir**()
当路径指向一个目录时，产生该路径下的对象的路径:

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

Path.**lchmod**(*mode*)
就像*Path.chmod()* 但是如果路径指向符号链接则是修改符号链接的模式，而不是修改符号链接的目标。

Path.**lstat**()
就和*Path.stat()* 一样，但是如果路径指向符号链接，则是返回符号链接而不是目标的信息。

Path.**mkdir**(*mode=0o777*, *parents=False*, *exist_ok=False*)
新建给定路径的目录。如果给出了 *mode* ，它将与当前进程的 umask 值合并来决定文件模式和访问标志。如果路径已经存在，则抛出*FileExistsError*。

如果 *parents* 为 true，任何找不到的父目录都会伴随着此路径被创建；它们会以默认权限被创建，而不考虑 *mode* 设置（模仿 POSIX 的 `mkdir -p` 命令）。

如果 *parents* 为 false（默认），则找不到的父级目录会导致*FileNotFoundError* 被抛出。

如果 *exist_ok* 为 false（默认），则在目标已存在的情况下抛出*FileExistsError*。

如果 *exist_ok* 为 true，则*FileExistsError* 异常将被忽略（和 POSIX `mkdir -p` 命令行为相同），但是只有在最后一个路径组件不是现存的非目录文件时才生效。

在 3.5 版更改: *exist_ok* 形参被加入。

Path.**open**(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)
打开路径指向的文件，就像内置的*open()* 函数所做的一样:

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

Path.**owner**()
返回拥有此文件的用户名。如果文件的 UID 无法在系统数据库中找到，则抛出*KeyError*。

Path.**read_bytes**()
以字节对象的形式返回路径指向的文件的二进制内容:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

3.5 新版功能.

Path.**read_text**(*encoding=None*, *errors=None*)
以字符串形式返回路径指向的文件的解码后文本内容。

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

文件先被打开然后关闭。有和*open()* 一样的可选形参。

3.5 新版功能.

Path.**rename**(*target*)
使用给定的 *target* 将文件重命名。在 Unix 上，如果 *target* 已经存在并且为文件，则只要用户拥有权限，其将被静默地被覆盖。*target* 可以是一个字符串或者另一个路径对象:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
>>> target.open().read()
'some text'
```

Path.**replace**(*target*)
使用给定的 *target* 重命名文件或目录。如果 *target* 指向现存的文件或目录，则将被无条件覆盖。

Path.**resolve**(*strict=False*)
将路径绝对化，解析任何符号链接。返回新的路径对象:

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

“..”组件也将被消除（只有这一种方法这么做）：

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

如果路径不存在并且 *strict* 设为 `True`，则抛出*FileNotFoundError*。如果 *strict* 为 `False`，则路径将被尽可能地解析并且任何剩余部分都会被不检查是否存在地追加。如果在解析路径上发生无限循环，则抛出*RuntimeError*。

3.6 新版功能: The *strict* argument.

Path.**rglob**(*pattern*)
This is like calling *Path.glob()* with “`**`” added in front of the given *pattern*:

```
>>> sorted(Path().rglob("*.py"))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Path.**rmdir**()
移除此目录。此目录必须为空的。

Path.**samefile**(*other_path*)
返回此目录是否指向与可能是字符串或者另一个路径对象的 *other_path* 相同的文件。语义类似于*os.path.samefile()* 与*os.path.samestat()*。

如果两者都以同一原因无法访问，则抛出*OSError*。

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

3.5 新版功能.

Path.**symlink_to**(*target*, *target_is_directory=False*)
将此路径创建为指向 *target* 的符号链接。在 Windows 下，如果链接的目标是一个目录则 *target_is_directory* 必须为 true （默认为 `False`）。在 POSIX 下，*target_is_directory* 的值将被忽略。

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

注解: 参数的顺序（link, target) 和*os.symlink()* 是相反的。

Path.**touch**(*mode=0o666*, *exist_ok=True*)
将给定的路径创建为文件。如果给出了 *mode* 它将与当前进程的 `umask` 值合并以确定文件的模式和访问标志。如果文件已经存在，则当 *exist_ok* 为 true 则函数仍会成功（并且将它的修改事件更新为当前事件），否则抛出*FileExistsError*。

Path.**unlink**()
移除此文件或符号链接。如果路径指向目录，则用*Path.rmdir()* 代替。

Path.**write_bytes**(*data*)
将文件以二进制模式打开，写入 *data* 并关闭:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

一个同名的现存文件将被覆盖。

3.5 新版功能.

Path.**write_text**(*data*, *encoding=None*, *errors=None*)
将文件以文本模式打开，写入 *data* 并关闭:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

3.5 新版功能.

11.2 os.path —常见路径操作

源代码: Lib/posixpath.py (适用 POSIX), Lib/ntpath.py (适用 Windows NT), and Lib/macpath.py (适用 Macintosh)

该模块在路径名上实现了一些有用的功能：如需读取或写入文件，请参见*open()*；有关访问文件系统的信息，请参见*os* 模块。路径参数可以字符串或字节形式传递。我们鼓励应用程序将文件名表示为（Unicode）字符串。不幸的是，某些文件名在 Unix 上可能无法用字符串表示，因此在 Unix 上平台上需要支持任意文件名的应用程序，应使用字节对象来表示路径名。反之亦然，在 Windows 平台上仅使用字节对象，不能表示的所有文件名（以标准 mbcs 编码），因此 Windows 应用程序应使用字符串对象来访问所有文件。

与 unix shell 不同，Python 不执行任何 自动路径扩展。当应用程序需要类似 shell 的路径扩展时，可以显式调用诸如*expanduser()* 和*expandvars()* 之类的函数。（另请参见*glob* 模块。）

参见:

pathlib 模块提供高级路径对象。

注解: 所有这些函数都仅接受字节或字符串对象作为其参数。如果返回路径或文件名，则结果是相同类型的对象。

注解: 由于不同的操作系统具有不同的路径名称约定，因此标准库中有此模块的几个版本。*os.path* 模块始终是适合 Python 运行的操作系统的路径模块，因此可用于本地路径。但是，如果操作的路径 总是以一种不同的格式显示，那么也可以分别导入和使用各个模块。它们都具有相同的界面：

- posixpath 用于 Unix 样式的路径
- ntpath 用于 Windows 路径
- *macpath* 用于旧 MacOS 样式的路径

---

os.path.**abspath**(*path*)
返回路径 *path* 的绝对路径（标准化的）。在大多数平台上，这等同于用 normpath(join(os.getcwd(), path)) 的方式调用*normpath()* 函数。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**basename**(*path*)
返回路径 *path* 的基本名称。这是将 *path* 传入函数*split()* 之后，返回的一对值中的第二个元素。请注意，此函数的结果与 Unix **basename** 程序不同。**basename** 在 '/foo/bar/' 上返回 'bar'，而*basename()* 函数返回一个空字符串 ('')。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**commonpath**(*paths*)
Return the longest common sub-path of each pathname in the sequence *paths*. Raise ValueError if *paths* contains both absolute and relative pathnames, or if *paths* is empty. Unlike *commonprefix()*, this returns a valid path.

Availability: Unix, Windows

3.5 新版功能.

在 3.6 版更改: 接受一个类路径对象 序列。

os.path.**commonprefix**(*list*)
接受包含多个路径的 列表，返回所有路径的最长公共前缀（逐字符比较）。如果 列表为空，则返回空字符串 (''')。

注解: 此函数是逐字符比较，因此可能返回无效路径。要获取有效路径，参见*commonpath()*。

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

在 3.6 版更改: 接受一个*path-like object*。

os.path.**dirname**(*path*)
返回路径 *path* 的目录名称。这是将 *path* 传入函数*split()* 之后，返回的一对值中的第一个元素。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**exists**(*path*)
如果 *path* 指向一个已存在的路径或已打开的文件描述符，返回 True。对于失效的符号链接，返回 False。在某些平台上，如果使用*os.stat()* 查询到目标文件没有执行权限，即使 *path* 确实存在，本函数也可能返回 False。

在 3.3 版更改: *path* 现在可以是一个整数：如果该整数是一个已打开的文件描述符，返回 True，否则返回 False。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**lexists**(*path*)
如果 *path* 指向一个已存在的路径，返回 True。对于失效的符号链接，返回 False。在缺失*os.lstat()* 的平台上等同于*exists()*。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**expanduser**(*path*)
在 Unix 和 Windows 上，将参数中开头部分的 ~ 或 ~user 替换为当前 用户的家目录并返回。

在 Unix 上，开头的 ~ 会被环境变量 HOME 代替，如果变量未设置，则通过内置模块*pwd* 在 password 目录中查找当前用户的主目录。以 ~user 开头则直接在 password 目录中查找。

在 Windows 上，如果设置了 HOME 和 USERPROFILE 则将使用它们，否则将使用 HOMEPATH 和 HOMEDRIVE 的组合。原本的 ~user 处理方式为从上述方法所生成的用户路径中截去最后一级目录。

如果展开路径失败，或者路径不是以波浪号开头，则路径将保持不变。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**expandvars**(*path*)
输入带有环境变量的路径作为参数，返回展开变量以后的路径。$name 或 ${name} 形式的子字符串被环境变量 *name* 的值替换。格式错误的变量名称和对不存在变量的引用保持不变。

在 Windows 上，除了 `$name` 和 `${name}` 外，还可以展开 `%name%`。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**getatime**(*path*)
Return the time of last access of *path*. The return value is a number giving the number of seconds since the epoch (see the *time* module). Raise *OSError* if the file does not exist or is inaccessible.

If *os.stat_float_times()* returns `True`, the result is a floating point number.

os.path.**getmtime**(*path*)
Return the time of last modification of *path*. The return value is a number giving the number of seconds since the epoch (see the *time* module). Raise *OSError* if the file does not exist or is inaccessible.

If *os.stat_float_times()* returns `True`, the result is a floating point number.

在 3.6 版更改: 接受一个*path-like object*。

os.path.**getctime**(*path*)
返回 *path* 在系统中的 ctime，在有些系统（比如 Unix）上，它是元数据的最后修改时间，其他系统（比如 Windows）上，它是 *path* 的创建时间。返回值是一个数，为纪元秒数（参见*time* 模块）。如果该文件不存在或不可访问，则抛出*OSError* 异常。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**getsize**(*path*)
返回 *path* 的大小，以字节为单位。如果该文件不存在或不可访问，则抛出*OSError* 异常。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**isabs**(*path*)
如果 *path* 是一个绝对路径，则返回 `True`。在 Unix 上，它就是以斜杠开头，而在 Windows 上，它可以是去掉驱动器号后以斜杠（或反斜杠）开头。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**isfile**(*path*)
如果 *path* 是现有的 常规文件，则返回 `True`。本方法会跟踪符号链接，因此，对于同一路径，*islink()* 和*isfile()* 都可能为 `True`。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**isdir**(*path*)
如果 *path* 是现有的 目录，则返回 `True`。本方法会跟踪符号链接，因此，对于同一路径，*islink()* 和*isdir()* 都可能为 `True`。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**islink**(*path*)
如果 *path* 指向的现有 目录条目是一个符号链接，则返回 `True`。如果 Python 运行时不支持符号链接，则总是返回 `False`。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**ismount**(*path*)
Return `True` if pathname *path* is a *mount point*: a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*'s parent, `path/..`, is on a different device than *path*, or whether `path/..` and *path* point to the same i-node on the same device —this should detect mount points for all Unix and POSIX variants. On Windows, a drive letter root and a share UNC are always mount points, and for any other path `GetVolumePathName` is called to see if it is different from the input path.

3.4 新版功能: 支持在 Windows 上检测非根安装点。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**join**(*path*, **paths*)
合理地拼接一个或多个路径部分。返回值是 *path* 和 **paths* 所有值的连接，每个非空部分后面都紧跟一个目录分隔符 (os.sep)，除了最后一部分。这意味着如果最后一部分为空，则结果将以分隔符结尾。如果参数中某个部分是绝对路径，则绝对路径前的路径都将被丢弃，并从绝对路径部分开始连接。

在 Windows 上，遇到绝对路径部分（例如 r'\foo'）时，不会重置盘符。如果某部分路径包含盘符，则会丢弃所有先前的部分，并重置盘符。请注意，由于每个驱动器都有一个“当前目录”，所以 os.path.join("c:", "foo") 表示驱动器 C: 上当前目录的相对路径 (c:foo)，而不是 c:\foo。

在 3.6 版更改: 接受一个类路径对象 用于 *path* 和 *paths* 。

os.path.**normcase**(*path*)
Normalize the case of a pathname. On Unix and Mac OS X, this returns the path unchanged; on case-insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to backward slashes. Raise a TypeError if the type of *path* is not str or bytes (directly or indirectly through the *os.PathLike* interface).

在 3.6 版更改: 接受一个*path-like object*。

os.path.**normpath**(*path*)
通过折叠多余的分隔符和对上级目录的引用来标准化路径名，所以 A//B、A/B/、A/./B 和 A/foo/../B 都会转换成 A/B。这个字符串操作可能会改变带有符号链接的路径的含义。在 Windows 上，本方法将正斜杠转换为反斜杠。要规范大小写，请使用*normcase()*。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**realpath**(*path*)
返回指定文件的规范路径，消除路径中存在的任何符号链接（如果操作系统支持）。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**relpath**(*path*, *start=os.curdir*)
返回从当前目录或 *start* 目录（可选）到达 *path* 之间要经过的相对路径。这仅仅是对路径的计算，不会访问文件系统来确认 *path* 或 *start* 的存在性或属性。

*开始*默认为*os.curdir*

Availability: Unix, Windows.

在 3.6 版更改: 接受一个*path-like object*。

os.path.**samefile**(*path1*, *path2*)
如果两个路径都指向相同的文件或目录，则返回 True。这由设备号和 inode 号确定，在任一路径上调用*os.stat()* 失败则抛出异常。

Availability: Unix, Windows.

在 3.2 版更改: 添加了 Windows 支持

在 3.4 版更改: Windows 现在使用与其他所有平台相同的实现。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**sameopenfile**(*fp1*, *fp2*)
如果文件描述符 *fp1* 和 *fp2* 指向相同文件，则返回 True。

Availability: Unix, Windows.

在 3.2 版更改: 添加了 Windows 支持

在 3.6 版更改: 接受一个*path-like object*。

os.path.**samestat**(*stat1*, *stat2*)
如果 stat 元组 *stat1* 和 *stat2* 指向相同文件，则返回 True。这些 stat 元组可能是由*os.fstat()*、*os.lstat()* 或*os.stat()* 返回的。本函数实现了*samefile()* 和*sameopenfile()* 底层所使用的比较过程。

Availability: Unix, Windows.

在 3.4 版更改: 添加了 Windows 支持

在 3.6 版更改: 接受一个*path-like object*。

os.path.**split**(*path*)
将路径 *path* 拆分为一对，即 (head, tail)，其中，*tail* 是路径的最后一部分，而 *head* 里是除最后部分外的所有内容。*tail* 部分不会包含斜杠，如果 *path* 以斜杠结尾，则 *tail* 将为空。如果 *path* 中没有斜杠，*head* 将为空。如果 *path* 为空，则 *head* 和 *tail* 均为空。*head* 末尾的斜杠会被去掉，除非它是根目录（即它仅包含一个或多个斜杠）。在所有情况下，join(head, tail) 指向的位置都与 *path* 相同（但字符串可能不同）。另请参见函数*dirname()* 和*basename()*。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**splitdrive**(*path*)
将路径 *path* 拆分为一对，即 (drive, tail)，其中 *drive* 是挂载点或空字符串。在没有驱动器概念的系统上，*drive* 将始终为空字符串。在所有情况下，drive + tail 都与 *path* 相同。

在 Windows 上，本方法将路径拆分为驱动器/UNC 根节点和相对路径。

如果路径 path 包含盘符，则 drive 将包含冒号及冒号前面的所有内容。例如 splitdrive("c:/dir") 返回 ("c:", "/dir")。

如果 path 是一个 UNC 路径，则 drive 将包含主机名和共享点，但不包括第四个分隔符。例如 splitdrive("//host/computer/dir") 返回 ("//host/computer", "/dir")。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**splitext**(*path*)
将路径 *path* 拆分为一对，即 (root, ext)，使 root + ext == path，其中 *ext* 为空或以英文句点开头，且最多包含一个句点。路径前的句点将被忽略，例如 splitext('.cshrc') 返回 ('.cshrc', '')。

在 3.6 版更改: 接受一个*path-like object*。

os.path.**splitunc**(*path*)
3.1 版后已移除: Use *splitdrive* instead.

Split the pathname *path* into a pair (unc, rest) so that *unc* is the UNC mount point (such as r'\\host\mount'), if present, and *rest* the rest of the path (such as r'\path\file.ext'). For paths containing drive letters, *unc* will always be the empty string.

Availability: Windows.

os.path.**supports_unicode_filenames**
如果（在文件系统限制下）允许将任意 Unicode 字符串用作文件名，则为 True。

11.3 fileinput —迭代来自多个输入流的行

源代码: Lib/fileinput.py

此模块实现了一个辅助类和一些函数用来快速编写访问标准输入或文件列表的循环。如果你只想要读写一个文件请参阅*open()*。

典型用法为:

```
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in `sys.argv[1:]`, defaulting to `sys.stdin` if the list is empty. If a filename is `'-'`, it is also replaced by `sys.stdin`. To specify an alternative list of filenames, pass it as the first argument to *input()*. A single file name is also allowed.

所有文件都默认以文本模式打开，但你可以通过在调用*input()* 或*FileInput* 时指定 *mode* 形参来重载此行为。如果在打开或读取文件时发生了 I/O 错误，将会引发*OSError*。

在 3.3 版更改: 原来会引发*IOError*；现在它是*OSError* 的别名。

如果 `sys.stdin` 被使用超过一次，则第二次之后的使用将不返回任何行，除非是被交互式的使用，或都是被显式地重置 (例如使用 `sys.stdin.seek(0)`)。

空文件打开后将立即被关闭；它们在文件列表中会被注意到的唯一情况只有当最后打开的文件为空的时候。

反回的行不会对换行符做任何处理，这意味着文件中的最后一行可能不带换行符。

想要控制文件的打开方式，你可以通过将 *openhook* 形参传给*fileinput.input()* 或*FileInput()* 来提供一个打开钩子。此钩子必须为一个函数，它接受两个参数，*filename* 和 *mode*，并返回一个以相应模式打开的文件类对象。此模块已经提供了两个有用的钩子。

以下函数是此模块的初始接口：

`fileinput.`**`input`**(*files=None*, *inplace=False*, *backup=''*, *bufsize=0*, *mode='r'*, *openhook=None*)

创建一个*FileInput* 类的实例。该实例将被用作此模块中函数的全局状态，并且还将在迭代期间被返回使用。此函数的形参将被继续传递给*FileInput* 类的构造器。

The *FileInput* instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs:

```
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

在 3.2 版更改: 可以被用作上下文管理器。

Deprecated since version 3.6, will be removed in version 3.8: *bufsize* 形参。

下列函数会使用*fileinput.input()* 所创建的全局状态；如果没有活动的状态，则会引发*RuntimeError*。

`fileinput.`**`filename`**()

返回当前被读取的文件名。在第一行被读取之前，返回 `None`。

`fileinput.`**`fileno`**()

返回以整数表示的当前文件 “文件描述符”。当未打开文件时（处在第一行和文件之间），返回 `-1`。

fileinput.**lineno**()
返回已被读取的累计行号。在第一行被读取之前，返回 0。在最后一个文件的最后一行被读取之后，返回该行的行号。

fileinput.**filelineno**()
返回当前文件中的行号。在第一行被读取之前，返回 0。在最后一个文件的最后一行被读取之后，返回此文件中该行的行号。

fileinput.**isfirstline**()
Returns true if the line just read is the first line of its file, otherwise returns false.

fileinput.**isstdin**()
Returns true if the last line was read from `sys.stdin`, otherwise returns false.

fileinput.**nextfile**()
关闭当前文件以使下次迭代将从下一个文件（如果存在）读取第一行；不是从该文件读取的行将不会被计入累计行数。直到下一个文件的第一行被读取之后文件名才会改变。在第一行被读取之前，此函数将不会生效；它不能被用来跳过第一个文件。在最后一个文件的最后一行被读取之后，此函数将不再生效。

fileinput.**close**()
关闭序列。

此模块所提供的实现了序列行为的类同样也可用于子类化：

class fileinput.**FileInput**(*files=None*, *inplace=False*, *backup=''*, *bufsize=0*, *mode='r'*, *openhook=None*)
类*FileInput* 是一个实现；它的方法*filename()*，*fileno()*，*lineno()*，*filelineno()*，*isfirstline()*，*isstdin()*，*nextfile()* 和*close()* 对应于此模块中具有相同名称的函数。此外它还有一个*readline()* 方法可返回下一个输入行，以及一个 `__getitem__()` 方法，该方法实现了序列行为。这种序列必须以严格的序列顺序来读写；随机读写和*readline()* 不可以被混用。

通过 *mode* 你可以指定要传给*open()* 的文件模式。它必须为 `'r'`, `'rU'`, `'U'` 和 `'rb'` 中的一个。

openhook 如果给出则必须为一个函数，它接受两个参数 *filename* 和 *mode*，并相应地返回一个打开的文件类对象。你不能同时使用 *inplace* 和 *openhook*。

A *FileInput* instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs:

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

在 3.2 版更改: 可以被用作上下文管理器。

3.4 版后已移除: `'rU'` 和 `'U'` 模式。

Deprecated since version 3.6, will be removed in version 3.8: *bufsize* 形参。

可选的原地过滤: 如果传递了关键字参数 `inplace=True` 给*fileinput.input()* 或*FileInput* 构造器，则文件会被移至备份文件并将标准输出定向到输入文件（如果已存在与备份文件同名的文件，它将被静默地替换）。这使得编写一个能够原地重写其输入文件的过滤器成为可能。如果给出了 *backup* 形参 (通常形式为 `backup='.<some extension>'`)，它将指定备份文件的扩展名，并且备份文件会被保留；默认情况下扩展名为 `'.bak'` 并且它会在输出文件关闭时被删除。在读取标准输入时原地过滤会被禁用。

此模块提供了以下两种打开文件钩子：

fileinput.**hook_compressed**(*filename*, *mode*)
使用*gzip* 和*bz2* 模块透明地打开 gzip 和 bzip2 压缩的文件（通过扩展名 `'.gz'` 和 `'.bz2'` 来识别）。如果文件扩展名不是 `'.gz'` 或 `'.bz2'`，文件会以正常方式打开（即使用*open()* 并且不带任何解压操作）。

使用示例: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed)`

`fileinput.`**`hook_encoded`**(*encoding*, *errors=None*)
返回一个通过*open()* 打开每个文件的钩子，使用给定的 *encoding* 和 *errors* 来读取文件。

使用示例: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`

在 3.6 版更改: 添加了可选的 *errors* 形参。

11.4 stat —解析 stat() 结果

源代码: Lib/stat.py

---

stat 模块定义了一些用于解析*os.stat()*, *os.fstat()* 和*os.lstat()* (如果它们存在) 输出结果的常量和函数。有关 `stat()`, `fstat()` 和 `lstat()` 调用的完整细节，请参阅你的系统文档。

在 3.4 版更改: stat 模块是通过 C 实现来支持的。

stat 模块定义了以下函数来检测特定文件类型:

`stat.`**`S_ISDIR`**(*mode*)
如果 mode 来自一个目录则返回非零值。

`stat.`**`S_ISCHR`**(*mode*)
如果 mode 来自一个字符专属的设备文件则返回非零值。

`stat.`**`S_ISBLK`**(*mode*)
如果 mode 来自一个块特殊设备文件则返回非零值。

`stat.`**`S_ISREG`**(*mode*)
如果 mode 来自一个常规文件则返回非零值。

`stat.`**`S_ISFIFO`**(*mode*)
如果 mode 来自一个 FIFO (命名管道) 则返回非零值。

`stat.`**`S_ISLNK`**(*mode*)
如果 mode 来自一个符号链接则返回非零值。

`stat.`**`S_ISSOCK`**(*mode*)
如果 mode 来自一个套接字则返回非零值。

`stat.`**`S_ISDOOR`**(*mode*)
如果 mode 来自一个门则返回非零值。

3.4 新版功能.

`stat.`**`S_ISPORT`**(*mode*)
如果 mode 来自一个事件端口则返回非零值。

3.4 新版功能.

`stat.`**`S_ISWHT`**(*mode*)
如果 mode 来自一个白输出则返回非零值。

3.4 新版功能.

定义了两个附加函数用于对文件模式进行更一般化的操作:

stat.**S_IMODE**(*mode*)
返回文件模式中可由*os.chmod()* 进行设置的部分—即文件的 permission 位，加上 sticky 位、set-group-id 以及 set-user-id 位（在支持这些部分的系统上）。

stat.**S_IFMT**(*mode*)
返回文件模式中描述文件类型的部分（供上面的 S_IS*() 函数使用）。

通常，你应当使用 os.path.is*() 函数来检测文件的类型；这里提供的函数则适用于当你要对同一文件执行多项检测并且希望避免每项检测的 stat() 系统调用开销的情况。这些函数也适用于检测有关未被*os.path* 处理的信息，例如检测块和字符设备等。

示例:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

另外还提供了一个附加的辅助函数用来将文件模式转换为人类易读的字符串：

stat.**filemode**(*mode*)
将文件模式转换为‘-rwxrwxrwx’形式的字符串。

3.3 新版功能.

在 3.4 版更改: 此函数支持*S_IFDOOR*, *S_IFPORT* and *S_IFWHT*。

以下所有变量是一些简单的符号索引，用于访问*os.stat()*, *os.fstat()* 或*os.lstat()* 所返回的 10 条目元组。

stat.**ST_MODE**
inode 保护模式。

stat.**ST_INO**
Inode 号

stat.**ST_DEV**
Inode 所在的设备。

stat.**ST_NLINK**
Inode 拥有的链接数量。

stat.**ST_UID**
所有者的用户 ID。

stat.**ST_GID**
所有者的用户组 ID。

stat.**ST_SIZE**
以字节为单位的普通文件大小；对于某些特殊文件的预期数据量。

stat.**ST_ATIME**
上次访问的时间。

stat.**ST_MTIME**
上次修改的时间。

stat.**ST_CTIME**
操作系统所报告的“ctime”。在某些系统上（例如 Unix）是元数据的最后修改时间，而在其他系统上（例如 Windows）则是创建时间（请参阅系统平台的文档了解相关细节）。

对于“文件大小”的解析可因文件类型的不同而变化。对于普通文件就是文件的字节数。对于大部分种类的 Unix（特别包括 Linux）的 FIFO 和套接字来说，“大小”则是指在调用*os.stat()*, *os.fstat()* 或*os.lstat()* 时等待读取的字节数；这在某些时候很有用处，特别是在一个非阻塞的打开后轮询这些特殊文件中的一个时。其他字符和块设备的文件大小字段的含义还会有更多变化，具体取决于底层系统调用的实现方式。

以下变量定义了在*ST_MODE* 字段中使用的旗标。

使用上面的函数会比使用第一组旗标更容易移植:

stat.**S_IFSOCK**
套接字

stat.**S_IFLNK**
符号链接。

stat.**S_IFREG**
普通文件。

stat.**S_IFBLK**
块设备

stat.**S_IFDIR**
目录

stat.**S_IFCHR**
字符设备。

stat.**S_IFIFO**
先进先出

stat.**S_IFDOOR**
门

3.4 新版功能.

stat.**S_IFPORT**
事件端口。

3.4 新版功能.

stat.**S_IFWHT**
白输出。

3.4 新版功能.

注解: *S_IFDOOR*, *S_IFPORT* or *S_IFWHT* 等文件类型在不受系统平台支持时会被定义为 0。

以下旗标还可以*os.chmod()* 的在 *mode* 参数中使用:

stat.**S_ISUID**
设置 UID 位。

stat.**S_ISGID**
设置分组 ID 位。这个位有几种特殊用途。对于目录它表示该目录将使用 BSD 语义: 在其中创建的文件将从目录继承其分组 ID，而不是从创建进程的有效分组 ID 继承，并且在其中创建的目录也将设置*S_ISGID* 位。对于没有设置分组执行位 (*S_IXGRP*) 的文件，设置分组 ID 位表示强制性文件/记录锁定 (另请参见*S_ENFMT*)。

stat.**S_ISVTX**
固定位。当对目录设置该位时则意味着此目录中的文件只能由文件所有者、目录所有者或特权进程来重命名或删除。

stat.**S_IRWXU**
文件所有者权限的掩码。

stat.**S_IRUSR**
所有者具有读取权限。

stat.**S_IWUSR**
所有者具有写入权限。

stat.**S_IXUSR**
所有者具有执行权限。

stat.**S_IRWXG**
组权限的掩码。

stat.**S_IRGRP**
组具有读取权限。

stat.**S_IWGRP**
组具有写入权限。

stat.**S_IXGRP**
组具有执行权限。

stat.**S_IRWXO**
其他人（不在组中）的权限掩码。

stat.**S_IROTH**
其他人具有读取权限。

stat.**S_IWOTH**
其他人具有写入权限。

stat.**S_IXOTH**
其他人具有执行权限。

stat.**S_ENFMT**
System V 执行文件锁定。此旗标是与*S_ISGID* 共享的: 文件/记录锁定会针对未设置分组执行位 (*S_IXGRP*) 的文件强制执行。

stat.**S_IREAD**
Unix V7 中*S_IRUSR* 的同义词。

stat.**S_IWRITE**
Unix V7 中*S_IWUSR* 的同义词。

stat.**S_IEXEC**
Unix V7 中*S_IXUSR* 的同义词。

以下旗标可以在*os.chflags()* 的 *flags* 参数中使用：

stat.**UF_NODUMP**
不要转储文件。

stat.**UF_IMMUTABLE**
文件不能更改。

stat.**UF_APPEND**
文件只能附加到。

stat.**UF_OPAQUE**
当通过联合堆栈查看时，目录是不透明的。

stat.**UF_NOUNLINK**
文件不能重命名或删除。

stat.**UF_COMPRESSED**
文件是压缩存储的（Mac OS X 10.6+）。

stat.**UF_HIDDEN**
文件不能显示在 GUI 中（Mac OS X 10.5+）。

stat.**SF_ARCHIVED**
文件可能已存档。

stat.**SF_IMMUTABLE**
文件不能更改。

stat.**SF_APPEND**
文件只能附加到。

stat.**SF_NOUNLINK**
文件不能重命名或删除。

stat.**SF_SNAPSHOT**
文件有一个快照文件

请参阅 *BSD 或 Mac OS 系统的指南页 *chflags(2)* 了解详情。

在 Windows 上，以下文件属性常量可用于检测*os.stat()* 所返回的 st_file_attributes 成员中的位。请参阅 Windows API 文档 了解有关这些常量含义的详情。

stat.**FILE_ATTRIBUTE_ARCHIVE**
stat.**FILE_ATTRIBUTE_COMPRESSED**
stat.**FILE_ATTRIBUTE_DEVICE**
stat.**FILE_ATTRIBUTE_DIRECTORY**
stat.**FILE_ATTRIBUTE_ENCRYPTED**
stat.**FILE_ATTRIBUTE_HIDDEN**
stat.**FILE_ATTRIBUTE_INTEGRITY_STREAM**
stat.**FILE_ATTRIBUTE_NORMAL**
stat.**FILE_ATTRIBUTE_NOT_CONTENT_INDEXED**
stat.**FILE_ATTRIBUTE_NO_SCRUB_DATA**
stat.**FILE_ATTRIBUTE_OFFLINE**
stat.**FILE_ATTRIBUTE_READONLY**
stat.**FILE_ATTRIBUTE_REPARSE_POINT**

stat.**FILE_ATTRIBUTE_SPARSE_FILE**
stat.**FILE_ATTRIBUTE_SYSTEM**
stat.**FILE_ATTRIBUTE_TEMPORARY**
stat.**FILE_ATTRIBUTE_VIRTUAL**
3.5 新版功能.

11.5 filecmp —文件及目录的比较

源代码: Lib/filecmp.py

---

filecmp 模块定义了用于比较文件及目录的函数，并且可以选取多种关于时间和准确性的折衷方案。对于文件的比较，另见*difflib* 模块。

filecmp 模块定义了如下函数:

filecmp.**cmp**(*f1*, *f2*, *shallow=True*)
比较名为 *f1* 和 *f2* 的文件，如果它们似乎相等则返回 `True` ，否则返回 `False` 。

如果 *shallow* 为真，那么具有相同*os.stat()* 签名的文件将会被认为是相等的。否则，将比较文件的内容。

需要注意，没有外部程序被该函数调用，这赋予了该函数可移植性与效率。

该函数会缓存过去的比较及其结果，且在文件的*os.stat()* 信息变化后缓存条目失效。所有的缓存可以通过使用*clear_cache()* 来清除。

filecmp.**cmpfiles**(*dir1*, *dir2*, *common*, *shallow=True*)
比较在两个目录 *dir1* 和 *dir2* 中，由 *common* 所确定名称的文件。

返回三组文件名列表：*match*, *mismatch*, *errors* 。*match* 含有相匹配的文件，*mismatch* 含有那些不匹配的，然后 *errors* 列出那些未被比较文件的名称。如果文件不存在于两目录中的任一个，或者用户缺少读取它们的权限，又或者因为其他的一些原因而无法比较，那么这些文件将会被列在 *errors* 中。

参数 *shallow* 具有同*filecmp.cmp()* 一致的含义与默认值。

例如，`cmpfiles('a', 'b', ['c', 'd/e'])` 将会比较 `a/c` 与 `b/c` 以及 `a/d/e` 与 `b/d/e` 。`'c'` 和 `'d/e'` 将会各自出现在返回的三个列表里的某一个列表中。

filecmp.**clear_cache**()
清除 filecmp 缓存。如果一个文件过快地修改，以至于超过底层文件系统记录修改时间的精度，那么该函数可能有助于比较该类文件。

3.4 新版功能.

11.5.1 dircmp 类

class filecmp.**dircmp**(*a*, *b*, *ignore=None*, *hide=None*)
创建一个用于比较目录 *a* 和 *b* 的新的目录比较对象。*ignore* 是需要忽略的文件名列表，且默认为*filecmp.DEFAULT_IGNORES* 。*hide* 是需要隐藏的文件名列表，且默认为 `[os.curdir, os.pardir]` 。

dircmp 类如*filecmp.cmp()* 中所描述的那样对文件进行 *shallow* 比较。

dircmp 类提供以下方法:

report()
将 *a* 与 *b* 之间的比较结果打印（到*sys.stdout* ）。

report_partial_closure()
打印 *a* 与 *b* 及共同直接子目录的比较结果。

report_full_closure()
打印 *a* 与 *b* 及共同子目录比较结果（递归地）。

dircmp 类提供了一些有趣的属性，用以得到关于参与比较的目录树的各种信息。

需要注意，通过 __getattr__() 钩子，所有的属性将会惰性求值，因此如果只使用那些计算简便的属性，将不会有速度损失。

left
目录 *a* 。

right
目录 *b* 。

left_list
经 *hide* 和 *ignore* 过滤，目录 *a* 中的文件与子目录。

right_list
经 *hide* 和 *ignore* 过滤，目录 *b* 中的文件与子目录。

common
同时存在于目录 *a* 和 *b* 中的文件和子目录。

left_only
仅在目录 *a* 中的文件和子目录。

right_only
仅在目录 *b* 中的文件和子目录。

common_dirs
同时存在于目录 *a* 和 *b* 中的子目录。

common_files
同时存在于目录 *a* 和 *b* 中的文件。

common_funny
在目录 *a* 和 *b* 中类型不同的名字，或者那些*os.stat()* 报告错误的名字。

same_files
在目录 *a* 和 *b* 中，使用类的文件比较操作符判定相等的文件。

diff_files
在目录 *a* 和 *b* 中，根据类的文件比较操作符判定内容不等的文件。

funny_files
在目录 *a* 和 *b* 中无法比较的文件。

subdirs
一个将*common_dirs* 中名称映射为*dircmp* 对象的字典。

filecmp.**DEFAULT_IGNORES**
3.4 新版功能.

默认被*dircmp* 忽略的目录列表。

下面是一个简单的例子，使用 subdirs 属性递归搜索两个目录以显示公共差异文件：

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
```

(下页继续)

(续上页)

```
...                 dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

11.6 tempfile —生成临时文件和目录

源代码： Lib/tempfile.py

该模块用于创建临时文件和目录，它可以跨平台使用。*TemporaryFile*、*NamedTemporaryFile*、*TemporaryDirectory* 和*SpooledTemporaryFile* 是带有自动清理功能的高级接口，可用作上下文管理器。*mkstemp()* 和*mkdtemp()* 是低级函数，使用完毕需手动清理。

所有由用户调用的函数和构造函数都带有参数，这些参数可以设置临时文件和临时目录的路径和名称。该模块生成的文件名包括一串随机字符，在公共的临时目录中，这些字符可以让创建文件更加安全。为了保持向后兼容性，参数的顺序有些奇怪。所以为了代码清晰，建议使用关键字参数。

这个模块定义了以下内容供用户调用：

tempfile.**TemporaryFile**(*mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*)

返回一个*file-like object* 作为临时存储区域。创建该文件使用了与*mkstemp()* 相同的安全规则。它将在关闭后立即销毁（包括垃圾回收机制关闭该对象时）。在 Unix 下，该文件在目录中的条目根本不创建，或者创建文件后立即就被删除了，其他平台不支持此功能。您的代码不应依赖使用此功能创建的临时文件名称，因为它在文件系统中的名称可能是可见的，也可能是不可见的。

生成的对象可以用作上下文管理器（参见*例子*）。完成文件对象的上下文或销毁后，临时文件将从文件系统中删除。

mode 参数默认值为 `'w+b'` 因此创建的文件可以读取或写入而不用关闭。因为是使用二进制模式，所以它在所有平台上的行为都保持一致而不用关心所存储的是什么数据。*buffering*, *encoding* 和 *newline* 的解读方式与*open()* 相同。

参数 *dir*、*prefix* 和 *suffix* 的含义和默认值都与它们在*mkstemp()* 中的相同。

在 POSIX 平台上，它返回的对象是真实的文件对象。在其他平台上，它是一个文件类对象 (file-like object)，它的 `file` 属性是底层的真实文件对象。

如果可用，则使用*os.O_TMPFILE* 标志（仅限于 Linux，需要 3.11 及更高版本的内核）。

在 3.5 版更改: 如果可用，现在用的是*os.O_TMPFILE* 标志。

tempfile.**NamedTemporaryFile**(*mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*)

此函数执行的操作与*TemporaryFile()* 完全相同，但确保了该临时文件在文件系统中具有可见的名称（在 Unix 上表现为目录条目不取消链接）。从返回的文件类对象的 `name` 属性中可以检索到文件名。在临时文件仍打开时，是否允许用文件名第二次打开文件，在各个平台上是不同的（在 Unix 上可以，但在 Windows NT 或更高版本上不行）。如果 *delete* 为 true（默认值），则文件会在关闭后立即被删除。该函数返回的对象始终是文件类对象 (file-like object)，它的 `file` 属性是底层的真实文件对象。文件类对象可以像普通文件一样在 `with` 语句中使用。

tempfile.**SpooledTemporaryFile**(*max_size=0*, *mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*)

此函数执行的操作与*TemporaryFile()* 完全相同，但会将数据缓存在内存中，直到文件大小

超过 *max_size*，或调用文件的 `fileno()` 方法为止，此时数据会被写入磁盘，并且写入操作与*TemporaryFile()* 相同。

此函数生成的文件对象有一个额外的方法——`rollover()`，可以忽略文件大小，让文件立即写入磁盘。

The returned object is a file-like object whose `_file` attribute is either an *io.BytesIO* or *io.StringIO* object (depending on whether binary or text *mode* was specified) or a true file object, depending on whether `rollover()` has been called. This file-like object can be used in a `with` statement, just like a normal file.

在 3.3 版更改: 现在，文件的 truncate 方法可接受一个 `size` 参数。

tempfile.**TemporaryDirectory**(*suffix=None*, *prefix=None*, *dir=None*)

此函数会安全地创建一个临时目录，且使用与*mkdtemp()* 相同的规则。此函数返回的对象可用作上下文管理器（参见*例子*）。完成上下文或销毁临时目录对象后，新创建的临时目录及其所有内容将从文件系统中删除。

The directory name can be retrieved from the `name` attribute of the returned object. When the returned object is used as a context manager, the `name` will be assigned to the target of the `as` clause in the `with` statement, if there is one.

可以调用 `cleanup()` 方法来手动清理目录。

3.2 新版功能.

tempfile.**mkstemp**(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

以最安全的方式创建一个临时文件。假设所在平台正确实现了*os.open()* 的*os.O_EXCL* 标志，则创建文件时不会有竞争的情况。该文件只能由创建者读写，如果所在平台用权限位来标记文件是否可执行，那么没有人有执行权。文件描述符不会过继给子进程。

与*TemporaryFile()* 不同，*mkstemp()* 用户用完临时文件后需要自行将其删除。

如果 *suffix* 不是 `None` 则文件名将以该后缀结尾，是 `None` 则没有后缀。*mkstemp()* 不会在文件名和后缀之间加点，如果需要加一个点号，请将其放在 *suffix* 的开头。

如果 *prefix* 不是 `None`，则文件名将以该前缀开头，是 `None` 则使用默认前缀。默认前缀是*gettempprefix()* 或*gettempprefixb()* 函数的返回值（自动调用合适的函数）。

如果 *dir* 不为 `None`，则在指定的目录创建文件，是 `None` 则使用默认目录。默认目录是从一个列表中选择出来的，这个列表不同平台不一样，但是用户可以设置 *TMPDIR*、*TEMP* 或 *TMP* 环境变量来设置目录的位置。因此，不能保证生成的临时文件路径很规范，比如，通过 `os.popen()` 将路径传递给外部命令时仍需要加引号。

如果 *suffix*、*prefix* 和 *dir* 中的任何一个不是 `None`，就要保证它们是同一数据类型。如果它们是 bytes，则返回的名称的类型就是 bytes 而不是 str。如果确实要用默认参数，但又想要返回值是 bytes 类型，请传入 `suffix=b''`。

如果指定了 *text* 参数，它表示的是以二进制模式（默认）还是文本模式打开文件。在某些平台上，两种模式没有区别。

mkstemp() 返回一个元组，元组中第一个元素是句柄，它是一个系统级句柄，指向一个打开的文件（等同于*os.open()* 的返回值），第二元素是该文件的绝对路径。

在 3.5 版更改: 现在，*suffix*、*prefix* 和 *dir* 可以以 bytes 类型按顺序提供，以获得 bytes 类型的返回值。之前只允许使用 str。*suffix* 和 *prefix* 现在可以接受 `None`，并且默认为 `None` 以使用合适的默认值。

tempfile.**mkdtemp**(*suffix=None*, *prefix=None*, *dir=None*)

以最安全的方式创建一个临时目录，创建该目录时不会有竞争的情况。该目录只能由创建者读取、写入和搜索。

mkdtemp() 用户用完临时目录后需要自行将其删除。

prefix、*suffix* 和 *dir* 的含义与它们在*mkstemp()* 中的相同。

mkdtemp() 返回新目录的绝对路径名。

在 3.5 版更改: 现在，*suffix*、*prefix* 和 *dir* 可以以 bytes 类型按顺序提供，以获得 bytes 类型的返回值。之前只允许使用 str。*suffix* 和 *prefix* 现在可以接受 None，并且默认为 None 以使用合适的默认值。

tempfile.**gettempdir**()
返回放置临时文件的目录的名称。这个方法的返回值就是本模块所有函数的 *dir* 参数的默认值。

Python 搜索标准目录列表，以找到调用者可以在其中创建文件的目录。这个列表是:

1. TMPDIR 环境变量指向的目录。
2. TEMP 环境变量指向的目录。
3. TMP 环境变量指向的目录。
4. 与平台相关的位置:
   - 在 Windows 上，依次为 C:\TEMP、C:\TMP、\TEMP 和 \TMP。
   - 在所有其他平台上，依次为 /tmp、/var/tmp 和 /usr/tmp。
5. 不得已时，使用当前工作目录。

搜索的结果会缓存起来，参见下面*tempdir* 的描述。

tempfile.**gettempdirb**()
与*gettempdir()* 相同，但返回值为字节类型。

3.5 新版功能.

tempfile.**gettempprefix**()
返回用于创建临时文件的文件名前缀，它不包含目录部分。

tempfile.**gettempprefixb**()
与*gettempprefix()* 相同，但返回值为字节类型。

3.5 新版功能.

本模块使用一个全局变量来存储由*gettempdir()* 返回的临时文件目录路径。可以直接给它赋值，这样可以覆盖自动选择的路径，但是不建议这样做。本模块中的所有函数都带有一个 *dir* 参数，该参数可用于指定目录，这是推荐的方法。

tempfile.**tempdir**
当设置为 None 以外的其他值时，此变量将决定本模块所有函数的 *dir* 参数的默认值。

如果在调用除*gettempprefix()* 外的上述任何函数时 tempdir 为 None (默认值) 则它会按照*gettempdir()* 中所描述的算法来初始化。

11.6.1 例子

以下是*tempfile* 模块典型用法的一些示例:

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
```

(下页继续)

(续上页)

```
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.6.2 已弃用的函数和变量

创建临时文件有一种历史方法，首先使用*mktemp()* 函数生成一个文件名，然后使用该文件名创建文件。不幸的是，这是不安全的，因为在调用*mktemp()* 与随后尝试创建文件的进程之间的时间里，其他进程可能会使用该名称创建文件。解决方案是将两个步骤结合起来，立即创建文件。这个方案目前被*mkstemp()* 和上述其他函数所采用。

tempfile.**mktemp**(*suffix=''*, *prefix='tmp'*, *dir=None*)

2.3 版后已移除: 使用*mkstemp()* 来代替。

返回一个绝对路径，这个路径指向的文件在调用本方法时不存在。*prefix*、*suffix* 和 *dir* 参数与*mkstemp()* 中的同名参数类似，不同之处在于不支持字节类型的文件名，不支持 `suffix=None` 和 `prefix=None`。

> **警告:** 使用此功能可能会在程序中引入安全漏洞。当你开始使用本方法返回的文件执行任何操作时，可能有人已经捷足先登了。*mktemp()* 的功能可以很轻松地用*NamedTemporaryFile()* 代替，当然需要传递 `delete=False` 参数:
>
> ```
> >>> f = NamedTemporaryFile(delete=False)
> >>> f.name
> '/tmp/tmptjujjt'
> >>> f.write(b"Hello World!\n")
> 13
> >>> f.close()
> >>> os.unlink(f.name)
> >>> os.path.exists(f.name)
> False
> ```

11.7 glob —Unix 风格路径名模式扩展

源代码: Lib/glob.py

---

glob 模块可根据 Unix 终端所用规则找出所有匹配特定模式的路径名，但会按不确定的顺序返回结果。波浪号扩展不会生效，但 `*`, `?` 以及表示为 `[]` 的字符范围将被正确地匹配。这是通过配合使用*os.scandir()* 和*fnmatch.fnmatch()* 函数来实现的，而不是通过实际发起调用子终端。请注意不同于*fnmatch.fnmatch()*，*glob* 会将以点号 (`.`) 开头的文件名作为特殊情况来处理。（对于波浪号和终端变量扩展，请使用*os.path.expanduser()* 和*os.path.expandvars()*。）

对于字面值匹配，请将原字符用方括号括起来。例如，`'[?]'` 将匹配字符 `'?'`。

参见:

pathlib 模块提供高级路径对象。

glob.**glob**(*pathname*, *, *recursive=False*)
返回匹配 *pathname* 的可能为空的路径名列表，路径名必须为包含一个路径描述的字符串。*pathname* 可以是绝对路径 (如 `/usr/src/Python-1.5/Makefile`) 或相对路径 (如 `../../Tools/*/*.gif`)，并且可包含 shell 风格的通配符。无效的符号链接可以包含在结果中 (与在 shell 中一样)。

If *recursive* is true, the pattern “`**`” will match any files and zero or more directories and subdirectories. If the pattern is followed by an `os.sep`, only directories and subdirectories match.

注解: 在一个较大的目录树中使用“`**`”模式可能会消耗非常多的时间。

在 3.5 版更改: 支持使用“`**`”的递归 glob.

glob.**iglob**(*pathname*, *, *recursive=False*)
返回一个*iterator*，它会产生与*glob()* 相同的结果，但不会实际地同时保存它们。

glob.**escape**(*pathname*)
转义所有特殊字符 (`'?'`, `'*'` 和 `'['`)。这适用于当你想要匹配可能带有特殊字符的任意字符串字面值的情况。在 drive/UNC 共享点中的特殊字符不会被转义，例如在 Windows 上 `escape('//?/c:/Quo vadis?.txt')` 将返回 `'//?/c:/Quo vadis[?].txt'`。

3.4 新版功能.

例如，考虑一个包含以下内容的目录：文件 `1.gif`, `2.txt`, `card.gif` 以及一个子目录 `sub` 其中只包含一个文件 `3.txt`. *glob()* 将产生如下结果。请注意路径的任何开头部分都将被保留。:

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

如果目录包含以 `.` 打头的文件，它们默认将不会被匹配。例如，考虑一个包含 `card.gif` 和 `.card.gif` 的目录:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

参见:

模块 *fnmatch* Shell 风格文件名（而非路径）扩展

11.8 fnmatch —Unix 文件名模式匹配

源代码: Lib/fnmatch.py

此模块提供了 Unix shell 风格的通配符，它们并不等同于正则表达式（关于后者的文档参见 *re* 模块）。shell 风格通配符所使用的特殊字符如下：

模式	含义
*	匹配所有
?	匹配任何单个字符
[seq]	匹配 *seq* 中的任何字符
[!seq]	匹配任何不在 *seq* 中的字符

对于字面值匹配，请将原字符用方括号括起来。例如，'[?]' 将匹配字符 '?'。

注意文件名分隔符 (Unix 上为 '/') 不是此模块所特有的。请参见 *glob* 模块了解文件名扩展 (*glob* 使用 *filter()* 来匹配文件名的各个部分)。类似地，以一个句点打头的文件名也不是此模块所特有的，可以通过 * 和 ? 模式来匹配。

fnmatch.**fnmatch**(*filename*, *pattern*)
检测 *filename* 字符串是否匹配 *pattern* 字符串，返回 *True* 或 *False*。两个形参都会使用 *os.path.normcase()* 进行大小写正规化。*fnmatchcase()* 可被用于执行大小写敏感的比较，无论这是否为所在操作系统的标准。

这个例子将打印当前目录下带有扩展名 .txt 的所有文件名:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

fnmatch.**fnmatchcase**(*filename*, *pattern*)
检测 *filename* 是否匹配 *pattern*，返回 *True* 或 *False*；此比较是大小写敏感的，并且不会应用 *os.path.normcase()*。

fnmatch.**filter**(*names*, *pattern*)
返回 *names* 列表中匹配 *pattern* 的子集。它等价于 [n for n in names if fnmatch(n, pattern)]，但其实现更为高效。

fnmatch.**translate**(*pattern*)
返回 shell 风格 *pattern* 转换成的正则表达式以便用于 *re.match()*。

示例:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<_sre.SRE_Match object; span=(0, 10), match='foobar.txt'>
```

参见:

模块*glob* Unix shell 风格路径扩展。

11.9 linecache —随机读写文本行

源代码: Lib/linecache.py

---

linecache 模块允许从一个 Python 源文件中获取任意的行，并会尝试使用缓存进行内部优化，常应用于从单个文件读取多行的场合。此模块被*traceback* 模块用来提取源码行以便包含在格式化的回溯中。

tokenize.open() 函数被用于打开文件。此函数使用*tokenize.detect_encoding()* 来获取文件的编码格式；如果未指明编码格式，则默认编码为 UTF-8。

linecache 模块定义了下列函数：

linecache.**getline**(*filename*, *lineno*, *module_globals=None*)
从名为 *filename* 的文件中获取 *lineno* 行，此函数绝不会引发异常—出现错误时它将返回 `''` (所有找到的行都将包含换行符作为结束)。

如果名为 *filename* 的文件未找到，该函数将在模块搜索路径 `sys.path` 中查找它，在此之前会先在 *module_globals* 中检查 **PEP 302** `__loader__`，以涵盖模块是从 zip 文件或其他非文件系统导入源导入的情况。

linecache.**clearcache**()
清空缓存。如果你不再需要之前使用*getline()* 从文件读取的行即可使用此函数。

linecache.**checkcache**(*filename=None*)
检查缓存有效性。如果缓存中的文件在磁盘上发生了改变，而你需要更新后的版本即可使用此函数。如果省略了 *filename*，它会检查缓存中的所有条目。

linecache.**lazycache**(*filename*, *module_globals*)
捕获有关某个非基于文件的模块的足够细节信息，以允许稍后再通过*getline()* 来获取其中的行，即使当稍后调用时 *module_globals* 为 `None`。这可以避免在实际需要读取行之前执行 I/O，也不必始终保持模块全局变量。

3.5 新版功能.

示例:

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.10 shutil —高阶文件操作

源代码： Lib/shutil.py

shutil 模块提供了一系列对文件和文件集合的高阶操作。特别是提供了一些支持文件拷贝和删除的函数。对于单个文件的操作，请参阅*os* 模块。

> **警告：** 即便是高阶文件拷贝函数 (*shutil.copy()*, *shutil.copy2()*) 也无法拷贝所有的文件元数据。
>
> 在 POSIX 平台上，这意味着将丢失文件所有者和组以及 ACL 数据。在 Mac OS 上，资源钩子和其他元数据不被使用。这意味着将丢失这些资源并且文件类型和创建者代码将不正确。在 Windows 上，将不会拷贝文件所有者、ACL 和替代数据流。

11.10.1 目录和文件操作

shutil.**copyfileobj**(*fsrc*, *fdst*[, *length*])
将文件类对象 *fsrc* 的内容拷贝到文件类对象 *fdst*。整数值 *length* 如果给出则为缓冲区大小。特别地，*length* 为负值表示拷贝数据时不对源数据进行分块循环处理；默认情况下会分块读取数据以避免不受控制的内存消耗。请注意如果 *fsrc* 对象的当前文件位置不为 0，则只有从当前文件位置到文件末尾的内容会被拷贝。

shutil.**copyfile**(*src*, *dst*, *, *follow_symlinks=True*)
将 *src* 文件的内容（不含元数据）拷贝到 *dst* 文件并返回 *dst*。*src* 和 *dst* 是字符串形式的路径。*dst* 必须是完整的目标文件名；对于接受一个目标目录路径的拷贝请参见*shutil.copy()*。如果 *src* 和 *dst* 指定了同一文件，则将引发*SameFileError*。

目标位置必须是可写的；否则将引发*OSError* 异常。如果 *dst* 已经存在，它将被替换。特殊文件如字符或块设备以及管道无法用此函数来拷贝。

如果 *follow_symlinks* 为假值且 *src* 为符号链接，则将创建一个新的符号链接而不是拷贝 *src* 所指向的文件。

在 3.3 版更改: 曾经是引发*IOError* 而不是*OSError*。增加了 *follow_symlinks* 参数。现在是返回 *dst*。

在 3.4 版更改: 引发*SameFileError* 而不是*Error*。由于前者是后者的子类，此改变是向后兼容的。

exception shutil.**SameFileError**
此异常会在*copyfile()* 中的源和目标为同一文件时被引发。

3.4 新版功能.

shutil.**copymode**(*src*, *dst*, *, *follow_symlinks=True*)
从 *src* 拷贝权限位到 *dst*。文件的内容、所有者和分组将不受影响。*src* 和 *dst* 均为字符串形式的路径名。如果 *follow_symlinks* 为假值，并且 *src* 和 *dst* 均为符号链接，*copymode()* 将尝试修改 *dst* 本身的模式（而非它所指向的文件）。此功能并不是在所有平台上均可用；请参阅*copystat()* 了解详情。如果*copymode()* 无法修改本机平台上的符号链接，而它被要求这样做，它将不做任何操作即返回。

在 3.3 版更改: 加入 *follow_symlinks* 参数。

shutil.**copystat**(*src*, *dst*, *, *follow_symlinks=True*)
从 *src* 拷贝权限位、最近访问时间、最近修改时间以及旗标到 *dst*。在 Linux 上，*copystat()* 还会在可能的情况下拷贝 “扩展属性”。文件内容、所有者和分组将不受影响。*src* 和 *dst* 均为字符串形式的路径名。

如果 *follow_symlinks* 为假值，并且 *src* 和 *dst* 均指向符号链接，*copystat()* 将作用于符号链接本身而非该符号链接所指向的文件—从 *src* 符号链接读取信息，并将信息写入 *dst* 符号链接。

注解: 并非所有平台者提供检查和修改符号链接的功能。Python 本身可以告诉你哪些功能是在本机上可用的。

- 如果 `os.chmod in os.supports_follow_symlinks` 为 `True`，则 *copystat()* 可以修改符号链接的权限位。
- 如果 `os.utime in os.supports_follow_symlinks` 为 `True`，则 *copystat()* 可以修改符号链接的最近访问和修改时间。
- 如果 `os.chflags in os.supports_follow_symlinks` 为 `True`，则 *copystat()* 可以修改符号链接的旗标。(`os.chflags` 不是在所有平台上均可用。)

在此功能部分或全部不可用的平台上，当被要求修改一个符号链接时，*copystat()* 将尽量拷贝所有内容。*copystat()* 一定不会返回失败信息。

更多信息请参阅 *os.supports_follow_symlinks*。

在 3.3 版更改: 添加了 *follow_symlinks* 参数并且支持 Linux 扩展属性。

shutil.**copy**(*src*, *dst*, ***, *follow_symlinks=True*)

将文件 *src* 拷贝到文件或目录 *dst*。*src* 和 *dst* 应为字符串。如果 *dst* 指定了一个目录，文件将使用 *src* 中的基准文件名拷贝到 *dst*。返回新创建文件所对应的路径。

如果 *follow_symlinks* 为假值且 *src* 为符号链接，则 *dst* 也将被创建为符号链接。如果 *follow_symlinks* 为真值且 *src* 为符号链接，*dst* 将成为 *src* 所指向的文件的一个副本。

copy() 会拷贝文件数据和文件的权限模式 (参见 *os.chmod()*)。其他元数据，例如文件的创建和修改时间不会被保留。要保留所有原有的元数据，请改用 *copy2()* 。

在 3.3 版更改: 添加了 *follow_symlinks* 参数。现在会返回新创建文件的路径。

shutil.**copy2**(*src*, *dst*, ***, *follow_symlinks=True*)

类似于 *copy()*，区别在于 *copy2()* 还会尝试保留文件的元数据。

当 *follow_symlinks* 为假值且 *src* 为符号链接时，*copy2()* 会尝试将来自 *src* 符号链接的所有元数据拷贝到新创建的 *dst* 符号链接。但是，此功能不是在所有平台上均可用。在此功能部分或全部不可用的平台上，*copy2()* 将尽量保留所有元数据；*copy2()* 一定不会返回失败的结果。

copy2() 会使用 *copystat()* 来拷贝文件元数据。请参阅 *copystat()* 了解有关修改符号链接元数据的平台支持的更多信息。

在 3.3 版更改: 添加了 *follow_symlinks* 参数，还会尝试拷贝扩展文件系统属性（目前仅限 Linux）。现在会返回新创建文件的路径。

shutil.**ignore_patterns**(**patterns*)

这个工厂函数会创建一个函数，它可被用作 *copytree()* 的 *ignore* 可调用对象参数，以忽略那些匹配所提供的 glob 风格的 *patterns* 之一的文件和目录。参见以下示例。

shutil.**copytree**(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*)

递归地拷贝以 *src* 为根路径的整个目录树，返回目标目录。名为 *dst* 的目标目录不必已存在；它本身和还不存在的父目录都将被自动创建。目录的权限和时间信息将通过 *copystat()* 来拷贝，单独的文件将使用 *shutil.copy2()* 来拷贝。

如果 *symlinks* 为真值，源目录树中的符号链接会在新目录树中表示为符号链接，并且原链接的元数据在平台允许的情况下也会被拷贝；如果为假值或省略，则会将被链接文件的内容和元数据拷贝到新目录树。

当 *symlinks* 为假值时，如果符号链接所指向的文件不存在，则会在拷贝进程的末尾将一个异常添加到 *Error* 异常中的错误列表。如果你希望屏蔽此异常那就将可选的 *ignore_dangling_symlinks* 旗标设为真值。请注意此选项在不支持 *os.symlink()* 的平台上将不起作用。

如果给出了 *ignore*，它必须是一个可调用对象，该对象将接受*copytree()* 所访问的目录以及*os.listdir()* 所返回的目录内容列表作为其参数。由于*copytree()* 是递归地被调用的，*ignore* 可调用对象对于每个被拷贝目录都将被调用一次。该可调用对象必须返回一个相对于当前目录的目录和文件名序列（即其第二个参数的子集）；随后这些名称将在拷贝进程中被忽略。*ignore_patterns()* 可被用于创建这种基于 glob 风格模式来忽略特定名称的可调用对象。

如果发生了异常，将引发一个附带原因列表的*Error*。

如果给出了 *copy_function*，它必须是一个将被用来拷贝每个文件的可调用对象。它在被调用时会将源路径和目标路径作为参数传入。默认情况下，*shutil.copy2()* 将被使用，但任何支持同样签名（与*shutil.copy()* 一致）的函数都可以使用。

在 3.3 版更改: 当 *symlinks* 为假值时拷贝元数据。现在会返回 *dst*。

在 3.2 版更改: 添加了 *copy_function* 参数以允许提供定制的拷贝函数。添加了 *ignore_dangling_symlinks* 参数以便在 *symlinks* 为假值时屏蔽符号链接错误。

shutil.**rmtree**(*path*, *ignore_errors=False*, *onerror=None*)

删除一个完整的目录树；*path* 必须指向一个目录（但不能是一个目录的符号链接）。如果 *ignore_errors* 为真值，删除失败导致的错误将被忽略；如果为假值或是省略，此类错误将通过调用由 *onerror* 所指定的处理程序来处理，或者如果此参数被省略则将引发一个异常。

注解: 在支持必要的基于 fd 的函数的平台上，默认会使用*rmtree()* 的可防御符号链接攻击的版本。在其他平台上，*rmtree()* 较易遭受符号链接攻击：给定适当的时间和环境，攻击者可以操纵文件系统中的符号链接来删除他们在其他情况下无法访问的文件。应用程序可以使用*rmtree.avoids_symlink_attacks* 函数属性来确定此类情况具体是哪一些。

如果提供了 *onerror*，它必须为接受三个形参的可调用对象: *function*, *path* 和 *excinfo*。

第一个形参 *function* 是引发异常的函数；它依赖于具体的平台和实现。第二个形参 *path* 将是传递给 *function* 的路径名。第三个形参 *excinfo* 将是由*sys.exc_info()* 所返回的异常信息。由 *onerror* 所引发的异常将不会被捕获。

在 3.3 版更改: 添加了一个防御符号链接攻击的版本，如果平台支持基于 fd 的函数就会被使用。

rmtree.**avoids_symlink_attacks**

指明当前平台和实现是否提供防御符号链接攻击的*rmtree()* 版本。目前它仅在平台支持基于 fd 的目录访问函数时才返回真值。

3.3 新版功能.

shutil.**move**(*src*, *dst*, *copy_function=copy2*)

递归地将一个文件或目录 (*src*) 移至另一位置 (*dst*) 并返回目标位置。

如果目标是已存在的目录，则 *src* 会被移至该目录下。如果目标已存在但不是目录，它可能会被覆盖，具体取决于*os.rename()* 的语义。

如果目标是在当前文件系统中，则会使用*os.rename()*。在其他情况下，*src* 将被拷贝至 *dst*，使用的函数为 *copy_function*，然后目标会被移除。对于符号链接，则将在 *dst* 之下或以其本身为名称创建一个指向 *src* 目标的新符号链接，并且 *src* 将被移除。

如果给出了 *copy_function*，则它必须为接受两个参数 *src* 和 *dst* 的可调用对象，并会在*os.rename()* 无法使用时被用来将 *src* 拷贝到 *dest*。如果源位置是一个目录，则会调用*copytree()*，并向它传入 `copy_function()`。默认的 *copy_function* 是*copy2()*。使用*copy()* 作为 *copy_function* 允许在无法附带拷贝元数据时让移动操作成功执行，但其代价是不拷贝任何元数据。

在 3.3 版更改: 为异类文件系统添加了显式的符号链接处理，以便使它适应 GNU 的 **mv** 的行为。现在会返回 *dst*。

在 3.5 版更改: 增加了 *copy_function* 关键字参数。

shutil.**disk_usage**(*path*)
返回给定路径的磁盘使用统计数据，形式为一个*named tuple*，其中包含 *total*, *used* 和 *free* 属性，分别表示总计、已使用和未使用空间的字节数。在 Windows 上，*path* 必须是一个目录；在 Unix 上，它可以是一个文件或一个目录。

3.3 新版功能.

Availability: Unix, Windows.

shutil.**chown**(*path*, *user=None*, *group=None*)
修改给定 *path* 的所有者 *user* 和/或 *group*。

user 可以是一个系统用户名或 uid；*group* 同样如此。要求至少有一个参数。

另请参阅下层的函数`os.chown()`。

Availability: Unix.

3.3 新版功能.

shutil.**which**(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)
返回当给定的 *cmd* 被调用时将要运行的可执行文件的路径。如果没有 *cmd* 会被调用则返回 None。

mode 是一个传递给`os.access()`的权限掩码，在默认情况下将确定文件是否存在并且为可执行文件。

当未指定 *path* 时，将会使用`os.environ()`的结果，返回“PATH”的值或回退为`os.defpath`。

在 Windows 上当前目录总是会被添加为 *path* 的第一项，无论你是否使用默认值或提供你自己的路径，这是命令行终端在查找可执行文件时所采用的行为方式。此外，当在 *path* 中查找 *cmd* 时，还会检查 PATHEXT 环境变量。例如，如果你调用 shutil.which("python")，`which()`将搜索 PATHEXT 来确定它要在 *path* 目录中查找 python.exe。例如，在 Windows 上:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

3.3 新版功能.

exception shutil.**Error**
此异常会收集在多文件操作期间所引发的异常。对于`copytree()`，此异常参数将是一个由三元组 (*srcname*, *dstname*, *exception*) 构成的列表。

copytree 示例

这个示例就是上面所描述的`copytree()`函数的实现，其中省略了文档字符串。它还展示了此模块所提供的许多其他函数。

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
```

(下页继续)

(续上页)

```
                copy2(srcname, dstname)
            # XXX What about devices, sockets etc.?
        except OSError as why:
            errors.append((srcname, dstname, str(why)))
        # catch the Error from the recursive copytree so that we can
        # continue with other files
        except Error as err:
            errors.extend(err.args[0])
    try:
        copystat(src, dst)
    except OSError as why:
        # can't copy file access times on Windows
        if why.winerror is None:
            errors.extend((src, dst, str(why)))
    if errors:
        raise Error(errors)
```

另一个使用*ignore_patterns()* 辅助函数的例子:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

这将会拷贝除 .pyc 文件和以 tmp 打头的文件或目录以外的所有条目.

另一个使用 *ignore* 参数来添加记录调用的例子:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

rmtree 示例

这个例子演示了如何在 Windows 上删除一个目录树，其中部分文件设置了只读属性位。它会使用 onerror 回调函数来清除只读属性位并再次尝试删除。任何后续的失败都将被传播。

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onerror=remove_readonly)
```

11.10.2 归档操作

3.2 新版功能.

在 3.5 版更改: 添加了对 *xztar* 格式的支持。

本模块也提供了用于创建和读取压缩和归档文件的高层级工具。它们依赖于*zipfile* 和*tarfile* 模块。

shutil.**make_archive**(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

创建一个归档文件（例如 zip 或 tar）并返回其名称。

base_name 是要创建的文件名称，包括路径，去除任何特定格式的扩展名。*format* 是归档格式：为 “zip”（如果*zlib* 模块可用），“tar”，“gztar”（如果*zlib* 模块可用），“bztar”（如果*bz2* 模块可用）或 “xztar”（如果*lzma* 模块可用）中的一个。

root_dir is a directory that will be the root directory of the archive; for example, we typically chdir into *root_dir* before creating the archive.

base_dir is the directory where we start archiving from; i.e. *base_dir* will be the common prefix of all files and directories in the archive.

root_dir 和 *base_dir* 默认均为当前目录。

如果 *dry_run* 为真值，则不会创建归档文件，但将要被执行的操作会被记录到 *logger*。

owner 和 *group* 将在创建 tar 归档文件时被使用。默认会使用当前的所有者和分组。

logger 必须是一个兼容 **PEP 282** 的对象，通常为*logging.Logger* 的实例。

verbose 参数已不再使用并进入弃用状态。

shutil.**get_archive_formats**()

返回支持的归档格式列表。所返回序列中的每个元素为一个元组 `(name, description)`。

默认情况下*shutil* 提供以下格式:

- *zip*: ZIP 文件（如果*zlib* 模块可用）。
- *tar*: 未压缩的 tar 文件。
- *gztar*: gzip 压缩的 tar 文件（如果*zlib* 模块可用）。
- *bztar*: bzip2 压缩的 tar 文件（如果*bz2* 模块可用）。
- *xztar*: xz 压缩的 tar 文件（如果*lzma* 模块可用）。

你可以通过使用*register_archive_format()* 注册新的格式或为任何现有格式提供你自己的归档程序。

shutil.**register_archive_format**(*name*, *function*[, *extra_args*[, *description*]])

为 *name* 格式注册一个归档程序。

function 是将被用来解包归档文件的可调用对象。该可调用对象将接收要创建文件的 *base_name*，再加上要归档内容的 *base_dir* (其默认值为*os.curdir*)。更多参数会被作为关键字参数传入: *owner*, *group*, *dry_run* 和 *logger* (与向*make_archive()* 传入的参数一致)。

如果给出了 *extra_args*，则其应为一个 `(name, value)` 对的序列，将在归档器可调用对象被使用时作为附加的关键字参数。

description 由*get_archive_formats()* 使用，它将返回归档器的列表。默认值为一个空字符串。

shutil.**unregister_archive_format**(*name*)

从支持的格式中移除归档格式 *name*。

shutil.**unpack_archive**(*filename*[, *extract_dir*[, *format*]])
解包一个归档文件。*filename* 是归档文件的完整路径。

extract_dir 是归档文件解包的目标目录名称。如果未提供，则将使用当前工作目录。

format 是归档格式：应为 "zip"，"tar"，"gztar"，"bztar" 或 "xztar" 之一。或者任何通过*register_unpack_format()* 注册的其他格式。如果未提供，*unpack_archive()* 将使用归档文件的后缀来检查是否注册了对应于该后缀的解包器。在未找到任何解包器的情况下，将引发*ValueError*。

shutil.**register_unpack_format**(*name*, *extensions*, *function*[, *extra_args*[, *description*]])
注册一个解包格式。*name* 为格式名称而 *extensions* 为对应于该格式的扩展名列表，例如 Zip 文件的扩展名为 `.zip`。

function 是将被用来解包归档文件的可调用对象。该可调用对象将接受归档文件的路径，加上该归档文件要被解包的目标目录。

如果提供了 *extra_args*，则其应为一个 `(name, value)` 元组的序列，将被作为关键字参数传递给该可调用对象。

可以提供 *description* 来描述该格式，它将被*get_unpack_formats()* 返回。

shutil.**unregister_unpack_format**(*name*)
撤销注册一个解包格式。*name* 为格式的名称。

shutil.**get_unpack_formats**()
返回所有已注册的解包格式列表。所返回序列中的每个元素为一个元组 `(name, extensions, description)`。

默认情况下*shutil* 提供以下格式:

- *zip*: ZIP 文件（只有在相应模块可用时才能解包压缩文件）。
- *tar*: 未压缩的 tar 文件。
- *gztar*: gzip 压缩的 tar 文件（如果*zlib* 模块可用）。
- *bztar*: bzip2 压缩的 tar 文件（如果*bz2* 模块可用）。
- *xztar*: xz 压缩的 tar 文件（如果*lzma* 模块可用）。

你可以通过使用*register_unpack_format()* 注册新的格式或为任何现有格式提供你自己的解包器。

归档程序示例

在这个示例中，我们创建了一个 gzip 压缩的 tar 归档文件，其中包含用户的 `.ssh` 目录下的所有文件:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

结果归档文件中包含有:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
```

(下页继续)

(续上页)

```
-rwx------ tarek/staff      668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff      609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff     1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff      397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff    37192 2010-02-06 18:23:10 ./known_hosts
```

11.10.3 查询输出终端的尺寸

shutil.**get_terminal_size**(*fallback=(columns, lines)*)

获取终端窗口的尺寸。

对于两个维度中的每一个，会分别检查环境变量 COLUMNS 和 LINES。如果定义了这些变量并且其值为正整数，则将使用这些值。

如果未定义 COLUMNS 或 LINES，这是通常的情况，则连接到*sys.__stdout__* 的终端将通过发起调用*os.get_terminal_size()* 被查询。

如果由于系统不支持查询，或是由于我们未连接到某个终端而导致查询终端尺寸不成功，则会使用在 fallback 形参中给出的值。fallback 默认为 (80, 24)，这是许多终端模拟器所使用的默认尺寸。

返回的值是一个*os.terminal_size* 类型的具名元组。

另请参阅: The Single UNIX Specification, Version 2, Other Environment Variables.

3.3 新版功能.

11.11 macpath —Mac OS 9 路径操作函数

源码： Lib/macpath.py

该模块是*os.path* 模块的 Mac OS 9（及更早版本）实现。它可用于在 Mac OS X（或任何其他平台）上操作旧式 Macintosh 路径名。

该模块中提供了以下函数：normcase()、normpath()、isabs()、join()、split()、isdir()、isfile()、walk()、exists()。对于其他可用的函数*os.path* 虚拟对应可用。

参见:

模块*os* 操作系统接口，包括处理比 Python 文件对象 更低级别文件的功能。

模块*io* Python 的内置 I/O 库，包括抽象类和一些具体的类，如文件 I/O 。

内置函数*open()* 使用 Python 打开文件进行读写的标准方法。

CHAPTER 12

数据持久化

本章中描述的模块支持在磁盘上以持久形式存储 Python 数据。*pickle* 和*marshal* 模块可以将许多 Python 数据类型转换为字节流，然后从字节中重新创建对象。各种与 DBM 相关的模块支持一系列基于散列的文件格式，这些格式存储字符串到其他字符串的映射。

本章中描述的模块列表是：

12.1 pickle ——Python 对象序列化

源代码： Lib/pickle.py

模块*pickle* 实现了对一个 Python 对象结构的二进制序列化和反序列化。“*Pickling*” 是将 Python 对象及其所拥有的层次结构转化为一个字节流的过程，而 “*unpickling*” 是相反的操作，会将（来自一个*binary file* 或者*bytes-like object* 的）字节流转化回一个对象层次结构。Pickling（和 unpickling）也被称为“序列化”，“编组”[1] 或者“平面化”。而为了避免混乱，此处采用术语“pickling”和“unpickling”。

警告： *pickle* 模块在接受被错误地构造或者被恶意地构造的数据时不安全。永远不要 unpickle 来自于不受信任的或者未经验证的来源的数据。

[1] 不要把它与*marshal* 模块混淆。

12.1.1 与其他 Python 模块间的关系

与 marshal 间的关系

Python 有一个更原始的序列化模块称为*marshal*，但一般地*pickle* 应该是序列化 Python 对象时的首选。*marshal* 存在主要是为了支持 Python 的 .pyc 文件.

pickle 模块与*marshal* 在如下几方面显著地不同：

- *pickle* 模块会跟踪已被序列化的对象，所以该对象之后再次被引用时不会再次被序列化。*marshal* 不会这么做。

  这隐含了递归对象和共享对象。递归对象指包含对自己的引用的对象。这种对象并不会被 marshal 接受，并且实际上尝试 marshal 递归对象会让你的 Python 解释器崩溃。对象共享发生在对象层级中存在多处引用同一对象时。*pickle* 只会存储这些对象一次，并确保其他的引用指向同一个主副本。共享对象将保持共享，这可能对可变对象非常重要。

- *marshal* 不能被用于序列化用户定义类及其实例。*pickle* 能够透明地存储并保存类实例，然而此时类定义必须能够从与被存储时相同的模块被引入。

- The *marshal* serialization format is not guaranteed to be portable across Python versions. Because its primary job in life is to support .pyc files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The *pickle* serialization format is guaranteed to be backwards compatible across Python releases.

与 json 模块的比较

Pickle 协议和 JSON (JavaScript Object Notation) 间有着本质的不同：

- JSON 是一个文本序列化格式（它输出 unicode 文本，尽管在大多数时候它会接着以 utf-8 编码），而 pickle 是一个二进制序列化格式；
- JSON 是我们可以直观阅读的，而 pickle 不是；
- JSON 是可互操作的，在 Python 系统之外广泛使用，而 pickle 则是 Python 专用的；
- 默认情况下，JSON 只能表示 Python 内置类型的子集，不能表示自定义的类；但 pickle 可以表示大量的 Python 数据类型（可以合理使用 Python 的对象内省功能自动地表示大多数类型，复杂情况可以通过实现*specific object APIs* 来解决）。

参见：

json 模块: 一个允许 JSON 序列化和反序列化的标准库模块

12.1.2 数据流格式

pickle 所使用的数据格式仅可用于 Python。这样做的好处是没有外部标准给该格式强加限制，比如 JSON 或 XDR（不能表示共享指针）标准；但这也意味着非 Python 程序可能无法重新读取 pickle 打包的 Python 对象。

默认情况下，*pickle* 格式使用相对紧凑的二进制来存储。如果需要让文件更小，可以高效地压缩 由 pickle 打包的数据。

pickletools 模块包含了相应的工具用于分析*pickle* 生成的数据流。*pickletools* 源码中包含了对 pickle 协议使用的操作码的大量注释。

当前用于 pickling 的协议共有 5 种。使用的协议版本越高，读取生成的 pickle 所需的 Python 版本就要越新。

- v0 版协议是原始的“人类可读”协议，并且向后兼容早期版本的 Python。

- v1 版协议是较早的二进制格式，它也与早期版本的 Python 兼容。
- v2 版协议是在 Python 2.3 中引入的。它为存储*new-style class* 提供了更高效的机制。欲了解有关第 2 版协议带来的改进，请参阅 **PEP 307**。
- v3 版协议添加于 Python 3.0。它具有对`bytes` 对象的显式支持，且无法被 Python 2.x 打开。这是目前默认使用的协议，也是在要求与其他 Python 3 版本兼容时的推荐协议。
- v4 版协议添加于 Python 3.4。它支持存储非常大的对象，能存储更多种类的对象，还包括一些针对数据格式的优化。有关第 4 版协议带来改进的信息，请参阅 **PEP 3154**。

注解： 序列化是一种比持久化更底层的概念，虽然`pickle` 读取和写入的是文件对象，但它不处理持久对象的命名问题，也不处理对持久对象的并发访问（甚至更复杂）的问题。`pickle` 模块可以将复杂对象转换为字节流，也可以将字节流转换为具有相同内部结构的对象。处理这些字节流最常见的做法是将它们写入文件，但它们也可以通过网络发送或存储在数据库中。`shelve` 模块提供了一个简单的接口，用于在 DBM 类型的数据库文件上 pickle 和 unpickle 对象。

12.1.3 模块接口

要序列化某个包含层次结构的对象，只需调用`dumps()` 函数即可。同样，要反序列化数据流，可以调用`loads()` 函数。但是，如果要对序列化和反序列化加以更多的控制，可以分别创建`Pickler` 或`Unpickler` 对象。

`pickle` 模块包含了以下常量：

pickle.**HIGHEST_PROTOCOL**
整数，可用的最高协议版本。此值可以作为 协议值传递给`dump()` 和`dumps()` 函数，以及`Pickler` 的构造函数。

pickle.**DEFAULT_PROTOCOL**
一个整数，表示封存操作使用的协议版本。它可能小于`HIGHEST_PROTOCOL`。当前默认协议版本为 3，它是一个为 Python 3 设计的新协议。

`pickle` 模块提供了以下方法，让打包过程更加方便。

pickle.**dump**(*obj*, *file*, *protocol=None*, ***, *fix_imports=True*)
Write a pickled representation of *obj* to the open *file object* *file*. This is equivalent to `Pickler(file, protocol).dump(obj)`.

可选参数 *protocol* 是一个整数，告知 pickler 使用指定的协议，可选择的协议范围从 0 到`HIGHEST_PROTOCOL`。如果没有指定，这一参数默认值为`DEFAULT_PROTOCOL`。指定一个负数就相当于指定`HIGHEST_PROTOCOL`。

参数 *file* 必须有一个 write() 方法，该 write() 方法要能接收字节作为其唯一参数。因此，它可以是一个打开的磁盘文件（用于写入二进制内容），也可以是一个`io.BytesIO` 实例，也可以是满足这一接口的其他任何自定义对象。

如果 *fix_imports* 为 True 且 *protocol* 小于 3，pickle 将尝试将 Python 3 中的新名称映射到 Python 2 中的旧模块名称，因此 Python 2 也可以读取打包出的数据流。

pickle.**dumps**(*obj*, *protocol=None*, ***, *fix_imports=True*)
Return the pickled representation of the object as a `bytes` object, instead of writing it to a file.

参数 *protocol* 和 *fix_imports* 的含义与它们在`dump()` 中的含义相同。

pickle.**load**(*file*, ***, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*)
Read a pickled object representation from the open *file object* *file* and return the reconstituted object hierarchy specified therein. This is equivalent to `Unpickler(file).load()`.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object's representation are ignored.

参数 *file* 必须有两个方法，其中 read() 方法接受一个整数参数，而 readline() 方法不需要参数。两个方法都应返回字节串。因此 *file* 可以是一个打开用于二进制读取的磁盘文件、一个*io.BytesIO* 对象，或者任何满足此接口要求的其他自定义对象。

可选的关键字参数是 *fix_imports*, *encoding* 和 *errors*，用于控制由 Python 2 生成的 pickle 流的兼容性。如果 *fix_imports* 为 true，则 pickle 将尝试将旧的 Python 2 名称映射到 Python 3 中对应的新名称。*encoding* 和 *errors* 参数告诉 pickle 如何解码 Python 2 存储的 8 位字符串实例；这两个参数默认分别为‘ASCII’和‘strict’。*encoding* 参数可置为‘bytes’来将这些 8 位字符串实例读取为字节对象。读取 NumPy array 和 Python 2 存储的*datetime*、*date* 和*time* 实例时，请使用 `encoding='latin1'`。

`pickle.`**`loads`**(*bytes_object*, *, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*)

Read a pickled object hierarchy from a *bytes* object and return the reconstituted object hierarchy specified therein.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object's representation are ignored.

可选的关键字参数是 *fix_imports*, *encoding* 和 *errors*，用于控制由 Python 2 生成的 pickle 流的兼容性。如果 *fix_imports* 为 true，则 pickle 将尝试将旧的 Python 2 名称映射到 Python 3 中对应的新名称。*encoding* 和 *errors* 参数告诉 pickle 如何解码 Python 2 存储的 8 位字符串实例；这两个参数默认分别为‘ASCII’和‘strict’。*encoding* 参数可置为‘bytes’来将这些 8 位字符串实例读取为字节对象。读取 NumPy array 和 Python 2 存储的*datetime*、*date* 和*time* 实例时，请使用 `encoding='latin1'`。

pickle 模块定义了以下 3 个异常：

exception `pickle.`**`PickleError`**

其他 pickle 异常的基类。它是*Exception* 的一个子类。

exception `pickle.`**`PicklingError`**

当*Pickler* 遇到无法解包的对象时抛出此错误。它是*PickleError* 的子类。

参考*可以被打包/解包的对象* 来了解哪些对象可以被打包。

exception `pickle.`**`UnpicklingError`**

当解包出错时抛出此异常，例如数据损坏或对象不安全。它是*PickleError* 的子类。

注意，解包时可能还会抛出其他异常，包括（但不限于）AttributeError、EOFError、ImportError 和 IndexError。

pickle 模块可导出两个类，*Pickler* 和*Unpickler*:

class `pickle.`**`Pickler`**(*file*, *protocol=None*, *, *fix_imports=True*)

它接受一个二进制文件用于写入 pickle 数据流。

可选参数 *protocol* 是一个整数，告知 pickler 使用指定的协议，可选择的协议范围从 0 到*HIGHEST_PROTOCOL*。如果没有指定，这一参数默认值为*DEFAULT_PROTOCOL*。指定一个负数就相当于指定*HIGHEST_PROTOCOL*。

参数 *file* 必须有一个 write() 方法，该 write() 方法要能接收字节作为其唯一参数。因此，它可以是一个打开的磁盘文件（用于写入二进制内容），也可以是一个*io.BytesIO* 实例，也可以是满足这一接口的其他任何自定义对象。

如果 *fix_imports* 为 True 且 *protocol* 小于 3，pickle 将尝试将 Python 3 中的新名称映射到 Python 2 中的旧模块名称，因此 Python 2 也可以读取打包出的数据流。

`dump`(*obj*)

Write a pickled representation of *obj* to the open file object given in the constructor.

`persistent_id`(*obj*)

默认什么也不做。它存在是为了让子类可以重载它。

如果*persistent_id()* 返回 None，*obj* 会被照常 pickle。如果返回其他值，*Pickler* 会将这个函数的返回值作为 *obj* 的持久化 ID（Pickler 本应得到序列化数据流并将其写入文件，若此函数有返回值，则得到此函数的返回值并写入文件）。这个持久化 ID 的解释应当定义在*Unpickler.persistent_load()* 中（该方法定义还原对象的过程，并返回得到的对象）。注意，*persistent_id()* 的返回值本身不能拥有持久化 ID。

参阅持久化外部对象 获取详情和使用示例。

dispatch_table

Pickler 对象的 dispatch 表是*copyreg.pickle()* 中用到的 *reduction* 函数的注册。dispatch 表本身是一个 class 到其 reduction 函数的映射键值对。一个 reduction 函数只接受一个参数，就是其关联的 class，函数行为应当遵守*__reduce__()* 接口规范。

Pickler 对象默认并没有*dispatch_table* 属性，该对象默认使用*copyreg* 模块中定义的全局 dispatch 表。如果要为特定 Pickler 对象自定义序列化过程，可以将*dispatch_table* 属性设置为类字典对象（dict-like object）。另外，如果*Pickler* 的子类设置了*dispatch_table* 属性，则该子类的实例会使用这个表作为默认的 dispatch 表。

参阅*Dispatch 表* 获取使用示例。

3.3 新版功能.

fast

已弃用。设为 True 则启用快速模式。快速模式禁用了“备忘录”(memo) 的使用，即不生成多余的 PUT 操作码来加快打包过程。不应将其与自指 (self-referential) 对象一起使用，否则将导致*Pickler* 无限递归。

如果需要进一步提高 pickle 的压缩率，请使用*pickletools.optimize()*。

class pickle.**Unpickler**(*file*, *, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*)

它接受一个二进制文件用于读取 pickle 数据流。

Pickle 协议版本是自动检测出来的，所以不需要参数来指定协议。

参数 *file* 必须有两个方法，其中 read() 方法接受一个整数参数，而 readline() 方法不需要参数。两个方法都应返回字节串。因此 *file* 可以是一个打开用于二进制读取的磁盘文件对象、一个*io.BytesIO* 对象，或者任何满足此接口要求的其他自定义对象。

可选的关键字参数有 *fix_imports*, *encoding* 和 *errors*，它们用于控制由 Python 2 所生成 pickle 流的兼容性支持。如果 *fix_imports* 为真值，则 pickle 将尝试把旧的 Python 2 名称映射到 Python 3 所使用的新名称。*encoding* 和 *errors* 将告知 pickle 如何解码由 Python 2 所封存的 8 位字符串实例；这两个参数的默认值分别为‘ASCII’和‘strict’。*encoding* 可设为‘bytes’以将这些 8 位字符串实例作为字节对象来读取。

load()

Read a pickled object representation from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein. Bytes past the pickled object’s representation are ignored.

persistent_load(*pid*)

默认抛出*UnpicklingError* 异常。

如果定义了此方法，*persistent_load()* 应当返回持久化 ID *pid* 所指定的对象。如果遇到无效的持久化 ID，则应当引发*UnpicklingError*。

参阅持久化外部对象 获取详情和使用示例。

find_class(*module*, *name*)

如有必要，导入 *module* 模块并返回其中名叫 *name* 的对象，其中 *module* 和 *name* 参数都是*str* 对象。注意，不要被这个函数的名字迷惑，*find_class()* 同样可以用来导入函数。

子类可以重载此方法，来控制加载对象的类型和加载对象的方式，从而尽可能降低安全风险。参阅限制全局变量 获取更详细的信息。

12.1.4 可以被打包/解包的对象

下列类型可以被打包：

- None、True 和 False
- 整数、浮点数、复数
- str、byte、bytearray
- 只包含可打包对象的集合，包括 tuple、list、set 和 dict
- 定义在模块顶层的函数（使用 def 定义，lambda 函数则不可以）
- 定义在模块顶层的内置函数
- 定义在模块顶层的类
- 某些类实例，这些类的*__dict__* 属性值或*__getstate__()* 函数的返回值可以被打包（详情参阅*打包类实例* 这一段）。

尝试打包不能被打包的对象会抛出*PicklingError* 异常，异常发生时，可能有部分字节已经被写入指定文件中。尝试打包递归层级很深的对象时，可能会超出最大递归层级限制，此时会抛出*RecursionError* 异常，可以通过*sys.setrecursionlimit()* 调整递归层级，不过请谨慎使用这个函数，因为可能会导致解释器崩溃。

注意，函数（内建函数或用户自定义函数）在被打包时，引用的是函数全名。[2] 这意味着只有函数所在的模块名，与函数名会被打包，函数体及其属性不会被打包。因此，在解包的环境中，函数所属的模块必须是可以被导入的，而且模块必须包含这个函数被打包时的名称，否则会抛出异常。[3]

同样的，类也只打包名称，所以在解包环境中也有和函数相同的限制。注意，类体及其数据不会被打包，所以在下面的例子中类属性 attr 不会存在于解包后的环境中：

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

这些限制决定了为什么必须在一个模块的顶层定义可打包的函数和类。

类似的，在打包类的实例时，其类体和类数据不会跟着实例一起被打包，只有实例数据会被打包。这样设计是有目的的，在将来修复类中的错误、给类增加方法之后，仍然可以载入原来版本类实例的打包数据来还原该实例。如果你准备长期使用一个对象，可能会同时存在较多版本的类体，可以为对象添加版本号，这样就可以通过类的*__setstate__()* 方法将老版本转换成新版本。

12.1.5 打包类实例

在本节中，我们描述了可用于定义、自定义和控制如何打包和解包类实例的通用流程。

通常，使一个实例可被打包不需要附加任何代码。Pickle 默认会通过 Python 的内省机制获得实例的类及属性。而当实例解包时，它的 __init__() 方法通常 *不会* 被调用。其默认动作是：先创建一个未初始化的实例，然后还原其属性，下面的代码展示了这种行为的实现机制：

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def load(cls, attributes):
```

(下页继续)

[2] This is why lambda functions cannot be pickled: all lambda functions share the same name: <lambda>.
[3] 抛出的异常有可能是*ImportError* 或*AttributeError*，也可能是其他异常。

(续上页)

```
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

类可以改变默认行为，只需定义以下一种或几种特殊方法：

object.**__getnewargs_ex__**()
对于使用第 2 版或更高版协议的 pickle，实现了*__getnewargs_ex__()* 方法的类可以控制在解包时传给 __new__() 方法的参数。本方法必须返回一对 (args, kwargs) 用于构建对象，其中 *args* 是表示位置参数的 tuple，而 *kwargs* 是表示命名参数的 dict。它们会在解包时传递给 __new__() 方法。

如果类的 __new__() 方法只接受关键字参数，则应当实现这个方法。否则，为了兼容性，更推荐实现*__getnewargs__()* 方法。

在 3.6 版更改: *__getnewargs_ex__()* 现在可用于第 2 和第 3 版协议。

object.**__getnewargs__**()
这个方法与上一个*__getnewargs_ex__()* 方法类似，但仅支持位置参数。它要求返回一个 tuple 类型的 args，用于解包时传递给 __new__() 方法。

如果定义了*__getnewargs_ex__()*，那么*__getnewargs__()* 就不会被调用。

在 3.6 版更改: 在 Python 3.6 前，第 2、3 版协议会调用*__getnewargs__()*，更高版本协议会调用*__getnewargs_ex__()*。

object.**__getstate__**()
类还可以进一步控制其实例的打包过程。如果类定义了*__getstate__()*，它就会被调用，其返回的对象是被当做实例内容来打包的，否则打包的是实例的 __dict__。如果*__getstate__()* 未定义，实例的*__dict__* 会被照常打包。

object.**__setstate__**(*state*)
当解包时，如果类定义了*__setstate__()*，就会在已解包状态下调用它。此时不要求实例的 state 对象必须是 dict。没有定义此方法的话，先前打包的 state 对象必须是 dict，且该 dict 内容会在解包时赋给新实例的 __dict__。

> **注解:** 如果*__getstate__()* 返回 False，那么在解包时就不会调用*__setstate__()* 方法。

参考*处理有状态的对象* 一段获取如何使用*__getstate__()* 和*__setstate__()* 方法的更多信息。

注解: At unpickling time, some methods like __getattr__(), __getattribute__(), or __setattr__() may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement *__getnewargs__()* or *__getnewargs_ex__()* to establish such an invariant; otherwise, neither __new__() nor __init__() will be called.

可以看出，其实 pickle 并不直接调用上面的几个函数。事实上，这几个函数是复制协议的一部分，它们实现了*__reduce__()* 这一特殊接口。复制协议提供了统一的接口，用于在打包或复制对象的过程中取得所需数据。[4]

尽管这个协议功能很强，但是直接在类中实现*__reduce__()* 接口容易产生错误。因此，设计类时应当尽可能的使用高级接口（比如*__getnewargs_ex__()*、*__getstate__()* 和*__setstate__()*）。后面仍然可以看到直接实现*__reduce__()* 接口的状况，可能别无他法，可能为了获得更好的性能，或者两者皆有之。

[4] *copy* 模块使用这一协议实现浅层 (shallow) 和深层 (deep) 复制操作。

object.**__reduce__**()
该接口当前定义如下。*__reduce__()* 方法不带任何参数，并且应返回字符串或最好返回一个元组（返回的对象通常称为“reduce 值”）。

如果返回字符串，该字符串会被当做一个全局变量的名称。它应该是对象相对于其模块的本地名称，pickle 模块会搜索模块命名空间来确定对象所属的模块。这种行为常在单例模式使用。

当返回的是一个元组时，它的长度必须在二至五项之间。可选项可以被省略或将值设为 None。每项的语义分别如下所示：

- 一个可调用对象，该对象会在创建对象的最初版本时调用。
- 可调用对象的参数，是一个元组。如果可调用对象不接受参数，必须提供一个空元组。
- 可选元素，用于表示对象的状态，将被传给前述的*__setstate__()* 方法。如果对象没有此方法，则这个元素必须是字典类型，并会被添加至*__dict__* 属性中。
- 可选元素，一个返回连续项的迭代器（而不是序列）。这些项会被 obj.append(item) 逐个加入对象，或被 obj.extend(list_of_items) 批量加入对象。这个元素主要用于 list 的子类，也可以用于那些正确实现了 append() 和 extend() 方法的类。（具体是使用 append() 还是 extend() 取决于 pickle 协议版本以及待插入元素的项数，所以这两个方法必须同时被类支持。）
- 可选元素，一个返回连续键值对的迭代器（而不是序列）。这些键值对将会以 obj[key] = value 的方式存储于对象中。该元素主要用于 dict 子类，也可以用于那些实现了 __setitem__() 的类。

object.**__reduce_ex__**(*protocol*)
作为替代选项，也可以实现*__reduce_ex__()* 方法。此方法的唯一不同之处在于它应接受一个整型参数用于指定协议版本。如果定义了这个函数，则会覆盖*__reduce__()* 的行为。此外，*__reduce__()* 方法会自动成为扩展版方法的同义词。这个函数主要用于为以前的 Python 版本提供向后兼容的 reduce 值。

持久化外部对象

为了获取对象持久化的利益，*pickle* 模块支持引用已封存数据流之外的对象。这样的对象是通过一个持久化 ID 来引用的，它应当是一个由字母数字类字符组成的字符串 (对于第 0 版协议)[5] 或是一个任意对象 (用于任意新版协议)。

The resolution of such persistent IDs is not defined by the *pickle* module; it will delegate this resolution to the user defined methods on the pickler and unpickler, *persistent_id()* and *persistent_load()* respectively.

To pickle objects that have an external persistent id, the pickler must have a custom *persistent_id()* method that takes an object as an argument and returns either None or the persistent id for that object. When None is returned, the pickler simply pickles the object as normal. When a persistent ID string is returned, the pickler will pickle that object, along with a marker so that the unpickler will recognize it as a persistent ID.

要解封外部对象，Unpickler 必须实现*persistent_load()* 方法，接受一个持久化 ID 对象作为参数并返回一个引用的对象。

下面是一个全面的例子，展示了如何使用持久化 ID 来封存外部对象。

```
# Simple example presenting how persistent ID can be used to pickle
# external objects by reference.

import pickle
import sqlite3
from collections import namedtuple
```

(下页继续)

[5] 对字母数字类字符的限制是由于持久化 ID 在协议版本 0 中是由分行符来分隔的。因此如果持久化 ID 中出现任何形式的分行符，封存结果就将变得无法读取。

(续上页)

```
# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and return it.
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")


def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
        )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))
```

(下页继续)

(续上页)

```
    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # Load the records from the pickle data stream.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)


if __name__ == '__main__':
    main()
```

Dispatch 表

如果想对某些类进行自定义封存，而又不想在类中增加用于封存的代码，就可以创建带有特殊 dispatch 表的 pickler。

在*copyreg* 模块的 `copyreg.dispatch_table` 中定义了全局 dispatch 表。因此，可以使用 `copyreg.dispatch_table` 修改后的副本作为自有 dispatch 表。

例如：

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

创建了一个带有自有 dispatch 表的*pickle.Pickler* 实例，它可以对 `SomeClass` 类进行特殊处理。另外，下列代码：

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

完成了相同的操作，但所有 `MyPickler` 的实例都会共用同一份 dispatch 表。使用*copyreg* 模块实现的等效代码是：

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

处理有状态的对象

下面的示例展示了如何修改类在封存时的行为。其中 TextReader 类打开了一个文本文件，每次调用其 readline() 方法则返回行号和该行的字符。在封存这个 TextReader 的实例时，除了文件对象，其他属性都会被保存。当解封实例时，需要重新打开文件，然后从上次的位置开始继续读取。实现这些功能需要实现 *__setstate__()* 和 *__getstate__()* 方法。

```
class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # Copy the object's state from self.__dict__ which contains
        # all our instance attributes. Always use the dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file
```

使用方法如下所示：

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

12.1.6 限制全局变量

默认情况下，解封将会导入在 pickle 数据中找到的任何类或函数。对于许多应用来说，此行为是不可接受的，因为它会允许解封器导入并发起调用任意代码。只须考虑当这个手工构建的 pickle 数据流被加载时会做什么:

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

在这个例子里，解封器导入*os.system()* 函数然后应用字符串参数 “echo hello world”。虽然这个例子不具攻击性，但是不难想象别人能够通过此方式对你的系统造成损害。

出于这样的理由，你可能会希望通过定制*Unpickler.find_class()* 来控制要解封的对象。与其名称所提示的不同，*Unpickler.find_class()* 会在执行对任何全局对象（例如一个类或一个函数）的请求时被调用。因此可以完全禁止全局对象或是将它们限制在一个安全的子集中。

下面的例子是一个解封器，它只允许某一些安全的来自*builtins* 模块的类被加载:

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

我们这个解封器的一个示例用法所达成的目标:

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
```

(下页继续)

(续上页)

```
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

正如我们这个例子所显示的，对于允许解封的对象你必须要保持谨慎。因此如果要保证安全，你可以考虑其他选择例如*xmlrpc.client* 中的编组 API 或是第三方解决方案。

12.1.7 性能

较新版本的 pickle 协议（第 2 版或更高）具有针对某些常见特性和内置类型的高效二进制编码格式。此外，*pickle* 模块还拥有一个以 C 编写的透明优化器。

12.1.8 例子

对于最简单的代码，请使用*dump()* 和*load()* 函数。

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

以下示例读取之前封存的数据。

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

参见:

模块*copyreg* 为扩展类型提供 pickle 接口所需的构造函数。

模块*pickletools* 用于处理和分析已打包数据的工具。

模块*shelve* 带索引的数据库，用于存放对象，使用了*pickle* 模块。

模块*copy* 浅层 (shallow) 和深层 (deep) 复制对象操作

模块*marshal* 高效地序列化内置类型的数据。

备注

12.2 copyreg —注意 pickle 支持函数

源代码: Lib/copyreg.py

---

copyreg 模块提供了可在封存特定对象时使用的一种定义函数方式。*pickle* 和*copy* 模块会在封存/拷贝特定对象时使用这些函数。此模块提供了非类对象构造器的相关配置信息。这样的构造器可以是工厂函数或类实例。

copyreg.**constructor**(*object*)
将 *object* 声明为一个有效的构造器。如果 *object* 是不可调用的（因而不是一个有效的构造器）则会引发*TypeError*。

copyreg.**pickle**(*type*, *function*, *constructor=None*)
声明该 *function* 应当被用作 *type* 类型对象的 “归约函数”。*function* 应当返回字符串或包含两到三个元素的元组。

如果提供了可选的 *constructor* 形参，它应当是一个可用来重建相应对象的可调用对象，在调用该对象时应传入由 *function* 所返回的参数元组。如果 *object* 是一个类或 *constructor* 是不可调用的则将引发*TypeError*。

请查看*pickle* 模块了解 *function* 和 *constructor* 所要求的接口的详情。请注意一个 pickler 对象或*pickle.Pickler* 的子类的*dispatch_table* 属性也可以被用来声明归约函数。

12.2.1 示例

以下示例将会显示如何注册一个封存函数，以及如何来使用它：

```
>>> import copyreg, copy, pickle
>>> class C(object):
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

12.3 shelve —Python 对象持久化

源代码: Lib/shelve.py

“shelf” 是一种持久化的类似字典的对象。与 “dbm” 数据库的区别在于 shelf 中的值（不是键！）实际上可以为任意 Python 对象—即*pickle* 模块能够处理的任何东西。这包括大部分类实例、递归数据类型，以及包含大量共享子对象的对象。键则为普通的字符串。

shelve.**open**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

打开一个持久化字典。filename 指定下层数据库的基准文件名。作为附带效果，会为 filename 添加一个扩展名并且可能创建更多的文件。默认情况下，下层数据库会以读写模式打开。可选的 *flag* 形参具有与*dbm.open()* *flag* 形参相同的含义。

默认会使用第 3 版 pickle 协议来序列化值。pickle 协议版本可通过 *protocol* 形参来指定。

由于 Python 语义的限制，shelf 无法确定一个可变的持久化字典条目在何时被修改。默认情况下 只有在被修改对象再赋值给 shelf 时才会写入该对象 (参见示例)。如果可选的 *writeback* 形参设为 `True`，则所有被访问的条目都将在内存中被缓存，并会在*sync()* 和*close()* 时被写入；这可以使得对持久化字典中可变条目的修改更方便，但是如果访问的条目很多，这会消耗大量内存作为缓存，并会使得关闭操作变得非常缓慢，因为所有被访问的条目都需要写回到字典（无法确定被访问的条目中哪个是可变的，也无法确定哪个被实际修改了）。

注解: 请不要依赖于 shelf 的自动关闭功能；当你不再需要时应当总是显式地调用*close()*，或者使用*shelve.open()* 作为上下文管理器:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

警告: 由于*shelve* 模块需要*pickle* 的支持，因此从不可靠的来源载入 shelf 是不安全的。与 pickle 一样，载入 shelf 时可以执行任意代码。

字典所支持的所有方法都被 shelf 对象所支持。因此很容易将基于字典的代码转换为需要持久化存储的代码。

额外支持的两个方法:

Shelf.**sync**()

如果 shelf 打开时将 *writeback* 设为*True* 则写回缓存中的所有条目。如果可行还会清空缓存并将持久化字典同步到磁盘。此方法会在使用*close()* 关闭 shelf 时自动被调用。

Shelf.**close**()

同步并关闭持久化 *dict* 对象。对已关闭 shelf 的操作将失败并引发*ValueError*。

参见:

持久化字典方案，使用了广泛支持的存储格式并具有原生字典的速度。

12.3.1 限制

- 可选择使用哪种数据库包 (例如*dbm.ndbm* 或*dbm.gnu*) 取决于支持哪种接口。因此使用*dbm* 直接打开数据库是不安全的。如果使用了*dbm*，数据库同样会（不幸地）受限于它—这意味着存储在数据库中的（封存形式的）对象尺寸应当较小，并且在少数情况下键冲突有可能导致数据库拒绝更新。
- *shelve* 模块不支持对 shelve 对象的 *并发读/写访问*。（多个同时读取访问则是安全的。）当一个程序打开一个 shelve 对象来写入时，不应再有其他程序同时打开它来读取或写入。Unix 文件锁定可被用来解决此问题，但这在不同 Unix 版本上会存在差异，并且需要有关所用数据库实现的细节知识。

class shelve.**Shelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

collections.abc.MutableMapping 的一个子类，它会将封存的值保存在 *dict* 对象中。

默认会使用第 3 版 pickle 协议来序列化值。pickle 协议版本可通过 *protocol* 形参来指定。请参阅*pickle* 文档来查看 pickle 协议的相关讨论。

如果 *writeback* 形参为 `True`，对象将为所有访问过的条目保留缓存并在同步和关闭时将它们写回到 *dict*。这允许对可变的条目执行自然操作，但是会消耗更多内存并让同步和关闭花费更长时间。

keyencoding 形参是在下层字典被使用之前用于编码键的编码格式。

Shelf 对象还可以被用作上下文管理器，在这种情况下它将在 `with` 语句块结束时自动被关闭。

在 3.2 版更改: 添加了 *keyencoding* 形参；之前，键总是使用 UTF-8 编码。

在 3.4 版更改: 添加了上下文管理器支持。

class shelve.**BsdDbShelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)

Shelf 的一个子类，将 `first()`, `next()`, `previous()`, `last()` 和 `set_location()` 对外公开，在来自 pybsddb 的第三方 `bsddb` 模块中可用，但在其他数据库模块中不可用。传给构造器的 *dict* 对象必须支持这些方法。这通常是通过调用 `bsddb.hashopen()`, `bsddb.btopen()` 或 `bsddb.rnopen()` 之一来完成的。可选的 *protocol*, *writeback* 和 *keyencoding* 形参具有与*Shelf* 类相同的含义。

class shelve.**DbfilenameShelf**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Shelf 的一个子类，它接受一个 *filename* 而非字典类对象。下层文件将使用*dbm.open()* 来打开。默认情况下，文件将以读写模式打开。可选的 *flag* 形参具有与*open()* 函数相同的含义。可选的 *protocol* 和 *writeback* 形参具有与*Shelf* 类相同的含义。

12.3.2 示例

对接口的总结如下 (`key` 为字符串，`data` 为任意对象):

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)
```

(下页继续)

(续上页)

```
# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]         # this works as expected, but...
d['xx'].append(3)           # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']              # extracts the copy
temp.append(5)              # mutates the copy
d['xx'] = temp              # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                   # close it
```

参见:

模块*dbm* dbm 风格数据库的泛型接口。

模块*pickle* *shelve* 所使用的对象序列化。

12.4 marshal —内部 Python 对象序列化

此模块包含一此能以二进制格式来读写 Python 值的函数。这种格式是 Python 专属的，但是独立于特定的机器架构（即你可以在一台 PC 上写入某个 Python 值，将文件传到一台 Sun 上并在那里读取它）。这种格式的细节有意不带文档说明；它可能在不同 Python 版本中发生改变（但这种情况极少发生）。[1]

这不是一个通用的“持久化”模块。对于通用的持久化以及通过 RPC 调用传递 Python 对象，请参阅*pickle* 和*shelve* 等模块。*marshal* 模块主要是为了支持读写 .pyc 文件形式“伪编译”代码的 Python 模块。因此，Python 维护者保留在必要时以不向下兼容的方式修改 marshal 格式的权利。如果你要序列化和反序列化 Python 对象，请改用*pickle* 模块–其执行效率相当，版本独立性有保证，并且 pickle 还支持比 marshal 更多样的对象类型。

警告: *marshal* 模块对于错误或恶意构建的数据来说是不安全的。永远不要 unmarshal 来自不受信任的或未经验证的来源的数据。

不是所有 Python 对象类型都受支持；一般来说，此模块只能写入和读取不依赖于特定 Python 调用的对象。下列类型是受支持的：布尔值、整数、浮点数、复数、字符串、字节串、字节数组、元组、列表、集合、冻结集合、字典和代码对象，需要了解的一点是元组、列表、集合、冻结集合和字典只在其所包含的值也是这些值时才受支持。单例对象*None*, *Ellipsis* and *StopIteration* 也可以被 marshal 和 unmarshal。对于 *version* 低于 3 的格式，递归列表、集合和字典无法被写入（见下文）。

有些函数可以读/写文件，还有些函数可以操作字节类对象。

这个模块定义了以下函数：

marshal.**dump**(*value*, *file*[, *version*])

向打开的文件写入值。值必须为受支持的类型。文件必须为可写的*binary file*。

[1] 此模块的名称来源于 Modula-3 (及其他语言) 的设计者所使用的术语，他们使用术语“marshal”来表示以自包含的形式传输数据。严格地说，将数据从内部形式转换为外部形式 (例如用于 RPC 缓冲区) 称为“marshal”而其逆过程则称为“unmarshal”。

如果值具有（或所包含的对象具有）不受支持的类型，则会引发*ValueError* —但是将向文件写入垃圾数据。对象也将不能正确地通过*load()* 重新读取。

version 参数指明 dump 应当使用的数据格式（见下文）。

marshal.**load**(*file*)
从打开的文件读取一个值并返回。如果读不到有效的值（例如由于数据为不同 Python 版本的不兼容 marshal 格式），则会引发*EOFError*, *ValueError* 或*TypeError*。文件必须为可读的*binary file*。

注解： 如果通过*dump()* marshal 了一个包含不受支持类型的对象，*load()* 将为不可 marshal 的类型替换 None。

---

marshal.**dumps**(*value*[, *version*])
返回将通过 dump(value, file) 被写入一个文件的字节串对象。值必须属于受支持的类型。如果值属于（或包含的对象属于）不受支持的类型则会引发*ValueError*。

version 参数指明 dumps 应当使用的数据类型（见下文）。

marshal.**loads**(*bytes*)
将*bytes-like object* 转换为一个值。如果找不到有效的值，则会引发*EOFError*, *ValueError* 或*TypeError*。输入的额外字节串会被忽略。

此外，还定义了以下常量：

marshal.**version**
指明模块所使用的格式。第 0 版为历史格式，第 1 版为共享固化的字符串，第 2 版对浮点数使用二进制格式。第 3 版添加了对于对象实例化和递归的支持。目前使用的为第 4 版。

备注

12.5 dbm —Unix “数据库”接口

源代码: Lib/dbm/__init__.py

---

dbm 是一种泛用接口，针对各种 DBM 数据库—包括*dbm.gnu* 或*dbm.ndbm*。如果未安装这些模块中的任何一种，则将使用*dbm.dumb* 模块中慢速但简单的实现。还有一个适用于 Oracle Berkeley DB 的 第三方接口。

exception dbm.**error**
一个元组，其中包含每个受支持的模块可引发的异常，另外还有一个名为*dbm.error* 的特殊异常作为第一项—后者最在引发*dbm.error* 时被使用。

dbm.**whichdb**(*filename*)
此函数会猜测各种简单数据库模块中的哪一个是可用的—*dbm.gnu*, *dbm.ndbm* 还是*dbm.dumb* —应该被用来打开给定的文件。

返回下列值中的一个：如果文件由于不可读或不存在而无法打开则返回 None；如果文件的格式无法猜测则返回空字符串 ('')；或是包含所需模块名称的字符串，例如 'dbm.ndbm' 或 'dbm.gnu'。

dbm.**open**(*file*, *flag='r'*, *mode=0o666*)
打开数据库文件 *file* 并返回一个相应的对象。

如果数据库文件已存在，则使用*whichdb()* 函数来确定其类型和要使用的适当模块；如果文件不存在，则会使用上述可导入模块中的第一个。

可选的 *flag* 参数可以是：

值	含义
'r'	以只读方式打开现有数据库（默认）
'w'	以读写方式打开现有数据库
'c'	以读写方式打开数据库，如果不存在则创建它
'n'	始终创建一个新的空数据库，以读写方式打开

可选的 *mode* 参数是文件的 Unix 模式，仅在要创建数据库时才会被使用。其默认值为八进制数 0o666 (并将被当前的 umask 所修改)。

open() 所返回的对象支持与字典相同的基本功能；可以存储、获取和删除键及其对应的值，并可使用 in 运算符和 keys() 方法，以及 get() 和 setdefault()。

在 3.2 版更改: 现在 get() 和 setdefault() 在所有数据库模块中均可用。

键和值总是被存储为字节串。这意味着当使用字符串时它们会在被存储之前隐式地转换至默认编码格式。

这些对象也支持在 with 语句中使用，当语句结束时将自动关闭它们。

在 3.4 版更改: 向*open()* 所返回的对象添加了上下文管理协议的原生支持。

以下示例记录了一些主机名和对应的标题，随后将数据库的内容打印出来。:

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception (most
    # likely a TypeError).
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

参见:

模块*shelve* 存储非字符串数据的持久化模块。

以下部分描述了各个单独的子模块。

12.5.1 dbm.gnu —GNU 对 dbm 的重解析

源代码: Lib/dbm/gnu.py

此模块与*dbm*模块很相似，但是改用 GNU 库 gdbm 来提供某些附加功能。请注意由*dbm.gnu*与*dbm.ndbm*所创建的文件格式是不兼容的。

*dbm.gnu*模块提供了对 GNU DBM 库的接口。dbm.gnu.gdbm 对象的行为类似于映射（字典），区别在于其键和值总是会在存储之前被转换为字节串。打印 gdbm 对象不会打印出键和值，并且 items() 和 values() 等方法也不受支持。

exception dbm.gnu.**error**
针对*dbm.gnu*专属错误例如 I/O 错误引发。*KeyError*的引发则针对一般映射错误例如指定了不正确的键。

dbm.gnu.**open**(*filename*[, *flag*[, *mode*]])
打开一个 gdbm 数据库并返回 gdbm 对象。*filename*参数为数据库文件名称。

可选的 *flag* 参数可以是：

值	含义
'r'	以只读方式打开现有数据库（默认）
'w'	以读写方式打开现有数据库
'c'	以读写方式打开数据库，如果不存在则创建它
'n'	始终创建一个新的空数据库，以读写方式打开

下列附加字符可被添加至旗标以控制数据库的打开方式：

值	含义
'f'	以快速模式打开数据库。写入数据库将不会同步。
's'	同步模式。这将导致数据库的更改立即写入文件。
'u'	不要锁定数据库。

不是所有旗标都可用于所有版本的 gdbm。模块常量 open_flags 为包含受支持旗标字符的字符串。如果指定了无效的旗标则会引发*error*。

可选的 *mode* 参数是文件的 Unix 模式，仅在要创建数据库时才会被使用。其默认值为八进制数 0o666。

除了与字典类似的方法，gdbm 对象还有以下方法：

gdbm.**firstkey**()
使用此方法和*nextkey()*方法可以循环遍历数据库中的每个键。遍历的顺序是按照 gdbm 的内部哈希值，而不会根据键的值排序。此方法将返回起始键。

gdbm.**nextkey**(*key*)
在遍历中返回 *key* 之后的的下一个键。以下代码将打印数据库 db 中的每个键，而不会在内存中创建一个包含所有键的列表：

```
k = db.firstkey()
while k != None:
    print(k)
    k = db.nextkey(k)
```

gdbm.**reorganize**()
如果你进行了大量删除操作并且想要缩减 gdbm 文件所使用的空间，此例程将可重新组织数据库。除非使用此重组功能否则 gdbm 对象不会缩减数据库文件大小；在其他情况下，被删除的文件空间将会保留并在添加新的 (键, 值) 对时被重用。

gdbm.**sync**()
当以快速模式打开数据库时，此方法会将任何未写入数据强制写入磁盘。

gdbm.**close**()
关闭 gdbm 数据库。

12.5.2 dbm.ndbm —基于 ndbm 的接口

源代码: Lib/dbm/ndbm.py

---

dbm.ndbm 模块提供了对 Unix “(n)dbm” 库的接口。Dbm 对象的行为类似于映射（字典），区别在于其键和值总是被存储为字节串。打印 dbm 对象不会打印出键和值，并且 items() 和 values() 等方法也不受支持。

此模块可与“经典 classic” ndbm 接口或 GNU GDBM 兼容接口一同使用。在 Unix 上，**configure** 脚本将尝试定位适当的头文件来简化此模块的构建。

exception dbm.ndbm.**error**
针对*dbm.ndbm* 专属错误例如 I/O 错误引发。*KeyError* 的引发则针对一般映射错误例如指定了不正确的键。

dbm.ndbm.**library**
所使用的 ndbm 实现库的名称。

dbm.ndbm.**open**(*filename*[, *flag*[, *mode*]])
打开一个 dbm 数据库并返回 ndbm 对象。*filename* 参数为数据库文件名称（不带 .dir 或 .pag 扩展名）。

可选的 *flag* 参数必须是下列值之一：

值	含义
'r'	以只读方式打开现有数据库（默认）
'w'	以读写方式打开现有数据库
'c'	以读写方式打开数据库，如果不存在则创建它
'n'	始终创建一个新的空数据库，以读写方式打开

可选的 *mode* 参数是文件的 Unix 模式，仅在要创建数据库时才会被使用。其默认值为八进制数 0o666 (并将被当前的 umask 所修改)。

除了与字典类似的方法，ndbm 对象还有以下方法：

ndbm.**close**()
关闭 ndbm 数据库。

12.5.3 dbm.dumb —便携式 DBM 实现

源代码: Lib/dbm/dumb.py

注解: *dbm.dumb* 模块的目的是在更健壮的模块不可用时作为*dbm* 模块的最终回退项。*dbm.dumb* 不是为高速运行而编写的，也不像其他数据库模块一样被经常使用。

---

dbm.dumb 模块提供了一个完全以 Python 编写的持久化字典类接口。不同于*dbm.gnu* 等其他模块，它不需要外部库。与其他持久化映射一样，它的键和值也总是被存储为字节串。

该模块定义以下内容：

exception dbm.dumb.**error**
针对*dbm.dumb* 专属错误例如 I/O 错误引发。*KeyError* 的引发则针对一般映射例如指定了不正确的键。

dbm.dumb.**open**(*filename*[, *flag*[, *mode*]])
打开一个 dumbdbm 数据库并返回 dumbdbm 对象。*filename* 参数为数据库文件的主名称（不带任何特定扩展名）。创建一个 dumbdbm 数据库时将创建多个带有 .dat 和 .dir 扩展名的文件。

可选的 *flag* 参数仅支持 'c' 和 'n' 这两个值的语义。其他值将默认设为数据库总是打开为可更新，并且在数据库不存在时将被创建。

可选的 *mode* 参数是文件的 Unix 模式，仅在要创建数据库时才会被使用。其默认值为八进制数 0o666 (并将被当前的 umask 所修改)。

在 3.5 版更改: *open()* 在 flag 值为 'n' 时将总是创建一个新的数据库。

Deprecated since version 3.6, will be removed in version 3.8: 创建数据库使用 'r' 和 'w' 模式。修改数据库使用 'r' 模式。

除了*collections.abc.MutableMapping* 类所提供的方法，dumbdbm 对象还提供了以下方法：

dumbdbm.**sync**()
同步磁盘上的目录和数据文件。此方法会由 Shelve.sync() 方法来调用。

dumbdbm.**close**()
关闭 dumbdbm 数据库。

12.6 sqlite3 —SQLite 数据库 DB-API 2.0 接口模块

源代码： Lib/sqlite3/

SQLite 是一个 C 语言库，它可以提供一种轻量级的基于磁盘的数据库，这种数据库不需要独立的服务器进程，也允许需要使用一种非标准的 SQL 查询语言来访问它。一些应用程序可以使用 SQLite 作为内部数据存储。可以用它来创建一个应用程序原型，然后再迁移到更大的数据库，比如 PostgreSQL 或 Oracle。

sqlite3 模块由 Gerhard Häring 编写。它提供了符合 DB-API 2.0 规范的接口，这个规范是 **PEP 249**。

要使用这个模块，必须先创建一个*Connection* 对象，它代表数据库。下面例子中，数据将存储在 example.db 文件中：

```
import sqlite3
conn = sqlite3.connect('example.db')
```

你也可以使用 :memory: 来创建一个内存中的数据库

当有了*Connection* 对象后，你可以创建一个*Cursor* 游标对象，然后调用它的*execute()* 方法来执行 SQL 语句：

```
c = conn.cursor()

# Create table
c.execute('''CREATE TABLE stocks
```

(下页继续)

(续上页)

```
             (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
conn.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
conn.close()
```

这些数据被持久化保存了，而且可以在之后的会话中使用它们：

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()
```

通常你的 SQL 操作需要使用一些 Python 变量的值。你不应该使用 Python 的字符串操作来创建你的查询语句，因为那样做不安全；它会使你的程序容易受到 SQL 注入攻击（在 https://xkcd.com/327/ 上有一个搞笑的例子，看看有什么后果）

推荐另外一种方法：使用 DB-API 的参数替换。在你的 SQL 语句中，使用 `?` 占位符来代替值，然后把对应的值组成的元组做为*execute()* 方法的第二个参数。（其他数据库可能会使用不同的占位符，比如 `%s` 或者 `:1`）例如：

```
# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print(c.fetchone())

# Larger example that inserts many records at a time
purchases = [('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
             ('2006-04-05', 'BUY', 'MSFT', 1000, 72.00),
             ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
            ]
c.executemany('INSERT INTO stocks VALUES (?,?,?,?,?)', purchases)
```

要在执行 SELECT 语句后获取数据，你可以把游标作为*iterator*，然后调用它的*fetchone()* 方法来获取一条匹配的行，也可以调用*fetchall()* 来得到包含多个匹配行的列表。

下面是一个使用迭代器形式的例子：

```
>>> for row in c.execute('SELECT * FROM stocks ORDER BY price'):
        print(row)

('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

参见：

https://github.com/ghaering/pysqlite pysqlite 的主页–sqlite3 在外部使用“pysqlite”名字进行开发。

https://www.sqlite.org SQLite 的主页；它的文档详细描述了它所支持的 SQL 方言的语法和可用的数据类型。

http://www.w3schools.com/sql/ 学习 SQL 语法的教程、参考和例子。

PEP 249 - DB-API 2.0 规范 Marc-André Lemburg 写的 PEP。

12.6.1 模块函数和常量

sqlite3.**version**
这个模块的版本号，是一个字符串。不是 SQLite 库的版本号。

sqlite3.**version_info**
这个模块的版本号，是一个由整数组成的元组。不是 SQLite 库的版本号。

sqlite3.**sqlite_version**
使用中的 SQLite 库的版本号，是一个字符串。

sqlite3.**sqlite_version_info**
使用中的 SQLite 库的版本号，是一个整数组成的元组。

sqlite3.**PARSE_DECLTYPES**
这个常量可以作为*connect()* 函数的 *detect_types* 参数。

设置这个参数后，*sqlite3* 模块将解析它返回的每一列申明的类型。它会申明的类型的第一个单词，比如“integer primary key”，它会解析出“integer”，再比如“number(10)”，它会解析出“number”。然后，它会在转换器字典里查找那个类型注册的转换器函数，并调用它。

sqlite3.**PARSE_COLNAMES**
这个常量可以作为*connect()* 函数的 *detect_types* 参数。

Setting this makes the SQLite interface parse the column name for each column it returns. It will look for a string formed [mytype] in there, and then decide that ‘mytype’ is the type of the column. It will try to find an entry of ‘mytype’ in the converters dictionary and then use the converter function found there to return the value. The column name found in *Cursor.description* is only the first word of the column name, i. e. if you use something like `'as "x [datetime]"'` in your SQL, then we will parse out everything until the first blank for the column name: the column name would simply be “x”.

sqlite3.**connect**(*database*[, *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, *uri*])
Opens a connection to the SQLite database file *database*. You can use `":memory:"` to open a database connection to a database that resides in RAM instead of on disk.

当一个数据库被多个连接访问的时候，如果其中一个进程修改这个数据库，在这个事务提交之前，这个 SQLite 数据库将会被一直锁定。*timeout* 参数指定了这个连接等待锁释放的超时时间，超时之后会引发一个异常。这个超时时间默认是 5.0（5 秒）。

isolation_level 参数，请查看*Connection* 对象的*isolation_level* 属性。

SQLite 原生只支持 5 种类型：TEXT，INTEGER，REAL，BLOB 和 NULL。如果你想用其它类型，你必须自己添加相应的支持。使用 *detect_types* 参数和模块级别的*register_converter()* 函数注册 ** 转换器 ** 可以简单的实现。

detect_types 默认为 0（即关闭，没有类型检测）。你也可以组合*PARSE_DECLTYPES* 和*PARSE_COLNAMES* 来开启类型检测。

默认情况下，*check_same_thread* 为*True*，只有当前的线程可以使用该连接。如果设置为*False*，则多个线程可以共享返回的连接。当多个线程使用同一个连接的时候，用户应该把写操作进行序列化，以避免数据损坏。

默认情况下，当调用 connect 方法的时候，*sqlite3* 模块使用了它的*Connection* 类。当然，你也可以创建*Connection* 类的子类，然后创建提供了 *factory* 参数的*connect()* 方法。

详情请查阅当前手册的*SQLite 与 Python 类型* 部分。

sqlite3 模块在内部使用语句缓存来避免 SQL 解析开销。如果要显式设置当前连接可以缓存的语句数，可以设置 *cached_statements* 参数。当前实现的默认值是缓存 100 条语句。

如果 *uri* 为真，则 *database* 被解释为 URI。它允许您指定选项。例如，以只读模式打开数据库：

```
db = sqlite3.connect('file:path/to/database?mode=ro', uri=True)
```

有关此功能的更多信息，包括已知选项的列表，可以在 ‘ SQLite URI 文档 <https://www.sqlite.org/uri.html>‘_ 中找到。

在 3.4 版更改: 增加了 *uri* 参数。

sqlite3.**register_converter**(*typename*, *callable*)
注册一个回调对象 *callable*, 用来转换数据库中的字节串为自定的 Python 类型。所有类型为 *typename* 的数据库的值在转换时，都会调用这个回调对象。通过指定*connect()* 函数的 *detect-types* 参数来设置类型检测的方式。注意，*typename* 与查询语句中的类型名进行匹配时不区分大小写。

sqlite3.**register_adapter**(*type*, *callable*)
注册一个回调对象 *callable*，用来转换自定义 Python 类型为一个 SQLite 支持的类型。这个回调对象 *callable* 仅接受一个 Python 值作为参数，而且必须返回以下某个类型的值：int，float，str 或 bytes。

sqlite3.**complete_statement**(*sql*)
如果字符串 *sql* 包含一个或多个完整的 SQL 语句（以分号结束）则返回*True*。它不会验证 SQL 语法是否正确，仅会验证字符串字面上是否完整，以及是否以分号结束。

它可以用来构建一个 SQLite shell，下面是一个例子：

```
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print("Enter your SQL commands to execute in sqlite3.")
print("Enter a blank line to exit.")

while True:
    line = input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print(cur.fetchall())
        except sqlite3.Error as e:
            print("An error occurred:", e.args[0])
        buffer = ""

con.close()
```

sqlite3.**enable_callback_tracebacks**(*flag*)
默认情况下，您不会获得任何用户定义函数中的回溯消息，比如聚合，转换器，授权器回调等。如果要调试它们，可以设置 *flag* 参数为 `True` 并调用此函数。之后，回调中的回溯信息将会输出到 `sys.stderr`。再次使用`False` 来禁用该功能。

12.6.2 连接对象（Connection）

class sqlite3.**Connection**
SQLite 数据库连接对象有如下的属性和方法：

isolation_level
获取或设置当前默认的隔离级别。表示自动提交模式的`None` 以及 “DEFERRED”，“IMMEDIATE” 或 “EXCLUSIVE” 其中之一。详细描述请参阅控制事务。

in_transaction
如果是在活动事务中（还没有提交改变），返回`True`，否则，返回`False`。它是一个只读属性。

3.2 新版功能.

cursor(*factory=Cursor*)
这个方法接受一个可选参数 *factory*，如果要指定这个参数，它必须是一个可调用对象，而且必须返回`Cursor` 类的一个实例或者子类。

commit()
这个方法提交当前事务。如果没有调用这个方法，那么从上一次提交 `commit()` 以来所有的变化在其他数据库连接上都是不可见的。如果你往数据库里写了数据，但是又查询不到，请检查是否忘记了调用这个方法。

rollback()
这个方法回滚从上一次调用`commit()` 以来所有数据库的改变。

close()
关闭数据库连接。注意，它不会自动调用`commit()` 方法。如果在关闭数据库连接之前没有调用`commit()`，那么你的修改将会丢失！

execute(*sql*[, *parameters*])
这是一个非标准的快捷方法，它会调用`cursor()` 方法来创建一个游标对象，并使用给定的 *parameters* 参数来调用游标对象的`execute()` 方法，最后返回这个游标对象。

executemany(*sql*[, *parameters*])
这是一个非标准的快捷方法，它会调用`cursor()` 方法来创建一个游标对象，并使用给定的 *parameters* 参数来调用游标对象的`executemany()` 方法，最后返回这个游标对象。

executescript(*sql_script*)
这是一个非标准的快捷方法，它会调用`cursor()` 方法来创建一个游标对象，并使用给定的 *sql_script* 参数来调用游标对象的`executescript()` 方法，最后返回这个游标对象。

create_function(*name*, *num_params*, *func*)
创建一个可以在 SQL 语句中使用的自定义函数，其中参数 *name* 为 SQL 语句中使用的函数名，*num_params* 是这个函数接受的参数个数（如果 *num_params* 为 -1，那这个函数可以接受任意数量的参数），最后一个参数 *func* 是作为 SQL 函数调用的一个 Python 可调用对象。

此函数可返回任何 SQLite 所支持的类型: bytes, str, int, float 和 `None`。

示例:

```
import sqlite3
import hashlib
```

(下页继续)

(续上页)

```
def md5sum(t):
    return hashlib.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", (b"foo",))
print(cur.fetchone()[0])
```

create_aggregate(*name*, *num_params*, *aggregate_class*)

创建一个自定义的聚合函数。

参数中 *aggregate_class* 类必须实现两个方法：`step` 和 `finalize`。`step` 方法接受 *num_params* 个参数（如果 *num_params* 为 -1，那么这个函数可以接受任意数量的参数）；`finalize` 方法返回最终的聚合结果。

`finalize` 方法可以返回任何 SQLite 支持的类型：bytes，str，int，float 和 `None`。

示例:

```
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print(cur.fetchone()[0])
```

create_collation(*name*, *callable*)

使用 *name* 和 *callable* 创建排序规则。这个 *callable* 接受两个字符串对象，如果第一个小于第二个则返回 -1，如果两个相等则返回 0，如果第一个大于第二个则返回 1。注意，这是用来控制排序的（SQL 中的 ORDER BY），所以它不会影响其它的 SQL 操作。

注意，这个 *callable* 可调用对象会把它的参数作为 Python 字节串，通常会以 UTF-8 编码格式对它进行编码。

以下示例显示了使用“错误方式”进行排序的自定义排序规则:

```
import sqlite3

def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
        return 1
```

(下页继续)

(续上页)

```
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print(row)
con.close()
```

要移除一个排序规则，需要调用 `create_collation` 并设置 callable 参数为 `None`。

```
con.create_collation("reverse", None)
```

interrupt()

可以从不同的线程调用这个方法来终止所有查询操作，这些查询操作可能正在连接上执行。此方法调用之后，查询将会终止，而且查询的调用者会获得一个异常。

set_authorizer(*authorizer_callback*)

此方法注册一个授权回调对象。每次在访问数据库中某个表的某一列的时候，这个回调对象将会被调用。如果要允许访问，则返回 `SQLITE_OK`，如果要终止整个 SQL 语句，则返回 `SQLITE_DENY`，如果这一列需要当做 NULL 值处理，则返回 `SQLITE_IGNORE`。这些常量可以在*sqlite3* 模块中找到。

回调的第一个参数表示要授权的操作类型。第二个和第三个参数将是参数或*None*，具体取决于第一个参数的值。第 4 个参数是数据库的名称（“main”，“temp” 等），如果需要的话。第 5 个参数是负责访问尝试的最内层触发器或视图的名称，或者如果此访问尝试直接来自输入 SQL 代码，则为*None*。

请参阅 SQLite 文档，了解第一个参数的可能值以及第二个和第三个参数的含义，具体取决于第一个参数。所有必需的常量都可以在*sqlite3* 模块中找到。

set_progress_handler(*handler*, *n*)

此例程注册回调。对 SQLite 虚拟机的每个多指令调用回调。如果要在长时间运行的操作期间从 SQLite 调用（例如更新用户界面），这非常有用。

如果要清除以前安装的任何进度处理程序，调用该方法时请将 *handler* 参数设置为*None*。

从处理函数返回非零值将终止当前正在执行的查询并导致它引发*OperationalError* 异常。

set_trace_callback(*trace_callback*)

为每个 SQLite 后端实际执行的 SQL 语句注册要调用的 *trace_callback*。

传递给回调的唯一参数是正在执行的语句（作为字符串）。回调的返回值将被忽略。请注意，后端不仅运行传递给*Cursor.execute()* 方法的语句。其他来源包括 Python 模块的事务管理和当前数据库中定义的触发器的执行。

将传入的 *trace_callback* 设为*None* 将禁用跟踪回调。

3.3 新版功能.

enable_load_extension(*enabled*)

此例程允许/禁止 SQLite 引擎从共享库加载 SQLite 扩展。SQLite 扩展可以定义新功能，聚合或全新的虚拟表实现。一个众所周知的扩展是与 SQLite 一起分发的全文搜索扩展。

默认情况下禁用可加载扩展。见[1].

3.2 新版功能.

```
import sqlite3

con = sqlite3.connect(":memory:")

# enable extension loading
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
    insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli␣
↪peppers cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin␣
↪onions garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli␣
↪cheese onions flour');
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin␣
↪sugar flour butter');
    """)
for row in con.execute("select rowid, name, ingredients from recipe where␣
↪name match 'pie'"):
    print(row)
```

load_extension(*path*)

此例程从共享库加载 SQLite 扩展。在使用此例程之前，必须使用*enable_load_extension()*启用扩展加载。

默认情况下禁用可加载扩展。见[1].

3.2 新版功能.

row_factory

您可以将此属性更改为可接受游标和原始行作为元组的可调用对象，并将返回实际结果行。这样，您可以实现更高级的返回结果的方法，例如返回一个可以按名称访问列的对象。

示例:

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d
```

(下页继续)

[1] sqlite3 模块默认没有构建可加载扩展支持，因为有一些平台带有不支持这个特性的 SQLite 库（特别是 Mac OS X）。要获得可加载扩展的支持，那么在编译配置的时候必须指定–enable-loadable-sqlite-extensions 选项。

(续上页)

```
con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone()["a"])
```

如果返回一个元组是不够的，并且你想要对列进行基于名称的访问，你应该考虑将*row_factory*设置为高度优化的*sqlite3.Row* 类型。*Row* 提供基于索引和不区分大小写的基于名称的访问，几乎没有内存开销。它可能比您自己的基于字典的自定义方法甚至基于 db_row 的解决方案更好。

text_factory

使用此属性可以控制为 TEXT 数据类型返回的对象。默认情况下，此属性设置为*str* 和*sqlite3* 模块将返回 TEXT 的 Unicode 对象。如果要返回字节串，可以将其设置为*bytes*。

您还可以将其设置为接受单个 bytestring 参数的任何其他可调用对象，并返回结果对象。

请参阅以下示例代码以进行说明：

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

AUSTRIA = "\xd6sterreich"

# by default, rows are returned as Unicode
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = bytes
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is bytes
# the bytestrings will be encoded in UTF-8, unless you stored garbage in the
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that appends "foo" to all strings
con.text_factory = lambda x: x.decode("utf-8") + "foo"
cur.execute("select ?", ("bar",))
row = cur.fetchone()
assert row[0] == "barfoo"
```

total_changes

返回自打开数据库连接以来已修改，插入或删除的数据库行的总数。

iterdump()

返回以 SQL 文本格式转储数据库的迭代器。保存内存数据库以便以后恢复时很有用。此函数提供与 **sqlite3** shell 中的 .dump 命令相同的功能。

示例：

```
# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3
```

(下页继续)

(续上页)

```
con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
```

12.6.3 Cursor 对象

class sqlite3.**Cursor**

Cursor 游标实例具有以下属性和方法。

execute(*sql*[, *parameters*])

执行 SQL 语句。可以是参数化 SQL 语句（即，在 SQL 语句中使用占位符）。*sqlite3* 模块支持两种占位符：问号（qmark 风格）和命名占位符（命名风格）。

以下是两种风格的示例：

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table people (name_last, age)")

who = "Yeltsin"
age = 72

# This is the qmark style:
cur.execute("insert into people values (?, ?)", (who, age))

# And this is the named style:
cur.execute("select * from people where name_last=:who and age=:age", {"who":␣
↪who, "age": age})

print(cur.fetchone())
```

execute() 将只执行一条单独的 SQL 语句。如果你尝试用它执行超过一条语句，将会引发*Warning*。如果你想要用一次调用执行多条 SQL 语句请使用*executescript()*。

executemany(*sql*, *seq_of_parameters*)

基于在序列 *seq_of_parameters* 中找到的所有形参序列或映射执行一条 SQL 命令。*sqlite3* 模块还允许使用*iterator* 代替序列来产生形参。

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def __next__(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
```

(下页继续)

(续上页)

```
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

cur.execute("select c from characters")
print(cur.fetchall())
```

这是一个使用生成器*generator* 的简短示例:

```
import sqlite3
import string

def char_generator():
    for c in string.ascii_lowercase:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print(cur.fetchall())
```

executescript(*sql_script*)

这是一个非标准的便捷方法，可用于一次执行多条 SQL 语句。它会首先执行一条 COMMIT 语句，再执行以形参方式获取的 SQL 脚本。

sql_script 可以是一个*str* 类的实例。

示例:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
```

(下页继续)

(续上页)

```
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
    """)
```

fetchone()
获取一个查询结果集的下一行，返回一个单独序列，或是在没有更多可用数据时返回*None*。

fetchmany(*size=cursor.arraysize*)
获取下一个多行查询结果集，返回一个列表。当没有更多可用行时将返回一个空列表。

每次调用获取的行数由 *size* 形参指定。如果没有给出该形参，则由 cursor 的 arraysize 决定要获取的行数。此方法将基于 size 形参值尝试获取指定数量的行。如果获取不到指定的行数，则可能返回较少的行。

请注意 *size* 形参会涉及到性能方面的考虑。为了获得优化的性能，通常最好是使用 arraysize 属性。如果使用 *size* 形参，则最好在从一个*fetchmany()* 调用到下一个调用之间保持相同的值。

fetchall()
获取一个查询结果的所有（剩余）行，返回一个列表。请注意 cursor 的 arraysize 属性会影响此操作的执行效率。当没有可用行时将返回一个空列表。

close()
立即关闭 cursor（而不是在当 __del__ 被调用的时候）。

从这一时刻起该 cursor 将不再可用，如果再尝试用该 cursor 执行任何操作将引发*ProgrammingError* 异常。

rowcount
虽然*sqlite3* 模块的*Cursor* 类实现了此属性，但数据库引擎本身对于确定“受影响行”/”已选择行”的支持并不完善。

对于*executemany()* 语句，修改行数会被汇总至*rowcount*。

根据 Python DB API 规格描述的要求，*rowcount* 属性“当未在 cursor 上执行 executeXX() 或者上一次操作的 rowcount 不是由接口确定时为 -1”。这包括 SELECT 语句，因为我们无法确定一次查询将产生的行计数，而要等获取了所有行时才会知道。。

在 SQLite 的 3.6.5 版之前，如果你执行 DELETE FROM table 时不附带任何条件，则*rowcount* 将被设为 0。

lastrowid
这个只读属性会提供最近修改行的 rowid。它只在你使用*execute()* 方法执行 INSERT 或 REPLACE 语句时会被设置。对于 INSERT 或 REPLACE 以外的操作或者当*executemany()* 被调用时，*lastrowid* 会被设为*None*。

如果 INSERT 或 REPLACE 语句操作失败则将返回上一次成功操作的 rowid。

在 3.6 版更改: 增加了 REPLACE 语句的支持。

arraysize
用于控制*fetchmany()* 返回行数的可读取/写入属性。该属性的默认值为 1，表示每次调用将获取单独一行。

description
这个只读属性将提供上一次查询的列名称。为了与 Python DB API 保持兼容，它会为每个列返回一个 7 元组，每个元组的最后六个条目均为*None*。

对于没有任何匹配行的 SELECT 语句同样会设置该属性。

connection
这个只读属性将提供*Cursor* 对象所使用的 SQLite 数据库*Connection*。通过调用*con.cursor()* 创建的*Cursor* 对象所包含的*connection* 属性将指向 *con*:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

12.6.4 行对象 *Row*

class sqlite3.**Row**
一个*Row* 实例，该实例将作为用于*Connection* 对象的高度优化的*row_factory*。它的大部分行为都会模仿元组的特性。

它支持使用列名称的映射访问以及索引、迭代、文本表示、相等检测和*len()* 等操作。

如果两个*Row* 对象具有完全相同的列并且其成员均相等，则它们的比较结果为相等。

keys()
此方法会在一次查询之后立即返回一个列名称的列表，它是*Cursor.description* 中每个元组的第一个成员。

在 3.5 版更改: 添加了对切片操作的支持。

让我们假设我们如上面的例子所示初始化一个表:

```
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')
c.execute("""insert into stocks
          values ('2006-01-05','BUY','RHAT',100,35.14)""")
conn.commit()
c.close()
```

现在我们将*Row* 插入:

```
>>> conn.row_factory = sqlite3.Row
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
```

(下页继续)

(续上页)

```
...     print(member)
...
2006-01-05
BUY
RHAT
100.0
35.14
```

12.6.5 异常

exception sqlite3.**Warning**
Exception 的一个子类。

exception sqlite3.**Error**
此模块中其他异常的基类。它是*Exception* 的一个子类。

exception sqlite3.**DatabaseError**
针对数据库相关错误引发的异常。

exception sqlite3.**IntegrityError**
当数据库的关系一致性受到影响时引发的异常。例如外键检查失败等。它是*DatabaseError* 的子类。

exception sqlite3.**ProgrammingError**
编程错误引发的异常，例如表未找到或已存在，SQL 语句存在语法错误，指定的形参数量错误等。它是*DatabaseError* 的子类。

exception sqlite3.**OperationalError**
与数据库操作相关而不一定能受程序员掌控的错误引发的异常，例如发生非预期的连接中断，数据源名称未找到，事务无法被执行等。它是*DatabaseError* 的子类。

exception sqlite3.**NotSupportedError**
在使用了某个数据库不支持的方法或数据库 API 时引发的异常，例如在一个不支持事务或禁用了事务的连接上调用*rollback()* 方法等。它是*DatabaseError* 的子类。

12.6.6 SQLite 与 Python 类型

概述

SQLite 原生支持如下的类型：NULL，INTEGER，REAL，TEXT，BLOB。

因此可以将以下 Python 类型发送到 SQLite 而不会出现任何问题：

Python 类型	SQLite 类型
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

这是 SQLite 类型默认转换为 Python 类型的方式：

SQLite 类型	Python 类型
NULL	*None*
INTEGER	*int*
REAL	*float*
TEXT	取决于*text_factory*，默认为*str*
BLOB	*bytes*

sqlite3 模块的类型系统可通过两种方式来扩展：你可以通过对象适配将额外的 Python 类型保存在 SQLite 数据库中，你也可以让*sqlite3* 模块通过转换器将 SQLite 类型转换为不同的 Python 类型。

使用适配器将额外的 Python 类型保存在 SQLite 数据库中。

如上文所述，SQLite 只包含对有限类型集的原生支持。要让 SQLite 能使用其他 Python 类型，你必须将它们**适配**至 sqlite3 模块所支持的 SQLite 类型中的一种：NoneType, int, float, str, bytes。

有两种方式能让*sqlite3* 模块将某个定制的 Python 类型适配为受支持的类型。

让对象自行调整

如果自己编写类，这是一种很好的方法。假设有这样的类::

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

现在你想将这种点对象保存在一个 SQLite 列中。首先你必须选择一种受支持的类型用来表示点对象。让我们就用 str 并使用一个分号来分隔坐标值。然后你需要给你的类加一个方法 `__conform__(self, protocol)`，它必须返回转换后的值。形参 *protocol* 将为 `PrepareProtocol`。

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])
```

注册可调用的适配器

另一种可能的做法是创建一个将该类型转换为字符串表示的函数并使用*register_adapter()* 注册该函数。

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])
```

sqlite3 模块有两个适配器可用于 Python 的内置*datetime.date* 和*datetime.datetime* 类型。现在假设我们想要存储*datetime.datetime* 对象，但不是表示为 ISO 格式，而是表示为 Unix 时间戳。

```
import sqlite3
import datetime
import time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print(cur.fetchone()[0])
```

将 SQLite 值转换为自定义 Python 类型

编写适配器让你可以将定制的 Python 类型发送给 SQLite。但要令它真正有用，我们需要实现从 Python 到 SQLite 再回到 Python 的双向转换。

输入转换器。

让我们回到 `Point` 类。我们以字符串形式在 SQLite 中存储了 x 和 y 坐标值。

首先，我们将定义一个转换器函数，它接受这样的字符串作为形参并根据该参数构造一个 `Point` 对象。

注解： 转换器函数在调用时 **总是**会附带一个*bytes* 对象，无论你将何种数据类型的值发给 SQLite。

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

现在你需要让*sqlite3*模块知道你从数据库中选取的其实是一个点对象。有两种方式都可以做到这件事：

- 隐式的声明类型
- 显式的通过列名

这两种方式会在模块函数和常量一节中描述，相应条目为*PARSE_DECLTYPES*和*PARSE_COLNAMES*常量。下面的示例说明了这两种方法。

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return ("%f;%f" % (point.x, point.y)).encode('ascii')

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

#########################
# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

#######################
# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print("with column names:", cur.fetchone()[0])
```

(下页继续)

(续上页)

```
cur.close()
con.close()
```

默认适配器和转换器

对于 datetime 模块中的 date 和 datetime 类型已提供了默认的适配器。它们将会以 ISO 日期/ISO 时间戳的形式发给 SQLite。

默认转换器使用的注册名称是针对*datetime.date*的 “date” 和针对*datetime.datetime*的 “timestamp”。

通过这种方式，你可以在大多数情况下使用 Python 的 date/timestamp 对象而无须任何额外处理。适配器的格式还与实验性的 SQLite date/time 函数兼容。

下面的示例演示了这一点。

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_
↪COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print(today, "=>", row[0], type(row[0]))
print(now, "=>", row[1], type(row[1]))

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"
↪')
row = cur.fetchone()
print("current_date", row[0], type(row[0]))
print("current_timestamp", row[1], type(row[1]))
```

如果存储在 SQLite 中的时间戳的小数位多于 6 个数字，则时间戳转换器会将该值截断至微秒精度。

12.6.7 控制事务

底层的 sqlite3 库默认会以 autocommit 模式运行，但 Python 的*sqlite3*模块默认则不使用此模式。

autocommit 模式意味着修改数据库的操作会立即生效。BEGIN 或 SAVEPOINT 语句会禁用 autocommit 模式，而用于结束外层事务的 COMMIT, ROLLBACK 或 RELEASE 则会恢复 autocommit 模式。

Python 的*sqlite3*模块默认会在数据修改语言 (DML) 类语句 (即 INSERT/UPDATE/DELETE/REPLACE) 之前隐式地执行一条 BEGIN 语句。

你可以控制*sqlite3*隐式执行的 BEGIN 语句的种类，具体做法是通过将 *isolation_level* 形参传给*connect()*调用，或者通过指定连接的 isolation_level 属性。如果你没有指定 *isolation_level*，将使用基本的 BEGIN，它等价于指定 DEFERRED。其他可能的值为 IMMEDIATE 和 EXCLUSIVE。

你可以禁用*sqlite3*模块的隐式事务管理，具体做法是将 isolation_level 设为 None。这将使得下层的 sqlite3 库采用 autocommit 模式。随后你可以通过在代码中显式地使用 BEGIN, ROLLBACK, SAVEPOINT 和 RELEASE 语句来完全控制事务状态。

在 3.6 版更改: 以前*sqlite3*会在 DDL 语句之前隐式地提交未完成事务。现在则不会再这样做。

12.6.8 有效使用 sqlite3

使用快捷方式

使用*Connection*对象的非标准 execute(), executemany() 和 executescript() 方法，可以更简洁地编写代码，因为不必显式创建（通常是多余的）*Cursor*对象。相反，*Cursor*对象是隐式创建的，这些快捷方法返回游标对象。这样，只需对*Connection*对象调用一次，就能直接执行 SELECT 语句并遍历对象。

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
    ]

con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table person(firstname, lastname)")

# Fill the table
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)

# Print the table contents
for row in con.execute("select firstname, lastname from person"):
    print(row)

print("I just deleted", con.execute("delete from person").rowcount, "rows")
```

通过名称而不是索引访问索引

*sqlite3*模块的一个有用功能是内置的*sqlite3.Row*类，它被设计用作行对象的工厂。

该类的行装饰器可以用索引（如元组）和不区分大小写的名称访问：

```
import sqlite3

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
for row in cur:
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]
```

使用连接作为上下文管理器

连接对象可以用来作为上下文管理器，它可以自动提交或者回滚事务。如果出现异常，事务会被回滚；否则，事务会被提交。

```
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("create table person (id integer primary key, firstname varchar unique)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into person(firstname) values (?)", ("Joe",))

# con.rollback() is called after the with block finishes with an exception, the
# exception is still raised and must be caught
try:
    with con:
        con.execute("insert into person(firstname) values (?)", ("Joe",))
except sqlite3.IntegrityError:
    print("couldn't add Joe twice")
```

12.6.9 常见问题

多线程

较老版本的 SQLite 在共享线程之间存在连接问题。这就是 Python 模块不允许线程之间共享连接和游标的原因。如果仍然尝试这样做，则在运行时会出现异常。

唯一的例外是调用*interrupt()*方法，该方法仅在从其他线程进行调用时才有意义。

备注

CHAPTER 13

数据压缩和存档

本章中描述的模块支持 zlib、gzip、bzip2 和 lzma 数据压缩算法，以及创建 ZIP 和 tar 格式的归档文件。参见由 *shutil* 模块提供的*归档操作* 。

13.1 zlib —与 gzip 兼容的压缩

此模块为需要数据压缩的程序提供了一系列函数，用于压缩和解压缩。这些函数使用了 zlib 库。zlib 库的项目主页是 http://www.zlib.net. 版本低于 1.1.3 的 zlib 与此 Python 模块之间存在已知的不兼容。1.1.3 版本的 zlib 存在一个安全漏洞，我们推荐使用 1.1.4 或更新的版本。

zlib 的函数有很多选项，一般需要按特定顺序使用。本文档没有覆盖全部的用法。更多详细信息请于 http://www.zlib.net/manual.html 参阅官方手册。

要读写 `.gz` 格式的文件，请参考 *gzip* 模块。

此模块中可用的异常和函数如下：

exception zlib.**error**
在压缩或解压缩过程中发生错误时的异常。

zlib.**adler32**(*data*[, *value*])
计算 *data* 的 Adler-32 校验值。(Adler-32 校验的可靠性与 CRC32 基本相当，但比计算 CRC32 更高效。) 计算的结果是一个 32 位的整数。参数 *value* 是校验时的起始值，其默认值为 1。借助参数 *value* 可为分段的输入计算校验值。此算法没有加密强度，不应用于身份验证和数字签名。此算法的目的仅为验证数据的正确性，不适合作为通用散列算法。

在 3.0 版更改: 返回值永远是无符号数。要在所有的 Python 版本和平台上获得相同的值，请使用 `adler32(data) & 0xffffffff`。

zlib.**compress**(*data*, *level=-1*)
压缩 *data* 中的字节，返回含有已压缩内容的 bytes 对象。参数 *level* 为整数，可取值为 `0` 到 `9` 或 `-1`，用于指定压缩等级。`1` (Z_BEST_SPEED) 表示最快速度和最低压缩率，`9` (Z_BEST_COMPRESSION) 表示最慢速度和最高压缩率。`0` (Z_NO_COMPRESSION) 表示不压缩。参数默认值为 `-1`

(Z_DEFAULT_COMPRESSION)。Z_DEFAULT_COMPRESSION 是速度和压缩率之间的平衡 (一般相当于设压缩等级为 6)。函数发生错误时抛出*error* 异常。

在 3.6 版更改: 现在，*level* 可作为关键字参数。

zlib.**compressobj**(*level=-1*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

返回一个压缩对象，用来压缩内存中难以容下的数据流。

参数 *level* 为压缩等级，是整数，可取值为 0 到 9 或 -1。1 (Z_BEST_SPEED) 表示最快速度和最低压缩率，9 (Z_BEST_COMPRESSION) 表示最慢速度和最高压缩率。0 (Z_NO_COMPRESSION) 表示不压缩。参数默认值为 -1 (Z_DEFAULT_COMPRESSION)。Z_DEFAULT_COMPRESSION 是速度和压缩率之间的平衡 (一般相当于设压缩等级为 6)。

method 表示压缩算法。现在只支持 DEFLATED 这个算法。

参数 *wbits* 指定压缩数据时所使用的历史缓冲区的大小 (窗口大小)，并指定压缩输出是否包含头部或尾部。参数的默认值是 15 (MAX_WBITS)。参数的值分为几个范围：

- +9 到 +15：窗口大小以 2 为底的对数。即这些值对应着 512 到 32768 的窗口大小。更大的值会提供更好的压缩，同时内存开销也会更大。压缩输出会包含 zlib 特定格式的头部和尾部。
- −9 到 −15：绝对值为窗口大小以 2 为底的对数。压缩输出仅包含压缩数据，没有头部和尾部。
- +25 到 +31 = 16 + (9 到 15)：后 4 个比特位为窗口大小以 2 为底的对数。压缩输出包含一个基本的 **gzip** 头部，并以校验和为尾部。

参数 *memLevel* 指定内部压缩操作时所占用内存大小。参数取 1 到 9。更大的值占用更多的内存，同时速度也更快输出也更小。

参数 *strategy* 用于调节压缩算法。可取值为 Z_DEFAULT_STRATEGY、Z_FILTERED、Z_HUFFMAN_ONLY、Z_RLE (zlib 1.2.0.1) 或 Z_FIXED (zlib 1.2.2.2)。

参数 *zdict* 指定预定义的压缩字典。它是一个字节序列 (如*bytes* 对象)，其中包含用户认为要压缩的数据中可能频繁出现的子序列。频率高的子序列应当放在字典的尾部。

在 3.3 版更改: 添加关键字参数 *zdict*。

zlib.**crc32**(*data*[, *value*])

计算 *data* 的 CRC (循环冗余校验) 值。计算的结果是一个 32 位的整数。参数 *value* 是校验时的起始值，其默认值为 0。借助参数 *value* 可为分段的输入计算校验值。此算法没有加密强度，不应用于身份验证和数字签名。此算法的目的仅为验证数据的正确性，不适合作为通用散列算法。

在 3.0 版更改: 返回值永远是无符号数。要在所有的 Python 版本和平台上获得相同的值，请使用 `crc32(data) & 0xffffffff`。

zlib.**decompress**(*data*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

解压 *data* 中的字节，返回含有已解压内容的 bytes 对象。参数 *wbits* 取决于 *data* 的格式，具体参见下边的说明。*bufsize* 为输出缓冲区的起始大小。函数发生错误时抛出*error* 异常。

wbits 形参控制历史缓冲区（或称“窗口尺寸”）的大小以及所期望的头部和尾部格式。它类似于*compressobj()* 的形参，但可接受更大范围的值：

- +8 至 +15：窗口尺寸以二为底的对数。输入必须包含 zlib 头部和尾部。
- 0：根据 zlib 头部自动确定窗口大小。只从 zlib 1.2.3.5 版起受支持。
- −8 to −15：使用 *wbits* 的绝对值作为窗口大小以二为底的对数。输入必须为原始数据流，没有头部和尾部。
- +24 至 +31 = 16 + (8 至 15)：使用后 4 个比特位作为窗口大小以二为底的对数。输入必须包括 gzip 头部和尾部。

- +40 至 +47 = 32 + (8 至 15)：使用后 4 个比特位作为窗口大小以二为底的对数，并且自动接受 zlib 或 gzip 格式。

当解压缩一个数据流时，窗口大小必须不小于用于压缩数据流的原始窗口大小；使用太小的值可能导致*error* 异常。默认 *wbits* 值对应于最大的窗口大小并且要求包括 zlib 头部和尾部。

bufsize 是用于存放解压数据的缓冲区初始大小。如果需要更大空间，缓冲区大小将按需增加，因此你不需要让这个值完全精确；对其进行调整仅会节省一点对 `malloc()` 的调用次数。

在 3.6 版更改: *wbits* 和 *bufsize* 可用作关键字参数。

`zlib.`**`decompressobj`**(*wbits=MAX_WBITS*[, *zdict*])
返回一个解压对象，用来解压无法被一次性放入内存的数据流。

wbits 形参控制历史缓冲区的大小（或称“窗口大小”）以及所期望的头部和尾部格式。它的含义与*对 decompress() 的描述* 相同。

zdict 形参指定指定一个预定义的压缩字典。如果提供了此形参，它必须与产生将解压数据的压缩器所使用的字典相同。

注解： 如果 *zdict* 是一个可变对象 (例如*bytearray*)，则你不可在对*decompressobj()* 的调用和对解压器的 `decompress()` 方法的调用之间修改其内容。

在 3.3 版更改: 增加了 *zdict* 形参。

压缩对象支持以下方法：

`Compress.`**`compress`**(*data*)
压缩 *data* 并返回 bytes 对象，这个对象含有 *data* 的部分或全部内容的已压缩数据。所得的对象必须拼接在上一次调用*compress()* 方法所得数据的后面。缓冲区中可能留存部分输入以供下一次调用。

`Compress.`**`flush`**([*mode*])
压缩所有缓冲区的数据并返回已压缩的数据。参数 *mode* 可以传入的常量为：`Z_NO_FLUSH`、`Z_PARTIAL_FLUSH`、`Z_SYNC_FLUSH`、`Z_FULL_FLUSH`、`Z_BLOCK` (zlib 1.2.3.4) 或 `Z_FINISH`。默认值为 `Z_FINISH`。`Z_FINISH` 关闭已压缩数据流并不允许再压缩其他数据，`Z_FINISH` 以外的值皆允许这个对象继续压缩数据。调用*flush()* 方法并将 *mode* 设为 `Z_FINISH` 后会无法再次调用*compress()*，此时只能删除这个对象。

`Compress.`**`copy`**()
返回此压缩对象的一个拷贝。它可以用来高效压缩一系列拥有相同前缀的数据。

解压缩对象支持以下方法：

`Decompress.`**`unused_data`**
一个 bytes 对象，其中包含压缩数据结束之后的任何字节数据。也就是说，它将为 `b""` 直到包含压缩数据的末尾字节可用。如果整个结果字节串都包含压缩数据，它将为一个空的 bytes 对象 `b""`。

`Decompress.`**`unconsumed_tail`**
一个 bytes 对象，其中包含未被上一次*decompress()* 调用所消耗的任何数据。此数据不能被 zlib 机制看到，因此你必须将其送回（可能要附带额外的数据拼接）到后续的*decompress()* 方法调用以获得正确的输出。

`Decompress.`**`eof`**
一个布尔值，指明是否已到达压缩数据流的末尾。

这使得区分正确构造的压缩数据流和不完整或被截断的压缩数据流成为可能。

3.3 新版功能.

`Decompress.`**`decompress`**(*data*, *max_length=0*)
解压缩 *data* 并返回 bytes 对象，其中包含对应于 *string* 中至少一部分数据的解压缩数据。此数据应当被

拼接到之前任何对*decompress()* 方法的调用所产生的输出。部分输入数据可能会被保留在内部缓冲区以供后续处理。

如果可选的形参 *max_length* 非零则返回值将不会长于 *max_length*。这可能意味着不是所有已压缩输入都能被处理；并且未被消耗的数据将被保存在*unconsumed_tail* 属性中。如果要继续解压缩则这个字节串必须被传给对*decompress()* 的后续调用。如果 *max_length* 为零则整个输入都会被解压缩，并且*unconsumed_tail* 将为空。

在 3.6 版更改: *max_length* 可用作关键字参数。

Decompress.**flush**([*length*])
所有挂起的输入会被处理，并且返回包含剩余未压缩输出的 bytes 对象。在调用*flush()* 之后，*decompress()* 方法将无法被再次调用；唯一可行的操作是删除该对象。

可选的形参 *length* 设置输出缓冲区的初始大小。

Decompress.**copy**()
返回解压缩对象的一个拷贝。它可以用来在数据流的中途保存解压缩器的状态以便加快随机查找数据流后续位置的速度。

通过下列常量可获取模块所使用的 zlib 库的版本信息：

zlib.**ZLIB_VERSION**
构建此模块时所用的 zlib 库的版本字符串。它的值可能与运行时所加载的 zlib 不同。运行时加载的 zlib 库的版本字符串为*ZLIB_RUNTIME_VERSION*。

zlib.**ZLIB_RUNTIME_VERSION**
解释器所加载的 zlib 库的版本字符串。

3.3 新版功能.

参见:

模块*gzip* 读写 **gzip** 格式的文件。

http://www.zlib.net zlib 库项目主页。

http://www.zlib.net/manual.html zlib 库用户手册。提供了库的许多功能的解释和用法。

13.2 gzip —对 gzip 格式的支持

源代码: Lib/gzip.py

此模块提供的简单接口帮助用户压缩和解压缩文件，功能类似于 GNU 应用程序 **gzip** 和 **gunzip**。

数据压缩由*zlib* 模块提供。

gzip 模块提供*GzipFile* 类和*open()*、*compress()*、*decompress()* 几个便利的函数。*GzipFile* 类可以读写 **gzip** 格式的文件，还能自动压缩和解压缩数据，这让操作压缩文件如同操作普通的*file object* 一样方便。

注意，此模块不支持部分可以被 **gzip** 和 **gunzip** 解压的格式，如利用 **compress** 或 **pack** 压缩所得的文件。

这个模块定义了以下内容：

gzip.**open**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)
以二进制方式或者文本方式打开一个 gzip 格式的压缩文件，返回一个*file object*。

filename 参数可以是一个实际的文件名 (一个 a *str* 对象或者*bytes* 对象), 或者是一个用来读写的已存在的文件对象。

mode 参数可以是二进制模式：'r', 'rb', 'a', 'ab', 'w', 'wb', 'x' or 'xb'，或者是文本模式 'rt', 'at', 'wt', or 'xt'。默认值是 'rb'。

The *compresslevel* argument is an integer from 0 to 9, as for the *GzipFile* constructor.

对于二进制模式，这个函数等价于*GzipFile* 构造器：GzipFile(filename, mode, compresslevel)。在这个例子中，*encoding*, *errors* 和 *newline* 三个参数一定不要设置。

对于文本模式，将会创建一个*GzipFile* 对象，并将它封装到一个*io.TextIOWrapper* 实例中，这个实例默认了指定编码，错误抓获行为和行。

在 3.3 版更改: 支持 *filename* 为一个文件对象，支持文本模式和 *encoding*, *errors* 和 *newline* 参数。

在 3.4 版更改: 支持 'x', 'xb' 和“’ xt’“ 三种模式。

在 3.6 版更改: 接受一个*path-like object*。

class gzip.**GzipFile**(*filename=None*, *mode=None*, *compresslevel=9*, *fileobj=None*, *mtime=None*)

GzipFile 类的构造器支持 truncate() 的异常，与*file object* 的大多数方法非常相似。*fileobj* 和 *filename* 至少有一个不为空。

新的实例基于 *fileobj*，它可以是一个普通文件，一个*io.BytesIO* 对象，或者任何一个与文件相似的对象。当 *filename* 是一个文件对象时，它的默认值是 None。

当 *fileobj* 为 None 时，*filename* 参数只用于 **gzip** 文件头中，这个文件有可能包含未压缩文件的源文件名。如果文件可以被识别，默认 *fileobj* 的文件名；否则默认为空字符串，在这种情况下文件头将不包含源文件名。

The *mode* argument can be any of 'r', 'rb', 'a', 'ab', 'w', 'wb', 'x', or 'xb', depending on whether the file will be read or written. The default is the mode of *fileobj* if discernible; otherwise, the default is 'rb'.

需要注意的是，文件默认使用二进制模式打开。如果要以文本模式打开文件一个压缩文件，请使用*open()* 方法 (或者使用*io.TextIOWrapper* 包装*GzipFile*)。

compresslevel 参数是一个从 0 to 9 的整数，用于控制压缩等级；1 最快但压缩比例最小，9 最慢但压缩比例最大。0 不压缩。默认为 9。

The *mtime* argument is an optional numeric timestamp to be written to the last modification time field in the stream when compressing. It should only be provided in compression mode. If omitted or None, the current time is used. See the *mtime* attribute for more details.

调用*GzipFile* 的 close() 方法不会关闭 *fileobj*，因为你可以希望增加其它内容到已经压缩的数中。你可以将一个*io.BytesIO* 对象作为 *fileobj*，也可以使用*io.BytesIO* 的*getvalue()* 方法从内存缓存中恢复数据。

GzipFile 支持*io.BufferedIOBase* 类的接口，包括迭代和 with 语句。只有 truncate() 方法没有实现。

GzipFile 还提供了以下的方法和属性:

peek(*n*)

在不移动文件指针的情况下读取 *n* 个未压缩字节。最多只有一个单独的读取流来服务这个方法调用。返回的字节数不一定刚好等于要求的数量。

注解: 调用*peek()* 并没有改变*GzipFile* 的文件指针，它可能改变潜在文件对象 (例如: *GzipFile* 使用 *fileobj* 参数进行初始化)。

3.2 新版功能.

mtime

在解压的过程中，最后修改时间字段的值可能来自于这个属性，以整数的形式出现。在读取任何文件头信息前，初始值为 None。

所有 **gzip** 东方压缩流中必须包含时间戳这个字段。以便于像 **gunzip** 这样的程序可以使用时间戳。格式与*time.time()* 的返回值和*os.stat()* 对象的*st_mtime* 属性值一样。

在 3.1 版更改: 支持 with 语句，构造器参数 *mtime* 和*mtime* 属性。

在 3.2 版更改: 添加了对零填充和不可搜索文件的支持。

在 3.3 版更改: 实现*io.BufferedIOBase.read1()* 方法。

在 3.4 版更改: 支持 'x' and 'xb' 两种模式。

在 3.5 版更改: 支持写入任意*bytes-like objects*。*read()* 方法可以接受 “None“为参数。

在 3.6 版更改: 接受一个*path-like object*。

gzip.**compress**(*data*, *compresslevel=9*)

压缩 *data* 返回一个包含压缩数据的*bytes* 对象。*compresslevel* 的意义与上面提到的*GzipFile* 构造器一至。

3.2 新版功能.

gzip.**decompress**(*data*)

解压数据，返回一个*bytes* 包含未解压数据的对象。

3.2 新版功能.

13.2.1 用法示例

读取压缩文件示例:

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

创建 GZIP 文件示例:

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

使用 GZIP 压缩已有的文件示例:

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

使用 GZIP 压缩二进制字符串示例:

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

参见:

模块*zlib* 支持 **gzip** 格式所需要的基本压缩模块。

13.3 bz2 —对 bzip2 压缩算法的支持

源代码： Lib/bz2.py

此模块提供了使用 bzip2 压缩算法压缩和解压数据的一套完整的接口。

bz2 模块包含：

- 用于读写压缩文件的*open()* 函数和*BZ2File* 类。
- 用于增量压缩和解压的*BZ2Compressor* 和*BZ2Decompressor* 类。
- 用于一次性压缩和解压的*compress()* 和*decompress()* 函数。

此模块中的所有类都能安全地从多个线程访问。

13.3.1 文件压缩和解压

bz2.**open**(*filename*, *mode='r'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)
以二进制或文本模式打开 bzip2 压缩文件，返回一个*file object*。

和*BZ2File* 的构造函数类似，*filename* 参数可以是一个实际的文件名（*str* 或*bytes* 对象），或是已有的可供读取或写入的文件对象。

mode 参数可设为二进制模式的 `'r'`、`'rb'`、`'w'`、`'wb'`、`'x'`、`'xb'`、`'a'` 或 `'ab'`，或者文本模式的 `'rt'`、`'wt'`、`'xt'` 或 `'at'`。默认是 `'rb'`。

compresslevel 参数是 1 到 9 的整数，和*BZ2File* 的构造函数一样。

对于二进制模式，这个函数等价于*BZ2File* 构造器：`BZ2File(filename, mode, compresslevel=compresslevel)`。在这种情况下，不可提供 *encoding*, *errors* 和 *newline* 参数。

对于文本模式，将会创建一个*BZ2File* 对象，并将它包装到一个*io.TextIOWrapper* 实例中，此实例带有指定的编码格式、错误处理行为和行结束符。

3.3 新版功能.

在 3.4 版更改: 添加了 `'x'` (仅创建) 模式。

在 3.6 版更改: 接受一个*path-like object*。

class bz2.**BZ2File**(*filename*, *mode='r'*, *buffering=None*, *compresslevel=9*)
用二进制模式打开 bzip2 压缩文件。

如果 *filename* 是一个*str* 或*bytes* 对象，则打开名称对应的文件目录。否则的话，*filename* 应当是一个*file object*，它将被用来读取或写入压缩数据。

mode 参数可以是表示读取的 `'r'` (默认值)，表示覆写的 `'w'`，表示单独创建的 `'x'`，或表示添加的 `'a'`。这些模式还可别以 `'rb'`, `'wb'`, `'xb'` 和 `'ab'` 的等价形式给出。

如果 *filename* 是一个文件对象（而不是实际的文件名），则 `'w'` 模式并不会截断文件，而是会等价于 `'a'`。

buffering 参数会被忽略。它已经被弃用。

If *mode* is `'w'` or `'a'`, *compresslevel* can be a number between `1` and `9` specifying the level of compression: `1` produces the least compression, and `9` (default) produces the most compression.

如果 *mode* 为 `'r'`，则输入文件可以为多个压缩流的拼接。

BZ2File 提供了*io.BufferedIOBase* 所指定的所有成员，但 `detach()` 和 `truncate()` 除外。并支持迭代和 `with` 语句。

BZ2File 还提供了以下方法：

peek([*n*])
返回缓冲的数据而不前移文件位置。至少将返回一个字节的数据（除非为 EOF）。实际返回的字节数不确定。

注解： 虽然调用*peek()* 不会改变*BZ2File* 的文件位置，但它可能改变下层文件对象的位置（举例来说如果*BZ2File* 是通过传为一个文件对象作为 *filename* 的话）。

3.3 新版功能.

在 3.1 版更改: 支持了 `with` 语句。

在 3.3 版更改: 添加了 `fileno()`, `readable()`, `seekable()`, `writable()`, `read1()` 和 `readinto()` 方法。

在 3.3 版更改: 添加了对 *filename* 使用*file object* 而非实际文件名的支持。

在 3.3 版更改: 添加了 `'a'` (append) 模式，以及对读取多数据流文件的支持。

在 3.4 版更改: 添加了 `'x'` (仅创建) 模式。

在 3.5 版更改: *read()* 方法现在接受 `None` 作为参数。

在 3.6 版更改: 接受一个*path-like object*。

13.3.2 增量压缩和解压

class bz2.**BZ2Compressor**(*compresslevel=9*)
创建一个新的压缩器对象。此对象可被用来执行增量数据压缩。对于一次性压缩，请改用*compress()* 函数。

compresslevel, if given, must be a number between `1` and `9`. The default is `9`.

compress(*data*)
向压缩器对象提供数据。在可能的情况下返回一段已压缩数据，否则返回空字节串。

当你已结束向压缩器提供数据时，请调用*flush()* 方法来完成压缩进程。

flush()
结束压缩进程，返回内部缓冲中剩余的压缩完成的数据。

调用此方法之后压缩器对象将不可再被使用。

class bz2.**BZ2Decompressor**
创建一个新的解压缩器对象。此对象可被用来执行增量数据解压缩。对于一次性解压缩，请改用*decompress()* 函数。

注解： 这个类不会透明地处理包含多个已压缩数据流的输入，这不同于*decompress()* 和*BZ2File*。如果你需要通过*BZ2Decompressor* 来解压缩多个数据流输入，你必须为每个数据流都使用新的解压缩器。

---

decompress(*data*, *max_length=-1*)
解压缩 *data* (一个*bytes-like object*)，返回字节串形式的解压缩数据。某些 *data* 可以在内部被缓冲，以便用于后续的*decompress()* 调用。返回的数据应当与之前任何*decompress()* 调用的输出进行拼接。

如果 *max_length* 为非负数，将返回至多 *max_length* 个字节的解压缩数据。如果达到此限制并且可以产生后续输出，则*needs_input* 属性将被设为 `False`。在这种情况下，下一次*decompress()* 调用提供的 *data* 可以为 `b''` 以获取更多的输出。

如果所有输入数据都已被解压缩并返回（或是因为它少于 *max_length* 个字节，或是因为 *max_length* 为负数），则*needs_input* 属性将被设为 `True`。

在到达数据流末尾之后再尝试解压缩数据会引发 *EOFError*。在数据流末尾之后获取的任何数据都会被忽略并存储至*unused_data* 属性。

在 3.5 版更改: 添加了 *max_length* 形参。

eof
若达到了数据流末尾标识符则为 `True`。

3.3 新版功能.

unused_data
压缩数据流的末尾还有数据。

如果在达到数据流末尾之前访问此属性，其值将为 `b''`。

needs_input
如果在要求新的未解压缩输入之前*decompress()* 方法可以提供更多的解压缩数据则为 `False`。

3.5 新版功能.

13.3.3 一次性压缩或解压

bz2.**compress**(*data*, *compresslevel=9*)
Compress *data*.

compresslevel, if given, must be a number between `1` and `9`. The default is `9`.

对于增量压缩，请改用*BZ2Compressor*。

bz2.**decompress**(*data*)
Decompress *data*.

如果 *data* 是多个压缩数据流的拼接，则解压缩所有数据流。

对于增量解压缩，请改用*BZ2Decompressor*。

在 3.3 版更改: 支持了多数据流的输入。

13.4 lzma —用 LZMA 算法压缩

3.3 新版功能.

源代码： Lib/lzma.py

此模块提供了可以压缩和解压缩使用 LZMA 压缩算法的数据的类和便携函数。其中还包含支持 **xz** 工具所使用的 `.xz` 和旧式 `.lzma` 文件格式的文件接口，以及相应的原始压缩数据流。

此模块所提供的接口与*bz2* 模块的非常类似。但是，请注意*LZMAFile* 不是线程安全的，这与*bz2.BZ2File* 不同，因此如果你需要在多个线程中使用单个*LZMAFile* 实例，则需要通过锁来保护它。

exception lzma.**LZMAError**
当在压缩或解压缩期间或是在初始化压缩器/解压缩器的状态期间发生错误时此异常会被引发。

13.4.1 读写压缩文件

lzma.**open**(*filename*, *mode="rb"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*, *encoding=None*, *errors=None*, *newline=None*)

以二进制或文本模式打开 LZMA 压缩文件，返回一个*file object*。

filename 参数可以是一个实际的文件名（以*str*, *bytes* 或*路径类* 对象的形式给出），在此情况下会打开指定名称的文件，或者可以是一个用于读写的现有文件对象。

mode 参数可以是二进制模式的 `"r"`, `"rb"`, `"w"`, `"wb"`, `"x"`, `"xb"`, `"a"` 或 `"ab"`，或者文本模式的 `"rt"`, `"wt"`, `"xt"` 或 `"at"`。默认值为 `"rb"`。

当打开一个文件用于读取时，*format* 和 *filters* 参数具有与*LZMADecompressor* 的参数相同的含义。在此情况下，*check* 和 *preset* 参数不应被使用。

当打开一个文件用于写入的，*format*, *check*, *preset* 和 *filters* 参数具有与*LZMACompressor* 的参数相同的含义。

对于二进制模式，这个函数等价于*LZMAFile* 构造器: `LZMAFile(filename, mode, ...)`。在这种情况下，不可提供 *encoding*, *errors* 和 *newline* 参数。

对于文本模式，将会创建一个*LZMAFile* 对象，并将它包装到一个*io.TextIOWrapper* 实例中，此实例带有指定的编码格式、错误处理行为和行结束符。

在 3.4 版更改: 增加了对 `"x"`, `"xb"` 和 `"xt"` 模式的支持。

在 3.6 版更改: 接受一个*path-like object*。

class lzma.**LZMAFile**(*filename=None*, *mode="r"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*)

以二进制模式打开一个 LZMA 压缩文件。

LZMAFile 可以包装在一个已打开的*file object* 中，或者是在给定名称的文件上直接操作。*filename* 参数指定所包装的文件对象，或是要打开的文件名称（类型为*str*, *bytes* 或*路径类* 对象）。如果是包装现有的文件对象，被包装的文件在*LZMAFile* 被关闭时将不会被关闭。

mode 参数可以是表示读取的 `"r"` (默认值)，表示覆写的 `"w"`，表示单独创建的 `"x"`，或表示添加的 `"a"`。这些模式还可以分别以 `"rb"`, `"wb"`, `"xb"` 和 `"ab"` 的等价形式给出。

如果 *filename* 是一个文件对象（而不是实际的文件名），则 `"w"` 模式并不会截断文件，而会等价于 `"a"`。

当打开一个文件用于读取时，输入文件可以为多个独立压缩流的拼接。它们会被作为单个逻辑流被透明地解码。

当打开一个文件用于读取时，*format* 和 *filters* 参数具有与*LZMADecompressor* 的参数相同的含义。在此情况下，*check* 和 *preset* 参数不应被使用。

当打开一个文件用于写入的，*format*, *check*, *preset* 和 *filters* 参数具有与*LZMACompressor* 的参数相同的含义。

LZMAFile 支持*io.BufferedIOBase* 所指定的所有成员，但 `detach()` 和 `truncate()` 除外。并支持迭代和 `with` 语句。

也提供以下方法：

peek(*size=-1*)

返回缓冲的数据而不前移文件位置。至少将返回一个字节的数据，除非已经到达 EOF。实际返回的字节数不确定（会忽略 *size* 参数）。

> **注解：** 虽然调用*peek()* 不会改变*LZMAFile* 的文件位置，但它可能改变下层文件对象的位置（举例来说如果*LZMAFile* 是通过传入一个文件对象作为 *filename* 的话）。

在 3.4 版更改: 增加了对 "x" 和 "xb" 模式的支持。

在 3.5 版更改: *read()* 方法现在接受 None 作为参数。

在 3.6 版更改: 接受一个*path-like object*。

13.4.2 在内存中压缩和解压缩数据

class lzma.**LZMACompressor**(*format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

创建一个压缩器对象，此对象可被用来执行增量压缩。

压缩单个数据块的更便捷方式请参阅*compress()*。

format 参数指定应当使用哪种压缩格式。可能的值有:

- **FORMAT_XZ: .xz 容器格式**。 这是默认格式。
- **FORMAT_ALONE: 传统的 .lzma 容器格式**。 这种格式相比 .xz 更为受限–它不支持一致性检查或多重过滤器。
- **FORMAT_RAW: 原始数据流，不使用任何容器格式**。 这个格式描述器不支持一致性检查，并且要求你必须指定一个自定义的过滤器链（用于压缩和解压缩）。此外，以这种方式压缩的数据不可使用 FORMAT_AUTO 来解压缩 (参见*LZMADecompressor*)。

check 参数指定要包含在压缩数据中的一致性检查类型。这种检查在解压缩时使用，以确保数据没有被破坏。可能的值是:

- CHECK_NONE: 没有一致性检查。这是 FORMAT_ALONE 和 FORMAT_RAW 的默认值（也是唯一可接受的值）。
- CHECK_CRC32: 32 位循环冗余检查。
- CHECK_CRC64: 64 位循环冗余检查。这是 FORMAT_XZ 的默认值。
- CHECK_SHA256: 256 位安全哈希算法。

如果指定的检查不受支持，则会引发*LZMAError*。

压缩设置可被指定为一个预设的压缩等级（通过 *preset* 参数）或以自定义过滤器链来详细设置（通过 *filters* 参数）。

preset 参数（如果提供）应当为一个 0 到 9 (包括边界) 之间的整数，可以选择与常数 PRESET_EXTREME 进行 OR 运算。如果 *preset* 和 *filters* 均未给出，则默认行为是使用 PRESET_DEFAULT (预设等级 6)。更高的预设等级会产生更小的输出，但会使得压缩过程更缓慢。

注解: 除了更加 CPU 密集，使用更高的预设等级来压缩还需要更多的内存（并产生需要更多内存来解压缩的输出）。例如使用预设等级 9 时，一个*LZMACompressor* 对象的开销可以高达 800 MiB。出于这样的原因，通常最好是保持使用默认预设等级。

---

filters 参数（如果提供）应当指定一个过滤器链。详情参见指定自定义的过滤器链。

compress(*data*)

压缩 *data* (一个*bytes* object)，返回包含针对输入的至少一部分已压缩数据的*bytes* 对象。一部 *data* 可能会被放入内部缓冲区，以便用于后续的*compress()* 和*flush()* 调用。返回的数据应当与之前任何*compress()* 调用的输出进行拼接。

flush()

结束压缩进程，返回包含保存在压缩器的内部缓冲区中的任意数据的*bytes* 对象。

调用此方法之后压缩器将不可再被使用。

class lzma.**LZMADecompressor**(*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)
创建一个压缩器对象，此对象可被用来执行增量解压缩。

一次性解压缩整个压缩数据流的更便捷方式请参阅*decompress()*。

format 参数指定应当被使用的容器格式。默认值为 FORMAT_AUTO，它可以解压缩 .xz 和 .lzma 文件。其他可能的值为 FORMAT_XZ, FORMAT_ALONE 和 FORMAT_RAW。

memlimit 参数指定解压缩器可以使用的内存上限（字节数）。当使用此参数时，如果不可能在给定内存上限之内解压缩输入数据则解压缩将失败并引发*LZMAError*。

filters 参数指定用于创建被解压缩数据流的过滤器链。此参数在 *format* 为 FORMAT_RAW 时要求提供，但对于其他格式不应使用。有关过滤器链的更多信息请参阅指定自定义的过滤器链。

注解： 这个类不会透明地处理包含多个已压缩数据流的输入，这不同于*decompress()* 和*LZMAFile*。要通过*LZMADecompressor* 来解压缩多个数据流输入，你必须为每个数据流都创建一个新的解压缩器。

---

decompress(*data*, *max_length=-1*)
解压缩 *data* (一个*bytes-like object*)，返回字节串形式的解压缩数据。某些 *data* 可以在内部被缓冲，以便用于后续的*decompress()* 调用。返回的数据应当与之前任何*decompress()* 调用的输出进行拼接。

如果 *max_length* 为非负数，将返回至多 *max_length* 个字节的解压缩数据。如果达到此限制并且可以产生后续输出，则*needs_input* 属性将被设为 False。在这种情况下，下一次*decompress()* 调用提供的 *data* 可以为 b'' 以获取更多的输出。

如果所有输入数据都已被解压缩并返回（或是因为它少于 *max_length* 个字节，或是因为 *max_length* 为负数），则*needs_input* 属性将被设为 True。

在到达数据流末尾之后再尝试解压缩数据会引发 *EOFError*。在数据流末尾之后获取的任何数据都会被忽略并存储至*unused_data* 属性。

在 3.5 版更改: 添加了 *max_length* 形参。

check
输入流使用的一致性检查的 ID。这可能为 CHECK_UNKNOWN 直到已解压了足够的输入数据来确定它所使用的一致性检查。

eof
若达到了数据流末尾标识符则为 True。

unused_data
压缩数据流的末尾还有数据。

在达到数据流末尾之前，这个值将为 b""。

needs_input
如果在要求新的未解压缩输入之前*decompress()* 方法可以提供更多的解压缩数据则为 False。

3.5 新版功能.

lzma.**compress**(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)
压缩 *data* (一个*bytes* 对象)，返回包含压缩数据的*bytes* 对象。

参见上文的*LZMACompressor* 了解有关 *format*, *check*, *preset* 和 *filters* 参数的说明。

lzma.**decompress**(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)
解压缩 *data* (一个*bytes* 对象)，返回包含解压缩数据的*bytes* 对象。

如果 *data* 是多个单独压缩数据流的拼接，则解压缩所有相应数据流，并返回结果的拼接。

参见上文的*LZMADecompressor* 了解有关 *format*, *memlimit* 和 *filters* 参数的说明。

13.4.3 杂项

lzma.**is_check_supported**(*check*)
Returns true if the given integrity check is supported on this system.

CHECK_NONE 和 CHECK_CRC32 总是受支持。CHECK_CRC64 和 CHECK_SHA256 或许不可用，如果你正在使用基于受限制特性集编译的 **liblzma** 版本的话。

13.4.4 指定自定义的过滤器链

过滤器链描述符是由字典组成的序列，其中每个字典包含单个过滤器的 ID 和选项。每个字典必须包含键 "id"，并可能包含额外的键用来指定基于过滤器的选项。有效的过滤器 ID 如下：

- **压缩过滤器：**
  - FILTER_LZMA1 (配合 FORMAT_ALONE 使用)
  - FILTER_LZMA2 (配合 FORMAT_XZ 和 FORMAT_RAW 使用)
- **增量过滤器：**
  - FILTER_DELTA
- **Branch-Call-Jump (BCJ) 过滤器：**
  - FILTER_X86
  - FILTER_IA64
  - FILTER_ARM
  - FILTER_ARMTHUMB
  - FILTER_POWERPC
  - FILTER_SPARC

一个过滤器链最多可由 4 个过滤器组成，并且不能为空。过滤器链中的最后一个过滤器必须为压缩过滤器，其他过滤器必须为 Delta 或 BCJ 过滤器。

压缩过滤器支持下列选项（指定为表示过滤器的字典中的附加条目）：

- preset: 压缩预设选项，用于作为未显式指定的选项的默认值的来源。
- dict_size: 以字节表示的字典大小。这应当在 4 KiB 和 1.5 GiB 之间（包含边界）。
- lc: 字面值上下文的位数。
- lp: 字面值位置的比特数。总计值 lc + lp 必须不大于 4。
- pb: 位置的比特数；必须不大于 4。
- mode: MODE_FAST 或 MODE_NORMAL。
- nice_len: 对于一个匹配应当被视为“适宜长度”的值。这应当小于或等于 273。
- mf: 要使用的匹配查找器–MF_HC3, MF_HC4, MF_BT2, MF_BT3 或 MF_BT4。
- depth: 匹配查找器使用的最大查找深度。0 (默认值) 表示基于其他过滤器选项自动选择。

Delta 过滤器保存字节数据之间的差值，在特定环境下可产生更具重复性的输入。它支持一个 dist 选项，指明要减去的字节之间的差值大小。默认值为 1，即相邻字节之间的差值。

BCJ 过滤器主要作用于机器码。它们会转换机器码内的相对分支、调用和跳转以使用绝对寻址，其目标是提升冗余度以供压缩器利用。这些过滤器支持一个 start_offset 选项，指明应当被映射到输入数据开头的地址。默认值为 0。

13.4.5 例子

在已压缩的数据中读取:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

创建一个压缩文件:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

在内存中压缩文件:

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

增量压缩:

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

写入已压缩数据到已打开的文件:

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

使用自定义过滤器链创建一个已压缩文件:

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.5 zipfile —使用 ZIP 存档

源代码： Lib/zipfile.py

ZIP 文件格式是一个常用的归档与压缩标准。这个模块提供了创建、读取、写入、添加及列出 ZIP 文件的工具。任何对此模块的进阶使用都将需要理解此格式，其定义参见 PKZIP 应用程序笔记。

此模块目前不能处理分卷 ZIP 文件。它可以处理使用 ZIP64 扩展（超过 4 GB 的 ZIP 文件）的 ZIP 文件。它支持解密 ZIP 归档中的加密文件，但是目前不能创建一个加密的文件。解密非常慢，因为它是使用原生 Python 而不是 C 实现的。

这个模块定义了以下内容：

exception zipfile.**BadZipFile**
为损坏的 ZIP 文件抛出的错误。

3.2 新版功能.

exception zipfile.**BadZipfile**
BadZipFile 的别名，与旧版本 Python 保持兼容性。

3.2 版后已移除.

exception zipfile.**LargeZipFile**
当 ZIP 文件需要 ZIP64 功能但是未启用时会抛出此错误。

class zipfile.**ZipFile**
用于读写 ZIP 文件的类。欲了解构造函数的描述，参阅段落*ZipFile 对象*。

class zipfile.**PyZipFile**
用于创建包含 Python 库的 ZIP 归档的类。

class zipfile.**ZipInfo**(*filename='NoName'*, *date_time=(1980, 1, 1, 0, 0, 0)*)
用于表示档案内一个成员信息的类。此类的实例会由*ZipFile* 对象的*getinfo()* 和*infolist()* 方法返回。大多数*zipfile* 模块的用户都不必创建它们，只需使用此模块所创建的实例。*filename* 应当是档案成员的全名，*date_time* 应当是包含六个字段的描述最近修改时间的元组；这些字段的描述请参阅*ZipInfo 对象*。

zipfile.**is_zipfile**(*filename*)
根据文件的 Magic Number，如果 *filename* 是一个有效的 ZIP 文件则返回 `True`，否则返回 `False`。*filename* 也可能是一个文件或类文件对象。

在 3.1 版更改: 支持文件或类文件对象。

zipfile.**ZIP_STORED**
未被压缩的归档成员的数字常数。

zipfile.**ZIP_DEFLATED**
常用的 ZIP 压缩方法的数字常数。需要*zlib* 模块。

zipfile.**ZIP_BZIP2**
BZIP2 压缩方法的数字常数。需要*bz2* 模块。

3.3 新版功能.

zipfile.**ZIP_LZMA**
LZMA 压缩方法的数字常数。需要*lzma* 模块。

3.3 新版功能.

注解： ZIP 文件格式规范包括自 2001 年以来对 bzip2 压缩的支持，以及自 2006 年以来对 LZMA 压缩的支持。但是，一些工具（包括较旧的 Python 版本）不支持这些压缩方法，并且可能拒绝完全处理 ZIP 文件，或者无法提取单个文件。

参见：

PKZIP 应用程序笔记 Phil Katz 编写的 ZIP 文件格式文档，此格式和使用的算法的创建者。

Info-ZIP 主页 有关 Info-ZIP 项目的 ZIP 存档程序和开发库的信息。

13.5.1 ZipFile 对象

class zipfile.**ZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*)

Open a ZIP file, where *file* can be a path to a file (a string), a file-like object or a *path-like object*. The *mode* parameter should be `'r'` to read an existing file, `'w'` to truncate and write a new file, `'a'` to append to an existing file, or `'x'` to exclusively create and write a new file. If *mode* is `'x'` and *file* refers to an existing file, a *FileExistsError* will be raised. If *mode* is `'a'` and *file* refers to an existing ZIP file, then additional files are added to it. If *file* does not refer to a ZIP file, then a new ZIP archive is appended to the file. This is meant for adding a ZIP archive to another file (such as `python.exe`). If *mode* is `'a'` and the file does not exist at all, it is created. If *mode* is `'r'` or `'a'`, the file should be seekable. *compression* is the ZIP compression method to use when writing the archive, and should be *ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA*; unrecognized values will cause *NotImplementedError* to be raised. If *ZIP_DEFLATED*, *ZIP_BZIP2* or *ZIP_LZMA* is specified but the corresponding module (*zlib*, *bz2* or *lzma*) is not available, *RuntimeError* is raised. The default is *ZIP_STORED*. If *allowZip64* is `True` (the default) zipfile will create ZIP files that use the ZIP64 extensions when the zipfile is larger than 4 GiB. If it is false *zipfile* will raise an exception when the ZIP file would require ZIP64 extensions.

如果创建文件时使用 `'w'`, `'x'` 或 `'a'` 模式并且未向归档添加任何文件就执行了*closed*，则会将适当的空归档 ZIP 结构写入文件。

ZipFile is also a context manager and therefore supports the `with` statement. In the example, *myzip* is closed after the `with` statement's suite is finished—even if an exception occurs:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

3.2 新版功能: 添加了将*ZipFile* 用作上下文管理员的功能。

在 3.3 版更改: 添加了对*bzip2* 和*lzma* 压缩的支持。

在 3.4 版更改: 默认启用 ZIP64 扩展。

在 3.5 版更改: 添加了对不可查找数据流的支持。并添加了对 `'x'` 模式的支持。

在 3.6 版更改: 在此之前，对于不可识别的压缩值将引发普通的*RuntimeError*。

在 3.6.2 版更改: *file* 形参接受一个*path-like object*。

ZipFile.**close**()

关闭归档文件。你必须在退出程序之前调用*close()* 否则将不会写入关键记录数据。

ZipFile.**getinfo**(*name*)

返回一个*ZipInfo* 对象，其中包含有关归档成员 *name* 的信息。针对一个目前并不包含于归档中的名称调用*getinfo()* 将会引发*KeyError*。

ZipFile.**infolist**()

返回一个列表，其中包含每个归档成员的*ZipInfo* 对象。如果是打开一个现有归档则这些对象的排列顺序与它们对应条目在磁盘上的实际 ZIP 文件中的顺序一致。

ZipFile.**namelist**()
返回按名称排序的归档成员列表。

ZipFile.**open**(*name*, *mode='r'*, *pwd=None*, *, *force_zip64=False*)
以二进制文件类对象的形式访问一个归档成员。*name* 可以是归档内某个文件的名称也可以是某个*ZipInfo* 对象。如果包含了 *mode* 形参，则它必须为 'r' (默认值) 或 'w'。*pwd* 为用于解密已加密 ZIP 文件的密码。

open() 也是一个上下文管理器，因此支持 with 语句:

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

With *mode* 'r' the file-like object (ZipExtFile) is read-only and provides the following methods: *read()*, *readline()*, *readlines()*, __iter__(), *__next__()*. These objects can operate independently of the ZipFile.

如果 mode='w' 则返回一个可写入的文件句柄，它将支持*write()* 方法。当一个可写入的文件句柄被打开时，尝试读写 ZIP 文件中的其他文件将会引发*ValueError*。

当写入一个文件时，如果文件大小不能预先确定但是可能超过 2 GiB，可传入 force_zip64=True 以确保标头格式能够支持超大文件。如果文件大小可以预先确定，则在构造*ZipInfo* 对象时应设置*file_size*，并将其用作 *name* 形参。

注解: *open()*, *read()* 和*extract()* 方法可接受文件名或*ZipInfo* 对象。当尝试读取一个包含重复名称成员的 ZIP 文件时你将发现此功能很有好处。

在 3.6 版更改: 移除了对 mode='U' 的支持。请使用*io.TextIOWrapper* 以在*universal newlines* 模式中读取已压缩的文本文件。

在 3.6 版更改: *open()* 现在可以被用来配合 mode='w' 选项来将文件写入归档。

在 3.6 版更改: 在已关闭的 ZipFile 上调用*open()* 将引发*ValueError*。在之前的版本中则会引发*RuntimeError*。

ZipFile.**extract**(*member*, *path=None*, *pwd=None*)
从归档中提取出一个成员放入当前工作目录；*member* 必须为成员的完整名称或*ZipInfo* 对象。成员的文件信息会尽可能精确地被提取。*path* 指定一个要提取到的不同目录。*member* 可以是一个文件名或*ZipInfo* 对象。*pwd* 是用于解密文件的密码。

返回所创建的经正规化的路径（对应于目录或新文件）。

注解: 如果一个成员文件名为绝对路径，则将去掉驱动器/UNC 共享点和前导的（反）斜杠，例如: ///foo/bar 在 Unix 上将变为 foo/bar，而 C:\foo\bar 在 Windows 上将变为 foo\bar。并且一个成员文件名中的所有 ".." 都将被移除，例如: ../../foo../../ba..r 将变为 foo../ba..r。在 Windows 上非法字符 (:, <, >, |, ", ?, and *) 会被替换为下划线 (_)。

在 3.6 版更改: 在已关闭的 ZipFile 上调用*extract()* 将引发*ValueError*。在之前的版本中则将引发*RuntimeError*。

在 3.6.2 版更改: *path* 形参接受一个*path-like object*。

ZipFile.**extractall**(*path=None*, *members=None*, *pwd=None*)
从归档中提取出所有成员放入当前工作目录。*path* 指定一个要提取到的不同目录。*members* 为可选项且必须为*namelist()* 所返回列表的一个子集。*pwd* 是用于解密文件的密码。

> **警告：** 绝不要未经预先检验就从不可靠的源中提取归档文件。这样有可能在 *path* 之外创建文件，例如某些成员具有以 "/" 开始的文件名或带有两个点号 ".." 的文件名。此模块会尝试防止这种情况。参见 *extract()* 的注释。

在 3.6 版更改: 在已关闭的 ZipFile 上调用 *extractall()* 将引发 *ValueError*。在之前的版本中则将引发 *RuntimeError*。

在 3.6.2 版更改: *path* 形参接受一个 *path-like object*。

ZipFile.**printdir**()
将归档的目录表打印到 sys.stdout。

ZipFile.**setpassword**(*pwd*)
设置 *pwd* 为用于提取已加密文件的默认密码。

ZipFile.**read**(*name*, *pwd=None*)
返回归档中文件 *name* 的字节数据。*name* 是归档中文件的名称，或是一个 *ZipInfo* 对象。归档必须以读取或追加方式打开。*pwd* 为用于已加密文件的密码，并且如果指定该参数则它将覆盖通过 *setpassword()* 设置的默认密码。on a ZipFile that uses a compression method 在使用 *ZIP_STORED*，*ZIP_DEFLATED*, *ZIP_BZIP2* 或 *ZIP_LZMA* 以外的压缩方法的 ZipFile 上调用 *read()* 将引发 *NotImplementedError*。如果相应的压缩模块不可用也会引发错误。

在 3.6 版更改: 在已关闭的 ZipFile 上调用 *read()* 将引发 *ValueError*。在之前的版本中则会引发 *RuntimeError*.

ZipFile.**testzip**()
读取归档中的所有文件并检查它们的 CRC 和文件头。返回第一个已损坏文件的名称，在其他情况下则返回 None。

在 3.6 版更改: 在已关闭的 ZipFile 上调用 *testzip()* 将引发 *ValueError*。在之前的版本中则将引发 *RuntimeError*。

ZipFile.**write**(*filename*, *arcname=None*, *compress_type=None*)
Write the file named *filename* to the archive, giving it the archive name *arcname* (by default, this will be the same as *filename*, but without a drive letter and with leading path separators removed). If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry. The archive must be open with mode 'w', 'x' or 'a'.

注解： 归档名称应当是基于归档根目录的相对路径，也就是说，它们不应以路径分隔符开头。

注解： 如果 arcname (或 filename，如果 arcname 未给出) 包含一个空字节，则归档中该文件的名称将在空字节位置被截断。

在 3.6 版更改: 在使用 'r' 模式创建的 ZipFile 或已关闭的 ZipFile 上调用 *write()* 将引发 *ValueError*。在之前的版本中则会引发 *RuntimeError*。

ZipFile.**writestr**(*zinfo_or_arcname*, *data*[, *compress_type*])
将一个文件写入归档。内容为 *data*，它可以是一个 *str* 或 *bytes* 的实例；如果是 *str*，则会先使用 UTF-8 进行编码。*zinfo_or_arcname* 可以是它在归档中将被给予的名称，或者是 *ZipInfo* 的实例。如果它是一个实例，则至少必须给定文件名、日期和时间。如果它是一个名称，则日期和时间会被设为当前日期和时间。归档必须以 'w', 'x' 或 'a' 模式打开。

If given, *compress_type* overrides the value given for the *compression* parameter to the constructor for the new entry, or in the *zinfo_or_arcname* (if that is a *ZipInfo* instance).

注解: 当传入一个*ZipInfo* 实例作为 *zinfo_or_arcname* 形参时，所使用的压缩方法将为在给定的*ZipInfo* 实例的 *compress_type* 成员中指定的方法。默认情况下，*ZipInfo* 构造器将将此成员设为*ZIP_STORED*。

在 3.2 版更改: *compress_type* 参数。

在 3.6 版更改: 在使用 `'r'` 模式创建的 ZipFile 或已关闭的 ZipFile 上调用*writestr()* 将引发*ValueError*。在之前的版本中则会引发*RuntimeError*。

以下数据属性也是可用的:

`ZipFile.`**`filename`**
ZIP 文件的名称。

`ZipFile.`**`debug`**
要使用的调试输出等级。这可以设为从 `0` (默认无输出) 到 `3` (最多输出) 的值。调试信息会被写入 `sys.stdout`。

`ZipFile.`**`comment`**
关联到 ZIP 文件的*bytes* 对象形式的说明。如果将说明赋给以 `'w'`, `'x'` 或 `'a'` 模式创建的*ZipFile* 实例，它的长度不应超过 65535 字节。超过此长度的说明将被截断。

13.5.2 PyZipFile 对象

PyZipFile 构造器接受与*ZipFile* 构造器相同的形参，以及一个额外的形参 *optimize*。

class `zipfile.`**`PyZipFile`**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *optimize=-1*)
3.2 新版功能: *optimize* 形参。

在 3.4 版更改: 默认启用 ZIP64 扩展。

实例在*ZipFile* 对象所具有的方法以外还附加了一个方法:

writepy(*pathname*, *basename=''*, *filterfunc=None*)
查找 `*.py` 文件并将相应的文件添加到归档。

如果*PyZipFile* 的 *optimize* 形参未给定或为 `-1`，则相应的文件为 `*.pyc` 文件，并在必要时进行编译。

如果*PyZipFile* 的 *optimize* 形参为 `0`, `1` 或 `2`，则限具有相应优化级别 (参见*compile()*) 的文件会被添加到归档，并在必要时进行编译。

If *pathname* is a file, the filename must end with `.py`, and just the (corresponding `*.pyc`) file is added at the top level (no path information). If *pathname* is a file that does not end with `.py`, a *RuntimeError* will be raised. If it is a directory, and the directory is not a package directory, then all the files `*.pyc` are added at the top level. If the directory is a package directory, then all `*.pyc` are added under the package name as a file path, and if any subdirectories are package directories, all of these are added recursively.

basename 仅限在内部使用。

如果给定 *filterfunc*，则它必须是一个接受单个字符串参数的函数。在将其添加到归档之前它将被传入每个路径（包括每个单独的完整路径）。如果 *filterfunc* 返回假值，则路径将不会被添加，而如果它是一个目录则其内容将被忽略。例如，如果我们的测试文件全都位于 `test` 目录或以字符串 `test_` 打头，则我们可以使用一个 *filterfunc* 来排除它们:

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
```

(下页继续)

(续上页)

```
...     return (not (fn == 'test' or fn.startswith('test_')))
>>> zf.writepy('myprog', filterfunc=notests)
```

writepy() 方法会产生带有这样一些文件名的归档:

```
string.pyc                      # Top level name
test/__init__.pyc               # Package directory
test/testall.pyc                # Module test.testall
test/bogus/__init__.pyc         # Subpackage directory
test/bogus/myfile.pyc           # Submodule test.bogus.myfile
```

3.4 新版功能: *filterfunc* 形参。

在 3.6.2 版更改: *pathname* 形参接受一个*path-like object*。

13.5.3 ZipInfo 对象

ZipInfo 类的实例会通过*getinfo()* 和*ZipFile* 对象的*infolist()* 方法返回。每个对象将存储关于 ZIP 归档的一个成员的信息。

有一个类方法可以为文件系统文件创建*ZipInfo* 实例:

classmethod ZipInfo.**from_file**(*filename*, *arcname=None*)

为文件系统中的文件构造一个*ZipInfo* 实例，并准备将其添加到一个 zip 文件。

filename 应为文件系统中某个文件或目录的路径。

如果指定了 *arcname*，它会被用作归档中的名称。如果未指定 *arcname*，则所用名称与 *filename* 相同，但将去除任何驱动器盘符和打头的路径分隔符。

3.6 新版功能.

在 3.6.2 版更改: *filename* 形参接受一个*path-like object*。

实例具有下列方法和属性:

ZipInfo.**is_dir**()

如果此归档成员是一个目录则返回 True。

这会使用条目的名称：目录应当总是以 / 结尾。

3.6 新版功能.

ZipInfo.**filename**

归档中的文件名称。

ZipInfo.**date_time**

上次修改存档成员的时间和日期。这是六个值的元组：

索引	值
0	Year (>= 1980)
1	月（1 为基数）
2	月份中的日期（1 为基数）
3	小时（0 为基数）
4	分钟（0 为基数）
5	秒（0 为基数）

注解： ZIP 文件格式不支持 1980 年以前的时间戳。

ZipInfo.**compress_type**
归档成员的压缩类型。

ZipInfo.**comment**
bytes 对象形式的单个归档成员的注释。

ZipInfo.**extra**
扩展字段数据。PKZIP Application Note 包含一些保存于该*bytes* 对象中的内部结构的注释。

ZipInfo.**create_system**
创建 ZIP 归档所用的系统。

ZipInfo.**create_version**
创建 ZIP 归档所用的 PKZIP 版本。

ZipInfo.**extract_version**
需要用来提取归档的 PKZIP 版本。

ZipInfo.**reserved**
必须为零。

ZipInfo.**flag_bits**
ZIP 标志位。

ZipInfo.**volume**
文件中的分卷号。

ZipInfo.**internal_attr**
内部属性。

ZipInfo.**external_attr**
外部文件属性。

ZipInfo.**header_offset**
文件头的字节偏移量。

ZipInfo.**CRC**
未压缩文件的 CRC-32。

ZipInfo.**compress_size**
已压缩数据的大小。

ZipInfo.**file_size**
未压缩文件的大小。

13.5.4 命令行界面

zipfile 模块提供了简单的命令行接口用于与 ZIP 归档的交互。

如果你想要创建一个新的 ZIP 归档，请在*-c* 选项后指定其名称然后列出应当被包含的文件名:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

传入一个字典也是可接受的:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

如果你想要将一个 ZIP 归档提取到指定的目录，请使用 `-e` 选项:

```
$ python -m zipfile -e monty.zip target-dir/
```

对于一个 ZIP 归档中的文件列表，请使用 `-l` 选项:

```
$ python -m zipfile -l monty.zip
```

命令行选项

-l <zipfile>
列出一个 zipfile 中的文件名。

-c <zipfile> <source1> ... <sourceN>
基于源文件创建 zipfile。

-e <zipfile> <output_dir>
将 zipfile 提取到目标目录中。

-t <zipfile>
检测 zipfile 是否有效。

13.6 tarfile —读写 tar 归档文件

源代码: Lib/tarfile.py

`tarfile` 模块可以用来读写 tar 归档，包括使用 gzip, bz2 和 lzma 压缩的归档。请使用 `zipfile` 模块来读写 `.zip` 文件，或者使用 *shutil* 的高层级函数。

一些事实和数字:

- 读写 `gzip`, `bz2` 和 `lzma` 解压的归档要求相应的模块可用。
- 支持读取 / 写入 POSIX.1-1988 (ustar) 格式。
- 对 GNU tar 格式的读/写支持，包括 *longname* 和 *longlink* 扩展，对所有种类 *sparse* 扩展的只读支持，包括 sparse 文件的恢复。
- 对 POSIX.1-2001 (pax) 格式的读/写支持。
- 处理目录、正常文件、硬链接、符号链接、fifo 管道、字符设备和块设备，并且能够获取和恢复文件信息例如时间戳、访问权限和所有者等。

在 3.3 版更改: 添加了对 `lzma` 压缩的支持。

tarfile.**open**(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)
针对路径名 *name* 返回 `TarFile` 对象。有关 `TarFile` 对象以及所允许的关键字参数的详细信息请参阅 *TarFile 对象*。

mode 必须是 `'filemode[:compression]'` 形式的字符串，其默认值为 `'r'`。以下是模式组合的完整列表:

模式	action
`'r' or 'r:*'`	打开和读取使用透明压缩（推荐）。
`'r:'`	打开和读取不使用压缩。
`'r:gz'`	打开和读取使用 gzip 压缩。
`'r:bz2'`	打开和读取使用 bzip2 压缩。
`'r:xz'`	打开和读取使用 lzma 压缩。
`'x' or 'x:'`	创建 tarfile 不进行压缩。如果文件已经存在，则抛出*FileExistsError* 异常。
`'x:gz'`	使用 gzip 压缩创建 tarfile。如果文件已经存在，则抛出*FileExistsError* 异常。
`'x:bz2'`	使用 bzip2 压缩创建 tarfile。如果文件已经存在，则抛出*FileExistsError* 异常。
`'x:xz'`	使用 lzma 压缩创建 tarfile。如果文件已经存在，则抛出*FileExistsError* 异常。
`'a' or 'a:'`	打开以便在没有压缩的情况下追加。如果文件不存在，则创建该文件。
`'w' or 'w:'`	打开用于未压缩的写入。
`'w:gz'`	打开用于 gzip 压缩的写入。
`'w:bz2'`	打开用于 bzip2 压缩的写入。
`'w:xz'`	打开用于 lzma 压缩的写入。

请注意 `'a:gz'`, `'a:bz2'` 或 `'a:xz'` 是不可能的组合。如果 *mode* 不适用于打开特定（压缩的）文件用于读取，则会引发*ReadError*。请使用 *mode* `'r'` 来避免这种情况。如果某种压缩方法不受支持，则会引发*CompressionError*。

如果指定了 *fileobj*，它会被用作对应于 *name* 的以二进制模式打开的*file object* 的替代。它会被设定为处在位置 0。

对于 `'w:gz'`, `'r:gz'`, `'w:bz2'`, `'r:bz2'`, `'x:gz'`, `'x:bz2'` 等模式，*tarfile.open()* 接受关键字参数 *compresslevel* (默认值为 9) 来指定文件的压缩等级。

针对特殊的目的，还存在第二种 *mode* 格式: `'filemode|[compression]'`。*tarfile.open()* 将返回一个将其数据作为数据块流来处理的*TarFile* 对象。对此文件将不能执行随机查找。如果给定了 *fileobj*，它可以是任何具有 `read()` 或 `write()` 方法 (由 *mode* 确定) 的对象。*bufsize* 指定块大小，默认值为 `20 * 512` 字节。可与此格式组合使用的有 `sys.stdin`, 套接字*file object* 或磁带设备等。但是，对于这样的*TarFile* 对象存在不允许随机访问的限制，参见例子。目前可用的模式如下:

模式	动作
`'r|*'`	打开 tar 块的 流以进行透明压缩读取。
`'r|'`	打开一个未压缩的 tar 块的 *stream* 用于读取。
`'r|gz'`	打开一个 gzip 压缩的 *stream* 用于读取。
`'r|bz2'`	打开一个 bzip2 压缩的 *stream* 用于读取。
`'r|xz'`	打开一个 lzma 压缩 *stream* 用于读取。
`'w|'`	打开一个未压缩的 *stream* 用于写入。
`'w|gz'`	打开一个 gzip 压缩的 *stream* 用于写入。
`'w|bz2'`	打开一个 bzip2 压缩的 *stream* 用于写入。
`'w|xz'`	打开一个 lzma 压缩的 *stream* 用于写入。

在 3.5 版更改: 添加了 `'x'` (仅创建) 模式。

在 3.6 版更改: *name* 形参接受一个*path-like object*。

class tarfile.**TarFile**

用于读取和写入 tar 归档的类。请不要直接使用这个类：而要使用*tarfile.open()*。参见*TarFile 对象*。

tarfile.**is_tarfile**(*name*)
如果 *name* 是一个*tarfile* 模块能读取的 tar 归档文件则返回*True*。

tarfile 模块定义了以下异常:

exception tarfile.**TarError**
所有*tarfile* 异常的基类。

exception tarfile.**ReadError**
当一个不能被*tarfile* 模块处理或者因某种原因而无效的 tar 归档被打开时将被引发。

exception tarfile.**CompressionError**
当一个压缩方法不受支持或者当数据无法被正确解码时将被引发。

exception tarfile.**StreamError**
当达到流式*TarFile* 对象的典型限制时将被引发。

exception tarfile.**ExtractError**
当使用*TarFile.extract()* 时针对 *non-fatal* 所引发的异常，但是仅限 TarFile.errorlevel==2。

exception tarfile.**HeaderError**
如果获取的缓冲区无效则会由*TarInfo.frombuf()* 引发的异常。

以下常量在模块层级上可用:

tarfile.**ENCODING**
默认的字符编码格式: 在 Windows 上为 'utf-8'，其他系统上则为*sys.getfilesystemencoding()* 所返回的值。

以下常量各自定义了一个*tarfile* 模块能够创建的 tar 归档格式。相关细节请参阅*受支持的 tar 格式* 小节。

tarfile.**USTAR_FORMAT**
POSIX.1-1988 (ustar) 格式。

tarfile.**GNU_FORMAT**
GNU tar 格式。

tarfile.**PAX_FORMAT**
POSIX.1-2001 (pax) 格式。

tarfile.**DEFAULT_FORMAT**
用于创建归档的默认格式。目前为*GNU_FORMAT*。

参见:

模块*zipfile* *zipfile* 标准模块的文档。

归档操作 标准*shutil* 模块所提供的高层级归档工具的文档。

GNU tar manual, Basic Tar Format 针对 tar 归档文件的文档，包含 GNU tar 扩展。

13.6.1 TarFile 对象

TarFile 对象提供了一个 tar 归档的接口。一个 tar 归档就是数据块的序列。一个归档成员（被保存文件）是由一个标头块加多个数据块组成的。一个文件可以在一个 tar 归档中多次被保存。每个归档成员都由一个*TarInfo* 对象来代表，详情参见*TarInfo 对象*。

TarFile 对象可在 with 语句中作为上下文管理器使用。当语句块结束时它将自动被关闭。请注意在发生异常事件时被打开用于写入的归档将不会被终结；只有内部使用的文件对象将被关闭。相关用例请参见*例子*。

3.2 新版功能: 添加了对上下文管理器协议的支持。

class tarfile.**TarFile**(*name=None*, *mode='r'*, *fileobj=None*, *format=DEFAULT_FORMAT*, *tarinfo=TarInfo*, *dereference=False*, *ignore_zeros=False*, *encoding=ENCODING*, *errors='surrogateescape'*, *pax_headers=None*, *debug=0*, *errorlevel=0*)

下列所有参数都是可选项并且也可作为实例属性来访问。

name 是归档的路径名称。*name* 可以是一个*path-like object*。如果给定了 *fileobj* 则它可以被省略。在此情况下，如果对象的 `name` 属性存在则它会被使用。

mode 可以为 `'r'` 表示从现有归档读取，`'a'` 表示将数据追加到现有文件，`'w'` 表示创建新文件覆盖现有文件，或者 `'x'` 表示仅在文件不存在时创建新文件。

如果给定了 *fileobj*，它会被用于读取或写入数据。如果可以被确定，则 *mode* 会被 *fileobj* 的模式所覆盖。*fileobj* 的使用将从位置 0 开始。

注解： 当`TarFile` 被关闭时，*fileobj* 不会被关闭。

---

format 控制归档格式。它必须为在模块层级定义的常量`USTAR_FORMAT`, `GNU_FORMAT` 或`PAX_FORMAT` 中的一个。

tarinfo 参数可以被用来将默认的`TarInfo` 类替换为另一个。

如果 *dereference* 为`False`，则会将符号链接和硬链接添加到归档中。如果为`True`，则会将目标文件的内容添加到归档中。在不支持符号链接的系统上参数将不起作用。

如果 *ignore_zeros* 为`False`，则会将空的数据块当作归档的末尾来处理。如果为`True`，则会跳过空的（和无效的）数据块并尝试获取尽可能多的成员。此参数仅适用于读取拼接的或损坏的归档。

debug 可设为从 `0` (无调试消息) 到 `3` (全部调试消息)。消息会被写入到 `sys.stderr`。

如果 *errorlevel* 为 `0`，则当使用`TarFile.extract()` 时会忽略所有错误。无论何种情况，当启用调试时它们都将被显示为调试输出的错误消息。如果为 `1`，则所有 *fatal* 错误会被作为`OSError` 异常被引发。如果为 `2`，则所有 *non-fatal* 错误也会被作为`TarError` 异常被引发。

encoding 和 *errors* 参数定义了读取或写入归档所使用的字符编码格式以及要如何处理转换错误。默认设置将适用于大多数用户。要深入了解详情可参阅*Unicode 问题* 小节。

可选的 *pax_headers* 参数是字符串的字典，如果 *format* 为`PAX_FORMAT` 它将被作为 pax 全局标头被添加。

在 3.2 版更改: 使用 `'surrogateescape'` 作为 *errors* 参数的默认值。

在 3.5 版更改: 添加了 `'x'` (仅创建) 模式。

在 3.6 版更改: *name* 形参接受一个*path-like object*。

classmethod TarFile.**open**(...)

作为替代的构造器。`tarfile.open()` 函数实际上是这个类方法的快捷方式。

TarFile.**getmember**(*name*)

返回成员 *name* 的`TarInfo` 对象。如果 *name* 在归档中找不到，则会引发`KeyError`。

注解： 如果一个成员在归档中出现超过一次，它的最后一次出现会被视为是最新的版本。

TarFile.**getmembers**()

以`TarInfo` 对象列表的形式返回归档的成员。列表的顺序与归档中成员的顺序一致。

TarFile.**getnames**()

以名称列表的形式返回成员。它的顺序与`getmembers()` 所返回列表的顺序一致。

TarFile.**list**(*verbose=True*, *, *members=None*)
将内容清单打印到 sys.stdout。如果 *verbose* 为*False*，则将只打印成员名称。如果为*True*，则输出将类似于 **ls -l** 的输出效果。如果给定了可选的 *members*，它必须为*getmembers()* 所返回的列表的一个子集。

在 3.5 版更改: 添加了 *members* 形参。

TarFile.**next**()
当*TarFile* 被打开用于读取时，以*TarInfo* 对象的形式返回归档的下一个成员。如果不再有可用对象则返回*None*。

TarFile.**extractall**(*path="."*, *members=None*, *, *numeric_owner=False*)
将归档中的所有成员提取到当前工作目录或 *path* 目录。如果给定了可选的 *members*，则它必须为*getmembers()* 所返回的列表的一个子集。字典信息例如所有者、修改时间和权限会在所有成员提取完毕后被设置。这样做是为了避免两个问题：目录的修改时间会在每当在其中创建文件时被重置。并且如果目录的权限不允许写入，提取文件到目录的操作将失败。

如果 *numeric_owner* 为*True*，则将使用来自 tarfile 的 uid 和 gid 数值来设置被提取文件的所有者/用户组。在其他情况下，则会使用来自 tarfile 的名称值。

警告: 绝不要未经预先检验就从不可靠的源中提取归档文件。这样有可能在 *path* 之外创建文件，例如某些成员具有以 "/" 开始的绝对路径文件名或带有两个点号 ".." 的文件名。

在 3.5 版更改: 添加了 *numeric_owner* 形参。

在 3.6 版更改: *path* 形参接受一个*path-like object*。

TarFile.**extract**(*member*, *path=""*, *set_attrs=True*, *, *numeric_owner=False*)
从归档中提取出一个成员放入当前工作目录，将使用其完整名称。成员的文件信息会尽可能精确地被提取。*member* 可以是一个文件名或*TarInfo* 对象。你可以使用 *path* 指定一个不同的目录。*path* 可以是一个*path-like object*。将会设置文件属性 (owner, mtime, mode) 除非 *set_attrs* 为假值。

如果 *numeric_owner* 为*True*，则将使用来自 tarfile 的 uid 和 gid 数值来设置被提取文件的所有者/用户组。在其他情况下，则会使用来自 tarfile 的名称值。

注解: *extract()* 方法不会处理某些提取问题。在大多数情况下你应当考虑使用*extractall()* 方法。

警告: 参看*extractall()* 的警告信息。

在 3.2 版更改: 添加了 *set_attrs* 形参。

在 3.5 版更改: 添加了 *numeric_owner* 形参。

在 3.6 版更改: *path* 形参接受一个*path-like object*。

TarFile.**extractfile**(*member*)
将归档中的一个成员提取为文件对象。*member* 可以是一个文件名或*TarInfo* 对象。如果 *member* 是一个常规文件或链接，则会返回一个*io.BufferedReader* 对象。在其他情况下将返回*None*。

在 3.3 版更改: 返回一个*io.BufferedReader* 对象。

TarFile.**add**(*name*, *arcname=None*, *recursive=True*, *exclude=None*, *, *filter=None*)
Add the file *name* to the archive. *name* may be any type of file (directory, fifo, symbolic link, etc.). If given, *arcname* specifies an alternative name for the file in the archive. Directories are added recursively by default. This can be avoided by setting *recursive* to *False*. If *exclude* is given, it must be a function that takes one filename

argument and returns a boolean value. Depending on this value the respective file is either excluded (*True*) or added (*False*). If *filter* is specified it must be a keyword argument. It should be a function that takes a *TarInfo* object argument and returns the changed *TarInfo* object. If it instead returns *None* the *TarInfo* object will be excluded from the archive. See 例子 for an example.

在 3.2 版更改: 添加了 *filter* 形参。

3.2 版后已移除: The *exclude* parameter is deprecated, please use the *filter* parameter instead.

TarFile.**addfile**(*tarinfo*, *fileobj=None*)
将*TarInfo* 对象 *tarinfo* 添加到归档。如果给定了 *fileobj*，它应当是一个*binary file*，并会从中读取 tarinfo.size 个字节添加到归档。你可以直接创建*TarInfo* 对象，或是通过使用*gettarinfo()*。

TarFile.**gettarinfo**(*name=None*, *arcname=None*, *fileobj=None*)
基于*os.stat()* 的结果或者现有文件的相同数据创建一个*TarInfo*。文件或者是命名为 *name*，或者是使用文件描述符指定为一个*file object* *fileobj*。*name* 可以是一个*path-like object*。如果给定了 *arcname*，则它将为归档中的文件指定一个替代名称，在其他情况下，名称将从 *fileobj* 的*name* 属性或 *name* 参数获取。名称应当是一个文本字符串。

你可以在使用*addfile()* 添加*TarInfo* 的某些属性之前修改它们。如果文件对象不是从文件开头进行定位的普通文件对象，*size* 之类的属性就可能需要修改。例如*GzipFile* 之类的文件就属于这种情况。*name* 也可以被修改，在这种情况下 *arcname* 可以是一个占位字符串。

在 3.6 版更改: *name* 形参接受一个*path-like object*。

TarFile.**close**()
关闭*TarFile*。在写入模式下，会向归档添加两个表示结束的零数据块。

TarFile.**pax_headers**
一个包含 pax 全局标头的键值对的字典。

13.6.2 TarInfo 对象

TarInfo 对象代表*TarFile* 中的一个文件。除了会存储所有必要的文件属性（例如文件类型、大小、时间、权限、所有者等），它还提供了一些确定文件类型的有用方法。此对象 并不包含文件数据本身。

TarInfo 对象可通过*TarFile* 的方法 getmember(), getmembers() 和 gettarinfo() 返回。

class tarfile.**TarInfo**(*name=""*)
创建一个*TarInfo* 对象。

classmethod TarInfo.**frombuf**(*buf*, *encoding*, *errors*)
基于字符串缓冲区 *buf* 创建并返回一个*TarInfo* 对象。

如果缓冲区无效则会引发*HeaderError*。

classmethod TarInfo.**fromtarfile**(*tarfile*)
从*TarFile* 对象 *tarfile* 读取下一个成员并将其作为*TarInfo* 对象返回。

TarInfo.**tobuf**(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)
基于*TarInfo* 对象创建一个字符串缓冲区。有关参数的信息请参见*TarFile* 类的构造器。

在 3.2 版更改: 使用 'surrogateescape' 作为 *errors* 参数的默认值。

TarInfo 对象具有以下公有数据属性:

TarInfo.**name**
归档成员的名称。

TarInfo.**size**
以字节表示的大小。

TarInfo.**mtime**
上次修改的时间。

TarInfo.**mode**
权限位。

TarInfo.**type**
文件类型。*type* 通常为以下常量之一: REGTYPE, AREGTYPE, LNKTYPE, SYMTYPE, DIRTYPE, FIFOTYPE, CONTTYPE, CHRTYPE, BLKTYPE, GNUTYPE_SPARSE。要更方便地确定一个*TarInfo*对象的类型，请使用下述的 is*() 方法。

TarInfo.**linkname**
目标文件名的名称，该属性仅在类型为 LNKTYPE 和 SYMTYPE 的*TarInfo*对象中存在。

TarInfo.**uid**
最初保存该成员的用户的用户 ID。

TarInfo.**gid**
最初保存该成员的用户的分组 ID。

TarInfo.**uname**
用户名。

TarInfo.**gname**
分组名。

TarInfo.**pax_headers**
一个包含所关联的 pax 扩展标头的键值对的字典。

TarInfo 对象还提供了一些便捷查询方法:

TarInfo.**isfile**()
如果 Tarinfo 对象为普通文件则返回*True*。

TarInfo.**isreg**()
与*isfile()* 相同。

TarInfo.**isdir**()
如果为目录则返回*True*。

TarInfo.**issym**()
如果为符号链接则返回*True*。

TarInfo.**islnk**()
如果为硬链接则返回*True*。

TarInfo.**ischr**()
如果为字符设备则返回*True*。

TarInfo.**isblk**()
如果为块设备则返回*True*。

TarInfo.**isfifo**()
如果为 FIFO 则返回*True*。.

TarInfo.**isdev**()
如果为字符设备、块设备或 FIFO 之一则返回*True*。

13.6.3 命令行界面

3.4 新版功能.

tarfile 模块提供了简单的命令行接口以便与 tar 归档进行交互。

如果你想要创建一个新的 tar 归档，请在*-c* 选项后指定其名称然后列出应当被包含的文件名:

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

传入一个字典也是可接受的:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

如果你想要将一个 tar 归档提取到指定的目录，请使用*-e* 选项:

```
$ python -m tarfile -e monty.tar
```

你也可以通过传入目录名称将一个 tar 归档提取到不同的目录:

```
$ python -m tarfile -e monty.tar  other-dir/
```

要获取一个 tar 归档中文件的列表，请使用*-l* 选项:

```
$ python -m tarfile -l monty.tar
```

命令行选项

-l <tarfile>
--list <tarfile>
列出一个 tarfile 中的文件名。

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>
基于源文件创建 tarfile。

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]
如果未指定 *output_dir* 则会将 tarfile 提取到当前目录。

-t <tarfile>
--test <tarfile>
检测 tarfile 是否有效。

-v, --verbose
更详细地输出结果。

13.6.4 例子

如何将整个 tar 归档提取到当前工作目录:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

如何通过*TarFile.extractall()* 使用生成器函数而非列表来提取一个 tar 归档的子集:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

如何基于一个文件名列表创建未压缩的 tar 归档:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

使用 `with` 语句的同一个示例:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

如何读取一个 gzip 压缩的 tar 归档并显示一些成员信息:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

如何创建一个归档并使用*TarFile.add()* 中的 *filter* 形参来重置用户信息:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
```

(下页继续)

(续上页)

```
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.6.5 受支持的 tar 格式

通过*tarfile*模块可以创建三种 tar 格式:

- The POSIX.1-1988 ustar 格式 (*USTAR_FORMAT*)。它支持最多 256 个字符的文件名长度和最多 100 个字符的链接名长度。文件大小上限为 8 GiB。这是一种老旧但广受支持的格式。
- GNU tar 格式 (*GNU_FORMAT*)。它支持长文件名和链接名、大于 8 GiB 的文件以及稀疏文件。它是 GNU/Linux 系统上的事实标准。*tarfile*完全支持针对长名称的 GNU tar 扩展，稀疏文件支持则限制为只读。
- POSIX.1-2001 pax 格式 (*PAX_FORMAT*)。它是几乎无限制的最灵活格式。它支持长文件名和链接名，大文件以及使用便捷方式存储路径名。但是并非所有现今的 tar 实现都能够正确地处理 pax 归档。

  pax 格式是对现有 *ustar* 格式的扩展。它会对无法以其他方式存储的信息使用附加标头。存在两种形式的 pax 标头：扩展标头只影响后续的文件标头，全局标头则适用于完整归档并会影响所有后续的文件。为了便于移植，在 pax 标头中的所有数据均以 *UTF-8* 编码。

还有一些 tar 格式的其他变种，它们可以被读取但不能被创建:

- 古老的 V7 格式。这是来自 Unix 第七版的第一个 tar 格式，它只存储常规文件和目录。名称长度不能超过 100 个字符，并且没有用户/分组名信息。某些归档在带有非 ASCII 字符字段的情况下会产生计算错误的标头校验和。
- SunOS tar 扩展格式。此格式是 POSIX.1-2001 pax 格式的一个变种，但并不保持兼容。

13.6.6 Unicode 问题

最初 tar 格式被设计用来在磁带机上生成备份，主要关注于保存文件系统信息。现在 tar 归档通常用于文件分发和在网络上交换归档。最初格式（它是所有其他格式的基础）的一个问题是它没有支持不同字符编码格式的概念。例如，一个在 *UTF-8* 系统上创建的普通 tar 归档如果包含非 *ASCII* 字符则将无法在 *Latin-1* 系统上被正确读取。文本元数据（例如文件名，链接名，用户/分组名）将变为损坏状态。不幸的是，没有什么办法能够自动检测一个归档的编码格式。pax 格式被设计用来解决这个问题。它使用通用字符编码格式 *UTF-8* 来存储非 ASCII 元数据。

在*tarfile*中字符转换的细节由*TarFile*类的 *encoding* 和 *errors* 关键字参数控制。

encoding 定义了用于归档中元数据的字符编码格式。默认值为*sys.getfilesystemencoding()*或是回退选项 `'ascii'`。根据归档是被读取还是被写入，元数据必须被解码或编码。如果没有正确设置 *encoding*，转换可能会失败。

errors 参数定义了不能被转换的字符将如何处理。可能的取值在错误处理方案 小节列出。默认方案为 `'surrogateescape'`，它也被 Python 用于文件系统调用，参见文件名，命令行参数，以及环境变量。。

对于*PAX_FORMAT*归档，*encoding* 通常是不必要的，因为所有元数据都使用 *UTF-8* 来存储。*encoding* 仅在解码二进制 pax 标头或存储带有替代字符的字符串等少数场景下会被使用。

CHAPTER 14

文件格式

本章中描述的模块解析各种不是标记语言且与电子邮件无关的杂项文件格式。

14.1 csv —CSV 文件读写

源代码： Lib/csv.py

CSV (Comma Separated Values) 格式是电子表格和数据库中最常见的输入、输出文件格式。在 **RFC 4180** 规范推出的很多年前，CSV 格式就已经被开始使用了，由于当时并没有合理的标准，不同应用程序读写的数据会存在细微的差别。这种差别让处理多个来源的 CSV 文件变得困难。但尽管分隔符会变化，此类文件的大致格式是相似的，所以编写一个单独的模块以高效处理此类数据，将程序员从读写数据的繁琐细节中解放出来是有可能的。

The *csv* module implements classes to read and write tabular data in CSV format. It allows programmers to say, "write this data in the format preferred by Excel," or "read data from this file which was generated by Excel," without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

csv 模块中的*reader* 类和*writer* 类可用于读写序列化的数据。也可使用*DictReader* 类和*DictWriter* 类以字典的形式读写数据。

参见：

该实现在 "Python 增强提议" - PEP *305* (CSV 文件 API) 中被提出《Python 增强提议》提出了对 Python 的这一补充。

14.1.1 模块内容

csv 模块定义了以下函数：

csv.**reader**(*csvfile*, *dialect='excel'*, ***fmtparams*)
返回一个 reader 对象，该对象将逐行遍历 *csvfile*。*csvfile* 可以是任何对象，只要这个对象支持*iterator* 协议并在每次调用 `__next__()` 方法时都返回字符串，文件对象 和列表对象均适用。如果 *csvfile* 是文件对象，则打开它时应使用 `newline=''`。[1] 可选参数 *dialect* 是用于不同的 CSV 方言的特定参数组。它可以是*Dialect* 类的子类的实例，也可以是*list_dialects()* 函数返回的字符串之一。另一个可选关键字参数 *fmtparams* 可以覆写当前方言格式中的单个格式设置。有关方言和格式设置参数的完整详细信息，请参见变种与格式参数 部分。

csv 文件的每一行都读取为一个由字符串组成的列表。除非指定了 `QUOTE_NONNUMERIC` 格式选项（在这种情况下，未引用的字段会转换为浮点数），否则不会执行自动数据类型转换。

一个简短的用法示例：

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

csv.**writer**(*csvfile*, *dialect='excel'*, ***fmtparams*)
返回一个 writer 对象，该对象负责将用户的数据在给定的文件类对象上转换为带分隔符的字符串。*csvfile* 可以是具有 `write()` 方法的任何对象。如果 *csvfile* 是文件对象，则打开它时应使用 `newline=''`。[1] 可选参数 *dialect* 是用于不同的 CSV 方言的特定参数组。它可以是*Dialect* 类的子类的实例，也可以是*list_dialects()* 函数返回的字符串之一。另一个可选关键字参数 *fmtparams* 可以覆写当前方言格式中的单个格式设置。有关方言和格式设置参数的完整详细信息，请参见变种与格式参数 部分。为了尽量简化与数据库 API 模块之间的对接，*None* 值会写入为空字符串。虽然这个转换是不可逆的，但它让 SQL 空数据值转储到 CSV 文件更容易，而无需预处理从 `cursor.fetch*` 调用返回的数据。写入前，所有非字符串数据都先用*str()* 转化为字符串再写入。

一个简短的用法示例：

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

csv.**register_dialect**(*name*[, *dialect*[, ***fmtparams*]])
将 *name* 与 *dialect* 关联起来。*name* 必须是字符串。要指定变种 (dialect)，可以给出*Dialect* 的子类，或给出 *fmtparams* 关键字参数，或两者都给出，此时关键字参数会覆盖 *dialect* 参数。有关变种和格式设置参数的完整详细信息，请参见变种与格式参数 部分。

csv.**unregister_dialect**(*name*)
从变种注册表中删除 *name* 对应的变种。如果 *name* 不是已注册的变种名称，则抛出*Error* 异常。

csv.**get_dialect**(*name*)
返回 *name* 对应的变种。如果 *name* 不是已注册的变种名称，则抛出*Error* 异常。该函数返回的是不可变的*Dialect* 对象。

[1] 如果没有指定 `newline=''`，则嵌入引号中的换行符将无法正确解析，并且在写入时，使用 `\r\n` 换行的平台会有多余的 `\r` 写入。由于 csv 模块会执行自己的（通用）换行符处理，因此指定 `newline=''` 应该总是安全的。

csv.**list_dialects**()
返回所有已注册变种的名称。

csv.**field_size_limit**([*new_limit*])
返回解析器当前允许的最大字段大小。如果指定了 *new_limit*，则它将成为新的最大字段大小。

csv 模块定义了以下类：

class csv.**DictReader**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)
创建一个对象，其操作类似于常规 reader 但会将每行中的信息映射到一个*OrderedDict*，其中的键由可选的 *fieldnames* 形参给出。

fieldnames 形参是一个*sequence*。如果省略 *fieldnames*，则文件 *f* 第一行中的值将被用作字段名。无论字段名是如何确定的，有序字典都将保留其原始顺序。

If a row has more fields than fieldnames, the remaining data is put in a list and stored with the fieldname specified by *restkey* (which defaults to `None`). If a non-blank row has fewer fields than fieldnames, the missing values are filled-in with `None`.

所有其他可选或关键字参数都传递给底层的*reader* 实例。

在 3.6 版更改: 返回的行现在的类型是 `OrderedDict`。

一个简短的用法示例:

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
OrderedDict([('first_name', 'John'), ('last_name', 'Cleese')])
```

class csv.**DictWriter**(*f*, *fieldnames*, *restval=''*, *extrasaction='raise'*, *dialect='excel'*, **args*, ***kwds*)
创建一个对象，该对象在操作上类似常规 writer，但能将字典映射到输出行。*fieldnames* 参数是由 key（键）组成的序列，用于指定字典中的 value （值）的顺序，这些值会按指定顺序传递给 `writerow()` 方法并写入 *f* 文件。如果字典缺少 *fieldnames* 中的键，则可选参数 *restval* 用于指定要写入的值。如果传递给 `writerow()` 方法的字典的某些键在 *fieldnames* 中找不到，则可选参数 *extrasaction* 用于指定要执行的操作。如果将其设置为 `'raise'` (默认值)，则会引发*ValueError*。如果将其设置为 `'ignore'`，则字典中的其他键值将被忽略。所有其他可选或关键字参数都传递给底层的*writer* 实例。

Note that unlike the *DictReader* class, the *fieldnames* parameter of the *DictWriter* is not optional. Since Python's *dict* objects are not ordered, there is not enough information available to deduce the order in which the row should be written to file *f*.

一个简短的用法示例:

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.**Dialect**
Dialect 类是主要依赖于其属性的容器类，用于将定义好的参数传递给特定的*reader* 或*writer* 实例。

class csv.**excel**
excel 类定义了 Excel 生成的 CSV 文件的常规属性。它在变种注册表中的名称是 'excel'。

class csv.**excel_tab**
excel_tab 类定义了 Excel 生成的、制表符分隔的 CSV 文件的常规属性。它在变种注册表中的名称是 'excel-tab'。

class csv.**unix_dialect**
unix_dialect 类定义了在 UNIX 系统上生成的 CSV 文件的常规属性，即使用 '\n' 作为换行符，且所有字段都有引号包围。它在变种注册表中的名称是 'unix'。

3.2 新版功能.

class csv.**Sniffer**
Sniffer 类用于推断 CSV 文件的格式。

Sniffer 类提供了两个方法：

sniff(*sample*, *delimiters=None*)
分析给定的 *sample* 并返回一个*Dialect* 子类，该子类中包含了分析出的格式参数。如果给出可选的 *delimiters* 参数，则该参数会被解释为字符串，该字符串包含了可能的有效分隔符。

has_header(*sample*)
分析示例文本（假定为 CSV 格式），如果第一行很可能是一系列列标题，则返回*True*。

使用*Sniffer* 的示例：

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

csv 模块定义了以下常量：

csv.**QUOTE_ALL**
指示*writer* 对象给所有字段加上引号。

csv.**QUOTE_MINIMAL**
指示*writer* 对象仅为包含特殊字符（例如 定界符、引号字符或 行结束符中的任何字符）的字段加上引号。

csv.**QUOTE_NONNUMERIC**
指示*writer* 对象为所有非数字字段加上引号。

指示 reader 将所有未用引号引出的字段转换为 *float* 类型。

csv.**QUOTE_NONE**
指示*writer* 对象不使用引号引出字段。当 定界符出现在输出数据中时，其前面应该有 转义符。如果未设置 转义符，则遇到任何需要转义的字符时，writer 都会抛出*Error* 异常。

指示*reader* 不对引号字符进行特殊处理。

csv 模块定义了以下异常：

exception csv.**Error**
该异常可能由任何发生错误的函数抛出。

14.1.2 变种与格式参数

为了更容易指定输入和输出记录的格式，特定的一组格式参数组合为一个 dialect（变种）。一个 dialect 是一个*Dialect* 类的子类，它具有一组特定的方法和一个 `validate()` 方法。创建*reader* 或*writer* 对象时，程序员可以将某个字符串或*Dialect* 类的子类指定为 dialect 参数。要想补充或覆盖 *dialect* 参数，程序员还可以单独指定某些格式参数，这些参数的名称与下面*Dialect* 类定义的属性相同。

Dialect 类支持以下属性：

`Dialect.`**`delimiter`**
一个用于分隔字段的单字符，默认为 `','`。

`Dialect.`**`doublequote`**
控制出现在字段中的 *引号字符*本身应如何被引出。当该属性为*True* 时，双写引号字符。如果该属性为*False*，则在 *引号字符*的前面放置 *转义符*。默认值为*True*。

在输出时，如果 *doublequote* 是*False*，且 *转义符*未指定，且在字段中发现 *引号字符*时，会抛出*Error* 异常。

`Dialect.`**`escapechar`**
一个用于 writer 的单字符，用来在 *quoting* 设置为*QUOTE_NONE* 的情况下转义 *定界符*，在 *doublequote* 设置为*False* 的情况下转义 *引号字符*。在读取时，*escapechar* 去除了其后所跟字符的任何特殊含义。该属性默认为*None*，表示禁用转义。

`Dialect.`**`lineterminator`**
放在*writer* 产生的行的结尾，默认为 `'\r\n'`。

注解： *reader* 经过硬编码，会识别 `'\r'` 或 `'\n'` 作为行尾，并忽略 *lineterminator*。未来可能会更改这一行为。

`Dialect.`**`quotechar`**
一个单字符，用于包住含有特殊字符的字段，特殊字符如 *定界符*或 *引号字符*或换行符。默认为 `'"'`。

`Dialect.`**`quoting`**
控制 writer 何时生成引号，以及 reader 何时识别引号。该属性可以等于任何 `QUOTE_*` 常量（参见模块内容 段落），默认为*QUOTE_MINIMAL*。

`Dialect.`**`skipinitialspace`**
如果为*True*，则忽略 *定界符*之后的空格。默认值为*False*。

`Dialect.`**`strict`**
如果为 `True`，则在输入错误的 CSV 时抛出*Error* 异常。默认值为 `False`。

14.1.3 Reader 对象

Reader 对象（*DictReader* 实例和*reader()* 函数返回的对象）具有以下公开方法：

`csvreader.`**`__next__`**`()`
返回 reader 的可迭代对象的下一行，返回值可能是列表（由*reader()* 返回的对象）或字典（由*DictReader* 返回的对象），解析是根据当前设置的变种进行的。通常应该这样调用它：`next(reader)`。

Reader 对象具有以下公开属性：

`csvreader.`**`dialect`**
变种描述，只读，供解析器使用。

csvreader.**line_num**
源迭代器已经读取了的行数。它与返回的记录数不同，因为记录可能跨越多行。

DictReader 对象具有以下公开属性：

csvreader.**fieldnames**
字段名称。如果在创建对象时未传入字段名称，则首次访问时或从文件中读取第一条记录时会初始化此属性。

14.1.4 Writer 对象

Writer 对象（*DictWriter* 实例和*writer()* 函数返回的对象）具有下面的公开方法。对于 Writer 对象，行必须是（一组可迭代的）字符串或数字。对于*DictWriter* 对象，行必须是一个字典，这个字典将字段名映射为字符串或数字（数字要先经过*str()* 转换类型）。请注意，输出的复数会有括号包围。这样其他程序读取 CSV 文件时可能会有一些问题（假设它们完全支持复数）。

csvwriter.**writerow**(*row*)
将 *row* 形参写入 writer 的文件对象，并按照当前设定形式进行格式化。

在 3.5 版更改: 开始支持任意类型的迭代器。

csvwriter.**writerows**(*rows*)
将 *rows** （即能迭代出多个上述 **row* 对象的迭代器）中的所有元素写入 writer 的文件对象，并根据当前设置的变种进行格式化。

Writer 对象具有以下公开属性：

csvwriter.**dialect**
变种描述，只读，供 writer 使用。

DictWriter 对象具有以下公开方法：

DictWriter.**writeheader**()
使用（构造器所规定的）字段名写入一行。

3.2 新版功能.

14.1.5 例子

读取 CSV 文件最简单的一个例子:

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

读取其他格式的文件:

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

相应最简单的写入示例是:

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

由于使用*open()*来读取 CSV 文件，因此默认情况下，将使用系统默认编码来解码文件并转换为 unicode（请参阅*locale.getpreferredencoding()*）。要使用其他编码来解码文件，请使用 open 的 encoding 参数:

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

这同样适用于写入非系统默认编码的内容：打开输出文件时，指定 encoding 参数。

注册一个新的变种:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

Reader 的更高级用法——捕获并报告错误:

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

尽管该模块不直接支持解析字符串，但仍可如下轻松完成:

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

备注

14.2 configparser —配置文件解析器

源代码: Lib/configparser.py

此模块提供了它实现一种基本配置语言*ConfigParser*类，这种语言所提供的结构与 Microsoft Windows INI 文件的类似。你可以使用这种语言来编写能够由最终用户来自定义的 Python 程序。

注解: 这个库 *并不能够*解析或写入在 Windows Registry 扩展版本 INI 语法中所使用的值-类型前缀。

参见:

模块*shlex* 支持创建可被用作应用配置文件的替代的 Unix 终端式微语言。

模块*json* json 模块实现了一个 JavaScript 语法的子集，它也可被用于这种目的。

14.2.1 快速起步

让我们准备一个非常基本的配置文件，它看起来是这样的:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

INI 文件的结构描述见以下章节。总的来说，这种文件由多个节组成，每个节包含多个带有值的键。*configparser* 类可以读取和写入这种文件。让我们先通过程序方式来创建上述的配置文件。

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['bitbucket.org'] = {}
>>> config['bitbucket.org']['User'] = 'hg'
>>> config['topsecret.server.com'] = {}
>>> topsecret = config['topsecret.server.com']
>>> topsecret['Port'] = '50022'     # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

如你所见，我们可以把配置解析器当作一个字典来处理。两者确实存在差异，将在后文说明，但是其行为非常接近于字典所具有一般行为。

现在我们已经创建并保存了一个配置文件，让我们再将它读取出来并探究其中包含的数据。

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['bitbucket.org', 'topsecret.server.com']
>>> 'bitbucket.org' in config
True
```

(下页继续)

(续上页)

```
>>> 'bytebong.com' in config
False
>>> config['bitbucket.org']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.com']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['bitbucket.org']: print(key)
...
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['bitbucket.org']['ForwardX11']
'yes'
```

正如我们在上面所看到的，相关的 API 相当直观。唯一有些神奇的地方是 DEFAULT 小节，它为所有其他小节提供了默认值[1]。还要注意小节中的键大小写不敏感并且会存储为小写形式[1]。

14.2.2 支持的数据类型

配置解析器并不会猜测配置文件中值的类型，而总是将它们在内部存储为字符串。这意味着如果你需要其他数据类型，你应当自己来转换:

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

由于这种任务十分常用，配置解析器提供了一系列便捷的获取方法来处理整数、浮点数和布尔值。最后一个类型的处理最为有趣，因为简单地将值传给 bool() 是没有用的，bool('False') 仍然会是 True。为解决这个问题配置解析器还提供了*getboolean()*。这个方法对大小写不敏感并可识别 'yes'/'no', 'on'/'off', 'true'/'false' 和 '1'/'0'[1] 等布尔值。例如:

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['bitbucket.org'].getboolean('ForwardX11')
True
>>> config.getboolean('bitbucket.org', 'Compression')
True
```

除了*getboolean()*，配置解析器还提供了同类的*getint()* 和*getfloat()* 方法。你可以注册你自己的转换器并或是定制已提供的转换器。[1]

[1] 配置解析器允许重度定制。如果你有兴趣改变脚注说明中所介绍的行为，请参阅*Customizing Parser Behaviour* 一节。

14.2.3 回退值

与字典类似，你可以使用某个小节的 `get()` 方法来提供回退值:

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

请注意默认值会优先于回退值。例如，在我们的示例中 `'CompressionLevel'` 键仅在 `'DEFAULT'` 小节被指定。如果你尝试在 `'topsecret.server.com'` 小节获取它，我们将总是获取到默认值，即使我们指定了一个回退值:

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

还需要注意的一点是解析器层级的 `get()` 方法提供了自定义的更复杂接口，它被维护用于向下兼容。当使用此方法时，回退值可以通过 `fallback` 仅限关键字参数来提供:

```
>>> config.get('bitbucket.org', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

同样的 `fallback` 参数也可在*getint()*，*getfloat()* 和*getboolean()* 方法中使用，例如:

```
>>> 'BatchMode' in topsecret
False
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 受支持的 INI 文件结构

配置文件是由小节组成的，每个小节都有一个 `[section]` 标头，加上多个由特定字符串 (默认为 `=` 或 `:`[1]) 分隔的键/值条目。默认情况下小节名对大小写敏感而键对大小写不敏感[1]。键和值开头和末尾的空格会被移除。值可以被省略，在此情况下键/值分隔符也可以被省略。值还可以跨越多行，只要其他行带有比值的第一行更深的缩进。依据解析器的具体模式，空白行可能被视为多行值的组成部分也可能被忽略。

配置文件可以包含注释，要带有指定字符前缀 (默认为 `#` 和 `;`[1])。注释可以单独出现于原本的空白行，并可使用缩进。[1]

例如:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
```

(下页继续)

(续上页)

```
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
    I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
        multiline_values = are
            handled just fine as
            long as they are indented
            deeper than the first line
            of a value
        # Did I mention we can indent comments, too?
```

14.2.5 值的插入

在核心功能之上，*ConfigParser* 还支持插值。这意味着值可以在被 `get()` 调用返回之前进行预处理。

class configparser.**BasicInterpolation**

默认实现由*ConfigParser* 来使用。它允许值包含引用了相同小节中其他值或者特殊的默认小节中的值的格式字符串[1]。额外的默认值可以在初始化时提供。

例如:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures
```

在上面的例子里，*ConfigParser* 的 *interpolation* 设为 `BasicInterpolation()`，这会将 `%(home_dir)s` 求解为 `home_dir` 的值 (在这里是 `/Users`)。`%(my_dir)s` 的将被实际求解为 `/Users/lumberjack`。所有插值都是按需进行的，这样引用链中使用的键不必以任何特定顺序在配置文件中指明。

当 `interpolation` 设为 `None` 时，解析器会简单地返回 `%(my_dir)s/Pictures` 作为 `my_pictures` 的值，并返回 `%(home_dir)s/lumberjack` 作为 `my_dir` 的值。

class configparser.**ExtendedInterpolation**

一个用于插值的替代处理程序实现了更高级的语法，它被用于 zc.buildout 中的实例。扩展插值使用 ${section:option} 来表示来自外部小节的值。插值可以跨越多个层级。为了方便使用，section: 部分可被省略，插值会默认作用于当前小节（可能会从特殊小节获取默认值）。

例如，上面使用基本插值描述的配置，使用扩展插值将是这个样子:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures
```

来自其他小节的值也可以被获取:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

14.2.6 映射协议访问

3.2 新版功能.

映射协议访问这个通用名称是指允许以字典的方式来使用自定义对象的功能。在*configparser* 中，映射接口的实现使用了 parser['section']['option'] 标记法。

parser['section'] 专门为解析器中的小节数据返回一个代理。这意味着其中的值不会被拷贝，而是在需要时从原始解析器中获取。更为重要的是，当值在小节代理上被修改时，它们其实是在原始解析器中发生了改变。

configparser 对象的行为会尽可能地接近真正的字典。映射接口是完整而且遵循*MutableMapping* ABC 规范的。但是，还是有一些差异应当被纳入考虑:

- 默认情况下，小节中的所有键是以大小写不敏感的方式来访问的[1]。例如 for option in parser["section"] 只会产生 optionxform 形式的选项键名称。也就是说默认使用小写字母键名。与此同时，对于一个包含键 'a' 的小节，以下两个表达式均将返回 True:

```
"a" in parser["section"]
"A" in parser["section"]
```

- All sections include DEFAULTSECT values as well which means that .clear() on a section may not leave the section visibly empty. This is because default values cannot be deleted from the section (because technically they are not there). If they are overridden in the section, deleting causes the default value to be visible again. Trying to delete a default value causes a KeyError.
- DEFAULTSECT 无法从解析器中被移除:

  - trying to delete it raises `ValueError`,
  - `parser.clear()` 会保留其原状，
  - `parser.popitem()` 绝不会将其返回。

- `parser.get(section, option, **kwargs)` - 第二个参数 **并非**回退值。但是请注意小节层级的 `get()` 方法可同时兼容映射协议和经典配置解析器 API。
- `parser.items()` 兼容映射协议（返回 *section_name*, *section_proxy* 对的列表，包括 DEFAULTSECT）。但是，此方法也可以带参数发起调用: `parser.items(section, raw, vars)`。这种调用形式返回指定 `section` 的 *option*, *value* 对的列表，将展开所有插值（除非提供了 `raw=True` 选项）。

映射协议是在现有的传统 API 之上实现的，以便重载原始接口的子类仍然具有符合预期的有效映射。

14.2.7 定制解析器行为

INI 格式的变种数量几乎和使用此格式的应用一样多。*configparser* 花费了很大力气来为尽量大范围的可用 INI 样式提供支持。默认的可用功能主要由历史状况来确定，你很可能会想要定制某些特性。

改变特定配置解析器行为的最常见方式是使用 `__init__()` 选项:

- *defaults*，默认值: `None`

  此选项接受一个键值对的字典，它将被首先放入 `DEFAULT` 小节。这实现了一种优雅的方式来支持简洁的配置文件，它不必指定与已记录的默认值相同的值。

  提示：如果你想要为特定的小节指定默认的值，请在读取实际文件之前使用 `read_dict()`。

- *dict_type*，默认值: *collections.OrderedDict*

  此选项主要影响映射协议的行为和写入配置文件的外观。使用默认的有序字典时，每个小节是按照它们被加入解析器的顺序保存的。在小节内的选项也是如此。

  还有其他替换的字典类型可用，例如在写回数据时对小节和选项进行排序。你也可以出于性能原因而使用普通字典。

  Please note: there are ways to add a set of key-value pairs in a single operation. When you use a regular dictionary in those operations, the order of the keys may be random. For example:

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                                'key2': 'value2',
...                                'key3': 'value3'},
...                   'section2': {'keyA': 'valueA',
...                                'keyB': 'valueB',
...                                'keyC': 'valueC'},
...                   'section3': {'foo': 'x',
...                                'bar': 'y',
...                                'baz': 'z'}
... })
>>> parser.sections()
['section3', 'section2', 'section1']
>>> [option for option in parser['section3']]
['baz', 'foo', 'bar']
```

  In these operations you need to use an ordered dictionary as well:

```
>>> from collections import OrderedDict
>>> parser = configparser.ConfigParser()
```

(下页继续)

(续上页)

```
>>> parser.read_dict(
...   OrderedDict((
...     ('s1',
...      OrderedDict((
...        ('1', '2'),
...        ('3', '4'),
...        ('5', '6'),
...      ))
...     ),
...     ('s2',
...      OrderedDict((
...        ('a', 'b'),
...        ('c', 'd'),
...        ('e', 'f'),
...      ))
...     ),
...   ))
... )
>>> parser.sections()
['s1', 's2']
>>> [option for option in parser['s1']]
['1', '3', '5']
>>> [option for option in parser['s2'].values()]
['b', 'd', 'f']
```

- *allow_no_value*，默认值: `False`

  已知某些配置文件会包括不带值的设置，但其在其他方面均符合*configparser*所支持的语法。构造器的 *allow_no_value* 形参可用于指明应当接受这样的值:

```
>>> import configparser

>>> sample_config = """
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
...   skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)

>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren't specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

- *delimiters*，默认值: `('=', ':')`

  分隔符是用于在小节内分隔键和值的子字符串。在一行中首次出现的分隔子字符串会被视为一个分隔符。这意味着值可以包含分隔符（但键不可以）。

  另请参见*ConfigParser.write()* 的 *space_around_delimiters* 参数。

- *comment_prefixes*，默认值: `('#', ';')`

- *inline_comment_prefixes*，默认值: `None`

  注释前缀是配置文件中用于标示一条有效注释的开头的字符串。*comment_prefixes* 仅用在被视为空白的行（可以缩进）之前而 *inline_comment_prefixes* 可用在每个有效值之后（例如小节名称、选项以及空白的行）。默认情况下禁用行内注释，并且 `'#'` 和 `';'` 都被用作完整行注释的前缀。

  在 3.2 版更改: 在之前的*configparser* 版本中行为匹配 `comment_prefixes=('#',';')` 和 `inline_comment_prefixes=(';',)`。

  请注意配置解析器不支持对注释前缀的转义，因此使用 *inline_comment_prefixes* 可能妨碍用户将被用作注释前缀的字符指定为可选值。当有疑问时，请避免设置 *inline_comment_prefixes*。在许多情况下，在多行值的一行开头存储注释前缀字符的唯一方式是进行前缀插值，例如:

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
...   ${hash}!/usr/bin/env python
...   ${hash} -*- coding: utf-8 -*-
...
... extensions =
...   enabled_extension
...   another_extension
...   #disabled_by_comment
...   yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...   line #2
...   line #3
... """)
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- *strict*，默认值: `True`

  当设为 `True` 时，解析器在从单一源读取 (使用 `read_file()`, `read_string()` 或 `read_dict()`) 期间将不允许任何小节或选项出现重复。推荐在新的应用中使用严格解析器。

  在 3.2 版更改: 在之前的*configparser* 版本中行为匹配 `strict=False`。

- *empty_lines_in_values*，默认值: `True`

  在配置解析器中，值可以包含多行，只要它们的缩进级别低于它们所对应的键。默认情况下解析器还会将空行视为值的一部分。于此同时，键本身也可以任意缩进以提升可读性。因此，当配置文件变得非常庞大而复杂时，用户很容易失去对文件结构的掌控。例如:

  ```
  [Section]
  key = multiline
    value with a gotcha

   this = is still a part of the multiline value of 'key'
  ```

  在用户查看时这可能会特别有问题，如果她是使用比例字体来编辑文件的话。这就是为什么当你的应用不需要带有空行的值时，你应该考虑禁用它们。这将使得空行每次都会作为键之间的分隔。在上面的示例中，空行产生了两个键，`key` 和 `this`。

- *default_section*，默认值: `configparser.DEFAULTSECT` (即: `"DEFAULT"`)

  允许设置一个保存默认值的特殊节在其他节或插值等目的中使用的惯例是这个库所拥有的一个强大概念，使得用户能够创建复杂的声明性配置。这种特殊节通常称为 `"DEFAULT"` 但也可以被定制为指向任何其他有效的节名称。一些典型的值包括: `"general"` 或 `"common"`。所提供的名称在从任意节读取的时候被用于识别默认的节，而且也会在将配置写回文件时被使用。它的当前值可以使用 `parser_instance.default_section` 属性来获取，并且可以在运行时被修改（即将文件从一种格式转换为另一种格式）。

- *interpolation*，默认值: `configparser.BasicInterpolation`

  插值行为可以用通过提供 *interpolation* 参数提供自定义处理程序的方式来定制。`None` 可用来完全禁用插值，`ExtendedInterpolation()` 提供了一种更高级的变体形式，它的设计受到了 `zc.buildout` 的启发。有关该主题的更多信息请参见专门的文档章节。*RawConfigParser* 具有默认的值 `None`。

- *converters*，默认值: 不设置

  配置解析器提供了可选的值获取方法用来执行类型转换。默认实现包括*getint()*, *getfloat()* 以及*getboolean()*。如果还需要其他获取方法，用户可以在子类中定义它们，或者传入一个字典，其中每个键都是一个转换器的名称而每个值都是一个实现了特定转换的可调用对象。例如，传入 `{'decimal': decimal.Decimal}` 将对解释器对象和所有节代理添加 `getdecimal()`。换句话说，可以同时编写 `parser_instance.getdecimal('section', 'key', fallback=0)` 和 `parser_instance['section'].getdecimal('key', 0)`。

  如果转换器需要访问解析器的状态，可以在配置解析器子类上作为一个方法来实现。如果该方法的名称是以 `get` 打头的，它将在所有节代理上以兼容字典的形式提供（参见上面的 `getdecimal()` 示例）。

更多高级定制选项可通过重载这些解析器属性的默认值来达成。默认值是在类中定义的，因此它们可以通过子类或属性赋值来重载。

`ConfigParser.`**`BOOLEAN_STATES`**

默认情况下当使用*getboolean()* 时，配置解析器会将下列值视为 `True`: `'1'`, `'yes'`, `'true'`, `'on'` 而将下列值视为 `False`: `'0'`, `'no'`, `'false'`, `'off'`。你可以通过指定一个自定义的字符串键及其对应的布尔值字典来重载此行为。例如:

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
```

(下页继续)

(续上页)

```
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

其他典型的布尔值对包括 accept/reject 或 enabled/disabled。

ConfigParser.**optionxform**(*option*)

这个方法会转换每次 read, get, 或 set 操作的选项名称。默认会将名称转换为小写形式。这也意味着当一个配置文件被写入时，所有键都将为小写形式。如果此行为不合适则要重载此方法。例如:

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

ConfigParser.**SECTCRE**

一个已编译正则表达式会被用来解析节标头。默认将 [section] 匹配到名称 "section"。空格会被视为节名称的一部分，因此 [larch] 将被读取为一个名称为 " larch " 的节。如果此行为不合适则要重载此属性。例如:

```
>>> config = """
... [Section 1]
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> typical = ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

注解： 虽然 ConfigParser 对象也使用 OPTCRE 属性来识别选项行，但并不推荐重载它，因为这会与构造器选项 *allow_no_value* 和 *delimiters* 产生冲突。

14.2.8 旧式 API 示例

主要出于向下兼容性的考虑，*configparser* 还提供了一种采用显式 get/set 方法的旧式 API。虽然以下介绍的方法存在有效的用例，但对于新项目仍建议采用映射协议访问。旧式 API 在多数时候都更复杂、更底层并且完全违反直觉。

一个写入配置文件的示例:

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

一个再次读取配置文件的示例:

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

要获取插值，请使用*ConfigParser*:

```
import configparser
```

(下页继续)

(续上页)

```
cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is %(baz)s!"

# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
      # -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

默认值在两种类型的 ConfigParser 中均可用。它们将在当某个选项未在别处定义时被用于插值。

```
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

print(config.get('Section1', 'foo'))      # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))      # -> "Life is hard!"
```

14.2.9 ConfigParser 对象

class configparser.**ConfigParser**(*defaults=None*, *dict_type=collections.OrderedDict*, *allow_no_value=False*, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*)

主配置解析器。当给定 *defaults* 时，它会被初始化为包含固有默认值的字典。当给定 *dict_type* 时，它将被用来创建包含节、节中的选项以及默认值的字典。

当给定 *delimiters* 时，它会被用作分隔键与值的子字符串的集合。当给定 *comment_prefixes* 时，它将被用作在否则为空行的注释的前缀子字符串的集合。注释可以被缩进。当给定 *inline_comment_prefixes* 时，

它将被用作非空行的注释的前缀子字符串的集合。

当 *strict* 为 `True` (默认值) 时，解析器在从单个源（文件、字符串或字典）读取时将不允许任何节或选项出现重复，否则会引发*DuplicateSectionError* 或*DuplicateOptionError*。当 *empty_lines_in_values* 为 `False` (默认值: `True`) 时，每个空行均表示一个选项的结束。在其他情况下，一个多行选项内部的空行会被保留为值的一部分。当 *allow_no_value* 为 `True` (默认值: `False`) 时，将接受没有值的选项；此种选项的值将为 `None` 并且它们会以不带末尾分隔符的形式被序列化。

当给定 *default_section* 时，它将指定存放其他节的默认值和用于插值的特殊节的名称 (通常命名为 `"DEFAULT"`)。该值可通过使用 `default_section` 实例属性在运行时被读取或修改。

插值行为可通过给出 *interpolation* 参数提供自定义处理程序的方式来定制。`None` 可用来完全禁用插值，`ExtendedInterpolation()` 提供了一种更高级的变体形式，它的设计受到了 `zc.buildout` 的启发。有关该主题的更多信息请参见专门的文档章节。

插值中使用的所有选项名称将像任何其他选项名称引用一样通过*optionxform()* 方法来传递。例如，使用*optionxform()* 的默认实现（它会将选项名称转换为小写形式）时，值 `foo %(bar)s` 和 `foo %(BAR)s` 是等价的。

当给定 *converters* 时，它应当为一个字典，其中每个键代表一个类型转换器的名称而每个值则为实现从字符串到目标数据类型的转换的可调用对象。每个转换器会获得其在解析器对象和节代理上对应的 `get*()` 方法。

在 3.1 版更改: 默认的 *dict_type* 为*collections.OrderedDict*。

在 3.2 版更改: 添加了 *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* 以及 *interpolation*。

在 3.5 版更改: 添加了 *converters* 参数。

defaults()
返回包含实例范围内默认值的字典。

sections()
返回可用节的列表；*default section* 不包括在该列表中。

add_section(*section*)
向实例添加一个名为 *section* 的节。如果给定名称的节已存在，将会引发*DuplicateSectionError*。如果转入了 *default section* 名称，则会引发*ValueError*。节名称必须为字符串；如果不是则会引发*TypeError*。

在 3.2 版更改: 非字符串的节名称将引发*TypeError*。

has_section(*section*)
指明相应名称的 *section* 是否存在于配置中。*default section* 不包含在内。

options(*section*)
返回指定 *section* 中可用选项的列表。

has_option(*section*, *option*)
如果给定的 *section* 存在并且包含给定的 *option* 则返回*True*；否则返回*False*。如果指定的 *section* 为*None* 或空字符串，则会使用 DEFAULT。

read(*filenames*, *encoding=None*)
尝试读取并解析一个包含文件名的可迭代对象，返回一个被成功解析的文件名列表。

If *filenames* is a string or *path-like object*, it is treated as a single filename. If a file named in *filenames* cannot be opened, that file will be ignored. This is designed so that you can specify an iterable of potential configuration file locations (for example, the current directory, the user's home directory, and some system-wide directory), and all existing configuration files in the iterable will be read.

如果名称对应的文件全都不存在，则*ConfigParser* 实例将包含一个空数据集。一个要求从文件加载初始值的应用应当在调用*read()* 来获取任何可选文件之前使用*read_file()* 来加载所要求的一个或多个文件:

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

3.2 新版功能: *encoding* 形参。在之前的版本中，所有文件都将使用*open()* 的默认编码格式来读取。

3.6.1 新版功能: *filenames* 形参接受一个*path-like object*。

read_file(*f*, *source=None*)
从 *f* 读取并解析配置数据，它必须是一个产生 Unicode 字符串的可迭代对象（例如以文本模式打开的文件）。

可选参数 *source* 指定要读取的文件名称。如果未给出并且 *f* 具有 `name` 属性，则该属性会被用作 *source*；默认值为 `'<???>'`。

3.2 新版功能: 替代*readfp()*。

read_string(*string*, *source='<string>'*)
从字符串中解析配置数据。

可选参数 *source* 指定一个所传入字符串的上下文专属名称。如果未给出，则会使用 `'<string>'`。这通常应为一个文件系统路径或 URL。

3.2 新版功能.

read_dict(*dictionary*, *source='<dict>'*)
从任意一个提供了类似于字典的 `items()` 方法的对象加载配置。键为节名称，值为包含节中所出现的键和值的字典。如果所用的字典类型会保留顺序，则节和其中的键将按顺序加入。值会被自动转换为字符串。

可选参数 *source* 指定一个所传入字曲的上下文专属名称。如果未给出，则会使用 `<dict>`。

此方法可被用于在解析器之间拷贝状态。

3.2 新版功能.

get(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
获取指定名称的 *section* 的一个 *option* 的值。如果提供了 *vars*，则它必须为一个字典。*option* 的查找顺序为 *vars** (如果有提供)、**section* 以及 *DEFAULTSECT*。如果未找到该键并且提供了 *fallback*，则它会被用作回退值。可以提供 `None` 作为 *fallback* 值。

所有 `'%'` 插值会在返回值中被展开，除非 *raw* 参数为真值。插值键所使用的值会按与选项相同的方式来查找。

在 3.2 版更改: *raw*, *vars* 和 *fallback* 都是仅限关键字参数，以防止用户试图使用第三个参数作业为 *fallback* 回退值（特别是在使用映射协议的时候）。

getint(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
将在 *section* 中指定的 *option* 强制转换为整数的便捷方法。参见*get()* 获取对于 *raw*, *vars* 和 *fallback* 的解释。

getfloat(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
将在 *section* 中指定的 *option* 强制转换为浮点数的便捷方法。参见*get()* 获取对于 *raw*, *vars* 和 *fallback* 的解释。

getboolean(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
将在指定 *section* 中的 *option* 强制转换为布尔值的便捷方法。请注意选项所接受的值为 `'1'`, `'yes'`, `'true'` 和 `'on'`，它们会使得此方法返回 `True`，以及 `'0'`, `'no'`, `'false'` 和 `'off'`，它们会使得此方法返回 `False`。这些字符串值会以对大小写不敏感的方式被检测。任何其他值都将导致引发*ValueError*。参见*get()* 获取对于 *raw*, *vars* 和 *fallback* 的解释。

items(*raw=False*, *vars=None*)
items(*section*, *raw=False*, *vars=None*)
当未给出 *section* 时，将返回由 *section_name*, *section_proxy* 对组成的列表，包括 DEFAULTSECT。

在其他情况下，将返回给定的 *section* 中的 option 的 *name*, *value* 对组成的列表。可选参数具有与*get()* 方法的参数相同的含义。

set(*section*, *option*, *value*)
如果给定的节存在，则将所给出的选项设为指定的值；在其他情况下将引发*NoSectionError*。*option* 和 *value* 必须为字符串；如果不是则将引发*TypeError*。

write(*fileobject*, *space_around_delimiters=True*)
将配置的表示形式写入指定的*file object*，该对象必须以文本模式打开（接受字符串）。此表示形式可由将来的*read()* 调用进行解析。如果 *space_around_delimiters* 为真值，键和值之前的分隔符两边将加上空格。

remove_option(*section*, *option*)
将指定的 *option* 从指定的 *section* 中移除。如果指定的节不存在则会引发*NoSectionError*。如果要移除的选项存在则返回*True*；在其他情况下将返回*False*。

remove_section(*section*)
从配置中移除指定的 *section*。如果指定的节确实存在则返回 `True`。在其他情况下将返回 `False`。

optionxform(*option*)
将选项名 *option* 转换为输入文件中的形式或客户端代码所传入的应当在内部结构中使用的形式。默认实现将返回 *option* 的小写形式版本；子类可以重载此行为，或者客户端代码也可以在实例上设置一个具有此名称的属性来影响此行为。

你不需要子类化解析器来使用此方法，你也可以在一个实例上设置它，或使用一个接受字符串参数并返回字符串的函数。例如将它设为 `str` 将使得选项名称变得大小写敏感:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

请注意当读取配置文件时，选项名称两边的空格将在调用*optionxform()* 之前被去除。

readfp(*fp*, *filename=None*)
3.2 版后已移除: 使用*read_file()* 来代替。

在 3.2 版更改: *readfp()* 现在将在 *fp* 上执行迭代而不是调用 `fp.readline()`。

对于调用*readfp()* 时传入不支持迭代的参数的现有代码，可以在文件类对象外使用以下生成器作为包装器:

```
def readline_generator(fp):
    line = fp.readline()
    while line:
        yield line
        line = fp.readline()
```

不再使用 `parser.readfp(fp)` 而是改用 `parser.read_file(readline_generator(fp))`。

configparser.**MAX_INTERPOLATION_DEPTH**
当 *raw* 形参为假值时 `get()` 所采用的递归插值的最大深度。这只在使用默认的 *interpolation* 时会起作用。

14.2.10 RawConfigParser 对象

class configparser.**RawConfigParser**(*defaults=None*, *dict_type=collections.OrderedDict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*[, *interpolation*])

Legacy variant of the *ConfigParser* with interpolation disabled by default and unsafe `add_section` and `set` methods.

注解: 考虑改用*ConfigParser*，它会检查内部保存的值的类型。如果你不想要插值，你可以使用 `ConfigParser(interpolation=None)`。

---

add_section(*section*)

向实例添加一个名为 *section* 的节。如果给定名称的节已存在，将会引发*DuplicateSectionError*。如果传入了 *default section* 名称，则会引发*ValueError*。

不检查 *section* 以允许用户创建以非字符串命名的节。此行为已不受支持并可能导致内部错误。

set(*section*, *option*, *value*)

如果给定的节存在，则将给定的选项设为指定的值；在其他情况下将引发*NoSectionError*。虽然可能使用*RawConfigParser*（或使用*ConfigParser* 并将 *raw* 形参设为真值）以便实现非字符串值的 *internal* 存储，但是完整功能（包括插值和输出到文件）只能使用字符串值来实现。

此方法允许用户在内部将非字符串值赋给键。此行为已不受支持并会在尝试写入到文件或在非原始模式下获取数据时导致错误。**请使用映射协议 API**，它不允许出现这样的赋值。

14.2.11 异常

exception configparser.**Error**

所有其他*configparser* 异常的基类。

exception configparser.**NoSectionError**

当找不到指定节时引发的异常。

exception configparser.**DuplicateSectionError**

当调用 `add_section()` 时传入已存在的节名称，或者在严格解析器中当单个输入文件、字符串或字典内出现重复的节时引发的异常。

3.2 新版功能: 将可选的 `source` 和 `lineno` 属性和参数添加到 `__init__()`。

exception configparser.**DuplicateOptionError**

当单个选项在从单个文件、字符串或字典读取时出现两次时引发的异常。这会捕获拼写错误和大小写敏感相关的错误，例如一个字典可能包含两个键分别代表同一个大小写不敏感的配置键。

exception configparser.**NoOptionError**

当指定的选项未在指定的节中被找到时引发的异常。

exception configparser.**InterpolationError**

当执行字符串插值发生问题时所引发的异常的基类。

exception configparser.**InterpolationDepthError**

当字符串插值由于迭代次数超出*MAX_INTERPOLATION_DEPTH* 而无法完成所引发的异常。为*InterpolationError* 的子类。

exception configparser.**InterpolationMissingOptionError**

当从某个值引用的选项并不存在时引发的异常。为*InterpolationError* 的子类。

exception configparser.**InterpolationSyntaxError**
当将要执行替换的源文本不符合要求的语法时引发的异常。为 *InterpolationError* 的子类。

exception configparser.**MissingSectionHeaderError**
当尝试解析一个不带节标头的文件时引发的异常。

exception configparser.**ParsingError**
当尝试解析一个文件而发生错误时引发的异常。

在 3.2 版更改: filename 属性和 __init__() 参数被重命名为 source 以保持一致性。

备注

14.3 netrc —netrc 文件处理

源代码: Lib/netrc.py

---

netrc 类解析并封装了 Unix 的 **ftp** 程序和其他 FTP 客户端所使用的 netrc 文件格式。

class netrc.**netrc**([*file*])
A *netrc* instance or subclass instance encapsulates data from a netrc file. The initialization argument, if present, specifies the file to parse. If no argument is given, the file .netrc in the user' s home directory will be read. Parse errors will raise *NetrcParseError* with diagnostic information including the file name, line number, and terminating token. If no argument is specified on a POSIX system, the presence of passwords in the .netrc file will raise a *NetrcParseError* if the file ownership or permissions are insecure (owned by a user other than the user running the process, or accessible for read or write by any other user). This implements security behavior equivalent to that of ftp and other programs that use .netrc.

在 3.4 版更改: 添加了 POSIX 权限检查。

exception netrc.**NetrcParseError**
当在源文本中遇到语法错误时由 *netrc* 类引发的异常。此异常的实例提供了三个有用属性: msg 为错误的文本说明，filename 为源文件的名称，而 lineno 给出了错误所在的行号。

14.3.1 netrc 对象

netrc 实例具有下列方法:

netrc.**authenticators**(*host*)
针对 *host* 的身份验证者返回一个 3 元组 (login, account, password)。如果 netrc 文件不包含针对给定主机的条目，则返回关联到 'default' 条目的元组。如果匹配的主机或默认条目均不可用，则返回 None。

netrc.**__repr__**()
将类数据以 netrc 文件的格式转储为一个字符串。(这会丢弃注释并可能重排条目顺序。)

netrc 的实例具有一些公共实例变量:

netrc.**hosts**
将主机名映射到 (login, account, password) 元组的字典。如果存在 'default' 条目，则会表示为使用该名称的伪主机。

netrc.**macros**
将宏名称映射到字符串列表的字典。

注解: 密码会被限制为 ASCII 字符集的一个子集。所有 ASCII 标点符号均可用作密码，但是要注意空白符和非打印字符不允许用作密码。这是.netrc 文件解析方式带来的限制，在未来可能会被解除。

14.4 xdrlib —编码与解码 XDR 数据

源代码: Lib/xdrlib.py

xdrlib 模块为外部数据表示标准提供支持，该标准的描述见 **RFC 1014**，由 Sun Microsystems, Inc. 在 1987 年 6 月撰写。它支持该 RFC 中描述的大部分数据类型。

xdrlib 模块定义了两个类，一个用于将变量打包为 XDR 表示形式，另一个用于从 XDR 表示形式解包。此外还有两个异常类。

class xdrlib.**Packer**
Packer 是用于将数据打包为 XDR 表示形式的类。*Packer* 类的实例化不附带参数。

class xdrlib.**Unpacker**(*data*)
Unpacker 是用于相应地从字符串缓冲区解包 XDR 数据值的类。输入缓冲区将作为 *data* 给出。

参见:

RFC 1014 - XDR: 外部数据表示标准 这个 RFC 定义了最初编写此模块时 XDR 所用的数据编码格式。显然它已被 **RFC 1832** 所淘汰。

RFC 1832 - XDR: 外部数据表示标准 更新的 RFC，它提供了经修订的 XDR 定义。

14.4.1 Packer 对象

Packer 实例具有下列方法:

Packer.**get_buffer**()
将当前打包缓冲区以字符串的形式返回。

Packer.**reset**()
将打包缓冲区重置为空字符串。

总体来说，你可以通过调用适当的 pack_type() 方法来打包任何最常见的 XDR 数据类型。每个方法都是接受单个参数，即要打包的值。受支持的简单数据类型打包方法如下: pack_uint(), pack_int(), pack_enum(), pack_bool(), pack_uhyper() 以及 pack_hyper()。

Packer.**pack_float**(*value*)
打包单精度浮点数 *value*。

Packer.**pack_double**(*value*)
打包双精度浮点数 *value*。

以下方法支持打包字符串、字节串以及不透明数据。

Packer.**pack_fstring**(*n*, *s*)
打包固定长度字符串 *s*。*n* 为字符串的长度，但它 不会被打包进数据缓冲区。如有必要字符串会以空字节串填充以保证 4 字节对齐。

Packer.**pack_fopaque**(*n*, *data*)
打包固定长度不透明数据流，类似于*pack_fstring()*。

Packer.**pack_string**(*s*)
打包可变长度字符串 *s*。先将字符串的长度打包为无符号整数，再用*pack_fstring()* 来打包字符串数据。

Packer.**pack_opaque**(*data*)
打包可变长度不透明数据流，类似于*pack_string()*。

Packer.**pack_bytes**(*bytes*)
打包可变长度字节流，类似于*pack_string()*。

下列方法支持打包数组和列表:

Packer.**pack_list**(*list*, *pack_item*)
打包由同质条目构成的 *list*。此方法适用于不确定长度的列表；即其长度无法在遍历整个列表之前获知。对于列表中的每个条目，先打包一个无符号整数 `1`，再添加列表中数据的值。*pack_item* 是在打包单个条目时要调用的函数。在列表的末尾，会再打包一个无符号整数 `0`.

例如，要打包一个整数列表，代码看起来会是这样:

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

Packer.**pack_farray**(*n*, *array*, *pack_item*)
打包由同质条目构成的固定长度列表 (*array*)。*n* 为列表长度；它 不会被打包到缓冲区，但是如果 `len(array)` 不等于 *n* 则会引发*ValueError*。如上所述，*pack_item* 是在打包每个元素时要使用的函数。

Packer.**pack_array**(*list*, *pack_item*)
打包由同质条目构成的可变长度 *list*。先将列表的长度打包为无符号整数，再像上面的*pack_farray()* 一样打包每个元素。

14.4.2 Unpacker 对象

Unpacker 类提供以下方法:

Unpacker.**reset**(*data*)
使用给定的 *data* 重置字符串缓冲区。

Unpacker.**get_position**()
返回数据缓冲区中的当前解包位置。

Unpacker.**set_position**(*position*)
将数据缓冲区的解包位置设为 *position*。你应当小心使用*get_position()* 和*set_position()*。

Unpacker.**get_buffer**()
将当前解包数据缓冲区以字符串的形式返回。

Unpacker.**done**()
表明解包完成。如果数据没有全部完成解包则会引发*Error* 异常。

此外，每种可通过*Packer* 打包的数据类型都可通过*Unpacker* 来解包。解包方法的形式为 `unpack_type()`，并且不接受任何参数。该方法将返回解包后的对象。

Unpacker.**unpack_float**()
解包单精度浮点数。

Unpacker.**unpack_double**()
解包双精度浮点数，类似于*unpack_float()*。

此外，以下方法可用来解包字符串、字节串以及不透明数据:

Unpacker.**unpack_fstring**(*n*)
解包并返回固定长度字符串。*n* 为期望的字符数量。会预设以空字节串填充以保证 4 字节对齐。

Unpacker.**unpack_fopaque**(*n*)
解包并返回固定长度数据流，类似于*unpack_fstring()*。

Unpacker.**unpack_string**()
解包并返回可变长度字符串。先将字符串的长度解包为无符号整数，再用*unpack_fstring()* 来解包字符串数据。

Unpacker.**unpack_opaque**()
解包并返回可变长度不透明数据流，类似于*unpack_string()*。

Unpacker.**unpack_bytes**()
解包并返回可变长度字节流，类似于*unpack_string()*。

下列方法支持解包数组和列表:

Unpacker.**unpack_list**(*unpack_item*)
解包并返回同质条目的列表。该列表每次解包一个元素，先解包一个无符号整数旗标。如果旗标为 1，则解包条目并将其添加到列表。旗标为 0 表明列表结束。*unpack_item* 为在解包条目时调用的函数。

Unpacker.**unpack_farray**(*n*, *unpack_item*)
解包并（以列表形式）返回由同质条目构成的固定长度数组。*n* 为期望的缓冲区内列表元素数量。如上所述，*unpack_item* 是解包每个元素时要使用的函数。

Unpacker.**unpack_array**(*unpack_item*)
解包并返回由同质条目构成的可变长度 *list*。先将列表的长度解包为无符号整数，再像上面的*unpack_farray()* 一样解包每个元素。

14.4.3 异常

此模块中的异常会表示为类实例代码:

exception xdrlib.**Error**
基本异常类。*Error* 具有一个公共属性 msg，其中包含对错误的描述。

exception xdrlib.**ConversionError**
从*Error* 所派生的类。不包含额外的实例变量。

以下是一个应该如何捕获这些异常的示例:

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print('packing the double failed:', instance.msg)
```

14.5 plistlib —生成与解析 Mac OS X .plist 文件

源代码: Lib/plistlib.py

此模块提供了读写主要用于 Mac OS X 的“property list”文件的接口，并同时支持二进制和 XML plist 文件。

property list (`.plist`) 文件格式是一种简单的序列化格式，它支持一些基本对象类型，例如字典、列表、数字和字符串等。通常使用一个字典作为最高层级对象。

要写入和解析 plist 文件，请使用*dump()* 和*load()* 函数。

要以字节串对象形式操作 plist 数据，请使用*dumps()* 和*loads()*。

值可以为字符串、整数、浮点数、布尔值、元组、列表、字典（但只允许用字符串作为键）、*Data*、*bytes*、bytesarray 或*datetime.datetime* 对象。

在 3.4 版更改: 新版 API，旧版 API 已被弃用。添加了对二进制 plist 格式的支持。

参见:

PList manual page 针对该文件格式的 Apple 文档。

这个模块定义了以下函数:

plistlib.**load**(*fp*, *, *fmt=None*, *use_builtin_types=True*, *dict_type=dict*)
读取 plist 文件。*fp* 应当可读并且为二进制文件对象。返回已解包的根对象（通常是一个字典）。

fmt 为文件的格式，有效的值如下:

- *None*: 自动检测文件格式
- *FMT_XML*: XML 文件格式
- *FMT_BINARY*: 二进制 plist 格式

如果 *use_builtin_types* 为真值（默认）则将以*bytes* 实例的形式返回二进制数据，否则将以*Data* 实例的形式返回。

The *dict_type* is the type used for dictionaries that are read from the plist file. The exact structure of the plist can be recovered by using *collections.OrderedDict* (although the order of keys shouldn’t be important in plist files).

FMT_XML 格式的 XML 数据会使用来自*xml.parsers.expat* 的 Expat 解析器–请参阅其文档了解错误格式 XML 可能引发的异常。未知元素将被 plist 解析器直接略过。

当文件无法被解析时二进制格式的解析器将引发 InvalidFileException。

3.4 新版功能.

plistlib.**loads**(*data*, *, *fmt=None*, *use_builtin_types=True*, *dict_type=dict*)
从一个 bytes 对象加载 plist。参阅*load()* 获取相应关键字参数的说明。

3.4 新版功能.

plistlib.**dump**(*value*, *fp*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*)
将 *value* 写入 plist 文件。*Fp* 应当可写并且为二进制文件对象。

fmt 参数指定 plist 文件的格式，可以是以下值之一:

- *FMT_XML*: XML 格式的 plist 文件
- *FMT_BINARY*: 二进制格式的 plist 文件

当 *sort_keys* 为真值（默认）时字典的键将经过排序再写入 plist，否则将按字典的迭代顺序写入。

当 *skipkeys* 为假值（默认）时该函数将在字典的键不为字符串时引发`TypeError`，否则将跳过这样的键。

如果对象是不受支持的类型或者是包含不受支持类型的对象的容器则将引发`TypeError`。

对于无法在（二进制）plist 文件中表示的整数值，将会引发`OverflowError`。

3.4 新版功能.

plistlib.**dumps**(*value*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*)
将 *value* 以 plist 格式字节串对象的形式返回。参阅`dump()` 的文档获取此函数的关键字参数的说明。

3.4 新版功能.

以下函数已被弃用:

plistlib.**readPlist**(*pathOrFile*)
读取 plist 文件。*pathOrFile* 可以是文件名或（可读并且为二进制的）文件对象。返回已解包的根对象（通常是一个字典）。

此函数会调用`load()` 来完成实际操作，请参阅该函数 的文档获取相应关键字参数的说明。

注解: Dict values in the result have a __getattr__ method that defers to __getitem_. This means that you can use attribute access to access items of these dictionaries.

3.4 版后已移除: 使用`load()` 来代替。

plistlib.**writePlist**(*rootObject*, *pathOrFile*)
将 *rootObject* 写入 XML plist 文件。*pathOrFile* 可以是文件名或（可写并且为二进制的）文件对象。

3.4 版后已移除: 使用`dump()` 来代替。

plistlib.**readPlistFromBytes**(*data*)
从字节串对象读取 plist 数据。返回根对象。

参阅`load()` 获取相应关键字参数的说明。

注解: Dict values in the result have a __getattr__ method that defers to __getitem_. This means that you can use attribute access to access items of these dictionaries.

3.4 版后已移除: 使用`loads()` 来代替。

plistlib.**writePlistToBytes**(*rootObject*)
将 *rootObject* 作为 XML plist 格式的字节串对象返回。

3.4 版后已移除: 使用`dumps()` 来代替。

可以使用以下的类:

Dict([dict]):
Return an extended mapping object with the same value as dictionary *dict*.

This class is a subclass of `dict` where attribute access can be used to access items. That is, `aDict.key` is the same as `aDict['key']` for getting, setting and deleting items in the mapping.

3.0 版后已移除.

class plistlib.**Data**(*data*)
返回一个包含字节串对象 *data* 的“data”包装器对象。该类将在对 plist 进行双向转换的函数中被用来代表 plist 中可用的 <data> 类型。

它具有一个属性 `data`，可被用来提取其中所保存的 Python 字节串对象。

3.4 版后已移除: 使用*bytes* 对象来代替。

可以使用以下的常量:

`plistlib.`**FMT_XML**
用于 plist 文件的 XML 格式。

3.4 新版功能.

`plistlib.`**FMT_BINARY**
用于 plist 文件的二进制格式。

3.4 新版功能.

14.5.1 例子

生成一个 plist:

```
pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime())),
)
with open(fileName, 'wb') as fp:
    dump(pl, fp)
```

解析一个 plist:

```
with open(fileName, 'rb') as fp:
    pl = load(fp)
print(pl["aKey"])
```

CHAPTER 15

加密服务

本章中描述的模块实现了加密性质的各种算法。它们可由安装人员自行决定。在 Unix 系统上，*crypt* 模块也可以使用。这是一个概述：

15.1 hashlib —安全哈希与消息摘要

源码： Lib/hashlib.py

这个模块针对不同的安全哈希和消息摘要算法实现了一个通用的接口。包括 FIPS 的 SHA1, SHA224, SHA256, SHA384, and SHA512 (定义于 FIPS 180-2) 算法，以及 RSA 的 MD5 算法 (定义于 Internet **RFC 1321**)。术语“安全哈希”和“消息摘要”是可互换的，较旧的算法被称为消息摘要，现代术语是安全哈希。

注解： 如果你想找到 adler32 或 crc32 哈希函数，它们在*zlib* 模块中。

警告： 有些算法已知存在哈希碰撞弱点，请参考最后的“另请参阅”段。

15.1.1 哈希算法

每种类型的 *hash* 都有一个构造器方法。它们都返回一个具有相同的简单接口的 hash 对象。例如，使用 use `sha256()` 创建一个 SHA-256 hash 对象。你可以使用 `update()` 方法向这个对象输入*字节类对象* (通常是*bytes*)。在任何时候你都可以使用 `digest()` 或 `hexdigest()` 方法获得到目前为止输入这个对象的拼接数据的 *digest*。

注解： 为了更好的多线程性能，在对象创建或者更新时，若数据大于 2047 字节则 Python 的*GIL* 会被释放。

注解: 向 update() 输入字符串对象是不被支持的，因为哈希基于字节而非字符。

此模块中总是可用的哈希算法构造器有 sha1(), sha224(), sha256(), sha384(), sha512(), *blake2b()* 和*blake2s()*。md5() 通常也是可用的，但如果你在使用少见的 “FIPS 兼容” 的 Python 编译版本则可能会找不到它。此外还可能有一些附加的算法，具体取决于你的平台上的 Python 所使用的 OpenSSL 库。在大部分平台上可用的还有 sha3_224(), sha3_256(), sha3_384(), sha3_512(), shake_128(), shake_256() 等等。

3.6 新版功能: SHA3 (Keccak) 和 SHAKE 构造器 sha3_224(), sha3_256(), sha3_384(), sha3_512(), shake_128(), shake_256().

3.6 新版功能: 添加了*blake2b()* 和*blake2s()* 。

例如，如果想获取字节串 b'Nobody inspects the spammish repetition' 的摘要:

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\x95\
↪x0fK\x94\x06'
>>> m.digest_size
32
>>> m.block_size
64
```

更简要的写法:

```
>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

hashlib.**new**(*name*[, *data*])
一个接受所希望的算法对应的字符串 *name* 作为第一个形参的通用构造器。它还允许访问上面列出的哈希算法以及你的 OpenSSL 库可能提供的任何其他算法。同名的构造器要比*new()* 更快所以应当优先使用。

使用*new()* 并附带由 OpenSSL 所提供了算法:

```
>>> h = hashlib.new('ripemd160')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Hashlib 提供下列常量属性:

hashlib.**algorithms_guaranteed**
一个集合，其中包含此模块在所有平台上都保证支持的哈希算法的名称。请注意‘md5’也在此清单中，虽然某些上游厂商提供了一个怪异的排除了此算法的 “FIPS 兼容” Python 编译版本。

3.2 新版功能.

hashlib.**algorithms_available**
一个集合，其中包含在所运行的 Python 解释器上可用的哈希算法的名称。将这些名称传给*new()* 时将可被识别。*algorithms_guaranteed* 将总是它的一个子集。同样的算法在此集合中可能以不同的名称出现多次（这是 OpenSSL 的原因）。

3.2 新版功能.

下列值会以构造器所返回的哈希对象的常量属性的形式被提供:

hash.**digest_size**
以字节表示的结果哈希对象的大小。

hash.**block_size**
以字节表示的哈希算法的内部块大小。

hash 对象具有以下属性:

hash.**name**
此哈希对象的规范名称，总是为小写形式并且总是可以作为*new()* 的形参用来创建另一个此类型的哈希对象。

在 3.4 版更改: 该属性名称自被引入起即存在于 CPython 中，但在 Python 3.4 之前并未正式指明，因此可能不存在于某些平台上。

哈希对象具有下列方法:

hash.**update**(*data*)
用*bytes-like object* 来更新哈希对象。重复调用相当于单次调用并传入所有参数的拼接结果: `m.update(a); m.update(b)` 等价于 `m.update(a+b)`。

在 3.1 版更改: 当使用 OpenSSL 提供的哈希算法在大于 2047 字节的数据上执行哈希更新时 Python GIL 会被释放以允许其他线程运行。

hash.**digest**()
返回当前已传给*update()* 方法的数据摘要。这是一个大小为*digest_size* 的字节串对象，字节串中可包含 0 至 255 的完整取值范围。

hash.**hexdigest**()
类似于*digest()* 但摘要会以两倍长度字符串对象的形式返回，其中仅包含十六进制数码。这可以被用于在电子邮件或其他非二进制环境中安全地交换数据值。

hash.**copy**()
返回哈希对象的副本（“克隆”）。这可被用来高效地计算共享相同初始子串的数据的摘要。

15.1.2 SHAKE 可变长度摘要

`shake_128()` 和 `shake_256()` 算法提供安全的 length_in_bits//2 至 128 或 256 位可变长度摘要。为此，它们的摘要需指定一个长度。SHAKE 算法不限制最大长度。

shake.**digest**(*length*)
返回当前已传给 `update()` 方法的数据摘要。这是一个大小为 *length* 的字节串对象，字节串中可包含 0 to 255 的完整取值范围。

shake.**hexdigest**(*length*)
类似于*digest()* 但摘要会以两倍长度字符串对象的形式返回，其中仅包含十六进制数码。这可以被用于在电子邮件或其他非二进制环境中安全地交换数据值。

15.1.3 密钥派生

密钥派生和密钥延展算法被设计用于安全密码哈希。`sha1(password)` 这样的简单算法无法防御暴力攻击。好的密码哈希函数必须可以微调、放慢步调，并且包含 加盐。

`hashlib.`**`pbkdf2_hmac`**(*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)

此函数提供 PKCS#5 基于密码的密钥派生函数 2。它使用 HMAC 作为伪随机函数。

字符串 *hash_name* 是要求用于 HMAC 的哈希摘要算法的名称，例如 'sha1' 或 'sha256'。*password* 和 *salt* 会以字节串缓冲区的形式被解析。应用和库应当将 *password* 限制在合理长度 (例如 1024)。*salt* 应当为适当来源例如`os.urandom()` 的大约 16 个或更多的字节串数据。

iterations 数值应当基于哈希算法和算力来选择。在 2013 年时，建议至少为 100,000 次 SHA-256 迭代。

dklen 为派生密钥的长度。如果 *dklen* 为 `None` 则会使用哈希算法 *hash_name* 的摘要大小，例如 SHA-512 为 64。

```
>>> import hashlib, binascii
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> binascii.hexlify(dk)
b'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

3.4 新版功能.

注解: 随同 OpenSSL 提供了一个快速的 *pbkdf2_hmac* 实现。Python 实现是使用`hmac` 的内联版本。它的速度大约要慢上三倍并且不会释放 GIL。

`hashlib.`**`scrypt`**(*password*, ***, *salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

此函数提供基于密码加密的密钥派生函数，其定义参见 **RFC 7914**。

password 和 *salt* 必须为字节类对象。应用和库应当将 *password* 限制在合理长度 (例如 1024)。*salt* 应当为适当来源例如`os.urandom()` 的大约 16 个或更多的字节串数据。

n is the CPU/Memory cost factor, *r* the block size, *p* parallelization factor and *maxmem* limits memory (OpenSSL 1.1.0 defaults to 32 MB). *dklen* is the length of the derived key.

Availability: OpenSSL 1.1+

3.6 新版功能.

15.1.4 BLAKE2

BLAKE2 是在 **RFC 7693** 中定义的加密哈希函数，它有两种形式:

- **BLAKE2b**，针对 64 位平台进行优化，并会生成长度介于 1 和 64 字节之间任意大小的摘要。
- **BLAKE2s**，针对 8 至 32 位平台进行优化，并会生成长度介于 1 和 32 字节之间任意大小的摘要。

BLAKE2 支持 **keyed mode** (HMAC 的更快速更简单的替代), **salted hashing**, **personalization** 和 **tree hashing**.

此模块的哈希对象遵循标准库`hashlib` 对象的 API。

创建哈希对象

新哈希对象可通过调用构造器函数来创建:

hashlib.**blake2b**(*data=b''*, ***, *digest_size=64*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*)

hashlib.**blake2s**(*data=b''*, ***, *digest_size=32*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*)

这些函数返回用于计算 BLAKE2b 或 BLAKE2s 的相应的哈希对象。它们接受下列可选通用形参:

- *data*: 要哈希的初始数据块，它必须为*bytes-like object*。它只能作为位置参数传入。
- *digest_size*: 以字节数表示的输出摘要大小。
- *key*: 用于密钥哈希的密钥（对于 BLAKE2b 最多 64 字节，对于 BLAKE2s 最多 32 字节）。
- *salt*: 用于随机哈希的盐值（对于 BLAKE2b 最长 16 字节，对于 BLAKE2s 最长 8 字节）。
- *person*: 个性化字符串（对于 BLAKE2b 最长 16 字节，对于 BLAKE2s 最长 8 字节）。

下表显示了常规参数的限制（以字节为单位）:

Hash	目标长度	长度（键）	长度（盐）	长度（个人）
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

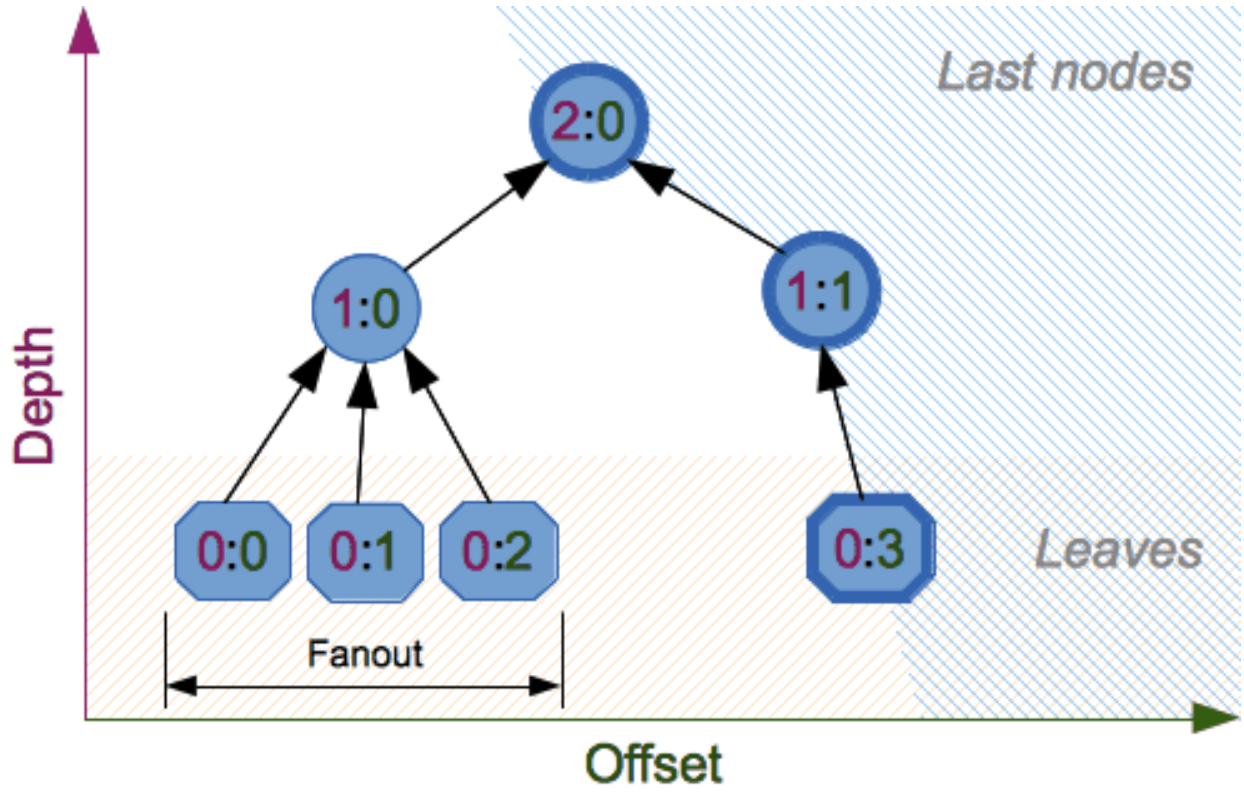
注解: BLAKE2 规格描述为盐值和个性化形参定义了固定的长度，但是为了方便起见，此实现接受指定在长度以内的任意大小的字节串。如果形参长度小于指定值，它将以零值进行填充，因此举例来说，`b'salt'` 和 `b'salt\x00'` 为相同的值 (*key* 的情况则并非如此。)

如下面的模块*constants* 所描述，这些是可用的大小取值。

构造器函数还接受下列树形哈希形参:

- *fanout*: 扇出值 (0 至 255，如无限制即为 0，连续模式下为 1)。
- *depth*: 树的最大深度 (1 至 255，如无限制则为 255，连续模式下为 1)。
- *leaf_size*: 叶子的最大字节长度 (0 至 2**32-1，如无限制或在连续模式下则为 0)。
- *node_offset*: 节点偏移量 (对于 BLAKE2b 为 0 至 2**64-1，对于 BLAKE2s 为 0 至 2**48-1，对于最左边第一个叶子或在连续模式下则为 0)。
- *node_depth*: 节点深度 (0 至 255，对于叶子或在连续模式下则为 0)。
- *inner_size*: 内部摘要大小 (对于 BLAKE2b 为 0 至 64，对于 BLAKE2s 为 0 至 32，连续模式下则为 0)。
- *last_node*: 一个布尔值，指明所处理的节点是否为最后一个 (连续模式下则为 *False*)。

请参阅 BLAKE2 规格描述 第 2.10 节了解有关树形哈希的完整介绍。



常量

blake2b.**SALT_SIZE**

blake2s.**SALT_SIZE**

盐值长度（构造器所接受的最大长度）。

blake2b.**PERSON_SIZE**

blake2s.**PERSON_SIZE**

盐值长度（构造器所接受的最大长度）。

blake2b.**MAX_KEY_SIZE**

blake2s.**MAX_KEY_SIZE**

最大密钥长度。

blake2b.**MAX_DIGEST_SIZE**

blake2s.**MAX_DIGEST_SIZE**

哈希函数可输出的最大摘要长度。

例子

简单哈希

要计算某个数据的哈希值，你应该首先通过调用适当的构造器函数 (*blake2b()* 或 *blake2s()*) 来构造一个哈希对象，然后通过在该对象上调用 `update()` 来更新目标数据，最后通过调用 `digest()` (或针对十六进制编码字符串的 `hexdigest()`) 来获取该对象的摘要。

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

作为快捷方式，你可以直接以位置参数的形式向构造器传入第一个数据块来直接更新:

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

你可以多次调用 *hash.update()* 至你所想要的任意次数以迭代地更新哈希值:

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

使用不同的摘要大小

BLAKE2 具有可配置的摘要大小，对于 BLAKE2b 最多 64 字节，对于 BLAKE2s 最多 32 字节。例如，要使用 BLAKE2b 来替代 SHA-1 而不改变输出大小，我们可以让 BLAKE2b 产生 20 个字节的摘要:

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

不同摘要大小的哈希对象具有完全不同的输出（较短哈希值 *并非* 较长哈希值的前缀）；即使输出长度相同，BLAKE2b 和 BLAKE2s 也会产生不同的输出:

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

密钥哈希

Keyed hashing can be used for authentication as a faster and simpler replacement for Hash-based message authentication code (HMAC). BLAKE2 can be securely used in prefix-MAC mode thanks to the indifferentiability property inherited from BLAKE.

这个例子演示了如何使用密钥 `b'pseudorandom key'` 来为 `b'message data'` 获取一个（十六进制编码的）128 位验证代码:

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

作为实际的例子，一个 Web 应用可为发送给用户的 cookies 进行对称签名，并在之后对其进行验证以确保它们没有被篡改:

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

即使存在原生的密钥哈希模式，BLAKE2 也同样可在*hmac* 模块的 HMAC 构造过程中使用:

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

随机哈希

用户可通过设置 *salt* 形参来为哈希函数引入随机化。随机哈希适用于防止对数字签名中使用的哈希函数进行碰撞攻击。

> 随机哈希被设计用来处理当一方（消息准备者）要生成由另一方（消息签名者）进行签名的全部或部分消息的情况。如果消息准备者能够找到加密哈希函数的碰撞现象（即两条消息产生相同的哈希值），则他们就可以准备将产生相同哈希值和数字签名但却具有不同结果的有意义的消息版本（例如向某个账户转入 $1,000,000 而不是 $10）。加密哈希函数的设计都是以防碰撞性能为其主要目标之一的，但是当前针对加密哈希函数的集中攻击可能导致特定加密哈希函数所提供的防碰撞性能低于预期。随机哈希为签名者提供了额外的保护，可以降低准备者在数字签名生成过程中使得两条或更多条消息最终产生相同哈希值的可能性—即使为特定哈希函数找到碰撞现象是可行的。但是，当消息的所有部分均由签名者准备时，使用随机哈希可能降低数字签名所提供的安全性。
>
> (NIST SP-800-106 "Randomized Hashing for Digital Signatures")

在 BLAKE2 中，盐值会在初始化期间作为对哈希函数的一次性输入而不是对每个压缩函数的输入来处理。

警告： 使用 BLAKE2 或任何其他通用加密哈希函数例如 SHA-256 进行 *加盐哈希* (或纯哈希) 并不适用于哈希密码。请参阅 BLAKE2 FAQ 了解更多信息。

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

个性化

出于不同的目的强制让哈希函数为相同的输入生成不同的摘要有时也是有用的。正如 Skein 哈希函数的作者所言:

> 我们建议所有应用设计者慎重考虑这种做法；我们已看到有许多协议在协议的某一部分中计算出来的哈希值在另一个完全不同的部分中也可以被使用，因为两次哈希计算是针对类似或相关的数据进行的，这样攻击者可以强制应用为相同的输入生成哈希值。个性化协议中所使用的每个哈希函数将有效地阻止这种类型的攻击。
>
> (Skein 哈希函数族, p. 21)

BLAKE2 可通过向 *person* 参数传入字节串来进行个性化:

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

个性化配合密钥模式也可被用来从单个密钥派生出多个不同密钥。

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

树形模式

以下是对包含两个叶子节点的最小树进行哈希的例子:

```
  10
 /  \
00  01
```

这个例子使用 64 字节内部摘要，返回 32 字节最终摘要:

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
```

(下页继续)

(续上页)

```
>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

开发人员

BLAKE2 是由 *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O'Hearn* 和 *Christian Winnerlein* 基于 *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier* 和 *Raphael C.-W. Phan* 所创造的 SHA-3 入围方案 BLAKE 进行设计的。

它使用的核心算法来自由 *Daniel J. Bernstein* 所设计的 ChaCha 加密。

stdlib 实现是基于 pyblake2 模块的。它由 *Dmitry Chestnykh* 在 *Samuel Neves* 所编写的 C 实现的基础上编写。此文档拷贝自 pyblake2 并由 *Dmitry Chestnykh* 撰写。

C 代码由 *Christian Heimes* 针对 Python 进行了部分的重写。

以下公共领域贡献同时适用于 C 哈希函数实现、扩展代码和本文档:

> 在法律许可的范围内，作者已将此软件的全部版权以及关联和邻接权利贡献到全球公共领域。此软件的发布不附带任何担保。
>
> You should have received a copy of the CC0 Public Domain Dedication along with this software. If not, see http://creativecommons.org/publicdomain/zero/1.0/.

根据创意分享公共领域贡献 1.0 通用规范，下列人士为此项目的开发提供了帮助或对公共领域的修改作出了贡献:

- *Alexandr Sokolovskiy*

参见:

模块*hmac* 使用哈希运算来生成消息验证代码的模块。

模块*base64* 针对非二进制环境对二进制哈希值进行编辑的另一种方式。

https://blake2.net BLAKE2 官方网站

http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf 有关安全哈希算法的 FIPS 180-2 出版物。

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms 包含关于哪些算法存在已知问题以及对其使用所造成的影响的信息的 Wikipedia 文章。

https://www.ietf.org/rfc/rfc2898.txt PKCS #5: 基于密码的加密规范描述 2.0 版

15.2 hmac —基于密钥的消息验证

源代码: Lib/hmac.py

此模块实现了 HMAC 算法，算法的描述参见 **RFC 2104**。

hmac.**new**(*key*, *msg=None*, *digestmod=None*)
返回一个新的 hmac 对象。*key* 是一个指定密钥的 bytes 或 bytearray 对象。如果提供了 *msg*，将会调用 `update(msg)` 方法。*digestmod* 为 HMAC 对象要使用的摘要名称、摘要构造器或模块。它支持任何适用于*hashlib.new()* 的名称并默认设为 `hashlib.md5` 构造器。

在 3.4 版更改: 形参 *key* 可以为 bytes 或 bytearray 对象。形参 *msg* 可以为*hashlib* 所支持的任意类型。形参 *digestmod* 可以为某种哈希算法的名称。

3.4 版后已移除: MD5 作为 *digestmod* 的隐式默认摘要已被弃用。

HMAC 对象具有下列方法:

HMAC.**update**(*msg*)
用 *msg* 来更新 hmac 对象。重复调用相当于单次调用并传入所有参数的拼接结果: `m.update(a); m.update(b)` 等价于 `m.update(a + b)`。

在 3.4 版更改: 形参 *msg* 可以为*hashlib* 所支持的任何类型。

HMAC.**digest**()
返回当前已传给*update()* 方法的字节串数据的摘要。这个字节串数据的长度将与传给构造器的摘要的长度 *digest_size* 相同。它可以包含非 ASCII 的字节，包括 NUL 字节。

> **警告:** 在验证例程运行期间将*digest()* 的输出与外部提供的摘要进行比较时，建议使用*compare_digest()* 函数而不是 == 运算符以减少定时攻击防御力的不足。

HMAC.**hexdigest**()
类似于*digest()* 但摘要会以两倍长度字符串的形式返回，其中仅包含十六进制数码。这可以被用于在电子邮件或其他非二进制环境中安全地交换数据值。

> **警告:** 在验证例程运行期间将*hexdigest()* 的输出与外部提供的摘要进行比较时，建议使用*compare_digest()* 函数而不是 == 运算符以减少定时攻击防御力的不足。

HMAC.**copy**()
返回 hmac 对象的副本（“克隆)。这可被用来高效地计算共享相同初始子串的数据的摘要。

hash 对象具有以下属性:

HMAC.**digest_size**
以字节表示的结果 HMAC 摘要的大小。

HMAC.**block_size**
以字节表示的哈希算法的内部块大小。

3.4 新版功能.

HMAC.**name**
HMAC 的规范名称，总是为小写形式，例如 `hmac-md5`。

3.4 新版功能.

这个模块还提供了下列辅助函数:

hmac.**compare_digest**(*a*, *b*)
返回 `a == b`。此函数使用一种经专门设计的方式通过避免基于内容的短路行为来防止定时分析，使得它适合处理密码。*a* 和 *b* 必须为相同的类型：或者是*str*（仅限 ASCII 字符，如*HMAC.hexdigest()* 的返回值），或者是*bytes-like object*。

注解: 如果 *a* 和 *b* 具有不同的长度，或者如果发生了错误，定时攻击在理论上可以获取有关 *a* 和 *b* 的类型和长度信息—但不能获取它们的值。

3.3 新版功能.

参见:

模块*hashlib* 提供安全哈希函数的 Python 模块。

15.3 secrets —生成安全随机数字用于管理密码

3.6 新版功能.

源代码: Lib/secrets.py

---

secrets 模块可用于生成高加密强度的随机数，适应管理密码、账户验证、安全凭据和相关机密数据管理的需要。

特别地，应当优先使用*secrets* 来替代*random* 模块中默认的伪随机数生成器，后者被设计用于建模和仿真，而不适用于安全和加密。

参见:

PEP 506

15.3.1 随机数

通过*secrets* 模块可以访问你的操作系统所能提供的最安全的随机性来源。

class secrets.**SystemRandom**
使用操作系统所提供的最高质量源来生成随机数的类。请参阅*random.SystemRandom* 了解更多细节。

secrets.**choice**(*sequence*)
返回从一个非空序列中随机选取的元素。

secrets.**randbelow**(*n*)
返回一个 [0, *n*) 范围之内的随机整数。

secrets.**randbits**(*k*)
返回一个具有 *k* 个随机比特位的整数。

15.3.2 生成凭据

secrets 模块提供了一些生成安全凭据的函数，适用于诸如密码重置、难以猜测的 URL 之类的应用场景。

secrets.**token_bytes**([*nbytes=None*])

返回一个包含 *nbytes* 个字节的随机字节串。如果 *nbytes* 为 None 或未提供，则会使用一个合理的默认值。

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

secrets.**token_hex**([*nbytes=None*])

返回一个十六进制数码形式的随机字符串。字符串具有 *nbytes* 个随机字节，每个字节转换为两个十六进制数码。如果 *nbytes* 为 None 或未提供，则会使用一个合理的默认值。

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

secrets.**token_urlsafe**([*nbytes=None*])

返回一个 URL 安全的随机字符串，包含 *nbytes* 个随机字节。文本将使用 Base64 编码，因此平均来说每个字节将对应 1.3 个结果字符。如果 *nbytes* 为 None 或未提供，则会使用一个合理的默认值。

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

凭据应当使用多少个字节？

为了在面对 暴力攻击 时保证安全，凭据必须具有足够的随机性。不幸的是，对随机性是否足够的标准会随着计算机越来越强大并能够在更短时间内进行更多猜测而不断提高。在 2015 年时，人们认为 32 字节（256 位）的随机性对于 *secrets* 模块所适合的典型用例来说是足够的。

作为想要自行管理凭据长度的用户，你可以通过为各种 token_* 函数指定一个 *int* 参数来显式地指定凭据要使用多大的随机性。该参数以字节数来表示要使用的随机性大小。

在其他情况下，如果未提供参数，或者如果参数为 None，则 token_* 函数将改用一个合理的默认值。

注解： 该默认值可能在任何时候被改变，包括在维护版本更新的时候。

15.3.3 其他功能

secrets.**compare_digest**(*a*, *b*)

Return True if strings *a* and *b* are equal, otherwise False, in such a way as to reduce the risk of timing attacks. See *hmac.compare_digest()* for additional details.

15.3.4 应用技巧与最佳实践

本节展示了一些使用*secrets*来管理基本安全级别的应用技巧和最佳实践。

生成长度为八个字符的字母数字密码:

```
import string
alphabet = string.ascii_letters + string.digits
password = ''.join(choice(alphabet) for i in range(8))
```

注解: 应用程序不能以可恢复的格式存储密码，无论是用纯文本还是加密。它们应当使用高加密强度的单向（不可恢复）哈希函数来加盐并生成哈希值。

生成长度为十个字符的字母数字密码，其中包含至少一个小写字母，至少一个大写字母以及至少三个数字:

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

Generate an XKCD-style passphrase:

```
# On standard Linux systems, use a convenient dictionary file.
# Other platforms may need to provide their own word-list.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(choice(words) for i in range(4))
```

生成难以猜测的临时 URL，其中包含适合密码恢复应用的安全凭据:

```
url = 'https://mydomain.com/reset=' + token_urlsafe()
```

CHAPTER 16

通用操作系统服务

本章中描述的各模块提供了在（几乎）所有的操作系统上可用的操作系统特性的接口，例如文件和时钟。这些接口通常以 Unix 或 C 接口为参考对象，不过在大多数其他系统上也可用。这里有一个概述：

16.1 os —操作系统接口模块

源代码： Lib/os.py

该模块提供了一些方便使用操作系统相关功能的函数。如果你是想读写一个文件，请参阅*open()*，如果你想操作路径，请参阅*os.path* 模块，如果你想在命令行上读取所有文件中的所有行请参阅*fileinput* 模块。有关创建临时文件和目录的方法，请参阅*tempfile* 模块，对于高级文件目录处理，请参阅*shutil* 模块。

关于这些函数的适用性的说明：

- 所有 Python 内建的操作系统相关的模块的设计都是为了使得在同一功能可用的情况下，保持接口的一致性；例如，函数 `os.stat(path)` 以相同的格式返回关于 *path* 的统计信息（这个函数同时也是起源于 POSIX 接口）。
- 针对特定的操作的拓展同样在可用于*os* 模块，但是使用它们必然会对可移植性产生威胁。
- 所有接受路径或文件名的函数都同时支持字节串和字符串对象，并在返回路径或文件名时使用相应类型的对象作为结果。
- An “Availability: Unix” note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.
- If not separately noted, all functions that claim “Availability: Unix” are supported on Mac OS X, which builds on a Unix core.

注解： All functions in this module raise *OSError* in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

exception os.**error**
内建的*OSError* 异常的一个别名。

os.**name**
导入的依赖特定操作系统的模块的名称。以下名称目前已注册: 'posix', 'nt', 'java'.

参见:

sys.platform 有更详细的描述. *os.uname()* 只给出系统提供的版本信息。

platform 模块对系统的标识有更详细的检查。

16.1.1 文件名，命令行参数，以及环境变量。

在 Python 中，使用字符串类型表示文件名、命令行参数和环境变量。在某些系统上，在将这些字符串传递给操作系统之前，必须将这些字符串解码为字节。Python 使用文件系统编码来执行此转换（请参阅*sys.getfilesystemencoding()* ）。

在 3.1 版更改: 在某些系统上，使用文件系统编码进行转换可能会失败。在这种情况下，Python 会使用*代理转义编码错误处理器*，这意味着在解码时，不可解码的字节被 Unicode 字符 U+DCxx 替换，并且这些字节在编码时再次转换为原始字节。

文件系统编码必须保证成功解码小于 128 的所有字节。如果文件系统编码无法提供此保证，API 函数可能会引发 UnicodeErrors 。

16.1.2 进程参数

这些函数和数据项提供了操作当前进程和用户的信息。

os.**ctermid**()
返回与进程控制终端对应的文件名。

Availability: Unix.

os.**environ**
一个表示字符串环境的*mapping* 对象。例如，environ['HOME'] 是你的主目录（在某些平台上）的路径名，相当于 C 中的 getenv("HOME")。

这个映射是在第一次导入*os* 模块时捕获的，通常作为 Python 启动时处理 site.py 的一部分。除了通过直接修改 os.environ 之外，在此之后对环境所做的更改不会反映在 os.environ 中。

如果平台支持*putenv()* 函数，这个映射除了可以用于查询环境外还能用于修改环境。当这个映射被修改时，*putenv()* 将被自动调用。

在 Unix 系统上，键和值会使用*sys.getfilesystemencoding()* 和 'surrogateescape' 的错误处理。如果你想使用其他的编码，使用*environb*。

注解: 直接调用*putenv()* 并不会影响 os.environ，所以推荐直接修改 “os.environ“。

注解: 在某些平台上，包括 FreeBSD 和 Mac OS X，设置 environ 可能导致内存泄露。参阅 putenv() 的系统文档。

如果平台没有提供*putenv()*，为了使启动的子进程使用修改后的环境，一份修改后的映射会被传给合适的进程创建函数。

如果平台支持*unsetenv()* 函数，你可以通过删除映射中元素的方式来删除对应的环境变量。当一个元素被从 os.environ 删除时，以及 pop() 或 clear() 被调用时，*unsetenv()* 会被自动调用。

os.**environb**
字节版本的*environ*: 一个以字节串表示环境的*mapping* 对象。*environ* 和*environb* 是同步的（修改*environb* 会更新*environ*，反之亦然）。

environb is only available if *supports_bytes_environ* is True.

3.2 新版功能.

os.**chdir**(*path*)
os.**fchdir**(*fd*)
os.**getcwd**()
以上函数请参阅文件和目录 。

os.**fsencode**(*filename*)
编码路径类 文件名为文件系统接受的形式，使用 'surrogateescape' 代理转义编码错误处理器，在 Windows 系统上会使用 'strict'；返回*bytes* 字节类型不变。

fsdecode() 是此函数的逆向函数。

3.2 新版功能.

在 3.6 版更改: 增加对实现了*os.PathLike* 接口的对象的支持。

os.**fsdecode**(*filename*)
从文件系统编码方式解码为路径类 文件名，使用 'surrogateescape' 代理转义编码错误处理器，在 Windows 系统上会使用 'strict'；返回*str* 字符串不变。

fsencode() 是此函数的逆向函数。

3.2 新版功能.

在 3.6 版更改: 增加对实现了*os.PathLike* 接口的对象的支持。

os.**fspath**(*path*)
返回路径的文件系统表示。

如果传入的是*str* 或*bytes* 类型的字符串，将原样返回。否则*__fspath__()* 将被调用，如果得到的是一个*str* 或*bytes* 类型的对象，那就返回这个值。其他所有情况则会抛出*TypeError* 异常。

3.6 新版功能.

class os.**PathLike**
描述表示一个文件系统路径的*abstract base class*，如*pathlib.PurePath* 。

3.6 新版功能.

abstractmethod **__fspath__**()
返回当前对象的文件系统表示。

这个方法只应该返回一个*str* 字符串或*bytes* 字节串，请优先选择*str* 字符串。

os.**getenv**(*key*, *default=None*)
如果存在，返回环境变量 *key* 的值，否则返回 *default*。*key*，*default* 和返回值均为 str 字符串类型。

在 Unix 系统上，键和值会使用*sys.getfilesystemencoding()* 和“’ surrogateescape’“ 错误处理进行解码。如果你想使用其他的编码，使用*os.getenvb()*。

Availability: most flavors of Unix, Windows.

os.**getenvb**(*key*, *default=None*)
如果存在环境变量 *key* 那么返回其值，否则返回 *default*。*key*，*default* 和返回值均为 bytes 字节串类型。

getenvb() is only available if *supports_bytes_environ* is True.

Availability: most flavors of Unix.

3.2 新版功能.

os.**get_exec_path**(*env=None*)
返回将用于搜索可执行文件的目录列表，与在外壳程序中启动一个进程时相似。指定的 *env* 应为用于搜索 PATH 的环境变量字典。默认情况下，当 *env* 为 None 时，将会使用*environ* 。

3.2 新版功能.

os.**getegid**()
返回当前进程的有效组 ID。对应当前进程执行文件的“set id”位。

Availability: Unix.

os.**geteuid**()
返回当前进程的有效用户 ID。

Availability: Unix.

os.**getgid**()
返回当前进程的实际组 ID。

Availability: Unix.

os.**getgrouplist**(*user*, *group*)
返回该用户所在的组 ID 列表。可能 *group* 参数没有在返回的列表中，实际上用户应该也是属于该 *group*。*group* 参数一般可以从储存账户信息的密码记录文件中找到。

Availability: Unix.

3.3 新版功能.

os.**getgroups**()
返回当前进程关联的附加组 ID 列表

Availability: Unix.

注解： 在 Mac OS X 系统中，*getgroups()* 会和其他 Unix 平台有些不同。如果 Python 解释器是在 10.5 或更早版本中部署，*getgroups()* 返回当前用户进程相关的有效组 ID 列表。该列表长度由于系统预设的接口限制，最长为 16。而且在适当的权限下，返回结果还会因*getgroups()* 而发生变化；如果 Python 解释器是在 10.5 以上版本中部署，*getgroups()* 返回进程所属有效用户 ID 所对应的用户的组 ID 列表，组用户列表可能因为进程的生存周期而发生变动，而且也不会因为*setgroups()* 的调用而发生，返回的组用户列表长度也没有长度 16 的限制。在部署中，Python 解释器用到的变量 MACOSX_DEPLOYMENT_TARGET 可以用*sysconfig.get_config_var()*。

---

os.**getlogin**()
返回通过控制终端进程进行登录的用户名。在多数情况下，使用*getpass.getuser()* 会更有效，因为后者会通过检查环境变量 LOGNAME 或 USERNAME 来查找用户，再由 pwd.getpwuid(os.getuid())[0] 来获取当前用户 ID 的登录名。

Availability: Unix, Windows.

os.**getpgid**(*pid*)
根据进程 id *pid* 返回进程的组 ID 列表。如果 *pid* 为 0，则返回当前进程的进程组 ID 列表

Availability: Unix.

os.**getpgrp**()
返回当时进程组的 ID

Availability: Unix.

os.**getpid**()
返回当前进程 ID

os.**getppid**()
返回父进程 ID。当父进程已经结束，在 Unix 中返回的 ID 是初始进程 (1) 中的一个，在 Windows 中仍然是同一个进程 ID，该进程 ID 有可能已经被进行进程所占用。

Availability: Unix, Windows.

在 3.2 版更改: 添加 WIndows 的支持。

os.**getpriority**(*which*, *who*)
获取程序调度优先级。*which* 参数值可以是*PRIO_PROCESS*，*PRIO_PGRP*，或*PRIO_USER* 中的一个，*who* 是相对于 *which* (*PRIO_PROCESS* 的进程标识符，*PRIO_PGRP* 的进程组标识符和*PRIO_USER* 的用户 ID)。当 *who* 为 0 时（分别）表示调用的进程，调用进程的进程组或调用进程所属的真实用户 ID。

Availability: Unix.

3.3 新版功能.

os.**PRIO_PROCESS**
os.**PRIO_PGRP**
os.**PRIO_USER**
函数*getpriority()* 和*setpriority()* 的参数。

Availability: Unix.

3.3 新版功能.

os.**getresuid**()
返回一个由 (ruid, euid, suid) 所组成的元组，分别表示当前进程的真实用户 ID，有效用户 ID 和暂存用户 ID。

Availability: Unix.

3.2 新版功能.

os.**getresgid**()
返回一个由 (rgid, egid, sgid) 所组成的元组，分别表示当前进程的真实组 ID，有效组 ID 和暂存组 ID。

Availability: Unix.

3.2 新版功能.

os.**getuid**()
返回当前进程的真实用户 ID。

Availability: Unix.

os.**initgroups**(*username*, *gid*)
调用系统 initgroups()，使用指定用户所在的所有值来初始化组访问列表，包括指定的组 ID。

Availability: Unix.

3.2 新版功能.

os.**putenv**(*key*, *value*)
将名为 *key* 的环境变量值设置为 *value*。该变量名修改会影响由*os.system()*，*popen()* ，*fork()* 和*execv()* 发起的子进程。

Availability: most flavors of Unix, Windows.

注解： 在一些平台，包括 FreeBSD 和 Mac OS X，设置 `environ` 可能导致内存泄露。详情参考 putenv 相关系统文档。

当系统支持*putenv()* 时，`os.environ` 中的参数赋值会自动转换为对*putenv()* 的调用。不过*putenv()* 的调用不会更新 `os.environ`，因此最好使用 `os.environ` 对变量赋值。

`os.`**`setegid`**(*egid*)
设置当前进程的有效组 ID。

Availability: Unix.

`os.`**`seteuid`**(*euid*)
设置当前进程的有效用户 ID。

Availability: Unix.

`os.`**`setgid`**(*gid*)
设置当前进程的组 ID。

Availability: Unix.

`os.`**`setgroups`**(*groups*)
将 *group* 参数值设置为与当进程相关联的附加组 ID 列表。*group* 参数必须为一个序列，每个元素应为每个组的数字 ID。该操作通常只适用于超级用户。

Availability: Unix.

注解： 在 Mac OS X 中，*groups* 的长度不能超过系统定义的最大有效组 ID 个数，一般为 16。如果它没有返回与调用 setgroups() 所设置的相同的组列表，请参阅*getgroups()* 的文档。

`os.`**`setpgrp`**()
根据已实现的版本（如果有）来调用系统 `setpgrp()` 或 `setpgrp(0, 0)`。相关说明，请参考 Unix 手册。

Availability: Unix.

`os.`**`setpgid`**(*pid*, *pgrp*)
使用系统调用 `setpgid()`，将 *pid* 对应进程的组 ID 设置为 *pgrp*。相关说明，请参考 Unix 手册。

Availability: Unix.

`os.`**`setpriority`**(*which*, *who*, *priority*)
设置程序调度优先级。*which* 的值为*PRIO_PROCESS*, *PRIO_PGRP* 或*PRIO_USER* 之一，而 *who* 会相对于 *which* (*PRIO_PROCESS* 的进程标识符, *PRIO_PGRP* 的进程组标识符和*PRIO_USER* 的用户 ID) 被解析。*who* 值为零 (分别) 表示调用进程，调用进程的进程组或调用进程的真实用户 ID。*priority* 是范围在 -20 至 19 的值。默认优先级为 0；较小的优先级数值会更优先被调度。

Availability: Unix

3.3 新版功能.

`os.`**`setregid`**(*rgid*, *egid*)
设置当前进程的真实和有效组 ID。

Availability: Unix.

`os.`**`setresgid`**(*rgid*, *egid*, *sgid*)
设置当前进程的真实，有效和暂存组 ID。

Availability: Unix.

3.2 新版功能.

os.**setresuid**(*ruid*, *euid*, *suid*)
设置当前进程的真实，有效和暂存用户 ID。

Availability: Unix.

3.2 新版功能.

os.**setreuid**(*ruid*, *euid*)
设置当前进程的真实和有效用户 ID。

Availability: Unix.

os.**getsid**(*pid*)
调用系统调用 `getsid()`。相关语义请参阅 Unix 手册。

Availability: Unix.

os.**setsid**()
使用系统调用 `getsid()`。相关说明，请参考 Unix 手册。

Availability: Unix.

os.**setuid**(*uid*)
设置当前进程的用户 ID。

Availability: Unix.

os.**strerror**(*code*)
根据 *code* 中的错误码返回错误消息。在某些平台上当给出未知错误码时 `strerror()` 将返回 `NULL` 并会引发*ValueError*。

os.**supports_bytes_environ**
如果操作系统上原生环境类型是字节型则为 `True` (例如在 Windows 上为 `False`)。

3.2 新版功能.

os.**umask**(*mask*)
设定当前数值掩码并返回之前的掩码。

os.**uname**()
返回当前操作系统的识别信息。返回值是一个有 5 个属性的对象：

- `sysname` - 操作系统名
- `nodename` - 机器在网络上的名称（需要先设定）
- `release` - 操作系统发行信息
- `version` - 操作系统版本信息
- `machine` - 硬件标识符

为了向后兼容，该对象也是可迭代的，像是一个按照 `sysname`，`nodename`，`release`，`version`，和 `machine` 顺序组成的元组。

有些系统会将 `nodename` 截短为 8 个字符或截短至前缀部分；获取主机名的一个更好方式是*socket.gethostname()* 或甚至可以用 `socket.gethostbyaddr(socket.gethostname())`。

Availability: recent flavors of Unix.

在 3.3 版更改: 返回结果的类型由元组变成一个类似元组的对象，同时具有命名的属性。

os.**unsetenv**(*key*)
取消设置（删除）名为 *key* 的环境变量。变量名的改变会影响由*os.system()*，*popen()*，*fork()* 和*execv()* 触发的子进程。

当系统支持*unsetenv()*，删除在 os.environ 中的变量会自动转换为对*unsetenv()* 的调用。但是*unsetenv()* 不能更新 os.environ，因此最好直接删除 os.environ 中的变量。

Availability: most flavors of Unix, Windows.

16.1.3 创建文件对象

该函数创建新的文件对象。(另见*open()* 关于打开文件的说明。)

os.**fdopen**(*fd*, **args*, ***kwargs*)
返回打开文件描述符 *fd* 对应文件的对象。类似内建*open()* 函数，二者接受同样的参数。不同之处在于*fdopen()* 第一个参数应该为整数。

16.1.4 文件描述符操作

这些函数对文件描述符所引用的 I/O 流进行操作。

文件描述符是一些小的整数，对应于当前进程所打开的文件。例如，标准输入的文件描述符通常是 0，标准输出是 1，标准错误是 2。之后被进程打开的文件的文件描述符会被依次指定为 3，4，5 等。“文件描述符”这个词有点误导性，在 Unix 平台中套接字和管道也被文件描述符所引用。

当需要时，可以用*fileno()* 可以获得*file object* 所对应的文件描述符。需要注意的是，直接使用文件描述符会绕过文件对象的方法，会忽略如数据内部缓冲等情况。

os.**close**(*fd*)
关闭文件描述符 *fd*。

注解: 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。关闭由内建函数*open()*，*popen()* 或*fdopen()* 返回的“文件对象”，则使用其相应的*close()* 方法。

---

os.**closerange**(*fd_low*, *fd_high*)
关闭从 *fd_low* （包括）到 *fd_high* （排除）间的文件描述符，并忽略错误。类似（但快于）：

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.**device_encoding**(*fd*)
如果连接到终端，则返回一个与 *fd* 关联的设备描述字符，否则返回*None*。

os.**dup**(*fd*)
返回一个文件描述符 *fd* 的副本。该文件描述符的副本是不可继承的。

在 Windows 中，当复制一个标准流（0: stdin, 1: stdout, 2: stderr）时，新的文件描述符是可继承的。

在 3.4 版更改: 新的文件描述符现在是不可继承的。

os.**dup2**(*fd*, *fd2*, *inheritable=True*)
Duplicate file descriptor *fd* to *fd2*, closing the latter first if necessary. The file descriptor *fd2* is *inheritable* by default, or non-inheritable if *inheritable* is False.

在 3.4 版更改: 添加可选参数 *inheritable*。

os.**fchmod**(*fd*, *mode*)
将 *fd* 指定文件的权限状态修改为 *mode*。可以参考*chmod()* 中列出 *mode* 的可用值。从 Python 3.3 开始，这相当于 os.chmod(fd, mode)。

Availability: Unix.

os.**fchown**(*fd*, *uid*, *gid*)
分别将 *fd* 指定文件的所有者和组 ID 修改为 *uid* 和 *gid* 的值。若不想变更其中的某个 ID，可将相应值设为 -1。参考*chown()*。从 Python 3.3 开始，这相当于 os.chown(fd, uid, gid)。

Availability: Unix.

os.**fdatasync**(*fd*)
强制将文件描述符 *fd* 指定文件写入磁盘。不强制更新元数据。

Availability: Unix.

注解： 该功能在 MacOS 中不可用。

---

os.**fpathconf**(*fd*, *name*)
返回与打开的文件有关的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX.1，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 pathconf_names 字典中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 pathconf_names，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

从 Python 3.3 起，此功能等价于 os.pathconf(fd, name)。

Availability: Unix.

os.**fstat**(*fd*)
获取文件描述符 *fd* 的状态. 返回一个*stat_result* 对象。

从 Python 3.3 起，此功能等价于 os.stat(fd)。

参见:

stat() 函数。

os.**fstatvfs**(*fd*)
返回文件系统的信息，该文件系统是文件描述符 *fd* 指向的文件所在的文件系统，与*statvfs()* 一样。从 Python 3.3 开始，它等效于 os.statvfs(fd)。

Availability: Unix.

os.**fsync**(*fd*)
强制将文件描述符 *fd* 指向的文件写入磁盘。在 Unix，这将调用原生 fsync() 函数；在 Windows，则是 MS _commit() 函数。

如果要写入的是缓冲区内的 Python 文件对象 *f*，请先执行 f.flush()，然后执行 os.fsync(f.fileno())，以确保与 *f* 关联的所有内部缓冲区都写入磁盘。

Availability: Unix, Windows.

os.**ftruncate**(*fd*, *length*)
将文件描述符 *fd* 对应的文件切分开，以使其最大为 *length* 字节。从 Python 3.3 开始，它等效于 os.truncate(fd, length)。

Availability: Unix, Windows.

在 3.5 版更改: 添加了 Windows 支持

os.**get_blocking**(*fd*)
获取文件描述符的阻塞模式：如果设置了*O_NONBLOCK* 标志位，返回 False，如果该标志位被清除，返回 True。

参见*set_blocking()* 和*socket.socket.setblocking()*。

Availability: Unix.

3.5 新版功能.

os.**isatty**(*fd*)
如果文件描述符 *fd* 打开且已连接至 tty 设备（或类 tty 设备），返回 True，否则返回 False。

os.**lockf**(*fd*, *cmd*, *len*)
在打开的文件描述符上，使用、测试或删除 POSIX 锁。*fd* 是一个打开的文件描述符。*cmd* 指定要进行的操作，它们是*F_LOCK*、*F_TLOCK*、*F_ULOCK* 或*F_TEST* 中的一个。*len* 指定哪部分文件需要锁定。

Availability: Unix.

3.3 新版功能.

os.**F_LOCK**
os.**F_TLOCK**
os.**F_ULOCK**
os.**F_TEST**
标志位，用于指定*lockf()* 进行哪一种操作。

Availability: Unix.

3.3 新版功能.

os.**lseek**(*fd*, *pos*, *how*)
将文件描述符 *fd* 的当前位置设置为 *pos*，位置的计算方式 *how* 如下：设置为*SEEK_SET* 或 0 表示从文件开头计算，设置为*SEEK_CUR* 或 1 表示从文件当前位置计算，设置为*SEEK_END* 或 2 表示文件末尾计算。返回新指针位置，这个位置是从文件开头计算的，单位是字节。

os.**SEEK_SET**
os.**SEEK_CUR**
os.**SEEK_END**
lseek() 函数的参数，它们的值分别为 0、1 和 2。

3.3 新版功能: 某些操作系统可能支持其他值，例如 os.SEEK_HOLE 或 os.SEEK_DATA。

os.**open**(*path*, *flags*, *mode=0o777*, *, *dir_fd=None*)
打开文件 *path*，根据 *flags* 设置各种标志位，并根据 *mode* 设置其权限模式。当计算 *mode* 时，会首先根据当前 umask 值将部分权限去除。本方法返回新文件的描述符。新的文件描述符是不可继承 的。

有关 flag 和 mode 取值的说明，请参见 C 运行时文档。标志位常量（如*O_RDONLY* 和*O_WRONLY*）在*os* 模块中定义。特别地，在 Windows 上需要添加*O_BINARY* 才能以二进制模式打开文件。

本函数带有 *dir_fd* 参数，支持基于目录描述符的相对路径。

在 3.4 版更改: 新的文件描述符现在是不可继承的。

注解: 本函数适用于底层的 I/O。常规用途请使用内置函数*open()*，该函数的 read() 和 write() 方法（及其他方法）会返回文件对象。要将文件描述符包装在文件对象中，请使用*fdopen()*。

3.3 新版功能: *dir_fd* 参数。

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

在 3.6 版更改: 接受一个*path-like object*。

以下常量是 *open()* 函数 *flags* 参数的选项。可以用按位或运算符 | 将它们组合使用。部分常量并非在所有平台上都可用。有关其可用性和用法的说明，请参阅 *open(2)* 手册（Unix 上）或 MSDN （Windows 上）。

os.O_RDONLY
os.O_WRONLY
os.O_RDWR
os.O_APPEND
os.O_CREAT
os.O_EXCL
os.O_TRUNC
上述常量在 Unix 和 Windows 上均可用。

os.O_DSYNC
os.O_RSYNC
os.O_SYNC
os.O_NDELAY
os.O_NONBLOCK
os.O_NOCTTY
os.O_CLOEXEC
这个常数仅在 Unix 系统中可用。

在 3.3 版更改: 增加 *O_CLOEXEC* 常量。

os.O_BINARY
os.O_NOINHERIT
os.O_SHORT_LIVED
os.O_TEMPORARY
os.O_RANDOM
os.O_SEQUENTIAL
os.O_TEXT
这个常数仅在 Windows 系统中可用。

os.O_ASYNC
os.O_DIRECT
os.O_DIRECTORY
os.O_NOFOLLOW
os.O_NOATIME
os.O_PATH
os.O_TMPFILE
os.O_SHLOCK
os.O_EXLOCK
上述常量是扩展常量，如果 C 库未定义它们，则不存在。

在 3.4 版更改: 在支持的系统上增加 *O_PATH*。增加 *O_TMPFILE*，仅在 Linux Kernel 3.11 或更高版本可用。

os.openpty()
打开一对新的伪终端，返回一对文件描述符 “（主，从）“，分别为 pty 和 tty。新的文件描述符是*不可继承* 的。对于（稍微）轻量一些的方法，请使用 *pty* 模块。

Availability: some flavors of Unix.

在 3.4 版更改: 新的文件描述符不再可继承。

os.**pipe**()
创建一个管道，返回一对分别用于读取和写入的文件描述符 (r, w)。新的文件描述符是不可继承 的。

Availability: Unix, Windows.

在 3.4 版更改: 新的文件描述符不再可继承。

os.**pipe2**(*flags*)
创建带有 *flags* 标志位的管道。可通过对以下一个或多个值进行“或”运算来构造这些 *flags*: *O_NONBLOCK*、*O_CLOEXEC*。返回一对分别用于读取和写入的文件描述符 (r, w)。

Availability: some flavors of Unix.

3.3 新版功能.

os.**posix_fallocate**(*fd*, *offset*, *len*)
确保为 *fd* 指向的文件分配了足够的磁盘空间，该空间从偏移量 *offset* 开始，到 *len* 字节为止。

Availability: Unix.

3.3 新版功能.

os.**posix_fadvise**(*fd*, *offset*, *len*, *advice*)
声明即将以特定模式访问数据，使内核可以提前进行优化。数据范围是从 *fd* 所指向文件的 *offset* 开始，持续 *len* 个字节。*advice* 的取值是如下之一: *POSIX_FADV_NORMAL*, *POSIX_FADV_SEQUENTIAL*, *POSIX_FADV_RANDOM*, *POSIX_FADV_NOREUSE*, *POSIX_FADV_WILLNEED* 或*POSIX_FADV_DONTNEED*。

Availability: Unix.

3.3 新版功能.

os.**POSIX_FADV_NORMAL**
os.**POSIX_FADV_SEQUENTIAL**
os.**POSIX_FADV_RANDOM**
os.**POSIX_FADV_NOREUSE**
os.**POSIX_FADV_WILLNEED**
os.**POSIX_FADV_DONTNEED**
用于*posix_fadvise()* 的 *advice* 参数的标志位，指定可能使用的访问模式。

Availability: Unix.

3.3 新版功能.

os.**pread**(*fd*, *buffersize*, *offset*)
Read from a file descriptor, *fd*, at a position of *offset*. It will read up to *buffersize* number of bytes. The file offset remains unchanged.

Availability: Unix.

3.3 新版功能.

os.**pwrite**(*fd*, *str*, *offset*)
Write *bytestring* to a file descriptor, *fd*, from *offset*, leaving the file offset unchanged.

Availability: Unix.

3.3 新版功能.

os.**read**(*fd*, *n*)
Read at most *n* bytes from file descriptor *fd*. Return a bytestring containing the bytes read. If the end of the file referred to by *fd* has been reached, an empty bytes object is returned.

注解： 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。读取由内建函数*open()*、*popen()*、*fdopen()* 或*sys.stdin* 返回的“文件对象”，则使用其相应的 `read()` 或 `readline()` 方法。

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

`os.`**`sendfile`**(*out*, *in*, *offset*, *count*)
`os.`**`sendfile`**(*out*, *in*, *offset*, *count*[, *headers*][, *trailers*], *flags=0*)
将文件描述符 *in* 中的 *count* 字节复制到文件描述符 *out* 的偏移位置 *offset* 处。返回复制的字节数，如果到达 EOF，返回 0。

定义了*sendfile()* 的所有平台均支持第一种函数用法。

在 Linux 上，将 *offset* 设置为 `None`，则从 *in* 的当前位置开始读取，并更新 *in* 的位置。

第二种函数用法可以在 Mac OS X 和 FreeBSD 上使用，其中，*headers* 和 *trailers* 是任意的缓冲区序列，它们分别在写入 *in* 的数据前、后被写入。返回值与第一种用法相同。

在 Mac OS X 和 FreeBSD 上，将 *count* 设为 0 表示持续复制直到 *in* 的结尾。

所有平台都支持将套接字作为 *out* 文件描述符，有些平台也支持其他类型（如常规文件或管道）。

跨平台应用程序不应使用 *headers*、*trailers* 和 *flags* 参数。

Availability: Unix.

注解： 有关*sendfile()* 的高级封装，参见*socket.socket.sendfile()*。

3.3 新版功能.

`os.`**`set_blocking`**(*fd*, *blocking*)
设置指定文件描述符的阻塞模式：如果 blocking 为 `False`，则为该描述符设置*O_NONBLOCK* 标志位，反之则清除该标志位。

参见*get_blocking()* 和*socket.socket.setblocking()*。

Availability: Unix.

3.5 新版功能.

`os.`**`SF_NODISKIO`**
`os.`**`SF_MNOWAIT`**
`os.`**`SF_SYNC`**
sendfile() 函数的参数（假设当前实现支持这些参数）。

Availability: Unix.

3.3 新版功能.

`os.`**`readv`**(*fd*, *buffers*)
Read from a file descriptor *fd* into a number of mutable *bytes-like objects* *buffers*. *readv()* will transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data. *readv()* returns the total number of bytes read (which may be less than the total capacity of all the objects).

Availability: Unix.

3.3 新版功能.

os.**tcgetpgrp**(*fd*)
返回与 *fd* 给定的终端相关联的进程组（*fd* 是由*os.open()* 返回的已打开文件的描述符）。

Availability: Unix.

os.**tcsetpgrp**(*fd*, *pg*)
设置与 *fd* 指定的终端相关联的进程组为 *pg*（*fd* 是由*os.open()* 返回的已打开的文件描述符）。

Availability: Unix.

os.**ttyname**(*fd*)
返回一个字符串，该字符串表示与文件描述符 *fd* 关联的终端。如果 *fd* 没有与终端关联，则抛出异常。

Availability: Unix.

os.**write**(*fd*, *str*)
Write the bytestring in *str* to file descriptor *fd*. Return the number of bytes actually written.

注解： 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。若要写入由内建函数*open()*、*popen()*、*fdopen()*、*sys.stdout* 或*sys.stderr* 返回的“文件对象”，则应使用其相应的 `write()` 方法。

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

os.**writev**(*fd*, *buffers*)
Write the contents of *buffers* to file descriptor *fd*. *buffers* must be a sequence of *bytes-like objects*. Buffers are processed in array order. Entire contents of first buffer is written before proceeding to second, and so on. The operating system may set a limit (sysconf() value SC_IOV_MAX) on the number of buffers that can be used.

writev() writes the contents of each object to the file descriptor and returns the total number of bytes written.

Availability: Unix.

3.3 新版功能.

查询终端的尺寸

3.3 新版功能.

os.**get_terminal_size**(*fd=STDOUT_FILENO*)
返回终端窗口的尺寸，格式为“（列，行）“，它是类型为*terminal_size* 的元组。

可选参数 `fd``（默认为 ``STDOUT_FILENO` 或标准输出）指定应查询的文件描述符。

如果文件描述符未连接到终端，则抛出*OSError* 异常。

shutil.get_terminal_size() 是供常规使用的高阶函数，`os.get_terminal_size` 是其底层的实现。

Availability: Unix, Windows.

class os.**terminal_size**
元组的子类，存储终端窗口尺寸 `(columns, lines)`。

columns
终端窗口的宽度，单位为字符。

lines
终端窗口的高度，单位为字符。

文件描述符的继承

3.4 新版功能.

每个文件描述符都有一个“inheritable”（可继承）标志位，该标志位控制了文件描述符是否可以由子进程继承。从 Python 3.4 开始，由 Python 创建的文件描述符默认是不可继承的。

在 UNIX 上，执行新程序时，子进程中不可继承的文件描述符会被关闭，其他文件描述符将被继承。

在 Windows 上，不可继承的句柄和文件描述符在子进程中是关闭的，但标准流（文件描述符 0、1 和 2 即标准输入、标准输出和标准错误）是始终继承的。如果使用 *spawn** 函数，所有可继承的句柄和文件描述符都将被继承。如果使用 *subprocess* 模块，将关闭除标准流以外的所有文件描述符，并且仅当 *close_fds* 参数为 `False` 时才继承可继承的句柄。

`os.`**`get_inheritable`**(*fd*)
获取指定文件描述符的“可继承”标志位（为布尔值）。

`os.`**`set_inheritable`**(*fd*, *inheritable*)
设置指定文件描述符的“可继承”标志位。

`os.`**`get_handle_inheritable`**(*handle*)
获取指定句柄的“可继承”标志位（为布尔值）。

Availability: Windows.

`os.`**`set_handle_inheritable`**(*handle*, *inheritable*)
设置指定句柄的“可继承”标志位。

Availability: Windows.

16.1.5 文件和目录

在某些 Unix 平台上，许多函数支持以下一项或多项功能：

- **specifying a file descriptor:** For some functions, the *path* argument can be not only a string giving a path name, but also a file descriptor. The function will then operate on the file referred to by the descriptor. (For POSIX systems, Python will call the `f...` version of the function.)

  You can check whether or not *path* can be specified as a file descriptor on your platform using *os.supports_fd*. If it is unavailable, using it will raise a *NotImplementedError*.

  If the function also supports *dir_fd* or *follow_symlinks* arguments, it is an error to specify one of those when supplying *path* as a file descriptor.

- **paths relative to directory descriptors:** If *dir_fd* is not `None`, it should be a file descriptor referring to a directory, and the path to operate on should be relative; path will then be relative to that directory. If the path is absolute, *dir_fd* is ignored. (For POSIX systems, Python will call the `...at` or `f...at` version of the function.)

  You can check whether or not *dir_fd* is supported on your platform using *os.supports_dir_fd*. If it is unavailable, using it will raise a *NotImplementedError*.

- **not following symlinks:** If *follow_symlinks* is `False`, and the last element of the path to operate on is a symbolic link, the function will operate on the symbolic link itself instead of the file the link points to. (For POSIX systems, Python will call the `l...` version of the function.)

  You can check whether or not *follow_symlinks* is supported on your platform using *os.supports_follow_symlinks*. If it is unavailable, using it will raise a *NotImplementedError*.

`os.`**`access`**(*path*, *mode*, *, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)
使用真实的 uid/gid 测试对 *path* 的访问。请注意，大多数测试操作将使用有效的 uid/gid，因此可以在 suid/sgid 环境中运用此例程，来测试调用用户是否具有对 *path* 的指定访问权限。*mode* 为 *F_OK* 时用于

测试 *path* 是否存在，也可以对*R_OK*、*W_OK* 和*X_OK* 中的一个或多个进行“或”运算来测试指定权限。允许访问则返回*True*，否则返回*False*。更多信息请参见 Unix 手册页 *access(2)*。

本函数支持指定基于目录描述符的相对路径 和不跟踪符号链接。

如果 *effective_ids* 为 `True`，*access()* 将使用有效用户 ID/用户组 ID 而非实际用户 ID/用户组 ID 进行访问检查。您的平台可能不支持 *effective_ids*，您可以使用*os.supports_effective_ids* 检查它是否可用。如果不可用，使用它时会抛出*NotImplementedError* 异常。

注解： 使用*access()* 来检查用户是否具有某项权限（如打开文件的权限），然后再使用*open()* 打开文件，这样做存在一个安全漏洞，因为用户可能会在检查和打开文件之间的时间里做其他操作。推荐使用*EAFP* 技术。如:

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

最好写成:

```
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    with fp:
        return fp.read()
```

注解： 即使*access()* 指示 I/O 操作会成功，但实际操作仍可能失败，尤其是对网络文件系统的操作，其权限语义可能超出常规的 POSIX 权限位模型。

在 3.3 版更改: 添加 *dir_fd*、*effective_ids* 和 *follow_symlinks* 参数。

在 3.6 版更改: 接受一个*path-like object*。

os.**F_OK**
os.**R_OK**
os.**W_OK**
os.**X_OK**
作为*access()* 的 *mode* 参数的可选值，分别测试 *path* 的存在性、可读性、可写性和可执行性。

os.**chdir**(*path*)
将当前工作目录更改为 *path*。

本函数支持指定文件描述符为参数。其中，描述符必须指向打开的目录，不能是打开的文件。

3.3 新版功能: 在某些平台上新增支持将 *path* 参数指定为文件描述符。

在 3.6 版更改: 接受一个*path-like object*。

os.**chflags**(*path*, *flags*, *, *follow_symlinks=True*)
将 *path* 标志设置为数字 标志。标志可以用以下值按位或组合起来（以下值在*stat* 模块中定义）:

- *stat.UF_NODUMP*
- *stat.UF_IMMUTABLE*
- *stat.UF_APPEND*

- *stat.UF_OPAQUE*
- *stat.UF_NOUNLINK*
- *stat.UF_COMPRESSED*
- *stat.UF_HIDDEN*
- *stat.SF_ARCHIVED*
- *stat.SF_IMMUTABLE*
- *stat.SF_APPEND*
- *stat.SF_NOUNLINK*
- *stat.SF_SNAPSHOT*

本函数支持不跟踪符号链接。

Availability: Unix.

3.3 新版功能: *follow_symlinks* 参数。

在 3.6 版更改: 接受一个*path-like object*。

os.**chmod**(*path*, *mode*, *, *dir_fd=None*, *follow_symlinks=True*)

将 *path* 的 mode 更改为其他由数字表示的 *mode*。*mode* 可以用以下值之一，也可以将它们按位或组合起来（以下值在*stat* 模块中定义）：

- *stat.S_ISUID*
- *stat.S_ISGID*
- *stat.S_ENFMT*
- *stat.S_ISVTX*
- *stat.S_IREAD*
- *stat.S_IWRITE*
- *stat.S_IEXEC*
- *stat.S_IRWXU*
- *stat.S_IRUSR*
- *stat.S_IWUSR*
- *stat.S_IXUSR*
- *stat.S_IRWXG*
- *stat.S_IRGRP*
- *stat.S_IWGRP*
- *stat.S_IXGRP*
- *stat.S_IRWXO*
- *stat.S_IROTH*
- *stat.S_IWOTH*
- *stat.S_IXOTH*

本函数支持指定指定文件描述符为参数、基于目录描述符的相对路径 和不跟踪符号链接。

注解: 尽管 Windows 支持*chmod()*，但只能用它设置文件的只读标志（`stat.S_IWRITE` 和 `stat.S_IREAD` 常量或对应的整数值）。所有其他标志位都会被忽略。

3.3 新版功能: 添加了指定 *path* 为文件描述符的支持，以及 *dir_fd* 和 *follow_symlinks* 参数。

在 3.6 版更改: 接受一个*path-like object*。

`os.`**`chown`**(*path*, *uid*, *gid*, *, *dir_fd=None*, *follow_symlinks=True*)
将 *path* 的用户和组 ID 分别修改为数字形式的 *uid* 和 *gid*。若要使其中某个 ID 保持不变，请将其置为 -1。

本函数支持指定指定文件描述符为参数、基于目录描述符的相对路径 和不跟踪符号链接。

参见更高阶的函数*shutil.chown()*，除了数字 ID 之外，它也接受名称。

Availability: Unix.

3.3 新版功能: Added support for specifying an open file descriptor for *path*, and the *dir_fd* and *follow_symlinks* arguments.

在 3.6 版更改: 支持类路径对象。

`os.`**`chroot`**(*path*)
将当前进程的根目录更改为 *path*。

Availability: Unix.

在 3.6 版更改: 接受一个*path-like object*。

`os.`**`fchdir`**(*fd*)
将当前工作目录更改为文件描述符 *fd* 指向的目录。fd 必须指向打开的目录而非文件。从 Python 3.3 开始，它等效于 `os.chdir(fd)`。

Availability: Unix.

`os.`**`getcwd`**()
返回表示当前工作目录的字符串。

`os.`**`getcwdb`**()
返回表示当前工作目录的字节串 (bytestring)。

`os.`**`lchflags`**(*path*, *flags*)
将 *path* 的 flags 设置为其他由数字表示的 *flags*，与*chflags()* 类似，但不跟踪符号链接。从 Python 3.3 开始，它等效于 `os.chflags(path, flags, follow_symlinks=False)`。

Availability: Unix.

在 3.6 版更改: 接受一个*path-like object*。

`os.`**`lchmod`**(*path*, *mode*)
将 *path* 的权限状态修改为 *mode*。如果 path 是符号链接，则影响符号链接本身而非链接目标。可以参考*chmod()* 中列出 *mode* 的可用值。从 Python 3.3 开始，这相当于 `os.chmod(path, mode, follow_symlinks=False)`。

Availability: Unix.

在 3.6 版更改: 接受一个*path-like object*。

`os.`**`lchown`**(*path*, *uid*, *gid*)
将 *path* 的用户和组 ID 分别修改为数字形式的 *uid* 和 *gid*，本函数不跟踪符号链接。从 Python 3.3 开始，它等效于 `os.chown(path, uid, gid, follow_symlinks=False)`。

Availability: Unix.

在 3.6 版更改: 接受一个*path-like object*。

os.**link**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*, *follow_symlinks=True*)
创建一个指向 *src* 的硬链接，名为 *dst*。

本函数支持将 *src_dir_fd* 和 *dst_dir_fd* 中的一个或两个指定为基于目录描述符的相对路径，支持不跟踪符号链接。

Availability: Unix, Windows.

在 3.2 版更改: 添加了 Windows 支持

3.3 新版功能: 添加 *src_dir_fd*、*dst_dir_fd* 和 *follow_symlinks* 参数。

在 3.6 版更改: 接受一个类路径对象 作为 *src* 和 *dst*。

os.**listdir**(*path='.'*)
返回一个列表，该列表包含了 *path* 中所有文件与目录的名称。该列表按任意顺序排列，并且不包含特殊条目 `'.'` 和 `'..'`，即使它们确实在目录中存在。

path 可以是路径类对象。如果 *path* 是（直接/间接由`PathLike`接口返回的）`bytes` 类型，则返回的文件名也将是 `bytes` 类型，其他情况下文件名类型是 `str`。

本函数也支持指定文件描述符为参数，其中描述符必须指向目录。

注解: 要将 `str` 类型的文件名编码为 `bytes`，请使用`fsencode()`。

参见:

`scandir()` 函数返回目录内文件名的同时，也返回文件属性信息，它在某些具体情况下能提供更好的性能。

在 3.2 版更改: *path* 变为可选参数。

3.3 新版功能: Added support for specifying an open file descriptor for *path*.

在 3.6 版更改: 接受一个*path-like object*。

os.**lstat**(*path*, *, *dir_fd=None*)
在给定路径上执行本函数，其操作相当于 `lstat()` 系统调用，类似于`stat()` 但不跟踪符号链接。返回值是`stat_result` 对象。

在不支持符号链接的平台上，本函数是`stat()` 的别名。

从 Python 3.3 起，此功能等价于 `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`。

本函数支持基于目录描述符的相对路径。

参见:

`stat()` 函数。

在 3.2 版更改: 添加对 Windows 6.0 (Vista) 符号链接的支持。

在 3.3 版更改: 添加了 *dir_fd* 参数。

在 3.6 版更改: 接受一个类路径对象 作为 *src* 和 *dst*。

os.**mkdir**(*path*, *mode=0o777*, *, *dir_fd=None*)
创建一个名为 *path* 的目录，应用以数字表示的权限模式 *mode*。

如果目录已存在，则抛出`FileExistsError` 异常。

某些系统会忽略 *mode*。如果没有忽略它，那么将首先从它中减去当前的 umask 值。如果除最后 9 位（即 *mode* 八进制的最后 3 位）之外，还设置了其他位，则其他位的含义取决于各个平台。在某些平台上，它们会被忽略，应显式调用 `chmod()` 进行设置。

本函数支持基于目录描述符的相对路径。

如果需要创建临时目录，请参阅 `tempfile` 模块中的 `tempfile.mkdtemp()` 函数。

3.3 新版功能: *dir_fd* 参数。

在 3.6 版更改: 接受一个 *path-like object*。

os.**makedirs**(*name*, *mode=0o777*, *exist_ok=False*)

递归目录创建函数。与 `mkdir()` 类似，但会自动创建到达最后一级目录所需要的中间目录。

The *mode* parameter is passed to `mkdir()`; see *the mkdir() description* for how it is interpreted.

If *exist_ok* is `False` (the default), an `OSError` is raised if the target directory already exists.

注解: 如果要创建的路径元素包含 `pardir` (如 UNIX 系统中的 “..”) `makedirs()` 将无法明确目标。

本函数能正确处理 UNC 路径。

3.2 新版功能: *exist_ok* 参数。

在 3.4.1 版更改: 在 Python 3.4.1 以前，如果 *exist_ok* 为 `True`，且目录已存在，且 *mode* 与现有目录的权限不匹配，`makedirs()` 仍会抛出错误。由于无法安全地实现此行为，因此在 Python 3.4.1 中将该行为删除。请参阅 bpo-21082。

在 3.6 版更改: 接受一个 *path-like object*。

os.**mkfifo**(*path*, *mode=0o666*, *, *dir_fd=None*)

创建一个名为 *path* 的 FIFO（命名管道，一种先进先出队列），具有以数字表示的权限状态 *mode*。将从 mode 中首先减去当前的 umask 值。

本函数支持基于目录描述符的相对路径。

FIFO 是可以像常规文件一样访问的管道。FIFO 如果没有被删除（如使用 `os.unlink()`），会一直存在。通常，FIFO 用作 “客户端” 和 “服务器” 进程之间的汇合点：服务器打开 FIFO 进行读取，而客户端打开 FIFO 进行写入。请注意，`mkfifo()` 不会打开 FIFO —它只是创建汇合点。

Availability: Unix.

3.3 新版功能: *dir_fd* 参数。

在 3.6 版更改: 接受一个 *path-like object*。

os.**mknod**(*path*, *mode=0o600*, *device=0*, *, *dir_fd=None*)

创建一个名为 *path* 的文件系统节点（文件，设备专用文件或命名管道）。*mode* 指定权限和节点类型，方法是将权限与下列节点类型 `stat.S_IFREG`、`stat.S_IFCHR`、`stat.S_IFBLK` 和 `stat.S_IFIFO` 之一（按位或）组合（这些常量可以在 `stat` 模块中找到）。对于 `stat.S_IFCHR` 和 `stat.S_IFBLK`，*device* 参数指定了新创建的设备专用文件（可能会用到 `os.makedev()`），否则该参数将被忽略。

本函数支持基于目录描述符的相对路径。

Availability: Unix.

3.3 新版功能: *dir_fd* 参数。

在 3.6 版更改: 接受一个 *path-like object*。

os.**major**(*device*)

提取主设备号，提取自原始设备号（通常是 `stat` 中的 `st_dev` 或 `st_rdev` 字段）。

os.**minor**(*device*)
提取次设备号，提取自原始设备号（通常是 stat 中的 st_dev 或 st_rdev 字段）。

os.**makedev**(*major*, *minor*)
将主设备号和次设备号组合成原始设备号。

os.**pathconf**(*path*, *name*)
返回所给名称的文件有关的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX.1，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 pathconf_names 字典中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 pathconf_names，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

本函数支持*指定文件描述符为参数*。

Availability: Unix.

在 3.6 版更改: 接受一个*path-like object*。

os.**pathconf_names**
字典，表示映射关系，为*pathconf()* 和*fpathconf()* 可接受名称与操作系统为这些名称定义的整数值之间的映射。这可用于判断系统已定义了哪些名称。

Availability: Unix.

os.**readlink**(*path*, *, *dir_fd=None*)
返回一个字符串，为符号链接指向的实际路径。其结果可以是绝对或相对路径。如果是相对路径，则可用 os.path.join(os.path.dirname(path), result) 转换为绝对路径。

如果 *path* 是字符串对象（直接传入或通过*PathLike* 接口间接传入），则结果也将是字符串对象，且此类调用可能会引发 UnicodeDecodeError。如果 *path* 是字节对象（直接传入或间接传入），则结果将会是字节对象。

本函数支持*基于目录描述符的相对路径*。

Availability: Unix, Windows

在 3.2 版更改: 添加对 Windows 6.0 (Vista) 符号链接的支持。

3.3 新版功能: *dir_fd* 参数。

在 3.6 版更改: 接受一个*path-like object*。

os.**remove**(*path*, *, *dir_fd=None*)
Remove (delete) the file *path*. If *path* is a directory, *OSError* is raised. Use *rmdir()* to remove directories.

本函数支持*基于目录描述符的相对路径*。

在 Windows 上，尝试删除正在使用的文件会抛出异常。而在 Unix 上，虽然该文件的条目会被删除，但分配给文件的存储空间仍然不可用，直到原始文件不再使用为止。

本函数在语义上与*unlink()* 相同。

3.3 新版功能: *dir_fd* 参数。

在 3.6 版更改: 接受一个*path-like object*。

os.**removedirs**(*name*)
递归删除目录。工作方式类似于*rmdir()*，不同之处在于，如果成功删除了末尾一级目录，*removedirs()* 会尝试依次删除 *path* 中提到的每个父目录，直到抛出错误为止（但该错误会被忽略，因为这通常表示父目录不是空目录）。例如，os.removedirs('foo/bar/baz') 将首先删除目

录 'foo/bar/baz'，然后如果 'foo/bar' 和 'foo' 为空，则继续删除它们。如果无法成功删除末尾一级目录，则抛出*OSError* 异常。

在 3.6 版更改: 接受一个*path-like object*。

os.**rename**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)
Rename the file or directory *src* to *dst*. If *dst* is a directory, *OSError* will be raised. On Unix, if *dst* exists and is a file, it will be replaced silently if the user has permission. The operation may fail on some Unix flavors if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement). On Windows, if *dst* already exists, *OSError* will be raised even if it is a file.

本函数支持将 *src_dir_fd* 和 *dst_dir_fd* 中的一个或两个指定为基于目录描述符的相对路径。

如果需要在不同平台上都能替换目标，请使用*replace()*。

3.3 新版功能: *src_dir_fd* 和 *dst_dir_fd* 参数。

在 3.6 版更改: 接受一个类路径对象 作为 *src* 和 *dst*。

os.**renames**(*old*, *new*)
递归重命名目录或文件。工作方式类似*rename()*，除了会首先创建新路径所需的中间目录。重命名后，将调用*removedirs()* 删除旧路径中不需要的目录。

注解: 如果用户没有权限删除末级的目录或文件，则本函数可能会无法建立新的目录结构。

在 3.6 版更改: 接受一个类路径对象 作为 *old* 和 *new*。

os.**replace**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)
将文件或目录 *src* 重命名为 *dst*。如果 *dst* 是目录，将抛出*OSError* 异常。如果 *dst* 已存在且为文件，则在用户具有权限的情况下，将对其进行静默替换。如果 *src* 和 *dst* 在不同的文件系统上，本操作可能会失败。如果成功，重命名操作将是一个原子操作（这是 POSIX 的要求）。

本函数支持将 *src_dir_fd* 和 *dst_dir_fd* 中的一个或两个指定为基于目录描述符的相对路径。

3.3 新版功能.

在 3.6 版更改: 接受一个类路径对象 作为 *src* 和 *dst*。

os.**rmdir**(*path*, *, *dir_fd=None*)
Remove (delete) the directory *path*. Only works when the directory is empty, otherwise, *OSError* is raised. In order to remove whole directory trees, *shutil.rmtree()* can be used.

本函数支持基于目录描述符的相对路径。

3.3 新版功能: *dir_fd* 参数。

在 3.6 版更改: 接受一个*path-like object*。

os.**scandir**(*path='.'*)
返回一个迭代出*os.DirEntry* 对象的迭代器，这些对象对应于 *path* 目录中的条目。条目的生成顺序是任意的，特殊条目 '.' 和 '..' 不包括在内。

如果需要文件类型或文件属性信息，使用*scandir()* 代替*listdir()* 可以大大提高这部分代码的性能，因为如果操作系统在扫描目录时返回的是*os.DirEntry* 对象，则该对象包含了这些信息。所有*os.DirEntry* 的方法都可能执行一次系统调用，但是*is_dir()* 和*is_file()* 通常只在有符号链接时才执行一次系统调用。*os.DirEntry.stat()* 在 Unix 上始终需要一次系统调用，而在 Windows 上只在有符号链接时才需要。

path 可以是类路径对象。如果 *path* 是（直接/间接由*PathLike* 接口返回的）"bytes" 类型，那么每个*os.DirEntry* 的*name* 和*path* 属性将是 bytes 类型，其他情况下是 str 类型。

scandir() 迭代器支持上下文管理 协议，并具有以下方法：

scandir.**close**()
关闭迭代器并释放占用的资源。

当迭代器迭代完毕，或垃圾回收，或迭代过程出错时，将自动调用本方法。但仍建议显式调用它或使用 with 语句。

3.6 新版功能.

下面的例子演示了*scandir()* 的简单用法，用来显示给定 *path* 中所有不以 '.' 开头的文件（不包括目录）。entry.is_file() 通常不会增加一次额外的系统调用：

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

注解: 在基于 Unix 的系统上，*scandir()* 使用系统的 opendir() 和 readdir() 函数。在 Windows 上，它使用 Win32 FindFirstFileW 和 FindNextFileW 函数。

3.5 新版功能.

3.6 新版功能: 添加了对上下文管理 协议和*close()* 方法的支持。如果*scandir()* 迭代器没有迭代完毕且没有显式关闭，其析构函数将发出*ResourceWarning* 警告。

本函数接受一个类路径对象。

class os.**DirEntry**
由*scandir()* 产生的对象，用于显示目录内某个条目的文件路径和其他文件属性。

scandir() 将在不进行额外系统调用的情况下，提供尽可能多的此类信息。每次进行 stat() 或 lstat() 系统调用时，os.DirEntry 对象会将结果缓存下来。

os.DirEntry 实例不适合存储在长期存在的数据结构中，如果你知道文件元数据已更改，或者自调用*scandir()* 以来已经经过了很长时间，请调用 os.stat(entry.path) 来获取最新信息。

因为 os.DirEntry 方法可以进行系统调用，所以它也可能抛出*OSError* 异常。如需精确定位错误，可以逐个调用 os.DirEntry 中的方法来捕获*OSError*，并适当处理。

为了能直接用作类路径对象，os.DirEntry 实现了*PathLike* 接口。

os.DirEntry 实例所包含的属性和方法如下：

name
本条目的基本文件名，是根据*scandir()* 的 *path* 参数得出的相对路径。

如果*scandir()* 的 *path* 参数是 bytes 类型，则*name* 属性也是 bytes 类型，否则为 str。使用*fsdecode()* 解码 byte 类型的文件名。

path
The entry’s full path name: equivalent to os.path.join(scandir_path, entry.name) where *scandir_path* is the *scandir()* *path* argument. The path is only absolute if the *scandir()* *path* argument was absolute.

如果*scandir()* 的 *path* 参数是 bytes 类型，则*path* 属性也是 bytes 类型，否则为 str。使用*fsdecode()* 解码 byte 类型的文件名。

inode()
返回本条目的索引节点号。

这一结果是缓存在 os.DirEntry 对象中的，请调用 os.stat(entry.path, follow_symlinks=False).st_ino 来获取最新信息。

一开始没有缓存时，在 Windows 上需要一次系统调用，但在 Unix 上不需要。

is_dir(*, *follow_symlinks=True*)
如果本条目是目录，或是指向目录的符号链接，则返回 True。如果本条目是文件，或指向任何其他类型的文件，或该文件不再存在，则返回 False。

如果 *follow_symlinks* 是 False，那么仅当本条目为目录时返回 True``（不跟踪符号链接），如果本条目是任何类型的文件，或该文件不再存在，则返回 ``False。

这一结果是缓存在 os.DirEntry 对象中的，且 *follow_symlinks* 为 True 和 False 时的缓存是分开的。请调用*os.stat()* 和*stat.S_ISDIR()* 来获取最新信息。

一开始没有缓存时，大多数情况下不需要系统调用。特别是对于非符号链接，Windows 和 Unix 都不需要系统调用，除非某些 Unix 文件系统（如网络文件系统）返回了 dirent.d_type == DT_UNKNOWN。如果本条目是符号链接，则需要一次系统调用来跟踪它（除非 *follow_symlinks* 为 False）。

本方法可能引发*OSError* 异常，如*PermissionError* 异常，但*FileNotFoundError* 异常会被内部捕获且不会引发。

is_file(*, *follow_symlinks=True*)
如果本条目是文件，或是指向文件的符号链接，则返回 True。如果本条目是目录，或指向目录，或指向其他非文件条目，或该文件不再存在，则返回 False。

如果 *follow_symlinks* 是 False，那么仅当本条目为文件时返回 True``（不跟踪符号链接），如果本条目是目录或其他非文件条目，或该文件不再存在，则返回 ``False。

这一结果是缓存在 os.DirEntry 对象中的。缓存、系统调用、异常抛出都与*is_dir()* 一致。

is_symlink()
如果本条目是符号链接（即使是断开的链接），返回 True。如果是目录或任何类型的文件，或本条目不再存在，返回 False。

这一结果是缓存在 os.DirEntry 对象中的，请调用*os.path.islink()* 来获取最新信息。

一开始没有缓存时，大多数情况下不需要系统调用。其实 Windows 和 Unix 都不需要系统调用，除非某些 Unix 文件系统（如网络文件系统）返回了 dirent.d_type == DT_UNKNOWN。

本方法可能引发*OSError* 异常，如*PermissionError* 异常，但*FileNotFoundError* 异常会被内部捕获且不会引发。

stat(*, *follow_symlinks=True*)
返回本条目对应的*stat_result* 对象。本方法默认会跟踪符号链接，要获取符号链接本身的 stat，请添加 follow_symlinks=False 参数。

On Unix, this method always requires a system call. On Windows, it only requires a system call if *follow_symlinks* is True and the entry is a symbolic link.

在 Windows 上，*stat_result* 的 st_ino、st_dev 和 st_nlink 属性总是为零。请调用*os.stat()* 以获得这些属性。

这一结果是缓存在 os.DirEntry 对象中的，且 *follow_symlinks* 为 True 和 False 时的缓存是分开的。请调用*os.stat()* 来获取最新信息。

注意，os.DirEntry 和*pathlib.Path* 的几个属性和方法之间存在很好的对应关系。具体来说是 name 属性，以及 is_dir()、is_file()、is_symlink() 和 stat() 方法，在两个类中具有相同的含义。

3.5 新版功能.

在 3.6 版更改: 添加了对*PathLike* 接口的支持。在 Windows 上添加了对*bytes* 类型路径的支持。

os.**stat**(*path*, *, *dir_fd=None*, *follow_symlinks=True*)
获取文件或文件描述符的状态。在所给路径上执行等效于 stat() 系统调用的操作。*path* 可以是

字符串类型，或（直接/间接由*PathLike* 接口返回的）bytes 类型，或打开的文件描述符。返回一个*stat_result* 对象。

本函数默认会跟踪符号链接，要获取符号链接本身的 stat，请添加 `follow_symlinks=False` 参数，或使用*lstat()*。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

示例:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

参见:

fstat() 和*lstat()* 函数。

3.3 新版功能: 增加 *dir_fd* 和 *follow_symlinks* 参数，可指定文件描述符代替路径。

在 3.6 版更改: 接受一个*path-like object*。

class os.**stat_result**

本对象的属性大致对应于 `stat` 结构体成员，主要作为*os.stat()*、*os.fstat()* 和*os.lstat()* 的返回值。

属性:

st_mode
文件模式：包括文件类型和文件模式位（即权限位）。

st_ino
与平台有关，但如果不为零，则根据 `st_dev` 值唯一地标识文件。通常:

- 在 Unix 上该值表示索引节点号 (inode number)。
- 在 Windows 上该值表示 文件索引号 。

st_dev
该文件所在设备的标识符。

st_nlink
硬链接的数量。

st_uid
文件所有者的用户 ID。

st_gid
文件所有者的用户组 ID。

st_size
文件大小（以字节为单位），文件可以是常规文件或符号链接。符号链接的大小是它包含的路径的长度，不包括末尾的空字节。

时间戳:

st_atime
最近的访问时间，以秒为单位。

st_mtime
最近的修改时间，以秒为单位。

st_ctime
取决于平台：

- 在 Unix 上表示最近的元数据更改时间，
- 在 Windows 上表示创建时间，以秒为单位。

st_atime_ns
最近的访问时间，以纳秒表示，为整数。

st_mtime_ns
最近的修改时间，以纳秒表示，为整数。

st_ctime_ns
取决于平台：

- 在 Unix 上表示最近的元数据更改时间，
- 在 Windows 上表示创建时间，以纳秒表示，为整数。

See also the *stat_float_times()* function.

注解： *st_atime*、*st_mtime* 和*st_ctime* 属性的确切含义和分辨率取决于操作系统和文件系统。例如，在使用 FAT 或 FAT32 文件系统的 Windows 上，*st_mtime* 有 2 秒的分辨率，而*st_atime* 仅有 1 天的分辨率。详细信息请参阅操作系统文档。

类似地，尽管*st_atime_ns*、*st_mtime_ns* 和*st_ctime_ns* 始终以纳秒表示，但许多系统并不提供纳秒精度。在确实提供纳秒精度的系统上，用于存储*st_atime*、*st_mtime* 和*st_ctime* 的浮点对象无法保留所有精度，因此不够精确。如果需要确切的时间戳，则应始终使用*st_atime_ns*、*st_mtime_ns* 和*st_ctime_ns*。

在某些 Unix 系统上（如 Linux 上），以下属性可能也可用：

st_blocks
为文件分配的字节块数，每块 512 字节。文件是稀疏文件时，它可能小于*st_size*/512。

st_blksize
“首选的”块大小，用于提高文件系统 I/O 效率。写入文件时块大小太小可能会导致读取-修改-重写效率低下。

st_rdev
设备类型（如果是 inode 设备）。

st_flags
用户定义的文件标志位。

在其他 Unix 系统上（如 FreeBSD 上），以下属性可能可用（但仅当 root 使用它们时才被填充）：

st_gen
文件生成号。

st_birthtime
文件创建时间。

在 Mac OS 系统上，以下属性可能也可用：

st_rsize
文件的实际大小。

st_creator
文件的创建者。

st_type
文件类型。

On Windows systems, the following attribute is also available:

st_file_attributes
Windows 文件属性：dwFileAttributes，由 GetFileInformationByHandle() 返回的 BY_HANDLE_FILE_INFORMATION 结构体的成员之一。请参阅*stat* 模块中的 FILE_ATTRIBUTE_* 常量。

标准模块*stat* 中定义了函数和常量，这些函数和常量可用于从 stat 结构体中提取信息。（在 Windows 上，某些项填充的是虚值。）

为了向后兼容，一个*stat_result* 实例还可以作为至少包含 10 个整数的元组访问，以提供 stat 结构中最重要（和可移植）的成员，整数顺序为*st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime*。某些实现可能在末尾还有更多项。为了与旧版 Python 兼容，以元组形式访问*stat_result* 始终返回整数。

3.3 新版功能: 添加了*st_atime_ns*、*st_mtime_ns* 和*st_ctime_ns* 成员。

3.5 新版功能: 在 Windows 上添加了*st_file_attributes* 成员。

在 3.5 版更改: 在 Windows 上，如果可用，会返回文件索引作为*st_ino* 的值。

os.**stat_float_times**([*newvalue*])
Determine whether *stat_result* represents time stamps as float objects. If *newvalue* is True, future calls to *stat()* return floats, if it is False, future calls return ints. If *newvalue* is omitted, return the current setting.

For compatibility with older Python versions, accessing *stat_result* as a tuple always returns integers.

Python now returns float values by default. Applications which do not work correctly with floating point time stamps can use this function to restore the old behaviour.

The resolution of the timestamps (that is the smallest possible fraction) depends on the system. Some systems only support second resolution; on these systems, the fraction will always be zero.

It is recommended that this setting is only changed at program startup time in the *__main__* module; libraries should never change this setting. If an application uses a library that works incorrectly if floating point time stamps are processed, this application should turn the feature off until the library has been corrected.

3.3 版后已移除.

os.**statvfs**(*path*)
Perform a statvfs() system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the statvfs structure, namely: f_bsize, f_frsize, f_blocks, f_bfree, f_bavail, f_files, f_ffree, f_favail, f_flag, f_namemax.

为 f_flag 属性位定义了两个模块级常量：如果存在 ST_RDONLY 位，则文件系统以只读挂载；如果存在 ST_NOSUID 位，则文件系统禁用或不支持 setuid/setgid 位。

为基于 GNU/glibc 的系统还定义了额外的模块级常量。它们是 ST_NODEV （禁止访问设备专用文件），ST_NOEXEC （禁止执行程序），ST_SYNCHRONOUS （写入后立即同步），ST_MANDLOCK （允许文件系统上的强制锁定），ST_WRITE （写入文件/目录/符号链接），ST_APPEND （仅追加文件），ST_IMMUTABLE （不可变文件），ST_NOATIME （不更新访问时间），ST_NODIRATIME （不更新目录访问时间），ST_RELATIME （相对于 mtime/ctime 更新访问时间）。

本函数支持指定文件描述符为参数。

Availability: Unix.

在 3.2 版更改: 添加了 ST_RDONLY 和 ST_NOSUID 常量。

3.3 新版功能: Added support for specifying an open file descriptor for *path*.

在 3.4 版更改: 添加了 ST_NODEV、ST_NOEXEC、ST_SYNCHRONOUS、ST_MANDLOCK、ST_WRITE、ST_APPEND、ST_IMMUTABLE、ST_NOATIME、ST_NODIRATIME 和 ST_RELATIME 常量。

在 3.6 版更改: 接受一个*path-like object*。

os.**supports_dir_fd**

A *Set* object indicating which functions in the *os* module permit use of their *dir_fd* parameter. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency's sakes, functions that support *dir_fd* always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits use of its *dir_fd* parameter, use the `in` operator on `supports_dir_fd`. As an example, this expression determines whether the *dir_fd* parameter of *os.stat()* is locally available:

```
os.stat in os.supports_dir_fd
```

目前 *dir_fd* 参数仅在 Unix 平台上有效，在 Windows 上均无效。

3.3 新版功能.

os.**supports_effective_ids**

A *Set* object indicating which functions in the *os* module permit use of the *effective_ids* parameter for *os.access()*. If the local platform supports it, the collection will contain *os.access()*, otherwise it will be empty.

To check whether you can use the *effective_ids* parameter for *os.access()*, use the `in` operator on `supports_effective_ids`, like so:

```
os.access in os.supports_effective_ids
```

Currently *effective_ids* only works on Unix platforms; it does not work on Windows.

3.3 新版功能.

os.**supports_fd**

A *Set* object indicating which functions in the *os* module permit specifying their *path* parameter as an open file descriptor. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency's sakes, functions that support *fd* always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits specifying an open file descriptor for its *path* parameter, use the `in` operator on `supports_fd`. As an example, this expression determines whether *os.chdir()* accepts open file descriptors when called on your local platform:

```
os.chdir in os.supports_fd
```

3.3 新版功能.

os.**supports_follow_symlinks**

A *Set* object indicating which functions in the *os* module permit use of their *follow_symlinks* parameter. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency's sakes, functions that support *follow_symlinks* always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits use of its *follow_symlinks* parameter, use the `in` operator on `supports_follow_symlinks`. As an example, this expression determines whether the *follow_symlinks* parameter of *os.stat()* is locally available:

```
os.stat in os.supports_follow_symlinks
```

3.3 新版功能.

os.**symlink**(*src*, *dst*, *target_is_directory=False*, *, *dir_fd=None*)
创建一个指向 *src* 的符号链接，名为 *dst*。

在 Windows 上，符号链接可以表示文件或目录两种类型，并且不会动态改变类型。如果目标存在，则新建链接的类型将与目标一致。否则，如果 *target_is_directory* 为 `True`，则符号链接将创建为目录链接，为 `False`（默认）将创建为文件链接。在非 Windows 平台上，*target_is_directory* 被忽略。

Symbolic link support was introduced in Windows 6.0 (Vista). *symlink()* will raise a *NotImplementedError* on Windows versions earlier than 6.0.

本函数支持基于目录描述符的相对路径。

注解： On Windows, the *SeCreateSymbolicLinkPrivilege* is required in order to successfully create symlinks. This privilege is not typically granted to regular users but is available to accounts which can escalate privileges to the administrator level. Either obtaining the privilege or running your application as an administrator are ways to successfully create symlinks.

当本函数由非特权账户调用时，抛出*OSError* 异常。

Availability: Unix, Windows.

在 3.2 版更改: 添加对 Windows 6.0 (Vista) 符号链接的支持。

3.3 新版功能: 添加了 *dir_fd* 参数，现在在非 Windows 平台上允许 *target_is_directory* 参数。

在 3.6 版更改: 接受一个类路径对象 作为 *src* 和 *dst*。

os.**sync**()
强制将所有内容写入磁盘。

Availability: Unix.

3.3 新版功能.

os.**truncate**(*path*, *length*)
截断 *path* 对应的文件，以使其最大为 *length* 字节大小。

本函数支持指定文件描述符为参数。

Availability: Unix, Windows.

3.3 新版功能.

在 3.5 版更改: 添加了 Windows 支持

在 3.6 版更改: 接受一个*path-like object*。

os.**unlink**(*path*, *, *dir_fd=None*)
移除（删除）文件 *path*。该函数在语义上与*remove()* 相同，`unlink` 是其传统的 Unix 名称。请参阅*remove()* 的文档以获取更多信息。

3.3 新版功能: *dir_fd* 参数。

在 3.6 版更改: 接受一个*path-like object*。

os.**utime**(*path*, *times=None*, **[, ns]*, *dir_fd=None*, *follow_symlinks=True*)
设置文件 *path* 的访问时间和修改时间。

utime() 有 *times* 和 *ns* 两个可选参数，它们指定了设置给 *path* 的时间，用法如下：

- 如果指定 *ns*，它必须是一个 `(atime_ns, mtime_ns)` 形式的二元组，其中每个成员都是一个表示纳秒的整数。
- 如果 *times* 不为 `None`，则它必须是 `(atime, mtime)` 形式的二元组，其中每个成员都是一个表示秒的 int 或 float。
- 如果 *times* 为 `None` 且未指定 *ns*，则相当于指定 `ns=(atime_ns, mtime_ns)`，其中两个时间均为当前时间。

同时为 *times* 和 *ns* 指定元组会出错。

Whether a directory can be given for *path* depends on whether the operating system implements directories as files (for example, Windows does not). Note that the exact times you set here may not be returned by a subsequent *stat()* call, depending on the resolution with which your operating system records access and modification times; see *stat()*. The best way to preserve exact times is to use the *st_atime_ns* and *st_mtime_ns* fields from the *os.stat()* result object with the *ns* parameter to *utime*.

本函数支持指定指定文件描述符为参数、基于目录描述符的相对路径 和不跟踪符号链接。

3.3 新版功能: Added support for specifying an open file descriptor for *path*, and the *dir_fd*, *follow_symlinks*, and *ns* parameters.

在 3.6 版更改: 接受一个*path-like object*。

os.**walk**(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)
生成目录树中的文件名，方式是按上-> 下或下-> 上顺序浏览目录树。对于以 *top* 为根的目录树中的每个目录（包括 *top* 本身），它都会生成一个三元组 `(dirpath, dirnames, filenames)`。

dirpath 是一个字符串，表示目录的路径。*dirnames* 是一个列表，内含 *dirpath* 中子目录的名称（不包括 `'.'` 和 `'..'`）。*filenames* 也是列表，内含 *dirpath* 中非目录文件的名称。注意，列表中的名称不包含路径部分。要获取 *dirpath* 中文件或目录的完整路径（从 *top* 起始），请执行 `os.path.join(dirpath, name)`。

如果可选参数 *topdown* 为 `True` 或未指定，则在所有子目录的三元组之前生成父目录的三元组（目录是自上而下生成的）。如果 *topdown* 为 `False`，则在所有子目录的三元组生成之后再生成父目录的三元组（目录是自下而上生成的）。无论 *topdown* 为何值，在生成目录及其子目录的元组之前，都将检索全部子目录列表。

当 *topdown* 为 `True` 时，调用者可以就地修改 *dirnames* 列表（也许用到了 `del` 或切片），而*walk()* 将仅仅递归到仍保留在 *dirnames* 中的子目录内。这可用于减少搜索、加入特定的访问顺序，甚至可在继续*walk()* 之前告知*walk()* 由调用者新建或重命名的目录的信息。当 *topdown* 为 `False` 时，修改 *dirnames* 对 walk 的行为没有影响，因为在自下而上模式中，*dirnames* 中的目录是在 *dirpath* 本身之前生成的。

默认将忽略*scandir()* 调用中的错误。如果指定了可选参数 *onerror*，它应该是一个函数。出错时它会被调用，参数是一个*OSError* 实例。它可以报告错误然后继续遍历，或者抛出异常然后中止遍历。注意，可以从异常对象的 `filename` 属性中获取出错的文件名。

walk() 默认不会递归进指向目录的符号链接。可以在支持符号链接的系统上将 *followlinks* 设置为 `True`，以访问符号链接指向的目录。

注解: 注意，如果链接指向自身的父目录，则将 *followlinks* 设置为 `True` 可能导致无限递归。*walk()* 不会记录它已经访问过的目录。

注解: 如果传入的是相对路径，请不要在恢复*walk()* 之间更改当前工作目录。*walk()* 不会更改当前目录，并假定其调用者也不会更改当前目录。

下面的示例遍历起始目录内所有子目录，打印每个目录内的文件占用的字节数，CVS 子目录不会被遍历:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

在下一个示例（*shutil.rmtree()* 的简单实现）中，必须使树自下而上，因为*rmdir()* 只允许在目录为空时删除目录:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

在 3.5 版更改: 现在，本函数调用的是*os.scandir()* 而不是*os.listdir()*，从而减少了调用*os.stat()* 的次数而变得更快。

在 3.6 版更改: 接受一个*path-like object*。

os.**fwalk**(*top='.'*, *topdown=True*, *onerror=None*, ***, *follow_symlinks=False*, *dir_fd=None*)

本方法的行为与*walk()* 完全一样，除了它产生的是 4 元组 `(dirpath, dirnames, filenames, dirfd)`，并且它支持 `dir_fd`。

dirpath、*dirnames* 和 *filenames* 与*walk()* 输出的相同，*dirfd* 是指向目录 *dirpath* 的文件描述符。

本函数始终支持*基于目录描述符的相对路径* 和*不跟踪符号链接*。但是请注意，与其他函数不同，*fwalk()* 的 *follow_symlinks* 的默认值为 `False`。

注解: 由于*fwalk()* 会生成文件描述符，而它们仅在下一个迭代步骤前有效，因此如果要将描述符保留更久，则应复制它们（比如使用*dup()*）。

下面的示例遍历起始目录内所有子目录，打印每个目录内的文件占用的字节数，CVS 子目录不会被遍历:

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
    print(root, "consumes", end="")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end="")
    print("bytes in", len(files), "non-directory files")
```

(下页继续)

(续上页)

```
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

在下一个示例中，必须使树自下而上遍历，因为*rmdir()* 只允许在目录为空时删除目录:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

Availability: Unix.

3.3 新版功能.

在 3.6 版更改: 接受一个*path-like object*。

Linux 扩展属性

3.3 新版功能.

这些函数仅在 Linux 上可用。

os.**getxattr**(*path*, *attribute*, *, *follow_symlinks=True*)
返回 *path* 的扩展文件系统属性 *attribute* 的值。*attribute* 可以是 bytes 或 str （直接/间接通过*PathLike* 接口返回的）。如果是 str，则使用文件系统编码来编码字符串。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

在 3.6 版更改: 接受一个类路径对象 作为 *path* 和 *attribute*。

os.**listxattr**(*path=None*, *, *follow_symlinks=True*)
返回一个列表，包含 *path* 的所有扩展文件系统属性。列表中的属性都表示为字符串，它们是根据文件系统编码解码出来的。如果 *path* 为 None，则*listxattr()* 将检查当前目录。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

在 3.6 版更改: 接受一个*path-like object*。

os.**removexattr**(*path*, *attribute*, *, *follow_symlinks=True*)
从 *path* 中删除扩展文件系统属性 *attribute*。*attribute* 应该是 bytes 或 str （直接/间接通过*PathLike* 接口返回的）。如果是字符串，则它应使用文件系统编码来编码。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

在 3.6 版更改: 接受一个类路径对象 作为 *path* 和 *attribute*。

os.**setxattr**(*path*, *attribute*, *value*, *flags=0*, *, *follow_symlinks=True*)
将 *path* 的扩展文件系统属性 *attribute* 设置为 *value*。*attribute* 必须是没有空字符的 bytes 或 str （直接传入或通过*PathLike* 接口间接传入）。如果是 str，则应使用文件系统编码进行编码。*flags* 可以是*XATTR_REPLACE* 或*XATTR_CREATE*。如果指定*XATTR_REPLACE* 而该属性不存在，则抛出 EEXISTS 异常。如果指定*XATTR_CREATE* 而该属性已经存在，则不会创建该属性，抛出 ENODATA 异常。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

注解： Linux kernel 2.6.39 以下版本的一个 bug 导致在某些文件系统上，flags 参数会被忽略。

在 3.6 版更改: 接受一个类路径对象 作为 *path* 和 *attribute*。

os.**XATTR_SIZE_MAX**
一条扩展属性的值的最大大小。在当前的 Linux 上是 64 KiB。

os.**XATTR_CREATE**
这是*setxattr()* 的 flags 参数的可取值，它表示该操作必须创建一个属性。

os.**XATTR_REPLACE**
这是*setxattr()* 的 flags 参数的可取值，它表示该操作必须替换现有属性。

16.1.6 进程管理

下列函数可用于创建和管理进程。

所有*exec** 函数都接受一个参数列表，用来给新程序加载到它的进程中。在所有情况下，传递给新程序的第一个参数是程序本身的名称，而不是用户在命令行上输入的参数。对于 C 程序员来说，这就是传递给 `main()` 函数的 `argv[0]`。例如，`os.execv('/bin/echo', ['foo', 'bar'])` 只会在标准输出上打印 `bar`，而 `foo` 会被忽略。

os.**abort**()
发送 `SIGABRT` 信号到当前进程。在 Unix 上，默认行为是生成一个核心转储。在 Windows 上，该进程立即返回退出代码 `3`。请注意，使用*signal.signal()* 可以为 `SIGABRT` 注册 Python 信号处理程序，而调用本函数将不会调用按前述方法注册的程序。

os.**execl**(*path*, *arg0*, *arg1*, ...)
os.**execle**(*path*, *arg0*, *arg1*, ..., *env*)
os.**execlp**(*file*, *arg0*, *arg1*, ...)
os.**execlpe**(*file*, *arg0*, *arg1*, ..., *env*)
os.**execv**(*path*, *args*)
os.**execve**(*path*, *args*, *env*)
os.**execvp**(*file*, *args*)
os.**execvpe**(*file*, *args*, *env*)
这些函数都将执行一个新程序，以替换当前进程。它们没有返回值。在 Unix 上，新程序会加载到当前进程中，且进程号与调用者相同。过程中的错误会被报告为*OSError* 异常。

当前进程会被立即替换。打开的文件对象和描述符都不会刷新，因此如果这些文件上可能缓冲了数据，则应在调用*exec** 函数之前使用 `sys.stdout.flush()` 或*os.fsync()* 刷新它们。

*exec** 函数的“l”和“v”变体不同在于命令行参数的传递方式。如果在编码时固定了参数数量，则“l”变体可能是最方便的，各参数作为 `execl*()` 函数的附加参数传入即可。当参数数量可变时，”v”变体更方便，参数以列表或元组的形式作为 *args* 参数传递。在这两种情况下，子进程的第一个参数都应该是即将运行的命令名称，但这不是强制性的。

结尾包含“p”的变体（*execlp()*、*execlpe()*、*execvp()* 和*execvpe()* ）将使用 `PATH` 环境变量来查找程序 *file*。当环境被替换时（使用下一段讨论的*exec*e* 变体之一），`PATH` 变量将来自于新环境。其他变体*execl()*、*execle()*、*execv()* 和*execve()* 不使用 `PATH` 变量来查找程序，因此 *path* 必须包含正确的绝对或相对路径。

对于*execle()*、*execlpe()*、*execve()* 和*execvpe()* （都以“e”结尾），*env* 参数是一个映射，用于定义新进程的环境变量（代替当前进程的环境变量）。而函数*execl()*、*execlp()*、*execv()* 和*execvp()* 会将当前进程的环境变量过继给新进程。

某些平台上的*execve()* 可以将 *path* 指定为打开的文件描述符。当前平台可能不支持此功能，可以使用*os.supports_fd* 检查它是否支持。如果不可用，则使用它会抛出*NotImplementedError* 异常。

Availability: Unix, Windows.

3.3 新版功能: Added support for specifying an open file descriptor for *path* for `execve()`.

在 3.6 版更改: 接受一个*path-like object*。

os.**_exit**(*n*)
以状态码 *n* 退出进程，不会调用清理处理程序，不会刷新 stdio，等等。

注解: 退出的标准方法是使用 `sys.exit(n)`。而`_exit()` 通常只应在`fork()` 出的子进程中使用。

以下是已定义的退出代码，可以用于`_exit()`，尽管它们不是必需的。这些退出代码通常用于 Python 编写的系统程序，例如邮件服务器的外部命令传递程序。

注解: 其中部分退出代码在部分 Unix 平台上可能不可用，因为平台间存在差异。如果底层平台定义了这些常量，那上层也会定义。

os.**EX_OK**
退出代码，表示未发生任何错误。

Availability: Unix.

os.**EX_USAGE**
退出代码，表示命令使用不正确，如给出的参数数量有误。

Availability: Unix.

os.**EX_DATAERR**
退出代码，表示输入数据不正确。

Availability: Unix.

os.**EX_NOINPUT**
退出代码，表示某个输入文件不存在或不可读。

Availability: Unix.

os.**EX_NOUSER**
退出代码，表示指定的用户不存在。

Availability: Unix.

os.**EX_NOHOST**
退出代码，表示指定的主机不存在。

Availability: Unix.

os.**EX_UNAVAILABLE**
退出代码，表示所需的服务不可用。

Availability: Unix.

os.**EX_SOFTWARE**
退出代码，表示检测到内部软件错误。

Availability: Unix.

os.**EX_OSERR**
退出代码，表示检测到操作系统错误，例如无法 fork 或创建管道。

Availability: Unix.

os.**EX_OSFILE**
退出代码，意味着某些系统文件不存在，无法打开或发生其他错误。

Availability: Unix.

os.**EX_CANTCREAT**
退出代码，表示无法创建用户指定的输出文件。

Availability: Unix.

os.**EX_IOERR**
退出代码，表示对某些文件进行读写时发生错误。

Availability: Unix.

os.**EX_TEMPFAIL**
退出代码，表示发生了暂时性故障。它可能并非意味着真正的错误，例如在可重试的情况下无法建立网络连接。

Availability: Unix.

os.**EX_PROTOCOL**
退出代码，表示协议交换是非法的、无效的或无法解读的。

Availability: Unix.

os.**EX_NOPERM**
退出代码，表示没有足够的权限执行该操作（但不适用于文件系统问题）。

Availability: Unix.

os.**EX_CONFIG**
退出代码，表示发生某种配置错误。

Availability: Unix.

os.**EX_NOTFOUND**
退出代码，表示的内容类似于“找不到条目”。

Availability: Unix.

os.**fork**()
Fork 出一个子进程。在子进程中返回 `0`，在父进程中返回子进程的进程号。如果发生错误，则抛出*OSError* 异常。

Note that some platforms including FreeBSD <= 6.3 and Cygwin have known issues when using fork() from a thread.

> **警告：** 有关 SSL 模块与 fork() 结合的应用，请参阅*ssl*。

Availability: Unix.

os.**forkpty**()
Fork 出一个子进程，使用新的伪终端作为子进程的控制终端。返回一对 `(pid, fd)`，其中 *pid* 在子进程中为 `0`，这是父进程中新子进程的进程号，而 *fd* 是伪终端主设备的文件描述符。对于更便于移植的方法，请使用*pty* 模块。如果发生错误，则抛出*OSError* 异常。

Availability: some flavors of Unix.

os.**kill**(*pid*, *sig*)
将信号 *sig* 发送至进程 *pid*。特定平台上可用的信号常量定义在*signal* 模块中。

Windows：*signal.CTRL_C_EVENT* 和 *signal.CTRL_BREAK_EVENT* 信号是特殊信号，只能发送给共享同一个控制台窗口的控制台进程，如某些子进程。*sig* 取任何其他值将导致该进程被 TerminateProcess API 无条件终止，且退出代码为 *sig*。Windows 版本的 *kill()* 还需要传入待结束进程的句柄。

另请参阅 *signal.pthread_kill()*。

3.2 新版功能: Windows 支持。

os.**killpg**(*pgid*, *sig*)
将信号 *sig* 发送给进程组 *pgid*。

Availability: Unix.

os.**nice**(*increment*)
将进程的优先级（nice 值）增加 *increment*，返回新的 nice 值。

Availability: Unix.

os.**plock**(*op*)
将程序段锁定到内存中。*op* 的值（定义在 <sys/lock.h> 中）决定了哪些段被锁定。

Availability: Unix.

os.**popen**(*cmd*, *mode='r'*, *buffering=-1*)
打开一个管道，它通往 / 接受自命令 *cmd*。返回值是连接到管道的文件对象，根据 *mode* 是 'r'（默认）还是 'w' 决定该对象可以读取还是写入。*buffering* 参数与内置函数 *open()* 相应的参数含义相同。返回的文件对象只能读写文本字符串，不能是字节类型。

如果子进程成功退出，则 close 方法返回 *None*。如果发生错误，则返回子进程的返回码。在 POSIX 系统上，如果返回码为正，则它就是进程返回值左移一个字节后的值。如果返回码为负，则进程是被信号终止的，返回码取反后就是该信号。(例如，如果子进程被终止，则返回值可能是 - signal.SIGKILL。) 在 Windows 系统上，返回值包含子进程的返回码（有符号整数）。

本方法是使用 *subprocess.Popen* 实现的，如需更强大的方法来管理和沟通子进程，请参阅该类的文档。

os.**spawnl**(*mode*, *path*, *...*)
os.**spawnle**(*mode*, *path*, *...*, *env*)
os.**spawnlp**(*mode*, *file*, *...*)
os.**spawnlpe**(*mode*, *file*, *...*, *env*)
os.**spawnv**(*mode*, *path*, *args*)
os.**spawnve**(*mode*, *path*, *args*, *env*)
os.**spawnvp**(*mode*, *file*, *args*)
os.**spawnvpe**(*mode*, *file*, *args*, *env*)
在新进程中执行程序 *path*。

(注意，*subprocess* 模块提供了更强大的工具来生成新进程并跟踪执行结果，使用该模块比使用这些函数更好。尤其应当检查 *使用 subprocess 模块替换旧函数* 部分。)

mode 为 *P_NOWAIT* 时，本函数返回新进程的进程号。*mode* 为 *P_WAIT* 时，如果进程正常退出，返回退出代码，如果被杀死，返回 -signal，其中 *signal* 是杀死进程的信号。在 Windows 上，进程号实际上是进程句柄，因此可以与 *waitpid()* 函数一起使用。

*spawn** 函数的 “l” 和 “v” 变体不同在于命令行参数的传递方式。如果在编码时固定了参数数量，则 “l” 变体可能是最方便的，各参数作为 spawnl*() 函数的附加参数传入即可。当参数数量可变时，” v” 变体更方便，参数以列表或元组的形式作为 *args* 参数传递。在这两种情况下，子进程的第一个参数都必须是即将运行的命令名称。

结尾包含第二个 “p” 的变体 (*spawnlp()*、*spawnlpe()*、*spawnvp()* 和 *spawnvpe()*) 将使用 PATH 环境变量来查找程序 *file*。当环境被替换时（使用下一段讨论的 *spawn*e* 变体之一），PATH 变量将来自于新环境。其他变体 *spawnl()*、*spawnle()*、*spawnv()* 和 *spawnve()* 不使用 PATH 变量来查找程序，因此 *path* 必须包含正确的绝对或相对路径。

对于*spawnle()*、*spawnlpe()*、*spawnve()* 和*spawnvpe()* （都以“e”结尾），*env* 参数是一个映射，用于定义新进程的环境变量（代替当前进程的环境变量）。而函数*spawnl()*、*spawnlp()*、*spawnv()* 和*spawnvp()* 会将当前进程的环境变量过继给新进程。注意，*env* 字典中的键和值必须是字符串。无效的键或值将导致函数出错，返回值为 127。

例如，以下对*spawnlp()* 和*spawnvpe()* 的调用是等效的:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Availability: Unix, Windows. *spawnlp()*, *spawnlpe()*, *spawnvp()* and *spawnvpe()* are not available on Windows. *spawnle()* and *spawnve()* are not thread-safe on Windows; we advise you to use the *subprocess* module instead.

在 3.6 版更改: 接受一个*path-like object*。

os.**P_NOWAIT**
os.**P_NOWAITO**
*spawn** 系列函数的 *mode* 参数的可取值。如果给出这些值中的任何一个，则 spawn*() 函数将在创建新进程后立即返回，且返回值为进程号。

Availability: Unix, Windows.

os.**P_WAIT**
*spawn** 系列函数的 *mode* 参数的可取值。如果将 *mode* 指定为该值，则 spawn*() 函数将在新进程运行完毕后返回，运行成功则返回进程的退出代码，被信号终止则返回 -signal。

Availability: Unix, Windows.

os.**P_DETACH**
os.**P_OVERLAY**
*spawn** 系列函数的 *mode* 参数的可取值。它们比上面列出的值可移植性差。*P_DETACH* 与*P_NOWAIT* 相似，但是新进程会与父进程的控制台脱离。使用*P_OVERLAY* 则会替换当前进程，*spawn** 函数将不会返回。

Availability: Windows.

os.**startfile**(*path*[, *operation*])
使用已关联的应用程序打开文件。

当 *operation* 未指定或指定为 'open' 时，这类似于在 Windows 资源管理器中双击文件，或在交互式命令行中将文件名作为 **start** 命令的参数：通过扩展名相关联的应用程序（如果有）打开文件。

当指定另一个 *operation* 时，它必须是一个“命令动词”（“command verb”），该词指定对文件执行的操作。Microsoft 文档中的常用动词有 'print' 和 'edit' （用于文件），以及 'explore' 和 'find' （用于目录）。

关联的应用程序启动后*startfile()* 就会立即返回。本函数没有等待应用程序关闭的选项，也没有办法检索应用程序的退出状态。*path* 参数是基于当前目录的相对路径。如果要使用绝对路径，请确保第一个字符不是斜杠 ('/')，是斜杠的话底层的 Win32 ShellExecute() 函数将失效。使用*os.path.normpath()* 函数确保路径已针对 Win32 正确编码。

为了减少解释器的启动开销，直到第一次调用本函数后，才解析 Win32 ShellExecute() 函数。如果无法解析该函数，则抛出*NotImplementedError* 异常。

Availability: Windows.

os.**system**(*command*)
在子 shell 中执行命令（字符串）。这是调用标准 C 函数 system() 来实现的，因此限制条件与该函数

相同。对*sys.stdin* 等的更改不会反映在执行命令的环境中。*command* 产生的任何输出将被发送到解释器标准输出流。

在 Unix 上，返回值是进程的退出状态，编码格式与为*wait()* 指定的格式相同。注意，POSIX 没有指定 C 函数 system() 返回值的含义，因此 Python 函数的返回值与系统有关。

在 Windows 上，返回值是运行 *command* 后系统 Shell 返回的值。该 Shell 由 Windows 环境变量 COMSPEC: 给出：通常是 **cmd.exe**，它会返回命令的退出状态。在使用非原生 Shell 的系统上，请查阅 Shell 的文档。

subprocess 模块提供了更强大的工具来生成新进程并跟踪执行结果，使用该模块比使用本函数更好。参阅*subprocess* 文档中的*使用 subprocess 模块替换旧函数* 部分以获取有用的帮助。

Availability: Unix, Windows.

os.**times**()
返回当前的全局进程时间。返回值是一个有 5 个属性的对象：

- user - 用户时间
- *system* - 系统时间
- children_user - 所有子进程的用户时间
- children_system - 所有子进程的系统时间
- elapsed - 从过去的固定时间点起，经过的真实时间

为了向后兼容，该对象的行为也类似于五元组，按照 user，*system*，children_user，children_system 和 elapsed 顺序组成。

See the Unix manual page *times(2)* or the corresponding Windows Platform API documentation. On Windows, only user and *system* are known; the other attributes are zero.

Availability: Unix, Windows.

在 3.3 版更改: 返回结果的类型由元组变成一个类似元组的对象，同时具有命名的属性。

os.**wait**()
等待子进程执行完毕，返回一个元组，包含其 pid 和退出状态指示：一个 16 位数字，其低字节是终止该进程的信号编号，高字节是退出状态码（信号编号为零的情况下），如果生成了核心文件，则低字节的高位会置位。

Availability: Unix.

os.**waitid**(*idtype*, *id*, *options*)
等待一个或多个子进程执行完毕。*idtype* 可以是*P_PID*、*P_PGID* 或*P_ALL*。*id* 指定要等待的 pid。*options* 是由*WEXITED*、*WSTOPPED* 或*WCONTINUED* 中的一个或多个进行或运算构造的，且额外可以与*WNOHANG* 或*WNOWAIT* 进行或运算。返回值是一个对象，对应着 siginfo_t 结构体中的数据，即: si_pid, si_uid, si_signo, si_status, si_code 或 None （如果指定了*WNOHANG* 且没有子进程处于等待状态）。

Availability: Unix.

3.3 新版功能.

os.**P_PID**
os.**P_PGID**
os.**P_ALL**
waitid() 函数的 *idtype* 参数的可取值。它们影响 *id* 的解释方式。

Availability: Unix.

3.3 新版功能.

os.**WEXITED**
os.**WSTOPPED**
os.**WNOWAIT**
用于*waitid()* 的 *options* 参数的标志位，指定要等待的子进程信号。

Availability: Unix.

3.3 新版功能.

os.**CLD_EXITED**
os.**CLD_DUMPED**
os.**CLD_TRAPPED**
os.**CLD_CONTINUED**
waitid() 返回的结果中，si_code 的可取值。

Availability: Unix.

3.3 新版功能.

os.**waitpid**(*pid*, *options*)
本函数的细节在 Unix 和 Windows 上有不同之处。

在 Unix 上：等待进程号为 *pid* 的子进程执行完毕，返回一个元组，内含其进程 ID 和退出状态指示（编码与*wait()* 相同）。调用的语义受整数 *options* 的影响，常规操作下该值应为 0。

如果 *pid* 大于 0，则*waitpid()* 会获取该指定进程的状态信息。如果 *pid* 为 0，则获取当前进程所在进程组中的所有子进程的状态。如果 *pid* 为 -1，则获取当前进程的子进程状态。如果 *pid* 小于 -1，则获取进程组 -pid （*pid* 的绝对值）中所有进程的状态。

当系统调用返回 -1 时，将抛出带有错误码的*OSError* 异常。

在 Windows 上：等待句柄为 *pid* 的进程执行完毕，返回一个元组，内含 *pid* 以及左移 8 位后的退出状态（移位简化了跨平台使用本函数）。小于或等于 0 的 *pid* 在 Windows 上没有特殊含义，且会抛出异常。整数值 *options* 无效。*pid* 可以指向任何 ID 已知的进程，不一定是子进程。调用*spawn** 函数时传入*P_NOWAIT* 将返回合适的进程句柄。

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

os.**wait3**(*options*)
与*waitpid()* 相似，差别在于没有进程 ID 参数，且返回一个 3 元组，其中包括子进程 ID，退出状态指示和资源使用信息。关于资源使用信息的详情，请参考*resource*.*getrusage()*。option 参数与传入*waitpid()* 和*wait4()* 的相同。

Availability: Unix.

os.**wait4**(*pid*, *options*)
与*waitpid()* 相似，差别在本方法返回一个 3 元组，其中包括子进程 ID，退出状态指示和资源使用信息。关于资源使用信息的详情，请参考*resource*.*getrusage()*。*wait4()* 的参数与*waitpid()* 的参数相同。

Availability: Unix.

os.**WNOHANG**
用于*waitpid()* 的选项，如果没有立即可用的子进程状态，则立即返回。在这种情况下，函数返回 (0, 0)。

Availability: Unix.

os.**WCONTINUED**
被作业控制停止的子进程，如果自上次报告状态以来，已继续进行，则此选项将报告这些子进程。

Availability: some Unix systems.

os.**WUNTRACED**
已停止的子进程，如果自停止以来尚未报告其当前状态，则此选项将报告这些子进程。

Availability: Unix.

下列函数采用进程状态码作为参数，状态码由*system()*、*wait()* 或*waitpid()* 返回。它们可用于确定进程上发生的操作。

os.**WCOREDUMP**(*status*)
如果为该进程生成了核心转储，返回 True，否则返回 False。

Availability: Unix.

os.**WIFCONTINUED**(*status*)
Return True if the process has been continued from a job control stop, otherwise return False.

Availability: Unix.

os.**WIFSTOPPED**(*status*)
Return True if the process has been stopped, otherwise return False.

Availability: Unix.

os.**WIFSIGNALED**(*status*)
Return True if the process exited due to a signal, otherwise return False.

Availability: Unix.

os.**WIFEXITED**(*status*)
Return True if the process exited using the *exit(2)* system call, otherwise return False.

Availability: Unix.

os.**WEXITSTATUS**(*status*)
If WIFEXITED(status) is true, return the integer parameter to the *exit(2)* system call. Otherwise, the return value is meaningless.

Availability: Unix.

os.**WSTOPSIG**(*status*)
返回导致进程停止的信号。

Availability: Unix.

os.**WTERMSIG**(*status*)
Return the signal which caused the process to exit.

Availability: Unix.

16.1.7 调度器接口

这些函数控制操作系统如何为进程分配 CPU 时间。它们仅在某些 Unix 平台上可用。更多细节信息请查阅你所用 Unix 的指南页面。

3.3 新版功能.

以下调度策略如果被操作系统支持就会对外公开。

os.**SCHED_OTHER**
默认调度策略。

os.**SCHED_BATCH**
用于 CPU 密集型进程的调度策略，它会尽量为计算机中的其余任务保留交互性。

os.**SCHED_IDLE**
用于极低优先级的后台任务的调度策略。

os.**SCHED_SPORADIC**
用于偶发型服务程序的调度策略。

os.**SCHED_FIFO**
先进先出的调度策略。

os.**SCHED_RR**
循环式的调度策略。

os.**SCHED_RESET_ON_FORK**
此旗标可与任何其他调度策略进行 OR 运算。当带有此旗标的进程设置分叉时，其子进程的调度策略和优先级会被重置为默认值。

class os.**sched_param**(*sched_priority*)
这个类表示在*sched_setparam()*, *sched_setscheduler()* 和*sched_getparam()* 中使用的可修改调度形参。它属于不可变对象。

目前它只有一个可能的形参：

sched_priority
一个调度策略的调度优先级。

os.**sched_get_priority_min**(*policy*)
获取 *policy* 的最低优先级数值。*policy* 是以上调度策略常量之一。

os.**sched_get_priority_max**(*policy*)
获取 *policy* 的最高优先级数值。*policy* 是以上调度策略常量之一。

os.**sched_setscheduler**(*pid*, *policy*, *param*)
根据进程的 PID *pid* 设置其调度策略。*pid* 为 0 指的是调用本方法的进程。*policy* 是以上调度策略常量之一。*param* 是一个*sched_param* 实例。

os.**sched_getscheduler**(*pid*)
返回 PID 为 *pid* 的进程的调度策略。*pid* 为 0 指的是调用本方法的进程。返回的结果是以上调度策略常量之一。

os.**sched_setparam**(*pid*, *param*)
设置 PID 为 *pid* 的进程的某个调度参数。*pid* 为 0 指的是调用本方法的进程。*param* 是一个*sched_param* 实例。

os.**sched_getparam**(*pid*)
返回 PID 为 *pid* 的进程的调度参数为一个*sched_param* 实例。*pid* 为 0 指的是调用本方法的进程。

os.**sched_rr_get_interval**(*pid*)
返回 PID 为 *pid* 的进程在时间片轮转调度下的时间片长度（单位为秒）。*pid* 为 0 指的是调用本方法的进程。

os.**sched_yield**()
自愿放弃 CPU。

os.**sched_setaffinity**(*pid*, *mask*)
将 PID 为 *pid* 的进程（为零则为当前进程）限制到一组 CPU 上。*mask* 是整数的可迭代对象，表示应将进程限制在其中的一组 CPU。

os.**sched_getaffinity**(*pid*)
返回 PID 为 *pid* 的进程（为零则为当前进程）被限制到的那一组 CPU。

16.1.8 其他系统信息

os.**confstr**(*name*)
返回字符串格式的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 `confstr_names` 字典的键中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 指定的配置值未定义，返回 `None`。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 `confstr_names`，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

Availability: Unix.

os.**confstr_names**
字典，表示映射关系，为*confstr()* 可接受名称与操作系统为这些名称定义的整数值之间的映射。这可用于判断系统已定义了哪些名称。

Availability: Unix.

os.**cpu_count**()
返回系统的 CPU 数量。不确定则返回 `None`。

该数量不同于当前进程可以使用的 CPU 数量。可用的 CPU 数量可以由 `len(os.sched_getaffinity(0))` 方法获得。

3.4 新版功能.

os.**getloadavg**()
返回系统运行队列中最近 1、5 和 15 分钟内的平均进程数。无法获得平均负载则抛出*OSError* 异常。

Availability: Unix.

os.**sysconf**(*name*)
返回整数格式的系统配置信息。如果 *name* 指定的配置值未定义，返回 `-1`。对*confstr()* 的 *name* 参数的注释在此处也适用。当前已知的配置名称在 `sysconf_names` 字典中提供。

Availability: Unix.

os.**sysconf_names**
字典，表示映射关系，为*sysconf()* 可接受名称与操作系统为这些名称定义的整数值之间的映射。这可用于判断系统已定义了哪些名称。

Availability: Unix.

以下数据值用于支持对路径本身的操作。所有平台都有定义。

对路径的高级操作在*os.path* 模块中定义。

os.**curdir**
操作系统用来表示当前目录的常量字符串。在 Windows 和 POSIX 上是 `'.'`。在*os.path* 中也可用。

os.**pardir**
操作系统用来表示父目录的常量字符串。在 Windows 和 POSIX 上是 `'..'`。在*os.path* 中也可用。

os.**sep**
操作系统用来分隔路径不同部分的字符。在 POSIX 上是 `'/'`，在 Windows 上是是 `'\\'`。注意，仅了解它不足以能解析或连接路径，请使用*os.path.split()* 和*os.path.join()*，但它有时是有用的。在*os.path* 中也可用。

os.**altsep**
操作系统用来分隔路径不同部分的替代字符。如果仅存在一个分隔符，则为 None。在 sep 是反斜杠的 Windows 系统上，该值被设为 '/'。在*os.path* 中也可用。

os.**extsep**
分隔基本文件名与扩展名的字符，如 os.py 中的 '.'。在*os.path* 中也可用。

os.**pathsep**
操作系统通常用于分隔搜索路径（如 PATH）中不同部分的字符，如 POSIX 上是 ':'，Windows 上是 ';'。在*os.path* 中也可用。

os.**defpath**
在环境变量没有 'PATH' 键的情况下，*exec*p** and *spawn*p** 使用的默认搜索路径。在*os.path* 中也可用。

os.**linesep**
当前平台用于分隔（或终止）行的字符串。它可以是单个字符，如 POSIX 上是 '\n'，也可以是多个字符，如 Windows 上是 '\r\n'。在写入以文本模式（默认模式）打开的文件时，请不要使用 *os.linesep* 作为行终止符，请在所有平台上都使用一个 '\n' 代替。

os.**devnull**
空设备的文件路径。如 POSIX 上为 '/dev/null'，Windows 上为 'nul'。在*os.path* 中也可用。

os.**RTLD_LAZY**
os.**RTLD_NOW**
os.**RTLD_GLOBAL**
os.**RTLD_LOCAL**
os.**RTLD_NODELETE**
os.**RTLD_NOLOAD**
os.**RTLD_DEEPBIND**
setdlopenflags() 和*getdlopenflags()* 函数所使用的标志。请参阅 Unix 手册页 *dlopen(3)* 获取不同标志的含义。

3.3 新版功能.

16.1.9 随机数

os.**getrandom**(*size*, *flags=0*)
获得最多为 *size* 的随机字节。本函数返回的字节数可能少于请求的字节数。

这些字节可用于为用户空间的随机数生成器提供种子，或用于加密目的。

getrandom() 依赖于从设备驱动程序和其他环境噪声源收集的熵。不必要地读取大量数据将对使用 /dev/random 和 /dev/urandom 设备的其他用户产生负面影响。

flags 参数是一个位掩码，可以是零个或多个下列值以或运算组合：*os.GRND_RANDOM* 和*GRND_NONBLOCK*。

另请参阅 Linux getrandom() 手册页 。

Availability: Linux 3.17 and newer.

3.6 新版功能.

os.**urandom**(*size*)
返回大小为 *size* 的字符串，它是适合加密使用的随机字节。

本函数从系统指定的随机源获取随机字节。对于加密应用程序，返回的数据应有足够的不可预测性，尽管其确切的品质取决于操作系统的实现。

在 Linux 上，如果 `getrandom()` 系统调用可用，它将以阻塞模式使用：阻塞直到系统的 urandom 熵池初始化完毕（内核收集了 128 位熵）。原理请参阅 **PEP 524**。在 Linux 上，*getrandom()* 可以以非阻塞模式使用（使用*GRND_NONBLOCK* 标志）来获取随机字节，或者轮询直到系统 urandom 熵池初始化完毕。

在类 Unix 系统上，随机字节是从 `/dev/urandom` 设备读取的。如果 `/dev/urandom` 设备不可用或不可读，则抛出*NotImplementedError* 异常。

在 Windows 上将使用 `CryptGenRandom()`。

参见:

secrets 模块提供了更高级的功能。所在平台会提供随机数生成器，有关其易于使用的接口，请参阅*random.SystemRandom*。

在 3.6.0 版更改: 在 Linux 上，`getrandom()` 现在以阻塞模式使用，以提高安全性。

在 3.5.2 版更改: 在 Linux 上，如果 `getrandom()` 系统调用阻塞（urandom 熵池尚未初始化完毕），则退回一步读取 `/dev/urandom`。

在 3.5 版更改: 在 Linux 3.17 和更高版本上，现在使用 `getrandom()` 系统调用（如果可用）。在 OpenBSD 5.6 和更高版本上，现在使用 `getentropy()` C 函数。这些函数避免了使用内部文件描述符。

`os.`**`GRND_NONBLOCK`**

默认情况下，从 `/dev/random` 读取时，如果没有可用的随机字节，则*getrandom()* 会阻塞；从 `/dev/urandom` 读取时，如果熵池尚未初始化，则会阻塞。

如果设置了*GRND_NONBLOCK* 标志，则这些情况下*getrandom()* 不会阻塞，而是立即抛出*BlockingIOError* 异常。

3.6 新版功能.

`os.`**`GRND_RANDOM`**

如果设置了此标志位，那么将从 `/dev/random` 池而不是 `/dev/urandom` 池中提取随机字节。

3.6 新版功能.

16.2 io —处理流的核心工具

源代码: Lib/io.py

16.2.1 概述

io 模块提供了 Python 用于处理各种 I/O 类型的主要工具。三种主要的 I/O 类型分别为: *文本 I/O*, *二进制 I/O* 和 *原始 I/O*。这些是泛型类型，有很多种后端存储可以用在他们上面。一个隶属于任何这些类型的具体对象被称作*file object*。其他同类的术语还有 *流*和 *类文件对象*。

独立于其类别，每个具体流对象也将具有各种功能：它可以是只读，只写或读写。它还可以允许任意随机访问（向前或向后寻找任何位置），或仅允许顺序访问（例如在套接字或管道的情况下）。

All streams are careful about the type of data you give to them. For example giving a *str* object to the `write()` method of a binary stream will raise a `TypeError`. So will giving a *bytes* object to the `write()` method of a text stream.

在 3.3 版更改: 由于*IOError* 现在是*OSError* 的别名，因此用于引发*IOError* 的操作现在会引发*OSError*。

文本 I/O

文本 I/O 预期并生成*str* 对象。这意味着，无论何时后台存储是由字节组成的（例如在文件的情况下），数据的编码和解码都是透明的，并且可以选择转换特定于平台的换行符。

创建文本流的最简单方法是使用*open()*，可以选择指定编码：

```
f = open("myfile.txt", "r", encoding="utf-8")
```

内存中文本流也可以作为*StringIO* 对象使用：

```
f = io.StringIO("some initial text data")
```

TextIOBase 的文档中详细描述了文本流的 API

二进制 I/O

二进制 I/O（也称为缓冲 I/O）预期*bytes-like objects* 并产生*bytes* 对象。不执行编码、解码或换行转换。这种类型的流可以用于所有类型的非文本数据，并且还可以在需要手动控制文本数据的处理时使用。

创建二进制流的最简单方法是使用*open()*，并在模式字符串中指定 `'b'`：

```
f = open("myfile.jpg", "rb")
```

内存中二进制流也可以作为*BytesIO* 对象使用：

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

BufferedIOBase 的文档中详细描述了二进制流 API。

其他库模块可以提供额外的方式来创建文本或二进制流。参见*socket.socket.makefile()* 的示例。

原始 I/O

原始 I/O（也称为 *非缓冲 I/O*）通常用作二进制和文本流的低级构建块。用户代码直接操作原始流的用法非常罕见。不过，可以通过在禁用缓冲的情况下以二进制模式打开文件来创建原始流：

```
f = open("myfile.jpg", "rb", buffering=0)
```

The raw stream API is described in detail in the docs of *RawIOBase*.

16.2.2 高阶模块接口

io.DEFAULT_BUFFER_SIZE
包含模块缓冲 I/O 类使用的默认缓冲区大小的 int。在可能的情况下*open()* 将使用文件的 blksize（由*os.stat()* 获得）。

io.open(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)
这是内置的*open()* 函数的别名。

exception io.BlockingIOError
这是内置的*BlockingIOError* 异常的兼容性别名。

exception io.UnsupportedOperation
在流上调用不支持的操作时引发的继承*OSError* 和*ValueError* 的异常。

内存中的流

也可以使用*str* 或*bytes-like object* 作为文件进行读取和写入。对于字符串，*StringIO* 可以像在文本模式下打开的文件一样使用。*BytesIO* 可以像以二进制模式打开的文件一样使用。两者都提供完整的随机读写功能。

参见:

sys 包含标准 IO 流: *sys.stdin*, *sys.stdout* 和*sys.stderr* 。

16.2.3 类的层次结构

I/O 流的实现被组织为类的层次结构。首先是抽象基类 (ABC)，用于指定流的各种类别，然后是提供标准流实现的具体类。

> **注解:** 抽象基类还提供某些方法的默认实现，以帮助实现具体的流类。例如*BufferedIOBase* 提供了 `readinto()` 和*readline()* 的未优化实现。

I/O 层次结构的顶部是抽象基类*IOBase* 。它定义了流的基本接口。但是请注意，对流的读取和写入之间没有分隔。如果实现不支持指定的操作，则会引发*UnsupportedOperation* 。

RawIOBase ABC 是*IOBase* 的子类。它负责将字节读取和写入流中。*RawIOBase* 的子类*FileIO* 提供计算机文件系统中文件的接口。

BufferedIOBase ABC 处理原始字节流 (*RawIOBase*) 上的缓冲。其子类*BufferedWriter*、*BufferedReader* 和*BufferedRWPair* 缓冲流是可读、可写以及可读写的。*BufferedRandom* 为随机访问流提供缓冲接口。*BufferedIOBase* 的另一个子类*BytesIO* 是内存中字节流。

TextIOBase ABC 是*IOBase* 的另一个子类，它处理字节表示文本的流，并处理字符串之间的编码和解码。*TextIOWrapper* 是其一个扩展，它是原始缓冲流 (*BufferedIOBase*) 的缓冲文本接口。最后，*StringIO* 是一个用于文本的内存流。

参数名不是规范的一部分，只有*open()* 的参数才用作关键字参数。

下表总结了抽象基类提供的*io* 模块:

抽象基类	继承	Stub 方法	Mixin 方法和属性
IOBase		`fileno`, `seek`, 和 `truncate`	`close`, `closed`, `__enter__`, `__exit__`, `flush`, `isatty`, `__iter__`, `__next__`, `readable`, `readline`, `readlines`, `seekable`, `tell`, `writable`, 和 `writelines`
RawIOBase	*IOBase*	`readinto` 和 `write`	继承*IOBase* 方法, `read`, 和 `readall`
BufferedIOBase	*IOBase*	`detach`, `read`, `read1`, 和 `write`	继承*IOBase* 方法, `readinto`, 和 `readinto1`
TextIOBase	*IOBase*	`detach`, `read`, `readline`, 和 `write`	继承*IOBase* 方法, `encoding`, `errors`, 和 `newlines`

I/O 基类

class io.**IOBase**

所有 I/O 类的抽象基类，作用于字节流。没有公共构造函数。

此类为许多方法提供了空的抽象实现，派生类可以有选择地重写。默认实现无法读取、写入或查找的文件。

Even though *IOBase* does not declare `read()`, `readinto()`, or `write()` because their signatures will vary, implementations and clients should consider those methods part of the interface. Also, implementations may raise a *ValueError* (or *UnsupportedOperation*) when operations they do not support are called.

The basic type used for binary data read from or written to a file is *bytes*. Other *bytes-like objects* are accepted as method arguments too. In some cases, such as *readinto()*, a writable object such as *bytearray* is required. Text I/O classes work with *str* data.

请注意，在关闭的流上调用任何方法（甚至查询）都是未定义的（undefined）。在这种情况下，实现可能会引发*ValueError* 。

IOBase （及其子类）支持迭代器协议，这意味着可以迭代*IOBase* 对象以产生流中的行。根据流是二进制流（产生字节）还是文本流（产生字符串），行的定义略有不同。请参见下面的*readline()* 。

IOBase is also a context manager and therefore supports the `with` statement. In this example, *file* is closed after the `with` statement's suite is finished—even if an exception occurs:

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

IOBase 提供以下数据属性和方法：

close()

刷新并关闭此流。如果文件已经关闭，则此方法无效。文件关闭后，对文件的任何操作（例如读取或写入）都会引发*ValueError* 。

为方便起见，允许多次调用此方法。但是，只有第一个调用才会生效。

closed

如果流关闭，则为 True。

fileno()

返回流的基础文件描述符（整数）—如果存在。如果 IO 对象不使用文件描述符，则会引发*OSError* 。

flush()

刷新流的写入缓冲区（如果适用）。这对只读和非阻塞流不起作用。

isatty()

如果流是交互式的（即连接到终端/tty 设备），则返回 `True` 。

readable()

如果可以读取流，则返回 `True` 。否则为 `False` ，且 `read()` 将引发*OSError* 错误。

readline(*size=-1*)

从流中读取并返回一行。如果指定了 *size*，将至多读取 *size* 个字节。

对于二进制文件行结束符总是 `b'\n'`；对于文本文件，可以用将 *newline* 参数传给*open()* 的方式来选择要识别的行结束符。

readlines(*hint=-1*)

从流中读取并返回包含多行的列表。可以指定 *hint* 来控制要读取的行数：如果（以字节/字符数表示的）所有行的总大小超出了 *hint* 则将不会读取更多的行。

请注意使用 `for line in file: ...` 就足够对文件对象进行迭代了，可以不必调用 `file.readlines()`。

seek(*offset*[, *whence*])
将流位置修改到给定的字节 *offset*。*offset* 将相对于由 *whence* 指定的位置进行解析。*whence* 的默认值为 `SEEK_SET`。*whence* 的可用值有:

- `SEEK_SET` 或 `0` –流的开头（默认值）；*offset* 应为零或正值
- `SEEK_CUR` or `1` –当前流位置；*offset* 可以为负值
- `SEEK_END` or `2` –流的末尾；*offset* 通常为负值

返回新的绝对位置。

3.1 新版功能: `SEEK_*` 常量.

3.3 新版功能: 某些操作系统还可支持其他的值，例如 `os.SEEK_HOLE` 或 `os.SEEK_DATA`。特定文件的可用值还会取决于它是以文本还是二进制模式打开。

seekable()
如果流支持随机访问则返回 `True`。如为 `False`，则*seek()*, *tell()* 和*truncate()* 将引发*OSError*。

tell()
返回当前流的位置。

truncate(*size=None*)
将流的大小调整为给定的 *size* 个字节（如果未指定 *size* 则调整至当前位置）。当前的流位置不变。这个调整操作可扩展或减小当前文件大小。在扩展的情况下，新文件区域的内容取决于具体平台（在大多数系统上，额外的字节会填充为零）。返回新的文件大小。

在 3.5 版更改: 现在 Windows 在扩展时将文件填充为零。

writable()
如果流支持写入则返回 `True`。如为 `False`，则 `write()` 和*truncate()* 将引发*OSError*。

writelines(*lines*)
将行列表写入到流。不会添加行分隔符，因此通常所提供的每一行都带有末尾行分隔符。

__del__()
为对象销毁进行准备。*IOBase* 提供了此方法的默认实现，该实现会调用实例的*close()* 方法。

class io.**RawIOBase**
原始二进制 I/O 的基类。它继承自*IOBase*。没有公共构造器。

原始二进制 I/O 通常提供对下层 OS 设备或 API 的低层级访问，而不尝试将其封装到高层级的基元中（这是留给缓冲 I/O 和 Text I/O 的，将在下文中描述）。

在*IOBase* 的属性和方法之外，*RawIOBase* 还提供了下列方法:

read(*size=-1*)
从对象中读取 *size* 个字节并将其返回。作为一个便捷选项，如果 *size* 未指定或为 -1，则返回所有字节直到 EOF。在其他情况下，仅会执行一次系统调用。如果操作系统调用返回字节数少于 *size* 则此方法也可能返回少于 *size* 个字节。

如果返回 0 个字节而 *size* 不为零 0，这表明到达文件末尾。如果处于非阻塞模式并且没有更多字节可用，则返回 `None`。

默认实现会转至*readall()* 和*readinto()*。

readall()
从流中读取并返回所有字节直到 EOF，如有必要将对流执行多次调用。

readinto(*b*)
Read bytes into a pre-allocated, writable *bytes-like object* *b*, and return the number of bytes read. If the object is in non-blocking mode and no bytes are available, `None` is returned.

write(*b*)
将给定的*bytes-like object* *b* 写入到下层的原始流，并返回所写入的字节数。这可以少于 *b* 的总字节数，具体取决于下层原始流的设定，特别是如果它处于非阻塞模式的话。如果原始流设为非阻塞并且不能真正向其写入单个字节时则返回 `None`。调用者可以在此方法返回后释放或改变 *b*，因此该实现应该仅在方法调用期间访问 *b*。

class io.**BufferedIOBase**
支持某种缓冲的二进制流的基类。它继承自*IOBase*。没有公共构造器。

与*RawIOBase* 的主要差别在于*read()*, *readinto()* 和*write()* 等方法将（分别）尝试按照要求读取尽可能多的输入或是耗尽所有给定的输出，其代价是可能会执行一次以上的系统调用。

除此之外，那些方法还可能引发*BlockingIOError*，如果下层的原始数据流处于非阻塞模式并且无法接受或给出足够数据的话；不同于对应的*RawIOBase* 方法，它们将永远不会返回 `None`。

并且，*read()* 方法也没有转向*readinto()* 的默认实现。

典型的*BufferedIOBase* 实现不应当继承自*RawIOBase* 实现，而要包装一个该实现，正如*BufferedWriter* 和*BufferedReader* 所做的那样。

BufferedIOBase 在*IOBase* 的现有成员以外还提供或重载了下列方法和属性:

raw
由*BufferedIOBase* 处理的下层原始流 (*RawIOBase* 的实例)。它不是*BufferedIOBase* API 的组成部分并且不存在于某些实现中。

detach()
从缓冲区分离出下层原始流并将其返回。

在原始流被分离之后，缓冲区将处于不可用的状态。

某些缓冲区例如*BytesIO* 并无可从此方法返回的单独原始流的概念。它们将会引发*UnsupportedOperation*。

3.1 新版功能.

read(*size=-1*)
读取并返回最多 *size* 个字节。如果此参数被省略、为 `None` 或为负值，则读取并返回所有数据直到 EOF。如果流已经到达 EOF 则返回一个空的*bytes* 对象。

如果此参数为正值，并且下层原始流不可交互，则可能发起多个原始读取以满足字节计数（直至先遇到 EOF）。但对于可交互原始流，则将至多发起一个原始读取，并且简短的结果并不意味着已到达 EOF。

BlockingIOError 会在下层原始流不处于阻塞模式，并且当前没有可用数据时被引发。

read1(*size=-1*)
通过至多一次对下层流的*read()* (或*readinto()*) 方法的调用读取并返回至多 *size* 个字节。这适用于在*BufferedIOBase* 对象之上实现你自己的缓冲区的情况。

readinto(*b*)
Read bytes into a pre-allocated, writable *bytes-like object* *b* and return the number of bytes read.

类似于*read()*，可能对下层原始流发起多次读取，除非后者为交互式。

BlockingIOError 会在下层原始流不处于阻塞模式，并且当前没有可用数据时被引发。

readinto1(*b*)
将字节数据读入预先分配的可写*bytes-like object* *b*，其中至多使用一次对下层原始流*read()* (或*readinto()*) 方法的调用。返回所读取的字节数。

BlockingIOError 会在下层原始流不处于阻塞模式，并且当前没有可用数据时被引发。

3.5 新版功能.

write(*b*)

写入给定的*bytes-like object* *b*，并返回写入的字节数 (总是等于 *b* 的字节长度，因为如果写入失败则会引发*OSError*)。根据具体实现的不同，这些字节可能被实际写入下层流，或是出于运行效率和冗余等考虑而暂存于缓冲区。

当处于非阻塞模式时，如果需要将数据写入原始流但它无法在不阻塞的情况下接受所有数据则将引发*BlockingIOError*。

调用者可能会在此方法返回后释放或改变 *b*，因此该实现应当仅在方法调用期间访问 *b*。

原始文件 I/O

class io.**FileIO**(*name*, *mode='r'*, *closefd=True*, *opener=None*)

FileIO 代表在 OS 层级上包含文件的字节数据。它实现了*RawIOBase* 接口（因而也实现了*IOBase* 接口）。

name 可以是以下两项之一:

- 代表将被打开的文件路径的字符串或*bytes* 对象。在此情况下 closefd 必须为 `True` (默认值) 否则将会引发异常。
- 代表一个现有 OS 层级文件描述符的号码的整数，作为结果的*FileIO* 对象将可访问该文件。当 FileIO 对象被关闭时此 fd 也将被关闭，除非 *closefd* 设为 `False`。

mode 可以为 `'r'`, `'w'`, `'x'` 或 `'a'` 分别表示读取（默认模式）、写入、独占新建或添加。如果以写入或添加模式打开的文件不存在将自动新建；当以写入模式打开时文件将先清空。以新建模式打开时如果文件已存在则将引发*FileExistsError*。以新建模式打开文件也意味着要写入，因此该模式的行为与 `'w'` 类似。在模式中附带 `'+'` 将允许同时读取和写入。

该类的 `read()` (当附带正值参数调用时), `readinto()` 和 `write()` 方法将只执行一次系统调用。

可以通过传入一个可调用对象作为 *opener* 来使用自定义文件打开器。然后通过调用 *opener* 并传入 (*name*, *flags*) 来获取文件对象所对应的下层文件描述符。*opener* 必须返回一个打开文件描述符（传入*os.open* 作为 *opener* 的结果在功能上将与传入 `None` 类似）。

新创建的文件是*不可继承的*。

有关 opener 参数的示例，请参见内置函数*open()* 。

在 3.3 版更改: 增加了 *opener* 参数。增加了 `'x'` 模式。

在 3.4 版更改: 文件现在禁止继承。

在*IOBase* 和*RawIOBase* 的属性和方法之外，*FileIO* 还提供了下列数据属性:

mode

构造函数中给定的模式。

name

文件名。当构造函数中没有给定名称时，这是文件的文件描述符。

缓冲流

相比原始 I/O，缓冲 I/O 流提供了针对 I/O 设备的更高层级接口.

class io.**BytesIO**([*initial_bytes*])
一个使用内存字节缓冲区的流实现。它继承自*BufferedIOBase*。在*close()* 方法被调用时将会丢弃缓冲区。

可选参数 *initial_bytes* 是一个包含初始数据的*bytes-like object*。

BytesIO 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重载了下列方法:

getbuffer()
返回一个对应于缓冲区内容的可读写视图而不必拷贝其数据。此外，改变视图将透明地更新缓冲区内容:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

注解: 只要视图保持存在，*BytesIO* 对象就无法被改变大小或关闭。

3.2 新版功能.

getvalue()
返回包含整个缓冲区内容的*bytes*。

read1()
In *BytesIO*, this is the same as `read()`.

readinto1()
In *BytesIO*, this is the same as `readinto()`.

3.5 新版功能.

class io.**BufferedReader**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
一个提供对可读的序列型*RawIOBase* 对象更高层级访问的缓冲区。它继承自*BufferedIOBase*。当从此对象读取数据时，可能会从下层原始流请求更大量的数据，并存放到内部缓冲区中。接下来可以在后续读取时直接返回缓冲数据。

根据给定的可读 *raw* 流和 *buffer_size* 创建*BufferedReader* 的构造器。如果省略 *buffer_size*，则会使用*DEFAULT_BUFFER_SIZE*。

BufferedReader 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重载了下列方法:

peek([*size*])
从流返回字节数据而不前移位置。完成此调用将至多读取一次原始流。返回的字节数量可能少于或多于请求的数量。

read([*size*])
读取并返回 *size* 个字节，如果 *size* 未给定或为负值，则读取至 EOF 或是在非阻塞模式下读取调用将会阻塞。

read1(*size*)
在原始流上通过单次调用读取并返回至多 *size* 个字节。如果至少缓冲了一个字节，则只返回缓冲的字节。在其他情况下，将执行一次原始流读取。

class io.**BufferedWriter**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

一个提供对可读的序列型*RawIOBase* 对象更高层级访问的缓冲区。它继承自*BufferedIOBase*。当写入到此对象时，数据通常会被放入到内部缓冲区中。缓冲区将在满足某些条件的情况下被写到下层的*RawIOBase* 对象，包括:

- 当缓冲区对于所有挂起数据而言太小时;
- 当*flush()* 被调用时
- 当（为*BufferedRandom* 对象）请求 seek() 时;
- 当*BufferedWriter* 对象被关闭或销毁时。

该构造器会为给定的可写 *raw* 流创建一个*BufferedWriter*。如果未给定 *buffer_size*，则使用默认的*DEFAULT_BUFFER_SIZE*。

BufferedWriter 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重载了下列方法:

flush()

将缓冲区中保存的字节数据强制放入原始流。如果原始流发生阻塞则应当引发*BlockingIOError*。

write(*b*)

写入*bytes-like object* *b* 并返回写入的字节数。当处于非阻塞模式时，如果缓冲区需要被写入但原始流发生阻塞则将引发*BlockingIOError*。

class io.**BufferedRandom**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)

随机访问流的带缓冲的接口。它继承自*BufferedReader* 和*BufferedWriter*，并进一步支持 seek() 和 tell() 功能。

该构造器会为在第一个参数中给定的可查找原始流创建一个读取器和定稿器。如果省略 *buffer_size* 则使用默认的*DEFAULT_BUFFER_SIZE*。

BufferedRandom 能做到*BufferedReader* 或*BufferedWriter* 所能做的任何事。

class io.**BufferedRWPair**(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*)

一个带缓冲的 I/O 对象，它将两个单向*RawIOBase* 对象一一个可读，另一个可写–组合为单个双向端点。它继承自*BufferedIOBase*。

reader 和 *writer* 分别是可读和可写的*RawIOBase* 对象。如果省略 *buffer_size* 则使用默认的*DEFAULT_BUFFER_SIZE*。

BufferedRWPair 实现了*BufferedIOBase* 的所有方法，但*detach()* 除外，调用该方法将引发*UnsupportedOperation*。

> **警告:** *BufferedRWPair* 不会尝试同步访问其下层的原始流。你不应当将传给它与读取器和写入器相同的对象；而要改用*BufferedRandom*。

文本 I/O

class io.**TextIOBase**

Base class for text streams. This class provides a character and line based interface to stream I/O. There is no readinto() method because Python’s character strings are immutable. It inherits *IOBase*. There is no public constructor.

TextIOBase 在来自*IOBase* 的成员以外还提供或重载了以下数据属性和方法:

encoding

用于将流的字节串解码为字符串以及将字符串编码为字节串的编码格式名称。

errors
解码器或编码器的错误设置。

newlines
一个字符串、字符串元组或者 None，表示目前已经转写的新行。根据具体实现和初始构造器旗标的不同，此属性或许会不可用。

buffer
由 *TextIOBase* 处理的下层二进制缓冲区（为一个 *BufferedIOBase* 的实例）。它不是 *TextIOBase* API 的组成部分并且不存在于某些实现中。

detach()
从 *TextIOBase* 分离出下层二进制缓冲区并将其返回。

在下层缓冲区被分离后，*TextIOBase* 将处于不可用的状态。

某些 *TextIOBase* 的实现，例如 *StringIO* 可能并无下层缓冲区的概念，因此调用此方法将引发 *UnsupportedOperation*。

3.1 新版功能.

read(*size=-1*)
从流中读取至多 *size* 个字符并以单个 *str* 的形式返回。如果 *size* 为负值或 None，则读取至 EOF。

readline(*size=-1*)
读取至换行符或 EOF 并返回单个 str。如果流已经到达 EOF，则将返回一个空字符串。

如果指定了 *size*，最多将读取 *size* 个字符。

seek(*offset*[, *whence*])
将流位置改为给定的偏移位置 *offset*。具体行为取决于 *whence* 形参。*whence* 的默认值为 SEEK_SET。

- SEEK_SET 或 0: 从流的开始位置起查找（默认值）；*offset* 必须为 *TextIOBase.tell()* 所返回的数值或为零。任何其他 *offset* 值都将导致未定义的行为。
- SEEK_CUR 或 1: “查找”到当前位置；*offset* 必须为零，表示无操作（所有其他值均不受支持）。
- SEEK_END 或 2: 查找到流的末尾；*offset* 必须为零（所有其他值均不受支持）。

以数字形式返回新的绝对位置。

3.1 新版功能: SEEK_* 常量.

tell()
以不透明数字形式返回当前流的位置。该数字通常并不代表下层二进制存储中对应的字节数。

write(*s*)
将字符串 *s* 写入到流并返回写入的字符数。

class io.**TextIOWrapper**(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)
一个基于 *BufferedIOBase* 二进制流的缓冲文本流。它继承自 *TextIOBase*。

encoding 给出流被解码或编码时将使用的编码格式。它默认为 *locale.getpreferredencoding(False)*。

errors 是一个可选的字符串，它指明编码格式和编码格式错误的处理方式。传入 'strict' 将在出现编码格式错误时引发 *ValueError*（默认值 None 具有相同的效果），传入 'ignore' 将忽略错误。（请注意忽略编码格式错误会导致数据丢失。）'replace' 会在出现错误数据时插入一个替换标记（例如 '?'）。'backslashreplace' 将把错误数据替换为一个反斜杠转义序列。在写入时，还可以使用 'xmlcharrefreplace'（替换为适当的 XML 字符引用）或 'namereplace'（替换为 \N{...} 转义序列）。任何其他通过 *codecs.register_error()* 注册的错误处理方式名称也可以被接受。

newline 控制行结束符处理方式。它可以为 None, '', '\n', '\r' 和 '\r\n'。其工作原理如下:

- 当从流中读取输入时，如果 *newline* 为 None，则会启用*universal newlines* 模式。输入中的行结束符可以为 '\n', '\r' 或 '\r\n'，在返回给调用者之前它们会被统一转写为 '\n'。如果参数为 ''，也会启用通用换行模式，但行结束符会不加转写即返回给调用者。如果它具有任何其他合法的值，则输入行将仅由给定的字符串结束，并且行结束符会不加转写即返回给调用者。
- 将输出写入流时，如果 *newline* 为 None，则写入的任何 '\n' 字符都将转换为系统默认行分隔符*os.linesep*。如果 *newline* 是 '' 或 '\n'，则不进行翻译。如果 *newline* 是任何其他合法值，则写入的任何 '\n' 字符将被转换为给定的字符串。

如果 *line_buffering* 为 True，则当一个写入调用包含换行符或回车时将会应用 flush()。

如果 *write_through* 为 True，对 write() 的调用会确保不被缓冲：在*TextIOWrapper* 对象上写入的任何数据会立即交给其下层的 *buffer* 来处理。

在 3.3 版更改: 已添加 *write_through* 参数

在 3.3 版更改: 默认的 *encoding* 现在将为 locale.getpreferredencoding(False) 而非 locale.getpreferredencoding()。不要使用*locale.setlocale()* 来临时改变区域编码格式，要使用当前区域编码格式而不是用户的首选编码格式。

TextIOWrapper provides one attribute in addition to those of *TextIOBase* and its parents:

line_buffering
是否启用行缓冲。

class io.**StringIO**(*initial_value=''*, *newline='\n'*)
用于文本 I/O 的内存数据流。当调用*close()* 方法时将会丢弃文本缓冲区。

缓冲区的初始值可通过提供 *initial_value* 来设置。如果启用了行结束符转写，换行将以*write()* 所用的方式被编码。数据流位置将被设为缓冲区的开头。

newline 参数的规则与*TextIOWrapper* 所用的一致。默认规则是仅将 \n 字符视为行结束符并且不执行换行符转写。如果 *newline* 设为 None，在所有平台上换行符都将被写入为 \n，但当读取时仍然会执行通用换行编码格式。

StringIO 在继承自*TextIOBase* 及其父类的现有成员以外还提供了以下方法:

getvalue()
返回一个包含缓冲区全部内容的 str。换行符会以与*read()* 相同的方式被编码，但是流的位置不会被改变。

用法示例:

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.**IncrementalNewlineDecoder**
用于在*universal newlines* 模式下解码换行符的辅助编解码器。它继承自*codecs.IncrementalDecoder*。

16.2.4 性能

本节讨论所提供的具体 I/O 实现的性能。

二进制 I/O

即使在用户请求单个字节时，也只读取和写入大块数据。通过该方法，缓冲 I/O 也隐藏了调用和执行操作系统无缓冲 I/O 例程时的任何低效性。。增益取决于操作系统和执行的 I/O 类型。例如，在某些现代操作系统上（例如 Linux），无缓冲磁盘 I/O 可以与缓冲 I/O 一样快。但最重要的是，无论平台和支持设备如何，缓冲 I/O 都能提供可预测的性能。因此，对于二进制数据，几乎总是首选使用缓冲的 I / O 而不是未缓冲的 I / O。

文本 I/O

二进制存储（如文件）上的文本 I/O 比同一存储上的二进制 I/O 慢得多，因为它需要使用字符编解码器在 Unicode 和二进制数据之间进行转换。这在处理大量文本数据（如大型日志文件）时会变得非常明显。此外，由于使用的重构算法 `TextIOWrapper.tell()` 和 `TextIOWrapper.seek()` 都相当慢。

StringIO 是原生的内存 Unicode 容器，速度与*BytesIO* 相似。

多线程

FileIO 对象是线程安全的，只要它们包装的操作系统调用（比如 Unix 下的 `read(2)`）也是线程安全的。

二进制缓冲对象（例如*BufferedReader*, *BufferedWriter*, *BufferedRandom* 和*BufferedRWPair*）使用锁来保护其内部结构；因此，可以安全地一次从多个线程中调用它们。

TextIOWrapper 对象不再是线程安全的。

可重入性

二进制缓冲对象（*BufferedReader*，*BufferedWriter*，*BufferedRandom* 和*BufferedRWPair* 的实例）不是可重入的。虽然在正常情况下不会发生可重入调用，但是可能会在*signal* 处理程序中执行 I/O 而产生。如果线程尝试重新输入已经访问的缓冲对象，则会引发*RuntimeError* 。注意，这并不禁止其他线程进入缓冲对象。

上面的内容隐含地扩展到文本文件，因为*open()* 函数会把缓冲对象包装在*TextIOWrapper* 中。这包括标准流，因此也会影响内置函数*print()* 。

16.3 `time` —时间的访问和转换

该模块提供了各种时间相关的函数。相关功能还可以参阅*datetime* 和*calendar* 模块。

尽管此模块始终可用，但并非所有平台上都提供所有功能。此模块中定义的大多数函数是调用了所在平台 C 语言库的同名函数。因为这些函数的语义因平台而异，所以使用时最好查阅平台相关文档。

下面是一些术语和惯例的解释.

- *epoch* 是时间开始的点，并且取决于平台。对于 Unix，epoch 是 1970 年 1 月 1 日 00:00:00（UTC）。要找出给定平台上的 epoch ，请查看 `time.gmtime(0)` 。

- 术语 *Unix 纪元秒数*是指自国际标准时间 1970 年 1 月 1 日零时以来经过的总秒数，通常不包括 闰秒。在所有符合 POSIX 标准的平台上，闰秒都会从总秒数中被扣除。
- 此模块中的功能可能无法处理纪元之前或将来的远期日期和时间。未来的截止点由 C 库决定；对于 32 位系统，它通常在 2038 年。
- **Year 2000 (Y2K) issues**: Python depends on the platform's C library, which generally doesn't have year 2000 issues, since all dates and times are represented internally as seconds since the epoch. Function *strptime()* can parse 2-digit years when given `%y` format code. When 2-digit years are parsed, they are converted according to the POSIX and ISO C standards: values 69–99 are mapped to 1969–1999, and values 0–68 are mapped to 2000–2068.
- UTC 是协调世界时（以前称为格林威治标准时间，或 GMT）。缩写 UTC 不是错误，而是英语和法语之间的妥协。
- DST 是夏令时，在一年中的一部分时间（通常）调整时区一小时。DST 规则很神奇（由当地法律确定），并且每年都会发生变化。C 库有一个包含本地规则的表（通常是从系统文件中读取以获得灵活性），并且在这方面是 True Wisdom 的唯一来源。
- 各种实时函数的精度可能低于表示其值或参数的单位所建议的精度。例如，在大多数 Unix 系统上，时钟“ticks”仅为每秒 50 或 100 次。
- 另一方面，*time()* 和*sleep()* 的精度优于它们的 Unix 等价物：时间表示为浮点数，*time()* 返回最准确的时间（使用 Unix `gettimeofday()` 如果可用），并且*sleep()* 将接受非零分数的时间（Unix `select()` 用于实现此功能，如果可用）。
- 时间值由*gmtime()*，*localtime()* 和*strptime()* 返回，并被*asctime()*，*mktime()* 和*strftime()* 接受，是一个 9 个整数的序列。*gmtime()*，*localtime()* 和*strptime()* 的返回值还提供各个字段的属性名称。

  请参阅*struct_time* 以获取这些对象的描述。

  在 3.3 版更改: 在平台支持相应的 `struct tm` 成员时，*struct_time* 类型被扩展提供 `tm_gmtoff` 和 `tm_zone` 属性。

  在 3.6 版更改: *struct_time* 的属性 `tm_gmtoff` 和 `tm_zone` 现在可在所有平台上使用。
- 使用以下函数在时间表示之间进行转换：

从	到	使用
seconds since the epoch	UTC 的*struct_time*	*gmtime()*
seconds since the epoch	本地时间的*struct_time*	*localtime()*
UTC 的*struct_time*	seconds since the epoch	*calendar.timegm()*
本地时间的*struct_time*	seconds since the epoch	*mktime()*

16.3.1 函数

`time.`**`asctime`**`([t])`

转换一个元组或*struct_time* 表示的时间，由*gmtime()* 或*localtime()* 返回为以下形式的字符串：`'Sun Jun 20 23:21:05 1993'`。如果未提供 *t*，则使用由*localtime()* 返回的当前时间。区域信息不被函数*asctime()* 使用。

> **注解:** 与同名的 C 函数不同，*asctime()* 不添加尾随换行符。

`time.`**`clock`**`()`

在 Unix 上，将当前处理器时间返回为以秒为单位的浮点数。精确度，实际上是“处理器时间”含义的定义，取决于同名 C 函数的精度。

在 Windows 上，此函数返回自第一次调用此函数以来经过的 wallclock 秒数，作为浮点数，基于 Win32 函数 `QueryPerformanceCounter()`。分辨率通常优于 1 微秒。

3.3 版后已移除: 此函数的行为取决于平台: 根据你的需求，使用*perf_counter()* 或*process_time()* 获得具有明确定义的行为。

`time.`**`clock_getres`**(*clk_id*)
返回指定时钟 *clk_id* 的分辨率（精度）。有关 *clk_id* 的可接受值列表，请参阅*Clock ID 常量* 。

Availability: Unix.

3.3 新版功能.

`time.`**`clock_gettime`**(*clk_id*)
返回指定 *clk_id* 时钟的时间。有关 *clk_id* 的可接受值列表，请参阅*Clock ID 常量* 。

Availability: Unix.

3.3 新版功能.

`time.`**`clock_settime`**(*clk_id*, *time*)
设置指定 *clk_id* 时钟的时间。目前，*CLOCK_REALTIME* 是 *clk_id* 唯一可接受的值。

Availability: Unix.

3.3 新版功能.

`time.`**`ctime`**([*secs*])
将以自 epoch 开始的秒数表示的时间转换为表示本地时间的字符串。如果未提供 *secs* 或为*None*，则使用*time()* 所返回的当前时间。`ctime(secs)` 相当于 `asctime(localtime(secs))`。区域信息不会被*ctime()* 使用。

`time.`**`get_clock_info`**(*name*)
获取有关指定时钟的信息作为命名空间对象。支持的时钟名称和读取其值的相应函数是:

- `'clock'`: *time.clock()*
- `'monotonic'`: *time.monotonic()*
- `'perf_counter'`: *time.perf_counter()*
- `'process_time'`: *time.process_time()*
- `'time'`: *time.time()*

结果具有以下属性:

- *adjustable*： 如果时钟可以自动更改（例如通过 NTP 守护程序）或由系统管理员手动更改，则为 `True`，否则为 `False`。
- *implementation*： 用于获取时钟值的基础 C 函数的名称。有关可能的值，请参阅*Clock ID 常量* 。
- *monotonic*： 如果时钟不能倒退，则为 `True`，否则为 `False`。
- *resolution*： 以秒为单位的时钟分辨率（*float*）

3.3 新版功能.

`time.`**`gmtime`**([*secs*])
将以自 epoch 开始的秒数表示的时间转换为 UTC 的*struct_time*，其中 dst 标志始终为零。如果未提供 *secs* 或为*None*，则使用*time()* 所返回的当前时间。一秒以内的小数将被忽略。有关*struct_time* 对象的说明请参见上文。有关此函数的逆操作请参阅*calendar.timegm()*。

`time.`**`localtime`**([*secs*])
与*gmtime()* 相似但转换为当地时间。如果未提供 *secs* 或为*None*，则使用由*time()* 返回的当前时间。当 DST 适用于给定时间时，dst 标志设置为 `1`。

time.**mktime**(*t*)
这是*localtime()* 的反函数。它的参数是*struct_time* 或者完整的 9 元组（因为需要 dst 标志；如果它是未知的则使用 -1 作为 dst 标志），它表示 *local* 的时间，而不是 UTC 。它返回一个浮点数，以便与*time()* 兼容。如果输入值不能表示为有效时间，则*OverflowError* 或*ValueError* 将被引发（这取决于 Python 或底层 C 库是否捕获到无效值）。它可以生成时间的最早日期取决于平台。

time.**monotonic**()
返回单调时钟的值（以小数秒为单位），即不能倒退的时钟。时钟不受系统时钟更新的影响。返回值的参考点未定义，因此只有连续调用结果之间的差异才有效。

On Windows versions older than Vista, *monotonic()* detects `GetTickCount()` integer overflow (32 bits, roll-over after 49.7 days). It increases an internal epoch (reference time) by 2^{32} each time that an overflow is detected. The epoch is stored in the process-local state and so the value of *monotonic()* may be different in two Python processes running for more than 49 days. On more recent versions of Windows and on other operating systems, *monotonic()* is system-wide.

3.3 新版功能.

在 3.5 版更改: The function is now always available.

time.**perf_counter**()
返回性能计数器的值（以小数秒为单位），即具有最高可用分辨率的时钟，以测量短持续时间。它确实包括睡眠期间经过的时间，并且是系统范围的。返回值的参考点未定义，因此只有连续调用结果之间的差异才有效。

3.3 新版功能.

time.**process_time**()
返回当前进程的系统和用户 CPU 时间总和的值（以小数秒为单位）。它不包括睡眠期间经过的时间。根据定义，它在整个进程范围中。返回值的参考点未定义，因此只有连续调用结果之间的差异才有效。

3.3 新版功能.

time.**sleep**(*secs*)
暂停执行调用线程达到给定的秒数。参数可以是浮点数，以指示更精确的睡眠时间。实际的暂停时间可能小于请求的时间，因为任何捕获的信号将在执行该信号的捕获例程后终止*sleep()* 。此外，由于系统中其他活动的安排，暂停时间可能比请求的时间长任意量。

在 3.5 版更改: 即使睡眠被信号中断，该函数现在至少睡眠 *secs* ，除非信号处理程序引发异常（参见 **PEP 475** 作为基本原理）。

time.**strftime**(*format*[, *t*])
转换一个元组或*struct_time* 表示的由*gmtime()* 或*localtime()* 返回的时间到由 *format* 参数指定的字符串。如果未提供 *t* ，则使用由*localtime()* 返回的当前时间。*format* 必须是一个字符串。如果 *t* 中的任何字段超出允许范围，则引发*ValueError* 。

0 是时间元组中任何位置的合法参数；如果它通常是非法的，则该值被强制改为正确的值。

以下指令可以嵌入 *format* 字符串中。它们显示时没有可选的字段宽度和精度规范，并被*strftime()* 结果中的指示字符替换：

指令	含义	注释
%a	本地化的缩写星期中每日的名称。	
%A	本地化的星期中每日的完整名称。	
%b	本地化的月缩写名称。	
%B	本地化的月完整名称。	
%c	本地化的适当日期和时间表示。	
%d	十进制数 [01,31] 表示的月中日。	
%H	十进制数 [00,23] 表示的小时（24 小时制）。	
%I	十进制数 [01,12] 表示的小时（12 小时制）。	
%j	十进制数 [001,366] 表示的年中日。	
%m	十进制数 [01,12] 表示的月。	
%M	十进制数 [00,59] 表示的分钟。	
%p	本地化的 AM 或 PM 。	(1)
%S	十进制数 [00,61] 表示的秒。	(2)
%U	十进制数 [00,53] 表示的一年中的周数（星期日作为一周的第一天）。在第一个星期日之前的新年中的所有日子都被认为是在第 0 周。	(3)
%w	十进制数 [0(星期日),6] 表示的周中日。	
%W	十进制数 [00,53] 表示的一年中的周数（星期一作为一周的第一天）。在第一个星期一之前的新年中的所有日子被认为是在第 0 周。	(3)
%x	本地化的适当日期表示。	
%X	本地化的适当时间表示。	
%y	十进制数 [00,99] 表示的没有世纪的年份。	
%Y	十进制数表示的带世纪的年份。	
%z	时区偏移以格式 +HHMM 或 -HHMM 形式的 UTC/GMT 的正或负时差指示，其中 H 表示十进制小时数字，M 表示小数分钟数字 [-23:59, +23:59] 。	
%Z	时区名称（如果不存在时区，则不包含字符）。	
%%	字面的 '%' 字符。	

注释:

(1) 当与*strptime()* 函数一起使用时，如果使用 %I 指令来解析小时，%p 指令只影响输出小时字段。

(2) 范围真的是 0 到 61 ；值 60 在表示 leap seconds 的时间戳中有效，并且由于历史原因支持值 61 。

(3) 当与*strptime()* 函数一起使用时，%U 和 %W 仅用于指定星期几和年份的计算。

下面是一个示例，一个与 **RFC 2822** Internet 电子邮件标准以兼容的日期格式。[1]

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

某些平台可能支持其他指令，但只有此处列出的指令具有 ANSI C 标准化的含义。要查看平台支持的完整格式代码集，请参阅 *strftime(3)* 文档。

在某些平台上，可选的字段宽度和精度规范可以按照以下顺序紧跟在指令的初始 '%' 之后；这也不可移植。字段宽度通常为 2，除了 %j ，它是 3。

time.**strptime**(*string*[, *format*])

根据格式解析表示时间的字符串。返回值为一个被*gmtime()* 或*localtime()* 返回的*struct_time* 。

[1] 现在不推荐使用 %z ，但是所有 ANSI C 库都不支持扩展为首选小时/分钟偏移量的 "%z"转义符。此外，严格的 1982 年原始 **RFC 822** 标准要求两位数的年份（%y 而不是%Y），但是实际在 2000 年之前很久就转移到了 4 位数年。之后，**RFC 822** 已经废弃了，4 位数的年份首先被推荐 **RFC 1123** ，然后被 **RFC 2822** 强制执行。

format 参数使用与*strftime()* 相同的指令。它默认为匹配*ctime()* 所返回的格式 "%a %b %d %H:%M:%S %Y"` 。如果 *string* 不能根据 *format* 来解析，或者解析后它有多余的数据，则会引发*ValueError*。当无法推断出更准确的值时，用于填充任何缺失数据的默认值是 (1900, 1, 1, 0, 0, 0, 0, 1, -1) 。*string* 和 *format* 都必须为字符串。

例如:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

支持 %Z 指令是基于 tzname 中包含的值以及 daylight 是否为真。因此，它是特定于平台的，除了识别始终已知的 UTC 和 GMT （并且被认为是非夏令时时区）。

仅支持文档中指定的指令。因为每个平台都实现了 strftime() ，它有时会提供比列出的指令更多的指令。但是 strptime() 独立于任何平台，因此不一定支持所有未记录为支持的可用指令。

class time.**struct_time**

返回的时间值序列的类型为*gmtime()* 、*localtime()* 和*strptime()* 。它是一个带有*named tuple* 接口的对象：可以通过索引和属性名访问值。存在以下值:

索引	属性	值
0	tm_year	（例如，1993）
1	tm_mon	range [1, 12]
2	tm_mday	range [1, 31]
3	tm_hour	range [0, 23]
4	tm_min	range [0, 59]
5	tm_sec	range [0, 61]；见*strftime()* 介绍中的 **(2)**
6	tm_wday	range [0, 6] ，周一为 0
7	tm_yday	range [1, 366]
8	tm_isdst	0, 1 或 -1；如下所示
N/A	tm_zone	时区名称的缩写
N/A	tm_gmtoff	以秒为单位的 UTC 以东偏离

请注意，与 C 结构不同，月份值是 [1,12] 的范围，而不是 [0,11] 。

在调用*mktime()* 时，tm_isdst 可以在夏令时生效时设置为 1，而在夏令时不生效时设置为 0。值-1 表示这是未知的，并且通常会导致填写正确的状态。

当一个长度不正确的元组被传递给期望*struct_time* 的函数，或者具有错误类型的元素时，会引发*TypeError* 。

time.**time**()

返回以浮点数表示的从*epoch* 开始的秒数的时间值。epoch 的具体日期和 leap seconds 的处理取决于平台。在 Windows 和大多数 Unix 系统中，epoch 是 1970 年 1 月 1 日 00:00:00 (UTC)，并且闰秒将不计入从 epoch 开始的秒数。这通常被称为 Unix 时间。要了解给定平台上 epoch 的具体定义，请查看 gmtime(0) 。

请注意，即使时间总是作为浮点数返回，但并非所有系统都提供高于 1 秒的精度。虽然此函数通常返回非递减值，但如果在两次调用之间设置了系统时钟，则它可以返回比先前调用更低的值。

返回的数字*time()* 可以通过将其传递给*gmtime()* 函数或转换为 UTC 中更常见的时间格式（即年、月、日、小时等）或通过将它传递给*localtime()* 函数获得本地时间。在这两种情况下都返回一个*struct_time* 对象，日历日期组件可以从中作为属性访问。

time.**tzset**()

Resets the time conversion rules used by the library routines. The environment variable TZ specifies how this is

done.

Availability: Unix.

注解: 虽然在很多情况下，更改 TZ 环境变量而不调用*tzset()* 可能会影响函数的输出，例如*localtime()*，不应该依赖此行为。

TZ 不应该包含空格。

TZ 环境变量的标准格式是（为了清晰起见，添加了空格）：

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

组件的位置是：

std 和 dst 三个或更多字母数字，给出时区缩写。这些将传到 time.tzname

offset 偏移量的形式为：± hh[:mm[:ss]]。这表示添加到达 UTC 的本地时间的值。如果前面有‘-‘，则时区位于本初子午线的东边；否则，在它是西边。如果 dst 之后没有偏移，则假设夏令时比标准时间提前一小时。

start[/time], end[/time] 指示何时更改为 DST 和从 DST 返回。开始日期和结束日期的格式为以下之一：

Jn Julian 日 *n*（1 <= *n* <= 365）。闰日不计算在内，因此在所有年份中，2 月 28 日是第 59 天，3 月 1 日是第 60 天。

n 从零开始的 Julian 日（0 <= *n* <= 365）。闰日计入，可以引用 2 月 29 日。

Mm.n.d 一年中 *m* 月的第 *n* 周（1 <= *n* <= 5，1 <= *m* <= 12，第 5 周表示“可能在 *m* 月第 4 周或第 5 周出现的最后第 *d* 日”）的第 *d* 天（0 <= *d* <= 6）。第 1 周是第 *d* 天发生的第一周。第 0 天是星期天。

time 的格式与 offset 的格式相同，但不允许使用前导符号（‘-‘或‘+’）。如果没有给出时间，则默认值为 02:00:00。

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

在许多 Unix 系统（包括 *BSD，Linux，Solaris 和 Darwin 上），使用系统的区域信息（*tzfile(5)*）数据库来指定时区规则会更方便。为此，将 TZ 环境变量设置为所需时区数据文件的路径，相对于系统‘zoneinfo’时区数据库的根目录，通常位于 /usr/share/zoneinfo。例如，'US/Eastern'、'Australia/Melbourne'、'Egypt' 或 'Europe/Amsterdam'。

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 Clock ID 常量

这些常量用作*clock_getres()* 和*clock_gettime()* 的参数。

time.**CLOCK_HIGHRES**
Solaris OS 有一个 CLOCK_HIGHRES 计时器，试图使用最佳硬件源，并可能提供接近纳秒的分辨率。CLOCK_HIGHRES 是不可调节的高分辨率时钟。

Availability: Solaris.

3.3 新版功能.

time.**CLOCK_MONOTONIC**
无法设置的时钟，表示自某些未指定的起点以来的单调时间。

Availability: Unix.

3.3 新版功能.

time.**CLOCK_MONOTONIC_RAW**
类似于*CLOCK_MONOTONIC*，但可以访问不受 NTP 调整影响的原始硬件时间。

Availability: Linux 2.6.28 or later.

3.3 新版功能.

time.**CLOCK_PROCESS_CPUTIME_ID**
来自 CPU 的高分辨率每进程计时器。

Availability: Unix.

3.3 新版功能.

time.**CLOCK_THREAD_CPUTIME_ID**
特定于线程的 CPU 时钟。

Availability: Unix.

3.3 新版功能.

以下常量是唯一可以发送到*clock_settime()* 的参数。

time.**CLOCK_REALTIME**
系统范围的实时时钟。设置此时钟需要适当的权限。

Availability: Unix.

3.3 新版功能.

16.3.3 时区常量

time.**altzone**
本地 DST 时区的偏移量，以 UTC 为单位的秒数，如果已定义。如果当地 DST 时区在 UTC 以东（如在西欧，包括英国），则是负数。只有当 daylight 非零时才使用它。见下面的注释。

time.**daylight**
如果定义了 DST 时区，则为非零。见下面的注释。

time.**timezone**
本地（非 DST）时区的偏移量，UTC 以西的秒数（西欧大部分地区为负，美国为正，英国为零）。见下面的注释。

time.**tzname**
两个字符串的元组：第一个是本地非 DST 时区的名称，第二个是本地 DST 时区的名称。如果未定义 DST 时区，则不应使用第二个字符串。见下面的注释。

注解： 对于上述时区常量（*altzone*、*daylight*、*timezone* 和 *tzname*），该值由模块加载时有效的时区规则确定，或者最后一次 *tzset()* 被调用时，并且在过去的时间可能不正确。建议使用来自 *localtime()* 结果的 tm_gmtoff 和 tm_zone 来获取时区信息。

参见：

模块 *datetime* 更多面向对象的日期和时间接口。

模块 *locale* 国际化服务。区域设置会影响 *strftime()* 和 *strptime()* 中许多格式说明符的解析。

模块 *calendar* 一般日历相关功能。这个模块的 timegm() 是函数 *gmtime()* 的反函数。

备注

16.4 argparse —命令行选项、参数和子命令解析器

3.2 新版功能.

源代码： Lib/argparse.py

教程

此页面包含该 API 的参考信息。有关 Python 命令行解析更细致的介绍，请参阅 argparse 教程。

argparse 模块可以让人轻松编写用户友好的命令行接口。程序定义它需要的参数，然后 *argparse* 将弄清如何从 *sys.argv* 解析出那些参数。*argparse* 模块还会自动生成帮助和使用手册，并在用户给程序传入无效参数时报出错误信息。

16.4.1 示例

以下代码是一个 Python 程序，它获取一个整数列表并计算总和或者最大值：

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

假设上面的 Python 代码保存在名为 prog.py 的文件中，它可以在命令行运行并提供有用的帮助信息：

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

当使用适当的参数运行时，它会输出命令行传入整数的总和或者最大值：

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

如果传入无效参数，则会报出错误：

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

以下部分将引导你完成这个示例。

创建一个解析器

使用*argparse*的第一步是创建一个*ArgumentParser*对象：

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

*ArgumentParser*对象包含将命令行解析成 Python 数据类型所需的全部信息。

添加参数

给一个*ArgumentParser*添加程序参数信息是通过调用*add_argument()*方法完成的。通常，这些调用指定*ArgumentParser*如何获取命令行字符串并将其转换为对象。这些信息在*parse_args()*调用时被存储和使用。例如：

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

然后，调用*parse_args()*将返回一个具有 `integers` 和 `accumulate` 两个属性的对象。`integers` 属性将是一个包含一个或多个整数的列表，而 `accumulate` 属性当命令行中指定了 `--sum` 参数时将是*sum()*函数，否则则是*max()*函数。

解析参数

ArgumentParser 通过*parse_args()* 方法解析参数。它将检查命令行，把每个参数转换为适当的类型然后调用相应的操作。在大多数情况下，这意味着一个简单的*Namespace* 对象将从命令行解析出的属性构建：

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

在脚本中，通常*parse_args()* 会被不带参数调用，而*ArgumentParser* 将自动从*sys.argv* 中确定命令行参数。

16.4.2 ArgumentParser 对象

class argparse.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=argparse.HelpFormatter*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*)

创建一个新的*ArgumentParser* 对象。所有的参数都应当作为关键字参数传入。每个参数在下面都有它更详细的描述，但简而言之，它们是：

- *prog* - 程序的名称（默认值：sys.argv[0]）
- *usage* - 描述程序用途的字符串（默认值：从添加到解析器的参数生成）
- *description* - 在参数帮助文档之前显示的文本（默认值：无）
- *epilog* - 在参数帮助文档之后显示的文本（默认值：无）
- *parents* - 一个*ArgumentParser* 对象的列表，它们的参数也应包含在内
- *formatter_class* - 用于自定义帮助文档输出格式的类
- *prefix_chars* - 可选参数的前缀字符集合（默认值：'-'）
- *fromfile_prefix_chars* - 当需要从文件中读取其他参数时，用于标识文件名的前缀字符集合（默认值：None）
- *argument_default* - 参数的全局默认值（默认值：None）
- *conflict_handler* - 解决冲突选项的策略（通常是不必要的）
- *add_help* - 为解析器添加一个 -h/--help 选项（默认值：True）
- *allow_abbrev* - 如果缩写是无歧义的，则允许缩写长选项（默认值：True）

在 3.5 版更改: 增加了 *allow_abbrev* 参数。

以下部分描述这些参数如何使用。

prog

默认情况下，*ArgumentParser* 对象使用 sys.argv[0] 来确定如何在帮助消息中显示程序名称。这一默认值几乎总是可取的，因为它将使帮助消息与从命令行调用此程序的方式相匹配。例如，对于有如下代码的名为 myprogram.py 的文件：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

该程序的帮助信息将显示 myprogram.py 作为程序名称（无论程序从何处被调用）：

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

要更改这样的默认行为，可以使用 prog= 参数为*ArgumentParser* 指定另一个值：

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
 -h, --help  show this help message and exit
```

需要注意的是，无论是从 sys.argv[0] 或是从 prog= 参数确定的程序名称，都可以在帮助消息里通过 %(prog)s 格式说明符来引用。

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

usage

默认情况下，*ArgumentParser* 根据它包含的参数来构建用法消息：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

可以通过 usage= 关键字参数覆盖这一默认消息：

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar           bar help

optional arguments:
 -h, --help    show this help message and exit
 --foo [FOO]   foo help
```

在用法消息中可以使用 `%(prog)s` 格式说明符来填入程序名称。

description

大多数对*ArgumentParser* 构造方法的调用都会使用 `description=` 关键字参数。这个参数简要描述这个程序做什么以及怎么做。在帮助消息中，这个描述会显示在命令行用法字符串和各种参数的帮助消息之间:

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit
```

在默认情况下，description 将被换行以便适应给定的空间。如果想改变这种行为，见*formatter_class* 参数。

epilog

一些程序喜欢在 description 参数后显示额外的对程序的描述。这种文字能够通过给*ArgumentParser*:: 提供 `epilog=` 参数而被指定。

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

和*description* 参数一样，`epilog=` text 在默认情况下会换行，但是这种行为能够被调整通过提供*formatter_class* 参数给 `ArgumentParse`.

parents

有些时候，少数解析器会使用同一系列参数。单个解析器能够通过提供 parents= 参数给*ArgumentParser* 而使用相同的参数而不是重复这些参数的定义。parents= 参数使用*ArgumentParser* 对象的列表，从它们那里收集所有的位置和可选的行为，然后将这写行为加到正在构建的*ArgumentParser* 对象。

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

请注意大多数父解析器会指定 add_help=False . 否则，ArgumentParse 将会看到两个 -h/--help 选项（一个在父参数中一个在子参数中）并且产生一个错误。

注解： 你在通过 “parents=“ 传递解析器之前必须完全初始化它们。如果你在子解析器之后改变父解析器，这些改变将不会反映在子解析器上。

formatter_class

ArgumentParser 对象允许通过指定备用格式化类来自定义帮助格式。目前，有四种这样的类。

class argparse.**RawDescriptionHelpFormatter**
class argparse.**RawTextHelpFormatter**
class argparse.**ArgumentDefaultsHelpFormatter**
class argparse.**MetavarTypeHelpFormatter**

RawDescriptionHelpFormatter 和*RawTextHelpFormatter* 在正文的描述和展示上给与了更多的控制。*ArgumentParser* 对象会将*description* 和*epilog* 的文字在命令行中自动换行。

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

optional arguments:
 -h, --help  show this help message and exit
```

(下页继续)

(续上页)

```
likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

传*RawDescriptionHelpFormatter* 给 formatter_class= 表示*description* 和*epilog* 已经被正确的格式化了，不能在命令行中被自动换行:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

optional arguments:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter 保留所有种类文字的空格，包括参数的描述。然而，多重的新行会被替换成一行。如果你想保留多重的空白行，可以在新行之间加空格。

ArgumentDefaultsHelpFormatter 自动添加默认的值的信息到每一个帮助信息的参数中:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar [bar ...]]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

MetavarTypeHelpFormatter 为它的值在每一个参数中使用*type* 的参数名当作它的显示名（而不是使用通常的格式*dest*):

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
```

(下页继续)

(续上页)

```
usage: PROG [-h] [--foo int] float

positional arguments:
  float

optional arguments:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars

许多命令行会使用 - 当作前缀，比如 -f/--foo。如果解析器需要支持不同的或者额外的字符，比如像 +f 或者 /foo 的选项，可以在参数解析构建器中使用 prefix_chars= 参数。

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

prefix_chars= 参数默认使用 '-'。提供一组不包括 - 的字符将导致 -f/--foo 选项不被允许。

fromfile_prefix_chars

有些时候，先举个例子，当处理一个特别长的参数列表的时候，把它存入一个文件中而不是在命令行打出来会很有意义。如果 fromfile_prefix_chars= 参数提供给*ArgumentParser* 构造函数，之后所有类型的字符的参数都会被当成文件处理，并且会被文件包含的参数替代。举个栗子:

```
>>> with open('args.txt', 'w') as fp:
...     fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

从文件读取的参数在默认情况下必须一个一行（但是可参见*convert_arg_line_to_args()*）并且它们被视为与命令行上的原始文件引用参数位于同一位置。所以在以上例子中，['-f', 'foo', '@args.txt'] 的表示和 ['-f', 'foo', '-f', 'bar'] 的表示相同。

fromfile_prefix_chars= 参数默认为 None，意味着参数不会被当作文件对待。

argument_default

一般情况下，参数默认会通过设置一个默认到*add_argument()* 或者调用带一组指定键值对的*ArgumentParser.set_defaults()* 方法。但是有些时候，为参数指定一个普遍适用的解析器会更有用。这能够通过传输 argument_default= 关键词参数给*ArgumentParser* 来完成。举个栗子，要全局禁止在*parse_args()* 中创建属性，我们提供 argument_default=SUPPRESS:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
```

(下页继续)

(续上页)

```
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

allow_abbrev

正常情况下，当你向*ArgumentParser* 的*parse_args()* 方法传入一个参数列表时，它会*recognizes abbreviations*。

这个特性可以设置 allow_abbrev 为 False 来关闭:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

3.5 新版功能.

conflict_handler

ArgumentParser 对象不允许在相同选项字符串下有两种行为。默认情况下，*ArgumentParser* 对象会产生一个异常如果去创建一个正在使用的选项字符串参数。

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

有些时候（例如: 使用*parents*），重写旧的有相同选项字符串的参数会更有用。为了产生这种行为，'resolve' 值可以提供给*ArgumentParser* 的 conflict_handler= 参数:

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

注意*ArgumentParser* 对象只能移除一个行为如果它所有的选项字符串都被重写。所以，在上面的例子中，旧的 -f/--foo 行为回合 -f 行为保持一样，因为只有 --foo 选项字符串被重写。

add_help

默认情况下，ArgumentParser 对象添加一个简单的显示解析器帮助信息的选项。举个栗子，考虑一个名为 myprogram.py 的文件包含如下代码:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

如果 -h or --help 在命令行中被提供, 参数解析器帮助信息会打印:

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

有时候可能会需要关闭额外的帮助信息。这可以通过在*ArgumentParser* 中设置 add_help= 参数为 False 来实现。

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]

optional arguments:
 --foo FOO  foo help
```

帮助选项一般为 -h/--help。如果 prefix_chars= 被指定并且没有包含 - 字符，在这种情况下，-h --help 不是有效的选项。此时，prefix_chars 的第一个字符将用作帮助选项的前缀。

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
  +h, ++help  show this help message and exit
```

16.4.3 add_argument() 方法

ArgumentParser.**add_argument**(*name or flags...*[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*])

定义单个的命令行参数应当如何解析。每个形参都在下面有它自己更多的描述，长话短说有:

- *name or flags* - 一个命名或者一个选项字符串的列表，例如 foo 或 -f, --foo。
- *action* - 当参数在命令行中出现时使用的动作基本类型。
- *nargs* - 命令行参数应当消耗的数目。
- *const* - 被一些*action* 和*nargs* 选择所需求的常数。
- *default* - 当参数未在命令行中出现时使用的值。
- *type* - 命令行参数应当被转换成的类型。
- *choices* - 可用的参数的容器。

- *required* - 此命令行选项是否可省略（仅选项可用）。
- *help* - 一个此选项作用的简单描述。
- *metavar* - 在使用方法消息中使用的参数值示例。
- *dest* - 被添加到*parse_args()* 所返回对象上的属性名。

以下部分描述这些参数如何使用。

name or flags

add_argument() 方法必须知道它是否是一个选项，例如 `-f` 或 `--foo`，或是一个位置参数，例如一组文件名。第一个传递给*add_argument()* 的参数必须是一系列 flags 或者是一个简单的参数名。例如，可以选项可以被这样创建:

```
>>> parser.add_argument('-f', '--foo')
```

而位置参数可以这么创建:

```
>>> parser.add_argument('bar')
```

当*parse_args()* 被调用，选项会以 - 前缀识别，剩下的参数则会被假定为位置参数:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

action

ArgumentParser 对象将命令行参数与动作相关联。这些动作可以做与它们相关联的命令行参数的任何事，尽管大多数动作只是简单的向*parse_args()* 返回的对象上添加属性。`action` 命名参数指定了这个命令行参数应当如何处理。供应的动作有:

- `'store'` - 存储参数的值。这是默认的动作。例如:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo')
  >>> parser.parse_args('--foo 1'.split())
  Namespace(foo='1')
  ```

- `'store_const'` - 存储被*const* 命名参数指定的值。`'store_const'` 动作通常用在选项中来指定一些标志。例如:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_const', const=42)
  >>> parser.parse_args(['--foo'])
  Namespace(foo=42)
  ```

- 'store_true' and 'store_false' - 这些是 'store_const' 分别用作存储 True 和 False 值的特殊用例。另外，它们的默认值分别为 False 和 True。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- 'append' - 存储一个列表，并且将每个参数值追加到列表中。在允许多次使用选项时很有用。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- 'append_const' - 这存储一个列表，并将*const* 命名参数指定的值追加到列表中。(注意*const* 命名参数默认为 None。) "' append_const' " 动作一般在多个参数需要在同一列表中存储常数时会有用。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- 'count' - 计算一个关键字参数出现的数目或次数。例如，对于一个增长的详情等级来说有用:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count')
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

- 'help' - 打印所有当前解析器中的选项和参数的完整帮助信息，然后退出。默认情况下，一个 help 动作会被自动加入解析器。关于输出是如何创建的，参与*ArgumentParser*。
- 'version' - 期望有一个 version= 命名参数在*add_argument()* 调用中，并打印版本信息并在调用后退出:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

您还可以通过传递 Action 子类或实现相同接口的其他对象来指定任意操作。建议的方法是扩展*Action*，覆盖 __call__ 方法和可选的 __init__ 方法。

一个自定义动作的例子:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super(FooAction, self).__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
```

(下页继续)

(续上页)

```
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

更多描述，见*Action*。

nargs

ArgumentParser 对象通常关联一个单独的命令行参数到一个单独的被执行的动作。nargs 命名参数关联不同数目的命令行参数到单一动作。支持的值有：

- N （一个整数）。命令行中的 N 个参数会被聚集到一个列表中。例如：

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs=2)
  >>> parser.add_argument('bar', nargs=1)
  >>> parser.parse_args('c --foo a b'.split())
  Namespace(bar=['c'], foo=['a', 'b'])
  ```

  注意 nargs=1 会产生一个单元素列表。这和默认的元素本身是不同的。

- '?'。如果可能的话，会从命令行中消耗一个参数，并产生一个单一项。如果当前没有命令行参数，则会产生*default* 值。注意，对于选项，有另外的用例 - 选项字符串出现但没有跟随命令行参数，则会产生*const* 值。一些说用用例：

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs='?', const='c', default='d')
  >>> parser.add_argument('bar', nargs='?', default='d')
  >>> parser.parse_args(['XX', '--foo', 'YY'])
  Namespace(bar='XX', foo='YY')
  >>> parser.parse_args(['XX', '--foo'])
  Namespace(bar='XX', foo='c')
  >>> parser.parse_args([])
  Namespace(bar='d', foo='d')
  ```

  nargs='?' 的一个更普遍用法是允许可选的输入或输出文件：

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
  ...                     default=sys.stdin)
  >>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
  ...                     default=sys.stdout)
  >>> parser.parse_args(['input.txt', 'output.txt'])
  Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
            outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
  >>> parser.parse_args([])
  Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
            outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
  ```

- '*'。所有当前命令行参数被聚集到一个列表中。注意通过 nargs='*' 来实现多个位置参数通常没有意义，但是多个选项是可能的。例如：

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- `'+'`。和 `'*'` 类似，所有当前命令行参数被聚集到一个列表中。另外，当前没有至少一个命令行参数时会产生一个错误信息。例如:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

- `argarse.REMAINDER`。所有剩余的命令行参数被聚集到一个列表中。这通常在从一个命令行功能传递参数到另一个命令行功能中时有用:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo')
>>> parser.add_argument('command')
>>> parser.add_argument('args', nargs=argparse.REMAINDER)
>>> print(parser.parse_args('--foo B cmd --arg1 XX ZZ'.split()))
Namespace(args=['--arg1', 'XX', 'ZZ'], command='cmd', foo='B')
```

如果不提供 `nargs` 命名参数，则消耗参数的数目将被*action* 决定。通常这意味着单一项目（非列表）消耗单一命令行参数。

const

add_argument() 的 “const“ 参数用于保存不从命令行中读取但被各种*ArgumentParser* 动作需求的常数值。最常用的两例为:

- 当*add_argument()* 通过 `action='store_const'` 或 `action='append_const` 调用时。这些动作将 `const` 值添加到*parse_args()* 返回的对象的属性中。在*action* 的描述中查看案例。
- 当*add_argument()* 通过选项（例如 `-f` 或 `--foo`）调用并且 `nargs='?'` 时。这会创建一个可以跟随零个或一个命令行参数的选项。当解析命令行时，如果选项后没有参数，则将用 `const` 代替。在*nargs* 描述中查看案例。

对 `'store_const'` 和 `'append_const'` 动作，`const` 命名参数必须给出。对其他动作，默认为 `None`。

默认值

所有选项和一些位置参数可能在命令行中被忽略。*add_argument()* 的命名参数 `default`，默认值为 `None`，指定了在命令行参数未出现时应当使用的值。对于选项，`default` 值在选项未在命令行中出现时使用:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
```

(下页继续)

(续上页)

```
>>> parser.parse_args([])
Namespace(foo=42)
```

如果 default 值是一个字符串，解析器解析此值就像一个命令行参数。特别是，在将属性设置在*Namespace* 的返回值之前，解析器应用任何提供的*type* 转换参数。否则解析器使用原值:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

对于*nargs* 等于 ? 或 * 的位置参数，default 值在没有命令行参数出现时使用。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

提供 default=argparse.SUPPRESS 导致命令行参数未出现时没有属性被添加:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type -类型

默认情况下，*ArgumentParser* 对象将命令行参数当作简单字符串读入。然而，命令行字符串经常需要被当作其它的类型，比如*float* 或者*int*。*add_argument()* 的 type 关键词参数允许任何的类型检查和类型转换。一般的内建类型和函数可以直接被 type 参数使用。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.add_argument('bar', type=open)
>>> parser.parse_args('2 temp.txt'.split())
Namespace(bar=<_io.TextIOWrapper name='temp.txt' encoding='UTF-8'>, foo=2)
```

当 type 参数被应用到默认参数时，请参考*default* 参数的部分。

为方便使用不同类型的文件，argparse 模块提供了 FileType 工厂类，该类接受 mode=, bufsize=, encoding= 和 errors= 等*open()* 函数参数。例如，FileType('w') 可被用来创建一个可写文件:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar', type=argparse.FileType('w'))
>>> parser.parse_args(['out.txt'])
Namespace(bar=<_io.TextIOWrapper name='out.txt' encoding='UTF-8'>)
```

type= 可接受任意可调用对象，该对象应传入单个字符串参数并返回转换后的值:

```
>>> def perfect_square(string):
...     value = int(string)
...     sqrt = math.sqrt(value)
...     if sqrt != int(sqrt):
...         msg = "%r is not a perfect square" % string
...         raise argparse.ArgumentTypeError(msg)
...     return value
...
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=perfect_square)
>>> parser.parse_args(['9'])
Namespace(foo=9)
>>> parser.parse_args(['7'])
usage: PROG [-h] foo
PROG: error: argument foo: '7' is not a perfect square
```

choices 关键词参数可能会使类型检查者更方便的检查一个范围的值。

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', type=int, choices=range(5, 10))
>>> parser.parse_args(['7'])
Namespace(foo=7)
>>> parser.parse_args(['11'])
usage: PROG [-h] {5,6,7,8,9}
PROG: error: argument foo: invalid choice: 11 (choose from 5, 6, 7, 8, 9)
```

详情请查阅*choices* 段落。

choices

某些命令行参数应当从一组受限值中选择。这可通过将一个容器对象作为 *choices* 关键字参数传给*add_argument()* 来处理。当执行命令行解析时，参数值将被检查，如果参数不是可接受的值之一就将显示错误消息:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

请注意 *choices* 容器包含的内容会在执行任意*type* 转换之后被检查，因此 *choices* 容器中对象的类型应当与指定的*type* 相匹配:

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

任何支持 `in` 运算符的对象都可作为 *choices* 值传入，因此*dict* 对象，*set* 对象，自定义容器等都是受支持的。

required

通常，*argparse* 模块会认为 `-f` 和 `--bar` 等旗标是指明 *可选的* 参数，它们总是可以在命令行中被忽略。要让一个选项成为 *必需的*，则可以将 `True` 作为 `required=` 关键字参数传给*add_argument()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: argparse.py [-h] [--foo FOO]
argparse.py: error: option --foo is required
```

如这个例子所示，如果一个选项被标记为 `required`，则当该选项未在命令行中出现时，*parse_args()* 将会报告一个错误。

注解： 必需的选项通常被认为是不适宜的，因为用户会预期 *options* 都是 *可选的*，因此在可能的情况下应当避免使用它们。

help

`help` 值是一个包含参数简短描述的字符串。当用户请求帮助时（一般是通过在命令行中使用 `-h` 或 `--help` 的方式），这些 `help` 描述将随每个参数一同显示:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                     help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
 bar     one of the bars to be frobbled

optional arguments:
 -h, --help  show this help message and exit
 --foo   foo the bars before frobbling
```

`help` 字符串可包括各种格式描述符以避免重复使用程序名称或参数*default* 等文本。有效的描述符包括程序名称 `%(prog)s` 和传给*add_argument()* 的大部分关键字参数，例如 `%(default)s`, `%(type)s` 等等:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

optional arguments:
 -h, --help  show this help message and exit
```

由于帮助字符串支持%-formatting，如果你希望在帮助字符串中显示 `%` 字面值，你必须将其转义为 `%%`。

argparse 支持静默特定选项的帮助，具体做法是将 help 的值设为 argparse.SUPPRESS:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

optional arguments:
  -h, --help  show this help message and exit
```

metavar

当*ArgumentParser* 生成帮助消息时，它需要用某种方式来引用每个预期的参数。默认情况下，ArgumentParser 对象使用*dest* 值作为每个对象的 "name"。默认情况下，对于位置参数动作，*dest* 值将被直接使用，而对于可选参数动作，*dest* 值将被转为大写形式。因此，一个位置参数 dest='bar' 的引用形式将为 bar。一个带有单独命令行参数的可选参数 --foo 的引用形式将为 FOO。示例如下:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
 bar

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO
```

可以使用 metavar 来指定一个替代名称:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

optional arguments:
 -h, --help  show this help message and exit
 --foo YYY
```

请注意 metavar 仅改变 显示的名称 - *parse_args()* 对象的属性名称仍然会由*dest* 值确定。

不同的 nargs 值可能导致 metavar 被多次使用。提供一个元组给 metavar 即为每个参数指定不同的显示信息:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
```

(下页继续)

(续上页)

```
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

dest

大多数*ArgumentParser* 动作会添加一些值作为*parse_args()* 所返回对象的一个属性。该属性的名称由*add_argument()* 的 dest 关键字参数确定。对于位置参数动作，dest 通常会作为*add_argument()* 的第一个参数提供:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

对于可选参数动作，dest 的值通常取自选项字符串。*ArgumentParser* 会通过接受第一个长选项字符串并去掉开头的 -- 字符串来生成 dest 的值。如果没有提供长选项字符串，则 dest 将通过接受第一个短选项字符串并去掉开头的 - 字符来获得。任何内部的 - 字符都将被转换为 _ 字符以确保字符串是有效的属性名称。下面的例子显示了这种行为:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

dest 允许提供自定义属性名称:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

Action 类

Action 类实现了 Action API，它是一个返回可调用对象的可调用对象，返回的可调用对象可处理来自命令行的参数。任何遵循此 API 的对象均可作为 action 形参传给 add_argument()。

class argparse.**Action**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Action 对象会被 ArgumentParser 用来表示解析从命令行中的一个或多个字符串中解析出单个参数所必须的信息。Action 类必须接受两个位置参数以及传给*ArgumentParser.add_argument()* 的任何关键字参数，除了 action 本身。

Action 的实例（或作为 or return value of any callable to the action 形参的任何可调用对象的返回值）应当定义“dest”，“option_strings”，“default”，“type”，“required”，“help”等属性。确保这些属性被定义的最容

易方式是调用 Action.__init__。

Action 的实例应当为可调用对象，因此所有子类都必须重载 __call__ 方法，该方法应当接受四个形参:

- parser - 包含此动作的 ArgumentParser 对象。
- namespace - 将由*parse_args()* 返回的*Namespace* 对象。大多数动作会使用*setattr()* 为此对象添加属性。
- values - 已关联的命令行参数，并提供相应的类型转换。类型转换由*add_argument()* 的*type* 关键字参数来指定。
- option_string - 被用来发起调用此动作的选项字符串。option_string 参数是可选的，且此参数在动作关联到位置参数时将被略去。

__call__ 方法可以执行任意动作，但通常将基于 dest 和 values 来设置 namespace 的属性。

16.4.4 parse_args() 方法

ArgumentParser.**parse_args**(*args=None*, *namespace=None*)

将参数字符串转换为对象并将其设为命名空间的属性。返回带有成员的命名空间。

之前对*add_argument()* 的调用决定了哪些对象被创建以及它们如何被赋值。请参阅*add_argument()* 的文档了解详情。

- *args* - 要解析的字符串列表。默认情况下是从*sys.argv* 获取。
- *namespace* - 用于获取属性的对象。默认值是一个新的空*Namespace* 对象。

选项值语法

parse_args() 方法支持多种指定选项值的方式（如果它接受选项的话）。在最简单的情况下，选项和它的值是作为两个单独参数传入的:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

对于长选项（名称长度超过一个字符的选项），选项和值也可以作为单个命令行参数传入，使用 = 分隔它们即可:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

对于短选项（长度只有一个字符的选项），选项和它的值可以拼接在一起:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

有些短选项可以使用单个 - 前缀来进行合并，如果仅有最后一个选项（或没有任何选项）需要值的话:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
```

(下页继续)

(续上页)

```
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

无效的参数

在解析命令行时，*parse_args()* 会检测多种错误，包括有歧义的选项、无效的类型、无效的选项、错误的位置参数个数等等。当遇到这种错误时，它将退出并打印出错误文本同时附带用法消息:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

包含 - 的参数

parse_args() 方法会在用户明显出错时尝试给出错误信息，但某些情况本身就存在歧义。例如，命令行参数 `-1` 可能是尝试指定一个选项也可能是尝试提供一个位置参数。*parse_args()* 方法在此会谨慎行事: 位置参数只有在它们看起来像负数并且解析器中没有任何选项看起来像负数时才能以 - 打头。:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')
```

(下页继续)

(续上页)

```
>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

如果你有必须以 - 打头的位置参数并且看起来不像负数，你可以插入伪参数 `'--'` 以告诉*parse_args()* 在那之后的内容是一个位置参数:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

参数缩写（前缀匹配）

parse_args() 方法在默认情况下 允许将长选项缩写为前缀，如果缩写无歧义（即前缀与一个特定选项相匹配）的话:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

可产生一个以上选项的参数会引发错误。此特定可通过将*allow_abbrev* 设为 `False` 来禁用。

在 sys.argv 以外

有时在*sys.argv* 以外用 ArgumentParser 解析参数也是有用的。这可以通过将一个字符串列表传给*parse_args()* 来实现。它适用于在交互提示符下进行检测:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

命名空间对象

class argparse.**Namespace**

由*parse_args()* 默认使用的简单类，可创建一个存放属性的对象并将其返回。

这个类被有意做得很简单，只是一个具有可读字符串表示形式的*object*。如果你更喜欢类似字典的属性视图，你可以使用标准 Python 中惯常的*vars()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

另一个用处是让*ArgumentParser* 为一个已存在对象而不是为一个新的*Namespace* 对象的属性赋值。这可以通过指定 namespace= 关键字参数来实现:

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

16.4.5 其它实用工具

子命令

ArgumentParser.**add_subparsers**([*title*][, *description*][, *prog*][, *parser_class*][, *action*][, *option_string*][, *dest*][, *help*][, *metavar*])

许多程序都会将其功能拆分为一系列子命令，例如，svn 程序包含的子命令有 svn checkout, svn update 和 svn commit。当一个程序能执行需要多组不同种类命令行参数时这种拆分功能的方式是一个非常好的主意。*ArgumentParser* 通过*add_subparsers()* 方法支持创建这样的子命令。*add_subparsers()* 方法通常不带参数地调用并返回一个特殊的动作对象。这种对象只有一个方法 add_parser()，它接受一个命令名称和任意多个*ArgumentParser* 构造器参数，并返回一个可以通常方式进行修改的*ArgumentParser* 对象。

形参的描述

- title - 输出帮助的子解析器分组的标题；如果提供了描述则默认为 “subcommands”，否则使用位置参数的标题
- description - 输出帮助中对子解析器的描述，默认为 None
- prog - 将与子命令帮助一同显示的用法信息，默认为程序名称和子解析器参数之前的任何位置参数。
- parser_class - 将被用于创建子解析器实例的类，默认为当前解析器类（例如 ArgumentParser）
- *action* - 当此参数在命令行中出现时要执行动作的基本类型
- *dest* - 将被用于保存子命令名称的属性名；默认为 None 即不保存任何值
- *help* - 在输出帮助中的子解析器分组帮助信息，默认为 None

- *metavar* - 帮助信息中表示可用子命令的字符串；默认为 None 并以 {cmd1, cmd2, ..} 的形式表示子命令

一些使用示例:

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

请注意*parse_args()* 返回的对象将只包含主解析器和由命令行所选择的子解析器的属性（而没有任何其他子解析器）。因此在上面的例子中，当指定了 a 命令时，将只存在 foo 和 bar 属性，而当指定了 b 命令时，则只存在 foo 和 baz 属性。

类似地，当从一个子解析器请求帮助消息时，只有该特定解析器的帮助消息会被打印出来。帮助消息将不包括父解析器或同级解析器的消息。（每个子解析器命令一条帮助消息，但是，也可以像上面那样通过提供 help= 参数给 add_parser() 来给出。）

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   sub-command help
    a     a help
    b     b help

optional arguments:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

optional arguments:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

add_subparsers() 方法也支持 `title` 和 `description` 关键字参数。当两者都存在时，子解析器的命令将出现在输出帮助消息中它们自己的分组内。例如:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

此外，add_parser 还支持附加的 `aliases` 参数，它允许多个字符串指向同一子解析器。这个例子类似于 `svn`，将别名 `co` 设为 `checkout` 的缩写形式:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

一个特别有效的处理子命令的方式是将*add_subparsers()* 方法与对*set_defaults()* 的调用结合起来使用，这样每个子解析器就能知道应当执行哪个 Python 函数。例如:

```
>>> # sub-command functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
```

(下页继续)

(续上页)

```
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

通过这种方式，你可以在参数解析结束后让*parse_args()* 执行调用适当函数的任务。像这样将函数关联到动作通常是你处理每个子解析器的不同动作的最简便方式。但是，如果有必要检查被发起调用的子解析器的名称，则*add_subparsers()* 调用的 dest 关键字参数将可实现:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

FileType 对象

class argparse.**FileType**(*mode='r'*, *bufsize=-1*, *encoding=None*, *errors=None*)

FileType 工厂类用于创建可作为*ArgumentParser.add_argument()* 的 type 参数传入的对象。以*FileType* 对象作为其类型的参数将使用命令行参数以所请求模式、缓冲区大小、编码格式和错误处理方式打开文件（请参阅*open()* 函数了解详情）:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=
↪<_io.FileIO name='raw.dat' mode='wb'>)
```

FileType 对象能理解伪参数 '-' 并会自动将其转换为 sys.stdin 用于可读的*FileType* 对象，或是 sys.stdout 用于可写的*FileType* 对象:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

3.4 新版功能: *encodings* 和 *errors* 关键字参数。

参数组

ArgumentParser.**add_argument_group**(*title=None*, *description=None*)

在默认情况下，*ArgumentParser* 会在显示帮助消息时将命令行参数分为“位置参数”和“可选参数”两组。当存在比默认更好的参数分组概念时，可以使用*add_argument_group()* 方法来创建适当的分组:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

add_argument_group() 方法返回一个具有*add_argument()* 方法的参数分组对象，这与常规的*ArgumentParser* 一样。当一个参数被加入分组时，解析器会将它视为一个正常的参数，但是会在不同的帮助消息分组中显示该参数。*add_argument_group()* 方法接受 *title* 和 *description* 参数，它们可被用来定制显示内容:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

请注意任意不在你的自定义分组中的参数最终都将回到通常的“位置参数”和“可选参数”分组中。

互斥

ArgumentParser.**add_mutually_exclusive_group**(*required=False*)

创建一个互斥组。*argparse* 将会确保互斥组中只有一个参数在命令行中可用:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
```

(下页继续)

(续上页)

```
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

add_mutually_exclusive_group() 方法也接受一个 *required* 参数，表示在互斥组中至少有一个参数是需要的:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

注意，目前互斥参数组不支持*add_argument_group()* 的 *title* 和 *description* 参数。

解析器默认值

ArgumentParser.**set_defaults**(***kwargs*)

在大多数时候，*parse_args()* 所返回对象的属性将完全通过检查命令行参数和参数动作来确定。*set_defaults()* 则允许加入一些无须任何命令行检查的额外属性:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

请注意解析器层级的默认值总是会覆盖参数层级的默认值:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

解析器层级默认值在需要多解析器时会特别有用。请参阅*add_subparsers()* 方法了解此类型的一个示例。

ArgumentParser.**get_default**(*dest*)

获取一个命名空间属性的默认值，该值是由*add_argument()* 或*set_defaults()* 设置的:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

打印帮助

在大多数典型应用中，*parse_args()* 将负责任何用法和错误消息的格式化和打印。但是，也可使用某些其他格式化方法:

ArgumentParser.**print_usage**(*file=None*)
打印一段简短描述，说明应当如何在命令行中发起调用*ArgumentParser*。如果 *file* 为 None，则默认使用*sys.stdout*。

ArgumentParser.**print_help**(*file=None*)
打印一条帮助消息，包括程序用法和通过*ArgumentParser* 注册的相关参数信息。如果 *file* 为 None，则默认使用*sys.stdout*。

还存在这些方法的几个变化形式，它们只返回字符串而不打印消息:

ArgumentParser.**format_usage**()
返回一个包含简短描述的字符串，说明应当如何在命令行中发起调用*ArgumentParser*。

ArgumentParser.**format_help**()
反回一个包含帮助消息的字符串，包括程序用法和通过*ArgumentParser* 注册的相关参数信息。

部分解析

ArgumentParser.**parse_known_args**(*args=None*, *namespace=None*)

有时一个脚本可能只解析部分命令行参数，而将其余的参数继续传递给另一个脚本或程序。在这种情况下，*parse_known_args()* 方法会很有用处。它的作用方式很类似*parse_args()* 但区别在于当存在额外参数时它不会产生错误。而是会返回一个由两个条目构成的元组，其中包含带成员的命名空间和剩余参数字符串的列表。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

警告: 前缀匹配 规则应用于 parse_known_args()。一个选项即使只是已知选项的前缀部分解析器也能识别该选项，不会将其放入剩余参数列表。

自定义文件解析

ArgumentParser.**convert_arg_line_to_args**(*arg_line*)
从文件读取的参数（见*ArgumentParser* 的 *fromfile_prefix_chars* 关键字参数）将是一行读取一个参数。*convert_arg_line_to_args()* 可被重载以使用更复杂的读取方式。

此方法接受从参数文件读取的字符串形式的单个参数 *arg_line*。它返回从该字符串解析出的参数列表。此方法将在每次按顺序从参数文件读取一行时被调用一次。

此方法的一个有用的重载是将每个以空格分隔的单词视为一个参数。下面的例子演示了如何实现此重载:

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

退出方法

ArgumentParser.**exit**(*status=0*, *message=None*)
此方法将终结程序，附带指定的 *status* 退出，并且如果给出了 *message* 则会在退出前将其打印输出。

ArgumentParser.**error**(*message*)
此方法将向标准错误打印包括 *message* 的用法消息并附带状态码 2 终结程序。

16.4.6 升级 optparse 代码

起初，*argparse* 曾经尝试通过*optparse* 来维持兼容性。但是，*optparse* 很难透明地扩展，特别是那些为支持新的 nargs= 描述方式和更好的用法消息所需的修改。当 When most everything in *optparse* 中几乎所有内容都已被复制粘贴或打上补丁时，维持向下兼容看来已是不切实际的。

argparse 模块在许多方面对标准库的*optparse* 模块进行了增强，包括:

- 处理位置参数。
- 支持子命令。
- 允许替代选项前缀例如 + 和 /。
- 处理零个或多个以及一个或多个风格的参数。
- 生成更具信息量的用法消息。
- 提供用于定制 type 和 action 的更为简单的接口。

从*optparse* 到*argparse* 的部分升级路径:

- 将所有*optparse.OptionParser.add_option()* 调用替换为*ArgumentParser.add_argument()* 调用。
- 将 (options, args) = parser.parse_args() 替换为 args = parser.parse_args() 并为位置参数添加额外的*ArgumentParser.add_argument()* 调用。请注意之前所谓的 options 在*argparse* 上下文中被称为 args。
- Replace *optparse.OptionParser.disable_interspersed_args()* by setting nargs of a positional argument to *argparse.REMAINDER*, or use *parse_known_args()* to collect unparsed argument strings in a separate list.
- 将回调动作和 callback_* 关键字参数替换为 type 或 action 参数。
- 将 type 关键字参数字符串名称替换为相应的类型对象（例如 int, float, complex 等）。
- 将 optparse.Values 替换为*Namespace* 并将 optparse.OptionError 和 optparse.OptionValueError 替换为 ArgumentError。
- 将隐式参数字符串例如使用标准 Python 字典语法的 %default 或 %prog 替换为格式字符串，即 %(default)s 和 %(prog)s。
- 将 OptionParser 构造器 version 参数替换为对 parser.add_argument('--version', action='version', version='<the version>') 的调用。

16.5 getopt —C 风格的命令行选项解析器

资源代码：资源：’Lib/getopt.py’

注解： *getopt* 模块是一个命令行选项解析器，其 API 设计会让 C `getopt()` 函数的用户感到熟悉。不熟悉 C `getopt()` 函数或者希望写更少代码并获得更完善帮助和错误消息的用户应当考虑改用*argparse* 模块。

此模块可协助脚本解析 `sys.argv` 中的命令行参数。它支持与 Unix `getopt()` 函数相同的惯例（包括形式如‘`-`‘与‘`--`‘的参数的特殊含义）。也能通过可选的第三个参数来使用与 GNU 软件所支持形式相类似的长选项。

此模块提供了两个函数和一个异常:

`getopt.`**`getopt`**(*args*, *shortopts*, *longopts=[]*)

解析命令行选项与形参列表。*args* 为要解析的参数列表，不包含最开头的对正在运行的程序的引用。通常这意味着 `sys.argv[1:]`。*shortopts* 为脚本所要识别的字母选项，包含要求后缀一个冒号 (`':'`；即与 Unix `getopt()` 所用的格式相同) 的选项。

注解： 与 GNU `getopt()` 不同，在非选项参数之后，所有后续参数都会被视为非选项。这类似于非 GNU Unix 系统的运作方式。

如果指定了 *longopts*，则必须为一个由应当被支持的长选项名称组成的列表。开头的 `'--'` 字符不应被包括在选项名称中。要求参数的长选项后应当带一个等号 (`'='`)。可选参数不被支持。如果想仅接受长选项，则 *shortopts* 应为一个空字符串。命令行中的长选项只要提供了恰好能匹配可接受选项之一的选项名称前缀即可被识别。举例来说，如果 *longopts* 为 `['foo', 'frob']`，则选项 `--fo` 将匹配为 `--foo`，但 `--f` 将不能得到唯一匹配，因此将引发*GetoptError*。

返回值由两个元素组成：第一个是 `(option, value)` 对的列表；第二个是在去除该选项列表后余下的程序参数列表（这也就是 *args* 的尾部切片）。每个被返回的选项与值对的第一个元素是选项，短选项前缀一个连字符 (例如 `'-x'`)，长选项则前缀两个连字符 (例如 `'--long-option'`)，第二个元素是选项参数，如果选项不带参数则为空字符串。列表中选项的排列顺序与它们被解析的顺序相同，因此允许多次出现。长选项与短选项可以混用。

`getopt.`**`gnu_getopt`**(*args*, *shortopts*, *longopts=[]*)

此函数与*getopt()* 类似，区别在于它默认使用 GNU 风格的扫描模式。这意味着选项和非选项参数可能会混在一起。*getopt()* 函数将在遇到非选项参数时立即停止处理选项。

如果选项字符串的第一个字符为 `'+'`，或者如果设置了环境变量 `POSIXLY_CORRECT`，则选项处理会在遇到非选项参数时立即停止。

exception `getopt.`**`GetoptError`**

This is raised 当参数列表中出现不可识别的选项或者当一个需要参数的选项未带参数时将引发此异常。此异常的参数是一个指明错误原因的字符串。对于长选项，将一个参数传给不需要参数的选项也将导致引发此异常。`msg` 和 `opt` 属性会给出错误消息和关联的选项；如果没有关联到异常的特定选项，则 `opt` 将为空字符串。

exception `getopt.`**`error`**

GetoptError 的别名；用于向后兼容。

一个仅使用 Unix 风格选项的例子:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

使用长选项名也同样容易:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

在脚本中，典型的用法类似这样:

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    # ...

if __name__ == "__main__":
    main()
```

请注意通过*argparse*模块可以使用更少的代码并附带更详细的帮助与错误消息生成等价的命令行接口:

```
import argparse

if __name__ == '__main__':
```

(下页继续)

(续上页)

```
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..
```

参见:

模块*argparse* 替代的命令行选项和参数解析库。

16.6 模块 logging —Python 的日志记录工具

源代码: Lib/logging/__init__.py

Important

此页面仅包含 API 参考信息。有关更多高级主题的教程信息和讨论，请参阅

- 基础教程
- 进阶教程
- Logging Cookbook

这个模块为应用与库定义了实现灵活的事件日志系统的函数与类.

使用标准库提提供的 logging API 最主要的好处是，所有的 Python 模块都可能参与日志输出，包括你的日志消息和第三方模块的日志消息。

这个模块提供许多强大而灵活的功能。如果你对 logging 不太熟悉的话，掌握它最好的方式就是查看它对应的教程（详见右侧的链接）。

该模块定义的基础类和函数都列在下面。

- 记录器暴露了应用程序代码直接使用的接口。
- 处理程序将日志记录（由记录器创建）发送到适当的目标。
- 过滤器提供了更精细的附加功能，用于确定要输出的日志记录。
- 格式化程序指定最终输出中日志记录的样式。

16.6.1 Logger 对象

Loggers have the following attributes and methods. Note that Loggers are never instantiated directly, but always through the module-level function `logging.getLogger(name)`. Multiple calls to *`getLogger()`* with the same name will always return a reference to the same Logger object.

`name` 是潜在的周期分割层级值, 像 “foo.bar.baz“ (例如, 抛出的可以只是明文的 “foo“)。Loggers 是进一步在子层次列表的更高 loggers 列表。例如，有个名叫 “foo“的 logger，名叫 “foo.bar“，`foo.bar.baz`, 和 `foo.bam` 都是 `foo``的衍生 logger. logger 的名字分级类似 Python 包的层级，并且相同的如果你组织你的 loggers 在每模块级别基本上使用推荐的结构 ``logging.getLogger(__name__)`。这是因为在模块里，在 Python 包的命名空间的模块名为 “__name__“。

class logging.**Logger**

propagate
如果这个属性为真，记录到这个记录器的事件将会传递给这个高级别管理器的记录器（原型），此外任何关联到这个记录器的管理器。消息会直接传递给原型记录器的管理器 - 既不是这个原型记录器的级别也不是过滤器是在考虑的问题。

如果等于假，记录消息将不会传递给这个原型记录器的管理器。

构造器将这个属性初始化为 `True`。

注解： 如果你关联了一个管理器 * 并且 * 到它自己的一个或多个记录器，它可能发出多次相同的记录。总体来说，你不需要关联管理器到一个或多个记录器 - 如果你只是关联它到一个合适的记录器等级中的最高级别记录器，它将会看到子记录器所有记录的事件，他们的传播剩下的设置为“True“。一个通用场景是只关联管理器到根记录器，并且让传播照顾剩下的.

---

setLevel(*level*)
给 logger 设置阈值为 *level* 。日志等级小于 *level* 会被忽略。严重性为 *level* 或更高的日志消息将由该 logger 的任何一个或多个 handler 发出，除非将处理程序的级别设置为比 *level* 更高的级别。

创建一个 logger 时，设置级别为 `NOTSET`（当 logger 是根 logger 时，将处理所有消息；当 logger 是非根 logger 时，所有消息会委派给父级）。注意根 logger 创建时使用的是 `WARNING` 级别。

委派给父级的意思是如果一个记录器的级别设置为 NOTSET，遍历其祖先记录器链，直到找到另一个 NOTSET 级别的祖先或到达根为止。

如果发现某个父级的级别不是 NOTSET ，那么该父级的级别将被视为发起搜索的记录器的有效级别，并用于确定如何处理日志事件。

如果到达根 logger ，并且其级别为 NOTSET，则将处理所有消息。否则，将使用根记录器的级别作为有效级别。

参见*日志级别* 级别列表。

在 3.2 版更改: 现在 *level* 参数可以接受形如‘INFO’的级别字符串表示形式，以代替形如 `INFO` 的整数常量。但是请注意，级别在内部存储为整数，并且*getEffectiveLevel()* 和*isEnabledFor()* 等方法的传入/返回值也为整数。

isEnabledFor(*lvl*)
Indicates if a message of severity *lvl* would be processed by this logger. This method checks first the module-level level set by `logging.disable(lvl)` and then the logger’s effective level as determined by *getEffectiveLevel()*.

getEffectiveLevel()
指示此记录器的有效级别。如果通过*setLevel()* 设置了除 `NOTSET` 以外的值，则返回该值。否则，将层次结构遍历到根，直到找到除 `NOTSET` 以外的其他值，然后返回该值。返回的值是一个整数，通常为 `logging.DEBUG`、`logging.INFO` 等等。

getChild(*suffix*)
返回由后缀确定的，是该记录器的后代的记录器。因此，`logging.getLogger('abc').getChild('def.ghi')` 与 `logging.getLogger('abc.def.ghi')` 将返回相同的记录器。这是一个便捷方法，当使用如 `__name__` 而不是字符串字面值命名父记录器时很有用。

3.2 新版功能.

debug(*msg*, **args*, ***kwargs*)
Logs a message with level `DEBUG` on this logger. The *msg* is the message format string, and the *args* are the arguments which are merged into *msg* using the string formatting operator. (Note that this means that you can use keywords in the format string, together with a single dictionary argument.)

There are three keyword arguments in *kwargs* which are inspected: *exc_info*, *stack_info*, and *extra*.

如果 *exc_info* 的求值结果不为 false，则它将异常信息添加到日志消息中。如果提供了一个异常元组（按照*sys.exc_info()* 返回的格式）或一个异常实例，则将其使用；否则，调用*sys.exc_info()* 以获取异常信息。

第二个可选关键字参数是 *stack_info*，默认为 `False`。如果为 `True`，则将堆栈信息添加到日志消息中，包括实际的日志调用。请注意，这与通过指定 *exc_info* 显示的堆栈信息不同：前者是从堆栈底部到当前线程中的日志记录调用的堆栈帧，而后者是在搜索异常处理程序时，跟踪异常而打开的堆栈帧的信息。

您可以独立于 *exc_info* 来指定 *stack_info*，例如，即使在未引发任何异常的情况下，也可以显示如何到达代码中的特定点。堆栈帧在标题行之后打印：

```
Stack (most recent call last):
```

这模仿了显示异常帧时所使用的 `Traceback (most recent call last):` 。

The third keyword argument is *extra* which can be used to pass a dictionary which is used to populate the __dict__ of the LogRecord created for the logging event with user-defined attributes. These custom attributes can then be used as you like. For example, they could be incorporated into logged messages. For example:

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

输出类似于

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection␣
↪reset
```

The keys in the dictionary passed in *extra* should not clash with the keys used by the logging system. (See the *Formatter* documentation for more information on which keys are used by the logging system.)

如果你选择在已记录的消息中使用这些属性，则需要格外小心。例如在上面的示例中，*Formatter* 已设置了格式字符串，其在 LogRecord 的属性字典中应有‘clientip’和‘user’。如果缺少这些属性，消息将不被记录，因为会引发字符串格式化异常，你始终需要传入带有这些键的 *extra* 字典。

尽管这可能很烦人，但此功能旨在用于特殊情况，例如在多个上下文中执行相同代码的多线程服务器，并且出现的有趣条件取决于此上下文（例如在上面的示例中就是远程客户端 IP 地址和已验证用户名）。在这种情况下，很可能将专门的*Formatter* 与特定的*Handler* 一起使用。

3.2 新版功能: 增加了 *stack_info* 参数。

在 3.5 版更改: The *exc_info* parameter can now accept exception instances.

info(*msg*, **args*, ***kwargs*)
在此记录器上记录 `INFO` 级别的消息。参数解释同*debug()*。

warning(*msg*, **args*, ***kwargs*)
在此记录器上记录 `WARNING` 级别的消息。参数解释同*debug()*。

注解: 有一个功能上与 `warning` 一致的方法 `warn`。由于 `warn` 已被弃用，请不要使用它——改为使用 `warning`。

---

error(*msg*, **args*, ***kwargs*)
在此记录器上记录 `ERROR` 级别的消息。参数解释同*debug()*。

critical(*msg*, **args*, ***kwargs*)
在此记录器上记录 CRITICAL 级别的消息。参数解释同*debug()*。

log(*lvl*, *msg*, **args*, ***kwargs*)
Logs a message with integer level *lvl* on this logger. The other arguments are interpreted as for *debug()*.

exception(*msg*, **args*, ***kwargs*)
在此记录器上记录 ERROR 级别的消息。参数解释同*debug()*。异常信息将添加到日志消息中。仅应从异常处理程序中调用此方法。

addFilter(*filter*)
将指定的过滤器 *filter* 添加到此记录器。

removeFilter(*filter*)
从此记录器中删除指定的处理程序 *filter*。

filter(*record*)
Applies this logger's filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be processed (passed to handlers). If one returns a false value, no further processing of the record occurs.

addHandler(*hdlr*)
将指定的处理程序 *hdlr* 添加到此记录器。

removeHandler(*hdlr*)
从此记录器中删除指定的处理器 *hdlr*。

findCaller(*stack_info=False*)
查找调用源的文件名和行号，以文件名，行号，函数名称和堆栈信息 4 元素元组的形式返回。堆栈信息将返回 None``除非 *stack_info* 为 ``True。

handle(*record*)
通过将记录传递给与此记录器及其祖先关联的所有处理器来处理（直到某个 *propagate* 值为 false）。此方法用于从套接字接收的未序列化的以及在本地创建的记录。使用*filter()* 进行记录程序级别过滤。

makeRecord(*name*, *lvl*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)
这是一种工厂方法，可以在子类中对其进行重写以创建专门的*LogRecord* 实例。

hasHandlers()
检查此记录器是否配置了任何处理器。通过在此记录器及其记录器层次结构中的父级中查找处理器完成此操作。如果找到处理器则返回 True，否则返回 "False"。只要找到 "propagate" 属性设置为 false 的记录器，该方法就会停止搜索层次结构——其将是最后一个检查处理器是否存在的记录器。

3.2 新版功能.

在 3.7 版更改: Loggers can now be picked and unpickled.

16.6.2 日志级别

日志记录级别的数值在下表中给出。如果你想要定义自己的级别，并且需要它们具有相对于预定义级别的特定值，那么这些内容可能是你感兴趣的。如果你定义具有相同数值的级别，它将覆盖预定义的值; 预定义的名称丢失。

级别	数值
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

16.6.3 处理器对象

Handler 有以下属性和方法。注意不要直接实例化*Handler*；这个类用来派生其他更有用的子类。但是，子类的 __init__() 方法需要调用*Handler.__init__()*。

class logging.**Handler**

__init__(*level=NOTSET*)
初始化*Handler*实例时，需要设置它的级别，将过滤列表置为空，并且创建锁（通过*createLock()*）来序列化对 I/O 的访问。

createLock()
初始化一个线程锁，用来序列化对底层的 I/O 功能的访问，底层的 I/O 功能可能不是线程安全的。

acquire()
使用*createLock()*获取线程锁。

release()
使用*acquire()*来释放线程锁。

setLevel(*level*)
给处理器设置阈值为 *level*。日志级别小于 *level* 将被忽略。创建处理器时，日志级别被设置为 NOTSET（所有的消息都会被处理）。

参见日志级别 级别列表。

在 3.2 版更改: *level* 形参现在接受像‘INFO’这样的字符串形式的级别表达方式，也可以使用像 INFO 这样的整数常量。

setFormatter(*fmt*)
将此处理器的*Formatter*设置为 *fmt*。

addFilter(*filter*)
将指定的过滤器 *filter* 添加到此处理器。

removeFilter(*filter*)
从此处理器中删除指定的过滤器 *filter*。

filter(*record*)
Applies this handler's filters to the record and returns a true value if the record is to be processed. The filters are consulted in turn, until one of them returns a false value. If none of them return a false value, the record will be emitted. If one returns a false value, the handler will not emit the record.

flush()
确保所有日志记录从缓存输出。此版本不执行任何操作，并且应由子类实现。

close()
整理处理器使用的所有资源。此版本不输出，但从内部处理器列表中删除处理器，内部处理器在*shutdown()*被调用时关闭。子类应确保从重写的*close()*方法中调用此方法。

handle(*record*)
经已添加到处理器的过滤器过滤后，有条件地发出指定的日志记录。用获取/释放 I/O 线程锁包装记录的实际发出行为。

handleError(*record*)
调用*emit()* 期间遇到异常时，应从处理器中调用此方法。如果模块级属性 raiseExceptions 是 False，则异常将被静默忽略。这是大多数情况下日志系统需要的——大多数用户不会关心日志系统中的错误，他们对应用程序错误更感兴趣。但是，你可以根据需要将其替换为自定义处理器。指定的记录是发生异常时正在处理的记录。(raiseExceptions 的默认值是 True，因为这在开发过程中是比较有用的)。

format(*record*)
如果设置了格式器则用其对记录进行格式化。否则，使用模块的默认格式器。

emit(*record*)
执行实际记录给定日志记录所需的操作。这个版本应由子类实现，因此这里直接引发*NotImplementedError* 异常。

有关作为标准随附的处理程序，请参见*logging.handlers*。

16.6.4 格式器对象

Formatter 对象拥有以下的属性和方法。一般情况下，它们负责将*LogRecord* 转换为可由人或外部系统解释的字符串。基础的*Formatter* 允许指定格式字符串。如果未提供任何值，则使用默认值 '%(message)s'，它仅将消息包括在日志记录调用中。要在格式化输出中包含其他信息（如时间戳），请阅读下文。

格式器可以使用格式化字符串来初始化，该字符串利用*LogRecord* 的属性——例如上述默认值，用户的消息和参数预先格式化为*LogRecord* 的 *message* 属性后被使用。此格式字符串包含标准的 Python %-s 样式映射键。有关字符串格式的更多信息，请参见*printf* 风格的字符串格式化。

The useful mapping keys in a *LogRecord* are given in the section on *LogRecord* 属性.

class logging.**Formatter**(*fmt=None*, *datefmt=None*, *style='%'*)
返回*Formatter* 类的新实例。实例将使用整个消息的格式字符串以及消息的日期/时间部分的格式字符串进行初始化。如果未指定 *fmt*，则使用 '%(message)s'。如果未指定 *datefmt*，则使用*formatTime()* 文档中描述的格式。

style 形参可以是‘%’，‘{‘或‘$’之一，它决定格式字符串如何与数据进行合并：使用%-formatting, *str.format()* 或*string.Template* 之一。请参阅 formatting-styles 了解有关在日志消息中使用 {- 和 $-formatting 的更多信息。

在 3.2 版更改: *style* 参数已加入.

format(*record*)
记录的属性字典用作字符串格式化操作的参数。返回结果字符串。在格式化字典之前，需要执行几个准备步骤。使用 *msg % args* 计算记录的 *message* 属性。如果格式化字符串包含 '(asctime)'，则调用*formatTime()* 来格式化事件时间。如果有异常信息，则使用*formatException()* 将其格式化并附加到消息中。请注意，格式化的异常信息缓存在属性 *exc_text* 中。这很有用，因为可以对异常信息进行序列化并通过网络发送，但是如果您有不止一个定制了异常信息格式的*Formatter* 子类，则应格外小心。在这种情况下，您必须在格式器完成格式化后清除缓存的值，以便下一个处理事件的格式化程序不使用缓存的值，而是重新计算它。

如果栈信息可用，它将被添加在异常信息之后，如有必要请使用*formatStack()* 来转换它。

formatTime(*record*, *datefmt=None*)
此方法应由想要使用格式化时间的格式器中的*format()* 调用。可以在格式器中重写此方法以提供任何特定要求，但是基本行为如下：如果指定了 *datefmt*（字符串），则将其用于*time.strftime()* 来格式化记录的创建时间。否则，使用格式‘%Y-%m-%d %H:%M:%S,uuu’，其中 uuu

部分是毫秒值，其他字母根据*time.strftime()* 文档。这种时间格式的示例为 2003-01-23 00:29:50,411。返回结果字符串。

此函数使用一个用户可配置函数将创建时间转换为元组。默认情况下，使用*time.localtime()*；要为特定格式化程序实例更改此项，请将实例的 converter 属性设为具有与*time.localtime()* 或*time.gmtime()* 相同签名的函数。要为所有格式化程序更改此项，例如当你希望所有日志时间都显示为 GMT，请在 Formatter 类中设置 converter 属性。

在 3.3 版更改: 在之前版本中，默认格式为写死的代码，例如这个例子: 2010-09-06 22:38:15,292 其中逗号之前的部分由 strptime 格式字符串 ('%Y-%m-%d %H:%M:%S') 处理，而逗号之后的部分为毫秒值。因为 strptime 没有表示毫秒的占位符，毫秒值使用了另外的格式字符串来添加 '%s,%03d' —这两个格式字符串代码都是写死在该方法中的。经过修改，这些字符串被定义为类层级的属性，当需要时可以在实例层级上被重载。属性的名称为 default_time_format (用于 strptime 格式字符串) 和 default_msec_format (用于添加毫秒值)。

formatException(*exc_info*)
将指定的异常信息（由*sys.exc_info()* 返回的标准异常元组）格式化为字符串。这个默认实现只使用了*traceback.print_exception()*。结果字符串将被返回。

formatStack(*stack_info*)
将指定的堆栈信息（由*traceback.print_stack()* 返回的字符串，但移除末尾的换行符）格式化为字符串。这个默认实现只是返回输入值。

16.6.5 Filter 对象

Filters 可被 Handlers 和 Loggers 用来实现比按层级提供更复杂的过滤操作。基本过滤器类只允许低于日志记录器层级结构中低于特定层级的事件。例如，一个用 'A.B' 初始化的过滤器将允许 'A.B'，'A.B.C'，'A.B.C.D'，'A.B.D' 等日志记录器所记录的事件。但 'A.BB'，'B.A.B' 等则不允许。如果用空字符串初始化，则所有事件都会被略过。

class logging.**Filter**(*name=''*)
返回一个*Filter* 类的实例。如果指定了 *name*，则它将被用来为日志记录器命名，该类及其子类将通过该过滤器允许指定事件通过。如果 *name* 为空字符串，则允许所有事件通过。

filter(*record*)
是否要记录指定的记录？返回零表示否，非零表示是。如果认为合适，则可以通过此方法就地修改记录。

请注意关联到处理程序的过滤器会在事件由处理程序发出之前被查询，而关联到日志记录器的过滤器则会在有事件被记录的的任何时候（使用*debug()*，*info()* 等等）在将事件发送给处理程序之前被查询。这意味着由后代日志记录器生成的事件将不会被日志记录器的过滤器设置所过滤，除非该过滤器也已被应用于后代日志记录器。

你实际上不需要子类化 Filter: 你可以将传入任何包含 filter 方法的具有相同语义的的实例。

在 3.2 版更改: 你不需要创建专门的 Filter 类，或使用具有 filter 方法的其他类：你可以使用一个函数（或其他可调用对象）作为过滤器。过滤逻辑将检查过滤器对象是否文化的 filter 属性：如果有，就会将它当作是 Filter 并调用它的*filter()* 方法。在其他情况下，则会将它当作是可调用对象并附带记录作为单一形参进行调用。返回值应当与*filter()* 的返回值相一致。

Although filters are used primarily to filter records based on more sophisticated criteria than levels, they get to see every record which is processed by the handler or logger they're attached to: this can be useful if you want to do things like counting how many records were processed by a particular logger or handler, or adding, changing or removing attributes in the LogRecord being processed. Obviously changing the LogRecord needs to be done with some care, but it does allow the injection of contextual information into logs (see filters-contextual).

16.6.6 LogRecord 属性

LogRecord 实例是每当有日志被记录时由*Logger* 自动创建的，并且可通过*makeLogRecord()* 手动创建（例如根据从网络接收的已封存事件创建）。

class logging.**LogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*)

包含与被记录的事件相关的所有信息。

主要信息是在 msg 和 args 中传递的，它们使用 msg % args 组合到一起以创建记录的 message 字段。

参数

- **name** –用于记录由此 LogRecord 所表示的事件的日志记录器名称。请注意此名称将始终为该值，即使它可以是由附加到不同（祖先）日志记录器的处理程序所发出的。
- **level** –以数字表示的日志记录事件层级（如 DEBUG, INFO 等）。请注意这会转换为 LogRecord 的 两个属性: levelno 为数字值而 levelname 为对应的层级名称。
- **pathname** –进行日志记录调用的文件的完整路径名。
- **lineno** –记录调用所在源文件中的行号。
- **msg** –事件描述消息，可能为带有可变数据占位符的格式字符串。
- **args** –要合并到 *msg* 参数以获得事件描述的可变数据。
- **exc_info** –包含当前异常信息的异常元组，或者如果没有可用异常信息则为 None。
- **func** –发起调用日志记录调用的函数或方法名称。
- **sinfo** –一个文本字符串，表示当前线程中从堆栈底部直到日志记录调用的堆栈信息。

getMessage()

在将*LogRecord* 实例与任何用户提供的参数合并之后，返回此实例的消息。如果用户提供给日志记录调用的消息参数不是字符串，则会在其上调用*str()* 以将它转换为字符串。此方法允许将用户定义的类用作消息，类的 __str__ 方法可以返回要使用的实际格式字符串。

在 3.2 版更改: The creation of a LogRecord has been made more configurable by providing a factory which is used to create the record. The factory can be set using *getLogRecordFactory()* and *setLogRecordFactory()* (see this for the factory's signature).

This functionality can be used to inject your own values into a LogRecord at creation time. You can use the following pattern:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

通过此模式，多个工厂方法可以被链接起来，并且只要它们不重载彼此的属性或是在无意中覆盖了上面列出的标准属性，就不会发生意外。

16.6.7 LogRecord 属性

LogRecord 具有许多属性，它们大多数来自于传递给构造器的形参。（请注意 LogRecord 构造器形参与 LogRecord 属性的名称并不总是完全彼此对应的。）这些属性可被用于将来自记录的数据合并到格式字符串中。下面的表格（按字母顺序）列出了属性名称、它们的含义以及相应的%-style 格式字符串内占位符。

如果是使用 {}-格式化 (*str.format()*)，你可以将 `{attrname}` 用作格式字符串内的占位符。如果是使用 $-格式化 (*string.Template*)，则会使用 `${attrname}` 的形式。当然在这两种情况下，都应当将 `attrname` 替换为你想要使用的实际属性名称。

在 {}-格式化的情况下，你可以在属性名称之后放置指定的格式化旗标，并用冒号来分隔两者。例如，占位符 `{msecs:03d}` 会将毫秒值 `4` 格式化为 `004`。请参看*str.format()* 文档了解你所能使用的选项的完整细节。

属性名称	格式	描述
args	不需要格式化。	合并到 `msg` 以产生 `message` 的包含参数的元组，或是其中的值将被用于合并的字典（当只有一个参数且其类型为字典时）。
asctime	`%(asctime)s`	表示*LogRecord* 何时被创建的供人查看时间值。默认形式为‘2003-07-08 16:49:45,896’(逗号之后的数字为时间的毫秒部分)。
created	`%(created)f`	*LogRecord* 被创建的时间（即*time.time()* 的返回值）。
exc_info	不需要格式化。	异常元组 (例如 `sys.exc_info`) 或者如未发生异常则为 `None`。
filename	`%(filename)s`	`pathname` 的文件名部分。
funcName	`%(funcName)s`	函数名包括调用日志记录.
levelname	`%(levelname)s`	消息文本记录级别 (`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`).
levelno	`%(levelno)s`	消息数字记录级别 (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`).
lineno	`%(lineno)d`	发出日志记录调用所在的源行号（如果可用）。
message	`%(message)s`	记入日志的消息，即 `msg % args` 的结果。这是在发起调用*Formatter.format()* 时设置的。
module 模块	`%(module)s`	模块 (`filename` 的名称部分)。
msecs	`%(msecs)d`	*LogRecord* 被创建的时间的毫秒部分。
msg	不需要格式化。	在原始日志记录调用中传入的格式字符串。与 `args` 合并以产生 `message`，或是一个任意对象 (参见 arbitrary-object-messages)。
名称	`%(name)s`	用于记录调用的日志记录器名称。
pathname	`%(pathname)s`	发出日志记录调用的源文件的完整路径名（如果可用）。
process	`%(process)d`	进程 ID（如果可用）
processName	`%(processName)s`	进程名（如果可用）
relativeCreated	`%(relativeCreated)d`	以毫秒数表示的 LogRecord 被创建的时间，即相对于 logging 模块被加载时间的差值。
stack_info	不需要格式化。	当前线程中从堆栈底部起向上直到包括日志记录调用并导致创建此记录的堆栈帧的堆栈帧信息（如果可用）。
thread	`%(thread)d`	线程 ID（如果可用）
threadName	`%(threadName)s`	线程名（如果可用）

在 3.1 版更改: 添加了 *processName*

16.6.8 LoggerAdapter 对象

LoggerAdapter 实例会被用来方便地将上下文信息传入日志记录调用。要获取用法示例，请参阅 添加上下文信息到你的日志记录输出部分。

class logging.**LoggerAdapter**(*logger*, *extra*)
返回一个*LoggerAdapter* 的实例，该实例的初始化带有一个下层的*Logger* 实例和一个字典类对象。

process(*msg*, *kwargs*)
修改传递给日志记录调用的消息和/或关键字参数以便插入上下文信息。此实现接受以 *extra* 形式传给构造器的对象并使用‘extra’键名将其加入 *kwargs*。返回值为一个 (*msg*, *kwargs*) 元组，其中传入了（可能经过修改的）参数版本。

在上述方法之外，*LoggerAdapter* 还支持*Logger* 的下列方法: *debug()*, *info()*, *warning()*, *error()*, *exception()*, *critical()*, *log()*, *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* 以及*hasHandlers()*。这些方法具有与它们在*Logger* 中的对应方法相同的签名，因此你可以互换使用这两种类型的实例。

在 3.2 版更改: *isEnabledFor()*, *getEffectiveLevel()*, *setLevel()* 和*hasHandlers()* 方法已被添加到*LoggerAdapter*。这些方法会委托给下层的日志记录器。

16.6.9 线程安全

logging 模块的目标是使客户端不必执行任何特殊操作即可确保线程安全。它通过使用线程锁来达成这个目标；用一个锁来序列化对模块共享数据的访问，并且每个处理程序也会创建一个锁来序列化对其下层 I/O 的访问。

如果你要使用*signal* 模块来实现异步信号处理程序，则可能无法在这些处理程序中使用 logging。这是因为*threading* 模块中的锁实现并非总是可重入的，所以无法从此类信号处理程序发起调用。

16.6.10 模块级别函数

在上述的类之外，还有一些模块层级的函数。

logging.**getLogger**(*name=None*)
返回具有指定 name 的日志记录器，或者当 name 为 None 时返回层级结构中作为根日志记录器的日志记录器。如果指定了 name，它通常是以点号分隔的带层级结构的名称如‘*a*’，‘*a.b*’或‘*a.b.c.d*’。这些名称的选择完全取决于使用 logging 的开发者。

附带给定 name 的所有对此函数的调用都将返回相同的日志记录器实例。这意味着日志记录器实例不需要在应用的不同部分间传递。

logging.**getLoggerClass**()
返回标准的*Logger* 类，或是最近传给*setLoggerClass()* 的类。此函数可以从一个新的类定义中调用，以确保安装自定义的*Logger* 类不会撤销其他代码已经应用的自定义操作。例如:

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

logging.**getLogRecordFactory**()
返回一个被用来创建*LogRecord* 的可调用对象。

3.2 新版功能: 此函数与*setLogRecordFactory()* 一起提供，以便允许开发者对如何构造表示日志记录事件的*LogRecord* 有更好的控制。

请参阅*setLogRecordFactory()* 了解有关如何调用该工厂方法的更多信息。

logging.**debug**(*msg*, **args*, ***kwargs*)
在根日志记录器上记录一条 DEBUG 级别的消息。*msg* 是消息格式字符串，而 *args* 是要使用字符串格式化运算符合并到 *msg* 的参数。（请注意这意味着你可以在格式字符串中使用关键字以及单个字典参数。）

在 *kwargs* 中有三个关键字参数会被检查: *exc_info* 参数如果不为假值则会将异常信息添加到日志记录消息。如果提供了异常元组（为*sys.exc_info()* 的返回值格式）或异常实例则它会被使用；在其他情况下，会调用*sys.exc_info()* 以获取异常信息。

第二个可选关键字参数是 *stack_info*，默认为 False。如果为 True，则将堆栈信息添加到日志消息中，包括实际的日志调用。请注意，这与通过指定 *exc_info* 显示的堆栈信息不同：前者是从堆栈底部到当前线程中的日志记录调用的堆栈帧，而后者是在搜索异常处理程序时，跟踪异常而打开的堆栈帧的信息。

您可以独立于 *exc_info* 来指定 *stack_info*，例如，即使在未引发任何异常的情况下，也可以显示如何到达代码中的特定点。堆栈帧在标题行之后打印：

```
Stack (most recent call last):
```

这模仿了显示异常帧时所使用的 Traceback (most recent call last): 。

第三个可选关键字参数是 *extra*，它可被用来传递一个字典，该字典会被用来填充为日志记录事件创建并附带用户自定义属性的 LogRecord 的 __dict__。之后将可按你的需要使用这些自定义属性。例如，可以将它们合并到已记录的消息中。举例来说:

```
FORMAT = '%(asctime)-15s %(clientip)s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning('Protocol problem: %s', 'connection reset', extra=d)
```

应当会打印出这样的内容:

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

The keys in the dictionary passed in *extra* should not clash with the keys used by the logging system. (See the *Formatter* documentation for more information on which keys are used by the logging system.)

如果你选择在已记录的消息中使用这些属性，则需要格外小心。例如在上面的示例中，*Formatter* 已设置了格式字符串，其在 LogRecord 的属性字典中应有‘clientip’和‘user’。如果缺少这些属性，消息将不被记录，因为会引发字符串格式化异常，你始终需要传入带有这些键的 *extra* 字典。

尽管这可能很烦人，但此功能旨在用于特殊情况，例如在多个上下文中执行相同代码的多线程服务器，并且出现的有趣条件取决于此上下文（例如在上面的示例中就是远程客户端 IP 地址和已验证用户名）。在这种情况下，很可能将专门的*Formatter* 与特定的*Handler* 一起使用。

3.2 新版功能: 增加了 *stack_info* 参数。

logging.**info**(*msg*, **args*, ***kwargs*)
在根日志记录器上记录一条 INFO 级别的消息。参数解释同*debug()*。

logging.**warning**(*msg*, **args*, ***kwargs*)
在根日志记录器上记录一条 WARNING 级别的消息。参数解释同*debug()*。

注解: 有一个已过时方法 warn 其功能与 warning 一致。由于 warn 已被弃用，请不要使用它——而要改用 warning。

---

logging.**error**(*msg*, **args*, ***kwargs*)
在根日志记录器上记录一条 ERROR 级别的消息。参数解释同*debug()*.

logging.**critical**(*msg*, **args*, ***kwargs*)
在根日志记录器上记录一条 CRITICAL 级别的消息。参数解释同*debug()*。

logging.**exception**(*msg*, **args*, ***kwargs*)
在根日志记录器上记录一条 ERROR 级别的消息。参数解释同*debug()*。异常信息将被添加到日志消息中。此函数应当仅从异常处理程序中调用。

logging.**log**(*level*, *msg*, **args*, ***kwargs*)
在根日志记录器上记录一条 *level* 级别的消息。其他参数解释同*debug()*。

注解: 上述模块层级的便捷函数均委托给根日志记录器，它们会调用*basicConfig()* 以确保至少有一个处理程序可用。因此，它们 不应在线程中使用，在 Python 2.7.1 和 3.2 之前的版本中，除非线程启动 之前已向根日志记录器添加了至少一个处理程序方可使用。在较早的 Python 版本中，由于*basicConfig()* 中存在线程安全性的不足，这（在少数情况下）可能导致处理程序被多次加入根日志记录器，这会进一步导致同一事件出现多条消息。

---

logging.**disable**(*lvl=CRITICAL*)
Provides an overriding level *lvl* for all loggers which takes precedence over the logger's own level. When the need arises to temporarily throttle logging output down across the whole application, this function can be useful. Its effect is to disable all logging calls of severity *lvl* and below, so that if you call it with a value of INFO, then all INFO and DEBUG events would be discarded, whereas those of severity WARNING and above would be processed according to the logger's effective level. If `logging.disable(logging.NOTSET)` is called, it effectively removes this overriding level, so that logging output again depends on the effective levels of individual loggers.

Note that if you have defined any custom logging level higher than `CRITICAL` (this is not recommended), you won't be able to rely on the default value for the *lvl* parameter, but will have to explicitly supply a suitable value.

在 3.7 版更改: The *lvl* parameter was defaulted to level `CRITICAL`. See Issue #28524 for more information about this change.

logging.**addLevelName**(*lvl*, *levelName*)
Associates level *lvl* with text *levelName* in an internal dictionary, which is used to map numeric levels to a textual representation, for example when a *Formatter* formats a message. This function can also be used to define your own levels. The only constraints are that all levels used must be registered using this function, levels should be positive integers and they should increase in increasing order of severity.

注解: 如果你考虑要定义你自己的级别，请参阅 custom-levels 部分。

---

logging.**getLevelName**(*lvl*)
Returns the textual representation of logging level *lvl*. If the level is one of the predefined levels `CRITICAL`, `ERROR`, `WARNING`, `INFO` or `DEBUG` then you get the corresponding string. If you have associated levels with names using *addLevelName()* then the name you have associated with *lvl* is returned. If a numeric value corresponding to one of the defined levels is passed in, the corresponding string representation is returned. Otherwise, the string 'Level %s' % lvl is returned.

注解: 级别在内部为整数（它们需要在日志记录逻辑中相互比较）。此函数被用于在整数级别与通过 `%(levelname)s` 格式描述符方式格式化的日志输出中显示的级别名称之间进行转换 (参见*LogRecord 属性*)。

在 3.4 版更改: 在早于 3.4 的 Python 版本中，此函数也可传入一个文本级别名称，并将返回对应的级别数字值。此未记入文档的行为被视为是一个错误，并在 Python 3.4 中被移除，但又在 3.4.2 中被恢复以保持向下兼容性。

logging.**makeLogRecord**(*attrdict*)
创建并返回一个新的*LogRecord* 实例，实例属性由 *attrdict* 定义。此函数适用于接受一个封存的*LogRecord* 属性字典，通过套接字传输，并在接收端将其重建为一个*LogRecord* 实例。

logging.**basicConfig**(***kwargs*)
通过使用默认的*Formatter* 创建一个*StreamHandler* 并将其加入根日志记录器来为日志记录系统执行基本配置。如果没有为根日志记录器定义处理程序则*debug()*, *info()*, *warning()*, *error()* 和*critical()* 等函数将自动调用*basicConfig()*。

This function does nothing if the root logger already has handlers configured for it.

注解: 此函数应当在其他线程启动之前从主线程被调用。在 2.7.1 和 3.2 之前的 Python 版本中，如果此函数从多个线程被调用，一个处理程序（在极少的情况下）有可能被多次加入根日志记录器，导致非预期的结果例如日志中的消息出现重复。

支持以下关键字参数。

格式	描述
filename	使用指定的文件名而不是 StreamHandler 创建 FileHandler。
filemode	如果指定了 *filename*，则用此模式 打开该文件。默认模式为 `'a'`。
format	处理器使用的指定格式字符串。
datefmt	使用指定的日期/时间格式，与*time.strftime()* 所接受的格式相同。
style	如果指定了 *format*，将为格式字符串使用此风格。`'%'`, `'{'` 或 `'$'` 分别对应于*printf* 风格, *str.format()* 或*string.Template*。默认为 `'%'`。
level	设置根记录器级别去指定*level*.
stream	使用指定的流初始化 StreamHandler。请注意此参数与 *filename* 是不兼容的 - 如果两者同时存在，则会引发 `ValueError`。
handlers	如果指定，这应为一个包含要加入根日志记录器的已创建处理程序的可迭代对象。任何尚未设置格式描述符的处理程序将被设置为在此函数中创建的默认格式描述符。请注意此参数与 *filename* 或 *stream* 不兼容——如果两者同时存在，则会引发 `ValueError`。

在 3.2 版更改: 增加了 *style* 参数。

在 3.3 版更改: 增加了 *handlers* 参数。增加了额外的检查来捕获指定不兼容参数的情况 (例如同时指定 *handlers* 与 *stream* 或 *filename*，或者同时指定 *stream* 与 *filename*)。

logging.**shutdown**()
通过刷新和关闭所有处理程序来通知日志记录系统执行有序停止。此函数应当在应用退出时被调用并且在此调用之后不应再使用日志记录系统。

logging.**setLoggerClass**(*klass*)
Tells the logging system to use the class *klass* when instantiating a logger. The class should define `__init__()` such that only a name argument is required, and the `__init__()` should call `Logger.__init__()`. This function is typically called before any loggers are instantiated by applications which need to use custom logger behavior.

logging.**setLogRecordFactory**(*factory*)
设置一个用来创建*LogRecord* 的可调用对象。

参数 factory –用来实例化日志记录的工厂可调用对象。

3.2 新版功能: 此函数与*getLogRecordFactory()* 一起提供，以便允许开发者对如何构造表示日志记录事件的*LogRecord* 有更好的控制。

可调用对象 factory 具有如下签名:

```
factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None,
**kwargs)
```

名称 日志记录器名称

level 日志记录级别（数字）。

fn 进行日志记录调用的文件的完整路径名。

lno 记录调用所在文件中的行号。

msg 日志消息。

args 日志记录消息的参数。

exc_info 异常元组，或 None 。

func 调用日志记录调用的函数或方法的名称。

sinfo 与*traceback.print_stack()* 所提供的类似的栈回溯信息，显示调用的层级结构。

kwargs 其他关键字参数。

16.6.11 模块级属性

logging.**lastResort**
通过此属性提供的“最后处理者”。这是一个以 WARNING 级别写入到 sys.stderr 的*StreamHandler*，用于在没有任何日志记录配置的情况下处理日志记录事件。最终结果就是将消息打印到 sys.stderr，这会替代先前形式为 “no handlers could be found for logger XYZ” 的错误消息。如果出于某种原因你需要先前的行为，可将 lastResort 设为 None。

3.2 新版功能.

16.6.12 与警告模块集成

captureWarnings() 函数可用来将*logging* 和*warnings* 模块集成。

logging.**captureWarnings**(*capture*)
此函数用于打开和关闭日志系统对警告的捕获。

如果 *capture* 是 True，则*warnings* 模块发出的警告将重定向到日志记录系统。具体来说，将使用*warnings.formatwarning()* 格式化警告信息，并将结果字符串使用 WARNING 等级记录到名为 'py.warnings' 的记录器中。

如果 *capture* 是 False，则将停止将警告重定向到日志记录系统，并且将警告重定向到其原始目标（即在 captureWarnings(True) 调用之前的有效目标）。

参见:

模块*logging.config* 日志记录模块的配置 API 。

模块*logging.handlers* 日志记录模块附带的有用处理程序。

PEP 282 - Logging 系统 该提案描述了 Python 标准库中包含的这个特性。

Original Python logging package 这是该*logging* 包的原始来源。该站点提供的软件包版本适用于 Python 1.5.2、2.1.x 和 2.2.x，它们不被*logging* 包含在标准库中。

16.7 logging.config —日志记录配置

源代码： Lib/logging/config.py

Important

此页面仅包含参考信息。有关教程，请参阅

- 基础教程
- 进阶教程
- Logging Cookbook

这一节描述了用于配置 logging 模块的 API。

16.7.1 配置函数

下列函数可配置 logging 模块。它们位于*logging.config* 模块中。它们的使用是可选的—要配置 logging 模块你可以使用这些函数，也可以通过调用主 API (在*logging* 本身定义) 并定义在*logging* 或*logging.handlers* 中声明的处理程序。

logging.config.**dictConfig**(*config*)

从一个字典获取日志记录配置。字典的内容描述见下文的配置字典架构。

如果在配置期间遇到错误，此函数将引发*ValueError*, *TypeError*, *AttributeError* 或*ImportError* 并附带适当的描述性消息。下面是将会引发错误的（可能不完整的）条件列表:

- level 不是字符串或者不是对应于实际日志记录级别的字符串。
- propagate 值不是布尔类型。
- id 没有对应的目标。
- 在增量调用期间发现不存在的处理程序 id。
- 无效的日志记录器名称。
- 无法解析为内部或外部对象。

解析由 DictConfigurator 类执行，该类的构造器可传入用于配置的字典，并且具有 configure() 方法。*logging.config* 模块具有可调用属性 dictConfigClass，其初始值设为 DictConfigurator。你可以使用你自己的适当实现来替换 dictConfigClass 的值。

dictConfig() 会调用 dictConfigClass 并传入指定的字典，然后在所返回的对象上调用 configure() 方法以使配置生效:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

例如，DictConfigurator 的子类可以在它自己的 __init__() 中调用 DictConfigurator.__init__()，然后设置可以在后续 configure() 调用中使用的自定义前缀。dictConfigClass 将被绑定到这个新的子类，然后就可以与在默认的未定制状态下完全相同的方式调用*dictConfig()*。

3.2 新版功能.

logging.config.**fileConfig**(*fname*, *defaults=None*, *disable_existing_loggers=True*)

从一个*configparser* 格式文件中读取日志记录配置。文件格式应当与配置文件格式 中的描述一致。此函数可在应用程序中被多次调用，以允许最终用户在多个预设配置中进行选择（如果开发者提供了展示选项并加载选定配置的机制）。

参数

- **fname** –一个文件名，或一个文件类对象，或是一个派生自*RawConfigParser* 的实例。如果传入了一个派生自 RawConfigParser 的实例，它会被原样使用。否则，将会实例化一个 Configparser，并且它会从作为 fname 传入的对象中读取配置。如果存在*readline()* 方法，则它会被当作一个文件类对象并使用*read_file()* 来读取；在其它情况下，它会被当作一个文件名并传递给*read()*。
- **defaults** –要传递给 ConfigParser 的默认值可在此参数中指定。
- **disable_existing_loggers** –If specified as False, loggers which exist when this call is made are left enabled. The default is True because this enables old behaviour in a backward-compatible way. This behaviour is to disable any existing loggers unless they or their ancestors are explicitly named in the logging configuration.

在 3.4 版更改: 现在接受*RawConfigParser* 子类的实例作为 fname 的值。这有助于:

- 使用一个配置文件，其中日志记录配置只是全部应用程序配置的一部分。
- 使用从一个文件读取的配置，它随后会在被传给 fileConfig 之前由使用配置的应用程序来修改（例如基于命令行参数或运行时环境的其他部分）。

logging.config.**listen**(*port=DEFAULT_LOGGING_CONFIG_PORT*, *verify=None*)

在指定的端口上启动套接字服务器，并监听新的配置。如果未指定端口，则会使用模块默认的 DEFAULT_LOGGING_CONFIG_PORT。日志记录配置将作为适合由*dictConfig()* 或*fileConfig()* 进行处理的文件来发送。返回一个*Thread* 实例，你可以在该实例上调用*start()* 来启动服务器，对该服务器你可以在适当的时候执行*join()*。要停止该服务器，请调用*stopListening()*。

如果指定 verify 参数，则它应当是一个可调用对象，该对象应当验证通过套接字接收的字节数据是否有效且应被处理。这可以通过对通过套接字发送的内容进行加密和/或签名来完成，这样 verify 可调用对象就能执行签名验证和/或解密。verify 可调用对象的调用会附带一个参数——通过套接字接收的字节数据——并应当返回要处理的字节数据，或者返回 None 来指明这些字节数据应当被丢弃。返回的字节数据可以与传入的字节数据相同（例如在只执行验证的时候），或者也可以完全不同（例如在可能执行了解密的时候）。

要将配置发送到套接字，请读取配置文件并将其作为字节序列发送到套接字，字节序列要以使用 struct.pack('>L', n) 打包为二进制格式的四字节长度的字符串打头。

注解: 因为配置的各部分是通过*eval()* 传递的，使用此函数可能让用户面临安全风险。虽然此函数仅绑定到 localhost 上的套接字，因此并不接受来自远端机器的连接，但在某些场景中不受信任的代码可以在调用*listen()* 的进程的账户下运行。具体来说，如果如果调用*listen()* 的进程在用户无法彼此信任的多用户机器上运行，则恶意用户就能简单地通过连接到受害者的*listen()* 套接字并发送运行攻击者想在受害者的进程上执行的任何代码的配置的方式，安排运行几乎任意的代码。如果是使用默认端口这会特别容易做到，即便使用了不同端口也不难做到。要避免发生这种情况的风险，请在*listen()* 中使用 verify 参数来防止未经认可的配置被应用。

在 3.4 版更改: 添加了 verify 参数。

注解: 如果你希望将配置发送给未禁用现有日志记录器的监听器，你将需要使用 JSON 格式的配置，该格式将使用*dictConfig()* 进行配置。此方法允许你在你发送的配置中将

disable_existing_loggers 指定为 False。

logging.config.**stopListening**()
停止通过对*listen()* 的调用所创建的监听服务器。此函数的调用通常会先于在*listen()* 的返回值上调用 join()。

16.7.2 配置字典架构

描述日志记录配置需要列出要创建的不同对象及它们之间的连接；例如，你可以创建一个名为‘console’的处理程序，然后名为‘startup’的日志记录器将可以把它的消息发送给‘console’处理程序。这些对象并不仅限于*logging* 模块所提供的对象，因为你还可以编写你自己的格式化或处理程序类。这些类的形参可能还需要包括 sys.stderr 这样的外部对象。描述这些对象和连接的语法会在下面的对象连接 中定义。

字典架构细节

传给*dictConfig()* 的字典必须包含以下的键:

- *version* - 应设为代表架构版本的整数值。目前唯一有效的值是 1，使用此键可允许架构在继续演化的同时保持向下兼容性。

所有其他键都是可选项，但如存在它们将根据下面的描述来解读。在下面提到‘configuring dict’的所有情况下，都将检查它的特殊键 '()' 以确定是否需要自定义实例化。如果需要，则会使用下面用户定义对象 所描述的机制来创建一个实例；否则，会使用上下文来确定要实例化的对象。

- *formatters* - 对应的值将是一个字典，其中每个键是一个格式器 ID 而每个值则是一个描述如何配置相应*Formatter* 实例的字典。

  将在配置字典中搜索键 format 和 datefmt (默认值均为 None) 并且这些键会被用于构造*Formatter* 实例。

- *filters* - 对应的值将是一个字典，其中每个键是一个过滤器 ID 而每个值则是一个描述如何配置相应 Filter 实例的字典。

  将在配置字典中搜索键 name (默认值为空字符串) 并且该键会被用于构造*logging.Filter* 实例。

- *handlers* - 对应的值将是一个字典，其中每个键是一个处理程序 ID 而每个值则是一个描述如何配置相应 Handler 实例的字典。

  将在配置字典中搜索下列键:

  - class (强制)。这是处理程序类的完整限定名称。
  - level (可选)。处理程序的级别。
  - formatter (可选)。处理程序所对应格式化器的 ID。
  - filters (可选)。由处理程序所对应过滤器的 ID 组成的列表。

  所有 其他键会被作为关键字参数传递给处理程序类的构造器。例如，给定如下代码段:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
```

(下页继续)

(续上页)

```
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

ID 为 `console` 的处理程序会被实例化为*`logging.StreamHandler`*，并使用 `sys.stdout` 作为下层流。ID 为 `file` 的处理程序会被实例化为*`logging.handlers.RotatingFileHandler`*，并附带关键字参数 `filename='logconfig.log', maxBytes=1024, backupCount=3`。

- *loggers* - 对应的值将是一个字典，其中每个键是一个日志记录器名称而每个值则是一个描述如何配置相应 Logger 实例的字典。

  将在配置字典中搜索下列键:

  - `level` (可选)。日志记录器的级别。
  - `propagate` (可选)。日志记录器的传播设置。
  - `filters` (可选)。由日志记录器对应过滤器的 ID 组成的列表。
  - `handlers` (可选)。由日志记录器对应处理程序的 ID 组成的列表。

  指定的日志记录器将根据指定的级别、传播、过滤器和处理程序来配置。

- *root* - 这将成为根日志记录器对应的配置。配置的处理方式将与所有日志记录器一致，除了 `propagate` 设置将不可用之外。

- *incremental* - 配置是否要被解读为在现有配置上新增。该值默认为 `False`，这意味着指定的配置将以与当前*`fileConfig()`* API 所使用的相同语义来替代现有的配置。

  如果指定的值为 `True`，配置会按照增量配置 部分所描述的方式来处理。

- *disable_existing_loggers* - whether any existing loggers are to be disabled. This setting mirrors the parameter of the same name in *`fileConfig()`*. If absent, this parameter defaults to `True`. This value is ignored if *incremental* is `True`.

增量配置

为增量配置提供完全的灵活性是很困难的。例如，由于过滤器和格式化器这样的对象是匿名的，一旦完成配置，在增加配置时就不可能引用这些匿名对象。

此外，一旦完成了配置，在运行时任意改变日志记录器、处理程序、过滤器、格式化器的对象图就不是很有必要；日志记录器和处理程序的详细程度只需通过设置级别即可实现控制（对于日志记录器则可设置传播旗标）。在多线程环境中以安全的方式任意改变对象图也许会导致问题；虽然并非不可能，但这样做的好处不足以抵销其所增加的实现复杂度。

这样，当配置字典的 `incremental` 键存在且为 `True` 时，系统将完全忽略任何 `formatters` 和 `filters` 条目，并仅会处理 `handlers` 条目中的 `level` 设置，以及 `loggers` 和 `root` 条目中的 `level` 和 `propagate` 设置。

使用配置字典中的值可让配置以封存字典对象的形式通过线路传送给套接字监听器。这样，长时间运行的应用程序的日志记录的详细程度可随时间改变而无须停止并重新启动应用程序。

对象连接

该架构描述了一组日志记录对象——日志记录器、处理程序、格式化器、过滤器——它们在对象图中彼此连接。因此，该架构需要能表示对象之间的连接。例如，在配置完成后，一个特定的日志记录器关联到了一个特定的处理程序。出于讨论的目的，我们可以说该日志记录器代表两者间连接的源头，而处理程序则代表对应的目标。当然在已配置对象中这是由包含对处理程序的引用的日志记录器来代表的。在配置字典中，这是通过给每个目标对象一个 ID 来无歧义地标识它，然后在源头对象中使用该 ID 来实现的。

因此，举例来说，考虑以下 YAML 代码段:

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

(注：这里使用 YAML 是因为它的可读性比表示字典的等价 Python 源码形式更好。)

日志记录器 ID 就是日志记录器的名称，它会在程序中被用来获取对日志记录器的引用，例如 `foo.bar.baz`。格式化器和过滤器的 ID 可以是任意字符串值 (例如上面的 `brief`, `precise`) 并且它们是瞬态的，因为它们仅对处理配置字典有意义并会被用来确定对象之间的连接，而当配置调用完成时不会在任何地方保留。

上面的代码片段指明名为 `foo.bar.baz` 的日志记录器应当关联到两个处理程序，它们的 ID 是 `h1` 和 `h2`。`h1` 的格式化器的 ID 是 `brief`，而 `h2` 的格式化器的 ID 是 `precise`。

用户定义对象

此架构支持用户定义对象作为处理程序、过滤器和格式化器。(日志记录器的不同实例不需要具有不同类型，因此这个配置架构并不支持用户定义日志记录器类。)

要配置的对象是由字典描述的，其中包含它们的配置详情。在某些地方，日志记录系统将能够从上下文中推断出如何实例化一个对象，但是当要实例化一个用户自定义对象时，系统将不知道要如何做。为了提供用户自定义对象实例化的完全灵活性，用户需要提供一个‘工厂’函数——即在调用时传入配置字典并返回实例化对象的可调用对象。这是用一个通过特殊键 `'()'` 来访问的工厂函数的绝对导入路径来标示的。下面是一个实际的例子:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
```

(下页继续)

(续上页)

```
      spam: 99.9
      answer: 42
```

上面的 YAML 代码片段定义了三个格式化器。第一个的 ID 为 brief，是带有特殊格式字符串的标准*logging.Formatter* 实例。第二个的 ID 为 default，具有更长的格式同时还显式地定义了时间格式，并将最终实例化一个带有这两个格式字符串的*logging.Formatter*。以 Python 源代码形式显示的 brief 和 default 格式化器具有下列配置子字典:

```
{
  'format' : '%(message)s'
}
```

和:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

并且由于这些字典不包含特殊键 '()'，实例化方式是从上下文中推断出来的：结果会创建标准的*logging.Formatter* 实例。第三个格式器的 ID 为 custom，对应配置子字典为:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

并且它包含特殊键 '()'，这意味着需要用户自定义实例化方式。在此情况下，将使用指定的工厂可调用对象。如果它本身就是一个可调用对象则将被直接使用——否则如果你指定了一个字符串（如这个例子所示）则将使用正常的导入机制来定位实例的可调用对象。调用该可调用对象将传入配置子字典中 **剩余的**条目作为关键字参数。在上面的例子中，调用将预期返回 ID 为 custom 的格式化器:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

将 '()' 用作特殊键是因为它不是一个有效的关键字形参名称，这样就不会与调用中使用的关键字参数发生冲突。'()' 还被用作表明对应值为可调用对象的助记符。

访问外部对象

有时一个配置需要引用配置以外的对象，例如 sys.stderr。如果配置字典是使用 Python 代码构造的，这会很直观，但是当配置是通过文本文件（例如 JSON, YAML）提供的时候就会引发问题。在一个文本文件中，没有将 sys.stderr 与字符串字面值 'sys.stderr' 区分开来的标准方式。为了实现这种区分，配置系统会在字符串值中查找规定的特殊前缀并对其做特殊处理。例如，如果在配置中将字符串字面值 'ext://sys.stderr' 作为一个值来提供，则 ext:// 将被去除而该值的剩余部分将使用正常导入机制来处理。

此类前缀的处理方式类似于协议处理：存在一种通用机制来查找与正则表达式 ^(?P<prefix>[a-z]+)://(?P<suffix>.*)$ 相匹配的前缀，如果识别出了 prefix，则 suffix 会以与前缀相对应的方式来处理并且处理的结果将替代原字符串值。如果未识别出前缀，则原字符串将保持不变。

访问内部对象

除了外部对象，有时还需要引用配置中的对象。这将由配置系统针对它所了解的内容隐式地完成。例如，在日志记录器或处理程序中表示 `level` 的字符串值 `'DEBUG'` 将被自动转换为值 `logging.DEBUG`，而 `handlers`, `filters` 和 `formatter` 条目将接受一个对象 ID 并解析为适当的目标对象。

但是，对于*logging* 模块所不了解的用户自定义对象则需要一种更通用的机制。例如，考虑*logging.handlers.MemoryHandler*，它接受一个 `target` 参数即其所委托的另一个处理程序。由于系统已经知道存在该类，因而在配置中，给定的 `target` 只需为相应目标处理程序的的对象 ID 即可，而系统将根据该 ID 解析出处理程序。但是，如果用户定义了一个具有 `alternate` 处理程序的 `my.package.MyHandler`，则配置程序将不知道 `alternate` 指向的是一个处理程序。为了应对这种情况，通用解析系统允许用户指定:

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

字符串字面值 `'cfg://handlers.file'` 将按照与 `ext://` 前缀类似的方式被解析为结果字符串，但查找操作是在配置自身而不是在导入命名空间中进行。该机制允许按点号或按索引来访问，与 `str.format` 所提供的方式类似。这样，给定以下代码段:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

在该配置中，字符串 `'cfg://handlers'` 将解析为带有 `handlers` 键的字典，字符串 `'cfg://handlers.email` 将解析为 `handlers` 字典中带有 `email` 键的字典，依此类推。字符串 `'cfg://handlers.email.toaddrs[1]` 将解析为 `'dev_team.domain.tld'` 而字符串 `'cfg://handlers.email.toaddrs[0]'` 将解析为值 `'support_team@domain.tld'`。`subject` 值可以使用 `'cfg://handlers.email.subject'` 或者等价的 `'cfg://handlers.email[subject]'` 来访问。后一种形式仅在键包含空格或非字母数字类字符的情况下才需要使用。如果一个索引仅由十进制数码构成，则将尝试使用相应的整数值来访问，如果有必要则将回退为字符串值。

给定字符串 `cfg://handlers.myhandler.mykey.123`，这将解析为 `config_dict['handlers']['myhandler']['mykey']['123']`。如果字符串被指定为 `cfg://handlers.myhandler.mykey[123]`，系统将尝试从 `config_dict['handlers']['myhandler']['mykey'][123]` 中提取值，并在尝试失败时回退为 `config_dict['handlers']['myhandler']['mykey']['123']`。

导入解析与定制导入器

导入解析默认使用内置的*__import__()* 函数来执行导入。你可能想要将其替换为你自己的导入机制：如果是这样的话，你可以替换 DictConfigurator 或其超类 BaseConfigurator 类的 importer 属性。但是你必须小心谨慎，因为函数是从类中通过描述器方式来访问的。如果你使用 Python 可调用对象来执行导入，并且你希望在类层级而不是在实例层级上定义它，则你需要用*staticmethod()* 来装饰它。例如:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

如果你是在一个配置器的 实例上设置导入可调用对象则你不需要用*staticmethod()* 来装饰。

16.7.3 配置文件格式

fileConfig() 所能理解的配置文件格式是基于*configparser* 功能的。该文件必须包含 [loggers], [handlers] 和 [formatters] 等小节，它们通过名称来标识文件中定义的每种类型的实体。对于每个这样的实体，都有单独的小节来标识实体的配置方式。因此，对于 [loggers] 小节中名为 log01 的日志记录器，相应的配置详情保存在 [logger_log01] 小节中。类似地，对于 [handlers] 小节中名为 hand01 的处理程序，其配置将保存在名为 [handler_hand01] 的小节中，而对于 [formatters] 小节中名为 form01 的格式化器，其配置将在名为 [formatter_form01] 的小节中指定。根日志记录器的配置必须在名为 [logger_root] 的小节中指定。

注解: *fileConfig()* API 比*dictConfig()* API 更旧因而没有提供涵盖日志记录特定方面的功能。例如，你无法配置*Filter* 对象，该对象使用*fileConfig()* 提供超出简单整数级别的消息过滤功能。如果你想要在你的日志记录配置中包含*Filter* 的实例，你将必须使用*dictConfig()*。请注意未来还将向*dictConfig()* 添加对配置功能的强化，因此值得考虑在方便的时候转换到这个新 API。

在文件中这些小节的例子如下所示。

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

根日志记录器必须指定一个级别和一个处理程序列表。根日志小节的例子如下所示。

```
[logger_root]
level=NOTSET
handlers=hand01
```

level 条目可以为 DEBUG, INFO, WARNING, ERROR, CRITICAL 或 NOTSET 之一。其中 NOTSET 仅适用于根日志记录器，表示将会记录所有消息。级别值会在 logging 包命名空间的上下文中通过*eval()* 来得出。

handlers 条目是以逗号分隔的处理程序名称列表，它必须出现于 [handlers] 小节并且在配置文件中有相应的小节。

对于根日志记录器以外的日志记录器，还需要某些附加信息。下面的例子演示了这些信息。

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

level 和 handlers 条目的解释方式与根日志记录器的一致，不同之处在于如果一个非根日志记录器的级别被指定为 NOTSET，则系统会咨询更高层级的日志记录器来确定该日志记录器的有效级别。propagate 条目设为 1 表示消息必须从此日志记录器传播到更高层级的处理程序，设为 0 表示消息 **不会**传播到更高层级的处理程序。qualname 条目是日志记录器的层级通道名称，也就是应用程序获取日志记录器所用的名称。

指定处理程序配置的小节说明如下。

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

class 条目指明处理程序的类（由 logging 包命名空间中的*eval()* 来确定）。level 会以与日志记录器相同的方式来解读，NOTSET 会被视为表示‘记录一切消息’。

formatter 条目指明此处理程序的格式化器的键名称。如为空白，则会使用默认的格式化器 (logging._defaultFormatter)。如果指定了名称，则它必须出现于 [formatters] 小节并且在配置文件中有相应的小节。

The args entry, when *eval()*uated in the context of the logging package’s namespace, is the list of arguments to the constructor for the handler class. Refer to the constructors for the relevant handlers, or to the examples below, to see how typical entries are constructed.

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)

[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
```

(下页继续)

(续上页)

```
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
```

指定格式化器配置的小节说明如下。

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

`format` 是整个格式字符串，而 `datefmt` 条目则是兼容 `strftime()` 的日期/时间格式字符串。如果为空，此包将替换任何接近于日期格式字符串 `'%Y-%m-%d %H:%M:%S'` 的内容。此格式还指定了毫秒，并使用逗号分隔符将其附加到结果当中。此格式的时间示例如 `2003-01-23 00:29:50,411`。

`class` 条目是可选的。它指明格式化器类的名称（形式为带点号的模块名加类名。）此选项适用于实例化 *Formatter* 的子类。*Formatter* 的子类可通过扩展或收缩格式来显示异常回溯信息。

注解： 由于如上所述使用了 *eval()*，因此使用 *listen()* 通过套接字来发送和接收配置会导致潜在的安全风险。此风险仅限于相互间没有信任的多个用户在同一台机器上运行代码的情况；请参阅 *listen()* 了解更多信息。

参见：

模块 *logging* 日志记录模块的 API 参考。

模块 *logging.handlers* 日志记录模块附带的有用处理程序。

16.8 logging.handlers —日志处理

源代码: Lib/logging/handlers.py

Important

此页面仅包含参考信息。有关教程，请参阅

- 基础教程
- 进阶教程
- Logging Cookbook

这个包提供了以下有用的处理程序。请注意有三个处理程序类 (*StreamHandler*, *FileHandler* 和*NullHandler*) 实际上是在*logging* 模块本身定义的，但其文档与其他处理程序一同记录在此。

16.8.1 StreamHandler

StreamHandler 类位于核心*logging* 包，它可将日志记录输出发送到数据流例如 *sys.stdout*, *sys.stderr* 或任何文件类对象（或者更精确地说，任何支持 `write()` 和 `flush()` 方法的对象）。

class logging.**StreamHandler**(*stream=None*)

返回一个新的*StreamHandler* 类。如果指定了 *stream*，则实例将用它作为日志记录输出；在其他情况下将使用 *sys.stderr*。

emit(*record*)

If a formatter is specified, it is used to format the record. The record is then written to the stream with a terminator. If exception information is present, it is formatted using *traceback.print_exception()* and appended to the stream.

flush()

通过调用流的*flush()* 方法来刷新它。请注意 `close()` 方法是继承自*Handler* 的所以没有输出，因此有时可能需要显式地调用*flush()*。

在 3.2 版更改: The `StreamHandler` class now has a `terminator` attribute, default value `'\n'`, which is used as the terminator when writing a formatted record to a stream. If you don't want this newline termination, you can set the handler instance's `terminator` attribute to the empty string. In earlier versions, the terminator was hardcoded as `'\n'`.

16.8.2 FileHandler

FileHandler 类位于核心*logging* 包，它可将日志记录输出到磁盘文件中。它从*StreamHandler* 继承了输出功能。

class logging.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*)

Returns a new instance of the *FileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely.

在 3.6 版更改: 除了字符串值，也接受*Path* 对象作为 *filename* 参数。

close()

关闭文件。

emit(*record*)
将记录输出到文件。

16.8.3 NullHandler

3.1 新版功能.

NullHandler 类位于核心*logging* 包，它不执行任何格式化或输出。它实际上是一个供库开发者使用的‘无操作’处理程序。

class logging.**NullHandler**
返回一个*NullHandler* 类的新实例。

emit(*record*)
此方法不执行任何操作。

handle(*record*)
此方法不执行任何操作。

createLock()
此方法会对锁返回 None，因为没有下层 I/O 的访问需要被序列化。

请参阅 library-config 了解有关如何使用*NullHandler* 的更多信息。

16.8.4 WatchedFileHandler

WatchedFileHandler 类位于*logging.handlers* 模块，这个 FileHandler 用于监视它所写入日志记录的文件。如果文件发生变化，它会被关闭并使用文件名重新打开。

发生文件更改可能是由于使用了执行文件轮换的程序例如 *newsyslog* 和 *logrotate*。这个处理程序在 Unix/Linux 下使用，它会监视文件来查看自上次发出数据后是否有更改。（如果文件的设备或 inode 发生变化就认为已被更改。）如果文件被更改，则会关闭旧文件流，并再打开文件以获得新文件流。

这个处理程序不适合在 Windows 下使用，因为在 Windows 下打开的日志文件无法被移动或改名——日志记录会使用排他的锁来打开文件——因此这样的处理程序是没有必要的。此外，*ST_INO* 在 Windows 下不受支持；*stat()* 将总是为该值返回零。

class logging.handlers.**WatchedFileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*)
Returns a new instance of the *WatchedFileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, 'a' is used. If *encoding* is not None, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely.

在 3.6 版更改: 除了字符串值，也接受*Path* 对象作为 *filename* 参数。

reopenIfNeeded()
检查文件是否已更改。如果已更改，则会刷新并关闭现有流然后重新打开文件，这通常是将记录输出到文件的先导操作。

3.6 新版功能.

emit(*record*)
将记录输出到文件，但如果文件已更改则会先调用*reopenIfNeeded()* 来重新打开它。

16.8.5 BaseRotatingHandler

BaseRotatingHandler 类位于*logging.handlers* 模块中，它是轮换文件处理程序类*RotatingFileHandler* 和*TimedRotatingFileHandler* 的基类。你不需要实例化此类，但它具有你可能需要重载的属性和方法。

class logging.handlers.**BaseRotatingHandler**(*filename*, *mode*, *encoding=None*, *delay=False*)

类的形参与 FileHandler 的相同。其属性有:

namer

如果此属性被设为一个可调用对象，则*rotation_filename()* 方法会委托给该可调用对象。传给该可调用对象的形参与传给*rotation_filename()* 的相同。

注解: namer 函数会在轮换期间被多次调用，因此它应当尽可能的简单快速。它还应当对给定的输入每次都返回相同的输出，否则轮换行为可能无法按预期工作。

3.3 新版功能.

rotator

如果此属性被设为一个可调用对象，则*rotate()* 方法会委托给该可调用对象。传给该可调用对象的形参与传给*rotate()* 的相同。

3.3 新版功能.

rotation_filename(*default_name*)

当轮换时修改日志文件的文件名。

提供该属性以便可以提供自定义文件名。

默认实现会调用处理程序的‘namer’属性，如果它是可调用对象的话，并传给它默认的名称。如果该属性不是可调用对象 (默认值为 None)，则将名称原样返回。

参数 default_name –日志文件的默认名称。

3.3 新版功能.

rotate(*source*, *dest*)

当执行轮换时，轮换当前日志。

默认实现会调用处理程序的‘rotator’属性，如果它是可调用对象的话，并传给它 source 和 dest 参数。如果该属性不是可调用对象 (默认值为 None)，则将源简单地重命名为目标。

参数

- **source** –源文件名。这通常为基本文件名，例如‘test.log’。
- **dest** –目标文件名。这通常是源被轮换后的名称，例如‘test.log.1’。

3.3 新版功能.

该属性存在的理由是让你不必进行子类化—— 你可以使用与*RotatingFileHandler* 和*TimedRotatingFileHandler* 的实例相同的可调用对象。如果 namer 或 rotator 可调用对象引发了异常，将会按照与 emit() 调用期间的任何其他异常相同的方式来处理，例如通过处理程序的 handleError() 方法。

如果你需要对轮换进程执行更多的修改，你可以重载这些方法。

请参阅 cookbook-rotator-namer 获取具体示例。

16.8.6 RotatingFileHandler

RotatingFileHandler 类位于 *logging.handlers* 模块，它支持磁盘日志文件的轮换。

class logging.handlers.**RotatingFileHandler**(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*)

Returns a new instance of the *RotatingFileHandler* class. The specified file is opened and used as the stream for logging. If *mode* is not specified, `'a'` is used. If *encoding* is not `None`, it is used to open the file with that encoding. If *delay* is true, then file opening is deferred until the first call to *emit()*. By default, the file grows indefinitely.

你可以使用 *maxBytes* 和 *backupCount* 值来允许文件以预定的大小执行 *rollover*。当即将超出预定大小时，将关闭旧文件并打开一个新文件用于输出。只要当前日志文件长度接近 *maxBytes* 就会发生轮换；但是如果 *maxBytes* 或 *backupCount* 两者之一的值为零，就不会发生轮换，因此你通常要设置 *backupCount* 至少为 1，而 *maxBytes* 不能为零。当 *backupCount* 为非零值时，系统将通过为原文件名添加扩展名 '.1'，'.2' 等来保存旧日志文件。例如，当 *backupCount* 为 5 而基本文件名为 `app.log` 时，你将得到 `app.log`, `app.log.1`, `app.log.2` 直至 `app.log.5`。当前被写入的文件总是 `app.log`。当此文件写满时，它会被关闭并重户名为 `app.log.1`，而如果文件 `app.log.1`, `app.log.2` 等存在，则它们会被分别重命名为 `app.log.2`, `app.log.3` 等等。

在 3.6 版更改: 除了字符串值，也接受 *Path* 对象作为 *filename* 参数。

doRollover()

执行上文所描述的轮换。

emit(*record*)

将记录输出到文件，以适应上文所描述的轮换。

16.8.7 TimedRotatingFileHandler

TimedRotatingFileHandler 类位于 *logging.handlers* 模块，它支持基于特定时间间隔的磁盘日志文件轮换。

class logging.handlers.**TimedRotatingFileHandler**(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*)

返回一个新的 *TimedRotatingFileHandler* 类实例。指定的文件会被打开并用作日志记录的流。对于轮换操作它还会设置文件名前缀。轮换的发生是基于 *when* 和 *interval* 的积。

你可以使用 *when* 来指定 *interval* 的类型。可能的值列表如下。请注意它们不是大小写敏感的。

值	间隔类型	如果/如何使用 *atTime*
`'S'`	秒	忽略
`'M'`	分钟	忽略
`'H'`	小时	忽略
`'D'`	天	忽略
`'W0'-'W6'`	工作日 (0= 星期一)	用于计算初始轮换时间
`'midnight'`	如果未指定 *atTime* 则在午夜执行轮换，否则将使用 *atTime*。	用于计算初始轮换时间

当使用基于星期的轮换时，星期一为 'W0'，星期二为 'W1'，以此类推直至星期日为 'W6'。在这种情况下，传入的 *interval* 值不会被使用。

系统将通过为文件名添加扩展名来保存旧日志文件。扩展名是基于日期和时间的，根据轮换间隔的长短使用 strftime 格式 `%Y-%m-%d_%H-%M-%S` 或是其中有变动的部分。

当首次计算下次轮换的时间时（即当处理程序被创建时），现有日志文件的上次被修改时间或者当前时间会被用来计算下次轮换的发生时间。

如果 *utc* 参数为真值，将使用 UTC 时间；否则会使用本地时间。

如果 *backupCount* 不为零，则最多将保留 *backupCount* 个文件，而如果当轮换发生时创建了更多的文件，则最旧的文件会被删除。删除逻辑使用间隔时间来确定要删除的文件，因此改变间隔时间可能导致旧文件被继续保留。

如果 *delay* 为真值，则会将文件打开延迟到首次调用*emit()* 的时候。

如果 *atTime* 不为 None，则它必须是一个 datetime.time 的实例，该实例指定轮换在一天内的发生时间，用于轮换被设为“在午夜”或“在每星期的某一天”之类的情况。请注意在这些情况下，*atTime* 值实际上会被用于计算*初始*轮换，而后续轮换将会通过正常的间隔时间计算来得出。

注解： 初始轮换时间的计算是在处理程序被初始化时执行的。后续轮换时间的计算则仅在轮换发生时执行，而只有当提交输出时轮换才会发生。如果不记住这一点，你就可能会感到困惑。例如，如果设置时间间隔为“每分钟”，那并不意味着你总会看到（文件名中）带有间隔一分钟时间的日志文件；如果在应用程序执行期间，日志记录输出的生成频率高于每分钟一次，*那么*你可以预期看到间隔一分钟时间的日志文件。另一方面，如果（假设）日志记录消息每五分钟才输出一次，那么文件时间将会存在对应于没有输出（因而没有轮换）的缺失。

在 3.4 版更改: 添加了 *atTime* 形参。

在 3.6 版更改: 除了字符串值，也接受*Path* 对象作为 *filename* 参数。

doRollover()
执行上文所描述的轮换。

emit(*record*)
将记录输出到文件，以适应上文所描述的轮换。

16.8.8 SocketHandler

SocketHandler 类位于*logging.handlers* 模块，它会将日志记录输出发送到网络套接字。基类所使用的是 TCP 套接字。

class logging.handlers.**SocketHandler**(*host*, *port*)
返回一个*SocketHandler* 类的新实例，该实例旨在与使用 *host* 与 *port* 给定地址的远程主机进行通信。

在 3.4 版更改: 如果 port 指定为 None，会使用 host 中的值来创建一个 Unix 域套接字——在其他情况下，则会创建一个 TCP 套接字。

close()
关闭套接字。

emit()
对记录的属性字典执行封存并以二进制格式将其写入套接字。如果套接字存在错误，则静默地丢弃数据包。如果连接在此之前丢失，则重新建立连接。要在接收端将记录解封并输出到*LogRecord*，请使用*makeLogRecord()* 函数。

handleError()
处理在*emit()* 期间发生的错误。最可能的原因是连接丢失。关闭套接字以便我们能在下次事件时重新尝试。

makeSocket()
这是一个工厂方法，它允许子类定义它们想要的套接字的准确类型。默认实现会创建一个 TCP 套接字 (*socket.SOCK_STREAM*)。

makePickle(*record*)
Pickles the record's attribute dictionary in binary format with a length prefix, and returns it ready for transmission across the socket.

请注意封存操作不是绝对安全的。如果你关心安全问题，你可能会想要重载此方法以实现更安全的机制。例如，你可以使用 HMAC 对封存对象进行签名然后在接收端验证它们，或者你也可以在接收端禁用全局对象的解封操作。

send(*packet*)
Send a pickled string *packet* to the socket. This function allows for partial sends which can happen when the network is busy.

createSocket()
尝试创建一个套接字；失败时将使用指数化回退算法处理。在失败初次发生时，处理程序将丢弃它正尝试发送的消息。当后续消息交由同一实例处理时，它将不会尝试连接直到经过一段时间以后。默认形参设置为初始延迟一秒，如果在延迟之后连接仍然无法建立，处理程序将每次把延迟翻倍直至达到 30 秒的最大值。

此行为由下列处理程序属性控制:

- `retryStart` (初始延迟，默认为 1.0 秒)。
- `retryFactor` (倍数，默认为 2.0)。
- `retryMax` (最大延迟，默认为 30.0 秒)。

这意味着如果远程监听器在处理程序被使用 之后启动，你可能会丢失消息（因为处理程序在延迟结束之前甚至不会尝试连接，而在延迟期间静默地丢弃消息）。

16.8.9 DatagramHandler

DatagramHandler 类位于*logging.handlers* 模块，它继承自*SocketHandler*，支持通过 UDP 套接字发送日志记录消息。

class logging.handlers.**DatagramHandler**(*host*, *port*)
返回一个*DatagramHandler* 类的新实例，该实例旨在与使用 *host* 与 *port* 给定地址的远程主机进行通信。

在 3.4 版更改: 如果 `port` 指定为 `None`，会使用 `host` 中的值来创建一个 Unix 域套接字——在其他情况下，则会创建一个 UDP 套接字。

emit()
对记录的属性字典执行封存并以二进制格式将其写入套接字。如果套接字存在错误，则静默地丢弃数据包。要在接收端将记录解封并输出到*LogRecord*，请使用*makeLogRecord()* 函数。

makeSocket()
SocketHandler 的工厂方法会在此被重载以创建一个 UDP 套接字 (*socket.SOCK_DGRAM*)。

send(*s*)
Send a pickled string to a socket.

16.8.10 SysLogHandler

SysLogHandler 类位于*logging.handlers* 模块，它支持将日志记录消息发送到远程或本地 Unix syslog。

class logging.handlers.**SysLogHandler**(*address=('localhost', SYSLOG_UDP_PORT), facility=LOG_USER, socktype=socket.SOCK_DGRAM*)

返回一个*SysLogHandler* 类的新实例用来与通过 *address* 以 (host, port) 元组形式给出地址的远程 Unix 机器进行通讯。如果未指定 *address*，则使用 ('localhost', 514)。该地址会被用于打开套接字。提供 (host, port) 元组的一种替代方式是提供字符串形式的地址，例如‘/dev/log’。在这种情况下，会使用 Unix 域套接字将消息发送到 syslog。如果未指定 *facility*，则使用 LOG_USER。打开的套接字类型取决于 *socktype* 参数，该参数的默认值为*socket.SOCK_DGRAM* 即打开一个 UDP 套接字。要打开一个 TCP 套接字（用来配合较新的 syslog 守护程序例如 rsyslog 使用），请指定值为*socket.SOCK_STREAM*。

请注意如果你的服务器不是在 UDP 端口 514 上进行侦听，则*SysLogHandler* 可能无法正常工作。在这种情况下，请检查你应当为域套接字所使用的地址——它依赖于具体的系统。例如，在 Linux 上通常为‘/dev/log’而在 OS/X 上则为‘/var/run/syslog’。你需要检查你的系统平台并使用适当的地址（如果你的应用程序需要在多个平台上运行则可能需要在运行时进行这样的检查）。在 Windows 上，你大概必须要使用 UDP 选项。

在 3.2 版更改: 添加了 *socktype*。

close()

关闭连接远程主机的套接字。

emit(*record*)

记录会被格式化，然后发送到 syslog 服务器。如果存在异常信息，则它 不会被发送到服务器。

在 3.2.1 版更改: (参见: bpo-12168。) 在较早的版本中，发送至 syslog 守护程序的消息总是以一个 NUL 字节结束，因为守护程序的早期版本期望接收一个以 NUL 结束的消息——即使它不包含于对应的规范说明 (**RFC 5424**)。这些守护程序的较新版本不再期望接收 NUL 字节，如果其存在则会将其去除，而最新的守护程序（更紧密地遵循 RFC 5424）会将 NUL 字节作为消息的一部分传递出去。

为了在面对所有这些不同守护程序行为时能够更方便地处理 syslog 消息，通过使用类层级属性 append_nul，添加 NUL 字节的操作已被作为可配置项。该属性默认为 True (保留现有行为) 但可在 SysLogHandler 实例上设为 False 以便让实例 不会添加 NUL 结束符。

在 3.3 版更改: (参见: bpo-12419。) 在较早的版本中，没有“ident”或“tag”前缀工具可以用来标识消息的来源。现在则可以使用一个类层级属性来设置它，该属性默认为 "" 表示保留现有行为，但可在 SysLogHandler 实例上重载以便让实例不会为所处理的每条消息添加标识。请注意所提供的标识必须为文本而非字节串，并且它会被原封不动地添加到消息中。

encodePriority(*facility*, *priority*)

将功能和优先级编码为一个整数。你可以传入字符串或者整数——如果传入的是字符串，则会使用内部的映射字典将其转换为整数。

符号 LOG_ 的值在*SysLogHandler* 中定义并且是 sys/syslog.h 头文件中所定义值的镜像。

优先级

名称（字符串）	符号值
alert	LOG_ALERT
crit 或 critical	LOG_CRIT
debug	LOG_DEBUG
emerg 或 panic	LOG_EMERG
err 或 error	LOG_ERR
info	LOG_INFO
notice	LOG_NOTICE
warn 或 warning	LOG_WARNING

设备

名称（字符串）	符号值
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority(*levelname*)

将日志记录级别名称映射到 syslog 优先级名称。如果你使用自定义级别，或者如果默认算法不适合你的需要，你可能需要重载此方法。默认算法将 DEBUG, INFO, WARNING, ERROR 和 CRITICAL 映射到等价的 syslog 名称，并将所有其他级别名称映射到‘warning’。

16.8.11 NTEventLogHandler

NTEventLogHandler 类位于 *logging.handlers* 模块，它支持将日志记录消息发送到本地 Windows NT, Windows 2000 或 Windows XP 事件日志。在你使用它之前，你需要安装 Mark Hammond 的 Python Win32 扩展。

class logging.handlers.**NTEventLogHandler**(*appname*, *dllname=None*, *logtype='Application'*)

返回一个 *NTEventLogHandler* 类的新实例。*appname* 用来定义出现在事件日志中的应用名称。将使用此名称创建适当的注册表条目。*dllname* 应当给出要包含在日志中的消息定义的.dll 或.exe 的完整限定路径名称（如未指定则会使用 'win32service.pyd' ——此文件随 Win32 扩展安装且包含一些基本的消息定义占位符。请注意使用这些占位符将使你的事件日志变得很大，因为整个消息源都会被放入日志。如果你希望有较小的日志，你必须自行传入包含你想要在事件日志中使用的消息定义的.dll 或.exe

名称)。*logtype* 为 'Application', 'System' 或 'Security' 之一，且默认值为 'Application'。

close()
这时，你就可以从注册表中移除作为事件日志条目来源的应用名称。但是，如果你这样做，你将无法如你所预期的那样在事件日志查看器中看到这些事件——它必须能访问注册表来获取.dll 名称。当前版本并不会这样做。

emit(*record*)
确定消息 ID，事件类别和事件类型，然后将消息记录到 NT 事件日志中。

getEventCategory(*record*)
返回记录的事件类别。如果你希望指定你自己的类别就要重载此方法。此版本将返回 0。

getEventType(*record*)
返回记录的事件类型。如果你希望指定你自己的类型就要重载此方法。此版本将使用处理程序的 typemap 属性来执行映射，该属性在 __init__() 被设置为一个字典，其中包含 DEBUG, INFO, WARNING, ERROR 和 CRITICAL 的映射。如果你使用你自己的级别，你将需要重载此方法或者在处理程序的 *typemap* 属性中放置一个合适的字典。

getMessageID(*record*)
返回记录的消息 ID。如果你使用你自己的消息，你可以通过将 *msg* 传给日志记录器作为 ID 而非格式字符串实现此功能。然后，你可以在这里使用字典查找来获取消息 ID。此版本将返回 1，是 win32service.pyd 中的基本消息 ID.

16.8.12 SMTPHandler

SMTPHandler 类位于 *logging.handlers* 模块，它支持将日志记录消息通过 SMTP 发送到一个电子邮件地址。

class logging.handlers.**SMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=1.0*)
返回一个 *SMTPHandler* 类的新实例。该实例使用电子邮件的发件人、收件人地址和主题行进行初始化。*toaddrs* 应当为字符串列表。要指定一个非标准 SMTP 端口，请使用 (host, port) 元组格式作为 *mailhost* 参数。如果你使用一个字符串，则会使用标准 SMTP 端口。如果你的 SMTP 服务器要求验证，你可以指定一个 (username, password) 元组作为 *credentials* 参数。

要指定使用安全协议 (TLS)，请传入一个元组作为 *secure* 参数。这将仅在提供了验证凭据时才能被使用。元组应当或是一个空元组，或是一个包含密钥文件名的单值元组，或是一个包含密钥文件和证书文件的 2 值元组。(此元组会被传给 *smtplib.SMTP.starttls()* 方法。)

可以使用 *timeout* 参数为与 SMTP 服务器的通信指定超时限制。

3.3 新版功能: 增加了 *timeout* 参数。

emit(*record*)
对记录执行格式化并将其发送到指定的地址。

getSubject(*record*)
如果你想要指定一个基于记录的主题行，请重载此方法。

16.8.13 MemoryHandler

MemoryHandler 类位于*logging.handlers* 模块，它支持在内存中缓冲日志记录，并定期将其刷新到 *target* 处理程序中。刷新会在缓冲区满的时候，或是在遇到特定或更高严重程度事件的时候发生。

MemoryHandler 是更通用的*BufferingHandler* 的子类，后者属于抽象类。它会在内存中缓冲日志记录。当每条记录被添加到缓冲区时，会通过调用 `shouldFlush()` 来检查缓冲区是否应当刷新。如果应当刷新，则要使用 `flush()` 来执行刷新。

class logging.handlers.**BufferingHandler**(*capacity*)
Initializes the handler with a buffer of the specified capacity.

emit(*record*)
Appends the record to the buffer. If *shouldFlush()* returns true, calls *flush()* to process the buffer.

flush()
你可以重载此方法来实现自定义的刷新行为。此版本只是将缓冲区清空。

shouldFlush(*record*)
Returns true if the buffer is up to capacity. This method can be overridden to implement custom flushing strategies.

class logging.handlers.**MemoryHandler**(*capacity*, *flushLevel=ERROR*, *target=None*, *flushOnClose=True*)
Returns a new instance of the *MemoryHandler* class. The instance is initialized with a buffer size of *capacity*. If *flushLevel* is not specified, `ERROR` is used. If no *target* is specified, the target will need to be set using *setTarget()* before this handler does anything useful. If *flushOnClose* is specified as `False`, then the buffer is *not* flushed when the handler is closed. If not specified or specified as `True`, the previous behaviour of flushing the buffer will occur when the handler is closed.

在 3.6 版更改: 增加了 *flushOnClose* 形参。

close()
调用*flush()*，设置目标为 `None` 并清空缓冲区。

flush()
对于*MemoryHandler*，刷新是指将缓冲的记录发送到目标，如果存在目标的话。当此行为发生时缓冲区也将被清空。如果你想要不同的行为请重载此方法。

setTarget(*target*)
设置此处理程序的目标处理程序。

shouldFlush(*record*)
检测缓冲区是否已满或是有记录为 *flushLevel* 或更高级别。

16.8.14 HTTPHandler

HTTPHandler 类位于*logging.handlers* 模块，它支持使用 `GET` 或 `POST` 语义将日志记录消息发送到 Web 服务器。

class logging.handlers.**HTTPHandler**(*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*, *context=None*)
返回一个*HTTPHandler* 类的新实例。*host* 可以为 `host:port` 的形式，如果你需要使用指定端口号的话。如果没有指定 *method*，则会使用 `GET`。如果 *secure* 为真值，则将使用 HTTPS 连接。*context* 形参可以设为一个*ssl.SSLContext* 实例以配置用于 HTTPS 连接的 SSL 设置。如果指定了 *credentials*，它应当为包含 userid 和 password 的二元组，该元组将被放入使用 Basic 验证的 HTTP ‘Authorization’ 标头中。如果你指定了 credentials，你还应当指定 secure=True 这样你的 userid 和 password 就不会以纯文本形式在线路上传输。

在 3.5 版更改: 增加了 *context* 形参。

mapLogRecord(*record*)
基于 record 提供一个字典，它将被执行 URL 编码并发送至 Web 服务器。默认实现仅返回 record.__dict__。在只需将*LogRecord* 的某个子集发送至 Web 服务器，或者需要对发送至服务器的内容进行更多定制时可以重载此方法。

emit(*record*)
将记录以经 URL 编码的形式发送至 Web 服务器。会使用*mapLogRecord()* 方法来将要发送的记录转换为字典。

注解： 由于记录发送至 Web 服务器所需的预处理与通用的格式化操作不同，使用*setFormatter()* 来指定一个*Formatter* 用于*HTTPHandler* 是没有效果的。此处理程序不会调用*format()*，而是调用*mapLogRecord()* 然后再调用*urllib.parse.urlencode()* 来以适合发送至 Web 服务器的形式对字典进行编码。

16.8.15 QueueHandler

3.2 新版功能.

QueueHandler 类位于*logging.handlers* 模块，它支持将日志记录消息发送到一个队列，例如在*queue* 或*multiprocessing* 模块中实现的队列。

配合*QueueListener* 类使用，*QueueHandler* 可用来使处理程序在与执行日志记录的线程不同的线程上完成工作。这对 Web 应用程序以及其他服务于客户端的线程需要尽可能快地响应的服务应用程序来说很重要，任何潜在的慢速操作（例如通过*SMTPHandler* 发送邮件）都要在单独的线程上完成。

class logging.handlers.**QueueHandler**(*queue*)
Returns a new instance of the *QueueHandler* class. The instance is initialized with the queue to send messages to. The queue can be any queue-like object; it's used as-is by the *enqueue()* method, which needs to know how to send messages to it.

emit(*record*)
Enqueues the result of preparing the LogRecord.

prepare(*record*)
准备用于队列的记录。此方法返回的对象会被排入队列。

The base implementation formats the record to merge the message and arguments, and removes unpickleable items from the record in-place.

如果你想要将记录转换为 dict 或 JSON 字符串，或者发送记录被修改后的副本而让初始记录保持原样，则你可能会想要重载此方法。

enqueue(*record*)
使用 put_nowait() 将记录排入队列；如果你想要使用阻塞行为，或超时设置，或自定义的队列实现，则你可能会想要重载此方法。

16.8.16 QueueListener

3.2 新版功能.

QueueListener 类位于*logging.handlers* 模块，它支持从一个队列接收日志记录消息，例如在*queue* 或*multiprocessing* 模块中实现的队列。消息是在内部线程中从队列接收并在同一线程上传递到一个或多个处理程序进行处理的。尽管*QueueListener* 本身并不是一个处理程序，但由于它要与*QueueHandler* 配合工作，因此也在此处介绍。

配合*QueueHandler* 类使用，*QueueListener* 可用来使处理程序在与执行日志记录的线程不同的线程上完成工作。这对 Web 应用程序以及其他服务与客户端的线程需要尽可能快地响应的服务应用程序来说很重要，任何潜在的慢速动作（例如通过*SMTPHandler* 发送邮件）都要在单独的线程上完成。

class logging.handlers.**QueueListener**(*queue*, **handlers*, *respect_handler_level=False*)

Returns a new instance of the *QueueListener* class. The instance is initialized with the queue to send messages to and a list of handlers which will handle entries placed on the queue. The queue can be any queue-like object; it's passed as-is to the *dequeue()* method, which needs to know how to get messages from it. If `respect_handler_level` is `True`, a handler's level is respected (compared with the level for the message) when deciding whether to pass messages to that handler; otherwise, the behaviour is as in previous Python versions - to always pass each message to each handler.

在 3.5 版更改: The `respect_handler_levels` argument was added.

dequeue(*block*)

从队列移出一条记录并将其返回，可以选择阻塞。

基本实现使用 `get()`。如果你想要使用超时设置或自定义的队列实现，则你可能会想要重载此方法。

prepare(*record*)

准备一条要处理的记录。

该实现只是返回传入的记录。如果你想要对记录执行任何自定义的 marshal 操作或在将其传给处理程序之前进行调整，则你可能会想要重载此方法。

handle(*record*)

处理一条记录。

此方法简单地循环遍历处理程序，向它们提供要处理的记录。实际传给处理程序的对象就是从*prepare()* 返回的对象。

start()

启动监听器。

此方法启动一个后台线程来监视 LogRecords 队列以进行处理。

stop()

停止监听器。

此方法要求线程终止，然后等待它完成终止操作。请注意在你的应用程序退出之前如果你没有调用此方法，则可能会有一些记录在留在队列中，它们将不会被处理。

enqueue_sentinel()

将一个标记写入队列以通知监听器退出。此实现会使用 `put_nowait()`。如果你想要使得超时设置或自定义的队列实现，则你可能会想要重载此方法。

3.3 新版功能.

参见:

模块*logging* 日志记录模块的 API 参考。

模块*logging.config* 日志记录模块的配置 API 。

16.9 getpass —便携式密码输入工具

源代码: Lib/getpass.py

---

getpass 模块提供了两个函数:

getpass.**getpass**(*prompt='Password: '*, *stream=None*)
提示用户输入一个密码且不会回显。用户会看到字符串 *prompt* 作为提示，其默认值为 `'Password: '`。在 Unix 上，如有必要提示会使用替换错误句柄写入到文件类对象 *stream*。*stream* 默认指向控制终端 (`/dev/tty`)，如果不可用则指向 `sys.stderr` (此参数在 Windows 上会被忽略)。

如果回显自由输入不可用则 getpass() 将回退为打印一条警告消息到 *stream* 并且从 `sys.stdin` 读取同时发出*GetPassWarning*。

注解: 如果你从 IDLE 内部调用 getpass，输入可能是在你启动 IDLE 的终端中而非在 IDEL 窗口本身中完成。

---

exception getpass.**GetPassWarning**
一个当密码输入可能被回显时发出的*UserWarning* 子类。

getpass.**getuser**()
返回用户的“登录名称”。

此函数会按顺序检查环境变量 `LOGNAME`, `USER`, `LNAME` 和 `USERNAME`，并返回其中第一个被设置为非空字符串的值。如果均未设置，则在支持*pwd* 模块的系统上将返回来自密码数据库的登录名，否则将引发一个异常。

通常情况下，此函数应优先于*os.getlogin()* 使用。

16.10 curses —终端字符单元显示的处理

---

curses 模块提供了 curses 库的接口，这是可移植高级终端处理的事实标准。

虽然 curses 在 Unix 环境中使用最为广泛，但也有适用于 Windows，DOS 以及其他可能的系统的版本。此扩展模块旨在匹配 ncurses 的 API，这是一个部署在 Linux 和 Unix 的 BSD 变体上的开源 curses 库。

注解: 每当文档提到 **字符**时，它可以被指定为一个整数，一个单字符 Unicode 字符串或者一个单字节的字节字符串。

每当此文档提到 **字符串**时，它可以被指定为一个 Unicode 字符串或者一个字节字符串。

注解: 从 5.4 版本开始，ncurses 库使用 `nl_langinfo` 函数来决定如何解释非 ASCII 数据。这意味着你需要在程序中调用*locale.setlocale()* 函数，并使用一种系统中可用的编码方法来编码 Unicode 字符串。这个例子使用了系统默认的编码:

```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

然后使用 *code* 作为*str.encode()* 调用的编码。

参见:

模块*curses.ascii* 在 ASCII 字符上工作的工具，无论你的区域设置是什么。

模块*curses.panel* 为 curses 窗口添加深度的面板栈扩展。

模块*curses.textpad* 用于使 curses 支持 **Emacs** 式绑定的可编辑文本部件。

curses-howto 关于配合 Python 使用 curses 的教学材料，由 Andrew Kuchling 和 Eric Raymond 撰写。

Python 源码发布包的 Tools/demo/ 目录包含了一些使用此模块所提供的 curses 绑定的示例程序。

16.10.1 函数

curses 模块定义了以下异常：

exception curses.**error**
当 curses 库中函数返回一个错误时引发的异常。

注解: 只要一个函数或方法的 *x* 或 *y* 参数是可选项，它们会默认为当前光标位置。而当 *attr* 是可选项时，它会默认为 A_NORMAL。

curses 模块定义了以下函数：

curses.**baudrate**()
以每秒比特数为单位返回终端输出速度。在软件终端模拟器上它将具有一个固定的最高值。此函数出于历史原因被包括；在以前，它被用于写输出循环以提供时间延迟，并偶尔根据线路速度来改变接口。

curses.**beep**()
发出短促的提醒声音。

curses.**can_change_color**()
根据程序员能否改变终端显示的颜色返回 True 或 False。

curses.**cbreak**()
进入 cbreak 模式。在 cbreak 模式（有时也称为“稀有”模式）通常的 tty 行缓冲会被关闭并且字符可以被一个一个地读取。但是，与原始模式不同，特殊字符（中断、退出、挂起和流程控制）会在 tty 驱动和调用程序上保留其效果。首先调用*raw()* 然后调用*cbreak()* 会将终端置于 cbreak 模式。

curses.**color_content**(*color_number*)
返回颜色值 *color_number* 中红、绿和蓝（RGB）分量的强度，此强度值必须介于 0 和 COLORS 之间。返回一个 3 元组，其中包含给定颜色的 R,G,B 值，它们必须介于 0 (无分量) 和 1000 (最大分量) 之间。

curses.**color_pair**(*color_number*)
返回用于显示指定颜色的文本的属性值。该属性值可与 A_STANDOUT, A_REVERSE 以及其他 A_* 属性组合使用。*pair_number()* 是此函数的对应操作。

curses.**curs_set**(*visibility*)
设置光标状态。*visibility* 可设为 0, 1 或 2 表示不可见、正常与高度可见。如果终端支持所请求的可见性，则返回之前的光标状态；否则会引发异常。在许多终端上，“正常可见”模式为下划线光标而“高度可见”模式为方块形光标。

curses.**def_prog_mode**()
将当前终端模式保存为“program”模式，即正在运行的程序使用 curses 时的模式。（与其相对的是“shell”模式，即程序不使用 curses。）对*reset_prog_mode()* 的后续调用将恢复此模式。

curses.**def_shell_mode**()
将当前终端模式保存为 “shell” 模式，即正在运行的程序不使用 curses 的模式。（与其相对的是 “program” 模式，即程序使用功能。）对*reset_shell_mode()* 的后续调用将恢复此模式。

curses.**delay_output**(*ms*)
在输出中插入 *ms* 毫秒的暂停。

curses.**doupdate**()
更新物理屏幕。curses 库会保留两个数据结构，一个代表当前物理屏幕的内容以及一个虚拟屏幕代表需要的后续状态。*doupdate()* 整体更新物理屏幕以匹配虚拟屏幕。

虚拟屏幕可以通过在写入操作例如在一个窗口上执行*addstr()* 之后调用*noutrefresh()* 来刷新。普通的*refresh()* 调用只是简单的 `noutrefresh()` 加 `doupdate()`；如果你需要更新多个窗口，你可以通过在所有窗口上发出 `noutrefresh()` 调用再加单次 `doupdate()` 来提升性能并可减少屏幕闪烁。

curses.**echo**()
进入 echo 模式。在 echo 模式下，输入的每个字符都会在输入后回显到屏幕上。

curses.**endwin**()
撤销库的初始化，使终端返回正常状态。

curses.**erasechar**()
将用户的当前擦除字符以单字节字节串对象的形式返回。在 Unix 操作系统下这是 curses 程序用来控制 tty 的属性，而不是由 curses 库本身来设置的。

curses.**filter**()
如果要使用*filter()* 例程，它必须在调用*initscr()* 之前被调用。其效果是在这些调用期间，`LINES` 会被设为 `1`；`clear`, `cup`, `cud`, `cud1`, `cuu1`, `cuu`, `vpa` 等功能会被禁用；而 `home` 字符串会被设为 `cr` 的值。其影响是光标会被限制在当前行内，屏幕刷新也是如此。这可被用于启用单字符模式的行编辑而不触及屏幕的其余部分。

curses.**flash**()
闪烁屏幕。也就是将其改为反显并在很短的时间内将其改回原状。有些人更喜欢这样的‘视觉响铃’而非*beep()* 所产生的听觉提醒信号。

curses.**flushinp**()
刷新所有输入缓冲区。这会丢弃任何已被用户输入但尚未被程序处理的预输入内容。

curses.**getmouse**()
After *getch()* returns `KEY_MOUSE` to signal a mouse event, this method should be call to retrieve the queued mouse event, represented as a 5-tuple `(id, x, y, z, bstate)`. *id* is an ID value used to distinguish multiple devices, and *x*, *y*, *z* are the event’s coordinates. (*z* is currently unused.) *bstate* is an integer value whose bits will be set to indicate the type of event, and will be the bitwise OR of one or more of the following constants, where *n* is the button number from 1 to 4: `BUTTONn_PRESSED`, `BUTTONn_RELEASED`, `BUTTONn_CLICKED`, `BUTTONn_DOUBLE_CLICKED`, `BUTTONn_TRIPLE_CLICKED`, `BUTTON_SHIFT`, `BUTTON_CTRL`, `BUTTON_ALT`.

curses.**getsyx**()
将当前虚拟屏幕光标的坐标作为元组 `(y, x)` 返回。如果*leaveok* 当前为 `True`，则返回 `(-1, -1)`。

curses.**getwin**(*file*)
读取由之前的 `putwin()` 调用存放在文件中的窗口相关数据。该例程随后将使用该数据创建并初始化一个新窗口，并返回该新窗口对象。

curses.**has_colors**()
如果终端能显示彩色则返回 `True`；否则返回 `False`。

curses.**has_ic**()
如果终端具有插入和删除字符的功能则返回 `True`。此函数仅是出于历史原因而被包括的，因为所有现代软件终端模拟器都具有这些功能。

curses.**has_il**()
如果终端具有插入和删除字符功能，或者能够使用滚动区域来模拟这些功能则返回 True。此函数仅是出于历史原因而被包括的，因为所有现代软件终端模拟器都具有这些功能。

curses.**has_key**(*ch*)
接受一个键值 *ch*，并在当前终端类型能识别出具有该值的键时返回 True。

curses.**halfdelay**(*tenths*)
用于半延迟模式，与 cbreak 模式的类似之处是用户所键入的字符会立即对程序可用。但是，在阻塞 *tenths* 个十分之一秒之后，如果还未输入任何内容则将引发异常。*tenths* 值必须为 1 和 255 之间的数字。使用*nocbreak()* 可退出半延迟模式。

curses.**init_color**(*color_number*, *r*, *g*, *b*)
更改某个颜色的定义，接受要更改的颜色编号以及三个 RGB 值（表示红绿蓝三分量的强度）。*color_number* 值必须为 0 和 COLORS 之间的数字。*r*, *g*, *b* 值分别必须为 0 和 1000 之间的数字。当使用*init_color()* 时，出现在屏幕上的对应颜色会立即按照新定义来更改。此函数在大多数终端上都是无操作的；它仅会在*can_change_color()* 返回 True 时生效。

curses.**init_pair**(*pair_number*, *fg*, *bg*)
更改某个颜色对的定义。它接受三个参数：要更改的颜色对编号，前景色编号和背景色编号。*pair_number* 值必须为 1 和 COLOR_PAIRS - 1 之间的数字（并且 0 号颜色对固定为黑底白字而无法更改）。*fg* 和 *bg* 参数值必须为 0 和 COLORS 之间的数字。如果颜色对之前已被初始化，则屏幕会被刷新使得出现在屏幕上的该颜色对会立即按照新定义来更改。

curses.**initscr**()
初始化库。返回代表整个屏幕的*窗口* 对象。

注解： 如果打开终端时发生错误，则下层的 curses 库可能会导致解释器退出。

---

curses.**is_term_resized**(*nlines*, *ncols*)
如果*resize_term()* 会修改窗口结构则返回 True，否则返回 False。

curses.**isendwin**()
如果*endwin()* 已经被调用（即 curses 库已经被撤销初始化则返回 True。

curses.**keyname**(*k*)
将编号为 *k* 的键名称作为字节串对象返回。生成可打印 ASCII 字符的键名称就是键所对应的字符。Ctrl-键组合的键名称则是一个两字节的字节串对象，它由插入符 (b'^') 加对应的可打印 ASCII 字符组成。Alt-键组合 (128–255) 的键名称则是由前缀 b'M-' 加对应的可打印 ASCII 字符组成的字节串对象。

curses.**killchar**()
将用户的当前行删除字符以单字节字节串对象的形式返回。在 Unix 操作系统下这是 curses 程序用来控制 tty 的属性，而不是由 curses 库本身来设置的。

curses.**longname**()
返回一个字节串对象，其中包含描述当前终端的 terminfo 长名称字段。详细描述的最大长度为 128 个字符。它仅在调用*initscr()* 之后才会被定义。

curses.**meta**(*flag*)
如果 *flag* 为 True，则允许输入 8 比特位的字符。如果 *flag* 为 False，则只允许 7 比特位的字符。

curses.**mouseinterval**(*interval*)
以毫秒为单位设置能够被识别为点击的按下和释放事件之间可以间隔的最长时间，并返回之前的间隔值。默认值为 200 毫秒，即五分之一秒。

curses.**mousemask**(*mousemask*)
设置要报告的鼠标事件，并返回一个元组 (availmask, oldmask)。*availmask* 表明指定的鼠标事件中哪些可以被报告；当完全失败时将返回 0。*oldmask* 是给定窗口的鼠标事件之前的掩码值。如果从未调用此函数，则不会报告任何鼠标事件。

curses.**napms**(*ms*)
休眠 *ms* 毫秒。

curses.**newpad**(*nlines*, *ncols*)
创建并返回一个指向具有给定行数和列数新的填充数据结构的指针。将填充作为窗口对象返回。

面板类似于窗口，区别在于它不受屏幕大小的限制，并且不必与屏幕的特定部分相关联。面板可以在需要使用大窗口时使用，并且每次只需将窗口的一部分放在屏幕上。面板不会发生自动刷新（例如由于滚动或输入回显）。面板的*refresh()* 和*noutrefresh()* 方法需要 6 个参数来指定面板要显示的部分以及要用于显示的屏幕位置。这些参数是 *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*；*p* 参数表示要显示的面板区域的左上角而 *s* 参数定义了要显示的面板区域在屏幕上的剪切框。

curses.**newwin**(*nlines*, *ncols*)
curses.**newwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
返回一个新的窗口，其左上角位于 `(begin_y, begin_x)`，并且其高度/宽度为 *nlines*/*ncols*。

默认情况下，窗口将从指定位置扩展到屏幕的右下角。

curses.**nl**()
进入换行模式。此模式会在输入时将回车转换为换行符，并在输出时将换行符转换为回车加换行。换行模式会在初始时启用。

curses.**nocbreak**()
退出 cbreak 模式。返回具有行缓冲的正常“cooked”模式。

curses.**noecho**()
退出 echo 模式。关闭输入字符的回显。

curses.**nonl**()
退出 newline 模式。停止在输入时将回车转换为换行，并停止在输出时从换行到换行/回车的底层转换（但这不会改变 `addch('\n')` 的行为，此行为总是在虚拟屏幕上执行相当于回车加换行的操作）。当停止转换时，curses 有时能使纵向移动加快一些；并且，它将能够在输入时检测回车键。

curses.**noqiflush**()
当使用 `noqiflush()` 例程时，与 `INTR`, `QUIT` 和 `SUSP` 字符相关联的输入和输出队列的正常刷新将不会被执行。如果你希望在处理程序退出后还能继续输出，就像没有发生过中断一样，你可能会想要在信号处理程序中调用 `noqiflush()`。

curses.**noraw**()
退出 raw 模式。返回具有行缓冲的正常“cooked”模式。

curses.**pair_content**(*pair_number*)
返回包含对应于所请求颜色对的颜色的元组 `(fg, bg)`。*pair_number* 的值必须在 `1` 和 `COLOR_PAIRS - 1` 之间。

curses.**pair_number**(*attr*)
返回通过属性值 *attr* 所设置的颜色对的编号。*color_pair()* 是此函数的对应操作。

curses.**putp**(*str*)
等价于 `tputs(str, 1, putchar)`；为当前终端发出指定 terminfo 功能的值。请注意*putp()* 的输出总是前往标准输出。

curses.**qiflush**([*flag*])
如果 *flag* 为 `False`，则效果与调用*noqiflush()* 相同。如果 *flag* 为 `True` 或未提供参数，则在读取这些控制字符时队列将被刷新。

curses.**raw**()
进入 raw 模式。在 raw 模式下，正常的行缓冲和对中断、退出、挂起和流程控制键的处理会被关闭；字符会被逐个地提交给 curses 输入函数。

curses.**reset_prog_mode**()
将终端恢复到“program”模式，如之前由*def_prog_mode()* 所保存的一样。

curses.**reset_shell_mode**()
将终端恢复到“shell”模式，如之前由*def_shell_mode()* 所保存的一样。

curses.**resetty**()
将终端模式恢复到最后一次调用*savetty()* 时的状态。

curses.**resize_term**(*nlines*, *ncols*)
由*resizeterm()* 用来执行大部分工作的后端函数；当调整窗口大小时，*resize_term()* 会以空白填充扩展区域。调用方应用程序应当以适当的数据填充这些区域。resize_term() 函数会尝试调整所有窗口的大小。但是，由于面板的调用约定，在不与应用程序进行额外交互的情况下是无法调整其大小的。

curses.**resizeterm**(*nlines*, *ncols*)
将标准窗口和当前窗口的大小调整为指定的尺寸，并调整由 curses 库所使用的记录窗口尺寸的其他记录数据（特别是 SIGWINCH 处理程序）。

curses.**savetty**()
将终端模式的当前状态保存在缓冲区中，可供*resetty()* 使用。

curses.**setsyx**(*y*, *x*)
将虚拟屏幕光标设置到 *y*, *x*。如果 *y* 和 *x* 均为 -1，则*leaveok* 将设为 True。

curses.**setupterm**(*term=None*, *fd=-1*)
初始化终端。*term* 为给出终端名称的字符串或为 None；如果省略或为 None，则将使用 TERM 环境变量的值。*fd* 是任何初始化序列将被发送到的文件描述符；如未指定或为 -1，则将使用 sys.stdout 的文件描述符。

curses.**start_color**()
如果程序员想要使用颜色，则必须在任何其他颜色操作例程被调用之前调用它。在*initscr()* 之后立即调用此例程是一个很好的做法。

start_color() 会初始化八种基本颜色（黑、红、绿、黄、蓝、品、青和白）以及*curses* 模块中的两个全局变量 COLORS 和 COLOR_PAIRS，其中包含终端可支持的颜色和颜色对的最大数量。它还会将终端中的颜色恢复为终端刚启动时的值。

curses.**termattrs**()
返回终端所支持的所有视频属性逻辑 OR 的值。此信息适用于当 curses 程序需要对屏幕外观进行完全控制的情况。

curses.**termname**()
将环境变量 TERM 的值截短至 14 个字节，作为字节串对象返回。

curses.**tigetflag**(*capname*)
将与 terminfo 功能名称 *capname* 相对应的布尔功能值以整数形式返回。如果 *capname* 不是一个布尔功能则返回 -1，如果其被取消或不存在于终端描述中则返回 0。

curses.**tigetnum**(*capname*)
将与 terminfo 功能名称 *capname* 相对应的数字功能值以整数形式返回。如果 *capname* 不是一个数字功能则返回 -2，如果其被取消或不存在于终端描述中则返回 -1。

curses.**tigetstr**(*capname*)
将与 terminfo 功能名称 *capname* 相对应的字符串功能值以字节串对象形式返回。如果 *capname* 不是一个 terminfo “字符串功能”或者如果其被取消或不存在于终端描述中则返回 None。

curses.**tparm**(*str*[, ...])
使用提供的形参初始化字节串对象 *str*，其中 *str* 应当是从 terminfo 数据库获取的参数化字符串。例如 tparm(tigetstr("cup"), 5, 3) 的结果可能为 b'\033[6;4H'，实际结果将取决于终端类型。

curses.**typeahead**(*fd*)
指定将被用于预输入检查的文件描述符 *fd*。如果 *fd* 为 -1，则不执行预输入检查。

curses 库会在更新屏幕时通过定期查找预输入来执行“断行优化”。如果找到了输入，并且输入是来自于 tty，则会将当前更新推迟至 refresh 或 doupdate 再次被调用的时候，以便允许更快地响应预先输入的命令。此函数允许为预输入检查指定其他的文件描述符。

curses.**unctrl**(*ch*)
返回一个字节串对象作为字符 *ch* 的可打印表示形式。控制字符会表示为一个变换符加相应的字符，例如 `b'^C'`。可打印字符则会保持原样。

curses.**ungetch**(*ch*)
推送 *ch* 以便让下一个*getch()* 返回该字符。

注解： 在 getch() 被调用之前只能推送一个 *ch*。

curses.**update_lines_cols**()
更新 LINES 和 COLS。适用于检测屏幕大小的手动调整。

3.5 新版功能.

curses.**unget_wch**(*ch*)
推送 *ch* 以便让下一个*get_wch()* 返回该字符。

注解： 在 get_wch() 被调用之前只能推送一个 *ch*。

3.3 新版功能.

curses.**ungetmouse**(*id*, *x*, *y*, *z*, *bstate*)
将 KEY_MOUSE 事件推送到输入队列，将其与给定的状态数据进行关联。

curses.**use_env**(*flag*)
如果使用此函数，则应当在调用*initscr()* 或 newterm 之前调用它。当 *flag* 为 False 时，将会使用在 terminfo 数据库中指定的行和列的值，即使设置了环境变量 LINES 和 COLUMNS (默认使用)，或者如果 curses 是在窗口中运行（在此情况下如果未设置 LINES 和 COLUMNS 则默认行为将是使用窗口大小）。

curses.**use_default_colors**()
允许在支持此特性的终端上使用默认的颜色值。使用此函数可在你的应用程序中支持透明效果。默认颜色会被赋给颜色编号 -1。举例来说，在调用此函数后，`init_pair(x, curses.COLOR_RED, -1)` 会将颜色对 *x* 初始化为红色前景和默认颜色背景。

curses.**wrapper**(*func*, ...)
初始化 curses 并调用另一个可调用对象 *func*，该对象应当为你的使用 curses 的应用程序的其余部分。如果应用程序引发了异常，此函数将在重新引发异常并生成回溯信息之前将终端恢复到正常状态。随后可调用对象 *func* 会被传入主窗口‘stdscr’作为其第一个参数，再带上其他所有传给 wrapper() 的参数。在调用 *func* 之前，wrapper() 会启用 cbreak 模式，关闭回显，启用终端键盘，并在终端具有颜色支持的情况下初始化颜色。在退出时（无论是正常退出还是异常退出）它会恢复 cooked 模式，打开回显，并禁用终端键盘。

16.10.2 Window 对象

Window 对象会由上面的*initscr()* 和*newwin()* 返回，它具有以下方法和属性:

window.**addch**(*ch*[, *attr*])
window.**addch**(*y*, *x*, *ch*[, *attr*])
Paint character *ch* at `(y, x)` with attributes *attr*, overwriting any character previously painter at that location. By default, the character position and attributes are the current settings for the window object.

注解: 在窗口、子窗口或面板之外写入会引发*curses.error*。尝试在窗口、子窗口或面板的右下角写入将在字符被打印之后导致异常被引发。

---

window.**addnstr**(*str*, *n*[, *attr*])
window.**addnstr**(*y*, *x*, *str*, *n*[, *attr*])
将带有属性 *attr* 的字符串 *str* 中的至多 *n* 个字符绘制到 `(y, x)`，覆盖之前在屏幕上的任何内容。

window.**addstr**(*str*[, *attr*])
window.**addstr**(*y*, *x*, *str*[, *attr*])
将带有属性 *attr* 的字符串 *str* 绘制到 `(y, x)`，覆盖之前在屏幕上的任何内容。

注解: 在窗口、子窗口或面板之外写入会引发*curses.error*。尝试在窗口、子窗口或面板的右下角写入将在字符串被打印之后导致异常被引发。

---

window.**attroff**(*attr*)
从应用于写入到当前窗口的“background”集中移除属性 *attr*。

window.**attron**(*attr*)
向应用于写入到当前窗口的“background”集中添加属性 *attr*。

window.**attrset**(*attr*)
将“background”属性集设为 *attr*。该集合初始时为 `0` (无属性)。

window.**bkgd**(*ch*[, *attr*])
将窗口 background 特征属性设为带有属性 *attr* 的字符 *ch*。随后此修改将应用于放置到该窗口中的每个字符。

- 窗口中每个字符的属性会被修改为新的 background 属性。
- 不论之前的 background 字符出现在哪里，它都会被修改为新的 background 字符。

window.**bkgdset**(*ch*[, *attr*])
设置窗口的背景。窗口的背景由字符和属性的任意组合构成。背景的属性部分会与写入窗口的所有非空白字符合并（即 OR 运算）。背景和字符和属性部分均会与空白字符合并。背景将成为字符的特征属性并在任何滚动与插入/删除行/字符操作中与字符一起移动。

window.**border**([*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]])
在窗口边缘绘制边框。每个参数指定用于边界特定部分的字符; 请参阅下表了解更多详情。

注解: 任何形参的值为 `0` 都将导致该形参使用默认字符。关键字形参 不可被使用。默认字符在下表中列出:

参数	描述	默认值
ls	左侧	ACS_VLINE
rs	右侧	ACS_VLINE
ts	顶部	ACS_HLINE
bs	底部	ACS_HLINE
tl	左上角	ACS_ULCORNER
tr	右上角	ACS_URCORNER
bl	左下角	ACS_LLCORNER
br	右下角	ACS_LRCORNER

window.**box**([*vertch*, *horch*])
类似于*border()*，但 *ls* 和 *rs* 均为 *vertch* 而 *ts* 和 *bs* 均为 *horch*。此函数总是会使用默认的转角字符。

window.**chgat**(*attr*)
window.**chgat**(*num*, *attr*)
window.**chgat**(*y*, *x*, *attr*)
window.**chgat**(*y*, *x*, *num*, *attr*)
在当前光标位置或是在所提供的位置（y, x）设置 *num* 个字符的属性。如果 *num* 未给出或为 -1，则将属性设置到所有字符上直至行尾。如果提供了位置（y, x）则此函数会将光标移至该位置。修改过的行将使用*touchline()* 方法处理以便下次窗口刷新时内容会重新显示。

window.**clear**()
类似于*erase()*，但还会导致在下次调用*refresh()* 时整个窗口被重新绘制。

window.**clearok**(*flag*)
如果 *flag* 为 True，则在下次调用*refresh()* 时将完全清除窗口。

window.**clrtobot**()
从光标位置开始擦除直至窗口末端：光标以下的所有行都会被删除，然后会执行*clrtoeol()* 的等效操作。

window.**clrtoeol**()
从光标位置开始擦除直至行尾。

window.**cursyncup**()
更新窗口所有上级窗口的当前光标位置以反映窗口的当前光标位置。

window.**delch**([*y*, *x*])
删除位于（y, x）的任何字符。

window.**deleteln**()
删除在光标之下的行。所有后续的行都会上移一行。

window.**derwin**(*begin_y*, *begin_x*)
window.**derwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
“derive window” 的缩写，*derwin()* 与调用*subwin()* 等效，不同之处在于 *begin_y* 和 *begin_x* 是想对于窗口的初始位置，而不是相对于整个屏幕。返回代表所派生窗口的窗口对象。

window.**echochar**(*ch*[, *attr*])
使用属性 *attr* 添加字符 *ch*，并立即在窗口上调用*refresh()*。

window.**enclose**(*y*, *x*)
检测给定的相对屏幕的字符-单元格坐标是否被给定的窗口所包围，返回 True 或 False。它适用于确定是哪个屏幕窗口子集包围着某个鼠标事件的位置。

window.**encoding**
用于编码方法参数（Unicode 字符串和字符）的编码格式。encoding 属性是在创建子窗口时从父窗口继承的，例如通过*window.subwin()*。默认情况下，会使用当前区域的编码格式 (参见*locale.getpreferredencoding()*)。

3.3 新版功能.

window.**erase**()
清空窗口。

window.**getbegyx**()
返回左上角坐标的元组 (y, x)。

window.**getbkgd**()
返回给定窗口的当前背景字符/属性对。

window.**getch**([*y*, *x*])
获取一个字符。请注意所返回的整数 不一定要在 ASCII 范围以内：功能键、小键盘键等等是由大于 255 的数字表示的。在无延迟模式下，如果没有输入则返回 -1，在其他情况下都会等待直至有键被按下。

window.**get_wch**([*y*, *x*])
获取一个宽字符。对于大多数键都是返回一个字符，对于功能键、小键盘键和其他特殊键则是返回一个整数。在无延迟模式下，如果没有输入则返回一个异常。

3.3 新版功能.

window.**getkey**([*y*, *x*])
获取一个字符，返回一个字符串而不是像*getch()* 那样返回一个整数。功能键、小键盘键和其他特殊键则是返回一个包含键名的多字节字符串。在无延迟模式下，如果没有输入则引发一个异常。

window.**getmaxyx**()
返回窗口高度和宽度的元组 (y, x)。

window.**getparyx**()
将此窗口相对于父窗口的起始坐标作为元组 (y, x) 返回。如果此窗口没有父窗口则返回 (-1, -1)。

window.**getstr**()
window.**getstr**(*n*)
window.**getstr**(*y*, *x*)
window.**getstr**(*y*, *x*, *n*)
从用户读取一个字节串对象，附带基本的行编辑功能。

window.**getyx**()
返回当前光标相对于窗口左上角的位置的元组 (y, x)。

window.**hline**(*ch*, *n*)
window.**hline**(*y*, *x*, *ch*, *n*)
显示一条起始于 (y, x) 长度为 *n* 个字符 *ch* 的水平线。

window.**idcok**(*flag*)
如果 *flag* 为 False，curses 将不再考虑使用终端的硬件插入/删除字符功能；如果 *flag* 为 True，则会启用字符插入和删除。当 curses 首次初始化时，默认会启用字符插入/删除。

window.**idlok**(*flag*)
如果 *flag* 为 True，*curses* 将尝试使用硬件行编辑功能。否则，行插入/删除会被禁用。

window.**immedok**(*flag*)
如果 *flag* 为 True，窗口图像中的任何改变都会自动导致窗口被刷新；你不必再自己调用*refresh()*。但是，这可能会由于重复调用 wrefresh 而显著降低性能。此选项默认被禁用。

window.**inch**([*y*, *x*])
返回窗口中给定位置上的字符。下面的 8 个比特位是字符本身，上面的比特位则为属性。

window.**insch**(*ch*[, *attr*])
window.**insch**(*y*, *x*, *ch*[, *attr*])
将带有属性 *attr* 的字符串 *ch* 绘制到 (y, x)，将该行从位置 *x* 开始右移一个字符。

window.**insdelln**(*nlines*)
在指定窗口的当前行上方插入 *nlines* 行。下面的 *nlines* 行将丢失。对于 *nlines* 为负值的情况，则从光标下方的行开始删除 *nlines* 行，并将其余的行向上移动。下面的 *nlines* 行会被清空。当前光标位置将保持不变。

window.**insertln**()
在光标下方插入一个空行。所有后续的行都会下移一行。

window.**insnstr**(*str*, *n*[, *attr*])
window.**insnstr**(*y*, *x*, *str*, *n*[, *attr*])
在光标下方的字符之前插入一个至多为 *n* 个字符的字符串（字符数量将与该行相匹配）。如果 *n* 为零或负数，则插入整个字符串。光标右边的所有字符将被右移，该行右端的字符将丢失。光标位置将保持不变（在移到可能指定的 *y*, *x* 之后）。

window.**insstr**(*str*[, *attr*])
window.**insstr**(*y*, *x*, *str*[, *attr*])
在光标下方的字符之前插入一个字符串（字符数量将与该行相匹配）。光标右边的所有字符将被右移，该行右端的字符将丢失。光标位置将保持不变（在移到可能指定的 *y*, *x* 之后）。

window.**instr**([*n*])
window.**instr**(*y*, *x*[, *n*])
返回从窗口的当前光标位置，或者指定的 *y*, *x* 开始提取的字符所对应的字节串对象。属性会从字符中去除。如果指定了 *n*，*instr()* 将返回长度至多为 *n* 个字符的字符串（不包括末尾的 NUL）。

window.**is_linetouched**(*line*)
如果指定的行自上次调用*refresh()* 后发生了改变则返回 `True`；否则返回 `False`。如果 *line* 对于给定的窗口不可用则会引发*curses.error* 异常。

window.**is_wintouched**()
如果指定的窗口自上次调用*refresh()* 后发生了改变则返回 `True`；否则返回 `False`。

window.**keypad**(*flag*)
如果 *flag* 为 `True`，则某些键（小键盘键、功能键等）生成的转义序列将由*curses* 来解析。如果 *flag* 为 `False`，转义序列将保持在输入流中的原样。

window.**leaveok**(*flag*)
如果 *flag* 为 `True`，则在更新时光标将停留在原地，而不是在“光标位置”。这将可以减少光标的移动。在可能的情况下光标将变为不可见。

如果 *flag* 为 `False`，光标在更新后将总是位于“光标位置”。

window.**move**(*new_y*, *new_x*)
将光标移至 `(new_y, new_x)`。

window.**mvderwin**(*y*, *x*)
让窗口在其父窗口内移动。窗口相对于屏幕的参数不会被更改。此例程用于在屏幕的相同物理位置显示父窗口的不同部分。

window.**mvwin**(*new_y*, *new_x*)
移动窗口以使其左上角位于 `(new_y, new_x)`。

window.**nodelay**(*flag*)
如果 *flag* 为 `True`，则*getch()* 将为非阻塞的。

window.**notimeout**(*flag*)
如果 *flag* 为 `True`，则转义序列将不会发生超时。

如果 *flag* 为 `False`，则在几毫秒之后，转义序列将不会被解析，并将保持在输入流中的原样。

window.**noutrefresh**()
标记为刷新但保持等待。此函数会更新代表预期窗口状态的数据结构，但并不强制更新物理屏幕。要完成后者，请调用*doupdate()*。

window.**overlay**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
将窗口覆盖在 *destwin* 上方。窗口的大小不必相同，只有重叠的区域会被复制。此复制是非破坏性的，这意味着当前背景字符不会覆盖掉 *destwin* 的旧内容。

为了获得对被复制区域的细粒度控制，可以使用*overlay()* 的第二种形式。*sminrow* 和 *smincol* 是源窗口的左上角坐标，而其他变量则在目标窗口中标记出一个矩形。

window.**overwrite**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
将窗口覆盖在 *destwin* 上方。窗口的大小不必相同，此时只有重叠的区域会被复制。此复制是破坏性的，这意味着当前背景字符会覆盖掉 *destwin* 的旧内容。

为了获得对被复制区域的细粒度控制，可以使用*overwrite()* 的第二种形式。*sminrow* 和 *smincol* 是源窗口的左上角坐标，而其他变量则在目标窗口中标记出一个矩形。

window.**putwin**(*file*)
将关联到窗口的所有数据写入到所提供的文件对象。此信息可在以后使用*getwin()* 函数来提取。

window.**redrawln**(*beg*, *num*)
指明从 *beg* 行开始的 *num* 个屏幕行已被破坏并且应当在下次*refresh()* 调用时完全重绘。

window.**redrawwin**()
标记整个窗口，以使其在下次*refresh()* 调用时完全重绘。

window.**refresh**([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])
立即更新显示（将实际屏幕与之前的绘制/删除方法进行同步）。

6 个可选参数仅在窗口为使用*newpad()* 创建的面板时可被指定。需要额外的形参来指定所涉及到的是面板和屏幕的哪一部分。*pminrow* 和 *pmincol* 指定要在面板中显示的矩形的左上角。*sminrow*, *smincol*, *smaxrow* 和 *smaxcol* 指定要在屏幕中显示的矩形的边。要在面板中显示的矩形的右下角是根据屏幕坐标计算出来的，由于矩形的大小必须相同。两个矩形都必须完全包含在其各自的结构之内。负的 *pminrow*, *pmincol*, *sminrow* 或 *smincol* 值会被视为将它们设为零值。

window.**resize**(*nlines*, *ncols*)
为 curses 窗口重新分配存储空间以将其尺寸调整为指定的值。如果任一维度的尺寸大于当前值，则窗口的数据将以具有合并了当前背景渲染（由*bkgdset()* 设置）的空白来填充。

window.**scroll**([*lines=1*])
将屏幕或滚动区域向上滚动 *lines* 行。

window.**scrollok**(*flag*)
控制当一个窗口的光标移出窗口或滚动区域边缘时会发生什么，这可能是在底端行执行换行操作，或者在最后一行输入最后一个字符导致的结果。如果 *flag* 为 `False`，光标会留在底端行。如果 *flag* 为 `True`，窗口会向上滚动一行。请注意为了在终端上获得实际的滚动效果，还需要调用*idlok()*。

window.**setscrreg**(*top*, *bottom*)
设置从 *top* 行至 *bottom* 行的滚动区域。所有滚动操作将在此区域中进行。

window.**standend**()
关闭 standout 属性。在某些终端上此操作会有关闭所有属性的副作用。

window.**standout**()
启用属性 *A_STANDOUT*。

window.**subpad**(*begin_y*, *begin_x*)
window.**subpad**(*nlines*, *ncols*, *begin_y*, *begin_x*)
返回一个子窗口，其左上角位于 `(begin_y, begin_x)`，并且其宽度/高度为 *ncols*/*nlines*。

window.**subwin**(*begin_y*, *begin_x*)
window.**subwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
返回一个子窗口，其左上角位于 `(begin_y, begin_x)`，并且其宽度/高度为 *ncols*/*nlines*。

默认情况下，子窗口将从指定位置扩展到窗口的右下角。

window.**syncdown**()
触碰已在上级窗口上被触碰的每个位置。此例程由*refresh()* 调用，因此几乎从不需要手动调用。

window.**syncok**(*flag*)
如果 *flag* 为 True，则*syncup()* 会在窗口发生改变的任何时候自动被调用。

window.**syncup**()
触碰已在窗口中被改变的此窗口的各个上级窗口中的所有位置。

window.**timeout**(*delay*)
为窗口设置阻塞或非阻塞读取行为。如果 *delay* 为负值，则会使用阻塞读取（这将无限期地等待输入）。如果 *delay* 为零，则会使用非阻塞读取，并且当没有输入在等待时*getch()* 将返回 -1。如果 *delay* 为正值，则*getch()* 将阻塞 *delay* 毫秒，并且当此延时结束时仍无输入将返回 -1。

window.**touchline**(*start*, *count*[, *changed*])
假定从行 *start* 开始的 *count* 行已被更改。如果提供了 *changed*，它将指明是将受影响的行标记为已更改 (*changed*=True) 还是未更改 (*changed*=False)。

window.**touchwin**()
假定整个窗口已被更改，其目的是用于绘制优化。

window.**untouchwin**()
将自上次调用*refresh()* 以来窗口中的所有行标记为未改变。

window.**vline**(*ch*, *n*)
window.**vline**(*y*, *x*, *ch*, *n*)
显示一条起始于 (y, x) 长度为 *n* 个字符 *ch* 的垂直线。

16.10.3 常量

curses 模块定义了以下数据成员:

curses.**ERR**
Some curses routines that return an integer, such as getch(), return *ERR* upon failure.

curses.**OK**
一些返回整数的 curses 例程，例如*napms()*，在成功时将返回*OK*。

curses.**version**
一个代表当前模块版本的字节串对象。也作 __version__。

有些常量可用于指定字符单元属性。实际可用的常量取决于具体的系统。

属性	含义
A_ALTCHARSET	备用字符集模式
A_BLINK	闪烁模式
A_BOLD	粗体模式
A_DIM	暗淡模式
A_INVIS	不可见或空白模式
A_NORMAL	正常属性
A_PROTECT	保护模式
A_REVERSE	反转背景色和前景色
A_STANDOUT	突出模式
A_UNDERLINE	下划线模式
A_HORIZONTAL	水平突出显示
A_LEFT	左高亮
A_LOW	底部高亮
A_RIGHT	右高亮
A_TOP	顶部高亮
A_VERTICAL	垂直突出显示
A_CHARTEXT	用于提取字符的位掩码

有几个常量可用于提取某些方法返回的相应属性。

位掩码	含义
A_ATTRIBUTES	用于提取属性的位掩码
A_CHARTEXT	用于提取字符的位掩码
A_COLOR	用于提取颜色对字段信息的位掩码

键由名称以 KEY_ 开头的整数常量引用。确切的可用键取决于系统。

关键常数	（Windows 注册表的）键
KEY_MIN	最小键值
KEY_BREAK	中断键（不可靠）
KEY_DOWN	向下箭头
KEY_UP	向上箭头
KEY_LEFT	向左箭头
KEY_RIGHT	向右箭头
KEY_HOME	Home 键 (上 + 左箭头)
KEY_BACKSPACE	退格（不可靠）
KEY_F0	功能键。支持至多 64 个功能键。
KEY_Fn	功能键 *n* 的值
KEY_DL	删除行
KEY_IL	插入行
KEY_DC	删除字符
KEY_IC	插入字符或进入插入模式
KEY_EIC	退出插入字符模式
KEY_CLEAR	清空屏幕
KEY_EOS	清空至屏幕底部
KEY_EOL	清空至行尾
KEY_SF	向前滚动 1 行
KEY_SR	向后滚动 1 行 (反转)
KEY_NPAGE	下一页

下页继续

表 1 – 续上页

关键常数	（Windows 注册表的）键
KEY_PPAGE	上一页
KEY_STAB	设置制表符
KEY_CTAB	清除制表制
KEY_CATAB	清除所有制表符
KEY_ENTER	回车或发送 (不可靠)
KEY_SRESET	软件 (部分) 重置 (不可靠)
KEY_RESET	重置或硬重置 (不可靠)
KEY_PRINT	打印
KEY_LL	Home 向下或到底 (左下)
KEY_A1	键盘的左上角
KEY_A3	键盘的右上角
KEY_B2	键盘的中心
KEY_C1	键盘左下方
KEY_C3	键盘右下方
KEY_BTAB	回退制表符
KEY_BEG	Beg (开始)
KEY_CANCEL	取消
KEY_CLOSE	关闭
KEY_COMMAND	Cmd (命令行)
KEY_COPY	复制
KEY_CREATE	创建
KEY_END	End
KEY_EXIT	退出
KEY_FIND	查找
KEY_HELP	帮助
KEY_MARK	标记
KEY_MESSAGE	消息
KEY_MOVE	移动
KEY_NEXT	下一个
KEY_OPEN	打开
KEY_OPTIONS	选项
KEY_PREVIOUS	Prev (上一个)
KEY_REDO	重做
KEY_REFERENCE	Ref (引用)
KEY_REFRESH	刷新
KEY_REPLACE	替换
KEY_RESTART	重启
KEY_RESUME	恢复
KEY_SAVE	保存
KEY_SBEG	Shift 的 Beg (开始)
KEY_SCANCEL	Shift 的 Cancel
KEY_SCOMMAND	Shift 的 Command
KEY_SCOPY	Shift + Copy
KEY_SCREATE	Shift + Create
KEY_SDC	Shift + 删除字符
KEY_SDL	Shift + 删除行
KEY_SELECT	选择
KEY_SEND	Shift + End
KEY_SEOL	Shift + 清空行

下页继续

表 1 – 续上页

关键常数	（Windows 注册表的）键
KEY_SEXIT	Shift + Exit
KEY_SFIND	Shift + 查找
KEY_SHELP	Shift + 帮助
KEY_SHOME	Shift + Home
KEY_SIC	Shift + 输入
KEY_SLEFT	Shift + 向左箭头
KEY_SMESSAGE	Shift + 消息
KEY_SMOVE	Shift + 移动
KEY_SNEXT	Shift + 下一个
KEY_SOPTIONS	Shift + 选项
KEY_SPREVIOUS	Shift + 上一个
KEY_SPRINT	Shift + 打印
KEY_SREDO	Shift + 重做
KEY_SREPLACE	Shift + 替换
KEY_SRIGHT	Shift + 向右箭头
KEY_SRSUME	Shift + 恢复
KEY_SSAVE	Shift + 保存
KEY_SSUSPEND	Shift + 挂起
KEY_SUNDO	Shift + 撤销
KEY_SUSPEND	挂起
KEY_UNDO	撤销操作
KEY_MOUSE	鼠标事件已发生
KEY_RESIZE	终端大小改变事件
KEY_MAX	最大键值

在 VT100 及其软件仿真（例如 X 终端仿真器）上，通常至少有四个功能键（KEY_F1, KEY_F2, KEY_F3, KEY_F4 ）可用，并且箭头键以明显的方式映射到 KEY_UP, KEY_DOWN, KEY_LEFT 和 KEY_RIGHT 。如果您的机器有一个 PC 键盘，可以安全地使用箭头键和十二个功能键（旧的 PC 键盘可能只有十个功能键）；此外，以下键盘映射是标准的：

键帽	常数
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

下表列出了替代字符集中的字符。这些字符继承自 VT100 终端，在 X 终端等软件模拟器上通常均为可用。当没有可用的图形时，curses 会回退为粗糙的可打印 ASCII 近似符号。

注解： 只有在调用 *initscr()* 之后才能使用它们

ACS 代码	含义
ACS_BBSS	右上角的别名
ACS_BLOCK	实心方块
ACS_BOARD	正方形

下页继续

表 2 – 续上页

ACS 代码	含义
ACS_BSBS	水平线的别名
ACS_BSSB	左上角的别名
ACS_BSSS	顶部 T 型的别名
ACS_BTEE	底部 T 型
ACS_BULLET	正方形
ACS_CKBOARD	棋盘（点刻）
ACS_DARROW	向下箭头
ACS_DEGREE	等级符
ACS_DIAMOND	菱形
ACS_GEQUAL	大于或等于
ACS_HLINE	水平线
ACS_LANTERN	灯形符号
ACS_LARROW	向左箭头
ACS_LEQUAL	小于或等于
ACS_LLCORNER	左下角
ACS_LRCORNER	右下角
ACS_LTEE	左侧 T 型
ACS_NEQUAL	不等号
ACS_PI	字母 π
ACS_PLMINUS	正负号
ACS_PLUS	加号
ACS_RARROW	向右箭头
ACS_RTEE	右侧 T 型
ACS_S1	扫描线 1
ACS_S3	扫描线 3
ACS_S7	扫描线 7
ACS_S9	扫描线 9
ACS_SBBS	右下角的别名
ACS_SBSB	垂直线的别名
ACS_SBSS	右侧 T 型的别名
ACS_SSBB	左下角的别名
ACS_SSBS	底部 T 型的别名
ACS_SSSB	左侧 T 型的别名
ACS_SSSS	交叉或大加号的替代名称
ACS_STERLING	英镑
ACS_TTEE	顶部 T 型
ACS_UARROW	向上箭头
ACS_ULCORNER	左上角
ACS_URCORNER	右上角
ACS_VLINE	垂线

下表列出了预定义的颜色：

常数	颜色
COLOR_BLACK	黑色
COLOR_BLUE	蓝色
COLOR_CYAN	青色（浅绿蓝色）
COLOR_GREEN	绿色
COLOR_MAGENTA	洋红色（紫红色）
COLOR_RED	红色
COLOR_WHITE	白色
COLOR_YELLOW	黄色

16.11 curses.textpad —用于 curses 程序的文本输入控件

curses.textpad 模块提供了一个 *Textbox* 类，该类在 curses 窗口中处理基本的文本编辑，支持一组与 Emacs 类似的键绑定（因此这也适用于 Netscape Navigator, BBedit 6.x, FrameMaker 和许多其他程序）。该模块还提供了一个绘制矩形的函数，适用于容纳文本框或其他目的。

curses.textpad 模块定义了以下函数:

curses.textpad.**rectangle**(*win*, *uly*, *ulx*, *lry*, *lrx*)
绘制一个矩形。第一个参数必须为窗口对象；其余参数均为相对于该窗口的坐标值。第二和第三个参数为要绘制的矩形的左上角的 y 和 x 坐标值；第四和第五个参数为其右下角的 y 和 x 坐标值。将会使用 VT100/IBM PC 形式的字符在可用的终端上（包括 xterm 和大多数其他软件终端模拟器）绘制矩形。在其他情况下则将使用 ASCII 横杠、竖线和加号绘制。

16.11.1 文本框对象

你可以通过如下方式实例化一个 *Textbox*:

class curses.textpad.**Textbox**(*win*)
返回一个文本框控件对象。*win* 参数必须是一个 curses 窗口 对象，文本框将被包含在其中。文本框的编辑光标在初始时位于包含窗口的左上角，坐标值为 (0, 0)。实例的 *stripspaces* 旗标初始时为启用。

Textbox 对象具有以下方法:

edit([*validator*])
这是你通常将使用的入口点。它接受编辑按键直到键入了一个终止按键。如果提供了 *validator*，它必须是一个函数。它将在每次按键时被调用并传入相应的按键作作为形参；命令发送将在结果上执行。此方法会以字符串形式返回窗口内容；是否包括窗口中的空白将受到 *stripspaces* 属性的影响。

do_command(*ch*)
处理单个按键命令。以下是支持的特殊按键:

按键	动作
Control-A	转到窗口的左边缘。
Control-B	光标向左，如果可能，包含前一行。
Control-D	删除光标下的字符。
Control-E	前往右边缘（stripspaces 关闭时）或者行尾（stripspaces 启用时）。
Control-F	向右移动光标，适当时换行到下一行。
Control-G	终止，返回窗口内容。
Control-H	向后删除字符。
Control-J	如果窗口是 1 行则终止，否则插入换行符。
Control-K	如果行为空，则删除它，否则清除到行尾。
Control-L	刷新屏幕。
Control-N	光标向下; 向下移动一行。
Control-O	在光标位置插入一个空行。
Control-P	光标向上; 向上移动一行。

如果光标位于无法移动的边缘，则移动操作不执行任何操作。在可能的情况下，支持以下同义词:

常数	按键
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

所有其他按键将被视为插入给定字符并右移的命令（带有自动折行）。

gather()
以字符串形式返回窗口内容；是否包括窗口中的空白将受到*stripspaces*成员的影响。

stripspaces
此属性是控制窗口中空白解读方式的旗标。当启用时，每一行的末尾空白会被忽略；任何将光标定位至末尾空白的光标动作都将改为前往该行末尾，并且在收集窗口内容时将去除末尾空白。

16.12 curses.ascii —用于 ASCII 字符的工具

curses.ascii 模块提供了一些 ASCII 字符的名称常量以及在各种 ASCII 字符类中执行成员检测的函数。所提供的控制字符常量如下:

名称	含义
NUL	
SOH	标题开始，控制台中断
STX	文本开始
ETX	文本结束
EOT	传输结束
ENQ	查询，附带 ACK 流量控制
ACK	确认
BEL	蜂鸣器
BS	退格

下页继续

表 3 – 续上页

名称	含义
TAB	Tab
HT	TAB 的别名：“水平制表符”
LF	换行
NL	LF 的别名：“新行”
VT	垂直制表符
FF	换页
CR	回车
SO	Shift-out，开始替换字符集
SI	Shift-in，恢复默认字符集
DLE	Data-link escape
DC1	XON，用于流程控制
DC2	Device control 2，阻塞模式流程控制
DC3	XOFF，用于流程控制
DC4	设备控制 4
NAK	否定确认
SYN	同步空闲
ETB	末端传输块
CAN	取消
EM	媒体结束
SUB	替换
ESC	退出
FS	文件分隔符
GS	组分隔符
RS	记录分隔符，块模式终结器
US	单位分隔符
SP	空格
DEL	删除

请注意其中有许多在现今已经没有实际作用。这些助记符是来源于数字计算机之前的电传打印机规范。

此模块提供了下列函数，对应于标准 C 库中的函数:

curses.ascii.**isalnum**(*c*)
检测 ASCII 字母数字类字符；它等价于 `isalpha(c) 或 isdigit(c)`。

curses.ascii.**isalpha**(*c*)
检测 ASCII 字母类字符；它等价于 `isupper(c) or islower(c)`。

curses.ascii.**isascii**(*c*)
检测字符值是否在 7 位 ASCII 集范围内。

curses.ascii.**isblank**(*c*)
检测 ASCII 空白字符；包括空格或水平制表符。

curses.ascii.**iscntrl**(*c*)
检测 ASCII 控制字符（在 0x00 到 0x1f 或 0x7f 范围内）。

curses.ascii.**isdigit**(*c*)
检测 ASCII 十进制数码，即 `'0'` 至 `'9'`。它等价于 `c in string.digits`。

curses.ascii.**isgraph**(*c*)
检测任意 ASCII 可打印字符，不包括空白符。

curses.ascii.**islower**(*c*)
检测 ASCII 小写字母字符。

curses.ascii.**isprint**(*c*)
检测任意 ASCII 可打印字符，包括空白符。

curses.ascii.**ispunct**(*c*)
检测任意 ASCII 可打印字符，不包括空白符或字母数字类字符。

curses.ascii.**isspace**(*c*)
检测 ASCII 空白字符；包括空格，换行，回车，进纸，水平制表和垂直制表。

curses.ascii.**isupper**(*c*)
检测 ASCII 大写字母字符。

curses.ascii.**isxdigit**(*c*)
检测 ASCII 十六进制数码。这等价于 `c in string.hexdigits`。

curses.ascii.**isctrl**(*c*)
检测 ASCII 控制字符（序号值 0 至 31）。

curses.ascii.**ismeta**(*c*)
检测非 ASCII 字符（码位值 0x80 及以上）。

这些函数接受整数或单字符字符串；当参数为字符串时，会先使用内置函数*ord()* 进行转换。

请注意所有这些函数都是检测根据你传入的字符串的字符所生成的码位值；它们实际上完全不会知晓本机的字符编码格式。

以下两个函数接受单字符字符串或整数形式的字节值；它们会返回相同类型的值。

curses.ascii.**ascii**(*c*)
返回对应于 *c* 的下个 7 比特位的 ASCII 值。

curses.ascii.**ctrl**(*c*)
返回对应于给定字符的控制字符（字符比特值会与 0x1f 进行按位与运算）。

curses.ascii.**alt**(*c*)
返回对应于给定 ASCII 字符的 8 比特位字符（字符比特值会与 0x80 进行按位或运算）。

以下函数接受单字符字符串或整数值；它会返回一个字符串。

curses.ascii.**unctrl**(*c*)
返回 ASCII 字符 *c* 的字符串表示形式。如果 *c* 是可打印字符，则字符串为字符本身。如果该字符是控制字符 (0x00–0x1f) 则字符串由一个插入符 (`'^'`) 加相应的大写字母组成。如果该字符是 ASCII 删除符 (0x7f) 则字符串为 `'^?'`。如果该字符设置了元比特位 (0x80)，元比特位会被去除，应用以上规则后将在结果之前添加 `'!'`。

curses.ascii.**controlnames**
一个 33 元素的字符串数据，其中按从 0 (NUL) 到 0x1f (US) 的顺序包含了三十二个 ASCII 控制字符的 ASCII 助记符，另加空格符的助记符 `SP`。

16.13 `curses.panel` —curses 的 panel 栈扩展

面板是具有添加深度功能的窗口，因此它们可以从上至下堆叠为栈，只有显示每个窗口的可见部分会显示出来。面板可以在栈中被添加、上移或下移，也可以被移除。

16.13.1 函数

curses.panel 模块定义了以下函数:

curses.panel.**bottom_panel**()
返回面板栈中的底部面板。

curses.panel.**new_panel**(*win*)
返回一个面板对象，将其与给定的窗口 *win* 相关联。请注意你必须显式地保持所返回的面板对象。如果你不这样做，面板对象会被垃圾回收并从面板栈中被移除。

curses.panel.**top_panel**()
返回面板栈中的顶部面板。

curses.panel.**update_panels**()
在面板栈发生改变后更新虚拟屏幕。这不会调用*curses.doupdate()*，因此你不必自己执行此操作。

16.13.2 Panel 对象

Panel 对象，如上面*new_panel()* 所返回的对象，是带有栈顺序的多个窗口。总是会有一个窗口与确定内容的面板相关联，面板方法会负责窗口在面板栈中的深度。

Panel 对象具有以下方法:

Panel.**above**()
返回当前面板之上的面板。

Panel.**below**()
返回当前面板之下的面板。

Panel.**bottom**()
将面板推至栈底部。

Panel.**hidden**()
如果面板被隐藏（不可见）则返回 `True`，否则返回 `False`。

Panel.**hide**()
隐藏面板。这不会删除对象，它只是让窗口在屏幕上不可见。

Panel.**move**(*y*, *x*)
将面板移至屏幕坐标 `(y, x)`。

Panel.**replace**(*win*)
将与面板相关联的窗口改为窗口 *win*。

Panel.**set_userptr**(*obj*)
将面板的用户指向设为 *obj*。这被用来将任意数据与面板相关联，数据可以是任何 Python 对象。

Panel.**show**()
显示面板（面板可能已被隐藏）。

Panel.**top**()
将面板推至栈顶部。

Panel.**userptr**()
返回面板的用户指针。这可以是任何 Python 对象。

Panel.**window**()
返回与面板相关联的窗口对象。

16.14 platform —获取底层平台的标识数据

示例代码： Lib/platform.py

注解： 特定平台按字母顺序排列，Linux 包括在 Unix 小节之中。

16.14.1 跨平台

platform.**architecture**(*executable=sys.executable*, *bits=''*, *linkage=''*)
查询给定的可执行文件（默认为 Python 解释器二进制码文件）来获取各种架构信息。

返回一个元素 (bits, linkage)，其中包含可执行文件所使用的位架构和链接格式信息。这两个值均以字符串形式返回。

无法确定的值将返回为形参预设所给出的值。如果给出的位数为 ''，则会使用 sizeof(pointer) (或者当 Python 版本 < 1.5.2 时为 sizeof(long)) 作为所支持的指针大小的提示。

此函数依赖于系统的 file 命令来执行实际的操作。这在几乎所有 Unix 平台和某些非 Unix 平台上只有当可执行文件指向 Python 解释器时才可用。当以上要求不满足时将会使用合理的默认值。

注解： 在 Mac OS X（也许还有其他平台）上，可执行文件可能是包含多种架构的通用文件。

要获取当前解释器的“64 位性”，更可靠的做法是查询*sys.maxsize* 属性:

```
is_64bits = sys.maxsize > 2**32
```

---

platform.**machine**()
返回机器类型，例如 'i386'。如果该值无法确定则会返回一个空字符串。

platform.**node**()
返回计算机的网络名称（可能不是完整限定名称!）。如果该值无法确定则会返回一个空字符串。

platform.**platform**(*aliased=0*, *terse=0*)
返回一个标识底层平台的字符串，其中带有尽可能多的有用信息。

输出信息的目标是“人类易读”而非机器易解析。它在不同平台上可能看起来不一致，这是有意为之的。

如果 *aliased* 为真值，此函数将使用各种平台不同与其通常名称的别名来报告系统名称，例如 SunOS 将被报告为 Solaris。*system_alias()* 函数将被用于实现此功能。

将 *terse* 设为真值将导致此函数只返回标识平台所必须的最小量信息。

platform.**processor**()
返回（真实的）处理器名称，例如 'amdk6'。

如果该值无法确定则将返回空字符串。请注意许多平台都不提供此信息或是简单地返回与*machine()* 相同的值。NetBSD 则会提供此信息。

platform.**python_build**()
返回一个元组 (buildno, builddate)，以字符串表示的 Python 编译代码和日期。

platform.**python_compiler**()
返回一个表示用于编译 Python 的编译器的的字符串。

platform.**python_branch**()
返回一个表示 Python 实现的 SCM 分支的字符串。

platform.**python_implementation**()
返回一个标识 Python 实现的字符串。可能的返回值有：‘CPython’，‘IronPython’，‘Jython’，‘PyPy’。

platform.**python_revision**()
返回一个标识 Python 实现的 SCM 修订版的字符串。

platform.**python_version**()
将 Python 版本以字符串 `'major.minor.patchlevel'` 形式返回。

请注意此返回值不同于 Python `sys.version`，它将总是包括 patchlevel (默认为 0)。

platform.**python_version_tuple**()
将 Python 版本以字符串元组 `(major, minor, patchlevel)` 形式返回。

请注意此返回值不同于 Python `sys.version`，它将总是包括 patchlevel (默认为 `'0'`)。

platform.**release**()
返回系统的发布版本，例如 `'2.2.0'` 或 `'NT'`，如果该值无法确定则将返回一个空字符串。

platform.**system**()
Returns the system/OS name, e.g. `'Linux'`, `'Windows'`, or `'Java'`. An empty string is returned if the value cannot be determined.

platform.**system_alias**(*system*, *release*, *version*)
返回别名为某些系统所使用的常见营销名称的 `(system, release, version)`。它还会在可能导致混淆的情况下对信息进行一些重排序操作。

platform.**version**()
返回系统的发布版本信息，例如 `'#3 on degas'`。如果该值无法确定则将返回一个空字符串。

platform.**uname**()
具有高可移植性的 uname 接口。返回包含六个属性的*namedtuple()*: *system*, *node*, *release*, *version*, *machine* 和*processor*。

请注意此函数添加的第六个属性 (*processor*) 并不存在于*os.uname()* 的结果中。并且前两个属性的属性名称也不一致；*os.uname()* 是将它们称为 `sysname` 和 `nodename`。

无法确定的条目会被设为 `''`。

在 3.3 版更改: 将结果从元组改为命名元组。

16.14.2 Java 平台

platform.**java_ver**(*release=''*, *vendor=''*, *vminfo=('', '', '')*, *osinfo=('', '', '')*)
Jython 的版本接口

返回一个元组 `(release, vendor, vminfo, osinfo)`，其中 *vminfo* 为元组 `(vm_name, vm_release, vm_vendor)` 而 *osinfo* 为元组 `(os_name, os_version, os_arch)`。无法确定的值将设为由形参所给出的默认值 (默认均为 `''`)。

16.14.3 Windows 平台

platform.**win32_ver**(*release=''*, *version=''*, *csd=''*, *ptype=''*)
从 Windows 注册表获取额外的版本信息并返回一个元组 (release, version, csd, ptype) 表示 OS 发行版, 版本号, CSD 级别 (Service Pack) 和 OS 类型 (多个/单个处理器)。

一点提示: *ptype* 在单个处理器的 NT 机器上为 'Uniprocessor Free' 而在多个处理器的机器上为 'Multiprocessor Free'。'*Free*' 是指该 OS 版本没有调试代码。它还可能显示 '*Checked*' 表示该 OS 版本使用了调试代码，即检测参数、范围等的代码。

注解: 此函数在安装了 Mark Hammond 的 win32all 包并且为 Python 2.3 或更新版本上效果最佳（此支持是在 Python 2.6 中增加的）。显然它只能在兼容 Win32 的平台上运行。

Win95/98 specific

platform.**popen**(*cmd*, *mode='r'*, *bufsize=-1*)
Portable *popen()* interface. Find a working popen implementation preferring win32pipe.popen(). On Windows NT, win32pipe.popen() should work; on Windows 9x it hangs due to bugs in the MS C library.

3.3 版后已移除: This function is obsolete. Use the *subprocess* module. Check especially the *使用 subprocess 模块替换旧函数* section.

16.14.4 Mac OS 平台

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)
获取 Mac OS 版本信息并将其返回为元组 (release, versioninfo, machine)，其中 *versioninfo* 是一个元组 (version, dev_stage, non_release_version)。

无法确定的条目会被设为 ''。所有元组条目均为字符串。

16.14.5 Unix 平台

platform.**dist**(*distname=''*, *version=''*, *id=''*, *supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...)*)
这是*linux_distribution()* 的另外一个名字。

Deprecated since version 3.5, will be removed in version 3.8: See alternative like the distro package.

platform.**linux_distribution**(*distname=''*, *version=''*, *id=''*, *supported_dists=('SuSE', 'debian', 'redhat', 'mandrake', ...)*, *full_distribution_name=1*)
Tries to determine the name of the Linux OS distribution name.

supported_dists may be given to define the set of Linux distributions to look for. It defaults to a list of currently supported Linux distributions identified by their release file name.

If full_distribution_name is true (default), the full distribution read from the OS is returned. Otherwise the short name taken from supported_dists is used.

Returns a tuple (distname,version,id) which defaults to the args given as parameters. id is the item in parentheses after the version number. It is usually the version codename.

Deprecated since version 3.5, will be removed in version 3.8: See alternative like the distro package.

platform.**libc_ver**(*executable=sys.executable*, *lib=''*, *version=''*, *chunksize=16384*)
尝试确定可执行文件（默认为 Python 解释器）所链接到的 libc 版本。返回一个字符串元组 (lib, version)，当查找失败时其默认值将设为给定的形参值。

请注意此函数对于不同 libc 版本向可执行文件添加符号的方式有深层的关联，可能仅适用于使用 **gcc** 编译出来的可执行文件。

文件将按 *chunksize* 个字节的分块来读取和扫描。

16.15 errno —标准 errno 系统符号

本模块提供标准的 errno 系统符号。每个符号的值是其对应的整数值。符号的名称和描述来自 linux/include/errno.h，应该是非常全面的。

errno.**errorcode**
提供从 errno 值到底层系统中字符串名称的映射的字典。例如，errno.errorcode[errno.EPERM] 映射为 'EPERM'。

如果要将数字的错误代码转换为错误信息，请使用*os.strerror()*。

在下面的列表中，当前平台上没有使用的符号没有被本模块定义。已定义的符号的具体列表可参见 errno.errorcode.keys()。可用的符号包括：

errno.**EPERM**
操作不被允许

errno.**ENOENT**
无此文件或目录

errno.**ESRCH**
无此进程

errno.**EINTR**
系统调用中断。

参见：

此错误被映射到异常*InterruptedError*。

errno.**EIO**
I/O 错误

errno.**ENXIO**
无此设备或地址

errno.**E2BIG**
参数列表过长

errno.**ENOEXEC**
执行格式错误

errno.**EBADF**
错误的文件号

errno.**ECHILD**
无子进程

errno.**EAGAIN**
重试

errno.**ENOMEM**
内存不足

errno.**EACCES**
没有权限

errno.**EFAULT**
错误的地址

errno.**ENOTBLK**
需要块设备

errno.**EBUSY**
设备或资源忙

errno.**EEXIST**
文件已存在

errno.**EXDEV**
跨设备链接

errno.**ENODEV**
无此设备

errno.**ENOTDIR**
不是目录

errno.**EISDIR**
是目录

errno.**EINVAL**
无效的参数

errno.**ENFILE**
文件表溢出

errno.**EMFILE**
打开的文件过多

errno.**ENOTTY**
不是打字机

errno.**ETXTBSY**
文本文件忙

errno.**EFBIG**
文件过大

errno.**ENOSPC**
设备已无可用空间

errno.**ESPIPE**
非法查找

errno.**EROFS**
只读文件系统

errno.**EMLINK**
链接过多

errno.**EPIPE**
管道已损坏

errno.**EDOM**
数学参数超出函数范围

errno.**ERANGE**
数学运算结果无法表示

errno.**EDEADLK**
将发生资源死锁

errno.**ENAMETOOLONG**
文件名过长

errno.**ENOLCK**
没有可用的记录锁

errno.**ENOSYS**
功能未实现

errno.**ENOTEMPTY**
目录非空

errno.**ELOOP**
遇到过多的符号链接

errno.**EWOULDBLOCK**
操作将阻塞

errno.**ENOMSG**
没有所需类型的消息

errno.**EIDRM**
标识符被移除

errno.**ECHRNG**
信道编号超出范围

errno.**EL2NSYNC**
级别 2 未同步

errno.**EL3HLT**
级别 3 已停止

errno.**EL3RST**
级别 3 重置

errno.**ELNRNG**
链接编号超出范围

errno.**EUNATCH**
未附加协议驱动

errno.**ENOCSI**
没有可用的 CSI 结构

errno.**EL2HLT**
级别 2 已停止

errno.**EBADE**
无效的交换

errno.**EBADR**
无效的请求描述符

errno.**EXFULL**
交换已满

errno.**ENOANO**
没有阳极

errno.**EBADRQC**
无效的请求码 ·

errno.**EBADSLT**
无效的槽位

errno.**EDEADLOCK**
文件锁定死锁错误

errno.**EBFONT**
错误的字体文件格式

errno.**ENOSTR**
设备不是流

errno.**ENODATA**
没有可用的数据

errno.**ETIME**
计时器已到期

errno.**ENOSR**
流资源不足

errno.**ENONET**
机器不在网络上

errno.**ENOPKG**
包未安装

errno.**EREMOTE**
对象是远程的

errno.**ENOLINK**
链接已被切断

errno.**EADV**
广告错误

errno.**ESRMNT**
挂载错误

errno.**ECOMM**
发送时通讯错误

errno.**EPROTO**
协议错误

errno.**EMULTIHOP**
已尝试多跳

errno.**EDOTDOT**
RFS 专属错误

errno.**EBADMSG**
非数据消息

errno.**EOVERFLOW**
值相对于已定义数据类型过大

errno.**ENOTUNIQ**
名称在网络上不唯一

errno.**EBADFD**
文件描述符处于错误状态

errno.**EREMCHG**
远端地址已改变

errno.**ELIBACC**
无法访问所需的共享库

errno.**ELIBBAD**
访问已损坏的共享库

errno.**ELIBSCN**
a.out 中的.lib 部分已损坏

errno.**ELIBMAX**
尝试链接过多的共享库

errno.**ELIBEXEC**
无法直接执行共享库

errno.**EILSEQ**
非法字节序列

errno.**ERESTART**
已中断系统调用需要重启

errno.**ESTRPIPE**
流管道错误

errno.**EUSERS**
用户过多

errno.**ENOTSOCK**
在非套接字上执行套接字操作

errno.**EDESTADDRREQ**
需要目标地址

errno.**EMSGSIZE**
消息过长

errno.**EPROTOTYPE**
套接字的协议类型错误

errno.**ENOPROTOOPT**
协议不可用

errno.**EPROTONOSUPPORT**
协议不受支持

errno.**ESOCKTNOSUPPORT**
套接字类型不受支持

errno.**EOPNOTSUPP**
操作在传输端点上不受支持

errno.**EPFNOSUPPORT**
协议族不受支持

errno.**EAFNOSUPPORT**
地址族不受协议支持

errno.**EADDRINUSE**
地址已被使用

errno.**EADDRNOTAVAIL**
无法分配要求的地址

errno.**ENETDOWN**
网络已断开

errno.**ENETUNREACH**
网络不可达

errno.**ENETRESET**
网络因重置而断开连接

errno.**ECONNABORTED**
软件导致连接中止

errno.**ECONNRESET**
连接被对方重置

errno.**ENOBUFS**
没有可用的缓冲区空间

errno.**EISCONN**
传输端点已连接

errno.**ENOTCONN**
传输端点未连接

errno.**ESHUTDOWN**
传输端点关闭后无法发送

errno.**ETOOMANYREFS**
引用过多：无法拼接

errno.**ETIMEDOUT**
连接超时

errno.**ECONNREFUSED**
连接被拒

errno.**EHOSTDOWN**
主机已关闭

errno.**EHOSTUNREACH**
没有到主机的路由

errno.**EALREADY**
操作已在进行

errno.**EINPROGRESS**
操作正在进行

errno.**ESTALE**
过期的 NFS 文件句柄

errno.**EUCLEAN**
结构需要清理

errno.**ENOTNAM**
不是 XENIX 命名类型文件

errno.**ENAVAIL**
没有可用的 XENIX 信标

errno.**EISNAM**
是命名类型文件

errno.**EREMOTEIO**
远程 I/O 错误

errno.**EDQUOT**
超出配额

16.16 ctypes —Python 的外部函数库

ctypes 是 Python 的外部函数库。它提供了与 C 兼容的数据类型，并允许调用 DLL 或共享库中的函数。可使用该模块以纯 Python 形式对这些库进行封装。

16.16.1 ctypes 教程

注意：在本教程中的示例代码使用*doctest* 进行过测试，保证其正确运行。由于有些代码在 Linux，Windows 或 Mac OS X 下的表现不同，这些代码会在 doctest 中包含相关的指令注解。

注意：部分示例代码引用了 ctypes *c_int* 类型。在 `sizeof(long) == sizeof(int)` 的平台上此类型是*c_long* 的一个别名。所以，在程序输出*c_long* 而不是你期望的*c_int* 时不必感到迷惑—它们实际上是同一种类型。

载入动态连接库

ctypes 导出了 *cdll* 对象，在 Windows 系统中还导出了 *windll* 和 *oledll* 对象用于载入动态连接库。

通过操作这些对象的属性，你可以载入外部的动态链接库。*cdll* 载入按标准的 `cdecl` 调用协议导出的函数，而 *windll* 导入的库按 `stdcall` 调用协议调用其中的函数。*oledll* 也按 `stdcall` 调用协议调用其中的函数，并假定该函数返回的是 Windows `HRESULT` 错误代码，并当函数调用失败时，自动根据该代码甩出一个*OSError* 异常。

在 3.3 版更改: 原来在 Windows 下甩出的异常类型*WindowsError* 现在是*OSError* 的一个别名。

这是一些 Windows 下的例子。注意：`msvcrt` 是微软 C 标准库，包含了大部分 C 标准函数，这些函数都是以 cdecl 调用协议进行调用的。

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows 会自动添加通常的 .dll 文件扩展名。

注解： 通过 cdll.msvcrt 调用的标准 C 函数，可能会导致调用一个过时的，与当前 Python 所不兼容的函数。因此，请尽量使用标准的 Python 函数，而不要使用 msvcrt 模块。

在 Linux 下，必须使用 包含文件扩展名的文件名来导入共享库。因此不能简单使用对象属性的方式来导入库。因此，你可以使用方法 LoadLibrary()，或构造 CDLL 对象来导入库。

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

操作导入的动态链接库中的函数

通过操作 dll 对象的属性来操作这些函数。

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

注意：Win32 系统的动态库，比如 kernel32 和 user32，通常会同时导出同一个函数的 ANSI 版本和 UNICODE 版本。UNICODE 版本通常会在名字最后以 W 结尾，而 ANSI 版本的则以 A 结尾。win32 的 GetModuleHandle 函数会根据一个模块名返回一个 模块句柄，该函数暨同时包含这样的两个版本的原型函数，并通过宏 UNICODE 是否定义，来决定宏 GetModuleHandle 导出的是哪个具体函数。

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll 不会通过这样的魔法手段来帮你决定选择哪一种函数，你必须显式的调用 GetModuleHandleA 或 GetModuleHandleW，并分别使用字节对象或字符串对象作参数。

有时候，dlls 的导出的函数名不符合 Python 的标识符规范，比如 "??2@YAPAXI@Z"。此时，你必须使用*getattr()* 方法来获得该函数。

```
>>> getattr(cdll.msvcrt, "??2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

Windows 下，有些 dll 导出的函数没有函数名，而是通过其顺序号调用。对此类函数，你也可以通过 dll 对象的数值索引来操作这些函数。

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

调用函数

你可以貌似是调用其它 Python 函数那样直接调用这些函数。在这个例子中，我们调用了 `time()` 函数，该函数返回一个系统时间戳（从 Unix 时间起点到现在的秒数），而 “GetModuleHandleA()“ 函数返回一个 win32 模块句柄。

This example calls both functions with a NULL pointer (`None` should be used as the NULL pointer):

```
>>> print(libc.time(None))
1150640792
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

注解： *ctypes* may raise a *ValueError* after calling the function, if it detects that an invalid number of arguments were passed. This behavior should not be relied upon. It is deprecated in 3.6.2, and will be removed in 3.7.

如果你用 `cdecl` 调用方式调用 `stdcall` 约定的函数，则会甩出一个异常*ValueError*。反之亦然。

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

你必须阅读这些库的头文件或说明文档来确定它们的正确的调用协议。

在 Windows 中，*ctypes* 使用 win32 结构化异常处理来防止由于在调用函数时使用非法参数导致的程序崩溃。

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

然而，总有许多办法，通过调用*ctypes* 使得 Python 程序崩溃。因此，你必须小心使用。*faulthandler* 模块可以用于帮助诊断程序崩溃的原因。（比如由于错误的 C 库函数调用导致的段错误）。

None，整型，字节对象和（UNICODE）字符串是仅有的可以直接作为函数参数使用的四种 Python 本地数据类型。*None*‘作为 C 的空指针 (NULL)，字节和字符串类型作为一个指向其保存数据的内存块指针（char * 或 wchar_t *)。Python 的整型则作为平台默认的 C 的 int 类型，他们的数值被截断以适应 C 类型的整型长度。

在我们开始调用函数前，我们必须先了解作为函数参数的*ctypes* 数据类型。

基础数据类型

ctypes 定义了一些和 C 兼容的基本数据类型：

ctypes 类型	C 类型	Python 类型
c_bool	_Bool	bool (1)
c_char	char	单字符字节对象
c_wchar	wchar_t	单字符字符串
c_byte	char	整型
c_ubyte	unsigned char	整型
c_short	short	整型
c_ushort	unsigned short	整型
c_int	int	整型
c_uint	unsigned int	整型
c_long	long	整型
c_ulong	unsigned long	整型
c_longlong	__int64 或 long long	整型
c_ulonglong	unsigned __int64 或 unsigned long long	整型
c_size_t	size_t	整型
c_ssize_t	ssize_t 或 Py_ssize_t	整型
c_float	float	浮点数
c_double	double	浮点数
c_longdouble	long double	浮点数
c_char_p	char * (以 NUL 结尾)	字节串对象或 None
c_wchar_p	wchar_t * (以 NUL 结尾)	字符串或 None
c_void_p	void *	int 或 None

(1) 构造函数接受任何具有真值的对象。

所有这些类型都可以通过使用正确类型和值的可选初始值调用它们来创建：

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

由于这些类型是可变的，它们的值也可以在以后更改：

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
```

(下页继续)

(续上页)

```
>>> print(i.value)
-99
>>>
```

当给指针类型的对象*c_char_p*, *c_wchar_p* 和*c_void_p* 等赋值时，将改变它们所指向的 内存地址，而不是它们所指向的内存区域的 内容 (这是理所当然的，因为 Python 的 bytes 对象是不可变的):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)              # the memory location has changed
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                # first object is unchanged
Hello, World
>>>
```

但你要注意不能将它们传递给会改变指针所指内存的函数。如果你需要可改变的内存块，ctypes 提供了*create_string_buffer()* 函数，它提供多种方式创建这种内存块。当前的内存块内容可以通过 `raw` 属性存取，如果你希望将它作为 NUL 结束的字符串，请使用 `value` 属性:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)            # create a 3 byte buffer, initialized to␣
↪NUL bytes
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")     # create a buffer containing a NUL␣
↪terminated string
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

create_string_buffer() 函数替代以前的 ctypes 版本中的 `c_buffer()` 函数 (仍然可当作别名使用) 和 `c_string()` 函数。*create_unicode_buffer()* 函数创建包含 unicode 字符的可变内存块，与之对应的 C 语言类型是 `wchar_t`。

调用函数，继续

注意 printf 将打印到真正标准输出设备，而 * 不是 * *sys.stdout*，因此这些实例只能在控制台提示符下工作，而不能在 *IDLE* 或 *PythonWin* 中运行。

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: Don't know how to convert parameter 2
>>>
```

正如前面所提到过的，除了整数、字符串以及字节串之外，所有的 Python 类型都必须使用它们对应的*ctypes*类型包装，才能够被正确地转换为所需的 C 语言类型。

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

使用自定义的数据类型调用函数

你也可以通过自定义*ctypes*参数转换方式来允许自定义类型作为参数。*ctypes*会寻找 _as_parameter_ 属性并使用它作为函数参数。当然，它必须是数字、字符串或者二进制字符串:

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

如果你不想把实例的数据存储到 _as_parameter_ 属性。可以通过定义*property*函数计算出这个属性。

指定必选参数的类型 (函数原型)

可以通过设置 `argtypes` 属性的方法指定从 DLL 中导出函数的必选参数类型。

`argtypes` 必须是一个 C 数据类型的序列 (这里的 `printf` 可能不是个好例子，因为它是变长参数，而且每个参数的类型依赖于格式化字符串，不过尝试这个功能也很方便):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

指定数据类型可以防止不合理的参数传递 (就像 C 函数的函数签名)，并且会自动尝试将参数转换为需要的类型:

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
13
>>>
```

如果你想通过自定义类型传递参数给函数，必须实现 `from_param()` 类方法，才能够将此自定义类型用于 `argtypes` 序列。`from_param()` 类方法接受一个 Python 对象作为函数输入，它应该进行类型检查或者其他必要的操作以保证接收到的对象是合法的，然后返回这个对象，或者它的 `_as_parameter_` 属性，或者其他你想要传递给 C 函数的参数。这里也一样，返回的结果必须是整型、字符串、二进制字符串、*ctypes* 类型，或者一个具有 `_as_parameter_` 属性的对象。

返回类型

默认情况下都会假定函数返回 C `int` 类型。其他返回类型可以通过设置函数对象的 `restype` 属性来指定。

这是个更高级的例子，它调用了 `strchr` 函数，这个函数接收一个字符串指针以及一个字符作为参数，返回另一个字符串指针。

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p     # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

如果希望避免上述的 `ord("x")` 调用，可以设置 `argtypes` 属性，第二个参数就会将单字符的 Python 二进制字符对象转换为 C 字符:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
'def'
>>> strchr(b"abcdef", b"def")
```

(下页继续)

(续上页)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
'def'
>>>
```

如果外部函数返回了一个整数，你也可以使用要给可调用的 Python 对象（比如函数或者类）作为 `restype` 属性的值。将会以 C 函数返回的 *整数*对象作为参数调用这个可调用对象，执行后的结果作为最终函数返回值。这在错误返回值校验和自动抛出异常等方面比较有用。

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>
```

`WinError` 函数可以调用 Windows 的 `FormatMessage()` API 获取错误码的字符串说明，然后 *返回*一个异常。`WinError` 接收一个可选的错误码作为参数，如果没有的话，它将调用*GetLastError()* 获取错误码。

请注意，使用 `errcheck` 属性可以实现更强大的错误检查手段；详情请见参考手册。

传递指针 (或者传递引用)

有时候 C 函数接口可能由于要往某个地址写入值，或者数据太大不适合作为值传递，从而希望接收一个 *指针*作为数据参数类型。这和 *传递参数引用*类似。

ctypes 暴露了*byref()* 函数用于通过引用传递参数，使用*pointer()* 函数也能达到同样的效果，只不过*pointer()* 需要更多步骤，因为它要先构造一个真实指针对象。所以在 Python 代码本身不需要使用这个指针对象的情况下，使用*byref()* 效率更高。

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

结构体和联合

结构体和联合必须继承自*ctypes* 模块中的*Structure* 和*Union* 。子类必须定义 _fields_ 属性。_fields_ 是一个二元组列表，二元组中包含 *field name* 和 *field type* 。

type 字段必须是一个*ctypes* 类型，比如*c_int*，或者其他*ctypes* 类型: 结构体、联合、数组、指针。

这是一个简单的 POINT 结构体，它包含名称为 *x* 和 *y* 的两个变量，还展示了如何通过构造函数初始化结构体。

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many initializers
>>>
```

当然，你可以构造更复杂的结构体。一个结构体可以通过设置 type 字段包含其他结构体或者自身。

这是以一个 RECT 结构体，他包含了两个 POINT ，分别叫 *upperleft* 和 *lowerright*:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

嵌套结构体可以通过几种方式构造初始化:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

可以通过 类获取字段*descriptor* ，它能提供很多有用的调试信息。

```
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>
>>>
```

警告： *ctypes* 不支持带位域的结构体、联合以值的方式传给函数。这可能在 32 位 x86 平台上可以正常工作，但是对于一般情况，这种行为是未定义的。带位域的结构体、联合应该总是通过指针传递给函数。

结构体/联合字段对齐及字节顺序

By default, Structure and Union fields are aligned in the same way the C compiler does it. It is possible to override this behavior be specifying a _pack_ class attribute in the subclass definition. This must be set to a positive integer and specifies the maximum alignment for the fields. This is what #pragma pack(n) also does in MSVC.

ctypes 中的结构体和联合使用的是本地字节序。要使用非本地字节序，可以使用*BigEndianStructure*, *LittleEndianStructure*, BigEndianUnion, and LittleEndianUnion 作为基类。这些类不能包含指针字段。

结构体和联合中的位域

结构体和联合中是可以包含位域字段的。位域只能用于整型字段，位长度通过 _fields_ 中的第三个参数指定:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16)
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

数组

数组是一个序列，包含指定个数元素，且必须类型相同。

创建数组类型的推荐方式是使用一个类型乘以一个正数:

```
TenPointsArrayType = POINT * 10
```

下面是一个构造的数据案例，结构体中包含了 4 个 POINT 和一些其他东西。

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

和平常一样，通过调用它创建实例:

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

以上代码会打印几行 0 0，因为数组内容被初始化为 0.

也能通过指定正确类型的数据来初始化:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

指针

指针可以通过*ctypes* 中的*pointer()* 函数进行创建:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

指针实例拥有*contents* 属性，它存储了指针指向的真实对象，如上面的 `i` 对象:

```
>>> pi.contents
c_long(42)
>>>
```

注意*ctypes* 并没有 OOR （返回原始对象），每次访问这个属性时都会构造返回一个新的相同对象:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

将这个指针的 contents 属性赋值为另一个*c_int* 实例将会导致该指针指向该实例的内存地址:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

指针对象也可以通过整数下标进行访问:

```
>>> pi[0]
99
>>>
```

通过整数下标赋值可以改变内容。

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

使用 0 以外的索引也是合法的，但是你必须确保这么做的后果，就像 C 语言中: 你可以访问或者修改任意内存内容。通常只会在函数接收指针是才会使用这种特性，而且你 知道这个指针指向的是一个数组而不是单个值。

内部细节, *pointer()* 函数不只是创建了一个指针实例，它首先创建了一个指针 类型。这是通过调用*POINTER()* 函数实现的，它接收*ctypes* 类型为参数，返回一个新的类型:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

无参调用指针类型可以创建一个 NULL 指针。NULL 指针的布尔值是 False

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

解引用指针的时候，*ctypes* 会帮你检测是否指针为 NULL (但是解引用无效的非 NULL 指针仍会导致 Python 崩溃):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

类型强制转换

通常情况下, ctypes 具有严格的类型检查。这代表着, 如果在函数 argtypes 中或者结构体定义成员中有 POINTER(c_int) 类型，只有相同类型的实例才会被接受。也有一些例外。比如，你可以传递兼容的数组实例给指针类型。所以，对于 POINTER(c_int) ，ctypes 也可以接受 c_int 类型的数组:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
...
1
```

(下页继续)

(续上页)

```
2
3
>>>
```

另外，如果一个函数 argtypes 列表中的参数显式的定义为指针类型 (如 POINTER(c_int))，指针所指向的类型 (这个例子中是 c_int) 也可以传递给函数。ctypes 会自动调用对应的*byref()* 转换。

可以给指针内容赋值为 None 将其设置为 Null

```
>>> bar.values = None
>>>
```

有时候你拥有一个不完整的类型。在 C 中，你可以将一个类型强制转换为另一个。*ctypes* 中的 a *cast()* 函数提供了相同的功能。上面的结构体 Bar 的 value 字段接收 POINTER(c_int) 指针或者*c_int* 数组，但是不能接受其他类型的实例:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

这种情况下, 需要手动使用*cast()* 函数。

cast() 函数可以将一个指针实例强制转换为另一种 ctypes 类型。*cast()* 接收两个参数，一个 ctypes 指针对象或者可以被转换为指针的其他类型对象，和一个 ctypes 指针类型。返回第二个类型的一个实例，该返回实例和第一个参数指向同一片内存空间:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

所以*cast()* 可以用来给结构体 Bar 的 values 字段赋值:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

不完整类型

不完整类型即还没有定义成员的结构体、联合或者数组。在 C 中，它们通常用于前置声明，然后在后面定义:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

直接翻译成 ctypes 的代码如下，但是这行不通:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

因为新的 `cell` 类在 class 语句结束之前还没有完成定义。在*ctypes* 中，我们可以先定义 `cell` 类，在 class 语句结束之后再设置 `_fields_` 属性:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

Lets try it. We create two instances of `cell`, and let them point to each other, and finally follow the pointer chain a few times:

```
>>> c1 = cell()
>>> c1.name = "foo"
>>> c2 = cell()
>>> c2.name = "bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

回调函数

ctypes 允许创建一个指向 Python 可调用对象的 C 函数。它们有时候被称为 *回调函数*。

首先，你必须为回调函数创建一个类，这个类知道调用约定，包括返回值类型以及函数接收的参数类型及个数。

CFUNCTYPE() 工厂函数使用 `cdecl` 调用约定创建回调函数类型。在 Windows 上，*WINFUNCTYPE()* 工厂函数使用 `stdcall` 调用约定为回调函数创建类型。

这些工厂函数都是用返回值类型作为第一个参数，回掉函数的参数类型作为剩余参数。

这里展示一个使用 C 标准库函数 `qsort()` 的例子，它使用一个回掉函数对数据进行排序。`qsort()` 将用来给整数数组排序:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
```

(下页继续)

(续上页)

```
>>> qsort.restype = None
>>>
```

qsort() 必须接收的参数，一个指向待排序数据的指针，元素个数，每个元素的大小，以及一个指向排序函数的指针，即回调函数。然后回调函数接收两个元素的指针，如果第一个元素小于第二个，则返回一个负整数，如果相等则返回 0，否则返回一个正整数。

所以，我们的回调函数要接收两个整数指针，返回一个整数。首先我们创建回调函数的 类 型

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

首先，这是一个简单的回调，它会显示传入的值:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

结果:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

现在我们可以比较两个元素并返回有用的结果了:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

我们可以轻易地验证，现在数组是有序的了:

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

这些工厂函数可以当作装饰器工厂，所以可以这样写:

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

注解： 请确保你维持*CFUNCTYPE()* 对象的引用与它们在 C 代码中的使用期一样长。*ctypes* 不会确保这一点，而如果你不这样做，它们可能会被垃圾回收，导致你的程序在执行回调函数时发生崩溃。

注意，如果回调函数在 Python 之外的另外一个线程使用 (比如，外部代码调用这个回调函数)，ctypes 会在每一次调用上创建一个虚拟 Python 线程。这个行为在大多数情况下是合理的，但也意味着如果有数据使用*threading.local* 方式存储，将无法访问，就算它们是在同一个 C 线程中调用的。

访问 dll 中导出的值

一些动态链接库不仅仅导出函数，也会导出变量。一个例子就是 Python 库本身的 `Py_OptimizeFlag`，根据启动选项 `-O`、`-OO` 的不同，它是值可能为 0、1、2 的整型。

ctypes 可以通过 `in_dll()` 类方法访问这类变量。*pythonapi* 是用于访问 Python C 接口的预定义符号:

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print(opt_flag)
c_long(0)
>>>
```

如果解释器使用 `-O` 选项启动，这个例子会打印 `c_long(1)`，如果使用 `-OO` 启动，则会打印 `c_long(2)`。

一个扩展例子, 同时也展示了使用指针访问 Python 导出的 `PyImport_FrozenModules` 指针对象。

对文档中这个值的解释说明

> This pointer is initialized to point to an array of `struct _frozen` records, terminated by one whose members are all *NULL* or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

这足以证明修改这个指针是很有用的。为了让实例大小不至于太长，这里只展示如何使用*ctypes* 读取这个表:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

我们定义了 `struct _frozen` 数据类型，接着就可以获取这张表的指针了:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

Since `table` is a `pointer` to the array of `struct_frozen` records, we can iterate over it, but we just have to make sure that our loop terminates, because pointers have no size. Sooner or later it would probably crash with an access violation or whatever, so it's better to break out of the loop when we hit the NULL entry:

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
__hello__ 161
__phello__ -161
__phello__.spam 161
>>>
```

The fact that standard Python has a frozen module and a frozen package (indicated by the negative size member) is not well known, it is only used for testing. Try it out with `import __hello__` for example.

意外

ctypes 也有自己的边界，有时候会发生一些意想不到的事情。

比如下面的例子:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

嗯。我们预想应该打印 `3 4 1 2` 。但是为什么呢? 这是 `rc.a, rc.b = rc.b, rc.a` 这行代码展开后的步骤:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

注意 `temp0` 和 `temp1` 对象始终引用了对象 `rc` 的内容。然后执行 `rc.a = temp0` 会把 `temp0` 的内容拷贝到 `rc` 的空间。这也改变了 `temp1` 的内容。最终导致赋值语句 `rc.b = temp1` 没有产生预想的效果。

记住，访问被包含在结构体、联合、数组中的对象并不会将其 复制出来，而是得到了一个代理对象，它是对根对象的内部内容进行了一层包装。

Another example that may behave different from what one would expect is this:

```
>>> s = c_char_p()
>>> s.value = "abc def ghi"
>>> s.value
'abc def ghi'
>>> s.value is s.value
False
>>>
```

为什么这里打印了 `False`？ctypes 实例是一些内存块加上一些用于访问这些内存块的*descriptor* 组成。将 Python 对象存储在内存块并不会存储对象本身，而是存储了对象的 内容。每次访问对象的内容都会构造一个新的 Python 对象。

变长数据类型

ctypes 对变长数组和结构体提供了一些支持。

The *resize()* function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a *ValueError* is raised if this is tried:

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

这非常好，但是要怎么访问数组中额外的元素呢？因为数组类型已经定义包含 4 个元素，women 访问新增元素会产生以下错误:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

使用*ctypes* 访问变长数据类型的一个可行方法是利用 Python 的动态特性，根据具体情况，在知道这个数据的大小后，(重新) 指定这个数据的类型。

16.16.2 ctypes 参考手册

寻找动态链接库

在编译型语言中，动态链接库会在编译、链接或者程序运行时访问。

The purpose of the `find_library()` function is to locate a library in a way similar to what the compiler or runtime loader does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

`ctypes.util` 模块提供了一个函数，可以帮助确定需要加载的库。

`ctypes.util.`**`find_library`**(*name*)
尝试寻找一个库然后返回其路径名，*name* 是库名称, 且去除了 *lib* 等前缀和 `.so` 、 `.dylib` 、版本号等后缀 (这是 posix 连接器 `-l` 选项使用的格式)。如果没有找到对应的库，则返回 `None` 。

确切的功能取决于系统。

在 Linux 上，`find_library()` 会尝试运行外部程序 (`/sbin/ldconfig`, `gcc`, `objdump` 以及 `ld`) 来寻找库文件。返回库文件的文件名。

在 3.6 版更改: 在 Linux 上，如果其他方式找不到的话，会使用环境变量 `LD_LIBRARY_PATH` 搜索动态链接库。

这是一些例子:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

在 OS X 上，`find_library()` 会尝试几种预定义的命名方案和路径来查找库，如果成功，则返回完整的路径名:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

在 Windows 上，`find_library()` 在系统路径中搜索，然后返回全路径，但是如果没有预定义的命名方案，`find_library("c")` 调用会返回 `None`

使用*ctypes* 包装动态链接库，更好的方式 可能是在开发的时候就确定名称，然后硬编码到包装模块中去，而不是在运行时使用 `find_library()` 寻找库。

加载动态链接库

有很多方式可以将动态链接库加载到 Python 进程。其中之一是实例化以下类的其中一个:

class ctypes.**CDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)
此类的实例即已加载的动态链接库。库中的函数使用标准 C 调用约定，并假定返回 `int` 。

class ctypes.**OleDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)
仅 Windows : 此类的实例即加载好的动态链接库，其中的函数使用 `stdcall` 调用约定，并且假定返回 windows 指定的*HRESULT* 返回码。*HRESULT* 的值包含的信息说明函数调用成功还是失败，以及额外错误码。如果返回值表示失败，会自动抛出*OSError* 异常。

在 3.3 版更改: 以前是引发*WindowsError*。

class ctypes.**WinDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*)
仅 Windows: 此类的实例即加载好的动态链接库，其中的函数使用 `stdcall` 调用约定，并假定默认返回 `int` 。

在 Windows CE 上，只能使用 stdcall 调用约定，为了方便，*WinDLL* 和*OleDLL* 在这个平台上都使用标准调用约定。

调用动态库导出的函数之前，Python 会释放*global interpreter lock* ，并在调用后重新获取。

class ctypes.**PyDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*)
这个类实例的行为与*CDLL* 类似，只不过 不会在调用函数的时候释放 GIL 锁，且调用结束后会检查 Python 错误码。如果错误码被设置，会抛出一个 Python 异常。

所以，它只在直接调用 Python C 接口函数的时候有用。

通过使用至少一个参数（共享库的路径名）调用它们，可以实例化所有这些类。也可以传入一个已加载的动态链接库作为 `handler` 参数，其他情况会调用系统底层的 `dlopen` 或 `LoadLibrary` 函数将库加载到进程，并获取其句柄。

mode 可以指定库加载方式。详情请参见 *dlopen(3)* 手册页。在 Windows 上，会忽略 *mode* ，在 posix 系统上，总是会加上 RTLD_NOW ，且无法配置。

use_errno 参数如果设置为 true，可以启用 ctypes 的机制，通过一种安全的方法获取系统的*errno* 错误码。*ctypes* 维护了一个线程局部变量，它是系统*errno* 的一份拷贝；如果调用了使用 `use_errno=True` 创建的外部函数，*errno* 的值会与 ctypes 自己拷贝的那一份进行交换，函数执行完后立即再交换一次。

The function *ctypes.get_errno()* returns the value of the ctypes private copy, and the function *ctypes.set_errno()* changes the ctypes private copy to a new value and returns the former value.

use_last_error 参数如果设置为 true，可以在 Windows 上启用相同的策略，它是通过 Windows API 函数*GetLastError()* 和 `SetLastError()` 管理的。*ctypes.get_last_error()* 和*ctypes.set_last_error()* 可用于获取和设置 ctypes 自己维护的 windows 错误码拷贝。

ctypes.**RTLD_GLOBAL**
用于 *mode* 参数的标识值。在此标识不可用的系统上，它被定义为整数 0。

ctypes.**RTLD_LOCAL**
Flag to use as *mode* parameter. On platforms where this is not available, it is the same as *RTLD_GLOBAL*.

ctypes.**DEFAULT_MODE**
加载动态链接库的默认模式。在 OSX 10.3 上，它是 *RTLD_GLOBAL* ，其余系统上是 *RTLD_LOCAL* 。

Instances of these classes have no public methods. Functions exported by the shared library can be accessed as attributes or by index. Please note that accessing the function through an attribute caches the result and therefore accessing it repeatedly returns the same object each time. On the other hand, accessing it through an index returns a new object each time:

```
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

还有下面这些属性可用，他们的名称以下划线开头，以避免和导出函数重名:

PyDLL.**_handle**
用于访问库的系统句柄。

PyDLL.**_name**
传入构造函数的库名称。

共享库也可以通用使用一个预制对象来加载，这种对象是*LibraryLoader* 类的实例，具体做法或是通过调用 LoadLibrary() 方法，或是通过将库作为加载实例的属性来提取。

class ctypes.**LibraryLoader**(*dlltype*)
加载共享库的类。*dlltype* 应当为*CDLL*, *PyDLL*, *WinDLL* 或*OleDLL* 类型之一。

__getattr__() 具有特殊的行为：它允许通过将一个共享库作为库加载器实例的属性进行访问来加载它。加载结果将被缓存，因此重复的属性访问每次都会返回相同的库。

LoadLibrary(*name*)
加载一个共享库到进程中并将其返回。此方法总是返回一个新的库实例。

可用的预制库加载器有如下这些:

ctypes.**cdll**
创建*CDLL* 实例。

ctypes.**windll**
仅 Windows 中: 创建*WinDLL* 实例.

ctypes.**oledll**
仅 Windows 中: 创建*OleDLL* 实例。

ctypes.**pydll**
创建*PyDLL* 实例。

要直接访问 C Python api，可以使用一个现成的 Python 共享库对象:

ctypes.**pythonapi**
一个*PyDLL* 的实例，它将 Python C API 函数作为属性公开。请注意所有这些函数都应返回 C int，当然这也不是绝对的，因此你必须分配正确的 restype 属性以使用这些函数。

外部函数

正如之前小节的说明，外部函数可作为被加载共享库的属性来访问。用此方式创建的函数对象默认接受任意数量的参数，接受任意 ctypes 数据实例作为参数，并且返回库加载器所指定的默认结果类型。它们是一个私有类的实例:

class ctypes.**_FuncPtr**
C 可调用外部函数的基类。

外部函数的实例也是兼容 C 的数据类型；它们代表 C 函数指针。

此行为可通过对外部函数对象的特殊属性赋值来自定义。

restype
赋值为一个 ctypes 类型来指定外部函数的结果类型。使用 None 表示 void，即不返回任何结果的函数。

赋值为一个不为 ctypes 类型的可调用 Python 对象也是可以的，在此情况下函数应返回 C `int`，该可调用对象将附带此整数被调用，以允许进一步的处理或错误检测。这种用法已被弃用，为了更灵活的后续处理或错误检测请使用一个 ctypes 数据类型作为*restype* 并将*errcheck* 属性赋值为一个可调用对象。

argtypes
赋值为一个 ctypes 类型的元组来指定函数所接受的参数类型。使用 `stdcall` 调用规范的函数只能附带与此元组长度相同数量的参数进行调用；使用 C 调用规范的函数还可接受额外的未指明参数。

当外部函数被调用时，每个实际参数都会被传给*argtypes* 元组中条目的 `from_param()` 类方法，此方法允许将实际参数适配为此外部函数所接受的对象。例如，*argtypes* 元组中的*c_char_p* 条目将使用 ctypes 约定规则把作为参数传入的字符串转换为字节串对象。

新增：现在可以将不是 ctypes 类型的条目放入 argtypes，但每个条目都必须具有 `from_param()` 方法用于返回可作为参数的值（整数、字符串、ctypes 实例）。这样就允许定义可将自定义对象适配为函数形参的适配器。

errcheck
将一个 Python 函数或其他可调用对象赋值给此属性。该可调用对象将附带三个及以上的参数被调用。

callable(*result*, *func*, *arguments*)
result 是外部函数返回的结果，由 `restype` 属性指明。

func 是外部函数对象本身，这样就允许重新使用相同的可调用对象来对多个函数进行检查或后续处理。

arguments 是一个包含最初传递给函数调用的形参的元组，这样就允许对所用参数的行为进行特别处理。

此函数所返回的对象将会由外部函数调用返回，但它还可以在外部函数调用失败时检查结果并引发异常。

exception ctypes.**ArgumentError**
此异常会在外部函数无法对某个传入参数执行转换时被引发。

函数原型

外部函数也可通过实例化函数原型来创建。函数原型类似于 C 中的函数原型；它们在不定义具体实现的情况下描述了一个函数（返回类型、参数类型、调用约定）。工厂函数必须使用函数所需要的结果类型和参数类型来调用，并可被用作装饰器工厂函数，在此情况下可以通过 `@wrapper` 语法应用于函数。请参阅回调函数了解有关示例。

ctypes.**CFUNCTYPE**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)
返回的函数原型会创建使用标准 C 调用约定的函数。该函数在调用过程中将释放 GIL。如果 *use_errno* 设为真值，则在调用之前和之后系统*errno* 变量的 ctypes 私有副本会与真正的*errno* 值进行交换；*use_last_error* 会为 Windows 错误码执行同样的操作。

ctypes.**WINFUNCTYPE**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)
仅限 Windows only：返回的函数原型会创建使用 `stdcall` 调用约定的函数，但在 Windows CE 上*WINFUNCTYPE()* 则会与*CFUNCTYPE()* 相同。该函数在调用过程中将释放 GIL。*use_errno* 和 *use_last_error* 具有与前面相同的含义。

ctypes.**PYFUNCTYPE**(*restype*, **argtypes*)
返回的函数原型会创建使用 Python 调用约定的函数。该函数在调用过程中将 不会释放 GIL。

这些工厂函数所创建的函数原型可通过不同的方式来实例化，具体取决于调用中的类型与数量:

prototype(*address*)
在指定地址上返回一个外部函数，地址值必须为整数。

prototype(*callable*)
基于 Python *callable* 创建一个 C 可调用函数（回调函数）。

prototype(*func_spec*[, *paramflags*])
返回由一个共享库导出的外部函数。*func_spec* 必须为一个 2 元组 `(name_or_ordinal, library)`。第一项是字符串形式的所导出函数名称，或小整数形式的所导出函数序号。第二项是该共享库实例。

prototype(*vtbl_index*, *name*[, *paramflags*[, *iid*]])
返回将调用一个 COM 方法的外部函数。*vtbl_index* 虚拟函数表中的索引。*name* 是 COM 方法的名称。*iid* 是可选的指向接口标识符的指针，它被用于扩展的错误报告。

COM 方法使用特殊的调用约定：除了在 `argtypes` 元组中指定的形参，它们还要求一个指向 COM 接口的指针作为第一个参数。

可选的 *paramflags* 形参会创建相比上述特性具有更多功能的外部函数包装器。

paramflags 必须为一个与 `argtypes` 长度相同的元组。

此元组中的每一项都包含有关形参的更多信息，它必须为包含一个、两个或更多条目的元组。

第一项是包含形参指令旗标组合的整数。

1 指定函数的一个输入形参。

2 输出形参。外部函数会填入一个值。

4 默认为整数零值的输入形参。

可选的第二项是字符串形式的形参名称。如果指定此项，则可以使用该形参名称来调用外部函数。

可选的第三项是该形参的默认值。

这个例子演示了如何包装 Windows 的 `MessageBoxW` 函数以使其支持默认形参和已命名参数。相应 windows 头文件的 C 声明是这样的:

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

这是使用*ctypes* 的包装:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from ctypes
↪"), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

现在 `MessageBox` 外部函数可以通过以下方式来调用:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

第二个例子演示了输出形参。这个 win32 GetWindowRect 函数通过将指定窗口的维度拷贝至调用者必须提供的 RECT 结构体来提取这些值。这是相应的 C 声明:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

这是使用*ctypes*的包装:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

带有输出形参的函数如果输出形参存在单一值则会自动返回该值，或是当输出形参存在多个值时返回包含这些值的元组，因此当 GetWindowRect 被调用时现在将返回一个 RECT 实例。

输出形参可以与 errcheck 协议相结合以执行进一步的输出处理与错误检查。Win32 GetWindowRect API 函数返回一个 BOOL 来表示成功或失败，因此此函数可执行错误检查，并在 API 调用失败时引发异常:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

如果 errcheck 不加更改地返回它所接收的参数元组，则*ctypes*会继续对输出形参执行常规处理。如果你希望返回一个窗口坐标的元组而非 RECT 实例，你可以从函数中提取这些字段并返回它们，常规处理将不会再执行:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

工具函数

ctypes.**addressof**(*obj*)
: 以整数形式返回内存缓冲区地址。*obj* 必须为一个 ctypes 类型的实例。

ctypes.**alignment**(*obj_or_type*)
: 返回一个 ctypes 类型的对齐要求。*obj_or_type* 必须为一个 ctypes 类型或实例。

ctypes.**byref**(*obj*[, *offset*])
: 返回指向 *obj* 的轻量指针，该对象必须为一个 ctypes 类型的实例。*offset* 默认值为零，且必须为一个将被添加到内部指针值的整数。

  byref(obj, offset) 对应于这段 C 代码:

```
(((char *)&obj) + offset)
```

返回的对象只能被用作外部函数调用形参。它的行为类似于 `pointer(obj)`，但构造起来要快很多。

ctypes.**cast**(*obj*, *type*)
此函数类似于 C 的强制转换运算符。它返回一个 *type* 的新实例，该实例指向与 *obj* 相同的内存块。*type* 必须为指针类型，而 *obj* 必须为可以被作为指针来解读的对象。

ctypes.**create_string_buffer**(*init_or_size*, *size=None*)
此函数会创建一个可变的字符缓冲区。返回的对象是一个*c_char* 的 ctypes 数组。

init_or_size 必须是一个指明数组大小的整数，或者是一个将被用来初始化数组条目的字节串对象。

如果将一个字节串对象指定为第一个参数，则将使缓冲区大小比其长度多一项以便数组的最后一项为一个 NUL 终结符。可以传入一个整数作为第二个参数以允许在不使用字节串长度的情况下指定数组大小。

ctypes.**create_unicode_buffer**(*init_or_size*, *size=None*)
此函数会创建一个可变的 unicode 字符缓冲区。返回的对象是一个*c_wchar* 的 ctypes 数组。

init_or_size 必须是一个指明数组大小的整数，或者是一个将被用来初始化数组条目的字符串。

如果将一个字符串指定为第一个参数，则将使缓冲区大小比其长度多一项以便数组的最后一项为一个 NUL 终结符。可以传入一个整数作为第二个参数以允许在不使用字符串长度的情况下指定数组大小。

ctypes.**DllCanUnloadNow**()
仅限 Windows：此函数是一个允许使用 ctypes 实现进程内 COM 服务的钩子。它将由 _ctypes 扩展 dll 所导出的 DllCanUnloadNow 函数来调用。

ctypes.**DllGetClassObject**()
仅限 Windows：此函数是一个允许使用 ctypes 实现进程内 COM 服务的钩子。它将由 `_ctypes` 扩展 dll 所导出的 DllGetClassObject 函数来调用。

ctypes.util.**find_library**(*name*)
尝试寻找一个库并返回路径名称。*name* 是库名称并且不带任何前缀如 `lib` 以及后缀如 `.so`，`.dylib` 或版本号（形式与 posix 链接器选项 `-l` 所用的一致）。如果找不到库，则返回 `None`。

确切的功能取决于系统。

ctypes.util.**find_msvcrt**()
仅限 Windows：返回 Python 以及扩展模块所使用的 VC 运行时库的文件名。如果无法确定库名称，则返回 `None`。

如果你需要通过调用 `free(void *)` 来释放内存，例如某个扩展模块所分配的内存，重要的一点是你应当使用分配内存的库中的函数。

ctypes.**FormatError**([*code*])
仅限 Windows：返回错误码 *code* 的文本描述。如果未指定错误码，则会通过调用 the last error code is used by calling the Windows api 函数 GetLastError 来获得最新的错误码。

ctypes.**GetLastError**()
仅限 Windows：返回 Windows 在调用线程中设置的最新错误码。此函数会直接调用 Windows *GetLastError()* 函数，它并不返回错误码的 ctypes 私有副本。

ctypes.**get_errno**()
返回调用线程中系统*errno* 变量的 ctypes 私有副本的当前值。

ctypes.**get_last_error**()
仅限 Windows：返回调用线程中系统 `LastError` 变量的 ctypes 私有副本的当前值。

ctypes.**memmove**(*dst*, *src*, *count*)
与标准 C memmove 库函数相同：将 *count* 个字节从 *src* 拷贝到 *dst*。*dst* 和 *src* 必须为整数或可被转换为指针的 ctypes 实例。

ctypes.**memset**(*dst*, *c*, *count*)
与标准 C memset 库函数相同：将位于地址 *dst* 的内存块用 *count* 个字节的 *c* 值填充。*dst* 必须为指定地址的整数或 ctypes 实例。

ctypes.**POINTER**(*type*)
这个工厂函数创建并返回一个新的 ctypes 指针类型。指针类型会被缓存并在内部重用，因此重复调用此函数耗费不大。*type* 必须为 ctypes 类型。

ctypes.**pointer**(*obj*)
此函数会创建一个新的指向 *obj* 的指针实例。返回的对象类型为 `POINTER(type(obj))`。

注意：如果你只是想向外部函数调用传递一个对象指针，你应当使用更为快速的 `byref(obj)`。

ctypes.**resize**(*obj*, *size*)
此函数可改变 *obj* 的内部内存缓冲区大小，其参数必须为 ctypes 类型的实例。没有可能将缓冲区设为小于对象类型的本机大小值，该值由 `sizeof(type(obj))` 给出，但将缓冲区加大则是可能的。

ctypes.**set_errno**(*value*)
设置调用线程中系统*errno* 变量的 ctypes 私有副本的当前值为 *value* 并返回原来的值。

ctypes.**set_last_error**(*value*)
仅限 Windows：设置调用线程中系统 `LastError` 变量的 ctypes 私有副本的当前值为 *value* 并返回原来的值。

ctypes.**sizeof**(*obj_or_type*)
返回 ctypes 类型或实例的内存缓冲区以字节表示的大小。其功能与 C `sizeof` 运算符相同。

ctypes.**string_at**(*address*, *size=-1*)
此函数返回从内存地址 *address* 开始的以字节串表示的 C 字符串。如果指定了 size，则将其用作长度，否则将假定字符串以零值结尾。

ctypes.**WinError**(*code=None*, *descr=None*)
仅限 Windows：此函数可能是 ctypes 中名字起得最差的函数。它会创建一个 OSError 的实例。如果未指定 *code*，则会调用 `GetLastError` 来确定错误码。如果未指定 *descr*，则会调用*FormatError()* 来获取错误的文本描述。

在 3.3 版更改: 以前是会创建一个*WindowsError* 的实例。

ctypes.**wstring_at**(*address*, *size=-1*)
此函数返回从内存地址 *address* 开始的以字符串表示的宽字节字符串。如果指定了 *size*，则将其用作字符串中的字符数量，否则将假定字符串以零值结尾。

数据类型

class ctypes.**_CData**
这个非公有类是所有 ctypes 数据类型的共同基类。另外，所有 ctypes 类型的实例都包含一个存放 C 兼容数据的内存块；该内存块的地址可由*addressof()* 辅助函数返回。还有一个实例变量被公开为*_objects*；此变量包含其他在内存块包含指针的情况下需要保持存活的 Python 对象。

ctypes 数据类型的通用方法，它们都是类方法（严谨地说，它们是*metaclass* 的方法）：

from_buffer(*source*[, *offset*])
此方法返回一个共享 *source* 对象缓冲区的 ctypes 实例。*source* 对象必须支持可写缓冲区接口。可选的 *offset* 形参指定以字节表示的源缓冲区内偏移量；默认值为零。如果源缓冲区不够大则会引发*ValueError*。

from_buffer_copy(*source*[, *offset*])
此方法创建一个 ctypes 实例，从 *source* 对象缓冲区拷贝缓冲区，该对象必须是可读的。可选的 *offset* 形参指定以字节表示的源缓冲区内偏移量；默认值为零。如果源缓冲区不够大则会引发 *ValueError*。

from_address(*address*)
此方法会使用 *address* 所指定的内存返回一个 ctypes 类型的实例，该参数必须为一个整数。

from_param(*obj*)
此方法会将 *obj* 适配为一个 ctypes 类型。它调用时会在当该类型存在于外部函数的 `argtypes` 元组时传入外部函数调用所使用的实际对象；它必须返回一个可被用作函数调用参数的对象。

所有 ctypes 数据类型都带有这个类方法的默认实现，它通常会返回 *obj*，如果该对象是此类型的实例的话。某些类型也能接受其他对象。

in_dll(*library*, *name*)
此方法返回一个由共享库导出的 ctypes 类型。*name* 为导出数据的符号名称，*library* 为所加载的共享库。

ctypes 数据类型的通用实例变量:

_b_base_
有时 ctypes 数据实例并不拥有它们所包含的内存块，它们只是共享了某个基对象的部分内存块。*_b_base_* 只读成员是拥有内存块的根 ctypes 对象。

_b_needsfree_
这个只读变量在 ctypes 数据实例自身已分配了内存块时为真值，否则为假值。

_objects
这个成员或者为 `None`，或者为一个包含需要保持存活以使内存块的内存保持有效的 Python 对象的字典。这个对象只是出于调试目的而对外公开；绝对不要修改此字典的内容。

基础数据类型

class ctypes.**_SimpleCData**
这个非公有类是所有基本 ctypes 数据类型的基类。它在这里被提及是因为它包含基本 ctypes 数据类型共有的属性。*_SimpleCData* 是 *_CData* 的子类，因此继承了其方法和属性。非指针及不包含指针的 ctypes 数据类型现在将可以被封存。

实例拥有一个属性:

value
这个属性包含实例的实际值。对于整数和指针类型，它是一个整数，对于字符类型，它是一个单字符字符串对象或字符串，对于字符指针类型，它是一个 Python 字节串对象或字符串。

当从 ctypes 实例提取 `value` 属性时，通常每次会返回一个新的对象。*ctypes* 并 *没有*实现原始对象返回，它总是会构造一个新的对象。所有其他 ctypes 对象实例也同样如此。

基本数据类型当作为外部函数调用结果被返回或者作为结构字段成员或数组项被提取时，会透明地转换为原生 Python 类型。换句话说，如果某个外部函数具有 *c_char_p* 的 `restype`，你将总是得到一个 Python 字节串对象，而 *不是*一个 *c_char_p* 实例。

基本数据类型的子类并 *没有*继续此行为。因此，如果一个外部函数的 `restype` 是 *c_void_p* 的一个子类，你将从函数调用得到一个该子类的实例。当然，你可以通过访问 `value` 属性来获取指针的值。

这些是基本 ctypes 数据类型:

class ctypes.**c_byte**
代表 C `signed char` 数据类型，并将值解读为一个小整数。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_char**
代表 C char 数据类型，并将值解读为单个字符。该构造器接受一个可选的字符串初始化器，字符串的长度必须恰好为一个字符。

class ctypes.**c_char_p**
当指向一个以零为结束符的字符串时代表 C char * 数据类型。对于通用字符指针来说也可能指向二进制数据，必须要使用 POINTER(c_char)。该构造器接受一个整数地址，或者一个字节串对象。

class ctypes.**c_double**
代表 C double 数据类型。该构造器接受一个可选的浮点数初始化器。

class ctypes.**c_longdouble**
代表 C long double 数据类型。该构造器接受一个可选的浮点数初始化器。在 sizeof(long double) == sizeof(double) 的平台上它是*c_double* 的一个别名。

class ctypes.**c_float**
代表 C float 数据类型。该构造器接受一个可选的浮点数初始化器。

class ctypes.**c_int**
代表 C signed int 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。在 sizeof(int) == sizeof(long) 的平台上它是*c_long* 的一个别名。

class ctypes.**c_int8**
代表 C 8 位 signed int 数据类型。通常是*c_byte* 的一个别名。

class ctypes.**c_int16**
代表 C 16 位 signed int 数据类型。通常是*c_short* 的一个别名。

class ctypes.**c_int32**
代表 C 32 位 signed int 数据类型。通常是*c_int* 的一个别名。

class ctypes.**c_int64**
代表 C 64 位 signed int 数据类型。通常是*c_longlong* 的一个别名。

class ctypes.**c_long**
代表 C signed long 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_longlong**
代表 C signed long long 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_short**
代表 C signed short 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_size_t**
代表 C size_t 数据类型。

class ctypes.**c_ssize_t**
代表 C ssize_t 数据类型。

3.2 新版功能.

class ctypes.**c_ubyte**
代表 C unsigned char 数据类型，它将值解读为一个小整数。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_uint**
代表 C unsigned int 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。在 sizeof(int) == sizeof(long) 的平台上它是*c_ulong* 的一个别名。

class ctypes.**c_uint8**
代表 C 8 位 unsigned int 数据类型。通常是*c_ubyte* 的一个别名。

class ctypes.**c_uint16**
代表 C 16 位 unsigned int 数据类型。通常是*c_ushort* 的一个别名。

class ctypes.**c_uint32**
代表 C 32 位 unsigned int 数据类型。通常是*c_uint* 的一个别名。.

class ctypes.**c_uint64**
代表 C 64 位 unsigned int 数据类型。通常是*c_ulonglong* 的一个别名。

class ctypes.**c_ulong**
代表 C unsigned long 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_ulonglong**
代表 C unsigned long long 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_ushort**
代表 C unsigned short 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_void_p**
代表 C void * 类型。该值被表示为整数形式。该构造器接受一个可选的整数初始化器。

class ctypes.**c_wchar**
代表 C wchar_t 数据类型，并将值解读为一单个字符的 unicode 字符串。该构造器接受一个可选的字符串初始化器，字符串的长度必须恰好为一个字符。

class ctypes.**c_wchar_p**
代表 C wchar_t * 数据类型，它必须为指向以零为结束符的宽字符串的指针。该构造器接受一个整数地址或者一个字符串。

class ctypes.**c_bool**
代表 C bool 数据类型 (更准确地说是 C99 _Bool)。它的值可以为 True 或 False，并且该构造器接受任何具有逻辑值的对象。

class ctypes.**HRESULT**
Windows 专属：代表一个 HRESULT 值，它包含某个函数或方法调用的成功或错误信息。

class ctypes.**py_object**
代表 C PyObject * 数据类型。不带参数地调用此构造器将创建一个 NULL PyObject * 指针。

ctypes.wintypes 模块提供了其他许多 Windows 专属的数据类型，例如 HWND, WPARAM 或 DWORD。还定义了一些有用的结构体例如 MSG 或 RECT。

结构化数据类型

class ctypes.**Union**(**args*, ***kw*)
本机字节序的联合所对应的抽象基类。

class ctypes.**BigEndianStructure**(**args*, ***kw*)
大端字节序的结构体所对应的抽象基类。

class ctypes.**LittleEndianStructure**(**args*, ***kw*)
小端字节序的结构体所对应的抽象基类。

非本机字节序的结构体不能包含指针类型字段，或任何其他包含指针类型字段的数据类型。

class ctypes.**Structure**(**args*, ***kw*)
本机字节序的结构体所对应的抽象基类。

实际的结构体和联合类型必须通过子类化这些类型之一来创建，并且至少要定义一个*_fields_* 类变量。*ctypes* 将创建*descriptor*，它允许通过直接属性访问来读取和写入字段。这些是

fields

一个定义结构体字段的序列。其中的条目必须为 2 元组或 3 元组。元组的第一项是字段名称，第二项指明字段类型；它可以是任何 ctypes 数据类型。

对于整数类型字段例如*c_int*，可以给定第三个可选项。它必须是一个定义字段比特位宽度的小正整数。

字段名称在一个结构体或联合中必须唯一。不会检查这个唯一性，但当名称出现重复时将只有一个字段可被访问。

可以在定义 Structure 子类的类语句 *之后* 再定义*_fields_* 类变量，这将允许创建直接或间接引用其自身的数据类型:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

但是，*_fields_* 类变量必须在类型第一次被使用（创建实例，调用*sizeof()* 等等）之前进行定义。在此之后对*_fields_* 类变量赋值将会引发 AttributeError。

It is possible to defined sub-subclasses of structure types, they inherit the fields of the base class plus the *_fields_* defined in the sub-subclass, if any.

pack

一个可选的小整数，它允许覆盖实体中结构体字段的对齐方式。当*_fields_* 被赋值时必须已经定义了*_pack_*，否则它将没有效果。

anonymous

一个可选的序列，它会列出未命名（匿名）字段的名称。当*_fields_* 被赋值时必须已经定义了*_anonymous_*，否则它将没有效果。

在此变量中列出的字段必须为结构体或联合类型字段。*ctypes* 将在结构体类型中创建描述器以允许直接访问嵌套字段，而无需创建结构体或联合字段。

以下是一个示例类型（Windows）：

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

TYPEDESC 结构体描述了一个 COM 数据类型，vt 字段指明哪个联合字段是有效的。由于 u 字段被定义为匿名字段，现在可以直接从 TYPEDESC 实例访问成员。td.lptdesc 和 td.u.lptdesc 是等价的，但前者速度更快，因为它不需要创建临时的联合实例:

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

It is possible to defined sub-subclasses of structures, they inherit the fields of the base class. If the subclass definition has a separate *_fields_* variable, the fields specified in this are appended to the fields of the base class.

结构体和联合的构造器均可接受位置和关键字参数。位置参数用于按照*_fields_* 中的出现顺序来初始化成员字段。构造器中的关键字参数会被解读为属性赋值，因此它们将以相应的名称来初始化*_fields_*，或为不存在于*_fields_* 中的名称创建新的属性。

数组与指针

class ctypes.**Array**(**args*)
数组的抽象基类。

创建实际数组类型的推荐方式是通过将任意*ctypes* 类型与一个正整数相乘。作为替代方式，你也可以子类化这个类型并定义*_length_* 和*_type_* 类变量。数组元素可使用标准的抽取和切片方式来读写；对于切片读取，结果对象本身 并非一个*Array*。

length
一个指明数组中元素数量的正整数。超出范围的抽取会导致*IndexError*。该值将由*len()* 返回。

type
指明数组中每个元素的类型。

Array 子类构造器可接受位置参数，用来按顺序初始化元素。

class ctypes.**_Pointer**
私有对象，指针的抽象基类。

实际的指针类型是通过调用*POINTER()* 并附带其将指向的类型来创建的；这会由*pointer()* 自动完成。

如果一个指针指向的是数组，则其元素可使用标准的抽取和切片方式来读写。指针对象没有长度，因此*len()* 将引发*TypeError*。抽取负值将会从指针 之前的内存中读取（与 C 一样），并且超出范围的抽取将可能因非法访问而导致崩溃（视你的运气而定）。

type
指明所指向的类型。

contents
返回指针所指向的对象。对此属性赋值会使指针改为指向所赋值的对象。

CHAPTER 17

并发执行

本章中描述的模块支持并发执行代码。适当的工具选择取决于要执行的任务（CPU 密集型或 IO 密集型）和偏好的开发风格（事件驱动的协作式多任务或抢占式多任务处理）。这是一个概述：

17.1 threading —基于线程的并行

源代码: Lib/threading.py

这个模块在较低级的模块*_thread* 基础上建立较高级的线程接口。参见： *queue* 模块。

The *dummy_threading* module is provided for situations where *threading* cannot be used because *_thread* is missing.

注解: 虽然他们没有在下面列出，这个模块仍然支持 Python 2.x 系列的这个模块下以 `camelCase` （驼峰法）命名的方法和函数。

这个模块定义了以下函数：

`threading.`**`active_count`**`()`
返回当前存活的*Thread* 对象的数量。返回值与*enumerate()* 所返回的列表长度一致。

`threading.`**`current_thread`**`()`
返回当前对应调用者的控制线程的*Thread* 对象。如果调用者的控制线程不是利用*threading* 创建，会返回一个功能受限的虚拟线程对象。

`threading.`**`get_ident`**`()`
返回当前线程的 “线程标识符”。它是一个非零的整数。它的值没有直接含义，主要是用作 magic cookie，比如作为含有线程相关数据的字典的索引。线程标识符可能会在线程退出，新线程创建时被复用。

3.3 新版功能.

threading.**enumerate**()
以列表形式返回当前所有存活的*Thread*对象。该列表包含守护线程，*current_thread()* 创建的虚拟线程对象和主线程。它不包含已终结的线程和尚未开始的线程。

threading.**main_thread**()
返回主*Thread*对象。一般情况下，主线程是 Python 解释器开始时创建的线程。

3.4 新版功能.

threading.**settrace**(*func*)
为所有*threading*模块开始的线程设置追踪函数。在每个线程的*run()*方法被调用前，*func* 会被传递给*sys.settrace()*。

threading.**setprofile**(*func*)
为所有*threading*模块开始的线程设置性能测试函数。在每个线程的*run()*方法被调用前，*func* 会被传递给*sys.setprofile()*。

threading.**stack_size**([*size*])
Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a *RuntimeError* is raised. If the specified stack size is invalid, a *ValueError* is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads.

这个模块同时定义了以下常量：

threading.**TIMEOUT_MAX**
阻塞函数（*Lock.acquire()*, *RLock.acquire()*, *Condition.wait()*, …）中形参 *timeout* 允许的最大值。传入超过这个值的 timeout 会抛出*OverflowError*异常。

3.2 新版功能.

这个模块定义了许多类，详见以下部分。

该模块的设计基于 Java 的线程模型。但是，在 Java 里面，锁和条件变量是每个对象的基础特性，而在 Python 里面，这些被独立成了单独的对象。Python 的*Thread*类只是 Java 的 Thread 类的一个子集；目前还没有优先级，没有线程组，线程还不能被销毁、停止、暂停、恢复或中断。Java 的 Thread 类的静态方法在实现时会映射为模块级函数。

下述方法的执行都是原子性的。

17.1.1 线程本地数据

线程本地数据是特定线程的数据。管理线程本地数据，只需要创建一个*local*（或者一个子类型）的实例并在实例中储存属性：

```
mydata = threading.local()
mydata.x = 1
```

在不同的线程中，实例的值会不同。

class threading.**local**
一个代表线程本地数据的类。

更多相关细节和大量示例，参见 _threading_local 模块的文档。

17.1.2 线程对象

The *Thread* class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the *run()* method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, *only* override the `__init__()` and *run()* methods of this class.

当线程对象一但被创建，其活动一定会因调用线程的*start()*方法开始。这会在独立的控制线程调用*run()*方法。

一旦线程活动开始，该线程会被认为是‘存活的’。当它的*run()*方法终结了（不管是正常的还是抛出未被处理的异常），就不是’存活的’。*is_alive()*方法用于检查线程是否存活。

其他线程可以调用一个线程的*join()*方法。这会阻塞调用该方法的线程，直到被调用*join()*方法的线程终结。

线程有名字。名字可以传递给构造函数，也可以通过*name*属性读取或者修改。

一个线程可以被标记成一个“守护线程”。这个标志的意义是，当剩下的线程都是守护线程时，整个 Python 程序将会退出。初始值继承于创建线程。这个标志可以通过*daemon*特征属性或者 *daemon* 构造器参数来设置。

注解： 守护线程在程序关闭时会突然关闭。他们的资源（例如已经打开的文档，数据库事务等等）可能没有被正确释放。如果你想你的线程正常停止，设置他们成为非守护模式并且使用合适的信号机制，例如：*Event*。

有个“主线程”对象；这对应 Python 程序里面初始的控制线程。它不是一个守护线程。

“虚拟线程对象”是可以被创建的。这些是对应于“外部线程”的线程对象，它们是在线程模块外部启动的控制线程，例如直接来自 C 代码。虚拟线程对象功能受限；他们总是被认为是存活的和守护模式，不能被*join()*。因为无法检测外来线程的终结，它们永远不会被删除。

class threading.**Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)

调用这个构造函数时，必需带有关键字参数。参数如下：

group 应该为 None；为了日后扩展 ThreadGroup 类实现而保留。

target 是用于*run()*方法调用的可调用对象。默认是 None，表示不需要调用任何方法。

name 是线程名称。默认情况下，由“Thread-*N*”格式构成一个唯一的名称，其中 *N* 是小的十进制数。

args 是用于调用目标函数的参数元组。默认是 ()。

kwargs 是用于调用目标函数的关键字参数字典。默认是 {}。

如果不是 None，*daemon* 参数将显式地设置该线程是否为守护模式。如果是 None (默认值)，线程将继承当前线程的守护模式属性。

如果子类型重载了构造函数，它一定要确保在做任何事前，先发起调用基类构造器 (Thread.__init__())。

在 3.3 版更改: 加入 *daemon* 参数。

start()

开始线程活动。

它在一个线程里最多只能被调用一次。它安排对象的*run()*方法在一个独立的控制进程中调用。

如果同一个线程对象中调用这个方法的次数大于一次，会抛出*RuntimeError*。

run()
代表线程活动的方法。

You may override this method in a subclass. The standard *run()* method invokes the callable object passed to the object's constructor as the *target* argument, if any, with sequential and keyword arguments taken from the *args* and *kwargs* arguments, respectively.

join(*timeout=None*)
等待，直到线程终结。这会阻塞调用这个方法的线程，直到被调用*join()* 的线程终结–不管是正常终结还是抛出未处理异常–或者直到发生超时，超时选项是可选的。

当 *timeout* 参数存在而且不是 None 时，它应该是一个用于指定操作超时的以秒为单位的浮点数（或者分数）。因为*join()* 总是返回 None ，所以你一定要在*join()* 后调用*is_alive()* 才能判断是否发生超时–如果线程仍然存活，则*join()* 超时。

当 *timeout* 参数不存在或者是 None ，这个操作会阻塞直到线程终结。

一个线程可以被*join()* 很多次。

如果尝试加入当前线程会导致死锁，*join()* 会引起*RuntimeError* 异常。如果尝试*join()* 一个尚未开始的线程，也会抛出相同的异常。

name
只用于识别的字符串。它没有语义。多个线程可以赋予相同的名称。初始名称由构造函数设置。

getName()
setName()
旧的*name* 取值/设值 API；请改为直接以特征属性方式使用它。

ident
这个线程的‘线程标识符’，如果线程尚未开始则为 None 。这是个非零整数。参见*get_ident()* 函数。当一个线程退出而另外一个线程被创建，线程标识符会被复用。即使线程退出后，仍可得到标识符。

is_alive()
返回线程是否存活。

当*run()* 方法刚开始直到*run()* 方法刚结束，这个方法返回 True 。模块函数*enumerate()* 返回包含所有存活线程的列表。

daemon
一个表示这个线程是（True）否（False）守护线程的布尔值。一定要在调用*start()* 前设置好，不然会抛出*RuntimeError* 。初始值继承于创建线程；主线程不是守护线程，因此主线程创建的所有线程默认都是*daemon* = False。

当没有存活的非守护线程时，整个 Python 程序才会退出。

isDaemon()
setDaemon()
旧的*daemon* 取值/设值 API；请改为直接以特性属性方式使用它。

CPython implementation detail: CPython 下，因为*Global Interpreter Lock*，一个时刻只有一个线程可以执行 Python 代码（尽管如此，某些性能导向的库可能会克服这个限制）。如果你想让你的应用更好的利用多核计算机的计算性能，推荐你使用*multiprocessing* 或者*concurrent.futures.ProcessPoolExecutor* 。但是如果你想同时运行多个 I/O 绑定任务，线程仍然是一个合适的模型。

17.1.3 锁对象

原始锁是一个在锁定时不属于特定线程的同步基元组件。在 Python 中，它是能用的最低级的同步基元组件，由*_thread* 扩展模块直接实现。

原始锁处于“锁定”或者“非锁定”两种状态之一。它被创建时为非锁定状态。它有两个基本方法，*acquire()* 和*release()* 。当状态为非锁定时，*acquire()* 将状态改为锁定并立即返回。当状态是锁定时，*acquire()* 将阻塞至其他线程调用*release()* 将其改为非锁定状态，然后*acquire()* 调用重置其为锁定状态并返回。*release()* 只在锁定状态下调用；它将状态改为非锁定并立即返回。如果尝试释放一个非锁定的锁，则会引发*RuntimeError* 异常。

锁同样支持上下文管理协议。

当多个线程在*acquire()* 等待状态转变为未锁定被阻塞，然后*release()* 重置状态为未锁定时，只有一个线程能继续执行；至于哪个等待线程继续执行没有定义，并且会根据实现而不同。

所有方法的执行都是原子性的。

class threading.**Lock**

实现原始锁对象的类。一旦一个线程获得一个锁，会阻塞随后尝试获得锁的线程，直到它被释放；任何线程都可以释放它。

需要注意的是 Lock 其实是一个工厂函数，返回平台支持的具体锁类中最有效的版本的实例。

acquire(*blocking=True*, *timeout=-1*)

可以阻塞或非阻塞地获得锁。

当调用时参数 *blocking* 设置为 True （缺省值），阻塞直到锁被释放，然后将锁锁定并返回 True 。

在参数 *blocking* 被设置为 False 的情况下调用，将不会发生阻塞。如果调用时 *blocking* 设为 True 会阻塞，并立即返回 False ； 否则，将锁锁定并返回 True。

当浮点型 *timeout* 参数被设置为正值调用时，只要无法获得锁，将最多阻塞 *timeout* 设定的秒数。*timeout* 参数被设置为 -1 时将无限等待。当 *blocking* 为 false 时，*timeout* 指定的值将被忽略。

如果成功获得锁，则返回 True，否则返回 False (例如发生 超时的时候)。

在 3.2 版更改: 新的 *timeout* 形参。

在 3.2 版更改: 现在如果底层线程实现支持，则可以通过 POSIX 上的信号中断锁的获取。

release()

释放一个锁。这个方法可以在任何线程中调用，不单指获得锁的线程。

当锁被锁定，将它重置为未锁定，并返回。如果其他线程正在等待这个锁解锁而被阻塞，只允许其中一个允许。

在未锁定的锁调用时，会引发*RuntimeError* 异常。

没有返回值。

17.1.4 递归锁对象

重入锁是一个可以被同一个线程多次获取的同步基元组件。在内部，它在基元锁的锁定/非锁定状态上附加了“所属线程”和“递归等级”的概念。在锁定状态下，某些线程拥有锁；在非锁定状态下，没有线程拥有它。

若要锁定锁，线程调用其*acquire()* 方法；一旦线程拥有了锁，方法将返回。若要解锁，线程调用*release()* 方法。*acquire()*/*release()* 对可以嵌套；只有最终*release()* (最外面一对的*release()*) 将锁解开，才能让其他线程继续处理*acquire()* 阻塞。

递归锁也支持上下文管理协议。

class threading.**RLock**

此类实现了重入锁对象。重入锁必须由获取它的线程释放。一旦线程获得了重入锁，同一个线程再次获取它将不阻塞；线程必须在每次获取它时释放一次。

需要注意的是 RLock 其实是一个工厂函数，返回平台支持的具体递归锁类中最有效的版本的实例。

acquire(*blocking=True*, *timeout=-1*)

可以阻塞或非阻塞地获得锁。

当无参数调用时：如果这个线程已经拥有锁，递归级别增加一，并立即返回。否则，如果其他线程拥有该锁，则阻塞至该锁解锁。一旦锁被解锁 (不属于任何线程)，则抢夺所有权，设置递归等级为一，并返回。如果多个线程被阻塞，等待锁被解锁，一次只有一个线程能抢到锁的所有权。在这种情况下，没有返回值。

When invoked with the *blocking* argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the *blocking* argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with the floating-point *timeout* argument set to a positive value, block for at most the number of seconds specified by *timeout* and as long as the lock cannot be acquired. Return true if the lock has been acquired, false if the timeout has elapsed.

在 3.2 版更改: 新的 *timeout* 形参。

release()

释放锁，自减递归等级。如果减到零，则将锁重置为非锁定状态 (不被任何线程拥有)，并且，如果其他线程正被阻塞着等待锁被解锁，则仅允许其中一个线程继续。如果自减后，递归等级仍然不是零，则锁保持锁定，仍由调用线程拥有。

只有当前线程拥有锁才能调用这个方法。如果锁被释放后调用这个方法，会引起*RuntimeError*异常。

没有返回值。

17.1.5 条件对象

条件变量总是与某种类型的锁对象相关联，锁对象可以通过传入获得，或者在缺省的情况下自动创建。当多个条件变量需要共享同一个锁时，传入一个锁很有用。锁是条件对象的一部分，你不必单独地跟踪它。

条件变量服从上下文管理协议：使用 with 语句会在它包围的代码块内获取关联的锁。*acquire()* 和*release()* 方法也能调用关联锁的相关方法。

其它方法必须在持有关联的锁的情况下调用。*wait()* 方法释放锁，然后阻塞直到其它线程调用*notify()* 方法或*notify_all()* 方法唤醒它。一旦被唤醒，*wait()* 方法重新获取锁并返回。它也可以指定超时时间。

The *notify()* method wakes up one of the threads waiting for the condition variable, if any are waiting. The *notify_all()* method wakes up all threads waiting for the condition variable.

注意：*notify()* 方法和*notify_all()* 方法并不会释放锁，这意味着被唤醒的线程不会立即从它们的*wait()* 方法调用中返回，而是会在调用了*notify()* 方法或*notify_all()* 方法的线程最终放弃了锁的所有权后返回。

使用条件变量的典型编程风格是将锁用于同步某些共享状态的权限，那些对状态的某些特定改变感兴趣的线程，它们重复调用*wait()* 方法，直到看到所期望的改变发生；而对于修改状态的线程，它们将当前状态改变为可能是等待者所期待的新状态后，调用*notify()* 方法或者*notify_all()* 方法。例如，下面的代码是一个通用的无限缓冲区容量的生产者-消费者情形：

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

使用 while 循环检查所要求的条件成立与否是有必要的，因为*wait()* 方法可能要经过不确定长度的时间后才会返回，而此时导致*notify()* 方法调用的那个条件可能已经不再成立。这是多线程编程所固有的问题。*wait_for()* 方法可自动化条件检查，并简化超时计算。

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

选择*notify()* 还是*notify_all()* ，取决于一次状态改变是只能被一个还是能被多个等待线程所用。例如在一个典型的生产者-消费者情形中，添加一个项目到缓冲区只需唤醒一个消费者线程。

class threading.**Condition**(*lock=None*)

实现条件变量对象的类。一个条件变量对象允许一个或多个线程在被其它线程所通知之前进行等待。

如果给出了非 None 的 *lock* 参数，则它必须为*Lock* 或者*RLock* 对象，并且它将被用作底层锁。否则，将会创建新的*RLock* 对象，并将其用作底层锁。

在 3.3 版更改: 从工厂函数变为类。

acquire(**args*)

请求底层锁。此方法调用底层锁的相应方法，返回值是底层锁相应方法的返回值。

release()

释放底层锁。此方法调用底层锁的相应方法。没有返回值。

wait(*timeout=None*)

等待直到被通知或发生超时。如果线程在调用此方法时没有获得锁，将会引发*RuntimeError* 异常。

这个方法释放底层锁，然后阻塞，直到在另外一个线程中调用同一个条件变量的*notify()* 或*notify_all()* 唤醒它，或者直到可选的超时发生。一旦被唤醒或者超时，它重新获得锁并返回。

当提供了 *timeout* 参数且不是 None 时，它应该是一个浮点数，代表操作的超时时间，以秒为单位（可以为小数）。

当底层锁是个*RLock* ，不会使用它的*release()* 方法释放锁，因为当它被递归多次获取时，实际上可能无法解锁。相反，使用了*RLock* 类的内部接口，即使多次递归获取它也能解锁它。然后，在重新获取锁时，使用另一个内部接口来恢复递归级别。

返回 True ，除非提供的 *timeout* 过期，这种情况下返回 False。

在 3.2 版更改: 很明显，方法总是返回 None。

wait_for(*predicate*, *timeout=None*)

等待，直到条件计算为真。*predicate* 应该是一个可调用对象而且它的返回值可被解释为一个布尔值。可以提供 *timeout* 参数给出最大等待时间。

这个实用方法会重复地调用*wait()* 直到满足判断式或者发生超时。返回值是判断式最后一个返回值，而且如果方法发生超时会返回 False 。

忽略超时功能，调用此方法大致相当于编写:

```
while not predicate():
    cv.wait()
```

因此，规则同样适用于*wait()* ： 锁必须在被调用时保持获取，并在返回时重新获取。随着锁定执行判断式。

3.2 新版功能.

notify(*n=1*)

默认唤醒一个等待这个条件的线程。如果调用线程在没有获得锁的情况下调用这个方法，会引发*RuntimeError* 异常。

这个方法唤醒最多 *n* 个正在等待这个条件变量的线程；如果没有线程在等待，这是一个空操作。

当前实现中，如果至少有 *n* 个线程正在等待，准确唤醒 *n* 个线程。但是依赖这个行为并不安全。未来，优化的实现有时会唤醒超过 *n* 个线程。

注意：被唤醒的线程实际上不会返回它调用的*wait()* ，直到它可以重新获得锁。因为*notify()* 不会释放锁，只有它的调用者应该这样做。

notify_all()

唤醒所有正在等待这个条件的线程。这个方法行为与*notify()* 相似，但并不只唤醒单一线程，而是唤醒所有等待线程。如果调用线程在调用这个方法时没有获得锁，会引发*RuntimeError* 异常。

17.1.6 信号量对象

这是计算机科学史上最古老的同步原语之一，早期的荷兰科学家 Edsger W. Dijkstra 发明了它。（他使用名称 P() 和 V() 而不是*acquire()* 和*release()* ）。

一个信号量管理一个内部计数器，该计数器因*acquire()* 方法的调用而递减，因*release()* 方法的调用而递增。计数器的值永远不会小于零；当*acquire()* 方法发现计数器为零时，将会阻塞，直到其它线程调用*release()* 方法。

信号量对象也支持上下文管理协议 。

class threading.**Semaphore**(*value=1*)

该类实现信号量对象。信号量对象管理一个原子性的计数器，代表*release()* 方法的调用次数减去*acquire()* 的调用次数再加上一个初始值。如果需要，*acquire()* 方法将会阻塞直到可以返回而不会使得计数器变成负数。在没有显式给出 *value* 的值时，默认为 1。

可选参数 *value* 赋予内部计数器初始值，默认值为 1 。如果 *value* 被赋予小于 0 的值，将会引发*ValueError* 异常。

在 3.3 版更改: 从工厂函数变为类。

acquire(*blocking=True*, *timeout=None*)

获取一个信号量。

在不带参数的情况下调用时:

- If the internal counter is larger than zero on entry, decrement it by one and return true immediately.
- If the internal counter is zero on entry, block until awoken by a call to *release()*. Once awoken (and the counter is greater than 0), decrement the counter by 1 and return true. Exactly one thread will be awoken by each call to *release()*. The order in which threads are awoken should not be relied on.

When invoked with *blocking* set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with a *timeout* other than `None`, it will block for at most *timeout* seconds. If acquire does not complete successfully in that interval, return false. Return true otherwise.

在 3.2 版更改: 新的 *timeout* 形参。

release()
释放一个信号量，将内部计数器的值增加 1。当计数器原先的值为 0 且有其它线程正在等待它再次大于 0 时，唤醒正在等待的线程。

class threading.**BoundedSemaphore**(*value=1*)
该类实现有界信号量。有界信号量通过检查以确保它当前的值不会超过初始值。如果超过了初始值，将会引发*ValueError* 异常。在大多情况下，信号量用于保护数量有限的资源。如果信号量被释放的次数过多，则表明出现了错误。没有指定时，*value* 的值默认为 1。

在 3.3 版更改: 从工厂函数变为类。

Semaphore 例子

信号量通常用于保护数量有限的资源，例如数据库服务器。在资源数量固定的任何情况下，都应该使用有界信号量。在生成任何工作线程前，应该在主线程中初始化信号量。

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

工作线程生成后，当需要连接服务器时，这些线程将调用信号量的 acquire 和 release 方法：

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

使用有界信号量能减少这种编程错误：信号量的释放次数多于其请求次数。

17.1.7 事件对象

这是线程之间通信的最简单机制之一：一个线程发出事件信号，而其他线程等待该信号。

一个事件对象管理一个内部标志，调用*set()* 方法可将其设置为 true，调用*clear()* 方法可将其设置为 false，调用*wait()* 方法将进入阻塞直到标志为 true。

class threading.**Event**
实现事件对象的类。事件对象管理一个内部标志，调用*set()* 方法可将其设置为 true。调用*clear()* 方法可将其设置为 false。调用*wait()* 方法将进入阻塞直到标志为 true。这个标志初始时为 false。

在 3.3 版更改: 从工厂函数变为类。

is_set()
Return true if and only if the internal flag is true.

set()
将内部标志设置为 true。所有正在等待这个事件的线程将被唤醒。当标志为 true 时，调用*wait()* 方法的线程不会被被阻塞。

clear()
将内部标志设置为 false。之后调用*wait()* 方法的线程将会被阻塞，直到调用*set()* 方法将内部标志再次设置为 true。

wait(*timeout=None*)
阻塞线程直到内部变量为 true。如果调用时内部标志为 true，将立即返回。否则将阻塞线程，直到调用*set()* 方法将标志设置为 true 或者发生可选的超时。

当提供了 timeout 参数且不是 None 时，它应该是一个浮点数，代表操作的超时时间，以秒为单位（可以为小数）。

This method returns true if and only if the internal flag has been set to true, either before the wait call or after the wait starts, so it will always return True except if a timeout is given and the operation times out.

在 3.1 版更改: 很明显，方法总是返回 None。

17.1.8 定时器对象

此类表示一个操作应该在等待一定的时间之后运行—相当于一个定时器。*Timer* 类是*Thread* 类的子类，因此可以像一个自定义线程一样工作。

与线程一样，通过调用 start() 方法启动定时器。而*cancel()* 方法可以停止计时器（在计时结束前），定时器在执行其操作之前等待的时间间隔可能与用户指定的时间间隔不完全相同。

例如

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.**Timer**(*interval*, *function*, *args=None*, *kwargs=None*)
创建一个定时器，在经过 *interval* 秒的间隔事件后，将会用参数 *args* 和关键字参数 *kwargs* 调用 *function*。如果 *args* 为 None （默认值），则会使用一个空列表。如果 *kwargs* 为 None （默认值），则会使用一个空字典。

在 3.3 版更改: 从工厂函数变为类。

cancel()
停止定时器并取消执行计时器将要执行的操作。仅当计时器仍处于等待状态时有效。

17.1.9 栅栏对象

3.2 新版功能.

栅栏类提供一个简单的同步原语，用于应对固定数量的线程需要彼此相互等待的情况。线程调用*wait()* 方法后将阻塞，直到所有线程都调用了*wait()* 方法。此时所有线程将被同时释放。

栅栏对象可以被多次使用，但进程的数量不能改变。

这是一个使用简便的方法实现客户端进程与服务端进程同步的例子：

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
```

(下页继续)

(续上页)

```
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.**Barrier**(*parties*, *action=None*, *timeout=None*)

创建一个需要 *parties* 个线程的栅栏对象。如果提供了可调用的 *action* 参数，它会在所有线程被释放时在其中一个线程中自动调用。*timeout* 是默认的超时时间，如果没有在*wait()* 方法中指定超时时间的话。

wait(*timeout=None*)

冲出栅栏。当栅栏中所有线程都已经调用了这个函数，它们将同时被释放。如果提供了 *timeout* 参数，这里的 *timeout* 参数优先于创建栅栏对象时提供的 *timeout* 参数。

函数返回值是一个整数，取值范围在 0 到 *parties* –1，在每个线程中的返回值不相同。可用于从所有线程中选择唯一的一个线程执行一些特别的工作。例如：

```
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

如果创建栅栏对象时在构造函数中提供了 *action* 参数，它将在其中一个线程释放前被调用。如果此调用引发了异常，栅栏对象将进入损坏态。

如果发生了超时，栅栏对象将进入破损态。

如果栅栏对象进入破损态，或重置栅栏时仍有线程等待释放，将会引发*BrokenBarrierError* 异常。

reset()

重置栅栏为默认的初始态。如果栅栏中仍有线程等待释放，这些线程将会收到*BrokenBarrierError* 异常。

注意使用此函数时，如果有某些线程状态未知，则可能需其它的同步来确保线程已被释放。如果栅栏进入了破损态，最好废弃它并新建一个栅栏。

abort()

使栅栏进入破损态。这将导致所有已经调用和未来调用的*wait()* 方法中引发*BrokenBarrierError* 异常。使用这个方法的一种情况是需要中止程序以避免死锁。

更好的方式是：创建栅栏时提供一个合理的超时时间，来自动避免某个线程出错。

parties

冲出栅栏所需要的线程数量。

n_waiting

当前时刻正在栅栏中阻塞的线程数量。

broken

一个布尔值，值为 `True` 表明栅栏为破损态。

exception threading.**BrokenBarrierError**

异常类，是*RuntimeError* 异常的子类，在*Barrier* 对象重置时仍有线程阻塞时和对象进入破损态时被引发。

17.1.10 Using locks, conditions, and semaphores in the `with` statement

这个模块提供的带有 `acquire()` 和 `release()` 方法的对象，可以被用作 `with` 语句的上下文管理器。当进入语句块时 `acquire()` 方法会被调用，退出语句块时 `release()` 会被调用。因此，以下片段:

```
with some_lock:
    # do something...
```

相当于:

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

现在*Lock* 、*RLock* 、*Condition* 、*Semaphore* 和*BoundedSemaphore* 对象可以用作 `with` 语句的上下文管理器。

17.2 `multiprocessing` —基于进程的并行

源代码 Lib/multiprocessing/

17.2.1 概述

multiprocessing 是一个用与*threading* 模块相似 API 的支持产生进程的包。*multiprocessing* 包同时提供本地和远程并发，使用子进程代替线程，有效避免*Global Interpreter Lock* 带来的影响。因此，*multiprocessing* 模块允许程序员充分利用机器上的多个核心。Unix 和 Windows 上都可以运行。

multiprocessing 模块还引入了在*threading* 模块中没有类似物的 API。这方面的一个主要例子是*Pool* 对象，它提供了一种方便的方法，可以跨多个输入值并行化函数的执行，跨进程分配输入数据（数据并行）。以下示例演示了在模块中定义此类函数的常见做法，以便子进程可以成功导入该模块。这个数据并行的基本例子使用*Pool* ，

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

将打印到标准输出

```
[1, 4, 9]
```

Process 类

在*multiprocessing* 中，通过创建一个*Process* 对象然后调用它的 start() 方法来生成进程。*Process* 和*threading.Thread* API 相同。一个简单的多进程程序示例是:

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

要显示所涉及的各个进程 ID，这是一个扩展示例:

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

为了解释为什么 if __name__ == '__main__' 部分是必需的，请参见编程指导。

上下文和启动方法

根据不同的平台，*multiprocessing* 支持三种启动进程的方法。这些 *启动方法*有

spawn 父进程启动一个新的 Python 解释器进程。子进程只会继承那些运行进程对象的*run()* 方法所需的资源。特别是父进程中非必须的文件描述符和句柄不会被继承。相对于使用 *fork* 或者 *forkserver*，使用这个方法启动进程相当慢。

可在 Unix 和 Windows 上使用。Windows 上的默认设置。

fork 父进程使用*os.fork()* 来产生 Python 解释器分叉。子进程在开始时实际上与父进程相同。父进程的所有资源都由子进程继承。请注意，安全分叉多线程进程是棘手的。

只存在于 Unix。Unix 中的默认值。

forkserver 程序启动并选择 * forkserver * 启动方法时，将启动服务器进程。从那时起，每当需要一个新进程时，父进程就会连接到服务器并请求它分叉一个新进程。分叉服务器进程是单线程的，因此使用*os.fork()* 是安全的。没有不必要的资源被继承。

可在 Unix 平台上使用，支持通过 Unix 管道传递文件描述符。

在 3.4 版更改: *spawn* 在所有 unix 平台上添加，并且为一些 unix 平台添加了 *forkserver* 。子进程不再继承 Windows 上的所有上级进程可继承的句柄。

在 Unix 上使用 *spawn* 或 *forkserver* 启动方法也将启动一个 *信号量跟踪器* 进程，该进程跟踪由程序进程创建的未链接的命名信号量。当所有进程退出时，信号量跟踪器取消链接任何剩余的信号量。通常不应该有，但如果一个进程被信号杀死，可能会有一些“泄露”的信号量。(取消链接命名的信号量是一个严重的问题，因为系统只允许有限的数量，并且在下次重新启动之前它们不会自动取消链接。)

要选择一个启动方法，你应该在主模块的 `if __name__ == '__main__'` 子句中调用*set_start_method()* 。例如:

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

在程序中*set_start_method()* 不应该被多次调用。

或者，你可以使用*get_context()* 来获取上下文对象。上下文对象与多处理模块具有相同的 API，并允许在同一程序中使用多个启动方法。:

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

请注意，对象在不同上下文创建的进程间可能并不兼容。特别是，使用 *fork* 上下文创建的锁不能传递给使用 *spawn* 或 *forkserver* 启动方法启动的进程。

想要使用特定启动方法的库应该使用*get_context()* 以避免干扰库用户的选择。

在进程之间交换对象

multiprocessing 支持进程之间的两种通信通道:

队列

Queue 类是一个近似*queue.Queue* 的克隆。例如:

```
from multiprocessing import Process, Queue

def f(q):
```

(下页继续)

(续上页)

```
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

队列是线程和进程安全的。

管道

Pipe() 函数返回一个由管道连接的连接对象，默认情况下是双工（双向）。例如:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

返回的两个连接对象*Pipe()* 表示管道的两端。每个连接对象都有 `send()` 和 `recv()` 方法（相互之间的）。请注意，如果两个进程（或线程）同时尝试读取或写入管道的 同一端，则管道中的数据可能会损坏。当然，同时使用管道的不同端的进程不存在损坏的风险。

进程之间的同步

multiprocessing 包含来自*threading* 的所有同步原语的等价物。例如，可以使用锁来确保一次只有一个进程打印到标准输出:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

不使用来自不同进程的锁输出容易产生混淆。

在进程之间共享状态

如上所述，在进行并发编程时，通常最好尽量避免使用共享状态。使用多个进程时尤其如此。

但是，如果你真的需要使用一些共享数据，那么*multiprocessing* 提供了两种方法。

共享内存

可以使用*Value* 或*Array* 将数据存储在共享内存映射中。例如，以下代码:

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

将打印

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

创建 num 和 arr 时使用的 'd' 和 'i' 参数是*array* 模块使用的类型的 typecode： 'd' 表示双精度浮点数，'i' 表示有符号整数。这些共享对象将是进程和线程安全的。

为了更灵活地使用共享内存，可以使用*multiprocessing.sharedctypes* 模块，该模块支持创建从共享内存分配的任意 ctypes 对象。

服务器进程

由 Manager() 返回的管理器对象控制一个服务器进程，该进程保存 Python 对象并允许其他进程使用代理操作它们。

Manager() 返回的管理器支持类型: *list* 、*dict* 、*Namespace* 、*Lock* 、*RLock* 、*Semaphore* 、*BoundedSemaphore* 、*Condition* 、*Event* 、*Barrier* 、*Queue* 、*Value* 和*Array* 。例如

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))
```

(下页继续)

(续上页)

```
        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```

将打印

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

服务器进程管理器比使用共享内存对象更灵活，因为它们可以支持任意对象类型。此外，单个管理器可以通过网络由不同计算机上的进程共享。但是，它们比使用共享内存慢。

使用工作进程

Pool 类表示一个工作进程池。它具有允许以几种不同方式将任务分配到工作进程的方法。

例如

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))      # runs in *only* one process
        print(res.get(timeout=1))             # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))             # prints the PID of that process

        # launching multiple evaluations asynchronously *may* use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 secs
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
```

(下页继续)

(续上页)

```
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")
```

请注意，池的方法只能由创建它的进程使用。

注解： 该软件包中的功能要求子项可以导入 __main__ 模块。这包含在编程指导 中，但值得指出。这意味着一些示例，例如*multiprocessing.pool.Pool* 示例在交互式解释器中不起作用。例如:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(如果你尝试这个，它实际上会以半随机的方式输出三个完整的回溯，然后你可能不得不以某种方式停止主进程。)

17.2.2 参考

multiprocessing 包大部分复制了*threading* 模块的 API。

Process 和异常

class multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)

进程对象表示在单独进程中运行的活动。*Process* 类等价于*threading.Thread* 。

应始终使用关键字参数调用构造函数。*group* 应该始终是 None ； 它仅用于兼容*threading.Thread* 。*target* 是由*run()* 方法调用的可调用对象。它默认为 None ，意味着什么都没有被调用。*name* 是进程名称（有关详细信息，请参阅*name* ）。*args* 是目标调用的参数元组。*kwargs* 是目标调用的关键字参数字典。如果提供，则键参数 *daemon* 将进程*daemon* 标志设置为 True 或 False 。如果是 None （默认值），则该标志将从创建的进程继承。

默认情况下，不会将任何参数传递给 *target* 。

如果子类重写构造函数，它必须确保它在对进程执行任何其他操作之前调用基类构造函数（Process.__init__() ）。

在 3.3 版更改: 加入 *daemon* 参数。

run()

表示进程活动的方法。

你可以在子类中重载此方法。标准*run()* 方法调用传递给对象构造函数的可调用对象作为目标参数（如果有），分别从 *args* 和 *kwargs* 参数中获取顺序和关键字参数。

start()
启动进程活动。

每个进程对象最多只能调用一次。它安排对象的*run()* 方法在一个单独的进程中调用。

join([*timeout*])
如果可选参数 *timeout* 是 None （默认值），则该方法将阻塞，直到调用*join()* 方法的进程终止。如果 *timeout* 是一个正数，它最多会阻塞 *timeout* 秒。请注意，如果进程终止或方法超时，则该方法返回 None 。检查进程的*exitcode* 以确定它是否终止。

一个进程可以合并多次。

进程无法并入自身，因为这会导致死锁。尝试在启动进程之前合并进程是错误的。

name
进程的名称。该名称是一个字符串，仅用于识别目的。它没有语义。可以为多个进程指定相同的名称。

初始名称由构造器设定。如果没有为构造器提供显式名称，则会构造一个形式为‘Process-N_1:N_2:…:N_k’的名称，其中每个 N_k 是其父亲的第 N 个孩子。

is_alive()
返回进程是否还活着。

粗略地说，从*start()* 方法返回到子进程终止之前，进程对象仍处于活动状态。

daemon
进程的守护标志，一个布尔值。这必须在*start()* 被调用之前设置。

初始值继承自创建进程。

当进程退出时，它会尝试终止其所有守护进程子进程。

请注意，不允许在守护进程中创建子进程。这是因为当守护进程由于父进程退出而中断时，其子进程会变成孤儿进程。另外，这些 **不是** Unix 守护进程或服务，它们是正常进程，如果非守护进程已经退出，它们将被终止（并且不被合并）。

除了*threading.Thread* API ，*Process* 对象还支持以下属性和方法：

pid
返回进程 ID。在生成该进程之前，这将是 None 。

exitcode
的退子进程出代码。如果进程尚未终止，这将是 None 。负值 *-N* 表示孩子被信号 *N* 终止。

authkey
进程的身份验证密钥（字节字符串）。

当*multiprocessing* 初始化时，主进程使用*os.urandom()* 分配一个随机字符串。

当创建*Process* 对象时，它将继承其父进程的身份验证密钥，尽管可以通过将*authkey* 设置为另一个字节字符串来更改。

参见*认证密码* 。

sentinel
系统对象的数字句柄，当进程结束时将变为“ready”。

如果要使用*multiprocessing.connection.wait()* 一次等待多个事件，可以使用此值。否则调用*join()* 更简单。

在 Windows 上，这是一个操作系统句柄，可以与 WaitForSingleObject 和 WaitForMultipleObjects 系列 API 调用一起使用。在 Unix 上，这是一个文件描述符，可以使用来自*select* 模块的原语。

3.3 新版功能.

terminate()
终止进程。在 Unix 上，这是使用 SIGTERM 信号完成的；在 Windows 上使用 TerminateProcess() 。请注意，不会执行退出处理程序和 finally 子句等。

请注意，进程的后代进程将不会被终止——它们将简单地变成孤立的。

> **警告：** 如果在关联进程使用管道或队列时使用此方法，则管道或队列可能会损坏，并可能无法被其他进程使用。类似地，如果进程已获得锁或信号量等，则终止它可能导致其他进程死锁。

注意*start()* 、*join()* 、*is_alive()* 、*terminate()* 和*exitcode* 方法只能由创建进程对象的进程调用。

Process 一些方法的示例用法：

```
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<Process(Process-1, initial)> False
>>> p.start()
>>> print(p, p.is_alive())
<Process(Process-1, started)> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<Process(Process-1, stopped[SIGTERM])> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.**ProcessError**
所有*multiprocessing* 异常的基类。

exception multiprocessing.**BufferTooShort**
当提供的缓冲区对象太小而无法读取消息时，Connection.recv_bytes_into() 引发的异常。

如果 e 是一个*BufferTooShort* 实例，那么 e.args[0] 将把消息作为字节字符串给出。

exception multiprocessing.**AuthenticationError**
出现身份验证错误时引发。

exception multiprocessing.**TimeoutError**
有超时的方法超时时引发。

管道和队列

使用多进程时，一般使用消息机制实现进程间通信，尽可能避免使用同步原语，例如锁。

消息机制包含：*Pipe()* (可以用于在两个进程间传递消息)，以及队列 (能够在多个生产者和消费者之间通信)。

Queue, *SimpleQueue* 以及 *JoinableQueue* 都是多生产者，多消费者，并且实现了 FIFO 的队列类型，其表现与标准库中的 *queue.Queue* 类相似。不同之处在于 *Queue* 缺少标准库的 *queue.Queue* 从 Python 2.5 开始引入的 *task_done()* 和 *join()* 方法。

如果你使用了 *JoinableQueue* ，那么你 **必须** 对每个已经移出队列的任务调用 *JoinableQueue.task_done()*。不然的话用于统计未完成任务的信号量最终会溢出并抛出异常。

另外还可以通过使用一个管理器对象创建一个共享队列，详见 数据管理器 。

注解： *multiprocessing* 使用了普通的 *queue.Empty* 和 *queue.Full* 异常去表示超时。你需要从 *queue* 中导入它们，因为它们并不在 *multiprocessing* 的命名空间中。

注解： 当一个对象被放入一个队列中时，这个对象首先会被一个后台线程用 pickle 序列化，并将序列化后的数据通过一个底层管道的管道传递到队列中。这种做法会有点让人惊讶，但一般不会出现什么问题。如果它们确实妨碍了你，你可以使用一个由管理器 *manager* 创建的队列替换它。

(1) 将一个对象放入一个空队列后，可能需要极小的延迟，队列的方法 *empty()* 才会返回 *False* 。而 *get_nowait()* 可以不抛出 *queue.Empty* 直接返回。

(2) 如果有多个进程同时将对象放入队列，那么在队列的另一端接受到的对象可能是无序的。但是由同一个进程放入的多个对象的顺序在另一端输出时总是一样的。

警告： 如果一个进程通过调用 *Process.terminate()* 或 *os.kill()* 在尝试使用 *Queue* 期间被终止了，那么队列中的数据很可能被破坏。这可能导致其他进程在尝试使用该队列时遇到异常。

警告： 正如刚才提到的，如果一个子进程将一些对象放进队列中 (并且它没有用 *JoinableQueue.cancel_join_thread* 方法)，那么这个进程在所有缓冲区的对象被刷新进管道之前，是不会终止的。

这意味着，除非你确定所有放入队列中的对象都已经被消费了，否则如果你试图等待这个进程，你可能会陷入死锁中。相似地，如果该子进程不是后台进程，那么父进程可能在试图等待所有非后台进程退出时挂起。

注意用管理器创建的队列不存在这个问题，详见 编程指导 。

该 例子 展示了如何使用队列实现进程间通信。

`multiprocessing.`**`Pipe`**`(`[*duplex*]`)`
返回一对 `Connection`对象 ``(conn1, conn2)`` ，分别表示管道的两端。

如果 *duplex* 被置为 `True` (默认值)，那么该管道是双向的。如果 *duplex* 被置为 `False` ，那么该管道是单向的，即 `conn1` 只能用于接收消息，而 `conn2` 仅能用于发送消息。

`class` `multiprocessing.`**`Queue`**`(`[*maxsize*]`)`
返回一个使用一个管道和少量锁和信号量实现的共享队列实例。当一个进程将一个对象放进队列中时，一个写入线程会启动并将对象从缓冲区写入管道中。

一旦超时，将抛出标准库*queue*模块中常见的异常*queue.Empty*和*queue.Full*。

除了*task_done()*和*join()*之外，*Queue*实现了标准库类*queue.Queue*中所有的方法。

qsize()
返回队列的大致长度。由于多线程或者多进程的上下文，这个数字是不可靠的。

注意，在 Unix 平台上，例如 Mac OS X，这个方法可能会抛出*NotImplementedError*异常，因为该平台没有实现 sem_getvalue() 。

empty()
如果队列是空的，返回 True ，反之返回 False 。由于多线程或多进程的环境，该状态是不可靠的。

full()
如果队列是满的，返回 True ，反之返回 False 。由于多线程或多进程的环境，该状态是不可靠的。

put(*obj*[, *block*[, *timeout*]])
将 obj 放入队列。如果可选参数 *block* 是 True (默认值) 而且 *timeout* 是 None (默认值), 将会阻塞当前进程，直到有空的缓冲槽。如果 *timeout* 是正数，将会在阻塞了最多 *timeout* 秒之后还是没有可用的缓冲槽时抛出*queue.Full*异常。反之 (*block* 是 False 时)，仅当有可用缓冲槽时才放入对象，否则抛出*queue.Full*异常 (在这种情形下 *timeout* 参数会被忽略)。

put_nowait(*obj*)
相当于 put(obj, False)。

get([*block*[, *timeout*]])
从队列中取出并返回对象。如果可选参数 *block* 是 True (默认值) 而且 *timeout* 是 None (默认值), 将会阻塞当前进程，直到队列中出现可用的对象。如果 *timeout* 是正数，将会在阻塞了最多 *timeout* 秒之后还是没有可用的对象时抛出*queue.Empty*异常。反之 (*block* 是 False 时)，仅当有可用对象能够取出时返回，否则抛出*queue.Empty*异常 (在这种情形下 *timeout* 参数会被忽略)。

get_nowait()
相当于 get(False) 。

*multiprocessing.Queue*类有一些在*queue.Queue*类中没有出现的方法。这些方法在大多数情形下并不是必须的。

close()
指示当前进程将不会再往队列中放入对象。一旦所有缓冲区中的数据被写入管道之后，后台的线程会退出。这个方法在队列被 gc 回收时会自动调用。

join_thread()
等待后台线程。这个方法仅在调用了*close()*方法之后可用。这会阻塞当前进程，直到后台线程退出，确保所有缓冲区中的数据都被写入管道中。

默认情况下，如果一个不是队列创建者的进程试图退出，它会尝试等待这个队列的后台线程。这个进程可以使用*cancel_join_thread()*让*join_thread()*方法什么都不做直接跳过。

cancel_join_thread()
防止*join_thread()*方法阻塞当前进程。具体而言，这防止进程退出时自动等待后台线程退出。详见*join_thread()*。

可能这个方法称为"allow_exit_without_flush() "会更好。这有可能会导致正在排队进入队列的数据丢失，大多数情况下你不需要用到这个方法，仅当你不关心底层管道中可能丢失的数据，只是希望进程能够马上退出时使用。

注解: 该类的功能依赖于宿主操作系统具有可用的共享信号量实现。否则该类将被禁用，任何试图实例化一个*Queue*对象的操作都会抛出*ImportError*异常，更多信息详见 bpo-3770 。后续说明的任何

专用队列对象亦如此。

class multiprocessing.**SimpleQueue**
这是一个简化的*Queue*类的实现，很像带锁的*Pipe*。

empty()
如果队列为空返回 True，否则返回 False。

get()
从队列中移出并返回一个对象。

put(*item*)
将 *item* 放入队列。

class multiprocessing.**JoinableQueue**([*maxsize*])
*JoinableQueue*类是*Queue*的子类，额外添加了*task_done()*和*join()*方法。

task_done()
指出之前进入队列的任务已经完成。由队列的消费者进程使用。对于每次调用*get()*获取的任务，执行完成后调用*task_done()*告诉队列该任务已经处理完成。

如果*join()*方法正在阻塞之中，该方法会在所有对象都被处理完的时候返回 (即对之前使用*put()*放进队列中的所有对象都已经返回了对应的*task_done()*)。

如果被调用的次数多于放入队列中的项目数量，将引发*ValueError*异常。

join()
阻塞至队列中所有的元素都被接收和处理完毕。

当条目添加到队列的时候，未完成任务的计数就会增加。每当消费者进程调用*task_done()*表示这个条目已经被回收，该条目所有工作已经完成，未完成计数就会减少。当未完成计数降到零的时候，join() 阻塞被解除。

杂项

multiprocessing.**active_children**()
返回当前进程存活的子进程的列表。

调用该方法有“等待”已经结束的进程的副作用。

multiprocessing.**cpu_count**()
返回系统的 CPU 数量。

该数量不同于当前进程可以使用的 CPU 数量。可用的 CPU 数量可以由 len(os.sched_getaffinity(0)) 方法获得。

可能引发*NotImplementedError*。

参见:

os.cpu_count()

multiprocessing.**current_process**()
返回与当前进程相对应的*Process*对象。

和*threading.current_thread()*相同。

multiprocessing.**freeze_support**()
为使用了*multiprocessing*的程序，提供冻结以产生 Windows 可执行文件的支持。(在 **py2exe**, **PyInstaller** 和 **cx_Freeze** 上测试通过)

需要在 main 模块的 if __name__ == '__main__' 该行之后马上调用该函数。例如:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

如果没有调用 freeze_support() 在尝试运行被冻结的可执行文件时会抛出*RuntimeError* 异常。

对 freeze_support() 的调用在非 Windows 平台上是无效的。如果该模块在 Windows 平台的 Python 解释器中正常运行 (该程序没有被冻结)，调用 “freeze_support()“ 也是无效的。

multiprocessing.**get_all_start_methods**()
返回支持的启动方法的列表，该列表的首项即为默认选项。可能的启动方法有 'fork', 'spawn' 和 “’ forkserver’。在 Windows 中，只有 ``'spawn' 是可用的。Unix 平台总是支持 “’ fork’ “ 和 “’ spawn’，且 ``'fork' 是默认值。

3.4 新版功能.

multiprocessing.**get_context**(*method=None*)
返回一个 Context 对象。该对象具有和*multiprocessing* 模块相同的 API。

如果 *method* 设置成 None 那么将返回默认上下文对象。否则 *method* 应该是 'fork', 'spawn', 'forkserver' 。如果指定的启动方法不存在，将抛出*ValueError* 异常。

3.4 新版功能.

multiprocessing.**get_start_method**(*allow_none=False*)
返回启动进程时使用的启动方法名。

如果启动方法已经固定，并且 *allow_none* 被设置成 False ，那么启动方法将被固定为默认的启动方法，并且返回其方法名。如果启动方法没有设定，并且 *allow_none* 被设置成 True ，那么将返回 None 。

返回值将为 'fork' ，'spawn' ，'forkserver' 或者 None 。'fork'``是 Unix 的默认值，``'spawn' 是 Windows 的默认值。

3.4 新版功能.

multiprocessing.**set_executable**()
设置在启动子进程时使用的 Python 解释器路径。(默认使用*sys.executable*) 嵌入式编程人员可能需要这样做:

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

以使他们可以创建子进程。

在 3.4 版更改: 现在在 Unix 平台上使用 'spawn' 启动方法时支持调用该方法。

multiprocessing.**set_start_method**(*method*)
设置启动子进程的方法。*method* 可以是 'fork' ，'spawn' 或者 'forkserver' 。

注意这最多只能调用一次，并且需要藏在 main 模块中，由 if __name__ == '__main__' 保护着。

3.4 新版功能.

注解: *multiprocessing* 并没有包含类似*threading.active_count()* , *threading.enumerate()* , *threading.settrace()* , *threading.setprofile()*, *threading.Timer* , 或者*threading.local* 的方法和类。

连接对象（Connection）

Connection 对象允许收发可以序列化的对象或字符串。它们可以看作面向消息的连接套接字。

通常使用*Pipe* 创建 Connection 对象。详见：*监听者及客户端*.

class multiprocessing.connection.**Connection**

send(*obj*)
将一个对象发送到连接的另一端，可以用*recv()* 读取。

The object must be picklable. Very large pickles (approximately 32 MB+, though it depends on the OS) may raise a *ValueError* exception.

recv()
返回一个由另一端使用*send()* 发送的对象。该方法会一直阻塞直到接收到对象。如果对端关闭了连接或者没有东西可接收，将抛出*EOFError* 异常。

fileno()
返回由连接对象使用的描述符或者句柄。

close()
关闭连接对象。

当连接对象被垃圾回收时会自动调用。

poll([*timeout*])
返回连接对象中是否有可以读取的数据。

如果未指定 *timeout*，此方法会马上返回。如果 *timeout* 是一个数字，则指定了最大阻塞的秒数。如果 *timeout* 是 None ，那么将一直等待，不会超时。

注意通过使用*multiprocessing.connection.wait()* 可以一次轮询多个连接对象。

send_bytes(*buffer*[, *offset*[, *size*]])
从一个*bytes-like object* （字节类对象）对象中取出字节数组并作为一条完整消息发送。

If *offset* is given then data is read from that position in *buffer*. If *size* is given then that many bytes will be read from buffer. Very large buffers (approximately 32 MB+, though it depends on the OS) may raise a *ValueError* exception

recv_bytes([*maxlength*])
以字符串形式返回一条从连接对象另一端发送过来的字节数据。此方法在接收到数据前将一直阻塞。如果连接对象被对端关闭或者没有数据可读取，将抛出*EOFError* 异常。

如果给定了 *maxlength* 并且消息长于 *maxlength* 那么将抛出*OSError* 并且该连接对象将不再可读。

在 3.3 版更改: 曾经该函数抛出*IOError*，现在这是*OSError* 的别名。

recv_bytes_into(*buffer*[, *offset*])
将一条完整的字节数据消息读入 *buffer* 中并返回消息的字节数。此方法在接收到数据前将一直阻塞。如果连接对象被对端关闭或者没有数据可读取，将抛出*EOFError* 异常。

buffer must be a writable *bytes-like object*. If *offset* is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of *buffer* (in bytes).

如果缓冲区太小，则将引发 BufferTooShort 异常，并且完整的消息将会存放在异常实例 e 的 e.args[0] 中。

在 3.3 版更改: 现在连接对象自身可以通过*Connection.send()* 和*Connection.recv()* 在进程之间传递。

3.3 新版功能: 连接对象现已支持上下文管理协议–参见 see 上下文管理器类型 。*__enter__()* 返回连接对象，*__exit__()* 会调用*close()* 。

例如:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

警告: *Connection.recv()* 方法会自动解封它收到的数据，除非你能够信任发送消息的进程，否则此处可能有安全风险。

因此，除非连接对象是由 `Pipe()` 产生的，否则你应该仅在使用了某种认证手段之后才使用*recv()* 和*send()* 方法。参考认证密码。

警告: 如果一个进程在试图读写管道时被终止了，那么管道中的数据很可能是不完整的，因为此时可能无法确定消息的边界。

同步原语

通常来说同步原语在多进程环境中并不像它们在多线程环境中那么必要。参考*threading* 模块的文档。

注意可以使用管理器对象创建同步原语，参考数据管理器 。

class multiprocessing.**Barrier**(*parties*[, *action*[, *timeout*]])
类似*threading.Barrier* 的栅栏对象。

3.3 新版功能.

class multiprocessing.**BoundedSemaphore**([*value*])
非常类似*threading.BoundedSemaphore* 的有界信号量对象。

一个小小的不同在于，它的 `acquire` 方法的第一个参数名是和*Lock.acquire()* 一样的 *block* 。

注解: 在 Mac OS X 平台上，该对象于*Semaphore* 不同在于 `sem_getvalue()` 方法并没有在该平台上实现。

class multiprocessing.**Condition**([*lock*])
条件变量: *threading.Condition* 的别名。

指定的 *lock* 参数应该是*multiprocessing* 模块中的*Lock* 或者*RLock* 对象。

在 3.3 版更改: 新增了*wait_for()* 方法。

class multiprocessing.**Event**
A clone of *threading.Event*.

class multiprocessing.**Lock**
原始锁（非递归锁）对象，类似于*threading.Lock* 。一旦一个进程或者线程拿到了锁，后续的任何其他进程或线程的其他请求都会被阻塞直到锁被释放。任何进程或线程都可以释放锁。除非另有说明，否则*multiprocessing.Lock* 用于进程或者线程的概念和行为都和*threading.Lock* 一致。

注意*Lock* 实际上是一个工厂函数。它返回由默认上下文初始化的 multiprocessing.synchronize.Lock 对象。

Lock supports the *context manager* protocol and thus may be used in with statements.

acquire(*block=True*, *timeout=None*)
可以阻塞或非阻塞地获得锁。

如果 *block* 参数被设为 True（默认值），对该方法的调用在锁处于释放状态之前都会阻塞，然后将锁设置为锁住状态并返回 True 。需要注意的是第一个参数名与*threading.Lock.acquire()* 的不同。

如果 *block* 参数被设置成 False ，方法的调用将不会阻塞。如果锁当前处于锁住状态，将返回 False ；否则将锁设置成锁住状态，并返回 True 。

当 *timeout* 是一个正浮点数时，会在等待锁的过程中最多阻塞等待 *timeout* 秒，当 *timeout* 是负数时，效果和 *timeout* 为 0 时一样，当 *timeout* 是 None （默认值）时，等待时间是无限长。需要注意的是，对于 *timeout* 是负数和 None 的情况，其行为与*threading.Lock.acquire()* 是不一样的。当 *block* 参数为 False 时，*timeout* 并没有实际用处，会直接忽略，当 *block* 参数为 True 时，函数会在拿到锁后返回 True 或者超时没拿到锁后返回 False 。

release()
释放锁，可以在任何进程、线程使用，并不限于锁的拥有者。

其行为与*threading.Lock.release()* 一样，只不过当尝试释放一个没有被持有的锁时，会抛出*ValueError* 异常。

class multiprocessing.**RLock**
递归锁对象: 类似于*threading.RLock* 。递归锁必须由持有线程、进程亲自释放。如果某个进程或者线程拿到了递归锁，这个进程或者线程可以再次拿到这个锁而不需要等待。但是这个进程或者线程的拿锁操作和释放锁操作的次数必须相同。

注意*RLock* 是一个工厂函数，调用后返回一个使用默认 context 初始化的 multiprocessing.synchronize.RLock 对象。

RLock 支持*context manager* 协议，因此可在 with 语句内使用。

acquire(*block=True*, *timeout=None*)
可以阻塞或非阻塞地获得锁。

当 *block* 设置为 True 时，会一直阻塞直到锁处于空闲状态（没有被任何进程、线程拥有），除非当前进程/线程已经拥有了这把锁。然后当前进程会持有这把锁（在锁没有被持有者的情况下），锁内的递归等级加一，并返回 True . 注意，这个函数第一个参数和*threading.RLock.acquire()* 有几个不同点，包括参数名本身。

当 *block* 参数是 False , 将不会阻塞，如果此时锁被其他进程或者线程持有，当前进程、线程获取锁操作失败，锁的递归等级也不会改变，函数返回 False , 如果当前锁已经处于释放状态，则当前进程、线程则会拿到锁，并且锁内的递归等级加一，函数返回 True 。

timeout 参数的使用方法及行为与*Lock.acquire()* 一样。但是要注意 *timeout* 的其中一些行为和*threading.RLock.acquire()* 中实现的行为是不同的。

release()
释放锁，使锁内的递归等级减一。如果释放后锁内的递归等级降低为 0，则会重置锁的状态为释放状态（即没有被任何进程、线程持有），重置后如果有有其他进程和线程在等待这把锁，他们中的一个会获得这个锁而继续运行。如果释放后锁内的递归等级还没到达 0，则这个锁仍将保持未释放状态且当前进程和线程仍然是持有者。

只有当前进程或线程是锁的持有者时，才允许调用这个方法。如果当前进程或线程不是这个锁的拥有者，或者这个锁处于已释放的状态(即没有任何拥有者)，调用这个方法会抛出*AssertionError*异常。注意这里抛出的异常类型和*threading.RLock.release()* 中实现的行为不一样。

class multiprocessing.**Semaphore**([*value*])
一种信号量对象: 类似于*threading.Semaphore*.

一个小小的不同在于，它的 acquire 方法的第一个参数名是和*Lock.acquire()* 一样的 *block* 。

注解: 在 Mac OS X 上，不支持 sem_timedwait ，所以，使用调用 acquire() 时如果使用 timeout 参数，会通过循环 sleep 来模拟 timeout 的行为。

注解: 假如信号 SIGINT 是来自于 Ctrl-C ，并且主线程被 BoundedSemaphore.acquire(), *Lock.acquire()*, *RLock.acquire()*, Semaphore.acquire(), Condition.acquire() 或 Condition.wait() 阻塞，则调用会立即中断同时抛出*KeyboardInterrupt* 异常。

这和*threading* 的行为不同，此模块中当执行对应的阻塞式调用时，SIGINT 会被忽略。

注解: 这个库的某些功能依赖于宿主机系统的共享信号量，如果系统没有这个特性，multiprocessing.synchronize 会被禁用，尝试导入这个包会引发*ImportError* 异常，详细信息请查看 bpo-3770 。

共享 ctypes 对象

在共享内存上创建可被子进程继承的共享对象时是可行的。

multiprocessing.**Value**(*typecode_or_type*, **args*, *lock=True*)
返回一个从共享内存上创建的*ctypes* 对象。默认情况下返回的实际上是经过了同步包装器包装过的。可以通过*Value* 的 *value* 属性访问这个对象本身。

typecode_or_type 指明了返回的对象类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。**args* 会透传给这个类的构造函数。

如果 *lock* 参数是 True （默认值），将会新建一个递归锁用于同步对于此值的访问操作。如果 *lock* 是*Lock* 或者*RLock* 对象，那么这个传入的锁将会用于同步对这个值的访问操作，如果 *lock* 是 False ，那么对这个对象的访问将没有锁保护，也就是说这个变量不是进程安全的。

诸如 += 这类的操作会引发独立的读操作和写操作，也就是说这类操作符并不具有原子性。所以，如果你想让递增操作具有原子性，这样的方式并不能达到要求:

```
counter.value += 1
```

假设共享对象内部关联的锁时递归锁 (默认情况下就是)，那么你可以采用这种方式

```
with counter.get_lock():
    counter.value += 1
```

注意 *locl* 只能是命名参数。

multiprocessing.**Array**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)
从共享内存中申请并返回一个具有 ctypes 类型的数组对象。默认情况下返回值实际上是被同步器包装过的数组对象。

typecode_or_type 指明了返回的数组中的元素类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。如果 *size_or_initializer* 是一个整数，那就会当做数组的长度，并且整个数组的内存会初始化为 0。否则，如果 *size_or_initializer* 会被当成一个序列用于初始化数组中的内一个元素，并且会根据元素个数自动判断数组的长度。

如果 *lock* 为 `True` (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个*Lock* 或*RLock* 对象则该对象将被用于同步对值的访问。如果 *lock* 为 `False` 则对返回对象的访问将不会自动得到锁的保护，也就是说它不是“进程安全的”。

请注意 *lock* 是一个仅限关键字参数。

请注意*ctypes.c_char* 的数组具有 *value* 和 *raw* 属性，允许被用来保存和提取字符串。

multiprocessing.sharedctypes 模块

multiprocessing.sharedctypes 模块提供了一些函数，用于分配来自共享内存的、可被子进程继承的*ctypes* 对象。

注解: 虽然可以将指针存储在共享内存中，但请记住它所引用的是特定进程地址空间中的位置。而且，指针很可能在第二个进程的上下文中无效，尝试从第二个进程对指针进行解引用可能会导致崩溃。

---

multiprocessing.sharedctypes.**RawArray**(*typecode_or_type*, *size_or_initializer*)
从共享内存中申请并返回一个 ctypes 数组。

typecode_or_type 指明了返回的数组中的元素类型: 它可能是一个 ctypes 类型或者*array* 模块中使用的类型字符。如果 *size_or_initializer* 是一个整数，那就会当做数组的长度，并且整个数组的内存会初始化为 0。否则，如果 *size_or_initializer* 会被当成一个序列用于初始化数组中的每一个元素，并且会根据元素个数自动判断数组的长度。

注意对元素的访问、赋值操作可能是非原子操作 - 使用*Array()* 来借助其中的锁保证操作的原子性。

multiprocessing.sharedctypes.**RawValue**(*typecode_or_type*, **args*)
从共享内存中申请并返回一个 ctypes 对象。

typecode_or_type 指明了返回的对象类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。**args* 会透传给这个类的构造函数。

注意对 value 的访问、赋值操作可能是非原子操作 - 使用*Value()* 来借助其中的锁保证操作的原子性。

请注意*ctypes.c_char* 的数组具有 *value* 和 *raw* 属性，允许被用来保存和提取字符串 - 请查看*ctypes* 文档。

multiprocessing.sharedctypes.**Array**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)
返回一个纯 ctypes 数组, 或者在此之上经过同步器包装过的对象，这取决于 *lock* 参数的值，除此之外，和*RawArray()* 一样。

如果 *lock* 为 `True` (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个*Lock* 或*RLock* 对象则该对象将被用于同步对值的访问。如果 *lock* 为 `False` 则对所返回对象的访问将不会自动得到锁的保护，也就是说它将不是“进程安全的”。

注意 *locl* 只能是命名参数。

multiprocessing.sharedctypes.**Value**(*typecode_or_type*, **args*, *lock=True*)
返回一个纯 ctypes 数组, 或者在此之上经过同步器包装过的进程安全的对象，这取决于 *lock* 参数的值，除此之外，和 `RawArray()` 一样。

如果 *lock* 为 `True` (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个*Lock* 或*RLock* 对象则该对象将被用于同步对值的访问。如果 *lock* 为 `False` 则对所返回对象的访问将不会自动得到锁的保护，也就是说它将不是“进程安全的”。

注意 *locl* 只能是命名参数。

`multiprocessing.sharedctypes.`**`copy`**(*obj*)
从共享内存中申请一片空间将 ctypes 对象 *obj* 过来，然后返回一个新的 ctypes 对象。

`multiprocessing.sharedctypes.`**`synchronized`**(*obj*[, *lock*])
将一个 ctypes 对象包装为进程安全的对象并返回，使用 *lock* 同步对于它的操作。如果 *lock* 是 `None` (默认值)，则会自动创建一个*multiprocessing.RLock* 对象。

同步器包装后的对象会在原有对象基础上额外增加两个方法: `get_obj()` 返回被包装的对象，`get_lock()` 返回内部用于同步的锁。

需要注意的是，访问包装后的 ctypes 对象会比直接访问原来的纯 ctypes 对象慢得多。

在 3.5 版更改: 同步器包装后的对象支持*context manager* 协议。

下面的表格对比了创建普通 ctypes 对象和基于共享内存上创建共享 ctypes 对象的语法。(表格中的 `MyStruct` 是*ctypes.Structure* 的子类)

ctypes	使用类型的共享 ctypes	使用 typecode 的共享 ctypes
c_double(2.4)	RawValue(c_double, 2.4)	RawValue(‘d’, 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray(‘h’, 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray(‘i’, (9, 2, 8))

下面是一个在子进程中修改多个 ctypes 对象的例子。

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()
```

(下页继续)

(续上页)

```
    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

输出如下

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

数据管理器

管理器提供了一种创建共享数据的方法，从而可以在不同进程中共享，甚至可以通过网络跨机器共享数据。管理器维护一个用于管理 共享对象的服务。其他进程可以通过代理访问这些共享对象。

multiprocessing.**Manager**()
返回一个已启动的*SyncManager* 管理器对象，这个对象可以用于在不同进程中共享数据。返回的管理器对象对应了一个 spawned 方式启动的子进程，并且拥有一系列方法可以用于创建共享对象、返回对应的代理。

当管理器被垃圾回收或者父进程退出时，管理器进程会立即退出。管理器类定义在*multiprocessing.managers* 模块:

class multiprocessing.managers.**BaseManager**([*address*[, *authkey*]])
创建一个 BaseManager 对象。

一旦创建，应该及时调用*start()* 或者 get_server().serve_forever() 以确保管理器对象对应的管理进程已经启动。

address 是管理器对象监听的地址。如果 *address* 是 None，则允许和任意主机的请求建立连接。

authkey 是认证标识，用于检查连接服务进程的请求合法性。如果 *authkey* 是 None，则会使用 current_process().authkey，否则，就使用 *authkey*，需要保证它必须是 byte 类型的字符串。

start([*initializer*[, *initargs*]])
为管理器开启一个子进程，如果 *initializer* 不是 None，子进程在启动时将会调用 initializer(*initargs)。

get_server()
返回一个 Server 对象，它是管理器在后台控制的真实的服务。Server 对象拥有 serve_forever() 方法。

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

Server 额外拥有一个*address* 属性。

connect()
将本地管理器对象连接到一个远程管理器进程:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 5000), authkey=b'abc')
>>> m.connect()
```

shutdown()
停止管理器的进程。这个方法只能用于已经使用*start()*启动的服务进程。

它可以被多次调用。

register(*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]]])
一个 classmethod，可以将一个类型或者可调用对象注册到管理器类。

typeid 是一种“类型标识”，用于唯一表示某种共享类型，必须是一个字符串。

callable 是一个用来为此类型标识符创建对象的可调用对象。如果一个管理器实例将使用*connect()*方法连接到服务器，或者 *create_method* 参数为 `False`，那么这里可留下 `None`。

proxytype 是*BaseProxy* 的子类，可以根据 *typeid* 为共享对象创建一个代理，如果是 `None`，则会自动创建一个代理类。

exposed 是一个函数名组成的序列，用来指明只有这些方法可以使用*BaseProxy._callmethod()* 代理。(如果 *exposed* 是 `None`, 则会在 `proxytype._exposed_` 存在的情况下转而使用它) 当暴露的方法列表没有指定的时候，共享对象的所有“公共方法”都会被代理。(这里的“公共方法”是指所有拥有 `__call__()` 方法并且不是以 `'_'` 开头的属性)

method_to_typeid 是一个映射，用来指定那些应该返回代理对象的暴露方法所返回的类型。(如果 *method_to_typeid* 是 `None`, 则 `proxytype._method_to_typeid_` 会在存在的情况下被使用) 如果方法名称不在这个映射中或者映射是 `None`，则方法返回的对象会是一个值拷贝。

create_method 指明，是否要创建一个以 *typeid* 命名并返回一个代理对象的函数，这个函数会被服务进程用于创建共享对象，默认为 `True`。

BaseManager 实例也有一个只读属性。

address
管理器所用的地址。

在 3.3 版更改: 管理器对象支持上下文管理协议 - 查看*上下文管理器类型*。*__enter__()* 启动服务进程（如果它还没有启动）并且返回管理器对象，*__exit__()* 会调用*shutdown()*。

在之前的版本中，如果管理器服务进程没有启动，*__enter__()* 不会负责启动它。

class multiprocessing.managers.**SyncManager**
BaseManager 的子类，可用于进程的同步。这个类型的对象使用 `multiprocessing.Manager()` 创建。

它拥有一系列方法，可以为大部分常用数据类型创建并返回*代理对象* 代理，用于进程间同步。甚至包括共享列表和字典。

Barrier(*parties*[, *action*[, *timeout*]])
创建一个共享的*threading.Barrier* 对象并返回它的代理。

3.3 新版功能.

BoundedSemaphore([*value*])
创建一个共享的*threading.BoundedSemaphore* 对象并返回它的代理。

Condition([*lock*])
创建一个共享的*threading.Condition* 对象并返回它的代理。

如果提供了 *lock* 参数，那它必须是*threading.Lock* 或*threading.RLock* 的代理对象。

在 3.3 版更改: 新增了*wait_for()* 方法。

Event()
创建一个共享的*threading.Event*对象并返回它的代理。

Lock()
创建一个共享的*threading.Lock*对象并返回它的代理。

Namespace()
创建一个共享的 Namespace` 对象并返回它的代理。

Queue([*maxsize*])
创建一个共享的*queue.Queue*对象并返回它的代理。

RLock()
创建一个共享的*threading.RLock*对象并返回它的代理。

Semaphore([*value*])
创建一个共享的*threading.Semaphore*对象并返回它的代理。

Array(*typecode*, *sequence*)
创建一个数组并返回它的代理。

Value(*typecode*, *value*)
创建一个具有可写 value 属性的对象并返回它的代理。

dict()
dict(*mapping*)
dict(*sequence*)
创建一个共享的*dict*对象并返回它的代理。

list()
list(*sequence*)
创建一个共享的*list*对象并返回它的代理。

在 3.6 版更改: 共享对象能够嵌套。例如, 共享的容器对象如共享列表，可以包含另一个共享对象，他们全都会在*SyncManager*中进行管理和同步。

class multiprocessing.managers.**Namespace**
一个可以注册到*SyncManager*的类型。

命名空间对象没有公共方法，但是拥有可写的属性。它的表示 (repr) 会显示所有属性的值。

值得一提的是，当对命名空间对象使用代理的时候，访问所有名称以 '_' 开头的属性都只是代理器上的属性，而不是命名空间对象的属性。

```
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

自定义管理器

要创建一个自定义的管理器，需要新建一个*BaseManager*的子类，然后使用这个管理器类上的*register()*类方法将新类型或者可调用方法注册上去。例如:

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # prints 7
        print(maths.mul(7, 8))         # prints 56
```

使用远程管理器

可以将管理器服务运行在一台机器上，然后使用客户端从其他机器上访问。(假设它们的防火墙允许这样的网络通信)

运行下面的代码可以启动一个服务，此付包含了一个共享队列，允许远程客户端访问:

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

远程客户端可以通过下面的方式访问服务:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

也可以通过下面的方式:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
```

(下页继续)

(续上页)

```
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

本地进程也可以访问这个队列，利用上面的客户端代码通过远程方式访问:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super(Worker, self).__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

代理对象

代理是一个 指向其他共享对象的对象，这个对象 (很可能) 在另外一个进程中。共享对象也可以说是代理 指涉的对象。多个代理对象可能指向同一个指涉对象。

代理对象代理了指涉对象的一系列方法调用 (虽然并不是指涉对象的每个方法都有必要被代理)。通过这种方式，代理的使用方法可以和它的指涉对象一样:

```
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

注意，对代理使用*str()* 函数会返回指涉对象的字符串表示，但是*repr()* 却会返回代理本身的内部字符串表示。

被代理的对象很重要的一点是必须可以被序列化，这样才能允许他们在进程间传递。因此，指涉对象可以包含*代理对象* 。这允许管理器中列表、字典或者其他*代理对象* 对象之间的嵌套。

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent of b
```

(下页继续)

(续上页)

```
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

类似地，字典和列表代理也可以相互嵌套:

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

如果指涉对象包含了普通*list* 或*dict* 对象，对这些内部可变对象的修改不会通过管理器传播，因为代理无法得知被包含的值什么时候被修改了。但是把存放在容器代理中的值本身是会通过管理器传播的（会触发代理对象中的 __setitem__ ）从而有效修改这些对象，所以可以把修改过的值重新赋值给容器代理:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

在大多是使用情形下，这种实现方式并不比嵌套代理对象 方便，但是依然演示了对于同步的一种控制级别。

注解: *multiprocessing* 中的代理类并没有提供任何对于代理值比较的支持。所以，我们会得到如下结果:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

当需要比较值的时候，应该替换为使用指涉对象的拷贝。

---

class multiprocessing.managers.**BaseProxy**

代理对象是*BaseProxy* 派生类的实例。

_callmethod(*methodname*[, *args*[, *kwds*]])

调用指涉对象的方法并返回结果。

如果 proxy 是一个代理且其指涉的是 obj，那么下面的表达式:

```
proxy._callmethod(methodname, args, kwds)
```

相当于求取以下表达式的值:

```
getattr(obj, methodname)(*args, **kwds)
```

于管理器进程。

返回结果会是一个值拷贝或者一个新的共享对象的代理 - 见函数*BaseManager.register()* 中关于参数 *method_to_typeid* 的文档。

如果这个调用熬出了异常，则这个异常会被*_callmethod()* 透传出来。如果是管理器进程本身抛出的一些其他异常，则会被*_callmethod()* 转换为 RemoteError 异常重新抛出。

特别注意，如果 *methodname* 没有 暴露出来，将会引发一个异常。

_callmethod() 的一个使用示例:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # equivalent to l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()
返回指涉对象的一份拷贝。

如果指涉对象无法序列化，则会抛出一个异常。

__repr__()
返回代理对象的字符串表示。

__str__()
返回指涉对象的字符串表示。

清理

代理对象使用了一个弱引用回调，当它被垃圾回收时，会将自己从拥有此指涉对象的管理器上反注册，

当共享对象没有被任何代理器引用时，会被管理器进程删除。

进程池

可以创建一个进程池，它将使用*Pool* 类执行提交给它的任务。

class multiprocessing.pool.**Pool**([*processes*[, *initializer*[, *initargs*[, *maxtasksperchild*[, *context*]]]]])

一个进程池对象，它控制可以提交作业的工作进程池。它支持带有超时和回调的异步结果，以及一个并行的 map 实现。

processes 是要使用的工作进程数目。如果 *processes* 为 None，则使用*os.cpu_count()* 返回的值。

如果 *initializer* 不为 None，则每个工作进程将会在启动时调用 initializer(*initargs)。

maxtasksperchild 是一个工作进程在它退出或被一个新的工作进程代替之前能完成的任务数量，为了释放未使用的资源。默认的 *maxtasksperchild* 是 None，意味着工作进程寿与池齐。

context 可被用于指定启动的工作进程的上下文。通常一个进程池是使用函数 multiprocessing.Pool() 或者一个上下文对象的*Pool()* 方法创建的。在这两种情况下，*context* 都是适当设置的。

注意，进程池对象的方法只有创建它的进程能够调用。

3.2 新版功能: *maxtasksperchild*

3.4 新版功能: *context*

注解: 通常来说，`Pool` 中的 Worker 进程的生命周期和进程池的工作队列一样长。一些其他系统中（如 Apache, mod_wsgi 等）也可以发现另一种模式，他们会让工作进程在完成一些任务后退出，清理、释放资源，然后启动一个新的进程代替旧的工作进程。`Pool` 的 *maxtasksperchild* 参数给用户提供了这种能力。

---

apply(*func*[, *args*[, *kwds*]])
使用 *args* 参数以及 *kwds* 命名参数调用 *func*，它会返回结果前阻塞。这种情况下，`apply_async()` 更适合并行化工作。另外 *func* 只会在一个进程池中的一个工作进程中执行。

apply_async(*func*[, *args*[, *kwds*[, *callback*[, *error_callback*]]]])
`apply()` 方法的一个变种，返回一个结果对象。

如果指定了 *callback*，它必须是一个接受单个参数的可调用对象。当执行成功时，*callback* 会被用于处理执行后的返回结果，否则，调用 *error_callback*。

如果指定了 *error_callback*，它必须是一个接受单个参数的可调用对象。当目标函数执行失败时，会将抛出的异常对象作为参数传递给 *error_callback* 执行。

回调函数应该立即执行完成，否则会阻塞负责处理结果的线程。

map(*func*, *iterable*[, *chunksize*])
A parallel equivalent of the `map()` built-in function (it supports only one *iterable* argument though). It blocks until the result is ready.

这个方法会将可迭代对象分割为许多块，然后提交给进程池。可以将 *chunksize* 设置为一个正整数从而（近似）指定每个块的大小可以。

map_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])
和`map()` 方法类似，但是返回一个结果对象。

如果指定了 *callback*，它必须是一个接受单个参数的可调用对象。当执行成功时，*callback* 会被用于处理执行后的返回结果，否则，调用 *error_callback*。

如果指定了 *error_callback*，它必须是一个接受单个参数的可调用对象。当目标函数执行失败时，会将抛出的异常对象作为参数传递给 *error_callback* 执行。

回调函数应该立即执行完成，否则会阻塞负责处理结果的线程。

imap(*func*, *iterable*[, *chunksize*])
A lazier version of `map()`.

chunksize 参数的作用和`map()` 方法的一样。对于很长的迭代器，给 *chunksize* 设置一个很大的值会比默认值 `1` **极大**地加快执行速度。

同样，如果 *chunksize* 是 `1`，那么`imap()` 方法所返回的迭代器的 `next()` 方法拥有一个可选的 *timeout* 参数：如果无法在 *timeout* 秒内执行得到结果，则 “next(timeout)“ 会抛出`multiprocessing.TimeoutError` 异常。

imap_unordered(*func*, *iterable*[, *chunksize*])
和`imap()` 相同，只不过通过迭代器返回的结果是任意的。（当进程池中只有一个工作进程的时候，返回结果的顺序才能认为是”有序”的）

starmap(*func*, *iterable*[, *chunksize*])
和`map()` 类似，不过 *iterable* 中的每一项会被解包再作为函数参数。

比如可迭代对象 `[(1,2), (3, 4)]` 会转化为等价于 `[func(1,2), func(3,4)]` 的调用。

3.3 新版功能.

starmap_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])
相当于*starmap()* 与*map_async()* 的结合，迭代 *iterable* 的每一项，解包作为 *func* 的参数并执行，返回用于获取结果的对象。

3.3 新版功能.

close()
阻止后续任务提交到进程池，当所有任务执行完成后，工作进程会退出。

terminate()
不必等待未完成的任务，立即停止工作进程。当进程池对象呗垃圾回收时，*terminate()* 会立即调用。

join()
等待工作进程结束。调用*join()* 前必须先调用*close()* 或者*terminate()* 。

3.3 新版功能: 进程池对象现在支持上下文管理器协议 - 参见上下文管理器类型 。*__enter__()* 返回进程池对象, *__exit__()* 会调用*terminate()* 。

class multiprocessing.pool.**AsyncResult**
Pool.apply_async() 和*Pool.map_async()* 返回对象所属的类。

get([*timeout*])
用于获取执行结果。如果 *timeout* 不是 None 并且在 *timeout* 秒内仍然没有执行完得到结果，则抛出*multiprocessing.TimeoutError* 异常。如果远程调用发生异常，这个异常会通过*get()* 重新抛出。

wait([*timeout*])
阻塞，直到返回结果，或者 *timeout* 秒后超时。

ready()
用于判断执行状态，是否已经完成。

successful()
Return whether the call completed without raising an exception. Will raise *AssertionError* if the result is not ready.

下面的例子演示了进程池的用法:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # start 4 worker processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a␣
↪single process
        print(result.get(timeout=1))         # prints "100" unless your computer is␣
↪*very* slow

        print(pool.map(f, range(10)))        # prints "[0, 1, 4,..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                      # prints "0"
        print(next(it))                      # prints "1"
        print(it.next(timeout=1))            # prints "4" unless your computer is␣
↪*very* slow
```

(下页继续)

(续上页)

```
        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))        # raises multiprocessing.TimeoutError
```

监听者及客户端

通常情况下，进程间通过队列或者*Pipe()* 返回的*Connection* 传递消息。

不过，*multiprocessing.connection* 其实提供了一些更灵活的特性。最基础的用法是通过它抽象出来的高级来操作 socket 或者 Windows 命名管道。也提供一些高级用法，如通过*hmac* 模块来支持 摘要认证，以及同时监听多个管道连接。

multiprocessing.connection.**deliver_challenge**(*connection*, *authkey*)
发送一个随机生成的消息到另一端，并等待回复。

如果收到的回复与使用 *authkey* 生成的信息摘要匹配成功，就会发送一个欢迎信息给管道另一端。否则抛出*AuthenticationError* 异常。

multiprocessing.connection.**answer_challenge**(*connection*, *authkey*)
接收一条信息，使用 *authkey* 作为键计算信息摘要，然后将摘要发送回去。

如果没有收到欢迎消息，就抛出*AuthenticationError* 异常。

multiprocessing.connection.**Client**(*address*[, *family*[, *authkey*]])
尝试在监听者上使用 *address* 地址初始化一个连接，返回*Connection* 。

连接的类型取决于 *family* 参数，但是通常可以省略，因为可以通过 *address* 的格式推导出来。(查看地址格式)

如果提供了 *authkey* 参数并且不是 None，那它必须是一个字符串并且会被当做基于 HMAC 认证的密钥。如果 *authkey* 是 None 则不会有认证行为。认证失败抛出*AuthenticationError* 异常，请查看 See 认证密码 。

class multiprocessing.connection.**Listener**([*address*[, *family*[, *backlog*[, *authkey*]]]])
可以监听连接请求，是对于绑定套接字或者 Windows 命名管道的封装。

address 是监听器对象中的绑定套接字或命名管道使用的地址。

注解: 如果使用‘0.0.0.0’作为监听地址，那么在 Windows 上这个地址无法建立连接。想要建立一个可连接的端点，应该使用‘127.0.0.1’。

---

family 是套接字 (或者命名管道) 使用的类型。它可以是以下一种: `'AF_INET'` (TCP 套接字类型), `'AF_UNIX'` (Unix 域套接字) 或者 `'AF_PIPE'` (Windows 命名管道)。其中只有第一个保证各平台可用。如果 *family* 是 `None` , 那么 family 会根据 *address* 的格式自动推导出来。如果 *address* 也是 `None` , 则取默认值。默认值为可用类型中速度最快的。见地址格式 。注意，如果 *family* 是 `'AF_UNIX'` 而 address 是 “None“ , 套接字会在一个*tempfile.mkstemp()* 创建的私有临时目录中创建。

如果监听器对象使用了套接字，*backlog* (默认值为 1) 会在套接字绑定后传递给它的*listen()* 方法。

如果提供了 *authkey* 参数并且不是 None，那它必须是一个字符串并且会被当做基于 HMAC 认证的密钥。如果 *authkey* 是 None 则不会有认证行为。认证失败抛出*AuthenticationError* 异常，请查看 See 认证密码 。

accept()
接受一个连接并返回一个*Connection* 对象，其连接到的监听器对象已绑定套接字或者命名管道。如果已经尝试过认证并且失败了，则会抛出*AuthenticationError* 异常。

close()
关闭监听器上的绑定套接字或者命名管道。此函数会在监听器被垃圾回收后自动调用。不过仍然建议显式调用函数关闭。

监听器对象拥有下列只读属性:

address
被监听器对象使用的地址。

last_accepted
最后一个连接所使用的地址。如果没有的话就是 None 。

3.3 新版功能: 监听器对象现在支持了上下文管理协议 - 见上下文管理器类型 。*__enter__()* 返回一个监听器对象, *__exit__()* 会调用*close()* 。

multiprocessing.connection.**wait**(*object_list*, *timeout=None*)
一直等待直到 *object_list* 中某个对象处于就绪状态。返回 *object_list* 中处于就绪状态的对象。如果 *timeout* 是一个浮点型，该方法会最多阻塞这么多秒。如果 *timeout* 是 None ， 则会允许阻塞的事件没有限制。timeout 为负数的情况下和为 0 的情况相同。

对于 Unix 和 Windows ， 下列对象都可以出现在 *object_list* 中

- 可读的*Connection* 对象；
- 一个已连接并且可读的*socket.socket* 对象；或者
- *Process* 对象中的*sentinel* 属性。

当一个连接或者套接字对象拥有有效的数据可被读取的时候，或者另一端关闭后，这个对象就处于就绪状态。

Unix: wait(object_list, timeout) 和 select.select(object_list, [], [], timeout) 几乎相同。差别在于，如果*select.select()* 被信号中断，它会抛出一个附带错误号为 EINTR 的*OSError* 异常，而*wait()* 不会。

Windows: *object_list* 中的元素必须是一个表示为整数的可等待的句柄 (按照 Win32 函数 WaitForMultipleObjects() 的文档中所定义) 或者一个拥有 fileno() 方法的对象，这个对象返回一个套接字句柄或者管道句柄。(注意管道和套接字两种句柄 **不是**可等待的句柄)

3.3 新版功能.

示例

下面的服务代码创建了一个使用 'secret password' 作为认证密码的监听器。它会等待连接然后发送一些数据给客户端:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

下面的代码连接到服务然后从服务器上 j 接收一些数据:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())            # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)                          # => array('i', [42, 1729, 0, 0, 0])
```

下面的代码使用了*wait()*，以便在同时等待多个进程发来消息。

```
import time, random
from multiprocessing import Process, Pipe, current_process
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it.  This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)
```

地址格式

- `'AF_INET'` 地址是（*主机*, *端口*）形式的元组类型，其中 *主机*是一个字符串，*端口*是整数。
- `'AF_UNIX'` 地址是文件系统上文件名的字符串。
- **`'AF_PIPE'` 是这种格式的字符串** `r'\.\pipe{PipeName}'` 。如果要用*Client()* 连接到一个名为 *ServerName* 的远程命名管道，应该替换为使用 `r'\`*ServerName*`\pipe{PipeName}'` 这种格式。

注意，使用两个反斜线开头的字符串默认被当做 `'AF_PIPE'` 地址而不是 `'AF_UNIX'` 。

认证密码

当使用*Connection.recv* 接收数据时，数据会自动被反序列化。不幸的是，对于一个不可信的数据源发来的数据，反序列化是存在安全风险的。所以*Listener* 和*Client()* 之间使用*hmac* 模块进行摘要认证。

认证密钥是一个 byte 类型的字符串，可以认为是和密码一样的东西，连接建立好后，双方都会要求另一方证明知道认证密钥。（这个证明过程不会通过连接发送密钥）

如果要求认证但是没有指定认证密钥，则会使用 `current_process().authkey` 的返回值 (参见*Process*)。这个值将被当前进程所创建的任何*Process* 对象自动继承。这意味着 (在默认情况下) 一个包含多进程的程序中的所有进程会在相互间建立连接的时候共享单个认证密钥。

os.urandom() 也可以用来生成合适的认证密钥。

日志记录

当前模块也提供了一些对 logging 的支持。注意，*logging* 模块本身并没有使用进程间共享的锁，所以来自于多个进程的日志可能（具体取决于使用的日志 handler）相互覆盖或者混杂。

`multiprocessing.`**`get_logger`**`()`
返回*multiprocessing* 使用的 logger，必要的话会创建一个新的。

如果创建的首个 logger 日志级别为 `logging.NOTSET` 并且没有默认 handler。通过这个 logger 打印的消息不会传递到根 logger。

注意在 Windows 上，子进程只会继承父进程 logger 的日志级别 - 对于 logger 的其他自定义项不会继承。

`multiprocessing.`**`log_to_stderr`**`()`
此函数会调用*get_logger()* 但是会在返回的 logger 上增加一个 handler，将所有输出都使用 `'[%(levelname)s/%(processName)s] %(message)s'` 的格式发送到*sys.stderr* 。

下面是一个在交互式解释器中打开日志功能的例子:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

要查看日志等级的完整列表，见*logging* 模块。

multiprocessing.dummy 模块

multiprocessing.dummy 复制了*multiprocessing* 的 API，不过是在*threading* 模块之上包装了一层。

17.2.3 编程指导

使用*multiprocessing* 时，应遵循一些指导原则和习惯用法。

所有启动方法

下面这些使用于所有启动方法。

避免共享状态

应该尽可能避免在进程间传递大量数据，越少越好。

最好坚持使用队列或者管道进行进程间通信，而不是底层的同步原语。

可序列化

保证所代理的方法的参数是可以序列化的。

代理的线程安全性

不要在多线程中同时使用一个代理对象，除非你用锁保护它。

（而在不同进程中使用 相同的代理对象从不会发生问题。）

使用 Join 避免僵尸进程

在 Unix 上，如果一个进程执行完成但是没有被 join，就会变成僵尸进程。一般来说，僵尸进程不会很多，因为每次新启动进程（或者*active_children()* 被调用）时，所有已执行完成且没有被 join 的进程都会自动被 join，而且对一个执行完的进程调用*Process.is_alive* 也会 join 这个进程。尽管如此，对自己启动的进程显式调用 join 依然是最佳实践。

继承优于序列化、反序列化

当使用 *spawn* 或者 *forkserver* 的启动方式时，*multiprocessing* 中的许多类型都必须是可序列化的，这样子进程才能使用它们。但是通常我们都应该避免使用管道和队列发送共享对象到另外一个进程，而是重新组织代码，对于其他进程创建出来的共享对象，让那些需要访问这些对象的子进程可以直接将这些对象从父进程继承过来。

避免杀死进程

听过*Process.terminate* 停止一个进程很容易导致这个进程正在使用的共享资源（如锁、信号量、管道和队列）损坏或者变得不可用，无法在其他进程中继续使用。

所以，最好只对那些从来不使用共享资源的进程调用*Process.terminate* 。

Join 使用队列的进程

记住，往队列放入数据的进程会一直等待直到所有缓存项被”feeder”线程传给底层管道。（子进程可以调用队列的*Queue.cancel_join_thread* 方法禁止这种行为）

这意味着，任何使用队列的时候，你都要确保在进程 join 之前，所有存放到队列中的项将会被其他进程、线程完全消费。否则不能保证这个写过队列的进程可以正常终止。记住非精灵进程会自动 join 。

下面是一个会导致死锁的例子:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

交换最后两行可以修复这个问题（或者直接删掉 `p.join()`）。

显式传递资源给子进程

在 Unix 上，使用 *fork* 方式启动的子进程可以使用父进程中全局创建的共享资源。

除了（部分原因）让代码兼容 Windows 以及其他的进程启动方式外，这种形式还保证了在子进程生命期这个对象是不会被父进程垃圾回收的。如果父进程中的某些对象被垃圾回收会导致资源释放，这就变得很重要。

所以对于实例：

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

应当重写成这样：

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

谨防将*sys.stdin*数据替换为“类似文件的对象”

multiprocessing 内部会无条件地这样调用：

```
os.close(sys.stdin.fileno())
```

在 `multiprocessing.Process._bootstrap()` 方法中——这会导致与”进程中的进程”相关的一些问题。这已经被修改成了：

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

它解决了进程相互冲突导致文件描述符错误的根本问题，但是对使用带缓冲的“文件类对象”替换*sys.stdin()* 作为输出的应用程序造成了潜在的危险。如果多个进程调用了此文件类对象的*close()* 方法，会导致相同的数据多次刷写到此对象，损坏数据。

如果你写入文件类对象并实现了自己的缓存，可以在每次追加缓存数据时记录当前进程 id，从而将其变成 fork 安全的，当发现进程 id 变化后舍弃之前的缓存，例如:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

需要更多信息，请查看 bpo-5155, bpo-5313 以及 bpo-5331

spawn 和 *forkserver* 启动方式

相对于 *fork* 启动方式，有一些额外的限制。

更依赖序列化

Process.__init__() 的所有参数都必须可序列化。同样的，当你继承*Process* 时，需要保证当调用*Process.start* 方法时，实例可以被序列化。

全局变量

记住，如果子进程中的代码尝试访问一个全局变量，它所看到的值可能和父进程中执行*Process.start* 那一刻的值不一样。

当全局变量知识模块级别的常量时，是不会有问题的。

安全导入主模块

确保主模块可以被新启动的 Python 解释器安全导入而不会引发什么副作用（比如又启动了一个子进程）

例如，使用 *spawn* 或 *forkserver* 启动方式执行下面的模块，会引发*RuntimeError* 异常而失败。

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

应该通过下面的方法使用 if __name__ == '__main__': ，从而保护程序”入口点”：

```
from multiprocessing import Process, freeze_support, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

（如果程序将正常运行而不是冻结，则可以省略 `freeze_support()` 行）

这允许新启动的 Python 解释器安全导入模块然后运行模块中的 `foo()` 函数。

如果主模块中创建了进程池或者管理器，这个规则也适用。

17.2.4 例子

创建和使用自定义管理器、代理的示例:

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)
```

(下页继续)

(续上页)

```
##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()
```

使用*Pool*:

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
```

(下页继续)

(续上页)

```
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
                [(plus, (i, 8)) for i in range(10)]

        results = [pool.apply_async(calculate, t) for t in TASKS]
        imap_it = pool.imap(calculatestar, TASKS)
        imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

        print('Ordered results using pool.apply_async():')
        for r in results:
            print('\t', r.get())
        print()

        print('Ordered results using pool.imap():')
        for x in imap_it:
            print('\t', x)
        print()

        print('Unordered results using pool.imap_unordered():')
```

(下页继续)

(续上页)

```
        for x in imap_unordered_it:
            print('\t', x)
        print()

        print('Ordered results using pool.map() --- will block till complete:')
        for x in pool.map(calculatestar, TASKS):
            print('\t', x)
        print()

        #
        # Test error handling
        #

        print('Testing error handling:')

        try:
            print(pool.apply(f, (5,)))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.apply()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(pool.map(f, list(range(10))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.map()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(list(pool.imap(f, list(range(10)))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from list(pool.imap())')
        else:
            raise AssertionError('expected ZeroDivisionError')

        it = pool.imap(f, list(range(10)))
        for i in range(10):
            try:
                x = next(it)
            except ZeroDivisionError:
                if i == 5:
                    pass
            except StopIteration:
                break
            else:
                if i == 5:
                    raise AssertionError('expected ZeroDivisionError')

        assert i == 9
        print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
        print()

        #
        # Testing timeouts
        #
```

(下页继续)

(续上页)

```
        print('Testing ApplyResult.get() with timeout:', end=' ')
        res = pool.apply_async(calculate, TASKS[0])
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()

        print('Testing IMapIterator.next() with timeout:', end=' ')
        it = pool.imap(calculatestar, TASKS)
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()


if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

一个演示如何使用队列来向一组工作进程提供任务并收集结果的例子：

```
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)
```

(下页继续)

(续上页)

```
#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('\t', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

17.3 concurrent 包

目前，此包中只有一个模块：

- *concurrent.futures* ——启动并行任务

17.4 concurrent.futures —启动并行任务

3.2 新版功能.

源码: Lib/concurrent/futures/thread.py 和 Lib/concurrent/futures/process.py

---

concurrent.futures 模块提供异步执行可调用对象高层接口。

异步执行可以由*ThreadPoolExecutor* 使用线程或由*ProcessPoolExecutor* 使用单独的进程来实现。两者都是实现抽像类*Executor* 定义的接口。

17.4.1 Executor 对象

class concurrent.futures.**Executor**

抽象类提供异步执行调用方法。要通过它的子类调用，而不是直接调用。

submit(*fn*, **args*, ***kwargs*)

调度可调用对象 *fn*，以 `fn(*args **kwargs)` 方式执行并返回*Future* 对像代表可调用对象的执行。：

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*func*, **iterables*, *timeout=None*, *chunksize=1*)

类似于*map(func, *iterables)* 函数，除了以下两点：

- *iterables* 是立即执行而不是延迟执行的；
- *func* 是异步执行的，对 *func* 的多个调用可以并发执行。

如果从原始调用到*Executor.map()* 经过 *timeout* 秒后，*__next__()* 已被调用且返回的结果还不可用，那么已返回的迭代器将触发*concurrent.futures.TimeoutError* 。*timeout* 可以是整数或浮点数。如果 *timeout* 没有指定或为 `None` ，则没有超时限制。

如果 *func* 调用引发一个异常，当从迭代器中取回它的值时这个异常将被引发。

使用*ProcessPoolExecutor* 时，这个方法会将 *iterables* 分割任务块并作为独立的任务并提交到执行池中。这些块的大概数量可以由 *chunksize* 指定正整数设置。对很长的迭代器来说，使用大的 *chunksize* 值比默认值 1 能显著地提高性能。*chunksize* 对*ThreadPoolExecutor* 没有效果。

在 3.5 版更改: 加入 *chunksize* 参数。

shutdown(*wait=True*)

当待执行的 future 对象完成执行后向执行者发送信号，它就会释放正在使用的任何资源。在关闭后调用*Executor.submit()* 和*Executor.map()* 将会引发*RuntimeError*。

如果 *wait* 为 `True` 则此方法只有在所有待执行的 future 对象完成执行且释放已分配的资源后才会返回。如果 *wait* 为 `False`，方法立即返回，所有待执行的 future 对象完成执行

后会释放已分配的资源。不管 *wait* 的值是什么，整个 Python 程序将等到所有待执行的 future 对象完成执行后才退出。

如果使用 `with` 语句，你就可以避免显式调用这个方法，它将会停止*Executor* (就好像*Executor.shutdown()* 调用时 *wait* 设为 `True` 一样等待):

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

17.4.2 ThreadPoolExecutor

ThreadPoolExecutor 是*Executor* 的子类，它使用线程池来异步执行调用。

当回调已关联了一个*Future* 然后再等待另一个*Future* 的结果时就会发产死锁情况。例如:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting on b.
    return 6


executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

与:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.**ThreadPoolExecutor**(*max_workers=None*, *thread_name_prefix=''*)

Executor 子类使用最多 *max_workers* 个线程的线程池来异步执行调用。

在 3.5 版更改: 如果 *max_workers* 为 `None` 或没有指定，将默认为机器处理器的个数，假如*ThreadPoolExecutor* 侧重于 I/O 操作而不是 CPU 运算，那么可以乘以 `5`，同时工作线程的数量可以比*ProcessPoolExecutor* 的数量高。

3.6 新版功能: 添加 *thread_name_prefix* 参数允许用户控制由线程池创建的*threading.Thread* 工作线程名称以方便调试。

ThreadPoolExecutor 例子

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://some-made-up-domain.com/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

17.4.3 ProcessPoolExecutor

ProcessPoolExecutor 是*Executor* 的子类，它使用进程池来实现异步执行调用。*ProcessPoolExecutor* 使用*multiprocessing* 回避*Global Interpreter Lock* 但也意味着只可以处理和返回可序列化的对象。

__main__ 模块必须可以被工作者子进程导入。这意味着*ProcessPoolExecutor* 不可以工作在交互式解释器中。

从可调用对象中调用*Executor* 或*Future* 的方法提交给*ProcessPoolExecutor* 会导致死锁。

class concurrent.futures.**ProcessPoolExecutor**(*max_workers=None*)

An *Executor* subclass that executes calls asynchronously using a pool of at most *max_workers* processes. If *max_workers* is None or not given, it will default to the number of processors on the machine. If *max_workers* is lower or equal to 0, then a *ValueError* will be raised.

在 3.3 版更改: 如果其中一个工作进程被突然终止，BrokenProcessPool 就会马上触发。可预计的行为没有定义，但执行器上的操作或它的 future 对象会被冻结或死锁。

ProcessPoolExecutor 例子

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

17.4.4 Future 对象

Future 类将可调用对象封装为异步执行。*Future* 实例由*Executor.submit()* 创建。

class concurrent.futures.**Future**

将可调用对象封装为异步执行。*Future* 实例由*Executor.submit()* 创建，除非测试，不应直接创建。

cancel()

Attempt to cancel the call. If the call is currently being executed and cannot be cancelled then the method will return `False`, otherwise the call will be cancelled and the method will return `True`.

cancelled()

如果调用成功取消返回 `True`。

running()

如果调用正在执行而且不能被取消那么返回 `True` 。

done()

如果调用已被取消或正常结束那么返回 `True`。

result(*timeout=None*)

返回调用返回的值。如果调用还没完成那么这个方法将等待 *timeout* 秒。如果在 *timeout* 秒内没有执行完成，*concurrent.futures.TimeoutError* 将会被触发。*timeout* 可以是整数或浮点数。如果 *timeout* 没有指定或为 `None`，那么等待时间就没有限制。

如果 futrue 在完成前被取消则*CancelledError* 将被触发。

如果调用引发了一个异常，这个方法也会引发同样的异常。

exception(*timeout=None*)
返回由调用引发的异常。如果调用还没完成那么这个方法将等待 *timeout* 秒。如果在 *timeout* 秒内没有执行完成，*concurrent.futures.TimeoutError* 将会被触发。*timeout* 可以是整数或浮点数。如果 *timeout* 没有指定或为 None，那么等待时间就没有限制。

如果 futrue 在完成前被取消则*CancelledError* 将被触发。

如果调用正常完成那么返回 None。

add_done_callback(*fn*)
附加可调用 *fn* 到 future 对象。当 future 对象被取消或完成运行时，将会调用 *fn*，而这个 future 对象将作为它唯一的参数。

加入的可调用对象总被属于添加它们的进程中的线程按加入的顺序调用。如果可调用对象引发一个*Exception* 子类，它会被记录下来并被忽略掉。如果可调用对象引发一个*BaseException* 子类，这个行为没有定义。

如果 future 对象已经完成或已取消，*fn* 会被立即调用。

下面这些*Future* 方法用于单元测试和*Executor* 实现。

set_running_or_notify_cancel()
这个方法只可以在执行关联*Future* 工作之前由*Executor* 实现调用或由单测试调用。

如果这个方法返回 False 那么*Future* 已被取消，即*Future.cancel()* 已被调用并返回 True 。等待*Future* 完成 (即通过*as_completed()* 或*wait()*) 的线程将被唤醒。

如果这个方法返回 True 那么*Future* 不会被取消并已将它变为正在运行状态，也就是说调用*Future.running()* 时将返回 *True*。

这个方法只可以被调用一次并且不能在调用*Future.set_result()* 或*Future.set_exception()* 之后再调用。

set_result(*result*)
设置将*Future* 关联工作的结果给 *result* 。

这个方法只可以由*Executor* 实现和单元测试使用。

set_exception(*exception*)
设置*Future* 关联工作的结果给*Exception* *exception* 。

这个方法只可以由*Executor* 实现和单元测试使用。

17.4.5 模块函数

concurrent.futures.**wait**(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)
Wait for the *Future* instances (possibly created by different *Executor* instances) given by *fs* to complete. Returns a named 2-tuple of sets. The first set, named done, contains the futures that completed (finished or were cancelled) before the wait completed. The second set, named not_done, contains uncompleted futures.

timeout 可以用来控制返回前最大的等待秒数。*timeout* 可以为 int 或 float 类型。如果 *timeout* 未指定或为 None ，则不限制等待时间。

return_when 指定此函数应在何时返回。它必须为以下常数之一:

常数	描述
FIRST_COMPLETED	函数将在任意可等待对象结束或取消时返回。
FIRST_EXCEPTION	函数将在任意可等待对象因引发异常而结束时返回。当没有引发任何异常时它就相当于 ALL_COMPLETED。
ALL_COMPLETED	函数将在所有可等待对象结束或取消时返回。

concurrent.futures.**as_completed**(*fs*, *timeout=None*)

Returns an iterator over the *Future* instances (possibly created by different *Executor* instances) given by *fs* that yields futures as they complete (finished or were cancelled). Any futures given by *fs* that are duplicated will be returned once. Any futures that completed before *as_completed()* is called will be yielded first. The returned iterator raises a *concurrent.futures.TimeoutError* if *__next__()* is called and the result isn't available after *timeout* seconds from the original call to *as_completed()*. *timeout* can be an int or float. If *timeout* is not specified or None, there is no limit to the wait time.

参见:

PEP 3148 –**future 对象 - 异步执行指令**。 该提案描述了 Python 标准库中包含的这个特性。

17.4.6 Exception 类

exception concurrent.futures.**CancelledError**

future 对象被取消时会触发。

exception concurrent.futures.**TimeoutError**

future 对象执行超出给定的超时数值时引发。

exception concurrent.futures.process.**BrokenProcessPool**

Derived from *RuntimeError*, this exception class is raised when one of the workers of a ProcessPoolExecutor has terminated in a non-clean fashion (for example, if it was killed from the outside).

3.3 新版功能.

17.5 subprocess —子进程管理

源代码: Lib/subprocess.py

---

subprocess 模块允许你生成新的进程，连接它们的输入、输出、错误管道，并且获取它们的返回码。此模块打算代替一些老旧的模块与功能：

```
os.system
os.spawn*
```

在下面的段落中，你可以找到关于*subprocess* 模块如何代替这些模块和功能的相关信息。

参见:

PEP 324 –提出 subprocess 模块的 PEP

17.5.1 使用 subprocess 模块

推荐的调用子进程的方式是在任何它支持的用例中使用*run()* 函数。对于更进阶的用例，也可以使用底层的*Popen* 接口。

run() 函数是在 Python 3.5 被添加的；如果你需要与旧版本保持兼容，查看较旧的高阶 *API* 段落。

subprocess.**run**(*args*, *, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*, *env=None*)

运行被 *arg* 描述的指令。等待指令完成，然后返回一个*CompletedProcess* 实例。

The arguments shown above are merely the most common ones, described below in 常用参数 (hence the use of keyword-only notation in the abbreviated signature). The full function signature is largely the same as that of the *Popen* constructor - apart from *timeout*, *input* and *check*, all the arguments to this function are passed through to that interface.

This does not capture stdout or stderr by default. To do so, pass *PIPE* for the *stdout* and/or *stderr* arguments.

timeout 参数将被传递给*Popen.communicate()*。如果发生超时，子进程将被杀死并等待。*TimeoutExpired* 异常将在子进程中断后被抛出。

The *input* argument is passed to *Popen.communicate()* and thus to the subprocess's stdin. If used it must be a byte sequence, or a string if *encoding* or *errors* is specified or *universal_newlines* is true. When used, the internal *Popen* object is automatically created with `stdin=PIPE`, and the *stdin* argument may not be used as well.

如果 *check* 设为 True, 并且进程以非零状态码退出, 一个*CalledProcessError* 异常将被抛出. 这个异常的属性将设置为参数, 退出码, 以及标准输出和标准错误, 如果被捕获到.

If *encoding* or *errors* are specified, or *universal_newlines* is true, file objects for stdin, stdout and stderr are opened in text mode using the specified *encoding* and *errors* or the *io.TextIOWrapper* default. Otherwise, file objects are opened in binary mode.

如果 *env* 不是 `None`, 它必须是一个字典, 为新的进程设置环境变量; 它用于替换继承的当前进程的环境的默认行为. 它将直接被传递给*Popen*.

示例:

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], stdout=subprocess.PIPE)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n')
```

3.5 新版功能.

在 3.6 版更改: 添加了 *encoding* 和 *errors* 形参.

class subprocess.**CompletedProcess**

run() 的返回值, 代表一个进程已经结束.

args

被用作启动进程的参数. 可能是一个列表或字符串.

returncode

子进程的退出状态码. 通常来说, 一个为 0 的退出码表示进程运行正常.

一个负值 -N 表示子进程被信号 N 中断 (仅 POSIX).

stdout
Captured stdout from the child process. A bytes sequence, or a string if *run()* was called with an encoding or errors. None if stdout was not captured.

如果你通过 stderr=subprocess.STDOUT 运行进程，标准输入和标准错误将被组合在这个属性中，并且*stderr* 将为 None。

stderr
Captured stderr from the child process. A bytes sequence, or a string if *run()* was called with an encoding or errors. None if stderr was not captured.

check_returncode()
如果*returncode* 非零, 抛出*CalledProcessError*.

3.5 新版功能.

subprocess.**DEVNULL**
可被*Popen* 的 *stdin*, *stdout* 或者 *stderr* 参数使用的特殊值, 表示使用特殊文件*os.devnull*.

3.3 新版功能.

subprocess.**PIPE**
可被*Popen* 的 *stdin*, *stdout* 或者 *stderr* 参数使用的特殊值, 表示打开标准流的管道. 常用于*Popen.communicate()*.

subprocess.**STDOUT**
可被*Popen* 的 *stdin* ， *stdout* 或者 *stderr* 参数使用的特殊值，表示标准错误与标准输出使用同一句柄。

exception subprocess.**SubprocessError**
此模块的其他异常的基类。

3.3 新版功能.

exception subprocess.**TimeoutExpired**
SubprocessError 的子类，等待子进程的过程中发生超时时被抛出。

cmd
用于创建子进程的指令。

timeout
超时秒数。

output
子进程的输出，如果被*run()* 或*check_output()* 捕获。否则为 None。

stdout
对 output 的别名，对应的有*stderr*。

stderr
子进程的标准错误输出，如果被*run()* 捕获。否则为 None。

3.3 新版功能.

在 3.5 版更改: 添加了 *stdout* 和 *stderr* 属性。

exception subprocess.**CalledProcessError**
SubprocessError 的子类，当一个被*check_call()* 或*check_output()* 函数运行的子进程返回了非零退出码时被抛出。

returncode
子进程的退出状态。如果程序由一个信号终止，这将会被设为一个负的信号码。

cmd
用于创建子进程的指令。

output
子进程的输出，如果被*run()* 或*check_output()* 捕获。否则为 None。

stdout
对 output 的别名，对应的有*stderr*。

stderr
子进程的标准错误输出，如果被*run()* 捕获。否则为 None。

在 3.5 版更改: 添加了 *stdout* 和 *stderr* 属性。

常用参数

为了支持丰富的使用案例，*Popen* 的构造函数（以及方便的函数）接受大量可选的参数。对于大多数典型的用例，许多参数可以被安全地留以它们的默认值。通常需要的参数有：

args 被所有调用需要，应当为一个字符串，或者一个程序参数序列。提供一个参数序列通常更好，它可以更小心地使用参数中的转义字符以及引用（例如允许文件名中的空格）。如果传递一个简单的字符串，则 *shell* 参数必须为*True* （见下文）或者该字符串中将被运行的程序名必须用简单的命名而不指定任何参数。

stdin，*stdout* 和 *stderr* 分别指定了执行的程序的标准输入、输出和标准错误文件句柄。合法的值有*PIPE* 、*DEVNULL* 、一个现存的文件描述符（一个正整数）、一个现存的文件对象以及 None。*PIPE* 表示应该新建一个对子进程的管道。*DEVNULL* 表示使用特殊的文件*os.devnull*。当使用默认设置 None 时，将不会进行重定向，子进程的文件流将继承自父进程。另外，*stderr* 可能为*STDOUT*，表示来自于子进程的标准错误数据应该被 *stdout* 相同的句柄捕获。

If *encoding* or *errors* are specified, or *universal_newlines* is true, the file objects *stdin*, *stdout* and *stderr* will be opened in text mode using the *encoding* and *errors* specified in the call or the defaults for *io.TextIOWrapper*.

当构造函数的 *newline* 参数为 None 时。对于 *stdin*，输入的换行符 '\n' 将被转换为默认的换行符*os.linesep*。对于 *stdout* 和 *stderr*，所有输出的换行符都被转换为 '\n'。更多信息，查看*io.TextIOWrapper* 类的文档。

如果文本模式未被使用，*stdin*，*stdout* 和 *stderr* 将会以二进制流模式打开。没有编码与换行符转换发生。

3.6 新版功能: 添加了 *encoding* 和 *errors* 形参。

注解: 文件对象*Popen.stdin* 、*Popen.stdout* 和*Popen.stderr* 的换行符属性不会被*Popen.communicate()* 方法更新。

如果 *shell* 设为 True,，则使用 shell 执行指定的指令。如果您主要使用 Python 增强的控制流（它比大多数系统 shell 提供的强大），并且仍然希望方便地使用其他 shell 功能，如 shell 管道、文件通配符、环境变量展开以及 ~ 展开到用户家目录，这将非常有用。但是，注意 Python 自己也实现了许多类似 shell 的特性（例如*glob*, *fnmatch*, *os.walk()*, *os.path.expandvars()*, *os.path.expanduser()* 和*shutil*）。

在 3.3 版更改: 当 *universal_newline* 被设为 True，则类使用*locale.getpreferredencoding(False)* 编码来代替 locale.getpreferredencoding()。关于它们的区别的更多信息，见*io.TextIOWrapper*。

注解： 在使用 shell=True 之前，请阅读*Security Considerations* 段落。

这些选项以及所有其他选项在*Popen* 构造函数文档中有更详细的描述。

Popen 构造函数

此模块的底层的进程创建与管理由*Popen* 类处理。它提供了很大的灵活性，因此开发者能够处理未被便利函数覆盖的不常见用例。

class subprocess.**Popen**(*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=False*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, *, *encoding=None*, *errors=None*)

在一个新的进程中执行子程序。在 POSIX，此类使用类似于*os.execvp()* 的行为来执行子程序。在 Windows，此类使用了 Windows CreateProcess() 函数。*Popen* 的参数如下：

args 应当是一个程序的参数列表或者一个简单的字符串。默认情况下，如果 *args* 是一个序列，将运行的程序是此序列的第一项。如果 *args* 是一个字符串，解释是平台相关的，如下所述。有关默认行为的其他差异，见 *shell* 和 *executable* 参数。除非另有说明，推荐将 *args* 作为序列传递。

在 POSIX，如果 *args* 是一个字符串，此字符串被作为将被执行的程序的命名或路径解释。但是，只有在不传递任何参数给程序的情况下才能这么做。

注解： *shlex.split()* can be useful when determining the correct tokenization for *args*, especially in complex cases:

```
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', "echo '
↪$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```

特别注意，由 shell 中的空格分隔的选项（例如 *-input*）和参数（例如 *eggs.txt* ）位于分开的列表元素中，而在需要时使用引号或反斜杠转义的参数在 shell （例如包含空格的文件名或上面显示的 *echo* 命令）是单独的列表元素。

在 Windows，如果 *args* 是一个序列，他将通过一个在*在 Windows 上将参数列表转换为一个字符串* 描述的方式被转换为一个字符串。这是因为底层的 CreateProcess() 只处理字符串。

参数 *shell* （默认为 False）指定是否使用 shell 执行程序。如果 *shell* 为 True，更推荐将 *args* 作为字符串传递而非序列。

在 POSIX，当 shell=True，shell 默认为 /bin/sh。如果 *args* 是一个字符串，此字符串指定将通过 shell 执行的命令。这意味着字符串的格式必须和在命令提示符中所输入的完全相同。这包括，例如，引号和反斜杠转义包含空格的文件名。如果 *args* 是一个序列，第一项指定了命令，另外的项目将作为传递给 shell （而非命令）的参数对待。也就是说，*Popen* 等同于：

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

在 Windows，使用 shell=True，环境变量 COMSPEC 指定了默认 shell。在 Windows 你唯一需要指定 shell=True 的情况是你想要执行内置在 shell 中的命令（例如 **dir** 或者 **copy**）。在运行一个批处理文件或者基于控制台的可执行文件时，不需要 shell=True。

注解： 在使用 `shell=True` 之前，请阅读*Security Considerations* 段落。

bufsize 将在`open()` 函数创建了 stdin/stdout/stderr 管道文件对象时作为对应的参数供应:

- `0` 表示不使用缓冲区（读取与写入是一个系统调用并且可以返回短内容）
- `1` 表示行缓冲（只有 `universal_newlines=True` 时才有用，例如，在文本模式中）
- 任何其他正值表示使用一个约为对应大小的缓冲区
- 负的 *bufsize* （默认）表示使用系统默认的 io.DEFAULT_BUFFER_SIZE。

在 3.3.1 版更改: *bufsize* 现在默认为 -1 来启用缓冲，以符合大多数代码所期望的行为。在 Python 3.2.4 和 3.3.1 之前的版本中，它错误地将默认值设为了为 `0`，这是无缓冲的并且允许短读取。这是无意的，并且与大多数代码所期望的 Python 2 的行为不一致。

executable 参数指定一个要执行的替换程序。这很少需要。当 `shell=True`，*executable* 替换 *args* 指定运行的程序。但是，原始的 *args* 仍然被传递给程序。大多数程序将被 *args* 指定的程序作为命令名对待，这可以与实际运行的程序不同。在 POSIX，*args* 名作为实际调用程序中可执行文件的显示名称，例如 **ps**。如果 `shell=True`，在 POSIX，*executable* 参数指定用于替换默认 shell `/bin/sh` 的 shell。

stdin, *stdout* 和 *stderr* 分别指定被运行的程序的标准输入、输出和标准错误的文件句柄。合法的值有`PIPE` ，`DEVNULL` ，一个存在的文件描述符（一个正整数），一个存在的`文件对象` 以及 `None`。`PIPE` 表示应创建一个新的对子进程的管道。`DEVNULL` 表示使用特殊的`os.devnull` 文件。使用默认的 `None`，则不进行成定向；子进程的文件流将继承自父进程。另外，*stderr* 可设为`STDOUT`，表示应用程序的标准错误数据应和标准输出一同捕获。

如果 *preexec_fn* 被设为一个可调用对象，此对象将在子进程刚创建时被调用。（仅 POSIX）

警告： *preexec_fn* 形参在应用程序中存在多线程时是不安全的。子进程在调用前可能死锁。如果你必须使用它，保持警惕！最小化你调用的库的数量。

注解： 如果你需要修改子进程环境，使用 *env* 形参而非在 *preexec_fn* 中进行。*start_new_session* 形参可以代替之前常用的 *preexec_fn* 来在子进程中调用 os.setsid()。

If *close_fds* is true, all file descriptors except `0`, `1` and `2` will be closed before the child process is executed. (POSIX only). The default varies by platform: Always true on POSIX. On Windows it is true when *stdin*/*stdout*/*stderr* are `None`, false otherwise. On Windows, if *close_fds* is true then no handles will be inherited by the child process. Note that on Windows, you cannot set *close_fds* to true and also redirect the standard handles by setting *stdin*, *stdout* or *stderr*.

在 3.2 版更改: *close_fds* 的默认值已经从`False` 修改为上述值。

pass_fds 是一个可选的在父子进程间保持打开的文件描述符序列。提供任何 *pass_fds* 将强制 *close_fds* 为`True`。（仅 POSIX）

3.2 新版功能: 加入了 *pass_fds* 形参。

如果 *cwd* 不为 `None`，此函数在执行子进程前改变当前工作目录为 *cwd*。*cwd* 可以为一个`str` 和*path-like* 对象。特别要注意，当可执行文件的路径为相对路径时，此函数按相对于 *cwd* 的路径来寻找 *executable* (或者 *args* 的第一项)。

在 3.6 版更改: *cwd* 形参接受一个*path-like object*。

如果 *restore_signals* 为 true（默认值），则 Python 设置为 SIG_IGN 的所有信号将在 exec 之前的子进程中恢复为 SIG_DFL。目前，这包括 SIGPIPE ， SIGXFZ 和 SIGXFSZ 信号。（仅 POSIX）

在 3.2 版更改: *restore_signals* 被加入。

如果 *start_new_session* 为 true，则 setsid() 系统调用将在子进程执行之前被执行。（仅 POSIX）

在 3.2 版更改: *start_new_session* 被添加。

如果 *env* 不为 `None`，则必须为一个为新进程定义了环境变量的字典；这些用于替换继承的当前进程环境的默认行为。

注解： 如果指定，*env* 必须提供所有被子进程需求的变量。在 Windows，为了运行一个 side-by-side assembly ，指定的 *env* **必须**包含一个有效的 `SystemRoot`。

If *encoding* or *errors* are specified, the file objects *stdin*, *stdout* and *stderr* are opened in text mode with the specified encoding and *errors*, as described above in 常用参数. If *universal_newlines* is `True`, they are opened in text mode with default encoding. Otherwise, they are opened as binary streams.

3.6 新版功能: *encoding* 和 *errors* 被添加。

If given, *startupinfo* will be a *STARTUPINFO* object, which is passed to the underlying `CreateProcess` function. *creationflags*, if given, can be *CREATE_NEW_CONSOLE* or *CREATE_NEW_PROCESS_GROUP*. (Windows only)

Popen 对象支持通过 `with` 语句作为上下文管理器，在退出时关闭文件描述符并等待进程:

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

在 3.2 版更改: 添加了上下文管理器支持。

在 3.6 版更改: 现在，如果 Popen 析构时子进程仍然在运行，则析构器会发送一个*ResourceWarning* 警告。

异常

Exceptions raised in the child process, before the new program has started to execute, will be re-raised in the parent. Additionally, the exception object will have one extra attribute called `child_traceback`, which is a string containing traceback information from the child's point of view.

最常见的被抛出异常是*OSError*。例如，当尝试执行一个不存在的文件时就会发生。应用程序需要为*OSError* 异常做好准备。

如果*Popen* 调用时有无效的参数，则一个*ValueError* 将被抛出。

check_call() 与*check_output()* 在调用的进程返回非零退出码时将抛出*CalledProcessError*。

所有接受 *timeout* 形参的函数与方法，例如*call()* 和*Popen.communicate()* 将会在进程退出前超时到期时抛出*TimeoutExpired*。

此模块中定义的异常都继承自*SubprocessError*。

3.3 新版功能: 基类*SubprocessError* 被添加。

17.5.2 安全考量

不像一些其他的 popen 功能，此实现绝不会隐式调用一个系统 shell。这意味着任何字符，包括 shell 元字符，可以安全地被传递给子进程。如果 shell 被明确地调用，通过 `shell=True` 设置，则确保所有空白字符和元字符被恰当地包裹在引号内以避免 shell 注入 漏洞就由应用程序负责了。

当使用 `shell=True`，*shlex.quote()* 函数可以作为在将被用于构造 shell 指令的字符串中转义空白字符以及 shell 元字符的方案。

17.5.3 Popen 对象

Popen 类的实例拥有以下方法：

Popen.**poll**()
检查子进程是否已被终止。设置并返回*returncode* 属性。否则返回 None。

Popen.**wait**(*timeout=None*)
等待子进程被终止。设置并返回*returncode* 属性。

如果进程在 *timeout* 秒后未中断，抛出一个*TimeoutExpired* 异常，可以安全地捕获此异常并重新等待。

注解： 当 `stdout=PIPE` 或者 `stderr=PIPE` 并且子进程产生了足以阻塞 OS 管道缓冲区接收更多数据的输出到管道时，将会发生死锁。当使用管道时用*Popen.communicate()* 来规避它。

注解： 此函数使用了一个 busy loop （非阻塞调用以及短睡眠）实现。使用*asyncio* 模块进行异步等待：参阅*asyncio.create_subprocess_exec*。

在 3.3 版更改: *timeout* 被添加

3.4 版后已移除: Do not use the *endtime* parameter. It is was unintentionally exposed in 3.3 but was left undocumented as it was intended to be private for internal use. Use *timeout* instead.

Popen.**communicate**(*input=None*, *timeout=None*)
与进程交互：向 stdin 传输数据。从 stdout 和 stderr 读取数据，直到文件结束符。等待进程终止。可选的 *input* 参数应当未被传输给子进程的数据，如果没有数据应被传输给子进程则为 None。如果流以文本模式打开，*input* 必须为字符串。否则，它必须为字节。

communicate() 返回一个 `(stdout_data, stderr_data)` 元组。如果文件以文本模式打开则为字符串；否则字节。

注意如果你想要向进程的 stdin 传输数据，你需要通过 `stdin=PIPE` 创建此 Popen 对象。类似的，要从结果元组获取任何非 None 值，你同样需要设置 `stdout=PIPE` 或者 `stderr=PIPE`。

如果进程在 *timeout* 秒后未终止，一个*TimeoutExpired* 异常将被抛出。捕获此异常并重新等待将不会丢失任何输出。

如果超时到期，子进程不会被杀死，所以为了正确清理一个行为良好的应用程序应该杀死子进程并完成通讯。

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

注解: 内存里数据读取是缓冲的，所以如果数据尺寸过大或无限，不要使用此方法。

在 3.3 版更改: *timeout* 被添加

Popen.**send_signal**(*signal*)
将信号 *signal* 发送给子进程。

注解: 在 Windows，SIGTERM 是一个*terminate()* 的别名。CTRL_C_EVENT 和 CTRL_BREAK_EVENT 可以被发送给以包含 CREATE_NEW_PROCESS 的 *creationflags* 形参启动的进程。

Popen.**terminate**()
Stop the child. On Posix OSs the method sends SIGTERM to the child. On Windows the Win32 API function TerminateProcess() is called to stop the child.

Popen.**kill**()
Kills the child. On Posix OSs the function sends SIGKILL to the child. On Windows *kill()* is an alias for *terminate()*.

以下属性也是可用的:

Popen.**args**
args 参数传递给*Popen* —一个程序参数的序列或者一个简单字符串。

3.3 新版功能.

Popen.**stdin**
如果 *stdin* 参数为*PIPE*，此属性是一个类似*open()* 返回的可写的流对象。如果 *encoding* 或 *errors* 参数被指定或者 *universal_newlines* 参数为 True，则此流是一个文本流，否则是字节流。如果 *stdin* 参数非*PIPE*，此属性为 None。

Popen.**stdout**
如果 *stdout* 参数是*PIPE*，此属性是一个类似*open()* 返回的可读流。从流中读取子进程提供的输出。如果 *encoding* 或 *errors* 参数被指定或者 *universal_newlines* 参数为 True，此流为文本流，否则为字节流。如果 *stdout* 参数非*PIPE*，此属性为 None。

Popen.**stderr**
如果 *stderr* 参数是*PIPE*，此属性是一个类似*open()* 返回的可读流。从流中读取子进程提供的输出。如果 *encoding* 或 *errors* 参数被指定或者 *universal_newlines* 参数为 True，此流为文本流，否则为字节流。如果 *stderr* 参数非*PIPE*，此属性为 None。

警告: 使用*communicate()* 而非*.stdin.write*，*.stdout.read* 或者*.stderr.read* 来避免由于任意其他 OS 管道缓冲区被子进程填满阻塞而导致的死锁。

Popen.**pid**
子进程的进程号。

注意如果你设置了 *shell* 参数为 True，则这是生成的子 shell 的进程号。

Popen.**returncode**
此进程的退出码，由*poll()* 和*wait()* 设置（以及直接由*communicate()* 设置）。一个 None 值表示此进程仍未结束。

一个负值 -N 表示子进程被信号 N 中断 (仅 POSIX).

17.5.4 Windows Popen 助手

STARTUPINFO 类和以下常数仅在 Windows 有效。

class subprocess.**STARTUPINFO**
Partial support of the Windows STARTUPINFO structure is used for *Popen* creation.

dwFlags
一个位字段，用于确定进程在创建窗口时是否使用某些*STARTUPINFO* 属性。

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

hStdInput
如果*dwFlags* 被指定为*STARTF_USESTDHANDLES*，则此属性是进程的标准输入句柄，如果*STARTF_USESTDHANDLES* 未指定，则默认的标准输入是键盘缓冲区。

hStdOutput
如果*dwFlags* 被指定为*STARTF_USESTDHANDLES*，则此属性是进程的标准输出句柄。除此之外，此此属性将被忽略并且默认标准输出是控制台窗口缓冲区。

hStdError
如果*dwFlags* 被指定为*STARTF_USESTDHANDLES*，则此属性是进程的标准错误句柄。除此之外，此属性将被忽略并且默认标准错误为控制台窗口的缓冲区。

wShowWindow
如果*dwFlags* 指定了*STARTF_USESHOWWINDOW*，此属性可为能被指定为函数 ShowWindow 的 nCmdShow 的形参的任意值，除了 SW_SHOWDEFAULT。如此之外，此属性被忽略。

SW_HIDE 被提供给此属性。它在*Popen* 由 shell=True 调用时使用。

Constants

subprocess 模块曝出以下常数。

subprocess.**STD_INPUT_HANDLE**
标准输入设备，这是控制台输入缓冲区 CONIN$。

subprocess.**STD_OUTPUT_HANDLE**
标准输出设备。最初，这是活动控制台屏幕缓冲区 CONOUT$。

subprocess.**STD_ERROR_HANDLE**
标准错误设备。最初，这是活动控制台屏幕缓冲区 CONOUT$。

subprocess.**SW_HIDE**
隐藏窗口。另一个窗口将被激活。

subprocess.**STARTF_USESTDHANDLES**
指明*STARTUPINFO.hStdInput*, *STARTUPINFO.hStdOutput* 和*STARTUPINFO.hStdError* 属性包含额外的信息。

subprocess.**STARTF_USESHOWWINDOW**
指明*STARTUPINFO.wShowWindow* 属性包含额外的信息。

subprocess.**CREATE_NEW_CONSOLE**
新的进程将有新的控制台，而不是继承父进程的（默认）控制台。

subprocess.**CREATE_NEW_PROCESS_GROUP**
Popen creationflags 形参用于指明将创建一个新的进程组。这个旗标对于在子进程上使用*os.kill()* 来说是必须的。

如果指定了*CREATE_NEW_CONSOLE* 则这个旗标会被忽略。

17.5.5 较旧的高阶 API

在 Python 3.5 之前，这三个函数组成了 subprocess 的高阶 API。现在你可以在许多情况下使用*run()*，但有大量现在代码仍会调用这些函数。

subprocess.**call**(*args*, ***, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*)
运行由 *args* 所描述的命令。等待命令完成，然后返回*returncode* 属性。

这相当于:

```
run(...).returncode
```

(except that the *input* and *check* parameters are not supported)

The arguments shown above are merely the most common ones. The full function signature is largely the same as that of the *Popen* constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

注解: 请不要在此函数中使用 `stdout=PIPE` 或 `stderr=PIPE`。如果子进程向管道生成了足以填满 OS 管理缓冲区的输出而管道还未被读取时它将会阻塞。

在 3.3 版更改: *timeout* 被添加

subprocess.**check_call**(*args*, ***, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*)
附带参数运行命令。等待命令完成。如果返回码为零则正常返回，否则引发*CalledProcessError*。*CalledProcessError* 对象将在*returncode* 属性中保存返回码。

这相当于:

```
run(..., check=True)
```

(except that the *input* parameter is not supported)

The arguments shown above are merely the most common ones. The full function signature is largely the same as that of the *Popen* constructor - this function passes all supplied arguments other than *timeout* directly through to that interface.

注解: 请不要在此函数中使用 `stdout=PIPE` 或 `stderr=PIPE`。如果子进程向管道生成了足以填满 OS 管理缓冲区的输出而管道还未被读取时它将会阻塞。

在 3.3 版更改: *timeout* 被添加

subprocess.**check_output**(*args*, ***, *stdin=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *errors=None*, *universal_newlines=False*, *timeout=None*)
附带参数运行命令并返回其输出。

如果返回码非零则会引发*CalledProcessError*。*CalledProcessError* 对象将在*returncode* 属性中保存返回码并在*output* 属性中保存所有输出。

这相当于:

```
run(..., check=True, stdout=PIPE).stdout
```

The arguments shown above are merely the most common ones. The full function signature is largely the same as that of *run()* - most arguments are passed directly through to that interface. However, explicitly passing `input=None` to inherit the parent's standard input file handle is not supported.

默认情况下，此函数将把数据返回为已编码的字节串。输出数据的实际编码格式将取决于发起调用的命令，因此解码为文本的操作往往需要在应用程序层级上进行处理。

This behaviour may be overridden by setting *universal_newlines* to `True` as described above in 常用参数.

要在结果中同时捕获标准错误，请使用 `stderr=subprocess.STDOUT`:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

3.1 新版功能.

在 3.3 版更改: *timeout* 被添加

在 3.4 版更改: 增加了对 *input* 关键字参数的支持。

在 3.6 版更改: 增加了 *encoding* 和 *errors*。详情参见*run()*。

17.5.6 使用 subprocess 模块替换旧函数

在这一节中，”a 改为 b” 意味着 b 可以被用作 a 的替代。

注解: 在这一节中的所有 “a” 函数会在找不到被执行的程序时（差不多）静默地失败；” b” 替代则会改为引发*OSError*。

此外，在使用*check_output()* 时如果替代函数所请求的操作产生了非零返回值则将失败并引发*CalledProcessError*。操作的输出仍能以所引发异常的*output* 属性的方式被访问。

在下列例子中，我们假定相关的函数都已从*subprocess* 模块中导入了。

Replacing /bin/sh shell backquote

```
output=`mycmd myarg`
```

改为:

```
output = check_output(["mycmd", "myarg"])
```

替代 shell 管道

```
output=`dmesg | grep hda`
```

改为:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

The p1.stdout.close() call after starting the p2 is important in order for p1 to receive a SIGPIPE if p2 exits before p1.

另外，对于受信任的输入，shell 本身的管道支持仍然可被直接使用:

```
output=`dmesg | grep hda`
```

改为:

```
output=check_output("dmesg | grep hda", shell=True)
```

替代 os.system()

```
sts = os.system("mycmd" + " myarg")
# becomes
sts = call("mycmd" + " myarg", shell=True)
```

注释:

- 通过 shell 来调用程序通常是不必要的。

一个更现实的例子如下所示:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

替代 os.spawn 函数族

P_NOWAIT 示例:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT 示例:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Vector 示例:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Environment 示例:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

替代 os.popen(), os.popen2(), os.popen3()

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

返回码以如下方式处理转写:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

来自 popen2 模块的替代函数

注解: 如果 popen2 函数的 cmd 参数是一个字符串，命令会通过 /bin/sh 来执行。如果是一个列表，命令会被直接执行。

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

`popen2.Popen3` 和 `popen2.Popen4` 基本上类似于*subprocess.Popen*，不同之处在于:

- *Popen* 如果执行失败会引发一个异常。
- the *capturestderr* argument is replaced with the *stderr* argument.
- 必须指定 `stdin=PIPE` 和 `stdout=PIPE`。
- popen2 默认会关闭所有文件描述符，但对于*Popen* 你必须指明 `close_fds=True` 以才能在所有平台或较旧的 Python 版本中确保此行为。

17.5.7 旧式的 Shell 发起函数

此模块还提供了以下来自 2.x `commands` 模块的旧版函数。这些操作会隐式地发起调用系统 shell 并且上文所描述的有关安全与异常处理一致性保证都不适用于这些函数。

`subprocess.`**`getstatusoutput`**`(`*cmd*`)`

返回在 shell 中执行 *cmd* 产生的 `(exitcode, output)`。

在 shell 中以 `Popen.check_output()` 执行字符串 *cmd* 并返回一个 2 元组 `(exitcode, output)`。会使用当前区域设置的编码格式；请参阅*常用参数* 中的说明来了解详情。

末尾的一个换行符会从输出中被去除。命令的退出码可被解读为子进程的返回码。例如:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

Availability: POSIX & Windows

在 3.3.4 版更改: 添加了 Windows 支持。

此函数现在返回 (exitcode, output) 而不是像 Python 3.3.3 及更早的版本那样返回 (status, output)。exitcode 的值与*returncode* 相同。

subprocess.**getoutput**(*cmd*)
返回在 shell 中执行 *cmd* 产生的输出（stdout 和 stderr）。

类似于*getstatusoutput()*，但退出码会被忽略并且返回值为包含命令输出的字符串。例如:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

Availability: POSIX & Windows

在 3.3.4 版更改: 添加了 Windows 支持

17.5.8 注释

在 Windows 上将参数列表转换为一个字符串

在 Windows 上，*args* 序列会被转换为可使用以下规则来解析的字符串（对应于 MS C 运行时所使用的规则）:

1. 参数以空白符分隔，即空格符或制表符。
2. 用双引号标示的字符串会被解读为单个参数，而不再考虑其中的空白符。一个参数可以嵌套用引号标示的字符串。
3. 带有一个反斜杠前缀的双引号会被解读为双引号字面值。
4. 反斜杠会按字面值解读，除非它是作为双引号的前缀。
5. 如果反斜杠被作为双引号的前缀，则每个反斜杠对会被解读为一个反斜杠字面值。如果反斜杠数量为奇数，则最后一个反斜杠会如规则 3 所描述的那样转义下一个双引号。

参见:

shlex 此模块提供了用于解析和转义命令行的函数。

17.6 sched —事件调度器

源码: Lib/sched.py

---

sched 模块定义了一个实现通用事件调度程序的类:

class sched.**scheduler**(*timefunc=time.monotonic*, *delayfunc=time.sleep*)
The *scheduler* class defines a generic interface to scheduling events. It needs two functions to actually deal with the "outside world" —*timefunc* should be callable without arguments, and return a number (the "time", in any units whatsoever). If time.monotonic is not available, the *timefunc* default is time.time instead. The *delayfunc* function should be callable with one argument, compatible with the output of *timefunc*, and should delay that many time units. *delayfunc* will also be called with the argument `0` after each event is run to allow other threads an opportunity to run in multi-threaded applications.

在 3.3 版更改: *timefunc* 和 *delayfunc* 参数是可选的。

在 3.3 版更改: *scheduler* 类可以安全的在多线程环境中使用。

示例:

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.run()
...     print(time.time())
...
>>> print_some_times()
930343690.257
From print_time 930343695.274 positional
From print_time 930343695.275 keyword
From print_time 930343700.273 default
930343700.276
```

17.6.1 调度器对象

scheduler 实例拥有以下方法和属性：

scheduler.**enterabs**(*time*, *priority*, *action*, *argument=()*, *kwargs={}*)
安排一个新事件。*time* 参数应该有一个数字类型兼容的返回值，与传递给构造函数的 *timefunc* 函数的返回值兼容。计划在相同 *time* 的事件将按其 *priority* 的顺序执行。数字越小表示优先级越高。

执行事件意为执行 `action(*argument, **kwargs)`。*argument* 是包含有 *action* 的位置参数的序列。*kwargs* 是包含 *action* 的关键字参数的字典。

返回值是一个事件，可用于以后取消事件（参见*cancel()*）。

在 3.3 版更改: *argument* 参数是可选的。

3.3 新版功能: 添加了 *kwargs* 形参。

scheduler.**enter**(*delay*, *priority*, *action*, *argument=()*, *kwargs={}*)
安排延后 *delay* 时间单位的事件。除了相对时间，其他参数、效果和返回值与*enterabs()* 的相同。

在 3.3 版更改: *argument* 参数是可选的。

3.3 新版功能: 添加了 *kwargs* 形参。

scheduler.**cancel**(*event*)
从队列中删除事件。如果 *event* 不是当前队列中的事件，则此方法将引发*ValueError*。

scheduler.**empty**()
Return true if the event queue is empty.

scheduler.**run**(*blocking=True*)
运行所有预定事件。此方法将等待（使用传递给构造函数的 `delayfunc()` 函数）进行下一个事件，然后执行它，依此类推，直到没有更多的计划事件。

如果 *blocking* 为 false，则执行由于最快到期（如果有）的预定事件，然后在调度程序中返回下一个预定调用的截止时间（如果有）。

action 或 *delayfunc* 都可以引发异常。在任何一种情况下，调度程序都将保持一致状态并传播异常。如果 *action* 引发异常，则在将来调用*run()* 时不会尝试该事件。

如果一系列事件的运行时间比下一个事件之前的可用时间长，那么调度程序将完全落后。不会发生任何事件；调用代码负责取消不再相关的事件。

3.3 新版功能: 添加了 *blocking* 形参。

scheduler.**queue**
只读属性按照将要运行的顺序返回即将发生的事件列表。每个事件都显示为*named tuple* ，包含以下字段：time、priority、action、argument、kwargs。

17.7 queue —一个同步的队列类

源代码: Lib/queue.py

---

queue 模块实现多生产者，多消费者队列。当信息必须安全的在多线程之间交换时，它在线程编程中是特别有用的。此模块中的*Queue* 类实现了所有锁定需求的语义。它依赖于 Python 支持的线程可用性；请参阅*threading* 模块。

模块实现了三种类型的队列，它们的区别仅仅是条目取回的顺序。在 FIFO 队列中，先添加的任务先取回。在 LIFO 队列中，最近被添加的条目先取回 (操作类似一个堆栈)。优先级队列中，条目将保持排序 (使用*heapq* 模块) 并且最小值的条目第一个返回。

Internally, the module uses locks to temporarily block competing threads; however, it is not designed to handle reentrancy within a thread.

queue 模块定义了下列类和异常：

class queue.**Queue**(*maxsize=0*)
Constructor for a FIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class queue.**LifoQueue**(*maxsize=0*)
LIFO 队列构造函数。*maxsize* 是个整数，用于设置可以放入队列中的项目数的上限。当达到这个大小的时候，插入操作将阻塞至队列中的项目被消费掉。如果 *maxsize* 小于等于零，队列尺寸为无限大。

class queue.**PriorityQueue**(*maxsize=0*)
优先级队列构造函数。*maxsize* 是个整数，用于设置可以放入队列中的项目数的上限。当达到这个大小的时候，插入操作将阻塞至队列中的项目被消费掉。如果 *maxsize* 小于等于零，队列尺寸为无限大。

最小值先被取出 (最小值条目是由 `sorted(list(entries))[0]` 返回的条目)。条目的典型模式是一个以下形式的元组：`(priority_number, data)` 。

exception queue.**Empty**
对空的*Queue* 对象，调用非阻塞的*get()* (or *get_nowait()*) 时，引发的异常。

exception queue.**Full**
对满的*Queue* 对象，调用非阻塞的*put()* (or *put_nowait()*) 时，引发的异常。

17.7.1 Queue 对象

队列对象 (*Queue*, *LifoQueue*, 或者*PriorityQueue*) 提供下列描述的公共方法。

Queue.**qsize**()
返回队列的大致大小。注意，qsize() > 0 不保证后续的 get() 不被阻塞，qsize() < maxsize 也不保证 put() 不被阻塞。

Queue.**empty**()
如果队列为空，返回 `True` ，否则返回 `False` 。如果 empty() 返回 `True` ，不保证后续调用的 put() 不被阻塞。类似的，如果 empty() 返回 `False` ，也不保证后续调用的 get() 不被阻塞。

Queue.**full**()
如果队列是满的返回 `True` ，否则返回 `False` 。如果 full() 返回 `True` 不保证后续调用的 get() 不被阻塞。类似的，如果 full() 返回 `False` 也不保证后续调用的 put() 不被阻塞。

Queue.**put**(*item*, *block=True*, *timeout=None*)
将 *item* 放入队列。如果可选参数 *block* 是 true 并且 *timeout* 是 `None` (默认)，则在必要时阻塞至有空闲插槽可用。如果 *timeout* 是个正数，将最多阻塞 *timeout* 秒，如果在这段时间没有可用的空闲插槽，将引发*Full* 异常。反之 (*block* 是 false)，如果空闲插槽立即可用，则把 *item* 放入队列，否则引发*Full* 异常 (在这种情况下，*timeout* 将被忽略)。

Queue.**put_nowait**(*item*)
相当于 `put(item, False)` 。

Queue.**get**(*block=True*, *timeout=None*)
从队列中移除并返回一个项目。如果可选参数 *block* 是 true 并且 *timeout* 是 `None` (默认值)，则在必要时阻塞至项目可得到。如果 *timeout* 是个正数，将最多阻塞 *timeout* 秒，如果在这段时间内项目不能得到，将引发*Empty* 异常。反之 (*block* 是 false) , 如果一个项目立即可得到，则返回一个项目，否则引发*Empty* 异常 (这种情况下，*timeout* 将被忽略)。

Queue.**get_nowait**()
相当于 `get(False)` 。

提供了两个方法，用于支持跟踪排队的任务是否被守护的消费者线程完整的处理。

Queue.**task_done**()
表示前面排队的任务已经被完成。被队列的消费者线程使用。每个*get()* 被用于获取一个任务，后续调用*task_done()* 告诉队列，该任务的处理已经完成。

如果*join()* 当前正在阻塞，在所有条目都被处理后，将解除阻塞 (意味着每个*put()* 进队列的条目的*task_done()* 都被收到)。

如果被调用的次数多于放入队列中的项目数量，将引发*ValueError* 异常。

Queue.**join**()
阻塞至队列中所有的元素都被接收和处理完毕。

当条目添加到队列的时候，未完成任务的计数就会增加。每当消费者线程调用*task_done()* 表示这个条目已经被回收，该条目所有工作已经完成，未完成计数就会减少。当未完成计数降到零的时候，*join()* 阻塞被解除。

如何等待排队的任务被完成的示例：

```
def worker():
    while True:
        item = q.get()
        if item is None:
            break
        do_work(item)
        q.task_done()
```

(下页继续)

(续上页)

```
q = queue.Queue()
threads = []
for i in range(num_worker_threads):
    t = threading.Thread(target=worker)
    t.start()
    threads.append(t)

for item in source():
    q.put(item)

# block until all tasks are done
q.join()

# stop workers
for i in range(num_worker_threads):
    q.put(None)
for t in threads:
    t.join()
```

参见:

类*multiprocessing.Queue* 一个用于多进程上下文的队列类（而不是多线程）。

collections.deque 是无界队列的替代实现，具有快速原子的*append()* 和*popleft()* 操作，不需要锁定。

以下是上述某些服务的支持模块:

17.8 dummy_threading —可直接替代 threading 模块。

源代码: Lib/dummy_threading.py

This module provides a duplicate interface to the *threading* module. It is meant to be imported when the *_thread* module is not provided on a platform.

Suggested usage is:

```
try:
    import threading
except ImportError:
    import dummy_threading as threading
```

如果线程需要阻塞等待另一个线程被创建的话，可能会造成死锁，这通常是由于阻塞 I/O 引起的。这种场景下请不要使用这个模块。

17.9 _thread —底层多线程 API

该模块提供了操作多个线程（也被称为 *轻量级进程* 或 *任务*）的底层原语——多个控制线程共享全局数据空间。为了处理同步问题，也提供了简单的锁机制（也称为 *互斥锁* 或 *二进制信号*）。*threading* 模块基于该模块提供了更易用的高级多线程 API。

The module is optional. It is supported on Windows, Linux, SGI IRIX, Solaris 2.x, as well as on systems that have a POSIX thread (a.k.a. "pthread") implementation. For systems lacking the *_thread* module, the *_dummy_thread* module is available. It duplicates this module's interface and can be used as a drop-in replacement.

It defines the following constants and functions:

exception _thread.**error**
发生线程相关错误时抛出。

在 3.3 版更改: 现在是内建异常*RuntimeError* 的别名。

_thread.**LockType**
锁对象的类型。

_thread.**start_new_thread**(*function*, *args*[, *kwargs*])
启动一个线程，并返回其标识符。线程会用 *args* 作为参数（必须是元组）执行 *function* 函数。可选的 *kwargs* 参数使用字典来指定有名参数。当函数返回时，线程会静默退出，当函数由于未处理的异常而中止时，会打印一条堆栈追踪信息，然后该线程会退出（但其他线程还是会继续运行）。

_thread.**interrupt_main**()
Raise a *KeyboardInterrupt* exception in the main thread. A subthread can use this function to interrupt the main thread.

_thread.**exit**()
抛出*SystemExit* 异常。如果没有捕获的话，这个异常会使线程退出。

_thread.**allocate_lock**()
返回一个新的锁对象。锁中的方法在后面描述。初始情况下锁处于解锁状态。

_thread.**get_ident**()
返回当前线程的 "线程标识符"。它是一个非零的整数。它的值没有直接含义，主要是用作 magic cookie，比如作为含有线程相关数据的字典的索引。线程标识符可能会在线程退出，新线程创建时被复用。

_thread.**stack_size**([*size*])
Return the thread stack size used when creating new threads. The optional *size* argument specifies the stack size to be used for subsequently created threads, and must be 0 (use platform or configured default) or a positive integer value of at least 32,768 (32 KiB). If *size* is not specified, 0 is used. If changing the thread stack size is unsupported, a *RuntimeError* is raised. If the specified stack size is invalid, a *ValueError* is raised and the stack size is unmodified. 32 KiB is currently the minimum supported stack size value to guarantee sufficient stack space for the interpreter itself. Note that some platforms may have particular restrictions on values for the stack size, such as requiring a minimum stack size > 32 KiB or requiring allocation in multiples of the system memory page size - platform documentation should be referred to for more information (4 KiB pages are common; using multiples of 4096 for the stack size is the suggested approach in the absence of more specific information). Availability: Windows, systems with POSIX threads.

_thread.**TIMEOUT_MAX**
`Lock.acquire()` 方法中 *timeout* 参数允许的最大值。传入超过这个值的 timeout 会抛出*OverflowError* 异常。

3.2 新版功能.

锁对象有以下方法：

lock.**acquire**(*waitflag=1*, *timeout=-1*)
没有任何可选参数时，该方法无条件申请获得锁，有必要的话会等待其他线程释放锁（同时只有一个线程能获得锁——这正是锁存在的原因）。

如果传入了整型参数 *waitflag*，具体的行为取决于传入的值：如果是 0 的话，只会在能够立刻获取到锁时才获取，不会等待，如果是非零的话，会像之前提到的一样，无条件获取锁。

如果传入正浮点数参数 *timeout*，相当于指定了返回之前等待得最大秒数。如果传入负的 *timeout*，相当于无限期等待。如果 *waitflag* 是 0 的话，不能指定 *timeout*。

如果成功获取到所会返回 `True`，否则返回 `False`。

在 3.2 版更改: 新的 *timeout* 形参。

在 3.2 版更改: 现在获取锁的操作可以被 POSIX 信号中断。

lock.**release**()
释放锁。锁必须已经被获取过，但不一定是同一个线程获取的。

lock.**locked**()
返回锁的状态：如果已被某个线程获取，返回 `True`，否则返回 `False`。

除了这些方法之外，锁对象也可以通过 `with` 语句使用，例如：

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

注意事项：

- 线程与中断奇怪地交互：*KeyboardInterrupt* 异常可能会被任意一个线程捕获。（如果*signal* 模块可用的话，中断总是会进入主线程。）
- 调用*sys.exit()* 或是抛出*SystemExit* 异常等效于调用*_thread.exit()*。
- 不可能中断锁的 `acquire()` 方法——*KeyboardInterrupt* 一场会在锁获取到之后发生。
- 当主线程退出时，由系统决定其他线程是否存活。在大多数系统中，这些线程会直接被杀掉，不会执行 `try …finally` 语句，也不会执行对象析构函数。
- 当主线程退出时，不会进行正常的清理工作（除非使用了 `try …finally` 语句），标准 I/O 文件也不会刷新。

17.10 _dummy_thread —_thread 的替代模块

源代码： Lib/_dummy_thread.py

This module provides a duplicate interface to the *_thread* module. It is meant to be imported when the *_thread* module is not provided on a platform.

Suggested usage is:

```
try:
    import _thread
except ImportError:
    import _dummy_thread as _thread
```

如果线程需要阻塞等待另一个线程被创建的话，可能会造成死锁，这通常是由于阻塞 I/O 引起的。这种场景下请不要使用这个模块。

CHAPTER 18

Interprocess Communication and Networking

The modules described in this chapter provide mechanisms for different processes to communicate.

某些模块仅适用于同一台机器上的两个进程，例如*signal* 和*mmap* 。其他模块支持两个或多个进程可用于跨机器通信的网络协议。

本章中描述的模块列表是：

18.1 socket —底层网络接口

源代码： Lib/socket.py

这个模块提供了访问 BSD* 套接字 * 的接口。在所有现代 Unix 系统、Windows、macOS 和其他一些平台上可用。

注解： 一些行为可能因平台不同而异，因为调用的是操作系统的套接字 API。

这个 Python 接口是用 Python 的面向对象风格对 Unix 系统调用和套接字库接口的直译：函数*socket()* 返回一个 *套接字对象*，其方法是对各种套接字系统调用的实现。形参类型一般与 C 接口相比更高级：例如在 Python 文件 `read()` 和 `write()` 操作中，接收操作的缓冲区分配是自动的，发送操作的缓冲区长度是隐式的。

参见：

模块*socketserver* 用于简化网络服务端编写的类。

模块*ssl* 套接字对象的 TLS/SSL 封装。

18.1.1 套接字协议族

根据系统以及构建选项，此模块提供了各种 socket 协议簇。

特定的 socket 对象需要的地址格式将根据此 socket 对象被创建时指定的地址族被自动选择。socket 地址表示如下：

- 一个绑定在文件系统节点上的*AF_UNIX* 套接字的地址表示为一个字符串，使用文件系统字符编码和 `'surrogateescape'` 错误回调方法（see **PEP 383**）。一个地址在 Linux 的抽象命名空间被返回为带有初始的 null 字节的*bytes-like object*；注意在这个命名空间种的套接字可能与普通文件系统套接字通信，所以打算运行在 Linux 上的程序可能需要解决两种地址类型。当传递为参数时，一个字符串或字节类对象可以用于任一类型的地址。

  在 3.3 版更改: 以前，*AF_UNIX* 套接字路径被假设使用 UTF-8 编码。

  在 3.5 版更改: 现在支持可写的字节类对象。

- 一对 `(host, port)` 被用于*AF_INET* 地址族，*host* 是一个代表互联网域名表示法之内主机名或者一个 IPv4 地址的字符串，例如 `'daring.cwi.nl'` 或 `'100.50.200.5'`，*port* 是一个整数。
  - 对于 IPv4 地址，有两种可接受的特殊形式被用来代替一个主机地址：`''` 代表 `INADDR_ANY`，用来绑定到所有接口；字符串 `'<broadcast>'` 代表 `INADDR_BROADCAST`。此行为不兼容 IPv6，因此，如果你的 Python 程序打算支持 IPv6，则可能需要避开这些。

- 对于*AF_INET6* 地址族，使用一个四元组 `(host, port, flowinfo, scopeid)`，*flowinfo* 和 *scopeid* 代表了 C 库里 `struct sockaddr_in6` 中的 `sin6_flowinfo` 和 `sin6_scope_id` 成员。对于*socket* 模块中的方法，*flowinfo* 和 *scopeid* 可以被省略，只为了向后兼容。注意，*scopeid* 的省略可能会导致 IPv6 地址的操作范围问题。

- `AF_NETLINK` 套接字由一对 `(pid, groups)` 表示。

- 指定 `AF_TIPC` 地址族可以使用仅 Linux 支持的 TIPC 协议。TIPC 是一种开放的、非基于 IP 的网络协议，旨在用于集群计算环境。其地址用元组表示，其中的字段取决于地址类型。一般元组形式为 `(addr_type, v1, v2, v3 [, scope])`，其中：
  - *addr_type* 取 `TIPC_ADDR_NAMESEQ`、`TIPC_ADDR_NAME` 或 `TIPC_ADDR_ID` 中的一个。
  - *scope* 取 `TIPC_ZONE_SCOPE`、`TIPC_CLUSTER_SCOPE` 和 `TIPC_NODE_SCOPE` 中的一个。
  - 如果 *addr_type* 为 `TIPC_ADDR_NAME`，那么 *v1* 是服务器类型，*v2* 是端口标识符，*v3* 应为 0。

    如果 *addr_type* 为 `TIPC_ADDR_NAMESEQ`，那么 *v1* 是服务器类型，*v2* 是端口号下限，而 *v3* 是端口号上限。

    如果 *addr_type* 为 `TIPC_ADDR_ID`，那么 *v1* 是节点 (node)，*v2* 是 ref，*v3* 应为 0。

- *AF_CAN* 地址族使用元组 `(interface, )`，其中 *interface* 是表示网络接口名称的字符串，如 `'can0'`。网络接口名 `''` 可以用于接收本族所有网络接口的数据包。

- `PF_SYSTEM` 协议簇的 `SYSPROTO_CONTROL` 协议接受一个字符串或元组 `(id, unit)`。其中字符串是内核控件的名称，该控件使用动态分配的 ID。而如果 ID 和内核控件的单元 (unit) 编号都已知，或使用了已注册的 ID，可以采用元组。

  3.3 新版功能.

- `AF_BLUETOOTH` 支持以下协议和地址格式：
  - `BTPROTO_L2CAP` 接受 `(bdaddr, psm)`，其中 `bdaddr` 为字符串格式的蓝牙地址，`psm` 是一个整数。
  - `BTPROTO_RFCOMM` 接受 `(bdaddr, channel)`，其中 `bdaddr` 为字符串格式的蓝牙地址，`channel` 是一个整数。

  - BTPROTO_HCI 接受 (device_id,)，其中 device_id 为整数或字符串，它表示接口对应的蓝牙地址（具体取决于你的系统，NetBSD 和 DragonFlyBSD 需要蓝牙地址字符串，其他系统需要整数）。

    在 3.2 版更改: 添加 NetBSD 和 DragonFlyBSD 支持。

  - BTPROTO_SCO 接受 bdaddr，其中 bdaddr 是*bytes*对象，其中含有字符串格式的蓝牙地址（如 b'12:23:34:45:56:67'），FreeBSD 不支持此协议。

- *AF_ALG* 是一个仅 Linux 可用的、基于套接字的接口，用于连接内核加密算法。算法套接字可用包括 2 至 4 个元素的元组来配置 (type, name [, feat [, mask]])，其中：

  - *type* 是表示算法类型的字符串，如 aead、hash、skcipher 或 rng。

  - *name* 是表示算法类型和操作模式的字符串，如 sha256、hmac(sha256)、cbc(aes) 或 drbg_nopr_ctr_aes256。

  - *feat* 和 *mask* 是无符号 32 位整数。

  Availability Linux 2.6.38, some algorithm types require more recent Kernels.

  3.6 新版功能.

- *AF_PACKET* 是一个底层接口，直接连接至网卡。数据包使用元组 (ifname, proto[, pkttype[, hatype[, addr]]]) 表示，其中：

  - *ifname* - 指定设备名称的字符串。

  - *proto* - 一个用网络字节序表示的整数，指定以太网协议版本号。

  - *pkttype* - 指定数据包类型的整数（可选）：

    * PACKET_HOST （默认）- 数据包寻址到本地宿主机。

    * PACKET_BROADCAST - 物理层广播报文。

    * PACKET_MULTIHOST - 数据包发送到物理层多播地址。

    * PACKET_OTHERHOST - 被（处于混杂模式的）网卡驱动捕获的、发送到其他主机的数据包。

    * PACKET_OUTGOING - 来自本地主机的、回环到一个套接字的数据包。

  - *hatype* - 可选整数，指定 ARP 硬件地址类型。

  - *addr* - 可选的类字节串对象，用于指定硬件物理地址，其解释取决于各设备。

如果你在 IPv4/v6 套接字地址的 *host* 部分中使用了一个主机名，此程序可能会表现不确定行为，因为 Python 使用 DNS 解析返回的第一个地址。套接字地址在实际的 IPv4/v6 中以不同方式解析，根据 DNS 解析和/或 host 配置。为了确定行为，在 *host* 部分中使用数字地址。

所有的错误都抛出异常。对于无效参数类型和内存溢出异常情况可能抛出普通异常；从 Python 3.3 开始，与套接字或地址语义有关的错误抛出*OSError*或它的子类之一（常用*socket.error*）。

可以用*setblocking()*设置非阻塞模式。一个基于超时的 generalization 通过*settimeout()*支持。

18.1.2 模块内容

socket 模块导出以下元素。

异常

exception socket.**error**
一个被弃用的*OSError* 的别名。

在 3.3 版更改: 根据 **PEP 3151**，这个类是*OSError* 的别名。

exception socket.**herror**
OSError 的子类，本异常通常表示与地址相关的错误，比如那些在 POSIX C API 中使用了 *h_errno* 的函数，包括*gethostbyname_ex()* 和*gethostbyaddr()*。附带的值是一对 (h_errno, string)，代表库调用返回的错误。*h_errno* 是一个数字，而 *string* 表示 *h_errno* 的描述，它们由 C 函数 hstrerror() 返回。

在 3.3 版更改: 此类是*OSError* 的子类。

exception socket.**gaierror**
OSError 的子类，本异常来自*getaddrinfo()* 和*getnameinfo()*，表示与地址相关的错误。附带的值是一对 (error, string)，代表库调用返回的错误。*string* 表示 *error* 的描述，它由 C 函数 gai_strerror() 返回。数字值 *error* 与本模块中定义的 EAI_* 常量之一匹配。

在 3.3 版更改: 此类是*OSError* 的子类。

exception socket.**timeout**
OSError 的子类，当套接字发生超时，且事先已调用过*settimeout()*（或隐式地通过*setdefaulttimeout()*）启用了超时，则会抛出此异常。附带的值是一个字符串，其值总是“timed out”。

在 3.3 版更改: 此类是*OSError* 的子类。

常量

AF_* 和 SOCK_* 常量现在都在 AddressFamily 和 SocketKind 这两个*IntEnum* 集合内。

3.4 新版功能.

socket.**AF_UNIX**
socket.**AF_INET**
socket.**AF_INET6**
这些常量表示地址（和协议）簇，用于*socket()* 的第一个参数。如果*AF_UNIX* 常量未定义，即表示不支持该协议。不同系统可能会有更多其他常量可用。

socket.**SOCK_STREAM**
socket.**SOCK_DGRAM**
socket.**SOCK_RAW**
socket.**SOCK_RDM**
socket.**SOCK_SEQPACKET**
这些常量表示套接字类型，用于*socket()* 的第二个参数。不同系统可能会有更多其他常量可用。(一般只有*SOCK_STREAM* 和*SOCK_DGRAM* 可用)

socket.**SOCK_CLOEXEC**
socket.**SOCK_NONBLOCK**
这两个常量（如果已定义）可以与上述套接字类型结合使用，并允许你设置一些原子性的 flags （从而避免可能的竞争条件和单独调用的需要）。

参见:

Secure File Descriptor Handling （安全地处理文件描述符） 提供了更详尽的解释。

Availability: Linux >= 2.6.27.

3.2 新版功能.

SO_*
socket.**SOMAXCONN**
MSG_*
SOL_*
SCM_*
IPPROTO_*
IPPORT_*
INADDR_*
IP_*
IPV6_*
EAI_*
AI_*
NI_*
TCP_*
此列表内的许多常量，记载在 Unix 文档中的套接字和/或 IP 协议部分，同时也定义在本 socket 模块中。它们通常用于套接字对象的 `setsockopt()` 和 `getsockopt()` 方法的参数中。在大多数情况下，仅那些在 Unix 头文件中有定义的符号会在本模块中定义，部分符号提供了默认值。

在 3.6 版更改: 添加了 `SO_DOMAIN`、`SO_PROTOCOL`、`SO_PEERSEC`、`SO_PASSSEC`、`TCP_USER_TIMEOUT`、`TCP_CONGESTION`。

在 3.6.5 版更改: 在 Windows 上，如果 Windows 运行时支持，则 `TCP_FASTOPEN`、`TCP_KEEPCNT` 可用。

socket.**AF_CAN**
socket.**PF_CAN**
SOL_CAN_*
CAN_*
此列表内的许多常量，记载在 Linux 文档中，同时也定义在本 socket 模块中。

Availability: Linux >= 2.6.25.

3.3 新版功能.

socket.**CAN_BCM**
CAN_BCM_*
CAN 协议簇内的 CAN_BCM 是广播管理器（Bbroadcast Manager –BCM）协议，广播管理器常量在 Linux 文档中有所记载，在本 socket 模块中也有定义。

Availability: Linux >= 2.6.25.

3.4 新版功能.

socket.**CAN_RAW_FD_FRAMES**
Enables CAN FD support in a CAN_RAW socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames; however, you one must accept both CAN and CAN FD frames when reading from the socket.

此常量在 Linux 文档中有所记载。

Availability: Linux >= 3.6.

3.5 新版功能.

socket.**AF_PACKET**

socket.**PF_PACKET**
PACKET_*
此列表内的许多常量，记载在 Linux 文档中，同时也定义在本 socket 模块中。

Availability: Linux >= 2.2.

socket.**AF_RDS**
socket.**PF_RDS**
socket.**SOL_RDS**
RDS_*
此列表内的许多常量，记载在 Linux 文档中，同时也定义在本 socket 模块中。

Availability: Linux >= 2.6.30.

3.3 新版功能.

socket.**SIO_RCVALL**
socket.**SIO_KEEPALIVE_VALS**
socket.**SIO_LOOPBACK_FAST_PATH**
RCVALL_*
Windows 的 WSAIoctl() 的常量。这些常量用于套接字对象的*ioctl()* 方法的参数。

在 3.6 版更改: 添加了 SIO_LOOPBACK_FAST_PATH。

TIPC_*
TIPC 相关常量，与 C socket API 导出的常量一致。更多信息请参阅 TIPC 文档。

socket.**AF_ALG**
socket.**SOL_ALG**
ALG_*
用于 Linux 内核加密算法的常量。

Availability: Linux >= 2.6.38.

3.6 新版功能.

socket.**AF_LINK**
Availability: BSD, OSX.

3.4 新版功能.

socket.**has_ipv6**
本常量为一个布尔值，该值指示当前平台是否支持 IPv6。

socket.**BDADDR_ANY**
socket.**BDADDR_LOCAL**
这些是字符串常量，包含蓝牙地址，这些地址具有特殊含义。例如，当用 BTPROTO_RFCOMM 指定绑定套接字时，*BDADDR_ANY* 表示“任何地址”。

socket.**HCI_FILTER**
socket.**HCI_TIME_STAMP**
socket.**HCI_DATA_DIR**
与 BTPROTO_HCI 一起使用。*HCI_FILTER* 在 NetBSD 或 DragonFlyBSD 上不可用。*HCI_TIME_STAMP* 和*HCI_DATA_DIR* 在 FreeBSD、NetBSD 或 DragonFlyBSD 上不可用。

函数

创建套接字

下列函数都能创建套接字对象.

socket.**socket**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

Create a new socket using the given address family, socket type and protocol number. The address family should be *AF_INET* (the default), *AF_INET6*, *AF_UNIX*, *AF_CAN*, *AF_PACKET*, or *AF_RDS*. The socket type should be *SOCK_STREAM* (the default), *SOCK_DGRAM*, *SOCK_RAW* or perhaps one of the other SOCK_ constants. The protocol number is usually zero and may be omitted or in the case where the address family is *AF_CAN* the protocol should be one of CAN_RAW or *CAN_BCM*. If *fileno* is specified, the other arguments are ignored, causing the socket with the specified file descriptor to return. Unlike *socket.fromfd()*, *fileno* will return the same socket and not a duplicate. This may help close a detached socket using *socket.close()*.

新创建的套接字是不可继承的。

在 3.3 版更改: 添加了 AF_CAN 簇。添加了 AF_RDS 簇。

在 3.4 版更改: 添加了 CAN_BCM 协议。

在 3.4 版更改: 返回的套接字现在是不可继承的。

socket.**socketpair**([*family*[, *type*[, *proto*]]])

构建一对已连接的套接字对象，使用给定的地址簇、套接字类型和协议号。地址簇、套接字类型和协议号与上述*socket()* 函数相同。默认地址簇为*AF_UNIX* （需要当前平台支持，不支持则默认为*AF_INET* ）。

新创建的套接字都是不可继承的。

在 3.2 版更改: 现在，返回的套接字对象支持全部套接字 API，而不是全部 API 的一个子集。

在 3.4 版更改: 返回的套接字现在都是不可继承的。

在 3.5 版更改: 添加了 Windows 支持。

socket.**create_connection**(*address*[, *timeout*[, *source_address*]])

连接到一个 TCP 服务，该服务正在侦听 Internet *address* （用二元组 `(host, port)` 表示）。连接后返回套接字对象。这是比*socket.connect()* 更高级的函数：如果 *host* 是非数字主机名，它将尝试从*AF_INET* 和*AF_INET6* 解析它，然后依次尝试连接到所有可能的地址，直到连接成功。这使得编写兼容 IPv4 和 IPv6 的客户端变得容易。

传入可选参数 *timeout* 可以在套接字实例上设置超时（在尝试连接前）。如果未提供 *timeout*，则使用由*getdefaulttimeout()* 返回的全局默认超时设置。

如果提供了 *source_address*，它必须为二元组 `(host, port)`，以便套接字在连接之前绑定为其源地址。如果 host 或 port 分别为 '' 或 0，则使用操作系统默认行为。

在 3.2 版更改: 添加了 *source_address* 参数

socket.**fromfd**(*fd*, *family*, *type*, *proto=0*)

复制文件描述符 *fd* （一个由文件对象的 `fileno()` 方法返回的整数），然后从结果中构建一个套接字对象。地址簇、套接字类型和协议号与上述*socket()* 函数相同。文件描述符应指向一个套接字，但不会专门去检查——如果文件描述符是无效的，则对该对象的后续操作可能会失败。本函数很少用到，但是在将套接字作为标准输入或输出传递给程序（如 Unix inet 守护程序启动的服务器）时，可以使用本函数获取或设置套接字选项。套接字须处于阻塞模式。

新创建的套接字是不可继承的。

在 3.4 版更改: 返回的套接字现在是不可继承的。

socket.**fromshare**(*data*)
根据*socket.share()* 方法获得的数据实例化套接字。套接字须处于阻塞模式。

Availability: Windows.

3.3 新版功能.

socket.**SocketType**
这是一个 Python 类型对象，表示套接字对象的类型。它等同于 type(socket(...))。

其他功能

socket 模块还提供各种与网络相关的服务：

socket.**getaddrinfo**(*host*, *port*, *family=0*, *type=0*, *proto=0*, *flags=0*)
将 *host*/*port* 参数转换为 5 元组的序列，其中包含创建（连接到某服务的）套接字所需的所有参数。*host* 是域名，是字符串格式的 IPv4/v6 地址或 None。*port* 是字符串格式的服务名称，如 'http' 、端口号（数字）或 None。传入 None 作为 *host* 和 *port* 的值，相当于将 NULL 传递给底层 C API。

可以指定 *family*、*type* 和 *proto* 参数，以缩小返回的地址列表。向这些参数分别传入 0 表示保留全部结果范围。*flags* 参数可以是 AI_* 常量中的一个或多个，它会影响结果的计算和返回。例如，AI_NUMERICHOST 会禁用域名解析，此时如果 *host* 是域名，则会抛出错误。

本函数返回的 5 元组列表具有以下结构：

```
(family, type, proto, canonname, sockaddr)
```

在这些元组中，*family*、*type*、*proto* 都是整数，可以用于传递给*socket()* 函数。如果 *flags* 参数有一部分是 AI_CANONNAME，那么 *canonname* 将是表示 *host* 的规范名称的字符串。否则 *canonname* 将为空。*sockaddr* 是一个表示套接字地址的元组，具体格式取决于返回的 *family*（对于*AF_INET*，是一个 (address, port) 二元组，对于*AF_INET6*，是一个 (address, port, flow info, scope id) 四元组），可以用于传递给*socket.connect()* 方法。

下面的示例获取了 TCP 连接地址信息，假设该连接通过 80 端口连接至 example.org（如果系统未启用 IPv6，则结果可能会不同）：

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(<AddressFamily.AF_INET6: 10>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('93.184.216.34', 80))]
```

在 3.2 版更改: 现在可以使用关键字参数的形式来传递参数。

socket.**getfqdn**([*name*])
返回 *name* 的全限定域名 (Fully Qualified Domain Name –FQDN)。如果 *name* 省略或为空，则将其解释为本地主机。为了查找全限定名称，首先将检查由*gethostbyaddr()* 返回的主机名，然后是主机的别名（如果存在）。选中第一个包含句点的名字。如果无法获取全限定域名，则返回由*gethostname()* 返回的主机名。

socket.**gethostbyname**(*hostname*)
将主机名转换为 IPv4 地址格式。IPv4 地址以字符串格式返回，如 '100.50.200.5'。如果主机名本身是 IPv4 地址，则原样返回。更完整的接口请参考*gethostbyname_ex()*。*gethostbyname()* 不支持 IPv6 域名解析，应使用*getaddrinfo()* 来支持 IPv4/v6 双协议栈。

socket.**gethostbyname_ex**(*hostname*)
将主机名转换为 IPv4 地址格式的扩展接口。返回三元组 (hostname, aliaslist, ipaddrlist)，其中 *hostname* 是响应给定 *ip_address* 的主要主机名，*aliaslist* 是相同地址的其他可用主机名的列表（可能为空），而 *ipaddrlist* 是 IPv4 地址列表，包含相同主机名、相同接口的不同地址（通常是一个地址，但不

总是如此)。*gethostbyname_ex()* 不支持 IPv6 名称解析，应使用*getaddrinfo()* 来支持 IPv4/v6 双协议栈。

socket.**gethostname**()
返回一个字符串，包含当前正在运行 Python 解释器的机器的主机名。

注意：*gethostname()* 并不总是返回全限定域名，必要的话请使用*getfqdn()*。

socket.**gethostbyaddr**(*ip_address*)
返回三元组 (hostname, aliaslist, ipaddrlist)，其中 *hostname* 是响应给定 *ip_address* 的主要主机名，*aliaslist* 是相同地址的其他可用主机名的列表（可能为空），而 *ipaddrlist* 是 IPv4/v6 地址列表，包含相同主机名、相同接口的不同地址（很可能仅包含一个地址)。要查询全限定域名，请使用函数*getfqdn()*。*gethostbyaddr()* 支持 IPv4 和 IPv6。

socket.**getnameinfo**(*sockaddr*, *flags*)
将套接字地址 *sockaddr* 转换为二元组 (host, port)。*host* 的形式可能是全限定域名，或是由数字表示的地址，具体取决于 *flags* 的设置。同样，*port* 可以包含字符串格式的端口名称或数字格式的端口号。

socket.**getprotobyname**(*protocolname*)
将 Internet 协议名称（如 'icmp'）转换为常量，该常量适用于*socket()* 函数的第三个（可选）参数。通常只有在“raw”模式 (*SOCK_RAW*) 中打开的套接字才需要使用该常量。对于正常的套接字模式，协议省略或为零时，会自动选择正确的协议。

socket.**getservbyname**(*servicename*[, *protocolname*])
将 Internet 服务名称和协议名称转换为该服务的端口号。协议名称是可选的，如果提供的话应为 'tcp' 或 'udp'，否则将匹配出所有协议。

socket.**getservbyport**(*port*[, *protocolname*])
将 Internet 端口号和协议名称转换为该服务的服务名称。协议名称是可选的，如果提供的话应为 'tcp' 或 'udp'，否则将匹配出所有协议。

socket.**ntohl**(*x*)
将 32 位正整数从网络字节序转换为主机字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 4 字节交换操作。

socket.**ntohs**(*x*)
将 16 位正整数从网络字节序转换为主机字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 2 字节交换操作。

socket.**htonl**(*x*)
将 32 位正整数从主机字节序转换为网络字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 4 字节交换操作。

socket.**htons**(*x*)
将 16 位正整数从主机字节序转换为网络字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 2 字节交换操作。

socket.**inet_aton**(*ip_string*)
将 IPv4 地址从点分十进制字符串格式（如‘123.45.67.89’）转换为 32 位压缩二进制格式，转换后为字节对象，长度为四个字符。与那些使用标准 C 库，且需要 struct in_addr 类型的对象的程序交换信息时，此功能很有用。该类型即此函数返回的 32 位压缩二进制的 C 类型。

inet_aton() 也接受句点数少于三的字符串，详情请参阅 Unix 手册 *inet(3)*。

如果传入本函数的 IPv4 地址字符串无效，则抛出*OSError*。注意，具体什么样的地址有效取决于 inet_aton() 的底层 C 实现。

inet_aton() 不支持 IPv6，在 IPv4/v6 双协议栈下应使用*inet_pton()* 来代替。

socket.**inet_ntoa**(*packed_ip*)
将 32 位压缩 IPv4 地址（一个类字节对象，长 4 个字节）转换为标准的点分十进制字符串形式（如

'123.45.67.89')。与那些使用标准 C 库，且需要 `struct in_addr` 类型的对象的程序交换信息时，本函数很有用。该类型即本函数参数中的 32 位压缩二进制数据的 C 类型。

如果传入本函数的字节序列长度不是 4 个字节，则抛出*OSError*。*inet_ntoa()* 不支持 IPv6，在 IPv4/v6 双协议栈下应使用*inet_ntop()* 来代替。

在 3.5 版更改: 现在支持可写的字节类对象。

`socket.`**`inet_pton`**(*address_family*, *ip_string*)
将特定地址簇的 IP 地址(字符串)转换为压缩二进制格式。当库或网络协议需要接受 `struct in_addr` 类型的对象（类似*inet_aton()* ）或 `struct in6_addr` 类型的对象时，*inet_pton()* 很有用。

目前 *address_family* 支持*AF_INET* 和*AF_INET6*。如果 IP 地址字符串 *ip_string* 无效，则抛出*OSError*。注意，具体什么地址有效取决于 *address_family* 的值和 `inet_pton()` 的底层实现。

Availability: Unix (maybe not all platforms), Windows.

在 3.4 版更改: 添加了 Windows 支持

`socket.`**`inet_ntop`**(*address_family*, *packed_ip*)
将压缩 IP 地址（一个类字节对象，数个字节长）转换为标准的、特定地址簇的字符串形式（如 `'7.10.0.5'` 或 `'5aef:2b::8'`)。当库或网络协议返回 `struct in_addr` 类型的对象（类似*inet_ntoa()* ）或 `struct in6_addr` 类型的对象时，*inet_ntop()* 很有用。

目前 *address_family* 支持*AF_INET* 和*AF_INET6*。如果字节对象 *packed_ip* 与指定的地址簇长度不符，则抛出*ValueError*。针对*inet_ntop()* 调用的错误则抛出*OSError*。

Availability: Unix (maybe not all platforms), Windows.

在 3.4 版更改: 添加了 Windows 支持

在 3.5 版更改: 现在支持可写的字节类对象。

`socket.`**`CMSG_LEN`**(*length*)
返回给定 *length* 所关联数据的辅助数据项的总长度（不带尾部填充)。此值通常用作*recvmsg()* 接收一个辅助数据项的缓冲区大小，但是 **RFC 3542** 要求可移植应用程序使用*CMSG_SPACE()*，以此将尾部填充的空间计入，即使该项在缓冲区的最后。如果 *length* 超出允许范围，则抛出*OverflowError*。

Availability: most Unix platforms, possibly others.

3.3 新版功能.

`socket.`**`CMSG_SPACE`**(*length*)
返回*recvmsg()* 所需的缓冲区大小，以接收给定 *length* 所关联数据的辅助数据项，带有尾部填充。接收多个项目所需的缓冲区空间是关联数据长度的*CMSG_SPACE()* 值的总和。如果 *length* 超出允许范围，则抛出*OverflowError*。

请注意，某些系统可能支持辅助数据，但不提供本函数。还需注意，如果使用本函数的结果来设置缓冲区大小，可能无法精确限制可接收的辅助数据量，因为可能会有其他数据写入尾部填充区域。

Availability: most Unix platforms, possibly others.

3.3 新版功能.

`socket.`**`getdefaulttimeout`**()
返回用于新套接字对象的默认超时（以秒为单位的浮点数)。值 `None` 表示新套接字对象没有超时。首次导入 socket 模块时，默认值为 `None`。

`socket.`**`setdefaulttimeout`**(*timeout*)
设置用于新套接字对象的默认超时（以秒为单位的浮点数)。首次导入 socket 模块时，默认值为 `None`。可能的取值及其各自的含义请参阅*settimeout()*。

`socket.`**`sethostname`**(*name*)
将计算机的主机名设置为 *name*。如果权限不足将抛出*OSError*。

Availability: Unix.

3.3 新版功能.

socket.**if_nameindex**()
返回一个列表，包含网络接口（网卡）信息二元组（整数索引，名称字符串）。系统调用失败则抛出*OSError*。

Availability: Unix.

3.3 新版功能.

socket.**if_nametoindex**(*if_name*)
返回网络接口名称相对应的索引号。如果没有所给名称的接口，则抛出*OSError*。

Availability: Unix.

3.3 新版功能.

socket.**if_indextoname**(*if_index*)
返回网络接口索引号相对应的接口名称。如果没有所给索引号的接口，则抛出*OSError*。

Availability: Unix.

3.3 新版功能.

18.1.3 套接字对象

套接字对象具有以下方法。除了*makefile()*，其他都与套接字专用的 Unix 系统调用相对应。

在 3.2 版更改: 添加了对上下文管理器 协议的支持。退出上下文管理器与调用*close()* 等效。

socket.**accept**()
接受一个连接。此 socket 必须绑定到一个地址上并且监听连接。返回值是一个 `(conn, address)` 对，其中 *conn* 是一个 新的套接字对象，用于在此连接上收发数据，*address* 是连接另一端的套接字所绑定的地址。

新创建的套接字是不可继承的。

在 3.4 版更改: 该套接字现在是不可继承的。

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**bind**(*address*)
将套接字绑定到 *address*。套接字必须尚未绑定。(*address* 的格式取决于地址簇——参见上文)

socket.**close**()
将套接字标记为关闭。当*makefile()* 创建的所有文件对象都关闭时，底层系统资源（如文件描述符）也将关闭。一旦上述情况发生，将来对套接字对象的所有操作都会失败。对端将接收不到任何数据（清空队列数据后）。

垃圾回收时，套接字会自动关闭，但建议显式*close()* 它们，或在它们周围使用 `with` 语句。

在 3.6 版更改: 现在，如果底层的 `close()` 调用出错，会抛出*OSError*。

注解: *close()* 释放与连接相关联的资源，但不一定立即关闭连接。如果需要及时关闭连接，请在调用*close()* 之前调用*shutdown()*。

---

socket.**connect**(*address*)
连接到 *address* 处的远程套接字。(*address* 的格式取决于地址簇——参见上文)

如果连接被信号中断，则本方法将等待，直到连接完成。如果信号处理程序未抛出异常，且套接字阻塞中或已超时，则在超时后抛出*socket.timeout*。对于非阻塞套接字，如果连接被信号中断，则本方法将抛出*InterruptedError*异常（或信号处理程序抛出的异常）。

在 3.5 版更改: 本方法现在将等待，直到连接完成，而不是在以下情况抛出*InterruptedError*异常。该情况为，连接被信号中断，信号处理程序未抛出异常，且套接字阻塞中或已超时（具体解释请参阅**PEP 475**）。

socket.**connect_ex**(*address*)
类似于 connect(address)，但是对于 C 级别的 connect() 调用返回的错误，本函数将返回错误指示器，而不是抛出异常（对于其他问题，如 “找不到主机”，仍然可以抛出异常）。如果操作成功，则错误指示器为 0，否则为 errno 变量的值。这对支持如异步连接很有用。

socket.**detach**()
将套接字对象置于关闭状态，而底层的文件描述符实际并不关闭。返回该文件描述符，使其可以重新用于其他目的。

3.2 新版功能.

socket.**dup**()
创建套接字的副本。

新创建的套接字是*不可继承的*。

在 3.4 版更改: 该套接字现在是不可继承的。

socket.**fileno**()
返回套接字的文件描述符（一个小整数），失败返回 -1。配合*select.select()*使用很有用。

在 Windows 下，此方法返回的小整数在允许使用文件描述符的地方无法使用（如*os.fdopen()*）。Unix 无此限制。

socket.**get_inheritable**()
获取套接字文件描述符或套接字句柄的*可继承标志*：如果子进程可以继承套接字则为 True，否则为 False。

3.4 新版功能.

socket.**getpeername**()
返回套接字连接到的远程地址。举例而言，这可以用于查找远程 IPv4/v6 套接字的端口号。（返回的地址格式取决于地址簇——参见上文。）部分系统不支持此函数。

socket.**getsockname**()
返回套接字本身的地址。举例而言，这可以用于查找 IPv4/v6 套接字的端口号。（返回的地址格式取决于地址簇——参见上文。）

socket.**getsockopt**(*level*, *optname*[, *buflen*])
返回指定套接字选项的值（参阅 Unix 手册页 *getsockopt(2)*）。所需的符号常量（SO_* 等）已定义在本模块中。如果未指定 *buflen*，则认为该选项值为整数，由本函数返回该整数值。如果指定 *buflen*，则它定义了用于存放选项值的缓冲区的最大长度，且该缓冲区将作为字节对象返回。对缓冲区的解码工作由调用者自行完成（针对编码为字节串的 C 结构，其解码方法请参阅可选的内置模块*struct*）。

socket.**gettimeout**()
返回套接字操作相关的超时秒数（浮点数），未设置超时则返回 None。它反映最后一次调用*setblocking()*或*settimeout()*后的设置。

socket.**ioctl**(*control*, *option*)

平台 Windows

*ioctl()*方法是 WSAIoctl 系统接口的有限接口。请参考 Win32 文档 以获取更多信息。

在其他平台上，可以使用通用的*fcntl.fcntl()* 和*fcntl.ioctl()* 函数，它们接受套接字对象作为第一个参数。

当前仅支持以下控制码：SIO_RCVALL、SIO_KEEPALIVE_VALS 和 SIO_LOOPBACK_FAST_PATH。

在 3.6 版更改: 添加了 SIO_LOOPBACK_FAST_PATH。

socket.**listen**([*backlog*])
启动一个服务器用于接受连接。如果指定 *backlog*，则它最低为 0（小于 0 会被置为 0），它指定系统允许暂未 accept 的连接数，超过后将拒绝新连接。未指定则自动设为合理的默认值。

在 3.5 版更改: *backlog* 参数现在是可选的。

socket.**makefile**(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)
返回与套接字关联的文件对象。返回的对象的具体类型取决于*makefile()* 的参数。这些参数的解释方式与内置的*open()* 函数相同，其中 *mode* 的值仅支持 'r'（默认），'w' 和 'b'。

套接字必须处于阻塞模式，它可以有超时，但是如果发生超时，文件对象的内部缓冲区可能会以不一致的状态结尾。

关闭*makefile()* 返回的文件对象不会关闭原始套接字，除非所有其他文件对象都已关闭且在套接字对象上调用了*socket.close()*。

注解: 在 Windows 上，由*makefile()* 创建的文件类对象无法作为带文件描述符的文件对象使用，如无法作为*subprocess.Popen()* 的流参数。

---

socket.**recv**(*bufsize*[, *flags*])
从套接字接收数据。返回值是一个字节对象，表示接收到的数据。*bufsize* 指定一次接收的最大数据量。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。

注解: 为了最佳匹配硬件和网络的实际情况，*bufsize* 的值应为 2 的相对较小的幂，如 4096。

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**recvfrom**(*bufsize*[, *flags*])
从套接字接收数据。返回值是一对 (bytes, address)，其中 *bytes* 是字节对象，表示接收到的数据，*address* 是发送端套接字的地址。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。(*address* 的格式取决于地址簇——参见上文)

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])
从套接字接收普通数据（至多 *bufsize* 字节）和辅助数据。*ancbufsize* 参数设置用于接收辅助数据的内部缓冲区的大小（以字节为单位），默认为 0，表示不接收辅助数据。可以使用*CMSG_SPACE()* 或*CMSG_LEN()* 计算辅助数据缓冲区的合适大小，无法放入缓冲区的项目可能会被截断或丢弃。*flags* 参数默认为 0，其含义与*recv()* 中的相同。

返回值是一个四元组：(data, ancdata, msg_flags, address)。*data* 项是一个*bytes* 对象，用于保存接收到的非辅助数据。*ancdata* 项是零个或多个元组 (cmsg_level, cmsg_type, cmsg_data) 组成的列表，表示接收到的辅助数据（控制消息）：*cmsg_level* 和 *cmsg_type* 是分别表示协议级别和协议类型的整数，而 *cmsg_data* 是保存相关数据的*bytes* 对象。*msg_flags* 项由各种标志按位或组成，表示接收消息的情况，详细信息请参阅系统文档。如果接收端套接字断开连接，则 *address* 是发送端套接字的地址（如果有），否则该值无指定。

某些系统上可以利用*AF_UNIX* 套接字通过*sendmsg()* 和*recvmsg()* 在进程之间传递文件描述符。使用此功能时（通常仅限于*SOCK_STREAM* 套接字），*recvmsg()* 将在其辅助数据中返回以下格式的项

(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds)，其中 *fds* 是一个*bytes*对象，是新文件描述符表示为原生 C int 类型的二进制数组。如果*recvmsg()*在系统调用返回后抛出异常，它将首先关闭此机制接收到的所有文件描述符。

对于仅接收到一部分的辅助数据项，一些系统没有指示其截断长度。如果某个项目可能超出了缓冲区的末尾，*recvmsg()*将发出*RuntimeWarning*，并返回其在缓冲区内的部分，前提是该对象被截断于关联数据开始后。

在支持 SCM_RIGHTS 机制的系统上，下方的函数将最多接收 *maxfds* 个文件描述符，返回消息数据和包含描述符的列表（同时忽略意外情况，如接收到无关的控制消息）。另请参阅*sendmsg()*。

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds * fds.
↪itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if (cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS):
            # Append data, ignoring any truncated integers at the end.
            fds.fromstring(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.
↪itemsize)])
    return msg, list(fds)
```

Availability: most Unix platforms, possibly others.

3.3 新版功能.

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError*异常 (原因详见 **PEP 475**)。

socket.**recvmsg_into**(*buffers*[, *ancbufsize*[, *flags*]])

从套接字接收普通数据和辅助数据，其行为与*recvmsg()*相同，但将非辅助数据分散到一系列缓冲区中，而不是返回新的字节对象。*buffers* 参数必须是可迭代对象，它迭代出可供写入的缓冲区（如*bytearray*对象），这些缓冲区将被连续的非辅助数据块填充，直到数据全部写完或缓冲区用完为止。在允许使用的缓冲区数量上，操作系统可能会有限制（*sysconf()*的 SC_IOV_MAX 值）。*ancbufsize* 和 *flags* 参数的含义与*recvmsg()*中的相同。

返回值为四元组：(nbytes, ancdata, msg_flags, address)，其中 *nbytes* 是写入缓冲区的非辅助数据的字节总数，而 *ancdata*、*msg_flags* 和 *address* 与*recvmsg()*中的相同。

示例:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

Availability: most Unix platforms, possibly others.

3.3 新版功能.

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])

从套接字接收数据，将其写入 *buffer* 而不是创建新的字节串。返回值是一对 (nbytes, address)，

其中 *nbytes* 是收到的字节数，*address* 是发送端套接字的地址。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。(*address* 的格式取决于地址簇——参见上文)

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])
从套接字接收至多 *nbytes* 个字节，将其写入缓冲区而不是创建新的字节串。如果 *nbytes* 未指定（或指定为 0），则接收至所给缓冲区的最大可用大小。返回接收到的字节数。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。

socket.**send**(*bytes*[, *flags*])
发送数据给套接字。本套接字必须已连接到远程套接字。可选参数 *flags* 的含义与上述 *recv()* 中的相同。本方法返回已发送的字节数。应用程序要负责检查所有数据是否已发送，如果仅传输了部分数据，程序需要自行尝试传输其余数据。有关该主题的更多信息，请参考 socket-howto。

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**sendall**(*bytes*[, *flags*])
发送数据给套接字。本套接字必须已连接到远程套接字。可选参数 *flags* 的含义与上述 *recv()* 中的相同。与 *send()* 不同，本方法持续从 *bytes* 发送数据，直到所有数据都已发送或发生错误为止。成功后会返回 None。出错后会抛出一个异常，此时并没有办法确定成功发送了多少数据。

在 3.5 版更改: 每次成功发送数据后，套接字超时不再重置。现在，套接字超时是发送所有数据的最大总持续时间。

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**sendto**(*bytes*, *address*)
socket.**sendto**(*bytes*, *flags*, *address*)
发送数据给套接字。本套接字不应连接到远程套接字，而应由 *address* 指定目标套接字。可选参数 *flags* 的含义与上述 *recv()* 中的相同。本方法返回已发送的字节数。(*address* 的格式取决于地址簇——参见上文。)

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**sendmsg**(*buffers*[, *ancdata*[, *flags*[, *address*]]])
将普通数据和辅助数据发送给套接字，将从一系列缓冲区中收集非辅助数据，并将其拼接为一条消息。*buffers* 参数指定的非辅助数据应为可迭代的字节类对象 (如 *bytes* 对象)，在允许使用的缓冲区数量上，操作系统可能会有限制 (*sysconf()* 的 SC_IOV_MAX 值)。*ancdata* 参数指定的辅助数据 (控制消息) 应为可迭代对象，迭代出零个或多个 (cmsg_level, cmsg_type, cmsg_data) 元组，其中 *cmsg_level* 和 *cmsg_type* 是分别指定协议级别和协议类型的整数，而 *cmsg_data* 是保存相关数据的字节类对象。请注意，某些系统 (特别是没有 *CMSG_SPACE()* 的系统) 可能每次调用仅支持发送一条控制消息。*flags* 参数默认为 0，与 *send()* 中的含义相同。如果 *address* 指定为除 None 以外的值，它将作为消息的目标地址。返回值是已发送的非辅助数据的字节数。

在支持 SCM_RIGHTS 机制的系统上，下方的函数通过一个 *AF_UNIX* 套接字来发送文件描述符列表 *fds*。另请参阅 *recvmsg()*。

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.
↪array("i", fds))])
```

Availability: most Unix platforms, possibly others.

3.3 新版功能.

在 3.5 版更改: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**sendmsg_afalg**([*msg*], *, *op*[, *iv*[, *assoclen*[, *flags*]]])
为*AF_ALG*套接字定制的*sendmsg()*版本。可为*AF_ALG*套接字设置模式、IV、AEAD 关联数据的长度和标志位。

Availability: Linux >= 2.6.38

3.6 新版功能.

socket.**sendfile**(*file*, *offset=0*, *count=None*)
使用高性能的*os.sendfile*发送文件，直到达到文件的 EOF 为止，返回已发送的字节总数。*file* 必须是一个以二进制模式打开的常规文件对象。如果*os.sendfile*不可用（如 Windows）或 *file* 不是常规文件，将使用*send()*代替。*offset* 指示从哪里开始读取文件。如果指定了 *count*，它确定了要发送的字节总数，而不会持续发送直到达到文件的 EOF。返回时或发生错误时，文件位置将更新，在这种情况下，*file.tell()*可用于确定已发送的字节数。套接字必须为*SOCK_STREAM*类型。不支持非阻塞的套接字。

3.5 新版功能.

socket.**set_inheritable**(*inheritable*)
设置套接字文件描述符或套接字句柄的可继承标志。

3.4 新版功能.

socket.**setblocking**(*flag*)
设置套接字为阻塞或非阻塞模式：如果 *flag* 为 false，则将套接字设置为非阻塞，否则设置为阻塞。

本方法是某些*settimeout()*调用的简写：

- `sock.setblocking(True)` 相当于 `sock.settimeout(None)`
- `sock.setblocking(False)` 相当于 `sock.settimeout(0.0)`

socket.**settimeout**(*value*)
为阻塞套接字的操作设置超时。*value* 参数可以是非负浮点数，表示秒，也可以是 `None`。如果赋为一个非零值，那么如果在操作完成前超过了超时时间 *value*，后续的套接字操作将抛出*timeout*异常。如果赋为 0，则套接字将处于非阻塞模式。如果指定为 `None`，则套接字将处于阻塞模式。

更多信息请查阅关于套接字超时的说明。

socket.**setsockopt**(*level*, *optname*, *value: int*)

socket.**setsockopt**(*level*, *optname*, *value: buffer*)

socket.**setsockopt**(*level*, *optname*, *None*, *optlen: int*)
Set the value of the given socket option (see the Unix manual page *setsockopt(2)*). The needed symbolic constants are defined in the *socket* module (`SO_*` etc.). The value can be an integer, `None` or a *bytes-like object* representing a buffer. In the later case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module *struct* for a way to encode C structures as bytestrings). When value is set to `None`, optlen argument is required. It's equivalent to call setsockopt C function with optval=NULL and optlen=optlen.

在 3.5 版更改: 现在支持可写的字节类对象。

在 3.6 版更改: 添加了 setsockopt(level, optname, None, optlen: int) 调用形式。

socket.**shutdown**(*how*)
关闭一半或全部的连接。如果 *how* 为 `SHUT_RD`，则后续不再允许接收。如果 *how* 为 `SHUT_WR`，则后续不再允许发送。如果 *how* 为 `SHUT_RDWR`，则后续的发送和接收都不允许。

socket.**share**(*process_id*)
复制套接字，并准备将其与目标进程共享。目标进程必须以 *process_id* 形式提供。然后可以利用某种形式的进程间通信，将返回的字节对象传递给目标进程，还可以使用*fromshare()*在新进程中重新创建套接字。一旦本方法调用完毕，就可以安全地将套接字关闭，因为操作系统已经为目标进程复制了该套接字。

Availability: Windows.

3.3 新版功能.

注意此处没有 `read()` 或 `write()` 方法，请使用不带 *flags* 参数的*recv()* 和*send()* 来替代。

套接字对象还具有以下（只读）属性，这些属性与传入*socket* 构造函数的值相对应。

`socket.`**`family`**
套接字的协议簇

`socket.`**`type`**
套接字的类型

`socket.`**`proto`**
套接字的协议。

18.1.4 关于套接字超时的说明

一个套接字对象可以处于以下三种模式之一：阻塞、非阻塞或超时。套接字默认以阻塞模式创建，但是可以调用*setdefaulttimeout()* 来更改。

- 在 *blocking mode*（阻塞模式）中，操作将阻塞，直到操作完成或系统返回错误（如连接超时）。
- 在 *non-blocking mode*（非阻塞模式）中，如果操作无法立即完成，则操作将失败（不幸的是，不同系统返回的错误不同）：位于*select* 中的函数可用于了解套接字何时以及是否可以读取或写入。
- 在 *timeout mode*（超时模式）下，如果无法在指定的超时内完成操作（抛出*timeout* 异常），或如果系统返回错误，则操作将失败。

注解： 在操作系统层面上，超时模式下的套接字在内部都设置为非阻塞模式。同时，阻塞和超时模式在文件描述符和套接字对象之间共享，这些描述符和对象均应指向同一个网络端点。如果，比如你决定使用套接字的*fileno()*，这一实现细节可能导致明显的结果。

超时与 connect 方法

connect() 操作也受超时设置的约束，通常建议在调用*connect()* 之前调用*settimeout()*，或将超时参数直接传递给*create_connection()*。但是，无论 Python 套接字超时设置如何，系统网络栈都有可能返回自带的连接超时错误。

超时与 accept 方法

如果*getdefaulttimeout()* 的值不是*None*，则*accept()* 方法返回的套接字将继承该超时值。若是 None，返回的套接字行为取决于侦听套接字的设置：

- 如果侦听套接字处于 *阻塞模式*或 *超时模式*，则*accept()* 返回的套接字处于 *阻塞模式*；
- 如果侦听套接字处于 *非阻塞模式*，那么*accept()* 返回的套接字是阻塞还是非阻塞取决于操作系统。如果要确保跨平台时的正确行为，建议手动覆盖此设置。

18.1.5 示例

以下是 4 个使用 TCP/IP 协议的最小示例程序：一台服务器，它将收到的所有数据原样返回（仅服务于一个客户端），还有一个使用该服务器的客户端。注意，服务器必须按序执行*socket()*，*bind()*，*listen()*，*accept()*（可能需要重复执行*accept()* 以服务多个客户端），而客户端仅需要按序执行*socket()*，*connect()*。还须注意，服务器不在侦听套接字上发送*sendall()*/*recv()*，而是在*accept()* 返回的新套接字上发送。

前两个示例仅支持 IPv4。

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

下两个例子与上两个很像，但是同时支持 IPv4 和 IPv6。服务端将监听第一个可用的地址族（它本应同时监听两个）。在大多数支持 IPv6 的系统上，IPv6 将有优先权并且服务端可能不会接受 IPv4 流量。客户端将尝试连接到作为名称解析结果被返回的所有地址，并将流量发送给连接成功的第一个地址。

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
```

(下页继续)

(续上页)

```
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

下面的例子演示了如何在 Windows 上使用原始套接字编写一个非常简单的网络嗅探器。这个例子需要管理员权限来修改接口:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
```

(下页继续)

(续上页)

```
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The last example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use CAN with the broadcast manager protocol instead, open a socket with:

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

在绑定 (CAN_RAW) 或连接 (*CAN_BCM*) socket 之后，你将可以在 socket 对象上正常使用*socket.send()* 以及*socket.recv()* 操作（及同类操作）。

This example might require special privileges:

```
import socket
import struct


# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])


# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')
```

(下页继续)

(续上页)

```
    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

如果多次运行示例，且两次运行间隔很短，可能引发此错误:

```
OSError: [Errno 98] Address already in use
```

这是因为前一次运行使套接字处于 TIME_WAIT 状态，无法立即重用。

要防止这种情况，需要设置一个*socket* 标志 socket.SO_REUSEADDR:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

SO_REUSEADDR 标志告诉内核将处于 TIME_WAIT 状态的本地套接字重新使用，而不必等到固有的超时到期。

参见:

关于套接字编程（C 语言）的介绍，请参阅以下文章:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*，作者 Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*，作者 Samuel J. Leffler et al,

两篇文章都在 UNIX 开发者手册，补充文档 1（第 PS1:7 和 PS1:8 节）中。那些特定于平台的参考资料，它们包含与套接字有关的各种系统调用，也是套接字语义细节的宝贵信息来源。对于 Unix，请参考手册页。对于 Windows，请参阅 WinSock（或 Winsock 2）规范。如果需要支持 IPv6 的 API，读者可能希望参考 **RFC 3493**，标题为 Basic Socket Interface Extensions for IPv6。

18.2 ssl —套接字对象的 TLS/SSL 封装

源代码: Lib/ssl.py

本模块保证了对传安全传输层协议（TLS）的访问 (SSL) 加密，并且提供客户端和服务端层面的网络嵌套字层面的对等连接认证技术。本模块使用 OpenSSL 库。适用于所有现代 Unix 系统、Windows 以及 Mac OS X。只要 Open SSL 存在的系统，都有机会正常使用。

注解: 某些行为可能具有平台依赖，因为调用是根据操作系统的嵌套字 API。不同版本的 Open SSL 也会引起差异：例如 Open SSL 版本 1.0.1 自带 TLSv1.1 和 TLSv1.2

警告: 在阅读安全考量 前不要使用此模块。这样做可能会导致虚假的安全感，因为 ssl 模块的默认设置不一定适合你的应用程序。

本文档记录”ssl”模块的对象和函数；更多关于 TLS,SSL, 和证书的信息，请参阅下方的“详情”选项

This module provides a class, *ssl.SSLSocket*, which is derived from the *socket.socket* type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional methods

such as `getpeercert()`, which retrieves the certificate of the other side of the connection, and `cipher()`,which retrieves the cipher being used for the secure connection.

对于更复杂的应用程序，*ssl.SSLContext* 类有助于管理设置项和证书，进而可以被使用*SSLContext.wrap_socket()* 方法创建的 SSL 套接字继承。

在 3.5.3 版更改: 更新以支持和 OpenSSL 1.1.0 的链接

在 3.6 版更改: OpenSSL 0.9.8、1.0.0 和 1.0.1 已过时，将不再被支持。在 ssl 模块未来的版本中，最低需要 OpenSSL 1.0.2 或 1.1.0。

18.2.1 方法、常量和异常处理

exception ssl.**SSLError**

引发此异常以提示来自下层 SSL 实现（目前由 OpenSSL 库提供）的错误。它表示在下层网络连接之上叠加的高层级加密和验证层存在某种问题。此错误是*OSError* 的一个子类型。*SSLError* 实例的错误和消息是由 OpenSSL 库提供的。

在 3.3 版更改: *SSLError* 曾经是*socket.error* 的一个子类型。

library

一个字符串形式的助词符，用来指明发生错误的 OpenSSL 子模块，例如 `SSL`, `PEM` 或 `X509`。可能的取值范围依赖于 OpenSSL 的版本。

3.3 新版功能.

reason

一个字符串形式的助记符，用来指明发生错误的原因，例如 `CERTIFICATE_VERIFY_FAILED`。可能的取值范围依赖于 OpenSSL 的版本。

3.3 新版功能.

exception ssl.**SSLZeroReturnError**

SSLError 的一个子类，当尝试读取或写入且 SSL 连接已被完全关闭时会被引发。请注意这并不意味着下层的传输（读取 TCP）已被关闭。

3.3 新版功能.

exception ssl.**SSLWantReadError**

SSLError 的一个子类，当尝试读取或写入但，并在请求被满足之前还需要在下层的 TCP 传输上接收更多数据时会被*非阻塞型 SSL 套接字* 引发。

3.3 新版功能.

exception ssl.**SSLWantWriteError**

SSLError 的一个子类，当尝试读取或写入数据，但在请求被满足之前还需要在下层的 TCP 传输上发送更多数据时会被*非阻塞型 SSL 套接字* 引发。

3.3 新版功能.

exception ssl.**SSLSyscallError**

SSLError 的子类，当尝试在 SSL 套接字上执行操作时遇到系统错误时会被引发。不幸的是，没有简单的方式能检查原始 errno 编号。

3.3 新版功能.

exception ssl.**SSLEOFError**

SSLError 的子类，当 SSL 连接被突然终止时会被引发。通常，当遇到此错误时你不应再尝试重用下层的传输。

3.3 新版功能.

exception ssl.**CertificateError**
Raised to signal an error with a certificate (such as mismatching hostname). Certificate errors detected by OpenSSL, though, raise an *SSLError*.

套接字创建

The following function allows for standalone socket creation. Starting from Python 3.2, it can be more flexible to use *SSLContext.wrap_socket()* instead.

ssl.**wrap_socket**(*sock*, *keyfile=None*, *certfile=None*, *server_side=False*, *cert_reqs=CERT_NONE*, *ssl_version={see docs}*, *ca_certs=None*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *ciphers=None*)
接受一个*socket.socket* 的实例 sock，并返回一个*ssl.SSLSocket* 的实例，该类型是*socket.socket* 的子类型，它将下层的套接字包装在一个 SSL 上下文中。sock 必须是一个*SOCK_STREAM* 套接字；其他套接字类型不被支持。

For client-side sockets, the context construction is lazy; if the underlying socket isn't connected yet, the context construction will be performed after connect() is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the accept() method. *wrap_socket()* may raise *SSLError*.

The keyfile and certfile parameters specify optional files which contain a certificate to be used to identify the local side of the connection. See the discussion of 证书 for more information on how the certificate is stored in the certfile.

形参 server_side 是一个布尔值，它标明希望从该套接字获得服务器端行为还是客户端行为。

The parameter cert_reqs specifies whether a certificate is required from the other side of the connection, and whether it will be validated if provided. It must be one of the three values *CERT_NONE* (certificates ignored), *CERT_OPTIONAL* (not required, but validated if provided), or *CERT_REQUIRED* (required and validated). If the value of this parameter is not *CERT_NONE*, then the ca_certs parameter must point to a file of CA certificates.

The ca_certs file contains a set of concatenated "certification authority" certificates, which are used to validate certificates passed from the other end of the connection. See the discussion of 证书 for more information about how to arrange the certificates in this file.

The parameter ssl_version specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server's choice. Most of the versions are not interoperable with the other versions. If not specified, the default is *PROTOCOL_TLS*; it provides the most compatibility with other versions.

这个表显示了客户端（横向）的哪个版本能够连接服务器（纵向）的哪个版本。

client / **server**	**SSLv2**	**SSLv3**	**TLS**[3]	**TLSv1**	**TLSv1.1**	**TLSv1.2**
SSLv2	是	否	no[1]	否	否	否
SSLv3	否	是	no[2]	否	否	否
TLS (SSLv23)[3]	no[1]	no[2]	是	是	是	是
TLSv1	否	否	是	是	否	否
TLSv1.1	否	否	是	否	是	否
TLSv1.2	否	否	是	否	否	是

[3] TLS 1.3 协议在 OpenSSL >= 1.1.1 中设置*PROTOCOL_TLS* 时可用。没有专门针对 TLS 1.3 的 PROTOCOL 常量。
[1] *SSLContext* 默认设置*OP_NO_SSLv2* 以禁用 SSLv2。
[2] *SSLContext* 默认设置*OP_NO_SSLv3* 以禁用 SSLv3。

备注

注解： Which connections succeed will vary depending on the version of OpenSSL. For example, before OpenSSL 1.0.0, an SSLv23 client would always attempt SSLv2 connections.

The *ciphers* parameter sets the available ciphers for this SSL object. It should be a string in the OpenSSL cipher list format.

形参 `do_handshake_on_connect` 指明是否要在调用 `socket.connect()` 之后自动执行 SSL 握手，还是要通过发起调用*SSLSocket.do_handshake()* 方法让应用程序显式地调用它。显式地调用*SSLSocket.do_handshake()* 可给予程序对握手中所涉及的套接字 I/O 阻塞行为的控制。

形参 `suppress_ragged_eofs` 指明 `SSLSocket.recv()` 方法应当如何从连接的另一端发送非预期的 EOF 信号。如果指定为*True* (默认值)，它将返回正常的 EOF (空字节串对象) 来响应从下层套接字引发的非预期的 EOF 错误；如果指定为*False*，它将向调用方引发异常。

在 3.2 版更改: New optional argument *ciphers*.

上下文创建

便捷函数，可以帮助创建*SSLContext* 对象，用于常见的目的。

`ssl.`**`create_default_context`**(*purpose=Purpose.SERVER_AUTH*, *cafile=None*, *capath=None*, *cadata=None*)

返回一个新的*SSLContext* 对象，使用给定 *purpose* 的默认设置。该设置由*ssl* 模块选择，并且通常是代表一个比直接调用*SSLContext* 构造器时更高的安全等级。

cafile, *capath*, *cadata* 代表用于进行证书核验的可选受信任 CA 证书，与*SSLContext.load_verify_locations()* 的一致。如果三个参数均为*None*，此函数可以转而选择信任系统的默认 CA 证书。

设置包括: *PROTOCOL_TLS*, *OP_NO_SSLv2* 以及*OP_NO_SSLv3*, 具有不带 RC4 和不带无身份验证密码套件的高度加密密码套件。传入*SERVER_AUTH* 作为 *purpose* 会把*verify_mode* 设为*CERT_REQUIRED* 并且加载指定 CA 证书（当给出 *cafile*, *capath* 和 *cadata* 中的至少一个）或者使用*SSLContext.load_default_certs()* 来加载默认 CA 证书。

注解： 协议、选项、密码和其他设置可随时更改为更具约束性的值而无须事先弃用。这些值代表了兼容性和安全性之间的合理平衡。

如果你的应用需要特定的设置，你应当创建一个*SSLContext* 并自行应用设置。

注解： 如果你发现当某些较旧的客户端或服务器尝试与用此函数创建的*SSLContext* 进行连接时收到了报错提示 "Protocol or cipher suite mismatch"，这可能是因为它们只支持 SSL3.0 而它被此函数用*OP_NO_SSLv3* 排除掉了。SSL3.0 被广泛认为 完全不可用。如果你仍希望继续使用此函数但仍允许 SSL 3.0 连接，你可以使用以下代码重新启用它们:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

3.4 新版功能.

在 3.4.4 版更改: RC4 被从默认密码字符串中丢弃。

在 3.6 版更改: ChaCha20/Poly1305 被添加到默认密码字符串中。

3DES 被从默认密码字符串中丢弃。

随机生成

ssl.**RAND_bytes**(*num*)
返回 *num* 个高加密强度伪随机字节数据。如果 PRNG 未使用足够的数据作为随机种子或者如果当前 RAND 方法不支持该操作则会引发*SSLError*。*RAND_status()* 可被用来检查 PRNG 的状态而*RAND_add()* 可被用来为 PRNG 设置随机种子。

对于几乎所有应用程序都更推荐使用*os.urandom()*。

Read the Wikipedia article, Cryptographically secure pseudorandom number generator (CSPRNG), to get the requirements of a cryptographically generator.

3.3 新版功能.

ssl.**RAND_pseudo_bytes**(*num*)
返回 (bytes, is_cryptographic): bytes 是 *num* 个伪随机字节数据，如果所生成的字节数据为高加密强度则 is_cryptographic 为 `True`。如果当前 RAND 方法不支持此操作则会引发*SSLError*。

所生成的伪随机字节序列如果具有足够的长度则将会具有唯一性，并是并非不可预测。它们可被用于非加密目的以及加密协议中的特定目的，但通常不可被用于密钥生成等目的。

对于几乎所有应用程序都更推荐使用*os.urandom()*。

3.3 新版功能.

3.6 版后已移除: OpenSSL 已弃用了*ssl.RAND_pseudo_bytes()*，请改用*ssl.RAND_bytes()*。

ssl.**RAND_status**()
如果 SSL 伪随机数生成器已使用‘足够的’随机性作为种子则返回 `True`，否则返回 `False`。你可以使用*ssl.RAND_egd()* 和*ssl.RAND_add()* 来增加伪随机数生成器的随机性。

ssl.**RAND_egd**(*path*)
如果你在某处运行了一个熵收集守护程序（EGD），且 *path* 是向其打开的套接字连接路径名，此函数将从该套接字读取 256 个字节的随机性数据，并将其添加到 SSL 伪随机数生成器以增加所生成密钥的安全性。此操作通常只在没有更好随机性源的系统上才是必要的。

请查看 http://egd.sourceforge.net/ 或 http://prngd.sourceforge.net/ 来了解有关熵收集守护程序源的信息。

Availability: not available with LibreSSL and OpenSSL > 1.1.0

ssl.**RAND_add**(*bytes*, *entropy*)
将给定的 *bytes* 混合到 SSL 伪随机数生成器中。形参 *entropy* (float 类型) 是数据所包含的熵的下界 (因此你可以总是使用 `0.0`)。请查看 **RFC 1750** 了解有关熵源的更多信息。

在 3.5 版更改: 现在支持可写的*字节类对象*。

证书处理

ssl.**match_hostname**(*cert*, *hostname*)
验证 *cert* (使用*SSLSocket.getpeercert()* 所返回的已解码格式) 是否匹配给定的 *hostname*。所应用的规则是在 **RFC 2818**, **RFC 5280** 和 **RFC 6125** 中描述的检查 HTTPS 服务器身份的规则。除了 HTTPS，此函数还应当适用于各种基于 SSL 协议的服务器身份检查操作，例如 FTPS, IMAPS, POPS 等等。

失败时引发*CertificateError*。成功时此函数无返回值:

```
>>> cert = {'subject': ((('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

3.2 新版功能.

在 3.3.3 版更改: 此函数现在遵循 **RFC 6125**, 6.4.3 小节，它不会匹配多个通配符 (例如 `*.*.com` 或 `*a*.example.org`) 也不匹配国际化域名 (IDN) 片段内部的通配符。IDN A 标签例如 `www*.xn--pthon-kva.org` 仍然受支持，但 `x*.python.org` 不再能匹配 `xn--tda.python.org`。

在 3.5 版更改: 现在支持匹配存在于证书的 subjectAltName 字段中的 IP 地址。

`ssl.`**`cert_time_to_seconds`**(*cert_time*)
返回距离 Unix 纪元零时的秒数，给定的 `cert_time` 字符串代表来自证书的 “notBefore” 或 “notAfter” 日期值，采用 `"%b %d %H:%M:%S %Y %Z"` strptime 格式（C 区域）。

以下为示例代码:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

“notBefore” 或 “notAfter” 日期值必须使用 GMT (**RFC 5280**)。

在 3.5 版更改: 将输入时间解读为 UTC 时间，基于输入字符串中指明的 ‘GMT’ 时区。在之前使用的是本地时区。返回一个整数（不带输入格式中秒的分数部分）

`ssl.`**`get_server_certificate`**(*addr*, *ssl_version=PROTOCOL_TLS*, *ca_certs=None*)
Given the address `addr` of an SSL-protected server, as a (*hostname*, *port-number*) pair, fetches the server’s certificate, and returns it as a PEM-encoded string. If `ssl_version` is specified, uses that version of the SSL protocol to attempt to connect to the server. If `ca_certs` is specified, it should be a file containing a list of root certificates, the same format as used for the same parameter in *`wrap_socket()`*. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails.

在 3.3 版更改: 此函数现在是 IPv6 兼容的。-compatible.

在 3.5 版更改: 默认的 *ssl_version* 从*`PROTOCOL_SSLv3`* 改为*`PROTOCOL_TLS`* 以保证与现代服务器的最大兼容性。

`ssl.`**`DER_cert_to_PEM_cert`**(*DER_cert_bytes*)
根据给定的 DER 编码字节块形式的证书，返回同一证书的 PEM 编码字符串版本。

`ssl.`**`PEM_cert_to_DER_cert`**(*PEM_cert_string*)
根据给定的 ASCII PEM 字符串形式的证书，返回同一证书的 DER 编码字节序列。

`ssl.`**`get_default_verify_paths`**()
返回包含 OpenSSL 的默认 cafile 和 capath 的路径的命名元组。此路径与*`SSLContext.set_default_verify_paths()`* 所使用的相同。返回值是一个*named tuple* `DefaultVerifyPaths`:

- `cafile` - 解析出的 cafile 路径或者如果文件不存在则为 `None`,
- `capath` - 解析出的 capath 路径或者如果目录不存在则为 `None`,
- `openssl_cafile_env` - 指向一个 cafile 的 OpenSSL 环境键,

- openssl_cafile - 一个 cafile 的硬编码路径,
- openssl_capath_env - 指向一个 capath 的 OpenSSL 环境键,
- openssl_capath - 一个 capath 目录的硬编码路径

Availability: LibreSSL ignores the environment vars openssl_cafile_env and openssl_capath_env

3.4 新版功能.

ssl.**enum_certificates**(*store_name*)
从 Windows 的系统证书库中检索证书。*store_name* 可以是 CA, ROOT 或 MY 中的一个。Windows 也可能会提供额外的证书库。

此函数返回一个包含 (cert_bytes, encoding_type, trust) 元组的列表。encoding_type 指明 cert_bytes 的编码格式。它可以为 x509_asn 以表示 X.509 ASN.1 数据或是 pkcs_7_asn 以表示 PKCS#7 ASN.1 数据。trust 以 OIDS 集合的形式指明证书的目的，或者如果证书对于所有目的都可以信任则为 True。

示例:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

Availability: Windows.

3.4 新版功能.

ssl.**enum_crls**(*store_name*)
Windows 的系统证书库中检索 CRL。*store_name* 可以是 CA, ROOT 或 MY 中的一个。Windows 也可能会提供额外的证书库。

此函数返回一个包含 (cert_bytes, encoding_type, trust) 元组的列表。encoding_type 指明 cert_bytes 的编码格式。它可以为 x509_asn 以表示 X.509 ASN.1 数据或是 pkcs_7_asn 以表示 PKCS#7 ASN.1 数据。

Availability: Windows.

3.4 新版功能.

常量

所有常量现在都是*enum.IntEnum*或*enum.IntFlag*多项集的成员。

3.6 新版功能.

ssl.**CERT_NONE**
SSLContext.verify_mode 或*wrap_socket()* 的 cert_reqs 形参可能的取值。*PROTOCOL_TLS_CLIENT* 除外，这是默认的模式。对于客户端套接字，几乎任何证书都是可接受的。验证错误例如不受信任或过期的证书错误会被忽略并且不会中止 TLS/SSL 握手。

在服务器模式下，不会从客户端请求任何证书，因此客户端不会发送任何用于客户端证书身份验证的证书。

参见下文对于安全考量 的讨论。

ssl.**CERT_OPTIONAL**
*SSLContext.verify_mode*或*wrap_socket()*的 cert_reqs 形参可能的取值。*CERT_OPTIONAL* 具有与*CERT_REQUIRED* 相同的含义。对于客户端套接字推荐改用*CERT_REQUIRED*。

在服务器模式下，客户端证书请求会被发送给客户端。客户端可以忽略请求也可以发送一个证书以执行 TLS 客户端证书身份验证。如果客户端选择发送证书，则将对其执行验证。任何验证错误都将立即中止 TLS 握手。

使用此设置要求将一组有效的 CA 证书传递给*SSLContext.load_verify_locations()* 或是作为*wrap_socket()* 的 ca_certs 形参值。

ssl.**CERT_REQUIRED**
SSLContext.verify_mode 或*wrap_socket()* 的 cert_reqs 形参可能的取值。在此模式下，需要从套接字连接的另一端获取证书；如果未提供证书或验证失败则将引发*SSLError*。此模式 **不能**在客户端模式下对证书进行验证，因为它不会匹配主机名。*check_hostname* 也必须被启用以验证证书的真实性。*PROTOCOL_TLS_CLIENT* 会使用*CERT_REQUIRED* 并默认启用*check_hostname*。

对于服务器套接字，此模式会提供强制性的 TLS 客户端证书验证。客户端证书请求会被发送给客户端并且客户端必须提供有效且受信任的证书。

使用此设置要求将一组有效的 CA 证书传递给*SSLContext.load_verify_locations()* 或是作为*wrap_socket()* 的 ca_certs 形参值。

class ssl.**VerifyMode**
CERT_* 常量的*enum.IntEnum* 多项集。

3.6 新版功能.

ssl.**VERIFY_DEFAULT**
SSLContext.verify_flags 可能的取值。在此模式下，证书吊销列表（CRL）并不会被检查。OpenSSL 默认不要求也不验证 CRL。

3.4 新版功能.

ssl.**VERIFY_CRL_CHECK_LEAF**
Possible value for *SSLContext.verify_flags*. In this mode, only the peer cert is check but non of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert’s issuer (its direct ancestor CA). If no proper has been loaded *SSLContext.load_verify_locations*, validation will fail.

3.4 新版功能.

ssl.**VERIFY_CRL_CHECK_CHAIN**
SSLContext.verify_flags 可能的取值。在此模式下，会检查对等证书链中所有证书的 CRL。

3.4 新版功能.

ssl.**VERIFY_X509_STRICT**
SSLContext.verify_flags 可能的取值，用于禁用已损坏 X.509 证书的绕过操作。

3.4 新版功能.

ssl.**VERIFY_X509_TRUSTED_FIRST**
SSLContext.verify_flags 可能的取值。它指示 OpenSSL 在构建用于验证某个证书的信任链时首选受信任的证书。此旗标将默认被启用。

3.4.4 新版功能.

class ssl.**VerifyFlags**
VERIFY_* 常量的*enum.IntFlag* 多项集。

3.6 新版功能.

ssl.**PROTOCOL_TLS**
选择客户端和服务器均支持的最高协议版本。此选项名称并不准确，实际上“SSL”和“TLS”协议均可被选择。

3.6 新版功能.

ssl.**PROTOCOL_TLS_CLIENT**
像*PROTOCOL_TLS* 一样地自动协商最高协议版本，但是只支持客户端*SSLSocket* 连接。此协议默认会启用*CERT_REQUIRED* 和*check_hostname*。

3.6 新版功能.

ssl.**PROTOCOL_TLS_SERVER**
像*PROTOCOL_TLS* 一样地自动协商最高协议版本，但是只支持服务器*SSLSocket* 连接。

3.6 新版功能.

ssl.**PROTOCOL_SSLv23**
Alias for data:*PROTOCOL_TLS*.

3.6 版后已移除: 请改用*PROTOCOL_TLS*。

ssl.**PROTOCOL_SSLv2**
选择 SSL 版本 2 作为通道加密协议。

如果 OpenSSL 编译时附带了 OPENSSL_NO_SSL2 旗标则此协议将不可用。

警告: SSL 版本 2 并不安全。极不建议使用它。

3.6 版后已移除: OpenSSL 已经移除了对 SSLv2 的支持。

ssl.**PROTOCOL_SSLv3**
选择 SSL 版本 3 作为通道加密协议。

如果 OpenSSL 编译时使用了 OPENSSL_NO_SSLv3 旗标则此协议将不可用。

警告: SSL 版本 3 并不安全。极不建议使用它。

3.6 版后已移除: OpenSSL 已经弃用了所有带有特定版本号的协议。请改用默认协议*PROTOCOL_TLS* 并附带*OP_NO_SSLv3* 等旗标。

ssl.**PROTOCOL_TLSv1**
选择 TLS 版本 1.0 作为通道加密协议。

3.6 版后已移除: OpenSSL 已经弃用了所有带有特定版本号的协议。请改用默认协议*PROTOCOL_TLS* 并附带*OP_NO_SSLv3* 等旗标。

ssl.**PROTOCOL_TLSv1_1**
选择 TLS 版本 1.1 作为通道加密协议。仅适用于 openssl 版本 1.0.1+。

3.4 新版功能.

3.6 版后已移除: OpenSSL 已经弃用了所有带有特定版本号的协议。请改用默认协议*PROTOCOL_TLS* 并附带*OP_NO_SSLv3* 等旗标。

ssl.**PROTOCOL_TLSv1_2**
选译 TLS 版本 1.2 作为通道加密协议。这是最新的版本，也应是能提供最大保护的最佳选择，如果通信双方都支持它的话。仅适用于 openssl 版本 1.0.1+。

3.4 新版功能.

3.6 版后已移除: OpenSSL 已经弃用了所有带有特定版本号的协议。请改用默认协议*PROTOCOL_TLS* 并附带*OP_NO_SSLv3* 等旗标。

ssl.**OP_ALL**
对存在于其他 SSL 实现中的各种缺陷启用绕过操作。默认会设置此选项。没有必要设置与 OpenSSL 的 SSL_OP_ALL 常量同名的旗标。

3.2 新版功能.

ssl.**OP_NO_SSLv2**
阻止 SSLv2 连接。此选项仅可与*PROTOCOL_TLS*结合使用。它会阻止对等方选择 SSLv2 作为协议版本。

3.2 新版功能.

3.6 版后已移除: SSLv2 已被弃用。

ssl.**OP_NO_SSLv3**
阻止 SSLv3 连接。此选项仅可与*PROTOCOL_TLS*结合使用。它会阻止对等方选择 SSLv3 作为协议版本。

3.2 新版功能.

3.6 版后已移除: SSLv3 已被弃用

ssl.**OP_NO_TLSv1**
阻止 TLSv1 连接。此选项仅可与*PROTOCOL_TLS*结合使用。它会阻止对等方选择 TLSv1 作为协议版本。

3.2 新版功能.

ssl.**OP_NO_TLSv1_1**
阻止 TLSv1.1 连接。此选项仅可与*PROTOCOL_TLS*结合使用。它会阻止对等方选择 TLSv1.1 作为协议版本。仅适用于 openssl 版本 1.0.1+。

3.4 新版功能.

ssl.**OP_NO_TLSv1_2**
阻止 TLSv1.2 连接。此选项仅可与*PROTOCOL_TLS*结合使用。它会阻止对等方选择 TLSv1.2 作为协议版本。仅适用于 openssl 版本 1.0.1+。

3.4 新版功能.

ssl.**OP_NO_TLSv1_3**
阻止 TLSv1.3 连接。此选项仅可与*PROTOCOL_TLS*结合使用。它会阻止对等方选择 TLSv1.3 作为协议版本。TLS 1.3 适用于 OpenSSL 1.1.1 或更新的版本。当 Python 编译是基于较旧版本的 OpenSSL 时，该旗标默认为 *0*。

3.6.3 新版功能.

ssl.**OP_CIPHER_SERVER_PREFERENCE**
使用服务器的密码顺序首选项，而不是客户端的首选项。此选项在客户端套接字和 SSLv2 服务器套接字上无效。

3.3 新版功能.

ssl.**OP_SINGLE_DH_USE**
阻止对于单独的 SSL 会话重用相同的 DH 密钥。这会提升前向保密性但需要更多的计算资源。此选项仅适用于服务器套接字。

3.3 新版功能.

ssl.**OP_SINGLE_ECDH_USE**
阻止对于单独的 SSL 会话重用相同的 ECDH 密钥。这会提升前向保密性但需要更多的计算资源。此选项仅适用于服务器套接字。

3.3 新版功能.

ssl.**OP_ENABLE_MIDDLEBOX_COMPAT**
在 TLS 1.3 握手中发送虚拟更改密码规格（CCS）消息以使得 TLS 1.3 连接看起来更像是 TLS 1.2 连接。

此选项仅适用于 OpenSSL 1.1.1 及更新的版本。

3.6.7 新版功能.

ssl.**OP_NO_COMPRESSION**
在 SSL 通道上禁用压缩。这适用于应用协议支持自己的压缩方案的情况。

此选项仅适用于 OpenSSL 1.0.0 及更新的版本。

3.3 新版功能.

class ssl.**Options**
OP_* 常量的*enum.IntFlag* 多项集。

ssl.**OP_NO_TICKET**
阻止客户端请求会话凭据。

3.6 新版功能.

ssl.**HAS_ALPN**
OpenSSL 库是否具有对 **RFC 7301** 中描述的 应用层协议协商 TLS 扩展的内置支持。

3.5 新版功能.

ssl.**HAS_ECDH**
Whether the OpenSSL library has built-in support for Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

3.3 新版功能.

ssl.**HAS_SNI**
OpenSSL 库是否具有对 服务器名称提示扩展（在 **RFC 6066** 中定义）的内置支持。

3.2 新版功能.

ssl.**HAS_NPN**
Whether the OpenSSL library has built-in support for *Next Protocol Negotiation* as described in the NPN draft specification. When true, you can use the *SSLContext.set_npn_protocols()* method to advertise which protocols you want to support.

3.3 新版功能.

ssl.**HAS_TLSv1_3**
OpenSSL 库是否具有对 TLS 1.3 协议的内置支持。

3.6.3 新版功能.

ssl.**CHANNEL_BINDING_TYPES**
受支持的 TLS 通道绑定类型组成的列表。此列表中的字符串可被用作传给*SSLSocket.get_channel_binding()* 的参数。

3.3 新版功能.

ssl.**OPENSSL_VERSION**
解释器所加载的 OpenSSL 库的版本字符串:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

3.2 新版功能.

ssl.**OPENSSL_VERSION_INFO**
代表 OpenSSL 库的版本信息的五个整数所组成的元组:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

3.2 新版功能.

ssl.**OPENSSL_VERSION_NUMBER**
OpenSSL 库的原始版本号，以单个整数表示:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

3.2 新版功能.

ssl.**ALERT_DESCRIPTION_HANDSHAKE_FAILURE**
ssl.**ALERT_DESCRIPTION_INTERNAL_ERROR**
ALERT_DESCRIPTION_*
来自 **RFC 5246** 等文档的警报描述。IANA TLS Alert Registry 中包含了这个列表及对定义其含义的 RFC 引用。

被用作*SSLContext.set_servername_callback()* 中的回调函数的返回值。

3.4 新版功能.

class ssl.**AlertDescription**
ALERT_DESCRIPTION_* 常量的*enum.IntEnum* 多项集。

3.6 新版功能.

Purpose.**SERVER_AUTH**
create_default_context() 和*SSLContext.load_default_certs()* 的选项值。这个值表明此上下文可以被用来验证 Web 服务器（因此，它将被用来创建客户端套接字）。

3.4 新版功能.

Purpose.**CLIENT_AUTH**
create_default_context() 和*SSLContext.load_default_certs()* 的选项值。这个值表明此上下文可以被用来验证 Web 客户端（因此，它将被用来创建服务器端套接字）。

3.4 新版功能.

class ssl.**SSLErrorNumber**
SSL_ERROR_* 常量的*enum.IntEnum* 多项集。

3.6 新版功能.

18.2.2 SSL 套接字

class ssl.**SSLSocket**(*socket.socket*)
SSL 套接字提供了套接字对象 的下列方法:

- *accept()*
- *bind()*
- *close()*
- *connect()*
- *detach()*
- *fileno()*
- *getpeername()*, *getsockname()*
- *getsockopt()*, *setsockopt()*

- *gettimeout()*, *settimeout()*, *setblocking()*
- *listen()*
- *makefile()*
- *recv()*, *recv_into()* (but passing a non-zero `flags` argument is not allowed)
- *send()*, *sendall()* (with the same limitation)
- *sendfile()* (but *os.sendfile* will be used for plain-text sockets only, else *send()* will be used)
- *shutdown()*

但是，由于 SSL（和 TLS）协议在 TCP 之上具有自己的框架，因此 SSL 套接字抽象在某些方面可能与常规的 OS 层级套接字存在差异。特别是要查看非阻塞型套接字说明。

Usually, *SSLSocket* are not created directly, but using the *SSLContext.wrap_socket()* method.

在 3.5 版更改: 新增了 `sendfile()` 方法。

在 3.5 版更改: `shutdown()` 不会在每次接收或发送字节数据后重置套接字超时。现在套接字超时为关闭的最大总持续时间。

3.6 版后已移除: 直接创建*SSLSocket* 实例的做法已被弃用，请使用*SSLContext.wrap_socket()* 来包装套接字。

SSL 套接字还具有下列方法和属性:

`SSLSocket.`**`read`**(*len=1024*, *buffer=None*)

从 SSL 套接字读取至多 *len* 个字节的数据并将结果作为 `bytes` 实例返回。如果指定了 *buffer*，则改为读取到缓冲区，并返回所读取的字节数。

如果套接字为非阻塞型 则会引发*SSLWantReadError* 或*SSLWantWriteError* 且读取将阻塞。

由于在任何时候重新协商都是可能的，因此调用*read()* 也可能导致写入操作。

在 3.5 版更改: 套接字超时在每次接收或发送字节数据后不会再被重置。现在套接字超时为读取至多 *len* 个字节数据的最大总持续时间。

3.6 版后已移除: 请使用 `recv()` 来代替*read()*。

`SSLSocket.`**`write`**(*buf*)

将 *buf* 写入到 SSL 套接字并返回所写入的字节数。*buf* 参数必须为支持缓冲区接口的对象。

如果套接字为非阻塞型 则会引发*SSLWantReadError* 或*SSLWantWriteError* 且读取将阻塞。

由于在任何时候重新协商都是可能的，因此调用*write()* 也可能导致读取操作。

在 3.5 版更改: 套接字超时在每次接收或发送字节数据后不会再被重置。现在套接字超时为写入 *buf* 的最大总持续时间。

3.6 版后已移除: 请使用 `send()` 来代替*write()*。

注解: *read()* 和*write()* 方法是读写未加密的应用级数据，并将其解密/加密为带加密的线路级数据的低层级方法。这些方法需要有激活的 SSL 连接，即握手已完成而*SSLSocket.unwrap()* 尚未被调用。

通常你应当使用套接字 API 方法例如*recv()* 和*send()* 来代替这些方法。

`SSLSocket.`**`do_handshake`**()

执行 SSL 设置握手。

在 3.4 版更改: 当套接字的*context* 的*check_hostname* 属性为真值时此握手方法还会执行*match_hostname()*。

在 3.5 版更改: 套接字超时在每次接收或发送字节数据时不会再被重置。现在套接字超时为握手的最大总持续时间。

SSLSocket.**getpeercert**(*binary_form=False*)

如果连接另一端的对等方没有证书，则返回 None。如果 SSL 握手还未完成，则会引发*ValueError*。

如果 binary_form 形参为*False*，并且从对等方接收到了证书，此方法将返回一个*dict* 实例。如果证书未通过验证，则字典将为空。如果证书通过验证，它将返回由多个密钥组成的字典，其中包括 subject (证书颁发给的主体) 和 issuer (颁发证书的主体)。如果证书包含一个 *Subject Alternative Name* 扩展的实例 (see **RFC 3280**)，则字典中还将有一个 subjectAltName 键。

subject 和 issuer 字段都是包含在证书中相应字段的数据结构中给出的相对专有名称（RDN）序列的元组，每个 RDN 均为 name-value 对的序列。这里是一个实际的示例:

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

注解: 要验证特定服务的证书，你可以使用*match_hostname()* 函数。

如果 binary_form 形参为*True*，并且提供了证书，此方法会将整个证书的 DER 编码形式作为字节序列返回，或者如果对等方未提供证书则返回*None*。对等方是否提供证书取决于 SSL 套接字的角色:

- 对于客户端 SSL 套接字，服务器将总是提供证书，无论是否需要进行验证;
- 对于服务器 SSL 套接字，客户端将仅在服务器要求时才提供证书；因此如果你使用了*CERT_NONE* (而不是*CERT_OPTIONAL* 或*CERT_REQUIRED*) 则*getpeercert()* 将返回*None*。

在 3.2 版更改: 返回的字典包括额外的条目例如 issuer 和 notBefore。

在 3.4 版更改: 如果握手未完成则会引发*ValueError*。返回的字典包括额外的 X509v3 扩展条目例如 crlDistributionPoints, caIssuers 和 OCSP URI。

SSLSocket.**cipher**()

返回由三个值组成的元组，其中包含所使用的密码名称，定义其使用方式的 SSL 协议，以及所使用的加密比特位数。如果尚未建立连接，则返回 None。

SSLSocket.**shared_ciphers**()

返回在握手期间由客户端共享的密码列表。所返回列表的每个条目都是由三个值组成的元组，其中包括密码名称，定义其使用方式的 SSL 协议版本，以及密码所使用的加密比特位数。如果尚未建立连接或套接字为客户端套接字则*shared_ciphers()* 将返回 None。

3.5 新版功能.

SSLSocket.**compression**()
以字符串形式返回所使用的压缩算法，或者如果连接没有使用压缩则返回 None。

如果高层级的协议支持自己的压缩机制，你可以使用*OP_NO_COMPRESSION* 来禁用 SSL 层级的压缩。

3.3 新版功能.

SSLSocket.**get_channel_binding**(*cb_type="tls-unique"*)
为当前连接获取字节串形式的通道绑定数据。如果尚未连接或握手尚未完成则返回 None。

cb_type 形参允许选择需要的通道绑定类型。有效的通道绑定类型在*CHANNEL_BINDING_TYPES* 列表中列出。目前只支持由 **RFC 5929** 所定义的‘tls-unique’通道绑定。如果请求了一个不受支持的通道绑定类型则将引发*ValueError*。

3.3 新版功能.

SSLSocket.**selected_alpn_protocol**()
返回在 TLS 握手期间所选择的协议。如果*SSLContext.set_alpn_protocols()* 未被调用，如果另一方不支持 ALPN，如果此套接字不支持任何客户端所用的协议，或者如果握手尚未发生，则将返回 None。

3.5 新版功能.

SSLSocket.**selected_npn_protocol**()
返回在 Return the higher-level protocol that was selected during the TLS/SSL 握手期间所选择的高层级协议。如果*SSLContext.set_npn_protocols()* 未被调用，或者如果另一方不支持 NPN，或者如果握手尚未发生，则将返回 None。

3.3 新版功能.

SSLSocket.**unwrap**()
执行 SSL 关闭握手，这会从下层的套接字中移除 TLS 层，并返回下层的套接字对象。这可被用来通过一个连接将加密操作转为非加密。返回的套接字应当总是被用于同连接另一方的进一步通信，而不是原始的套接字。

SSLSocket.**verify_client_post_handshake**()
向一个 TLS 1.3 客户端请求握手后身份验证(PHA)。只有在初始 TLS 握手之后且双方都启用了 PHA 的情况下才能为服务器端套接字的 TLS 1.3 连接启用 PHA，参见*SSLContext.post_handshake_auth*。

此方法不会立即执行证书交换。服务器端会在下一次写入事件期间发送 CertificateRequest 并期待客户端在下一次读取事件期间附带证书进行响应。

如果有任何前置条件未被满足（例如非 TLS 1.3，PHA 未启用），则会引发*SSLError*。

3.6.7 新版功能.

注解: 仅在 OpenSSL 1.1.1 且 TLS 1.3 被启用时可用。没有 TLS 1.3 支持，此方法将引发*NotImplementedError*。

SSLSocket.**version**()
以字符串形式返回由连接协商确定的实际 SSL 协议版本，或者如果未建立安全连接则返回 None。在撰写本文档时，可能的返回值包括 "SSLv2", "SSLv3", "TLSv1", "TLSv1.1" 和 "TLSv1.2"。最新的 OpenSSL 版本可能会定义更多的返回值。

3.5 新版功能.

SSLSocket.**pending**()
返回在连接上等待被读取的已解密字节数。

SSLSocket.**context**
The *SSLContext* object this SSL socket is tied to. If the SSL socket was created using the top-level

wrap_socket() function (rather than *SSLContext.wrap_socket()*), this is a custom context object created for this SSL socket.

3.2 新版功能.

SSLSocket.**server_side**
一个布尔值，对于服务器端套接字为 True 而对于客户端套接字则为 False。

3.2 新版功能.

SSLSocket.**server_hostname**
服务器的主机名: *str* 类型，对于服务器端套接字或者如果构造器中未指定主机名则为 None。

3.2 新版功能.

SSLSocket.**session**
用于 SSL 连接的*SSLSession*。该会话将在执行 TLS 握手后对客户端和服务器端套接字可用。对于客户端套接字该会话可以在调用*do_handshake()* 之前被设置以重用一个会话。

3.6 新版功能.

SSLSocket.**session_reused**
3.6 新版功能.

18.2.3 SSL 上下文

3.2 新版功能.

SSL 上下文可保存各种比单独 SSL 连接寿命更长的数据，例如 SSL 配置选项，证书和私钥等。它还可为服务器端套接字管理缓存，以加快来自相同客户端的重复连接。

class ssl.**SSLContext**(*protocol=PROTOCOL_TLS*)
Create a new SSL context. You may pass *protocol* which must be one of the PROTOCOL_* constants defined in this module. *PROTOCOL_TLS* is currently recommended for maximum interoperability and default value.

参见:

create_default_context() 让*ssl* 为特定目标选择安全设置。

在 3.6 版更改: 上下文会使用安全默认值来创建。默认设置的选项有*OP_NO_COMPRESSION*, *OP_CIPHER_SERVER_PREFERENCE*, *OP_SINGLE_DH_USE*, *OP_SINGLE_ECDH_USE*, *OP_NO_SSLv2* (except for *PROTOCOL_SSLv2*) 和*OP_NO_SSLv3* (except for *PROTOCOL_SSLv3*)。初始密码集列表只包含 HIGH 密码，不包含 NULL 密码和 MD5 密码 (*PROTOCOL_SSLv2* 除外)。

SSLContext 对象具有以下方法和属性:

SSLContext.**cert_store_stats**()
获取以字典表示的有关已加载的 X.509 证书数量，被标记为 CA 证书的 X.509 证书数量以及证书吊销列表的的统计信息。

具有一个 CA 证书和一个其他证书的上下文示例:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

3.4 新版功能.

SSLContext.**load_cert_chain**(*certfile*, *keyfile=None*, *password=None*)
加载一个私钥及对应的证书。*certfile* 字符串必须为以 PEM 格式表示的单个文件路径，该文件中包含证书以及确立证书真实性所需的任意数量的 CA 证书。如果存在 *keyfile* 字符串，它必须指向一个包含私钥的文件。否则私钥也将从 *certfile* 中提取。请参阅证书 中的讨论来了解有关如何将证书存储至 *certfile* 的更多信息。

password 参数可以是一个函数，调用时将得到用于解密私钥的密码。它在私钥被加密且需要密码时才会被调用。它调用时将不带任何参数，并且应当返回一个字符串、字节串或字节数组。如果返回值是一个字符串，在用它解密私钥之前它将以 UTF-8 进行编码。或者也可以直接将字符串、字节串或字节数组值作为 *password* 参数提供。如果私钥未被加密且不需要密码则它将被忽略。

如果未指定 *password* 参数且需要一个密码，将会使用 OpenSSL 内置的密码提示机制来交互式地提示用户输入密码。

如果私钥不能匹配证书则会引发*SSLError*。

在 3.3 版更改: 新增可选参数 *password*。

SSLContext.**load_default_certs**(*purpose=Purpose.SERVER_AUTH*)

从默认位置加载一组默认的“证书颁发机构”(CA) 证书。在 Windows 上它将从 CA 和 ROOT 系统存储中加载 CA 证书。在其他系统上它会调用*SSLContext.set_default_verify_paths()*。将来该方法也可能会从其他位置加载 CA 证书。

purpose 旗标指明要加载哪一类 CA 证书。默认设置*Purpose.SERVER_AUTH* 加载被标记且被信任用于 TLS Web 服务器验证（客户端套接字）的证书。*Purpose.CLIENT_AUTH* 则加载用于在服务器端进行客户端证书验证的 CA 证书。

3.4 新版功能.

SSLContext.**load_verify_locations**(*cafile=None*, *capath=None*, *cadata=None*)

当*verify_mode* 不为*CERT_NONE* 时加载一组用于验证其他对等方证书的“证书颁发机构”(CA) 证书。必须至少指定 *cafile* 或 *capath* 中的一个。

此方法还可加载 PEM 或 DER 格式的证书吊销列表 (CRL)，为此必须正确配置*SSLContext.verify_flags*。

如果存在 *cafile* 字符串，它应为 PEM 格式的级联 CA 证书文件的路径。请参阅证书 中的讨论来了解有关如何处理此文件中的证书的更多信息。

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an OpenSSL specific layout.

如果存在 *cadata* 对象，它应为一个或多个 PEM 编码的证书的 ASCII 字符串或者 DER 编码的证书的*bytes-like object*。与 *capath* 一样 PEM 编码的证书之外的多余行会被忽略，但至少要有一个证书。

在 3.4 版更改: 新增可选参数 *cadata*

SSLContext.**get_ca_certs**(*binary_form=False*)

获取已离开法人“证书颁发机构”(CA) 证书列表。如果 binary_form 形参为*False* 则每个列表条目都是一个类似于*SSLSocket.getpeercert()* 输出的字典。在其他情况下此方法将返回一个 DER 编码的证书的列表。返回的列表不包含来自 *capath* 的证书，除非 SSL 连接请求并加载了一个证书。

注解: capath 目录中的证书不会被加载，除非它们已至少被使用过一次。

3.4 新版功能.

SSLContext.**get_ciphers**()

获取已启用密码的列表。该列表将按密码的优先级排序。参见*SSLContext.set_ciphers()*。

示例:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()  # OpenSSL 1.0.x
[{'alg_bits': 256,
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
```

(下页继续)

(续上页)

```
                'Enc=AESGCM(256) Mac=AEAD',
  'id': 50380848,
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 256},
 {'alg_bits': 128,
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                'Enc=AESGCM(128) Mac=AEAD',
  'id': 50380847,
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 128}]
```

在 OpenSSL 1.1 及更新的版本中密码字典会包含额外的字段:

```
>>> ctx.get_ciphers()  # OpenSSL 1.1+
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'digest': None,
  'id': 50380847,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1.2',
  'strength_bits': 128,
  'symmetric': 'aes-128-gcm'}]
```

Availability: OpenSSL 1.0.2+

3.6 新版功能.

SSLContext.**set_default_verify_paths**()

从构建 OpenSSL 库时定义的文件系统路径中加载一组默认的“证书颁发机构”(CA) 证书。不幸的是，没有一种简单的方式能知道此方法是否执行成功：如果未找到任何证书也不会返回错误。不过，当 OpenSSL 库是作为操作系统的一部分被提供时，它的配置应当是正确的。

SSLContext.**set_ciphers**(*ciphers*)

为使用此上下文创建的套接字设置可用密码。它应当为 OpenSSL 密码列表格式 的字符串。如果没有可被选择的密码（由于编译时选项或其他配置禁止使用所指定的任何密码），则将引发*SSLError*。

注解: 在连接后，SSL 套接字的*SSLSocket.cipher()* 方法将给出当前所选择的密码。

在默认情况下 OpenSSL 1.1.1 会启用 TLS 1.3 密码套件。该套件不能通过*set_ciphers()* 来禁用。

SSLContext.**set_alpn_protocols**(*protocols*)
指定在 SSL/TLS 握手期间套接字应当通告的协议。它应为由 ASCII 字符串组成的列表，例如 ['http/1.1', 'spdy/2']，按首选顺序排列。协议的选择将在握手期间发生，并依据 **RFC 7301** 来执行。在握手成功后，*SSLSocket.selected_alpn_protocol()* 方法将返回已达成一致的协议。

This method will raise *NotImplementedError* if *HAS_ALPN* is False.

当双方都支持 ALPN 但不能就协议达成一致时 OpenSSL 1.1.0 至 1.1.0e 将中止并引发*SSLError*。1.1.0f+ 的行为类似于 1.0.2，*SSLSocket.selected_alpn_protocol()* 返回 None。

3.5 新版功能.

SSLContext.**set_npn_protocols**(*protocols*)
Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like ['http/1.1', 'spdy/2'], ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the NPN draft specification. After a successful handshake, the *SSLSocket.selected_npn_protocol()* method will return the agreed-upon protocol.

This method will raise *NotImplementedError* if *HAS_NPN* is False.

3.3 新版功能.

SSLContext.**set_servername_callback**(*server_name_callback*)
注册一个回调函数，当 TLS 客户端指定了一个服务器名称提示时，该回调函数将在 SSL/TLS 服务器接收到 TLS Client Hello 握手消息后被调用。服务器名称提示机制的定义见 **RFC 6066** section 3 - Server Name Indication。

Only one callback can be set per SSLContext. If *server_name_callback* is None then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function, *server_name_callback*, will be called with three arguments; the first being the *ssl.SSLSocket*, the second is a string that represents the server name that the client is intending to communicate (or *None* if the TLS Client Hello does not contain a server name) and the third argument is the original *SSLContext*. The server name argument is the IDNA decoded server name.

此回调的一个典型用法是将*ssl.SSLSocket* 的*SSLSocket.context* 属性修改为一个*SSLContext* 类型的新对象，该对象代表与服务器相匹配的证书链。

由于 TLS 连接处于早期协商阶段，因此仅能使用有限的方法和属性例如*SSLSocket.selected_alpn_protocol()* 和*SSLSocket.context*。*SSLSocket.getpeercert()*，*SSLSocket.getpeercert()*，*SSLSocket.cipher()* 和 SSLSocket.compress() 方法要求 TLS 连接已经过 TLS Client Hello 因而将既不包含返回有意义的值，也不能安全地调用它们。

The *server_name_callback* function must return None to allow the TLS negotiation to continue. If a TLS failure is required, a constant *ALERT_DESCRIPTION_** can be returned. Other return values will result in a TLS fatal error with *ALERT_DESCRIPTION_INTERNAL_ERROR*.

If there is an IDNA decoding error on the server name, the TLS connection will terminate with an *ALERT_DESCRIPTION_INTERNAL_ERROR* fatal TLS alert message to the client.

If an exception is raised from the *server_name_callback* function the TLS connection will terminate with a fatal TLS alert message *ALERT_DESCRIPTION_HANDSHAKE_FAILURE*.

如果 OpenSSL library 库在构建时定义了 OPENSSL_NO_TLSEXT 则此方法将返回*NotImplementedError*。

3.4 新版功能.

SSLContext.**load_dh_params**(*dhfile*)
加载密钥生成参数用于 Diffie-Hellman (DH) 密钥交换。使用 DH 密钥交换能以消耗（服务器和客户端

的）计算资源为代价提升前向保密性。*dhfile* 参数应当为指向一个包含 PEM 格式的 DH 形参的文件的路径。

此设置不会应用于客户端套接字。你还可以使用*OP_SINGLE_DH_USE* 选项来进一步提升安全性。

3.3 新版功能.

SSLContext.**set_ecdh_curve**(*curve_name*)
为基于椭圆曲线的 Elliptic Curve-based Diffie-Hellman (ECDH) 密钥交换设置曲线名称。ECDH 显著快于常规 DH 同时据信同样安全。*curve_name* 形参应为描述某个知名椭圆曲线的字符串，例如受到广泛支持的曲线 `prime256v1`。

此设置不会应用于客户端套接字。你还可以使用*OP_SINGLE_ECDH_USE* 选项来进一步提升安全性。

如果*HAS_ECDH* 为 `False` 则此方法将不可用。

3.3 新版功能.

参见:

SSL/TLS & Perfect Forward Secrecy Vincent Bernat。

SSLContext.**wrap_socket**(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)
Wrap an existing Python socket *sock* and return an *SSLSocket* object. *sock* must be a *SOCK_STREAM* socket; other socket types are unsupported.

The returned SSL socket is tied to the context, its settings and certificates. The parameters *server_side*, *do_handshake_on_connect* and *suppress_ragged_eofs* have the same meaning as in the top-level *wrap_socket()* function.

在客户端连接上，可选形参 *server_hostname* 指定所要连接的服务的主机名。这允许单个服务器托管具有单独证书的多个基于 SSL 的服务，很类似于 HTTP 虚拟主机。如果 *server_side* 为真值则指定 *server_hostname* 将引发*ValueError*。

session，参见*session*。

在 3.5 版更改: 总是允许传送 server_hostname，即使 OpenSSL 没有 SNI。

在 3.6 版更改: 增加了 *session* 参数。

SSLContext.**wrap_bio**(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)
Create a new *SSLObject* instance by wrapping the BIO objects *incoming* and *outgoing*. The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

server_side, *server_hostname* 和 *session* 形参具有与*SSLContext.wrap_socket()* 中相同的含义。

在 3.6 版更改: 增加了 *session* 参数。

SSLContext.**session_stats**()
获取由此上下文所创建或管理的 SSL 会话的相关统计信息。返回将每个 信息片 的名称映射到其数字值的字典。例如，以下是自上下文被创建以来会话缓存中命中和未命中的总数:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.**check_hostname**
Whether to match the peer cert's hostname with *match_hostname()* in *SSLSocket.do_handshake()*. The context's *verify_mode* must be set to *CERT_OPTIONAL* or *CERT_REQUIRED*, and you must pass *server_hostname* to *wrap_socket()* in order to match the hostname.

示例:

```
import socket, ssl

context = ssl.SSLContext()
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

3.4 新版功能.

注解： 此特性要求 OpenSSL 0.9.8f 或更新的版本。

SSLContext.**options**

一个代表此上下文中所启用的 SSL 选项集的整数。默认值为*OP_ALL*，但你也可以通过在选项间进行 OR 运算来指定其他选项例如*OP_NO_SSLv2*。

注解： With versions of OpenSSL older than 0.9.8m, it is only possible to set options, not to clear them. Attempting to clear an option (by resetting the corresponding bits) will raise a `ValueError`.

在 3.6 版更改: *SSLContext.options* 返回*Options* 旗标:

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

SSLContext.**post_handshake_auth**

启用 TLS 1.3 握手后客户端身份验证。握手后验证默认是被禁用的，服务器只能在初始握手期间请求 TLS 客户端证书。当启用时，服务器可以在握手之后的任何时候请求 TLS 客户端证书。

当在客户端套接字上启用时，客户端会向服务器发信号说明它支持握手后身份验证。

当在服务器端套接字上启用时，*SSLContext.verify_mode* 也必须被设为*CERT_OPTIONAL* 或*CERT_REQUIRED*。实际的客户端证书交换会被延迟直至*SSLSocket.verify_client_post_handshake()* 被调用并执行了一些 I/O 操作后再进行。

3.6.7 新版功能.

注解： 仅在 OpenSSL 1.1.1 且 TLS 1.3 被启用时可用。如果没有 TLS 1.3 支持，该属性值将为 None 且不可被更改

SSLContext.**protocol**

构造上下文时所选择的协议版本。这个属性是只读的。

SSLContext.**verify_flags**

证书验证操作的旗标。你可以通过对*VERIFY_CRL_CHECK_LEAF* 等值执行 OR 运算来设置组合旗标。在默认情况下 OpenSSL 不会要求也不会验证证书吊销列表（CRL）。仅在 openssl 版本 0.9.8+ 上可用。

3.4 新版功能.

在 3.6 版更改: *SSLContext.verify_flags* 返回*VerifyFlags* 旗标:

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.**verify_mode**

是否要尝试验证其他对等方的证书以及如果验证失败应采取何种行为。该属性值必须为*CERT_NONE*, *CERT_OPTIONAL* 或*CERT_REQUIRED* 之一。

在 3.6 版更改: *SSLContext.verify_mode* 返回*VerifyMode* 枚举:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

18.2.4 证书

总的来说证书是公钥/私钥系统的一个组成部分。在这个系统中，每个 *主体* (可能是一台机器、一个人或者一个组织) 都会分配到唯一的包含两部分的加密密钥。一部分密钥是公开的，称为 *公钥*；另一部分密钥是保密的，称为 *私钥*。这两个部分是互相关联的，就是说如果你用其中一个部分来加密一条消息，你将能用并且 **只能**用另一个部分来解密它。

在一个证书中包含有两个主体的相关信息。它包含 *目标方* 的名称和目标方的公钥。它还包含由第二个主体 *颁发方* 所发布的声明：目标方的身份与他们所宣称的一致，包含的公钥也确实是目标方的公钥。颁发方的声明使用颁发方的私钥进行签名，该私钥的内容只有颁发方自己才知道。但是，任何人都可以找到颁发方的公钥，用它来解密这个声明，并将其与证书中的其他信息进行比较来验证颁发方声明的真实性。证书还包含有关其有效期限的信息。这被表示为两个字段，即“notBefore”和“notAfter”。

在 Python 中应用证书时，客户端或服务器可以用证书来证明自己的身份。还可以要求网络连接的另一方提供证书，提供的证书可以用于验证以满足客户端或服务器的验证要求。如果验证失败，连接尝试可被设置为引发一个异常。验证是由下层的 OpenSSL 框架来自动执行的；应用程序本身不必关注其内部的机制。但是应用程序通常需要提供一组证书以允许此过程的发生。

Python 使用文件来包含证书。它们应当采用“PEM”格式 (参见 **RFC 1422**)，这是一种带有头部行和尾部行的 base-64 编码包装形式:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

证书链

包含证书的 Python 文件可以包含一系列的证书，有时被称为 *证书链*。这个证书链应当以“作为”客户端或服务器的主体的专属证书打头，然后是证书颁发方的证书，然后是 *上述* 证书的颁发方的证书，证书链就这样不断上溯直到你得到一个 *自签名* 的证书，即具有相同目标方和颁发方的证书，有时也称为 *根证书*。在证书文件中这些证书应当被拼接为一体。例如，假设我们有一个包含三个证书的证书链，以我们的服务器证书打头，然后是为我们的服务器证书签名的证书颁发机构的证书，最后是为证书颁发机构的证书颁发证书的机构的根证书:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
```

(下页继续)

(续上页)

```
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA 证书

如果你想要求对连接的另一方的证书进行验证，你必须提供一个 "CA 证书" 文件，其中包含了你愿意信任的每个颁发方的证书链。同样地，这个文件的内容就是这些证书链拼接在一起的结果。为了进行验证，Python 将使用它在文件中找到的第一个匹配的证书链。可以通过调用*SSLContext.load_default_certs()* 来使用系统平台的证书文件，这可以由*create_default_context()* 自动完成。

合并的密钥和证书

私钥往往与证书存储在相同的文件中；在此情况下，只需要将 certfile 形参传给*SSLContext.load_cert_chain()* 和*wrap_socket()*。如果私钥是与证书一起存储的，则它应当放在证书链的第一个证书之前:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

自签名证书

如果你准备创建一个提供 SSL 加密连接服务的服务器，你需要为该服务获取一份证书。有许多方式可以获取合适的证书，例如从证书颁发机构购买。另一种常见做法是生成自签名证书。生成自签名证书的最简单方式是使用 OpenSSL 软件包，代码如下所示:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

自签名证书的缺点在于它是它自身的根证书，因此不会存在于别人的已知（且信任的）根证书缓存当中。

18.2.5 例子

检测 SSL 支持

要检测一个 Python 安装版中是否带有 SSL 支持，用户代码应当使用以下例程:

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

客户端操作

这个例子创建了一个 SSL 上下文并使用客户端套接字的推荐安全设置，包括自动证书验证:

```
>>> context = ssl.create_default_context()
```

如果你喜欢自行调整安全设置，你可能需要从头创建一个上下文（但是请请注意避免不正确的设置）:

```
>>> context = ssl.SSLContext()
>>> context.verify_mode = ssl.CERT_REQUIRED
>>> context.check_hostname = True
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(这段代码假定你的操作系统将所有 CA 证书打包存放于 `/etc/ssl/certs/ca-bundle.crt`；如果不是这样，你将收到报错信息，必须修改此位置)

When you use the context to connect to a server, *CERT_REQUIRED* validates the server certificate: it ensures that the server certificate was signed with one of the CA certificates, and checks the signature for correctness:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

你可以随后获取该证书:

```
>>> cert = conn.getpeercert()
```

可视化检查显示证书能够证明目标服务 (即 HTTPS 主机 `www.python.org`) 的身份:

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt
↪',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
```

(下页继续)

(续上页)

```
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro,'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

现在 SSL 通道已建立并已验证了证书，你可以继续与服务器对话了:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

参见下文对于安全考量 的讨论。

服务器端操作

对于服务器操作，通常你需要在文件中存放服务器证书和私钥各一份。你将首先创建一个包含密钥和证书的上下文，这样客户端就能检查你的身份真实性。然后你将打开一个套接字，将其绑定到一个端口，在其上调用 `listen()`，并开始等待客户端连接:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)
```

当有客户端连接时，你将在套接字上调用 `accept()` 以从另一端获取新的套接字，并使用上下文的*SSLContext.wrap_socket()* 方法来为连接创建一个服务器端 SSL 套接字:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

随后你将从 `connstream` 读取数据并对其进行处理，直至你结束与客户端的会话（或客户端结束与你的会话）:

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

并返回至监听新的客户端连接（当然，真正的服务器应当会在单独的线程中处理每个客户端连接，或者将套接字设为非阻塞模式 并使用事件循环）。

18.2.6 关于非阻塞套接字的说明

在非阻塞模式下 SSL 套接字的行为与常规套接字略有不同。当使用非阻塞模式时，你需要注意下面这些事情:

- 如果一个 I/O 操作会阻塞，大多数*SSLSocket* 方法都将引发*SSLWantWriteError* 或*SSLWantReadError* 而非*BlockingIOError*。如果有必要在下层套接字上执行读取操作将引发*SSLWantReadError*，在下层套接字上执行写入操作则将引发*SSLWantWriteError*。请注意尝试 写入到 SSL 套接字可能需要先从下层套接字 读取，而尝试从 SSL 套接字 读取则可能需要先向下层套接字 写入。

  在 3.5 版更改: 在较早的 Python 版本中，`SSLSocket.send()` 方法会返回零值而非引发*SSLWantWriteError* 或*SSLWantReadError*。

- 调用*select()*将告诉你可以从 OS 层级的套接字读取（或向其写入），但这并不意味着在上面的 SSL 层有足够的数据。例如，可能只有部分 SSL 帧已经到达。因此，你必须准备好处理 `SSLSocket.recv()` 和 `SSLSocket.send()` 失败的情况，并在再次调用*select()*之后重新尝试。
- 相反地，由于 SSL 层具有自己的帧机制，一个 SSL 套接字可能仍有可读取的数据而*select()*并不知道这一点。因此，你应当先调用 `SSLSocket.recv()` 取走所有潜在的可用数据，然后只在必要时对*select()*调用执行阻塞。

  (当然，类似的保留规则在使用其他原语例如*poll()*，或*selectors*模块中的原语时也适用)
- SSL 握手本身将是非阻塞的: *SSLSocket.do_handshake()*方法必须不断重试直至其成功返回。下面是一个使用*select()*来等待套接字就绪的简短例子:

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

参见:

*asyncio*模块支持*非阻塞 SSL 套接字*并提供了更高层级的 API。它会使用*selectors*模块来轮询事件并处理*SSLWantWriteError*, *SSLWantReadError*和*BlockingIOError*等异常。它还会异步地执行 SSL 握手。

18.2.7 内存 BIO 支持

3.5 新版功能.

自从 SSL 模块在 Python 2.6 起被引入之后，*SSLSocket*类提供了两个互相关联但彼此独立的功能分块:

- SSL 协议处理
- 网络 IO

网络 IO API 与*socket.socket*所提供的功能一致，*SSLSocket*也是从那里继承而来的。这允许 SSL 套接字被用作常规套接字的替代，使得向现有应用程序添加 SSL 支持变得非常容易。

将 SSL 协议处理与网络 IO 结合使用通常都能运行良好，但在某些情况下则不能。此情况的一个例子是 async IO 框架，该框架要使用不同的 IO 多路复用模型而非 (基于就绪状态的) “在文件描述器上执行选择/轮询” 模型，该模型是*socket.socket*和内部 OpenSSL 套接字 IO 例程正常运行的假设前提。这种情况在该模型效率不高的 Windows 平台上最为常见。为此还提供了一个*SSLSocket*的简化形式，称为*SSLObject*。

class ssl.**SSLObject**

*SSLSocket*的简化形式，表示一个不包含任何网络 IO 方法的 SSL 协议实例。这个类通常由想要通过内存缓冲区为 SSL 实现异步 IO 的框架作者来使用。

这个类在低层级 SSL 对象上实现了一个接口，与 OpenSSL 所实现的类似。此对象会捕获 SSL 连接的状态但其本身不提供任何网络 IO。IO 需要通过单独的 “BIO” 对象来执行，该对象是 OpenSSL 的 IO 抽象层。

An *SSLObject* instance can be created using the *wrap_bio()* method. This method will create the *SSLObject* instance and bind it to a pair of BIOs. The *incoming* BIO is used to pass data from Python to the SSL protocol instance, while the *outgoing* BIO is used to pass data the other way around.

可以使用以下方法:

- *context*

- *server_side*
- *server_hostname*
- *session*
- *session_reused*
- *read()*
- *write()*
- *getpeercert()*
- *selected_npn_protocol()*
- *cipher()*
- *shared_ciphers()*
- *compression()*
- *pending()*
- *do_handshake()*
- *unwrap()*
- *get_channel_binding()*

与*SSLSocket* 相比，此对象缺少下列特性:

- 任何形式的网络 IO; `recv()` 和 `send()` 仅对下层的*MemoryBIO* 缓冲区执行读取和写入。
- 不存在 *do_handshake_on_connect* 机制。你必须总是手动调用*do_handshake()* 来开始握手操作。
- 不存在对 *suppress_ragged_eofs* 的处理。所有违反协议的文件结束条件将通过*SSLEOFError* 异常来报告。
- 方法*unwrap()* 的调用不返回任何东西，不会如 SSL 套接字那样返回下层的套接字。
- *server_name_callback* 回调被传给*SSLContext.set_servername_callback()* 时将获得一个*SSLObject* 实例而非*SSLSocket* 实例作为其第一个形参。

有关*SSLObject* 用法的一些说明:

- 在*SSLObject* 上的所有 IO 都是非阻塞的。这意味着例如*read()* 在其需要比 incoming BIO 可用的更多数据时将会引发*SSLWantReadError*。
- 不存在模块层级的 `wrap_bio()` 调用，就像*wrap_socket()* 那样。*SSLObject* 总是通过*SSLContext* 来创建。

SSLObject 会使用内存缓冲区与外部世界通信。*MemoryBIO* 类提供了可被用于此目的的内存缓冲区。它包装了一个 OpenSSL 内存 BIO (Basic IO) 对象:

class ssl.**MemoryBIO**

一个可被用来在 Python 和 SSL 协议实例之间传递数据的内存缓冲区。

pending

返回当前存在于内存缓冲区的字节数。

eof

一个表明内存 BIO 目前是否位于文件末尾的布尔值。

read(*n=-1*)

从内存缓冲区读取至多 *n* 个字节。如果 *n* 未指定或为负值，则返回全部字节数据。

write(*buf*)
将字节数据从 *buf* 写入到内存 BIO。*buf* 参数必须为支持缓冲区协议的对象。

返回值为写入的字节数，它总是与 *buf* 的长度相等。

write_eof()
将一个 EOF 标记写入到内存 BIO。在此方法被调用以后，再调用 *write()* 将是非法的。属性 *eof* will 在缓冲区当前的所有数据都被读取之后将变为真值。

18.2.8 SSL 会话

3.6 新版功能.

class ssl.**SSLSession**
session 所使用的会话对象。

id

time

timeout

ticket_lifetime_hint

has_ticket

18.2.9 安全考量

最佳默认值

针对 **客户端使用**，如果你对于安全策略没有任何特殊要求，则强烈推荐你使用 *create_default_context()* 函数来创建你的 SSL 上下文。它将加载系统的受信任 CA 证书，启用证书验证和主机名检查，并尝试合理地选择安全的协议和密码设置。

例如，以下演示了你应当如何使用 *smtplib.SMTP* 类来创建指向一个 SMTP 服务器的受信任且安全的连接:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

如果连接需要客户端证书，可使用 *SSLContext.load_cert_chain()* 来添加。

作为对比，如果你通过自行调用 *SSLContext* 构造器来创建 SSL 上下文，它默认将不会启用证书验证和主机名检查。如果你这样做，请阅读下面的段落以达到良好的安全级别。

手动设置

验证证书

当直接调用*SSLContext* 构造器时，默认会使用*CERT_NONE*。由于它不会验证对等方的身份真实性，因此是不安全的，特别是在客户端模式下，大多数时候你都希望能保证你所连接的服务器的身份真实性。因此，当处于客户端模式时，强烈推荐使用*CERT_REQUIRED*。但是，光这样还不够；你还必须检查服务器证书，这可以通过调用*SSLSocket.getpeercert()* 来获取并匹配目标服务。对于许多协议和应用来说，服务可通过主机名来标识；在此情况下，可以使用*match_hostname()* 函数。这种通用检测会在*SSLContext.check_hostname* 被启用时自动执行。

在服务器模式下，如果你想要使用 SSL 层来验证客户端（而不是使用更高层级的验证机制），你也必须要指定*CERT_REQUIRED* 并以类似方式检查客户端证书。

协议版本

SSL 版本 2 和 3 被认为是不安全的因而使用它们会有风险。如果你想要客户端和服务器之间有最大的兼容性，推荐使用*PROTOCOL_TLS_CLIENT* 或*PROTOCOL_TLS_SERVER* 作为协议版本。SSLv2 和 SSLv3 默认会被禁用。

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.options |= ssl.OP_NO_TLSv1
>>> client_context.options |= ssl.OP_NO_TLSv1_1
```

前面创建的 SSL 上下文将只允许 TLSv1.2 及更新版本（如果你的系统支持）的服务器连接。*PROTOCOL_TLS_CLIENT* 默认会使用证书验证和主机名检查。你必须将证书加载到上下文中。

密码选择

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the *SSLContext.set_ciphers()* method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL' s documentation about the cipher list format. If you want to check which ciphers are enabled by a given cipher list, use *SSLContext.get_ciphers()* or the `openssl ciphers` command on your system.

多进程

如果使用此模块作为多进程应用的一部分（例如使用*multiprocessing* 或*concurrent.futures* 模块），请注意 OpenSSL 的内部随机数字生成器并不能正确处理分支进程。应用程序必须修改父进程的 PRNG 状态，如果它们要使用任何包含*os.fork()* 的 SSL 特性的话。任何对*RAND_add()*, *RAND_bytes()* 或*RAND_pseudo_bytes()* 都可以。

18.2.10 LibreSSL 支持

LibreSSL 是 OpenSSL 1.0.1 的一个分支。ssl 模块包含对 LibreSSL 的有限支持。当 ssl 模块使用 LibreSSL 进行编译时某些特性将不可用。

- LibreSSL >= 2.6.1 不再支持 NPN。*SSLContext.set_npn_protocols()* 和*SSLSocket.selected_npn_protocol()* 方法将不可用。
- *SSLContext.set_default_verify_paths()* 会忽略环境变量 SSL_CERT_FILE 和 SSL_CERT_PATH，虽然*get_default_verify_paths()* 仍然支持它们。

参见:

Class *socket.socket* 下层*socket* 类的文档

SSL/TLS 高强度加密：概述 Apache HTTP Server 文档介绍

RFC 1422: 因特网电子邮件的隐私加强：第二部分：基于证书的密钥管理 Steve Kent

RFC 4086: 确保安全的随机性要求 Donald E., Jeffrey I. Schiller

RFC 5280: 互联网 X.509 公钥基础架构证书和证书吊销列表 (CRL) 配置文件 D. Cooper

RFC 5246: 传输层安全性 (TLS) 协议版本 1.2 T. Dierks et. al.

RFC 6066: 传输层安全性 (TLS) 的扩展 D. Eastlake

IANA TLS: 传输层安全性 (TLS) 的参数 IANA

RFC 7525: 传输层安全性 (TLS) 和数据报传输层安全性 (DTLS) 的安全使用建议 IETF

Mozilla 的服务器端 TLS 建议 Mozilla

18.3 select —Waiting for I/O 完成

该模块提供了对 select() 和 poll() 函数的访问，这些函数在大多数操作系统中是可用的。在 Solaris 及其衍生版本上可用 devpoll()，在 Linux 2.5+ 上可用 epoll()，在大多数 BSD 上可用 kqueue()。注意，在 Windows 上，本模块仅适用于套接字；在其他操作系统上，本模块也适用于其他文件类型（特别地，在 Unix 上也适用于管道）。本模块不能用于常规文件，不能检测出（自上次读取文件后）文件是否有新数据写入。

注解: *selectors* 模块是在*select* 模块原型的基础上进行高级且高效的 I/O 复用。推荐用户改用*selectors* 模块，除非用户希望对 OS 级的函数原型进行精确控制。

该模块定义以下内容：

exception select.**error**
一个被弃用的*OSError* 的别名。

在 3.3 版更改: 根据 **PEP 3151**，这个类是*OSError* 的别名。

select.**devpoll**()
（仅支持 Solaris 及其衍生版本）返回一个 /dev/poll 轮询对象，请参阅下方*/dev/poll 轮询对象* 获取 devpoll 对象所支持的方法。

devpoll() 对象与实例化时允许的文件描述符数量有关，如果在程序中降低了此数值，devpoll() 调用将失败。如果程序提高了此数值，devpoll() 可能会返回一个不完整的活动文件描述符列表。

新的文件描述符*non-inheritable*.

3.3 新版功能.

在 3.4 版更改: 新的文件描述符现在是不可继承的。

select.**epoll**(*sizehint=-1*, *flags=0*)
（仅支持 Linux 2.5.44 或更高版本）返回一个轮询对象，该对象可作为 I/O 事件的边缘触发或水平触发接口。

sizehint 指示 epoll 预计需要注册的事件数。它必须为正数，或为 *-1* 以使用默认值。它仅在 epoll_create1() 不可用的旧系统上有用，其他情况下它没有任何作用（尽管仍会检查其值）。

flags 已经弃用且完全被忽略。但是，如果提供该值，则它必须是 0 或 select.EPOLL_CLOEXEC，否则会抛出 OSError 异常。

请参阅下方*边缘触发和水平触发的轮询 (epoll) 对象* 获取 epoll 对象所支持的方法。

epoll 对象支持上下文管理器：当在 with 语句中使用时，新建的文件描述符会在运行至语句块结束时自动销毁。

新的文件描述符*non-inheritable*.

在 3.3 版更改: 增加了 *flags* 参数。

在 3.4 版更改: 增加了对 with 语句的支持。新的文件描述符现在是不可继承的。

3.4 版后已移除: *flags* 参数。现在默认采用 select.EPOLL_CLOEXEC 标志。使用*os.set_inheritable()* 来让文件描述符可继承。

select.**poll**()
（部分操作系统不支持）返回一个轮询对象，该对象支持注册和注销文件描述符，支持对描述符进行轮询以获取 I/O 事件。有关轮询对象支持的方法，请参阅下方*Poll 对象* 部分。

select.**kqueue**()
（仅支持 BSD）返回一个内核队列对象，请参阅下方*Kqueue 对象* 获取 kqueue 对象所支持的方法。

新的文件描述符*non-inheritable*.

在 3.4 版更改: 新的文件描述符现在是不可继承的。

select.**kevent**(*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*)
（仅支持 BSD）返回一个内核事件对象，请参阅下方*Kevent 对象* 获取 kevent 对象所支持的方法。

select.**select**(*rlist*, *wlist*, *xlist*[, *timeout*])
This is a straightforward interface to the Unix select() system call. The first three arguments are sequences of ‘waitable objects’: either integers representing file descriptors or objects with a parameterless method named *fileno()* returning such an integer:

- *rlist*：等待，直到有内容可以读取
- *wlist*：等待，直到可以开始写入
- *xlist*：等待“异常情况”（请参阅当前系统的手册，以获取哪些情况称为异常情况）

Empty sequences are allowed, but acceptance of three empty sequences is platform-dependent. (It is known to work on Unix but not on Windows.) The optional *timeout* argument specifies a time-out as a floating point number in seconds. When the *timeout* argument is omitted the function blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

返回值是三个列表，包含已就绪对象，返回的三个列表是前三个参数的子集。当超时时间已到且没有对象就绪时，返回三个空列表。

Among the acceptable object types in the sequences are Python *file objects* (e.g. sys.stdin, or objects returned by *open()* or *os.popen()*), socket objects returned by *socket.socket()*. You may also define a *wrapper*

class yourself, as long as it has an appropriate *fileno()* method (that really returns a file descriptor, not just a random integer).

注解： Windows 上不接受文件对象，但接受套接字。在 Windows 上，底层的 `select()` 函数由 WinSock 库提供，且不处理不是源自 WinSock 的文件描述符。

在 3.5 版更改: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出 *InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

`select.`**`PIPE_BUF`**

当一个管道已经被 *select()*、*poll()* 或本模块中的某个接口报告为可写入时，可以在不阻塞该管道的情况下写入的最小字节数。它不适用于套接字等其他类型的文件类对象。

This value is guaranteed by POSIX to be at least 512. Availability: Unix.

3.2 新版功能.

18.3.1 `/dev/poll` 轮询对象

Solaris 及其衍生版本具备 `/dev/poll`。`select()` 复杂度为 O（最高文件描述符），`poll()` 为 O（文件描述符数量），而 `/dev/poll` 为 O（活动的文件描述符）。

`/dev/poll` 的行为与标准 `poll()` 对象十分类似。

`devpoll.`**`close`**`()`

关闭轮询对象的文件描述符。

3.4 新版功能.

`devpoll.`**`closed`**

如果轮询对象已关闭，则返回 `True`。

3.4 新版功能.

`devpoll.`**`fileno`**`()`

返回轮询对象的文件描述符对应的数字。

3.4 新版功能.

`devpoll.`**`register`**`(`*fd*[, *eventmask*]`)`

在轮询对象中注册文件描述符。这样，将来调用 *poll()* 方法时将检查文件描述符是否有未处理的 I/O 事件。*fd* 可以是整数，也可以是带有 *fileno()* 方法的对象（该方法返回一个整数）。文件对象已经实现了 `fileno()`，因此它们也可以用作参数。

eventmask 是可选的位掩码，用于指定要检查的事件类型。这些常量与 `poll()` 对象所用的相同。本参数的默认值是常量 `POLLIN`、`POLLPRI` 和 `POLLOUT` 的组合。

> **警告：** 注册已注册过的文件描述符不会报错，但是结果是不确定的。正确的操作是先注销或直接修改它。与 `poll()` 相比，这是一个重要的区别。

`devpoll.`**`modify`**`(`*fd*[, *eventmask*]`)`

此方法先执行 *unregister()* 后执行 *register()*。直接执行此操作效率（稍微）高一些。

`devpoll.`**`unregister`**`(`*fd*`)`

删除轮询对象正在跟踪的某个文件描述符。与 *register()* 方法类似，*fd* 可以是整数，也可以是带有 *fileno()* 方法的对象（该方法返回一个整数）。

尝试删除从未注册过的文件描述符将被安全地忽略。

devpoll.**poll**([*timeout*])
轮询已注册的文件描述符的集合，并返回一个列表，列表可能为空，也可能有多个 (fd, event) 二元组，其中包含了要报告事件或错误的描述符。*fd* 是文件描述符，*event* 是一个位掩码，表示该描述符所报告的事件—POLLIN 表示可以读取，POLLOUT 表示该描述符可以写入，依此类推。空列表表示调用超时，没有任何文件描述符报告事件。如果指定了 *timeout*，它将指定系统等待事件时，等待多长时间后返回（以毫秒为单位）。如果 *timeout* 为空，-1 或 None，则本调用将阻塞，直到轮询对象发生事件为止。

在 3.5 版更改: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出 InterruptedError 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

18.3.2 边缘触发和水平触发的轮询 (epoll) 对象

http://linux.die.net/man/4/epoll

屏蔽事件

常数	含义
EPOLLIN	可读
EPOLLOUT	可写
EPOLLPRI	紧急数据读取
EPOLLERR	在关联的文件描述符上有错误情况发生
EPOLLHUP	关联的文件描述符已挂起
EPOLLET	设置触发方式为边缘触发，默认为水平触发
EPOLLONESHOT	设置 one-shot 模式。触发一次事件后，该描述符会在轮询对象内部被禁用。
EPOLLEXCLUSIVE	当已关联的描述符发生事件时，仅唤醒一个 epoll 对象。默认（如果未设置此标志）是唤醒所有轮询该描述符的 epoll 对象。
EPOLLRDHUP	流套接字的对侧关闭了连接或关闭了写入到一半的连接。
EPOLLRDNORM	Equivalent to EPOLLIN
EPOLLRDBAND	可以读取优先数据带。
EPOLLWRNORM	Equivalent to EPOLLOUT
EPOLLWRBAND	可以写入优先级数据。
EPOLLMSG	忽略

epoll.**close**()
关闭用于控制 epoll 对象的那个文件描述符。

epoll.**closed**
如果 epoll 对象已关闭，则返回 True。

epoll.**fileno**()
返回用于控制 epoll 对象的文件描述符对应的数字。

epoll.**fromfd**(*fd*)
根据给定的文件描述符创建 epoll 对象。

epoll.**register**(*fd*[, *eventmask*])
在 epoll 对象中注册一个文件描述符。

epoll.**modify**(*fd*, *eventmask*)
修改一个已注册的文件描述符。

epoll.**unregister**(*fd*)
从 epoll 对象中删除一个已注册的文件描述符。

epoll.**poll**(*timeout=-1*, *maxevents=-1*)
等待事件发生，timeout 是浮点数，单位为秒。

在 3.5 版更改: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出 *InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

18.3.3 Poll 对象

大多数 Unix 系统支持 poll() 系统调用，为服务器提供了更好的可伸缩性，使服务器可以同时服务于大量客户端。poll() 的伸缩性更好，因为该调用内部仅列出所关注的文件描述符，而 select() 会构造一个 bitmap，在其中将所关注的描述符所对应的 bit 打开，然后重新遍历整个 bitmap。因此 select() 复杂度是 O（最高文件描述符），而 poll() 是 O（文件描述符数量）。

poll.**register**(*fd*[, *eventmask*])
在轮询对象中注册文件描述符。这样，将来调用 *poll()* 方法时将检查文件描述符是否有未处理的 I/O 事件。*fd* 可以是整数，也可以是带有 *fileno()* 方法的对象（该方法返回一个整数）。文件对象已经实现了 fileno()，因此它们也可以用作参数。

eventmask 是可选的位掩码，用于指定要检查的事件类型，它可以是常量 POLLIN、POLLPRI 和 POLLOUT 的组合，如下表所述。如果未指定本参数，默认将会检查所有 3 种类型的事件。

常数	含义
POLLIN	有要读取的数据
POLLPRI	有紧急数据需要读取
POLLOUT	准备输出：写不会阻塞
POLLERR	某种错误条件
POLLHUP	挂起
POLLRDHUP	流套接字对等体关闭连接，或关闭写入一半连接
POLLNVAL	无效的请求：描述符未打开

注册已注册过的文件描述符不会报错，且等同于只注册一次该描述符。

poll.**modify**(*fd*, *eventmask*)
修改一个已注册的文件描述符，等同于 register(fd, eventmask)。尝试修改未注册的文件描述符会抛出 *OSError* 异常，错误码为 ENOENT。

poll.**unregister**(*fd*)
删除轮询对象正在跟踪的某个文件描述符。与 *register()* 方法类似，*fd* 可以是整数，也可以是带有 *fileno()* 方法的对象（该方法返回一个整数）。

尝试删除从未注册过的文件描述符会抛出 *KeyError* 异常。

poll.**poll**([*timeout*])
轮询已注册的文件描述符的集合，并返回一个列表，列表可能为空，也可能有多个 (fd, event) 二元组，其中包含了要报告事件或错误的描述符。*fd* 是文件描述符，*event* 是一个位掩码，表示该描述符所报告的事件—POLLIN 表示可以读取，POLLOUT 表示该描述符可以写入，依此类推。空列表表示调用超时，没有任何文件描述符报告事件。如果指定了 *timeout*，它将指定系统等待事件时，等待多长时间后返回（以毫秒为单位）。如果 *timeout* 为空、负数或 *None*，则本调用将阻塞，直到轮询对象发生事件为止。

在 3.5 版更改: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出 *InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

18.3.4 Kqueue 对象

kqueue.**close**()
关闭用于控制 kqueue 对象的文件描述符。

kqueue.**closed**
如果 kqueue 对象已关闭，则返回 True。

kqueue.**fileno**()
返回用于控制 epoll 对象的文件描述符对应的数字。

kqueue.**fromfd**(*fd*)
根据给定的文件描述符创建 kqueue 对象。

kqueue.**control**(*changelist*, *max_events*[, *timeout=None*]) → eventlist
Kevent 的低级接口

- changelist must be an iterable of kevent object or None
- max_events 必须是 0 或一个正整数。
- timeout in seconds (floats possible)

在 3.5 版更改: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出 *InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

18.3.5 Kevent 对象

https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.**ident**
用于区分事件的标识值。其解释取决于筛选器，但该值通常是文件描述符。在构造函数中，该标识值可以是整数或带有 *fileno()* 方法的对象。kevent 在内部存储整数。

kevent.**filter**
内核过滤器的名称。

常数	含义
KQ_FILTER_READ	获取描述符，并在有数据可读时返回
KQ_FILTER_WRITE	获取描述符，并在有数据可写时返回
KQ_FILTER_AIO	AIO 请求
KQ_FILTER_VNODE	当在 *fflag* 中监视的一个或多个请求事件发生时返回
KQ_FILTER_PROC	监视进程 ID 上的事件
KQ_FILTER_NETDEV	观察网络设备上的事件 [在 Mac OS X 上不可用]
KQ_FILTER_SIGNAL	每当监视的信号传递到进程时返回
KQ_FILTER_TIMER	建立一个任意的计时器

kevent.**flags**
筛选器操作。

常数	含义
KQ_EV_ADD	添加或修改事件
KQ_EV_DELETE	从队列中删除事件
KQ_EV_ENABLE	Permitscontrol() 返回事件
KQ_EV_DISABLE	禁用事件
KQ_EV_ONESHOT	在第一次发生后删除事件
KQ_EV_CLEAR	检索事件后重置状态
KQ_EV_SYSFLAGS	内部事件
KQ_EV_FLAG1	内部事件
KQ_EV_EOF	筛选特定 EOF 条件
KQ_EV_ERROR	请参阅返回值

kevent.**fflags**
筛选特定标志。

KQ_FILTER_READ 和 KQ_FILTER_WRITE 过滤标志：

常数	含义
KQ_NOTE_LOWAT	套接字缓冲区的低水线

KQ_FILTER_VNODE 过滤标志：

常数	含义
KQ_NOTE_DELETE	已调用 *unlink()*
KQ_NOTE_WRITE	发生写入
KQ_NOTE_EXTEND	文件已扩展
KQ_NOTE_ATTRIB	属性已更改
KQ_NOTE_LINK	链接计数已更改
KQ_NOTE_RENAME	文件已重命名
KQ_NOTE_REVOKE	对文件的访问权限已被撤销

KQ_FILTER_PROC filter flags:

常数	含义
KQ_NOTE_EXIT	进程已退出
KQ_NOTE_FORK	该进程调用了 *fork()*
KQ_NOTE_EXEC	进程已执行新进程
KQ_NOTE_PCTRLMASK	内部过滤器标志
KQ_NOTE_PDATAMASK	内部过滤器标志
KQ_NOTE_TRACK	跨 *fork()* 执行进程
KQ_NOTE_CHILD	在 *NOTE_TRACK* 的子进程上返回
KQ_NOTE_TRACKERR	无法附加到子对象

KQ_FILTER_NETDEV 过滤器标志（在 Mac OS X 上不可用）：

常数	含义
KQ_NOTE_LINKUP	链接已建立
KQ_NOTE_LINKDOWN	链接已断开
KQ_NOTE_LINKINV	链接状态无效

kevent.**data**
过滤特定数据。

kevent.**udata**
用户定义的值。

18.4 selectors —高级 I/O 复用库

3.4 新版功能.

源码: Lib/selectors.py

18.4.1 概述

此模块允许高层级且高效率的 I/O 复用，它建立在*select* 模块原型的基础之上。推荐用户改用此模块，除非他们希望对所使用的 OS 层级原型进行精确控制。

它定义了一个*BaseSelector* 抽象基类，以及多个实际的实现 (*KqueueSelector*, *EpollSelector*…)，它们可被用于在多个文件对象上等待 I/O 就绪通知。在下文中，”文件对象”是指任何具有 `fileno()` 方法的对象，或是一个原始文件描述器。参见*file object*。

DefaultSelector 是一个指向当前平台上可用的最高效实现的别名：这应为大多数用户的默认选择。

注解: 受支持的文件对象类型取决于具体平台：在 Windows 上，支持套接字但不支持管道，而在 Unix 上两者均受支持（某些其他类型也可能受支持，例如 fifo 或特殊文件设备等）。

参见:

select 低层级的 I/O 多路复用模块。

18.4.2 类

类的层次结构:

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

下文中，*events* 一个位掩码，指明哪些 I/O 事件要在给定的文件对象上执行等待。它可以是以下模块级常量的组合:

常数	含义
EVENT_READ	可读
EVENT_WRITE	可写

class selectors.**SelectorKey**

SelectorKey 是一个*namedtuple*，用来将文件对象关联到其下层的文件描述器、选定事件掩码和附加数据等。它会被某些*BaseSelector* 方法返回。

fileobj
已注册的文件对象。

fd
下层的文件描述器。

events
必须在此文件对象上被等待的事件。

data
可选的关联到此文件对象的不透明数据：例如，这可被用来存储各个客户端的会话 ID。

class selectors.**BaseSelector**

一个*BaseSelector*，用来在多个文件对象上等待 I/O 事件就绪。它支持文件流注册、注销，以及在这些流上等待 I/O 事件的方法。它是一个抽象基类，因此不能被实例化。请改用*DefaultSelector*，或者*SelectSelector*，*KqueueSelector* 等。如果你想要指明使用某个实现，并且你的平台支持它的话。*BaseSelector* 及其具体实现支持*context manager* 协议。

abstractmethod register(*fileobj*, *events*, *data=None*)
注册一个用于选择的文件对象，在其上监视 I/O 事件。

fileobj 是要监视的文件对象。它可以是整数形式的文件描述符或者具有 `fileno()` 方法的对象。*events* 是要监视的事件的位掩码。*data* 是一个不透明对象。

这将返回一个新的*SelectorKey* 实例，或在出现无效事件掩码或文件描述符时引发*ValueError*，或在文件对象已被注册时引发*KeyError*。

abstractmethod unregister(*fileobj*)
注销对一个文件对象的选择，移除对它的监视。在文件对象被关闭之前应当先将其注销。

fileobj 必须是之前已注册的文件对象。

这将返回已关联的*SelectorKey* 实例，或者如果 *fileobj* 未注册则会引发*KeyError*。It will raise *ValueError* 如果 *fileobj* 无效（例如它没有 `fileno()` 方法或其 `fileno()` 方法返回无效值）。

modify(*fileobj*, *events*, *data=None*)
更改已注册文件对象所监视的事件或所附带的数据。

这等价于 `BaseSelector.unregister(fileobj)()` 加 `BaseSelector.register(fileobj, events, data)()`，区别在于它可以被更高效地实现。

这将返回一个新的*SelectorKey* 实例，或在出现无效事件掩码或文件描述符时引发*ValueError*，或在文件对象未被注册时引发*KeyError*。

abstractmethod select(*timeout=None*)
等待直到有已注册的文件对象就绪，或是超过时限。

如果 `timeout > 0`，这指定以秒数表示的最大等待时间。如果 `timeout <= 0`，调用将不会阻塞，并将报告当前就绪的文件对象。如果 *timeout* 为 `None`，调用将阻塞直到某个被监视的文件对象就绪。

这将返回由 `(key, events)` 元组构成的列表，每项各表示一个就绪的文件对象。

key 是对应于就绪文件对象的*SelectorKey* 实例。*events* 是在此文件对象上等待的事件位掩码。

注解： 如果当前进程收到一个信号，此方法可在任何文件对象就绪之前或超出时限时返回：在此情况下，将返回一个空列表。

在 3.5 版更改: 现在当被某个信号中断时，如果信号处理程序没有引发异常，选择器会用重新计算的超时值进行重试（请查看 PEP 475 其理由），而不是在超时之前返回空的事件列表。

close()
关闭选择器。

必须调用这个方法以确保下层资源会被释放。选择器被关闭后将不可再使用。

get_key(*fileobj*)
返回关联到某个已注册文件对象的键。

此方法将返回关联到文件对象的*SelectorKey* 实例，或在文件对象未注册时引发*KeyError*。

abstractmethod get_map()
返回从文件对象到选择器键的映射。

这将返回一个将已注册文件对象映射到与其相关联的*SelectorKey* 实例的*Mapping* 实例。

class selectors.**DefaultSelector**
默认的选择器类，使用当前平台上可用的最高效实现。这应为大多数用户的默认选择。

class selectors.**SelectSelector**
基于*select.select()* 的选择器。

class selectors.**PollSelector**
基于*select.poll()* 的选择器。

class selectors.**EpollSelector**
基于*select.epoll()* 的选择器。

fileno()
此方法将返回由下层*select.epoll()* 对象所使用的文件描述符。

class selectors.**DevpollSelector**
基于*select.devpoll()* 的选择器。

fileno()
此方法将返回由下层*select.devpoll()* 对象所使用的文件描述符。

3.5 新版功能.

class selectors.**KqueueSelector**
基于*select.kqueue()* 的选择器。

fileno()
此方法将返回由下层*select.kqueue()* 对象所使用的文件描述符。

18.4.3 例子

下面是一个简单的回显服务器实现:

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)
```

(下页继续)

(续上页)

```
def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

18.5 asyncio —Asynchronous I/O, event loop, coroutines and tasks

3.4 新版功能.

Source code: Lib/asyncio/

This module provides infrastructure for writing single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives. Here is a more detailed list of the package contents:

- a pluggable *event loop* with various system-specific implementations;
- *transport* and *protocol* abstractions (similar to those in Twisted);
- concrete support for TCP, UDP, SSL, subprocess pipes, delayed calls, and others (some may be system-dependent);
- a `Future` class that mimics the one in the `concurrent.futures` module, but adapted for use with the event loop;
- coroutines and tasks based on `yield from` (**PEP 380**), to help write concurrent code in a sequential fashion;
- cancellation support for `Future`s and coroutines;
- *synchronization primitives* for use between coroutines in a single thread, mimicking those in the `threading` module;
- an interface for passing work off to a threadpool, for times when you absolutely, positively have to use a library that makes blocking I/O calls.

Asynchronous programming is more complex than classical "sequential" programming: see the *Develop with asyncio* page which lists common traps and explains how to avoid them. *Enable the debug mode* during development to detect common issues.

Table of contents:

18.5.1 Base Event Loop

Source code: Lib/asyncio/events.py

The event loop is the central execution device provided by *asyncio*. It provides multiple facilities, including:

- Registering, executing and cancelling delayed calls (timeouts).
- Creating client and server *transports* for various kinds of communication.
- Launching subprocesses and the associated *transports* for communication with an external program.
- Delegating costly function calls to a pool of threads.

class asyncio.**BaseEventLoop**
This class is an implementation detail. It is a subclass of *AbstractEventLoop* and may be a base class of concrete event loop implementations found in *asyncio*. It should not be used directly; use *AbstractEventLoop* instead. BaseEventLoop should not be subclassed by third-party code; the internal interface is not stable.

class asyncio.**AbstractEventLoop**
Abstract base class of event loops.

This class is *not thread safe*.

Run an event loop

AbstractEventLoop.**run_forever**()
Run until *stop()* is called. If *stop()* is called before *run_forever()* is called, this polls the I/O selector once with a timeout of zero, runs all callbacks scheduled in response to I/O events (and those that were already scheduled), and then exits. If *stop()* is called while *run_forever()* is running, this will run the current batch of callbacks and then exit. Note that callbacks scheduled by callbacks will not run in that case; they will run the next time *run_forever()* is called.

在 3.5.1 版更改.

AbstractEventLoop.**run_until_complete**(*future*)
Run until the *Future* is done.

If the argument is a *coroutine object*, it is wrapped by *ensure_future()*.

Return the Future's result, or raise its exception.

AbstractEventLoop.**is_running**()
Returns running status of event loop.

AbstractEventLoop.**stop**()
Stop running the event loop.

This causes *run_forever()* to exit at the next suitable opportunity (see there for more details).

在 3.5.1 版更改.

AbstractEventLoop.**is_closed**()
Returns True if the event loop was closed.

3.4.2 新版功能.

AbstractEventLoop.**close**()
Close the event loop. The loop must not be running. Pending callbacks will be lost.

This clears the queues and shuts down the executor, but does not wait for the executor to finish.

This is idempotent and irreversible. No other methods should be called after this one.

coroutine AbstractEventLoop.**shutdown_asyncgens**()
Schedule all currently open *asynchronous generator* objects to close with an `aclose()` call. After calling this method, the event loop will issue a warning whenever a new asynchronous generator is iterated. Should be used to finalize all scheduled asynchronous generators reliably. Example:

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

3.6 新版功能.

Calls

Most *asyncio* functions don't accept keywords. If you want to pass keywords to your callback, use *functools.partial()*. For example, `loop.call_soon(functools.partial(print, "Hello", flush=True))` will call `print("Hello", flush=True)`.

注解: *functools.partial()* is better than `lambda` functions, because *asyncio* can inspect *functools.partial()* object to display parameters in debug mode, whereas `lambda` functions have a poor representation.

AbstractEventLoop.**call_soon**(*callback*, **args*)
Arrange for a callback to be called as soon as possible. The callback is called after *call_soon()* returns, when control returns to the event loop.

This operates as a FIFO queue, callbacks are called in the order in which they are registered. Each callback will be called exactly once.

Any positional arguments after the callback will be passed to the callback when it is called.

An instance of *asyncio.Handle* is returned, which can be used to cancel the callback.

Use functools.partial to pass keywords to the callback.

AbstractEventLoop.**call_soon_threadsafe**(*callback*, **args*)
Like *call_soon()*, but thread safe.

参见*concurrency and multithreading* 部分的文档。

Delayed calls

The event loop has its own internal clock for computing timeouts. Which clock is used depends on the (platform-specific) event loop implementation; ideally it is a monotonic clock. This will generally be a different clock than *time.time()*.

注解: Timeouts (relative *delay* or absolute *when*) should not exceed one day.

AbstractEventLoop.**call_later**(*delay*, *callback*, **args*)
Arrange for the *callback* to be called after the given *delay* seconds (either an int or float).

An instance of *asyncio.Handle* is returned, which can be used to cancel the callback.

callback will be called exactly once per call to *call_later()*. If two callbacks are scheduled for exactly the same time, it is undefined which will be called first.

The optional positional *args* will be passed to the callback when it is called. If you want the callback to be called with some named arguments, use a closure or *functools.partial()*.

Use functools.partial to pass keywords to the callback.

AbstractEventLoop.**call_at**(*when*, *callback*, **args*)
Arrange for the *callback* to be called at the given absolute timestamp *when* (an int or float), using the same time reference as *AbstractEventLoop.time()*.

本方法的行为和*call_later()* 方法相同。

An instance of *asyncio.Handle* is returned, which can be used to cancel the callback.

Use functools.partial to pass keywords to the callback.

AbstractEventLoop.**time**()
Return the current time, as a *float* value, according to the event loop's internal clock.

参见:

asyncio.sleep() 函数。

Futures

AbstractEventLoop.**create_future**()
Create an *asyncio.Future* object attached to the loop.

This is a preferred way to create futures in asyncio, as event loop implementations can provide alternative implementations of the Future class (with better performance or instrumentation).

3.5.2 新版功能.

Tasks

AbstractEventLoop.**create_task**(*coro*)
Schedule the execution of a *coroutine object*: wrap it in a future. Return a *Task* object.

第三方的事件循环可以使用它们自己的*Task* 子类来满足互操作性。这种情况下结果类型是一个*Task* 的子类。

This method was added in Python 3.4.2. Use the *async()* function to support also older Python versions.

3.4.2 新版功能.

AbstractEventLoop.**set_task_factory**(*factory*)
Set a task factory that will be used by *AbstractEventLoop.create_task()*.

If *factory* is `None` the default task factory will be set.

If *factory* is a *callable*, it should have a signature matching `(loop, coro)`, where *loop* will be a reference to the active event loop, *coro* will be a coroutine object. The callable must return an *asyncio.Future* compatible object.

3.4.4 新版功能.

AbstractEventLoop.**get_task_factory**()
Return a task factory, or `None` if the default one is in use.

3.4.4 新版功能.

Creating connections

coroutine AbstractEventLoop.**create_connection**(*protocol_factory*, *host=None*, *port=None*, *, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*)

Create a streaming transport connection to a given Internet *host* and *port*: socket family *AF_INET* or *AF_INET6* depending on *host* (or *family* if specified), socket type *SOCK_STREAM*. *protocol_factory* must be a callable returning a *protocol* instance.

This method is a *coroutine* which will try to establish the connection in the background. When successful, the coroutine returns a `(transport, protocol)` pair.

底层操作的大致的执行顺序是这样的：

1. The connection is established, and a *transport* is created to represent it.
2. *protocol_factory* is called without arguments and must return a *protocol* instance.
3. The protocol instance is tied to the transport, and its `connection_made()` method is called.
4. The coroutine returns successfully with the `(transport, protocol)` pair.

被创建的传输对象是一个实现相关的双向流。

注解： *protocol_factory* can be any kind of callable, not necessarily a class. For example, if you want to use a pre-created protocol instance, you can pass `lambda: my_protocol`.

Options that change how the connection is created:

- *ssl*: if given and not false, a SSL/TLS transport is created (by default a plain TCP transport is created). If *ssl* is a *ssl.SSLContext* object, this context is used to create the transport; if *ssl* is *True*, a context with some unspecified default settings is used.

  参见：

  SSL/TLS security considerations

- *server_hostname*, is only for use together with *ssl*, and sets or overrides the hostname that the target server's certificate will be matched against. By default the value of the *host* argument is used. If *host* is empty, there is no default and you must pass a value for *server_hostname*. If *server_hostname* is an empty string, hostname matching is disabled (which is a serious security risk, allowing for man-in-the-middle-attacks).
- *family*, *proto*, *flags* 是可选的地址族、协议和标志，它们会被传递给 getaddrinfo() 来对 *host* 进行解析。如果要指定的话，这些都应该是来自于*socket* 模块的对应常量。
- *sock*，如果指定的话，其应该是一个已经存在，并且已经处于连接状态的*socket.socket* 对象，其会被传输对象使用。如果指定了 *sock* ，那么 *host*, *port*, *family*, *proto*, *flags* 和 *local_addr* 就都不应该被指定了。
- *local_addr*, if given, is a `(local_host, local_port)` tuple used to bind the socket to locally. The *local_host* and *local_port* are looked up using getaddrinfo(), similarly to *host* and *port*.

在 3.5 版更改: On Windows with *ProactorEventLoop*, SSL/TLS is now supported.

参见：

The *open_connection()* function can be used to get a pair of (*StreamReader*, *StreamWriter*) instead of a protocol.

coroutine AbstractEventLoop.**create_datagram_endpoint**(*protocol_factory*, *local_addr=None*, *remote_addr=None*, ***, *family=0*, *proto=0*, *flags=0*, *reuse_address=None*, *reuse_port=None*, *allow_broadcast=None*, *sock=None*)

注解: 形参 *reuse_address* 已不再受支持，因为使用 SO_REUSEADDR 会对 UDP 造成显著的安全问题。显式地传入 reuse_address=True 将会引发异常。

当具有不同 UID 的多个进程将套接字赋给具有 SO_REUSEADDR 的相同 UDP 套接字地址时，传入的数据包可能会在套接字间随机分配。

对于受支持的平台，*reuse_port* 可以被用作类似功能的替代。通过 *reuse_port* 将改用 SO_REUSEPORT，它能够防止具有不同 UID 的进程将套接字赋给相同的套接字地址。

创建一个数据报连接。

Create datagram connection: socket family *AF_INET* or *AF_INET6* depending on *host* (or *family* if specified), socket type *SOCK_DGRAM*. *protocol_factory* must be a callable returning a *protocol* instance.

This method is a *coroutine* which will try to establish the connection in the background. When successful, the coroutine returns a (transport, protocol) pair.

Options changing how the connection is created:

- *local_addr*，如果指定的话，就是一个 (local_host, local_port) 元组，用于在本地绑定套接字。*local_host* 和 *local_port* 是使用*getaddrinfo()* 来查找的。
- *remote_addr*，如果指定的话，就是一个 (remote_host, remote_port) 元组，用于同一个远程地址连接。*remote_host* 和 *remote_port* 是使用*getaddrinfo()* 来查找的。
- *family*, *proto*, *flags* 是可选的地址族，协议和标志，其会被传递给*getaddrinfo()* 来完成 *host* 的解析。如果要指定的话，这些都应该是来自于*socket* 模块的对应常量。
- *reuse_port* tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows and some UNIX's. If the SO_REUSEPORT constant is not defined then this capability is unsupported.
- *allow_broadcast* 告知内核允许此端点向广播地址发送消息。
- *sock* 可选择通过指定此值用于使用一个预先存在的，已经处于连接状态的*socket.socket* 对象，并将其提供给此传输对象使用。如果指定了这个值，*local_addr* 和 *remote_addr* 就应该被忽略 (必须为*None*)。

On Windows with *ProactorEventLoop*, this method is not supported.

参见*UDP echo 客户端协议* 和*UDP echo 服务端协议* 的例子。

在 3.4.4 版更改: 添加了 *family*, *proto*, *flags*, *reuse_address*, *reuse_port*, *allow_broadcast* 和 *sock* 等参数。

在 3.6.10 版更改: The *reuse_address* parameter is no longer supporter due to security concerns

coroutine AbstractEventLoop.**create_unix_connection**(*protocol_factory*, *path*, ***, *ssl=None*, *sock=None*, *server_hostname=None*)

Create UNIX connection: socket family *AF_UNIX*, socket type *SOCK_STREAM*. The *AF_UNIX* socket family is used to communicate between processes on the same machine efficiently.

This method is a *coroutine* which will try to establish the connection in the background. When successful, the coroutine returns a `(transport, protocol)` pair.

path is the name of a UNIX domain socket, and is required unless a *sock* parameter is specified. Abstract UNIX sockets, `str`, and `bytes` paths are supported.

See the `AbstractEventLoop.create_connection()` method for parameters.

Availability: UNIX.

Creating listening connections

coroutine AbstractEventLoop.**create_server**(*protocol_factory*, *host=None*, *port=None*, ***, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*)

Create a TCP server (socket type `SOCK_STREAM`) bound to *host* and *port*.

Return a `Server` object, its `sockets` attribute contains created sockets. Use the `Server.close()` method to stop the server: close listening sockets.

参数：

- The *host* parameter can be a string, in that case the TCP server is bound to *host* and *port*. The *host* parameter can also be a sequence of strings and in that case the TCP server is bound to all hosts of the sequence. If *host* is an empty string or `None`, all interfaces are assumed and a list of multiple sockets will be returned (most likely one for IPv4 and another one for IPv6).
- *family* can be set to either `socket.AF_INET` or `AF_INET6` to force the socket to use IPv4 or IPv6. If not set it will be determined from host (defaults to `socket.AF_UNSPEC`).
- *flags* 是用于`getaddrinfo()` 的位掩码。
- *sock* can optionally be specified in order to use a preexisting socket object. If specified, *host* and *port* should be omitted (must be `None`).
- *backlog* 是传递给`listen()` 的最大排队连接的数量（默认为 100）。
- *ssl* 可被设置为一个`SSLContext` 以在接受的连接上启用 SSL。
- *reuse_address* tells the kernel to reuse a local socket in TIME_WAIT state, without waiting for its natural timeout to expire. If not specified will automatically be set to `True` on UNIX.
- *reuse_port* 告知内核，只要在创建的时候都设置了这个标志，就允许此端点绑定到其它端点列表所绑定的同样的端口上。这个选项在 Windows 上是不支持的。

This method is a *coroutine*.

在 3.5 版更改: On Windows with `ProactorEventLoop`, SSL/TLS is now supported.

参见:

The function `start_server()` creates a (`StreamReader`, `StreamWriter`) pair and calls back a function with this pair.

在 3.5.1 版更改: The *host* parameter can now be a sequence of strings.

coroutine AbstractEventLoop.**create_unix_server**(*protocol_factory*, *path=None*, ***, *sock=None*, *backlog=100*, *ssl=None*)

Similar to `AbstractEventLoop.create_server()`, but specific to the socket family `AF_UNIX`.

This method is a *coroutine*.

Availability: UNIX.

coroutine BaseEventLoop.**connect_accepted_socket**(*protocol_factory*, *sock*, *, *ssl=None*)
Handle an accepted connection.

This is used by servers that accept connections outside of asyncio but that use asyncio to handle them.

参数:

- *sock* is a preexisting socket object returned from an `accept` call.
- *ssl* 可被设置为一个*SSLContext* 以在接受的连接上启用 SSL。

This method is a *coroutine*. When completed, the coroutine returns a `(transport, protocol)` pair.

3.5.3 新版功能.

Watch file descriptors

On Windows with *SelectorEventLoop*, only socket handles are supported (ex: pipe file descriptors are not supported).

On Windows with *ProactorEventLoop*, these methods are not supported.

AbstractEventLoop.**add_reader**(*fd*, *callback*, **args*)
Start watching the file descriptor for read availability and then call the *callback* with specified arguments.

Use functools.partial to pass keywords to the callback.

AbstractEventLoop.**remove_reader**(*fd*)
Stop watching the file descriptor for read availability.

AbstractEventLoop.**add_writer**(*fd*, *callback*, **args*)
Start watching the file descriptor for write availability and then call the *callback* with specified arguments.

Use functools.partial to pass keywords to the callback.

AbstractEventLoop.**remove_writer**(*fd*)
Stop watching the file descriptor for write availability.

The *watch a file descriptor for read events* example uses the low-level *AbstractEventLoop.add_reader()* method to register the file descriptor of a socket.

Low-level socket operations

coroutine AbstractEventLoop.**sock_recv**(*sock*, *nbytes*)
Receive data from the socket. Modeled after blocking *socket.socket.recv()* method.

The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by *nbytes*.

With *SelectorEventLoop* event loop, the socket *sock* must be non-blocking.

This method is a *coroutine*.

coroutine AbstractEventLoop.**sock_sendall**(*sock*, *data*)
Send data to the socket. Modeled after blocking *socket.socket.sendall()* method.

The socket must be connected to a remote socket. This method continues to send data from *data* until either all data has been sent or an error occurs. `None` is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully processed by the receiving end of the connection.

With *SelectorEventLoop* event loop, the socket *sock* must be non-blocking.

This method is a *coroutine*.

coroutine AbstractEventLoop.**sock_connect**(*sock*, *address*)
Connect to a remote socket at *address*. Modeled after blocking *socket.socket.connect()* method.

With *SelectorEventLoop* event loop, the socket *sock* must be non-blocking.

This method is a *coroutine*.

在 3.5.2 版更改: address no longer needs to be resolved. sock_connect will try to check if the *address* is already resolved by calling *socket.inet_pton()*. If not, *AbstractEventLoop.getaddrinfo()* will be used to resolve the *address*.

参见:

AbstractEventLoop.create_connection() and *asyncio.open_connection()*.

coroutine AbstractEventLoop.**sock_accept**(*sock*)
Accept a connection. Modeled after blocking *socket.socket.accept()*.

此 *scoket* 必须绑定到一个地址上并且监听连接。返回值是一个 (conn, address) 对，其中 *conn* 是一个新 * 的套接字对象，用于在此连接上收发数据，*address* 是连接的另一端的套接字所绑定的地址。

The socket *sock* must be non-blocking.

This method is a *coroutine*.

参见:

AbstractEventLoop.create_server() and *start_server()*.

Resolve host name

coroutine AbstractEventLoop.**getaddrinfo**(*host*, *port*, *, *family=0*, *type=0*, *proto=0*, *flags=0*)
This method is a *coroutine*, similar to *socket.getaddrinfo()* function but non-blocking.

coroutine AbstractEventLoop.**getnameinfo**(*sockaddr*, *flags=0*)
This method is a *coroutine*, similar to *socket.getnameinfo()* function but non-blocking.

Connect pipes

On Windows with *SelectorEventLoop*, these methods are not supported. Use *ProactorEventLoop* to support pipes on Windows.

coroutine AbstractEventLoop.**connect_read_pipe**(*protocol_factory*, *pipe*)
Register read pipe in eventloop.

protocol_factory should instantiate object with *Protocol* interface. *pipe* is a *file-like object*. Return pair (transport, protocol), where *transport* supports the *ReadTransport* interface.

使用*SelectorEventLoop* 事件循环，*pipe* 被设置为非阻塞模式。

This method is a *coroutine*.

coroutine AbstractEventLoop.**connect_write_pipe**(*protocol_factory*, *pipe*)
Register write pipe in eventloop.

protocol_factory should instantiate object with BaseProtocol interface. *pipe* is *file-like object*. Return pair (transport, protocol), where *transport* supports *WriteTransport* interface.

使用*SelectorEventLoop* 事件循环，*pipe* 被设置为非阻塞模式。

This method is a *coroutine*.

参见:

The *AbstractEventLoop.subprocess_exec()* and *AbstractEventLoop.subprocess_shell()* methods.

UNIX signals

Availability: UNIX only.

AbstractEventLoop.**add_signal_handler**(*signum*, *callback*, **args*)
Add a handler for a signal.

如果信号数字非法或者不可捕获，就抛出一个*ValueError* 。如果建立处理器的过程中出现问题，会抛出一个*RuntimeError* 。

Use functools.partial to pass keywords to the callback.

AbstractEventLoop.**remove_signal_handler**(*sig*)
Remove a handler for a signal.

Return `True` if a signal handler was removed, `False` if not.

参见:

signal 模块。

Executor

Call a function in an *Executor* (pool of threads or pool of processes). By default, an event loop uses a thread pool executor (*ThreadPoolExecutor*).

coroutine AbstractEventLoop.**run_in_executor**(*executor*, *func*, **args*)
Arrange for a *func* to be called in the specified executor.

The *executor* argument should be an *Executor* instance. The default executor is used if *executor* is `None`.

*Use functools.partial to pass keywords to the *func**.

This method is a *coroutine*.

在 3.5.3 版更改: `BaseEventLoop.run_in_executor()` no longer configures the `max_workers` of the thread pool executor it creates, instead leaving it up to the thread pool executor (*ThreadPoolExecutor*) to set the default.

AbstractEventLoop.**set_default_executor**(*executor*)
Set the default executor used by *run_in_executor()*.

错误处理 API

允许自定义事件循环中如何去处理异常。

AbstractEventLoop.**set_exception_handler**(*handler*)
将 *handler* 设置为新的事件循环异常处理器。

If *handler* is `None`, the default exception handler will be set.

If *handler* is a callable object, it should have a matching signature to `(loop, context)`, where `loop` will be a reference to the active event loop, `context` will be a `dict` object (see *call_exception_handler()* documentation for details about context).

AbstractEventLoop.**get_exception_handler**()
Return the exception handler, or None if the default one is in use.

3.5.2 新版功能.

AbstractEventLoop.**default_exception_handler**(*context*)
默认的异常处理器。

This is called when an exception occurs and no exception handler is set, and can be called by a custom exception handler that wants to defer to the default behavior.

context 参数和*call_exception_handler()* 中的同名参数完全相同。

AbstractEventLoop.**call_exception_handler**(*context*)
调用当前事件循环的异常处理器。

context is a dict object containing the following keys (new keys may be introduced later):

- 'message'：错误消息；
- 'exception'（可选）：异常对象；
- 'future'（可选）：*asyncio.Future* 实例；
- 'handle'（可选）：*asyncio.Handle* 实例；
- 'protocol'（可选）：*Protocol* 实例；
- 'transport'（可选）：*Transport* 实例；
- 'socket'（可选）：*socket.socket* 实例。

注解： Note: this method should not be overloaded in subclassed event loops. For any custom exception handling, use *set_exception_handler()* method.

Debug mode

AbstractEventLoop.**get_debug**()
获取事件循环调试模式设置（*bool*）。

如果环境变量 PYTHONASYNCIODEBUG 是一个非空字符串，就返回 True ，否则就返回 False 。

3.4.2 新版功能.

AbstractEventLoop.**set_debug**(*enabled: bool*)
设置事件循环的调试模式。

3.4.2 新版功能.

参见:

debug mode of asyncio.

Server

class asyncio.**Server**
Server listening on sockets.

Object created by the *AbstractEventLoop.create_server()* method and the *start_server()* function. Don't instantiate the class directly.

close()
停止服务：关闭监听的套接字并且设置*sockets* 属性为 None 。

用于表示已经连进来的客户端连接会保持打开的状态。

服务器是被异步关闭的，使用*wait_closed()* 协程来等待服务器关闭。

coroutine wait_closed()
等待*close()* 方法执行完毕。

This method is a *coroutine*.

sockets
List of *socket.socket* objects the server is listening to, or None if the server is closed.

Handle

class asyncio.**Handle**
A callback wrapper object returned by *AbstractEventLoop.call_soon()*, *AbstractEventLoop.call_soon_threadsafe()*, *AbstractEventLoop.call_later()*, and *AbstractEventLoop.call_at()*.

cancel()
Cancel the call. If the callback is already canceled or executed, this method has no effect.

Event loop examples

call_soon() 的 Hello World 示例。

Example using the *AbstractEventLoop.call_soon()* method to schedule a callback. The callback displays "Hello World" and then stops the event loop:

```
import asyncio

def hello_world(loop):
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
loop.run_forever()
loop.close()
```

参见:

The *Hello World coroutine* example uses a *coroutine*.

使用 call_later() 来展示当前的日期

Example of callback displaying the current date every second. The callback uses the *AbstractEventLoop.call_later()* method to reschedule itself during 5 seconds, and then stops the event loop:

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
loop.run_forever()
loop.close()
```

参见:

The *coroutine displaying the current date* example uses a *coroutine*.

监控一个文件描述符的读事件

Wait until a file descriptor received some data using the *AbstractEventLoop.add_reader()* method and then close the event loop:

```
import asyncio
try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()
loop = asyncio.get_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())
    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)
    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
```

(下页继续)

(续上页)

```
loop.call_soon(wsock.send, 'abc'.encode())

# Run the event loop
loop.run_forever()

# We are done, close sockets and the event loop
rsock.close()
wsock.close()
loop.close()
```

参见:

The *register an open socket to wait for data using a protocol* example uses a low-level protocol created by the *AbstractEventLoop.create_connection()* method.

The *register an open socket to wait for data using streams* example uses high-level streams created by the *open_connection()* function in a coroutine.

为 SIGINT 和 SIGTERM 设置信号处理器

Register handlers for signals `SIGINT` and `SIGTERM` using the *AbstractEventLoop.add_signal_handler()* method:

```
import asyncio
import functools
import os
import signal

def ask_exit(signame):
    print("got signal %s: exit" % signame)
    loop.stop()

loop = asyncio.get_event_loop()
for signame in ('SIGINT', 'SIGTERM'):
    loop.add_signal_handler(getattr(signal, signame),
                            functools.partial(ask_exit, signame))

print("Event loop running forever, press Ctrl+C to interrupt.")
print("pid %s: send SIGINT or SIGTERM to exit." % os.getpid())
try:
    loop.run_forever()
finally:
    loop.close()
```

This example only works on UNIX.

18.5.2 事件循环

源码: Lib/asyncio/events.py

事件循环函数

下面的函数是访问全局策略方法的便利快捷方式。注意这只是提供了对默认策略的访问，除非要在进程开始之前设置替代的策略。

asyncio.**get_event_loop**()
与调用 get_event_loop_policy().get_event_loop() 等价。

asyncio.**set_event_loop**(*loop*)
等价于调用 get_event_loop_policy().set_event_loop(loop)。

asyncio.**new_event_loop**()
等价于调用 get_event_loop_policy().new_event_loop()。

可用的事件循环

asyncio 当前提供两种事件循环实现：*SelectorEventLoop* 和*ProactorEventLoop* 。

class asyncio.**SelectorEventLoop**
基于*selectors* 模块的事件循环。*AbstractEventLoop* 的子类。

使用对应平台上最高效的选择器。

在 Windows 上，只支持套接字（举例：不支持管线）：参见 *MSDN 关于 select 的文档 <https://msdn.microsoft.com/en-us/library/windows/desktop/ms740141%28v=vs.85%29.aspx>* _ 。

class asyncio.**ProactorEventLoop**
Windows 上的 Proactor 事件循环，使用的是“I/O 完成端口”，也叫做 IOCP。为*AbstractEventLoop* 的子类。

可用性：Windows。

参见:

MSDN 上关于 I/O 完成端口的文档 。

在 Windows 上使用*ProactorEventLoop* 的例子:

```
import asyncio, sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

平台支持

asyncio 模块被设计为可移植的，但是每个平台仍旧有细微的不同，而且不一定支持*asyncio* 的所有特性。

Windows

Windows 事件循环的一般限制：

- 不支持*create_unix_connection()* 和*create_unix_server()*： *socket.AF_UNIX* 套接字族只在 UNIX 上可用
- 不支持*add_signal_handler()* 和*remove_signal_handler()*
- 不支持 `EventLoopPolicy.set_child_watcher()` 。*ProactorEventLoop* 支持子进程。监控子进程的实现只有一种，不需要对其进行设置。

特定于*SelectorEventLoop* 的限制：

- *SelectSelector* 只被用于支持套接字，并且最多支持 512 个套接字。
- *add_reader()* 和*add_writer()* 只接受套接字的文件描述符
- 不支持管线（示例： *connect_read_pipe()*, *connect_write_pipe()*）
- 不支持*Subprocesses* （示例: *subprocess_exec()*, *subprocess_shell()*）

特定于*ProactorEventLoop* 的限制：

- 不支持*create_datagram_endpoint()* (UDP)
- 不支持*add_reader()* 和*add_writer()*

Windows 上单调时钟的分辨率大约为 15.6 毫秒。最佳的分辨率是 0.5 毫秒。分辨率依赖于具体的硬件（HPET 的可用性）和 Windows 的设置。参见:ref:*asyncio delayed calls <asyncio-delayed-calls>* 。

在 3.5 版更改: *ProactorEventLoop* 现在支持 SSL 了。

Mac OS X

类似于 PTY 的字符设备在 Mavericks （Mac OS 10.9）之前并未有很好的支持。在 Mac OS 10.5 以及更老的版本上则根本不支持。

在 Mac OS 10.6、10.7 和 10.8 上，默认的事件循环是*SelectorEventLoop* ，其使用了*selectors.KqueueSelector* 。*selectors.KqueueSelector* 在这些版本上并不支持字符设备。*SelectorEventLoop* 可以同*SelectSelector* 或者*PollSelector* 一同使用来在这些版本的 Mac OS X 上支持块设备。例如:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

事件循环策略和默认策略

使用一个 策略模式对事件循环进行管理，用以为特定平台和框架提供最大限度的灵活性。在一个进程的执行中，唯一的全局策略对象，基于调用的上下文，管理着此进程所有可用的事件循环。一个策略就是一个实现了*AbstractEventLoopPolicy* 的对象。

对于*asyncio* 的多数使用者来说，从来都不需要显式地处理策略，因为默认的全局策略就已经足够了（看下面）。

模块级别的函数*get_event_loop()* 和*set_event_loop()* 对于默认策略管理的事件循环的便利访问方式。

事件循环策略接口

一个事件循环必须实现如下的接口：

class asyncio.**AbstractEventLoopPolicy**
事件循环策略。

get_event_loop()
为当前上下文获取事件循环。

返回一个实现了*AbstractEventLoop* 接口的事件循环对象。如果是从一个协程中调用，就会返回当前正在运行的事件循环。

如果没有为当前上下文设置一个事件循环，并且当前策略不能创建一个事件循环，就会抛出一个错误。其务必不能返回 None 。

在 3.6 版更改.

set_event_loop(*loop*)
将当前上下文的事件循环设置为 *loop* 。

new_event_loop()
根据策略的规则，创建并返回一个新的事件循环对象。

如果有需要设置这个循环作为当前上下文的事件循环，就必须显式调用*set_event_loop()* 。

The default policy defines context as the current thread, and manages an event loop per thread that interacts with *asyncio*. If the current thread doesn't already have an event loop associated with it, the default policy's *get_event_loop()* method creates one when called from the main thread, but raises *RuntimeError* otherwise.

访问全局循环策略

asyncio.**get_event_loop_policy**()
获取当前事件循环策略。

asyncio.**set_event_loop_policy**(*policy*)
设置当前事件循环策略。如果 *policy* 为 None ，就会恢复使用默认的策略。

自定义事件循环策略

要实现一个新的事件循环策略，推荐你继承具体的默认事件循环策略 `DefaultEventLoopPolicy` 并且重写你想要改变行为的方法，例如:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

18.5.3 Tasks and coroutines

Source code: Lib/asyncio/tasks.py

Source code: Lib/asyncio/coroutines.py

协程

Coroutines used with *asyncio* may be implemented using the `async def` statement, or by using *generators*. The `async def` type of coroutine was added in Python 3.5, and is recommended if there is no need to support older Python versions.

Generator-based coroutines should be decorated with *@asyncio.coroutine*, although this is not strictly enforced. The decorator enables compatibility with `async def` coroutines, and also serves as documentation. Generator-based coroutines use the `yield from` syntax introduced in **PEP 380**, instead of the original `yield` syntax.

The word "coroutine", like the word "generator", is used for two different (though related) concepts:

- The function that defines a coroutine (a function definition using `async def` or decorated with `@asyncio.coroutine`). If disambiguation is needed we will call this a *coroutine function* (*iscoroutinefunction()* returns `True`).
- The object obtained by calling a coroutine function. This object represents a computation or an I/O operation (usually a combination) that will complete eventually. If disambiguation is needed we will call it a *coroutine object* (*iscoroutine()* returns `True`).

Things a coroutine can do:

- `result = await future` or `result = yield from future` – suspends the coroutine until the future is done, then returns the future's result, or raises an exception, which will be propagated. (If the future is cancelled, it will raise a `CancelledError` exception.) Note that tasks are futures, and everything said about futures also applies to tasks.
- `result = await coroutine` or `result = yield from coroutine` – wait for another coroutine to produce a result (or raise an exception, which will be propagated). The `coroutine` expression must be a *call* to another coroutine.
- `return expression` – produce a result to the coroutine that is waiting for this one using `await` or `yield from`.

- `raise exception` –raise an exception in the coroutine that is waiting for this one using `await` or `yield from`.

Calling a coroutine does not start its code running –the coroutine object returned by the call doesn't do anything until you schedule its execution. There are two basic ways to start it running: call `await coroutine` or `yield from coroutine` from another coroutine (assuming the other coroutine is already running!), or schedule its execution using the *`ensure_future()`* function or the *`AbstractEventLoop.create_task()`* method.

Coroutines (and tasks) can only run when the event loop is running.

`@asyncio.`**`coroutine`**

Decorator to mark generator-based coroutines. This enables the generator use `yield from` to call `async def` coroutines, and also enables the generator to be called by `async def` coroutines, for instance using an `await` expression.

There is no need to decorate `async def` coroutines themselves.

If the generator is not yielded from before it is destroyed, an error message is logged. See *Detect coroutines never scheduled*.

注解: In this documentation, some methods are documented as coroutines, even if they are plain Python functions returning a *`Future`*. This is intentional to have a freedom of tweaking the implementation of these functions in the future. If such a function is needed to be used in a callback-style code, wrap its result with *`ensure_future()`*.

Example: Hello World coroutine

Example of coroutine displaying `"Hello World"`:

```
import asyncio

async def hello_world():
    print("Hello World!")

loop = asyncio.get_event_loop()
# Blocking call which returns when the hello_world() coroutine is done
loop.run_until_complete(hello_world())
loop.close()
```

参见:

The *Hello World with call_soon()* example uses the *`AbstractEventLoop.call_soon()`* method to schedule a callback.

Example: Coroutine displaying the current date

Example of coroutine displaying the current date every second during 5 seconds using the *`sleep()`* function:

```
import asyncio
import datetime

async def display_date(loop):
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
```

(下页继续)

(续上页)

```
        if (loop.time() + 1.0) >= end_time:
            break
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()
# Blocking call which returns when the display_date() coroutine is done
loop.run_until_complete(display_date(loop))
loop.close()
```

参见:

The *display the current date with call_later()* example uses a callback with the *AbstractEventLoop.call_later()* method.

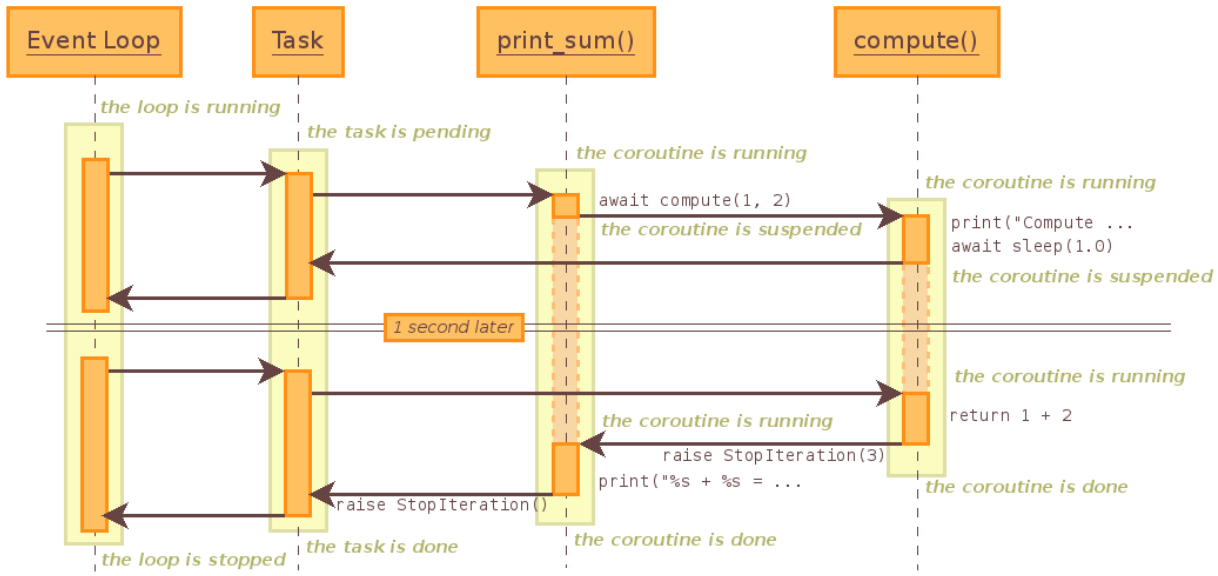
Example: Chain coroutines

Example chaining coroutines:

```
import asyncio

async def compute(x, y):
    print("Compute %s + %s ..." % (x, y))
    await asyncio.sleep(1.0)
    return x + y

async def print_sum(x, y):
    result = await compute(x, y)
    print("%s + %s = %s" % (x, y, result))

loop = asyncio.get_event_loop()
loop.run_until_complete(print_sum(1, 2))
loop.close()
```

`compute()` is chained to `print_sum()`: `print_sum()` coroutine waits until `compute()` is completed before returning its result.

Sequence diagram of the example:



The "Task" is created by the *AbstractEventLoop.run_until_complete()* method when it gets a coroutine object instead of a task.

The diagram shows the control flow, it does not describe exactly how things work internally. For example, the sleep coroutine creates an internal future which uses *AbstractEventLoop.call_later()* to wake up the task in 1 second.

InvalidStateError

exception asyncio.**InvalidStateError**
The operation is not allowed in this state.

TimeoutError

exception asyncio.**TimeoutError**
The operation exceeded the given deadline.

注解: This exception is different from the builtin *TimeoutError* exception!

Future

class asyncio.**Future**(*, *loop=None*)
This class is *almost* compatible with *concurrent.futures.Future*.

Differences:

- *result()* and *exception()* do not take a timeout argument and raise an exception when the future isn't done yet.
- Callbacks registered with *add_done_callback()* are always called via the event loop's *call_soon()*.

- This class is not compatible with the *wait()* and *as_completed()* functions in the *concurrent.futures* package.

This class is *not thread safe*.

cancel()
Cancel the future and schedule callbacks.

If the future is already done or cancelled, return `False`. Otherwise, change the future's state to cancelled, schedule the callbacks and return `True`.

cancelled()
Return `True` if the future was cancelled.

done()
Return `True` if the future is done.

Done means either that a result / exception are available, or that the future was cancelled.

result()
Return the result this future represents.

If the future has been cancelled, raises `CancelledError`. If the future's result isn't yet available, raises *InvalidStateError*. If the future is done and has an exception set, this exception is raised.

exception()
Return the exception that was set on this future.

The exception (or `None` if no exception was set) is returned only if the future is done. If the future has been cancelled, raises `CancelledError`. If the future isn't done yet, raises *InvalidStateError*.

add_done_callback(*fn*)
Add a callback to be run when the future becomes done.

The callback is called with a single argument - the future object. If the future is already done when this is called, the callback is scheduled with *call_soon()*.

Use functools.partial to pass parameters to the callback. For example, `fut.add_done_callback(functools.partial(print, "Future:", flush=True))` will call `print("Future:", fut, flush=True)`.

remove_done_callback(*fn*)
Remove all instances of a callback from the "call when done" list.

Returns the number of callbacks removed.

set_result(*result*)
Mark the future done and set its result.

If the future is already done when this method is called, raises *InvalidStateError*.

set_exception(*exception*)
Mark the future done and set an exception.

If the future is already done when this method is called, raises *InvalidStateError*.

Example: Future with run_until_complete()

Example combining a *Future* and a *coroutine function*:

```
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
loop.run_until_complete(future)
print(future.result())
loop.close()
```

The coroutine function is responsible for the computation (which takes 1 second) and it stores the result into the future. The *run_until_complete()* method waits for the completion of the future.

注解： The *run_until_complete()* method uses internally the *add_done_callback()* method to be notified when the future is done.

Example: Future with run_forever()

The previous example can be written differently using the *Future.add_done_callback()* method to describe explicitly the control flow:

```
import asyncio

async def slow_operation(future):
    await asyncio.sleep(1)
    future.set_result('Future is done!')

def got_result(future):
    print(future.result())
    loop.stop()

loop = asyncio.get_event_loop()
future = asyncio.Future()
asyncio.ensure_future(slow_operation(future))
future.add_done_callback(got_result)
try:
    loop.run_forever()
finally:
    loop.close()
```

In this example, the future is used to link `slow_operation()` to `got_result()`: when `slow_operation()` is done, `got_result()` is called with the result.

Task

class asyncio.**Task**(*coro*, *, *loop=None*)

Schedule the execution of a *coroutine*: wrap it in a future. A task is a subclass of *Future*.

A task is responsible for executing a coroutine object in an event loop. If the wrapped coroutine yields from a future, the task suspends the execution of the wrapped coroutine and waits for the completion of the future. When the future is done, the execution of the wrapped coroutine restarts with the result or the exception of the future.

Event loops use cooperative scheduling: an event loop only runs one task at a time. Other tasks may run in parallel if other event loops are running in different threads. While a task waits for the completion of a future, the event loop executes a new task.

The cancellation of a task is different from the cancelation of a future. Calling *cancel()* will throw a *CancelledError* to the wrapped coroutine. *cancelled()* only returns `True` if the wrapped coroutine did not catch the *CancelledError* exception, or raised a *CancelledError* exception.

If a pending task is destroyed, the execution of its wrapped *coroutine* did not complete. It is probably a bug and a warning is logged: see *Pending task destroyed*.

Don't directly create *Task* instances: use the *ensure_future()* function or the *AbstractEventLoop.create_task()* method.

This class is *not thread safe*.

classmethod all_tasks(*loop=None*)

返回一个事件循环中所有任务的集合。

By default all tasks for the current event loop are returned.

classmethod current_task(*loop=None*)

Return the currently running task in an event loop or `None`.

By default the current task for the current event loop is returned.

`None` is returned when called not in the context of a *Task*.

cancel()

Request that this task cancel itself.

This arranges for a *CancelledError* to be thrown into the wrapped coroutine on the next cycle through the event loop. The coroutine then has a chance to clean up or even deny the request using try/except/finally.

Unlike *Future.cancel()*, this does not guarantee that the task will be cancelled: the exception might be caught and acted upon, delaying cancellation of the task or preventing cancellation completely. The task may also return a value or raise a different exception.

Immediately after this method is called, *cancelled()* will not return `True` (unless the task was already cancelled). A task will be marked as cancelled when the wrapped coroutine terminates with a *CancelledError* exception (even if *cancel()* was not called).

get_stack(*, *limit=None*)

Return the list of stack frames for this task's coroutine.

If the coroutine is not done, this returns the stack where it is suspended. If the coroutine has completed successfully or was cancelled, this returns an empty list. If the coroutine was terminated by an exception, this returns the list of traceback frames.

框架总是从按从旧到新排序。

The optional limit gives the maximum number of frames to return; by default all available frames are returned. Its meaning differs depending on whether a stack or a traceback is returned: the newest frames of a stack are

returned, but the oldest frames of a traceback are returned. (This matches the behavior of the traceback module.)

For reasons beyond our control, only one stack frame is returned for a suspended coroutine.

print_stack(*, *limit=None*, *file=None*)
Print the stack or traceback for this task's coroutine.

This produces output similar to that of the traceback module, for the frames retrieved by get_stack(). The limit argument is passed to get_stack(). The file argument is an I/O stream to which the output is written; by default output is written to sys.stderr.

Example: Parallel execution of tasks

Example executing 3 tasks (A, B, C) in parallel:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number+1):
        print("Task %s: Compute factorial(%s)..." % (name, i))
        await asyncio.sleep(1)
        f *= i
    print("Task %s: factorial(%s) = %s" % (name, number, f))

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.gather(
    factorial("A", 2),
    factorial("B", 3),
    factorial("C", 4),
))
loop.close()
```

Output:

```
Task A: Compute factorial(2)...
Task B: Compute factorial(2)...
Task C: Compute factorial(2)...
Task A: factorial(2) = 2
Task B: Compute factorial(3)...
Task C: Compute factorial(3)...
Task B: factorial(3) = 6
Task C: Compute factorial(4)...
Task C: factorial(4) = 24
```

A task is automatically scheduled for execution when it is created. The event loop stops when all tasks are done.

Task functions

注解: In the functions below, the optional *loop* argument allows explicitly setting the event loop object used by the underlying task or coroutine. If it's not provided, the default event loop is used.

asyncio.**as_completed**(*fs*, *, *loop=None*, *timeout=None*)
Return an iterator whose values, when waited for, are *Future* instances.

如果在所有 Future 对象完成前发生超时则将引发*asyncio.TimeoutError*。

示例:

```
for f in as_completed(fs):
    result = yield from f  # The 'yield from' may raise
    # Use result
```

注解: The futures `f` are not necessarily members of fs.

asyncio.**ensure_future**(*coro_or_future*, *, *loop=None*)
Schedule the execution of a *coroutine object*: wrap it in a future. Return a *Task* object.

If the argument is a *Future*, it is returned directly.

3.4.4 新版功能.

在 3.5.1 版更改: The function accepts any *awaitable* object.

参见:

The *AbstractEventLoop.create_task()* method.

asyncio.**async**(*coro_or_future*, *, *loop=None*)
A deprecated alias to *ensure_future()*.

3.4.4 版后已移除.

asyncio.**wrap_future**(*future*, *, *loop=None*)
Wrap a *concurrent.futures.Future* object in a *Future* object.

asyncio.**gather**(**coros_or_futures*, *loop=None*, *return_exceptions=False*)
Return a future aggregating results from the given coroutine objects or futures.

All futures must share the same event loop. If all the tasks are done successfully, the returned future's result is the list of results (in the order of the original sequence, not necessarily the order of results arrival). If *return_exceptions* is true, exceptions in the tasks are treated the same as successful results, and gathered in the result list; otherwise, the first raised exception will be immediately propagated to the returned future.

Cancellation: if the outer Future is cancelled, all children (that have not completed yet) are also cancelled. If any child is cancelled, this is treated as if it raised *CancelledError* –the outer Future is *not* cancelled in this case. (This is to prevent the cancellation of one child to cause other children to be cancelled.)

在 3.6.6 版更改: 如果 *gather* 本身被取消，则无论 *return_exceptions* 取值为何，消息都会被传播。

asyncio.**iscoroutine**(*obj*)
Return `True` if *obj* is a *coroutine object*, which may be based on a generator or an `async def` coroutine.

asyncio.**iscoroutinefunction**(*func*)
Return `True` if *func* is determined to be a *coroutine function*, which may be a decorated generator function or an `async def` function.

asyncio.**run_coroutine_threadsafe**(*coro*, *loop*)
Submit a *coroutine object* to a given event loop.

Return a *concurrent.futures.Future* to access the result.

This function is meant to be called from a different thread than the one where the event loop is running. Usage:

```
# Create a coroutine
coro = asyncio.sleep(1, result=3)
# Submit the coroutine to a given loop
future = asyncio.run_coroutine_threadsafe(coro, loop)
# Wait for the result with an optional timeout argument
assert future.result(timeout) == 3
```

If an exception is raised in the coroutine, the returned future will be notified. It can also be used to cancel the task in the event loop:

```
try:
    result = future.result(timeout)
except asyncio.TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print('The coroutine raised an exception: {!r}'.format(exc))
else:
    print('The coroutine returned: {!r}'.format(result))
```

参见*concurrency and multithreading* 部分的文档。

注解： Unlike other functions from the module, *run_coroutine_threadsafe()* requires the *loop* argument to be passed explicitly.

3.5.1 新版功能.

coroutine asyncio.**sleep**(*delay*, *result=None*, *, *loop=None*)
Create a *coroutine* that completes after a given time (in seconds). If *result* is provided, it is produced to the caller when the coroutine completes.

The resolution of the sleep depends on the *granularity of the event loop*.

This function is a *coroutine*.

coroutine asyncio.**shield**(*arg*, *, *loop=None*)
Wait for a future, shielding it from cancellation.

以下语句:

```
res = yield from shield(something())
```

is exactly equivalent to the statement:

```
res = yield from something()
```

except that if the coroutine containing it is cancelled, the task running in `something()` is not cancelled. From the point of view of `something()`, the cancellation did not happen. But its caller is still cancelled, so the yield-from expression still raises *CancelledError*. Note: If `something()` is cancelled by other means this will still cancel `shield()`.

If you want to completely ignore cancellation (not recommended) you can combine `shield()` with a try/except clause, as follows:

```
try:
    res = yield from shield(something())
except CancelledError:
    res = None
```

coroutine asyncio.**wait**(*futures*, *, *loop=None*, *timeout=None*, *return_when=ALL_COMPLETED*)

Wait for the Futures and coroutine objects given by the sequence *futures* to complete. Coroutines will be wrapped in Tasks. Returns two sets of *Future*: (done, pending).

The sequence *futures* must not be empty.

timeout can be used to control the maximum number of seconds to wait before returning. *timeout* can be an int or float. If *timeout* is not specified or `None`, there is no limit to the wait time.

return_when indicates when this function should return. It must be one of the following constants of the *concurrent.futures* module:

常数	描述
FIRST_COMPLETED	函数将在任意可等待对象结束或取消时返回。
FIRST_EXCEPTION	函数将在任意可等待对象因引发异常而结束时返回。当没有引发任何异常时它就相当于 ALL_COMPLETED。
ALL_COMPLETED	函数将在所有可等待对象结束或取消时返回。

This function is a *coroutine*.

用法:

```
done, pending = yield from asyncio.wait(fs)
```

注解: This does not raise *asyncio.TimeoutError*! Futures that aren't done when the timeout occurs are returned in the second set.

coroutine asyncio.**wait_for**(*fut*, *timeout*, *, *loop=None*)

Wait for the single *Future* or *coroutine object* to complete with timeout. If *timeout* is `None`, block until the future completes.

Coroutine will be wrapped in *Task*.

Returns result of the Future or coroutine. When a timeout occurs, it cancels the task and raises *asyncio.TimeoutError*. To avoid the task cancellation, wrap it in *shield()*.

If the wait is cancelled, the future *fut* is also cancelled.

This function is a *coroutine*, usage:

```
result = yield from asyncio.wait_for(fut, 60.0)
```

在 3.4.3 版更改: If the wait is cancelled, the future *fut* is now also cancelled.

18.5.4 Transports and protocols (callback based API)

Source code: Lib/asyncio/transports.py

Source code: Lib/asyncio/protocols.py

传输

Transports are classes provided by *asyncio* in order to abstract various kinds of communication channels. You generally won't instantiate a transport yourself; instead, you will call an *AbstractEventLoop* method which will create the transport and try to initiate the underlying communication channel, calling you back when it succeeds.

Once the communication channel is established, a transport is always paired with a *protocol* instance. The protocol can then call the transport's methods for various purposes.

asyncio currently implements transports for TCP, UDP, SSL, and subprocess pipes. The methods available on a transport depend on the transport's kind.

传输类属于线程不安全 。

在 3.6 版更改: The socket option `TCP_NODELAY` is now set by default.

BaseTransport

class asyncio.**BaseTransport**

Base class for transports.

close()

Close the transport. If the transport has a buffer for outgoing data, buffered data will be flushed asynchronously. No more data will be received. After all buffered data is flushed, the protocol's `connection_lost()` method will be called with *None* as its argument.

is_closing()

返回 `True` ，如果传输正在关闭或已经关闭。

3.5.1 新版功能.

get_extra_info(*name*, *default=None*)

Return optional transport information. *name* is a string representing the piece of transport-specific information to get, *default* is the value to return if the information doesn't exist.

This method allows transport implementations to easily expose channel-specific information.

- 套接字:
  - `'peername'`: 套接字链接时的远端地址，*socket.socket.getpeername()* 方法的结果 (出错时为 `None`)
  - `'socket'`: *socket.socket* 实例
  - `'sockname'`: 套接字本地地址，*socket.socket.getsockname()* 方法的结果
- SSL 套接字
  - `'compression'`: 用字符串指定压缩算法，或者链接没有压缩时为 `None` ；*ssl.SSLSocket.compression()* 的结果。
  - `'cipher'`: 一个三值的元组，包含使用密码的名称，定义使用的 SSL 协议的版本，使用的加密位数。*ssl.SSLSocket.cipher()* 的结果。
  - `'peercert'`: 远端凭证；*ssl.SSLSocket.getpeercert()* 结果。

- 'sslcontext': *ssl.SSLContext* 实例
- 'ssl_object': *ssl.SSLObject* 或*ssl.SSLSocket* 实例

- 管道:
  - 'pipe': 管道对象
- 子进程:
  - 'subprocess': *subprocess.Popen* 实例

set_protocol(*protocol*)
Set a new protocol. Switching protocol should only be done when both protocols are documented to support the switch.

3.5.3 新版功能.

get_protocol()
返回当前协议。

3.5.3 新版功能.

在 3.5.1 版更改: 'ssl_object' info was added to SSL sockets.

ReadTransport

class asyncio.**ReadTransport**
Interface for read-only transports.

pause_reading()
Pause the receiving end of the transport. No data will be passed to the protocol's data_received() method until *resume_reading()* is called.

在 3.6.7 版更改: 这个方法幂等的，它可以在传输已经暂停或关闭时调用。

resume_reading()
Resume the receiving end. The protocol's data_received() method will be called once again if some data is available for reading.

在 3.6.7 版更改: 这个方法幂等的，它可以在传输已经准备好读取数据时调用。

WriteTransport

class asyncio.**WriteTransport**
Interface for write-only transports.

abort()
Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's connection_lost() method will eventually be called with *None* as its argument.

can_write_eof()
Return *True* if the transport supports *write_eof()*, *False* if not.

get_write_buffer_size()
返回传输使用输出缓冲区的当前大小。

get_write_buffer_limits()
Get the *high*- and *low*-water limits for write flow control. Return a tuple (low, high) where *low* and *high* are positive number of bytes.

使用*set_write_buffer_limits()* 设置限制。

3.4.2 新版功能.

set_write_buffer_limits(*high=None*, *low=None*)
Set the *high*- and *low*-water limits for write flow control.

These two values (measured in number of bytes) control when the protocol's `pause_writing()` and `resume_writing()` methods are called. If specified, the low-water limit must be less than or equal to the high-water limit. Neither *high* nor *low* can be negative.

`pause_writing()` is called when the buffer size becomes greater than or equal to the *high* value. If writing has been paused, `resume_writing()` is called when the buffer size becomes less than or equal to the *low* value.

The defaults are implementation-specific. If only the high-water limit is given, the low-water limit defaults to an implementation-specific value less than or equal to the high-water limit. Setting *high* to zero forces *low* to zero as well, and causes `pause_writing()` to be called whenever the buffer becomes non-empty. Setting *low* to zero causes `resume_writing()` to be called only once the buffer is empty. Use of zero for either limit is generally sub-optimal as it reduces opportunities for doing I/O and computation concurrently.

Use *get_write_buffer_limits()* to get the limits.

write(*data*)
将一些 *data* 字节串写入传输。

此方法不会阻塞；它会缓冲数据并安排其被异步地发出。

writelines(*list_of_data*)
将数据字节串的列表（或任意可迭代对象）写入传输。这在功能上等价于在可迭代对象产生的每个元素上调用*write()*，但其实现可能更为高效。

write_eof()
Close the write end of the transport after flushing buffered data. Data may still be received.

This method can raise *NotImplementedError* if the transport (e.g. SSL) doesn't support half-closes.

DatagramTransport

DatagramTransport.**sendto**(*data*, *addr=None*)
将 *data* 字节串发送到 *addr* (基于传输的目标地址) 所给定的远端对等方。如果 *addr* 为*None*，则将数据发送到传输创建时给定的目标地址。

此方法不会阻塞；它会缓冲数据并安排其被异步地发出。

DatagramTransport.**abort**()
Close the transport immediately, without waiting for pending operations to complete. Buffered data will be lost. No more data will be received. The protocol's `connection_lost()` method will eventually be called with *None* as its argument.

BaseSubprocessTransport

class asyncio.**BaseSubprocessTransport**

get_pid()
将子进程的进程 ID 以整数形式返回。

get_pipe_transport(*fd*)
返回对应于整数文件描述符 *fd* 的通信管道的传输:

- 0: 标准输入 (*stdin*) 的可读流式传输，如果子进程创建时未设置 stdin=PIPE 则为*None*。
- 1: 标准输出 writable streaming transport of the standard output (*stdout*) 的可写流式传输，如果子进程创建时未设置 stdout=PIPE 则为*None*。
- 2: 标准错误 (*stderr*) 的可写流式传输，如果子进程创建时未设置 stderr=PIPE 则为*None*
- 其他 *fd*: *None*

get_returncode()
Return the subprocess returncode as an integer or *None* if it hasn't returned, similarly to the *subprocess.Popen.returncode* attribute.

kill()
Kill the subprocess, as in *subprocess.Popen.kill()*.

在 POSIX 系统中，函数会发送 SIGKILL 到子进程。在 Windows 中，此方法是*terminate()* 的别名。

send_signal(*signal*)
发送 *signal* 编号到子进程，与*subprocess.Popen.send_signal()* 一样。

terminate()
Ask the subprocess to stop, as in *subprocess.Popen.terminate()*. This method is an alias for the *close()* method.

在 POSIX 系统中，此方法会发送 SIGTERM 到子进程。在 Windows 中，则会调用 Windows API 函数 TerminateProcess() 来停止子进程。

close()
Ask the subprocess to stop by calling the *terminate()* method if the subprocess hasn't returned yet, and close transports of all pipes (*stdin*, *stdout* and *stderr*).

协议

asyncio provides base classes that you can subclass to implement your network protocols. Those classes are used in conjunction with *transports* (see below): the protocol parses incoming data and asks for the writing of outgoing data, while the transport is responsible for the actual I/O and buffering.

When subclassing a protocol class, it is recommended you override certain methods. Those methods are callbacks: they will be called by the transport on certain events (for example when some data is received); you shouldn't call them yourself, unless you are implementing a transport.

注解: All callbacks have default implementations, which are empty. Therefore, you only need to implement the callbacks for the events in which you are interested.

Protocol classes

class asyncio.**Protocol**
The base class for implementing streaming protocols (for use with e.g. TCP and SSL transports).

class asyncio.**DatagramProtocol**
The base class for implementing datagram protocols (for use with e.g. UDP transports).

class asyncio.**SubprocessProtocol**
The base class for implementing protocols communicating with child processes (through a set of unidirectional pipes).

Connection callbacks

These callbacks may be called on *Protocol*, *DatagramProtocol* and *SubprocessProtocol* instances:

BaseProtocol.**connection_made**(*transport*)
链接建立时被调用。

The *transport* argument is the transport representing the connection. You are responsible for storing it somewhere (e.g. as an attribute) if you need to.

BaseProtocol.**connection_lost**(*exc*)
链接丢失或关闭时被调用。

方法的参数是一个异常对象或为*None*。后者意味着收到了常规的 EOF，或者连接被连接的一端取消或关闭。

connection_made() and *connection_lost()* are called exactly once per successful connection. All other callbacks will be called between those two methods, which allows for easier resource management in your protocol implementation.

The following callbacks may be called only on *SubprocessProtocol* instances:

SubprocessProtocol.**pipe_data_received**(*fd*, *data*)
Called when the child process writes data into its stdout or stderr pipe. *fd* is the integer file descriptor of the pipe. *data* is a non-empty bytes object containing the data.

SubprocessProtocol.**pipe_connection_lost**(*fd*, *exc*)
Called when one of the pipes communicating with the child process is closed. *fd* is the integer file descriptor that was closed.

SubprocessProtocol.**process_exited**()
子进程退出时被调用。

Streaming protocols

The following callbacks are called on *Protocol* instances:

Protocol.**data_received**(*data*)
当收到数据时被调用。*data* 为包含入站数据的非空字节串对象。

注解： Whether the data is buffered, chunked or reassembled depends on the transport. In general, you shouldn't rely on specific semantics and instead make your parsing generic and flexible enough. However, data is always received in the correct order.

Protocol.**eof_received**()
Called when the other end signals it won't send any more data (for example by calling write_eof(), if the other end also uses asyncio).

This method may return a false value (including None), in which case the transport will close itself. Conversely, if this method returns a true value, closing the transport is up to the protocol. Since the default implementation returns None, it implicitly closes the connection.

注解: Some transports such as SSL don't support half-closed connections, in which case returning true from this method will not prevent closing the connection.

data_received() can be called an arbitrary number of times during a connection. However, eof_received() is called at most once and, if called, data_received() won't be called after it.

状态机:

start -> *connection_made()* [-> *data_received()* *] [-> *eof_received()* ?] -> *connection_lost()* -> end

Datagram protocols

The following callbacks are called on *DatagramProtocol* instances.

DatagramProtocol.**datagram_received**(*data*, *addr*)
当接收到数据报时被调用。*data* 是包含传入数据的字节串对象。*addr* 是发送数据的对等端地址；实际的格式取决于具体传输。

DatagramProtocol.**error_received**(*exc*)
当前一个发送或接收操作引发*OSError* 时被调用。*exc* 是*OSError* 的实例。

This method is called in rare conditions, when the transport (e.g. UDP) detects that a datagram couldn't be delivered to its recipient. In many conditions though, undeliverable datagrams will be silently dropped.

Flow control callbacks

These callbacks may be called on *Protocol*, *DatagramProtocol* and *SubprocessProtocol* instances:

BaseProtocol.**pause_writing**()
Called when the transport's buffer goes over the high-water mark.

BaseProtocol.**resume_writing**()
Called when the transport's buffer drains below the low-water mark.

pause_writing() and resume_writing() calls are paired —pause_writing() is called once when the buffer goes strictly over the high-water mark (even if subsequent writes increases the buffer size even more), and eventually resume_writing() is called once when the buffer size reaches the low-water mark.

注解: If the buffer size equals the high-water mark, pause_writing() is not called —it must go strictly over. Conversely, resume_writing() is called when the buffer size is equal or lower than the low-water mark. These end conditions are important to ensure that things go as expected when either mark is zero.

注解: On BSD systems (OS X, FreeBSD, etc.) flow control is not supported for *DatagramProtocol*, because send failures caused by writing too many packets cannot be detected easily. The socket always appears 'ready' and excess

packets are dropped; an *OSError* with errno set to *errno.ENOBUFS* may or may not be raised; if it is raised, it will be reported to *DatagramProtocol.error_received()* but otherwise ignored.

Coroutines and protocols

Coroutines can be scheduled in a protocol method using *ensure_future()*, but there is no guarantee made about the execution order. Protocols are not aware of coroutines created in protocol methods and so will not wait for them.

To have a reliable execution order, use *stream objects* in a coroutine with `yield from`. For example, the *StreamWriter.drain()* coroutine can be used to wait until the write buffer is flushed.

Protocol examples

TCP echo client protocol

TCP echo client using the *AbstractEventLoop.create_connection()* method, send data and wait until the connection is closed:

```
import asyncio

class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        print('Stop the event loop')
        self.loop.stop()

loop = asyncio.get_event_loop()
message = 'Hello World!'
coro = loop.create_connection(lambda: EchoClientProtocol(message, loop),
                              '127.0.0.1', 8888)
loop.run_until_complete(coro)
loop.run_forever()
loop.close()
```

The event loop is running twice. The *run_until_complete()* method is preferred in this short example to raise an exception if the server is not listening, instead of having to write a short coroutine to handle the exception and stop the running loop. At *run_until_complete()* exit, the loop is no longer running, so there is no need to stop the loop in case of an error.

参见:

The *TCP echo client using streams* example uses the *asyncio.open_connection()* function.

TCP echo server protocol

TCP echo server using the *AbstractEventLoop.create_server()* method, send back received data and close the connection:

```
import asyncio

class EchoServerClientProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()

loop = asyncio.get_event_loop()
# Each client connection will create a new protocol instance
coro = loop.create_server(EchoServerClientProtocol, '127.0.0.1', 8888)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

`Transport.close()` can be called immediately after *WriteTransport.write()* even if data are not sent yet on the socket: both methods are asynchronous. `yield from` is not needed because these transport methods are not coroutines.

参见:

The *TCP echo server using streams* example uses the *asyncio.start_server()* function.

UDP echo client protocol

UDP echo client using the *AbstractEventLoop.create_datagram_endpoint()* method, send data and close the transport when we received the answer:

```
import asyncio

class EchoClientProtocol:
    def __init__(self, message, loop):
        self.message = message
        self.loop = loop
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Socket closed, stop the event loop")
        loop = asyncio.get_event_loop()
        loop.stop()

loop = asyncio.get_event_loop()
message = "Hello World!"
connect = loop.create_datagram_endpoint(
    lambda: EchoClientProtocol(message, loop),
    remote_addr=('127.0.0.1', 9999))
transport, protocol = loop.run_until_complete(connect)
loop.run_forever()
transport.close()
loop.close()
```

UDP echo server protocol

UDP echo server using the *AbstractEventLoop.create_datagram_endpoint()* method, send back received data:

```
import asyncio

class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
```

(下页继续)

(续上页)

```
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)

loop = asyncio.get_event_loop()
print("Starting UDP server")
# One protocol instance will be created to serve all client requests
listen = loop.create_datagram_endpoint(
    EchoServerProtocol, local_addr=('127.0.0.1', 9999))
transport, protocol = loop.run_until_complete(listen)

try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

transport.close()
loop.close()
```

Register an open socket to wait for data using a protocol

Wait until a socket receives data using the *AbstractEventLoop.create_connection()* method with a protocol, and then close the event loop

```
import asyncio
try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils import socketpair

# Create a pair of connected sockets
rsock, wsock = socketpair()
loop = asyncio.get_event_loop()

class MyProtocol(asyncio.Protocol):
    transport = None

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport (it will call connection_lost())
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed, stop the event loop
        loop.stop()

# Register the socket to wait for data
connect_coro = loop.create_connection(MyProtocol, sock=rsock)
transport, protocol = loop.run_until_complete(connect_coro)

# Simulate the reception of data from the network
```

(下页继续)

(续上页)

```
loop.call_soon(wsock.send, 'abc'.encode())

# Run the event loop
loop.run_forever()

# We are done, close sockets and the event loop
rsock.close()
wsock.close()
loop.close()
```

参见:

The *watch a file descriptor for read events* example uses the low-level `AbstractEventLoop.add_reader()` method to register the file descriptor of a socket.

The *register an open socket to wait for data using streams* example uses high-level streams created by the `open_connection()` function in a coroutine.

18.5.5 Streams (coroutine based API)

Source code: Lib/asyncio/streams.py

Stream functions

注解: The top-level functions in this module are meant as convenience wrappers only; there's really nothing special there, and if they don't do exactly what you want, feel free to copy their code.

coroutine asyncio.**open_connection**(*host=None*, *port=None*, *, *loop=None*, *limit=None*, ***kwds*)

A wrapper for `create_connection()` returning a (reader, writer) pair.

The reader returned is a `StreamReader` instance; the writer is a `StreamWriter` instance.

The arguments are all the usual arguments to `AbstractEventLoop.create_connection()` except *protocol_factory*; most common are positional host and port, with various optional keyword arguments following.

Additional optional keyword arguments are *loop* (to set the event loop instance to use) and *limit* (to set the buffer limit passed to the `StreamReader`).

This function is a *coroutine*.

coroutine asyncio.**start_server**(*client_connected_cb*, *host=None*, *port=None*, *, *loop=None*, *limit=None*, ***kwds*)

Start a socket server, with a callback for each client connected. The return value is the same as `create_server()`.

The *client_connected_cb* parameter is called with two parameters: *client_reader*, *client_writer*. *client_reader* is a `StreamReader` object, while *client_writer* is a `StreamWriter` object. The *client_connected_cb* parameter can either be a plain callback function or a *coroutine function*; if it is a coroutine function, it will be automatically converted into a `Task`.

The rest of the arguments are all the usual arguments to `create_server()` except *protocol_factory*; most common are positional *host* and *port*, with various optional keyword arguments following.

Additional optional keyword arguments are *loop* (to set the event loop instance to use) and *limit* (to set the buffer limit passed to the `StreamReader`).

This function is a *coroutine*.

coroutine asyncio.**open_unix_connection**(*path=None*, *, *loop=None*, *limit=None*, ***kwds*)
A wrapper for `create_unix_connection()` returning a (reader, writer) pair.

See `open_connection()` for information about return value and other details.

This function is a *coroutine*.

Availability: UNIX.

coroutine asyncio.**start_unix_server**(*client_connected_cb*, *path=None*, *, *loop=None*, *limit=None*, ***kwds*)
Start a UNIX Domain Socket server, with a callback for each client connected.

See `start_server()` for information about return value and other details.

This function is a *coroutine*.

Availability: UNIX.

StreamReader

class asyncio.**StreamReader**(*limit=_DEFAULT_LIMIT*, *loop=None*)
This class is *not thread safe*.

The *limit* argument's default value is set to _DEFAULT_LIMIT which is 2**16 (64 KiB)

exception()
Get the exception.

feed_eof()
Acknowledge the EOF.

feed_data(*data*)
Feed *data* bytes in the internal buffer. Any operations waiting for the data will be resumed.

set_exception(*exc*)
Set the exception.

set_transport(*transport*)
Set the transport.

coroutine read(*n=-1*)
读取 *n* 个 byte. 如果没有设置 *n* , 则自动置为 `-1` ， 读至 EOF 并返回所有读取的 byte。

If the EOF was received and the internal buffer is empty, return an empty `bytes` object.

This method is a *coroutine*.

coroutine readline()
读取一行，其中“行”指的是以 `\n` 结尾的字节序列。

If EOF is received, and `\n` was not found, the method will return the partial read bytes.

If the EOF was received and the internal buffer is empty, return an empty `bytes` object.

This method is a *coroutine*.

coroutine readexactly(*n*)
Read exactly *n* bytes. Raise an `IncompleteReadError` if the end of the stream is reached before *n* can be read, the `IncompleteReadError.partial` attribute of the exception contains the partial read bytes.

This method is a *coroutine*.

coroutine readuntil(*separator=b'\n'*)
Read data from the stream until `separator` is found.

成功后，数据和指定的 separator 将从内部缓冲区中删除 (或者说被消费掉)。返回的数据将包括在末尾的指定 separator。

Configured stream limit is used to check result. Limit sets the maximal length of data that can be returned, not counting the separator.

If an EOF occurs and the complete separator is still not found, an *IncompleteReadError* exception will be raised, and the internal buffer will be reset. The *IncompleteReadError.partial* attribute may contain the separator partially.

If the data cannot be read because of over limit, a *LimitOverrunError* exception will be raised, and the data will be left in the internal buffer, so it can be read again.

3.5.2 新版功能.

at_eof()
如果缓冲区为空并且*feed_eof()* 被调用，则返回 `True` 。

StreamWriter

class asyncio.**StreamWriter**(*transport*, *protocol*, *reader*, *loop*)
Wraps a Transport.

This exposes *write()*, *writelines()*, *can_write_eof()*, *write_eof()*, *get_extra_info()* and *close()*. It adds *drain()* which returns an optional *Future* on which you can wait for flow control. It also adds a transport attribute which references the `Transport` directly.

This class is *not thread safe*.

transport
Transport.

can_write_eof()
Return *True* if the transport supports *write_eof()*, *False* if not. See *WriteTransport.can_write_eof()*.

close()
Close the transport: see *BaseTransport.close()*.

coroutine drain()
Let the write buffer of the underlying transport a chance to be flushed.

The intended use is to write:

```
w.write(data)
yield from w.drain()
```

When the size of the transport buffer reaches the high-water limit (the protocol is paused), block until the size of the buffer is drained down to the low-water limit and the protocol is resumed. When there is nothing to wait for, the yield-from continues immediately.

Yielding from *drain()* gives the opportunity for the loop to schedule the write operation and flush the buffer. It should especially be used when a possibly large amount of data is written to the transport, and the coroutine does not yield-from between calls to *write()*.

This method is a *coroutine*.

get_extra_info(*name*, *default=None*)
Return optional transport information: see *BaseTransport.get_extra_info()*.

write(*data*)
Write some *data* bytes to the transport: see *WriteTransport.write()*.

writelines(*data*)
Write a list (or any iterable) of data bytes to the transport: see *WriteTransport.writelines()*.

write_eof()
Close the write end of the transport after flushing buffered data: see *WriteTransport.write_eof()*.

StreamReaderProtocol

class asyncio.**StreamReaderProtocol**(*stream_reader*, *client_connected_cb=None*, *loop=None*)
Trivial helper class to adapt between *Protocol* and *StreamReader*. Subclass of *Protocol*.

stream_reader is a *StreamReader* instance, *client_connected_cb* is an optional function called with (stream_reader, stream_writer) when a connection is made, *loop* is the event loop instance to use.

(This is a helper class instead of making *StreamReader* itself a *Protocol* subclass, because the *StreamReader* has other potential uses, and to prevent the user of the *StreamReader* from accidentally calling inappropriate methods of the protocol.)

IncompleteReadError

exception asyncio.**IncompleteReadError**

Incomplete read error, subclass of *EOFError*.

expected
Total number of expected bytes (*int*).

partial
Read bytes string before the end of stream was reached (*bytes*).

LimitOverrunError

exception asyncio.**LimitOverrunError**
Reached the buffer limit while looking for a separator.

consumed
Total number of to be consumed bytes.

Stream examples

使用流的 TCP 回显客户端

使用*asyncio.open_connection()* 函数的 TCP 回显客户端:

```
import asyncio

@asyncio.coroutine
def tcp_echo_client(message, loop):
    reader, writer = yield from asyncio.open_connection('127.0.0.1', 8888,
                                                        loop=loop)
```

(下页继续)

(续上页)

```
    print('Send: %r' % message)
    writer.write(message.encode())

    data = yield from reader.read(100)
    print('Received: %r' % data.decode())

    print('Close the socket')
    writer.close()

message = 'Hello World!'
loop = asyncio.get_event_loop()
loop.run_until_complete(tcp_echo_client(message, loop))
loop.close()
```

参见:

The *TCP echo client protocol* example uses the *AbstractEventLoop.create_connection()* method.

使用流的 TCP 回显服务器

TCP 回显服务器使用*asyncio.start_server()* 函数:

```
import asyncio

@asyncio.coroutine
def handle_echo(reader, writer):
    data = yield from reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')
    print("Received %r from %r" % (message, addr))

    print("Send: %r" % message)
    writer.write(data)
    yield from writer.drain()

    print("Close the client socket")
    writer.close()

loop = asyncio.get_event_loop()
coro = asyncio.start_server(handle_echo, '127.0.0.1', 8888, loop=loop)
server = loop.run_until_complete(coro)

# Serve requests until Ctrl+C is pressed
print('Serving on {}'.format(server.sockets[0].getsockname()))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass

# Close the server
server.close()
loop.run_until_complete(server.wait_closed())
loop.close()
```

参见:

The *TCP echo server protocol* example uses the *AbstractEventLoop.create_server()* method.

获取 HTTP 标头

查询命令行传入 URL 的 HTTP 标头的简单示例:

```
import asyncio
import urllib.parse
import sys

@asyncio.coroutine
def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        connect = asyncio.open_connection(url.hostname, 443, ssl=True)
    else:
        connect = asyncio.open_connection(url.hostname, 80)
    reader, writer = yield from connect
    query = ('HEAD {path} HTTP/1.0\r\n'
             'Host: {hostname}\r\n'
             '\r\n').format(path=url.path or '/', hostname=url.hostname)
    writer.write(query.encode('latin-1'))
    while True:
        line = yield from reader.readline()
        if not line:
            break
        line = line.decode('latin1').rstrip()
        if line:
            print('HTTP header> %s' % line)

    # Ignore the body, close the socket
    writer.close()

url = sys.argv[1]
loop = asyncio.get_event_loop()
task = asyncio.ensure_future(print_http_headers(url))
loop.run_until_complete(task)
loop.close()
```

用法:

```
python example.py http://example.com/path/page.html
```

或使用 HTTPS:

```
python example.py https://example.com/path/page.html
```

注册一个打开的套接字以等待使用流的数据

使用*open_connection()* 函数实现等待直到套接字接收到数据的协程:

```
import asyncio
try:
    from socket import socketpair
except ImportError:
    from asyncio.windows_utils import socketpair
```

(下页继续)

(续上页)

```
@asyncio.coroutine
def wait_for_data(loop):
    # Create a pair of connected sockets
    rsock, wsock = socketpair()

    # Register the open socket to wait for data
    reader, writer = yield from asyncio.open_connection(sock=rsock, loop=loop)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = yield from reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()

    # Close the second socket
    wsock.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(wait_for_data(loop))
loop.close()
```

参见:

The *register an open socket to wait for data using a protocol* example uses a low-level protocol created by the `AbstractEventLoop.create_connection()` method.

The *watch a file descriptor for read events* example uses the low-level `AbstractEventLoop.add_reader()` method to register the file descriptor of a socket.

18.5.6 Subprocess

Source code: Lib/asyncio/subprocess.py

Windows event loop

On Windows, the default event loop is `SelectorEventLoop` which does not support subprocesses. `ProactorEventLoop` should be used instead. Example to use it on Windows:

```
import asyncio, sys

if sys.platform == 'win32':
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
```

参见:

Available event loops and *Platform support*.

Create a subprocess: high-level API using Process

coroutine asyncio.**create_subprocess_exec**(**args*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

创建一个子进程。

The *limit* parameter sets the buffer limit passed to the `StreamReader`. See `AbstractEventLoop.subprocess_exec()` for other parameters.

返回一个`Process`实例。

This function is a *coroutine*.

coroutine asyncio.**create_subprocess_shell**(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

Run the shell command *cmd*.

The *limit* parameter sets the buffer limit passed to the `StreamReader`. See `AbstractEventLoop.subprocess_shell()` for other parameters.

返回一个`Process`实例。

It is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities. The `shlex.quote()` function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

This function is a *coroutine*.

Use the `AbstractEventLoop.connect_read_pipe()` and `AbstractEventLoop.connect_write_pipe()` methods to connect pipes.

Create a subprocess: low-level API using subprocess.Popen

Run subprocesses asynchronously using the `subprocess` module.

coroutine AbstractEventLoop.**subprocess_exec**(*protocol_factory*, **args*, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

Create a subprocess from one or more string arguments (character strings or bytes strings encoded to the *filesystem encoding*), where the first string specifies the program to execute, and the remaining strings specify the program's arguments. (Thus, together the string arguments form the `sys.argv` value of the program, assuming it is a Python script.) This is similar to the standard library `subprocess.Popen` class called with shell=False and the list of strings passed as the first argument; however, where `Popen` takes a single argument which is list of strings, `subprocess_exec()` takes multiple string arguments.

The *protocol_factory* must instantiate a subclass of the `asyncio.SubprocessProtocol` class.

Other parameters:

- *stdin*: Either a file-like object representing the pipe to be connected to the subprocess's standard input stream using `connect_write_pipe()`, or the constant `subprocess.PIPE` (the default). By default a new pipe will be created and connected.
- *stdout*: Either a file-like object representing the pipe to be connected to the subprocess's standard output stream using `connect_read_pipe()`, or the constant `subprocess.PIPE` (the default). By default a new pipe will be created and connected.
- *stderr*: Either a file-like object representing the pipe to be connected to the subprocess's standard error stream using `connect_read_pipe()`, or one of the constants `subprocess.PIPE` (the default) or

subprocess.STDOUT. By default a new pipe will be created and connected. When *subprocess.STDOUT* is specified, the subprocess' standard error stream will be connected to the same pipe as the standard output stream.

- All other keyword arguments are passed to *subprocess.Popen* without interpretation, except for *bufsize*, *universal_newlines* and *shell*, which should not be specified at all.

Returns a pair of `(transport, protocol)`, where *transport* is an instance of *BaseSubprocessTransport*.

This method is a *coroutine*.

See the constructor of the *subprocess.Popen* class for parameters.

coroutine AbstractEventLoop.**subprocess_shell**(*protocol_factory*, *cmd*, *, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

Create a subprocess from *cmd*, which is a character string or a bytes string encoded to the *filesystem encoding*, using the platform's "shell" syntax. This is similar to the standard library *subprocess.Popen* class called with `shell=True`.

The *protocol_factory* must instantiate a subclass of the *asyncio.SubprocessProtocol* class.

See *subprocess_exec()* for more details about the remaining arguments.

Returns a pair of `(transport, protocol)`, where *transport* is an instance of *BaseSubprocessTransport*.

It is the application's responsibility to ensure that all whitespace and metacharacters are quoted appropriately to avoid shell injection vulnerabilities. The *shlex.quote()* function can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

This method is a *coroutine*.

参见:

The *AbstractEventLoop.connect_read_pipe()* and *AbstractEventLoop.connect_write_pipe()* methods.

常量

asyncio.subprocess.**PIPE**
Special value that can be used as the *stdin*, *stdout* or *stderr* argument to *create_subprocess_shell()* and *create_subprocess_exec()* and indicates that a pipe to the standard stream should be opened.

asyncio.subprocess.**STDOUT**
Special value that can be used as the *stderr* argument to *create_subprocess_shell()* and *create_subprocess_exec()* and indicates that standard error should go into the same handle as standard output.

asyncio.subprocess.**DEVNULL**
Special value that can be used as the *stdin*, *stdout* or *stderr* argument to *create_subprocess_shell()* and *create_subprocess_exec()* and indicates that the special file *os.devnull* will be used.

Process

class asyncio.subprocess.**Process**

A subprocess created by the *create_subprocess_exec()* or the *create_subprocess_shell()* function.

The API of the *Process* class was designed to be close to the API of the *subprocess.Popen* class, but there are some differences:

- There is no explicit *poll()* method
- The *communicate()* and *wait()* methods don't take a *timeout* parameter: use the *wait_for()* function
- The *universal_newlines* parameter is not supported (only bytes strings are supported)
- The *wait()* method of the *Process* class is asynchronous whereas the *wait()* method of the *Popen* class is implemented as a busy loop.

This class is *not thread safe*. See also the *Subprocess and threads* section.

coroutine wait()

Wait for child process to terminate. Set and return *returncode* attribute.

This method is a *coroutine*.

注解: This will deadlock when using `stdout=PIPE` or `stderr=PIPE` and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data. Use the *communicate()* method when using pipes to avoid that.

---

coroutine communicate(*input=None*)

Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional *input* argument should be data to be sent to the child process, or `None`, if no data should be sent to the child. The type of *input* must be bytes.

communicate() returns a tuple `(stdout_data, stderr_data)`.

If a *BrokenPipeError* or *ConnectionResetError* exception is raised when writing *input* into stdin, the exception is ignored. It occurs when the process exits before all data are written into stdin.

Note that if you want to send data to the process's stdin, you need to create the Process object with `stdin=PIPE`. Similarly, to get anything other than `None` in the result tuple, you need to give `stdout=PIPE` and/or `stderr=PIPE` too.

This method is a *coroutine*.

注解: The data read is buffered in memory, so do not use this method if the data size is large or unlimited.

在 3.4.2 版更改: The method now ignores *BrokenPipeError* and *ConnectionResetError*.

send_signal(*signal*)

将信号 *signal* 发送给子进程。

注解: 在 Windows 上，`SIGTERM` 是*terminate()* 的别名。`CTRL_C_EVENT` 和 `CTRL_BREAK_EVENT` 可被发送给创建时附带 *creationflags* 形参且其中包括 `CREATE_NEW_PROCESS_GROUP` 的进程。

---

terminate()
Stop the child. On Posix OSs the method sends `signal.SIGTERM` to the child. On Windows the Win32 API function `TerminateProcess()` is called to stop the child.

kill()
Kills the child. On Posix OSs the function sends `SIGKILL` to the child. On Windows *kill()* is an alias for *terminate()*.

stdin
Standard input stream (*StreamWriter*), `None` if the process was created with `stdin=None`.

stdout
Standard output stream (*StreamReader*), `None` if the process was created with `stdout=None`.

stderr
Standard error stream (*StreamReader*), `None` if the process was created with `stderr=None`.

> **警告:** Use the *communicate()* method rather than *.stdin.write*, *.stdout.read* or *.stderr.read* to avoid deadlocks due to streams pausing reading or writing and blocking the child process.

pid
The identifier of the process.

Note that for processes created by the *create_subprocess_shell()* function, this attribute is the process identifier of the spawned shell.

returncode
Return code of the process when it exited. A `None` value indicates that the process has not terminated yet.

A negative value `-N` indicates that the child was terminated by signal `N` (Unix only).

Subprocess and threads

asyncio supports running subprocesses from different threads, but there are limits:

- An event loop must run in the main thread
- The child watcher must be instantiated in the main thread, before executing subprocesses from other threads. Call the `get_child_watcher()` function in the main thread to instantiate the child watcher.

The *asyncio.subprocess.Process* class is not thread safe.

参见:

asyncio 中的并发和多线程 章节。

Subprocess examples

Subprocess using transport and protocol

Example of a subprocess protocol using to get the output of a subprocess and to wait for the subprocess exit. The subprocess is created by the *AbstractEventLoop.subprocess_exec()* method:

```
import asyncio
import sys
```

(下页继续)

(续上页)

```
class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exit_future.set_result(True)

@asyncio.coroutine
def get_date(loop):
    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by the protocol DateProtocol,
    # redirect the standard output into a pipe
    create = loop.subprocess_exec(lambda: DateProtocol(exit_future),
                                  sys.executable, '-c', code,
                                  stdin=None, stderr=None)
    transport, protocol = yield from create

    # Wait for the subprocess exit using the process_exited() method
    # of the protocol
    yield from exit_future

    # Close the stdout pipe
    transport.close()

    # Read the output which was collected by the pipe_data_received()
    # method of the protocol
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

if sys.platform == "win32":
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
else:
    loop = asyncio.get_event_loop()

date = loop.run_until_complete(get_date(loop))
print("Current date: %s" % date)
loop.close()
```

Subprocess using streams

Example using the *Process* class to control the subprocess and the *StreamReader* class to read from the standard output. The subprocess is created by the *create_subprocess_exec()* function:

```
import asyncio.subprocess
import sys

@asyncio.coroutine
def get_date():
```

(下页继续)

(续上页)

```
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess, redirect the standard output into a pipe
    create = asyncio.create_subprocess_exec(sys.executable, '-c', code,
                                            stdout=asyncio.subprocess.PIPE)
    proc = yield from create

    # Read one line of output
    data = yield from proc.stdout.readline()
    line = data.decode('ascii').rstrip()

    # Wait for the subprocess exit
    yield from proc.wait()
    return line

if sys.platform == "win32":
    loop = asyncio.ProactorEventLoop()
    asyncio.set_event_loop(loop)
else:
    loop = asyncio.get_event_loop()

date = loop.run_until_complete(get_date())
print("Current date: %s" % date)
loop.close()
```

18.5.7 Synchronization primitives

Source code: Lib/asyncio/locks.py

Locks:

- *Lock*
- *Event*
- *Condition*

Semaphores:

- *Semaphore*
- *BoundedSemaphore*

asyncio lock API was designed to be close to classes of the *threading* module (*Lock*, *Event*, *Condition*, *Semaphore*, *BoundedSemaphore*), but it has no *timeout* parameter. The *asyncio.wait_for()* function can be used to cancel a task after a timeout.

Locks

Lock

class asyncio.**Lock**(*, *loop=None*)

Primitive lock objects.

A primitive lock is a synchronization primitive that is not owned by a particular coroutine when locked. A primitive lock is in one of two states, 'locked' or 'unlocked'.

It is created in the unlocked state. It has two basic methods, *acquire()* and *release()*. When the state is unlocked, acquire() changes the state to locked and returns immediately. When the state is locked, acquire() blocks until a call to release() in another coroutine changes it to unlocked, then the acquire() call resets it to locked and returns. The release() method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a *RuntimeError* will be raised.

When more than one coroutine is blocked in acquire() waiting for the state to turn to unlocked, only one coroutine proceeds when a release() call resets the state to unlocked; first coroutine which is blocked in acquire() is being processed.

acquire() is a coroutine and should be called with `yield from`.

Locks also support the context management protocol. `(yield from lock)` should be used as the context manager expression.

This class is *not thread safe*.

Usage:

```
lock = Lock()
...
yield from lock
try:
    ...
finally:
    lock.release()
```

Context manager usage:

```
lock = Lock()
...
with (yield from lock):
    ...
```

Lock objects can be tested for locking state:

```
if not lock.locked():
    yield from lock
else:
    # lock is acquired
    ...
```

locked()

Return `True` if the lock is acquired.

coroutine acquire()

Acquire a lock.

This method blocks until the lock is unlocked, then sets it to locked and returns `True`.

This method is a *coroutine*.

release()
Release a lock.

When the lock is locked, reset it to unlocked, and return. If any other coroutines are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

在未锁定的锁调用时，会引发*RuntimeError* 异常。

There is no return value.

Event

class asyncio.**Event**(*, *loop=None*)
An Event implementation, asynchronous equivalent to *threading.Event*.

Class implementing event objects. An event manages a flag that can be set to true with the *set()* method and reset to false with the *clear()* method. The *wait()* method blocks until the flag is true. The flag is initially false.

This class is *not thread safe*.

clear()
Reset the internal flag to false. Subsequently, coroutines calling *wait()* will block until *set()* is called to set the internal flag to true again.

is_set()
Return `True` if and only if the internal flag is true.

set()
Set the internal flag to true. All coroutines waiting for it to become true are awakened. Coroutine that call *wait()* once the flag is true will not block at all.

coroutine wait()
Block until the internal flag is true.

If the internal flag is true on entry, return `True` immediately. Otherwise, block until another coroutine calls *set()* to set the flag to true, then return `True`.

This method is a *coroutine*.

Condition

class asyncio.**Condition**(*lock=None*, *, *loop=None*)
A Condition implementation, asynchronous equivalent to *threading.Condition*.

This class implements condition variable objects. A condition variable allows one or more coroutines to wait until they are notified by another coroutine.

If the *lock* argument is given and not `None`, it must be a *Lock* object, and it is used as the underlying lock. Otherwise, a new *Lock* object is created and used as the underlying lock.

This class is *not thread safe*.

coroutine acquire()
获取下层的锁。

This method blocks until the lock is unlocked, then sets it to locked and returns `True`.

This method is a *coroutine*.

notify(*n=1*)
By default, wake up one coroutine waiting on this condition, if any. If the calling coroutine has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method wakes up at most *n* of the coroutines waiting for the condition variable; it is a no-op if no coroutines are waiting.

注解： An awakened coroutine does not actually return from its *wait()* call until it can reacquire the lock. Since *notify()* does not release the lock, its caller should.

---

locked()
如果下层的锁已被获取则返回 `True`。

notify_all()
Wake up all coroutines waiting on this condition. This method acts like *notify()*, but wakes up all waiting coroutines instead of one. If the calling coroutine has not acquired the lock when this method is called, a *RuntimeError* is raised.

release()
释放下层的锁。

When the lock is locked, reset it to unlocked, and return. If any other coroutines are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

在未锁定的锁调用时，会引发*RuntimeError* 异常。

There is no return value.

coroutine wait()
等待直至收到通知。

If the calling coroutine has not acquired the lock when this method is called, a *RuntimeError* is raised.

This method releases the underlying lock, and then blocks until it is awakened by a *notify()* or *notify_all()* call for the same condition variable in another coroutine. Once awakened, it re-acquires the lock and returns `True`.

This method is a *coroutine*.

coroutine wait_for(*predicate*)
Wait until a predicate becomes true.

The predicate should be a callable which result will be interpreted as a boolean value. The final predicate value is the return value.

This method is a *coroutine*.

Semaphores

Semaphore

class asyncio.**Semaphore**(*value=1*, *, *loop=None*)
A Semaphore implementation.

A semaphore manages an internal counter which is decremented by each *acquire()* call and incremented by each *release()* call. The counter can never go below zero; when *acquire()* finds that it is zero, it blocks, waiting until some other coroutine calls *release()*.

Semaphores also support the context management protocol.

The optional argument gives the initial value for the internal counter; it defaults to `1`. If the value given is less than `0`, *`ValueError`* is raised.

This class is *not thread safe*.

coroutine acquire()
获取一个信号量。

If the internal counter is larger than zero on entry, decrement it by one and return `True` immediately. If it is zero on entry, block, waiting until some other coroutine has called *`release()`* to make it larger than `0`, and then return `True`.

This method is a *coroutine*.

locked()
如果信号量对象无法被立即获取则返回 `True`。

release()
Release a semaphore, incrementing the internal counter by one. When it was zero on entry and another coroutine is waiting for it to become larger than zero again, wake up that coroutine.

BoundedSemaphore

class asyncio.**BoundedSemaphore**(*value=1*, *, *loop=None*)
A bounded semaphore implementation. Inherit from *`Semaphore`*.

This raises *`ValueError`* in *`release()`* if it would increase the value above the initial value.

18.5.8 队列集

Source code: Lib/asyncio/queues.py

Queues:

- *`Queue`*
- *`PriorityQueue`*
- *`LifoQueue`*

asyncio queue API was designed to be close to classes of the *`queue`* module (*`Queue`*, *`PriorityQueue`*, *`LifoQueue`*), but it has no *timeout* parameter. The *`asyncio.wait_for()`* function can be used to cancel a task after a timeout.

队列

class asyncio.**Queue**(*maxsize=0*, *, *loop=None*)
A queue, useful for coordinating producer and consumer coroutines.

If *maxsize* is less than or equal to zero, the queue size is infinite. If it is an integer greater than `0`, then `yield from put()` will block when the queue reaches *maxsize*, until an item is removed by *`get()`*.

Unlike the standard library *`queue`*, you can reliably know this Queue's size with *`qsize()`*, since your single-threaded asyncio application won't be interrupted between calling *`qsize()`* and doing an operation on the Queue.

这个类不是线程安全的（*not thread safe*）。

在 3.4.4 版更改: New *`join()`* and *`task_done()`* methods.

empty()
如果队列为空返回 `True` ，否则返回 `False` 。

full()
如果有*maxsize* 个条目在队列中，则返回 `True` 。

注解： If the Queue was initialized with `maxsize=0` (the default), then *full()* is never `True`.

---

coroutine get()
从队列中删除并返回一个元素。如果队列为空，则等待，直到队列中有元素。

This method is a *coroutine*.

参见：

The *empty()* method.

get_nowait()
Remove and return an item from the queue.

立即返回一个队列中的元素，如果队列内有值，否则引发异常*QueueEmpty* 。

coroutine join()
Block until all items in the queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls *task_done()* to indicate that the item was retrieved and all work on it is complete. When the count of unfinished tasks drops to zero, *join()* unblocks.

This method is a *coroutine*.

3.4.4 新版功能.

coroutine put(*item*)
Put an item into the queue. If the queue is full, wait until a free slot is available before adding item.

This method is a *coroutine*.

参见：

The *full()* method.

put_nowait(*item*)
不阻塞的放一个元素入队列。

如果没有立即可用的空闲槽，引发*QueueFull* 异常。

qsize()
Number of items in the queue.

task_done()
表明前面排队的任务已经完成，即 get 出来的元素相关操作已经完成。

由队列使用者控制。每个*get()* 用于获取一个任务，任务最后调用*task_done()* 告诉队列，这个任务已经完成。

如果*join()* 当前正在阻塞，在所有条目都被处理后，将解除阻塞 (意味着每个*put()* 进队列的条目的*task_done()* 都被收到)。

如果被调用的次数多于放入队列中的项目数量，将引发*ValueError* 。

3.4.4 新版功能.

maxsize
队列中可存放的元素数量。

PriorityQueue

class asyncio.**PriorityQueue**
A subclass of *Queue*; retrieves entries in priority order (lowest first).

Entries are typically tuples of the form: (priority number, data).

LifoQueue

class asyncio.**LifoQueue**
A subclass of *Queue* that retrieves most recently added entries first.

异常

exception asyncio.**QueueEmpty**
Exception raised when the *get_nowait()* method is called on a *Queue* object which is empty.

exception asyncio.**QueueFull**
Exception raised when the *put_nowait()* method is called on a *Queue* object which is full.

18.5.9 Develop with asyncio

Asynchronous programming is different than classical "sequential" programming. This page lists common traps and explains how to avoid them.

Debug mode of asyncio

The implementation of *asyncio* has been written for performance. In order to ease the development of asynchronous code, you may wish to enable *debug mode*.

To enable all debug checks for an application:

- Enable the asyncio debug mode globally by setting the environment variable `PYTHONASYNCIODEBUG` to `1`, or by calling *AbstractEventLoop.set_debug()*.
- Set the log level of the *asyncio logger* to `logging.DEBUG`. For example, call `logging.basicConfig(level=logging.DEBUG)` at startup.
- Configure the *warnings* module to display *ResourceWarning* warnings. For example, use the `-Wdefault` command line option of Python to display them.

Examples debug checks:

- Log *coroutines defined but never "yielded from"*
- *call_soon()* and *call_at()* methods raise an exception if they are called from the wrong thread.
- Log the execution time of the selector
- Log callbacks taking more than 100 ms to be executed. The `AbstractEventLoop.slow_callback_duration` attribute is the minimum duration in seconds of "slow" callbacks.
- *ResourceWarning* warnings are emitted when transports and event loops are *not closed explicitly*.

参见:

The *AbstractEventLoop.set_debug()* method and the *asyncio logger*.

Cancellation

Cancellation of tasks is not common in classic programming. In asynchronous programming, not only is it something common, but you have to prepare your code to handle it.

Futures and tasks can be cancelled explicitly with their *Future.cancel()* method. The *wait_for()* function cancels the waited task when the timeout occurs. There are many other cases where a task can be cancelled indirectly.

Don't call *set_result()* or *set_exception()* method of *Future* if the future is cancelled: it would fail with an exception. For example, write:

```
if not fut.cancelled():
    fut.set_result('done')
```

Don't schedule directly a call to the *set_result()* or the *set_exception()* method of a future with *AbstractEventLoop.call_soon()*: the future can be cancelled before its method is called.

If you wait for a future, you should check early if the future was cancelled to avoid useless operations. Example:

```
@coroutine
def slow_operation(fut):
    if fut.cancelled():
        return
    # ... slow computation ...
    yield from fut
    # ...
```

The *shield()* function can also be used to ignore cancellation.

Concurrency and multithreading

An event loop runs in a thread and executes all callbacks and tasks in the same thread. While a task is running in the event loop, no other task is running in the same thread. But when the task uses `yield from`, the task is suspended and the event loop executes the next task.

To schedule a callback from a different thread, the *AbstractEventLoop.call_soon_threadsafe()* method should be used. Example:

```
loop.call_soon_threadsafe(callback, *args)
```

Most asyncio objects are not thread safe. You should only worry if you access objects outside the event loop. For example, to cancel a future, don't call directly its *Future.cancel()* method, but:

```
loop.call_soon_threadsafe(fut.cancel)
```

为了能够处理信号和执行子进程，事件循环必须运行于主线程中。

To schedule a coroutine object from a different thread, the *run_coroutine_threadsafe()* function should be used. It returns a *concurrent.futures.Future* to access the result:

```
future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
result = future.result(timeout)  # Wait for the result with a timeout
```

The *AbstractEventLoop.run_in_executor()* method can be used with a thread pool executor to execute a callback in different thread to not block the thread of the event loop.

参见:

The *Synchronization primitives* section describes ways to synchronize tasks.

The *Subprocess and threads* section lists asyncio limitations to run subprocesses from different threads.

Handle blocking functions correctly

Blocking functions should not be called directly. For example, if a function blocks for 1 second, other tasks are delayed by 1 second which can have an important impact on reactivity.

For networking and subprocesses, the *asyncio* module provides high-level APIs like *protocols*.

An executor can be used to run a task in a different thread or even in a different process, to not block the thread of the event loop. See the *AbstractEventLoop.run_in_executor()* method.

参见:

The *Delayed calls* section details how the event loop handles time.

日志记录

The *asyncio* module logs information with the *logging* module in the logger `'asyncio'`.

The default log level for the *asyncio* module is `logging.INFO`. For those not wanting such verbosity from *asyncio* the log level can be changed. For example, to change the level to `logging.WARNING`:

```
logging.getLogger('asyncio').setLevel(logging.WARNING)
```

Detect coroutine objects never scheduled

When a coroutine function is called and its result is not passed to *ensure_future()* or to the *AbstractEventLoop.create_task()* method, the execution of the coroutine object will never be scheduled which is probably a bug. *Enable the debug mode of asyncio* to *log a warning* to detect it.

Example with the bug:

```
import asyncio

@asyncio.coroutine
def test():
    print("never scheduled")

test()
```

调试模式的输出:

```
Coroutine test() at test.py:3 was never yielded from
Coroutine object created at (most recent call last):
  File "test.py", line 7, in <module>
    test()
```

The fix is to call the *ensure_future()* function or the *AbstractEventLoop.create_task()* method with the coroutine object.

参见:

Pending task destroyed.

Detect exceptions never consumed

Python usually calls *sys.excepthook()* on unhandled exceptions. If *Future.set_exception()* is called, but the exception is never consumed, *sys.excepthook()* is not called. Instead, *a log is emitted* when the future is deleted by the garbage collector, with the traceback where the exception was raised.

Example of unhandled exception:

```
import asyncio

@asyncio.coroutine
def bug():
    raise Exception("not consumed")

loop = asyncio.get_event_loop()
asyncio.ensure_future(bug())
loop.run_forever()
loop.close()
```

输出:

```
Task exception was never retrieved
future: <Task finished coro=<coro() done, defined at asyncio/coroutines.py:139>␣
↪exception=Exception('not consumed',)>
Traceback (most recent call last):
  File "asyncio/tasks.py", line 237, in _step
    result = next(coro)
  File "asyncio/coroutines.py", line 141, in coro
    res = func(*args, **kw)
  File "test.py", line 5, in bug
    raise Exception("not consumed")
Exception: not consumed
```

Enable the debug mode of asyncio to get the traceback where the task was created. Output in debug mode:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3> exception=Exception(
↪'not consumed',) created at test.py:8>
source_traceback: Object created at (most recent call last):
  File "test.py", line 8, in <module>
    asyncio.ensure_future(bug())
Traceback (most recent call last):
  File "asyncio/tasks.py", line 237, in _step
    result = next(coro)
  File "asyncio/coroutines.py", line 79, in __next__
    return next(self.gen)
  File "asyncio/coroutines.py", line 141, in coro
    res = func(*args, **kw)
  File "test.py", line 5, in bug
    raise Exception("not consumed")
Exception: not consumed
```

There are different options to fix this issue. The first option is to chain the coroutine in another coroutine and use classic try/except:

```
@asyncio.coroutine
def handle_exception():
    try:
        yield from bug()
    except Exception:
        print("exception consumed")

loop = asyncio.get_event_loop()
asyncio.ensure_future(handle_exception())
loop.run_forever()
loop.close()
```

Another option is to use the *AbstractEventLoop.run_until_complete()* function:

```
task = asyncio.ensure_future(bug())
try:
    loop.run_until_complete(task)
except Exception:
    print("exception consumed")
```

参见:

The *Future.exception()* method.

Chain coroutines correctly

When a coroutine function calls other coroutine functions and tasks, they should be chained explicitly with `yield from`. Otherwise, the execution is not guaranteed to be sequential.

Example with different bugs using *asyncio.sleep()* to simulate slow operations:

```
import asyncio

@asyncio.coroutine
def create():
    yield from asyncio.sleep(3.0)
    print("(1) create file")

@asyncio.coroutine
def write():
    yield from asyncio.sleep(1.0)
    print("(2) write into file")

@asyncio.coroutine
def close():
    print("(3) close file")

@asyncio.coroutine
def test():
    asyncio.ensure_future(create())
    asyncio.ensure_future(write())
    asyncio.ensure_future(close())
    yield from asyncio.sleep(2.0)
    loop.stop()
```

(下页继续)

(续上页)

```
loop = asyncio.get_event_loop()
asyncio.ensure_future(test())
loop.run_forever()
print("Pending tasks at exit: %s" % asyncio.Task.all_tasks(loop))
loop.close()
```

Expected output:

```
(1) create file
(2) write into file
(3) close file
Pending tasks at exit: set()
```

Actual output:

```
(3) close file
(2) write into file
Pending tasks at exit: {<Task pending create() at test.py:7 wait_for=<Future pending␣
↪cb=[Task._wakeup()]>>}
Task was destroyed but it is pending!
task: <Task pending create() done at test.py:5 wait_for=<Future pending cb=[Task._
↪wakeup()]>>
```

The loop stopped before the `create()` finished, `close()` has been called before `write()`, whereas coroutine functions were called in this order: `create()`, `write()`, `close()`.

To fix the example, tasks must be marked with `yield from`:

```
@asyncio.coroutine
def test():
    yield from asyncio.ensure_future(create())
    yield from asyncio.ensure_future(write())
    yield from asyncio.ensure_future(close())
    yield from asyncio.sleep(2.0)
    loop.stop()
```

Or without `asyncio.ensure_future()`:

```
@asyncio.coroutine
def test():
    yield from create()
    yield from write()
    yield from close()
    yield from asyncio.sleep(2.0)
    loop.stop()
```

Pending task destroyed

If a pending task is destroyed, the execution of its wrapped *coroutine* did not complete. It is probably a bug and so a warning is logged.

Example of log:

```
Task was destroyed but it is pending!
task: <Task pending coro=<kill_me() done, defined at test.py:5> wait_for=<Future␣
↪pending cb=[Task._wakeup()]>>
```

Enable the debug mode of asyncio to get the traceback where the task was created. Example of log in debug mode:

```
Task was destroyed but it is pending!
source_traceback: Object created at (most recent call last):
  File "test.py", line 15, in <module>
    task = asyncio.ensure_future(coro, loop=loop)
task: <Task pending coro=<kill_me() done, defined at test.py:5> wait_for=<Future␣
↪pending cb=[Task._wakeup()] created at test.py:7> created at test.py:15>
```

参见:

Detect coroutine objects never scheduled.

Close transports and event loops

When a transport is no more needed, call its `close()` method to release resources. Event loops must also be closed explicitly.

If a transport or an event loop is not closed explicitly, a *ResourceWarning* warning will be emitted in its destructor. By default, *ResourceWarning* warnings are ignored. The *Debug mode of asyncio* section explains how to display them.

参见:

The *asyncio* module was designed in **PEP 3156**. For a motivational primer on transports and protocols, see **PEP 3153**.

18.6 asyncore —异步 socket 处理器

源码: Lib/asyncore.py

3.6 版后已移除: 请使用*asyncio* 替代。

注解: 该模块仅为提供向后兼容。我们推荐在新代码中使用*asyncio* 。

该模块提供用于编写异步套接字服务客户端与服务端的基础构件。

只有两种方法让单个处理器上的程序“同一时间完成不止一件事”。多线程编程是最简单和最流行的方法，但是还有另一种非常不同的技术，它可以让你拥有多线程的几乎所有优点，而无需实际使用多线程。它仅仅在你的程序主要受 I/O 限制时有用，那么。如果你的程序受处理器限制，那么先发制人的预定线程可能就是你真正需要的。但是，网络服务器很少受处理器限制。

如果你的操作系统在其 I/O 库中支持 `select()` 系统调用（几乎所有操作系统），那么你可以使用它来同时处理多个通信通道；在 I/O 正在“后台”时进行其他工作。虽然这种策略看起来很奇怪和复杂，特别是起初，

它在很多方面比多线程编程更容易理解和控制。*asyncore* 模块为您解决了许多难题，使得构建复杂的高性能网络服务器和客户端的任务变得轻而易举。对于“会话”应用程序和协议，伴侣*asynchat* 模块是非常宝贵的。

这两个模块背后的基本思想是创建一个或多个网络 *通道*，类的实例*asyncore.dispatcher* 和*asynchat.async_chat* 。创建通道会将它们添加到全局映射中，如果你不为它提供自己的 *映射*，则由*loop()* 函数使用。

一旦创建了初始通道，调用*loop()* 函数将激活通道服务，该服务将一直持续到最后一个通道（包括在异步服务期间已添加到映射中的任何通道）关闭。

asyncore.**loop**([*timeout*[, *use_poll*[, *map*[, *count*]]]])
进入一个轮询循环，其在循环计数超出或所有打开的通道关闭后终止。所有参数都是可选的。*count* 形参默认为 None ，导致循环仅在所有通道关闭时终止。*timeout* 形参为适当的*select()* 或*poll()* 调用设置超时参数，以秒为单位; 默认值为 30 秒。*use_poll* 形参，如果为 True ，则表示*poll()* 应优先使用*select()* (默认为 “False“)。

map 形参是一个条目为所监视通道的字典。当通道关闭时它们会被从映射中删除。如果省略 *map*，则会使用一个全局映射。通道 (*asyncore.dispatcher*, *asynchat.async_chat* 及其子类的实例) 可以在映射中任意混合。

class asyncore.**dispatcher**
dispatcher 类是对低层级套接字对象的轻量包装器。要让它更有用处，可以从异步循环调用一些事件处理方法。在其他方面，它可以被当作是普通的非阻塞型套接字对象。

在特定时间或特定连接状态下触发的低层级事件可通知异步循环发生了特定的高层级事件。例如，如果我们请求了一个套接字以连接到另一台主机，我们会在套接字首次变得可写时得知连接已建立（在此刻你将知道可以向其写入并预期能够成功）。包含的高层级事件有:

Event	描述
handle_connect()	由首个读取或写入事件所包含
handle_close()	由不带可用数据的读取事件引起
handle_accepted()	由在监听套接字上的读取事件引起

在异步处理过程中，每个已映射通道的*readable()* 和*writable()* 方法会被用来确定是否要将通道的套接字添加到已执行 select() 或 poll() 用于读取和写入事件的通道列表中。

因此，通道事件的集合要大于基本套接字事件。可以在你的子类中被重载的全部方法集合如下:

handle_read()
当异步循环检测到通道的套接字上的 read() 调用将要成功时会被调用。

handle_write()
当异步循环检测到一个可写套接字可以被写入时会被调用。通常此方法将实现必要的缓冲机制以保证运行效率。例如:

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt()
当一个套接字连接存在带外（OOB）数据时会被调用。这几乎从来不会发生，因为 OOB 虽然受支持但很少被使用。

handle_connect()
当活动打开方的套接字实际建立连接时会被调用。可能会发送一条“欢迎”消息，或者向远程端点发起协议协商等。

handle_close()
当套接字关闭时会被调用。

handle_error()
当一个异常被引发并且未获得其他处理时会被调用。默认版本将打印精简的回溯信息。

handle_accept()
当可以与发起对本地端点的*connect()* 调用的新远程端点建立连接时会在侦听通道（被动打开方）上被调用。在 3.2 版中已被弃用；请改用*handle_accepted()*。

3.2 版后已移除.

handle_accepted(*sock*, *addr*)
当与发起对本地端点的*connect()* 调用的新远程端点已建立连接时会在侦听通道（被动打开方）上被调用。*sock* 是可被用于在连接上发送和接收数据的 *新建*套接字对象，而 *addr* 是绑定到连接另一端的套接字的地址。

3.2 新版功能.

readable()
每次在异步循环之外被调用以确定是否应当将一个通道的套接字添加到可能在其上发生读取事件的列表中。默认方法会简单地返回 `True`，表示在默认情况下，所有通道都希望能读取事件。

writable()
每次在异步循环之外被调用以确定是否应当将一个通道的套接字添加到可能在其上发生写入事件的列表中。默认方法会简单地返回 `True`，表示在默认情况下，所有通道都希望能写入事件。

此外，每个通道都委托或扩展了许多套接字方法。它们大部分都与其套接字的对应方法几乎一样。

create_socket(*family=socket.AF_INET*, *type=socket.SOCK_STREAM*)
这与普通套接字的创建相同，并会使用同样的创建选项。请参阅*socket* 文档了解有关创建套接字的信息。

在 3.3 版更改: *family* 和 *type* 参数可以被省略。

connect(*address*)
与普通套接字对象一样，*address* 是一个元组，它的第一个元素是要连接的主机，第二个元素是端口号。

send(*data*)
将 *data* 发送到套接字的远程端点。

recv(*buffer_size*)
从套接字的远程端点读取至多 *buffer_size* 个字节。读到空字节串表明通道已从另一端被关闭。

请注意*recv()* 可能会引发*BlockingIOError*，即使*select.select()* 或*select.poll()* 报告套接字已准备好被读取。

listen(*backlog*)
侦听与套接字的连接。*backlog* 参数指明排入连接队列的最大数量且至少应为 1；最大值取决于具体系统（通常为 5）。

bind(*address*)
将套接字绑定到 *address*。套接字必须尚未被绑定。(*address* 的格式取决于具体的地址族—请参阅*socket* 文档了解更多信息。) 要将套接字标记为可重用的 (设置 `SO_REUSEADDR` 选项)，请调用*dispatcher* 对象的 `set_reuse_addr()` 方法。

accept()
接受一个连接。此套接字必须绑定到一个地址上并且侦听连接。返回值可以是 `None` 或一个 `(conn, address)` 对，其中 *conn* 是一个可用来在此连接上发送和接收数据的 *新的*套接字对象，而 *address* 是绑定到连接另一端套接字的地址。当返回 `None` 时意味着连接没有建立，在此情况下服务器应当忽略此事件并继续侦听后续的入站连接。

close()
关闭套接字。在此套接字对象上的后续操作都将失败。远程端点将不再接收任何数据（在排入队列的数据被清空之后）。当套接字被垃圾回收时会自动关闭。

class asyncore.**dispatcher_with_send**
dispatcher 的一个添加了简单缓冲输出功能的子类，适用于简单客户端。对于更复杂的用法请使用*asynchat.async_chat*。

class asyncore.**file_dispatcher**
A file_dispatcher takes a file descriptor or *file object* along with an optional map argument and wraps it for use with the `poll()` or `loop()` functions. If provided a file object or anything with a `fileno()` method, that method will be called and passed to the *file_wrapper* constructor. Availability: UNIX.

class asyncore.**file_wrapper**
A file_wrapper takes an integer file descriptor and calls *os.dup()* to duplicate the handle so that the original handle may be closed independently of the file_wrapper. This class implements sufficient methods to emulate a socket for use by the *file_dispatcher* class. Availability: UNIX.

18.6.1 asyncore 示例基本 HTTP 客户端

下面是一个非常基本的 HTTP 客户端，它使用了*dispatcher* 类来实现套接字处理:

```
import asyncore

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect( (host, 80) )
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' %
                            (path, host), 'ascii')

    def handle_connect(self):
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print(self.recv(8192))

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]


client = HTTPClient('www.python.org', '/')
asyncore.loop()
```

18.6.2 asyncore 示例基本回显服务器

下面是一个基本的回显服务器，它使用了*dispatcher* 类来接受连接并将入站连接发送给处理程序:

```
import asyncore

class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accepted(self, sock, addr):
        print('Incoming connection from %s' % repr(addr))
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```

18.7 asynchat —异步 socket 指令/响应处理器

源代码: Lib/asynchat.py

3.6 版后已移除: 请使用*asyncio* 替代。

注解: 该模块仅为提供向后兼容。我们推荐在新代码中使用*asyncio* 。

此模块在*asyncore* 框架之上构建，简化了异步客户端和服务器并使得处理元素为以任意字符串结束或者为可变长度的协议更加容易。*asynchat* 定义了一个可以由你来子类化的抽象类*async_chat*，提供了 `collect_incoming_data()` 和 `found_terminator()` 等方法的实现。它使用与*asyncore* 相同的异步循环，并且可以在通道映射中自由地混合*asyncore.dispatcher* 和*asynchat.async_chat* 这两种类型的通道。一般来说*asyncore.dispatcher* 服务器通道在接收到传入的连接请求时会生成新的*asynchat.async_chat* 通道对象。

class asynchat.**async_chat**

这个类是*asyncore.dispatcher* 的抽象子类。对于实际使用的代码你必须子类化*async_chat*，提供有意义的*collect_incoming_data()* 和*found_terminator()* 方法。*asyncore.dispatcher* 的方法也可以被使用，但它们在消息/响应上下文中并不是全都有意义。

与*asyncore.dispatcher* 类似，*async_chat* 也定义了一组通过对 `select()` 调用之后的套接字条件进行分析所生成的事件。一旦启动轮询循环*async_chat* 对象的方法就会被事件处理框架调用而无须程序员方面做任何操作。

两个可被修改的类属性，用以提升性能，甚至也可能会节省内存。

ac_in_buffer_size
异步输入缓冲区大小 (默认为 4096)。

ac_out_buffer_size
异步输出缓冲区大小 (默认为 4096)。

与*asyncore.dispatcher* 不同，*async_chat* 允许你定义一个 FIFO 队列 *producers*。其中的生产者只需要一个方法 more()，该方法应当返回要在通道上传输的数据。生产者通过让其 more() 方法返回空字节串对象来表明其处于耗尽状态 (意即它已不再包含数据)。此时*async_chat* 对象会将该生产者从队列中移除并开始使用下一个生产者，如果有下一个的话。当生产者队列为空时 handle_write() 方法将不执行任何操作。你要使用通道对象的*set_terminator()* 方法来描述如何识别来自远程端点的入站传输的结束或是重要的中断点。

要构建一个可用的*async_chat* 子类，你的输入方法*collect_incoming_data()* 和*found_terminator()* 必须要处理通道异步接收的数据。这些参数的描述见下文。

async_chat.**close_when_done**()
将 None 推入生产者队列。当此生产者被弹出队列时它将导致通道被关闭。

async_chat.**collect_incoming_data**(*data*)
调用时附带 *data*，其中包含任意数量的已接收数据。必须被重载的默认方法将引发一个*NotImplementedError* 异常。

async_chat.**discard_buffers**()
在紧急情况下此方法将丢弃输入和/或输出缓冲区以及生产者队列中的任何数据。

async_chat.**found_terminator**()
当输入数据流能匹配*set_terminator()* 所设定的终结条件时会被调用。必须被重载的默认方法将引发一个*NotImplementedError* 异常。被缓冲的输入数据应当可以通过实例属性来获取。

async_chat.**get_terminator**()
返回通道的当前终结器。

async_chat.**push**(*data*)
将数据推入通道的队列以确保其被传输。要让通道将数据写到网络中你只需要这样做就足够了，虽然以更复杂的方式使用你自己的生产者也是有可能的，例如为了实现加密和分块。

async_chat.**push_with_producer**(*producer*)
获取一个生产者对象并将其加入到与通道相关联的生产者队列中。当所有当前已推入的生产者都已被耗尽时通道将通过调用其 more() 方法来耗用此生产者的数据并将数据发送至远程端点。

async_chat.**set_terminator**(*term*)
设置可在通道上被识别的终结条件。term 可以是三种类型值中的任意一种，对应于处理入站协议数据的三种不同方式。

term	描述
string	当在输入流中发现该字符串时将会调用*found_terminator()*
integer	当接收到指定数量的字符时将会调用*found_terminator()*
None	通道会不断地持续收集数据

请注意终结器之后的任何数据将可在*found_terminator()* 被调用后由通道来读取。

18.7.1 asynchat 示例

下面的例子片段显示了如何通过*async_chat* 来读取 HTTP 请求。Web 服务器可以为每个入站的客户端连接创建 `http_request_handler` 对象。请注意在初始时通道终结器会被设置为匹配 HTTP 请求头末尾的空行，并且会用一个旗标来指明请求头正在被读取。

一旦完成了标头的读取，如果请求类型为 POST (表明输入流中存在更多的数据) 则会使用 `Content-Length:` 标头来设置一个数值终结器以从通道读取适当数量的数据。

一旦完成了对所有相关输入的处理，将会在设置通道终结器为 `None` 以确保忽略掉 Web 客户端所发送的任何无关数据之后调用 `handle_request()` 方法。

```
import asynchat

class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = b""
        self.set_terminator(b"\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data"""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers(b"".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == b"POST":
                clen = self.headers.getheader("content-length")
                self.set_terminator(int(clen))
            else:
                self.handling = True
                self.set_terminator(None)
                self.handle_request()
        elif not self.handling:
            self.set_terminator(None)  # browsers sometimes over-send
            self.cgi_data = parse(self.headers, b"".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle_request()
```

18.8 signal —设置异步事件处理程序

该模块提供了在 Python 中使用信号处理程序的机制。

18.8.1 一般规则

The *signal.signal()* function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed: SIGPIPE is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and SIGINT is translated into a *KeyboardInterrupt* exception.

一旦设置，特定信号的处理程序将保持安装，直到它被显式重置（Python 模拟 BSD 样式接口而不管底层实现），但 SIGCHLD 的处理程序除外，它遵循底层实现。

执行 Python 信号处理程序

Python 信号处理程序不会在低级（C ）信号处理程序中执行。相反，低级信号处理程序设置一个标志，告诉*virtual machine* 稍后执行相应的 Python 信号处理程序（例如在下一个*bytecode* 指令）。这会导致：

- 捕获同步错误是没有意义的，例如 SIGFPE 或 SIGSEGV ，它们是由 C 代码中的无效操作引起的。Python 将从信号处理程序返回到 C 代码，这可能会再次引发相同的信号，导致 Python 显然的挂起。从 Python 3.3 开始，你可以使用*faulthandler* 模块来报告同步错误。
- 纯 C 中实现的长时间运行的计算（例如在大量文本上的正则表达式匹配）可以在任意时间内不间断地运行，而不管接收到任何信号。计算完成后将调用 Python 信号处理程序。

信号与线程

Python 信号处理程序总是在主 Python 线程中执行，即使信号是在另一个线程中接收的。这意味着信号不能用作线程间通信的手段。你可以使用*threading* 模块中的同步原函数。

此外，只允许主线程设置新的信号处理程序。

18.8.2 模块内容

在 3.5 版更改: 信号（SIG* ），处理程序（*SIG_DFL* ，*SIG_IGN*）和 sigmask（*SIG_BLOCK* ，*SIG_UNBLOCK* ，*SIG_SETMASK* ）下面列出的相关常量变成了*enums* 。*getsignal()* ，*pthread_sigmask()* ，*sigpending()* 和*sigwait()* 函数返回人类可读的*enums* 。

在*signal* 模块中定义的变量是：

signal.**SIG_DFL**
: 这是两种标准信号处理选项之一；它只会执行信号的默认函数。例如，在大多数系统上，对于 SIGQUIT 的默认操作是转储核心并退出，而对于 SIGCHLD 的默认操作是简单地忽略它。

signal.**SIG_IGN**
: 这是另一个标准信号处理程序，它将简单地忽略给定的信号。

SIG*
: All the signal numbers are defined symbolically. For example, the hangup signal is defined as signal.SIGHUP; the variable names are identical to the names used in C programs, as found in <signal.h>. The Unix man page for ‘signal()’ lists the existing signals (on some systems this is *signal(2)*, on others the list is in

signal(7)). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

signal.**CTRL_C_EVENT**
对应于 Ctrl+C 击键事件的信号。此信号只能用于*os.kill()* 。

Availability: Windows.

3.2 新版功能.

signal.**CTRL_BREAK_EVENT**
对应于 Ctrl+Break 击键事件的信号。此信号只能用于*os.kill()* 。

Availability: Windows.

3.2 新版功能.

signal.**NSIG**
比最高信号数多一。

signal.**ITIMER_REAL**
实时递减间隔计时器，并在到期时发送 SIGALRM 。

signal.**ITIMER_VIRTUAL**
仅在进程执行时递减间隔计时器，并在到期时发送 SIGVTALRM 。

signal.**ITIMER_PROF**
当进程执行时以及当系统替进程执行时都会减小间隔计时器。这个计时器与 ITIMER_VIRTUAL 相配结，通常被用于分析应用程序在用户和内核空间中花费的时间。SIGPROF 会在超期时被发送。

signal.**SIG_BLOCK**
pthread_sigmask() 的 *how* 形参的一个可能的值，表明信号将会被阻塞。

3.3 新版功能.

signal.**SIG_UNBLOCK**
pthread_sigmask() 的 *how* 形参的是个可能的值，表明信号将被解除阻塞。

3.3 新版功能.

signal.**SIG_SETMASK**
pthread_sigmask() 的 *how* 形参的一个可能的值，表明信号掩码将要被替换。

3.3 新版功能.

signal 模块定义了一个异常:

exception signal.**ItimerError**
作为来自下层*setitimer()* 或*getitimer()* 实现错误的信号被引发。如果将无效的定时器或负的时间值传给*setitimer()* 就导致这个错误。此错误是*OSError* 的子类型。

3.3 新版功能: 此错误是*IOError* 的子类型，现在则是*OSError* 的别名。

signal 模块定义了以下函数:

signal.**alarm**(*time*)
If *time* is non-zero, this function requests that a SIGALRM signal be sent to the process in *time* seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If *time* is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled. (See the Unix man page *alarm(2)*.) Availability: Unix.

signal.**getsignal**(*signalnum*)
返回当前用于信号 *signalnum* 的信号处理程序。返回值可以是一个 Python 可调用对象，或是特殊值*signal.SIG_IGN*, *signal.SIG_DFL* 或*None* 之一。在这里，*signal.SIG_IGN* 表示信号在之

前被忽略，*signal.SIG_DFL* 表示之前在使用默认的信号处理方式，而 None 表示之前的信号处理程序未由 Python 安装。

signal.**pause**()
Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing. Not on Windows. (See the Unix man page *signal(2)*.)

另请参阅*sigwait()*, *sigwaitinfo()*, *sigtimedwait()* 和*sigpending()*。

signal.**pthread_kill**(*thread_id*, *signalnum*)
Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be *executed by the main thread*. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with *InterruptedError*.

使用*threading.get_ident()* 或*threading.Thread* 对象的*ident* 属性为 *thread_id* 获取合适的值。

如果 *signalnum* 为 0，则不会发送信号，但仍然会执行错误检测；这可被用来检测目标线程是否仍在运行。

Availability: Unix (see the man page *pthread_kill(3)* for further information).

另请参阅*os.kill()*。

3.3 新版功能.

signal.**pthread_sigmask**(*how*, *mask*)
获取和/或修改调用方线程的信号掩码。信号掩码是一组传送过程目前为调用者而阻塞的信号集。返回旧的信号掩码作为一组信号。

该调用的行为取决于 *how* 的值，具体见下。

- *SIG_BLOCK*: 被阻塞的信号集是当前集与 *mask* 参数的并集。
- *SIG_UNBLOCK*: *mask* 中的信号会从当前已阻塞信号集中被移除。允许尝试取消对一个非阻塞信号的阻塞。
- *SIG_SETMASK*: 已阻塞信号集会被设为 *mask* 参数的值。

mask is a set of signal numbers (e.g. {signal.SIGINT, signal.SIGTERM}). Use range(1, signal.NSIG) for a full mask including all signals.

例如，signal.pthread_sigmask(signal.SIG_BLOCK, []) 会读取调用方线程的信号掩码。

Availability: Unix. See the man page *sigprocmask(3)* and *pthread_sigmask(3)* for further information.

另请参阅*pause()*, *sigpending()* 和*sigwait()*。

3.3 新版功能.

signal.**setitimer**(*which*, *seconds*[, *interval*])
Sets given interval timer (one of *signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL* or *signal.ITIMER_PROF*) specified by *which* to fire after *seconds* (float is accepted, different from *alarm()*) and after that every *interval* seconds. The interval timer specified by *which* can be cleared by setting seconds to zero.

当一个间隔计时器启动时，会有信号发送至进程。所发送的具体信号取决于所使用的计时器；*signal.ITIMER_REAL* 将发送 SIGALRM, *signal.ITIMER_VIRTUAL* 将发送 SIGVTALRM, 而*signal.ITIMER_PROF* 将发送 SIGPROF.

原有的值会以元组: (delay, interval) 的形式被返回。

Attempting to pass an invalid interval timer will cause an *ItimerError*. Availability: Unix.

signal.**getitimer**(*which*)
Returns current value of a given interval timer specified by *which*. Availability: Unix.

signal.**set_wakeup_fd**(*fd*)
将唤醒文件描述符设为 *fd*。当接收到信号时，会将信号编号以单个字节的形式写入 fd。这可被其它库用来唤醒一次 poll 或 select 调用，以允许该信号被完全地处理。

原有的唤醒 fd 会被返回（或者如果未启用文件描述符唤醒则返回 -1）。如果 *fd* 为 -1，文件描述符唤醒会被禁用。如果不为 -1，则 *fd* 必须为非阻塞型。需要由库来负责在重新调用 poll 或 select 之前从 *fd* 移除任何字节数据。

Use for example `struct.unpack('%uB' % len(data), data)` to decode the signal numbers list.

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a *ValueError* exception to be raised.

在 3.5 版更改: 在 Windows 上，此函数现在也支持套接字处理。

signal.**siginterrupt**(*signalnum*, *flag*)
Change system call restart behaviour: if *flag* is *False*, system calls will be restarted when interrupted by signal *signalnum*, otherwise system calls will be interrupted. Returns nothing. Availability: Unix (see the man page *siginterrupt(3)* for further information).

请注意用*signal()* 安装信号处理程序将重启行为重置为可通过显式调用 `siginterrupt()` 并为给定信号的 *flag* 设置真值来实现中断。

signal.**signal**(*signalnum*, *handler*)
Set the handler for signal *signalnum* to the function *handler*. *handler* can be a callable Python object taking two arguments (see below), or one of the special values *signal.SIG_IGN* or *signal.SIG_DFL*. The previous signal handler will be returned (see the description of *getsignal()* above). (See the Unix man page *signal(2)*.)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a *ValueError* exception to be raised.

handler 将附带两个参数调用：信号编号和当前堆栈帧 (`None` 或一个帧对象；有关帧对象的描述请参阅类型层级结构描述或者参阅*inspect* 模块中的属性描述)。

在 Windows 上，*signal()* 调用只能附带 `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, `SIGTERM` 或 `SIGBREAK`。任何其他值都将引发*ValueError*。请注意不是所有系统都定义了同样的信号名称集合；如果一个信号名称未被定义为 `SIG*` 模块层级常量则将引发*AttributeError*。

signal.**sigpending**()
检查正在等待传送给调用方线程的信号集合（即在阻塞期间被引发的信号）。返回正在等待的信号集合。

Availability: Unix (see the man page *sigpending(2)* for further information).

另请参阅*pause()*, *pthread_sigmask()* 和*sigwait()*。

3.3 新版功能.

signal.**sigwait**(*sigset*)
挂起调用方线程的执行直到信号集合 *sigset* 中指定的信号之一被传送。此函数会接受该信号（将其从等待信号列表中移除），并返回信号编号。

Availability: Unix (see the man page *sigwait(3)* for further information).

另请参阅*pause()*, *pthread_sigmask()*, *sigpending()*, *sigwaitinfo()* 和*sigtimedwait()*。

3.3 新版功能.

signal.**sigwaitinfo**(*sigset*)
挂起调用方线程的执行直到信号集合 *sigset* 中指定的信号之一被传送。此函数会接受该信号并将其从等

待信号列表中移除。如果 *sigset* 中的信号之一已经在等待调用方线程，此函数将立即返回并附带有关该信号的信息。被传送信号的信号处理程序不会被调用。如果该函数被某个不在 *sigset* 中的信号中断则会引发*InterruptedError*。

返回值是一个代表 siginfo_t 结构体所包含数据的对象，具体为: si_signo, si_code, si_errno, si_pid, si_uid, si_status, si_band。

Availability: Unix (see the man page *sigwaitinfo(2)* for further information).

另请参阅*pause()*, *sigwait()* 和*sigtimedwait()*。

3.3 新版功能.

在 3.5 版更改: 当被某个不在 *sigset* 中的信号中断时本函数将结束并且信号处理程序不会引发异常 (其理由参见 **PEP 475**)。

signal.**sigtimedwait**(*sigset*, *timeout*)

类似于*sigwaitinfo()*，但会接受一个额外的 *timeout* 参数来指定超时限制。如果将 *timeout* 指定为 0，则会执行轮询。如果发生超时则返回*None*。

Availability: Unix (see the man page *sigtimedwait(2)* for further information).

另请参阅*pause()*, *sigwait()* 和*sigwaitinfo()*。

3.3 新版功能.

在 3.5 版更改: 现在当此函数被某个不在 *sigset* 中的信号中断时将以计算出的 *timeout* 进行重试并且信号处理程序不会引发异常（请参阅 **PEP 475** 了解其理由）。

18.8.3 示例

这是一个最小示例程序。它使用*alarm()* 函数来限制等待打开一个文件所花费的时间；这在文件为无法开启的串行设备时会很有用处，此情况通常会导致*os.open()* 无限期地挂起。解决办法是在打开文件之前设置 5 秒钟的 alarm；如果操作耗时过长，将会发送 alarm 信号，并且处理程序会引发一个异常。

```
import signal, os

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

18.9 mmap —内存映射文件支持

内存映射（mmap）文件对象的行为既像*bytearray* 又像文件对象。你可以在大部分接受*bytearray* 的地方使用 mmap 对象；例如，你可以使用*re* 模块来搜索一个内存映射文件。你也可以通过执行 `obj[index] = 97` 来修改单个字节，或者通过对切片赋值来修改一个子序列: `obj[i1:i2] = b'...'`。你还可以在文件的当前位置开始读取和写入数据，并使用 `seek()` 前往另一个位置。

内存映射文件是由*mmap* 构造函数创建的，其在 Unix 和 Windows 上是不同的。无论哪种情况，你都必须为一个打开的文件提供文件描述符以进行更新。如果你希望映射一个已有的 Python 文件对象，请使用该对象的 `fileno()` 方法来获取 *fileno* 参数的正确值。否则，你可以使用*os.open()* 函数来打开这个文件，这会直接返回一个文件描述符（结束时仍然需要关闭该文件）。

注解: 如果要为可写的缓冲文件创建内存映射，则应当首先*flush()* 该文件。这确保了对缓冲区的本地修改在内存映射中可用。

For both the Unix and Windows versions of the constructor, *access* may be specified as an optional keyword parameter. *access* accepts one of three values: `ACCESS_READ`, `ACCESS_WRITE`, or `ACCESS_COPY` to specify read-only, write-through or copy-on-write memory respectively. *access* can be used on both Unix and Windows. If *access* is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an `ACCESS_READ` memory map raises a *TypeError* exception. Assignment to an `ACCESS_WRITE` memory map affects both memory and the underlying file. Assignment to an `ACCESS_COPY` memory map affects memory but does not update the underlying file.

要映射匿名内存，应将 -1 作为 fileno 和 length 一起传递。

class mmap.**mmap**(*fileno*, *length*, *tagname=None*, *access=ACCESS_DEFAULT*[, *offset*])

(Windows 版本) 映射被文件句柄 *fileno* 指定的文件的 *length* 个字节，并创建一个 mmap 对象。如果 *length* 大于当前文件大小，则文件将扩展为包含 *length* 个字节。如果 *length* 为 `0`，则映射的最大长度为当前文件大小。如果文件为空，Windows 会引发异常（你无法在 Windows 上创建空映射）。

如果 *tagname* 被指定且不是 `None`，则是为映射提供标签名称的字符串。Windows 允许你对同一文件拥有许多不同的映射。如果指定现有标签的名称，则会打开该标签，否则将创建该名称的新标签。如果省略此参数或设置为 `None`，则创建的映射不带名称。避免使用 tag 参数将有助于使代码在 Unix 和 Windows 之间可移植。

offset 可以被指定为非负整数偏移量。mmap 引用将相对于从文件开头的偏移。*offset* 默认为 0。*offeset* 必须是 `ALLOCATIONGRANULARITY` 的倍数。

class mmap.**mmap**(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE|PROT_READ*, *access=ACCESS_DEFAULT*[, *offset*])

(Unix 版本) 映射文件描述符 *fileno* 指定的文件的 *length* 个字节，并返回一个 mmap 对象。如果 *length* 为 `0`，则当调用*mmap* 时，映射的最大长度将为文件的当前大小。

flags 指明映射的性质。`MAP_PRIVATE` 会创建私有的写入时拷贝映射，因此对 mmap 对象内容的修改将为该进程所私有，而 `MAP_SHARED` 会创建与其他映射同一文件区域的进程所共享的映射。默认值为 `MAP_SHARED`。

如果指明了 *prot*，它将给出所需的内存保护方式；最有用的两个值是 `PROT_READ` 和 `PROT_WRITE`，分别指明页面为可读或可写。*prot* 默认为 `PROT_READ | PROT_WRITE`。

可以指定 *access* 作为替代 *flags* 和 *prot* 的可选关键字形参。同时指定 *flags*, *prot* 和 *access* 将导致错误。请参阅上文中 *access* 的描述了解有关如何使用此形参的信息。

offset 可以被指定为非负整数偏移量。mmap 引用将相对于从文件开头的偏移。*offset* 默认为 0。*offset* 必须是 `ALLOCATIONGRANULARITY` 的倍数，它在 Unix 系统上等价于 `PAGESIZE`。

要确保已创建内存映射的有效性，描述符 *fileno* 所指定的文件在 Mac OS X 和 OpenVMS 上会与物理后备存储进行内部自动同步。

这个例子演示了使用*mmap* 的简单方式:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    mm.close()
```

mmap 也可以在 `with` 语句中被用作上下文管理器:

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

3.2 新版功能: 上下文管理器支持。

下面的例子演示了如何创建一个匿名映射并在父进程和子进程之间交换数据。:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

映射内存的文件对象支持以下方法:

close()
关闭 mmap。后续调用该对象的其他方法将导致引发 ValueError 异常。此方法将不会关闭打开的文件。

closed
如果文件已关闭则返回 `True`。

3.2 新版功能.

find(*sub*[, *start*[, *end*]])
返回子序列 *sub* 在对象内被找到的最小索引号，使得 *sub* 被包含在 [*start*, *end*] 范围中。可选参数 *start* 和 *end* 会被解读为切片表示法。如果未找到则返回 `-1`。

在 3.5 版更改: 现在支持可写的字节类对象。

flush([*offset*[, *size*]])
将对文件的内存副本的修改刷新至磁盘。如果不使用此调用则无法保证在对象被销毁前将修改写回存储。如果指定了 *offset* 和 *size*，则只将对指定范围内字节的修改刷新至磁盘；在其他情况下，映射的全部范围都会被刷新。*offset* 必须为 `PAGESIZE` 或 `ALLOCATIONGRANULARITY` 的倍数。

(Windows version) A nonzero value returned indicates success; zero indicates failure.

(Unix 版本) 返回零值以表示成功。当调用失败时将引发异常。

move(*dest*, *src*, *count*)
将从偏移量 *src* 开始的 *count* 个字节拷贝到目标索引号 *dest*。如果 mmap 创建时设置了 `ACCESS_READ`，则调用 move 将引发*TypeError* 异常。

read([*n*])
返回一个*bytes*，其中包含从当前文件位置开始的至多 *n* 个字节。如果参数省略，为 `None` 或负数，则返回从当前文件位置开始直至映射结尾的所有字节。文件位置会被更新为返回字节数据之后的位置。

在 3.3 版更改: 参数可被省略或为 `None`。

read_byte()
将当前文件位置上的一个字节以整数形式返回，并让文件位置前进 1。

readline()
Returns a single line, starting at the current file position and up to the next newline.

resize(*newsize*)
改变映射以及下层文件的大小，如果存在的话。如果 mmap 创建时设置了 `ACCESS_READ` 或 `ACCESS_COPY`，则改变映射大小将引发*TypeError* 异常。

rfind(*sub*[, *start*[, *end*]])
返回子序列 *sub* 在对象内被找到的最大索引号，使得 *sub* 被包含在 [*start*, *end*] 范围中。可选参数 *start* 和 *end* 会被解读为切片表示法。如果未找到则返回 `-1`。

在 3.5 版更改: 现在支持可写的字节类对象。

seek(*pos*[, *whence*])
设置文件的当前位置。*whence* 参数为可选项并且默认为 `os.SEEK_SET` 或 `0` (绝对文件定位)；其他值还有 `os.SEEK_CUR` 或 `1` (相对当前位置查找) 和 `os.SEEK_END` 或 `2` (相对文件末尾查找)。

size()
返回文件的长度，该数值可以大于内存映射区域的大小。

tell()
返回文件指针的当前位置。

write(*bytes*)
将 *bytes* 中的字节数据写入文件指针当前位置的内存并返回写入的字节总数 (一定不小于 `len(bytes)`，因为如果写入失败，将会引发*ValueError*)。在字节数据被写入后文件位置将会更新。如果 mmap 创建时设置了 `ACCESS_READ`，则向其写入将引发*TypeError* 异常。

在 3.5 版更改: 现在支持可写的字节类对象。

在 3.6 版更改: 现在会返回写入的字节总数。

write_byte(*byte*)
将整数值 *byte* 写入文件指针当前位置的内存；文件位置前进 1。如果 mmap 创建时设置了 ACCESS_READ，则向其写入将引发*TypeError*异常。

CHAPTER 19

互联网数据处理

本章介绍了支持处理互联网上常用数据格式的模块。

19.1 email —电子邮件与 MIME 处理包

源码: Lib/email/__init__.py

email 包是一个用于管理电子邮件消息的库。它 并非被设计为执行向 SMTP (**RFC 2821**), NNTP 或其他服务器发送电子邮件消息的操作；这些是*smtplib* 和*nntplib* 等模块的功能。*email* 包试图尽可能遵循 RFC，支持 **RFC 5233** 和 **RFC 6532**，以及与 MIME 相关的各个 RFC 例如 **RFC 2045**, **RFC 2046**, **RFC 2047**, **RFC 2183** 和 **RFC 2231**。

email 包的总体结构可以分为三个主要部分，另外还有第四个部分用于控制其他部分的行为。

这个包的中心组件是代表电子邮件消息的“对象模型”。应用程序主要通过在*message* 子模块中定义的对象模型接口与这个包进行交互。应用程序可以使用此 API 来询问有关现有电子邮件的问题、构造新的电子邮件，或者添加或移除自身也使用相同对象模型接口的电子邮件子组件。也就是说，遵循电子邮件消息及其 MIME 子组件的性质，电子邮件对象模型是所有提供*EmailMessage* API 的对象所构成的树状结构。

这个包的另外两个主要组件是*parser* 和*generator*。parser 接受电子邮件消息的序列化版本（字节流）并将其转换为*EmailMessage* 对象树。generator 接受*EmailMessage* 并将其转回序列化的字节流。(parser 和 generator 还能处理文本字符流，但不建议这种用法，因为这很容易导致某种形式的无效消息。

控制组件是*policy* 模块。每一个*EmailMessage*、每一个*generator* 和每一个*parser* 都有一个相关联的*policy* 对象来控制其行为。通常应用程序只有在*EmailMessage* 被创建时才需要指明控制策略，或者通过直接实例代*EmailMessage* 来新建电子邮件，或者通过使用*parser* 来解析输入流。但是策略也可以在使用*generator* 序列化消息时被更改。例如，这允许从磁盘解析通用电子邮件消息，而在将消息发送到电子邮件服务器时使用标准 SMTP 设置对其进行序列化。

email 包会尽量地对应用程序隐藏各种控制类 RFC 的细节。从概念上讲应用程序应当能够将电子邮件消息视为 Unicode 文本和二进制附件的结构化树，而不必担心在序列化时要如何表示它们。但在实际中，经常有必要至少了解一部分控制类 MIME 消息及其结构的规划，特别是 MIME “内容类型”的名称和性质以及它们是如何标识多部分文档的。在大多数情况下这些知识应当仅对于更复杂的应用程序来说才是必需的，并且即

便在那时它也应当仅是特定的高层级结构，而不是如何表示这些结构的细节信息。由于 MIME 内容类型被广泛应用于现代因特网软件（而非只是电子邮件），因此这对许多程序员来说将是很熟悉的概念。

以下小节描述了*email* 包的具体功能。我们会从*message* 对象模型开始，它是应用程序将要使用的主要接口，之后是*parser* 和*generator* 组件。然后我们会介绍*policy* 控制组件，它将完成对这个库的主要组件的处理。

接下来的三个小节会介绍这个包可能引发的异常以及*parser* 可能检测到的缺陷（即与 RFC 不相符）。然后我们会介绍*headerregistry* 和*contentmanager* 子组件，它们分别提供了用于更精细地操纵标题和载荷的工具。这两个组件除了包含使用与生成非简单消息的相关特性，还记录了它们的可扩展性 API，这将是高级应用程序所感兴趣的内容。

在此之后是一组使用之前小节所介绍的 API 的基本部分的示例。

前面的内容是 email 包的现代（对 Unicode 支持良好）API。从*Message* 类开始的其余小节则介绍了旧式*compat32* API，它会更直接地处理如何表示电子邮件消息的细节。*compat32* API 不会向应用程序隐藏 RFC 的相关细节，但对于需要进行此种层级操作的应用程序来说将是很有用的工具。此文档对于因向下兼容理由而仍然使用*compat32* API 的应用程序也是很适合的。

在 3.6 版更改: 文档经过重新组织和撰写以鼓励使用新的*EmailMessage*/*EmailPolicy* API。

email 包文档的内容:

19.1.1 email.message: 表示一封电子邮件信息

源代码: Lib/email/message.py

3.6 新版功能:[1]

位于*email* 包的中心的类就是*EmailMessage* 类。这个类导入自*email.message* 模块。它是*email* 对象模型的基类。*EmailMessage* 为设置和查询头字段内容、访问信息体的内容、以及创建和修改结构化信息提供了核心功能。

一份电子邮件信息由 * 头 * 和 * 负载 *（又被称为 * 内容 *）组成。头遵循 **RFC 5322** 或者 **RFC 6532** 风格的字段名和值，字段名和字段值之间由一个冒号隔开。这个冒号既不属于字段名，也不属于字段值。信息的负载可能是一段简单的文字消息，也可能是一个二进制的对象，更可能是由多个拥有各自头和负载的子信息组成的结构化子信息序列。对于后者类型的负载，信息的 MIME 类型将会被指明为诸如 *multipart/** 或 *message/rfc822* 的类型。

EmailMessage 对象所提供的抽象概念模型是一个头字段组成的有序字典加一个代表 **RFC 5322** 标准的信息体的 * 负载 *。负载有可能是一系列子 “EmailMessage“对象的列表。你除了可以通过一般的字典方法来访问头字段名和值，还可以使用特制方法来访问头的特定字段（比如说 MIME 内容类型字段）、操纵负载、生成信息的序列化版本、递归遍历对象树。

EmailMessage 的类字典接口的字典索引是头字段名，头字段名必须是 ASCII 值。字典值是带有一些附加方法的字符串。虽然头字段的存储和获取都是保留其原始大小写的，但是字段名的匹配是大小写不敏感的。与真正的字典不同，键与键之间不但存在顺序关系，还可以重复。我们提供了额外的方法来处理含有重复键的头。

payload 是多样的。对于简单的消息对象，它是字符串或字节串对象；对于诸如 *multipart/** 和 *message/rfc822* 消息对象的 MIME 容器文档，它是一个*EmailMessage* 对象列表。

class email.message.**EmailMessage**(*policy=default*)

如果指定了 *policy*，消息将由这个 *policy* 所指定的规则来更新和序列化信息的表达。如果没有指定 *policy*，其将默认使用*default* 策略。这个策略遵循电子邮件的 RFC 标准，除了行终止符号（RFC 要求使用 “rn“，此策略使用 Python 标准的 “n“行终止符）。请前往*policy* 的文档获取更多信息。

[1] 原先在 3.4 版本中以*provisional module* 添加。过时的文档被移动至*email.message.Message: 使用 compat32 API 来表示电子邮件消息* 。

as_string(*unixfrom=False*, *maxheaderlen=None*, *policy=None*)
以一段字符串的形式返回整个消息对象。若可选的 *unixform* 参数为真，返回的字符串会包含信封头。*unixform* 的默认值是 `False`。为了保持与基类*Message* 的兼容性，*maxheaderlen* 是被接受的，但是其默认值是 `None`。这个默认值表示行长度由策略的 `max_line_length` 属性所控制。从信息实例所获取到的策略可以通过 *policy* 参数重写。这样可以对该方法所产生的输出进行略微的控制，因为指定的 *policy* 会被传递到*Generator* 当中。

扁平化信息可能会对*EmailMessage* 做出修改。这是因为为了完成向字符串的转换，一些内容需要使用默认值填入（举个例子，MIME 边界字段可能会被生成或被修改）。

请注意，这个方法是为了便利而提供，不一定是适合你的应用程序的最理想的序列化信息的方法。这在你处理多封信息的时候尤甚。如果你需要使用更加灵活的 API 来序列化信息，请参见*email.generator.Generator* 。同时请注意，当*utf8* 属性为 “False“的时候（这是默认值），本方法将限制其行为为生成以 “7 bit clean” 方式序列化的信息。

在 3.6 版更改: *maxheaderlen* 没有被指定时的默认行为从默认为 *0* 修改为默认为策略的 *max_line_length*。

__str__()
与 “as_string(policy=self.policy.clone(utf8=True)“等价。这将让 “str(msg)“产生的字符串包含人类可读的的序列化信息内容。

在 3.4 版更改: 本方法开始使用 “utf8=True“，而非*as_string()* 的直接替身。使用 “utf8=True“会产生类似于 **RFC 6531** 的信息表达。

as_bytes(*unixfrom=False*, *policy=None*)
以字节串对象的形式返回整个扁平化后的消息。当可选的 *unixfrom* 为真值时，返回的字符串会包含信封标头。*unixfrom* 的默认值为 `False`。*policy* 参数可被用于重载从消息实例获取的默认 policy。这可被用来控制该方法所产生的部分格式效果，因为指定的 *policy* 将被传递给*BytesGenerator*。

扁平化信息可能会对*EmailMessage* 做出修改。这是因为为了完成向字符串的转换，一些内容需要使用默认值填入（举个例子，MIME 边界字段可能会被生成或被修改）。

请注意，这个方法是为了便利而提供，不一定是适合你的应用程序的最理想的序列化信息的方法。这在你处理多封信息的时候尤甚。如果你需要使用更加灵活的 API 来序列化信息，请参见*email.generator.BytesGenerator* 。

__bytes__()
与*as_bytes()* 等价。这将让 “bytes(msg)“产生一个包含序列化信息内容的字节序列对象。

is_multipart()
如果该信息的负载是一个子*EmailMessage* 对象列表，返回 `True` ；否则返回 `False` 。在*is_multipart()* 返回 `True` 的场合下，负载应当是一个字符串对象（有可能是一个使用了内容传输编码进行编码的二进制负载）。请注意，*is_multipart()* 返回 `True` 不意味着 `msg.get_content_maintype() == 'multipart'` 也会返回 `True` 。举个例子，`is_multipart` 在*EmailMessage* 是 `message/rfc822` 类型的信息的情况下，其返回值也是 `True` 。

set_unixfrom(*unixfrom*)
将信息的信封头设置为 *unixform* ，这应当是一个字符串。（在*mboxMessage* 中有关于这个头的一段简短介绍。）

get_unixfrom()
返回消息的信封头。如果信封头从未被设置过，默认返回 `None` 。

以下方法实现了对信息的头字段进行访问的类映射接口。请留意，只是类映射接口，这与平常的映射接口（比如说字典映射）有一些语义上的不同。举个例子，在一个字典当中，键之间不可重复，但是信息头字段是可以重复的。不光如此，在字典当中调用*keys()* 方法返回的结果，其顺序没有保证；但是在一个*EmailMessage* 对象当中，返回的头字段永远以其在原信息当中出现的顺序，或以其加入信息的顺序为序。任何删了后又重新加回去的头字段总是添加在当时列表的末尾。

这些语义上的不同是刻意而为之的，是出于在绝大多数常见使用情景中都方便的初衷下设计的。

还请留意，无论在什么情况下，消息当中的任何信封头字段都不会包含在映射接口当中。

__len__()
返回头字段的总数，重复的也计算在内。

__contains__(*name*)
Return true if the message object has a field named *name*. Matching is done without regard to case and *name* does not include the trailing colon. Used for the `in` operator. For example:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)
返回头字段名对应的字段值。*name* 不含冒号分隔符。如果字段未找到，返回 None 。*KeyError* 异常永不抛出。

请注意，如果对应名字的字段找到了多个，具体返回哪个字段值是未定义的。请使用*get_all()* 方法获取匹配字段名的所有字段值。

使用标准策略（非 compat32）时，返回值是*email.headerregistry.BaseHeader* 的某个子类的一个实例。

__setitem__(*name*, *val*)
在信息头中添加名为 *name* 值为 *val* 的字段。这个字段会被添加在已有字段列表的结尾处。

请注意，这个方法 既不会覆盖 也不会删除任何字段名重名的已有字段。如果你确实想保证新字段是整个信息头当中唯一拥有 *name* 字段名的字段，你需要先把旧字段删除。例如：

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

如果 policy 明确要求某些字段是唯一的（至少标准策略就有这么做），对这些字段在已有同名字段的情况下仍然尝试为字段名赋值会引发*ValueError* 异常。这是为了一致性而刻意设计出的行为，不过我们随时可能会突然觉得“还是在这种情况下自动把旧字段删除比较好吧”而把这个行为改掉，所以不要以为这是特性而依赖这个行为。

__delitem__(*name*)
删除信息头当中字段名匹配 *name* 的所有字段。如果匹配指定名称的字段没有找到，也不会抛出任何异常。

keys()
以列表形式返回消息头中所有的字段名。

values()
以列表形式返回消息头中所有的字段值。

items()
以二元元组的列表形式返回消息头中所有的字段名和字段值。

get(*name*, *failobj=None*)
返回对应字段名的字段值。这个方法与*__getitem__()* 是一样的，只不过如果对应字段名的字段没有找到，该方法会返回 *failobj* 。这个参数是可选的（默认值为 None ）。

以下是一些与头有关的更多有用方法：

get_all(*name*, *failobj=None*)
返回字段名为 *name* 的所有字段值的列表。如果信息内不存在匹配的字段，返回 *failobj*（其默认值为 None ）。

add_header(*_name*, *_value*, ***_params*)
高级头字段设定。这个方法与*__setitem__()* 类似，不过你可以使用关键字参数为字段提供附加参数。*_name* 是字段名，*_value* 是字段 主值。

对于关键字参数字典 *_params* 的每个键值对而言，它的键被用作参数的名字，其中下划线被替换为短横杠（毕竟短横杠不是合法的 Python 标识符）。一般来讲，参数以 键 =" 值" 的方式添加，除非值是 None 。要真的是这样的话，只有键会被添加。

如果值含有非 ASCII 字符，你可以将值写成（CHARSET, LANGUAGE, VALUE）形式的三元组，这样你可以人为控制字符的字符集和语言。CHARSET 是一个字符串，它为你的值的编码命名；LANGUAGE 一般可以直接设为 None ，也可以直接设为空字符串（其他可能取值参见:rfc‘2231‘）；`VALUE 是一个字符串值，其包含非 ASCII 的码点。如果你没有使用三元组，你的字符串又含有非 ASCII 字符，那么它就会使用 **RFC 2231** 中，CHARSET 为 utf-8 ，LANGUAGE 为 None 的格式编码。

例如：

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

会添加一个形如下文的头字段：

```
Content-Disposition: attachment; filename="bud.gif"
```

带有非 ASCII 字符的拓展接口：

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header(*_name*, *_value*)
替换头字段。只会替换掉信息内找到的第一个字段名匹配 *_name* 的字段值。字段的顺序不变，原字段名的大小写也不变。如果没有找到匹配的字段，抛出*KeyError* 异常。

get_content_type()
返回信息的内容类型，其形如 *maintype/subtype* ，强制全小写。如果信息的 *Content-Type* 头字段不存在则返回*get_default_type()* 的返回值；如果信息的 *Content-Type* 头字段无效则返回 text/plain 。

（根据 **RFC 2045** 所述，信息永远都有一个默认类型，所以*get_content_type()* 一定会返回一个值。**RFC 2045** 定义信息的默认类型为 *text/plain* 或 *message/rfc822* ，其中后者仅出现在消息头位于一个 *multipart/digest* 容器中的场合中。如果消息头的 *Content-Type* 字段所指定的类型是无效的，**RFC 2045** 令其默认类型为 *text/plain* 。）

get_content_maintype()
返回信息的主要内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *maintype* 部分。

get_content_subtype()
返回信息的子内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *subtype* 部分。

get_default_type()
返回默认的内容类型。绝大多数的信息，其默认内容类型都是 *text/plain* 。作为 *multipart/digest* 容器内子部分的信息除外，它们的默认内容类型是 *message/rfc822* 。

set_default_type(*ctype*)
设置默认的内容类型。尽管并非强制，但是 *ctype* 仍应当是 *text/plain* 或 *message/rfc822* 二者取一。默认内容类型并不存储在 *Content-Type* 头字段当中，所以设置此项的唯一作用就是决定当 *Content-Type* 头字段在信息中不存在时，get_content_type 方法的返回值。

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)
在 *Content-Type* 头字段当中设置一个参数。如果该参数已于字段中存在，将其旧值替换为 *value* 。如果 *header* 是 Content-Type （默认值），并且该头字段于信息中尚未存在，则会先添加该字

段，将其值设置为 *text/plain* ，并附加参数值。可选的 *header* 可以让你指定 *Content-Type* 之外的另一个头字段。

如果值包含非 ASCII 字符，其字符集和语言可以通过可选参数 *charset* 和 *language* 显式指定。可选参数 *language* 指定 **RFC 2231** 当中的语言，其默认值是空字符串。*charset* 和 *language* 都应当字符串。默认使用的是 utf8 *charset* ， *language* 为 None 。

如果 *replace* 为 False （默认值），该头字段会被移动到所有头字段的末尾。如果 *replace* 为 True ，字段会被原地更新。

于*EmailMessage* 对象而言，*requote* 参数已被弃用。

请注意，头字段已有的参数值可以通过头字段的 params 属性来访问（举例: msg['Content-Type'].params['charset'] ）。

在 3.4 版更改: 添加了 replace 关键字。

del_param(*param*, *header='content-type'*, *requote=True*)
从 *Content-Type* 头字段中完全移去给定的参数。头字段会被原地重写，重写后的字段不含参数和值。可选的 *header* 可以让你指定 *Content-Type* 之外的另一个字段。

于*EmailMessage* 对象而言，*requote* 参数已被弃用。

get_filename(*failobj=None*)
返回信息头当中 *Content-Disposition* 字段当中名为 filename 的参数值。如果该字段当中没有此参数，该方法会退而寻找 *Content-Type* 字段当中的 name 参数值。如果这个也没有找到，或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照*email.utils.unquote()* 方法去除引号。

get_boundary(*failobj=None*)
返回信息头当中 *Content-Type* 字段当中名为 boundary 的参数值。如果字段当中没有此参数，或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照*email.utils.unquote()* 方法去除引号。

set_boundary(*boundary*)
将 *Content-Type* 头字段的 boundary 参数设置为 *boundary* 。*set_boundary()* 方法永远都会在必要的时候为 *boundary* 添加引号。如果信息对象中没有 *Content-Type* 头字段，抛出*HeaderParseError* 异常。

请注意使用这个方法与直接删除旧的 *Content-Type* 头字段然后使用*add_header()* 方法添加一个带有新边界值参数的 *Content-Type* 头字段有细微差距。*set_boundary()* 方法会保留 *Content-Type* 头字段在原信息头当中的位置。

get_content_charset(*failobj=None*)
返回 *Content-Type* 头字段中的 charset 参数，强制小写。如果字段当中没有此参数，或者这个字段压根不存在，返回 *failobj* 。

get_charsets(*failobj=None*)
返回一个包含了信息内所有字符集名字的列表。如果信息是 *multipart* 类型的，那么列表当中的每一项都对应其负载的子部分的字符集名字。否则，该列表是一个长度为 1 的列表。

列表当中的每一项都是一个字符串，其值为对应子部分的 *Content-Type* 头字段的 charset 参数值。如果该子部分没有此头字段，或者没有此参数，或者其主要 MIME 类型并非 *text* ，那么列表中的那一项即为 *failobj* 。

is_attachment()
如果信息头当中存在一个名为 *Content-Disposition* 的字段，且该字段的值为 attachment （大小写无关），返回 True 。否则，返回 False 。

在 3.4.2 版更改: 为了与*is_multipart()* 方法一致，is_attachment 现在是一个方法，不再是属性了。

get_content_disposition()
如果信息的 *Content-Disposition* 头字段存在，返回其字段值；否则返回 None 。返回的值均为小写，不包含参数。如果信息遵循 **RFC 2183** 标准，则返回值只可能在 *inline* 、*attachment* 和 None 之间选择。

3.5 新版功能.

下列方法与信息内容（负载）之访问与操控有关。

walk()
walk() 方法是一个多功能生成器。它可以被用来以深度优先顺序遍历信息对象树的所有部分和子部分。一般而言，*walk()* 会被用作 for 循环的迭代器，每一次迭代都返回其下一个子部分。

以下例子会打印出一封具有多部分结构之信息的每个部分的 MIME 类型。

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

walk 会遍历所有*is_multipart()* 方法返回 True 的部分之子部分，哪怕 msg.get_content_maintype() == 'multipart' 返回的是 False 。使用 _structure 除错帮助函数可以帮助我们在下面这个例子当中看清楚这一点：

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

在这里，message 的部分并非 multiparts ，但是它们真的包含子部分！is_multipart() 返回 True ， walk 也深入进这些子部分中。

get_body(*preferencelist=('related', 'html', 'plain')*)
返回信息的 MIME 部分。这个部分是最可能成为信息体的部分。

preferencelist 必须是一个字符串序列，其内容从 related 、html 和 plain 这三者组成的集合中选取。这个序列代表着返回的部分的内容类型之偏好。

在 get_body 方法被调用的对象上寻找匹配的候选者。

如果 related 未包括在 *preferencelist* 中，可考虑将所遇到的任意相关的根部分（或根部分的子部分）在该（子）部分与一个首选项相匹配时作为候选项。

当遇到一个 multipart/related 时，将检查 start 形参并且如果找到了一个匹配 *Content-ID* 的部分，在查找候选匹配时只考虑它。在其他情况下则只考虑 multipart/related 的第一个（默认的根）部分。

如果一个部分具有 *Content-Disposition* 标头，则当标头值为 inline 时将只考虑将该部分作为候选匹配。

如果没有任何候选部分匹配 *preferencelist* 中的任何首选项，则返回 None。

注: (1) 对于大多数应用来说有意义的 *preferencelist* 组合仅有 ('plain',), ('html', 'plain') 以及默认的 ('related', 'html', 'plain')。(2) 由于匹配是从调用 get_body 的对象开始的，因此在 multipart/related 上调用 get_body 将返回对象本身，除非 *preferencelist* 具有非默认值。(3) 未指定 *Content-Type* 或者 *Content-Type* 标头无效的消息（或消息部分）将被当作具有 text/plain 类型来处理，这有时可能导致 get_body 返回非预期的结果。

iter_attachments()
返回包含所有不是候选 “body” 部分的消息的直接子部分的迭代器。也就是说，跳过首次出现的每个 text/plain, text/html, multipart/related 或 multipart/alternative (除非通过 *Content-Disposition: attachment* 将它们显式地标记为附件)，并返回所有的其余部分。当直接应用于 multipart/related 时，将返回包含除根部分之外所有相关部分的迭代器（即由 start 形参所指向的部分，或者当没有 start 形参或 start 形参不能匹配任何部分的 *Content-ID* 时则为第一部分）。当直接应用于 multipart/alternative 或非 multipart 时，将返回一个空迭代器。

iter_parts()
返回包含消息的所有直接子部分的迭代器，对于非 multipart 将为空对象。(另请参阅*walk()*。)

get_content(**args*, *content_manager=None*, ***kw*)
调用 *content_manager* 的*get_content()* 方法，将自身作为消息对象传入，并将其他参数或关键字作为额外参数传入。如果未指定 *content_manager*，则会使用当前*policy* 所指定的 content_manager。

set_content(**args*, *content_manager=None*, ***kw*)
调用 *content_manager* 的*set_content()* 方法，将自身作为消息传入，并将其他参数或关键字作为额外参数传入。如果未指定 *content_manager*，则会使用当前*policy* 所指定的 content_manager。

make_related(*boundary=None*)
将非 multipart 消息转换为 multipart/related 消息，将任何现有的 *Content-* 标头和载荷移入 multipart 的（新加）首部分。如果指定了 *boundary*，会用它作为 multipart 中的分界字符串，否则会在必要时自动创建分界（例如当消息被序列化时）。

make_alternative(*boundary=None*)
将非 multipart 或 multipart/related 转换为 multipart/alternative，将任何现有的 *Content-* 标头和载荷移入 multipart 的（新加）首部分。如果指定了 *boundary*，会用它作为 multipart 中的分界字符串，否则会在必要时自动创建分界（例如当消息被序列化时）。

make_mixed(*boundary=None*)
将非 multipart, multipart/related 或 multipart-alternative 转换为 multipart/mixed，将任何现有的 *Content-* 标头和载荷移入 multipart 的（新加）首部分。如果指定了 *boundary*，会用它作为 multipart 中的分界字符串，否则会在必要时自动创建分界（例如当消息被序列化时）。

add_related(**args*, *content_manager=None*, ***kw*)
如果消息为 multipart/related，则创建一个新的消息对象，将所有参数传给其*set_content()* 方法，并将其*attach()* 到 multipart。如果消息为非 multipart，则先调用*make_related()* 然后再继续上述步骤。如果消息为任何其他类型的 multipart，则会引发*TypeError*。如果未指定 *content_manager*，则使用当前*policy* 所指定的 content_manager。如果添加的部分没有 *Content-Disposition* 标头，则会添加一个值为 inline 的标头。

add_alternative(**args*, *content_manager=None*, ***kw*)
如果消息为 multipart/alternative，则创建一个新的消息对象，将所有参数传给其*set_content()* 方法，并将其*attach()* 到 multipart。如果消息为非 multipart 或 multipart/related，则先调用*make_alternative()* 然后再继续上述步骤。如果消息为任何其他类型的 multipart，则会引发*TypeError*。如果未指定 *content_manager*，则会使用当前*policy* 所指定的 content_manager。

add_attachment(**args*, *content_manager=None*, ***kw*)
如果消息为 multipart/mixed，则创建一个新的消息对象，将所有参数传给其*set_content()* 方法，并将其*attach()* 到 multipart。如果消息为非 multipart，multipart/related 或 multipart/alternative，则先调用*make_mixed()* 然后再继续上述步骤。如果未指定 *content_manager*，则使用当前*policy* 所指定的 content_manager。如果添加的部分没有 *Content-Disposition* 标头，则会添加一个值为 attachment 的标头。此方法对于显式附件 (*Content-Disposition: attachment*) 和 inline 附件 (*Content-Disposition: inline*) 均可使用，只须向 content_manager 传入适当的选项即可。

clear()
移除所有载荷和所有标头。

clear_content()
移除载荷以及所有 Content- 标头，其他标头不加改变并且保持其原有顺序。

EmailMessage 对象具有下列实例属性:

preamble
MIME 文档格式在标头之后的空白行以及第一个多部分的分界字符串之间允许添加一些文本，通常，此文本在支持 MIME 的邮件阅读器中永远不可见，因为它处在标准 MIME 防护范围之外。但是，当查看消息的原始文本，或当在不支持 MIME 的阅读器中查看消息时，此文本会变得可见。

preamble 属性包含 MIME 文档开头部分的这些处于保护范围之外的文本。当*Parser* 在标头之后及第一个分界字符串之前发现一些文本时，它会将这些文本赋值给消息的 *preamble* 属性。当*Generator* 写出 MIME 消息的纯文本表示形式时，如果它发现消息具有 *preamble* 属性，它将在标头及第一个分界之间区域写出这些文本。请参阅*email.parser* 和*email.generator* 了解更多细节。

请注意如果消息对象没有前导文本，则 *preamble* 属性将为 None。

epilogue
epilogue 属性的作用方式与 *preamble* 相同，区别在于它包含在最后一个分界及消息结尾之间出现的文本。与*preamble* 类似，如果没有附加文本，则此属性将为 None。

defects
defects 属性包含在解析消息时发现的所有问题的列表。请参阅*email.errors* 了解可能的解析缺陷的详细描述。

class email.message.**MIMEPart**(*policy=default*)
这个类代表 MIME 消息的子部分。它与*EmailMessage* 相似，不同之处在于当*set_content()* 被调用时不会添加 *MIME-Version* 标头，因为子部分不需要有它们自己的 *MIME-Version* 标头。

备注

19.1.2 email.parser: 解析电子邮件信息

源代码: Lib/email/parser.py

使用以下两种方法的其中一种以创建消息对象结构：直接创建一个*EmailMessage* 对象，使用字典接口添加消息头，并且使用*set_content()* 和其他相关方法添加消息负载；或者通过解析一个电子邮件消息的序列化表达来创建消息对象结构。

email 包提供了一个可以理解包含 MIME 文档在内的绝大多数电子邮件文档结构的标准语法分析程序。你可以传递给语法分析程序一个字节串、字符串或者文件对象，语法分析程序会返回给你对应于该对象结构的根*EmailMessage* 实例。对于简单的、非 MIME 的消息，这个根对象的负载很可能就是一个包含了该消息文字内容的字符串。对于 MIME 消息，调用根对象的*is_multipart()* 方法会返回 `True`，其子项可以通过负载操纵方法来进行访问，例如*get_body()*、*iter_parts()* 还有*walk()*。

事实上你可以使用的语法分析程序接口有两种: *Parser* API 和增量式的*FeedParser* API。当你的全部消息内容都在内存当中，或者整个消息都保存在文件系统内的一个文件当中的时候，*Parser* API 非常有用。当你从可能会为了等待更多输入而阻塞的数据流当中读取消息（比如从套接字当中读取电子邮件消息）的时候，*FeedParser* 会更合适。*FeedParser* 会增量读取并解析消息，并且只有在你关闭语法分析程序的时候才会返回根对象。

请注意，语法分析程序可以进行有限的拓展，你当然也可以完全从零开始实现你自己的语法分析程序。将*email* 包中内置的语法分析程序和*EmailMessage* 类连接起来的所有逻辑代码都包含在 `policy` 类当中，所以如有必要，自定义的语法分析程序可以通过实现自定义的对应 `policy` 方法来创建对应的消息对象树。

FeedParser API

从 `email.feedparser` 模块导入的*BytesFeedParser* 类提供了一个适合于增量解析电子邮件消息的 API，比如说在从一个可能会阻塞（例如套接字）的源当中读取消息文字的场合中它就会变得很有用。当然，*BytesFeedParser* 也可以用来解析一个已经完整包含在一个*bytes-like object* 、字符串或文件内的电子邮件消息，但是在这些场合下使用*BytesParser* API 可能会更加方便。这两个语法分析程序 API 的语义和最终结果是一致的。

BytesFeedParser 的 API 十分简洁易懂：你创建一个语法分析程序的实例，向它不断输入大量的字节直到尽头，然后关闭这个语法分析程序就可以拿到根消息对象了。在处理符合标准的消息的时候*BytesFeedParser* 非常准确；在处理不符合标准的消息的时候它做的也不差，但这视消息损坏的程度而定。它会向消息对象的*defects* 属性中写入它从消息中找到的问题列表。关于它能找到的所有问题类型的列表，详见*email.errors* 模块。

这里是*BytesFeedParser* 的 API:

class email.parser.**BytesFeedParser**(*_factory=None*, ***, *policy=policy.compat32*)

创建一个*BytesFeedParser* 实例。可选的 *_factory* 参数是一个不带参数的可调用对象；如果没有被指定，就会使用 *policy* 参数的*message_factory* 属性。每当需要一个新的消息对象的时候，*_factory* 都会被调用。

如果指定了 *policy* 参数，它就会使用这个参数所指定的规则来更新消息的表达方式。如果没有设定 *policy* 参数，它就会使用*compat32* 策略。这个策略维持了对 Python 3.2 版本的 email 包的后向兼容性，并且使用*Message* 作为默认的工厂。其他策略使用*EmailMessage* 作为默认的 *_factory*。关于 *policy* 还会控制什么，参见*policy* 的文档。

注: **一定要指定 policy 关键字**。在未来版本的 Python 当中，它的默认值会变成*email.policy.default*。

3.2 新版功能.

在 3.3 版更改: 添加了 *policy* 关键字。

在 3.6 版更改: *_factory* 默认为策略 `message_factory`。

feed(*data*)
向语法分析程序输入更多数据。*data* 应当是一个包含一行或多行内容的*bytes-like object*。行内容可以是不完整的，语法分析程序会妥善的将这些不完整的行缝合在一起。每一行可以使用以下三种常见的终止符号的其中一种：回车符、换行符或回车符加换行符（三者甚至可以混合使用）。

close()
完成之前输入的所有数据的解析并返回根消息对象。如果在这个方法被调用之后仍然调用*feed()*方法，结果是未定义的。

class email.parser.**FeedParser**(*_factory=None*, *, *policy=policy.compat32*)
行为跟*BytesFeedParser*类一致，只不过向*feed()*方法输入的内容必须是字符串。它的实用性有限，因为这种消息只有在其只含有 ASCII 文字，或者 `utf8` 被设置为 `True` 且没有二进制格式的附件的时候，才会有效。

在 3.3 版更改: 添加了 *policy* 关键字。

Parser API

*BytesParser*类从*email.parser*模块导入，当消息的完整内容包含在一个*bytes-like object*或文件中时它提供了可用于解析消息的 API。*email.parser*模块还提供了*Parser*用来解析字符串，以及只用来解析消息头的*BytesHeaderParser*和*HeaderParser*，如果你只对消息头感兴趣就可以使用后两者。在这种场合下*BytesHeaderParser*和*HeaderParser*速度非常快，因为它们并不会尝试解析消息体，而是将载荷设为原始数据。

class email.parser.**BytesParser**(*_class=None*, *, *policy=policy.compat32*)
创建一个*BytesParser*实例。*_class* 和 *policy* 参数在含义和语义上与*BytesFeedParser*的 *_factory* 和 *policy* 参数一致。

注: **一定要指定 policy 关键字**。在未来版本的 Python 当中，它的默认值会变成*email.policy.default*。

在 3.3 版更改: 移除了在 2.4 版本中被弃用的 *strict* 参数。新增了 *policy* 关键字。

在 3.6 版更改: *_class* 默认为策略 `message_factory`。

parse(*fp*, *headersonly=False*)
从二进制的类文件对象 *fp* 中读取全部数据、解析其字节内容、并返回消息对象。*fp* 必须同时支持*readline()*方法和 `read()` 方法。

fp 内包含的字节内容必须是一块遵循 **RFC 5322** （如果 `utf8` 为 `True`，则为 **RFC 6532** ）格式风格的消息头和消息头延续行，并可能紧跟一个信封头。头块要么以数据结束，要么以一个空行为终止。跟着头块的是消息体（消息体内可能包含 MIME 编码的子部分，这也包括 *Content-Transfer-Encoding* 字段为 `8bit` 的子部分）。

可选的 *headersonly* 指示了是否应当在读取完消息头后就终止。默认值为 “False“，意味着它会解析整个文件的全部内容。

parsebytes(*bytes*, *headersonly=False*)
与*parse()*方法类似，只不过它要求输入为一个*bytes-like object*而不是类文件对象。于一个*bytes-like object*调用此方法相当于先将这些字节包装于一个*BytesIO*实例中，然后调用*parse()*方法。

可选的 *headersonly* 与*parse()*方法中的 *headersonly* 是一致的。

3.2 新版功能.

class email.parser.**BytesHeaderParser**(*_class=None*, *, *policy=policy.compat32*)
除了 *headersonly* 默认为 True，其他与*BytesParser* 类完全一样。

3.3 新版功能.

class email.parser.**Parser**(*_class=None*, *, *policy=policy.compat32*)
这个类与*BytesParser* 一样，但是处理字符串输入。

在 3.3 版更改: 移除了 *strict* 参数。添加了 *policy* 关键字。

在 3.6 版更改: *_class* 默认为策略 message_factory。

parse(*fp*, *headersonly=False*)
从文本模式的文件类对象 *fp* 读取所有数据，解析所读取的文本，并返回根消息对象。*fp* 必须同时支持文件类对象上的*readline()* 和*read()* 方法。

除了文字模式的要求外，这个方法跟*BytesParser.parse()* 的运行方式一致。

parsestr(*text*, *headersonly=False*)
与*parse()* 方法类似，只不过它要求输入为一个字符串而不是类文件对象。于一个字符串对象调用此方法相当于先将 *text* 包装于一个*StringIO* 实例中，然后调用*parse()* 方法。

可选的 *headersonly* 与*parse()* 方法中的 *headersonly* 是一致的。

class email.parser.**HeaderParser**(*_class=None*, *, *policy=policy.compat32*)
除了 *headersonly* 默认为 True，其他与*Parser* 类完全一样。

考虑到从一个字符串或一个文件对象中创建一个消息对象是非常常见的任务，我们提供了四个方便的函数。它们于顶层*email* 包命名空间内可用。

email.**message_from_bytes**(*s*, *_class=None*, *, *policy=policy.compat32*)
从一个*bytes-like object* 中返回消息对象。这与 BytesParser().parsebytes(s) 等价。可选的 *_class* 和 *policy* 参数与*BytesParser* 类的构造函数的参数含义一致。

3.2 新版功能.

在 3.3 版更改: 移除了 *strict* 参数。添加了 *policy* 关键字。

email.**message_from_binary_file**(*fp*, *_class=None*, *, *policy=policy.compat32*)
从打开的二进制*file object* 中返回消息对象。这与 BytesParser().parse(fp) 等价。*_class* 和 *policy* 参数与*BytesParser* 类的构造函数的参数含义一致。

3.2 新版功能.

在 3.3 版更改: 移除了 *strict* 参数。添加了 *policy* 关键字。

email.**message_from_string**(*s*, *_class=None*, *, *policy=policy.compat32*)
从一个字符串中返回消息对象。这与 Parser().parsestr(s) 等价。*_class* 和 *policy* 参数与*Parser* 类的构造函数的参数含义一致。

在 3.3 版更改: 移除了 *strict* 参数。添加了 *policy* 关键字。

email.**message_from_file**(*fp*, *_class=None*, *, *policy=policy.compat32*)
从一个打开的*file object* 中返回消息对象。这与 Parser().parse(fp) 等价。*_class* 和 *policy* 参数与*Parser* 类的构造函数的参数含义一致。

在 3.3 版更改: 移除了 *strict* 参数。添加了 *policy* 关键字。

在 3.6 版更改: *_class* 默认为策略 message_factory。

这里是一个展示了你如何在 Python 交互式命令行中使用*message_from_bytes()* 的例子:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

附加说明

在解析语义的时候请注意：

- 大多数非 *multipart* 类型的消息都会被解析为一个带有字符串负载的消息对象。这些对象在调用*is_multipart()* 的时候会返回 “False“，调用*iter_parts()* 的时候会返回一个空列表。
- 所有 *multipart* 类型的消息都会被解析成一个容器消息对象。该对象的负载是一个子消息对象列表。外层的容器消息在调用*is_multipart()* 的时候会返回 “True“，在调用*iter_parts()* 的时候会产生一个子部分列表。
- 大多数内容类型为 *message/** （例如 *message/delivery-status* 和 *message/rfc822* ）的消息也会被解析为一个负载是长度为 1 的列表的容器对象。在它们身上调用*is_multipart()* 方法会返回 “True“，调用*iter_parts()* 所产生的单个元素会是一个子消息对象。
- 一些不遵循标准的消息在其内部关于它是否为 *multipart* 类型前后不一。这些消息可能在消息头的 *Content-Type* 字段中写明为 *multipart* ，但它们的*is_multipart()* 方法的返回值可能是 “False“。如果这种消息被*FeedParser* 类解析，它们的 *defects* 属性列表当中会有一个 `MultipartInvariantViolationDefect` 类的实例。关于更多信息，详见*email.errors* 。

19.1.3 email.generator: 生成 MIME 文档

源代码: Lib/email/generator.py

最普通的一种任务是生成由消息对象结构体表示的电子邮件消息的扁平（序列化）版本。如果你想通过*smtplib.SMTP.sendmail()* 或*nntplib* 模块来发送你的消息或是将消息打印到控制台就将需要这样做。接受一个消息对象结构体并生成其序列化表示就是这些生成器类的工作。

与*email.parser* 模块一样，你并不会受限于已捆绑生成器的功能；你可以自己从头写一个。不过，已捆绑生成器知道如何以符合标准的方式来生成大多数电子邮件，应该能够很好地处理 MIME 和非 MIME 电子邮件消息，并且被设计为面向字节的解析和生成操作是互逆的，它假定两者都使用同样的非转换型*policy*。也就是说，通过*BytesParser* 类来解析序列化字节流然后再使用*BytesGenerator* 来重新生成序列化字节流应当得到与输入相同的结果[1]。（而另一方面，在由程序所构造的*EmailMessage* 上使用生成器可能导致对默认填入的*EmailMessage* 对象的改变。。）

可以使用*Generator* 类将消息扁平化为文本（而非二进制数据）的序列化表示形式，但是由于 Unicode 无法直接表示二进制数据，因此消息有必要被转换为仅包含 ASCII 字符的数据，这将使用标准电子邮件 RFC 内容传输编码格式技术来编码电子邮件消息以便通过非 “8 比特位兼容” 的信道来传输。

class email.generator.**BytesGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

返回一个*BytesGenerator* 对象，该对象将把提供给*flatten()* 方法的任何消息或者提供给*write()* 方法的任何经过代理转义编码的文本写入到*file-like object* *outfp*。*outfp* 必须支持接受二进制数据的 `write` 方法。

If optional *mangle_from_* is `True`, put a > character in front of any line in the body that starts with the exact string `"From "`, that is `From` followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the *mangle_from_* setting of the *policy* (which is `True` for the *compat32* policy and `False` for all others). *mangle_from_* is intended for use when messages are stored in unix mbox format (see *mailbox* and WHY THE CONTENT-LENGTH FORMAT IS BAD).

如果 *maxheaderlen* 不为 `None`，则重新折叠任何长于 *maxheaderlen* 的标头行，或者如果为 `0`，则不重新包装任何标头。如果 *manheaderlen* 为 `None` (默认值)，则根据 *policy* 设置包装标头和其他消息行。

[1] 此语句假定你使用了正确的 `unixfrom` 设置，并且没有用于自动调整的 `policy` 设置调用（例如 `refold_source` 必须为 `none`，这 不是默认值）。这也不是 100% 为真，因为如果消息不遵循 RFC 标准则有时实际原始文本的信息会在解析错误恢复时丢失。它的目标是在可能的情况下修复这些后续边缘情况。

如果指定了 *policy*，则使用该策略来控制消息的生成。如果 *policy* 为 None (默认值)，则使用与传递给 flatten 的*Message* 或*EmailMessage* 对象相关联的策略来控制消息的生成。请参阅*email.policy* 了解有关 *policy* 所控制内容的详情。

3.2 新版功能.

在 3.3 版更改: 添加了 *policy* 关键字。

在 3.6 版更改: *mangle_from_* 和 *maxheaderlen* 形参的默认行为是遵循策略。

flatten(*msg*, *unixfrom=False*, *linesep=None*)

将将以 *msg* 为根的消息对象结构体的文本表示形式打印到创建*BytesGenerator* 实例时指定的输出文件。

如果*policy* 选项*cte_type* 为 8bit (默认值)，则会将未被修改的原始已解析消息中的任何标头拷贝到输出，其中会重新生成与原始数据相同的高比特位组字节数据，并保留具有它们的任何消息体部分的非 ASCII *Content-Transfer-Encoding*。如果 cte_type 为 7bit，则会根据需要使用兼容 ASCII 的 *Content-Transfer-Encoding* 来转换高比特位组字节数据。也就是说，将具有非 ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) 的部分转换为兼容 ASCII 的 *Content-Transfer-Encoding*，并使用 MIME unknown-8bit 字符集来编码标头中不符合 RFC 的非 ASCII 字节数据，以使其符合 RFC。

如果 *unixfrom* 为 True，则会在根消息对象的第一个 **RFC 5322** 标头之前打印 Unix mailbox 格式 (参见*mailbox*) 所使用的封包标头分隔符。如果根对象没有封包标头，则会创建一个标准标头。默认值为 False。请注意对于子部分来说，不会打印任何封包标头。

如果 *linesep* 不为 None，则会将其用作扁平化消息的所有行之间的分隔符。如果 *linesep* 为 None (默认值)，则使用在 *policy* 中指定的值。

clone(*fp*)

返回此 Return an independent clone of this *BytesGenerator* 实例的独立克隆，具有完全相同的选项设置，而 *fp* 为新的 *outfp*。

write(*s*)

使用 ASCII 编解码器和 surrogateescape 错误处理程序编码 *s*，并将其传递给传入到*BytesGenerator* 的构造器的 *outfp* 的 *write* 方法。

作为一个便捷工具，*EmailMessage* 提供了*as_bytes()* 和 bytes(aMessage) (即*__bytes__()*) 等方法，它们简单地生成一个消息对象的序列化二进制表示形式。更多细节请参阅*email.message*。

因为字符串无法表示二进制数据，*Generator* 类必须将任何消息中扁平化的任何二进制数据转换为兼容 ASCII 的格式，具体将其转换为兼容 ASCII 的 *Content-Transfer_Encoding*。使用电子邮件 RFC 的术语，你可以将其视作*Generator* 序列化为不“支持 8 比特”的 I/O 流。换句话说，大部分应用程序将需要使用*BytesGenerator*，而非*Generator*。

class email.generator.**Generator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

返回一个*Generator*，它将把提供给*flatten()* 方法的任何消息，或者提供给*write()* 方法的任何文本写入到*file-like object* *outfp*。*outfp* 必须支持接受字符串数据的 write 方法。

If optional *mangle_from_* is True, put a > character in front of any line in the body that starts with the exact string "From ", that is From followed by a space at the beginning of a line. *mangle_from_* defaults to the value of the *mangle_from_* setting of the *policy* (which is True for the *compat32* policy and False for all others). *mangle_from_* is intended for use when messages are stored in unix mbox format (see *mailbox* and WHY THE CONTENT-LENGTH FORMAT IS BAD).

如果 *maxheaderlen* 不为 None，则重新折叠任何长于 *maxheaderlen* 的标头行，或者如果为 0，则不重新包装任何标头。如果 *manheaderlen* 为 None (默认值)，则根据 *policy* 设置包装标头和其他消息行。

如果指定了 *policy*，则使用该策略来控制消息的生成。如果 *policy* 为 None (默认值)，则使用与传递

给 flatten 的*Message* 或*EmailMessage* 对象相关联的策略来控制消息的生成。请参阅*email.policy* 了解有关 *policy* 所控制内容的详情。

在 3.3 版更改: 添加了 *policy* 关键字。

在 3.6 版更改: *mangle_from_* 和 *maxheaderlen* 形参的默认行为是遵循策略。

flatten(*msg*, *unixfrom=False*, *linesep=None*)

将以 *msg* 为根的消息对象结构体的文本表示形式打印到当*Generator* 实例被创建时所指定的输出文件。

If the *policy* option *cte_type* is 8bit, generate the message as if the option were set to 7bit. (This is required because strings cannot represent non-ASCII bytes.) Convert any bytes with the high bit set as needed using an ASCII-compatible *Content-Transfer-Encoding*. That is, transform parts with non-ASCII *Cotnent-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) to an ASCII compatible *Content-Transfer-Encoding*, and encode RFC-invalid non-ASCII bytes in headers using the MIME unknown-8bit character set, thus rendering them RFC-compliant.

如果 *unixfrom* 为 True，则会在根消息对象的第一个 **RFC 5322** 标头之前打印 Unix mailbox 格式(参见*mailbox*) 所使用的封包标头分隔符。如果根对象没有封包标头，则会创建一个标准标头。默认值为 False。请注意对于子部分来说，不会打印任何封包标头。

如果 *linesep* 不为 None，则会将其用作扁平化消息的所有行之间的分隔符。如果 *linesep* 为 None (默认值)，则使用在 *policy* 中指定的值。

在 3.2 版更改: 添加了对重编码 8bit 消息体的支持，以及 *linesep* 参数。

clone(*fp*)

返回此*Generator* 实例的独立克隆，具有完全相同的选项设置，而 *fp* 为新的 *outfp*。

write(*s*)

将 *s* 写入到传给*Generator* 的构造器的 *outfp* 的 *write* 方法。这足够为*Generator* 实际提供可用于*print()* 函数的文件类 API。

作为一个便捷工具，*EmailMessage* 提供了*as_string()* 和 str(aMessage) (即*__str__()*) 等方法，它们简单地生成一个消息对象的已格式化字符串表示形式。更多细节请参阅*email.message*。

email.generator 模块还提供了一个派生类*DecodedGenerator*，它类似于*Generator* 基类，不同之处在于非 *text* 部分不会被序列化，而是被表示为基于模板并填写了有关该部分的信息的字符串输出流的形式。

class email.generator.**DecodedGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *fmt=None*, ***, *policy=None*)

行为类似于*Generator*，不同之处在于对传给*Generator.flatten()* 的消息的任何子部分，如果该子部分的主类型为 *text*，则打印该子部分的已解码载荷，而如果其主类型不为 *text*，则不直接打印它而是使用来自该部分的信息填入字符串 *fmt* 并将填写完成的字符串打印出来。

要填入 *fmt*，则执行 fmt % part_info，其中 part_info 是由下列键和值组成的字典:

- type –非 *text* 部分的完整 MIME 类型
- maintype –非 *text* 部分的主 MIME 类型
- subtype –非 *text* 部分的子 MIME 类型
- filename –非 *text* 部分的文件名
- description –与非 *text* 部分相关联的描述
- encoding –非 *text* 部分的内容转换编码格式

如果 *fmt* 为 None，则使用下列默认 *fmt*:

"[忽略消息的非文本 (%(type)s) 部分，文件名%(filename)s]"

可选的 *_mangle_from_* 和 *maxheaderlen* 与*Generator* 基类的相同。

备注

19.1.4 email.policy: 策略对象

3.3 新版功能.

源代码: Lib/email/policy.py

---

email 的主要焦点是按照各种电子邮件和 MIME RFC 的描述来处理电子邮件消息。但是电子邮件消息的基本格式（一个由名称加冒号加值的标头字段构成的区块，后面再加一个空白行和任意的‘消息体’）是在电子邮件领域以外也获得应用的格式。这些应用的规则有些与主要电子邮件 RFC 十分接近，有些则很不相同。即使是操作电子邮件，有时也可能需要打破严格的 RFC 规则，例如生成可与某些并不遵循标准的电子邮件服务器互联的电子邮件，或者是实现希望应用某些破坏标准的操作方式的扩展。

策略对象给予 email 包处理这些不同用例的灵活性。

Policy 对象封装了一组属性和方法用来在使用期间控制 email 包中各个组件的行为。*Policy* 实例可以被传给 email 包中的多个类和方法以更改它们的默认行为。可设置的值及其默认值如下所述。

在 email 包中的所有类会使用一个默认的策略。对于所有*parser* 类及相关的便捷函数，还有对于*Message* 类来说，它是*Compat32* 策略，通过其对应的预定义实例*compat32* 来使用。这个策略提供了与 Python3.3 版之前的 email 包的完全向下兼容性（在某些情况下，也包括对缺陷的兼容性）。

传给*EmailMessage* 的 *policy* 关键字的默认值是*EmailPolicy* 策略，表示为其预定义的实例*default*。

在创建*Message* 或*EmailMessage* 对象时，它需要一个策略。如果消息是由*parser* 创建的，则传给该解析器的策略将是它所创建的消息所使用的策略。如果消息是由程序创建的，则该策略可以在创建它的时候指定。当消息被传递给*generator* 时，生成器默认会使用来自该消息的策略，但你也可以将指定的策略传递给生成器，这将覆盖存储在消息对象上的策略。

email.parser 类和解析器便捷函数的 *policy* 关键字的默认值在未来的 Python 版本中 **将会改变**。因此在调用任何*parser* 模块所描述的类和函数时你应当 **总是显式地指定你想要使用的策略**。

本文档的第一部分介绍了*Policy* 的特性，它是一个*abstract base class*，定义了所有策略对象包括*compat32* 的共有特性。这些特性包括一些由 email 包内部调用的特定钩子方法，自定义策略可以重载这些方法以获得不同行为。第二部分描述了实体类*EmailPolicy* 和*Compat32*，它们分别实现了提供标准行为和向下兼容行为与特性的钩子。

Policy 实例是不可变的，但它们可以被克隆，接受与类构造器一致的关键字参数并返回一个新的*Policy* 实例，新实例是原实例的副本，但具有被改变的指定属性。

例如，以下代码可以被用来从一个 Unix 系统的磁盘文件中读取电子邮件消息并将其传递给系统的 sendmail 程序:

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
>>> p.stdin.close()
>>> rc = p.wait()
```

这里我们让*BytesGenerator* 在创建要送入 sendmail's stdin 的二进制字串时使用符合 RFC 的行分隔字符，默认的策略将会使用 \n 行分隔符。

某些 email 包的方法接受一个 *policy* 关键字参数，允许为该方法重载策略。例如，以下代码使用了来自之前示例的 *msg* 对象的*as_bytes()* 方法并使用其运行所在平台的本机行分隔符将消息写入一个文件:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

Policy 对象也可使用附加运算符进行组合来产生一个新策略对象，其设置是被加总对象的非默认值的组合:

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

此运算不满足交换律；也就是说对象的添加顺序很重要。见以下演示:

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.**Policy**(***kw*)

这是所有策略类的*abstract base class*。它提供了一些简单方法的默认实现，以及不可变特征属性，*clone()* 方法以及构造器语义的实现。

可以向策略类的构造器传入各种关键字参数。可以指定的参数是该类的任何非方法特征属性，以及实体类的任何额外非方法特征属性。在构造器中指定的值将覆盖相应属性的默认值。

这个类定义了下列特征属性，因此下列值可以被传给任何策略类的构造器:

max_line_length

序列化输出中任何行的最大长度，不计入行字符的末尾。默认值为 78，基于 **RFC 5322**。值为 0 或*None* 表示完全没有行包装。

linesep

用来在序列化输出中确定行的字符串。默认值为 \n 因为这是 Python 所使用的内部行结束符规范，但 RFC 的要求是 \r\n。

cte_type

控制可能要求使用的内容传输编码格式类型。可能的值包括:

7bit	所有数据必须为“纯 7 比特位”(仅 ASCII)。这意味着在必要情况下数据将使用可打印引用形式或 base64 编码格式进行编码。
8bit	数据不会被限制为纯 7 比特位。标头中的数据仍要求仅 ASCII 因此将被编码（参阅下文的*fold_binary()* 和*utf8* 了解例外情况），但消息体部分可能使用 8bit CTE。

cte_type 值为 8bit 仅适用于 BytesGenerator 而非 Generator，因为字符串不能包含二进制数据。如果 Generator 运行于指定了 cte_type=8bit 的策略，它的行为将与 cte_type 为 7bit 相同。

raise_on_defect
如为*True*，则遇到的任何缺陷都将引发错误。如为*False* (默认值)，则缺陷将被传递给*register_defect()* 方法。

mangle_from_
如为*True*，则消息体中以 “*From* “开头的行会通过在其前面放一个 > 来进行转义。当消息被生成器执行序列化时会使用此形参。默认值 t: *False*。

3.5 新版功能: *mangle_from_* 形参。

message_factory
用来构造新的空消息对象的工厂函数。在构建消息时由解析器使用。默认为 None，在此情况下会使用*Message*。

3.6 新版功能.

下列*Policy* 方法是由使用 email 库的代码来调用以创建具有自室外设置的策略实例:

clone(***kw*)
返回一个新的*Policy* 实例，其属性与当前实例具有相同的值，除非是那些由关键字参数给出了新值的属性。

其余的*Policy* 方法是由 email 包代码来调用的，而不应当被使用 email 包的应用程序所调用。自定义的策略必须实现所有这些方法。

handle_defect(*obj*, *defect*)
处理在 *obj* 上发现的 *defect*。当 email 包调用此方法时，*defect* 将总是 Defect 的一个子类。

默认实现会检查*raise_on_defect* 旗标。如果其为 True，则 *defect* 会被作为异常来引发。如果其为 False (默认值)，则 *obj* 和 *defect* 会被传递给*register_defect()*。

register_defect(*obj*, *defect*)
在 *obj* 上注册一个 *defect*。在 email 包中，*defect* 将总是 Defect 的一个子类。

默认实现会调用 *obj* 的 defects 属性的 append 方法。当 email 包调用*handle_defect* 时，*obj* 通常将具有一个带 append 方法的 defects 属性。配合 email 包使用的自定义对象类型（例如自定义的 Message 对象）也应当提供这样的属性，否则在被解析消息中的缺陷将引发非预期的错误。

header_max_count(*name*)
返回名为 *name* 的标头的最大允许数量。

当添加一个标头到*EmailMessage* 或*Message* 对象时被调用。如果返回值不为 0 或 None，并且已有的名称为 *name* 的标头数量大于等于所返回的值，则会引发*ValueError*。

由于 Message.__setitem__ 的默认行为是将值添加到标头列表，因此很容易不知情地创建重复的标头。此方法允许在程序中限制可以被添加到 Message 中的特定标头的实例数量。(解析器不会考虑此限制，它将忠实地产生被解析消息中存在的任意数量的标头。)

默认实现对于所有标头名称都返回 None。

header_source_parse(*sourcelines*)
email 包调用此方法时将传入一个字符串列表，其中每个字符串以在被解析源中找到的行分隔符结束。第一行包括字段标头名称和分隔符。源中的所有空白符都会被保留。此方法应当返回 (name, value) 元组以保存至 Message 中来代表被解析的标头。

如果一个实现希望保持与现有 email 包策略的兼容性，则 *name* 应当为保留大小写形式的名称（所有字符直至 ‘:’ 分隔符)，而 *value* 应当为展开后的值（移除所有行分隔符，但空白符保持不变)，并移除开头的空白符。

sourcelines 可以包含经替代转义的二进制数据。

此方法没有默认实现

header_store_parse(*name*, *value*)
当一个应用通过程序代码修改 Message (而不是由解析器创建 Message) 时，email 包会调用此方法时并附带应用程序所提供的名称和值。此方法应当返回 (name, value) 元组以保存至 Message 中用来表示标头。

如果一个实现希望保持与现有 email 包策略的兼容性，则 *name* 和 *value* 应当为字符串或字符串的子类，它们不会修改在参数中传入的内容。

此方法没有默认实现

header_fetch_parse(*name*, *value*)
当标头被应用程序所请求时，email 包会调用此方法并附带当前保存在 Message 中的 *name* 和 *value*，并且无论此方法返回什么它都会被回传给应用程序作为被提取标头的值。请注意可能会有多个相同名称的标头被保存在 Message 中；此方法会将指定标头的名称和值返回给应用程序。

value 可能包含经替代转义的二进制数据。此方法所返回的值应当没有经替代转义的二进制数据。

此方法没有默认实现

fold(*name*, *value*)
email 包调用此方法时会附带当前保存在 Message 中的给定标头的 *name* 和 *value*。此方法应当返回一个代表该标头的（根据策略设置）通过处理 *name* 和 *value* 并在适当位置插入 *linesep* 字符来正确地“折叠”的字符串。请参阅 **RFC 5322** 了解有关折叠电子邮件标头的规则的讨论。

value 可能包含经替代转义的二进制数据。此方法所返回的字符串应当没有经替代转义的二进制数据。

fold_binary(*name*, *value*)
与 *fold()* 类似，不同之处在于返回的值应当为字节串对象而非字符串。

value 可能包含经替代转义的二进制数据。这些数据可以在被返回的字节串对象中被转换回二进制数据。

class email.policy.**EmailPolicy**(***kw*)
这个实体 *Policy* 提供了完全遵循当前电子邮件 RFC 的行为。这包括 (但不限于) **RFC 5322**, **RFC 2047** 以及当前的各种 MIME RFC。

此策略添加了新的标头解析和折叠算法。标头不是简单的字符串，而是带有依赖于字段类型的属性的 str 的子类。这个解析和折叠算法完整实现了 **RFC 2047** 和 **RFC 5322**。

message_factory 属性的默认值为 *EmailMessage*。

除了上面列出的适用于所有策略的可设置属性，此策略还添加了下列额外属性:

3.6 新版功能:[1]

utf8
如为 False，则遵循 **RFC 5322**，通过编码为“已编码字”来支持标头中的非 ASCII 字符。如为 True，则遵循 **RFC 6532** 并对标头使用 utf-8 编码格式。以此方式格式化的消息可被传递给支持 SMTPUTF8 扩展 (**RFC 6531**) 的 SMTP 服务器。

refold_source
如果 Message 对象中标头的值源自 *parser* (而非由程序设置)，此属性会表明当将消息转换回序列化形式时是否应当由生成器来重新折叠该值。可能的值如下:

none	所有源值使用原始折叠
long	具有任何长度超过 max_line_length 的行的源值将被折叠
all	所有值会被重新折叠。

默认值为 long。

[1] 最初在 3.3 中作为暂定特性 添加。

header_factory
该可调用对象接受两个参数，name 和 value，其中 name 为标头字段名而 value 为展开后的标头字段值，并返回一个表示该标头的字符串子类。已提供的默认 header_factory (参见*headerregistry*) 支持对各种地址和日期 **RFC 5322** 标头字段类型及主要 MIME 标头字段类型的自定义解析。未来还将添加对其他自定义解析的支持。

content_manager
此对象至少有两个方法: get_content 和 set_content。当一个*EmailMessage* 对象的*get_content()* 或*set_content()* 方法被调用时，它会调用此对象的相应方法，将消息对象作为其第一个参数，并将传给它的任何参数或关键字作为附加参数传入。默认情况下 content_manager 会被设为*raw_data_manager*。

3.4 新版功能.

这个类提供了下列对*Policy* 的抽象方法的具体实现:

header_max_count(*name*)
返回用来表示具有给定名称的标头的专用类的*max_count* 属性的值。

header_source_parse(*sourcelines*)
此名称会被作为到 ':' 止的所有内容来解析。该值是通过从第一行的剩余部分去除前导空格，再将所有后续行连接到一起，并去除所有末尾回车符或换行符来确定的。

header_store_parse(*name*, *value*)
name 将会被原样返回。如果输入值具有 name 属性并可在忽略大小写的情况下匹配 *name*，则 value 也会被原样返回。在其他情况下 *name* 和 *value* 会被传递给 header_factory，并将结果标头对象作为值返回。在此情况下如果输入值包含 CR 或 LF 字符则会引发 ValueError。

header_fetch_parse(*name*, *value*)
如果值具有 name 属性，它会被原样返回。在其他情况下 *name* 和移除了所有 CR 和 LF 字符的 *value* 会被传递给 header_factory，并返回结果标头对象。任何经替代转义的字节串会被转换为 unicode 未知字符字形。

fold(*name*, *value*)
标头折叠是由*refold_source* 策略设置来控制的。当且仅当一个值没有 name 属性（具有 name 属性就意味着它是某种标头对象）它才会被当作是 "源值"。如果一个原值需要按照策略来重新折叠，则会通过将 *name* 和去除了所有 CR 和 LF 字符的 *value* 传递给 header_factory 来将其转换为标头对象。标头对象的折叠是通过调用其 fold 方法并附带当前策略来完成的。

源值会使用*splitlines()* 来拆分成多行。如果该值不被重新折叠，则会使用策略中的 linesep 重新合并这些行并将其返回。例外的是包含非 ascii 二进制数据的行。在此情况下无论 refold_source 如何设置该值都会被重新折叠，这会导致二进制数据使用 unknown-8bit 字符集进行 CTE 编码。

fold_binary(*name*, *value*)
如果*cte_type* 为 7bit 则与*fold()* 类似，不同之处在于返回的值是字节串。

如果*cte_type* 为 8bit，则将非 ASCII 二进制数据转换回字节串。带有二进制数据的标头不会被重新折叠，无论 refold_header 设置如何，因为无法知晓该二进制数据是由单字节字符还是多字节字符组成的。

以下*EmailPolicy* 的实例提供了适用于特定应用领域的默认值。请注意在未来这些实例（特别是 HTTP 实例）的行为可能会被调整以便更严格地遵循与其领域相关的 RFC。

email.policy.**default**
一个未改变任何默认值的 EmailPolicy 实例。此策略使用标准的 Python \n 行结束符而非遵循 RFC 的 \r\n。

email.policy.**SMTP**
适用于按照符合电子邮件 RFC 的方式来序列化消息。与 default 类似，但 linesep 被设为遵循 RFC 的 \r\n。

email.policy.**SMTPUTF8**
与 SMTP 类似但是*utf8* 为 True。适用于在不使用标头内已编码字的情况下对消息进行序列化。如果发送方或接收方地址具有非 ASCII 字符则应当只被用于 SMTP 传输 (*smtplib.SMTP.send_message()* 方法会自动如此处理)。

email.policy.**HTTP**
适用于序列化标头以在 HTTP 通信中使用。与 SMTP 类似但是 max_line_length 被设为 None (无限制)。

email.policy.**strict**
便捷实例。与 default 类似但是 raise_on_defect 被设为 True。这样可以允许通过以下写法来严格地设置任何策略:

```
somepolicy + policy.strict
```

因为所有这些*EmailPolicies*，email 包的高效 API 相比 Python 3.2 API 发生了以下几方面变化:

- 在*Message* 中设置标头将使得该标头被解析并创建一个标头对象。
- 从*Message* 提取标头将使得该标头被解析并创建和返回一个标头对象。
- 任何标头对象或任何由于策略设置而被重新折叠的标头都会使用一种完全实现了 RFC 折叠算法的算法来进行折叠，包括知道在休息需要并允许已编码字。

从应用程序的视角来看，这意味着任何通过*EmailMessage* 获得的标头都是具有附加属性的标头对象，其字符串值都是该标头的完全解码后的 unicode 值。类似地，可以使用 unicode 对象为一个标头赋予新的值，或创建一个新的标头对象，并且该策略将负责把该 unicode 字符串转换为正确的 RFC 已编码形式。

标头对象及其属性的描述见*headerregistry*。

class email.policy.**Compat32**(**kw*)
这个实体*Policy* 为向下兼容策略。它复制了 Python 3.2 中 email 包的行为。*policy* 模块还定义了该类的一个实例*compat32*，用来作为默认策略。因此 email 包的默认行为会保持与 Python 3.2 的兼容性。

下列属性具有与*Policy* 默认值不同的值:

mangle_from_
默认值为 True。

这个类提供了下列对*Policy* 的抽象方法的具体实现:

header_source_parse(*sourcelines*)
此名称会被作为到‘:’止的所有内容来解析。该值是通过从第一行的剩余部分去除前导空格，再将所有后续行连接到一起，并去除所有末尾回车符或换行符来确定的。

header_store_parse(*name*, *value*)
name 和 value 会被原样返回。

header_fetch_parse(*name*, *value*)
如果 value 包含二进制数据，则会使用 unknown-8bit 字符集来将其转换为*Header* 对象。在其他情况下它会被原样返回。

fold(*name*, *value*)
标头会使用*Header* 折叠算法进行折叠，该算法保留 value 中现有的换行，并将结果行长度折叠至 max_line_length。非 ASCII 二进制数据会使用 unknown-8bit 字符串进行 CTE 编码。

fold_binary(*name*, *value*)
标头会使用*Header* 折叠算法进行折叠，该算法保留 value 中现有的换行，并将每个结果行的长度折叠至 max_line_length。如果 cte_type 为 7bit，则非 ascii 二进制数据会使用 unknown-8bit 字符集进行 CTE 编码。在其他情况下则会使用原始的源标头，这将保留其现有的换行和所包含的任何（不符合 RFC 的）二进制数据。

email.policy.**compat32**
Compat32 的实例，提供与 Python 3.2 中的 email 包行为的向下兼容性。

备注

19.1.5 email.errors: 异常和缺陷类

源代码: Lib/email/errors.py

以下异常类在*email.errors* 模块中定义：

exception email.errors.**MessageError**
这是*email* 包可以引发的所有异常的基类。它源自标准异常*Exception* 类，这个类没有定义其他方法。

exception email.errors.**MessageParseError**
这是由*Parser* 类引发的异常的基类。它派生自*MessageError*。*headerregistry* 使用的解析器也在内部使用这个类。

exception email.errors.**HeaderParseError**
在解析消息的 **RFC 5322** 标头时，某些错误条件下会触发，此类派生自*MessageParseError*。如果在调用方法时内容类型未知，则*set_boundary()* 方法将引发此错误。当尝试创建一个看起来包含嵌入式标头的标头时*Header* 可能会针对某些 base64 解码错误引发此错误（也就是说，应该是一个没有前导空格并且看起来像标题的延续行）。

exception email.errors.**BoundaryError**
已弃用和不再使用的。

exception email.errors.**MultipartConversionError**
当使用 add_payload() 将有效负载添加到*Message* 对象时，有效负载已经是一个标量，而消息的 *Content-Type* 主类型不是 *multipart* 或者缺少时触发该异常。*MultipartConversionError* 多重继承自*MessageError* 和内置的*TypeError*。

由于 Message.add_payload() 已被弃用，此异常实际上极少会被引发。但是如果在派生自*MIMENonMultipart* 的类 (例如*MIMEImage*) 的实例上调用*attach()* 方法也可以引发此异常。

以下是*FeedParser* 在解析消息时可发现的缺陷列表。请注意这些缺陷会在问题被发现时加入到消息中，因此举例来说，如果某条嵌套在 *multipart/alternative* 中的消息具有错误的标头，该嵌套消息对象就会有一条缺陷，但外层消息对象则没有。

所有缺陷类都是 email.errors.MessageDefect 的子类。

- NoBoundaryInMultipartDefect –一条消息宣称有多个部分，但却没有 *boundary* 形参。
- StartBoundaryNotFoundDefect –在 *Content-Type* 标头中宣称的开始边界无法被找到。
- CloseBoundaryNotFoundDefect –找到了开始边界，但相应的结束边界无法被找到。

  3.3 新版功能.
- FirstHeaderLineIsContinuationDefect –消息以一个继续行作为其第一个标头行。
- MisplacedEnvelopeHeaderDefect - 在标头块中间发现了一个“Unix From”标头。
- MissingHeaderBodySeparatorDefect - 在解析没有前缀空格但又不包含‘:’的标头期间找到一行内容。解析将假定该行表示消息体的第一行以继续执行。

  3.3 新版功能.

- `MalformedHeaderDefect` –找到一个缺失了冒号或格式错误的标头。

  3.3 版后已移除: 此缺陷在近几个 Python 版本中已不再使用。

- `MultipartInvariantViolationDefect` –A message claimed to be a *multipart*, but no subparts were found. Note that when a message has this defect, its *is_multipart()* method may return false even though its content type claims to be *multipart*.

- `InvalidBase64PaddingDefect` –当解码一个 base64 编码的字节分块时，填充的数据不正确。虽然添加了足够的填充数据以执行解码，但作为结果的已解码字节串可能无效。

- `InvalidBase64CharactersDefect` –当解码一个 base64 编码的字节分块时，遇到了 base64 字符表以外的字符。这些字符会被忽略，但作为结果的已解码字节串可能无效。

- `InvalidBase64LengthDefect` –当解码一个 base64 编码的字节分块时，非填充 base64 字符的数量无效（比 4 的倍数多 1）。已编码分块会保持原样。

19.1.6 email.headerregistry: 自定义标头对象

源代码: Lib/email/headerregistry.py

3.6 新版功能:[1]

标头是由*str* 的自定义子类来表示的。用于表示给定标头的特定类则由创建标头时生效的*policy* 的*header_factory* 确定。这一节记录了 email 包为处理兼容 **RFC 5322** 的电子邮件消息所实现的特定 `header_factory`，它不仅为各种标头类型提供了自定义的标头对象，还为应用程序提供了添加其自定义标头类型的扩展机制。

当使用派生自*EmailPolicy* 的任何策略对象时，所有标头都通过*HeaderRegistry* 产生并且以*BaseHeader* 作为其最后一个基类。每个标头类都有一个由该标头类型确定的附加基类。例如，许多标头都以*UnstructuredHeader* 类作为其另一个基类。一个标头专用的第二个类是由标头名称使用存储在*HeaderRegistry* 中的查找表来确定的。所有这些都针对典型应用程序进行透明的管理，但也为修改默认行为提供了接口，以便由更复杂的应用使用。

以下各节首先记录了标头基类及其属性，然后是用于修改*HeaderRegistry* 行为的 API，最后是用于表示从结构化标头解析的数据的支持类。

class email.headerregistry.**BaseHeader**(*name*, *value*)

name 和 *value* 会从*header_factory* 调用传递给 `BaseHeader`。任何标头对象的字符串值都是完成解码为 unicode 的 *value*。

这个基类定义了下列只读属性:

name

标头的名称（字段在 ':' 之前的部分）。这就是 *name* 的*header_factory* 调用所传递的值；也就是说会保持大小写形式。

defects

一个包含 `HeaderDefect` 实例的元组，这些实例报告了在解析期间发现的任何 RFC 合规性问题。email 包会尝试尽可能地检测合规性问题。请参阅*errors* 模块了解可能被报告的缺陷类型的相关讨论。

max_count

此类型标头可具有相同 `name` 的最大数量。`None` 值表示无限制。此属性的 `BaseHeader` 值为 `None`；专用的标头类预期将根据需要重载这个值。

`BaseHeader` 还提供了以下方法，它由 email 库代码调用，通常不应当由应用程序来调用。

[1] 最初在 3.3 中作为暂定模块 添加

fold(*, *policy*)
返回一个字符串，其中包含用来根据 *policy* 正确地折叠标头的 *linesep* 字符。*cte_type* 为 8bit 时将被作为 7bit 来处理，因为标头不能包含任意二进制数据。如果 *utf8* 为 False，则非 ASCII 数据将根据 RFC 2047 来编码。

BaseHeader 本身不能被用于创建标头对象。它定义了一个与每个专用标头相配合的协议以便生成标头对象。具体来说，BaseHeader 要求专用类提供一个名为 parse 的 *classmethod()*。此方法的调用形式如下:

```
parse(string, kwds)
```

kwds 是包含了一个预初始化键 defects 的字典。defects 是一个空列表。parse 方法应当将任何已检测到的缺陷添加到此列表中。在返回时，kwds 字典 *必须* 至少包含 decoded 和 defects 等键的值。decoded 应当是标头的字符串值（即完全解码为 unicode 的标头值）。parse 方法应当假定 *string* 可能包含 content-transfer-encoded 部分，但也应当正确地处理全部有效的 unicode 字符以便它能解析未经编码的标头值。

随后 BaseHeader 的 __new__ 会创建标头实例，并调用其 init 方法。专属类如果想要设置 BaseHeader 自身所提供的属性之外的附加属性，只需提供一个 init 方法。这样的 init 看起来应该是这样:

```
def init(self, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

也就是说，专属类放入 kwds 字典的任何额外内容都应当被移除和处理，并且 kw (和 args) 的剩余内容会被传递给 BaseHeader init 方法。

class email.headerregistry.**UnstructuredHeader**
“非结构化”标头是 RFC 5322 中默认的标头类型。任何没有指定语法的标头都会被视为是非结构化的。非结构化标头的经典例子是 *Subject* 标头。

在 RFC 5322 中，非结构化标头是指一段以 ASCII 字符集表示的任意文本。但是 RFC 2047 具有一个 RFC 5322 兼容机制用来将标头值中的非 ASCII 文本编码为 ASCII 字符。当包含已编码字的 *value* 被传递给构造器时，UnstructuredHeader 解析器会按照非结构化文本的 RFC 2047 规则将此类已编码字转换为 unicode。解析器会使用启发式机制来尝试解码一些不合规的已编码字。在此种情况下各类缺陷，例如已编码字或未编码文本中的无效字符问题等缺陷将会被注册。

此标头类型未提供附加属性。

class email.headerregistry.**DateHeader**
RFC 5322 为电子邮件标头内的日期指定了非常明确的格式。DateHeader 解析器会识别该日期格式，并且也能识别间或出现的一些“不规范”变种形式。

这个标头类型提供了以下附加属性。

datetime
如果标头值能被识别为某一种有效的日期形式，此属性将包含一个代表该日期的 *datetime* 实例。如果输入日期的时区被指定为 -0000 (表示为 UTC 但不包含源时区的相关信息)，则 *datetime* 将为简单型 *datetime*。如果找到了特定的时区时差值 (包括 *+0000*)，则 *datetime* 将包含使用 *datetime.timezone* 来记录时区时差时的感知型 datetime。

标头的 decoded 值是由按照 RFC 5322 对 datetime 进行格式化来确定的；也就是说，它会被设为:

```
email.utils.format_datetime(self.datetime)
```

当创建 DateHeader 时，*value* 可以为 *datetime* 实例。例如这意味着以下代码是有效的并能实现人们预期的行为:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

因为这是个简单型 datetime 它将被解读为 UTC 时间戳，并且结果值的时区将为 -0000。使用来自*utils* 模块的*localtime()* 函数会更有用:

```
msg['Date'] = utils.localtime()
```

这个例子将日期标头设为使用当前时区时差值的当前时间和日期。

class email.headerregistry.**AddressHeader**

地址标头是最复杂的结构化标头类型之一。AddressHeader 类提供了适合任何地址标头的泛用型接口。

这个标头类型提供了以下附加属性。

groups

编码了在标头值中找到的地址和分组的*Group* 对象的元组。非分组成员的地址在此列表中表示为*display_name* 为 None 的单地址 Groups。

addresses

编码了来自标头值的所有单独地址的*Address* 对象的元组。如果标头值包含任何分组，则来自分组的单个地址将包含在该分组出现在值中的点上列出（也就是说，地址列表会被“展平”为一维列表）。

标头的 decoded 值将为所有已编码字解码为 unicode 的结果。*idna* 编码的域名也将被解码为 unicode。decoded 值是通过对 groups 属性的元素的*str* 值使用 ', ' 进行*join* 来设置的。

可以使用*Address* 与*Group* 对象的任意组合的列表来设置一个地址标头的值。display_name 为 None 的 Group 对象将被解读为单独地址，这允许一个地址列表可以附带通过使用从源标头的 groups 属性获取的列表而保留原分组。

class email.headerregistry.**SingleAddressHeader**

AddressHeader 的子类，添加了一个额外的属性:

address

由标头值编码的单个地址。如果标头值实际上包含一个以上的地址（这在默认*policy* 下将违反 RFC），则访问此属性将导致*ValueError*。

上述类中许多还具有一个 Unique 变体 (例如 UniqueUnstructuredHeader)。其唯一差别是在 Unique 变体中*max_count* 被设为 1。

class email.headerregistry.**MIMEVersionHeader**

实际上 *MIME-Version* 标头只有一个有效的值，即 1.0。为了将来的扩展，这个标头类还支持其他的有效版本号。如果一个版本号是 **RFC 2045** 的有效值，则标头对象的以下属性将具有不为 None 的值:

version

字符串形式的版本号。任何空格和/或注释都会被移除。

major

整数形式的主版本号

minor

整数形式的次版本号

class email.headerregistry.**ParameterizedMIMEHeader**

MIME 标头都以前缀‘Content-‘打头。每个特定标头都具有特定的值，其描述在该标头的类之中。有些也可以接受一个具有通用格式的补充形参形表。这个类被用作所有接受形参的 MIME 标头的基类。

params

一个将形参名映射到形参值的字典。

class email.headerregistry.**ContentTypeHeader**
处理 *Content-Type* 标头的*ParameterizedMIMEHeader* 类。

content_type
maintype/subtype 形式的内容类型字符串。

maintype

subtype

class email.headerregistry.**ContentDispositionHeader**
处理 *Content-Disposition* 标头的*ParameterizedMIMEHeader* 类。

content-disposition
inline 和 attachment 是仅有的常用有效值。

class email.headerregistry.**ContentTransferEncoding**
处理 *Content-Transfer-Encoding* 标头。

cte
可用的有效值为 7bit, 8bit, base64 和 quoted-printable。更多信息请参阅 **RFC 2045**。

class email.headerregistry.**HeaderRegistry**(*base_class=BaseHeader*, *default_class=UnstructuredHeader*, *use_default_map=True*)
这是由*EmailPolicy* 在默认情况下使用的工厂函数。HeaderRegistry 会使用 *base_class* 和从它所保存的注册表中获取的专用类来构建用于动态地创建标头实例的类。当给定的标头名称未在注册表中出现时，则会使用由 *default_class* 所指定的类作为专用类。当 *use_default_map* 为 True (默认值) 时，则会在初始化期间把将标头名称与类的标准映射拷贝到注册表中。*base_class* 始终会是所生成类的 __bases__ 列表中的最后一个类。

默认的映射有:

subject UniqueUnstructuredHeader

date UniqueDateHeader

resent-date DateHeader

orig-date UniqueDateHeader

sender UniqueSingleAddressHeader

resent-sender SingleAddressHeader

到 UniqueAddressHeader

resent-to AddressHeader

cc UniqueAddressHeader

resent-cc AddressHeader

从 UniqueAddressHeader

resent-from AddressHeader

reply-to UniqueAddressHeader

HeaderRegistry 具有下列方法:

map_to_type(*self*, *name*, *cls*)
name 是要映射的标头名称。它将在注册表中被转换为小写形式。*cls* 是要与 *base_class* 一起被用来创建用于实例化与 *name* 相匹配的标头的类的专用类。

__getitem__(*name*)
构造并返回一个类来处理 *name* 标头的创建。

__call__(*name*, *value*)
从注册表获得与 *name* 相关联的专用标头 (如果 *name* 未在注册表中出现则使用 *default_class*) 并将其与 *base_class* 相组合以产生类，调用被构造类的构造器，传入相同的参数列表，并最终返回由此创建的类实例。

以下的类是用于表示从结构化标头解析的数据的类，并且通常会由应用程序使用以构造结构化的值并赋给特定的标头。

class email.headerregistry.**Address**(*display_name=''*, *username=''*, *domain=''*, *addr_spec=None*)
用于表示电子邮件地址的类。地址的一般形式为:

```
[display_name] <username@domain>
```

或者:

```
username@domain
```

其中每个部分都必须符合在 **RFC 5322** 中阐述的特定语法规则。

为了方便起见可以指定 *addr_spec* 来替代 *username* 和 *domain*，在此情况下 *username* 和 *domain* 将从 *addr_spec* 中解析。*addr_spec* 应当是一个正确地引用了 RFC 的字符串；如果它不是 `Address` 则将引发错误。Unicode 字符也允许使用并将在序列化时被正确地编码。但是，根据 RFC，地址的 username 部分 *不允许*有 unicode。

display_name
地址的显示名称部分（如果有的话）并去除所有引用项。如果地址没有显示名称，则此属性将为空字符串。

username
地址的 `username` 部分，去除所有引用项。

domain
地址的 `domain` 部分。

addr_spec
地址的 `username@domain` 部分，经过正确引用处理以作为纯地址使用（上面显示的第二种形式）。此属性不可变。

__str__()
对象的 `str` 值是根据 **RFC 5322** 规则进行引用处理的地址，但不带任何非 ASCII 字符的 Content Transfer Encoding。

为了支持 SMTP (**RFC 5321**)，`Address` 会处理一种特殊情况：如果 `username` 和 `domain` 均为空字符串 (或为 `None`)，则 `Address` 的字符串值为 `<>`。

class email.headerregistry.**Group**(*display_name=None*, *addresses=None*)
用于表示地址组的类。地址组的一般形式为:

```
display_name: [address-list];
```

作为处理由组和单个地址混合构成的列表的便捷方式，`Group` 也可以通过将 *display_name* 设为 `None` 以用来表示不是某个组的一部分的单个地址并提供单个地址的列表作为 *addresses*。

display_name
组的 `display_name`。如果其为 `None` 并且恰好有一个 `Address` 在 `addresses` 中，则 `Group` 表示一个不在某个组中的单独地址。

addresses
一个可能为空的表示组中地址的包含*Address* 对象的元组。

__str__()
Group 的 str 值会根据 **RFC 5322** 进行格式化，但不带任何非 ASCII 字符的 Content Transfer Encoding。如果 display_name 为空值且只有一个单独 Address 在 addresses 列表中，则 str 值将与该单独 Address 的 str 相同。

备注

19.1.7 email.contentmanager: 管理 MIME 内容

源代码: Lib/email/contentmanager.py

3.6 新版功能:[1]

class email.contentmanager.**ContentManager**
内容管理器的基类。提供注册 MIME 内容和其他表示形式间转换器的标准注册机制，以及 get_content 和 set_content 发送方法。

get_content(*msg*, **args*, ***kw*)
基于 *msg* 的 mimetype 查找处理函数（参见下一段），调用该函数，传递所有参数，并返回调用的结果。期望的效果是处理程序将从 *msg* 中提取有效载荷并返回编码了有关被提取数据信息的对象。

要找到处理程序，将在注册表中查找以下键，找到第一个键即停止:

- 表示完整 MIME 类型的字符串 (maintype/subtype)
- 表示 maintype 的字符串
- 空字符串

如果这些键都没有产生处理程序，则为完整 MIME 类型引发一个*KeyError*。

set_content(*msg*, *obj*, **args*, ***kw*)
如果 maintype 为 multipart，则引发*TypeError*；否则基于 *obj* 的类型（参见下一段）查找处理函数，在 *msg* 上调用*clear_content()*，并调用处理函数，传递所有参数。预期的效果是处理程序将转换 *obj* 并存入 *msg*，并可能对 *msg* 进行其他更改，例如添加各种 MIME 标头来编码需要用来解释所存储数据的信息。

要找到处理程序，将获取 *obj* 的类型 (typ = type(obj))，并在注册表中查找以下键，找到第一个键即停止:

- 类型本身 (typ)
- 类型的完整限定名称 (typ.__module__ + '.' + typ.__qualname__)。
- 类型的 qualname (typ.__qualname__)
- 类型的 name (typ.__name__)。

如果未匹配到上述的任何一项，则在*MRO* (typ.__mro__) 中为每个类型重复上述的所有检测。最后，如果没有其他键产生处理程序，则为 None 键检测处理程序。如果也没有 None 的处理程序，则为该类型的完整限定名称引发*KeyError*。

并会添加一个 *MIME-Version* 标头，如果没有的话 (另请参见*MIMEPart*)。

add_get_handler(*key*, *handler*)
将 *handler* 函数记录为 *key* 的处理程序。对于可能的 *key* 键，请参阅*get_content()*。

[1] 最初在 3.4 中作为暂定模块 添加

add_set_handler(*typekey*, *handler*)
将 *handler* 记录为当一个匹配 *typekey* 的类型对象被传递给*set_content()* 时所要调用的函数。对于可能的 *typekey* 值，请参阅*set_content()*。

内容管理器实例

目前 email 包只提供了一个实体内容管理器*raw_data_manager*，不过在未来可能会添加更多。*raw_data_manager* 是由*EmailPolicy* 及其衍生工具所提供的*content_manager*。

email.contentmanager.**raw_data_manager**
这个内容管理器仅提供了超出*Message* 本身提供内容的最小接口：它只处理文本、原始字节串和*Message* 对象。不过相比基础 API，它具有显著的优势：在文本部分上执行 get_content 将返回一个 unicode 字符串而无需由应用程序来手动解码，set_content 为控制添加到一个部分的标头和控制内容传输编码格式提供了丰富的选项集合，并且它还启用了多种 add_ 方法，从而简化了多部分消息的创建过程。

email.contentmanager.**get_content**(*msg*, *errors='replace'*)
将指定部分的有效载荷作为字符串（对于 text 部分），*EmailMessage* 对象（对于 message/rfc822 部分）或 bytes 对象（对于所有其他非多部分类型）返回。如果是在 multipart 上调用则会引发*KeyError*。如果指定部分是一个 text 部分并且指明了 *errors*，则会在将载荷解码为 unicode 时将其用作错误处理程序。默认的错误处理程序是 replace。

email.contentmanager.**set_content**(*msg*, *<'str'>*, *subtype="plain"*, *charset='utf-8'* *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
email.contentmanager.**set_content**(*msg*, *<'bytes'>*, *maintype*, *subtype*, *cte="base64"*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
email.contentmanager.**set_content**(*msg*, *<'EmailMessage'>*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
向 *msg* 添加标头和有效载荷:

添加一个带有 maintype/subtype 值的 *Content-Type* 标头。

- 对于 str，将 MIME maintype 设为 text，如果指定了子类型 *subtype* 则设为指定值，否则设为 plain。
- 对于 bytes，将使用指定的 *maintype* 和 *subtype*，如果未指定则会引发*TypeError*。
- 对于*EmailMessage* 对象，将 maintype 设为 message，并将指定的 subtype 设为 *subtype*，如果未指定则设为 rfc822。如果 *subtype* 为 partial，则引发一个错误（必须使用 bytes 对象来构造 message/partial 部分）。

如果提供了 *charset* (这只对 str 适用)，则使用指定的字符集将字符串编码为字节串。默认值为 utf-8。如果指定的 *charset* 是某个标准 MIME 字符集名称的已知别名，则会改用该标准字符集。

如果设置了 *cte*，则使用指定的内容传输编码格式对有效载荷进行编码，并将 *Content-Transfer-Encoding* 标头设为该值。可能的 *cte* 值有 quoted-printable, base64, 7bit, 8bit 和 binary。如果输入无法以指定的编码格式被编码 (例如，对于包含非 ASCII 值的输入指定 *cte* 值为 7bit)，则会引发*ValueError*。

- 对于 str 对象，如果 *cte* 未设置则会使用启发方式来确定最紧凑的编码格式。
- 对于*EmailMessage*，按照 **RFC 2046**，如果为 *subtype* rfc822 请求的 *cte* 为 quoted-printable 或 base64，而为 7bit 以外的任何 *cte* 为 *subtype* external-body 则会引发一个错误。对于 message/rfc822，如果 *cte* 未指定则会使用 8bit。对于所有其他 *subtype* 值，都会使用 7bit。

注解： *cte* 值为 `binary` 实际上还不能正确工作。由 `set_content` 所修改的 `EmailMessage` 对象是正确的，但 *BytesGenerator* 不会正确地将其序列化。

如果设置了 *disposition*，它会被用作 *Content-Disposition* 标头的值。如果未设置，并且指定了 *filename*，则添加值为 `attachment` 的标头。如果未设置 *disposition* 并且也未指定 *filename*，则不添加标头。*disposition* 的有效值仅有 `attachment` 和 `inline`。

如果设置了 *filename*，则将其用作 *Content-Disposition* 标头的 `filename` 参数的值。

如果设置了 *cid*，则添加一个 *Content-ID* 标头并将其值设为 *cid*。

如果设置了 *params*，则迭代其 `items` 方法并使用输出的 `(key, value)` 结果对在 *Content-Type* 标头上设置附加参数。

如果设置了 *headers* 并且为 `headername: headervalue` 形式的字符串的列表或为 `header` 对象的列表（通过一个 `name` 属性与字符串相区分），则将标头添加到 *msg*。

备注

19.1.8 email: 示例

以下是一些如何使用 *email* 包来读取、写入和发送简单电子邮件以及更复杂的 MIME 邮件的示例。

首先，让我们看看如何创建和发送简单的文本消息（文本内容和地址都可能包含 unicode 字符）：

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
# you == the recipient's email address
msg['Subject'] = 'The contents of %s' % textfile
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

解析 **RFC 822** 标题可以通过使用 *parser* 模块中的类来轻松完成：

```
# Import the email modules we'll need
from email.parser import BytesParser, Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
# with open(messagefile, 'rb') as fp:
```

(下页继续)

(续上页)

```
#     headers = BytesParser(policy=default).parse(fp)

#  Or for parsing headers in a string (this is an uncommon operation), use:
headers = Parser(policy=default).parsestr(
        'From: Foo Bar <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

以下是如何发送包含可能在目录中的一系列家庭照片的 MIME 消息示例：

```
# Import smtplib for the actual sending function
import smtplib

# And imghdr to find the types of our images
import imghdr

# Here are the email package modules we'll need
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'Our family reunion'

# Open the files in binary mode.  Use imghdr to figure out the
# MIME subtype for each specific image.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
                                 subtype=imghdr.what(None, img_data))

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

以下是如何将目录的全部内容作为电子邮件消息发送的示例：[1]

```
#!/usr/bin/env python3
```

(下页继续)

[1] 感谢 Matthew Dixon Cowles 提供最初的灵感和示例。

(续上页)

```
"""Send the contents of a directory as a MIME message."""

import os
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP


def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the specified directory,
                        otherwise use the current directory.  Only the regular
                        files in the directory are sent, and we don't recurse to
                        subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
                        sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')
    args = parser.parse_args()
    directory = args.directory
    if not directory:
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = 'Contents of directory %s' % os.path.abspath(directory)
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
```

(下页继续)

(续上页)

```
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                               maintype=maintype,
                               subtype=subtype,
                               filename=filename)
    # Now send or store the message
    if args.output:
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)


if __name__ == '__main__':
    main()
```

以下是如何将上述 MIME 消息解压缩到文件目录中的示例：

```
#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import email
import mimetypes

from email.policy import default

from argparse import ArgumentParser


def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
```

(下页继续)

(续上页)

```
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = 'part-%03d%s' % (counter, ext)
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))


if __name__ == '__main__':
    main()
```

以下是如何使用备用纯文本版本创建 HTML 消息的示例。为了让事情变得更有趣，我们在 html 部分中包含了一个相关的图像，我们保存了一份我们要发送的内容到硬盘中，然后发送它。

```
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Ayons asperges pour le déjeuner"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
             Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
Salut!

Cela ressemble à un excellent recipie[1] déjeuner.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")

# Add the html version.  This converts the message into a multipart/alternative
# container, with the original text message as the first part and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
```

(下页继续)

(续上页)

```
    <p>Cela ressemble à un excellent
        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
            recipie
        </a> déjeuner.
    </p>
    <img src="cid:{asparagus_cid}" />
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# Send the message via local SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

如果我们发送最后一个示例中的消息，这是我们可以处理它的一种方法：

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser

# An imaginary module that would make this work and be safe.
from imaginary import magic_html_parser

# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will
# be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can extract whatever
# the least formatted payload is and print the first three lines.  Of course,
# if the message has no plain text part printing the first three lines of html
# is probably useless, but this is just a conceptual example.
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))
```

(下页继续)

(续上页)

```
ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to display it:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest.get_content_type()))
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())[1]
        else:
            extension = mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
            f.write(part.get_content())
            # again strip the <> to go from email form of cid to html form.
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    # The magic_html_parser has to rewrite the href="cid:...." attributes to
    # point to the filenames in partfiles.  It also has to do a safety-sanitize
    # of the html.  It could be written using html.parser.
    f.write(magic_html_parser(body.get_content(), partfiles))
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# Of course, there are lots of email messages that could break this simple
# minded program, but it will handle the most common ones.
```

直到输出提示，上面的输出是：

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Ayons asperges pour le déjeuner

Salut!

Cela ressemble à un excellent recipie[1] déjeuner.
```

备注

旧式 API:

19.1.9 email.message.Message: 使用 compat32 API 来表示电子邮件消息

Message 类与*EmailMessage* 类非常相似，但没有该类所添加的方法，并且某些方法的默认行为也略有不同。我们还在这里记录了一些虽然被*EmailMessage* 类所支持但并不推荐的方法，除非你是在处理旧有代码。

在其他情况下这两个类的理念和结构都是相同的。

本文档描述了默认 (对于*Message*) 策略*Compat32* 之下的行为。如果你要使用其他策略，你应当改用*EmailMessage* 类。

电子邮件消息是由多个 标头和一个 *payload* 载荷组成的。标头必须为 **RFC 5233** 风格的名称和值，其中字段名和值由冒号分隔。冒号不是字段名或字段值的组成部分。载荷可以是简单的文本消息，或是二进制对象，或是多个子消息的结构化序列，每个子消息都有自己的标头集合和自己的载荷。后一种类型的载荷是由具有 *multipart/** 或 *message/rfc822* 等 MIME 类型的消息来指明的。

Message 对象所提供了概念化模型是由标头组成的有序字典，加上用于访问标头中的特殊信息以及访问载荷的额外方法，以便能生成消息的序列化版本，并递归地遍历对象树。请注意重复的标头是受支持的，但必须使用特殊的方法来访问它们。

Message 伪字典以标头名作为索引，标头名必须为 ASCII 值。字典的值为应当只包含 ASCII 字符的字符串；对于非 ASCII 输入有一些特殊处理，但这并不总能产生正确的结果。标头以保留原大小写的形式存储和返回，但字段名称匹配对大小写不敏感。还可能会有一个单独的封包标头，也称 *Unix-From* 标头或 `From_` 标头。载荷对于简单消息对象的情况是一个字符串或字节串，对于 MIME 容器文档的情况 (例如 *multipart/** 和 *message/rfc822*) 则是一个*Message* 对象。

以下是*Message* 类的方法:

class email.message.**Message**(*policy=compat32*)

如果指定了 *policy* (它必须为*policy* 类的实例) 则使用它所设置的规则来更新和序列化消息的表示形式。如果未设置 *policy*，则使用*compat32* 策略，该策略会保持对 Python 3.2 版 email 包的向下兼容性。更多信息请参阅*policy* 文档。

在 3.3 版更改: 增加了 *policy* 关键字参数。

as_string(*unixfrom=False*, *maxheaderlen=0*, *policy=None*)

以展平的字符串形式返回整个消息对象。或可选的 *unixfrom* 为真值，返回的字符串会包括封包标头。*unixfrom* 的默认值是 `False`。出于保持向下兼容性的原因，*maxheaderlen* 的默认值是 `0`，因此如果你想要不同的值你必须显式地重载它（在策略中为 *max_line_length* 指定的值将被此方法忽略）。*policy* 参数可被用于覆盖从消息实例获取的默认策略。这可以用来对该方法所输出的格式进行一些控制，因为指定的 *policy* 将被传递给 `Generator`。

如果需要填充默认值以完成对字符串的转换则展平消息可能触发对*Message* 的修改（例如，MIME 边界可能会被生成或被修改）。

请注意此方法是出于便捷原因提供的，可能无法总是以你想要的方式格式化消息。例如，在默认情况下它不会按 unix mbox 格式的要求对以 `From` 打头的行执行调整。为了获得更高灵活性，请实例化一个*Generator* 实例并直接使用其*flatten()* 方法。例如:

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

如果消息对象包含未按照 RFC 标准进行编码的二进制数据，则这些不合规数据将被 unicode "unknown character" 码位值所替代。(另请参阅*as_bytes()* 和*BytesGenerator*。)

在 3.4 版更改: 增加了 *policy* 关键字参数。

__str__()
与*as_string()* 等价。这将让 str(msg) 产生一个包含已格式化消息的字符号。

as_bytes(*unixfrom=False*, *policy=None*)
以字节串对象的形式返回整个扁平化后的消息。当可选的 *unixfrom* 为真值时，返回的字符串会包括封包标头。*unixfrom* 的默认值为 False。*policy* 参数可被用于重载从消息实例获取的默认策略。这可被用来控制该方法所产生的部分格式化效果，因为指定的 *policy* 将被传递给 BytesGenerator。

如果需要填充默认值以完成对字符串的转换则展平消息可能触发对*Message* 的修改（例如，MIME 边界可能会被生成或被修改)。

请注意此方法是出于便捷原因提供的，可能无法总是以你想要的方式格式化消息。例如，在默认情况下它不会按 unix mbox 格式的要求对以 From 打头的行执行调整。为了获得更高灵活性，请实例化一个*BytesGenerator* 实例并直接使用其*flatten()* 方法。例如:

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

3.4 新版功能.

__bytes__()
与*as_bytes()* 等价。这将让 bytes(msg) 产生一个包含已格式化消息的字节串对象。

3.4 新版功能.

is_multipart()
如果该消息的载荷是一个子*Message* 对象列表则返回 True，否则返回 False。当*is_multipart()* 返回 False 时，载荷应当是一个字符串对象（有可能是一个 CTE 编码的二进制载荷)。(请注意*is_multipart()* 返回 True 并不意味着 "msg.get_content_maintype() == 'multipart'" 将返回 True。例如，is_multipart 在*Message* 类型为 message/rfc822 时也将返回 True。)

set_unixfrom(*unixfrom*)
将消息的封包标头设为 *unixfrom*，这应当是一个字符串。

get_unixfrom()
返回消息的信封头。如果信封头从未被设置过，默认返回 None 。

attach(*payload*)
将给定的 *payload* 添加到当前载荷中，当前载荷在该调用之前必须为 None 或是一个*Message* 对象列表。在调用之后，此载荷将总是一个*Message* 对象列表。如果你想将此载荷设为一个标量对象（如字符串)，请改用*set_payload()*。

这是一个过时的方法。在 EmailMessage 类上它的功能已被*set_content()* 及相应的 make 和 add 方法所替代。

get_payload(*i=None*, *decode=False*)
返回当前的载荷，它在*is_multipart()* 为 True 时将是一个*Message* 对象列表，在*is_multipart()* 为 False 时则是一个字符串。如果该载荷是一个列表且你修改了这个列表对象，那么你就是原地修改了消息的载荷。

传入可选参数 *i* 时，如果*is_multipart()* 为 True，*get_payload()* 将返回载荷从零开始计数的第 *i* 个元素。如果 *i* 小于 0 或大于等于载荷中的条目数则将引发*IndexError*。如果载荷是一个字符串 (即*is_multipart()* 为 False) 且给出了 *i*，则会引发*TypeError*。

可选的 *decode* 是一个指明载荷是否应根据 *Content-Transfer-Encoding* 标头被解码的旗标。当其值为 True 且消息没有多个部分时，如果此标头值为 quoted-printable 或 base64 则载荷将被解码。如果使用了其他编码格式，或者找不到 *Content-Transfer-Encoding* 标头时，载荷将被原样返回（不编码）。在所有情况下返回值都是二进制数据。如果消息有多个部分且 *decode* 旗标为 True，则将返回 None。如果载荷为 base64 但内容不完全正确（如缺少填充符、存在 base64 字母表以外的字符等），则将在消息的缺陷属性中添加适当的缺陷值 (分别为 InvalidBase64PaddingDefect 或 InvalidBase64CharactersDefect)。

当 *decode* 为 False (默认值) 时消息体会作为字符串返回而不解码 *Content-Transfer-Encoding*。但是，对于 *Content-Transfer-Encoding* 为 8bit 的情况，会尝试使用 *Content-Type* 标头指定的 charset 来解码原始字节串，并使用 replace 错误处理程序。如果未指定 charset，或者如果指定的 charset 未被 email 包所识别，则会使用默认的 ASCII 字符集来解码消息体。

这是一个过时的方法。在 EmailMessage 类上它的功能已被*get_content()* 和*iter_parts()* 方法所替代。

set_payload(*payload*, *charset=None*)

将整个消息对象的载荷设为 *payload*。客户端要负责确保载荷的不变性。可选的 *charset* 用于设置消息的默认字符集；详情请参阅*set_charset()*。

这是一个过时的方法。在 EmailMessage 类上它的功能已被*set_content()* 方法所替代。

set_charset(*charset*)

将载荷的字符集设为 *charset*，它可以是*Charset* 实例 (参见*email.charset*)、字符集名称字符串或 None。如果是字符串，它将被转换为一个*Charset* 实例。如果 *charset* 是 None，charset 形参将从 *Content-Type* 标头中被删除（消息将不会进行其他修改）。任何其他值都将导致*TypeError*。

如果 *MIME-Version* 标头不存在则将被添加。如果 *Content-Type* 标头不存在，则将添加一个值为 *text/plain* 的该标头。无论 *Content-Type* 标头是否已存在，其 charset 形参都将被设为 *charset.output_charset*。如果 *charset.input_charset* 和 *charset.output_charset* 不同，则载荷将被重编码为 *output_charset*。如果 *Content-Transfer-Encoding* 标头不存在，则载荷将在必要时使用指定的*Charset* 来转换编码，并将添加一个具有相应值的标头。如果 *Content-Transfer-Encoding* 标头已存在，则会假定载荷已使用该 *Content-Transfer-Encoding* 进行正确编码并不会再被修改。

这是一个过时的方法。在 EmailMessage 类上它的功能已被 email.emailmessage.EmailMessage.set_content() 方法的 *charset* 形参所替代。

get_charset()

返回与消息的载荷相关联的*Charset* 实例。

这是一个过时的方法。在 EmailMessage 类上它将总是返回 None。

以下方法实现了用于访问消息的 **RFC 2822** 标头的类映射接口。请注意这些方法和普通映射（例如字典）接口之间存在一些语义上的不同。举例来说，在一个字典中不能有重复的键，但消息标头则可能有重复。并且，在字典中由*keys()* 返回的键的顺序是没有保证的，但在*Message* 对象中，标头总是会按它们在原始消息中的出现或后继加入顺序返回。任何已删除再重新加入的标头总是会添加到标头列表的末尾。

这些语义上的差异是有意为之且其目的是为了提供最大的便利性。

还请留意，无论在什么情况下，消息当中的任何信封头字段都不会包含在映射接口当中。

在由字节串生成的模型中，任何包含非 ASCII 字节数据（违反 RFC）的标头值当通过此接口来获取时，将被表示为使用 *unknown-8bit* 字符集的*Header* 对象。

__len__()
返回头字段的总数，重复的也计算在内。

__contains__(*name*)
Return true if the message object has a field named *name*. Matching is done case-insensitively and *name* should not include the trailing colon. Used for the `in` operator, e.g.:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)
返回指定名称标头字段的值。*name* 不应包括作为字段分隔符的冒号。如果标头未找到，则返回 None；*KeyError* 永远不会被引发。

请注意如果指定名称的字段在消息标头中多次出现，具体将返回哪个字段值是未定义的。请使用*get_all()* 方法来获取所有指定名称标头的值。

__setitem__(*name*, *val*)
将具有字段名 *name* 和值 *val* 的标头添加到消息中。字段会被添加到消息的现有字段的末尾。

请注意，这个方法 既不会覆盖 也不会删除任何字段名重名的已有字段。如果你确实想保证新字段是整个信息头当中唯一拥有 *name* 字段名的字段，你需要先把旧字段删除。例如：

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)
删除信息头当中字段名匹配 *name* 的所有字段。如果匹配指定名称的字段没有找到，也不会抛出任何异常。

keys()
以列表形式返回消息头中所有的字段名。

values()
以列表形式返回消息头中所有的字段值。

items()
以二元元组的列表形式返回消息头中所有的字段名和字段值。

get(*name*, *failobj=None*)
返回指定名称标头字段的值。这与*__getitem__()* 是一样的，不同之处在于如果指定名称标头未找到则会返回可选的 *failobj* (默认为 None)。

以下是一些有用的附加方法:

get_all(*name*, *failobj=None*)
返回字段名为 *name* 的所有字段值的列表。如果信息内不存在匹配的字段，返回 *failobj*（其默认值为 None ）。

add_header(*_name*, *_value*, ***_params*)
高级头字段设定。这个方法与*__setitem__()* 类似，不过你可以使用关键字参数为字段提供附加参数。*_name* 是字段名，*_value* 是字段 主值。

对于关键字参数字典 *_params* 中的每一项，其键会被当作形参名，并执行下划线和连字符间的转换（因为连字符不是合法的 Python 标识符）。通常，形参将以 `key="value"` 的形式添加，除非值为 None，在这种情况下将只添加键。如果值包含非 ASCII 字符，可将其指定为格式为 `(CHARSET, LANGUAGE, VALUE)` 的三元组，其中 `CHARSET` 为要用来编码值的字符集名称字符串，`LANGUAGE` 通常可设为 None 或空字符串（请参阅 **RFC 2231** 了解其他可能的取值），而 `VALUE` 为包含非 ASCII 码位的字符串值。如果不是传入一个三元组且值包含非 ASCII 字符，则会自动以 **RFC 2231** 格式使用 `CHARSET` 为 `utf-8` 和 `LANGUAGE` 为 None 对其进行编码。

以下是为示例代码:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

会添加一个形如下文的头字段:

```
Content-Disposition: attachment; filename="bud.gif"
```

使用非 ASCII 字符的示例代码:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

它的输出结果为

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)
替换一个标头。将替换在匹配 *_name* 的消息中找到的第一个标头，标头顺序和字段名大小写保持不变。如果未找到匹配的标头，则会引发*KeyError*。

get_content_type()
返回消息的内容类型。返回的字符串会强制转换为 *maintype/subtype* 的全小写形式。如果消息中没有 *Content-Type* 标头则将返回由*get_default_type()* 给出的默认类型。因为根据 **RFC 2045**，消息总是要有一个默认类型，所以*get_content_type()* 将总是返回一个值。

RFC 2045 将消息的默认类型定义为 *text/plain*，除非它是出现在 *multipart/digest* 容器内，在这种情况下其类型应为 *message/rfc822*。如果 *Content-Type* 标头指定了无效的类型，**RFC 2045** 规定其默认类型应为 *text/plain*。

get_content_maintype()
返回信息的主要内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *maintype* 部分。

get_content_subtype()
返回信息的子内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *subtype* 部分。

get_default_type()
返回默认的内容类型。绝大多数的信息，其默认内容类型都是 *text/plain* 。作为 *multipart/digest* 容器内子部分的信息除外，它们的默认内容类型是 *message/rfc822* 。

set_default_type(*ctype*)
设置默认的内容类型。*ctype* 应当为 *text/plain* 或者 *message/rfc822*，尽管这并非强制。默认的内容类型不会存储在 *Content-Type* 标头中。

get_params(*failobj=None*, *header='content-type'*, *unquote=True*)
将消息的 *Content-Type* 形参作为列表返回。所返回列表的元素为以 '=' 号拆分出的键/值对 2 元组。'=' 左侧的为键，右侧的为值。如果形参值中没有 '=' 号，否则该将值如*get_param()* 描述并且在可选 *unquote* 为 True (默认值) 时会被取消转义。

可选的 *failobj* 是在没有 *Content-Type* 标头时要返回的对象。可选的 *header* 是要替代 *Content-Type* 被搜索的标头。

这是一个过时的方法。在 EmailMessage 类上它的功能已被标头访问方法所返回的单独标头对象的 *params* 特征属性所替代。

get_param(*param*, *failobj=None*, *header='content-type'*, *unquote=True*)
将 *Content-Type* 标头的形参 *param* 作为字符串返回。如果消息没有 *Content-Type* 标头或者没有这样的形参，则返回 *failobj* (默认为 None)。

如果给出可选的 *header*，它会指定要替代 *Content-Type* 来使用的消息标头。

形参的键总是以大小写不敏感的方式来比较的。返回值可以是一个字符串，或者如果形参以 **RFC 2231** 编码则是一个 3 元组。当为 3 元组时，值中的元素采用 (CHARSET, LANGUAGE, VALUE) 的形式。请注意 CHARSET 和 LANGUAGE 都可以为 None，在此情况下你应当将 VALUE 当作以 us-ascii 字符集来编码。你可以总是忽略 LANGUAGE。

如果你的应用不关心形参是否以 **RFC 2231** 来编码，你可以通过调用*email.utils.collapse_rfc2231_value()* 来展平形参值，传入来自*get_param()* 的返回值。当值为元组时这将返回一个经适当编码的 Unicode 字符串，否则返回未经转义的原字符串。例如:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

无论在哪种情况下，形参值（或为返回的字符串，或为 3 元组形式的 VALUE 条目）总是未经转换的，除非 *unquote* 被设为 False。

这是一个过时的方法。在 EmailMessage 类上它的功能已被标头访问方法所返回的单独标头对象的 *params* 特征属性所替代。

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)

在 *Content-Type* 标头中设置一个形参。如果该形参已存在于标头中，它的值将被替换为 *value*。如果此消息还未定义 *Content-Type* 标头，它将被设为 *text/plain* 且新的形参值将按 **RFC 2045** 的要求添加。

可选的 *header* 指定一个 *Content-Type* 的替代标头，并且所有形参将根据需要被转换，除非可选的 *requote* 为 False (默认为 True)。

如果指定了可选的 *charset*，形参将按照 **RFC 2231** 来编码。可选的 *language* 指定了 RFC 2231 的语言，默认为空字符串。*charset* 和 *language* 都应为字符串。

如果 *replace* 为 False （默认值），该头字段会被移动到所有头字段的末尾。如果 *replace* 为 True，字段会被原地更新。

在 3.4 版更改: 添加了 replace 关键字。

del_param(*param*, *header='content-type'*, *requote=True*)

从 *Content-Type* 标头中完全移除给定的形参。标头将被原地重写并不带该形参或它的值。所有的值将根据需要被转换，除非 *requote* 为 False (默认为 True)。可选的 *header* 指定 *Content-Type* 的一个替代项。

set_type(*type*, *header='Content-Type'*, *requote=True*)

设置 *Content-Type* 标头的主类型和子类型。*type* 必须为 *maintype/subtype* 形式的字符串，否则会引发*ValueError*。

此方法可替换 *Content-Type* 标头，并保持所有形参不变。如果 *requote* 为 False，这会保持原有标头引用转换不变，否则形参将被引用转换（默认行为）。

可以在 *header* 参数中指定一个替代标头。当 *Content-Type* 标头被设置时也会添加一个 *MIME-Version* 标头。

这是一个过时的方法。在 EmailMessage 类上它的功能已被 make_ 和 add_ 方法所替代。

get_filename(*failobj=None*)

返回信息头当中 *Content-Disposition* 字段当中名为 filename 的参数值。如果该字段当中没有此参数，该方法会退而寻找 *Content-Type* 字段当中的 name 参数值。如果这个也没有找到，或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照*email.utils.unquote()* 方法去除引号。

get_boundary(*failobj=None*)

返回信息头当中 *Content-Type* 字段当中名为 boundary 的参数值。如果字段当中没有此参数，

或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照*email.utils.unquote()* 方法去除引号。

set_boundary(*boundary*)
将 *Content-Type* 头字段的 boundary 参数设置为 *boundary* 。*set_boundary()* 方法永远都会在必要的时候为 *boundary* 添加引号。如果信息对象中没有 *Content-Type* 头字段，抛出*HeaderParseError* 异常。

请注意使用这个方法与删除旧的 *Content-Type* 标头并通过*add_header()* 添加一个带有新边界的新标头有细微的差异，因为*set_boundary()* 会保留 *Content-Type* 标头在原标头列表中的顺序。但是，它 不会保留原 *Content-Type* 标头中可能存在的任何连续的行。

get_content_charset(*failobj=None*)
返回 *Content-Type* 头字段中的 charset 参数，强制小写。如果字段当中没有此参数，或者这个字段压根不存在，返回 *failobj* 。

请注意此方法不同于*get_charset()*，后者会返回*Charset* 实例作为消息体的默认编码格式。

get_charsets(*failobj=None*)
返回一个包含了信息内所有字符集名字的列表。如果信息是 *multipart* 类型的，那么列表当中的每一项都对应其负载的子部分的字符集名字。否则，该列表是一个长度为 1 的列表。

列表中的每一项都是字符串，它们是其所表示的子部分的 *Content-Type* 标头中 charset 形参的值。但是，如果该子部分没有 *Content-Type* 标头，或没有 charset 形参，或者主 MIME 类型不是 *text*，则所返回列表中的对应项将为 *failobj*。

get_content_disposition()
如果信息的 *Content-Disposition* 头字段存在，返回其字段值；否则返回 None 。返回的值均为小写，不包含参数。如果信息遵循 **RFC 2183** 标准，则返回值只可能在 *inline* 、*attachment* 和 None 之间选择。

3.5 新版功能.

walk()
walk() 方法是一个多功能生成器。它可以被用来以深度优先顺序遍历信息对象树的所有部分和子部分。一般而言，*walk()* 会被用作 for 循环的迭代器，每一次迭代都返回其下一个子部分。

以下例子会打印出一封具有多部分结构之信息的每个部分的 MIME 类型。

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

walk 会遍历所有*is_multipart()* 方法返回 True 的部分之子部分，哪怕 msg.get_content_maintype() == 'multipart' 返回的是 False 。使用 _structure 除错帮助函数可以帮助我们在下面这个例子当中看清楚这一点：

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
```

(下页继续)

(续上页)

```
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
message/delivery-status
    text/plain
    text/plain
message/rfc822
    text/plain
```

在这里，`message` 的部分并非 `multiparts`，但是它们真的包含子部分！`is_multipart()` 返回 `True`，`walk` 也深入进这些子部分中。

Message 对象也可以包含两个可选的实例属性，它们可被用于生成纯文本的 MIME 消息。

preamble
MIME 文档格式在标头之后的空白行以及第一个多部分的分界字符串之间允许添加一些文本，通常，此文本在支持 MIME 的邮件阅读器中永远不可见，因为它处在标准 MIME 防护范围之外。但是，当查看消息的原始文本，或当在不支持 MIME 的阅读器中查看消息时，此文本会变得可见。

preamble 属性包含 MIME 文档开头部分的这些处于保护范围之外的文本。当*Parser* 在标头之后及第一个分界字符串之前发现一些文本时，它会将这些文本赋值给消息的 *preamble* 属性。当*Generator* 写出 MIME 消息的纯文本表示形式时，如果它发现消息具有 *preamble* 属性，它将在标头及第一个分界之间区域写出这些文本。请参阅*email.parser* 和*email.generator* 了解更多细节。

请注意如果消息对象没有前导文本，则 *preamble* 属性将为 `None`。

epilogue
epilogue 属性的作用方式与 *preamble* 属性相同，区别在于它包含出现于最后一个分界与消息结尾之间的文本。

你不需要将 epilogue 设为空字符串以便让*Generator* 在文件末尾打印一个换行符。

defects
defects 属性包含在解析消息时发现的所有问题的列表。请参阅*email.errors* 了解可能的解析缺陷的详细描述。

19.1.10 email.mime: 从头创建电子邮件和 MIME 对象

源代码: Lib/email/mime/

此模块是旧版 (`Compat32`) 电子邮件 API 的组成部分。它的功能在新版 API 中被*contentmanager* 部分替代，但在某些应用中这些类仍可能有用，即使是在非旧版代码中。

通常，你是通过传递一个文件或一些文本到解析器来获得消息对象结构体的，解析器会解析文本并返回根消息对象。不过你也可以从头开始构建一个完整的消息结构体，甚至是手动构建单独的*Message* 对象。实际上，你也可以接受一个现有的结构体并添加新的*Message* 对象并移动它们。这为切片和分割 MIME 消息提供了非常方便的接口。

你可以通过创建*Message* 实例并手动添加附件和所有适当的标头来创建一个新的对象结构体。不过对于 MIME 消息来说，*email* 包提供了一些便捷子类来让事情变得更容易。

这些类列示如下:

class email.mime.base.**MIMEBase**(*_maintype*, *_subtype*, *, *policy=compat32*, ***_params*)
模块: email.mime.base

这是*Message* 的所有 MIME 专属子类。通常你不会创建专门的*MIMEBase* 实例，尽管你可以这样做。*MIMEBase* 主要被提供用来作为更具体的 MIME 感知子类的便捷基类。

_maintype 是 *Content-Type* 的主类型 (例如 *text* 或 *image*)，而 *_subtype* 是 *Content-Type* 的次类型 (例如 *plain* 或 *gif*)。*_params* 是一个形参键/值字典并会被直接传递给*Message.add_header*。

如果指定了 *policy* (默认为*compat32* 策略)，它将被传递给*Message*。

MIMEBase 类总是会添加一个 *Content-Type* 标头 (基于 *_maintype*, *_subtype* 和 *_params*)，以及一个 *MIME-Version* 标头 (总是设为 1.0)。

在 3.6 版更改: 添加了 *policy* 仅限关键字形参。

class email.mime.nonmultipart.**MIMENonMultipart**
模块: email.mime.nonmultipart

MIMEBase 的子类，这是用于非 *multipart* MIME 消息的中间基类。这个类的主要目标是避免使用*attach()* 方法，该方法仅对 *multipart* 消息有意义。如果*attach()* 被调用，则会引发*MultipartConversionError* 异常。

class email.mime.multipart.**MIMEMultipart**(*_subtype='mixed'*, *boundary=None*, *_subparts=None*, *, *policy=compat32*, ***_params*)
模块: email.mime.multipart

MIMEBase 的子类，这是用于 *multipart* MIME 消息的中间基类。可选的 *_subtype* 默认为 *mixed*，但可被用来指定消息的子类型。将会在消息对象中添加一个:mimetype:*multipart/_subtype* 的 *Content-Type* 标头。并还将添加一个 *MIME-Version* 标头。

可选的 *boundary* 是多部分边界字符串。当为 None (默认值) 时，则会在必要时（例如当消息被序列化时）计算边界。

_subparts 是载荷初始子部分的序列。此序列必须可以被转换为列表。你总是可以使用*Message.attach* 方法将新的子部分附加到消息中。

可选的 *policy* 参数默认为*compat32*。

用于 *Content-Type* 标头的附加形参会从关键字参数中获取，或者传入到 *_params* 参数，该参数是一个关键字的字典。

在 3.6 版更改: 添加了 *policy* 仅限关键字形参。

class email.mime.application.**MIMEApplication**(*_data*, *_subtype='octet-stream'*, *_encoder=email.encoders.encode_base64*, *, *policy=compat32*, ***_params*)
模块: email.mime.application

MIMENonMultipart 的子类，*MIMEApplication* 类被用来表示主类型为 *application* 的 MIME 消息。*_data* 是包含原始字节数据的字符串。可选的 *_subtype* 指定 MIME 子类型并默认为 *octet-stream*。

可选的 *_encoder* 是一个可调用对象（即函数），它将执行实际的数据编码以便传输。这个可调用对象接受一个参数，该参数是*MIMEApplication* 的实例。它应当使用*get_payload()* 和*set_payload()* 来将载荷改为已编码形式。它还应根据需要将任何 *Content-Transfer-Encoding* 或其他标头添加到消息对象中。默认编码格式为 base64。请参阅*email.encoders* 模块来查看内置编码器列表。

可选的 *policy* 参数默认为*compat32*。

_params 会被直接传递给基类的构造器。

在 3.6 版更改: 添加了 *policy* 仅限关键字形参。

class email.mime.audio.**MIMEAudio**(*_audiodata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

模块: email.mime.audio

MIMENonMultipart 的子类，*MIMEAudio* 类被用来创建主类型为 *audio* 的 MIME 消息。*_audiodata* 是包含原始音频数据的字符串。如果此数据可由标准 Python 模块*sndhdr* 来编码，则其子类型将被自动包括在 *Content-Type* 标头。在其他情况下你可以通过 *_subtype* 参数显式地指定音频子类型。如果无法猜测出主类型并且未给出 *_subtype*，则会引发*TypeError*。

可选的 *_encoder* 是一个可调用对象（即函数），它将执行实际的音频数据编码以便传输。这个可调用对象接受一个参数，该参数是*MIMEAudio* 的实例。它应当使用*get_payload()* 和*set_payload()* 来将载荷改为已编码形式。它还应根据需要将任何 *Content-Transfer-Encoding* 或其他标头添加到消息对象中。默认编码格式为 base64。请参阅*email.encoders* 模块来查看内置编码器列表。

可选的 *policy* 参数默认为*compat32*。

_params 会被直接传递给基类的构造器。

在 3.6 版更改: 添加了 *policy* 仅限关键字形参。

class email.mime.image.**MIMEImage**(*_imagedata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

模块: email.mime.image

MIMENonMultipart 的子类，*MIMEImage* 类被用来创建主类型为 *image* 的 MIME 消息对象。*_imagedata* 是包含原始图像数据的字符串。如果此数据可由标准 Python 模块*imghdr* 来解码，则其子类型将被自动包括在 *Content-Type* 标头中。在其他情况下你可以通过 *_subtype* 参数显式地指定图像子类型。如果无法猜测出主类型并且未给出 *_subtype*，则会引发*TypeError*。

可选的 *_encoder* 是一个可调用对象（即函数），它将执行实际的图像数据编码以便传输。这个可调用对象接受一个参数，该参数是*MIMEImage* 的实例。它应当使用*get_payload()* 和*set_payload()* 来将载荷改为已编码形式。它还应根据需要将任何 *Content-Transfer-Encoding* 或其他标头添加到消息对象中。默认编码格式为 base64。请参阅*email.encoders* 模块来查看内置编码器列表。

可选的 *policy* 参数默认为*compat32*。

_params 会被直接传递给*MIMEBase* 构造器。

在 3.6 版更改: 添加了 *policy* 仅限关键字形参。

class email.mime.message.**MIMEMessage**(*_msg*, *_subtype='rfc822'*, ***, *policy=compat32*)

模块: email.mime.message

MIMENonMultipart 的子类，*MIMEMessage* 类被用来创建主类型为 *message* 的 MIME 对象。*_msg* 将被用作载荷，并且必须为*Message* 类（或其子类）的实例，否则会引发*TypeError*。

可选的 *_subtype* 设置消息的子类型；它的默认值为 *rfc822*。

可选的 *policy* 参数默认为*compat32*。

在 3.6 版更改: 添加了 *policy* 仅限关键字形参。

class email.mime.text.**MIMEText**(*_text*, *_subtype='plain'*, *_charset=None*, ***, *policy=compat32*)

模块: email.mime.text

MIMENonMultipart 的子类，*MIMEText* 类被用来创建主类型为 *text* 的 MIME 对象。*_text* 是用作载荷的字符串。*_subtype* 指定子类型并且默认为 *plain*。*_charset* 是文本的字符集并会作为参数传递给*MIMENonMultipart* 构造器；如果该字符串仅包含 ascii 码位则其默认值为 us-ascii，否则为 utf-8。*_charset* 形参接受一个字符串或是一个*Charset* 实例。

除非 *_charset* 参数被显式地设为 None，否则所创建的 MIMEText 对象将同时具有附带 charset 形参的 *Content-Type* 标头，以及 *Content-Transfer-Encoding* 标头。这意味着后续的 set_payload

调用将不再产生已编码的载荷，即使它在 set_payload 命令中被传入。你可以通过删除 Content-Transfer-Encoding 标头来“重置”此行为，在此之后的 set_payload 调用将自动编码新的载荷（并添加新的 *Content-Transfer-Encoding* 标头）。

可选的 *policy* 参数默认为 *compat32*。

在 3.5 版更改: *_charset* 也可接受 *Charset* 实例。

在 3.6 版更改: 添加了 *policy* 仅限关键字形参。

19.1.11 email.header: 国际化标头

源代码: Lib/email/header.py

此模块是旧式 (Compat32) email API 的一部分。在当前的 API 中标头的编码和解码是由 *EmailMessage* 类的字典型 API 来透明地处理的。除了在旧有代码中使用，此模块在需要完全控制当编码标头时所使用的字符集时也很有用处。

本段落中的剩余文本是该模块的原始文档。

RFC 2822 是描述电子邮件消息格式的基础标准。它派生自更早的 **RFC 822** 标准，该标准在大多数电子邮件仅由 ASCII 字符组成时已被广泛使用。**RFC 2822** 所描述的规范假定电子邮件都只包含 7 位 ASCII 字符。

当然，随着电子邮件在全球部署，它已经变得国际化了，例如电子邮件消息中现在可以使用特定语言的专属字符集。这个基础标准仍然要求电子邮件消息只使用 7 位 ASCII 字符来进行传输，为此编写了大量 RFC 来描述如何将包含非 ASCII 字符的电子邮件编码为符合 **RFC 2822** 的格式。这些 RFC 包括 **RFC 2045**, **RFC 2046**, **RFC 2047** 和 **RFC 2231**。*email* 包在其 *email.header* 和 *email.charset* 模块中支持了这些标准。

如果你想在你的电子邮件标头中包括非 ASCII 字符，比如说是在 *Subject* 或 *To* 字段中，你应当使用 *Header* 类并将 *Message* 对象中的字段赋值为 *Header* 的实例而不是使用字符串作为字段值。请从 *email.header* 模块导入 *Header* 类。例如:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

是否注意到这里我们是如何希望 *Subject* 字段包含非 ASCII 字符的？我们通过创建一个 *Header* 实例并传入字节串编码所用的字符集来做到这一点。当后续的 *Message* 实例被展平时，*Subject* 字段会正确地按 **RFC 2047** 来编码。可感知 MIME 的电子邮件阅读器将会使用嵌入的 ISO-8859-1 字符来显示此标头。

以下是 *Header* 类描述:

class email.header.**Header**(*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*, *errors='strict'*)

创建符合 MIME 要求的标头，其中可包含不同字符集的字符串。

可选的 *s* 是初始标头值。如果为 None (默认值)，则表示初始标头值未设置。你可以在稍后使用 *append()* 方法调用向标头添加新值。*s* 可以是 *bytes* 或 *str* 的实例，注意参阅 *append()* 文档了解相关语义。

可选的 *charset* 用于两种目的：它的含义与 *append()* 方法的 *charset* 参数相同。它还会为所有省略了 *charset* 参数的后续 *append()* 调用设置默认字符集。如果 *charset* 在构造器中未提供（默认设置），则会将 us-ascii 字符集用作 *s* 的初始字符集以及后续 *append()* 调用的默认字符集。

通过 *maxlinelen* 可以显式指定最大行长度。要将第一行拆分为更短的值 (以适应未被包括在 to account for the field header which isn't included in *s* 中的字段标头，例如 *Subject*)，则将字段名称作为 *header_name* 传入。*maxlinelen* 默认值为 76，而 *header_name* 默认值为 None，表示不考虑拆分超长标头的第一行。

可选的 *continuation_ws* 必须为符合 **RFC 2822** 的折叠用空白符，通常是空格符或硬制表符。这个字符将被加缀至连续行的开头。*continuation_ws* 默认为一个空格符。

可选的 *errors* 会被直接传递给*append()* 方法。

append(*s*, *charset=None*, *errors='strict'*)
将字符串 *s* 添加到 MIME 标头。

如果给出可选的 *charset*，它应当是一个*Charset* 实例 (参见*email.charset*) 或字符集名称，该参数将被转换为一个*Charset* 实例。如果为 None (默认值) 则表示会使用构造器中给出的 *charset*。

s 可以是*bytes* 或*str* 的实例。如果它是*bytes* 的实例，则 *charset* 为该字节串的编码格式，如果字节串无法用该字符集来解码则将引发*UnicodeError*。

如果 *s* 是*str* 的实例，则 *charset* 是用来指定字符串中字符字符集的提示。

在这两种情况下，当使用 **RFC 2047** 规则产生符合 **RFC 2822** 的标头时，将使用指定字符集的输出编解码器来编码字符串。如果字符串无法使用该输出编解码器来编码，则将引发 UnicodeError。

可选的 *errors* 会在 *s* 为字节串时被作为 errors 参数传递给 decode 调用。

encode(*splitchars=';, \t'*, *maxlinelen=None*, *linesep='\n'*)
将消息标头编码为符合 RFC 的格式，可能会对过长的行采取折行并将非 ASCII 部分以 base64 或 quoted-printable 编码格式进行封装。

可选的 *splitchars* 是一个字符串，其中包含应在正常的标头折行处理期间由拆分算法赋予额外权重的字符。这是对于 **RFC 2822** 中‘更高层级语法拆分’的很粗略的支持：在拆分期间会首选在 splitchar 之前的拆分点，字符的优先级是基于它们在字符串中的出现顺序。字符串中可包含空格和制表符以指明当其他拆分字符未在被拆分行中出现时是否要将某个字符作为优先于另一个字符的首选拆分点。拆分字符不会影响以 **RFC 2047** 编码的行。

如果给出 *maxlinelen*，它将覆盖实例的最大行长度值。

linesep 指定用来分隔已折叠标头行的字符。它默认为 Python 应用程序代码中最常用的值 (\n)，但也可以指定为 \r\n 以便产生带有符合 RFC 的行分隔符的标头。

在 3.2 版更改: 增加了 *linesep* 参数。

Header 类还提供了一些方法以支持标准运算符和内置函数。

__str__()
以字符串形式返回*Header* 的近似表示，使用不受限制的行长度。所有部分都会使用指定编码格式转换为 unicode 并适当地连接起来。任何带有 'unknown-8bit' 字符集的部分都会使用 'replace' 错误处理程序解码为 ASCII。

在 3.2 版更改: 增加对 'unknown-8bit' 字符集的处理。

__eq__(*other*)
这个方法允许你对两个*Header* 实例进行相等比较。

__ne__(*other*)
这个方法允许你对两个*Header* 实例进行不等比较。

email.header 模块还提供了下列便捷函数。

email.header.**decode_header**(*header*)
在不转换字符集的情况下对消息标头值进行解码。*header* 为标头值。

这个函数返回一个 (decoded_string, charset) 对的列表，其中包含标头的每个已解码部分。对于标头的未编码部分 *charset* 为 None，在其他情况下则为一个包含已编码字符串中所指定字符集名称的小写字符串。

以下是为示例代码:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
```

email.header.**make_header**(*decoded_seq*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*)
基于*decode_header()* 所返回的数据对序列创建一个*Header* 实例。

decode_header() 接受一个标头值字符串并返回格式为 (decoded_string, charset) 的数据对序列，其中 *charset* 是字符集名称。

这个函数接受这样的数据对序列并返回一个*Header* 实例。可选的 *maxlinelen*, *header_name* 和 *continuation_ws* 与*Header* 构造器中的含义相同。

19.1.12 email.charset: 表示字符集

源代码: Lib/email/charset.py

此模块是旧版 (Compat32) email API 的组成部分。在新版 API 中只会使用其中的别名表。

本段落中的剩余文本是该模块的原始文档。

此模块提供了一个*Charset* 类用来表示电子邮件消息中的字符集和字符集转换操作，以及一个字符集注册表和几个用于操作此注册表的便捷方法。*Charset* 的实例在*email* 包的其他几个模块中也有使用。

请从*email.charset* 模块导入这个类。

class email.charset.**Charset**(*input_charset=DEFAULT_CHARSET*)
将字符集映射到其 email 特征属性。

这个类提供了特定字符集对于电子邮件的要求的相关信息。考虑到适用编解码器的可用性，它还为字符集之间的转换提供了一些便捷例程。在给定字符集的情况下，它将尽可能地以符合 RFC 的方式在电子邮件消息中提供有关如何使用该字符集的信息。

特定字符集当在电子邮件标头或消息体中使用时必须以 quoted-printable 或 base64 来编码。某些字符集则必须被立即转换，不允许在电子邮件中使用。

可选的 *input_charset* 说明如下；它总是会被强制转为小写。在进行别名正规化后它还会被用来查询字符集注册表以找出用于该字符集的标头编码格式、消息体编码格式和输出转换编解码器。举例来说，如果 *input_charset* 为 iso-8859-1，则标头和消息体将会使用 quoted-printable 来编码并且不需要输出转换编解码器。如果 *input_charset* 为 euc-jp，则标头将使用 base64 来编码，消息体将不会被编码，但输出文本将从 euc-jp 字符集转换为 iso-2022-jp 字符集。

Charset 实例具有下列数据属性:

input_charset
指定的初始字符集。通用别名会被转换为它们的 官方 电子邮件名称 (例如 latin_1 会被转换为 iso-8859-1)。默认值为 7 位 us-ascii。

header_encoding
如果字符集在用于电子邮件标头之前必须被编码，此属性将被设为 Charset.QP (表示 quoted-printable 编码格式), Charset.BASE64 (表示 base64 编码格式) 或 Charset.SHORTEST 表示 QP 或 BASE64 编码格式中最简短的一个。在其他情况下，该属性将为 None.

body_encoding
与 *header_encoding* 一样，但是用来描述电子邮件消息体的编码格式，它实际上可以与标头编码格式不同。`Charset.SHORTEST` 不允许被用作 *body_encoding*。

output_charset
某些字符集在用于电子邮件标头或消息体之前必须被转换。如果 *input_charset* 是这些字符集之一，该属性将包含输出将要转换的字符集名称。在其他情况下，该属性将为 `None`。

input_codec
用于将 *input_charset* 转换为 Unicode 的 Python 编解码器名称。如果不需要任何转换编解码器，该属性将为 `None`。

output_codec
用于将 Unicode 转换为 *output_charset* 的 Python 编解码器名称。如果不需要任何转换编解码器，该属性将具有与 *input_codec* 相同的值。

Charset 实例还有下列方法:

get_body_encoding()
返回用于消息体编码的内容转换编码格式。

根据所使用的编码格式返回 `quoted-printable` 或 `base64`，或是返回一个函数，在这种情况下你应当调用该函数并附带一个参数，即被编码的消息对象。该函数应当自行将 *Content-Transfer-Encoding* 标头设为适当的值。

如果 *body_encoding* 为 `QP` 则返回字符串 `quoted-printable`，如果 *body_encoding* 为 `BASE64` 则返回字符串 `base64`，并在其他情况下返回字符串 `7bit`。

get_output_charset()
返回输出字符集。

如果 *output_charset* 属性不为 `None` 则返回该属性，否则返回 *input_charset*。

header_encode(*string*)
对字符串 *string* 执行标头编码。

编码格式的类型 (base64 或 quoted-printable) 将取决于 *header_encoding* 属性。

header_encode_lines(*string*, *maxlengths*)
通过先将 *string* 转换为字节串来对其执行标头编码。

这类似于 *header_encode()*，区别是字符串会被调整至参数 *maxlengths* 所给出的最大行长度，它应当是一个迭代器：该迭代器返回的每个元素将提供下一个最大行长度。

body_encode(*string*)
对字符串 *string* 执行消息体编码。

编码格式的类型 (base64 或 quoted-printable) 将取决于 *body_encoding* 属性。

Charset 类还提供了一些方法以支持标准运算和内置函数。

__str__()
将 *input_charset* 以强制转为小写的字符串形式返回。`__repr__()` 是 *__str__()* 的别名。

__eq__(*other*)
这个方法允许你对两个 *Charset* 实例进行相等比较。

__ne__(*other*)
这个方法允许你对两个 *Charset* 实例进行相等比较。

email.charset 模块还提供了下列函数用于向全局字符集、别名以及编解码器注册表添加新条目:

email.charset.**add_charset**(*charset*, *header_enc=None*, *body_enc=None*, *output_charset=None*)
向全局注册表添加字符特征属性。

charset 是输入字符集，它必须为某个字符集的正规名称。

可选的 *header_enc* 和 *body_enc* 可以是 `Charset.QP` 表示 quoted-printable, `Charset.BASE64` 表示 base64 编码格式, `Charset.SHORTEST` 表示 quoted-printable 或 base64 编码格式中较短的一个，或者为 `None` 表示没有编码格式。`SHORTEST` 是 *header_enc* 的唯一有效值。默认值为 `None` 表示没有编码格式。

可选的 *output_charset* 是输出所应当采用的字符集。当 `Charset.convert()` 方法被调用时将会执行从输入字符集到输出字符集的转换。默认情况下输出字符集将与输入字符集相同。

input_charset 和 *output_charset* 都必须在模块的字符集-编解码器映射中具有 Unicode 编解码器条目；使用*add_codec()* 可添加本模块还不知道的编解码器。请参阅*codecs* 模块的文档来了解更多信息。

全局字符集注册表保存在模块全局字典 `CHARSETS` 中。

`email.charset.`**`add_alias`**(*alias*, *canonical*)
添加一个字符集别名。*alias* 为特定的别名，例如 `latin-1`。*canonical* 是字符集的正规名称，例如 `iso-8859-1`。

全局字符集注册表保存在模块全局字典 `ALIASES` 中。

`email.charset.`**`add_codec`**(*charset*, *codecname*)
添加在给定字符集的字符和 Unicode 之间建立映射的编解码器。

charset 是某个字符集的正规名称。*codecname* 是某个 Python 编解码器的名称，可以被用来作为*str* 的*encode()* 方法的第二个参数。

19.1.13 `email.encoders`: 编码器

源代码: Lib/email/encoders.py

此模块是旧版 (`Compat32`) email API 的组成部分。在新版 API 中将由*set_content()* 方法的 *cte* 形参提供该功能。

本段落中的剩余文本是该模块的原始文档。

当创建全新的*Message* 对象时，你经常需要对载荷编码以便通过兼容的邮件服务器进行传输。对于包含二进制数据的 *image/** 和 *text/** 类型的消息来说尤其如此。

email 包在其 `encoders` 模块中提供了一些方便的编码。这些编码器实际上由*MIMEAudio* 和*MIMEImage* 类构造函数使用，以提供默认编码。所有编码器函数只接受一个参数，即要编码的消息对象。它们通常提取有效数据，对其进行编码，并将有效数据重置为此新编码的值。他们还应该根据需要设置 *Content-Transfer-Encoding* 标头。

请注意，这些函数对于多段消息没有意义。它们必须应用到各个单独的段上面，而不是整体。如果直接传递一个多段类型的消息，会产生一个*TypeError* 错误。

下面是提供的编码函数：

`email.encoders.`**`encode_quopri`**(*msg*)
将有效数据编码为 quoted-printable 形式，并将:mailheader:*Content-Transfer-Encoding*‘标头设置为 *“quoted-printable*[1]. 。当大多数实际的数据是普通的可打印数据但包含少量不可打印的字符时，这是一个很好的编码。

`email.encoders.`**`encode_base64`**(*msg*)
将有效载荷编码为 base64 形式，并将 *Content-Transfer-Encoding* 标头设为 `base64`。当你的载荷主要包含不可打印数据时这是一种很好用的编码格式，因为它比 quoted-printable 更紧凑。base64 编码格式的缺点是它会使文本变成人类不可读的形式。

[1] 请注意使用*encode_quopri()* 编码格式还会对数据中的所有制表符和空格符进行编码。

email.encoders.**encode_7or8bit**(*msg*)
这并不实际改变消息的有效载荷，但它会基于载荷数据将 *Content-Transfer-Encoding* 标头相应地设为 7bit 或 8bit。

email.encoders.**encode_noop**(*msg*)
这样什么都不会做；它甚至不会设置 *Content-Transfer-Encoding* 标头。

备注

19.1.14 email.utils: 其他工具

源代码: Lib/email/utils.py

---

email.utils 模块提供如下几个工具

email.utils.**localtime**(*dt=None*)
以感知型 datetime 对象返回当地时间。如果调用时参数为空，则返回当前时间。否则 *dt* 参数应该是一个*datetime* 实例，并根据系统时区数据库转换为当地时区。如果 *dt* 是简单型的 (即 dt.tzinfo 是 None)，则假定为当地时间。在这种情况下，为正值或零的 *isdst* 会使 localtime 假定夏季时间（例如，夏令时）对指定时间生效或不生效。负值 *isdst* 会使 localtime 预测夏季时间对指定时间是否生效。

3.3 新版功能.

email.utils.**make_msgid**(*idstring=None*, *domain=None*)
返回一个适合作为兼容 **RFC 2822** 的 *Message-ID* 标头的字符串。可选参数 *idstring* 可传入一字符串以增强该消息 ID 的唯一性。可选参数 *domain* 可用于提供消息 ID 中字符‘@’之后的部分，其默认值是本机的主机名。正常情况下无需覆盖此默认值，但在特定情况下覆盖默认值可能会有用，比如构建一个分布式系统，在多台主机上采用一致的域名。

在 3.2 版更改: 增加了关键字 *domain*

下列函数是旧（Compat32）电子邮件 API 的一部分。新 API 提供的解析和格式化在标头解析机制中已经被自动完成，故在使用新 API 时没有必要直接使用它们函数。

email.utils.**quote**(*str*)
返回一个新的字符串，*str* 中的反斜杠被替换为两个反斜杠，并且双引号被替换为反斜杠加双引号。

email.utils.**unquote**(*str*)
返回 *str* 被去除引用后的字符串。如果 *str* 开头和结尾均是双引号，则这对双引号被去除。类似地，如果 *str* 开头和结尾都是尖角括号，这对尖角括号会被去除。

email.utils.**parseaddr**(*address*)
将地址（应为诸如 *To* 或者 *Cc* 之类包含地址的字段值）解析为构成之的 *真实名字* 和 *电子邮件地址* 部分。返回包含这两个信息的一个元组；如若解析失败，则返回一个二元组 ('', '')。

email.utils.**formataddr**(*pair*, *charset='utf-8'*)
是*parseaddr()* 的逆操作，接受一个（*真实名字*, *电子邮件地址*）的二元组，并返回适合于 *To* or *Cc* 标头的字符串。如果第一个元素为假性值，则第二个元素将被原样返回。

可选地，如果指定 *charset*，则被视为一符合 **RFC 2047** 的编码字符集，用于编码 *真实名字* 中的非 ASCII 字符。可以是一个*str* 类的实例，或者一个*Charset* 类。默认为 utf-8 。

在 3.3 版更改: 添加了 *charset* 选项。

email.utils.**getaddresses**(*fieldvalues*)
该方法返回一个形似 parseaddr() 返回的二元组的列表。*fieldvalues* 是一个序列，包含了形似*Message.get_all* 返回值的标头字段值。获取了一消息的所有收件人的简单示例如下：

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

email.utils.**parsedate**(*date*)

尝试根据 **RFC 2822** 的规则解析一个日期。然而，有些寄信人不严格遵守这一格式，所以这种情况下*parsedate()* 会尝试猜测其形式。*date* 是一个字符串包含了一个形如 `"Mon, 20 Nov 1995 19:12:08 -0500"` 的 **RFC 2822** 格式日期。如果日期解析成功，*parsedate()* 将返回一个九元组，可直接传递给*time.mktime()*；否则返回 `None`。注意返回的元组中下标为 6、7、8 的部分是无用的。

email.utils.**parsedate_tz**(*date*)

Performs the same function as *parsedate()*, but returns either `None` or a 10-tuple; the first 9 elements make up a tuple that can be passed directly to *time.mktime()*, and the tenth is the offset of the date's timezone from UTC (which is the official term for Greenwich Mean Time)[1]. If the input string has no timezone, the last element of the tuple returned is `None`. Note that indexes 6, 7, and 8 of the result tuple are not usable.

email.utils.**parsedate_to_datetime**(*date*)

format_datetime() 的逆操作。执行与*parsedate()* 相同的功能，但会在成功时返回一个*datetime*。如果输入日期的时区值为 `-0000`，则 `datetime` 将为一个简单型 `datetime`，而如果日期符合 RFC 标准则它将代表一个 UTC 时间，但是并不指明日期所在消息的实际源时区。如果输入日期具有任何其他有效的时区差值，则 `datetime` 将为一个感知型 `datetime` 并与*timezone tzinfo* 相对应。

3.3 新版功能.

email.utils.**mktime_tz**(*tuple*)

将*parsedate_tz()* 所返回的 10 元组转换为一个 UTC 时间戳（相距 Epoch 纪元初始的秒数）。如果元组中的时区项为 `None`，则视为当地时间。

email.utils.**formatdate**(*timeval=None*, *localtime=False*, *usegmt=False*)

返回 **RFC 2822** 标准的日期字符串，例如:

```
Fri, 09 Nov 2001 01:08:47 -0000
```

可选的 *timeval* 如果给出，则是一个可被*time.gmtime()* 和*time.localtime()* 接受的浮点数时间值，否则会使得当前时间。

可选的 *localtime* 是一个旗标，当为 `True` 时，将会解析 *timeval*，并返回一个相对于当地时区而非 UTC 的日期值，并会适当地考虑夏令时。默认值 `False` 表示使用 UTC。

可选的 *usegmt* 是一个旗标，当为 `True` 时，将会输出一个日期字符串，其中时区表示为 ascii 字符串 `GMT` 而非数字形式的 `-0000`。这对某些协议（例如 HTTP）来说是必要的。这仅在 *localtime* 为 `False` 时应用。默认值为 `False`。

email.utils.**format_datetime**(*dt*, *usegmt=False*)

类似于 `formatdate`，但输入的是一个*datetime* 实例。如果实例是一个简单型 datetime，它会被视为“不带源时区信息的 UTC”，并且使用传统的 `-0000` 作为时区。如果实例是一个感知型 `datetime`，则会使用数字形式的时区时差。如果实例是感知型且时区时差为零，则 *usegmt* 可能会被设为 `True`，在这种情况下将使用字符串 `GMT` 而非数字形式的时区时差。这提供了一种生成符合标准 HTTP 日期标头的方式。

3.3 新版功能.

[1] 请注意时区时差的符号与同一时区的 `time.timezone` 变量的符号相反；后者遵循 POSIX 标准而此模块遵循 **RFC 2822**。

email.utils.**decode_rfc2231**(*s*)
根据 RFC 2231 解码字符串 *s*。

email.utils.**encode_rfc2231**(*s*, *charset=None*, *language=None*)
根据 RFC 2231 对字符串 *s* 进行编码。可选的 *charset* 和 *language* 如果给出，则为指明要使用的字符集名称和语言名称。如果两者均未给出，则会原样返回 *s*。如果给出 *charset* 但未给出 *language*，则会使用空字符串作为 *language* 值来对字符串进行编码。

email.utils.**collapse_rfc2231_value**(*value*, *errors='replace'*, *fallback_charset='us-ascii'*)
当以 RFC 2231 格式来编码标头形参时，`Message.get_param` 可能返回一个包含字符集、语言和值的 3 元组。`collapse_rfc2231_value()` 会将此返回为一个 unicode 字符串。可选的 *errors* 会被传递给`str` 的`encode()` 方法的 *errors* 参数；它的默认值为 `'replace'`。可选的 *fallback_charset* 指定当 RFC 2231 标头中的字符集无法被 Python 识别时要使用的字符集；它的默认值为 `'us-ascii'`。

为方便起见，如果传给`collapse_rfc2231_value()` 的 *value* 不是一个元组，则应为一个字符串并会将其原样返回。

email.utils.**decode_params**(*params*)
根据 RFC 2231 解码参数列表。*params* 是一个包含 `(content-type, string-value)` 形式的元素的 2 元组的序列。

备注

19.1.15 email.iterators: 迭代器

源代码: Lib/email/iterators.py

通过`Message.walk` 方法来迭代消息对象树是相当容易的。`email.iterators` 模块提供了一些适用于消息对象树的高层级迭代器。

email.iterators.**body_line_iterator**(*msg*, *decode=False*)
此对象会迭代 *msg* 的所有子部分中的所有载荷，逐行返回字符串载荷。它会跳过所有子部分的标头，并且它也会跳过任何包含不为 Python 字符串的载荷的子部分。这基本上等价于使用`readline()` 从一个文件读取消息的纯文本表示形式，并跳过所有中间的标头。

可选的 *decode* 会被传递给`Message.get_payload`。

email.iterators.**typed_subpart_iterator**(*msg*, *maintype='text'*, *subtype=None*)
此函数会迭代 *msg* 的所有子部分，只返回其中与 *maintype* 和 *subtype* 所指定的 MIME 类型相匹配的子部分。

请注意 *subtype* 是可选项；如果省略，则仅使用主类型来进行子部分 MIME 类型的匹配。*maintype* 也是可选项；它的默认值为 `text`。

因此，在默认情况下`typed_subpart_iterator()` 会返回每一个 MIME 类型为 `text/*` 的子部分。

增加了以下函数作为有用的调试工具。它 不应当被视为该包所支持的公共接口的组成部分。

email.iterators.**_structure**(*msg*, *fp=None*, *level=0*, *include_default=False*)
打印消息对象结构的内容类型的缩进表示形式。例如:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
```

(下页继续)

(续上页)

```
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

可选项 *fp* 是一个作为打印输出目标的文件类对象。它必须适用于 Python 的*print()* 函数。*level* 是供内部使用的。*include_default* 如果为真值，则会同时打印默认类型。

参见:

***smtplib* 模块** SMTP (简单邮件传输协议) 客户端

***poplib* 模块** POP (邮局协议) 客户端

***imaplib* 模块** IMAP (互联网消息访问协议) 客户端

***nntplib* 模块** NNTP (网络新闻传输协议) 客户端

***mailbox* 模块** 创建、读取和管理使用保存在磁盘中的多种标准格式的消息集的工具。

***smtpd* 模块** SMTP 服务器框架 (主要适用于测试)

19.2 json —JSON 编码和解码器

源代码: Lib/json/__init__.py

JSON (JavaScript Object Notation)，由 **RFC 7159** (which obsoletes **RFC 4627**) 和 ECMA-404 指定，是一个受 JavaScript 的对象字面量语法启发的轻量级数据交换格式，尽管它不仅仅是一个严格意义上的 JavaScript 的字集[1]。

json 提供了与标准库*marshal* 和*pickle* 相似的 API 接口。

对基本的 Python 对象层次结构进行编码:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
```

(下页继续)

[1] 正如 RFC 7159 的勘误表 所说明的，JSON 允许以字符串表示字面值字符 U+2028 (LINE SEPARATOR) 和 U+2029 (PARAGRAPH SEPARATOR)，而 JavaScript (在 ECMAScript 5.1 版中) 不允许。

(续上页)

```
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

紧凑编码：

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

美化输出：

```
>>> import json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

JSON 解码：

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

特殊 JSON 对象解码：

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

扩展*JSONEncoder*：

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, obj)
```

(下页继续)

(续上页)

```
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

从命令行使用*json.tool*来验证并美化输出：

```
$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

详细文档请参见命令行界面。

注解： JSON 是 YAML 1.2 的一个子集。由该模块的默认设置生成的 JSON （尤其是默认的“分隔符”设置值）也是 YAML 1.0 and 1.1 的一个子集。因此该模块也能够用于序列化为 YAML。

19.2.1 基本使用

json.**dump**(*obj*, *fp*, ***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

使用这个转换表 将 *obj* 序列化为 JSON 格式化流形式的 *fp* (支持 `.write()` 的*file-like object*)。

如果 *skipkeys* 是 true （默认为 `False`），那么那些不是基本对象（包括*str*, *int*、*float*、*bool*、`None`）的字典的键会被跳过；否则引发一个*TypeError*。

json 模块始终产生*str* 对象而非*bytes* 对象。因此，`fp.write()` 必须支持*str* 输入。

如果 *ensure_ascii* 是 true （即默认值），输出保证将所有输入的非 ASCII 字符转义。如果 *ensure_ascii* 是 false，这些字符会原样输出。

如果 *check_circular* 是为假值 (默认为 `True`)，那么容器类型的循环引用检验会被跳过并且循环引用会引发一个*OverflowError* (或者更糟的情况)。

如果 *allow_nan* 是 false （默认为 `True`），那么在对严格 JSON 规格范围外的*float* 类型值（`nan`、`inf` 和 `-inf`）进行序列化时会引发一个*ValueError*。如果 *allow_nan* 是 true，则使用它们的 JavaScript 等价形式（`NaN`、`Infinity` 和 `-Infinity`）。

如果 *indent* 是一个非负整数或者字符串，那么 JSON 数组元素和对象成员会被美化输出为该值指定的缩进等级。如果缩进等级为零、负数或者 `""`，则只会添加换行符。`None` (默认值) 选择最紧凑的表达。使用一个正整数会让每一层缩进同样数量的空格。如果 *indent* 是一个字符串 (比如 `"\t"`)，那个字符串会被用于缩进每一层。

在 3.2 版更改: 现允许使用字符串作为 *indent* 而不再仅仅是整数。

当被指定时，*separators* 应当是一个 `(item_separator, key_separator)` 元组。当 *indent* 为 `None` 时，默认值取 `(', ', ': ')`，否则取 `(',', ': ')`。为了得到最紧凑的 JSON 表达式，你应该指定其为 `(',', ':')` 以消除空白字符。

在 3.4 版更改: 现当 *indent* 不是 `None` 时，采用 `(',', ': ')` 作为默认值。

当 *default* 被指定时，其应该是一个函数，每当某个对象无法被序列化时它会被调用。它应该返回该对象的一个可以被 JSON 编码的版本或者引发一个*TypeError*。如果没有被指定，则会直接引发*TypeError*。

如果 *sort_keys* 是 true（默认为 `False`），那么字典的输出会以键的顺序排序。

为了使用一个自定义的*JSONEncoder* 子类（比如：覆盖了 `default()` 方法来序列化额外的类型），通过 *cls* 关键字参数来指定；否则将使用*JSONEncoder*。

在 3.6 版更改: 所有可选形参现在都是仅限关键字参数。

注解： 与*pickle* 和*marshal* 不同，JSON 不是一个具有框架的协议，所以尝试多次使用同一个 *fp* 调用*dump()* 来序列化多个对象会产生一个不合规的 JSON 文件。

---

json.**dumps**(*obj*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, **kw*)

使用这个转换表 将 *obj* 序列化为 JSON 格式的*str*。其参数的含义与*dump()* 中的相同。

注解： JSON 中的键-值对中的键永远是*str* 类型的。当一个对象被转化为 JSON 时，字典中所有的键都会被强制转换为字符串。这所造成的结果是字典被转换为 JSON 然后转换回字典时可能和原来的不相等。换句话说，如果 x 具有非字符串的键，则有 `loads(dumps(x)) != x`。

---

json.**load**(*fp*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, **kw*)

使用这个转换表 将 *fp* (一个支持 `.read()` 并包含一个 JSON 文档的*text file* 或者*binary file*) 反序列化为一个 Python 对象。

object_hook 是一个可选的函数，它会被调用于每一个解码出的对象字面量（即一个*dict*）。*object_hook* 的返回值会取代原本的*dict*。这一特性能够被用于实现自定义解码器（如 JSON-RPC 的类型提示）。

object_pairs_hook is an optional function that will be called with the result of any object literal decoded with an ordered list of pairs. The return value of *object_pairs_hook* will be used instead of the *dict*. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, *collections.OrderedDict()* will remember the order of insertion). If *object_hook* is also defined, the *object_pairs_hook* takes priority.

在 3.1 版更改: 添加了对 *object_pairs_hook* 的支持。

parse_float , 如果指定, 将与每个要解码 JSON 浮点数一同调用。默认状态下, 相当于 `float(num_str)` 。可以用于对 JSON 浮点数使用其它数据类型和解析器（比如*decimal.Decimal* ）。

parse_int ， 如果指定，将与每个要解码 JSON 整数的字符串一同调用。默认状态下，相当于 `int(num_str)` 。可以用于对 JSON 整数使用其它数据类型和语法分析程序（比如*float* ）。

parse_constant ， 如果指定，将要与以下字符串中的一个一同调用: `'-Infinity'` ， `'Infinity'` ， `'NaN'` 。如果遇到无效的 JSON 数字它会被用于抛出异常。

在 3.1 版更改: *parse_constant* 不再调用 ‘null’, ‘true’, ‘false’。

要使用自定义的*JSONDecoder* 子类，用 `cls` 指定他；否则使用*JSONDecoder* 。额外的关键词参数会通过类的构造函数传递。

如果反序列化的数据不是有效 JSON 文档，抛出*JSONDecodeError* 错误。

在 3.6 版更改: 所有可选形参现在都是仅限关键字参数。

在 3.6 版更改: *fp* 现在可以是*binary file* 。输入编码应当是 UTF-8 ， UTF-16 或者 UTF-32 。

json.**loads**(*s*, *, *encoding=None*, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

使用这个转换表 将 *s* (一个包含 JSON 文档的*str*, *bytes* 或*bytearray* 实例) 反序列化为 Python 对象。

其他参数与在*load()* 中的含义相同，只有 *encoding* 被忽略和弃用。

如果反序列化的数据不是有效 JSON 文档，抛出*JSONDecodeError* 错误。

在 3.6 版更改: *s* 现在可以为*bytes* 或*bytearray* 类型。输入编码应为 UTF-8, UTF-16 或 UTF-32。

19.2.2 编码器和解码器

class json.**JSONDecoder**(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

简单的 JSON 解码器。

默认情况下，解码执行以下翻译:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

它还将“NaN”、“Infinity”和“-Infinity”理解为它们对应的“float”值，这超出了 JSON 规范。

object_hook，如果指定，会被每个解码的 JSON 对象的结果调用，并且返回值会替代给定*dict* 。它可被用于提供自定义反序列化（比如去支持 JSON-RPC 类的暗示）。

object_pairs_hook, if specified will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of *object_pairs_hook* will be used instead of the *dict*. This feature can be used to implement custom decoders that rely on the order that the key and value pairs are decoded (for example, *collections.OrderedDict()* will remember the order of insertion). If *object_hook* is also defined, the *object_pairs_hook* takes priority.

在 3.1 版更改: 添加了对 *object_pairs_hook* 的支持。

parse_float，如果指定，将与每个要解码 JSON 浮点数一同调用。默认状态下，相当于 `float(num_str)` 。可以用于对 JSON 浮点数使用其它数据类型和解析器（比如*decimal.Decimal* ）。

parse_int，如果指定，将与每个要解码 JSON 整数的字符串一同调用。默认状态下，相当于 `int(num_str)` 。可以用于对 JSON 整数使用其它数据类型和语法分析程序（比如*float* ）。

parse_constant，如果指定，将要与以下字符串中的一个一同调用: `'-Infinity'` , `'Infinity'` , `'NaN'` 。如果遇到无效的 JSON 数字它会被用于抛出异常。

如果 *strict* 为 false （默认为 `True` ），那么控制字符将被允许在字符串内。在此上下文中的控制字符是那些编码于范围 0–31 的字符，包括 `'\t'` (制表符)，`'\n'` 和 `'\0'`。

如果反序列化的数据不是有效 JSON 文档，抛出*JSONDecodeError* 错误。

在 3.6 版更改: 所有形参现在都是仅限关键字参数。

decode(*s*)

返回 *s* 的 Python 表示形式（包含一个 JSON 文档的*str* 实例）。

如果给定的 JSON 文档无效则将引发*JSONDecodeError*。

raw_decode(*s*)
从 *s* 中解码出 JSON 文档（以 JSON 文档开头的一个*str* 对象）并返回一个 Python 表示形式为 2 元组以及指明该文档在 *s* 中结束位置的序号。

这可以用于从一个字符串解码 JSON 文档，该字符串的末尾可能有无关的数据。

class json.**JSONEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
用于 Python 数据结构的可扩展 JSON 编码器。

默认支持以下对象和类型：

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int 和 float 派生的枚举	number
True	true
False	false
None	null

在 3.4 版更改: 添加了对 int 和 float 派生的枚举类的支持

为了将其拓展至识别其他对象，需要子类化并实现*default()* 方法于另一种返回 o 的可序列化对象的方法如果可行，否则它应该调用超类实现（来引发*TypeError* ）。

如果 *skipkeys* 为假值（默认），则尝试对不是*str*, *int*, *float* 或 None 的键进行编码将会引发*TypeError*。如果 *skipkeys* 为真值，这些条目将被直接跳过。

如果 *ensure_ascii* 是 true （即默认值），输出保证将所有输入的非 ASCII 字符转义。如果 *ensure_ascii* 是 false，这些字符会原样输出。

如果 *check_circular* 为 true （默认），那么列表，字典，和自定义编码的对象在编码期间会被检查重复循环引用防止无限递归（无限递归将导致*OverflowError* ）。否则，这样进行检查。

如果 *allow_nan* 为 true （默认），那么 NaN ， Infinity ， 和 -Infinity 进行编码。此行为不符合 JSON 规范，但与大多数的基于 Javascript 的编码器和解码器一致。否则，它将是一个*ValueError* 来编码这些浮点数。

如果 *sort_keys* 为 true （默认为： False ），那么字典的输出是按照键排序；这对回归测试很有用，以确保可以每天比较 JSON 序列化。

如果 *indent* 是一个非负整数或者字符串，那么 JSON 数组元素和对象成员会被美化输出为该值指定的缩进等级。如果缩进等级为零、负数或者 ""，则只会添加换行符。None (默认值) 选择最紧凑的表达。使用一个正整数会让每一层缩进同样数量的空格。如果 *indent* 是一个字符串 (比如 "\t")，那个字符串会被用于缩进每一层。

在 3.2 版更改: 现允许使用字符串作为 *indent* 而不再仅仅是整数。

当被指定时，*separators* 应当是一个 (item_separator, key_separator) 元组。当 *indent* 为 None 时，默认值取 (', ', ': ')，否则取 (',', ': ')。为了得到最紧凑的 JSON 表达式，你应该指定其为 (',', ':') 以消除空白字符。

在 3.4 版更改: 现当 *indent* 不是 None 时，采用 (',', ': ') 作为默认值。

当 *default* 被指定时，其应该是一个函数，每当某个对象无法被序列化时它会被调用。它应该返回该对象的一个可以被 JSON 编码的版本或者引发一个*TypeError*。如果没有被指定，则会直接引发*TypeError*。

在 3.6 版更改: 所有形参现在都是仅限关键字参数。

default(*o*)
在子类中实现这种方法使其返回 *o* 的可序列化对象，或者调用基础实现（引发*TypeError*）

比如说，为了支持任意迭代器，你可以像这样实现默认设置:

```
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
       pass
   else:
       return list(iterable)
   # Let the base class default method raise the TypeError
   return json.JSONEncoder.default(self, o)
```

encode(*o*)
返回 Python *o* 数据结构的 JSON 字符串表达方式。比如说:

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)
编码给定对象 *o*，并且让每个可用的字符串表达方式。例如:

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

19.2.3 异常

exception json.**JSONDecodeError**(*msg*, *doc*, *pos*)
拥有以下额外属性的*ValueError*的子类:

msg
未格式化的错误消息。

doc
正在解析的 JSON 文档。

pos
The start index of *doc* where parsing failed.

lineno
The line corresponding to *pos*.

colno
The column corresponding to *pos*.

3.5 新版功能.

19.2.4 标准符合性和互操作性

JSON 格式由 **RFC 7159** 和 ECMA-404 指定。此段落详细讲了这个模块符合 RFC 的级别。简单来说，*JSONEncoder* 和*JSONDecoder* 子类，和明确提到的参数以外的参数，不作考虑。

此模块不严格遵循于 RFC ，它实现了一些扩展是有效的 Javascript 但不是有效的 JSON。尤其是：

- 无限和 NaN 数值是可接受的并输出；
- 对象内的重复名称是接受的，并且仅使用最后一对名称-值对。

自从 RFC 允许符合 RFC 的语法分析程序接收不符合 RFC 的输入文本以来，这个模块的解串器在默认状态下默认符合 RFC 。

字符编码

RFC 要求使用 UTF-8 ，UTF-16 ，或 UTF-32 之一来表示 JSON ，为了最大互通性推荐使用 UTF-8 。

RFC 允许，尽管不是必须的，这个模块的序列化默认设置为 *ensure_ascii=True* ，这样消除输出以便结果字符串至容纳 ASCII 字符。

ensure_ascii 参数以外，此模块是严格的按照在 Python 对象和*Unicode strings* 间的转换定义的，并且因此不能直接解决字符编码的问题。

RFC 禁止添加字符顺序标记（BOM ）在 JSON 文本的开头，这个模块的序列化器不添加 BOM 标记在它的输出上。RFC，准许 JSON 反序列化器忽略它们输入中的初始 BOM 标记，但不要求。此模块的反序列化器引发*ValueError* 当出现初始 BOM 标记。

RFC 不会明确禁止包含字节序列的 JSON 字符串这不对应有效的 Unicode 字符（比如不成对的 UTF-16 的替代物），但是它确实指出它们可能会导致互操作性问题。默认下，模块对这样的序列接受和输出（当在原始*str* 存在时）代码点。

Infinite 和 NaN 数值。

RFC 不允许 infinite 或者 NaN 数值的表达方式。尽管这样，默认情况下，此模块接受并且输出 `Infinity` ，`-Infinity`，和 `NaN` 好像它们是有效的 JSON 数字字面值

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

序列化器中，*allow_nan* 参数可用于替代这个行为。反序列化器中，*parse_constant* 参数，可用于替代这个行为。

对象中的重复名称

RFC 具体说明了在 JSON 对象里的名字应该是唯一的，但没有规定如何处理 JSON 对象中的重复名称。默认下，此模块不引发异常；作为替代，对于给定名它将忽略除姓-值对之外的所有对:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

The *object_pairs_hook* parameter can be used to alter this behavior.

顶级非对象，非数组值

过时的 **RFC 4627** 指定的旧版本 JSON 要求 JSON 文本顶级值必须是 JSON 对象或数组（Python *dict* 或*list*），并且不能是 JSON null 值，布尔值，数值或者字符串值。**RFC 7159** 移除这个限制，此模块没有并且从未在序列化器和反序列化器中实现这个限制。

无论如何，为了最大化地获取互操作性，你可能希望自己遵守该原则。

实现限制

一些 JSON 反序列化器的实现应该在以下方面做出限制:

- 接受的 JSON 文本大小
- 嵌套 JSON 对象和数组的最高水平
- JSON 数字的范围和精度
- JSON 字符串的内容和最大长度

此模块不强制执行任何上述限制，除了相关的 Python 数据类型本身或者 Python 解释器本身的限制以外。

当序列化为 JSON ，在应用中当心此类限制这可能破坏你的 JSON 。特别是，通常将 JSON 数字反序列化为 IEEE 754 双精度数字，从而受到该表示形式的范围和精度限制。这是特别相关的，当序列化非常大的 Python *int* 值时，或者当序列化“exotic”数值类型的实例时比如*decimal.Decimal* 。

19.2.5 命令行界面

** 源代码: ** Lib/json/tool.py

The *json.tool* module provides a simple command line interface to validate and pretty-print JSON objects.

如果未指定可选的 `infile` 和 `outfile` 参数，则将分别使用*sys.stdin* 和*sys.stdout*:

```
$ echo '{"json": "obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

在 3.5 版更改: 输出现在将与输入顺序保持一致。请使用*--sort-keys* 选项来将输出按照键的字母顺序排序。

命令行选项

infile
要被验证或美化打印的 JSON 文件：

```
$ python -m json.tool mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    },
    {
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

如果 *infile* 未指定，则从*sys.stdin* 读取。

outfile
将 *infile* 输出写入到给定的 *outfile*。在其他情况下写入到*sys.stdout*。

--sort-keys
将字典输出按照键的字母顺序排序。

3.5 新版功能.

-h, --help
显示帮助消息。

备注

19.3 mailcap —Mailcap 文件处理

源代码: Lib/mailcap.py

Mailcap 文件可用来配置支持 MIME 的应用例如邮件阅读器和 Web 浏览器如何响应具有不同 MIME 类型的文件。(" mailcap" 这个名称源自短语" mail capability"。) 例如，一个 mailcap 文件可能包含 `video/mpeg; xmpeg %s` 这样的行。然后，如果用户遇到 MIME 类型为 *video/mpeg* 的邮件消息或 Web 文档时，`%s` 将被替换为一个文件名（通常是一个临时文件）并且将自动启动 **xmpeg** 程序来查看该文件。

The mailcap format is documented in **RFC 1524**, "A User Agent Configuration Mechanism For Multimedia Mail Format Information," but is not an Internet standard. However, mailcap files are supported on most Unix systems.

mailcap.**findmatch**(*caps*, *MIMEtype*, *key='view'*, *filename='/dev/null'*, *plist=[]*)
返回一个 2 元组；其中第一个元素是包含所要执行命令的字符串 (它可被传递给*os.system()*)，第二个元素是对应于给定 MIME 类型的 mailcap 条目。如果找不到匹配的 MIME 类型，则将返回 `(None, None)`。

key 是所需字段的名称，它代表要执行的活动类型；默认值是 'view'，因为在最通常的情况下你只是想要查看 MIME 类型数据的正文。其他可能的值还有 'compose' 和 'edit'，分别用于想要创建给定 MIME 类型正文或修改现有正文数据的情况。请参阅 **RFC 1524** 获取这些字段的完整列表。

filename 是在命令行中用来替换 `%s` 的文件名；默认值 `'/dev/null'` 几乎肯定不是你想要的，因此通常你要通过指定一个文件名来重载它。

plist 可以是一个包含命名形参的列表；默认值只是一个空列表。列表中的每个条目必须为包含形参名称的字符串、等于号 (`'='`) 以及形参的值。Mailcap 条目可以包含形如 `%{foo}` 的命名形参，它将由名为‘foo’的形参的值所替换。例如，如果命令行 `showpartial %{id} %{number} %{total}` 是在一个 mailcap 文件中，并且 *plist* 被设为 `['id=1', 'number=2', 'total=3']`，则结果命令行将为 `'showpartial 1 2 3'`。

在 mailcap 文件中，可以指定可选的“test”字段来检测某些外部条件（例如所使用的机器架构或窗口系统）来确定是否要应用 mailcap 行。*findmatch()* 将自动检查此类条件并在检查未通过时跳过条目。

`mailcap.`**`getcaps`**`()`
返回一个将 MIME 类型映射到 mailcap 文件条目列表的字典。此字典必须被传给*findmatch()* 函数。条目会被存储为字典列表，但并不需要了解此表示形式的细节。

此信息来自在系统中找到的所有 mailcap 文件。用户的 mailcap 文件 `$HOME/.mailcap` 中的设置将覆盖系统 mailcap 文件 `/etc/mailcap`, `/usr/etc/mailcap` 和 `/usr/local/etc/mailcap` 中的设置。

一个用法示例:

```
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

19.4 mailbox —操作多种格式的邮箱

源代码： Lib/mailbox.py

本模块定义了两个类，*Mailbox* 和*Message*，用于访问和操作磁盘中的邮箱及其所包含的电子邮件。*Mailbox* 提供了类似字典的从键到消息的映射。*Message* 为*email.message* 模块的*Message* 类增加了特定格式专属的状态和行为。支持的邮箱格式有 Maildir, mbox, MH, Babyl 以及 MMDF。

参见:

模块*email* 表示和操作邮件消息。

19.4.1 Mailbox 对象

class `mailbox.`**`Mailbox`**
一个邮箱，它可以被检视和修改。

Mailbox 类定义了一个接口并且它不应被实例化。而是应该让格式专属的子类继承*Mailbox* 并且你的代码应当实例化一个特定的子类。

Mailbox 接口类似于字典，其中每个小键都有对应的消息。键是由*Mailbox* 实例发出的，它们将由实例来使用并且只对该*Mailbox* 实例有意义。键会持续标识一条消息，即使对应的消息已被修改，例如被另一条消息所替代。

可以使用 set 型方法*add()* 将消息添加到*Mailbox* 并可以使用 `del` 语句或 set 型方法*remove()* 和*discard()* 将其移除。

Mailbox 接口语义在某些值得注意的方面与字典语义有所不同。每次请求消息时，都会基于邮箱的当前状态生成一个新的表示形式（通常为*Message* 实例）。类似地，当向*Mailbox* 实例添加消息时，所提供的消息表示形式的内容将被复制。无论在哪种情况下*Mailbox* 实例都不会保留对消息表示形式的引用。

默认的*Mailbox* 迭代器会迭代消息表示形式，而不像默认的字典迭代器那样迭代键。此外，在迭代期间修改邮箱是安全且有明确定义的。在创建迭代器之后被添加到邮箱的消息将对该迭代不可见。在迭代器产出消息之前被从邮箱移除的消息将被静默地跳过，但是使用来自迭代器的键也有可能导致*KeyError* 异常，如果对应的消息后来被移除的话。

> **警告：** 在修改可能同时被其他某个进程修改的邮箱时要非常小心。用于此种任务的最安全邮箱格式是 Maildir；请尽量避免使用 mbox 之类的单文件格式进行并发写入。如果你正在修改一个邮箱，你 *必须*在读取文件中的任何消息或者执行添加或删除消息等修改操作 *之前*通过调用*lock()* 以及*unlock()* 方法来锁定它。如果未锁定邮箱则可能导致丢失消息或损坏整个邮箱的风险。

Mailbox 实例具有下列方法:

add(*message*)
将 *message* 添加到邮箱并返回分配给它的键。

形参 *message* 可以是*Message* 实例、*email.message.Message* 实例、字符串、字节串或文件类对象（应当以二进制模式打开）。如果 *message* 是适当的格式专属*Message* 子类的实例（例如，如果它是一个*mboxMessage* 实例而这是一个*mbox* 实例），将使用其格式专属的信息。在其他情况下，则会使用合理的默认值作为格式专属的信息。

在 3.2 版更改: 增加了对二进制输入的支持。

remove(*key*)
__delitem__(*key*)
discard(*key*)
从邮箱中删除对应于 *key* 的消息。

当消息不存在时，如果此方法是作为*remove()* 或*__delitem__()* 调用则会引发*KeyError* 异常，而如果此方法是作为*discard()* 调用则不会引发异常。如果下层邮箱格式支持来自其他进程的并发修改则*discard()* 的行为可能是更为适合的。

__setitem__(*key*, *message*)
将 *key* 所对应的消息替换为 *message*。如果没有与 *key* 所对应的消息则会引发*KeyError* 异常。

与*add()* 一样，形参 *message* 可以是*Message* 实例、*email.message.Message* 实例、字符串、字节串或文件类对象（应当以二进制模式打开）。如果 *message* 是适当的格式专属*Message* 子类的实例（举例来说，如果它是一个*mboxMessage* 实例而这是一个*mbox* 实例），将使用其格式专属的信息。在其他情况下，当前与 *key* 所对应的消息的格式专属信息则会保持不变。

iterkeys()
keys()
如果通过*iterkeys()* 调用则返回一个迭代所有键的迭代器，或者如果通过*keys()* 调用则返回一个键列表。

itervalues()
__iter__()
values()
如果通过*itervalues()* 或*__iter__()* 调用则返回一个迭代所有消息的表示形式的迭代器，或者如果通过*values()* 调用则返回一个由这些表示形式组成的列表。消息会被表示为适当的格式专属*Message* 子类的实例，除非当*Mailbox* 实际被初始化时指定了自定义的消息工厂函数。

> **注解：** *__iter__()* 的行为与字典不同，后者是对键进行迭代。

iteritems()
items()
如果通过*iteritems()* 调用则返回一个迭代 (*key*, *message*) 对的迭代器，其中 *key* 为键而 *message*

为消息的表示形式，或者如果通过 or return a list of such pairs if called as *items()* 调用则返回一个由这种键值对组成的列表。消息会被表示为适当的格式专属*Message* 子类的实例，除非当*Mailbox* 实例被初始化时指定了自定义的消息工厂函数。

get(*key*, *default=None*)

__getitem__(*key*)

返回对应于 *key* 的消息的表示形式。当对应的消息不存在时，如果通过*get()* 调用则返回 *default* 而如果通过*__getitem__()* 调用此方法则会引发*KeyError* 异常。消息会被表示为适当的格式专属*Message* 子类的实例，除非当*Mailbox* 实例被初始化时指定了自定义的消息工厂函数。

get_message(*key*)

将对应于 *key* 的消息的表示形式作为适当的格式专属*Message* 子类的实例返回，或者如果对应的消息不存在则会引发*KeyError* 异常。

get_bytes(*key*)

返回对应于 *key* 的消息的字节表示形式，或者如果对应的消息不存在则会引发*KeyError* 异常。

3.2 新版功能.

get_string(*key*)

返回对应于 *key* 的消息的字符串表示形式，或者如果对应的消息不存在则会引发*KeyError* 异常。消息是通过*email.message.Message* 处理来将其转换为纯 7bit 表示形式的。

get_file(*key*)

返回对应于 *key* 的消息的文件类表示形式，或者如果对应的消息不存在则会引发*KeyError* 异常。文件类对象的行为相当于以二进制模式打开。当不再需要此文件时应当将其关闭。

在 3.2 版更改: 此文件对象实际上是二进制文件；之前它被不正确地以文本模式返回。并且，此文件类对象现在还支持上下文管理协议：你可以使用 `with` 语句来自动关闭它。

注解: 不同于其他消息表示形式，文件类表示形式并不一定独立于创建它们的*Mailbox* 实例或下层的邮箱。每个子类都会提供更具体的文档。

__contains__(*key*)

如果 *key* 有对应的消息则返回 `True`，否则返回 `False`。

__len__()

返回邮箱中消息的数量。

clear()

从邮箱中删除所有消息。

pop(*key*, *default=None*)

返回对应于 *key* 的消息的表示形式并删除该消息。如果对应的消息不存在则返回 *default*。消息会被表示为适当的格式专属*Message* 子类的实例，除非当*Mailbox* 实例被初始化时指定了自定义的消息工厂函数。

popitem()

返回一个任意的 (*key*, *message*) 对，其中 *key* 为键而 *message* 为消息的表示形式，并删除对应的消息。如果邮箱为空，则会引发*KeyError* 异常。消息会被表示为适当的格式专属*Message* 子类的实例，除非当*Mailbox* 实例被初始化时指定了自定义的消息工厂函数。

update(*arg*)

形参 *arg* 应当是 *key* 到 *message* 的映射或 (*key*, *message*) 对的可迭代对象。用来更新邮箱以使得对于每个给定的 *key* 和 *message*，与 *key* 相对应的消息会被设为 *message*，就像通过使用*__setitem__()* 一样。类似于*__setitem__()*，每个 *key* 都必须在邮箱中有一个对应的消息否则将会引发*KeyError* 异常，因此在通常情况下将 *arg* 设为*Mailbox* 实例是不正确的。

> **注解：** 与字典不同，关键字参数是不受支持的。

flush()
将所有待定的更改写入到文件系统。对于某些*Mailbox* 子类来说，更改总是被立即写入因而*flush()* 并不会做任何事，但您仍然应当养成调用此方法的习惯。

lock()
在邮箱上获取一个独占式咨询锁以使其他进程知道不能修改它。如果锁无法被获取则会引发*ExternalClashError*。所使用的具体锁机制取决于邮箱的格式。在对邮箱内容进行任何修改之前你应当 *总是* 锁定它。

unlock()
释放邮箱上的锁，如果存在的话。

close()
刷新邮箱，如果必要则将其解锁。并关闭所有打开的文件。对于某些*Mailbox* 子类来说，此方法并不会做任何事。

Maildir

class mailbox.**Maildir**(*dirname*, *factory=None*, *create=True*)
Mailbox 的一个子类，用于 Maildir 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*MaildirMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

使用 *dirname* 这个名称而不使用 *path* 是出于历史原因。

Maildir 是一种基于目录的邮箱格式，它是针对 qmail 邮件传输代理而发明的，现在也得到了其他程序的广泛支持。Maildir 邮箱中的消息存储在一个公共目录结构中的单独文件内。这样的设计允许 Maildir 邮箱被多个彼此无关的程序访问和修改而不会导致数据损坏，因此文件锁定操作是不必要的。

Maildir 邮箱包含三个子目录，分别是: `tmp`, `new` 和 `cur`。消息会不时地在 `tmp` 子目录中创建然后移至 `new` 子目录来结束投递。后续电子邮件客户端可能将消息移至 `cur` 子目录并将有关消息状态的信息存储在附带到其文件名的特殊“info”小节中。

Courier 电子邮件传输代理所引入的文件夹风格也是受支持的。主邮箱中任何子目录只要其名称的第一个字符是 `'.'` 就会被视为文件夹。文件夹名称会被*Maildir* 表示为不带前缀 `'.'` 的形式。每个文件夹自身都是一个 Maildir 邮箱但不应包含其他文件夹。逻辑嵌套关系是使用 `'.'` 来划定层级，例如“Archived.2005.07”。

> **注解：** Maildir 规范要求使用在特定消息文件名中使用冒号 (`':'`)。但是，某些操作系统不允许将此字符用于文件名，如果你希望在这些操作系统上使用类似 Maildir 的格式，你应当指定改用另一个字符。叹号 (`'!'`) 是一个受欢迎的选择。例如:

```
import mailbox
mailbox.Maildir.colon = '!'
```

`colon` 属性也可以在每个实例上分别设置。

Maildir 实例具有*Mailbox* 的所有方法及下列附加方法:

list_folders()
返回所有文件夹名称的列表。

get_folder(*folder*)
返回表示名称为 *folder* 的文件夹的*Maildir* 实例。如果文件夹不存在则会引发*NoSuchMailboxError* 异常。

add_folder(*folder*)
创建名称为 *folder* 的文件夹并返回表示它的*Maildir* 实例。

remove_folder(*folder*)
删除名称为 *folder* 的文件夹。如果文件夹包含任何消息，则将引发*NotEmptyError* 异常且该文件夹将不会被删除。

clean()
从邮箱中删除最近 36 小时内未被访问过的临时文件。Maildir 规范要求邮件阅读程序应当时常进行此操作。

Maildir 所实现的某些方法*Mailbox* 值得进行特别的说明:

add(*message*)
__setitem__(*key*, *message*)
update(*arg*)

> **警告:** 这些方法会基于当前进程 ID 来生成唯一文件名。当使用多线程时，可能发生未被检测到的名称冲突并导致邮箱损坏，除非是对线程进行协调以避免使用这些方法同时操作同一个邮箱。

flush()
对 Maildir 邮箱的所有更改都会立即被应用，所以此方法并不会做任何事情。

lock()
unlock()
Maildir 邮箱不支持（或要求）锁定操作，所以此方法并不会做任何事情。

close()
Maildir 实例不保留任何打开的文件并且下层的邮箱不支持锁定操作，所以此方法不会做任何事情。

get_file(*key*)
根据主机平台的不同，当返回的文件保持打开状态时可能无法修改或移除下层的消息。

参见:

maildir man page from qmail The original specification of the format.

使用 maildir 格式 Maildir 发明者对它的说明。包括已更新的名称创建方案和“info”语义的相关细节。

Courier 上的 maildir 指南页面 Another specification of the format. Describes a common extension for supporting folders.

mbox

class mailbox.**mbox**(*path*, *factory=None*, *create=True*)

Mailbox 的子类，用于 mbox 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*mboxMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

mbox 格式是在 Unix 系统上存储电子邮件的经典格式。mbox 邮箱中的所有消息都存储在一个单独文件中，每条消息的开头由前五个字符为“From ”的行来指明。

还有一些 mbox 格式变种对原始格式中发现的缺点做了改进，*mbox* 只实现原始格式，有时被称为 *mboxo*。这意味着当存储消息时 *Content-Length* 标头如果存在则会被忽略并且消息体中出现于行开头的任何“From ”会被转换为“>From “，但是当读取消息时“>From ”则不会被转换为“From “。

mbox 所实现的某些*Mailbox* 方法值得进行特别的说明:

get_file(*key*)
在*mbox* 实例上调用 `flush()` 或 `close()` 之后再使用文件可能产生无法预料的结果或者引发异常。

lock()
unlock()
使用三种锁机制—dot 锁，以及可能情况下的 `flock()` 和 `lockf()` 系统调用。

参见:

mbox man page from qmail A specification of the format and its variations.

tin 上的 mbox 指南页面 Another specification of the format, with details on locking.

在 Unix 上配置 Netscape Mail: 为何 Content-Length 格式是不好的 使用原始 mbox 格式而非其变种的一些理由。

“mbox”是由多个彼此不兼容的邮箱格式构成的家族 有关 mbox 变种的历史。

MH

class mailbox.**MH**(*path*, *factory=None*, *create=True*)

Mailbox 的子类，用于 MH 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*MHMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

MH 是一种基于目录的邮箱格式，它是针对 MH Message Handling System 电子邮件用户代理而发明的。在 MH 邮箱的每条消息都放在单独文件中。MH 邮箱中除了邮件消息还可以包含其他 MH 邮箱 (称为 *文件夹*)。文件夹可以无限嵌套。MH 邮箱还支持 *序列*，这是一种命名列表，用来对消息进行逻辑分组而不必将其移入子文件夹。序列是在每个文件夹中名为 `.mh_sequences` 的文件内定义的。

MH 类可以操作 MH 邮箱，但它并不试图模拟 **mh** 的所有行为。特别地，它并不会修改 `context` 或 `.mh_profile` 文件也不会受其影响，这两个文件是 **mh** 用来存储状态和配置数据的。

MH 实例具有*Mailbox* 的所有方法及下列附加方法:

list_folders()
返回所有文件夹名称的列表。

get_folder(*folder*)
返回表示名称为 *folder* 的文件夹的*MH* 实例。如果文件夹不存在则会引发*NoSuchMailboxError* 异常。

add_folder(*folder*)
创建名称为 *folder* 的文件夹并返回表示它的*MH* 实例。

remove_folder(*folder*)
删除名称为 *folder* 的文件夹。如果文件夹包含任何消息，则将引发*NotEmptyError* 异常且该文件夹将不会被删除。

get_sequences()
返回映射到键列表的序列名称字典。如果不存在任何序列，则返回空字典。

set_sequences(*sequences*)
根据由映射到键列表的名称组成的字典 *sequences* 来重新定义邮箱中的序列，该字典与*get_sequences()* 返回值的形式一样。

pack()
根据需要重命名邮箱中的消息以消除序号中的空缺。序列列表中的条目会做相应的修改。

注解： 已发送的键会因此操作而失效并且不应当被继续使用。

MH 所实现的某些*Mailbox* 方法值得进行特别的说明:

remove(*key*)
__delitem__(*key*)
discard(*key*)
这些方法会立即删除消息。通过在名称前加缀一个冒号作为消息删除标记的 MH 惯例不会被使用。

lock()
unlock()
使用三种锁机制—dot 锁，以及可能情况下下 `flock()` 和 `lockf()` 系统调用。对于 MH 邮箱来说，锁定邮箱意味着锁定 `.mh_sequences` 文件，并且仅在执行会影响单独消息文件的操作期间锁定单独消息文件。

get_file(*key*)
根据主机平台的不同，当返回的文件保持打开状态时可能无法移除下层的消息。

flush()
对 MH 邮箱的所有更改都会立即被应用，所以此方法并不会做任何事情。

close()
MH 实例不保留任何打开的文件，所以此方法等价于*unlock()*。

参见:

nmh - Message Handling System **nmh** 的主页，这是原始 **mh** 的更新版本。

MH & nmh: Email for Users & Programmers 使用 GPL 许可证的介绍 **mh** 与 **nmh** 的图书，包含有关该邮箱格式的各种信息。

Babyl

class mailbox.**Babyl**(*path*, *factory=None*, *create=True*)
Mailbox 的子类，用于 Babyl 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*BabylMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

Babyl 是 Rmail 电子邮箱用户代理所使用单文件邮箱格式，包括在 Emacs 中。每条消息的开头由一个包含 Control-Underscore (`'\037'`) 和 Control-L (`'\014'`) 这两个字符的行来指明。消息的结束由下一条消息的开头来指明，或者当为最后一条消息时则由一个包含 Control-Underscore (`'\037'`) 字符的行来指明。

Babyl 邮箱中的消息带有两组标头：原始标头和所谓的可见标头。可见标头通常为原始标头经过重格式化和删减以更易读的子集。Babyl 邮箱中的每条消息都附带了一个 标签列表，即记录消息相关额外信息的短字符串，邮箱中所有的用户定义标签列表会存储于 Babyl 的选项部分。

Babyl 实例具有*Mailbox* 的所有方法及下列附加方法:

get_labels()
返回邮箱中使用的所有用户定义标签名称的列表。

> **注解:** 邮箱中存在哪些标签会通过检查实际的消息而非查询 Babyl 选项部分的标签列表，但 Babyl 选项部分会在邮箱被修改时更新。

Babyl 所实现的某些*Mailbox* 方法使得进行特别的说明:

get_file(*key*)
在 Babyl 邮箱中，消息的标头并不是与消息体存储在一起的。要生成文件类表示形式，标头和消息体会被一起拷贝到一个*io.BytesIO* 实例中，它具有与文件相似的 API。因此，文件类对象实际上独立于下层邮箱，但与字符串表达形式相比并不会更节省内存。

lock()
unlock()
使用三种锁机制—dot 锁，以及可能情况下的 `flock()` 和 `lockf()` 系统调用。

参见:

Format of Version 5 Babyl Files Babyl 格式的规格说明。

Reading Mail with Rmail Rmail 的帮助手册，包含了有关 Babyl 语义的一些信息。

MMDF

class mailbox.**MMDF**(*path*, *factory=None*, *create=True*)
Mailbox 的子类，用于 MMDF 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*MMDFMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

MMDF 是一种专用于电子邮件传输代理 Multichannel Memorandum Distribution Facility 的单文件邮箱格式。每条消息使用与 mbox 消息相同的形式，但其前后各有包含四个 Control-A (`'\001'`) 字符的行。与 mbox 格式一样，每条消息的开头由一个前五个字符为“From ”的行来指明，但当存储消息时额外出现的“From ”不会被转换为“>From ”因为附加的消息分隔符可防止将这些内容误认为是后续消息的开头。

MMDF 所实现的某些*Mailbox* 方法值得进行特别的说明:

get_file(*key*)
在*MMDF* 实例上调用 `flush()` 或 `close()` 之后再使用文件可能产生无法预料的结果或者引发异常。

lock()
unlock()
使用三种锁机制—dot 锁，以及可能情况下的 `flock()` 和 `lockf()` 系统调用。

参见:

tin 上的 mmdf 指南页面 MMDF 格式的规格说明，来自新闻阅读器 tin 的文档。

MMDF 一篇描述 Multichannel Memorandum Distribution Facility 的维基百科文章。

19.4.2 Message 对象

class mailbox.**Message**(*message=None*)

email.message 模块的*Message* 的子类。*mailbox.Message* 的子类添加了特定邮箱格式专属的状态和行为。

如果省略了 *message*，则新实例会以默认的空状态被创建。如果 *message* 是一个*email.message.Message* 实例，其内容会被拷贝；此外，如果 *message* 是一个*Message* 实例，则任何格式专属信息会尽可能地被拷贝。如果 *message* 是一个字符串、字节串或文件，则它应当包含兼容 **RFC 2822** 的消息，该消息会被读取和解析。文档应当以二进制模式打开，但文本模式的文件也会被接受以向下兼容。

各个子类所提供的格式专属状态和行为各有不同，但总的来说只有那些不仅限于特定邮箱的特性才会被支持（虽然这些特性可能专属于特定邮箱格式）。例如，例如，单文件邮箱格式的文件偏移量和基于目录的邮箱格式的文件名都不会被保留，因为它们都仅适用于对应的原始邮箱。但消息是否已被用户读取或标记为重要等状态则会被保留，因为它们适用于消息本身。

不要求用*Message* 实例来表示使用*Mailbox* 实例所提取到的消息。在某些情况下，生成*Message* 表示形式所需的时间和内存空间可能是不可接受的。对于此类情况，*Mailbox* 实例还提供了字符串和文件类表示形式，并可在初始化*Mailbox* 实例时指定自定义的消息工厂函数。

MaildirMessage

class mailbox.**MaildirMessage**(*message=None*)

具有 Maildir 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

通常，邮件用户代理应用程序会在用户第一次打开并关闭邮箱之后将 new 子目录中的所有消息移至 cur 子目录，将这些消息记录为旧消息，无论它们是否真的已被阅读。cur 下的每条消息都有一个“info”部分被添加到其文件名中以存储有关其状态的信息。(某些邮件阅读器还会把“info”部分也添加到 new 下的消息中。)“info”部分可以采用两种形式之一：它可能包含“2,”后面跟一个经标准化的旗标列表（例如“2,FR”）或者它可能包含“1,”后面跟所谓的实验性信息。Maildir 消息的标准旗标如下:

标志	含义	解释
D	草稿	正在撰写中
F	已标记	标记为重要
P	已读	转发，重新发送或退回
R	已回复	回复给
S	查看	读取
T	已删除	标记为以后删除

MaildirMessage 实例提供以下方法:

get_subdir()

返回“new”(如果消息应当被存储在 new 子目录下) 或者“cur”(如果消息应当被存储在 cur 子目录下)。

> **注解:** 消息通常会在其邮箱被访问后被从 new 移至 cur，无论该消息是否已被阅读。如果 msg.get_flags()“ 中的 "S" 为 ``True 则说明消息 msg 已被阅读。

set_subdir(*subdir*)

设置消息应当被存储到的子目录。形参 *subdir* 必须为“new”或“cur”。

get_flags()

返回一个指明当前所设旗标的字符串。如果消息符合标准的 Maildir 格式，则结果为零或按字母顺序各自出现一次的 'D', 'F', 'P', 'R', 'S' 和 'T' 的拼接。如果未设任何旗标或者如果“info”包含实验性语义则返回空字符串。

set_flags(*flags*)
设置由 *flags* 所指定的旗标并重置所有其它旗标。

add_flag(*flag*)
设置由 *flag* 所指明的旗标而不改变其他旗标。要一次性添加一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。当前“info”会被覆盖，无论它是否只包含实验性信息而非旗标。

remove_flag(*flag*)
重置由 *flag* 所指明的旗标而不改变其他旗标。要一次性移除一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。如果“info”包含实验性信息而非旗标，则当前的“info”不会被修改。

get_date()
以表示 Unix 纪元秒数的浮点数形式返回消息的发送日期。

set_date(*date*)
将消息的发送日期设为 *date*，一个表示 Unix 纪元秒数的浮点数。

get_info()
返回一个包含消息的“info”的字符串。这适用于访问和修改实验性的“info”(即不是由旗标组成的列表)。

set_info(*info*)
将“info”设为 *info*，这应当是一个字符串。

当一个*MaildirMessage* 实例基于*mboxMessage* 或*MMDFMessage* 实例被创建时，将会忽略 *Status* 和 *X-Status* 标头并进行下列转换:

结果状态	*mboxMessage* 或*MMDFMessage* 状态
“cur”子目录	O 标记
F 标记	F 标记
R 标记	A 标记
S 标记	R 标记
T 标记	D 标记

当一个*MaildirMessage* 实例基于*MHMessage* 实例被创建时，将进行下列转换:

结果状态	*MHMessage* 状态
“cur”子目录	“unseen”序列
“cur”子目录和 S 旗标	非“unseen”序列
F 标记	“flagged”序列
R 标记	“replied”序列

当一个*MaildirMessage* 实例基于*BabylMessage* 实例被创建时，将进行下列转换:

结果状态	*BabylMessage* 状态
“cur”子目录	“unseen”标签
“cur”子目录和 S 旗标	非“unseen”标签
P 标记	“forwarded”或“resent”标签
R 标记	“answered”标签
T 标记	“deleted”标签

mboxMessage

class mailbox.**mboxMessage**(*message=None*)

具有 mbox 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

mbox 邮箱中的消息会一起存储在单个文件中。发件人的信封地址和发送时间通常存储在指明每条消息的起始的以“From ”打头的行中，不过在 mbox 的各种实现之间此数据的确切格式具有相当大的差异。指明消息状态的各种旗标，例如是否已读或标记为重要等等通常存储在 *Status* 和 *X-Status* 标头中。

传统的 mbox 消息旗标如下:

标志	含义	解释
R	读取	读取
O	旧的	以前由 MUA 检测
D	已删除	标记为以后删除
F	已标记	标记为重要
A	已回复	回复给

“R”和“O”旗标存储在 *Status* 标头中，而“D”，“F”和“A”旗标存储在 *X-Status* 标头中。旗标和标头通常会按上述顺序显示。

mboxMessage 实例提供了下列方法:

get_from()

返回一个表示在 mbox 邮箱中标记消息起始的“From ”行的字符串。开头的“From ”和末尾的换行符会被去除。

set_from(*from_*, *time_=None*)

将“From ”行设为 *from_*，这应当被指定为不带开头的“From ”或末尾的换行符。为方便起见，可以指定 *time_* 并将经过适当的格式化再添加到 *from_*。如果指定了 *time_*，它应当是一个*time.struct_time* 实例，适合传入*time.strftime()* 的元组或者 `True`(以使用*time.gmtime()*)。

get_flags()

返回一个指明当前所设旗标的字符串。如果消息符合规范格式，则结果为零或各自出现一次的 `'R'`, `'O'`, `'D'`, `'F'` 和 `'A'` 按上述顺序的拼接。

set_flags(*flags*)

设置由 *flags* 所指明的旗标并重启所有其他旗标。形参 *flags* 应当为零或各自出现多次的 `'R'`, `'O'`, `'D'`, `'F'` 和 `'A'` 按任意顺序的拼接。

add_flag(*flag*)

设置由 *flag* 所指明的旗标而不改变其他旗标。要一次性添加一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

remove_flag(*flag*)

重置由 *flag* 所指明的旗标而不改变其他旗标。要一次性移除一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

当一个*mboxMessage* 实例基于*MaildirMessage* 实例被创建时，”From ”行会基于*MaildirMessage* 实例的发送时间被生成，并进行下列转换:

结果状态	*MaildirMessage* 状态
R 标记	S 标记
O 标记	“cur”子目录
D 标记	T 标记
F 标记	F 标记
A 标记	R 标记

当一个*mboxMessage* 实例基于*MHMessage* 实例被创建时，将进行下列转换:

结果状态	*MHMessage* 状态
R 标记和 O 标记	非“unseen”序列
O 标记	“unseen”序列
F 标记	“flagged”序列
A 标记	“replied”序列

当一个*mboxMessage* 实例基于*BabylMessage* 实例被创建时，将进行下列转换:

结果状态	*BabylMessage* 状态
R 标记和 O 标记	非“unseen”标签
O 标记	“unseen”标签
D 标记	“deleted”标签
A 标记	“answered”标签

当一个*Message* 实例基于*MMDFMessage* 实例被创建时，” From ” 行会被拷贝并直接对应所有旗标。

结果状态	*MMDFMessage* 状态
R 标记	R 标记
O 标记	O 标记
D 标记	D 标记
F 标记	F 标记
A 标记	A 标记

MHMessage

class mailbox.**MHMessage**(*message=None*)

具有 MH 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

MH 消息不支持传统意义上的标记或旗标，但它们支持序列，即对任意消息的逻辑分组。某些邮件阅读程序 (但不包括标准 **mh** 和 **nmh**) 以与其他格式使用旗标类似的方式来使用序列，如下所示:

序列	解释
未读	未读取，但先前被 MUA 检测到
已回复	回复给
已标记	标记为重要

MHMessage 实例提供了下列方法:

get_sequences()

返回一个包含此消息的序列的名称的列表。

set_sequences(*sequences*)

设置包含此消息的序列的列表。

add_sequence(*sequence*)

将 *sequence* 添加到包含此消息的序列的列表。

remove_sequence(*sequence*)

将 *sequence* 从包含此消息的序列的列表中移除。

当一个*MHMessage* 实例基于*MaildirMessage* 实例被创建时，将进行下列转换:

结果状态	*MaildirMessage* 状态
“unseen” 序列	非 S 标记
“replied” 序列	R 标记
“flagged” 序列	F 标记

当一个*MHMessage* 实例基于*mboxMessage* 或*MMDFMessage* 实例被创建时，将会忽略 *Status* 和 *X-Status* 标头并进行下列转换:

结果状态	*mboxMessage* 或*MMDFMessage* 状态
“unseen” 序列	非 R 标记
“replied” 序列	A 标记
“flagged” 序列	F 标记

当一个*MHMessage* 实例基于*BabylMessage* 实例被创建时，将进入下列转换:

结果状态	*BabylMessage* 状态
“unseen” 序列	“unseen” 标签
“replied” 序列	“answered” 标签

BabylMessage

class mailbox.**BabylMessage**(*message=None*)

具有 Babyl 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

某些消息标签被称为 *属性*，根据惯例被定义为具有特殊的含义。这些属性如下所示:

标签	解释
未读	未读取，但先前被 MUA 检测到
deleted	标记为以后删除
filed	复制到另一个文件或邮箱
answered	回复给
forwarded	已转发
edited	由用户修改
resent	已重发

默认情况下，Rmail 只显示可见标头。不过*BabylMessage* 类会使用原始标头因为它们更完整。如果需要可以显式地访问可见标头。

BabylMessage 实例提供了下列方法:

get_labels()

返回邮件上的标签列表。

set_labels(*labels*)

将消息上的标签列表设置为 *labels* 。

add_label(*label*)

将 *label* 添加到消息上的标签列表中。

remove_label(*label*)

从消息上的标签列表中删除 *label* 。

get_visible()
返回一个*Message* 实例，其标头为消息的可见标头而其消息体为空。

set_visible(*visible*)
将消息的可见标头设为与 *message* 中的标头一致。形参 *visible* 应当是一个*Message* 实例，*email.message.Message* 实例，字符串或文件类对象（且应当以文本模式打开）。

update_visible()
当一个*BabylMessage* 实例的原始标头被修改时，可见标头不会自动进行对应修改。此方法将按以下方式更新可见标头：每个具有对应原始标头的可见标头会被设为原始标头的值，每个没有对应原始标头的可见标头会被移除，而任何存在于原始标头但不存在于可见标头中的 *Date*, *From*, *Reply-To*, *To*, *CC* 和 *Subject* 会被添加至可见标头。

当一个*BabylMessage* 实例基于*MaildirMessage* 实例被创建时，将进行下列转换:

结果状态	*MaildirMessage* 状态
“unseen” 标签	非 S 标记
“deleted” 标签	T 标记
“answered” 标签	R 标记
“forwarded” 标签	P 标记

当一个*BabylMessage* 实例基于*mboxMessage* 或*MMDFMessage* 实例被创建时，将会忽略 *Status* 和 *X-Status* 标头并进入下列转换:

结果状态	*mboxMessage* 或*MMDFMessage* 状态
“unseen” 标签	非 R 标记
“deleted” 标签	D 标记
“answered” 标签	A 标记

当一个*BabylMessage* 实例基于*MHMessage* 实例被创建时，将进入下列转换:

结果状态	*MHMessage* 状态
“unseen” 标签	“unseen” 序列
“answered” 标签	“replied” 序列

MMDFMessage

class mailbox.**MMDFMessage**(*message=None*)
具有 MMDF 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

与 mbox 邮箱中的消息类似，MMDF 消息会与将发件人的地址和发送日期作为以 “From ” 打头的初始行一起存储。同样地，指明消息状态的旗标通常存储在 *Status* 和 *X-Status* 标头中。

传统的 MMDF 消息旗标与 mbox 消息的类似，如下所示:

标志	含义	解释
R	读取	读取
O	旧的	以前由 MUA 检测
D	已删除	标记为以后删除
F	已标记	标记为重要
A	已回复	回复给

“R” 和 “O” 旗标存储在 *Status* 标头中，而 “D”，“F” 和 “A” 旗标存储在 *X-Status* 标头中。旗标和标头通常会按上述顺序显示。

MMDFMessage 实例提供了下列方法，与*mboxMessage* 所提供的类似:

get_from()
返回一个表示在 mbox 邮箱中标记消息起始的“From ”行的字符串。开头的“From ”和末尾的换行符会被去除。

set_from(*from_*, *time_*=*None*)
将“From ”行设为 *from_*，这应当被指定为不带开头的“From ”或末尾的换行符。为方便起见，可以指定 *time_* 并将经过适当的格式化再添加到 *from_*。如果指定了 *time_*，它应当是一个*time.struct_time* 实例，适合传入*time.strftime()* 的元组或者 `True`(以使用*time.gmtime()*)。

get_flags()
返回一个指明当前所设旗标的字符串。如果消息符合规范格式，则结果为零或各自出现一次的 `'R'`, `'O'`, `'D'`, `'F'` 和 `'A'` 按上述顺序的拼接。

set_flags(*flags*)
设置由 *flags* 所指明的旗标并重启所有其他旗标。形参 *flags* 应当为零或各自出现多次的 `'R'`, `'O'`, `'D'`, `'F'` 和 `'A'` 按任意顺序的拼接。

add_flag(*flag*)
设置由 *flag* 所指明的旗标而不改变其他旗标。要一次性添加一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

remove_flag(*flag*)
重置由 *flag* 所指明的旗标而不改变其他旗标。要一次性移除一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

当一个*MMDFMessage* 实例基于*MaildirMessage* 实例被创建时，”From ”行会基于*MaildirMessage* 实例的发送日期被生成，并进入下列转换:

结果状态	*MaildirMessage* 状态
R 标记	S 标记
O 标记	“cur”子目录
D 标记	T 标记
F 标记	F 标记
A 标记	R 标记

当一个*MMDFMessage* 实例基于*MHMessage* 实例被创建时，将进入下列转换:

结果状态	*MHMessage* 状态
R 标记和 O 标记	非“unseen”序列
O 标记	“unseen”序列
F 标记	“flagged”序列
A 标记	“replied”序列

当一个*MMDFMessage* 实例基于*BabylMessage* 实例被创建时，将进行下列转换:

结果状态	*BabylMessage* 状态
R 标记和 O 标记	非“unseen”标签
O 标记	“unseen”标签
D 标记	“deleted”标签
A 标记	“answered”标签

当一个*MMDFMessage* 实例基于*mboxMessage* 实例被创建时，”From ”行会被拷贝并直接对应所有旗标:

结果状态	*mboxMessage* 状态
R 标记	R 标记
O 标记	O 标记
D 标记	D 标记
F 标记	F 标记
A 标记	A 标记

19.4.3 异常

mailbox 模块中定义了下列异常类:

exception mailbox.**Error**
所有其他模块专属异常的基类。

exception mailbox.**NoSuchMailboxError**
在期望获得一个邮箱但未找到时被引发，例如当使用不存在的路径来实例化一个*Mailbox* 子类时 (且将 *create* 形参设为 `False`)，或是当打开一个不存在的路径时。

exception mailbox.**NotEmptyError**
在期望一个邮箱为空但不为空时被引发，例如当删除一个包含消息的文件夹时。

exception mailbox.**ExternalClashError**
在某些邮箱相关条件超出了程序控制范围导致其无法继续运行时被引发，例如当要获取的锁已被另一个程序获取时，或是当要生成的唯一性文件名已存在时。

exception mailbox.**FormatError**
在某个文件中的数据无法被解析时被引发，例如当一个*MH* 实例尝试读取已损坏的 `.mh_sequences` 文件时。

19.4.4 例子

一个打印指定邮箱中所有消息的主题的简单示例:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

要将所有邮件从 Babyl 邮箱拷贝到 MH 邮箱，请转换所有可转换的格式专属信息:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

这个示例将来自多个邮件列表的邮件分类放入不同的邮箱，小心避免由于其他程序的并发修改导致的邮件损坏，由于程序中断导致的邮件丢失，或是由于邮箱中消息格式错误导致的意外终止:

```
import mailbox
import email.errors
```

(下页继续)

(续上页)

```
list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop looking.

for box in boxes.itervalues():
    box.close()
```

19.5 mimetypes —映射文件夹到 MIME 类型

源代码: Lib/mimetypes.py

---

mimetypes 模块可以在文件名或 URL 和关联到文件扩展名的 MIME 类型之间执行转换。所提供的转换包括从文件名到 MIME 类型和从 MIME 类型到文件扩展名；后一种转换不支持编码格式。

该模块提供了一个类和一些便捷函数。这些函数是该模块通常的接口，但某些应用程序可能也会希望使用类。

下列函数提供了此模块的主要接口。如果此模块尚未被初始化，它们将会调用*init()*，如果它们依赖于*init()* 所设置的信息的话。

mimetypes.**guess_type**(*url*, *strict=True*)

Guess the type of a file based on its filename or URL, given by *url*. The return value is a tuple `(type, encoding)` where *type* is `None` if the type can't be guessed (missing or unknown suffix) or a string of the form `'type/subtype'`, usable for a MIME *content-type* header.

encoding 在无编码格式时为 None，或者为程序所用的编码格式 (例如 **compress** 或 **gzip**)。它可以作为 *Content-Encoding* 标头，但 **不可**作为 *Content-Transfer-Encoding* 标头。映射是表格驱动的。编码格式前缀对大小写敏感；类型前缀会先以大小写敏感方式检测再以大小写不敏感方式检测。

可选的 *strict* 参数是一个旗标，指明要将已知 MIME 类型限制在 IANA 已注册 的官方类型之内。当 *strict* 为 True 时（默认值），则仅支持 IANA 类型；当 *strict* 为 False 时，则还支持某些附加的非标准但常用的 MIME 类型。

mimetypes.**guess_all_extensions**(*type*, *strict=True*)
根据由 *type* 给出的文件 MIME 类型猜测其扩展名。返回值是由所有可能的文件扩展名组成的字符串列表，包括开头的点号 ('.')。这些扩展名不保证能关联到任何特定的数据流，但是将会由*guess_type()* MIME 类型 *type*。

可选的 *strict* 参数具有与*guess_type()* 函数一致的含义。

mimetypes.**guess_extension**(*type*, *strict=True*)
根据由 *type* 给出的文件 MIME 类型猜测其扩展名。返回值是一个表示文件扩展名的字符串，包括开头的点号 ('.')。该扩展名不保证能关联到任何特定的数据流，但是将会由*guess_type()* 映射到 MIME 类型 *type*。如果不能猜测出 *type* 的扩展名，则将返回 None。

可选的 *strict* 参数具有与*guess_type()* 函数一致的含义。

有一些附加函数和数据项可被用于控制此模块的行为。

mimetypes.**init**(*files=None*)
初始化内部数据结构。*files* 如果给出则必须是一个文件名序列，它应当被用于协助默认的类型映射。如果省略则要使用的文件名会从*knownfiles* 中获取；在 Windows 上，将会载入当前注册表设置。在 *files* 或*knownfiles* 中指定的每个文件名的优先级将高于在它之前的文件名。*init()* 允许被重复调用。

为 *files* 指定一个空列表将防止应用系统默认选项：将只保留来自内置列表的常用值。

在 3.2 版更改: 在之前版本中，Windows 注册表设置会被忽略。

mimetypes.**read_mime_types**(*filename*)
载入在文件 *filename* 中给定的类型映射，如果文件存在的话。返回的类型映射会是一个字典，其中的键值对为文件扩展名包括开头的点号 ('.') 与 'type/subtype' 形式的字符串。如果文件 *filename* 不存在或无法被读取，则返回 None。

mimetypes.**add_type**(*type*, *ext*, *strict=True*)
添加一个从 MIME 类型 *type* 到扩展名 *ext* 的映射。当扩展名已知时，新类型将替代旧类型。当类型已知时，扩展名将被添加到已知扩展名列表。

当 *strict* 为 True 时（默认值），映射将被添加到官方 MIME 类型，否则添加到非标准类型。

mimetypes.**inited**
指明全局数据结构是否已被初始化的旗标。这会由*init()* 设为 True。

mimetypes.**knownfiles**
通常安装的类型映射文件名列表。这些文件一般被命名为 mime.types 并会由不同的包安装在不同的位置。

mimetypes.**suffix_map**
将后缀映射到其他后缀的字典。它被用来允许识别已编码的文件，其编码格式和类型是由相同的扩展名来指明的。例如，.tgz 扩展名被映射到 .tar.gz 以允许编码格式和类型被分别识别。

mimetypes.**encodings_map**
映射扩展名到编码格式类型的字典。

mimetypes.**types_map**
映射文件扩展名到 MIME 类型的字典。

mimetypes.**common_types**
映射文件扩展名到非标准但常见的 MIME 类型的字典。

此模块一个使用示例:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

19.5.1 MimeTypes 对象

MimeTypes 类可以被用于那些需要多个 MIME 类型数据库的应用程序；它提供了与*mimetypes* 模块所提供的类似接口。

class mimetypes.**MimeTypes**(*filenames=()*, *strict=True*)
这个类表示 MIME 类型数据库。默认情况下，它提供了对与此模块其余部分一致的数据库的访问权限。这个初始数据库是此模块所提供数据库的一个副本，并可以通过使用*read()* 或*readfp()* 方法将附加的 mime.types 样式文载入到数据库中来进行扩展。如果不需要默认数据的话这个映射字典也可以在载入附加数据之前先被清空。

可选的 *filenames* 形参可被用来让附加文件被载入到默认数据库“之上”。

suffix_map
将后缀映射到其他后缀的字典。它被用来允许识别已编码的文件，其编码格式和类型是由相同的扩展名来指明的。例如，.tgz 扩展名被映射到 .tar.gz 以允许编码格式和类型被分别识别。这是在模块中定义的全局*suffix_map* 的一个副本。

encodings_map
映射文件扩展名到编码格式类型的字典。这是在模块中定义的全局*encodings_map* 的一个副本。

types_map
包含两个字典的元组，将文件扩展名映射到 MIME 类型：第一个字典针对非标准类型而第二个字典针对标准类型。它们会由*common_types* 和*types_map* 来初始化。

types_map_inv
包含两个字典的元组，将 MIME 类型映射到文件扩展名列表：第一个字典针对非标准类型而第二个字典针对标准类型。它们会由*common_types* 和*types_map* 来初始化。

guess_extension(*type*, *strict=True*)
类似于*guess_extension()* 函数，使用存储的表作为对象的一部分。

guess_type(*url*, *strict=True*)
类似于*guess_type()* 函数，使用存储的表作为对象的一部分。

guess_all_extensions(*type*, *strict=True*)
类似于*guess_all_extensions()* 函数，使用存储的表作为对象的一部分。

read(*filename*, *strict=True*)
从名称为 *filename* 的文件载入 MIME 信息。此方法使用*readfp()* 来解析文件。

如果 *strict* 为 True，信息将被添加到标准类型列表，否则添加到非标准类型列表。

readfp(*fp*, *strict=True*)
从打开的文件 *fp* 载入 MIME 类型信息。文件必须具有标准 `mime.types` 文件的格式。

如果 *strict* 为 `True`，信息将被添加到标准类型列表，否则添加到非标准类型列表。

read_windows_registry(*strict=True*)
Load MIME type information from the Windows registry. Availability: Windows.

如果 *strict* 为 `True`，信息将被添加到标准类型列表，否则添加到非标准类型列表。

3.2 新版功能.

19.6 base64 —Base16, Base32, Base64, Base85 数据编码

源代码： Lib/base64.py

此模块提供了将二进制数据编码为可打印的 ASCII 字符以及将这些编码解码回二进制数据的函数。它为 **RFC 3548** 指定的 Base16, Base32 和 Base64 编码以及已被广泛接受的 Ascii85 和 Base85 编码提供了编码和解码函数。

RFC 3548 编码的目的是使得二进制数据可以作为电子邮件的内容正确地发送，用作 URL 的一部分，或者作为 HTTP POST 请求的一部分。其中的编码算法和 **uuencode** 程序是不同的。

此模块提供了两个接口。新的接口提供了从*类字节对象* 到 ASCII 字节*bytes* 的编码，以及将 ASCII 的*类字节对象* 或字符串解码到*bytes* 的操作。此模块支持定义在 **RFC 3548** 中的所有 base-64 字母表（普通的、URL 安全的和文件系统安全的）。

旧的接口不提供从字符串的解码操作，但提供了操作*文件对象* 的编码和解码函数。旧接口只支持标准的 Base64 字母表，并且按照 **RFC 2045** 的规范每 76 个字符增加一个换行符。注意：如果你需要支持 **RFC 2045**，那么使用*email* 模块可能更加合适。

在 3.3 版更改: 新的接口提供的解码函数现在已经支持只包含 ASCII 的 Unicode 字符串。

在 3.4 版更改: 所有*类字节对象* 现在已经被所有编码和解码函数接受。添加了对 Ascii85/Base85 的支持。

新的接口提供：

base64.**b64encode**(*s*, *altchars=None*)
对*bytes-like object* *s* 进行 Base64 编码，并返回编码后的*bytes*。

可选项 *altchars* 必须是一个长 2 字节的*bytes-like object*，它指定了用于替换 + 和 / 的字符。这允许应用程序生成 URL 或文件系统安全的 Base64 字符串。默认值是 `None`，使用标准 Base64 字母表。

base64.**b64decode**(*s*, *altchars=None*, *validate=False*)
解码 Base64 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的*bytes*。

可选项 *altchars* 必须是一个长 2 字节的*bytes-like object*，它指定了用于替换 + 和 / 的字符。

如果 *s* 被不正确地填写，一个*binascii.Error* 错误将被抛出。

如果 *validate* 值为 `False` （默认情况），则在填充检查前，将丢弃既不在标准 base-64 字母表之中也不在备用字母表中的字符。如果 *validate* 为 `True`，这些非 base64 字符将导致*binascii.Error*。

base64.**standard_b64encode**(*s*)
编码*bytes-like object* *s*，使用标准 Base64 字母表并返回编码过的*bytes*。

base64.**standard_b64decode**(*s*)
解码*bytes-like object* 或 ASCII 字符串 *s*，使用标准 Base64 字母表并返回编码过的*bytes*。

base64.**urlsafe_b64encode**(*s*)
编码*bytes-like object* *s*，使用 URL 与文件系统安全的字母表，使用 - 以及 _ 代替标准 Base64 字母表中的 + 和 / 。返回编码过的*bytes*。结果中可能包含 =。

base64.**urlsafe_b64decode**(*s*)
解码*bytes-like object* 或 ASCII 字符串 *s*，使用 URL 与文件系统安全的字母表，使用 - 以及 _ 代替标准 Base64 字母表中的 + 和 / 。返回解码过的*bytes*

base64.**b32encode**(*s*)
用 Base32 编码*bytes-like object* *s* 并返回编码过的*bytes*

base64.**b32decode**(*s*, *casefold=False*, *map01=None*)
解码 Base32 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的*bytes*。

可选的 *casefold* 是一个指定小写字幕是否可接受为输入的标志。为了安全考虑，默认值为 False。

RFC 3548 允许将字母 0(zero) 映射为字母 O(oh)，并可以选择是否将字母 1(one) 映射为 I(eye) 或 L(el)。可选参数 *map01* 不是 None 时，它的值指定 1 的映射目标 (I 或 l)，当 *map01* 非 None 时，0 总是被映射为 O。为了安全考虑，默认值被设为 None，如果是这样，0 和 1 不允许被作为输入。

如果 *s* 被错误地填写或输入中存在字母表之外的字符，将抛出*binascii.Error*。

base64.**b16encode**(*s*)
用 Base16 编码*bytes-like object* *s* 并返回编码过的*bytes*

base64.**b16decode**(*s*, *casefold=False*)
解码 Base16 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的*bytes*。

可选的 *casefold* 是一个指定小写字幕是否可接受为输入的标志。为了安全考虑，默认值为 False。

如果 *s* 被错误地填写或输入中存在字母表之外的字符，将抛出*binascii.Error*。

base64.**a85encode**(*b*, ***, *foldspaces=False*, *wrapcol=0*, *pad=False*, *adobe=False*)
用 Ascii85 编码*bytes-like object* *s* 并返回编码过的*bytes*

foldspaces 是一个可选的标志，使用特殊的短序列‘y’代替‘btoa’提供的 4 个连续空格 (ASCII 0x20)。这个特性不被“标准”Ascii85 编码支持。

wrapcol 控制了输出是否包含换行符 (b'\n'). 如果该值非零, 则每一行只有该值所限制的字符长度.

pad 控制在编码之前输入是否填充为 4 的倍数。请注意，btoa 实现总是填充。

adobe 控制编码后的字节序列是否要加上 <~ 和 ~>，这是 Adobe 实现所使用的。

3.4 新版功能.

base64.**a85decode**(*b*, ***, *foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\v'*)
解码 Ascii85 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的*bytes*。

foldspaces 旗标指明是否应接受‘y’短序列作为 4 个连续空格 (ASCII 0x20) 的快捷方式。此特性不被“标准”Ascii85 编码格式所支持。

adobe 控制输入序列是否为 Adobe Ascii85 格式 (即附加 <~ 和 ~>)。

ignorechars 应当是一个*bytes-like object* 或 ASCII 字符串，其中包含要从输入中忽略的字符。这应当只包含空白字符，并且默认包含 ASCII 中所有的空白字符。

3.4 新版功能.

base64.**b85encode**(*b*, *pad=False*)
用 base85（如 git 风格的二进制 diff 数据所用格式）编码*bytes-like object* *b* 并返回编码后的*bytes*。

如果 *pad* 为真值，输入将以 b'\0' 填充以使其编码前长度为 4 字节的倍数。

3.4 新版功能.

base64.**b85decode**(*b*)
解码 base85 编码过的*bytes-like object* 或 ASCII 字符串 *b* 并返回解码过的*bytes*。如有必要，填充会被隐式地移除。

3.4 新版功能.

旧式接口:

base64.**decode**(*input*, *output*)
解码二进制 *input* 文件的内容并将结果二进制数据写入 *output* 文件。*input* 和 *output* 必须为文件对象. *input* 将被读取直至 `input.readline()` 返回空字节串对象。

base64.**decodebytes**(*s*)
解码*bytes-like object* *s*，该对象必须包含一行或多行 base64 编码的数据，并返回已解码的*bytes*。

3.1 新版功能.

base64.**decodestring**(*s*)
已弃用的*decodebytes()* 的别名。

3.1 版后已移除.

base64.**encode**(*input*, *output*)
编码二进制 *input* 文件的内容并将经 base64 编码的数据写入 *output* 文件。*input* 和 *output* 必须为文件对象。*input* 将被读取直到 `input.read()` 返回空字节串对象。*encode()* 会在每输出 76 个字节之后插入一个换行符 (`b'\n'`)，并会确保输出总是以换行符来结束，如 **RFC 2045** (MIME) 所规定的那样。

base64.**encodebytes**(*s*)
编码*bytes-like object* *s*，其中可以包含任意二进制数据，并返回包含经 base64 编码数据的*bytes*，每输出 76 个字节之后将带一个换行符 (`b'\n'`)，并会确保在末尾也有一个换行符，如 **RFC 2045** (MIME) 所规定的那样。

3.1 新版功能.

base64.**encodestring**(*s*)
一个被弃用的*encodebytes()* 的别名。

3.1 版后已移除.

此模块的一个使用示例:

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

参见:

模块*binascii* 支持模块，包含 ASCII 到二进制和二进制到 ASCII 转换。

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) 第一部分：规定并描述因特网消息体的格式的机制。
第 5.2 节，“Base64 内容转换编码格式”提供了 base64 编码格式的定义。

19.7 binhex —对 binhex4 文件进行编码和解码

源代码: Lib/binhex.py

此模块以 binhe4 格式对文件进行编码和解码，该格式允许 Macintosh 文件以 ASCII 格式表示。仅处理数据分支。

binhex 模块定义了以下功能：

binhex.**binhex**(*input*, *output*)
将带有文件名 *输入*的二进制文件转换为 binhex 文件 *输出*。输出参数可以是文件名或类文件对象（`write()` 和 `close()` 方法的任何对象）。

binhex.**hexbin**(*input*, *output*)
解码 binhex 文件输入。*输入*可以是支持 `read()` 和 `close()` 方法的文件名或类文件对象。生成的文件将写入名为 *output* 的文件，除非参数为 `None`，在这种情况下，从 binhex 文件中读取输出文件名。

还定义了以下异常：

exception binhex.**Error**
当无法使用 binhex 格式编码某些内容时（例如，文件名太长而无法放入文件名字段中），或者输入未正确编码的 binhex 数据时，会引发异常。

参见:

模块*binascii* 支持模块，包含 ASCII 到二进制和二进制到 ASCII 转换。

19.7.1 注释

还有一个替代的、功能更强大的编码器和解码器接口，详细信息请参见源代码。

如果您在非 Macintosh 平台上编码或解码文本文件，它们仍将使用旧的 Macintosh 换行符约定（回车符作为行尾）。

19.8 binascii —二进制和 ASCII 码互转

---

binascii 模块包含很多在二进制和二进制表示的各种 ASCII 码之间转换的方法。通常情况不会直接使用这些函数，而是使用像*uu*，*base64*，或*binhex* 这样的封装模块。为了执行效率高，*binascii* 模块含有许多用 C 写的低级函数，这些底层函数被一些高级模块所使用。

注解: `a2b_*` 函数接受只含有 ASCII 码的 Unicode 字符串。其他函数只接受*字节类对象*（例如*bytes*，*bytearray* 和其他支持缓冲区协议的对象）。

在 3.3 版更改: ASCII-only unicode strings are now accepted by the `a2b_*` functions.

---

binascii 模块定义了以下函数：

binascii.**a2b_uu**(*string*)
将单行 uu 编码数据转换成二进制数据并返回。uu 编码每行的数据通常包含 45 个（二进制）字节，最后一行除外。每行数据后面可能跟有空格。

binascii.**b2a_uu**(*data*)
Convert binary data to a line of ASCII characters, the return value is the converted line, including a newline char. The length of *data* should be at most 45.

binascii.**a2b_base64**(*string*)
将 base64 数据块转换成二进制并以二进制数据形式返回。一次可以传递多行数据。

binascii.**b2a_base64**(*data*, *, *newline=True*)
将二进制数据转换为一行用 base64 编码的 ASCII 字符串。返回值是转换后的行数据，如果 *newline* 为 true，则返回值包括换行符。该函数的输出符合：rfc：*3548*。

在 3.6 版更改: 增加 *newline* 形参。

binascii.**a2b_qp**(*data*, *header=False*)
将一个引号可打印的数据块转换成二进制数据并返回。一次可以转换多行。如果可选参数 *header* 存在且为 true，则数据中的下划线将被解码成空格。

binascii.**b2a_qp**(*data*, *quotetabs=False*, *istext=True*, *header=False*)
将二进制数据转换为一行或多行带引号可打印编码的 ASCII 字符串。返回值是转换后的行数据。如果可选参数 *quotetabs* 存在且为真值，则对所有制表符和空格进行编码。如果可选参数 *istext* 存在且为真值，则不对新行进行编码，但将对尾随空格进行编码。如果可选参数 *header* 存在且为 true，则空格将被编码为下划线 **RFC 1522**。如果可选参数 *header* 存在且为假值，则也会对换行符进行编码; 不进行换行转换编码可能会破坏二进制数据流。

binascii.**a2b_hqx**(*string*)
将 binhex4 格式的 ASCII 数据不进行 RLE 解压缩直接转换为二进制数据。该字符串应包含完整数量的二进制字节，或者（在 binhex4 数据最后部分）剩余位为零。

binascii.**rledecode_hqx**(*data*)
根据 binhex4 标准对数据执行 RLE 解压缩。该算法在一个字节的数据后使用 `0x90` 作为重复指示符，然后计数。计数 `0` 指定字节值 `0x90` 。该例程返回解压缩的数据，输入数据以孤立的重复指示符结束的情况下，将引发 *Incomplete* 异常。

在 3.2 版更改: 仅接受 bytestring 或 bytearray 对象作为输入。

binascii.**rlecode_hqx**(*data*)
在 *data* 上执行 binhex4 游程编码压缩并返回结果。

binascii.**b2a_hqx**(*data*)
执行 hexbin4 类型二进制到 ASCII 码的转换并返回结果字符串。输入数据应经过 RLE 编码，且数据长度可被 3 整除（除了最后一个片段）。

binascii.**crc_hqx**(*data*, *value*)
以 *value* 作为初始 CRC 计算 *data* 的 16 位 CRC 值，返回其结果。这里使用 CRC-CCITT 生成多项式 $x^{16} + x^{12} + x^5 + 1$ ，通常表示为 0x1021。该 CRC 被用于 binhex4 格式。

binascii.**crc32**(*data*[, *value*])
计算 CRC-32 ，从 *value* 的初始 CRC 开始计算 *data* 的 32 位校验和。默认初始 CRC 为零。该算法与 ZIP 文件校验和一致。由于该算法被设计用作校验和算法，因此不适合用作通用散列算法。使用方法如下：

```
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

在 3.0 版更改: 校验结果始终是无符号类型的。要在所有 Python 版本和平台上生成相同的数值，请使用 `crc32(data) & 0xffffffff` 。

binascii.**b2a_hex**(*data*)

binascii.**hexlify**(*data*)
返回二进制数据 *data* 的十六进制表示形式。*data* 的每个字节都被转换为相应的 2 位十六进制表示形式。因此返回的字节对象的长度是 *data* 的两倍。

binascii.**a2b_hex**(*hexstr*)
binascii.**unhexlify**(*hexstr*)
返回由十六进制字符串 *hexstr* 表示的二进制数据。此函数功能与*b2a_hex()* 相反。*hexstr* 必须包含偶数个十六进制数字（可以是大写或小写），否则会引发*Error* 异常。

exception binascii.**Error**
通常是因为编程错误引发的异常。

exception binascii.**Incomplete**
数据不完整引发的异常。通常不是编程错误导致的，可以通过读取更多的数据并再次尝试来处理该异常。

参见:

模块*base64* 支持在 16，32，64，85 进制中进行符合 RFC 协议的 base64 样式编码。

模块*binhex* 支持在 Macintosh 上使用的 binhex 格式。

模块*uu* 支持在 Unix 上使用的 UU 编码。

模块*quopri* 支持在 MIME 版本电子邮件中使用引号可打印编码。

19.9 quopri —编码与解码经过 MIME 转码的可打印数据

源代码: Lib/quopri.py

此模块会执行转换后可打印的传输编码与解码，具体定义见 **RFC 1521**: “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies”。转换后可打印的编码格式被设计用于只包含相对较少的不可打印字符的数据；如果存在大量这样的字符，通过*base64* 模块所提供的 base64 编码方案会更为紧凑，例如当发送图片文件时。

quopri.**decode**(*input*, *output*, *header=False*)
解码 *input* 文件的内容并将已解码二进制数据结果写入 *output* 文件。*input* 和 *output* 必须为二进制文件对象。如果提供了可选参数 *header* 且为真值，下划线将被解码为空格。此函数可用于解码 “Q” 编码的头数据，具体描述见 **RFC 1522**: “MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text”。

quopri.**encode**(*input*, *output*, *quotetabs*, *header=False*)
编码 *input* 文件的内容并将转换后可打印的数据结果写入 *output* 文件。*input* 和 *output* 必须为二进制文件对象. *quotetabs* 是一个非可选的旗标，它控制是否要编码内嵌的空格与制表符；当为真值时将编码此类内嵌空白符，当为假值时则保持原样不进行编码。请注意出现在行尾的空格与制表符总是会被编码，具体描述见 **RFC 1521**。*header* 旗标控制空格符是否要编码为下划线，具体描述见 **RFC 1522**。

quopri.**decodestring**(*s*, *header=False*)
类似*decode()*，区别在于它接受一个源*bytes* 并返回对应的已解码*bytes*。

quopri.**encodestring**(*s*, *quotetabs=False*, *header=False*)
类型*encode()*，区别在于它接受一个源*bytes* 并返回对应的已编码*bytes*。在默认情况下，它会发送 `False` 值给*encode()* 函数的 *quotetabs* 形参。

参见:

模块*base64* 编码与解码 MIME base64 数据

19.10 uu —对 uuencode 文件进行编码与解码

源代码: Lib/uu.py

此模块使用 uuencode 格式来编码和解码文件，以便任意二进制数据可通过仅限 ASCII 码的连接进行传输。在任何要求文件参数的地方，这些方法都接受文件类对象。为了保持向下兼容，也接受包含路径名称的字符串，并且将打开相应的文件进行读写；路径名称 `'-'` 被解读为标准输入或输出。但是，此接口已被弃用；在 Windows 中调用者最好是自行打开文件，并在需要时确保模式为 `'rb'` or `'wb'`。

此代码由 Lance Ellinghouse 贡献，并由 Jack Jansen 修改。

uu 模块定义了以下函数:

uu.**encode**(*in_file*, *out_file*, *name=None*, *mode=None*)
Uuencode file *in_file* into file *out_file*. The uuencoded file will have the header specifying *name* and *mode* as the defaults for the results of decoding the file. The default defaults are taken from *in_file*, or `'-'` and `0o666` respectively.

uu.**decode**(*in_file*, *out_file=None*, *mode=None*, *quiet=False*)
调用此函数会解码 uuencod 编码的 *in_file* 文件并将结果放入 *out_file* 文件。如果 *out_file* 是一个路径名称，*mode* 会在必须创建文件时用于设置权限位。*out_file* 和 *mode* 的默认值会从 uuencode 标头中提取。但是，如果标头中指定的文件已存在，则会引发*uu.Error*。

如果输入由不正确的 uuencode 编码器生成，*decode()* 可能会打印一条警告到标准错误，这样 Python 可以从该错误中恢复。将 *quiet* 设为真值可以屏蔽此警告。

exception uu.**Error**
Exception 的子类，此异常可由*uu.decode()* 在多种情况下引发，如上文所述，此外还包括格式错误的标头或被截断的输入文件等。

参见:

模块*binascii* 支持模块，包含 ASCII 到二进制和二进制到 ASCII 转换。

CHAPTER 20

结构化标记处理工具

Python 支持各种模块，以处理各种形式的结构化数据标记。这包括使用标准通用标记语言（SGML）和超文本标记语言（HTML）的模块，以及使用可扩展标记语言（XML）的几个接口。

20.1 html —超文本标记语言支持

源码： Lib/html/__init__.py

该模块定义了操作 HTML 的工具。

html.**escape**(*s*, *quote=True*)
将字符串 *s* 中的字符 “&“ 、< 和 > 转换为安全的 HTML 序列。如果需要在 HTML 中显示可能包含此类字符的文本，请使用此选项。如果可选的标志 *quote* 为真值，则字符 (") 和 (') 也被转换；这有助于包含在由引号分隔的 HTML 属性中，如 <a href="...">。

3.2 新版功能.

html.**unescape**(*s*)
将字符串 *s* 中的所有命名和数字字符引用 (例如 >, >, >) 转换为相应的 Unicode 字符。此函数使用 HTML 5 标准为有效和无效字符引用定义的规则，以及*HTML 5 命名字符引用列表*。

3.4 新版功能.

html 包中的子模块是：

- *html.parser* ——具有宽松解析模式的 HTML / XHTML 解析器
- *html.entities* –HTML 实体定义

20.2 html.parser —简单的 HTML 和 XHTML 解析器

源代码： Lib/html/parser.py

这个模块定义了一个*HTMLParser* 类，为 HTML（超文本标记语言）和 XHTML 文本文件解析提供基础。

class html.parser.**HTMLParser**(*, *convert_charrefs=True*)

创建一个能解析无效标记的解析器实例。

如果 *convert_charrefs* 为 True (默认值)，则所有字符引用 (script/style 元素中的除外) 都会自动转换为相应的 Unicode 字符。

一个*HTMLParser* 类的实例用来接受 HTML 数据，并在标记开始、标记结束、文本、注释和其他元素标记出现的时候调用对应的方法。要实现具体的行为，请使用*HTMLParser* 的子类并重载其方法。

这个解析器不检查结束标记是否与开始标记匹配，也不会因外层元素完毕而隐式关闭了的元素引发结束标记处理。

在 3.4 版更改: *convert_charrefs* 关键字参数被添加。

在 3.5 版更改: *convert_charrefs* 参数的默认值现在为 True。

20.2.1 HTML 解析器的示例程序

下面是简单的 HTML 解析器的一个基本示例，使用*HTMLParser* 类，当遇到开始标记、结束标记以及数据的时候将内容打印出来。

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

输出是:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

20.2.2 HTMLParser 方法

HTMLParser 实例有下列方法：

HTMLParser.**feed**(*data*)
填充一些文本到解析器中。如果包含完整的元素，则被处理；如果数据不完整，将被缓冲直到更多的数据被填充，或者*close()* 被调用。*data* 必须为*str* 类型。

HTMLParser.**close**()
如同后面跟着一个文件结束标记一样，强制处理所有缓冲数据。这个方法能被派生类重新定义，用于在输入的末尾定义附加处理，但是重定义的版本应当始终调用基类*HTMLParser* 的*close()* 方法。

HTMLParser.**reset**()
重置实例。丢失所有未处理的数据。在实例化阶段被隐式调用。

HTMLParser.**getpos**()
返回当前行号和偏移值。

HTMLParser.**get_starttag_text**()
返回最近打开的开始标记中的文本。结构化处理时通常应该不需要这个，但在处理“已部署”的 HTML 或是在以最小改变来重新生成输入时可能会有用处（例如可以保留属性间的空格等）。

下列方法将在遇到数据或者标记元素的时候被调用。他们需要在子类中重载。基类的实现中没有任何实际操作（除了*handle_startendtag()* ）：

HTMLParser.**handle_starttag**(*tag*, *attrs*)
这个方法在标签开始的时候被调用（例如：`<div id="main">` ）。

tag 参数是小写的标记名。*attrs* 参数是一个 `(name, value)` 形式的列表，包含了所有在标记的 <> 括号中找到的属性。*name* 转换为小写，*value* 的引号被去除，字符和实体引用都会被替换。

实例中，对于标签 `<A HREF="https://www.cwi.nl/">`，这个方法将以下列形式被调用 `handle_starttag('a', [('href', 'https://www.cwi.nl/')])` 。

html.entities 中的所有实体引用，会被替换为属性值。

HTMLParser.**handle_endtag**(*tag*)
此方法被用来处理元素的结束标记（例如：`</div>` ）。

tag 参数是小写的标签名。

HTMLParser.**handle_startendtag**(*tag*, *attrs*)
类似于*handle_starttag()*，只是在解析器遇到 XHTML 样式的空标记时被调用（`<img ... />`）。这个方法能被需要这种特殊词法信息的子类重载；默认实现仅简单调用*handle_starttag()* 和*handle_endtag()* 。

HTMLParser.**handle_data**(*data*)
这个方法被用来处理任意数据（例如：文本节点和 `<script>...</script>` 以及 `<style>...</style>` 中的内容）。

HTMLParser.**handle_entityref**(*name*)
这个方法被用于处理 `&name;` 形式的命名字符引用（例如 `>`），其中 *name* 是通用的实体引用（例如：`'gt'`）。如果 *convert_charrefs* 为 `True`，该方法永远不会被调用。

HTMLParser.**handle_charref**(*name*)
这个方法被用来处理 `&#NNN;` 和 `&#xNNN;` 形式的十进制和十六进制字符引用。例如，`>` 等效的十进制形式为 `&#62;` ，而十六进制形式为 `&#x3E;` ；在这种情况下，方法将收到 `'62'` 或 `'x3E'` 。如果 *convert_charrefs* 为 `True` ，则该方法永远不会被调用。

HTMLParser.**handle_comment**(*data*)
这个方法在遇到注释的时候被调用（例如：`<!--comment-->` ）。

例如，`<!-- comment -->` 这个注释会用 `' comment '` 作为参数调用此方法。

Internet Explorer 条件注释（condcoms）的内容也被发送到这个方法，因此，对于 `<!--[if IE 9]>IE9-specific content<![endif]-->`，这个方法将接收到 `'[if IE 9]>IE9-specific content<![endif]'`。

HTMLParser.**handle_decl**(*decl*)
这个方法用来处理 HTML doctype 申明（例如 `<!DOCTYPE html>`）。

decl 形参为 `<!...>` 标记中的所有内容（例如：`'DOCTYPE html'`）。

HTMLParser.**handle_pi**(*data*)
此方法在遇到处理指令的时候被调用。*data* 形参将包含整个处理指令。例如，对于处理指令 `<?proc color='red'>`，这个方法将以 `handle_pi("proc color='red'")` 形式被调用。它旨在被派生类重载；基类实现中无任何实际操作。

注解： *HTMLParser* 类使用 SGML 语法规则处理指令。使用 `'?'` 结尾的 XHTML 处理指令将导致 `'?'` 包含在 *data* 中。

HTMLParser.**unknown_decl**(*data*)
当解析器读到无法识别的声明时，此方法被调用。

data 形参为 `<![...]>` 标记中的所有内容。某些时候对派生类的重载很有用。基类实现中无任何实际操作。

20.2.3 例子

下面的类实现了一个解析器，用于更多示例的演示:

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)

    def handle_data(self, data):
        print("Data     :", data)

    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent  :", c)
```

(下页继续)

(续上页)

```
    def handle_decl(self, data):
        print("Decl     :", data)

parser = MyHTMLParser()
```

解析一个文档类型声明:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
↪html4/strict.dtd"
```

解析一个具有一些属性和标题的元素:

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

`script` 和 `style` 元素中的内容原样返回，无需进一步解析:

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

解析注释:

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

解析命名或数字形式的字符引用，并把他们转换到正确的字符（注意：这 3 种转义都是 `'>'`）：

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

填充不完整的块给*feed()* 执行，*handle_data()* 可能会多次调用（除非 *convert_charrefs* 被设置为 `True`）：

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     : text
End tag  : span
```

解析无效的 HTML (例如：未引用的属性）也能正常运行:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

20.3 html.entities —HTML 一般实体的定义

源码： Lib/html/entities.py

该模块定义了四个词典，*html5*、*name2codepoint*、*codepoint2name*、以及*entitydefs*。

html.entities.**html5**
: 将 HTML5 命名字符引用[1] 映射到等效的 Unicode 字符的字典，例如 `html5['gt;'] == '>'`。请注意，尾随的分号包含在名称中（例如 `'gt;'`），但是即使没有分号，一些名称也会被标准接受，在这种情况下，名称出现时带有和不带有 `';'`。另见*html.unescape()*。

  3.3 新版功能.

html.entities.**entitydefs**
: 将 XHTML 1.0 实体定义映射到 ISO Latin-1 中的替换文本的字典。

html.entities.**name2codepoint**
: 将 HTML 实体名称映射到 Unicode 代码点的字典。

html.entities.**codepoint2name**
: 将 Unicode 代码点映射到 HTML 实体名称的字典。

备注

20.4 XML 处理模块

源码： Lib/xml/

用于处理 XML 的 Python 接口分组在 `xml` 包中。

[1] 参见 https://www.w3.org/TR/html5/syntax.html#named-character-references

警告： XML 模块对于错误或恶意构造的数据是不安全的。如果需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞* 和*defusedxml 和 defusedexpat 软件包* 部分。

值得注意的是`xml` 包中的模块要求至少有一个 SAX 兼容的 XML 解析器可用。在 Pythonm 中包含 Expat 解析器，因此`xml.parsers.expat` 模块将始终可用。

`xml.dom` 和`xml.sax` 包的文档是 DOM 和 SAX 接口的 Python 绑定的定义。

XML 处理子模块包括:

- `xml.etree.ElementTree`：ElementTree API，一个简单而轻量级的 XML 处理器
- `xml.dom`：DOM API 定义
- `xml.dom.minidom`：最小的 DOM 实现
- `xml.dom.pulldom`：支持构建部分 DOM 树
- `xml.sax`：SAX2 基类和便利函数
- `xml.parsers.expat`：Expat 解析器绑定

20.4.1 XML 漏洞

XML 处理模块对于恶意构造的数据是不安全的。攻击者可能滥用 XML 功能来执行拒绝服务攻击、访问本地文件、生成与其它计算机的网络连接或绕过防火墙。

下表概述了已知的攻击以及各种模块是否容易受到攻击。

种类	sax	etree	minidom	pulldom	xmlrpc
billion laughs	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
quadratic blowup	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
external entity expansion	Safe (5)	Safe (2)	Safe (3)	Safe (5)	安全 (4)
DTD retrieval	Safe (5)	安全	安全	Safe (5)	安全
decompression bomb	安全	安全	安全	安全	**易受攻击**

1. Expat 2.4.1 and newer is not vulnerable to the “billion laughs” and “quadratic blowup” vulnerabilities. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.
2. `xml.etree.ElementTree` 不会扩展外部实体并在实体发生时引发 `ParserError`。
3. `xml.dom.minidom` 不会扩展外部实体，只是简单地返回未扩展的实体。
4. `xmlrpclib` 不扩展外部实体并省略它们。
5. Since Python 3.6.7, external general entities are no longer processed by default.

billion laughs / exponential entity expansion （狂笑/递归实体扩展） Billion Laughs 攻击–也称为递归实体扩展–使用多级嵌套实体。每个实体多次引用另一个实体，最终实体定义包含一个小字符串。指数级扩展导致几千 GB 的文本，并消耗大量内存和 CPU 时间。

quadratic blowup entity expansion（二次爆炸实体扩展） 二次爆炸攻击类似于 Billion Laughs 攻击，它也滥用实体扩展。它不是嵌套实体，而是一遍又一遍地重复一个具有几千个字符的大型实体。攻击不如递归情况有效，但它避免触发禁止深度嵌套实体的解析器对策。

external entity expansion 实体声明可以包含的不仅仅是替换文本。它们还可以指向外部资源或本地文件。XML 解析器访问资源并将内容嵌入到 XML 文档中。

DTD retrieval Python 的一些 XML 库*xml.dom.pulldom* 从远程或本地位置检索文档类型定义。该功能与外部实体扩展问题具有相似的含义。

decompression bomb Decompression bombs（解压炸弹，又名 ZIP bomb）适用于所有可以解析压缩 XML 流（例如 gzip 压缩的 HTTP 流或 LZMA 压缩的文件）的 XML 库。对于攻击者来说，它可以将传输的数据量减少三个量级或更多。

PyPI 上 defusedxml 的文档包含有关所有已知攻击向量的更多信息以及示例和参考。

20.4.2 defusedxml 和 defusedexpat 软件包

defusedxml 是一个纯 Python 软件包，它修改了所有标准库 XML 解析器的子类，可以防止任何潜在的恶意操作。对于解析不受信任的 XML 数据的任何服务器代码，建议使用此程序包。该软件包还提供了有关更多 XML 漏洞（如 XPath 注入）的示例漏洞和扩展文档。

defusedexpat 提供了一个修改过的 libexpat 和一个打过补丁的 `pyexpat` 模块，它有针对实体扩展 DoS 攻击的对策。`defusedexpat` 模块仍然允许合理且可配置的实体扩展量。这些修改可能包含在 Python 的某些未来版本中，但不会包含在 Python 的任何修复版本中，因为它们会破坏向后兼容性。

20.5 xml.etree.ElementTree —ElementTree XML API

源代码： Lib/xml/etree/ElementTree.py

---

xml.etree.ElementTree 模块实现了一个简单高效的 API，用于解析和创建 XML 数据。

在 3.3 版更改: 只要有可能，这个模块将使用快速实现。`xml.etree.cElementTree` 模块已弃用。

> **警告：** *xml.etree.ElementTree* 模块对于恶意构建的数据是不安全的。如果需要解析不可信或未经身份验证的数据，请参见*XML 漏洞* 。

20.5.1 教程

这是一个使用*xml.etree.ElementTree* （简称 `ET` ）的简短教程。目标是演示模块的一些构建块和基本概念。

XML 树和元素

XML 是一种固有的分层数据格式，最自然的表示方法是使用树。为此，`ET` 有两个类*ElementTree* 将整个 XML 文档表示为一个树，*Element* 表示该树中的单个节点。与整个文档的交互（读写文件）通常在*ElementTree* 级别完成。与单个 XML 元素及其子元素的交互是在*Element* 级别完成的。

解析 XML

我们将使用以下 XML 文档作为本节的示例数据：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

我们可以通过从文件中读取来导入此数据：

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

或直接从字符串中：

```
root = ET.fromstring(country_data_as_string)
```

fromstring() 将 XML 从字符串直接解析为*Element*，该元素是已解析树的根元素。其他解析函数可能会创建一个*ElementTree*。确切的信息请检查文档。

作为一个*Element*，root 有一个标记和一个属性字典：

```
>>> root.tag
'data'
>>> root.attrib
{}
```

它还有我们可以迭代的子节点：

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

子级是可以嵌套的，我们可以通过索引访问特定的子级节点：

```
>>> root[0][1].text
'2008'
```

注解： 并非 XML 输入的所有元素都将作为解析树的元素结束。目前，此模块跳过输入中的任何 XML 注释、处理指令和文档类型声明。然而，使用这个模块的 API 而不是从 XML 文本解析构建的树可以包含注释和处理指令，生成 XML 输出时同样包含这些注释和处理指令。可以通过将自定义*TreeBuilder* 实例传递给*XMLParser* 构造函数来访问文档类型声明。

Pull API 进行非阻塞解析

此模块所提供了大多数解析函数都要求在返回任何结果之前一次性读取整个文档。可以使用*XMLParser* 并以渐进方式添加数据，但这是在回调目标上调用方法的推送式 API。有时用户真正想要的是能够以渐进方式解析 XML 而无需阻塞操作，同时享受完整的已构造*Element* 对象。

针对此需求的最强大工具是*XMLPullParser*。它不要求通过阻塞式读取来获得 XML 数据，而是通过执行*XMLPullParser.feed()* 调用来渐进式地添加数据。要获得已解析的 XML 元素，应调用*XMLPullParser.read_events()*。下面是一个示例：

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
```

常见的用例是针对以非阻塞方式进行的应用程序，其中 XML 是从套接字接收或从某些存储设备渐进式读取的。在这些用例中，阻塞式读取是不可接受的。

因为其非常灵活，*XMLPullParser* 在更简单的用例中使用起来可能并不方便。如果你不介意你的应用程序在读取 XML 数据时造成阻塞但仍希望具有增量解析能力，可以考虑*iterparse()*。它在你读取大型 XML 文档并且不希望将它完全放去内存时会很适用。

寻找有趣的元素

Element 有一些很有效的方法，可帮助递归遍历其下的所有子树（包括子级，子级的子级，等等）。例如*Element.iter()*：

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall() 仅查找当前元素的直接子元素中带有指定标签的元素。*Element.find()* 找带有特定标签的 第一个子级，然后可以用*Element.text* 访问元素的文本内容。*Element.text* 访问元素的属性:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

通过使用*XPath*，可以更精确地指定要查找的元素。

修改 XML 文件

ElementTree 提供了一种构建 XML 文档并将其写入文件的简单方法。*ElementTree.write()* 方法可达到此目的。

创建后可以直接操作*Element* 对象。例如：使用*Element.text* 修改文本字段，使用*Element.set()* 方法添加和修改属性，以及使用*Element.append()* 添加新的子元素。

假设我们要在每个国家/地区的中添加一个排名，并在 rank 元素中添加一个 `updated` 属性：

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

生成的 XML 现在看起来像这样：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

可以使用*Element.remove()* 删除元素。假设我们要删除排名高于 50 的所有国家/地区：

```
>>> for country in root.findall('country'):
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

生成的 XML 现在看起来像这样:

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

构建 XML 文档

SubElement() 函数还提供了一种便捷方法来为给定元素创建新的子元素:

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

使用命名空间解析 XML

如果 XML 输入带有 命名空间，则具有前缀的 `prefix:sometag` 形式的标记和属性将被扩展为 `{uri}sometag`，其中 *prefix* 会被完整 *URI* 所替换。并且，如果存在 默认命名空间，则完整 URI 会被添加到所有未加前缀的标记之前。

下面的 XML 示例包含两个命名空间，一个具有前缀 “fictional” 而另一个则作为默认命名空间:

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
```

(下页继续)

(续上页)

```
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

搜索和探查这个 XML 示例的一种方式是手动为*find()* 或*findall()* 的 xpath 中的每个标记或属性添加 URI:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

一种更好的方式是搜索带命名空间的 XML 示例创建一个字典来存放你自己的前缀并在搜索函数中使用它们:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

这两种方式都会输出:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

其他资源

请访问 http://effbot.org/zone/element-index.htm 获取教程和其他文档的链接。

20.5.2 XPath 支持

此模块提供了对 XPath 表达式 的有限支持用于在树中定位元素。其目标是支持一个简化语法的较小子集；完整的 XPath 引擎超出了此模块的适用范围。

示例

下面是一个演示此模块的部分 XPath 功能的例子。我们将使用来自*解析 XML* 小节的 `countrydata` XML 文档:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

支持的 XPath 语法

语法	含义
`tag`	Selects all child elements with the given tag. For example, `spam` selects all child elements named `spam`, and `spam/egg` selects all grandchildren named `egg` in all children named `spam`.
`*`	Selects all child elements. For example, `*/egg` selects all grandchildren named `egg`.
`.`	选择当前节点。这在路径的开头非常有用，用于指示它是相对路径。
`//`	选择所有子元素在当前元素的所有下级中选择所有下级元素。例如，`.//egg` 是在整个树中选择所有 `egg` 元素。
`..`	选择父元素。如果路径试图前往起始元素的上级（元素的 `find` 被调用）则返回 `None`。
`[@attrib]`	选择具有给定属性的所有元素。
`[@attrib='value']`	选择给定属性具有给定值的所有元素。该值不能包含引号。
`[tag]`	选择所有包含 `tag` 子元素的元素。只支持直系子元素。
`[tag='text']`	选择所有包含名为 `tag` 的子元素的元素，这些子元素（包括后代）的完整文本内容等于给定的 `text` 。
`[position]`	选择位于给定位置的所有元素。位置可以是一个整数 (1 表示首位)，表达式 `last()` (表示末位)，或者相对于末位的位置 (例如 `last()-1`)。

谓词（方括号内的表达式）之前必须带有标签名称，星号或其他谓词。`position` 谓词前必须有标签名称。

20.5.3 参考

函数

xml.etree.ElementTree.**Comment**(*text=None*)
注释元素工厂函数。这个工厂函数可创建一个特殊元素，它将被标准序列化器当作 XML 注释来进行序列化。注释字串可以是字节串或是 Unicode 字符串。*text* 是包含注释字串的字符串。返回一个表示注释的元素实例。

请注意*XMLParser* 会跳过输入中的注释而不会为其创建注释对象。*ElementTree* 将只在当使用某个*Element* 方法向树插入了注释节点时才会包含注释节点。

xml.etree.ElementTree.**dump**(*elem*)
将一个元素树或元素结构体写入到 sys.stdout。此函数应当只被用于调试。

实际输出格式是依赖于具体实现的。在这个版本中，它将以普通 XML 文件的格式写入。

elem 是一个元素树或单独元素。

xml.etree.ElementTree.**fromstring**(*text*)
Parses an XML section from a string constant. Same as *XML()*. *text* is a string containing XML data. Returns an *Element* instance.

xml.etree.ElementTree.**fromstringlist**(*sequence*, *parser=None*)
根据一个字符串片段序列解析 XML 文档。*sequence* 是包含 XML 数据片段的列表或其他序列对象。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个*Element* 实例。

3.2 新版功能.

xml.etree.ElementTree.**iselement**(*element*)
Checks if an object appears to be a valid element object. *element* is an element instance. Returns a true value if this is an element object.

xml.etree.ElementTree.**iterparse**(*source*, *events=None*, *parser=None*)
Parses an XML section into an element tree incrementally, and reports what's going on to the user. *source* is a filename or *file object* containing XML data. *events* is a sequence of events to report back. The supported events are the strings `"start"`, `"end"`, `"start-ns"` and `"end-ns"` (the "ns"events are used to get detailed namespace information). If *events* is omitted, only `"end"` events are reported. *parser* is an optional parser instance. If not given, the standard *XMLParser* parser is used. *parser* must be a subclass of *XMLParser* and can only use the default *TreeBuilder* as a target. Returns an *iterator* providing `(event, elem)` pairs.

请注意虽然*iterparse()* 是以增量方式构建树，但它会对 *source*（或其所指定的文件）发出阻塞式读取。因此，它不适用于不可执行阻塞式读取的应用。对于完全非阻塞式的解析，请参看*XMLPullParser*。

注解： *iterparse()* 只会确保当发出“start”事件时看到了开始标记的“>”字符，因而在这个点上属性已被定义，但文本容和末尾属性还未被定义。这同样适用于元素的下级；它们可能存在也可能不存在。

如果你需要已完全填充的元素，请改为查找“end”事件。

3.4 版后已移除: *parser* 参数。

xml.etree.ElementTree.**parse**(*source*, *parser=None*)
将一个 XML 的节解析为元素树。*source* 是包含 XML 数据的文件名或文件对象。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个*ElementTree* 实例。

xml.etree.ElementTree.**ProcessingInstruction**(*target*, *text=None*)
PI 元素工厂函数。这个工厂函数可创建一个特殊元素，它将被当作 XML 处理指令来进行序列化。*target* 是包含 PI 目标的字符串。*text* 如果给出则是包含 PI 内容的字符串。返回一个表示处理指令的元素实例。

请注意*XMLParser* 会跳过输入中的处理指令而不会为其创建注释对象。*ElementTree* 将只在当使用某个*Element* 方法向树插入了处理指令节点时才会包含处理指令节点。

xml.etree.ElementTree.**register_namespace**(*prefix*, *uri*)
注册一个命名空间前缀。这个注册表是全局的，并且任何对应给定前缀或命名空间 URI 的现有映射都会被移除。*prefix* 是命名空间前缀。*uri* 是命名空间 URI。如果可能的话，这个命名空间中的标记和属性将附带给定的前缀来进行序列化。

3.2 新版功能.

xml.etree.ElementTree.**SubElement**(*parent*, *tag*, *attrib={}*, ***extra*)
子元素工厂函数。这个函数会创建一个元素实例，并将其添加到现有的元素。

元素名、属性名和属性值可以是字节串或 Unicode 字符串。*parent* 是父元素。*tag* 是子元素名。*attrib* 是一个可选的字典，其中包含元素属性。*extra* 包含额外的属性，以关键字参数形式给出。返回一个元素实例。

xml.etree.ElementTree.**tostring**(*element*, *encoding="us-ascii"*, *method="xml"*, ***, *short_empty_elements=True*)
Generates a string representation of an XML element, including all subelements. *element* is an *Element* instance. *encoding*[1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *short_empty_elements* has the same meaning as in *ElementTree.write()*. Returns an (optionally) encoded string containing the XML data.

3.4 新版功能: *short_empty_elements* 形参。

xml.etree.ElementTree.**tostringlist**(*element*, *encoding="us-ascii"*, *method="xml"*, ***, *short_empty_elements=True*)
Generates a string representation of an XML element, including all subelements. *element* is an *Element* instance. *encoding*[1] is the output encoding (default is US-ASCII). Use `encoding="unicode"` to generate a Unicode string (otherwise, a bytestring is generated). *method* is either `"xml"`, `"html"` or `"text"` (default is `"xml"`). *short_empty_elements* has the same meaning as in *ElementTree.write()*. Returns a list of (optionally) encoded strings containing the XML data. It does not guarantee any specific sequence, except that `b"".join(tostringlist(element)) == tostring(element)`.

3.2 新版功能.

3.4 新版功能: *short_empty_elements* 形参。

xml.etree.ElementTree.**XML**(*text*, *parser=None*)
根据一个字符串常量解析 XML 的节。此函数可被用于在 Python 代码中嵌入 “XML 字面值”。*text* 是包含 XML 数据的字符串。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个*Element* 实例。

xml.etree.ElementTree.**XMLID**(*text*, *parser=None*)
根据一个字符串常量解析 XML 的节，并且还将返回一个将元素的 id:s 映射到元素的字典。*text* 是包含 XML 数据的字符串。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个包含*Element* 实例和字典的元组。

[1] 包括在 XML 输出中的编码格式字符串应当符合适当的标准。例如 “UTF-8” 是有效的，但 “UTF8” 是无效的。请参阅 https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 和 https://www.iana.org/assignments/character-sets/character-sets.xhtml。

元素对象

class xml.etree.ElementTree.**Element**(*tag*, *attrib={}*, ***extra*)
元素类。这个类定义了 Element 接口，并提供了这个接口的引用实现。

元素名、属性名和属性值可以是字节串或 Unicode 字符串。*tag* 是元素名。*attrib* 是一个可选的字典，其中包含元素属性。*extra* 包含额外的属性，以关键字参数形式给出。

tag
一个标识此元素意味着何种数据的字符串 (换句话说，元素类型)。

text
tail
这些属性可被用于存放与元素相关联的额外数据。它们的值通常为字符串但也可以是任何应用专属的对象。如果元素是基于 XML 文件创建的，*text* 属性会存放元素的开始标记及其第一个子元素或结束标记之间的文本，或者为 None，而 *tail* 属性会存放元素的结束标记及下一个标记之间的文本，或者为 None。对于 XML 数据

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

a 元素的 *text* 和 *tail* 属性均为 None，*b* 元素的 *text* 为 "1" 而 *tail* 为 "4"，*c* 元素的 *text* 为 "2" 而 *tail* 为 None，*d* 元素的 *text* 为 None 而 *tail* 为 "3"。

要获取一个元素的内部文本，请参阅*itertext()*，例如 "".join(element.itertext())。

应用程序可以将任意对象存入这些属性。

attrib
一个包含元素属性的字典。请注意虽然 *attrib* 值总是一个真正可变的 Python 字典，但 ElementTree 实现可以选择其他内部表示形式，并只在有需要时才创建字典。为了发挥这种实现的优势，请在任何可能情况下使用下列字典方法。

以下字典类方法可作用于元素属性。

clear()
重设一个元素。此方法会移除所有子元素，清空所有属性，并将 text 和 tail 属性设为 None。

get(*key*, *default=None*)
获取名为 *key* 的元素属性。

返回属性的值，或者如果属性未找到则返回 *default*。

items()
将元素属性以 (name, value) 对序列的形式返回。所返回属性的顺序任意。

keys()
将元素属性名称以列表的形式返回。所返回名称的顺序任意。

set(*key*, *value*)
将元素的 *key* 属性设为 *value*。

以下方法作用于元素的下级（子元素）。

append(*subelement*)
将元素 *subelement* 添加到此元素的子元素内部列表。如果 *subelement* 不是一个*Element* 则会引发*TypeError*。

extend(*subelements*)
使用具有零个或多个元素的序列对象添加 *subelements*。如果某个子元素不是*Element* 则会引发*TypeError*。

3.2 新版功能.

find(*match*, *namespaces=None*)
Finds the first subelement matching *match*. *match* may be a tag name or a *path*. Returns an element instance or `None`. *namespaces* is an optional mapping from namespace prefix to full name.

findall(*match*, *namespaces=None*)
Finds all matching subelements, by tag name or *path*. Returns a list containing all matching elements in document order. *namespaces* is an optional mapping from namespace prefix to full name.

findtext(*match*, *default=None*, *namespaces=None*)
Finds text for the first subelement matching *match*. *match* may be a tag name or a *path*. Returns the text content of the first matching element, or *default* if no element was found. Note that if the matching element has no text content an empty string is returned. *namespaces* is an optional mapping from namespace prefix to full name.

getchildren()
3.2 版后已移除: Use `list(elem)` or iteration.

getiterator(*tag=None*)
3.2 版后已移除: Use method *Element.iter()* instead.

insert(*index*, *subelement*)
将 *subelement* 插入到此元素的给定位置中。如果 *subelement* 不是一个*Element* 则会引发*TypeError*。

iter(*tag=None*)
创建一个以当前元素为根元素的树的*iterator*。该迭代器将以文档（深度优先）顺序迭代此元素及其所有下级元素。如果 *tag* 不为 `None` 或 `'*'`，则迭代器只返回标记为 *tag* 的元素。如果树结构在迭代期间被修改，则结果是未定义的。

3.2 新版功能.

iterfind(*match*, *namespaces=None*)
根据标记名称或者*路径* 查找所有匹配的子元素。返回一个按文档顺序产生所有匹配元素的可迭代对象。*namespaces* 是可选的从命名空间前缀到完整名称的映射。

3.2 新版功能.

itertext()
创建一个文本迭代器。该迭代器将按文档顺序遍历此元素及其所有子元素，并返回所有内部文本。

3.2 新版功能.

makeelement(*tag*, *attrib*)
创建一个与此元素类型相同的新元素对象。请不要调用此方法，而应改用*SubElement()* 工厂函数。

remove(*subelement*)
从元素中移除 *subelement*。与 find* 方法不同的是此方法会基于实例的标识来比较元素，而不是基于标记的值或内容。

Element 对象还支持下列序列类型方法以配合子元素使用: `__delitem__()`, `__getitem__()`, `__setitem__()`, `__len__()`。

注意: 不带子元素的元素将被检测为 `False`。此行为将在未来的版本中发生变化。请改用 `len(elem)` 或 `elem is None` 进行检测。

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")
```

(下页继续)

(续上页)

```
if element is None:
    print("element not found")
```

ElementTree 对象

class xml.etree.ElementTree.**ElementTree**(*element=None*, *file=None*)

ElementTree 包装器类。这个类表示一个完整的元素层级结构，并添加了一些对于标准 XML 序列化的额外支持。

element 是根元素。如果给出 XML *file* 则将使用其内容来初始化树结构。

_setroot(*element*)

替换该树结构的根元素。这将丢弃该树结构的当前内容，并将其替换为给定的元素。请小心使用。*element* 是一个元素实例。

find(*match*, *namespaces=None*)

与*Element.find()* 类似，从树的根节点开始。

findall(*match*, *namespaces=None*)

与*Element.findall()* 类似，从树的根节点开始。

findtext(*match*, *default=None*, *namespaces=None*)

与*Element.findtext()* 类似，从树的根节点开始。

getiterator(*tag=None*)

3.2 版后已移除: Use method *ElementTree.iter()* instead.

getroot()

返回这个树的根元素。

iter(*tag=None*)

创建并返回根元素的树结构迭代器。该迭代器会以节顺序遍历这个树的所有元素。*tag* 是要查找的标记（默认返回所有元素）。

iterfind(*match*, *namespaces=None*)

与*Element.iterfind()* 类似，从树的根节点开始。

3.2 新版功能.

parse(*source*, *parser=None*)

将一个外部 XML 节载入到此元素树。*source* 是一个文件名或*file object*。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回该节的根元素。

write(*file*, *encoding="us-ascii"*, *xml_declaration=None*, *default_namespace=None*, *method="xml"*, ***, *short_empty_elements=True*)

将元素树以 XML 格式写入到文件。*file* 为文件名，或是以写入模式打开的*file object*。*encoding*[1] 为输出编码格式 (默认为 US-ASCII)。*xml_declaration* 控制是否要将 XML 声明添加到文件中。使用 `False` 表示从不添加，`True` 表示总是添加，`None` 表示仅在非 US-ASCII 或 UTF-8 或 Unicode 时添加 (默认为 `None`)。*default_namespace* 设置默认 XML 命名空间 (用于 “xmlns”)。*method* 为 `"xml"`, `"html"` 或 `"text"` (默认为 `"xml"`)。仅限关键字形参 *short_empty_elements* 控制不包含内容的元素的格式。如为 `True` (默认值)，它们会被输出为单个自结束标记，否则它们会被输出为一对开始/结束标记。

输出是一个字符串 (*str*) 或字节串 (*bytes*)。由 *encoding* 参数来控制。如果 *encoding* 为 `"unicode"`，则输出是一个字符串；否则为字节串；请注意这可能与 *file* 的类型相冲突，如果它是一个打开的*file object* 的话；请确保你不会试图写入字符串到二进制流或者反向操作。

3.4 新版功能: *short_empty_elements* 形参。

这是将要被操作的 XML 文件:

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

修改第一段中的每个链接的“target”属性的示例:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")      # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))    # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:              # Iterates through all found links
...     i.attrib["target"] = "blank"
>>> tree.write("output.xhtml")
```

QName 对象

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

QName 包装器。这可被用来包装 QName 属性值，以便在输出中获得适当的命名空间处理。*text_or_uri* 是一个包含 QName 值的字符串，其形式为 {uri}local，或者如果给出了 tag 参数，则为 QName 的 URI 部分。如果给出了 *tag*，则第一个参数会被解读为 URI，而这个参数会被解读为本地名称。*QName* 实例是不透明的。

TreeBuilder 对象

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*)

Generic element structure builder. This builder converts a sequence of start, data, and end method calls to a well-formed element structure. You can use this class to build an element structure using a custom XML parser, or a parser for some other XML-like format. *element_factory*, when given, must be a callable accepting two positional arguments: a tag and a dict of attributes. It is expected to return a new element instance.

close()

刷新构建器缓存，并返回最高层级的文档元素。返回一个*Element* 实例。

data(*data*)

将文本添加到当前元素。*data* 为要添加的文本。这应当是一个字节串或 Unicode 字符串。

end(*tag*)

结束当前元素。*tag* 是元素名称。返回已结束的元素。

start(*tag*, *attrs*)

打开一个新元素。*tag* 是元素名称。*attrs* 是包含元素属性的字典。返回打开的元素。

In addition, a custom *TreeBuilder* object can provide the following method:

doctype(*name*, *pubid*, *system*)
处理一条 doctype 声明。*name* 为 doctype 名称。*pubid* 为公有标识。*system* 为系统标识。此方法不存在于默认的`TreeBuilder`类中。

3.2 新版功能.

XMLParser 对象

class xml.etree.ElementTree.**XMLParser**(*html=0*, *target=None*, *encoding=None*)
This class is the low-level building block of the module. It uses `xml.parsers.expat` for efficient, event-based parsing of XML. It can be fed XML data incrementally with the `feed()` method, and parsing events are translated to a push API - by invoking callbacks on the *target* object. If *target* is omitted, the standard `TreeBuilder` is used. The *html* argument was historically used for backwards compatibility and is now deprecated. If *encoding*[1] is given, the value overrides the encoding specified in the XML file.

3.4 版后已移除: The *html* argument. The remaining arguments should be passed via keyword to prepare for the removal of the *html* argument.

close()
结束向解析器提供数据。返回调用在构造期间传入的 *target* 的 `close()` 方法的结果；在默认情况下，这是最高层级的文档元素。

doctype(*name*, *pubid*, *system*)
3.2 版后已移除: Define the `TreeBuilder.doctype()` method on a custom TreeBuilder target.

feed(*data*)
将数据送入解析器。*data* 是编码后的数据。

`XMLParser.feed()` calls *target*'s `start(tag, attrs_dict)` method for each opening tag, its `end(tag)` method for each closing tag, and data is processed by method `data(data)`. `XMLParser.close()` calls *target*'s method `close()`. `XMLParser` can be used not only for building a tree structure. This is an example of counting the maximum depth of an XML file:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with data.
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
```

(下页继续)

(续上页)

```
...      </c>
...    </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

XMLPullParser 对象

class xml.etree.ElementTree.**XMLPullParser**(*events=None*)

A pull parser suitable for non-blocking applications. Its input-side API is similar to that of *XMLParser*, but instead of pushing calls to a callback target, *XMLPullParser* collects an internal list of parsing events and lets the user read from it. *events* is a sequence of events to report back. The supported events are the strings "start", "end", "start-ns" and "end-ns" (the "ns" events are used to get detailed namespace information). If *events* is omitted, only "end" events are reported.

feed(*data*)

将给定的字节数据送入解析器。

close()

通知解析器数据流已终结。不同于*XMLParser.close()*，此方法总是返回*None*。当解析器被关闭时任何还未被获取的事件仍可通过*read_events()* 被读取。

read_events()

Return an iterator over the events which have been encountered in the data fed to the parser. The iterator yields (event, elem) pairs, where *event* is a string representing the type of event (e.g. "end") and *elem* is the encountered *Element* object.

在之前对*read_events()* 的调用中提供的事件将不会被再次产生。事件仅当它们从迭代器中被取出时才会在内部队列中被消费，因此多个读取方对获取自*read_events()* 的迭代器进行平行迭代将产生无法预料的结果。

注解： *XMLPullParser* 只会确保当发出“start”事件时看到了开始标记的“>”字符，因而在这个点上属性已被定义，但文本内容和末尾属性还未被定义。这同样适用于元素的下级；它们可能存在也可能不存在。

如果你需要已完全填充的元素，请改为查找“end”事件。

3.4 新版功能.

异常

class xml.etree.ElementTree.**ParseError**

XML 解析器错误，由此模块中的多个解析方法在解析失败时引发。此异常的实例的字符串表示将包含用户友好的错误消息。此外，它将具有下列可用属性:

code

来自外部解析器的数字错误代码。请参阅*xml.parsers.expat* 的文档查看错误代码列表及它们的含义。

position

一个包含 *line*, *column* 数值的元组，指明错误发生的位置。

备注

20.6 xml.dom —文档对象模型 API

源代码: Lib/xml/dom/__init__.py

文档对象模型“DOM”是一个来自万维网联盟（W3C）的跨语言 API，用于访问和修改 XML 文档。DOM 的实现将 XML 文档以树结构表示，或者允许客户端代码从头构建这样的结构。然后它会通过一组提供通用接口的对象赋予对结构的访问权。

DOM 特别适用于进行随机访问的应用。SAX 仅允许你每次查看文档的一小部分。如果你正在查看一个 SAX 元素，你将不能访问其他元素。如果你正在查看一个文本节点，你将不能访问包含它的元素。当你编写一个 SAX 应用时，你需要在你自己的代码的某个地方记住你的程序在文档中的位置。SAX 不会帮你做这件事。并且，如果你想要在 XML 文档中向前查看，你是绝对办不到的。

有些应用程序在不能访问树的事件驱动模型中是根本无法编写的。当然你可以在 SAX 事件中自行构建某种树，但是 DOM 可以使你避免编写这样的代码。DOM 是针对 XML 数据的标准树表示形式。

文档对象模型是由 W3C 分阶段定义的，在其术语中称为“层级”。Python 中该 API 的映射大致是基于 DOM 第 2 层级的建议。

DOM 应用程序通常从将某些 XML 解析为 DOM 开始。此操作如何实现完全未被 DOM 第 1 层级所涉及，而第 2 层级也只提供了有限的改进：有一个 `DOMImplementation` 对象类，它提供对 `Document` 创建方法的访问，但却没有办法以不依赖具体实现的方式访问 XML 读取器/解析器/文档创建器。也没有当不存在 `Document` 对象的情况下访问这些方法的定义良好的方式。在 Python 中，每个 DOM 实现将提供一个函数 *getDOMImplementation()*。DOM 第 3 层级增加了一个载入/存储规格说明，它定义了与读取器的接口，但这在 Python 标准库中尚不可用。

一旦你得到了 DOM 文档对象，你就可以通过 XML 文档的属性和方法访问它的各个部分。这些属性定义在 DOM 规格说明当中；参考指南的这一部分描述了 Python 对此规格说明的解读。

W3C 提供的规格说明定义了适用于 Java, ECMAScript 和 OMG IDL 的 DOM API。这里定义的 Python 映射很大程度上是基于此规格说明的 IDL 版本，但并不要求严格映射（但具体实现可以自由地支持对 IDL 的严格映射）。请参阅一致性 一节查看有关映射要求的详细讨论。

参见:

Document Object Model (DOM) Level 2 Specification 被 Python DOM API 作为基础的 W3C 建议。

文档对象模型 (DOM) 第 1 层级规格说明 被 *xml.dom.minidom* 所支持的 W3C 针对 DOM 的建议。

Python Language Mapping Specification 此文档指明了从 OMG IDL 到 Python 的映射。

20.6.1 模块内容

xml.dom 包含下列函数:

xml.dom.**registerDOMImplementation**(*name*, *factory*)
注册 *factory* 函数并使用名称 *name*。该工厂函数应当返回一个实现了 `DOMImplementation` 接口的对象。该工厂函数可每次都返回相同对象，或每次调用都返回新的对象，视具体实现的要求而定（例如该实现是否支持某些定制功能）。

xml.dom.**getDOMImplementation**(*name=None*, *features=()*)
返回一个适当的 DOM 实现。*name* 是通用名称、DOM 实现的模块名称或者 `None`。如果它不为 `None`，则会导入相应模块并在导入成功时返回一个 `DOMImplementation` 对象。如果没有给出名称，并且如果设置了 `PYTHON_DOM` 环境变量，此变量会被用来查找相应的实现。

如果未给出 name，此函数会检查可用的实现来查找具有所需特性集的一个。如果找不到任何实现，则会引发 *ImportError*。features 集必须是包含 (feature, version) 对的序列，它会被传给可用的 DOMImplementation 对象上的 hasFeature() 方法。

还提供了一些便捷常量:

xml.dom.**EMPTY_NAMESPACE**
该值用于指明没有命名空间被关联到 DOM 中的某个节点。它通常被作为某个节点的 namespaceURI，或者被用作某个命名空间专属方法的 *namespaceURI* 参数。

xml.dom.**XML_NAMESPACE**
关联到保留前缀 xml 的命名空间 URI，如 XML 中的命名空间 （第 4 节）所定义的。

xml.dom.**XMLNS_NAMESPACE**
命名空间声明的命名空间 URI，如 文档对象模型 (DOM) 第 2 层级核心规格说明 (第 1.1.8 节) 所定义的。

xml.dom.**XHTML_NAMESPACE**
XHTML 命名空间的 URI，如 XHTML 1.0: 扩展超文本标记语言 (第 3.1.1 节) 所定义的。

此外，*xml.dom* 还包含一个基本 Node 类和一些 DOM 异常类。此模块提供的 Node 类未实现 DOM 规格描述所定义的任何方法和属性；实际的 DOM 实现必须提供它们。提供 Node 类作为此模块的一部分并没有提供用于实际的 Node 对象的 nodeType 属性的常量；它们是位于类内而不是位于模块层级以符合 DOM 规格描述。

20.6.2 DOM 中的对象

DOM 的权威文档是来自 W3C 的 DOM 规范。

请注意，DOM 属性也可以作为节点而不是简单的字符串进行操作。然而，必须这样做的情况相当少见，所以这种用法还没有记录下来。

接口	部件	目的
DOMImplementation	*DOMImplementation* 对象	底层实现的接口。
Node	节点对象	文档中大多数对象的基本接口。
NodeList	节点列表对象	节点序列的接口。
DocumentType	文档类型对象	有关处理文档所需声明的信息。
Document	文档对象	表示整个文档的对象。
Element	元素对象	文档层次结构中的元素节点。
Attr	*Attr* 对象	元素节点上的属性值节点。
Comment	注释对象	源文档中注释的表示形式。
Text	*Text* 和 *CDATASection* 对象	包含文档中文本内容的节点。
ProcessingInstruction	*ProcessingInstruction* 对象	处理指令表示形式。

另一节描述了在 Python 中使用 DOM 定义的异常。

DOMImplementation 对象

DOMImplementation 接口提供了一种让应用程序确定他们所使用的 DOM 中某一特性可用性的方式。DOM 第 2 级还添加了使用 DOMImplementation 来创建新的 Document 和 DocumentType 对象的能力。

DOMImplementation.**hasFeature**(*feature*, *version*)
Return true if the feature identified by the pair of strings *feature* and *version* is implemented.

DOMImplementation.**createDocument**(*namespaceUri*, *qualifiedName*, *doctype*)
返回一个新的 Document 对象 (DOM 的根节点)，包含一个具有给定 *namespaceUri* 和 *qualifiedName* 的

下级 Element 对象。*doctype* 必须为由*createDocumentType()* 创建的 DocumentType 对象，或者为 None。在 Python DOM API 中，前两个参数也可为 None 以表示不要创建任何下级 Element。

DOMImplementation.**createDocumentType**(*qualifiedName*, *publicId*, *systemId*)
返回一个新的封装了给定 *qualifiedName*, *publicId* 和 *systemId* 字符串的 DocumentType 对象，它表示包含在 XML 文档类型声明中的信息。

节点对象

XML 文档的所有组成部分都是 Node 的子类。

Node.**nodeType**
一个代表节点类型的整数。类型符号常量在 Node 对象上: ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE, ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE。这是个只读属性。

Node.**parentNode**
当前节点的上级，或者对于文档节点则为 None。该值总是一个 Node 对象或者 None。对于 Element 节点，这将为上级元素，但对于根元素例外，在此情况下它将为 Document 对象。对于 Attr 节点，它将总是为 None。这是个只读属性。

Node.**attributes**
属性对象的 NamedNodeMap。这仅对元素才有实际值；其它对象会为该属性提供 None 值。这是个只读属性。

Node.**previousSibling**
在此节点之前具有相同上级的相邻节点。例如结束标记紧接在在 *self* 元素的开始标记之前的元素。当然，XML 文档并非只是由元素组成，因此之前相邻节点可以是文本、注释或者其他内容。如果此节点是上级的第一个子节点，则该属性将为 None。这是一个只读属性。

Node.**nextSibling**
在此节点之后具有相同上级的相邻节点。另请参见*previousSibling*。如果此节点是上级的最后一个子节点，则该属性将为 None。这是一个只读属性。

Node.**childNodes**
包含在此节点中的节点列表。这是一个只读属性。

Node.**firstChild**
节点的第一个下级，如果有的话，否则为 None。这是个只读属性。

Node.**lastChild**
节点的最后一个下级，如果有的话，否则为 None。这是个只读属性。

Node.**localName**
tagName 在冒号之后的部分，如果有的话，否则为整个 tagName。该值为一个字符串。

Node.**prefix**
tagName 在冒号之前的部分，如果有冒号的话，否则为空字符串。该值为一个字符串或者为 None。

Node.**namespaceURI**
关联到元素名称的命名空间。这将是一个字符串或为 None。这是个只读属性。

Node.**nodeName**
这对于每种节点类型具有不同的含义；请查看 DOM 规格说明来了解详情。你总是可以从其他特征属性例如元素的 tagName 特征属性或属性的 name 特征属性获取你能从这里获取的信息。对于所有节点类型，这个属性的值都将是一个字符串或为 None。这是一个只读属性。

Node.**nodeValue**
这对于每种节点类型具有不同的含义；请查看 DOM 规格说明来了解详情。具体情况与*nodeName* 的类似。该值是一个字符串或为 None。

Node.**hasAttributes**()
Returns true if the node has any attributes.

Node.**hasChildNodes**()
Returns true if the node has any child nodes.

Node.**isSameNode**(*other*)
Returns true if *other* refers to the same node as this node. This is especially useful for DOM implementations which use any sort of proxy architecture (because more than one object can refer to the same node).

注解: 这是基于已提议的 DOM 第 3 等级 API，目前尚处于“起草”阶段，但这个特定接口看来并不存在争议。来自 W3C 的修改将不会影响 Python DOM 接口中的这个方法（不过针对它的任何新 W3C API 也将受到支持）。

Node.**appendChild**(*newChild*)
在子节点列表末尾添加一个新的子节点，返回 *newChild*。如果节点已存在于树结构中，它将先被移除。

Node.**insertBefore**(*newChild*, *refChild*)
在现有的子节点之前插入一个新的子节点。它必须属于 *refChild* 是这个节点的子节点的情况；如果不是，则会引发*ValueError*。*newChild* 会被返回。如果 *refChild* 为 None，它会将 *newChild* 插入到子节点列表的末尾。

Node.**removeChild**(*oldChild*)
移除一个子节点。*oldChild* 必须是这个节点的子节点；如果不是，则会引发*ValueError*。成功时 *oldChild* 会被返回。如果 *oldChild* 将不再被继续使用，则将调用它的 unlink() 方法。

Node.**replaceChild**(*newChild*, *oldChild*)
将一个现有节点替换为新的节点。这必须属于 *oldChild* 是该节点的子节点的情况；如果不是，则会引发*ValueError*。

Node.**normalize**()
合并相邻的文本节点以便将所有文本段存储为单个 Text 实例。这可以简化许多应用程序处理来自 DOM 树文本的操作。

Node.**cloneNode**(*deep*)
克隆此节点。设置 *deep* 表示也克隆所有子节点。此方法将返回克隆的节点。

节点列表对象

NodeList 代表一个节点列表。在 DOM 核心建议中这些对象有两种使用方式: 由 Element 对象提供作为其子节点列表，以及由 Node 的 getElementsByTagName() 和 getElementsByTagNameNS() 方法通过此接口返回对象来表示查询结果。

DOM 第 2 层级建议为这些对象定义一个方法和一个属性:

NodeList.**item**(*i*)
从序列中返回第 *i* 项，如果序列不为空的话，否则返回 None。索引号 *i* 不允许小于零或大于等于序列的长度。

NodeList.**length**
序列中的节点数量。

此外，Python DOM 接口还要求提供一些额外支持来允许将 NodeList 对象用作 Python 序列。所有 NodeList 实现都必须包括对 __len__() 和 __getitem__() 的支持；这样 NodeList 就允许使用 for 语句进行迭代并能正确地支持 *len()* 内置函数。

如果一个 DOM 实现支持文档的修改，则 NodeList 实现还必须支持 __setitem__() 和 __delitem__() 方法。

文档类型对象

有关一个文档所声明的标注和实体的信息（包括解析器所使用并能提供信息的外部子集）可以从 DocumentType 对象获取。文档的 DocumentType 可从 Document 对象的 doctype 属性中获取；如果一个文档没有 DOCTYPE 声明，则该文档的 doctype 属性将被设为 None 而非此接口的一个实例。

DocumentType 是 Node 是专门化，并增加了下列属性:

DocumentType.**publicId**
: 文档类型定义的外部子集的公有标识。这将为一个字符串或者为 None。

DocumentType.**systemId**
: 文档类型定义的外部子集的系统标识。这将为一个字符串形式的 URI，或者为 None。

DocumentType.**internalSubset**
: 一个给出来自文档的完整内部子集的字符串。这不包括子集外面的圆括号。如果文档没有内部子集，则应为 None。

DocumentType.**name**
: DOCTYPE 声明中给出的根元素名称，如果有的话。

DocumentType.**entities**
: 这是给出外部实体定义的 NamedNodeMap。对于多次定义的实体名称，则只提供第一次的定义（其他的会按照 XML 建议被忽略）。这可能为 None，如果解析器未提供此信息，或者如果未定义任何实体的话。

DocumentType.**notations**
: 这是给出标注定义的 NamedNodeMap。对于多次定义的标注，则只提供第一次的定义（其他的会按照 XML 建议被忽略）。这可能为 None，如果解析器未提供此信息，或者如果未定义任何注释的话。

文档对象

Document 代表一个完整的 XML 文档，包括其组成元素、属性、处理指令和注释等。请记住它会继承来自 Node 的属性。

Document.**documentElement**
: 文档唯一的根元素。

Document.**createElement**(*tagName*)
: 创建并返回一个新的元素节点。当元素被创建时不会被插入到文档中。你需要通过某个其他方法例如 insertBefore() 或 appendChild() 来显示地插入它。

Document.**createElementNS**(*namespaceURI*, *tagName*)
: 创建并返回一个新的带有命名空间的元素。*tagName* 可以带有前缀。当元素被创建时不会被插入到文档中。你需要通过某个其他方法例如 insertBefore() 或 appendChild() 来显式地插入它。

Document.**createTextNode**(*data*)
: 创建并返回一个包含作为形参被传入的数据的文本节点。与其他创建方法一样，此方法不会将节点插入到树中。

Document.**createComment**(*data*)
创建并返回一个作为形参被传入的数据的注释节点。与其他创建方法一样，此方法不会将节点插入到树中。

Document.**createProcessingInstruction**(*target*, *data*)
创建并返回一个包含作为形参被传入的 *target* 和 *data* 的处理指令节点。与其他创建方法一样，此方法不会将节点插入到树中。

Document.**createAttribute**(*name*)
创建并返回一个属性节点。此方法不会将属性关联到任何特定的元素。你必须在正确的 Element 对象上使用 setAttributeNode() 来使用新创建的属性实例。

Document.**createAttributeNS**(*namespaceURI*, *qualifiedName*)
创建并返回一个带有命名空间的属性节点。*tagName* 可以带有前缀。此方法不会将属性节点关联到任何特定的元素。你必须在正确的 Element 对象上使用 setAttributeNode() 来使用新创建的属性实例。

Document.**getElementsByTagName**(*tagName*)
搜索全部具有特定元素类型名称的后继元素（直接下级、下级的下级等等）。

Document.**getElementsByTagNameNS**(*namespaceURI*, *localName*)
搜索全部具有特定命名空间 URI 和 localname 的后继元素（直接下级、下级的下级等等）。localname 是命名空间在前缀之后的部分。

元素对象

Element 是 Node 的子类，因此会继承该类的全部属性。

Element.**tagName**
元素类型名称。在使用命名空间的文档中它可能包含冒号。该值是一个字符串。

Element.**getElementsByTagName**(*tagName*)
与 Document 类中的对应方法相同。

Element.**getElementsByTagNameNS**(*namespaceURI*, *localName*)
与 Document 类中的对应方法相同。

Element.**hasAttribute**(*name*)
Returns true if the element has an attribute named by *name*.

Element.**hasAttributeNS**(*namespaceURI*, *localName*)
Returns true if the element has an attribute named by *namespaceURI* and *localName*.

Element.**getAttribute**(*name*)
将名称为 *name* 的属性的值作为字符串返回。如果指定属性不存在，则返回空字符串，就像该属性没有对应的值一样。

Element.**getAttributeNode**(*attrname*)
返回名称为 *attrname* 的属性对应的 Attr 节点。

Element.**getAttributeNS**(*namespaceURI*, *localName*)
将名称为 *namespaceURI* 加 *localName* 的属性的值作为字符串返回。如果指定属性不存在，则返回空字符串，就像该属性没有对应的值一样。

Element.**getAttributeNodeNS**(*namespaceURI*, *localName*)
将给定 *namespaceURI* 加 *localName* 的属性的值作为节点返回。

Element.**removeAttribute**(*name*)
移除指定名称的节点。如果没有匹配的属性，则会引发*NotFoundErr*。

Element.**removeAttributeNode**(*oldAttr*)
从属性列表中移除并返回 *oldAttr*，如果该属性存在的话。如果 *oldAttr* 不存在，则会引发*NotFoundErr*。

Element.**removeAttributeNS**(*namespaceURI*, *localName*)
移除指定名称的属性。请注意它是使用 localName 而不是 qname。如果没有匹配的属性也不会引发异常。

Element.**setAttribute**(*name*, *value*)
将属性值设为指定的字符串。

Element.**setAttributeNode**(*newAttr*)
将一个新的属性节点添加到元素，当匹配到 name 属性时如有必要会替换现有的属性。如果发生了替换，将返回原有属性节点。如果 *newAttr* 已经被使用，则会引发*InuseAttributeErr*。

Element.**setAttributeNodeNS**(*newAttr*)
将一个新的属性节点添加到元素，当匹配到 namespaceURI 和 localName 属性时如有必要会替换现有的属性。如果发生了替换，将返回原有属性节点。如果 *newAttr* 已经被使用，则会引发*InuseAttributeErr*。

Element.**setAttributeNS**(*namespaceURI*, *qname*, *value*)
将属性值设为 *namespaceURI* 和 *qname* 所给出的字符串。请注意 qname 是整个属性名称。这与上面的方法不同。

Attr 对象

Attr 继承自 Node，因此会继承其全部属性。

Attr.**name**
属性名称。在使用命名空间的文档中可能会包括冒号。

Attr.**localName**
名称在冒号之后的部分，如果有的话，否则为完整名称。这是个只读属性。

Attr.**prefix**
名称在冒号之前的部分，如果有冒号的话，否则为空字符串。

Attr.**value**
属性的文本值。这与 nodeValue 属性同义。

NamedNodeMap 对象

NamedNodeMap 不是继承自 Node。

NamedNodeMap.**length**
属性列表的长度。

NamedNodeMap.**item**(*index*)
返回特定带有索引号的属性。获取属性的顺序是强制规定的，但在 DOM 的生命期内会保持一致。其中每一项均为属性节点。可使用 value 属性获取其值。

还有一些试验性方法给予这个类更多的映射行为。你可以使用这些方法或者使用 Element 对象上标准化的 getAttribute*() 方法族。

注释对象

Comment 代表 XML 文档中的注释。它是 Node 的子类，但不能拥有下级节点。

Comment.**data**
注释的内容是一个字符串。该属性包含在开头 <!-- 和末尾 --> 之间的所有字符，但不包括这两个符号。

Text 和 CDATASection 对象

Text 接口代表 XML 文档中的文本。如果解析器和 DOM 实现支持 DOM 的 XML 扩展，则包裹在 CDATA 标记的节中的部分会被存储到 CDATASection 对象中。这两个接口很相似，但是提供了不同的 nodeType 属性值。

这些接口扩展了 Node 接口。它们不能拥有下级节点。

Text.**data**
字符串形式的文本节点内容。

注解： CDATASection 节点的使用并不表示该节点代表一个完整的 CDATA 标记节，只是表示该节点的内容是 CDATA 节的一部分。单个 CDATA 节可以由文档树中的多个节点来表示。没有什么办法能确定两个相邻的 CDATASection 节点是否代表不同的 CDATA 标记节。

ProcessingInstruction 对象

代表 XML 文档中的处理指令。它继承自 Node 接口并且不能拥有下级节点。

ProcessingInstruction.**target**
处理指令的内容至第一个空格符为止。这是个只读属性。

ProcessingInstruction.**data**
在第一个空格符之后的处理指令内容。

异常

DOM 第 2 层级推荐定义一个异常*DOMException*，以及多个变量用来允许应用程序确定发生了何种错误。*DOMException* 实例带有*code* 属性用来提供特定异常的对应值。

Python DOM 接口提供了一些常量，但还扩展了异常集以使 DOM 所定义的每个异常代码都存在特定的异常。接口的具体实现必须引发正确的特定异常，它们各自带有正确的*code* 属性值。

exception xml.dom.**DOMException**
所有特定 DOM 异常所使用的异常基类。该异常类不可被直接实例化。

exception xml.dom.**DomstringSizeErr**
当指定范围的文本不能适配一个字符串时被引发。此异常在 Python DOM 实现中尚不可用，但可从不是以 Python 编写的 DOM 实现中接收。

exception xml.dom.**HierarchyRequestErr**
当尝试插入一个节点但该节点类型不被允许时被引发。

exception xml.dom.**IndexSizeErr**
当一个方法的索引或大小参数为负值或超出允许的值范围时被引发。

exception xml.dom.**InuseAttributeErr**
当尝试插入一个 Attr 节点但该节点已存在于文档中的某处时被引发。

exception xml.dom.**InvalidAccessErr**
当某个参数或操作在底层对象中不受支持时被引发。

exception xml.dom.**InvalidCharacterErr**
当某个字符串参数包含的字符在使用它的上下文中不被 XML 1.0 标准建议所允许时引发。例如，尝试创建一个元素类型名称中带有空格的 Element 节点将导致此错误被引发。

exception xml.dom.**InvalidModificationErr**
当尝试修改某个节点的类型时被引发。

exception xml.dom.**InvalidStateErr**
当尝试使用未定义或不再可用的的对象时被引发。

exception xml.dom.**NamespaceErr**
如果试图以 XML 中的命名空间 建议所不允许的方式修改任何对象，则会引发此异常。

exception xml.dom.**NotFoundErr**
当某个节点不存在于被引用的上下文中时引发的异常。例如，NamedNodeMap.removeNamedItem() 将在所传入的节点不在于于映射中时引发此异常。

exception xml.dom.**NotSupportedErr**
当具体实现不支持所请求的对象类型或操作时被引发。

exception xml.dom.**NoDataAllowedErr**
当为某个不支持数据的节点指定数据时被引发。

exception xml.dom.**NoModificationAllowedErr**
当尝试修改某个不允许修改的对象（例如只读节点）时被引发。

exception xml.dom.**SyntaxErr**
当指定了无效或非法的字符串时被引发。

exception xml.dom.**WrongDocumentErr**
当将某个节点插入非其当前所属的另一个文档，并且具体实现不支持从一个文档向一个文档迁移节点时被引发。

DOM 建议映射中针对上述异常而定义的异常代码如下表所示:

常数	异常
DOMSTRING_SIZE_ERR	*DomstringSizeErr*
HIERARCHY_REQUEST_ERR	*HierarchyRequestErr*
INDEX_SIZE_ERR	*IndexSizeErr*
INUSE_ATTRIBUTE_ERR	*InuseAttributeErr*
INVALID_ACCESS_ERR	*InvalidAccessErr*
INVALID_CHARACTER_ERR	*InvalidCharacterErr*
INVALID_MODIFICATION_ERR	*InvalidModificationErr*
INVALID_STATE_ERR	*InvalidStateErr*
NAMESPACE_ERR	*NamespaceErr*
NOT_FOUND_ERR	*NotFoundErr*
NOT_SUPPORTED_ERR	*NotSupportedErr*
NO_DATA_ALLOWED_ERR	*NoDataAllowedErr*
NO_MODIFICATION_ALLOWED_ERR	*NoModificationAllowedErr*
SYNTAX_ERR	*SyntaxErr*
WRONG_DOCUMENT_ERR	*WrongDocumentErr*

20.6.3 一致性

本节描述了 Python DOM API、W3C DOM 建议以及 Python 的 OMG IDL 映射之间的一致性要求和关系。

类型映射

将根据下表，将 DOM 规范中使用的 IDL 类型映射为 Python 类型。

IDL 类型	Python 类型
`boolean`	`bool` 或 `int`
`int`	`int`
`long int`	`int`
`unsigned int`	`int`
`DOMString`	`str` 或 `bytes`
`null`	`None`

访问器方法

从 OMG IDL 到 Python 的映射以类似于 Java 映射的方式定义了针对 IDL `attribute` 声明的访问器函数。映射以下 IDL 声明

```
readonly attribute string someValue;
         attribute string anotherValue;
```

会产生三个访问器函数: `someValue` 的 “get” 方法 (`_get_someValue()`)，以及 `anotherValue` 的 “get” 和 “set” 方法 (`_get_anotherValue()` 和 `_set_anotherValue()`)。特别地，该映射不要求 IDL 属性像普通 Python 属性那样可访问: `object.someValue` 并非必须可用，并可能引发 *AttributeError*。

但是，Python DOM API 则确实要求普通属性访问可用。这意味着由 Python IDL 解译器生成的典型代理有可能会不可用，如果 DOM 对象是通过 CORBA 来访问则在客户端可能需要有包装对象。虽然这确实要求为 CORBA DOM 客户端进行额外的考虑，但具有从 Python 通过 CORBA 使用 DOM 经验的实现并不会认为这是个问题。已经声明了 `readonly` 的属性不必在所有 DOM 实现中限制写入访问。

在 Python DOM API 中，访问器函数不是必须的。如果提供，则它们应当采用由 Python IDL 映射所定义的形式，但这些方法会被认为不必要，因为这些属性可以从 Python 直接访问。永远都不要为 `readonly` 属性提供 “set” 访问器。

IDL 定义没有完全体现 W3C DOM API 的要求，如特定对象的概念，又如 `getElementsByTagName()` 的返回值为 “live” 等。Python DOM API 并不强制具体实现执行这些要求。

20.7 `xml.dom.minidom` —最小化的 DOM 实现

源代码: Lib/xml/dom/minidom.py

---

xml.dom.minidom 是文档对象模型接口的最小化实现，具有与其他语言类似的 API。它的目标是比完整 DOM 更简单并且更为小巧。对于 DOM 还不十分熟悉的用户则应当考虑改用 *xml.etree.ElementTree* 模块来进行 XML 处理。

警告： *xml.dom.minidom* 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

DOM 应用程序通常会从将某个 XML 解析为 DOM 开始。使用*xml.dom.minidom* 时，这是通过各种解析函数来完成的:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

parse() 函数可接受一个文件名或者打开的文件对象。

xml.dom.minidom.**parse**(*filename_or_file*, *parser=None*, *bufsize=None*)
根据给定的输入返回一个 Document。*filename_or_file* 可以是一个文件名，或是一个文件类对象。如果给定 *parser* 则它必须是一个 SAX2 解析器对象。此函数将修改解析器的处理程序并激活命名空间支持；其他解析器配置（例如设置一个实体求解器）必须已经提前完成。

如果你将 XML 存放为字符串，则可以改用*parseString()* 函数:

xml.dom.minidom.**parseString**(*string*, *parser=None*)
返回一个代表 *string* 的 Document。此方法会为指定字符串创建一个*io.StringIO* 对象并将其传递给*parse()*。

两个函数均返回一个代表文档内容的 Document 对象。object representing the content of the document.

parse() 和*parseString()* 函数所做的是将 XML 解析器连接到一个 “DOM 构建器”，它可以从任意 SAX 解析器接收解析事件并将其转换为 DOM 树结构。这两个函数的名称可能有些误导性，但在学习此接口时是很容易掌握的。文档解析操作将在这两个函数返回之前完成；简单地说这两个函数本身并不提供解析器实现。

你也可以通过在一个 “DOM 实现” 对象上调用方法来创建 Document。此对象可通过调用*xml.dom* 包或者*xml.dom.minidom* 模块中的 getDOMImplementation() 函数来获取。一旦你获得了一个 Document，你就可以向它添加子节点来填充 DOM:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

一旦你得到了 DOM 文档对象，你就可以通过其属性和方法访问对应 XML 文档的各个部分。这些属性定义在 DOM 规格说明当中；文档对象的主要特征属性是 documentElement。它给出了 XML 文档中的主元素：即包含了所有其他元素的元素。以下是一个示例程序:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

当你完成对一个 DOM 树的处理时，你可以选择调用 unlink() 方法以鼓励尽早清除不再需要的对象。unlink() 是*xml.dom.minidom* 针对 DOM API 的专属扩展，它会将特定节点及其下级标记为不再有用。在其他情况下，Python 的垃圾回收器将负责最终处理树结构中的对象。

参见:

文档对象模型 (DOM) 第 1 层级规格说明 被*xml.dom.minidom*所支持的 W3C 针对 DOM 的建议。

20.7.1 DOM 对象

Python 的 DOM API 定义被作为*xml.dom*模块文档的一部分给出。这一节列出了该 API 和*xml.dom.minidom*之间的差异。

Node.**unlink**()
破坏 DOM 的内部引用以便它能在没有循环 GC 的 Python 版本上垃圾回收器回收。即使在循环 GC 可用的时候，使用此方法也可让大量内存更快变为可用，因此当 DOM 对象不再被需要时尽早调用它们的这个方法是很好的做法。此方法只须在 Document 对象上调用，但也可以在下级节点上调用以丢弃该节点的下级节点。

You can avoid calling this method explicitly by using the with statement. The following code will automatically unlink *dom* when the with block is exited:

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # Work with dom.
```

Node.**writexml**(*writer*, *indent=""*, *addindent=""*, *newl=""*)
Write XML to the writer object. The writer should have a write() method which matches that of the file object interface. The *indent* parameter is the indentation of the current node. The *addindent* parameter is the incremental indentation to use for subnodes of the current one. The *newl* parameter specifies the string to use to terminate newlines.

对于 Document 节点，可以使用附加的关键字参数 *encoding* 来指定 XML 标头的编码格式字段。

Node.**toxml**(*encoding=None*)
返回一个包含 XML DOM 节点所代表的 XML 的字符串或字节串。

带有显式的 *encoding*[1] 参数时，结果为使用指定编码格式的字节串。没有 *encoding* 参数时，结果为 Unicode 字符串，并且结果字符串中的 XML 声明将不指定编码格式。使用 UTF-8 以外的编码格式对此字符串进行编码通常是不正确的，因为 UTF-8 是 XML 的默认编码格式。

Node.**toprettyxml**(*indent="\t"*, *newl="\n"*, *encoding=None*)
返回文档的美化打印版本。*indent* 指定缩进字符串并默认为制表符；*newl* 指定标示每行结束的字符串并默认为 \n。

encoding 参数的行为类似于*toxml()*的对应参数。

20.7.2 DOM 示例

此示例程序是个相当实际的简单程序示例。在这个特定情况中，我们没有过多地利用 DOM 的灵活性。

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
```

(下页继续)

[1] 包括在 XML 输出中的编码格式名称应当遵循适当的标准。例如，" UTF-8" 是有效的，但 "UTF8" 在 XML 文档的声明中是无效的，即使 Python 接受其作为编码格式名称。详情参见 https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 和 https://www.iana.org/assignments/character-sets/character-sets.xhtml。

(续上页)

```
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print("<title>%s</title>" % getText(title.childNodes))

def handleSlideTitle(title):
    print("<h2>%s</h2>" % getText(title.childNodes))

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")

def handlePoint(point):
    print("<li>%s</li>" % getText(point.childNodes))

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print("<p>%s</p>" % getText(title.childNodes))

handleSlideshow(dom)
```

20.7.3 minidom 和 DOM 标准

xml.dom.minidom 模块实际上是兼容 DOM 1.0 的 DOM 并带有部分 DOM 2 特性（主要是命名空间特性）。

Python 中 DOM 接口的用法十分直观。会应用下列映射规则:

- 接口是通过实例对象来访问的。应用程序不应实例化这些类本身；它们应当使用 `Document` 对象提供的创建器函数。派生的接口支持上级接口的所有操作（和属性），并添加了新的操作。
- 操作以方法的形式使用。因由 DOM 只使用 `in` 形参，参数是以正常顺序传入的（从左至右）。不存在可选参数。`void` 操作返回 `None`。
- IDL 属性会映射到实例属性。为了兼容针对 Python 的 OMG IDL 语言映射，属性 `foo` 也可通过访问器方法 `_get_foo()` 和 `_set_foo()` 来访问。`readonly` 属性不可被修改；运行时并不强制要求这一点。
- `short int`, `unsigned int`, `unsigned long long` 和 `boolean` 类型都会映射到 Python 整数类型。
- `DOMString` 类型会映射为 Python 字符串。*xml.dom.minidom* 支持字节串或字符串，但通常是产生字符串。`DOMString` 类型的值也可以为 `None`，W3C 的 DOM 规格说明允许其具有 IDL `null` 值。
- `const` 声明会映射为它们各自的作用域内的变量 (例如 `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`)；它们不可被修改。
- `DOMException` 目前不被*xml.dom.minidom* 所支持。*xml.dom.minidom* 会改为使用标准 Python 异常例如*TypeError* 和*AttributeError*。
- `NodeList` 对象是使用 Python 内置列表类型来实现的。这些对象提供了 DOM 规格说明中定义的接口，但在较早版本的 Python 中它们不支持官方 API。相比在 W3C 建议中定义的接口，它们要更加的“Pythonic”。

下列接口未在*xml.dom.minidom* 中实现:

- `DOMTimeStamp`
- `DocumentType`
- `DOMImplementation`
- `CharacterData`
- `CDATASection`
- `Notation`
- `Entity`
- `EntityReference`
- `DocumentFragment`

这些接口所反映的 XML 文档信息对于大多数 DOM 用户来说没有什么帮助。

备注

20.8 xml.dom.pulldom —支持构建部分 DOM 树

源代码: Lib/xml/dom/pulldom.py

---

xml.dom.pulldom 模块提供了一个“拉取解析器”，它能在必要时被用于产生文件的可访问 DOM 的片段。其基本概念包括从输入的 XML 流拉取“事件”并处理它们。与同样地同时应用了事件驱动处理模型加回调函数的 SAX 不同，拉取解析器的用户要负责显式地从流拉取事件，并循环遍历这些事件直到处理结束或者发生了错误条件。

警告: *xml.dom.pulldom* 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

在 3.6.7 版更改: SAX 解析器默认不再处理一般外部实体以提升在默认情况下的安全性。要启用外部实体处理，请传入一个自定义的解析器实例:

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

示例:

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

event 是一个常量，可以取下列值之一:

- START_ELEMENT
- END_ELEMENT
- COMMENT
- START_DOCUMENT
- END_DOCUMENT
- CHARACTERS
- PROCESSING_INSTRUCTION
- IGNORABLE_WHITESPACE

node 是一个 xml.dom.minidom.Document, xml.dom.minidom.Element 或 xml.dom.minidom.Text 类型的对象。

由于文档是被当作“展平”的事件流来处理的，文档“树”会被隐式地遍历并且无论所需元素在树中的深度如何都会被找到。换句话说，不需要考虑层级问题，例如文档节点的递归搜索等，但是如果元素的内容很重要，则有必要保留一些上下文相关的状态（例如记住任意给定点在文档中的位置）或者使用*DOMEventStream.expandNode()* 方法并切换到 DOM 相关的处理过程。

class xml.dom.pulldom.**PullDom**(*documentFactory=None*)
xml.sax.handler.ContentHandler 的子类。

class xml.dom.pulldom.**SAX2DOM**(*documentFactory=None*)
xml.sax.handler.ContentHandler 的子类。

xml.dom.pulldom.**parse**(*stream_or_string*, *parser=None*, *bufsize=None*)
基于给定的输入返回一个*DOMEventStream*。*stream_or_string* 可以是一个文件名，或是一个文件类对象。*parser* 如果给出，则必须是一个*XMLReader* 对象。此函数将改变解析器的文档处理程序并激活命名空间支持；其他解析器配置（例如设置实体解析器）必须在之前已完成。

如果你将 XML 存放为字符串，则可以改用*parseString()* 函数:

xml.dom.pulldom.**parseString**(*string*, *parser=None*)
返回一个*DOMEventStream* 来表示 (Unicode) *string*。

xml.dom.pulldom.**default_bufsize**
将 *bufsize* 形参的默认值设为*parse()*。

此变量的值可在调用*parse()* 之前修改并使新值生效。

20.8.1 DOMEventStream 对象

class xml.dom.pulldom.**DOMEventStream**(*stream*, *parser*, *bufsize*)

getEvent()
返回一个元组，其中包含 *event* 和 xml.dom.minidom.Document 形式的当前 *node* 如果 event 等于 START_DOCUMENT，包含 xml.dom.minidom.Element 如果 event 等于 START_ELEMENT 或 END_ELEMENT 或者 xml.dom.minidom.Text 如果 event 等于 CHARACTERS。当前 node 不包含有关其子节点的信息，除非*expandNode()* 被调用。

expandNode(*node*)
将 *node* 的所有子节点扩展到 *node* 中。例如:

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # Following statement only prints '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its children '<p>Some text
↪<div>and more</div></p>'
        print(node.toxml())
```

reset()

20.9 xml.sax —支持 SAX2 解析器

源代码: Lib/xml/sax/__init__.py

---

xml.sax 包提供多个模块，它们在 Python 上实现了用于 XML (SAX) 接口的简单 API。这个包本身为 SAX API 用户提供了一些最常用的 SAX 异常和便捷函数。

> **警告:** *xml.sax* 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

在 3.6.7 版更改: SAX 解析器默认不会再处理通用外部实体以便提升安全性。在此之前，解析器会创建网络连接来获取远程文件或是从 DTD 和实体文件系统中加载本地文件。此特性可通过在解析器对象上调用*setFeature()* 对象并传入参数*feature_external_ges* 来重新启用。

可用的便捷函数如下所列:

xml.sax.**make_parser**(*parser_list=[]*)
Create and return a SAX *XMLReader* object. The first parser found will be used. If *parser_list* is provided, it must be a list of strings which name modules that have a function named `create_parser()`. Modules listed in *parser_list* will be used before modules in the default list of parsers.

xml.sax.**parse**(*filename_or_stream*, *handler*, *error_handler=handler.ErrorHandler()*)
创建一个 SAX 解析器并用它来解析文档。用于传入文档的 *filename_or_stream* 可以是一个文件名或文件对象。*handler* 形参必须是一个 SAX *ContentHandler* 实例。如果给出了 *error_handler*，则它必须是一个 SAX *ErrorHandler* 实例；如果省略，则对于任何错误都将引发*SAXParseException*。此函数没有返回值；所有操作必须由传入的 *handler* 来完成。

xml.sax.**parseString**(*string*, *handler*, *error_handler=handler.ErrorHandler()*)
类似于*parse()*，但解析对象是作为形参传入的缓冲区 *string*。*string* 必须为*str* 实例或者*bytes-like object*。

在 3.5 版更改: 增加了对*str* 实例的支持。

典型的 SAX 应用程序会使用三种对象：读取器、处理句柄和输入源。“读取器”在此上下文中与解析器同义，即某个从输入源读取字节或字符，并产生事件序列的代码段。事件随后将被分发给处理句柄对象，即由读取器发起调用处理句柄上的某个方法。因此 SAX 应用程序必须获取一个读取器对象，创建或打开输入源，创建处理句柄，并一起连接到这些对象。作为准备工作的最后一步，将调用读取器来解析输入内容。在解析过程中，会根据来自输入数据的结构化和语义化事件来调用处理句柄对象上的方法。

就这些对象而言，只有接口部分是需要关注的；它们通常不是由应用程序本身来实例化。由于 Python 没有显式的接口标记法，它们的正式引入形式是类，但应用程序可能会使用并非从已提供的类继承而来的实现。*InputSource*, *Locator*, `Attributes`, `AttributesNS` 以及*XMLReader* 接口是在*xml.sax.xmlreader* 模块中定义的。处理句柄接口是在*xml.sax.handler* 中定义的。为了方便起见，*InputSource* (它往往会被直接实例化) 和处理句柄类也可以从*xml.sax* 获得。这些接口的描述如下。

除了这些类，*xml.sax* 还提供了如下异常类。

exception xml.sax.**SAXException**(*msg*, *exception=None*)
封装某个 XML 错误或警告。这个类可以包含来自 XML 解析器或应用程序的基本错误或警告信息：它可以被子类化以提供额外的功能或是添加本地化信息。请注意虽然在*ErrorHandler* 接口中定义的处理句柄可以接收该异常的实例，但是并不要求实际引发该异常—它也可以被用作信息的容器。

当实例化时，*msg* 应当是适合人类阅读的错误描述。如果给出了可选的 *exception* 形参，它应当为 `None` 或者解析代码所捕获的异常并会被作为信息传递出去。

这是其他 SAX 异常类的基类。

exception xml.sax.**SAXParseException**(*msg*, *exception*, *locator*)
SAXException 的子类，针对解析错误引发。这个类的实例会被传递给 SAX *ErrorHandler* 接口的方法来提供关于解析错误的信息。这个类支持 SAX *Locator* 接口以及*SAXException* 接口。

exception xml.sax.**SAXNotRecognizedException**(*msg*, *exception=None*)
SAXException 的子类，当 SAX *XMLReader* 遇到不可识别的特性或属性时引发。SAX 应用程序和扩展可能会出于类似目的而使用这个类。

exception xml.sax.**SAXNotSupportedException**(*msg*, *exception=None*)
SAXException 的子类，当 SAX *XMLReader* 被要求启用某个不受支持的特性，或者将某个属性设为具体实现不支持的值时引发。SAX 应用程序和扩展可能会出于类似目的而使用这个类。

参见:

SAX: The Simple API for XML 这个网站是 SAX API 定义的焦点。它提供了一个 Java 实现以及在线文档。还包括其他实现的链接和历史信息。

***xml.sax.handler* 模块** 应用程序所提供对象的接口定义。

***xml.sax.saxutils* 模块** 可在 SAX 应用程序中使用的便捷函数。

***xml.sax.xmlreader* 模块** 解析器所提供对象的接口定义。

20.9.1 SAXException 对象

SAXException 异常类支持下列方法:

SAXException.**getMessage**()
返回描述错误条件的适合人类阅读的消息。

SAXException.**getException**()
返回一个封装的异常对象或者 None。

20.10 xml.sax.handler —SAX 处理程序的基类

源代码: Lib/xml/sax/handler.py

SAX API 定类了四种处理句柄：内容句柄、DTD 句柄、错误句柄以及实体解析器。应用程序通常只需要实现他们感兴趣的事件对应的接口；他们可以在单个对象或多个对象中实现这些接口。处理句柄的实现应当继承自在*xml.sax.handler* 模块中提供的基类，以便所有方法都能获得默认的实现。

class xml.sax.handler.**ContentHandler**
这是 SAX 中的主回调接口，也是对应用程序来说最重要的一个接口。此接口中事件的顺序反映了文档中信息的顺序。

class xml.sax.handler.**DTDHandler**
处理 DTD 事件。

这个接口仅指定了基本解析（未解析的实体和属性）所需的那些 DTD 事件。

class xml.sax.handler.**EntityResolver**
用于解析实体的基本接口。如果你创建了实现此接口的对象，然后用你的解析器注册该对象，该解析器将调用你的对象中的方法来解析所有外部实体。

class xml.sax.handler.**ErrorHandler**
解析器用来向应用程序表示错误和警告的接口。这个对象的方法控制错误是要立即转换为异常还是以某种其他该来来处理。

除了这些类，*xml.sax.handler* 还提供了表示特性和属性名称的符号常量。

xml.sax.handler.**feature_namespaces**

value: "http://xml.org/sax/features/namespaces"
true: 执行命名空间处理。
false: 可选择不执行命名空间处理 (这意味着 namespace-prefixes; default)。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_namespace_prefixes**

value: "http://xml.org/sax/features/namespace-prefixes"
true: 报告用于命名空间声明的原始带前缀名称和属性。
false: 不报告用于命名空间声明的属性，可选择不报告原始带前缀名称（默认）。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_string_interning**

value: "http://xml.org/sax/features/string-interning"
true: 所有元素名称、前缀、属性名称、命名空间 URI 以及本地名称都使用内置的 intern 函数进行内化。
false: 名称不要求被内化，但也可以被内化（默认）。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_validation**

value: "http://xml.org/sax/features/validation"
true: 报告所有的验证错误（包括 external-general-entities 和 external-parameter-entities）。
false: 不报告验证错误。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_external_ges**

value: "http://xml.org/sax/features/external-general-entities"
true: 包括所有的外部通用（文本）实体。
false: 不包括外部通用实体。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_external_pes**

value: "http://xml.org/sax/features/external-parameter-entities"
true: 包括所有的外部参数实体，包括外部 DTD 子集。
false: 不包括任何外部参数实体，也不包括外部 DTD 子集。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**all_features**
全部特性列表。

xml.sax.handler.**property_lexical_handler**

value: "http://xml.org/sax/properties/lexical-handler"
数据类型: xml.sax.sax2lib.LexicalHandler (在 Python 2 中不受支持)

描述: 可选的扩展处理句柄，用于注释等词法事件。
访问: 读/写

xml.sax.handler.**property_declaration_handler**

值: "http://xml.org/sax/properties/declaration-handler"
数据类型: xml.sax.sax2lib.DeclHandler (在 Python 2 中不受支持)
描述: 可选的扩展处理句柄，用于标注和未解析实体以外的 DTD 相关事件。
访问: 读/写

xml.sax.handler.**property_dom_node**

值: "http://xml.org/sax/properties/dom-node"
数据类型: org.w3c.dom.Node (在 Python 2 中不受支持)
描述: 在解析时，如果这是一个 DOM 迭代器则为当前被访问的 DOM 节点；不在解析时，则将根 DOM 节点用于迭代。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**property_xml_string**

值: "http://xml.org/sax/properties/xml-string"
数据类型: String
描述: 作为当前事件来源的字符串字面值。
访问: 只读

xml.sax.handler.**all_properties**
已知属性名称列表。

20.10.1 ContentHandler 对象

用户应当子类化*ContentHandler* 来支持他们的应用程序。以下方法会由解析器在输入文档的适当事件上调用:

ContentHandler.**setDocumentLocator**(*locator*)
由解析器调用来给予应用程序一个定位器以确定文档事件来自何处。

强烈建议（虽然不是绝对的要求）SAX 解析器提供一个定位器：如果提供的话，它必须在发起调用 DocumentHandler 接口的任何其他方法之前通过发起调用此方法来提供定位器。

定位器允许应用程序确定任何文档相关事件的结束位置，即使解析器没有报告错误。通常，应用程序将使用这些信息来报告它自己的错误（例如未匹配到应用程序业务规则的字符内容）。定位器所返回的信息可能不足以与搜索引擎配合使用。

请注意定位器只有在发起调用此接口中的事件时才会返回正确的信息。应用程序不应试图在其他任何时刻使用它。

ContentHandler.**startDocument**()
接收一个文档开始的通知。

SAX 解析器将只发起调用这个方法一次，并且会在调用这个接口或 DTDHandler 中的任何其他方法之前 (*setDocumentLocator()* 除外)。

ContentHandler.**endDocument**()
接收一个文档结束的通知。

SAX 解析器将只发起调用这个方法一次，并且它将是在解析过程中最后发起调用的方法。解析器在（因不可恢复的错误）放弃解析或到达输入的终点之前不应发起调用这个方法。

ContentHandler.**startPrefixMapping**(*prefix*, *uri*)
开始一个前缀 URI 命名空间映射的范围。

来自此事件的信息对于一般命名空间处理来说是不必要的：当 feature_namespaces 特性被启用时（默认）SAX XML 读取器将自动为元素和属性名称替换前缀。

但是也存在一些情况，当应用程序需要在字符数据或属性值中使用前缀，而它们无法被安全地自动扩展；*startPrefixMapping()* 和*endPrefixMapping()* 事件会向应用程序提供信息以便在这些上下文内部扩展前缀，如果有必要的话。

请注意*startPrefixMapping()* 和*endPrefixMapping()* 事件并不保证能够相对彼此被正确地嵌套：所有*startPrefixMapping()* 事件都将在对应的*startElement()* 事件之前发生，而所有*endPrefixMapping()* 事件都将在对应的*endElement()* 事件之后发生，但它们的并不保证一致。

ContentHandler.**endPrefixMapping**(*prefix*)
结束一个前缀 URI 映射的范围。

请参看*startPrefixMapping()* 了解详情。此事件将总是会在对应的*endElement()* 事件之后发生，但*endPrefixMapping()* 事件的顺序则并没有保证。

ContentHandler.**startElement**(*name*, *attrs*)
在非命令空间模式下指示一个元素的开始。

name 形参包含字符串形式的元素类型原始 XML 1.0 名称而 *attrs* 形参存放包含元素属性的 Attributes 接口对象 (参见*Attributes 接口*)。作为 *attrs* 传入的对象可能被解析器所重用；维持一个对它的引用不是保持属性副本的可靠方式。要保持这些属性的一个副本，请使用 *attrs* 对象的*copy()* 方法。

ContentHandler.**endElement**(*name*)
在非命名空间模式下指示一个元素的结束。

name 形参包含元素类型的名称，与*startElement()* 事件的一样。

ContentHandler.**startElementNS**(*name*, *qname*, *attrs*)
在命名空间模式下指示一个元素的开始。

name 形参包含以 (uri, localname) 元组表示的元素类型名称，*qname* 形参包含源文档中使用的原始 XML 1.0 名称，而 *attrs* 形参存放包含元素属性的 AttributesNS 接口实例 (参见*AttributesNS 接口*)。如果没有命名空间被关联到元素，则 *name* 的 *uri* 部分将为 None。作为 *attrs* 传入的对象可能被解析器所重用；维持一个对它的引用不是保持属性副本的可靠方式。要保持这些属性的一个副本，请使用 *attrs* 对象的*copy()* 方法。

解析器可将 *qname* 形参设为 None，除非 feature_namespace_prefixes 特性已被激活。

ContentHandler.**endElementNS**(*name*, *qname*)
在命名空间模式下指示一个元素的结束。

name 形参包含元素类型的名称，与*startElementNS()* 方法的一样，*qname* 形参也是类似的。

ContentHandler.**characters**(*content*)
接收字符数据的通知。

解析器将调用此方法来报告每一个字符数据分块。SAX 解析器可以将所有连续字符数据返回为一个单独分块，或者将其拆成几个分块；但是，在任意单个事件中的所有字符都必须来自同一个外部实体以便定位器提供有用的信息。

content 可以是一个字符串或字节串实例；expat 读取器模块总是会产生字符串。

注解： Python XML 特别关注小组所提供的早期 SAX 1 接口针对此方法使用了一个更类似于 Java 的接口。由于 Python 所使用的大多数解析器都没有利用老式的接口，因而选择了更简单的签名来替代它。要将旧代码转换为新接口，请使用 *content* 而不要通过旧的 *offset* 和 *length* 形参来对内容进行切片。

ContentHandler.**ignorableWhitespace**(*whitespace*)
接收元素内容中可忽略空白符的通知。

验证解析器必须使用此方法来报告每个可忽略的空白符分块（参见 W3C XML 1.0 建议第 2.10 节）：非验证解析器如果能够解析并使用内容模型的话也可以使用此方法。

SAX 解析器可以将所有连续字符数据返回为一个单独分块，或者将其拆成几个分块；但是，在任意单个事件中的所有字符都必须来自同一个外部实体以便定位器提供有用的信息。

ContentHandler.**processingInstruction**(*target*, *data*)
接受一条处理指令的通知。

解析器将为已找到的每条处理指令发起调用该方法一次：请注意处理指令可能出现在主文档元素之前或之后。

SAX 解析器绝不应当使用此方法来报告 XML 声明（XML 1.0 第 2.8 节）或文本声明（XML 1.0 第 4.3.1 节）。

ContentHandler.**skippedEntity**(*name*)
接收一个已跳过实体的通知。

解析器将为每个已跳过实体发起调用此方法一次。非验证处理程序可能会跳过未看到声明的实体（例如，由于实体是在一个外部 because, for example, the entity was declared in an external DTD 子集中声明的）。所有处理程序都可以跳过外部实体，具体取决于 `feature_external_ges` 和 `feature_external_pes` 属性的值。

20.10.2 DTDHandler 对象

DTDHandler 实例提供了下列方法:

DTDHandler.**notationDecl**(*name*, *publicId*, *systemId*)
处理标注声明事件。

DTDHandler.**unparsedEntityDecl**(*name*, *publicId*, *systemId*, *ndata*)
处理未解析的实体声明事件。

20.10.3 EntityResolver 对象

EntityResolver.**resolveEntity**(*publicId*, *systemId*)
求解一个实体的系统标识符并返回一个字符串形式的系统标识符作为读取源，或是一个 InputSource 作为读取源。默认的实现会返回 *systemId*。

20.10.4 ErrorHandler 对象

带有这个接口的对象被用于接收来自*XMLReader*的错误和警告信息。如果你创建了一个实现这个接口的对象，然后用你的*XMLReader*注册这个对象，则解析器将调用你的对象中的这个方法来报告所有的警告和错误。有三个可用的错误级别：警告、（或许）可恢复的错误和不可恢复的错误。所有方法都接受SAXParseException作为唯一的形参。错误和警告可以通过引发所传入的异常对象来转换为异常。

ErrorHandler.**error**(*exception*)
当解析器遇到一个可恢复错误时调用。如果此方法没有引发异常，则解析可能会继续，但是应用程序不能预期获得更多的文档信息。允许解析器继续可能会允许在输入文档中发现额外的错误。

ErrorHandler.**fatalError**(*exception*)
当解析器遇到一个无法恢复的错误时调用；在此方法返回时解析应当终止。

ErrorHandler.**warning**(*exception*)
当解析器向应用程序提供次要警告信息时调用。在此方法返回时解析应当继续，并且文档信息将继续被传递给应用程序。在此方法中引发异常将导致解析结束。

20.11 xml.sax.saxutils —SAX 工具集

源代码: Lib/xml/sax/saxutils.py

---

*xml.sax.saxutils*模块包含一些在创建 SAX 应用程序时十分有用的类和函数，它们可以被直接使用，或者是作为基类使用。

xml.sax.saxutils.**escape**(*data*, *entities={}*)
对数据字符串中对'&'，'<'和'>'进行转义。

你可以通过传入一个字典作为可选的*entities*形参来对其他字符串数据进行转义。字典的键和值必须为字符串；每个键将被替换为其所对应的值。字符'&'，'<'和'>'总是会被转义，即使提供了*entities*。

xml.sax.saxutils.**unescape**(*data*, *entities={}*)
对字符串数据中的'&'，'<'和'>'进行反转义。

你可以通过传入一个字典作为可选的*entities*形参来对其他数据字符串进行转义。字典的键和值必须都为字符串；每个键将被替换为所对应的值。'&amp'，'<'和'>'将总是保持不被转义，即使提供了*entities*。

xml.sax.saxutils.**quoteattr**(*data*, *entities={}*)
类似于*escape()*，但还会对*data*进行处理以将其用作属性值。返回值是*data*加上任何额外要求的替换的带引号版本。*quoteattr()*将基于*data*的内容选择一个引号字符，以尽量避免在字符串中编码任何引号字符。如果单双引号字符在*data*中都存在，则双引号字符将被编码并且*data*将使用双引号来标记。结果字符串可被直接用作属性值:

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

此函数适用于为 HTML 或任何使用引用实体语法的 SGML 生成属性值。

class xml.sax.saxutils.**XMLGenerator**(*out=None*, *encoding='iso-8859-1'*, *short_empty_elements=False*)
这个类通过将 SAX 事件写回到 XML 文档来实现*ContentHandler*接口。换句话说，使用*XMLGenerator*作为内容处理程序将重新产生所解析的原始文档。*out*应当为一个文件类对象，它默认将为*sys.stdout*。*encoding*为输出流的编码格式，它默认将为'iso-8859-1'。*short_empty_elements*控制不包含内容的元素的格式化：如为False(默认值)则它们会以开始/结束标记对的形式被发送，如果设为True则它们会以单个自结束标记的形式被发送。

3.2 新版功能: *short_empty_elements* 形参。

class xml.sax.saxutils.**XMLFilterBase**(*base*)
这个类被设计用来分隔*XMLReader*和客户端应用的事件处理程序。在默认情况下，它除了将请求传送给读取器并将事件传送给处理程序之外什么都不做，但其子类可以重载特定的方法以在传送它们的时候修改事件流或配置请求。

xml.sax.saxutils.**prepare_input_source**(*source*, *base=''*)
此函数接受一个输入源和一个可选的基准 URL 并返回一个经过完整解析可供读取的*InputSource*。输入源的给出形式可以为字符串、文件类对象或*InputSource*对象；解析器将使用此函数来针对它们的 parse() 方法实现多态 *source* 参数。

20.12 xml.sax.xmlreader —用于 XML 解析器的接口

源代码: Lib/xml/sax/xmlreader.py

SAX 解析器实现了*XMLReader*接口。它们是在一个 Python 模块中实现的，该模块必须提供一个 create_parser() 函数。该函数由*xml.sax.make_parser()*不带参数地发起调用来创建新的解析器对象。

class xml.sax.xmlreader.**XMLReader**
可由 SAX 解析器继承的基类。

class xml.sax.xmlreader.**IncrementalParser**
在某些情况下，最好不要一次性地解析输入源，而是在可用的时候分块送入。请注意读取器通常不会读取整个文件，它同样也是分块读取的；并且 parse() 在处理完整个文档之前不会返回。所以如果不希望 parse() 出现阻塞行为则应当使用这些接口。

当解析器被实例化时它已准备好立即开始接受来自 feed 方法的数据。在通过调用 close 方法结束解析时 reset 方法也必须被调用以使解析器准备好接受新的数据，无论它是来自于 feed 还是使用 parse 方法。

请注意这些方法 不可在解析期间被调用，即在 parse 被调用之后及其返回之前。

默认情况下，该类还使用 IncrementalParser 接口的 feed, close 和 reset 方法来实现 XMLReader 接口的 parse 方法以方便 SAX 2.0 驱动的编写者。

class xml.sax.xmlreader.**Locator**
用于关联一个 SAX 事件与一个文档位置的接口。定位器对象只有在调用 DocumentHandler 的方法期间才会返回有效的结果；在其他任何时候，结果都是不可预测的。如果信息不可用，这些方法可能返回 None。

class xml.sax.xmlreader.**InputSource**(*system_id=None*)
*XMLReader*读取实体所需信息的封装。

这个类可能包括了关于公有标识符、系统标识符、字节流（可能带有字符编码格式信息）和/或一个实体的字符流的信息。

应用程序将创建这个类的对象以便在*XMLReader.parse()*方法中使用或是用于从 EntityResolver.resolveEntity 返回值。

*InputSource*属于应用程序，*XMLReader*不能修改从应用程序传递给它的*InputSource*对象，但它可以创建副本并进行修改。

class xml.sax.xmlreader.**AttributesImpl**(*attrs*)
这是 Attributes 接口（参见*Attributes 接口*一节）的具体实现。这是一个 startElement() 调用中的元素属性的字典类对象。除了最有用处的字典操作，它还支持接口所描述的一些其他方法。该类的对象应当由读取器来实例化；*attrs* 必须为包含从属性名到属性值的映射的字典类对象。

class xml.sax.xmlreader.**AttributesNSImpl**(*attrs*, *qnames*)
可感知命名空间的*AttributesImpl* 变体形式，它将被传递给 startElementNS()。它派生自*AttributesImpl*，但会将属性名称解读为 *namespaceURI* 和 *localname* 二元组。此外，它还提供了一些期望接收在原始文档中出现的限定名称的方法。这个类实现了 AttributesNS 接口（参见*AttributesNS 接口* 一节）。

20.12.1 XMLReader 对象

XMLReader 接口支持下列方法:

XMLReader.**parse**(*source*)
Process an input source, producing SAX events. The *source* object can be a system identifier (a string identifying the input source –typically a file name or a URL), a file-like object, or an *InputSource* object. When *parse()* returns, the input is completely processed, and the parser object can be discarded or reset.

在 3.5 版更改: 添加了对字符流的支持。

XMLReader.**getContentHandler**()
返回当前的*ContentHandler*。

XMLReader.**setContentHandler**(*handler*)
设置当前的*ContentHandler*。如果没有设置*ContentHandler*，内容事件将被丢弃。

XMLReader.**getDTDHandler**()
返回当前的*DTDHandler*。

XMLReader.**setDTDHandler**(*handler*)
设置当前的*DTDHandler*。如果没有设置*DTDHandler*，DTD 事件将被德育。

XMLReader.**getEntityResolver**()
返回当前的*EntityResolver*。

XMLReader.**setEntityResolver**(*handler*)
设置当前的*EntityResolver*。如果没有设置*EntityResolver*，尝试解析一个外部实体将导致打开该实体的系统标识符，并且如果它不可用则操作将失败。

XMLReader.**getErrorHandler**()
返回当前的*ErrorHandler*。

XMLReader.**setErrorHandler**(*handler*)
设置当前的错误处理句柄。如果没有设置*ErrorHandler*，错误将作为异常被引发，并将打印警告信息。

XMLReader.**setLocale**(*locale*)
允许应用程序为错误和警告设置语言区域。

SAX 解析器不要求为错误和警告提供本地化信息；但是如果它们无法支持所请求的语言区域，则必须引发一个 SAX 异常。应用程序可以在解析的中途请求更改语言区域。

XMLReader.**getFeature**(*featurename*)
返回 *featurename* 特性的当前设置。如果特性无法被识别，则会引发 SAXNotRecognizedException。在*xml.sax.handler* 模块中列出了常见的特性名称。

XMLReader.**setFeature**(*featurename*, *value*)
将 *featurename* 设为 *value*。如果特性无法被识别，则会引发 SAXNotRecognizedException。如果特性或其设置不被解析器所支持，则会引发 *SAXNotSupportedException*。

XMLReader.**getProperty**(*propertyname*)
返回 *propertyname* 属性的当前设置。如果属性无法被识别，则会引发 SAXNotRecognizedException。在*xml.sax.handler* 模块中列出了常见的属性名称。

XMLReader.**setProperty**(*propertyname*, *value*)
将 *propertyname* 设为 *value*。如果属性无法被识别，则会引发 SAXNotRecognizedException。如果属性或其设置不被解析器所支持，则会引发 *SAXNotSupportedException*。

20.12.2 IncrementalParser 对象

IncrementalParser 的实例额外提供了下列方法:

IncrementalParser.**feed**(*data*)
处理 *data* 的一个分块。

IncrementalParser.**close**()
确定文档的结尾。这将检查只能在结尾处检查的格式是否良好的条件，发起调用处理程序，并可能会清理在解析期间分配的资源。

IncrementalParser.**reset**()
此方法会在调用 close 来重置解析器以便其准备好解析新的文档之后被调用。在 close 之后未调用 reset 即调用 parse 或 feed 的结果是未定义的。

20.12.3 Locator 对象

Locator 的实例提供了下列方法:

Locator.**getColumnNumber**()
返回当前事件开始位置的列号。

Locator.**getLineNumber**()
返回当前事件开始位置的行号。

Locator.**getPublicId**()
返回当前事件的公有标识符。

Locator.**getSystemId**()
返回当前事件的系统标识符。

20.12.4 InputSource 对象

InputSource.**setPublicId**(*id*)
设置该*InputSource* 的公有标识符。

InputSource.**getPublicId**()
返回此*InputSource* 的公有标识符。

InputSource.**setSystemId**(*id*)
设置此*InputSource* 的系统标识符。

InputSource.**getSystemId**()
返回此*InputSource* 的系统标识符。

InputSource.**setEncoding**(*encoding*)
设置此*InputSource* 的字符编码格式。

编码格式必须是 XML 编码声明可接受的字符串（参见 XML 建议规范第 4.3.3 节）。

如果*InputSource* 还包含一个字符流则*InputSource* 的 encoding 属性会被忽略。

InputSource.**getEncoding**()
获取此 InputSource 的字符编码格式。

InputSource.**setByteStream**(*bytefile*)
设置此输入源的字节流（为*binary file* 对象）。

如果还指定了一个字符流被则 SAX 解析器会忽略此设置，但它将优先使用字节流而不是自己打开一个 URI 连接。

如果应用程序知道字节流的字符编码格式，它应当使用 setEncoding 方法来设置它。

InputSource.**getByteStream**()
获取此输入源的字节流。

getEncoding 方法将返回该字节流的字符编码格式，如果未知则返回 None。

InputSource.**setCharacterStream**(*charfile*)
设置此输入源的字符流 (为*text file* 对象)。

如果指定了一个字符流，SAX 解析器将忽略任何字节流并且不会尝试打开一个指向系统标识符的 URI 连接。

InputSource.**getCharacterStream**()
获取此输入源的字符流。

20.12.5 Attributes 接口

Attributes 对象实现了一部分映射协议，包括 copy(), get(), __contains__(), items(), keys() 和 values() 等方法。还提供了下列方法:

Attributes.**getLength**()
返回属性的数量。

Attributes.**getNames**()
返回属性的名称。

Attributes.**getType**(*name*)
返回属性 *name* 的类型，通常为 'CDATA'。

Attributes.**getValue**(*name*)
返回属性 *name* 的值。

20.12.6 AttributesNS 接口

此接口是 Attributes 接口（参见*Attributes 接口* 章节）的一个子类型。那个接口所支持的所有方法在 AttributesNS 对象上也都可用。

下列方法也是可用的:

AttributesNS.**getValueByQName**(*name*)
返回一个限定名称的值。

AttributesNS.**getNameByQName**(*name*)
返回限定名称 *name* 的 (namespace, localname) 对。

AttributesNS.**getQNameByName**(*name*)
返回 (namespace, localname) 对的限定名称。

AttributesNS.**getQNames**()
返回所有属性的限定名称。

20.13 xml.parsers.expat —使用 Expat 的快速 XML 解析

警告： pyexpat 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

xml.parsers.expat 模块是针对 Expat 非验证 XML 解析器的 Python 接口。此模块提供了一个扩展类型 xmlparser，它代表一个 XML 解析器的当前状态。在创建一个 xmlparser 对象之后，该对象的各个属性可被设置为相应的处理句柄函数。随后当将一个 XML 文档送入解析器时，就会为该 XML 文档中的字符数据和标记调用处理句柄函数。

此模块使用 pyexpat 模块来提供对 Expat 解析器的访问。直接使用 pyexpat 模块的方式已被弃用。

此模块提供了一个异常和一个类型对象:

exception xml.parsers.expat.**ExpatError**
此异常会在 Expat 报错时被引发。请参阅*ExpatError 异常* 一节了解有关解读 Expat 错误的更多信息。

exception xml.parsers.expat.**error**
ExpatError 的别名。

xml.parsers.expat.**XMLParserType**
来自*ParserCreate()* 函数的返回值的类型。

xml.parsers.expat 模块包含两个函数:

xml.parsers.expat.**ErrorString**(*errno*)
返回给定错误号 *errno* 的解释性字符串。

xml.parsers.expat.**ParserCreate**(*encoding=None*, *namespace_separator=None*)
创建并返回一个新的 xmlparser 对象。如果指定了 *encoding*，它必须为指定 XML 数据所使用的编码格式名称的字符串。Expat 支持的编码格式没有 Python 那样多，而且它的编码格式库也不能被扩展；它支持 UTF-8, UTF-16, ISO-8859-1 (Latin1) 和 ASCII。如果给出了 *encoding*[1] 则它将覆盖隐式或显式指定的文档编码格式。

可以选择让 Expat 为你做 XML 命名空间处理，这是通过提供 *namespace_separator* 值来启用的。该值必须是一个单字符的字符串；如果字符串的长度不合法则将引发*ValueError* (None 被视为等同于省略)。当命名空间处理被启用时，属于特定命名空间的元素类型名称和属性名称将被展开。传递给 The element name passed to the 元素处理句柄 StartElementHandler 和 EndElementHandler 的元素名称将为命名空间 URI，命名空间分隔符和名称的本地部分的拼接。如果命名空间分隔符是一个零字节 (chr(0)) 则命名空间 URI 和本地部分将被直接拼接而不带任何分隔符。

举例来说，如果 *namespace_separator* 被设为空格符 (' ') 并对以下文档进行解析:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

StartElementHandler 将为每个元素获取以下字符串:

[1] 包括在 XML 输出中的编码格式字符串应当符合适当的标准。例如 “UTF-8” 是有效的，但 “UTF8” 是无效的。请参阅 https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 和 https://www.iana.org/assignments/character-sets/character-sets.xhtml。

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

由于 pyexpat 所使用的 Expat 库的限制，被返回的 xmlparser 实例只能被用来解析单个 XML 文档。请为每个文档调用 ParserCreate 来提供单独的解析器实例。

参见:

The Expat XML Parser Expat 项目的主页。

20.13.1 XMLParser 对象

xmlparser 对象具有以下方法:

xmlparser.**Parse**(*data*[, *isfinal*])
解析字符串 *data* 的内容，调用适当的处理函数来处理解析后的数据。在对此方法的最后一次调用时 *isfinal* 必须为真值；它允许以片段形式解析单个文件，而不是提交多个文件。*data* 在任何时候都可以为空字符串。

xmlparser.**ParseFile**(*file*)
解析从对象 *file* 读取的 XML 数据。*file* 仅需提供 read(nbytes) 方法，当没有更多数据可读时将返回空字符串。

xmlparser.**SetBase**(*base*)
设置要用于解析声明中的系统标识符的相对 URI 的基准。解析相对标识符的任务会留给应用程序进行：这个值将作为 *base* 参数传递给*ExternalEntityRefHandler()*, *NotationDeclHandler()* 和*UnparsedEntityDeclHandler()* 函数。

xmlparser.**GetBase**()
返回包含之前调用*SetBase()* 所设置的基准位置的字符串，或者如果未调用*SetBase()* 则返回 None。

xmlparser.**GetInputContext**()
将生成当前事件的输入数据以字符串形式返回。数据为包含文本的实体的编码格式。如果被调用时未激活事件处理句柄，则返回值将为 None。

xmlparser.**ExternalEntityParserCreate**(*context*[, *encoding*])
创建一个“子”解析器，可被用来解析由父解析器解析的内容所引用的外部解析实体。*context* 形参应当是传递给*ExternalEntityRefHandler()* 处理函数的字符串，具体如下所述。子解析器创建时*ordered_attributes* 和*specified_attributes* 会被设为此解析器的值。

xmlparser.**SetParamEntityParsing**(*flag*)
控制参数实体（包括外部 DTD 子集）的解析。可能的 *flag* 值有 XML_PARAM_ENTITY_PARSING_NEVER, XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE 和 XML_PARAM_ENTITY_PARSING_ALWAYS。如果该旗标设置成功则返回真值。

xmlparser.**UseForeignDTD**([*flag*])
调用时将 *flag* 设为真值（默认）将导致 Expat 调用*ExternalEntityRefHandler* 时将所有参数设为*None* 以允许加载替代的 DTD。如果文档不包含文档类型声明，*ExternalEntityRefHandler* 仍然会被调用，但*StartDoctypeDeclHandler* 和*EndDoctypeDeclHandler* 将不会被调用。

为 *flag* 传入假值将将撤消之前传入真值的调用，除此之外没有其他影响。

此方法只能在调用*Parse()* 或*ParseFile()* 方法之前被调用；在已调用过这两个方法之后调用它会导致引发*ExpatError* 且*code* 属性被设为 errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]。

xmlparser 对象具有下列属性:

xmlparser.**buffer_size**
当*buffer_text*为真值时所使用的缓冲区大小。可以通过将此属性赋一个新的整数值来设置一个新的缓冲区大小。当大小发生改变时，缓冲区将被刷新。

xmlparser.**buffer_text**
将此属性设为真值会使得 xmlparser 对象缓冲 Expat 所返回的文本内容以尽可能地避免多次调用*CharacterDataHandler()*回调。这可以显著地提升性能，因为 Expat 通常会将字符数据在每个行结束的位置上进行分块。此属性默认为假值，但可以在任何时候被更改。

xmlparser.**buffer_used**
当*buffer_text*被启用时，缓冲区中存储的字节数。这些字节数据表示以 UTF-8 编码的文本。当*buffer_text*为假值时此属性没有任何实际意义。

xmlparser.**ordered_attributes**
将该属性设为非零整数会使得各个属性被报告为列表而非字典。各个属性会按照在文档文本中的出现顺序显示。对于每个属性，将显示两个列表条目：属性名和属性值。（该模块的较旧版本也使用了此格式。）默认情况下，该属性为假值；它可以在任何时候被更改。

xmlparser.**specified_attributes**
如果设为非零整数，解析器将只报告在文档实例中指明的属性而不报告来自属性声明的属性。设置此属性的应用程序需要特别小心地使用从声明中获得的附加信息以符合 XML 处理程序的行为标准。默认情况下，该属性为假值；它可以在任何时候被更改。

下列属性包含与 xmlparser 对象遇到的最近发生的错误有关联的值，并且一旦对 Parse() 或 ParseFile() 的调用引发了*xml.parsers.expat.ExpatError*异常就将只包含正确的值。

xmlparser.**ErrorByteIndex**
错误发生位置的字节索引号。

xmlparser.**ErrorCode**
指明问题的的数字代码。该值可被传给*ErrorString()*函数，或是与在 errors 对象中定义的常量之一进行比较。

xmlparser.**ErrorColumnNumber**
错误发生位置的列号。

xmlparser.**ErrorLineNumber**
错误发生位置的行号。

下列属性包含 xmlparser 对象中关联到当前解析位置的值。在回调报告解析事件期间它们将指示生成事件的字符序列的第一个字符的位置。当在回调的外部被调用时，所指示的位置将恰好位于最后的解析事件之后（无论是否存在关联的回调）。

xmlparser.**CurrentByteIndex**
解析器输入的当前字节索引号。

xmlparser.**CurrentColumnNumber**
解析器输入的当前列号。

xmlparser.**CurrentLineNumber**
解析器输入的当前行号。

可被设置的处理句柄列表。要在一个 xmlparser 对象 *o* 上设置处理句柄，请使用 o.handlername = func。*handlername* 必须从下面的列表中获取，而 *func* 必须为接受正确数量参数的可调用对象。所有参数均为字符串，除非另外指明。

xmlparser.**XmlDeclHandler**(*version*, *encoding*, *standalone*)
当解析 XML 声明时被调用。XML 声明是 XML 建议适用版本、文档文本的编码格式，以及可选的“独立”声明的（可选）声明。*version* 和 *encoding* 将为字符串，而 *standalone* 在文档被声明为独立时将为 1，在文档被声明为非独立时将为 0，或者在 standalone 短语被省略时则为 -1。这仅适用于 Expat 的 1.95.0 或更新版本。

xmlparser.**StartDoctypeDeclHandler**(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)
当 Expat 开始解析文档类型声明 (<!DOCTYPE ...) 时被调用。*doctypeName* 会完全按所显示的被提供。*systemId* 和 *publicId* 形参给出所指定的系统和公有标识符，如果被省略则为 None。如果文档包含内部文档声明子集则 *has_internal_subset* 将为真值。这要求 Expat 1.2 或更新的版本。

xmlparser.**EndDoctypeDeclHandler**()
当 Expat 完成解析文档类型声明时被调用。这要求 Expat 1.2 或更新版本。

xmlparser.**ElementDeclHandler**(*name*, *model*)
为每个元素类型声明调用一次。*name* 为元素类型名称，而 *model* 为内容模型的表示形式。

xmlparser.**AttlistDeclHandler**(*elname*, *attname*, *type*, *default*, *required*)
为一个元素类型的每个已声明属性执行调用。如果一个属性列表声明声明了三个属性，这个处理句柄会被调用三次，每个属性一次。*elname* 是声明所适用的元素的名称而 *attname* 是已声明的属性的名称。属性类型是作为 *type* 传入的字符串；可能的值有 'CDATA', 'ID', 'IDREF', ... *default* 给出了当属性未被文档实例所指明时该属性的默认值，或是为 None，如果没有默认值 (#IMPLIED 值) 的话。如果属性必须在文档实例中给出，则 *required* 将为真值。这要求 Expat 1.95.0 或更新的版本。

xmlparser.**StartElementHandler**(*name*, *attributes*)
在每个元素开始时被调用。*name* 是包含元素名称的字符串，而 *attributes* 是元素的属性。如果*ordered_attributes* 为真值，则属性为列表形式 (完整描述参见*ordered_attributes*)。否则为将名称映射到值的字典。

xmlparser.**EndElementHandler**(*name*)
在每个元素结束时被调用。

xmlparser.**ProcessingInstructionHandler**(*target*, *data*)
在每次处理指令时调用。

xmlparser.**CharacterDataHandler**(*data*)
针对字符数据调用。此方法将被用于普通字符数据、CDATA 标记内容以及可忽略的空白符。需要区分这几种情况的应用程序可以使用*StartCdataSectionHandler*, *EndCdataSectionHandler* 和*ElementDeclHandler* 回调来收集必要的信息。

xmlparser.**UnparsedEntityDeclHandler**(*entityName*, *base*, *systemId*, *publicId*, *notationName*)
针对未解析（NDATA）实体声明调用。此方法仅存在于 Expat 库的 1. 2 版；对于更新的版本，请改用*EntityDeclHandler*。(下层 Expat 库中的对应函数已被声明为过时。)

xmlparser.**EntityDeclHandler**(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)
针对所有实体声明被调用。对于形参和内部实体，*value* 将为给出实体的声明内容的字符串；对于外部实体将为 None。*notationName* 形参对于已解析实体将为 None，对于未解析实体则为标注的名称。如果实体为形参实体则 *is_parameter_entity* 将为真值而如果为普通实体则为假值（大多数应用程序只需要关注普通实体）。此方法仅从 1.95.0 版 Expat 库开始才可用。

xmlparser.**NotationDeclHandler**(*notationName*, *base*, *systemId*, *publicId*)
针对标注声明被调用。*notationName*, *base*, *systemId* 和 *publicId* 如果给出则均应为字符串。如果省略公有标识符，则 *publicId* 将为 None。

xmlparser.**StartNamespaceDeclHandler**(*prefix*, *uri*)
当一个元素包含命名空间声明时被调用。命名空间声明会在为声明所在的元素调用*StartElementHandler* 之前被处理。

xmlparser.**EndNamespaceDeclHandler**(*prefix*)
当到达包含命名空间声明的元素的关闭标记时被调用。此方法会按照调用*StartNamespaceDeclHandler* 以指明每个命名空间作用域的开始的逆顺序为元素上的每个命名空间声明调用一次。对这个处理句柄的调用是在相应的*EndElementHandler* 之后针对元素的结束而进行的。

xmlparser.**CommentHandler**(*data*)
针对注释被调用。*data* 是注释的文本，不包括开头的 '<!--' 和末尾的 '-->'。

xmlparser.**StartCdataSectionHandler**()
在一个 CDATA 节的开头被调用。需要此方法和*EndCdataSectionHandler* 以便能够标识 CDATA 节的语法开始和结束。

xmlparser.**EndCdataSectionHandler**()
在一个 CDATA 节的末尾被调用。

xmlparser.**DefaultHandler**(*data*)
针对 XML 文档中没有指定适用处理句柄的任何字符被调用。这包括了所有属于可被报告的结构的一部分，但未提供处理句柄的字符。

xmlparser.**DefaultHandlerExpand**(*data*)
这与*DefaultHandler()* 相同，但不会抑制内部实体的扩展。实体引用将不会被传递给默认处理句柄。

xmlparser.**NotStandaloneHandler**()
当 XML 文档未被声明为独立文档时被调用。这种情况发生在出现外部子集或对参数实体的引用，但 XML 声明没有在 XML 声明中将 standalone 设为 `yes` 的时候。如果这个处理句柄返回 `0`，那么解析器将引发 `XML_ERROR_NOT_STANDALONE` 错误。如果这个处理句柄没有被设置，那么解析器就不会为这个条件引发任何异常。

xmlparser.**ExternalEntityRefHandler**(*context*, *base*, *systemId*, *publicId*)
为对外部实体的引用执行调用。*base* 为当前的基准，由之前对*SetBase()* 的调用设置。公有和系统标识符 *systemId* 和 *publicId* 如果给出则🄴为字符串；如果公有标识符未给出，则 *publicId* 将为 `None`。*context* 是仅根据以下说明来使用的不透明值。

对于要解析的外部实体，这个处理句柄必须被实现。它负责使用 `ExternalEntityParserCreate(context)` 来创建子解析器，通过适当的回调将其初始化，并对实体进行解析。这个处理句柄应当返回一个整数；如果它返回 `0`，则解析器将引发 `XML_ERROR_EXTERNAL_ENTITY_HANDLING` 错误，否则解析将会继续。

如果未提供这个处理句柄，外部实体会由*DefaultHandler* 回调来报告，如果提供了该回调的话。

20.13.2 ExpatError 异常

ExpatError 异常包含几个有趣的属性:

ExpatError.**code**
Expat 对于指定错误的内部错误号。*errors.messages* 字典会将这些错误号映射到 Expat 的错误消息。例如:

```
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

errors 模块也提供了一些错误消息常量和一个将这些消息映射回错误码的字典*codes*，参见下文。

ExpatError.**lineno**
检测到错误所在的行号。首行的行号为 `1`。

ExpatError.**offset**
错误发生在行中的字符偏移量。首列的列号为 `0`。

20.13.3 示例

以下程序定义了三个处理句柄，会简单地打印出它们的参数。:

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

来自这个程序的输出是:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

20.13.4 内容模型描述

内容模型是使用嵌套的元组来描述的。每个元素包含四个值：类型、限定符、名称和一个子元组。子元组就是附加的内容模型描述。

前两个字段的值是在*xml.parsers.expat.model* 模块中定义的常量。这些常量可分为两组：模型类型组和限定符组。

模型类型组中的常量有:

xml.parsers.expat.model.**XML_CTYPE_ANY**
: 模型名称所指定的元素被声明为具有 ANY 内容模型。

xml.parsers.expat.model.**XML_CTYPE_CHOICE**
: 命名元素允许从几个选项中选择；这被用于内容模型例如 (A | B | C)。

xml.parsers.expat.model.**XML_CTYPE_EMPTY**
: 被声明为 EMPTY 的元素具有此模型类型。

xml.parsers.expat.model.**XML_CTYPE_MIXED**

xml.parsers.expat.model.**XML_CTYPE_NAME**

xml.parsers.expat.model.**XML_CTYPE_SEQ**
代表彼此相连的一系列模型的模型用此模型类型来指明。这被用于 (A, B, C) 这样的模型。

限定符组中的常量有:

xml.parsers.expat.model.**XML_CQUANT_NONE**
未给出限定符，这样它可以只出现一次，例如 A。

xml.parsers.expat.model.**XML_CQUANT_OPT**
可选的模型：它可以出现一次或完全不出现，例如 A?。

xml.parsers.expat.model.**XML_CQUANT_PLUS**
模型必须出现一次或多次 (例如 A+)。

xml.parsers.expat.model.**XML_CQUANT_REP**
模型必须出现零次或多次，例如 A*。

20.13.5 Expat 错误常量

下列常量是在*xml.parsers.expat.errors* 模块中提供的。这些常量在有错误发生时解读被引发的 ExpatError 异常对象的某些属性时很有用处。出于保持向下兼容性的理由，这些常量的值是错误 *消息* 而不是数字形式的错误 *代码*，为此你可以将它的*code* 属性和 errors.codes[errors.XML_ERROR_*CONSTANT_NAME*] 进行比较。

errors 模块具有以下属性:

xml.parsers.expat.errors.**codes**
将数字形式的错误代码映射到其字符串描述的字典。

3.2 新版功能.

xml.parsers.expat.errors.**messages**
将字符串描述映射到其错误代码的字典。

3.2 新版功能.

xml.parsers.expat.errors.**XML_ERROR_ASYNC_ENTITY**

xml.parsers.expat.errors.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**
属性值中指向一个外部实体而非内部实体的实体引用。

xml.parsers.expat.errors.**XML_ERROR_BAD_CHAR_REF**
指向一个在 XML 不合法的字符的字符引用 (例如，字符 0 或 '&#0;')。

xml.parsers.expat.errors.**XML_ERROR_BINARY_ENTITY_REF**
指向一个使用标注声明，因而无法被解析的实体的实体引用。

xml.parsers.expat.errors.**XML_ERROR_DUPLICATE_ATTRIBUTE**
一个属性在一个开始标记中被使用超过一次。

xml.parsers.expat.errors.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.errors.**XML_ERROR_INVALID_TOKEN**
当一个输入字节无法被正确分配给一个字符时引发；例如，在 UTF-8 输入流中的 NUL 字节 (值为 0).

xml.parsers.expat.errors.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**
在文档元素之后出现空白符以外的内容。

xml.parsers.expat.errors.**XML_ERROR_MISPLACED_XML_PI**
在输入数据开始位置以外的地方发现 XML 声明。

xml.parsers.expat.errors.**XML_ERROR_NO_ELEMENTS**
文档不包含任何元素（XML 要求所有文档都包含恰好一个最高层级元素）。

xml.parsers.expat.errors.**XML_ERROR_NO_MEMORY**
Expat 无法在内部分配内存。

xml.parsers.expat.errors.**XML_ERROR_PARAM_ENTITY_REF**
在不被允许的位置发现一个参数实体引用。

xml.parsers.expat.errors.**XML_ERROR_PARTIAL_CHAR**
在输入中发出一个不完整的字符。

xml.parsers.expat.errors.**XML_ERROR_RECURSIVE_ENTITY_REF**
一个实体引用包含了对同一实体的另一个引用；可能是通过不同的名称，并可能是间接的引用。

xml.parsers.expat.errors.**XML_ERROR_SYNTAX**
遇到了某个未指明的语法错误。

xml.parsers.expat.errors.**XML_ERROR_TAG_MISMATCH**
一个结束标记不能匹配到最内层的未关闭开始标记。

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_TOKEN**
某些记号（例如开始标记）在流结束或遇到下一个记号之前还未关闭。

xml.parsers.expat.errors.**XML_ERROR_UNDEFINED_ENTITY**
对一个未定义的实体进行了引用。

xml.parsers.expat.errors.**XML_ERROR_UNKNOWN_ENCODING**
文档编码格式不被 Expat 所支持。

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_CDATA_SECTION**
一个 CDATA 标记节还未关闭。

xml.parsers.expat.errors.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.errors.**XML_ERROR_NOT_STANDALONE**
解析器确定文档不是“独立的”但它却在 XML 声明中声明自己是独立的，并且 `NotStandaloneHandler` 被设置为返回 `0`。

xml.parsers.expat.errors.**XML_ERROR_UNEXPECTED_STATE**

xml.parsers.expat.errors.**XML_ERROR_ENTITY_DECLARED_IN_PE**

xml.parsers.expat.errors.**XML_ERROR_FEATURE_REQUIRES_XML_DTD**
请求了一个需要已编译 DTD 支持的操作，但 Expat 被配置为不带 DTD 支持。此错误应当绝对不会被*xml.parsers.expat* 模块的标准构建版本所报告。

xml.parsers.expat.errors.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**
在解析开始之后请求一个只能在解析开始之前执行的行为改变。此错误（目前）只能由 `UseForeignDTD()` 所引发。

xml.parsers.expat.errors.**XML_ERROR_UNBOUND_PREFIX**
当命名空间处理被启用时发现一个未声明的前缀。

xml.parsers.expat.errors.**XML_ERROR_UNDECLARING_PREFIX**
文档试图移除与某个前缀相关联的命名空间声明。

xml.parsers.expat.errors.**XML_ERROR_INCOMPLETE_PE**
一个参数实体包含不完整的标记。

xml.parsers.expat.errors.**XML_ERROR_XML_DECL**
文档完全未包含任何文档元素。

xml.parsers.expat.errors.**XML_ERROR_TEXT_DECL**
解析一个外部实体中的文本声明时出现错误。

xml.parsers.expat.errors.**XML_ERROR_PUBLICID**
在公有 id 中发现不被允许的字符。

xml.parsers.expat.errors.**XML_ERROR_SUSPENDED**
在挂起的解析器上请求执行操作，但未获得允许。这包括提供额外输入或停止解析器的尝试。

xml.parsers.expat.errors.**XML_ERROR_NOT_SUSPENDED**
在解析器未被挂起的时候执行恢复解析器的尝试。

xml.parsers.expat.errors.**XML_ERROR_ABORTED**
此错误不应当被报告给 Python 应用程序。

xml.parsers.expat.errors.**XML_ERROR_FINISHED**
在一个已经完成解析输入的解析器上请求执行操作，但未获得允许。这包括提供额外输入或停止解析器的尝试。

xml.parsers.expat.errors.**XML_ERROR_SUSPEND_PE**

备注

CHAPTER 21

互联网协议和支持

本章介绍的模块实现了互联网协议并支持相关技术。它们都是用 Python 实现的。这些模块中的大多数都需要存在依赖于系统的模块*socket*，目前大多数流行平台都支持它。这是一个概述：

21.1 webbrowser —方便的 Web 浏览器控制器

源码： Lib/webbrowser.py

webbrowser 模块提供了一个高级接口，允许向用户显示基于 Web 的文档。在大多数情况下，只需从该模块调用*open()* 函数就可以了。

在 Unix 下，图形浏览器在 X11 下是首选，但如果图形浏览器不可用或 X11 显示不可用，则将使用文本模式浏览器。如果使用文本模式浏览器，则调用进程将阻塞，直到用户退出浏览器。

如果存在环境变量 BROWSER，则将其解释为*os.pathsep* 分隔的浏览器列表，以便在平台默认值之前尝试。当列表部分的值包含字符串 %s 时，它被解释为一个文字浏览器命令行，用于替换 %s 的参数 URL；如果该部分不包含 %s，则它只被解释为要启动的浏览器的名称。[1]

对于非 Unix 平台，或者当 Unix 上有远程浏览器时，控制过程不会等待用户完成浏览器，而是允许远程浏览器在显示界面上维护自己的窗口。如果 Unix 上没有远程浏览器，控制进程将启动一个新的浏览器并等待。

脚本 **webbrowser** 可以用作模块的命令行界面。它接受一个 URL 作为参数。还接受以下可选参数：-n 如果可能，在新的浏览器窗口中打开 URL；-t 在新的浏览器页面（“标签”）中打开 URL。这些选择当然是相互排斥的。用法示例：

```
python -m webbrowser -t "http://www.python.org"
```

定义了以下异常：

exception webbrowser.**Error**
发生浏览器控件错误时引发异常。

[1] 这里命名的不带完整路径的可执行文件将在 PATH 环境变量给出的目录中搜索。

定义了以下函数：

webbrowser.**open**(*url*, *new=0*, *autoraise=True*)
使用默认浏览器显示 *url*。如果 *new* 为 0，则尽可能在同一浏览器窗口中打开 *url*。如果 *new* 为 1，则尽可能打开新的浏览器窗口。如果 *new* 为 2，则尽可能打开新的浏览器页面（“标签”）。如果 *autoraise* 为 “True”，则会尽可能置前窗口（请注意，在许多窗口管理器下，无论此变量的设置如何，都会置前窗口）。

请注意，在某些平台上，尝试使用此函数打开文件名，可能会起作用并启动操作系统的关联程序。但是，这种方式不被支持也不可移植。

webbrowser.**open_new**(*url*)
如果可能，在默认浏览器的新窗口中打开 *url*，否则，在唯一的浏览器窗口中打开 *url*。

webbrowser.**open_new_tab**(*url*)
如果可能，在默认浏览器的新页面（“标签”）中打开 *url*，否则等效于*open_new()*。

webbrowser.**get**(*using=None*)
返回浏览器类型为 *using* 指定的控制器对象。如果 *using* 为 None，则返回适用于调用者环境的默认浏览器的控制器。

webbrowser.**register**(*name*, *constructor*, *instance=None*)
注册 *name* 浏览器类型。注册浏览器类型后，*get()* 函数可以返回该浏览器类型的控制器。如果没有提供 *instance*，或者为 None，*constructor* 将在没有参数的情况下被调用，以在需要时创建实例。如果提供了 *instance*，则永远不会调用 *constructor*，并且可能是 None。

This entry point is only useful if you plan to either set the BROWSER variable or call *get()* with a nonempty argument matching the name of a handler you declare.

预定义了许多浏览器类型。此表给出了可以传递给*get()* 函数的类型名称以及控制器类的相应实例化，这些都在此模块中定义。

类型名	类名	注释
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'netscape'	Mozilla('netscape')	
'galeon'	Galeon('galeon')	
'epiphany'	Galeon('epiphany')	
'skipstone'	BackgroundBrowser('skipstone')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'mosaic'	BackgroundBrowser('mosaic')	
'opera'	Opera()	
'grail'	Grail()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSX('default')	(3)
'safari'	MacOSX('safari')	(3)
'google-chrome'	Chrome('google-chrome')	
'chrome'	Chrome('chrome')	
'chromium'	Chromium('chromium')	
'chromium-browser'	Chromium('chromium-browser')	

注释:

(1) “Konqueror” 是 Unix 的 KDE 桌面环境的文件管理器，只有在 KDE 运行时才有意义。一些可靠地检测 KDE 的方法会很好；仅检查 KDEDIR 变量是不够的。另请注意，KDE 2 的 **konqueror** 命令，会使用名称 “kfm” —此实现选择运行的 Konqueror 的最佳策略。

(2) 仅限 Windows 平台。

(3) 仅限 Mac OS X 平台。

3.3 新版功能: 添加了对 Chrome/Chromium 的支持。

以下是一些简单的例子:

```
url = 'http://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

21.1.1 浏览器控制器对象

浏览器控制器提供三个与模块级便捷函数相同的方法:

controller.**open**(*url*, *new=0*, *autoraise=True*)
使用此控制器处理的浏览器显示 *url*。如果 *new* 为 1，则尽可能打开新的浏览器窗口。如果 *new* 为 2，则尽可能打开新的浏览器页面（“标签”）。

controller.**open_new**(*url*)
如果可能，在此控制器处理的浏览器的新窗口中打开 *url* ，否则，在唯一的浏览器窗口中打开 *url* 。别名*open_new()*。

controller.**open_new_tab**(*url*)
如果可能，在此控制器处理的浏览器的新页面（“标签”）中打开 *url*，否则等效于*open_new()*。

备注

21.2 cgi —通用网关接口支持

源代码: Lib/cgi.py

通用网关接口 (CGI) 脚本的支持模块

本模块定义了一些工具供以 Python 编写的 CGI 脚本使用。

21.2.1 概述

CGI 脚本是由 HTTP 服务器发起调用，通常用来处理通过 HTML <FORM> 或 <ISINDEX> 元素提交的用户输入。

在大多数情况下，CGI 脚本存放在服务器的 cgi-bin 特殊目录下。HTTP 服务器将有关请求的各种信息（例如客户端的主机名、所请求的 URL、查询字符串以及许多其他内容）放在脚本的 shell 环境中，然后执行脚本，并将脚本的输出发回到客户端。

脚本的输入也会被连接到客户端，并且有时表单数据也会以此方式来读取；在其他时候表单数据会通过 URL 的“查询字符串”部分来传递。本模块的目标是处理不同的应用场景并向 Python 脚本提供一个更为简单的接口。它还提供了一些工具为脚本调试提供帮助，而最近增加的还有对通过表单上传文件的支持（如果你的浏览器支持该功能的话）。

CGI 脚本的输出应当由两部分组成，并由一个空行分隔。前一部分包含一些标头，它们告诉客户端后面会提供何种数据。生成一个最小化标头部分的 Python 代码如下所示:

```
print("Content-Type: text/html")    # HTML is following
print()                             # blank line, end of headers
```

后一部分通常为 HTML，提供给客户端软件来显示格式良好包含标题的文本、内联图片等内容。下面是打印一段简单 HTML 的 Python 代码:

```
print("<TITLE>CGI script output</TITLE>")
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")
```

21.2.2 使用 cgi 模块。

通过敲下 import cgi 来开始。

当你在写一个新脚本时，考虑加上这些语句:

```
import cgitb
cgitb.enable()
```

This activates a special exception handler that will display detailed reports in the Web browser if any errors occur. If you'd rather not show the guts of your program to users of your script, you can have the reports saved to files instead, with code like this:

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

在脚本开发期间使用此特性会很有帮助。*cgitb* 所产生的报告提供了在追踪程序问题时能为你节省大量时间的信息。你可以在完成测试你的脚本并确信它能正确工作之后再移除 cgitb 行。

要获取提交的表单数据，请使用 FieldStorage 类。如果表单包含非 ASCII 字符，请使用 *encoding* 关键字参数并设置为文档所定义的编码格式值。它通常包含在 HTML 文档的 HEAD 部分的 META 标签中或者由 *Content-Type* 标头所指明。这会从标准输入或环境读取表单内容（取决于根据 CGI 标准设置的多个环境变量的值）。由于它可能会消耗标准输入，它应当只被实例化一次。

FieldStorage 实例可以像 Python 字典一样来检索。它允许通过 in 运算符进行成员检测，也支持标准字典方法*keys()* 和内置函数*len()*。包含空字符串的表单字段会被忽略而不会出现在字典中；要保留这样的值，请在创建 FieldStorage 实例时为可选的 *keep_blank_values* 关键字形参提供一个真值。

举例来说，下面的代码（假定 *Content-Type* 标头和空行已经被打印）会检查字段 name 和 addr 是否均被设为非空字符串:

```
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print("<H1>Error</H1>")
    print("Please fill in the name and addr fields.")
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
...further form processing here...
```

在这里的字段通过 form[key] 来访问，它们本身就是 FieldStorage (或 MiniFieldStorage，取决于表单的编码格式) 的实例。实例的 value 属性会产生字段的字符串值。getvalue() 方法直接返回这个字符串；它还接受可选的第二个参数作为当请求的键不存在时要返回的默认值。

如果提交的表单数据包含一个以上的同名字段，由 form[key] 所提取的对象将不是一个 FieldStorage 或 MiniFieldStorage 实例而是由这种实例组成的列表。类似地，在这种情况下，form.getvalue(key) 将会返回一个字符串列表。如果你预计到这种可能性（当你的 HTML 表单包含多个同名字段时），请使用*getlist()* 方法，它总是返回一个值的列表（这样你就不需要对只有单个项的情况进行特别处理）。例如，这段代码拼接了任意数量的 username 字段，以逗号进行分隔:

```
value = form.getlist("username")
usernames = ",".join(value)
```

如果一个字段是代表上传的文件，请通过 value 属性访问该值或是通过 getvalue() 方法以字节形式将整个文件读入内存。这可能不是你想要的结果。你可以通过测试 filename 属性或 file 属性来检测上传的文件。然后你可以从 file 属性读取数据，直到它作为 FieldStorage 实例的垃圾回收的一部分被自动关闭 (*read()* 和*readline()* 方法将返回字节数据):

```
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

FieldStorage 对象还支持在 with 语句中使用，该语句结束时将自动关闭它们。

如果在获取上传文件的内容时遇到错误（例如，当用户点击回退或取消按钮中断表单提交时）该字段中对象的 done 属性值将被设为 -1。

文件上传标准草案考虑到了从一个字段上传多个文件的可能性（使用递归的 *multipart/** 编码格式）。当这种情况发生时，该条目将是一个类似字典的 FieldStorage 条目。这可以通过检测它的 type 属性来确定，该属性应当是 *multipart/form-data* (或者可能是匹配 *multipart/** 的其他 MIME 类型)。在这种情况下，它可以像最高层级的表单对象一样被递归地迭代处理。

当一个表单按“旧”格式提交时（即以查询字符串或是单个 *application/x-www-form-urlencoded* 类型的数据部分的形式），这些条目实际上将是 MiniFieldStorage 类的实例。在这种情况下，list, file 和 filename 属性将总是为 None。

通过 POST 方式提交并且也带有查询字符串的表单将同时包含 FieldStorage 和 MiniFieldStorage 条目。

在 3.4 版更改: file 属性会在创建 FieldStorage 实例的垃圾回收操作中被自动关闭。

在 3.5 版更改: 为 FieldStorage 类增加了上下文管理协议支持。

21.2.3 更高层级的接口

The previous section explains how to read CGI form data using the `FieldStorage` class. This section describes a higher level interface which was added to this class to allow one to do it in a more readable and intuitive way. The interface doesn't make the techniques described in previous sections obsolete —they are still useful to process file uploads efficiently, for example.

The interface consists of two simple methods. Using the methods you can process form data in a generic way, without the need to worry whether only one or more values were posted under one name.

In the previous section, you learned to write following code anytime you expected a user to post more than one value under one name:

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

This situation is common for example when a form contains a group of multiple checkboxes with the same name:

```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

In most situations, however, there's only one form control with a particular name in a form and then you expect and need only one value associated with this name. So you write a script containing for example this code:

```
user = form.getvalue("user").upper()
```

The problem with the code is that you should never expect that a client will provide valid input to your scripts. For example, if a curious user appends another `user=foo` pair to the query string, then the script would crash, because in this situation the `getvalue("user")` method call returns a list instead of a string. Calling the *upper()* method on a list is not valid (since lists do not have a method of this name) and results in an *AttributeError* exception.

Therefore, the appropriate way to read form data values was to always use the code which checks whether the obtained value is a single value or a list of values. That's annoying and leads to less readable scripts.

A more convenient approach is to use the methods *getfirst()* and *getlist()* provided by this higher level interface.

`FieldStorage.`**`getfirst`**`(`*name*, *default=None*`)`
This method always returns only one value associated with form field *name*. The method returns only the first value in case that more values were posted under such name. Please note that the order in which the values are received may vary from browser to browser and should not be counted on.[1] If no such form field or value exists then the method returns the value specified by the optional parameter *default*. This parameter defaults to `None` if not specified.

`FieldStorage.`**`getlist`**`(`*name*`)`
This method always returns a list of values associated with form field *name*. The method returns an empty list if no such form field or value exists for *name*. It returns a list consisting of one item if only one such value exists.

Using these methods you can write nice compact code:

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
```

(下页继续)

[1] Note that some recent versions of the HTML specification do state what order the field values should be supplied in, but knowing whether a request was received from a conforming browser, or even from a browser at all, is tedious and error-prone.

(续上页)

```
for item in form.getlist("item"):
    do_something(item)
```

21.2.4 函数

These are useful if you want more control, or if you want to employ some of the algorithms implemented in this module in other circumstances.

cgi.**parse**(*fp=None*, *environ=os.environ*, *keep_blank_values=False*, *strict_parsing=False*, *separator="&"*)
Parse a query in the environment or from a file (the file defaults to `sys.stdin`). The *keep_blank_values*, *strict_parsing* and *separator* parameters are passed to `urllib.parse.parse_qs()` unchanged.

cgi.**parse_qs**(*qs*, *keep_blank_values=False*, *strict_parsing=False*)
This function is deprecated in this module. Use `urllib.parse.parse_qs()` instead. It is maintained here only for backward compatibility.

cgi.**parse_qsl**(*qs*, *keep_blank_values=False*, *strict_parsing=False*)
This function is deprecated in this module. Use `urllib.parse.parse_qsl()` instead. It is maintained here only for backward compatibility.

cgi.**parse_multipart**(*fp*, *pdict*)
Parse input of type `multipart/form-data` (for file uploads). Arguments are *fp* for the input file and *pdict* for a dictionary containing other parameters in the `Content-Type` header.

Returns a dictionary just like `urllib.parse.parse_qs()` keys are the field names, each value is a list of values for that field. This is easy to use but not much good if you are expecting megabytes to be uploaded —in that case, use the `FieldStorage` class instead which is much more flexible.

Note that this does not parse nested multipart parts —use `FieldStorage` for that.

在 3.6.13 版更改: Added the *separator* parameter.

cgi.**parse_header**(*string*)
Parse a MIME header (such as `Content-Type`) into a main value and a dictionary of parameters.

cgi.**test**()
Robust test CGI script, usable as main program. Writes minimal HTTP headers and formats all information provided to the script in HTML form.

cgi.**print_environ**()
Format the shell environment in HTML.

cgi.**print_form**(*form*)
Format a form in HTML.

cgi.**print_directory**()
Format the current directory in HTML.

cgi.**print_environ_usage**()
Print a list of useful (used by CGI) environment variables in HTML.

cgi.**escape**(*s*, *quote=False*)
Convert the characters `'&'`, `'<'` and `'>'` in string *s* to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. If the optional flag *quote* is true, the quotation mark character (`"`) is also translated; this helps for inclusion in an HTML attribute value delimited by double quotes, as in `<a href="...">`. Note that single quotes are never translated.

3.2 版后已移除: This function is unsafe because *quote* is false by default, and therefore deprecated. Use `html.escape()` instead.

21.2.5 Caring about security

There's one important rule: if you invoke an external program (via the *os.system()* or *os.popen()* functions. or others with similar functionality), make very sure you don't pass arbitrary strings received from the client to the shell. This is a well-known security hole whereby clever hackers anywhere on the Web can exploit a gullible CGI script to invoke arbitrary shell commands. Even parts of the URL or field names cannot be trusted, since the request doesn't have to come from your form!

To be on the safe side, if you must pass a string gotten from a form to a shell command, you should make sure the string contains only alphanumeric characters, dashes, underscores, and periods.

21.2.6 Installing your CGI script on a Unix system

Read the documentation for your HTTP server and check with your local system administrator to find the directory where CGI scripts should be installed; usually this is in a directory `cgi-bin` in the server tree.

Make sure that your script is readable and executable by "others"; the Unix file mode should be `0o755` octal (use `chmod 0755 filename`). Make sure that the first line of the script contains `#!` starting in column 1 followed by the pathname of the Python interpreter, for instance:

```
#!/usr/local/bin/python
```

Make sure the Python interpreter exists and is executable by "others".

Make sure that any files your script needs to read or write are readable or writable, respectively, by "others"—their mode should be `0o644` for readable and `0o666` for writable. This is because, for security reasons, the HTTP server executes your script as user "nobody", without any special privileges. It can only read (write, execute) files that everybody can read (write, execute). The current directory at execution time is also different (it is usually the server's cgi-bin directory) and the set of environment variables is also different from what you get when you log in. In particular, don't count on the shell's search path for executables (`PATH`) or the Python module search path (`PYTHONPATH`) to be set to anything interesting.

If you need to load modules from a directory which is not on Python's default module search path, you can change the path in your script, before importing other modules. For example:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

(This way, the directory inserted last will be searched first!)

Instructions for non-Unix systems will vary; check your HTTP server's documentation (it will usually have a section on CGI scripts).

21.2.7 Testing your CGI script

Unfortunately, a CGI script will generally not run when you try it from the command line, and a script that works perfectly from the command line may fail mysteriously when run from the server. There's one reason why you should still test your script from the command line: if it contains a syntax error, the Python interpreter won't execute it at all, and the HTTP server will most likely send a cryptic error to the client.

Assuming your script has no syntax errors, yet it does not work, you have no choice but to read the next section.

21.2.8 Debugging CGI scripts

First of all, check for trivial installation errors —reading the section above on installing your CGI script carefully can save you a lot of time. If you wonder whether you have understood the installation procedure correctly, try installing a copy of this module file (`cgi.py`) as a CGI script. When invoked as a script, the file will dump its environment and the contents of the form in HTML form. Give it the right mode etc, and send it a request. If it's installed in the standard `cgi-bin` directory, it should be possible to send it a request by entering a URL into your browser of the form:

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

If this gives an error of type 404, the server cannot find the script –perhaps you need to install it in a different directory. If it gives another error, there's an installation problem that you should fix before trying to go any further. If you get a nicely formatted listing of the environment and form content (in this example, the fields should be listed as "addr" with value "At Home" and "name" with value "Joe Blow"), the `cgi.py` script has been installed correctly. If you follow the same procedure for your own script, you should now be able to debug it.

The next step could be to call the *cgi* module's *test()* function from your script: replace its main code with the single statement

```
cgi.test()
```

This should produce the same results as those gotten from installing the `cgi.py` file itself.

When an ordinary Python script raises an unhandled exception (for whatever reason: of a typo in a module name, a file that can't be opened, etc.), the Python interpreter prints a nice traceback and exits. While the Python interpreter will still do this when your CGI script raises an exception, most likely the traceback will end up in one of the HTTP server's log files, or be discarded altogether.

Fortunately, once you have managed to get your script to execute *some* code, you can easily send tracebacks to the Web browser using the *cgitb* module. If you haven't done so already, just add the lines:

```
import cgitb
cgitb.enable()
```

to the top of your script. Then try running it again; when a problem occurs, you should see a detailed report that will likely make apparent the cause of the crash.

If you suspect that there may be a problem in importing the *cgitb* module, you can use an even more robust approach (which only uses built-in modules):

```
import sys
sys.stderr = sys.stdout
print("Content-Type: text/plain")
print()
...your code here...
```

This relies on the Python interpreter to print the traceback. The content type of the output is set to plain text, which disables all HTML processing. If your script works, the raw HTML will be displayed by your client. If it raises an exception, most likely after the first two lines have been printed, a traceback will be displayed. Because no HTML interpretation is going on, the traceback will be readable.

21.2.9 Common problems and solutions

- Most HTTP servers buffer the output from CGI scripts until the script is completed. This means that it is not possible to display a progress report on the client's display while the script is running.
- Check the installation instructions above.
- Check the HTTP server's log files. (`tail -f logfile` in a separate window may be useful!)
- Always check a script for syntax errors first, by doing something like `python script.py`.
- If your script does not have any syntax errors, try adding `import cgitb; cgitb.enable()` to the top of the script.
- When invoking external programs, make sure they can be found. Usually, this means using absolute path names —`PATH` is usually not set to a very useful value in a CGI script.
- When reading or writing external files, make sure they can be read or written by the userid under which your CGI script will be running: this is typically the userid under which the web server is running, or some explicitly specified userid for a web server's `suexec` feature.
- Don't try to give a CGI script a set-uid mode. This doesn't work on most systems, and is a security liability as well.

备注

21.3 cgitb —用于 CGI 脚本的回溯管理器

源代码: Lib/cgitb.py

---

cgitb 模块提供了用于 Python 脚本的特殊异常处理程序。(这个名称有一点误导性。它最初是设计用来显示 HTML 格式的 CGI 脚本详细回溯信息。但后来被一般化为也可显示纯文本格式的回溯信息。) 激活这个模块之后，如果发生了未被捕获的异常，将会显示详细的已格式化的报告。报告显示内容包括每个层级的源代码摘录，还有当前正在运行的函数的参数和局部变量值，以帮助你调试问题。你也可以选择将此信息保存至文件而不是将其发送至浏览器。

要启用此特性，只需简单地将此代码添加到你的 CGI 脚本的最顶端:

```
import cgitb
cgitb.enable()
```

enable() 函数的选项可以控制是将报告显示在浏览器中，还是将报告记录到文件以供随后进行分析。

cgitb.**enable**(*display=1*, *logdir=None*, *context=5*, *format="html"*)
此函数可通过设置*sys.excepthook* 的值以使*cgitb* 模块接管解释器默认的异常处理机制。

可选参数 *display* 默认为 `1` 并可被设为 `0` 来停止将回溯发送至浏览器。如果给出了参数 *logdir*，则回溯会被写入文件。*logdir* 的值应当是一个用于存放所写入文件的目录。可选参数 *context* 是要在回溯中的当前源代码行前后显示的上下文行数；默认为 `5`。如果可选参数 *format* 为 `"html"`，输出将为 HTML 格式。任何其它值都会强制启用纯文本输出。默认取值为 `"html"`。

cgitb.**handler**(*info=None*)
此函数使用默认设置处理异常（即在浏览器中显示报告，但不记录到文件）。当你捕获了一个异常并希望使用*cgitb* 来报告它时可以使用此函数。可选的 *info* 参数应为一个包含异常类型，异常值和回溯对象的 3 元组，与*sys.exc_info()* 所返回的元组完全一致。如果未提供 *info* 参数，则会从*sys.exc_info()* 获取当前异常。

21.4 wsgiref —WSGI Utilities and Reference Implementation

The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

Only authors of web servers and programming frameworks need to know every detail and corner case of the WSGI design. You don't need to understand every detail of WSGI just to install a WSGI application or to write a web application using an existing framework.

wsgiref is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (**PEP 3333**).

See https://wsgi.readthedocs.org/ for more information about WSGI, and links to tutorials and other resources.

21.4.1 wsgiref.util –WSGI environment utilities

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in **PEP 3333**. All of the functions taking an *environ* parameter expect a WSGI-compliant dictionary to be supplied; please see **PEP 3333** for a detailed specification.

wsgiref.util.**guess_scheme**(*environ*)
: Return a guess for whether `wsgi.url_scheme` should be "http" or "https", by checking for a `HTTPS` environment variable in the *environ* dictionary. The return value is a string.

  This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a `HTTPS` variable with a value of "1" "yes", or "on" when a request is received via SSL. So, this function returns "https" if such a value is found, and "http" otherwise.

wsgiref.util.**request_uri**(*environ*, *include_query=True*)
: Return the full request URI, optionally including the query string, using the algorithm found in the "URL Reconstruction" section of **PEP 3333**. If *include_query* is false, the query string is not included in the resulting URI.

wsgiref.util.**application_uri**(*environ*)
: Similar to *request_uri()*, except that the `PATH_INFO` and `QUERY_STRING` variables are ignored. The result is the base URI of the application object addressed by the request.

wsgiref.util.**shift_path_info**(*environ*)
: Shift a single name from `PATH_INFO` to `SCRIPT_NAME` and return the name. The *environ* dictionary is *modified* in-place; use a copy if you need to keep the original `PATH_INFO` or `SCRIPT_NAME` intact.

  If there are no remaining path segments in `PATH_INFO`, `None` is returned.

  Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at `/foo`, and the request URI path is `/foo/bar/baz`, and the WSGI application at `/foo` calls *shift_path_info()*, it will receive the string "bar", and the environment will be updated to be suitable for passing to a WSGI application at `/foo/bar`. That is, `SCRIPT_NAME` will change from `/foo` to `/foo/bar`, and `PATH_INFO` will change from `/bar/baz` to `/baz`.

When PATH_INFO is just a "/", this routine returns an empty string and appends a trailing slash to SCRIPT_NAME, even though empty path segments are normally ignored, and SCRIPT_NAME doesn't normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in /x from ones ending in /x/ when using this routine to do object traversal.

wsgiref.util.**setup_testing_defaults**(*environ*)

Update *environ* with trivial defaults for testing purposes.

This routine adds various parameters required for WSGI, including HTTP_HOST, SERVER_NAME, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, PATH_INFO, and all of the **PEP 3333**-defined wsgi.* variables. It only supplies default values, and does not replace any existing settings for these variables.

This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

用法示例：

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

In addition to the environment functions above, the *wsgiref.util* module also provides these miscellaneous utilities:

wsgiref.util.**is_hop_by_hop**(*header_name*)

Return true if 'header_name' is an HTTP/1.1 "Hop-by-Hop" header, as defined by **RFC 2616**.

class wsgiref.util.**FileWrapper**(*filelike*, *blksize=8192*)

A wrapper to convert a file-like object to an *iterator*. The resulting objects support both __getitem__() and __iter__() iteration styles, for compatibility with Python 2.1 and Jython. As the object is iterated over, the optional *blksize* parameter will be repeatedly passed to the *filelike* object's read() method to obtain bytestrings to yield. When read() returns an empty bytestring, iteration is ended and is not resumable.

If *filelike* has a close() method, the returned object will also have a close() method, and it will invoke the *filelike* object's close() method when called.

用法示例：

```
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
```

(下页继续)

(续上页)

```
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```

21.4.2 wsgiref.headers –WSGI response header tools

This module provides a single class, *Headers*, for convenient manipulation of WSGI response headers using a mapping-like interface.

class wsgiref.headers.**Headers**([*headers*])

Create a mapping-like object wrapping *headers*, which must be a list of header name/value tuples as described in **PEP 3333**. The default value of *headers* is an empty list.

Headers objects support typical mapping operations including `__getitem__()`, `get()`, `__setitem__()`, `setdefault()`, `__delitem__()` and `__contains__()`. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers' existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, *Headers* objects do not raise an error when you try to get or delete a key that isn't in the wrapped header list. Getting a nonexistent header just returns `None`, and deleting a nonexistent header does nothing.

Headers objects also support `keys()`, `values()`, and `items()` methods. The lists returned by `keys()` and `items()` can include the same key more than once if there is a multi-valued header. The `len()` of a *Headers* object is the same as the length of its `items()`, which is the same as the length of the wrapped header list. In fact, the `items()` method just returns a copy of the wrapped header list.

Calling `bytes()` on a *Headers* object returns a formatted bytestring suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the bytestring is terminated with a blank line.

In addition to their mapping interface and formatting features, *Headers* objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

get_all(*name*)

Return a list of all the values for the named header.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

add_header(*name*, *value*, ***_params*)

Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

name is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or `None`. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form `name="value"`. If it is `None`, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

The above will add a header that looks like this:

```
Content-Disposition: attachment; filename="bud.gif"
```

在 3.5 版更改: *headers* parameter is optional.

21.4.3 wsgiref.simple_server –a simple WSGI HTTP server

This module implements a simple HTTP server (based on *http.server*) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses PATH_INFO to select which application to invoke for each request. (E.g., using the shift_path_info() function from *wsgiref.util*.)

wsgiref.simple_server.**make_server**(*host*, *port*, *app*, *server_class=WSGIServer*, *handler_class=WSGIRequestHandler*)

Create a new WSGI server listening on *host* and *port*, accepting connections for *app*. The return value is an instance of the supplied *server_class*, and will process requests using the specified *handler_class*. *app* must be a WSGI application object, as defined by **PEP 3333**.

用法示例:

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```

wsgiref.simple_server.**demo_app**(*environ*, *start_response*)

This function is a small but complete WSGI application that returns a text page containing the message "Hello world!" and a list of the key/value pairs provided in the *environ* parameter. It's useful for verifying that a WSGI server (such as *wsgiref.simple_server*) is able to run a simple WSGI application correctly.

class wsgiref.simple_server.**WSGIServer**(*server_address*, *RequestHandlerClass*)

Create a *WSGIServer* instance. *server_address* should be a (host,port) tuple, and *RequestHandlerClass* should be the subclass of *http.server.BaseHTTPRequestHandler* that will be used to process requests.

You do not normally need to call this constructor, as the *make_server()* function can handle all the details for you.

WSGIServer is a subclass of *http.server.HTTPServer*, so all of its methods (such as serve_forever() and handle_request()) are available. *WSGIServer* also provides these WSGI-specific methods:

set_app(*application*)

Sets the callable *application* as the WSGI application that will receive requests.

get_app()

Returns the currently-set application callable.

Normally, however, you do not need to use these additional methods, as *set_app()* is normally called by *make_server()*, and the *get_app()* exists mainly for the benefit of request handler instances.

class wsgiref.simple_server.**WSGIRequestHandler**(*request*, *client_address*, *server*)

Create an HTTP handler for the given *request* (i.e. a socket), *client_address* (a (host,port) tuple), and *server* (*WSGIServer* instance).

You do not need to create instances of this class directly; they are automatically created as needed by *WSGIServer* objects. You can, however, subclass this class and supply it as a *handler_class* to the *make_server()* function. Some possibly relevant methods for overriding in subclasses:

get_environ()
Returns a dictionary containing the WSGI environment for a request. The default implementation copies the contents of the *WSGIServer* object's `base_environ` dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in **PEP 3333**.

get_stderr()
Return the object that should be used as the `wsgi.errors` stream. The default implementation just returns `sys.stderr`.

handle()
Process the HTTP request. The default implementation creates a handler instance using a *wsgiref.handlers* class to implement the actual WSGI application interface.

21.4.4 `wsgiref.validate` —WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using *wsgiref.validate*. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete **PEP 3333** compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the `paste.lint` module from Ian Bicking's "Python Paste" library.

wsgiref.validate.**validator**(*application*)
Wrap *application* and return a new WSGI application object. The returned application will forward all requests to the original *application*, and will check that both the *application* and the server invoking it are conforming to the WSGI specification and to **RFC 2616**.

Any detected nonconformance results in an *AssertionError* being raised; note, however, that how these errors are handled is server-dependent. For example, *wsgiref.simple_server* and other servers based on *wsgiref.handlers* (that don't override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to `sys.stderr` or some other error stream.

This wrapper may also generate output using the *warnings* module to indicate behaviors that are questionable but which may not actually be prohibited by **PEP 3333**. Unless they are suppressed using Python command-line options or the *warnings* API, any such warnings will be written to `sys.stderr` (*not* `wsgi.errors`, unless they happen to be the same object).

用法示例:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)
```

(下页继续)

(续上页)

```
    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return b"Hello World"

# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

21.4.5 wsgiref.handlers –server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

class wsgiref.handlers.**CGIHandler**

CGI-based invocation via sys.stdin, sys.stdout, sys.stderr and os.environ. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke CGIHandler().run(app), where app is the WSGI application object you wish to invoke.

This class is a subclass of *BaseCGIHandler* that sets wsgi.run_once to true, wsgi.multithread to false, and wsgi.multiprocess to true, and always uses *sys* and *os* to obtain the necessary CGI streams and environment.

class wsgiref.handlers.**IISCGIHandler**

A specialized alternative to *CGIHandler*, for use when deploying on Microsoft's IIS web server, without having set the config allowPathInfo option (IIS>=7) or metabase allowPathInfoForScriptMappings (IIS<7).

By default, IIS gives a PATH_INFO that duplicates the SCRIPT_NAME at the front, causing problems for WSGI applications that wish to implement routing. This handler strips any such duplicated path.

IIS can be configured to pass the correct PATH_INFO, but this causes another bug where PATH_TRANSLATED is wrong. Luckily this variable is rarely used and is not guaranteed by WSGI. On IIS<7, though, the setting can only be made on a vhost level, affecting all other script mappings, many of which break when exposed to the PATH_TRANSLATED bug. For this reason IIS<7 is almost never deployed with the fix. (Even IIS7 rarely uses it because there is still no UI for it.)

There is no way for CGI code to tell whether the option was set, so a separate handler class is provided. It is used in the same way as *CGIHandler*, i.e., by calling IISCGIHandler().run(app), where app is the WSGI application object you wish to invoke.

3.2 新版功能.

class wsgiref.handlers.**BaseCGIHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *CGIHandler*, but instead of using the *sys* and *os* modules, the CGI environment and I/O streams are specified explicitly. The *multithread* and *multiprocess* values are used to set the wsgi.multithread and wsgi.multiprocess flags for any applications run by the handler instance.

This class is a subclass of *SimpleHandler* intended for use with software other than HTTP "origin servers". If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a Status: header to send an HTTP status, you probably want to subclass this instead of *SimpleHandler*.

class wsgiref.handlers.**SimpleHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Similar to *BaseCGIHandler*, but designed for use with HTTP origin servers. If you are writing an HTTP server implementation, you will probably want to subclass this instead of *BaseCGIHandler*.

This class is a subclass of *BaseHandler*. It overrides the `__init__()`, `get_stdin()`, `get_stderr()`, `add_cgi_vars()`, `_write()`, and `_flush()` methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the `stdin`, `stdout`, `stderr`, and `environ` attributes.

The *write()* method of *stdout* should write each chunk in full, like *io.BufferedIOBase*.

class wsgiref.handlers.**BaseHandler**

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

BaseHandler instances have only one method intended for external use:

run(*app*)

Run the specified WSGI application, *app*.

All of the other *BaseHandler* methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:

_write(*data*)

Buffer the bytes *data* for transmission to the client. It's okay if this method actually transmits the data; *BaseHandler* just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

_flush()

Force buffered data to be transmitted to the client. It's okay if this method is a no-op (i.e., if *_write()* actually sends the data).

get_stdin()

Return an input stream object suitable for use as the `wsgi.input` of the request currently being processed.

get_stderr()

Return an output stream object suitable for use as the `wsgi.errors` of the request currently being processed.

add_cgi_vars()

Insert CGI variables for the current request into the `environ` attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized *BaseHandler* subclass.

Attributes and methods for customizing the WSGI environment:

wsgi_multithread

The value to be used for the `wsgi.multithread` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_multiprocess

The value to be used for the `wsgi.multiprocess` environment variable. It defaults to true in *BaseHandler*, but may have a different default (or be set by the constructor) in the other subclasses.

wsgi_run_once

The value to be used for the `wsgi.run_once` environment variable. It defaults to false in *BaseHandler*, but *CGIHandler* sets it to true by default.

os_environ

The default environment variables to be included in every request's WSGI environment. By default, this is

a copy of `os.environ` at the time that *wsgiref.handlers* was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

server_software
If the *origin_server* attribute is set, this attribute's value is used to set the default `SERVER_SOFTWARE` WSGI environment variable, and also to set a default `Server:` header in HTTP responses. It is ignored for handlers (such as *BaseCGIHandler* and *CGIHandler*) that are not HTTP origin servers.

在 3.3 版更改: The term "Python" is replaced with implementation specific term like "CPython", "Jython" etc.

get_scheme()
Return the URL scheme being used for the current request. The default implementation uses the `guess_scheme()` function from *wsgiref.util* to guess whether the scheme should be "http" or "https", based on the current request's `environ` variables.

setup_environ()
Set the `environ` attribute to a fully-populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the *get_stdin()*, *get_stderr()*, and *add_cgi_vars()* methods and the *wsgi_file_wrapper* attribute. It also inserts a `SERVER_SOFTWARE` key if not present, as long as the *origin_server* attribute is a true value and the *server_software* attribute is set.

Methods and attributes for customizing exception handling:

log_exception(*exc_info*)
Log the *exc_info* tuple in the server log. *exc_info* is a `(type, value, traceback)` tuple. The default implementation simply writes the traceback to the request's `wsgi.errors` stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

traceback_limit
The maximum number of frames to include in tracebacks output by the default *log_exception()* method. If `None`, all frames are included.

error_output(*environ*, *start_response*)
This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error information using `sys.exc_info()`, and should pass that information to *start_response* when calling it (as described in the "Error Handling" section of **PEP 3333**).

The default implementation just uses the *error_status*, *error_headers*, and *error_body* attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it's not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn't include any.

error_status
The HTTP status used for error responses. This should be a status string as defined in **PEP 3333**; it defaults to a 500 code and message.

error_headers
The HTTP headers used for error responses. This should be a list of WSGI response headers (`(name, value)` tuples), as described in **PEP 3333**. The default list just sets the content type to `text/plain`.

error_body
The error response body. This should be an HTTP response body bytestring. It defaults to the plain text, "A server error occurred. Please contact the administrator."

Methods and attributes for **PEP 3333**'s "Optional Platform-Specific File Handling" feature:

wsgi_file_wrapper
A `wsgi.file_wrapper` factory, or `None`. The default value of this attribute is the *`wsgiref.util.FileWrapper`* class.

sendfile()
Override to implement platform-specific file transmission. This method is called only if the application's return value is an instance of the class specified by the *`wsgi_file_wrapper`* attribute. It should return a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

origin_server
This attribute should be set to a true value if the handler's *`_write()`* and *`_flush()`* are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special `Status:` header.

This attribute's default value is true in *`BaseHandler`*, but false in *`BaseCGIHandler`* and *`CGIHandler`*.

http_version
If *`origin_server`* is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to `"1.0"`.

wsgiref.handlers.**read_environ**()
Transcode CGI variables from `os.environ` to PEP 3333 "bytes in unicode" strings, returning a new dictionary. This function is used by *`CGIHandler`* and *`IISCGIHandler`* in place of directly using `os.environ`, which is not necessarily WSGI-compliant on all platforms and web servers using Python 3 –specifically, ones where the OS's actual environment is Unicode (i.e. Windows), or ones where the environment is bytes, but the system encoding used by Python to decode it is anything other than ISO-8859-1 (e.g. Unix systems using UTF-8).

If you are implementing a CGI-based handler of your own, you probably want to use this routine instead of just copying values out of `os.environ` directly.

3.2 新版功能.

21.4.6 例子

This is a working "Hello World" WSGI application:

```
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable
# object that accepts two arguments. For that purpose, we're going to
# use a function (note that you're not limited to a function, you can
# use a class for example). The first argument passed to the function
# is a dictionary containing CGI-style environment variables and the
# second variable is the callable object (see PEP 333).
def hello_world_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain; charset=utf-8')]  # HTTP Headers
    start_response(status, headers)
```

(下页继续)

(续上页)

```
    # The returned object is going to be printed
    return [b"Hello World"]

with make_server('', 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

21.5 urllib —URL 处理模块

源代码: Lib/urllib/

---

urllib 是一个收集了多个用到 URL 的模块的包：

- *urllib.request* 打开和读取 URL
- *urllib.error* 包含*urllib.request* 抛出的异常
- *urllib.parse* 用于解析 URL
- *urllib.robotparser* 用于解析 robots.txt 文件

21.6 urllib.request —用于打开 URL 的可扩展库

源码： Lib/urllib/request.py

---

urllib.request 模块定义了适用于在各种复杂情况下打开 URL（主要为 HTTP）的函数和类—例如基本认证、摘要认证、重定向、cookies 及其它。

参见:

The Requests package is recommended for a higher-level HTTP client interface.

urllib.request 模块定义了以下函数：

urllib.request.**urlopen**(*url*, *data=None*[, *timeout*], *, *cafile=None*, *capath=None*, *cadefault=False*, *context=None*)

打开统一资源定位地址 *url*，可以是一个字符串或一个*Request* 对象。

data 必须是一个对象，用于给出要发送到服务器的附加数据，若不需要发送数据则为 None。详情请参阅*Request* 。

urllib.request 模块使用 HTTP/1.1 并且在其 HTTP 请求中包含 Connection:close 头。

timeout 为可选参数，用于指定阻塞操作（如连接尝试）的超时时间，单位为秒。如未指定，将使用全局默认超时参数）。本参数实际仅对 HTTP、HTTPS 和 FTP 连接有效。

如果给定了 *context* 参数，则必须是一个*ssl.SSLContext* 实例，用于描述各种 SSL 参数。更多详情请参阅*HTTPSConnection* 。

cafile 和 *capath* 为可选参数，用于为 HTTPS 请求指定一组受信 CA 证书。*cafile* 应指向包含 CA 证书的单个文件，*capath* 则应指向哈希证书文件的目录。更多信息可参阅*ssl.SSLContext.load_verify_locations()* 。

cadefault 将被忽略。

This function always returns an object which can work as a *context manager* and has methods such as

- `geturl()` —return the URL of the resource retrieved, commonly used to determine if a redirect was followed
- `info()` —return the meta-information of the page, such as headers, in the form of an *email.message_from_string()* instance (see Quick Reference to HTTP Headers)
- `getcode()` –return the HTTP status code of the response.

对于 HTTP 和 HTTPS 的 URL 而言，本函数将返回一个稍经修改的*http.client.HTTPResponse* 对象。除了上述 3 个新的方法之外，还有 msg 属性包含了与*reason* 属性相同的信息—服务器返回的原因描述文字，而不是*HTTPResponse* 的文档所述的响应头部信息。

对于 FTP、文件、数据的 URL，以及由传统的*URLopener* 和*FancyURLopener* 类处理的请求，本函数将返回一个 `urllib.response.addinfourl` 对象。

协议错误时引发*URLError*。

请注意，如果没有处理函数对请求进行处理，则有可能会返回 `None` 。尽管默认安装的全局*OpenerDirector* 会用*UnknownHandler* 来确保不会发生这种情况。

此外，如果检测到设置了代理（比如设置了 `http_proxy` 之类的环境变量），默认会安装*ProxyHandler* 并确保通过代理处理请求。

Python 2.6 以下版本中留存的 `urllib.urlopen` 函数已停止使用了；*urllib.request.urlopen()* 对应于传统的 `urllib2.urlopen` 。对代理服务的处理是通过将字典参数传给 `urllib.urlopen` 来完成的，可以用*ProxyHandler* 对象获取到代理处理函数。

在 3.2 版更改: 增加了 *cafile* 与 *capath*。

在 3.2 版更改: 现在如果可行（指*ssl.HAS_SNI* 为真），支持 HTTPS 虚拟主机

3.2 新版功能: *data* 可以是一个可迭代对象。

在 3.3 版更改: 增加了 *cadefault*。

在 3.4.3 版更改: 增加了 *context*。

3.6 版后已移除: *cafile* 、*capath* 和 *cadefault* 已废弃，转而推荐使用 *context*。请改用*ssl.SSLContext.load_cert_chain()* 或让*ssl.create_default_context()* 选取系统信任的 CA 证书。

`urllib.request.`**`install_opener`**(*opener*)

安装一个*OpenerDirector* 实例，作为默认的全局打开函数。仅当 urlopen 用到该打开函数时才需要安装；否则，只需调用*OpenerDirector.open()* 而不是*urlopen()*。代码不会检查是否真的属于*OpenerDirector* 类，所有具备适当接口的类都能适用。

`urllib.request.`**`build_opener`**([*handler*, ...])

返回一个*OpenerDirector* 实例，以给定顺序把处理函数串联起来。处理函数可以是*BaseHandler* 的实例，也可以是*BaseHandler* 的子类（这时构造函数必须允许不带任何参数的调用）。以下类的实例将位于 处理函数之前，除非 处理函数已包含这些类、其实例或其子类: *ProxyHandler*（如果检测到代理设置）、*UnknownHandler* 、*HTTPHandler* 、*HTTPDefaultErrorHandler* 、*HTTPRedirectHandler* 、*FTPHandler* 、*FileHandler* 、*HTTPErrorProcessor* 。

若 Python 安装有 SLL 支持（指可以导入*ssl* 模块），亦会加入*HTTPSHandler*。

A *BaseHandler* subclass may also change its `handler_order` attribute to modify its position in the handlers list.

urllib.request.**pathname2url**(*path*)
将路径名 *path* 从路径本地写法转换为 URL 路径部件采用的格式。这不会生成完整的 URL。返回值将会用*quote()* 函数加以编码。

urllib.request.**url2pathname**(*path*)
从百分号编码的 URL 中将 *path* 部分转换为本地路径的写法。本函数不接受完整的 URL。本函数利用*unquote()* 解码 *path* 。

urllib.request.**getproxies**()
本辅助函数将返回一个字典，表示各方案映射的代理服务器 URL。本函数扫描名为 <scheme>_proxy 的环境变量，不区分大小写，首先会考虑所有操作系统。如果环境变量无法找到，则会从 Mac OS X 的 Mac OSX 系统配置或 Windows 系统的注册表查找代理服务器信息。如果同时存在小写和大写环境变量（且内容不一致），则首选小写。

注解： 如果存在环境变量 REQUEST_METHOD ，通常表示脚本运行于 CGI 环境中，则环境变量 HTTP_PROXY （大写的 _PROXY）将会被忽略。这是因其可以由客户端用 HTTP 头部信息 “Proxy:” 注入。若要在 CGI 环境中使用 HTTP 代理，请显式使用 `ProxyHandler ，或确保变量名称为小写（或至少是 _proxy 后缀）。

提供了以下类：

class urllib.request.**Request**(*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *method=None*)
URL 请求的抽象类。

url 应该是是一个含有一个有效的统一资源定位地址的字符串。

data must be an object specifying additional data to send to the server, or None if no such data is needed. Currently HTTP requests are the only ones that use *data*. The supported object types include bytes, file-like objects, and iterables. If no Content-Length nor Transfer-Encoding header field has been provided, *HTTPHandler* will set these headers according to the type of *data*. Content-Length will be used to send bytes objects, while Transfer-Encoding: chunked as specified in **RFC 7230**, Section 3.3.1 will be used to send files and other iterables.

对于 HTTP POST 请求方法而言，*data* 应该是标准 *application/x-www-form-urlencoded* 格式的缓冲区。*urllib.parse.urlencode()* 函数的参数为映射对象或二元组序列，并返回一个该编码格式的 ASCII 字符串。在用作 *data* 参数之前，应将其编码为字节串。

headers 应为字典对象，视同于用每个键和值作为参数去调用*add_header()*。通常用于 User-Agent 头部数据的 “伪装”，浏览器用这些头部数据标识自己——某些 HTTP 服务器只允许来自普通浏览器的请求，而不接受来自脚本的请求。例如，Mozilla Firefox 可能将自己标识为 "Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"，而*urllib* 的默认用户代理字符串则是 "Python-urllib/2.6" (在 Python 2.6 上)。

如果给出了 *data* 参数，则应当包含合适的 Content-Type 头部信息。若未提供且 *data* 不是 None，则会把 Content-Type: application/x-www-form-urlencoded 加入作为默认值。

The final two arguments are only of interest for correct handling of third-party HTTP cookies:

origin_req_host 应为发起初始会话的请求主机，定义参见 **RFC 2965** 。默认指为 “http.cookiejar.request_host(self)“ 。这是用户发起初始请求的主机名或 IP 地址。假设请求是针对 HTML 文档中的图片数据发起的，则本属性应为对包含图像的页面发起请求的主机。

unverifiable 应该标示出请求是否无法验证，定义参见 **RFC 2965** 。默认值为 False 。所谓无法验证的请求，是指用户没有机会对请求的 URL 做验证。例如，如果请求是针对 HTML 文档中的图像，用户没有机会去许可能自动读取图像，则本属性应为 True。

method 应为字符串，标示要采用的 HTTP 请求方法（例如 'HEAD' ）。如果给出本参数，其值会存储在*method* 属性中，并由*get_method()* 使用。如果 *data* 为 “None“ 则默认值为 'GET' ，否则为

'POST'。子类可以设置*method*属性来标示不同的默认请求方法。

注解: 如果 data 对象无法分多次传递其内容（比如文件或只能生成一次内容的可迭代对象）并且由于 HTTP 重定向或身份验证而发生请求重试行为，则该请求不会正常工作。*data* 是紧挨着头部信息发送给 HTTP 服务器的。现有库不支持 HTTP 100-continue 的征询。

在 3.3 版更改: Request 类增加了*Request.method*参数。

在 3.4 版更改: 默认*Request.method*可以在类中标明。

在 3.6 版更改: 如果给出了 Content-Length，且 *data* 既不为 None 也不是字节串对象，则不会触发错误。而会退而求其次采用分块传输的编码格式。

class urllib.request.**OpenerDirector**
OpenerDirector 类通过串接在一起的*BaseHandler*打开 URL，并负责管理 handler 链及从错误中恢复。

class urllib.request.**BaseHandler**
这是所有已注册 handler 的基类，只做了简单的注册机制。

class urllib.request.**HTTPDefaultErrorHandler**
为 HTTP 错误响应定义的默认 handler，所有响应都会转为*HTTPError*异常。

class urllib.request.**HTTPRedirectHandler**
一个用于处理重定向的类。

class urllib.request.**HTTPCookieProcessor**(*cookiejar=None*)
一个用于处理 HTTP Cookies 的类。

class urllib.request.**ProxyHandler**(*proxies=None*)
让请求转去代理服务。若给出了 *proxies*，则其必须是一个将协议名称映射为代理 URL 的字典对象。默认是从环境变量 <protocol>_proxy 中读取代理服务的列表。如果没有设置代理环境变量，则 Windows 会从注册表的 Internet 设置部分获取代理设置，而 Mac OS X 则会从 OS X 系统配置框架中读取代理信息。

若要禁用自动检测出来的代理，请传入空的字典对象。

环境变量 no_proxy 可用于指定不必通过代理访问的主机；应为逗号分隔的主机名后缀列表，可加上 :port，例如 cern.ch,ncsa.uiuc.edu,some.host:8080。

注解: 如果设置了 REQUEST_METHOD 变量，则会忽略 HTTP_PROXY；参阅*getproxies()*文档。

---

class urllib.request.**HTTPPasswordMgr**
维护 (realm, uri) -> (user, password) 映射数据库。

class urllib.request.**HTTPPasswordMgrWithDefaultRealm**
维护 (realm, uri) -> (user, password) 映射数据库。realm 为 None 视作全匹配，若没有其他合适的安全区域就会检索它。

class urllib.request.**HTTPPasswordMgrWithPriorAuth**
HTTPPasswordMgrWithDefaultRealm 的一个变体，也带有 uri -> is_authenticated 映射数据库。可被 BasicAuth 处理函数用于确定立即发送身份认证凭据的时机，而不是先等待 401 响应。

3.5 新版功能.

class urllib.request.**AbstractBasicAuthHandler**(*password_mgr=None*)
这是一个帮助完成 HTTP 身份认证的混合类，对远程主机和代理都适用。参数 *password_mgr* 应与*HTTPPasswordMgr*兼容；关于必须支持哪些接口，请参阅*HTTPPasswordMgr 对象*对象的

章节。如果 *password_mgr* 还提供 `is_authenticated` 和 `update_authenticated` 方法（请参阅*HTTPPasswordMgrWithPriorAuth 对象* 对象），则 handler 将对给定 URI 用到 `is_authenticated` 的结果，来确定是否随请求发送身份认证凭据。如果该 URI 的 `is_authenticated` 返回 `True`，则发送凭据。如果 `is_authenticated` 为 `False`，则不发送凭据，然后若收到 `401` 响应，则使用身份认证凭据重新发送请求。如果身份认证成功，则调用 `update_authenticated` 设置该 URI 的 `is_authenticated` 为 `True`，这样后续对该 URI 或其所有父 URI 的请求将自动包含该身份认证凭据。

3.5 新版功能: 增加了对 `is_authenticated` 的支持。

class urllib.request.**HTTPBasicAuthHandler**(*password_mgr=None*)
处理与远程主机的身份验证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。如果给出错误的身份认证方式，HTTPBasicAuthHandler 将会触发*ValueError* 。

class urllib.request.**ProxyBasicAuthHandler**(*password_mgr=None*)
处理有代理服务时的身份验证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。

class urllib.request.**AbstractDigestAuthHandler**(*password_mgr=None*)
这是一个帮助完成 HTTP 身份认证的混合类，对远程主机和代理都适用。参数 *password_mgr* 应与*HTTPPasswordMgr* 兼容；关于必须支持哪些接口，请参阅*HTTPPasswordMgr 对象* 的章节。

class urllib.request.**HTTPDigestAuthHandler**(*password_mgr=None*)
处理远程主机的身份验证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。如果同时添加了 digest 身份认证 handler 和 basic 身份认证 handler，则会首先尝试 digest 身份认证。如果 digest 身份认证再返回 40x 响应，会再发送到 basic 身份验证 handler 进行处理。如果给出 Digest 和 Basic 之外的身份认证方式，本 handler 方法将会触发*ValueError* 。

在 3.3 版更改: 碰到不支持的认证方式时，将会触发*ValueError* 。

class urllib.request.**ProxyDigestAuthHandler**(*password_mgr=None*)
处理有代理服务时的身份验证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。

class urllib.request.**HTTPHandler**
用于打开 HTTP URL 的 handler 类。

class urllib.request.**HTTPSHandler**(*debuglevel=0*, *context=None*, *check_hostname=None*)
用于打开 HTTPS URL 的 handler 类。*context* 和 *check_hostname* 的含义与*http.client.HTTPSConnection* 的一样。

在 3.2 版更改: 添加 *context* 和 *check_hostname* 参数。

class urllib.request.**FileHandler**
打开本地文件。

class urllib.request.**DataHandler**
打开数据 URL。

3.4 新版功能.

class urllib.request.**FTPHandler**
打开 FTP 统一资源定位地址。

class urllib.request.**CacheFTPHandler**
打开 FTP URL，并将打开的 FTP 连接存入缓存，以便最大程度减少延迟。

class urllib.request.**UnknownHandler**
处理所有未知类型 URL 的兜底类。

class urllib.request.**HTTPErrorProcessor**
处理出错的 HTTP 响应。

21.6.1 Request 对象

以下方法介绍了*Request* 的公开接口，因此子类可以覆盖所有这些方法。这里还定义了几个公开属性，客户端可以利用这些属性了解经过解析的请求。

Request.**full_url**
传给构造函数的原始 URL。

在 3.4 版更改.

Request.full_url 是一个带有 setter、getter 和 deleter 的属性。读取*full_url* 属性将会返回分段的初始请求 URL。

Request.**type**
URI 方式。

Request.**host**
URI 权限，通常是整个主机，但也有可能带有冒号分隔的端口号。

Request.**origin_req_host**
请求的原始主机，不含端口。

Request.**selector**
URI 路径。若*Request* 使用代理，选择器将会是传给代理的完整 URL。

Request.**data**
请求的数据体，未给出则为 None 。

在 3.4 版更改: 现在如果修改*Request.data* 的值，则会删除之前设置或计算过的 “Content-Length” 头部信息。

Request.**unverifiable**
布尔，表明请求是否为 **RFC 2965** 中定义的无法证实的。

Request.**method**
要采用的 HTTP 请求方法。默认为*None*，表示*get_method()* 将对方法进行正常处理。设置本值可以覆盖*get_method()* 中的默认处理过程，设置方式可以是在*Request* 的子类中给出默认值，也可以通过 *method* 参数给*Request* 构造函数传入一个值。

3.3 新版功能.

在 3.4 版更改: 现在可以在子类中设置默认值；而之前只能通过构造函数参数进行设置。

Request.**get_method**()
返回表示 HTTP 请求方法的字符串。如果*Request.method* 不为，则返回其值。否则若*Request.data* 为 None 则返回 'GET'，不为 None 则返回 'POST' 。只对 HTTP 请求有效。

在 3.3 版更改: 现在 get_method 会兼顾*Request.method* 的值。

Request.**add_header**(*key*, *val*)
向请求添加一项头部信息。目前只有 HTTP handler 才会处理头部信息，将其加入发给服务器的头部信息列表中。请注意，同名的头部信息只能出现一次，如果 *key* 发生冲突，后续的调用结果将会覆盖先前的。目前，这并不会减少 HTTP 的功能，因为所有多次使用仍有意义的头部信息，都有一种特定方式获得与只用一次时相同的功能。

Request.**add_unredirected_header**(*key*, *header*)
添加一个不会被加入重定向请求的头部。

Request.**has_header**(*header*)
返回本实例是否带有命名头部信息（对常规数据和非重定向数据都会检测）。

Request.**remove_header**(*header*)
从本请求实例中移除指定命名的头部信息（对常规数据和非重定向数据都会检测）。

3.4 新版功能.

Request.**get_full_url**()
返回构造器中给定的 URL。

在 3.4 版更改.

返回*Request.full_url*

Request.**set_proxy**(*host*, *type*)
连接代理服务器，为当前请求做准备。*host* 和 *type* 将会取代本实例中的对应值，selector 将会是构造函数中给出的初始 URL。

Request.**get_header**(*header_name*, *default=None*)
返回给定头部信息的数据。如果该头部信息不存在，返回默认值。

Request.**header_items**()
返回头部信息，形式为（名称, 数据）的元组列表。

在 3.4 版更改: 自 3.3 起已弃用的下列方法已被删除：add_data、has_data、get_data、get_type、get_host、get_selector、get_origin_req_host 和 is_unverifiable 。

21.6.2 OpenerDirector 对象

OpenerDirector 实例有以下方法：

OpenerDirector.**add_handler**(*handler*)
handler should be an instance of *BaseHandler*. The following methods are searched, and added to the possible chains (note that HTTP errors are a special case).

- `protocol_open()` —signal that the handler knows how to open *protocol* URLs.
- `http_error_type()` —signal that the handler knows how to handle HTTP errors with HTTP error code *type*.
- `protocol_error()` —signal that the handler knows how to handle errors from (non-`http`) *protocol*.
- `protocol_request()` —signal that the handler knows how to pre-process *protocol* requests.
- `protocol_response()` —signal that the handler knows how to post-process *protocol* responses.

OpenerDirector.**open**(*url*, *data=None*[, *timeout*])
打开给定的 *url** （可以是 *request* 对象或字符串），可以传入 **data* 。参数、返回值和引发的异常均与*urlopen()* 相同，其实只是去调用了当前安装的全局*OpenerDirector* 中的*open()* 方法。可选的 *timeout* 参数指定了阻塞操作（如尝试连接）的超时值（以秒为单位）（若未指定则采用全局默认的超时设置）。实际上，超时特性仅适用于 HTTP、HTTPS 和 FTP 连接。

OpenerDirector.**error**(*proto*, **args*)
Handle an error of the given protocol. This will call the registered error handlers for the given protocol with the given arguments (which are protocol specific). The HTTP protocol is a special case which uses the HTTP response code to determine the specific error handler; refer to the `http_error_*()` methods of the handler classes.

返回值和异常均与*urlopen()* 相同。

OpenerDirector 对象分 3 个阶段打开 URL：

每个阶段中调用这些方法的次序取决于 handler 实例的顺序。

1. Every handler with a method named like `protocol_request()` has that method called to pre-process the request.
2. Handlers with a method named like `protocol_open()` are called to handle the request. This stage ends when a handler either returns a non-*None* value (ie. a response), or raises an exception (usually *URLError*). Exceptions are allowed to propagate.

   In fact, the above algorithm is first tried for methods named `default_open()`. If all such methods return *None*, the algorithm is repeated for methods named like `protocol_open()`. If all such methods return *None*, the algorithm is repeated for methods named `unknown_open()`.

   请注意，这些方法的代码可能会调用*OpenerDirector* 父实例的*open()* 和*error()* 方法。
3. Every handler with a method named like `protocol_response()` has that method called to post-process the response.

21.6.3 BaseHandler 对象

BaseHandler 对象提供了一些直接可用的方法，以及其他一些可供派生类使用的方法。以下是可供直接使用的方法：

`BaseHandler.`**`add_parent`**`(`*director*`)`
将 director 加为父 handler。

`BaseHandler.`**`close`**`()`
移除所有父 OpenerDirector。

以下属性和方法仅供*BaseHandler* 的子类使用：

注 解: The convention has been adopted that subclasses defining `protocol_request()` or `protocol_response()` methods are named `*Processor`; all others are named `*Handler`.

`BaseHandler.`**`parent`**
一个可用的*OpenerDirector*，可用于以其他协议打开 URI，或处理错误。

`BaseHandler.`**`default_open`**`(`*req*`)`
本方法在*BaseHandler* 中 未予定义，但其子类若要捕获所有 URL 则应进行定义。

若实现了本方法，则会被父类*OpenerDirector* 调用。应返回一个类文件对象，类似于*OpenerDirector* 的*open()* 的返回值，或返回 “None”。触发的异常应为*URLError*，除非发生真的异常（比如*MemoryError* 就不应变为 URLError）。

本方法将会在所有协议的 open 方法之前被调用。

`BaseHandler.`**`protocol_open`**`(`*req*`)`
本方法在*BaseHandler* 中 未予定义，但其子类若要处理给定协议的 URL 则应进行定义。

若定义了本方法，将会被父*OpenerDirector* 对象调用。返回值和*default_open()* 的一样。

`BaseHandler.`**`unknown_open`**`(`*req*`)`
本方法在*BaseHandler* 中 未予定义，但其子类若要捕获并打开所有未注册 handler 的 URL，则应进行定义。

若实现了本方法，将会被*parent* 属性指向的父*OpenerDirector* 调用。返回值和*default_open()* 的一样。

`BaseHandler.`**`http_error_default`**`(`*req*, *fp*, *code*, *msg*, *hdrs*`)`
本方法在*BaseHandler* 中 未予定义，但其子类若要为所有未定义 handler 的 HTTP 错误提供一个兜底

方法，则应进行覆盖。*OpenerDirector* 会自动调用本方法，获取错误信息，而通常在其他时候不应去调用。

req 会是一个*Request* 对象，*fp* 是一个带有 HTTP 错误体的文件类对象，*code* 是三位数的错误码，*msg* 是供用户阅读的解释信息，*hdrs* 则是一个包含出错头部信息的映射对象。

返回值和触发的异常应与*urlopen()* 的相同。

BaseHandler.**http_error_nnn**(*req*, *fp*, *code*, *msg*, *hdrs*)
nnn 应为三位数的 HTTP 错误码。本方法在*BaseHandler* 中也未予定义，但当子类的实例发生代码为 *nnn* 的 HTTP 错误时，若方法存在则会被调用。

子类应该重写本方法，以便能处理指定的 HTTP 错误。

参数、返回值和触发的异常应与*http_error_default()* 相同。

BaseHandler.**protocol_request**(*req*)
本方法在*BaseHandler* 中 未予定义，但其子类若要对给定协议的请求进行预处理，则应进行定义。

若实现了本方法，将会被父*OpenerDirector* 调用。*req* 将为*Request* 对象。返回值应为*Request* 对象。

BaseHandler.**protocol_response**(*req*, *response*)
本方法在*BaseHandler* 中 未予定义，但其子类若要对给定协议的请求进行后处理，则应进行定义。

若实现了本方法，将会被父*OpenerDirector* 调用。*req* 将为*Request* 对象。*response* 应实现与*urlopen()* 返回值相同的接口。返回值应实现与*urlopen()* 返回值相同的接口。

21.6.4 HTTPRedirectHandler 对象

注解： 某些 HTTP 重定向操作需要用到本模块的客户端代码。这时会触发*HTTPError*。有关各种重定向代码的确切含义，请参阅 **RFC 2616** 。

如果 HTTPRedirectHandler 呈现的重定向 URL 不是 HTTP、HTTPS 或 FTP URL，则出于安全考虑将触发 HTTPError 异常。

HTTPRedirectHandler.**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)
返回*Request* 或 None 对象作为重定向行为的响应。当服务器接收到重定向请求时，http_error_30*() 方法的默认实现代码将会调用本方法。如果确实应该发生重定向，则返回一个新的*Request* 对象，使得 http_error_30*() 能重定向至 *newurl*。否则，若没有 handler 会处理此 URL，则会引发*HTTPError*；或者本方法不能处理但或许会有其他 handler 会处理，则返回 None 。

注解： 本方法的默认实现代码并未严格遵循 **RFC 2616**，即 POST 请求的 301 和 302 响应不得在未经用户确认的情况下自动进行重定向。现实情况下，浏览器确实允许自动重定向这些响应，将 POST 更改为 GET ，于是默认实现代码就复现了这种处理方式。

HTTPRedirectHandler.**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)
重定向到 Location: 或 URI: URL。这个方法会在得到 HTTP ‘moved permanently’ 响应时由上级*OpenerDirector* 来调用。

HTTPRedirectHandler.**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)
与*http_error_301()* 相同，不过是发生 “found” 响应时的调用。

HTTPRedirectHandler.**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)
与*http_error_301()* 相同，不过是发生 “see other” 响应时的调用。

HTTPRedirectHandler.**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)
与*http_error_301()* 相同，不过是发生“temporary redirect”响应时的调用。

21.6.5 HTTPCookieProcessor 对象

HTTPCookieProcessor 的实例具备一个属性：

HTTPCookieProcessor.**cookiejar**
cookie 存放在*http.cookiejar.CookieJar* 中。

21.6.6 ProxyHandler 对象

ProxyHandler.**protocol_open**(*request*)
The *ProxyHandler* will have a method `protocol_open()` for every *protocol* which has a proxy in the *proxies* dictionary given in the constructor. The method will modify requests to go through the proxy, by calling `request.set_proxy()`, and call the next handler in the chain to actually execute the protocol.

21.6.7 HTTPPasswordMgr 对象

以下方法*HTTPPasswordMgr* 和*HTTPPasswordMgrWithDefaultRealm* 对象均有提供。

HTTPPasswordMgr.**add_password**(*realm*, *uri*, *user*, *passwd*)
uri 可以是单个 URI，也可以是 URI 列表。*realm*、*user* 和 *passwd* 必须是字符串。这使得在为 *realm* 和超级 URI 进行身份认证时，`(user, passwd)` 可用作认证令牌。

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)
为给定 realm 和 URI 获取用户名和密码。如果没有匹配的用户名和密码，本方法将会返回 `(None, None)`。

对于*HTTPPasswordMgrWithDefaultRealm* 对象，如果给定 *realm* 没有匹配的用户名和密码，将搜索 realm `None`。

21.6.8 HTTPPasswordMgrWithPriorAuth 对象

这是*HTTPPasswordMgrWithDefaultRealm* 的扩展，以便对那些需要一直发送认证凭证的 URI 进行跟踪。

HTTPPasswordMgrWithPriorAuth.**add_password**(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)
realm、*uri*、*user*、*passwd* 的含义与*HTTPPasswordMgr.add_password()* 的相同。*is_authenticated* 为给定 URI 或 URI 列表设置 `is_authenticated` 标志的初始值。如果 *is_authenticated* 设为 `True`，则会忽略 *realm*。

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)
与*HTTPPasswordMgrWithDefaultRealm* 对象的相同。

HTTPPasswordMgrWithPriorAuth.**update_authenticated**(*self*, *uri*, *is_authenticated=False*)
更新给定 *uri* 或 URI 列表的 `is_authenticated` 标志。

HTTPPasswordMgrWithPriorAuth.**is_authenticated**(*self*, *authuri*)
返回给定 URI `is_authenticated` 标志的当前状态。

21.6.9 AbstractBasicAuthHandler 对象

`AbstractBasicAuthHandler.`**`http_error_auth_reqed`**(*authreq*, *host*, *req*, *headers*)
通过获取用户名和密码并重新尝试请求，以处理身份认证请求。*authreq* 应该是请求中包含 realm 的头部信息名称，*host* 指定了需要进行身份认证的 URL 和路径，*req* 应为 (已失败的) `Request` 对象，*headers* 应该是出错的头部信息。

host 要么是一个认证信息（例如 `"python.org"`），要么是一个包含认证信息的 URL（如 “” http://python.org/” “ ）。不论是哪种格式，认证信息中都不能包含用户信息（因此，`"python.org"` 和 `"python.org:80"` 没问题，而 `"joe:password@python.org"` 则不行）。

21.6.10 HTTPBasicAuthHandler 对象

`HTTPBasicAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)
如果可用的话，请用身份认证信息重试请求。

21.6.11 ProxyBasicAuthHandler 对象

`ProxyBasicAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)
如果可用的话，请用身份认证信息重试请求。

21.6.12 AbstractDigestAuthHandler 对象

`AbstractDigestAuthHandler.`**`http_error_auth_reqed`**(*authreq*, *host*, *req*, *headers*)
authreq 应为请求中有关 realm 的头部信息名称，*host* 应为需要进行身份认证的主机，*req* 应为（已失败的）`Request` 对象，*headers* 则应为出错的头部信息。

21.6.13 HTTPDigestAuthHandler 对象

`HTTPDigestAuthHandler.`**`http_error_401`**(*req*, *fp*, *code*, *msg*, *hdrs*)
如果可用的话，请用身份认证信息重试请求。

21.6.14 ProxyDigestAuthHandler 对象

`ProxyDigestAuthHandler.`**`http_error_407`**(*req*, *fp*, *code*, *msg*, *hdrs*)
如果可用的话，请用身份认证信息重试请求。

21.6.15 HTTPHandler 对象

`HTTPHandler.`**`http_open`**(*req*)
发送 HTTP 请求，根据 `req.has_data()` 的结果，可能是 GET 或 POST 格式。

21.6.16 HTTPSHandler 对象

HTTPSHandler.**https_open**(*req*)
发送 HTTPS 请求，根据 req.has_data() 的结果，可能是 GET 或 POST 格式。

21.6.17 FileHandler 对象

FileHandler.**file_open**(*req*)
若无主机名或主机名为 'localhost'，则打开本地文件。

在 3.2 版更改: 本方法仅适用于本地主机名。如果给出的是远程主机名，将会触发*URLError*。

21.6.18 DataHandler 对象

DataHandler.**data_open**(*req*)
读取内含数据的 URL。这种 URL 本身包含了经过编码的数据。**RFC 2397** 中给出了数据 URL 的语法定义。目前的代码库将忽略经过 base64 编码的数据 URL 中的空白符，因此 URL 可以放入任何源码文件中。如果数据 URL 的 base64 编码尾部缺少填充，即使某些浏览器不介意，但目前的代码库仍会引发*ValueError*。

21.6.19 FTPHandler 对象

FTPHandler.**ftp_open**(*req*)
打开由 *req* 给出的 FTP 文件。登录时的用户名和密码总是为空。

21.6.20 CacheFTPHandler 对象

CacheFTPHandler 对象即为加入以下方法的*FTPHandler* 对象：

CacheFTPHandler.**setTimeout**(*t*)
设置连接超时为 *t* 秒。

CacheFTPHandler.**setMaxConns**(*m*)
设置已缓存的最大连接数为 *m* 。

21.6.21 UnknownHandler 对象

UnknownHandler.**unknown_open**()
触发*URLError* 异常。

21.6.22 HTTPErrorProcessor 对象

HTTPErrorProcessor.**http_response**(*request*, *response*)
处理出错的 HTTP 响应。

对于 200 错误码，响应对象应立即返回。

For non-200 error codes, this simply passes the job on to the protocol_error_code() handler methods, via *OpenerDirector.error()*. Eventually, *HTTPDefaultErrorHandler* will raise an *HTTPError* if no other handler handles the error.

HTTPErrorProcessor.**https_response**(*request*, *response*)
HTTPS 出错响应的处理。

与*http_response()* 方法相同。

21.6.23 例子

urllib-howto 中给出了更多的示例。

以下示例将读取 python.org 主页并显示前 300 个字节的内容：

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming '
```

请注意，urlopen 将返回字节对象。这是因为 urlopen 无法自动确定由 HTTP 服务器收到的字节流的编码。通常，只要能确定或猜出编码格式，就应将返回的字节对象解码为字符串。

下述 W3C 文档 https://www.w3.org/International/O-charset列出了可用于指明（X）HTML 或 XML 文档编码信息的多种方案。

python.org 网站已在 meta 标签中指明，采用的是 *utf-8* 编码，因此这里将用同样的格式对字节串进行解码。

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

不用*context manager* 方法也能获得同样的结果：

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> print(f.read(100).decode('utf-8'))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

以下示例将会把数据流发送给某 CGI 的 stdin，并读取返回数据。请注意，该示例只能工作于 Python 装有 SSL 支持的环境。

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

上述示例中的 CGI 代码如下所示：

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

下面是利用*Request* 发送 “PUT“ 请求的示例：

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA,method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

基本 HTTP 认证示例：

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

build_opener() 默认提供了很多现成的 handler，包括*ProxyHandler*。默认情况下，*ProxyHandler* 会使用名为 <scheme>_proxy 的环境变量，其中的 <scheme> 是相关的 URL 协议。例如，可以读取 http_proxy 环境变量来获取 HTTP 代理的 URL。

以下示例将默认的*ProxyHandler* 替换为自己的 handler，由程序提供代理 URL，并利用*ProxyBasicAuthHandler* 加入代理认证的支持：

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

添加 HTTP 头部信息：

可利用*Request* 构造函数的 *headers* 参数，或者是：

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib.request.urlopen(req)
```

OpenerDirector 自动会在每个*Request* 中加入一项 *User-Agent* 头部信息。若要修改，请参见以下语句：

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

另请记得，当*Request* 传给*urlopen()* （或*OpenerDirector.open()*）时，会加入一些标准的头部信息（*Content-Length* 、*Content-Type* 和 *Host*）。

以下会话示例用 GET 方法读取包含参数的 URL。

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

以下示例换用 POST 方法。请注意 urlencode 输出结果先被编码为字节串 data，再送入 urlopen。

```
>>> import urllib.request
>>> import urllib.parse
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
...
```

以下示例显式指定了 HTTP 代理，以覆盖环境变量中的设置：

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

以下示例根本不用代理，也覆盖了环境变量中的设置：

```
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

21.6.24 已停用的接口

以下函数和类是由 Python 2 模块 `urllib``（相对早于 ``urllib2`）移植过来的。将来某个时候可能会停用。

`urllib.request.`**`urlretrieve`**(*url*, *filename=None*, *reporthook=None*, *data=None*)

将 URL 形式的网络对象复制为本地文件。如果 URL 指向本地文件，则必须提供文件名才会执行复制。返回值为元组（filename, headers），其中 *filename* 是保存网络对象的本地文件名，*headers* 是由*urlopen()* 返回的远程对象 info() 方法的调用结果。可能触发的异常与*urlopen()* 相同。

第二个参数指定文件的保存位置（若未给出，则会是名称随机生成的临时文件）。第三个参数是个可调用对象，在建立网络连接时将会调用一次，之后每次读完数据块后会调用一次。该可调用对象将会传

入 3 个参数：已传输的块数、块的大小（以字节为单位）和文件总的大小。如果面对的是老旧 FTP 服务器，文件大小参数可能会是 -1，这些服务器响应读取请求时不会返回文件大小。

以下例子演示了大部分常用场景：

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

If the *url* uses the `http:` scheme identifier, the optional *data* argument may be given to specify a `POST` request (normally the request type is `GET`). The *data* argument must be a bytes object in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

urlretrieve() will raise `ContentTooShortError` when it detects that the amount of data available was less than the expected amount (which is the size reported by a *Content-Length* header). This can occur, for example, when the download is interrupted.

The *Content-Length* is treated as a lower bound: if there's more data to read, urlretrieve reads more data, but if less data is available, it raises the exception.

You can still retrieve the downloaded data in this case, it is stored in the `content` attribute of the exception instance.

If no *Content-Length* header was supplied, urlretrieve can not check the size of the data it has downloaded, and just returns it. In this case you just have to assume that the download was successful.

`urllib.request.`**`urlcleanup`**`()`
Cleans up temporary files that may have been left behind by previous calls to *urlretrieve()*.

class `urllib.request.`**`URLopener`**(*proxies=None*, ***x509*)
3.3 版后已移除.

Base class for opening and reading URLs. Unless you need to support opening objects using schemes other than `http:`, `ftp:`, or `file:`, you probably want to use *FancyURLopener*.

By default, the *URLopener* class sends a *User-Agent* header of `urllib/VVV`, where *VVV* is the *urllib* version number. Applications can define their own *User-Agent* header by subclassing *URLopener* or *FancyURLopener* and setting the class attribute *version* to an appropriate string value in the subclass definition.

The optional *proxies* parameter should be a dictionary mapping scheme names to proxy URLs, where an empty dictionary turns proxies off completely. Its default value is `None`, in which case environmental proxy settings will be used if present, as discussed in the definition of *urlopen()*, above.

Additional keyword parameters, collected in *x509*, may be used for authentication of the client when using the `https:` scheme. The keywords *key_file* and *cert_file* are supported to provide an SSL key and certificate; both are needed to support client authentication.

URLopener objects will raise an *OSError* exception if the server returns an error code.

open(*fullurl*, *data=None*)
Open *fullurl* using the appropriate protocol. This method sets up cache and proxy information, then calls the appropriate open method with its input arguments. If the scheme is not recognized, *open_unknown()* is called. The *data* argument has the same meaning as the *data* argument of *urlopen()*.

open_unknown(*fullurl*, *data=None*)
Overridable interface to open unknown URL types.

retrieve(*url*, *filename=None*, *reporthook=None*, *data=None*)
Retrieves the contents of *url* and places it in *filename*. The return value is a tuple consisting of a local filename and either an *email.message.Message* object containing the response headers (for remote URLs) or

None (for local URLs). The caller must then open and read the contents of *filename*. If *filename* is not given and the URL refers to a local file, the input filename is returned. If the URL is non-local and *filename* is not given, the filename is the output of *tempfile.mktemp()* with a suffix that matches the suffix of the last path component of the input URL. If *reporthook* is given, it must be a function accepting three numeric parameters: A chunk number, the maximum size chunks are read in and the total size of the download (-1 if unknown). It will be called once at the start and after each chunk of data is read from the network. *reporthook* is ignored for local URLs.

If the *url* uses the http: scheme identifier, the optional *data* argument may be given to specify a POST request (normally the request type is GET). The *data* argument must in standard *application/x-www-form-urlencoded* format; see the *urllib.parse.urlencode()* function.

version
Variable that specifies the user agent of the opener object. To get *urllib* to tell servers that it is a particular user agent, set this in a subclass as a class variable or in the constructor before calling the base constructor.

class urllib.request.**FancyURLopener**(...)
3.3 版后已移除.

FancyURLopener subclasses *URLopener* providing default handling for the following HTTP response codes: 301, 302, 303, 307 and 401. For the 30x response codes listed above, the *Location* header is used to fetch the actual URL. For 401 response codes (authentication required), basic HTTP authentication is performed. For the 30x response codes, recursion is bounded by the value of the *maxtries* attribute, which defaults to 10.

For all other response codes, the method http_error_default() is called which you can override in subclasses to handle the error appropriately.

注解: According to the letter of **RFC 2616**, 301 and 302 responses to POST requests must not be automatically redirected without confirmation by the user. In reality, browsers do allow automatic redirection of these responses, changing the POST to a GET, and *urllib* reproduces this behaviour.

The parameters to the constructor are the same as those for *URLopener*.

注 解: When performing basic authentication, a *FancyURLopener* instance calls its *prompt_user_passwd()* method. The default implementation asks the users for the required information on the controlling terminal. A subclass may override this method to support more appropriate behavior if needed.

The *FancyURLopener* class offers one additional method that should be overloaded to provide the appropriate behavior:

prompt_user_passwd(*host*, *realm*)
Return information needed to authenticate the user at the given host in the specified security realm. The return value should be a tuple, (user, password), which can be used for basic authentication.

The implementation prompts for this information on the terminal; an application should override this method to use an appropriate interaction model in the local environment.

21.6.25 urllib.request Restrictions

- Currently, only the following protocols are supported: HTTP (versions 0.9 and 1.0), FTP, local files, and data URLs.

  在 3.4 版更改: Added support for data URLs.

- The caching feature of *urlretrieve()* has been disabled until someone finds the time to hack proper processing of Expiration time headers.
- There should be a function to query whether a particular URL is in the cache.
- For backward compatibility, if a URL appears to point to a local file but the file can't be opened, the URL is re-interpreted using the FTP protocol. This can sometimes cause confusing error messages.
- The *urlopen()* and *urlretrieve()* functions can cause arbitrarily long delays while waiting for a network connection to be set up. This means that it is difficult to build an interactive Web client using these functions without using threads.
- The data returned by *urlopen()* or *urlretrieve()* is the raw data returned by the server. This may be binary data (such as an image), plain text or (for example) HTML. The HTTP protocol provides type information in the reply header, which can be inspected by looking at the *Content-Type* header. If the returned data is HTML, you can use the module *html.parser* to parse it.
- The code handling the FTP protocol cannot differentiate between a file and a directory. This can lead to unexpected behavior when attempting to read a URL that points to a file that is not accessible. If the URL ends in a `/`, it is assumed to refer to a directory and will be handled accordingly. But if an attempt to read a file leads to a 550 error (meaning the URL cannot be found or is not accessible, often for permission reasons), then the path is treated as a directory in order to handle the case when a directory is specified by a URL but the trailing `/` has been left off. This can cause misleading results when you try to fetch a file whose read permissions make it inaccessible; the FTP code will try to read it, fail with a 550 error, and then perform a directory listing for the unreadable file. If fine-grained control is needed, consider using the *ftplib* module, subclassing *FancyURLopener*, or changing *_urlopener* to meet your needs.

21.7 urllib.response —urllib 使用的 Response 类

The *urllib.response* module defines functions and classes which define a minimal file like interface, including `read()` and `readline()`. The typical response object is an addinfourl instance, which defines an `info()` method and that returns headers and a `geturl()` method that returns the url. Functions defined by this module are used internally by the *urllib.request* module.

21.8 urllib.parse 用于解析 URL

源代码: Lib/urllib/parse.py

该模块定义了一个标准接口，用于 URL 字符串按组件 (协议、网络位置、路径等) 分解，或将组件组合回 URL 字符串，并将“相对 URL“转换为给定“基础 URL“的绝对 URL。

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: `file`, `ftp`, `gopher`, `hdl`, `http`, `https`, `imap`, `mailto`, `mms`, `news`, `nntp`, `prospero`, `rsync`, `rtsp`, `rtspu`, `sftp`, `shttp`, `sip`, `sips`, `snews`, `svn`, `svn+ssh`, `telnet`, `wais`, `ws`, `wss`.

The *urllib.parse* module defines functions that fall into two broad categories: URL parsing and URL quoting. These are covered in detail in the following sections.

21.8.1 URL 解析

URL 解析功能可以将一个 URL 字符串分割成其组件，或者将 URL 组件组合成一个 URL 字符串。

urllib.parse.**urlparse**(*urlstring*, *scheme=''*, *allow_fragments=True*)

Parse a URL into six components, returning a 6-tuple. This corresponds to the general structure of a URL: `scheme://netloc/path;parameters?query#fragment`. Each tuple item is a string, possibly empty. The components are not broken up in smaller parts (for example, the network location is a single string), and % escapes are not expanded. The delimiters as shown above are not part of the result, except for a leading slash in the *path* component, which is retained if present. For example:

```
>>> from urllib.parse import urlparse
>>> o = urlparse('http://www.cwi.nl:80/%7Eguido/Python.html')
>>> o
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> o.scheme
'http'
>>> o.port
80
>>> o.geturl()
'http://www.cwi.nl:80/%7Eguido/Python.html'
```

Following the syntax specifications in **RFC 1808**, urlparse recognizes a netloc only if it is properly introduced by '//'. Otherwise the input is presumed to be a relative URL and thus to start with a path component.

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

scheme 参数给出了默认的协议，只有在 URL 未指定协议的情况下才会被使用。它应该是与 *urlstring* 相同的类型（文本或字节串），除此之外默认值 `''` 也总是被允许，并会在适当情况下自动转换为 `b''`。

If the *allow_fragments* argument is false, fragment identifiers are not recognized. Instead, they are parsed as part of the path, parameters or query component, and `fragment` is set to the empty string in the return value.

The return value is actually an instance of a subclass of *tuple*. This class has the following additional read-only convenience attributes:

属性	索引	值	值（如果不存在）
scheme	0	URL 方案说明符	*scheme* parameter
netloc	1	网络位置部分	空字符串
path	2	分层路径	空字符串
params	3	最后路径元素的参数	空字符串
query	4	查询组件	空字符串
fragment	5	片段识别	空字符串
username		用户名	*None*
password		密码	*None*
hostname		主机名（小写）	*None*
port		端口号为整数（如果存在）	*None*

如果在 URL 中指定了无效的端口，读取 port 属性将引发*ValueError*。有关结果对象的更多信息请参阅结构化解析结果 一节。

Unmatched square brackets in the netloc attribute will raise a *ValueError*.

Characters in the netloc attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of /, ?, #, @, or : will raise a *ValueError*. If the URL is decomposed before parsing, no error will be raised.

在 3.2 版更改: 添加了 IPv6 URL 解析功能。

在 3.3 版更改: The fragment is now parsed for all URL schemes (unless *allow_fragment* is false), in accordance with **RFC 3986**. Previously, a whitelist of schemes that support fragments existed.

在 3.6 版更改: Out-of-range port numbers now raise *ValueError*, instead of returning *None*.

在 3.6.9 版更改: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

urllib.parse.**parse_qs**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a *ValueError* exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a *ValueError* if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to &.

Use the *urllib.parse.urlencode()* function (with the doseq parameter set to True) to convert such dictionaries into query strings.

在 3.2 版更改: Add *encoding* and *errors* parameters.

在 3.6.8 版更改: Added *max_num_fields* parameter.

在 3.6.13 版更改: Added *separator* parameter with the default value of &. Python versions earlier than Python 3.6.13 allowed using both ; and & as query parameter separator. This has been changed to allow only a single separator key, with & as the default separator.

urllib.parse.**parse_qsl**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

Parse a query string given as a string argument (data of type *application/x-www-form-urlencoded*). Data are returned as a list of name, value pairs.

The optional argument *keep_blank_values* is a flag indicating whether blank values in percent-encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

The optional argument *strict_parsing* is a flag indicating what to do with parsing errors. If false (the default), errors are silently ignored. If true, errors raise a *ValueError* exception.

The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

The optional argument *max_num_fields* is the maximum number of fields to read. If set, then throws a *ValueError* if there are more than *max_num_fields* fields read.

The optional argument *separator* is the symbol to use for separating the query arguments. It defaults to `&`.

Use the *urllib.parse.urlencode()* function to convert such lists of pairs into query strings.

在 3.2 版更改: Add *encoding* and *errors* parameters.

在 3.6.8 版更改: Added *max_num_fields* parameter.

在 3.6.13 版更改: Added *separator* parameter with the default value of `&`. Python versions earlier than Python 3.6.13 allowed using both `;` and `&` as query parameter separator. This has been changed to allow only a single separator key, with `&` as the default separator.

urllib.parse.**urlunparse**(*parts*)

Construct a URL from a tuple as returned by `urlparse()`. The *parts* argument can be any six-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a `?` with an empty query; the RFC states that these are equivalent).

urllib.parse.**urlsplit**(*urlstring*, *scheme=''*, *allow_fragments=True*)

This is similar to *urlparse()*, but does not split the params from the URL. This should generally be used instead of *urlparse()* if the more recent URL syntax allowing parameters to be applied to each segment of the *path* portion of the URL (see **RFC 2396**) is wanted. A separate function is needed to separate the path segments and parameters. This function returns a 5-tuple: (addressing scheme, network location, path, query, fragment identifier).

The return value is actually an instance of a subclass of *tuple*. This class has the following additional read-only convenience attributes:

属性	索引	值	值（如果不存在）
`scheme`	0	URL 方案说明符	*scheme* parameter
`netloc`	1	网络位置部分	空字符串
`path`	2	分层路径	空字符串
`query`	3	查询组件	空字符串
`fragment`	4	片段识别	空字符串
`username`		用户名	*None*
`password`		密码	*None*
`hostname`		主机名（小写）	*None*
`port`		端口号为整数（如果存在）	*None*

如果在 URL 中指定了无效的端口，读取 `port` 属性将引发*ValueError*。有关结果对象的更多信息请参阅结构化解析结果 一节。

Unmatched square brackets in the `netloc` attribute will raise a *ValueError*.

Characters in the `netloc` attribute that decompose under NFKC normalization (as used by the IDNA encoding) into any of `/`, `?`, `#`, `@`, or `:` will raise a *ValueError*. If the URL is decomposed before parsing, no error will be raised.

Following the WHATWG spec that updates RFC 3986, ASCII newline `\n`, `\r` and tab `\t` characters are stripped from the URL.

在 3.6 版更改: Out-of-range port numbers now raise *ValueError*, instead of returning *None*.

在 3.6.9 版更改: Characters that affect netloc parsing under NFKC normalization will now raise *ValueError*.

在 3.6.14 版更改: ASCII newline and tab characters are stripped from the URL.

`urllib.parse.`**`urlunsplit`**(*parts*)

Combine the elements of a tuple as returned by *urlsplit()* into a complete URL as a string. The *parts* argument can be any five-item iterable. This may result in a slightly different, but equivalent URL, if the URL that was parsed originally had unnecessary delimiters (for example, a ? with an empty query; the RFC states that these are equivalent).

`urllib.parse.`**`urljoin`**(*base*, *url*, *allow_fragments=True*)

Construct a full ("absolute") URL by combining a "base URL" (*base*) with another URL (*url*). Informally, this uses components of the base URL, in particular the addressing scheme, the network location and (part of) the path, to provide missing components in the relative URL. For example:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

The *allow_fragments* argument has the same meaning and default as for *urlparse()*.

注解: If *url* is an absolute URL (that is, starting with `//` or `scheme://`), the *url*'s host name and/or scheme will be present in the result. For example:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

If you do not want that behavior, preprocess the *url* with *urlsplit()* and *urlunsplit()*, removing possible *scheme* and *netloc* parts.

在 3.5 版更改: Behaviour updated to match the semantics defined in **RFC 3986**.

`urllib.parse.`**`urldefrag`**(*url*)

If *url* contains a fragment identifier, return a modified version of *url* with no fragment identifier, and the fragment identifier as a separate string. If there is no fragment identifier in *url*, return *url* unmodified and an empty string.

The return value is actually an instance of a subclass of *tuple*. This class has the following additional read-only convenience attributes:

属性	索引	值	值（如果不存在）
`url`	0	URL with no fragment	空字符串
`fragment`	1	片段识别	空字符串

See section 结构化解析结果 for more information on the result object.

在 3.2 版更改: Result is a structured object rather than a simple 2-tuple.

21.8.2 解析 ASCII 编码字节

The URL parsing functions were originally designed to operate on character strings only. In practice, it is useful to be able to manipulate properly quoted and encoded URLs as sequences of ASCII bytes. Accordingly, the URL parsing functions in this module all operate on *bytes* and *bytearray* objects in addition to *str* objects.

If *str* data is passed in, the result will also contain only *str* data. If *bytes* or *bytearray* data is passed in, the result will contain only *bytes* data.

Attempting to mix *str* data with *bytes* or *bytearray* in a single function call will result in a *TypeError* being raised, while attempting to pass in non-ASCII byte values will trigger *UnicodeDecodeError*.

To support easier conversion of result objects between *str* and *bytes*, all return values from URL parsing functions provide either an `encode()` method (when the result contains *str* data) or a `decode()` method (when the result contains *bytes* data). The signatures of these methods match those of the corresponding *str* and *bytes* methods (except that the default encoding is `'ascii'` rather than `'utf-8'`). Each produces a value of a corresponding type that contains either *bytes* data (for `encode()` methods) or *str* data (for `decode()` methods).

Applications that need to operate on potentially improperly quoted URLs that may contain non-ASCII data will need to do their own decoding from bytes to characters before invoking the URL parsing methods.

The behaviour described in this section applies only to the URL parsing functions. The URL quoting functions use their own rules when producing or consuming byte sequences as detailed in the documentation of the individual URL quoting functions.

在 3.2 版更改: URL parsing functions now accept ASCII encoded byte sequences

21.8.3 结构化解析结果

The result objects from the *urlparse()*, *urlsplit()* and *urldefrag()* functions are subclasses of the *tuple* type. These subclasses add the attributes listed in the documentation for those functions, the encoding and decoding support described in the previous section, as well as an additional method:

urllib.parse.SplitResult.**geturl**()

Return the re-combined version of the original URL as a string. This may differ from the original URL in that the scheme may be normalized to lower case and empty components may be dropped. Specifically, empty parameters, queries, and fragment identifiers will be removed.

For *urldefrag()* results, only empty fragment identifiers will be removed. For *urlsplit()* and *urlparse()* results, all noted changes will be made to the URL returned by this method.

The result of this method remains unchanged if passed back through the original parsing function:

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

The following classes provide the implementations of the structured parse results when operating on *str* objects:

class urllib.parse.**DefragResult**(*url*, *fragment*)

Concrete class for *urldefrag()* results containing *str* data. The `encode()` method returns a *DefragResultBytes* instance.

3.2 新版功能.

class urllib.parse.**ParseResult**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)
Concrete class for *urlparse()* results containing *str* data. The encode() method returns a *ParseResultBytes* instance.

class urllib.parse.**SplitResult**(*scheme*, *netloc*, *path*, *query*, *fragment*)
Concrete class for *urlsplit()* results containing *str* data. The encode() method returns a *SplitResultBytes* instance.

The following classes provide the implementations of the parse results when operating on *bytes* or *bytearray* objects:

class urllib.parse.**DefragResultBytes**(*url*, *fragment*)
Concrete class for *urldefrag()* results containing *bytes* data. The decode() method returns a *DefragResult* instance.

3.2 新版功能.

class urllib.parse.**ParseResultBytes**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)
Concrete class for *urlparse()* results containing *bytes* data. The decode() method returns a *ParseResult* instance.

3.2 新版功能.

class urllib.parse.**SplitResultBytes**(*scheme*, *netloc*, *path*, *query*, *fragment*)
Concrete class for *urlsplit()* results containing *bytes* data. The decode() method returns a *SplitResult* instance.

3.2 新版功能.

21.8.4 URL Quoting

The URL quoting functions focus on taking program data and making it safe for use as URL components by quoting special characters and appropriately encoding non-ASCII text. They also support reversing these operations to recreate the original data from the contents of a URL component if that task isn't already covered by the URL parsing functions above.

urllib.parse.**quote**(*string*, *safe='/'*, *encoding=None*, *errors=None*)
Replace special characters in *string* using the %xx escape. Letters, digits, and the characters '_.-' are never quoted. By default, this function is intended for quoting the path section of URL. The optional *safe* parameter specifies additional ASCII characters that should not be quoted —its default value is '/'.

string may be either a *str* or a *bytes*.

The optional *encoding* and *errors* parameters specify how to deal with non-ASCII characters, as accepted by the *str.encode()* method. *encoding* defaults to 'utf-8'. *errors* defaults to 'strict', meaning unsupported characters raise a *UnicodeEncodeError*. *encoding* and *errors* must not be supplied if *string* is a *bytes*, or a *TypeError* is raised.

Note that quote(string, safe, encoding, errors) is equivalent to quote_from_bytes(string.encode(encoding, errors), safe).

Example: quote('/El Niño/') yields '/El%20Ni%C3%B1o/'.

urllib.parse.**quote_plus**(*string*, *safe=''*, *encoding=None*, *errors=None*)
Like *quote()*, but also replace spaces by plus signs, as required for quoting HTML form values when building up a query string to go into a URL. Plus signs in the original string are escaped unless they are included in *safe*. It also does not have *safe* default to '/'.

Example: quote_plus('/El Niño/') yields '%2FEl+Ni%C3%B1o%2F'.

urllib.parse.**quote_from_bytes**(*bytes*, *safe='/'*)
Like *quote()*, but accepts a *bytes* object rather than a *str*, and does not perform string-to-bytes encoding.

Example: `quote_from_bytes(b'a&\xef')` yields `'a%26%EF'`.

urllib.parse.**unquote**(*string*, *encoding='utf-8'*, *errors='replace'*)
Replace `%xx` escapes by their single-character equivalent. The optional *encoding* and *errors* parameters specify how to decode percent-encoded sequences into Unicode characters, as accepted by the *bytes.decode()* method.

string must be a *str*.

encoding defaults to `'utf-8'`. *errors* defaults to `'replace'`, meaning invalid sequences are replaced by a placeholder character.

Example: `unquote('/El%20Ni%C3%B1o/')` yields `'/El Niño/'`.

urllib.parse.**unquote_plus**(*string*, *encoding='utf-8'*, *errors='replace'*)
Like *unquote()*, but also replace plus signs by spaces, as required for unquoting HTML form values.

string must be a *str*.

Example: `unquote_plus('/El+Ni%C3%B1o/')` yields `'/El Niño/'`.

urllib.parse.**unquote_to_bytes**(*string*)
Replace `%xx` escapes by their single-octet equivalent, and return a *bytes* object.

string may be either a *str* or a *bytes*.

If it is a *str*, unescaped non-ASCII characters in *string* are encoded into UTF-8 bytes.

Example: `unquote_to_bytes('a%26%EF')` yields `b'a&\xef'`.

urllib.parse.**urlencode**(*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*)
Convert a mapping object or a sequence of two-element tuples, which may contain *str* or *bytes* objects, to a percent-encoded ASCII text string. If the resultant string is to be used as a *data* for POST operation with the *urlopen()* function, then it should be encoded to bytes, otherwise it would result in a *TypeError*.

The resulting string is a series of `key=value` pairs separated by `'&'` characters, where both *key* and *value* are quoted using the *quote_via* function. By default, *quote_plus()* is used to quote the values, which means spaces are quoted as a `'+'` character and '/' characters are encoded as `%2F`, which follows the standard for GET requests (`application/x-www-form-urlencoded`). An alternate function that can be passed as *quote_via* is *quote()*, which will encode spaces as `%20` and not encode '/' characters. For maximum control of what is quoted, use `quote` and specify a value for *safe*.

When a sequence of two-element tuples is used as the *query* argument, the first element of each tuple is a key and the second is a value. The value element in itself can be a sequence and in that case, if the optional parameter *doseq* is evaluates to `True`, individual `key=value` pairs separated by `'&'` are generated for each element of the value sequence for the key. The order of parameters in the encoded string will match the order of parameter tuples in the sequence.

The *safe*, *encoding*, and *errors* parameters are passed down to *quote_via* (the *encoding* and *errors* parameters are only passed when a query element is a *str*).

To reverse this encoding process, *parse_qs()* and *parse_qsl()* are provided in this module to parse query strings into Python data structures.

Refer to *urllib examples* to find out how urlencode method can be used for generating query string for a URL or data for POST.

在 3.2 版更改: 查询参数支持字节和字符串对象。

3.5 新版功能: *quote_via* 参数.

参见:

WHATWG - URL Living standard Working Group for the URL Standard that defines URLs, domains, IP addresses, the application/x-www-form-urlencoded format, and their API.

RFC 3986 - 统一资源标识符 This is the current standard (STD66). Any changes to urllib.parse module should conform to this. Certain deviations could be observed, which are mostly for backward compatibility purposes and for certain de-facto parsing requirements as commonly observed in major browsers.

RFC 2732 - Format for Literal IPv6 Addresses in URL's. This specifies the parsing requirements of IPv6 URLs.

RFC 2396 - 统一资源标识符（URI）：通用语法 Document describing the generic syntactic requirements for both Uniform Resource Names (URNs) and Uniform Resource Locators (URLs).

RFC 2368 - The mailto URL scheme. Parsing requirements for mailto URL schemes.

RFC 1808 - Relative Uniform Resource Locators This Request For Comments includes the rules for joining an absolute and a relative URL, including a fair number of "Abnormal Examples" which govern the treatment of border cases.

RFC 1738 - Uniform Resource Locators (URL) This specifies the formal syntax and semantics of absolute URLs.

21.9 `urllib.error` —urllib.request 引发的异常类

源代码： Lib/urllib/error.py

---

urllib.error 模块为*urllib.request* 所引发的异常定义了异常类。基础异常类是*URLError*。

下列异常会被*urllib.error* 按需引发：

exception urllib.error.**URLError**
处理程序在遇到问题时会引发此异常（或其派生的异常）。它是*OSError* 的一个子类。

reason
此错误的原因。它可以是一个消息字符串或另一个异常实例。

在 3.3 版更改: *URLError* 已被设为*OSError* 而不是*IOError* 的子类。

exception urllib.error.**HTTPError**
虽然是一个异常 (*URLError* 的一个子类)，*HTTPError* 也可以作为一个非异常的文件类返回值 (与*urlopen()* 返回的对象相同)。这适用于处理特殊 HTTP 错误例如作为认证请求的时候。

code
一个 HTTP 状态码，具体定义见 **RFC 2616**。这个数字值对应于存放在*http.server.BaseHTTPRequestHandler.responses* 代码字典中的某个值。

reason
这通常是一个解释本次错误原因的字符串。

headers
导致了*HTTPError* 的特定 HTTP 请求的 HTTP 响应头。

3.4 新版功能.

exception urllib.error.**ContentTooShortError**(*msg*, *content*)
此异常会在*urlretrieve()* 函数检测到已下载的数据量小于期待的数据量（由 *Content-Length* 头给定）时被引发。content 属性中将存放已下载（可能被截断）的数据。

21.10 urllib.robotparser —robots.txt 语法分析程序

源代码： Lib/urllib/robotparser.py

此模块提供了一个单独的类*RobotFileParser*，它可以回答关于某个特定用户代理是否能在 Web 站点获取发布 robots.txt 文件的 URL 的问题。有关 robots.txt 文件结构的更多细节请参阅 http://www.robotstxt.org/orig.html。

class urllib.robotparser.**RobotFileParser**(*url=''*)

这个类提供了一些可以读取、解析和回答关于 *url* 上的 robots.txt 文件的问题的方法。

set_url(*url*)

设置指向 robots.txt 文件的 URL。

read()

读取 robots.txt URL 并将其输入解析器。

parse(*lines*)

解析行参数。

can_fetch(*useragent*, *url*)

如果允许 *useragent* 按照被解析 robots.txt 文件中的规则来获取 *url* 则返回 True。

mtime()

返回最近一次获取 robots.txt 文件的时间。这适用于需要定期检查 robots.txt 文件更新情况的长时间运行的网页爬虫。

modified()

将最近一次获取 robots.txt 文件的时间设置为当前时间。

crawl_delay(*useragent*)

为指定的 *useragent* 从 robots.txt 返回 Crawl-delay 形参。如果此形参不存在或不适用于指定的 *useragent* 或者此形参的 robots.txt 条目存在语法错误，则返回 None。

3.6 新版功能.

request_rate(*useragent*)

以*named tuple* RequestRate(requests, seconds) 的形式从 robots.txt 返回 Request-rate 形参的内容。如果此形参不存在或不适用于指定的 *useragent* 或者此形参的 robots.txt 条目存在语法错误，则返回 None。

3.6 新版功能.

下面的例子演示了*RobotFileParser* 类的基本用法:

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
```

(下页继续)

(续上页)

```
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

21.11 http —HTTP 模块

源代码: Lib/http/__init__.py

---

http 是一个包，它收集了多个用于处理超文本传输协议的模块:

- *http.client* 是一个低层级的 HTTP 协议客户端；对于高层级的 URL 访问请使用*urllib.request*
- *http.server* 包含基于*socketserver* 的基本 HTTP 服务类
- *http.cookies* 包含一些有用来实现通过 cookies 进行状态管理的工具
- *http.cookiejar* 提供了 cookies 的持久化

http 也是一个通过*http.HTTPStatus* 枚举定义了一些 HTTP 状态码以及相关联消息的模块

class http.**HTTPStatus**

3.5 新版功能.

enum.IntEnum 的子类，它定义了组 HTTP 状态码，原理短语以及用英语书写的长描述文本。

用法:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
<HTTPStatus.OK: 200>
>>> HTTPStatus.OK == 200
True
>>> http.HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[<HTTPStatus.CONTINUE: 100>, <HTTPStatus.SWITCHING_PROTOCOLS: 101>, ...]
```

21.11.1 HTTP 状态码

已支持并且已在*http.HTTPStatus* IANA 注册 的状态码有:

双字母代码	映射名	详情
100	CONTINUE: 继续	HTTP/1.1 RFC 7231, Section 6.2.1
101	SWITCHING_PROTOCOLS	HTTP/1.1 RFC 7231, Section 6.2.2
102	PROCESSING	WebDAV RFC 2518, Section 10.1
200	OK	HTTP/1.1 RFC 7231, Section 6.3.1
201	CREATED	HTTP/1.1 RFC 7231, Section 6.3.2
202	ACCEPTED	HTTP/1.1 RFC 7231, Section 6.3.3
203	NON_AUTHORITATIVE_INFORMATION	HTTP/1.1 RFC 7231, Section 6.3.4

表 1 – 续上页

双字母代码	映射名	详情
204	NO_CONTENT: 没有内容	HTTP/1.1 RFC 7231, Section 6.3.5
205	RESET_CONTENT	HTTP/1.1 RFC 7231, Section 6.3.6
206	PARTIAL_CONTENT	HTTP/1.1 RFC 7233, Section 4.1
207	MULTI_STATUS	WebDAV RFC 4918, Section 11.1
208	ALREADY_REPORTED	WebDAV Binding Extensions RFC 5842, Se
226	IM_USED	Delta Encoding in HTTP RFC 3229, Section
300	MULTIPLE_CHOICES: 有多种资源可选择	HTTP/1.1 RFC 7231, Section 6.4.1
301	MOVED_PERMANENTLY: 永久移动	HTTP/1.1 RFC 7231, Section 6.4.2
302	FOUND: 临时移动	HTTP/1.1 RFC 7231, Section 6.4.3
303	SEE_OTHER: 已经移动	HTTP/1.1 RFC 7231, Section 6.4.4
304	NOT_MODIFIED: 没有修改	HTTP/1.1 RFC 7232, Section 4.1
305	USE_PROXY: 使用代理	HTTP/1.1 RFC 7231, Section 6.4.5
307	TEMPORARY_REDIRECT: 临时重定向	HTTP/1.1 **RFC 7231**, Section 6.4.7
308	PERMANENT_REDIRECT: 永久重定向	Permanent Redirect **RFC 7238**, Section 3 (E
400	BAD_REQUEST: 错误请求	HTTP/1.1 RFC 7231, Section 6.5.1
401	UNAUTHORIZED: 未授权	HTTP/1.1 Authentication RFC 7235, Section
402	PAYMENT_REQUIRED: 保留，将来使用	HTTP/1.1 RFC 7231, Section 6.5.2
403	FORBIDDEN: 禁止	HTTP/1.1 RFC 7231, Section 6.5.3
404	NOT_FOUND: 没有找到	HTTP/1.1 RFC 7231, Section 6.5.4
405	METHOD_NOT_ALLOWED: 该请求方法不允许	HTTP/1.1 RFC 7231, Section 6.5.5
406	NOT_ACCEPTABLE: 不可接受	HTTP/1.1 RFC 7231, Section 6.5.6
407	PROXY_AUTHENTICATION_REQUIRED: 要求使用代理验明正身	HTTP/1.1 Authentication RFC 7235, Section
408	REQUEST_TIMEOUT: 请求超时	HTTP/1.1 RFC 7231, Section 6.5.7
409	CONFLICT: 冲突	HTTP/1.1 RFC 7231, Section 6.5.8
410	GONE: 已经不在了	HTTP/1.1 RFC 7231, Section 6.5.9
411	LENGTH_REQUIRED: 长度要求	HTTP/1.1 RFC 7231, Section 6.5.10
412	PRECONDITION_FAILED: 前提条件错误	HTTP/1.1 RFC 7232, Section 4.2
413	REQUEST_ENTITY_TOO_LARGE: 请求体太大了	HTTP/1.1 RFC 7231, Section 6.5.11
414	REQUEST_URI_TOO_LONG: 请求 URI 太长了	HTTP/1.1 RFC 7231, Section 6.5.12
415	UNSUPPORTED_MEDIA_TYPE: 不支持的媒体格式	HTTP/1.1 RFC 7231, Section 6.5.13
416	REQUEST_RANGE_NOT_SATISFIABLE	HTTP/1.1 Range Requests RFC 7233, Sectio
417	EXPECTATION_FAILED: 期望失败	HTTP/1.1 RFC 7231, Section 6.5.14
422	UNPROCESSABLE_ENTITY: 可加工实体	WebDAV RFC 4918, Section 11.2
423	LOCKED: 锁着	WebDAV RFC 4918, Section 11.3
424	FAILED_DEPENDENCY: 失败的依赖	WebDAV RFC 4918, Section 11.4
426	UPGRADE_REQUIRED: 升级需要	HTTP/1.1 RFC 7231, Section 6.5.15
428	PRECONDITION_REQUIRED: 先决条件要求	Additional HTTP Status Codes RFC 6585
429	TOO_MANY_REQUESTS: 太多的请求	Additional HTTP Status Codes RFC 6585
431	REQUEST_HEADER_FIELDS_TOO_LARGE: 请求头太大	Additional HTTP Status Codes RFC 6585
500	INTERNAL_SERVER_ERROR: 内部服务错误	HTTP/1.1 RFC 7231, Section 6.6.1
501	NOT_IMPLEMENTED: 不可执行	HTTP/1.1 RFC 7231, Section 6.6.2
502	BAD_GATEWAY: 无效网关	HTTP/1.1 RFC 7231, Section 6.6.3
503	SERVICE_UNAVAILABLE: 服务不可用	HTTP/1.1 RFC 7231, Section 6.6.4
504	GATEWAY_TIMEOUT: 网关超时	HTTP/1.1 RFC 7231, Section 6.6.5
505	HTTP_VERSION_NOT_SUPPORTED: HTTP 版本不支持	HTTP/1.1 RFC 7231, Section 6.6.6
506	VARIANT_ALSO_NEGOTIATES: 服务器存在内部配置错误	透明内容协商在: HTTP **RFC 2295**, Sectio
507	INSUFFICIENT_STORAGE: 存储不足	WebDAV RFC 4918, Section 11.5
508	LOOP_DETECTED: 循环检测	WebDAV Binding Extensions RFC 5842, Se
510	NOT_EXTENDED: 不扩展	An HTTP Extension Framework **RFC 2774**,

表 1 – 续上页

双字母代码	映射名	详情
511	NETWORK_AUTHENTICATION_REQUIRED：要求网络身份验证	Additional HTTP Status Codes RFC 6585, S

为了保持向后兼容性，枚举值也以常量形式出现在*http.client* 模块中，。枚举名等于常量名 (例如 `http.HTTPStatus.OK` 也可以是 `http.client.OK`)。

21.12 http.client —HTTP 协议客户端

源代码： Lib/http/client.py

这个模块定义了实现 HTTP 和 HTTPS 协议客户端的类。它通常不直接使用—模块*urllib.request* 用它来处理使用 HTTP 和 HTTPS 的 URL。

参见：

The Requests package is recommended for a higher-level HTTP client interface.

注解： HTTPS 支持仅在编译 Python 时启用了 SSL 支持的情况下（通过*ssl* 模块）可用。

该模块支持以下类：

class http.client.**HTTPConnection**(*host*, *port=None*[, *timeout*], *source_address=None*)

An *HTTPConnection* instance represents one transaction with an HTTP server. It should be instantiated passing it a host and optional port number. If no port number is passed, the port is extracted from the host string if it has the form `host:port`, else the default HTTP port (80) is used. If the optional *timeout* parameter is given, blocking operations (like connection attempts) will timeout after that many seconds (if it is not given, the global default timeout setting is used). The optional *source_address* parameter may be a tuple of a (host, port) to use as the source address the HTTP connection is made from.

举个例子，以下调用都是创建连接到同一主机和端口的服务器的实例：

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

在 3.2 版更改: 添加了 *source_address* 参数

在 3.4 版更改: 删除了 *strict* 参数，不再支持 HTTP 0.9 风格的“简单响应”。

class http.client.**HTTPSConnection**(*host*, *port=None*, *key_file=None*, *cert_file=None*[, *timeout*], *source_address=None*, *, *context=None*, *check_hostname=None*)

HTTPConnection 的子类，使用 SSL 与安全服务器进行通信。默认端口为 `443`。如果指定了 *context*，它必须为一个描述 SSL 各选项的*ssl.SSLContext* 实例。

请参阅安全考量 了解有关最佳实践的更多信息。

在 3.2 版更改: 添加了 *source_address*, *context* 和 *check_hostname*。

在 3.2 版更改: 这个类目前会在可能的情况下（即如果*ssl.HAS_SNI* 为真值）支持 HTTPS 虚拟主机。

在 3.4 版更改: 删除了 *strict* 参数，不再支持 HTTP 0.9 风格的“简单响应”。

在 3.4.3 版更改: 目前这个类在默认情况下会执行所有必要的证书和主机检查。要回复到先前的非验证行为，可以将 `ssl._create_unverified_context()` 传递给 *context* 参数。

3.6 版后已移除: *key_file* 和 *cert_file* 已弃用并转而推荐 *context*。请改用*ssl.SSLContext.load_cert_chain()* 或让*ssl.create_default_context()* 为你选择系统所信任的 CA 证书。

check_hostname 参数也已弃用；应当改用 *context* 的*ssl.SSLContext.check_hostname* 属性。

class http.client.**HTTPResponse**(*sock*, *debuglevel=0*, *method=None*, *url=None*)
在成功连接后返回类的实例，而不是由用户直接实例化。

在 3.4 版更改: 删除了 *strict* 参数，不再支持 HTTP 0.9 风格的 “简单响应”。

下列异常可以适当地被引发:

exception http.client.**HTTPException**
此模块中其他异常的基类。它是*Exception* 的一个子类。

exception http.client.**NotConnected**
HTTPException 的一个子类。

exception http.client.**InvalidURL**
HTTPException 的一个子类，如果给出了一个非数字或为空值的端口就会被引发。

exception http.client.**UnknownProtocol**
HTTPException 的一个子类。

exception http.client.**UnknownTransferEncoding**
HTTPException 的一个子类。

exception http.client.**UnimplementedFileMode**
HTTPException 的一个子类。

exception http.client.**IncompleteRead**
HTTPException 的一个子类。

exception http.client.**ImproperConnectionState**
HTTPException 的一个子类。

exception http.client.**CannotSendRequest**
ImproperConnectionState 的一个子类。

exception http.client.**CannotSendHeader**
ImproperConnectionState 的一个子类。

exception http.client.**ResponseNotReady**
ImproperConnectionState 的一个子类。

exception http.client.**BadStatusLine**
HTTPException 的一个子类。如果服务器反馈了一个我们不理解的 HTTP 状态码就会被引发。

exception http.client.**LineTooLong**
HTTPException 的一个子类。如果在 HTTP 协议中从服务器接收到过长的行就会被引发。

exception http.client.**RemoteDisconnected**
ConnectionResetError 和*BadStatusLine* 的一个子类。当尝试读取响应时的结果是未从连接读取到数据时由*HTTPConnection.getresponse()* 引发，表明远端已关闭连接。

3.5 新版功能: 在此之前引发的异常为*BadStatusLine*`('')`。

此模块中定义的常量为:

http.client.**HTTP_PORT**
HTTP 协议默认的端口号 (总是 `80`)。

http.client.**HTTPS_PORT**
HTTPS 协议默认的端口号 (总是 443)。

http.client.**responses**
这个字典把 HTTP 1.1 状态码映射到 W3C 名称。

例如: http.client.responses[http.client.NOT_FOUND] 是 'NOT FOUND （未发现）。

本模块中可用的 HTTP 状态码常量可以参见*HTTP 状态码* 。

21.12.1 HTTPConnection 对象

HTTPConnection 实例拥有以下方法:

HTTPConnection.**request**(*method*, *url*, *body=None*, *headers={}*, ***, *encode_chunked=False*)
这会使用 HTTP 请求方法 *method* 和选择器 *url* 向服务器发送请求。

如果给定 *body*，那么给定的数据会在信息头完成之后发送。它可能是一个*str* 、一个*bytes-like object* 、一个打开的*file object*，或者*bytes* 迭代器。如果 *body* 是字符串，它会按 HTTP 默认的 ISO-8859-1 编码；如果是一个字节类对象，它会按原样发送；如果是*file object* ，文件的内容会被发送，这个文件对象应该支持 read() 方法。如果这个文件对象是一个*io.TextIOBase* 实例，read() 方法返回的数据会按 ISO-8859-1 编码，否则 read() 方法返回的数据会按原样发送；如果 *body* 是一个迭代器，迭代器中的元素会被发送，直到迭代器耗尽。

headers 参数应是额外的随请求发送的 HTTP 信息头的字典。

如果 *headers* 既不包含 Content-Length 也没有 Transfer-Encoding，但存在请求正文，那么这些头字段中的一个会自动设定。如果 *body* 是 None，那么对于要求正文的方法 (PUT，POST，和 PATCH)，Content-Length 头会被设为 0。如果 *body* 是字符串或者类似字节的对象，并且也不是文件，Content-Length 头会设为正文的长度。任何其他类型的 *body* （一般是文件或迭代器）会按块编码，这时会自动设定 Transfer-Encoding 头以代替 Content-Length。

在 *headers* 中指定 Transfer-Encoding 时，*encode_chunked* 是唯一相关的参数。如果 *encode_chunked* 为 False，HTTPConnection 对象会假定所有的编码都由调用代码处理。如果为 True，正文会按块编码。

注解: HTTP 协议在 1.1 版中添加了块传输编码。除非明确知道 HTTP 服务器可以处理 HTTP 1.1，调用者要么必须指定 Content-Length，要么必须传入*str* 或字节类对象，注意该对象不能是表达 body 的文件。

3.2 新版功能: *body* 现在可以是可迭代对象了。

在 3.6 版更改: 如果 Content-Length 和 Transfer-Encoding 都没有在 *headers* 中设置，文件和可迭代的 *body* 对象现在会按块编码。添加了 *encode_chunked* 参数。不会尝试去确定文件对象的 Content-Length。

HTTPConnection.**getresponse**()
应当在发送一个请求从服务器获取响应时被调用。返回一个*HTTPResponse* 的实例。

注解: 请注意你必须在读取了整个响应之后才能向服务器发送新的请求。

在 3.5 版更改: 如果引发了*ConnectionError* 或其子类，*HTTPConnection* 对象将在发送新的请求时准备好重新连接。

HTTPConnection.**set_debuglevel**(*level*)
设置调试等级。默认的调试等级为 0，意味着不会打印调试输出。任何大于 0 的值将使得所有当前定义的调试输出被打印到 stdout。debuglevel 会被传给任何新创建的*HTTPResponse* 对象。

3.1 新版功能.

HTTPConnection.**set_tunnel**(*host*, *port=None*, *headers=None*)
为 HTTP 连接隧道设置主机和端口。这将允许通过代理服务器运行连接。

host 和 port 参数指明隧道连接的位置（即 CONNECT 请求所包含的地址，而 不是代理服务器的地址）。

headers 参数应为一个随 CONNECT 请求发送的额外 HTTP 标头的映射。

例如，要通过一个运行于本机 8080 端口的 HTTPS 代理服务器隧道，我们应当向*HTTPSConnection* 构造器传入代理的地址，并将我们最终想要访问的主机地址传给*set_tunnel()* 方法:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

3.2 新版功能.

HTTPConnection.**connect**()
当对象被创建后连接到指定的服务器。默认情况下，如果客户端还未建立连接，此函数会在发送请求时自动被调用。

HTTPConnection.**close**()
关闭到服务器的连接。

作为对使用上述 request() 方法的替代同，你也可以通过使用下面的四个函数，分步骤发送请的请求。

HTTPConnection.**putrequest**(*method*, *url*, *skip_host=False*, *skip_accept_encoding=False*)
This should be the first call after the connection to the server has been made. It sends a line to the server consisting of the *method* string, the *url* string, and the HTTP version (HTTP/1.1). To disable automatic sending of Host: or Accept-Encoding: headers (for example to accept additional content encodings), specify *skip_host* or *skip_accept_encoding* with non-False values.

HTTPConnection.**putheader**(*header*, *argument*[, ...])
Send an **RFC 822**-style header to the server. It sends a line to the server consisting of the header, a colon and a space, and the first argument. If more arguments are given, continuation lines are sent, each consisting of a tab and an argument.

HTTPConnection.**endheaders**(*message_body=None*, ***, *encode_chunked=False*)
Send a blank line to the server, signalling the end of the headers. The optional *message_body* argument can be used to pass a message body associated with the request.

If *encode_chunked* is True, the result of each iteration of *message_body* will be chunk-encoded as specified in **RFC 7230**, Section 3.3.1. How the data is encoded is dependent on the type of *message_body*. If *message_body* implements the buffer interface the encoding will result in a single chunk. If *message_body* is a collections.Iterable, each iteration of *message_body* will result in a chunk. If *message_body* is a *file object*, each call to .read() will result in a chunk. The method automatically signals the end of the chunk-encoded data immediately after *message_body*.

注解： Due to the chunked encoding specification, empty chunks yielded by an iterator body will be ignored by the chunk-encoder. This is to avoid premature termination of the read of the request by the target server due to malformed encoding.

3.6 新版功能: Chunked encoding support. The *encode_chunked* parameter was added.

HTTPConnection.**send**(*data*)
Send data to the server. This should be used directly only after the *endheaders()* method has been called and before *getresponse()* is called.

21.12.2 HTTPResponse 对象

An *HTTPResponse* instance wraps the HTTP response from the server. It provides access to the request headers and the entity body. The response is an iterable object and can be used in a with statement.

在 3.5 版更改: The *io.BufferedIOBase* interface is now implemented and all of its reader operations are supported.

HTTPResponse.**read**([*amt*])
Reads and returns the response body, or up to the next *amt* bytes.

HTTPResponse.**readinto**(*b*)
Reads up to the next len(b) bytes of the response body into the buffer *b*. Returns the number of bytes read.

3.3 新版功能.

HTTPResponse.**getheader**(*name*, *default=None*)
Return the value of the header *name*, or *default* if there is no header matching *name*. If there is more than one header with the name *name*, return all of the values joined by ‘, ‘. If ‘default’ is any iterable other than a single string, its elements are similarly returned joined by commas.

HTTPResponse.**getheaders**()
Return a list of (header, value) tuples.

HTTPResponse.**fileno**()
Return the `fileno` of the underlying socket.

HTTPResponse.**msg**
A `http.client.HTTPMessage` instance containing the response headers. `http.client.HTTPMessage` is a subclass of *email.message.Message*.

HTTPResponse.**version**
HTTP protocol version used by server. 10 for HTTP/1.0, 11 for HTTP/1.1.

HTTPResponse.**status**
由服务器返回的状态码。

HTTPResponse.**reason**
Reason phrase returned by server.

HTTPResponse.**debuglevel**
A debugging hook. If *debuglevel* is greater than zero, messages will be printed to stdout as the response is read and parsed.

HTTPResponse.**closed**
Is `True` if the stream is closed.

21.12.3 例子

Here is an example session that uses the `GET` method:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
```

(下页继续)

(续上页)

```
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while not r1.closed:
...     print(r1.read(200))  # 200 bytes
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

Here is an example session that uses the `HEAD` method. Note that the `HEAD` method never returns any data.

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

Here is an example session that shows how to `POST` requests:

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action':
↪'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/
↪issue12524</a>'
>>> conn.close()
```

Client side `HTTP PUT` requests are very similar to `POST` requests. The difference lies only the server side where HTTP server will allow resources to be created via `PUT` request. It should be noted that custom HTTP methods +are also handled in *`urllib.request.Request`* by sending the appropriate +method attribute.Here is an example session that shows how to do `PUT` request using http.client:

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
```

(下页继续)

(续上页)

```
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

21.12.4 HTTPMessage Objects

An `http.client.HTTPMessage` instance holds the headers from an HTTP response. It is implemented using the *email.message.Message* class.

21.13 ftplib —FTP 协议客户端

源代码： Lib/ftplib.py

本模块定义了*FTP* 类和一些相关项目。*FTP* 类实现了 FTP 协议的客户端。可以用该类编写 Python 程序，执行各种自动化的 FTP 任务，如镜像其他 FTP 服务器。*urllib.request* 模块也用它来处理使用了 FTP 的 URL。关于 FTP（文件传输协议）的详情请参阅 Internet **RFC 959**。

以下是使用*ftplib* 模块的会话示例:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.debian.org')     # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> ftp.retrbinary('RETR README', open('README', 'wb').write)
'226 Transfer complete.'
>>> ftp.quit()
```

这个模块定义了以下内容:

class ftplib.**FTP**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*)

Return a new instance of the *FTP* class. When *host* is given, the method call `connect(host)` is made. When *user* is given, additionally the method call `login(user, passwd, acct)` is made (where *passwd* and *acct* default to the empty string when not given). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if is not specified, the global default timeout setting will be used). *source_address* is a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting.

FTP 类支持 `with` 语句，例如:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

在 3.2 版更改: 支持了 with 语句。

在 3.3 版更改: 添加了 *source_address* 参数。

class ftplib.**FTP_TLS**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *keyfile=None*, *certfile=None*, *context=None*, *timeout=None*, *source_address=None*)

一个*FTP* 的子类，它为 FTP 添加了 TLS 支持，如 **RFC 4217** 所述。它将像通常一样连接到 21 端口，暗中保护在身份验证前的 FTP 控制连接。而保护数据连接需要用户明确调用*prot_p()* 方法。*context* 是一个*ssl.SSLContext* 对象，该对象可以将 SSL 配置选项、证书和私钥打包放入一个单独的（可以长久存在的）结构中。请阅读*安全考量* 以获取最佳实践。

keyfile 和 *certfile* 是可以代替 *context* 的传统方案，它们可以分别指向 PEM 格式的私钥和证书链文件，用于进行 SSL 连接。

3.2 新版功能.

在 3.3 版更改: 添加了 *source_address* 参数。

在 3.4 版更改: 本类现在支持通过*ssl.SSLContext.check_hostname* 和服务器名称提示（参阅*ssl.HAS_SNI*）进行主机名检查。

3.6 版后已移除: *keyfile* 和 *certfile* 已弃用并转而推荐 *context*。请改用*ssl.SSLContext.load_cert_chain()* 或让*ssl.create_default_context()* 为你选择系统所信任的 CA 证书。

以下是使用*FTP_TLS* 类的会话示例:

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-jedi
↪', 'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu', 'ignore',
↪'libpuzzle', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-user-variables',
↪'php-jenkins-hash', 'php-skein-hash', 'php-webdav', 'phpaudit', 'phpbench',
↪'pincaster', 'ping', 'posto', 'pub', 'public', 'public_keys', 'pure-ftpd',
↪'qscan', 'qtc', 'sharedance', 'skycache', 'sound', 'tmp', 'ucarp']
```

exception ftplib.**error_reply**

从服务器收到意外答复时，将引发本异常。

exception ftplib.**error_temp**

收到表示临时错误的错误代码（响应代码在 400–499 范围内）时，将引发本异常。

exception ftplib.**error_perm**

收到表示永久性错误的错误代码（响应代码在 500–599 范围内）时，将引发本异常。

exception ftplib.**error_proto**
从服务器收到不符合 FTP 响应规范的答复，比如以数字 1–5 开头时，将引发本异常。

ftplib.**all_errors**
The set of all exceptions (as a tuple) that methods of *FTP* instances may raise as a result of problems with the FTP connection (as opposed to programming errors made by the caller). This set includes the four exceptions listed above as well as *OSError*.

参见:

***netrc* 模块** .netrc 文件格式解析器。FTP 客户端在响应用户之前，通常使用 .netrc 文件来加载用户认证信息。

21.13.1 FTP 对象

一些方法可以按照两种方式来使用：一种处理文本文件，另一种处理二进制文件。方法名称与相应的命令相同，文本版中命令后面跟着 lines，二进制版中命令后面跟着 binary。

FTP 实例具有下列方法:

FTP.**set_debuglevel**(*level*)
设置实例的调试级别，它控制着调试信息的数量。默认值 0 不产生调试信息。值 1 产生中等数量的调试信息，通常每个请求产生一行。大于或等于 2 的值产生的调试信息最多，FTP 控制连接上发送和接收的每一行都将被记录下来。

FTP.**connect**(*host=''*, *port=0*, *timeout=None*, *source_address=None*)
连接到给定的主机和端口。默认端口号由 FTP 协议规范规定，为 21。偶尔才需要指定其他端口号。每个实例只应调用一次本函数，如果在创建实例时就传入了 host，则根本不应调用它。所有其他方法只能在建立连接后使用。可选参数 *timeout* 指定连接尝试的超时（以秒为单位）。如果没有传入 *timeout*，将使用全局默认超时设置。*source_address* 是一个 2 元组 (host, port)，套接字在连接前绑定它，作为其源地址。

在 3.3 版更改: 添加了 *source_address* 参数。

FTP.**getwelcome**()
返回服务器发送的欢迎消息，作为连接开始的回复。(该消息有时包含与用户有关的免责声明或帮助信息。)

FTP.**login**(*user='anonymous'*, *passwd=''*, *acct=''*)
以 *user* 的身份登录。*passwd* 和 *acct* 是可选参数，默认为空字符串。如果没有指定 *user*，则默认为 'anonymous'。如果 *user* 为 'anonymous'，那么默认的 *passwd* 是 'anonymous@'。连接建立后，每个实例只应调用一次本函数；如果在创建实例时传入了 host 和 user，则完全不应该调用本函数。在客户端登录后，才允许执行大多数 FTP 命令。*acct* 参数提供记账信息 (“accounting information”)；仅少数系统实现了该特性。

FTP.**abort**()
中止正在进行的文件传输。本方法并不总是有效，但值得一试。

FTP.**sendcmd**(*cmd*)
将一条简单的命令字符串发送到服务器，返回响应的字符串。

FTP.**voidcmd**(*cmd*)
将一条简单的命令字符串发送到服务器，并处理响应内容。如果收到的响应代码表示的是成功（代码范围 200–299），则不返回任何内容。否则将引发*error_reply*。

FTP.**retrbinary**(*cmd*, *callback*, *blocksize=8192*, *rest=None*)
以二进制传输模式下载文件。*cmd* 应为恰当的 RETR 命令：'RETR 文件名'。*callback* 函数会在收到每个数据块时调用，传入的参数是表示数据块的一个字节。为执行实际传输，创建了底层套接字对象，可

选参数 *blocksize* 指定了读取该对象时的最大块大小（这也是传入 *callback* 的数据块的最大大小）。已经选择了合理的默认值。*rest* 的含义与*transfercmd()* 方法中的含义相同。

FTP.**retrlines**(*cmd*, *callback=None*)
Retrieve a file or directory listing in ASCII transfer mode. *cmd* should be an appropriate RETR command (see *retrbinary()*) or a command such as LIST or NLST (usually just the string 'LIST'). LIST retrieves a list of files and information about those files. NLST retrieves a list of file names. The *callback* function is called for each line with a string argument containing the line with the trailing CRLF stripped. The default *callback* prints the line to sys.stdout.

FTP.**set_pasv**(*val*)
如果 *val* 为 true，则打开“被动”模式，否则禁用被动模式。默认下被动模式是打开的。

FTP.**storbinary**(*cmd*, *fp*, *blocksize=8192*, *callback=None*, *rest=None*)
以二进制传输模式存储文件。*cmd* 应为恰当的 STOR 命令: "STOR filename"。*fp* 是一个文件对象 (以二进制模式打开)，将使用它的 read() 方法读取它，用于提供要存储的数据，直到遇到 EOF，读取时的块大小为 *blocksize*。参数 *blocksize* 的默认值为 8192。可选参数 *callback* 是单参数函数，在每个数据块发送后都会用该数据块调用它。*rest* 的含义与*transfercmd()* 方法中的含义相同。

在 3.2 版更改: 添加了 *rest* 参数。

FTP.**storlines**(*cmd*, *fp*, *callback=None*)
Store a file in ASCII transfer mode. *cmd* should be an appropriate STOR command (see *storbinary()*). Lines are read until EOF from the *file object* *fp* (opened in binary mode) using its *readline()* method to provide the data to be stored. *callback* is an optional single parameter callable that is called on each line after it is sent.

FTP.**transfercmd**(*cmd*, *rest=None*)
在 FTP 数据连接上开始传输数据。如果传输处于活动状态，传输命令由 *cmd* 指定，需发送 EPRT 或 PORT 命令，然后接受连接 (accept)。如果服务器是被动服务器，需发送 EPSV 或 PASV 命令，连接到服务器 (connect)，然后启动传输命令。两种方式都将返回用于连接的套接字。

If optional *rest* is given, a REST command is sent to the server, passing *rest* as an argument. *rest* is usually a byte offset into the requested file, telling the server to restart sending the file’s bytes at the requested offset, skipping over the initial bytes. Note however that **RFC 959** requires only that *rest* be a string containing characters in the printable range from ASCII code 33 to ASCII code 126. The *transfercmd()* method, therefore, converts *rest* to a string, but no check is performed on the string’s contents. If the server does not recognize the REST command, an *error_reply* exception will be raised. If this happens, simply call *transfercmd()* without a *rest* argument.

FTP.**ntransfercmd**(*cmd*, *rest=None*)
类似于*transfercmd()*，但返回一个元组，包括数据连接和数据的预计大小。如果预计大小无法计算，则返回的预计大小为 None。*cmd* 和 *rest* 的含义与*transfercmd()* 中的相同。

FTP.**mlsd**(*path=""*, *facts=[]*)
使用 MLSD 命令以标准格式列出目录内容 (**RFC 3659**)。如果省略 *path* 则使用当前目录。*facts* 是字符串列表，表示所需的信息类型（如 ["type", "size", "perm"]）。返回一个生成器对象，每个在 path 中找到的文件都将在该对象中生成两个元素的元组。第一个元素是文件名，第二个元素是该文件的 facts 的字典。该字典的内容受 *facts* 参数限制，但不能保证服务器会返回所有请求的 facts。

3.3 新版功能.

FTP.**nlst**(*argument*[, ...])
返回一个文件名列表，文件名由 NLST 命令返回。可选参数 *argument* 是待列出的目录（默认为当前服务器目录）。可以使用多个参数，将非标准选项传递给 NLST 命令。

注解: 如果目标服务器支持相关命令，那么*mlsd()* 提供的 API 更好。

FTP.**dir**(*argument*[, ...])
生成目录列表，即 LIST 命令所返回的结果，并将其打印到标准输出。可选参数 *argument* 是待列出的目录（默认为当前服务器目录）。可以使用多个参数，将非标准选项传递给 LIST 命令。如果最后一个参数是一个函数，它将被用作 *callback* 函数，与*retrlines()* 中的相同，默认将打印到 sys.stdout。本方法返回 None。

注解： 如果目标服务器支持相关命令，那么*mlsd()* 提供的 API 更好。

FTP.**rename**(*fromname*, *toname*)
将服务器上的文件 *fromname* 重命名为 *toname*。

FTP.**delete**(*filename*)
将服务器上名为 *filename* 的文件删除。如果删除成功，返回响应文本，如果删除失败，在权限错误时引发*error_perm*，在其他错误时引发*error_reply*。

FTP.**cwd**(*pathname*)
设置服务器端的当前目录。

FTP.**mkd**(*pathname*)
在服务器上创建一个新目录。

FTP.**pwd**()
返回服务器上当前目录的路径。

FTP.**rmd**(*dirname*)
将服务器上名为 *dirname* 的目录删除。

FTP.**size**(*filename*)
请求服务器上名为 *filename* 的文件大小。成功后以整数返回文件大小，未成功则返回 None。注意，SIZE 不是标准命令，但通常许多服务器的实现都支持该命令。

FTP.**quit**()
向服务器发送 QUIT 命令并关闭连接。这是关闭一个连接的“礼貌”方式，但是如果服务器对 QUIT 命令的响应带有错误消息则这会引发一个异常。这意味着对*close()* 方法的调用，它将使得*FTP* 实例对后继调用无效（见下文）。

FTP.**close**()
单方面关闭连接。这不该被应用于已经关闭的连接，例如成功调用*quit()* 之后的连接。在此调用之后*FTP* 实例不应被继续使用（在调用*close()* 或*quit()* 之后你不能通过再次发起调用*login()* 方法重新打开连接）。

21.13.2 FTP_TLS 对象

FTP_TLS 类继承自*FTP*，它定义了下述其他对象：

FTP_TLS.**ssl_version**
欲采用的 SSL 版本（默认为*ssl.PROTOCOL_SSLv23*）。

FTP_TLS.**auth**()
通过使用 TLS 或 SSL 来设置一个安全控制连接，具体取决于*ssl_version* 属性是如何设置的。

在 3.4 版更改: 此方法现在支持使用*ssl.SSLContext.check_hostname* 和 服务器名称指示 (参见*ssl.HAS_SNI*) 进行主机名检查。

FTP_TLS.**ccc**()
将控制通道回复为纯文本。这适用于发挥知道如何使用非安全 FTP 处理 NAT 而无需打开固定端口的防火墙的优势。

3.3 新版功能.

FTP_TLS.**prot_p**()
设置加密数据连接。

FTP_TLS.**prot_c**()
设置明文数据连接。

21.14 poplib —POP3 协议客户端

源代码： Lib/poplib.py

本模块定义了一个*POP3* 类，该类封装了到 POP3 服务器的连接过程，并实现了 **RFC 1939** 中定义的协议。*POP3* 类同时支持 **RFC 1939** 中最小的和可选的命令集。*POP3* 类还支持在 **RFC 2595** 中引入的 STLS 命令，用于在已建立的连接上启用加密通信。

本模块额外提供一个*POP3_SSL* 类，在连接到 POP3 服务器时，该类为使用 SSL 作为底层协议层提供了支持。

注意，尽管 POP3 具有广泛的支持，但它已经过时。POP3 服务器的实现质量差异很大，而且大多很糟糕。如果邮件服务器支持 IMAP，则最好使用*imaplib.IMAP4* 类，因为 IMAP 服务器一般实现得更好。

poplib 模块提供了两个类：

class poplib.**POP3**(*host*, *port=POP3_PORT*[, *timeout*])
本类实现实际的 POP3 协议。实例初始化时，连接就会建立。如果省略 *port*，则使用标准 POP3 端口（110）。可选参数 *timeout* 指定连接尝试的超时时间（以秒为单位，如果未指定超时，将使用全局默认超时设置）。

class poplib.**POP3_SSL**(*host*, *port=POP3_SSL_PORT*, *keyfile=None*, *certfile=None*, *timeout=None*, *context=None*)
一个*POP3* 的子类，它使用经 SSL 加密的套接字连接到服务器。如果端口 *port* 未指定，则使用 995，它是标准的 POP3-over-SSL 端口。*timeout* 的作用与*POP3* 构造函数中的相同。*context* 是一个可选的*ssl.SSLContext* 对象，该对象可以将 SSL 配置选项、证书和私钥打包放入一个单独的（可以长久存在的）结构中。请阅读安全考量 以获取最佳实践。

keyfile 和 *certfile* 是可以代替 *context* 的传统方案，它们可以分别指向 PEM 格式的私钥和证书链文件，用于进行 SSL 连接。

在 3.2 版更改: 添加了 *context* 参数。

在 3.4 版更改: 本类现在支持通过*ssl.SSLContext.check_hostname* 和服务器名称提示（参阅*ssl.HAS_SNI*）进行主机名检查。

3.6 版后已移除: *keyfile* 和 *certfile* 已弃用并转而推荐 *context*。请改用*ssl.SSLContext.load_cert_chain()* 或让*ssl.create_default_context()* 为你选择系统所信任的 CA 证书。

定义了一个异常，它是作为*poplib* 模块的属性定义的：

exception poplib.**error_proto**
此模块的所有错误都将引发本异常（来自*socket* 模块的错误不会被捕获）。异常的原因将以字符串的形式传递给构造函数。

参见：

***imaplib* 模块** 标准的 Python IMAP 模块。

有关 Fetchmail 的常见问题 **fetchmail** POP/IMAP 客户端的“常见问题”收集了 POP3 服务器之间的差异和 RFC 不兼容的信息，如果要编写基于 POP 协议的应用程序，这可能会很有用。

21.14.1 POP3 对象

所有 POP3 命令均以同名的方法表示，小写，大多数方法返回的是服务器发送的响应文本。

POP3 实例具有下列方法：

POP3.**set_debuglevel**(*level*)
设置实例的调试级别，它控制着调试信息的数量。默认值 0 不产生调试信息。值 1 产生中等数量的调试信息，通常每个请求产生一行。大于或等于 2 的值产生的调试信息最多，FTP 控制连接上发送和接收的每一行都将被记录下来。

POP3.**getwelcome**()
返回 POP3 服务器发送的问候语字符串。

POP3.**capa**()
查询服务器支持的功能，这些功能在 **RFC 2449** 中有说明。返回一个 `{'name': ['param'...]}` 形式的字典。

3.4 新版功能.

POP3.**user**(*username*)
发送 user 命令，返回的响应应该指示需要一个密码。

POP3.**pass_**(*password*)
发送密码，响应包括邮件数和邮箱大小。注意：在调用 `quit()` 前，服务器上的邮箱都是锁定的。

POP3.**apop**(*user*, *secret*)
使用更安全的 APOP 身份验证登录到 POP3 服务器。

POP3.**rpop**(*user*)
使用 RPOP 身份验证（类似于 Unix r-命令）登录到 POP3 服务器。

POP3.**stat**()
获取邮箱状态。结果为 2 个整数组成的元组：`(message count, mailbox size)`。

POP3.**list**([*which*])
请求消息列表，结果的形式为 `(response, ['mesg_num octets', ...], octets)`。如果设置了 *which*，它表示需要列出的消息。

POP3.**retr**(*which*)
检索编号为 *which* 的整条消息，并设置其已读标志位。结果的形式为 `(response, ['line', ...], octets)`。

POP3.**dele**(*which*)
将编号为 *which* 的消息标记为待删除。在多数服务器上，删除操作直到 QUIT 才会实际执行（主要例外是 Eudora QPOP，它在断开连接时执行删除，故意违反了 RFC）。

POP3.**rset**()
移除邮箱中的所有待删除标记。

POP3.**noop**()
什么都不做。可以用于保持活动状态。

POP3.**quit**()
登出：提交更改，解除邮箱锁定，断开连接。

POP3.**top**(*which*, *howmuch*)
检索编号为 *which* 的消息，范围是消息头加上消息头往后数 *howmuch* 行。结果的形式为 `(response, ['line', ...], octets)`。

本方法使用 POP3 TOP 命令，不同于 RETR 命令，它不设置邮件的已读标志位。不幸的是，TOP 在 RFC 中说明不清晰，且在小众的服务器软件中往往不可用。信任并使用它之前，请先手动对目标 POP3 服务器测试本方法。

POP3.**uidl**(*which=None*)
返回消息摘要（唯一 ID）列表。如果指定了 *which*，那么结果将包含那条消息的唯一 ID，形式为 `'response mesgnum uid`，如果未指定，那么结果为列表 `(response, ['mesgnum uid', ...], octets)`。

POP3.**utf8**()
尝试切换至 UTF-8 模式。成功则返回服务器的响应，失败则引发*error_proto* 异常。在 **RFC 6856** 中有说明。

3.5 新版功能.

POP3.**stls**(*context=None*)
在活动连接上开启 TLS 会话，在 **RFC 2595** 中有说明。仅在用户身份验证前允许这样做。

context 参数是一个*ssl.SSLContext* 对象，该对象可以将 SSL 配置选项、证书和私钥打包放入一个单独的（可以长久存在的）结构中。请阅读*安全考量* 以获取最佳实践。

此方法支持通过*ssl.SSLContext.check_hostname* 和 服务器名称指示 (参见*ssl.HAS_SNI*) 进行主机名检查。

3.4 新版功能.

POP3_SSL 实例没有额外方法。该子类的接口与其父类的相同。

21.14.2 POP3 示例

以下是一个最短示例（不带错误检查），该示例将打开邮箱，检索并打印所有消息:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

模块的最后有一段测试，其中包含的用法示例更加广泛。

21.15 imaplib —IMAP4 协议客户端

源代码： Lib/imaplib.py

本模块定义了三个类：*IMAP4* 、*IMAP4_SSL* 和*IMAP4_stream* 。这三个类封装了与 IMAP4 服务器的连接并实现了 **RFC 2060** 当中定义的大多数 IMAP4rev1 客户端协议。其与 IMAP4（**RFC 1730** ）服务器后向兼容，但是 STATUS 指令在 IMAP4 中不支持。

imaplib 模块提供了三个类，其中*IMAP4* 是基类:

class imaplib.**IMAP4**(*host=''*, *port=IMAP4_PORT*)
This class implements the actual IMAP4 protocol. The connection is created and protocol version (IMAP4 or IMAP4rev1) is determined when the instance is initialized. If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4 port (143) is used.

The *IMAP4* class supports the `with` statement. When used like this, the IMAP4 `LOGOUT` command is issued automatically when the `with` statement exits. E.g.:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

在 3.5 版更改: 支持了 `with` 语句。

Three exceptions are defined as attributes of the *IMAP4* class:

exception `IMAP4.`**error**
Exception raised on any errors. The reason for the exception is passed to the constructor as a string.

exception `IMAP4.`**abort**
IMAP4 server errors cause this exception to be raised. This is a sub-class of *IMAP4.error*. Note that closing the instance and instantiating a new one will usually allow recovery from this exception.

exception `IMAP4.`**readonly**
This exception is raised when a writable mailbox has its status changed by the server. This is a sub-class of *IMAP4.error*. Some other client now has write permission, and the mailbox will need to be re-opened to re-obtain write permission.

There's also a subclass for secure connections:

class `imaplib.`**IMAP4_SSL**(*host=''*, *port=IMAP4_SSL_PORT*, *keyfile=None*, *certfile=None*, *ssl_context=None*)
This is a subclass derived from *IMAP4* that connects over an SSL encrypted socket (to use this class you need a socket module that was compiled with SSL support). If *host* is not specified, `''` (the local host) is used. If *port* is omitted, the standard IMAP4-over-SSL port (993) is used. *ssl_context* is a *ssl.SSLContext* object which allows bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure. Please read *安全考量* for best practices.

keyfile and *certfile* are a legacy alternative to *ssl_context* - they can point to PEM-formatted private key and certificate chain files for the SSL connection. Note that the *keyfile*/*certfile* parameters are mutually exclusive with *ssl_context*, a *ValueError* is raised if *keyfile*/*certfile* is provided along with *ssl_context*.

在 3.3 版更改: *ssl_context* parameter added.

在 3.4 版更改: 本类现在支持通过*ssl.SSLContext.check_hostname* 和 服务器名称提示（参阅*ssl.HAS_SNI*）进行主机名检查。

3.6 版后已移除: *keyfile* and *certfile* are deprecated in favor of *ssl_context*. Please use *ssl.SSLContext.load_cert_chain()* instead, or let *ssl.create_default_context()* select the system's trusted CA certificates for you.

The second subclass allows for connections created by a child process:

class `imaplib.`**IMAP4_stream**(*command*)
This is a subclass derived from *IMAP4* that connects to the `stdin/stdout` file descriptors created by passing *command* to `subprocess.Popen()`.

The following utility functions are defined:

`imaplib.`**Internaldate2tuple**(*datestr*)
Parse an IMAP4 `INTERNALDATE` string and return corresponding local time. The return value is a *time.struct_time* tuple or `None` if the string has wrong format.

`imaplib.`**Int2AP**(*num*)
Converts an integer into a string representation using characters from the set [`A` .. `P`].

imaplib.**ParseFlags**(*flagstr*)
Converts an IMAP4 FLAGS response to a tuple of individual flags.

imaplib.**Time2Internaldate**(*date_time*)
Convert *date_time* to an IMAP4 INTERNALDATE representation. The return value is a string in the form: "DD-Mmm-YYYY HH:MM:SS +HHMM" (including double-quotes). The *date_time* argument can be a number (int or float) representing seconds since epoch (as returned by *time.time()*), a 9-tuple representing local time an instance of *time.struct_time* (as returned by *time.localtime()*), an aware instance of *datetime.datetime*, or a double-quoted string. In the last case, it is assumed to already be in the correct format.

Note that IMAP4 message numbers change as the mailbox changes; in particular, after an EXPUNGE command performs deletions the remaining messages are renumbered. So it is highly advisable to use UIDs instead, with the UID command.

模块的最后有一段测试，其中包含的用法示例更加广泛。

参见:

Documents describing the protocol, and sources and binaries for servers implementing it, can all be found at the University of Washington's *IMAP Information Center* (https://www.washington.edu/imap/).

21.15.1 IMAP4 Objects

All IMAP4rev1 commands are represented by methods of the same name, either upper-case or lower-case.

All arguments to commands are converted to strings, except for AUTHENTICATE, and the last argument to APPEND which is passed as an IMAP4 literal. If necessary (the string contains IMAP4 protocol-sensitive characters and isn't enclosed with either parentheses or double quotes) each string is quoted. However, the *password* argument to the LOGIN command is always quoted. If you want to avoid having an argument string quoted (eg: the *flags* argument to STORE) then enclose the string in parentheses (eg: r'(\Deleted)').

Each command returns a tuple: (type, [data, ...]) where *type* is usually 'OK' or 'NO', and *data* is either the text from the command response, or mandated results from the command. Each *data* is either a string, or a tuple. If a tuple, then the first part is the header of the response, and the second part contains the data (ie: 'literal' value).

The *message_set* options to commands below is a string specifying one or more messages to be acted upon. It may be a simple message number ('1'), a range of message numbers ('2:4'), or a group of non-contiguous ranges separated by commas ('1:3,6:9'). A range can contain an asterisk to indicate an infinite upper bound ('3:*').

An *IMAP4* instance has the following methods:

IMAP4.**append**(*mailbox*, *flags*, *date_time*, *message*)
Append *message* to named mailbox.

IMAP4.**authenticate**(*mechanism*, *authobject*)
Authenticate command —requires response processing.

mechanism specifies which authentication mechanism is to be used - it should appear in the instance variable capabilities in the form AUTH=mechanism.

authobject must be a callable object:

```
data = authobject(response)
```

It will be called to process server continuation responses; the *response* argument it is passed will be bytes. It should return bytes *data* that will be base64 encoded and sent to the server. It should return None if the client abort response * should be sent instead.

在 3.5 版更改: string usernames and passwords are now encoded to utf-8 instead of being limited to ASCII.

IMAP4.**check**()
Checkpoint mailbox on server.

IMAP4.**close**()
Close currently selected mailbox. Deleted messages are removed from writable mailbox. This is the recommended command before LOGOUT.

IMAP4.**copy**(*message_set*, *new_mailbox*)
Copy *message_set* messages onto end of *new_mailbox*.

IMAP4.**create**(*mailbox*)
Create new mailbox named *mailbox*.

IMAP4.**delete**(*mailbox*)
Delete old mailbox named *mailbox*.

IMAP4.**deleteacl**(*mailbox*, *who*)
Delete the ACLs (remove any rights) set for who on mailbox.

IMAP4.**enable**(*capability*)
Enable *capability* (see **RFC 5161**). Most capabilities do not need to be enabled. Currently only the UTF8=ACCEPT capability is supported (see **RFC 6855**).

3.5 新版功能: The *enable()* method itself, and **RFC 6855** support.

IMAP4.**expunge**()
Permanently remove deleted items from selected mailbox. Generates an EXPUNGE response for each deleted message. Returned data contains a list of EXPUNGE message numbers in order received.

IMAP4.**fetch**(*message_set*, *message_parts*)
Fetch (parts of) messages. *message_parts* should be a string of message part names enclosed within parentheses, eg: "(UID BODY[TEXT])". Returned data are tuples of message part envelope and data.

IMAP4.**getacl**(*mailbox*)
Get the ACLs for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**getannotation**(*mailbox*, *entry*, *attribute*)
Retrieve the specified ANNOTATIONs for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**getquota**(*root*)
Get the quota *root*'s resource usage and limits. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**getquotaroot**(*mailbox*)
Get the list of quota roots for the named *mailbox*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**list**([*directory*[, *pattern*]])
List mailbox names in *directory* matching *pattern*. *directory* defaults to the top-level mail folder, and *pattern* defaults to match anything. Returned data contains a list of LIST responses.

IMAP4.**login**(*user*, *password*)
Identify the client using a plaintext password. The *password* will be quoted.

IMAP4.**login_cram_md5**(*user*, *password*)
Force use of CRAM-MD5 authentication when identifying the client to protect the password. Will only work if the server CAPABILITY response includes the phrase AUTH=CRAM-MD5.

IMAP4.**logout**()
Shutdown connection to server. Returns server BYE response.

IMAP4.**lsub**(*directory=""*, *pattern='*'*)
List subscribed mailbox names in directory matching pattern. *directory* defaults to the top level directory and *pattern* defaults to match any mailbox. Returned data are tuples of message part envelope and data.

IMAP4.**myrights**(*mailbox*)
Show my ACLs for a mailbox (i.e. the rights that I have on mailbox).

IMAP4.**namespace**()
Returns IMAP namespaces as defined in **RFC 2342**.

IMAP4.**noop**()
Send NOOP to server.

IMAP4.**open**(*host*, *port*)
Opens socket to *port* at *host*. This method is implicitly called by the *IMAP4* constructor. The connection objects established by this method will be used in the *IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()*, and *IMAP4.shutdown()* methods. You may override this method.

IMAP4.**partial**(*message_num*, *message_part*, *start*, *length*)
Fetch truncated part of a message. Returned data is a tuple of message part envelope and data.

IMAP4.**proxyauth**(*user*)
Assume authentication as *user*. Allows an authorised administrator to proxy into any user's mailbox.

IMAP4.**read**(*size*)
Reads *size* bytes from the remote server. You may override this method.

IMAP4.**readline**()
Reads one line from the remote server. You may override this method.

IMAP4.**recent**()
Prompt server for an update. Returned data is None if no new messages, else value of RECENT response.

IMAP4.**rename**(*oldmailbox*, *newmailbox*)
Rename mailbox named *oldmailbox* to *newmailbox*.

IMAP4.**response**(*code*)
Return data for response *code* if received, or None. Returns the given code, instead of the usual type.

IMAP4.**search**(*charset*, *criterion*[, ...])
Search mailbox for matching messages. *charset* may be None, in which case no CHARSET will be specified in the request to the server. The IMAP protocol requires that at least one criterion be specified; an exception will be raised when the server returns an error. *charset* must be None if the UTF8=ACCEPT capability was enabled using the *enable()* command.

示例:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select**(*mailbox='INBOX'*, *readonly=False*)
Select a mailbox. Returned data is the count of messages in *mailbox* (EXISTS response). The default *mailbox* is 'INBOX'. If the *readonly* flag is set, modifications to the mailbox are not allowed.

IMAP4.**send**(*data*)
Sends data to the remote server. You may override this method.

IMAP4.**setacl**(*mailbox*, *who*, *what*)
Set an ACL for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**setannotation**(*mailbox*, *entry*, *attribute*[, ...])
Set ANNOTATIONs for *mailbox*. The method is non-standard, but is supported by the Cyrus server.

IMAP4.**setquota**(*root*, *limits*)
Set the quota *root*'s resource *limits*. This method is part of the IMAP4 QUOTA extension defined in rfc2087.

IMAP4.**shutdown**()
Close connection established in open. This method is implicitly called by *IMAP4.logout()*. You may override this method.

IMAP4.**socket**()
Returns socket instance used to connect to server.

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion*[, ...])
The sort command is a variant of search with sorting semantics for the results. Returned data contains a space separated list of matching message numbers.

Sort has two arguments before the *search_criterion* argument(s); a parenthesized list of *sort_criteria*, and the searching *charset*. Note that unlike search, the searching *charset* argument is mandatory. There is also a uid sort command which corresponds to sort the way that uid search corresponds to search. The sort command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the numbers of matching messages.

This is an IMAP4rev1 extension command.

IMAP4.**starttls**(*ssl_context=None*)
Send a STARTTLS command. The *ssl_context* argument is optional and should be a *ssl.SSLContext* object. This will enable encryption on the IMAP connection. Please read 安全考量 for best practices.

3.2 新版功能.

在 3.4 版更改: 此方法现在支持使用*ssl.SSLContext.check_hostname* 和 服务器名称指示 (参见*ssl.HAS_SNI*) 进行主机名检查。

IMAP4.**status**(*mailbox*, *names*)
Request named status conditions for *mailbox*.

IMAP4.**store**(*message_set*, *command*, *flag_list*)
Alters flag dispositions for messages in mailbox. *command* is specified by section 6.4.6 of **RFC 2060** as being one of "FLAGS", "+FLAGS", or "-FLAGS", optionally with a suffix of ".SILENT".

For example, to set the delete flag on all messages:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

注解: Creating flags containing ']' (for example: "[test]") violates **RFC 3501** (the IMAP protocol). However, imaplib has historically allowed creation of such tags, and popular IMAP servers, such as Gmail, accept and produce such flags. There are non-Python programs which also create such tags. Although it is an RFC violation and IMAP clients and servers are supposed to be strict, imaplib nonetheless continues to allow such tags to be created for backward compatibility reasons, and as of python 3.6, handles them if they are sent from the server, since this improves real-world compatibility.

IMAP4.**subscribe**(*mailbox*)
Subscribe to new mailbox.

IMAP4.**thread**(*threading_algorithm*, *charset*, *search_criterion*[, ...])
The thread command is a variant of search with threading semantics for the results. Returned data contains a space separated list of thread members.

Thread members consist of zero or more messages numbers, delimited by spaces, indicating successive parent and child.

Thread has two arguments before the *search_criterion* argument(s); a *threading_algorithm*, and the searching *charset*. Note that unlike `search`, the searching *charset* argument is mandatory. There is also a `uid thread` command which corresponds to `thread` the way that `uid search` corresponds to `search`. The `thread` command first searches the mailbox for messages that match the given searching criteria using the charset argument for the interpretation of strings in the searching criteria. It then returns the matching messages threaded according to the specified threading algorithm.

This is an `IMAP4rev1` extension command.

`IMAP4.`**`uid`**(*command*, *arg*[, ...])
Execute command args with messages identified by UID, rather than message number. Returns response appropriate to command. At least one argument must be supplied; if none are provided, the server will return an error and an exception will be raised.

`IMAP4.`**`unsubscribe`**(*mailbox*)
Unsubscribe from old mailbox.

`IMAP4.`**`xatom`**(*name*[, ...])
Allow simple extension commands notified by server in `CAPABILITY` response.

The following attributes are defined on instances of *IMAP4*:

`IMAP4.`**`PROTOCOL_VERSION`**
The most recent supported protocol in the `CAPABILITY` response from the server.

`IMAP4.`**`debug`**
Integer value to control debugging output. The initialize value is taken from the module variable `Debug`. Values greater than three trace each command.

`IMAP4.`**`utf8_enabled`**
Boolean value that is normally `False`, but is set to `True` if an *enable()* command is successfully issued for the `UTF8=ACCEPT` capability.

3.5 新版功能.

21.15.2 IMAP4 Example

以下是一个最短示例（不带错误检查），该示例将打开邮箱，检索并打印所有消息:

```
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

21.16 nntplib —NNTP protocol client

Source code: Lib/nntplib.py

This module defines the class *NNTP* which implements the client side of the Network News Transfer Protocol. It can be used to implement a news reader or poster, or automated news processors. It is compatible with **RFC 3977** as well as the older **RFC 977** and **RFC 2980**.

Here are two small examples of how it can be used. To list some statistics about a newsgroup and print the subjects of the last 10 articles:

```
>>> s = nntplib.NNTP('news.gmane.org')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print('Group', name, 'has', count, 'articles, range', first, 'to', last)
Group gmane.comp.python.committers has 1096 articles, range 1 to 1096
>>> resp, overviews = s.over((last - 9, last))
>>> for id, over in overviews:
...     print(id, nntplib.decode_header(over['subject']))
...
1087 Re: Commit privileges for Łukasz Langa
1088 Re: 3.2 alpha 2 freeze
1089 Re: 3.2 alpha 2 freeze
1090 Re: Commit privileges for Łukasz Langa
1091 Re: Commit privileges for Łukasz Langa
1092 Updated ssh key
1093 Re: Updated ssh key
1094 Re: Updated ssh key
1095 Hello fellow committers!
1096 Re: Hello fellow committers!
>>> s.quit()
'205 Bye!'
```

To post an article from a binary file (this assumes that the article has valid headers, and that you have right to post on the particular newsgroup):

```
>>> s = nntplib.NNTP('news.gmane.org')
>>> f = open('article.txt', 'rb')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'
```

The module itself defines the following classes:

class nntplib.**NNTP**(*host*, *port=119*, *user=None*, *password=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

Return a new *NNTP* object, representing a connection to the NNTP server running on host *host*, listening at port *port*. An optional *timeout* can be specified for the socket connection. If the optional *user* and *password* are provided, or if suitable credentials are present in `/.netrc` and the optional flag *usenetrc* is true, the `AUTHINFO USER` and `AUTHINFO PASS` commands are used to identify and authenticate the user to the server. If the optional flag *readermode* is true, then a `mode reader` command is sent before authentication is performed. Reader mode is sometimes necessary if you are connecting to an NNTP server on the local machine and intend to call reader-specific commands, such as `group`. If you get unexpected *NNTPPermanentError*s, you might need to set *readermode*. The *NNTP* class supports the `with` statement to unconditionally consume *OSError* exceptions and to close the NNTP connection when done, e.g.:

```
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.org') as n:
...     n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.comp.
↪python.committers')
>>>
```

在 3.2 版更改: *usenetrc* is now `False` by default.

在 3.3 版更改: 支持了 `with` 语句。

class nntplib.**NNTP_SSL**(*host*, *port=563*, *user=None*, *password=None*, *ssl_context=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

Return a new *NNTP_SSL* object, representing an encrypted connection to the NNTP server running on host *host*, listening at port *port*. *NNTP_SSL* objects have the same methods as *NNTP* objects. If *port* is omitted, port 563 (NNTPS) is used. *ssl_context* is also optional, and is a *SSLContext* object. Please read 安全考量 for best practices. All other parameters behave the same as for *NNTP*.

Note that SSL-on-563 is discouraged per **RFC 4642**, in favor of STARTTLS as described below. However, some servers only support the former.

3.2 新版功能.

在 3.4 版更改: 本类现在支持通过*ssl.SSLContext.check_hostname* 和服务器名称提示（参阅*ssl.HAS_SNI*）进行主机名检查。

exception nntplib.**NNTPError**

Derived from the standard exception *Exception*, this is the base class for all exceptions raised by the *nntplib* module. Instances of this class have the following attribute:

response

The response of the server if available, as a *str* object.

exception nntplib.**NNTPReplyError**

从服务器收到意外答复时，将引发本异常。

exception nntplib.**NNTPTemporaryError**

Exception raised when a response code in the range 400–499 is received.

exception nntplib.**NNTPPermanentError**

Exception raised when a response code in the range 500–599 is received.

exception nntplib.**NNTPProtocolError**

Exception raised when a reply is received from the server that does not begin with a digit in the range 1–5.

exception nntplib.**NNTPDataError**

Exception raised when there is some error in the response data.

21.16.1 NNTP Objects

When connected, *NNTP* and *NNTP_SSL* objects support the following methods and attributes.

Attributes

NNTP.**nntp_version**
An integer representing the version of the NNTP protocol supported by the server. In practice, this should be 2 for servers advertising **RFC 3977** compliance and 1 for others.

3.2 新版功能.

NNTP.**nntp_implementation**
A string describing the software name and version of the NNTP server, or *None* if not advertised by the server.

3.2 新版功能.

方法

The *response* that is returned as the first item in the return tuple of almost all methods is the server's response: a string beginning with a three-digit code. If the server's response indicates an error, the method raises one of the above exceptions.

Many of the following methods take an optional keyword-only argument *file*. When the *file* argument is supplied, it must be either a *file object* opened for binary writing, or the name of an on-disk file to be written to. The method will then write any data returned by the server (except for the response line and the terminating dot) to the file; any list of lines, tuples or objects that the method normally returns will be empty.

在 3.2 版更改: Many of the following methods have been reworked and fixed, which makes them incompatible with their 3.1 counterparts.

NNTP.**quit**()
Send a QUIT command and close the connection. Once this method has been called, no other methods of the NNTP object should be called.

NNTP.**getwelcome**()
返回服务器发送的欢迎消息，作为连接开始的回复。(该消息有时包含与用户有关的免责声明或帮助信息。)

NNTP.**getcapabilities**()
Return the **RFC 3977** capabilities advertised by the server, as a *dict* instance mapping capability names to (possibly empty) lists of values. On legacy servers which don't understand the CAPABILITIES command, an empty dictionary is returned instead.

```
>>> s = NNTP('news.gmane.org')
>>> 'POST' in s.getcapabilities()
True
```

3.2 新版功能.

NNTP.**login**(*user=None*, *password=None*, *usenetrc=True*)
Send AUTHINFO commands with the user name and password. If *user* and *password* are None and *usenetrc* is true, credentials from ~/.netrc will be used if possible.

Unless intentionally delayed, login is normally performed during the *NNTP* object initialization and separately calling this function is unnecessary. To force authentication to be delayed, you must not set *user* or *password* when creating the object, and must set *usenetrc* to False.

3.2 新版功能.

NNTP.**starttls**(*context=None*)
Send a STARTTLS command. This will enable encryption on the NNTP connection. The *context* argument is optional and should be a *ssl.SSLContext* object. Please read 安全考量 for best practices.

Note that this may not be done after authentication information has been transmitted, and authentication occurs by default if possible during a *NNTP* object initialization. See *NNTP.login()* for information on suppressing this behavior.

3.2 新版功能.

在 3.4 版更改: 此方法现在支持使用*ssl.SSLContext.check_hostname* 和 服务器名称指示 (参见*ssl.HAS_SNI*) 进行主机名检查。

NNTP.**newgroups**(*date*, *, *file=None*)

Send a NEWGROUPS command. The *date* argument should be a *datetime.date* or *datetime.datetime* object. Return a pair (response, groups) where *groups* is a list representing the groups that are new since the given *date*. If *file* is supplied, though, then *groups* will be empty.

```
>>> from datetime import date, timedelta
>>> resp, groups = s.newgroups(date.today() - timedelta(days=3))
>>> len(groups)
85
>>> groups[0]
GroupInfo(group='gmane.network.tor.devel', last='4', first='1', flag='m')
```

NNTP.**newnews**(*group*, *date*, *, *file=None*)

Send a NEWNEWS command. Here, *group* is a group name or '*', and *date* has the same meaning as for *newgroups()*. Return a pair (response, articles) where *articles* is a list of message ids.

This command is frequently disabled by NNTP server administrators.

NNTP.**list**(*group_pattern=None*, *, *file=None*)

Send a LIST or LIST ACTIVE command. Return a pair (response, list) where *list* is a list of tuples representing all the groups available from this NNTP server, optionally matching the pattern string *group_pattern*. Each tuple has the form (group, last, first, flag), where *group* is a group name, *last* and *first* are the last and first article numbers, and *flag* usually takes one of these values:

- y: Local postings and articles from peers are allowed.
- m: The group is moderated and all postings must be approved.
- n: No local postings are allowed, only articles from peers.
- j: Articles from peers are filed in the junk group instead.
- x: No local postings, and articles from peers are ignored.
- =foo.bar: Articles are filed in the foo.bar group instead.

If *flag* has another value, then the status of the newsgroup should be considered unknown.

This command can return very large results, especially if *group_pattern* is not specified. It is best to cache the results offline unless you really need to refresh them.

在 3.2 版更改: *group_pattern* was added.

NNTP.**descriptions**(*grouppattern*)

Send a LIST NEWSGROUPS command, where *grouppattern* is a wildmat string as specified in **RFC 3977** (it's essentially the same as DOS or UNIX shell wildcard strings). Return a pair (response, descriptions), where *descriptions* is a dictionary mapping group names to textual descriptions.

```
>>> resp, descs = s.descriptions('gmane.comp.python.*')
>>> len(descs)
295
>>> descs.popitem()
('gmane.comp.python.bio.general', 'BioPython discussion list (Moderated)')
```

NNTP.**description**(*group*)
Get a description for a single group *group*. If more than one group matches (if 'group' is a real wildmat string), return the first match. If no group matches, return an empty string.

This elides the response code from the server. If the response code is needed, use *descriptions()*.

NNTP.**group**(*name*)
Send a GROUP command, where *name* is the group name. The group is selected as the current group, if it exists. Return a tuple (response, count, first, last, name) where *count* is the (estimated) number of articles in the group, *first* is the first article number in the group, *last* is the last article number in the group, and *name* is the group name.

NNTP.**over**(*message_spec*, *, *file=None*)
Send an OVER command, or an XOVER command on legacy servers. *message_spec* can be either a string representing a message id, or a (first, last) tuple of numbers indicating a range of articles in the current group, or a (first, None) tuple indicating a range of articles starting from *first* to the last article in the current group, or *None* to select the current article in the current group.

Return a pair (response, overviews). *overviews* is a list of (article_number, overview) tuples, one for each article selected by *message_spec*. Each *overview* is a dictionary with the same number of items, but this number depends on the server. These items are either message headers (the key is then the lower-cased header name) or metadata items (the key is then the metadata name prepended with ":"). The following items are guaranteed to be present by the NNTP specification:

- the subject, from, date, message-id and references headers
- the :bytes metadata: the number of bytes in the entire raw article (including headers and body)
- the :lines metadata: the number of lines in the article body

The value of each item is either a string, or *None* if not present.

It is advisable to use the *decode_header()* function on header values when they may contain non-ASCII characters:

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, overviews = s.over((last, last))
>>> art_num, over = overviews[0]
>>> art_num
117216
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references', 'date', 'message-id', 'subject
↪']
>>> over['from']
'=?UTF-8?B?Ik1hcnRpbiB2LiBMw7Z3aXMi?= <martin@v.loewis.de>'
>>> nntplib.decode_header(over['from'])
'"Martin v. Löwis" <martin@v.loewis.de>'
```

3.2 新版功能.

NNTP.**help**(*, *file=None*)
Send a HELP command. Return a pair (response, list) where *list* is a list of help strings.

NNTP.**stat**(*message_spec=None*)
Send a STAT command, where *message_spec* is either a message id (enclosed in '<' and '>') or an article number in the current group. If *message_spec* is omitted or *None*, the current article in the current group is considered. Return a triple (response, number, id) where *number* is the article number and *id* is the message id.

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, number, message_id = s.stat(first)
```

(下页继续)

(续上页)

```
>>> number, message_id
(9099, '<20030112190404.GE29873@epoch.metaslash.com>')
```

NNTP.**next**()
Send a NEXT command. Return as for *stat()*.

NNTP.**last**()
Send a LAST command. Return as for *stat()*.

NNTP.**article**(*message_spec=None*, *, *file=None*)
Send an ARTICLE command, where *message_spec* has the same meaning as for *stat()*. Return a tuple `(response, info)` where *info* is a *namedtuple* with three attributes *number*, *message_id* and *lines* (in that order). *number* is the article number in the group (or 0 if the information is not available), *message_id* the message id as a string, and *lines* a list of lines (without terminating newlines) comprising the raw message including headers and body.

```
>>> resp, info = s.article('<20030112190404.GE29873@epoch.metaslash.com>')
>>> info.number
0
>>> info.message_id
'<20030112190404.GE29873@epoch.metaslash.com>'
>>> len(info.lines)
65
>>> info.lines[0]
b'Path: main.gmane.org!not-for-mail'
>>> info.lines[1]
b'From: Neal Norwitz <neal@metaslash.com>'
>>> info.lines[-3:]
[b'There is a patch for 2.3 as well as 2.2.', b'', b'Neal']
```

NNTP.**head**(*message_spec=None*, *, *file=None*)
Same as *article()*, but sends a HEAD command. The *lines* returned (or written to *file*) will only contain the message headers, not the body.

NNTP.**body**(*message_spec=None*, *, *file=None*)
Same as *article()*, but sends a BODY command. The *lines* returned (or written to *file*) will only contain the message body, not the headers.

NNTP.**post**(*data*)
Post an article using the POST command. The *data* argument is either a *file object* opened for binary reading, or any iterable of bytes objects (representing raw lines of the article to be posted). It should represent a well-formed news article, including the required headers. The *post()* method automatically escapes lines beginning with `.` and appends the termination line.

If the method succeeds, the server's response is returned. If the server refuses posting, a *NNTPReplyError* is raised.

NNTP.**ihave**(*message_id*, *data*)
Send an IHAVE command. *message_id* is the id of the message to send to the server (enclosed in `'<'` and `'>'`). The *data* parameter and the return value are the same as for *post()*.

NNTP.**date**()
Return a pair `(response, date)`. *date* is a *datetime* object containing the current date and time of the server.

NNTP.**slave**()
Send a SLAVE command. Return the server's *response*.

NNTP.**set_debuglevel**(*level*)
Set the instance's debugging level. This controls the amount of debugging output printed. The default, `0`, produces no debugging output. A value of `1` produces a moderate amount of debugging output, generally a single line per request or response. A value of `2` or higher produces the maximum amount of debugging output, logging each line sent and received on the connection (including message text).

The following are optional NNTP extensions defined in **RFC 2980**. Some of them have been superseded by newer commands in **RFC 3977**.

NNTP.**xhdr**(*hdr*, *str*, *, *file=None*)
Send an `XHDR` command. The *hdr* argument is a header keyword, e.g. `'subject'`. The *str* argument should have the form `'first-last'` where *first* and *last* are the first and last article numbers to search. Return a pair `(response, list)`, where *list* is a list of pairs `(id, text)`, where *id* is an article number (as a string) and *text* is the text of the requested header for that article. If the *file* parameter is supplied, then the output of the `XHDR` command is stored in a file. If *file* is a string, then the method will open a file with that name, write to it then close it. If *file* is a *file object*, then it will start calling `write()` on it to store the lines of the command output. If *file* is supplied, then the returned *list* is an empty list.

NNTP.**xover**(*start*, *end*, *, *file=None*)
Send an `XOVER` command. *start* and *end* are article numbers delimiting the range of articles to select. The return value is the same of for *over()*. It is recommended to use *over()* instead, since it will automatically use the newer `OVER` command if available.

NNTP.**xpath**(*id*)
Return a pair `(resp, path)`, where *path* is the directory path to the article with message ID *id*. Most of the time, this extension is not enabled by NNTP server administrators.

3.3 版后已移除: The XPATH extension is not actively used.

21.16.2 工具函数

The module also defines the following utility function:

nntplib.**decode_header**(*header_str*)
Decode a header value, un-escaping any escaped non-ASCII characters. *header_str* must be a *str* object. The unescaped value is returned. Using this function is recommended to display some headers in a human readable form:

```
>>> decode_header("Some subject")
'Some subject'
>>> decode_header("=?ISO-8859-15?Q?D=E9buter_en_Python?=")
'Débuter en Python'
>>> decode_header("Re: =?UTF-8?B?cHJvYmzDqG1lIGRlIG1hdHJpY2U=?=")
'Re: problème de matrice'
```

21.17 smtplib —SMTP 协议客户端

源代码： Lib/smtplib.py

---

smtplib 模块定义了一个 SMTP 客户端会话对象，该对象可将邮件发送到 Internet 上带有 SMTP 或 ESMTP 接收程序的计算机。关于 SMTP 和 ESMTP 操作的详情请参阅 **RFC 821** (简单邮件传输协议) 和 **RFC 1869** (SMTP 服务扩展)。

class smtplib.**SMTP**(*host=''*, *port=0*, *local_hostname=None*[, *timeout*], *source_address=None*)

一个*SMTP*实例就是一个封装好的 SMTP 连接。该实例具有的方法支持所有 SMTP 和 ESMTP 操作。如果传入了可选参数 host 和 port，那么将在初始化时使用这些参数调用 SMTP *connect()* 方法。如果传入了 *local_hostname*，它将在 HELO/EHLO 命令中被用作本地主机的 FQDN。否则将使用*socket.getfqdn()* 找到本地主机名。如果*connect()* 返回了成功码以外的内容，则引发*SMTPConnectError*异常。可选参数 *timeout* 指定阻塞操作（如连接尝试）的超时（以秒为单位，如果未指定超时，将使用全局默认超时设置）。到达超时时长后会引发*socket.timeout* 异常。可选参数 source_address 允许在有多张网卡的计算机中绑定到某些特定的源地址，和/或绑定到某些特定的源 TCP 端口。在连接前，套接字需要绑定一个 2 元组 (host, port) 作为其源地址。如果省略，或者是主机为 `''` 和/或端口为 0，则将使用操作系统默认行为。

正常使用时，只需要初始化或 connect 方法，*sendmail()* 方法，再加上*SMTP.quit()* 方法即可。下文包括了一个示例。

The *SMTP* class supports the `with` statement. When used like this, the SMTP `QUIT` command is issued automatically when the `with` statement exits. E.g.:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

在 3.3 版更改: 支持了 `with` 语句。

在 3.3 版更改: source_address argument was added.

3.5 新版功能: The SMTPUTF8 extension (**RFC 6531**) is now supported.

class smtplib.**SMTP_SSL**(*host=''*, *port=0*, *local_hostname=None*, *keyfile=None*, *certfile=None*[, *timeout*], *context=None*, *source_address=None*)

*SMTP_SSL*实例与*SMTP*实例的行为完全相同。在开始连接就需要 SSL，且 `starttls()` 不适合的情况下，应该使用*SMTP_SSL*。如果未指定 *host*，则使用 localhost。如果 *port* 为 0，则使用标准 SMTP-over-SSL 端口（465）。可选参数 *local_hostname*、*timeout* 和 *source_address* 的含义与*SMTP*类中的相同。可选参数 *context* 是一个*SSLContext*对象，可以从多个方面配置安全连接。请阅读*安全考量*以获取最佳实践。

keyfile and *certfile* are a legacy alternative to *context*, and can point to a PEM formatted private key and certificate chain file for the SSL connection.

在 3.3 版更改: 增加了 *context*。

在 3.3 版更改: source_address argument was added.

在 3.4 版更改: 本类现在支持通过*ssl.SSLContext.check_hostname* 和服务器名称提示（参阅*ssl.HAS_SNI*）进行主机名检查。

3.6 版后已移除: *keyfile* 和 *certfile* 已弃用并转而推荐 *context*。请改用*ssl.SSLContext.load_cert_chain()* 或让*ssl.create_default_context()* 为你选择系统所信任的 CA 证书。

class smtplib.**LMTP**(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*)

LMTP 协议与 ESMTP 非常相似，它很大程度上基于标准的 SMTP 客户端。将 Unix 套接字用于 LMTP 是很常见的，因此 `connect()` 方法支持 Unix 套接字，也支持常规的 host:port 服务器。可选参数 local_hostname 和 source_address 的含义与*SMTP*类中的相同。要指定 Unix 套接字，*host* 必须使用绝对路径，以‘/’开头。

Authentication is supported, using the regular SMTP mechanism. When using a Unix socket, LMTP generally don’t support or require any authentication, but your mileage might vary.

A nice selection of exceptions is defined as well:

exception smtplib.**SMTPException**
Subclass of *OSError* that is the base exception class for all the other exceptions provided by this module.

在 3.4 版更改: SMTPException became subclass of *OSError*

exception smtplib.**SMTPServerDisconnected**
This exception is raised when the server unexpectedly disconnects, or when an attempt is made to use the *SMTP* instance before connecting it to a server.

exception smtplib.**SMTPResponseException**
Base class for all exceptions that include an SMTP error code. These exceptions are generated in some instances when the SMTP server returns an error code. The error code is stored in the `smtp_code` attribute of the error, and the `smtp_error` attribute is set to the error message.

exception smtplib.**SMTPSenderRefused**
Sender address refused. In addition to the attributes set by on all *SMTPResponseException* exceptions, this sets 'sender' to the string that the SMTP server refused.

exception smtplib.**SMTPRecipientsRefused**
All recipient addresses refused. The errors for each recipient are accessible through the attribute `recipients`, which is a dictionary of exactly the same sort as *SMTP.sendmail()* returns.

exception smtplib.**SMTPDataError**
The SMTP server refused to accept the message data.

exception smtplib.**SMTPConnectError**
Error occurred during establishment of a connection with the server.

exception smtplib.**SMTPHeloError**
The server refused our `HELO` message.

exception smtplib.**SMTPNotSupportedError**
The command or option attempted is not supported by the server.

3.5 新版功能.

exception smtplib.**SMTPAuthenticationError**
SMTP authentication went wrong. Most probably the server didn't accept the username/password combination provided.

参见:

RFC 821 - Simple Mail Transfer Protocol Protocol definition for SMTP. This document covers the model, operating procedure, and protocol details for SMTP.

RFC 1869 - SMTP Service Extensions Definition of the ESMTP extensions for SMTP. This describes a framework for extending SMTP with new commands, supporting dynamic discovery of the commands provided by the server, and defines a few additional commands.

21.17.1 SMTP Objects

An *SMTP* instance has the following methods:

SMTP.**set_debuglevel**(*level*)
Set the debug output level. A value of 1 or `True` for *level* results in debug messages for connection and for all messages sent to and received from the server. A value of 2 for *level* results in these messages being timestamped.

在 3.5 版更改: Added debuglevel 2.

SMTP.**docmd**(*cmd*, *args=''*)
Send a command *cmd* to the server. The optional argument *args* is simply concatenated to the command, separated by a space.

This returns a 2-tuple composed of a numeric response code and the actual response line (multiline responses are joined into one long line.)

In normal operation it should not be necessary to call this method explicitly. It is used to implement other methods and may be useful for testing private extensions.

If the connection to the server is lost while waiting for the reply, *SMTPServerDisconnected* will be raised.

SMTP.**connect**(*host='localhost'*, *port=0*)
连接到某个主机的某个端口。默认是连接到 localhost 的标准 SMTP 端口（25）上。如果主机名以冒号（':'）结尾，后跟数字，则该后缀将被删除，且数字将视作要使用的端口号。如果在实例化时指定了 host，则构造函数会自动调用本方法。返回包含响应码和响应消息的 2 元组，它们由服务器在其连接响应中发送。

SMTP.**helo**(*name=''*)
Identify yourself to the SMTP server using HELO. The hostname argument defaults to the fully qualified domain name of the local host. The message returned by the server is stored as the helo_resp attribute of the object.

In normal operation it should not be necessary to call this method explicitly. It will be implicitly called by the *sendmail()* when necessary.

SMTP.**ehlo**(*name=''*)
使用 EHLO 向 ESMTP 服务器标识自身。hostname 参数默认为 localhost 的标准域名。使用*has_extn()*来检查响应中的 ESMTP 选项，并将它们保存起来。还给一些信息性的属性赋值：服务器返回的消息存储为 ehlo_resp 属性；根据服务器是否支持 ESMTP，将 does_esmtp 设置为 true 或 false；而 esmtp_features 是一个字典，包含该服务器支持的 SMTP 服务扩展的名称及参数（如果有参数）。

Unless you wish to use *has_extn()* before sending mail, it should not be necessary to call this method explicitly. It will be implicitly called by *sendmail()* when necessary.

SMTP.**ehlo_or_helo_if_needed**()
This method calls *ehlo()* and/or *helo()* if there has been no previous EHLO or HELO command this session. It tries ESMTP EHLO first.

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTP.**has_extn**(*name*)
Return *True* if *name* is in the set of SMTP service extensions returned by the server, *False* otherwise. Case is ignored.

SMTP.**verify**(*address*)
Check the validity of an address on this server using SMTP VRFY. Returns a tuple consisting of code 250 and a full **RFC 822** address (including human name) if the user address is valid. Otherwise returns an SMTP error code of 400 or greater and an error string.

注解： Many sites disable SMTP VRFY in order to foil spammers.

SMTP.**login**(*user*, *password*, *, *initial_response_ok=True*)
Log in on an SMTP server that requires authentication. The arguments are the username and the password to authenticate with. If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first. This method will return normally if the authentication was successful, or may raise the following exceptions:

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPAuthenticationError The server didn't accept the username/password combination.

SMTPNotSupportedError The AUTH command is not supported by the server.

SMTPException No suitable authentication method was found.

Each of the authentication methods supported by *smtplib* are tried in turn if they are advertised as supported by the server. See *auth()* for a list of supported authentication methods. *initial_response_ok* is passed through to *auth()*.

Optional keyword argument *initial_response_ok* specifies whether, for authentication methods that support it, an "initial response" as specified in **RFC 4954** can be sent along with the AUTH command, rather than requiring a challenge/response.

在 3.5 版更改: *SMTPNotSupportedError* may be raised, and the *initial_response_ok* parameter was added.

SMTP.**auth**(*mechanism*, *authobject*, *, *initial_response_ok=True*)

Issue an SMTP AUTH command for the specified authentication *mechanism*, and handle the challenge response via *authobject*.

mechanism specifies which authentication mechanism is to be used as argument to the AUTH command; the valid values are those listed in the auth element of esmtp_features.

authobject must be a callable object taking an optional single argument:

data = authobject(challenge=None)

If optional keyword argument *initial_response_ok* is true, authobject() will be called first with no argument. It can return the **RFC 4954** "initial response" ASCII str which will be encoded and sent with the AUTH command as below. If the authobject() does not support an initial response (e.g. because it requires a challenge), it should return None when called with challenge=None. If *initial_response_ok* is false, then authobject() will not be called first with None.

If the initial response check returns None, or if *initial_response_ok* is false, authobject() will be called to process the server's challenge response; the *challenge* argument it is passed will be a bytes. It should return ASCII str *data* that will be base64 encoded and sent to the server.

The SMTP class provides authobjects for the CRAM-MD5, PLAIN, and LOGIN mechanisms; they are named SMTP.auth_cram_md5, SMTP.auth_plain, and SMTP.auth_login respectively. They all require that the user and password properties of the SMTP instance are set to appropriate values.

User code does not normally need to call auth directly, but can instead call the *login()* method, which will try each of the above mechanisms in turn, in the order listed. auth is exposed to facilitate the implementation of authentication methods not (or not yet) supported directly by *smtplib*.

3.5 新版功能.

SMTP.**starttls**(*keyfile=None*, *certfile=None*, *context=None*)

Put the SMTP connection in TLS (Transport Layer Security) mode. All SMTP commands that follow will be encrypted. You should then call *ehlo()* again.

If *keyfile* and *certfile* are provided, they are used to create an *ssl.SSLContext*.

Optional *context* parameter is an *ssl.SSLContext* object; This is an alternative to using a keyfile and a certfile and if specified both *keyfile* and *certfile* should be None.

If there has been no previous EHLO or HELO command this session, this method tries ESMTP EHLO first.

3.6 版后已移除: *keyfile* 和 *certfile* 已弃用并转而推荐 *context*。请改用*ssl.SSLContext.load_cert_chain()* 或让*ssl.create_default_context()* 为你选择系统所信任的 CA 证书。

SMTPHeloError The server didn't reply properly to the HELO greeting.

SMTPNotSupportedError The server does not support the STARTTLS extension.

RuntimeError SSL/TLS support is not available to your Python interpreter.

在 3.3 版更改: 增加了 *context*。

在 3.4 版更改: The method now supports hostname check with `SSLContext.check_hostname` and *Server Name Indicator* (see *HAS_SNI*).

在 3.5 版更改: The error raised for lack of STARTTLS support is now the *SMTPNotSupportedError* subclass instead of the base *SMTPException*.

SMTP.**sendmail**(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)
发送邮件。必要参数是一个 **RFC 822** 发件地址字符串，一个 **RFC 822** 收件地址字符串列表（裸字符串将被视为含有 1 个地址的列表），以及一个消息字符串。调用者可以将 ESMTP 选项列表（如 `8bitmime`）作为 *mail_options* 传入，用于 `MAIL FROM` 命令。需要与所有 `RCPT` 命令一起使用的 ESMTP 选项（如 `DSN` 命令）可以作为 *rcpt_options* 传入。（如果需要对不同的收件人使用不同的 ESMTP 选项，则必须使用底层的方法来发送消息，如 `mail()`, `rcpt()` 和 `data()`。）

注解: The *from_addr* and *to_addrs* parameters are used to construct the message envelope used by the transport agents. `sendmail` does not modify the message headers in any way.

---

msg may be a string containing characters in the ASCII range, or a byte string. A string is encoded to bytes using the ascii codec, and lone `\r` and `\n` characters are converted to `\r\n` characters. A byte string is not modified.

If there has been no previous `EHLO` or `HELO` command this session, this method tries ESMTP `EHLO` first. If the server does ESMTP, message size and each of the specified options will be passed to it (if the option is in the feature set the server advertises). If `EHLO` fails, `HELO` will be tried and ESMTP options suppressed.

This method will return normally if the mail is accepted for at least one recipient. Otherwise it will raise an exception. That is, if this method does not raise an exception, then someone should get your mail. If this method does not raise an exception, it returns a dictionary, with one entry for each recipient that was refused. Each entry contains a tuple of the SMTP error code and the accompanying error message sent by the server.

If `SMTPUTF8` is included in *mail_options*, and the server supports it, *from_addr* and *to_addrs* may contain non-ASCII characters.

This method may raise the following exceptions:

SMTPRecipientsRefused All recipients were refused. Nobody got the mail. The `recipients` attribute of the exception object is a dictionary with information about the refused recipients (like the one returned when at least one recipient was accepted).

SMTPHeloError The server didn't reply properly to the `HELO` greeting.

SMTPSenderRefused The server didn't accept the *from_addr*.

SMTPDataError The server replied with an unexpected error code (other than a refusal of a recipient).

SMTPNotSupportedError `SMTPUTF8` was given in the *mail_options* but is not supported by the server.

Unless otherwise noted, the connection will be open even after an exception is raised.

在 3.2 版更改: *msg* may be a byte string.

在 3.5 版更改: `SMTPUTF8` support added, and *SMTPNotSupportedError* may be raised if `SMTPUTF8` is specified but the server does not support it.

SMTP.**send_message**(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)
本方法是一种快捷方法，用于带着消息调用*sendmail()*，消息由*email.message.Message* 对象表示。参数的含义与*sendmail()* 中的相同，除了 *msg*，它是一个 `Message` 对象。

如果 *from_addr* 为 `None` 或 *to_addrs* 为 `None`，那么 "send_message"将根据 **RFC 5322**，从 *msg* 头部提取地址填充下列参数：如果头部存在 *Sender* 字段，则用它填充 *from_addr*，不存在则用 *From* 字段填充 *from_addr*。*to_addrs* 组合了 *msg* 中的 *To*, *Cc* 和 *Bcc* 字段的值（字段存在的情况下）。如果一组

*Resent-** 头部恰好出现在 message 中，那么就忽略常规的头部，改用 *Resent-** 头部。如果 message 包含多组 *Resent-** 头部，则引发*ValueError*，因为无法明确检测出哪一组 *Resent-* 头部是最新的。

send_message 使用*BytesGenerator* 来序列化 *msg*，且将 \r\n 作为 *linesep*，并调用*sendmail()* 来传输序列化后的结果。无论 *from_addr* 和 *to_addrs* 的值为何，send_message 都不会传输 *msg* 中可能出现的 *Bcc* 或 *Resent-Bcc* 头部。如果 *from_addr* 和 *to_addrs* 中的某个地址包含非 ASCII 字符，且服务器没有声明支持 SMTPUTF8，则引发 SMTPNotSupported 错误。如果服务器支持，则 Message 将按新克隆的*policy* 进行序列化，其中的*utf8* 属性被设置为 True，且 SMTPUTF8 和 BODY=8BITMIME 被添加到 *mail_options* 中。

3.2 新版功能.

3.5 新版功能: Support for internationalized addresses (SMTPUTF8).

SMTP.**quit**()
Terminate the SMTP session and close the connection. Return the result of the SMTP QUIT command.

Low-level methods corresponding to the standard SMTP/ESMTP commands HELP, RSET, NOOP, MAIL, RCPT, and DATA are also supported. Normally these do not need to be called directly, so they are not documented here. For details, consult the module code.

21.17.2 SMTP Example

This example prompts the user for addresses needed in the message envelope ('To' and 'From' addresses), and the message to be delivered. Note that the headers to be included with the message must be included in the message as entered; this example doesn't do any processing of the **RFC 822** headers. In particular, the 'To' and 'From' addresses must be included in the message headers explicitly.

```
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

注解: In general, you will want to use the *email* package's features to construct an email message, which you can then send via *send_message()*; see *email: 示例*.

21.18 smtpd —SMTP 服务器

源代码: Lib/smtpd.py

This module offers several classes to implement SMTP (email) servers.

参见:

The aiosmtpd package is a recommended replacement for this module. It is based on *asyncio* and provides a more straightforward API. *smtpd* should be considered deprecated.

Several server implementations are present; one is a generic do-nothing implementation, which can be overridden, while the other two offer specific mail-sending strategies.

Additionally the SMTPChannel may be extended to implement very specific interaction behaviour with SMTP clients.

The code supports **RFC 5321**, plus the **RFC 1870** SIZE and **RFC 6531** SMTPUTF8 extensions.

21.18.1 SMTPServer 对象

class smtpd.**SMTPServer**(*localaddr*, *remoteaddr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*)

Create a new *SMTPServer* object, which binds to local address *localaddr*. It will treat *remoteaddr* as an upstream SMTP relayer. Both *localaddr* and *remoteaddr* should be a *(host, port)* tuple. The object inherits from *asyncore.dispatcher*, and so will insert itself into *asyncore*'s event loop on instantiation.

data_size_limit specifies the maximum number of bytes that will be accepted in a `DATA` command. A value of `None` or `0` means no limit.

map is the socket map to use for connections (an initially empty dictionary is a suitable value). If not specified the *asyncore* global socket map is used.

enable_SMTPUTF8 determines whether the `SMTPUTF8` extension (as defined in **RFC 6531**) should be enabled. The default is `False`. When `True`, `SMTPUTF8` is accepted as a parameter to the `MAIL` command and when present is passed to *process_message()* in the `kwargs['mail_options']` list. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. When *decode_data* is `False` (the default), the server advertises the `8BITMIME` extension (**RFC 6152**), accepts the `BODY=8BITMIME` parameter to the `MAIL` command, and when present passes it to *process_message()* in the `kwargs['mail_options']` list. *decode_data* and *enable_SMTPUTF8* cannot be set to `True` at the same time.

process_message(*peer*, *mailfrom*, *rcpttos*, *data*, ***kwargs*)

Raise a *NotImplementedError* exception. Override this in subclasses to do something useful with this message. Whatever was passed in the constructor as *remoteaddr* will be available as the `_remoteaddr` attribute. *peer* is the remote host's address, *mailfrom* is the envelope originator, *rcpttos* are the envelope recipients and *data* is a string containing the contents of the e-mail (which should be in **RFC 5321** format).

If the *decode_data* constructor keyword is set to `True`, the *data* argument will be a unicode string. If it is set to `False`, it will be a bytes object.

kwargs is a dictionary containing additional information. It is empty if `decode_data=True` was given as an init argument, otherwise it contains the following keys:

***mail_options*:** a list of all received parameters to the `MAIL` command (the elements are uppercase strings; example: `['BODY=8BITMIME', 'SMTPUTF8']`).

***rcpt_options*:** same as *mail_options* but for the `RCPT` command. Currently no `RCPT TO` options are supported, so for now this will always be an empty list.

Implementations of `process_message` should use the `**kwargs` signature to accept arbitrary keyword arguments, since future feature enhancements may add keys to the kwargs dictionary.

Return `None` to request a normal `250 Ok` response; otherwise return the desired response string in **RFC 5321** format.

channel_class
Override this in subclasses to use a custom *SMTPChannel* for managing SMTP clients.

3.4 新版功能: The *map* constructor argument.

在 3.5 版更改: *localaddr* and *remoteaddr* may now contain IPv6 addresses.

3.5 新版功能: The *decode_data* and *enable_SMTPUTF8* constructor parameters, and the *kwargs* parameter to *process_message()* when *decode_data* is `False`.

在 3.6 版更改: *decode_data* is now `False` by default.

21.18.2 DebuggingServer 对象

class smtpd.**DebuggingServer**(*localaddr*, *remoteaddr*)
Create a new debugging server. Arguments are as per *SMTPServer*. Messages will be discarded, and printed on stdout.

21.18.3 PureProxy 对象

class smtpd.**PureProxy**(*localaddr*, *remoteaddr*)
Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*. Note that running this has a good chance to make you into an open relay, so please be careful.

21.18.4 MailmanProxy 对象

class smtpd.**MailmanProxy**(*localaddr*, *remoteaddr*)
Create a new pure proxy server. Arguments are as per *SMTPServer*. Everything will be relayed to *remoteaddr*, unless local mailman configurations knows about an address, in which case it will be handled via mailman. Note that running this has a good chance to make you into an open relay, so please be careful.

21.18.5 SMTPChannel 对象

class smtpd.**SMTPChannel**(*server*, *conn*, *addr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*)

Create a new *SMTPChannel* object which manages the communication between the server and a single SMTP client.

conn and *addr* are as per the instance variables described below.

data_size_limit specifies the maximum number of bytes that will be accepted in a DATA command. A value of None or 0 means no limit.

enable_SMTPUTF8 determines whether the SMTPUTF8 extension (as defined in **RFC 6531**) should be enabled. The default is False. *decode_data* and *enable_SMTPUTF8* cannot be set to True at the same time.

A dictionary can be specified in *map* to avoid using a global socket map.

decode_data specifies whether the data portion of the SMTP transaction should be decoded using UTF-8. The default is False. *decode_data* and *enable_SMTPUTF8* cannot be set to True at the same time.

To use a custom SMTPChannel implementation you need to override the *SMTPServer.channel_class* of your *SMTPServer*.

在 3.5 版更改: The *decode_data* and *enable_SMTPUTF8* parameters were added.

在 3.6 版更改: *decode_data* is now False by default.

The *SMTPChannel* has the following instance variables:

smtp_server
Holds the *SMTPServer* that spawned this channel.

conn
Holds the socket object connecting to the client.

addr
Holds the address of the client, the second value returned by *socket.accept*

received_lines
Holds a list of the line strings (decoded using UTF-8) received from the client. The lines have their "\r\n" line ending translated to "\n".

smtp_state
Holds the current state of the channel. This will be either COMMAND initially and then DATA after the client sends a "DATA" line.

seen_greeting
Holds a string containing the greeting sent by the client in its "HELO".

mailfrom
Holds a string containing the address identified in the "MAIL FROM:" line from the client.

rcpttos
Holds a list of strings containing the addresses identified in the "RCPT TO:" lines from the client.

received_data
Holds a string containing all of the data sent by the client during the DATA state, up to but not including the terminating "\r\n.\r\n".

fqdn
Holds the fully-qualified domain name of the server as returned by *socket.getfqdn()*.

peer
Holds the name of the client peer as returned by conn.getpeername() where conn is *conn*.

The *SMTPChannel* operates by invoking methods named `smtp_<command>` upon reception of a command line from the client. Built into the base *SMTPChannel* class are methods for handling the following commands (and responding to them appropriately):

命令	所采取的行动
HELO	接受来自客户端的问候语，并将其存储在*seen_greeting* 中。将服务器设置为基本命令模式。
EHLO	接受来自客户的问候并将其存储在*seen_greeting* 中。将服务器设置为扩展命令模式。
NOOP	不采取任何措施。
QUIT	干净地关闭连接。
MAIL	Accepts the "MAIL FROM:" syntax and stores the supplied address as *mailfrom*. In extended command mode, accepts the **RFC 1870** SIZE attribute and responds appropriately based on the value of *data_size_limit*.
RCPT	Accepts the "RCPT TO:" syntax and stores the supplied addresses in the *rcpttos* list.
RSET	重置*mailfrom*, *rcpttos*, 和*received_data* ，但不重置问候语。
DATA	Sets the internal state to `DATA` and stores remaining lines from the client in *received_data* until the terminator `"\r\n.\r\n"` is received.
HELP	返回有关命令语法的最少信息
VRFY	返回代码 252（服务器不知道该地址是否有效）
EXPN	报告该命令未实现。

21.19 `telnetlib` –Telnet 客户端

源代码: Lib/telnetlib.py

---

telnetlib 模块提供一个实现 Telnet 协议的类*Telnet*。关于此协议的细节请参见 **RFC 854** 。此外，它还为协议字符（见下文）和 telnet 选项提供了对应的符号常量。telnet 选项对应的符号名遵循 `arpa/telnet.h` 中的定义，但删除了前缀 "TELNET_"。对于不在 `arpa/telnet.h` 的选项的符号常量名，请参考本模块源码。

telnet 命令的符号常量名有：IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

class telnetlib.**Telnet**(*host=None*, *port=0*[, *timeout*])

Telnet represents a connection to a Telnet server. The instance is initially not connected by default; the *open()* method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor too, in which case the connection to the server will be established before the constructor returns. The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used).

不要重新打开一个已经连接的实例。

这个类有很多 `read_*()` 方法。请注意，其中一些方法在读取结束时会触发*EOFError* 异常，这是由于连接对象可能出于其它原因返回一个空字符串。请参阅下面的个别描述。

A *Telnet* object is a context manager and can be used in a `with` statement. When the `with` block ends, the *close()* method is called:

```
>>> from telnetlib import Telnet
>>> with Telnet('localhost', 23) as tn:
...     tn.interact()
...
```

在 3.6 版更改: 添加了上下文管理器的支持

参见:

RFC 854 - Telnet 协议规范 Telnet 协议的定义。

21.19.1 Telnet 对象

Telnet 实例有以下几种方法:

Telnet.**read_until**(*expected*, *timeout=None*)
读取直到遇到给定字节串 *expected* 或 *timeout* 秒已经过去。

当没有找到匹配时，返回可用的内容，也可能返回空字节。如果连接已关闭且没有可用的数据，将触发*EOFError*。

Telnet.**read_all**()
读取数据，直到遇到 EOF；连接关闭前都会保持阻塞。

Telnet.**read_some**()
在达到 EOF 前，读取至少一个字节的熟数据。如果命中 EOF，返回 `b''`。如果没有立即可用的数据，则阻塞。

Telnet.**read_very_eager**()
在不阻塞 I/O 的情况下（急切地）读取所有的内容。

如果连接关闭并且没有可用的熟数据，将会触发*EOFError* 。如果没有熟数据可用返回 `b''` 。除非在一个 IAC 序列的中间，否则不要阻塞。

Telnet.**read_eager**()
读取现成的数据。

如果连接关闭并且没有可用的熟数据，将会触发*EOFError* 。如果没有熟数据可用返回 `b''` 。除非在一个 IAC 序列的中间，否则不要阻塞。

Telnet.**read_lazy**()
处理并返回已经在队列中的数据（lazy）。

如果连接已关闭并且没有可用的数据，将会触发*EOFError* 。如果没有熟数据可用则返回 `b''` 。除非在一个 IAC 序列的中间，否则不要进行阻塞。

Telnet.**read_very_lazy**()
返回熟数据队列任何可用的数据（very lazy）。

如果连接已关闭并且没有可用的数据，将会触发*EOFError* 。如果没有熟数据可用则返回 `b''` 。该方法永远不会阻塞。

Telnet.**read_sb_data**()
返回在 SB/SE 对之间收集的数据（子选项 begin/end）。当使用 `SE` 命令调用回调函数时，该回调函数应该访问这些数据。该方法永远不会阻塞。

Telnet.**open**(*host*, *port=0*[, *timeout*])
连接主机。第二个可选参数是端口号，默认为标准 Telnet 端口（23）。可选参数 *timeout* 指定一个以秒为单位的超时时间用于像连接尝试这样的阻塞操作（如果没有指定，将使用全局默认超时设置）。

不要尝试重新打开一个已经连接的实例。

Telnet.**msg**(*msg*, **args*)
当 debug level > 0 时打印一条 debug 信息。如果存在额外参数，则它们会被替换在使用标准字符串格式化操作符的信息中。

Telnet.**set_debuglevel**(*debuglevel*)
设置调试级别

Telnet.**close**()
关闭连接对象。

Telnet.**get_socket**()
返回内部使用的套接字对象。

Telnet.**fileno**()
返回内部使用的套接字对象的文件描述符。

Telnet.**write**(*buffer*)
向套接字写入一个字节字符串，将所有 IAC 字符加倍。如果连接被阻塞，这可能也会阻塞。如果连接关闭可能触发*OSError*。

在 3.3 版更改: 曾经该函数抛出*socket.error*，现在这是*OSError* 的别名。

Telnet.**interact**()
交互函数，模拟一个非常笨拙的 Telnet 客户端。

Telnet.**mt_interact**()
多线程版的*interact()*.

Telnet.**expect**(*list*, *timeout=None*)
一直读取，直到匹配列表中的某个正则表达式。

第一个参数是一个正则表达式列表，可以是已编译的 (*正则表达式对象*)，也可以是未编译的 (字节串)。第二个可选参数是超时，单位是秒；默认一直阻塞。

返回一个包含三个元素的元组：列表中的第一个匹配的正则表达式的索引；返回的匹配对象；包括匹配在内的读取过的字节。

如果找到了文件的结尾且没有字节被读取，触发*EOFError*。否则，当没有匹配时, 返回 `(-1, None, data)`，其中 *data* 是到目前为止接受到的字节（如果发生超时，则可能是空字节）。

如果一个正则表达式以贪婪匹配结束（例如 `.*`），或者多个表达式可以匹配同一个输出，则结果是不确定的，可能取决于 I/O 计时。

Telnet.**set_option_negotiation_callback**(*callback*)
每次在输入流上读取 telnet 选项时，这个带有如下参数的 *callback*（如果设置了）会被调用：callback(telnet socket, command (DO/DONT/WILL/WONT), option)。telnetlib 之后不会再执行其它操作。

21.19.2 Telnet 示例

一个简单的说明性典型用法例子:

```
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
tn.write(user.encode('ascii') + b"\n")
if password:
    tn.read_until(b"Password: ")
```

(下页继续)

(续上页)

```
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```

21.20 uuid —UUID objects according to RFC 4122

Source code: Lib/uuid.py

这个模块提供了不可变的*UUID*对象 (*UUID*类) 和*uuid1()*, *uuid3()*, *uuid4()*, *uuid5()* 等函数用于生成 **RFC 4122** 所定义的第 1, 3, 4 和 5 版 UUID。

If all you want is a unique ID, you should probably call *uuid1()* or *uuid4()*. Note that *uuid1()* may compromise privacy since it creates a UUID containing the computer's network address. *uuid4()* creates a random UUID.

class uuid.**UUID**(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*)

Create a UUID from either a string of 32 hexadecimal digits, a string of 16 bytes in big-endian order as the *bytes* argument, a string of 16 bytes in little-endian order as the *bytes_le* argument, a tuple of six integers (32-bit *time_low*, 16-bit *time_mid*, 16-bit *time_hi_version*, 8-bit *clock_seq_hi_variant*, 8-bit *clock_seq_low*, 48-bit *node*) as the *fields* argument, or a single 128-bit integer as the *int* argument. When a string of hex digits is given, curly braces, hyphens, and a URN prefix are all optional. For example, these expressions all yield the same UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

Exactly one of *hex*, *bytes*, *bytes_le*, *fields*, or *int* must be given. The *version* argument is optional; if given, the resulting UUID will have its variant and version number set according to **RFC 4122**, overriding bits in the given *hex*, *bytes*, *bytes_le*, *fields*, or *int*.

Comparison of UUID objects are made by way of comparing their *UUID.int* attributes. Comparison with a non-UUID object raises a *TypeError*.

`str(uuid)` returns a string in the form `12345678-1234-5678-1234-567812345678` where the 32 hexadecimal digits represent the UUID.

UUID instances have these read-only attributes:

UUID.**bytes**

The UUID as a 16-byte string (containing the six integer fields in big-endian byte order).

UUID.**bytes_le**

The UUID as a 16-byte string (with *time_low*, *time_mid*, and *time_hi_version* in little-endian byte order).

UUID.**fields**

以元组形式存放的 UUID 的 6 个整数域，有六个单独的属性和两个派生属性：

域	含义
time_low	UUID 的前 32 位
time_mid	接前一域的 16 位
time_hi_version	接前一域的 16 位
clock_seq_hi_variant	接前一域的 8 位
clock_seq_low	接前一域的 8 位
node	UUID 的最后 48 位
time	UUID 的总长 60 位的时间戳
clock_seq	14 位的序列号

UUID.**hex**
The UUID as a 32-character hexadecimal string.

UUID.**int**
The UUID as a 128-bit integer.

UUID.**urn**
在 **RFC 4122** 中定义的 URN 形式的 UUID。

UUID.**variant**
The UUID variant, which determines the internal layout of the UUID. This will be one of the constants *RESERVED_NCS*, *RFC_4122*, *RESERVED_MICROSOFT*, or *RESERVED_FUTURE*.

UUID.**version**
The UUID version number (1 through 5, meaningful only when the variant is *RFC_4122*).

The *uuid* module defines the following functions:

uuid.**getnode**()
Get the hardware address as a 48-bit positive integer. The first time this runs, it may launch a separate program, which could be quite slow. If all attempts to obtain the hardware address fail, we choose a random 48-bit number with its eighth bit set to 1 as recommended in **RFC 4122**. "Hardware address" means the MAC address of a network interface, and on a machine with multiple network interfaces the MAC address of any one of them may be returned.

uuid.**uuid1**(*node=None*, *clock_seq=None*)
Generate a UUID from a host ID, sequence number, and the current time. If *node* is not given, *getnode()* is used to obtain the hardware address. If *clock_seq* is given, it is used as the sequence number; otherwise a random 14-bit sequence number is chosen.

uuid.**uuid3**(*namespace*, *name*)
Generate a UUID based on the MD5 hash of a namespace identifier (which is a UUID) and a name (which is a string).

uuid.**uuid4**()
Generate a random UUID.

uuid.**uuid5**(*namespace*, *name*)
Generate a UUID based on the SHA-1 hash of a namespace identifier (which is a UUID) and a name (which is a string).

The *uuid* module defines the following namespace identifiers for use with *uuid3()* or *uuid5()*.

uuid.**NAMESPACE_DNS**
When this namespace is specified, the *name* string is a fully-qualified domain name.

uuid.**NAMESPACE_URL**
When this namespace is specified, the *name* string is a URL.

uuid.**NAMESPACE_OID**
When this namespace is specified, the *name* string is an ISO OID.

uuid.**NAMESPACE_X500**
When this namespace is specified, the *name* string is an X.500 DN in DER or a text output format.

The *uuid* module defines the following constants for the possible values of the `variant` attribute:

uuid.**RESERVED_NCS**
Reserved for NCS compatibility.

uuid.**RFC_4122**
Specifies the UUID layout given in **RFC 4122**.

uuid.**RESERVED_MICROSOFT**
为微软的兼容性保留。

uuid.**RESERVED_FUTURE**
Reserved for future definition.

参见:

RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace This specification defines a Uniform Resource Name namespace for UUIDs, the internal format of UUIDs, and methods of generating UUIDs.

21.20.1 示例

Here are some examples of typical usage of the *uuid* module:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
```

(下页继续)

(续上页)

```
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

21.21 socketserver —A framework for network servers

Source code: Lib/socketserver.py

The *socketserver* module simplifies the task of writing network servers.

There are four basic concrete server classes:

class socketserver.**TCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke *server_bind()* and *server_activate()*. The other parameters are passed to the *BaseServer* base class.

class socketserver.**UDPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for *TCPServer*.

class socketserver.**UnixStreamServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
class socketserver.**UnixDatagramServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they're not available on non-Unix platforms. The parameters are the same as for *TCPServer*.

These four classes process requests *synchronously*; each request must be completed before the next request can be started. This isn't suitable if each request takes a long time to complete, because it requires a lot of computation, or because it returns a lot of data which the client is slow to process. The solution is to create a separate process or thread to handle each request; the *ForkingMixIn* and *ThreadingMixIn* mix-in classes can be used to support asynchronous behaviour.

Creating a server requires several steps. First, you must create a request handler class by subclassing the *BaseRequestHandler* class and overriding its *handle()* method; this method will process incoming requests. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class. It is recommended to use the server in a `with` statement. Then call the *handle_request()* or *serve_forever()* method of the server object to process one or many requests. Finally, call *server_close()* to close the socket (unless you used a `with` statement).

When inheriting from *ThreadingMixIn* for threaded connection behavior, you should explicitly declare how you want your threads to behave on an abrupt shutdown. The *ThreadingMixIn* class defines an attribute *daemon_threads*, which indicates whether or not the server should wait for thread termination. You should set the flag explicitly if you would like threads to behave autonomously; the default is *False*, meaning that Python will not exit until all threads created by *ThreadingMixIn* have exited.

Server classes have the same external methods and attributes, no matter what network protocol they use.

21.21.1 Server Creation Notes

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:

```
+------------+
| BaseServer |
+------------+
      |
      v
+-----------+        +------------------+
| TCPServer |------->| UnixStreamServer |
+-----------+        +------------------+
      |
      v
+-----------+        +--------------------+
| UDPServer |------->| UnixDatagramServer |
+-----------+        +--------------------+
```

Note that *UnixDatagramServer* derives from *UDPServer*, not from *UnixStreamServer* —the only difference between an IP and a Unix stream server is the address family, which is simply repeated in both Unix server classes.

class socketserver.**ForkingMixIn**
class socketserver.**ThreadingMixIn**
Forking and threading versions of each type of server can be created using these mix-in classes. For instance, *ThreadingUDPServer* is created as follows:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

The mix-in class comes first, since it overrides a method defined in *UDPServer*. Setting the various attributes also changes the behavior of the underlying server mechanism.

ForkingMixIn and the Forking classes mentioned below are only available on POSIX platforms that support *fork()*.

class socketserver.**ForkingTCPServer**
class socketserver.**ForkingUDPServer**
class socketserver.**ThreadingTCPServer**
class socketserver.**ThreadingUDPServer**
These classes are pre-defined using the mix-in classes.

To implement a service, you must derive a class from *BaseRequestHandler* and redefine its *handle()* method. You can then run various versions of the service by combining one of the server classes with your request handler class. The request handler class must be different for datagram or stream services. This can be hidden by using the handler subclasses *StreamRequestHandler* or *DatagramRequestHandler*.

Of course, you still have to use your head! For instance, it makes no sense to use a forking server if the service contains state in memory that can be modified by different requests, since the modifications in the child process would never reach the initial state kept in the parent process and passed to each child. In this case, you can use a threading server, but you will probably have to use locks to protect the integrity of the shared data.

On the other hand, if you are building an HTTP server where all data is stored externally (for instance, in the file system), a synchronous class will essentially render the service "deaf" while one request is being handled –which may be for a very long time if a client is slow to receive all the data it has requested. Here a threading or forking server is appropriate.

In some cases, it may be appropriate to process part of a request synchronously, but to finish processing in a forked child depending on the request data. This can be implemented by using a synchronous server and doing an explicit fork in the request handler class *handle()* method.

Another approach to handling multiple simultaneous requests in an environment that supports neither threads nor *fork()* (or where these are too expensive or inappropriate for the service) is to maintain an explicit table of partially finished requests and to use *selectors* to decide which request to work on next (or whether to handle a new incoming request). This is particularly important for stream services where each client can potentially be connected for a long time (if threads or subprocesses cannot be used). See *asyncore* for another way to manage this.

21.21.2 Server 对象

class socketserver.**BaseServer**(*server_address*, *RequestHandlerClass*)

This is the superclass of all Server objects in the module. It defines the interface, given below, but does not implement most of the methods, which is done in subclasses. The two parameters are stored in the respective *server_address* and *RequestHandlerClass* attributes.

fileno()

Return an integer file descriptor for the socket on which the server is listening. This function is most commonly passed to *selectors*, to allow monitoring multiple servers in the same process.

handle_request()

Process a single request. This function calls the following methods in order: *get_request()*, *verify_request()*, and *process_request()*. If the user-provided *handle()* method of the handler class raises an exception, the server's *handle_error()* method will be called. If no request is received within *timeout* seconds, *handle_timeout()* will be called and *handle_request()* will return.

serve_forever(*poll_interval=0.5*)

Handle requests until an explicit *shutdown()* request. Poll for shutdown every *poll_interval* seconds. Ignores the *timeout* attribute. It also calls *service_actions()*, which may be used by a subclass or mixin to provide actions specific to a given service. For example, the *ForkingMixIn* class uses *service_actions()* to clean up zombie child processes.

在 3.3 版更改: Added `service_actions` call to the `serve_forever` method.

service_actions()

This is called in the *serve_forever()* loop. This method can be overridden by subclasses or mixin classes to perform actions specific to a given service, such as cleanup actions.

3.3 新版功能.

shutdown()

Tell the *serve_forever()* loop to stop and wait until it does.

server_close()

Clean up the server. May be overridden.

address_family

The family of protocols to which the server's socket belongs. Common examples are *socket.AF_INET* and *socket.AF_UNIX*.

RequestHandlerClass

The user-provided request handler class; an instance of this class is created for each request.

server_address

The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the *socket* module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

socket

The socket object on which the server will listen for incoming requests.

The server classes support the following class variables:

allow_reuse_address
Whether the server will allow the reuse of an address. This defaults to *False*, and can be set in subclasses to change the policy.

request_queue_size
The size of the request queue. If it takes a long time to process a single request, any requests that arrive while the server is busy are placed into a queue, up to *request_queue_size* requests. Once the queue is full, further requests from clients will get a "Connection denied" error. The default value is usually 5, but this can be overridden by subclasses.

socket_type
The type of socket used by the server; *socket.SOCK_STREAM* and *socket.SOCK_DGRAM* are two common values.

timeout
Timeout duration, measured in seconds, or *None* if no timeout is desired. If *handle_request()* receives no incoming requests within the timeout period, the *handle_timeout()* method is called.

There are various server methods that can be overridden by subclasses of base server classes like *TCPServer*; these methods aren't useful to external users of the server object.

finish_request(*request*, *client_address*)
Actually processes the request by instantiating *RequestHandlerClass* and calling its *handle()* method.

get_request()
Must accept a request from the socket, and return a 2-tuple containing the *new* socket object to be used to communicate with the client, and the client's address.

handle_error(*request*, *client_address*)
This function is called if the *handle()* method of a *RequestHandlerClass* instance raises an exception. The default action is to print the traceback to standard error and continue handling further requests.

在 3.6 版更改: Now only called for exceptions derived from the *Exception* class.

handle_timeout()
This function is called when the *timeout* attribute has been set to a value other than *None* and the timeout period has passed with no requests being received. The default action for forking servers is to collect the status of any child processes that have exited, while in threading servers this method does nothing.

process_request(*request*, *client_address*)
Calls *finish_request()* to create an instance of the *RequestHandlerClass*. If desired, this function can create a new process or thread to handle the request; the *ForkingMixIn* and *ThreadingMixIn* classes do this.

server_activate()
Called by the server's constructor to activate the server. The default behavior for a TCP server just invokes *listen()* on the server's socket. May be overridden.

server_bind()
Called by the server's constructor to bind the socket to the desired address. May be overridden.

verify_request(*request*, *client_address*)
Must return a Boolean value; if the value is *True*, the request will be processed, and if it's *False*, the request will be denied. This function can be overridden to implement access controls for a server. The default implementation always returns *True*.

在 3.6 版更改: Support for the *context manager* protocol was added. Exiting the context manager is equivalent to calling *server_close()*.

21.21.3 Request Handler Objects

class socketserver.**BaseRequestHandler**

This is the superclass of all request handler objects. It defines the interface, given below. A concrete request handler subclass must define a new *handle()* method, and can override any of the other methods. A new instance of the subclass is created for each request.

setup()

Called before the *handle()* method to perform any initialization actions required. The default implementation does nothing.

handle()

This function must do all the work required to service a request. The default implementation does nothing. Several instance attributes are available to it; the request is available as `self.request`; the client address as `self.client_address`; and the server instance as `self.server`, in case it needs access to per-server information.

The type of `self.request` is different for datagram or stream services. For stream services, `self.request` is a socket object; for datagram services, `self.request` is a pair of string and socket.

finish()

Called after the *handle()* method to perform any clean-up actions required. The default implementation does nothing. If *setup()* raises an exception, this function will not be called.

class socketserver.**StreamRequestHandler**

class socketserver.**DatagramRequestHandler**

These *BaseRequestHandler* subclasses override the *setup()* and *finish()* methods, and provide `self.rfile` and `self.wfile` attributes. The `self.rfile` and `self.wfile` attributes can be read or written, respectively, to get the request data or return data to the client.

The `rfile` attributes of both classes support the *io.BufferedIOBase* readable interface, and `DatagramRequestHandler.wfile` supports the *io.BufferedIOBase* writable interface.

在 3.6 版更改: `StreamRequestHandler.wfile` also supports the *io.BufferedIOBase* writable interface.

21.21.4 例子

socketserver.TCPServer Example

This is the server side:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
```

(下页继续)

(续上页)

```
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

An alternative request handler class that makes use of streams (file-like objects that simplify communication by providing the standard file interface):

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

The difference is that the `readline()` call in the second handler will call `recv()` multiple times until it encounters a newline character, while the single `recv()` call in the first handler will just return what has been sent from the client in one `sendall()` call.

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look something like this:

Server:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

Client:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

socketserver.UDPServer Example

This is the server side:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print("{} wrote:".format(self.client_address[0]))
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

This is the client side:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")
```

(下页继续)

(续上页)

```
print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

The output of the example should look exactly like for the TCP server example.

Asynchronous Mixins

To build asynchronous handlers, use the *ThreadingMixIn* and *ForkingMixIn* classes.

An example for the *ThreadingMixIn* class:

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will then start one
        # more thread for each request
        server_thread = threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread terminates
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

The output of the example should look something like this:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

The *ForkingMixIn* class is used in the same way, except that the server will spawn a new process for each request. Available only on POSIX platforms that support *fork()*.

21.22 http.server —HTTP 服务器

源代码： Lib/http/server.py

这个模块定义了实现 HTTP 服务器（Web 服务器）的类。

> **警告：** *http.server* is not recommended for production. It only implements only basic security checks.

HTTPServer 是*socketserver.TCPServer* 的一个子类。它会创建和侦听 HTTP 套接字，并将请求调度给处理程序。用于创建和运行服务器的代码看起来像这样：

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class http.server.**HTTPServer**(*server_address*, *RequestHandlerClass*)
该类基于*TCPServer* 类，并会将服务器地址存入名为 server_name 和 server_port 的实例变量中。服务器可被处理程序通过 server 实例变量访问。

The *HTTPServer* must be given a *RequestHandlerClass* on instantiation, of which this module provides three different variants:

class http.server.**BaseHTTPRequestHandler**(*request*, *client_address*, *server*)
这个类用于处理到达服务器的 HTTP 请求。它本身无法响应任何实际的 HTTP 请求；它必须被子类化以处理每个请求方法（例如 GET 或 POST）。*BaseHTTPRequestHandler* 提供了许多供子类使用的类和实例变量以及方法。

这个处理程序将解析请求和标头，然后调用特定请求类型对应的方法。方法名称将根据请求来构造。例如，对于请求方法 SPAM，将不带参数地调用 do_SPAM() 方法。所有相关信息会被保存在该处理程序的实际变量中。子类不需要重载或扩展 __init__() 方法。

BaseHTTPRequestHandler 具有下列实例变量:

client_address
包含 (host, port) 形式的指向客户端地址的元组。

server
包含服务器实例。

close_connection
应当在*handle_one_request()* 返回之前设定的布尔值，指明是否要期待另一个请求，还是应当关闭连接。

requestline
包含 HTTP 请求行的字符串表示。末尾的 CRLF 会被去除。该属性应当由*handle_one_request()* 来设定。如果无有效请求行被处理，则它应当被设为空字符串。

command
包含具体的命令（请求类型）。例如 `'GET'`。

path
包含请求路径。

request_version
包含请求的版本字符串。例如 `'HTTP/1.0'`。

headers
Holds an instance of the class specified by the *MessageClass* class variable. This instance parses and manages the headers in the HTTP request. The `parse_headers()` function from *http.client* is used to parse the headers and it requires that the HTTP request provide a valid **RFC 2822** style header.

rfile
An *io.BufferedIOBase* input stream, ready to read from the start of the optional input data.

wfile
Contains the output stream for writing a response back to the client. Proper adherence to the HTTP protocol must be used when writing to this stream in order to achieve successful interoperation with HTTP clients.

在 3.6 版更改: This is an *io.BufferedIOBase* stream.

BaseHTTPRequestHandler has the following attributes:

server_version
Specifies the server software version. You may want to override this. The format is multiple whitespace-separated strings, where each string is of the form name[/version]. For example, `'BaseHTTP/0.2'`.

sys_version
Contains the Python system version, in a form usable by the *version_string* method and the *server_version* class variable. For example, `'Python/1.4'`.

error_message_format
Specifies a format string that should be used by *send_error()* method for building an error response to the client. The string is filled by default with variables from *responses* based on the status code that passed to *send_error()*.

error_content_type
Specifies the Content-Type HTTP header of error responses sent to the client. The default value is `'text/html'`.

protocol_version
This specifies the HTTP protocol version used in responses. If set to `'HTTP/1.1'`, the server will permit HTTP persistent connections; however, your server *must* then include an accurate `Content-Length` header (using *send_header()*) in all of its responses to clients. For backwards compatibility, the setting defaults to `'HTTP/1.0'`.

MessageClass
Specifies an *email.message.Message*-like class to parse HTTP headers. Typically, this is not overridden, and it defaults to `http.client.HTTPMessage`.

responses
This attribute contains a mapping of error code integers to two-element tuples containing a short and long message. For example, `{code: (shortmessage, longmessage)}`. The *shortmessage* is usu-

ally used as the *message* key in an error response, and *longmessage* as the *explain* key. It is used by *send_response_only()* and *send_error()* methods.

A *BaseHTTPRequestHandler* instance has the following methods:

handle()
Calls *handle_one_request()* once (or, if persistent connections are enabled, multiple times) to handle incoming HTTP requests. You should never need to override it; instead, implement appropriate `do_*()` methods.

handle_one_request()
This method will parse and dispatch the request to the appropriate `do_*()` method. You should never need to override it.

handle_expect_100()
When a HTTP/1.1 compliant server receives an `Expect: 100-continue` request header it responds back with a `100 Continue` followed by `200 OK` headers. This method can be overridden to raise an error if the server does not want the client to continue. For e.g. server can chose to send `417 Expectation Failed` as a response header and `return False`.

3.2 新版功能.

send_error(*code*, *message=None*, *explain=None*)
Sends and logs a complete error reply to the client. The numeric *code* specifies the HTTP error code, with *message* as an optional, short, human readable description of the error. The *explain* argument can be used to provide more detailed information about the error; it will be formatted using the *error_message_format* attribute and emitted, after a complete set of headers, as the response body. The *responses* attribute holds the default values for *message* and *explain* that will be used if no value is provided; for unknown codes the default value for both is the string `???`. The body will be empty if the method is HEAD or the response code is one of the following: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`.

在 3.4 版更改: The error response includes a Content-Length header. Added the *explain* argument.

send_response(*code*, *message=None*)
Adds a response header to the headers buffer and logs the accepted request. The HTTP response line is written to the internal buffer, followed by *Server* and *Date* headers. The values for these two headers are picked up from the *version_string()* and *date_time_string()* methods, respectively. If the server does not intend to send any other headers using the *send_header()* method, then *send_response()* should be followed by an *end_headers()* call.

在 3.3 版更改: Headers are stored to an internal buffer and *end_headers()* needs to be called explicitly.

send_header(*keyword*, *value*)
Adds the HTTP header to an internal buffer which will be written to the output stream when either *end_headers()* or *flush_headers()* is invoked. *keyword* should specify the header keyword, with *value* specifying its value. Note that, after the send_header calls are done, *end_headers()* MUST BE called in order to complete the operation.

在 3.2 版更改: Headers are stored in an internal buffer.

send_response_only(*code*, *message=None*)
Sends the response header only, used for the purposes when `100 Continue` response is sent by the server to the client. The headers not buffered and sent directly the output stream.If the *message* is not specified, the HTTP message corresponding the response *code* is sent.

3.2 新版功能.

end_headers()
Adds a blank line (indicating the end of the HTTP headers in the response) to the headers buffer and calls *flush_headers()*.

在 3.2 版更改: The buffered headers are written to the output stream.

flush_headers()
Finally send the headers to the output stream and flush the internal headers buffer.

3.3 新版功能.

log_request(*code='-'*, *size='-'*)
Logs an accepted (successful) request. *code* should specify the numeric HTTP code associated with the response. If a size of the response is available, then it should be passed as the *size* parameter.

log_error(...)
Logs an error when a request cannot be fulfilled. By default, it passes the message to *log_message()*, so it takes the same arguments (*format* and additional values).

log_message(*format*, ...)
Logs an arbitrary message to `sys.stderr`. This is typically overridden to create custom error logging mechanisms. The *format* argument is a standard printf-style format string, where the additional arguments to *log_message()* are applied as inputs to the formatting. The client ip address and current date and time are prefixed to every message logged.

version_string()
Returns the server software's version string. This is a combination of the *server_version* and *sys_version* attributes.

date_time_string(*timestamp=None*)
Returns the date and time given by *timestamp* (which must be `None` or in the format returned by *time.time()*), formatted for a message header. If *timestamp* is omitted, it uses the current date and time.

结果看起来像 `'Sun, 06 Nov 1994 08:49:37 GMT'`。

log_date_time_string()
返回当前的日期和时间，为日志格式化

address_string()
返回客户端的地址

在 3.3 版更改: Previously, a name lookup was performed. To avoid name resolution delays, it now always returns the IP address.

class http.server.**SimpleHTTPRequestHandler**(*request*, *client_address*, *server*)
这个类为当前目录及以下的文件提供服务，直接映射目录结构到 HTTP 请求

A lot of the work, such as parsing the request, is done by the base class *BaseHTTPRequestHandler*. This class implements the *do_GET()* and *do_HEAD()* functions.

The following are defined as class-level attributes of *SimpleHTTPRequestHandler*:

server_version
This will be `"SimpleHTTP/" + __version__`, where `__version__` is defined at the module level.

extensions_map
A dictionary mapping suffixes into MIME types. The default is signified by an empty string, and is considered to be `application/octet-stream`. The mapping is used case-insensitively, and so should contain only lower-cased keys.

The *SimpleHTTPRequestHandler* class defines the following methods:

do_HEAD()
This method serves the `'HEAD'` request type: it sends the headers it would send for the equivalent `GET` request. See the *do_GET()* method for a more complete explanation of the possible headers.

do_GET()
这一请求通过把其解释为当前工作目录的相对路径来映射到本地文件

If the request was mapped to a directory, the directory is checked for a file named `index.html` or `index.htm` (in that order). If found, the file's contents are returned; otherwise a directory listing is generated by calling the `list_directory()` method. This method uses *os.listdir()* to scan the directory, and returns a `404` error response if the *listdir()* fails.

If the request was mapped to a file, it is opened and the contents are returned. Any *OSError* exception in opening the requested file is mapped to a `404`, `'File not found'` error. Otherwise, the content type is guessed by calling the `guess_type()` method, which in turn uses the *extensions_map* variable.

A `'Content-type:'` header with the guessed content type is output, followed by a `'Content-Length:'` header with the file's size and a `'Last-Modified:'` header with the file's modification time.

Then follows a blank line signifying the end of the headers, and then the contents of the file are output. If the file's MIME type starts with `text/` the file is opened in text mode; otherwise binary mode is used.

For example usage, see the implementation of the *test()* function invocation in the *http.server* module.

The *SimpleHTTPRequestHandler* class can be used in the following manner in order to create a very basic webserver serving files relative to the current directory:

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

http.server can also be invoked directly using the `-m` switch of the interpreter with a `port number` argument. Similar to the previous example, this serves files relative to the current directory:

```
python -m http.server 8000
```

By default, server binds itself to all interfaces. The option `-b/--bind` specifies a specific address to which it should bind. For example, the following command causes the server to bind to localhost only:

```
python -m http.server 8000 --bind 127.0.0.1
```

3.4 新版功能: `--bind` argument was introduced.

class http.server.**CGIHTTPRequestHandler**(*request*, *client_address*, *server*)
This class is used to serve either files or output of CGI scripts from the current directory and below. Note that mapping HTTP hierarchic structure to local directory structure is exactly as in *SimpleHTTPRequestHandler*.

注解: CGI scripts run by the *CGIHTTPRequestHandler* class cannot execute redirects (HTTP code 302), because code 200 (script output follows) is sent prior to execution of the CGI script. This pre-empts the status code.

The class will however, run the CGI script, instead of serving it as a file, if it guesses it to be a CGI script. Only directory-based CGI are used —the other common server configuration is to treat special extensions as denoting

CGI scripts.

The `do_GET()` and `do_HEAD()` functions are modified to run CGI scripts and serve the output, instead of serving files, if the request leads to somewhere below the `cgi_directories` path.

The *CGIHTTPRequestHandler* defines the following data member:

cgi_directories
This defaults to `['/cgi-bin', '/htbin']` and describes directories to treat as containing CGI scripts.

The *CGIHTTPRequestHandler* defines the following method:

do_POST()
This method serves the `'POST'` request type, only allowed for CGI scripts. Error 501, "Can only POST to CGI scripts", is output when trying to POST to a non-CGI url.

Note that CGI scripts will be run with UID of user nobody, for security reasons. Problems with the CGI script will be translated to error 403.

CGIHTTPRequestHandler can be enabled in the command line by passing the `--cgi` option:

```
python -m http.server --cgi 8000
```

21.23 http.cookies —HTTP 状态管理

源代码: Lib/http/cookies.py

The *http.cookies* module defines classes for abstracting the concept of cookies, an HTTP state management mechanism. It supports both simple string-only cookies, and provides an abstraction for having any serializable data-type as cookie value.

The module formerly strictly applied the parsing rules described in the **RFC 2109** and **RFC 2068** specifications. It has since been discovered that MSIE 3.0x doesn't follow the character rules outlined in those specs and also many current day browsers and servers have relaxed parsing rules when comes to Cookie handling. As a result, the parsing rules used are a bit less strict.

The character set, *string.ascii_letters*, *string.digits* and `!#$%&'*+-.^_`|~:` denote the set of valid characters allowed by this module in Cookie name (as *key*).

在 3.3 版更改: Allowed ':' as a valid Cookie name character.

注解: On encountering an invalid cookie, *CookieError* is raised, so if your cookie data comes from a browser you should always prepare for invalid data and catch *CookieError* on parsing.

---

exception http.cookies.**CookieError**
Exception failing because of **RFC 2109** invalidity: incorrect attributes, incorrect *Set-Cookie* header, etc.

class http.cookies.**BaseCookie**([*input*])
This class is a dictionary-like object whose keys are strings and whose values are *Morsel* instances. Note that upon setting a key to a value, the value is first converted to a *Morsel* containing the key and the value.

If *input* is given, it is passed to the *load()* method.

class http.cookies.**SimpleCookie**([*input*])
This class derives from *BaseCookie* and overrides `value_decode()` and `value_encode()` to be the identity and *str()* respectively.

参见:

Module *http.cookiejar* HTTP cookie handling for web *clients*. The *http.cookiejar* and *http.cookies* modules do not depend on each other.

RFC 2109 - HTTP State Management Mechanism This is the state management specification implemented by this module.

21.23.1 Cookie 对象

BaseCookie.**value_decode**(*val*)
Return a decoded value from a string representation. Return value can be any type. This method does nothing in *BaseCookie* —it exists so it can be overridden.

BaseCookie.**value_encode**(*val*)
Return an encoded value. *val* can be any type, but return value must be a string. This method does nothing in *BaseCookie* —it exists so it can be overridden.

In general, it should be the case that *value_encode()* and *value_decode()* are inverses on the range of *value_decode*.

BaseCookie.**output**(*attrs=None*, *header='Set-Cookie:'*, *sep='\r\n'*)
Return a string representation suitable to be sent as HTTP headers. *attrs* and *header* are sent to each *Morsel*'s *output()* method. *sep* is used to join the headers together, and is by default the combination `'\r\n'` (CRLF).

BaseCookie.**js_output**(*attrs=None*)
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP headers was sent.

The meaning for *attrs* is the same as in *output()*.

BaseCookie.**load**(*rawdata*)
If *rawdata* is a string, parse it as an `HTTP_COOKIE` and add the values found there as *Morsel*s. If it is a dictionary, it is equivalent to:

```
for k, v in rawdata.items():
    cookie[k] = v
```

21.23.2 Morsel 对象

class http.cookies.**Morsel**
Abstract a key/value pair, which has some **RFC 2109** attributes.

Morsels are dictionary-like objects, whose set of keys is constant —the valid **RFC 2109** attributes, which are

- `expires`
- `path`
- `comment`
- `domain`
- `max-age`
- `secure`
- `version`
- `httponly`

The attribute `httponly` specifies that the cookie is only transferred in HTTP requests, and is not accessible through JavaScript. This is intended to mitigate some forms of cross-site scripting.

The keys are case-insensitive and their default value is `''`.

在 3.5 版更改: `__eq__()` now takes *key* and *value* into account.

`Morsel.`**value**
Cookie 的值。

3.5 版后已移除: assigning to `value`; use *set()* instead.

`Morsel.`**coded_value**
The encoded value of the cookie —this is what should be sent.

3.5 版后已移除: assigning to `coded_value`; use *set()* instead.

`Morsel.`**key**
The name of the cookie.

3.5 版后已移除: assigning to `key`; use *set()* instead.

`Morsel.`**set**(*key*, *value*, *coded_value*)
Set the *key*, *value* and *coded_value* attributes.

3.5 版后已移除: The undocumented *LegalChars* parameter is ignored and will be removed in a future version.

`Morsel.`**isReservedKey**(*K*)
Whether *K* is a member of the set of keys of a *Morsel*.

`Morsel.`**output**(*attrs=None*, *header='Set-Cookie:'*)
Return a string representation of the Morsel, suitable to be sent as an HTTP header. By default, all the attributes are included, unless *attrs* is given, in which case it should be a list of attributes to use. *header* is by default `"Set-Cookie:"`.

`Morsel.`**js_output**(*attrs=None*)
Return an embeddable JavaScript snippet, which, if run on a browser which supports JavaScript, will act the same as if the HTTP header was sent.

The meaning for *attrs* is the same as in *output()*.

`Morsel.`**OutputString**(*attrs=None*)
Return a string representing the Morsel, without any surrounding HTTP or JavaScript.

The meaning for *attrs* is the same as in *output()*.

`Morsel.`**update**(*values*)
Update the values in the Morsel dictionary with the values in the dictionary *values*. Raise an error if any of the keys in the *values* dict is not a valid **RFC 2109** attribute.

在 3.5 版更改: an error is raised for invalid keys.

`Morsel.`**copy**(*value*)
Return a shallow copy of the Morsel object.

在 3.5 版更改: return a Morsel object instead of a dict.

`Morsel.`**setdefault**(*key*, *value=None*)
Raise an error if key is not a valid **RFC 2109** attribute, otherwise behave the same as *dict.setdefault()*.

21.23.3 示例

The following example demonstrates how to use the *http.cookies* module.

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

21.24 http.cookiejar ——HTTP 客户端的 Cookie 处理

源代码： Lib/http/cookiejar.py

---

http.cookiejar 模块定义了用于自动处理 HTTP cookie 的类。这对访问需要小段数据——*cookies* 的网站很有用，这些数据由 Web 服务器的 HTTP 响应在客户端计算机上设置，然后在以后的 HTTP 请求中返回给服务器。

常规的 Netscape Cookie 协议和 **RFC 2965** 定义的协议都可以处理。RFC 2965 处理默认情况下处于关闭状态。**RFC 2109** cookie 被解析为 Netscape cookie，随后根据有效的‘policy’被视为 Netscape 或 RFC 2965 cookie。请注意，Internet 上的大多数 cookie 是 Netscape cookie。*http.cookiejar* 尝试遵循事实上的 Netscape cookie 协议（与原始 Netscape 规范中所设定的协议大不相同），包括注意 `max-age` 和 `port` RFC 2965 引入的 cookie 属性。

注解： 在 *Set-Cookie* 和 *Set-Cookie2* 头中找到的各种命名参数通常指 *attributes*。为了不与 Python 属性相混淆，模块文档使用 *cookie-attribute* 代替。

此模块定义了以下异常：

exception http.cookiejar.**LoadError**
FileCookieJar 实例在从文件加载 cookies 出错时抛出这个异常。*LoadError* 是 *OSError* 的一个子类。

在 3.3 版更改: LoadError 成为 *OSError* 的子类而不是 *IOError*。

提供了以下类：

class http.cookiejar.**CookieJar**(*policy=None*)
policy 是实现了 *CookiePolicy* 接口的一个对象。

CookieJar 类储存 HTTP cookies。它从 HTTP 请求提取 cookies，并在 HTTP 响应中返回它们。*CookieJar* 实例在必要时自动处理包含 cookie 的到期情况。子类还负责储存和从文件或数据库中查找 cookies。

class http.cookiejar.**FileCookieJar**(*filename*, *delayload=None*, *policy=None*)
policy 是实现了 *CookiePolicy* 接口的一个对象。对于其他参数，参考相应属性的文档。

一个可以从磁盘文件中加载或保存 cookie 的 *CookieJar*。cookie **不会**从指定的文件中加载，直到调用 *load()* 或 *revert()* 方法。该类的子类文档请参阅 *FileCookieJar subclasses and co-operation with web browsers*。

class http.cookiejar.**CookiePolicy**
此类负责确定是否应从服务器接受每个 cookie 或将其返回给服务器。

class http.cookiejar.**DefaultCookiePolicy**(*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*)
构造器的参数应当只作为关键字参数传入。*blocked_domains* 是一个域名序列，程序将绝不接受其 cookie 也绝不向其返回 cookie。*allowed_domains* 如果不为 *None*，则也应是一个域名序列，程序将只对其中的

域名接受和返回 cookie。对于所有其他参数，请参阅*CookiePolicy* 和*DefaultCookiePolicy* 对象的文档。

DefaultCookiePolicy implements the standard accept / reject rules for Netscape and **RFC 2965** cookies. By default, **RFC 2109** cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) are treated according to the RFC 2965 rules. However, if RFC 2965 handling is turned off or *rfc2109_as_netscape* is `True`, RFC 2109 cookies are 'downgraded' by the *CookieJar* instance to Netscape cookies, by setting the `version` attribute of the *Cookie* instance to 0. *DefaultCookiePolicy* also provides some parameters to allow some fine-tuning of policy.

class `http.cookiejar.`**Cookie**

This class represents Netscape, **RFC 2109** and **RFC 2965** cookies. It is not expected that users of *http.cookiejar* construct their own *Cookie* instances. Instead, if necessary, call `make_cookies()` on a *CookieJar* instance.

参见:

Module *urllib.request* URL opening with automatic cookie handling.

Module *http.cookies* HTTP cookie classes, principally useful for server-side code. The *http.cookiejar* and *http.cookies* modules do not depend on each other.

https://curl.haxx.se/rfc/cookie_spec.html The specification of the original Netscape cookie protocol. Though this is still the dominant protocol, the 'Netscape cookie protocol' implemented by all the major browsers (and *http.cookiejar*) only bears a passing resemblance to the one sketched out in `cookie_spec.html`.

RFC 2109 - HTTP State Management Mechanism Obsoleted by **RFC 2965**. Uses *Set-Cookie* with version=1.

RFC 2965 - HTTP State Management Mechanism The Netscape protocol with the bugs fixed. Uses *Set-Cookie2* in place of *Set-Cookie*. Not widely used.

http://kristol.org/cookie/errata.html Unfinished errata to **RFC 2965**.

RFC 2964 - Use of HTTP State Management

21.24.1 CookieJar 和 FileCookieJar 对象

CookieJar objects support the *iterator* protocol for iterating over contained *Cookie* objects.

CookieJar has the following methods:

`CookieJar.`**add_cookie_header**(*request*)

Add correct *Cookie* header to *request*.

If policy allows (ie. the `rfc2965` and `hide_cookie2` attributes of the *CookieJar*'s *CookiePolicy* instance are true and false respectively), the *Cookie2* header is also added when appropriate.

The *request* object (usually a `urllib.request..Request` instance) must support the methods `get_full_url()`, `get_host()`, `get_type()`, `unverifiable()`, `has_header()`, `get_header()`, `header_items()`, `add_unredirected_header()` and `origin_req_host` attribute as documented by *urllib.request*.

在 3.3 版更改: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`CookieJar.`**extract_cookies**(*response*, *request*)

Extract cookies from HTTP *response* and store them in the *CookieJar*, where allowed by policy.

The *CookieJar* will look for allowable *Set-Cookie* and *Set-Cookie2* headers in the *response* argument, and store cookies as appropriate (subject to the *CookiePolicy.set_ok()* method's approval).

The *response* object (usually the result of a call to *urllib.request.urlopen()*, or similar) should support an `info()` method, which returns an *email.message.Message* instance.

The *request* object (usually a *urllib.request.Request* instance) must support the methods `get_full_url()`, `get_host()`, `unverifiable()`, and `origin_req_host` attribute, as documented by *urllib.request*. The request is used to set default values for cookie-attributes as well as for checking that the cookie is allowed to be set.

在 3.3 版更改: *request* object needs `origin_req_host` attribute. Dependency on a deprecated method `get_origin_req_host()` has been removed.

`CookieJar.`**`set_policy`**(*policy*)
Set the *CookiePolicy* instance to be used.

`CookieJar.`**`make_cookies`**(*response*, *request*)
Return sequence of *Cookie* objects extracted from *response* object.

See the documentation for *extract_cookies()* for the interfaces required of the *response* and *request* arguments.

`CookieJar.`**`set_cookie_if_ok`**(*cookie*, *request*)
Set a *Cookie* if policy says it's OK to do so.

`CookieJar.`**`set_cookie`**(*cookie*)
Set a *Cookie*, without checking with policy to see whether or not it should be set.

`CookieJar.`**`clear`**([*domain*[, *path*[, *name*]]])
Clear some cookies.

If invoked without arguments, clear all cookies. If given a single argument, only cookies belonging to that *domain* will be removed. If given two arguments, cookies belonging to the specified *domain* and URL *path* are removed. If given three arguments, then the cookie with the specified *domain*, *path* and *name* is removed.

Raises *KeyError* if no matching cookie exists.

`CookieJar.`**`clear_session_cookies`**()
Discard all session cookies.

Discards all contained cookies that have a true `discard` attribute (usually because they had either no `max-age` or `expires` cookie-attribute, or an explicit `discard` cookie-attribute). For interactive browsers, the end of a session usually corresponds to closing the browser window.

Note that the `save()` method won't save session cookies anyway, unless you ask otherwise by passing a true *ignore_discard* argument.

FileCookieJar implements the following additional methods:

`FileCookieJar.`**`save`**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Save cookies to a file.

This base class raises *NotImplementedError*. Subclasses may leave this method unimplemented.

filename is the name of file in which to save cookies. If *filename* is not specified, `self.filename` is used (whose default is the value passed to the constructor, if any); if `self.filename` is *None*, *ValueError* is raised.

ignore_discard: save even cookies set to be discarded. *ignore_expires*: save even cookies that have expired

The file is overwritten if it already exists, thus wiping all the cookies it contains. Saved cookies can be restored later using the *load()* or *revert()* methods.

`FileCookieJar.`**`load`**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Load cookies from a file.

Old cookies are kept unless overwritten by newly loaded ones.

Arguments are as for *save()*.

The named file must be in the format understood by the class, or *LoadError* will be raised. Also, *OSError* may be raised, for example if the file does not exist.

在 3.3 版更改: 过去触发的*IOError*，现在是*OSError*的别名。

FileCookieJar.**revert**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
Clear all cookies and reload cookies from a saved file.

revert() can raise the same exceptions as *load()*. If there is a failure, the object's state will not be altered.

FileCookieJar instances have the following public attributes:

FileCookieJar.**filename**
Filename of default file in which to keep cookies. This attribute may be assigned to.

FileCookieJar.**delayload**
If true, load cookies lazily from disk. This attribute should not be assigned to. This is only a hint, since this only affects performance, not behaviour (unless the cookies on disk are changing). A *CookieJar* object may ignore it. None of the *FileCookieJar* classes included in the standard library lazily loads cookies.

21.24.2 FileCookieJar subclasses and co-operation with web browsers

The following *CookieJar* subclasses are provided for reading and writing.

class http.cookiejar.**MozillaCookieJar**(*filename*, *delayload=None*, *policy=None*)
A *FileCookieJar* that can load from and save cookies to disk in the Mozilla `cookies.txt` file format (which is also used by the Lynx and Netscape browsers).

注解: This loses information about **RFC 2965** cookies, and also about newer or non-standard cookie-attributes such as `port`.

警告: Back up your cookies before saving if you have cookies whose loss / corruption would be inconvenient (there are some subtleties which may lead to slight changes in the file over a load / save round-trip).

Also note that cookies saved while Mozilla is running will get clobbered by Mozilla.

class http.cookiejar.**LWPCookieJar**(*filename*, *delayload=None*, *policy=None*)
A *FileCookieJar* that can load from and save cookies to disk in format compatible with the libwww-perl library's `Set-Cookie3` file format. This is convenient if you want to store cookies in a human-readable file.

21.24.3 CookiePolicy 对象

Objects implementing the *CookiePolicy* interface have the following methods:

CookiePolicy.**set_ok**(*cookie*, *request*)
Return boolean value indicating whether cookie should be accepted from server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.extract_cookies()*.

CookiePolicy.**return_ok**(*cookie*, *request*)
Return boolean value indicating whether cookie should be returned to server.

cookie is a *Cookie* instance. *request* is an object implementing the interface defined by the documentation for *CookieJar.add_cookie_header()*.

CookiePolicy.**domain_return_ok**(*domain*, *request*)
Return false if cookies should not be returned, given cookie domain.

This method is an optimization. It removes the need for checking every cookie with a particular domain (which might involve reading many files). Returning true from *domain_return_ok()* and *path_return_ok()* leaves all the work to *return_ok()*.

If *domain_return_ok()* returns true for the cookie domain, *path_return_ok()* is called for the cookie path. Otherwise, *path_return_ok()* and *return_ok()* are never called for that cookie domain. If *path_return_ok()* returns true, *return_ok()* is called with the *Cookie* object itself for a full check. Otherwise, *return_ok()* is never called for that cookie path.

Note that *domain_return_ok()* is called for every *cookie* domain, not just for the *request* domain. For example, the function might be called with both `".example.com"` and `"www.example.com"` if the request domain is `"www.example.com"`. The same goes for *path_return_ok()*.

The *request* argument is as documented for *return_ok()*.

CookiePolicy.**path_return_ok**(*path*, *request*)
Return false if cookies should not be returned, given cookie path.

See the documentation for *domain_return_ok()*.

In addition to implementing the methods above, implementations of the *CookiePolicy* interface must also supply the following attributes, indicating which protocols should be used, and how. All of these attributes may be assigned to.

CookiePolicy.**netscape**
Implement Netscape protocol.

CookiePolicy.**rfc2965**
Implement **RFC 2965** protocol.

CookiePolicy.**hide_cookie2**
Don't add *Cookie2* header to requests (the presence of this header indicates to the server that we understand **RFC 2965** cookies).

The most useful way to define a *CookiePolicy* class is by subclassing from *DefaultCookiePolicy* and overriding some or all of the methods above. *CookiePolicy* itself may be used as a 'null policy' to allow setting and receiving any and all cookies (this is unlikely to be useful).

21.24.4 DefaultCookiePolicy 对象

Implements the standard rules for accepting and returning cookies.

Both **RFC 2965** and Netscape cookies are covered. RFC 2965 handling is switched off by default.

The easiest way to provide your own policy is to override this class and call its methods in your overridden implementations before adding your own additional checks:

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
```

(下页继续)

(续上页)

```
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

In addition to the features required to implement the *CookiePolicy* interface, this class allows you to block and allow domains from setting and receiving cookies. There are also some strictness switches that allow you to tighten up the rather loose Netscape protocol rules a little bit (at the cost of blocking some benign cookies).

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the *blocked_domains* constructor argument, and `blocked_domains()` and `set_blocked_domains()` methods (and the corresponding argument and methods for *allowed_domains*). If you set a whitelist, you can turn it off again by setting it to *None*.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, `"example.com"` matches a blacklist entry of `"example.com"`, but `"www.example.com"` does not. Domains that do start with a dot are matched by more specific domains too. For example, both `"www.example.com"` and `"www.coyote.example.com"` match `".example.com"` (but `"example.com"` itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains `"192.168.1.2"` and `".168.1.2"`, 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy implements the following additional methods:

`DefaultCookiePolicy.`**`blocked_domains`**`()`
Return the sequence of blocked domains (as a tuple).

`DefaultCookiePolicy.`**`set_blocked_domains`**`(`*blocked_domains*`)`
Set the sequence of blocked domains.

`DefaultCookiePolicy.`**`is_blocked`**`(`*domain*`)`
Return whether *domain* is on the blacklist for setting or receiving cookies.

`DefaultCookiePolicy.`**`allowed_domains`**`()`
Return *None*, or the sequence of allowed domains (as a tuple).

`DefaultCookiePolicy.`**`set_allowed_domains`**`(`*allowed_domains*`)`
Set the sequence of allowed domains, or *None*.

`DefaultCookiePolicy.`**`is_not_allowed`**`(`*domain*`)`
Return whether *domain* is not on the whitelist for setting or receiving cookies.

DefaultCookiePolicy instances have the following attributes, which are all initialised from the constructor arguments of the same name, and which may all be assigned to.

`DefaultCookiePolicy.`**`rfc2109_as_netscape`**
If true, request that the *CookieJar* instance downgrade **RFC 2109** cookies (ie. cookies received in a *Set-Cookie* header with a version cookie-attribute of 1) to Netscape cookies by setting the version attribute of the *Cookie* instance to 0. The default value is *None*, in which case RFC 2109 cookies are downgraded if and only if **RFC 2965** handling is turned off. Therefore, RFC 2109 cookies are downgraded by default.

General strictness switches:

`DefaultCookiePolicy.`**`strict_domain`**
Don't allow sites to set two-component domains with country-code top-level domains like `.co.uk`, `.gov.uk`, `.co.nz`.etc. This is far from perfect and isn't guaranteed to work!

RFC 2965 protocol strictness switches:

`DefaultCookiePolicy.`**`strict_rfc2965_unverifiable`**
Follow **RFC 2965** rules on unverifiable transactions (usually, an unverifiable transaction is one resulting from a

redirect or a request for an image hosted on another site). If this is false, cookies are *never* blocked on the basis of verifiability

Netscape protocol strictness switches:

`DefaultCookiePolicy.`**`strict_ns_unverifiable`**
Apply **RFC 2965** rules on unverifiable transactions even to Netscape cookies.

`DefaultCookiePolicy.`**`strict_ns_domain`**
Flags indicating how strict to be with domain-matching rules for Netscape cookies. See below for acceptable values.

`DefaultCookiePolicy.`**`strict_ns_set_initial_dollar`**
Ignore cookies in Set-Cookie: headers that have names starting with `'$'`.

`DefaultCookiePolicy.`**`strict_ns_set_path`**
Don't allow setting cookies whose path doesn't path-match request URI.

`strict_ns_domain` is a collection of flags. Its value is constructed by or-ing together (for example, `DomainStrictNoDots|DomainStrictNonDomain` means both flags are set).

`DefaultCookiePolicy.`**`DomainStrictNoDots`**
When setting cookies, the 'host prefix' must not contain a dot (eg. `www.foo.bar.com` can't set a cookie for `.bar.com`, because `www.foo` contains a dot).

`DefaultCookiePolicy.`**`DomainStrictNonDomain`**
Cookies that did not explicitly specify a `domain` cookie-attribute can only be returned to a domain equal to the domain that set the cookie (eg. `spam.example.com` won't be returned cookies from `example.com` that had no `domain` cookie-attribute).

`DefaultCookiePolicy.`**`DomainRFC2965Match`**
When setting cookies, require a full **RFC 2965** domain-match.

The following attributes are provided for convenience, and are the most useful combinations of the above flags:

`DefaultCookiePolicy.`**`DomainLiberal`**
Equivalent to 0 (ie. all of the above Netscape domain strictness flags switched off).

`DefaultCookiePolicy.`**`DomainStrict`**
Equivalent to `DomainStrictNoDots|DomainStrictNonDomain`.

21.24.5 Cookie 对象

`Cookie` instances have Python attributes roughly corresponding to the standard cookie-attributes specified in the various cookie standards. The correspondence is not one-to-one, because there are complicated rules for assigning default values, because the `max-age` and `expires` cookie-attributes contain equivalent information, and because **RFC 2109** cookies may be 'downgraded' by *`http.cookiejar`* from version 1 to version 0 (Netscape) cookies.

Assignment to these attributes should not be necessary other than in rare circumstances in a *`CookiePolicy`* method. The class does not enforce internal consistency, so you should know what you're doing if you do that.

`Cookie.`**`version`**
Integer or *`None`*. Netscape cookies have *`version`* 0. **RFC 2965** and **RFC 2109** cookies have a `version` cookie-attribute of 1. However, note that *`http.cookiejar`* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *`version`* is 0.

`Cookie.`**`name`**
Cookie name (a string).

`Cookie.`**`value`**
Cookie value (a string), or *`None`*.

Cookie.**port**
String representing a port or a set of ports (eg. '80', or '80,8080'), or *None*.

Cookie.**path**
Cookie path (a string, eg. '/acme/rocket_launchers').

Cookie.**secure**
True if cookie should only be returned over a secure connection.

Cookie.**expires**
Integer expiry date in seconds since epoch, or *None*. See also the *is_expired()* method.

Cookie.**discard**
True if this is a session cookie.

Cookie.**comment**
String comment from the server explaining the function of this cookie, or *None*.

Cookie.**comment_url**
URL linking to a comment from the server explaining the function of this cookie, or *None*.

Cookie.**rfc2109**
True if this cookie was received as an **RFC 2109** cookie (ie. the cookie arrived in a *Set-Cookie* header, and the value of the Version cookie-attribute in that header was 1). This attribute is provided because *http.cookiejar* may 'downgrade' RFC 2109 cookies to Netscape cookies, in which case *version* is 0.

Cookie.**port_specified**
True if a port or set of ports was explicitly specified by the server (in the *Set-Cookie* / *Set-Cookie2* header).

Cookie.**domain_specified**
True if a domain was explicitly specified by the server.

Cookie.**domain_initial_dot**
True if the domain explicitly specified by the server began with a dot ('.').

Cookies may have additional non-standard cookie-attributes. These may be accessed using the following methods:

Cookie.**has_nonstandard_attr**(*name*)
Return true if cookie has the named cookie-attribute.

Cookie.**get_nonstandard_attr**(*name*, *default=None*)
If cookie has the named cookie-attribute, return its value. Otherwise, return *default*.

Cookie.**set_nonstandard_attr**(*name*, *value*)
Set the value of the named cookie-attribute.

The *Cookie* class also defines the following method:

Cookie.**is_expired**(*now=None*)
True if cookie has passed the time at which the server requested it should expire. If *now* is given (in seconds since the epoch), return whether the cookie has expired at the specified time.

21.24.6 例子

The first example shows the most common usage of *http.cookiejar*:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

This example illustrates how to open a URL using your Netscape, Mozilla, or Lynx cookies (assumes Unix/Netscape convention for location of the cookies file):

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

The next example illustrates the use of *DefaultCookiePolicy*. Turn on **RFC 2965** cookies, be more strict about domains when setting and returning Netscape cookies, and block some domains from setting cookies or having them returned:

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

21.25 xmlrpc —XMLRPC 服务端与客户端模块

XML-RPC 是一种远程过程调用方法，它使用通过 HTTP 传递的 XML 作为载体。有了它，客户端可以在远程服务器上调用带参数的方法（服务器以 URI 命名）并获取结构化的数据。

`xmlrpc` 是一个集合了 XML-RPC 服务端与客户端实现模块的包。这些模块是:

- *xmlrpc.client*
- *xmlrpc.server*

21.26 xmlrpc.client —XML-RPC 客户端访问

源代码: Lib/xmlrpc/client.py

XML-RPC 是一种远程过程调用方法，它以使用 HTTP(S) 传递的 XML 作为载体。通过它，客户端可以在远程服务器（服务器以 URI 指明）上调用带参数的方法并获取结构化的数据。本模块支持编写 XML-RPC 客户端代码；它会处理在通用 Python 对象和 XML 之间进行在线翻译的所有细节。

警告: *xmlrpc.client* 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

在 3.5 版更改: 对于 HTTPS URI，现在*xmlrpc.client* 默认会执行所有必要的证书和主机名检查。

class xmlrpc.client.**ServerProxy**(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, *, *context=None*)

在 3.3 版更改: The *use_builtin_types* flag was added.

A *ServerProxy* instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for https: URLs and an internal HTTP `Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag.

The following parameters govern the use of the returned proxy instance. If *allow_none* is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a *TypeError*. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The *use_builtin_types* flag can be used to cause date/time values to be presented as *datetime.datetime* objects and binary data to be presented as *bytes* objects; this flag is false by default. *datetime.datetime*, *bytes* and *bytearray* objects may be passed to calls. The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP 'Authorization' header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password. If an HTTPS URL is provided, *context* may be *ssl.SSLContext* and configures the SSL settings of the underlying HTTPS connection.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):

XML-RPC 类型	Python 类型
`boolean`	*bool*
`int`, `i1`, `i2`, `i4`, `i8` 或者 `biginteger`	*int* 的范围从 -2147483648 到 2147483647。值将获得 `<int>` 标志。
`double` 或 `float`	*float*. 值将获得 `<double>` 标志.
`string`	*str*
`array`	*list* 或*tuple* 包含整合元素。数组以*lists* 形式返回。
`struct`	*dict*. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their *__dict__* attribute is transmitted.
`dateTime.iso8601`	*DateTime* 或*datetime.datetime*。返回的类型取决于 *use_builtin_types* 和 *use_datetime* 标志的值。
`base64`	*Binary*, *bytes* 或*bytearray*。返回的类型取决于 *use_builtin_types* 标志的值。
`nil`	`None` 常量。仅当 *allow_none* 为 true 时才允许传递。
`bigdecimal`	*decimal.Decimal*. 仅返回类型。

This is the full set of data types supported by XML-RPC. Method calls may also raise a special *Fault* instance, used to signal XML-RPC server errors, or *ProtocolError* used to signal an error in the HTTP/HTTPS transport layer. Both *Fault* and *ProtocolError* derive from a base class called `Error`. Note that the xmlrpc client module currently does not marshal instances of subclasses of built-in types.

When passing strings, characters special to XML such as <, >, and & will be automatically escaped. However, it's the caller's responsibility to ensure that the string is free of characters that aren't allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn't well-formed XML. If you have to pass arbitrary bytes via XML-RPC, use *bytes* or *bytearray* classes or the *Binary* wrapper class described below.

`Server` is retained as an alias for *ServerProxy* for backwards compatibility. New code should use *ServerProxy*.

在 3.5 版更改: Added the *context* argument.

在 3.6 版更改: Added support of type tags with prefixes (e.g. `ex:nil`). Added support of unmarshalling additional types used by Apache XML-RPC implementation for numerics: `i1`, `i2`, `i8`, `biginteger`, `float` and `bigdecimal`. See http://ws.apache.org/xmlrpc/types.html for a description.

参见:

XML-RPC HOWTO A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.

XML-RPC Introspection Describes the XML-RPC protocol extension for introspection.

XML-RPC Specification The official specification.

Unofficial XML-RPC Errata Fredrik Lundh's "unofficial errata, intended to clarify certain details in the XML-RPC specification, as well as hint at 'best practices' to use when designing your own XML-RPC implementations."

21.26.1 ServerProxy 对象

A *ServerProxy* instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a *Fault* or *ProtocolError* object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` attribute:

`ServerProxy.system.`**`listMethods`**`()`
This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

`ServerProxy.system.`**`methodSignature`**`(`*name*`)`
This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers and returns a string, its signature is "string, int, int, int".

If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

ServerProxy.system.**methodHelp**(*name*)
: This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.

在 3.5 版更改: Instances of *ServerProxy* support the *context manager* protocol for closing the underlying transport.

A working example follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

21.26.2 DateTime 对象

class xmlrpc.client.**DateTime**
: This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a *datetime.datetime* instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

  decode(*string*)
  : Accept a string as the instance's new time value.

  encode(*out*)
  : Write the XML-RPC encoding of this *DateTime* item to the *out* stream object.

  It also supports certain of Python's built-in operators through rich comparison and __repr__() methods.

A working example follows. The server code:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

21.26.3 Binary 对象

class xmlrpc.client.**Binary**

This class may be initialized from bytes data (which may include NULs). The primary access to the content of a *Binary* object is provided by an attribute:

data

The binary data encapsulated by the *Binary* instance. The data is provided as a *bytes* object.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

decode(*bytes*)

Accept a base64 *bytes* object and decode it as the instance's new data.

encode(*out*)

Write the XML-RPC base 64 encoding of this binary item to the *out* stream object.

The encoded data will have newlines every 76 characters as per **RFC 2045 section 6.8**, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through `__eq__()` and `__ne__()` methods.

Example usage of the binary objects. We're going to transfer an image over XMLRPC:

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

The client gets the image and saves it to a file:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

21.26.4 Fault 对象

class xmlrpc.client.**Fault**

A *Fault* object encapsulates the content of an XML-RPC fault tag. Fault objects have the following attributes:

faultCode

A string indicating the fault type.

faultString

A string containing a diagnostic message associated with the fault.

In the following example we're going to intentionally cause a *Fault* by returning a complex type object. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

21.26.5 ProtocolError 对象

class xmlrpc.client.**ProtocolError**

A *ProtocolError* object describes a protocol error in the underlying transport layer (such as a 404 'not found' error if the server named by the URI does not exist). It has the following attributes:

url

The URI or URL that triggered the error.

errcode

The error code.

errmsg

The error message or diagnostic string.

headers

A dict containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we're going to intentionally cause a *ProtocolError* by providing an invalid URI:

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

21.26.6 MultiCall 对象

The *MultiCall* object provides a way to encapsulate multiple calls to a remote server into a single request[1].

class xmlrpc.client.**MultiCall**(*server*)

Create an object used to boxcar method calls. *server* is the eventual target of the call. Calls can be made to the result object, but they will immediately return `None`, and only store the call name and parameters in the *MultiCall* object. Calling the object itself causes all stored calls to be transmitted as a single `system.multicall` request. The result of this call is a *generator*; iterating over this generator yields the individual results.

A usage example of this class follows. The server code:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

The client code for the preceding server:

```
import xmlrpc.client
```

(下页继续)

[1] This approach has been first presented in a discussion on xmlrpc.com.

(续上页)

```
proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

21.26.7 Convenience Functions

xmlrpc.client.**dumps**(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

Convert *params* into an XML-RPC request. or into a response if *methodresponse* is true. *params* can be either a tuple of arguments or an instance of the `Fault` exception class. If *methodresponse* is true, only a single value can be returned, meaning that *params* must be of length 1. *encoding*, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's `None` value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for *allow_none*.

xmlrpc.client.**loads**(*data*, *use_datetime=False*, *use_builtin_types=False*)

Convert an XML-RPC request or response into Python objects, a `(params, methodname)`. *params* is a tuple of argument; *methodname* is a string, or `None` if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a `Fault` exception. The *use_builtin_types* flag can be used to cause date/time values to be presented as `datetime.datetime` objects and binary data to be presented as `bytes` objects; this flag is false by default.

The obsolete *use_datetime* flag is similar to *use_builtin_types* but it applies only to date/time values.

在 3.3 版更改: The *use_builtin_types* flag was added.

21.26.8 Example of Client Usage

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

To access an XML-RPC server through a HTTP proxy, you need to define a custom transport. The following example shows how:

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):
```

(下页继续)

(续上页)

```
    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com', transport=transport)
print(server.examples.getStateName(41))
```

21.26.9 Example of Client and Server Usage

See *SimpleXMLRPCServer Example*.

备注

21.27 xmlrpc.server —基本 XML-RPC 服务器

源代码: Lib/xmlrpc/server.py

The *xmlrpc.server* module provides a basic server framework for XML-RPC servers written in Python. Servers can either be free standing, using *SimpleXMLRPCServer*, or embedded in a CGI environment, using *CGIXMLRPCRequestHandler*.

> **警告:** The *xmlrpc.server* module is not secure against maliciously constructed data. If you need to parse untrusted or unauthenticated data see *XML 漏洞*.

class xmlrpc.server.**SimpleXMLRPCServer**(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

Create a new server instance. This class provides methods for registration of functions that can be called by the XML-RPC protocol. The *requestHandler* parameter should be a factory for request handler instances; it defaults to *SimpleXMLRPCRequestHandler*. The *addr* and *requestHandler* parameters are passed to the *socketserver.TCPServer* constructor. If *logRequests* is true (the default), requests will be logged; setting this parameter to false will turn off logging. The *allow_none* and *encoding* parameters are passed on to *xmlrpc.client* and control the XML-RPC responses that will be returned from the server. The *bind_and_activate* parameter controls whether `server_bind()` and `server_activate()` are called immediately by the constructor; it defaults to true. Setting it to false allows code to manipulate the *allow_reuse_address* class variable before the address is bound. The *use_builtin_types* parameter is passed to the *loads()* function and controls which types are processed when date/times values or binary data are received; it defaults to false.

在 3.3 版更改: The *use_builtin_types* flag was added.

class xmlrpc.server.**CGIXMLRPCRequestHandler**(*allow_none=False*, *encoding=None*, *use_builtin_types=False*)

Create a new instance to handle XML-RPC requests in a CGI environment. The *allow_none* and *encoding* parameters are passed on to `xmlrpc.client` and control the XML-RPC responses that will be returned from the server. The *use_builtin_types* parameter is passed to the `loads()` function and controls which types are processed when date/times values or binary data are received; it defaults to false.

在 3.3 版更改: The *use_builtin_types* flag was added.

class xmlrpc.server.**SimpleXMLRPCRequestHandler**

Create a new request handler instance. This request handler supports `POST` requests and modifies logging so that the *logRequests* parameter to the `SimpleXMLRPCServer` constructor parameter is honored.

21.27.1 SimpleXMLRPCServer Objects

The `SimpleXMLRPCServer` class is based on `socketserver.TCPServer` and provides a means of creating simple, stand alone XML-RPC servers.

SimpleXMLRPCServer.**register_function**(*function*, *name=None*)

Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with *function*, otherwise `function.__name__` will be used. *name* can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

SimpleXMLRPCServer.**register_instance**(*instance*, *allow_dotted_names=False*)

Register an object which is used to expose method names which have not been registered using `register_function()`. If *instance* contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request. Its API is `def _dispatch(self, method, params)` (note that *params* does not represent a variable argument list). If it calls an underlying function to perform its task, that function is called as `func(*params)`, expanding the parameter list. The return value from `_dispatch()` is returned to the client as the result. If *instance* does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method.

If the optional *allow_dotted_names* argument is true and the instance does not have a `_dispatch()` method, then if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

> **警告:** Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this option on a secure, closed network.

SimpleXMLRPCServer.**register_introspection_functions**()

Registers the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

SimpleXMLRPCServer.**register_multicall_functions**()

Registers the XML-RPC multicall function system.multicall.

SimpleXMLRPCRequestHandler.**rpc_paths**

An attribute value that must be a tuple listing valid path portions of the URL for receiving XML-RPC requests. Requests posted to other paths will result in a 404 "no such page" HTTP error. If this tuple is empty, all paths will be considered valid. The default value is `('/', '/RPC2')`.

SimpleXMLRPCServer Example

Server code:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(("localhost", 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x,y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```

The following client code will call the methods made available by the preceding server:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

The following example included in the `Lib/xmlrpc/server.py` module shows a server allowing dotted names and registering a multicall function.

> **警告：** Enabling the *allow_dotted_names* option allows intruders to access your module's global variables and may allow intruders to execute arbitrary code on your machine. Only use this example only within a secure, closed network.

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

This ExampleService demo can be invoked from the command line:

```
python -m xmlrpc.server
```

The client that interacts with the above server is included in *Lib/xmlrpc/client.py*:

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

This client which interacts with the demo XMLRPC server can be invoked as:

```
python -m xmlrpc.client
```

21.27.2 CGIXMLRPCRequestHandler

The *CGIXMLRPCRequestHandler* class can be used to handle XML-RPC requests sent to Python CGI scripts.

CGIXMLRPCRequestHandler.**register_function**(*function*, *name=None*)
Register a function that can respond to XML-RPC requests. If *name* is given, it will be the method name associated with function, otherwise *function.__name__* will be used. *name* can be either a normal or Unicode string, and may contain characters not legal in Python identifiers, including the period character.

CGIXMLRPCRequestHandler.**register_instance**(*instance*)
Register an object which is used to expose method names which have not been registered using *register_function()*. If instance contains a `_dispatch()` method, it is called with the requested method name and the parameters from the request; the return value is returned to the client as the result. If instance does not have a `_dispatch()` method, it is searched for an attribute matching the name of the requested method; if the requested method name contains periods, each component of the method name is searched for individually, with the effect that a simple hierarchical search is performed. The value found from this search is then called with the parameters from the request, and the return value is passed back to the client.

CGIXMLRPCRequestHandler.**register_introspection_functions**()
Register the XML-RPC introspection functions `system.listMethods`, `system.methodHelp` and `system.methodSignature`.

CGIXMLRPCRequestHandler.**register_multicall_functions**()
Register the XML-RPC multicall function `system.multicall`.

CGIXMLRPCRequestHandler.**handle_request**(*request_text=None*)
Handle an XML-RPC request. If *request_text* is given, it should be the POST data provided by the HTTP server, otherwise the contents of stdin will be used.

示例:

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

21.27.3 Documenting XMLRPC server

These classes extend the above classes to serve HTML documentation in response to HTTP GET requests. Servers can either be free standing, using *DocXMLRPCServer*, or embedded in a CGI environment, using *DocCGIXMLRPCRequestHandler*.

class xmlrpc.server.**DocXMLRPCServer**(*addr*, *requestHandler=DocXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=True*)
Create a new server instance. All parameters have the same meaning as for *SimpleXMLRPCServer*; *requestHandler* defaults to *DocXMLRPCRequestHandler*.

在 3.3 版更改: The *use_builtin_types* flag was added.

class xmlrpc.server.**DocCGIXMLRPCRequestHandler**
Create a new instance to handle XML-RPC requests in a CGI environment.

class xmlrpc.server.**DocXMLRPCRequestHandler**
Create a new request handler instance. This request handler supports XML-RPC POST requests, documentation GET requests, and modifies logging so that the *logRequests* parameter to the *DocXMLRPCServer* constructor parameter is honored.

21.27.4 DocXMLRPCServer Objects

The *DocXMLRPCServer* class is derived from *SimpleXMLRPCServer* and provides a means of creating self-documenting, stand alone XML-RPC servers. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocXMLRPCServer.**set_server_title**(*server_title*)
Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocXMLRPCServer.**set_server_name**(*server_name*)
Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocXMLRPCServer.**set_server_documentation**(*server_documentation*)
Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

21.27.5 DocCGIXMLRPCRequestHandler

The *DocCGIXMLRPCRequestHandler* class is derived from *CGIXMLRPCRequestHandler* and provides a means of creating self-documenting, XML-RPC CGI scripts. HTTP POST requests are handled as XML-RPC method calls. HTTP GET requests are handled by generating pydoc-style HTML documentation. This allows a server to provide its own web-based documentation.

DocCGIXMLRPCRequestHandler.**set_server_title**(*server_title*)
Set the title used in the generated HTML documentation. This title will be used inside the HTML "title" element.

DocCGIXMLRPCRequestHandler.**set_server_name**(*server_name*)
Set the name used in the generated HTML documentation. This name will appear at the top of the generated documentation inside a "h1" element.

DocCGIXMLRPCRequestHandler.**set_server_documentation**(*server_documentation*)
Set the description used in the generated HTML documentation. This description will appear as a paragraph, below the server name, in the documentation.

21.28 ipaddress —IPv4/IPv6 操作库

源代码： Lib/ipaddress.py

---

ipaddress 提供了创建、处理和操作 IPv4 和 IPv6 地址和网络的功能。

该模块中的函数和类可以直接处理与 IP 地址相关的各种任务，包括检查两个主机是否在同一个子网中，遍历某个子网中的所有主机，检查一个字符串是否是一个有效的 IP 地址或网络定义等等。

这是完整的模块 API 参考—若要查看概述，请见 ipaddress-howto.

3.3 新版功能.

21.28.1 方便的工厂函数

ipaddress 模块提供来工厂函数来方便地创建 IP 地址，网络和接口：

ipaddress.**ip_address**(*address*)
返回一个*IPv4Address* 或*IPv6Address* 对象，取决于作为参数传递的 IP 地址。可以提供 IPv4 或 IPv6 地址，小于 2**32 的整数默认被认为是 IPv4。如果 *address* 不是有效的 IPv4 或 IPv6 地址，则会抛出*ValueError*。

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress.**ip_network**(*address*, *strict=True*)
返回一个*IPv4Network* 或*IPv6Network* 对象，具体取决于作为参数传入的 IP 地址。*address* 是表示 IP 网址的字符串或整数。可以提供 IPv4 或 IPv6 网址；小于 2**32 的整数默认被视为 IPv4。*strict* 会被传给*IPv4Network* 或*IPv6Network* 构造器。如果 *address* 不表示有效的 IPv4 或 IPv6 网址，或者网络设置了 host 比特位，则会引发*ValueError*。

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.**ip_interface**(*address*)
返回一个*IPv4Interface* 或*IPv6Interface* 对象，取决于作为参数传递的 IP 地址。*address* 是代表 IP 地址的字符串或整数。可以提供 IPv4 或 IPv6 地址，小于 2**32 的整数默认认为是 IPv4。如果 *address* 不是有效的 IPv4 或 IPv6 地址，则会抛出一个*ValueError*。

这些方便的函数的一个缺点是需要同时处理 IPv4 和 IPv6 格式，这意味着提供的错误信息并不精准，因为函数不知道是打算采用 IPv4 还是 IPv6 格式。更详细的错误报告可以通过直接调用相应版本的类构造函数来获得。

21.28.2 IP 地址

地址对象

IPv4Address 和*IPv6Address* 对象有很多共同的属性。一些只对 IPv6 地址有意义的属性也在*IPv4Address* 对象实现，以便更容易编写正确处理两种 IP 版本的代码。地址对象是可哈希的*hashable*，所以它们可以作为字典中的键来使用。

class ipaddress.**IPv4Address**(*address*)
构造一个 IPv4 地址。如果 *address* 不是一个有效的 IPv4 地址，会抛出*AddressValueError* 。

以下是有效的 IPv4 地址：

1. 以十进制小数点表示的字符串，由四个十进制整数组成，范围为 0–255，用点隔开 (例如 `192.168.0.1`)。每个整数代表地址中的八位（一个字节）。只有对于小于 8 的值，才允许使用前导零 (因为对这种字符串的十进制和八进制解释之间没有任何歧义)。
2. 一个 32 位可容纳的整数。
3. 一个长度为 4 的封装在*bytes* 对象中的整数 (高位优先)。

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

version
合适的版本号：IPv4 为 `4` ，IPv6 为 `6` 。

max_prefixlen
在该版本的地址表示中，比特数的总数：IPv4 为 `32` ；IPv6 为 `128` 。

The prefix defines the number of leading bits in an address that are compared to determine whether or not an address is part of a network.

compressed

exploded
The string representation in dotted decimal notation. Leading zeroes are never included in the representation.

As IPv4 does not define a shorthand notation for addresses with octets set to zero, these two attributes are always the same as `str(addr)` for IPv4 addresses. Exposing these attributes makes it easier to write display code that can handle both IPv4 and IPv6 addresses.

packed
The binary representation of this address - a *bytes* object of the appropriate length (most significant octet first). This is 4 bytes for IPv4 and 16 bytes for IPv6.

reverse_pointer
The name of the reverse DNS PTR record for the IP address, e.g.:

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

This is the name that could be used for performing a PTR lookup, not the resolved hostname itself.

3.5 新版功能.

is_multicast
如果该地址被保留用作多播用途，返回 `True` 。关于多播地址，请参见 **RFC 3171** （IPv4）和 **RFC 2373** （IPv6）。

is_private
如果该地址被分配至私有网络，返回 `True` 。关于公共网络，请参见 iana-ipv4-special-registry （针对 IPv4）和 iana-ipv6-special-registry （针对 IPv6）。

is_global
如果该地址被分配至公共网络，返回 `True` 。关于公共网络，请参见 iana-ipv4-special-registry （针对 IPv4）和 iana-ipv6-special-registry （针对 IPv6）。

3.4 新版功能.

is_unspecified
`True` if the address is unspecified. See **RFC 5735** (for IPv4) or **RFC 2373** (for IPv6).

is_reserved
如果该地址属于互联网工程任务组（IETF）所规定的其他保留地址，返回 `True` 。

is_loopback
如果该地址为一个回环地址，返回 `True` 。关于回环地址，请见 **RFC 3330** （IPv4）和 **RFC 2373** （IPv6）

is_link_local
`True` if the address is reserved for link-local usage. See **RFC 3927**.

class `ipaddress.`**IPv6Address**(*address*)
构造一个 IPv6 地址。如果 *address* 不是一个有效的 IPv6 地址，会抛出*`AddressValueError`* 。

以下是有效的 IPv6 地址：

1. A string consisting of eight groups of four hexadecimal digits, each group representing 16 bits. The groups are separated by colons. This describes an *exploded* (longhand) notation. The string can also be *compressed* (shorthand notation) by various means. See **RFC 4291** for details. For example, `"0000:0000:0000:0000:0000:0abc:0007:0def"` can be compressed to `"::abc:7:def"`.
2. An integer that fits into 128 bits.
3. An integer packed into a *`bytes`* object of length 16, big-endian.

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
```

compressed

The short form of the address representation, with leading zeroes in groups omitted and the longest sequence of groups consisting entirely of zeroes collapsed to a single empty group.

This is also the value returned by `str(addr)` for IPv6 addresses.

exploded

The long form of the address representation, with all leading zeroes and groups consisting entirely of zeroes included.

For the following attributes, see the corresponding documentation of the *`IPv4Address`* class:

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

is_unspecified

is_reserved

is_loopback

is_link_local
3.4 新版功能: is_global

is_site_local
`True` if the address is reserved for site-local usage. Note that the site-local address space has been deprecated by **RFC 3879**. Use *`is_private`* to test if this address is in the space of unique local addresses as defined by **RFC 4193**.

ipv4_mapped
For addresses that appear to be IPv4 mapped addresses (starting with `::FFFF/96`), this property will report the embedded IPv4 address. For any other address, this property will be `None`.

sixtofour
For addresses that appear to be 6to4 addresses (starting with `2002::/16`) as defined by **RFC 3056**, this property will report the embedded IPv4 address. For any other address, this property will be `None`.

teredo
For addresses that appear to be Teredo addresses (starting with `2001::/32`) as defined by **RFC 4380**, this property will report the embedded `(server, client)` IP address pair. For any other address, this property will be `None`.

Conversion to Strings and Integers

To interoperate with networking interfaces such as the socket module, addresses must be converted to strings or integers. This is handled using the *str()* and *int()* builtin functions:

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

运算符

Address objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

比较运算符

Address objects can be compared with the usual set of comparison operators. Some examples:

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
```

算术运算符

Integers can be added to or subtracted from address objects. Some examples:

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4 address
```

21.28.3 IP 网络的定义

The *IPv4Network* and *IPv6Network* objects provide a mechanism for defining and inspecting IP network definitions. A network definition consists of a *mask* and a *network address*, and as such defines a range of IP addresses that equal the network address when masked (binary AND) with the mask. For example, a network definition with the mask `255.255.255.0` and the network address `192.168.1.0` consists of IP addresses in the inclusive range `192.168.1.0` to `192.168.1.255`.

Prefix, net mask and host mask

There are several equivalent ways to specify IP network masks. A *prefix* `/<nbits>` is a notation that denotes how many high-order bits are set in the network mask. A *net mask* is an IP address with some number of high-order bits set. Thus the prefix `/24` is equivalent to the net mask `255.255.255.0` in IPv4, or `ffff:ff00::` in IPv6. In addition, a *host mask* is the logical inverse of a *net mask*, and is sometimes used (for example in Cisco access control lists) to denote a network mask. The host mask equivalent to `/24` in IPv4 is `0.0.0.255`.

Network objects

All attributes implemented by address objects are implemented by network objects as well. In addition, network objects implement additional attributes. All of these are common between *IPv4Network* and *IPv6Network*, so to avoid duplication they are only documented for *IPv4Network*. Network objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Network**(*address*, *strict=True*)

Construct an IPv4 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional mask, separated by a slash (`/`). The IP address is the network address, and the mask can be either a single number, which means it's a *prefix*, or a string representation of an IPv4 address. If it's the latter, the mask is interpreted as a *net mask* if it starts with a non-zero field, or as a *host mask* if it starts with a zero field, with the single exception of an all-zero mask which is treated as a *net mask*. If no mask is provided, it's considered to be `/32`.

   For example, the following *address* specifications are equivalent: `192.168.1.0/24`, `192.168.1.0/255.255.255.0` and `192.168.1.0/0.0.0.255`.

2. An integer that fits into 32 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/32`.

3. An integer packed into a *bytes* object of length 4, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 32-bits integer, a 4-bytes packed integer, or an existing IPv4Address object; and the netmask is either an integer representing the prefix length (e.g. `24`) or a string representing the prefix mask (e.g. `255.255.255.0`).

An *AddressValueError* is raised if *address* is not a valid IPv4 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv4 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

Unless stated otherwise, all network methods accepting other network/address objects will raise *TypeError* if the argument's IP version is incompatible to `self`.

在 3.5 版更改: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen
Refer to the corresponding attribute documentation in *IPv4Address*.

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local
These attributes are true for the network as a whole if they are true for both the network address and the broadcast address.

network_address
The network address for the network. The network address and the prefix length together uniquely define a network.

broadcast_address
The broadcast address for the network. Packets sent to the broadcast address should be received by every host on the network.

hostmask
The host mask, as an *IPv4Address* object.

netmask
The net mask, as an *IPv4Address* object.

with_prefixlen

compressed

exploded
A string representation of the network, with the mask in prefix notation.

`with_prefixlen` and `compressed` are always the same as `str(network)`. `exploded` uses the exploded form the network address.

with_netmask
A string representation of the network, with the mask in net mask notation.

with_hostmask
A string representation of the network, with the mask in host mask notation.

num_addresses
The total number of addresses in the network.

prefixlen
Length of the network prefix, in bits.

hosts()
Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the network address itself and the network broadcast address. For networks with a mask length of 31, the network address and network broadcast address are also included in the result.

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
```

overlaps(*other*)
`True` if this network is partly or wholly contained in *other* or *other* is wholly contained in this network.

address_exclude(*network*)
Computes the network definitions resulting from removing the given *network* from this one. Returns an iterator of network objects. Raises *ValueError* if *network* is not completely contained in this network.

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets(*prefixlen_diff=1*, *new_prefix=None*)
The subnets that join to make the current network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be increased by. *new_prefix* is the desired new prefix of the subnets; it must be larger than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns an iterator of network objects.

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet(*prefixlen_diff=1*, *new_prefix=None*)
The supernet containing this network definition, depending on the argument values. *prefixlen_diff* is the amount our prefix length should be decreased by. *new_prefix* is the desired new prefix of the supernet; it must be smaller than our prefix. One and only one of *prefixlen_diff* and *new_prefix* must be set. Returns a single network object.

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

compare_networks(*other*)

Compare this network to *other*. In this comparison only the network addresses are considered; host bits aren't. Returns either `-1`, `0` or `1`.

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

class ipaddress.**IPv6Network**(*address*, *strict=True*)

Construct an IPv6 network definition. *address* can be one of the following:

1. A string consisting of an IP address and an optional prefix length, separated by a slash (`/`). The IP address is the network address, and the prefix length must be a single number, the *prefix*. If no prefix length is provided, it's considered to be `/128`.

   Note that currently expanded netmasks are not supported. That means `2001:db00::0/24` is a valid argument while `2001:db00::0/ffff:ff00::` not.

2. An integer that fits into 128 bits. This is equivalent to a single-address network, with the network address being *address* and the mask being `/128`.

3. An integer packed into a *bytes* object of length 16, big-endian. The interpretation is similar to an integer *address*.

4. A two-tuple of an address description and a netmask, where the address description is either a string, a 128-bits integer, a 16-bytes packed integer, or an existing IPv6Address object; and the netmask is an integer representing the prefix length.

An *AddressValueError* is raised if *address* is not a valid IPv6 address. A *NetmaskValueError* is raised if the mask is not valid for an IPv6 address.

If *strict* is `True` and host bits are set in the supplied address, then *ValueError* is raised. Otherwise, the host bits are masked out to determine the appropriate network address.

在 3.5 版更改: Added the two-tuple form for the *address* constructor parameter.

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()
Returns an iterator over the usable hosts in the network. The usable hosts are all the IP addresses that belong to the network, except the Subnet-Router anycast address. For networks with a mask length of 127, the Subnet-Router anycast address is also included in the result.

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff=1*, *new_prefix=None*)

supernet(*prefixlen_diff=1*, *new_prefix=None*)

compare_networks(*other*)
Refer to the corresponding attribute documentation in *IPv4Network*.

is_site_local
These attribute is true for the network as a whole if it is true for both the network address and the broadcast address.

运算符

Network objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Network objects can be compared with the usual set of logical operators. Network objects are ordered first by network address, then by net mask.

迭代

Network objects can be iterated to list all the addresses belonging to the network. For iteration, *all* hosts are returned, including unusable hosts (for usable hosts, use the *hosts()* method). An example:

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

Networks as containers of addresses

Network objects can act as containers of addresses. Some examples:

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

21.28.4 Interface objects

Interface objects are *hashable*, so they can be used as keys in dictionaries.

class ipaddress.**IPv4Interface**(*address*)

Construct an IPv4 interface. The meaning of *address* is as in the constructor of *IPv4Network*, except that arbitrary host addresses are always accepted.

IPv4Interface is a subclass of *IPv4Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

The address (*IPv4Address*) without network information.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network
The network (*IPv4Network*) this interface belongs to.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen
A string representation of the interface with the mask in prefix notation.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask
A string representation of the interface with the network as a net mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask
A string representation of the interface with the network as a host mask.

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.**IPv6Interface**(*address*)
Construct an IPv6 interface. The meaning of *address* is as in the constructor of *IPv6Network*, except that arbitrary host addresses are always accepted.

IPv6Interface is a subclass of *IPv6Address*, so it inherits all the attributes from that class. In addition, the following attributes are available:

ip

network

with_prefixlen

with_netmask

with_hostmask
Refer to the corresponding attribute documentation in *IPv4Interface*.

运算符

Interface objects support some operators. Unless stated otherwise, operators can only be applied between compatible objects (i.e. IPv4 with IPv4, IPv6 with IPv6).

Logical operators

Interface objects can be compared with the usual set of logical operators.

For equality comparison (== and !=), both the IP address and network must be the same for the objects to be equal. An interface will not compare equal to any address or network object.

For ordering (<, >, etc) the rules are different. Interface and address objects with the same IP version can be compared, and the address objects will always sort before the interface objects. Two interface objects are first compared by their networks and, if those are the same, then by their IP addresses.

21.28.5 Other Module Level Functions

The module also provides the following module level functions:

ipaddress.**v4_int_to_packed**(*address*)
Represent an address as 4 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv4 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv4 IP address.

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.**v6_int_to_packed**(*address*)
Represent an address as 16 packed bytes in network (big-endian) order. *address* is an integer representation of an IPv6 IP address. A *ValueError* is raised if the integer is negative or too large to be an IPv6 IP address.

ipaddress.**summarize_address_range**(*first*, *last*)
Return an iterator of the summarized network range given the first and last IP addresses. *first* is the first *IPv4Address* or *IPv6Address* in the range and *last* is the last *IPv4Address* or *IPv6Address* in the range. A *TypeError* is raised if *first* or *last* are not IP addresses or are not of the same version. A *ValueError* is raised if *last* is not greater than *first* or if *first* address version is not 4 or 6.

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...    ipaddress.IPv4Address('192.0.2.0'),
...    ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.
↪130/32')]
```

ipaddress.**collapse_addresses**(*addresses*)
Return an iterator of the collapsed *IPv4Network* or *IPv6Network* objects. *addresses* is an iterator of *IPv4Network* or *IPv6Network* objects. A *TypeError* is raised if *addresses* contains mixed version objects.

```
>>> [ipaddr for ipaddr in
... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.**get_mixed_type_key**(*obj*)
Return a key suitable for sorting between networks and addresses. Address and Network objects are not sortable by default; they're fundamentally different, so the expression:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

doesn't make sense. There are some times however, where you may wish to have *ipaddress* sort these anyway. If you need to do this, you can use this function as the *key* argument to *sorted()*.

obj is either a network or address object.

21.28.6 Custom Exceptions

To support more specific error reporting from class constructors, the module defines the following exceptions:

exception ipaddress.**AddressValueError**(*ValueError*)
Any value error related to the address.

exception ipaddress.**NetmaskValueError**(*ValueError*)
Any value error related to the net mask.

CHAPTER 22

多媒体服务

本章描述的模块实现了主要用于多媒体应用的各种算法或接口。它们可在安装时自行决定。这是一个概述：

22.1 audioop —处理原始音频数据

audioop 模块包含针对声音片段的一些有用操作。它操作的声音片段由 8、16、24 或 32 位宽的有符号整型样本组成，存储在类字节串对象 中。除非特别说明，否则所有标量项目均为整数。

在 3.4 版更改: 增加了对 24 位样本的支持。现在，所有函数都接受任何类字节串对象。而传入字符串会立即导致错误。

本模块提供对 a-LAW、u-LAW 和 Intel/DVI ADPCM 编码的支持。

部分更复杂的操作仅接受 16 位样本，而其他操作始终需要样本大小（以字节为单位）作为该操作的参数。

此模块定义了下列变量和函数：

exception audioop.**error**
所有错误都会抛出此异常，比如样本的字节数未知等等。

audioop.**add**(*fragment1*, *fragment2*, *width*)
两个样本作为参数传入，返回一个片段，该片段是两个样本的和。*width* 是样本宽度（以字节为单位），可以取 1, 2, 3 或 4。两个片段的长度应相同。如果发生溢出，较长的样本将被截断。

audioop.**adpcm2lin**(*adpcmfragment*, *width*, *state*)
将 Intel/DVI ADPCM 编码的片段解码为线性片段。关于 ADPCM 编码的详情请参阅*lin2adpcm()* 的描述。返回一个元组 (sample, newstate)，其中 sample 的位宽由 *width* 指定。

audioop.**alaw2lin**(*fragment*, *width*)
将 a-LAW 编码的声音片段转换为线性编码声音片段。由于 a-LAW 编码样本始终为 8 位，因此这里的 *width* 仅指输出片段的样本位宽。

audioop.**avg**(*fragment*, *width*)
返回片段中所有样本的平均值。

audioop.**avgpp**(*fragment*, *width*)
返回片段中所有样本的平均峰峰值。由于没有进行过滤，因此该例程的实用性尚存疑。

audioop.**bias**(*fragment*, *width*, *bias*)
返回一个片段，该片段由原始片段中的每个样本加上偏差组成。在溢出时样本会回卷 (wrap around)。

audioop.**byteswap**(*fragment*, *width*)
“按字节交换”片段中的所有样本，返回修改后的片段。将大端序样本转换为小端序样本，反之亦然。

3.4 新版功能.

audioop.**cross**(*fragment*, *width*)
将片段作为参数传入，返回其中过零点的数量。

audioop.**findfactor**(*fragment*, *reference*)
返回一个系数 *F* 使得 `rms(add(fragment, mul(reference, -F)))` 最小，即返回乘以 *reference* 后最匹配 *fragment* 的那个乘数。两个片段都应包含 2 字节宽的样本。

本例程所需的时间与 `len(fragment)` 成正比。

audioop.**findfit**(*fragment*, *reference*)
尽可能尝试让 *reference* 匹配 *fragment* 的一部分（*fragment* 应较长）。从概念上讲，完成这些靠从 *fragment* 中取出切片，使用*findfactor()* 计算最佳匹配，并最小化结果。两个片段都应包含 2 字节宽的样本。返回一个元组 `(offset, factor)`，其中 *offset* 是在 *fragment* 中的偏移量（整数），表示从此处开始最佳匹配，而 *factor* 是由*findfactor()* 定义的因数（浮点数）。

audioop.**findmax**(*fragment*, *length*)
在 *fragment* 中搜索所有长度为 *length* 的样本切片（不是字节!）中，能量最大的那一个切片，即返回 *i* 使得 `rms(fragment[i*2:(i+length)*2])` 最大。两个片段都应包含 2 字节宽的样本。

本例程所需的时间与 `len(fragment)` 成正比。

audioop.**getsample**(*fragment*, *width*, *index*)
返回片段中样本索引 *index* 的值。

audioop.**lin2adpcm**(*fragment*, *width*, *state*)
将样本转换为 4 位 Intel/DVI ADPCM 编码。ADPCM 编码是一种自适应编码方案，其中每个 4 比特数字是一个采样值与下一个采样值之间的差除以（不定的）步长。IMA 已选择使用 Intel/DVI ADPCM 算法，因此它很可能成为标准。

state 是一个表示编码器状态的元组。编码器返回一个元组 `(adpcmfrag, newstate)`，而 *newstate* 要在下一次调用*lin2adpcm()* 时传入。在初始调用中，可以将 `None` 作为 state 传递。*adpcmfrag* 是 ADPCM 编码的片段，每个字节打包了 2 个 4 比特值。

audioop.**lin2alaw**(*fragment*, *width*)
将音频片段中的采样值转换为 a-LAW 编码，并将其作为字节对象返回。a-LAW 是一种音频编码格式，仅使用 8 位样本即可获得大约 13 位的动态范围。Sun 音频硬件等使用该编码。

audioop.**lin2lin**(*fragment*, *width*, *newwidth*)
将采样在 1、2、3 和 4 字节格式之间转换。

注解: 在某些音频格式（如.WAV 文件）中，16、24 和 32 位采样是有符号的，但 8 位采样是无符号的。因此，当将这些格式转换为 8 位宽采样时，还需使结果加上 128:

```
new_frames = audioop.lin2lin(frames, old_width, 1)
new_frames = audioop.bias(new_frames, 1, 128)
```

反之，将 8 位宽的采样转换为 16、24 或 32 位时，必须采用相同的处理。

---

audioop.**lin2ulaw**(*fragment*, *width*)
将音频片段中的采样值转换为 u-LAW 编码，并将其作为字节对象返回。u-LAW 是一种音频编码格式，仅使用 8 位采样即可获得大约 14 位的动态范围。Sun 音频硬件等使用该编码。

audioop.**max**(*fragment*, *width*)
返回片段中所有采样值的最大 绝对值。

audioop.**maxpp**(*fragment*, *width*)
返回声音片段中的最大峰峰值。

audioop.**minmax**(*fragment*, *width*)
返回声音片段中所有采样值的最小值和最大值组成的元组。

audioop.**mul**(*fragment*, *width*, *factor*)
返回一个片段，该片段由原始片段中的每个采样值乘以浮点值 *factor* 组成。如果发生溢出，采样将被截断。

audioop.**ratecv**(*fragment*, *width*, *nchannels*, *inrate*, *outrate*, *state*[, *weightA*[, *weightB*]])
转换输入片段的帧速率。

state 是一个表示转换器状态的元组。转换器返回一个元组 `(newfragment, newstate)`，而 *newstate* 要在下一次调用 *ratecv()* 时传入。初始调用应传入 `None` 作为 state。

参数 *weightA* 和 *weightB* 是简单数字滤波器的参数，默认分别为 `1` 和 `0`。

audioop.**reverse**(*fragment*, *width*)
将片段中的采样值反转，返回修改后的片段。

audioop.**rms**(*fragment*, *width*)
返回片段的均方根值，即 `sqrt(sum(S_i^2)/n)`。

测量音频信号的能量。

audioop.**tomono**(*fragment*, *width*, *lfactor*, *rfactor*)
将立体声片段转换为单声道片段。左通道乘以 *lfactor*，右通道乘以 *rfactor*，然后两个通道相加得到单声道信号。

audioop.**tostereo**(*fragment*, *width*, *lfactor*, *rfactor*)
由单声道片段生成立体声片段。立体声片段中的两对采样都是从单声道计算而来的，即左声道是乘以 *lfactor*，右声道是乘以 *rfactor*。

audioop.**ulaw2lin**(*fragment*, *width*)
将 u-LAW 编码的声音片段转换为线性编码声音片段。由于 u-LAW 编码采样值始终为 8 位，因此这里的 *width* 仅指输出片段的采样位宽。

请注意，诸如 *mul()* 或 *max()* 之类的操作在单声道和立体声间没有区别，即所有采样都作相同处理。如果出现问题，应先将立体声片段拆分为两个单声道片段，之后再重组。以下是如何进行该操作的示例:

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

如果使用 ADPCM 编码器构造网络数据包，并且希望协议是无状态的（即能够容忍数据包丢失），则不仅需要传输数据，还应该传输状态。请注意，必须将 * 初始 * 状态（传入 *lin2adpcm()* 的状态）发送给解码器，不能发送最终状态（编码器返回的状态）。如果要使用 *struct.Struct* 以二进制保存状态，可以将第一个元素（预测值）用 16 位编码，将第二个元素（增量索引）用 8 位编码。

本 ADPCM 编码器从不与其他 ADPCM 编码器对立，仅针对自身。本开发者可能会误读标准，这种情况下它们将无法与相应标准互操作。

乍看之下 `find*()` 例程可能有些可笑。它们主要是用于回声消除，一种快速有效的方法是选取输出样本中能量最高的片段，在输入样本中定位该片段，然后从输入样本中减去整个输出样本:

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)    # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #              out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

22.2 `aifc` —读写 AIFF 和 AIFC 文件

源代码： Lib/aifc.py

本模块提供读写 AIFF 和 AIFF-C 文件的支持。AIFF 是音频交换文件格式 (Audio Interchange File Format)，一种用于在文件中存储数字音频采样的格式。AIFF-C 是该格式的更新版本，其中包括压缩音频数据的功能。

音频文件内有许多参数，用于描述音频数据。采样率或帧率是每秒对声音采样的次数。通道数表示音频是单声道，双声道还是四声道。每个通道的每个帧包含一次采样。采样大小是以字节表示的每次采样的大小。因此，一帧由 `nchannels * samplesize`（通道数 * 采样大小）字节组成，而一秒钟的音频包含 `nchannels * samplesize * framerate`（通道数 * 采样大小 * 帧率）字节。

例如，CD 质量的音频采样大小为 2 字节（16 位），使用 2 个声道（立体声），且帧速率为 44,100 帧/秒。这表示帧大小为 4 字节 (2*2)，一秒钟占用 2*2*44100 字节（176,400 字节）。

aifc 模块定义了以下函数:

aifc.**open**(*file*, *mode=None*)
: Open an AIFF or AIFF-C file and return an object instance with methods that are described below. The argument *file* is either a string naming a file or a *file object*. *mode* must be `'r'` or `'rb'` when the file must be opened for reading, or `'w'` or `'wb'` when the file must be opened for writing. If omitted, `file.mode` is used if it exists, otherwise `'rb'` is used. When used for writing, the file object should be seekable, unless you know ahead of time how many samples you are going to write in total and use `writeframesraw()` and `setnframes()`. The *open()* function may be used in a `with` statement. When the `with` block completes, the *close()* method is called.

  在 3.4 版更改: 支持了 `with` 语句。

当打开文件用于读取时，由*open()* 返回的对象有以下几种方法:

aifc.**getnchannels**()
: 返回音频的通道数（单声道为 1，立体声为 2）。

aifc.**getsampwidth**()
: 返回以字节表示的单个采样的大小。

aifc.**getframerate**()
: 返回采样率（每秒的音频帧数）。

aifc.**getnframes**()
返回文件中的音频帧总数。

aifc.**getcomptype**()
返回一个长度为 4 的字节数组，描述了音频文件中使用的压缩类型。对于 AIFF 文件，返回值为 b'NONE'。

aifc.**getcompname**()
返回一个字节数组，可转换为人类可读的描述，描述的是音频文件中使用的压缩类型。对于 AIFF 文件，返回值为 b'not compressed'。

aifc.**getparams**()
返回一个*namedtuple()* (nchannels, sampwidth, framerate, nframes, comptype, compname)，与 get*() 方法的输出相同。

aifc.**getmarkers**()
返回一个列表，包含音频文件中的所有标记。标记由一个 3 元素的元组组成。第一个元素是标记 ID（整数），第二个是标记位置，从数据开头算起的帧数（整数），第三个是标记的名称（字符串）。

aifc.**getmark**(*id*)
根据传入的标记 *id* 返回元组，元组与*getmarkers()* 中描述的一致。

aifc.**readframes**(*nframes*)
从音频文件读取并返回后续 *nframes* 个帧。返回的数据是一个字符串，包含每个帧所有通道的未压缩采样值。

aifc.**rewind**()
倒回读取指针。下一次*readframes()* 将从头开始。

aifc.**setpos**(*pos*)
移动读取指针到指定的帧上。

aifc.**tell**()
返回当前的帧号。

aifc.**close**()
关闭 AIFF 文件。调用此方法后，对象将无法再使用。

打开文件用于写入时，*open()* 返回的对象具有上述所有方法，但 readframes() 和 setpos() 除外，并额外具备了以下方法。只有调用了 set*() 方法之后，才能调用相应的 get*() 方法。在首次调用 writeframes() 或 writeframesraw() 之前，必须填写除帧数以外的所有参数。

aifc.**aiff**()
创建一个 AIFF 文件，默认创建 AIFF-C 文件，除非文件名以 '.aiff' 为后缀，在此情况下默认创建 AIFF 文件。

aifc.**aifc**()
创建一个 AIFF-C 文件。默认创建 AIFF-C 文件，除非文件名以 '.aiff' 为后缀，在此情况下默认创建 AIFF 文件。

aifc.**setnchannels**(*nchannels*)
指明音频文件中的通道数。

aifc.**setsampwidth**(*width*)
指明以字节为单位的音频采样大小。

aifc.**setframerate**(*rate*)
指明以每秒帧数表示的采样频率。

aifc.**setnframes**(*nframes*)
指明要写入到音频文件的帧数。如果未设定此形参或者未正确设定，则文件需要支持位置查找。

aifc.**setcomptype**(*type*, *name*)
指明压缩类型。如果未指明，则音频数据将不会被压缩。在 AIFF 文件中，压缩是无法实现的。name 参数应当为以字节数组表示的人类可读的压缩类型描述，type 参数应当为长度为 4 的字节数组。目前支持的压缩类型如下: `b'NONE'`, `b'ULAW'`, `b'ALAW'`, `b'G722'`。

aifc.**setparams**(*nchannels*, *sampwidth*, *framerate*, *comptype*, *compname*)
一次性设置上述所有参数。该参数是由多个形参组成的元组。这意味着可以使用*getparams()* 调用的结果作为*setparams()* 的参数。

aifc.**setmark**(*id*, *pos*, *name*)
添加具有给定 id (大于 0)，以及在给定位置上给定名称的标记。此方法可在*close()* 之前的任何时候被调用。

aifc.**tell**()
返回输出文件中的当前写入位置。适用于与*setmark()* 进行协同配合。

aifc.**writeframes**(*data*)
将数据写入到输出文件。此方法只能在设置了音频文件形参之后被调用。

在 3.4 版更改: 现在可接受任意*bytes-like object*。

aifc.**writeframesraw**(*data*)
类似于*writeframes()*，不同之处在于音频文件的标头不会被更新。

在 3.4 版更改: 现在可接受任意*bytes-like object*。

aifc.**close**()
关闭 AIFF 文件。文件的标头会被更新以反映音频数据的实际大小。在调用此方法之后，对象将无法再被使用。

22.3 sunau —读写 Sun AU 文件

源代码: Lib/sunau.py

---

sunau 模拟提供了一个处理 Sun AU 声音格式的便利接口。请注意此模块与*aifc* 和*wave* 是兼容接口的。

音频文件由标头和数据组成。标头的字段为:

域	目录
magic word	四个字节 `.snd`
header size	标头的大小，包括信息，以字节为单位。
data size	数据的物理大小，以字节为单位。
编码	指示音频样本的编码方式。
sample rate	采样率
# of channels	的通道数。
info	提供音频文件描述的 ASCII 字符串（用空字节填充）。

Apart from the info field, all header fields are 4 bytes in size. They are all 32-bit unsigned integers encoded in big-endian byte order.

The *sunau* module defines the following functions:

sunau.**open**(*file*, *mode*)
If *file* is a string, open the file by that name, otherwise treat it as a seekable file-like object. *mode* can be any of

'r' 只读模式。

'w' 只写模式。

Note that it does not allow read/write files.

A *mode* of 'r' returns an AU_read object, while a *mode* of 'w' or 'wb' returns an AU_write object.

sunau.**openfp**(*file*, *mode*)
同*open()*，用于向后兼容。

The *sunau* module defines the following exception:

exception sunau.**Error**
An error raised when something is impossible because of Sun AU specs or implementation deficiency.

The *sunau* module defines the following data items:

sunau.**AUDIO_FILE_MAGIC**
An integer every valid Sun AU file begins with, stored in big-endian form. This is the string .snd interpreted as an integer.

sunau.**AUDIO_FILE_ENCODING_MULAW_8**
sunau.**AUDIO_FILE_ENCODING_LINEAR_8**
sunau.**AUDIO_FILE_ENCODING_LINEAR_16**
sunau.**AUDIO_FILE_ENCODING_LINEAR_24**
sunau.**AUDIO_FILE_ENCODING_LINEAR_32**
sunau.**AUDIO_FILE_ENCODING_ALAW_8**
Values of the encoding field from the AU header which are supported by this module.

sunau.**AUDIO_FILE_ENCODING_FLOAT**
sunau.**AUDIO_FILE_ENCODING_DOUBLE**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G721**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G722**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_3**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_5**
Additional known values of the encoding field from the AU header, but which are not supported by this module.

22.3.1 AU_read 对象

AU_read objects, as returned by *open()* above, have the following methods:

AU_read.**close**()
Close the stream, and make the instance unusable. (This is called automatically on deletion.)

AU_read.**getnchannels**()
Returns number of audio channels (1 for mono, 2 for stereo).

AU_read.**getsampwidth**()
返回采样字节长度。

AU_read.**getframerate**()
返回采样频率。

AU_read.**getnframes**()
返回音频总帧数。

AU_read.**getcomptype**()
Returns compression type. Supported compression types are 'ULAW', 'ALAW' and 'NONE'.

AU_read.**getcompname**()
Human-readable version of *getcomptype()*. The supported types have the respective names 'CCITT G.711 u-law', 'CCITT G.711 A-law' and 'not compressed'.

AU_read.**getparams**()
返回一个*namedtuple()* (nchannels, sampwidth, framerate, nframes, comptype, compname)，与 get*() 方法的输出相同。

AU_read.**readframes**(*n*)
Reads and returns at most *n* frames of audio, as a *bytes* object. The data will be returned in linear format. If the original data is in u-LAW format, it will be converted.

AU_read.**rewind**()
重置文件指针至音频开头.

以下两个方法都使用指针，具体实现由其底层决定。

AU_read.**setpos**(*pos*)
Set the file pointer to the specified position. Only values returned from *tell()* should be used for *pos*.

AU_read.**tell**()
Return current file pointer position. Note that the returned value has nothing to do with the actual position in the file.

The following two functions are defined for compatibility with the *aifc*, and don't do anything interesting.

AU_read.**getmarkers**()
返回 None。

AU_read.**getmark**(*id*)
引发错误异常。

22.3.2 AU_write 对象

AU_write objects, as returned by *open()* above, have the following methods:

AU_write.**setnchannels**(*n*)
设置声道数。

AU_write.**setsampwidth**(*n*)
Set the sample width (in bytes.)

在 3.4 版更改: Added support for 24-bit samples.

AU_write.**setframerate**(*n*)
Set the frame rate.

AU_write.**setnframes**(*n*)
Set the number of frames. This can be later changed, when and if more frames are written.

AU_write.**setcomptype**(*type*, *name*)
Set the compression type and description. Only 'NONE' and 'ULAW' are supported on output.

AU_write.**setparams**(*tuple*)
The *tuple* should be (nchannels, sampwidth, framerate, nframes, comptype, compname), with values valid for the set*() methods. Set all parameters.

AU_write.**tell**()
Return current position in the file, with the same disclaimer for the *AU_read.tell()* and *AU_read.setpos()* methods.

AU_write.**writeframesraw**(*data*)
写入音频数据但不更新 *nframes*。

在 3.4 版更改: 现在可接受任意*bytes-like object*。

AU_write.**writeframes**(*data*)
写入音频数据并更新 *nframes*。

在 3.4 版更改: 现在可接受任意*bytes-like object*。

AU_write.**close**()
Make sure *nframes* is correct, and close the file.

This method is called upon deletion.

Note that it is invalid to set any parameters after calling `writeframes()` or `writeframesraw()`.

22.4 wave —读写 WAV 格式文件

源代码: Lib/wave.py

---

wave 模块提供了一个处理 WAV 声音格式的便利接口。它不支持压缩/解压，但是支持单声道/立体声。

wave 模块定义了以下函数和异常:

wave.**open**(*file*, *mode=None*)
如果 *file* 是一个字符串，打开对应文件名的文件。否则就把它作为文件类对象来处理。*mode* 可以为以下值:

'rb' 只读模式。

'wb' 只写模式。

注意不支持同时读写 WAV 文件。

mode 设为 `'rb'` 时返回一个 `Wave_read` 对象，而 *mode* 设为 `'wb'` 时返回一个 `Wave_write` 对象。如果省略 *mode* 并指定 *file* 来传入一个文件类对象，则 `file.mode` 会被用作 *mode* 的默认值。

如果操作的是文件对象，当使用 wave 对象的 `close()` 方法时，并不会真正关闭文件对象，这需要调用者负责来关闭文件对象。

The *open()* function may be used in a `with` statement. When the `with` block completes, the *Wave_read.close()* or *Wave_write.close()* method is called.

在 3.4 版更改: 添加了对不可搜索文件的支持。

wave.**openfp**(*file*, *mode*)
同*open()*，用于向后兼容。

exception wave.**Error**
当不符合 WAV 格式或无法操作时引发的错误。

22.4.1 Wave_read 对象

由*open()* 返回的 Wave_read 对象，有以下几种方法:

Wave_read.**close**()
关闭*wave* 打开的数据流并使对象不可用。当对象销毁时会自动调用。

Wave_read.**getnchannels**()
返回声道数量（`1` 为单声道，`2` 为立体声）

Wave_read.**getsampwidth**()
返回采样字节长度。

Wave_read.**getframerate**()
返回采样频率。

Wave_read.**getnframes**()
返回音频总帧数。

Wave_read.**getcomptype**()
返回压缩类型（只支持 'NONE' 类型）

Wave_read.**getcompname**()
getcomptype() 的通俗版本。使用 'not compressed' 代替 'NONE'。

Wave_read.**getparams**()
返回一个*namedtuple()* (nchannels, sampwidth, framerate, nframes, comptype, compname)，与 get*() 方法的输出相同。

Wave_read.**readframes**(*n*)
读取并返回以*bytes* 对象表示的最多 *n* 帧音频。

Wave_read.**rewind**()
重置文件指针至音频开头.

后面两个方法是为了和*aifc* 保持兼容，实际不做任何事情。

Wave_read.**getmarkers**()
返回 None。

Wave_read.**getmark**(*id*)
引发错误异常。

以下两个方法都使用指针，具体实现由其底层决定。

Wave_read.**setpos**(*pos*)
设置文件指针到指定位置。

Wave_read.**tell**()
返回当前文件指针位置。

22.4.2 Wave_write 对象

对于可查找的输出流，wave 头将自动更新以反映实际写入的帧数。对于不可查找的流，当写入第一帧时 *nframes* 值必须准确。获取准确的 *nframes* 值可以通过调用*setnframes()* 或*setparams()* 并附带*close()* 被调用之前将要写入的帧数，然后使用*writeframesraw()* 来写入帧数据，或者通过调用*writeframes()* 并附带所有要写入的帧。在后一种情况下*writeframes()* 将计算数据中的帧数并在写入帧数据之前相应地设置 *nframes*。

由*open()* 返回的 Wave_write 对象，有以下几种方法:

在 3.4 版更改: 添加了对不可搜索文件的支持。

Wave_write.**close**()
确保 *nframes* 是正确的，并在文件被*wave* 打开时关闭它。此方法会在对象收集时被调用。如果输出流不可查找且 *nframes* 与实际写入的帧数不匹配时引发异常。

Wave_write.**setnchannels**(*n*)
设置声道数。

Wave_write.**setsampwidth**(*n*)
设置采样字节长度为 *n*。

Wave_write.**setframerate**(*n*)
设置采样频率为 *n*。

在 3.2 版更改: 对此方法的非整数输入会被舍入到最接近的整数。

Wave_write.**setnframes**(*n*)
设置总帧数为 *n*。如果与之后实际写入的帧数不一致此值将会被更改（如果输出流不可查找则此更改尝试将引发错误）。

Wave_write.**setcomptype**(*type*, *name*)
设置压缩格式。目前只支持 NONE 即无压缩格式。

Wave_write.**setparams**(*tuple*)
tuple 应该是 (nchannels, sampwidth, framerate, nframes, comptype, compname)，每项的值应可用于 set*() 方法。设置所有形参。

Wave_write.**tell**()
返回当前文件指针，其指针含义和*Wave_read.tell()* 以及*Wave_read.setpos()* 是一致的。

Wave_write.**writeframesraw**(*data*)
写入音频数据但不更新 *nframes*。

在 3.4 版更改: 现在可接受任意*bytes-like object*。

Wave_write.**writeframes**(*data*)
写入音频帧并确保 *nframes* 是正确的。如果输出流不可查找且在 *data* 被写入之后写入的总帧数与之前设定的 has been written does not match the previously set value for *nframes* 值不匹配将会引发错误。

在 3.4 版更改: 现在可接受任意*bytes-like object*。

注意在调用 writeframes() 或 writeframesraw() 之后再设置任何格式参数是无效的，而且任何这样的尝试将引发*wave.Error*。

22.5 chunk —读取 IFF 分块数据

源代码: Lib/chunk.py

本模块提供了一个读取使用 EA IFF 85 分块的数据的接口 chunks.[1] 这种格式使用的场合有 Audio Interchange File Format (AIFF/AIFF-C) 和 Real Media File Format (RMFF) 等。与它们密切相关的 WAVE 音频文件也可使用此模块来读取。

一个 chunk 具有以下结构:

偏移	长度	目录
0	4	区块 ID
4	4	大端字节顺序的块大小，不包括头
8	*n*	数据字节，其中 *n* 是前一字段中给出的大小
8 + *n*	0 或 1	如果 *n* 为奇数且使用块对齐，则需要填充字节

ID 是一个 4 字节的字符串，用于标识块的类型。

大小字段（32 位的值，使用大端字节序编码）给出分块数据的大小，不包括 8 字节的标头。

使用由一个或更多分块组成的 IFF 类型文件。此处定义的*Chunk* 类的建议使用方式是在每个分块开始时实例化一个实例并从实例读取直到其末尾，在那之后可以再实例化新的实例。到达文件末尾时，创建新实例将会失败并引发*EOFError* 异常。

[1] “EA IFF 85” 交换格式文件标准, Jerry Morrison, Electronic Arts, 1985 年 1 月。

class chunk.**Chunk**(*file*, *align=True*, *bigendian=True*, *inclheader=False*)

代表一个分块的类。*file* 参数预期为一个文件类对象。特别地也允许该类的实例。唯一必需的方法是 `read()`。如果存在`seek()` 和`tell()` 方法并且没有引发异常，它们也会被使用。如果存在这些方法并且引发了异常，则它们不应改变目标对象。如果可选参数 *align* 为真值，则分块应当以 2 字节边界对齐。如果 *align* 为假值，则不使用对齐。此参数默认为真值。如果可选参数 *bigendian* 为假值，分块大小应当为小端序。这对于 WAVE 音频文件是必须的。此参数默认为真值。如果可选参数 *inclheader* 为真值，则分块标头中给出的大小将包括标头的大小。此参数默认为假值。

`Chunk` 对象支持下列方法：

getname()

返回分块的名称（ID）。这是分块的头 4 个字节。

getsize()

返回分块的大小。

close()

关闭并跳转到分块的末尾。这不会关闭下层的文件。

在`close()` 方法已被调用后其余方法将会引发`OSError`。在 Python 3.3 之前，它们曾会引发`IOError`，现在这是`OSError` 的一个别名。

isatty()

返回 `False`。

seek(*pos*, *whence=0*)

设置分块的当前位置。*whence* 参数为可选项并且默认为 `0` (绝对文件定位)；其他值还有 `1` (相对当前位置查找) 和 `2` (相对文件末尾查找)。没有返回值。如果下层文件不支持查找，则只允许向前查找。

tell()

将当前位置返回到分块。

read(*size=-1*)

从分块读取至多 *size* 个字节（如果在获得 *size* 个字节之前已到达分块末尾则读取的字节会少于此数量）。如果 *size* 参数为负值或被省略，则读取所有字节直到分块末尾。当立即遇到分块末尾则返回空字节串对象。

skip()

跳到分块末尾。此后对分块再次调用`read()` 将返回 `b''`。如果你对分块的内容不感兴趣，则应当调用此方法以使文件指向下一分块的开头。

备注

22.6 colorsys —颜色系统间的转换

源代码： Lib/colorsys.py

`colorsys` 模块定义了计算机显示器所用的 RGB (Red Green Blue) 色彩空间与三种其他色彩坐标系统 YIQ, HLS (Hue Lightness Saturation) 和 HSV (Hue Saturation Value) 表示的颜色值之间的双向转换。所有这些色彩空间的坐标都使用浮点数值来表示。在 YIQ 空间中，Y 坐标取值为 0 和 1 之间，而 I 和 Q 坐标均可以为正数或负数。在所有其他空间中，坐标取值均为 0 和 1 之间。

参见：

More information about color spaces can be found at http://www.poynton.com/ColorFAQ.html and https://www.cambridgeincolour.com/tutorials/color-spaces.htm.

colorsys 模块定义了如下函数：

colorsys.**rgb_to_yiq**(*r*, *g*, *b*)
把颜色从 RGB 值转为 YIQ 值。

colorsys.**yiq_to_rgb**(*y*, *i*, *q*)
把颜色从 YIQ 值转为 RGB 值。

colorsys.**rgb_to_hls**(*r*, *g*, *b*)
把颜色从 RGB 值转为 HLS 值。

colorsys.**hls_to_rgb**(*h*, *l*, *s*)
把颜色从 HLS 值转为 RGB 值。

colorsys.**rgb_to_hsv**(*r*, *g*, *b*)
把颜色从 RGB 值转为 HSV 值。

colorsys.**hsv_to_rgb**(*h*, *s*, *v*)
把颜色从 HSV 值转为 RGB 值。

示例：

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

22.7 imghdr —推测图像类型

源代码 Lib/imghdr.py

---

imghdr 模块推测文件或字节流中的图像的类型。

imghdr 模块定义了以下类型：

imghdr.**what**(*filename*, *h=None*)
测试包含在命名为 *filename* 的文件中的图像数据，并且返回描述此类图片的字符串。如果可选的 *h* 被提供，*filename* 将被忽略并且 *h* 包含将被测试的二进制流。

在 3.6 版更改: 接受一个*path-like object*。

接下来的图像类型是可识别的，返回值来自*what()*：

值	图像格式
'rgb'	SGI 图像库文件
'gif'	GIF 87a 和 89a 文件
'pbm'	便携式位图文件
'pgm'	便携式灰度图文件
'ppm'	便携式像素表文件
'tiff'	TIFF 文件
'rast'	Sun 光栅文件
'xbm'	X 位图文件
'jpeg'	JFIF 或 Exif 格式的 JPEG 数据
'bmp'	BMP 文件
'png'	便携式网络图像
'webp'	WebP 文件
'exr'	OpenEXR 文件

3.5 新版功能: *exr* 和 *webp* 格式被添加。

你可以扩展此*imghdr* 可以被追加的这个变量识别的文件格式的列表:

imghdr.**tests**
执行单个测试的函数列表。每个函数都有两个参数:字节流和类似开放文件的对象。当*what()* 用字节流调用时,类文件对象将是 None。

如果测试成功,这个测试函数应当返回一个描述图像类型的字符串,否则返回 None。

示例:

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

22.8 sndhdr —推测声音文件的类型

源代码 Lib/sndhdr.py

---

sndhdr 提供了企图猜测文件中的声音数据类型的功能函数。当这些函数可以推测出存储在文件中的声音数据的类型是,它们返回一个*collections.namedtuple()*,包含了五种属性:(filetype, framerate, nchannels, nframes, sampwidth)。这些 *type* 的值表示数据的类型,会是以下字符串之一: 'aifc', 'aiff', 'au', 'hcom', 'sndr', 'sndt', 'voc', 'wav', '8svx', 'sb', 'ub', or 'ul' 。*sampling_rate* 可能是实际值或者当未知或者难以解码时的 0。类似的,*channels* 也会返回实际值或者在无法推测或者难以解码时返回 0。*frames* 则是实际值或 -1。元组的最后一项,*bits_per_sample* 将会为比特表示的 sample 大小或者 A-LAW 时为 'A',u-LAW 时为 'U'。

sndhdr.**what**(*filename*)
使用*whathdr()* 推测存储在 *filename* 文件中的声音数据的类型。如果成功,返回上述的命名元组,否则返回 None。

在 3.5 版更改: 将结果从元组改为命名元组。

sndhdr.**whathdr**(*filename*)
基于文件头推测存储在文件中的声音数据类型。文件名由 *filename* 给出。这个函数在成功时返回上述命名元组,或者在失败时返回 None。

在 3.5 版更改: 将结果从元组改为命名元组。

22.9 ossaudiodev —访问兼容 OSS 的音频设备

该模块允许您访问 OSS（开放式音响系统）音频接口。OSS 可用于广泛的开源和商业 Unices，并且是 Linux 和最新版本的 FreeBSD 的标准音频接口。

在 3.3 版更改: 此模块中过去会引发 *IOError* 的操作现在将引发 *OSError*。

参见:

开放之声系统程序员手册 OSS C API 的官方文档

该模块定义了大量由 OSS 设备驱动提供的常量；请参阅 "<sys/soundcard.h>" Linux 或 FreeBSD 上的列表。

ossaudiodev defines the following variables and functions:

exception ossaudiodev.**OSSAudioError**

This exception is raised on certain errors. The argument is a string describing what went wrong.

(If *ossaudiodev* receives an error from a system call such as `open()`, `write()`, or `ioctl()`, it raises *OSError*. Errors detected directly by *ossaudiodev* result in *OSSAudioError*.)

(For backwards compatibility, the exception class is also available as `ossaudiodev.error`.)

ossaudiodev.**open**(*mode*)
ossaudiodev.**open**(*device*, *mode*)

Open an audio device and return an OSS audio device object. This object supports many file-like methods, such as `read()`, `write()`, and `fileno()` (although there are subtle differences between conventional Unix read/write semantics and those of OSS audio devices). It also supports a number of audio-specific methods; see below for the complete list of methods.

device is the audio device filename to use. If it is not specified, this module first looks in the environment variable `AUDIODEV` for a device to use. If not found, it falls back to `/dev/dsp`.

mode is one of `'r'` for read-only (record) access, `'w'` for write-only (playback) access and `'rw'` for both. Since many sound cards only allow one process to have the recorder or player open at a time, it is a good idea to open the device only for the activity needed. Further, some sound cards are half-duplex: they can be opened for reading or writing, but not both at once.

Note the unusual calling syntax: the *first* argument is optional, and the second is required. This is a historical artifact for compatibility with the older `linuxaudiodev` module which *ossaudiodev* supersedes.

ossaudiodev.**openmixer**([*device*])

Open a mixer device and return an OSS mixer device object. *device* is the mixer device filename to use. If it is not specified, this module first looks in the environment variable `MIXERDEV` for a device to use. If not found, it falls back to `/dev/mixer`.

22.9.1 Audio Device Objects

Before you can write to or read from an audio device, you must call three methods in the correct order:

1. `setfmt()` to set the output format
2. `channels()` to set the number of channels
3. `speed()` to set the sample rate

Alternately, you can use the `setparameters()` method to set all three audio parameters at once. This is more convenient, but may not be as flexible in all cases.

The audio device objects returned by *open()* define the following methods and (read-only) attributes:

oss_audio_device.**close**()
Explicitly close the audio device. When you are done writing to or reading from an audio device, you should explicitly close it. A closed device cannot be used again.

oss_audio_device.**fileno**()
Return the file descriptor associated with the device.

oss_audio_device.**read**(*size*)
Read *size* bytes from the audio input and return them as a Python string. Unlike most Unix device drivers, OSS audio devices in blocking mode (the default) will block *read()* until the entire requested amount of data is available.

oss_audio_device.**write**(*data*)
Write a *bytes-like object* *data* to the audio device and return the number of bytes written. If the audio device is in blocking mode (the default), the entire data is always written (again, this is different from usual Unix device semantics). If the device is in non-blocking mode, some data may not be written—see *writeall()*.

在 3.5 版更改: 现在支持可写的字节类对象。

oss_audio_device.**writeall**(*data*)
Write a *bytes-like object* *data* to the audio device: waits until the audio device is able to accept data, writes as much data as it will accept, and repeats until *data* has been completely written. If the device is in blocking mode (the default), this has the same effect as *write()*; *writeall()* is only useful in non-blocking mode. Has no return value, since the amount of data written is always equal to the amount of data supplied.

在 3.5 版更改: 现在支持可写的字节类对象。

在 3.2 版更改: Audio device objects also support the context management protocol, i.e. they can be used in a `with` statement.

The following methods each map to exactly one `ioctl()` system call. The correspondence is obvious: for example, `setfmt()` corresponds to the `SNDCTL_DSP_SETFMT` ioctl, and `sync()` to `SNDCTL_DSP_SYNC` (this can be useful when consulting the OSS documentation). If the underlying `ioctl()` fails, they all raise *OSError*.

oss_audio_device.**nonblock**()
Put the device into non-blocking mode. Once in non-blocking mode, there is no way to return it to blocking mode.

oss_audio_device.**getfmts**()
Return a bitmask of the audio output formats supported by the soundcard. Some of the formats supported by OSS are:

格式	描述
AFMT_MU_LAW	a logarithmic encoding (used by Sun `.au` files and `/dev/audio`)
AFMT_A_LAW	a logarithmic encoding
AFMT_IMA_ADPCM	a 4:1 compressed format defined by the Interactive Multimedia Association
AFMT_U8	Unsigned, 8-bit audio
AFMT_S16_LE	Signed, 16-bit audio, little-endian byte order (as used by Intel processors)
AFMT_S16_BE	Signed, 16-bit audio, big-endian byte order (as used by 68k, PowerPC, Sparc)
AFMT_S8	Signed, 8 bit audio
AFMT_U16_LE	Unsigned, 16-bit little-endian audio
AFMT_U16_BE	Unsigned, 16-bit big-endian audio

Consult the OSS documentation for a full list of audio formats, and note that most devices support only a subset of these formats. Some older devices only support `AFMT_U8`; the most common format used today is `AFMT_S16_LE`.

oss_audio_device.**setfmt**(*format*)
Try to set the current audio format to *format*—see *getfmts()* for a list. Returns the audio format that the device was set to, which may not be the requested format. May also be used to return the current audio format—do this by passing an "audio format" of `AFMT_QUERY`.

oss_audio_device.**channels**(*nchannels*)
Set the number of output channels to *nchannels*. A value of 1 indicates monophonic sound, 2 stereophonic. Some devices may have more than 2 channels, and some high-end devices may not support mono. Returns the number of channels the device was set to.

oss_audio_device.**speed**(*samplerate*)
Try to set the audio sampling rate to *samplerate* samples per second. Returns the rate actually set. Most sound devices don't support arbitrary sampling rates. Common rates are:

采样率	描述
8000	/dev/audio 的默认采样率
11025	语音录音
22050	
44100	CD 质量的音频（16 位采样和 2 通道）
96000	DVD 质量的音频（24 位采样）

oss_audio_device.**sync**()
Wait until the sound device has played every byte in its buffer. (This happens implicitly when the device is closed.) The OSS documentation recommends closing and re-opening the device rather than using *sync()*.

oss_audio_device.**reset**()
Immediately stop playing or recording and return the device to a state where it can accept commands. The OSS documentation recommends closing and re-opening the device after calling *reset()*.

oss_audio_device.**post**()
Tell the driver that there is likely to be a pause in the output, making it possible for the device to handle the pause more intelligently. You might use this after playing a spot sound effect, before waiting for user input, or before doing disk I/O.

The following convenience methods combine several ioctls, or one ioctl and some simple calculations.

oss_audio_device.**setparameters**(*format*, *nchannels*, *samplerate*[, *strict=False*])
Set the key audio sampling parameters—sample format, number of channels, and sampling rate—in one method call. *format*, *nchannels*, and *samplerate* should be as specified in the *setfmt()*, *channels()*, and *speed()* methods. If *strict* is true, *setparameters()* checks to see if each parameter was actually set to the requested value, and raises *OSSAudioError* if not. Returns a tuple (*format*, *nchannels*, *samplerate*) indicating the parameter values that were actually set by the device driver (i.e., the same as the return values of *setfmt()*, *channels()*, and *speed()*).

For example,

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

is equivalent to

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

oss_audio_device.**bufsize**()
Returns the size of the hardware buffer, in samples.

oss_audio_device.**obufcount**()
Returns the number of samples that are in the hardware buffer yet to be played.

oss_audio_device.**obuffree**()
Returns the number of samples that could be queued into the hardware buffer to be played without blocking.

Audio device objects also support several read-only attributes:

oss_audio_device.**closed**
Boolean indicating whether the device has been closed.

oss_audio_device.**name**
String containing the name of the device file.

oss_audio_device.**mode**
The I/O mode for the file, either "r", "rw", or "w".

22.9.2 Mixer Device Objects

The mixer object provides two file-like methods:

oss_mixer_device.**close**()
This method closes the open mixer device file. Any further attempts to use the mixer after this file is closed will raise an *OSError*.

oss_mixer_device.**fileno**()
Returns the file handle number of the open mixer device file.

在 3.2 版更改: Mixer objects also support the context management protocol.

The remaining methods are specific to audio mixing:

oss_mixer_device.**controls**()
This method returns a bitmask specifying the available mixer controls ("Control" being a specific mixable "channel", such as SOUND_MIXER_PCM or SOUND_MIXER_SYNTH). This bitmask indicates a subset of all available mixer controls—the SOUND_MIXER_* constants defined at module level. To determine if, for example, the current mixer object supports a PCM mixer, use the following Python code:

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

For most purposes, the SOUND_MIXER_VOLUME (master volume) and SOUND_MIXER_PCM controls should suffice—but code that uses the mixer should be flexible when it comes to choosing mixer controls. On the Gravis Ultrasound, for example, SOUND_MIXER_VOLUME does not exist.

oss_mixer_device.**stereocontrols**()
Returns a bitmask indicating stereo mixer controls. If a bit is set, the corresponding control is stereo; if it is unset, the control is either monophonic or not supported by the mixer (use in combination with *controls()* to determine which).

See the code example for the *controls()* function for an example of getting data from a bitmask.

oss_mixer_device.**reccontrols**()
Returns a bitmask specifying the mixer controls that may be used to record. See the code example for *controls()* for an example of reading from a bitmask.

oss_mixer_device.**get**(*control*)
Returns the volume of a given mixer control. The returned volume is a 2-tuple (left_volume, right_volume). Volumes are specified as numbers from 0 (silent) to 100 (full volume). If the control is monophonic, a 2-tuple is still returned, but both volumes are the same.

Raises *OSSAudioError* if an invalid control is specified, or *OSError* if an unsupported control is specified.

oss_mixer_device.**set**(*control*, *(left*, *right)*)
Sets the volume for a given mixer control to (left,right). left and right must be ints and between 0

(silent) and 100 (full volume). On success, the new volume is returned as a 2-tuple. Note that this may not be exactly the same as the volume specified, because of the limited resolution of some soundcard's mixers.

Raises *OSSAudioError* if an invalid mixer control was specified, or if the specified volumes were out-of-range.

oss_mixer_device.**get_recsrc**()
This method returns a bitmask indicating which control(s) are currently being used as a recording source.

oss_mixer_device.**set_recsrc**(*bitmask*)
Call this function to specify a recording source. Returns a bitmask indicating the new recording source (or sources) if successful; raises *OSError* if an invalid source was specified. To set the current recording source to the microphone input:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```

CHAPTER 23

国际化

本章中介绍的模块通过提供选择要在程序信息中使用的语言的机制或通过定制输出以匹配本地约定来帮助你编写不依赖于语言和区域设置的软件。

本章中描述的模块列表是：

23.1 gettext —多语种国际化服务

源代码： Lib/gettext.py

The *gettext* module provides internationalization (I18N) and localization (L10N) services for your Python modules and applications. It supports both the GNU `gettext` message catalog API and a higher level, class-based API that may be more appropriate for Python files. The interface described below allows you to write your module and application messages in one natural language, and provide a catalog of translated messages for running under different natural languages.

同时还给出一些本地化 Python 模块及应用程序的小技巧。

23.1.1 GNU gettext API

模块*gettext* 定义了下列 API，这与 **gettext** API 类似。如果你使用该 API，将会对整个应用程序产生全局的影响。如果你的应用程序支持多语种，而语言选择取决于用户的区域设置，这通常正是你所想要的。而如果你正在本地化某个 Python 模块，或者你的应用程序需要在运行时切换语言，相反你或许想用基于类的 API。

`gettext.`**`bindtextdomain`**(*domain*, *localedir=None*)

Bind the *domain* to the locale directory *localedir*. More concretely, *gettext* will look for binary `.mo` files for the given domain using the path (on Unix): `localedir/language/LC_MESSAGES/domain.mo`, where *languages* is searched for in the environment variables `LANGUAGE`, `LC_ALL`, `LC_MESSAGES`, and `LANG` respectively.

如果遗漏了 *localedir* 或者设置为 None，那么将返回当前 *domain* 所绑定的值[1]

gettext.**bind_textdomain_codeset**(*domain*, *codeset=None*)
将 *domain* 绑定到 *codeset*，修改*lgettext()*，*ldgettext()*，*lngettext()* 和*ldngettext()* 函数返回的字节串的字符编码。如果省略了 *codeset*，则返回当前绑定的编码集。

gettext.**textdomain**(*domain=None*)
修改或查询当前的全局域。如果 *domain* 为 None，则返回当前的全局域，不为 None 则将全局域设置为 *domain*，并返回它。

gettext.**gettext**(*message*)
返回 *message* 的本地化翻译，依据包括当前的全局域、语言和区域目录。本函数在本地命名空间中通常有别名 _() （参考下面的示例）。

gettext.**dgettext**(*domain*, *message*)
与*gettext()* 类似，但在指定的 *domain* 中查找 message。

gettext.**ngettext**(*singular*, *plural*, *n*)
与*gettext()* 类似，但考虑了复数形式。如果找到了翻译，则将 *n* 代入复数公式，然后返回得出的消息（某些语言具有两种以上的复数形式）。如果未找到翻译，则 *n* 为 1 时返回 *singular*，为其他数时返回 *plural*。

复数公式取自编目头文件。它是 C 或 Python 表达式，有一个自变量 *n*，该表达式计算的是所需复数形式在编目中的索引号。关于在 .po 文件中使用的确切语法和各种语言的公式，请参阅 GNU gettext 文档 。

gettext.**dngettext**(*domain*, *singular*, *plural*, *n*)
与*ngettext()* 类似，但在指定的 *domain* 中查找 message。

gettext.**lgettext**(*message*)

gettext.**ldgettext**(*domain*, *message*)

gettext.**lngettext**(*singular*, *plural*, *n*)

gettext.**ldngettext**(*domain*, *singular*, *plural*, *n*)
与前缀中没有 l 的相应函数等效（*gettext()*，*dgettext()*，*ngettext()* 和*dngettext()*），但是如果没有用*bind_textdomain_codeset()* 显式设置其他编码，则返回的翻译将以首选系统编码来编码字节串。

> **警告：** These functions should be avoided in Python 3, because they return encoded bytes. It's much better to use alternatives which return Unicode strings instead, since most Python applications will want to manipulate human readable text as strings instead of bytes. Further, it's possible that you may get unexpected Unicode-related exceptions if there are encoding problems with the translated strings. It is possible that the l*() functions will be deprecated in future Python versions due to their inherent problems and limitations.

注意，GNU **gettext** 还定义了 dcgettext() 方法，但它被认为不实用，因此目前没有实现它。

这是该 API 的典型用法示例:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

[1] The default locale directory is system dependent; for example, on RedHat Linux it is /usr/share/locale, but on Solaris it is /usr/lib/locale. The *gettext* module does not try to support these system dependent defaults; instead its default is sys.prefix/share/locale. For this reason, it is always best to call *bindtextdomain()* with an explicit absolute path at the start of your application.

23.1.2 基于类的 API

The class-based API of the *gettext* module gives you more flexibility and greater convenience than the GNU **gettext** API. It is the recommended way of localizing your Python applications and modules. `gettext` defines a "translations" class which implements the parsing of GNU `.mo` format files, and has methods for returning strings. Instances of this "translations" class can also install themselves in the built-in namespace as the function `_()`.

`gettext.`**`find`**(*domain*, *localedir=None*, *languages=None*, *all=False*)

This function implements the standard `.mo` file search algorithm. It takes a *domain*, identical to what *textdomain()* takes. Optional *localedir* is as in *bindtextdomain()* Optional *languages* is a list of strings, where each string is a language code.

如果没有传入 *localedir*，则使用默认的系统区域设置目录。[2] 如果没有传入 *languages*，则搜索以下环境变量：`LANGUAGE`、`LC_ALL`、`LC_MESSAGES` 和 `LANG`。从这些变量返回的第一个非空值将用作 *languages* 变量。环境变量应包含一个语言列表，由冒号分隔，该列表会被按冒号拆分，以产生所需的语言代码字符串列表。

find() 将扩展并规范化 language，然后遍历它们，搜索由这些组件构建的现有文件:

localedir`/`*language*`/LC_MESSAGES/`*domain*`.mo`

find() 返回找到类似的第一个文件名。如果找不到这样的文件，则返回 `None`。如果传入了 *all*，它将返回一个列表，包含所有文件名，并按它们在语言列表或环境变量中出现的顺序排列。

`gettext.`**`translation`**(*domain*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*, *codeset=None*)

Return a `Translations` instance based on the *domain*, *localedir*, and *languages*, which are first passed to *find()* to get a list of the associated `.mo` file paths. Instances with identical `.mo` file names are cached. The actual class instantiated is either *class_* if provided, otherwise *GNUTranslations*. The class's constructor must take a single *file object* argument. If provided, *codeset* will change the charset used to encode translated strings in the *lgettext()* and *lngettext()* methods.

如果找到多个文件，后找到的文件将用作先前文件的替补。为了设置替补，将使用*copy.copy()* 从缓存中克隆每个 translation 对象。实际的实例数据仍在缓存中共享。

如果 `.mo` 文件未找到，且 *fallback* 为 false（默认值），则本函数引发*OSError* 异常，如果 *fallback* 为 true，则返回一个*NullTranslations* 实例。

在 3.3 版更改: *IOError* 代替*OSError* 被引发。

`gettext.`**`install`**(*domain*, *localedir=None*, *codeset=None*, *names=None*)

根据传入*translation()* 函数的 *domain*、*localedir* 和 *codeset*，在 Python 内建命名空间中安装 `_()` 函数。

names 参数的信息请参阅 translation 对象的*install()* 方法的描述。

如下所示，通常将字符串包括在 `_()` 函数的调用中，以标记应用程序中待翻译的字符串，就像这样:

```
print(_('This string will be translated.'))
```

为了方便，一般将 `_()` 函数安装在 Python 内建命名空间中，以便在应用程序的所有模块中轻松访问它。

[2] 参阅上方*bindtextdomain()* 的脚注。

NullTranslations 类

translation 类实际实现的是，将原始源文件消息字符串转换为已翻译的消息字符串。所有 translation 类使用的基类为*NullTranslations*，它提供了基本的接口，可用于编写自己定制的 translation 类。以下是 NullTranslations 的方法：

class gettext.**NullTranslations**(*fp=None*)

接受一个可选参数文件对象 *fp*，该参数会被基类忽略。初始化由派生类设置的“protected”（受保护的）实例变量 *_info* 和 *_charset*，与 *_fallback* 类似，但它是通过*add_fallback()* 来设置的。如果 *fp* 不为 None，就会调用 self._parse(fp)。

_parse(*fp*)

No-op’d in the base class, this method takes file object *fp*, and reads the data from the file, initializing its message catalog. If you have an unsupported message catalog file format, you should override this method to parse your format.

add_fallback(*fallback*)

添加 *fallback* 为当前 translation 对象的替补对象。如果 translation 对象无法为指定消息提供翻译，则应向替补查询。

gettext(*message*)

如果设置了替补，则转发 gettext() 给替补。否则返回 *message*。须在派生类中重写。

ngettext(*singular*, *plural*, *n*)

如果设置了替补，则转发 ngettext() 给替补。否则，*n* 为 1 时返回 *singular*，为其他时返回 *plural*。须在派生类中重写。

lgettext(*message*)

lngettext(*singular*, *plural*, *n*)

与*gettext()* 和*ngettext()* 等效，但是如果没有用*set_output_charset()* 显式设置编码，则返回的翻译将以首选系统编码来编码字节串。在派生类中被重写。

> **警告：** 应避免在 Python 3 中使用这些方法。请参阅*lgettext()* 函数的警告。

info()

Return the “protected” _info variable.

charset()

返回消息编目文件的编码。

output_charset()

返回由*lgettext()* 和*lngettext()* 翻译的消息的编码。

set_output_charset(*charset*)

更改翻译出的消息的编码。

install(*names=None*)

本方法将*gettext()* 安装至内建命名空间，并绑定为 _。

If the *names* parameter is given, it must be a sequence containing the names of functions you want to install in the builtins namespace in addition to _(). Supported names are 'gettext', 'ngettext', 'lgettext' and 'lngettext'.

注意，这仅仅是使 _() 函数在应用程序中生效的一种方法，尽管也是最方便的方法。由于它会影响整个应用程序全局，特别是内建命名空间，因此已经本地化的模块不应该安装 _()，而是应该用下列代码使 _() 在各自模块中生效:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

这只将 `_()` 放在其模块的全局命名空间中，所以只影响其模块内的调用。

GNUTranslations 类

gettext 模块提供了一个派生自 *NullTranslations* 的其他类：*GNUTranslations*。该类重写了 `_parse()`，同时能以大端序和小端序格式读取 GNU **gettext** 格式的 `.mo` 文件。

GNUTranslations parses optional meta-data out of the translation catalog. It is convention with GNU **gettext** to include meta-data as the translation for the empty string. This meta-data is in **RFC 822**-style `key: value` pairs, and should contain the `Project-Id-Version` key. If the key `Content-Type` is found, then the `charset` property is used to initialize the "protected" `_charset` instance variable, defaulting to `None` if not found. If the charset encoding is specified, then all message ids and message strings read from the catalog are converted to Unicode using this encoding, else ASCII encoding is assumed.

由于消息 ID 也是作为 Unicode 字符串读取的，因此所有 `*gettext()` 方法都假定消息 ID 为 Unicode 字符串，而不是字节串。

整个键/值对集合将被放入一个字典，并设置为"protected"（受保护的）`_info` 实例变量。

如果 `.mo` 文件的魔法值 (magic number) 无效，或遇到意外的主版本号，或在读取文件时发生其他问题，则实例化 *GNUTranslations* 类会引发 *OSError*。

class gettext.**GNUTranslations**

下列方法是根据基类实现重写的：

gettext(*message*)

在编目中查找 *message* ID，并以 Unicode 字符串形式返回相应的消息字符串。如果在编目中没有 *message* ID 条目，且配置了替补，则查找请求将被转发到替补的 *gettext()* 方法。否则，返回 *message* ID。

ngettext(*singular*, *plural*, *n*)

查找消息 ID 的复数形式。*singular* 用作消息 ID，用于在编目中查找，同时 *n* 用于确定使用哪种复数形式。返回的消息字符串是 Unicode 字符串。

如果在编目中没有找到消息 ID，且配置了替补，则查找请求将被转发到替补的 *ngettext()* 方法。否则，当 *n* 为 1 时返回 *singular*，其他情况返回 *plural*。

例如：

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

lgettext(*message*)

lngettext(*singular*, *plural*, *n*)

与 *gettext()* 和 *ngettext()* 等效，但是如果没有用 *set_output_charset()* 显式设置编码，则返回的翻译将以首选系统编码来编码字节串。

> **警告：** 应避免在 Python 3 中使用这些方法。请参阅*lgettext()* 函数的警告。

Solaris 消息编目支持

Solaris 操作系统定义了自己的二进制 `.mo` 文件格式，但由于找不到该格式的文档，因此目前不支持该格式。

编目构造器

GNOME 用的 *gettext* 模块是 James Henstridge 写的版本，但该版本的 API 略有不同。它文档中的用法是:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

为了与本模块的旧版本兼容，`Catalog()` 函数是上述*translation()* 函数的别名。

本模块与 Henstridge 的模块有一个区别：他的编目对象支持通过映射 API 进行访问，但是该特性似乎从未使用过，因此目前不支持该特性。

23.1.3 国际化 (I18N) 你的程序和模块

国际化 (I18N) 是指使程序可切换多种语言的操作。本地化 (L10N) 是指程序的适配能力，一旦程序被国际化，就能适配当地的语言和文化习惯。为了向 Python 程序提供不同语言的消息，需要执行以下步骤:

1. 准备程序或模块，将可翻译的字符串特别标记起来
2. 在已标记的文件上运行一套工具，用来生成原始消息编目
3. create language specific translations of the message catalogs
4. 使用*gettext* 模块，以便正确翻译消息字符串

为了准备代码以达成 I18N，需要巡视文件中的所有字符串。所有要翻译的字符串都应包括在 `_('...')` 内，以此打上标记——也就是调用 `_()` 函数。例如:

```
filename = 'mylog.txt'
message = _('writing a log message')
fp = open(filename, 'w')
fp.write(message)
fp.close()
```

在这个例子中，字符串 `'writing a log message'` 被标记为待翻译，而字符串 `'mylog.txt'` 和 `'w'` 没有被标记。

有一些工具可以将待翻译的字符串提取出来。原版的 GNU **gettext** 仅支持 C 或 C++ 源代码，但其扩展版 **xgettext** 可以扫描多种语言的代码（包括 Python），来找出标记为可翻译的字符串。Babel 是一个 Python 国际化库，其中包含一个 `pybabel` 脚本，用于提取并编译消息编目。François Pinard 写的称为 **xpot** 的程序也能完成类似的工作，可从他的 po-utils 软件包 中获取。

（Python 还包括了这些程序的纯 Python 版本，称为 **pygettext.py** 和 **msgfmt.py**，某些 Python 发行版已经安装了它们。**pygettext.py** 类似于 **xgettext**，但只能理解 Python 源代码，无法处理诸如 C 或 C++ 的其他编程语言。**pygettext.py** 支持的命令行界面类似于 **xgettext**，查看其详细用法请运行 `pygettext.py --help`。**msgfmt.py** 与 GNU **msgfmt** 是二进制兼容的。有了这两个程序，可以不需要 GNU **gettext** 包来国际化 Python 应用程序。）

xgettext、**pygettext** 或类似工具生成的 .po 文件就是消息编目。它们是结构化的人类可读文件，包含源代码中所有被标记的字符串，以及这些字符串的翻译的占位符。

然后把这些 .po 文件的副本交给各个人工译者，他们为所支持的每种自然语言编写翻译。译者以 < 语言名称>.po 文件的形式发送回翻译完的某个语言的版本，将该文件用 **msgfmt** 程序编译为机器可读的 .mo 二进制编目文件。*gettext* 模块使用 .mo 文件在运行时进行实际的翻译处理。

如何在代码中使用*gettext* 模块取决于国际化单个模块还是整个应用程序。接下来的两节将讨论每种情况。

本地化你的模块

If you are localizing your module, you must take care not to make global changes, e.g. to the built-in namespace. You should not use the GNU `gettext` API but instead the class-based API.

假设你的模块叫做 "spam"，并且该模块的各种自然语言翻译 .mo 文件存放于 /usr/share/locale，为 GNU **gettext** 格式。以下内容应放在模块顶部:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

本地化你的应用程序

如果正在本地化应用程序，可以将 _() 函数全局安装到内建命名空间中，通常在应用程序的主文件中安装。这将使某个应用程序的所有文件都能使用 _('...')，而不必在每个文件中显式安装它。

最简单的情况，就只需将以下代码添加到应用程序的主程序文件中:

```
import gettext
gettext.install('myapplication')
```

如果需要设置语言环境目录，可以将其传递给*install()* 函数:

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

即时更改语言

如果程序需要同时支持多种语言，则可能需要创建多个翻译实例，然后在它们之间进行显式切换，如下所示:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

延迟翻译

在大多数代码中，字符串会在编写位置进行翻译。但偶尔需要将字符串标记为待翻译，实际翻译却推迟到后面。一个典型的例子是:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print(a)
```

此处希望将 animals 列表中的字符串标记为可翻译，但不希望在打印之前对它们进行翻译。

这是处理该情况的一种方式:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

该方法之所以可行，是因为 _() 的虚定义只是简单地返回了原本的字符串。并且该虚定义将临时覆盖内建命名空间中 _() 的定义（直到 del 命令）。即使先前在本地命名空间中已经有了 _() 的定义也请注意。

注意，第二次使用 _() 不会认为“a”可以传递给 **gettext** 程序去翻译，因为该参数不是字符串文字。

解决该问题的另一种方法是下面这个例子:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

这种情况下标记可翻译的字符串使用的是函数 N_()，该函数不会与 _() 的任何定义冲突。但是，需要教会消息提取程序去寻找用 N_() 标记的可翻译字符串。**xgettext**, **pygettext**, pybabel extract 和 **xpot** 都支持此功能，使用 -k 命令行开关即可。这里选择 N_() 为名称完全是任意的，它也能轻易改为 MarkThisStringForTranslation()。

23.1.4 致谢

以下人员为创建此模块贡献了代码、反馈、设计建议、早期实现和宝贵的经验：

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

备注

23.2 locale —国际化服务

源代码： Lib/locale.py

The *locale* module opens access to the POSIX locale database and functionality. The POSIX locale mechanism allows programmers to deal with certain cultural issues in an application, without requiring the programmer to know all the specifics of each country where the software is executed.

The *locale* module is implemented on top of the _locale module, which in turn uses an ANSI C locale implementation if available.

The *locale* module defines the following exception and functions:

exception locale.**Error**
Exception raised when the locale passed to *setlocale()* is not recognized.

locale.**setlocale**(*category*, *locale=None*)
If *locale* is given and not None, *setlocale()* modifies the locale setting for the *category*. The available categories are listed in the data description below. *locale* may be a string, or an iterable of two strings (language code and encoding). If it's an iterable, it's converted to a locale name using the locale aliasing engine. An empty string specifies the user's default settings. If the modification of the locale fails, the exception *Error* is raised. If successful, the new locale setting is returned.

If *locale* is omitted or None, the current setting for *category* is returned.

setlocale() is not thread-safe on most systems. Applications typically start with a call of

```
import locale
locale.setlocale(locale.LC_ALL, '')
```

This sets the locale for all categories to the user's default setting (typically specified in the LANG environment variable). If the locale is not changed thereafter, using multithreading should not cause problems.

locale.**localeconv**()
以字典的形式返回本地约定的数据库。此字典具有以下字符串作为键：

类别	（Windows 注册表的）键	含义
LC_NUMERIC	'decimal_point'	小数点字符。
	'grouping'	Sequence of numbers specifying which relative positions the 'thousands_sep' is expected. If the sequence is terminated with *CHAR_MAX*, no further grouping is performed. If the sequence terminates with a 0, the last group size is repeatedly used.
	'thousands_sep'	组之间使用的字符。
LC_MONETARY	'int_curr_symbol'	国际货币符号。
	'currency_symbol'	当地货币符号。
	'p_cs_precedes/n_cs_precedes'	货币符号是否在值之前（对于正值或负值）。
	'p_sep_by_space/n_sep_by_space'	货币符号是否通过空格与值分隔（对于正值或负值）。
	'mon_decimal_point'	用于货币金额的小数点。
	'frac_digits'	货币值的本地格式中使用的小数位数。
	'int_frac_digits'	货币价值的国际格式中使用的小数位数。
	'mon_thousands_sep'	用于货币值的组分隔符。
	'mon_grouping'	相当于 'grouping'，用于货币价值。
	'positive_sign'	用于标注正货币价值的符号。
	'negative_sign'	用于注释负货币价值的符号。
	'p_sign_posn/n_sign_posn'	符号的位置（对于正值或负值），见下文。

可以将所有数值设置为*CHAR_MAX*，以指示此语言环境中未指定任何值。

下面给出了 'p_sign_posn' 和 'n_sign_posn' 的可能值。

值	解释
0	被括号括起来的货币和金额。
1	该标志应位于值和货币符号之前。
2	该标志应位于值和货币符号之后。
3	标志应该紧跟在值之前。
4	标志应该紧跟值项。
CHAR_MAX	此语言环境中未指定任何内容。

The function sets temporarily the LC_CTYPE locale to the LC_NUMERIC locale or the LC_MONETARY locale if locales are different and numeric or monetary strings are non-ASCII. This temporary change affects other threads.

在 3.6.5 版更改: The function now sets temporarily the LC_CTYPE locale to the LC_NUMERIC locale in some cases.

locale.**nl_langinfo**(*option*)

Return some locale-specific information as a string. This function is not available on all systems, and the set of possible options might also vary across platforms. The possible argument values are numbers, for which symbolic constants are available in the locale module.

The *nl_langinfo()* function accepts one of the following keys. Most descriptions are taken from the corresponding description in the GNU C library.

locale.**CODESET**
Get a string with the name of the character encoding used in the selected locale.

locale.**D_T_FMT**
Get a string that can be used as a format string for *time.strftime()* to represent date and time in a locale-specific way.

locale.**D_FMT**
Get a string that can be used as a format string for *time.strftime()* to represent a date in a locale-specific way.

locale.**T_FMT**
Get a string that can be used as a format string for *time.strftime()* to represent a time in a locale-specific way.

locale.**T_FMT_AMPM**
Get a format string for *time.strftime()* to represent time in the am/pm format.

DAY_1 ... DAY_7
Get the name of the n-th day of the week.

注解： This follows the US convention of DAY_1 being Sunday, not the international convention (ISO 8601) that Monday is the first day of the week.

ABDAY_1 ... ABDAY_7
Get the abbreviated name of the n-th day of the week.

MON_1 ... MON_12
Get the name of the n-th month.

ABMON_1 ... ABMON_12
Get the abbreviated name of the n-th month.

locale.**RADIXCHAR**
Get the radix character (decimal dot, decimal comma, etc.).

locale.**THOUSEP**
Get the separator character for thousands (groups of three digits).

locale.**YESEXPR**
Get a regular expression that can be used with the regex function to recognize a positive response to a yes/no question.

注解： The expression is in the syntax suitable for the regex() function from the C library, which might differ from the syntax used in *re*.

---

locale.**NOEXPR**
Get a regular expression that can be used with the regex(3) function to recognize a negative response to a yes/no question.

locale.**CRNCYSTR**
Get the currency symbol, preceded by "-" if the symbol should appear before the value, "+" if the symbol should appear after the value, or "." if the symbol should replace the radix character.

locale.**ERA**
Get a string that represents the era used in the current locale.

Most locales do not define this value. An example of a locale which does define this value is the Japanese one. In Japan, the traditional representation of dates includes the name of the era corresponding to the then-emperor's reign.

Normally it should not be necessary to use this value directly. Specifying the `E` modifier in their format strings causes the *time.strftime()* function to use this information. The format of the returned string is not specified, and therefore you should not assume knowledge of it on different systems.

locale.**ERA_D_T_FMT**
Get a format string for *time.strftime()* to represent date and time in a locale-specific era-based way.

locale.**ERA_D_FMT**
Get a format string for *time.strftime()* to represent a date in a locale-specific era-based way.

locale.**ERA_T_FMT**
Get a format string for *time.strftime()* to represent a time in a locale-specific era-based way.

locale.**ALT_DIGITS**
Get a representation of up to 100 values used to represent the values 0 to 99.

locale.**getdefaultlocale**([*envvars*])
Tries to determine the default locale settings and returns them as a tuple of the form `(language code, encoding)`.

According to POSIX, a program which has not called `setlocale(LC_ALL, '')` runs using the portable `'C'` locale. Calling `setlocale(LC_ALL, '')` lets it use the default locale as defined by the `LANG` variable. Since we do not want to interfere with the current locale setting we thus emulate the behavior in the way described above.

To maintain compatibility with other platforms, not only the `LANG` variable is tested, but a list of variables given as envvars parameter. The first found to be defined will be used. *envvars* defaults to the search path used in GNU gettext; it must always contain the variable name `'LANG'`. The GNU gettext search path contains `'LC_ALL'`, `'LC_CTYPE'`, `'LANG'` and `'LANGUAGE'`, in that order.

Except for the code `'C'`, the language code corresponds to **RFC 1766**. *language code* and *encoding* may be `None` if their values cannot be determined.

locale.**getlocale**(*category=LC_CTYPE*)
Returns the current setting for the given locale category as sequence containing *language code*, *encoding*. *category* may be one of the `LC_*` values except *LC_ALL*. It defaults to *LC_CTYPE*.

Except for the code `'C'`, the language code corresponds to **RFC 1766**. *language code* and *encoding* may be `None` if their values cannot be determined.

locale.**getpreferredencoding**(*do_setlocale=True*)
Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

On some systems, it is necessary to invoke *setlocale()* to obtain the user preferences, so this function is not thread-safe. If invoking setlocale is not necessary or desired, *do_setlocale* should be set to `False`.

locale.**normalize**(*localename*)
Returns a normalized locale code for the given locale name. The returned locale code is formatted for use with *setlocale()*. If normalization fails, the original name is returned unchanged.

If the given encoding is not known, the function defaults to the default encoding for the locale code just like *setlocale()*.

locale.**resetlocale**(*category=LC_ALL*)
Sets the locale for *category* to the default setting.

The default setting is determined by calling *getdefaultlocale()*. *category* defaults to *LC_ALL*.

locale.**strcoll**(*string1*, *string2*)
Compares two strings according to the current *LC_COLLATE* setting. As any other compare function, returns a negative, or a positive value, or `0`, depending on whether *string1* collates before or after *string2* or is equal to it.

locale.**strxfrm**(*string*)
Transforms a string to one that can be used in locale-aware comparisons. For example, `strxfrm(s1) < strxfrm(s2)` is equivalent to `strcoll(s1, s2) < 0`. This function can be used when the same string is compared repeatedly, e.g. when collating a sequence of strings.

locale.**format**(*format*, *val*, *grouping=False*, *monetary=False*)
Formats a number *val* according to the current *LC_NUMERIC* setting. The format follows the conventions of the `%` operator. For floating point values, the decimal point is modified if appropriate. If *grouping* is true, also takes the grouping into account.

If *monetary* is true, the conversion uses monetary thousands separator and grouping strings.

Please note that this function will only work for exactly one %char specifier. For whole format strings, use *format_string()*.

locale.**format_string**(*format*, *val*, *grouping=False*)
Processes formatting specifiers as in `format % val`, but takes the current locale settings into account.

locale.**currency**(*val*, *symbol=True*, *grouping=False*, *international=False*)
Formats a number *val* according to the current *LC_MONETARY* settings.

The returned string includes the currency symbol if *symbol* is true, which is the default. If *grouping* is true (which is not the default), grouping is done with the value. If *international* is true (which is not the default), the international currency symbol is used.

Note that this function will not work with the 'C' locale, so you have to set a locale via *setlocale()* first.

locale.**str**(*float*)
Formats a floating point number using the same format as the built-in function `str(float)`, but takes the decimal point into account.

locale.**delocalize**(*string*)
Converts a string into a normalized number string, following the *LC_NUMERIC* settings.

3.5 新版功能.

locale.**atof**(*string*)
Converts a string to a floating point number, following the *LC_NUMERIC* settings.

locale.**atoi**(*string*)
Converts a string to an integer, following the *LC_NUMERIC* conventions.

locale.**LC_CTYPE**
Locale category for the character type functions. Depending on the settings of this category, the functions of module *string* dealing with case change their behaviour.

locale.**LC_COLLATE**
Locale category for sorting strings. The functions *strcoll()* and *strxfrm()* of the *locale* module are affected.

locale.**LC_TIME**
Locale category for the formatting of time. The function *time.strftime()* follows these conventions.

locale.**LC_MONETARY**
Locale category for formatting of monetary values. The available options are available from the *localeconv()* function.

locale.**LC_MESSAGES**
Locale category for message display. Python currently does not support application specific locale-aware messages. Messages displayed by the operating system, like those returned by *os.strerror()* might be affected by this category.

locale.**LC_NUMERIC**
Locale category for formatting numbers. The functions *format()*, *atoi()*, *atof()* and *str()* of the *locale* module are affected by that category. All other numeric formatting operations are not affected.

locale.**LC_ALL**
Combination of all locale settings. If this flag is used when the locale is changed, setting the locale for all categories is attempted. If that fails for any category, no category is changed at all. When the locale is retrieved using this flag, a string indicating the setting for all categories is returned. This string can be later used to restore the settings.

locale.**CHAR_MAX**
This is a symbolic constant used for different values returned by *localeconv()*.

示例:

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')    # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')   # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)   # restore saved locale
```

23.2.1 Background, details, hints, tips and caveats

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementation are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

Initially, when a program is started, the locale is the `C` locale, no matter what the user's preferred locale is. There is one exception: the *LC_CTYPE* category is changed at startup to set the current locale encoding to the user's preferred locale encoding. The program must explicitly say that it wants the user's preferred locale settings for other categories by calling `setlocale(LC_ALL, '')`.

It is generally a bad idea to call *setlocale()* in some library routine, since as a side effect it affects the entire program. Saving and restoring it is almost as bad: it is expensive and affects other threads that happen to run before the settings have been restored.

If, when coding a module for general use, you need a locale independent version of an operation that is affected by the locale (such as certain formats used with *time.strftime()*), you will have to find a way to do it without using the standard library routine. Even better is convincing yourself that using locale settings is okay. Only as a last resort should you document that your module is not compatible with non-`C` locale settings.

The only way to perform numeric operations according to the locale is to use the special functions defined by this module: *atof()*, *atoi()*, *format()*, *str()*.

There is no way to perform case conversions and character classifications according to the locale. For (Unicode) text strings these are done according to the character value only, while for byte strings, the conversions and classifications are

done according to the ASCII value of the byte, and bytes whose high bit is set (i.e., non-ASCII bytes) are never converted or considered part of a character class such as letter or whitespace.

23.2.2 For extension writers and programs that embed Python

Extension modules should never call *setlocale()*, except to find out what the current locale is. But since the return value can only be used portably to restore it, that is not very useful (except perhaps to find out whether or not the locale is C).

When Python code uses the *locale* module to change the locale, this also affects the embedding application. If the embedding application doesn't want this to happen, it should remove the _locale extension module (which does all the work) from the table of built-in modules in the config.c file, and make sure that the _locale module is not accessible as a shared library.

23.2.3 Access to message catalogs

locale.**gettext**(*msg*)

locale.**dgettext**(*domain*, *msg*)

locale.**dcgettext**(*domain*, *msg*, *category*)

locale.**textdomain**(*domain*)

locale.**bindtextdomain**(*domain*, *dir*)

The locale module exposes the C library's gettext interface on systems that provide this interface. It consists of the functions gettext(), dgettext(), dcgettext(), textdomain(), bindtextdomain(), and bind_textdomain_codeset(). These are similar to the same functions in the *gettext* module, but use the C library's binary format for message catalogs, and the C library's search algorithms for locating message catalogs.

Python applications should normally find no need to invoke these functions, and should use *gettext* instead. A known exception to this rule are applications that link with additional C libraries which internally invoke gettext() or dcgettext(). For these applications, it may be necessary to bind the text domain, so that the libraries can properly locate their message catalogs.

CHAPTER 24

程序框架

本章中描述的模块是很大程度上决定程序结构的框架。目前，这里描述的模块都面向编写命令行接口。

本章描述的完整模块列表如下：

24.1 `turtle` —海龟绘图

源码： Lib/turtle.py

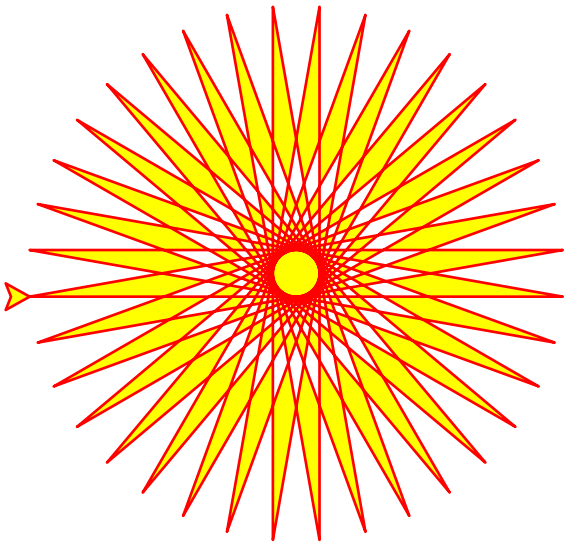
24.1.1 概述

Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzig and Seymour Papert in 1966.

请想象绘图区有一只机器海龟，起始位置在 x-y 平面的 (0, 0) 点。先执行 `import turtle`，再执行 `turtle.forward(15)`，它将 (在屏幕上) 朝所面对的 x 轴正方向前进 15 像素，随着它的移动画出一条线段。再执行 `turtle.right(25)`，它将原地右转 25 度。

Turtle star

使用海龟绘图可以编写重复执行简单动作的程序画出精细复杂的形状。



```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

通过组合使用此类命令，可以轻松地绘制出精美的形状和图案。

turtle 模块是基于 Python 标准发行版 2.5 以来的同名模块重新编写并进行了功能扩展。

新模块尽量保持了原模块的特点，并且 (几乎)100% 与其兼容。这就意味着初学编程者能够以交互方式使用模块的所有命令、类和方法——运行 IDLE 时注意加 `-n` 参数。

turtle 模块提供面向对象和面向过程两种形式的海龟绘图基本组件。由于它使用*tkinter* 实现基本图形界面，因此需要安装了 Tk 支持的 Python 版本。

面向对象的接口主要使用“2+2”个类：

1. *TurtleScreen* 类定义图形窗口作为绘图海龟的运动场。它的构造器需要一个 `tkinter.Canvas` 或*ScrolledCanvas* 作为参数。应在*turtle* 作为某个程序的一部分的时候使用。

   Screen() 函数返回一个*TurtleScreen* 子类的单例对象。此函数应在*turtle* 作为独立绘图工具时使用。作为一个单例对象，其所属的类是不可被继承的。

   TurtleScreen/Screen 的所有方法还存在对应的函数，即作为面向过程的接口组成部分。

2. *RawTurtle* (别名: *RawPen*) 类定义海龟对象在*TurtleScreen* 上绘图。它的构造器需要一个 Canvas, ScrolledCanvas 或 TurtleScreen 作为参数，以指定 RawTurtle 对象在哪里绘图。

   从 RawTurtle 派生出子类*Turtle* (别名: `Pen`)，该类对象在*Screen* 实例上绘图，如果实例不存在则会自动创建。

   RawTurtle/Turtle 的所有方法也存在对应的函数，即作为面向过程的接口组成部分。

过程式接口提供与*Screen*和*Turtle*类的方法相对应的函数。函数名与对应的方法名相同。当 Screen 类的方法对应函数被调用时会自动创建一个 Screen 对象。当 Turtle 类的方法对应函数被调用时会自动创建一个(匿名的) Turtle 对象。

如果屏幕上需要有多个海龟，就必须使用面向对象的接口。

注解： 以下文档给出了函数的参数列表。对于方法来说当然还有额外的第一个参数 *self*，这里省略了。

24.1.2 可用的 Turtle 和 Screen 方法概览

Turtle 方法

海龟动作

移动和绘制

forward() | *fd()* 前进
backward() | *bk()* | *back()* 后退
right() | *rt()* 右转
left() | *lt()* 左转
goto() | *setpos()* | *setposition()* 前往/定位
setx() 设置 x 坐标
sety() 设置 y 坐标
setheading() | *seth()* 设置朝向
home() 返回原点
circle() 画圆
dot() 画点
stamp() 印章
clearstamp() 清除印章
clearstamps() 清除多个印章
undo() 撤消
speed() 速度

获取海龟的状态

position() | *pos()* 位置
towards() 目标方向
xcor() x 坐标
ycor() y 坐标
heading() 朝向
distance() 距离

设置与度量单位

degrees() 角度
radians() 弧度

画笔控制

绘图状态

pendown() | *pd()* | *down()* 画笔落下
penup() | *pu()* | *up()* 画笔抬起

pensize() | *width()* 画笔粗细
pen() 画笔
isdown() 画笔是否落下

颜色控制

color() 颜色
pencolor() 画笔颜色
fillcolor() 填充颜色

填充

filling() 是否填充
begin_fill() 开始填充
end_fill() 结束填充

更多绘图控制

reset() 重置
clear() 清空
write() 书写

海龟状态

可见性

showturtle() | *st()* 显示海龟
hideturtle() | *ht()* 隐藏海龟
isvisible() 是否可见

外观

shape() 形状
resizemode() 大小调整模式
shapesize() | *turtlesize()* 形状大小
shearfactor() 剪切因子
settiltangle() 设置倾角
tiltangle() 倾角
tilt() 倾斜
shapetransform() 变形
get_shapepoly() 获取形状多边形

使用事件

onclick() 当鼠标点击
onrelease() 当鼠标释放
ondrag() 当鼠标拖动

特殊海龟方法

begin_poly() 开始记录多边形
end_poly() 结束记录多边形
get_poly() 获取多边形
clone() 克隆
getturtle() | *getpen()* 获取海龟画笔
getscreen() 获取屏幕
setundobuffer() 设置撤消缓冲区

undobufferentries() 撤消缓冲区条目数

TurtleScreen/Screen 方法

窗口控制

bgcolor() 背景颜色
bgpic() 背景图片
clear() | *clearscreen()* 清屏
reset() | *resetscreen()* 重置
screensize() 屏幕大小
setworldcoordinates() 设置世界坐标系

动画控制

delay() 延迟
tracer() 追踪
update() 更新

使用屏幕事件

listen() 监听
onkey() | *onkeyrelease()* 当键盘按下并释放
onkeypress() 当键盘按下
onclick() | *onscreenclick()* 当点击屏幕
ontimer() 当达到定时
mainloop() | *done()* 主循环

设置与特殊方法

mode()
colormode() 颜色模式
getcanvas() 获取画布
getshapes() 获取形状
register_shape() | *addshape()* 添加形状
turtles() 所有海龟
window_height() 窗口高度
window_width() 窗口宽度

输入方法

textinput() 文本输入
numinput() 数字输入

Screen 专有方法

bye() 退出
exitonclick() 当点击时退出
setup() 设置
title() 标题

24.1.3 RawTurtle/Turtle 方法和对应函数

本节中的大部分示例都使用 Turtle 类的一个实例，命名为 `turtle`。

海龟动作

`turtle.`**`forward`**(*distance*)
`turtle.`**`fd`**(*distance*)

参数 distance —一个数值 (整型或浮点型)

海龟前进 *distance* 指定的距离，方向为海龟的朝向。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

`turtle.`**`back`**(*distance*)
`turtle.`**`bk`**(*distance*)
`turtle.`**`backward`**(*distance*)

参数 distance —一个数值

海龟后退 *distance* 指定的距离，方向与海龟的朝向相反。不改变海龟的朝向。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

`turtle.`**`right`**(*angle*)
`turtle.`**`rt`**(*angle*)

参数 angle —一个数值 (整型或浮点型)

海龟右转 *angle* 个单位。(单位默认为角度，但可通过*degrees()* 和*radians()* 函数改变设置。) 角度的正负由海龟模式确定，参见*mode()*。

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

`turtle.`**`left`**(*angle*)
`turtle.`**`lt`**(*angle*)

参数 angle —一个数值 (整型或浮点型)

海龟左转 *angle* 个单位。(单位默认为角度，但可通过*degrees()* 和*radians()* 函数改变设置。) 角度的正负由海龟模式确定，参见*mode()*。

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.**goto**(*x*, *y=None*)
turtle.**setpos**(*x*, *y=None*)
turtle.**setposition**(*x*, *y=None*)

参数

- **x** – 一个数值或数值对/向量
- **y** – 一个数值或 None

如果 *y* 为 None，*x* 应为一个表示坐标的数值对或 *Vec2D* 类对象 (例如 *pos()* 返回的对象).

海龟移动到一个绝对坐标。如果画笔已落下将会画线。不改变海龟的朝向。

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.**setx**(*x*)

参数 x – 一个数值 (整型或浮点型)

设置海龟的横坐标为 *x*，纵坐标保持不变。

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

参数 y – 一个数值 (整型或浮点型)

设置海龟的纵坐标为 *y*，横坐标保持不变。

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.**setheading**(*to_angle*)
turtle.**seth**(*to_angle*)

参数 to_angle – 一个数值 (整型或浮点型)

设置海龟的朝向为 *to_angle*。以下是以角度表示的几个常用方向：

标准模式	logo 模式
0 - 东	0 - 北
90 - 北	90 - 东
180 - 西	180 - 南
270 - 南	270 - 西

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.**home**()

海龟移至初始坐标 (0,0)，并设置朝向为初始方向 (由海龟模式确定，参见*mode()*)。

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**circle**(*radius*, *extent=None*, *steps=None*)

参数

- **radius** —一个数值
- **extent** —一个数值 (或 None)
- **steps** —一个整型数 (或 None)

绘制一个 *radius* 指定半径的圆。圆心在海龟左边 *radius* 个单位；*extent* 为一个夹角，用来决定绘制圆的一部分。如未指定 *extent** 则绘制整个圆。如果 **extent* 不是完整圆周，则以当前画笔位置为一个端点绘制圆弧。如果 *radius* 为正值则朝逆时针方向绘制圆弧，否则朝顺时针方向。最终海龟的朝向会依据 *extent* 的值而改变。

圆实际是以其内切正多边形来近似表示的，其边的数量由 *steps* 指定。如果未指定边数则会自动确定。此方法也可用来绘制正多边形。

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.**dot**(*size=None*, **color*)

参数

- **size** —一个整型数 >= 1 (如果指定)
- **color** —一个颜色字符串或颜色数值元组

绘制一个直径为 *size*，颜色为 *color* 的圆点。如果 *size* 未指定，则直径取 pensize+4 和 2*pensize 中的较大值。

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.**stamp**()

在海龟当前位置印制一个海龟形状。返回该印章的 stamp_id，印章可以通过调用 clearstamp(stamp_id) 来删除。

```
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

turtle.**clearstamp**(*stampid*)

参数 stampid —一个整型数，必须是之前*stamp()* 调用的返回值

删除 *stampid* 指定的印章。

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle.**clearstamps**(*n=None*)

参数 n —一个整型数 (或 None)

删除全部或前/后 *n* 个海龟印章。如果 *n* 为 None 则删除全部印章，如果 $n > 0$ 则删除前 *n* 个印章，否则如果 $n < 0$ 则删除后 *n* 个印章。

```
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
15
16
17
18
```

(下页继续)

(续上页)

```
19
20
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.**undo**()

撤消 (或连续撤消) 最近的一个 (或多个) 海龟动作。可撤消的次数由撤消缓冲区的大小决定。

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.**speed**(*speed=None*)

参数 **speed** —一个 0..10 范围内的整型数或速度字符串 (见下)

设置海龟移动的速度为 0..10 表示的整型数值。如未指定参数则返回当前速度。

如果输入数值大于 10 或小于 0.5 则速度设为 0。速度字符串与速度值的对应关系如下:

- “fastest” : 0 最快
- “fast” : 10 快
- “normal” : 6 正常
- “slow” : 3 慢
- “slowest” : 1 最慢

速度值从 1 到 10，画线和海龟转向的动画效果逐级加快。

注意: *speed* = 0 表示 没有动画效果。forward/back 将使海龟向前/向后跳跃，同样的 left/right 将使海龟立即改变朝向。

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

获取海龟的状态

turtle.**position**()
turtle.**pos**()

返回海龟当前的坐标 (x,y) (为*Vec2D* 矢量类对象)。

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

参数

- **x** –一个数值或数值对/矢量，或一个海龟实例
- **y** –一个数值——如果 *x* 是一个数值，否则为 None

从海龟位置到由 (x,y)，矢量或另一海龟对应位置的连线的夹角。此数值依赖于海龟初始朝向 - 由“standard”/”world”或“logo”模式设置所决定)。

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()
返回海龟的 x 坐标。

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.**ycor**()
返回海龟的 y 坐标。

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.**heading**()
返回海龟当前的朝向 (数值依赖于海龟模式参见*mode()*)。

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*, *y=None*)

参数

- **x** –一个数值或数值对/矢量，或一个海龟实例
- **y** –一个数值——如果 *x* 是一个数值，否则为 None

返回从海龟位置到由 (x,y)，适量或另一海龟对应位置的单位距离。

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

度量单位设置

turtle.**degrees**(*fullcircle=360.0*)

> **参数 fullcircle** –一个数值

设置角度的度量单位，即设置一个圆周为多少“度”。默认值为 360 度。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.**radians**()

设置角度的度量单位为弧度。其值等于 `degrees(2*math.pi)`。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

画笔控制

绘图状态

turtle.**pendown**()
turtle.**pd**()
turtle.**down**()

画笔落下–移动时将画线。

turtle.**penup**()
turtle.**pu**()
turtle.**up**()

画笔抬起–移动时不画线。

turtle.**pensize**(*width=None*)
turtle.**width**(*width=None*)

> **参数 width** –一个正数值

设置线条的粗细为 *width* 或返回该值。如果 resizemode 设为“auto”并且 turtleshape 为多边形，该多边形也以同样组细的线条绘制。如未指定参数，则返回当前的 pensize。

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # from here on lines of width 10 are drawn
```

turtle.**pen**(*pen=None*, ***pendict*)

参数

- **pen** —一个包含部分或全部下列键的字典
- **pendict** —一个或多个以下列键为关键字的关键字参数

返回或设置画笔的属性，以一个包含以下键值对的“画笔字典”表示:

- “shown”：True/False
- “pendown”：True/False
- “pencolor”：颜色字符串或颜色元组
- “fillcolor”：颜色字符串或颜色元组
- “pensize”：正数值
- “speed”：0..10 范围内的数值
- “resizemode”：“auto”或“user”或“noresize”
- “stretchfactor”：(正数值, 正数值)
- “outline”：正数值
- “tilt”：数值

此字典可作为后续调用*pen()*时的参数，以恢复之前的画笔状态。另外还可将这些属性作为关键词参数提交。使用此方式可以用一条语句设置画笔的多个属性。

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

turtle.**isdown**()

如果画笔落下返回 `True`，如果画笔抬起返回 `False`。

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

颜色控制

turtle.**pencolor**(**args*)

返回或设置画笔颜色。

允许以下四种输入格式:

pencolor() 返回以颜色描述字符串或元组 (见示例) 表示的当前画笔颜色。可用作其他 color/pencolor/fillcolor 调用的输入。

pencolor(colorstring) 设置画笔颜色为 *colorstring* 指定的 Tk 颜色描述字符串，例如 `"red"`、`"yellow"` 或 `"#33cc8c"`。

pencolor((r, g, b)) 设置画笔颜色为以 *r*, *g*, *b* 元组表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode，colormode 的值为 1.0 或 255 (参见*colormode()*)。

pencolor(r, g, b)

设置画笔颜色为以 *r*, *g*, *b* 表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode。

如果 turtleshape 为多边形，该多边形轮廓也以新设置的画笔颜色绘制。

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.**fillcolor**(**args*)

返回或设置填充颜色。

允许以下四种输入格式:

fillcolor() 返回以颜色描述字符串或元组 (见示例) 表示的当前填充颜色。可用作其他 color/pencolor/fillcolor 调用的输入。

fillcolor(colorstring) 设置填充颜色为 *colorstring* 指定的 Tk 颜色描述字符串，例如 `"red"`、`"yellow"` 或 `"#33cc8c"`。

fillcolor((r, g, b)) 设置填充颜色为以 *r*, *g*, *b* 元组表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode，colormode 的值为 1.0 或 255 (参见*colormode()*)。

fillcolor(r, g, b)

设置填充颜色为 *r*, *g*, *b* 表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode。

如果 turtleshape 为多边形，该多边形内部也以新设置的填充颜色填充。

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> col = turtle.pencolor()
>>> col
(50.0, 193.0, 143.0)
>>> turtle.fillcolor(col)
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.**color**(**args*)
返回或设置画笔颜色和填充颜色。

允许多种输入格式。使用如下 0 至 3 个参数:

color() 返回以一对颜色描述字符串或元组表示的当前画笔颜色和填充颜色，两者可分别由*pencolor()* 和*fillcolor()* 返回。

color(colorstring),color((r,g,b)),color(r,g,b) 输入格式与*pencolor()* 相同，同时设置填充颜色和画笔颜色为指定的值。

color(colorstring1, colorstring2),color((r1,g1,b1), (r2,g2,b2))

相当于 pencolor(colorstring1) 加 fillcolor(colorstring2)，使用其他输入格式的方法也与之类似。

如果 turtleshape 为多边形，该多边形轮廓与填充也使用新设置的颜色。

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

另参见: Screen 方法*colormode()*。

填充

turtle.**filling**()
返回填充状态 (填充为 True，否则为 False)。

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

turtle.**begin_fill**()
在绘制要填充的形状之前调用。

turtle.**end_fill**()
填充上次调用*begin_fill()* 之后绘制的形状。

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

更多绘图控制

turtle.**reset**()
从屏幕中删除海龟的绘图，海龟回到原点并设置所有变量为默认值。

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**clear**()
从屏幕中删除指定海龟的绘图。不移动海龟。海龟的状态和位置以及其他海龟的绘图不受影响。

turtle.**write**(*arg*, *move=False*, *align="left"*, *font=("Arial", 8, "normal")*)

参数

- **arg** –要书写到 TurtleScreen 的对象
- **move** –True/False
- **align** –字符串“left”，“center”或“right”
- **font** –一个三元组 (fontname, fontsize, fonttype)

书写文本 - *arg* 指定的字符串 - 到当前海龟位置，*align* 指定对齐方式（“left”，“center”或 right”），font 指定字体。如果 *move* 为 True，画笔会移动到文本的右下角。默认 *move* 为 `False`。

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

海龟状态

可见性

turtle.**hideturtle**()
turtle.**ht**()
使海龟不可见。当你绘制复杂图形时这是个好主意，因为隐藏海龟可显著加快绘制速度。

```
>>> turtle.hideturtle()
```

turtle.**showturtle**()
turtle.**st**()
使海龟可见。

```
>>> turtle.showturtle()
```

turtle.**isvisible**()
如果海龟显示返回 True，如果海龟隐藏返回 False。

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

外观

turtle.**shape**(*name=None*)

参数 **name** –一个有效的形状名字符串

设置海龟形状为 *name* 指定的形状名，如未指定形状名则返回当前的形状名。*name* 指定的形状名应存在于 TurtleScreen 的 shape 字典中。多边形的形状初始时有以下几种: “arrow”，“turtle”，“circle”，“square”，“triangle”，“classic”。要了解如何处理形状请参看 Screen 方法*register_shape()*。

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode=None*)

参数 **rmode** –字符串“auto”，“user”，“noresize”其中之一

设置大小调整模式为以下值之一: “auto”，“user”，“noresize”。如未指定 *rmode* 则返回当前的大小调整模式。不同的大小调整模式的效果如下:

- “auto”：根据画笔粗细值调整海龟的外观。
- “user”：根据拉伸因子和轮廓宽度 (outline) 值调整海龟的外观，两者是由*shapesize()* 设置的。
- “noresize”：不调整海龟的外观大小。

大小调整模式 (“user”) 会在*shapesize()* 带参数调用时生效。

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)
turtle.**turtlesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

参数

- **stretch_wid** –正数值
- **stretch_len** –正数值
- **outline** –正数值

返回或设置画笔的属性 x/y-拉伸因子和/或轮廓。设置大小调整模式为 “user”。当且仅当大小调整模式设为 “user” 时海龟会基于其拉伸因子调整外观: *stretch_wid* 为垂直于其朝向的宽度拉伸因子，*stretch_len* 为平等于其朝向的长度拉伸因子，决定形状轮廓线的粗细。

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**shearfactor**(*shear=None*)

参数 shear – 数值 (可选)

设置或返回当前的剪切因子。根据 share 指定的剪切因子即剪切角度的切线来剪切海龟形状。不改变海龟的朝向 (移动方向)。如未指定 shear 参数: 返回当前的剪切因子即剪切角度的切线，与海龟朝向平行的线条将被剪切。

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.**tilt**(*angle*)

参数 angle – 一个数值

海龟形状自其当前的倾角转动 *angle* 指定的角度，但 不改变海龟的朝向 (移动方向)。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**settiltangle**(*angle*)

参数 angle – 一个数值

旋转海龟形状使其指向 *angle* 指定的方向，忽略其当前的倾角，不改变海龟的朝向 (移动方向)。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

3.1 版后已移除.

turtle.**tiltangle**(*angle=None*)

参数 angle – 一个数值 (可选)

设置或返回当前的倾角。如果指定 angle 则旋转海龟形状使其指向 angle 指定的方向，忽略其当前的倾角。不改变海龟的朝向 (移动方向)。如果未指定 angle: 返回当前的倾角，即海龟形状的方向和海龟朝向 (移动方向) 之间的夹角。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

turtle.**shapetransform**(*t11=None*, *t12=None*, *t21=None*, *t22=None*)

参数

- **t11** –一个数值 (可选)
- **t12** –一个数值 (可选)
- **t21** –一个数值 (可选)
- **t12** –一个数值 (可选)

设置或返回海龟形状的当前变形矩阵。

如不指定任何矩阵元素，则返回以 4 元素元组表示的变形矩阵。否则使用指定元素设置变形矩阵改变海龟形状，矩阵第一排的值为 t11, t12，第二排的值为 t21, t22。行列式 t11 * t22 - t12 * t21 的值不能为零，否则会出错。根据指定的矩阵修改拉伸因子，剪切因子和倾角。

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

turtle.**get_shapepoly**()

返回以坐标值对元组表示的当前形状多边形。这可以用于定义一个新形状或一个复合形状的多个组成部分。

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

使用事件

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

参数

- **fun** –一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** –鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** –`True` 或 `False` –如为 `True` 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到鼠标点击此海龟事件。如果 *fun* 值为 `None`，则移除现有的绑定。以下为使用匿名海龟即过程式的示例:

```
>>> def turn(x, y):
...     left(180)
...
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

turtle.**onrelease**(*fun*, *btn=1*, *add=None*)

参数

- **fun** –一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** –鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** –True 或 False –如为 True 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到在此海龟上释放鼠标按键事件。如果 *fun* 值为 None，则移除现有的绑定。

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)      # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transparent.
```

turtle.**ondrag**(*fun*, *btn=1*, *add=None*)

参数

- **fun** –一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** –鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** –True 或 False –如为 True 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到在此海龟上移动鼠标事件。如果 *fun* 值为 None，则移除现有的绑定。

注: 在海龟上移动鼠标事件之前应先发生在此海龟上点击鼠标事件。

```
>>> turtle.ondrag(turtle.goto)
```

在此之后点击并拖动海龟可在屏幕上手绘线条 (如果画笔为落下)。

特殊海龟方法

turtle.**begin_poly**()
开始记录多边形的顶点。当前海龟位置为多边形的第一个顶点。

turtle.**end_poly**()
停止记录多边形的顶点。当前海龟位置为多边形的最后一个顶点。它将连线到第一个顶点。

turtle.**get_poly**()
返回最新记录的多边形。

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
```

(下页继续)

(续上页)

```
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.**clone**()
创建并返回海龟的克隆体，具有相同的位置、朝向和海龟属性。

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.**getturtle**()
turtle.**getpen**()
返回海龟对象自身。唯一合理的用法: 作为一个函数来返回“匿名海龟”：

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen**()
返回作为海龟绘图场所的*TurtleScreen*类对象。该对象将可调用 TurtleScreen 方法。

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.**setundobuffer**(*size*)

参数 size —一个整型数值或 None

设置或禁用撤消缓冲区。如果 *size* 为一个整型数则将开辟一个指定大小的空缓冲区。*size* 表示可使用*undo()*方法/函数撤消的海龟命令的次数上限。如果 *size* 为 None 则禁用撤消缓冲区。

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**()
返回撤销缓冲区里的条目数。

```
>>> while undobufferentries():
...     undo()
```

复合形状

要使用由多个不同颜色多边形构成的复合海龟形状，你必须明确地使用辅助类*Shape*，具体步骤如下:

1. 创建一个空 Shape 对象，类型为 “compound”。
2. 按照需要使用 addcomponent() 方法向此对象添加多个部件。

   例如:

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. 接下来将 Shape 对象添加到 Screen 对象的形状列表并使用它:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

注解: *Shape* 类在*register_shape()* 方法的内部以多种方式使用。应用程序编写者 只有在使用上述的复合形状时才需要处理 Shape 类。

24.1.4 TurtleScreen/Screen 方法及对应函数

本节中的大部分示例都使用 TurtleScreen 类的一个实例，命名为 screen。

窗口控制

turtle.**bgcolor**(**args*)

> **参数 args** –一个颜色字符串或三个取值范围 0..colormode 内的数值或一个取值范围相同的数值 3 元组

设置或返回 TurtleScreen 的背景颜色。

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

turtle.**bgpic**(*picname=None*)

> **参数 picname** –一个字符串, gif-文件名, "nopic", 或 None

设置背景图片或返回当前背景图片名称。如果 *picname* 为一个文件名，则将相应图片设为背景。如果 *picname* 为 "nopic"，则删除当前背景图片。如果 *picname* 为 None，则返回当前背景图片文件名。:

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

turtle.**clear**()
turtle.**clearscreen**()
从中删除所有海龟的全部绘图。将已清空的 TurtleScreen 重置为初始状态: 白色背景，无背景片，无事件绑定并启用追踪。

注解: 此 TurtleScreen 方法作为全局函数时只有一个名字 `clearscreen`。全局函数 `clear` 所对应的是 Turtle 方法 `clear`。

---

turtle.**reset**()
turtle.**resetscreen**()
重置屏幕上的所有海龟为其初始状态。

注解: 此 TurtleScreen 方法作为全局函数时只有一个名字 `resetscreen`。全局函数 `reset` 所对应的是 Turtle 方法 `reset`。

---

turtle.**screensize**(*canvwidth=None*, *canvheight=None*, *bg=None*)

参数

- **canvwidth** –正整型数，以像素表示画布的新宽度值
- **canvheight** –正整型数，以像素表示画面的新高度值
- **bg** –颜色字符串或颜色元组，新的背景颜色

如未指定任何参数，则返回当前的 (canvaswidth, canvasheight)。否则改变作为海龟绘图场所的画布大小。不改变绘图窗口。要观察画布的隐藏区域，可以使用滚动条。通过此方法可以令之前绘制于画布之外的图形变为可见。

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

也可以用来寻找意外逃走的海龟;-)

turtle.**setworldcoordinates**(*llx*, *lly*, *urx*, *ury*)

参数

- **llx** –一个数值, 画布左下角的 x-坐标
- **lly** –一个数值, 画布左下角的 y-坐标
- **urx** –一个数值, 画面右上角的 x-坐标
- **ury** –一个数值, 画布右上角的 y-坐标

设置用户自定义坐标系并在必要时切换模式为“world”。这会执行一次 `screen.reset()`。如果“world”模式已激活，则所有图形将根据新的坐标系重绘。

注意: 在用户自定义坐标系中，角度可能显得扭曲。

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
```

(下页继续)

(续上页)

```
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

动画控制

turtle.**delay**(*delay=None*)

参数 **delay** –正整型数

设置或返回以毫秒数表示的延迟值 *delay*。(这约等于连续两次画布刷新的间隔时间。) 绘图延迟越长，动画速度越慢。

可选参数:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.**tracer**(*n=None*, *delay=None*)

参数

- **n** –非负整型数
- **delay** –非负整型数

启用/禁用海龟动画并设置刷新图形的延迟时间。如果指定 *n* 值，则只有每第 n 次屏幕刷新会实际执行。(可被用来加速复杂图形的绘制。) 如果调用时不带参数，则返回当前保存的 n 值。第二个参数设置延迟值 (参见*delay()*)。

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.**update**()
执行一次 TurtleScreen 刷新。在禁用追踪时使用。

另参见 RawTurtle/Turtle 方法*speed()*。

使用屏幕事件

turtle.**listen**(*xdummy=None*, *ydummy=None*)
设置焦点到 TurtleScreen (以便接收按键事件)。使用两个 Dummy 参数以便能够传递*listen()* 给 onclick 方法。

turtle.**onkey**(*fun*, *key*)
turtle.**onkeyrelease**(*fun*, *key*)

参数

- **fun** –一个无参数的函数或 None
- **key** –一个字符串: 键 (例如“a”) 或键标 (例如“space”)

绑定 *fun* 指定的函数到按键释放事件。如果 *fun* 值为 None，则移除事件绑定。注: 为了能够注册按键事件，TurtleScreen 必须得到焦点。(参见 method *listen()* 方法。)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onkeypress**(*fun*, *key=None*)

参数

- **fun** —一个无参数的函数或 None
- **key** —一个字符串: 键 (例如“a”) 或键标 (例如“space”)

绑定 *fun* 指定的函数到指定键的按下事件。如未指定键则绑定到任意键的按下事件。注: 为了能够注册按键事件，必须得到焦点。(参见 *listen()* 方法。)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onclick**(*fun*, *btn=1*, *add=None*)
turtle.**onscreenclick**(*fun*, *btn=1*, *add=None*)

参数

- **fun** —一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** –鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** –True 或 False –如为 True 则将添加一个新绑定，否则将取代先前的绑定

绑定 *fun* 指定的函数到鼠标点击屏幕事件。如果 *fun* 值为 None，则移除现有的绑定。

以下示例使用一个 TurtleScreen 实例 screen 和一个 Turtle 实例 turtle:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen will
>>>                             # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

注解: 此 TurtleScreen 方法作为全局函数时只有一个名字 onscreenclick。全局函数 onclick 所对应的是 Turtle 方法 onclick。

turtle.**ontimer**(*fun*, *t=0*)

参数

- **fun** —一个无参数的函数
- **t** —一个数值 >= 0

安装一个计时器，在 *t* 毫秒后调用 *fun* 函数。

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

turtle.**mainloop**()
turtle.**done**()

开始事件循环 - 调用 Tkinter 的 mainloop 函数。必须作为一个海龟绘图程序的结束语句。如果一个脚本是在以 -n 模式 (无子进程) 启动的 IDLE 中运行时 不可使用 - 用于实现海龟绘图的交互功能。:

```
>>> screen.mainloop()
```

输入方法

turtle.**textinput**(*title*, *prompt*)

参数

- **title** –string
- **prompt** –string

弹出一个对话框窗口用来输入一个字符串。形参 title 为对话框窗口的标题，prompt 为一条文本，通常用来提示要输入什么信息。返回输入的字符串。如果对话框被取消则返回 None。:

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

参数

- **title** –string
- **prompt** –string
- **default** –数值 (可选)
- **minval** –数值 (可选)
- **maxval** –数值 (可选)

弹出一个对话框窗口用来输入一个数值。title 为对话框窗口的标题，prompt 为一条文本，通常用来描述要输入的数值信息。default: 默认值, minval: 可输入的最小值, maxval: 可输入的最大值。输入数值的必须在指定的 minval .. maxval 范围之内，否则将给出一条提示，对话框保持打开等待修改。返回输入的数值。如果对话框被取消则返回 None。:

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

设置与特殊方法

turtle.**mode**(*mode=None*)

参数 mode –字符串“standard”，“logo”或“world”其中之一

设置海龟模式（“standard”，“logo”或“world”）并执行重置。如未指定模式则返回当前的模式。

“standard”模式与旧的*turtle*兼容。”logo”模式与大部分 Logo 海龟绘图兼容。”world”模式使用用户自定义的“世界坐标系”。**注意**: 在此模式下，如果 x/y 单位比率不等于 1 则角度会显得扭曲。

模式	初始海龟朝向	正数角度
“standard”	朝右 (东)	逆时针
“logo”	朝上 (北)	顺时针

```
>>> mode("logo")   # resets turtle heading to north
>>> mode()
'logo'
```

turtle.**colormode**(*cmode=None*)

参数 cmode –数值 1.0 或 255 其中之一

返回颜色模式或将其设为 1.0 或 255。构成颜色三元组的 *r*, *g*, *b* 数值必须在 0..*cmode* 范围之内。

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.**getcanvas**()

返回此 TurtleScreen 的 Canvas 对象。供了解 Tkinter 的 Canvas 对象内部机理的人士使用。

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.**getshapes**()

返回所有当前可用海龟形状的列表。

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.**register_shape**(*name*, *shape=None*)
turtle.**addshape**(*name*, *shape=None*)

调用此函数有三种不同方式:

(1) *name* 为一个 gif 文件的文件名，*shape* 为 None: 安装相应的图像形状。:

```
>>> screen.register_shape("turtle.gif")
```

注解: 当海龟转向时图像形状 不会转动，因此无法显示海龟的朝向!

(2) *name* 为指定的字符串，*shape* 为由坐标值对构成的元组: 安装相应的多边形形状。

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(3) *name* 为指定的字符串，为一个 (复合) *Shape* 类对象: 安装相应的复合形状。

将一个海龟形状加入 TurtleScreen 的形状列表。只有这样注册过的形状才能通过执行 `shape(shapename)` 命令来使用。

`turtle.`**`turtles`**`()`
返回屏幕上的海龟列表。

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.`**`window_height`**`()`
返回海龟窗口的高度。:

```
>>> screen.window_height()
480
```

`turtle.`**`window_width`**`()`
返回海龟窗口的宽度。:

```
>>> screen.window_width()
640
```

Screen 专有方法, 而非继承自 TurtleScreen

`turtle.`**`bye`**`()`
关闭海龟绘图窗口。

`turtle.`**`exitonclick`**`()`
将 bye() 方法绑定到 Screen 上的鼠标点击事件。

如果配置字典中 “using_IDLE” 的值为 `False` (默认值) 则同时进入主事件循环。注: 如果启动 IDLE 时使用了 `-n` 开关 (无子进程)，`turtle.cfg` 中此数值应设为 `True`。在此情况下 IDLE 本身的主事件循环同样会作用于客户脚本。

`turtle.`**`setup`**`(`*width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"]*`)`
设置主窗口的大小和位置。默认参数值保存在配置字典中，可通过 `turtle.cfg` 文件进行修改。

参数

- **`width`** –如为一个整型数值，表示大小为多少像素，如为一个浮点数值，则表示屏幕的占比；默认为屏幕的 50%
- **`height`** –如为一个整型数值，表示高度为多少像素，如为一个浮点数值，则表示屏幕的占比；默认为屏幕的 75%
- **`startx`** –如为正值，表示初始位置距离屏幕左边缘多少像素，负值表示距离右边缘，`None` 表示窗口水平居中
- **`starty`** –如为正值，表示初始位置距离屏幕上边缘多少像素，负值表示距离下边缘，`None` 表示窗口垂直居中

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>              # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>              # sets window to 75% of screen by 50% of screen and centers
```

turtle.**title**(*titlestring*)

参数 titlestring –一个字符串，显示为海龟绘图窗口的标题栏文本

设置海龟窗口标题为 *titlestring* 指定的文本。

```
>>> screen.title("Welcome to the turtle zoo!")
```

24.1.5 公共类

class turtle.**RawTurtle**(*canvas*)
class turtle.**RawPen**(*canvas*)

参数 canvas –一个 tkinter.Canvas，*ScrolledCanvas* 或 *TurtleScreen* 类对象

创建一个海龟。海龟对象具有“Turtle/RawTurtle 方法”一节所述的全部方法。

class turtle.**Turtle**
RawTurtle 的子类，具有相同的接口，但其绘图场所为默认的 *Screen* 类对象，在首次使用时自动创建。

class turtle.**TurtleScreen**(*cv*)

参数 cv –一个 tkinter.Canvas 类对象

提供面向屏幕的方法例如 setbg() 等。说明见上文。

class turtle.**Screen**
TurtleScreen 的子类，*增加了四个方法*.

class turtle.**ScrolledCanvas**(*master*)

参数 master –可容纳 ScrolledCanvas 的 Tkinter 部件，即添加了滚动条的 Tkinter-canvas

由 Screen 类使用，使其能够自动提供一个 ScrolledCanvas 作为海龟的绘图场所。

class turtle.**Shape**(*type_*, *data*)

参数 type_ –字符串“polygon”，“image”，“compound”其中之一

实现形状的数据结构。(type_, data) 必须遵循以下定义:

*type_*	*data*
“polygon”	一个多边形元组，即由坐标值对构成的元组
“image”	一个图片 (此形式仅限内部使用!)
“compound”	None (复合形状必须使用 *addcomponent()* 方法来构建)

addcomponent(*poly*, *fill*, *outline=None*)

参数

- **poly** –一个多边形，即由数值对构成的元组
- **fill** –一种颜色，将用来填充 *poly* 指定的多边形
- **outline** –一种颜色，用于多边形的轮廓 (如有指定)

示例:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

参见复合形状。

class turtle.**Vec2D**(*x*, *y*)

一个二维矢量类，用来作为实现海龟绘图的辅助类。也可能在海龟绘图程序中使用。派生自元组，因此矢量也属于元组!

提供的运算 (*a*, *b* 为矢量, *k* 为数值):

- `a + b` 矢量加法
- `a - b` 矢量减法
- `a * b` 内积
- `k * a` 和 `a * k` 与标量相乘
- `abs(a)` a 的绝对值
- `a.rotate(angle)` 旋转

24.1.6 帮助与配置

如何使用帮助

Screen 和 Turtle 类的公用方法以文档字符串提供了详细的文档。因此可以利用 Python 帮助工具获取这些在线帮助信息:

- 当使用 IDLE 时，输入函数/方法调用将弹出工具提示显示其签名和文档字符串的头几行。
- 对文法或函数调用*help()* 将显示其文档字符串:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


      >>> screen.bgcolor("orange")
      >>> screen.bgcolor()
      "orange"
      >>> screen.bgcolor(0.5,0,0.5)
      >>> screen.bgcolor()
      "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.
```

(下页继续)

(续上页)

```
    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- 方法对应函数的文档字符串的形式会有一些修改:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

这些修改版文档字符串是在导入时与方法对应函数的定义一起自动生成的。

文档字符串翻译为不同的语言

可使用工具创建一个字典，键为方法名，值为 Screen 和 Turtle 类公共方法的文档字符串。

turtle.**write_docstringdict**(*filename="turtle_docstringdict"*)

> **参数 filename** —一个字符串，表示文件名
>
> 创建文档字符串字典并将其写入 filename 指定的 Python 脚本文件。此函数必须显示地调用 (海龟绘图类并不使用此函数)。文档字符串字典将被写入到 Python 脚本文件 *filename*.py。该文件可作为模板用来将文档字符串翻译为不同语言。

如果你 (或你的学生) 想使用本国语言版本的*turtle* 在线帮助，你必须翻译文档字符串并保存结果文件，例如 turtle_docstringdict_german.py.

如果你在 turtle.cfg 文件中加入了相应的条目，此字典将在导入模块时被读取并替代原有的英文版文档字符串。

在撰写本文档时已经有了德语和意大利语版的文档字符串字典。(更多需求请联系 glingl@aon.at)

如何配置 Screen 和 Turtle

内置的默认配置是模仿旧 turtle 模块的外观和行为，以便尽可能地与其保持兼容。

如果你想使用不同的配置，以便更好地反映此模块的特性或是更适合你的需求，例如在课堂中使用，你可以准备一个配置文件 `turtle.cfg`，该文件将在导入模块时被读取并根据其中的设定修改模块配置。

内置的配置对应以下的 turtle.cfg:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

选定条目的简短说明:

- 开头的四行对应 `Screen.setup()` 方法的参数。
- 第 5 和 6 行对应 `Screen.screensize()` 方法的参数。
- *shape* 可以是任何内置形状，即: arrow, turtle 等。更多信息可用 `help(shape)` 查看。
- 如果你想使用无填充色 (即令海龟变透明)，你必须写 `fillcolor = ""` (但 cfg 文件中所有非空字符串都不可加引号)。
- 如果你想令海龟反映其状态，你必须使用 `resizemode = auto`。
- 如果你设置语言例如 `language = italian` 则文档字符串字典 `turtle_docstringdict_italian.py` 将在导入模块时被加载 (如果导入路径即*turtle* 的目录中存在此文件。
- *exampleturtle* 和 *examplescreen* 条目定义了相应对象在文档字符串中显示的名称。方法文档字符串转换为函数文档字符串时将从文档字符串中删去这些名称。
- *using_IDLE*: 如果你经常使用 IDLE 并启用其 -n 开关 (“无子进程”) 则应将此项设为 `True`，这将阻止*exitonclick()* 进入主事件循环。

`turtle.cfg` 文件可以保存于*turtle* 所在目录，当前工作目录也可以有一个同名文件。后者会重载覆盖前者的设置。

`Lib/turtledemo` 目录中也有一个 `turtle.cfg` 文件。你可以将其作为示例进行研究，并在运行演示时查看其作用效果 (但最好不要在演示查看器中运行)。

24.1.7 turtledemo —演示脚本集

turtledemo 包汇集了一组演示脚本。这些脚本可以通过以下命令打开所提供的演示查看器运行和查看:

```
python -m turtledemo
```

此外，你也可以单独运行其中的演示脚本。例如，:

```
python -m turtledemo.bytedesign
```

turtledemo 包目录中的内容:

- 一个演示查看器 __main__.py，可用来查看脚本的源码并即时运行。
- 多个脚本文件，演示*turtle* 模块的不同特性。所有示例可通过 Examples 菜单打开。也可以单独运行每个脚本。
- 一个 turtle.cfg 文件，作为说明如何编写并使用模块配置文件的示例模板。

演示脚本清单如下:

名称	描述	相关特性
bytedesign	复杂的传统海龟绘图模式	tracer(), delay, update()
chaos	绘制 Verhulst 动态模型，演示通过计算机的运算可能会生成令人惊叹的结果	世界坐标系
clock	绘制模拟时钟显示本机的当前时间	海龟作为表针, ontimer
colormixer	试验 r, g, b 颜色模式	ondrag() 当鼠标拖动
forest	绘制 3 棵广度优先树	随机化
fractalcurves	绘制 Hilbert & Koch 曲线	递归
lindenmayer	文化数学 (印度装饰艺术)	L-系统
minimal_hanoi	汉诺塔	矩形海龟作为汉诺盘 (shape, shapesize)
nim	玩经典的“尼姆”游戏，开始时有三堆小棒，与电脑对战。	海龟作为小棒，事件驱动 (鼠标, 键盘)
paint	超极简主义绘画程序	onclick() 当鼠标点击
peace	初级技巧	海龟: 外观与动画
penrose	非周期性地使用风筝和飞镖形状铺满平面	stamp() 印章
planet_and_moon	模拟引力系统	复合开关, Vec2D 类
round_dance	两两相对并不断旋转舞蹈的海龟	复合形状, clone shapesize, tilt, get_shapepoly, update
sorting_animate	动态演示不同的排序方法	简单对齐, 随机化
tree	一棵 (图形化的) 广度优先树 (使用生成器)	clone() 克隆
two_canvases	简单设计	两块画布上的海龟
wikipedia	一个来自介绍海龟绘图的维基百科文章的图案	clone(), undo()
yinyang	另一个初级示例	circle() 画圆

祝你玩得开心!

24.1.8 Python 2.6 之后的变化

- `Turtle.tracer()`, `Turtle.window_width()` 和 `Turtle.window_height()` 方法已被去除。具有这些名称和功能的方法现在只限于 `Screen` 类的方法。但其对应的函数仍然可用。(实际上在 Python 2.6 中这些方法就已经只是从对应的 `TurtleScreen`/`Screen` 类的方法复制而来。)
- `Turtle.fill()` 方法已被去除。`begin_fill()` 和 `end_fill()` 的行为则有细微改变: 现在每个填充过程必须以一个 `end_fill()` 调用来结束。
- 新增了一个 `Turtle.filling()` 方法。该方法返回一个布尔值: 如果填充过程正在进行为 `True`，否则为 `False`。此行为相当于 Python 2.6 中不带参数的 `fill()` 调用。

24.1.9 Python 3.0 之后的变化

- 新增了 `Turtle.shearfactor()`, `Turtle.shapetransform()` 和 `Turtle.get_shapepoly()` 方法。这样就可以使用所有标准线性变换来调整海龟形状。`Turtle.tiltangle()` 的功能已被加强: 现在可被用来获取或设置倾角。`Turtle.settiltangle()` 已弃用。
- 新增了 `Screen.onkeypress()` 方法作为对 `Screen.onkey()` 的补充，实际就是将行为绑定到 keyrelease 事件。后者相应增加了一个别名: `Screen.onkeyrelease()`。
- 新增了 `Screen.mainloop()` 方法。这样当仅需使用 Screen 和 Turtle 对象时不需要再额外导入 `mainloop()`。
- 新增了两个方法 `Screen.textinput()` 和 `Screen.numinput()`。用来弹出对话框接受输入并分别返回字符串和数值。
- 两个新的示例脚本 `tdemo_nim.py` 和 `tdemo_round_dance.py` 被加入到 `Lib/turtledemo` 目录中。

24.2 cmd —支持面向行的命令解释器

源代码: Lib/cmd.py

Cmd 类提供简单框架用于编写面向行的命令解释器。这些通常对测试工具，管理工具和原型有用，这些工具随后将被包含在更复杂的接口中。

class cmd.**Cmd**(*completekey='tab'*, *stdin=None*, *stdout=None*)

一个*Cmd* 实例或子类实例是面向行的解释器框架结构。实例化*Cmd* 本身是没有充分理由的，它作为自定义解释器类的超类是非常有用的为了继承*Cmd* 的方法并且封装动作方法。

可选参数 *completekey* 是完成键的*readline* 名称；默认是 `Tab` 。如果 *completekey* 不是*None* 并且*readline* 是可用的，命令完成会自动完成。

可选参数 *stdin* 和 *stdout* 指定了 Cmd 实例或子类实例将用于输入和输出的输入和输出文件对象。如果没有指定，他们将默认为*sys.stdin* 和*sys.stdout* 。

如果你想要使用一个给定的 *stdin* ，确保将实例的*use_rawinput* 属性设置为 `False` ，否则 *stdin* 将被忽略。

24.2.1 Cmd 对象

Cmd 实例有下列方法：

Cmd.**cmdloop**(*intro=None*)
反复发出提示，接受输入，从收到的输入中解析出一个初始前缀，并分派给操作方法，将其余的行作为参数传递给它们。

可选参数是在第一个提示之前发布的横幅或介绍字符串（这将覆盖*intro* 类属性）。

如果*readline* 继承模块被加载，输入将自动继承类似 **bash** 的历史列表编辑（例如，Control-P 滚动回到最后一个命令，Control-N 转到下一个命令，以 Control-F 非破坏性的方式向右 Control-B 移动光标，破坏性地等）。

输入的文件结束符被作为字符串传回 'EOF'。

解释器实例将会识别命令名称 foo 当且仅当它有方法 do_foo()。有一个特殊情况，分派始于字符 '?' 的行到方法 do_help()。另一种特殊情况，分派始于字符 '!' 的行到方法 do_shell()（如果定义了这个方法）

这个方法将返回当*postcmd()* 方法返回一个真值。参数 *stop* 到*postcmd()* 是命令对应的返回值 do_*() 的方法。

如果激活了完成，全部命令将会自动完成，并且通过调用 complete_foo() 参数 *text* , *line*, *begidx* , 和 *endidx* 完成全部命令参数。*text* 是我们试图匹配的字符串前缀，所有返回的匹配项必须以它为开头。*line* 是删除了前导空格的当前的输入行，*begidx* 和 *endidx* 是前缀文本的开始和结束索引。，可以用于根据参数位置提供不同的完成。

所有*Cmd* 的子类继承一个预定义 do_help()。这个方法使用参数 'bar' 调用，调用对应的方法 help_bar()，如果不存在，打印 do_bar() 的文档字符串，如果可用。没有参数的情况下，do_help() 方法会列出所有可用的帮助主题（即所有具有相应的 help_*() 方法或命令的文档字符串），也会列举所有未被记录的命令。

Cmd.**onecmd**(*str*)
解释该参数，就好像它是为响应提示而键入的一样。这可能会被覆盖，但通常不应该被覆盖; 请参阅: *precmd()* 和*postcmd()* 方法，用于执行有用的挂钩。返回值是一个标志，指示解释器对命令的解释是否应该停止。如果命令 *str* 有一个 do_*() 方法，则返回该方法的返回值，否则返回*default()* 方法的返回值。

Cmd.**emptyline**()
在响应提示输入空行时调用的方法。如果此方法未被覆盖，则重复输入的最后一个非空命令。

Cmd.**default**(*line*)
当命令前缀不能被识别的时候在输入行调用的方法。如果此方法未被覆盖，它将输出一个错误信息并返回。

Cmd.**completedefault**(*text*, *line*, *begidx*, *endidx*)
当没有特定于命令的 complete_*() 方法可用时，调用此方法完成输入行。默认情况下，它返回一个空列表。

Cmd.**precmd**(*line*)
钩方法在命令行 *line* 被解释之前执行，但是在输入提示被生成和发出后。这个方法是一个在*Cmd* 中的存根；它的存在是为了被子类覆盖。返回值被用作*onecmd()* 方法执行的命令；*precmd()* 的实现或许会重写命令或者简单的返回 *line* 不变。

Cmd.**postcmd**(*stop*, *line*)
钩方法只在命令调度完成后执行。这个方法是一个在*Cmd* 中的存根；它的存在是为了子类被覆盖。*line* 是被执行的命令行，*stop* 是一个表示在调用*postcmd()* 之后是否终止执行的标志；这将作为*onecmd()* 方法的返回值。这个方法的返回值被用作与 *stop* 相关联的内部标志的新值；返回 false 将导致解释继续。

Cmd.**preloop**()
钩方法当*cmdloop()* 被调用时执行一次。方法是一个在*Cmd* 中的存根；它的存在是为了被子类覆盖。

Cmd.**postloop**()
钩方法在*cmdloop()* 即将返回时执行一次。这个方法是一个在*Cmd* 中的存根；它的存在是为了被子类覆盖。

Instances of *Cmd* subclasses have some public instance variables:

Cmd.**prompt**
发出提示以请求输入。

Cmd.**identchars**
接受命令前缀的字符串。

Cmd.**lastcmd**
看到最后一个非空命令前缀。

Cmd.**cmdqueue**
排队的输入行列表。当需要新的输入时，在*cmdloop()* 中检查 cmdqueue 列表；如果它不是空的，它的元素将被按顺序处理，就像在提示符处输入一样。

Cmd.**intro**
要作为简介或横幅发出的字符串。可以通过给*cmdloop()* 方法一个参数来覆盖它。

Cmd.**doc_header**
如果帮助输出具有记录命令的段落，则发出头文件。

Cmd.**misc_header**
如果帮助输出其他帮助主题的部分（即与 `do_*()` 方法没有关联的 `help_*()` 方法），则发出头文件。

Cmd.**undoc_header**
如果帮助输出未被记录命令的部分（即与 `help_*()` 方法没有关联的 `do_*()` 方法），则发出头文件。

Cmd.**ruler**
用于在帮助信息标题的下方绘制分隔符的字符，如果为空，则不绘制标尺线。这个字符默认是 `'='` 。

Cmd.**use_rawinput**
这是一个标志，默认为 true 。如果为 true ，，*cmdloop()* 使用*input()* 先是提示并且阅读下一个命令；如果为 false ， `sys.stdout.write()` 和 `sys.stdin.readline()` 被使用。（这意味着解释器将会自动支持类似于 **Emacs** 的行编辑和命令历史记录按键操作，通过导入*readline* 在支持它的系统上。）

24.2.2 Cmd 例子

The *cmd* module is mainly useful for building custom shells that let a user work with a program interactively.

这部分提供了一个简单的例子来介绍如何使用一部分在*turtle* 模块中的命令构建一个 shell 。

基础的 turtle 命令比如*forward()* 被添加进一个*Cmd* 子类，方法名为 `do_forward()` 。参数被转换成数字并且分发至 turtle 模块中。docstring 是 shell 提供的帮助实用程序。

例子也包含使用*precmd()* 方法实现基础的记录和回放的功能，这个方法负责将输入转换为小写并且将命令写入文件。`do_playback()` 方法读取文件并添加记录命令至 `cmdqueue` 用于即时回放:

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell.   Type help or ? to list commands.\n'
    prompt = '(turtle) '
```

(下页继续)

(续上页)

```
    file = None

    # ----- basic turtle commands -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance:  FORWARD 10'
        forward(*parse(arg))
    def do_right(self, arg):
        'Turn turtle right by given number of degrees:  RIGHT 20'
        right(*parse(arg))
    def do_left(self, arg):
        'Turn turtle left by given number of degrees:  LEFT 90'
        left(*parse(arg))
    def do_goto(self, arg):
        'Move turtle to an absolute position with changing orientation.  GOTO 100 200'
        goto(*parse(arg))
    def do_home(self, arg):
        'Return turtle to the home position:  HOME'
        home()
    def do_circle(self, arg):
        'Draw circle with given radius an options extent and steps:  CIRCLE 50'
        circle(*parse(arg))
    def do_position(self, arg):
        'Print the current turtle position:  POSITION'
        print('Current position is %d %d\n' % position())
    def do_heading(self, arg):
        'Print the current turtle heading in degrees:  HEADING'
        print('Current heading is %d\n' % (heading(),))
    def do_color(self, arg):
        'Set the color:  COLOR BLUE'
        color(arg.lower())
    def do_undo(self, arg):
        'Undo (repeatedly) the last turtle action(s):  UNDO'
    def do_reset(self, arg):
        'Clear the screen and return turtle to center:  RESET'
        reset()
    def do_bye(self, arg):
        'Stop recording, close the turtle window, and exit:  BYE'
        print('Thank you for using Turtle')
        self.close()
        bye()
        return True

    # ----- record and playback -----
    def do_record(self, arg):
        'Save future commands to filename:  RECORD rose.cmd'
        self.file = open(arg, 'w')
    def do_playback(self, arg):
        'Playback commands from a file:  PLAYBACK rose.cmd'
        self.close()
        with open(arg) as f:
            self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
        return line
```

(下页继续)

(续上页)

```
    def close(self):
        if self.file:
            self.file.close()
            self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

这是一个示例会话，其中 turtle shell 显示帮助功能，使用空行重复命令，以及简单的记录和回放功能：

```
Welcome to the turtle shell.   Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
========================================
bye     color    goto     home  playback  record  right
circle  forward  heading  left  position  reset   undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
```

(下页继续)

(续上页)

```
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
```

24.3 shlex —Simple lexical analysis

Source code: Lib/shlex.py

The *shlex* class makes it easy to write lexical analyzers for simple syntaxes resembling that of the Unix shell. This will often be useful for writing minilanguages, (for example, in run control files for Python applications) or for parsing quoted strings.

The *shlex* module defines the following functions:

shlex.**split**(*s*, *comments=False*, *posix=True*)
Split the string *s* using shell-like syntax. If *comments* is *False* (the default), the parsing of comments in the given string will be disabled (setting the *commenters* attribute of the *shlex* instance to the empty string). This function operates in POSIX mode by default, but uses non-POSIX mode if the *posix* argument is false.

注解: Since the *split()* function instantiates a *shlex* instance, passing None for *s* will read the string to split from standard input.

---

shlex.**quote**(*s*)
Return a shell-escaped version of the string *s*. The returned value is a string that can safely be used as one token in a shell command line, for cases where you cannot use a list.

This idiom would be unsafe:

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

quote() lets you plug the security hole:

```
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
```

(下页继续)

(续上页)

```
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

The quoting is compatible with UNIX shells and with *split()*:

```
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

3.3 新版功能.

The *shlex* module defines the following class:

class shlex.**shlex**(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

A *shlex* instance or subclass instance is a lexical analyzer object. The initialization argument, if present, specifies where to read characters from. It must be a file-/stream-like object with *read()* and *readline()* methods, or a string. If no argument is given, input will be taken from `sys.stdin`. The second optional argument is a filename string, which sets the initial value of the *infile* attribute. If the *instream* argument is omitted or equal to `sys.stdin`, this second argument defaults to "stdin". The *posix* argument defines the operational mode: when *posix* is not true (default), the *shlex* instance will operate in compatibility mode. When operating in POSIX mode, *shlex* will try to be as close as possible to the POSIX shell parsing rules. The *punctuation_chars* argument provides a way to make the behaviour even closer to how real shells parse. This can take a number of values: the default value, `False`, preserves the behaviour seen under Python 3.5 and earlier. If set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters (considered punctuation characters) is returned as a single token. If set to a non-empty string of characters, those characters will be used as the punctuation characters. Any characters in the *wordchars* attribute that appear in *punctuation_chars* will be removed from *wordchars*. See *Improved Compatibility with Shells* for more information.

在 3.6 版更改: The *punctuation_chars* parameter was added.

参见:

Module ***configparser*** Parser for configuration files similar to the Windows `.ini` files.

24.3.1 shlex Objects

A *shlex* instance has the following methods:

shlex.**get_token**()

Return a token. If tokens have been stacked using *push_token()*, pop a token off the stack. Otherwise, read one from the input stream. If reading encounters an immediate end-of-file, *eof* is returned (the empty string (`''`) in non-POSIX mode, and `None` in POSIX mode).

shlex.**push_token**(*str*)

Push the argument onto the token stack.

shlex.**read_token**()

Read a raw token. Ignore the pushback stack, and do not interpret source requests. (This is not ordinarily a useful entry point, and is documented here only for the sake of completeness.)

shlex.**sourcehook**(*filename*)

When *shlex* detects a source request (see *source* below) this method is given the following token as argument, and expected to return a tuple consisting of a filename and an open file-like object.

Normally, this method first strips any quotes off the argument. If the result is an absolute pathname, or there was no previous source request in effect, or the previous source was a stream (such as `sys.stdin`), the result is left alone. Otherwise, if the result is a relative pathname, the directory part of the name of the file immediately before it on the source inclusion stack is prepended (this behavior is like the way the C preprocessor handles `#include "file.h"`).

The result of the manipulations is treated as a filename, and returned as the first component of the tuple, with *open()* called on it to yield the second component. (Note: this is the reverse of the order of arguments in instance initialization!)

This hook is exposed so that you can use it to implement directory search paths, addition of file extensions, and other namespace hacks. There is no corresponding 'close' hook, but a shlex instance will call the *close()* method of the sourced input stream when it returns EOF.

For more explicit control of source stacking, use the *push_source()* and *pop_source()* methods.

shlex.**push_source**(*newstream*, *newfile=None*)
Push an input source stream onto the input stack. If the filename argument is specified it will later be available for use in error messages. This is the same method used internally by the *sourcehook()* method.

shlex.**pop_source**()
Pop the last-pushed input source from the input stack. This is the same method used internally when the lexer reaches EOF on a stacked input stream.

shlex.**error_leader**(*infile=None*, *lineno=None*)
This method generates an error message leader in the format of a Unix C compiler error label; the format is `'"%s", line %d: '`, where the `%s` is replaced with the name of the current source file and the `%d` with the current input line number (the optional arguments can be used to override these).

This convenience is provided to encourage *shlex* users to generate error messages in the standard, parseable format understood by Emacs and other Unix tools.

Instances of *shlex* subclasses have some public instance variables which either control lexical analysis or can be used for debugging:

shlex.**commenters**
The string of characters that are recognized as comment beginners. All characters from the comment beginner to end of line are ignored. Includes just `'#'` by default.

shlex.**wordchars**
The string of characters that will accumulate into multi-character tokens. By default, includes all ASCII alphanumerics and underscore. In POSIX mode, the accented characters in the Latin-1 set are also included. If *punctuation_chars* is not empty, the characters `~-./*?=`, which can appear in filename specifications and command line parameters, will also be included in this attribute, and any characters which appear in `punctuation_chars` will be removed from `wordchars` if they are present there.

shlex.**whitespace**
Characters that will be considered whitespace and skipped. Whitespace bounds tokens. By default, includes space, tab, linefeed and carriage-return.

shlex.**escape**
Characters that will be considered as escape. This will be only used in POSIX mode, and includes just `'\'` by default.

shlex.**quotes**
Characters that will be considered string quotes. The token accumulates until the same quote is encountered again (thus, different quote types protect each other as in the shell.) By default, includes ASCII single and double quotes.

shlex.**escapedquotes**
Characters in *quotes* that will interpret escape characters defined in *escape*. This is only used in POSIX mode, and includes just `'"'` by default.

shlex.**whitespace_split**
If True, tokens will only be split in whitespaces. This is useful, for example, for parsing command lines with *shlex*, getting tokens in a similar way to shell arguments. If this attribute is True, *punctuation_chars* will have no effect, and splitting will happen only on whitespaces. When using *punctuation_chars*, which is intended to provide parsing closer to that implemented by shells, it is advisable to leave whitespace_split as False (the default value).

shlex.**infile**
The name of the current input file, as initially set at class instantiation time or stacked by later source requests. It may be useful to examine this when constructing error messages.

shlex.**instream**
The input stream from which this *shlex* instance is reading characters.

shlex.**source**
This attribute is None by default. If you assign a string to it, that string will be recognized as a lexical-level inclusion request similar to the source keyword in various shells. That is, the immediately following token will be opened as a filename and input will be taken from that stream until EOF, at which point the *close()* method of that stream will be called and the input source will again become the original input stream. Source requests may be stacked any number of levels deep.

shlex.**debug**
If this attribute is numeric and 1 or more, a *shlex* instance will print verbose progress output on its behavior. If you need to use this, you can read the module source code to learn the details.

shlex.**lineno**
Source line number (count of newlines seen so far plus one).

shlex.**token**
The token buffer. It may be useful to examine this when catching exceptions.

shlex.**eof**
Token used to determine end of file. This will be set to the empty string (''), in non-POSIX mode, and to None in POSIX mode.

shlex.**punctuation_chars**
Characters that will be considered punctuation. Runs of punctuation characters will be returned as a single token. However, note that no semantic validity checking will be performed: for example, '»>' could be returned as a token, even though it may not be recognised as such by shells.

3.6 新版功能.

24.3.2 Parsing Rules

When operating in non-POSIX mode, *shlex* will try to obey to the following rules.

- Quote characters are not recognized within words (Do"Not"Separate is parsed as the single word Do"Not"Separate);
- Escape characters are not recognized;
- Enclosing characters in quotes preserve the literal value of all characters within the quotes;
- Closing quotes separate words ("Do"Separate is parsed as "Do" and Separate);
- If *whitespace_split* is False, any character not declared to be a word character, whitespace, or a quote will be returned as a single-character token. If it is True, *shlex* will only split words in whitespaces;
- EOF is signaled with an empty string ('');
- It's not possible to parse empty strings, even if quoted.

When operating in POSIX mode, *shlex* will try to obey to the following parsing rules.

- Quotes are stripped out, and do not separate words (`"Do"Not"Separate"` is parsed as the single word `DoNotSeparate`);
- Non-quoted escape characters (e.g. `'\'`) preserve the literal value of the next character that follows;
- Enclosing characters in quotes which are not part of *escapedquotes* (e.g. `"'"`) preserve the literal value of all characters within the quotes;
- Enclosing characters in quotes which are part of *escapedquotes* (e.g. `'"'`) preserves the literal value of all characters within the quotes, with the exception of the characters mentioned in *escape*. The escape characters retain its special meaning only when followed by the quote in use, or the escape character itself. Otherwise the escape character will be considered a normal character.
- EOF is signaled with a *None* value;
- Quoted empty strings (`''`) are allowed.

24.3.3 Improved Compatibility with Shells

3.6 新版功能.

The *shlex* class provides compatibility with the parsing performed by common Unix shells like `bash`, `dash`, and `sh`. To take advantage of this compatibility, specify the `punctuation_chars` argument in the constructor. This defaults to `False`, which preserves pre-3.6 behaviour. However, if it is set to `True`, then parsing of the characters `();<>|&` is changed: any run of these characters is returned as a single token. While this is short of a full parser for shells (which would be out of scope for the standard library, given the multiplicity of shells out there), it does allow you to perform processing of command lines more easily than you could otherwise. To illustrate, you can see the difference in the following snippet:

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> list(shlex.shlex(text))
['a', '&', '&', 'b', ';', 'c', '&', '&', 'd', '|', '|', 'e', ';', 'f', '>',
"'abc'", ';', '(', 'def', '"ghi"', ')']
>>> list(shlex.shlex(text, punctuation_chars=True))
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', "'abc'",
';', '(', 'def', '"ghi"', ')']
```

Of course, tokens will be returned which are not valid for shells, and you'll need to implement your own error checks on the returned tokens.

Instead of passing `True` as the value for the punctuation_chars parameter, you can pass a string with specific characters, which will be used to determine which characters constitute punctuation. For example:

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

注解： When `punctuation_chars` is specified, the *wordchars* attribute is augmented with the characters `~-./*?=`. That is because these characters can appear in file names (including wildcards) and command-line arguments (e.g. `--color=auto`). Hence:

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
```

(下页继续)

(续上页)

```
...                  punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

For best effect, `punctuation_chars` should be set in conjunction with `posix=True`. (Note that `posix=False` is the default for *shlex*.)

CHAPTER 25

Tk 图形用户界面 (GUI)

Tcl/Tk 集成到 Python 中已经有一些年头了。Python 程序员可以通过*tkinter* 包和它的扩展，*tkinter.tix* 模块和*tkinter.ttk* 模块，来使用这套鲁棒的、平台无关的窗口工具集。

tkinter 包使用面向对象的方式对 Tcl/Tk 进行了一层薄包装。使用*tkinter*，你不需要写 Tcl 代码，但可能需要参考 Tk 文档，甚至 Tcl 文档。*tkinter* 使用 Python 类，对 Tk 的窗体小部件（Widgets）进行了一系列的封装。除此之外，内部模块 _tkinter 针对 Python 和 Tcl 之间的交互，提供了一套线程安全的机制。

tkinter 最大的优点就一个字：快，再一个，是 Python 自带的。尽管官方文档不太完整，但有其他资源可以参考，比如 Tk 手册，教程等。*tkinter* 也以比较过时的外观为人所知，但在 Tk 8.5 中，这一点得到了极大的改观。除此之外，如果有兴趣，还有其他的一些 GUI 库可供使用。更多信息，请参考*其他图形用户界面（GUI）包* 小节。

25.1 tkinter —Tcl/Tk 的 Python 接口

源代码: Lib/tkinter/__init__.py

The *tkinter* package (“Tk interface”) is the standard Python interface to the Tk GUI toolkit. Both Tk and *tkinter* are available on most Unix platforms, as well as on Windows systems. (Tk itself is not part of Python; it is maintained at ActiveState.)

在命令行中运行 `python -m tkinter`，应该会弹出一个 Tk 界面的窗口，表明*tkinter* 包已经正确安装，而且告诉你 Tcl/Tk 的版本号，通过这个版本号，你就可以参考对应的 Tcl/Tk 文档了。

参见:

Tkinter 文档:

Python Tkinter 资源 The Python Tkinter Topic Guide 提供了在 Python 中使用 Tk 的很多信息，同时包含了 Tk 其他信息的链接。

TKDocs 大量的教程，部分可视化组件的介绍说明。

Tkinter 8.5 reference: a GUI for Python 在线参考资料。

Tkinter docs from effbot effbot.org 提供的 tkinter 在线参考资料。

使用 Python 编程 由 Mark Lutz 所著的书籍，对 Tkinter 进行了完美的介绍。

为繁忙的 Python 开发者所准备的现代 Tkinter 由 Mark Rozerman 所著的关于如何使用 Python 和 Tkinter 来搭建有吸引力的和现代化的图形用户界面的书籍

Python 和 Tkinter 编程 作者：John Grayson (ISBN 1-884777-81-3).

Tcl/Tk 文档:

Tk 命令 多数命令以*tkinter*或者*tkinter.ttk*类的形式存在。改变‘8.6’以匹配所安装的 Tcl/Tk 版本。

Tcl/Tk 最新手册页面 www.tcl.tk 上面最新的 Tcl/Tk 手册。

ActiveState Tcl Home Page Tk/Tcl 的多数开发工作发生在 ActiveState 。

Tcl 及 Tk 工具集 由 Tcl 发明者 John Ousterhout 所著的书籍。

‘ Tcl 和 Tk 编程实战 <http://www.beedub.com/book/>‘_ Brent Welch 所著的百科全局式书籍。

25.1.1 Tkinter 模块

在大多数时候你只需要*tkinter*就足够了，但也有一些额外的模块可供使用。Tk 接口位于一个名字 `_tkinter` 的二进制模块当中。此模块包含了低层级的 Tk 接口，它不应该被应用程序员所直接使用。它通常是一个共享库（或 DLL），但在某些情况下也可能被静态链接到 Python 解释器。

除了 Tk 接口，*tkinter*也包含了若干 Python 模块，`tkinter.constants` 是其中最重要的。导入*tkinter*会自动导入 `tkinter.constants` ，所以，要使用 Tkinter 通常你只需要一条简单的 import 语句:

```
import tkinter
```

或者更常用的:

```
from tkinter import *
```

class `tkinter.`**Tk**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=1*)
*Tk*类被初始化时无参数。此时会创建一个 Tk 顶级控件，通常是应用程序的主窗口。每个实例都有自己关联的 Tcl 解释器。

`tkinter.`**Tcl**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=0*)
*Tcl()*函数是一个工厂函数，它创建的对象与*Tk*类创建的对象非常相似，只是它不初始化 Tk 子系统。在不想创建或无法创建（如没有 X Server 的 Unix/Linux 系统）额外的顶层窗口的环境中驱动 Tcl 解释器时，这一点非常有用。由*Tcl()*对象创建的对象可以通过调用其 `loadtk()` 方法来创建顶层窗口（并初始化 Tk 子系统）。

提供 Tk 支持的其他模块包括:

tkinter.scrolledtext Text widget with a vertical scroll bar built in.

tkinter.colorchooser 让用户选择颜色的对话框。

tkinter.commondialog 在此处列出的其他模块中定义的对话框的基类。

tkinter.filedialog Common dialogs to allow the user to specify a file to open or save.

tkinter.font Utilities to help work with fonts.

tkinter.messagebox Access to standard Tk dialog boxes.

tkinter.simpledialog Basic dialogs and convenience functions.

tkinter.dnd Drag-and-drop support for *tkinter*. This is experimental and should become deprecated when it is replaced with the Tk DND.

turtle Turtle graphics in a Tk window.

25.1.2 Tkinter Life Preserver

This section is not designed to be an exhaustive tutorial on either Tk or Tkinter. Rather, it is intended as a stop gap, providing some introductory orientation on the system.

Credits:

- Tk was written by John Ousterhout while at Berkeley.
- Tkinter was written by Steen Lumholt and Guido van Rossum.
- This Life Preserver was written by Matt Conway at the University of Virginia.
- The HTML rendering, and some liberal editing, was produced from a FrameMaker version by Ken Manheimer.
- Fredrik Lundh elaborated and revised the class interface descriptions, to get them current with Tk 4.2.
- Mike Clarkson converted the documentation to LaTeX, and compiled the User Interface chapter of the reference manual.

How To Use This Section

This section is designed in two parts: the first half (roughly) covers background material, while the second half can be taken to the keyboard as a handy reference.

When trying to answer questions of the form "how do I do blah", it is often best to find out how to do "blah" in straight Tk, and then convert this back into the corresponding *tkinter* call. Python programmers can often guess at the correct Python command by looking at the Tk documentation. This means that in order to use Tkinter, you will have to know a little bit about Tk. This document can't fulfill that role, so the best we can do is point you to the best documentation that exists. Here are some hints:

- The authors strongly suggest getting a copy of the Tk man pages. Specifically, the man pages in the `manN` directory are most useful. The `man3` man pages describe the C interface to the Tk library and thus are not especially helpful for script writers.
- Addison-Wesley publishes a book called Tcl and the Tk Toolkit by John Ousterhout (ISBN 0-201-63337-X) which is a good introduction to Tcl and Tk for the novice. The book is not exhaustive, and for many details it defers to the man pages.
- `tkinter/__init__.py` is a last resort for most, but can be a good place to go when nothing else makes sense.

A Simple Hello World Program

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()
```

(下页继续)

(续上页)

```
    def create_widgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there["text"] = "Hello World\n(click me)"
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack(side="top")

        self.quit = tk.Button(self, text="QUIT", fg="red",
                              command=self.master.destroy)
        self.quit.pack(side="bottom")

    def say_hi(self):
        print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()
```

25.1.3 A (Very) Quick Look at Tcl/Tk

The class hierarchy looks complicated, but in actual practice, application programmers almost always refer to the classes at the very bottom of the hierarchy.

注释:

- These classes are provided for the purposes of organizing certain functions under one namespace. They aren't meant to be instantiated independently.
- The *Tk* class is meant to be instantiated only once in an application. Application programmers need not instantiate one explicitly, the system creates one whenever any of the other classes are instantiated.
- The `Widget` class is not meant to be instantiated, it is meant only for subclassing to make "real" widgets (in C++, this is called an 'abstract class').

To make use of this reference material, there will be times when you will need to know how to read short passages of Tk and how to identify the various parts of a Tk command. (See section *Mapping Basic Tk into Tkinter* for the *tkinter* equivalents of what's below.)

Tk scripts are Tcl programs. Like all Tcl programs, Tk scripts are just lists of tokens separated by spaces. A Tk widget is just its *class*, the *options* that help configure it, and the *actions* that make it do useful things.

To make a widget in Tk, the command is always of the form:

```
classCommand newPathname options
```

classCommand denotes which kind of widget to make (a button, a label, a menu…)

newPathname is the new name for this widget. All names in Tk must be unique. To help enforce this, widgets in Tk are named with *pathnames*, just like files in a file system. The top level widget, the *root*, is called `.` (period) and children are delimited by more periods. For example, `.myApp.controlPanel.okButton` might be the name of a widget.

options configure the widget's appearance and in some cases, its behavior. The options come in the form of a list of flags and values. Flags are preceded by a '-', like Unix shell command flags, and values are put in quotes if they are more than one word.

例如

```
button   .fred   -fg red -text "hi there"
   ^       ^     \_____________________/
   |       |                |
 class    new            options
command  widget  (-opt val -opt val ...)
```

Once created, the pathname to the widget becomes a new command. This new *widget command* is the programmer's handle for getting the new widget to perform some *action*. In C, you'd express this as someAction(fred, someOptions), in C++, you would express this as fred.someAction(someOptions), and in Tk, you say:

```
.fred someAction someOptions
```

Note that the object name, `.fred`, starts with a dot.

As you'd expect, the legal values for *someAction* will depend on the widget's class: `.fred disable` works if fred is a button (fred gets greyed out), but does not work if fred is a label (disabling of labels is not supported in Tk).

The legal values of *someOptions* is action dependent. Some actions, like `disable`, require no arguments, others, like a text-entry box's `delete` command, would need arguments to specify what range of text to delete.

25.1.4 Mapping Basic Tk into Tkinter

Class commands in Tk correspond to class constructors in Tkinter.

```
button .fred                =====>  fred = Button()
```

The master of an object is implicit in the new name given to it at creation time. In Tkinter, masters are specified explicitly.

```
button .panel.fred          =====>  fred = Button(panel)
```

The configuration options in Tk are given in lists of hyphened tags followed by values. In Tkinter, options are specified as keyword-arguments in the instance constructor, and keyword-args for configure calls or as instance indices, in dictionary style, for established instances. See section *Setting Options* on setting options.

```
button .fred -fg red        =====>  fred = Button(panel, fg="red")
.fred configure -fg red     =====>  fred["fg"] = red
                            OR ==>  fred.config(fg="red")
```

In Tk, to perform an action on a widget, use the widget name as a command, and follow it with an action name, possibly with arguments (options). In Tkinter, you call methods on the class instance to invoke actions on the widget. The actions (methods) that a given widget can perform are listed in `tkinter/__init__.py`.

```
.fred invoke                =====>  fred.invoke()
```

To give a widget to the packer (geometry manager), you call pack with optional arguments. In Tkinter, the Pack class holds all this functionality, and the various forms of the pack command are implemented as methods. All widgets in *tkinter* are subclassed from the Packer, and so inherit all the packing methods. See the *tkinter.tix* module documentation for additional information on the Form geometry manager.

```
pack .fred -side left       =====>  fred.pack(side="left")
```

25.1.5 How Tk and Tkinter are Related

From the top down:

Your App Here (Python) A Python application makes a *tkinter* call.

tkinter (Python Package) This call (say, for example, creating a button widget), is implemented in the *tkinter* package, which is written in Python. This Python function will parse the commands and the arguments and convert them into a form that makes them look as if they had come from a Tk script instead of a Python script.

_tkinter (C) These commands and their arguments will be passed to a C function in the `_tkinter` - note the underscore - extension module.

Tk Widgets (C and Tcl) This C function is able to make calls into other C modules, including the C functions that make up the Tk library. Tk is implemented in C and some Tcl. The Tcl part of the Tk widgets is used to bind certain default behaviors to widgets, and is executed once at the point where the Python *tkinter* package is imported. (The user never sees this stage).

Tk (C) The Tk part of the Tk Widgets implement the final mapping to ···

Xlib (C) the Xlib library to draw graphics on the screen.

25.1.6 Handy Reference

Setting Options

Options control things like the color and border width of a widget. Options can be set in three ways:

At object creation time, using keyword arguments

```
fred = Button(self, fg="red", bg="blue")
```

After object creation, treating the option name like a dictionary index

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

Use the config() method to update multiple attrs subsequent to object creation

```
fred.config(fg="red", bg="blue")
```

For a complete explanation of a given option and its behavior, see the Tk man pages for the widget in question.

Note that the man pages list "STANDARD OPTIONS" and "WIDGET SPECIFIC OPTIONS" for each widget. The former is a list of options that are common to many widgets, the latter are the options that are idiosyncratic to that particular widget. The Standard Options are documented on the *options(3)* man page.

No distinction between standard and widget-specific options is made in this document. Some options don't apply to some kinds of widgets. Whether a given widget responds to a particular option depends on the class of the widget; buttons have a `command` option, labels do not.

The options supported by a given widget are listed in that widget's man page, or can be queried at runtime by calling the `config()` method without arguments, or by calling the `keys()` method on that widget. The return value of these calls is a dictionary whose key is the name of the option as a string (for example, `'relief'`) and whose values are 5-tuples.

Some options, like `bg` are synonyms for common options with long names (`bg` is shorthand for "background"). Passing the `config()` method the name of a shorthand option will return a 2-tuple, not 5-tuple. The 2-tuple passed back will contain the name of the synonym and the "real" option (such as `('bg', 'background')`).

索引	含义	示例
0	选项名称	`'relief'`
1	数据库查找的选项名称	`'relief'`
2	数据库查找的选项类	`'Relief'`
3	默认值	`'raised'`
4	当前值	`'groove'`

示例:

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

Of course, the dictionary printed will include all the options available and their values. This is meant only as an example.

The Packer

The packer is one of Tk's geometry-management mechanisms. Geometry managers are used to specify the relative positioning of the positioning of widgets within their container - their mutual *master*. In contrast to the more cumbersome *placer* (which is used less commonly, and we do not cover here), the packer takes qualitative relationship specification - *above*, *to the left of*, *filling*, etc - and works everything out to determine the exact placement coordinates for you.

The size of any *master* widget is determined by the size of the "slave widgets" inside. The packer is used to control where slave widgets appear inside the master into which they are packed. You can pack widgets into frames, and frames into other frames, in order to achieve the kind of layout you desire. Additionally, the arrangement is dynamically adjusted to accommodate incremental changes to the configuration, once it is packed.

Note that widgets do not appear until they have had their geometry specified with a geometry manager. It's a common early mistake to leave out the geometry specification, and then be surprised when the widget is created but nothing appears. A widget will appear only after it has had, for example, the packer's `pack()` method applied to it.

The pack() method can be called with keyword-option/value pairs that control where the widget is to appear within its container, and how it is to behave when the main application window is resized. Here are some examples:

```
fred.pack()                     # defaults to side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

Packer Options

For more extensive information on the packer and the options that it can take, see the man pages and page 183 of John Ousterhout's book.

anchor Anchor type. Denotes where the packer is to place each slave in its parcel.

expand Boolean, `0` or `1`.

fill Legal values: `'x'`, `'y'`, `'both'`, `'none'`.

ipadx 和 ipady A distance - designating internal padding on each side of the slave widget.

padx 和 pady A distance - designating external padding on each side of the slave widget.

side Legal values are: `'left'`, `'right'`, `'top'`, `'bottom'`.

Coupling Widget Variables

The current-value setting of some widgets (like text entry widgets) can be connected directly to application variables by using special options. These options are `variable`, `textvariable`, `onvalue`, `offvalue`, and `value`. This connection works both ways: if the variable changes for any reason, the widget it's connected to will be updated to reflect the new value.

Unfortunately, in the current implementation of *tkinter* it is not possible to hand over an arbitrary Python variable to a widget through a `variable` or `textvariable` option. The only kinds of variables for which this works are variables that are subclassed from a class called Variable, defined in *tkinter*.

There are many useful subclasses of Variable already defined: `StringVar`, `IntVar`, `DoubleVar`, and `BooleanVar`. To read the current value of such a variable, call the `get()` method on it, and to change its value you call the `set()` method. If you follow this protocol, the widget will always track the value of the variable, with no further intervention on your part.

例如

```
class App(Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

        self.entrythingy = Entry()
        self.entrythingy.pack()

        # here is the application variable
        self.contents = StringVar()
        # set it to some value
        self.contents.set("this is a variable")
        # tell the entry widget to watch this variable
        self.entrythingy["textvariable"] = self.contents

        # and here we get a callback when the user hits return.
        # we will have the program print out the value of the
        # application variable when the user hits return
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("hi. contents of entry is now ---->",
              self.contents.get())
```

The Window Manager

In Tk, there is a utility command, `wm`, for interacting with the window manager. Options to the `wm` command allow you to control things like titles, placement, icon bitmaps, and the like. In *tkinter*, these commands have been implemented as methods on the `Wm` class. Toplevel widgets are subclassed from the `Wm` class, and so can call the `Wm` methods directly.

To get at the toplevel window that contains a given widget, you can often just refer to the widget's master. Of course if the widget has been packed inside of a frame, the master won't represent a toplevel window. To get at the toplevel window that contains an arbitrary widget, you can call the `_root()` method. This method begins with an underscore to denote the fact that this function is part of the implementation, and not an interface to Tk functionality.

Here are some examples of typical usage:

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Tk Option Data Types

anchor Legal values are points of the compass: `"n"`, `"ne"`, `"e"`, `"se"`, `"s"`, `"sw"`, `"w"`, `"nw"`, and also `"center"`.

位图 There are eight built-in, named bitmaps: `'error'`, `'gray25'`, `'gray50'`, `'hourglass'`, `'info'`, `'questhead'`, `'question'`, `'warning'`. To specify an X bitmap filename, give the full path to the file, preceded with an `@`, as in `"@/usr/contrib/bitmap/gumby.bit"`.

boolean You can pass integers 0 or 1 or the strings `"yes"` or `"no"`.

callback –回调 This is any Python function that takes no arguments. For example:

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

color Colors can be given as the names of X colors in the rgb.txt file, or as strings representing RGB values in 4 bit: `"#RGB"`, 8 bit: `"#RRGGBB"`, 12 bit" `"#RRRGGGBBB"`, or 16 bit `"#RRRRGGGGBBBB"` ranges, where R,G,B here represent any legal hex digit. See page 160 of Ousterhout's book for details.

cursor The standard X cursor names from `cursorfont.h` can be used, without the `XC_` prefix. For example to get a hand cursor (`XC_hand2`), use the string `"hand2"`. You can also specify a bitmap and mask file of your own. See page 179 of Ousterhout's book.

distance Screen distances can be specified in either pixels or absolute distances. Pixels are given as numbers and absolute distances as strings, with the trailing character denoting units: `c` for centimetres, `i` for inches, `m` for millimetres, `p` for printer's points. For example, 3.5 inches is expressed as `"3.5i"`.

font Tk uses a list font name format, such as `{courier 10 bold}`. Font sizes with positive numbers are measured in points; sizes with negative numbers are measured in pixels.

geometry This is a string of the form `widthxheight`, where width and height are measured in pixels for most widgets (in characters for widgets displaying text). For example: `fred["geometry"] = "200x100"`.

justify Legal values are the strings: `"left"`, `"center"`, `"right"`, and `"fill"`.

region This is a string with four space-delimited elements, each of which is a legal distance (see above). For example: `"2 3 4 5"` and `"3i 2i 4.5i 2i"` and `"3c 2c 4c 10.43c"` are all legal regions.

relief Determines what the border style of a widget will be. Legal values are: `"raised"`, `"sunken"`, `"flat"`, `"groove"`, and `"ridge"`.

scrollcommand This is almost always the `set()` method of some scrollbar widget, but can be any widget method that takes a single argument.

wrap: Must be one of: `"none"`, `"char"`, or `"word"`.

Bindings and Events

The bind method from the widget command allows you to watch for certain events and to have a callback function trigger when that event type occurs. The form of the bind method is:

```
def bind(self, sequence, func, add=''):
```

where:

sequence –序列 is a string that denotes the target kind of event. (See the bind man page and page 201 of John Ousterhout's book for details).

func is a Python function, taking one argument, to be invoked when the event occurs. An Event instance will be passed as the argument. (Functions deployed this way are commonly known as *callbacks*.)

add is optional, either `''` or `'+'`. Passing an empty string denotes that this binding is to replace any other bindings that this event is associated with. Passing a `'+'` means that this function is to be added to the list of functions bound to this event type.

例如

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

Notice how the widget field of the event is being accessed in the `turn_red()` callback. This field contains the widget that caught the X event. The following table lists the other event fields you can access, and how they are denoted in Tk, which can be useful when referring to the Tk man pages.

Tk	Tkinter Event Field	Tk	Tkinter Event Field
%f	焦点	%A	char
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	状况	%N	keysym_num
%t	time	%T	type –类型
%w	宽度	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

The index Parameter

A number of widgets require "index" parameters to be passed. These are used to point at a specific place in a Text widget, or to particular characters in an Entry widget, or to particular menu items in a Menu widget.

Entry widget indexes (index, view index, etc.) Entry widgets have options that refer to character positions in the text being displayed. You can use these *tkinter* functions to access these special points in text widgets:

Text widget indexes The index notation for Text widgets is very rich and is best described in the Tk man pages.

Menu indexes (menu.invoke(), menu.entryconfig(), etc.) Some options and methods for menus manipulate specific menu entries. Anytime a menu index is needed for an option or a parameter, you may pass in:

- an integer which refers to the numeric position of the entry in the widget, counted from the top, starting with 0;
- the string `"active"`, which refers to the menu position that is currently under the cursor;
- the string `"last"` which refers to the last menu item;
- An integer preceded by `@`, as in `@6`, where the integer is interpreted as a y pixel coordinate in the menu's coordinate system;
- the string `"none"`, which indicates no menu entry at all, most often used with menu.activate() to deactivate all entries, and finally,
- a text string that is pattern matched against the label of the menu entry, as scanned from the top of the menu to the bottom. Note that this index type is considered after all the others, which means that matches for menu items labelled `last`, `active`, or `none` may be interpreted as the above literals, instead.

Images

Images of different formats can be created through the corresponding subclass of `tkinter.Image`:

- `BitmapImage` for images in XBM format.
- `PhotoImage` for images in PGM, PPM, GIF and PNG formats. The latter is supported starting with Tk 8.6.

Either type of image is created through either the `file` or the `data` option (other options are available as well).

The image object can then be used wherever an `image` option is supported by some widget (e.g. labels, buttons, menus). In these cases, Tk will not keep a reference to the image. When the last Python reference to the image object is deleted, the image data is deleted as well, and Tk will display an empty box wherever the image was used.

参见:

The Pillow package adds support for formats such as BMP, JPEG, TIFF, and WebP, among others.

25.1.7 File Handlers

Tk allows you to register and unregister a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. Only one handler may be registered per file descriptor. Example code:

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

This feature is not available on Windows.

Since you don't know how many bytes are available for reading, you may not want to use the *BufferedIOBase* or *TextIOBase* *read()* or *readline()* methods, since these will insist on reading a predefined number of bytes. For sockets, the *recv()* or *recvfrom()* methods will work fine; for other files, use raw reads or `os.read(file.fileno(), maxbytecount)`.

`Widget.tk.`**`createfilehandler`**(*file*, *mask*, *func*)

Registers the file handler callback function *func*. The *file* argument may either be an object with a *fileno()* method (such as a file or socket object), or an integer file descriptor. The *mask* argument is an ORed combination of any of the three constants below. The callback is called as follows:

```
callback(file, mask)
```

`Widget.tk.`**`deletefilehandler`**(*file*)

Unregisters a file handler.

`tkinter.`**`READABLE`**
`tkinter.`**`WRITABLE`**
`tkinter.`**`EXCEPTION`**

Constants used in the *mask* arguments.

25.2 `tkinter.ttk` —Tk 主题小部件

源代码: Lib/tkinter/ttk.py

The *tkinter.ttk* module provides access to the Tk themed widget set, introduced in Tk 8.5. If Python has not been compiled against Tk 8.5, this module can still be accessed if *Tile* has been installed. The former method using Tk 8.5 provides additional benefits including anti-aliased font rendering under X11 and window transparency (requiring a composition window manager on X11).

The basic idea for *tkinter.ttk* is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

参见:

Tk Widget Styling Support 一份文档介绍 Tk 支持的主题

25.2.1 使用 Ttk

开始使用 Ttk, 导入模块:

```
from tkinter import ttk
```

重写基础 Tk 部件, 导入应跟随 Tk 导入:

```
from tkinter import *
from tkinter.ttk import *
```

That code causes several *tkinter.ttk* widgets (`Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` and `Scrollbar`) to automatically replace the Tk widgets.

This has the direct benefit of using the new widgets which gives a better look and feel across platforms; however, the replacement widgets are not completely compatible. The main difference is that widget options such as "fg", "bg" and

others related to widget styling are no longer present in Ttk widgets. Instead, use the `ttk.Style` class for improved styling effects.

参见:

Converting existing applications to use Tile widgets A monograph (using Tcl terminology) about differences typically encountered when moving applications to use the new widgets.

25.2.2 Ttk 部件

Ttk comes with 17 widgets, eleven of which already existed in tkinter: `Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` and `Scrollbar`. The other six are new: *`Combobox`*, *`Notebook`*, *`Progressbar`*, `Separator`, `Sizegrip` and *`Treeview`*. And all them are subclasses of *`Widget`*.

Using the Ttk widgets gives the application an improved look and feel. As discussed above, there are differences in how the styling is coded.

Tk 代码:

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk 代码:

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

有关*TtkStyling* 的更多信息，请参阅*`Style`* 类文档。

25.2.3 控件

`ttk.Widget` 定义了 Tk 主题小部件支持的标准选项和方法，不应该直接实例化。

标准选项

所有 `ttk` 小部件接受以下选项：

选项	描述
class –类	指定窗口类。在查询选项数据库中窗口的其他选项时，使用该类，确定窗口的默认绑定标签，以及选择窗口小部件的默认布局和样式。此选项仅为只读，并且只能在创建窗口时指定。
cursor	指定要用于窗口小部件的鼠标光标。如果设置为空字符串（默认值），则为父窗口小部件继承光标.
takefocus	确定窗口是否在键盘遍历期间接受焦点。返回 0 或 1，返回空字符串。如果返回 0，则表示在键盘遍历期间应该跳过该窗口。如果为 1，则表示只要可以查看窗口就应该接收输入焦点。并且空字符串意味着遍历脚本决定是否关注窗口。
style	可用于指定自定义窗口小部件样式。

可滚动控件选项

控件支持以下选项使用滚动条控制.

选项	描述
xscroll 命令	用于与垂直滚动条通讯. 当视图在部件窗口改变, 部件将会基于 scroll 命令生成 Tcl 命令. Usually this option consists of the method `Scrollbar.set()` of some scrollbar. This will cause the scrollbar to be updated whenever the view in the window changes.
yscroll 命令	用于与垂直滚动条通讯. 更多信息请参考上面的信息.

标签选项

以下选项支持标签, 按钮已经其他类按钮的控件.

选项	描述
文本	Specifies a text string to be displayed inside the widget.
文本变量	Specifies a name whose value will be used in place of the text option resource.
下划线	If set, specifies the index (0-based) of a character to underline in the text string. The underline character is used for mnemonic activation.
图片	Specifies an image to display. This is a list of 1 or more elements. The first element is the default image name. The rest of the list if a sequence of statespec/value pairs as defined by *`Style.map()`*, specifying different images to use when the widget is in a particular state or a combination of states. All images in the list should have the same size.
compound	Specifies how to display the image relative to the text, in the case both text and images options are present. Valid values are: • text: 只显示文本 • image: 只显示图片 • top, bottom, left, right: 分别显示图片的上, 下, 左, 右的文本. • none: 默认. 如果设置显示图片, 否则文本.
宽度	If greater than zero, specifies how much space, in character widths, to allocate for the text label, if less than zero, specifies a minimum width. If zero or unspecified, the natural width of the text label is used.

兼容性选项

选项	描述
状况	May be set to “normal” or “disabled” to control the “disabled” state bit. This is a write-only option: setting it changes the widget state, but the *`Widget.state()`* method does not affect this option.

控件状态

控件状态是无关状态标志的位图.

标志	描述
活动	The mouse cursor is over the widget and pressing a mouse button will cause some action to occur
禁用	在程序控制下控件是禁用的
焦点	控件有键盘焦点
按压	Widget is being pressed
选择	“On”, “true”, or “current” for things like Checkbuttons and radiobuttons
背景	Windows and Mac have a notion of an “active” or foreground window. The *background* state is set for widgets in a background window, and cleared for those in the foreground window
只读	Widget should not allow user modification
alternate	A widget-specific alternate display format
无效的	控件的值是无效的

A state specification is a sequence of state names, optionally prefixed with an exclamation point indicating that the bit is off.

ttk.Widget

Besides the methods described below, the `ttk.Widget` supports the methods `tkinter.Widget.cget()` and `tkinter.Widget.configure()`.

class tkinter.ttk.**Widget**

identify(*x*, *y*)
Returns the name of the element at position *x* *y*, or the empty string if the point does not lie within any element.

x and *y* are pixel coordinates relative to the widget.

instate(*statespec*, *callback=None*, **args*, ***kw*)
Test the widget’s state. If a callback is not specified, returns `True` if the widget state matches *statespec* and `False` otherwise. If callback is specified then it is called with args if widget state matches *statespec*.

state(*statespec=None*)
Modify or inquire widget state. If *statespec* is specified, sets the widget state according to it and return a new *statespec* indicating which flags were changed. If *statespec* is not specified, returns the currently-enabled state flags.

statespec will usually be a list or a tuple.

25.2.4 组合框

The `ttk.Combobox` widget combines a text field with a pop-down list of values. This widget is a subclass of `Entry`.

Besides the methods inherited from *`Widget`*: `Widget.cget()`, `Widget.configure()`, *`Widget.identify()`*, *`Widget.instate()`* and *`Widget.state()`*, and the following inherited from `Entry`: `Entry.bbox()`, `Entry.delete()`, `Entry.icursor()`, `Entry.index()`, `Entry.insert()`, `Entry.selection()`, `Entry.xview()`, it has some other methods, described at `ttk.Combobox`.

选项

This widget accepts the following specific options:

选项	描述
exportselection	Boolean value. If set, the widget selection is linked to the Window Manager selection (which can be returned by invoking Misc.selection_get, for example).
justify	Specifies how the text is aligned within the widget. One of “left”, “center”, or “right”.
height	Specifies the height of the pop-down listbox, in rows.
postcommand	A script (possibly registered with Misc.register) that is called immediately before displaying the values. It may specify which values to display.
状况	One of “normal”, “readonly”, or “disabled”. In the “readonly” state, the value may not be edited directly, and the user can only selection of the values from the dropdown list. In the “normal” state, the text field is directly editable. In the “disabled” state, no interaction is possible.
文本变量	Specifies a name whose value is linked to the widget value. Whenever the value associated with that name changes, the widget value is updated, and vice versa. See `tkinter.StringVar`.
值	Specifies the list of values to display in the drop-down listbox.
宽度	Specifies an integer value indicating the desired width of the entry window, in average-size characters of the widget’s font.

虚拟事件

The combobox widgets generates a **«ComboboxSelected»** virtual event when the user selects an element from the list of values.

ttk.Combobox

class tkinter.ttk.**Combobox**

current(*newindex=None*)
If *newindex* is specified, sets the combobox value to the element position *newindex*. Otherwise, returns the index of the current value or -1 if the current value is not in the values list.

get()
Returns the current value of the combobox.

set(*value*)
Sets the value of the combobox to *value*.

25.2.5 笔记本

Ttk Notebook widget manages a collection of windows and displays a single one at a time. Each child window is associated with a tab, which the user may select to change the currently-displayed window.

选项

This widget accepts the following specific options:

选项	描述
height	If present and greater than zero, specifies the desired height of the pane area (not including internal padding or tabs). Otherwise, the maximum height of all panes is used.
padding	Specifies the amount of extra space to add around the outside of the notebook. The padding is a list up to four length specifications left top right bottom. If fewer than four elements are specified, bottom defaults to top, right defaults to left, and top defaults to left.
宽度	If present and greater than zero, specified the desired width of the pane area (not including internal padding). Otherwise, the maximum width of all panes is used.

Tab 选项

There are also specific options for tabs:

选项	描述
状况	Either “normal”, “disabled” or “hidden”. If “disabled”, then the tab is not selectable. If “hidden”, then the tab is not shown.
sticky	Specifies how the child window is positioned within the pane area. Value is a string containing zero or more of the characters “n”, “s”, “e” or “w”. Each letter refers to a side (north, south, east or west) that the child window will stick to, as per the `grid()` geometry manager.
padding	Specifies the amount of extra space to add between the notebook and this pane. Syntax is the same as for the option padding used by this widget.
文本	Specifies a text to be displayed in the tab.
图片	Specifies an image to display in the tab. See the option image described in *`Widget`*.
compound	Specifies how to display the image relative to the text, in the case both options text and image are present. See *Label Options* for legal values.
下划线	Specifies the index (0-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if *`Notebook.enable_traversal()`* is called.

Tab Identifiers

The tab_id present in several methods of `ttk.Notebook` may take any of the following forms:

- An integer between zero and the number of tabs
- The name of a child window
- A positional specification of the form “@x,y”, which identifies the tab
- The literal string “current”, which identifies the currently-selected tab
- The literal string “end”, which returns the number of tabs (only valid for *`Notebook.index()`*)

Virtual Events

This widget generates a **«NotebookTabChanged»** virtual event after a new tab is selected.

ttk.Notebook

class tkinter.ttk.**Notebook**

add(*child*, ***kw*)
Adds a new tab to the notebook.

If window is currently managed by the notebook but hidden, it is restored to its previous position.

See *Tab Options* for the list of available options.

forget(*tab_id*)
Removes the tab specified by *tab_id*, unmaps and unmanages the associated window.

hide(*tab_id*)
Hides the tab specified by *tab_id*.

The tab will not be displayed, but the associated window remains managed by the notebook and its configuration remembered. Hidden tabs may be restored with the `add()` command.

identify(*x*, *y*)
Returns the name of the tab element at position *x*, *y*, or the empty string if none.

index(*tab_id*)
Returns the numeric index of the tab specified by *tab_id*, or the total number of tabs if *tab_id* is the string "end".

insert(*pos*, *child*, ***kw*)
Inserts a pane at the specified position.

pos is either the string "end", an integer index, or the name of a managed child. If *child* is already managed by the notebook, moves it to the specified position.

See *Tab Options* for the list of available options.

select(*tab_id=None*)
Selects the specified *tab_id*.

The associated child window will be displayed, and the previously-selected window (if different) is unmapped. If *tab_id* is omitted, returns the widget name of the currently selected pane.

tab(*tab_id*, *option=None*, ***kw*)
Query or modify the options of the specific *tab_id*.

If *kw* is not given, returns a dictionary of the tab option values. If *option* is specified, returns the value of that *option*. Otherwise, sets the options to the corresponding values.

tabs()
Returns a list of windows managed by the notebook.

enable_traversal()
Enable keyboard traversal for a toplevel window containing this notebook.

This will extend the bindings for the toplevel window containing the notebook as follows:

- `Control-Tab`: selects the tab following the currently selected one.
- `Shift-Control-Tab`: selects the tab preceding the currently selected one.

- `Alt-K`: where *K* is the mnemonic (underlined) character of any tab, will select that tab.

Multiple notebooks in a single toplevel may be enabled for traversal, including nested notebooks. However, notebook traversal only works properly if all panes have the notebook they are in as master.

25.2.6 Progressbar

The `ttk.Progressbar` widget shows the status of a long-running operation. It can operate in two modes: 1) the determinate mode which shows the amount completed relative to the total amount of work to be done and 2) the indeterminate mode which provides an animated display to let the user know that work is progressing.

选项

This widget accepts the following specific options:

选项	描述
orient	One of “horizontal” or “vertical”. Specifies the orientation of the progress bar.
length	Specifies the length of the long axis of the progress bar (width if horizontal, height if vertical).
模式	One of “determinate” or “indeterminate”.
maximum	A number specifying the maximum value. Defaults to 100.
值	The current value of the progress bar. In “determinate” mode, this represents the amount of work completed. In “indeterminate” mode, it is interpreted as modulo *maximum*; that is, the progress bar completes one “cycle” when its value increases by *maximum*.
variable	A name which is linked to the option value. If specified, the value of the progress bar is automatically set to the value of this name whenever the latter is modified.
phase	Read-only option. The widget periodically increments the value of this option whenever its value is greater than 0 and, in determinate mode, less than maximum. This option may be used by the current theme to provide additional animation effects.

ttk.Progressbar

class tkinter.ttk.**Progressbar**

start(*interval=None*)
Begin autoincrement mode: schedules a recurring timer event that calls *Progressbar.step()* every *interval* milliseconds. If omitted, *interval* defaults to 50 milliseconds.

step(*amount=None*)
Increments the progress bar’s value by *amount*.

amount defaults to 1.0 if omitted.

stop()
Stop autoincrement mode: cancels any recurring timer event initiated by *Progressbar.start()* for this progress bar.

25.2.7 Separator

The `ttk.Separator` widget displays a horizontal or vertical separator bar.

It has no other methods besides the ones inherited from `ttk.Widget`.

选项

This widget accepts the following specific option:

选项	描述
orient	One of “horizontal” or “vertical”. Specifies the orientation of the separator.

25.2.8 Sizegrip

The `ttk.Sizegrip` widget (also known as a grow box) allows the user to resize the containing toplevel window by pressing and dragging the grip.

This widget has neither specific options nor specific methods, besides the ones inherited from `ttk.Widget`.

Platform-specific notes

- On MacOS X, toplevel windows automatically include a built-in size grip by default. Adding a `Sizegrip` is harmless, since the built-in grip will just mask the widget.

Bugs

- If the containing toplevel’s position was specified relative to the right or bottom of the screen (e.g. …), the `Sizegrip` widget will not resize the window.
- This widget supports only “southeast” resizing.

25.2.9 Treeview

The `ttk.Treeview` widget displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

The order in which data values are displayed may be controlled by setting the widget option `displaycolumns`. The tree widget can also display column headings. Columns may be accessed by number or symbolic names listed in the widget option columns. See *Column Identifiers*.

Each item is identified by a unique name. The widget will generate item IDs if they are not supplied by the caller. There is a distinguished root item, named `{}`. The root item itself is not displayed; its children appear at the top level of the hierarchy.

Each item also has a list of tags, which can be used to associate event bindings with individual items and control the appearance of the item.

The Treeview widget supports horizontal and vertical scrolling, according to the options described in *Scrollable Widget Options* and the methods *`Treeview.xview()`* and *`Treeview.yview()`*.

选项

This widget accepts the following specific options:

选项	描述
columns	A list of column identifiers, specifying the number of columns and their names.
displaycolumns	A list of column identifiers (either symbolic or integer indices) specifying which data columns are displayed and the order in which they appear, or the string “#all”.
height	Specifies the number of rows which should be visible. Note: the requested width is determined from the sum of the column widths.
padding	Specifies the internal padding for the widget. The padding is a list of up to four length specifications.
selectmode	Controls how the built-in class bindings manage the selection. One of “extended”, “browse” or “none”. If set to “extended” (the default), multiple items may be selected. If “browse”, only a single item will be selected at a time. If “none”, the selection will not be changed. Note that the application code and tag bindings can set the selection however they wish, regardless of the value of this option.
show	A list containing zero or more of the following values, specifying which elements of the tree to display. • tree: display tree labels in column #0. • headings: display the heading row. The default is “tree headings”, i.e., show all elements. **Note**: Column #0 always refers to the tree column, even if show=”tree” is not specified.

Item Options

The following item options may be specified for items in the insert and item widget commands.

选项	描述
文本	The textual label to display for the item.
图片	A Tk Image, displayed to the left of the label.
值	The list of values associated with the item. Each item should have the same number of values as the widget option columns. If there are fewer values than columns, the remaining values are assumed empty. If there are more values than columns, the extra values are ignored.
open	True/False value indicating whether the item’s children should be displayed or hidden.
tags	A list of tags associated with this item.

Tag Options

The following options may be specified on tags:

选项	描述
foreground	Specifies the text foreground color.
背景	Specifies the cell or item background color.
font	Specifies the font to use when drawing text.
图片	Specifies the item image, in case the item’s image option is empty.

Column Identifiers

Column identifiers take any of the following forms:

- A symbolic name from the list of columns option.
- An integer n, specifying the nth data column.
- A string of the form #n, where n is an integer, specifying the nth display column.

注释:

- Item’s option values may be displayed in a different order than the order in which they are stored.
- Column #0 always refers to the tree column, even if show=”tree” is not specified.

A data column number is an index into an item’s option values list; a display column number is the column number in the tree where the values are displayed. Tree labels are displayed in column #0. If option displaycolumns is not set, then data column n is displayed in column #n+1. Again, **column #0 always refers to the tree column**.

Virtual Events

The Treeview widget generates the following virtual events.

Event	描述
«TreeviewSelect»	Generated whenever the selection changes.
«TreeviewOpen»	Generated just before settings the focus item to open=True.
«TreeviewClose»	Generated just after setting the focus item to open=False.

The *`Treeview.focus()`* and *`Treeview.selection()`* methods can be used to determine the affected item or items.

ttk.Treeview

class tkinter.ttk.**Treeview**

bbox(*item*, *column=None*)

Returns the bounding box (relative to the treeview widget’s window) of the specified *item* in the form (x, y, width, height).

If *column* is specified, returns the bounding box of that cell. If the *item* is not visible (i.e., if it is a descendant of a closed item or is scrolled offscreen), returns an empty string.

get_children(*item=None*)
Returns the list of children belonging to *item*.

If *item* is not specified, returns root children.

set_children(*item*, **newchildren*)
Replaces *item*' s child with *newchildren*.

Children present in *item* that are not present in *newchildren* are detached from the tree. No items in *newchildren* may be an ancestor of *item*. Note that not specifying *newchildren* results in detaching *item*' s children.

column(*column*, *option=None*, ***kw*)
Query or modify the options for the specified *column*.

If *kw* is not given, returns a dict of the column option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **id** Returns the column name. This is a read-only option.
- **anchor: One of the standard Tk anchor values.** Specifies how the text in this column should be aligned with respect to the cell.
- **minwidth: width** The minimum width of the column in pixels. The treeview widget will not make the column any smaller than specified by this option when the widget is resized or the user drags a column.
- **stretch: True/False** 指明列宽度是否应该在部件大小被改变时进行相应的调整。
- **width: width** 以像素表示的列宽度。

要配置树的列，则调用此方法并附带参数 column = “#0”

delete(**items*)
Delete all specified *items* and all their descendants.

The root item may not be deleted.

detach(**items*)
Unlinks all of the specified *items* from the tree.

The items and all of their descendants are still present, and may be reinserted at another point in the tree, but will not be displayed.

The root item may not be detached.

exists(*item*)
Returns `True` if the specified *item* is present in the tree.

focus(*item=None*)
If *item* is specified, sets the focus item to *item*. Otherwise, returns the current focus item, or ‘’ if there is none.

heading(*column*, *option=None*, ***kw*)
Query or modify the heading options for the specified *column*.

If *kw* is not given, returns a dict of the heading option values. If *option* is specified then the value for that *option* is returned. Otherwise, sets the options to the corresponding values.

The valid options/values are:

- **text: text** The text to display in the column heading.
- **image: imageName** Specifies an image to display to the right of the column heading.

- **anchor: anchor** Specifies how the heading text should be aligned. One of the standard Tk anchor values.
- **command: callback** A callback to be invoked when the heading label is pressed.

To configure the tree column heading, call this with column = "#0".

identify (*component*, *x*, *y*)
Returns a description of the specified *component* under the point given by *x* and *y*, or the empty string if no such *component* is present at that position.

identify_row (*y*)
Returns the item ID of the item at position *y*.

identify_column (*x*)
Returns the data column identifier of the cell at position *x*.

The tree column has ID #0.

identify_region (*x*, *y*)
Returns one of:

region	meaning
heading	Tree heading area.
separator	Space between two columns headings.
tree	The tree area.
cell	A data cell.

Availability: Tk 8.6.

identify_element (*x*, *y*)
Returns the element at position *x*, *y*.

Availability: Tk 8.6.

index (*item*)
Returns the integer index of *item* within its parent's list of children.

insert (*parent*, *index*, *iid=None*, ***kw*)
Creates a new item and returns the item identifier of the newly created item.

parent is the item ID of the parent item, or the empty string to create a new top-level item. *index* is an integer, or the value "end", specifying where in the list of parent's children to insert the new item. If *index* is less than or equal to zero, the new node is inserted at the beginning; if *index* is greater than or equal to the current number of children, it is inserted at the end. If *iid* is specified, it is used as the item identifier; *iid* must not already exist in the tree. Otherwise, a new unique identifier is generated.

See *Item Options* for the list of available points.

item (*item*, *option=None*, ***kw*)
Query or modify the options for the specified *item*.

If no options are given, a dict with options/values for the item is returned. If *option* is specified then the value for that option is returned. Otherwise, sets the options to the corresponding values as given by *kw*.

move (*item*, *parent*, *index*)
Moves *item* to position *index* in *parent*'s list of children.

It is illegal to move an item under one of its descendants. If *index* is less than or equal to zero, *item* is moved to the beginning; if greater than or equal to the number of children, it is moved to the end. If *item* was detached it is reattached.

next (*item*)
Returns the identifier of *item*'s next sibling, or '' if *item* is the last child of its parent.

parent (*item*)
Returns the ID of the parent of *item*, or '' if *item* is at the top level of the hierarchy.

prev (*item*)
Returns the identifier of *item*'s previous sibling, or '' if *item* is the first child of its parent.

reattach (*item*, *parent*, *index*)
An alias for `Treeview.move()`.

see (*item*)
Ensure that *item* is visible.

Sets all of *item*'s ancestors open option to `True`, and scrolls the widget if necessary so that *item* is within the visible portion of the tree.

selection (*selop=None*, *items=None*)
If *selop* is not specified, returns selected items. Otherwise, it will act according to the following selection methods.

Deprecated since version 3.6, will be removed in version 3.8: Using `selection()` for changing the selection state is deprecated. Use the following selection methods instead.

selection_set (**items*)
items becomes the new selection.

在 3.6 版更改: *items* can be passed as separate arguments, not just as a single tuple.

selection_add (**items*)
Add *items* to the selection.

在 3.6 版更改: *items* can be passed as separate arguments, not just as a single tuple.

selection_remove (**items*)
Remove *items* from the selection.

在 3.6 版更改: *items* can be passed as separate arguments, not just as a single tuple.

selection_toggle (**items*)
Toggle the selection state of each item in *items*.

在 3.6 版更改: *items* can be passed as separate arguments, not just as a single tuple.

set (*item*, *column=None*, *value=None*)
With one argument, returns a dictionary of column/value pairs for the specified *item*. With two arguments, returns the current value of the specified *column*. With three arguments, sets the value of given *column* in given *item* to the specified *value*.

tag_bind (*tagname*, *sequence=None*, *callback=None*)
Bind a callback for the given event *sequence* to the tag *tagname*. When an event is delivered to an item, the callbacks for each of the item's tags option are called.

tag_configure (*tagname*, *option=None*, ***kw*)
Query or modify the options for the specified *tagname*.

If *kw* is not given, returns a dict of the option settings for *tagname*. If *option* is specified, returns the value for that *option* for the specified *tagname*. Otherwise, sets the options to the corresponding values for the given *tagname*.

tag_has (*tagname*, *item=None*)
If *item* is specified, returns 1 or 0 depending on whether the specified *item* has the given *tagname*. Otherwise, returns a list of all items that have the specified tag.

Availability: Tk 8.6

xview(**args*)
Query or modify horizontal position of the treeview.

yview(**args*)
Query or modify vertical position of the treeview.

25.2.10 Ttk Styling

Each widget in `ttk` is assigned a style, which specifies the set of elements making up the widget and how they are arranged, along with dynamic and default settings for element options. By default the style name is the same as the widget's class name, but it may be overridden by the widget's style option. If you don't know the class name of a widget, use the method `Misc.winfo_class()` (somewidget.winfo_class()).

参见:

Tcl'2004 conference presentation This document explains how the theme engine works

class tkinter.ttk.**Style**
This class is used to manipulate the style database.

configure(*style*, *query_opt=None*, ***kw*)
Query or set the default value of the specified option(s) in *style*.

Each key in *kw* is an option and each value is a string identifying the value for that option.

For example, to change every default button to be a flat button with some padding and a different background color:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(*style*, *query_opt=None*, ***kw*)
Query or sets dynamic values of the specified option(s) in *style*.

Each key in *kw* is an option and each value should be a list or a tuple (usually) containing statespecs grouped in tuples, lists, or some other preference. A statespec is a compound of one or more states and then a value.

An example may make it more understandable:

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
```

(下页继续)

(续上页)

```
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

Note that the order of the (states, value) sequences for an option does matter, if the order is changed to `[('active', 'blue'), ('pressed', 'red')]` in the foreground option, for example, the result would be a blue foreground when the widget were in active or pressed states.

lookup (*style*, *option*, *state=None*, *default=None*)

Returns the value specified for *option* in *style*.

If *state* is specified, it is expected to be a sequence of one or more states. If the *default* argument is set, it is used as a fallback value in case no specification for option is found.

To check what font a Button uses by default:

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout (*style*, *layoutspec=None*)

Define the widget layout for given *style*. If *layoutspec* is omitted, return the layout specification for given style.

layoutspec, if specified, is expected to be a list or some other sequence type (excluding strings), where each item should be a tuple and the first item is the layout name and the second item should have the format described in *Layouts*.

To understand the format, see the following example (it is not intended to do anything useful):

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.layout("TMenubutton", [
   ("Menubutton.background", None),
   ("Menubutton.button", {"children":
       [("Menubutton.focus", {"children":
           [("Menubutton.padding", {"children":
               [("Menubutton.label", {"side": "left", "expand": 1})]
           })]
       })]
   }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create (*elementname*, *etype*, **args*, ***kw*)

Create a new element in the current theme, of the given *etype* which is expected to be either "image", "from" or "vsapi". The latter is only available in Tk 8.6a for Windows XP and Vista and is not described here.

If "image" is used, *args* should contain the default image name followed by statespec/value pairs (this is the imagespec), and *kw* may have the following options:

- **border=padding** padding is a list of up to four integers, specifying the left, top, right, and bottom borders, respectively.
- **height=height** Specifies a minimum height for the element. If less than zero, the base image's height is used as a default.
- **padding=padding** Specifies the element's interior padding. Defaults to border's value if not specified.
- **sticky=spec** Specifies how the image is placed within the final parcel. spec contains zero or more characters "n", "s", "w", or "e".
- **width=width** Specifies a minimum width for the element. If less than zero, the base image's width is used as a default.

If "from" is used as the value of *etype*, `element_create()` will clone an existing element. *args* is expected to contain a themename, from which the element will be cloned, and optionally an element to clone from. If this element to clone from is not specified, an empty element will be used. *kw* is discarded.

element_names()
Returns the list of elements defined in the current theme.

element_options(*elementname*)
Returns the list of *elementname*'s options.

theme_create(*themename*, *parent=None*, *settings=None*)
Create a new theme.

It is an error if *themename* already exists. If *parent* is specified, the new theme will inherit styles, elements and layouts from the parent theme. If *settings* are present they are expected to have the same syntax used for `theme_settings()`.

theme_settings(*themename*, *settings*)
Temporarily sets the current theme to *themename*, apply specified *settings* and then restore the previous theme.

Each key in *settings* is a style and each value may contain the keys 'configure', 'map', 'layout' and 'element create' and they are expected to have the same format as specified by the methods `Style.configure()`, `Style.map()`, `Style.layout()` and `Style.element_create()` respectively.

As an example, let's change the Combobox for the default theme a bit:

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})
```

(下页继续)

(续上页)

```
combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()
Returns a list of all known themes.

theme_use(*themename=None*)
If *themename* is not given, returns the theme in use. Otherwise, sets the current theme to *themename*, refreshes all widgets and emits a «ThemeChanged» event.

Layouts

A layout can be just `None`, if it takes no options, or a dict of options specifying how to arrange the element. The layout mechanism uses a simplified version of the pack geometry manager: given an initial cavity, each element is allocated a parcel. Valid options/values are:

- **side: whichside** Specifies which side of the cavity to place the element; one of top, right, bottom or left. If omitted, the element occupies the entire cavity.
- **sticky: nswe** Specifies where the element is placed inside its allocated parcel.
- **unit: 0 or 1** If set to 1, causes the element and all of its descendants to be treated as a single element for the purposes of *`Widget.identify()`* et al. It's used for things like scrollbar thumbs with grips.
- **children: [sublayout…]** Specifies a list of elements to place inside the element. Each element is a tuple (or other sequence type) where the first item is the layout name, and the other is a *Layout*.

25.3 `tkinter.tix` —TK 扩展包

源代码: Lib/tkinter/tix.py

3.6 版后已移除: 这个 TK 扩展已无人维护所以请不要在新代码中使用。请改用*`tkinter.ttk`*。

The *`tkinter.tix`* (Tk Interface Extension) module provides an additional rich set of widgets. Although the standard Tk library has many useful widgets, they are far from complete. The *`tkinter.tix`* library provides most of the commonly needed widgets that are missing from standard Tk: *`HList`*, *`ComboBox`*, *`Control`* (a.k.a. SpinBox) and an assortment of scrollable widgets. *`tkinter.tix`* also includes many more widgets that are generally useful in a wide range of applications: *`NoteBook`*, *`FileEntry`*, *`PanedWindow`*, etc; there are more than 40 of them.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

参见:

Tix Homepage The home page for `Tix`. This includes links to additional documentation and downloads.

Tix Man Pages On-line version of the man pages and reference material.

Tix Programming Guide On-line version of the programmer's reference material.

Tix Development Applications Tix applications for development of Tix and Tkinter programs. Tide applications work under Tk or Tkinter, and include **`TixInspect`**, an inspector to remotely modify and debug Tix/Tk/Tkinter applications.

25.3.1 Using Tix

class tkinter.tix.**Tk**(*screenName=None*, *baseName=None*, *className='Tix'*)
Toplevel widget of Tix which represents mostly the main window of an application. It has an associated Tcl interpreter.

Classes in the *tkinter.tix* module subclasses the classes in the *tkinter*. The former imports the latter, so to use *tkinter.tix* with Tkinter, all you need to do is to import one module. In general, you can just import *tkinter.tix*, and replace the toplevel call to *tkinter.Tk* with `tix.Tk`:

```
from tkinter import tix
from tkinter.constants import *
root = tix.Tk()
```

To use *tkinter.tix*, you must have the Tix widgets installed, usually alongside your installation of the Tk widgets. To test your installation, try the following:

```
from tkinter import tix
root = tix.Tk()
root.tk.eval('package require Tix')
```

If this fails, you have a Tk installation problem which must be resolved before proceeding. Use the environment variable `TIX_LIBRARY` to point to the installed Tix library directory, and make sure you have the dynamic object library (`tix8183.dll` or `libtix8183.so`) in the same directory that contains your Tk dynamic object library (`tk8183.dll` or `libtk8183.so`). The directory with the dynamic object library should also have a file called `pkgIndex.tcl` (case sensitive), which contains the line:

```
package ifneeded Tix 8.1 [list load "[file join $dir tix8183.dll]" Tix]
```

25.3.2 Tix Widgets

Tix introduces over 40 widget classes to the *tkinter* repertoire.

Basic Widgets

class tkinter.tix.**Balloon**
A Balloon that pops up over a widget to provide help. When the user moves the cursor inside a widget to which a Balloon widget has been bound, a small pop-up window with a descriptive message will be shown on the screen.

class tkinter.tix.**ButtonBox**
The ButtonBox widget creates a box of buttons, such as is commonly used for `Ok Cancel`.

class tkinter.tix.**ComboBox**
The ComboBox widget is similar to the combo box control in MS Windows. The user can select a choice by either typing in the entry subwidget or selecting from the listbox subwidget.

class tkinter.tix.**Control**
The Control 控件又名 `SpinBox` 控件。用户可通过点按两个方向键或直接输入内容来调整数值。更新的数值将被检查是否在用户定义的上下限之内。

class tkinter.tix.**LabelEntry**
The LabelEntry widget packages an entry widget and a label into one mega widget. It can be used to simplify the creation of "entry-form" type of interface.

class tkinter.tix.**LabelFrame**
The LabelFrame widget packages a frame widget and a label into one mega widget. To create widgets inside a LabelFrame widget, one creates the new widgets relative to the `frame` subwidget and manage them inside the `frame` subwidget.

class tkinter.tix.**Meter**
The Meter widget can be used to show the progress of a background job which may take a long time to execute.

class tkinter.tix.**OptionMenu**
The OptionMenu creates a menu button of options.

class tkinter.tix.**PopupMenu**
The PopupMenu widget can be used as a replacement of the `tk_popup` command. The advantage of the `Tix` *`PopupMenu`* widget is it requires less application code to manipulate.

class tkinter.tix.**Select**
The Select widget is a container of button subwidgets. It can be used to provide radio-box or check-box style of selection options for the user.

class tkinter.tix.**StdButtonBox**
The StdButtonBox widget is a group of standard buttons for Motif-like dialog boxes.

File Selectors

class tkinter.tix.**DirList**
The DirList widget displays a list view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.**DirTree**
The DirTree widget displays a tree view of a directory, its previous directories and its sub-directories. The user can choose one of the directories displayed in the list or change to another directory.

class tkinter.tix.**DirSelectDialog**
The DirSelectDialog widget presents the directories in the file system in a dialog window. The user can use this dialog window to navigate through the file system to select the desired directory.

class tkinter.tix.**DirSelectBox**
The *`DirSelectBox`* is similar to the standard Motif(TM) directory-selection box. It is generally used for the user to choose a directory. DirSelectBox stores the directories mostly recently selected into a ComboBox widget so that they can be quickly selected again.

class tkinter.tix.**ExFileSelectBox**
The ExFileSelectBox widget is usually embedded in a tixExFileSelectDialog widget. It provides a convenient method for the user to select files. The style of the *`ExFileSelectBox`* widget is very similar to the standard file dialog on MS Windows 3.1.

class tkinter.tix.**FileSelectBox**
The FileSelectBox is similar to the standard Motif(TM) file-selection box. It is generally used for the user to choose a file. FileSelectBox stores the files mostly recently selected into a *`ComboBox`* widget so that they can be quickly selected again.

class tkinter.tix.**FileEntry**
The FileEntry widget can be used to input a filename. The user can type in the filename manually. Alternatively, the user can press the button widget that sits next to the entry, which will bring up a file selection dialog.

Hierarchical ListBox

class tkinter.tix.**HList**
The HList widget can be used to display any data that have a hierarchical structure, for example, file system directory trees. The list entries are indented and connected by branch lines according to their places in the hierarchy.

class tkinter.tix.**CheckList**
The CheckList widget displays a list of items to be selected by the user. CheckList acts similarly to the Tk checkbutton or radiobutton widgets, except it is capable of handling many more items than checkbuttons or radiobuttons.

class tkinter.tix.**Tree**
The Tree widget can be used to display hierarchical data in a tree form. The user can adjust the view of the tree by opening or closing parts of the tree.

Tabular ListBox

class tkinter.tix.**TList**
The TList widget can be used to display data in a tabular format. The list entries of a *TList* widget are similar to the entries in the Tk listbox widget. The main differences are (1) the *TList* widget can display the list entries in a two dimensional format and (2) you can use graphical images as well as multiple colors and fonts for the list entries.

Manager Widgets

class tkinter.tix.**PanedWindow**
The PanedWindow widget allows the user to interactively manipulate the sizes of several panes. The panes can be arranged either vertically or horizontally. The user changes the sizes of the panes by dragging the resize handle between two panes.

class tkinter.tix.**ListNoteBook**
The ListNoteBook widget is very similar to the `TixNoteBook` widget: it can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages (windows). At one time only one of these pages can be shown. The user can navigate through these pages by choosing the name of the desired page in the `hlist` subwidget.

class tkinter.tix.**NoteBook**
The NoteBook widget can be used to display many windows in a limited space using a notebook metaphor. The notebook is divided into a stack of pages. At one time only one of these pages can be shown. The user can navigate through these pages by choosing the visual "tabs" at the top of the NoteBook widget.

Image Types

The *tkinter.tix* module adds:

- pixmap capabilities to all *tkinter.tix* and *tkinter* widgets to create color images from XPM files.
- Compound image types can be used to create images that consists of multiple horizontal lines; each line is composed of a series of items (texts, bitmaps, images or spaces) arranged from left to right. For example, a compound image can be used to display a bitmap and a text string simultaneously in a Tk `Button` widget.

Miscellaneous Widgets

class tkinter.tix.**InputOnly**
The InputOnly widgets are to accept inputs from the user, which can be done with the `bind` command (Unix only).

Form Geometry Manager

In addition, *tkinter.tix* augments *tkinter* by providing:

class tkinter.tix.**Form**
The Form geometry manager based on attachment rules for all Tk widgets.

25.3.3 Tix Commands

class tkinter.tix.**tixCommand**
The tix commands provide access to miscellaneous elements of `Tix`'s internal state and the `Tix` application context. Most of the information manipulated by these methods pertains to the application as a whole, or to a screen or display, rather than to a particular window.

To view the current settings, the common usage is:

```
from tkinter import tix
root = tix.Tk()
print(root.tix_configure())
```

tixCommand.**tix_configure**(*cnf=None*, ***kw*)
Query or modify the configuration options of the Tix application context. If no option is specified, returns a dictionary all of the available options. If option is specified with no value, then the method returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the method modifies the given option(s) to have the given value(s); in this case the method returns an empty string. Option may be any of the configuration options.

tixCommand.**tix_cget**(*option*)
Returns the current value of the configuration option given by *option*. Option may be any of the configuration options.

tixCommand.**tix_getbitmap**(*name*)
Locates a bitmap file of the name `name.xpm` or `name` in one of the bitmap directories (see the *tix_addbitmapdir()* method). By using *tix_getbitmap()*, you can avoid hard coding the pathnames of the bitmap files in your application. When successful, it returns the complete pathname of the bitmap file, prefixed with the character `@`. The returned value can be used to configure the `bitmap` option of the Tk and Tix widgets.

tixCommand.**tix_addbitmapdir**(*directory*)
Tix maintains a list of directories under which the *tix_getimage()* and *tix_getbitmap()* methods will search for image files. The standard bitmap directory is `$TIX_LIBRARY/bitmaps`. The *tix_addbitmapdir()* method adds *directory* into this list. By using this method, the image files of an applications can also be located using the *tix_getimage()* or *tix_getbitmap()* method.

tixCommand.**tix_filedialog**([*dlgclass*])
Returns the file selection dialog that may be shared among different calls from this application. This method will create a file selection dialog widget when it is called the first time. This dialog will be returned by all subsequent calls to *tix_filedialog()*. An optional dlgclass parameter can be passed as a string to specified what type of file selection dialog widget is desired. Possible options are `tix`, `FileSelectDialog` or `tixExFileSelectDialog`.

tixCommand.**tix_getimage**(*self*, *name*)
Locates an image file of the name `name.xpm`, `name.xbm` or `name.ppm` in one of the bitmap directories (see the *tix_addbitmapdir()* method above). If more than one file with the same name (but different extensions) exist, then the image type is chosen according to the depth of the X display: xbm images are chosen on monochrome displays and color images are chosen on color displays. By using *tix_getimage()*, you can avoid hard coding the pathnames of the image files in your application. When successful, this method returns the name of the newly created image, which can be used to configure the `image` option of the Tk and Tix widgets.

tixCommand.**tix_option_get**(*name*)
Gets the options maintained by the Tix scheme mechanism.

tixCommand.**tix_resetoptions**(*newScheme*, *newFontSet*[, *newScmPrio*])
Resets the scheme and fontset of the Tix application to *newScheme* and *newFontSet*, respectively. This affects only those widgets created after this call. Therefore, it is best to call the resetoptions method before the creation of any widgets in a Tix application.

The optional parameter *newScmPrio* can be given to reset the priority level of the Tk options set by the Tix schemes.

Because of the way Tk handles the X option database, after Tix has been has imported and inited, it is not possible to reset the color schemes and font sets using the `tix_config()` method. Instead, the *tix_resetoptions()* method must be used.

25.4 tkinter.scrolledtext —滚动文字控件

源代码： Lib/tkinter/scrolledtext.py

---

tkinter.scrolledtext 模块提供一个同名的类，实现了一个带有垂直滚动条的文字控件。使用 `ScrolledText` 类会比直接配置一个文本控件和滚动条简单。它的构造函数与 `tkinter.Text` 类相同。

文本控件与滚动条打包在一个 `Frame` 中，`Grid` 方法和 `Pack` 方法的布局管理器从 `Frame` 对象中获得。这允许 `ScrolledText` 控件可以直接用于实现大多数正常的布局管理行为。

如果需要更具体的控制，可以使用以下属性：

`ScrolledText`.**frame**
围绕文本和滚动条控件的框架。

`ScrolledText`.**vbar**
滚动条控件。

25.5 IDLE

源代码： Lib/idlelib/

IDLE 是 Python 所内置的开发与学习环境。

IDLE 具有以下特性：

- 编码于 100% 纯正的 Python，使用名为*tkinter* 的图形用户界面工具
- 跨平台：在 Windows、Unix 和 macOS 上工作近似。
- 提供输入输出高亮和错误信息的 Python 命令行窗口（交互解释器）

- 提供多次撤销操作、Python 语法高亮、智能缩进、函数调用提示、自动补全等功能的多窗口文本编辑器
- 在多个窗口中检索，在编辑器中替换文本，以及在多个文件中检索（通过 grep）
- 提供持久保存的断点调试、单步调试、查看本地和全局命名空间功能的调试器
- 配置、浏览以及其它对话框

25.5.1 目录

IDEL 具有两个主要窗口类型，分别是命令行窗口和编辑器窗口。用户可以同时打开多个编辑器窗口。对于 Windows 和 Linux 平台，都有各自的主菜单。如下记录的每个菜单标识着与之关联的窗口类型。

导出窗口，例如使用编辑 => 在文件中查找是编辑器窗口的的一个子类型。它们目前有着相同的主菜单，但是默认标题和上下文菜单不同。

On macOS, there is one application menu. It dynamically changes according to the window currently selected. It has an IDLE menu, and some entries described below are moved around to conform to Apple guidlines.

文件目录（命令行和编辑器）

新建文件 创建一个文件编辑器窗口。

打开… 使用打开窗口以打开一个已存在的文件。

近期文件 打开一个近期文件列表，选取一个以打开它。

打开模块… 打开一个已存在的模块（搜索 sys.path）

类浏览器 于当前所编辑的文件中使用树形结构展示函数、类以及方法。在命令行中，首先打开一个模块。

路径浏览 在树状结构中展示 sys.path 目录、模块、函数、类和方法。

保存 如果文件已经存在，则将当前窗口保存至对应的文件。自打开或上次保存之后经过修改的文件的窗口标题栏首尾将出现星号 * 。如果没有对应的文件，则使用“另存为”代替。

保存为… 使用“保存为”对话框保存当前窗口。被保存的文件将作为当前窗口新的对应文件。

另存为副本… 保存当前窗口至另一个文件，而不修改当前对应文件。

打印窗口 通过默认打印机打印当前窗口。

关闭 关闭当前窗口（如果未保存则询问）。

退出 关闭所有窗口并退出 IDLE（如果未保存则询问）

编辑目录（命令行和编辑器）

撤销操作 撤销当前窗口的最近一次操作。最高可以撤回 1000 条操作记录。

重做 重做当前窗口最近一次所撤销的操作。

剪切 复制选区至系统剪贴板，然后删除选区。

复制 复制选区至系统剪贴板。

粘贴 插入系统剪贴板的内容至当前窗口。

剪贴板功能也可用于上下文目录。

全选 选择当前窗口的全部内容。

查找… 打开一个提供多选项的查找窗口。

再次查找 重复上次搜索，如果结果存在。

查找选区 查找当前选中的字符串，如果存在

在文件中查找… 打开文件查找对话框。将结果输出至新的输出窗口。

替换… 打开查找并替换对话框。

前往行 Move cursor to the line number requested and make that line visible.

提示完成 Open a scrollable list allowing selection of keywords and attributes. See Completions in the Tips sections below.

展开文本 展开键入的前缀以匹配同一窗口中的完整单词；重复以获得不同的扩展。

显示调用贴士 After an unclosed parenthesis for a function, open a small window with function parameter hints.

显示周围括号 突出显示周围的括号。

格式菜单（仅 window 编辑器）

缩进区域 将选定的行右移缩进宽度（默认为 4 个空格）。

区域减少缩进 将选定的行向左移动缩进宽度（默认为 4 个空格）。

区域注释 在所选行的前面插入 ##。

区域取消注释 从所选行中删除开头的#或 ##。

区域添加制表符 将 前导空格变成制表符。（注意：我们建议使用 4 个空格来缩进 Python 代码。）

区域取消制表符 将 所有制表符转换为正确的空格数。

切换标签 打开一个对话框，以在缩进和空格之间切换。

新缩进宽度 打开一个对话框以更改缩进宽度。Python 社区接受的默认值为 4 个空格。

格式段落 在注释块或多行字符串或字符串中的选定行中，重新格式化当前以空行分隔的段落。段落中的所有行的格式都将少于 N 列，其中 N 默认为 72。

尾随空格 Remove trailing space and other whitespace characters after the last non-whitespace character of a line by applying str.rstrip to each line, including lines within multiline strings.

运行菜单（仅 window 编辑器）

Python Shell 打开或唤醒 Python Shell 窗口。

检查模块 检查“编辑器”窗口中当前打开的模块的语法。如果尚未保存该模块，则 IDLE 会提示用户保存或自动保存，如在“空闲设置”对话框的“常规”选项卡中所选择的那样。如果存在语法错误，则会在“编辑器”窗口中指示大概位置。

运行模块 Do Check Module (above). If no error, restart the shell to clean the environment, then execute the module. Output is displayed in the Shell window. Note that output requires use of `print` or `write`. When execution is complete, the Shell retains focus and displays a prompt. At this point, one may interactively explore the result of execution. This is similar to executing a file with `python -i file` at a command line.

Shell 菜单（仅 window 编辑器）

查看最近重启 将 Shell 窗口滚动到上一次 Shell 重启。

重启 Shell 重新启动 shell 以清理环境。

中断执行 停止正在运行的程序。

调试菜单（仅 window 编辑器）

跳转到文件/行 Look on the current line. with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.

调试器（切换） 激活后，在 Shell 中输入的代码或从编辑器中运行的代码将在调试器下运行。在编辑器中，可以使用上下文菜单设置断点。此功能不完整，具有实验性。

堆栈查看器 在树状目录中显示最后一个异常的堆栈回溯，可以访问本地和全局。

自动打开堆栈查看器 在未处理的异常上切换自动打开堆栈查看器。

选项菜单（命令行和编辑器）

配置 IDLE Open a configuration dialog and change preferences for the following: fonts, indentation, keybindings, text color themes, startup windows and size, additional help sources, and extensions (see below). On macOS, open the configuration dialog by selecting Preferences in the application menu. To use a new built-in color theme (IDLE Dark) with older IDLEs, save it as a new custom theme.

Non-default user settings are saved in a .idlerc directory in the user’s home directory. Problems caused by bad user configuration files are solved by editing or deleting one or more of the files in .idlerc.

Code Context (toggle)(Editor Window only) Open a pane at the top of the edit window which shows the block context of the code which has scrolled above the top of the window. Clicking a line in this pane exposes that line at the top of the editor.

Window 菜单（命令行和编辑器）

Zoom Height Toggles the window between normal size and maximum height. The initial size defaults to 40 lines by 80 chars unless changed on the General tab of the Configure IDLE dialog.

The rest of this menu lists the names of all open windows; select one to bring it to the foreground (deiconifying it if necessary).

帮助菜单（命令行和编辑器）

关于 IDLE 显示版本，版权，许可证，荣誉等。

IDLE 帮助 显示此 IDLE 文档，详细介绍菜单选项，基本编辑和导航以及其他技巧。

Python 文档 访问本地 Python 文档（如果已安装），或启动 Web 浏览器并打开 docs.python.org 显示最新的 Python 文档。

海龟演示 Run the turtledemo module with example python code and turtle drawings.

Additional help sources may be added here with the Configure IDLE dialog under the General tab. See the “Help sources” subsection below for more on Help menu choices.

上下文菜单

Open a context menu by right-clicking in a window (Control-click on macOS). Context menus have the standard clipboard functions also on the Edit menu.

剪切 复制选区至系统剪贴板，然后删除选区。

复制 复制选区至系统剪贴板。

粘贴 插入系统剪贴板的内容至当前窗口。

Editor windows also have breakpoint functions. Lines with a breakpoint set are specially marked. Breakpoints only have an effect when running under the debugger. Breakpoints for a file are saved in the user's .idlerc directory.

设置断点 在当前行设置断点

清除断点 清除当前行断点

shell 和输出窗口还具有以下内容。

跳转到文件/行 与调试菜单相同。

The Shell window also has an output squeezing facility explained in the the *Python Shell window* subsection below.

压缩 If the cursor is over an output line, squeeze all the output between the code above and the prompt below down to a 'Squeezed text' label.

25.5.2 编辑和导航

编辑窗口

IDLE may open editor windows when it starts, depending on settings and how you start IDLE. Thereafter, use the File menu. There can be only one open editor window for a given file.

The title bar contains the name of the file, the full path, and the version of Python and IDLE running the window. The status bar contains the line number ('Ln') and column number ('Col'). Line numbers start with 1; column numbers with 0.

IDLE assumes that files with a known .py* extension contain Python code and that other files do not. Run Python code with the Run menu.

按键绑定

在本节中，'C'是指 Windows 和 Unix 上的 `Control` 键，以及 macOS 上的 `Command` 键。

- `Backspace` 向左删除; `Del` 向右删除
- `C-Backspace` 向左删除单词; `C-Del` 向右删除单词
- 方向键和 `Page Up`/`Page Down` 移动
- `C-LeftArrow` 和 `C-RightArrow` 按字移动
- `Home`/`End` 跳转到行首/尾
- `C-Home`/`C-End` 跳转到文档首/尾
- 一些有用的 Emacs 绑定是从 Tcl / Tk 继承的:
  - `C-a` 行首
  - `C-e` 行尾

- C-k 删除行（但未将其放入剪贴板）
- C-l center window around the insertion point
- C-b go backward one character without deleting (usually you can also use the cursor key for this)
- C-f go forward one character without deleting (usually you can also use the cursor key for this)
- C-p go up one line (usually you can also use the cursor key for this)
- C-d 删除下一个字符

Standard keybindings (like C-c to copy and C-v to paste) may work. Keybindings are selected in the Configure IDLE dialog.

自动缩进

After a block-opening statement, the next line is indented by 4 spaces (in the Python Shell window by one tab). After certain keywords (break, return etc.) the next line is dedented. In leading indentation, Backspace deletes up to 4 spaces if they are there. Tab inserts spaces (in the Python Shell window one tab), number depends on Indent width. Currently, tabs are restricted to four spaces due to Tcl/Tk limitations.

See also the indent/dedent region commands in the edit menu.

完成

Completions are supplied for functions, classes, and attributes of classes, both built-in and user-defined. Completions are also provided for filenames.

The AutoCompleteWindow (ACW) will open after a predefined delay (default is two seconds) after a '.' or (in a string) an os.sep is typed. If after one of those characters (plus zero or more other characters) a tab is typed the ACW will open immediately if a possible continuation is found.

If there is only one possible completion for the characters entered, a Tab will supply that completion without opening the ACW.

'Show Completions' will force open a completions window, by default the C-space will open a completions window. In an empty string, this will contain the files in the current directory. On a blank line, it will contain the built-in and user-defined functions and classes in the current namespaces, plus any modules imported. If some characters have been entered, the ACW will attempt to be more specific.

If a string of characters is typed, the ACW selection will jump to the entry most closely matching those characters. Entering a tab will cause the longest non-ambiguous match to be entered in the Editor window or Shell. Two tab in a row will supply the current ACW selection, as will return or a double click. Cursor keys, Page Up/Down, mouse selection, and the scroll wheel all operate on the ACW.

"Hidden" attributes can be accessed by typing the beginning of hidden name after a '.', e.g. '_'. This allows access to modules with __all__ set, or to class-private attributes.

Completions and the 'Expand Word' facility can save a lot of typing!

Completions are currently limited to those in the namespaces. Names in an Editor window which are not via __main__ and *sys.modules* will not be found. Run the module once with your imports to correct this situation. Note that IDLE itself places quite a few modules in sys.modules, so much can be found by default, e.g. the re module.

If you don't like the ACW popping up unbidden, simply make the delay longer or disable the extension.

提示

A calltip is shown when one types (after the name of an *accessible* function. A name expression may include dots and subscripts. A calltip remains until it is clicked, the cursor is moved out of the argument area, or) is typed. When the cursor is in the argument part of a definition, the menu or shortcut display a calltip.

A calltip consists of the function signature and the first line of the docstring. For builtins without an accessible signature, the calltip consists of all lines up the fifth line or the first blank line. These details may change.

The set of *accessible* functions depends on what modules have been imported into the user process, including those imported by Idle itself, and what definitions have been run, all since the last restart.

For example, restart the Shell and enter `itertools.count(`. A calltip appears because Idle imports itertools into the user process for its own use. (This could change.) Enter `turtle.write(` and nothing appears. Idle does not import turtle. The menu or shortcut do nothing either. Enter `import turtle` and then `turtle.write(` will work.

In an editor, import statements have no effect until one runs the file. One might want to run a file after writing the import statements at the top, or immediately run an existing file before editing.

Python Shell 窗口

With IDLE's Shell, one enters, edits, and recalls complete statements. Most consoles and terminals only work with a single physical line at a time.

When one pastes code into Shell, it is not compiled and possibly executed until one hits `Return`. One may edit pasted code first. If one pastes more that one statement into Shell, the result will be a *`SyntaxError`* when multiple statements are compiled as if they were one.

The editing features described in previous subsections work when entering code interactively. IDLE's Shell window also responds to the following keys.

- `C-c` 中断执行命令
- `C-d` sends end-of-file; closes window if typed at a >>> prompt
- `Alt-/` (Expand word) is also useful to reduce typing

  历史命令

  - `Alt-p` retrieves previous command matching what you have typed. On macOS use `C-p`.
  - `Alt-n` retrieves next. On macOS use `C-n`.
  - `Return` while on any previous command retrieves that command

文本颜色

Idle defaults to black on white text, but colors text with special meanings. For the shell, these are shell output, shell error, user output, and user error. For Python code, at the shell prompt or in an editor, these are keywords, builtin class and function names, names following `class` and `def`, strings, and comments. For any text window, these are the cursor (when present), found text (when possible), and selected text.

Text coloring is done in the background, so uncolorized text is occasionally visible. To change the color scheme, use the Configure IDLE dialog Highlighting tab. The marking of debugger breakpoint lines in the editor and text in popups and dialogs is not user-configurable.

25.5.3 启动和代码执行

Upon startup with the `-s` option, IDLE will execute the file referenced by the environment variables `IDLESTARTUP` or `PYTHONSTARTUP`. IDLE first checks for `IDLESTARTUP`; if `IDLESTARTUP` is present the file referenced is run. If `IDLESTARTUP` is not present, IDLE checks for `PYTHONSTARTUP`. Files referenced by these environment variables are convenient places to store functions that are used frequently from the IDLE shell, or for executing import statements to import common modules.

In addition, `Tk` also loads a startup file if it is present. Note that the Tk file is loaded unconditionally. This additional file is `.Idle.py` and is looked for in the user's home directory. Statements in this file will be executed in the Tk namespace, so this file is not useful for importing functions to be used from IDLE's Python shell.

命令行语法

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-           run stdin in shell (- must be last option before args)
```

如果有参数:

- If `-`, `-c`, or `r` is used, all arguments are placed in `sys.argv[1:...]` and `sys.argv[0]` is set to `''`, `'-c'`, or `'-r'`. No editor window is opened, even if that is the default set in the Options dialog.
- Otherwise, arguments are files opened for editing and `sys.argv` reflects the arguments passed to IDLE itself.

启动失败

IDLE uses a socket to communicate between the IDLE GUI process and the user code execution process. A connection must be established whenever the Shell starts or restarts. (The latter is indicated by a divider line that says 'RESTART'). If the user process fails to connect to the GUI process, it displays a `Tk` error box with a 'cannot connect' message that directs the user here. It then exits.

A common cause of failure is a user-written file with the same name as a standard library module, such as *random.py* and *tkinter.py*. When such a file is located in the same directory as a file that is about to be run, IDLE cannot import the stdlib file. The current fix is to rename the user file.

Though less common than in the past, an antivirus or firewall program may stop the connection. If the program cannot be taught to allow the connection, then it must be turned off for IDLE to work. It is safe to allow this internal connection because no data is visible on external ports. A similar problem is a network mis-configuration that blocks connections.

Python installation issues occasionally stop IDLE: multiple versions can clash, or a single installation might need admin access. If one undo the clash, or cannot or does not want to run as admin, it might be easiest to completely remove Python and start over.

A zombie pythonw.exe process could be a problem. On Windows, use Task Manager to detect and stop one. Sometimes a restart initiated by a program crash or Keyboard Interrupt (control-C) may fail to connect. Dismissing the error box or Restart Shell on the Shell menu may fix a temporary problem.

When IDLE first starts, it attempts to read user configuration files in ~/.idlerc/ (~ is one's home directory). If there is a problem, an error message should be displayed. Leaving aside random disk glitches, this can be prevented by never editing the files by hand, using the configuration dialog, under Options, instead Options. Once it happens, the solution may be to delete one or more of the configuration files.

If IDLE quits with no message, and it was not started from a console, try starting from a console (`python -m idlelib`) and see if a message appears.

运行用户代码

With rare exceptions, the result of executing Python code with IDLE is intended to be the same as executing the same code by the default method, directly with Python in a text-mode system console or terminal window. However, the different interface and operation occasionally affect visible results. For instance, `sys.modules` starts with more entries, and `threading.activeCount()` returns 2 instead of 1.

By default, IDLE runs user code in a separate OS process rather than in the user interface process that runs the shell and editor. In the execution process, it replaces `sys.stdin`, `sys.stdout`, and `sys.stderr` with objects that get input from and send output to the Shell window. The original values stored in `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__` are not touched, but may be `None`.

When Shell has the focus, it controls the keyboard and screen. This is normally transparent, but functions that directly access the keyboard and screen will not work. These include system-specific functions that determine whether a key has been pressed and if so, which.

IDLE's standard stream replacements are not inherited by subprocesses created in the execution process, whether directly by user code or by modules such as multiprocessing. If such subprocess use `input` from sys.stdin or `print` or `write` to sys.stdout or sys.stderr, IDLE should be started in a command line window. The secondary subprocess will then be attached to that window for input and output.

If `sys` is reset by user code, such as with `importlib.reload(sys)`, IDLE's changes are lost and input from the keyboard and output to the screen will not work correctly.

Shell 中的用户输出

When a program outputs text, the result is determined by the corresponding output device. When IDLE executes user code, `sys.stdout` and `sys.stderr` are connected to the display area of IDLE's Shell. Some of its features are inherited from the underlying Tk Text widget. Others are programmed additions. Where it matters, Shell is designed for development rather than production runs.

For instance, Shell never throws away output. A program that sends unlimited output to Shell will eventually fill memory, resulting in a memory error. In contrast, some system text windows only keep the last n lines of output. A Windows console, for instance, keeps a user-settable 1 to 9999 lines, with 300 the default.

Text widgets display a subset of Unicode, the Basic Multilingual Plane (BMP). Which characters get a proper glyph instead of a replacement box depends on the operating system and installed fonts. Newline characters cause following text to appear on a new line, but other control characters are either replaced with a box or deleted. However, `repr()`, which is used for interactive echo of expression values, replaces control characters, some BMP codepoints, and all non-BMP characters with escape codes before they are output.

Normal and error output are generally kept separate (on separate lines) from code input and each other. They each get different highlight colors.

For SyntaxError tracebacks, the normal '^' marking where the error was detected is replaced by coloring the text with an error highlight. When code run from a file causes other exceptions, one may right click on a traceback line to jump to the corresponding line in an IDLE editor. The file will be opened if necessary.

Shell has a special facility for squeezing output lines down to a 'Squeezed text' label. This is done automatically for output over N lines (N = 50 by default). N can be changed in the PyShell section of the General page of the Settings dialog. Output with fewer lines can be squeezed by right clicking on the output. This can be useful lines long enough to slow down scrolling.

Squeezed output is expanded in place by double-clicking the label. It can also be sent to the clipboard or a separate view window by right-clicking the label.

开发 tkinter 应用程序

IDLE is intentionally different from standard Python in order to facilitate development of tkinter programs. Enter `import tkinter as tk; root = tk.Tk()` in standard Python and nothing appears. Enter the same in IDLE and a tk window appears. In standard Python, one must also enter `root.update()` to see the window. IDLE does the equivalent in the background, about 20 times a second, which is about every 50 milleseconds. Next enter `b = tk.Button(root, text='button'); b.pack()`. Again, nothing visibly changes in standard Python until one enters `root.update()`.

Most tkinter programs run `root.mainloop()`, which usually does not return until the tk app is destroyed. If the program is run with `python -i` or from an IDLE editor, a `>>>` shell prompt does not appear until `mainloop()` returns, at which time there is nothing left to interact with.

When running a tkinter program from an IDLE editor, one can comment out the mainloop call. One then gets a shell prompt immediately and can interact with the live application. One just has to remember to re-enable the mainloop call when running in standard Python.

在没有子进程的情况下运行

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the Internet. If firewall software complains anyway, you can ignore it.

If the attempt to make the socket connection fails, Idle will notify you. Such failures are sometimes transient, but if persistent, the problem may be either a firewall blocking the connection or misconfiguration of a particular system. Until the problem is fixed, one can run Idle with the -n command line switch.

If IDLE is started with the -n command line switch it will run in a single process and will not create the subprocess which runs the RPC Python execution server. This can be useful if Python cannot create the subprocess or the RPC socket interface on your platform. However, in this mode user code is not isolated from IDLE itself. Also, the environment is not restarted when Run/Run Module (F5) is selected. If your code has been modified, you must reload() the affected modules and re-import any specific items (e.g. from foo import baz) if the changes are to take effect. For these reasons, it is preferable to run IDLE with the default subprocess if at all possible.

3.4 版后已移除.

25.5.4 帮助和偏好

帮助资源

Help menu entry "IDLE Help" displays a formatted html version of the IDLE chapter of the Library Reference. The result, in a read-only tkinter text window, is close to what one sees in a web browser. Navigate through the text with a mousewheel, the scrollbar, or up and down arrow keys held down. Or click the TOC (Table of Contents) button and select a section header in the opened box.

Help menu entry “Python Docs” opens the extensive sources of help, including tutorials, available at docs.python.org/x.y, where ‘x.y’ is the currently running Python version. If your system has an off-line copy of the docs (this may be an installation option), that will be opened instead.

Selected URLs can be added or removed from the help menu at any time using the General tab of the Configure IDLE dialog .

偏好设定

The font preferences, highlighting, keys, and general preferences can be changed via Configure IDLE on the Option menu. Keys can be user defined; IDLE ships with four built-in key sets. In addition, a user can create a custom key set in the Configure IDLE dialog under the keys tab.

macOS 上的 IDLE

Under System Preferences: Dock, one can set “Prefer tabs when opening documents” to “Always”. This setting is not compatible with the tk/tkinter GUI framework used by IDLE, and it breaks a few IDLE features.

扩展

IDLE contains an extension facility. Preferences for extensions can be changed with the Extensions tab of the preferences dialog. See the beginning of config-extensions.def in the idlelib directory for further information. The only current default extension is zzdummy, an example also used for testing.

25.6 其他图形用户界面（GUI）包

Python 可用的主要跨平台（Windows，Mac OS X，类 Unix）GUI 工具：

参见：

PyGObject PyGObject provides introspection bindings for C libraries using GObject. One of these libraries is the GTK+ 3 widget set. GTK+ comes with many more widgets than Tkinter provides. An online Python GTK+ 3 Tutorial is available.

PyGTK PyGTK 提供了对较旧版本的库 GTK+ 2 的绑定。它使用面向对象接口，比 C 库的抽象层级略高。此外也有对 GNOME 的绑定。请参阅在线 教程。

PyQt PyQt 是一个针对 Qt 工具集通过 **sip** 包装的绑定。Qt 是一个庞大的 C++ GUI 应用开发框架，同时适用于 Unix, Windows 和 Mac OS X。**sip** 是一个用于为 C++ 库生成 Python 类绑定的库，它是针对 Python 特别设计的。

PySide PySide is a newer binding to the Qt toolkit, provided by Nokia. Compared to PyQt, its licensing scheme is friendlier to non-open source applications.

wxPython wxPython 是一个针对 Python 的跨平台 GUI 工具集，它基于热门的 wxWidgets (原名 wxWindows) C++ 工具集进行构建。它为 Windows, Mac OS X 和 Unix 系统上的应用提供了原生的外观效果，在可能的情况下尽量使用各平台的原生可视化部件。(在类 Unix 系统上使用 GTK+)。除了包含庞大的可视化部件集，wxPython 还提供了许多类用于在线文档和上下文感知帮助、打印、HTML 视图、低层级设备上下文绘图、拖放操作、系统剪贴板访问、基于 XML 的资源格式等等，并且包含一个不断增长的用户贡献模块库。

PyGTK, PyQt, and wxPython, all have a modern look and feel and more widgets than Tkinter. In addition, there are many other GUI toolkits for Python, both cross-platform, and platform-specific. See the GUI Programming page in the Python Wiki for a much more complete list, and also for links to documents where the different GUI toolkits are compared.

CHAPTER 26

开发工具

本章中描述的各模块可帮你编写 Python 程序。例如，*pydoc* 模块接受一个模块并根据该模块的内容来生成文档。*doctest* 和*unittest* 这两个模块包含了用于编写单元测试的框架，并可用于自动测试所编写的代码，验证预期的输出是否产生。**2to3** 程序能够将 Python 2.x 源代码翻译成有效的 Python 3.x 源代码。

本章中描述的模块列表是：

26.1 typing —类型标注支持

3.5 新版功能.

源码： Lib/typing.py

注解： The typing module has been included in the standard library on a *provisional basis*. New features might be added and API may change even between minor releases if deemed necessary by the core developers.

此模块支持 **PEP 484** 和 **PEP 526** 指定的类型提示。最基本的支持由*Any*，*Union*，*Tuple*，*Callable*，*TypeVar* 和*Generic* 类型组成。有关完整的规范，请参阅 **PEP 484**。有关类型提示的简单介绍，请参阅 **PEP 483**。

函数接受并返回一个字符串，注释像下面这样：

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

在函数 greeting 中，参数 name 预期是*str* 类型，并且返回*str* 类型。子类型允许作为参数。

26.1.1 类型别名

类型别名通过将类型分配给别名来定义。在这个例子中，`Vector` 和 `List[float]` 将被视为可互换的同义词:

```
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

类型别名可用于简化复杂类型签名。例如:

```
from typing import Dict, Tuple, List

ConnectionOptions = Dict[str, str]
Address = Tuple[str, int]
Server = Tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: List[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
        message: str,
        servers: List[Tuple[Tuple[str, int], Dict[str, str]]]) -> None:
    ...
```

请注意，`None` 作为类型提示是一种特殊情况，并且由 `type(None)` 取代。

26.1.2 NewType

使用*NewType()* 辅助函数创建不同的类型:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

静态类型检查器会将新类型视为它是原始类型的子类。这对于帮助捕捉逻辑错误非常有用:

```
def get_user_name(user_id: UserId) -> str:
    ...

# typechecks
user_a = get_user_name(UserId(42351))

# does not typecheck; an int is not a UserId
user_b = get_user_name(-1)
```

您仍然可以对 `UserId` 类型的变量执行所有的 `int` 支持的操作，但结果将始终为 `int` 类型。这可以让你在需要 `int` 的地方传入 `UserId`，但会阻止你以无效的方式无意中创建 `UserId`:

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

请注意，这些检查仅通过静态类型检查程序强制执行。在运行时，Derived = NewType('Derived', Base) 将 Derived 一个函数，该函数立即返回您传递它的任何参数。这意味着表达式 Derived(some_value) 不会创建一个新的类或引入任何超出常规函数调用的开销。

更确切地说，表达式 some_value is Derived(some_value) 在运行时总是为真。

这也意味着无法创建 Derived 的子类型，因为它是运行时的标识函数，而不是实际的类型:

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not typecheck
class AdminUserId(UserId): pass
```

但是，可以基于’derived’ NewType 创建*NewType()*

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

并且 ProUserId 的类型检查将按预期工作。

有关更多详细信息，请参阅 **PEP 484**。

注解: 回想一下，使用类型别名声明两种类型彼此 等效。Alias = Original 将使静态类型检查对待所有情况下 Alias 完全等同于 Original。当您想简化复杂类型签名时，这很有用。

相反，NewType 声明一种类型是另一种类型的子类型。Derived = NewType('Derived', Original) 将使静态类型检查器将 Derived 当作 Original 的子类，这意味着 Original 类型的值不能用于 Derived 类型的值需要的地方。当您想以最小的运行时间成本防止逻辑错误时，这非常有用。

3.5.2 新版功能.

26.1.3 Callable

期望特定签名的回调函数的框架可以将类型标注为 Callable[[Arg1Type, Arg2Type], ReturnType]。

例如

```
from typing import Callable

def feeder(get_next_item: Callable[[], str]) -> None:
    # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    # Body
```

通过用文字省略号替换类型提示中的参数列表：`Callable[..., ReturnType]`，可以声明可调用的返回类型，而无需指定调用签名。

26.1.4 泛型 (Generic)

由于无法以通用方式静态推断有关保存在容器中的对象的类型信息，因此抽象基类已扩展为支持订阅以表示容器元素的预期类型。

```
from typing import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

泛型可以通过使用 typing 模块中名为*TypeVar*的新工厂进行参数化。

```
from typing import Sequence, TypeVar

T = TypeVar('T')      # Declare type variable

def first(l: Sequence[T]) -> T:   # Generic function
    return l[0]
```

26.1.5 用户定义的泛型类型

用户定义的类可以定义为泛型类。

```
from typing import TypeVar, Generic
from logging import Logger

T = TypeVar('T')

class LoggedVar(Generic[T]):
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

`Generic[T]` 作为基类定义了类 `LoggedVar` 采用单个类型参数 `T`。这也使得 `T` 作为类体内的一个类型有效。

The *Generic* base class uses a metaclass that defines `__getitem__()` so that `LoggedVar[t]` is valid as a type:

```
from typing import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

泛型类型可以有任意数量的类型变量，并且类型变量可能会受到限制:

```
from typing import TypeVar, Generic
...

T = TypeVar('T')
S = TypeVar('S', int, str)

class StrangePair(Generic[T, S]):
    ...
```

Generic 每个参数的类型变量必须是不同的。这是无效的:

```
from typing import TypeVar, Generic
...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

您可以对*Generic* 使用多重继承:

```
from typing import TypeVar, Generic, Sized

T = TypeVar('T')

class LinkedList(Sized, Generic[T]):
    ...
```

从泛型类继承时，某些类型变量可能是固定的:

```
from typing import TypeVar, Mapping

T = TypeVar('T')

class MyDict(Mapping[str, T]):
    ...
```

在这种情况下，MyDict 只有一个参数，T。

在不指定类型参数的情况下使用泛型类别会为每个位置假设*Any*。在下面的例子中，MyIterable 不是泛型，但是隐式继承自 Iterable[Any]:

```
from typing import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
```

用户定义的通用类型别名也受支持。例子:

```
from typing import TypeVar, Iterable, Tuple, Union
S = TypeVar('S')
Response = Union[Iterable[S], int]

# Return type here is same as Union[Iterable[str], int]
def response(query: str) -> Response[str]:
    ...

T = TypeVar('T', int, float, complex)
Vec = Iterable[Tuple[T, T]]

def inproduct(v: Vec[T]) -> T: # Same as Iterable[Tuple[T, T]]
    return sum(x*y for x, y in v)
```

The metaclass used by *Generic* is a subclass of *abc.ABCMeta*. A generic class can be an ABC by including abstract methods or properties, and generic classes can also have ABCs as base classes without a metaclass conflict. Generic metaclasses are not supported. The outcome of parameterizing generics is cached, and most types in the typing module are hashable and comparable for equality.

26.1.6 Any 类型

Any 是一种特殊的类型。静态类型检查器将所有类型视为与*Any* 兼容，反之亦然，*Any* 也与所有类型相兼容。

这意味着可对类型为*Any* 的值执行任何操作或方法调用，并将其赋值给任何变量:

```
from typing import Any

a = None    # type: Any
a = []      # OK
a = 2       # OK

s = ''      # type: str
s = a       # OK

def foo(item: Any) -> int:
    # Typechecks; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

需要注意的是，将*Any* 类型的值赋值给另一个更具体的类型时，Python 不会执行类型检查。例如，当把 a 赋值给 s 时，即使 s 被声明为*str* 类型，在运行时接收到的是*int* 值，静态类型检查器也不会报错。

此外，所有返回值无类型或形参无类型的函数将隐式地默认使用*Any* 类型:

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

当需要混用动态类型和静态类型的代码时，上述行为可以让*Any* 被用作 *应急出口*。

Any 和*object* 的行为对比。与*Any* 相似，所有的类型都是*object* 的子类型。然而不同于*Any*，反之并不成立：*object* 不是其他所有类型的子类型。

这意味着当一个值的类型是*object* 的时候，类型检查器会拒绝对它的几乎所有的操作。把它赋值给一个指定了类型的变量（或者当作返回值）是一个类型错误。比如说：

```
def hash_a(item: object) -> int:
    # Fails; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Typechecks
    item.magic()
    ...

# Typechecks, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Typechecks, since Any is compatible with all types
hash_b(42)
hash_b("foo")
```

使用*object* 示意一个值可以类型安全地兼容任何类型。使用*Any* 示意一个值地类型是动态定义的。

26.1.7 类, 函数和修饰器.

这个模块定义了如下的类, 模块和修饰器.

class typing.**TypeVar**

类型变量

用法:

```
T = TypeVar('T')  # Can be anything
A = TypeVar('A', str, bytes)  # Must be str or bytes
```

Type variables exist primarily for the benefit of static type checkers. They serve as the parameters for generic types as well as for generic function definitions. See class Generic for more information on generic types. Generic functions work as follows:

```
def repeat(x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n

def longest(x: A, y: A) -> A:
    """Return the longest of two strings."""
    return x if len(x) >= len(y) else y
```

本质上，后例的签名重载了 `(str, str) -> str` 与 `(bytes, bytes) -> bytes`。注意，参数是*str* 子类的实例时，返回类型仍是纯*str*。

`isinstance(x, T)` 会在运行时抛出*TypeError* 异常。一般地说，*isinstance()* 和*issubclass()* 不应该和类型一起使用。

通过 `covariant=True` 或 `contravariant=True` 可以把类型变量标记为协变量或逆变量。详见 **PEP 484**。默认情况下，类型变量是不变量。类型变量还可以用 `bound=<type>` 指定上限。这里的意

思是，（显式或隐式地）取代类型变量的实际类型必须是限定类型的子类，详见 **PEP 484**。

class typing.**Generic**
用于泛型类型的抽象基类。

泛型类型一般通过继承含一个或多个类型变量的类实例进行声明。例如，泛型映射类型定义如下：

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

这个类之后可以被这样用：

```
X = TypeVar('X')
Y = TypeVar('Y')

def lookup_name(mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

class typing.**Type**(*Generic[CT_co]*)
一个注解为 C 的变量可以接受一个类型为 C 的值。相对地，一个注解为 Type[C] 的变量可以接受本身为类的值——更精确地说它接受 C 的 *类对象*，例如：

```
a = 3         # Has type 'int'
b = int       # Has type 'Type[int]'
c = type(a)   # Also has type 'Type[int]'
```

注意 Type[C] 是协变的：

```
class User: ...
class BasicUser(User): ...
class ProUser(User): ...
class TeamUser(User): ...

# Accepts User, BasicUser, ProUser, TeamUser, ...
def make_new_user(user_class: Type[User]) -> User:
    # ...
    return user_class()
```

Type[C] 是协变的这一事实暗示了任何 C 的子类应当实现与 C 相同的构造器签名和类方法签名。类型检查器应当标记违反的情况，但应当也允许子类中调用构造器符合指示的基类。类型检查器被要求如何处理这种情况可能会在 **PEP 484** 将来的版本中改变。

Type 合法的参数仅有类、*Any* 、*类型变量* 以及上述类型的联合类型。例如：

```
def new_non_team_user(user_class: Type[Union[BaseUser, ProUser]]): ...
```

Type[Any] 等价于 Type，因此继而等价于 type，它是 Python 的元类层级的根部。

3.5.2 新版功能.

class typing.**Iterable**(*Generic[T_co]*)
collections.abc.Iterable 的泛型版本。

class typing.**Iterator**(*Iterable[T_co]*)
collections.abc.Iterator 的泛型版本。

class typing.**Reversible**(*Iterable[T_co]*)
collections.abc.Reversible 的泛型版本。

class typing.**SupportsInt**
含抽象方法 __int__ 的抽象基类。

class typing.**SupportsFloat**
含抽象方法 __float__ 的抽象基类。

class typing.**SupportsComplex**
含抽象方法 __complex__ 的抽象基类。

class typing.**SupportsBytes**
含抽象方法 __bytes__ 的抽象基类（ABC）

class typing.**SupportsAbs**
含抽象方法 __abs__ 的抽象基类（ABC）是其返回类型里的协变量。

class typing.**SupportsRound**
含抽象方法 __round__ 的抽象基类，是其返回类型的协变量。

class typing.**Container**(*Generic[T_co]*)
collections.abc.Container 的泛型版本。

class typing.**Hashable**
collections.abc.Hashable 的别名。

class typing.**Sized**
collections.abc.Sized 的别名。

class typing.**Collection**(*Sized, Iterable[T_co], Container[T_co]*)
collections.abc.Collection 的泛型版本。

3.6 新版功能.

class typing.**AbstractSet**(*Sized, Collection[T_co]*)
collections.abc.Set 的泛型版本。

class typing.**MutableSet**(*AbstractSet[T]*)
collections.abc.MutableSet 的泛型版本。

class typing.**Mapping**(*Sized, Collection[KT], Generic[VT_co]*)
A generic version of collections.abc.Mapping.

class typing.**MutableMapping**(*Mapping[KT, VT]*)
collections.abc.MutableMapping 的泛型版本。

class typing.**Sequence**(*Reversible[T_co], Collection[T_co]*)
collections.abc.Sequence 的泛型版本。

class typing.**MutableSequence**(*Sequence[T]*)
collections.abc.MutableSequence 的泛型版本。

class typing.**ByteString**(*Sequence[int]*)
collections.abc.ByteString 的泛型版本。

This type represents the types bytes, bytearray, and memoryview.

作为该类型的简称，bytes 可用于标注上述任意类型的参数。

class typing.**Deque**(*deque, MutableSequence[T]*)
collections.deque 的泛型版本。

3.6.1 新版功能.

class typing.**List**(*list, MutableSequence[T]*)
Generic version of *list*. Useful for annotating return types. To annotate arguments it is preferred to use abstract collection types such as *Mapping*, *Sequence*, or *AbstractSet*.

这个类型可以这样用:

```
T = TypeVar('T', int, float)

def vec2(x: T, y: T) -> List[T]:
    return [x, y]

def keep_positives(vector: Sequence[T]) -> List[T]:
    return [item for item in vector if item > 0]
```

class typing.**Set**(*set, MutableSet[T]*)
A generic version of *builtins.set*.

class typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)
builtins.frozenset 的泛型版本。

class typing.**MappingView**(*Sized, Iterable[T_co]*)
collections.abc.MappingView 的泛型版本。

class typing.**KeysView**(*MappingView[KT_co], AbstractSet[KT_co]*)
collections.abc.KeysView 的泛型版本。

class typing.**ItemsView**(*MappingView, Generic[KT_co, VT_co]*)
collections.abc.ItemsView 的泛型版本。

class typing.**ValuesView**(*MappingView[VT_co]*)
collections.abc.ValuesView 的泛型版本。

class typing.**Awaitable**(*Generic[T_co]*)
collections.abc.Awaitable 的泛型版本。

class typing.**Coroutine**(*Awaitable[V_co], Generic[T_co T_contra, V_co]*)
collections.abc.Coroutine 的泛型版本。类型变量的的差异和顺序与*Generator* 的内容相对应，例如:

```
from typing import List, Coroutine
c = None # type: Coroutine[List[str], str, int]
...
x = c.send('hi') # type: List[str]
async def bar() -> None:
    x = await c # type: int
```

class typing.**AsyncIterable**(*Generic[T_co]*)
collections.abc.AsyncIterable 的泛型版本。

class typing.**AsyncIterator**(*AsyncIterable[T_co]*)
collections.abc.AsyncIterator 的泛型版本。

class typing.**ContextManager**(*Generic[T_co]*)
contextlib.AbstractContextManager 的泛型版本。

3.6 新版功能.

class typing.**AsyncContextManager**(*Generic[T_co]*)
An ABC with async abstract `__aenter__()` and `__aexit__()` methods.

3.6 新版功能.

class typing.**Dict**(*dict, MutableMapping[KT, VT]*)
A generic version of *dict*. The usage of this type is as follows:

```
def get_position_in_index(word_list: Dict[str, int], word: str) -> int:
    return word_list[word]
```

class typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)
collections.defaultdict 的泛型版本。

3.5.2 新版功能.

class typing.**Counter**(*collections.Counter, Dict[T, int]*)
collections.Counter 的泛型版本。

3.6.1 新版功能.

class typing.**ChainMap**(*collections.ChainMap, MutableMapping[KT, VT]*)
collections.ChainMap 的泛型版本。

3.6.1 新版功能.

class typing.**Generator**(*Iterator[T_co], Generic[T_co, T_contra, V_co]*)
生成器可以由泛型类型 `Generator[YieldType, SendType, ReturnType]` 注解。例如:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

注意，与 typing 模块里的其他泛型不同，*Generator* 的 “SendType“ 的操作是逆变的，不是协变，也是不变。

如果生成器只生成值，可将 `SendType` 与 `ReturnType` 设为 `None`:

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

另外，把生成器注解为返回类型

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

class typing.**AsyncGenerator**(*AsyncIterator[T_co], Generic[T_co, T_contra]*)
异步生成器可由泛型类型 `AsyncGenerator[YieldType, SendType]` 注解。例如:

```
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

与常规生成器不同，异步生成器不能返回值，因此没有 `ReturnType` 类型参数。与*Generator* 类似，`SendType` 也属于逆变行为。

如果生成器只产生值，可将 `SendType` 设置为 `None`:

```
async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

此外，可用 AsyncIterable[YieldType] 或 AsyncIterator[YieldType] 注解生成器的类型：

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

3.5.4 新版功能.

class typing.**Text**

Text 是 str 的别名。提供了对 Python 2 代码的向下兼容：Python 2 中，Text 是 “unicode“ 的别名。

用

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

3.5.2 新版功能.

class typing.**IO**
class typing.**TextIO**
class typing.**BinaryIO**

泛型类型 IO[AnyStr] 及其子类 TextIO(IO[str]) 与 BinaryIO(IO[bytes]) 表示 I/O 流的类型，例如*open()* 所返回的对象。

class typing.**Pattern**
class typing.**Match**

这些类型对应的是从*re.compile()* 和*re.match()* 返回的类型。这些类型（及相应的函数）是 AnyStr 中的泛型并可通过编写 Pattern[str], Pattern[bytes], Match[str] 或 Match[bytes] 来具体指定。

class typing.**NamedTuple**

Typed version of namedtuple.

用法:

```
class Employee(NamedTuple):
    name: str
    id: int
```

这相当于:

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

为字段提供默认值，要在类体内赋值：

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

带默认值的字段必须在不带默认值的字段后面。

The resulting class has two extra attributes: `_field_types`, giving a dict mapping field names to types, and `_field_defaults`, a dict mapping field names to default values. (The field names are in the `_fields` attribute, which is part of the namedtuple API.)

`NamedTuple` 子类也支持文档字符串与方法：

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

反向兼容用法：

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

在 3.6 版更改: 添加了对 **PEP 526** 中变量注解句法的支持。

在 3.6.1 版更改: 添加了对默认值、方法、文档字符串的支持。

`typing.`**`NewType`**(*typ*)

A helper function to indicate a distinct types to a typechecker, see *NewType*. At runtime it returns a function that returns its argument. Usage:

```
UserId = NewType('UserId', int)
first_user = UserId(1)
```

3.5.2 新版功能.

`typing.`**`cast`**(*typ*, *val*)

把值强制转换为类型。

不变更返回值。对类型检查器来说，这代表了返回值具有指定的类型，但在运行时，故意不做任何检查（目的是让该检查速度尽量快）。

`typing.`**`get_type_hints`**(*obj*[, *globals*[, *locals*]])

返回一个字典，字典内含有函数、方法、模块或类对象的类型提示。

一般情况下，与 `obj.__annotations__` 相同。此外，可通过在 `globals` 与 `locals` 命名空间里进行评估，以此来处理编码为字符串字面量的前向引用。如有需要，在默认值设置为 `None`` 时，可为函数或方法注解添加 ``Optional[t]`。对于类 `C`，则返回一个由所有 `__annotations__` 与 `C.__mro__` 逆序合并而成的字典。

`@typing.`**`overload`**

The `@overload` decorator allows describing functions and methods that support multiple different combinations of argument types. A series of `@overload`-decorated definitions must be followed by exactly one non-`@overload`-decorated definition (for the same function/method). The `@overload`-decorated definitions are for the benefit of the type checker only, since they will be overwritten by the non-`@overload`-decorated definition, while the latter is used at runtime but should be ignored by a type checker. At runtime, calling a `@overload`-decorated function directly will raise `NotImplementedError`. An example of overload that gives a more precise type than can be expressed using a union or a type variable:

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> Tuple[int, str]:
```

(下页继续)

(续上页)

```
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    <actual implementation>
```

详见 **PEP 484**，与其他类型语义进行对比。

`@typing.`**`no_type_check`**
用于指明标注不是类型提示的装饰器。

此*decorator* 装饰器生效于类或函数上。如果作用于类上的话，它会递归地作用于这个类的所定义的所有方法上（但是对于超类或子类所定义的方法不会生效）。

此方法会就地地修改函数。

`@typing.`**`no_type_check_decorator`**
使其它装饰器起到*no_type_check()* 效果的装饰器。

本装饰器用*no_type_check()* 里的装饰函数打包其他装饰器。

`typing.`**`Any`**
特殊类型，表明类型没有任何限制。

- 每一个类型都对*Any* 兼容。
- *Any* 对每一个类型都兼容。

`typing.`**`NoReturn`**
标记一个函数没有返回值的特殊类型。比如说:

```
from typing import NoReturn

def stop() -> NoReturn:
    raise RuntimeError('no way')
```

3.6.5 新版功能.

`typing.`**`Union`**
联合类型；`Union[X, Y]` 意味着：要不是 X，要不是 Y。

使用形如 `Union[int, str]` 的形式来定义一个联合类型。细节如下:

- 参数必须是类型，而且必须至少有一个参数。
- 联合类型的联合类型会被展开打平，比如:

```
Union[Union[int, str], float] == Union[int, str, float]
```

- 仅有一个参数的联合类型会坍缩成参数自身，比如:

```
Union[int] == int  # The constructor actually returns int
```

- 多余的参数会被跳过，比如:

```
Union[int, str, int] == Union[int, str]
```

- 在比较联合类型的时候，参数顺序会被忽略，比如:

```
Union[int, str] == Union[str, int]
```

- When a class and its subclass are present, the latter is skipped, e.g.:

```
Union[int, object] == object
```

- 你不能继承或者实例化一个联合类型。
- 你不能写成 `Union[X][Y]` 。
- 你可以使用 `Optional[X]` 作为 `Union[X, None]` 的缩写。

typing.**Optional**
可选类型。

`Optional[X]` 等价于 `Union[X, None]` 。

请注意，这与可选参数并非相同的概念。可选参数是一个具有默认值的参数。可选参数的类型注解并不因为它是可选的就需要 `Optional` 限定符。例如：

```
def foo(arg: int = 0) -> None:
    ...
```

另一方面，如果允许显式地传递值 `None` ，使用 `Optional` 也是正当的，无论该参数是否是可选的。例如：

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

typing.**Tuple**
Tuple type; `Tuple[X, Y]` is the type of a tuple of two items with the first item of type X and the second of type Y.

举例：`Tuple[T1, T2]` 是一个二元组，类型分别为 T1 和 T2。`Tuple[int, float, str]` 是一个由整数、浮点数和字符串组成的三元组。

为表达一个同类型元素的变长元组，使用省略号字面量，如 `Tuple[int, ...]` 。单独的一个*Tuple* 等价于 `Tuple[Any, ...]`，进而等价于*tuple* 。

typing.**Callable**
可调用类型；`Callable[[int], str]` 是一个函数，接受一个 int 参数，返回一个 str 。

下标值的语法必须恰为两个值：参数列表和返回类型。参数列表必须是一个类型和省略号组成的列表；返回值必须是单一一个类型。

不存在语法来表示可选的或关键词参数，这类函数类型罕见用于回调函数。`Callable[..., ReturnType]` （使用字面省略号）能被用于提示一个可调用对象，接受任意数量的参数并且返回 `ReturnType`。单独的*Callable* 等价于 `Callable[..., Any]` ，并且进而等价于*collections.abc.Callable* 。

typing.**ClassVar**
特殊的类型构造器，用以标记类变量。

在 **PEP 526** 中被引入，ClassVar 包裹起来的变量注解指示了给定属性预期用于类变量，并且不应在类的实例上被设置。用法：

```
class Starship:
    stats: ClassVar[Dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

ClassVar 仅接受类型，并且不能被再次下标。

ClassVar 本身并不是一个类，并且不应与*isinstance()* or *issubclass()* 一起使用。*ClassVar* 并不改变 Python 运行时行为，但它可以被用于第三方类型检查器。例如，某个类型检查器可能会标记以下代码为错误的：

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}     # This is OK
```

3.5.3 新版功能.

typing.**AnyStr**

AnyStr 类型变量的定义为 AnyStr = TypeVar('AnyStr', str, bytes)。

这里指的是，它可以接受任意同类字符串，但不支持混用不同类别的字符串。例如：

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat(u"foo", u"bar")  # Ok, output has type 'unicode'
concat(b"foo", b"bar")  # Ok, output has type 'bytes'
concat(u"foo", b"bar")  # Error, cannot mix unicode and bytes
```

typing.**TYPE_CHECKING**

被第三方静态类型检查器假定为 True 的特殊常量。在运行时为 False。用法如下：

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

Note that the first type annotation must be enclosed in quotes, making it a "forward reference", to hide the expensive_mod reference from the interpreter runtime. Type annotations for local variables are not evaluated, so the second annotation does not need to be enclosed in quotes.

3.5.2 新版功能.

26.2 pydoc —文档生成器和在线帮助系统

源代码: Lib/pydoc.py

The *pydoc* module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.

对于模块、类、函数和方法，显示的文档内容取自文档字符串（即 __doc__ 属性），并会递归地从其带文档的成员中获取。如果没有文档字符串，*pydoc* 会尝试从类、函数或方法定义上方，或是模块顶部的注释行段落获取 (参见*inspect.getcomments()*).

内置函数*help()* 会发起调用交互式解释器的在线帮助系统，该系统使用*pydoc* 在终端上生成文本形式的文档内容。同样的文本文档也可以在 Python 解释器以外通过在操作系统的命令提示符下以脚本方式运行 **pydoc** 来查看。例如，运行

```
pydoc sys
```

在终端提示符下将通过*sys* 模块显示文档内容，其样式类似于 Unix **man** 命令所显示的指南页面。**pydoc** 的参数可以为函数、模块、包，或带点号的对模块中的类、方法或函数以及包中的模块的引用。如果传给 **pydoc** 的参数像是一个路径（即包含所在操作系统的路径分隔符，例如 Unix 的正斜杠），并且其指向一个现有的 Python 源文件，则会为该文件生成文档内容。

注解： 为了找到对象及其文档内容，*pydoc* 会导入文档所在的模块。因此，任何模块层级的代码都将被执行。请使用 `if __name__ == '__main__':` 语句来确保一个文件的特定代码仅在作为脚本被发起调用时执行而不是在被导入时执行。

当打印输出到控制台时，**pydoc** 会尝试对输出进行分页以方便阅读。如果设置了 `PAGER` 环境变量，**pydoc** 将使用该变量值作为分页程序。

在参数前指定 `-w` 旗标将把 HTML 文档写入到当前目录下的一个文件中，而不是在控制台中显示文本。

在参数前指定 `-k` 旗标将在全部可用模块的提要行中搜索参数所给定的关键字，具体方式同样类似于 Unix **man** 命令。模块的提要行就是其文档字符串的第一行。

你还可以使用 **pydoc** 在本机上启动一个 HTTP 服务，这将向来访的 Web 浏览器提供文档服务。**pydoc -p 1234** 将在 1234 端口上启动 HTTP 服务，允许你在你喜欢的 Web 服务器中通过 `http://localhost:1234/` 浏览文档内容。指定 `0` 作为端口号将会任意选择一个未使用的端口。

pydoc -b 将启动服务并额外打开一个 Web 浏览器访问模块索引页。所发布的每个页面顶端都带有导航栏，你可以点击 *Get* 获取特定条目的帮助，点击 *Search* 在所有模块的提要行中搜索特定关键词，或是点击 *Module index*, *Topics* 和 *Keywords* 前往相应的页面。

当 **pydoc** 生成文档内容时，它会使用当前环境和路径来定位模块。因此，发起调用 **pydoc spam** 得到的文档版本会与你启动 Python 解释器并输入 `import spam` 时得到的模块版本完全相同。

核心模块的模块文档位置对应于 `https://docs.python.org/X.Y/library/` 其中 `X` 和 `Y` 是 Python 解释器的主要版本号和小版本号。这可通过设置 `PYTHONDOCS` 环境变量来重载为指向不同的 URL 或包含 Library Reference Manual 页面的本地目录。

在 3.2 版更改: 添加 `-b` 选项。

在 3.3 版更改: 命令行选项 `-g` 已经移除。

在 3.4 版更改: *pydoc* 现在会使用*inspect.signature()* 而非*inspect.getfullargspec()* 来从可调用对象中提取签名信息。

26.3 doctest —测试交互性的 Python 示例

** 源代码 ** Lib/doctest.py

---

doctest 模块寻找像 Python 交互式代码的文本，然后执行这些代码来确保它们的确就像展示的那样正确运行，有许多方法来使用 doctest：

- 通过验证所有交互式示例仍然按照记录的方式工作，以此来检查模块的文档字符串是否是最新的。
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation".

下面是一个小却完整的示例模块:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

如果你直接在命令行里运行 `example.py`，*doctest* 将发挥它的作用。

```
$ python example.py
$
```

There's no output! That's normal, and it means all the examples worked. Pass `-v` to the script, and *doctest* prints a detailed log of what it's trying, and prints a summary at the end:

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

以此类推，最终以：

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 tests in __main__
   8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$
```

That's all you need to know to start making productive use of *doctest*! Jump in. The following sections provide full details. Note that there are many examples of doctests in the standard Python test suite and libraries. Especially useful examples can be found in the standard test file `Lib/test/test_doctest.py`.

26.3.1 简单用法：检查 Docstrings 中的示例

开始使用 doctest 的最简单方法（但不一定是你将继续这样做的方式）是结束每个模块 `M` 使用：

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

doctest then examines docstrings in module `M`.

Running the module as a script causes the examples in the docstrings to get executed and verified:

```
python M.py
```

This won't display anything unless an example fails, in which case the failing example(s) and the cause(s) of the failure(s) are printed to stdout, and the final line of output is `***Test Failed*** N failures.`, where *N* is the number of examples that failed.

Run it with the -v switch instead:

```
python M.py -v
```

and a detailed report of all examples tried is printed to standard output, along with assorted summaries at the end.

You can force verbose mode by passing verbose=True to *testmod()*, or prohibit it by passing verbose=False. In either of those cases, sys.argv is not examined by *testmod()* (so passing -v or not has no effect).

There is also a command line shortcut for running *testmod()*. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the module name(s) on the command line:

```
python -m doctest -v example.py
```

This will import example.py as a standalone module and run *testmod()* on it. Note that this may not work correctly if the file is part of a package and imports other submodules from that package.

For more information on *testmod()*, see section *Basic API*.

26.3.2 Simple Usage: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the *testfile()* function:

```
import doctest
doctest.testfile("example.txt")
```

That short script executes and verifies any interactive Python examples contained in the file example.txt. The file content is treated as if it were a single giant docstring; the file doesn't need to contain a Python program! For example, perhaps example.txt contains this:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

Running doctest.testfile("example.txt") then finds the error in this documentation:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

As with *testmod()*, *testfile()* won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to stdout, using the same format as *testmod()*.

By default, *testfile()* looks for files in the calling module's directory. See section *Basic API* for a description of the optional arguments that can be used to tell it to look for files in other locations.

Like *testmod()*, *testfile()*'s verbosity can be set with the `-v` command-line switch or with the optional keyword argument *verbose*.

There is also a command line shortcut for running *testfile()*. You can instruct the Python interpreter to run the doctest module directly from the standard library and pass the file name(s) on the command line:

```
python -m doctest -v example.txt
```

Because the file name does not end with `.py`, *doctest* infers that it must be run with *testfile()*, not *testmod()*.

For more information on *testfile()*, see section *Basic API*.

26.3.3 How It Works

This section examines in detail how doctest works: which docstrings it looks at, how it finds interactive examples, what execution context it uses, how it handles exceptions, and how option flags can be used to control its behavior. This is the information that you need to know to write doctest examples; for information about actually running doctest on these examples, see the following sections.

Which Docstrings Are Examined?

The module docstring, and all function, class and method docstrings are searched. Objects imported into the module are not searched.

In addition, if `M.__test__` exists and "is true", it must be a dict, and each entry maps a (string) name to a function object, class object, or string. Function and class object docstrings found from `M.__test__` are searched, and strings are treated as if they were docstrings. In output, a key `K` in `M.__test__` appears with name

```
<name of M>.__test__.K
```

Any classes found are recursively searched similarly, to test docstrings in their contained methods and nested classes.

CPython implementation detail: Prior to version 3.4, extension modules written in C were not fully searched by doctest.

How are Docstring Examples Recognized?

In most cases a copy-and-paste of an interactive console session works fine, but doctest isn't trying to do an exact emulation of any specific Python shell.

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
```

(下页继续)

(续上页)

```
...
no
NO
NO!!!
>>>
```

Any expected output must immediately follow the final `'>>> '` or `'... '` line containing the code, and the expected output (if any) extends to the next `'>>> '` or all-whitespace line.

The fine print:

- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put `<BLANKLINE>` in your doctest example each place a blank line is expected.
- All hard tab characters are expanded to spaces, using 8-column tab stops. Tabs in output generated by the tested code are not modified. Because any hard tabs in the sample output *are* expanded, this means that if the code output includes hard tabs, the only way the doctest can pass is if the *NORMALIZE_WHITESPACE* option or *directive* is in effect. Alternatively, the test can be rewritten to capture the output and compare it to an expected value as part of the test. This handling of tabs in the source was arrived at through trial and error, and has proven to be the least error prone way of handling them. It is possible to use a different algorithm for handling tabs by writing a custom *DocTestParser* class.
- Output to stdout is captured, but not output to stderr (exception tracebacks are captured via a different means).
- If you continue a line via backslashing in an interactive session, or for any other reason use a backslash, you should use a raw docstring, which will preserve your backslashes exactly as you type them:

  ```
  >>> def f(x):
  ...     r'''Backslashes in a raw docstring: m\n'''
  >>> print(f.__doc__)
  Backslashes in a raw docstring: m\n
  ```

  Otherwise, the backslash will be interpreted as part of the string. For example, the `\n` above would be interpreted as a newline character. Alternatively, you can double each backslash in the doctest version (and not use a raw string):

  ```
  >>> def f(x):
  ...     '''Backslashes in a raw docstring: m\\n'''
  >>> print(f.__doc__)
  Backslashes in a raw docstring: m\n
  ```

- The starting column doesn't matter:

  ```
  >>> assert "Easy!"
        >>> import math
            >>> math.floor(1.9)
            1
  ```

  and as many leading whitespace characters are stripped from the expected output as appeared in the initial `'>>> '` line that started the example.

What's the Execution Context?

By default, each time *doctest* finds a docstring to test, it uses a *shallow copy* of M's globals, so that running tests doesn't change the module's real globals, and so that one test in M can't leave behind crumbs that accidentally allow another test to work. This means examples can freely use any names defined at top-level in M, and names defined earlier in the docstring being run. Examples cannot see names defined in other docstrings.

You can force use of your own dict as the execution context by passing `globs=your_dict` to *testmod()* or *testfile()* instead.

What About Exceptions?

No problem, provided that the traceback is the only output produced by the example: just paste in the traceback.[1] Since tracebacks contain details that are likely to change rapidly (for example, exact file paths and line numbers), this is one case where doctest works hard to be flexible in what it accepts.

Simple example:

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

That doctest succeeds if *ValueError* is raised, with the `list.remove(x): x not in list` detail as shown.

The expected output for an exception must start with a traceback header, which may be either of the following two lines, indented the same as the first line of the example:

```
Traceback (most recent call last):
Traceback (innermost last):
```

The traceback header is followed by an optional traceback stack, whose contents are ignored by doctest. The traceback stack is typically omitted, or copied verbatim from an interactive session.

The traceback stack is followed by the most interesting part: the line(s) containing the exception type and detail. This is usually the last line of a traceback, but can extend across multiple lines if the exception has a multi-line detail:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

The last three lines (starting with *ValueError*) are compared against the exception's type and detail, and the rest are ignored.

Best practice is to omit the traceback stack, unless it adds significant documentation value to the example. So the last example is probably better as:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
```

(下页继续)

[1] Examples containing both expected output and an exception are not supported. Trying to guess where one ends and the other begins is too error-prone, and that also makes for a confusing test.

(续上页)

```
    line
detail
```

Note that tracebacks are treated very specially. In particular, in the rewritten example, the use of `...` is independent of doctest's *ELLIPSIS* option. The ellipsis in that example could be left out, or could just as well be three (or three hundred) commas or digits, or an indented transcript of a Monty Python skit.

Some details you should read once, but won't need to remember:

- Doctest can't guess whether your expected output came from an exception traceback or from ordinary printing. So, e.g., an example that expects `ValueError: 42 is prime` will pass whether *ValueError* is actually raised or if the example merely prints that traceback text. In practice, ordinary output rarely begins with a traceback header line, so this doesn't create real problems.
- Each line of the traceback stack (if present) must be indented further than the first line of the example, *or* start with a non-alphanumeric character. The first line following the traceback header indented the same and starting with an alphanumeric is taken to be the start of the exception detail. Of course this does the right thing for genuine tracebacks.
- When the *IGNORE_EXCEPTION_DETAIL* doctest option is specified, everything following the leftmost colon and any module information in the exception name is ignored.
- The interactive shell omits the traceback header line for some *SyntaxError*s. But doctest uses the traceback header line to distinguish exceptions from non-exceptions. So in the rare case where you need to test a *SyntaxError* that omits the traceback header, you will need to manually add the traceback header line to your test example.
- For some *SyntaxError*s, Python displays the character position of the syntax error, using a ^ marker:

  ```
  >>> 1 1
    File "<stdin>", line 1
      1 1
        ^
  SyntaxError: invalid syntax
  ```

  Since the lines showing the position of the error come before the exception type and detail, they are not checked by doctest. For example, the following test would pass, even though it puts the ^ marker in the wrong location:

  ```
  >>> 1 1
    File "<stdin>", line 1
      1 1
      ^
  SyntaxError: invalid syntax
  ```

Option Flags

A number of option flags control various aspects of doctest's behavior. Symbolic names for the flags are supplied as module constants, which can be bitwise ORed together and passed to various functions. The names can also be used in *doctest directives*, and may be passed to the doctest command line interface via the `-o` option.

3.4 新版功能: The `-o` command line option.

The first group of options define test semantics, controlling aspects of how doctest decides whether actual output matches an example's expected output:

`doctest.`**`DONT_ACCEPT_TRUE_FOR_1`**
: By default, if an expected output block contains just `1`, an actual output block containing just `1` or just `True` is

considered to be a match, and similarly for `0` versus `False`. When *DONT_ACCEPT_TRUE_FOR_1* is specified, neither substitution is allowed. The default behavior caters to that Python changed the return type of many functions from integer to boolean; doctests expecting "little integer" output still work in these cases. This option will probably go away, but not for several years.

doctest.**DONT_ACCEPT_BLANKLINE**
By default, if an expected output block contains a line containing only the string `<BLANKLINE>`, then that line will match a blank line in the actual output. Because a genuinely blank line delimits the expected output, this is the only way to communicate that a blank line is expected. When *DONT_ACCEPT_BLANKLINE* is specified, this substitution is not allowed.

doctest.**NORMALIZE_WHITESPACE**
When specified, all sequences of whitespace (blanks and newlines) are treated as equal. Any sequence of whitespace within the expected output will match any sequence of whitespace within the actual output. By default, whitespace must match exactly. *NORMALIZE_WHITESPACE* is especially useful when a line of expected output is very long, and you want to wrap it across multiple lines in your source.

doctest.**ELLIPSIS**
When specified, an ellipsis marker (`...`) in the expected output can match any substring in the actual output. This includes substrings that span line boundaries, and empty substrings, so it's best to keep usage of this simple. Complicated uses can lead to the same kinds of "oops, it matched too much!" surprises that `.*` is prone to in regular expressions.

doctest.**IGNORE_EXCEPTION_DETAIL**
When specified, an example that expects an exception passes if an exception of the expected type is raised, even if the exception detail does not match. For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail, e.g., if *TypeError* is raised.

It will also ignore the module name used in Python 3 doctest reports. Hence both of these variations will work with the flag specified, regardless of whether the test is run under Python 2.7 or Python 3.2 (or later versions):

```
>>> raise CustomError('message')
Traceback (most recent call last):
CustomError: message

>>> raise CustomError('message')
Traceback (most recent call last):
my_module.CustomError: message
```

Note that *ELLIPSIS* can also be used to ignore the details of the exception message, but such a test may still fail based on whether or not the module details are printed as part of the exception name. Using *IGNORE_EXCEPTION_DETAIL* and the details from Python 2.3 is also the only clear way to write a doctest that doesn't care about the exception detail yet continues to pass under Python 2.3 or earlier (those releases do not support *doctest directives* and ignore them as irrelevant comments). For example:

```
>>> (1, 2)[3] = 'moo'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object doesn't support item assignment
```

passes under Python 2.3 and later Python versions with the flag specified, even though the detail changed in Python 2.4 to say "does not" instead of "doesn't".

在 3.2 版更改: *IGNORE_EXCEPTION_DETAIL* now also ignores any information relating to the module containing the exception under test.

doctest.**SKIP**
When specified, do not run the example at all. This can be useful in contexts where doctest examples serve as both documentation and test cases, and an example should be included for documentation purposes, but should not be

checked. E.g., the example's output might be random; or the example might depend on resources which would be unavailable to the test driver.

The SKIP flag can also be used for temporarily "commenting out" examples.

doctest.**COMPARISON_FLAGS**
A bitmask or'ing together all the comparison flags above.

The second group of options controls how test failures are reported:

doctest.**REPORT_UDIFF**
When specified, failures that involve multi-line expected and actual outputs are displayed using a unified diff.

doctest.**REPORT_CDIFF**
When specified, failures that involve multi-line expected and actual outputs will be displayed using a context diff.

doctest.**REPORT_NDIFF**
When specified, differences are computed by `difflib.Differ`, using the same algorithm as the popular `ndiff.py` utility. This is the only method that marks differences within lines as well as across lines. For example, if a line of expected output contains digit `1` where actual output contains letter `l`, a line is inserted with a caret marking the mismatching column positions.

doctest.**REPORT_ONLY_FIRST_FAILURE**
When specified, display the first failing example in each doctest, but suppress output for all remaining examples. This will prevent doctest from reporting correct examples that break because of earlier failures; but it might also hide incorrect examples that fail independently of the first failure. When *REPORT_ONLY_FIRST_FAILURE* is specified, the remaining examples are still run, and still count towards the total number of failures reported; only the output is suppressed.

doctest.**FAIL_FAST**
When specified, exit after the first failing example and don't attempt to run the remaining examples. Thus, the number of failures reported will be at most 1. This flag may be useful during debugging, since examples after the first failure won't even produce debugging output.

The doctest command line accepts the option `-f` as a shorthand for `-o FAIL_FAST`.

3.4 新版功能.

doctest.**REPORTING_FLAGS**
A bitmask or'ing together all the reporting flags above.

There is also a way to register new option flag names, though this isn't useful unless you intend to extend *doctest* internals via subclassing:

doctest.**register_optionflag**(*name*)
Create a new option flag with a given name, and return the new flag's integer value. *register_optionflag()* can be used when subclassing *OutputChecker* or *DocTestRunner* to create new options that are supported by your subclasses. *register_optionflag()* should always be called using the following idiom:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

Directives

Doctest directives may be used to modify the *option flags* for an individual example. Doctest directives are special Python comments following an example's source code:

```
directive             ::=   "#" "doctest:" directive_options
directive_options     ::=   directive_option ("," directive_option)\*
directive_option      ::=   on_or_off directive_option_name
on_or_off             ::=   "+" \| "-"
directive_option_name ::=   "DONT_ACCEPT_BLANKLINE" \| "NORMALIZE_WHITESPACE" \| ...
```

Whitespace is not allowed between the + or − and the directive option name. The directive option name can be any of the option flag names explained above.

An example's doctest directives modify doctest's behavior for that single example. Use + to enable the named behavior, or − to disable it.

For example, this test passes:

```
>>> print(list(range(20)))
[0,   1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Without the directive it would fail, both because the actual output doesn't have two blanks before the single-digit list elements, and because the actual output is on a single line. This test also passes, and also requires a directive to do so:

```
>>> print(list(range(20)))
[0, 1, ..., 18, 19]
```

Multiple directives can be used on a single physical line, separated by commas:

```
>>> print(list(range(20)))
[0,    1, ...,   18,    19]
```

If multiple directive comments are used for a single example, then they are combined:

```
>>> print(list(range(20)))
...
[0,    1, ...,   18,    19]
```

As the previous example shows, you can add `...` lines to your example containing only directives. This can be useful when an example is too long for a directive to comfortably fit on the same line:

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
...
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

Note that since all options are disabled by default, and directives apply only to the example they appear in, enabling options (via + in a directive) is usually the only meaningful choice. However, option flags can also be passed to functions that run doctests, establishing different defaults. In such cases, disabling an option via − in a directive can be useful.

警告

doctest is serious about requiring exact matches in expected output. If even a single character doesn't match, the test fails. This will probably surprise you a few times, as you learn exactly what Python does and doesn't guarantee about output. For example, when printing a dict, Python doesn't guarantee that the key-value pairs will be printed in any particular order, so a test like

```
>>> foo()
{"Hermione": "hippogryph", "Harry": "broomstick"}
```

is vulnerable! One workaround is to do

```
>>> foo() == {"Hermione": "hippogryph", "Harry": "broomstick"}
True
```

instead. Another is to do

```
>>> d = sorted(foo().items())
>>> d
[('Harry', 'broomstick'), ('Hermione', 'hippogryph')]
```

There are others, but you get the idea.

Another bad idea is to print things that embed an object address, like

```
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C()    # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

The *ELLIPSIS* directive gives a nice approach for the last example:

```
>>> C()
<__main__.C instance at 0x...>
```

Floating-point numbers are also subject to small output variations across platforms, because Python defers to the platform C library for float formatting, and C libraries vary widely in quality here.

```
>>> 1./7  # risky
0.14285714285714285
>>> print(1./7) # safer
0.142857142857
>>> print(round(1./7, 6)) # much safer
0.142857
```

Numbers of the form `I/2.**J` are safe across all platforms, and I often contrive doctest examples to produce numbers of that form:

```
>>> 3./4  # utterly safe
0.75
```

Simple fractions are also easier for people to understand, and that makes for better documentation.

26.3.4 Basic API

The functions `testmod()` and `testfile()` provide a simple interface to doctest that should be sufficient for most basic uses. For a less formal introduction to these two functions, see sections 简单用法：检查 *Docstrings* 中的示例 and *Simple Usage: Checking Examples in a Text File*.

doctest.**testfile**(*filename*, *module_relative=True*, *name=None*, *package=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *parser=DocTestParser()*, *encoding=None*)

All arguments except *filename* are optional, and should be specified in keyword form.

Test examples in the file named *filename*. Return `(failure_count, test_count)`.

Optional argument *module_relative* specifies how the filename should be interpreted:

- If *module_relative* is `True` (the default), then *filename* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, *filename* should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then *filename* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *name* gives the name of the test; by default, or if `None`, `os.path.basename(filename)` is used.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for a module-relative filename. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *globs* gives a dict to be used as the globals when executing examples. A new shallow copy of this dict is created for the doctest, so its examples start with a clean slate. By default, or if `None`, a new empty dict is used.

Optional argument *extraglobs* gives a dict merged into the globals used to execute examples. This works like `dict.update()`: if *globs* and *extraglobs* have a common key, the associated value in *extraglobs* appears in the combined dict. By default, or if `None`, no extra globals are used. This is an advanced feature that allows parameterization of doctests. For example, a doctest can be written for a base class, using a generic name for the class, then reused to test any number of subclasses by passing an *extraglobs* dict mapping the generic name to the subclass to be tested.

Optional argument *verbose* prints lots of stuff if true, and prints only failures if false; by default, or if `None`, it's true if and only if `'-v'` is in `sys.argv`.

Optional argument *report* prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else the summary is very brief (in fact, empty if all tests passed).

Optional argument *optionflags* (default value 0) takes the bitwise OR of option flags. See section *Option Flags*.

Optional argument *raise_on_error* defaults to false. If true, an exception is raised upon the first failure or unexpected exception in an example. This allows failures to be post-mortem debugged. Default behavior is to continue running examples.

Optional argument *parser* specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

doctest.**testmod**(*m=None*, *name=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *exclude_empty=False*)

All arguments are optional, and all except for *m* should be specified in keyword form.

Test examples in docstrings in functions and classes reachable from module *m* (or module *__main__* if *m* is not supplied or is None), starting with m.__doc__.

Also test examples reachable from dict m.__test__, if it exists and is not None. m.__test__ maps names (strings) to functions, classes and strings; function and class docstrings are searched for examples; strings are searched directly, as if they were docstrings.

Only docstrings attached to objects belonging to module *m* are searched.

Return (failure_count, test_count).

Optional argument *name* gives the name of the module; by default, or if None, m.__name__ is used.

Optional argument *exclude_empty* defaults to false. If true, objects for which no doctests are found are excluded from consideration. The default is a backward compatibility hack, so that code still using doctest.master.summarize() in conjunction with *testmod()* continues to get output for objects with no tests. The *exclude_empty* argument to the newer *DocTestFinder* constructor defaults to true.

Optional arguments *extraglobs*, *verbose*, *report*, *optionflags*, *raise_on_error*, and *globs* are the same as for function *testfile()* above, except that *globs* defaults to m.__dict__.

doctest.**run_docstring_examples**(*f*, *globs*, *verbose=False*, *name="NoName"*, *compileflags=None*, *optionflags=0*)

Test examples associated with object *f*; for example, *f* may be a string, a module, a function, or a class object.

A shallow copy of dictionary argument *globs* is used for the execution context.

Optional argument *name* is used in failure messages, and defaults to "NoName".

If optional argument *verbose* is true, output is generated even if there are no failures. By default, output is generated only in case of an example failure.

Optional argument *compileflags* gives the set of flags that should be used by the Python compiler when running the examples. By default, or if None, flags are deduced corresponding to the set of future features found in *globs*.

Optional argument *optionflags* works as for function *testfile()* above.

26.3.5 Unittest API

As your collection of doctest'ed modules grows, you'll want a way to run all their doctests systematically. *doctest* provides two functions that can be used to create *unittest* test suites from modules and text files containing doctests. To integrate with *unittest* test discovery, include a load_tests() function in your test module:

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

There are two main functions for creating *unittest.TestSuite* instances from text files and modules with doctests:

doctest.**DocFileSuite**(**paths*, *module_relative=True*, *package=None*, *setUp=None*, *tearDown=None*, *globs=None*, *optionflags=0*, *parser=DocTestParser()*, *encoding=None*)

Convert doctest tests from one or more text files to a *unittest.TestSuite*.

The returned *unittest.TestSuite* is to be run by the unittest framework and runs the interactive examples in each file. If an example in any file fails, then the synthesized unit test fails, and a failureException exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Pass one or more paths (as strings) to text files to be examined.

Options may be provided as keyword arguments:

Optional argument *module_relative* specifies how the filenames in *paths* should be interpreted:

- If *module_relative* is `True` (the default), then each filename in *paths* specifies an OS-independent module-relative path. By default, this path is relative to the calling module's directory; but if the *package* argument is specified, then it is relative to that package. To ensure OS-independence, each filename should use `/` characters to separate path segments, and may not be an absolute path (i.e., it may not begin with `/`).
- If *module_relative* is `False`, then each filename in *paths* specifies an OS-specific path. The path may be absolute or relative; relative paths are resolved with respect to the current working directory.

Optional argument *package* is a Python package or the name of a Python package whose directory should be used as the base directory for module-relative filenames in *paths*. If no package is specified, then the calling module's directory is used as the base directory for module-relative filenames. It is an error to specify *package* if *module_relative* is `False`.

Optional argument *setUp* specifies a set-up function for the test suite. This is called before running the tests in each file. The *setUp* function will be passed a `DocTest` object. The setUp function can access the test globals as the *globs* attribute of the test passed.

Optional argument *tearDown* specifies a tear-down function for the test suite. This is called after running the tests in each file. The *tearDown* function will be passed a `DocTest` object. The setUp function can access the test globals as the *globs* attribute of the test passed.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *optionflags* specifies the default doctest options for the tests, created by or-ing together individual option flags. See section *Option Flags*. See function `set_unittest_reportflags()` below for a better way to set reporting options.

Optional argument *parser* specifies a `DocTestParser` (or subclass) that should be used to extract tests from the files. It defaults to a normal parser (i.e., `DocTestParser()`).

Optional argument *encoding* specifies an encoding that should be used to convert the file to unicode.

The global `__file__` is added to the globals provided to doctests loaded from a text file using `DocFileSuite()`.

doctest.**DocTestSuite**(*module=None*, *globs=None*, *extraglobs=None*, *test_finder=None*, *setUp=None*, *tearDown=None*, *checker=None*)

Convert doctest tests for a module to a `unittest.TestSuite`.

The returned `unittest.TestSuite` is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a `failureException` exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

Optional argument *module* provides the module to be tested. It can be a module object or a (possibly dotted) module name. If not specified, the module calling this function is used.

Optional argument *globs* is a dictionary containing the initial global variables for the tests. A new copy of this dictionary is created for each test. By default, *globs* is a new empty dictionary.

Optional argument *extraglobs* specifies an extra set of global variables, which is merged into *globs*. By default, no extra globals are used.

Optional argument *test_finder* is the `DocTestFinder` object (or a drop-in replacement) that is used to extract doctests from the module.

Optional arguments *setUp*, *tearDown*, and *optionflags* are the same as for function `DocFileSuite()` above.

This function uses the same search technique as *testmod()*.

在 3.5 版更改: *DocTestSuite()* returns an empty *unittest.TestSuite* if *module* contains no docstrings instead of raising *ValueError*.

Under the covers, *DocTestSuite()* creates a *unittest.TestSuite* out of `doctest.DocTestCase` instances, and `DocTestCase` is a subclass of *unittest.TestCase*. `DocTestCase` isn't documented here (it's an internal detail), but studying its code can answer questions about the exact details of *unittest* integration.

Similarly, *DocFileSuite()* creates a *unittest.TestSuite* out of `doctest.DocFileCase` instances, and `DocFileCase` is a subclass of `DocTestCase`.

So both ways of creating a *unittest.TestSuite* run instances of `DocTestCase`. This is important for a subtle reason: when you run *doctest* functions yourself, you can control the *doctest* options in use directly, by passing option flags to *doctest* functions. However, if you're writing a *unittest* framework, *unittest* ultimately controls when and how tests get run. The framework author typically wants to control *doctest* reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through *unittest* to *doctest* test runners.

For this reason, *doctest* also supports a notion of *doctest* reporting flags specific to *unittest* support, via this function:

`doctest.`**`set_unittest_reportflags`**(*flags*)

Set the *doctest* reporting flags to use.

Argument *flags* takes the bitwise OR of option flags. See section *Option Flags*. Only "reporting flags" can be used.

This is a module-global setting, and affects all future doctests run by module *unittest*: the `runTest()` method of `DocTestCase` looks at the option flags specified for the test case when the `DocTestCase` instance was constructed. If no reporting flags were specified (which is the typical and expected case), *doctest*'s *unittest* reporting flags are bitwise ORed into the option flags, and the option flags so augmented are passed to the *DocTestRunner* instance created to run the doctest. If any reporting flags were specified when the `DocTestCase` instance was constructed, *doctest*'s *unittest* reporting flags are ignored.

The value of the *unittest* reporting flags in effect before the function was called is returned by the function.

26.3.6 Advanced API

The basic API is a simple wrapper that's intended to make doctest easy to use. It is fairly flexible, and should meet most users' needs; however, if you require more fine-grained control over testing, or wish to extend doctest's capabilities, then you should use the advanced API.

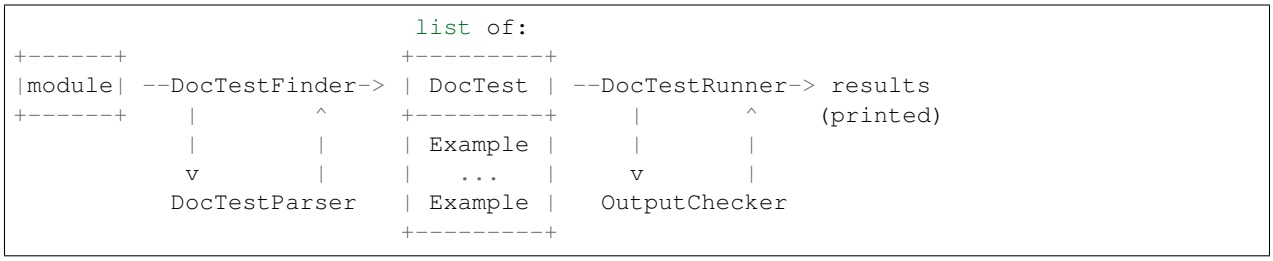
The advanced API revolves around two container classes, which are used to store the interactive examples extracted from doctest cases:

- *Example*: A single Python *statement*, paired with its expected output.
- *DocTest*: A collection of *Example*s, typically extracted from a single docstring or text file.

Additional processing classes are defined to find, parse, and run, and check doctest examples:

- *DocTestFinder*: Finds all docstrings in a given module, and uses a *DocTestParser* to create a *DocTest* from every docstring that contains interactive examples.
- *DocTestParser*: Creates a *DocTest* object from a string (such as an object's docstring).
- *DocTestRunner*: Executes the examples in a *DocTest*, and uses an *OutputChecker* to verify their output.
- *OutputChecker*: Compares the actual output from a doctest example with the expected output, and decides whether they match.

The relationships among these processing classes are summarized in the following diagram:



DocTest 对象

class doctest.**DocTest**(*examples*, *globs*, *name*, *filename*, *lineno*, *docstring*)

A collection of doctest examples that should be run in a single namespace. The constructor arguments are used to initialize the attributes of the same names.

DocTest defines the following attributes. They are initialized by the constructor, and should not be modified directly.

examples

A list of *Example* objects encoding the individual interactive Python examples that should be run by this test.

globs

The namespace (aka globals) that the examples should be run in. This is a dictionary mapping names to values. Any changes to the namespace made by the examples (such as binding new variables) will be reflected in *globs* after the test is run.

name

A string name identifying the *DocTest*. Typically, this is the name of the object or file that the test was extracted from.

filename

The name of the file that this *DocTest* was extracted from; or None if the filename is unknown, or if the *DocTest* was not extracted from a file.

lineno

The line number within *filename* where this *DocTest* begins, or None if the line number is unavailable. This line number is zero-based with respect to the beginning of the file.

docstring

The string that the test was extracted from, or None if the string is unavailable, or if the test was not extracted from a string.

Example Objects

class doctest.**Example**(*source*, *want*, *exc_msg=None*, *lineno=0*, *indent=0*, *options=None*)

A single interactive example, consisting of a Python statement and its expected output. The constructor arguments are used to initialize the attributes of the same names.

Example defines the following attributes. They are initialized by the constructor, and should not be modified directly.

source

A string containing the example's source code. This source code consists of a single Python statement, and always ends with a newline; the constructor adds a newline when necessary.

want
The expected output from running the example's source code (either from stdout, or a traceback in case of exception). *want* ends with a newline unless no output is expected, in which case it's an empty string. The constructor adds a newline when necessary.

exc_msg
The exception message generated by the example, if the example is expected to generate an exception; or `None` if it is not expected to generate an exception. This exception message is compared against the return value of *traceback.format_exception_only()*. *exc_msg* ends with a newline unless it's `None`. The constructor adds a newline if needed.

lineno
The line number within the string containing this example where the example begins. This line number is zero-based with respect to the beginning of the containing string.

indent
The example's indentation in the containing string, i.e., the number of space characters that precede the example's first prompt.

options
A dictionary mapping from option flags to `True` or `False`, which is used to override default options for this example. Any option flags not contained in this dictionary are left at their default value (as specified by the *DocTestRunner*'s `optionflags`). By default, no options are set.

DocTestFinder 对象

class doctest.**DocTestFinder**(*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)
A processing class used to extract the *DocTest*s that are relevant to a given object, from its docstring and the docstrings of its contained objects. *DocTest*s can be extracted from modules, classes, functions, methods, staticmethods, classmethods, and properties.

The optional argument *verbose* can be used to display the objects searched by the finder. It defaults to `False` (no output).

The optional argument *parser* specifies the *DocTestParser* object (or a drop-in replacement) that is used to extract doctests from docstrings.

If the optional argument *recurse* is false, then *DocTestFinder.find()* will only examine the given object, and not any contained objects.

If the optional argument *exclude_empty* is false, then *DocTestFinder.find()* will include tests for objects with empty docstrings.

DocTestFinder defines the following method:

find(*obj[, name][, module][, globs][, extraglobs]*)
Return a list of the *DocTest*s that are defined by *obj*'s docstring, or by any of its contained objects' docstrings.

The optional argument *name* specifies the object's name; this name will be used to construct names for the returned *DocTest*s. If *name* is not specified, then `obj.__name__` is used.

The optional parameter *module* is the module that contains the given object. If the module is not specified or is `None`, then the test finder will attempt to automatically determine the correct module. The object's module is used:

- As a default namespace, if *globs* is not specified.

- To prevent the DocTestFinder from extracting DocTests from objects that are imported from other modules. (Contained objects with modules other than *module* are ignored.)
- To find the name of the file containing the object.
- To help find the line number of the object within its file.

If *module* is `False`, no attempt to find the module will be made. This is obscure, of use mostly in testing doctest itself: if *module* is `False`, or is `None` but cannot be found automatically, then all objects are considered to belong to the (non-existent) module, so all contained objects will (recursively) be searched for doctests.

The globals for each `DocTest` is formed by combining *globs* and *extraglobs* (bindings in *extraglobs* override bindings in *globs*). A new shallow copy of the globals dictionary is created for each `DocTest`. If *globs* is not specified, then it defaults to the module's *__dict__*, if specified, or `{}` otherwise. If *extraglobs* is not specified, then it defaults to `{}`.

DocTestParser 对象

class doctest.**DocTestParser**

A processing class used to extract interactive examples from a string, and use them to create a `DocTest` object.

`DocTestParser` defines the following methods:

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)

Extract all doctest examples from the given string, and collect them into a `DocTest` object.

globs, *name*, *filename*, and *lineno* are attributes for the new `DocTest` object. See the documentation for `DocTest` for more information.

get_examples(*string*, *name='<string>'*)

Extract all doctest examples from the given string, and return them as a list of `Example` objects. Line numbers are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

parse(*string*, *name='<string>'*)

Divide the given string into examples and intervening text, and return them as a list of alternating `Example`s and strings. Line numbers for the `Example`s are 0-based. The optional argument *name* is a name identifying this string, and is only used for error messages.

DocTestRunner 对象

class doctest.**DocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

A processing class used to execute and verify the interactive examples in a `DocTest`.

The comparison between expected outputs and actual outputs is done by an `OutputChecker`. This comparison may be customized with a number of option flags; see section *Option Flags* for more information. If the option flags are insufficient, then the comparison may also be customized by passing a subclass of `OutputChecker` to the constructor.

The test runner's display output can be controlled in two ways. First, an output function can be passed to `TestRunner.run()`; this function will be called with strings that should be displayed. It defaults to `sys.stdout.write`. If capturing the output is not sufficient, then the display output can be also customized by subclassing DocTestRunner, and overriding the methods `report_start()`, `report_success()`, `report_unexpected_exception()`, and `report_failure()`.

The optional keyword argument *checker* specifies the `OutputChecker` object (or drop-in replacement) that should be used to compare the expected outputs to the actual outputs of doctest examples.

The optional keyword argument *verbose* controls the `DocTestRunner`'s verbosity. If *verbose* is `True`, then information is printed about each example, as it is run. If *verbose* is `False`, then only failures are printed. If *verbose* is unspecified, or `None`, then verbose output is used iff the command-line switch `-v` is used.

The optional keyword argument *optionflags* can be used to control how the test runner compares expected output to actual output, and how it displays failures. For more information, see section *Option Flags*.

`DocTestParser` defines the following methods:

report_start(*out*, *test*, *example*)
Report that the test runner is about to process the given example. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *test* is the test *containing example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_success(*out*, *test*, *example*, *got*)
Report that the given example ran successfully. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_failure(*out*, *test*, *example*, *got*)
Report that the given example failed. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *got* is the actual output from the example. *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)
Report that the given example raised an unexpected exception. This method is provided to allow subclasses of `DocTestRunner` to customize their output; it should not be called directly.

example is the example about to be processed. *exc_info* is a tuple containing information about the unexpected exception (as returned by `sys.exc_info()`). *test* is the test containing *example*. *out* is the output function that was passed to `DocTestRunner.run()`.

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)
Run the examples in *test* (a `DocTest` object), and display the results using the writer function *out*.

The examples are run in the namespace `test.globs`. If *clear_globs* is true (the default), then this namespace will be cleared after the test runs, to help with garbage collection. If you would like to examine the namespace after the test completes, then use *clear_globs=False*.

compileflags gives the set of flags that should be used by the Python compiler when running the examples. If not specified, then it will default to the set of future-import flags that apply to *globs*.

The output of each example is checked using the `DocTestRunner`'s output checker, and the results are formatted by the `DocTestRunner.report_*()` methods.

summarize(*verbose=None*)
Print a summary of all the test cases that have been run by this DocTestRunner, and return a *named tuple* `TestResults(failed, attempted)`.

The optional *verbose* argument controls how detailed the summary is. If the verbosity is not specified, then the `DocTestRunner`'s verbosity is used.

OutputChecker 对象

class doctest.**OutputChecker**

A class used to check the whether the actual output from a doctest example matches the expected output. *OutputChecker* defines two methods: *check_output()*, which compares a given pair of outputs, and returns true if they match; and *output_difference()*, which returns a string describing the differences between two outputs.

OutputChecker defines the following methods:

check_output(*want*, *got*, *optionflags*)

Return `True` iff the actual output from an example (*got*) matches the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option flags the test runner is using, several non-exact match types are also possible. See section *Option Flags* for more information about option flags.

output_difference(*example*, *got*, *optionflags*)

Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*). *optionflags* is the set of option flags used to compare *want* and *got*.

26.3.7 调试

Doctest provides several mechanisms for debugging doctest examples:

- Several functions convert doctests to executable Python programs, which can be run under the Python debugger, *pdb*.
- The *DebugRunner* class is a subclass of *DocTestRunner* that raises an exception for the first failing example, containing information about that example. This information can be used to perform post-mortem debugging on the example.
- The *unittest* cases generated by *DocTestSuite()* support the *debug()* method defined by *unittest.TestCase*.
- You can add a call to *pdb.set_trace()* in a doctest example, and you'll drop into the Python debugger when that line is executed. Then you can inspect current values of variables, and so on. For example, suppose `a.py` contains just this module docstring:

```
"""
>>> def f(x):
...     g(x*2)
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

Then an interactive Python session may look like this:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
```

(下页继续)

(续上页)

```
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

Functions that convert doctests to Python code, and possibly run the synthesized code under the debugger:

doctest.**script_from_examples**(*s*)

Convert text with examples to a script.

Argument *s* is a string containing doctest examples. The string is converted to a Python script, where doctest examples in *s* are converted to regular code, and everything else is converted to Python comments. The generated script is returned as a string. For example,

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
"""))
```

displays:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

This function is used internally by other functions (see below), but can also be useful when you want to transform an interactive Python session into a Python script.

doctest.**testsource**(*module*, *name*)

Convert the doctest for an object to a script.

Argument *module* is a module object, or dotted name of a module, containing the object whose doctests are of interest. Argument *name* is the name (within the module) of the object with the doctests of interest. The result is a string, containing the object's docstring converted to a Python script, as described for *script_from_examples()* above. For example, if module `a.py` contains a top-level function `f()`, then

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

prints a script version of function `f()`'s docstring, with doctests converted to code, and the rest placed in comments.

`doctest.`**`debug`**(*module*, *name*, *pm=False*)

Debug the doctests for an object.

The *module* and *name* arguments are the same as for function *testsource()* above. The synthesized Python script for the named object's docstring is written to a temporary file, and then that file is run under the control of the Python debugger, *pdb*.

A shallow copy of `module.__dict__` is used for both local and global execution context.

Optional argument *pm* controls whether post-mortem debugging is used. If *pm* has a true value, the script file is run directly, and the debugger gets involved only if the script terminates via raising an unhandled exception. If it does, then post-mortem debugging is invoked, via *pdb.post_mortem()*, passing the traceback object from the unhandled exception. If *pm* is not specified, or is false, the script is run under the debugger from the start, via passing an appropriate *exec()* call to *pdb.run()*.

`doctest.`**`debug_src`**(*src*, *pm=False*, *globs=None*)

Debug the doctests in a string.

This is like function *debug()* above, except that a string containing doctest examples is specified directly, via the *src* argument.

Optional argument *pm* has the same meaning as in function *debug()* above.

Optional argument *globs* gives a dictionary to use as both local and global execution context. If not specified, or `None`, an empty dictionary is used. If specified, a shallow copy of the dictionary is used.

The *DebugRunner* class, and the special exceptions it may raise, are of most interest to testing framework authors, and will only be sketched here. See the source code, and especially *DebugRunner*'s docstring (which is a doctest!) for more details:

`class` `doctest.`**`DebugRunner`**(*checker=None*, *verbose=None*, *optionflags=0*)

A subclass of *DocTestRunner* that raises an exception as soon as a failure is encountered. If an unexpected exception occurs, an *UnexpectedException* exception is raised, containing the test, the example, and the original exception. If the output doesn't match, then a *DocTestFailure* exception is raised, containing the test, the example, and the actual output.

For information about the constructor parameters and methods, see the documentation for *DocTestRunner* in section *Advanced API*.

There are two exceptions that may be raised by *DebugRunner* instances:

`exception` `doctest.`**`DocTestFailure`**(*test*, *example*, *got*)

An exception raised by *DocTestRunner* to signal that a doctest example's actual output did not match its expected output. The constructor arguments are used to initialize the attributes of the same names.

DocTestFailure defines the following attributes:

`DocTestFailure.`**`test`**

The *DocTest* object that was being run when the example failed.

DocTestFailure.**example**
The *Example* that failed.

DocTestFailure.**got**
The example's actual output.

exception doctest.**UnexpectedException**(*test*, *example*, *exc_info*)
An exception raised by *DocTestRunner* to signal that a doctest example raised an unexpected exception. The constructor arguments are used to initialize the attributes of the same names.

UnexpectedException defines the following attributes:

UnexpectedException.**test**
The *DocTest* object that was being run when the example failed.

UnexpectedException.**example**
The *Example* that failed.

UnexpectedException.**exc_info**
A tuple containing information about the unexpected exception, as returned by *sys.exc_info()*.

26.3.8 Soapbox

As mentioned in the introduction, *doctest* has grown to have three primary uses:

1. Checking examples in docstrings.
2. Regression testing.
3. Executable documentation / literate testing.

These uses have different requirements, and it is important to distinguish them. In particular, filling your docstrings with obscure test cases makes for bad documentation.

When writing a docstring, choose docstring examples with care. There's an art to this that needs to be learned—it may not be natural at first. Examples should add genuine value to the documentation. A good example can often be worth many words. If done with care, the examples will be invaluable for your users, and will pay back the time it takes to collect them many times over as the years go by and things change. I'm still amazed at how often one of my *doctest* examples stops working after a "harmless" change.

Doctest also makes an excellent tool for regression testing, especially if you don't skimp on explanatory text. By interleaving prose and examples, it becomes much easier to keep track of what's actually being tested, and why. When a test fails, good prose can make it much easier to figure out what the problem is, and how it should be fixed. It's true that you could write extensive comments in code-based testing, but few programmers do. Many have found that using doctest approaches instead leads to much clearer tests. Perhaps this is simply because doctest makes writing prose a little easier than writing code, while writing comments in code is a little harder. I think it goes deeper than just that: the natural attitude when writing a doctest-based test is that you want to explain the fine points of your software, and illustrate them with examples. This in turn naturally leads to test files that start with the simplest features, and logically progress to complications and edge cases. A coherent narrative is the result, instead of a collection of isolated functions that test isolated bits of functionality seemingly at random. It's a different attitude, and produces different results, blurring the distinction between testing and explaining.

Regression testing is best confined to dedicated objects or files. There are several options for organizing tests:

- Write text files containing test cases as interactive examples, and test the files using *testfile()* or *DocFileSuite()*. This is recommended, although is easiest to do for new projects, designed from the start to use doctest.
- Define functions named `_regrtest_topic` that consist of single docstrings, containing test cases for the named topics. These functions can be included in the same file as the module, or separated out into a separate test file.

- Define a `__test__` dictionary mapping from regression test topics to docstrings containing test cases.

When you have placed your tests in a module, the module can itself be the test runner. When a test fails, you can arrange for your test runner to re-run only the failing doctest while you debug the problem. Here is a minimal example of such a test runner:

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print("{} failures out of {} tests".format(fail, total))
```

备注

26.4 unittest —单元测试框架

源代码： Lib/unittest/__init__.py

(如果你已经对测试的概念比较熟悉了，你可能想直接跳转到这一部分*断言方法*。)

unittest 单元测试框架是受到 JUnit 的启发，与其他语言中的主流单元测试框架有着相似的风格。其支持测试自动化，配置共享和关机代码测试。支持将测试样例聚合到测试集中，并将测试与报告框架独立。

为了实现这些，*unittest* 通过面向对象的方式支持了一些重要的概念。

测试脚手架 *test fixture* 表示为了开展一项或多项测试所需要进行的准备工作，以及所有相关的清理操作。举个例子，这可能包含创建临时或代理的数据库、目录，再或者启动一个服务器进程。

测试用例 一个测试用例是一个独立的测试单元。它检查输入特定的数据时的响应。*unittest* 提供一个基类：*TestCase*，用于新建测试用例。

测试套件 *test suite* 是一系列的测试用例，或测试套件，或两者皆有。它用于归档需要一起执行的测试。

测试运行器（test runner） *test runner* 是一个用于执行和输出测试结果的组件。这个运行器可能使用图形接口、文本接口，或返回一个特定的值表示运行测试的结果。

参见：

***doctest*—文档测试模块** 另一个风格完全不同的测试模块。

简单 Smalltalk 测试：便用模式 Kent Beck 有关使用*unittest* 所共享的模式的测试框架的原创论文。

Nose and pytest Third-party unittest frameworks with a lighter-weight syntax for writing tests. For example, `assert func(10) == 42`.

Python 测试工具分类 一个 Python 测试工具的详细列表，包含测试框架和模拟对象库。

Python 中的测试邮件列表 一个讨论 Python 中的测试和测试工具的特别兴趣小组。

包含在源代码分发中的 Tools/unittestgui/unittestgui.py 文件是一个用于显示测试覆盖率和执行情况的 GUI 工具。它的目的是给那些单元测试的新手们提供方便。在生产环境中建议测试应由持续集成系统（continuous integration system）驱动，例如：Buildbot、Jenkins 或 Hudson。

26.4.1 基本实例

unittest 模块提供了一系列创建和运行测试的工具。这一段落演示了这些工具的一小部分，但也足以满足大部分用户的需求。

这是一段简短的代码，来测试三种字符串方法:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

继承*unittest.TestCase* 就创建了一个测试样例。上述三个独立的测试是三个类的方法，这些方法的命名都以 test 开头。这个命名约定告诉测试运行者类的哪些方法表示测试。

每个测试的关键是：调用*assertEqual()* 来检查预期的输出；调用*assertTrue()* 或*assertFalse()* 来验证一个条件；调用*assertRaises()* 来验证抛出了一个特定的异常。使用这些方法而不是 assert 语句是为了让测试运行者能聚合所有的测试结果并产生结果报告。

通过*setUp()* 和*tearDown()* 方法，可以设置测试开始前与完成后需要执行的指令。在*组织你的测试代码* 中，对此有更为详细的描述。

最后的代码块中，演示了运行测试的一个简单的方法。*unittest.main()* 提供了一个测试脚本的命令行接口。当在命令行运行该测试脚本，上文的脚本生成如以下格式的输出:

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

在调用测试脚本时添加 -v 参数使*unittest.main()* 显示更为详细的信息，生成如以下形式的输出:

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok
```

(下页继续)

(续上页)

```
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

以上例子演示了*unittest* 中最常用的、足够满足许多日常测试需求的特性。文档的剩余部分详述该框架的完整特性。

26.4.2 命令行界面

unittest 模块可以通过命令行运行模块、类和独立测试方法的测试:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

你可以传入模块名、类或方法名或他们的任意组合。

同样的，测试模块可以通过文件路径指定:

```
python -m unittest tests/test_something.py
```

这样就可以使用 shell 的文件名补全指定测试模块。所指定的文件仍需要可以被作为模块导入。路径通过去除‘.py’、把分隔符转换为‘.’转换为模块名。若你需要执行不能被作为模块导入的测试文件，你需要直接执行该测试文件。

在运行测试时，你可以通过添加 -v 参数获取更详细（更多的冗余）的信息。

```
python -m unittest -v test_module
```

当运行时不包含参数，开始探索性测试

```
python -m unittest
```

用于获取命令行选项列表:

```
python -m unittest -h
```

在 3.2 版更改: 在早期版本中，只支持运行独立的测试方法，而不支持模块和类。

命令行选项

unittest supports these command-line options:

-b, --buffer
在测试运行时，标准输出流与标准错误流会被放入缓冲区。成功的测试的运行时输出会被丢弃；测试不通过时，测试运行中的输出会正常显示，错误会被加入到测试失败信息。

-c, --catch
当测试正在运行时，Control-C 会等待当前测试完成，并在完成后报告已执行的测试的结果。当再次按下 Control-C 时，引发平常的*KeyboardInterrupt* 异常。

See *Signal Handling* for the functions that provide this functionality.

-f, --failfast
当出现第一个错误或者失败时，停止运行测试。

--locals
在回溯中显示局部变量。

3.2 新版功能: 添加命令行选项 -b, -c 和 -f 。

3.5 新版功能: 命令行选项 --locals 。

命令行亦可用于探索性测试，以运行一个项目的所有测试或其子集。

26.4.3 探索性测试

3.2 新版功能.

Unittest 支持简单的测试搜索。若需要使用探索性测试，所有的测试文件必须是 modules 或 packages （包括*namespace packages*) 并可从项目根目录导入（即它们的文件名必须是有效的 identifiers)。

探索性测试在*TestLoader.discover()* 中实现，但也可以通过命令行使用。它在命令行中的基本用法如下:

```
cd project_directory
python -m unittest discover
```

注解: 方便起见，python -m unittest 与 python -m unittest discover 等价。如果你需要向探索性测试传入参数，必须显式地使用 discover 子命令。

discover 有以下选项:

-v, --verbose
更详细地输出结果。

-s, --start-directory directory
开始进行搜索的目录 (默认值为当前目录 .)。

-p, --pattern pattern
用于匹配测试文件的模式（默认为 test*.py)。

-t, --top-level-directory directory
指定项目的最上层目录（通常为开始时所在目录)。

-s ， *-p* 和*-t* 选项可以按顺序作为位置参数传入。以下两条命令是等价的:

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

正如可以传入路径那样，传入一个包名作为起始目录也是可行的，如 myproject.subpackage.test 。你提供的包名会被导入，它在文件系统中的位置会被作为起始目录。

警告: 探索性测试通过导入测试对测试进行加载。在找到所有你指定的开始目录下的所有测试文件后，它把路径转换为包名并进行导入。如 foo/bar/baz.py 会被导入为 foo.bar.baz 。

如果你有一个全局安装的包，并尝试对这个包的副本进行探索性测试，可能会从错误的地方开始导入。如果出现这种情况，测试会输出警告并退出。

如果你使用包名而不是路径作为开始目录，搜索时会假定它导入的是你想要的目录，所以你不会收到警告。

测试模块和包可以通过*load_tests protocol* 自定义测试的加载和搜索。

在 3.4 版更改: 探索性测试支持命名空间包（*namespace packages* ）。

26.4.4 组织你的测试代码

单元测试的构建单位是 *test cases*： 独立的、包含执行条件与正确性检查的方案。在*unittest* 中，测试用例表示为*unittest.TestCase* 的实例。通过编写*TestCase* 的子类或使用*FunctionTestCase* 编写你自己的测试用例。

一个*TestCase* 实例的测试代码必须是完全自含的，因此它可以独立运行，或与其它任意组合任意数量的测试用例一起运行。

TestCase 的最简单的子类需要实现一个测试方法（例如一个命名以 `test` 开头的方法）以执行特定的测试代码:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

可以看到，为了进行测试，我们使用了基类*TestCase* 提供的其中一个 `assert*()` 方法。若测试不通过，将会引发一个带有说明信息的异常，并且*unittest* 会将这个测试用例标记为测试不通过。任何其它类型的异常将会被当做错误处理。

可能同时存在多个前置操作相同的测试，我们可以把测试的前置操作从测试代码中拆解出来，并实现测试前置方法*setUp()* 。在运行测试时，测试框架会自动地为每个单独测试调用前置方法。

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

注解: 多个测试运行的顺序由内置字符串排序方法对测试名进行排序的结果决定。

在测试运行时，若*setUp()* 方法引发异常，测试框架会认为测试发生了错误，因此测试方法不会被运行。

相似的，我们提供了一个*tearDown()* 方法在测试方法运行后进行清理工作。

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')
```

(下页继续)

(续上页)

```
    def tearDown(self):
        self.widget.dispose()
```

若*setUp()* 成功运行，无论测试方法是否成功，都会运行*tearDown()* 。

这样的一个测试代码运行的环境被称为 *test fixture* 。一个新的 TestCase 实例作为一个测试脚手架，用于运行各个独立的测试方法。在运行每个测试时，*setUp()* 、*tearDown()* 和 __init__() 会被调用一次。

建议你根据所测试的功能，将测试用 TestCase 实现集合起来。*unittest* 为此提供了机制：*test suite*，以*unittest* 的类*TestSuite* 为代表。大部分情况下，调用*unittest.main()* 即可，并且它会为你集合所有模块的测试用例并执行。

然而，如果你需要自定义你的测试套件的话，你可以参考以下方法组织你的测试：

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

你可以把测试用例和测试套件放在与被测试代码相同的模块中（比如 widget.py），但将测试代码放在单独的模块中（比如 test_widget.py）有几个优势。

- 测试模块可以从命令行被独立调用。
- 更容易在分发的代码中剥离测试代码。
- 降低没有好理由的情况下修改测试代码以通过测试的诱惑。
- 测试代码应比被测试代码更少地被修改。
- 被测试代码可以更容易地被重构。
- 对用 C 语言写成的模块无论如何都得单独写成一个模块，为什么不保持一致呢?
- 如果测试策略发生了改变，没有必要修改源代码。

26.4.5 复用已有的测试代码

一些用户希望直接使用*unittest* 运行已有的测试代码，而不需要把已有的每个测试函数转化为一个*TestCase* 的子类。

因此，*unittest* 提供*FunctionTestCase* 类。这个*TestCase* 的子类可用于打包已有的测试函数，并支持设置前置与后置函数。

假定有一个测试函数：

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

可以创建等价的测试用例如下，其中前置和后置方法是可选的。

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

注解： 可以用*FunctionTestCase*快速将现有的测试转换成基于*unittest*的测试，但不推荐。花点时间继承*TestCase*会让以后重构测试无比轻松。

In some cases, the existing tests may have been written using the *doctest* module. If so, *doctest* provides a DocTestSuite class that can automatically build *unittest.TestSuite* instances from the existing *doctest*-based tests.

26.4.6 跳过测试与预计的失败

3.1 新版功能.

Unittest 支持跳过单个或整组的测试用例。它还支持把测试标注成“预期失败”的测试。这些坏测试会失败，但不会算进*TestResult*的失败里。

Skipping a test is simply a matter of using the *skip() decorator* or one of its conditional variants.

跳过测试的基本用法如下:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass
```

在F嗦模式下运行以上测试例子时，程序输出如下:

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library version'
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'

----------------------------------------------------------------------
Ran 3 tests in 0.005s

OK (skipped=3)
```

跳过测试类的写法跟跳过测试方法的写法相似:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
```

(下页继续)

(续上页)

```
    def test_not_run(self):
        pass
```

TestCase.setUp() 也可以跳过测试。可以用于所需资源不可用的情况下跳过接下来的测试。

使用*expectedFailure()* 装饰器表明这个测试预计失败。:

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

It's easy to roll your own skipping decorators by making a decorator that calls *skip()* on the test when it wants it to be skipped. This decorator skips the test unless the passed object has a certain attribute:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

The following decorators implement test skipping and expected failures:

`@unittest.`**`skip`**`(`*reason*`)`
跳过被此装饰器装饰的测试。*reason* 为测试被跳过的原因。

`@unittest.`**`skipIf`**`(`*condition*, *reason*`)`
当 *condition* 为真时，跳过被装饰的测试。

`@unittest.`**`skipUnless`**`(`*condition*, *reason*`)`
跳过被装饰的测试，除非 *condition* 为真。

`@unittest.`**`expectedFailure`**
Mark the test as an expected failure. If the test fails when run, the test is not counted as a failure.

`exception` `unittest.`**`SkipTest`**`(`*reason*`)`
引发此异常以跳过一个测试。

通常来说，你可以使用*TestCase.skipTest()* 或其中一个跳过测试的装饰器实现跳过测试的功能，而不是直接引发此异常。

被跳过的测试的*setUp()* 和*tearDown()* 不会被运行。被跳过的类的*setUpClass()* 和*tearDownClass()* 不会被运行。被跳过的模组的 `setUpModule()` 和 `tearDownModule()` 不会被运行。

26.4.7 Distinguishing test iterations using subtests

3.4 新版功能.

When there are very small differences among your tests, for instance some parameters, unittest allows you to distinguish them inside the body of a test method using the *subTest()* context manager.

例如，以下测试:

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
```

(下页继续)

(续上页)

```
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

可以得到以下输出:

```
======================================================================
FAIL: test_even (__main__.NumbersTest) (i=1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=3)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=5)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

如果不使用子测试，程序遇到第一次错误之后就会停止。而且因为 “i“的值不显示，错误也更难找。

```
======================================================================
FAIL: test_even (__main__.NumbersTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

26.4.8 类与函数

本节深入介绍了*unittest* 的 API。

测试用例

class unittest.**TestCase**(*methodName='runTest'*)

Instances of the *TestCase* class represent the logical test units in the *unittest* universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the tests, and methods that the test code can use to check for and report various kinds of failure.

Each instance of *TestCase* will run a single base method: the method named *methodName*. In most uses of *TestCase*, you will neither change the *methodName* nor reimplement the default `runTest()` method.

在 3.2 版更改: *TestCase* can be instantiated successfully without providing a *methodName*. This makes it easier to experiment with *TestCase* from the interactive interpreter.

TestCase instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

第一组（用于运行测试的）方法是:

setUp()

Method called to prepare the test fixture. This is called immediately before calling the test method; other than *AssertionError* or *SkipTest*, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

tearDown()

Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than *AssertionError* or *SkipTest*, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the *setUp()* succeeds, regardless of the outcome of the test method. The default implementation does nothing.

setUpClass()

A class method called before tests in an individual class are run. `setUpClass` is called with the class as the only argument and must be decorated as a *classmethod()*:

```
@classmethod
def setUpClass(cls):
    ...
```

查看*Class and Module Fixtures* 获取更详细的说明。

3.2 新版功能.

tearDownClass()

A class method called after tests in an individual class have run. `tearDownClass` is called with the class as the only argument and must be decorated as a *classmethod()*:

```
@classmethod
def tearDownClass(cls):
    ...
```

查看*Class and Module Fixtures* 获取更详细的说明。

3.2 新版功能.

run(*result=None*)

Run the test, collecting the result into the `TestResult` object passed as *result*. If *result* is omitted or `None`, a temporary result object is created (by calling the `defaultTestResult()` method) and used. The result object is returned to `run()`'s caller.

The same effect may be had by simply calling the `TestCase` instance.

在 3.3 版更改: Previous versions of `run` did not return the result. Neither did calling an instance.

skipTest(*reason*)

Calling this during a test method or `setUp()` skips the current test. See 跳过测试与预计的失败 for more information.

3.1 新版功能.

subTest(*msg=None*, ***params*)

Return a context manager which executes the enclosed code block as a subtest. *msg* and *params* are optional, arbitrary values which are displayed whenever a subtest fails, allowing you to identify them clearly.

A test case can contain any number of subtest declarations, and they can be arbitrarily nested.

查看*Distinguishing test iterations using subtests* 获取更详细的信息。

3.4 新版功能.

debug()

Run the test without collecting the result. This allows exceptions raised by the test to be propagated to the caller, and can be used to support running tests under a debugger.

The `TestCase` class provides several assert methods to check for and report failures. The following table lists the most commonly used methods (see the tables below for more assert methods):

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

All the assert methods accept a *msg* argument that, if specified, is used as the error message on failure (see also `longMessage`). Note that the *msg* keyword argument can be passed to `assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, `assertWarnsRegex()` only when they are used as a context manager.

assertEqual(*first*, *second*, *msg=None*)

Test that *first* and *second* are equal. If the values do not compare equal, the test will fail.

In addition, if *first* and *second* are the exact same type and one of list, tuple, dict, set, frozenset or str or any type that a subclass registers with `addTypeEqualityFunc()` the type-specific equality function will be called in order to generate a more useful default error message (see also the *list of type-specific methods*).

在 3.1 版更改: Added the automatic calling of type-specific equality function.

在 3.2 版更改: *assertMultiLineEqual()* added as the default type equality function for comparing strings.

assertNotEqual(*first*, *second*, *msg=None*)
Test that *first* and *second* are not equal. If the values do compare equal, the test will fail.

assertTrue(*expr*, *msg=None*)
assertFalse(*expr*, *msg=None*)
Test that *expr* is true (or false).

Note that this is equivalent to `bool(expr) is True` and not to `expr is True` (use `assertIs(expr, True)` for the latter). This method should also be avoided when more specific methods are available (e.g. `assertEqual(a, b)` instead of `assertTrue(a == b)`), because they provide a better error message in case of failure.

assertIs(*first*, *second*, *msg=None*)
assertIsNot(*first*, *second*, *msg=None*)
Test that *first* and *second* evaluate (or don't evaluate) to the same object.

3.1 新版功能.

assertIsNone(*expr*, *msg=None*)
assertIsNotNone(*expr*, *msg=None*)
Test that *expr* is (or is not) `None`.

3.1 新版功能.

assertIn(*first*, *second*, *msg=None*)
assertNotIn(*first*, *second*, *msg=None*)
Test that *first* is (or is not) in *second*.

3.1 新版功能.

assertIsInstance(*obj*, *cls*, *msg=None*)
assertNotIsInstance(*obj*, *cls*, *msg=None*)
Test that *obj* is (or is not) an instance of *cls* (which can be a class or a tuple of classes, as supported by *isinstance()*). To check for the exact type, use *assertIs(type(obj), cls)*.

3.2 新版功能.

It is also possible to check the production of exceptions, warnings, and log messages using the following methods:

Method	Checks that	New in
*assertRaises(exc, fun, *args, **kwds)*	`fun(*args, **kwds)` raises *exc*	
*assertRaisesRegex(exc, r, fun, *args, **kwds)*	`fun(*args, **kwds)` raises *exc* and the message matches regex *r*	3.1
*assertWarns(warn, fun, *args, **kwds)*	`fun(*args, **kwds)` raises *warn*	3.2
*assertWarnsRegex(warn, r, fun, *args, **kwds)*	`fun(*args, **kwds)` raises *warn* and the message matches regex *r*	3.2
assertLogs(logger, level)	The `with` block logs on *logger* with minimum *level*	3.4

assertRaises(*exception*, *callable*, **args*, ***kwds*)
assertRaises(*exception*, *, *msg=None*)
Test that an exception is raised when *callable* is called with any positional or keyword arguments that are also passed to *assertRaises()*. The test passes if *exception* is raised, is an error if another exception

is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as *exception*.

If only the *exception* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertRaises(SomeException):
    do_something()
```

When used as a context manager, *assertRaises()* accepts the additional keyword argument *msg*.

The context manager will store the caught exception object in its `exception` attribute. This can be useful if the intention is to perform additional checks on the exception raised:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

在 3.1 版更改: Added the ability to use *assertRaises()* as a context manager.

在 3.2 版更改: Added the `exception` attribute.

在 3.3 版更改: Added the *msg* keyword argument when used as a context manager.

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)
assertRaisesRegex(*exception*, *regex*, *, *msg=None*)

Like *assertRaises()* but also tests that *regex* matches on the string representation of the raised exception. *regex* may be a regular expression object or a string containing a regular expression suitable for use by *re.search()*. Examples:

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

或者:

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

3.1 新版功能: under the name `assertRaisesRegexp`.

在 3.2 版更改: Renamed to *assertRaisesRegex()*.

在 3.3 版更改: Added the *msg* keyword argument when used as a context manager.

assertWarns(*warning*, *callable*, **args*, ***kwds*)
assertWarns(*warning*, *, *msg=None*)

Test that a warning is triggered when *callable* is called with any positional or keyword arguments that are also passed to *assertWarns()*. The test passes if *warning* is triggered and fails if it isn't. Any exception is an error. To catch any of a group of warnings, a tuple containing the warning classes may be passed as *warnings*.

If only the *warning* and possibly the *msg* arguments are given, return a context manager so that the code under test can be written inline rather than as a function:

```
with self.assertWarns(SomeWarning):
    do_something()
```

When used as a context manager, *assertWarns()* accepts the additional keyword argument *msg*.

The context manager will store the caught warning object in its `warning` attribute, and the source line which triggered the warnings in the `filename` and `lineno` attributes. This can be useful if the intention is to perform additional checks on the warning caught:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

This method works regardless of the warning filters in place when it is called.

3.2 新版功能.

在 3.3 版更改: Added the *msg* keyword argument when used as a context manager.

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)
assertWarnsRegex(*warning*, *regex*, *, *msg=None*)

Like *assertWarns()* but also tests that *regex* matches on the message of the triggered warning. *regex* may be a regular expression object or a string containing a regular expression suitable for use by *re.search()*. Example:

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

或者:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

3.2 新版功能.

在 3.3 版更改: Added the *msg* keyword argument when used as a context manager.

assertLogs(*logger=None*, *level=None*)

A context manager to test that at least one message is logged on the *logger* or one of its children, with at least the given *level*.

If given, *logger* should be a *logging.Logger* object or a *str* giving the name of a logger. The default is the root logger, which will catch all messages.

If given, *level* should be either a numeric logging level or its string equivalent (for example either `"ERROR"` or `logging.ERROR`). The default is `logging.INFO`.

The test passes if at least one message emitted inside the `with` block matches the *logger* and *level* conditions, otherwise it fails.

The object returned by the context manager is a recording helper which keeps tracks of the matching log messages. It has two attributes:

records
A list of *logging.LogRecord* objects of the matching log messages.

output
A list of *str* objects with the formatted output of matching messages.

示例:

```
with self.assertLogs('foo', level='INFO') as cm:
   logging.getLogger('foo').info('first message')
   logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

3.4 新版功能.

There are also other methods used to perform more specific checks, such as:

Method	Checks that	New in
assertAlmostEqual(a, b)	`round(a-b, 7) == 0`	
assertNotAlmostEqual(a, b)	`round(a-b, 7) != 0`	
assertGreater(a, b)	`a > b`	3.1
assertGreaterEqual(a, b)	`a >= b`	3.1
assertLess(a, b)	`a < b`	3.1
assertLessEqual(a, b)	`a <= b`	3.1
assertRegex(s, r)	`r.search(s)`	3.1
assertNotRegex(s, r)	`not r.search(s)`	3.2
assertCountEqual(a, b)	*a* and *b* have the same elements in the same number, regardless of their order	3.2

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)
assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)

Test that *first* and *second* are approximately (or not approximately) equal by computing the difference, rounding to the given number of decimal *places* (default 7), and comparing to zero. Note that these methods round the values to the given number of *decimal places* (i.e. like the *round()* function) and not *significant digits*.

If *delta* is supplied instead of *places* then the difference between *first* and *second* must be less or equal to (or greater than) *delta*.

Supplying both *delta* and *places* raises a *TypeError*.

在 3.2 版更改: *assertAlmostEqual()* automatically considers almost equal objects that compare equal. *assertNotAlmostEqual()* automatically fails if the objects compare equal. Added the *delta* keyword argument.

assertGreater(*first*, *second*, *msg=None*)
assertGreaterEqual(*first*, *second*, *msg=None*)
assertLess(*first*, *second*, *msg=None*)
assertLessEqual(*first*, *second*, *msg=None*)

Test that *first* is respectively >, >=, < or <= than *second* depending on the method name. If not, the test will fail:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

3.1 新版功能.

assertRegex(*text*, *regex*, *msg=None*)
assertNotRegex(*text*, *regex*, *msg=None*)

测试一个 *regex* 是否匹配 文本。如果不匹配，错误信息中将包含匹配模式和 文本 *（或部分匹配

失败的 * 文本)。*regex* 可以是正则表达式对象或能够用于*re.search()* 的包含正则表达式的字符串。

3.1 新版功能: under the name `assertRegexpMatches`.

在 3.2 版更改: 方法 `assertRegexpMatches()` 已被改名为*assertRegex()*。

3.2 新版功能: *assertNotRegex()*

3.5 新版功能: `assertNotRegexpMatches` 这个名字是*assertNotRegex()* 的已被弃用的别名。

assertCountEqual(*first*, *second*, *msg=None*)

Test that sequence *first* contains the same elements as *second*, regardless of their order. When they don't, an error message listing the differences between the sequences will be generated.

Duplicate elements are *not* ignored when comparing *first* and *second*. It verifies whether each element has the same count in both sequences. Equivalent to: `assertEqual(Counter(list(first)), Counter(list(second)))` but works with sequences of unhashable objects as well.

3.2 新版功能.

The *assertEqual()* method dispatches the equality check for objects of the same type to different type-specific methods. These methods are already implemented for most of the built-in types, but it's also possible to register new methods using *addTypeEqualityFunc()*:

addTypeEqualityFunc(*typeobj*, *function*)

Registers a type-specific method called by *assertEqual()* to check if two objects of exactly the same *typeobj* (not subclasses) compare equal. *function* must take two positional arguments and a third msg=None keyword argument just as *assertEqual()* does. It must raise *self.failureException(msg)* when inequality between the first two parameters is detected –possibly providing useful information and explaining the inequalities in details in the error message.

3.1 新版功能.

以下是*assertEqual()* 自动选用的不同类型的比较方法。一般情况下不需要直接在测试中调用这些方法。

Method	用作比较	New in
assertMultiLineEqual(a, b)	字符串	3.1
assertSequenceEqual(a, b)	序列	3.1
assertListEqual(a, b)	列表	3.1
assertTupleEqual(a, b)	元组	3.1
assertSetEqual(a, b)	集合	3.1
assertDictEqual(a, b)	字典	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

Test that the multiline string *first* is equal to the string *second*. When not equal a diff of the two strings highlighting the differences will be included in the error message. This method is used by default when comparing strings with *assertEqual()*.

3.1 新版功能.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

Tests that two sequences are equal. If a *seq_type* is supplied, both *first* and *second* must be instances of *seq_type* or a failure will be raised. If the sequences are different an error message is constructed that shows the difference between the two.

This method is not called directly by *assertEqual()*, but it's used to implement *assertListEqual()* and *assertTupleEqual()*.

3.1 新版功能.

assertListEqual(*first*, *second*, *msg=None*)
assertTupleEqual(*first*, *second*, *msg=None*)
Tests that two lists or tuples are equal. If not, an error message is constructed that shows only the differences between the two. An error is also raised if either of the parameters are of the wrong type. These methods are used by default when comparing lists or tuples with *assertEqual()*.

3.1 新版功能.

assertSetEqual(*first*, *second*, *msg=None*)
Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets. This method is used by default when comparing sets or frozensets with *assertEqual()*.

Fails if either of *first* or *second* does not have a `set.difference()` method.

3.1 新版功能.

assertDictEqual(*first*, *second*, *msg=None*)
Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries. This method will be used by default to compare dictionaries in calls to *assertEqual()*.

3.1 新版功能.

Finally the *TestCase* provides the following methods and attributes:

fail(*msg=None*)
Signals a test failure unconditionally, with *msg* or `None` for the error message.

failureException
This class attribute gives the exception raised by the test method. If a test framework needs to use a specialized exception, possibly to carry additional information, it must subclass this exception in order to "play fair" with the framework. The initial value of this attribute is *AssertionError*.

longMessage
This class attribute determines what happens when a custom failure message is passed as the msg argument to an assertXYY call that fails. `True` is the default value. In this case, the custom message is appended to the end of the standard failure message. When set to `False`, the custom message replaces the standard message.

The class setting can be overridden in individual test methods by assigning an instance attribute, self.longMessage, to `True` or `False` before calling the assert methods.

The class setting gets reset before each test call.

3.1 新版功能.

maxDiff
This attribute controls the maximum length of diffs output by assert methods that report diffs on failure. It defaults to 80*8 characters. Assert methods affected by this attribute are *assertSequenceEqual()* (including all the sequence comparison methods that delegate to it), *assertDictEqual()* and *assertMultiLineEqual()*.

Setting `maxDiff` to `None` means that there is no maximum length of diffs.

3.2 新版功能.

Testing frameworks can use the following methods to collect information on the test:

countTestCases()
Return the number of tests represented by this test object. For *TestCase* instances, this will always be `1`.

defaultTestResult()
Return an instance of the test result class that should be used for this test case class (if no other result instance is provided to the *run()* method).

For *TestCase* instances, this will always be an instance of *TestResult*; subclasses of *TestCase* should override this as necessary.

id()
Return a string identifying the specific test case. This is usually the full name of the test method, including the module and class name.

shortDescription()
Returns a description of the test, or `None` if no description has been provided. The default implementation of this method returns the first line of the test method's docstring, if available, or `None`.

在 3.1 版更改: In 3.1 this was changed to add the test name to the short description even in the presence of a docstring. This caused compatibility issues with unittest extensions and adding the test name was moved to the *TextTestResult* in Python 3.2.

addCleanup(*function*, **args*, ***kwargs*)
Add a function to be called after *tearDown()* to cleanup resources used during the test. Functions will be called in reverse order to the order they are added (LIFO). They are called with any arguments and keyword arguments passed into *addCleanup()* when they are added.

If *setUp()* fails, meaning that *tearDown()* is not called, then any cleanup functions added will still be called.

3.1 新版功能.

doCleanups()
This method is called unconditionally after *tearDown()*, or after *setUp()* if *setUp()* raises an exception.

It is responsible for calling all the cleanup functions added by *addCleanup()*. If you need cleanup functions to be called *prior* to *tearDown()* then you can call *doCleanups()* yourself.

doCleanups() pops methods off the stack of cleanup functions one at a time, so it can be called at any time.

3.1 新版功能.

class unittest.**FunctionTestCase**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)
This class implements the portion of the *TestCase* interface which allows the test runner to drive the test, but does not provide the methods which test code can use to check and report errors. This is used to create test cases using legacy test code, allowing it to be integrated into a *unittest*-based test framework.

Deprecated aliases

For historical reasons, some of the *TestCase* methods had one or more aliases that are now deprecated. The following table lists the correct names along with their deprecated aliases:

方法名	Deprecated alias	Deprecated alias
assertEqual()	failUnlessEqual	assertEquals
assertNotEqual()	failIfEqual	assertNotEquals
assertTrue()	failUnless	assert_
assertFalse()	failIf	
assertRaises()	failUnlessRaises	
assertAlmostEqual()	failUnlessAlmostEqual	assertAlmostEquals
assertNotAlmostEqual()	failIfAlmostEqual	assertNotAlmostEquals
assertRegex()		assertRegexpMatches
assertNotRegex()		assertNotRegexpMatches
assertRaisesRegex()		assertRaisesRegexp

3.1 版后已移除: the fail* aliases listed in the second column.

3.2 版后已移除: the assert* aliases listed in the third column.

3.2 版后已移除: `assertRegexpMatches` and `assertRaisesRegexp` have been renamed to *`assertRegex()`* and *`assertRaisesRegex()`*.

3.5 版后已移除: the `assertNotRegexpMatches` name in favor of *`assertNotRegex()`*.

Grouping tests

class unittest.**TestSuite**(*tests=()*)

This class represents an aggregation of individual test cases and test suites. The class presents the interface needed by the test runner to allow it to be run as any other test case. Running a *`TestSuite`* instance is the same as iterating over the suite, running each test individually.

If *tests* is given, it must be an iterable of individual test cases or other test suites that will be used to build the suite initially. Additional methods are provided to add test cases and suites to the collection later on.

`TestSuite` objects behave much like *`TestCase`* objects, except they do not actually implement a test. Instead, they are used to aggregate tests into groups of tests that should be run together. Some additional methods are available to add tests to *`TestSuite`* instances:

addTest(*test*)

Add a *`TestCase`* or *`TestSuite`* to the suite.

addTests(*tests*)

Add all the tests from an iterable of *`TestCase`* and *`TestSuite`* instances to this test suite.

This is equivalent to iterating over *tests*, calling *`addTest()`* for each element.

`TestSuite` shares the following methods with *`TestCase`*:

run(*result*)

Run the tests associated with this suite, collecting the result into the test result object passed as *result*. Note that unlike *`TestCase.run()`*, *`TestSuite.run()`* requires the result object to be passed in.

debug()

Run the tests associated with this suite without collecting the result. This allows exceptions raised by the test to be propagated to the caller and can be used to support running tests under a debugger.

countTestCases()

Return the number of tests represented by this test object, including all individual tests and sub-suites.

__iter__()

Tests grouped by a *`TestSuite`* are always accessed by iteration. Subclasses can lazily provide tests by overriding *`__iter__()`*. Note that this method may be called several times on a single suite (for example when counting tests or comparing for equality) so the tests returned by repeated iterations before *`TestSuite.run()`* must be the same for each call iteration. After *`TestSuite.run()`*, callers should not rely on the tests returned by this method unless the caller uses a subclass that overrides `TestSuite._removeTestAtIndex()` to preserve test references.

在 3.2 版更改: In earlier versions the *`TestSuite`* accessed tests directly rather than through iteration, so overriding *`__iter__()`* wasn't sufficient for providing tests.

在 3.4 版更改: In earlier versions the *`TestSuite`* held references to each *`TestCase`* after *`TestSuite.run()`*. Subclasses can restore that behavior by overriding `TestSuite._removeTestAtIndex()`.

In the typical usage of a *`TestSuite`* object, the *`run()`* method is invoked by a `TestRunner` rather than by the end-user test harness.

Loading and running tests

class unittest.**TestLoader**

The *TestLoader* class is used to create test suites from classes and modules. Normally, there is no need to create an instance of this class; the *unittest* module provides an instance that can be shared as *unittest.defaultTestLoader*. Using a subclass or instance, however, allows customization of some configurable properties.

TestLoader objects have the following attributes:

errors

A list of the non-fatal errors encountered while loading tests. Not reset by the loader at any point. Fatal errors are signalled by the relevant a method raising an exception to the caller. Non-fatal errors are also indicated by a synthetic test that will raise the original error when run.

3.5 新版功能.

TestLoader objects have the following methods:

loadTestsFromTestCase(*testCaseClass*)

Return a suite of all test cases contained in the *TestCase*-derived `testCaseClass`.

A test case instance is created for each method named by *getTestCaseNames()*. By default these are the method names beginning with `test`. If *getTestCaseNames()* returns no methods, but the `runTest()` method is implemented, a single test case is created for that method instead.

loadTestsFromModule(*module*, *pattern=None*)

Return a suite of all test cases contained in the given module. This method searches *module* for classes derived from *TestCase* and creates an instance of the class for each test method defined for the class.

注解: While using a hierarchy of *TestCase*-derived classes can be convenient in sharing fixtures and helper functions, defining test methods on base classes that are not intended to be instantiated directly does not play well with this method. Doing so, however, can be useful when the fixtures are different and defined in subclasses.

If a module provides a `load_tests` function it will be called to load the tests. This allows modules to customize test loading. This is the *load_tests protocol*. The *pattern* argument is passed as the third argument to `load_tests`.

在 3.2 版更改: Support for `load_tests` added.

在 3.5 版更改: The undocumented and unofficial *use_load_tests* default argument is deprecated and ignored, although it is still accepted for backward compatibility. The method also now accepts a keyword-only argument *pattern* which is passed to `load_tests` as the third argument.

loadTestsFromName(*name*, *module=None*)

Return a suite of all test cases given a string specifier.

The specifier *name* is a "dotted name" that may resolve either to a module, a test case class, a test method within a test case class, a *TestSuite* instance, or a callable object which returns a *TestCase* or *TestSuite* instance. These checks are applied in the order listed here; that is, a method on a possible test case class will be picked up as "a test method within a test case class", rather than "a callable object".

For example, if you have a module `SampleTests` containing a *TestCase*-derived class `SampleTestCase` with three test methods (`test_one()`, `test_two()`, and `test_three()`), the specifier `'SampleTests.SampleTestCase'` would cause this method to return a suite which will run all three test methods. Using the specifier `'SampleTests.SampleTestCase.test_two'` would cause it to return a test suite which will run only the `test_two()` test method. The specifier can refer to modules and packages which have not been imported; they will be imported as a side-effect.

The method optionally resolves *name* relative to the given *module*.

在 3.5 版更改: If an `ImportError` or `AttributeError` occurs while traversing *name* then a synthetic test that raises that error when run will be returned. These errors are included in the errors accumulated by self.errors.

loadTestsFromNames(*names*, *module=None*)

Similar to `loadTestsFromName()`, but takes a sequence of names rather than a single name. The return value is a test suite which supports all the tests defined for each name.

getTestCaseNames(*testCaseClass*)

Return a sorted sequence of method names found within *testCaseClass*; this should be a subclass of `TestCase`.

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)

Find all the test modules by recursing into subdirectories from the specified start directory, and return a TestSuite object containing them. Only test files that match *pattern* will be loaded. (Using shell style pattern matching.) Only module names that are importable (i.e. are valid Python identifiers) will be loaded.

All test modules must be importable from the top level of the project. If the start directory is not the top level directory then the top level directory must be specified separately.

If importing a module fails, for example due to a syntax error, then this will be recorded as a single error and discovery will continue. If the import failure is due to `SkipTest` being raised, it will be recorded as a skip instead of an error.

If a package (a directory containing a file named `__init__.py`) is found, the package will be checked for a `load_tests` function. If this exists then it will be called `package.load_tests(loader, tests, pattern)`. Test discovery takes care to ensure that a package is only checked for tests once during an invocation, even if the load_tests function itself calls `loader.discover`.

If `load_tests` exists then discovery does *not* recurse into the package, `load_tests` is responsible for loading all tests in the package.

The pattern is deliberately not stored as a loader attribute so that packages can continue discovery themselves. *top_level_dir* is stored so `load_tests` does not need to pass this argument in to `loader.discover()`.

start_dir can be a dotted module name as well as a directory.

3.2 新版功能.

在 3.4 版更改: Modules that raise `SkipTest` on import are recorded as skips, not errors. Discovery works for *namespace packages*. Paths are sorted before being imported so that execution order is the same even if the underlying file system's ordering is not dependent on file name.

在 3.5 版更改: Found packages are now checked for `load_tests` regardless of whether their path matches *pattern*, because it is impossible for a package name to match the default pattern.

The following attributes of a `TestLoader` can be configured either by subclassing or assignment on an instance:

testMethodPrefix

String giving the prefix of method names which will be interpreted as test methods. The default value is `'test'`.

This affects `getTestCaseNames()` and all the `loadTestsFrom*()` methods.

sortTestMethodsUsing

Function to be used to compare method names when sorting them in `getTestCaseNames()` and all the `loadTestsFrom*()` methods.

suiteClass
Callable object that constructs a test suite from a list of tests. No methods on the resulting object are needed. The default value is the *TestSuite* class.

This affects all the `loadTestsFrom*()` methods.

class unittest.**TestResult**
This class is used to compile information about which tests have succeeded and which have failed.

A *TestResult* object stores the results of a set of tests. The *TestCase* and *TestSuite* classes ensure that results are properly recorded; test authors do not need to worry about recording the outcome of tests.

Testing frameworks built on top of *unittest* may want access to the *TestResult* object generated by running a set of tests for reporting purposes; a *TestResult* instance is returned by the `TestRunner.run()` method for this purpose.

TestResult instances have the following attributes that will be of interest when inspecting the results of running a set of tests:

errors
A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.

failures
A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the `TestCase.assert*()` methods.

skipped
A list containing 2-tuples of *TestCase* instances and strings holding the reason for skipping the test.

3.1 新版功能.

expectedFailures
A list containing 2-tuples of *TestCase* instances and strings holding formatted tracebacks. Each tuple represents an expected failure of the test case.

unexpectedSuccesses
A list containing *TestCase* instances that were marked as expected failures, but succeeded.

shouldStop
Set to `True` when the execution of tests should stop by *stop()*.

testsRun
The total number of tests run so far.

buffer
If set to true, `sys.stdout` and `sys.stderr` will be buffered in between *startTest()* and *stopTest()* being called. Collected output will only be echoed onto the real `sys.stdout` and `sys.stderr` if the test fails or errors. Any output is also attached to the failure / error message.

3.2 新版功能.

failfast
If set to true *stop()* will be called on the first failure or error, halting the test run.

3.2 新版功能.

tb_locals
If set to true then local variables will be shown in tracebacks.

3.5 新版功能.

wasSuccessful()
Return `True` if all tests run so far have passed, otherwise returns `False`.

在 3.4 版更改: Returns `False` if there were any *unexpectedSuccesses* from tests marked with the *expectedFailure()* decorator.

stop()
This method can be called to signal that the set of tests being run should be aborted by setting the *shouldStop* attribute to `True`. `TestRunner` objects should respect this flag and return without running any additional tests.

For example, this feature is used by the *TextTestRunner* class to stop the test framework when the user signals an interrupt from the keyboard. Interactive tools which provide `TestRunner` implementations can use this in a similar manner.

The following methods of the *TestResult* class are used to maintain the internal data structures, and may be extended in subclasses to support additional reporting requirements. This is particularly useful in building tools which support interactive reporting while tests are being run.

startTest(*test*)
Called when the test case *test* is about to be run.

stopTest(*test*)
Called after the test case *test* has been executed, regardless of the outcome.

startTestRun()
Called once before any tests are executed.

3.1 新版功能.

stopTestRun()
Called once after all tests are executed.

3.1 新版功能.

addError(*test*, *err*)
Called when the test case *test* raises an unexpected exception. *err* is a tuple of the form returned by *sys.exc_info()*: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *errors* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addFailure(*test*, *err*)
Called when the test case *test* signals a failure. *err* is a tuple of the form returned by *sys.exc_info()*: `(type, value, traceback)`.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *failures* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addSuccess(*test*)
Called when the test case *test* succeeds.

The default implementation does nothing.

addSkip(*test*, *reason*)
Called when the test case *test* is skipped. *reason* is the reason the test gave for skipping.

The default implementation appends a tuple `(test, reason)` to the instance's *skipped* attribute.

addExpectedFailure(*test*, *err*)
Called when the test case *test* fails, but was marked with the *expectedFailure()* decorator.

The default implementation appends a tuple `(test, formatted_err)` to the instance's *expectedFailures* attribute, where *formatted_err* is a formatted traceback derived from *err*.

addUnexpectedSuccess(*test*)
Called when the test case *test* was marked with the *expectedFailure()* decorator, but succeeded.

The default implementation appends the test to the instance's *unexpectedSuccesses* attribute.

addSubTest(*test*, *subtest*, *outcome*)

Called when a subtest finishes. *test* is the test case corresponding to the test method. *subtest* is a custom *TestCase* instance describing the subtest.

If *outcome* is *None*, the subtest succeeded. Otherwise, it failed with an exception where *outcome* is a tuple of the form returned by *sys.exc_info()*: `(type, value, traceback)`.

The default implementation does nothing when the outcome is a success, and records subtest failures as normal failures.

3.4 新版功能.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*)

A concrete implementation of *TestResult* used by the *TextTestRunner*.

3.2 新版功能: This class was previously named `_TextTestResult`. The old name still exists as an alias but is deprecated.

unittest.**defaultTestLoader**

Instance of the *TestLoader* class intended to be shared. If no customization of the *TestLoader* is needed, this instance can be used instead of repeatedly creating new instances.

class unittest.**TextTestRunner**(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, ***, *tb_locals=False*)

A basic test runner implementation that outputs results to a stream. If *stream* is `None`, the default, *sys.stderr* is used as the output stream. This class has a few configurable parameters, but is essentially very simple. Graphical applications which run test suites should provide alternate implementations. Such implementations should accept `**kwargs` as the interface to construct runners changes when features are added to unittest.

By default this runner shows *DeprecationWarning*, *PendingDeprecationWarning*, *ResourceWarning* and *ImportWarning* even if they are *ignored by default*. Deprecation warnings caused by *deprecated unittest methods* are also special-cased and, when the warning filters are `'default'` or `'always'`, they will appear only once per-module, in order to avoid too many warning messages. This behavior can be overridden using Python's `-Wd` or `-Wa` options (see Warning control) and leaving *warnings* to `None`.

在 3.2 版更改: Added the `warnings` argument.

在 3.2 版更改: The default stream is set to *sys.stderr* at instantiation time rather than import time.

在 3.5 版更改: Added the tb_locals parameter.

_makeResult()

This method returns the instance of `TestResult` used by *run()*. It is not intended to be called directly, but can be overridden in subclasses to provide a custom `TestResult`.

`_makeResult()` instantiates the class or callable passed in the `TextTestRunner` constructor as the `resultclass` argument. It defaults to *TextTestResult* if no `resultclass` is provided. The result class is instantiated with the following arguments:

```
stream, descriptions, verbosity
```

run(*test*)

This method is the main public interface to the *TextTestRunner*. This method takes a *TestSuite* or *TestCase* instance. A *TestResult* is created by calling *_makeResult()* and the test(s) are run and the results printed to stdout.

unittest.**main**(*module='__main__'*, *defaultTest=None*, *argv=None*, *testRunner=None*, *testLoader=unittest.defaultTestLoader*, *exit=True*, *verbosity=1*, *failfast=None*, *catchbreak=None*, *buffer=None*, *warnings=None*)

A command-line program that loads a set of tests from *module* and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

```
if __name__ == '__main__':
    unittest.main()
```

You can run tests with more detailed information by passing in the verbosity argument:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

The *defaultTest* argument is either the name of a single test or an iterable of test names to run if no test names are specified via *argv*. If not specified or `None` and no test names are provided via *argv*, all tests found in *module* are run.

The *argv* argument can be a list of options passed to the program, with the first element being the program name. If not specified or `None`, the values of *sys.argv* are used.

The *testRunner* argument can either be a test runner class or an already created instance of it. By default `main` calls *sys.exit()* with an exit code indicating success or failure of the tests run.

The *testLoader* argument has to be a *TestLoader* instance, and defaults to *defaultTestLoader*.

`main` supports being used from the interactive interpreter by passing in the argument `exit=False`. This displays the result on standard output without calling *sys.exit()*:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

The *failfast*, *catchbreak* and *buffer* parameters have the same effect as the same-name *command-line options*.

The *warnings* argument specifies the *warning filter* that should be used while running the tests. If it's not specified, it will remain `None` if a `-W` option is passed to **python** (see Warning control), otherwise it will be set to `'default'`.

Calling `main` actually returns an instance of the `TestProgram` class. This stores the result of the tests run as the `result` attribute.

在 3.1 版更改: The *exit* parameter was added.

在 3.2 版更改: The *verbosity*, *failfast*, *catchbreak*, *buffer* and *warnings* parameters were added.

在 3.4 版更改: The *defaultTest* parameter was changed to also accept an iterable of test names.

load_tests Protocol

3.2 新版功能.

Modules or packages can customize how tests are loaded from them during normal test runs or test discovery by implementing a function called `load_tests`.

If a test module defines `load_tests` it will be called by *TestLoader.loadTestsFromModule()* with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

where *pattern* is passed straight through from `loadTestsFromModule`. It defaults to `None`.

It should return a *TestSuite*.

loader is the instance of *TestLoader* doing the loading. *standard_tests* are the tests that would be loaded by default from the module. It is common for test modules to only want to add or remove tests from the standard set of tests. The third argument is used when loading packages as part of test discovery.

A typical `load_tests` function that loads tests from a specific set of *TestCase* classes may look like:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

If discovery is started in a directory containing a package, either from the command line or by calling *TestLoader.discover()*, then the package `__init__.py` will be checked for `load_tests`. If that function does not exist, discovery will recurse into the package as though it were just another directory. Otherwise, discovery of the package's tests will be left up to `load_tests` which is called with the following arguments:

```
load_tests(loader, standard_tests, pattern)
```

This should return a *TestSuite* representing all the tests from the package. (`standard_tests` will only contain tests collected from `__init__.py`.)

Because the pattern is passed into `load_tests` the package is free to continue (and potentially modify) test discovery. A 'do nothing' `load_tests` function for a test package would look like:

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

在 3.5 版更改: Discovery no longer checks package names for matching *pattern* due to the impossibility of package names matching the default pattern.

26.4.9 Class and Module Fixtures

Class and module level fixtures are implemented in *TestSuite*. When the test suite encounters a test from a new class then `tearDownClass()` from the previous class (if there is one) is called, followed by `setUpClass()` from the new class.

Similarly if a test is from a different module from the previous test then `tearDownModule` from the previous module is run, followed by `setUpModule` from the new module.

After all the tests have run the final `tearDownClass` and `tearDownModule` are run.

Note that shared fixtures do not play well with [potential] features like test parallelization and they break test isolation. They should be used with care.

The default ordering of tests created by the unittest test loaders is to group all tests from the same modules and classes together. This will lead to `setUpClass` / `setUpModule` (etc) being called exactly once per class and module. If you randomize the order, so that tests from different modules and classes are adjacent to each other, then these shared fixture functions may be called multiple times in a single test run.

Shared fixtures are not intended to work with suites with non-standard ordering. A `BaseTestSuite` still exists for frameworks that don't want to support shared fixtures.

If there are any exceptions raised during one of the shared fixture functions the test is reported as an error. Because there is no corresponding test instance an `_ErrorHolder` object (that has the same interface as a *TestCase*) is created to represent the error. If you are just using the standard unittest test runner then this detail doesn't matter, but if you are a framework author it may be relevant.

setUpClass and tearDownClass

These must be implemented as class methods:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

If you want the `setUpClass` and `tearDownClass` on base classes called then you must call up to them yourself. The implementations in *TestCase* are empty.

If an exception is raised during a `setUpClass` then the tests in the class are not run and the `tearDownClass` is not run. Skipped classes will not have `setUpClass` or `tearDownClass` run. If the exception is a *SkipTest* exception then the class will be reported as having been skipped instead of as an error.

setUpModule and tearDownModule

These should be implemented as functions:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

If an exception is raised in a `setUpModule` then none of the tests in the module will be run and the `tearDownModule` will not be run. If the exception is a *SkipTest* exception then the module will be reported as having been skipped instead of as an error.

26.4.10 信号处理

3.2 新版功能.

The *-c/--catch* command-line option to unittest, along with the `catchbreak` parameter to *unittest.main()*, provide more friendly handling of control-C during a test run. With catch break behavior enabled control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a *KeyboardInterrupt* in the usual way.

The control-c handling signal handler attempts to remain compatible with code or tests that install their own `signal.SIGINT` handler. If the `unittest` handler is called but *isn't* the installed `signal.SIGINT` handler, i.e. it has been replaced by the system under test and delegated to, then it calls the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need `unittest` control-c handling disabled the *removeHandler()* decorator can be used.

There are a few utility functions for framework authors to enable control-c handling functionality within test frameworks.

unittest.**installHandler**()
: Install the control-c handler. When a `signal.SIGINT` is received (usually in response to the user pressing control-c) all registered results have *stop()* called.

unittest.**registerResult**(*result*)
: Register a *TestResult* object for control-c handling. Registering a result stores a weak reference to it, so it doesn't prevent the result from being garbage collected.

  Registering a *TestResult* object has no side-effects if control-c handling is not enabled, so test frameworks can unconditionally register all results they create independently of whether or not handling is enabled.

unittest.**removeResult**(*result*)
: Remove a registered result. Once a result has been removed then *stop()* will no longer be called on that result object in response to a control-c.

unittest.**removeHandler**(*function=None*)
: When called without arguments this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler while the test is being executed:

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

26.5 unittest.mock —模拟对象库

3.3 新版功能.

源代码： Lib/unittest/mock.py

---

unittest.mock 是一个用于测试的 Python 库。它允许使用模拟对象来替换受测系统的部分，并对它们如何已经被使用进行断言。

unittest.mock 提供了一个核心类*Mock* 用于消除了在整个测试套件中创建大量存根 (stub) 的需求。创建后，就可以断言调用了哪些方法/属性及其参数。还可以以常规方式指定返回值并设置所需的属性。

此外，mock 提供了用于修补测试范围内模块和类级别属性的*patch()* 装饰器，和用于创建独特对象的*sentinel* 。阅读*quick guide* 中的案例了解如何使用*Mock* ，*MagicMock* 和*patch()* 。

Mock 是为*unittest* 而设计，且简单易用。模拟基于‘action -> assertion’模式，而不是许多模拟框架所使用的‘record -> replay’模式。

在 Python 的早期版本要单独使用*unittest.mock* ，在 PyPI 获取 mock

26.5.1 快速上手

当您访问对象时，*Mock* 和*MagicMock* 将创建所有属性和方法，并保存他们在使用时的细节。你可以通过配置，指定返回值或者限制可访问属性，然后断言他们如何被调用。

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

通过 side_effect 设置副作用 (side effects)，可以是一个 mock 被调用是抛出的异常

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

Mock 还可以通过其他方法配置和控制其行为。例如 mock 可以通过设置 *spec* 参数来从一个对象中获取其规格 (specification)。如果访问 mock 的属性或方法不在 spec 中，会报*AttributeError* 错误。

The *patch()* decorator / context manager makes it easy to mock classes or objects in a module under test. The object you specify will be replaced with a mock (or other object) during the test and restored when the test ends:

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

注解： When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *python* order that decorators are applied). This means from the bottom up, so in the example above the mock for module.ClassName1 is passed in first.

在查找对象的名称空间中修补对象使用*patch()* 。使用起来很简单，阅读*在哪里打补丁* 来快速上手。

---

patch() 也可以 with 语句中使用上下文管理。

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

还有一个*patch.dict()* 用于在一定范围内设置字典中的值，并在测试结束时将字典恢复为其原始状态：

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock 支持 Python 魔术方法 。使用模式方法最简单的方式是使用*MagicMock* class. 。它可以做如下事情：

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock 能指定函数（或其他 Mock 实例）为魔术方法，它们将被适当地调用。*MagicMock* 是预先创建了所有魔术方法（所有有用的方法）的 Mock 。

下面是一个使用了普通 Mock 类的魔术方法的例子

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

使用*auto-speccing* 可以保证测试中的模拟对象与要替换的对象具有相同的 api 。在 patch 中可以通过 *autospec* 参数实现自动推断，或者使用*create_autospec()* 函数。自动推断会创建一个与要替换对象相同的属相和方法的模拟对象，并且任何函数和方法（包括构造函数）都具有与真实对象相同的调用签名。

这么做是为了因确保不当地使用 mock 导致与生产代码相同的失败：

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
 ...
TypeError: <lambda>() takes exactly 3 arguments (1 given)
```

在类中使用*create_autospec()* 时，会复制 __init__ 的签名，另外在可调用对象上使用时，会复制 __call__ 的签名。

26.5.2 Mock 类

Mock 是一个可以灵活的替换存根 (stubs) 的对象，可以测试所有代码。Mock 是可调用的，在访问其属性时创建一个新的 mock[1] 。访问相同的属性只会返回相同的 mock 。Mock 保存调用记录，可以通过断言获悉代码的调用。

MagicMock 是*Mock* 的子类，它有所有预创建且可使用的魔术方法。在需要模拟不可调用对象时，可以使用*NonCallableMock* 和*NonCallableMagicMock*

patch() 装饰器使得用*Mock* 对象临时替换特定模块中的类非常方便。默认情况下*patch()* 将为你创建一个*MagicMock*。你可以使用*patch()* 的 *new_callable* 参数指定替代*Mock* 的类。

class unittest.mock.**Mock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

创建一个新的*Mock* 对象。通过可选参数指定*Mock* 对象的行为：

- *spec*: 可以是要给字符串列表，也可以是充当模拟对象规范的现有对象（类或实例）。如果传入一个对象，则通过在该对象上调用 dir 来生成字符串列表（不支持的魔法属性和方法除外）。访问不在此列表中的任何属性都将引发*AttributeError* 。

  如果 *spec* 是一个对象（而不是字符串列表），则*__class__* 返回 *spec* 对象的类。这允许模拟程序通过*isinstance()* 测试。

- *spec_set*： *spec* 的更严格的变体。如果使用了该属性，尝试模拟 *set* 或 *get* 的属性不在 *spec_set* 所包含的对象中时，会抛出*AttributeError* 。

- *side_effect*： 每当调用 Mock 时都会调用的函数。参见*side_effect* 属性。对于引发异常或动态更改返回值很有用。该函数使用与 mock 函数相同的参数调用，并且除非返回*DEFAULT* ，否则该函数的返回值将用作返回值。

  另外，*side_effect* 可以是异常类或实例。此时，调用模拟程序时将引发异常。

  如果 *side_effect* 是可迭代对象，则每次调用 mock 都将返回可迭代对象的下一个值。

  设置 *side_effect* 为 `None` 即可清空。

- *return_value*： 调用 mock 的返回值。默认情况下，是一个新的 Mock（在首次访问时创建）。参见*return_value* 属性。

- *unsafe*： 默认情况下，如果任何以 *assert* 或 *assret* 开头的属性都将引发*AttributeError* 。当 `unsafe=True` 时可以访问。

  3.5 新版功能.

- *wraps*： 要包装的 mock 对象。如果 *wraps* 不是 `None` ，那么调用 Mock 会将调用传递给 *wraps* 的对象（返回实际结果）。对模拟的属性访问将返回一个 Mock 对象，该对象包装了 *wraps* 对象的相应属性（因此，尝试访问不存在的属性将引发*AttributeError* ）。

  如果该 mock 明确指定 *return_value* ，调用是，不会返回包装对象，而是返回 *return_value* 。

- *name*： mock 的名称。在调试时很有用。名称会传递到子 mock 。

还可以使用任意关键字参数来调用 mock 。创建模拟后，将使用这些属性来设置 mock 的属性。有关详细信息，请参见*configure_mock()* 方法。

assert_called(**args*, ***kwargs*)

断言该 mock 至少被调用过一次。

[1] The only exceptions are magic methods and attributes (those that have leading and trailing double underscores). Mock doesn't create these but instead raises an *AttributeError*. This is because the interpreter will often implicitly request these methods, and gets *very* confused to get a new Mock object when it expects a magic method. If you need magic method support see *magic methods*.

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

3.6 新版功能.

assert_called_once(**args*, ***kwargs*)
断言仅被调用一次。

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
```

3.6 新版功能.

assert_called_with(**args*, ***kwargs*)
This method is a convenient way of asserting that calls are made in a particular way:

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(**args*, ***kwargs*)
断言仅被调用一次，并且该调用是使用指定的参数进行的。

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

assert_any_call(**args*, ***kwargs*)
断言使用指定的参数调用。

The assert passes if the mock has *ever* been called, unlike *assert_called_with()* and *assert_called_once_with()* that only pass if the call is the most recent one, and in the case of *assert_called_once_with()* it must also be the only call.

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(*calls*, *any_order=False*)
assert the mock has been called with the specified calls. The *mock_calls* list is checked for the calls.

If *any_order* is false (the default) then the calls must be sequential. There can be extra calls before or after the specified calls.

If *any_order* is true then the calls can be in any order, but they must all appear in *mock_calls*.

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called()

Assert the mock was never called.

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...
AssertionError: Expected 'hello' to not have been called. Called 1 times.
```

3.5 新版功能.

reset_mock(*, *return_value=False*, *side_effect=False*)

The reset_mock method resets all the call attributes on a mock object:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

在 3.6 版更改: Added two keyword only argument to the reset_mock function.

This can be useful where you want to make a series of assertions that reuse the same object. Note that *reset_mock() doesn't* clear the return value, *side_effect* or any child attributes you have set using normal assignment by default. In case you want to reset *return_value* or *side_effect*, then pass the corresponding parameter as `True`. Child mocks and the return value mock (if any) are reset as well.

注解: *return_value*, and *side_effect* are keyword only argument.

---

mock_add_spec(*spec*, *spec_set=False*)

Add a spec to a mock. *spec* can either be an object or a list of strings. Only attributes on the *spec* can be fetched as attributes from the mock.

If *spec_set* is true then only attributes on the spec can be set.

attach_mock(*mock*, *attribute*)

Attach a mock as an attribute of this one, replacing its name and parent. Calls to the attached mock will be recorded in the *method_calls* and *mock_calls* attributes of this one.

configure_mock(***kwargs*)
Set attributes on the mock through keyword arguments.

Attributes plus return values and side effects can be set on child mocks using standard dot notation and unpacking a dictionary in the method call:

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

The same thing can be achieved in the constructor call to mocks:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

configure_mock() exists to make it easier to do configuration after the mock has been created.

__dir__()
Mock objects limit the results of `dir(some_mock)` to useful results. For mocks with a *spec* this includes all the permitted attributes for the mock.

See *FILTER_DIR* for what this filtering does, and how to switch it off.

_get_child_mock(***kw*)
Create the child mocks for attributes and return value. By default child mocks will be the same type as the parent. Subclasses of Mock may want to override this to customize the way child mocks are made.

For non-callable mocks the callable variant will be used (rather than any custom subclass).

called
A boolean representing whether or not the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count
An integer telling you how many times the mock object has been called:

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
```

(下页继续)

(续上页)

```
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

Set this to configure the value returned by calling the mock:

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

The default return value is a mock object and you can configure it in the normal way:

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

return_value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

This can either be a function to be called when the mock is called, an iterable or an exception (class or instance) to be raised.

If you pass in a function it will be called with same arguments as the mock and unless the function returns the *DEFAULT* singleton the call to the mock will then return whatever the function returns. If the function returns *DEFAULT* then the mock will return its normal value (from the *return_value*).

If you pass in an iterable, it is used to retrieve an iterator which must yield a value on every call. This value can either be an exception instance to be raised, or a value to be returned from the call to the mock (*DEFAULT* handling is identical to the function case).

An example of a mock that raises an exception (to test exception handling of an API):

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Using *side_effect* to return a sequence of values:

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

Using a callable:

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

side_effect can be set in the constructor. Here's an example that adds one to the value the mock is called with and returns it:

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

Setting *side_effect* to `None` clears it:

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
  ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

This is either `None` (if the mock hasn't been called), or the arguments that the mock was last called with. This will be in the form of a tuple: the first member is any ordered arguments the mock was called with (or an empty tuple) and the second member is any keyword arguments (or an empty dictionary).

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
```

call_args, along with members of the lists *call_args_list*, *method_calls* and *mock_calls* are *call* objects. These are tuples, so they can be unpacked to get at the individual arguments and make more complex assertions. See *calls as tuples*.

call_args_list

This is a list of all the calls made to the mock object in sequence (so the length of the list is the number of

times it has been called). Before any calls have been made it is an empty list. The *call* object can be used for conveniently constructing lists of calls to compare with *call_args_list*.

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True
```

Members of *call_args_list* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

method_calls

As well as tracking calls to themselves, mocks also track calls to methods and attributes, and *their* methods and attributes:

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

Members of *method_calls* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

mock_calls

mock_calls records *all* calls to the mock object, its methods, magic methods *and* return value mocks.

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

Members of *mock_calls* are *call* objects. These can be unpacked as tuples to get at the individual arguments. See *calls as tuples*.

注解： The way *mock_calls* are recorded means that where nested calls are made, the parameters of ancestor calls are not recorded and so will always compare equal:

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
```

(下页继续)

(续上页)

```
<MagicMock name='mock.top().bottom()' id='...'>
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

Normally the *__class__* attribute of an object will return its type. For a mock object with a `spec`, `__class__` returns the spec class instead. This allows mock objects to pass *isinstance()* tests for the object they are replacing / masquerading as:

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

__class__ is assignable to, this allows a mock to pass an *isinstance()* check without forcing you to use a spec:

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class unittest.mock.**NonCallableMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

A non-callable version of *Mock*. The constructor parameters have the same meaning of *Mock*, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

Mock objects that use a class or an instance as a `spec` or `spec_set` are able to pass *isinstance()* tests:

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

The *Mock* classes have support for mocking magic methods. See *magic methods* for the full details.

The mock classes and the *patch()* decorators all take arbitrary keyword arguments for configuration. For the *patch()* decorators the keywords are passed to the constructor of the mock being created. The keyword arguments are for configuring attributes of the mock:

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

The return value and side effect of child mocks can be set in the same way, using dotted notation. As you can't use dotted names directly in a call you have to create a dictionary and unpack it using `**`:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
```

(下页继续)

(续上页)

```
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

A callable mock which was created with a *spec* (or a *spec_set*) will introspect the specification object's signature when matching calls to the mock. Therefore, it can match the actual call's arguments regardless of whether they were passed positionally or by name:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

This applies to *assert_called_with()*, *assert_called_once_with()*, *assert_has_calls()* and *assert_any_call()*. When *Autospeccing*, it will also apply to method calls on the mock object.

在 3.4 版更改: Added signature introspection on specced and autospecced mock objects.

class unittest.mock.**PropertyMock**(**args*, ***kwargs*)

A mock intended to be used as a property, or other descriptor, on a class. *PropertyMock* provides `__get__()` and `__set__()` methods so you can specify a return value when it is fetched.

Fetching a *PropertyMock* instance from an object calls the mock, with no args. Setting it calls the mock with the value being set.

```
>>> class Foo:
...     @property
...     def foo(self):
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

Because of the way mock attributes are stored you can't directly attach a *PropertyMock* to a mock object. Instead you can attach it to the mock type object:

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
```

(下页继续)

(续上页)

```
3
>>> p.assert_called_once_with()
```

Calling

Mock objects are callable. The call will return the value set as the *return_value* attribute. The default return value is a new Mock object; it is created the first time the return value is accessed (either explicitly or by calling the Mock) - but it is stored and the same one returned each time.

Calls made to the object will be recorded in the attributes like *call_args* and *call_args_list*.

If *side_effect* is set then it will be called after the call has been recorded, so if `side_effect` raises an exception the call is still recorded.

The simplest way to make a mock raise an exception when called is to make *side_effect* an exception class or instance:

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

If `side_effect` is a function then whatever that function returns is what calls to the mock return. The `side_effect` function is called with the same arguments as the mock. This allows you to vary the return value of the call dynamically, based on the input:

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

If you want the mock to still return the default return value (a new mock), or any set return value, then there are two ways of doing this. Either return `mock.return_value` from inside `side_effect`, or return *DEFAULT*:

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
```

(下页继续)

(续上页)

```
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

To remove a `side_effect`, and return to the default behaviour, set the `side_effect` to `None`:

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

The `side_effect` can also be any iterable object. Repeated calls to the mock will return values from the iterable (until the iterable is exhausted and a *StopIteration* is raised):

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

If any members of the iterable are exceptions they will be raised instead of returned:

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

Deleting Attributes

Mock objects create attributes on demand. This allows them to pretend to be objects of any type.

You may want a mock object to return `False` to a *hasattr()* call, or raise an *AttributeError* when an attribute is fetched. You can do this by providing an object as a `spec` for a mock, but that isn't always convenient.

You "block" attributes by deleting them. Once deleted, accessing an attribute will raise an *AttributeError*.

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

Mock names and the name attribute

Since "name" is an argument to the *Mock* constructor, if you want your mock object to have a "name" attribute you can't just pass it in at creation time. There are two alternatives. One option is to use *configure_mock()*:

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

A simpler option is to simply set the "name" attribute after mock creation:

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

Attaching Mocks as Attributes

When you attach a mock as an attribute of another mock (or as the return value) it becomes a "child" of that mock. Calls to the child are recorded in the *method_calls* and *mock_calls* attributes of the parent. This is useful for configuring child mocks and then attaching them to the parent, or for attaching mocks to a parent that records all calls to the children and allows you to make assertions about the order of calls between mocks:

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

The exception to this is if the mock has a name. This allows you to prevent the "parenting" if for some reason you don't want it to happen.

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

Mocks created for you by *patch()* are automatically given names. To attach mocks that have names to a parent you use the *attach_mock()* method:

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

26.5.3 The patchers

The patch decorators are used for patching objects only within the scope of the function they decorate. They automatically handle the unpatching for you, even if exceptions are raised. All of these functions can also be used in with statements or as class decorators.

patch

注解: *patch()* is straightforward to use. The key is to do the patching in the right namespace. See the section *where to patch*.

---

unittest.mock.**patch**(*target*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch() acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the *target* is patched with a *new* object. When the function/with statement exits the patch is undone.

If *new* is omitted, then the target is replaced with a *MagicMock*. If *patch()* is used as a decorator and *new* is omitted, the created mock is passed in as an extra argument to the decorated function. If *patch()* is used as a context manager the created mock is returned by the context manager.

target should be a string in the form `'package.module.ClassName'`. The *target* is imported and the specified object replaced with the *new* object, so the *target* must be importable from the environment you are calling *patch()* from. The target is imported when the decorated function is executed, not at decoration time.

The *spec* and *spec_set* keyword arguments are passed to the *MagicMock* if patch is creating one for you.

In addition you can pass `spec=True` or `spec_set=True`, which causes patch to pass in the object being mocked as the spec/spec_set object.

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default *MagicMock* is used.

A more powerful form of *spec* is *autospec*. If you set `autospec=True` then the mock will be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a *TypeError* if they are called with the wrong signature. For mocks replacing a class, their return value (the 'instance') will have the same spec as the class. See the *create_autospec()* function and *Autospeccing*.

Instead of `autospec=True` you can pass `autospec=some_object` to use an arbitrary object as the spec instead of the one being replaced.

By default *patch()* will fail to replace attributes that don't exist. If you pass in `create=True`, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again afterwards. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

注解: 在 3.5 版更改: If you are patching builtins in a module then you don't need to pass `create=True`, it will be added by default.

Patch can be used as a `TestCase` class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. *patch()* finds tests by looking for method names that start with `patch.TEST_PREFIX`. By default this is `'test'`, which matches the way *unittest* finds tests. You can specify an alternative prefix by setting `patch.TEST_PREFIX`.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use "as" then the patched object will be bound to the name after the "as"; very useful if *patch()* is creating a mock object for you.

patch() takes arbitrary keyword arguments. These will be passed to the *Mock* (or *new_callable*) on construction.

`patch.dict(...)`, `patch.multiple(...)` and `patch.object(...)` are available for alternate use-cases.

patch() as function decorator, creating the mock for you and passing it into the decorated function:

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

Patching a class replaces the class with a *MagicMock* *instance*. If the class is instantiated in the code under test then it will be the *return_value* of the mock that will be used.

If the class is instantiated multiple times you could use *side_effect* to return a new mock each time. Alternatively you can set the *return_value* to be anything you want.

To configure return values on methods of *instances* on the patched class you must do this on the `return_value`. For example:

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
```

(下页继续)

(续上页)

```
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

If you use *spec* or *spec_set* and `patch()` is replacing a *class*, then the return value of the created mock will have the same spec.

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

The *new_callable* argument is useful where you want to use an alternative class to the default `MagicMock` for the created mock. For example, if you wanted a `NonCallableMock` to be used:

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

Another use case might be to replace an object with an `io.StringIO` instance:

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

When `patch()` is creating a mock for you, it is common that the first thing you need to do is to configure the mock. Some of that configuration can be done in the call to patch. Any arbitrary keywords you pass into the call will be used to set attributes on the created mock:

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

As well as attributes on the created mock attributes, like the `return_value` and `side_effect`, of child mocks can also be configured. These aren't syntactically valid to pass in directly as keyword arguments, but a dictionary with these as keys can still be expanded into a `patch()` call using `**`:

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

patch.object

patch.**object**(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch the named member (*attribute*) on an object (*target*) with a mock object.

`patch.object()` can be used as a decorator, class decorator or a context manager. Arguments *new*, *spec*, *create*, *spec_set*, *autospec* and *new_callable* have the same meaning as for `patch()`. Like `patch()`, `patch.object()` takes arbitrary keyword arguments for configuring the mock object it creates.

When used as a class decorator `patch.object()` honours `patch.TEST_PREFIX` for choosing which methods to wrap.

You can either call `patch.object()` with three arguments or two arguments. The three argument form takes the object to be patched, the attribute name and the object to replace the attribute with.

When calling with the two argument form you omit the replacement object, and a mock is created for you and passed in as an extra argument to the decorated function:

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

spec, *create* and the other arguments to `patch.object()` have the same meaning as they do for `patch()`.

patch.dict

patch.**dict**(*in_dict*, *values=()*, *clear=False*, ***kwargs*)

Patch a dictionary, or dictionary like object, and restore the dictionary to its original state after the test.

in_dict can be a dictionary or a mapping like container. If it is a mapping then it must at least support getting, setting and deleting items plus iterating over keys.

in_dict can also be a string specifying the name of the dictionary, which will then be fetched by importing it.

values can be a dictionary of values to set in the dictionary. *values* can also be an iterable of `(key, value)` pairs.

If *clear* is true then the dictionary will be cleared before the new values are set.

`patch.dict()` can also be called with arbitrary keyword arguments to set values in the dictionary.

`patch.dict()` can be used as a context manager, decorator or class decorator. When used as a class decorator `patch.dict()` honours `patch.TEST_PREFIX` for choosing which methods to wrap.

patch.dict() can be used to add members to a dictionary, or simply let a test change a dictionary, and ensure the dictionary is restored when the test ends.

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

Keywords can be used in the *patch.dict()* call to set values in the dictionary:

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

patch.dict() can be used with dictionary like objects that aren't actually dictionaries. At the very minimum they must support item getting, setting, deleting and either iteration or membership test. This corresponds to the magic methods `__getitem__()`, `__setitem__()`, `__delitem__()` and either `__iter__()` or `__contains__()`.

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

patch.**multiple**(*target*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

Perform multiple patches in a single call. It takes the object to be patched (either as an object or a string to fetch the object by importing) and keyword arguments for the patches:

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

Use *DEFAULT* as the value if you want *patch.multiple()* to create mocks for you. In this case the created mocks are passed into a decorated function by keyword, and a dictionary is returned when *patch.multiple()* is used as a context manager.

patch.multiple() can be used as a decorator, class decorator or a context manager. The arguments *spec*, *spec_set*, *create*, *autospec* and *new_callable* have the same meaning as for *patch()*. These arguments will be applied to *all* patches done by *patch.multiple()*.

When used as a class decorator *patch.multiple()* honours `patch.TEST_PREFIX` for choosing which methods to wrap.

If you want *patch.multiple()* to create mocks for you, then you can use *DEFAULT* as the value. If you use *patch.multiple()* as a decorator then the created mocks are passed into the decorated function by keyword.

```
>>> thing = object()
>>> other = object()
```

```
>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

patch.multiple() can be nested with other `patch` decorators, but put arguments passed by keyword *after* any of the standard arguments created by *patch()*:

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

If *patch.multiple()* is used as a context manager, the value returned by the context manger is a dictionary where created mocks are keyed by name:

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

patch methods: start and stop

All the patchers have `start()` and `stop()` methods. These make it simpler to do patching in `setUp` methods or where you want to do multiple patches without nesting decorators or with statements.

To use them call *patch()*, *patch.object()* or *patch.dict()* as normal and keep a reference to the returned `patcher` object. You can then call `start()` to put the patch in place and `stop()` to undo it.

If you are using *patch()* to create a mock for you then it will be returned by the call to `patcher.start`.

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

A typical use case for this might be for doing multiple patches in the `setUp` method of a `TestCase`:

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

警告： If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the `setUp` then `tearDown` is not called. *unittest.TestCase.addCleanup()* makes this easier:

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
```

As an added bonus you no longer need to keep a reference to the `patcher` object.

It is also possible to stop all patches which have been started by using *patch.stopall()*.

patch.**stopall**()
Stop all active patches. Only stops patches started with `start`.

patch builtins

You can patch any builtins within a module. The following example patches builtin *ord()*:

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

All of the patchers can be used as class decorators. When used in this way they wrap every test method on the class. The patchers recognise methods that start with `'test'` as being test methods. This is the same way that the *unittest.TestLoader* finds test methods by default.

It is possible that you want to use a different prefix for your tests. You can inform the patchers of the different prefix by setting `patch.TEST_PREFIX`:

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

Nesting Patch Decorators

If you want to perform multiple patches then you can simply stack up the decorators.

You can stack up multiple patch decorators using this pattern:

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
```

(下页继续)

(续上页)

```
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

Note that the decorators are applied from the bottom upwards. This is the standard way that Python applies decorators. The order of the created mocks passed into your test function matches this order.

Where to patch

patch() works by (temporarily) changing the object that a *name* points to with another one. There can be many names pointing to any individual object, so for patching to work you must ensure that you patch the name used by the system under test.

The basic principle is that you patch where an object is *looked up*, which is not necessarily the same place as where it is defined. A couple of examples will help to clarify this.

Imagine we have a project that we want to test with the following structure:

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

Now we want to test `some_function` but we want to mock out `SomeClass` using *patch()*. The problem is that when we import module b, which we will have to do then it imports `SomeClass` from module a. If we use *patch()* to mock out `a.SomeClass` then it will have no effect on our test; module b already has a reference to the *real* `SomeClass` and it looks like our patching had no effect.

The key is to patch out `SomeClass` where it is used (or where it is looked up). In this case `some_function` will actually look up `SomeClass` in module b, where we have imported it. The patching should look like:

```
@patch('b.SomeClass')
```

However, consider the alternative scenario where instead of `from a import SomeClass` module b does `import a` and `some_function` uses `a.SomeClass`. Both of these import forms are common. In this case the class we want to patch is being looked up in the module and so we have to patch `a.SomeClass` instead:

```
@patch('a.SomeClass')
```

Patching Descriptors and Proxy Objects

Both *patch* and *patch.object* correctly patch and restore descriptors: class methods, static methods and properties. You should patch these on the *class* rather than an instance. They also work with *some* objects that proxy attribute access, like the django settings object.

26.5.4 MagicMock and magic method support

Mocking Magic Methods

Mock supports mocking the Python protocol methods, also known as "magic methods". This allows mock objects to replace containers or other objects that implement Python protocols.

Because magic methods are looked up differently from normal methods[2], this support has been specially implemented. This means that only specific magic methods are supported. The supported list includes *almost* all of them. If there are any missing that you need please let us know.

You mock magic methods by setting the method you are interested in to a function or a mock instance. If you are using a function then it *must* take `self` as the first argument[3].

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

One use case for this is for mocking objects used as context managers in a `with` statement:

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

Calls to magic methods do not appear in *method_calls*, but they are recorded in *mock_calls*.

注解: If you use the *spec* keyword argument to create a mock then attempting to set a magic method that isn't in the spec will raise an *AttributeError*.

The full list of supported magic methods is:

- `__hash__`, `__sizeof__`, `__repr__` and `__str__`

[2] Magic methods *should* be looked up on the class rather than the instance. Different versions of Python are inconsistent about applying this rule. The supported protocol methods should work with all supported versions of Python.

[3] The function is basically hooked up to the class, but each `Mock` instance is kept isolated from the others.

- __dir__, __format__ and __subclasses__
- __floor__, __trunc__ and __ceil__
- Comparisons: __lt__, __gt__, __le__, __ge__, __eq__ and __ne__
- Container methods: __getitem__, __setitem__, __delitem__, __contains__, __len__, __iter__, __reversed__ and __missing__
- Context manager: __enter__ and __exit__
- Unary numeric methods: __neg__, __pos__ and __invert__
- The numeric methods (including right hand and in-place variants): __add__, __sub__, __mul__, __matmul__, __div__, __truediv__, __floordiv__, __mod__, __divmod__, __lshift__, __rshift__, __and__, __xor__, __or__, and __pow__
- Numeric conversion methods: __complex__, __int__, __float__ and __index__
- Descriptor methods: __get__, __set__ and __delete__
- Pickling: __reduce__, __reduce_ex__, __getinitargs__, __getnewargs__, __getstate__ and __setstate__

The following methods exist but are *not* supported as they are either in use by mock, can't be set dynamically, or can cause problems:

- __getattr__, __setattr__, __init__ and __new__
- __prepare__, __instancecheck__, __subclasscheck__, __del__

Magic Mock

There are two MagicMock variants: *MagicMock* and *NonCallableMagicMock*.

class unittest.mock.**MagicMock**(**args*, ***kw*)

MagicMock is a subclass of *Mock* with default implementations of most of the magic methods. You can use MagicMock without having to configure the magic methods yourself.

The constructor parameters have the same meaning as for *Mock*.

If you use the *spec* or *spec_set* arguments then *only* magic methods that exist in the spec will be created.

class unittest.mock.**NonCallableMagicMock**(**args*, ***kw*)

A non-callable version of *MagicMock*.

The constructor parameters have the same meaning as for *MagicMock*, with the exception of *return_value* and *side_effect* which have no meaning on a non-callable mock.

The magic methods are setup with *MagicMock* objects, so you can configure them and use them in the usual way:

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

By default many of the protocol methods are required to return objects of a specific type. These methods are preconfigured with a default return value, so that they can be used without you having to do anything if you aren't interested in the return value. You can still *set* the return value manually if you want to change the default.

Methods and their defaults:

- __lt__: NotImplemented
- __gt__: NotImplemented
- __le__: NotImplemented
- __ge__: NotImplemented
- __int__: 1
- __contains__: False
- __len__: 0
- __iter__: iter([])
- __exit__: False
- __complex__: 1j
- __float__: 1.0
- __bool__: True
- __index__: 1
- __hash__: default hash for the mock
- __str__: default str for the mock
- __sizeof__: default sizeof for the mock

例如:

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

The two equality methods, `__eq__()` and `__ne__()`, are special. They do the default equality comparison on identity, using the *side_effect* attribute, unless you change their return value to return something else:

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

The return value of `MagicMock.__iter__()` can be any iterable object and isn't required to be an iterator:

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

If the return value *is* an iterator, then iterating over it once will consume it and subsequent iterations will result in an empty list:

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

MagicMock has all of the supported magic methods configured except for some of the obscure and obsolete ones. You can still set these up if you want.

Magic methods that are supported but not setup by default in MagicMock are:

- __subclasses__
- __dir__
- __format__
- __get__, __set__ and __delete__
- __reversed__ and __missing__
- __reduce__, __reduce_ex__, __getinitargs__, __getnewargs__, __getstate__ and __setstate__
- __getformat__ and __setformat__

26.5.5 Helpers

sentinel

unittest.mock.**sentinel**

The sentinel object provides a convenient way of providing unique objects for your tests.

Attributes are created on demand when you access them by name. Accessing the same attribute will always return the same object. The objects returned have a sensible repr so that test failure messages are readable.

The sentinel attributes don't preserve their identity when they are *copied* or *pickled*.

Sometimes when testing you need to test that a specific object is passed as an argument to another method, or returned. It can be common to create named sentinel objects to test this. *sentinel* provides a convenient way of creating and testing the identity of objects like this.

In this example we monkey patch method to return sentinel.some_object:

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> sentinel.some_object
sentinel.some_object
```

DEFAULT

unittest.mock.**DEFAULT**
The *DEFAULT* object is a pre-created sentinel (actually sentinel.DEFAULT). It can be used by *side_effect* functions to indicate that the normal return value should be used.

call

unittest.mock.**call**(**args*, ***kwargs*)
call() is a helper object for making simpler assertions, for comparing with *call_args*, *call_args_list*, *mock_calls* and *method_calls*. *call()* can also be used with *assert_has_calls()*.

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

call.**call_list**()
For a call object that represents multiple calls, *call_list()* returns a list of all the intermediate calls as well as the final call.

call_list is particularly useful for making assertions on "chained calls". A chained call is multiple calls on a single line of code. This results in multiple entries in *mock_calls* on a mock. Manually constructing the sequence of calls can be tedious.

call_list() can construct the sequence of calls from the same chained call:

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

A call object is either a tuple of (positional args, keyword args) or (name, positional args, keyword args) depending on how it was constructed. When you construct them yourself this isn't particularly interesting, but the call objects that are in the *Mock.call_args*, *Mock.call_args_list* and *Mock.mock_calls* attributes can be introspected to get at the individual arguments they contain.

The call objects in *Mock.call_args* and *Mock.call_args_list* are two-tuples of (positional args, keyword args) whereas the call objects in *Mock.mock_calls*, along with ones you construct yourself, are three-tuples of (name, positional args, keyword args).

You can use their "tupleness" to pull out the individual arguments for more complex introspection and assertions. The positional arguments are a tuple (an empty tuple if there are no positional arguments) and the keyword arguments are a dictionary:

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
```

(下页继续)

(续上页)

```
>>> args, kwargs = kall
>>> args
(1, 2, 3)
>>> kwargs
{'arg2': 'two', 'arg': 'one'}
>>> args is kall[0]
True
>>> kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg2': 'three', 'arg': 'two'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

unittest.mock.**create_autospec**(*spec*, *spec_set=False*, *instance=False*, ***kwargs*)

Create a mock object using another object as a spec. Attributes on the mock will use the corresponding attribute on the *spec* object as their spec.

Functions or methods being mocked will have their arguments checked to ensure that they are called with the correct signature.

If *spec_set* is `True` then attempting to set attributes that don't exist on the spec object will raise an *AttributeError*.

If a class is used as a spec then the return value of the mock (the instance of the class) will have the same spec. You can use a class as the spec for an instance object by passing `instance=True`. The returned mock will only be callable if instances of the mock are callable.

create_autospec() also takes arbitrary keyword arguments that are passed to the constructor of the created mock.

See *Autospeccing* for examples of how to use auto-speccing with *create_autospec()* and the *autospec* argument to *patch()*.

ANY

unittest.mock.**ANY**

Sometimes you may need to make assertions about *some* of the arguments in a call to mock, but either not care about some of the arguments or want to pull them individually out of *call_args* and make more complex assertions on them.

To ignore certain arguments you can pass in objects that compare equal to *everything*. Calls to *assert_called_with()* and *assert_called_once_with()* will then succeed no matter what was passed in.

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

ANY can also be used in comparisons with call lists like *mock_calls*:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

FILTER_DIR

unittest.mock.**FILTER_DIR**

FILTER_DIR is a module level variable that controls the way mock objects respond to *dir()* (only for Python 2.6 or more recent). The default is `True`, which uses the filtering described below, to only show useful members. If you dislike this filtering, or need to switch it off for diagnostic purposes, then set `mock.FILTER_DIR = False`.

With filtering on, `dir(some_mock)` shows only useful attributes and will include any dynamically created attributes that wouldn't normally be shown. If the mock was created with a *spec* (or *autospec* of course) then all the attributes from the original are shown, even if they haven't been accessed yet:

```
>>> dir(Mock())
['assert_any_call',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

Many of the not-very-useful (private to *Mock* rather than the thing being mocked) underscore and double underscore prefixed attributes have been filtered from the result of calling *dir()* on a *Mock*. If you dislike this behaviour you can switch it off by setting the module level switch *FILTER_DIR*:

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...
```

Alternatively you can just use `vars(my_mock)` (instance members) and `dir(type(my_mock))` (type members) to bypass the filtering irrespective of `mock.FILTER_DIR`.

mock_open

`unittest.mock.`**`mock_open`**(*mock=None*, *read_data=None*)

A helper function to create a mock to replace the use of *open()*. It works for *open()* called directly or used as a context manager.

The *mock* argument is the mock object to configure. If `None` (the default) then a *MagicMock* will be created for you, with the API limited to methods or attributes available on standard file handles.

read_data is a string for the `read()`, *readline()*, and *readlines()* methods of the file handle to return. Calls to those methods will take data from *read_data* until it is depleted. The mock of these methods is pretty simplistic: every time the *mock* is called, the *read_data* is rewound to the start. If you need more control over the data that you are feeding to the tested code you will need to customize this mock for yourself. When that is insufficient, one of the in-memory filesystem packages on PyPI can offer a realistic filesystem for testing.

在 3.4 版更改: Added *readline()* and *readlines()* support. The mock of `read()` changed to consume *read_data* rather than returning it on each call.

在 3.5 版更改: *read_data* is now reset on each call to the *mock*.

Using *open()* as a context manager is a great way to ensure your file handles are closed properly and is becoming common:

```
with open('/some/path', 'w') as f:
    f.write('something')
```

The issue is that even if you mock out the call to *open()* it is the *returned object* that is used as a context manager (and has `__enter__()` and `__exit__()` called).

Mocking context managers with a *MagicMock* is common enough and fiddly enough that a helper function is useful.

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
```

(下页继续)

(续上页)

```
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

And for reading files:

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

Autospeccing

Autospeccing is based on the existing `spec` feature of mock. It limits the api of mocks to the api of an original object (the spec), but it is recursive (implemented lazily) so that attributes of mocks only have the same api as the attributes of the spec. In addition mocked functions / methods have the same call signature as the original so they raise a *TypeError* if they are called incorrectly.

Before I explain how auto-speccing works, here's why it is needed.

Mock is a very powerful and flexible object, but it suffers from two flaws when used to mock out objects from a system under test. One of these flaws is specific to the *Mock* api and the other is a more general problem with using mock objects.

First the problem specific to *Mock*. *Mock* has two assert methods that are extremely handy: *assert_called_with()* and *assert_called_once_with()*.

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

Because mocks auto-create attributes on demand, and allow you to call them with arbitrary arguments, if you misspell one of these assert methods then your assertion is gone:

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assret_called_once_with(4, 5, 6)
```

Your tests can pass silently and incorrectly because of the typo.

The second issue is more general to mocking. If you refactor some of your code, rename members and so on, any tests for code that is still using the *old api* but uses mocks instead of the real objects will still pass. This means your tests can all pass even though your code is broken.

Note that this is another reason why you need integration tests as well as unit tests. Testing everything in isolation is all fine and dandy, but if you don't test how your units are "wired together" there is still lots of room for bugs that tests might have caught.

`mock` already provides a feature to help with this, called speccing. If you use a class or instance as the `spec` for a mock then you can only access attributes on the mock that exist on the real class:

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

The spec only applies to the mock itself, so we still have the same issue with any methods on the mock:

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assret_called_with()
```

Auto-speccing solves this problem. You can either pass `autospec=True` to *patch()* / *patch.object()* or use the *create_autospec()* function to create a mock with a spec. If you use the `autospec=True` argument to *patch()* then the object that is being replaced will be used as the spec object. Because the speccing is done "lazily" (the spec is created as attributes on the mock are accessed) you can use it with very complex or deeply nested objects (like modules that import modules that import modules) without a big performance hit.

Here's an example of it in use:

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

You can see that `request.Request` has a spec. `request.Request` takes two arguments in the constructor (one of which is *self*). Here's what happens if we try to call it incorrectly:

```
>>> req = request.Request()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

The spec also applies to instantiated classes (i.e. the return value of specced mocks):

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

`Request` objects are not callable, so the return value of instantiating our mocked out `request.Request` is a non-callable mock. With the spec in place any typos in our asserts will raise the correct error:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

In many cases you will just be able to add `autospec=True` to your existing *patch()* calls and then be protected against bugs due to typos and api changes.

As well as using *autospec* through *patch()* there is a *create_autospec()* for creating autospecced mocks directly:

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

This isn't without caveats and limitations however, which is why it is not the default behaviour. In order to know what attributes are available on the spec object, autospec has to introspect (access attributes) the spec. As you traverse attributes on the mock a corresponding traversal of the original object is happening under the hood. If any of your specced objects have properties or descriptors that can trigger code execution then you may not be able to use autospec. On the other hand it is much better to design your objects so that introspection is safe[4].

A more serious problem is that it is common for instance attributes to be created in the `__init__()` method and not to exist on the class at all. *autospec* can't know about any dynamically created attributes and restricts the api to visible attributes.

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

There are a few different ways of resolving this problem. The easiest, but not necessarily the least annoying, way is to simply set the required attributes on the mock after creation. Just because *autospec* doesn't allow you to fetch attributes that don't exist on the spec it doesn't prevent you setting them:

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

There is a more aggressive version of both *spec* and *autospec* that *does* prevent you setting non-existent attributes. This is useful if you want to ensure your code only *sets* valid attributes too, but obviously it prevents this particular scenario:

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...   thing = Something()
...   thing.a = 33
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

Probably the best way of solving the problem is to add class attributes as default values for instance members initialised in `__init__()`. Note that if you are only setting default attributes in `__init__()` then providing them via class attributes (shared between instances of course) is faster too. e.g.

```
class Something:
    a = 33
```

[4] This only applies to classes or already instantiated objects. Calling a mocked class to create a mock instance *does not* create a real instance. It is only attribute lookups - along with calls to *dir()* - that are done.

This brings up another issue. It is relatively common to provide a default value of `None` for members that will later be an object of a different type. `None` would be useless as a spec because it wouldn't let you access *any* attributes or methods on it. As `None` is *never* going to be useful as a spec, and probably indicates a member that will normally of some other type, autospec doesn't use a spec for members that are set to `None`. These will just be ordinary mocks (well - MagicMocks):

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

If modifying your production classes to add defaults isn't to your liking then there are more options. One of these is simply to use an instance as the spec rather than the class. The other is to create a subclass of the production class and add the defaults to the subclass without affecting the production class. Both of these require you to use an alternative object as the spec. Thankfully *patch()* supports this - you can simply pass the alternative object as the *autospec* argument:

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> class SomethingForTest(Something):
...   a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

26.6 unittest.mock 上手指南

3.3 新版功能.

26.6.1 使用 mock

模拟方法调用

使用*Mock* 的常见场景：

- 模拟函数调用
- 记录“对象上的方法调用”

你可能需要替换一个对象上的方法，用于确认此方法被系统中的其他部分调用过，并且调用时使用了正确的参数。

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

使用了 mock（本例中的 `real.method`）之后，它有方法和属性可以让你针对它是被如何使用的下断言。

注解： 在多数示例中，*Mock* 与*MagicMock* 两个类可以相互替换，而 MagicMock 是一个更适用的类，通常情况下，使用它就可以了。

如果 mock 被调用，它的*called* 属性就会变成 True，更重要的是，我们可以使用*assert_called_with()* 或者*assert_called_once_with()* 方法来确认它在被调用时使用了正确的参数。

在如下的测试示例中，验证对于 ProductionClass().method 的调用会导致 something 的调用。

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

对象上的方法调用的 mock

In the last example we patched a method directly on an object to check that it was called correctly. Another common use case is to pass an object into a method (or some part of the system under test) and then check that it is used in the correct way.

The simple ProductionClass below has a closer method. If it is called with an object then it calls close on it.

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

So to test it we need to pass in an object with a close method and check that it was called correctly.

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

We don't have to do any work to provide the 'close' method on our mock. Accessing close creates it. So, if 'close' hasn't already been called then accessing it in the test will create it, but *assert_called_with()* will raise a failure exception.

Mocking Classes

A common use case is to mock out classes instantiated by your code under test. When you patch a class, then that class is replaced with a mock. Instances are created by *calling the class*. This means you access the "mock instance" by looking at the return value of the mocked class.

In the example below we have a function some_function that instantiates Foo and calls a method on it. The call to *patch()* replaces the class Foo with a mock. The Foo instance is the result of calling the mock, so it is configured by modifying the mock *return_value*.

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

Naming your mocks

It can be useful to give your mocks a name. The name is shown in the repr of the mock and can be helpful when the mock appears in test failure messages. The name is also propagated to attributes or methods of the mock:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

Tracking all Calls

Often you want to track more than a single call to a method. The *mock_calls* attribute records all calls to child attributes of the mock - and also to their children.

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

If you make an assertion about `mock_calls` and any unexpected methods have been called, then the assertion will fail. This is useful because as well as asserting that the calls you expected have been made, you are also checking that they were made in the right order and with no additional calls:

You use the *call* object to construct lists for comparing with `mock_calls`:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

However, parameters to calls that return mocks are not recorded, which means it is not possible to track nested calls where the parameters used to create ancestors are important:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

Setting Return Values and Attributes

Setting the return values on a mock object is trivially easy:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

Of course you can do the same for methods on the mock:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

The return value can also be set in the constructor:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

If you need an attribute setting on your mock, just do it:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

Sometimes you want to mock up a more complex situation, like for example `mock.connection.cursor().execute("SELECT 1")`. If we wanted this call to return a list, then we have to configure the result of the nested call.

We can use *call* to construct the set of calls in a "chained call" like this for easy assertion afterwards:

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

It is the call to `.call_list()` that turns our call object into a list of calls representing the chained calls.

Raising exceptions with mocks

A useful attribute is *side_effect*. If you set this to an exception class or instance then the exception will be raised when the mock is called.

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

Side effect functions and iterables

`side_effect` can also be set to a function or an iterable. The use case for `side_effect` as an iterable is where your mock is going to be called several times, and you want each call to return a different value. When you set `side_effect` to an iterable every call to the mock returns the next value from the iterable:

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

For more advanced use cases, like dynamically varying the return values depending on what the mock is called with, `side_effect` can be a function. The function will be called with the same arguments as the mock. Whatever the function returns is what the call returns:

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

Creating a Mock from an Existing Object

One problem with over use of mocking is that it couples your tests to the implementation of your mocks rather than your real code. Suppose you have a class that implements `some_method`. In a test for another class, you provide a mock of this object that *also* provides `some_method`. If later you refactor the first class, so that it no longer has `some_method` - then your tests will continue to pass even though your code is now broken!

Mock allows you to provide an object as a specification for the mock, using the *spec* keyword argument. Accessing methods / attributes on the mock that don't exist on your specification object will immediately raise an attribute error. If you change the implementation of your specification, then tests that use that class will start failing immediately without you having to instantiate the class in those tests.

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
```

(下页继续)

(续上页)

```
    ...
AttributeError: object has no attribute 'old_method'
```

Using a specification also enables a smarter matching of calls made to the mock, regardless of whether some parameters were passed as positional or named arguments:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

If you want this smarter matching to also work with method calls on the mock, you can use *auto-speccing*.

If you want a stronger form of specification that prevents the setting of arbitrary attributes as well as the getting of them then you can use *spec_set* instead of *spec*.

26.6.2 Patch Decorators

注解: 在查找对象的名称空间中修补对象使用*patch()*。使用起来很简单，阅读在哪里打补丁 来快速上手。

A common need in tests is to patch a class attribute or a module attribute, for example patching a builtin or patching a class in a module to test that it is instantiated. Modules and classes are effectively global, so patching on them has to be undone after the test or the patch will persist into other tests and cause hard to diagnose problems.

mock provides three convenient decorators for this: *patch()*, *patch.object()* and *patch.dict()*. patch takes a single string, of the form `package.module.Class.attribute` to specify the attribute you are patching. It also optionally takes a value that you want the attribute (or class or whatever) to be replaced with. 'patch.object' takes an object and the name of the attribute you would like patched, plus optionally the value to patch it with.

patch.object:

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original
```

```
>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

If you are patching a module (including *builtins*) then use *patch()* instead of *patch.object()*:

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
```

(下页继续)

(续上页)

```
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

The module name can be ‘dotted’, in the form `package.module` if needed:

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

A nice pattern is to actually decorate test methods themselves:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

If you want to patch with a Mock, you can use *patch()* with only one argument (or *patch.object()* with two arguments). The mock will be created for you and passed into the test function / method:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

You can stack up multiple patch decorators using this pattern:

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()
```

When you nest patch decorators the mocks are passed in to the decorated function in the same order they applied (the normal *python* order that decorators are applied). This means from the bottom up, so in the example above the mock for `test_module.ClassName2` is passed in first.

还有一个*patch.dict()* 用于在一定范围内设置字典中的值，并在测试结束时将字典恢复为其原始状态：

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
```

(下页继续)

(续上页)

```
...
>>> assert foo == original
```

patch, patch.object and patch.dict can all be used as context managers.

Where you use *patch()* to create a mock for you, you can get a reference to the mock using the "as" form of the with statement:

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

As an alternative patch, patch.object and patch.dict can be used as class decorators. When used in this way it is the same as applying the decorator individually to every method whose name starts with "test".

26.6.3 Further Examples

Here are some more examples for some slightly more advanced scenarios.

Mocking chained calls

Mocking chained calls is actually straightforward with mock once you understand the *return_value* attribute. When a mock is called for the first time, or you fetch its return_value before it has been called, a new *Mock* is created.

This means that you can see how the object returned from a call to a mocked object has been used by interrogating the return_value mock:

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

From here it is a simple step to configure and then make assertions about chained calls. Of course another alternative is writing your code in a more testable way in the first place…

So, suppose we have some code that looks a little bit like this:

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam', 'eggs
↪').start_call()
...         # more code
```

Assuming that BackendProvider is already well tested, how do we test method()? Specifically, we want to test that the code section # more code uses the response object in the correct way.

As this chain of calls is made from an instance attribute we can monkey patch the `backend` attribute on a `Something` instance. In this particular case we are only interested in the return value from the final call to `start_call` so we don't have much configuration to do. Let's assume the object it returns is 'file-like', so we'll ensure that our response object uses the builtin *open()* as its `spec`.

To do this we create a mock instance as our mock backend and create a mock response object for it. To set the response as the return value for that final `start_call` we could do this:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_
↪value = mock_response
```

We can do that in a slightly nicer way using the *configure_mock()* method to directly set the return value for us:

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.return_
↪value': mock_response}
>>> mock_backend.configure_mock(**config)
```

With these we monkey patch the "mock backend" in place and can make the real call:

```
>>> something.backend = mock_backend
>>> something.method()
```

Using *mock_calls* we can check the chained call with a single assert. A chained call is several calls in one line of code, so there will be several entries in `mock_calls`. We can use *call.call_list()* to create this list of calls for us:

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

Partial mocking

In some tests I wanted to mock out a call to *datetime.date.today()* to return a known date, but I didn't want to prevent the code under test from creating new date objects. Unfortunately *datetime.date* is written in C, and so I couldn't just monkey-patch out the static `date.today()` method.

I found a simple way of doing this that involved effectively wrapping the date class with a mock, but passing through calls to the constructor to the real class (and returning real instances).

The *patch decorator* is used here to mock out the `date` class in the module under test. The `side_effect` attribute on the mock date class is then set to a lambda function that returns a real date. When the mock date class is called a real date will be constructed and returned by `side_effect`.

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
...
```

Note that we don't patch *datetime.date* globally, we patch `date` in the module that *uses* it. See *where to patch*.

When `date.today()` is called a known date is returned, but calls to the `date(...)` constructor still return normal dates. Without this you can find yourself having to calculate an expected result using exactly the same algorithm as the code under test, which is a classic testing anti-pattern.

Calls to the date constructor are recorded in the `mock_date` attributes (`call_count` and friends) which may also be useful for your tests.

An alternative way of dealing with mocking dates, or other builtin classes, is discussed in this blog entry.

Mocking a Generator Method

A Python generator is a function or method that uses the `yield` statement to return a series of values when iterated over[1].

A generator method / function is called to return the generator object. It is the generator object that is then iterated over. The protocol method for iteration is *`__iter__()`*, so we can mock this using a *`MagicMock`*.

Here's an example class with an "iter" method implemented as a generator:

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

How would we mock this class, and in particular its "iter" method?

To configure the values returned from the iteration (implicit in the call to *`list`*), we need to configure the object returned by the call to `foo.iter()`.

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

Applying the same patch to every test method

If you want several patches in place for multiple test methods the obvious way is to apply the patch decorators to every method. This can feel like unnecessary repetition. For Python 2.6 or more recent you can use *`patch()`* (in all its various forms) as a class decorator. This applies the patches to all test methods on the class. A test method is identified by methods whose names start with `test`:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
```

(下页继续)

[1] There are also generator expressions and more advanced uses of generators, but we aren't concerned about them here. A very good introduction to generators and how powerful they are is: Generator Tricks for Systems Programmers.

(续上页)

```
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

An alternative way of managing patches is to use the *patch methods: start and stop*. These allow you to move the patching into your `setUp` and `tearDown` methods.

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

If you use this technique you must ensure that the patching is "undone" by calling `stop`. This can be fiddlier than you might think, because if an exception is raised in the setUp then tearDown is not called. *`unittest.TestCase.addCleanup()`* makes this easier:

```
>>> class MyTest(TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

Mocking Unbound Methods

Whilst writing tests today I needed to patch an *unbound method* (patching the method on the class rather than on the instance). I needed self to be passed in as the first argument because I want to make asserts about which objects were calling this particular method. The issue is that you can't patch with a mock for this, because if you replace an unbound method with a mock it doesn't become a bound method when fetched from the instance, and so it doesn't get self passed in. The workaround is to patch the unbound method with a real function instead. The *`patch()`* decorator makes it so simple to patch out methods with a mock that having to create a real function becomes a nuisance.

If you pass `autospec=True` to patch then it does the patching with a *real* function object. This function object has the same signature as the one it is replacing, but delegates to a mock under the hood. You still get your mock auto-created in exactly the same way as before. What it means though, is that if you use it to patch out an unbound method on a class the mocked function will be turned into a bound method if it is fetched from an instance. It will have `self` passed in as the first argument, which is exactly what I wanted:

```
>>> class Foo:
...   def foo(self):
...     pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

If we don't use `autospec=True` then the unbound method is patched out with a Mock instance instead, and isn't called with `self`.

Checking multiple calls with mock

mock has a nice API for making assertions about how your mock objects are used.

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

If your mock is only being called once you can use the `assert_called_once_with()` method that also asserts that the `call_count` is one.

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
AssertionError: Expected to be called once. Called 2 times.
```

Both `assert_called_with` and `assert_called_once_with` make assertions about the *most recent* call. If your mock is going to be called several times, and you want to make assertions about *all* those calls you can use *`call_args_list`*:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

The *`call`* helper makes it easy to make assertions about these calls. You can build up a list of expected calls and compare it to `call_args_list`. This looks remarkably similar to the repr of the `call_args_list`:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

Coping with mutable arguments

Another situation is rare, but can bite you, is when your mock is called with mutable arguments. `call_args` and `call_args_list` store *references* to the arguments. If the arguments are mutated by the code under test then you can no longer make assertions about what the values were when the mock was called.

Here's some example code that shows the problem. Imagine the following functions defined in 'mymodule':

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

When we try to test that `grob` calls `frob` with the correct argument look what happens:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

One possibility would be for mock to copy the arguments you pass in. This could then cause problems if you do assertions that rely on object identity for equality.

Here's one solution that uses the `side_effect` functionality. If you provide a `side_effect` function for a mock then `side_effect` will be called with the same args as the mock. This gives us an opportunity to copy the arguments and store them for later assertions. In this example I'm using *another* mock to store the arguments so that I can use the mock methods for doing the assertion. Again a helper function sets this up for me.

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

`copy_call_args` is called with the mock that will be called. It returns a new mock that we do the assertion on. The `side_effect` function makes a copy of the args and calls our `new_mock` with the copy.

注解： If your mock is only going to be used once there is an easier way of checking arguments at the point they are called. You can simply do the checking inside a `side_effect` function.

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

An alternative approach is to create a subclass of *Mock* or *MagicMock* that copies (using *copy.deepcopy()*) the arguments. Here's an example implementation:

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super(CopyingMock, self).__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: Expected call: mock({1})
Actual call: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

When you subclass `Mock` or `MagicMock` all dynamically created attributes, and the `return_value` will use your subclass automatically. That means all children of a `CopyingMock` will also have the type `CopyingMock`.

Nesting Patches

Using patch as a context manager is nice, but if you do multiple patches you can end up with nested with statements indenting further and further to the right:

```
>>> class MyTest(TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
```

(下页继续)

(续上页)

```
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

With unittest `cleanup` functions and the *patch methods: start and stop* we can achieve the same effect without the nested indentation. A simple helper method, `create_patch`, puts the patch in place and returns the created mock for us:

```
>>> class MyTest(TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

Mocking a dictionary with MagicMock

You may want to mock a dictionary, or other container object, recording all access to it whilst having it still behave like a dictionary.

We can do this with *MagicMock*, which will behave like a dictionary, and using *side_effect* to delegate dictionary access to a real underlying dictionary that is under our control.

When the `__getitem__()` and `__setitem__()` methods of our `MagicMock` are called (normal dictionary access) then `side_effect` is called with the key (and in the case of `__setitem__` the value too). We can also control what is returned.

After the `MagicMock` has been used we can use attributes like *call_args_list* to assert about how the dictionary was used:

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
...      return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

注解： An alternative to using `MagicMock` is to use `Mock` and *only* provide the magic methods you specifically want:

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

A *third* option is to use `MagicMock` but passing in `dict` as the *spec* (or *spec_set*) argument so that the `MagicMock` created only has dictionary magic methods available:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

With these side effect functions in place, the `mock` will behave like a normal dictionary but recording the access. It even raises a *KeyError* if you try to access a key that doesn't exist.

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

After it has been used you can make assertions about the access using the normal mock methods and attributes:

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'c': 3, 'b': 'fish', 'd': 'eggs'}
```

Mock subclasses and their attributes

There are various reasons why you might want to subclass *Mock*. One reason might be to add helper methods. Here's a silly example:

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
```

(下页继续)

(续上页)

```
>>> mymock()
>>> mymock.has_been_called()
True
```

The standard behaviour for `Mock` instances is that attributes and the return value mocks are of the same type as the mock they are accessed on. This ensures that `Mock` attributes are `Mocks` and `MagicMock` attributes are `MagicMocks`[2]. So if you're subclassing to add helper methods then they'll also be available on the attributes and return value mock of instances of your subclass.

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

Sometimes this is inconvenient. For example, one user is subclassing mock to created a Twisted adaptor. Having this applied to attributes too actually causes errors.

`Mock` (in all its flavours) uses a method called `_get_child_mock` to create these "sub-mocks" for attributes and return values. You can prevent your subclass being used for attributes by overriding this method. The signature is that it takes arbitrary keyword arguments (`**kwargs`) which are then passed onto the mock constructor:

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

Mocking imports with patch.dict

One situation where mocking can be hard is where you have a local import inside a function. These are harder to mock because they aren't using an object from the module namespace that we can patch out.

Generally local imports are to be avoided. They are sometimes done to prevent circular dependencies, for which there is *usually* a much better way to solve the problem (refactor the code) or to prevent "up front costs" by delaying the import. This can also be solved in better ways than an unconditional local import (store the module as a class or module attribute and only do the import on first use).

That aside there is a way to use `mock` to affect the results of an import. Importing fetches an *object* from the *sys.modules* dictionary. Note that it fetches an *object*, which need not be a module. Importing a module for the first time results in a module object being put in *sys.modules*, so usually when you import something you get a module back. This need not be the case however.

This means you can use *patch.dict()* to *temporarily* put a mock in place in *sys.modules*. Any imports whilst this patch is active will fetch the mock. When the patch is complete (the decorated function exits, the with statement

[2] An exception to this rule are the non-callable mocks. Attributes use the callable variant because otherwise non-callable mocks couldn't have callable methods.

body is complete or `patcher.stop()` is called) then whatever was there previously will be restored safely.

Here's an example that mocks out the 'fooble' module.

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

As you can see the `import fooble` succeeds, but on exit there is no 'fooble' left in *sys.modules*.

This also works for the `from module import name` form:

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

With slightly more work you can also mock package imports:

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

Tracking order of calls and less verbose call assertions

The *Mock* class allows you to track the *order* of method calls on your mock objects through the *method_calls* attribute. This doesn't allow you to track the order of calls between separate mock objects, however we can use *mock_calls* to achieve the same effect.

Because mocks track calls to child mocks in `mock_calls`, and accessing an arbitrary attribute of a mock creates a child mock, we can create our separate mocks from a parent one. Calls to those child mock will then all be recorded, in order, in the `mock_calls` of the parent:

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

We can then assert about the calls, including the order, by comparing with the `mock_calls` attribute on the manager mock:

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

If `patch` is creating, and putting in place, your mocks then you can attach them to a manager mock using the *attach_mock()* method. After attaching calls will be recorded in `mock_calls` of the manager.

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
...
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
 call.MockClass1().foo(),
 call.MockClass2(),
 call.MockClass2().bar()]
```

If many calls have been made, but you're only interested in a particular sequence of them then an alternative is to use the *assert_has_calls()* method. This takes a list of calls (constructed with the *call* object). If that sequence of calls are in *mock_calls* then the assert succeeds.

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

Even though the chained call `m.one().two().three()` aren't the only calls that have been made to the mock, the assert still succeeds.

Sometimes a mock may have several calls made to it, and you are only interested in asserting about *some* of those calls. You may not even care about the order. In this case you can pass `any_order=True` to `assert_has_calls`:

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

More complex argument matching

Using the same basic concept as *ANY* we can implement matchers to do more complex assertions on objects used as arguments to mocks.

Suppose we expect some object to be passed to a mock that by default compares equal based on object identity (which is the Python default for user defined classes). To use *assert_called_with()* we would need to pass in the exact same object. If we are only interested in some of the attributes of this object then we can create a matcher that will check these attributes for us.

You can see in this example how a 'standard' call to `assert_called_with` isn't sufficient:

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
    ...
AssertionError: Expected: call(<__main__.Foo object at 0x...>)
Actual call: call(<__main__.Foo object at 0x...>)
```

A comparison function for our `Foo` class might look something like this:

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

And a matcher object that can use comparison functions like this for its equality operation would look something like this:

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

Putting all this together:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

The `Matcher` is instantiated with our compare function and the `Foo` object we want to compare against. In `assert_called_with` the `Matcher` equality method will be called, which compares the object the mock was called with against the one we created our matcher with. If they match then `assert_called_with` passes, and if they don't an *AssertionError* is raised:

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
```

(下页继续)

(续上页)

```
Traceback (most recent call last):
    ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

With a bit of tweaking you could have the comparison function raise the *AssertionError* directly and provide a more useful failure message.

As of version 1.5, the Python testing library PyHamcrest provides similar functionality, that may be useful here, in the form of its equality matcher (hamcrest.library.integration.match_equality).

26.7 2to3 - 自动将 Python 2 代码转为 Python 3 代码

2to3 是一个 Python 程序，它可以用来读取 Python 2.x 版本的代码，并使用一系列的 修复器来将其转换为合法的 Python 3.x 代码。标准库中已经包含了丰富的修复器，这足以处理绝大多数代码。不过 2to3 的支持库 *lib2to3* 是一个很灵活通用的库，所以你也可以为 2to3 编写你自己的修复器。*lib2to3* 也可以用在那些需要自动处理 Python 代码的应用中。

26.7.1 使用 2to3

2to3 通常会作为脚本和 Python 解释器一起安装，你可以在 Python 根目录的 `Tools/scripts` 文件夹下找到它。

2to3 的基本调用参数是一个需要转换的文件或目录列表。对于目录，会递归地寻找其中的 Python 源码。

这里有一个 Python 2.x 的源码文件，`example.py`：

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

它可以在命令行中使用 2to3 转换成 Python 3.x 版本的代码：

```
$ 2to3 example.py
```

这个命令会打印出和源文件的区别。通过传入 `-w` 参数，2to3 也可以把需要的修改写回到原文件中（除非传入了 `-n` 参数，否则会为原始文件创建一个副本）：

```
$ 2to3 -w example.py
```

在转换完成后，`example.py` 看起来像是这样：

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
name = input()
greet(name)
```

注释和缩进都会在转换过程中保持不变。

默认情况下，2to3 会执行预定义修复器 的集合。使用 -l 参数可以列出所有可用的修复器。使用 -f 参数可以明确指定需要使用的修复器集合。而使用 -x 参数则可以明确指定不使用的修复器。下面的例子会只使用 imports 和 has_key 修复器运行：

```
$ 2to3 -f imports -f has_key example.py
```

这个命令会执行除了 apply 之外的所有修复器：

```
$ 2to3 -x apply example.py
```

有一些修复器是需要 显式指定的，它们默认不会执行，必须在命令行中列出才会执行。比如下面的例子，除了默认的修复器以外，还会执行 idioms 修复器：

```
$ 2to3 -f all -f idioms example.py
```

注意这里使用 all 来启用所有默认的修复器。

有些情况下 2to3 会找到源码中有一些需要修改，但是无法自动处理的代码。在这种情况下，2to3 会在差异处下面打印一个警告信息。你应该定位到相应的代码并对其进行修改，以使其兼容 Python 3.x。

2to3 也可以重构 doctests。使用 -d 开启这个模式。需要注意 * 只有 * doctests 会被重构。这种模式下不需要文件是合法的 Python 代码。举例来说，reST 文档中类似 doctests 的示例也可以使用这个选项进行重构。

-v 选项可以输出更多转换程序的详细信息。

由于某些 print 语句可被解读为函数调用或是语句，2to3 并不是总能读取包含 print 函数的文件。当 2to3 检测到存在 from __future__ import print_function 编译器指令时，会修改其内部语法将*print()* 解读为函数。这一变动也可以使用 -p 选项手动开启。使用 -p 来为已经转换过 print 语句的代码运行修复器。

-o 或 --output-dir 选项可以指定将转换后的文件写入其他目录中。由于这种情况下不会覆写原始文件，所以创建副本文件毫无意义，因此也需要使用 -n 选项来禁用创建副本。

3.2.3 新版功能: 增加了 -o 选项。

-W 或 --write-unchanged-files 选项用来告诉 2to3 始终需要输出文件，即使没有任何改动。这在使用 -o 参数时十分有用，这样就可以将整个 Python 源码包完整地转换到另一个目录。这个选项隐含了 -w 选项，否则等于没有作用。

3.2.3 新版功能: 增加了 -W 选项。

--add-suffix 选项接受一个字符串，用来作为后缀附加在输出文件名后面的后面。由于写入的文件名与原始文件不同，所以没有必要创建副本，因此 -n 选项也是必要的。举个例子：

```
$ 2to3 -n -W --add-suffix=3 example.py
```

这样会把转换后的文件写入 example.py3 文件。

3.2.3 新版功能: 增加了 --add-suffix 选项。

将整个项目从一个目录转换到另一个目录可以用这样的命令：

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

26.7.2 修复器

转换代码的每一个步骤都封装在修复器中。可以使用 2to3 -l 来列出可用的修复器。之前已经提到，每个修复器都可以独立地打开或是关闭。下面会对各个修复器做更详细的描述。

apply
移除对 apply() 的使用，举例来说，apply(function, *args, **kwargs) 会被转换成 function(*args, **kwargs)。

asserts
将已弃用的*unittest* 方法替换为正确的。

从	到
failUnlessEqual(a, b)	*assertEqual(a, b)*
assertEquals(a, b)	*assertEqual(a, b)*
failIfEqual(a, b)	*assertNotEqual(a, b)*
assertNotEquals(a, b)	*assertNotEqual(a, b)*
failUnless(a)	*assertTrue(a)*
assert_(a)	*assertTrue(a)*
failIf(a)	*assertFalse(a)*
failUnlessRaises(exc, cal)	*assertRaises(exc, cal)*
failUnlessAlmostEqual(a, b)	*assertAlmostEqual(a, b)*
assertAlmostEquals(a, b)	*assertAlmostEqual(a, b)*
failIfAlmostEqual(a, b)	*assertNotAlmostEqual(a, b)*
assertNotAlmostEquals(a, b)	*assertNotAlmostEqual(a, b)*

basestring
将 basestring 转换为*str*。

buffer
将 buffer 转换为*memoryview*。这个修复器是可选的，因为*memoryview* API 和 buffer 很相似，但不完全一样。

dict
修复字典迭代方法。dict.iteritems() 会转换成*dict.items()*，dict.iterkeys() 会转换成*dict.keys()*，dict.itervalues() 会转换成*dict.values()*。类似的，dict.viewitems()，dict.viewkeys() 和 dict.viewvalues() 会分别转换成*dict.items()*，*dict.keys()* 和*dict.values()*。另外也会将原有的*dict.items()*，*dict.keys()* 和*dict.values()* 方法调用用*list* 包装一层。

except
将 except X, T 转换为 except X as T。

exec
将 exec 语句转换为*exec()* 函数调用。

execfile
移除 execfile() 的使用。execfile() 的实参会使用*open()*，*compile()* 和*exec()* 包装。

exitfunc
将对 sys.exitfunc 的赋值改为使用*atexit* 模块代替。

filter
将*filter()* 函数用*list* 包装一层。

funcattrs
修复已经重命名的函数属性。比如 my_function.func_closure 会被转换为 my_function.__closure__。

future
移除 `from __future__ import new_feature` 语句。

getcwdu
将 `os.getcwdu()` 重命名为*os.getcwd()*。

has_key
将 `dict.has_key(key)` 转换为 `key in dict`。

idioms
这是一个可选的修复器，会进行多种转换，将 Python 代码变成更加常见的写法。类似 `type(x) is SomeClass` 和 `type(x) == SomeClass` 的类型对比会被转换成 `isinstance(x, SomeClass)`。`while 1` 转换成 `while True`。这个修复器还会在合适的地方使用*sorted()* 函数。举个例子，这样的代码块：

```
L = list(some_iterable)
L.sort()
```

会被转换为：

```
L = sorted(some_iterable)
```

import
检测 sibling imports，并将其转换成相对 import。

imports
处理标准库模块的重命名。

imports2
处理标准库中其他模块的重命名。这个修复器由于一些技术上的限制，因此和*imports* 拆分开了。

input
将 `input(prompt)` 转换为 `eval(input(prompt))`。

intern
将 `intern()` 转换为*sys.intern()*。

isinstance
修复*isinstance()* 函数第二个实参中重复的类型。举例来说，`isinstance(x, (int, int))` 会转换为 `isinstance(x, int)`，`isinstance(x, (int, float, int))` 会转换为 `isinstance(x, (int, float))`。

itertools_imports
移除 `itertools.ifilter()`，`itertools.izip()` 以及 `itertools.imap()` 的 import。对 `itertools.ifilterfalse()` 的 import 也会替换成*itertools.filterfalse()*。

itertools
修改 `itertools.ifilter()`，`itertools.izip()` 和 `itertools.imap()` 的调用为对应的内建实现。`itertools.ifilterfalse()` 会替换成*itertools.filterfalse()*。

long
将 `long` 重命名为*int*。

map
用*list* 包装*map()*。同时也会将 `map(None, x)` 替换为 `list(x)`。使用 `from future_builtins import map` 禁用这个修复器。

metaclass
将老的元类语法（类体中的 `__metaclass__ = Meta`）替换为新的（`class X(metaclass=Meta)`）。

methodattrs
修复老的方法属性名。例如 `meth.im_func` 会被转换为 `meth.__func__`。

ne
转换老的不等语法，将 <> 转为 !=。

next
将迭代器的 next() 方法调用转为*next()* 函数。也会将*next()* 方法重命名为*__next__()*。

nonzero
将 __nonzero__() 转换为 __bool__()。

numliterals
将八进制字面量转为新的语法。

operator
Converts calls to various functions in the *operator* module to other, but equivalent, function calls. When needed, the appropriate `import` statements are added, e.g. `import collections`. The following mapping are made:

从	到
`operator.isCallable(obj)`	`hasattr(obj, '__call__')`
`operator.sequenceIncludes(obj)`	`operator.contains(obj)`
`operator.isSequenceType(obj)`	`isinstance(obj, collections.Sequence)`
`operator.isMappingType(obj)`	`isinstance(obj, collections.Mapping)`
`operator.isNumberType(obj)`	`isinstance(obj, numbers.Number)`
`operator.repeat(obj, n)`	`operator.mul(obj, n)`
`operator.irepeat(obj, n)`	`operator.imul(obj, n)`

paren
在列表生成式中增加必须的括号。例如将 `[x for x in 1, 2]` 转换为 `[x for x in (1, 2)]`。

print
将 print 语句转换为*print()* 函数。

raise
将 `raise E, V` 转换为 `raise E(V)`，将 `raise E, V, T` 转换为 `raise E(V).with_traceback(T)`。如果 E 是元组，这样的转换是不正确的，因为用元组代替异常的做法在 3.0 中已经移除了。

raw_input
将 raw_input() 转换为*input()*。

reduce
将 reduce() 转换为*functools.reduce()*。

reload
Converts `reload()` to *imp.reload()*.

renames
将 sys.maxint 转换为*sys.maxsize*。

repr
将反引号 repr 表达式替换为*repr()* 函数。

set_literal
将*set* 构造函数替换为 set literals 写法。这个修复器是可选的。

standarderror
将 StandardError 重命名为*Exception*。

sys_exc
将弃用的 sys.exc_value，sys.exc_type，sys.exc_traceback 替换为*sys.exc_info()* 的用法。

throw
修复生成器的 `throw()` 方法的 API 变更。

tuple_params
移除隐式的元组参数解包。这个修复器会插入临时变量。

types
修复*type* 模块中一些成员的移除引起的代码问题。

unicode
将 `unicode` 重命名为*str*。

urllib
将*urllib* 和 `urllib2` 重命名为*urllib* 包。

ws_comma
移除逗号分隔的元素之间多余的空白。这个修复器是可选的。

xrange
将 `xrange()` 重命名为*range()*，并用*list* 包装原有的*range()*。

xreadlines
将 `for x in file.xreadlines()` 转换为 `for x in file`。

zip
用*list* 包装*zip()*。如果使用了 `from future_builtins import zip` 的话会禁用。

26.7.3 lib2to3 ——2to3 支持库

源代码： Lib/lib2to3/

注解： *lib2to3* API 并不稳定，并可能在未来大幅修改。

26.8 test —Python 回归测试包

注解： The *test* package is meant for internal use by Python only. It is documented for the benefit of the core developers of Python. Any use of this package outside of Python's standard library is discouraged as code mentioned here can change or be removed without notice between releases of Python.

The *test* package contains all regression tests for Python as well as the modules *test.support* and `test.regrtest`. *test.support* is used to enhance your tests while `test.regrtest` drives the testing suite.

Each module in the *test* package whose name starts with `test_` is a testing suite for a specific module or feature. All new tests should be written using the *unittest* or *doctest* module. Some older tests are written using a "traditional" testing style that compares output printed to `sys.stdout`; this style of test is considered deprecated.

参见：

模块*unittest* 编写 PyUnit 回归测试.

***doctest*—文档测试模块** Tests embedded in documentation strings.

26.8.1 Writing Unit Tests for the `test` package

It is preferred that tests that use the *unittest* module follow a few guidelines. One is to name the test module by starting it with `test_` and end it with the name of the module being tested. The test methods in the test module should start with `test_` and end with a description of what the method is testing. This is needed so that the methods are recognized by the test driver as test methods. Also, no documentation string for the method should be included. A comment (such as `# Tests function returns only True or False`) should be used to provide documentation for test methods. This is done because documentation strings get printed out if they exist and thus what test is being run is not stated.

A basic boilerplate is often used:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

This code pattern allows the testing suite to be run by `test.regrtest`, on its own as a script that supports the *unittest* CLI, or via the `python -m unittest` CLI.

The goal for regression testing is to try to break code. This leads to a few guidelines to be followed:

- The testing suite should exercise all classes, functions, and constants. This includes not just the external API that is to be presented to the outside world but also "private" code.
- Whitebox testing (examining the code being tested when the tests are being written) is preferred. Blackbox testing (testing only the published user interface) is not complete enough to make sure all boundary and edge cases are tested.
- Make sure all possible values are tested including invalid ones. This makes sure that not only all valid values are acceptable but also that improper values are handled correctly.
- Exhaust as many code paths as possible. Test where branching occurs and thus tailor input to make sure as many different paths through the code are taken.

- Add an explicit test for any bugs discovered for the tested code. This will make sure that the error does not crop up again if the code is changed in the future.
- Make sure to clean up after your tests (such as close and remove all temporary files).
- If a test is dependent on a specific condition of the operating system then verify the condition already exists before attempting the test.
- Import as few modules as possible and do it as soon as possible. This minimizes external dependencies of tests and also minimizes possible anomalous behavior from side-effects of importing a module.
- Try to maximize code reuse. On occasion, tests will vary by something as small as what type of input is used. Minimize code duplication by subclassing a basic test class with a class that specifies the input:

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

  When using this pattern, remember that all classes that inherit from *unittest.TestCase* are run as tests. The `Mixin` class in the example above does not have any data and so can't be run by itself, thus it does not inherit from *unittest.TestCase*.

参见:

Test Driven Development A book by Kent Beck on writing tests before code.

26.8.2 Running tests using the command-line interface

The *test* package can be run as a script to drive Python's regression test suite, thanks to the `-m` option: **python -m test**. Under the hood, it uses `test.regrtest`; the call **python -m test.regrtest** used in previous Python versions still works. Running the script by itself automatically starts running all regression tests in the *test* package. It does this by finding all modules in the package whose name starts with `test_`, importing them, and executing the function `test_main()` if present or loading the tests via unittest.TestLoader.loadTestsFromModule if `test_main` does not exist. The names of tests to execute may also be passed to the script. Specifying a single regression test (**python -m test test_spam**) will minimize output and only print whether the test passed or failed.

Running *test* directly allows what resources are available for tests to use to be set. You do this by using the `-u` command-line option. Specifying `all` as the value for the `-u` option enables all possible resources: **python -m test -uall**. If all but one resource is desired (a more common case), a comma-separated list of resources that are not desired may be listed after `all`. The command **python -m test -uall,-audio,-largefile** will run *test* with all resources except the `audio` and `largefile` resources. For a list of all resources and more command-line options, run **python -m test -h**.

Some other ways to execute the regression tests depend on what platform the tests are being executed on. On Unix, you can run **make test** at the top-level directory where Python was built. On Windows, executing **rt.bat** from your `PCBuild` directory will run all regression tests.

26.9 test.support —Utilities for the Python test suite

The *test.support* module provides support for Python's regression test suite.

注解: *test.support* is not a public module. It is documented here to help Python developers write tests. The API of this module is subject to change without backwards compatibility concerns between releases.

This module defines the following exceptions:

exception test.support.**TestFailed**
Exception to be raised when a test fails. This is deprecated in favor of *unittest*-based tests and *unittest.TestCase*'s assertion methods.

exception test.support.**ResourceDenied**
Subclass of *unittest.SkipTest*. Raised when a resource (such as a network connection) is not available. Raised by the *requires()* function.

The *test.support* module defines the following constants:

test.support.**verbose**
`True` when verbose output is enabled. Should be checked when more detailed information is desired about a running test. *verbose* is set by `test.regrtest`.

test.support.**is_jython**
`True` if the running interpreter is Jython.

test.support.**TESTFN**
Set to a name that is safe to use as the name of a temporary file. Any temporary file that is created should be closed and unlinked (removed).

The *test.support* module defines the following functions:

test.support.**forget**(*module_name*)
Remove the module named *module_name* from `sys.modules` and delete any byte-compiled files of the module.

test.support.**is_resource_enabled**(*resource*)
Return `True` if *resource* is enabled and available. The list of available resources is only set when `test.regrtest` is executing the tests.

test.support.**requires**(*resource*, *msg=None*)
Raise *ResourceDenied* if *resource* is not available. *msg* is the argument to *ResourceDenied* if it is raised. Always returns `True` if called by a function whose `__name__` is `'__main__'`. Used when tests are executed by `test.regrtest`.

test.support.**findfile**(*filename*, *subdir=None*)
Return the path to the file named *filename*. If no match is found *filename* is returned. This does not equal a failure since it could be the path to the file.

Setting *subdir* indicates a relative path to use to find the file rather than looking directly in the path directories.

test.support.**run_unittest**(**classes*)
Execute *unittest.TestCase* subclasses passed to the function. The function scans the classes for methods starting with the prefix `test_` and executes the tests individually.

It is also legal to pass strings as parameters; these should be keys in `sys.modules`. Each associated module will be scanned by `unittest.TestLoader.loadTestsFromModule()`. This is usually seen in the following `test_main()` function:

```
def test_main():
    support.run_unittest(__name__)
```

This will run all tests defined in the named module.

test.support.**run_doctest**(*module*, *verbosity=None*)
Run *doctest.testmod()* on the given *module*. Return `(failure_count, test_count)`.

If *verbosity* is `None`, *doctest.testmod()* is run with verbosity set to *verbose*. Otherwise, it is run with verbosity set to `None`.

test.support.**check_warnings**(**filters*, *quiet=True*)
A convenience wrapper for *warnings.catch_warnings()* that makes it easier to test that a warning was correctly raised. It is approximately equivalent to calling `warnings.catch_warnings(record=True)` with *warnings.simplefilter()* set to `always` and with the option to automatically validate the results that are recorded.

`check_warnings` accepts 2-tuples of the form `("message regexp", WarningCategory)` as positional arguments. If one or more *filters* are provided, or if the optional keyword argument *quiet* is `False`, it checks to make sure the warnings are as expected: each specified filter must match at least one of the warnings raised by the enclosed code or the test fails, and if any warnings are raised that do not match any of the specified filters the test fails. To disable the first of these checks, set *quiet* to `True`.

If no arguments are specified, it defaults to:

```
check_warnings(("", Warning), quiet=True)
```

In this case all warnings are caught and no errors are raised.

On entry to the context manager, a `WarningRecorder` instance is returned. The underlying warnings list from *catch_warnings()* is available via the recorder object's *warnings* attribute. As a convenience, the attributes of the object representing the most recent warning can also be accessed directly through the recorder object (see example below). If no warning has been raised, then any of the attributes that would otherwise be expected on an object representing a warning will return `None`.

The recorder object also has a `reset()` method, which clears the warnings list.

The context manager is designed to be used like this:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

In this case if either warning was not raised, or some other warning was raised, *check_warnings()* would raise an error.

When a test needs to look more deeply into the warnings, rather than just checking whether or not they occurred, code like this can be used:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

Here all warnings will be caught, and the test code tests the captured warnings directly.

在 3.2 版更改: New optional arguments *filters* and *quiet*.

test.support.**captured_stdin**()
test.support.**captured_stdout**()
test.support.**captured_stderr**()
A context managers that temporarily replaces the named stream with *io.StringIO* object.

Example use with output streams:

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

Example use with input stream:

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.**temp_dir**(*path=None*, *quiet=False*)
A context manager that creates a temporary directory at *path* and yields the directory.

If *path* is None, the temporary directory is created using *tempfile.mkdtemp()*. If *quiet* is False, the context manager raises an exception on error. Otherwise, if *path* is specified and cannot be created, only a warning is issued.

test.support.**change_cwd**(*path*, *quiet=False*)
A context manager that temporarily changes the current working directory to *path* and yields the directory.

If *quiet* is False, the context manager raises an exception on error. Otherwise, it issues only a warning and keeps the current working directory the same.

test.support.**temp_cwd**(*name='tempcwd'*, *quiet=False*)
A context manager that temporarily creates a new directory and changes the current working directory (CWD).

The context manager creates a temporary directory in the current directory with name *name* before temporarily changing the current working directory. If *name* is None, the temporary directory is created using *tempfile.mkdtemp()*.

If *quiet* is False and it is not possible to create or change the CWD, an error is raised. Otherwise, only a warning is raised and the original CWD is used.

test.support.**temp_umask**(*umask*)
A context manager that temporarily sets the process umask.

test.support.**can_symlink**()
Return True if the OS supports symbolic links, False otherwise.

@test.support.**skip_unless_symlink**
A decorator for running tests that require support for symbolic links.

@test.support.**anticipate_failure**(*condition*)
A decorator to conditionally mark tests with *unittest.expectedFailure()*. Any use of this decorator should have an associated comment identifying the relevant tracker issue.

@test.support.**run_with_locale**(*catstr*, **locales*)
A decorator for running a function in a different locale, correctly resetting it after it has finished. *catstr* is the locale category as a string (for example "LC_ALL"). The *locales* passed will be tried sequentially, and the first valid locale will be used.

test.support.**make_bad_fd**()
Create an invalid file descriptor by opening and closing a temporary file, and returning its descriptor.

test.support.**import_module**(*name*, *deprecated=False*)
This function imports and returns the named module. Unlike a normal import, this function raises *unittest.SkipTest* if the module cannot be imported.

Module and package deprecation messages are suppressed during this import if *deprecated* is True.

3.1 新版功能.

test.support.**import_fresh_module**(*name*, *fresh=()*, *blocked=()*, *deprecated=False*)
This function imports and returns a fresh copy of the named Python module by removing the named module from sys.modules before doing the import. Note that unlike reload(), the original module is not affected by this operation.

fresh is an iterable of additional module names that are also removed from the sys.modules cache before doing the import.

blocked is an iterable of module names that are replaced with None in the module cache during the import to ensure that attempts to import them raise *ImportError*.

The named module and any modules named in the *fresh* and *blocked* parameters are saved before starting the import and then reinserted into sys.modules when the fresh import is complete.

Module and package deprecation messages are suppressed during this import if *deprecated* is True.

This function will raise *ImportError* if the named module cannot be imported.

Example use:

```
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

3.1 新版功能.

test.support.**bind_port**(*sock*, *host=HOST*)
Bind the socket to a free port and return the port number. Relies on ephemeral ports in order to ensure we are using an unbound port. This is important as many tests may be running simultaneously, especially in a buildbot environment. This method raises an exception if the sock.family is *AF_INET* and sock.type is *SOCK_STREAM*, and the socket has SO_REUSEADDR or SO_REUSEPORT set on it. Tests should never set these socket options for TCP/IP sockets. The only case for setting these options is testing multicasting via multiple UDP sockets.

Additionally, if the SO_EXCLUSIVEADDRUSE socket option is available (i.e. on Windows), it will be set on the socket. This will prevent anyone else from binding to our host/port for the duration of the test.

test.support.**find_unused_port**(*family=socket.AF_INET*, *socktype=socket.SOCK_STREAM*)
Returns an unused port that should be suitable for binding. This is achieved by creating a temporary socket with the same family and type as the sock parameter (default is *AF_INET*, *SOCK_STREAM*), and binding it to the specified host address (defaults to 0.0.0.0) with the port set to 0, eliciting an unused ephemeral port from the OS. The temporary socket is then closed and deleted, and the ephemeral port is returned.

Either this method or *bind_port()* should be used for any tests where a server socket needs to be bound to a particular port for the duration of the test. Which one to use depends on whether the calling code is creating a python socket, or if an unused port needs to be provided in a constructor or passed to an external program (i.e. the `-accept` argument to openssl's s_server mode). Always prefer *bind_port()* over *find_unused_port()* where possible. Using a hard coded port is discouraged since it can make multiple instances of the test impossible to run simultaneously, which is a problem for buildbots.

`test.support.`**load_package_tests**(*pkg_dir*, *loader*, *standard_tests*, *pattern*)

Generic implementation of the *unittest* `load_tests` protocol for use in test packages. *pkg_dir* is the root directory of the package; *loader*, *standard_tests*, and *pattern* are the arguments expected by `load_tests`. In simple cases, the test package's `__init__.py` can be the following:

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

`test.support.`**detect_api_mismatch**(*ref_api*, *other_api*, *, *ignore=()*)

Returns the set of attributes, functions or methods of *ref_api* not found on *other_api*, except for a defined list of items to be ignored in this check specified in *ignore*.

By default this skips private attributes beginning with '_' but includes all magic methods, i.e. those starting and ending in '__'.

3.5 新版功能.

`test.support.`**check__all__**(*test_case*, *module*, *name_of_module=None*, *extra=()*, *blacklist=()*)

Assert that the `__all__` variable of *module* contains all public names.

The module's public names (its API) are detected automatically based on whether they match the public name convention and were defined in *module*.

The *name_of_module* argument can specify (as a string or tuple thereof) what module(s) an API could be defined in in order to be detected as a public API. One case for this is when *module* imports part of its public API from other modules, possibly a C backend (like `csv` and its `_csv`).

The *extra* argument can be a set of names that wouldn't otherwise be automatically detected as "public", like objects without a proper `__module__` attribute. If provided, it will be added to the automatically detected ones.

The *blacklist* argument can be a set of names that must not be treated as part of the public API even though their names indicate otherwise.

Example use:

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        blacklist = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
```

(下页继续)

(续上页)

```
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, blacklist=blacklist)
```

3.6 新版功能.

The *test.support* module defines the following classes:

class test.support.**TransientResource**(*exc*, ***kwargs*)
Instances are a context manager that raises *ResourceDenied* if the specified exception type is raised. Any keyword arguments are treated as attribute/value pairs to be compared against any exception raised within the `with` statement. Only if all pairs match properly against attributes on the exception is *ResourceDenied* raised.

class test.support.**EnvironmentVarGuard**
Class used to temporarily set or unset environment variables. Instances can be used as a context manager and have a complete dictionary interface for querying/modifying the underlying `os.environ`. After exit from the context manager all changes to environment variables done through this instance will be rolled back.

在 3.1 版更改: Added dictionary interface.

EnvironmentVarGuard.**set**(*envvar*, *value*)
Temporarily set the environment variable `envvar` to the value of `value`.

EnvironmentVarGuard.**unset**(*envvar*)
Temporarily unset the environment variable `envvar`.

class test.support.**SuppressCrashReport**
A context manager used to try to prevent crash dialog popups on tests that are expected to crash a subprocess.

On Windows, it disables Windows Error Reporting dialogs using SetErrorMode.

On UNIX, *resource.setrlimit()* is used to set *resource.RLIMIT_CORE*'s soft limit to 0 to prevent coredump file creation.

On both platforms, the old value is restored by `__exit__()`.

class test.support.**WarningsRecorder**
Class used to record warnings for unit tests. See documentation of *check_warnings()* above for more details.

class test.support.**FakePath**(*path*)
Simple *path-like object*. It implements the `__fspath__()` method which just returns the *path* argument. If *path* is an exception, it will be raised in `__fspath__()`.

CHAPTER 27

调试和分析

这些库可以帮助你进行 Python 开发：调试器使你能够逐步执行代码，分析堆栈帧并设置断点等，而分析器运行代码并为你提供执行时间的详细分类，从而使你能够找出你程序中的瓶颈。

27.1 bdb —Debugger framework

Source code: Lib/bdb.py

The *bdb* module handles basic debugger functions, like setting breakpoints or managing execution via the debugger.

定义了以下异常：

exception bdb.**BdbQuit**
Exception raised by the *Bdb* class for quitting the debugger.

The *bdb* module also defines two classes:

class bdb.**Breakpoint**(*self*, *file*, *line*, *temporary=0*, *cond=None*, *funcname=None*)
This class implements temporary breakpoints, ignore counts, disabling and (re-)enabling, and conditionals.

Breakpoints are indexed by number through a list called `bpbynumber` and by `(file, line)` pairs through `bplist`. The former points to a single instance of class *Breakpoint*. The latter points to a list of such instances since there may be more than one breakpoint per line.

When creating a breakpoint, its associated filename should be in canonical form. If a *funcname* is defined, a breakpoint hit will be counted when the first line of that function is executed. A conditional breakpoint always counts a hit.

Breakpoint instances have the following methods:

deleteMe()
Delete the breakpoint from the list associated to a file/line. If it is the last breakpoint in that position, it also deletes the entry for the file/line.

enable()
Mark the breakpoint as enabled.

disable()
Mark the breakpoint as disabled.

bpformat()
Return a string with all the information about the breakpoint, nicely formatted:

- The breakpoint number.
- If it is temporary or not.
- Its file,line position.
- The condition that causes a break.
- If it must be ignored the next N times.
- The breakpoint hit count.

3.2 新版功能.

bpprint(*out=None*)
Print the output of *bpformat()* to the file *out*, or if it is None, to standard output.

class bdb.**Bdb**(*skip=None*)
The *Bdb* class acts as a generic Python debugger base class.

This class takes care of the details of the trace facility; a derived class should implement user interaction. The standard debugger class (*pdb.Pdb*) is an example.

The *skip* argument, if given, must be an iterable of glob-style module name patterns. The debugger will not step into frames that originate in a module that matches one of these patterns. Whether a frame is considered to originate in a certain module is determined by the __name__ in the frame globals.

3.1 新版功能: *skip* 参数。

The following methods of *Bdb* normally don't need to be overridden.

canonic(*filename*)
Auxiliary method for getting a filename in a canonical form, that is, as a case-normalized (on case-insensitive filesystems) absolute path, stripped of surrounding angle brackets.

reset()
Set the botframe, stopframe, returnframe and quitting attributes with values ready to start debugging.

trace_dispatch(*frame*, *event*, *arg*)
This function is installed as the trace function of debugged frames. Its return value is the new trace function (in most cases, that is, itself).

The default implementation decides how to dispatch a frame, depending on the type of event (passed as a string) that is about to be executed. *event* can be one of the following:

- "line": A new line of code is going to be executed.
- "call": A function is about to be called, or another code block entered.
- "return": A function or other code block is about to return.
- "exception": An exception has occurred.
- "c_call": A C function is about to be called.
- "c_return": A C function has returned.

- `"c_exception"`: A C function has raised an exception.

For the Python events, specialized functions (see below) are called. For the C events, no action is taken.

The *arg* parameter depends on the previous event.

See the documentation for *sys.settrace()* for more information on the trace function. For more information on code and frame objects, refer to types.

dispatch_line(*frame*)
If the debugger should stop on the current line, invoke the *user_line()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_line()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_call(*frame*, *arg*)
If the debugger should stop on this function call, invoke the *user_call()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_call()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_return(*frame*, *arg*)
If the debugger should stop on this function return, invoke the *user_return()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_return()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

dispatch_exception(*frame*, *arg*)
If the debugger should stop at this exception, invokes the *user_exception()* method (which should be overridden in subclasses). Raise a *BdbQuit* exception if the `Bdb.quitting` flag is set (which can be set from *user_exception()*). Return a reference to the *trace_dispatch()* method for further tracing in that scope.

Normally derived classes don't override the following methods, but they may if they want to redefine the definition of stopping and breakpoints.

stop_here(*frame*)
This method checks if the *frame* is somewhere below `botframe` in the call stack. `botframe` is the frame in which debugging started.

break_here(*frame*)
This method checks if there is a breakpoint in the filename and line belonging to *frame* or, at least, in the current function. If the breakpoint is a temporary one, this method deletes it.

break_anywhere(*frame*)
This method checks if there is a breakpoint in the filename of the current frame.

Derived classes should override these methods to gain control over debugger operation.

user_call(*frame*, *argument_list*)
This method is called from *dispatch_call()* when there is the possibility that a break might be necessary anywhere inside the called function.

user_line(*frame*)
This method is called from *dispatch_line()* when either *stop_here()* or *break_here()* yields `True`.

user_return(*frame*, *return_value*)
This method is called from *dispatch_return()* when *stop_here()* yields `True`.

user_exception(*frame*, *exc_info*)
This method is called from *dispatch_exception()* when *stop_here()* yields `True`.

do_clear(*arg*)
Handle how a breakpoint must be removed when it is a temporary one.

This method must be implemented by derived classes.

Derived classes and clients can call the following methods to affect the stepping state.

set_step()
Stop after one line of code.

set_next(*frame*)
Stop on the next line in or below the given frame.

set_return(*frame*)
Stop when returning from the given frame.

set_until(*frame*)
Stop when the line with the line no greater than the current one is reached or when returning from current frame.

set_trace([*frame*])
Start debugging from *frame*. If *frame* is not specified, debugging starts from caller's frame.

set_continue()
Stop only at breakpoints or when finished. If there are no breakpoints, set the system trace function to `None`.

set_quit()
Set the `quitting` attribute to `True`. This raises *BdbQuit* in the next call to one of the `dispatch_*()` methods.

Derived classes and clients can call the following methods to manipulate breakpoints. These methods return a string containing an error message if something went wrong, or `None` if all is well.

set_break(*filename*, *lineno*, *temporary=0*, *cond*, *funcname*)
Set a new breakpoint. If the *lineno* line doesn't exist for the *filename* passed as argument, return an error message. The *filename* should be in canonical form, as described in the *canonic()* method.

clear_break(*filename*, *lineno*)
Delete the breakpoints in *filename* and *lineno*. If none were set, an error message is returned.

clear_bpbynumber(*arg*)
Delete the breakpoint which has the index *arg* in the `Breakpoint.bpbynumber`. If *arg* is not numeric or out of range, return an error message.

clear_all_file_breaks(*filename*)
Delete all breakpoints in *filename*. If none were set, an error message is returned.

clear_all_breaks()
Delete all existing breakpoints.

get_bpbynumber(*arg*)
Return a breakpoint specified by the given number. If *arg* is a string, it will be converted to a number. If *arg* is a non-numeric string, if the given breakpoint never existed or has been deleted, a *ValueError* is raised.

3.2 新版功能.

get_break(*filename*, *lineno*)
Check if there is a breakpoint for *lineno* of *filename*.

get_breaks(*filename*, *lineno*)
Return all breakpoints for *lineno* in *filename*, or an empty list if none are set.

get_file_breaks(*filename*)
Return all breakpoints in *filename*, or an empty list if none are set.

get_all_breaks()
Return all breakpoints that are set.

Derived classes and clients can call the following methods to get a data structure representing a stack trace.

get_stack(*f*, *t*)
Get a list of records for a frame and all higher (calling) and lower frames, and the size of the higher part.

format_stack_entry(*frame_lineno*, *lprefix=': '*)
Return a string with information about a stack entry, identified by a `(frame, lineno)` tuple:

- The canonical form of the filename which contains the frame.
- The function name, or `"<lambda>"`.
- The input arguments.
- The return value.
- The line of code (if it exists).

The following two methods can be called by clients to use a debugger to debug a *statement*, given as a string.

run(*cmd*, *globals=None*, *locals=None*)
Debug a statement executed via the *exec()* function. *globals* defaults to `__main__.__dict__`, *locals* defaults to *globals*.

runeval(*expr*, *globals=None*, *locals=None*)
Debug an expression executed via the *eval()* function. *globals* and *locals* have the same meaning as in *run()*.

runctx(*cmd*, *globals*, *locals*)
For backwards compatibility. Calls the *run()* method.

runcall(*func*, **args*, ***kwds*)
Debug a single function call, and return its result.

Finally, the module defines the following functions:

bdb.**checkfuncname**(*b*, *frame*)
Check whether we should break here, depending on the way the breakpoint *b* was set.

If it was set via line number, it checks if `b.line` is the same as the one in the frame also passed as argument. If the breakpoint was set via function name, we have to check we are in the right frame (the right function) and if we are in its first executable line.

bdb.**effective**(*file*, *line*, *frame*)
Determine if there is an effective (active) breakpoint at this line of code. Return a tuple of the breakpoint and a boolean that indicates if it is ok to delete a temporary breakpoint. Return `(None, None)` if there is no matching breakpoint.

bdb.**set_trace**()
Start debugging with a *Bdb* instance from caller's frame.

27.2 faulthandler —Dump the Python traceback

3.3 新版功能.

This module contains functions to dump Python tracebacks explicitly, on a fault, after a timeout, or on a user signal. Call *faulthandler.enable()* to install fault handlers for the SIGSEGV, SIGFPE, SIGABRT, SIGBUS, and SIGILL signals. You can also enable them at startup by setting the PYTHONFAULTHANDLER environment variable or by using the -X faulthandler command line option.

The fault handler is compatible with system fault handlers like Apport or the Windows fault handler. The module uses an alternative stack for signal handlers if the sigaltstack() function is available. This allows it to dump the traceback even on a stack overflow.

The fault handler is called on catastrophic cases and therefore can only use signal-safe functions (e.g. it cannot allocate memory on the heap). Because of this limitation traceback dumping is minimal compared to normal Python tracebacks:

- Only ASCII is supported. The backslashreplace error handler is used on encoding.
- Each string is limited to 500 characters.
- Only the filename, the function name and the line number are displayed. (no source code)
- It is limited to 100 frames and 100 threads.
- The order is reversed: the most recent call is shown first.

By default, the Python traceback is written to *sys.stderr*. To see tracebacks, applications must be run in the terminal. A log file can alternatively be passed to *faulthandler.enable()*.

The module is implemented in C, so tracebacks can be dumped on a crash or when Python is deadlocked.

27.2.1 Dumping the traceback

faulthandler.**dump_traceback**(*file=sys.stderr*, *all_threads=True*)
Dump the tracebacks of all threads into *file*. If *all_threads* is False, dump only the current thread.

在 3.5 版更改: Added support for passing file descriptor to this function.

27.2.2 Fault handler state

faulthandler.**enable**(*file=sys.stderr*, *all_threads=True*)
Enable the fault handler: install handlers for the SIGSEGV, SIGFPE, SIGABRT, SIGBUS and SIGILL signals to dump the Python traceback. If *all_threads* is True, produce tracebacks for every running thread. Otherwise, dump only the current thread.

The *file* must be kept open until the fault handler is disabled: see *issue with file descriptors*.

在 3.5 版更改: Added support for passing file descriptor to this function.

在 3.6 版更改: On Windows, a handler for Windows exception is also installed.

faulthandler.**disable**()
Disable the fault handler: uninstall the signal handlers installed by *enable()*.

faulthandler.**is_enabled**()
Check if the fault handler is enabled.

27.2.3 Dumping the tracebacks after a timeout

faulthandler.**dump_traceback_later**(*timeout*, *repeat=False*, *file=sys.stderr*, *exit=False*)
Dump the tracebacks of all threads, after a timeout of *timeout* seconds, or every *timeout* seconds if *repeat* is `True`. If *exit* is `True`, call `_exit()` with status=1 after dumping the tracebacks. (Note `_exit()` exits the process immediately, which means it doesn't do any cleanup like flushing file buffers.) If the function is called twice, the new call replaces previous parameters and resets the timeout. The timer has a sub-second resolution.

The *file* must be kept open until the traceback is dumped or *cancel_dump_traceback_later()* is called: see *issue with file descriptors*.

This function is implemented using a watchdog thread and therefore is not available if Python is compiled with threads disabled.

在 3.5 版更改: Added support for passing file descriptor to this function.

faulthandler.**cancel_dump_traceback_later**()
Cancel the last call to *dump_traceback_later()*.

27.2.4 Dumping the traceback on a user signal

faulthandler.**register**(*signum*, *file=sys.stderr*, *all_threads=True*, *chain=False*)
Register a user signal: install a handler for the *signum* signal to dump the traceback of all threads, or of the current thread if *all_threads* is `False`, into *file*. Call the previous handler if chain is `True`.

The *file* must be kept open until the signal is unregistered by *unregister()*: see *issue with file descriptors*.

Not available on Windows.

在 3.5 版更改: Added support for passing file descriptor to this function.

faulthandler.**unregister**(*signum*)
Unregister a user signal: uninstall the handler of the *signum* signal installed by *register()*. Return `True` if the signal was registered, `False` otherwise.

Not available on Windows.

27.2.5 Issue with file descriptors

enable(), *dump_traceback_later()* and *register()* keep the file descriptor of their *file* argument. If the file is closed and its file descriptor is reused by a new file, or if *os.dup2()* is used to replace the file descriptor, the traceback will be written into a different file. Call these functions again each time that the file is replaced.

27.2.6 示例

Example of a segmentation fault on Linux with and without enabling the fault handler:

```
$ python3 -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python3 -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
```

(下页继续)

(续上页)

```
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

27.3 pdb —Python 的调试器

源代码： Lib/pdb.py

---

pdb 模块定义了一个交互式源代码调试器，用于 Python 程序。它支持在源码行间设置（有条件的）断点和单步执行，检视堆栈帧，列出源码列表，以及在任何堆栈帧的上下文中运行任意 Python 代码。它还支持事后调试，可以在程序控制下调用。

调试器是可扩展的——调试器实际被定义为*Pdb* 类。该类目前没有文档，但通过阅读源码很容易理解它。扩展接口使用了*bdb* 和*cmd* 模块。

调试器的提示符是 (Pdb)。在调试器的控制下运行程序的典型用法是:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

在 3.3 版更改: 由*readline* 模块实现的 Tab 补全可用于补全本模块的命令和命令的参数，例如，p 命令的参数可补全为当前的全局变量和局部变量。

也可以将 pdb.py 作为脚本调用来调试其他脚本。例如:

```
python3 -m pdb myscript.py
```

当作为脚本调用时，如果要调试的程序异常退出，pdb 调试将自动进入事后调试。事后调试之后（或程序正常退出之后），pdb 将重新启动程序。自动重启会保留 pdb 的状态（如断点），在大多数情况下，这比在退出程序的同时退出调试器更加实用。

3.2 新版功能: pdb.py 现在接受 -c 选项，可以执行命令，这与将该命令写入 .pdbrc 文件相同，请参阅调试器命令。

从正在运行的程序进入调试器的典型用法是插入

```
import pdb; pdb.set_trace()
```

到你想进入调试器的位置。然后就可以单步执行上述语句之后的代码，要关闭调试器继续运行，请使用*continue* 命令。

检查已崩溃程序的典型用法是:

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print(spam)
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print(spam)
(Pdb)
```

本模块定义了下列函数，每个函数进入调试器的方式略有不同：

pdb.**run**(*statement*, *globals=None*, *locals=None*)
在调试器控制范围内执行 *statement*（以字符串或代码对象的形式提供）。调试器提示符会在执行代码前出现，你可以设置断点并键入`continue`，也可以使用`step`或`next`逐步执行语句（上述所有命令在后文有说明）。可选参数 *globals* 和 *locals* 指定代码执行环境，默认时使用`__main__`模块的字典。（请参阅内置函数`exec()`或`eval()`的说明。）

pdb.**runeval**(*expression*, *globals=None*, *locals=None*)
在调试器控制范围内执行 *expression*（以字符串或代码对象的形式提供）。`runeval()`返回时将返回表达式的值。本函数在其他方面与`run()`类似。

pdb.**runcall**(*function*, **args*, ***kwds*)
使用给定的参数调用 *function*（以函数或方法对象的形式提供，不能是字符串）。`runcall()`返回的是所调用函数的返回值。调试器提示符将在进入函数后立即出现。

pdb.**set_trace**()
Enter the debugger at the calling stack frame. This is useful to hard-code a breakpoint at a given point in a program, even if the code is not otherwise being debugged (e.g. when an assertion fails).

pdb.**post_mortem**(*traceback=None*)
进入 *traceback* 对象的事后调试。如果没有给定 *traceback*，默认使用当前正在处理的异常之一（默认时，必须存在正在处理的异常）。

pdb.**pm**()
在`sys.last_traceback`中查找 traceback，并进入其事后调试。

`run*`函数和`set_trace()`都是别名，用于实例化`Pdb`类和调用同名方法。如果要使用其他功能，则必须自己执行以下操作：

class pdb.**Pdb**(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*, *nosigint=False*, *readrc=True*)
`Pdb`是调试器类。

completekey、*stdin* 和 *stdout* 参数都会传递给底层的`cmd.Cmd`类，请参考相应的描述。

如果给出 *skip* 参数，则它必须是一个迭代器，可以迭代出 glob-style 样式的模块名称。如果遇到匹配上述样式的模块，调试器将不会进入来自该模块的堆栈帧。[1]

默认情况下，当发出 `continue` 命令时，Pdb 将为 SIGINT 信号设置一个处理程序（SIGINT 信号是用户在控制台按 `Ctrl-C` 时发出的）。这使用户可以按 `Ctrl-C` 再次进入调试器。如果希望 Pdb 不要修改 SIGINT 处理程序，请将 *nosigint* 设置为 true。

readrc 参数默认为 true，它控制 Pdb 是否从文件系统加载.pdbrc 文件。

启用跟踪且带有 *skip* 参数的调用示范：

[1] 一个帧是否会被认为源自特定模块是由帧全局变量 `__name__` 来决定的。

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

3.1 新版功能: *skip* 参数。

3.2 新版功能: *nosigint* 参数。在这之前，Pdb 不为 SIGINT 设置处理程序。

在 3.6 版更改: *readrc* 参数。

run(*statement*, *globals=None*, *locals=None*)
runeval(*expression*, *globals=None*, *locals=None*)
runcall(*function*, **args*, ***kwds*)
set_trace()
请参阅上文解释同名函数的文档。

27.3.1 调试器命令

下方列出的是调试器可接受的命令。如下所示，大多数命令可以缩写为一个或两个字母。如 `h(elp)` 表示可以输入 `h` 或 `help` 来输入帮助命令（但不能输入 `he` 或 `hel`，也不能是 `H` 或 `Help` 或 `HELP`）。命令的参数必须用空格（空格符或制表符）分隔。在命令语法中，可选参数括在方括号 (`[]`) 中，使用时请勿输入方括号。命令语法中的选择项由竖线 (`|`) 分隔。

输入一个空白行将重复最后输入的命令。例外：如果最后一个命令是*list* 命令，则会列出接下来的 11 行。

调试器无法识别的命令将被认为是 Python 语句，并在正在调试的程序的上下文中执行。Python 语句也可以用感叹号 (`!`) 作为前缀。这是检查正在调试的程序的强大方法，甚至可以修改变量或调用函数。当此类语句发生异常，将打印异常名称，但调试器的状态不会改变。

调试器支持*别名*。别名可以有参数，使得调试器对被检查的上下文有一定程度的适应性。

在一行中可以输入多个命令，以 `;;` 分隔。（不能使用单个 `;`，因为它用于分隔传递给 Python 解释器的一行中的多条语句。）切分命令很无脑，总是在第一个 `;;` 对处将输入切分开，即使它位于带引号的字符串之中。

如果文件 `.pdbrc` 存在于用户主目录或当前目录中，则将其读入并执行，等同于在调试器提示符下键入该文件。这对于别名很有用。若两个文件都存在则首先读取主目录中的文件，且本地文件可以覆盖其中定义的别名。

在 3.2 版更改: `.pdbrc` 现在可以包含继续调试的命令，如*continue* 或*next*。文件中的这些命令以前是无效的。

h(elp) `[command]`
不带参数时，显示可用的命令列表。参数为 *command* 时，打印有关该命令的帮助。`help pdb` 显示完整文档（即*pdb* 模块的文档字符串）。由于 *command* 参数必须是标识符，因此要获取 `!` 的帮助必须输入 `help exec`。

w(here)
打印堆栈回溯，最新一帧在底部。有一个箭头指向当前帧，该帧决定了大多数命令的上下文。

d(own) `[count]`
在堆栈回溯中，将当前帧向下移动 *count* 级（默认为 1 级，移向更新的帧）。

u(p) `[count]`
在堆栈回溯中，将当前帧向上移动 *count* 级（默认为 1 级，移向更老的帧）。

b(reak) `[([filename:]lineno | function) [, condition]]`
如果带有 *lineno* 参数，则在当前文件相应行处设置一个断点。如果带有 *function* 参数，则在该函数的第一条可执行语句处设置一个断点。行号可以加上文件名和冒号作为前缀，以在另一个文件（可能是尚未加载的文件）中设置一个断点。另一个文件将在*sys.path* 范围内搜索。请注意，每个断点都分配有一个编号，其他所有断点命令都引用该编号。

如果第二个参数存在，它应该是一个表达式，且它的计算值为 true 时断点才起作用。

如果不带参数执行，将列出所有中断，包括每个断点、命中该断点的次数、当前的忽略次数以及关联的条件（如果有）。

tbreak [([filename:]lineno | function) [, condition]]
临时断点，在第一次命中时会自动删除。它的参数与*break* 相同。

cl(ear) [filename:lineno | bpnumber [bpnumber ...]]
如果参数是 *filename:lineno*，则清除此行上的所有断点。如果参数是空格分隔的断点编号列表，则清除这些断点。如果不带参数，则清除所有断点（但会先提示确认）。

disable [bpnumber [bpnumber ...]]
禁用断点，断点以空格分隔的断点编号列表给出。禁用断点表示它不会导致程序停止执行，但是与清除断点不同，禁用的断点将保留在断点列表中并且可以（重新）启用。

enable [bpnumber [bpnumber ...]]
启用指定的断点。

ignore bpnumber [count]
为指定的断点编号设置忽略次数。如果省略 count，则忽略次数将设置为 0。忽略次数为 0 时断点将变为活动状态。如果为非零值，在每次达到断点，且断点未禁用，且关联条件计算值为 true 的情况下，该忽略次数会递减。

condition bpnumber [condition]
为断点设置一个新 *condition*，它是一个表达式，且它的计算值为 true 时断点才起作用。如果没有给出 *condition*，则删除现有条件，也就是将断点设为无条件。

commands [bpnumber]
为编号是 *bpnumber* 的断点指定一系列命令。命令内容将显示在后续的几行中。输入仅包含 end 的行来结束命令列表。举个例子:

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

To remove all commands from a breakpoint, type commands and follow it immediately with end; that is, give no commands.

With no *bpnumber* argument, commands refers to the last breakpoint set.

You can use breakpoint commands to start your program up again. Simply use the continue command, or step, or any other command that resumes execution.

Specifying any command resuming execution (currently continue, step, next, return, jump, quit and their abbreviations) terminates the command list (as if that command was immediately followed by end). This is because any time you resume execution (even with a simple next or step), you may encounter another breakpoint—which could have its own command list, leading to ambiguities about which list to execute.

如果在命令列表中加入‘silent’命令，那么在该断点处停下时就不会打印常规信息。如果希望断点打印特定信息后继续运行，这可能是理想的。如果没有其他命令来打印一些信息，则看不到已达到断点的迹象。

s(tep)
执行当前行，在第一个可以停止的位置（在调用的函数中或在当前函数的下一行）停下。

n(ext)
继续运行，直到运行到当前函数的下一行，或当前函数返回为止。（*next* 和*step* 之间的区别在于，*step* 进入被调用函数内部并停止，而*next* （几乎）全速运行被调用函数，仅在当前函数的下一行停止。）

unt(il) [lineno]
如果不带参数，则继续运行，直到行号比当前行大时停止。

如果带有行号，则继续运行，直到行号大于或等于该行号时停止。在这两种情况下，当前帧返回时也将停止。

在 3.2 版更改: 允许明确给定行号。

r(eturn)
继续运行，直到当前函数返回。

c(ont(inue))
继续运行，仅在遇到断点时停止。

j(ump) lineno
设置即将运行的下一行。仅可用于堆栈最底部的帧。它可以往回跳来再次运行代码，也可以往前跳来跳过不想运行的代码。

需要注意的是，不是所有的跳转都是允许的–例如，不能跳转到 `for` 循环的中间或跳出 `finally` 子句。

l(ist) [first[, last]]
列出当前文件的源代码。如果不带参数，则列出当前行周围的 11 行，或继续前一个列表。如果用 `.` 作为参数，则列出当前行周围的 11 行。如果带有一个参数，则列出那一行周围的 11 行。如果带有两个参数，则列出所给的范围中的代码；如果第二个参数小于第一个参数，则将其解释为列出行数的计数。

当前帧中的当前行用 -> 标记。如果正在调试异常，且最早抛出或传递该异常的行不是当前行，则那一行用 >> 标记。

3.2 新版功能: >> 标记。

ll | longlist
列出当前函数或帧的所有源代码。相关行的标记与*list* 相同。

3.2 新版功能.

a(rgs)
打印当前函数的参数列表。

p expression
在当前上下文中运行 *expression* 并打印它的值。

注解: `print()` 也可以使用，但它不是一个调试器命令—它执行 Python *print()* 函数。

---

pp expression
与*p* 命令类似，但表达式的值使用*pprint* 模块美观地打印。

whatis expression
打印 *expression* 的类型。

source expression
尝试获取给定对象的源代码并显示出来。

3.2 新版功能.

display [expression]
如果表达式的值发生改变则显示它的值，每次将停止执行当前帧。

不带表达式则列出当前帧的所有显示表达式。

3.2 新版功能.

undisplay [expression]
不再显示当前帧中的表达式。不带表达式则清除当前帧的所有显示表达式。

3.2 新版功能.

interact
启动一个交互式解释器（使用*code* 模块），它的全局命名空间将包含当前作用域中的所有（全局和局部）名称。

3.2 新版功能.

alias [name [command]]
创建一个标识为 *name* 的别名来执行 *command*。执行的命令 不可加上引号。可替换形参可通过 %1, %2 等来标示，而 %* 会被所有形参所替换。如果没有给出命令，则会显示 *name* 的当前别名。如果没有给出参数，则会列出所有别名。

别名允许嵌套并可包含能在 pdb 提示符下合法输入的任何内容。请注意内部 pdb 命令 可以被别名所覆盖。这样的命令将被隐藏直到别名被移除。别名会递归地应用到命令行的第一个单词；行内的其他单词不会受影响。

作为示例，这里列出了两个有用的别名（特别适合放在 .pdbrc 文件中）：

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print("%1.",k,"=",%1.__dict__[k])
# Print instance variables in self
alias ps pi self
```

unalias name
删除指定的别名。

! statement
在当前堆栈帧的上下文中执行 (单行) *statement*。感叹号可以被省略，除非语句的第一个单词与调试器命令重名。要设置全局变量，你可以在同一行上为赋值命令添加前缀的 global 语句，例如:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]
restart [args ...]
重启被调试的 Python 程序。如果提供了参数，它会用*shlex* 来拆分且拆分结果将被用作新的*sys.argv*。历史、中断点、动作和调试器选项将被保留。*restart* 是*run* 的一个别名。

q(uit)
退出调试器。被执行的程序将被中止。

备注

27.4 Python 分析器

源代码： Lib/profile.py and Lib/pstats.py

27.4.1 分析器简介

cProfile 和*profile* 提供了 Python 程序的 *deterministic profiling* 。*profile* 是一组统计数据，描述程序的各个部分执行的频率和时间。这些统计数据可以通过*pstats* 模块格式化为报告。

Python 标准库提供了同一分析接口的两种不同实现：

1. 对于大多数用户，建议使用*cProfile*；这是一个 C 扩展插件，开销合理，适合于分析长时间运行的程序。该插件基于 `lsprof`，由 Brett Rosen 和 Ted Chaotter 贡献。
2. *profile* 是一个纯 Python 模块（*cProfile* 就是模仿其接口的 C 实现），但它会显著增加配置程序的开销。如果你正在尝试以某种方式扩展分析器，则使用此模块可能会更容易完成任务。该模块最初由 Jim Roskind 设计和编写。

注解： profiler 分析器模块被设计为给指定的程序提供执行概要文件，而不是用于基准测试目的（*timeit* 才是用于此目标的，它能获得合理准确的结果）。这特别适用于将 Python 代码与 C 代码进行基准测试：分析器为 Python 代码引入开销，但不会为 C 级函数引入开销，因此 C 代码似乎比任何 Python 代码都更快。

27.4.2 即时用户手册

本节是为“不想阅读手册”的用户提供的。它提供了非常简短的概述，并允许用户快速对现有应用程序执行评测。

要分析采用单个参数的函数，可以执行以下操作：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

（如果*cProfile* 在您的系统上不可用，请使用*profile* 。）

上述操作将运行*re.compile()* 并打印分析结果，如下所示：

```
      197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

第一行显示监听了 197 个调用。在这些调用中，有 192 个是 *原始的*，这意味着调用不是通过递归引发的。下一行: `Ordered by: standard name`，表示最右边列中的文本字符串用于对输出进行排序。列标题包括：

ncalls 调用次数

tottime 在指定函数中花费的总时间（不包括调用子函数的时间）

percall 是 `tottime` 除以 `ncalls` 的商

cumtime 指定的函数及其所有子函数（从调用到退出）消耗的累积时间。这个数字对于递归函数来说是准确的。

percall 是 `cumtime` 除以原始调用（次数）的商

filename:lineno(function) 提供相应数据的每个函数

如果第一列中有两个数字（例如 3/1），则表示函数递归。第二个值是原始调用次数，第一个是调用的总次数。请注意，当函数不递归时，这两个值是相同的，并且只打印单个数字。

profile 运行结束时，打印输出不是必须的。也可以通过为 `run()` 函数指定文件名，将结果保存到文件中：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

pstats.Stats 类从文件中读取 profile 结果，并以各种方式对其进行格式化。

The file *cProfile* can also be invoked as a script to profile another script. For example:

```
python -m cProfile [-o output_file] [-s sort_order] myscript.py
```

`-o` 将 profile 结果写入文件而不是标准输出

`-s` 指定*sort_stats()* 排序值之一以对输出进行排序。这仅适用于未提供 `-o` 的情况

The *pstats* module’s *Stats* class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The *strip_dirs()* method removed the extraneous path from all the module names. The *sort_stats()* method sorted all the entries according to the standard module/line/name string that is printed. The *print_stats()* method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats('name')
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats('cumulative').print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats('time').print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

您也可以尝试：

```
p.sort_stats('file').print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats('time', 'cumulative').print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: `.5`) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p` is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the *pstats* module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using *cmd*) and interactive help.

27.4.3 profile 和 cProfile 模块参考

profile 和*cProfile* 模块都提供下列函数：

profile.**run**(*command*, *filename=None*, *sort=-1*)

This function takes a single argument that can be passed to the *exec()* function, and an optional file name. In all cases this routine executes:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a *Stats* instance and prints a simple profiling report. If the sort value is specified, it is passed to this *Stats* instance to control how the results are sorted.

profile.**runctx**(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

This function is similar to *run()*, with added arguments to supply the globals and locals dictionaries for the *command* string. This routine executes:

```
exec(command, globals, locals)
```

and gathers profiling statistics as in the *run()* function above.

class profile.**Profile**(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

This class is normally only used if more precise control over profiling is needed than what the `cProfile.run()` function provides.

A custom timer can be supplied for measuring how long code takes to run via the *timer* argument. This must be a function that returns a single number representing the current time. If the number is an integer, the *timeunit* specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be `.001`.

Directly using the *Profile* class allows formatting profile results without writing the profile data to a file:

```
import cProfile, pstats, io
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
```

(下页继续)

(续上页)

```
s = io.StringIO()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

enable()
Start collecting profiling data.

disable()
Stop collecting profiling data.

create_stats()
停止收集分析数据，并在内部将结果记录为当前 profile。

print_stats(*sort=-1*)
Create a *Stats* object based on the current profile and print the results to stdout.

dump_stats(*filename*)
将当前 profile 的结果写入 *filename* 。

run(*cmd*)
Profile the cmd via *exec()*.

runctx(*cmd*, *globals*, *locals*)
Profile the cmd via *exec()* with the specified global and local environment.

runcall(*func*, **args*, ***kwargs*)
Profile `func(*args, **kwargs)`

Note that profiling will only work if the called command/function actually returns. If the interpreter is terminated (e.g. via a *sys.exit()* call during the called command/function execution) no profiling results will be printed.

27.4.4 Stats 类

Analysis of the profiler data is done using the *Stats* class.

class pstats.**Stats**(**filenames or profile*, *stream=sys.stdout*)
This class constructor creates an instance of a "statistics object" from a *filename* (or list of filenames) or from a `Profile` instance. Output will be printed to the stream specified by *stream*.

The file selected by the above constructor must have been created by the corresponding version of *profile* or *cProfile*. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing *Stats* object, the *add()* method can be used.

Instead of reading the profile data from a file, a `cProfile.Profile` or *profile.Profile* object can be used as the profile data source.

Stats 对象有以下方法:

strip_dirs()
This method for the *Stats* class removes all leading path information from file names. It is very useful in reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If *strip_dirs()* causes two

function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add(**filenames*)
This method of the *Stats* class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of *profile.run()* or `cProfile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

dump_stats(*filename*)
Save the data loaded into the *Stats* object to a file named *filename*. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the *profile.Profile* and `cProfile.Profile` classes.

sort_stats(**keys*)
This method modifies the *Stats* object by sorting it according to the supplied criteria. The argument is typically a string identifying the basis of a sort (example: `'time'` or `'name'`).

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats('name', 'file')` will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

Abbreviations can be used for any key names, as long as the abbreviation is unambiguous. The following are the keys currently defined:

Valid Arg	含义
`'calls'`	调用次数
`'cumulative'`	累积时间
`'cumtime'`	累积时间
`'file'`	文件名
`'filename'`	文件名
`'module'`	文件名
`'ncalls'`	调用次数
`'pcalls'`	原始调用计数
`'line'`	行号
`'name'`	函数名称
`'nfl'`	名称/文件/行
`'stdname'`	标准名称
`'time'`	内部时间
`'tottime'`	内部时间

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between `'nfl'` and `'stdname'` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `'nfl'` does a numeric compare of the line numbers. In fact, `sort_stats('nfl')` is the same as `sort_stats('name', 'file', 'line')`.

For backward-compatibility reasons, the numeric arguments `-1`, `0`, `1`, and `2` are permitted. They are interpreted as `'stdname'`, `'calls'`, `'time'`, and `'cumulative'` respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

reverse_order()
This method for the *Stats* class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

print_stats(**restrictions*)
This method for the *Stats* class prints out a report as described in the *profile.run()* definition.

The order of the printing is based on the last *sort_stats()* operation done on the object (subject to caveats in *add()* and *strip_dirs()*).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a string that will interpreted as a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `.*foo:`. In contrast, the command:

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names `.*foo:`, and then proceed to only print the first 10% of them.

print_callers(**restrictions*)
This method for the *Stats* class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by *print_stats()*, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With *profile*, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With *cProfile*, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

print_callees(**restrictions*)
This method for the *Stats* class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the *print_callers()* method.

27.4.5 什么是确定性性能分析？

确定性性能分析旨在反映这样一个事实：即所有 函数调用、函数返回和 异常事件都被监控，并且对这些事件之间的间隔（在此期间用户的代码正在执行）进行精确计时。相反，统计分析（不是由该模块完成）随机采样有效指令指针，并推断时间花费在哪里。后一种技术传统上涉及较少的开销（因为代码不需要检测），但只提供了时间花在哪里的相对指示。

In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a *hook* (optional callback) for each event. In addition, the interpreted nature of Python tends to add so much overhead to execution, that deterministic profiling tends to only add small processing overhead in typical applications. The result is that deterministic profiling is not that expensive, yet provides extensive run time statistics about the execution of a Python program.

调用计数统计信息可用于识别代码中的错误（意外计数），并识别可能的内联扩展点（高频调用）。内部时间统计可用于识别应仔细优化的“热循环”。累积时间统计可用于识别算法选择上的高级别错误。注意，该分析器中对累积时间的异常处理允许将算法的递归实现与迭代实现的统计信息直接进行比较。

27.4.6 局限性

一个限制是关于时间信息的准确性。确定性性能分析存在一个涉及精度的基本问题。最明显的限制是，底层的“时钟”只以大约 0.001 秒的速度（通常）运行。因此，没有什么测量会比底层时钟更精确。如果进行了足够的测量，那么“误差”将趋于平均。不幸的是，删除第一个错误会导致第二个错误来源。

第二个问题是，从调度事件到分析器获取时间的调用实际获取时钟状态，这需要“一段时间”。类似地，从获取时钟值（然后保存）开始，直到再次执行用户代码为止，退出分析器事件句柄时也存在一定的延迟。因此，多次调用单个函数或调用多个函数通常会累积此错误。以这种方式累积的误差通常小于时钟的精度（小于一个时钟周期），但它 可以累积并变得非常客观。

与开销较低的*cProfile*相比，*profile*的问题更为严重。出于这个原因，*profile*提供了一种针对指定平台的自我校准方法，以便可以在很大程度上（平均地）消除此误差。

27.4.7 准确估量

*profile*模块的 profiler 会从每个事件处理时间中减去一个常量，以补偿调用 time 函数和存储结果的开销。默认情况下，常数为 0。对于特定的平台，可用以下程序获得更好修正常数（*局限性*）。

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running Mac OS X, and using Python’s time.clock() as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is *very* fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

当你有一个一致的答案时，有三种方法可以使用：

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will “less often” show up as negative in profile statistics.

27.4.8 使用自定义计时器

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the `Profile` class constructor:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using *profile.Profile* or `cProfile.Profile`, `your_time_func`'s return value will be interpreted differently:

profile.Profile `your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what *os.times()* returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see 准确估量). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (*os.times()* is *pretty* bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

cProfile.Profile `your_time_func` should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if `your_integer_time_func` returns times measured in thousands of seconds, you would construct the `Profile` instance as follows:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprof` module.

Python 3.3 adds several new functions in *time* that can be used to make precise measurements of process or wall-clock time. For example, see *time.perf_counter()*.

27.5 `timeit` —测量小代码片段的执行时间

源码： Lib/timeit.py

该模块提供了一种简单的方法来计算一小段 Python 代码的耗时。它有命令行界面 以及一个可调用 方法。它避免了许多用于测量执行时间的常见陷阱。另见 Tim Peters 对 O'Reilly 出版的 *Python Cookbook* 中“算法”章节的介绍。

27.5.1 基本示例

以下示例显示了如何使用命令行界面 来比较三个不同的表达式：

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 3: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 3: 27.5 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 3: 23.2 usec per loop
```

这可以通过*Python 接口* 实现

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

Note however that `timeit` will automatically determine the number of repetitions only when the command-line interface is used. In the *例子* section you can find more advanced examples.

27.5.2 Python 接口

该模块定义了三个便利函数和一个公共类:

timeit.**timeit**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *number=1000000*, *globals=None*)
使用给定语句、*setup* 代码和 *timer* 函数创建一个`Timer` 实例，并执行 *number* 次其`timeit()` 方法。可选的 *globals* 参数指定用于执行代码的命名空间。

在 3.5 版更改: 添加可选参数 *globals* 。

timeit.**repeat**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *repeat=3*, *number=1000000*, *globals=None*)
使用给定语句、*setup* 代码和 *timer* 函数创建一个`Timer` 实例，并使用给定的 *repeat* 计数和 *number* 执行运行其`repeat()` 方法。可选的 *globals* 参数指定用于执行代码的命名空间。

在 3.5 版更改: 添加可选参数 *globals* 。

timeit.**default_timer**()
默认的计时器，总是`time.perf_counter()` 。

在 3.3 版更改: `time.perf_counter()` 现在是默认计时器。

class timeit.**Timer**(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*, *globals=None*)
用于小代码片段的计数执行速度的类。

构造函数接受一个将计时的语句、一个用于设置的附加语句和一个定时器函数。两个语句都默认为 `'pass'`；计时器函数与平台有关（请参阅模块文档字符串）。*stmt* 和 *setup* 也可能包含多个以 `;` 或换行符分隔的语句，只要它们不包含多行字符串文字即可。该语句默认在 timeit 的命名空间内执行；可以通过将命名空间传递给 *globals* 来控制此行为。

要测量第一个语句的执行时间，请使用`timeit()` 方法。`repeat()` 和`autorange()` 方法是方便的方法来调用`timeit()` 多次。

setup 的执行时间从总体计时执行中排除。

stmt 和 *setup* 参数也可以使用不带参数的可调用对象。这将在一个计时器函数中嵌入对它们的调用，然后由`timeit()` 执行。请注意，由于额外的函数调用，在这种情况下，计时开销会略大一些。

在 3.5 版更改: 添加可选参数 *globals* 。

timeit(*number=1000000*)
执行 *number* 次主要语句。这将执行一次 setup 语句，然后返回执行主语句多次所需的时间，以秒为单位测量为浮点数。参数是通过循环的次数，默认为一百万。要使用的主语句、setup 语句和 timer 函数将传递给构造函数。

注解： By default, `timeit()` temporarily turns off *garbage collection* during the timing. The advantage of this approach is that it makes independent timings more comparable. This disadvantage is that GC may be

an important component of the performance of the function being measured. If so, GC can be re-enabled as the first statement in the *setup* string. For example:

```
timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
```

autorange(*callback=None*)
自动决定调用多少次*timeit()*。

This is a convenience function that calls *timeit()* repeatedly so that the total time >= 0.2 second, returning the eventual (number of loops, time taken for that number of loops). It calls *timeit()* with *number* set to successive powers of ten (10, 100, 1000, ···) up to a maximum of one billion, until the time taken is at least 0.2 second, or the maximum is reached.

如果给出 *callback* 并且不是 None，则在每次试验后将使用两个参数调用它：`callback(number, time_taken)`。

3.6 新版功能.

repeat(*repeat=3*, *number=1000000*)
调用*timeit()*几次。

这是一个方便的函数，它反复调用*timeit()*，返回结果列表。第一个参数指定调用*timeit()*的次数。第二个参数指定*timeit()*的 *number* 参数。

注解： 从结果向量计算并报告平均值和标准差这些是很诱人的。但是，这不是很有用。在典型情况下，最低值给出了机器运行给定代码段的速度的下限；结果向量中较高的值通常不是由 Python 的速度变化引起的，而是由于其他过程干扰你的计时准确性。所以结果的*min()*可能是你应该感兴趣的唯一数字。之后，你应该看看整个向量并应用常识而不是统计。

print_exc(*file=None*)
帮助程序从计时代码中打印回溯。

典型使用:

```
t = Timer(...)       # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except Exception:
    t.print_exc()
```

与标准回溯相比，优势在于将显示已编译模板中的源行。可选的 *file* 参数指向发送回溯的位置；它默认为*sys.stderr*。

27.5.3 命令行界面

从命令行调用程序时，使用以下表单:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-t] [-c] [-h] [statement ...]
```

如果了解以下选项:

-n N, **--number**=N
执行‘语句’多少次

-r N, **--repeat**=N
how many times to repeat the timer (default 3)

-s S, **--setup**=S
最初要执行一次的语句（默认为 pass ）

-p, **--process**
测量进程时间，而不是 wallclock 时间，使用*time.process_time()* 而不是*time.perf_counter()* ，这是默认值

3.3 新版功能.

-t, **--time**
use *time.time()* (deprecated)

-u, **--unit**=U

specify a time unit for timer output; can select usec, msec, or sec

3.5 新版功能.

-c, **--clock**
use *time.clock()* (deprecated)

-v, **--verbose**
打印原始计时结果；重复更多位数精度

-h, **--help**
打印一条简短的使用信息并退出

可以通过将每一行指定为单独的语句参数来给出多行语句；通过在引号中包含参数并使用前导空格可以缩进行。多个*-s* 选项的处理方式相似。

If *-n* is not given, a suitable number of loops is calculated by trying successive powers of 10 until the total time is at least 0.2 seconds.

default_timer() measurements can be affected by other programs running on the same machine, so the best thing to do when accurate timing is necessary is to repeat the timing a few times and use the best time. The *-r* option is good for this; the default of 3 repetitions is probably enough in most cases. You can use *time.process_time()* to measure CPU time.

注解： 执行 pass 语句会产生一定的基线开销。这里的代码不会试图隐藏它，但你应该知道它。可以通过不带参数调用程序来测量基线开销，并且 Python 版本之间可能会有所不同。

27.5.4 例子

可以提供一个在开头只执行一次的 setup 语句：

```
$ python -m timeit -s 'text = "sample string"; char = "g"'  'char in text'
10000000 loops, best of 3: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"'  'text.find(char)'
1000000 loops, best of 3: 0.342 usec per loop
```

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

使用*Timer* 类及其方法可以完成同样的操作：

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40193588800002544, 0.3960157959998014, 0.39594301399984033]
```

以下示例显示如何计算包含多行的表达式。在这里我们对比使用*hasattr()* 与 try/except 的开销来测试缺失与提供对象属性:

```
$ python -m timeit 'try:' '  str.__bool__' 'except AttributeError:' '  pass'
100000 loops, best of 3: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__bool__"): pass'
100000 loops, best of 3: 4.26 usec per loop

$ python -m timeit 'try:' '  int.__bool__' 'except AttributeError:' '  pass'
1000000 loops, best of 3: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, "__bool__"): pass'
100000 loops, best of 3: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

要让*timeit* 模块访问你定义的函数，你可以传递一个包含 import 语句的 *setup* 参数:

```
def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

另一种选择是将*globals()* 传递给 *globals* 参数，这将导致代码在当前的全局命名空间中执行。这比单独指

定 import 更方便

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

27.6 trace —跟踪 Python 语句执行

源代码: Lib/trace.py

模块*trace* module 允许你跟踪程序的执行过程，生成带注释的语句覆盖率列表，打印调用/被调用关系以及列出在程序运行期间执行过的函数。可以在其他程序或者命令行中使用它

参见:

Coverage.py A popular third-party coverage tool that provides HTML output along with advanced features such as branch coverage.

27.6.1 Command-Line Usage

The *trace* module can be invoked from the command line. It can be as simple as

```
python -m trace --count -C . somefile.py ...
```

The above will execute `somefile.py` and generate annotated listings of all Python modules imported during the execution into the current directory.

--help
Display usage and exit.

--version
Display the version of the module and exit.

Main options

At least one of the following options must be specified when invoking *trace*. The *--listfuncs* option is mutually exclusive with the *--trace* and *--count* options. When *--listfuncs* is provided, neither *--count* nor *--trace* are accepted, and vice versa.

-c, --count
Produce a set of annotated listing files upon program completion that shows how many times each statement was executed. See also *--coverdir*, *--file* and *--no-report* below.

-t, --trace
Display lines as they are executed.

-l, --listfuncs
Display the functions executed by running the program.

-r, --report
Produce an annotated list from an earlier program run that used the *--count* and *--file* option. This does not execute any code.

-T, --trackcalls
Display the calling relationships exposed by running the program.

Modifiers

-f, --file=<file>
Name of a file to accumulate counts over several tracing runs. Should be used with the *--count* option.

-C, --coverdir=<dir>
Directory where the report files go. The coverage report for `package.module` is written to file *dir/package/module*`.cover`.

-m, --missing
When generating annotated listings, mark lines which were not executed with >>>>>>.

-s, --summary
When using *--count* or *--report*, write a brief summary to stdout for each file processed.

-R, --no-report
Do not generate annotated listings. This is useful if you intend to make several runs with *--count*, and then produce a single set of annotated listings at the end.

-g, --timing
Prefix each line with the time since the program started. Only used while tracing.

Filters

These options may be repeated multiple times.

--ignore-module=<mod>
Ignore each of the given module names and its submodules (if it is a package). The argument can be a list of names separated by a comma.

--ignore-dir=<dir>
Ignore all modules and packages in the named directory and subdirectories. The argument can be a list of directories separated by *os.pathsep*.

27.6.2 编程接口

class `trace.`**Trace**(*count=1*, *trace=1*, *countfuncs=0*, *countcallers=0*, *ignoremods=()*, *ignoredirs=()*, *infile=None*, *outfile=None*, *timing=False*)
Create an object to trace execution of a single statement or expression. All parameters are optional. *count* enables counting of line numbers. *trace* enables line execution tracing. *countfuncs* enables listing of the functions called during the run. *countcallers* enables call relationship tracking. *ignoremods* is a list of modules or packages to ignore. *ignoredirs* is a list of directories whose modules or packages should be ignored. *infile* is the name of the file from which to read stored count information. *outfile* is the name of the file in which to write updated count information. *timing* enables a timestamp relative to when tracing was started to be displayed.

run(*cmd*)
Execute the command and gather statistics from the execution with the current tracing parameters. *cmd* must be a string or code object, suitable for passing into *exec()*.

runctx(*cmd*, *globals=None*, *locals=None*)
Execute the command and gather statistics from the execution with the current tracing parameters, in the defined global and local environments. If not defined, *globals* and *locals* default to empty dictionaries.

runfunc(*func*, **args*, ***kwds*)
Call *func* with the given arguments under control of the `Trace` object with the current tracing parameters.

results()
Return a `CoverageResults` object that contains the cumulative results of all previous calls to `run`, `runctx` and `runfunc` for the given `Trace` instance. Does not reset the accumulated trace results.

class trace.**CoverageResults**
A container for coverage results, created by `Trace.results()`. Should not be created directly by the user.

update(*other*)
Merge in data from another `CoverageResults` object.

write_results(*show_missing=True*, *summary=False*, *coverdir=None*)
Write coverage results. Set *show_missing* to show lines that had no hits. Set *summary* to include in the output the coverage summary per module. *coverdir* specifies the directory into which the coverage result files will be output. If `None`, the results for each source file are placed in its directory.

A simple example demonstrating the use of the programmatic interface:

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)

# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

27.7 tracemalloc —跟踪内存分配

3.4 新版功能.

源代码： Lib/tracemalloc.py

The tracemalloc module is a debug tool to trace memory blocks allocated by Python. It provides the following information:

- Traceback where an object was allocated
- Statistics on allocated memory blocks per filename and per line number: total size, number and average size of allocated memory blocks

- Compute the differences between two snapshots to detect memory leaks

To trace most memory blocks allocated by Python, the module should be started as early as possible by setting the `PYTHONTRACEMALLOC` environment variable to `1`, or by using `-X tracemalloc` command line option. The *`tracemalloc.start()`* function can be called at runtime to start tracing Python memory allocations.

By default, a trace of an allocated memory block only stores the most recent frame (1 frame). To store 25 frames at startup: set the `PYTHONTRACEMALLOC` environment variable to `25`, or use the `-X tracemalloc=25` command line option.

27.7.1 例子

显示前 10 项

显示内存分配最多的 10 个文件：

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Python 测试套件的输出示例：

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108␣
↪B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

We can see that Python loaded `4855 KiB` data (bytecode and constants) from modules and that the *`collections`* module allocated `244 KiB` to build *`namedtuple`* types.

See *`Snapshot.statistics()`* for more options.

计算差异

获取两个快照并显示差异:

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output before/after running some tests of the Python test suite:

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369),␣
↪average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106),␣
↪average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589),␣
↪average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526),␣
↪average=139 B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334),␣
↪average=86 B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1),␣
↪average=96.0 KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109),␣
↪average=784 B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143),␣
↪average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969),␣
↪average=76 B
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126),␣
↪average=546 B
```

We can see that Python has loaded `8173 KiB` of module data (bytecode and constants), and that this is `4428 KiB` more than had been loaded before the tests, when the previous snapshot was taken. Similarly, the *linecache* module has cached `940 KiB` of Python source code to format tracebacks, all of it since the previous snapshot.

If the system has little free memory, snapshots can be written on disk using the *Snapshot.dump()* method to analyze the snapshot offline. Then use the *Snapshot.load()* method reload the snapshot.

Get the traceback of a memory block

Code to display the traceback of the biggest memory block:

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

Example of output of the Python test suite (traceback limited to 25 frames):

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

We can see that the most memory was allocated in the *importlib* module to load data (bytecode and constants) from

modules: 870.1 KiB. The traceback is where the *importlib* loaded data most recently: on the import pdb line of the *doctest* module. The traceback may change if a new module is loaded.

Pretty top

Code to display the 10 lines allocating the most memory with a pretty output, ignoring <frozen importlib._bootstrap> and <unknown> files:

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        # replace "/path/to/module/file.py" with "module/file.py"
        filename = os.sep.join(frame.filename.split(os.sep)[-2:])
        print("#%s: %s:%s: %.1f KiB"
              % (index, filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Python 测试套件的输出示例：

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
```

(下页继续)

(续上页)

```
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

See *Snapshot.statistics()* for more options.

27.7.2 API

函数

tracemalloc.**clear_traces**()
Clear traces of memory blocks allocated by Python.

See also *stop()*.

tracemalloc.**get_object_traceback**(*obj*)
Get the traceback where the Python object *obj* was allocated. Return a *Traceback* instance, or None if the *tracemalloc* module is not tracing memory allocations or did not trace the allocation of the object.

See also *gc.get_referrers()* and *sys.getsizeof()* functions.

tracemalloc.**get_traceback_limit**()
Get the maximum number of frames stored in the traceback of a trace.

The *tracemalloc* module must be tracing memory allocations to get the limit, otherwise an exception is raised.

The limit is set by the *start()* function.

tracemalloc.**get_traced_memory**()
Get the current size and peak size of memory blocks traced by the *tracemalloc* module as a tuple: (current: int, peak: int).

tracemalloc.**get_tracemalloc_memory**()
Get the memory usage in bytes of the *tracemalloc* module used to store traces of memory blocks. Return an *int*.

tracemalloc.**is_tracing**()
True if the *tracemalloc* module is tracing Python memory allocations, False otherwise.

See also *start()* and *stop()* functions.

tracemalloc.**start**(*nframe: int=1*)
Start tracing Python memory allocations: install hooks on Python memory allocators. Collected tracebacks of traces will be limited to *nframe* frames. By default, a trace of a memory block only stores the most recent frame: the limit is 1. *nframe* must be greater or equal to 1.

Storing more than 1 frame is only useful to compute statistics grouped by 'traceback' or to compute cumulative statistics: see the *Snapshot.compare_to()* and *Snapshot.statistics()* methods.

Storing more frames increases the memory and CPU overhead of the *tracemalloc* module. Use the *get_tracemalloc_memory()* function to measure how much memory is used by the *tracemalloc* module.

The PYTHONTRACEMALLOC environment variable (PYTHONTRACEMALLOC=NFRAME) and the -X tracemalloc=NFRAME command line option can be used to start tracing at startup.

See also *stop()*, *is_tracing()* and *get_traceback_limit()* functions.

tracemalloc.**stop**()
Stop tracing Python memory allocations: uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Call *take_snapshot()* function to take a snapshot of traces before clearing them.

See also *start()*, *is_tracing()* and *clear_traces()* functions.

tracemalloc.**take_snapshot**()
Take a snapshot of traces of memory blocks allocated by Python. Return a new *Snapshot* instance.

The snapshot does not include memory blocks allocated before the *tracemalloc* module started to trace memory allocations.

Tracebacks of traces are limited to *get_traceback_limit()* frames. Use the *nframe* parameter of the *start()* function to store more frames.

The *tracemalloc* module must be tracing memory allocations to take a snapshot, see the *start()* function.

See also the *get_object_traceback()* function.

域过滤器

class tracemalloc.**DomainFilter**(*inclusive: bool*, *domain: int*)
Filter traces of memory blocks by their address space (domain).

3.6 新版功能.

inclusive
If *inclusive* is True (include), match memory blocks allocated in the address space *domain*.

If *inclusive* is False (exclude), match memory blocks not allocated in the address space *domain*.

domain
Address space of a memory block (int). Read-only property.

过滤器

class tracemalloc.**Filter**(*inclusive: bool*, *filename_pattern: str*, *lineno: int=None*, *all_frames: bool=False*, *domain: int=None*)
对内存块的跟踪进行筛选。

See the *fnmatch.fnmatch()* function for the syntax of *filename_pattern*. The '.pyc' file extension is replaced with '.py'.

示例：

- Filter(True, subprocess.__file__) only includes traces of the *subprocess* module
- Filter(False, tracemalloc.__file__) excludes traces of the *tracemalloc* module
- Filter(False, "<unknown>") excludes empty tracebacks

在 3.5 版更改: The `'.pyo'` file extension is no longer replaced with `'.py'`.

在 3.6 版更改: Added the *domain* attribute.

domain
Address space of a memory block (`int` or `None`).

inclusive
If *inclusive* is `True` (include), only match memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

If *inclusive* is `False` (exclude), ignore memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

lineno
Line number (`int`) of the filter. If *lineno* is `None`, the filter matches any line number.

filename_pattern
Filename pattern of the filter (`str`). Read-only property.

all_frames
If *all_frames* is `True`, all frames of the traceback are checked. If *all_frames* is `False`, only the most recent frame is checked.

This attribute has no effect if the traceback limit is `1`. See the *get_traceback_limit()* function and *Snapshot.traceback_limit* attribute.

Frame

class tracemalloc.**Frame**
Frame of a traceback.

The *Traceback* class is a sequence of *Frame* instances.

filename
文件名 (`str`).

lineno
行号 (`int`).

快照

class tracemalloc.**Snapshot**
Snapshot of traces of memory blocks allocated by Python.

The *take_snapshot()* function creates a snapshot instance.

compare_to(*old_snapshot: Snapshot*, *key_type: str*, *cumulative: bool=False*)
Compute the differences with an old snapshot. Get statistics as a sorted list of *StatisticDiff* instances grouped by *key_type*.

See the *Snapshot.statistics()* method for *key_type* and *cumulative* parameters.

The result is sorted from the biggest to the smallest by: absolute value of *StatisticDiff.size_diff*, *StatisticDiff.size*, absolute value of *StatisticDiff.count_diff*, *Statistic.count* and then by *StatisticDiff.traceback*.

dump(*filename*)
将快照写入文件

使用*load()* 重载快照。

filter_traces(*filters*)
Create a new *Snapshot* instance with a filtered *traces* sequence, *filters* is a list of *DomainFilter* and *Filter* instances. If *filters* is an empty list, return a new *Snapshot* instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matches it.

在 3.6 版更改: *DomainFilter* instances are now also accepted in *filters*.

classmethod load(*filename*)
从文件载入快照。

另见*dump()*.

statistics(*key_type: str*, *cumulative: bool=False*)
获取*Statistic*信息列表，按 *key_type* 分组排序：

key_type	描述
`'filename'`	filename
`'lineno'`	文件名和行号
`'traceback'`	回溯

If *cumulative* is `True`, cumulate size and count of memory blocks of all frames of the traceback of a trace, not only the most recent frame. The cumulative mode can only be used with *key_type* equals to `'filename'` and `'lineno'`.

The result is sorted from the biggest to the smallest by: *Statistic.size*, *Statistic.count* and then by *Statistic.traceback*.

traceback_limit
Maximum number of frames stored in the traceback of *traces*: result of the *get_traceback_limit()* when the snapshot was taken.

traces
Traces of all memory blocks allocated by Python: sequence of *Trace* instances.

The sequence has an undefined order. Use the *Snapshot.statistics()* method to get a sorted list of statistics.

统计

class tracemalloc.**Statistic**
统计内存分配

Snapshot.statistics() 返回*Statistic*实例的列表。.

参见*StatisticDiff*类。

count
内存块数（整形）。

size
Total size of memory blocks in bytes (`int`).

traceback
Traceback where the memory block was allocated, *Traceback* instance.

StatisticDiff

class tracemalloc.**StatisticDiff**
Statistic difference on memory allocations between an old and a new *Snapshot* instance.

Snapshot.compare_to() returns a list of *StatisticDiff* instances. See also the *Statistic* class.

count
Number of memory blocks in the new snapshot (int): 0 if the memory blocks have been released in the new snapshot.

count_diff
Difference of number of memory blocks between the old and the new snapshots (int): 0 if the memory blocks have been allocated in the new snapshot.

size
Total size of memory blocks in bytes in the new snapshot (int): 0 if the memory blocks have been released in the new snapshot.

size_diff
Difference of total size of memory blocks in bytes between the old and the new snapshots (int): 0 if the memory blocks have been allocated in the new snapshot.

traceback
Traceback where the memory blocks were allocated, *Traceback* instance.

跟踪

class tracemalloc.**Trace**
Trace of a memory block.

The *Snapshot.traces* attribute is a sequence of *Trace* instances.

size
Size of the memory block in bytes (int).

traceback
Traceback where the memory block was allocated, *Traceback* instance.

回溯

class tracemalloc.**Traceback**
Sequence of *Frame* instances sorted from the most recent frame to the oldest frame.

A traceback contains at least 1 frame. If the tracemalloc module failed to get a frame, the filename "<unknown>" at line number 0 is used.

When a snapshot is taken, tracebacks of traces are limited to *get_traceback_limit()* frames. See the *take_snapshot()* function.

The *Trace.traceback* attribute is an instance of *Traceback* instance.

format(*limit=None*)
Format the traceback as a list of lines with newlines. Use the *linecache* module to retrieve lines from the source code. If *limit* is set, only format the *limit* most recent frames.

Similar to the *traceback.format_tb()* function, except that *format()* does not include newlines.

示例:

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

输出:

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

CHAPTER 28

软件打包和分发

这些库可帮助你发布和安装 Python 软件。虽然这些模块设计为与 'Python 包索引 <https://pypi.org>'__ 结合使用，但它们也可以与本地索引服务器一起使用，或者根本不使用任何索引服务器。

28.1 `distutils` —构建和安装 Python 模块

`distutils` 包为将待构建和安装的额外的模块，打包成 Python 安装包提供支持。新模块既可以是百分百的纯 Python，也可以是用 C 写的扩展模块，或者可以是一组包含了同时用 Python 和 C 编码的 Python 包。

大多数 Python 用户 不会想要直接使用这个包，而是使用 Python 包官方维护的跨版本工具。特别地，setuptools 是一个对于提供的*`distutils`* 的增强选项。

- 对声明项目依赖的支持。
- 额外的用于配置哪些文件包含在源代码发布中的装置（包括与版本控制系统集成需要的插件）
- 生成项目“进入点”的能力，进入点可用作应用插件系统的基础
- 自动在安装时间生成 Windows 命令行可执行文件的能力，而不是需要预编译它们
- 跨所有受支持的 Python 版本上的一致的表现

推荐的 pip 安装器用 `setuptools` 运行所有的 `setup.py` 脚本，即使脚本本身只引了 `distutils` 包。参考 Python Packaging User Guide 获得更多信息。

为了打包工具的作者和用户能更好理解当前的打包和分发系统，遗留的基于*`distutils`* 的用户文档和 API 参考保持可用：

- install-index
- distutils-index

28.2 ensurepip —Bootstrapping the pip installer

3.4 新版功能.

The *ensurepip* package provides support for bootstrapping the pip installer into an existing Python installation or virtual environment. This bootstrapping approach reflects the fact that pip is an independent project with its own release cycle, and the latest available stable version is bundled with maintenance and feature releases of the CPython reference interpreter.

In most cases, end users of Python shouldn't need to invoke this module directly (as pip should be bootstrapped by default), but it may be needed if installing pip was skipped when installing Python (or when creating a virtual environment) or after explicitly uninstalling pip.

注解: This module *does not* access the internet. All of the components needed to bootstrap pip are included as internal parts of the package.

参见:

installing-index The end user guide for installing Python packages

PEP 453: Explicit bootstrapping of pip in Python installations The original rationale and specification for this module.

28.2.1 Command line interface

The command line interface is invoked using the interpreter's -m switch.

The simplest possible invocation is:

```
python -m ensurepip
```

This invocation will install pip if it is not already installed, but otherwise does nothing. To ensure the installed version of pip is at least as recent as the one bundled with ensurepip, pass the --upgrade option:

```
python -m ensurepip --upgrade
```

By default, pip is installed into the current virtual environment (if one is active) or into the system site packages (if there is no active virtual environment). The installation location can be controlled through two additional command line options:

- --root <dir>: Installs pip relative to the given root directory rather than the root of the currently active virtual environment (if any) or the default root for the current Python installation.
- --user: Installs pip into the user site packages directory rather than globally for the current Python installation (this option is not permitted inside an active virtual environment).

By default, the scripts pipX and pipX.Y will be installed (where X.Y stands for the version of Python used to invoke ensurepip). The scripts installed can be controlled through two additional command line options:

- --altinstall: if an alternate installation is requested, the pipX script will *not* be installed.
- **--default-pip: if a "default pip" installation is requested, the** pip script will be installed in addition to the two regular scripts.

Providing both of the script selection options will trigger an exception.

在 3.6.3 版更改: The exit status is non-zero if the command fails.

28.2.2 Module API

ensurepip exposes two functions for programmatic use:

ensurepip.**version**()
Returns a string specifying the bundled version of pip that will be installed when bootstrapping an environment.

ensurepip.**bootstrap**(*root=None*, *upgrade=False*, *user=False*, *altinstall=False*, *default_pip=False*, *verbosity=0*)
Bootstraps `pip` into the current or designated environment.

root specifies an alternative root directory to install relative to. If *root* is `None`, then installation uses the default install location for the current environment.

upgrade indicates whether or not to upgrade an existing installation of an earlier version of `pip` to the bundled version.

user indicates whether to use the user scheme rather than installing globally.

By default, the scripts `pipX` and `pipX.Y` will be installed (where X.Y stands for the current version of Python).

If *altinstall* is set, then `pipX` will *not* be installed.

If *default_pip* is set, then `pip` will be installed in addition to the two regular scripts.

Setting both *altinstall* and *default_pip* will trigger *ValueError*.

verbosity controls the level of output to *sys.stdout* from the bootstrapping operation.

注解： The bootstrapping process has side effects on both `sys.path` and `os.environ`. Invoking the command line interface in a subprocess instead allows these side effects to be avoided.

注解： The bootstrapping process may install additional modules required by `pip`, but other software should not assume those dependencies will always be present by default (as the dependencies may be removed in a future version of `pip`).

28.3 venv —创建虚拟环境

3.3 新版功能.

源码： Lib/venv/

The *venv* module provides support for creating lightweight "virtual environments" with their own site directories, optionally isolated from system site directories. Each virtual environment has its own Python binary (allowing creation of environments with various Python versions) and can have its own independent set of installed Python packages in its site directories.

有关 Python 虚拟环境的更多信息，请参阅 **PEP 405** 。

注解： 从 Python 3.6 开始，不推荐使用 `pyvenv` 脚本，而是使用 `python3 -m venv` 来帮助防止任何关于虚拟环境将基于哪个 Python 解释器的混淆。

28.3.1 创建虚拟环境

通过执行 venv 指令来创建一个虚拟环境:

```
python3 -m venv /path/to/new/virtual/environment
```

Running this command creates the target directory (creating any parent directories that don't exist already) and places a pyvenv.cfg file in it with a home key pointing to the Python installation from which the command was run. It also creates a bin (or Scripts on Windows) subdirectory containing a copy/symlink of the Python binary/binaries (as appropriate for the platform or arguments used at environment creation time). It also creates an (initially empty) lib/pythonX.Y/site-packages subdirectory (on Windows, this is Lib\site-packages). If an existing directory is specified, it will be re-used.

3.6 版后已移除: pyvenv 是 Python 3.3 和 3.4 中创建虚拟环境的推荐工具，不过 在 Python 3.6 中已弃用。

在 3.5 版更改: 现在推荐使用 venv 来创建虚拟环境。

参见:

Python 软件包用户指南：创建和使用虚拟环境

在 Windows 上，调用 venv 命令如下:

```
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

或者，如果已经为 Python 安装配置好 PATH 和 PATHEXT 变量:

```
c:\>python -m venv c:\path\to\myenv
```

本命令如果以 -h 参数运行，将显示可用的选项:

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

positional arguments:
  ENV_DIR               A directory to create the environment in.

optional arguments:
  -h, --help            show this help message and exit
  --system-site-packages
                        Give the virtual environment access to the system
                        site-packages dir.
  --symlinks            Try to use symlinks rather than copies, when symlinks
                        are not the default for the platform.
  --copies              Try to use copies rather than symlinks, even when
                        symlinks are the default for the platform.
  --clear               Delete the contents of the environment directory if it
                        already exists, before environment creation.
  --upgrade             Upgrade the environment directory to use this version
                        of Python, assuming Python has been upgraded in-place.
  --without-pip         Skips installing or upgrading pip in the virtual
                        environment (pip is bootstrapped by default)

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

在 3.4 版更改: 默认安装 pip，并添加 --without-pip 和 --copies 选项

在 3.4 版更改: 在早期版本中，如果目标目录已存在，将引发错误，除非使用了 `--clear` 或 `--upgrade` 选项。

生成的 `pyvenv.cfg` 文件还包括 `include-system-site-packages` 键，如果运行 `venv` 带有 `--system-site-packages` 选项，则键值为 `true`，否则为 `false`。

除非采用 `--without-pip` 选项，否则将会调用*ensurepip* 将 `pip` 引导到虚拟环境中。

可以向 `venv` 传入多个路径，此时将根据给定的选项，在所给的每个路径上创建相同的虚拟环境。

创建虚拟环境后，可以使用虚拟环境的二进制目录中的脚本来“激活”该环境。不同平台调用的脚本是不同的（须将 <*venv*> 替换为包含虚拟环境的目录路径）：

平台	Shell	用于激活虚拟环境的命令
Posix	bash/zsh	$ source <venv>/bin/activate
	fish	$. <venv>/bin/activate.fish
	csh/tcsh	$ source <venv>/bin/activate.csh
Windows	cmd.exe	C:\> <venv>\Scripts\activate.bat
	PowerShell	PS C:\> <venv>\Scripts\Activate.ps1

激活环境不是 *必须的*，激活只是将虚拟环境的二进制目录添加到搜索路径中，这样“python”命令将调用虚拟环境的 Python 解释器，可以运行其中已安装的脚本，而不必输入其完整路径。但是，安装在虚拟环境中的所有脚本都应在不激活的情况下可运行，并自动与虚拟环境的 Python 一起运行。

You can deactivate a virtual environment by typing “deactivate” in your shell. The exact mechanism is platform-specific: for example, the Bash activation script defines a “deactivate” function, whereas on Windows there are separate scripts called `deactivate.bat` and `Deactivate.ps1` which are installed when the virtual environment is created.

3.4 新版功能: `fish` 和 `csh` 激活脚本。

注解: 虚拟环境是一个 Python 环境，安装到其中的 Python 解释器、库和脚本与其他虚拟环境中的内容是隔离的，且（默认）与“系统级”Python（操作系统的一部分）中安装的库是隔离的。

虚拟环境是一个目录树，其中包含 Python 可执行文件和其他文件，其他文件指示了这是一个是虚拟环境。

Common installation tools such as `Setuptools` and `pip` work as expected with virtual environments. In other words, when a virtual environment is active, they install Python packages into the virtual environment without needing to be told to do so explicitly.

当虚拟环境被激活（即虚拟环境的 Python 解释器正在运行），属性*sys.prefix* 和*sys.exec_prefix* 指向的是虚拟环境的基础目录，而*sys.base_prefix* 和*sys.base_exec_prefix* 指向非虚拟环境的 Python 安装，即曾用于创建虚拟环境的那个 Python 安装。如果虚拟环境没有被激活，则*sys.prefix* 与*sys.base_prefix* 相同，且*sys.exec_prefix* 与*sys.base_exec_prefix* 相同（它们均指向非虚拟环境的 Python 安装）。

When a virtual environment is active, any options that change the installation path will be ignored from all distutils configuration files to prevent projects being inadvertently installed outside of the virtual environment.

When working in a command shell, users can make a virtual environment active by running an `activate` script in the virtual environment’s executables directory (the precise filename is shell-dependent), which prepends the virtual environment’s directory for executables to the `PATH` environment variable for the running shell. There should be no need in other circumstances to activate a virtual environment—scripts installed into virtual environments have a “shebang” line which points to the virtual environment’s Python interpreter. This means that the script will run with that interpreter regardless of the value of `PATH`. On Windows, “shebang” line processing is supported if you have the Python Launcher for Windows installed (this was added to Python in 3.3 - see **PEP 397** for more details). Thus, double-clicking an installed script in a Windows Explorer window should run the script with the correct interpreter without there needing to be any reference to its virtual environment in `PATH`.

28.3.2 API

上述的高级方法使用了一个简单的 API，该 API 提供了一种机制，第三方虚拟环境创建者可以根据其需求自定义环境创建过程，该 API 为*EnvBuilder* 类。

class venv.**EnvBuilder**(*system_site_packages=False*, *clear=False*, *symlinks=False*, *upgrade=False*, *with_pip=False*, *prompt=None*)

EnvBuilder 类在实例化时接受以下关键字参数：

- `system_site_packages` –一个布尔值，要求系统 Python 的 site-packages 对环境可用（默认为 `False`）。
- `clear` –一个布尔值，如果为 true，则在创建环境前将删除目标目录的现有内容。
- `symlinks` –a Boolean value indicating whether to attempt to symlink the Python binary (and any necessary DLLs or other binaries, e.g. `pythonw.exe`), rather than copying.
- `upgrade` –一个布尔值，如果为 true，则将使用当前运行的 Python 去升级一个现有的环境，这主要在原位置的 Python 更新后使用（默认为 `False`）。
- `with_pip` –一个布尔值，如果为 true，则确保在虚拟环境中已安装 pip。这使用的是带有 `--default-pip` 选项的*ensurepip*。
- `prompt` –a String to be used after virtual environment is activated (defaults to `None` which means directory name of the environment would be used).

在 3.4 版更改: 添加 `with_pip` 参数

3.6 新版功能: 添加 `prompt` 参数

Creators of third-party virtual environment tools will be free to use the provided `EnvBuilder` class as a base class.

返回的 env-builder 是一个对象，包含一个 `create` 方法：

create(*env_dir*)

This method takes as required argument the path (absolute or relative to the current directory) of the target directory which is to contain the virtual environment. The `create` method will either create the environment in the specified directory, or raise an appropriate exception.

The `create` method of the `EnvBuilder` class illustrates the hooks available for subclass customization:

```
def create(self, env_dir):
    """
    Create a virtualized Python environment in a directory.
    env_dir is the target directory to create an environment in.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

每个方法*ensure_directories()*, *create_configuration()*, *setup_python()*, *setup_scripts()* 和*post_setup()* 都可以被重写。

ensure_directories(*env_dir*)

创建环境目录和所有必需的目录，并返回一个上下文对象。该对象只是一个容器，保存属性（如路径），供其他方法使用。允许目录已经存在，如果指定了 `clear` 或 `upgrade` 就允许在现有环境目录上进行操作。

create_configuration(*context*)
在环境中创建 pyvenv.cfg 配置文件。

setup_python(*context*)
Creates a copy of the Python executable (and, under Windows, DLLs) in the environment. On a POSIX system, if a specific executable python3.x was used, symlinks to python and python3 will be created pointing to that executable, unless files with those names already exist.

setup_scripts(*context*)
将适用于平台的激活脚本安装到虚拟环境中。

post_setup(*context*)
占位方法，可以在第三方实现中重写，用于在虚拟环境中预安装软件包，或是其他创建后要执行的步骤。

此外，*EnvBuilder* 提供了如下实用方法，可以从子类的*setup_scripts()* 或*post_setup()* 调用，用来将自定义脚本安装到虚拟环境中。

install_scripts(*context*, *path*)
path 是一个目录的路径，该目录应包含子目录 “common”，“posix”，“nt”，每个子目录存有发往对应环境中 bin 目录的脚本。在下列占位符替换完毕后，将复制 “common” 的内容和与*os.name* 对应的子目录：

- __VENV_DIR__ 会被替换为环境目录的绝对路径。
- __VENV_NAME__ 会被替换为环境名称（环境目录的最后一个字段）。
- __VENV_PROMPT__ 会被替换为提示符（用括号括起来的环境名称紧跟着一个空格）。
- __VENV_BIN_NAME__ 会被替换为 bin 目录的名称（bin 或 Scripts ）。
- __VENV_PYTHON__ 会被替换为环境可执行文件的绝对路径。

允许目录已存在（用于升级现有环境时）。

有一个方便实用的模块级别的函数：

venv.**create**(*env_dir*, *system_site_packages=False*, *clear=False*, *symlinks=False*, *with_pip=False*)
通过关键词参数来创建一个*EnvBuilder*，并且使用 *env_dir* 参数来调用它的*create()* 方法。

在 3.4 版更改: 添加 with_pip 参数

28.3.3 一个扩展 EnvBuilder 的例子

下面的脚本展示了如何通过实现一个子类来扩展*EnvBuilder*。这个子类会安装 setuptotols 和 pip 的到被创建的虚拟环境中。

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.
```

(下页继续)

(续上页)

```
    :param nodist: If True, setuptools and pip are not installed into the
                   created virtual environment.
    :param nopip: If True, pip is not installed into the created
                  virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
                     installation can be monitored by passing a progress
                     callable. If specified, it is called with two
                     arguments: a string indicating some progress, and a
                     context indicating where the string is coming from.
                     The context argument can have one of three values:
                     'main', indicating that it is called from virtualize()
                     itself, and 'stdout' and 'stderr', which are obtained
                     by reading lines from the output streams of a subprocess
                     which is used to install the app.

                     If a callable is not specified, default progress
                     information is output to sys.stderr.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        Set up any packages which need to be pre-installed into the
        virtual environment being created.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # Can't install pip without setuptools
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        Read lines from a subprocess' output stream and either pass to a progress
        callable (if specified) or write progress information to sys.stderr.
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break
            if progress is not None:
                progress(s, context)
            else:
                if not self.verbose:
```

(下页继续)

(续上页)

```
                    sys.stderr.write('.')
                else:
                    sys.stderr.write(s.decode('utf-8'))
                sys.stderr.flush()
        stream.close()

    def install_script(self, context, name, url):
        _, _, path, _, _, _ = urlparse(url)
        fn = os.path.split(path)[-1]
        binpath = context.bin_path
        distpath = os.path.join(binpath, fn)
        # Download script into the virtual environment's binaries folder
        urlretrieve(url, distpath)
        progress = self.progress
        if self.verbose:
            term = '\n'
        else:
            term = ''
        if progress is not None:
            progress('Installing %s ...%s' % (name, term), 'main')
        else:
            sys.stderr.write('Installing %s ...%s' % (name, term))
            sys.stderr.flush()
        # Install in the virtual environment
        args = [context.env_exe, fn]
        p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
        t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
        t1.start()
        t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
        t2.start()
        p.wait()
        t1.join()
        t2.join()
        if progress is not None:
            progress('done.', 'main')
        else:
            sys.stderr.write('done.\n')
        # Clean up - no longer needed
        os.unlink(distpath)

    def install_setuptools(self, context):
        """
        Install setuptools in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://bitbucket.org/pypa/setuptools/downloads/ez_setup.py'
        self.install_script(context, 'setuptools', url)
        # clear up the setuptools archive which gets downloaded
        pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
        files = filter(pred, os.listdir(context.bin_path))
        for f in files:
            f = os.path.join(context.bin_path, f)
            os.unlink(f)
```

(下页继续)

(续上页)

```
    def install_pip(self, context):
        """
        Install pip in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://raw.github.com/pypa/pip/master/contrib/get-pip.py'
        self.install_script(context, 'pip', url)

def main(args=None):
    compatible = True
    if sys.version_info < (3, 3):
        compatible = False
    elif not hasattr(sys, 'base_prefix'):
        compatible = False
    if not compatible:
        raise ValueError('This script is only for use with '
                         'Python 3.3 or later')
    else:
        import argparse

        parser = argparse.ArgumentParser(prog=__name__,
                                         description='Creates virtual Python '
                                                     'environments in one or '
                                                     'more target '
                                                     'directories.')
        parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                            help='A directory in which to create the
                                 'virtual environment.')
        parser.add_argument('--no-setuptools', default=False,
                            action='store_true', dest='nodist',
                            help="Don't install setuptools or pip in the "
                                 "virtual environment.")
        parser.add_argument('--no-pip', default=False,
                            action='store_true', dest='nopip',
                            help="Don't install pip in the virtual "
                                 "environment.")
        parser.add_argument('--system-site-packages', default=False,
                            action='store_true', dest='system_site',
                            help='Give the virtual environment access to the '
                                 'system site-packages dir.')
        if os.name == 'nt':
            use_symlinks = False
        else:
            use_symlinks = True
        parser.add_argument('--symlinks', default=use_symlinks,
                            action='store_true', dest='symlinks',
                            help='Try to use symlinks rather than copies, '
                                 'when symlinks are not the default for '
                                 'the platform.')
        parser.add_argument('--clear', default=False, action='store_true',
                            dest='clear', help='Delete the contents of the '
                                               'virtual environment '
                                               'directory if it already '
                                               'exists, before virtual '
```

(下页继续)

(续上页)

```
                                            'environment creation.')
        parser.add_argument('--upgrade', default=False, action='store_true',
                            dest='upgrade', help='Upgrade the virtual '
                                                 'environment directory to '
                                                 'use this version of '
                                                 'Python, assuming Python '
                                                 'has been upgraded '
                                                 'in-place.')
        parser.add_argument('--verbose', default=False, action='store_true',
                            dest='verbose', help='Display the output '
                                            'from the scripts which '
                                            'install setuptools and pip.')
        options = parser.parse_args(args)
        if options.upgrade and options.clear:
            raise ValueError('you cannot supply --upgrade and --clear together.')
        builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                                     clear=options.clear,
                                     symlinks=options.symlinks,
                                     upgrade=options.upgrade,
                                     nodist=options.nodist,
                                     nopip=options.nopip,
                                     verbose=options.verbose)
        for d in options.dirs:
            builder.create(d)

if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```

This script is also available for download online.

28.4 zipapp —Manage executable python zip archives

3.5 新版功能.

Source code: Lib/zipapp.py

This module provides tools to manage the creation of zip files containing Python code, which can be executed directly by the Python interpreter. The module provides both a 命令行界面 and a *Python API*.

28.4.1 Basic Example

The following example shows how the 命令行界面 can be used to create an executable archive from a directory containing Python code. When run, the archive will execute the `main` function from the module `myapp` in the archive.

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

28.4.2 命令行界面

When called as a program from the command line, the following form is used:

```
$ python -m zipapp source [options]
```

If *source* is a directory, this will create an archive from the contents of *source*. If *source* is a file, it should be an archive, and it will be copied to the target archive (or the contents of its shebang line will be displayed if the –info option is specified).

The following options are understood:

-o <output>, **--output**=<output>
: Write the output to a file named *output*. If this option is not specified, the output filename will be the same as the input *source*, with the extension `.pyz` added. If an explicit filename is given, it is used as is (so a `.pyz` extension should be included if required).

  An output filename must be specified if the *source* is an archive (and in that case, *output* must not be the same as *source*).

-p <interpreter>, **--python**=<interpreter>
: Add a `#!` line to the archive specifying *interpreter* as the command to run. Also, on POSIX, make the archive executable. The default is to write no `#!` line, and not make the file executable.

-m <mainfn>, **--main**=<mainfn>
: Write a `__main__.py` file to the archive that executes *mainfn*. The *mainfn* argument should have the form "pkg.mod:fn", where "pkg.mod" is a package/module in the archive, and "fn" is a callable in the given module. The `__main__.py` file will execute that callable.

  --main cannot be specified when copying an archive.

--info
: Display the interpreter embedded in the archive, for diagnostic purposes. In this case, any other options are ignored and SOURCE must be an archive, not a directory.

-h, --help
: Print a short usage message and exit.

28.4.3 Python API

The module defines two convenience functions:

zipapp.**create_archive**(*source*, *target=None*, *interpreter=None*, *main=None*)
: Create an application archive from *source*. The source can be any of the following:

  - The name of a directory, or a *pathlib.Path* object referring to a directory, in which case a new application archive will be created from the content of that directory.

- The name of an existing application archive file, or a *pathlib.Path* object referring to such a file, in which case the file is copied to the target (modifying it to reflect the value given for the *interpreter* argument). The file name should include the `.pyz` extension, if required.
- A file object open for reading in bytes mode. The content of the file should be an application archive, and the file object is assumed to be positioned at the start of the archive.

The *target* argument determines where the resulting archive will be written:

- If it is the name of a file, or a `pathlb.Path` object, the archive will be written to that file.
- If it is an open file object, the archive will be written to that file object, which must be open for writing in bytes mode.
- If the target is omitted (or `None`), the source must be a directory and the target will be a file with the same name as the source, with a `.pyz` extension added.

The *interpreter* argument specifies the name of the Python interpreter with which the archive will be executed. It is written as a "shebang" line at the start of the archive. On POSIX, this will be interpreted by the OS, and on Windows it will be handled by the Python launcher. Omitting the *interpreter* results in no shebang line being written. If an interpreter is specified, and the target is a filename, the executable bit of the target file will be set.

The *main* argument specifies the name of a callable which will be used as the main program for the archive. It can only be specified if the source is a directory, and the source does not already contain a `__main__.py` file. The *main* argument should take the form "pkg.module:callable" and the archive will be run by importing "pkg.module" and executing the given callable with no arguments. It is an error to omit *main* if the source is a directory and does not contain a `__main__.py` file, as otherwise the resulting archive would not be executable.

If a file object is specified for *source* or *target*, it is the caller's responsibility to close it after calling create_archive.

When copying an existing archive, file objects supplied only need `read` and `readline`, or `write` methods. When creating an archive from a directory, if the target is a file object it will be passed to the `zipfile.ZipFile` class, and must supply the methods needed by that class.

`zipapp.`**`get_interpreter`**(*archive*)

Return the interpreter specified in the `#!` line at the start of the archive. If there is no `#!` line, return *None*. The *archive* argument can be a filename or a file-like object open for reading in bytes mode. It is assumed to be at the start of the archive.

28.4.4 例子

Pack up a directory into an archive, and run it.

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

The same can be done using the *create_archive()* functon:

```
>>> import zipapp
>>> zipapp.create_archive('myapp.pyz', 'myapp')
```

To make the application directly executable on POSIX, specify an interpreter to use.

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

To replace the shebang line on an existing archive, create a modified archive using the *create_archive()* function:

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

To update the file in place, do the replacement in memory using a `BytesIO` object, and then overwrite the source afterwards. Note that there is a risk when overwriting a file in place that an error will result in the loss of the original file. This code does not protect against such errors, but production code should do so. Also, this method will only work if the archive fits in memory:

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

28.4.5 Specifying the Interpreter

Note that if you specify an interpreter and then distribute your application archive, you need to ensure that the interpreter used is portable. The Python launcher for Windows supports most common forms of POSIX `#!` line, but there are other issues to consider:

- If you use "/usr/bin/env python" (or other forms of the "python" command, such as "/usr/bin/python"), you need to consider that your users may have either Python 2 or Python 3 as their default, and write your code to work under both versions.
- If you use an explicit version, for example "/usr/bin/env python3" your application will not work for users who do not have that version. (This may be what you want if you have not made your code Python 2 compatible).
- There is no way to say "python X.Y or later", so be careful of using an exact version like "/usr/bin/env python3.4" as you will need to change your shebang line for users of Python 3.5, for example.

Typically, you should use an "/usr/bin/env python2" or "/usr/bin/env python3", depending on whether your code is written for Python 2 or 3.

28.4.6 Creating Standalone Applications with zipapp

Using the *zipapp* module, it is possible to create self-contained Python programs, which can be distributed to end users who only need to have a suitable version of Python installed on their system. The key to doing this is to bundle all of the application's dependencies into the archive, along with the application code.

The steps to create a standalone archive are as follows:

1. Create your application in a directory as normal, so you have a `myapp` directory containing a `__main__.py` file, and any supporting application code.
2. Install all of your application's dependencies into the `myapp` directory, using pip:

   ```
   $ python -m pip install -r requirements.txt --target myapp
   ```

   (this assumes you have your project requirements in a `requirements.txt` file - if not, you can just list the dependencies manually on the pip command line).
3. Optionally, delete the `.dist-info` directories created by pip in the `myapp` directory. These hold metadata for pip to manage the packages, and as you won't be making any further use of pip they aren't required - although it won't do any harm if you leave them.
4. Package the application using:

```
$ python -m zipapp -p "interpreter" myapp
```

This will produce a standalone executable, which can be run on any machine with the appropriate interpreter available. See *Specifying the Interpreter* for details. It can be shipped to users as a single file.

On Unix, the `myapp.pyz` file is executable as it stands. You can rename the file to remove the `.pyz` extension if you prefer a "plain" command name. On Windows, the `myapp.pyz[w]` file is executable by virtue of the fact that the Python interpreter registers the `.pyz` and `.pyzw` file extensions when installed.

Making a Windows executable

On Windows, registration of the `.pyz` extension is optional, and furthermore, there are certain places that don't recognise registered extensions "transparently" (the simplest example is that `subprocess.run(['myapp'])` won't find your application - you need to explicitly specify the extension).

On Windows, therefore, it is often preferable to create an executable from the zipapp. This is relatively easy, although it does require a C compiler. The basic approach relies on the fact that zipfiles can have arbitrary data prepended, and Windows exe files can have arbitrary data appended. So by creating a suitable launcher and tacking the `.pyz` file onto the end of it, you end up with a single-file executable that runs your application.

A suitable launcher can be as simple as the following:

```
#define Py_LIMITED_API 1
#include "Python.h"

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#ifdef WINDOWS
int WINAPI wWinMain(
    HINSTANCE hInstance,      /* handle to current instance */
    HINSTANCE hPrevInstance,  /* handle to previous instance */
    LPWSTR lpCmdLine,         /* pointer to command line */
    int nCmdShow              /* show state of window */
)
#else
int wmain()
#endif
{
    wchar_t **myargv = _alloca((__argc + 1) * sizeof(wchar_t*));
    myargv[0] = __wargv[0];
    memcpy(myargv + 1, __wargv, __argc * sizeof(wchar_t *));
    return Py_Main(__argc+1, myargv);
}
```

If you define the `WINDOWS` preprocessor symbol, this will generate a GUI executable, and without it, a console executable.

To compile the executable, you can either just use the standard MSVC command line tools, or you can take advantage of the fact that distutils knows how to compile Python source:

```
>>> from distutils.ccompiler import new_compiler
>>> import distutils.sysconfig
>>> import sys
>>> import os
>>> from pathlib import Path

>>> def compile(src):
```

(下页继续)

(续上页)

```
>>>     src = Path(src)
>>>     cc = new_compiler()
>>>     exe = src.stem
>>>     cc.add_include_dir(distutils.sysconfig.get_python_inc())
>>>     cc.add_library_dir(os.path.join(sys.base_exec_prefix, 'libs'))
>>>     # First the CLI executable
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe)
>>>     # Now the GUI executable
>>>     cc.define_macro('WINDOWS')
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe + 'w')

>>> if __name__ == "__main__":
>>>     compile("zastub.c")
```

The resulting launcher uses the "Limited ABI", so it will run unchanged with any version of Python 3.x. All it needs is for Python (`python3.dll`) to be on the user's `PATH`.

For a fully standalone distribution, you can distribute the launcher with your application appended, bundled with the Python "embedded" distribution. This will run on any PC with the appropriate architecture (32 bit or 64 bit).

Caveats

There are some limitations to the process of bundling your application into a single file. In most, if not all, cases they can be addressed without needing major changes to your application.

1. If your application depends on a package that includes a C extension, that package cannot be run from a zip file (this is an OS limitation, as executable code must be present in the filesystem for the OS loader to load it). In this case, you can exclude that dependency from the zipfile, and either require your users to have it installed, or ship it alongside your zipfile and add code to your `__main__.py` to include the directory containing the unzipped module in `sys.path`. In this case, you will need to make sure to ship appropriate binaries for your target architecture(s) (and potentially pick the correct version to add to `sys.path` at runtime, based on the user's machine).
2. If you are shipping a Windows executable as described above, you either need to ensure that your users have `python3.dll` on their PATH (which is not the default behaviour of the installer) or you should bundle your application with the embedded distribution.
3. The suggested launcher above uses the Python embedding API. This means that in your application, `sys.executable` will be your application, and *not* a conventional Python interpreter. Your code and its dependencies need to be prepared for this possibility. For example, if your application uses the *`multiprocessing`* module, it will need to call *`multiprocessing.set_executable()`* to let the module know where to find the standard Python interpreter.

28.4.7 The Python Zip Application Archive Format

Python has been able to execute zip files which contain a `__main__.py` file since version 2.6. In order to be executed by Python, an application archive simply has to be a standard zip file containing a `__main__.py` file which will be run as the entry point for the application. As usual for any Python script, the parent of the script (in this case the zip file) will be placed on *`sys.path`* and thus further modules can be imported from the zip file.

The zip file format allows arbitrary data to be prepended to a zip file. The zip application format uses this ability to prepend a standard POSIX "shebang" line to the file (`#!/path/to/interpreter`).

Formally, the Python zip application format is therefore:

1. An optional shebang line, containing the characters `b'#!'` followed by an interpreter name, and then a newline (`b'\n'`) character. The interpreter name can be anything acceptable to the OS "shebang" processing, or the Python launcher on Windows. The interpreter should be encoded in UTF-8 on Windows, and in *sys.getfilesystemencoding()* on POSIX.
2. Standard zipfile data, as generated by the *zipfile* module. The zipfile content *must* include a file called `__main__.py` (which must be in the "root" of the zipfile - i.e., it cannot be in a subdirectory). The zipfile data can be compressed or uncompressed.

If an application archive has a shebang line, it may have the executable bit set on POSIX systems, to allow it to be executed directly.

There is no requirement that the tools in this module are used to create application archives - the module is a convenience, but archives in the above format created by any means are acceptable to Python.

CHAPTER 29

Python 运行时服务

本章里描述的模块提供了和 Python 解释器及其环境交互相关的广泛服务。以下是综述：

29.1 sys —系统相关的参数和函数

该模块提供了一些变量和函数。这些变量可能被解释器使用，也可能由解释器提供。这些函数会影响解释器。本模块总是可用的。

sys.**abiflags**
在 POSIX 系统上，以标准的 configure 脚本构建的 Python 中，这个变量会包含 **PEP 3149** 中定义的 ABI 标签。

3.2 新版功能.

sys.**argv**
一个列表，其中包含了被传递给 Python 脚本的命令行参数。argv[0] 为脚本的名称（是否是完整的路径名取决于操作系统）。如果是通过 Python 解释器的命令行参数 -c 来执行的，argv[0] 会被设置成字符串 '-c' 。如果没有脚本名被传递给 Python 解释器，argv[0] 为空字符串。

为了遍历标准输入，或者通过命令行传递的文件列表，参照*fileinput* 模块

sys.**base_exec_prefix**
在 site.py 运行之前，Python 启动的时候被设置为跟*exec_prefix* 同样的值。如果不是运行在虚拟环境 中，两个值会保持相同；如果 site.py 发现处于一个虚拟环境中，*prefix* 和*exec_prefix* 将会指向虚拟环境。然而*base_prefix* 和*base_exec_prefix* 将仍然会指向基础的 Python 环境（用来创建虚拟环境的 Python 环境）

3.3 新版功能.

sys.**base_prefix**
在 site.py 运行之前，Python 启动的时候被设置为跟*prefix* 同样的值。如果不是运行在虚拟环境 中, 两个值会保持相同；如果 site.py 发现处于一个虚拟环境中，*prefix* 和*exec_prefix* 将会指

向虚拟环境。然而*base_prefix* 和*base_exec_prefix* 将仍然会指向基础的 Python 环境（用来创建虚拟环境的 Python 环境）

3.3 新版功能.

sys.**byteorder**
本地字节顺序的指示符。在大端序（最高有效位优先）操作系统上值为 'big'，在小端序（最低有效位优先）操作系统上为 'little'。

sys.**builtin_module_names**
一个元素为字符串的元组。包含了所有的被编译进 Python 解释器的模块。（这个信息无法通过其他的办法获取，modules.keys() 只包括被导入过的模块。）

sys.**call_tracing**(*func*, *args*)
在启用跟踪时调用 func(*args) 来保存跟踪状态，然后恢复跟踪状态。这将从检查点的调试器调用，以便递归地调试其他的一些代码。

sys.**copyright**
一个字符串，包含了 Python 解释器有关的版权信息

sys.**_clear_type_cache**()
清除内部的类型缓存。类型缓存是为了加速查找方法和属性的。在调试引用泄漏的时候调用这个函数只会清除不必要的引用。

这个函数应该只在内部为了一些特定的目的使用。

sys.**_current_frames**()
返回一个字典，将每个线程的标识符映射到调用函数时该线程中当前活动的最顶层堆栈帧。注意*traceback* 模块中的函数可以在给定帧的情况下构建调用堆栈。

这对于调试死锁最有用：本函数不需要死锁线程的配合，并且只要这些线程的调用栈保持死锁，它们就是冻结的。在调用本代码来检查栈顶的帧的那一刻，非死锁线程返回的帧可能与该线程当前活动的帧没有任何关系。

这个函数应该只在内部为了一些特定的目的使用。

sys.**_debugmallocstats**()
将有关 CPython 内存分配器状态的底层的信息打印至 stderr。

如果 Python 被配置为–with-pydebug，本方法还将执行一些开销较大的内部一致性检查。

3.3 新版功能.

CPython implementation detail: 本函数仅限 CPython。此处没有定义确切的输出格式，且可能会更改。

sys.**dllhandle**
Integer specifying the handle of the Python DLL. Availability: Windows.

sys.**displayhook**(*value*)
如果 *value* 不是 None，则本函数会将 repr(value) 打印至 sys.stdout，并将 *value* 保存在 builtins._ 中。如果 repr(value) 无法用 sys.stdout.errors 错误回调方法（可能较为“严格”）编码为 sys.stdout.encoding，请用 'backslashreplace' 错误回调方法将其编码为 sys.stdout.encoding。

在交互式 Python 会话中运行*expression* 产生结果后，将在结果上调用 sys.displayhook。若要自定义这些 value 的显示，可以将 sys.displayhook 指定为另一个单参数函数。

伪代码:

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
```

(下页继续)

(续上页)

```
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

在 3.2 版更改: 在发生*UnicodeEncodeError* 时使用 'backslashreplace' 错误回调方法。

sys.**dont_write_bytecode**
如果该值为 true，则 Python 在导入源码模块时将不会尝试写入 .pyc 文件。该值会被初始化为 True 或 False，依据是 -B 命令行选项和 PYTHONDONTWRITEBYTECODE 环境变量，可以自行设置该值，来控制是否生成字节码文件。

sys.**excepthook**(*type*, *value*, *traceback*)
本函数会将所给的回溯和异常输出到 sys.stderr 中。

当抛出一个异常，且未被捕获时，解释器将调用 sys.excepthook 并带有三个参数：异常类、异常实例和一个回溯对象。在交互式会话中，这会在控制权返回到提示符之前发生。在 Python 程序中，这会在程序退出之前发生。如果要自定义此类顶级异常的处理过程，可以将另一个 3 个参数的函数赋给 sys.excepthook。

sys.**__displayhook__**
sys.**__excepthook__**
These objects contain the original values of displayhook and excepthook at the start of the program. They are saved so that displayhook and excepthook can be restored in case they happen to get replaced with broken objects.

sys.**exc_info**()
本函数返回的元组包含三个值，它们给出当前正在处理的异常的信息。返回的信息仅限于当前线程和当前堆栈帧。如果当前堆栈帧没有正在处理的异常，则信息将从调用的下级堆栈帧或上级调用者等位置获取，依此类推，直到找到正在处理异常的堆栈帧为止。此处的“处理异常”被定义为“执行 except 子句”。任何堆栈帧都只能访问当前正在处理的异常的信息。

If no exception is being handled anywhere on the stack, a tuple containing three None values is returned. Otherwise, the values returned are (type, value, traceback). Their meaning is: *type* gets the type of the exception being handled (a subclass of *BaseException*); *value* gets the exception instance (an instance of the exception type); *traceback* gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

sys.**exec_prefix**
一个字符串，给出特定域的目录前缀，该目录中安装了与平台相关的 Python 文件，默认也是 '/usr/local'。该目录前缀可以在构建时使用 **configure** 脚本的 --exec-prefix 参数进行设置。具体而言，所有配置文件（如 pyconfig.h 头文件）都安装在目录 *exec_prefix*/lib/python*X.Y*/config 中，共享库模块安装在 *exec_prefix*/lib/python*X.Y*/lib-dynload 中，其中 *X.Y* 是 Python 的版本号，如 3.2。

注解: 如果在一个虚拟环境 中，那么该值将在 site.py 中被修改，指向虚拟环境。Python 安装位置仍然可以用*base_exec_prefix* 来获取。

---

sys.**executable**
一个字符串，提供 Python 解释器的可执行二进制文件的绝对路径，仅在部分系统中此值有意义。如果 Python 无法获取其可执行文件的真实路径，则*sys.executable* 将为空字符串或 None。

sys.**exit**([*arg*])
从 Python 中退出。实现方式是抛出一个*SystemExit* 异常。异常抛出后 try 声明的 finally 分支语句的清除动作将被出发。此动作有可能打断更外层的退出尝试。

可选参数 *arg* 可以是表示退出状态的整数（默认为 0），也可以是其他类型的对象。如果它是整数，则 shell 等将 0 视为“成功终止”，非零值视为“异常终止”。大多数系统要求该值的范围是 0–127，否则会产生不确定的结果。某些系统为退出代码约定了特定的含义，但通常尚不完善；Unix 程序通常用 2 表示命令行语法错误，用 1 表示所有其他类型的错误。传入其他类型的对象，如果传入 None 等同于传入 0，如果传入其他对象则将其打印至*stderr*，且退出代码为 1。特别地，sys.exit("some error message") 可以在发生错误时快速退出程序。

由于*exit()* 最终“只是”抛出一个异常，因此当从主线程调用时，只会从进程退出；而异常不会因此被打断。

在 3.6 版更改: 在 Python 解释器捕获*SystemExit* 后，如果在清理中发生错误（如清除标准流中的缓冲数据时出错），则退出状态码将变为 120。

sys.**flags**
The *struct sequence* *flags* exposes the status of command line flags. The attributes are read only.

attribute –属性	标志
debug	-d
inspect	-i
interactive	-i
isolated	-I
optimize	-O 或 -OO
dont_write_bytecode	-B
no_user_site	-s
no_site	-S
ignore_environment	-E
verbose	-v
bytes_warning	-b
quiet	-q
hash_randomization	-R

在 3.2 版更改: 为新的 -q 标志添加了 quiet 属性

3.2.3 新版功能: hash_randomization 属性

在 3.3 版更改: 删除了过时的 division_warning 属性

在 3.4 版更改: 为 -I isolated 标志添加了 isolated 属性。

sys.**float_info**
A *struct sequence* holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file float.h for the ‘C’ programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [C99], ‘Characteristics of floating types’, for details.

attribute –属性	float.h 宏	解释
epsilon	DBL_EPSILON	difference between 1 and the least value greater than 1 that is representable as a float
dig	DBL_DIG	浮点数可以真实表示的最大十进制数字；见下文
mant_dig	DBL_MANT_DIG	浮点数精度：radix 基数下的浮点数有效位数
max	DBL_MAX	maximum representable finite float
max_exp	DBL_MAX_EXP	maximum integer e such that radix**(e-1) is a representable finite float
max_10_exp	DBL_MAX_10_EXP	maximum integer e such that 10**e is in the range of representable finite floats
min	DBL_MIN	minimum positive normalized float
min_exp	DBL_MIN_EXP	minimum integer e such that radix**(e-1) is a normalized float
min_10_exp	DBL_MIN_10_EXP	minimum integer e such that 10**e is a normalized float
radix	FLT_RADIX	指数表示法中采用的基数
rounds	FLT_ROUNDS	整数常数，表示算术运算中的舍入方式。它反映了解释器启动时系统的 FLT_ROUNDS 宏的值。关于可能的值及其含义的说明，请参阅 C99 标准 5.2.4.2.2 节。

关于 sys.float_info.dig 属性的进一步说明。如果 s 是表示十进制数的字符串，而该数最多有 sys.float_info.dig 位有效数字，则将 s 转换为 float 再转回去将恢复原先相同十进制值的字符串:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'    # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'
```

但是对于超过 sys.float_info.dig 位有效数字的字符串，转换前后并非总是相同:

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

sys.**float_repr_style**

一个字符串，反映*repr()* 函数在浮点数上的行为。如果该字符串是 'short'，那么对于（非无穷的）浮点数 x，repr(x) 将会生成一个短字符串，满足 float(repr(x)) == x 的特性。这是 Python 3.1 及更高版本中的常见行为。否则 float_repr_style 的值将是 'legacy'，此时 repr(x) 的行为方式将与 Python 3.1 之前的版本相同。

3.1 新版功能.

sys.**getallocatedblocks**()

返回解释器当前已分配的内存块数，无论它们大小如何。本函数主要用于跟踪和调试内存泄漏。因为解释器有内部缓存，所以不同调用之间结果会变化。可能需要调用*_clear_type_cache()* 和*gc.collect()* 使结果更容易预测。

如果当前 Python 构建或实现无法合理地计算此信息，允许*getallocatedblocks()* 返回 0。

3.4 新版功能.

sys.**getcheckinterval**()

Return the interpreter’s “check interval”; see *setcheckinterval()*.

3.2 版后已移除: Use *getswitchinterval()* instead.

sys.**getdefaultencoding**()
返回当前 Unicode 实现所使用的默认字符串编码名称。

sys.**getdlopenflags**()
Return the current value of the flags that are used for `dlopen()` calls. Symbolic names for the flag values can be found in the *os* module (`RTLD_xxx` constants, e.g. *os.RTLD_LAZY*). Availability: Unix.

sys.**getfilesystemencoding**()
返回编码名称，该编码用于在 Unicode 文件名和 bytes 文件名之间转换。为获得最佳兼容性，任何时候都应使用 str 表示文件名，尽管用字节来表示文件名也是支持的。函数如果需要接受或返回文件名，它应支持 str 或 bytes，并在内部将其转换为系统首选的表示形式。

该编码始终是 ASCII 兼容的。

应使用*os.fsencode()* 和*os.fsdecode()* 来保证所采用的编码和错误模式都是正确的。

- On Mac OS X, the encoding is `'utf-8'`.
- 在 Unix 上，编码是语言环境编码。
- 在 Windows 上取决于用户配置，编码可能是 `'utf-8'` 或 `'mbcs'`。

在 3.2 版更改: *getfilesystemencoding()* 的结果将不再有可能是 `None`。

在 3.6 版更改: Windows 不再保证会返回 `'mbcs'`。详情请参阅 **PEP 529** 和*_enablelegacywindowsfsencoding()*。

sys.**getfilesystemencodeerrors**()
返回错误回调函数的名称，该错误回调函数将在 Unicode 文件名和 bytes 文件名转换时生效。编码的名称是由*getfilesystemencoding()* 返回的。

应使用*os.fsencode()* 和*os.fsdecode()* 来保证所采用的编码和错误模式都是正确的。

3.6 新版功能.

sys.**getrefcount**(*object*)
返回 *object* 的引用计数。返回的计数通常比预期的多一，因为它包括了作为*getrefcount()* 参数的这一次（临时）引用。

sys.**getrecursionlimit**()
返回当前的递归限制值，即 Python 解释器堆栈的最大深度。此限制可防止无限递归导致的 C 堆栈溢出和 Python 崩溃。该值可以通过*setrecursionlimit()* 设置。

sys.**getsizeof**(*object*[, *default*])
返回对象的大小（以字节为单位）。该对象可以是任何类型。所有内建对象返回的结果都是正确的，但对于第三方扩展不一定正确，因为这与具体实现有关。

只计算直接分配给对象的内存消耗，不计算它所引用的对象的内存消耗。

对象不提供计算大小的方法时，如果有给出 *default* 则返回它，否则抛出*TypeError* 异常。

如果对象由垃圾回收器管理，则*getsizeof()* 将调用对象的 `__sizeof__` 方法，并在上层添加额外的垃圾回收器。

可以参考 recursive sizeof recipe 中的示例，关于递归调用*getsizeof()* 来得到各个容器及其所有内容物的大小。

sys.**getswitchinterval**()
返回解释器的“线程切换间隔时间”，请参阅*setswitchinterval()*。

3.2 新版功能.

sys.**_getframe**([*depth*])
返回来自调用栈的一个帧对象。如果传入可选整数 *depth*，则返回从栈顶往下相应调用层数的帧对象。如果该数比调用栈更深，则抛出*ValueError*。*depth* 的默认值是 0，返回调用栈顶部的帧。

CPython implementation detail: 这个函数应该只在内部为了一些特定的目的使用。不保证它在所有 Python 实现中都存在。

sys.**getprofile**()
返回由*setprofile()* 设置的性能分析函数

sys.**gettrace**()
返回由*settrace()* 设置的跟踪函数。

CPython implementation detail: *gettrace()* 函数仅用于实现调试器，性能分析器，打包工具等。它的行为是实现平台的一部分，而不是语言定义的一部分，因此并非在所有 Python 实现中都可用。

sys.**getwindowsversion**()
返回一个具名元组，描述当前正在运行的 Windows 版本。元素名称包括 *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* 和 *platform_version*。*service_pack* 包含一个字符串，*platform_version* 包含一个三元组，其他所有值都是整数。元素也可以通过名称来访问，所以 `sys.getwindowsversion()[0]` 与 `sys.getwindowsversion().major` 是等效的。为保持与旧版本的兼容性，只有前 5 个元素可以用索引检索。

platform 将会是 `2 (VER_PLATFORM_WIN32_NT)`。

product_type 可能是以下值之一：

常数	含义
`1 (VER_NT_WORKSTATION)`	系统是工作站。
`2 (VER_NT_DOMAIN_CONTROLLER)`	系统是域控制器。
`3 (VER_NT_SERVER)`	系统是服务器，但不是域控制器。

本函数包装了 Win32 `GetVersionEx()` 函数，参阅 Microsoft 文档有关 `OSVERSIONINFOEX()` 的内容可获取这些字段的更多信息。

platform_version 返回的是当前操作系统真实准确的主要版本、次要版本和内部版本号，不是为该进程模拟的版本。它旨在用于记录日志，不用于检测功能。

Availability: Windows.

在 3.2 版更改: 更改为具名元组，添加 *service_pack_minor*, *service_pack_major*, *suite_mask* 和 *product_type*。

在 3.6 版更改: 添加了 *platform_version*

sys.**get_asyncgen_hooks**()
Returns an *asyncgen_hooks* object, which is similar to a *namedtuple* of the form *(firstiter, finalizer)*, where *firstiter* and *finalizer* are expected to be either `None` or functions which take an *asynchronous generator iterator* as an argument, and are used to schedule finalization of an asychronous generator by an event loop.

3.6 新版功能: 详情请参阅 **PEP 525**。

注解: 本函数已添加至暂定软件包（详情请参阅 **PEP 411** ）。

---

sys.**get_coroutine_wrapper**()
Returns `None`, or a wrapper set by *set_coroutine_wrapper()*.

3.5 新版功能: See **PEP 492** for more details.

注解: 本函数已添加至暂定软件包（详情请参阅 **PEP 411** ）。仅将其用于调试目的。

---

sys.**hash_info**

A *struct sequence* giving parameters of the numeric hash implementation. For more details about hashing of numeric types, see 数字类型的哈希运算.

attribute –属性	解释
width	用于哈希值的位宽度
modulus	用于数字散列方案的素数模数 P。
inf	为正无穷大返回的哈希值
nan	为 nan 返回的哈希值
imag	用于复数虚部的乘数
algorithm	字符串、字节和内存视图的哈希算法的名称
hash_bits	哈希算法的内部输出大小。
seed_bits	散列算法的种子密钥的大小

3.2 新版功能.

在 3.4 版更改: 添加了 *algorithm*, *hash_bits* 和 *seed_bits*

sys.**hexversion**

编码为单个整数的版本号。该整数会确保每个版本都自增，其中适当包括了未发布版本。举例来说，要测试 Python 解释器的版本不低于 1.5.2，请使用:

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

This is called `hexversion` since it only really looks meaningful when viewed as the result of passing it to the built-in *`hex()`* function. The *struct sequence* *`sys.version_info`* may be used for a more human-friendly encoding of the same information.

关于 `hexversion` 的更多信息可以在 apiabiversion 中找到。

sys.**implementation**

一个对象，该对象包含当前运行的 Python 解释器的实现信息。所有 Python 实现中都必须存在下列属性。

name 是当前实现的标识符，如 `'cpython'`。实际的字符串由 Python 实现定义，但保证是小写字母。

version 是一个具名元组，格式与*`sys.version_info`* 相同。它表示 Python 实现的版本。另一个（由 `sys.version_info` 表示）是当前解释器遵循的相应 Python 语言的版本，两者具有不同的含义。例如，对于 PyPy 1.8，`sys.implementation.version` 可能是 `sys.version_info(1, 8, 0, 'final', 0)`，而 `sys.version_info` 则是 `sys.version_info(2, 7, 2, 'final', 0)`。对于 CPython 而言两个值是相同的，因为它是参考实现。

hexversion 是十六进制的实现版本，类似于*`sys.hexversion`*。

cache_tag 是导入机制使用的标记，用于已缓存模块的文件名。按照惯例，它将由实现的名称和版本组成，如 `'cpython-33'`。但如果合适，Python 实现可以使用其他值。如果 `cache_tag` 被置为 `None`，表示模块缓存已禁用。

`sys.implementation` 可能包含相应 Python 实现的其他属性。这些非标准属性必须以下划线开头，此处不详细阐述。无论其内容如何，*`sys.implementation`* 在解释器运行期间或不同实现版本之间都不会更改。（但是不同 Python 语言版本间可能会不同。）详情请参阅 **PEP 421**。

3.3 新版功能.

sys.**int_info**
A *struct sequence* that holds information about Python's internal representation of integers. The attributes are read only.

属性	解释
bits_per_digit	每个数字占有的位数。Python 内部将整数存储在基底 2**int_info.bits_per_digit
sizeof_digit	用于表示数字的 C 类型的字节大小

3.1 新版功能.

sys.**__interactivehook__**
当本属性存在，则以 交互模式启动解释器时，将自动（不带参数地）调用本属性的值。该过程是在读取 PYTHONSTARTUP 文件之后完成的，所以可以在该文件中设置这一钩子。*site* 模块设置了这一属性。

3.4 新版功能.

sys.**intern**(*string*)
将 *string* 插入 "interned"（驻留）字符串表，返回被插入的字符串–它是 *string* 本身或副本。驻留字符串对提高字典查找的性能很有用–如果字典中的键已驻留，且所查找的键也已驻留，则键（取散列后）的比较可以用指针代替字符串来比较。通常，Python 程序使用到的名称会被自动驻留，且用于保存模块、类或实例属性的字典的键也已驻留。

驻留字符串不是永久存在的，对*intern()* 返回值的引用必须保留下来，才能发挥驻留字符串的优势。

sys.**is_finalizing**()
如果 Python 解释器正在关闭，则返回*True*，否则返回*False*。

3.5 新版功能.

sys.**last_type**
sys.**last_value**
sys.**last_traceback**
这三个变量并非总是有定义，仅当有异常未处理，且解释器打印了错误消息和堆栈回溯时，才会给它们赋值。它们的预期用途，是允许交互中的用户导入调试器模块，进行事后调试，而不必重新运行导致错误的命令。（通常使用 import pdb; pdb.pm() 进入事后调试器，详情请参阅*pdb* 模块。）

这些变量的含义与上述*exc_info()* 返回值的含义相同。

sys.**maxsize**
一个整数，表示 Py_ssize_t 类型的变量可以取到的最大值。在 32 位平台上通常为 2**31 - 1，在 64 位平台上通常为 2**63 - 1。

sys.**maxunicode**
一个整数，表示最大的 Unicode 码点值，如 1114111 （十六进制为 0x10FFFF ）。

在 3.3 版更改: 在 **PEP 393** 之前，sys.maxunicode 曾是 0xFFFF 或 0x10FFFF，具体取决于配置选项，该选项指定将 Unicode 字符存储为 UCS-2 还是 UCS-4。

sys.**meta_path**
一个由元路径查找器 对象组成的列表，当查找需要导入的模块时，会调用这些对象的*find_spec()* 方法，观察这些对象是否能找到所需模块。调用*find_spec()* 方法最少需要传入待导入模块的绝对名称。如果待导入模块包含在一个包中，则父包的 __path__ 属性将作为第二个参数被传入。该方法返回模块规格，找不到模块则返回 None。

参见:

importlib.abc.MetaPathFinder 抽象基类，定义了*meta_path* 内的查找器对象的接口。

importlib.machinery.ModuleSpec *find_spec()* 返回的实例所对应的具体类。

在 3.4 版更改: 在 Python 3.4 中通过 **PEP 451** 引入了模块规格。早期版本的 Python 会寻找一个称为 *find_module()* 的方法。如果某个 *meta_path* 条目没有 *find_spec()* 方法，就会回退去调用前一种方法。

sys.**modules**
一个字典，将模块名称映射到已加载的模块。可以操作该字典来强制重新加载模块，或是实现其他技巧。但是，替换的字典不一定会按预期工作，并且从字典中删除必要的项目可能会导致 Python 崩溃。

sys.**path**
一个由字符串组成的列表，用于指定模块的搜索路径。初始化自环境变量 PYTHONPATH，再加上一条与安装有关的默认路径。

程序启动时将初始化该列表，列表的第一项 path[0] 目录含有调用 Python 解释器的脚本。如果脚本目录不可用（比如以交互方式调用了解释器，或脚本是从标准输入中读取的），则 path[0] 为空字符串，这将导致 Python 优先搜索当前目录中的模块。注意，脚本目录将插入在 PYTHONPATH 的条目 * 之前 *。

程序可以随意修改本列表用于自己的目的。只能向 *sys.path* 中添加 string 和 bytes 类型，其他数据类型将在导入期间被忽略。

参见:

site 模块，该模块描述了如何使用.pth 文件来扩展 *sys.path*。

sys.**path_hooks**
一个由可调用对象组成的列表，这些对象接受一个路径作为参数，并尝试为该路径创建一个查找器。如果成功创建查找器，则可调用对象将返回它，否则将引发 *ImportError* 异常。

本特性最早在 **PEP 302** 中被提及。

sys.**path_importer_cache**
一个字典，作为查找器 对象的缓存。key 是传入 *sys.path_hooks* 的路径，value 是相应已找到的查找器。如果路径是有效的文件系统路径，但在 *sys.path_hooks* 中未找到查找器，则存入 None。

本特性最早在 **PEP 302** 中被提及。

在 3.3 版更改: 未找到查找器时，改为存储 None，而不是 *imp.NullImporter*。

sys.**platform**
本字符串是一个平台标识符，举例而言，该标识符可用于将特定平台的组件追加到 *sys.path* 中。

For Unix systems, except on Linux, this is the lowercased OS name as returned by uname -s with the first part of the version as returned by uname -r appended, e.g. 'sunos5' or 'freebsd8', *at the time when Python was built*. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:

```
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
```

对于其他系统，值是:

系统	平台值
Linux	'linux'
Windows	'win32'
Windows/Cygwin	'cygwin'
Mac OS X	'darwin'

在 3.3 版更改: 在 Linux 上，*sys.platform* 将不再包含副版本号。它将总是 'linux' 而不是

'linux2' 或 'linux3'。由于旧版本的 Python 会包含该版本号，因此推荐总是使用上述 startswith 习惯用法。

参见:

os.name 更加简略。*os.uname()* 提供系统的版本信息。

platform 模块对系统的标识有更详细的检查。

sys.**prefix**
一个字符串，给出特定域的目录前缀，该目录中安装了与平台不相关的 Python 文件，默认为 '/usr/local'。该目录前缀可以在构建时使用 **configure** 脚本的 --prefix 参数进行设置。Python 库模块的的主要集合安装在目录 *prefix*/lib/python*X.Y* ，而与平台无关的头文件 (除了 pyconfig.h) 保存在 *prefix*/include/python*X.Y*, 其中 *X.Y* 是 Python 的版本号, 例如 3.2.

注解: 如果在一个虚拟环境 中，那么该值将在 site.py 中被修改，指向虚拟环境。Python 安装位置仍然可以用*base_prefix* 来获取。

---

sys.**ps1**
sys.**ps2**
字符串，指定解释器的首要和次要提示符。仅当解释器处于交互模式时，它们才有定义。这种情况下，它们的初值为 '>>> ' 和 '... '。如果赋给其中某个变量的是非字符串对象，则每次解释器准备读取新的交互式命令时，都会重新运行该对象的*str()*，这可以用来实现动态的提示符。

sys.**setcheckinterval**(*interval*)
Set the interpreter's "check interval". This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is 100, meaning the check is performed every 100 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value <= 0 checks every virtual instruction, maximizing responsiveness as well as overhead.

3.2 版后已移除: This function doesn't have an effect anymore, as the internal logic for thread switching and asynchronous tasks has been rewritten. Use *setswitchinterval()* instead.

sys.**setdlopenflags**(*n*)
设置解释器在调用 dlopen() 时用到的标志，例如解释器在加载扩展模块时。首先，调用 sys.setdlopenflags(0) 将在导入模块时对符号启用惰性解析。要在扩展模块之间共享符号，请调用 sys.setdlopenflags(os.RTLD_GLOBAL)。标志值的符号名称可以在*os* 模块中找到（即 RTLD_xxx 常数，如*os.RTLD_LAZY* ）。

Availability: Unix.

sys.**setprofile**(*profilefunc*)
Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter *Python 分析器* for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see *settrace()*), but it is called with different events, for example it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return None.

性能分析函数应接收三个参数：*frame*、*event* 和 *arg*。*frame* 是当前的堆栈帧。*event* 是一个字符串：'call'、'return'、'c_call'、'c_return' 或 'c_exception'。*arg* 取决于事件类型。

这些事件具有以下含义：

'call' 表示调用了某个函数（或进入了其他的代码块）。性能分析函数将被调用，*arg* 为 None。

'return' 表示某个函数（或别的代码块）即将返回。性能分析函数将被调用，*arg* 是即将返回的值，如果此次事件是由抛出的异常引起的，*arg* 为 None。

'c_call' 表示即将调用某个 C 函数。它可能是扩展函数或是内建函数。*arg* 是 C 函数对象。

'c_return' 表示返回了某个 C 函数。*arg* 是 C 函数对象。

'c_exception' 表示某个 C 函数抛出了异常。*arg* 是 C 函数对象。

sys.**setrecursionlimit**(*limit*)
将 Python 解释器堆栈的最大深度设置为 *limit*。此限制可防止无限递归导致的 C 堆栈溢出和 Python 崩溃。

不同平台所允许的最高限值不同。当用户有需要深度递归的程序且平台支持更高的限值，可能就需要调高限值。进行该操作需要谨慎，因为过高的限值可能会导致崩溃。

如果新的限值低于当前的递归深度，将抛出*RecursionError* 异常。

在 3.5.1 版更改: 如果新的限值低于当前的递归深度，现在将抛出*RecursionError* 异常。

sys.**setswitchinterval**(*interval*)
设置解释器的线程切换间隔时间（单位为秒）。该浮点数决定了“时间片”的理想持续时间，时间片将分配给同时运行的 Python 线程。请注意，实际值可能更高，尤其是使用了运行时间长的内部函数或方法时。同时，在时间间隔末尾调度哪个线程是操作系统的决定。解释器没有自己的调度程序。

3.2 新版功能.

sys.**settrace**(*tracefunc*)
Set the system’s trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must be registered using *settrace()* for each thread being debugged.

Trace functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'line'`, `'return'` or `'exception'`. *arg* depends on the event type.

The trace function is invoked (with *event* set to `'call'`) whenever a new local scope is entered; it should return a reference to a local trace function to be used that scope, or `None` if the scope shouldn’t be traced.

本地跟踪函数应返回对自身的引用（或对另一个函数的引用，用来在其作用范围内进行进一步的跟踪），或者返回 `None` 来停止跟踪其作用范围。

这些事件具有以下含义：

'call' 表示调用了某个函数（或进入了其他的代码块）。全局跟踪函数将被调用，*arg* 为 `None`。返回值取决于本地跟踪函数。

'line' The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; *arg* is `None`; the return value specifies the new local trace function. See `Objects/lnotab_notes.txt` for a detailed explanation of how this works.

'return' 表示某个函数（或别的代码块）即将返回。局部跟踪函数将被调用，*arg* 是即将返回的值，如果此次返回事件是由于抛出异常，*arg* 为 `None`。跟踪函数的返回值将被忽略。

'exception' 表示发生了某个异常。局部跟踪函数将被调用，*arg* 是一个 `(exception, value, traceback)` 元组，返回值将指定新的局部跟踪函数。

注意，由于异常是在链式调用中传播的，所以每一级都会产生一个 `'exception'` 事件。

关于代码对象和帧对象的更多信息请参考 types。

CPython implementation detail: *settrace()* 函数仅用于实现调试器，性能分析器，打包工具等。它的行为是实现平台的一部分，而不是语言定义的一部分，因此并非在所有 Python 实现中都可用。

sys.**set_asyncgen_hooks**(*firstiter*, *finalizer*)
接受两个可选的关键字参数，要求它们是可调用对象，且接受一个异步生成器迭代器 作为参数。*firstiter* 对象将在异步生成器第一次迭代时调用。*finalizer* 将在异步生成器即将被销毁时调用。

3.6 新版功能: 更多详情请参阅 **PEP 525**，*finalizer* 方法的参考示例可参阅 Lib/asyncio/base_events.py 中 `asyncio.Loop.shutdown_asyncgens` 的实现。

注解: 本函数已添加至暂定软件包（详情请参阅 **PEP 411** ）。

`sys.`**`set_coroutine_wrapper`**(*wrapper*)

Allows intercepting creation of *coroutine* objects (only ones that are created by an `async def` function; generators decorated with *types.coroutine()* or *asyncio.coroutine()* will not be intercepted).

The *wrapper* argument must be either:

- a callable that accepts one argument (a coroutine object);
- `None`, to reset the wrapper.

If called twice, the new wrapper replaces the previous one. The function is thread-specific.

The *wrapper* callable cannot define new coroutines directly or indirectly:

```
def wrapper(coro):
    async def wrap(coro):
        return await coro
    return wrap(coro)
sys.set_coroutine_wrapper(wrapper)

async def foo():
    pass

# The following line will fail with a RuntimeError, because
# ``wrapper`` creates a ``wrap(coro)`` coroutine:
foo()
```

See also *get_coroutine_wrapper()*.

3.5 新版功能: See **PEP 492** for more details.

注解: 本函数已添加至暂定软件包（详情请参阅 **PEP 411** ）。仅将其用于调试目的。

`sys.`**`_enablelegacywindowsfsencoding`**()

将默认文件系统编码和错误处理方案分别更改为‘mbcs’和‘replace’，这是为了与 Python 3.6 前的版本保持一致。

这等同于在启动 Python 前先定义好 `PYTHONLEGACYWINDOWSFSENCODING` 环境变量。

Availability: Windows

3.6 新版功能: 有关更多详细信息，请参阅 **PEP 529**。

`sys.`**`stdin`**
`sys.`**`stdout`**
`sys.`**`stderr`**

解释器用于标准输入、标准输出和标准错误的文件对象:

- `stdin` 用于所有交互式输入（包括对*input()* 的调用）;
- `stdout` 用于*print()* 和*expression* 语句的输出，以及用于*input()* 的提示符;
- 解释器自身的提示符和它的错误消息都发往 `stderr`。

这些流都是常规文本文件，与*open()* 函数返回的对象一致。它们的参数选择如下:

- The character encoding is platform-dependent. Under Windows, if the stream is interactive (that is, if its `isatty()` method returns `True`), the console codepage is used, otherwise the ANSI code page. Under other platforms, the locale encoding is used (see *`locale.getpreferredencoding()`*).

  Under all platforms though, you can override this value by setting the `PYTHONIOENCODING` environment variable before starting Python.

- When interactive, standard streams are line-buffered. Otherwise, they are block-buffered like regular text files. You can override this value with the `-u` command-line option.

注解: 要从标准流写入或读取二进制数据，请使用底层二进制*`buffer`* 对象。例如，要将字节写入*`stdout`*，请使用 `sys.stdout.buffer.write(b'abc')`。

但是，如果你在写一个库（并且不限制执行库代码时的上下文），那么请注意，标准流可能会被替换为文件类对象，如*`io.StringIO`*，它们是不支持 `buffer` 属性的。

`sys.`**`__stdin__`**
`sys.`**`__stdout__`**
`sys.`**`__stderr__`**

程序开始时，这些对象存有 `stdin`、`stderr` 和 `stdout` 的初始值。它们在程序结束前都可以使用，且在需要向实际的标准流打印内容时很有用，无论 `sys.std*` 对象是否已重定向。

如果实际文件已经被覆盖成一个损坏的对象了，那它也可用于将实际文件还原成能正常工作的文件对象。但是，本过程的最佳方法应该是，在原来的流被替换之前就显式地保存它，并使用这一保存的对象来还原。

注解: 某些情况下的 `stdin`、`stdout` 和 `stderr` 以及初始值 `__stdin__`、`__stdout__` 和 `__stderr__` 可以是 `None`。通常发生在未连接到控制台的 Windows GUI app 中，以及在用 **`pythonw`** 启动的 Python app 中。

`sys.`**`thread_info`**

A *struct sequence* holding information about the thread implementation.

属性	解释
`name`	线程实现的名称： • `'nt'`: Windows 线程 • `'pthread'`: POSIX 线程 • `'solaris'`: Solaris 线程
`lock`	锁实现的名称： • `'semaphore'`: 锁使用信号量 • `'mutex+cond'`: 锁使用互斥和条件变量 • `None` 如果此信息未知
`version`	Name and version of the thread library. It is a string, or `None` if these informations are unknown.

3.3 新版功能.

`sys.`**`tracebacklimit`**

当该变量值设置为整数，在发生未处理的异常时，它将决定打印的回溯信息的最大层级数。默认为 `1000`。当将其设置为 `0` 或小于 0，将关闭所有回溯信息，并且只打印异常类型和异常值。

sys.**version**
一个包含 Python 解释器版本号加编译版本号以及所用编译器等额外信息的字符串。此字符串会在交互式解释器启动时显示。请不要从中提取版本信息，而应当使用*version_info* 以及*platform* 模块所提供的函数。

sys.**api_version**
这个解释器的 C API 版本。当你在调试 Python 及期扩展模板的版本冲突这个功能非常有用。

sys.**version_info**
一个包含版本号五部分的元组: *major*, *minor*, *micro*, *releaselevel* 和 *serial*。除 *releaselevel* 外的所有值均为整数；发布级别值则为 'alpha', 'beta', 'candidate' 或 'final'。对应于 Python 版本 2.0 的 version_info 值为 (2, 0, 0, 'final', 0)。这些部分也可按名称访问，因此 sys.version_info[0] 就等价于 sys.version_info.major，依此类推。

在 3.1 版更改: 增加了以名称表示的各部分属性。

sys.**warnoptions**
这是警告框架的一个实现细节；请不要修改此值。有关警告框架的更多信息请参阅*warnings* 模块。

sys.**winver**
The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the first three characters of *version*. It is provided in the *sys* module for informational purposes; modifying this value has no effect on the registry keys used by Python. Availability: Windows.

sys.**_xoptions**
由多个具体实现专属的通过 -X 命令行选项传递的旗标构成的字典。选项名称将会映射到显式指定的值或者*True*。例如:

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython implementation detail: 这是 CPython 专属的访问通过 -X 传递的选项的方式。其他实现可能会通过其他方式导出它们，或者完全不导出。

3.2 新版功能.

引用

29.2 sysconfig —Provide access to Python's configuration information

3.2 新版功能.

源代码： Lib/sysconfig.py

The *sysconfig* module provides access to Python's configuration information like the list of installation paths and the configuration variables relevant for the current platform.

29.2.1 配置变量

A Python distribution contains a `Makefile` and a `pyconfig.h` header file that are necessary to build both the Python binary itself and third-party C extensions compiled using *distutils*.

sysconfig puts all variables found in these files in a dictionary that can be accessed using *get_config_vars()* or *get_config_var()*.

Notice that on Windows, it's a much smaller set.

`sysconfig.`**`get_config_vars`**`(`*`*args`*`)`
With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

For each argument, if the value is not found, return `None`.

`sysconfig.`**`get_config_var`**`(`*`name`*`)`
Return the value of a single variable *name*. Equivalent to `get_config_vars().get(name)`.

If *name* is not found, return `None`.

用法示例:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

29.2.2 安装路径

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in *sysconfig* under unique identifiers based on the value returned by *os.name*.

Every new component that is installed using *distutils* or a Distutils-based system will follow the same scheme to copy its file in the right places.

Python currently supports seven schemes:

- *posix_prefix*: scheme for Posix platforms like Linux or Mac OS X. This is the default scheme used when Python or a component is installed.
- *posix_home*: scheme for Posix platforms used when a *home* option is used upon installation. This scheme is used when a component is installed through Distutils with a specific home prefix.
- *posix_user*: scheme for Posix platforms used when a component is installed through Distutils and the *user* option is used. This scheme defines paths located under the user home directory.
- *nt*: scheme for NT platforms like Windows.
- *nt_user*: scheme for NT platforms, when the *user* option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths:

- *stdlib*: directory containing the standard Python library files that are not platform-specific.
- *platstdlib*: directory containing the standard Python library files that are platform-specific.

- *platlib*: directory for site-specific, platform-specific files.
- *purelib*: directory for site-specific, non-platform-specific files.
- *include*: directory for non-platform-specific header files.
- *platinclude*: directory for platform-specific header files.
- *scripts*: directory for script files.
- *data*: directory for data files.

sysconfig provides some functions to determine these paths.

sysconfig.**get_scheme_names**()
Return a tuple containing all schemes currently supported in *sysconfig*.

sysconfig.**get_path_names**()
Return a tuple containing all path names currently supported in *sysconfig*.

sysconfig.**get_path**(*name*[, *scheme*[, *vars*[, *expand*]]])
Return an installation path corresponding to the path *name*, from the install scheme named *scheme*.

name has to be a value from the list returned by *get_path_names()*.

sysconfig stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the *stdlib* path for the *nt* scheme is: `{base}/Lib`.

get_path() will use the variables returned by *get_config_vars()* to expand the path. All variables have default values for each platform so one may call this function and get the default value.

If *scheme* is provided, it must be a value from the list returned by *get_scheme_names()*. Otherwise, the default scheme for the current platform is used.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary return by *get_config_vars()*.

If *expand* is set to `False`, the path will not be expanded using the variables.

If *name* is not found, return `None`.

sysconfig.**get_paths**([*scheme*[, *vars*[, *expand*]]])
Return a dictionary containing all installation paths corresponding to an installation scheme. See *get_path()* for more information.

If *scheme* is not provided, will use the default scheme for the current platform.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If *expand* is set to false, the paths will not be expanded.

If *scheme* is not an existing scheme, *get_paths()* will raise a *KeyError*.

29.2.3 其他功能

sysconfig.**get_python_version**()
Return the `MAJOR.MINOR` Python version number as a string. Similar to `'%d.%d' % sys.version_info[:2]`.

sysconfig.**get_platform**()
Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by *os.uname()*), although the exact

information included depends on the OS; e.g. for IRIX the architecture isn't particularly important (IRIX only runs on SGI hardware), but for Linux the kernel version isn't particularly important.

返回值的示例：

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u
- irix-5.3
- irix64-6.2

Windows 将返回以下之一：

- win-amd64 (在 AMD64, aka x86_64, Intel64, 和 EM64T 上的 64 位 Windows)
- win-ia64 (64bit Windows on Itanium)
- win32 (all others - specifically, sys.platform is returned)

Mac OS X 返回：

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

对于其他非-POSIX 平台，目前只是返回*sys.platform*。

sysconfig.**is_python_build**()
Return `True` if the running Python interpreter was built from source and is being run from its built location, and not from a location resulting from e.g. running `make install` or installing via a binary installer.

sysconfig.**parse_config_h**(*fp*[, *vars*])
Parse a `config.h`-style file.

fp is a file-like object pointing to the `config.h`-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

sysconfig.**get_config_h_filename**()
返回 `pyconfig.h` 的目录

sysconfig.**get_makefile_filename**()
返回 `Makefile` 的目录

29.2.4 Using `sysconfig` as a script

You can use *sysconfig* as a script with Python's *-m* option:

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
        data = "/usr/local"
```

(下页继续)

(续上页)

```
        include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
        platinclude = "."
        platlib = "/usr/local/lib/python3.2/site-packages"
        platstdlib = "/usr/local/lib/python3.2"
        purelib = "/usr/local/lib/python3.2/site-packages"
        scripts = "/usr/local/bin"
        stdlib = "/usr/local/lib/python3.2"

Variables:
        AC_APPLE_UNIVERSAL_BUILD = "0"
        AIX_GENUINE_CPLUSPLUS = "0"
        AR = "ar"
        ARFLAGS = "rc"
        ...
```

This call will print in the standard output the information returned by *get_platform()*, *get_python_version()*, *get_path()* and *get_config_vars()*.

29.3 builtins —内建对象

该模块提供对 Python 的所有 “内置” 标识符的直接访问；例如，builtins.open 是内置函数的全名*open()*。请参阅内置函数 和内置常量 的文档。

大多数应用程序通常不会显式访问此模块，但在提供与内置值同名的对象的模块中可能很有用，但其中还需要内置该名称。例如，在一个想要实现*open()* 函数的模块中，它包装了内置的*open()*，这个模块可以直接使用:

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

作为一个实现细节，大多数模块都将名称 __builtins__ 作为其全局变量的一部分提供。__builtins__ 的值通常是这个模块或者这个模块的值__*dict*__ 属性。由于这是一个实现细节，因此 Python 的替代实现可能不会使用它。

29.4 __main__ —顶层脚本环境

'__main__' 是顶层代码执行的作用域的名称。模块的 __name__ 在通过标准输入、脚本文件或是交互式命令读入的时候会等于 '__main__'。

模块可以通过检查自己的 __name__ 来得知是否运行在 main 作用域中，这使得模块可以在作为脚本或是通过 python -m 运行时条件性地执行一些代码，而在被 import 时不会执行。

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

对软件包来说，通过加入 __main__.py 模块可以达到同样的效果，当使用 -m 运行模块时，其中的代码会被执行。

29.5 warnings —Warning control

源代码： Lib/warnings.py

Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn't warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the *warn()* function defined in this module. (C programmers use PyErr_WarnEx(); see exceptionhandling for details).

Warning messages are normally written to sys.stderr, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category (see below), the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling *filterwarnings()* and reset to its default state by calling *resetwarnings()*.

The printing of warning messages is done by calling *showwarning()*, which may be overridden; the default implementation of this function formats the message by calling *formatwarning()*, which is also available for use by custom implementations.

参见：

logging.captureWarnings() allows you to handle all warnings with the standard logging infrastructure.

29.5.1 警告类别

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings. The following warnings category classes are currently defined:

Class	描述
Warning	This is the base class of all warning category classes. It is a subclass of *Exception*.
UserWarning	The default category for *warn()*.
DeprecationWarning	Base category for warnings about deprecated features (ignored by default).
SyntaxWarning	Base category for warnings about dubious syntactic features.
RuntimeWarning	Base category for warnings about dubious runtime features.
FutureWarning	Base category for warnings about constructs that will change semantically in the future.
PendingDeprecationWarning	Base category for warnings about features that will be deprecated in the future (ignored by default).
ImportWarning	Base category for warnings triggered during the process of importing a module (ignored by default).
UnicodeWarning	Base category for warnings related to Unicode.
BytesWarning	Base category for warnings related to *bytes* and *bytearray*.
ResourceWarning	Base category for warnings related to resource usage.

While these are technically built-in exceptions, they are documented here, because conceptually they belong to the warnings mechanism.

User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the *Warning* class.

29.5.2 The Warnings Filter

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the match determines the disposition of the match. Each entry is a tuple of the form (*action*, *message*, *category*, *module*, *lineno*), where:

- *action* is one of the following strings:

值	处置
"error"	将匹配警告转换为异常
"ignore"	从不打印匹配的警告
"always"	总是打印匹配的警告
"default"	print the first occurrence of matching warnings for each location where the warning is issued
"module"	print the first occurrence of matching warnings for each module where the warning is issued
"once"	无论位置如何，仅打印第一次出现的匹配警告

- *message* is a string containing a regular expression that the start of the warning message must match. The expression is compiled to always be case-insensitive.
- *category* is a class (a subclass of *Warning*) of which the warning category must be a subclass in order to match.
- *module* is a string containing a regular expression that the module name must match. The expression is compiled to be case-sensitive.

- *lineno* is an integer that the line number where the warning occurred must match, or `0` to match all line numbers.

Since the *Warning* class is derived from the built-in *Exception* class, to turn a warning into an error we simply raise `category(message)`.

The warnings filter is initialized by `-W` options passed to the Python interpreter command line. The interpreter saves the arguments for all `-W` options without interpretation in `sys.warnoptions`; the *warnings* module parses these when it is first imported (invalid options are ignored, after printing a message to `sys.stderr`).

Default Warning Filters

By default, Python installs several warning filters, which can be overridden by the command-line options passed to `-W` and calls to *filterwarnings()*.

- *DeprecationWarning* and *PendingDeprecationWarning*, and *ImportWarning* are ignored.
- *BytesWarning* is ignored unless the `-b` option is given once or twice; in this case this warning is either printed (`-b`) or turned into an exception (`-bb`).
- *ResourceWarning* is ignored unless Python was built in debug mode.

在 3.2 版更改: *DeprecationWarning* is now ignored by default in addition to *PendingDeprecationWarning*.

29.5.3 暂时禁止警告

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning, then it is possible to suppress the warning using the *catch_warnings* context manager:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

While within the context manager all warnings will simply be ignored. This allows you to use known-deprecated code without having to see the warning while not suppressing the warning for other code that might not be aware of its use of deprecated code. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the *catch_warnings* context manager at the same time, the behavior is undefined.

29.5.4 测试警告

To test warnings raised by code, use the *catch_warnings* context manager. With it you can temporarily mutate the warnings filter to facilitate your testing. For instance, do the following to capture all raised warnings to check:

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
```

(下页继续)

(续上页)

```
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

One can also cause all warnings to be exceptions by using `error` instead of `always`. One thing to be aware of is that if a warning has already been raised because of a `once`/`default` rule, then no matter what filters are set the warning will not be seen again unless the warnings registry related to the warning has been cleared.

Once the context manager exits, the warnings filter is restored to its state when the context was entered. This prevents tests from changing the warnings filter in unexpected ways between tests and leading to indeterminate test results. The *showwarning()* function in the module is also restored to its original value. Note: this can only be guaranteed in a single-threaded application. If two or more threads use the *catch_warnings* context manager at the same time, the behavior is undefined.

When testing multiple operations that raise the same kind of warning, it is important to test them in a manner that confirms each operation is raising a new warning (e.g. set warnings to be raised as exceptions and check the operations raise exceptions, check that the length of the warning list continues to increase after each operation, or else delete the previous entries from the warnings list before each new operation).

29.5.5 Updating Code For New Versions of Python

Warnings that are only of interest to the developer are ignored by default. As such you should make sure to test your code with typically ignored warnings made visible. You can do this from the command-line by passing `-Wd` to the interpreter (this is shorthand for `-W default`). This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you simply change what argument is passed to `-W`, e.g. `-W error`. See the `-W` flag for more details on what is possible.

To programmatically do the same as `-Wd`, use:

```
warnings.simplefilter('default')
```

Make sure to execute this code as soon as possible. This prevents the registering of what warnings have been raised from unexpectedly influencing how future warnings are treated.

Having certain warnings ignored by default is done to prevent a user from seeing warnings that are only of interest to the developer. As you do not necessarily have control over what interpreter a user uses to run their code, it is possible that a new version of Python will be released between your release cycles. The new interpreter release could trigger new warnings in your code that were not there in an older interpreter, e.g. *DeprecationWarning* for a module that you are using. While you as a developer want to be notified that your code is using a deprecated module, to a user this information is essentially noise and provides no benefit to them.

The *unittest* module has been also updated to use the `'default'` filter while running tests.

29.5.6 Available Functions

warnings.**warn**(*message*, *category=None*, *stacklevel=1*, *source=None*)

Issue a warning, or maybe ignore it or raise an exception. The *category* argument, if given, must be a warning category class (see above); it defaults to `UserWarning`. Alternatively *message* can be a `Warning` instance, in which case *category* will be ignored and `message.__class__` will be used. In this case the message text will be `str(message)`. This function raises an exception if the particular warning issued is changed into an error by the warnings filter see above. The *stacklevel* argument can be used by wrapper functions written in Python, like this:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to `deprecation()`'s caller, rather than to the source of `deprecation()` itself (since the latter would defeat the purpose of the warning message).

source, if supplied, is the destroyed object which emitted a `ResourceWarning`.

在 3.6 版更改: Added *source* parameter.

warnings.**warn_explicit**(*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)

This is a low-level interface to the functionality of `warn()`, passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the `__warningregistry__` dictionary of the module). The module name defaults to the filename with `.py` stripped; if no registry is passed, the warning is never suppressed. *message* must be a string and *category* a subclass of `Warning` or *message* may be a `Warning` instance, in which case *category* will be ignored.

module_globals, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources).

source, if supplied, is the destroyed object which emitted a `ResourceWarning`.

在 3.6 版更改: Add the *source* parameter.

warnings.**showwarning**(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)

Write a warning to a file. The default implementation calls `formatwarning(message, category, filename, lineno, line)` and writes the resulting string to *file*, which defaults to `sys.stderr`. You may replace this function with any callable by assigning to `warnings.showwarning`. *line* is a line of source code to be included in the warning message; if *line* is not supplied, `showwarning()` will try to read the line specified by *filename* and *lineno*.

warnings.**formatwarning**(*message*, *category*, *filename*, *lineno*, *line=None*)

Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline. *line* is a line of source code to be included in the warning message; if *line* is not supplied, `formatwarning()` will try to read the line specified by *filename* and *lineno*.

warnings.**filterwarnings**(*action*, *message=''*, *category=Warning*, *module=''*, *lineno=0*, *append=False*)

Insert an entry into the list of *warnings filter specifications*. The entry is inserted at the front by default; if *append* is true, it is inserted at the end. This checks the types of the arguments, compiles the *message* and *module* regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

warnings.**simplefilter**(*action*, *category=Warning*, *lineno=0*, *append=False*)

Insert a simple entry into the list of *warnings filter specifications*. The meaning of the function parameters is as for `filterwarnings()`, but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

warnings.**resetwarnings**()
Reset the warnings filter. This discards the effect of all previous calls to *filterwarnings()*, including that of the -W command line options and calls to *simplefilter()*.

29.5.7 Available Context Managers

class warnings.**catch_warnings**(*, *record=False*, *module=None*)
A context manager that copies and, upon exit, restores the warnings filter and the *showwarning()* function. If the *record* argument is *False* (the default) the context manager returns *None* on entry. If *record* is *True*, a list is returned that is progressively populated with objects as seen by a custom *showwarning()* function (which also suppresses output to sys.stdout). Each object in the list has attributes with the same names as the arguments to *showwarning()*.

The *module* argument takes a module that will be used instead of the module returned when you import *warnings* whose filter will be protected. This argument exists primarily for testing the *warnings* module itself.

注 解: The *catch_warnings* manager works by replacing and then later restoring the module's *showwarning()* function and internal list of filter specifications. This means the context manager is modifying global state and therefore is not thread-safe.

29.6 contextlib —Utilities for with-statement contexts

源代码 Lib/contextlib.py

此模块为涉及 with 语句的常见任务提供了实用的程序。更多信息请参见上下文管理器类型 和 context-managers。

29.6.1 工具

提供的函数和类:

class contextlib.**AbstractContextManager**
一个为实现了 object.__aenter__() 与 object.__aexit__() 的类提供的*abstract base class*。为 object.__aenter__() 提供的一个默认实现是返回 self 而 object.__aexit__() 是一个默认返回 None 的抽象方法。参见 async-context-managers 的定义。

3.6 新版功能.

@contextlib.**contextmanager**
这个函数是一个*decorator*，它可以定义一个支持 with 语句上下文的工厂函数，而不需要创建一个类或区 __enter__() 与 __exit__() 方法。

尽管许多对象原生支持使用 with 语句，但有些需要被管理的资源并不是上下文管理器，并且没有实现 close() 方法而不能使用 contextlib.closing 。

下面是一个抽象的示例，展示如何确保正确的资源管理:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)

>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

The function being decorated must return a *generator*-iterator when called. This iterator must yield exactly one value, which will be bound to the targets in the `with` statement's `as` clause, if any.

At the point where the generator yields, the block nested in the `with` statement is executed. The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a `try…except…finally` statement to trap the error (if any), or ensure that some cleanup takes place. If an exception is trapped merely in order to log it or to perform some action (rather than to suppress it entirely), the generator must reraise that exception. Otherwise the generator context manager will indicate to the `with` statement that the exception has been handled, and execution will resume with the statement immediately following the `with` statement.

contextmanager() 使用*ContextDecorator* 因此它创建的上下文管理器不仅可以用在 `with` 语句中，还可以用作一个装饰器。当它用作一个装饰器时，每一次函数调用时都会隐式创建一个新的生成器实例（这使得*contextmanager()* 创建的上下文管理器满足了支持多次调用以用作装饰器的需求，而非“一次性”的上下文管理器）。

在 3.2 版更改: *ContextDecorator* 的使用。

`contextlib.`**`closing`**(*thing*)

返回一个在语句块执行完成时关闭 *things* 的上下文管理器。这基本上等价于

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

并允许你编写这样的代码

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('http://www.python.org')) as page:
    for line in page:
        print(line)
```

而无需显式地关闭 `page` 。即使发生错误，在退出 `with` 语句块时，`page.close()` 也同样会被调用。

`contextlib.`**`suppress`**(**exceptions*)

返回一个上下文管理器，如果任何一个指定的异常发生在使用该上下文管理器的 with 语句中，该异常

将被它抑制，然后程序将从 with 语句结束后的第一个语句开始恢复执行。

与完全抑制异常的任何其他机制一样，该上下文管理器应当只用来抑制非常具体的错误，并确保该场景下静默地继续执行程序是通用的正确做法。

例如

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

这段代码等价于：

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

该上下文管理器是*reentrant* 。

3.4 新版功能.

contextlib.**redirect_stdout**(*new_target*)

用于将*sys.stdout* 临时重定向到一个文件或类文件对象的上下文管理器。

该工具给已有的将输出硬编码写到 stdout 的函数或类提供了额外的灵活性。

例如，*help()* 通常把输出写到 *sys.stdout* 。你可以通过重定向到一个*io.StringIO* 来捕获该输出到一个字符串中。

```
f = io.StringIO()
with redirect_stdout(f):
    help(pow)
s = f.getvalue()
```

如果要把*help()* 的输出写到磁盘上的一个文件，重定向该输出到一个常规文件：

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

如果要把*help()* 的输出写到 *sys.stderr*：

```
with redirect_stdout(sys.stderr):
    help(pow)
```

需要注意的点在于，*sys.stdout* 的全局副作用意味着此上下文管理器不适合在库代码和大多数多线程应用程序中使用。它对子进程的输出没有影响。不过对于许多工具脚本而言，它仍然是一个有用的方法。

该上下文管理器是*reentrant* 。

3.4 新版功能.

contextlib.**redirect_stderr**(*new_target*)
与*redirect_stdout()* 类似，不过是将*sys.stderr* 重定向到一个文件或类文件对象。

该上下文管理器是*reentrant* 。

3.5 新版功能.

class contextlib.**ContextDecorator**
一个使上下文管理器能用作装饰器的基类。

与往常一样，继承自 ContextDecorator 的上下文管理器必须实现 __enter__ 与 __exit__ 。即使用作装饰器，__exit__ 依旧会保持可能的异常处理。

ContextDecorator 被用在*contextmanager()* 中，因此你自然获得了这项功能。

ContextDecorator 的示例:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False

>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing

>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

这个改动只是针对如下形式的一个语法糖:

```
def f():
    with cm():
        # Do stuff
```

ContextDecorator 使得你可以这样改写:

```
@cm()
def f():
    # Do stuff
```

这能清楚地表明，cm 作用于整个函数，而不仅仅是函数的一部分（同时也能保持不错的缩进层级）。

现有的上下文管理器即使已经有基类，也可以使用 ContextDecorator 作为混合类进行扩展:

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

注解： As the decorated function must be able to be called multiple times, the underlying context manager must support use in multiple `with` statements. If this is not the case, then the original construct with the explicit `with` statement inside the function should be used.

3.2 新版功能.

class contextlib.**ExitStack**

该上下文管理器的设计目标是使得在编码中组合其他上下文管理器和清理函数更加容易，尤其是那些可选的或由输入数据驱动的上下文管理器。

例如，通过一个如下的 with 语句可以很容易处理一组文件：

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

每个实例维护一个注册了一组回调的栈，这些回调在实例关闭时以相反的顺序被调用（显式或隐式地在 `with` 语句的末尾）。请注意，当一个栈实例被垃圾回收时，这些回调将 不会被隐式调用。

通过使用这个基于栈的模型，那些通过 `__init__` 方法获取资源的上下文管理器（如文件对象）能够被正确处理。

由于注册的回调函数是按照与注册相反的顺序调用的，因此最终的行为就像多个嵌套的 `with` 语句用在这些注册的回调函数上。这个行为甚至扩展到了异常处理：如果内部的回调函数抑制或替换了异常，则外部回调收到的参数是基于该更新后的状态得到的。

这是一个相对底层的 API，它负责正确处理栈里回调退出时依次展开的细节。它为相对高层的上下文管理器提供了一个合适的基础，使得它能根据应用程序的需求使用特定方式操作栈。

3.3 新版功能.

enter_context(*cm*)

Enters a new context manager and adds its `__exit__()` method to the callback stack. The return value is the result of the context manager's own `__enter__()` method.

These context managers may suppress exceptions just as they normally would if used directly as part of a `with` statement.

push(*exit*)

Adds a context manager's `__exit__()` method to the callback stack.

As `__enter__` is *not* invoked, this method can be used to cover part of an `__enter__()` implementation with a context manager's own `__exit__()` method.

If passed an object that is not a context manager, this method assumes it is a callback with the same signature as a context manager's `__exit__()` method and adds it directly to the callback stack.

By returning true values, these callbacks can suppress exceptions the same way context manager `__exit__()` methods can.

The passed in object is returned from the function, allowing this method to be used as a function decorator.

callback(*callback*, **args*, ***kwds*)
Accepts an arbitrary callback function and arguments and adds it to the callback stack.

Unlike the other methods, callbacks added this way cannot suppress exceptions (as they are never passed the exception details).

The passed in callback is returned from the function, allowing this method to be used as a function decorator.

pop_all()
Transfers the callback stack to a fresh *ExitStack* instance and returns it. No callbacks are invoked by this operation - instead, they will now be invoked when the new stack is closed (either explicitly or implicitly at the end of a `with` statement).

For example, a group of files can be opened as an "all or nothing" operation as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be
    # closed automatically. If all files are opened successfully,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

close()
Immediately unwinds the callback stack, invoking callbacks in the reverse order of registration. For any context managers and exit callbacks registered, the arguments passed in will indicate that no exception occurred.

29.6.2 例子和配方

This section describes some examples and recipes for making effective use of the tools provided by *contextlib*.

Supporting a variable number of context managers

The primary use case for *ExitStack* is the one given in the class documentation: supporting a variable number of context managers and other cleanup operations in a single `with` statement. The variability may come from the number of context managers needed being driven by user input (such as opening a user specified collection of files), or from some of the context managers being optional:

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # Perform operations that use the acquired resources
```

As shown, *ExitStack* also makes it quite easy to use `with` statements to manage arbitrary resources that don't natively support the context management protocol.

Simplifying support for single optional context managers

In the specific case of a single optional context manager, *ExitStack* instances can be used as a "do nothing" context manager, allowing a context manager to easily be omitted without affecting the overall structure of the source code:

```
def debug_trace(details):
    if __debug__:
        return TraceContext(details)
    # Don't do anything special with the context in release mode
    return ExitStack()

with debug_trace():
    # Suite is traced in debug mode, but runs normally otherwise
```

Catching exceptions from __enter__ methods

It is occasionally desirable to catch exceptions from an `__enter__` method implementation, *without* inadvertently catching exceptions from the `with` statement body or the context manager's `__exit__` method. By using *ExitStack* the steps in the context management protocol can be separated slightly in order to allow this:

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case
```

Actually needing to do this is likely to indicate that the underlying API should be providing a direct resource management interface for use with `try`/`except`/`finally` statements, but not all APIs are well designed in that regard. When a context manager is the only resource management API provided, then *ExitStack* can make it easier to handle various situations that can't be handled directly in a `with` statement.

Cleaning up in an __enter__ implementation

As noted in the documentation of *ExitStack.push()*, this method can be useful in cleaning up an already allocated resource if later steps in the `__enter__()` implementation fail.

Here's an example of doing this for a context manager that accepts resource acquisition and release functions, along with an optional validation function, and maps them to the context management protocol:

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok
```

(下页继续)

(续上页)

```
    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # The validation check passed and didn't raise an exception
            # Accordingly, we want to keep the resource, and pass it
            # back to our caller
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource

    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource release logic
        self.release_resource()
```

Replacing any use of `try-finally` and flag variables

A pattern you will sometimes see is a `try-finally` statement with a flag variable to indicate whether or not the body of the `finally` clause should be executed. In its simplest form (that can't already be handled just by using an `except` clause instead), it looks something like this:

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

As with any `try` statement based code, this can cause problems for development and review, because the setup code and the cleanup code can end up being separated by arbitrarily long sections of code.

ExitStack makes it possible to instead register a callback for execution at the end of a `with` statement, and then later decide to skip executing that callback:

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

This allows the intended cleanup up behaviour to be made explicit up front, rather than requiring a separate flag variable.

If a particular application uses this pattern a lot, it can be simplified even further by means of a small helper class:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, *args, **kwds):
        super(Callback, self).__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

If the resource cleanup isn't already neatly bundled into a standalone function, then it is still possible to use the decorator form of *ExitStack.callback()* to declare the resource cleanup in advance:

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
    if result:
        stack.pop_all()
```

Due to the way the decorator protocol works, a callback function declared this way cannot take any parameters. Instead, any resources to be released must be accessed as closure variables.

Using a context manager as a function decorator

ContextDecorator makes it possible to use a context manager in both an ordinary `with` statement and also as a function decorator.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, inheriting from *ContextDecorator* provides both capabilities in a single definition:

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

Instances of this class can be used as both a context manager:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

And also as a function decorator:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Note that there is one additional limitation when using context managers as function decorators: there's no way to access the return value of `__enter__()`. If that value is needed, then it is still necessary to use an explicit `with` statement.

参见:

PEP 343 - "with" 语句 Python `with` 语句的规范描述、背景和示例。

29.6.3 Single use, reusable and reentrant context managers

Most context managers are written in a way that means they can only be used effectively in a `with` statement once. These single use context managers must be created afresh each time they're used - attempting to use them a second time will trigger an exception or otherwise not work correctly.

This common limitation means that it is generally advisable to create context managers directly in the header of the `with` statement where they are used (as shown in all of the usage examples above).

Files are an example of effectively single use context managers, since the first `with` statement will close the file, preventing any further IO operations using that file object.

Context managers created using *`contextmanager()`* are also single use context managers, and will complain about the underlying generator failing to yield if an attempt is made to use them a second time:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

Reentrant context managers

More sophisticated context managers may be "reentrant". These context managers can not only be used in multiple `with` statements, but may also be used *inside* a `with` statement that is already using the same context manager.

threading.RLock is an example of a reentrant context manager, as are *suppress()* and *redirect_stdout()*. Here's a very simple example of reentrant use:

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

Real world examples of reentrancy are more likely to involve multiple functions calling each other and hence be far more complicated than this example.

Note also that being reentrant is *not* the same thing as being thread safe. *redirect_stdout()*, for example, is definitely not thread safe, as it makes a global modification to the system state by binding *sys.stdout* to a different stream.

Reusable context managers

Distinct from both single use and reentrant context managers are "reusable" context managers (or, to be completely explicit, "reusable, but not reentrant" context managers, since reentrant context managers are also reusable). These context managers support being used multiple times, but will fail (or otherwise not work correctly) if the specific context manager instance has already been used in a containing with statement.

threading.Lock is an example of a reusable, but not reentrant, context manager (for a reentrant lock, it is necessary to use *threading.RLock* instead).

Another example of a reusable, but not reentrant, context manager is *ExitStack*, as it invokes *all* currently registered callbacks when leaving any with statement, regardless of where those callbacks were added:

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
```

(下页继续)

(续上页)

```
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

As the output from the example shows, reusing a single stack object across multiple with statements works correctly, but attempting to nest them will cause the stack to be cleared at the end of the innermost with statement, which is unlikely to be desirable behaviour.

Using separate *ExitStack* instances instead of reusing a single instance avoids that problem:

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

29.7 abc —抽象基类

源代码： Lib/abc.py

该模块提供了在 Python 中定义抽象基类 (ABC) 的组件，在 **PEP 3119** 中已有概述。查看 PEP 文档了解为什么需要在 Python 中增加这个模块。（也可查看 **PEP 3141** 以及*numbers* 模块了解基于 ABC 的数字类型继承关系。）

collections 模块中有一些派生自 ABC 的具体类；当然这些类还可以进一步被派生。此外，*collections.abc* 子模块中有一些 ABC 可被用于测试一个类或实例是否提供特定的接口，例如它是否可哈希或它是否为映射等。

该模块提供了一个元类*ABCMeta*，可以用来定义抽象类，另外还提供一个工具类*ABC*，可以用它以继承的方式定义抽象基类。

class abc.**ABC**

一个使用*ABCMeta* 作为元类的工具类。抽象基类可以通过从*ABC* 派生来简单地创建，这就避免了在某些情况下会令人混淆的元类用法，例如：

```
from abc import ABC

class MyABC(ABC):
    pass
```

注意*ABC* 的类型仍然是*ABCMeta*，因此继承*ABC* 仍然需要关注元类使用中的注意事项，比如可能会导致元类冲突的多重继承。当然你也可以直接使用*ABCMeta* 作为元类来定义抽象基类，例如：

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

3.4 新版功能.

class abc.**ABCMeta**

用于定义抽象基类（ABC）的元类。

使用该元类以创建抽象基类。抽象基类可以像 mix-in 类一样直接被子类继承。你也可以将不相关的具体类（包括内建类）和抽象基类注册为“抽象子类”——这些类以及它们的子类会被内建函数*issubclass()* 识别为对应的抽象基类的子类，但是该抽象基类不会出现在其 MRO（Method Resolution Order，方法解析顺序）中，抽象基类中实现的方法也不可调用（即使通过*super()* 调用也不行）。[1]

使用*ABCMeta* 作为元类创建的类含有如下方法：

register(*subclass*)

将“子类”注册为该抽象基类的“抽象子类”，例如：

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

在 3.3 版更改: 返回注册的子类，使其能够作为类装饰器。

在 3.4 版更改: 你可以使用*get_cache_token()* 函数来检测对*register()* 的调用。

你也可以在虚基类中重载这个方法。

__subclasshook__(*subclass*)

（必须定义为类方法。）

检查 *subclass* 是否是该抽象基类的子类。也就是说对于那些你希望定义为该抽象基类的子类的类，你不用对每个类都调用*register()* 方法了，而是可以直接自定义 issubclass 的行为。（这个类方法是在抽象基类的 __subclasscheck__() 方法中调用的。）

该方法必须返回 True, False 或是 NotImplemented。如果返回 True，*subclass* 就会被认为是这个抽象基类的子类。如果返回 False，无论正常情况是否应该认为是其子类，统一视为不是。如果返回 NotImplemented，子类检查会按照正常机制继续执行。

为了对这些概念做一演示，请看以下定义 ABC 的示例：

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
```

(下页继续)

[1] C++ 程序员需要注意：Python 中虚基类的概念和 C++ 中的并不相同。

(续上页)

```
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented

MyIterable.register(Foo)
```

ABC `MyIterable` 定义了标准的迭代方法*__iter__()* 作为一个抽象方法。这里给出的实现仍可在子类中被调用。`get_iterator()` 方法也是 `MyIterable` 抽象基类的一部分，但它并非必须被非抽象派生类所重载。

这里定义的*__subclasshook__()* 类方法指明了任何在其*__dict__* (或在其通过*__mro__* 列表访问的基类) 中具有*__iter__()* 方法的类也都会被视为 `MyIterable`。

最后，末尾行使得 `Foo` 成为 `MyIterable` 的一个虚子类，即使它没有定义*__iter__()* 方法（它使用了以 `__len__()` 和 `__getitem__()` 术语定义的旧式可迭代对象协议）。请注意这将不会使 `get_iterator` 成为 `Foo` 的一个可用方法，它是被另外提供的。

此外，*abc* 模块还提供了这些装饰器：

@abc.**abstractmethod**

用于声明抽象方法的装饰器。

使用此装饰器要求类的元类是*ABCMeta* 或是从该类派生。一个具有派生自*ABCMeta* 的元类的类不可以被实例化，除非它全部的抽象方法和特征属性均已被重载。抽象方法可通过任何普通的“super”调用机制来调用。*abstractmethod()* 可被用于声明特性属性和描述器的抽象方法。

不支持动态添加抽象方法到一个类，或试图在方法或类被创建后修改其抽象状态等操作。*abstractmethod()* 只会影响使用常规继承所派生的子类；通过 ABC 的 `register()` 方法注册的“虚子类”不会受到影响。

当*abstractmethod()* 与其他方法描述符配合应用时，它应当被应用为最内层的装饰器，如以下用法示例所示：

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, ...):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
    @staticmethod
    @abstractmethod
```

(下页继续)

(续上页)

```
    def my_abstract_staticmethod(...):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

为了能正确地与抽象基类机制实现互操作，描述符必须使用 `__isabstractmethod__` 将自身标识为抽象的。通常，如果被用于组成描述符的任何方法都是抽象的则此属性应当为 `True`。例如，Python 的内置*property* 所做的就等价于:

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                   f in (self._fget, self._fset, self._fdel))
```

注解: 不同于 Java 抽象方法，这些抽象方法可能具有一个实现。这个实现可在重载它的类上通过*super()* 机制来调用。这在使用协作多重继承的框架中可以被用作超调用的一个端点。

abc 模块还支持下列旧式装饰器:

@abc.**abstractclassmethod**

3.2 新版功能.

3.3 版后已移除: 现在可以让*classmethod* 配合*abstractmethod()* 使用，使得此装饰器变得冗余。

内置*classmethod()* 的子类，指明一个抽象类方法。在其他情况下它都类似于*abstractmethod()*。

这个特殊情况已被弃用，因为现在*classmethod()* 当将装饰器应用于抽象方法时它会被正确地标识为抽象的:

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, ...):
        ...
```

@abc.**abstractstaticmethod**

3.2 新版功能.

3.3 版后已移除: 现在可以让*staticmethod* 配合*abstractmethod()* 使用，使得此装饰器变得冗余。

内置*staticmethod()* 的子类，指明一个抽象静态方法。在其他方面它都类似于*abstractmethod()*。

这个特殊情况已被弃用，因为现在当*staticmethod()* 装饰器应用于抽象方法时它会被正确地标识为抽象的:

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(...):
        ...
```

@abc.**abstractproperty**

3.3 版后已移除: 现在可以让*property*，property.getter()，property.setter() 和 property.deleter() 配合*abstractmethod()* 使用，使得此装饰器变得冗余。

内置*property()* 的子类，指明一个抽象特性属性。

这个特例已被弃用，因为现在当*property()* 装饰器应用于抽象方法时它会被正确地标识为抽象的:

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

上面的例子定义了一个只读特征属性；你也可以通过适当地将一个或多个下层方法标记为抽象的来定义可读写的抽象特征属性:

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

如果只有某些组件是抽象的，则只需更新那些组件即可在子类中创建具体的特征属性:

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

abc 模块还提供了这些函数:

abc.**get_cache_token**()

返回当前抽象基类的缓存令牌

此令牌是一个不透明对象（支持相等性测试），用于为虚子类标识抽象基类缓存的当前版本。此令牌会在任何 ABC 上每次调用*ABCMeta.register()* 时发生更改。

3.4 新版功能.

备注

29.8 atexit —退出处理器

atexit 模块定义了清理函数的注册和反注册函数. 被注册的函数会在解释器正常终止时执行. *atexit* 会按照注册顺序的 * 逆序 * 执行; 如果你注册了 A, B 和 C, 那么在解释器终止时会依序执行 C, B, A.

注意: 通过该模块注册的函数, 在程序被未被 Python 捕获的信号杀死时并不会执行, 在检测到 Python 内部致命错误以及调用了*os._exit()* 时也不会执行.

atexit.**register**(*func*, **args*, ***kwargs*)

将 *func* 注册为终止时执行的函数. 任何传给 *func* 的可选的参数都应当作为参数传给*register()*. 可以多次注册同样的函数及参数.

在正常的程序终止时 (举例来说, 当调用了*sys.exit()* 或是主模块的执行完成时), 所有注册过的函数都会以后进先出的顺序执行. 这样做是假定更底层的模块通常会比高层模块更早引入, 因此需要更晚清理.

如果在 exit 处理程序执行期间引发了异常，将会打印回溯信息 (除非引发的是*SystemExit*) 并且异常信息会被保存。在所有 exit 处理程序获得运行机会之后，所引发的最后一个异常会被重新引发。

这个函数返回 *func* 对象，可以把它当作装饰器使用。

atexit.**unregister**(*func*)

从解释器关闭前要运行的函数列表中移除 *func*。在调用*unregister()* 之后，当解释器关闭时会确保 *func* 不会被调用，即使它被多次注册。如果 *func* 之前没有被注册，*unregister()* 会静默地不做任何操作。

参见:

模块*readline* 使用*atexit* 读写*readline* 历史文件的有用的例子。

29.8.1 atexit 示例

以下简单例子演示了一个模块在被导入时如何从文件初始化一个计数器，并在程序终结时自动保存计数器的更新值，此操作不依赖于应用在终结时对此模块进行显式调用。:

```
try:
    with open("counterfile") as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    with open("counterfile", "w") as outfile:
        outfile.write("%d" % _count)

import atexit
atexit.register(savecounter)
```

位置和关键字参数也可传入*register()* 以便传递给被调用的已注册函数:

```
def goodbye(name, adjective):
    print('Goodbye, %s, it was %s to meet you.' % (name, adjective))

import atexit
atexit.register(goodbye, 'Donny', 'nice')

# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

作为*decorator*: 使用:

```
import atexit

@atexit.register
def goodbye():
    print("You are now leaving the Python sector.")
```

只有在函数不需要任何参数调用时才能工作.

29.9 traceback —打印或检索堆栈回溯

源代码： Lib/traceback.py

该模块提供了一个标准接口来提取、格式化和打印 Python 程序的堆栈跟踪结果。它完全模仿 Python 解释器在打印堆栈跟踪结果时的行为。当您想要在程序控制下打印堆栈跟踪结果时，例如在“封装”解释器时，这是非常有用的。

这个模块使用 traceback 对象——这是存储在*sys.last_traceback* 中的对象类型变量，并作为*sys.exc_info()* 的第三项被返回。

这个模块定义了以下函数:

traceback.**print_tb**(*tb*, *limit=None*, *file=None*)
如果 *limit 是正整数，那么从 traceback 对象“tb”输出最高 *limit* 个（从调用函数开始的）栈的堆栈回溯 条目；如果 *limit* 是负数就输出 abs(limit) 个回溯条目；又如果 *limit* 被省略或者为 None，那么就会输出所有回溯条目。如果 *file* 被省略或为 None 那么就会输出至标准输出 “sys.stderr“否则它应该是一个打开的文件或者文件类对象来接收输出

在 3.5 版更改: 添加了对负数值 *limit* 的支持

traceback.**print_exception**(*etype*, *value*, *tb*, *limit=None*, *file=None*, *chain=True*)
打印回溯对象 *tb* 到 *file* 的异常信息和整个堆栈回溯。这和*print_tb()* 比有以下方面不同:

- 如果 *tb* 不为 None，它将打印头部 Traceback (most recent call last):
- 它将在输出完堆栈回溯后，输出异常中的 *etype* 和 *value* 信息
- if *type(value)* is *SyntaxError* and *value* has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

The optional *limit* argument has the same meaning as for *print_tb()*. If *chain* is true (the default), then chained exceptions (the __cause__ or __context__ attributes of the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

在 3.5 版更改: The *etype* argument is ignored and inferred from the type of *value*.

traceback.**print_exc**(*limit=None*, *file=None*, *chain=True*)
This is a shorthand for `print_exception(*sys.exc_info(), limit, file, chain)`.

traceback.**print_last**(*limit=None*, *file=None*, *chain=True*)
This is a shorthand for `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file, chain)`. In general it will work only after an exception has reached an interactive prompt (see *sys.last_type*).

traceback.**print_stack**(*f=None*, *limit=None*, *file=None*)
Print up to *limit* stack trace entries (starting from the invocation point) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *file* argument has the same meaning as for *print_tb()*.

在 3.5 版更改: 添加了对负数值 *limit* 的支持

traceback.**extract_tb**(*tb*, *limit=None*)
Return a *StackSummary* object representing a list of "pre-processed" stack trace entries extracted from the traceback object *tb*. It is useful for alternate formatting of stack traces. The optional *limit* argument has the same meaning as for *print_tb()*. A "pre-processed" stack trace entry is a *FrameSummary* object containing attributes `filename`, `lineno`, `name`, and `line` representing the information that is usually printed for a stack trace. The `line` is a string with leading and trailing whitespace stripped; if the source is not available it is `None`.

traceback.**extract_stack**(*f=None*, *limit=None*)
Extract the raw traceback from the current stack frame. The return value has the same format as for *extract_tb()*. The optional *f* and *limit* arguments have the same meaning as for *print_stack()*.

traceback.**format_list**(*extracted_list*)
Given a list of tuples or *FrameSummary* objects as returned by *extract_tb()* or *extract_stack()*, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not `None`.

traceback.**format_exception_only**(*etype*, *value*)
Format the exception part of a traceback. The arguments are the exception type and value such as given by `sys.last_type` and `sys.last_value`. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for *SyntaxError* exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. The message indicating which exception occurred is the always last string in the list.

traceback.**format_exception**(*etype*, *value*, *tb*, *limit=None*, *chain=True*)
Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to *print_exception()*. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does *print_exception()*.

在 3.5 版更改: The *etype* argument is ignored and inferred from the type of *value*.

traceback.**format_exc**(*limit=None*, *chain=True*)
This is like `print_exc(limit)` but returns a string instead of printing to a file.

traceback.**format_tb**(*tb*, *limit=None*)
A shorthand for `format_list(extract_tb(tb, limit))`.

traceback.**format_stack**(*f=None*, *limit=None*)
A shorthand for `format_list(extract_stack(f, limit))`.

traceback.**clear_frames**(*tb*)
Clears the local variables of all the stack frames in a traceback *tb* by calling the `clear()` method of each frame object.

3.4 新版功能.

traceback.**walk_stack**(*f*)
Walk a stack following `f.f_back` from the given frame, yielding the frame and line number for each frame. If *f* is `None`, the current stack is used. This helper is used with *StackSummary.extract()*.

3.5 新版功能.

traceback.**walk_tb**(*tb*)
Walk a traceback following `tb_next` yielding the frame and line number for each frame. This helper is used with *StackSummary.extract()*.

3.5 新版功能.

The module also defines the following classes:

29.9.1 TracebackException Objects

3.5 新版功能.

TracebackException objects are created from actual exceptions to capture data for later printing in a lightweight fashion.

class traceback.**TracebackException**(*exc_type*, *exc_value*, *exc_traceback*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*)
Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the *StackSummary* class.

Note that when locals are captured, they are also shown in the traceback.

__cause__
A *TracebackException* of the original `__cause__`.

__context__
A *TracebackException* of the original `__context__`.

__suppress_context__
The `__suppress_context__` value from the original exception.

stack
A *StackSummary* representing the traceback.

exc_type
The class of the original traceback.

filename
For syntax errors - the file name where the error occurred.

lineno
For syntax errors - the line number where the error occurred.

text
For syntax errors - the text where the error occurred.

offset
For syntax errors - the offset into the text where the error occurred.

msg
For syntax errors - the compiler error message.

classmethod from_exception(*exc*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*)
Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the *StackSummary* class.

Note that when locals are captured, they are also shown in the traceback.

format(*, *chain=True*)
Format the exception.

If *chain* is not `True`, `__cause__` and `__context__` will not be formatted.

The return value is a generator of strings, each ending in a newline and some containing internal newlines. *print_exception()* is a wrapper around this method which just prints the lines to a file.

The message indicating which exception occurred is always the last string in the output.

format_exception_only()
Format the exception part of the traceback.

The return value is a generator of strings, each ending in a newline.

Normally, the generator emits a single string; however, for *SyntaxError* exceptions, it emits several lines that (when printed) display detailed information about where the syntax error occurred.

The message indicating which exception occurred is always the last string in the output.

29.9.2 `StackSummary` Objects

3.5 新版功能.

StackSummary objects represent a call stack ready for formatting.

class traceback.**StackSummary**

classmethod extract(*frame_gen*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)
Construct a *StackSummary* object from a frame generator (such as is returned by *walk_stack()* or *walk_tb()*).

If *limit* is supplied, only this many frames are taken from *frame_gen*. If *lookup_lines* is `False`, the returned *FrameSummary* objects will not have read their lines in yet, making the cost of creating the *StackSummary* cheaper (which may be valuable if it may not actually get formatted). If *capture_locals* is `True` the local variables in each *FrameSummary* are captured as object representations.

classmethod from_list(*a_list*)
Construct a *StackSummary* object from a supplied list of *FrameSummary* objects or old-style list of tuples. Each tuple should be a 4-tuple with filename, lineno, name, line as the elements.

format()
Returns a list of strings ready for printing. Each string in the resulting list corresponds to a single frame from the stack. Each string ends in a newline; the strings may contain internal newlines as well, for those items with source text lines.

For long sequences of the same frame and line, the first few repetitions are shown, followed by a summary line stating the exact number of further repetitions.

在 3.6 版更改: Long sequences of repeated frames are now abbreviated.

29.9.3 FrameSummary Objects

3.5 新版功能.

FrameSummary objects represent a single frame in a traceback.

class traceback.**FrameSummary**(*filename*, *lineno*, *name*, *lookup_line=True*, *locals=None*, *line=None*)

Represent a single frame in the traceback or stack that is being formatted or printed. It may optionally have a stringified version of the frames locals included in it. If *lookup_line* is `False`, the source code is not looked up until the *FrameSummary* has the `line` attribute accessed (which also happens when casting it to a tuple). `line` may be directly provided, and will prevent line lookups happening at all. *locals* is an optional local variable dictionary, and if supplied the variable representations are stored in the summary for later display.

29.9.4 Traceback Examples

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the *code* module.

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback:

```
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print("*** print_tb:")
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print("*** print_exception:")
    # exc_type below is ignored on 3.5 and later
    traceback.print_exception(exc_type, exc_value, exc_traceback,
                              limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
```

(下页继续)

(续上页)

```
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    # exc_type below is ignored on 3.5 and later
    print(repr(traceback.format_exception(exc_type, exc_value,
                                          exc_traceback)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc_traceback)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc_traceback)))
    print("*** tb_lineno:", exc_traceback.tb_lineno)
```

The output for the example would look similar to this:

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_death>]
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n']
*** tb_lineno: 10
```

The following example shows the different ways to print and format the stack:

```
>>> import traceback
>>> def another_function():
```

(下页继续)

(续上页)

```
...      lumberstack()
...
>>> def lumberstack():
...      traceback.print_stack()
...      print(repr(traceback.extract_stack()))
...      print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_
↪stack()))\n']
```

This last example demonstrates the final few formatting functions:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(type(an_error), an_error)
['IndexError: tuple index out of range\n']
```

29.10 __future__ —Future 语句定义

源代码： Lib/__future__.py

__future__ 是一个真正的模块，这主要有 3 个原因：

- 避免混淆已有的分析 import 语句并查找 import 的模块的工具。
- 确保 future 语句在 2.1 之前的版本运行时至少能抛出 runtime 异常（import *__future__* 会失败，因为 2.1 版本之前没有这个模块）。
- 当引入不兼容的修改时，可以记录其引入的时间以及强制使用的时间。这是一种可执行的文档，并且可以通过 import *__future__* 来做程序性的检查。

__future__.py 中的每一条语句都是以下格式的：

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

通常 *OptionalRelease* 要比 *MandatoryRelease* 小，并且都是和*sys.version_info* 格式一致的 5 元素元组。

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

OptionalRelease 记录了一个特性首次发布时的 Python 版本。

在 *MandatoryRelases* 还没有发布时，*MandatoryRelease* 表示该特性会变成语言的一部分的预测时间。

其他情况下，*MandatoryRelease* 用来记录这个特性是何时成为语言的一部分的。从该版本往后，使用该特性将不需要 future 语句，不过很多人还是会加上对应的 import。

MandatoryRelease 也可能是 `None`, 表示这个特性已经被撤销。

`_Feature` 类的实例有两个对应的方法，`getOptionalRelease()` 和 `getMandatoryRelease()`。

CompilerFlag 是一个（位）标记，对于动态编译的代码，需要将这个标记作为第四个参数传入内建函数 *compile()* 中以开启对应的特性。这个标记存储在 `_Feature` 类实例的 `compiler_flag` 属性中。

__future__ 中不会删除特性的描述。从 Python 2.1 中首次加入以来，通过这种方式引入了以下特性：

特性	可选版本	强制加入版本	效果
nested_scopes	2.1.0b1	2.2	**PEP 227**: 静态嵌套作用域
generators	2.2.0a1	2.3	**PEP 255**: 简单生成器
division	2.2.0a2	3.0	**PEP 238**: 修改除法运算符
absolute_import	2.5.0a1	3.0	**PEP 328**: 导入：多行与绝对/相对
with_statement	2.5.0a1	2.6	**PEP 343**: * “with” 语句 *
print_function	2.6.0a2	3.0	**PEP 3105**: *print* 改为函数
unicode_literals	2.6.0a2	3.0	**PEP 3112**: *Python 3000* 中的字节字面值
generator_stop	3.5.0b1	3.7	**PEP 479**: 在生成器中处理 *StopIteration*

参见:

future 编译器怎样处理 future import。

29.11 gc —垃圾回收器接口

此模块提供可选的垃圾回收器的接口，提供的功能包括：关闭收集器、调整收集频率、设置调试选项。它同时提供对回收器找到但是无法释放的不可达对象的访问。由于 Python 使用了带有引用计数的回收器，如果你确定你的程序不会产生循环引用，你可以关闭回收器。可以通过调用 `gc.disable()` 关闭自动垃圾回收。若要调试一个存在内存泄漏的程序，调用 `gc.set_debug(gc.DEBUG_LEAK)`；需要注意的是，它包含 `gc.DEBUG_SAVEALL`，使得被垃圾回收的对象会被存放在 gc.garbage 中以待检查。

gc 模块提供下列函数：

`gc.`**`enable`**`()`
启用自动垃圾回收

gc.**disable**()
停用自动垃圾回收

gc.**isenabled**()
Returns true if automatic collection is enabled.

gc.**collect**(*generation=2*)
若被调用时不包含参数，则启动完全的垃圾回收。可选的参数 *generation* 可以是一个整数，指明需要回收哪一代（从 0 到 2 ）的垃圾。当参数 *generation* 无效时，会引发*ValueError* 异常。返回发现的不可达对象的数目。

每当运行完整收集或最高代 (2) 收集时，为多个内置类型所维护的空闲列表会被清空。由于特定类型特别是*float* 的实现，在某些空闲列表中并非所有项都会被释放。

gc.**set_debug**(*flags*)
设置垃圾回收器的调试标识位。调试信息会被写入 sys.stderr 。此文档末尾列出了各个标志位及其含义；可以使用位操作对多个标志位进行设置以控制调试器。

gc.**get_debug**()
返回当前调试标识位。

gc.**get_objects**()
返回一个含有所有被收集器跟踪的对象的列表。返回的列表中不包括该函数返回的对象。

gc.**get_stats**()
返回一个包含三个字典对象的列表，每个字典分别包含对应代的从解释器开始运行的垃圾回收统计数据。字典的键的数目在将来可能发生改变，目前每个字典包含以下内容：

- collections 是该代被回收的次数；
- collected 是该代中被回收的对象总数；
- uncollectable 是在这一代中被发现无法收集的对象总数（因此被移动到*garbage* 列表中）。

3.4 新版功能.

gc.**set_threshold**(*threshold0*[, *threshold1*[, *threshold2*]])
设置垃圾回收阈值（收集频率）。将 *threshold0* 设为零会禁用回收。

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation 0). If an object survives a collection it is moved into the next older generation. Since generation 2 is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds *threshold0*, collection starts. Initially only generation 0 is examined. If generation 0 has been examined more than *threshold1* times since generation 1 has been examined, then generation 1 is examined as well. Similarly, *threshold2* controls the number of collections of generation 1 before collecting generation 2.

gc.**get_count**()
将当前回收计数以形为 (count0, count1, count2) 的元组返回。

gc.**get_threshold**()
将当前回收阈值以形为 (threshold0, threshold1, threshold2) 的元组返回。

gc.**get_referrers**(**objs*)
返回直接引用任意一个 *ojbs* 的对象列表。这个函数只定位支持垃圾回收的容器；引用了其它对象但不支持垃圾回收的扩展类型不会被找到。

需要注意的是，已经解除对 *objs* 引用的对象，但仍存在于循环引用中未被回收时，仍然会被作为引用者出现在返回的列表当中。若要获取当前正在引用 *objs* 的对象，需要调用*collect()* 然后再调用*get_referrers()* 。

在使用*get_referrers()* 返回的对象时必须要小心，因为其中一些对象可能仍在构造中因此处于暂时的无效状态。不要把*get_referrers()* 用于调试以外的其它目的。

gc.**get_referents**(**objs*)
返回被任意一个参数中的对象直接引用的对象的列表。返回的被引用对象是被参数中的对象的 C 语言级别方法（若存在）tp_traverse 访问到的对象，可能不是所有的实际直接可达对象。只有支持垃圾回收的对象支持 tp_traverse 方法，并且此方法只会在需要访问涉及循环引用的对象时使用。因此，可以有以下例子：一个整数对其中一个参数是直接可达的，这个整数有可能出现或不出现在返回的结果列表当中。

gc.**is_tracked**(*obj*)
当对象正在被垃圾回收器监控时返回 True，否则返回 False。一般来说，原子类的实例不会被监控，而非原子类（如容器、用户自定义的对象）会被监控。然而，会有一些特定类型的优化以便减少垃圾回收器在简单实例（如只含有原子性的键和值的字典）上的消耗。

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

3.1 新版功能.

提供以下变量仅供只读访问（你可以修改但不应该重绑定它们）：

gc.**garbage**
A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). Starting with Python 3.4, this list should be empty most of the time, except when using instances of C extension types with a non-NULL tp_del slot.

如果设置了*DEBUG_SAVEALL*，则所有不可访问对象将被添加至该列表而不会被释放。

在 3.2 版更改: 当*interpreter shutdown* 即解释器关闭时，若此列表非空，会产生*ResourceWarning*，即资源警告，在默认情况下此警告不会被提醒。如果设置了*DEBUG_UNCOLLECTABLE*，所有无法被回收的对象会被打印。

在 3.4 版更改: 根据 **PEP 442**，带有 __del__() 方法的对象最终不再会进入*gc.garbage*。

gc.**callbacks**
在垃圾回收器开始前和完成后会被调用的一系列回调函数。这些回调函数在被调用时使用两个参数：*phase* 和 *info*。

phase 可为以下两值之一：

“start”：垃圾回收即将开始。

“stop”：垃圾回收已结束。

info is a dict providing more information for the callback. The following keys are currently defined:

“generation”（代）：正在被回收的最久远的一代。

“collected”（已回收的）：当 *phase* 为 “stop” 时，被成功回收的对象的数目。

“uncollectable”（不可回收的）：当 *phase* 为 “stop” 时，不能被回收并被放入*garbage* 的对象的数目。

应用程序可以把他们自己的回调函数加入此列表。主要的使用场景有：

统计垃圾回收的数据，如：不同代的回收频率、回收所花费的时间。

使应用程序可以识别和清理他们自己的在*garbage* 中的不可回收类型的对象。

3.3 新版功能.

以下常量被用于*set_debug()*：

gc.**DEBUG_STATS**
在回收完成后打印统计信息。当回收频率设置较高时，这些信息会比较有用。

gc.**DEBUG_COLLECTABLE**
当发现可回收对象时打印信息。

gc.**DEBUG_UNCOLLECTABLE**
打印找到的不可回收对象的信息（指不能被回收器回收的不可达对象）。这些对象会被添加到 garbage 列表中。

在 3.2 版更改: 当*interpreter shutdown* 时，即解释器关闭时，若*garbage* 列表中存在对象，这些对象也会被打印输出。

gc.**DEBUG_SAVEALL**
设置后，所有回收器找到的不可达对象会被添加进 *garbage* 而不是直接被释放。这在调试一个内存泄漏的程序时会很有用。

gc.**DEBUG_LEAK**
调试内存泄漏的程序时，使回收器打印信息的调试标识位。（等价于 DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL）。

29.12 inspect —检查对象

源代码: Lib/inspect.py

---

inspect 模块提供了一些有用的函数帮助获取对象的信息，例如模块、类、方法、函数、回溯、帧对象以及代码对象。例如它可以帮助你检查类的内容，获取某个方法的源代码，取得并格式化某个函数的参数列表，或者获取你需要显示的回溯的详细信息。

该模块提供了 4 种主要的功能：类型检查、获取源代码、检查类与函数、检查解释器的调用堆栈。

29.12.1 类型和成员

getmembers() 函数获取对象的成员，例如类或模块。函数名以” is” 开始的函数主要作为*getmembers()* 的第 2 个参数使用。它们也可用于判定某对象是否有如下的特殊属性:

类型	属性	描述
module 模块	__doc__	文档字符串
	__file__	文件名 (内置模块没有文件名)
class –类	__doc__	文档字符串
	__name__	类定义时所使用的名称
	__qualname__	qualified name –限定名称
	__module__	该类型被定义时所在的模块的名称
method 方法	__doc__	文档字符串

下页

表 1 – 续上页

类型	属性	描述
	__name__	该方法定义时所使用的名称
	__qualname__	qualified name –限定名称
	__func__	实现该方法的函数对象
	__self__	该方法被绑定的实例，若没有绑定则为 None
function –函数	__doc__	文档字符串
	__name__	用于定义此函数的名称
	__qualname__	qualified name –限定名称
	__code__	包含已编译函数的代码对象*bytecode*
	__defaults__	所有位置或关键字参数的默认值的元组
	__kwdefaults__	mapping of any default values for keyword-only parameters
	__globals__	global namespace in which this function was defined
	__annotations__	mapping of parameters names to annotations; "return" key is reserved for return annotat
回溯	tb_frame	此级别的框架对象
	tb_lasti	index of last attempted instruction in bytecode
	tb_lineno	current line number in Python source code
	tb_next	next inner traceback object (called by this level)
框架	f_back	next outer frame object (this frame’s caller)
	f_builtins	builtins namespace seen by this frame
	f_code	code object being executed in this frame
	f_globals	global namespace seen by this frame
	f_lasti	index of last attempted instruction in bytecode
	f_lineno	current line number in Python source code
	f_locals	local namespace seen by this frame
	f_trace	tracing function for this frame, or None
code	co_argcount	number of arguments (not including keyword only arguments, * or ** args)
	co_code	原始编译字节码的字符串
	co_cellvars	单元变量名称的元组 (通过包含作用域引用)
	co_consts	字节码中使用的常量元组
	co_filename	创建此代码对象的文件的名称
	co_firstlineno	number of first line in Python source code
	co_flags	bitmap of CO_* flags, read more *here*
	co_lnotab	编码的行号到字节码索引的映射
	co_freevars	tuple of names of free variables (referenced via a function’s closure)
	co_kwonlyargcount	number of keyword only arguments (not including ** arg)
	co_name	定义此代码对象的名称
	co_names	局部变量名称的元组
	co_nlocals	局部变量的数量
	co_stacksize	需要虚拟机堆栈空间
	co_varnames	参数名和局部变量的元组
generator –生成器	__name__	名称
	__qualname__	qualified name –限定名称
	gi_frame	框架
	gi_running	生成器在运行吗?
	gi_code	code
	gi_yieldfrom	object being iterated by yield from, or None
coroutine –协程	__name__	名称
	__qualname__	qualified name –限定名称
	cr_await	object being awaited on, or None
	cr_frame	框架

下页

表 1 – 续上页

类型	属性	描述
	cr_running	is the coroutine running?
	cr_code	code
builtin	__doc__	文档字符串
	__name__	此函数或方法的原始名称
	__qualname__	qualified name –限定名称
	__self__	instance to which a method is bound, or `None`

在 3.5 版更改: Add `__qualname__` and `gi_yieldfrom` attributes to generators.

The `__name__` attribute of generators is now set from the function name, instead of the code name, and it can now be modified.

`inspect.`**`getmembers`**(*object*[, *predicate*])

Return all the members of an object in a list of (name, value) pairs sorted by name. If the optional *predicate* argument is supplied, only members for which the predicate returns a true value are included.

注解: *`getmembers()`* will only return class attributes defined in the metaclass when the argument is a class and those attributes have been listed in the metaclass' custom `__dir__()`.

`inspect.`**`getmodulename`**(*path*)

Return the name of the module named by the file *path*, without including the names of enclosing packages. The file extension is checked against all of the entries in *`importlib.machinery.all_suffixes()`*. If it matches, the final path component is returned with the extension removed. Otherwise, `None` is returned.

Note that this function *only* returns a meaningful name for actual Python modules - paths that potentially refer to Python packages will still return `None`.

在 3.3 版更改: The function is based directly on *`importlib`*.

`inspect.`**`ismodule`**(*object*)

Return true if the object is a module.

`inspect.`**`isclass`**(*object*)

Return true if the object is a class, whether built-in or created in Python code.

`inspect.`**`ismethod`**(*object*)

Return true if the object is a bound method written in Python.

`inspect.`**`isfunction`**(*object*)

Return true if the object is a Python function, which includes functions created by a *lambda* expression.

`inspect.`**`isgeneratorfunction`**(*object*)

Return true if the object is a Python generator function.

`inspect.`**`isgenerator`**(*object*)

Return true if the object is a generator.

`inspect.`**`iscoroutinefunction`**(*object*)

Return true if the object is a *coroutine function* (a function defined with an `async def` syntax).

3.5 新版功能.

`inspect.`**`iscoroutine`**(*object*)

Return true if the object is a *coroutine* created by an `async def` function.

3.5 新版功能.

inspect.**isawaitable**(*object*)
Return true if the object can be used in `await` expression.

Can also be used to distinguish generator-based coroutines from regular generators:

```
def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

3.5 新版功能.

inspect.**isasyncgenfunction**(*object*)
Return true if the object is an *asynchronous generator* function, for example:

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

3.6 新版功能.

inspect.**isasyncgen**(*object*)
Return true if the object is an *asynchronous generator iterator* created by an *asynchronous generator* function.

3.6 新版功能.

inspect.**istraceback**(*object*)
Return true if the object is a traceback.

inspect.**isframe**(*object*)
Return true if the object is a frame.

inspect.**iscode**(*object*)
Return true if the object is a code.

inspect.**isbuiltin**(*object*)
Return true if the object is a built-in function or a bound built-in method.

inspect.**isroutine**(*object*)
Return true if the object is a user-defined or built-in function or method.

inspect.**isabstract**(*object*)
Return true if the object is an abstract base class.

inspect.**ismethoddescriptor**(*object*)
Return true if the object is a method descriptor, but not if *ismethod()*, *isclass()*, *isfunction()* or *isbuiltin()* are true.

This, for example, is true of `int.__add__`. An object passing this test has a `__get__()` method but not a `__set__()` method, but beyond that the set of attributes varies. A *__name__* attribute is usually sensible, and `__doc__` often is.

Methods implemented via descriptors that also pass one of the other tests return false from the *ismethoddescriptor()* test, simply because the other tests promise more –you can, e.g., count on having the `__func__` attribute (etc) when an object passes *ismethod()*.

inspect.**isdatadescriptor**(*object*)
Return true if the object is a data descriptor.

Data descriptors have both a __get__ and a __set__ method. Examples are properties (defined in Python), getsets, and members. The latter two are defined in C and there are more specific tests available for those types, which is robust across Python implementations. Typically, data descriptors will also have *__name__* and __doc__ attributes (properties, getsets, and members have both of these attributes), but this is not guaranteed.

inspect.**isgetsetdescriptor**(*object*)
Return true if the object is a getset descriptor.

CPython implementation detail: getsets are attributes defined in extension modules via PyGetSetDef structures. For Python implementations without such types, this method will always return False.

inspect.**ismemberdescriptor**(*object*)
Return true if the object is a member descriptor.

CPython implementation detail: Member descriptors are attributes defined in extension modules via PyMemberDef structures. For Python implementations without such types, this method will always return False.

29.12.2 Retrieving source code

inspect.**getdoc**(*object*)
Get the documentation string for an object, cleaned up with *cleandoc()*. If the documentation string for an object is not provided and the object is a class, a method, a property or a descriptor, retrieve the documentation string from the inheritance hierarchy.

在 3.5 版更改: Documentation strings are now inherited if not overridden.

inspect.**getcomments**(*object*)
Return in a single string any lines of comments immediately preceding the object's source code (for a class, function, or method), or at the top of the Python source file (if the object is a module). If the object's source code is unavailable, return None. This could happen if the object has been defined in C or the interactive shell.

inspect.**getfile**(*object*)
Return the name of the (text or binary) file in which an object was defined. This will fail with a *TypeError* if the object is a built-in module, class, or function.

inspect.**getmodule**(*object*)
Try to guess which module an object was defined in.

inspect.**getsourcefile**(*object*)
Return the name of the Python source file in which an object was defined. This will fail with a *TypeError* if the object is a built-in module, class, or function.

inspect.**getsourcelines**(*object*)
Return a list of source lines and starting line number for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a list of the lines corresponding to the object and the line number indicates where in the original source file the first line of code was found. An *OSError* is raised if the source code cannot be retrieved.

在 3.3 版更改: *OSError* is raised instead of *IOError*, now an alias of the former.

inspect.**getsource**(*object*)
Return the text of the source code for an object. The argument may be a module, class, method, function, traceback, frame, or code object. The source code is returned as a single string. An *OSError* is raised if the source code cannot be retrieved.

在 3.3 版更改: *OSError* is raised instead of *IOError*, now an alias of the former.

inspect.**cleandoc**(*doc*)
Clean up indentation from docstrings that are indented to line up with blocks of code.

All leading whitespace is removed from the first line. Any leading whitespace that can be uniformly removed from the second line onwards is removed. Empty lines at the beginning and end are subsequently removed. Also, all tabs are expanded to spaces.

29.12.3 Introspecting callables with the Signature object

3.3 新版功能.

The Signature object represents the call signature of a callable object and its return annotation. To retrieve a Signature object, use the *signature()* function.

inspect.**signature**(*callable*, *, *follow_wrapped=True*)
Return a *Signature* object for the given `callable`:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b:int, **kwargs)'

>>> str(sig.parameters['b'])
'b:int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

Accepts a wide range of python callables, from plain functions and classes to *functools.partial()* objects.

Raises *ValueError* if no signature can be provided, and *TypeError* if that type of object is not supported.

3.5 新版功能: `follow_wrapped` parameter. Pass `False` to get a signature of `callable` specifically (`callable.__wrapped__` will not be used to unwrap decorated callables.)

注解: Some callables may not be introspectable in certain implementations of Python. For example, in CPython, some built-in functions defined in C provide no metadata about their arguments.

---

class inspect.**Signature**(*parameters=None*, *, *return_annotation=Signature.empty*)
A Signature object represents the call signature of a function and its return annotation. For each parameter accepted by the function it stores a *Parameter* object in its *parameters* collection.

The optional *parameters* argument is a sequence of *Parameter* objects, which is validated to check that there are no parameters with duplicate names, and that the parameters are in the right order, i.e. positional-only first, then positional-or-keyword, and that parameters with defaults follow parameters without defaults.

The optional *return_annotation* argument, can be an arbitrary Python object, is the "return" annotation of the callable.

Signature objects are *immutable*. Use *Signature.replace()* to make a modified copy.

在 3.5 版更改: Signature objects are picklable and hashable.

empty
A special class-level marker to specify absence of a return annotation.

parameters
An ordered mapping of parameters' names to the corresponding *Parameter* objects.

return_annotation
The "return" annotation for the callable. If the callable has no "return" annotation, this attribute is set to *Signature.empty*.

bind(**args*, ***kwargs*)
Create a mapping from positional and keyword arguments to parameters. Returns *BoundArguments* if `*args` and `**kwargs` match the signature, or raises a *TypeError*.

bind_partial(**args*, ***kwargs*)
Works the same way as *Signature.bind()*, but allows the omission of some required arguments (mimics *functools.partial()* behavior.) Returns *BoundArguments*, or raises a *TypeError* if the passed arguments do not match the signature.

replace(**[, parameters][, return_annotation]*)
Create a new Signature instance based on the instance replace was invoked on. It is possible to pass different `parameters` and/or `return_annotation` to override the corresponding properties of the base signature. To remove return_annotation from the copied Signature, pass in *Signature.empty*.

```
>>> def test(a, b):
...     pass
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

classmethod from_callable(*obj*, ***, *follow_wrapped=True*)
Return a *Signature* (or its subclass) object for a given callable `obj`. Pass `follow_wrapped=False` to get a signature of `obj` without unwrapping its `__wrapped__` chain.

This method simplifies subclassing of *Signature*:

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(min)
assert isinstance(sig, MySignature)
```

3.5 新版功能.

class inspect.**Parameter**(*name*, *kind*, ***, *default=Parameter.empty*, *annotation=Parameter.empty*)
Parameter objects are *immutable*. Instead of modifying a Parameter object, you can use *Parameter.replace()* to create a modified copy.

在 3.5 版更改: Parameter objects are picklable and hashable.

empty
A special class-level marker to specify absence of default values and annotations.

name
The name of the parameter as a string. The name must be a valid Python identifier.

CPython implementation detail: CPython generates implicit parameter names of the form `.0` on the code objects used to implement comprehensions and generator expressions.

在 3.6 版更改: These parameter names are exposed by this module as names like `implicit0`.

default
The default value for the parameter. If the parameter has no default value, this attribute is set to *Parameter.empty*.

annotation
The annotation for the parameter. If the parameter has no annotation, this attribute is set to *Parameter.empty*.

kind
Describes how argument values are bound to the parameter. Possible values (accessible via *Parameter*, like `Parameter.KEYWORD_ONLY`):

名称	含义
POSITIONAL_ONLY	Value must be supplied as a positional argument. Python has no explicit syntax for defining positional-only parameters, but many built-in and extension module functions (especially those that accept only one or two parameters) accept them.
POSITIONAL_OR_KEYWORD	Value may be supplied as either a keyword or positional argument (this is the standard binding behaviour for functions implemented in Python.)
VAR_POSITIONAL	A tuple of positional arguments that aren't bound to any other parameter. This corresponds to a `*args` parameter in a Python function definition.
KEYWORD_ONLY	Value must be supplied as a keyword argument. Keyword only parameters are those which appear after a `*` or `*args` entry in a Python function definition.
VAR_KEYWORD	A dict of keyword arguments that aren't bound to any other parameter. This corresponds to a `**kwargs` parameter in a Python function definition.

Example: print all keyword-only arguments without default values:

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                         param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

replace(**[, name][, kind][, default][, annotation]*)

Create a new Parameter instance based on the instance replaced was invoked on. To override a *Parameter* attribute, pass the corresponding argument. To remove a default value or/and an annotation from a Parameter, pass *Parameter.empty*.

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'

>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'
```

(下页继续)

(续上页)

```
>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo:'spam'"
```

在 3.4 版更改: In Python 3.3 Parameter objects were allowed to have `name` set to `None` if their `kind` was set to `POSITIONAL_ONLY`. This is no longer permitted.

class inspect.**BoundArguments**

Result of a *Signature.bind()* or *Signature.bind_partial()* call. Holds the mapping of arguments to the function's parameters.

arguments

An ordered, mutable mapping (*collections.OrderedDict*) of parameters' names to arguments' values. Contains only explicitly bound arguments. Changes in *arguments* will reflect in *args* and *kwargs*.

Should be used in conjunction with *Signature.parameters* for any argument processing purposes.

注解: Arguments for which *Signature.bind()* or *Signature.bind_partial()* relied on a default value are skipped. However, if needed, use *BoundArguments.apply_defaults()* to add them.

---

args

A tuple of positional arguments values. Dynamically computed from the *arguments* attribute.

kwargs

A dict of keyword arguments values. Dynamically computed from the *arguments* attribute.

signature

A reference to the parent *Signature* object.

apply_defaults()

Set default values for missing arguments.

For variable-positional arguments (`*args`) the default is an empty tuple.

For variable-keyword arguments (`**kwargs`) the default is an empty dict.

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
OrderedDict([('a', 'spam'), ('b', 'ham'), ('args', ())])
```

3.5 新版功能.

The *args* and *kwargs* properties can be used to invoke functions:

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

参见:

PEP 362 - Function Signature Object. The detailed specification, implementation details and examples.

29.12.4 类与函数

inspect.**getclasstree**(*classes*, *unique=False*)

Arrange the given list of classes into a hierarchy of nested lists. Where a nested list appears, it contains classes derived from the class whose entry immediately precedes the list. Each entry is a 2-tuple containing a class and a tuple of its base classes. If the *unique* argument is true, exactly one entry appears in the returned structure for each class in the given list. Otherwise, classes using multiple inheritance and their descendants will appear multiple times.

inspect.**getargspec**(*func*)

Get the names and default values of a Python function's parameters. A *named tuple* `ArgSpec(args, varargs, keywords, defaults)` is returned. *args* is a list of the parameter names. *varargs* and *keywords* are the names of the `*` and `**` parameters or `None`. *defaults* is a tuple of default argument values or `None` if there are no default arguments; if this tuple has *n* elements, they correspond to the last *n* elements listed in *args*.

3.0 版后已移除: Use *getfullargspec()* for an updated API that is usually a drop-in replacement, but also correctly handles function annotations and keyword-only parameters.

Alternatively, use *signature()* and *Signature Object*, which provide a more structured introspection API for callables.

inspect.**getfullargspec**(*func*)

Get the names and default values of a Python function's parameters. A *named tuple* is returned:

```
FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults,
annotations)
```

args is a list of the positional parameter names. *varargs* is the name of the `*` parameter or `None` if arbitrary positional arguments are not accepted. *varkw* is the name of the `**` parameter or `None` if arbitrary keyword arguments are not accepted. *defaults* is an *n*-tuple of default argument values corresponding to the last *n* positional parameters, or `None` if there are no such defaults defined. *kwonlyargs* is a list of keyword-only parameter names. *kwonlydefaults* is a dictionary mapping parameter names from *kwonlyargs* to the default values used if no argument is supplied. *annotations* is a dictionary mapping parameter names to annotations. The special key `"return"` is used to report the function return value annotation (if any).

Note that *signature()* and *Signature Object* provide the recommended API for callable introspection, and support additional behaviours (like positional-only arguments) that are sometimes encountered in extension module APIs. This function is retained primarily for use in code that needs to maintain compatibility with the Python 2 `inspect` module API.

在 3.4 版更改: This function is now based on *signature()*, but still ignores `__wrapped__` attributes and includes the already bound first parameter in the signature output for bound methods.

在 3.6 版更改: This method was previously documented as deprecated in favour of *signature()* in Python 3.5, but that decision has been reversed in order to restore a clearly supported standard interface for single-source Python 2/3 code migrating away from the legacy *getargspec()* API.

inspect.**getargvalues**(*frame*)

Get information about arguments passed into a particular frame. A *named tuple* `ArgInfo(args, varargs, keywords, locals)` is returned. *args* is a list of the argument names. *varargs* and *keywords* are the names of the `*` and `**` arguments or `None`. *locals* is the locals dictionary of the given frame.

注解: This function was inadvertently marked as deprecated in Python 3.5.

---

inspect.**formatargspec**(*args*[, *varargs*, *varkw*, *defaults*, *kwonlyargs*, *kwonlydefaults*, *annotations*[, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*, *formatreturns*, *formatannotations*]])

Format a pretty argument spec from the values returned by *getfullargspec()*.

The first seven arguments are (`args`, `varargs`, `varkw`, `defaults`, `kwonlyargs`, `kwonlydefaults`, `annotations`).

The other six arguments are functions that are called to turn argument names, `*` argument name, `**` argument name, default values, return annotation and individual annotations into strings, respectively.

例如:

```
>>> from inspect import formatargspec, getfullargspec
>>> def f(a: int, b: float):
...     pass
...
>>> formatargspec(*getfullargspec(f))
'(a: int, b: float)'
```

3.5 版后已移除: Use *`signature()`* and *Signature Object*, which provide a better introspecting API for callables.

`inspect.`**`formatargvalues`**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

Format a pretty argument spec from the four values returned by *`getargvalues()`*. The format* arguments are the corresponding optional formatting functions that are called to turn names and values into strings.

注解: This function was inadvertently marked as deprecated in Python 3.5.

`inspect.`**`getmro`**(*cls*)

Return a tuple of class cls' s base classes, including cls, in method resolution order. No class appears more than once in this tuple. Note that the method resolution order depends on cls' s type. Unless a very peculiar user-defined metatype is in use, cls will be the first element of the tuple.

`inspect.`**`getcallargs`**(*func*, **args*, ***kwds*)

Bind the *args* and *kwds* to the argument names of the Python function or method *func*, as if it was called with them. For bound methods, bind also the first argument (typically named `self`) to the associated instance. A dict is returned, mapping the argument names (including the names of the `*` and `**` arguments, if any) to their values from *args* and *kwds*. In case of invoking *func* incorrectly, i.e. whenever `func(*args, **kwds)` would raise an exception because of incompatible signature, an exception of the same type and the same or similar message is raised. For example:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

3.2 新版功能.

3.5 版后已移除: Use *`Signature.bind()`* and *`Signature.bind_partial()`* instead.

`inspect.`**`getclosurevars`**(*func*)

Get the mapping of external name references in a Python function or method *func* to their current values. A *named tuple* `ClosureVars(nonlocals, globals, builtins, unbound)` is returned. *nonlocals* maps referenced names to lexical closure variables, *globals* to the function' s module globals and *builtins* to the

builtins visible from the function body. *unbound* is the set of names referenced in the function that could not be resolved at all given the current module globals and builtins.

`TypeError` is raised if *func* is not a Python function or method.

3.3 新版功能.

`inspect.`**`unwrap`**(*func*, *, *stop=None*)

Get the object wrapped by *func*. It follows the chain of `__wrapped__` attributes returning the last object in the chain.

stop is an optional callback accepting an object in the wrapper chain as its sole argument that allows the unwrapping to be terminated early if the callback returns a true value. If the callback never returns a true value, the last object in the chain is returned as usual. For example, `signature()` uses this to stop unwrapping if any object in the chain has a `__signature__` attribute defined.

`ValueError` is raised if a cycle is encountered.

3.4 新版功能.

29.12.5 The interpreter stack

When the following functions return "frame records," each record is a *named tuple* `FrameInfo(frame, filename, lineno, function, code_context, index)`. The tuple contains the frame object, the filename, the line number of the current line, the function name, a list of lines of context from the source code, and the index of the current line within that list.

在 3.5 版更改: Return a named tuple instead of a tuple.

注解: Keeping references to frame objects, as found in the first element of the frame records these functions return, can cause your program to create reference cycles. Once a reference cycle has been created, the lifespan of all objects which can be accessed from the objects which form the cycle can become much longer even if Python's optional cycle detector is enabled. If such cycles must be created, it is important to ensure they are explicitly broken to avoid the delayed destruction of objects and increased memory consumption which occurs.

Though the cycle detector will catch these, destruction of the frames (and local variables) can be made deterministic by removing the cycle in a `finally` clause. This is also important if the cycle detector was disabled when Python was compiled or using `gc.disable()`. For example:

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

If you want to keep the frame around (for example to print a traceback later), you can also break reference cycles by using the `frame.clear()` method.

The optional *context* argument supported by most of these functions specifies the number of lines of context to return, which are centered around the current line.

`inspect.`**`getframeinfo`**(*frame*, *context=1*)

Get information about a frame or traceback object. A *named tuple* `Traceback(filename, lineno, function, code_context, index)` is returned.

`inspect.`**`getouterframes`**(*frame*, *context=1*)

Get a list of frame records for a frame and all outer frames. These frames represent the calls that lead to the creation

of *frame*. The first entry in the returned list represents *frame*; the last entry represents the outermost call on *frame*'s stack.

在 3.5 版更改: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.`**`getinnerframes`**(*traceback*, *context=1*)

Get a list of frame records for a traceback's frame and all inner frames. These frames represent calls made as a consequence of *frame*. The first entry in the list represents *traceback*; the last entry represents where the exception was raised.

在 3.5 版更改: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.`**`currentframe`**()

Return the frame object for the caller's stack frame.

CPython implementation detail: This function relies on Python stack frame support in the interpreter, which isn't guaranteed to exist in all implementations of Python. If running in an implementation without Python stack frame support this function returns `None`.

`inspect.`**`stack`**(*context=1*)

Return a list of frame records for the caller's stack. The first entry in the returned list represents the caller; the last entry represents the outermost call on the stack.

在 3.5 版更改: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

`inspect.`**`trace`**(*context=1*)

Return a list of frame records for the stack between the current frame and the frame in which an exception currently being handled was raised in. The first entry in the list represents the caller; the last entry represents where the exception was raised.

在 3.5 版更改: A list of *named tuples* `FrameInfo(frame, filename, lineno, function, code_context, index)` is returned.

29.12.6 Fetching attributes statically

Both *`getattr()`* and *`hasattr()`* can trigger code execution when fetching or checking for the existence of attributes. Descriptors, like properties, will be invoked and `__getattr__()` and `__getattribute__()` may be called.

For cases where you want passive introspection, like documentation tools, this can be inconvenient. *`getattr_static()`* has the same signature as *`getattr()`* but avoids executing code when it fetches attributes.

`inspect.`**`getattr_static`**(*obj*, *attr*, *default=None*)

Retrieve attributes without triggering dynamic lookup via the descriptor protocol, `__getattr__()` or `__getattribute__()`.

Note: this function may not be able to retrieve all attributes that getattr can fetch (like dynamically created attributes) and may find attributes that getattr can't (like descriptors that raise AttributeError). It can also return descriptors objects instead of instance members.

If the instance *`__dict__`* is shadowed by another member (for example a property) then this function will be unable to find instance members.

3.2 新版功能.

`getattr_static()` does not resolve descriptors, for example slot descriptors or getset descriptors on objects implemented in C. The descriptor object is returned instead of the underlying attribute.

You can handle these with code like the following. Note that for arbitrary getset descriptors invoking these may trigger code execution:

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
        # in which case the descriptor itself will
        # have to do
        pass
```

29.12.7 Current State of Generators and Coroutines

When implementing coroutine schedulers and for other advanced uses of generators, it is useful to determine whether a generator is currently executing, is waiting to start or resume or execution, or has already terminated. *getgeneratorstate()* allows the current state of a generator to be determined easily.

inspect.**getgeneratorstate**(*generator*)

Get current state of a generator-iterator.

Possible states are:

- GEN_CREATED: Waiting to start execution.
- GEN_RUNNING: Currently being executed by the interpreter.
- GEN_SUSPENDED: Currently suspended at a yield expression.
- GEN_CLOSED: Execution has completed.

3.2 新版功能.

inspect.**getcoroutinestate**(*coroutine*)

Get current state of a coroutine object. The function is intended to be used with coroutine objects created by `async def` functions, but will accept any coroutine-like object that has `cr_running` and `cr_frame` attributes.

Possible states are:

- CORO_CREATED: Waiting to start execution.
- CORO_RUNNING: Currently being executed by the interpreter.
- CORO_SUSPENDED: Currently suspended at an await expression.
- CORO_CLOSED: Execution has completed.

3.5 新版功能.

The current internal state of the generator can also be queried. This is mostly useful for testing purposes, to ensure that internal state is being updated as expected:

inspect.**getgeneratorlocals**(*generator*)
Get the mapping of live local variables in *generator* to their current values. A dictionary is returned that maps from variable names to values. This is the equivalent of calling *locals()* in the body of the generator, and all the same caveats apply.

If *generator* is a *generator* with no currently associated frame, then an empty dictionary is returned. *TypeError* is raised if *generator* is not a Python generator object.

CPython implementation detail: This function relies on the generator exposing a Python stack frame for introspection, which isn't guaranteed to be the case in all implementations of Python. In such cases, this function will always return an empty dictionary.

3.3 新版功能.

inspect.**getcoroutinelocals**(*coroutine*)
This function is analogous to *getgeneratorlocals()*, but works for coroutine objects created by `async def` functions.

3.5 新版功能.

29.12.8 Code Objects Bit Flags

Python code objects have a `co_flags` attribute, which is a bitmap of the following flags:

inspect.**CO_OPTIMIZED**
The code object is optimized, using fast locals.

inspect.**CO_NEWLOCALS**
If set, a new dict will be created for the frame's `f_locals` when the code object is executed.

inspect.**CO_VARARGS**
The code object has a variable positional parameter (`*args`-like).

inspect.**CO_VARKEYWORDS**
The code object has a variable keyword parameter (`**kwargs`-like).

inspect.**CO_NESTED**
The flag is set when the code object is a nested function.

inspect.**CO_GENERATOR**
The flag is set when the code object is a generator function, i.e. a generator object is returned when the code object is executed.

inspect.**CO_NOFREE**
The flag is set if there are no free or cell variables.

inspect.**CO_COROUTINE**
The flag is set when the code object is a coroutine function. When the code object is executed it returns a coroutine object. See **PEP 492** for more details.

3.5 新版功能.

inspect.**CO_ITERABLE_COROUTINE**
The flag is used to transform generators into generator-based coroutines. Generator objects with this flag can be used in `await` expression, and can `yield from` coroutine objects. See **PEP 492** for more details.

3.5 新版功能.

inspect.**CO_ASYNC_GENERATOR**
The flag is set when the code object is an asynchronous generator function. When the code object is executed it returns an asynchronous generator object. See **PEP 525** for more details.

3.6 新版功能.

注解： The flags are specific to CPython, and may not be defined in other Python implementations. Furthermore, the flags are an implementation detail, and can be removed or deprecated in future Python releases. It's recommended to use public APIs from the *inspect* module for any introspection needs.

29.12.9 命令行界面

The *inspect* module also provides a basic introspection capability from the command line.

By default, accepts the name of a module and prints the source of that module. A class or function within the module can be printed instead by appended a colon and the qualified name of the target object.

--details
Print information about the specified object rather than the source code

29.13 site ——站点专属的配置钩子

源代码: Lib/site.py

这个模块将在初始化时被自动导入。此自动导入可以通过使用解释器的 -S 选项来屏蔽。

导入此模块将会附加域特定的路径到模块搜索路径并且添加一些内建对象，除非使用了 -S 选项。那样的话，模块可以被安全地导入，而不会对模块搜索路径和内建对象有自动的修改或添加。要明确地触发通常域特定的添加，调用函数*site.main()*。

在 3.3 版更改: 在之前即便使用了 -S，导入此模块仍然会触发路径操纵。

它会从头部和尾部构建至多四个目录作为起点。对于头部，它会使用 sys.prefix 和 sys.exec_prefix；空的头部会被跳过。对于尾部，它会使用空字符串然后是 lib/site-packages (在 Windows 上) 或 lib/pythonX.Y/site-packages (在 Unix 和 Macintosh 上)。对于每个不同的头-尾组合，它会查看其是否指向现有的目录，如果是的话，则将其添加到 sys.path 并且检查新添加目录中的配置文件。

在 3.5 版更改: 对“site-python”目录的支持已被移除。

If a file named “pyvenv.cfg” exists one directory above sys.executable, sys.prefix and sys.exec_prefix are set to that directory and it is also checked for site-packages (sys.base_prefix and sys.base_exec_prefix will always be the “real” prefixes of the Python installation). If “pyvenv.cfg” (a bootstrap configuration file) contains the key “include-system-site-packages” set to anything other than “false” (case-insensitive), the system-level prefixes will still also be searched for site-packages; otherwise they won't.

一个路径配置文件是具有 *name*.pth 命名格式的文件，并且存在上面提到的四个目录之一中；它的内容是要添加到 sys.path 中的额外项目（每行一个）。不存在的项目不会添加到 sys.path，并且不会检查项目指向的是目录还是文件。项目不会被添加到 sys.path 超过一次。空行和由 # 起始的行会被跳过。以 import 开始的行（跟着空格或 TAB）会被执行。

例如，假设 sys.prefix 和 sys.exec_prefix 已经被设置为 /usr/local。Python X.Y 的库之后被安装为 /usr/local/lib/pythonX.Y。假设有一个拥有三个孙目录 foo, bar 和 spam 的子目录 /usr/local/lib/pythonX.Y/site-packages，并且有两个路径配置文件 foot.pth 和 bar.pth。假定 foo.pth 内容如下:

```
# foo package configuration

foo
bar
bletch
```

并且 bar.pth 包含:

```
# bar package configuration

bar
```

则下面特定版目录将以如下顺序被添加到 sys.path。

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

请注意 bletch 已被省略因为它并不存在；bar 目前在 foo 目录之前因为 bar.pth 按字母顺序排在 foo.pth 之前；而 spam 已被省略因为它在两个路径配置文件中都未被提及。

After these path manipulations, an attempt is made to import a module named sitecustomize, which can perform arbitrary site-specific customizations. It is typically created by a system administrator in the site-packages directory. If this import fails with an *ImportError* exception, it is silently ignored. If Python is started without output streams available, as with pythonw.exe on Windows (which is used by default to start IDLE), attempted output from sitecustomize is ignored. Any exception other than *ImportError* causes a silent and perhaps mysterious failure of the process.

After this, an attempt is made to import a module named usercustomize, which can perform arbitrary user-specific customizations, if *ENABLE_USER_SITE* is true. This file is intended to be created in the user site-packages directory (see below), which is part of sys.path unless disabled by -s. An *ImportError* will be silently ignored.

请注意对于某些非 Unix 系统来说，sys.prefix 和 sys.exec_prefix 均为空值，并且路径操作会被跳过；但是仍然会尝试导入 sitecustomize 和 usercustomize。

29.13.1 Readline（类库）配置

在支持*readline* 的系统上，这个模块也将导入并配置*rlcompleter* 模块，如果 Python 是以 交互模式启动并且不带 -S 选项的话。默认的行为是启用 tab 键补全并使用 ~/.python_history 作为历史存档文件。要禁用它，请删除（或重载）你的 sitecustomize 或 usercustomize 模块或 PYTHONSTARTUP 文件中的*sys.__interactivehook__* 属性。

在 3.4 版更改: rlcompleter 和 history 会被自动激活。

29.13.2 模块内容

site.**PREFIXES**
: site-packages 目录的前缀列表。

site.**ENABLE_USER_SITE**
: 显示用户 site-packages 目录状态的旗标。True 意味着它被启用并被添加到 sys.path。False 意味着它按照用户请求被禁用 (通过 -s 或 PYTHONNOUSERSITE)。None 意味着它因安全理由（user 或 group id 和 effective id 之间不匹配）或是被管理员所禁用。

site.**USER_SITE**
: 正在运行的 Python 的用户级 site-packages 的路径。它可以为 None，如果*getusersitepackages()*

尚未被调用的话。默认值在 UNIX 和非框架 Mac OS X 编译版上为 ~/.local/lib/pythonX.Y/site-packages，在 Mac 框架编译版上为 ~/Library/Python/X.Y/lib/python/site-packages，而在 Windows 上则为 %APPDATA%\Python\PythonXY\site-packages。此目录属于站点目录，这意味着其中的 .pth 文件将会被处理。

site.**USER_BASE**
用户级 site-packages 的基准目录的路径。它可以为 None，如果*getuserbase()* 尚未被调用的话。默认值在 UNIX 和 Mac OS X 非框架编译版上为 ~/.local，在 Mac 框架编译版上为 ~/Library/Python/X.Y，而在 Windows 上则为 %APPDATA%\Python。这个值会被 Distutils 用来计算脚本、数据文件和 Python 模块等的安装目录。对于 用户安装规范。另请参阅 PYTHONUSERBASE。

site.**main**()
将所有的标准站点专属目录添加到模块搜索路径。这个函数会在导入此模块时被自动调用，除非 Python 解释器启动时附带了 -S 旗标。

在 3.3 版更改: 这个函数使用无条件调用。

site.**addsitedir**(*sitedir*, *known_paths=None*)
将一个目录添加到 sys.path 并处理其 .pth 文件。通常被用于 sitecustomize 或 usercustomize (见下文)。

site.**getsitepackages**()
返回包含所有全局 site-packages 目录的列表。

3.2 新版功能.

site.**getuserbase**()
返回用户基准目录的路径*USER_BASE*。如果它尚未被初始化，则此函数还将参照 PYTHONUSERBASE 来设置它。

3.2 新版功能.

site.**getusersitepackages**()
Return the path of the user-specific site-packages directory, *USER_SITE*. If it is not initialized yet, this function will also set it, respecting PYTHONNOUSERSITE and *USER_BASE*.

3.2 新版功能.

site 模块还提供了一个从命令行获取用户目录的方式:

```
$ python3 -m site --user-site
/home/user/.local/lib/python3.3/site-packages
```

如果它被不带参数地调用，它将在标准输出打印*sys.path* 的内容，再打印*USER_BASE* 的值以及该目录是否存在，然后打印*USER_SITE* 的相应信息，最后打印*ENABLE_USER_SITE* 的值。

--user-base
输出用户基本的路径。

--user-site
将路径输出到用户 site-packages 目录。

如果同时给出了两个选项，则将打印用户基准目录和用户站点信息（总是按此顺序），并以*os.pathsep* 分隔。

如果给出了其中一个选项，脚本将退出并返回以下值中的一个：如果用户级 site-packages 目录被启用则为 0，如果它被用户禁用则为 1，如果它因安全理由或被管理员禁用则为 2，如果发生错误则为大于 2 的值。

参见:

PEP 370 –分用户的 site-packages 目录

29.14 fpectl —Floating point exception control

注解: The *fpectl* module is not built by default, and its usage is discouraged and may be dangerous except in the hands of experts. See also the section *Limitations and other considerations* on limitations for more details.

Most computers carry out floating point operations in conformance with the so-called IEEE-754 standard. On any real computer, some floating point operations produce results that cannot be expressed as a normal floating point value. For example, try

```
>>> import math
>>> math.exp(1000)
inf
>>> math.exp(1000) / math.exp(1000)
nan
```

(The example above will work on many platforms. DEC Alpha may be one exception.) "Inf" is a special, non-numeric value in IEEE-754 that stands for "infinity", and "nan" means "not a number." Note that, other than the non-numeric results, nothing special happened when you asked Python to carry out those calculations. That is in fact the default behaviour prescribed in the IEEE-754 standard, and if it works for you, stop reading now.

In some circumstances, it would be better to raise an exception and stop processing at the point where the faulty operation was attempted. The *fpectl* module is for use in that situation. It provides control over floating point units from several hardware manufacturers, allowing the user to turn on the generation of SIGFPE whenever any of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid Operation occurs. In tandem with a pair of wrapper macros that are inserted into the C code comprising your python system, SIGFPE is trapped and converted into the Python *FloatingPointError* exception.

The *fpectl* module defines the following functions and may raise the given exception:

fpectl.**turnon_sigfpe**()
Turn on the generation of SIGFPE, and set up an appropriate signal handler.

fpectl.**turnoff_sigfpe**()
Reset default handling of floating point exceptions.

exception fpectl.**FloatingPointError**
After *turnon_sigfpe()* has been executed, a floating point operation that raises one of the IEEE-754 exceptions Division by Zero, Overflow, or Invalid operation will in turn raise this standard Python exception.

29.14.1 Example

The following example demonstrates how to start up and test operation of the *fpectl* module.

```
>>> import fpectl
>>> import fpetest
>>> fpectl.turnon_sigfpe()
>>> fpetest.test()
overflow        PASS
FloatingPointError: Overflow

div by 0        PASS
FloatingPointError: Division by zero
  [ more output from test elided ]
```

(下页继续)

(续上页)

```
>>> import math
>>> math.exp(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: in math_1
```

29.14.2 Limitations and other considerations

Setting up a given processor to trap IEEE-754 floating point errors currently requires custom code on a per-architecture basis. You may have to modify *fpectl* to control your particular hardware.

Conversion of an IEEE-754 exception to a Python exception requires that the wrapper macros `PyFPE_START_PROTECT` and `PyFPE_END_PROTECT` be inserted into your code in an appropriate fashion. Python itself has been modified to support the *fpectl* module, but many other codes of interest to numerical analysts have not.

The *fpectl* module is not thread-safe.

参见:

Some files in the source distribution may be interesting in learning more about how this module operates. The include file `Include/pyfpe.h` discusses the implementation of this module at some length. `Modules/fpetestmodule.c` gives several examples of use. Many additional examples can be found in `Objects/floatobject.c`.

CHAPTER 30

自定义 Python 解释器

本章中描述的模块允许编写类似于 Python 的交互式解释器的接口。如果你想要一个支持附加一些特殊功能到 Python 语言的 Python 解释器，你应该看看 *code* 模块。（*codeop* 模块是低层级的，用于支持编译可能不完整的 Python 代码块。）

本章描述的完整模块列表如下：

30.1 code —解释器基础类

源代码： Lib/code.py

code 模块提供了在 Python 中实现 read-eval-print 循环的功能。它包含两个类和一些快捷功能，可用于构建提供交互式解释器的应用程序。

class code.**InteractiveInterpreter**(*locals=None*)

这个类处理解析器和解释器状态（用户命名空间的）；它不处理缓冲器、终端提示区或着输入文件名（文件名总是显示地传递）。可选的 *locals* 参数指定一个字典，字典里面包含将在此类执行的代码；它默认创建新的字典，其键 '__name__' 设置为 '__console__'，键 '__doc__' 设置为 None 。

class code.**InteractiveConsole**(*locals=None*, *filename="<console>"*)

尽可能模拟交互式 Python 解释器的行为。此类建立在 *InteractiveInterpreter* 的基础上，使用熟悉的 sys.ps1 和 sys.ps2 作为输入提示符，并有输入缓冲。

code.**interact**(*banner=None*, *readfunc=None*, *local=None*, *exitmsg=None*)

运行一个 read-eval-print 循环的便捷函数。这会创建一个新的 *InteractiveConsole* 实例。如果提供了 *readfunc*，会设置为 *InteractiveConsole.raw_input()* 方法。如果提供了 *local*，则将其传递给 *InteractiveConsole* 的构造函数，以用作解释器循环的默认命名空间。然后，如果提供了 *banner* 和 *exitmsg*，实例的 *interact()* 方法会以此为标题和退出消息。控制台对象在使用后将被丢弃。

在 3.6 版更改: 加入 *exitmsg* 参数。

code.**compile_command**(*source*, *filename="<input>"*, *symbol="single"*)

这个函数主要用来模拟 Python 解释器的主循环（即 read-eval-print 循环）。难点的部分是当用户输入不

完整命令时，判断能否通过之后的输入来完成（要么成为完整的命令，要么语法错误）。该函数 几乎和实际的解释器主循环的判断是相同的。

source is the source string; *filename* is the optional filename from which source was read, defaulting to `'<input>'`; and *symbol* is the optional grammar start symbol, which should be either `'single'` (the default) or `'eval'`.

如果命令完整且有效则返回一个代码对象(等价于 `compile(source, filename, symbol)`)；如果命令不完整则返回 `None`；如果命令完整但包含语法错误则会引发*SyntaxError* 或*OverflowError* 而如果命令包含无效字面值则将引发*ValueError*。

30.1.1 交互解释器对象

`InteractiveInterpreter.`**`runsource`**(*source*, *filename="<input>"*, *symbol="single"*)

Compile and run some source in the interpreter. Arguments are the same as for *compile_command()*; the default for *filename* is `'<input>'`, and for *symbol* is `'single'`. One several things can happen:

- 输入不正确；*compile_command()* 引发了一个异常 (*SyntaxError* 或*OverflowError*)。将通过调用*showsyntaxerror()* 方法打印语法回溯信息。*runsource()* 返回 `False`。
- 输入不完整，需要更多输入；函数*compile_command()* 返回 `None` 。方法*runsource()* 返回 `True` 。
- 输入完整；*compile_command()* 返回了一个代码对象。将通过调用*runcode()* 执行代码（该方法也会处理运行时异常，*SystemExit* 除外)。*runsource()* 返回 `False`。

该返回值用于决定使用 `sys.ps1` 还是 `sys.ps2` 来作为下一行的输入提示符。

`InteractiveInterpreter.`**`runcode`**(*code*)

执行一个代码对象。当发生异常时，调用*showtraceback()* 来显示回溯。除*SystemExit* （允许传播）以外的所有异常都会被捕获。

有关*KeyboardInterrupt* 的说明，该异常可能发生于此代码的其他位置，并且并不总能被捕获。调用者应当准备好处理它。

`InteractiveInterpreter.`**`showsyntaxerror`**(*filename=None*)

显示刚发生的语法错误。这不会显示堆栈回溯因为语法错误并无此种信息。如果给出了 *filename*，它会被放入异常来替代 Python 解析器所提供的默认文件名，因为它在从一个字符串读取时总是会使用 `'<string>'`。输出将由*write()* 方法来写入。

`InteractiveInterpreter.`**`showtraceback`**()

显示刚发生的异常。我们移除了第一个堆栈条目因为它从属于解释器对象的实现。输出将由*write()* 方法来写入。

在 3.5 版更改: 将显示完整的链式回溯，而不只是主回溯。

`InteractiveInterpreter.`**`write`**(*data*)

将一个字符串写入到标准错误流 (`sys.stderr`)。所有派生类都应重载此方法以提供必要的正确输出处理。

30.1.2 交互式控制台对象

InteractiveConsole 类是*InteractiveInterpreter* 的子类，因此它提供了解释器对象的所有方法，还有以下的额外方法。

InteractiveConsole.**interact**(*banner=None*, *exitmsg=None*)
近似地模拟交互式 Python 终端。可选的 *banner* 参数指定要在第一次交互前打印的条幅；默认情况下会类似于标准 Python 解释器所打印的内容，并附上外加圆括号的终端对象类名（这样就不会与真正的解释器混淆——因为确实太像了!）

可选的 *exitmsg* 参数指定要在退出时打印的退出消息。传入空字符串可以屏蔽退出消息。如果 *exitmsg* 未给出或为 None，则将打印默认消息。

在 3.4 版更改: 要禁止打印任何标志，请传递一个空字符串。

在 3.6 版更改: 退出时打印相关消息。

InteractiveConsole.**push**(*line*)
将一行源文本推入解释器。行内容不应带有末尾换行符；它可以有内部换行符。行内容会被添加到一个缓冲区并且会调用解释器的 runsource() 方法，附带缓冲区内容的拼接结果作为源文本。如果显示命令已执行或不合法，缓冲区将被重置；否则，则命令尚未结束，缓冲区将在添加行后保持原样。如果要求更多输入则返回值为 True，如果行已按某种方式被处理则返回值为 False (这与 runsource() 相同)。

InteractiveConsole.**resetbuffer**()
从输入缓冲区中删除所有未处理的内容。

InteractiveConsole.**raw_input**(*prompt=""*)
输出提示并读取一行。返回的行不包含末尾的换行符。当用户输入 EOF 键序列时，会引发*EOFError* 异常。默认实现是从 sys.stdin 读取；子类可以用其他实现代替。

30.2 codeop —编译 Python 代码

源代码： Lib/codeop.py

---

codeop 模块提供了可以模拟 Python 读取-执行-打印循环的实用程序，就像在*code* 模块中一样。因此，您可能不希望直接使用该模块；如果你想在程序中包含这样一个循环，你可能需要使用*code* 模块。

这个任务有两个部分:

1. 能够判断一行输入是否完成了一个 Python 语句：简而言之，告诉我们是否要打印‘>>>’或‘...’。
2. 记住用户已输入了哪些 future 语句，这样后续的输入可以在这些语句被启用的状态下被编译。

codeop 模块提供了分别以及同时执行这两个部分的方式。

只执行前一部分:

codeop.**compile_command**(*source*, *filename="<input>"*, *symbol="single"*)
尝试编译 *source*，这应当是一个 Python 代码字符串，并且在 *source* 是有效的 Python 代码时返回一个代码对象。在此情况下，代码对象的 filename 属性将为 *filename*，其默认值为 '<input>'。如果 *source* 不是有效的 Python 代码而是有效的 Python 代码的一个前缀时将返回 None。

如果 *source* 存在问题，将引发异常。如果存在无效的 Python 语法将引发*SyntaxError*，而如果存在无效的字面值则将引发*OverflowError* 或*ValueError*。

The *symbol* argument determines whether *source* is compiled as a statement ('single', the default) or as an *expression* ('eval'). Any other value will cause *ValueError* to be raised.

注解: 解析器有可能（但很不常见）会在到达源码结尾之前停止解析并成功输出结果；在这种情况下，末尾的符号可能会被忽略而不是引发错误。例如，一个反斜杠加两个换行符之后可以跟随任何无意义的符号。一旦解析器 API 得到改进将修正这个问题。

class codeop.**Compile**
这个类的实例具有 __call__() 方法，其签名与内置函数*compile()* 相似，区别在于如果该实例编译了包含*__future__* 语句的程序文本，则实例会‘记住’并使用已生效的语句编译所有后续程序文本。

class codeop.**CommandCompiler**
这个类的实例具有 __call__() 方法，其签名与*compile_command()* 相似；区别在于如果该实例编译了包含 __future__ 语句的程序文本，则实例会‘记住’并使用已生效的语句编译编译所有后续程序文本。

CHAPTER 31

导入模块

本章中介绍的模块提供了导入其他 Python 模块和挂钩以自定义导入过程的新方法。

本章描述的完整模块列表如下：

31.1 zipimport —从 Zip 存档中导入模块

此模块添加了从 ZIP 格式档案中导入 Python 模块（*.py ， *.pyc ）和包的能力。通常不需要明确地使用*zipimport* 模块，内置的 import 机制会自动将此模块用于 ZIP 档案路径的*sys.path* 项目上。

通常，*sys.path* 是字符串的目录名称列表。此模块同样允许*sys.path* 的一项成为命名 ZIP 文件档案的字符串。ZIP 档案可以容纳子目录结构去支持包的导入，并且可以将归档文件中的路径指定为仅从子目录导入。比如说，路径 example.zip/lib/ 将只会从档案中的 lib/ 子目录导入。

任何文件都可以存在于 ZIP 档案之中，但是只有 .py 和 .pyc 文件是能够导入的。不允许导入 ZIP 中的动态模组（.pyd ， .so ）。请注意，如果档案中只包含 .py 文件，Python 不会尝试通过添加对应的 .pyc 文件修改档案，意思是如果 ZIP 档案不包含 .pyc 文件，导入或许会变慢。

目前不支持带有档案注释的 ZIP 归档。

参见：

PKZIP Application Note Phil Katz 编写的 ZIP 文件格式文档，此格式和使用的算法的创建者。

PEP 273 - 从 ZIP 压缩包导入模块 由 James C. Ahlstrom 编写，他还提供了一个具体实现。Python 2.3 遵循 PEP 273 的规范，但使用了一个由 van Rossum 本人所编写的实现，该实现使用了 PEP 302 所描述的导入钩子。

PEP 302 - 新导入钩 PEP 添加导入钩来有助于模块运作。

此模块定义了一个异常：

exception zipimport.**ZipImportError**
异常由 zipimporter 对象引发。这是*ImportError* 的子类，因此，也可以捕获为*ImportError* 。

31.1.1 zipimporter 对象

zipimporter 是用于导入 ZIP 文件的类。

class zipimport.**zipimporter**(*archivepath*)
创建新的 zipimporter 实例。*archivepath* 必须是指向 ZIP 文件的路径，或者 ZIP 文件中的特定路径。例如，`foo/bar.zip/lib` 的 *archivepath* 将在 ZIP 文件 `foo/bar.zip` 中的 `lib` 目录中查找模块。

如果 *archivepath* 没有指向一个有效的 ZIP 档案，引发*ZipImportError* 。

find_module(*fullname*[, *path*])
搜索由 *fullname* 指定的模块。*fullname* 必须是完全合格的（含加点作分割的拓展名）模块名。它返回 zipimporter 实例本身如果模块被找到，或者返回*None* 如果没找到指定模块。可选的 *path* 被忽略，这是为了与导入器协议兼容。

get_code(*fullname*)
返回指定模块的代码对象。如果不能找到模块会引发*ZipImportError* 错误。

get_data(*pathname*)
返回与 *pathname* 相关联的数据。如果不能找到文件则引发*OSError* 错误。

在 3.3 版更改: *IOError* 代替*OSError* 被引发。

get_filename(*fullname*)
如果导入了指定的模块 `__file__` ，则返回为该模块设置的值。如果未找到模块则引发*ZipImportError* 错误。

3.1 新版功能.

get_source(*fullname*)
返回指定模块的源代码。如果没有找到模块则引发*ZipImportError* ，如果档案包含模块但是没有源代码，返回*None* 。

is_package(*fullname*)
如果由 *fullname* 指定的模块是一个包则返回 `True` 。如果不能找到模块则引发*ZipImportError* 错误。

load_module(*fullname*)
加载由 *fullname* 指定的模块。*fullname* 必须是完全合格的（含加点作为分割的拓展名）模块名。它返回已加载模块，或者当找不到模块时引发*ZipImportError* 错误。

archive
导入器关联的 ZIP 文件的文件名，不含可能的子路径。

prefix
ZIP 文件中搜索的模块的子路径。这是一个指向 ZIP 文件根目录的 zipimporter 对象的空字符串。

当与斜杠结合使用时，*archive* 和*prefix* 属性等价于赋予*zipimporter* 构造器的原始 *archivepath* 参数。

31.1.2 例子

这是一个从 ZIP 档案中导入模块的例子 - 请注意*zipimport* 不需要明确地使用。

```
$ unzip -l example.zip
Archive:  example.zip
  Length      Date    Time    Name
 --------    ----    ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
```

(下页继续)

(续上页)

```
     8467                    1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

31.2 pkgutil —包扩展模块工具

源代码: Lib/pkgutil.py

该模块为导入系统提供了工具，尤其是在包支持方面。

class pkgutil.**ModuleInfo**(*module_finder*, *name*, *ispkg*)
一个包含模块信息的简短摘要的命名元组。

3.6 新版功能.

pkgutil.**extend_path**(*path*, *name*)
Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package's __init__.py:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

This will add to the package's __path__ all subdirectories of directories on sys.path named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for *.pkg files beginning where * matches the *name* argument. This feature is similar to *.pth files (see the *site* module for more information), except that it doesn't special-case lines starting with import. A *.pkg file is trusted at face value: apart from checking for duplicates, all entries found in a *.pkg file are added to the path, regardless of whether they exist on the filesystem. (This is a feature.)

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that *sys.path* is a sequence. Items of *sys.path* that are not strings referring to existing directories are ignored. Unicode items on *sys.path* that cause errors when used as filenames may cause this function to raise an exception (in line with *os.path.isdir()* behavior).

class pkgutil.**ImpImporter**(*dirname=None*)
PEP 302 Finder that wraps Python's "classic" import algorithm.

If *dirname* is a string, a **PEP 302** finder is created that searches that directory. If *dirname* is None, a **PEP 302** finder is created that searches the current *sys.path*, plus any modules that are frozen or built-in.

Note that *ImpImporter* does not currently support being used by placement on *sys.meta_path*.

3.3 版后已移除: This emulation is no longer needed, as the standard import mechanism is now fully PEP 302 compliant and available in *importlib*.

class pkgutil.**ImpLoader**(*fullname*, *file*, *filename*, *etc*)
Loader that wraps Python's "classic" import algorithm.

3.3 版后已移除: This emulation is no longer needed, as the standard import mechanism is now fully PEP 302 compliant and available in *importlib*.

pkgutil.**find_loader**(*fullname*)
Retrieve a module *loader* for the given *fullname*.

This is a backwards compatibility wrapper around *importlib.util.find_spec()* that converts most failures to *ImportError* and only returns the loader rather than the full ModuleSpec.

在 3.3 版更改: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

在 3.4 版更改: Updated to be based on **PEP 451**

pkgutil.**get_importer**(*path_item*)
Retrieve a *finder* for the given *path_item*.

The returned finder is cached in *sys.path_importer_cache* if it was newly created by a path hook.

The cache (or part of it) can be cleared manually if a rescan of *sys.path_hooks* is necessary.

在 3.3 版更改: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

pkgutil.**get_loader**(*module_or_name*)
Get a *loader* object for *module_or_name*.

If the module or package is accessible via the normal import mechanism, a wrapper around the relevant part of that machinery is returned. Returns None if the module cannot be found or imported. If the named module is not already imported, its containing package (if any) is imported, in order to establish the package __path__.

在 3.3 版更改: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

在 3.4 版更改: Updated to be based on **PEP 451**

pkgutil.**iter_importers**(*fullname=''*)
Yield *finder* objects for the given module name.

If fullname contains a '.', the finders will be for the package containing fullname, otherwise they will be all registered top level finders (i.e. those on both sys.meta_path and sys.path_hooks).

If the named module is in a package, that package is imported as a side effect of invoking this function.

If no module name is specified, all top level finders are produced.

在 3.3 版更改: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

pkgutil.**iter_modules**(*path=None*, *prefix=''*)
Yields *ModuleInfo* for all submodules on *path*, or, if *path* is None, all top-level modules on sys.path.

path should be either None or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

注解: Only works for a *finder* which defines an iter_modules() method. This interface is non-standard, so the module also provides implementations for *importlib.machinery.FileFinder* and *zipimport.zipimporter*.

在 3.3 版更改: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

pkgutil.**walk_packages**(*path=None*, *prefix=''*, *onerror=None*)

Yields *ModuleInfo* for all modules recursively on *path*, or, if *path* is None, all accessible modules.

path should be either None or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note that this function must import all *packages* (*not* all modules!) on the given *path*, in order to access the __path__ attribute to find submodules.

onerror is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no *onerror* function is supplied, *ImportError*s are caught and ignored, while all other exceptions are propagated, terminating the search.

示例:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

注解: Only works for a *finder* which defines an iter_modules() method. This interface is non-standard, so the module also provides implementations for *importlib.machinery.FileFinder* and *zipimport.zipimporter*.

在 3.3 版更改: Updated to be based directly on *importlib* rather than relying on the package internal PEP 302 import emulation.

pkgutil.**get_data**(*package*, *resource*)

从包中获取一个资源。

This is a wrapper for the *loader* *get_data* API. The *package* argument should be the name of a package, in standard module format (foo.bar). The *resource* argument should be in the form of a relative filename, using / as the path separator. The parent directory name .. is not allowed, and nor is a rooted name (starting with a /).

The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

If the package cannot be located or loaded, or it uses a *loader* which does not support *get_data*, then None is returned. In particular, the *loader* for *namespace packages* does not support *get_data*.

31.3 modulefinder —查找脚本使用的模块

源码: Lib/modulefinder.py

该模块提供了一个*ModuleFinder* 类，可用于确定脚本导入的模块集。modulefinder.py 也可以作为脚本运行，给出 Python 脚本的文件名作为参数，之后将打印导入模块的报告。

modulefinder.**AddPackagePath**(*pkg_name*, *path*)

记录名为 *pkg_name* 的包可以在指定的 *path* 中找到。

modulefinder.**ReplacePackage**(*oldname*, *newname*)
允许指定名为 *oldname* 的模块实际上是名为 *newname* 的包。

class modulefinder.**ModuleFinder**(*path=None*, *debug=0*, *excludes=[]*, *replace_paths=[]*)
该类提供*run_script()* 和*report()* 方法，用于确定脚本导入的模块集。*path* 可以是搜索模块的目录列表；如果没有指定，则使用 sys.path 。*debug* 设置调试级别；更高的值使类打印调试消息，关于它正在做什么。*excludes* 是要从分析中排除的模块名称列表。*replace_paths* 是将在模块路径中替换的 (oldpath, newpath) 元组的列表。

report()
将报告打印到标准输出，列出脚本导入的模块及其路径，以及缺少或似乎缺失的模块。

run_script(*pathname*)
分析 *pathname* 文件的内容，该文件必须包含 Python 代码。

modules
一个将模块名称映射到模块的字典。请参阅*ModuleFinder 的示例用法*。

31.3.1 ModuleFinder 的示例用法

稍后将分析的脚本（bacon.py）：

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

将输出 bacon.py 报告的脚本:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(','.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

输出样例（可能因架构而异）：

```
Loaded modules:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
sre_compile:  isstring,_sre,_optimize_unicode
_sre:
```

(下页继续)

(续上页)

```
sre_constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
sre_parse:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
------------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

31.4 runpy —Locating and executing Python modules

Source code: Lib/runpy.py

The *runpy* module is used to locate and run Python modules without importing them first. Its main use is to implement the `-m` command line switch that allows scripts to be located using the Python module namespace rather than the filesystem.

Note that this is *not* a sandbox module - all code is executed in the current process, and any side effects (such as cached imports of other modules) will remain in place after the functions have returned.

Furthermore, any functions and classes defined by the executed code are not guaranteed to work correctly after a *runpy* function has returned. If that limitation is not acceptable for a given use case, *importlib* is likely to be a more suitable choice than this module.

The *runpy* module provides two functions:

runpy.**run_module**(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

Execute the code of the specified module and return the resulting module globals dictionary. The module's code is first located using the standard import mechanism (refer to **PEP 302** for details) and then executed in a fresh module namespace.

The *mod_name* argument should be an absolute module name. If the module name refers to a package rather than a normal module, then that package is imported and the `__main__` submodule within that package is then executed and the resulting module globals dictionary returned.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by *run_module()*.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

`__name__` is set to *run_name* if this optional argument is not *None*, to `mod_name + '.__main__'` if the named module is a package and to the *mod_name* argument otherwise.

`__spec__` will be set appropriately for the *actually* imported module (that is, `__spec__.name` will always be *mod_name* or `mod_name + '.__main__'`, never *run_name*).

`__file__`, `__cached__`, `__loader__` and `__package__` are set as normal based on the module spec.

If the argument *alter_sys* is supplied and evaluates to `True`, then `sys.argv[0]` is updated with the value of `__file__` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. Both `sys.argv[0]` and `sys.modules[__name__]` are restored to their original values before the function returns.

Note that this manipulation of `sys` is not thread-safe. Other threads may see the partially initialised module, as well as the altered list of arguments. It is recommended that the `sys` module be left alone when invoking this function from threaded code.

参见:

The `-m` option offering equivalent functionality from the command line.

在 3.1 版更改: Added ability to execute packages by looking for a `__main__` submodule.

在 3.2 版更改: Added `__cached__` global variable (see **PEP 3147**).

在 3.4 版更改: Updated to take advantage of the module spec feature added by **PEP 451**. This allows `__cached__` to be set correctly for modules run this way, as well as ensuring the real module name is always accessible as `__spec__.name`.

`runpy.`**`run_path`**(*file_path*, *init_globals=None*, *run_name=None*)

Execute the code at the named filesystem location and return the resulting module globals dictionary. As with a script name supplied to the CPython command line, the supplied path may refer to a Python source file, a compiled bytecode file or a valid sys.path entry containing a `__main__` module (e.g. a zipfile containing a top-level `__main__.py` file).

For a simple script, the specified code is simply executed in a fresh module namespace. For a valid sys.path entry (typically a zipfile or directory), the entry is first added to the beginning of `sys.path`. The function then looks for and executes a `__main__` module using the updated path. Note that there is no special protection against invoking an existing `__main__` entry located elsewhere on `sys.path` if there is no such module at the specified location.

The optional dictionary argument *init_globals* may be used to pre-populate the module's globals dictionary before the code is executed. The supplied dictionary will not be modified. If any of the special global variables below are defined in the supplied dictionary, those definitions are overridden by `run_path()`.

The special global variables `__name__`, `__spec__`, `__file__`, `__cached__`, `__loader__` and `__package__` are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

`__name__` is set to *run_name* if this optional argument is not `None` and to `'<run_path>'` otherwise.

If the supplied path directly references a script file (whether as source or as precompiled byte code), then `__file__` will be set to the supplied path, and `__spec__`, `__cached__`, `__loader__` and `__package__` will all be set to `None`.

If the supplied path is a reference to a valid sys.path entry, then `__spec__` will be set appropriately for the imported `__main__` module (that is, `__spec__.name` will always be `__main__`). `__file__`, `__cached__`, `__loader__` and `__package__` will be set as normal based on the module spec.

A number of alterations are also made to the `sys` module. Firstly, `sys.path` may be altered as described above. `sys.argv[0]` is updated with the value of `file_path` and `sys.modules[__name__]` is updated with a temporary module object for the module being executed. All modifications to items in `sys` are reverted before the function returns.

Note that, unlike `run_module()`, the alterations made to `sys` are not optional in this function as these adjustments are essential to allowing the execution of sys.path entries. As the thread-safety limitations still apply, use of this function in threaded code should be either serialised with the import lock or delegated to a separate process.

参见:

using-on-interface-options for equivalent functionality on the command line (`python path/to/script`).

3.2 新版功能.

在 3.4 版更改: Updated to take advantage of the module spec feature added by **PEP 451**. This allows `__cached__` to be set correctly in the case where `__main__` is imported from a valid sys.path entry rather than being executed directly.

参见:

PEP 338 –**将模块作为脚本执行** PEP 由 Nick Coghlan 撰写并实现。

PEP 366 –**Main module explicit relative imports** PEP 由 Nick Coghlan 撰写并实现。

PEP 451 –**A ModuleSpec Type for the Import System** PEP written and implemented by Eric Snow

using-on-general - CPython command line details

The *importlib.import_module()* function

31.5 importlib —The implementation of import

3.1 新版功能.

源代码 Lib/importlib/__init__.py

31.5.1 概述

The purpose of the *importlib* package is two-fold. One is to provide the implementation of the `import` statement (and thus, by extension, the *__import__()* function) in Python source code. This provides an implementation of `import` which is portable to any Python interpreter. This also provides an implementation which is easier to comprehend than one implemented in a programming language other than Python.

第二个目的是实现 `import` 的部分被公开在这个包中，使得用户更容易创建他们自己的自定义对象 (通常被称为*importer*) 来参与到导入过程中。

参见:

import `import` 语句的语言参考

包规格说明 包的初始规范。自从编写这个文档开始，一些语义已经发生改变了（比如基于*sys.modules* 中 `None` 的重定向）。

***__import__()* 函数** `import` 语句是这个函数的语法糖。

PEP 235 在忽略大小写的平台上进行导入

PEP 263 定义 Python 源代码编码

PEP 302 新导入钩子

PEP 328 导入：多行和绝对/相对

PEP 366 主模块显式相对导入

PEP 420 隐式命名空间包

PEP 451 导入系统的一个模块规范类型

PEP 488 消除 PYO 文件

PEP 489 多阶段扩展模块初始化

PEP 3120 使用 UTF-8 作为默认的源编码

PEP 3147 PYC 仓库目录

31.5.2 函数

importlib.**__import__**(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)
内置*__import__()* 函数的实现。

> **注解:** 程序式地导入模块应该使用*import_module()* 而不是这个函数。

importlib.**import_module**(*name*, *package=None*)
导入一个模块。参数 *name* 指定了以绝对或相对导入方式导入什么模块 (比如要么像这样 `pkg.mod` 或者这样 `..mod`)。如果参数 name 使用相对导入的方式来指定，那么那个参数 *package* 必须设置为那个包名，这个包名作为解析这个包名的锚点 (比如 `import_module('..mod', 'pkg.subpkg')` 将会导入 `pkg.mod`)。

import_module() 函数是一个对*importlib.__import__()* 进行简化的包装器。这意味着该函数的所有主义都来自于*importlib.__import__()*。这两个函数之间最重要的不同点在于*import_module()* 返回指定的包或模块 (例如 `pkg.mod`)，而*__import__()* 返回最高层级的包或模块 (例如 `pkg`)。

如果动态导入一个自从解释器开始执行以来被创建的模块（即创建了一个 Python 源代码文件），为了让导入系统知道这个新模块，可能需要调用*invalidate_caches()*。

在 3.3 版更改: 父包会被自动导入。

importlib.**find_loader**(*name*, *path=None*)
查找一个模块的加载器，可选择地在指定的 *path* 里面。如果这个模块是在*sys.modules*，那么返回 `sys.modules[name].__loader__` (除非这个加载器是 `None` 或者是没有被设置，在这样的情况下，会引起*ValueError* 异常)。否则使用*sys.meta_path* 的一次搜索就结束。如果未发现加载器，则返回 `None`。

点状的名称没有使得它父包或模块隐式地导入，因为它需要加载它们并且可能不需要。为了适当地导入一个子模块，需要导入子模块的所有父包并且使用正确的参数提供给 *path*。

3.3 新版功能.

在 3.4 版更改: 如果没有设置 `__loader__`，会引起*ValueError* 异常，就像属性设置为 `None` 的时候一样。

3.4 版后已移除: 使用*importlib.util.find_spec()* 来代替。

importlib.**invalidate_caches**()
使查找器存储在*sys.meta_path* 中的内部缓存无效。如果一个查找器实现了 `invalidate_caches()`，那么它会被调用来执行那个无效过程。如果创建/安装任何模块，同时正在运行的程序是为了保证所有的查找器知道新模块的存在，那么应该调用这个函数。

3.3 新版功能.

importlib.**reload**(*module*)
重新加载之前导入的 *module*。那个参数必须是一个模块对象，所以它之前必须已经成功导入了。这样做是有用的，如果使用外部编辑器编已经辑过了那个模块的源代码文件并且想在退出 Python 解释器之前试验这个新版本的模块。函数的返回值是那个模块对象（如果重新导入导致一个不同的对象放置在*sys.modules* 中，那么那个模块对象是有可能会不同）。

当执行*reload()* 的时候:

- Python 模块的代码会被重新编译并且那个模块级的代码被重新执行，通过重新使用一开始加载那个模块的*loader*，定义一个新的绑定在那个模块字典中的名称的对象集合。扩展模块的 “init“函数不会被调用第二次。
- 与 Python 中的所有的其它对象一样，旧的对象只有在它们的引用计数为 0 之后才会被回收。
- 模块命名空间中的名称重新指向任何新的或更改后的对象。
- 其他旧对象的引用（例如那个模块的外部名称）不会被重新绑定到引用的新对象的，并且如果有需要，必须在出现的每个命名空间中进行更新。

有一些其他注意事项：

当一个模块被重新加载的时候，它的字典（包含了那个模块的全区变量）会被保留。名称的重新定义会覆盖旧的定义，所以通常来说这不是问题。如果一个新模块没有定义在旧版本模块中定义的名称，则将保留旧版本中的定义。这一特性可用于作为那个模块的优点，如果它维护一个全局表或者对象的缓存——使用 `try` 语句，就可以测试表的存在并且跳过它的初始化，如果有需要的话：

```
try:
    cache
except NameError:
    cache = {}
```

重新加载内置的或者动态加载模块，通常来说不是很有用处。不推荐重新加载” *sys*，*__main__*，*builtins* 和其它关键模块。在很多例子中，扩展模块并不是设计为不止一次的初始化，并且当重新加载时，可能会以任意方式失败。

If a module imports objects from another module using `from …import …`, calling *reload()* for the other module does not redefine the objects imported from it —one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module.name*) instead.

如果一个模块创建一个类的实例，重新加载定义那个类的模块不影响那些实例的方法定义———它们继续使用旧类中的定义。对于子类来说同样是正确的。

3.4 新版功能.

31.5.3 `importlib.abc` ——关于导入的抽象基类

源代码： Lib/importlib/abc.py

The *importlib.abc* module contains all of the core abstract base classes used by `import`. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

ABC 类的层次结构：

```
object
 +-- Finder (deprecated)
 |    +-- MetaPathFinder
 |    +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class importlib.abc.**Finder**
代表*finder* 的一个抽象基类

3.3 版后已移除: 使用*MetaPathFinder* 或*PathEntryFinder* 来代替。

abstractmethod find_module(*fullname*, *path=None*)
为指定的模块查找*loader* 定义的抽象方法。本来是在 **PEP 302** 指定的，这个方法是在*sys.meta_path* 和基于路径的导入子系统中使用。

在 3.4 版更改: 当被调用的时候，返回 None 而不是引发*NotImplementedError*。

class importlib.abc.**MetaPathFinder**
代表*meta path finder* 的一个抽象基类。为了保持兼容性，这是*Finder* 的一个子类。

3.3 新版功能.

find_spec(*fullname*, *path*, *target=None*)
An abstract method for finding a *spec* for the specified module. If this is a top-level import, *path* will be None. Otherwise, this is a search for a subpackage or module and *path* will be the value of __path__ from the parent package. If a spec cannot be found, None is returned. When passed in, target is a module object that the finder may use to make a more educated guess about what spec to return.

3.4 新版功能.

find_module(*fullname*, *path*)
一个用于查找指定的模块中*loader* 的遗留方法。如果这是最高层级的导入，*path* 的值将会是 None。否则，这是一个查找子包或者模块的方法，并且 *path* 的值将会是来自父包的 __path__ 的值。如果未发现加载器，返回 None。

如果定义了*find_spec()* 方法，则提供了向后兼容的功能。

在 3.4 版更改: 当调用这个方法的时候返回 None 而不是引发*NotImplementedError*。可以使用*find_spec()* 来提供功能。

3.4 版后已移除: 使用*find_spec()* 来代替。

invalidate_caches()
当被调用的时候，一个可选的方法应该将查找器使用的任何内部缓存进行无效。将在*sys.meta_path* 上的所有查找器的缓存进行无效的时候，这个函数被*importlib.invalidate_caches()* 所使用。

在 3.4 版更改: 当方法被调用的时候，方法返回是 None 而不是 NotImplemented 。

class importlib.abc.**PathEntryFinder**
path entry finder 的一个抽象基类。尽管这个基类和*MetaPathFinder* 有一些相似之处，但是 PathEntryFinder 只在由 PathFinder 提供的基于路径导入子系统中使用。这个抽象类是*Finder* 的一个子类，仅仅是因为兼容性的原因。

3.3 新版功能.

find_spec(*fullname*, *target=None*)
An abstract method for finding a *spec* for the specified module. The finder will search for the module only within the *path entry* to which it is assigned. If a spec cannot be found, None is returned. When passed in, target is a module object that the finder may use to make a more educated guess about what spec to return.

3.4 新版功能.

find_loader(*fullname*)
一个用于在模块中查找一个*loader* 的遗留方法。返回一个 (loader, portion) 的 2 元组，portion 是一个贡献给命名空间包部分的文件系统位置的序列。加载器可能是 None，同时正在指定的 portion 表示的是贡献给命名空间包的文件系统位置。portion 可以使用一个空列表来表示加载器不是命名空间包的一部分。如果 loader 是 None 并且 portion 是一个空列表，那么命名空间包中无加载器或者文件系统位置可查找到（即在那个模块中未能找到任何东西）。

如果定义了*find_spec()*，则提供了向后兼容的功能。

在 3.4 版更改: 返回 (None, []) 而不是引发*NotImplementedError*。当可于提供相应的功能的时候，使用*find_spec()*。

3.4 版后已移除: 使用*find_spec()* 来代替。

find_module(*fullname*)

Finder.find_module`的 具 体 实 现， 该 方 法 等 价 于 ``self.find_loader(fullname)[0]`()。

3.4 版后已移除: 使用*find_spec()* 来代替。

invalidate_caches()

当被调用的时候，一个可选的方法应该将查找器使用的任何内部缓存进行无效。当将所有缓存的查找器的缓存进行无效的时候，该函数被 PathFinder.invalidate_caches() 使用。

class importlib.abc.**Loader**

loader 的抽象基类。关于一个加载器的实际定义请查看 **PEP 302**。

create_module(*spec*)

当导入一个模块的时候，一个返回将要使用的那个模块对象的方法。这个方法可能返回 None，这暗示着应该发生默认的模块创建语义。”

3.4 新版功能.

在 3.5 版更改: 从 Python 3.6 开始，当定义了*exec_module()* 的时候，这个方法将不会是可选的。

exec_module(*module*)

当一个模块被导入或重新加载时，一个抽象方法在它自己的命名空间中执行那个模块。当调用 exec_module() 的时候，那个模块应该已经被初始化了。当这个方法存在时，必须定义*create_module()* 。

3.4 新版功能.

在 3.6 版更改: *create_module()* 也必须被定义。

load_module(*fullname*)

用于加载一个模块的传统方法。如果这个模块不能被导入，将引起*ImportError* 异常，否则返回那个被加载的模块。

如果请求的模块已经存在*sys.modules*，应该使用并且重新加载那个模块。否则加载器应该是创建一个新的模块并且在任何家过程开始之前将这个新模块插入到*sys.modules* 中，来阻止递归导入。如果加载器插入了一个模块并且加载失败了，加载器必须从*sys.modules* 中将这个模块移除。在加载器开始执行之前，已经在*sys.modules* 中的模块应该被忽略 (查看*importlib.util.module_for_loader()*)。

加载器应该在模块上面设置几个属性。(要知道当重新加载一个模块的时候，那些属性某部分可以改变):

- **__name__** 模块的名字
- **__file__** 模块数据存储的路径 (不是为了内置的模块而设置)
- **__cached__** 被存储或应该被存储的模块的编译版本的路径 (当这个属性不恰当的时候不设置)。
- **__path__** 指定在一个包中搜索路径的一个字符串列表。这个属性不在模块上面进行设置。
- **__package__** 模块/包的父包。如果这个模块是最上层的，那么它是一个为空字符串的值。*importlib.util.module_for_loader()* 装饰器可以处理 __package__ 的细节。
- **__loader__** 用来加载那个模块的加载器。*importlib.util.module_for_loader()* 装饰器可以处理 __package__ 的细节。

当*exec_module()* 可用的时候，那么则提供了向后兼容的功能。

在 3.4 版更改: 当这个方法被调用的时候，触发*ImportError* 异常而不是*NotImplementedError*。当*exec_module()* 可用的时候，使用它的功能。

3.4 版后已移除: 加载模块推荐的使用的 API 是*exec_module()* (和*create_module()*)。加载器应该实现它而不是 load_module()。当 exec_module() 被实现的时候，导入机制关心的是 load_module() 所有其他的责任。

module_repr(*module*)
一个遗留方法，在实现时计算并返回给定模块的 repr，作为字符串。模块类型的默认 repr() 将根据需要使用此方法的结果。

3.3 新版功能.

在 3.4 版更改: 是可选的方法而不是一个抽象方法。

3.4 版后已移除: 现在导入机制会自动地关注这个方法。

class importlib.abc.**ResourceLoader**
一个*loader* 的抽象基类，它实现了可选的 **PEP 302** 协议用于从存储后端加载任意资源。

abstractmethod get_data(*path*)
一个用于返回位于 *path* 的字节数据的抽象方法。有一个允许存储任意数据的类文件存储后端的加载器能够实现这个抽象方法来直接访问这些被存储的数据。如果不能够找到 *path*，则会引发*OSError* 异常。*path* 被希望使用一个模块的 __file__ 属性或来自一个包的 __path__ 来构建。

在 3.4 版更改: 引发*OSError* 异常而不是*NotImplementedError* 异常。

class importlib.abc.**InspectLoader**
一个实现加载器检查模块可选的 **PEP 302** 协议的*loader* 的抽象基类。

get_code(*fullname*)
返回一个模块的代码对象，或如果模块没有一个代码对象（例如，对于内置的模块来说，这会是这种情况），则为 None。如果加载器不能找到请求的模块，则引发*ImportError* 异常。

注解: 当这个方法有一个默认的实现的时候，出于性能方面的考虑，如果有可能的话，建议覆盖它。

在 3.4 版更改: 不再抽象并且提供一个具体的实现。

abstractmethod get_source(*fullname*)
一个返回模块源的抽象方法。使用*universal newlines* 作为文本字符串被返回，将所有可识别行分割符翻译成 '\n' 字符。如果没有可用的源（例如，一个内置模块），则返回 None。如果加载器不能找到指定的模块，则引发*ImportError* 异常。

在 3.4 版更改: 引发*ImportError* 而不是*NotImplementedError*。

is_package(*fullname*)
一个抽象方法，如果这个模块是一个包则返回真值，否则返回假值。如果*loader* 不能找到这个模块，则引发*ImportError*。

在 3.4 版更改: 引发*ImportError* 而不是*NotImplementedError*。

static source_to_code(*data*, *path='<string>'*)
创建一个来自 Python 源码的代码对象。

参数 *data* 可以是任意*compile()* 函数支持的类型（例如字符串或字节串）。参数 *path* 应该是源代码来源的路径，这可能是一个抽象概念（例如位于一个 zip 文件中）。

在有后续代码对象的情况下，可以在一个模块中通过运行 “exec(code, module.__dict__)“来执行它。

3.4 新版功能.

在 3.5 版更改: 使得这个方法变成静态的。

exec_module(*module*)
Loader.exec_module() 的实现。

3.4 新版功能.

load_module(*fullname*)
Loader.load_module() 的实现。

3.4 版后已移除: 使用*exec_module()* 来代替。

class importlib.abc.**ExecutionLoader**
一个继承自*InspectLoader* 的抽象基类，当被实现时，帮助一个模块作为脚本来执行。这个抽象基类表示可选的 **PEP 302** 协议。

abstractmethod get_filename(*fullname*)
一个用来为指定模块返回 __file__ 的值的抽象方法。如果无路径可用，则引发*ImportError*。

如果源代码可用，那么这个方法返回源文件的路径，不管是否是用来加载模块的字节码。

在 3.4 版更改: 引发*ImportError* 而不是*NotImplementedError*。

class importlib.abc.**FileLoader**(*fullname*, *path*)
一个继承自*ResourceLoader* 和*ExecutionLoader*，提供*ResourceLoader.get_data()* 和*ExecutionLoader.get_filename()* 具体实现的抽象基类。

参数 *fullname* 是加载器要处理的模块的完全解析的名字。参数 *path* 是模块文件的路径。

3.3 新版功能.

name
加载器可以处理的模块的名字。

path
模块的文件路径

load_module(*fullname*)
调用 super 的 “load_module()“。

3.4 版后已移除: 使用*Loader.exec_module()* 来代替。

abstractmethod get_filename(*fullname*)
返回*path*。

abstractmethod get_data(*path*)
读取 *path* 作为二进制文件并且返回来自它的字节数据。

class importlib.abc.**SourceLoader**
一个用于实现源文件（和可选地字节码）加载的抽象基类。这个类继承自*ResourceLoader* 和*ExecutionLoader*，需要实现：

- *ResourceLoader.get_data()*
- ***ExecutionLoader.get_filename()*** 应该是只返回源文件的路径；不支持无源加载。

由这个类定义的抽象方法用来添加可选的字节码文件支持。不实现这些可选的方法（或导致它们引发*NotImplementedError* 异常）导致这个加载器只能与源代码一起工作。实现这些方法允许加载器能与源 *和*字节码文件一起工作。不允许只提供字节码的 *无源式*加载。字节码文件是通过移除 Python 编译器的解析步骤来加速加载的优化，并且因此没有开放出字节码专用的 API。

path_stats(*path*)
返回一个包含关于指定路径的元数据的*dict* 的可选的抽象方法。支持的字典键有：

- `'mtime'` (必选项): 一个表示源码修改时间的整数或浮点数;
- `'size'` (可选项): 源码的字节大小。

字典中任何其他键会被忽略，以允许将来的扩展。如果不能处理该路径，则会引发*OSError*。

3.3 新版功能.

在 3.4 版更改: 引发*OSError* 而不是*NotImplemented*。

path_mtime(*path*)
返回指定文件路径修改时间的可选的抽象方法。

3.3 版后已移除: 在有了*path_stats()* 的情况下，这个方法被弃用了。没必要去实现它了，但是为了兼容性，它依然处于可用状态。如果文件路径不能被处理，则引发*OSError* 异常。

在 3.4 版更改: 引发*OSError* 而不是*NotImplemented*。

set_data(*path*, *data*)
往一个文件路径写入指定字节的的可选的抽象方法。任何中间不存在的目录不会被自动创建。

由于路径是只读的，当写入的路径产生错误时（*errno.EACCES*/*PermissionError*），不会传播异常。

在 3.4 版更改: 当被调用时，不再引起*NotImplementedError* 异常。

get_code(*fullname*)
InspectLoader.get_code() 的具体实现。

exec_module(*module*)
Loader.exec_module() 的具体实现。

3.4 新版功能.

load_module(*fullname*)
Concrete implementation of *Loader.load_module()*.

3.4 版后已移除: 使用*exec_module()* 来代替。

get_source(*fullname*)
InspectLoader.get_source() 的具体实现。

is_package(*fullname*)
InspectLoader.is_package() 的具体实现。一个模块被确定为一个包的条件是：它的文件路径（由*ExecutionLoader.get_filename()* 提供）当文件扩展名被移除时是一个命名为 `__init__` 的文件，**并且**这个模块名字本身不是以 "__init__"结束。

31.5.4 importlib.machinery –Importers and path hooks

Source code: Lib/importlib/machinery.py

This module contains the various objects that help `import` find and load modules.

`importlib.machinery.`**SOURCE_SUFFIXES**
A list of strings representing the recognized file suffixes for source modules.

3.3 新版功能.

`importlib.machinery.`**DEBUG_BYTECODE_SUFFIXES**
A list of strings representing the file suffixes for non-optimized bytecode modules.

3.3 新版功能.

3.5 版后已移除: Use *BYTECODE_SUFFIXES* instead.

importlib.machinery.**OPTIMIZED_BYTECODE_SUFFIXES**
A list of strings representing the file suffixes for optimized bytecode modules.

3.3 新版功能.

3.5 版后已移除: Use *BYTECODE_SUFFIXES* instead.

importlib.machinery.**BYTECODE_SUFFIXES**
A list of strings representing the recognized file suffixes for bytecode modules (including the leading dot).

3.3 新版功能.

在 3.5 版更改: The value is no longer dependent on __debug__.

importlib.machinery.**EXTENSION_SUFFIXES**
A list of strings representing the recognized file suffixes for extension modules.

3.3 新版功能.

importlib.machinery.**all_suffixes**()
Returns a combined list of strings representing all file suffixes for modules recognized by the standard import machinery. This is a helper for code which simply needs to know if a filesystem path potentially refers to a module without needing any details on the kind of module (for example, *inspect.getmodulename()*).

3.3 新版功能.

class importlib.machinery.**BuiltinImporter**
An *importer* for built-in modules. All known built-in modules are listed in *sys.builtin_module_names*. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

在 3.5 版更改: As part of **PEP 489**, the builtin importer now implements Loader.create_module() and Loader.exec_module()

class importlib.machinery.**FrozenImporter**
An *importer* for frozen modules. This class implements the *importlib.abc.MetaPathFinder* and *importlib.abc.InspectLoader* ABCs.

Only class methods are defined by this class to alleviate the need for instantiation.

class importlib.machinery.**WindowsRegistryFinder**
Finder for modules declared in the Windows registry. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

3.3 新版功能.

3.6 版后已移除: Use *site* configuration instead. Future versions of Python may not enable this finder by default.

class importlib.machinery.**PathFinder**
A *Finder* for *sys.path* and package __path__ attributes. This class implements the *importlib.abc.MetaPathFinder* ABC.

Only class methods are defined by this class to alleviate the need for instantiation.

classmethod find_spec(*fullname*, *path=None*, *target=None*)
Class method that attempts to find a *spec* for the module specified by *fullname* on *sys.path* or, if defined, on *path*. For each path entry that is searched, *sys.path_importer_cache* is checked. If a non-false object is found then it is used as the *path entry finder* to look for the module being searched for. If no entry is found in *sys.path_importer_cache*, then *sys.path_hooks* is searched for a finder for the

path entry and, if found, is stored in *sys.path_importer_cache* along with being queried about the module. If no finder is ever found then `None` is both stored in the cache and returned.

3.4 新版功能.

在 3.5 版更改: If the current working directory –represented by an empty string –is no longer valid then `None` is returned but no value is cached in *sys.path_importer_cache*.

classmethod find_module(*fullname*, *path=None*)
A legacy wrapper around *find_spec()*.

3.4 版后已移除: 使用*find_spec()* 来代替。

classmethod invalidate_caches()
Calls *importlib.abc.PathEntryFinder.invalidate_caches()* on all finders stored in *sys.path_importer_cache*.

在 3.4 版更改: Calls objects in *sys.path_hooks* with the current working directory for `''` (i.e. the empty string).

class importlib.machinery.**FileFinder**(*path*, **loader_details*)
A concrete implementation of *importlib.abc.PathEntryFinder* which caches results from the file system.

The *path* argument is the directory for which the finder is in charge of searching.

The *loader_details* argument is a variable number of 2-item tuples each containing a loader and a sequence of file suffixes the loader recognizes. The loaders are expected to be callables which accept two arguments of the module's name and the path to the file found.

The finder will cache the directory contents as necessary, making stat calls for each module search to verify the cache is not outdated. Because cache staleness relies upon the granularity of the operating system's state information of the file system, there is a potential race condition of searching for a module, creating a new file, and then searching for the module the new file represents. If the operations happen fast enough to fit within the granularity of stat calls, then the module search will fail. To prevent this from happening, when you create a module dynamically, make sure to call *importlib.invalidate_caches()*.

3.3 新版功能.

path
The path the finder will search in.

find_spec(*fullname*, *target=None*)
Attempt to find the spec to handle *fullname* within *path*.

3.4 新版功能.

find_loader(*fullname*)
Attempt to find the loader to handle *fullname* within *path*.

invalidate_caches()
Clear out the internal cache.

classmethod path_hook(**loader_details*)
A class method which returns a closure for use on *sys.path_hooks*. An instance of *FileFinder* is returned by the closure using the path argument given to the closure directly and *loader_details* indirectly.

If the argument to the closure is not an existing directory, *ImportError* is raised.

class importlib.machinery.**SourceFileLoader**(*fullname*, *path*)
A concrete implementation of *importlib.abc.SourceLoader* by subclassing *importlib.abc.FileLoader* and providing some concrete implementations of other methods.

3.3 新版功能.

name
The name of the module that this loader will handle.

path
The path to the source file.

is_package(*fullname*)
Return true if *path* appears to be for a package.

path_stats(*path*)
Concrete implementation of *importlib.abc.SourceLoader.path_stats()*.

set_data(*path*, *data*)
Concrete implementation of *importlib.abc.SourceLoader.set_data()*.

load_module(*name=None*)
Concrete implementation of *importlib.abc.Loader.load_module()* where specifying the name of the module to load is optional.

3.6 版后已移除: Use *importlib.abc.Loader.exec_module()* instead.

class importlib.machinery.**SourcelessFileLoader**(*fullname*, *path*)
A concrete implementation of *importlib.abc.FileLoader* which can import bytecode files (i.e. no source code files exist).

Please note that direct use of bytecode files (and thus not source code files) inhibits your modules from being usable by all Python implementations or new versions of Python which change the bytecode format.

3.3 新版功能.

name
The name of the module the loader will handle.

path
The path to the bytecode file.

is_package(*fullname*)
Determines if the module is a package based on *path*.

get_code(*fullname*)
Returns the code object for *name* created from *path*.

get_source(*fullname*)
Returns `None` as bytecode files have no source when this loader is used.

load_module(*name=None*)

Concrete implementation of *importlib.abc.Loader.load_module()* where specifying the name of the module to load is optional.

3.6 版后已移除: Use *importlib.abc.Loader.exec_module()* instead.

class importlib.machinery.**ExtensionFileLoader**(*fullname*, *path*)
A concrete implementation of *importlib.abc.ExecutionLoader* for extension modules.

The *fullname* argument specifies the name of the module the loader is to support. The *path* argument is the path to the extension module's file.

3.3 新版功能.

name
Name of the module the loader supports.

path
Path to the extension module.

create_module(*spec*)
Creates the module object from the given specification in accordance with **PEP 489**.

3.5 新版功能.

exec_module(*module*)
Initializes the given module object in accordance with **PEP 489**.

3.5 新版功能.

is_package(*fullname*)
Returns `True` if the file path points to a package's `__init__` module based on *EXTENSION_SUFFIXES*.

get_code(*fullname*)
Returns `None` as extension modules lack a code object.

get_source(*fullname*)
Returns `None` as extension modules do not have source code.

get_filename(*fullname*)
返回*path*。

3.4 新版功能.

class importlib.machinery.**ModuleSpec**(*name*, *loader*, *, *origin=None*, *loader_state=None*, *is_package=None*)
A specification for a module's import-system-related state. This is typically exposed as the module's `__spec__` attribute. In the descriptions below, the names in parentheses give the corresponding attribute available directly on the module object. E.g. `module.__spec__.origin == module.__file__`. Note however that while the *values* are usually equivalent, they can differ since there is no synchronization between the two objects. Thus it is possible to update the module's `__path__` at runtime, and this will not be automatically reflected in `__spec__.submodule_search_locations`.

3.4 新版功能.

name

(`__name__`)

A string for the fully-qualified name of the module.

loader

(`__loader__`)

The loader to use for loading. For namespace packages this should be set to `None`.

origin

(`__file__`)

Name of the place from which the module is loaded, e.g. "builtin" for built-in modules and the filename for modules loaded from source. Normally "origin" should be set, but it may be `None` (the default) which indicates it is unspecified.

submodule_search_locations

(`__path__`)

List of strings for where to find submodules, if a package (`None` otherwise).

loader_state

Container of extra module-specific data for use during loading (or `None`).

cached

(__cached__)

String for where the compiled module should be stored (or None).

parent

(__package__)

(Read-only) Fully-qualified name of the package to which the module belongs as a submodule (or None).

has_location

Boolean indicating whether or not the module's "origin" attribute refers to a loadable location.

31.5.5 importlib.util –Utility code for importers

Source code: Lib/importlib/util.py

This module contains the various objects that help in the construction of an *importer*.

importlib.util.**MAGIC_NUMBER**
The bytes which represent the bytecode version number. If you need help with loading/writing bytecode then consider *importlib.abc.SourceLoader*.

3.4 新版功能.

importlib.util.**cache_from_source**(*path*, *debug_override=None*, *, *optimization=None*)
Return the **PEP 3147**/**PEP 488** path to the byte-compiled file associated with the source *path*. For example, if *path* is /foo/bar/baz.py the return value would be /foo/bar/__pycache__/baz.cpython-32.pyc for Python 3.2. The cpython-32 string comes from the current magic tag (see get_tag(); if sys.implementation.cache_tag is not defined then *NotImplementedError* will be raised).

The *optimization* parameter is used to specify the optimization level of the bytecode file. An empty string represents no optimization, so /foo/bar/baz.py with an *optimization* of '' will result in a bytecode path of /foo/bar/__pycache__/baz.cpython-32.pyc. None causes the interpter's optimization level to be used. Any other value's string representation being used, so /foo/bar/baz.py with an *optimization* of 2 will lead to the bytecode path of /foo/bar/__pycache__/baz.cpython-32.opt-2.pyc. The string representation of *optimization* can only be alphanumeric, else *ValueError* is raised.

The *debug_override* parameter is deprecated and can be used to override the system's value for __debug__. A True value is the equivalent of setting *optimization* to the empty string. A False value is the same as setting *optimization* to 1. If both *debug_override* an *optimization* are not None then *TypeError* is raised.

3.4 新版功能.

在 3.5 版更改: The *optimization* parameter was added and the *debug_override* parameter was deprecated.

在 3.6 版更改: 接受一个*path-like object*。

importlib.util.**source_from_cache**(*path*)
Given the *path* to a **PEP 3147** file name, return the associated source code file path. For example, if *path* is /foo/bar/__pycache__/baz.cpython-32.pyc the returned path would be /foo/bar/baz.py. *path* need not exist, however if it does not conform to **PEP 3147** or **PEP 488** format, a ValueError is raised. If sys.implementation.cache_tag is not defined, *NotImplementedError* is raised.

3.4 新版功能.

在 3.6 版更改: 接受一个*path-like object*。

importlib.util.**decode_source**(*source_bytes*)
Decode the given bytes representing source code and return it as a string with universal newlines (as required by *importlib.abc.InspectLoader.get_source()*).

3.4 新版功能.

importlib.util.**resolve_name**(*name*, *package*)
Resolve a relative module name to an absolute one.

If **name** has no leading dots, then **name** is simply returned. This allows for usage such as `importlib.util.resolve_name('sys', __package__)` without doing a check to see if the **package** argument is needed.

ValueError is raised if **name** is a relative module name but package is a false value (e.g. `None` or the empty string). *ValueError* is also raised a relative name would escape its containing package (e.g. requesting `..bacon` from within the `spam` package).

3.3 新版功能.

importlib.util.**find_spec**(*name*, *package=None*)
Find the *spec* for a module, optionally relative to the specified **package** name. If the module is in *sys.modules*, then `sys.modules[name].__spec__` is returned (unless the spec would be `None` or is not set, in which case *ValueError* is raised). Otherwise a search using *sys.meta_path* is done. `None` is returned if no spec is found.

If **name** is for a submodule (contains a dot), the parent module is automatically imported.

name and **package** work the same as for `import_module()`.

3.4 新版功能.

importlib.util.**module_from_spec**(*spec*)
Create a new module based on **spec** and *spec.loader.create_module*.

If *spec.loader.create_module* does not return `None`, then any pre-existing attributes will not be reset. Also, no *AttributeError* will be raised if triggered while accessing **spec** or setting an attribute on the module.

This function is preferred over using *types.ModuleType* to create a new module as **spec** is used to set as many import-controlled attributes on the module as possible.

3.5 新版功能.

@importlib.util.**module_for_loader**
A *decorator* for *importlib.abc.Loader.load_module()* to handle selecting the proper module object to load with. The decorated method is expected to have a call signature taking two positional arguments (e.g. `load_module(self, module)`) for which the second argument will be the module **object** to be used by the loader. Note that the decorator will not work on static methods because of the assumption of two arguments.

The decorated method will take in the **name** of the module to be loaded as expected for a *loader*. If the module is not found in *sys.modules* then a new one is constructed. Regardless of where the module came from, `__loader__` set to **self** and `__package__` is set based on what *importlib.abc.InspectLoader.is_package()* returns (if available). These attributes are set unconditionally to support reloading.

If an exception is raised by the decorated method and a module was added to *sys.modules*, then the module will be removed to prevent a partially initialized module from being in left in *sys.modules*. If the module was already in *sys.modules* then it is left alone.

在 3.3 版更改: `__loader__` and `__package__` are automatically set (when possible).

在 3.4 版更改: Set `__name__`, `__loader__` `__package__` unconditionally to support reloading.

3.4 版后已移除: The import machinery now directly performs all the functionality provided by this function.

@importlib.util.**set_loader**
A *decorator* for *importlib.abc.Loader.load_module()* to set the __loader__ attribute on the returned module. If the attribute is already set the decorator does nothing. It is assumed that the first positional argument to the wrapped method (i.e. self) is what __loader__ should be set to.

在 3.4 版更改: Set __loader__ if set to None, as if the attribute does not exist.

3.4 版后已移除: The import machinery takes care of this automatically.

@importlib.util.**set_package**
A *decorator* for *importlib.abc.Loader.load_module()* to set the __package__ attribute on the returned module. If __package__ is set and has a value other than None it will not be changed.

3.4 版后已移除: The import machinery takes care of this automatically.

importlib.util.**spec_from_loader**(*name*, *loader*, *, *origin=None*, *is_package=None*)
A factory function for creating a ModuleSpec instance based on a loader. The parameters have the same meaning as they do for ModuleSpec. The function uses available *loader* APIs, such as InspectLoader.is_package(), to fill in any missing information on the spec.

3.4 新版功能.

importlib.util.**spec_from_file_location**(*name*, *location*, *, *loader=None*, *submodule_search_locations=None*)
A factory function for creating a ModuleSpec instance based on the path to a file. Missing information will be filled in on the spec by making use of loader APIs and by the implication that the module will be file-based.

3.4 新版功能.

在 3.6 版更改: 接受一个*path-like object*。

class importlib.util.**LazyLoader**(*loader*)
A class which postpones the execution of the loader of a module until the module has an attribute accessed.

This class **only** works with loaders that define *exec_module()* as control over what module type is used for the module is required. For those same reasons, the loader's *create_module()* method must return None or a type for which its __class__ attribute can be mutated along with not using *slots*. Finally, modules which substitute the object placed into *sys.modules* will not work as there is no way to properly replace the module references throughout the interpreter safely; *ValueError* is raised if such a substitution is detected.

注解: For projects where startup time is critical, this class allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this class is **heavily** discouraged due to error messages created during loading being postponed and thus occurring out of context.

3.5 新版功能.

在 3.6 版更改: Began calling *create_module()*, removing the compatibility warning for *importlib.machinery.BuiltinImporter* and *importlib.machinery.ExtensionFileLoader*.

classmethod factory(*loader*)
A static method which returns a callable that creates a lazy loader. This is meant to be used in situations where the loader is passed by class instead of by instance.

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

31.5.6 例子

Importing programmatically

To programmatically import a module, use *importlib.import_module()*.

```
import importlib

itertools = importlib.import_module('itertools')
```

Checking if a module can be imported

If you need to find out if a module can be imported without actually doing the import, then you should use *importlib.util.find_spec()*.

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

spec = importlib.util.find_spec(name)
if spec is None:
    print("can't find the itertools module")
else:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    # Adding the module to sys.modules is optional.
    sys.modules[name] = module
```

Importing a source file directly

To import a Python source file directly, use the following recipe (Python 3.5 and newer only):

```
import importlib.util
import sys

# For illustrative purposes.
import tokenize
file_path = tokenize.__file__
module_name = tokenize.__name__

spec = importlib.util.spec_from_file_location(module_name, file_path)
module = importlib.util.module_from_spec(spec)
spec.loader.exec_module(module)
# Optional; only necessary if you want to be able to import the module
# by name later.
sys.modules[module_name] = module
```

Setting up an importer

For deep customizations of import, you typically want to implement an *importer*. This means managing both the *finder* and *loader* side of things. For finders there are two flavours to choose from depending on your needs: a *meta path finder* or a *path entry finder*. The former is what you would put on `sys.meta_path` while the latter is what you create using a *path entry hook* on `sys.path_hooks` which works with `sys.path` entries to potentially create a finder. This example will show you how to register your own importers so that import will use them (for creating an importer for yourself, read the documentation for the appropriate classes defined within this package):

```
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)

# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

Approximating `importlib.import_module()`

Import itself is implemented in Python code, making it possible to expose most of the import machinery through importlib. The following helps illustrate the various APIs that importlib exposes by providing an approximate implementation of `importlib.import_module()` (Python 3.4 and newer for the importlib usage, Python 3.6 and newer for other parts of the code).

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
```

(下页继续)

(续上页)

```
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(module)
    sys.modules[absolute_name] = module
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

CHAPTER 32

Python 语言服务

Python 提供了许多模块来帮助使用 Python 语言。这些模块支持标记化、解析、语法分析、字节码反汇编以及各种其他工具。

这些模块包括：

32.1 `parser` —访问 Python 解析树

The *parser* module provides an interface to Python's internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because parsing is performed in a manner identical to the code forming the application. It is also faster.

注解： From Python 2.5 onward, it's much more convenient to cut in at the Abstract Syntax Tree (AST) generation and compilation stage, using the *ast* module.

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code, but some examples of using the *parser* module are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to reference-index. The parser itself is created from a grammar specification defined in the file `Grammar/Grammar` in the standard Python distribution. The parse trees stored in the ST objects created by this module are the actual output from the internal parser when created by the *expr()* or *suite()* functions, described below. The ST objects created by *sequence2st()* faithfully simulate those structures. Be aware that the values of the sequences which are considered "correct" will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, whereas source code has always been forward-compatible.

Each element of the sequences returned by *st2list()* or *st2tuple()* has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file `Include/graminit.h` and the Python module *symbol*. Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an `if_stmt`, are included in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, 'if')`, where `1` is the numeric value associated with all `NAME` tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as `(1, 'if', 12)`, where the `12` represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module *token*.

The ST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple "wrapper" class may be created in Python to hide the use of ST objects.

The *parser* module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

参见:

模块*symbol* 代表解析树内部节点的有用常量。

模块*token* 代表解析树叶子节点和测试节点值的函数的有用常量。

32.1.1 创建 ST 对象

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the `'eval'` and `'exec'` forms.

`parser.`**`expr`**(*source*)
: The *expr()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'eval')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.`**`suite`**(*source*)
: The *suite()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'exec')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.`**`sequence2st`**(*sequence*)
: This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a *ParserError* exception is raised. An ST object created this way should not be assumed to compile correctly; normal exceptions raised by compilation may still be initiated when the ST object is passed to *compilest()*. This may indicate problems not related to syntax (such as a *MemoryError* exception), but may also be due to constructs such as the result of parsing `del f(0)`, which escapes the Python parser but is checked by the bytecode compiler.

  Sequences representing terminal tokens may be represented as either two-element lists of the form `(1, 'name')` or as three-element lists of the form `(1, 'name', 56)`. If the third element is present, it is assumed to be a

valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

parser.**tuple2st**(*sequence*)
This is the same function as *sequence2st()*. This entry point is maintained for backward compatibility.

32.1.2 转换 ST 对象

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple-trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

parser.**st2list**(*st*, *line_info=False*, *col_info=False*)
This function accepts an ST object from the caller in *st* and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, *st2tuple()* should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token *ends*. This information is omitted if the flag is false or omitted.

parser.**st2tuple**(*st*, *line_info=False*, *col_info=False*)
This function accepts an ST object from the caller in *st* and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to *st2list()*.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

parser.**compilest**(*st*, *filename='<syntax-tree>'*)
The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of a call to the built-in *exec()* or *eval()* functions. This function provides the interface to the compiler, passing the internal parse tree from *st* to the parser, using the source file name specified by the *filename* parameter. The default value supplied for *filename* indicates that the source was an ST object.

Compiling an ST object may result in exceptions related to compilation; an example would be a *SyntaxError* caused by the parse tree for `del f(0)`: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The *SyntaxError* raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the *parser* module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.

32.1.3 Queries on ST Objects

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via *expr()* or *suite()* or from a parse tree via *sequence2st()*.

parser.**isexpr**(*st*)
When *st* represents an `'eval'` form, this function returns true, otherwise it returns false. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by *compilest()* cannot be queried like this either, and are identical to those created by the built-in *compile()* function.

parser.**issuite**(*st*)
This function mirrors *isexpr()* in that it reports whether an ST object represents an `'exec'` form, commonly

known as a "suite." It is not safe to assume that this function is equivalent to `not isexpr(st)`, as additional syntactic fragments may be supported in the future.

32.1.4 异常和错误处理

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

exception parser.**ParserError**
Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built-in *SyntaxError* raised during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to *sequence2st()* and an explanatory string. Calls to *sequence2st()* need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions *compilest()*, *expr()*, and *suite()* may raise exceptions which are normally raised by the parsing and compilation process. These include the built in exceptions *MemoryError*, *OverflowError*, *SyntaxError*, and *SystemError*. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

32.1.5 ST 对象

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the *pickle* module) is also supported.

parser.**STType**
The type of the objects returned by *expr()*, *suite()* and *sequence2st()*.

ST 对象具有以下方法:

ST.**compile**(*filename='<syntax-tree>'*)
和 `compilest(st, filename)` 相同.

ST.**isexpr**()
和 `isexpr(st)` 相同。

ST.**issuite**()
和 `issuite(st)` 相同。

ST.**tolist**(*line_info=False*, *col_info=False*)
和 `st2list(st, line_info, col_info)` 相同。

ST.**totuple**(*line_info=False*, *col_info=False*)
和 `st2tuple(st, line_info, col_info)` 相同。

32.1.6 示例: `compile()` 的模拟

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the *parser* module to produce an intermediate data structure is equivalent to the code

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the *parser* module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions:

```
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```

32.2 ast —抽象语法树

源代码： Lib/ast.py

---

ast 模块帮助 Python 程序处理 Python 语法的抽象语法树。抽象语法或许会随着 Python 的更新发布而改变；该模块能够帮助理解当前语法在编程层面的样貌。

抽象语法树可通过将 `ast.PyCF_ONLY_AST` 作为旗标传递给*compile()* 内置函数来生成，或是使用此模块中提供的*parse()* 辅助函数。返回结果将是一个对象树，，其中的类都继承自*ast.AST*。抽象语法树可被内置的*compile()* 函数编译为一个 Python 代码对象。

32.2.1 节点类

class ast.**AST**

这是所有 AST 节点类的基类。实际上，这些节点类派生自 `Parser/Python.asdl` 文件，其中定义的语法树示例如下。它们在 C 语言模块 `_ast` 中定义，并被导出至*ast* 模块。

抽象语法定义的每个左侧符号 (比方说，`ast.stmt` 或者 `ast.expr`) 定义了一个类。另外，在抽象语法定义的右侧，对每一个构造器也定义了一个类；这些类继承自树左侧的类。比如，`ast.BinOp` 继承自 `ast.expr`。对于多分支产生式 (也就是”和规则”)，树右侧的类是抽象的；只有特定构造器结点的实例能被构造。

_fields

每个具体类都有个属性*_fields*, 用来给出所有子节点的名字。

每个具体类的实例对它每个子节点都有一个属性，对应类型如文法中所定义。比如，`ast.BinOp` 的实例有个属性 `left`，类型是 `ast.expr`.

如果这些属性在文法中标记为可选（使用问号），对应值可能会是 `None`。如果这些属性有零或多个（用星号标记），对应值会用 Python 的列表来表示。所有可能的属性必须在用*compile()* 编译得到 AST 时给出，且是有效的值。

lineno
col_offset
ast.expr 和 ast.stmt 子类的实例有*lineno*和*col_offset* 属性。*lineno* 是源代码的行数 (从 1 开始, 所以第一行行数是 1)，而*col_offset* 是该生成节点第一个 token 的 UTF-8 字节偏移量。记录下 UTF-8 偏移量的原因是 parser 内部使用 UTF-8。

一个类的构造器 ast.T 像下面这样 parse 它的参数。

- 如果有位置参数，它们必须和 T._fields 中的元素一样多；他们会像这些名字的属性一样被赋值。
- 如果有关键字参数，它们必须被设为和给定值同名的属性。

比方说，要创建和填充节点 ast.UnaryOp，你得用

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Num()
node.operand.n = 5
node.operand.lineno = 0
node.operand.col_offset = 0
node.lineno = 0
node.col_offset = 0
```

或者更紧凑点

```
node = ast.UnaryOp(ast.USub(), ast.Num(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

32.2.2 抽象文法

抽象文法目前定义如下

```
-- ASDL's 7 builtin types are:
-- identifier, int, string, bytes, object, singleton, constant
--
-- singleton: None, True or False
-- constant can be None, whereas None means "no value" for object.

module Python
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)

        -- not really an actual node but useful in Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns)
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list, expr? returns)

          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
```

(下页继续)

(续上页)

```
           expr* decorator_list)
         | Return(expr? value)

         | Delete(expr* targets)
         | Assign(expr* targets, expr value)
         | AugAssign(expr target, operator op, expr value)
         -- 'simple' indicates that we annotate simple name without parens
         | AnnAssign(expr target, expr annotation, expr? value, int simple)

         -- use 'orelse' because else is a keyword in target languages
         | For(expr target, expr iter, stmt* body, stmt* orelse)
         | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse)
         | While(expr test, stmt* body, stmt* orelse)
         | If(expr test, stmt* body, stmt* orelse)
         | With(withitem* items, stmt* body)
         | AsyncWith(withitem* items, stmt* body)

         | Raise(expr? exc, expr? cause)
         | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
         | Assert(expr test, expr? msg)

         | Import(alias* names)
         | ImportFrom(identifier? module, alias* names, int? level)

         | Global(identifier* names)
         | Nonlocal(identifier* names)
         | Expr(expr value)
         | Pass | Break | Continue

         -- XXX Jython will be different
         -- col_offset is the byte offset in the utf8 string the parser uses
         attributes (int lineno, int col_offset)

         -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
         | BinOp(expr left, operator op, expr right)
         | UnaryOp(unaryop op, expr operand)
         | Lambda(arguments args, expr body)
         | IfExp(expr test, expr body, expr orelse)
         | Dict(expr* keys, expr* values)
         | Set(expr* elts)
         | ListComp(expr elt, comprehension* generators)
         | SetComp(expr elt, comprehension* generators)
         | DictComp(expr key, expr value, comprehension* generators)
         | GeneratorExp(expr elt, comprehension* generators)
         -- the grammar constrains where yield expressions can occur
         | Await(expr value)
         | Yield(expr? value)
         | YieldFrom(expr value)
         -- need sequences for compare to distinguish between
         -- x < 4 < 3 and (x < 4) < 3
         | Compare(expr left, cmpop* ops, expr* comparators)
         | Call(expr func, expr* args, keyword* keywords)
         | Num(object n) -- a number as a PyObject.
         | Str(string s) -- need to specify raw, unicode, etc?
         | FormattedValue(expr value, int? conversion, expr? format_spec)
```

(下页继续)

(续上页)

```
          | JoinedStr(expr* values)
          | Bytes(bytes s)
          | NameConstant(singleton value)
          | Ellipsis
          | Constant(constant value)

          -- the following expression can appear in assignment context
          | Attribute(expr value, identifier attr, expr_context ctx)
          | Subscript(expr value, slice slice, expr_context ctx)
          | Starred(expr value, expr_context ctx)
          | Name(identifier id, expr_context ctx)
          | List(expr* elts, expr_context ctx)
          | Tuple(expr* elts, expr_context ctx)

           -- col_offset is the byte offset in the utf8 string the parser uses
           attributes (int lineno, int col_offset)

    expr_context = Load | Store | Del | AugLoad | AugStore | Param

    slice = Slice(expr? lower, expr? upper, expr? step)
          | ExtSlice(slice* dims)
          | Index(expr value)

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset)

    arguments = (arg* args, arg? vararg, arg* kwonlyargs, expr* kw_defaults,
                 arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation)
           attributes (int lineno, int col_offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)

    withitem = (expr context_expr, expr? optional_vars)
}
```

32.2.3 ast 中的辅助函数

除了节点类，*ast* 模块里为遍历抽象语法树定义了这些工具函数和类:

ast.**parse**(*source*, *filename='<unknown>'*, *mode='exec'*)
把源码解析为 AST 节点。和 compile(source, filename, mode,ast.PyCF_ONLY_AST) 等价。

> **警告:** 足够复杂或是巨大的字符串可能导致 Python 解释器的崩溃，因为 Python 的 AST 编译器是有栈深限制的。

ast.**literal_eval**(*node_or_string*)
对表达式节点以及包含 Python 字面量或容器的字符串进行安全的求值。传入的字符串或者节点里可能只包含下列的 Python 字面量结构: 字符串，字节对象 (bytes)，数值，元组，列表，字典，集合，布尔值和 None。

This can be used for safely evaluating strings containing Python values from untrusted sources without the need to parse the values oneself. It is not capable of evaluating arbitrarily complex expressions, for example involving operators or indexing.

> **警告:** 足够复杂或是巨大的字符串可能导致 Python 解释器的崩溃，因为 Python 的 AST 编译器是有栈深限制的。

在 3.2 版更改: 目前支持字节和集合。

ast.**get_docstring**(*node*, *clean=True*)
Return the docstring of the given *node* (which must be a FunctionDef, ClassDef or Module node), or None if it has no docstring. If *clean* is true, clean up the docstring's indentation with *inspect.cleandoc()*.

ast.**fix_missing_locations**(*node*)
When you compile a node tree with *compile()*, the compiler expects lineno and col_offset attributes for every node that supports them. This is rather tedious to fill in for generated nodes, so this helper adds these attributes recursively where not already set, by setting them to the values of the parent node. It works recursively starting at *node*.

ast.**increment_lineno**(*node*, *n=1*)
Increment the line number of each node in the tree starting at *node* by *n*. This is useful to "move code" to a different location in a file.

ast.**copy_location**(*new_node*, *old_node*)
Copy source location (lineno and col_offset) from *old_node* to *new_node* if possible, and return *new_node*.

ast.**iter_fields**(*node*)
Yield a tuple of (fieldname, value) for each field in node._fields that is present on *node*.

ast.**iter_child_nodes**(*node*)
Yield all direct child nodes of *node*, that is, all fields that are nodes and all items of fields that are lists of nodes.

ast.**walk**(*node*)
Recursively yield all descendant nodes in the tree starting at *node* (including *node* itself), in no specified order. This is useful if you only want to modify nodes in place and don't care about the context.

class ast.**NodeVisitor**
A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the *visit()* method.

This class is meant to be subclassed, with the subclass adding visitor methods.

visit(*node*)
Visit a node. The default implementation calls the method called `self.visit_classname` where *classname* is the name of the node class, or *generic_visit()* if that method doesn't exist.

generic_visit(*node*)
This visitor calls *visit()* on all children of the node.

Note that child nodes of nodes that have a custom visitor method won't be visited unless the visitor calls *generic_visit()* or visits them itself.

Don't use the *NodeVisitor* if you want to apply changes to nodes during traversal. For this a special visitor exists (*NodeTransformer*) that allows modifications.

class ast.**NodeTransformer**
子类*NodeVisitor* 用于遍历抽象语法树，并允许修改节点。

NodeTransformer 将遍历抽象语法树并使用 visitor 方法的返回值去替换或移除旧节点。如果 visitor 方法的返回值为 `None`，则该节点将从其位置移除，否则将替换为返回值。当返回值是原始节点时，无需替换。

如下是一个转换器示例，它将所有出现的名称 (`foo`) 重写为 `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return copy_location(Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        ), node)
```

请记住，如果您正在操作的节点具有子节点，则必须先转换其子节点或为该节点调用 `generic_visit()` 方法。

对于属于语句集合（适用于所有语句节点）的节点，访问者还可以返回节点列表而不仅仅是单个节点。

通常你可以像这样使用转换器:

```
node = YourTransformer().visit(node)
```

ast.**dump**(*node*, *annotate_fields=True*, *include_attributes=False*)
Return a formatted dump of the tree in *node*. This is mainly useful for debugging purposes. The returned string will show the names and the values for fields. This makes the code impossible to evaluate, so if evaluation is wanted *annotate_fields* must be set to `False`. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, *include_attributes* can be set to `True`.

参见:

Green Tree Snakes, an external documentation resource, has good details on working with Python ASTs.

32.3 symtable —Access to the compiler’s symbol tables

Source code: Lib/symtable.py

Symbol tables are generated by the compiler from AST just before bytecode is generated. The symbol table is responsible for calculating the scope of every identifier in the code. *symtable* provides an interface to examine these tables.

32.3.1 Generating Symbol Tables

symtable.**symtable**(*code*, *filename*, *compile_type*)
Return the toplevel *SymbolTable* for the Python source *code*. *filename* is the name of the file containing the code. *compile_type* is like the *mode* argument to *compile()*.

32.3.2 Examining Symbol Tables

class symtable.**SymbolTable**
A namespace table for a block. The constructor is not public.

get_type()
Return the type of the symbol table. Possible values are `'class'`, `'module'`, and `'function'`.

get_id()
Return the table’s identifier.

get_name()
Return the table’s name. This is the name of the class if the table is for a class, the name of the function if the table is for a function, or `'top'` if the table is global (*get_type()* returns `'module'`).

get_lineno()
Return the number of the first line in the block this table represents.

is_optimized()
Return `True` if the locals in this table can be optimized.

is_nested()
Return `True` if the block is a nested class or function.

has_children()
Return `True` if the block has nested namespaces within it. These can be obtained with *get_children()*.

has_exec()
Return `True` if the block uses `exec`.

get_identifiers()
Return a list of names of symbols in this table.

lookup(*name*)
Lookup *name* in the table and return a *Symbol* instance.

get_symbols()
Return a list of *Symbol* instances for names in the table.

get_children()
Return a list of the nested symbol tables.

class symtable.**Function**
A namespace for a function or method. This class inherits *SymbolTable*.

get_parameters()
Return a tuple containing names of parameters to this function.

get_locals()
Return a tuple containing names of locals in this function.

get_globals()
Return a tuple containing names of globals in this function.

get_frees()
Return a tuple containing names of free variables in this function.

class symtable.**Class**
A namespace of a class. This class inherits *SymbolTable*.

get_methods()
Return a tuple containing the names of methods declared in the class.

class symtable.**Symbol**
An entry in a *SymbolTable* corresponding to an identifier in the source. The constructor is not public.

get_name()
Return the symbol's name.

is_referenced()
Return `True` if the symbol is used in its block.

is_imported()
Return `True` if the symbol is created from an import statement.

is_parameter()
Return `True` if the symbol is a parameter.

is_global()
Return `True` if the symbol is global.

is_declared_global()
Return `True` if the symbol is declared global with a global statement.

is_local()
Return `True` if the symbol is local to its block.

is_free()
Return `True` if the symbol is referenced in its block, but not assigned to.

is_assigned()
Return `True` if the symbol is assigned to in its block.

is_namespace()
Return `True` if name binding introduces new namespace.

If the name is used as the target of a function or class statement, this will be true.

例如

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

Note that a single name can be bound to multiple objects. If the result is `True`, the name may also be bound to other objects, like an int or list, that does not introduce a new namespace.

get_namespaces()
Return a list of namespaces bound to this name.

get_namespace()
Return the namespace bound to this name. If more than one namespace is bound, *ValueError* is raised.

32.4 symbol —与 Python 解析树一起使用的常量

源代码: Lib/symbol.py

此模块提供用于表示解析树内部节点数值的常量。与大多数 Python 不同，这些常量使用小写字符名称。请参阅 Python 发行版中的 Grammar/Grammar 文件来获取该语言语法上下文中对这些名称的定义。这些名称所映射的特定数字值可能会在 Python 版本之间更改。

此模块还提供了一个额外的数据对象:

symbol.**sym_name**
将此模块中定义的常量的数值映射回名称字符串的字典，允许生成更加人类可读的解析树表示。

32.5 token —与 Python 解析树一起使用的常量

源码: Lib/token.py

此模块提供表示解析树（终端令牌）的叶节点的数值的常量。请参阅 Python 发行版中的文件 Grammar/Grammar，以获取语言语法上下文中名称的定义。名称映射到的特定数值可能会在 Python 版本之间更改。

该模块还提供从数字代码到名称和一些函数的映射。这些函数镜像了 Python C 头文件中的定义。

token.**tok_name**
将此模块中定义的常量的数值映射回名称字符串的字典，允许生成更加人类可读的解析树表示。

token.**ISTERMINAL**(*x*)
Return true for terminal token values.

token.**ISNONTERMINAL**(*x*)
Return true for non-terminal token values.

token.**ISEOF**(*x*)
Return true if *x* is the marker indicating the end of input.

标记常量是:

token.**ENDMARKER**
token.**NAME**
token.**NUMBER**
token.**STRING**
token.**NEWLINE**
token.**INDENT**
token.**DEDENT**
token.**LPAR**
token.**RPAR**
token.**LSQB**
token.**RSQB**
token.**COLON**

token.**COMMA**
token.**SEMI**
token.**PLUS**
token.**MINUS**
token.**STAR**
token.**SLASH**
token.**VBAR**
token.**AMPER**
token.**LESS**
token.**GREATER**
token.**EQUAL**
token.**DOT**
token.**PERCENT**
token.**LBRACE**
token.**RBRACE**
token.**EQEQUAL**
token.**NOTEQUAL**
token.**LESSEQUAL**
token.**GREATEREQUAL**
token.**TILDE**
token.**CIRCUMFLEX**
token.**LEFTSHIFT**
token.**RIGHTSHIFT**
token.**DOUBLESTAR**
token.**PLUSEQUAL**
token.**MINEQUAL**
token.**STAREQUAL**
token.**SLASHEQUAL**
token.**PERCENTEQUAL**
token.**AMPEREQUAL**
token.**VBAREQUAL**
token.**CIRCUMFLEXEQUAL**
token.**LEFTSHIFTEQUAL**
token.**RIGHTSHIFTEQUAL**
token.**DOUBLESTAREQUAL**
token.**DOUBLESLASH**
token.**DOUBLESLASHEQUAL**
token.**AT**
token.**ATEQUAL**
token.**RARROW**
token.**ELLIPSIS**
token.**OP**
token.**AWAIT**
token.**ASYNC**
token.**ERRORTOKEN**
token.**N_TOKENS**
token.**NT_OFFSET**

在 3.5 版更改: Added *AWAIT* and *ASYNC* tokens. Starting with Python 3.7, "async" and "await" will be tokenized as *NAME* tokens, and *AWAIT* and *ASYNC* will be removed.

32.6 keyword —检验 Python 关键字

源码： Lib/keyword.py

This module allows a Python program to determine if a string is a keyword.

keyword.**iskeyword**(*s*)
Return true if *s* is a Python keyword.

keyword.**kwlist**
Sequence containing all the keywords defined for the interpreter. If any keywords are defined to only be active when particular *__future__* statements are in effect, these will be included as well.

32.7 tokenize –对 Python 代码使用的标记解析器

源码： Lib/tokenize.py

The *tokenize* module provides a lexical scanner for Python source code, implemented in Python. The scanner in this module returns comments as tokens as well, making it useful for implementing "pretty-printers," including colorizers for on-screen displays.

为了简化标记流的处理，所有的 运算符和 定界符以及*Ellipsis* 返回时都会打上通用的*OP* 标记。可以通过*tokenize.tokenize()* 返回的*named tuple* 对象的 exact_type 属性来获得确切的标记类型。

32.7.1 对输入进行解析标记

主要的入口是一个*generator*:

tokenize.**tokenize**(*readline*)
生成器*tokenize()* 需要一个 *readline* 参数，这个参数必须是一个可调用对象，且能提供与文件对象的*io.IOBase.readline()* 方法相同的接口。每次调用这个函数都要返回字节类型输入的一行数据。

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple (srow, scol) of ints specifying the row and column where the token begins in the source; a 2-tuple (erow, ecol) of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed (the last tuple item) is the *logical* line; continuation lines are included. The 5 tuple is returned as a *named tuple* with the field names: type string start end line.

The returned *named tuple* has an additional property named exact_type that contains the exact operator type for *token.OP* tokens. For all other token types exact_type equals the named tuple type field.

在 3.1 版更改: Added support for named tuples.

在 3.3 版更改: Added support for exact_type.

tokenize() determines the source encoding of the file by looking for a UTF-8 BOM or encoding cookie, according to **PEP 263**.

All constants from the *token* module are also exported from *tokenize*, as are three additional token type values:

tokenize.**COMMENT**
Token value used to indicate a comment.

tokenize.**NL**
Token value used to indicate a non-terminating newline. The NEWLINE token indicates the end of a logical line of Python code; NL tokens are generated when a logical line of code is continued over multiple physical lines.

tokenize.**ENCODING**
Token value that indicates the encoding used to decode the source bytes into text. The first token returned by *tokenize()* will always be an ENCODING token.

Another function is provided to reverse the tokenization process. This is useful for creating tools that tokenize a script, modify the token stream, and write back the modified script.

tokenize.**untokenize**(*iterable*)
Converts tokens back into Python source code. The *iterable* must return sequences with at least two elements, the token type and the token string. Any additional sequence elements are ignored.

The reconstructed script is returned as a single string. The result is guaranteed to tokenize back to match the input so that the conversion is lossless and round-trips are assured. The guarantee applies only to the token type and token string as the spacing between tokens (column positions) may change.

It returns bytes, encoded using the ENCODING token, which is the first token sequence output by *tokenize()*.

tokenize() needs to detect the encoding of source files it tokenizes. The function it uses to do this is available:

tokenize.**detect_encoding**(*readline*)
The *detect_encoding()* function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, readline, in the same way as the *tokenize()* generator.

It will call readline a maximum of twice, and return the encoding used (as a string) and a list of any lines (not decoded from bytes) it has read in.

It detects the encoding from the presence of a UTF-8 BOM or an encoding cookie as specified in **PEP 263**. If both a BOM and a cookie are present, but disagree, a SyntaxError will be raised. Note that if the BOM is found, `'utf-8-sig'` will be returned as an encoding.

If no encoding is specified, then the default of `'utf-8'` will be returned.

Use *open()* to open Python source files: it uses *detect_encoding()* to detect the file encoding.

tokenize.**open**(*filename*)
Open a file in read only mode using the encoding detected by *detect_encoding()*.

3.2 新版功能.

exception tokenize.**TokenError**
Raised when either a docstring or expression that may be split over several lines is not completed anywhere in the file, for example:

```
"""Beginning of
docstring
```

或者：

```
[1,
 2,
 3
```

Note that unclosed single-quoted strings do not cause an error to be raised. They are tokenized as `ERRORTOKEN`, followed by the tokenization of their contents.

32.7.2 Command-Line Usage

3.3 新版功能.

The *tokenize* module can be executed as a script from the command line. It is as simple as:

```
python -m tokenize [-e] [filename.py]
```

The following options are accepted:

-h, --help
show this help message and exit

-e, --exact
display token names using the exact type

If `filename.py` is specified its contents are tokenized to stdout. Otherwise, tokenization is performed on stdin.

32.7.3 例子

Example of a script rewriter that transforms float literals into Decimal objects:

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows).  Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
        else:
```

(下页继续)

(续上页)

```
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

Example of tokenizing from the command line. The script:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

will be tokenized to the following output where the first column is the range of the line/column coordinates where the token is found, the second column is the name of the token, and the final column is the value of the token (if any)

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

The exact token type names can be displayed using the `-e` option:

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          RPAR           ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           LPAR           '('
4,10-4,11:          RPAR           ')'
```

(下页继续)

(续上页)

```
4,11-4,12:          NEWLINE         '\n'
5,0-5,0:            ENDMARKER       ''
```

32.8 tabnanny —模糊缩进检测

源代码: Lib/tabnanny.py

目前，该模块旨在作为脚本调用。但是可以使用下面描述的*check()* 函数将其导入 IDE。

注解: 此模块提供的 API 可能会在将来的版本中更改；此类更改可能无法向后兼容。

---

tabnanny.**check**(*file_or_dir*)
如果 *file_or_dir* 是目录而非符号链接，则递归地在名为 *file_or_dir* 的目录树中下行，沿途检查所有 .py 文件。如果 *file_or_dir* 是一个普通 Python 源文件，将检查其中的空格相关问题。诊断消息将使用*print()* 函数写入到标准输出。

tabnanny.**verbose**
此标志指明是否打印详细消息。如果作为脚本调用则是通过 -v 选项来增加。

tabnanny.**filename_only**
此旗标指明是否只打印包含空格相关问题文件的文件名。如果作为脚本调用则是通过 -q 选项来设为真值。

exception tabnanny.**NannyNag**
如果检测到模糊缩进则由*process_tokens()* 引发。在*check()* 中捕获并处理。

tabnanny.**process_tokens**(*tokens*)
此函数由*check()* 用来处理由*tokenize* 模块所生成的标记。

参见:

模块*tokenize* 用于 Python 源代码的词法扫描程序。

32.9 pyclbr —Python class browser support

源代码: Lib/pyclbr.py

The *pyclbr* module can be used to determine some limited information about the classes, methods and top-level functions defined in a module. The information provided is sufficient to implement a traditional three-pane class browser. The information is extracted from the source code rather than by importing the module, so this module is safe to use with untrusted code. This restriction makes it impossible to use this module with modules not implemented in Python, including all standard and optional extension modules.

pyclbr.**readmodule**(*module*, *path=None*)
Read a module and return a dictionary mapping class names to class descriptor objects. The parameter *module* should be the name of a module as a string; it may be the name of a module within a package. The *path* parameter should be a sequence, and is used to augment the value of sys.path, which is used to locate module source code.

pyclbr.**readmodule_ex**(*module*, *path=None*)
Like *readmodule()*, but the returned dictionary, in addition to mapping class names to class descriptor objects, also maps top-level function names to function descriptor objects. Moreover, if the module being read is a package, the key `'__path__'` in the returned dictionary has as its value a list which contains the package search path.

32.9.1 类对象

The `Class` objects used as values in the dictionary returned by *readmodule()* and *readmodule_ex()* provide the following data attributes:

Class.**module**
The name of the module defining the class described by the class descriptor.

Class.**name**
类名称。

Class.**super**
A list of `Class` objects which describe the immediate base classes of the class being described. Classes which are named as superclasses but which are not discoverable by *readmodule()* are listed as a string with the class name instead of as `Class` objects.

Class.**methods**
A dictionary mapping method names to line numbers.

Class.**file**
Name of the file containing the `class` statement defining the class.

Class.**lineno**
The line number of the `class` statement within the file named by *file*.

32.9.2 函数对象

The `Function` objects used as values in the dictionary returned by *readmodule_ex()* provide the following attributes:

Function.**module**
The name of the module defining the function described by the function descriptor.

Function.**name**
函数名称。

Function.**file**
Name of the file containing the `def` statement defining the function.

Function.**lineno**
The line number of the `def` statement within the file named by *file*.

32.10 py_compile —编译 Python 源文件

源代码: Lib/py_compile.py

---

py_compile 模块提供了用来从源文件生成字节码的函数和另一个用于当模块源文件作为脚本被调用时的函数。

虽然不太常用，但这个函数在安装共享模块时还是很有用的，特别是当一些用户可能没有权限在包含源代码的目录中写字节码缓存文件时。

exception py_compile.**PyCompileError**
当编译文件过程中发生错误时，抛出的异常。

py_compile.**compile**(*file*, *cfile=None*, *dfile=None*, *doraise=False*, *optimize=-1*)
将源文件编译成字节码，并写出字节码缓存文件。源代码从名为 *file* 的文件中加载。字节码写入 *cfile*，默认为 **PEP 3147/PEP 488** 路径，以 .pyc 结尾。例如，如果 *file* 是 /foo/bar/baz.py，那么对于 Python 3.2，*cfile* 默认为 /foo/bar/__pycache__/baz.cpython-32.pyc。如果指定了 *dfile*，那么在错误信息中，它将代替 *file* 作为源文件的名称。如果 *doraise* 为 true，当编译 *file* 遇到错误时，会抛出一个*PyCompileError*。如果 *doraise* 为 false (默认值)，错误字符串将写入 sys.stderr，但不会抛出异常。该函数返回编译后字节文件的路径，即 *cfile* 的值。

如果 *cfile* 所表示（显式指定或计算得出）的路径为符号链接或非常规文件，则将引发*FileExistsError*。此行为是用来警告如果允许写入编译后字节码文件到这些路径则导入操作将会把它们转为常规文件。这是使用文件重命名来将最终编译后字节码文件放置到位以防止并发文件写入问题的导入操作的附带效果。

optimize 控制优化级别并会被传给内置的*compile()* 函数。默认值 -1 表示选择当前解释器的优化级别。

在 3.2 版更改: 将 *cfile* 的默认值改成与 **PEP 3147** 兼容。之前的默认值是 *file* + 'c' (如果启用优化则为 'o')。同时也添加了 *optimize* 形参。

在 3.4 版更改: 将代码更改为使用*importlib* 执行字节码缓存文件写入。这意味着文件创建/写入的语义现在与*importlib* 所做的相匹配，例如权限、写入和移动语义等等。同时也添加了当 *cfile* 为符号链接或非常规文件时引发*FileExistsError* 的预警设置。

py_compile.**main**(*args=None*)
编译多个源文件。在 *args* 中（或者当 *args* 为 None 时则是在命令行中）指定的文件会被编译并将结果字节码以正常方式来缓存。此函数不会搜索目录结构来定位源文件；它只编译显式指定的文件。如果 '-' 是 args 中唯一的值，则会从标准输入获取文件列表。

在 3.2 版更改: 添加了对 '-' 的支持。

当此模块作为脚本运行时，会使用*main()* 来编译命令行中指定的所有文件。如果某个文件无法被编译则退出状态将为非零值。

参见:

模块*compileall* 编译一个目录树中所有 Python 源文件的工具。

32.11 compileall —Byte-compile Python libraries

Source code: Lib/compileall.py

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached byte-code files at library installation time, which makes them available for use even by users who don't have write permission to the library directories.

32.11.1 Command-line use

This module can work as a script (using **python -m compileall**) to compile Python sources.

directory ...
file ...
Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was `-l <directories from sys.path>`.

-l
Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

-f
Force rebuild even if timestamps are up-to-date.

-q
Do not print the list of files compiled. If passed once, error messages will still be printed. If passed twice (`-qq`), all output is suppressed.

-d destdir
Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

-x regex
regex is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.

-i list
Read the file `list` and add each line that it contains to the list of files and directories to compile. If `list` is `-`, read lines from `stdin`.

-b
Write the byte-code files to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

-r
Control the maximum recursion level for subdirectories. If this is given, then `-l` option will not be taken into account. **python -m compileall <directory> -r 0** is equivalent to **python -m compileall <directory> -l**.

-j N
Use *N* workers to compile the files within the given directory. If `0` is used, then the result of *os.cpu_count()* will be used.

在 3.2 版更改: Added the `-i`, `-b` and `-h` options.

在 3.5 版更改: Added the `-j`, `-r`, and `-qq` options. `-q` option was changed to a multilevel value. `-b` will always produce a byte-code file ending in `.pyc`, never `.pyo`.

There is no command-line option to control the optimization level used by the *compile()* function, because the Python interpreter itself already provides the option: **python -O -m compileall**.

32.11.2 Public functions

compileall.**compile_dir**(*dir*, *maxlevels=10*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *workers=1*)

Recursively descend the directory tree named by *dir*, compiling all `.py` files along the way. Return a true value if all the files compiled successfully, and a false value otherwise.

The *maxlevels* parameter is used to limit the depth of the recursion; it defaults to `10`.

If *ddir* is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *force* is true, modules are re-compiled even if the timestamps are up to date.

If *rx* is given, its search method is called on the complete path to each file considered for compilation, and if it returns a true value, the file is skipped.

If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in *compile()* function.

The argument *workers* specifies how many workers are used to compile files in parallel. The default is to not use multiple workers. If the platform can't use multiple workers and *workers* argument is given, then sequential compilation will be used as a fallback. If *workers* is lower than `0`, a *ValueError* will be raised.

在 3.2 版更改: Added the *legacy* and *optimize* parameter.

在 3.5 版更改: Added the *workers* parameter.

在 3.5 版更改: *quiet* parameter was changed to a multilevel value.

在 3.5 版更改: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

在 3.6 版更改: 接受一个*path-like object*。

compileall.**compile_file**(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*)

Compile the file with path *fullname*. Return a true value if the file compiled successfully, and a false value otherwise.

If *ddir* is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *rx* is given, its search method is passed the full path name to the file being compiled, and if it returns a true value, the file is not compiled and `True` is returned.

If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in *compile()* function.

3.2 新版功能.

在 3.5 版更改: *quiet* parameter was changed to a multilevel value.

在 3.5 版更改: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

`compileall.`**compile_path**(*skip_curdir=True*, *maxlevels=0*, *force=False*, *quiet=0*, *legacy=False*, *optimize=-1*)

Byte-compile all the `.py` files found along `sys.path`. Return a true value if all the files compiled successfully, and a false value otherwise.

If *skip_curdir* is true (the default), the current directory is not included in the search. All other parameters are passed to the *compile_dir()* function. Note that unlike the other compile functions, `maxlevels` defaults to `0`.

在 3.2 版更改: Added the *legacy* and *optimize* parameter.

在 3.5 版更改: *quiet* parameter was changed to a multilevel value.

在 3.5 版更改: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

To force a recompile of all the `.py` files in the `Lib/` subdirectory and all its subdirectories:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

参见:

模块*py_compile* Byte-compile a single source file.

32.12 dis —Python 字节码反汇编器

Source code: Lib/dis.py

---

dis 模块通过反汇编支持 CPython 的*bytecode* 分析。该模块作为输入的 CPython 字节码在文件 `Include/opcode.h` 中定义，并由编译器和解释器使用。

CPython implementation detail: 字节码是 CPython 解释器的实现细节。不保证不会在 Python 版本之间添加、删除或更改字节码。不应考虑将此模块的跨 Python VM 或 Python 版本的使用。

在 3.6 版更改: 每条指令使用 2 个字节。以前字节数因指令而异。

示例：给出函数 `myfunc()`:

```
def myfunc(alist):
    return len(alist)
```

可以使用以下命令显示 myfunc() 的反汇编

```
>>> dis.dis(myfunc)
  2           0 LOAD_GLOBAL              0 (len)
              2 LOAD_FAST                0 (alist)
              4 CALL_FUNCTION            1
              6 RETURN_VALUE
```

(“2”是行号)。

32.12.1 字节码分析

3.4 新版功能.

字节码分析 API 允许将 Python 代码片段包装在*Bytecode* 对象中，以便轻松访问已编译代码的详细信息。

class dis.**Bytecode**(*x*, *, *first_line=None*, *current_offset=None*)

Analyse the bytecode corresponding to a function, generator, method, string of source code, or a code object (as returned by *compile()*).

这是下面列出的许多函数的便利包装，最值得注意的是*get_instructions()* ，迭代于*Bytecode* 的实例产生字节码操作*Instruction* 的实例。

如果 *first_line* 不是 None ，则表示应该为反汇编代码中的第一个源代码行报告的行号。否则，源行信息（如果有的话）直接来自反汇编的代码对象。

如果 *current_offset* 不是 None ，则它指的是反汇编代码中的指令偏移量。设置它意味着*dis()* 将针对指定的操作码显示“当前指令”标记。

classmethod from_traceback(*tb*)

从给定回溯构造一个*Bytecode* 实例，设置 *current_offset* 为异常负责的指令。

codeobj

已编译的代码对象。

first_line

代码对象的第一个源代码行（如果可用）

dis()

返回字节码操作的格式化视图（与*dis.dis()* 打印相同，但作为多行字符串返回）。

info()

返回带有关于代码对象的详细信息的格式化多行字符串，如*code_info()* 。

示例:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```

32.12.2 分析函数

dis 模块还定义了以下分析函数，它们将输入直接转换为所需的输出。如果只执行单个操作，它们可能很有用，因此中间分析对象没用：

dis.**code_info**(*x*)
Return a formatted multi-line string with detailed code object information for the supplied function, generator, method, source code string or code object.

请注意，代码信息字符串的确切内容是高度依赖于实现的，它们可能会在 Python VM 或 Python 版本中任意更改。

3.2 新版功能.

dis.**show_code**(*x*, *, *file=None*)
将提供的函数、方法。源代码字符串或代码对象的详细代码对象信息打印到 *file* （如果未指定 *file* ，则为 `sys.stdout` ）。

这是 `print(code_info(x), file= file)` 的便捷简写，用于在解释器提示符下进行交互式探索。

3.2 新版功能.

在 3.4 版更改: 添加 *file* 参数。

dis.**dis**(*x=None*, *, *file=None*)
Disassemble the *x* object. *x* can denote either a module, a class, a method, a function, a generator, a code object, a string of source code or a byte sequence of raw bytecode. For a module, it disassembles all functions. For a class, it disassembles all methods (including class and static methods). For a code object or sequence of raw bytecode, it prints one line per bytecode instruction. Strings are first compiled to code objects with the *compile()* built-in function before being disassembled. If no object is provided, this function disassembles the last traceback.

如果提供的话，反汇编将作为文本写入提供的 *file* 参数，否则写入 `sys.stdout` 。

在 3.4 版更改: 添加 *file* 参数。

dis.**distb**(*tb=None*, *, *file=None*)
如果没有传递，则使用最后一个回溯来反汇编回溯的堆栈顶部函数。指示了导致异常的指令。

如果提供的话，反汇编将作为文本写入提供的 *file* 参数，否则写入 `sys.stdout` 。

在 3.4 版更改: 添加 *file* 参数。

dis.**disassemble**(*code*, *lasti=-1*, *, *file=None*)
dis.**disco**(*code*, *lasti=-1*, *, *file=None*)
反汇编代码对象，如果提供了 *lasti* ，则指示最后一条指令。输出分为以下几列：

1. 行号，用于每行的第一条指令
2. 当前指令，表示为 `-->` ，
3. 一个标记的指令，用 >> 表示，
4. 指令的地址，
5. 操作码名称，
6. 操作参数，和
7. 括号中的参数解释。

参数解释识别本地和全局变量名称、常量值、分支目标和比较运算符。

如果提供的话，反汇编将作为文本写入提供的 *file* 参数，否则写入 `sys.stdout` 。

在 3.4 版更改: 添加 *file* 参数。

dis.**get_instructions**(*x*, *, *first_line=None*)
在所提供的函数、方法、源代码字符串或代码对象中的指令上返回一个迭代器。

迭代器生成一系列*Instruction*，命名为元组，提供所提供代码中每个操作的详细信息。

如果 *first_line* 不是 None，则表示应该为反汇编代码中的第一个源代码行报告的行号。否则，源行信息（如果有的话）直接来自反汇编的代码对象。

3.4 新版功能.

dis.**findlinestarts**(*code*)
此生成器函数使用代码对象 *code* 的 co_firstlineno 和 co_lnotab 属性来查找源代码中行开头的偏移量。它们生成为 (offset, lineno)` `对。请参阅:source:`objects/lnotab_notes.txt` ，了解 ``co_lnotab 格式以及如何解码它。

在 3.6 版更改: 行号可能会减少。以前，他们总是在增加。

dis.**findlabels**(*code*)
Detect all offsets in the code object *code* which are jump targets, and return a list of these offsets.

dis.**stack_effect**(*opcode*[, *oparg*])
使用参数 *oparg* 计算 *opcode* 的堆栈效果。

3.4 新版功能.

32.12.3 Python 字节码说明

get_instructions() 函数和*Bytecode* 类提供字节码指令的详细信息的*Instruction* 实例：

class dis.**Instruction**
字节码操作的详细信息

opcode
操作的数字代码，对应于下面列出的操作码值和操作码集合 中的字节码值。

opname
人类可读的操作名称

arg
操作的数字参数（如果有的话），否则为 None

argval
已解析的 arg 值（如果已知），否则与 arg 相同

argrepr
人类可读的操作参数描述

offset
在字节码序列中启动操作索引

starts_line
行由此操作码（如果有）启动，否则为 None

is_jump_target
如果其他代码跳到这里，则为 True，否则为 False

3.4 新版功能.

Python 编译器当前生成以下字节码指令。

一般指令

NOP
什么都不做。用作字节码优化器的占位符。

POP_TOP
删除堆栈顶部（TOS）项。

ROT_TWO
交换两个最顶层的堆栈项。

ROT_THREE
将第二个和第三个堆栈项向上提升一个位置，顶项移动到位置三。

DUP_TOP
复制堆栈顶部的引用。

3.2 新版功能.

DUP_TOP_TWO
复制堆栈顶部的两个引用，使它们保持相同的顺序。

3.2 新版功能.

一元操作

一元操作获取堆栈顶部元素，应用操作，并将结果推回堆栈。

UNARY_POSITIVE
实现 `TOS = +TOS` 。

UNARY_NEGATIVE
实现 `TOS = -TOS` 。

UNARY_NOT
实现 `TOS = not TOS` 。

UNARY_INVERT
实现 `TOS = ~TOS` 。

GET_ITER
实现 `TOS = iter(TOS)` 。

GET_YIELD_FROM_ITER
如果 `TOS` 是一个*generator iterator* 或*coroutine* 对象则保持原样。否则实现 `TOS = iter(TOS)` 。

3.5 新版功能.

二元操作

二元操作从堆栈中删除堆栈顶部（TOS）和第二个最顶层堆栈项（TOS1）。它们执行操作，并将结果放回堆栈。

BINARY_POWER
实现 `TOS = TOS1 ** TOS` 。

BINARY_MULTIPLY
实现 `TOS = TOS1 * TOS` 。

BINARY_MATRIX_MULTIPLY
实现 `TOS = TOS1 @ TOS` 。

3.5 新版功能.

BINARY_FLOOR_DIVIDE
实现 `TOS = TOS1 // TOS`。

BINARY_TRUE_DIVIDE
实现 `TOS = TOS1 / TOS`。

BINARY_MODULO
实现 `TOS = TOS1 % TOS`。

BINARY_ADD
实现 `TOS = TOS1 + TOS`。

BINARY_SUBTRACT
实现 `TOS = TOS1 - TOS`。

BINARY_SUBSCR
实现 `TOS = TOS1[TOS]`。

BINARY_LSHIFT
实现 `TOS = TOS1 << TOS`。

BINARY_RSHIFT
实现 `TOS = TOS1 >> TOS`。

BINARY_AND
实现 `TOS = TOS1 & TOS`。

BINARY_XOR
实现 `TOS = TOS1 ^ TOS`。

BINARY_OR
实现 `TOS = TOS1 | TOS`。

就地操作

就地操作就像二元操作，因为它们删除了 TOS 和 TOS1，并将结果推回到堆栈上，但是当 TOS1 支持它时，操作就地完成，并且产生的 TOS 可能是（但不一定）原来的 TOS1。

INPLACE_POWER
就地实现 `TOS = TOS1 ** TOS`。

INPLACE_MULTIPLY
就地实现 `TOS = TOS1 * TOS`。

INPLACE_MATRIX_MULTIPLY
就地实现 `TOS = TOS1 @ TOS`。

3.5 新版功能.

INPLACE_FLOOR_DIVIDE
就地实现 `TOS = TOS1 // TOS`。

INPLACE_TRUE_DIVIDE
就地实现 `TOS = TOS1 / TOS`。

INPLACE_MODULO
就地实现 `TOS = TOS1 % TOS`。

INPLACE_ADD
就地实现 `TOS = TOS1 + TOS`。

INPLACE_SUBTRACT
就地实现 `TOS = TOS1 - TOS`。

INPLACE_LSHIFT
就地实现 `TOS = TOS1 << TOS`。

INPLACE_RSHIFT
就地实现 TOS = TOS1 >> TOS 。

INPLACE_AND
就地实现 TOS = TOS1 & TOS 。

INPLACE_XOR
就地实现 TOS = TOS1 ^ TOS 。

INPLACE_OR
就地实现 TOS = TOS1 | TOS 。

STORE_SUBSCR
实现 TOS1[TOS] = TOS2 。

DELETE_SUBSCR
实现 del TOS1[TOS] 。

协程操作码

GET_AWAITABLE
实现 TOS = get_awaitable(TOS) ，其中 get_awaitable(o) 返回 o 如果 o 是一个有 CO_ITERABLE_COROUTINE 标志的协程对象或生成器对象，否则解析 o.__await__ 。

3.5 新版功能.

GET_AITER
Implements TOS = get_awaitable(TOS.__aiter__()). See GET_AWAITABLE for details about get_awaitable

3.5 新版功能.

GET_ANEXT
实现 PUSH(get_awaitable(TOS.__anext__())) 。参见 GET_AWAITABLE 获取更多 get_awaitable 的细节

3.5 新版功能.

BEFORE_ASYNC_WITH
从栈顶元素解析 __aenter__ 和 __aexit__ 。将 __aexit__ 和 __aenter__() 的结果推入堆栈。

3.5 新版功能.

SETUP_ASYNC_WITH
创建一个新的帧对象。

3.5 新版功能.

其他操作码

PRINT_EXPR
实现交互模式的表达式语句。TOS 从堆栈中被移除并打印。在非交互模式下，表达式语句以*POP_TOP* 终止。

BREAK_LOOP
Terminates a loop due to a break statement.

CONTINUE_LOOP (*target*)
Continues a loop due to a continue statement. *target* is the address to jump to (which should be a *FOR_ITER* instruction).

SET_ADD (*i*)
调用 set.add(TOS1[-i], TOS) 。用于实现集合推导。

LIST_APPEND (*i*)
调用 `list.append(TOS[-i], TOS)` 。用于实现列表推导。

MAP_ADD (*i*)
Calls `dict.setitem(TOS1[-i], TOS, TOS1)`. Used to implement dict comprehensions.

3.1 新版功能.

对于所有*SET_ADD* 、*LIST_APPEND* 和*MAP_ADD* 指令，当弹出添加的值或键值对时，容器对象保留在堆栈上，以便它可用于循环的进一步迭代。

RETURN_VALUE
返回 TOS 到函数的调用者。

YIELD_VALUE
弹出 TOS 并从一个*generator* 生成它。

YIELD_FROM
弹出 TOS 并将其委托给它作为*generator* 的子迭代器。

3.3 新版功能.

SETUP_ANNOTATIONS
检查 `__annotations__` 是否在 `locals()` 中定义，如果没有，它被设置为空 `dict` 。只有在类或模块体静态地包含*variable annotations* 时才会发出此操作码。

3.6 新版功能.

IMPORT_STAR
将所有不以 `'_'` 开头的符号直接从模块 TOS 加载到本地名称空间。加载所有名称后弹出该模块。这个操作码实现了 `from module import *` 。

POP_BLOCK
Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

POP_EXCEPT
从块堆栈中删除一个块。弹出的块必须是异常处理程序块，在进入 except 处理程序时隐式创建。除了从帧堆栈弹出无关值之外，最后三个弹出值还用于恢复异常状态。

END_FINALLY
Terminates a `finally` clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

LOAD_BUILD_CLASS
将 `builtins .__ build_class__()` 推到堆栈上。它之后被*CALL_FUNCTION* 调用来构造一个类。

SETUP_WITH (*delta*)
This opcode performs several operations before a with block starts. First, it loads `__exit__()` from the context manager and pushes it onto the stack for later use by `WITH_CLEANUP`. Then, `__enter__()` is called, and a finally block pointing to *delta* is pushed. Finally, the result of calling the enter method is pushed onto the stack. The next opcode will either ignore it (*POP_TOP*), or store it in (a) variable(s) (*STORE_FAST*, *STORE_NAME*, or *UNPACK_SEQUENCE*).

3.2 新版功能.

WITH_CLEANUP_START
Cleans up the stack when a `with` statement block exits. TOS is the context manager's `__exit__()` bound method. Below TOS are 1–3 values indicating how/why the finally clause was entered:

- SECOND = `None`
- (SECOND, THIRD) = (`WHY_{RETURN,CONTINUE}`), retval

- SECOND = `WHY_*`; no retval below it
- (SECOND, THIRD, FOURTH) = exc_info()

In the last case, `TOS(SECOND, THIRD, FOURTH)` is called, otherwise `TOS(None, None, None)`. Pushes SECOND and result of the call to the stack.

WITH_CLEANUP_FINISH
Pops exception type and result of 'exit' function call from the stack.

If the stack represents an exception, *and* the function call returns a 'true' value, this information is "zapped" and replaced with a single `WHY_SILENCED` to prevent *END_FINALLY* from re-raising the exception. (But non-local gotos will still be resumed.)

以下所有操作码均使用其参数。

STORE_NAME (*namei*)
实现 `name = TOS`。*namei* 是 *name* 在代码对象的 `co_names` 属性中的索引。在可能的情况下，编译器会尝试使用*STORE_FAST* 或*STORE_GLOBAL*。

DELETE_NAME (*namei*)
实现 `del name`，其中 *namei* 是代码对象的 `co_names` 属性的索引。

UNPACK_SEQUENCE (*count*)
将 TOS 解包为 *count* 个单独的值，它们将按从右至左的顺序被放入堆栈。

UNPACK_EX (*counts*)
实现使用带星号的目标进行赋值：将 TOS 中的可迭代对象解包为单独的值，其中值的总数可以小于可迭代对象中的项数：新值之一将是由所有剩余项构成的列表。

counts 的低字节是列表值之前的值的数量，*counts* 中的高字节则是之后的值的数量。结果值会按从右至左的顺序入栈。

STORE_ATTR (*namei*)
实现 `TOS.name = TOS1`，其中 *namei* 是 name 在 `co_names` 中的索引号。

DELETE_ATTR (*namei*)
实现 `del TOS.name`，使用 *namei* 作为 `co_names` 中的索引号。

STORE_GLOBAL (*namei*)
类似于*STORE_NAME* 但会将 name 存储为全局变量。

DELETE_GLOBAL (*namei*)
类似于*DELETE_NAME* 但会删除一个全局变量。

LOAD_CONST (*consti*)
将 `co_consts[consti]` 推入栈顶。

LOAD_NAME (*namei*)
将与 `co_names[namei]` 相关联的值推入栈顶。

BUILD_TUPLE (*count*)
创建一个使用了来自栈的 *count* 个项的元组，并将结果元组推入栈顶。

BUILD_LIST (*count*)
类似于*BUILD_TUPLE* 但会创建一个列表。

BUILD_SET (*count*)
类似于*BUILD_TUPLE* 但会创建一个集合。

BUILD_MAP (*count*)
将一个新字典对象推入栈顶。弹出 `2 * count` 项使得字典包含 *count* 个条目：`{..., TOS3: TOS2, TOS1: TOS}`。

在 3.5 版更改: 字典是根据栈中的项创建而不是创建一个预设大小包含 *count* 项的空字典。

BUILD_CONST_KEY_MAP (*count*)
The version of *BUILD_MAP* specialized for constant keys. *count* values are consumed from the stack. The top element on the stack contains a tuple of keys.

3.6 新版功能.

BUILD_STRING (*count*)
拼接 *count* 个来自栈的字符串并将结果字符串推入栈顶。

3.6 新版功能.

BUILD_TUPLE_UNPACK (*count*)
从栈中弹出 *count* 个可迭代对象，将它们合并为单个元组，并将结果推入栈顶。实现可迭代对象解包为元组形式 `(*x, *y, *z)`。

3.5 新版功能.

BUILD_TUPLE_UNPACK_WITH_CALL (*count*)
这类似于*BUILD_TUPLE_UNPACK* 但专用于 `f(*x, *y, *z)` 调用语法。栈中 `count + 1` 位置上的项应当是相应的可调用对象 `f`。

3.6 新版功能.

BUILD_LIST_UNPACK (*count*)
这类似于*BUILD_TUPLE_UNPACK* 但会将一个列表而非元组推入栈顶。实现可迭代对象解包为列表形式 `[*x, *y, *z]`。

3.5 新版功能.

BUILD_SET_UNPACK (*count*)
这类似于*BUILD_TUPLE_UNPACK* 但会将一个集合而非元组推入栈顶。实现可迭代对象解包为集合形式 `{*x, *y, *z}`。

3.5 新版功能.

BUILD_MAP_UNPACK (*count*)
从栈中弹出 *count* 个映射对象，将它们合并为单个字典，并将结果推入栈顶。实现字典解包为字典形式 `{**x, **y, **z}`。

3.5 新版功能.

BUILD_MAP_UNPACK_WITH_CALL (*count*)
这类似于*BUILD_MAP_UNPACK* 但专用于 `f(**x, **y, **z)` 调用语法。栈中 `count + 2` 位置上的项应当是相应的可调用对象 `f`。

3.5 新版功能.

在 3.6 版更改: 可迭代对象的位置的确定方式是将操作码参数加 2 而不是将其编码到参数的第二个字节。

LOAD_ATTR (*namei*)
将 TOS 替换为 `getattr(TOS, co_names[namei])`。

COMPARE_OP (*opname*)
执行布尔运算操作。操作名称可在 `cmp_op[opname]` 中找到。

IMPORT_NAME (*namei*)
导入模块 `co_names[namei]`。会弹出 TOS 和 TOS1 以提供 *fromlist* 和 *level* 参数给*__import__()*。模块对象会被推入栈顶。当前命名空间不受影响：对于一条标准 import 语句，会执行后续的*STORE_FAST* 指令来修改命名空间。

IMPORT_FROM(*namei*)
从在 TOS 内找到的模块中加载属性 co_names[namei]。结果对象会被推入栈顶，以便由后续的*STORE_FAST* 指令来保存。

JUMP_FORWARD(*delta*)
将字节码计数器的值增加 *delta*。

POP_JUMP_IF_TRUE(*target*)
如果 TOS 为真值，则将字节码计数器的值设为 *target*。TOS 会被弹出。

3.1 新版功能.

POP_JUMP_IF_FALSE(*target*)
如果 TOS 为假值，则将字节码计数器的值设为 *target*。TOS 会被弹出。

3.1 新版功能.

JUMP_IF_TRUE_OR_POP(*target*)
如果 TOS 为真值，则将字节码计数器的值设为 *target* 并将 TOS 留在栈顶。否则（如 TOS 为假值），TOS 会被弹出。

3.1 新版功能.

JUMP_IF_FALSE_OR_POP(*target*)
如果 TOS 为假值，则将字节码计数器的值设为 *target* 并将 TOS 留在栈顶。否则（如 TOS 为真值），TOS 会被弹出。

3.1 新版功能.

JUMP_ABSOLUTE(*target*)
将字节码计数器的值设为 *target*。

FOR_ITER(*delta*)
TOS 是一个*iterator*。可调用它的*__next__()* 方法。如果产生了一个新值，则将其推入栈顶（将迭代器留在其下方）。如果迭代器提示已耗尽则 TOS 会被弹出，并将字节码计数器的值增加 *delta*。

LOAD_GLOBAL(*namei*)
加载名称为 co_names[namei] 的全局对象推入栈顶。

SETUP_LOOP(*delta*)
Pushes a block for a loop onto the block stack. The block spans from the current instruction with a size of *delta* bytes.

SETUP_EXCEPT(*delta*)
Pushes a try block from a try-except clause onto the block stack. *delta* points to the first except block.

SETUP_FINALLY(*delta*)
Pushes a try block from a try-except clause onto the block stack. *delta* points to the finally block.

LOAD_FAST(*var_num*)
将指向局部对象 co_varnames[var_num] 的引用推入栈顶。

STORE_FAST(*var_num*)
将 TOS 存放到局部变量 co_varnames[var_num]。

DELETE_FAST(*var_num*)
移除局部对象 co_varnames[var_num]。

STORE_ANNOTATION(*namei*)
Stores TOS as `locals()['__annotations__'][co_names[namei]] = TOS`.

3.6 新版功能.

LOAD_CLOSURE(*i*)
将一个包含在单元的第 *i* 个空位中的对单元的引用推入栈顶并释放可用的存储空间。如果 *i* 小于 *co_cellvars* 的长度则变量的名称为 `co_cellvars[i]`。否则为 `co_freevars[i - len(co_cellvars)]`。

LOAD_DEREF(*i*)
加载包含在单元的第 *i* 个空位中的单元并释放可用的存储空间。将一个对单元所包含对象的引用推入栈顶。

LOAD_CLASSDEREF(*i*)
类似于*LOAD_DEREF* 但在查询单元之前会首先检查局部对象字典。这被用于加载类语句体中的自由变量。

3.4 新版功能.

STORE_DEREF(*i*)
将 TOS 存放到包含在单元的第 *i* 个空位中的单元内并释放可用存储空间。

DELETE_DEREF(*i*)
清空包含在单元的第 *i* 个空位中的单元并释放可用存储空间。被用于 `del` 语句。

3.2 新版功能.

RAISE_VARARGS(*argc*)
Raises an exception. *argc* indicates the number of arguments to the raise statement, ranging from 0 to 3. The handler will find the traceback as TOS2, the parameter as TOS1, and the exception as TOS.

CALL_FUNCTION(*argc*)
调用一个可调用对象并传入位置参数。*argc* 指明位置参数的数量。栈顶包含位置参数，其中最右边的参数在最顶端。在参数之下是一个待调用的可调用对象。`CALL_FUNCTION` 会从栈中弹出所有参数以及可调用对象，附带这些参数调用该可调用对象，并将可调用对象所返回的返回值推入栈顶。

在 3.6 版更改: 此操作码仅用于附带位置参数的调用。

CALL_FUNCTION_KW(*argc*)
调用一个可调用对象并传入位置参数（如果有的话）和关键字参数。*argc* 指明位置参数和关键字参数的总数量。栈顶元素包含一个关键字参数名称的元组。在元组之下是根据元组排序的关键字参数。在关键字参数之下是位置参数，其中最右边的参数在最顶端。在参数之下是一个待调用的可调用对象。`CALL_FUNCTION_KW` 会从栈中弹出所有参数以及可调用对象，附带这些参数调用该可调用对象，并将可调用对象所返回的返回值推入栈顶。

在 3.6 版更改: 关键字参数会被打包为一个元组而非字典，*argc* 指明参数的总数量。

CALL_FUNCTION_EX(*flags*)
调用一个可调用对象并附带位置参数和关键字参数变量集合。如果设置了 *flags* 的最低位，则栈顶包含一个由额外关键字参数组成的映射对象。在该对象之下是一个包含位置参数的可迭代对象和一个待调用的可调用对象。*BUILD_MAP_UNPACK_WITH_CALL* 和*BUILD_TUPLE_UNPACK_WITH_CALL* 可用于合并多个映射对象和包含参数的可迭代对象。在该可调用对象被调用之前，映射对象和可迭代对象会被分别 “解包” 并将它们的内容分别作为关键字参数和位置参数传入。`CALL_FUNCTION_EX` 会从栈中弹出所有参数以及可调用对象，附带这些参数调用该可调用对象，并将可调用对象所返回的返回值推入栈顶。

3.6 新版功能.

MAKE_FUNCTION(*argc*)
将一个新函数对象推入栈顶。从底端到顶端，如果参数带有指定的旗标值则所使用的栈必须由这些值组成。

- `0x01` 是一个默认值的元组，用于按位置排序的仅限位置形参以及位置或关键字形参
- `0x02` 是一个仅限关键字形参的默认值的字典

- 0x04 是一个标注字典
- 0x08 一个包含用于自由变量的单元的元组，生成一个闭包
- 与函数 (在 TOS1) 相关联的代码
- 函数的*qualified name* (在 TOS)

BUILD_SLICE (*argc*)
将一个切片对象推入栈顶。*argc* 必须为 2 或 3。如果为 2，则推入 slice(TOS1, TOS)；如果为 3，则推入 slice(TOS2, TOS1, TOS)。请参阅*slice()* 内置函数了解详细信息。

EXTENDED_ARG (*ext*)
Prefixes any opcode which has an argument too big to fit into the default two bytes. *ext* holds two additional bytes which, taken together with the subsequent opcode's argument, comprise a four-byte argument, *ext* being the two most-significant bytes.

FORMAT_VALUE (*flags*)
用于实现格式化字面值字符串（f-字符串）。从栈中弹出一个可选的 *fmt_spec*，然后是一个必须的 *value*。*flags* 的解读方式如下：

- (flags & 0x03) == 0x00: *value* 按原样格式化。
- (flags & 0x03) == 0x01: 在格式化 *value* 之前调用其*str()*。
- (flags & 0x03) == 0x02: 在格式化 *value* 之前调用其*repr()*。
- (flags & 0x03) == 0x03: 在格式化 *value* 之前调用其*ascii()*。
- (flags & 0x04) == 0x04: 从栈中弹出 *fmt_spec* 并使用它，否则使用空的 *fmt_spec*。

使用 PyObject_Format() 执行格式化。结果会被推入栈顶。

3.6 新版功能.

HAVE_ARGUMENT
这不是一个真正的操作码。它用于标明使用参数和不使用参数的操作码 (分别为 < HAVE_ARGUMENT 和 >= HAVE_ARGUMENT) 之间的分隔线。

在 3.6 版更改: 现在每条指令都带有参数，但操作码 < HAVE_ARGUMENT 会忽略它。之前仅限操作码 >= HAVE_ARGUMENT 带有参数。

32.12.4 操作码集合

提供这些集合用于字节码指令的自动内省：

dis.**opname**
操作名称序列，可使用字节码来索引。

dis.**opmap**
映射操作名称到字节码的字典

dis.**cmp_op**
所有比较操作名称的序列。

dis.**hasconst**
访问常量的字节码序列。

dis.**hasfree**
访问自由变量的字节码序列（请注意这里所说的‘自由’是指在当前作用域中被内部作用域所引用的名称，或在外部作用域中被此作用域所引用的名称。它 *并不* 包括对全局或内置作用域的引用）。

dis.**hasname**
按名称访问属性的字节码序列

dis.**hasjrel**
具有相对跳转目标的字节码序列。

dis.**hasjabs**
具有绝对跳转目标的字节码序列。

dis.**haslocal**
访问局部变量的字节码序列。

dis.**hascompare**
布尔运算的字节码序列

32.13 pickletools —pickle 开发者工具集

源代码: Lib/pickletools.py

此模块包含与*pickle*模块内部细节有关的多个常量，一些关于具体实现的详细注释，以及一些能够分析封存数据的有用函数。此模块的内容对需要操作*pickle*的 Python 核心开发者来说很有用处；*pickle*的一般用户则可能会感觉*pickletools*模块与他们无关。

32.13.1 命令行语法

3.2 新版功能.

当从命令行发起调用时，`python -m pickletools`将对一个或更多 pickle 文件的内容进行拆解。请注意如果你查看 pickle 中保存的 Python 对象而非 pickle 格式的细节，你可能需要改用`-m pickle`。但是，当你想检查的 pickle 文件来自某个不受信任的源时，`-m pickletools`是更安全的选择，因为它不会执行 pickle 字节码。

例如，对于一个封存在文件`x.pickle`中的元组`(1, 2)`:

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

命令行选项

-a, --annotate
使用简短的操作码描述来标注每一行。

-o, --output=<file>
输出应当写入到的文件名称。

-l, --indentlevel=<num>
一个新的 MARK 层级所需缩进的空格数。

-m, --memo
当拆解多个对象时，保留各个拆解的备忘记录。

-p, --preamble=<preamble>
当指定一个以上的 pickle 文件时，在每次反汇编之前打印给定的前言。

32.13.2 编程接口

pickletools.**dis**(*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)
将 pickle 的符号化反汇编数据输出到文件类对象 *out*，默认为 `sys.stdout`。*pickle* 可以是一个字符串或一个文件类对象。*memo* 可以是一个将被用作 pickle 的备忘记录的 Python 字典；它可被用来对由同一封存器创建的多个封存对象执行反汇编。由 `MARK` 操作码指明的每个连续级别将会缩进 *indentlevel* 个空格。如果为 *annotate* 指定了一个非零值，则输出中的每个操作码将以一个简短描述来标注。*annotate* 的值会被用作标注所应开始的列的提示。

3.2 新版功能: *annotate* 参数。

pickletools.**genops**(*pickle*)
提供包含 pickle 中所有操作码的*iterator*，返回一个 `(opcode, arg, pos)` 三元组的序列。*opcode* 是 `OpcodeInfo` 类的一个实例；*arg* 是 Python 对象形式的 opcode 参数的已解码值；*pos* 是 opcode 所在的位置。*pickle* 可以是一个字符串或一个文件类对象。

pickletools.**optimize**(*picklestring*)
在消除未使用的 `PUT` 操作码之后返回一个新的等效 pickle 字符串。优化后的 pickle 将更为简短，耗费更为的传输时间，要求更少的存储空间并能更高效地解封。

CHAPTER 33

杂项服务

本章中介绍的模块提供了所有 Python 版本中提供的各种杂项服务。这是一个概述：

33.1 formatter —通用格式化输出

3.4 版后已移除: 因为被使用的次数很少，此格式化模块已经被弃用了。

This module supports two interface definitions, each with multiple implementations: The *formatter* interface, and the *writer* interface which is required by the formatter interface.

Formatter objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of "change back" operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.

33.1.1 The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

`formatter.`**`AS_IS`**
: Value which can be used in the font specification passed to the `push_font()` method described below, or as the new value to any other `push_property()` method. Pushing the `AS_IS` value allows the corresponding `pop_property()` method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

formatter.**writer**
The writer instance with which the formatter interacts.

formatter.**end_paragraph**(*blanklines*)
Close any open paragraphs and insert at least *blanklines* before the next paragraph.

formatter.**add_line_break**()
Add a hard line break if one does not already exist. This does not break the logical paragraph.

formatter.**add_hor_rule**(**args*, ***kw*)
Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the writer's `send_line_break()` method.

formatter.**add_flowing_data**(*data*)
Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to *add_flowing_data()* is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

formatter.**add_literal_data**(*data*)
Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of *data*.

formatter.**add_label_data**(*format*, *counter*)
Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the *format* value is a string, it is interpreted as a format specification for *counter*, which should be an integer. The result of this formatting becomes the value of the label; if *format* is not a string it is used as the label value directly. The label value is passed as the only argument to the writer's `send_label_data()` method. Interpretation of non-string label values is dependent on the associated writer.

Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character `'1'` represents the counter value formatter as an Arabic number, the characters `'A'` and `'a'` represent alphabetic representations of the counter value in upper and lower case, respectively, and `'I'` and `'i'` represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

formatter.**flush_softspace**()
Send any pending whitespace buffered from a previous call to *add_flowing_data()* to the associated writer object. This should be called before any direct manipulation of the writer object.

formatter.**push_alignment**(*align*)
Push a new alignment setting onto the alignment stack. This may be *AS_IS* if no change is desired. If the alignment value is changed from the previous setting, the writer's `new_alignment()` method is called with the *align* value.

formatter.**pop_alignment**()
Restore the previous alignment.

formatter.**push_font**(*(size*, *italic*, *bold*, *teletype)*)
Change some or all font properties of the writer object. Properties which are not set to *AS_IS* are set to the values passed in while others are maintained at their current settings. The writer's `new_font()` method is called with the fully resolved font specification.

formatter.**pop_font**()
Restore the previous font.

formatter.**push_margin**(*margin*)
Increase the number of left margin indentations by one, associating the logical tag *margin* with the new indentation. The initial margin level is `0`. Changed values of the logical tag must be true values; false values other than *AS_IS* are not sufficient to change the margin.

formatter.**pop_margin**()
Restore the previous margin.

formatter.**push_style**(**styles*)
Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including *AS_IS* values, is passed to the writer's `new_styles()` method.

formatter.**pop_style**(*n=1*)
Pop the last *n* style specifications passed to *push_style()*. A tuple representing the revised stack, including *AS_IS* values, is passed to the writer's `new_styles()` method.

formatter.**set_spacing**(*spacing*)
Set the spacing style for the writer.

formatter.**assert_line_data**(*flag=1*)
Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional *flag* argument can be set to false if the writer manipulations produced a hard line break at the end of the output.

33.1.2 Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

class formatter.**NullFormatter**(*writer=None*)
A formatter which does nothing. If *writer* is omitted, a *NullWriter* instance is created. No methods of the writer are called by *NullFormatter* instances. Implementations should inherit from this class if implementing a writer interface but don't need to inherit any implementation.

class formatter.**AbstractFormatter**(*writer*)
The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.

33.1.3 The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the *AbstractFormatter* class as a formatter, the writer must typically be provided by the application.

writer.**flush**()
Flush any buffered output or device control events.

writer.**new_alignment**(*align*)
Set the alignment style. The *align* value can be any object, but by convention is a string or `None`, where `None` indicates that the writer's "preferred" alignment should be used. Conventional *align* values are `'left'`, `'center'`, `'right'`, and `'justify'`.

writer.**new_font**(*font*)
Set the font style. The value of *font* will be `None`, indicating that the device's default font should be used, or a tuple of the form `(size, italic, bold, teletype)`. Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The *italic*, *bold*, and *teletype* values are Boolean values specifying which of those font attributes should be used.

writer.**new_margin**(*margin*, *level*)
Set the margin level to the integer *level* and the logical tag to *margin*. Interpretation of the logical tag is at the writer's discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of *level*.

writer.**new_spacing**(*spacing*)
Set the spacing style to *spacing*.

writer.**new_styles**(*styles*)
Set additional styles. The *styles* value is a tuple of arbitrary values; the value `AS_IS` should be ignored. The *styles* tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

writer.**send_line_break**()
Break the current line.

writer.**send_paragraph**(*blankline*)
Produce a paragraph separation of at least *blankline* blank lines, or the equivalent. The *blankline* value will be an integer. Note that the implementation will receive a call to `send_line_break()` before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

writer.**send_hor_rule**(**args*, ***kw*)
Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via `send_line_break()`.

writer.**send_flowing_data**(*data*)
Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.

writer.**send_literal_data**(*data*)
Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the `send_formatted_data()` interface.

writer.**send_label_data**(*data*)
Set *data* to the left of the current left margin, if possible. The value of *data* is not restricted; treatment of non-string values is entirely application- and writer-dependent. This method will only be called at the beginning of a line.

33.1.4 Writer Implementations

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the `NullWriter` class.

class formatter.**NullWriter**
A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

class formatter.**AbstractWriter**
A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

class formatter.**DumbWriter**(*file=None*, *maxcol=72*)
Simple writer class which writes output on the *file object* passed in as *file* or, if *file* is omitted, on standard output.

The output is simply word-wrapped to the number of columns specified by *maxcol*. This class is suitable for reflowing a sequence of paragraphs.

CHAPTER 34

Windows 系统相关模块

本章节叙述的模块只在 Windows 平台上可用。

34.1 msilib —Read and write Microsoft Installer files

Source code: Lib/msilib/__init__.py

The *msilib* supports the creation of Microsoft Installer (`.msi`) files. Because these files often contain an embedded "cabinet" file (`.cab`), it also exposes an API to create CAB files. Support for reading `.cab` files is currently not implemented; read support for the `.msi` database is possible.

This package aims to provide complete access to all tables in an `.msi` file, therefore, it is a fairly low-level API. Two primary applications of this package are the *distutils* command `bdist_msi`, and the creation of Python installer package itself (although that currently uses a different version of `msilib`).

The package contents can be roughly split into four parts: low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

`msilib.`**`FCICreate`**(*cabname*, *files*)
Create a new CAB file named *cabname*. *files* must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.

Callbacks to Python for the various steps of MSI creation are currently not exposed.

`msilib.`**`UuidCreate`**()
Return the string representation of a new unique identifier. This wraps the Windows API functions `UuidCreate()` and `UuidToString()`.

`msilib.`**`OpenDatabase`**(*path*, *persist*)
Return a new database object by calling MsiOpenDatabase. *path* is the file name of the MSI file; *persist* can be one of the constants `MSIDBOPEN_CREATEDIRECT`, `MSIDBOPEN_CREATE`, `MSIDBOPEN_DIRECT`,

MSIDBOPEN_READONLY, or MSIDBOPEN_TRANSACT, and may include the flag MSIDBOPEN_PATCHFILE. See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

msilib.**CreateRecord**(*count*)
Return a new record object by calling MSICreateRecord(). *count* is the number of fields of the record.

msilib.**init_database**(*name*, *schema*, *ProductName*, *ProductCode*, *ProductVersion*, *Manufacturer*)
Create and return a new database *name*, initialize it with *schema*, and set the properties *ProductName*, *ProductCode*, *ProductVersion*, and *Manufacturer*.

schema must be a module object containing tables and _Validation_records attributes; typically, *msilib.schema* should be used.

The database will contain just the schema and the validation records when this function returns.

msilib.**add_data**(*database*, *table*, *records*)
Add all *records* to the table named *table* in *database*.

The *table* argument must be one of the predefined tables in the MSI schema, e.g. 'Feature', 'File', 'Component', 'Dialog', 'Control', etc.

records should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, None can be passed.

Field values can be ints, strings, or instances of the Binary class.

class msilib.**Binary**(*filename*)
Represents entries in the Binary table; inserting such an object using *add_data()* reads the file named *filename* into the table.

msilib.**add_tables**(*database*, *module*)
Add all table content from *module* to *database*. *module* must contain an attribute *tables* listing all tables for which content should be added, and one attribute per table that has the actual content.

This is typically used to install the sequence tables.

msilib.**add_stream**(*database*, *name*, *path*)
Add the file *path* into the _Stream table of *database*, with the stream name *name*.

msilib.**gen_uuid**()
Return a new UUID, in the format that MSI typically requires (i.e. in curly braces, and with all hexdigits in uppercase).

参见:

FCICreate UuidCreate UuidToString

34.1.1 Database Objects

Database.**OpenView**(*sql*)
Return a view object, by calling MSIDatabaseOpenView(). *sql* is the SQL statement to execute.

Database.**Commit**()
Commit the changes pending in the current transaction, by calling MSIDatabaseCommit().

Database.**GetSummaryInformation**(*count*)
Return a new summary information object, by calling MsiGetSummaryInformation(). *count* is the maximum number of updated values.

参见:

MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation

34.1.2 View Objects

`View.`**`Execute`**(*params*)
Execute the SQL query of the view, through `MSIViewExecute()`. If *params* is not `None`, it is a record describing actual values of the parameter tokens in the query.

`View.`**`GetColumnInfo`**(*kind*)
Return a record describing the columns of the view, through calling `MsiViewGetColumnInfo()`. *kind* can be either `MSICOLINFO_NAMES` or `MSICOLINFO_TYPES`.

`View.`**`Fetch`**()
Return a result record of the query, through calling `MsiViewFetch()`.

`View.`**`Modify`**(*kind*, *data*)
Modify the view, by calling `MsiViewModify()`. *kind* can be one of `MSIMODIFY_SEEK`, `MSIMODIFY_REFRESH`, `MSIMODIFY_INSERT`, `MSIMODIFY_UPDATE`, `MSIMODIFY_ASSIGN`, `MSIMODIFY_REPLACE`, `MSIMODIFY_MERGE`, `MSIMODIFY_DELETE`, `MSIMODIFY_INSERT_TEMPORARY`, `MSIMODIFY_VALIDATE`, `MSIMODIFY_VALIDATE_NEW`, `MSIMODIFY_VALIDATE_FIELD`, or `MSIMODIFY_VALIDATE_DELETE`.

data must be a record describing the new data.

`View.`**`Close`**()
Close the view, through `MsiViewClose()`.

参见:

MsiViewExecute MSIViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose

34.1.3 Summary Information Objects

`SummaryInformation.`**`GetProperty`**(*field*)
Return a property of the summary, through `MsiSummaryInfoGetProperty()`. *field* is the name of the property, and can be one of the constants `PID_CODEPAGE`, `PID_TITLE`, `PID_SUBJECT`, `PID_AUTHOR`, `PID_KEYWORDS`, `PID_COMMENTS`, `PID_TEMPLATE`, `PID_LASTAUTHOR`, `PID_REVNUMBER`, `PID_LASTPRINTED`, `PID_CREATE_DTM`, `PID_LASTSAVE_DTM`, `PID_PAGECOUNT`, `PID_WORDCOUNT`, `PID_CHARCOUNT`, `PID_APPNAME`, or `PID_SECURITY`.

`SummaryInformation.`**`GetPropertyCount`**()
Return the number of summary properties, through `MsiSummaryInfoGetPropertyCount()`.

`SummaryInformation.`**`SetProperty`**(*field*, *value*)
Set a property through `MsiSummaryInfoSetProperty()`. *field* can have the same values as in *`GetProperty()`*, *value* is the new value of the property. Possible value types are integer and string.

`SummaryInformation.`**`Persist`**()
Write the modified properties to the summary information stream, using `MsiSummaryInfoPersist()`.

参见:

MsiSummaryInfoGetProperty MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty MsiSummaryInfoPersist

34.1.4 Record Objects

Record.**GetFieldCount**()
Return the number of fields of the record, through MsiRecordGetFieldCount().

Record.**GetInteger**(*field*)
Return the value of *field* as an integer where possible. *field* must be an integer.

Record.**GetString**(*field*)
Return the value of *field* as a string where possible. *field* must be an integer.

Record.**SetString**(*field*, *value*)
Set *field* to *value* through MsiRecordSetString(). *field* must be an integer; *value* a string.

Record.**SetStream**(*field*, *value*)
Set *field* to the contents of the file named *value*, through MsiRecordSetStream(). *field* must be an integer; *value* a string.

Record.**SetInteger**(*field*, *value*)
Set *field* to *value* through MsiRecordSetInteger(). Both *field* and *value* must be an integer.

Record.**ClearData**()
Set all fields of the record to 0, through MsiRecordClearData().

参见:

MsiRecordGetFieldCount MsiRecordSetString MsiRecordSetStream MsiRecordSetInteger MsiRecordClearData

34.1.5 Errors

All wrappers around MSI functions raise MSIError; the string inside the exception will contain more detail.

34.1.6 CAB Objects

class msilib.**CAB**(*name*)
The class *CAB* represents a CAB file. During MSI construction, files will be added simultaneously to the Files table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

name is the name of the CAB file in the MSI file.

append(*full*, *file*, *logical*)
Add the file with the pathname *full* to the CAB file, under the name *logical*. If there is already a file named *logical*, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

commit(*database*)
Generate a CAB file, add it as a stream to the MSI file, put it into the Media table, and remove the generated file from the disk.

34.1.7 Directory Objects

class msilib.**Directory**(*database*, *cab*, *basedir*, *physical*, *logical*, *default*[, *componentflags*])

Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through *start_component()*, or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be None), the path to the physical directory, and a logical directory name. *default* specifies the DefaultDir slot in the directory table. *componentflags* specifies the default flags that new components get.

start_component(*component=None*, *feature=None*, *flags=None*, *keyfile=None*, *uuid=None*)

Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no *feature* is given, the current feature is used. If no *flags* are given, the directory's default flags are used. If no *keyfile* is given, the KeyPath is left null in the Component table.

add_file(*file*, *src=None*, *version=None*, *language=None*)

Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the *src* file is specified, it is interpreted relative to the current directory. Optionally, a *version* and a *language* can be specified for the entry in the File table.

glob(*pattern*, *exclude=None*)

Add a list of files to the current component as specified in the glob pattern. Individual files can be excluded in the *exclude* list.

remove_pyc()

Remove .pyc files on uninstall.

参见:

Directory Table File Table Component Table FeatureComponents Table

34.1.8 相关特性

class msilib.**Feature**(*db*, *id*, *title*, *desc*, *display*, *level=1*, *parent=None*, *directory=None*, *attributes=0*)

Add a new record to the Feature table, using the values *id*, *parent.id*, *title*, *desc*, *display*, *level*, *directory*, and *attributes*. The resulting feature object can be passed to the start_component() method of *Directory*.

set_current()

Make this feature the current feature of *msilib*. New components are automatically added to the default feature, unless a feature is explicitly specified.

参见:

Feature Table

34.1.9 GUI classes

msilib provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use `bdist_msi` to create MSI files with a user-interface for installing Python packages.

class msilib.**Control**(*dlg*, *name*)
Base class of the dialog controls. *dlg* is the dialog object the control belongs to, and *name* is the control's name.

event(*event*, *argument*, *condition=1*, *ordering=None*)
Make an entry into the `ControlEvent` table for this control.

mapping(*event*, *attribute*)
Make an entry into the `EventMapping` table for this control.

condition(*action*, *condition*)
Make an entry into the `ControlCondition` table for this control.

class msilib.**RadioButtonGroup**(*dlg*, *name*, *property*)
Create a radio button control named *name*. *property* is the installer property that gets set when a radio button is selected.

add(*name*, *x*, *y*, *width*, *height*, *text*, *value=None*)
Add a radio button named *name* to the group, at the coordinates *x*, *y*, *width*, *height*, and with the label *text*. If *value* is `None`, it defaults to *name*.

class msilib.**Dialog**(*db*, *name*, *x*, *y*, *w*, *h*, *attr*, *title*, *first*, *default*, *cancel*)
Return a new *Dialog* object. An entry in the `Dialog` table is made, with the specified coordinates, dialog attributes, title, name of the first, default, and cancel controls.

control(*name*, *type*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *control_next*, *help*)
Return a new *Control* object. An entry in the `Control` table is made with the specified parameters.

This is a generic method; for specific types, specialized methods are provided.

text(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*)
Add and return a `Text` control.

bitmap(*name*, *x*, *y*, *width*, *height*, *text*)
Add and return a `Bitmap` control.

line(*name*, *x*, *y*, *width*, *height*)
Add and return a `Line` control.

pushbutton(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*, *next_control*)
Add and return a `PushButton` control.

radiogroup(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `RadioButtonGroup` control.

checkbox(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `CheckBox` control.

参见:

Dialog Table Control Table Control Types ControlCondition Table ControlEvent Table EventMapping Table RadioButton Table

34.1.10 Precomputed tables

msilib provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

msilib.**schema**
This is the standard MSI schema for MSI 2.0, with the *tables* variable providing a list of table definitions, and *_Validation_records* providing the data for MSI validation.

msilib.**sequence**
This module contains table contents for the standard sequence tables: *AdminExecuteSequence*, *AdminUISequence*, *AdvtExecuteSequence*, *InstallExecuteSequence*, and *InstallUISequence*.

msilib.**text**
This module contains definitions for the UIText and ActionText tables, for the standard installer actions.

34.2 msvcrt —来自 MS VC++ 运行时的有用例程

这些函数提供了对 Windows 平台上一些有用功能的访问。一些更高级别的模块使用这些函数来构建其服务的 Windows 实现。例如，*getpass* 模块在实现*getpass()* 函数时使用了这些函数。

关于这些函数的更多信息可以在平台 API 文档中找到。

该模块实现了控制台 I/O API 的普通和宽字符变体。普通的 API 只处理 ASCII 字符，国际化应用受限。应该尽可能地使用宽字符 API 。

在 3.3 版更改: 此模块中过去会引发*IOError* 的操作现在将引发*OSError*。

34.2.1 文件操作

msvcrt.**locking**(*fd*, *mode*, *nbytes*)
基于文件描述符 *fd* 从 C 运行时锁定文件的某一部分。失败时引发*OSError*。锁定的文件区域从当前文件位置开始扩展 *nbytes* 个字节，并可能持续到超出文件末尾。*mode* 必须为下面列出的 LK_* 之一。一个文件中的多个区域可以被同时锁定，但是不能重叠。相邻区域不会被合并；它们必须单独被解锁。

msvcrt.**LK_LOCK**
msvcrt.**LK_RLCK**
锁定指定的字节数据。如果字节数据无法被锁定，程序会在 1 秒之后立即重试。如果在 10 次尝试后字节数据仍无法被锁定，则会引发*OSError*。

msvcrt.**LK_NBLCK**
msvcrt.**LK_NBRLCK**
锁定指定的字节数据。如果字节数据无法被锁定，则会引发*OSError*。

msvcrt.**LK_UNLCK**
解锁指定的字节数据，该对象必须在之前被锁定。

msvcrt.**setmode**(*fd*, *flags*)
设置文件描述符 *fd* 的行结束符转写模式。要将其设为文本模式，则 *flags* 应当为*os.O_TEXT*；设为二进制模式，则应当为*os.O_BINARY*。

msvcrt.**open_osfhandle**(*handle*, *flags*)
基于文件句柄 *handle* 创建一个 C 运行时文件描述符。*flags* 形参应当*os.O_APPEND*, *os.O_RDONLY* 和*os.O_TEXT* 按位 OR 的结果。返回的文件描述符可以被用作*os.fdopen()* 的形参以创建一个文件对象。

msvcrt.**get_osfhandle**(*fd*)
返回文件描述符 *fd* 的文件句柄。如果 *fd* 不能被识别则会引发*OSError*。

34.2.2 控制台 I/O

msvcrt.**kbhit**()
Return true if a keypress is waiting to be read.

msvcrt.**getch**()
读取一个按键并将结果字符返回为一个字节串。不会有内容回显到缝制台。如果还没有任何键被按下此调用将会阻塞，但它将不会等待 Enter 被按下。如果按下的键是一个特殊功能键，此函数将返回 '\000' 或 '\xe0'；下一次调用将返回键代码。Control-C 按钮无法使用此函数来读取。

msvcrt.**getwch**()
getch() 的宽字符版本，返回一个 Unicode 值。

msvcrt.**getche**()
类似于*getch()*，但按键如果表示一个可打印字符则它将被回显。

msvcrt.**getwche**()
getche() 的宽字符版本，返回一个 Unicode 值。

msvcrt.**putch**(*char*)
将字符串 *char* 打印到终端，不使用缓冲区。

msvcrt.**putwch**(*unicode_char*)
putch() 的宽字符版本，接受一个 Unicode 值。

msvcrt.**ungetch**(*char*)
使得字节串 *char* 被“推回”终端缓冲区；它将是被*getch()* 或*getche()* 读取的下一个字符。

msvcrt.**ungetwch**(*unicode_char*)
ungetch() 的宽字符版本，接受一个 Unicode 值。

34.2.3 其他函数

msvcrt.**heapmin**()
强制 malloc() 堆清空自身并将未使用的块返回给操作系统。失败时，这将引发*OSError*。

34.3 winreg —Windows 注册表访问

这些函数将 Windows 注册表 API 暴露给 Python。为了确保即便程序员忽略了显式关闭句柄，该句柄依然能够正确关闭，它使用了一个*handle 对象* 而不是整数来作为注册表句柄。

在 3.3 版更改: 该模块中的几个函数被用于引发*WindowsError*，该异常现在是*OSError* 的别名。

34.3.1 函数

该模块提供了下列函数：

winreg.**CloseKey**(*hkey*)
关闭之前打开的注册表键。参数 *hkey* 指之前打开的键。

> **注解：** 如果没有使用该方法关闭 *hkey* (或者通过*hkey.Close()*)，在对象 *hkey* 被 Python 销毁时会将其关闭。

winreg.**ConnectRegistry**(*computer_name*, *key*)
建立到另一台计算上上的预定义注册表句柄的连接，并返回一个*handle 对象*.

computer_name 是远程计算机的名称，以 `r"\\computername"` 的形式。如果是 `None`，将会使用本地计算机。

key 是所连接到的预定义句柄。

返回值是所开打键的句柄。如果函数失败，则引发一个*OSError* 异常。

在 3.3 版更改: 参考上文。

winreg.**CreateKey**(*key*, *sub_key*)
创建或打开特定的键，返回一个*handle 对象*。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 是用于命名该方法所打开或创建的键的字符串。

如果 *key* 是预定义键之一，*sub_key* 可能会是 `None`。该情况下，返回的句柄就是传入函数的句柄。

如果键已经存在，则该函数打开已经存在的该键。

返回值是所开打键的句柄。如果函数失败，则引发一个*OSError* 异常。

在 3.3 版更改: 参考上文。

winreg.**CreateKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)
创建或打开特定的键，返回一个*handle 对象*。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 是用于命名该方法所打开或创建的键的字符串。

reserved 是一个保留的证书，必须是零。默认值为零。

access 为一个整数，用于给键的预期安全访问指定访问掩码。默认值为*KEY_WRITE*。参阅*Access Rights* 了解其它允许值。

如果 *key* 是预定义键之一，*sub_key* 可能会是 `None`。该情况下，返回的句柄就是传入函数的句柄。

如果键已经存在，则该函数打开已经存在的该键。

返回值是所开打键的句柄。如果函数失败，则引发一个*OSError* 异常。

3.2 新版功能.

在 3.3 版更改: 参考上文。

winreg.**DeleteKey**(*key*, *sub_key*)
删除指定的键。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 这个字符串必须是由 *key* 参数所指定键的一个子项。该值项不可以是 None，同时键也不可以有子项。

该方法不能删除带有子项的键。

如果方法成功，则整个键，包括其所有值项都会被移除。如果方法失败，则引发一个*OSError* 异常。

在 3.3 版更改: 参考上文。

winreg.**DeleteKeyEx**(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)
删除指定的键。

注解: 函数*DeleteKeyEx()* 通过 RegDeleteKeyEx 这个 Windows API 函数实现，该函数为 Windows 的 64 位版本专属。参阅 RegDeleteKeyEx 文档。

---

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

sub_key 这个字符串必须是由 *key* 参数所指定键的一个子项。该值项不可以是 None，同时键也不可以有子项。

reserved 是一个保留的证书，必须是零。默认值为零。

access 为一个整数，用于给键的预期安全访问指定访问掩码。默认值为常量 _WOW64_64KEY 。参阅*Access Rights* 了解其它允许值。

该方法不能删除带有子项的键。

如果方法成功，则整个键，包括其所有值项都会被移除。如果方法失败，则引发一个*OSError* 异常。

在不支持的 Windows 版本之上，将会引发*NotImplementedError* 异常。

3.2 新版功能.

在 3.3 版更改: 参考上文。

winreg.**DeleteValue**(*key*, *value*)
从某个注册键中删除一个命名值项。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

value 为标识所要删除值项的字符串。

winreg.**EnumKey**(*key*, *index*)
列举某个已经打开注册表键的子项，并返回一个字符串。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

index 为一个整数，用于标识所获取键的索引。

每次调用该函数都会获取一个子项的名字。通常它会被反复调用，直到引发*OSError* 异常，这说明已经没有更多的可用值了。

在 3.3 版更改: 参考上文。

winreg.**EnumValue**(*key*, *index*)
列举某个已经打开注册表键的值项，并返回一个元组。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

index 为一个整数，用于标识要获取值项的索引。

每次调用该函数都会获取一个子项的名字。通常它会被反复调用，直到引发*OSError* 异常，这说明已经没有更多的可用值了。

结果为 3 元素的元组。

索引	含义
0	用于标识值项名称的字符串。
1	保存值项数据的对象，其类型取决于背后的注册表类型。
2	标识值项数据类型的整数。（请查阅*SetValueEx()* 文档中的表格）

在 3.3 版更改: 参考上文。

winreg.**ExpandEnvironmentStrings**(*str*)
Expands environment variable placeholders %NAME% in strings like *REG_EXPAND_SZ*:

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

winreg.**FlushKey**(*key*)
将某个键的所有属性写入注册表。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

It is not necessary to call *FlushKey()* to change a key. Registry changes are flushed to disk by the registry using its lazy flusher. Registry changes are also flushed to disk at system shutdown. Unlike *CloseKey()*, the *FlushKey()* method returns only when all the data has been written to the registry. An application should only call *FlushKey()* if it requires absolute certainty that registry changes are on disk.

注解: If you don't know whether a *FlushKey()* call is required, it probably isn't.

---

winreg.**LoadKey**(*key*, *sub_key*, *file_name*)
Creates a subkey under the specified key and stores registration information from a specified file into that subkey.

key is a handle returned by *ConnectRegistry()* or one of the constants *HKEY_USERS* or *HKEY_LOCAL_MACHINE*.

sub_key is a string that identifies the subkey to load.

file_name is the name of the file to load registry data from. This file must have been created with the *SaveKey()* function. Under the file allocation table (FAT) file system, the filename may not have an extension.

A call to *LoadKey()* fails if the calling process does not have the SE_RESTORE_PRIVILEGE privilege. Note that privileges are different from permissions –see the RegLoadKey documentation for more details.

If *key* is a handle returned by *ConnectRegistry()*, then the path specified in *file_name* is relative to the remote computer.

winreg.**OpenKey**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)
winreg.**OpenKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)
Opens the specified key, returning a *handle object*.

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

sub_key is a string that identifies the sub_key to open.

reserved is a reserved integer, and must be zero. The default is zero.

access is an integer that specifies an access mask that describes the desired security access for the key. Default is *KEY_READ*. See *Access Rights* for other allowed values.

The result is a new handle to the specified key.

If the function fails, *OSError* is raised.

在 3.2 版更改: Allow the use of named arguments.

在 3.3 版更改: 参考上文。

winreg.**QueryInfoKey**(*key*)
Returns information about a key, as a tuple.

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

结果为 3 元素的元组。

索引	含义
0	An integer giving the number of sub keys this key has.
1	An integer giving the number of values this key has.
2	An integer giving when the key was last modified (if available) as 100’s of nanoseconds since Jan 1, 1601.

winreg.**QueryValue**(*key*, *sub_key*)
Retrieves the unnamed value for a key, as a string.

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

sub_key is a string that holds the name of the subkey with which the value is associated. If this parameter is `None` or empty, the function retrieves the value set by the *`SetValue()`* method for the key identified by *key*.

Values in the registry have name, type, and data components. This method retrieves the data for a key’s first value that has a NULL name. But the underlying API call doesn’t return the type, so always use *`QueryValueEx()`* if possible.

winreg.**QueryValueEx**(*key*, *value_name*)
Retrieves the type and data for a specified value name associated with an open registry key.

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

value_name is a string indicating the value to query.

The result is a tuple of 2 items:

索引	含义
0	The value of the registry item.
1	An integer giving the registry type for this value (see table in docs for *`SetValueEx()`*)

winreg.**SaveKey**(*key*, *file_name*)
Saves the specified key, and all its subkeys to the specified file.

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

file_name is the name of the file to save registry data to. This file cannot already exist. If this filename includes an extension, it cannot be used on file allocation table (FAT) file systems by the *`LoadKey()`* method.

If *key* represents a key on a remote computer, the path described by *file_name* is relative to the remote computer. The caller of this method must possess the `SeBackupPrivilege` security privilege. Note that privileges are different than permissions –see the Conflicts Between User Rights and Permissions documentation for more details.

This function passes NULL for *security_attributes* to the API.

winreg.**SetValue**(*key*, *sub_key*, *type*, *value*)
Associates a value with a specified key.

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

sub_key is a string that names the subkey with which the value is associated.

type is an integer that specifies the type of the data. Currently this must be *REG_SZ*, meaning only strings are supported. Use the *SetValueEx()* function for support for other data types.

value is a string that specifies the new value.

If the key specified by the *sub_key* parameter does not exist, the SetValue function creates it.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

The key identified by the *key* parameter must have been opened with *KEY_SET_VALUE* access.

winreg.**SetValueEx**(*key*, *value_name*, *reserved*, *type*, *value*)
Stores data in the value field of an open registry key.

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

value_name is a string that names the subkey with which the value is associated.

reserved can be anything – zero is always passed to the API.

type is an integer that specifies the type of the data. See *Value Types* for the available types.

value is a string that specifies the new value.

This method can also set additional value and type information for the specified key. The key identified by the key parameter must have been opened with *KEY_SET_VALUE* access.

To open the key, use the *CreateKey()* or *OpenKey()* methods.

Value lengths are limited by available memory. Long values (more than 2048 bytes) should be stored as files with the filenames stored in the configuration registry. This helps the registry perform efficiently.

winreg.**DisableReflectionKey**(*key*)
Disables registry reflection for 32-bit processes running on a 64-bit operating system.

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

Will generally raise *NotImplemented* if executed on a 32-bit operating system.

If the key is not on the reflection list, the function succeeds but has no effect. Disabling reflection for a key does not affect reflection of any subkeys.

winreg.**EnableReflectionKey**(*key*)
Restores registry reflection for the specified disabled key.

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

Will generally raise *NotImplemented* if executed on a 32-bit operating system.

Restoring reflection for a key does not affect reflection of any subkeys.

winreg.**QueryReflectionKey**(*key*)
Determines the reflection state for the specified key.

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

Returns `True` if reflection is disabled.

Will generally raise *NotImplemented* if executed on a 32-bit operating system.

34.3.2 常量

The following constants are defined for use in many `_winreg` functions.

HKEY_* Constants

`winreg.`**`HKEY_CLASSES_ROOT`**
Registry entries subordinate to this key define types (or classes) of documents and the properties associated with those types. Shell and COM applications use the information stored under this key.

`winreg.`**`HKEY_CURRENT_USER`**
Registry entries subordinate to this key define the preferences of the current user. These preferences include the settings of environment variables, data about program groups, colors, printers, network connections, and application preferences.

`winreg.`**`HKEY_LOCAL_MACHINE`**
Registry entries subordinate to this key define the physical state of the computer, including data about the bus type, system memory, and installed hardware and software.

`winreg.`**`HKEY_USERS`**
Registry entries subordinate to this key define the default user configuration for new users on the local computer and the user configuration for the current user.

`winreg.`**`HKEY_PERFORMANCE_DATA`**
Registry entries subordinate to this key allow you to access performance data. The data is not actually stored in the registry; the registry functions cause the system to collect the data from its source.

`winreg.`**`HKEY_CURRENT_CONFIG`**
Contains information about the current hardware profile of the local computer system.

`winreg.`**`HKEY_DYN_DATA`**
This key is not used in versions of Windows after 98.

Access Rights

For more information, see Registry Key Security and Access.

`winreg.`**`KEY_ALL_ACCESS`**
Combines the STANDARD_RIGHTS_REQUIRED, *KEY_QUERY_VALUE*, *KEY_SET_VALUE*, *KEY_CREATE_SUB_KEY*, *KEY_ENUMERATE_SUB_KEYS*, *KEY_NOTIFY*, and *KEY_CREATE_LINK* access rights.

`winreg.`**`KEY_WRITE`**
Combines the STANDARD_RIGHTS_WRITE, *KEY_SET_VALUE*, and *KEY_CREATE_SUB_KEY* access rights.

`winreg.`**`KEY_READ`**
Combines the STANDARD_RIGHTS_READ, *KEY_QUERY_VALUE*, *KEY_ENUMERATE_SUB_KEYS*, and *KEY_NOTIFY* values.

`winreg.`**`KEY_EXECUTE`**
Equivalent to *KEY_READ*.

`winreg.`**`KEY_QUERY_VALUE`**
Required to query the values of a registry key.

`winreg.`**`KEY_SET_VALUE`**
Required to create, delete, or set a registry value.

winreg.**KEY_CREATE_SUB_KEY**
Required to create a subkey of a registry key.

winreg.**KEY_ENUMERATE_SUB_KEYS**
Required to enumerate the subkeys of a registry key.

winreg.**KEY_NOTIFY**
Required to request change notifications for a registry key or for subkeys of a registry key.

winreg.**KEY_CREATE_LINK**
Reserved for system use.

64-bit Specific

For more information, see Accessing an Alternate Registry View.

winreg.**KEY_WOW64_64KEY**
Indicates that an application on 64-bit Windows should operate on the 64-bit registry view.

winreg.**KEY_WOW64_32KEY**
Indicates that an application on 64-bit Windows should operate on the 32-bit registry view.

Value Types

For more information, see Registry Value Types.

winreg.**REG_BINARY**
Binary data in any form.

winreg.**REG_DWORD**
32-bit number.

winreg.**REG_DWORD_LITTLE_ENDIAN**
A 32-bit number in little-endian format. Equivalent to *REG_DWORD*.

winreg.**REG_DWORD_BIG_ENDIAN**
A 32-bit number in big-endian format.

winreg.**REG_EXPAND_SZ**
Null-terminated string containing references to environment variables (`%PATH%`).

winreg.**REG_LINK**
A Unicode symbolic link.

winreg.**REG_MULTI_SZ**
A sequence of null-terminated strings, terminated by two null characters. (Python handles this termination automatically.)

winreg.**REG_NONE**
No defined value type.

winreg.**REG_QWORD**
A 64-bit number.

3.6 新版功能.

winreg.**REG_QWORD_LITTLE_ENDIAN**
A 64-bit number in little-endian format. Equivalent to *REG_QWORD*.

3.6 新版功能.

winreg.**REG_RESOURCE_LIST**
A device-driver resource list.

winreg.**REG_FULL_RESOURCE_DESCRIPTOR**
A hardware setting.

winreg.**REG_RESOURCE_REQUIREMENTS_LIST**
A hardware resource list.

winreg.**REG_SZ**
A null-terminated string.

34.3.3 Registry Handle Objects

This object wraps a Windows HKEY object, automatically closing it when the object is destroyed. To guarantee cleanup, you can call either the *Close()* method on the object, or the *CloseKey()* function.

All registry functions in this module return one of these objects.

All registry functions in this module which accept a handle object also accept an integer, however, use of the handle object is encouraged.

Handle objects provide semantics for __bool__() –thus

```
if handle:
    print("Yes")
```

will print Yes if the handle is currently valid (has not been closed or detached).

The object also support comparison semantics, so handle objects will compare true if they both reference the same underlying Windows handle value.

Handle objects can be converted to an integer (e.g., using the built-in *int()* function), in which case the underlying Windows handle value is returned. You can also use the *Detach()* method to return the integer handle, and also disconnect the Windows handle from the handle object.

PyHKEY.**Close**()
Closes the underlying Windows handle.

If the handle is already closed, no error is raised.

PyHKEY.**Detach**()
Detaches the Windows handle from the handle object.

The result is an integer that holds the value of the handle before it is detached. If the handle is already detached or closed, this will return zero.

After calling this function, the handle is effectively invalidated, but the handle is not closed. You would call this function when you need the underlying Win32 handle to exist beyond the lifetime of the handle object.

PyHKEY.**__enter__**()
PyHKEY.**__exit__**(**exc_info*)
The HKEY object implements __enter__() and __exit__() and thus supports the context protocol for the with statement:

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

will automatically close *key* when control leaves the with block.

34.4 winsound —Sound-playing interface for Windows

The *winsound* module provides access to the basic sound-playing machinery provided by Windows platforms. It includes functions and several constants.

winsound.**Beep**(*frequency*, *duration*)
Beep the PC’ s speaker. The *frequency* parameter specifies frequency, in hertz, of the sound, and must be in the range 37 through 32,767. The *duration* parameter specifies the number of milliseconds the sound should last. If the system is not able to beep the speaker, *RuntimeError* is raised.

winsound.**PlaySound**(*sound*, *flags*)
Call the underlying `PlaySound()` function from the Platform API. The *sound* parameter may be a filename, a system sound alias, audio data as a *bytes-like object*, or `None`. Its interpretation depends on the value of *flags*, which can be a bitwise ORed combination of the constants described below. If the *sound* parameter is `None`, any currently playing waveform sound is stopped. If the system indicates an error, *RuntimeError* is raised.

winsound.**MessageBeep**(*type=MB_OK*)
Call the underlying `MessageBeep()` function from the Platform API. This plays a sound as specified in the registry. The *type* argument specifies which sound to play; possible values are `-1`, `MB_ICONASTERISK`, `MB_ICONEXCLAMATION`, `MB_ICONHAND`, `MB_ICONQUESTION`, and `MB_OK`, all described below. The value `-1` produces a “simple beep”; this is the final fallback if a sound cannot be played otherwise. If the system indicates an error, *RuntimeError* is raised.

winsound.**SND_FILENAME**
The *sound* parameter is the name of a WAV file. Do not use with *SND_ALIAS*.

winsound.**SND_ALIAS**
The *sound* parameter is a sound association name from the registry. If the registry contains no such name, play the system default sound unless *SND_NODEFAULT* is also specified. If no default sound is registered, raise *RuntimeError*. Do not use with *SND_FILENAME*.

All Win32 systems support at least the following; most systems support many more:

PlaySound() name	Corresponding Control Panel Sound name
`'SystemAsterisk'`	Asterisk
`'SystemExclamation'`	Exclamation
`'SystemExit'`	Exit Windows
`'SystemHand'`	Critical Stop
`'SystemQuestion'`	Question

例如

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.**SND_LOOP**
Play the sound repeatedly. The *SND_ASYNC* flag must also be used to avoid blocking. Cannot be used with *SND_MEMORY*.

winsound.**SND_MEMORY**
The *sound* parameter to `PlaySound()` is a memory image of a WAV file, as a *bytes-like object*.

> **注解：** This module does not support playing from a memory image asynchronously, so a combination of this flag and `SND_ASYNC` will raise `RuntimeError`.

winsound.**SND_PURGE**
Stop playing all instances of the specified sound.

> **注解：** This flag is not supported on modern Windows platforms.

winsound.**SND_ASYNC**
Return immediately, allowing sounds to play asynchronously.

winsound.**SND_NODEFAULT**
If the specified sound cannot be found, do not play the system default sound.

winsound.**SND_NOSTOP**
Do not interrupt sounds currently playing.

winsound.**SND_NOWAIT**
Return immediately if the sound driver is busy.

> **注解：** This flag is not supported on modern Windows platforms.

winsound.**MB_ICONASTERISK**
Play the `SystemDefault` sound.

winsound.**MB_ICONEXCLAMATION**
Play the `SystemExclamation` sound.

winsound.**MB_ICONHAND**
Play the `SystemHand` sound.

winsound.**MB_ICONQUESTION**
Play the `SystemQuestion` sound.

winsound.**MB_OK**
Play the `SystemDefault` sound.

CHAPTER 35

Unix 专有服务

本章描述的模块提供了 Unix 操作系统独有特性的接口，在某些情况下也适用于它的某些或许多衍生版。以下为模块概览：

35.1 posix —最常见的 POSIX 系统调用

此模块提供了对基于 C 标准和 POSIX 标准（一种稍加修改的 Unix 接口）进行标准化的系统功能的访问。

请勿直接导入此模块。而应导入*os*模块，它提供了此接口的 可移植版本。在 Unix 上，*os*模块提供了*posix*接口的一个超集。在非 Unix 操作系统上*posix*模块将不可用，但会通过*os*接口提供它的一个可用子集。一旦导入了*os*，用它替代*posix*时就 没有性能惩罚。此外，*os*还提供了一些附加功能，例如在 os.environ 中的某个条目被修改时会自动调用*putenv()*。

错误将作为异常被报告；对于类型错误会给出普通异常，而系统调用所报告的异常则会引发*OSError*。

35.1.1 大文件支持

某些操作系统（包括 AIX, HP-UX, Irix 和 Solaris）可对 int 和 long 为 32 位值的 C 编程模型提供大于 2 GiB 的文件的支持。这在通常情况下是以将相关数据长度和偏移类型定义为 64 位值的方式来实现的。这样的文件有时被称为 大文件。

Large file support is enabled in Python when the size of an off_t is larger than a long and the long long type is available and is at least as large as an off_t. It may be necessary to configure and compile Python with certain compiler flags to enable this mode. For example, it is enabled by default with recent versions of Irix, but with Solaris 2.6 and 2.7 you need to do something like:

```
CFLAGS="`getconf LFS_CFLAGS`" OPT="-g -O2 $CFLAGS" \
        ./configure
```

在支持大文件的 Linux 系统中，可以这样做：

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

35.1.2 重要的模块内容

除了*os*模块文档已说明的许多函数，*posix*还定义了下列数据项:

posix.**environ**
一个表示解释器启动时间点的字符串环境的字典。键和值的类型在 Unix 上为 bytes 而在 Windows 上为 str。例如，environ[b'HOME'] (Windows 上的 environ['HOME']) 是你的家目录的路径名，等价于 C 中的 getenv("HOME")。

修改此字典不会影响由*execv()*，*popen()* 或*system()* 所传入的字符串环境；如果你需要修改环境，请将 environ 传给*execve()* 或者为*system()* 或*popen()* 的命令字符串添加变量赋值和 export 语句。

在 3.2 版更改: 在 Unix 上，键和值为 bytes 类型。

注解: *os*模块提供了对 environ 的替代实现，它会在被修改时更新环境。还要注意更新*os.environ*将导致此字典失效。推荐使用这个*os*模块版本而不是直接访问*posix*模块。

35.2 pwd —用户密码数据库

此模块可以访问 Unix 用户账户名及密码数据库，在所有 Unix 版本上均可使用。

密码数据库中的条目以元组对象返回，属性对应 passwd 中的结构（属性如下所示，可参考 <pwd.h>):

索引	属性	含义
0	pw_name	登录名
1	pw_passwd	密码，可能已经加密
2	pw_uid	用户 ID 数值
3	pw_gid	组 ID 数值
4	pw_gecos	用户名或备注
5	pw_dir	用户主目录
6	pw_shell	用户的命令解释器

其中 uid 和 gid 是整数，其他是字符串，如果找不到对应的项目，抛出*KeyError*异常。

注解: 传统的 Unix 系统中，pw_passwd 的值通常使用 DES 导出的算法加密（参阅*crypt*模块）。不过现在的 unix 系统使用 *影子密码系统*。在这些 unix 上，*pw_passwd* 只包含星号（'*'）或字母（'x'），而加密的密码存储在文件 /etc/shadow 中，此文件不是全局可读的。在 *pw_passwd* 中是否包含有用信息是系统相关的。如果可以访问到加密的密码，就需要使用*spwd*模块了。

本模块定义如下内容:

pwd.**getpwuid**(*uid*)
给定用户的数值 ID，返回密码数据库的对应项目。

pwd.**getpwnam**(*name*)
给定用户名，返回密码数据库的对应项目。

pwd.**getpwall**()
返回密码数据库中所有项目的列表，顺序不是固定的。

参见:

模块*grp* 针对用户组数据库的接口，与本模块类似。

模块*spwd* 针对影子密码数据库的接口，与本模块类似。

35.3 spwd —The shadow password database

This module provides access to the Unix shadow password database. It is available on various Unix versions.

You must have enough privileges to access the shadow password database (this usually means you have to be root).

Shadow password database entries are reported as a tuple-like object, whose attributes correspond to the members of the spwd structure (Attribute field below, see <shadow.h>):

索引	属性	含义
0	sp_namp	登录名
1	sp_pwdp	Encrypted password
2	sp_lstchg	Date of last change
3	sp_min	Minimal number of days between changes
4	sp_max	Maximum number of days between changes
5	sp_warn	Number of days before password expires to warn user about it
6	sp_inact	Number of days after password expires until account is disabled
7	sp_expire	Number of days since 1970-01-01 when account expires
8	sp_flag	Reserved

The sp_namp and sp_pwdp items are strings, all others are integers. *KeyError* is raised if the entry asked for cannot be found.

定义了以下函数:

spwd.**getspnam**(*name*)
Return the shadow password database entry for the given user name.

在 3.6 版更改: Raises a *PermissionError* instead of *KeyError* if the user doesn't have privileges.

spwd.**getspall**()
Return a list of all available shadow password database entries, in arbitrary order.

参见:

模块*grp* 针对用户组数据库的接口，与本模块类似。

模块*pwd* An interface to the normal password database, similar to this.

35.4 grp —组数据库

该模块提供对 Unix 组数据库的访问。它在所有 Unix 版本上都可用。

组数据库条目被报告为类似元组的对象，其属性对应于 group 结构的成员（下面的属性字段，请参见 <pwd.h> ）：

索引	属性	含义
0	gr_name	组名
1	gr_passwd	（加密的）组密码；通常为空
2	gr_gid	数字组 ID
3	gr_mem	组内所有成员的用户名

gid 是整数，名称和密码是字符串，成员列表是字符串列表。（注意，大多数用户未根据密码数据库显式列为所属组的成员。请检查两个数据库以获取完整的成员资格信息。还要注意，以 + 或 - 开头的 gr_name 可能是 YP/NIS 引用，可能无法通过*getgrnam()* 或*getgrgid()* 访问。）

本模块定义如下内容：

grp.**getgrgid**(*gid*)
返回给定数字组 ID 的组数据库条目。如果请求的条目无法找到则会引发*KeyError*。

3.6 版后已移除: 从 Python 3.6 开始，不再支持对*getgrgid()* 中的 float 或 string 等非 integer 参数的支持。

grp.**getgrnam**(*name*)
返回给定组名的组数据库条目。如果找不到要求的条目，则会引发*KeyError* 错误。

grp.**getgrall**()
以任意顺序返回所有可用组条目的列表。

参见:

模块*pwd* 用户数据库的接口，与此类似。

模块*spwd* 针对影子密码数据库的接口，与本模块类似。

35.5 crypt —Function to check Unix passwords

Source code: Lib/crypt.py

This module implements an interface to the *crypt(3)* routine, which is a one-way hash function based upon a modified DES algorithm; see the Unix man page for further details. Possible uses include storing hashed passwords so you can check passwords without storing the actual password, or attempting to crack Unix passwords with a dictionary.

Notice that the behavior of this module depends on the actual implementation of the *crypt(3)* routine in the running system. Therefore, any extensions available on the current implementation will also be available on this module.

35.5.1 Hashing Methods

3.3 新版功能.

The *crypt* module defines the list of hashing methods (not all methods are available on all platforms):

`crypt.`**`METHOD_SHA512`**
A Modular Crypt Format method with 16 character salt and 86 character hash. This is the strongest method.

`crypt.`**`METHOD_SHA256`**
Another Modular Crypt Format method with 16 character salt and 43 character hash.

`crypt.`**`METHOD_MD5`**
Another Modular Crypt Format method with 8 character salt and 22 character hash.

`crypt.`**`METHOD_CRYPT`**
The traditional method with a 2 character salt and 13 characters of hash. This is the weakest method.

35.5.2 Module Attributes

3.3 新版功能.

`crypt.`**`methods`**
A list of available password hashing algorithms, as `crypt.METHOD_*` objects. This list is sorted from strongest to weakest.

35.5.3 模块函数

The *crypt* module defines the following functions:

`crypt.`**`crypt`**(*word*, *salt=None*)
word will usually be a user's password as typed at a prompt or in a graphical interface. The optional *salt* is either a string as returned from *mksalt()*, one of the `crypt.METHOD_*` values (though not all may be available on all platforms), or a full encrypted password including salt, as returned by this function. If *salt* is not provided, the strongest method will be used (as returned by *methods()*).

Checking a password is usually done by passing the plain-text password as *word* and the full results of a previous *crypt()* call, which should be the same as the results of this call.

salt (either a random 2 or 16 character string, possibly prefixed with `$digit$` to indicate the method) which will be used to perturb the encryption algorithm. The characters in *salt* must be in the set `[./a-zA-Z0-9]`, with the exception of Modular Crypt Format which prefixes a `$digit$`.

Returns the hashed password as a string, which will be composed of characters from the same alphabet as the salt.

Since a few *crypt(3)* extensions allow different values, with different sizes in the *salt*, it is recommended to use the full crypted password as salt when checking for a password.

在 3.3 版更改: Accept `crypt.METHOD_*` values in addition to strings for *salt*.

`crypt.`**`mksalt`**(*method=None*)
Return a randomly generated salt of the specified method. If no *method* is given, the strongest method available as returned by *methods()* is used.

The return value is a string either of 2 characters in length for `crypt.METHOD_CRYPT`, or 19 characters starting with `$digit$` and 16 random characters from the set `[./a-zA-Z0-9]`, suitable for passing as the *salt* argument to *crypt()*.

3.3 新版功能.

35.5.4 例子

A simple example illustrating typical use (a constant-time comparison operation is needed to limit exposure to timing attacks. *hmac.compare_digest()* is suitable for this purpose):

```
import pwd
import crypt
import getpass
from hmac import compare_digest as compare_hash

def login():
    username = input('Python login: ')
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise ValueError('no support for shadow passwords')
        cleartext = getpass.getpass()
        return compare_hash(crypt.crypt(cleartext, cryptedpasswd), cryptedpasswd)
    else:
        return True
```

To generate a hash of a password using the strongest available method and check it against the original:

```
import crypt
from hmac import compare_digest as compare_hash

hashed = crypt.crypt(plaintext)
if not compare_hash(hashed, crypt.crypt(plaintext, hashed)):
    raise ValueError("hashed version doesn't validate against original")
```

35.6 termios —POSIX 风格的 tty 控制

此模块提供了针对 tty I/O 控制的 POSIX 调用的接口。有关此类调用的完整描述，请参阅 *termios(3)* Unix 指南页。它仅在当安装时配置了支持 POSIX *termios* 风格的 tty I/O 控制的 Unix 版本上可用。

此模块中的所有函数均接受一个文件描述符 *fd* 作为第一个参数。这可以是一个整数形式的文件描述符，例如 `sys.stdin.fileno()` 所返回的对象，或是一个*file object*，例如 `sys.stdin` 本身。

这个模块还定义了与此处所提供的函数一起使用的所有必要的常量；这些常量与它们在 C 中的对应常量同名。请参考你的系统文档了解有关如何使用这些终端控制接口的更多信息。

这个模块定义了以下函数：

`termios.`**`tcgetattr`**(*fd*)
对于文件描述符 *fd* 返回一个包含 tty 属性的列表，形式如下：`[iflag, oflag, cflag, lflag, ispeed, ospeed, cc]`，其中 *cc* 为一个包含 tty 特殊字符的列表（每一项都是长度为 1 的字符串，索引号为 `VMIN` 和 `VTIME` 的项除外，这些字段如有定义则应为整数）。对旗标和速度以及 *cc* 数组中索引的解读必须使用在*termios* 模块中定义的符号常量来完成。

`termios.`**`tcsetattr`**(*fd*, *when*, *attributes*)
根据 *attributes* 列表设置文件描述符 *fd* 的 tty 属性，该列表即*tcgetattr()* 所返回的对象。*when* 参数确定何时改变属性：`TCSANOW` 表示立即改变，`TCSADRAIN` 表示在传输所有队列输出后再改变，或 `TCSAFLUSH` 表示在传输所有队列输出并丢失所有队列输入后再改变。

termios.**tcsendbreak**(*fd*, *duration*)
在文件描述符 *fd* 上发送一个中断。*duration* 为零表示发送时长为 0.25–0.5 秒的中断；*duration* 非零值的含义取决于具体系统。

termios.**tcdrain**(*fd*)
进入等待状态直到写入文件描述符 *fd* 的所有输出都传送完毕。

termios.**tcflush**(*fd*, *queue*)
在文件描述符 *fd* 上丢弃队列数据。*queue* 选择器指定哪个队列: TCIFLUSH 表示输入队列，TCOFLUSH 表示输出队列，或 TCIOFLUSH 表示两个队列同时。

termios.**tcflow**(*fd*, *action*)
在文件描述符 *fd* 上挂起一战恢复输入或输出。*action* 参数可以为 TCOOFF 表示挂起输出，TCOON 表示重启输出，TCIOFF 表示挂起输入，或 TCION 表示重启输入。

参见:

模块*tty* 针对常用终端控制操作的便捷函数。

35.6.1 示例

这个函数可提示输入密码并且关闭回显。请注意其采取的技巧是使用一个单独的*tcgetattr()* 调用和一个 try …finally 语句来确保旧的 tty 属性无论在何种情况下都会被原样保存:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
        passwd = input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

35.7 tty —终端控制功能

Source code: Lib/tty.py

---

tty 模块定义了将 tty 放入 cbreak 和 raw 模式的函数。

因为它需要*termios* 模块，所以只能在 Unix 上运行。

tty 模块定义了以下函数:

tty.**setraw**(*fd*, *when=termios.TCSAFLUSH*)
将文件描述符 *fd* 的模式更改为 raw 。如果 *when* 被省略，则默认为 termios.TCSAFLUSH ，并传递给*termios.tcsetattr()* 。

tty.**setcbreak**(*fd*, *when=termios.TCSAFLUSH*)
将文件描述符 *fd* 的模式更改为 cbreak 。如果 *when* 被省略，则默认为 termios.TCSAFLUSH ，并传递给*termios.tcsetattr()* 。

参见:

模块*termios* 低级终端控制接口。

35.8 pty —伪终端工具

源代码: Lib/pty.py

---

pty 模块定义了一些处理“伪终端”概念的操作：启动另一个进程并能以程序方式在其控制终端中进行读写。

由于伪终端处理高度依赖于具体平台，因此此功能只有针对 Linux 的代码。(Linux 代码也可在其他平台上工作，但是未经测试。)

pty 模块定义了下列函数:

pty.**fork**()
: 分叉。将子进程的控制终端连接到一个伪终端。返回值为 `(pid, fd)`。请注意子进程获得 *pid* 0 而 *fd* 为 *invalid*。父进程返回值为子进程的 *pid* 而 *fd* 为一个连接到子进程的控制终端（并同时连接到子进程的标准输入和输出）的文件描述符。

pty.**openpty**()
: 打开一个新的伪终端对，如果可能将使用*os.openpty()*，或是针对通用 Unix 系统的模拟代码。返回一个文件描述符对 `(master, slave)`，分别表示主从两端。

pty.**spawn**(*argv*[, *master_read*[, *stdin_read*]])
: Spawn a process, and connect its controlling terminal with the current process's standard io. This is often used to baffle programs which insist on reading from the controlling terminal.

  The functions *master_read* and *stdin_read* should be functions which read from a file descriptor. The defaults try to read 1024 bytes each time they are called.

  在 3.4 版更改: *spawn()* 现在会从子进程的*os.waitpid()* 返回状态值。

35.8.1 示例

以下程序的作用类似于 Unix 命令 *script(1)*，它使用一个伪终端来记录一个“typescript”里终端进程的所有输入和输出:

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
```

(下页继续)

(续上页)

```
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

35.9 fcntl —The fcntl and ioctl system calls

This module performs file control and I/O control on file descriptors. It is an interface to the `fcntl()` and `ioctl()` Unix routines. For a complete description of these calls, see *fcntl(2)* and *ioctl(2)* Unix manual pages.

All functions in this module take a file descriptor *fd* as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or an *io.IOBase* object, such as `sys.stdin` itself, which provides a *fileno()* that returns a genuine file descriptor.

在 3.3 版更改: Operations in this module used to raise an *IOError* where they now raise an *OSError*.

这个模块定义了以下函数:

`fcntl.`**`fcntl`**(*fd*, *cmd*, *arg=0*)

Perform the operation *cmd* on file descriptor *fd* (file objects providing a *fileno()* method are accepted as well). The values used for *cmd* are operating system dependent, and are available as constants in the *fcntl* module, using the same names as used in the relevant C header files. The argument *arg* can either be an integer value, or a *bytes* object. With an integer value, the return value of this function is the integer return value of the C `fcntl()` call. When the argument is bytes it represents a binary structure, e.g. created by *struct.pack()*. The binary data is copied to a buffer whose address is passed to the C `fcntl()` call. The return value after a successful call is the contents of the buffer, converted to a *bytes* object. The length of the returned object will be the same as the length of the *arg* argument. This is limited to 1024 bytes. If the information returned in the buffer by the operating system is larger than 1024 bytes, this is most likely to result in a segmentation violation or a more subtle data corruption.

If the `fcntl()` fails, an *OSError* is raised.

`fcntl.`**`ioctl`**(*fd*, *request*, *arg=0*, *mutate_flag=True*)

This function is identical to the *fcntl()* function, except that the argument handling is even more complicated.

The *request* parameter is limited to values that can fit in 32-bits. Additional constants of interest for use as the *request* argument can be found in the *termios* module, under the same names as used in the relevant C header files.

The parameter *arg* can be one of an integer, an object supporting the read-only buffer interface (like *bytes*) or an object supporting the read-write buffer interface (like *bytearray*).

In all but the last case, behaviour is as for the *fcntl()* function.

If a mutable buffer is passed, then the behaviour is determined by the value of the *mutate_flag* parameter.

If it is false, the buffer's mutability is ignored and behaviour is as for a read-only buffer, except that the 1024 byte limit mentioned above is avoided –so long as the buffer you pass is at least as long as what the operating system wants to put there, things should work.

If *mutate_flag* is true (the default), then the buffer is (in effect) passed to the underlying *ioctl()* system call, the latter's return code is passed back to the calling Python, and the buffer's new contents reflect the action of the *ioctl()*. This is a slight simplification, because if the supplied buffer is less than 1024 bytes long it is first copied into a static buffer 1024 bytes long which is then passed to *ioctl()* and copied back into the supplied buffer.

If the `ioctl()` fails, an *OSError* exception is raised.

举个例子:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

`fcntl.`**`flock`**(*fd*, *operation*)

Perform the lock operation *operation* on file descriptor *fd* (file objects providing a *fileno()* method are accepted as well). See the Unix manual *flock(2)* for details. (On some systems, this function is emulated using `fcntl()`.)

If the `flock()` fails, an *OSError* exception is raised.

`fcntl.`**`lockf`**(*fd*, *cmd*, *len=0*, *start=0*, *whence=0*)

This is essentially a wrapper around the *fcntl()* locking calls. *fd* is the file descriptor of the file to lock or unlock, and *cmd* is one of the following values:

- `LOCK_UN` –unlock
- `LOCK_SH` –acquire a shared lock
- `LOCK_EX` –acquire an exclusive lock

When *cmd* is `LOCK_SH` or `LOCK_EX`, it can also be bitwise ORed with `LOCK_NB` to avoid blocking on lock acquisition. If `LOCK_NB` is used and the lock cannot be acquired, an *OSError* will be raised and the exception will have an *errno* attribute set to `EACCES` or `EAGAIN` (depending on the operating system; for portability, check for both values). On at least some systems, `LOCK_EX` can only be used if the file descriptor refers to a file opened for writing.

len is the number of bytes to lock, *start* is the byte offset at which the lock starts, relative to *whence*, and *whence* is as with *io.IOBase.seek()*, specifically:

- `0` –relative to the start of the file (*os.SEEK_SET*)
- `1` –relative to the current buffer position (*os.SEEK_CUR*)
- `2` –relative to the end of the file (*os.SEEK_END*)

The default for *start* is 0, which means to start at the beginning of the file. The default for *len* is 0 which means to lock to the end of the file. The default for *whence* is also 0.

Examples (all on a SVR4 compliant system):

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

Note that in the first example the return value variable *rv* will hold an integer value; in the second example it will hold a *bytes* object. The structure lay-out for the *lockdata* variable is system dependent —therefore using the *flock()* call may be better.

参见:

模块 *os* If the locking flags *O_SHLOCK* and *O_EXLOCK* are present in the *os* module (on BSD only), the *os.open()* function provides an alternative to the *lockf()* and *flock()* functions.

35.10 pipes —终端管道接口

源代码: Lib/pipes.py

---

pipes 定义了一个类用来抽象 *pipeline* 的概念—将数据从一个文件转到另一文件的转换器序列。

由于模块使用了 **/bin/sh** 命令行，因此要求有 POSIX 或兼容*os.system()* 和*os.popen()* 的终端程序。

pipes 模块定义了以下的类:

class pipes.**Template**
对 pipeline 的抽象。

示例:

```
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```

35.10.1 模板对象

模板对象有以下方法:

Template.**reset**()
将一个管道模板恢复为初始状态。

Template.**clone**()
返回一个新的等价的管道模板。

Template.**debug**(*flag*)
如果 *flag* 为真值，则启用调试。否则禁用调试。当启用调试时，要执行的命令会被打印出来，并且会给予终端 `set -x` 命令以输出更详细的信息。

Template.**append**(*cmd*, *kind*)
在末尾添加一个新的动作。*cmd* 变量必须为一个有效的 bourne 终端命令。*kind* 变量由两个字母组成。

第一个字母可以为 '-' (这表示命令将读取其标准输入), 'f' (这表示命令将读取在命令行中给定的文件) 或 '.' (这表示命令将不读取输入，因而必须放在前面。)

类似地，第二个字母可以为 '-' (这表示命令将写入到标准输出), 'f' (这表示命令将写入在命令行中给定的文件) 或 '.' (这表示命令将不执行写入，因而必须放在末尾。)

Template.**prepend**(*cmd*, *kind*)
在开头添加一个新的动作。请参阅*append()* 获取相应参数的说明。

Template.**open**(*file*, *mode*)
返回一个文件类对象，打开到 *file*，但是将从管道读取或写入。请注意只能给出 'r', 'w' 中的一个。

Template.**copy**(*infile*, *outfile*)
通过管道将 *infile* 拷贝到 *outfile*。

35.11 resource —Resource usage information

This module provides basic mechanisms for measuring and controlling system resources utilized by a program.

Symbolic constants are used to specify particular system resources and to request usage information about either the current process or its children.

An *OSError* is raised on syscall failure.

exception resource.**error**
一个被弃用的*OSError* 的别名。

在 3.3 版更改: 根据 **PEP 3151**，这个类是*OSError* 的别名。

35.11.1 Resource Limits

Resources usage can be limited using the *setrlimit()* function described below. Each resource is controlled by a pair of limits: a soft limit and a hard limit. The soft limit is the current limit, and may be lowered or raised by a process over time. The soft limit can never exceed the hard limit. The hard limit can be lowered to any value greater than the soft limit, but not raised. (Only processes with the effective UID of the super-user can raise a hard limit.)

The specific resources that can be limited are system dependent. They are described in the *getrlimit(2)* man page. The resources listed below are supported when the underlying operating system supports them; resources which cannot be checked or controlled by the operating system are not defined in this module for those platforms.

resource.**RLIM_INFINITY**
Constant used to represent the limit for an unlimited resource.

resource.**getrlimit**(*resource*)
Returns a tuple (soft, hard) with the current soft and hard limits of *resource*. Raises *ValueError* if an invalid resource is specified, or *error* if the underlying system call fails unexpectedly.

resource.**setrlimit**(*resource*, *limits*)
Sets new limits of consumption of *resource*. The *limits* argument must be a tuple (soft, hard) of two integers describing the new limits. A value of *RLIM_INFINITY* can be used to request a limit that is unlimited.

Raises *ValueError* if an invalid resource is specified, if the new soft limit exceeds the hard limit, or if a process tries to raise its hard limit. Specifying a limit of *RLIM_INFINITY* when the hard or system limit for that resource

is not unlimited will result in a *ValueError*. A process with the effective UID of super-user can request any valid limit value, including unlimited, but *ValueError* will still be raised if the requested limit exceeds the system imposed limit.

`setrlimit` may also raise *error* if the underlying system call fails.

`resource.`**`prlimit`**(*pid*, *resource*[, *limits*])

Combines *setrlimit()* and *getrlimit()* in one function and supports to get and set the resources limits of an arbitrary process. If *pid* is 0, then the call applies to the current process. *resource* and *limits* have the same meaning as in *setrlimit()*, except that *limits* is optional.

When *limits* is not given the function returns the *resource* limit of the process *pid*. When *limits* is given the *resource* limit of the process is set and the former resource limit is returned.

Raises *ProcessLookupError* when *pid* can't be found and *PermissionError* when the user doesn't have `CAP_SYS_RESOURCE` for the process.

Availability: Linux 2.6.36 or later with glibc 2.13 or later

3.4 新版功能.

These symbols define resources whose consumption can be controlled using the *setrlimit()* and *getrlimit()* functions described below. The values of these symbols are exactly the constants used by C programs.

The Unix man page for *getrlimit(2)* lists the available resources. Note that not all systems use the same symbol or same value to denote the same resource. This module does not attempt to mask platform differences —symbols not defined for a platform will not be available from this module on that platform.

`resource.`**`RLIMIT_CORE`**

The maximum size (in bytes) of a core file that the current process can create. This may result in the creation of a partial core file if a larger core would be required to contain the entire process image.

`resource.`**`RLIMIT_CPU`**

The maximum amount of processor time (in seconds) that a process can use. If this limit is exceeded, a `SIGXCPU` signal is sent to the process. (See the *signal* module documentation for information about how to catch this signal and do something useful, e.g. flush open files to disk.)

`resource.`**`RLIMIT_FSIZE`**

The maximum size of a file which the process may create.

`resource.`**`RLIMIT_DATA`**

The maximum size (in bytes) of the process's heap.

`resource.`**`RLIMIT_STACK`**

The maximum size (in bytes) of the call stack for the current process. This only affects the stack of the main thread in a multi-threaded process.

`resource.`**`RLIMIT_RSS`**

The maximum resident set size that should be made available to the process.

`resource.`**`RLIMIT_NPROC`**

The maximum number of processes the current process may create.

`resource.`**`RLIMIT_NOFILE`**

The maximum number of open file descriptors for the current process.

`resource.`**`RLIMIT_OFILE`**

The BSD name for *RLIMIT_NOFILE*.

`resource.`**`RLIMIT_MEMLOCK`**

The maximum address space which may be locked in memory.

resource.**RLIMIT_VMEM**
The largest area of mapped memory which the process may occupy.

resource.**RLIMIT_AS**
The maximum area (in bytes) of address space which may be taken by the process.

resource.**RLIMIT_MSGQUEUE**
The number of bytes that can be allocated for POSIX message queues.

Availability: Linux 2.6.8 or later.

3.4 新版功能.

resource.**RLIMIT_NICE**
The ceiling for the process’ s nice level (calculated as 20 - rlim_cur).

Availability: Linux 2.6.12 or later.

3.4 新版功能.

resource.**RLIMIT_RTPRIO**
The ceiling of the real-time priority.

Availability: Linux 2.6.12 or later.

3.4 新版功能.

resource.**RLIMIT_RTTIME**
The time limit (in microseconds) on CPU time that a process can spend under real-time scheduling without making a blocking syscall.

Availability: Linux 2.6.25 or later.

3.4 新版功能.

resource.**RLIMIT_SIGPENDING**
The number of signals which the process may queue.

Availability: Linux 2.6.8 or later.

3.4 新版功能.

resource.**RLIMIT_SBSIZE**
The maximum size (in bytes) of socket buffer usage for this user. This limits the amount of network memory, and hence the amount of mbufs, that this user may hold at any time.

Availability: FreeBSD 9 or later.

3.4 新版功能.

resource.**RLIMIT_SWAP**
The maximum size (in bytes) of the swap space that may be reserved or used by all of this user id’ s processes. This limit is enforced only if bit 1 of the vm.overcommit sysctl is set. Please see *tuning(7)* for a complete description of this sysctl.

Availability: FreeBSD 9 or later.

3.4 新版功能.

resource.**RLIMIT_NPTS**
The maximum number of pseudo-terminals created by this user id.

Availability: FreeBSD 9 or later.

3.4 新版功能.

35.11.2 Resource Usage

These functions are used to retrieve resource usage information:

resource.**getrusage**(*who*)

This function returns an object that describes the resources consumed by either the current process or its children, as specified by the *who* parameter. The *who* parameter should be specified using one of the RUSAGE_* constants described below.

The fields of the return value each describe how a particular system resource has been used, e.g. amount of time spent running is user mode or number of times the process was swapped out of main memory. Some values are dependent on the clock tick internal, e.g. the amount of memory the process is using.

For backward compatibility, the return value is also accessible as a tuple of 16 elements.

The fields ru_utime and ru_stime of the return value are floating point values representing the amount of time spent executing in user mode and the amount of time spent executing in system mode, respectively. The remaining values are integers. Consult the *getrusage(2)* man page for detailed information about these values. A brief summary is presented here:

索引	域	Resource
0	ru_utime	time in user mode (float)
1	ru_stime	time in system mode (float)
2	ru_maxrss	maximum resident set size
3	ru_ixrss	shared memory size
4	ru_idrss	unshared memory size
5	ru_isrss	unshared stack size
6	ru_minflt	page faults not requiring I/O
7	ru_majflt	page faults requiring I/O
8	ru_nswap	number of swap outs
9	ru_inblock	block input operations
10	ru_oublock	block output operations
11	ru_msgsnd	messages sent
12	ru_msgrcv	messages received
13	ru_nsignals	signals received
14	ru_nvcsw	voluntary context switches
15	ru_nivcsw	involuntary context switches

This function will raise a *ValueError* if an invalid *who* parameter is specified. It may also raise *error* exception in unusual circumstances.

resource.**getpagesize**()

Returns the number of bytes in a system page. (This need not be the same as the hardware page size.)

The following RUSAGE_* symbols are passed to the *getrusage()* function to specify which processes information should be provided for.

resource.**RUSAGE_SELF**

Pass to *getrusage()* to request resources consumed by the calling process, which is the sum of resources used by all threads in the process.

resource.**RUSAGE_CHILDREN**

Pass to *getrusage()* to request resources consumed by child processes of the calling process which have been terminated and waited for.

resource.**RUSAGE_BOTH**

Pass to *getrusage()* to request resources consumed by both the current process and child processes. May not

be available on all systems.

resource.**RUSAGE_THREAD**
Pass to *getrusage()* to request resources consumed by the current thread. May not be available on all systems.

3.2 新版功能.

35.12 nis —Sun 的 NIS (黄页) 接口

nis 模块提供了对 NIS 库的轻量级包装，适用于多个主机的集中管理。

因为 NIS 仅存在于 Unix 系统，此模块仅在 Unix 上可用。

nis 模块定义了以下函数：

nis.**match**(*key*, *mapname*, *domain=default_domain*)
返回 *key* 在映射 *mapname* 中的匹配结果，如无结果则会引发错误 (*nis.error*)。两个参数都应为字符串，*key* 定长 8 个比特。返回值为任意字节数组（可包含 NULL 和其他特殊值）。

请注意如果 *mapname* 是另一名称的别名则会先检查别名。

domain 参数可允许重载用于查找的 NIS 域。如果未指定，则会在默认 NIS 域中查找。

nis.**cat**(*mapname*, *domain=default_domain*)
返回一个字典，其元素为 *key* 到 *value* 的映射使得 `match(key, mapname)==value`。请注意字典的键和值均为任意字节数组。

请注意如果 *mapname* 是另一名称的别名则会先检查别名。

domain 参数可允许重载用于查找的 NIS 域。如果未指定，则会在默认 NIS 域中查找。

nis.**maps**(*domain=default_domain*)
返回全部可用映射的列表。

domain 参数可允许重载用于查找的 NIS 域。如果未指定，则会在默认 NIS 域中查找。

nis.**get_default_domain**()
返回系统默认的 NIS 域。

nis 模块定义了以下异常：

exception nis.**error**
当 NIS 函数返回一个错误码时引发的异常。

35.13 Unix syslog 库例程

此模块提供一个接口到 Unix `syslog` 日常库. 参考 Unix 手册页关于 `syslog` 设施的详细描述.

此模块包装了系统的 `syslog` 例程族。一个能与 syslog 服务器对话的纯 Python 库则以*logging.handlers* 模块中 `SysLogHandler` 类的形式提供。

这个模块定义了以下函数：

syslog.**syslog**(*message*)

syslog.**syslog**(*priority*, *message*)
将字符串 *message* 发送到系统日志记录器。如有必要会添加末尾换行符。每条消息都带有一个由 *facility* 和 *level* 组成的优先级标价签。可选的 *priority* 参数默认值为 LOG_INFO，它确定消息的优先级。如果未在 *priority* 中使用逻辑或 (LOG_INFO | LOG_USER) 对 facility 进行编码，则会使用在*openlog()* 调用中所给定的值。

如果*openlog()* 未在对*syslog()* 的调用之前被调用，则将不带参数地调用 openlog()。

syslog.**openlog**([*ident*[, *logoption*[, *facility*]]])
后续*syslog()* 调用的日志选项可以通过调用*openlog()* 来设置。如果日志当前未打开则*syslog()* 将不带参数地调用*openlog()*。

可选的 *ident* 关键字参数是在每条消息前添加的字符串，默认为 sys.argv[0] 去除打头的路径部分。可选的 *logoption* 关键字参数（默认为 0）是一个位字段–请参见下文了解可能的组合值。可选的 *facility* 关键字参数 (默认为 LOG_USER) 为没有显式编码 facility 的消息设置默认的 facility。

在 3.2 版更改: In previous versions, keyword arguments were not allowed, and *ident* was required. The default for *ident* was dependent on the system libraries, and often was python instead of the name of the python program file.

syslog.**closelog**()
重置日志模块值并且调用系统库 closelog().

这使得此模块在初始导入时行为固定。例如，*openlog()* 将在首次调用*syslog()* 时被调用（如果*openlog()* 还未被调用过），并且 *ident* 和其他*openlog()* 形参会被重置为默认值。

syslog.**setlogmask**(*maskpri*)
将优先级掩码设为 *maskpri* 并返回之前的掩码值。调用*syslog()* 并附带未在 *maskpri* 中设置的优先级将会被忽略。默认设置为记录所有优先级。函数 LOG_MASK(pri) 可计算单个优先级 *pri* 的掩码。函数 LOG_UPTO(pri) 可计算包括 *pri* 在内的所有优先级的掩码。

此莫款定义了一下常量:

优先级级别 (高到低): LOG_EMERG, LOG_ALERT, LOG_CRIT, LOG_ERR, LOG_WARNING, LOG_NOTICE, LOG_INFO, LOG_DEBUG.

设施: LOG_KERN, LOG_USER, LOG_MAIL, LOG_DAEMON, LOG_AUTH, LOG_LPR, LOG_NEWS, LOG_UUCP, LOG_CRON, LOG_SYSLOG, LOG_LOCAL0 to LOG_LOCAL7，如果 <syslog.h> 中有定义则还有 LOG_AUTHPRIV。

日志选项: LOG_PID, LOG_CONS, LOG_NDELAY，如果 <syslog.h> 中有定义则还有 LOG_ODELAY, LOG_NOWAIT 以及 LOG_PERROR。

35.13.1 例子

简单示例

一个简单的示例集:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

一个设置多种日志选项的示例，其中有在日志消息中包含进程 ID，以及将消息写入用于邮件日志记录的目标设施等:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

CHAPTER 36

被取代的模块

本章中描述的模块均已弃用，仅保留用于向后兼容。它们已经被其他模块所取代。

36.1 optparse —解析器的命令行选项

源代码： Lib/optparse.py

3.2 版后已移除: *optparse* 模块已被弃用并且将不再继续开发；开发将转至*argparse* 模块进行。

optparse 是一个相比原有*getopt* 模块更为方便、灵活和强大的命令行选项解析库。*optparse* 使用更为显明的命令行解析风格：创建一个*OptionParser* 的实例，向其中填充选项，然后解析命令行。*optparse* 允许用户以传统的 GNU/POSIX 语法来指定选项，并为你生成额外的用法和帮助消息。

下面是在一个简单脚本中使用*optparse* 的示例:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the "usual thing" on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, *optparse* sets attributes of the `options` object returned by `parse_args()` based on user-supplied command-line values. When `parse_args()` returns from parsing this command line, `options.filename` will be `"outfile"` and `options.verbose` will be `False`. *optparse* supports both long and short

options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of

```
<yourscript> -h
<yourscript> --help
```

and *optparse* will print out a brief summary of your script's options:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).

36.1.1 背景

optparse was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

术语

argument –参数 a string entered on the command-line, and passed by the shell to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term "word".

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read "argument" as "an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`".

选项 an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen ("-") followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by *optparse*.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)
- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`
- a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by *optparse*, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting VMS, MS-DOS, and/or Windows.

可选参数: an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With *optparse*, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab`? Because of this ambiguity, *optparse* does not support this feature.

positional argument –位置参数 something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

必选选项 an option that must be supplied on the command-line; note that the phrase "required option" is self-contradictory in English. *optparse* doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are `find`, `tar`, and `dd`—all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's *not optional*! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

位置位置

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter *how* you get that information from the user—most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply—use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the "Preferences" dialog of a GUI, or command-line options—the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.

36.1.2 教程

While *optparse* is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any *optparse*-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell *optparse* what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to *OptionParser.add_option()* are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line; in reality, *optparse* encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct *optparse* to parse your program's command line:

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to `parse_args()`, but that's rarely necessary: by default it uses `sys.argv[1:]`.)

`parse_args()` 返回两个值:

- `options`, an object containing values for all of your options—e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option

- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: *action*, *type*, *dest* (destination), and *help*. Of these, *action* is the most fundamental.

Understanding option actions

Actions tell *optparse* what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into *optparse*; adding new actions is an advanced topic covered in section *Extending optparse*. Most actions tell *optparse* to store a value in some variable—for example, take a string from the command line and store it in an attribute of `options`.

If you don't specify an option action, *optparse* defaults to `store`.

The store action

The most common option action is `store`, which tells *optparse* to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

例如

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask *optparse* to parse it:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When *optparse* sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in `options.filename`. So, after this call to `parse_args()`, `options.filename` is `"foo.txt"`.

Some other option types supported by *optparse* are `int` and `float`. Here's an option that expects an integer argument:

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is `store`.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option: since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

will print `42`.

If you don't specify a type, *optparse* assumes `string`. Combined with the fact that the default action is `store`, that means our first example can be a lot shorter:

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, *optparse* figures out a sensible default from the option strings: if the first long option string is `--foo-bar`, then the default destination is `foo_bar`. If there are no long option strings, *optparse* looks at the first short option string: the default destination for `-f` is `f`.

optparse also includes the built-in `complex` type. Adding types is covered in section *Extending optparse*.

Handling boolean (flag) options

Flag options—set a variable to true or false when a particular option is seen—are quite common. *optparse* supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q`:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values—see below.)

When *optparse* encounters `-v` on the command line, it sets `options.verbose` to `True`; when it encounters `-q`, `options.verbose` is set to `False`.

Other actions

Some other actions supported by *optparse* are:

`"store_const"` store a constant value

`"append"` append this option's argument to a list

`"count"` increment a counter by one

`"callback"` 调用指定函数

These are covered in section *参考指南*, Reference Guide and section *Option Callbacks*.

默认值

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen? Since we didn't supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. *optparse* lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want *optparse* to set `verbose` to `True` unless `-q` is seen, then we can do this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the *destination* rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

考虑一下:

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults()` method of OptionParser, which you can call at any time before calling `parse_args()`:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

optparse's ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a *help* value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

If *optparse* encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(If the help output is triggered by a help option, *optparse* exits after printing the help text.)

There's a lot going on here to help *optparse* generate the best possible help message:

- the script defines its own usage message:

  ```
  usage = "usage: %prog [options] arg1 arg2"
  ```

  optparse expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

  If you don't supply a usage string, *optparse* uses a bland but sensible default: `"Usage: %prog [options]"`, which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping—*optparse* takes care of wrapping lines and making the help output look good.

- options that take a value indicate this fact in their automatically-generated help message, e.g. for the "mode" option:

```
-m MODE, --mode=MODE
```

  Here, "MODE" is called the meta-variable: it stands for the argument that the user is expected to supply to -m/--mode. By default, *optparse* converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want—for example, the --filename option explicitly sets metavar="FILE", resulting in this automatically-generated option description:

```
-f FILE, --filename=FILE
```

  This is important for more than just saving space, though: the manually written help text uses the meta-variable FILE to clue the user in that there's a connection between the semi-formal syntax -f FILE and the informal semantic description "write output to FILE". This is a simple but effective way to make your help text a lot clearer and more useful for end users.

- options that have a default value can include %default in the help string—*optparse* will replace it with *str()* of the option's default value. If an option has no default value (or the default value is None), %default expands to none.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An *OptionParser* can contain several option groups, each of which can contain several options.

An option group is obtained using the class *OptionGroup*:

class optparse.**OptionGroup**(*parser*, *title*, *description=None*)

where

- parser is the *OptionParser* instance the group will be inserted in to
- title is the group title
- description, optional, is a long description of the group

OptionGroup inherits from OptionContainer (like *OptionParser*) and so the add_option() method can be used to add an option to the group.

Once all the options are declared, using the *OptionParser* method add_option_group() the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an *OptionGroup* to a parser is easy:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

This would result in the following help output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
```

(下页继续)

(续上页)

```
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

A bit more complete example might involve using more than one group: still extending the previous example:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

that results in the following output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.

  Debug Options:
    -d, --debug         Print debug information
    -s, --sql           Print all SQL statements executed
    -e                  Print every action done
```

Another interesting method, in particular when working programmatically with option groups is:

OptionParser.**get_option_group**(*opt_str*)

Return the *OptionGroup* to which the short or long option string *opt_str* (e.g. `'-o'` or `'--option'`) belongs. If there's no such *OptionGroup*, return `None`.

Printing a version string

Similar to the brief usage string, *optparse* can also print a version string for your program. You have to supply the string as the `version` argument to OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` is expanded just like it is in `usage`. Apart from that, `version` can contain anything you like. When you supply it, *optparse* automatically adds a `--version` option to your parser. If it encounters this option on the command line, it expands your `version` string (by replacing `%prog`), prints it to stdout, and exits.

For example, if your script is called `/usr/bin/foo`:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the `version` string:

`OptionParser.`**`print_version`**`(`*file=None*`)`
: Print the version message for the current program (`self.version`) to *file* (default stdout). As with *print_usage()*, any occurrence of `%prog` in `self.version` is replaced with the name of the current program. Does nothing if `self.version` is empty or undefined.

`OptionParser.`**`get_version`**`()`
: Same as *print_version()* but returns the version string instead of printing it.

How `optparse` handles errors

There are two broad classes of errors that *optparse* has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to *OptionParser.add_option()*, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either `optparse.OptionError` or *TypeError*) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. *optparse* can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error()` to signal an application-defined error condition:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, *optparse* handles the error the same way: it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling `OptionParser.error()` from your application code.

If *optparse*'s default error-handling behaviour does not suit your needs, you'll need to subclass OptionParser and override its `exit()` and/or `error()` methods.

Putting it all together

Here's what *optparse*-based scripts usually look like:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

36.1.3 参考指南

创建解析器

The first step in using *optparse* is to create an OptionParser instance.

class optparse.**OptionParser**(...)

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

usage (默认: **"%prog [options]"**) The usage summary to print when your program is run incorrectly or with a help option. When *optparse* prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

option_list (默认: **[]**) A list of Option objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by OptionParser

subclasses), but before any version or help options. Deprecated; use *add_option()* after creating the parser instead.

option_class (默认: optparse.Option) Class to use when adding options to the parser in *add_option()*.

version (默认: None) A version string to print when the user supplies a version option. If you supply a true value for `version`, *optparse* automatically adds a version option with the single option string `--version`. The substring `%prog` is expanded the same as for `usage`.

conflict_handler (默认: "error") Specifies what to do when options with conflicting option strings are added to the parser; see section *Conflicts between options*.

description (默认: None) A paragraph of text giving a brief overview of your program. *optparse* reformats this paragraph to fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

formatter (default: a new IndentedHelpFormatter) An instance of optparse.HelpFormatter that will be used for printing help text. *optparse* provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

add_help_option (默认: True) If true, *optparse* will add a help option (with option strings `-h` and `--help`) to the parser.

prog The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

epilog (默认: None) A paragraph of help text to print after the option help.

填充解析器

There are several ways to populate the parser with options. The preferred way is by using *OptionParser.add_option()*, as shown in section 教程. `add_option()` can be called in one of two ways:

- pass it an Option instance (as returned by `make_option()`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(`make_option()` is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of *optparse* may split Option into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate Option directly.)

定义选项

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an `Option` instance is with the `add_option()` method of *`OptionParser`*.

`OptionParser.`**`add_option`**(*option*)
`OptionParser.`**`add_option`**(**opt_str*, *attr=value*, ...)

To define an option with only a short option string:

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is *`action`*, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, *`optparse`* raises an `OptionError` exception explaining your mistake.

An option's *action* determines what *`optparse`* does when it encounters this option on the command-line. The standard option actions hard-coded into *`optparse`* are:

`"store"` 存储此选项的参数（默认）

`"store_const"` store a constant value

`"store_true"` store a true value

`"store_false"` store a false value

`"append"` append this option's argument to a list

`"append_const"` 将常量值附加到列表

`"count"` increment a counter by one

`"callback"` 调用指定函数

`"help"` 打印用法消息，包括所有选项和文档

(If you don't supply an action, the default is `"store"`. For this action, you may also supply *`type`* and *`dest`* option attributes; see *Standard option actions*.)

As you can see, most actions involve storing or updating a value somewhere. *`optparse`* always creates a special object for this, conventionally called `options` (it happens to be an instance of `optparse.Values`). Option arguments (and various other values) are stored as attributes of this object, according to the *`dest`* (destination) option attribute.

For example, when you call

```
parser.parse_args()
```

one of the first things *`optparse`* does is create the `options` object:

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following:

```
-ffoo
-f foo
--file=foo
--file foo
```

then *optparse*, on seeing this option, will do the equivalent of

```
options.filename = "foo"
```

The *type* and *dest* option attributes are almost as important as *action*, but *action* is the only one that makes sense for *all* options.

Option attributes

The following option attributes may be passed as keyword arguments to *OptionParser.add_option()*. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, *optparse* raises OptionError.

Option.**action**
(默认: "store")

Determines *optparse*'s behaviour when this option is seen on the command line; the available options are documented *here*.

Option.**type**
(默认: "string")

The argument type expected by this option (e.g., "string" or "int"); the available option types are documented *here*.

Option.**dest**
(default: derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells *optparse* where to write it: *dest* names an attribute of the options object that *optparse* builds as it parses the command line.

Option.**default**
The value to use for this option's destination if the option is not seen on the command line. See also *OptionParser.set_defaults()*.

Option.**nargs**
(默认: 1)

How many arguments of type *type* should be consumed when this option is seen. If > 1, *optparse* will store a tuple of values to *dest*.

Option.**const**
For actions that store a constant value, the constant value to store.

Option.**choices**
For options of type "choice", the list of strings the user may choose from.

Option.**callback**
For options with action "callback", the callable to call when this option is seen. See section *Option Callbacks* for detail on the arguments passed to the callable.

Option.**callback_args**
Option.**callback_kwargs**
Additional positional and keyword arguments to pass to callback after the four standard callback arguments.

Option.**help**
Help text to print for this option when listing all available options after the user supplies a *help* option (such as `--help`). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value `optparse.SUPPRESS_HELP`.

Option.**metavar**
(default: derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section 教程 for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide *optparse*' s behaviour; a few have required attributes, which you must specify for any option using that action.

- `"store"` [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is converted to a value according to *type* and stored in *dest*. If *nargs* > 1, multiple arguments will be consumed from the command line; all will be converted according to *type* and stored to *dest* as a tuple. See the *Standard option types* section.

  If *choices* is supplied (a list or tuple of strings), the type defaults to `"choice"`.

  If *type* is not supplied, it defaults to `"string"`.

  If *dest* is not supplied, *optparse* derives a destination from the first long option string (e.g., `--foo-bar` implies `foo_bar`). If there are no long option strings, *optparse* derives a destination from the first short option string (e.g., `-f` implies `f`).

  示例:

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

  As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

  optparse will set

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- `"store_const"` [required: *const*; relevant: *dest*]

  The value *const* is stored in *dest*.

  示例:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

  If `--noisy` is seen, *optparse* will set

```
options.verbose = 2
```

- "store_true" [relevant: *dest*]

  A special case of "store_const" that stores a true value to *dest*.

- "store_false" [relevant: *dest*]

  Like "store_true", but stores a false value.

  示例:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is appended to the list in *dest*. If no default value for *dest* is supplied, an empty list is automatically created when *optparse* first encounters this option on the command-line. If *nargs* > 1, multiple arguments are consumed, and a tuple of length *nargs* is appended to *dest*.

  The defaults for *type* and *dest* are the same as for the "store" action.

  示例:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

  If -t3 is seen on the command-line, *optparse* does the equivalent of:

```
options.tracks = []
options.tracks.append(int("3"))
```

  If, a little later on, --tracks=4 is seen, it does:

```
options.tracks.append(int("4"))
```

  The append action calls the append method on the current value of the option. This means that any default value specified must have an append method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

```
>>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

- "append_const" [required: *const*; relevant: *dest*]

  Like "store_const", but the value *const* is appended to *dest*; as with "append", *dest* defaults to None, and an empty list is automatically created the first time the option is encountered.

- "count" [relevant: *dest*]

  Increment the integer stored at *dest*. If no default value is supplied, *dest* is set to zero before being incremented the first time.

  示例:

```
parser.add_option("-v", action="count", dest="verbosity")
```

  The first time -v is seen on the command line, *optparse* does the equivalent of:

```
options.verbosity = 0
options.verbosity += 1
```

Every subsequent occurrence of `-v` results in

```
options.verbosity += 1
```

- `"callback"` [required: *callback*; relevant: *type*, *nargs*, *callback_args*, *callback_kwargs*]

  Call the function specified by *callback*, which is called as

  ```
  func(option, opt_str, value, parser, *args, **kwargs)
  ```

  See section *Option Callbacks* for more detail.

- `"help"`

  Prints a complete help message for all the options in the current option parser. The help message is constructed from the `usage` string passed to OptionParser's constructor and the *help* string passed to every option.

  If no *help* string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value `optparse.SUPPRESS_HELP`.

  optparse automatically adds a *help* option to all OptionParsers, so you do not normally need to create one.

  示例:

  ```
  from optparse import OptionParser, SUPPRESS_HELP

  # usually, a help option is added automatically, but that can
  # be suppressed using the add_help_option argument
  parser = OptionParser(add_help_option=False)

  parser.add_option("-h", "--help", action="help")
  parser.add_option("-v", action="store_true", dest="verbose",
                    help="Be moderately verbose")
  parser.add_option("--file", dest="filename",
                    help="Input file to read data from")
  parser.add_option("--secret", help=SUPPRESS_HELP)
  ```

  If *optparse* sees either `-h` or `--help` on the command line, it will print something like the following help message to stdout (assuming `sys.argv[0]` is `"foo.py"`):

  ```
  Usage: foo.py [options]

  Options:
    -h, --help        Show this help message and exit
    -v                Be moderately verbose
    --file=FILENAME   Input file to read data from
  ```

  After printing the help message, *optparse* terminates your process with `sys.exit(0)`.

- `"version"`

  Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with *help* options, you will rarely create `version` options, since *optparse* automatically adds them when needed.

Standard option types

optparse has five built-in option types: "string", "int", "choice", "float" and "complex". If you need to add new option types, see section *Extending optparse*.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type "int") are parsed as follows:

- if the number starts with 0x, it is parsed as a hexadecimal number
- if the number starts with 0, it is parsed as an octal number
- if the number starts with 0b, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling *int()* with the appropriate base (2, 8, 10, or 16). If this fails, so will *optparse*, although with a more useful error message.

"float" and "complex" option arguments are converted directly with *float()* and *complex()*, with similar error-handling.

"choice" options are a subtype of "string" options. The *choices* option attribute (a sequence of strings) defines the set of allowed option arguments. optparse.check_choice() compares user-supplied option arguments against this master list and raises OptionValueError if an invalid string is given.

解析参数

The whole point of creating and populating an OptionParser is to call its parse_args() method:

```
(options, args) = parser.parse_args(args=None, values=None)
```

输入参数的位置

args the list of arguments to process (default: sys.argv[1:])

values an optparse.Values object to store option arguments in (default: a new instance of Values) –if you give an existing object, the option defaults will not be initialized on it

and the return values are

options the same object that was passed in as values, or the optparse.Values instance created by *optparse*

args the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply values, it will be modified with repeated *setattr()* calls (roughly one for every option argument stored to an option destination) and returned by parse_args().

If parse_args() encounters any errors in the argument list, it calls the OptionParser's error() method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. OptionParser provides several methods to help you out:

`OptionParser.`**`disable_interspersed_args`**`()`
Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, *optparse* normally accepts this syntax:

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call *disable_interspersed_args()*. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

`OptionParser.`**`enable_interspersed_args`**`()`
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

`OptionParser.`**`get_option`**`(`*opt_str*`)`
Returns the Option instance with the option string *opt_str*, or `None` if no options have that option string.

`OptionParser.`**`has_option`**`(`*opt_str*`)`
Return true if the OptionParser has an option with option string *opt_str* (e.g., `-q` or `--verbose`).

`OptionParser.`**`remove_option`**`(`*opt_str*`)`
If the *OptionParser* has an option corresponding to *opt_str*, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If *opt_str* does not occur in any option belonging to this *OptionParser*, raises *ValueError*.

Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, *optparse* checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

`"error"` **(默认)** assume option conflicts are a programming error and raise `OptionConflictError`

"resolve" resolve option conflicts intelligently (see below)

As an example, let's define an *OptionParser* that resolves conflicts intelligently and add conflicting options to it:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, *optparse* detects that a previously-added option is already using the `-n` option string. Since `conflict_handler` is `"resolve"`, it resolves the situation by removing `-n` from the earlier option's list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It's possible to whittle away the option strings for a previously-added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, *optparse* removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n`/`--dry-run` option is no longer accessible, so *optparse* removes it, leaving this help text:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

清理

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling `destroy()` on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods:

`OptionParser.`**`set_usage`**`(`*usage*`)`
Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

`OptionParser.`**`print_usage`**`(`*file=None*`)`
Print the usage message for the current program (`self.usage`) to *file* (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

`OptionParser.`**`get_usage`**`()`
Same as *print_usage()* but returns the usage string instead of printing it.

OptionParser.**set_defaults**(*dest=value*, ...)

Set default values for several option destinations at once. Using *set_defaults()* is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several "mode" options all set the same destination, any one of them can set the default, and the last one wins:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")    # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")  # overrides above setting
```

To avoid this confusion, use *set_defaults()*:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

36.1.4 Option Callbacks

When *optparse*'s built-in actions and types aren't quite enough for your needs, you have two choices: extend *optparse* or define a callback option. Extending *optparse* is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the `"callback"` action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the *OptionParser.add_option()* method. Apart from *action*, the only option attribute you must specify is `callback`, the function to call:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, *optparse* doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments—the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

optparse always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via *callback_args* and *callback_kwargs*. Thus, the minimal callback function signature is:

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

type has its usual meaning: as with the `"store"` or `"append"` actions, it instructs *optparse* to consume one argument and convert it to *type*. Rather than storing the converted value(s) anywhere, though, *optparse* passes it to your callback function.

nargs also has its usual meaning: if it is supplied and > 1, *optparse* will consume *nargs* arguments, each of which must be convertible to *type*. It then passes a tuple of converted values to your callback.

callback_args a tuple of extra positional arguments to pass to the callback

callback_kwargs a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

option is the Option instance that's calling the callback

opt_str is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string—e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `"--foobar"`.)

value is the argument to this option seen on the command-line. *optparse* will only expect an argument if *type* is set; the type of `value` will be the type implied by the option's type. If *type* for this option is `None` (no argument expected), then `value` will be `None`. If *nargs* > 1, `value` will be a tuple of values of the appropriate type.

parser is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

parser.largs the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of `parse_args()`.)

parser.rargs the current list of remaining arguments, ie. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

parser.values the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of *optparse* for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args is a tuple of arbitrary positional arguments supplied via the *callback_args* option attribute.

kwargs is a dictionary of arbitrary keyword arguments supplied via *callback_kwargs*.

Raising errors in a callback

The callback function should raise `OptionValueError` if there are any problems with the option or its argument(s). *optparse* catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1: trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the `"store_true"` action.

Callback example 2: check option order

Here's a slightly more interesting example: record the fact that `-a` is seen, but blow up if it comes after `-b` in the command-line.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

Callback example 3: check option order (generalized)

If you want to re-use this callback for several similar options (set a flag, but blow up if `-b` has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

Callback example 4: check arbitrary condition

Of course, you could put any condition in there—you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a "store" or "append" option: if you define *type*, then the option takes one argument that must be convertible to that type; if you further define *nargs*, then the option takes *nargs* arguments.

Here's an example that just emulates the standard "store" action:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

Note that *optparse* takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don't need a callback for this example.)

Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as *optparse* doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that *optparse* normally handles for you. In particular, callbacks should implement the conventional rules for bare -- and - arguments:

- either -- or - can be option arguments
- bare -- (if not the argument to some option): halt command-line processing and discard the --
- bare - (if not the argument to some option): halt command-line processing but keep the - (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why *optparse* doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)
```

(下页继续)

(续上页)

```
    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

36.1.5 Extending `optparse`

Since the two major controlling factors in how *optparse* interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of *optparse*'s `Option` class. This class has a couple of attributes that define *optparse*'s types: *TYPES* and *TYPE_CHECKER*.

`Option.`**`TYPES`**
: A tuple of type names; in your subclass, simply define a new tuple *TYPES* that builds on the standard one.

`Option.`**`TYPE_CHECKER`**
: A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```
def check_mytype(option, opt, value)
```

  where `option` is an `Option` instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type `mytype`. The value returned by a type-checking function will wind up in the OptionValues instance returned by `OptionParser.parse_args()`, or be passed to a callback as the `value` parameter.

  Your type-checking function should raise `OptionValueError` if it encounters any problems. `OptionValueError` takes a single string argument, which is passed as-is to *OptionParser*'s `error()` method, which in turn prepends the program name and the string `"error:"` and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a `"complex"` option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because *optparse* 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the *TYPE_CHECKER* class attribute of your Option subclass):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a *copy()* of *Option.TYPE_CHECKER*, we would end up modifying the *TYPE_CHECKER* attribute of *optparse*'s Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it! Now you can write a script that uses the new option type just like any other *optparse*-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don't use `add_option()` in the above way, you don't need to tell OptionParser which option class to use:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that *optparse* has a couple of classifications for actions:

"store" actions actions that result in *optparse* storing a value to an attribute of the current OptionValues instance; these options require a *dest* attribute to be supplied to the Option constructor.

"typed" actions actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a *type* attribute to the Option constructor.

These are overlapping sets: some default "store"actions are `"store"`, `"store_const"`, `"append"`, and `"count"`, while the default "typed" actions are `"store"`, `"append"`, and `"callback"`.

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

`Option.`**`ACTIONS`**
All actions must be listed in ACTIONS.

`Option.`**`STORE_ACTIONS`**
"store" actions are additionally listed here.

`Option.`**`TYPED_ACTIONS`**
"typed" actions are additionally listed here.

`Option.`**`ALWAYS_TYPED_ACTIONS`**
Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that *optparse* assigns the default type, `"string"`, to options with no explicit type whose action is listed in *ALWAYS_TYPED_ACTIONS*.

In order to actually implement your new action, you must override Option's `take_action()` method and add a case that recognizes your action.

For example, let's add an `"extend"` action. This is similar to the standard `"append"` action, but instead of taking a single value from the command-line and appending it to an existing list, `"extend"` will take multiple values in a single

comma-delimited string, and extend an existing list with them. That is, if `--names` is an `"extend"` option of type `"string"`, the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Features of note:

- `"extend"` both expects a value on the command-line and stores that value somewhere, so it goes in both *STORE_ACTIONS* and *TYPED_ACTIONS*.
- to ensure that *optparse* assigns the default type of `"string"` to `"extend"` actions, we put the `"extend"` action in *ALWAYS_TYPED_ACTIONS* as well.
- `MyOption.take_action()` implements just this one new action, and passes control back to `Option.take_action()` for the standard *optparse* actions.
- `values` is an instance of the optparse_parser.Values class, which provides the very useful `ensure_value()` method. `ensure_value()` is essentially *getattr()* with a safety valve; it is called as

  ```
  values.ensure_value(attr, value)
  ```

  If the `attr` attribute of `values` doesn't exist or is `None`, then ensure_value() first sets it to `value`, and then returns 'value. This is very handy for actions like `"extend"`, `"append"`, and `"count"`, all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using `ensure_value()` means that scripts using your action don't have to worry about setting a default value for the option destinations in question; they can just leave the default as `None` and `ensure_value()` will take care of getting it right when it's needed.

36.2 imp —Access the import internals

Source code: Lib/imp.py

3.4 版后已移除: The *imp* package is pending deprecation in favor of *importlib*.

This module provides an interface to the mechanisms used to implement the `import` statement. It defines the following constants and functions:

`imp.`**`get_magic`**`()`
Return the magic string value used to recognize byte-compiled code files (`.pyc` files). (This value may be different for each Python version.)

3.4 版后已移除: Use *importlib.util.MAGIC_NUMBER* instead.

`imp.`**`get_suffixes`**`()`
Return a list of 3-element tuples, each describing a particular type of module. Each triple has the form `(suffix, mode, type)`, where *suffix* is a string to be appended to the module name to form the filename to search for, *mode* is the mode string to pass to the built-in *open()* function to open the file (this can be `'r'` for text files or `'rb'` for binary files), and *type* is the file type, which has one of the values *PY_SOURCE*, *PY_COMPILED*, or *C_EXTENSION*, described below.

3.3 版后已移除: Use the constants defined on *importlib.machinery* instead.

`imp.`**`find_module`**`(`*name*[, *path*]`)`
Try to find the module *name*. If *path* is omitted or `None`, the list of directory names given by `sys.path` is searched, but first a few special places are searched: the function tries to find a built-in module with the given name (*C_BUILTIN*), then a frozen module (*PY_FROZEN*), and on some systems some other places are looked in as well (on Windows, it looks in the registry which may point to a specific file).

Otherwise, *path* must be a list of directory names; each directory is searched for files with any of the suffixes returned by *get_suffixes()* above. Invalid names in the list are silently ignored (but all list items must be strings).

If search is successful, the return value is a 3-element tuple `(file, pathname, description)`:

file is an open *file object* positioned at the beginning, *pathname* is the pathname of the file found, and *description* is a 3-element tuple as contained in the list returned by *get_suffixes()* describing the kind of module found.

If the module does not live in a file, the returned *file* is `None`, *pathname* is the empty string, and the *description* tuple contains empty strings for its suffix and mode; the module type is indicated as given in parentheses above. If the search is unsuccessful, *ImportError* is raised. Other exceptions indicate problems with the arguments or environment.

If the module is a package, *file* is `None`, *pathname* is the package path and the last item in the *description* tuple is *PKG_DIRECTORY*.

This function does not handle hierarchical module names (names containing dots). In order to find *P.M*, that is, submodule *M* of package *P*, use *find_module()* and *load_module()* to find and load package *P*, and then use *find_module()* with the *path* argument set to `P.__path__`. When *P* itself has a dotted name, apply this recipe recursively.

3.3 版后已移除: Use *importlib.util.find_spec()* instead unless Python 3.3 compatibility is required, in which case use *importlib.find_loader()*. For example usage of the former case, see the 例子 section of the *importlib* documentation.

`imp.`**`load_module`**`(`*name*, *file*, *pathname*, *description*`)`
Load a module that was previously found by *find_module()* (or by an otherwise conducted search yielding compatible results). This function does more than importing the module: if the module was already imported, it

will reload the module! The *name* argument indicates the full module name (including the package name, if this is a submodule of a package). The *file* argument is an open file, and *pathname* is the corresponding file name; these can be `None` and `''`, respectively, when the module is a package or not being loaded from a file. The *description* argument is a tuple, as would be returned by *get_suffixes()*, describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually *ImportError*) is raised.

Important: the caller is responsible for closing the *file* argument, if it was not `None`, even when an exception is raised. This is best done using a `try ··· finally` statement.

3.3 版后已移除: If previously used in conjunction with *imp.find_module()* then consider using *importlib.import_module()*, otherwise use the loader returned by the replacement you chose for *imp.find_module()*. If you called *imp.load_module()* and related functions directly with file path arguments then use a combination of *importlib.util.spec_from_file_location()* and *importlib.util.module_from_spec()*. See the 例子 section of the *importlib* documentation for details of the various approaches.

imp.**new_module**(*name*)

Return a new empty module object called *name*. This object is *not* inserted in `sys.modules`.

3.4 版后已移除: Use *importlib.util.module_from_spec()* instead.

imp.**reload**(*module*)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the *module* argument).

When `reload(module)` is executed:

- Python modules' code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module's dictionary. The `init` function of extension modules is not called a second time.
- 与 Python 中的所有的其它对象一样，旧的对象只有在它们的引用计数为 0 之后才会被回收。
- 模块命名空间中的名称重新指向任何新的或更改后的对象。
- 其他旧对象的引用（例如那个模块的外部名称）不会被重新绑定到引用的新对象的，并且如果有需要，必须在出现的每个命名空间中进行更新。

有一些其他注意事项：

当一个模块被重新加载的时候，它的字典（包含了那个模块的全区变量）会被保留。名称的重新定义会覆盖旧的定义，所以通常来说这不是问题。如果一个新模块没有定义在旧版本模块中定义的名称，则将保留旧版本中的定义。这一特性可用于作为那个模块的优点，如果它维护一个全局表或者对象的缓存——使用 `try` 语句，就可以测试表的存在并且跳过它的初始化，如果有需要的话：

```
try:
    cache
except NameError:
    cache = {}
```

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for *sys*, *__main__* and *builtins*. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from ··· import ···`, calling *reload()* for the other module does not redefine the objects imported from it —one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module*.*name*) instead.

如果一个模块创建一个类的实例，重新加载定义那个类的模块不影响那些实例的方法定义———它们继续使用旧类中的定义。对于子类来说同样是正确的。

在 3.3 版更改: Relies on both __name__ and __loader__ being defined on the module being reloaded instead of just __name__.

3.4 版后已移除: Use *importlib.reload()* instead.

The following functions are conveniences for handling **PEP 3147** byte-compiled file paths.

3.2 新版功能.

imp.**cache_from_source**(*path*, *debug_override=None*)

Return the **PEP 3147** path to the byte-compiled file associated with the source *path*. For example, if *path* is /foo/bar/baz.py the return value would be /foo/bar/__pycache__/baz.cpython-32.pyc for Python 3.2. The cpython-32 string comes from the current magic tag (see *get_tag()*; if sys.implementation.cache_tag is not defined then *NotImplementedError* will be raised). By passing in True or False for *debug_override* you can override the system's value for __debug__, leading to optimized bytecode.

path need not exist.

在 3.3 版更改: If sys.implementation.cache_tag is None, then *NotImplementedError* is raised.

3.4 版后已移除: Use *importlib.util.cache_from_source()* instead.

在 3.5 版更改: The *debug_override* parameter no longer creates a .pyo file.

imp.**source_from_cache**(*path*)

Given the *path* to a **PEP 3147** file name, return the associated source code file path. For example, if *path* is /foo/bar/__pycache__/baz.cpython-32.pyc the returned path would be /foo/bar/baz.py. *path* need not exist, however if it does not conform to **PEP 3147** format, a ValueError is raised. If sys.implementation.cache_tag is not defined, *NotImplementedError* is raised.

在 3.3 版更改: Raise *NotImplementedError* when sys.implementation.cache_tag is not defined.

3.4 版后已移除: Use *importlib.util.source_from_cache()* instead.

imp.**get_tag**()

Return the **PEP 3147** magic tag string matching this version of Python's magic number, as returned by *get_magic()*.

3.4 版后已移除: Use sys.implementation.cache_tag directly starting in Python 3.3.

The following functions help interact with the import system's internal locking mechanism. Locking semantics of imports are an implementation detail which may vary from release to release. However, Python ensures that circular imports work without any deadlocks.

imp.**lock_held**()

Return True if the global import lock is currently held, else False. On platforms without threads, always return False.

On platforms with threads, a thread executing an import first holds a global import lock, then sets up a per-module lock for the rest of the import. This blocks other threads from importing the same module until the original import completes, preventing other threads from seeing incomplete module objects constructed by the original thread. An exception is made for circular imports, which by construction have to expose an incomplete module object at some point.

在 3.3 版更改: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

3.4 版后已移除.

imp.**acquire_lock**()
Acquire the interpreter's global import lock for the current thread. This lock should be used by import hooks to ensure thread-safety when importing modules.

Once a thread has acquired the import lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

On platforms without threads, this function does nothing.

在 3.3 版更改: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

3.4 版后已移除.

imp.**release_lock**()
Release the interpreter's global import lock. On platforms without threads, this function does nothing.

在 3.3 版更改: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

3.4 版后已移除.

The following constants with integer values, defined in this module, are used to indicate the search result of *find_module()*.

imp.**PY_SOURCE**
The module was found as a source file.

3.3 版后已移除.

imp.**PY_COMPILED**
The module was found as a compiled code object file.

3.3 版后已移除.

imp.**C_EXTENSION**
The module was found as dynamically loadable shared library.

3.3 版后已移除.

imp.**PKG_DIRECTORY**
The module was found as a package directory.

3.3 版后已移除.

imp.**C_BUILTIN**
The module was found as a built-in module.

3.3 版后已移除.

imp.**PY_FROZEN**
The module was found as a frozen module.

3.3 版后已移除.

class imp.**NullImporter**(*path_string*)
The *NullImporter* type is a **PEP 302** import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises *ImportError*. Otherwise, a *NullImporter* instance is returned.

Instances have only one method:

find_module(*fullname*[, *path*])
This method always returns None, indicating that the requested module could not be found.

在 3.3 版更改: None is inserted into sys.path_importer_cache instead of an instance of *NullImporter*.

3.4 版后已移除: Insert None into sys.path_importer_cache instead.

36.2.1 例子

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This *implementation* wouldn't work in that version, since *find_module()* has been extended and *load_module()* has been added in 1.4.)

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

CHAPTER 37

未创建文档的模块

以下是目前未创建文档模块的速览列表，但其它们应创建文档。欢迎为他们提供文档！（通过电子邮件发送到 docs@python.org ）

本章的想法和原内容取自 Fredrik Lundh 的帖子；本章的具体内容已经大幅修改。

37.1 平台特定模块

这些模块用于实现*os.path* 模块，除此之外没有文档。几乎没有必要创建这些文档。

ntpath —在 Win32 和 Win64 平台上实现*os.path* 。

posixpath —在 POSIX 上实现*os.path* 。

APPENDIX A

术语对照表

>>> 交互式终端中默认的 Python 提示符。往往会显示于能以交互方式在解释器里执行的样例代码之前。

... The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3 一个将 Python 2.x 代码转换为 Python 3.x 代码的工具，能够处理大部分通过解析源码并遍历解析树可检测到的不兼容问题。

2to3 包含在标准库中，模块名为*lib2to3*；并提供一个独立入口点 `Tools/scripts/2to3`。参见*2to3 - 自动将 Python 2 代码转为 Python 3 代码*。

abstract base class –抽象基类 抽象基类简称 ABC，是对*duck-typing* 的补充，它提供了一种定义接口的新方式，相比之下其他技巧例如*hasattr()* 显得过于笨拙或有微妙错误（例如使用 魔术方法）。ABC 引入了虚拟子类，这种类并非继承自其他类，但却仍能被*isinstance()* 和*issubclass()* 所认可；详见*abc* 模块文档。Python 自带许多内置的 ABC 用于实现数据结构（在*collections.abc* 模块中）、数字（在*numbers* 模块中）、流（在*io* 模块中）、导入查找器和加载器（在*importlib.abc* 模块中）。你可以使用*abc* 模块来创建自己的 ABC。

annotation –标注 关联到某个变量、类属性、函数形参或返回值的标签，被约定作为*type hint* 来使用。

局部变量的标注在运行时不可访问，但全局变量、类属性和函数的标注会分别存放模块、类和函数的 `__annotations__` 特殊属性中。

参见*variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，对此功能均有介绍。

argument –参数 在调用函数时传给*function* （或*method* ）的值。参数分为两种：

- *关键字参数*: 在函数调用中前面带有标识符（例如 `name=`）或者作为包含在前面带有 `**` 的字典里的值传入。举例来说，`3` 和 `5` 在以下对*complex()* 的调用中均属于关键字参数:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *位置参数*: 不属于关键字参数的参数。位置参数可出现于参数列表的开头以及/或者作为前面带有 `*` 的*iterable* 里的元素被传入。举例来说，`3` 和 `5` 在以下调用中均属于位置参数:

```
complex(3, 5)
complex(*(3, 5))
```

参数会被赋值给函数体中对应的局部变量。有关赋值规则参见 calls 一节。根据语法，任何表达式都可用来表示一个参数；最终算出的值会被赋给对应的局部变量。

另参见*parameter* 术语表条目，常见问题中 参数与形参的区别以及 **PEP 362**。

asynchronous context manager –异步上下文管理器 此种对象通过定义 `__aenter__()` 和 `__aexit__()` 方法来对 `async with` 语句中的环境进行控制。由 **PEP 492** 引入。

asynchronous generator –异步生成器 返回值为*asynchronous generator iterator* 的函数。它与使用 `async def` 定义的协程函数很相似，不同之处在于它包含 `yield` 表达式以产生一系列可在 `async for` 循环中使用的值。

此术语通常是指异步生成器函数，但在某些情况下则可能是指 异步生成器迭代器。如果需要清楚表达具体含义，请使用全称以避免歧义。

一个异步生成器函数可能包含 `await` 表达式或者 `async for` 以及 `async with` 语句。

asynchronous generator iterator –异步生成器迭代器 *asynchronous generator* 函数所创建的对象。

此对象属于*asynchronous iterator*，当使用 `__anext__()` 方法调用时会返回一个可等待对象来执行异步生成器函数的代码直到下一个 `yield` 表达式。

每个 `yield` 会临时暂停处理，记住当前位置执行状态 (包括局部变量和挂起的 try 语句)。当该 异步生成器迭代器与其他 `__anext__()` 返回的可等待对象有效恢复时，它会从离开位置继续执行。参见 **PEP 492** 和 **PEP 525**。

asynchronous iterable –异步可迭代对象 可在 `async for` 语句中被使用的对象。必须通过它的 `__aiter__()` 方法返回一个*asynchronous iterator*。由 **PEP 492** 引入。

asynchronous iterator –异步迭代器 实现了 `__aiter__()` 和 `__anext__()` 方法的对象。`__anext__` 必须返回一个*awaitable* 对象。`async for` 会处理异步迭代器的 `__anext__()` 方法所返回的可等待对象，直到其引发一个*`StopAsyncIteration`* 异常。由 **PEP 492** 引入。

attribute –属性 关联到一个对象的值，可以使用点号表达式通过其名称来引用。例如，如果一个对象 *o* 具有一个属性 *a*，就可以用 *o.a* 来引用它。

awaitable –可等待对象 能在 `await` 表达式中使用的对象。可以是*coroutine* 或是具有 `__await__()` 方法的对象。参见 **PEP 492**。

BDFL Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python’s creator.

binary file –二进制文件 *file object* 能够读写*字节类对象*。二进制文件的例子包括以二进制模式（`'rb'`, `'wb'` or `'rb+'`）打开的文件、`sys.stdin.buffer`、`sys.stdout.buffer` 以及*`io.BytesIO`* 和*`gzip.GzipFile`* 的实例。

另请参见*text file* 了解能够读写*`str`* 对象的文件对象。

bytes-like object –字节类对象 支持 bufferobjects 并且能导出 C-*contiguous* 缓冲的对象。这包括所有*`bytes`*、*`bytearray`* 和*`array.array`* 对象，以及许多普通*`memoryview`* 对象。字节类对象可在多种二进制数据操作中使用；这些操作包括压缩、保存为二进制文件以及通过套接字发送等。

某些操作需要可变的二进制数据。这种对象在文档中常被称为“可读写字节类对象”。可变缓冲对象的例子包括*`bytearray`* 以及*`bytearray`* 的*`memoryview`*。其他操作要求二进制数据存放于不可变对象（“只读字节类对象”）；这种对象的例子包括*`bytes`* 以及*`bytes`* 对象的*`memoryview`*。

bytecode –字节码 Python 源代码会被编译为字节码，即 CPython 解释器中表示 Python 程序的内部代码。字节码还会缓存在 `.pyc` 文件中，这样第二次执行同一文件时速度更快（可以免去将源码重新编译为字节码）。这种“中间语言”运行在根据字节码执行相应机器码的*virtual machine* 之上。请注意不同 Python 虚拟机上的字节码不一定通用，也不一定能在不同 Python 版本上兼容。

字节码指令列表可以在*dis 模块* 的文档中查看。

class –类 用来创建用户定义对象的模板。类定义通常包含对该类的实例进行操作的方法定义。

class variable –类变量 在类中定义的变量，并且仅限在类的层级上修改 (而不是在类的实例中修改)。

coercion –强制类型转换 The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

complex number –复数 对普通实数系统的扩展，其中所有数字都被表示为一个实部和一个虚部的和。虚数是虚数单位（`-1` 的平方根）的实倍数，通常在数学中写为 `i`，在工程学中写为 `j`。Python 内置了对复数的支持，采用工程学标记方式；虚部带有一个 `j` 后缀，例如 `3+1j`。如果需要*math* 模块内对象的对应复数版本，请使用*cmath*，复数的使用是一个比较高级的数学特性。如果你感觉没有必要，忽略它们也几乎不会有任何问题。

context manager –上下文管理器 在 `with` 语句中使用，通过定义 `__enter__()` 和 `__exit__()` 方法来控制环境状态的对象。参见 **PEP 343**。

contiguous –连续 一个缓冲如果是 *C 连续*或 *Fortran 连续*就会被认为是连续的。零维缓冲是 C 和 Fortran 连续的。在一维数组中，所有条目必须在内存中彼此相邻地排列，采用从零开始的递增索引顺序。在多维 C-连续数组中，当按内存地址排列时用最后一个索引访问条目时速度最快。但是在 Fortran 连续数组中则是用第一个索引最快。

coroutine –协程 Coroutines is a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also **PEP 492**.

coroutine function –协程函数 返回一个*coroutine* 对象的函数。协程函数可通过 `async def` 语句来定义，并可能包含 `await`、`async for` 和 `async with` 关键字。这些特性是由 **PEP 492** 引入的。

CPython Python 编程语言的规范实现，在 python.org 上发布。”CPython” 一词用于在必要时将此实现与其他实现例如 Jython 或 IronPython 相区别。

decorator –装饰器 返回值为另一个函数的函数，通常使用 `@wrapper` 语法形式来进行函数变换。装饰器的常见例子包括*classmethod()* 和*staticmethod()*。

装饰器语法只是一种语法糖，以下两个函数定义在语义上完全等价:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

同的样概念也适用于类，但通常较少这样使用。有关装饰器的详情可参见 函数定义和 类定义的文档。

descriptor –描述器 任何定义了 `__get__()`, `__set__()` 或 `__delete__()` 方法的对象。当一个类属性为描述器时，它的特殊绑定行为就会在属性查找时被触发。通常情况下，使用 *a.b* 来获取、设置或删除一个属性时会在 *a* 的类字典中查找名称为 *b* 的对象，但如果 *b* 是一个描述器，则会调用对应的描述器方法。理解描述器的概念是更深层次理解 Python 的关键，因为这是许多重要特性的基础，包括函数、方法、属性、类方法、静态方法以及对超类的引用等等。

有关描述符的方法的详情可参看 descriptors。

dictionary –字典 一个关联数组，其中的任意键都映射到相应的值。键可以是任何具有 `__hash__()` 和 `__eq__()` 方法的对象。在 Perl 语言中称为 hash。

dictionary view –字典视图 从*dict.keys()*, *dict.values()* 和*dict.items()* 返回的对象被称为字典视图。它们提供了字典条目的一个动态视图，这意味着当字典改变时，视图也会相应改变。要将字典视图强制转换为真正的列表，可使用 `list(dictview)`。参见字典视图对象。

docstring –文档字符串 作为类、函数或模块之内的第一个表达式出现的字符串字面值。它在代码执行时会被忽略，但会被解释器识别并放入所在类、函数或模块的 `__doc__` 属性中。由于它可用于代码内省，因此是对象存放文档的规范位置。

duck-typing –鸭子类型 指一种编程风格，它并不依靠查找对象类型来确定其是否具有正确的接口，而是直接调用或使用其方法或属性（“看起来像鸭子，叫起来也像鸭子，那么肯定就是鸭子。”）由于强调接口而非特定类型，设计良好的代码可通过允许多态替代来提升灵活性。鸭子类型避免使用*type()* 或*isinstance()* 检测。(但要注意鸭子类型可以使用抽象基类 作为补充。) 而往往会采用*hasattr()* 检测或是*EAFP* 编程。

EAFP “求原谅比求许可更容易”的英文缩写。这种 Python 常用代码编写风格会假定所需的键或属性存在，并在假定错误时捕获异常。这种简洁快速风格的特点就是大量运用 `try` 和 `except` 语句。于其相对的则是所谓*LBYL* 风格，常见于 C 等许多其他语言。

expression –表达式 A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `if`. Assignments are also statements, not expressions.

extension module –扩展模块 以 C 或 C++ 编写的模块，使用 Python 的 C API 来与语言核心以及用户代码进行交互。

f-string –f-字符串 带有 `'f'` 或 `'F'` 前缀的字符串字面值通常被称为“f-字符串”即 格式化字符串字面值的简写。参见 **PEP 498**。

file object –文件对象 对外提供面向文件 API 以使用下层资源的对象（带有 `read()` 或 `write()` 这样的方法）。根据其创建方式的不同，文件对象可以处理对真实磁盘文件，对其他类型存储，或是对通讯设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等等）。文件对象也被称为 *文件类对象* 或 *流*。

实际上共有三种类别的文件对象: 原始二进制文件, 缓冲二进制文件 以及文本文件。它们的接口定义均在*io* 模块中。创建文件对象的规范方式是使用*open()* 函数。

file-like object –文件类对象 *file object* 的同义词。

finder –查找器 一种会尝试查找被导入模块的*loader* 的对象。

从 Python 3.3 起存在两种类型的查找器: *元路径查找器* 配合*sys.meta_path* 使用，以及*path entry finders* 配合*sys.path_hooks* 使用。

更多详情可参见 **PEP 302**, **PEP 420** 和 **PEP 451**。

floor division –向下取整除法 向下舍入到最接近的整数的数学除法。向下取整除法的运算符是 `//`。例如，表达式 `11 // 4` 的计算结果是 `2`，而与之相反的是浮点数的真正除法返回 `2.75`。注意 `(-11) // 4` 会返回 `-3` 因为这是 `-2.75` 向下舍入得到的结果。见 **PEP 238**。

function –函数 可以向调用者返回某个值的一组语句。还可以向其传入零个或多个参数 并在函数体执行中被使用。另见*parameter*, *method* 和 function 等节。

function annotation –函数标注 即针对函数形参或返回值的*annotation*。

函数标注通常用于类型提示：例如以下函数预期接受两个*int* 参数并预期返回一个*int* 值:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

函数标注语法的详解见 function 一节。

请参看*variable annotation* 和 **PEP 484** 对此功能的描述。

__future__ 一种伪模块，可被程序员用来启用与当前解释器不兼容的新语言特性。

通过导入*__future__* 模块并对其中的变量求值，你可以查看新特性何时首次加入语言以及何时成为默认:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection –垃圾回收 释放不再被使用的内存空间的过程。Python 是通过引用计数和一个能够检测和打破循环引用的循环垃圾回收器来执行垃圾回收的。可以使用*gc* 模块来控制垃圾回收器。

generator –生成器 返回一个*generator iterator* 的函数。它看起来很像普通函数，不同点在于其包含 `yield` 表达式以便产生一系列值供给 for-循环使用或是通过*next()* 函数逐一获取。

通常是指生成器函数，但在某些情况下也可能是指 生成器迭代器。如果需要清楚表达具体含义，请使用全称以避免歧义。

generator iterator –生成器迭代器 *generator* 函数所创建的对象。

每个 `yield` 会临时暂停处理，记住当前位置执行状态（包括局部变量和挂起的 try 语句）。当该 生成器迭代器恢复时，它会从离开位置继续执行（这与每次调用都从新开始的普通函数差别很大）。

generator expression –生成器表达式 An expression that returns an iterator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function –泛型函数 为不同的类型实现相同操作的多个函数所组成的函数。在调用时会由调度算法来确定应该使用哪个实现。

另请参见*single dispatch* 术语表条目、*functools.singledispatch()* 装饰器以及 **PEP 443**。

GIL 参见*global interpreter lock*。

global interpreter lock –全局解释器锁 *CPython* 解释器所采用的一种机制，它确保同一时刻只有一个线程在执行 Python *bytecode*。此机制通过设置对象模型（包括*dict* 等重要内置类型）针对并发访问的隐式安全简化了 CPython 实现。给整个解释器加锁使得解释器多线程运行更方便，其代价则是牺牲了在多处理器上的并行性。

不过，某些标准库或第三方库的扩展模块被设计为在执行计算密集型任务如压缩或哈希时释放 GIL。此外，在执行 I/O 操作时也总是会释放 GIL。

创建一个（以更精细粒度来锁定共享数据的）“自由线程”解释器的努力从未获得成功，因为这会牺牲在普通单处理器情况下的性能。据信克服这种性能问题的措施将导致实现变得更复杂，从而更难以维护。

hashable –可哈希 一个对象的哈希值如果在其生命周期内绝不改变，就被称为 可哈希（它需要具有 `__hash__()` 方法），并可以同其他对象进行比较（它需要具有 `__eq__()` 方法）。可哈希对象必须具有相同的哈希值比较结果才会相同。

可哈希性使得对象能够作为字典键或集合成员使用，因为这些数据结构要在内部使用哈希值。

All of Python’s immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their *id()*.

IDLE Python 的 IDE，“集成开发与学习环境”的英文缩写。是 Python 标准发行版附带的基本编辑器和解释器环境。

immutable –不可变 具有固定值的对象。不可变对象包括数字、字符串和元组。这样的对象不能被改变。如果必须存储一个不同的值，则必须创建新的对象。它们在需要常量哈希值的地方起着重要作用，例如作为字典中的键。

import path –导入路径 由多个位置（或路径条目）组成的列表，会被模块的*path based finder* 用来查找导入目标。在导入时，此位置列表通常来自*sys.path*，但对次级包来说也可能来自上级包的 `__path__` 属性。

importing –导入 令一个模块中的 Python 代码能为另一个模块中的 Python 代码所使用的过程。

importer –导入器 查找并加载模块的对象；此对象既属于*finder* 又属于*loader*。

interactive –交互 Python 带有一个交互式解释器，即你可以在解释器提示符后输入语句和表达式，立即执行并查看其结果。只需不带参数地启动 `python` 命令（也可以在你的计算机开始菜单中选择相应菜单项）。在测试新想法或检验模块和包的时候用这种方式会非常方便（请记得使用 `help(x)`）。

interpreted –解释型 Python 一是种解释型语言，与之相对的是编译型语言，虽然两者的区别由于字节码编译器的存在而会有所模糊。这意味着源文件可以直接运行而不必显式地创建可执行文件再运行。解释型语言通常具有比编译型语言更短的开发/调试周期，但是其程序往往运行得更慢。参见*interactive*。

interpreter shutdown –解释器关闭 当被要求关闭时，Python 解释器将进入一个特殊运行阶段并逐步释放所有已分配资源，例如模块和各种关键内部结构等。它还会多次调用垃圾回收器。这会触发用户定义析构器或弱引用回调中的代码执行。在关闭阶段执行的代码可能会遇到各种异常，因为其所依赖的资源已不再有效（常见的例子有库模块或警告机制等）。

解释器需要关闭的主要原因有 `__main__` 模块或所运行的脚本已完成执行。

iterable –可迭代对象 能够逐一返回其成员项的对象。可迭代对象的例子包括所有序列类型（例如*list*、*str* 和*tuple*）以及某些非序列类型例如*dict*、文件对象 以及定义了 `__iter__()` 方法或是实现了*Sequence* 语义的 `__getitem__()` 方法的任意自定义类对象。

可迭代对象被可用于 `for` 循环以及许多其他需要一个序列的地方（*zip()*、*map()* …）。当一个可迭代对象作为参数传给内置函数*iter()* 时，它会返回该对象的迭代器。这种迭代器适用于对值集合的一次性遍历。在使用可迭代对象时，你通常不需要调用*iter()* 或者自己处理迭代器对象。`for` 语句会为你自动处理那些操作，创建一个临时的未命名变量用来在循环期间保存迭代器。参见*iterator*、*sequence* 以及*generator*。

iterator –迭代器 用来表示一连串数据流的对象。重复调用迭代器的*__next__()* 方法（或将其传给内置函数*next()*）将逐个返回流中的项。当没有数据可用时则将引发*StopIteration* 异常。到这时迭代器对象中的数据项已耗尽，继续调用其 `__next__()` 方法只会再次引发*StopIteration* 异常。迭代器必须具有 `__iter__()` 方法用来返回该迭代器对象自身，因此迭代器必定也是可迭代对象，可被用于其他可迭代对象适用的大部分场合。一个显著的例外是那些会多次重复访问迭代项的代码。容器对象（例如*list*）在你每次向其传入*iter()* 函数或是在 `for` 循环中使用它时都会产生一个新的迭代器。如果在此情况下你尝试用迭代器则会返回在之前迭代过程中被耗尽的同一迭代器对象，使其看起来就像是一个空容器。

更多信息可查看迭代器类型。

key function –键函数 键函数或称整理函数，是能够返回用于排序或排位的值的可调用对象。例如，*locale.strxfrm()* 可用于生成一个符合特定区域排序约定的排序键。

Python 中有许多工具都允许用键函数来控制元素的排位或分组方式。其中包括*min()*，*max()*，*sorted()*，*list.sort()*，*heapq.merge()*，*heapq.nsmallest()*，*heapq.nlargest()* 以及*itertools.groupby()*。

要创建一个键函数有多种方式。例如，*str.lower()* 方法可以用作忽略大小写排序的键函数。另外，键函数也可通过 `lambda` 表达式来创建，例如 `lambda r: (r[0], r[2])`。还有*operator* 模块提供了三个键函数构造器：*attrgetter()*、*itemgetter()* 和*methodcaller()*。请查看 如何排序 一节以获取创建和使用键函数的示例。

keyword argument –关键字参数 参见*argument*。

lambda 由一个单独*expression* 构成的匿名内联函数，表达式会在调用时被求值。创建 lambda 函数的句法为 `lambda [parameters]: expression`

LBYL “先查看后跳跃”的英文缩写。这种代码编写风格会在进行调用或查找之前显式地检查前提条件。此风格与*EAFP* 方式恰成对比，其特点是大量使用 `if` 语句。

在多线程环境中，LBYL 方式会导致“查看”和“跳跃”之间发生条件竞争风险。例如，以下代码 `if key in mapping: return mapping[key]` 可能由于在检查操作之后其他线程从 *mapping* 中移除了 *key* 而出错。这种问题可通过加锁或使用 EAFP 方式来解决。

list –**列表** Python 内置的一种*sequence*。虽然名为列表，但更类似于其他语言中的数组而非链接列表，因为访问元素的时间复杂度为 O(1)。

list comprehension –**列表推导式** 处理一个序列中的所有或部分元素并返回结果列表的一种紧凑写法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 将生成一个 0 到 255 范围内的十六进制偶数对应字符串（0x..）的列表。其中 `if` 子句是可选的，如果省略则 `range(256)` 中的所有元素都会被处理。

loader –**加载器** 负责加载模块的对象。它必须定义名为 `load_module()` 的方法。加载器通常由一个*finder* 返回。详情参见 **PEP 302**，对于*abstract base class* 可参见*importlib.abc.Loader*。

mapping –**映射** 一种支持任意键查找并实现了*Mapping* 或*MutableMapping* *抽象基类* 中所规定方法的容器对象。此类对象的例子包括*dict*, *collections.defaultdict*, *collections.OrderedDict* 以及*collections.Counter*。

meta path finder –**元路径查找器** *sys.meta_path* 的搜索所返回的*finder*。元路径查找器与*path entry finders* 存在关联但并不相同。

请查看*importlib.abc.MetaPathFinder* 了解元路径查找器所实现的方法。

metaclass –**元类** 一种用于创建类的类。类定义包含类名、类字典和基类列表。元类负责接受上述三个参数并创建相应的类。大部分面向对象的编程语言都会提供一个默认实现。Python 的特别之处在于可以创建自定义元类。大部分用户永远不需要这个工具，但当需要出现时，元类可提供强大而优雅的解决方案。它们已被用于记录属性访问日志、添加线程安全性、跟踪对象创建、实现单例，以及其他许多任务。

更多详情参见 metaclasses。

method **方法** 在类内部定义的函数。如果作为该类的实例的一个属性来调用，方法将会获取实例对象作为其第一个*argument* (通常命名为 `self`)。参见*function* 和*nested scope*。

method resolution order –**方法解析顺序** 方法解析顺序就是在查找成员时搜索全部基类所用的先后顺序。请查看 Python 2.3 方法解析顺序 了解自 2.3 版起 Python 解析器所用相关算法的详情。

module **模块** 此对象是 Python 代码的一种组织单位。各模块具有独立的命名空间，可包含任意 Python 对象。模块可通过*importing* 操作被加载到 Python 中。

另见*package*。

module spec –**模块规格** 一个命名空间，其中包含用于加载模块的相关导入信息。是*importlib.machinery.ModuleSpec* 的实例。

MRO 参见*method resolution order*。

mutable –**可变** 可变对象可以在其*id()* 保持固定的情况下改变其取值。另请参见*immutable*。

named tuple –**具名元组** Any tuple-like class whose indexable elements are also accessible using named attributes (for example, *time.localtime()* returns a tuple-like object where the *year* is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as *time.struct_time*, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function *collections.namedtuple()*. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

namespace –命名空间 命名空间是存放变量的场所。命名空间有局部、全局和内置的，还有对象中的嵌套命名空间（在方法之内）。命名空间通过防止命名冲突来支持模块化。例如，函数*builtins.open* 与*os.open()* 可通过各自的命名空间来区分。命名空间还通过明确哪个模块实现那个函数来帮助提高可读性和可维护性。例如，*random.seed()* 或*itertools.islice()* 这种写法明确了这些函数是由*random* 与*itertools* 模块分别实现的。

namespace package –命名空间包 **PEP 420** 所引入的一种仅被用作子包的容器的*package*，命名空间包可以没有实体表示物，其描述方式与*regular package* 不同，因为它们没有 `__init__.py` 文件。

另可参见*module*。

nested scope –嵌套作用域 在一个定义范围内引用变量的能力。例如，在另一函数之内定义的函数可以引用前者的变量。请注意嵌套作用域默认只对引用有效而对赋值无效。局部变量的读写都受限于最内层作用域。类似的，全局变量的读写则作用于全局命名空间。通过 `nonlocal` 关键字可允许写入外层作用域。

new-style class –新式类 对于目前已被应于所有类对象的类形式的旧称谓。在早先的 Python 版本中，只有新式类能够使用 Python 新增的更灵活特性，例如 `__slots__`、描述符、特征属性、`__getattribute__()`、类方法和静态方法等。

object –对象 任何具有状态（属性或值）以及预定义行为（方法）的数据。object 也是任何*new-style class* 的最顶层基类名。

package –包 一种可包含子模块或递归地包含子包的 Python *module*。从技术上说，包是带有 `__path__` 属性的 Python 模块。

另参见*regular package* 和*namespace package*。

parameter –形参 *function*（或方法）定义中的命名实体，它指定函数可以接受的一个*argument*（或在某些情况下，多个实参）。有五种形参：

- *positional-or-keyword*：位置或关键字，指定一个可以作为*位置参数* 传入也可以作为*关键字参数* 传入的实参。这是默认的形参类型，例如下面的 *foo* 和 *bar*：

```
def func(foo, bar=None): ...
```

- *positional-only*：仅限位置，指定一个只能按位置传入的参数。Python 中没有定义仅限位置形参的语法。但是一些内置函数有仅限位置形参（比如*abs()*）。
- *keyword-only*：仅限关键字，指定一个只能通过关键字传入的参数。仅限关键字形参可通过在函数定义的形参列表中包含单个可变位置形参或者在多个可变位置形参之前放一个 `*` 来定义，例如下面的 *kw_only1* 和 *kw_only2*：

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*：可变位置，指定可以提供由一个任意数量的位置参数构成的序列（附加在其他形参已接受的位置参数之后）。这种形参可通过在形参名称前加缀 `*` 来定义，例如下面的 *args*：

```
def func(*args, **kwargs): ...
```

- *var-keyword*：可变关键字，指定可以提供任意数量的关键字参数（附加在其他形参已接受的关键字参数之后）。这种形参可通过在形参名称前加缀 `**` 来定义，例如上面的 *kwargs*。

形参可以同时指定可选和必选参数，也可以为某些可选参数指定默认值。

另参见*argument* 术语表条目、参数与形参的区别中的常见问题、*inspect.Parameter* 类、function 一节以及 **PEP 362**。

path entry –路径入口 *import path* 中的一个单独位置，会被*path based finder* 用来查找要导入的模块。

path entry finder –路径入口查找器 任一可调用对象使用*sys.path_hooks*（即*path entry hook*）返回的*finder*，此种对象能通过*path entry* 来定位模块。

请参看*importlib.abc.PathEntryFinder* 以了解路径入口查找器所实现的各个方法。

path entry hook –路径入口钩子 一种可调用对象，在知道如何查找特定*path entry* 中的模块的情况下能够使用 sys.path_hook 列表返回一个*path entry finder*。

path based finder –基于路径的查找器 默认的一种*元路径查找器*，可在一个*import path* 中查找模块。

path-like object –路径类对象 代表一个文件系统路径的对象。类路径对象可以是一个表示路径的*str* 或者*bytes* 对象，还可以是一个实现了*os.PathLike* 协议的对象。一个支持*os.PathLike* 协议的对象可通过调用*os.fspath()* 函数转换为*str* 或者*bytes* 类型的文件系统路径；*os.fsdecode()* 和*os.fsencode()* 可被分别用来确保获得*str* 或*bytes* 类型的结果。此对象是由 **PEP 519** 引入的。

PEP “Python 增强提议”的英文缩写。一个 PEP 就是一份设计文档，用来向 Python 社区提供信息，或描述一个 Python 的新增特性及其进度或环境。PEP 应当提供精确的技术规格和所提议特性的原理说明。

PEP 应被作为提出主要新特性建议、收集社区对特定问题反馈以及为必须加入 Python 的设计决策编写文档的首选机制。PEP 的作者有责任在社区内部建立共识，并应将不同意见也记入文档。

参见 **PEP 1**。

portion –部分 构成一个命名空间包的单个目录内文件集合（也可能存放于一个 zip 文件内），具体定义见 **PEP 420**。

positional argument –位置参数 参见*argument*。

provisional API –暂定 API 暂定 API 是指被有意排除在标准库的向后兼容性保证之外的应用编程接口。虽然此类接口通常不会再有重大改变，但只要其被标记为暂定，就可能在核心开发者确定有必要的情况下进行向后不兼容的更改（甚至包括移除该接口）。此种更改并不会随意进行–仅在 API 被加入之前未考虑到的严重基础性缺陷被发现时才可能会这样做。

即便是对暂定 API 来说，向后不兼容的更改也会被视为“最后的解决方案”——任何问题被确认时都会尽可能先尝试找到一种向后兼容的解决方案。

这种处理过程允许标准库持续不断地演进，不至于被有问题的长期性设计缺陷所困。详情见 **PEP 411**。

provisional package –暂定包 参见*provisional API*。

Python 3000 Python 3.x 发布路线的昵称（这个名字在版本 3 的发布还遥遥无期的时候就已出现了）。有时也被缩写为“Py3k”。

Pythonic 指一个思路或一段代码紧密遵循了 Python 语言最常用的风格和理念，而不是使用其他语言中通用的概念来实现代码。例如，Python 的常用风格是使用 for 语句循环来遍历一个可迭代对象中的所有元素。许多其他语言没有这样的结构，因此不熟悉 Python 的人有时会选择使用一个数字计数器:

```
for i in range(len(food)):
    print(food[i])
```

而相应的更简洁更 Pythonic 的方法是这样的:

```
for piece in food:
    print(piece)
```

qualified name –限定名称 一个以点号分隔的名称，显示从模块的全局作用域到该模块中定义的某个类、函数或方法的“路径”，相关定义见 **PEP 3155**。对于最高层级的函数和类，限定名称与对象名称一致:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
```

(下页继续)

(续上页)

```
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

当被用于引用模块时，完整限定名称意为标示该模块的以点号分隔的整个路径，其中包含其所有的父包，例如 email.mime.text:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count –引用计数 对特定对象的引用的数量。当一个对象的引用计数降为零时，所分配资源将被释放。引用计数对 Python 代码来说通常是不可见的，但它是*CPython* 实现的一个关键元素。*sys* 模块定义了一个*getrefcount()* 函数，程序员可调用它来返回特定对象的引用计数。

regular package –常规包 传统型的*package*，例如包含有一个 __init__.py 文件的目录。

另参见*namespace package*。

__slots__ 一种写在类内部的声明，通过预先声明实例属性等对象并移除实例字典来节省内存。虽然这种技巧很流行，但想要用好却并不容易，最好是只保留在少数情况下采用，例如极耗内存的应用程序，并且其中包含大量实例。

sequence –序列 一种*iterable*，它支持通过 __getitem__() 特殊方法来使用整数索引进行高效的元素访问，并定义了一个返回序列长度的 __len__() 方法。内置的序列类型有*list*、*str*、*tuple* 和*bytes*。注意虽然*dict* 也支持 __getitem__() 和 __len__()，但它被认为属于映射而非序列，因为它查找时使用任意的*immutable* 键而非整数。

collections.abc.Sequence 抽象基类定义了一个更丰富的接口，它超越了 __getitem__() 和 __len__()，添加了 count(), index(), __contains__() 和 __reversed__() 。可以使用 register() 显式注册实现此扩展接口的类型。

single dispatch –单分派 一种*generic function* 分派形式，其实现是基于单个参数的类型来选择的。

slice –切片 通常只包含了特定*sequence* 的一部分的对象。切片是通过使用下标标记来创建的，在 [] 中给出几个以冒号分隔的数字，例如 variable_name[1:3:5]。方括号（下标）标记在内部使用*slice* 对象。

special method –特殊方法 一种由 Python 隐式调用的方法，用来对某个类型执行特定操作例如相加等等。这种方法的名称的首尾都为双下划线。特殊方法的文档参见 specialnames。

statement –语句 语句是程序段（一个代码“块”）的组成单位。一条语句可以是一个*expression* 或某个带有关键字的结构，例如 if、while 或 for。

struct sequence A tuple with named elements. Struct sequences expose an interface similar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like *_make()* or *_asdict()*. Examples of struct sequences include *sys.float_info* and the return value of *os.stat()*.

text encoding –文本编码 用于将 Unicode 字符串编码为字节串的编码器。

text file –文本文件 一种能够读写*str* 对象的*file object*。通常一个文本文件实际是访问一个面向字节的数据流并自动处理*text encoding*。文本文件的例子包括以文本模式（'r' 或 'w'）打开的文件、*sys.stdin*、*sys.stdout* 以及*io.StringIO* 的实例。

另请参看*binary file* 了解能够读写字节类对象 的文件对象。

triple-quoted string –三引号字符串 首尾各带三个连续双引号（"）或者单引号（'）的字符串。它们在功能上与首尾各用一个引号标注的字符串没有什么不同，但是有多种用处。它们允许你在字符串内包含未经

转义的单引号和双引号，并且可以跨越多行而无需使用连接符，在编写文档字符串时特别好用。

type –**类型** 类型决定一个 Python 对象属于什么种类；每个对象都具有一种类型。要知道对象的类型，可以访问它的*__class__* 属性，或是通过 `type(obj)` 来获取。

type alias –**类型别名** 一个类型的同义词，创建方式是把类型赋值给特定的标识符。

类型别名的作用是简化*类型提示*。例如:

```
from typing import List, Tuple

def remove_gray_shades(
        colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

可以这样提高可读性:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

参见*typing* 和 **PEP 484**，其中有对此功能的详细描述。

type hint –**类型提示** *annotation* 为变量、类属性、函数的形参或返回值指定预期的类型。

类型提示属于可选项，Python 不要求提供，但其可对静态类型分析工具起作用，并可协助 IDE 实现代码补全与重构。

全局变量、类属性和函数的类型提示可以使用*typing.get_type_hints()* 来访问，但局部变量则不可以。

参见*typing* 和 **PEP 484**，其中有对此功能的详细描述。

universal newlines –**通用换行** 一种解读文本流的方式，将以下所有符号都识别为行结束标志：Unix 的行结束约定 `'\n'`、Windows 的约定 `'\r\n'` 以及旧版 Macintosh 的约定 `'\r'`。参见 **PEP 278** 和 **PEP 3116** 和*bytes.splitlines()* 了解更多用法说明。

variable annotation –**变量标注** 对变量或类属性的*annotation*。

在标注变量或类属性时，还可选择为其赋值:

```
class C:
    field: 'annotation'
```

变量标注通常被用作*类型提示*：例如以下变量预期接受*int* 类型的值:

```
count: int = 0
```

变量标注语法的详细解释见 annassign 一节。

请参看*function annotation*、**PEP 484** 和 **PEP 526**，其中对此功能有详细描述。

virtual environment –**虚拟环境** 一种采用协作式隔离的运行时环境，允许 Python 用户和应用程序在安装和升级 Python 分发包时不会干扰到同一系统上运行的其他 Python 应用程序的行为。

另参见*venv*。

virtual machine –**虚拟机** 一台完全通过软件定义的计算机。Python 虚拟机可执行字节码编译器所生成的*bytecode*。

Zen of Python –Python 之禅 列出 Python 设计的原则与哲学，有助于理解与使用这种语言。查看其具体内容可在交互模式提示符中输入“`import this`”。

APPENDIX B

文档说明

这些文档生成自 reStructuredText 原文档，由 Sphinx （一个专门为 Python 文档写的文档生成器）创建。

本文档和它所用工具链的开发完全是由志愿者完成的，这和 Python 本身一样。如果您想参与进来，请阅读 reporting-bugs 了解如何参与。我们随时欢迎新的志愿者！

特别鸣谢：

- Fred L. Drake, Jr.，创造了用于早期 Python 文档的工具链，以及撰写了非常多的文档；
- Docutils 软件包 项目，创建了 reStructuredText 文本格式和 Docutils 软件套件；
- Fredrik Lundh，Sphinx 从他的 Alternative Python Reference 项目中获得了很多好的想法。

B.1 Python 文档的贡献者

有很多对 Python 语言，Python 标准库和 Python 文档有贡献的人，随 Python 源代码发布的 Misc/ACKS 文件列出了部分贡献者。

有了 Python 社区的输入和贡献，Python 才有了如此出色的文档 - 谢谢你们！

APPENDIX C

历史和许可证

C.1 该软件的历史

Python 由荷兰数学和计算机科学研究学会（CWI，见 https://www.cwi.nl/ ）的 Guido van Rossum 于 1990 年代初设计，作为一门叫做 ABC 的语言的替代品。尽管 Python 包含了许多来自其他人的贡献，Guido 仍是其主要作者。

1995 年，Guido 在弗吉尼亚州的国家创新研究公司（CNRI，见 https://www.cnri.reston.va.us/ ）继续他在 Python 上的工作，并在那里发布了该软件的多个版本。

2000 年五月，Guido 和 Python 核心开发团队转到 BeOpen.com 并组建了 BeOpen PythonLabs 团队。同年十月，PythonLabs 团队转到 Digital Creations (现为 Zope Corporation；见 https://www.zope.org/)。2001 年，Python 软件基金会 (PSF，见 https://www.python.org/psf/) 成立，这是一个专为拥有 Python 相关知识产权而创建的非营利组织。Zope Corporation 现在是 PSF 的赞助成员。

所有的 Python 版本都是开源的（有关开源的定义参阅 https://opensource.org/ ）。历史上，绝大多数 Python 版本是 GPL 兼容的；下表总结了各个版本情况。

发布版本	源自	年份	所有者	GPL 兼容？
0.9.0 至 1.2	n/a	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	否
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 及更高	2.1.1	2001 至今	PSF	是

注解： GPL 兼容并不意味着 Python 在 GPL 下发布。与 GPL 不同，所有 Python 许可证都允许您分发修改后

的版本，而无需开源所做的更改。GPL 兼容的许可证使得 Python 可以与其它在 GPL 下发布的软件结合使用；但其它的许可证则不行。

感谢众多在 Guido 指导下工作的外部志愿者，使得这些发布成为可能。

C.2 获取或以其他方式使用 Python 的条款和条件

C.2.1 用于 PYTHON 3.6.15 的 PSF 许可协议

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.6.15 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.6.15 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2021 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.6.15 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.6.15 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   3.6.15.

4. PSF is making Python 3.6.15 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 3.6.15 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.6.15
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.6.15, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
```

```
6. This License Agreement will automatically terminate upon a material breach␣
↪of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 3.6.15, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 用于 PYTHON 2.0 的 BEOPEN.COM 许可协议

BEOPEN PYTHON 开源许可协议第 1 版

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
```

(下页继续)

(续上页)

```
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 用于 PYTHON 1.6.1 的 CNRI 许可协议

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
```

(下页继续)

(续上页)

```
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 用于 PYTHON 0.9.0 至 1.2 的 CWI 许可协议

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.3 被收录软件的许可证与鸣谢

本节是 Python 发行版中收录的第三方软件的许可和致谢清单，该清单是不完整且不断增长的。

C.3.1 Mersenne Twister

_random 模块包含基于 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 下载的代码。以下是原始代码的完整注释（声明）：

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.
```

(下页继续)

(续上页)

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 套接字

socket 模块使用 `getaddrinfo()` 和 `getnameinfo()` 函数，这些函数源代码在 WIDE 项目（http://www.wide.ad.jp/）的单独源文件中。

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

(下页继续)

(续上页)

```
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Floating point exception control

The source for the *fpectl* module includes the following notice:

```
  ---------------------------------------------------------------------
 /                       Copyright (c) 1996.                           \
|          The Regents of the University of California.                  |
|                         All rights reserved.                          |
|                                                                       |
|   Permission to use, copy, modify, and distribute this software for   |
|   any purpose without fee is hereby granted, provided that this en-   |
|   tire notice is included in all copies of any software which is or   |
|   includes  a  copy  or  modification  of  this software and in all   |
|   copies of the supporting documentation for such software.           |
|                                                                       |
|   This  work was produced at the University of California, Lawrence   |
|   Livermore National Laboratory under  contract  no.  W-7405-ENG-48   |
|   between  the  U.S.  Department  of  Energy and The Regents of the   |
|   University of California for the operation of UC LLNL.              |
|                                                                       |
|                              DISCLAIMER                               |
|                                                                       |
|   This  software was prepared as an account of work sponsored by an   |
|   agency of the United States Government. Neither the United States   |
|   Government  nor the University of California nor any of their em-   |
|   ployees, makes any warranty, express or implied, or  assumes  any   |
|   liability  or  responsibility  for the accuracy, completeness, or   |
|   usefulness of any information,  apparatus,  product,  or  process   |
|   disclosed,   or  represents  that  its  use  would  not  infringe   |
|   privately-owned rights. Reference herein to any specific  commer-   |
|   cial  products,  process,  or  service  by trade name, trademark,   |
|   manufacturer, or otherwise, does not  necessarily  constitute  or   |
|   imply  its endorsement, recommendation, or favoring by the United   |
|   States Government or the University of California. The views  and   |
|   opinions  of authors expressed herein do not necessarily state or   |
|   reflect those of the United States Government or  the  University   |
|   of  California,  and shall not be used for advertising or product   |
 \  endorsement purposes.                                              /
  ---------------------------------------------------------------------
```

C.3.4 异步套接字服务

asynchat and *asyncore* 模块包含以下声明:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Cookie 管理

http.cookies 模块包含以下声明:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.6 执行追踪

trace 模块包含以下声明:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.7 UUencode 与 UUdecode 函数

uu 模块包含以下声明:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
```

(下页继续)

(续上页)

```
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.8 XML 远程过程调用

xmlrpc.client 模块包含以下声明:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.9 test_epoll

`test_epoll` 模块包含以下声明:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(下页继续)

(续上页)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.10 Select kqueue

select 模块关于 kqueue 的接口包含以下声明:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.11 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski' implementation of Dan Bernstein's SipHash24 algorithm. The contains the following note:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>
```

(下页继续)

(续上页)

```
Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.12 strtod and dtoa

Python/dtoa.c 文件提供了 C 语言的 dtoa 和 strtod 函数，用于将 C 语言的双精度型和字符串进行转换，该文件由 David M. Gay 的同名文件派生而来，当前可从 http://www.netlib.org/fp/ 下载。2009 年 3 月 16 日检索到的原始文件包含以下版权和许可声明:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.13 OpenSSL

如果操作系统可用，则*hashlib*, *posix*, *ssl*, *crypt* 模块使用 OpenSSL 库来提高性能。此外，适用于 Python 的 Windows 和 Mac OS X 安装程序可能包括 OpenSSL 库的拷贝，所以在此处也列出了 OpenSSL 许可证的拷贝:

```
 LICENSE ISSUES
 ==============

 The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
 the OpenSSL License and the original SSLeay license apply to the toolkit.
 See below for the actual license texts. Actually both licenses are BSD-style
 Open Source licenses. In case of any license issues related to OpenSSL
 please contact openssl-core@openssl.org.

 OpenSSL License
 ---------------

   /* ====================================================================
```

(下页继续)

(续上页)

```
    * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
    *
    * 1. Redistributions of source code must retain the above copyright
    *    notice, this list of conditions and the following disclaimer.
    *
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in
    *    the documentation and/or other materials provided with the
    *    distribution.
    *
    * 3. All advertising materials mentioning features or use of this
    *    software must display the following acknowledgment:
    *    "This product includes software developed by the OpenSSL Project
    *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
    *
    * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
    *    endorse or promote products derived from this software without
    *    prior written permission. For written permission, please contact
    *    openssl-core@openssl.org.
    *
    * 5. Products derived from this software may not be called "OpenSSL"
    *    nor may "OpenSSL" appear in their names without prior written
    *    permission of the OpenSSL Project.
    *
    * 6. Redistributions of any form whatsoever must retain the following
    *    acknowledgment:
    *    "This product includes software developed by the OpenSSL Project
    *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
    *
    * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
    * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
    * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
    * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------
```

(下页继续)

(续上页)

```
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.  The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * The licence and distribution terms for any publically available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution licence
 * [including the GNU Public Licence.]
 */
```

C.3.14 expat

除非使用 --with-system-expat 配置了构建，否则 pyexpat 扩展都是用包含 expat 源的拷贝构建的:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.15 libffi

除非使用 --with-system-libffi 配置了构建，否则 _ctypes 扩展都是包含 libffi 源的拷贝构建的:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.16 zlib

如果系统上找到的 zlib 版本太旧而无法用于构建，则使用包含 zlib 源代码的拷贝来构建*zlib* 扩展:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.17 cfuhash

tracemalloc 使用的哈希表的实现基于 cfuhash 项目:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
```

(下页继续)

(续上页)

```
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.18 libmpdec

除非使用 `--with-system-libmpdec` 配置了构建，否则 `_decimal` 模块都是用包含 libmpdec 库的拷贝构建的。

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

APPENDIX D

Copyright

有关完整的许可证和许可信息，参见历史和许可证。

Bibliography

[Frie09] Friedl, Jeffrey. Mastering Regular Expressions. 第三版, O’Reilly Media, 2009. 第三版不再使用 Python, 但第一版提供了编写正则表达式的良好细节。

[C99] ISO/IEC 9899:1999. “Programming languages –C.” 该标准的公开草案可从 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf获得。

Python 模块索引

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

v

w

X

Z

索引

非字母

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z