The Python Library Reference

發🄵 *3.9.13*

Guido van Rossum
and the Python development team

9 月 06, 2022

Python Software Foundation
Email: docs@python.org

Contents

reference-index F明 Python 這門語言確切的文法及語意，而這份函式庫參考手F則是F明隨著 Python 一起發F的標准函式庫，除此之外，其F容也包含一些時常出現在 Python 發F版本中的非必要套件。

Python 的標准函式庫是非常龐大的，其提供了如下所述極多且涵蓋用途極廣的許多模組。包含一些用 C 語言撰寫，可以操作像是檔案讀寫等系統相關功能的F建模組，當然也有用 Python 撰寫，F使用標准解法解F許多常見問題的模組。其中有些模組則是特F針對 Python 的可F性去設計的，F此特地將一些平台特殊相依性的功能抽象化成可跨平台的 API。

Python 的 Windows 安裝檔基本上包含整個標准函式庫，且通常也包含許多附加的組件；而在類 Unix 作業系統方面，Python 通常是以一系列的套件被安裝，因此對於某些或全部的可選組件，可能都必須使用該作業系統提供的套件管理工具來安裝。

在標准函式庫之外，還有成千上萬且不斷增加的組件（從個F的程式、模組、套件到完整的應用程式開發框架），可以從 Python 套件索引 (Python Package Index) 中取得。

CHAPTER 1

簡介

「Python 函式庫」包含了許多不同的部分。

函式庫中包括被視🄵程式語言「核心」部分的資料型態，像是數字 (number) 或是串列 (list)。對於這些型🄵，Python 核心對這些字面值 (literal) 的形式做定義，🄵對它們的語意制定了一些限制，但在此同時🄵不把文字對應的語意完全定義。（另一方面，Python 在語法面上有確實的定義，例如拼字或是運算元次序）

Python 函式庫也囊括了🄵建函數與例外處理——這些物件都可以不用透過 `import` 陳述式來引入 Python 程式中就能使用。函式庫中有部份是被 Python 核心所定義的，但在這🄵僅解釋最核心的語意部分。

整個函式庫中包含了許多模組，有許多方法可以從函式庫中取用這些模組。有些模組是以 C 語言撰寫🄵建置於 Python 編譯器之中，其他的是由 Python 撰寫🄵以源碼的方式 (source form) 引入。有些模組提供的功能是專屬於 Python 的，像是把 stack trace 印出來；有些則是針對特定作業系統，去試著存取特定硬體；還有些提供對特定應用的功能與操作介面，像是 World Wide Web。模組的使用情🄵會因機器與 Python 的版本而不同，部分模組是開放所有版本以及 Port 的 Python 來使用的，但有些會因系統支援或需求在某些版本或系統下無法使用，甚至有些僅限在特定的設定環境下才能使用。

這個手🄵會「深入淺出」地介紹 Python 函式庫。它會先介紹一些🄵建函式、資料型態、和一些例外處理，再來一章章的主題式介紹相關模組。

這代表如果你從這個手🄵的最開始讀起，🄵在感到無聊時跳到下一個章節，你仍然可以得到一個對 Python 函式庫所支援的模組與其合理應用的概觀。當然，你🄵有必要像是在讀一本小🄵一樣讀這本手🄵——你可以快速🄵覽目🄵（在手🄵的最前頭）、或是你可以利用最後面的索引來查詢特定的函式或模組。最後，如果你享受🄵讀一些隨機的主題，你可以選擇一個隨機的數字🄵開始🄵讀（見 *random* 模組）。不管你想要以什🄵順序來🄵讀這個手🄵，🄵建函式會是一個很好的入門，因🄵手🄵中其他章節都預設你已經對這個章節有一定的熟悉程度。

讓我們開始吧！

1.1 可用性之🄵釋

- 如果出現「適用：Unix」🄵釋，則代表該函數普遍存在於 Unix 系統中，但這🄵不保證其存在於某特定作業系統。
- 如果🄵有分🄵🄵釋的話，有標明「適用：Unix」🄵釋的所有函式也都於 macOS 上支援，因其建於 Unix 核心之上。

CHAPTER 2

🄵建函式

Python 直譯器有🄵建數十個函式，隨時都可以使用這些函式。以下按照英文字母排序列出。

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

abs(*x*)
返回一个数的绝对值。参数可以是整数、浮点数或任何实现了 `__abs__()` 的对象。如果参数是一个复数，则返回它的模。

all(*iterable*)
如果 *iterable* 的所有元素均为真值（或可迭代对象为空）则返回 `True` 。等价于:

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

any(*iterable*)
如果 *iterable* 的任一元素为真值则返回 `True`。如果可迭代对象为空，返回 `False`。等价于:

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

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

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

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

如果不一定需要前缀“0b”，还可以使用如下的方法。

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

另见*format()* 获取更多信息。

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

3.7 版更變: *x* 现在只能作为位置参数。

breakpoint(**args*, ***kws*)
此函数会在调用时将你陷入调试器中。具体来说，它调用*sys.breakpointhook()* ，直接传递 `args` 和 `kws` 。默认情况下，`sys.breakpointhook()` 调用*pdb.set_trace()* 且没有参数。在这种情况下，它纯粹是一个便利函数，因此您不必显式导入*pdb* 且键入尽可能少的代码即可进入调试器。但是，*sys.breakpointhook()* 可以设置为其他一些函数并被*breakpoint()* 自动调用，以允许进入你想用的调试器。

引发一个审计事件 `builtins.breakpoint` 并附带参数 `breakpointhook`。

3.7 版新加入.

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

可选形参 *source* 可以用不同的方式来初始化数组:

- 如果是一个 *string*，您必须提供 *encoding* 参数（*errors* 参数仍是可选的）；*bytearray()* 会使用*str.encode()* 方法来将 string 转变成 bytes。
- 如果是一个 *integer*，会初始化大小为该数字的数组，并使用 null 字节填充。

- 如果是一个遵循 缓冲区接口的对象，该对象的只读缓冲区将被用来初始化字节数组。
- 如果是一个 *iterable* 可迭代对象，它的元素的范围必须是 `0 <= x < 256` 的整数，它会被用作数组的初始内容。

如果没有实参，则创建大小为 0 的数组。

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

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

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

因此，构造函数的实参和*bytearray()* 相同。

字节对象还可以用字面值创建，参见 strings。

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

callable(*object*)

如果参数 *object* 是可调用的就返回*True*，否则返回*False*。如果返回 `True`，调用仍可能失败，但如果返回 `False`，则调用 *object* 将肯定不会成功。请注意类是可调用的（调用类将返回一个新的实例）；如果实例所属的类有 `__call__()` 则它就是可调用的。

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

chr(*i*)

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

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

@classmethod

把一个方法封装成类方法。

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

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

`@classmethod` 这样的形式称为函数的*decorator* -- 详情参阅 function。

类方法的调用可以在类上进行 (例如 `C.f()`) 也可以在实例上进行 (例如 `C().f()`)。其所属类以外的类实例会被忽略。如果类方法在其所属类的派生类上调用，则该派生类对象会被作为隐含的第一个参数被传入。

类方法与 C++ 或 Java 中的静态方法不同。如果你需要后者，请参阅本节中的*staticmethod()*。有关类方法的更多信息，请参阅 types。

3.9 版更變: 类方法现在可以包装其他描述器 例如*property()*。

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

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

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

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

可选参数 *flags* 和 *dont_inherit* 控制应当激活哪个编译器选项 以及应当允许哪个 future 特性。如果两者都未提供 (或都为零) 则代码会应用与调用 *compile()* 的代码相同的旗标来编译。如果给出了 *flags* 参数而未给出 *dont_inherit* (或者为零) 则会在无论如何都将被使用的旗标之外还会额外使用 *flags* 参数所指定的编译器选项和 future 语句。如果 *dont_inherit* 为非零整数，则只使用 *flags* 参数 -- 外围代码中的旗标 (future 特性和编译器选项) 会被忽略。

编译器选项和 future 语句是由比特位来指明的。比特位可以通过一起按位 OR 来指明多个选项。指明特定 future 特性所需的比特位可以在 *__future__* 模块的 _Feature 实例的 compiler_flag 属性中找到。编译器旗标 可以在 *ast* 模块中查找带有 PyCF_ 前缀的名称。

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

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

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

引发一个审计事件 compile 附带参数 source, filename。

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

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

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

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

3.8 版新加入: ast.PyCF_ALLOW_TOP_LEVEL_AWAIT 现在可在旗标中传入以启用对最高层级 await, async for 和 async with 的支持。

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

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

对于一个普通 Python 对象 x，complex(x) 会委托给 x.__complex__()。如果 __complex__() 未定义则将回退至 __float__()。如果 __float__() 未定义则将回退至 __index__()。

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

数字类型 --- int, float, complex 描述了复数类型。

3.6 版更變: 您可以使用下划线将代码文字中的数字进行分组。

3.8 版更變: 如果 __complex__() 和 __float__() 未定义则回退至 __index__()。

delattr(*object*, *name*)

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

class dict(***kwarg*)
class dict(*mapping*, ***kwarg*)
class dict(*iterable*, ***kwarg*)
创建一个新的字典。*dict* 对象是一个字典类。参见*dict* 和*映射类型 --- dict* 了解这个类。

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

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

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

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

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

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

返回的列表按字母表排序。例如：

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

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

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

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

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

等价于:

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

eval(*expression*[, *globals*[, *locals*]])

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

expression 参数会作为一个 Python 表达式（从技术上说是一个条件列表）被解析并求值，并使用 *globals* 和 *locals* 字典作为全局和局部命名空间。如果 *globals* 字典存在且不包含以 `__builtins__` 为键的值，则会在解析 *expression* 之前插入以此为键的对内置模块*builtins* 的引用。这意味着 *expression* 通常具有对标准*builtins* 模块的完全访问权限且受限的环境会被传播。如果省略 *locals* 字典则其默认值为 *globals* 字典。如果两个字典同时省略，则表达式执行时会使用*eval()* 被调用的环境中的 *globals* 和 *locals*。请注意，*eval()* 并没有对外围环境下的 (非局部) 嵌套作用域 的访问权限。

返回值就是表达式的求值结果。语法错误将作为异常被报告。例如:

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

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

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

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

引发一个审计事件 `exec` 附带参数 `code_object`。

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

This function supports dynamic execution of Python code. *object* must be either a string or a code object. If it is a string, the string is parsed as a suite of Python statements which is then executed (unless a syntax error occurs).[1] If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file input (see the section file-input in the Reference Manual). Be aware that the `nonlocal`, `yield`, and `return` statements may not be used outside of function definitions even within the context of code passed to the *exec()* function. The return value is `None`.

无论哪种情况，如果省略了可选项，代码将在当前作用域内执行。如果只提供了 *globals*，则它必须是一个字典（不能是字典的子类），该字典将同时被用于全局和局部变量。如果同时提供了 *globals* 和 *locals*，它们会分别被用于全局和局部变量。如果提供了 *locals*，则它可以是任何映射对象。请记住在模块层级上，globals 和 locals 是同一个字典。如果 exec 得到两个单独对象作为 *globals* 和 *locals*，则代码将如同嵌入类定义的情况一样执行。

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

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

引发一个审计事件 exec 附带参数 code_object。

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

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

---

filter(*function*, *iterable*)

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

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

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

class float([*x*])

返回从数字或字符串 *x* 生成的浮点数。

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

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

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

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

对于一个普通 Python 对象 x，float(x) 会委托给 x.__float__()。如果 __float__() 未定义则将回退至 __index__()。

如果没有实参，则返回 0.0 。

例如:

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

(下页继续)

(繼續上一頁)

```
>>> float('-Infinity')
-inf
```

数字类型 --- int, float, complex 描述了浮点类型。

3.6 版更變: 您可以使用下划线将代码文字中的数字进行分组。

3.7 版更變: *x* 现在只能作为位置参数。

3.8 版更變: 如果 `__float__()` 未定义则回退至 `__index__()`。

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

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

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

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

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

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

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

備𝔽: 由于 私有名称混合发生在编译时，因此必须手动混合私有属性（以两个下划线打头的属性）名称以使使用 *getattr()* 来提取它。

---

globals()
Return the dictionary implementing the current module namespace. For code within functions, this is set when the function is defined and remains the same regardless of where the function is called.

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

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

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

---

help([*object*])
启动内置的帮助系统（此函数主要在交互式中使用）。如果没有实参，解释器控制台里会启动交互式帮

助系统。如果实参是一个字符串，则在模块、函数、类、方法、关键字或文档主题中搜索该字符串，并在控制台上打印帮助信息。如果实参是其他任意对象，则会生成该对象的帮助页。

请注意如果在函数的形参列表中出现了斜杠 (/)，则它在发起调用*help()* 的时候意味着斜杠之前的均为仅限位置形参。更多相关信息，请参阅 有关仅限位置形参的 FAQ 条目。

该函数通过*site* 模块加入到内置命名空间。

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

hex(*x*)

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

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

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

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

另见*format()* 获取更多信息。

另请参阅*int()* 将十六进制字符串转换为以 16 为基数的整数。

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

---

id(*object*)

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

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

引发一个审计事件 `builtins.id`，附带参数 `id`。

input([*prompt*])

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

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

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

引发一个审计事件 `builtins.input` 附带参数 `prompt`。

在成功读取输入之后引发一个审计事件 `builtins.input/result` 附带结果。

class int([*x*])

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

返回一个基于数字或字符串 *x* 构造的整数对象，或者在未给出参数时返回 `0`。如果 *x* 定义了 `__int__()`，

int(x) 将返回 x.__int__()。如果 *x* 定义了 __index__()，它将返回 x.__index__()。如果 *x* 定义了 __trunc__()，它将返回 x.__trunc__()。对于浮点数，它将向零舍入。

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

整数类型定义请参阅*数字类型 --- int, float, complex* 。

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

3.6 版更變: 您可以使用下划线将代码文字中的数字进行分组。

3.7 版更變: *x* 现在只能作为位置参数。

3.8 版更變: 如果 __int__() 未定义则回退至 __index__()。

3.9.14 版更變: *int* string inputs and string representations can be limited to help avoid denial of service attacks. A *ValueError* is raised when the limit is exceeded while converting a string *x* to an *int* or when converting an *int* into a string would exceed the limit. See the *integer string conversion length limitation* documentation.

isinstance(*object*, *classinfo*)
如果参数 *object* 是参数 *classinfo* 的实例或者是其 (直接、间接或虚拟) 子类则返回 True。如果 *object* 不是给定类型的对象，函数将总是返回 False。如果 *classinfo* 是类型对象元组（或由其他此类元组递归组成的元组），那么如果 *object* 是其中任何一个类型的实例就返回 True。如果 *classinfo* 既不是类型，也不是类型元组或类型元组的元组，则将引发*TypeError* 异常。

issubclass(*class*, *classinfo*)
Return True if *class* is a subclass (direct, indirect or *virtual*) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects (or recursively, other such tuples), in which case return True if *class* is a subclass of any entry in *classinfo*. In any other case, a *TypeError* exception is raised.

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

另请参阅迭代器类型。

适合*iter()* 的第二种形式的应用之一是构建块读取器。例如，从二进制数据库文件中读取固定宽度的块，直至到达文件的末尾:

```
from functools import partial
with open('mydata.db', 'rb') as f:
    for block in iter(partial(f.read, 64), b''):
        process_block(block)
```

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

CPython implementation detail: len 对于大于*sys.maxsize* 的长度如*range(2 ** 100)* 会引发*OverflowError*。

class list([*iterable*])
虽然被称为函数，`list` 实际上是一种可变序列类型，详情请参阅*List（串列）* 和*序列类型 --- list, tuple, range*。

locals()
更新并返回表示当前本地符号表的字典。在函数代码块但不是类代码块中调用`locals()` 时将返回自由变量。请注意在模块层级上，`locals()` 和`globals()` 是同一个字典。

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

---

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

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

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

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

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

3.4 版新加入: keyword-only 实参 *default* 。

3.8 版更變: *key* 可以为 `None`。

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

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

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

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

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

3.4 版新加入: keyword-only 实参 *default* 。

3.8 版更變: *key* 可以为 `None`。

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

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

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

oct(*x*)

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

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

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

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

另见*format()* 获取更多信息。

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

打开 *file* 并返回对应的*file object*。如果该文件不能被打开，则引发*OSError*。请参阅 tut-files 获取此函数的更多用法示例。

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

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

字符	意义
`'r'`	读取（默认）
`'w'`	写入，并先截断文件
`'x'`	排它性创建，如果文件已存在则失败
`'a'`	写入，如果文件存在则在末尾追加
`'b'`	二进制模式
`'t'`	文本模式（默认）
`'+'`	打开用于更新（读取与写入）

默认模式为 `'r'` (打开用于读取文本，与 `'rt'` 同义)。模式 `'w+'` 与 `'w+b'` 将打开文件并清空内容。模式 `'r+'` 与 `'r+b'` 将打开文件并不清空内容。

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

此外还允许使用一个模式字符 'U'，该字符已不再具有任何效果，并被视为已弃用。之前它会在文本模式中启用*universal newlines*，这在 Python 3.0 中成为默认行为。请参阅*newline* 形参的文档了解更多细节。

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

---

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size in bytes of a fixed-size chunk buffer. Note that specifying a buffer size this way applies for binary buffered I/O, but `TextIOWrapper` (i.e., files opened with `mode='r+'`) would have another buffering. To disable buffering in `TextIOWrapper`, consider using the `write_through` flag for *io.TextIOWrapper.reconfigure()*. When no *buffering* argument is given, the default buffering policy works as follows:

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

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

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

- 如果存在编码错误，`'strict'` 会引发*ValueError* 异常。默认值 `None` 具有相同的效果。
- `'ignore'` 忽略错误。请注意，忽略编码错误可能会导致数据丢失。
- `'replace'` 会将替换标记（例如 `'?'` ）插入有错误数据的地方。
- `'surrogateescape'` 将把任何不正确的字节表示为 U+DC80 至 U+DCFF 范围内的下方替代码位。当在写入数据时使用 `surrogateescape` 错误处理句柄时这些替代码位会被转回到相同的字节。这适用于处理具有未知编码格式的文件。
- 只有在写入文件时才支持 `'xmlcharrefreplace'`。编码不支持的字符将替换为相应的 XML 字符引用 `&#nnn;`。
- `'backslashreplace'` 用 Python 的反向转义序列替换格式错误的数据。
- `'namereplace'` （也只在编写时支持）用 `\N{...}` 转义序列替换不支持的字符。

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

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

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

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

新创建的文件是*不可继承的*。

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

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

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

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

引发一个*审计事件* open 附带参数 file, mode, flags。

mode 与 flags 参数可以在原始调用的基础上被修改或传递。

3.3 版更變:

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

3.4 版更變:

- 文件现在禁止继承。

Deprecated since version 3.4, will be removed in version 3.10: 'U' 模式。

3.5 版更變:

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

3.6 版更變:

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

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

pow(*base*, *exp*[, *mod*])
返回 *base* 的 *exp* 次幂；如果 *mod* 存在，则返回 *base* 的 *exp* 次幂对 *mod* 取余（比 `pow(base, exp) % mod` 更高效）。两参数形式 `pow(base, exp)` 等价于乘方运算符: `base**exp`。

The arguments must have numeric types. With mixed operand types, the coercion rules for binary arithmetic operators apply. For *int* operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `pow(10, 2)` returns `100`, but `pow(10, -2)` returns `0.01`. For a negative base of type *int* or *float* and a non-integral exponent, a complex result is delivered. For example, `pow(-9, 0.5)` returns a value close to `3j`.

对于*int* 操作数 *base* 和 *exp*，如果给出 *mod*，则 *mod* 必须为整数类型并且 *mod* 必须不为零。如果给出 *mod* 并且 *exp* 为负值，则 *base* 必须相对于 *mod* 不可整除。在这种情况下，将会返回 `pow(inv_base, -exp, mod)`，其中 *inv_base* 为 *base* 的倒数对 *mod* 取余。

下面的例子是 `38` 的倒数对 `97` 取余:

```
>>> pow(38, -1, mod=97)
23
>>> 23 * 38 % 97 == 1
True
```

3.8 版更變: 对于*int* 操作数，三参数形式的 `pow` 现在允许第二个参数为负值，即可以计算倒数的余数。

3.8 版更變: 允许关键字参数。之前只支持位置参数。

print(**objects*, *sep=' '*, *end='\n'*, *file=sys.stdout*, *flush=False*)
将 *objects* 打印到 *file* 指定的文本流，以 *sep* 分隔并在末尾加上 *end*。*sep*, *end*, *file* 和 *flush* 如果存在，它们必须以关键字参数的形式给出。

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

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

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

3.3 版更變: 增加了 *flush* 关键字参数。

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

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

一个典型的用法是定义一个托管属性 `x`:

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

    def getx(self):
        return self._x

    def setx(self, value):
```

(下页继续)

(繼續上一頁)

```
        self._x = value

    def delx(self):
        del self._x

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

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

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

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

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

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

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

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

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

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

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

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

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

3.5 版更變: 特征属性对象的文档字符串现在是可写的。

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

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

repr(*object*)

返回包含一个对象的可打印表示形式的字符串。对于许多类型来说，该函数会尝试返回的字符串将会与该对象被传递给*eval()* 时所生成的对象具有相同的值，在其他情况下表示形式会是一个括在尖括

号中的字符串，其中包含对象类型的名称与通常包括对象名称和地址的附加信息。类可以通过定义 `__repr__()` 方法来控制此函数为它的实例所返回的内容。

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

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

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

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

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

---

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

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

setattr(*object*, *name*, *value*)
此函数与*getattr()* 两相对应。其参数为一个对象、一个字符串和一个任意值。字符串指定一个现有属性或者新增属性。函数会将值赋给该属性，只要对象允许这种操作。例如，`setattr(x, 'foobar', 123)` 等价于 `x.foobar = 123`。

備🄵: 由于 私有名称混合发生在编译时，因此必须手动混合私有属性（以两个下划线打头的属性）名称以便使用*setattr()* 来设置它。

---

class slice(*stop*)
class slice(*start*, *stop*[, *step*])
Return a *slice* object representing the set of indices specified by `range(start, stop, step)`. The *start* and *step* arguments default to `None`. Slice objects have read-only data attributes `start`, `stop` and `step` which merely return the argument values (or their default). They have no other explicit functionality; however they are used by NumPy and other third party packages. Slice objects are also generated when extended indexing syntax is used. For example: `a[start:stop:step]` or `a[start:stop, i]`. See *itertools.islice()* for an alternate version that returns an iterator.

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

有兩個選擇性參數，只能使用關鍵字參數指定。

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

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

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

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

The sort algorithm uses only < comparisons between items. While defining an `__lt__()` method will suffice for sorting, **PEP 8** recommends that all six rich comparisons be implemented. This will help avoid bugs when using the same data with other ordering tools such as *max()* that rely on a different underlying method. Implementing all six comparisons also helps avoid confusion for mixed type comparisons which can call reflected the `__gt__()` method.

有关排序示例和简要排序教程，请参阅 sortinghowto 。

@staticmethod

将方法转换为静态方法。

静态方法不会接收隐式的第一个参数。要声明一个静态方法，请使用此语法

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

`@staticmethod` 这样的形式称为函数的*decorator* -- 详情参阅 function。

静态方法的调用可以在类上进行 (例如 `C.f()`) 也可以在实例上进行 (例如 `C().f()`)。

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

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

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

想了解更多有关静态方法的信息，请参阅 types 。

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

返回一个*str* 版本的 *object* 。有关详细信息，请参阅*str()* 。

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

sum(*iterable*, */*, *start=0*)

从 *start* 开始自左向右对 *iterable* 的项求和并返回总计值。*iterable* 的项通常为数字，而 start 值则不允许为字符串。

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

3.8 版更變: *start* 形参可用关键字参数形式来指定。

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

返回一个代理对象，它会将方法调用委托给 *type* 的父类或兄弟类。这对于访问已在类中被重载的继承方法很有用。

object-or-type 确定用于搜索的*method resolution order*。搜索会从 *type* 之后的类开始。

举例来说，如果 *object-or-type* 的*__mro__* 为 `D -> B -> C -> A -> object` 并且 *type* 的值为 `B`，则*super()* 将会搜索 `C -> A -> object`。

object-or-type 的*__mro__* 属性列出了*getattr()* 和*super()* 所共同使用的方法解析搜索顺序。该属性是动态的，可以在任何继承层级结构发生更新的时候被改变。

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

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

第二个用例是在动态执行环境中支持协作多重继承。此用例为 Python 所独有而不存在于静态编码语言或仅支持单继承的语言当中。这使用实现“菱形图”成为可能，即有多个基类实现相同的方法。好的设计强制要求这样的方法在每个情况下都具有相同的调用签名（因为调用顺序是在运行时确定的，也因为这个顺序要适应类层级结构的更改，还因为这个顺序可能包括在运行时之前未知的兄弟类）。

对于以上两个用例，典型的超类调用看起来是这样的:

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

除了方法查找之外，*super()* 也可用于属性查找。一个可能的应用场合是在上级或同级类中调用描述器。

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

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

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

class tuple([*iterable*])

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

class type(*object*)

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

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

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

传入三个参数时，返回一个新的 type 对象。这在本质上是 `class` 语句的一种动态形式，*name* 字符串即类名并会成为 *__name__* 属性；*bases* 元组包含基类并会成为 *__bases__* 属性；如果为空则会添加所有类的终极基类 *object*。*dict* 字典包含类主体的属性和方法定义；它在成为 *__dict__* 属性之前可能会被拷贝或包装。下面两条语句会创建相同的 *type* 对象:

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

另请参阅类型对象。

提供给三参数形式的关键字参数会被传递给适当的元类机制 (通常为 `__init_subclass__()`)，相当于类定义中关键字 (除了 *metaclass*) 的行为方式。

另请参阅 class-customization。

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

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

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

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

如果指定了一个对象但它没有*__dict__* 属性（例如，当它所属的类定义了 `__slots__` 属性时）则会引发*TypeError* 异常。

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

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

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

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

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

zip() 与 * 运算符相结合可以用来拆解一个列表:

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

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

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

此函数会由 import 语句发起调用。它可以被替换 (通过导入*builtins* 模块并赋值给 builtins.__import__) 以便修改 import 语句的语义，但是 **强烈**不建议这样做，因为使用导入钩子 (参见 **PEP 302**) 通常更容易实现同样的目标，并且不会导致代码问题，因为许多代码都会假定所用的是默认实现。同样也不建议直接使用*__import__()* 而应该用*importlib.import_module()*。

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

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

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

例如，语句 import spam 的结果将为与以下代码作用相同的字节码:

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

语句 import spam.ham 的结果将为以下调用:

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

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

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

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

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

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

3.3 版更變: *level* 的值不再支持负数（默认值也修改为 0）。

3.9 版更變: 当使用了命令行参数 -E 或 -I 时，环境变量 PYTHONCASEOK 现在将被忽略。

𝔽解

CHAPTER 3

🄵建常數

有少数的常量存在于内置命名空间中。它们是：

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

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

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

NotImplemented
: 双目运算特殊方法（如 `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()` 等）应返回的特殊值，用于表示运算没有针对其他类型的实现；也可由原地双目运算特殊方法（如 `__imul__()`, `__iand__()` 等）出于同样的目的而返回。它不应被作为布尔值来解读。

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

参见实现算术运算 为例。

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

3.9 版更變: 作为布尔值来解读 `NotImplemented` 已被弃用。虽然它目前会被解读为真值，但将同时发出*DeprecationWarning*。它将在未来的 Python 版本中引发*TypeError*。

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

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

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

3.1 由 site 模块添加的常量

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

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

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

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

CHAPTER 4

🄵建型態

以下部分描述了解释器中内置的标准类型。

主要内置类型有数字、序列、映射、类、实例和异常。

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

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

4.1 逻辑值检测

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

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

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

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

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

4.2 布尔运算 --- and, or, not

这些属于布尔运算，按优先级升序排列:

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

F解:

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

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

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

4.3 比较

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

此表格汇总了比较运算:

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

除不同的数字类型外，不同类型的对象不能进行相等比较。== 运算符总有定义，但对于某些对象类型（例如，类对象），它等于 `is` 。其他 <、<=、> 和 >= 运算符仅在有意义的地方定义。例如，当参与比较的参数之一为复数时，它们会抛出 *`TypeError`* 异常。

Non-identical instances of a class normally compare as non-equal unless the class defines the `__eq__()` method.

Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()` (in general, `__lt__()` and `__eq__()` are sufficient, if you want the conventional meanings of the comparison operators).

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

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

4.4 数字类型 --- int, float, complex

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

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

Python 完全支持混合运算：当一个二元算术运算符的操作数有不同数值类型时，" 较窄" 类型的操作数会拓宽到另一个操作数的类型，其中整数比浮点数窄，浮点数比复数窄。不同类型的数字之间的比较，同比较这些数字的精确值一样。[2]

构造函数*int()*、*float()* 和*complex()* 可以用来构造特定类型的数字。

所有数字类型（复数除外）都支持下列运算（有关运算优先级，请参阅：operator-summary）：

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

發F解:

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

(2) 不可用于复数。而应在适当条件下使用*abs()* 转换为浮点数。

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

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

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

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

请参阅 https://www.unicode.org/Public/13.0.0/ucd/extracted/DerivedNumericType.txt 查看具有 Nd 特征属性的代码点的完整列表。

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

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

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

有关更多的数字运算请参阅*math* 和*cmath* 模块。

4.4.1 整数类型的按位运算

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

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

此表格是以优先级升序排序的按位运算列表:

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

F解:

(1) 负的移位数是非法的，会导致引发*ValueError*。

(2) 左移 *n* 位等价于乘以 `pow(2, n)` 。

(3) 右移 *n* 位等价于除以 `pow(2, n)` ，作向下取整除法。

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

4.4.2 整数类型的附加方法

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

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

返回以二进制表示一个整数所需要的位数，不包括符号位和前面的零:

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

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

等价于:

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

3.1 版新加入.

int.**to_bytes**(*length*, *byteorder*, *, *signed=False*)
返回表示一个整数的字节数组。

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

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

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

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

3.2 版新加入.

classmethod int.**from_bytes**(*bytes*, *byteorder*, *, *signed=False*)
返回由给定字节数组所表示的整数。

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

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

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

signed 参数指明是否使用二的补码来表示整数。

3.2 版新加入.

int.**as_integer_ratio**()
返回一对整数，其比率正好等于原整数并且分母为正数。整数的比率总是用这个整数本身作为分子，1 作为分母。

3.8 版新加入.

4.4.3 浮点类型的附加方法

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

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

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

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

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

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

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

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

十六进制字符串采用的形式为:

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

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

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

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

对 3740.0 应用反向转换会得到另一个代表相同数值的十六进制数字符串:

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

4.4.4 数字类型的哈希运算

For numbers `x` and `y`, possibly of different types, it's a requirement that `hash(x) == hash(y)` whenever `x == y` (see the `__hash__()` method documentation for more details). For ease of implementation and efficiency across a variety of numeric types (including *int*, *float*, *decimal.Decimal* and *fractions.Fraction*) Python's hash for numeric types is based on a single mathematical function that's defined for any rational number, and hence applies to all instances of *int* and *fractions.Fraction*, and all finite instances of *float* and *decimal.Decimal*. Essentially, this function is given by reduction modulo `P` for a fixed prime `P`. The value of `P` is made available to Python as the `modulus` attribute of *sys.hash_info*.

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

详细规则如下所述:

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

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

```
import sys, math

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

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

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

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

(下页继续)

(繼續上一頁)

```
    return hash_value

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

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

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

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

4.5 迭代器类型

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

容器对象要提供迭代支持，必须定义一个方法:

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

迭代器对象自身需要支持以下两个方法，它们共同组成了 *迭代器协议*:

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

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

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

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

4.5.1 生成器类型

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

4.6 序列类型 --- list, tuple, range

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

4.6.1 通用序列操作

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

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

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

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

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

發F解：

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

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

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

[3] 它们必须如此，因为解析器无法区分这些操作数的类型。

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

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

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

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

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

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

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

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

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

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

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

4.6.2 不可变序列类型

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

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

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

4.6.3 可变序列类型

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

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

运算	结果	F解
`s[i] = x`	将 *s* 的第 *i* 项替换为 *x*	
`s[i:j] = t`	将 *s* 从 *i* 到 *j* 的切片替换为可迭代对象 *t* 的内容	
`del s[i:j]`	等同于 `s[i:j] = []`	
`s[i:j:k] = t`	将 `s[i:j:k]` 的元素替换为 *t* 的元素	(1)
`del s[i:j:k]`	从列表中移除 `s[i:j:k]` 的元素	
`s.append(x)`	将 *x* 添加到序列的末尾 (等同于 `s[len(s):len(s)] = [x]`)	
`s.clear()`	从 *s* 中移除所有项 (等同于 `del s[:]`)	(5)
`s.copy()`	创建 *s* 的浅拷贝 (等同于 `s[:]`)	(5)
`s.extend(t)` 或 `s += t`	用 *t* 的内容扩展 *s* (基本上等同于 `s[len(s):len(s)] = t`)	
`s *= n`	使用 *s* 的内容重复 *n* 次来对其进行更新	(6)
`s.insert(i, x)`	在由 *i* 给出的索引位置将 *x* 插入 *s* (等同于 `s[i:i] = [x]`)	
`s.pop()` 或 `s.pop(i)`	提取在 *i* 位置上的项，并将其从 *s* 中移除	(2)
`s.remove(x)`	删除 *s* 中第一个 `s[i]` 等于 *x* 的项目。	(3)
`s.reverse()`	就地将列表中的元素逆序。	(4)

F解:

(1) *t* 必须与它所替换的切片具有相同的长度。

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

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

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

(5) 包括 `clear()` 和 `copy()` 是为了与不支持切片操作的可变容器 (例如*dict* 和*set*) 的接口保持一致。`copy()` 不是*collections.abc.MutableSequence* ABC 的一部分，但大多数具体的可变序列类都提供了它。

3.3 版新加入: `clear()` 和 `copy()` 方法。

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

4.6.4 List（串列）

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

class list([*iterable*])

可以用多种方式构建列表：

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

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

其它许多操作也会产生列表，包括*sorted()* 内置函数。

列表实现了所有一般 和可变 序列的操作。列表还额外提供了以下方法：

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

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

sort() 接受两个仅限以关键字形式传入的参数 (仅限关键字参数):

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

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

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

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

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

有关排序示例和简要排序教程，请参阅 sortinghowto 。

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

4.6.5 元组

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

class tuple([*iterable*])

可以用多种方式构建元组：

- 使用一对圆括号来表示空元组: `()`
- 使用一个后缀的逗号来表示单元组: `a,` 或 `(a,)`

- 使用以逗号分隔的多个项: `a, b, c` or `(a, b, c)`
- 使用内置的*tuple()*: `tuple()` 或 `tuple(iterable)`

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

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

元组实现了所有一般 序列的操作。

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

4.6.6 range 对象

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

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

The arguments to the range constructor must be integers (either built-in *int* or any object that implements the `__index__()` special method). If the *step* argument is omitted, it defaults to `1`. If the *start* argument is omitted, it defaults to `0`. If *step* is zero, *ValueError* is raised.

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

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

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

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

一些 range 对象的例子:

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

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

start
start 形参的值 (如果该形参未提供则为 0)

stop
stop 形参的值

step
step 形参的值 (如果该形参未提供则为 1)

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

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

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

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

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

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

3.3 版新加入: *start*, *stop* 和*step* 属性。

也参考:

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

4.7 文本序列类型 --- str

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

- 单引号: '允许包含有" 双" 引号'
- Double quotes: "allows embedded 'single' quotes"
- 三重引号: '''三重单引号''', """ 三重双引号"""

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

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

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

字符串也可以通过使用*str* 构造器从其他对象创建。

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

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

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

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

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

If neither *encoding* nor *errors* is given, str(object) returns type(object).__str__(object), which is the "informal" or nicely printable string representation of *object*. For string objects, this is the string itself. If *object* does not have a __str__() method, then *str()* falls back to returning *repr(object)*.

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

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

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

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

4.7.1 字符串的方法

字符串实现了所有一般 序列的操作，还额外提供了以下列出的一些附加方法。

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

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

str.**capitalize**()

返回原字符串的副本，其首个字符大写，其余为小写。

3.8 版更變: 第一个字符现在被放入了 titlecase 而不是 uppercase。这意味着复合字母类字符将只有首个字母改为大写，而再不是全部字符大写。

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

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

消除大小写算法的描述请参见 Unicode 标准的 3.13 节。

3.3 版新加入.

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

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

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

errors 参数默认不会被检查，以获得最佳性能，而只在第一次编码错误时使用。启用*Python 开发模式*，或者使用调试构建来检查 *errors*。

3.1 版更變: 加入了对关键字参数的支持。

3.9 版更變: *errors* 现在在开发模式和调试模式下都会被检查。

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

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

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

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

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

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

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

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

请参阅格式字符串语法 了解有关可以在格式字符串中指定的各种格式选项的说明。

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

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

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

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

3.2 版新加入.

str.**index**(*sub*[, *start*[, *end*]])
类似于*find()*，但在找不到子类时会引发*ValueError*。

str.**isalnum**()
如果字符串中的所有字符都是字母或数字且至少有一个字符，则返回 True，否则返回 False。如果 `c.isalpha()`，`c.isdecimal()`，`c.isdigit()`，或 `c.isnumeric()` 之中有一个返回 True，则字符 “c“是字母或数字。

str.**isalpha**()
如果字符串中的所有字符都是字母，并且至少有一个字符，返回 True，否则返回 False。字母字符是指那些在 Unicode 字符数据库中定义为”Letter” 的字符，即那些具有”Lm”、”Lt”、”Lu”、”Ll” 或”Lo” 之一的通用类别属性的字符。注意，这与 Unicode 标准中定义的” 字母” 属性不同。

str.**isascii**()
如果字符串为空或字符串中的所有字符都是 ASCII，返回 True，否则返回 False。ASCII 字符的码点范围是 U+0000-U+007F。

3.7 版新加入.

str.**isdecimal**()
如果字符串中的所有字符都是十进制字符且该字符串至少有一个字符，则返回 True，否则返回 False。十进制字符指那些可以用来组成 10 进制数字的字符，例如 U+0660，即阿拉伯字母数字 0。严格地讲，十进制字符是 Unicode 通用类别”Nd” 中的一个字符。

str.**isdigit**()
如果字符串中的所有字符都是数字，并且至少有一个字符，返回 True ，否则返回 False 。数字包括十进制字符和需要特殊处理的数字，如兼容性上标数字。这包括了不能用来组成 10 进制数的数字，如 Kharosthi 数。严格地讲，数字是指属性值为 Numeric_Type=Digit 或 Numeric_Type=Decimal 的字符。

str.**isidentifier**()
如果字符串是有效的标识符，返回 True ，依据语言定义，identifiers 节。

调用*keyword.iskeyword()* 来检测字符串 s 是否为保留标识符，例如 def 和 class。

示例：

```
>>> from keyword import iskeyword

>>> 'hello'.isidentifier(), iskeyword('hello')
(True, False)
>>> 'def'.isidentifier(), iskeyword('def')
(True, True)
```

str.**islower**()
如果字符串中至少有一个区分大小写的字符[4] 且此类字符均为小写则返回 True ，否则返回 False 。

str.**isnumeric**()
如果字符串中至少有一个字符且所有字符均为数值字符则返回 True ，否则返回 False 。数值字符包括数字字符，以及所有在 Unicode 中设置了数值特性属性的字符，例如 U+2155, VULGAR FRACTION ONE FIFTH。正式的定义为：数值字符就是具有特征属性值 Numeric_Type=Digit, Numeric_Type=Decimal 或 Numeric_Type=Numeric 的字符。

str.**isprintable**()
如果字符串中所有字符均为可打印字符或字符串为空则返回 True ，否则返回 False 。不可打印字符是在 Unicode 字符数据库中被定义为”Other” 或”Separator” 的字符，例外情况是 ASCII 空格字符 (0x20) 被视作可打印字符。(请注意在此语境下可打印字符是指当对一个字符串发起调用*repr()* 时不必被转义的字符。它们与字符串写入*sys.stdout* 或*sys.stderr* 时所需的处理无关。)

str.**isspace**()
如果字符串中只有空白字符且至少有一个字符则返回 True ，否则返回 False 。

空白字符是指在 Unicode 字符数据库 (参见*unicodedata*) 中主要类别为 Zs (”Separator, space”) 或所属双向类为 WS, B 或 S 的字符。

str.**istitle**()
如果字符串中至少有一个字符且为标题字符串则返回 True ，例如大写字符之后只能带非大写字符而小写字符必须有大写字符打头。否则返回 False 。

str.**isupper**()
如果字符串中至少有一个区分大小写的字符[4] 且此类字符均为大写则返回 True ，否则返回 False 。

```
>>> 'BANANA'.isupper()
True
>>> 'banana'.isupper()
False
>>> 'baNana'.isupper()
False
>>> ' '.isupper()
False
```

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

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

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

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

所用转换小写算法的描述请参见 Unicode 标准的 3.13 节。

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

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

参见*str.removeprefix()*，该方法将删除单个前缀字符串，而不是全部给定集合中的字符。例如:

```
>>> 'Arthur: three!'.lstrip('Arthur: ')
'ee!'
>>> 'Arthur: three!'.removeprefix('Arthur: ')
'three!'
```

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

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

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

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

str.**removeprefix**(*prefix*, */*)
如果字符串以 *prefix* 字符串开头，返回 `string[len(prefix):]`。否则，返回原始字符串的副本:

```
>>> 'TestHook'.removeprefix('Test')
'Hook'
>>> 'BaseTestCase'.removeprefix('Test')
'BaseTestCase'
```

3.9 版新加入.

str.**removesuffix**(*suffix*, */*)
如果字符串以 *suffix* 字符串结尾，并且 *suffix* 非空，返回 `string[:-len(suffix)]`。否则，返回原始字符串的副本:

```
>>> 'MiscTests'.removesuffix('Tests')
'Misc'
>>> 'TmpDirMixin'.removesuffix('Tests')
'TmpDirMixin'
```

3.9 版新加入.

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

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

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

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

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

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

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

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

要删除单个后缀字符串，而不是全部给定集合中的字符，请参见*str.removesuffix()* 方法。例如:

```
>>> 'Monty Python'.rstrip(' Python')
'M'
>>> 'Monty Python'.removesuffix(' Python')
'Monty'
```

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

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

例如:

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

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

例如:

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

str.**splitlines**(*keepends=False*)

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

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

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

3.2 版更變: \v 和 \f 被添加到行边界列表

例如:

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

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

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

作为比较，split('\n') 的结果为:

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

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

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

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

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

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

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

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

例如:

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

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

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

The *string.capwords()* function does not have this problem, as it splits words on spaces only.

Alternatively, a workaround for apostrophes can be constructed using regular expressions:

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

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

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

另请参阅*codecs* 模块以了解定制字符映射的更灵活方式。

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

所用转换大写算法的描述请参见 Unicode 标准的 3.13 节。

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

例如:

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

4.7.2 printf 风格的字符串格式化

備🄵: 此处介绍的格式化操作具有多种怪异特性，可能导致许多常见错误（例如无法正确显示元组和字典）。使用较新的 格式化字符串字面值，*str.format()* 接口或模板字符串 有助于避免这样的错误。这些替代方案中的每一种都更好地权衡并提供了简单、灵活以及可扩展性优势。

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

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

转换标记符包含两个或更多字符并具有以下组成，且必须遵循此处规定的顺序：

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

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

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

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

在此情况下格式中不能出现 * 标记符（因其需要一个序列类的参数列表）。

转换旗标为：

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

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

转换类型为：

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

F解：

(1) 此替代形式会在第一个数码之前插入标示八进制数的前缀 ('0o')。

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

(3) 此替代形式总是会在结果中包含一个小数点，即使其后并没有数码。

小数点后的数码位数由精度决定，默认为 6。

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

小数点前后的有效数码位数由精度决定，默认为 6。

(5) 如果精度为 N，输出将截短为 N 个字符。

(6) 参见 **PEP 237**。

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

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

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

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

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

4.8.1 bytes 对象

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

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

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

- 单引号: `b'同样允许嵌入" 双" 引号'`。
- Double quotes: `b"still allows embedded 'single' quotes"`
- 三重引号: `b'''三重单引号'''`, `b""" 三重双引号"""`

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

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

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

除了字面值形式，bytes 对象还可以通过其他几种方式来创建：

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

另请参阅*bytes* 内置类型。

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

classmethod fromhex(*string*)

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

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

3.7 版更變: *bytes.fromhex()* 现在会忽略所有 ASCII 空白符而不只是空格符。

存在一个反向转换函数，可以将 bytes 对象转换为对应的十六进制表示。

hex([*sep*[, *bytes_per_sep*]])
返回一个字符串对象，该对象包含实例中每个字节的两个十六进制数字。

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

如果你希望令十六进制数字符串更易读，你可以指定单个字符分隔符作为 *sep* 形参包含于输出中。默认会放在每个字节之间。第二个可选的 *bytes_per_sep* 形参控制间距。正值会从右开始计算分隔符的位置，负值则是从左开始。

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

3.5 版新加入.

3.8 版更變: *bytes.hex()* 现在支持可选的 *sep* 和 *bytes_per_sep* 形参以在十六进制输出的字节之间插入分隔符。

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

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

4.8.2 bytearray 对象

bytearray 对象是*bytes* 对象的可变对应物。

class bytearray([*source*[, *encoding*[, *errors*]]])
bytearray 对象没有专属的字面值语法，它们总是通过调用构造器来创建：

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

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

另请参见*bytearray* 内置类型。

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

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

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

3.7 版更變: *bytearray.fromhex()* 现在会忽略所有 ASCII 空白符而不只是空格符。

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

hex([*sep*[, *bytes_per_sep*]])
返回一个字符串对象，该对象包含实例中每个字节的两个十六进制数字。

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

3.5 版新加入.

3.8 版更變: 与*bytes.hex()* 相似，*bytearray.hex()* 现在支持可选的 *sep* 和 *bytes_per_sep* 参数以在十六进制输出的字节之间插入分隔符。

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

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

4.8.3 bytes 和 bytearray 操作

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

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

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

和:

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

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

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

---

bytes 和 bytearray 对象的下列方法可以用于任意二进制数据。

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

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

3.3 版更變: 也接受 0 至 255 范围内的整数作为子序列。

`bytes.`**removeprefix**(*prefix*, */*)

bytearray.**removeprefix**(*prefix*, /)

如果二进制数据以 *prefix* 字符串开头，返回 `bytes[len(prefix):]`。否则，返回原始二进制数据的副本：

```
>>> b'TestHook'.removeprefix(b'Test')
b'Hook'
>>> b'BaseTestCase'.removeprefix(b'Test')
b'BaseTestCase'
```

prefix 可以是任意*bytes-like object*。

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

3.9 版新加入.

bytes.**removesuffix**(*suffix*, /)
bytearray.**removesuffix**(*suffix*, /)

如果二进制数据以 *suffix* 字符串结尾，并且 *suffix* 非空，返回 `bytes[:-len(suffix)]`。否则，返回原始二进制数据的副本：

```
>>> b'MiscTests'.removesuffix(b'Tests')
b'Misc'
>>> b'TmpDirMixin'.removesuffix(b'Tests')
b'TmpDirMixin'
```

suffix 可以是任意*bytes-like object*。

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

3.9 版新加入.

bytes.**decode**(*encoding="utf-8"*, *errors="strict"*)
bytearray.**decode**(*encoding="utf-8"*, *errors="strict"*)

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

errors 参数默认不会被检查，以获得最佳性能，而只在第一次解码错误时使用。启用*Python 开发模式*，或者使用调试构建来检查 *errors*。

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

3.1 版更變: 加入了对关键字参数的支持。

3.9 版更變: *errors* 现在在开发模式和调试模式下都会被检查。

bytes.**endswith**(*suffix*[, *start*[, *end*]])
bytearray.**endswith**(*suffix*[, *start*[, *end*]])

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

要搜索的后缀可以是任意*bytes-like object*。

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

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

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

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

3.3 版更變: 也接受 0 至 255 范围内的整数作为子序列。

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

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

3.3 版更變: 也接受 0 至 255 范围内的整数作为子序列。

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

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

3.1 版新加入.

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

要搜索的分隔符可以是任意*bytes-like object*。

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

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

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

---

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

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

3.3 版更變: 也接受 0 至 255 范围内的整数作为子序列。

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

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

3.3 版更變: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**rpartition**(*sep*)
bytearray.**rpartition**(*sep*)
在 *sep* 最后一次出现的位置拆分序列，返回一个 3 元组，其中包含分隔符之前的部分，分隔符本身或其 bytearray 副本，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含两个空的 bytes 或 bytearray 对象以及原序列的副本。

要搜索的分隔符可以是任意*bytes-like object*。

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

要搜索的前缀可以是任意*bytes-like object*。

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

你可以使用`bytes.maketrans()` 方法来创建转换表。

对于仅需移除字符的转换，请将 *table* 参数设为 `None`:

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

3.6 版更變: 现在支持将 *delete* 作为关键字参数。

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

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

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

---

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

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

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

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

要移除的二进制序列可以是任意*bytes-like object* 。要删除单个前缀字符串，而不是全部给定集合中的字符，请参见*str.removeprefix()* 方法。例如:

```
>>> b'Arthur: three!'.lstrip(b'Arthur: ')
b'ee!'
>>> b'Arthur: three!'.removeprefix(b'Arthur: ')
b'three!'
```

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

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

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

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

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

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

要移除的二进制序列可以是任意*bytes-like object* 。要删除单个后缀字符串，而不是全部给定集合中的字符，请参见*str.removesuffix()* 方法。例如:

```
>>> b'Monty Python'.rstrip(b' Python')
b'M'
```

(下页继续)

(繼續上一頁)

```
>>> b'Monty Python'.removesuffix(b' Python')
b'Monty'
```

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

bytes.**split**(*sep=None*, *maxsplit=-1*)
bytearray.**split**(*sep=None*, *maxsplit=-1*)

将二进制序列拆分为相同类型的子序列，使用 *sep* 作为分隔符。如果给出了 *maxsplit* 且非负值，则最多进行 *maxsplit* 次拆分（因此，列表最多会有 `maxsplit+1` 个元素）。如果 *maxsplit* 未指定或为 `-1`，则不限制拆分次数（进行所有可能的拆分）。

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

例如:

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

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

例如:

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

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

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

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

要移除的字节值二进制序列可以是任意*bytes-like object*。

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

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

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

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

---

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

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

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

---

bytes.**isalnum**()
bytearray.**isalnum**()
如果序列中所有字节都是字母类 ASCII 字符或 ASCII 十进制数码并且序列非空则返回 `True`，否则返回 `False`。字母类 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。ASCII 十进制数码就是字节值包含在序列 `b'0123456789'` 中的字符。

例如:

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

bytes.**isalpha**()
bytearray.**isalpha**()
如果序列中所有字节都是字母类 ASCII 字符并且序列不非空则返回 `True`，否则返回 `False`。字母类 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

例如:

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

bytes.**isascii**()

bytearray.**isascii**()
如果序列为空或序列中所有字节都是 ASCII 字节则返回 True，否则返回 False。ASCII 字节的取值范围是 0-0x7F。

3.7 版新加入.

bytes.**isdigit**()
bytearray.**isdigit**()
如果序列中所有字节都是 ASCII 十进制数码并且序列非空则返回 True，否则返回 False。ASCII 十进制数码就是字节值包含在序列 b'0123456789' 中的字符。

例如:

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

bytes.**islower**()
bytearray.**islower**()
如果序列中至少有一个小写的 ASCII 字符并且没有大写的 ASCII 字符则返回 True，否则返回 False。

例如:

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

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

bytes.**isspace**()
bytearray.**isspace**()
如果序列中所有字节都是 ASCII 空白符并且序列非空则返回 True，否则返回 False。ASCII 空白符就是字节值包含在序列 b' \t\n\r\x0b\f' (空格, 制表, 换行, 回车, 垂直制表, 进纸) 中的字符。

bytes.**istitle**()
bytearray.**istitle**()
如果序列为 ASCII 标题大小写形式并且序列非空则返回 True，否则返回 False。请参阅*bytes.title()* 了解有关“标题大小写”的详细定义。

例如:

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

bytes.**isupper**()
bytearray.**isupper**()
如果序列中至少有一个大写字母 ASCII 字符并且没有小写 ASCII 字符则返回 True，否则返回 False。

例如:

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

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

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

例如:

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

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

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

---

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

例如:

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

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

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

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

例如:

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

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

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

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

---

bytes.**title**()

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

例如:

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

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

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

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

可以使用正则表达式来构建针对撇号的特别处理:

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

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

---

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

例如:

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

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

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

---

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

例如:

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

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

4.8.4 printf 风格的字节串格式化

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

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

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

转换标记符包含两个或更多字符并具有以下组成，且必须遵循此处规定的顺序：

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

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

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

在此情况下格式中不能出现 `*` 标记符（因其需要一个序列类的参数列表）。

转换旗标为：

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

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

转换类型为：

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

𝔽解：

(1) 此替代形式会在第一个数码之前插入标示八进制数的前缀 ('0o')。

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

(3) 此替代形式总是会在结果中包含一个小数点，即使其后并没有数码。

小数点后的数码位数由精度决定，默认为 6。

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

小数点前后的有效数码位数由精度决定，默认为 6。

(5) 如果精度为 N，输出将截短为 N 个字符。

(6) b'%s' 已弃用，但在 3.x 系列中将不会被移除。

(7) b'%r' 已弃用，但在 3.x 系列中将不会被移除。

(8) 参见 **PEP 237**。

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

也参考：

PEP 461 - 为 bytes 和 bytearray 添加% 格式化

3.5 版新加入.

4.8.5 内存视图

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

class memoryview(*object*)

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

memoryview 有 **元素**的概念，**元素**指由原始 *object* 处理的原子内存单元。对于许多简单的类型，如*bytes* 和*bytearray* ，一个元素是一个字节，但其他类型，如*array.array* 可能有更大的元素。

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

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

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

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

这里是一个使用非字节格式的例子:

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

如果下层对象是可写的，则内存视图支持一维切片赋值。改变大小则不被允许:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
```

(下页继续)

(繼續上一頁)

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

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

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

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

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

3.5 版更變: 内存视图现在可使用整数元组进行索引。

memoryview 具有以下一些方法:

__eq__(*exporter*)

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

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

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

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

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
```

(下页继续)

(繼續上一頁)

```
>>> a == point
False
>>> a == b
False
```

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

3.3 版更變: 之前的版本比较原始内存时会忽略条目的格式与逻辑数组结构。

tobytes(*order=None*)

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

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

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

3.8 版新加入: *order* 可以为 {'C', 'F', 'A'}。当 *order* 为'C' 或'F' 时，原始数组的数据会被转换至 C 或 Fortran 顺序。对于连续视图，'A' 会返回物理内存的精确副本。特别地，内存中的 Fortran 顺序会被保留。对于非连续视图，数据会先被转换为 C 形式。*order=None* 与 *order='C'* 是相同的。

hex([*sep*[, *bytes_per_sep*]])

返回一个字符串对象，其中分别以两个十六进制数码表示缓冲区里的每个字节。

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

3.5 版新加入.

3.8 版更變: 与*bytes.hex()* 相似，*memoryview.hex()* 现在支持可选的 *sep* 和 *bytes_per_sep* 参数以在十六进制输出的字节之间插入分隔符。

tolist()

将缓冲区内的数据以一个元素列表的形式返回。

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

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

toreadonly()

返回 memoryview 对象的只读版本。原始的 memoryview 对象不会被改变。

```
>>> m = memoryview(bytearray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[89, 98, 99]
>>> mm[0] = 42
```

(下页继续)

(繼續上一頁)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

3.8 版新加入.

release()

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

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

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

使用 with 语句，可以通过上下文管理协议达到类似的效果:

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

3.2 版新加入.

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

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

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

将 1D/long 转换为 1D/unsigned bytes:

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

(下页继续)

(繼續上一頁)

```
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

将 1D/unsigned bytes 转换为 1D/char:

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

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

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

将 1D/unsigned long 转换为 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
```

(下页继续)

(繼續上一頁)

```
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

3.3 版新加入.

3.5 版更變: 当转换为字节视图时，源格式将不再受限。

还存在一些可用的只读属性:

obj

内存视图的下层对象:

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

3.3 版新加入.

nbytes

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

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

多维数组:

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

3.3 版新加入.

readonly

一个表明内存是否只读的布尔值。

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

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

itemsize
memoryview 中每个元素以字节表示的大小:

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

ndim
一个整数，表示内存所代表的多维数组具有多少个维度。

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

3.3 版更變: 当 ndim = 0 时值为空元组而不再为 None。

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

3.3 版更變: 当 ndim = 0 时值为空元组而不再为 None。

suboffsets
供 PIL 风格的数组内部使用。该值仅作为参考信息。

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

3.3 版新加入.

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

3.3 版新加入.

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

3.3 版新加入.

4.9 集合类型 --- set, frozenset

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

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

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

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

两个类的构造器具有相同的作用方式:

class set([*iterable*])
class frozenset([*iterable*])

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

集合可用多种方式来创建:

- 使用花括号内以逗号分隔元素的方式: `{'jack', 'sjoerd'}`
- 使用集合推导式: `{c for c in 'abracadabra' if c not in 'abc'}`
- 使用类型构造器: `set()`, `set('foobar')`, `set(['a', 'b', 'foo'])`

`set` 和`frozenset` 的实例提供以下操作:

len(s)
返回集合 *s* 中的元素数量（即 *s* 的基数)。

x in s
检测 *x* 是否为 *s* 中的成员。

x not in s
检测 *x* 是否非 *s* 中的成员。

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

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

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

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

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

union(**others*)

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

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

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

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

copy()
返回原集合的浅拷贝。

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

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

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

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

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

集合的元素，与字典的键类似，必须为*hashable*。

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

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

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

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

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

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

add(*elem*)
将元素 *elem* 添加到集合中。

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

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

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

clear()
从集合中移除所有元素。

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

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

4.10 映射类型 --- dict

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

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

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

字典可用多种方式来创建:

- 使用花括号内以逗号分隔 键: 值对的方式: `{'jack': 4098, 'sjoerd': 4127}` or `{4098: 'jack', 4127: 'sjoerd'}`
- 使用字典推导式: `{}`, `{x: x ** 2 for x in range(10)}`
- 使用类型构造器: `dict()`, `dict([('foo', 100), ('bar', 200)])`, `dict(foo=100, bar=200)`

如果没有给出位置参数，将创建一个空字典。如果给出一个位置参数并且其属于映射对象，将创建一个具有与映射对象相同键值对的字典。否则的话，位置参数必须为一个*iterable* 对象。该可迭代对象中的每一项本身必须为一个刚好包含两个元素的可迭代对象。每一项中的第一个对象将成为新字典的一个键，第二个对象将成为其对应的值。如果一个键出现一次以上，该键的最后一个值将成为其在新字典中对应的值。

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

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

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

(下页继续)

(繼續上一頁)

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

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

这些是字典所支持的操作（因而自定义的映射类型也应当支持）：

list(d)
返回字典 *d* 中使用的所有键的列表。

len(d)
返回字典 *d* 中的项数。

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

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

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

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

d[key] = value
将 `d[key]` 设为 *value*。

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

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

key not in d
等价于 `not key in d`。

iter(d)
返回以字典的键为元素的迭代器。这是 `iter(d.keys())` 的快捷方式。

clear()
移除字典中的所有元素。

copy()
返回原字典的浅拷贝。

classmethod fromkeys(*iterable*[, *value*])
使用来自 *iterable* 的键创建一个新字典，并将键值设为 *value*。

fromkeys() 是一个返回新字典的类方法。*value* 默认为 None。所有值都只引用一个单独的实例，因此让 *value* 成为一个可变对象例如空列表通常是没有意义的。要获取不同的值，请改用 字典推导式。

get(*key*[, *default*])
如果 *key* 存在于字典中则返回 *key* 的值，否则返回 *default*。如果 *default* 未给出则默认为 None，因而此方法绝不会引发*KeyError*。

items()
返回由字典项 ((键，值) 对) 组成的一个新视图。参见视图对象文档。

keys()
返回由字典键组成的一个新视图。参见视图对象文档。

pop(*key*[, *default*])
如果 *key* 存在于字典中则将其移除并返回其值，否则返回 *default*。如果 *default* 未给出且 *key* 不存在于字典中，则会引发*KeyError*。

popitem()
从字典中移除并返回一个 (键，值) 对。键值对会按 LIFO (后进先出) 的顺序被返回。

popitem() 适用于对字典进行消耗性的迭代，这在集合算法中经常被使用。如果字典为空，调用*popitem()* 将引发*KeyError*。

3.7 版更變: 现在会确保采用 LIFO 顺序。在之前的版本中，*popitem()* 会返回一个任意的键/值对。

reversed(d)
返回一个逆序获取字典键的迭代器。这是 reversed(d.keys()) 的快捷方式。

3.8 版新加入.

setdefault(*key*[, *default*])
如果字典存在键 *key*，返回它的值。如果不存在，插入值为 *default* 的键 *key*，并返回 *default*。*default* 默认为 None。

update([*other*])
使用来自 *other* 的键/值对更新字典，覆盖原有的键。返回 None。

update() 接受另一个字典对象，或者一个包含键/值对（以长度为二的元组或其他可迭代对象表示）的可迭代对象。如果给出了关键字参数，则会以其所指定的键/值对更新字典: d.update(red=1, blue=2)。

values()
返回由字典值组成的一个新视图。参见视图对象文档。

两个 dict.values() 视图之间的相等性比较将总是返回 False。这在 dict.values() 与其自身比较时也同样适用:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

d | other
合并 *d* 和 *other* 中的键和值来创建一个新的字典，两者必须都是字典。当 *d* 和 *other* 有相同键时，*other* 的值优先。

3.9 版新加入.

d |= other
用 *other* 的键和值更新字典 *d*，*other* 可以是*mapping* 或*iterable* 的键值对。当 *d* 和 *other* 有相同键时，*other* 的值优先。

3.9 版新加入.

两个字典的比较当且仅当它们具有相同的（键，值）对时才会相等（不考虑顺序）。排序比较 ('<', '<=', '>=', '>') 会引发*TypeError*。

字典会保留插入时的顺序。请注意对键的更新不会影响顺序。删除并再次添加的键将被插入到末尾。

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

3.7 版更變: 字典顺序会确保为插入顺序。此行为是自 3.6 版开始的 CPython 实现细节。

字典和字典视图都是可逆的。

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

3.8 版更變: 字典现在是可逆的。

也参考:

types.MappingProxyType 可被用来创建一个*dict* 的只读视图。

4.10.1 字典视图对象

由*dict.keys()*, *dict.values()* 和*dict.items()* 所返回的对象是 视图对象。该对象提供字典条目的一个动态视图，这意味着当字典改变时，视图也会相应改变。

字典视图可以被迭代以产生与其对应的数据，并支持成员检测：

len(dictview)
返回字典中的条目数。

iter(dictview)
返回字典中的键、值或项（以（键，值）为元素的元组表示）的迭代器。

键和值是按插入时的顺序进行迭代的。这样就允许使用*zip()* 来创建（值，键）对: `pairs = zip(d.values(), d.keys())`。另一个创建相同列表的方式是`pairs = [(v, k) for (k, v) in d.items()]`.

在添加或删除字典中的条目期间对视图进行迭代可能引发*RuntimeError* 或者无法完全迭代所有条目。

3.7 版更變: 字典顺序会确保为插入顺序。

x in dictview
如果 *x* 是对应字典中存在的键、值或项（在最后一种情况下 *x* 应为一个（键，值）元组）则返回 `True`。

reversed(dictview)
返回一个逆序获取字典键、值或项的迭代器。视图将按与插入时相反的顺序进行迭代。

3.8 版更變: 字典视图现在是可逆的。

键视图类似于集合，因为其条目不重复且可哈希。如果所有值都是可哈希的，即（键，值）对也是不重复且可哈希的，那么条目视图也会类似于集合。（值视图则不被视为类似于集合，因其条目通常都是有重复的。）对于类似于集合的视图，为抽象基类*collections.abc.Set* 所定义的全部操作都是有效的 (例如 ==, < 或 ^)。

一个使用字典视图的示例:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion order)
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

4.11 上下文管理器类型

Python 的 `with` 语句支持通过上下文管理器所定义的运行时上下文这一概念。此对象的实现使用了一对专门方法，允许用户自定义类来定义运行时上下文，在语句体被执行前进入该上下文，并在语句执行完毕时退出该上下文：

`contextmanager.__enter__()`
进入运行时上下文并返回此对象或关联到该运行时上下文的其他对象。此方法的返回值会绑定到使用此上下文管理器的 `with` 语句的 `as` 子句中的标识符。

一个返回其自身的上下文管理器的例子是*file object*。文件对象会从 __enter__() 返回其自身，以允许*open()* 被用作 `with` 语句中的上下文表达式。

一个返回关联对象的上下文管理器的例子是*decimal.localcontext()* 所返回的对象。此种管理器会将活动的 decimal 上下文设为原始 decimal 上下文的一个副本并返回该副本。这允许对 `with` 语句的语句体中的当前 decimal 上下文进行更改，而不会影响 `with` 语句以外的代码。

`contextmanager.__exit__`(*exc_type*, *exc_val*, *exc_tb*)
退出运行时上下文并返回一个布尔值旗标来表明所发生的任何异常是否应当被屏蔽。如果在执行 `with` 语句的语句体期间发生了异常，则参数会包含异常的类型、值以及回溯信息。在其他情况下三个参数均为 `None`。

自此方法返回一个真值将导致 `with` 语句屏蔽异常并继续执行紧随在 `with` 语句之后的语句。否则异常将在此方法结束执行后继续传播。在此方法执行期间发生的异常将会取代 `with` 语句的语句体中发生的任何异常。

传入的异常绝对不应当被显式地重新引发——相反地，此方法应当返回一个假值以表明方法已成功完成并且不希望屏蔽被引发的异常。这允许上下文管理代码方便地检测*__exit__()* 方法是否确实已失败。

Python 定义了一些上下文管理器来支持简易的线程同步、文件或其他对象的快速关闭，以及更方便地操作活动的十进制算术上下文。除了实现上下文管理协议以外，不同类型不会被特殊处理。请参阅*contextlib* 模块查看相关的示例。

Python's *generator*s and the *contextlib.contextmanager* decorator provide a convenient way to implement these protocols. If a generator function is decorated with the *contextlib.contextmanager* decorator, it will return a context manager implementing the necessary *__enter__()* and *__exit__()* methods, rather than the iterator produced by an undecorated generator function.

请注意，Python/C API 中 Python 对象的类型结构中并没有针对这些方法的专门槽位。想要定义这些方法的扩展类型必须将它们作为普通的 Python 可访问方法来提供。与设置运行时上下文的开销相比，单个类字典查找的开销可以忽略不计。

4.12 GenericAlias 类型

`GenericAlias` objects are generally created by subscripting a class. They are most often used with container classes, such as *list* or *dict*. For example, `list[int]` is a `GenericAlias` object created by subscripting the `list` class with the argument *int*. `GenericAlias` objects are intended primarily for use with *type annotations*.

備 F: It is generally only possible to subscript a class if the class implements the special method `__class_getitem__()`.

A `GenericAlias` object acts as a proxy for a *generic type*, implementing *parameterized generics*.

For a container class, the argument(s) supplied to a subscription of the class may indicate the type(s) of the elements an object contains. For example, `set[bytes]` can be used in type annotations to signify a *set* in which all the elements are of type *bytes*.

For a class which defines `__class_getitem__()` but is not a container, the argument(s) supplied to a subscription of the class will often indicate the return type(s) of one or more methods defined on an object. For example, *regular expressions* can be used on both the *str* data type and the *bytes* data type:

- If `x = re.search('foo', 'foo')`, `x` will be a *re.Match* object where the return values of `x.group(0)` and `x[0]` will both be of type *str*. We can represent this kind of object in type annotations with the `GenericAlias re.Match[str]`.
- If `y = re.search(b'bar', b'bar')`, (note the `b` for *bytes*), `y` will also be an instance of `re.Match`, but the return values of `y.group(0)` and `y[0]` will both be of type *bytes*. In type annotations, we would represent this variety of *re.Match* objects with `re.Match[bytes]`.

`GenericAlias` objects are instances of the class *types.GenericAlias*, which can also be used to create `GenericAlias` objects directly.

`T[X, Y, ...]`
Creates a `GenericAlias` representing a type `T` parameterized by types *X*, *Y*, and more depending on the `T` used. For example, a function expecting a *list* containing *float* elements:

```
def average(values: list[float]) -> float:
    return sum(values) / len(values)
```

另一个例子是关于*mapping* 对象的，用到了*dict*，泛型的两个类型参数分别代表了键类型和值类型。本例中的函数需要一个 `dict`，其键的类型为*str*，值的类型为*int*:。

```
def send_post_request(url: str, body: dict[str, int]) -> None:
    ...
```

内置函数*isinstance()* 和*issubclass()* 不接受第二个参数为 “GenericAlias“ 类型：

```
>>> isinstance([1, 2], list[str])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: isinstance() argument 2 cannot be a parameterized generic
```

The Python runtime does not enforce *type annotations*. This extends to generic types and their type parameters. When creating a container object from a `GenericAlias`, the elements in the container are not checked against their type. For example, the following code is discouraged, but will run without errors:

```
>>> t = list[str]
>>> t([1, 2, 3])
[1, 2, 3]
```

不仅如此，在创建对象的过程中，参数化的泛型还会抹除类型参数：

```
>>> t = list[str]
>>> type(t)
<class 'types.GenericAlias'>

>>> l = t()
>>> type(l)
<class 'list'>
```

在泛型上调用*repr()* 或*str()* 会显示应用参数之后的类型：

```
>>> repr(list[int])
'list[int]'

>>> str(list[int])
'list[int]'
```

The `__getitem__()` method of generic containers will raise an exception to disallow mistakes like `dict[str][str]`:

```
>>> dict[str][str]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: There are no type variables left in dict[str]
```

However, such expressions are valid when *type variables* are used. The index must have as many elements as there are type variable items in the `GenericAlias` object's *`__args__`*.

```
>>> from typing import TypeVar
>>> Y = TypeVar('Y')
>>> dict[str, Y][int]
dict[str, int]
```

4.12.1 Standard Generic Classes

The following standard library classes support parameterized generics. This list is non-exhaustive.

- *`tuple`*
- *`list`*
- *`dict`*
- *`set`*
- *`frozenset`*
- *`type`*
- *`collections.deque`*
- *`collections.defaultdict`*
- *`collections.OrderedDict`*
- *`collections.Counter`*
- *`collections.ChainMap`*
- *`collections.abc.Awaitable`*
- *`collections.abc.Coroutine`*
- *`collections.abc.AsyncIterable`*
- *`collections.abc.AsyncIterable`*
- *`collections.abc.AsyncGenerator`*
- *`collections.abc.Iterable`*
- *`collections.abc.Iterator`*
- *`collections.abc.Generator`*

- *collections.abc.Reversible*
- *collections.abc.Container*
- *collections.abc.Collection*
- *collections.abc.Callable*
- *collections.abc.Set*
- *collections.abc.MutableSet*
- *collections.abc.Mapping*
- *collections.abc.MutableMapping*
- *collections.abc.Sequence*
- *collections.abc.MutableSequence*
- *collections.abc.ByteString*
- *collections.abc.MappingView*
- *collections.abc.KeysView*
- *collections.abc.ItemsView*
- *collections.abc.ValuesView*
- *contextlib.AbstractContextManager*
- *contextlib.AbstractAsyncContextManager*
- *dataclasses.Field*
- *functools.cached_property*
- *functools.partialmethod*
- *os.PathLike*
- *queue.LifoQueue*
- *queue.Queue*
- *queue.PriorityQueue*
- *queue.SimpleQueue*
- *re.Pattern*
- *re.Match*
- *shelve.BsdDbShelf*
- *shelve.DbfilenameShelf*
- *shelve.Shelf*
- *types.MappingProxyType*
- *weakref.WeakKeyDictionary*
- *weakref.WeakMethod*
- *weakref.WeakSet*
- *weakref.WeakValueDictionary*

4.12.2 Special Attributes of GenericAlias objects

应用参数后的泛型都实现了一些特殊的只读属性：

genericalias.**__origin__**
本属性指向未应用参数之前的泛型类：

```
>>> list[int].__origin__
<class 'list'>
```

genericalias.**__args__**
This attribute is a *tuple* (possibly of length 1) of generic types passed to the original `__class_getitem__()` of the generic class:

```
>>> dict[str, list[int]].__args__
(<class 'str'>, list[int])
```

genericalias.**__parameters__**
该属性是延迟计算出来的一个元组（可能为空），包含了 `__args__` 中的唯一类型变量。

```
>>> from typing import TypeVar

>>> T = TypeVar('T')
>>> list[T].__parameters__
(~T,)
```

也参考：

PEP 484 - Type Hints Introducing Python's framework for type annotations.

PEP 585 - Type Hinting Generics In Standard Collections Introducing the ability to natively parameterize standard-library classes, provided they implement the special class method `__class_getitem__()`.

泛型（Generic）, *user-defined generics* **and** *typing.Generic* Documentation on how to implement generic classes that can be parameterized at runtime and understood by static type-checkers.

3.9 版新加入.

4.13 其他内置类型

解释器支持一些其他种类的对象。这些对象大都仅支持一两种操作。

4.13.1 模块

模块唯一的特殊操作是属性访问: `m.name`，这里 *m* 为一个模块而 *name* 访问定义在 *m* 的符号表中的一个名称。模块属性可以被赋值。（请注意 `import` 语句严格来说也是对模块对象的一种操作；`import foo` 不要求存在一个名为 *foo* 的模块对象，而是要求存在一个对于名为 *foo* 的模块的 (永久性) 定义。）

每个模块都有一个特殊属性 *__dict__*。这是包含模块的符号表的字典。修改此字典将实际改变模块的符号表，但是无法直接对 *__dict__* 赋值 (你可以写 `m.__dict__['a'] = 1`，这会将 `m.a` 定义为 `1`，但是你不能写 `m.__dict__ = {}`)。不建议直接修改 *__dict__*。

内置于解释器中的模块会写成这样: `<module 'sys' (built-in)>`。如果是从一个文件加载，则会写成 `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`。

4.13.2 类与类实例

关于这些类型请参阅 objects 和 class。

4.13.3 函数

函数对象是通过函数定义创建的。对函数对象的唯一操作是调用它: `func(argument-list)`。

实际上存在两种不同的函数对象：内置函数和用户自定义函数。两者支持同样的操作（调用函数），但实现方式不同，因此对象类型也不同。

更多信息请参阅 function。

4.13.4 方法

方法是使用属性表示法来调用的函数。存在两种形式：内置方法（例如列表的 `append()` 方法）和类实例方法。内置方法由支持它们的类型来描述。

如果你通过一个实例来访问方法（即定义在类命名空间内的函数），你会得到一个特殊对象: *绑定方法* (或称 *实例方法*) 对象。当被调用时，它会将 `self` 参数添加到参数列表。绑定方法具有两个特殊的只读属性: `m.__self__` 操作该方法的对象，而 `m.__func__` 是实现该方法的函数。调用 `m(arg-1, arg-2, ..., arg-n)` 完全等价于调用 `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`。

与函数对象类似，绑定方法对象也支持获取任意属性。但是，由于方法属性实际上保存于下层的函数对象中 (`meth.__func__`)，因此不允许设置绑定方法的方法属性。尝试设置方法的属性将会导致引发*AttributeError*。想要设置方法属性，你必须在下层的函数对象中显式地对其进行设置:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
>>> c.method.whoami
'my name is method'
```

更多信息请参阅 types。

4.13.5 代码对象

代码对象被具体实现用来表示“伪编译”的可执行 Python 代码，例如一个函数体。它们不同于函数对象，因为它们不包含对其全局执行环境的引用。代码对象由内置的*compile()* 函数返回，并可通过从函数对象的 `__code__` 属性从中提取。另请参阅*code* 模块。

访问 `__code__` 会触发审计事件 `object.__getattr__`，参数为 `obj` 和 `"__code__"`。

可以通过将代码对象（而非源码字符串）传给*exec()* 或*eval()* 内置函数来执行或求值。

更多信息请参阅 types。

4.13.6 类型对象

类型对象表示各种对象类型。对象的类型可通过内置函数*type()* 来获取。类型没有特殊的操作。标准库模块*types* 定义了所有标准内置类型的名称。

类型以这样的写法来表示: `<class 'int'>`。

4.13.7 空对象

此对象会由不显式地返回值的函数所返回。它不支持任何特殊的操作。空对象只有一种值 `None` (这是个内置名称)。`type(None)()` 会生成同一个单例。

该对象的写法为 `None`。

4.13.8 省略符对象

此对象常被用于切片 (参见 slicings)。它不支持任何特殊的操作。省略符对象只有一种值*Ellipsis* (这是个内置名称)。`type(Ellipsis)()` 会生成*Ellipsis* 单例。

该对象的写法为 `Ellipsis` 或 `...`。

4.13.9 未实现对象

此对象会被作为比较和二元运算被应用于它们所不支持的类型时的返回值。请参阅 comparisons 了解更多信息。未实现对象只有一种值 `NotImplemented`。`type(NotImplemented)()` 会生成这个单例。

该对象的写法为 `NotImplemented`。

4.13.10 布尔值

布尔值是两个常量对象 `False` 和 `True`。它们被用来表示逻辑上的真假（不过其他值也可被当作真值或假值）。在数字类的上下文中（例如被用作算术运算符的参数时），它们的行为分别类似于整数 0 和 1。内置函数*bool()* 可被用来将任意值转换为布尔值，只要该值可被解析为一个逻辑值（参见之前的逻辑值检测 部分）。

该对象的写法分别为 `False` 和 `True`。

4.13.11 内部对象

有关此对象的信息请参阅 types。其中描述了栈帧对象、回溯对象以及切片对象等等。

4.14 特殊属性

语言实现为部分对象类型添加了一些特殊的只读属性，它们具有各自的作用。其中一些并不会被*dir()* 内置函数所列出。

object.**__dict__**
一个字典或其他类型的映射对象，用于存储对象的（可写）属性。

instance.**__class__**
类实例所属的类。

class.**__bases__**
由类对象的基类所组成的元组。

definition.**__name__**
类、函数、方法、描述器或生成器实例的名称。

definition.**__qualname__**
类、函数、方法、描述器或生成器实例的*qualified name*。

3.3 版新加入.

class.**__mro__**
此属性是由类组成的元组，在方法解析期间会基于它来查找基类。

class.**mro**()
此方法可被一个元类来重载，以为其实例定制方法解析顺序。它会在类实例化时被调用，其结果存储于*__mro__* 之中。

class.**__subclasses__**()
每个类都存有对直接子类的弱引用列表。本方法返回所有存活引用的列表。列表的顺序按照子类定义的排列。例如：

```
>>> int.__subclasses__()
[<class 'bool'>]
```

4.15 Integer string conversion length limitation

CPython has a global limit for converting between *int* and *str* to mitigate denial of service attacks. This limit *only* applies to decimal or other non-power-of-two number bases. Hexadecimal, octal, and binary conversions are unlimited. The limit can be configured.

The *int* type in CPython is an abitrary length number stored in binary form (commonly known as a ”bignum”). There exists no algorithm that can convert a string to a binary integer or a binary integer to a string in linear time, *unless* the base is a power of 2. Even the best known algorithms for base 10 have sub-quadratic complexity. Converting a large value such as `int('1' * 500_000)` can take over a second on a fast CPU.

Limiting conversion size offers a practical way to avoid CVE-2020-10735.

The limit is applied to the number of digit characters in the input or output string when a non-linear conversion algorithm would be involved. Underscores and the sign are not counted towards the limit.

When an operation would exceed the limit, a *ValueError* is raised:

```
>>> import sys
>>> sys.set_int_max_str_digits(4300)  # Illustrative, this is the default.
>>> _ = int('2' * 5432)
```

(下页继续)

(繼續上一頁)

```
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300) for integer string conversion: value has 5432␣
↪digits.
>>> i = int('2' * 4300)
>>> len(str(i))
4300
>>> i_squared = i*i
>>> len(str(i_squared))
Traceback (most recent call last):
...
ValueError: Exceeds the limit (4300) for integer string conversion: value has 8599␣
↪digits.
>>> len(hex(i_squared))
7144
>>> assert int(hex(i_squared), base=16) == i*i  # Hexadecimal is unlimited.
```

The default limit is 4300 digits as provided in *sys.int_info.default_max_str_digits*. The lowest limit that can be configured is 640 digits as provided in *sys.int_info.str_digits_check_threshold*.

Verification:

```
>>> import sys
>>> assert sys.int_info.default_max_str_digits == 4300, sys.int_info
>>> assert sys.int_info.str_digits_check_threshold == 640, sys.int_info
>>> msg = int('578966293710682886880994035146873798396722250538762761564'
...           '9252925514383915483333812743580549779436104706260696366600'
...           '571186405732').to_bytes(53, 'big')
...
```

3.9.14 版新加入.

4.15.1 Affected APIs

The limitation only applies to potentially slow conversions between *int* and *str* or *bytes*:

- `int(string)` with default base 10.
- `int(string, base)` for all bases that are not a power of 2.
- `str(integer)`.
- `repr(integer)`
- any other string conversion to base 10, for example `f"{integer}"`, `"{}".format(integer)`, or `b"%d" % integer`.

The limitations do not apply to functions with a linear algorithm:

- `int(string, base)` with base 2, 4, 8, 16, or 32.
- *int.from_bytes()* and *int.to_bytes()*.
- *hex()*, *oct()*, *bin()*.
- 格式规格迷你语言 for hex, octal, and binary numbers.
- *str* to *float*.
- *str* to *decimal.Decimal*.

4.15.2 Configuring the limit

Before Python starts up you can use an environment variable or an interpreter command line flag to configure the limit:

- `PYTHONINTMAXSTRDIGITS`, e.g. `PYTHONINTMAXSTRDIGITS=640 python3` to set the limit to 640 or `PYTHONINTMAXSTRDIGITS=0 python3` to disable the limitation.
- `-X int_max_str_digits`, e.g. `python3 -X int_max_str_digits=640`
- `sys.flags.int_max_str_digits` contains the value of `PYTHONINTMAXSTRDIGITS` or `-X int_max_str_digits`. If both the env var and the `-X` option are set, the `-X` option takes precedence. A value of *-1* indicates that both were unset, thus a value of `sys.int_info.default_max_str_digits` was used during initilization.

From code, you can inspect the current limit and set a new one using these *sys* APIs:

- *sys.get_int_max_str_digits()* and *sys.set_int_max_str_digits()* are a getter and setter for the interpreter-wide limit. Subinterpreters have their own limit.

Information about the default and minimum can be found in *sys.int_info*:

- *sys.int_info.default_max_str_digits* is the compiled-in default limit.
- *sys.int_info.str_digits_check_threshold* is the lowest accepted value for the limit (other than 0 which disables it).

3.9.14 版新加入.

> **警示：** Setting a low limit *can* lead to problems. While rare, code exists that contains integer constants in decimal in their source that exceed the minimum threshold. A consequence of setting the limit is that Python source code containing decimal integer literals longer than the limit will encounter an error during parsing, usually at startup time or import time or even at installation time - anytime an up to date `.pyc` does not already exist for the code. A workaround for source that contains such large constants is to convert them to `0x` hexadecimal form as it has no limit.
>
> Test your application thoroughly if you use a low limit. Ensure your tests run with the limit set early via the environment or flag so that it applies during startup and even during any installation step that may invoke Python to precompile `.py` sources to `.pyc` files.

4.15.3 Recommended configuration

The default `sys.int_info.default_max_str_digits` is expected to be reasonable for most applications. If your application requires a different limit, set it from your main entry point using Python version agnostic code as these APIs were added in security patch releases in versions before 3.11.

Example:

```
>>> import sys
>>> if hasattr(sys, "set_int_max_str_digits"):
...     upper_bound = 68000
...     lower_bound = 4004
...     current_limit = sys.get_int_max_str_digits()
...     if current_limit == 0 or current_limit > upper_bound:
...         sys.set_int_max_str_digits(upper_bound)
...     elif current_limit < lower_bound:
...         sys.set_int_max_str_digits(lower_bound)
```

If you need to disable it entirely, set it to `0`.

F解

CHAPTER 5

🄵建的例外

在 Python 中，所有异常必须为一个派生自*BaseException* 的类的实例。在带有提及一个特定类的 except 子句的 try 语句中，该子句也会处理任何派生自该类的异常类（但不处理 它所派生出的异常类）。通过子类化创建的两个不相关异常类永远是不等效的，既使它们具有相同的名称。

下面列出的内置异常可通过解释器或内置函数来生成。除非另有说明，它们都会具有一个提示导致错误详细原因的“关联值”。这可以是一个字符串或由多个信息项（例如一个错误码和一个解释错误的字符串）组成的元组。关联值通常会作为参数被传递给异常类的构造器。

用户代码可以引发内置异常。这可被用于测试异常处理程序或报告错误条件，“就像”在解释器引发了相同异常的情况时一样；但是请注意，没有任何机制能防止用户代码引发不适当的错误。

内置异常类可以被子类化以定义新的异常；鼓励程序员从*Exception* 类或它的某个子类而不是从*BaseException* 来派生新的异常。关于定义异常的更多信息可以在 Python 教程的 tut-userexceptions 部分查看。

5.1 Exception context

When raising a new exception while another exception is already being handled, the new exception's __context__ attribute is automatically set to the handled exception. An exception may be handled when an except or finally clause, or a with statement, is used.

This implicit exception context can be supplemented with an explicit cause by using from with raise:

```
raise new_exc from original_exc
```

跟在 from 之后一表达式必须为一个异常或 None。它将在所引发的异常上被设置为 __cause__。设置 __cause__ 还会隐式地将 __suppress_context__ 属性设为 True，这样使用 raise new_exc from None 可以有效地将旧异常替换为新异常来显示其目的 (例如将*KeyError* 转换为*AttributeError*)，同时让旧异常在 __context__ 中保持可用状态以便在调试时进行内省。

除了异常本身的回溯以外，默认的回溯还会显示这些串连的异常。__cause__ 中的显式串连异常如果存在将总是显示。__context__ 中的隐式串连异常仅在 __cause__ 为*None* 并且 __suppress_context__ 为假值时显示。

不论在哪种情况下，异常本身总会在任何串连异常之后显示，以便回溯的最后一行总是显示所引发的最后一个异常。

5.2 繼承自𝔽建的例外

User code can create subclasses that inherit from an exception type. It's recommended to only subclass one exception type at a time to avoid any possible conflicts between how the bases handle the `args` attribute, as well as due to possible memory layout incompatibilities.

CPython implementation detail: Most built-in exceptions are implemented in C for efficiency, see: Objects/exceptions.c. Some have custom memory layouts which makes it impossible to create a subclass that inherits from multiple exception types. The memory layout of a type is an implementation detail and might change between Python versions, leading to new conflicts in the future. Therefore, it's recommended to avoid subclassing multiple exception types altogether.

5.3 基类

下列异常主要被用作其他异常的基类。

exception BaseException
所有内置异常的基类。它不应该被用户自定义类直接继承 (这种情况请使用*Exception*)。如果在此类的实例上调用*str()*，则会返回实例的参数表示，或者当没有参数时返回空字符串。

args
传给异常构造器的参数元组。某些内置异常 (例如*OSError*) 接受特定数量的参数并赋予此元组中的元素特殊的含义，而其他异常通常只接受一个给出错误信息的单独字符串。

with_traceback(*tb*)
此方法将 *tb* 设为异常的新回溯信息并返回该异常对象。它通常以如下的形式在异常处理程序中使用:

```
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

exception Exception
所有内置的非系统退出类异常都派生自此类。所有用户自定义异常也应当派生自此类。

exception ArithmeticError
此基类用于派生针对各种算术类错误而引发的内置异常: *OverflowError*, *ZeroDivisionError*, *FloatingPointError*。

exception BufferError
当与 缓冲区相关的操作无法执行时将被引发。

exception LookupError
此基类用于派生当映射或序列所使用的键或索引无效时引发的异常: *IndexError*, *KeyError*。这可以通过*codecs.lookup()* 来直接引发。

5.4 具体异常

以下异常属于经常被引发的异常。

exception AssertionError
当 assert 语句失败时将被引发。

exception AttributeError
当属性引用 (参见 attribute-references) 或赋值失败时将被引发。(当一个对象根本不支持属性引用或属性赋值时则将引发 *TypeError*。)

exception EOFError
当 *input()* 函数未读取任何数据即达到文件结束条件 (EOF) 时将被引发。(另外，io.IOBase.read() 和 *io.IOBase.readline()* 方法在遇到 EOF 则将返回一个空字符串。)

exception FloatingPointError
目前未被使用。

exception GeneratorExit
当一个 *generator* 或 *coroutine* 被关闭时将被引发；参见 generator.close() 和 coroutine.close()。它直接继承自 *BaseException* 而不是 *Exception*，因为从技术上来说它并不是一个错误。

exception ImportError
当 import 语句尝试加载模块遇到麻烦时将被引发。并且当 from ... import 中的"from list" 存在无法找到的名称时也会被引发。

name 与 path 属性可通过对构造器使用仅关键字参数来设定。设定后它们将分别表示被尝试导入的模块名称与触发异常的任意文件所在路径。

3.3 版更變: 添加了 name 与 path 属性。

exception ModuleNotFoundError
ImportError 的子类，当一个模块无法被定位时将由 import 引发。当在 *sys.modules* 中找到 None 时也会被引发。

3.6 版新加入.

exception IndexError
当序列抽取超出范围时将被引发。(切片索引会被静默截短到允许的范围；如果指定索引不是整数则 *TypeError* 会被引发。)

exception KeyError
当在现有键集合中找不到指定的映射（字典）键时将被引发。

exception KeyboardInterrupt
当用户按下中断键 (通常为 Control-C 或 Delete) 时将被引发。在执行期间，会定期检测中断信号。该异常继承自 *BaseException* 以确保不会被处理 *Exception* 的代码意外捕获，这样可以避免退出解释器。

備🄵: Catching a *KeyboardInterrupt* requires special consideration. Because it can be raised at unpredictable points, it may, in some circumstances, leave the running program in an inconsistent state. It is generally best to allow *KeyboardInterrupt* to end the program as quickly as possible or avoid raising it entirely. (See *Note on Signal Handlers and Exceptions*.)

---

exception MemoryError
当一个操作耗尽内存但情况仍可（通过删除一些对象）进行挽救时将被引发。关联的值是一个字符串，指明是哪种（内部）操作耗尽了内存。请注意由于底层的内存管理架构（C 的 malloc() 函数），解释器也许并不总是能够从这种情况下完全恢复；但它毕竟可以引发一个异常，这样就能打印出栈回溯信息，以便找出导致问题的失控程序。

exception NameError
当某个局部或全局名称未找到时将被引发。此异常仅用于非限定名称。关联的值是一条错误信息，其中包含未找到的名称。

exception NotImplementedError
此异常派生自*RuntimeError*。在用户自定义的基类中，抽象方法应当在其要求所派生类重载该方法，或是在其要求所开发的类提示具体实现尚待添加时引发此异常。

備𝔽: 它不应当用来表示一个运算符或方法根本不能被支持 -- 在此情况下应当让特定运算符 / 方法保持未定义，或者在子类中将其设为*None*。

備𝔽: `NotImplementedError` 和 `NotImplemented` 不可互换，即使它们有相似的名称和用途。请参阅*NotImplemented* 了解有关何时使用它们的详细说明。

---

exception OSError([*arg*])
exception OSError(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])
此异常在一个系统函数返回系统相关的错误时将被引发，此类错误包括 I/O 操作失败例如" 文件未找到" 或" 磁盘已满" 等（不包括非法参数类型或其他偶然性错误）。

构造器的第二种形式可设置如下所述的相应属性。如果未指定这些属性则默认为*None*。为了能向下兼容，如果传入了三个参数，则*args* 属性将仅包含由前两个构造器参数组成的 2 元组。

构造器实际返回的往往是*OSError* 的某个子类，如下文*OS exceptions* 中所描述的。具体的子类取决于最终的*errno* 值。此行为仅在直接或通过别名来构造*OSError* 时发生，并且在子类化时不会被继承。

errno
来自于 C 变量 `errno` 的数字错误码。

winerror
在 Windows 下，此参数将给出原生的 Windows 错误码。而*errno* 属性将是该原生错误码在 POSIX 平台下的近似转换形式。

在 Windows 下，如果 *winerror* 构造器参数是一个整数，则*errno* 属性会根据 Windows 错误码来确定，而 *errno* 参数会被忽略。在其他平台上，*winerror* 参数会被忽略，并且*winerror* 属性将不存在。

strerror
操作系统所提供的相应错误信息。它在 POSIX 平台中由 C 函数 `perror()` 来格式化，在 Windows 中则是由 `FormatMessage()`。

filename
filename2
对于与文件系统路径有关 (例如*open()* 或*os.unlink()*) 的异常，*filename* 是传给函数的文件名。对于涉及两个文件系统路径的函数 (例如*os.rename()*)，*filename2* 将是传给函数的第二个文件名。

3.3 版更變: *EnvironmentError*, *IOError*, *WindowsError*, *socket.error*, *select.error* 与 `mmap.error` 已被合并到*OSError*，构造器可能返回其中一个子类。

3.4 版更變: *filename* 属性现在将是传给函数的原始文件名，而不是经过编码或基于文件系统编码进行解码之后的名称。此外还添加了 *filename2* 构造器参数和属性。

exception OverflowError
当算术运算的结果大到无法表示时将被引发。这对整数来说不可能发生（宁可引发*MemoryError* 也不会放弃尝试）。但是出于历史原因，有时也会在整数超出要求范围的情况下引发 OverflowError。因为在 C 中缺少对浮点异常处理的标准化，大多数浮点运算都不会做检查。

exception RecursionError
此异常派生自*RuntimeError*。它会在解释器检测发现超过最大递归深度 (参见*sys.getrecursionlimit()*) 时被引发。

3.5 版新加入: 在此之前将只引发*RuntimeError*。

exception ReferenceError
此异常将在使用*weakref.proxy()* 函数所创建的弱引用来访问该引用的某个已被作为垃圾回收的属性时被引发。有关弱引用的更多信息请参阅*weakref* 模块。

exception RuntimeError
当检测到一个不归属于任何其他类别的错误时将被引发。关联的值是一个指明究竟发生了什么问题的字符串。

exception StopIteration
由内置函数*next()* 和*iterator* 的*__next__()* 方法所引发，用来表示该迭代器不能产生下一项。

该异常对象只有一个属性 `value`，它在构造该异常时作为参数给出，默认值为*None*。

当一个*generator* 或*coroutine* 函数返回时，将引发一个新的*StopIteration* 实例，函数返回的值将被用作异常构造器的 `value` 形参。

如果某个生成器代码直接或间接地引发了*StopIteration*，它会被转换为*RuntimeError* (并将*StopIteration* 保留为导致新异常的原因)。

3.3 版更變: 添加了 `value` 属性及其被生成器函数用作返回值的功能。

3.5 版更變: 引入了通过 `from __future__ import generator_stop` 来实现 RuntimeError 转换，参见 **PEP 479**。

3.7 版更變: 默认对所有代码启用 **PEP 479**: 在生成器中引发的*StopIteration* 错误将被转换为*RuntimeError*。

exception StopAsyncIteration
必须由一个*asynchronous iterator* 对象的 `__anext__()` 方法来引发以停止迭代操作。

3.5 版新加入.

exception SyntaxError(*message*, *details*)
当解析器遇到语法错误时引发。这可以发生在 `import` 语句，对内置函数*compile()*, *exec()* 或*eval()* 的调用，或是读取原始脚本或标准输入（也包括交互模式）的时候。

异常实例的*str()* 只返回错误消息。错误详情为一个元组，其成员也可在单独的属性中分别获取。

filename
发生语法错误所在文件的名称。

lineno
发生错误所在文件中的行号。行号索引从 1 开始：文件中首行的 `lineno` 为 1。

offset
发生错误所在文件中的列号。列号索引从 1 开始：行中首个字符的 `offset` 为 1。

text
错误所涉及的源代码文本。

对于 f-字符串字段中的错误，消息会带有"f-string: " 前缀并且其位置是基于替换表达式构建的文本中的位置。例如，编译 f'Bad {a b} field' 将产生以下 args 属性: ('f-string: ...', ('', 1, 4, '(a b)n'))。

exception IndentationError
与不正确的缩进相关的语法错误的基类。这是*SyntaxError* 的一个子类。

exception TabError
当缩进包含对制表符和空格符不一致的使用时将被引发。这是*IndentationError* 的一个子类。

exception SystemError
当解释器发现内部错误，但情况看起来尚未严重到要放弃所有希望时将被引发。关联的值是一个指明发生了什么问题的字符串（表示为低层级的符号）。

你应当将此问题报告给你所用 Python 解释器的作者或维护人员。请确认报告 Python 解释器的版本号 (`sys.version`; 它也会在交互式 Python 会话开始时被打印出来)，具体的错误消息（异常所关联的值）以及可能触发该错误的程序源码。

exception SystemExit
此异常由*sys.exit()* 函数引发。它继承自*BaseException* 而不是*Exception* 以确保不会被处理*Exception* 的代码意外捕获。这允许此异常正确地向上传播并导致解释器退出。如果它未被处理，则 Python 解释器就将退出；不会打印任何栈回溯信息。构造器接受的可选参数与传递给*sys.exit()* 的相同。如果该值为一个整数，则它指明系统退出状态码（会传递给 C 的 `exit()` 函数）；如果该值为 `None`，则退出状态码为零；如果该值为其他类型（例如字符串），则会打印对象的值并将退出状态码设为一。

对*sys.exit()* 的调用会被转换为一个异常以便能执行清理处理程序 (`try` 语句的 `finally` 子句)，并且使得调试器可以执行一段脚本而不必冒失去控制的风险。如果绝对确实地需要立即退出（例如在调用*os.fork()* 之后的子进程中）则可使用*os._exit()*.

code
传给构造器的退出状态码或错误信息（默认为 `None`。）

exception TypeError
当一个操作或函数被应用于类型不适当的对象时将被引发。关联的值是一个字符串，给出有关类型不匹配的详情。

此异常可以由用户代码引发，以表明尝试对某个对象进行的操作不受支持也不应当受支持。如果某个对象应当支持给定的操作但尚未提供相应的实现，所要引发的适当异常应为*NotImplementedError*。

传入参数的类型错误 (例如在要求*int* 时却传入了*list*) 应当导致*TypeError*，但传入参数的值错误 (例如传入要求范围之外的数值) 则应当导致*ValueError*。

exception UnboundLocalError
当在函数或方法中对某个局部变量进行引用，但该变量并未绑定任何值时将被引发。此异常是*NameError* 的一个子类。

exception UnicodeError
当发生与 Unicode 相关的编码或解码错误时将被引发。此异常是*ValueError* 的一个子类。

UnicodeError 具有一些描述编码或解码错误的属性。例如 `err.object[err.start:err.end]` 会给出导致编解码器失败的特定无效输入。

encoding
引发错误的编码名称。

reason
描述特定编解码器错误的字符串。

object
编解码器试图要编码或解码的对象。

start
object 中无效数据的开始位置索引。

end
object 中无效数据的末尾位置索引（不含）。

exception UnicodeEncodeError
当在编码过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

exception UnicodeDecodeError
当在解码过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

exception UnicodeTranslateError
在转写过程中发生与 Unicode 相关的错误时将被引发。此异常是*UnicodeError* 的一个子类。

exception ValueError
当操作或函数接收到具有正确类型但值不适合的参数，并且情况不能用更精确的异常例如*IndexError* 来描述时将被引发。

exception ZeroDivisionError
当除法或取余运算的第二个参数为零时将被引发。关联的值是一个字符串，指明操作数和运算的类型。

下列异常被保留以与之前的版本相兼容；从 Python 3.3 开始，它们都是*OSError* 的别名。

exception EnvironmentError

exception IOError

exception WindowsError
限在 Windows 中可用。

5.4.1 OS 异常

下列异常均为*OSError* 的子类，它们将根据系统错误代码被引发。

exception BlockingIOError
Raised when an operation would block on an object (e.g. socket) set for non-blocking operation. Corresponds to `errno` *EAGAIN*, *EALREADY*, *EWOULDBLOCK* and *EINPROGRESS*.

除了*OSError* 已有的属性，*BlockingIOError* 还有一个额外属性：

characters_written
一个整数，表示在被阻塞前已写入到流的字符数。当使用来自*io* 模块的带缓冲 I/O 类时此属性可用。

exception ChildProcessError
Raised when an operation on a child process failed. Corresponds to `errno` *ECHILD*.

exception ConnectionError
与连接相关问题的基类。

其子类有*BrokenPipeError*，*ConnectionAbortedError*，*ConnectionRefusedError* 和*ConnectionResetError*。

exception BrokenPipeError
A subclass of *ConnectionError*, raised when trying to write on a pipe while the other end has been closed, or trying to write on a socket which has been shutdown for writing. Corresponds to `errno` *EPIPE* and *ESHUTDOWN*.

exception ConnectionAbortedError
A subclass of *ConnectionError*, raised when a connection attempt is aborted by the peer. Corresponds to `errno` *ECONNABORTED*.

exception ConnectionRefusedError
A subclass of *ConnectionError*, raised when a connection attempt is refused by the peer. Corresponds to `errno` *ECONNREFUSED*.

exception ConnectionResetError
A subclass of *ConnectionError*, raised when a connection is reset by the peer. Corresponds to `errno` *ECONNRESET*.

exception FileExistsError
Raised when trying to create a file or directory which already exists. Corresponds to errno *EEXIST*.

exception FileNotFoundError
Raised when a file or directory is requested but doesn't exist. Corresponds to errno *ENOENT*.

exception InterruptedError
当系统调用被输入信号中断时将被引发。对应于 errno *EINTR*。

3.5 版更變: 当系统调用被某个信号中断时，Python 现在会重试系统调用，除非该信号的处理程序引发了其它异常 (原理参见 **PEP 475**) 而不是引发*InterruptedError*。

exception IsADirectoryError
Raised when a file operation (such as *os.remove()*) is requested on a directory. Corresponds to errno *EISDIR*.

exception NotADirectoryError
Raised when a directory operation (such as *os.listdir()*) is requested on something which is not a directory. On most POSIX platforms, it may also be raised if an operation attempts to open or traverse a non-directory file as if it were a directory. Corresponds to errno *ENOTDIR*.

exception PermissionError
Raised when trying to run an operation without the adequate access rights - for example filesystem permissions. Corresponds to errno *EACCES* and *EPERM*.

exception ProcessLookupError
Raised when a given process doesn't exist. Corresponds to errno *ESRCH*.

exception TimeoutError
Raised when a system function timed out at the system level. Corresponds to errno *ETIMEDOUT*.

3.3 版新加入: 添加了以上所有*OSError* 的子类。

也参考:

PEP 3151 - 重写 OS 和 IO 异常的层次结构

5.5 警告

下列异常被用作警告类别；请参阅警告类别 文档了解详情。

exception Warning
警告类别的基类。

exception UserWarning
用户代码所产生警告的基类。

exception DeprecationWarning
如果所发出的警告是针对其他 Python 开发者的，则以此作为与已弃用特性相关警告的基类。

会被默认警告过滤器忽略，在 __main__ 模块中的情况除外 (**PEP 565**)。启用*Python 开发模式* 时会显示此警告。

The deprecation policy is described in **PEP 387**.

exception PendingDeprecationWarning
对于已过时并预计在未来弃用，但目前尚未弃用的特性相关警告的基类。

这个类很少被使用，因为针对未来可能的弃用发出警告的做法并不常见，而针对当前已有的弃用则推荐使用*DeprecationWarning*。

会被默认警告过滤器忽略。启用*Python 开发模式* 时会显示此警告。

The deprecation policy is described in **PEP 387**.

exception SyntaxWarning
与模糊的语法相关的警告的基类。

exception RuntimeWarning
与模糊的运行时行为相关的警告的基类。

exception FutureWarning
如果所发出的警告是针对以 Python 所编写应用的最终用户的，则以此作为与已弃用特性相关警告的基类。

exception ImportWarning
与在模块导入中可能的错误相关的警告的基类。

会被默认警告过滤器忽略。启用*Python 开发模式* 时会显示此警告。

exception UnicodeWarning
与 Unicode 相关的警告的基类。

exception BytesWarning
与*bytes* 和*bytearray* 相关的警告的基类。

exception ResourceWarning
资源使用相关警告的基类。

会被默认警告过滤器忽略。启用*Python 开发模式* 时会显示此警告。

3.2 版新加入.

5.6 异常层次结构

内置异常的类层级结构如下：

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
```

(下页继续)

(繼續上一頁)

```
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

CHAPTER 6

文本處理 (Text Processing) 服務

本章節介紹的模組 (module) 提供了廣泛的字串操作與其他文本處理服務。

在二进制数据服务 下所描述的*codecs* 模組也與文本處理高度相關。另外也請參🄵在文本序列类型 --- *str* 中所描述的 Python 🄵建字串型🄵。

6.1 `string` --- 常见的字符串操作

源代码： Lib/string.py

也参考：

文本序列类型 --- *str*

字符串的方法

6.1.1 字符串常量

此模块中定义的常量为：

`string.`**`ascii_letters`**
下文所述*ascii_lowercase* 和*ascii_uppercase* 常量的拼连。该值不依赖于语言区域。

`string.`**`ascii_lowercase`**
小写字母 `'abcdefghijklmnopqrstuvwxyz'`。该值不依赖于语言区域，不会发生改变。

`string.`**`ascii_uppercase`**
大写字母 `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`。该值不依赖于语言区域，不会发生改变。

`string.`**`digits`**
字符串 `'0123456789'`。

string.**hexdigits**
字符串 '0123456789abcdefABCDEF'。

string.**octdigits**
字符串 '01234567'。

string.**punctuation**
由在 C 区域设置中被视为标点符号的 ASCII 字符所组成的字符串: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~.

string.**printable**
由被视为可打印符号的 ASCII 字符组成的字符串。这是*digits*, *ascii_letters*, *punctuation* 和*whitespace* 的总和。

string.**whitespace**
由被视为空白符号的 ASCII 字符组成的字符串。其中包括空格、制表、换行、回车、进纸和纵向制表符。

6.1.2 自定义字符串格式化

内置的字符串类提供了通过使用 **PEP 3101** 所描述的*format()* 方法进行复杂变量替换和值格式化的能力。*string* 模块中的*Formatter* 类允许你使用与内置*format()* 方法相同的实现来创建并定制你自己的字符串格式化行为。

class string.**Formatter**
Formatter 类包含下列公有方法:

format(*format_string*, */*, **args*, ***kwargs*)
首要的 API 方法。它接受一个格式字符串和任意一组位置和关键字参数。它只是一个调用*vformat()* 的包装器。

3.7 版更變: 格式字符串参数现在是仅限位置参数。

vformat(*format_string*, *args*, *kwargs*)
此函数执行实际的格式化操作。它被公开为一个单独的函数，用于需要传入一个预定义字母作为参数，而不是使用 *args 和 **kwargs 语法将字典解包为多个单独参数并重打包的情况。*vformat()* 完成将格式字符串分解为字符数据和替换字段的工作。它会调用下文所述的几种不同方法。

此外，*Formatter* 还定义了一些旨在被子类替换的方法:

parse(*format_string*)
循环遍历 format_string 并返回一个由可迭代对象组成的元组 (*literal_text*, *field_name*, *format_spec*, *conversion*)。它会被*vformat()* 用来将字符串分解为文本字面值或替换字段。

元组中的值在概念上表示一段字面文本加上一个替换字段。如果没有字面文本（如果连续出现两个替换字段就会发生这种情况），则 *literal_text* 将是一个长度为零的字符串。如果没有替换字段，则 *field_name*, *format_spec* 和 *conversion* 的值将为 None。

get_field(*field_name*, *args*, *kwargs*)
给定 *field_name* 作为*parse()* (见上文) 的返回值，将其转换为要格式化的对象。返回一个元组 (obj, used_key)。默认版本接受在 **PEP 3101** 所定义形式的字符串，例如"0[name]" 或"label.title"。*args* 和 *kwargs* 与传给*vformat()* 的一样。返回值 *used_key* 与*get_value()* 的 *key* 形参具有相同的含义。

get_value(*key*, *args*, *kwargs*)
提取给定的字段值。*key* 参数将为整数或字符串。如果是整数，它表示 *args* 中位置参数的索引；如果是字符串，它表示 *kwargs* 中的关键字参数名。

args 形参会被设为*vformat()* 的位置参数列表，而 *kwargs* 形参会被设为由关键字参数组成的字典。

对于复合字段名称，仅会为字段名称的第一个组件调用这些函数；后续组件会通过普通属性和索引操作来进行处理。

因此举例来说，字段表达式'0.name'将导致调用*get_value()*时附带 *key* 参数值 0。在*get_value()*通过调用内置的*getattr()*函数返回后将会查找 name 属性。

如果索引或关键字引用了一个不存在的项，则将引发*IndexError*或*KeyError*。

check_unused_args(*used_args*, *args*, *kwargs*)
在必要时实现对未使用参数进行检测。此函数的参数是是格式字符串中实际引用的所有参数键的集合（整数表示位置参数，字符串表示名称参数），以及被传给 vformat 的 *args* 和 *kwargs* 的引用。未使用参数的集合可以根据这些形参计算出来。如果检测失败则*check_unused_args()*应会引发一个异常。

format_field(*value*, *format_spec*)
*format_field()*会简单地调用内置全局函数*format()*。提供该方法是为了让子类能够重载它。

convert_field(*value*, *conversion*)
使用给定的转换类型（来自*parse()*方法所返回的元组）来转换（由*get_field()*所返回的）值。默认版本支持's' (str), 'r' (repr) 和'a' (ascii) 等转换类型。

6.1.3 格式字符串语法

*str.format()*方法和*Formatter*类共享相同的格式字符串语法（虽然对于*Formatter*来说，其子类可以定义它们自己的格式字符串语法）。具体语法与 格式化字符串字面值相似，但较为简单一些，并且关键的一点是不支持任意表达式。

格式字符串包含有以花括号 {} 括起来的 "替换字段"。不在花括号之内的内容被视为字面文本，会不加修改地复制到输出中。如果你需要在字面文本中包含花括号字符，可以通过重复来转义: {{ and }}。

替换字段的语法如下：

```
replacement_field  ::=  "{" [field_name] ["!" conversion] [":" format_spec] "}"
field_name         ::=  arg_name ("." attribute_name | "[" element_index "]")*
arg_name           ::=  [identifier | digit+]
attribute_name     ::=  identifier
element_index      ::=  digit+ | index_string
index_string       ::=  <any source character except "]"> +
conversion         ::=  "r" | "s" | "a"
format_spec        ::=  <described in the next section>
```

用不太正式的术语来描述，替换字段开头可以用一个 *field_name* 指定要对值进行格式化并取代替换字符被插入到输出结果的对象。*field_name* 之后有可选的 *conversion* 字段，它是一个感叹号 '!' 加一个 *format_spec*，并以一个冒号 ':' 打头。这些指明了替换值的非默认格式。

另请参阅格式规格迷你语言 一节。

field_name 本身以一个数字或关键字 *arg_name* 打头。如果为数字，则它指向一个位置参数，而如果为关键字，则它指向一个命名关键字参数。如果格式字符串中的数字 arg_names 为 0, 1, 2, ... 的序列，它们可以全部省略（而非部分省略），数字 0, 1, 2, ... 将会按顺序自动插入。由于 *arg_name* 不使用引号分隔，因此无法在格式字符串中指定任意的字典键 (例如字符串 '10' 或 ':-]')。*arg_name* 之后可以带上任意数量的索引或属性表达式。'.name' 形式的表达式会使用*getattr()*选择命名属性，而 '[index]' 形式的表达式会使用 __getitem__() 执行索引查找。

3.1 版更變: 位置参数说明符对于*str.format()*可以省略，因此 '{} {}'.format(a, b) 等价于 '{0} {1}'.format(a, b)。

3.4 版更變: 位置参数说明符对于*Formatter* 可以省略。

一些简单的格式字符串示例

```
"First, thou shalt count to {0}"  # References first positional argument
"Bring me a {}"                   # Implicitly references the first positional␣
↪argument
"From {} to {}"                   # Same as "From {0} to {1}"
"My quest is {name}"              # References keyword argument 'name'
"Weight in tons {0.weight}"       # 'weight' attribute of first positional arg
"Units destroyed: {players[0]}"   # First element of keyword argument 'players'.
```

使用 *conversion* 字段在格式化之前进行类型强制转换。通常，格式化值的工作由值本身的 `__format__()` 方法来完成。但是，在某些情况下最好强制将类型格式化为一个字符串，覆盖其本身的格式化定义。通过在调用 `__format__()` 之前将值转换为字符串，可以绕过正常的格式化逻辑。

目前支持的转换旗标有三种: `'!s'` 会对值调用*str()*，`'!r'` 调用*repr()* 而 `'!a'` 则调用*ascii()*。

一些範例:

```
"Harold's a clever {0!s}"         # Calls str() on the argument first
"Bring out the holy {name!r}"     # Calls repr() on the argument first
"More {!a}"                       # Calls ascii() on the argument first
```

format_spec 字段包含值应如何呈现的规格描述，例如字段宽度、对齐、填充、小数精度等细节信息。每种值类型可以定义自己的“格式化迷你语言”或对 *format_spec* 的解读方式。

大多数内置类型都支持同样的格式化迷你语言，具体描述见下一节。

format_spec 字段还可以在其内部包含嵌套的替换字段。这些嵌套的替换字段可能包括字段名称、转换旗标和格式规格描述，但是不再允许更深层的嵌套。format_spec 内部的替换字段会在解读 *format_spec* 字符串之前先被解读。这将允许动态地指定特定值的格式。

请参阅格式示例 一节查看相关示例。

格式规格迷你语言

“格式规格”在格式字符串所包含的替换字段内部使用，用于定义单个值应如何呈现 (参见格式字符串语法 和 f-strings)。它们也可以被直接传给内置的*format()* 函数。每种可格式化的类型都可以自行定义如何对格式规格进行解读。

大多数内置类型都为格式规格实现了下列选项，不过某些格式化选项只被数值类型所支持。

一般约定空的格式描述将产生与在值上调用*str()* 相同的结果。非空格式描述通常会修改此结果。

标准格式说明符的一般形式如下:

```
format_spec      ::=  [[fill]align][sign][#][0][width][grouping_option][.precision][type]
fill             ::=  <any character>
align            ::=  "<" | ">" | "=" | "^"
sign             ::=  "+" | "-" | " "
width            ::=  digit+
grouping_option  ::=  "_" | ","
precision        ::=  digit+
type             ::=  "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s
```

如果指定了一个有效的 *align* 值，则可以在该值前面加一个 *fill* 字符，它可以为任意字符，如果省略则默认为空格符。在 格式化字符串字面值或在使用*str.format()* 方法时是无法使用花括号字面值 (“{” or “}”) 作为

fill 字符的。但是，通过嵌套替换字段插入花括号则是可以的。这个限制不会影响`format()`函数。

各种对齐选项的含义如下：

选项	意义
'<'	强制字段在可用空间内左对齐（这是大多数对象的默认值）。
'>'	强制字段在可用空间内右对齐（这是数字的默认值）。
'='	强制将填充放置在符号（如果有）之后但在数字之前。这用于以“+000000120”形式打印字段。此对齐选项仅对数字类型有效。当'0'紧接在字段宽度之前时，它成为默认值。
'^'	强制字段在可用空间内居中。

请注意，除非定义了最小字段宽度，否则字段宽度将始终与填充它的数据大小相同，因此在这种情况下，对齐选项没有意义。

sign 选项仅对数字类型有效，可以是以下之一：

选项	意义
'+'	表示标志应该用于正数和负数。
'-'	表示标志应仅用于负数（这是默认行为）。
space	表示应在正数上使用前导空格，在负数上使用减号。

'#' 选项可让“替代形式”被用于执行转换。替代形式会针对不同的类型分别定义。此选项仅适用于整数、浮点数和复数类型。对于整数类型，当使用二进制、八进制或十六进制输出时，此选项会为输出值分别添加相应的 '0b','0o','0x' 或 '0X' 前缀。对于浮点数和复数类型，替代形式会使得转换结果总是包含小数点符号，即使其不带小数部分。通常只有在带有小数部分的情况下，此类转换的结果中才会出现小数点符号。此外，对于 'g' 和 'G' 转换，末尾的零不会从结果中被移除。

',' 选项表示使用逗号作为千位分隔符。对于感应区域设置的分隔符，请改用 'n' 整数表示类型。

3.1 版更變: 添加了 ',' 选项 (另请参阅 **PEP 378**)。

'_' 选项表示对浮点表示类型和整数表示类型 'd' 使用下划线作为千位分隔符。对于整数表示类型 'b', 'o','x' 和 'X'，将为每 4 个数位插入一个下划线。对于其他表示类型指定此选项则将导致错误。

3.6 版更變: 添加了 '_' 选项 (另请参阅 **PEP 515**)。

width 是一个定义最小总字段宽度的十进制整数，包括任何前缀、分隔符和其他格式化字符。如果未指定，则字段宽度将由内容确定。

当未显式给出对齐方式时，在 *width* 字段前加一个零 ('0') 字段将为数字类型启用感知正负号的零填充。这相当于设置 *fill* 字符为 '0' 且 *alignment* 类型为 '='。

The *precision* is a decimal integer indicating how many digits should be displayed after the decimal point for presentation types 'f' and 'F', or before and after the decimal point for presentation types 'g' or 'G'. For string presentation types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The *precision* is not allowed for integer presentation types.

最后，*type* 确定了数据应如何呈现。

可用的字符串表示类型是：

类型	意义
's'	字符串格式。这是字符串的默认类型，可以省略。
None	和 's' 一样。

可用的整数表示类型是：

类型	意义
'b'	二进制格式。输出以 2 为基数的数字。
'c'	字符。在打印之前将整数转换为相应的 unicode 字符。
'd'	十进制整数。输出以 10 为基数的数字。
'o'	八进制格式。输出以 8 为基数的数字。
'x'	十六进制格式。输出以 16 为基数的数字，使用小写字母表示 9 以上的数码。
'X'	十六进制格式。输出以 16 为基数的数字，使用大写字母表示 9 以上的数码。在指定 '#' 的情况下，前缀 '0x' 也将被转为大写形式 '0X'。
'n'	数字。这与 'd' 相似，不同之处在于它会使用当前区域设置来插入适当的数字分隔字符。
None	和 'd' 相同。

在上述的表示类型之外，整数还可以通过下列的浮点表示类型来格式化 (除了 'n' 和 None)。当这样做时，会在格式化之前使用*float()* 将整数转换为浮点数。

float 和*Decimal* 值的可用表示类型有:

类型	意义
'e'	科学计数法。对于一个给定的精度 p，将数字格式化为以字母'e' 分隔系数和指数的科学计数法形式。系数在小数点之前有一位，之后有 p 位，总计 p + 1 个有效数位。如未指定精度，则会对*float* 采用小数点之后 6 位精度，而对*Decimal* 则显示所有系数位。如果小数点之后没有数位，则小数点也会被略去，除非使用了 # 选项。
'E'	科学计数法。与 'e' 相似，不同之处在于它使用大写字母'E' 作为分隔字符。
'f'	定点表示法。对于一个给定的精度 p，将数字格式化为在小数点之后恰好有 p 位的小数形式。如未指定精度，则会对*float* 采用小数点之后 6 位精度，而对*Decimal* 则使用大到足够显示所有系数位的精度。如果小数点之后没有数位，则小数点也会被略去，除非使用了 # 选项。
'F'	定点表示。与 'f' 相似，但会将 nan 转为 NAN 并将 inf 转为 INF。
'g'	常规格式。对于给定精度 p >= 1，这会将数值舍入到 p 个有效数位，再将结果以定点表示法或科学计数法进行格式化，具体取决于其值的大小。精度 0 会被视为等价于精度 1。 准确的规则如下：假设使用表示类型 'e' 和精度 p-1 进行格式化的结果具有指数值 exp。那么如果 m <= exp < p，其中 m 以 -4 表示浮点值而以 -6 表示*Decimal* 值，该数字将使用类型 'f' 和精度 p-1-exp 进行格式化。否则的话，该数字将使用表示类型 'e' 和精度 p-1 进行格式化。在两种情况下，都会从有效数字中移除无意义的末尾零，如果小数点之后没有余下数字则小数点也会被移除，除非使用了 '#' 选项。 如未指定精度，会对*float* 采用 6 个有效数位的精度。对于*Decimal*，结果的系数会沿用原值的系数数位；对于绝对值小于 1e-6 的值以及最小有效数位的位值大于 1 的数值将会使用科学计数法，在其他情况下则会使用定点表示法。 正负无穷，正负零和 nan 会分别被格式化为 inf, -inf, 0, -0 和 nan，无论精度如何设定。
'G'	常规格式。类似于 'g'，不同之处在于当数值非常大时会切换为 'E'。无穷与 NaN 也会表示为大写形式。
'n'	数字。这与 'g' 相似，不同之处在于它会使用当前区域设置来插入适当的数字分隔字符。
'%'	百分比。将数字乘以 100 并显示为定点 ('f') 格式，后面带一个百分号。
None	对于*float* 来说这类似于 'g'，不同之处在于当使用定点表示法时，小数点之后将至少显示一位。所用的精度会大到足以精确表示给定的值。 对于*Decimal* 来说这相当于 'g' 或 'G'，具体取决于当前 decimal 上下文的 context.capitals 值。 总体效果是将*str()* 的输出匹配为其他格式化因子所调整出的样子。

格式示例

本节包含*str.format()* 语法的示例以及与旧式 % 格式化的比较。

该语法在大多数情况下与旧式的 % 格式化类似，只是增加了 {} 和 : 来取代 %。例如，, '%03.2f' 可以被改写为 '{:03.2f}'。

新的格式语法还支持新增的不同选项，将在以下示例中说明。

按位置访问参数:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')  # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
```

(下页继续)

(繼續上一頁)

```
>>> '{2}, {1}, {0}'.format(*'abc')      # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')   # arguments' indices can be repeated
'abracadabra'
```

按名称访问参数:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.
↪81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

访问参数的属性:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}.').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.
↪0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

访问参数的项:

```
>>> coord = (3, 5)
>>> 'X: {0[0]};  Y: {0[1]}'.format(coord)
'X: 3;  Y: 5'
```

替代 `%s` 和 `%r`:

```
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1', 'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
```

对齐文本以及指定宽度:

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '
>>> '{:>30}'.format('right aligned')
'                 right aligned'
>>> '{:^30}'.format('centered')
'           centered           '
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'***********centered***********'
```

替代 `%+f`, `%-f` 和 `% f` 以及指定正负号:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14)  # show it always
'+3.140000; -3.140000'
```

(下页继续)

(繼續上一頁)

```
>>> '{: f}; {: f}'.format(3.14, -3.14)  # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)  # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

替代 %x 和 %o 以及转换基于不同进位制的值:

```
>>> # format also supports binary numbers
>>> "int: {0:d};  hex: {0:x};  oct: {0:o};  bin: {0:b}".format(42)
'int: 42;  hex: 2a;  oct: 52;  bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d};  hex: {0:#x};  oct: {0:#o};  bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

使用逗号作为千位分隔符:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

表示为百分数:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

使用特定类型的专属格式化:

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

嵌套参数以及更复杂的示例:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
    5     5     5   101
    6     6     6   110
```

(下页继续)

(繼續上一頁)

```
 7     7     7   111
 8     8    10  1000
 9     9    11  1001
10     A    12  1010
11     B    13  1011
```

6.1.4 模板字符串

模板字符串提供了由 **PEP 292** 所描述的更简便的字符串替换方式。模板字符串的一个主要用例是文本国际化 (i18n)，因为在此场景下，更简单的语法和功能使得文本翻译过程比使用 Python 的其他内置字符串格式化工具更为方便。作为基于模板字符串构建以实现 i18n 的库的一个示例，请参看 flufl.i18n 包。

模板字符串支持基于 `$` 的替换，使用以下规则：

- `$$` 为转义符号；它会被替换为单个的 `$`。
- `$identifier` 为替换占位符，它会匹配一个名为 `"identifier"` 的映射键。在默认情况下，`"identifier"` 限制为任意 ASCII 字母数字（包括下划线）组成的字符串，不区分大小写，以下划线或 ASCII 字母开头。在 `$` 字符之后的第一个非标识符字符将表明占位符的终结。
- `${identifier}` 等价于 `$identifier`。当占位符之后紧跟着有效的但又不是占位符一部分的标识符字符时需要使用，例如 `"${noun}ification"`。

在字符串的其他位置出现 `$` 将导致引发*ValueError*。

string 模块提供了实现这些规则的*Template* 类。*Template* 有下列方法：

class string.**Template**(*template*)

该构造器接受一个参数作为模板字符串。

substitute(*mapping={}*, */*, ***kwds*)

执行模板替换，返回一个新字符串。*mapping* 为任意字典类对象，其中的键将匹配模板中的占位符。或者你也可以提供一组关键字参数，其中的关键字即对应占位符。当同时给出 *mapping* 和 *kwds* 并且存在重复时，则以 *kwds* 中的占位符为优先。

safe_substitute(*mapping={}*, */*, ***kwds*)

类似于*substitute()*，不同之处是如果有占位符未在 *mapping* 和 *kwds* 中找到，不是引发*KeyError*异常，而是将原始占位符不加修改地显示在结果字符串中。另一个与*substitute()*的差异是任何在其他情况下出现的 `$` 将简单地返回 `$` 而不是引发*ValueError*。

此方法被认为“安全”，因为虽然仍有可能发生其他异常，但它总是尝试返回可用的字符串而不是引发一个异常。从另一方面来说，*safe_substitute()* 也可能根本算不上安全，因为它将静默地忽略错误格式的模板，例如包含多余的分隔符、不成对的花括号或不是合法 Python 标识符的占位符等等。

Template 的实例还提供一个公有数据属性：

template

这是作为构造器的 *template* 参数被传入的对象。一般来说，你不应该修改它，但并不强制要求只读访问。

以下是一个如何使用模版的示例：

```
>>> from string import Template
>>> s = Template('$who likes $what')
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
```

(下页继续)

(繼續上一頁)

```
>>> Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

进阶用法：你可以派生 *Template* 的子类来自定义占位符语法、分隔符，或用于解析模板字符串的整个正则表达式。为此目的，你可以重载这些类属性：

- *delimiter* -- 这是用来表示占位符的起始的分隔符的字符串字面值。默认值为 `$`。请注意此参数 不能为正则表达式，因为其实现将在必要时对此字符串调用 *re.escape()*。还要注意你不能在创建类之后改变此分隔符（例如在子类的类命名空间中必须设置不同的分隔符）。
- *idpattern* -- 这是用来描述不带花括号的占位符的模式的正则表达式。默认值为正则表达式 `(?a:[_a-z][_a-z0-9]*)`。如果给出了此属性并且 *braceidpattern* 为 `None` 则此模式也将作用于带花括号的占位符。

  備F: 由于默认的 *flags* 为 `re.IGNORECASE`，模式 `[a-z]` 可以匹配某些非 ASCII 字符。因此我们在这里使用了局部旗标 `a`。

  3.7 版更變: *braceidpattern* 可被用来定义对花括号内部和外部进行区分的模式。

- *braceidpattern* -- 此属性类似于 *idpattern* 但是用来描述带花括号的占位符的模式。默认值 `None` 意味着回退到 *idpattern* (即在花括号内部和外部使用相同的模式)。如果给出此属性，这将允许你为带花括号和不带花括号的占位符定义不同的模式。

  3.7 版新加入.

- *flags* -- 将在编译用于识别替换内容的正则表达式被应用的正则表达式旗标。默认值为 `re.IGNORECASE`。请注意 `re.VERBOSE` 总是会被加为旗标，因此自定义的 *idpattern* 必须遵循详细正则表达式的约定。

  3.2 版新加入.

作为另一种选项，你可以通过重载类属性 *pattern* 来提供整个正则表达式模式。如果你这样做，该值必须为一个具有四个命名捕获组的正则表达式对象。这些捕获组对应于上面已经给出的规则，以及无效占位符的规则：

- *escaped* -- 这个组匹配转义序列，在默认模式中即 `$$`。
- *named* -- 这个组匹配不带花括号的占位符名称；它不应当包含捕获组中的分隔符。
- *braced* -- 这个组匹配带有花括号的占位符名称；它不应当包含捕获组中的分隔符或者花括号。
- *invalid* -- 这个组匹配任何其他分隔符模式（通常为单个分隔符），并且它应当出现在正则表达式的末尾。

6.1.5 辅助函数

string.**capwords**(*s*, *sep=None*)
使用*str.split()* 将参数拆分为单词，使用*str.capitalize()* 将单词转为大写形式，使用*str.join()* 将大写的单词进行拼接。如果可选的第二个参数 *sep* 被省略或为 None，则连续的空白字符会被替换为单个空格符并且开头和末尾的空白字符会被移除，否则 *sep* 会被用来拆分和拼接单词。

6.2 re --- 正则表达式操作

源代码: Lib/re.py

本模块提供了与 Perl 语言类似的正则表达式匹配操作。

模式和被搜索的字符串既可以是 Unicode 字符串 (*str*)，也可以是 8 位字节串 (*bytes*)。但是，Unicode 字符串与 8 位字节串不能混用：也就是说，不能用字节串模式匹配 Unicode 字符串，反之亦然；同理，替换操作时，替换字符串的类型也必须与所用的模式和搜索字符串的类型一致。

正则表达式用反斜杠字符 ('\') 表示特殊形式，或是允许在使用特殊字符时，不引发它们的特殊含义。这与 Python 的字符串字面值中对相同字符出于相同目的的用法产生冲突；例如，要匹配一个反斜杠字面值，用户可能必须写成 '\\\\' 来作为模式字符串，因为正则表达式必须为 \\，而每个反斜杠在普通 Python 字符串字面值中又必须表示为 \\。而且还要注意，在 Python 的字符串字面值中使用的反斜杠如果有任何无效的转义序列，现在会触发*DeprecationWarning*，但以后会改为*SyntaxError*。此行为即使对于正则表达式来说有效的转义字符同样会发生。

解决办法是对于正则表达式样式使用 Python 的原始字符串表示法；在带有 'r' 前缀的字符串字面值中，反斜杠不必做任何特殊处理。因此 r"\n" 表示包含 '\' 和 'n' 两个字符的字符串，而 "\n" 则表示只包含一个换行符的字符串。样式在 Python 代码中通常都使用原始字符串表示法。

绝大多数正则表达式操作都提供为模块函数和方法，在编译正则表达式. 这些函数是一个捷径，不需要先编译正则对象，但是损失了一些优化参数。

也参考:

第三方模块 regex，提供了与标准库*re* 模块兼容的 API 接口，同时，还提供了更多功能和更全面的 Unicode 支持。

6.2.1 正则表达式语法

正则表达式（或 RE）指定了一组与之匹配的字符串；模块内的函数可以检查某个字符串是否与给定的正则表达式匹配（或者正则表达式是否匹配到字符串，这两种说法含义相同）。

正则表达式可以拼接；如果 *A* 和 *B* 都是正则表达式，则 *AB* 也是正则表达式。通常，如果字符串 *p* 匹配 *A*，并且另一个字符串 *q* 匹配 *B*，那么 *pq* 可以匹配 AB。除非 *A* 或者 *B* 包含低优先级操作，*A* 和 *B* 存在边界条件；或者命名组引用。所以，复杂表达式可以很容易的从这里描述的简单源语表达式构建。更多正则表达式理论和实现，详见 the Friedl book [Frie09]，或者其他构建编译器的书籍。

以下是正则表达式格式的简要说明。更详细的信息和演示，参考 regex-howto。

正则表达式可以包含普通或者特殊字符。绝大部分普通字符，比如 'A', 'a', 或者 '0'，都是最简单的正则表达式。它们就匹配自身。你可以拼接普通字符，所以 last 匹配字符串 'last'. （在这一节的其他部分，我们将用 this special style 这种方式表示正则表达式，通常不带引号，要匹配的字符串用 'in single quotes'，单引号形式。）

有些字符，比如 '|' 或者 '('，属于特殊字符。特殊字符既可以表示它的普通含义，也可以影响它旁边的正则表达式的解释。

重复修饰符 (*, +, ?, {m,n}, 等) 不能直接嵌套。这样避免了非贪婪后缀 ? 修饰符，和其他实现中的修饰符产生的多义性。要应用一个内层重复嵌套，可以使用括号。比如，表达式 (?:a{6})* 匹配 6 个 'a' 字符重复任意次数。

特殊字符有：

. (点) 在默认模式，匹配除了换行的任意字符。如果指定了标签*DOTALL*，它将匹配包括换行符的任意字符。

^ (插入符号) 匹配字符串的开头，并且在*MULTILINE* 模式也匹配换行后的首个符号。

$ 匹配字符串尾或者在字符串尾的换行符的前一个字符，在*MULTILINE* 模式下也会匹配换行符之前的文本。foo 匹配'foo' 和'foobar'，但正则表达式 foo$ 只匹配'foo'。更有趣的是，在 'foo1\nfoo2\n' 中搜索 foo.$，通常匹配'foo2'，但在*MULTILINE* 模式下可以匹配到'foo1'；在 'foo\n' 中搜索 $ 会找到两个（空的）匹配：一个在换行符之前，一个在字符串的末尾。

* 对它前面的正则式匹配 0 到任意次重复，尽量多的匹配字符串。ab* 会匹配 'a'，'ab'，或者 'a' 后面跟随任意个 'b'。

+ 对它前面的正则式匹配 1 到任意次重复。ab+ 会匹配 'a' 后面跟随 1 个以上到任意个 'b'，它不会匹配 'a'。

? 对它前面的正则式匹配 0 到 1 次重复。ab? 会匹配 'a' 或者 'ab'。

?, +?, ?? '', '+'，和 '?' 修饰符都是 贪婪的；它们在字符串进行尽可能多的匹配。有时候并不需要这种行为。如果正则式 <.*> 希望找到 '<a> b <c>'，它将会匹配整个字符串，而不仅是 '<a>'。在修饰符之后添加 ? 将使样式以 非贪婪 *方式或者:dfn:*最小方式进行匹配；尽量 少的字符将会被匹配。使用正则式 <.*?> 将会仅仅匹配 '<a>'。

{m} 对其之前的正则式指定匹配 *m* 个重复；少于 *m* 的话就会导致匹配失败。比如，a{6} 将匹配 6 个 'a'，但是不能是 5 个。

{m,n} 对正则式进行 *m* 到 *n* 次匹配，在 *m* 和 *n* 之间取尽量多。比如，a{3,5} 将匹配 3 到 5 个 'a'。忽略 *m* 意为指定下界为 0，忽略 *n* 指定上界为无限次。比如 a{4,}b 将匹配 'aaaab' 或者 1000 个 'a' 尾随一个 'b'，但不能匹配 'aaab'。逗号不能省略，否则无法辨别修饰符应该忽略哪个边界。

{m,n}? 前一个修饰符的非贪婪模式，只匹配尽量少的字符次数。比如，对于 'aaaaaa'，a{3,5} 匹配 5 个 'a'，而 a{3,5}? 只匹配 3 个 'a'。

\ 转义特殊字符（允许你匹配 '*', '?', 或者此类其他），或者表示一个特殊序列；特殊序列之后进行讨论。

如果你没有使用原始字符串（r'raw'）来表达样式，要牢记 Python 也使用反斜杠作为转义序列；如果转义序列不被 Python 的分析器识别，反斜杠和字符才能出现在字符串中。如果 Python 可以识别这个序列，那么反斜杠就应该重复两次。这将导致理解障碍，所以高度推荐，就算是最简单的表达式，也要使用原始字符串。

[] 用于表示一个字符集合。在一个集合中：

- 字符可以单独列出，比如 [amk] 匹配 'a'，'m'，或者 'k'。
- 可以表示字符范围，通过用 '-' 将两个字符连起来。比如 [a-z] 将匹配任何小写 ASCII 字符，[0-5][0-9] 将匹配从 00 到 59 的两位数字，[0-9A-Fa-f] 将匹配任何十六进制数位。如果 - 进行了转义（比如 [a\-z]）或者它的位置在首位或者末尾（如 [-a] 或 [a-]），它就只表示普通字符 '-'。
- 特殊字符在集合中，失去它的特殊含义。比如 [(+*)] 只会匹配这几个文法字符 '(', '+', '*', or ')'。
- 字符类如 \w 或者 \S (如下定义) 在集合内可以接受，它们可以匹配的字符由*ASCII* 或者*LOCALE* 模式决定。
- 不在集合范围内的字符可以通过 取反来进行匹配。如果集合首字符是 '^'，所有 不在集合内的字符将会被匹配，比如 [^5] 将匹配所有字符，除了 '5'，[^^] 将匹配所有字符，除了 '^'. ^ 如果不在集合首位，就没有特殊含义。

- 在集合内要匹配一个字符 `']'`，有两种方法，要么就在它之前加上反斜杠，要么就把它放到集合首位。比如，`[()[\]{}]` 和 `[]()[{}]` 都可以匹配括号。
- Unicode Technical Standard #18 里的嵌套集合和集合操作支持可能在未来添加。这将会改变语法，所以为了帮助这个改变，一个*FutureWarning* 将会在有多义的情况里被 `raise`，包含以下几种情况，集合由 `'['` 开始，或者包含下列字符序列 `'--'`, `'&&'`, `'~~'`, 和 `'||'`。为了避免警告，需要将它们用反斜杠转义。

3.7 版更變: 如果一个字符串构建的语义在未来会改变的话，一个*FutureWarning* 会 `raise` 。

| `A|B`，*A* 和 *B* 可以是任意正则表达式，创建一个正则表达式，匹配 *A* 或者 *B*. 任意个正则表达式可以用 `'|'` 连接。它也可以在组合（见下列）内使用。扫描目标字符串时，`'|'` 分隔开的正则样式从左到右进行匹配。当一个样式完全匹配时，这个分支就被接受。意思就是，一旦 *A* 匹配成功，*B* 就不再进行匹配，即便它能产生一个更好的匹配。或者说，`'|'` 操作符绝不贪婪。如果要匹配 `'|'` 字符，使用 `\|`，或者把它包含在字符集里，比如 `[|]`.

(...)（组合），匹配括号内的任意正则表达式，并标识出组合的开始和结尾。匹配完成后，组合的内容可以被获取，并可以在之后用 `\number` 转义序列进行再次匹配，之后进行详细说明。要匹配字符 `'('` 或者 `')'`, 用 `\(` 或 `\)`, 或者把它们包含在字符集合里: `[(]`, `[)]`.

(?…) 这是个扩展标记法（一个 `'?'` 跟随 `'('` 并无含义）。`'?'` 后面的第一个字符决定了这个构建采用什么样的语法。这种扩展通常并不创建新的组合；`(?P<name>...)` 是唯一的例外。以下是目前支持的扩展。

(?aiLmsux) (`'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'` 中的一个或多个) 这个组合匹配一个空字符串；这些字符对正则表达式设置以下标记*re.A* (只匹配 ASCII 字符), *re.I* (忽略大小写), *re.L* (语言依赖), *re.M* (多行模式), *re.S* (点 dot 匹配全部字符), `re.U` (Unicode 匹配), and *re.X* (冗长模式)。(这些标记在模块内容 中描述) 如果你想将这些标记包含在正则表达式中，这个方法就很有用，免去了在*re.compile()* 中传递 *flag* 参数。标记应该在表达式字符串首位表示。

(?:…) 正则括号的非捕获版本。匹配在括号内的任何正则表达式，但该分组所匹配的子字符串 不能在执行匹配后被获取或是之后在模式中被引用。

(?aiLmsux-imsx:…) (`'a'`, `'i'`, `'L'`, `'m'`, `'s'`, `'u'`, `'x'` 中的 0 或者多个，之后可选跟随 `'-'` 在后面跟随 `'i'` , `'m'` , `'s'` , `'x'` 中的一到多个.) 这些字符为表达式的其中一部分 设置或者 去除相应标记*re.A* (只匹配 ASCII), *re.I* (忽略大小写), *re.L* (语言依赖), *re.M* (多行), *re.S* (点匹配所有字符), `re.U` (Unicode 匹配), and *re.X* (冗长模式)。(标记描述在模块内容 .)

`'a'`, `'L'` and `'u'` 作为内联标记是相互排斥的，所以它们不能结合在一起，或者跟随 `'-'` 。当他们中的某个出现在内联组中，它就覆盖了括号组内的匹配模式。在 Unicode 样式中，`(?a:...)` 切换为只匹配 ASCII，`(?u:...)` 切换为 Unicode 匹配 (默认). 在 byte 样式中 `(?L:...)` 切换为语言依赖模式，`(?a:...)` 切换为只匹配 ASCII (默认)。这种方式只覆盖组合内匹配，括号外的匹配模式不受影响。

3.6 版新加入.

3.7 版更變: 符号 `'a'`, `'L'` 和 `'u'` 同样可以用在一个组合内。

(?P<name>…)（命名组合）类似正则组合，但是匹配到的子串组在外部是通过定义的 *name* 来获取的。组合名必须是有效的 Python 标识符，并且每个组合名只能用一个正则表达式定义，只能定义一次。一个符号组合同样是一个数字组合，就像这个组合没有被命名一样。

命名组合可以在三种上下文中引用。如果样式是 `(?P<quote>['"]).*?(?P=quote)` （也就是说，匹配单引号或者双引号括起来的字符串):

引用组合"quote" 的上下文	引用方法
在正则式自身内	• `(?P=quote)` (如示) • `\1`
处理匹配对象 *m*	• `m.group('quote')` • `m.end('quote')` (等)
传递到 `re.sub()` 里的 *repl* 参数中	• `\g<quote>` • `\g<1>` • `\1`

(?P=name) 反向引用一个命名组合；它匹配前面那个叫 *name* 的命名组中匹配到的串同样的字串。

(?#…) 注释；里面的内容会被忽略。

(?=…) 匹配 …的内容，但是并不消费样式的内容。这个叫做 *lookahead assertion*。比如，`Isaac (?=Asimov)` 匹配 `'Isaac '` 只有在后面是 `'Asimov'` 的时候。

(?!…) 匹配 …不符合的情况。这个叫 *negative lookahead assertion*（前视取反）。比如说，`Isaac (?!Asimov)` 只有后面 不是 `'Asimov'` 的时候才匹配 `'Isaac '` 。

(?<=…) 匹配字符串的当前位置，它的前面匹配 `...` 的内容到当前位置。这叫 *positive lookbehind assertion*（正向后视断定）。`(?<=abc)def` 会在 `'abcdef'` 中找到一个匹配，因为后视会往后看 3 个字符并检查是否包含匹配的样式。包含的匹配样式必须是定长的，意思就是 `abc` 或 `a|b` 是允许的，但是 `a*` 和 `a{3,4}` 不可以。注意以 positive lookbehind assertions 开始的样式，如 `(?<=abc)def` ，并不是从 a 开始搜索，而是从 d 往回看的。你可能更加愿意使用*`search()`* 函数，而不是*`match()`* 函数：

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

这个例子搜索一个跟随在连字符后的单词：

```
>>> m = re.search(r'(?<=-)\w+', 'spam-egg')
>>> m.group(0)
'egg'
```

3.5 版更變: 添加定长组合引用的支持。

(?<!…) 匹配当前位置之前不是 `...` 的样式。这个叫 *negative lookbehind assertion* （后视断定取非）。类似正向后视断定，包含的样式匹配必须是定长的。由 negative lookbehind assertion 开始的样式可以从字符串搜索开始的位置进行匹配。

(?(id/name)yes-pattern|no-pattern) 如果给定的 *id* 或 *name* 存在，将会尝试匹配 `yes-pattern` ，否则就尝试匹配 `no-pattern`，`no-pattern` 可选，也可以被忽略。比如，`(<)?(\w+@\w+(?:\.\w+)+)(?(1)>|$)` 是一个 email 样式匹配，将匹配 `'<user@host.com>'` 或 `'user@host.com'` ，但不会匹配 `'<user@host.com'` ，也不会匹配 `'user@host.com>'`。

由 `'\'` 和一个字符组成的特殊序列在以下列出。如果普通字符不是 ASCII 数位或者 ASCII 字母，那么正则样式将匹配第二个字符。比如，`\$` 匹配字符 `'$'`.

\number 匹配数字代表的组合。每个括号是一个组合，组合从 1 开始编号。比如 `(.+) \1` 匹配 `'the the'` 或者 `'55 55'`，但不会匹配 `'thethe'` (注意组合后面的空格)。这个特殊序列只能用于匹配前面 99 个

组合。如果 *number* 的第一个数位是 0，或者 *number* 是三个八进制数，它将不会被看作是一个组合，而是八进制的数字值。在 `'['` 和 `']'` 字符集合内，任何数字转义都被看作是字符。

\A 只匹配字符串开始。

\b 匹配空字符串，但只在单词开始或结尾的位置。一个单词被定义为一个单词字符的序列。注意，通常 `\b` 定义为 `\w` 和 `\W` 字符之间，或者 `\w` 和字符串开始/结尾的边界，意思就是 `r'\bfoo\b'` 匹配 `'foo'`, `'foo.'`, `'(foo)'`, `'bar foo baz'` 但不匹配 `'foobar'` 或者 `'foo3'`。

默认情况下，Unicode 字母和数字是在 Unicode 样式中使用的，但是可以用*ASCII* 标记来更改。如果*LOCALE* 标记被设置的话，词的边界是由当前语言区域设置决定的，`\b` 表示退格字符，以便与 Python 字符串文本兼容。

\B 匹配空字符串，但 不能在词的开头或者结尾。意思就是 `r'py\B'` 匹配 `'python'`, `'py3'`, `'py2'`, 但不匹配 `'py'`, `'py.'`, 或者 `'py!'`. `\B` 是 `\b` 的取非，所以 Unicode 样式的词语是由 Unicode 字母，数字或下划线构成的，虽然可以用*ASCII* 标志来改变。如果使用了*LOCALE* 标志，则词的边界由当前语言区域设置。

\d

对于 Unicode (str) 样式： 匹配任何 Unicode 十进制数（就是在 Unicode 字符目录 [Nd] 里的字符）。这包括了 `[0-9]`，和很多其他的数字字符。如果设置了*ASCII* 标志，就只匹配 `[0-9]`。

对于 8 位 (bytes) 样式： 匹配任何十进制数，就是 `[0-9]`。

\D 匹配任何非十进制数字的字符。就是 `\d` 取非。如果设置了*ASCII* 标志，就相当于 `[^0-9]`。

\s

对于 Unicode (str) 样式： 匹配任何 Unicode 空白字符（包括 `[ \t\n\r\f\v]`，还有很多其他字符，比如不同语言排版规则约定的不换行空格）。如果*ASCII* 被设置，就只匹配 `[ \t\n\r\f\v]`。

对于 8 位 (bytes) 样式： 匹配 ASCII 中的空白字符，就是 `[ \t\n\r\f\v]`。

\S 匹配任何非空白字符。就是 `\s` 取非。如果设置了*ASCII* 标志，就相当于 `[^ \t\n\r\f\v]`。

\w

对于 Unicode (str) 样式： 匹配 Unicode 词语的字符，包含了可以构成词语的绝大部分字符，也包括数字和下划线。如果设置了*ASCII* 标志，就只匹配 `[a-zA-Z0-9_]`。

对于 8 位 (bytes) 样式： 匹配 ASCII 字符中的数字和字母和下划线，就是 `[a-zA-Z0-9_]`。如果设置了*LOCALE* 标记，就匹配当前语言区域的数字和字母和下划线。

\W 匹配非单词字符的字符。这与 `\w` 正相反。如果使用了*ASCII* 旗标，这就等价于 `[^a-zA-Z0-9_]`。如果使用了*LOCALE* 旗标，则会匹配当前区域中既非字母数字也非下划线的字符。

\Z 只匹配字符串尾。

绝大部分 Python 的标准转义字符也被正则表达式分析器支持。:

```
\a      \b      \f      \n
\N      \r      \t      \u
\U      \v      \x      \\
```

（注意 `\b` 被用于表示词语的边界，它只在字符集合内表示退格，比如 `[\b]`。）

`'\u'`, `'\U'` 和 `'\N'` 转义序列只在 Unicode 模式中可被识别。在 bytes 模式中它们会导致错误。未知的 ASCII 字母转义序列保留在未来使用，会被当作错误来处理。

八进制转义包含为一个有限形式。如果首位数字是 0，或者有三个八进制数位，那么就认为它是八进制转义。其他的情况，就看作是组引用。对于字符串文本，八进制转义最多有三个数位长。

3.3 版更變: 增加了 `'\u'` 和 `'\U'` 转义序列。

3.6 版更變: 由 '\' 和一个 ASCII 字符组成的未知转义会被看成错误。

3.8 版更變: 添加了 '\N{name}' 转义序列。与在字符串字面值中一样，它扩展了命名 Unicode 字符 (例如 '\N{EM DASH}')。

6.2.2 模块内容

模块定义了几个函数、常量，和一个异常。有些函数是编译后的正则表达式方法的简化版本（少了一些特性）。重要的应用程序大多会在使用前先编译正则表达式。

3.6 版更變: 标志常量现在是 RegexFlag 类的实例，这个类是*enum.IntFlag* 的子类。

re.**compile**(*pattern*, *flags=0*)

将正则表达式的样式编译为一个*正则表达式对象* （正则对象），可以用于匹配，通过这个对象的方法*match()*, *search()* 以及其他如下描述。

这个表达式的行为可以通过指定 标记的值来改变。值可以是以下任意变量，可以通过位的 OR 操作来结合（| 操作符）。

序列

```
prog = re.compile(pattern)
result = prog.match(string)
```

等价于

```
result = re.match(pattern, string)
```

如果需要多次使用这个正则表达式的话，使用*re.compile()* 和保存这个正则对象以便复用，可以让程序更加高效。

備𝔽: 通过*re.compile()* 编译后的样式，和模块级的函数会被缓存，所以少数的正则表达式使用无需考虑编译的问题。

---

re.**A**

re.**ASCII**

让 \w, \W, \b, \B, \d, \D, \s 和 \S 只匹配 ASCII，而不是 Unicode。这只对 Unicode 样式有效，会被 byte 样式忽略。相当于前面语法中的内联标志 (?a) 。

注意，为了保持向后兼容，re.U 标记依然存在（还有他的同义 re.UNICODE 和嵌入形式 (?u) ），但是这些在 Python 3 是冗余的，因为默认字符串已经是 Unicode 了（并且 Unicode 匹配不允许 byte 出现）。

re.**DEBUG**

显示编译时的 debug 信息，没有内联标记。

re.**I**

re.**IGNORECASE**

进行忽略大小写匹配；表达式如 [A-Z] 也会匹配小写字符。Unicode 匹配（比如 Ü 匹配 ü）同样有用，除非设置了*re.ASCII* 标记来禁用非 ASCII 匹配。当前语言区域不会改变这个标记，除非设置了*re.LOCALE* 标记。这个相当于内联标记 (?i) 。

注意，当设置了*IGNORECASE* 标记，搜索 Unicode 样式 [a-z] 或 [A-Z] 的结合时，它将会匹配 52 个 ASCII 字符和 4 个额外的非 ASCII 字符：'İ' (U+0130, 拉丁大写的 I 带个点在上面), 'ı' (U+0131, 拉丁小写没有点的 I), 'ſ' (U+017F, 拉丁小写长 s) and 'K' (U+212A, 开尔文符号). 如果使用*ASCII* 标记，就只匹配'a' 到'z' 和'A' 到'Z' 。

re.**L**

re.**LOCALE**
由当前语言区域决定 \w, \W, \b, \B 和大小写敏感匹配。这个标记只能对 byte 样式有效。这个标记不推荐使用，因为语言区域机制很不可靠，它一次只能处理一个" 习惯"，而且只对 8 位字节有效。Unicode 匹配在 Python 3 里默认启用，并可以处理不同语言。这个对应内联标记（?L）。

3.6 版更變: *re.LOCALE* 只能用于 byte 样式，而且不能和*re.ASCII* 一起用。

3.7 版更變: 设置了*re.LOCALE* 标记的编译正则对象不再在编译时依赖语言区域设置。语言区域设置只在匹配的时候影响其结果。

re.**M**
re.**MULTILINE**
设置以后，样式字符 '^' 匹配字符串的开始，和每一行的开始（换行符后面紧跟的符号）；样式字符 '$' 匹配字符串尾，和每一行的结尾（换行符前面那个符号）。默认情况下，' ^' 匹配字符串头，'$' 匹配字符串尾。对应内联标记（?m）。

re.**S**
re.**DOTALL**
让 '.' 特殊字符匹配任何字符，包括换行符；如果没有这个标记，'.' 就匹配 除了换行符的其他任意字符。对应内联标记（?s）。

re.**X**
re.**VERBOSE**
这个标记允许你编写更具可读性更友好的正则表达式。通过分段和添加注释。空白符号会被忽略，除非在一个字符集合当中或者由反斜杠转义，或者在 *?, (?: or (?P<…> 分组之内。当一个行内有 # 不在字符集和转义序列，那么它之后的所有字符都是注释。

意思就是下面两个正则表达式等价地匹配一个十进制数字：

```
a = re.compile(r"""\d +  # the integral part
                   \.    # the decimal point
                   \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

对应内联标记（?x）。

re.**search**(*pattern*, *string*, *flags=0*)
扫描整个 字 符 串找到匹配样式的第一个位置，并返回一个相应的匹配对象。如果没有匹配，就返回一个 None；注意这和找到一个零长度匹配是不同的。

re.**match**(*pattern*, *string*, *flags=0*)
如果 *string* 开始的 0 或者多个字符匹配到了正则表达式样式，就返回一个相应的匹配对象 。如果没有匹配，就返回 None；注意它跟零长度匹配是不同的。

注意即便是*MULTILINE* 多行模式，*re.match()* 也只匹配字符串的开始位置，而不匹配每行开始。

如果你想定位 *string* 的任何位置，使用*search()* 来替代（也可参考*search() vs. match()* ）

re.**fullmatch**(*pattern*, *string*, *flags=0*)
如果整个 *string* 匹配到正则表达式样式，就返回一个相应的匹配对象 。否则就返回一个 None；注意这跟零长度匹配是不同的。

3.4 版新加入.

re.**split**(*pattern*, *string*, *maxsplit=0*, *flags=0*)
用 *pattern* 分开 *string* 。如果在 *pattern* 中捕获到括号，那么所有的组里的文字也会包含在列表里。如果 *maxsplit* 非零，最多进行 *maxsplit* 次分隔，剩下的字符全部返回到列表的最后一个元素。

```
>>> re.split(r'\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'(\W+)', 'Words, words, words.')
```

(下页继续)

(繼續上一頁)

```
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split(r'\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

如果分隔符里有捕获组合，并且匹配到字符串的开始，那么结果将会以一个空字符串开始。对于结尾也是一样

```
>>> re.split(r'(\W+)', '...words, words...')
['', '...', 'words', ', ', 'words', '...', '']
```

这样的话，分隔组将会出现在结果列表中同样的位置。

样式的空匹配仅在与前一个空匹配不相邻时才会拆分字符串。

```
>>> re.split(r'\b', 'Words, words, words.')
['', 'Words', ', ', 'words', ', ', 'words', '.']
>>> re.split(r'\W*', '...words...')
['', '', 'w', 'o', 'r', 'd', 's', '', '']
>>> re.split(r'(\W*)', '...words...')
['', '...', '', '', 'w', '', 'o', '', 'r', '', 'd', '', 's', '...', '', '', '']
```

3.1 版更變: 增加了可选标记参数。

3.7 版更變: 增加了空字符串的样式分隔。

re.**findall**(*pattern*, *string*, *flags=0*)

Return all non-overlapping matches of *pattern* in *string*, as a list of strings or tuples. The *string* is scanned left-to-right, and matches are returned in the order found. Empty matches are included in the result.

The result depends on the number of capturing groups in the pattern. If there are no groups, return a list of strings matching the whole pattern. If there is exactly one group, return a list of strings matching that group. If multiple groups are present, return a list of tuples of strings matching the groups. Non-capturing groups do not affect the form of the result.

```
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.findall(r'(\w+)=(\d+)', 'set width=20 and height=10')
[('width', '20'), ('height', '10')]
```

3.7 版更變: 非空匹配现在可以在前一个空匹配之后出现了。

re.**finditer**(*pattern*, *string*, *flags=0*)

pattern 在 *string* 里所有的非重复匹配，返回为一个迭代器*iterator* 保存了匹配对象 。*string* 从左到右扫描，匹配按顺序排列。空匹配也包含在结果里。

3.7 版更變: 非空匹配现在可以在前一个空匹配之后出现了。

re.**sub**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)

返回通过使用 *repl* 替换在 *string* 最左边非重叠出现的 *pattern* 而获得的字符串。如果样式没有找到，则不加改变地返回 *string*。*repl* 可以是字符串或函数；如为字符串，则其中任何反斜杠转义序列都会被处理。也就是说，\n 会被转换为一个换行符，\r 会被转换为一个回车符，依此类推。未知的 ASCII 字符转义序列保留在未来使用，会被当作错误来处理。其他未知转义序列例如 \& 会保持原样。向后引用像是 \6 会用样式中第 6 组所匹配到的子字符串来替换。例如:

```
>>> re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):',
...        r'static PyObject*\npy_\1(void)\n{',
```

(下页继续)

(繼續上一頁)

```
...         'def myfunc():')
'static PyObject*\npy_myfunc(void)\n{'
```

如果 *repl* 是一个函数，那它会对每个非重复的 *pattern* 的情况调用。这个函数只能有一个匹配对象 参数，并返回一个替换后的字符串。比如

```
>>> def dashrepl(matchobj):
...     if matchobj.group(0) == '-': return ' '
...     else: return '-'
>>> re.sub('-{1,2}', dashrepl, 'pro----gram-files')
'pro--gram files'
>>> re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam', flags=re.IGNORECASE)
'Baked Beans & Spam'
```

样式可以是一个字符串或者一个样式对象 。

可选参数 *count* 是要替换的最大次数；*count* 必须是非负整数。如果省略这个参数或设为 0，所有的匹配都会被替换。样式的空匹配仅在与前一个空匹配不相邻时才会被替换，所以 `sub('x*', '-', 'abxd')` 返回 `'-a-b--d-'` 。

在字符串类型的 *repl* 参数里，如上所述的转义和向后引用中，`\g<name>` 会使用命名组合 `name`，（在 `(?P<name>…)` 语法中定义）`\g<number>` 会使用数字组；`\g<2>` 就是 `\2`，但它避免了二义性，如 `\g<2>0`。`\20` 就会被解释为组 20，而不是组 2 后面跟随一个字符 `'0'`。向后引用 `\g<0>` 把 *pattern* 作为一整个组进行引用。

3.1 版更變: 增加了可选标记参数。

3.5 版更變: 不匹配的组合替换为空字符串。

3.6 版更變: *pattern* 中的未知转义（由 `'\'` 和一个 ASCII 字符组成）被视为错误。

3.7 版更變: *repl* 中的未知转义（由 `'\'` 和一个 ASCII 字符组成）被视为错误。

3.7 版更變: 样式中的空匹配相邻接时会被替换。

`re.`**`subn`**(*pattern*, *repl*, *string*, *count=0*, *flags=0*)
行为与*sub()* 相同，但是返回一个元组（字符串， 替换次数）.

3.1 版更變: 增加了可选标记参数。

3.5 版更變: 不匹配的组合替换为空字符串。

`re.`**`escape`**(*pattern*)
转义 *pattern* 中的特殊字符。如果你想对任意可能包含正则表达式元字符的文本字符串进行匹配，它就是有用的。比如

```
>>> print(re.escape('https://www.python.org'))
https://www\.python\.org

>>> legal_chars = string.ascii_lowercase + string.digits + "!#$%&'*+-.^_`|~:"
>>> print('[%s]+' % re.escape(legal_chars))
[abcdefghijklmnopqrstuvwxyz0123456789!\#\$%\&'\*\+\-\.\^_`\|\~:]+

>>> operators = ['+', '-', '*', '/', '**']
>>> print('|'.join(map(re.escape, sorted(operators, reverse=True))))
/|\-|\+|\*\*|\*
```

这个函数不能被用于*sub()* 和*subn()* 的替换字符串，只有反斜杠应该被转义。例如:

```
>>> digits_re = r'\d+'
>>> sample = '/usr/sbin/sendmail - 0 errors, 12 warnings'
>>> print(re.sub(digits_re, digits_re.replace('\\', r'\\'), sample))
/usr/sbin/sendmail - \d+ errors, \d+ warnings
```

3.3 版更變: `'_'` 不再被转义。

3.7 版更變: 只有在正则表达式中具有特殊含义的字符才会被转义。因此，`'!'`, `'"'`, `'%'`, `"'"`, `','`, `'/'`, `':'`, `';'`, `'<'`, `'='`, `'>'`, `'@'` 和 `"`"` 将不再会被转义。

`re.purge()`
清除正则表达式的缓存。

exception `re.error`(*msg*, *pattern=None*, *pos=None*)
当传递给函数的正则表达式不合法（比如括号不匹配），或者在编译或匹配过程中出现其他错误时，会引发异常。所给字符串不匹配所给模式不会引发异常。异常实例有以下附加属性:

msg
未格式化的错误消息。

pattern
正则表达式的模式串。

pos
编译失败的 *pattern* 的位置索引（可以是 `None` ）。

lineno
对应 *pos* (可以是 `None`) 的行号。

colno
对应 *pos* (可以是 `None`) 的列号。

3.5 版更變: 增加了额外的属性。

6.2.3 正则表达式对象（正则对象）

编译后的正则表达式对象支持以下方法和属性:

`Pattern.search`(*string*[, *pos*[, *endpos*]])
扫描整个 *string* 寻找第一个匹配的位置，并返回一个相应的匹配对象。如果没有匹配，就返回 `None`；注意它和零长度匹配是不同的。

可选的第二个参数 *pos* 给出了字符串中开始搜索的位置索引；默认为 `0`，它不完全等价于字符串切片；`'^'` 样式字符匹配字符串真正的开头，和换行符后面的第一个字符，但不会匹配索引规定开始的位置。

可选参数 *endpos* 限定了字符串搜索的结束；它假定字符串长度到 *endpos* ，所以只有从 `pos` 到 `endpos - 1` 的字符会被匹配。如果 *endpos* 小于 *pos*，就不会有匹配产生；另外，如果 *rx* 是一个编译后的正则对象，`rx.search(string, 0, 50)` 等价于 `rx.search(string[:50], 0)`。

```
>>> pattern = re.compile("d")
>>> pattern.search("dog")     # Match at index 0
<re.Match object; span=(0, 1), match='d'>
>>> pattern.search("dog", 1)  # No match; search doesn't include the "d"
```

`Pattern.match`(*string*[, *pos*[, *endpos*]])
如果 *string* 的 *开始位置*能够找到这个正则样式的任意个匹配，就返回一个相应的匹配对象。如果不匹配，就返回 `None`；注意它与零长度匹配是不同的。

可选参数 *pos* 和 *endpos* 与 *search()* 含义相同。

```
>>> pattern = re.compile("o")
>>> pattern.match("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.match("dog", 1)   # Match as "o" is the 2nd character of "dog".
<re.Match object; span=(1, 2), match='o'>
```

如果你想定位匹配在 *string* 中的位置，使用`search()`来替代（另参考*search() vs. match()*）。

Pattern.**fullmatch**(*string*[, *pos*[, *endpos*]])
如果整个 *string* 匹配这个正则表达式，就返回一个相应的匹配对象。否则就返回 None；注意跟零长度匹配是不同的。

可选参数 *pos* 和 *endpos* 与`search()`含义相同。

```
>>> pattern = re.compile("o[gh]")
>>> pattern.fullmatch("dog")      # No match as "o" is not at the start of "dog".
>>> pattern.fullmatch("ogre")     # No match as not the full string matches.
>>> pattern.fullmatch("doggie", 1, 3)   # Matches within given limits.
<re.Match object; span=(1, 3), match='og'>
```

3.4 版新加入.

Pattern.**split**(*string*, *maxsplit=0*)
等价于`split()`函数，使用了编译后的样式。

Pattern.**findall**(*string*[, *pos*[, *endpos*]])
类似函数`findall()`，使用了编译后样式，但也可以接收可选参数 *pos* 和 *endpos*，限制搜索范围，就像`search()`。

Pattern.**finditer**(*string*[, *pos*[, *endpos*]])
类似函数`finditer()`，使用了编译后样式，但也可以接收可选参数 *pos* 和 *endpos*，限制搜索范围，就像`search()`。

Pattern.**sub**(*repl*, *string*, *count=0*)
等价于`sub()`函数，使用了编译后的样式。

Pattern.**subn**(*repl*, *string*, *count=0*)
等价于`subn()`函数，使用了编译后的样式。

Pattern.**flags**
正则匹配标记。这是可以传递给`compile()`的参数，任何 (?…) 内联标记，隐性标记比如 UNICODE 的结合。

Pattern.**groups**
捕获到的模式串中组的数量。

Pattern.**groupindex**
映射由 (?P<id>) 定义的命名符号组合和数字组合的字典。如果没有符号组，那字典就是空的。

Pattern.**pattern**
编译对象的原始样式字符串。

3.7 版更變: 添加`copy.copy()`和`copy.deepcopy()`函数的支持。编译后的正则表达式对象被认为是原子性的。

6.2.4 匹配对象

匹配对象总是有一个布尔值 `True`。如果没有匹配的话*match()* 和*search()* 返回 `None` 所以你可以简单的用 `if` 语句来判断是否匹配

```
match = re.search(pattern, string)
if match:
    process(match)
```

匹配对象支持以下方法和属性:

`Match.`**`expand`**`(`*template*`)`

对 *template* 进行反斜杠转义替换并且返回，就像*sub()* 方法中一样。转义如同 `\n` 被转换成合适的字符，数字引用 (`\1`, `\2`) 和命名组合 (`\g<1>`, `\g<name>`) 替换为相应组合的内容。

3.5 版更變: 不匹配的组合替换为空字符串。

`Match.`**`group`**`(`[*group1*, ...]`)`

返回一个或者多个匹配的子组。如果只有一个参数，结果就是一个字符串，如果有多个参数，结果就是一个元组（每个参数对应一个项），如果没有参数，组 1 默认到 0（整个匹配都被返回）。如果一个组 N 参数值为 0，相应的返回值就是整个匹配字符串；如果它是一个范围 [1..99]，结果就是相应的括号组字符串。如果一个组号是负数，或者大于样式中定义的组数，就引发一个*IndexError* 异常。如果一个组包含在样式的一部分，并被匹配多次，就返回最后一个匹配。:

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m.group(0)       # The entire match
'Isaac Newton'
>>> m.group(1)       # The first parenthesized subgroup.
'Isaac'
>>> m.group(2)       # The second parenthesized subgroup.
'Newton'
>>> m.group(1, 2)    # Multiple arguments give us a tuple.
('Isaac', 'Newton')
```

如果正则表达式使用了 `(?P<name>...)` 语法，*groupN* 参数就也可能是命名组合的名字。如果一个字符串参数在样式中未定义为组合名，就引发一个*IndexError* 异常。

一个相对复杂的例子

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.group('first_name')
'Malcolm'
>>> m.group('last_name')
'Reynolds'
```

命名组合同样可以通过索引值引用

```
>>> m.group(1)
'Malcolm'
>>> m.group(2)
'Reynolds'
```

如果一个组匹配成功多次，就只返回最后一个匹配

```
>>> m = re.match(r"(..)+", "a1b2c3")  # Matches 3 times.
>>> m.group(1)                        # Returns only the last match.
'c3'
```

Match.**__getitem__**(*g*)
这个等价于 m.group(g)。这允许更方便的引用一个匹配

```
>>> m = re.match(r"(\w+) (\w+)", "Isaac Newton, physicist")
>>> m[0]       # The entire match
'Isaac Newton'
>>> m[1]       # The first parenthesized subgroup.
'Isaac'
>>> m[2]       # The second parenthesized subgroup.
'Newton'
```

3.6 版新加入.

Match.**groups**(*default=None*)
返回一个元组，包含所有匹配的子组，在样式中出现的从 1 到任意多的组合。*default* 参数用于不参与匹配的情况，默认为 None。

例如

```
>>> m = re.match(r"(\d+)\.(\d+)", "24.1632")
>>> m.groups()
('24', '1632')
```

如果我们使小数点可选，那么不是所有的组都会参与到匹配当中。这些组合默认会返回一个 None，除非指定了 *default* 参数。

```
>>> m = re.match(r"(\d+)\.?(\d+)?", "24")
>>> m.groups()      # Second group defaults to None.
('24', None)
>>> m.groups('0')   # Now, the second group defaults to '0'.
('24', '0')
```

Match.**groupdict**(*default=None*)
返回一个字典，包含了所有的 命名子组。key 就是组名。*default* 参数用于不参与匹配的组合；默认为 None。例如

```
>>> m = re.match(r"(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
>>> m.groupdict()
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

Match.**start**([*group*])
Match.**end**([*group*])
返回 *group* 匹配到的字串的开始和结束标号。*group* 默认为 0（意思是整个匹配的子串）。如果 *group* 存在，但未产生匹配，就返回 -1 。对于一个匹配对象 *m*，和一个未参与匹配的组 *g* ，组 *g* (等价于 m.group(g)) 产生的匹配是

```
m.string[m.start(g):m.end(g)]
```

注意 m.start(group) 将会等于 m.end(group) ，如果 *group* 匹配一个空字符串的话。比如，在 m = re.search('b(c?)', 'cba') 之后，m.start(0) 为 1, m.end(0) 为 2, m.start(1) 和 m.end(1) 都是 2, m.start(2) 引发一个*IndexError* 异常。

这个例子会从 email 地址中移除掉 *remove_this*

```
>>> email = "tony@tiremove_thisger.net"
>>> m = re.search("remove_this", email)
>>> email[:m.start()] + email[m.end():]
'tony@tiger.net'
```

Match.**span**([*group*])
对于一个匹配 *m*，返回一个二元组 (m.start(group), m.end(group))。注意如果 *group* 没有在这个匹配中，就返回 (-1, -1)。*group* 默认为 0，就是整个匹配。

Match.**pos**
pos 的值，会传递给*search()* 或*match()* 的方法 a 正则对象 。这个是正则引擎开始在字符串搜索一个匹配的索引位置。

Match.**endpos**
endpos 的值，会传递给*search()* 或*match()* 的方法 a 正则对象 。这个是正则引擎停止在字符串搜索一个匹配的索引位置。

Match.**lastindex**
捕获组的最后一个匹配的整数索引值，或者 None 如果没有匹配产生的话。比如，对于字符串 'ab'，表达式 (a)b, ((a)(b)), 和 ((ab)) 将得到 lastindex == 1，而 (a)(b) 会得到 lastindex == 2。

Match.**lastgroup**
最后一个匹配的命名组名字，或者 None 如果没有产生匹配的话。

Match.**re**
返回产生这个实例的正则对象，这个实例是由正则对象的*match()* 或*search()* 方法产生的。

Match.**string**
传递到*match()* 或*search()* 的字符串。

3.7 版更變: 添加了对*copy.copy()* 和*copy.deepcopy()* 的支持。匹配对象被看作是原子性的。

6.2.5 正则表达式例子

检查对子

在这个例子里，我们使用以下辅助函数来更好地显示匹配对象:

```
def displaymatch(match):
    if match is None:
        return None
    return '<Match: %r, groups=%r>' % (match.group(), match.groups())
```

假设你在写一个扑克程序，一个玩家的一手牌为五个字符的串，每个字符表示一张牌，"a" 就是 A, "k" K，"q" Q, "j" J, "t" 为 10, "2" 到"9" 表示 2 到 9。

要看给定的字符串是否有效，我们可以按照以下步骤

```
>>> valid = re.compile(r"^[a2-9tjqk]{5}$")
>>> displaymatch(valid.match("akt5q"))  # Valid.
"<Match: 'akt5q', groups=()>"
>>> displaymatch(valid.match("akt5e"))  # Invalid.
>>> displaymatch(valid.match("akt"))    # Invalid.
>>> displaymatch(valid.match("727ak"))  # Valid.
"<Match: '727ak', groups=()>"
```

最后一手牌，"727ak"，包含了一个对子，或者两张同样数值的牌。要用正则表达式匹配它，应该使用向后引用如下

```
>>> pair = re.compile(r".*(.).*\1")
>>> displaymatch(pair.match("717ak"))     # Pair of 7s.
"<Match: '717', groups=('7',)>"
```

(下页继续)

(繼續上一頁)

```
>>> displaymatch(pair.match("718ak"))      # No pairs.
>>> displaymatch(pair.match("354aa"))      # Pair of aces.
"<Match: '354aa', groups=('a',)>"
```

要找出对子由什么牌组成，开发者可以按照下面的方式来使用匹配对象的*group()* 方法:

```
>>> pair = re.compile(r".*(.).*\1")
>>> pair.match("717ak").group(1)
'7'

# Error because re.match() returns None, which doesn't have a group() method:
>>> pair.match("718ak").group(1)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    re.match(r".*(.).*\1", "718ak").group(1)
AttributeError: 'NoneType' object has no attribute 'group'

>>> pair.match("354aa").group(1)
'a'
```

模拟 scanf()

Python 目前没有一个类似 c 函数 `scanf()` 的替代品。正则表达式通常比 `scanf()` 格式字符串要更强大一些，但也带来更多复杂性。下面的表格提供了 `scanf()` 格式符和正则表达式大致相同的映射。

scanf() 格式符	正则表达式
%c	.
%5c	.{5}
%d	[-+]?\d+
%e, %E, %f, %g	[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?
%i	[-+]?(0[xX][\dA-Fa-f]+|0[0-7]*|\d+)
%o	[-+]?[0-7]+
%s	\S+
%u	\d+
%x, %X	[-+]?(0[xX])?[\dA-Fa-f]+

从文件名和数字提取字符串

```
/usr/sbin/sendmail - 0 errors, 4 warnings
```

你可以使用 `scanf()` 格式化

```
%s - %d errors, %d warnings
```

等价的正则表达式是:

```
(\S+) - (\d+) errors, (\d+) warnings
```

search() vs. match()

Python 提供了两种不同的操作：基于*re.match()* 检查字符串开头，或者*re.search()* 检查字符串的任意位置（默认 Perl 中的行为）。

例如

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("c", "abcdef")   # Match
<re.Match object; span=(2, 3), match='c'>
```

在*search()* 中，可以用 '^' 作为开始来限制匹配到字符串的首位

```
>>> re.match("c", "abcdef")    # No match
>>> re.search("^c", "abcdef")  # No match
>>> re.search("^a", "abcdef")  # Match
<re.Match object; span=(0, 1), match='a'>
```

注意*MULTILINE* 多行模式中函数*match()* 只匹配字符串的开始，但使用*search()* 和以 '^' 开始的正则表达式会匹配每行的开始

```
>>> re.match('X', 'A\nB\nX', re.MULTILINE)  # No match
>>> re.search('^X', 'A\nB\nX', re.MULTILINE)  # Match
<re.Match object; span=(4, 5), match='X'>
```

制作一个电话本

split() 将字符串用参数传递的样式分隔开。这个方法对于转换文本数据到易读而且容易修改的数据结构，是很有用的，如下面的例子证明。

首先，这里是输入。它通常来自一个文件，这里我们使用三重引号字符串语法

```
>>> text = """Ross McFluff: 834.345.1254 155 Elm Street
...
... Ronald Heathmore: 892.345.3428 436 Finley Avenue
... Frank Burger: 925.541.7625 662 South Dogwood Way
...
...
... Heather Albrecht: 548.326.4584 919 Park Place"""
```

条目用一个或者多个换行符分开。现在我们将字符串转换为一个列表，每个非空行都有一个条目:

```
>>> entries = re.split("\n+", text)
>>> entries
['Ross McFluff: 834.345.1254 155 Elm Street',
'Ronald Heathmore: 892.345.3428 436 Finley Avenue',
'Frank Burger: 925.541.7625 662 South Dogwood Way',
'Heather Albrecht: 548.326.4584 919 Park Place']
```

最终，将每个条目分割为一个由名字、姓氏、电话号码和地址组成的列表。我们为*split()* 使用了 maxsplit 形参，因为地址中包含有被我们作为分割模式的空格符:

```
>>> [re.split(":? ", entry, 3) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]
```

:? 样式匹配姓后面的冒号，因此它不出现在结果列表中。如果 maxsplit 设置为 4 ，我们还可以从地址中获取到房间号:

```
>>> [re.split(":? ", entry, 4) for entry in entries]
[['Ross', 'McFluff', '834.345.1254', '155', 'Elm Street'],
['Ronald', 'Heathmore', '892.345.3428', '436', 'Finley Avenue'],
['Frank', 'Burger', '925.541.7625', '662', 'South Dogwood Way'],
['Heather', 'Albrecht', '548.326.4584', '919', 'Park Place']]
```

文字整理

sub() 替换字符串中出现的样式的每一个实例。这个例子证明了使用*sub()* 来整理文字，或者随机化每个字符的位置，除了首位和末尾字符

```
>>> def repl(m):
...     inner_word = list(m.group(2))
...     random.shuffle(inner_word)
...     return m.group(1) + "".join(inner_word) + m.group(3)
>>> text = "Professor Abdolmalek, please report your absences promptly."
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Poefsrosr Aealmlobdk, pslaee reorpt your abnseces plmrptoy.'
>>> re.sub(r"(\w)(\w+)(\w)", repl, text)
'Pofsroser Aodlambelk, plasee reoprt yuor asnebces potlmrpy.'
```

查找所有副词

findall() 匹配样式 所有的出现，不仅是像*search()* 中的第一个匹配。比如，如果一个作者希望找到文字中的所有副词，他可能会按照以下方法用*findall()*

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> re.findall(r"\w+ly\b", text)
['carefully', 'quickly']
```

查找所有的副词及其位置

如果需要匹配样式的更多信息，*finditer()* 可以起到作用，它提供了匹配对象 作为返回值，而不是字符串。继续上面的例子，如果一个作者希望找到所有副词和它的位置，可以按照下面方法使用*finditer()*

```
>>> text = "He was carefully disguised but captured quickly by police."
>>> for m in re.finditer(r"\w+ly\b", text):
...     print('%02d-%02d: %s' % (m.start(), m.end(), m.group(0)))
07-16: carefully
40-47: quickly
```

原始字符串标记

原始字符串记法 (r"text") 保持正则表达式正常。否则，每个正则式里的反斜杠 ('\') 都必须前缀一个反斜杠来转义。比如，下面两行代码功能就是完全一致的

```
>>> re.match(r"\W(.)\1\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
>>> re.match("\\W(.)\\1\\W", " ff ")
<re.Match object; span=(0, 4), match=' ff '>
```

当需要匹配一个字符反斜杠，它必须在正则表达式中转义。在原始字符串记法，就是 r"\\"。否则就必须用 "\\\\"，来表示同样的意思

```
>>> re.match(r"\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
>>> re.match("\\\\", r"\\")
<re.Match object; span=(0, 1), match='\\'>
```

写一个词法分析器

一个 词法器或词法分析器 分析字符串，并分类成目录组。这是写一个编译器或解释器的第一步。

文字目录是由正则表达式指定的。这个技术是通过将这些样式合并为一个主正则式，并且循环匹配来实现的

```
from typing import NamedTuple
import re

class Token(NamedTuple):
    type: str
    value: str
    line: int
    column: int

def tokenize(code):
    keywords = {'IF', 'THEN', 'ENDIF', 'FOR', 'NEXT', 'GOSUB', 'RETURN'}
    token_specification = [
        ('NUMBER',   r'\d+(\.\d*)?'),  # Integer or decimal number
        ('ASSIGN',   r':='),           # Assignment operator
        ('END',      r';'),            # Statement terminator
        ('ID',       r'[A-Za-z]+'),    # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('MISMATCH', r'.'),            # Any other character
    ]
    tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in token_specification)
    line_num = 1
    line_start = 0
    for mo in re.finditer(tok_regex, code):
        kind = mo.lastgroup
        value = mo.group()
        column = mo.start() - line_start
        if kind == 'NUMBER':
            value = float(value) if '.' in value else int(value)
        elif kind == 'ID' and value in keywords:
            kind = value
```

(下页继续)

(繼續上一頁)

```
        elif kind == 'NEWLINE':
            line_start = mo.end()
            line_num += 1
            continue
        elif kind == 'SKIP':
            continue
        elif kind == 'MISMATCH':
            raise RuntimeError(f'{value!r} unexpected on line {line_num}')
        yield Token(kind, value, line_num, column)

statements = '''
    IF quantity THEN
        total := total + price * quantity;
        tax := price * 0.05;
    ENDIF;
'''

for token in tokenize(statements):
    print(token)
```

该词法器产生以下的输出

```
Token(type='IF', value='IF', line=2, column=4)
Token(type='ID', value='quantity', line=2, column=7)
Token(type='THEN', value='THEN', line=2, column=16)
Token(type='ID', value='total', line=3, column=8)
Token(type='ASSIGN', value=':=', line=3, column=14)
Token(type='ID', value='total', line=3, column=17)
Token(type='OP', value='+', line=3, column=23)
Token(type='ID', value='price', line=3, column=25)
Token(type='OP', value='*', line=3, column=31)
Token(type='ID', value='quantity', line=3, column=33)
Token(type='END', value=';', line=3, column=41)
Token(type='ID', value='tax', line=4, column=8)
Token(type='ASSIGN', value=':=', line=4, column=12)
Token(type='ID', value='price', line=4, column=15)
Token(type='OP', value='*', line=4, column=21)
Token(type='NUMBER', value=0.05, line=4, column=23)
Token(type='END', value=';', line=4, column=27)
Token(type='ENDIF', value='ENDIF', line=5, column=4)
Token(type='END', value=';', line=5, column=9)
```

6.3 difflib --- 计算差异的辅助工具

源代码: Lib/difflib.py

此模块提供用于比较序列的类和函数。例如，它可被用于比较文件，并可产生多种格式的不同文件差异信息，包括 HTML 和上下文以及统一的 diff 数据。有关比较目录和文件，另请参阅*filecmp* 模块。

class difflib.**SequenceMatcher**

这是一个灵活的类，可用于比较任何类型的序列对，只要序列元素为*hashable* 对象。其基本算法要早于由 Ratcliff 和 Obershelp 于 1980 年代末期发表并以“格式塔模式匹配”的夸张名称命名的算法，并且更加有趣一些。其思路是找到不包含“垃圾”元素的最长连续匹配子序列；所谓“垃圾”元素是指其在某

种意义上没有价值，例如空白行或空白符。（处理垃圾元素是对 Ratcliff 和 Obershelp 算法的一个扩展。）然后同样的思路将递归地应用于匹配序列的左右序列片段。这并不能产生最小编辑序列，但确实能产生在人们看来“正确”的匹配。

耗时: 基本 Ratcliff-Obershelp 算法在最坏情况下为立方时间而在一般情况下为平方时间。*SequenceMatcher* 在最坏情况下为平方时间而在一般情况下的行为受到序列中有多少相同元素这一因素的微妙影响；在最佳情况下则为线性时间。

自动垃圾启发式计算: *SequenceMatcher* 支持使用启发式计算来自动将特定序列项视为垃圾。这种启发式计算会统计每个单独项在序列中出现的次数。如果某一项（在第一项之后）的重复次数超过序列长度的 1% 并且序列长度至少有 200 项，该项会被标记为“热门”并被视为序列匹配中的垃圾。这种启发式计算可以通过在创建*SequenceMatcher* 时将 `autojunk` 参数设为 `False` 来关闭。

3.2 版新加入: *autojunk* 形参。

class difflib.**Differ**

这个类的作用是比较由文本行组成的序列，并产生可供人阅读的差异或增量信息。Differ 统一使用*SequenceMatcher* 来完成行序列的比较以及相似（接近匹配）行内部字符序列的比较。

Differ 增量的每一行均以双字母代码打头：

双字母代码	意义
`'- '`	行为序列 1 所独有
`'+ '`	行为序列 2 所独有
`'  '`	行在两序列中相同
`'? '`	行不存在于任一输入序列

以’?’ 打头的行尝试将视线引至行以外而不存在于任一输入序列的差异。如果序列包含制表符则这些行可能会令人感到迷惑。

class difflib.**HtmlDiff**

这个类可用于创建 HTML 表格（或包含表格的完整 HTML 文件）以并排地逐行显示文本比较，行间与行外的更改将突出显示。此表格可以基于完全或上下文差异模式来生成。

这个类的构造函数：

__init__(*tabsize=8*, *wrapcolumn=None*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

初始化*HtmlDiff* 的实例。

tabsize 是一个可选关键字参数，指定制表位的间隔，默认值为 `8`。

wrapcolumn 是一个可选关键字参数，指定行文本自动打断并换行的列位置，默认值为 `None` 表示不自动换行。

linejunk 和 *charjunk* 均是可选关键字参数，会传入*ndiff()* (被*HtmlDiff* 用来生成并排显示的 HTML 差异)。请参阅*ndiff()* 文档了解参数默认值及其说明。

下列是公开的方法

make_file(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*, ***, *charset='utf-8'*)

比较 *fromlines* 和 *tolines* (字符串列表) 并返回一个字符串，表示一个完整 HTML 文件，其中包含各行差异的表格，行间与行外的更改将突出显示。

fromdesc 和 *todesc* 均是可选关键字参数，指定来源/目标文件的列标题字符串（默认均为空白字符串）。

context 和 *numlines* 均是可选关键字参数。当只要显示上下文差异时就将 *context* 设为 `True`，否则默认值 `False` 为显示完整文件。*numlines* 默认为 `5`。当 *context* 为 `True` 时 *numlines* 将控制围绕突出显示差异部分的上下文行数。当 *context* 为 `False` 时 *numlines* 将控制在使用”next” 超链接时突出显示差异部分之前所显示的行数（设为零则会导致”next” 超链接将下一个突出显示差异部分放在浏览器顶端，不添加任何前导上下文）。

> **備🄴:** *fromdesc* 和 *todesc* 会被当作未转义的 HTML 来解读，当接收不可信来源的输入时应该适当地进行转义。

3.5 版更變: 增加了 *charset* 关键字参数。HTML 文档的默认字符集从 `'ISO-8859-1'` 更改为 `'utf-8'`。

make_table(*fromlines*, *tolines*, *fromdesc=''*, *todesc=''*, *context=False*, *numlines=5*)
比较 *fromlines* 和 *tolines* (字符串列表) 并返回一个字符串，表示一个包含各行差异的完整 HTML 表格，行间与行外的更改将突出显示。

此方法的参数与*make_file()* 方法的相同。

`Tools/scripts/diff.py` 是这个类的命令行前端，其中包含一个很好的使用示例。

difflib.**context_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)
比较 *a* 和 *b* (字符串列表)；返回上下文差异格式的增量信息 (一个产生增量行的*generator*)。

所谓上下文差异是一种只显示有更改的行再加几个上下文行的紧凑形式。更改被显示为之前/之后的样式。上下文行数由 *n* 设定，默认为三行。

默认情况下，差异控制行（以 `***` or `---` 表示）是通过末尾换行符来创建的。这样做的好处是从*io.IOBase.readlines()* 创建的输入将得到适用于*io.IOBase.writelines()* 的差异信息，因为输入和输出都带有末尾换行符。

对于没有末尾换行符的输入，应将 *lineterm* 参数设为 `""`，这样输出内容将统一不带换行符。

上下文差异格式通常带有一个记录文件名和修改时间的标头。这些信息的部分或全部可以使用字符串 *fromfile*, *tofile*, *fromfiledate* 和 *tofiledate* 来指定。修改时间通常以 ISO 8601 格式表示。如果未指定，这些字符串默认为空。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(context_diff(s1, s2, fromfile='before.py', tofile=
↪'after.py'))
*** before.py
--- after.py
***************
*** 1,4 ****
! bacon
! eggs
! ham
  guido
--- 1,4 ----
! python
! eggy
! hamster
  guido
```

请参阅*difflib 的命令行接口* 获取更详细的示例。

difflib.**get_close_matches**(*word*, *possibilities*, *n=3*, *cutoff=0.6*)
返回由最佳“近似”匹配构成的列表。*word* 为一个指定目标近似匹配的序列（通常为字符串），*possibilities* 为一个由用于匹配 *word* 的序列构成的列表（通常为字符串列表）。

可选参数 *n* (默认为 `3`) 指定最多返回多少个近似匹配；*n* 必须大于 `0`.

可选参数 *cutoff* (默认为 `0.6`) 是一个 [0, 1] 范围内的浮点数。与 *word* 相似度得分未达到该值的候选匹配将被忽略。

候选匹配中（不超过 *n* 个）的最佳匹配将以列表形式返回，按相似度得分排序，最相似的排在最前面。

```
>>> get_close_matches('appel', ['ape', 'apple', 'peach', 'puppy'])
['apple', 'ape']
>>> import keyword
>>> get_close_matches('wheel', keyword.kwlist)
['while']
>>> get_close_matches('pineapple', keyword.kwlist)
[]
>>> get_close_matches('accept', keyword.kwlist)
['except']
```

difflib.**ndiff**(*a*, *b*, *linejunk=None*, *charjunk=IS_CHARACTER_JUNK*)

比较 *a* 和 *b* (字符串列表)；返回*Differ* 形式的增量信息 (一个产生增量行的*generator*)。

可选关键字形参 *linejunk* 和 *charjunk* 均为过滤函数 (或为 None)：

linejunk: 此函数接受单个字符串参数，如果其为垃圾字符串则返回真值，否则返回假值。默认为 None。此外还有一个模块层级的函数*IS_LINE_JUNK()*，它会过滤掉没有可见字符的行，除非该行添加了至多一个井号符 ('#') -- 但是下层的*SequenceMatcher* 类会动态分析哪些行的重复频繁到足以形成噪音，这通常会比使用此函数的效果更好。

charjunk: 此函数接受一个字符（长度为 1 的字符串），如果其为垃圾字符则返回真值，否则返回假值。默认为模块层级的函数*IS_CHARACTER_JUNK()*，它会过滤掉空白字符（空格符或制表符；但包含换行符可不是个好主意!）。

Tools/scripts/ndiff.py 是这个函数的命令行前端。

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> print(''.join(diff), end="")
- one
?  ^
+ ore
?  ^
- two
- three
?  -
+ tree
+ emu
```

difflib.**restore**(*sequence*, *which*)

返回两个序列中产生增量的那一个。

给出一个由*Differ.compare()* 或*ndiff()* 产生的 序列，提取出来自文件 1 或 2 (*which* 形参) 的行，去除行前缀。

示例:

```
>>> diff = ndiff('one\ntwo\nthree\n'.splitlines(keepends=True),
...              'ore\ntree\nemu\n'.splitlines(keepends=True))
>>> diff = list(diff) # materialize the generated delta into a list
>>> print(''.join(restore(diff, 1)), end="")
one
two
three
>>> print(''.join(restore(diff, 2)), end="")
ore
tree
emu
```

difflib.**unified_diff**(*a*, *b*, *fromfile=''*, *tofile=''*, *fromfiledate=''*, *tofiledate=''*, *n=3*, *lineterm='\n'*)
比较 *a* 和 *b* (字符串列表)；返回统一差异格式的增量信息 (一个产生增量行的*generator*)。

所以统一差异是一种只显示有更改的行再加几个上下文行的紧凑形式。更改被显示为内联的样式（而不是分开的之前/之后文本块)。上下文行数由 *n* 设定，默认为三行。

默认情况下，差异控制行 (以 ---, +++ 或 @@ 表示) 是通过末尾换行符来创建的。这样做的好处是从`io.IOBase.readlines()` 创建的输入将得到适用于`io.IOBase.writelines()` 的差异信息，因为输入和输出都带有末尾换行符。

对于没有末尾换行符的输入，应将 *lineterm* 参数设为 `""`，这样输出内容将统一不带换行符。

上下文差异格式通常带有一个记录文件名和修改时间的标头。这些信息的部分或全部可以使用字符串 *fromfile*, *tofile*, *fromfiledate* 和 *tofiledate* 来指定。修改时间通常以 ISO 8601 格式表示。如果未指定，这些字符串默认为空。

```
>>> s1 = ['bacon\n', 'eggs\n', 'ham\n', 'guido\n']
>>> s2 = ['python\n', 'eggy\n', 'hamster\n', 'guido\n']
>>> sys.stdout.writelines(unified_diff(s1, s2, fromfile='before.py', tofile=
↪'after.py'))
--- before.py
+++ after.py
@@ -1,4 +1,4 @@
-bacon
-eggs
-ham
+python
+eggy
+hamster
 guido
```

请参阅*difflib 的命令行接口* 获取更详细的示例。

difflib.**diff_bytes**(*dfunc*, *a*, *b*, *fromfile=b''*, *tofile=b''*, *fromfiledate=b''*, *tofiledate=b''*, *n=3*, *lineterm=b'\n'*)
使用 *dfunc* 比较 *a* 和 *b* (字节串对象列表)；产生以 *dfunc* 所返回格式表示的差异行列表（也是字节串)。*dfunc* 必须是可调用对象，通常为`unified_diff()` 或`context_diff()`。

允许你比较编码未知或不一致的数据。除 *n* 之外的所有输入都必须为字节串对象而非字符串。作用方式为无损地将所有输入 (除 *n* 之外) 转换为字符串，并调用 `dfunc(a, b, fromfile, tofile, fromfiledate, tofiledate, n, lineterm)`。*dfunc* 的输出会被随即转换回字节串，这样你所得到的增量行将具有与 *a* 和 *b* 相同的未知/不一致编码。

3.5 版新加入.

difflib.**IS_LINE_JUNK**(*line*)
对于可忽略的行返回 `True`。如果 *line* 为空行或只包含单个 `'#'` 则 *line* 行就是可忽略的，否则就是不可忽略的。此函数被用作较旧版本`ndiff()` 中 *linejunk* 形参的默认值。

difflib.**IS_CHARACTER_JUNK**(*ch*)
对于可忽略的字符返回 `True`。字符 *ch* 如果为空格符或制表符则 *ch* 就是可忽略的，否则就是不可忽略的。此函数被用作`ndiff()` 中 *charjunk* 形参的默认值。

也参考:

模式匹配：格式塔方法 John W. Ratcliff 和 D. E. Metzener 对于一种类似算法的讨论。此文于 1988 年 7 月发表于 Dr. Dobb's Journal。

6.3.1 SequenceMatcher 对象

SequenceMatcher 类具有这样的构造器：

class difflib.**SequenceMatcher**(*isjunk=None*, *a=''*, *b=''*, *autojunk=True*)

可选参数 *isjunk* 必须为 None (默认值) 或为接受一个序列元素并当且仅当其为应忽略的“垃圾”元素时返回真值的单参数函数。传入 None 作为 *isjunk* 的值就相当于传入 `lambda x: False`；也就是说不忽略任何值。例如，传入：

```
lambda x: x in " \t"
```

如果你以字符序列的形式对行进行比较，并且不希望区分空格符或硬制表符。

可选参数 *a* 和 *b* 为要比较的序列；两者默认为空字符串。两个序列的元素都必须为*hashable*。

可选参数 *autojunk* 可用于启用自动垃圾启发式计算。

3.2 版新加入: *autojunk* 形参。

SequenceMatcher 对象接受三个数据属性: *bjunk* 是 *b* 当中 *isjunk* 为 True 的元素集合；*bpopular* 是被启发式计算（如果其未被禁用）视为热门候选的非垃圾元素集合；*b2j* 是将 *b* 当中剩余元素映射到一个它们出现位置列表的字典。所有三个数据属性将在 *b* 通过*set_seqs()* 或*set_seq2()* 重置时被重置。

3.2 版新加入: *bjunk* 和 *bpopular* 属性。

SequenceMatcher 对象具有以下方法：

set_seqs(*a*, *b*)

设置要比较的两个序列。

SequenceMatcher 计算并缓存有关第二个序列的详细信息，这样如果你想要将一个序列与多个序列进行比较，可使用*set_seq2()* 一次性地设置该常用序列并重复地对每个其他序列各调用一次*set_seq1()*。

set_seq1(*a*)

设置要比较的第一个序列。要比较的第二个序列不会改变。

set_seq2(*b*)

设置要比较的第二个序列。要比较的第一个序列不会改变。

find_longest_match(*alo=0*, *ahi=None*, *blo=0*, *bhi=None*)

找出 a[alo:ahi] 和 b[blo:bhi] 中的最长匹配块。

如果 *isjunk* 被省略或为 None，*find_longest_match()* 将返回 (i, j, k) 使得 a[i:i+k] 等于 b[j:j+k]，其中 alo <= i <= i+k <= ahi 并且 blo <= j <= j+k <= bhi。对于所有满足这些条件的 (i', j', k')，如果 i == i', j <= j' 也被满足，则附加条件 k >= k', i <= i'。换句话说，对于所有最长匹配块，返回在 *a* 当中最先出现的一个，而对于在 *a* 当中最先出现的所有最长匹配块，则返回在 *b* 当中最先出现的一个。

```
>>> s = SequenceMatcher(None, " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=0, b=4, size=5)
```

如果提供了 *isjunk*，将按上述规则确定第一个最长匹配块，但额外附加不允许块内出现垃圾元素的限制。然后将通过（仅）匹配两边的垃圾元素来尽可能地扩展该块。这样结果块绝对不会匹配垃圾元素，除非同样的垃圾元素正好与有意义的匹配相邻。

这是与之前相同的例子，但是将空格符视为垃圾。这将防止 ' abcd' 直接与第二个序列末尾的 ' abcd' 相匹配。而只可以匹配 'abcd'，并且是匹配第二个序列最左边的 'abcd'：

```
>>> s = SequenceMatcher(lambda x: x==" ", " abcd", "abcd abcd")
>>> s.find_longest_match(0, 5, 0, 9)
Match(a=1, b=0, size=4)
```

如果未找到匹配块，此方法将返回 `(alo, blo, 0)`。

此方法将返回一个*named tuple* `Match(a, b, size)`。

3.9 版更變: 加入默认参数。

get_matching_blocks()
返回描述非重叠匹配子序列的三元组列表。每个三元组的形式为 `(i, j, n)`，其含义为 `a[i:i+n] == b[j:j+n]`。这些三元组按 *i* 和 *j* 单调递增排列。

最后一个三元组用于占位，其值为 `(len(a), len(b), 0)`。它是唯一 `n == 0` 的三元组。如果 `(i, j, n)` 和 `(i', j', n')` 是在列表中相邻的三元组，且后者不是列表中的最后一个三元组，则 `i+n < i'` 或 `j+n < j'`；换句话说，相邻的三元组总是描述非相邻的相等块。

```
>>> s = SequenceMatcher(None, "abxcd", "abcd")
>>> s.get_matching_blocks()
[Match(a=0, b=0, size=2), Match(a=3, b=2, size=2), Match(a=5, b=4, size=0)]
```

get_opcodes()
返回描述如何将 *a* 变为 *b* 的 5 元组列表，每个元组的形式为 `(tag, i1, i2, j1, j2)`。在第一个元组中 `i1 == j1 == 0`，而在其余的元组中 *i1* 等于前一个元组的 *i2*，并且 *j1* 也等于前一个元组的 *j2*。

tag 值为字符串，其含义如下：

值	意义
`'replace'`	`a[i1:i2]` 应由 `b[j1:j2]` 替换。
`'delete'`	`a[i1:i2]` 应被删除。请注意在此情况下 `j1 == j2`。
`'insert'`	`b[j1:j2]` 应插入到 `a[i1:i1]`。请注意在此情况下 `i1 == i2`。
`'equal'`	`a[i1:i2] == b[j1:j2]` (两个子序列相同)。

例如：

```
>>> a = "qabxcd"
>>> b = "abycdf"
>>> s = SequenceMatcher(None, a, b)
>>> for tag, i1, i2, j1, j2 in s.get_opcodes():
...     print('{:7}   a[{}:{}] --> b[{}:{}] {!r:>8} --> {!r}'.format(
...         tag, i1, i2, j1, j2, a[i1:i2], b[j1:j2]))
delete    a[0:1] --> b[0:0]      'q' --> ''
equal     a[1:3] --> b[0:2]     'ab' --> 'ab'
replace   a[3:4] --> b[2:3]      'x' --> 'y'
equal     a[4:6] --> b[3:5]     'cd' --> 'cd'
insert    a[6:6] --> b[5:6]       '' --> 'f'
```

get_grouped_opcodes(*n=3*)
返回一个带有最多 *n* 行上下文的分组的*generator*。

从*get_opcodes()* 所返回的组开始，此方法会拆分出较小的更改簇并消除没有更改的间隔区域。

这些分组以与*get_opcodes()* 相同的格式返回。

ratio()
返回一个取值范围 [0, 1] 的浮点数作为序列相似性度量。

其中 T 是两个序列中元素的总数量，M 是匹配的数量，即 2.0*M / T。请注意如果两个序列完全相同则该值为 `1.0`，如果两者完全不同则为 `0.0`。

如果*get_matching_blocks()* 或*get_opcodes()* 尚未被调用则此方法运算消耗较大，在此情况下你可能需要先调用*quick_ratio()* 或*real_quick_ratio()* 来获取一个上界。

備F: 注意: *ratio()* 调用的结果可能会取决于参数的顺序。例如:

```
>>> SequenceMatcher(None, 'tide', 'diet').ratio()
0.25
>>> SequenceMatcher(None, 'diet', 'tide').ratio()
0.5
```

---

quick_ratio()
相对快速地返回一个*ratio()* 的上界。

real_quick_ratio()
非常快速地返回一个*ratio()* 的上界。

这三个返回匹配部分占字符总数的比率的方法可能由于不同的近似级别而给出不一样的结果，但是 `quick_ratio()` 和 `real_quick_ratio()` 总是会至少与 `ratio()` 一样大:

```
>>> s = SequenceMatcher(None, "abcd", "bcde")
>>> s.ratio()
0.75
>>> s.quick_ratio()
0.75
>>> s.real_quick_ratio()
1.0
```

6.3.2 SequenceMatcher 的示例

以下示例比较两个字符串，并将空格视为“垃圾”:

```
>>> s = SequenceMatcher(lambda x: x == " ",
...                     "private Thread currentThread;",
...                     "private volatile Thread currentThread;")
```

`ratio()` 返回一个 [0, 1] 范围内的整数作为两个序列相似性的度量。根据经验，`ratio()` 值超过 0.6 就意味着两个序列是近似匹配的:

```
>>> print(round(s.ratio(), 3))
0.866
```

如果你只对两个序列相匹配的位置感兴趣，则 `get_matching_blocks()` 就很方便:

```
>>> for block in s.get_matching_blocks():
...     print("a[%d] and b[%d] match for %d elements" % block)
a[0] and b[0] match for 8 elements
a[8] and b[17] match for 21 elements
a[29] and b[38] match for 0 elements
```

请注意 `get_matching_blocks()` 返回的最后一个元组总是只用于占位的 `(len(a), len(b), 0)`，这也是元组末尾元素（匹配的元素数量）为 `0` 的唯一情况。

如果你想要知道如何将第一个序列转成第二个序列，可以使用 `get_opcodes()`:

```
>>> for opcode in s.get_opcodes():
...     print("%6s a[%d:%d] b[%d:%d]" % opcode)
 equal a[0:8] b[0:8]
insert a[8:8] b[8:17]
 equal a[8:29] b[17:38]
```

也参考:

- 此模块中的*get_close_matches()* 函数显示了如何基于*SequenceMatcher* 构建简单的代码来执行有用的功能。
- 使用*SequenceMatcher* 构建小型应用的 简易版本控制方案。

6.3.3 Differ 对象

请注意*Differ* 所生成的增量并不保证是 **最小**差异。相反，最小差异往往是违反直觉的，因为它们会同步任何可能的地方，有时甚至意外产生相距 100 页的匹配。将同步点限制为连续匹配保留了一些局部性概念，这偶尔会带来产生更长差异的代价。

Differ 类具有这样的构造器:

class difflib.**Differ**(*linejunk=None*, *charjunk=None*)
可选关键字形参 *linejunk* 和 *charjunk* 均为过滤函数 (或为 None):

linejunk: 接受单个字符串作为参数的函数，如果其为垃圾字符串则返回真值。默认值为 None，意味着没有任何行会被视为垃圾行。

charjunk: 接受单个字符（长度为 1 的字符串）作为参数的函数，如果其为垃圾字符则返回真值。默认值为 None，意味着没有任何字符会被视为垃圾字符。

这些垃圾过滤函数可加快查找差异的匹配速度，并且不会导致任何差异行或字符被忽略。请阅读*find_longest_match()* 方法的 *isjunk* 形参的描述了解详情。

Differ 对象是通过一个单独方法来使用（生成增量）的:

compare(*a*, *b*)
比较两个由行组成的序列，并生成增量（一个由行组成的序列）。

每个序列必须包含一个以换行符结尾的单行字符串。这样的序列可以通过文件类对象的*readlines()* 方法来获取。所生成的增量同样由以换行符结尾的字符串构成，可以通过文件类对象的*writelines()* 方法原样打印出来。

6.3.4 Differ 示例

此示例比较两段文本。首先我们设置文本为以换行符结尾的单行字符串构成的序列（这样的序列也可以通过文件类对象的 readlines() 方法来获取):

```
>>> text1 = '''  1. Beautiful is better than ugly.
...   2. Explicit is better than implicit.
...   3. Simple is better than complex.
...   4. Complex is better than complicated.
... '''.splitlines(keepends=True)
>>> len(text1)
4
>>> text1[0][-1]
'\n'
>>> text2 = '''  1. Beautiful is better than ugly.
```

(下页继续)

(繼續上一頁)

```
...    3.   Simple is better than complex.
...    4. Complicated is better than complex.
...    5. Flat is better than nested.
... '''.splitlines(keepends=True)
```

接下来我们实例化一个 Differ 对象：

```
>>> d = Differ()
```

请注意在实例化*Differ* 对象时我们可以传入函数来过滤掉“垃圾”行和字符。详情参见*Differ()* 构造器说明。

最后，我们比较两个序列：

```
>>> result = list(d.compare(text1, text2))
```

result 是一个字符串列表，让我们将其美化打印出来：

```
>>> from pprint import pprint
>>> pprint(result)
['    1. Beautiful is better than ugly.\n',
 '-   2. Explicit is better than implicit.\n',
 '-   3. Simple is better than complex.\n',
 '+   3.   Simple is better than complex.\n',
 '?     ++\n',
 '-   4. Complex is better than complicated.\n',
 '?            ^                     ---- ^\n',
 '+   4. Complicated is better than complex.\n',
 '?           ++++ ^                         ^\n',
 '+   5. Flat is better than nested.\n']
```

作为单独的多行字符串显示出来则是这样：

```
>>> import sys
>>> sys.stdout.writelines(result)
    1. Beautiful is better than ugly.
-   2. Explicit is better than implicit.
-   3. Simple is better than complex.
+   3.   Simple is better than complex.
?     ++
-   4. Complex is better than complicated.
?            ^                     ---- ^
+   4. Complicated is better than complex.
?           ++++ ^                         ^
+   5. Flat is better than nested.
```

6.3.5 difflib 的命令行接口

这个实例演示了如何使用 difflib 来创建一个类似于 `diff` 的工具。它同样包含在 Python 源码发布包中，文件名为 `Tools/scripts/diff.py`。

```
#!/usr/bin/env python3
""" Command line interface to difflib.py providing diffs in four formats:

* ndiff:    lists every line and highlights interline changes.
* context:  highlights clusters of changes in a before/after format.
* unified:  highlights clusters of changes in an inline format.
* html:     generates side by side comparison with change highlights.

"""

import sys, os, difflib, argparse
from datetime import datetime, timezone

def file_mtime(path):
    t = datetime.fromtimestamp(os.stat(path).st_mtime,
                               timezone.utc)
    return t.astimezone().isoformat()

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('-c', action='store_true', default=False,
                        help='Produce a context format diff (default)')
    parser.add_argument('-u', action='store_true', default=False,
                        help='Produce a unified format diff')
    parser.add_argument('-m', action='store_true', default=False,
                        help='Produce HTML side by side diff '
                             '(can use -c and -l in conjunction)')
    parser.add_argument('-n', action='store_true', default=False,
                        help='Produce a ndiff format diff')
    parser.add_argument('-l', '--lines', type=int, default=3,
                        help='Set number of context lines (default 3)')
    parser.add_argument('fromfile')
    parser.add_argument('tofile')
    options = parser.parse_args()

    n = options.lines
    fromfile = options.fromfile
    tofile = options.tofile

    fromdate = file_mtime(fromfile)
    todate = file_mtime(tofile)
    with open(fromfile) as ff:
        fromlines = ff.readlines()
    with open(tofile) as tf:
        tolines = tf.readlines()

    if options.u:
        diff = difflib.unified_diff(fromlines, tolines, fromfile, tofile, fromdate,␣
↪todate, n=n)
    elif options.n:
        diff = difflib.ndiff(fromlines, tolines)
    elif options.m:
```

(下页继续)

(繼續上一頁)

```
        diff = difflib.HtmlDiff().make_file(fromlines,tolines,fromfile,tofile,
↪context=options.c,numlines=n)
    else:
        diff = difflib.context_diff(fromlines, tolines, fromfile, tofile, fromdate,␣
↪todate, n=n)

    sys.stdout.writelines(diff)

if __name__ == '__main__':
    main()
```

6.4 textwrap --- 文本自动换行与填充

源代码: Lib/textwrap.py

---

textwrap 模块提供了一些快捷函数，以及可以完成所有工作的类*TextWrapper*。如果你只是要对一两个文本字符串进行自动换行或填充，快捷函数应该就够用了；否则的话，你应该使用*TextWrapper* 的实例来提高效率。

textwrap.**wrap**(*text*, *width=70*, ***, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder='[...]'*)

对 *text* (字符串) 中的单独段落自动换行以使每行长度最多为 *width* 个字符。返回由输出行组成的列表，行尾不带换行符。

可选的关键字参数对应于*TextWrapper* 的实例属性，具体文档见下。

请参阅*TextWrapper.wrap()* 方法了解有关*wrap()* 行为的详细信息。

textwrap.**fill**(*text*, *width=70*, ***, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, *max_lines=None*, *placeholder='[...]'*)

对 *text* 中的单独段落自动换行，并返回一个包含被自动换行段落的单独字符串。*fill()* 是以下语句的快捷方式

```
"\n".join(wrap(text, ...))
```

特别要说明的是，*fill()* 接受与*wrap()* 完全相同的关键字参数。

textwrap.**shorten**(*text*, *width*, ***, *fix_sentence_endings=False*, *break_long_words=True*, *break_on_hyphens=True*, *placeholder='[...]'*)

折叠并截短给定的 *text* 以符合给定的 *width*。

首先，将折叠 *text* 中的空格（所有连续空格替换为单个空格）。如果结果能适合 *width* 则将其返回。否则将丢弃足够数量的末尾单词以使得剩余单词加 `placeholder` 能适合 `width`:

```
>>> textwrap.shorten("Hello  world!", width=12)
'Hello world!'
>>> textwrap.shorten("Hello  world!", width=11)
'Hello [...]'
>>> textwrap.shorten("Hello world", width=10, placeholder="...")
'Hello...'
```

可选的关键字参数对应于*TextWrapper* 的实际属性，具体见下文。请注意文本在被传入*TextWrapper* 的*fill()* 函数之前会被折叠，因此改变*tabsize*, *expand_tabs*, *drop_whitespace* 和*replace_whitespace* 的值将没有任何效果。

3.4 版新加入.

textwrap.**dedent**(*text*)

移除 *text* 中每一行的任何相同前缀空白符。

这可以用来清除三重引号字符串行左侧空格，而仍然在源码中显示为缩进格式。

请注意制表符和空格符都被视为是空白符，但它们并不相等：以下两行 `"  hello"` 和 `"\thello"` 不会被视为具有相同的前缀空白符。

只包含空白符的行会在输入时被忽略并在输出时被标准化为单个换行符。

例如:

```
def test():
    # end first line with \ to avoid the empty line!
    s = '''\
    hello
      world
    '''
    print(repr(s))          # prints '    hello\n      world\n    '
    print(repr(dedent(s)))  # prints 'hello\n  world\n'
```

textwrap.**indent**(*text*, *prefix*, *predicate=None*)

将 *prefix* 添加到 *text* 中选定行的开头。

通过调用 `text.splitlines(True)` 来对行进行拆分。

默认情况下，*prefix* 会被添加到所有不是只由空白符（包括任何行结束符）组成的行。

例如:

```
>>> s = 'hello\n\n \nworld'
>>> indent(s, '  ')
'  hello\n\n \n  world'
```

可选的 *predicate* 参数可用来控制哪些行要缩进。例如，可以很容易地为空行或只有空白符的行添加 *prefix*:

```
>>> print(indent(s, '+ ', lambda line: True))
+ hello
+
+
+ world
```

3.3 版新加入.

wrap(), *fill()* 和*shorten()* 的作用方式为创建一个*TextWrapper* 实例并在其上调用单个方法。该实例不会被重用，因此对于要使用*wrap()* 和/或*fill()* 来处理许多文本字符串的应用来说，创建你自己的*TextWrapper* 对象可能会更有效率。

文本最好在空白符位置自动换行，包括带连字符单词的连字符之后；长单词仅在必要时会被拆分，除非*TextWrapper.break_long_words* 被设为假值。

class textwrap.**TextWrapper**(***kwargs*)

TextWrapper 构造器接受多个可选的关键字参数。每个关键字参数对应一个实例属性，比如说

```
wrapper = TextWrapper(initial_indent="* ")
```

相当于：

```
wrapper = TextWrapper()
wrapper.initial_indent = "* "
```

你可以多次重用相同的*TextWrapper* 对象，并且你也可以在使用期间通过直接向实例属性赋值来修改它的任何选项。

TextWrapper 的实例属性（以及构造器的关键字参数）如下所示：

width
(默认: 70) 自动换行的最大行长度。只要输入文本中没有长于*width* 的单个单词，*TextWrapper* 就能保证没有长于*width* 个字符的输出行。

expand_tabs
(默认: True) 如果为真值，则 *text* 中所有的制表符将使用 *text* 的 expandtabs() 方法扩展为空格符。

tabsize
(默认: 8) 如果*expand_tabs* 为真值，则 *text* 中所有的制表符将扩展为零个或多个空格，具体取决于当前列位置和给定的制表宽度。

3.3 版新加入.

replace_whitespace
(default: True) 如果为真值，在制表符扩展之后、自动换行之前，*wrap()* 方法将把每个空白字符都替换为单个空格。会被替换的空白字符如下：制表，换行，垂直制表，进纸和回车 ('\t\n\v\f\r')。

備🄵: 如果*expand_tabs* 为假值且*replace_whitespace* 为真值，每个制表符将被替换为单个空格，这与制表符扩展是 不一样的。

備🄵: 如果*replace_whitespace* 为假值，在一行的中间有可能出现换行符并导致怪异的输出。因此，文本应当（使用*str.splitlines()* 或类似方法）拆分为段落并分别进行自动换行。

---

drop_whitespace
(默认: True) 如果为真值，每一行开头和末尾的空白字符（在包装之后、缩进之前）会被丢弃。但是段落开头的空白字符如果后面不带任何非空白字符则不会被丢弃。如果被丢弃的空白字符占据了一个整行，则该整行将被丢弃。

initial_indent
(默认: '') 将被添加到被自动换行输出内容的第一行的字符串。其长度会被计入第一行的长度。空字符串不会被缩进。

subsequent_indent
(default: '') 将被添加到被自动换行输出内容除第一行外的所有行的字符串。其长度会被计入除行一行外的所有行的长度。

fix_sentence_endings
(默认: False) 如果为真值，*TextWrapper* 将尝试检测句子结尾并确保句子间总是以恰好两个空格符分隔。对于使用等宽字体的文本来说通常都需要这样。但是，句子检测算法并不完美：它假定句子结尾是一个小写字母加字符 '.', '!' 或 '?' 中的一个，并可能带有字符 '"' 或 "'"，最后以一个空格结束。此算法的问题之一是它无法区分以下文本中的"Dr."

```
[...] Dr. Frankenstein's monster [...]
```

和以下文本中的”Spot.”

```
[...] See Spot. See Spot run [...]
```

fix_sentence_endings 默认为假值。

由于句子检测算法依赖于 `string.lowercase` 来确定“小写字母”，以及约定在句点后使用两个空格来分隔处于同一行的句子，因此只适用于英语文本。

break_long_words
(默认: `True`) 如果为真值，则长度超过*width* 的单词将被分开以保证行的长度不会超过*width*。如果为假值，超长单词不会被分开，因而某些行的长度可能会超过*width*。(超长单词将被单独作为一行，以尽量减少超出*width* 的情况。)

break_on_hyphens
(默认: `True`) 如果为真值，将根据英语的惯例首选在空白符和复合词的连字符之后自动换行。如果为假值，则只有空白符会被视为合适的潜在断行位置，但如果你确实不希望出现分开的单词则你必须将*break_long_words* 设为假值。之前版本的默认行为总是允许分开带有连字符的单词。

max_lines
(默认: `None`) 如果不为 `None`，则输出内容将最多包含 *max_lines* 行，并使 *placeholder* 出现在输出内容的末尾。

3.4 版新加入.

placeholder
(默认: `' [...]'`) 该文本将在输出文本被截短时出现在文本末尾。

3.4 版新加入.

TextWrapper 还提供了一些公有方法，类似于模块层级的便捷函数:

wrap(*text*)
对 *text* (字符串) 中的单独段落自动换行以使每行长度最多为*width* 个字符。所有自动换行选项均获取自*TextWrapper* 实例的实例属性。返回由输出行组成的列表，行尾不带换行符。如果自动换行输出结果没有任何内容，则返回空列表。

fill(*text*)
对 *text* 中的单独段落自动换行并返回包含被自动换行段落的单独字符串。

6.5 unicodedata --- Unicode 数据库

此模块提供了对 Unicode Character Database (UCD) 的访问，其中定义了所有 Unicode 字符的字符属性。此数据库中包含的数据编译自 UCD 版本 13.0.0。

该模块使用与 Unicode 标准附件 #44 “Unicode 字符数据库” 中所定义的相同名称和符号。它定义了以下函数:

unicodedata.**lookup**(*name*)
按名称查找字符。如果找到具有给定名称的字符，则返回相应的字符。如果没有找到，则*KeyError* 被引发。

3.3 版更變: 已添加对名称别名[1] 和命名序列[2] 的支持。

[1] https://www.unicode.org/Public/13.0.0/ucd/NameAliases.txt
[2] https://www.unicode.org/Public/13.0.0/ucd/NamedSequences.txt

unicodedata.**name**(*chr*[, *default*])
返回分配给字符 *chr* 的名称作为字符串。如果没有定义名称，则返回 *default* ，如果没有给出，则*ValueError* 被引发。

unicodedata.**decimal**(*chr*[, *default*])
返回分配给字符 *chr* 的十进制值作为整数。如果没有定义这样的值，则返回 *default* ，如果没有给出，则*ValueError* 被引发。

unicodedata.**digit**(*chr*[, *default*])
返回分配给字符 *chr* 的数字值作为整数。如果没有定义这样的值，则返回 *default* ，如果没有给出，则*ValueError* 被引发。

unicodedata.**numeric**(*chr*[, *default*])
返回分配给字符 *chr* 的数值作为浮点数。如果没有定义这样的值，则返回 *default* ，如果没有给出，则*ValueError* 被引发。

unicodedata.**category**(*chr*)
返回分配给字符 *chr* 的常规类别为字符串。

unicodedata.**bidirectional**(*chr*)
返回分配给字符 *chr* 的双向类作为字符串。如果未定义此类值，则返回空字符串。

unicodedata.**combining**(*chr*)
返回分配给字符 *chr* 的规范组合类作为整数。如果没有定义组合类，则返回 `0` 。

unicodedata.**east_asian_width**(*chr*)
返回分配给字符 *chr* 的东亚宽度作为字符串。

unicodedata.**mirrored**(*chr*)
返回分配给字符 *chr* 的镜像属性为整数。如果字符在双向文本中被识别为“镜像”字符，则返回 `1` ，否则返回 `0` 。

unicodedata.**decomposition**(*chr*)
返回分配给字符 *chr* 的字符分解映射作为字符串。如果未定义此类映射，则返回空字符串。

unicodedata.**normalize**(*form*, *unistr*)
返回 Unicode 字符串 *unistr* 的正常形式 *form* 。*form* 的有效值为'NFC' 、'NFKC' 、'NFD' 和'NFKD' 。

Unicode 标准基于规范等价和兼容性等效的定义定义了 Unicode 字符串的各种规范化形式。在 Unicode 中，可以以各种方式表示多个字符。例如，字符 U+00C7（带有 CEDILLA 的 LATIN CAPITAL LETTER C ）也可以表示为序列 U+0043（LATIN CAPITAL LETTER C ）U+0327（COMBINING CEDILLA ）。

对于每个字符，有两种正规形式：正规形式 C 和正规形式 D 。正规形式 D（NFD）也称为规范分解，并将每个字符转换为其分解形式。正规形式 C（NFC）首先应用规范分解，然后再次组合预组合字符。

除了这两种形式之外，还有两种基于兼容性等效的其他常规形式。在 Unicode 中，支持某些字符，这些字符通常与其他字符统一。例如，U+2160（ROMAN NUMERAL ONE）与 U+0049（LATIN CAPITAL LETTER I）完全相同。但是，Unicode 支持它与现有字符集（例如 gb2312 ）的兼容性。

正规形式 KD（NFKD）将应用兼容性分解，即用其等价项替换所有兼容性字符。正规形式 KC（NFKC）首先应用兼容性分解，然后是规范组合。

即使两个 unicode 字符串被规范化并且人类读者看起来相同，如果一个具有组合字符而另一个没有，则它们可能无法相等。

unicodedata.**is_normalized**(*form*, *unistr*)
判断 Unicode 字符串 *unistr* 是否为正规形式 *form*。*form* 的有效值为'NFC', 'NFKC', 'NFD' 和'NFKD'。

3.8 版新加入.

此外，该模块暴露了以下常量：

unicodedata.**unidata_version**
此模块中使用的 Unicode 数据库的版本。

unicodedata.**ucd_3_2_0**
这是一个与整个模块具有相同方法的对象，但对于需要此特定版本的 Unicode 数据库（如 IDNA ）的应用程序，则使用 Unicode 数据库版本 3.2 。

示例：

```
>>> import unicodedata
>>> unicodedata.lookup('LEFT CURLY BRACKET')
'{'
>>> unicodedata.name('/')
'SOLIDUS'
>>> unicodedata.decimal('9')
9
>>> unicodedata.decimal('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not a decimal
>>> unicodedata.category('A')  # 'L'etter, 'u'ppercase
'Lu'
>>> unicodedata.bidirectional('\u0660') # 'A'rabic, 'N'umber
'AN'
```

🄵解

6.6 stringprep --- 因特网字符串预备

源代码: Lib/stringprep.py

在标识因特网上的事物（例如主机名），经常需要比较这些标识是否（相等）。这种比较的具体执行可能会取决于应用域的不同，例如是否要区分大小写等等。有时也可能需要限制允许的标识为仅由“可打印”字符组成。

RFC 3454 定义了在因特网协议中 Unicode 字符串的“预备”过程。在将字符串连线传输之前，它们会先使用预备过程进行处理，之后它们将具有特定的标准形式。该 RFC 定义了一系列表格，它们可以被组合为选项配置。每个配置必须定义所使用的表格，stringprep 过程的其他可选项也是配置的组成部分。stringprep 配置的一个例子是 nameprep，它被用于国际化域名。

模块*stringprep* 仅公开了来自 **RFC 3454** 的表格。由于这些如果表格如果表示为字典或列表将会非常庞大，该模块在内部使用 Unicode 字符数据库。该模块本身的源代码是使用 mkstringprep.py 工具生成的。

因此，这些表格以函数而非数据结构的形式公开。在 RFC 中有两种表格：集合与映射。对于集合，*stringprep* 提供了“特征函数”，即如果形参是集合的一部分则返回值为 True 的函数。对于映射，它提供了映射函数：它会根据给定的键返回所关联的值。以下是模块中所有可用函数的列表。

stringprep.**in_table_a1**(*code*)
确定 *code* 是否属于 tableA.1 (Unicode 3.2 中的未分配码位)。

stringprep.**in_table_b1**(*code*)
确定 *code* 是否属于 tableB.1 (通常映射为空值)。

stringprep.**map_table_b2**(*code*)
返回 *code* 依据 tableB.2 (配合 NFKC 使用的大小写转换映射) 所映射的值。

stringprep.**map_table_b3**(*code*)
　　返回 *code* 依据 tableB.3 (不附带正规化的大小写折叠映射) 所映射的值。

stringprep.**in_table_c11**(*code*)
　　确定 *code* 是否属于 tableC.1.1 (ASCII 空白字符)。

stringprep.**in_table_c12**(*code*)
　　确定 *code* 是否属于 tableC.1.2 (非 ASCII 空白字符)。

stringprep.**in_table_c11_c12**(*code*)
　　确定 *code* 是否属于 tableC.1 (空白字符，C.1.1 和 C.1.2 的并集)。

stringprep.**in_table_c21**(*code*)
　　确定 *code* 是否属于 tableC.2.1 (ASCII 控制字符)。

stringprep.**in_table_c22**(*code*)
　　确定 *code* 是否属于 tableC.2.2 (非 ASCII 控制字符)。

stringprep.**in_table_c21_c22**(*code*)
　　确定 *code* 是否属于 tableC.2 (控制字符，C.2.1 和 C.2.2 的并集)。

stringprep.**in_table_c3**(*code*)
　　确定 *code* 是否属于 tableC.3 (私有使用)。

stringprep.**in_table_c4**(*code*)
　　确定 *code* 是否属于 tableC.4 (非字符码位)。

stringprep.**in_table_c5**(*code*)
　　确定 *code* 是否属于 tableC.5 (替代码)。

stringprep.**in_table_c6**(*code*)
　　确定 *code* 是否属于 tableC.6 (不适用于纯文本)。

stringprep.**in_table_c7**(*code*)
　　确定 *code* 是否属于 tableC.7 (不适用于规范表示)。

stringprep.**in_table_c8**(*code*)
　　确定 *code* 是否属于 tableC.8 (改变显示属性或已弃用)。

stringprep.**in_table_c9**(*code*)
　　确定 *code* 是否属于 tableC.9 (标记字符)。

stringprep.**in_table_d1**(*code*)
　　确定 *code* 是否属于 tableD.1 (带有双向属性"R" 或"AL" 的字符)。

stringprep.**in_table_d2**(*code*)
　　确定 *code* 是否属于 tableD.2 (带有双向属性"L" 的字符)。

6.7 readline --- GNU readline 接口

readline 模块定义了许多方便从 Python 解释器完成和读取/写入历史文件的函数。此模块可以直接使用，或通过支持在交互提示符下完成 Python 标识符的*rlcompleter* 模块使用。使用此模块进行的设置会同时影响解释器的交互提示符以及内置*input()* 函数提供的提示符。

Readline keybindings may be configured via an initialization file, typically `.inputrc` in your home directory. See Readline Init File in the GNU Readline manual for information about the format and allowable constructs of that file, and the capabilities of the Readline library in general.

備𝔽: 底层的 Readline 库 API 可能使用 `libedit` 库来实现而不是 GNU readline。在 macOS 上*`readline`* 模块会在运行时检测所使用的是哪个库。

`libedit` 所用的配置文件与 GNU readline 的不同。如果你要在程序中载入配置字符串你可以在 `readline.__doc__` 中检测文本"libedit" 来区分 GNU readline 和 libedit。

如果你是在 macOS 上使用 *editline*/`libedit` readline 模拟，则位于你的主目录中的初始化文件名称为 `.editrc`。例如，`~/.editrc` 中的以下内容将开启 *vi* 按键绑定以及 TAB 补全:

```
python:bind -v
python:bind ^I rl_complete
```

6.7.1 初始化文件

下列函数与初始化文件和用户配置有关:

`readline.`**`parse_and_bind`**(*string*)
执行在 *string* 参数中提供的初始化行。此函数会调用底层库中的 `rl_parse_and_bind()`。

`readline.`**`read_init_file`**([*filename*])
执行一个 readline 初始化文件。默认文件名为最近所使用的文件名。此函数会调用底层库中的 `rl_read_init_file()`。

6.7.2 行缓冲区

下列函数会在行缓冲区上操作。

`readline.`**`get_line_buffer`**()
返回行缓冲区的当前内容 (底层库中的 `rl_line_buffer`)。

`readline.`**`insert_text`**(*string*)
将文本插入行缓冲区的当前游标位置。该函数会调用底层库中的 `rl_insert_text()`，但会忽略其返回值。

`readline.`**`redisplay`**()
改变屏幕的显示以反映行缓冲区的当前内容。该函数会调用底层库中的 `rl_redisplay()`。

6.7.3 历史文件

下列函数会在历史文件上操作:

`readline.`**`read_history_file`**([*filename*])
载入一个 readline 历史文件，并将其添加到历史列表。默认文件名为 `~/.history`。此函数会调用底层库中的 `read_history()`。

`readline.`**`write_history_file`**([*filename*])
将历史列表保存为 readline 历史文件，覆盖任何现有文件。默认文件名为 `~/.history`。此函数会调用底层库中的 `write_history()`。

`readline.`**`append_history_file`**(*nelements*[, *filename*])
将历史列表的最后 *nelements* 项添加到历史文件。默认文件名为 `~/.history`。文件必须已存在。此函数会调用底层库中的 `append_history()`。此函数仅当 Python 编译包带有支持此功能的库版本时才会存在。

3.5 版新加入.

readline.**get_history_length**()
readline.**set_history_length**(*length*)
设置或返回需要保存到历史文件的行数。*write_history_file()* 函数会通过调用底层库中的 history_truncate_file() 以使用该值来截取历史文件。负值意味着不限制历史文件的大小。

6.7.4 历史列表

以下函数会在全局历史列表上操作：

readline.**clear_history**()
清除当前历史。此函数会调用底层库的 clear_history()。此 Python 函数仅当 Python 编译包带有支持此功能的库版本时才会存在。

readline.**get_current_history_length**()
返回历史列表的当前项数。(此函数不同于*get_history_length()*，后者是返回将被写入历史文件的最大行数。)

readline.**get_history_item**(*index*)
返回序号为 *index* 的历史条目的当前内容。条目序号从一开始。此函数会调用底层库中的 history_get()。

readline.**remove_history_item**(*pos*)
从历史列表中移除指定位置上的历史条目。条目位置从零开始。此函数会调用底层库中的 remove_history()。

readline.**replace_history_item**(*pos*, *line*)
将指定位置上的历史条目替换为 *line*。条目位置从零开始。此函数会调用底层库中的 replace_history_entry()。

readline.**add_history**(*line*)
将 *line* 添加到历史缓冲区，相当于是最近输入的一行。此函数会调用底层库中的 add_history()。

readline.**set_auto_history**(*enabled*)
启用或禁用当通过 readline 读取输入时自动调用 add_history()。*enabled* 参数应为一个布尔值，当其为真值时启用自动历史，当其为假值时禁用自动历史。

3.6 版新加入.

CPython implementation detail: Auto history is enabled by default, and changes to this do not persist across multiple sessions.

6.7.5 启动钩子

readline.**set_startup_hook**([*function*])
设置或移除底层库的 rl_startup_hook 回调所发起调用的函数。如果指定了 *function*，它将被用作新的钩子函数；如果省略或为 None，任何已安装的函数将被移除。钩子函数将在 readline 打印第一个提示信息之前不带参数地被调用。

readline.**set_pre_input_hook**([*function*])
设置或移除底层库的 rl_pre_input_hook 回调所发起调用的函数。如果指定了 *function*，它将被用作新的钩子函数；如果省略或为 None，任何已安装的函数将被移除。钩子函数将在打印第一个提示信息之后、readline 开始读取输入字符之前不带参数地被调用。此函数仅当 Python 编译包带有支持此功能的库版本时才会存在。

6.7.6 Completion

以下函数与自定义单词补全函数的实现有关。这通常使用 Tab 键进行操作，能够提示并自动补全正在输入的单词。默认情况下，Readline 设置为由*rlcompleter* 来补全交互模式解释器的 Python 标识符。如果*readline* 模块要配合自定义的补全函数来使用，则需要设置不同的单词分隔符。

readline.**set_completer**([*function*])
设置或移除补全函数。如果指定了 *function*，它将被用作新的补全函数；如果省略或为 None，任何已安装的补全函数将被移除。补全函数的调用形式为 function(text, state)，其中 *state* 为 0, 1, 2, ...，直至其返回一个非字符串值。它应当返回下一个以 *text* 开头的候选补全内容。

已安装的补全函数将由传递给底层库中 rl_completion_matches() 的 *entry_func* 回调函数来发起调用。*text* 字符串来自于底层库中 rl_attempted_completion_function 回调函数的第一个形参。

readline.**get_completer**()
获取补全函数，如果没有设置补全函数则返回 None。

readline.**get_completion_type**()
获取正在尝试的补全类型。此函数会将底层库中的 rl_completion_type 变量作为一个整数返回。

readline.**get_begidx**()
readline.**get_endidx**()
获取补全域的开始和结束序号。这些序号就是传给底层库中 rl_attempted_completion_function 回调函数的 *start* 和 *end* 参数。

readline.**set_completer_delims**(*string*)
readline.**get_completer_delims**()
设置或获取补全的单词分隔符。此分隔符确定了要考虑补全的单词的开始和结束位置（补全域）。这些函数会访问底层库的 rl_completer_word_break_characters 变量。

readline.**set_completion_display_matches_hook**([*function*])
设置或移除补全显示函数。如果指定了 *function*，它将被用作新的补全显示函数；如果省略或为 None，任何已安装的补全显示函数将被移除。此函数会设置或清除底层库的 rl_completion_display_matches_hook 回调函数。补全显示函数会在每次需要显示匹配项时以 function(substitution, [matches], longest_match_length) 的形式被调用。

6.7.7 示例

以下示例演示了如何使用*readline* 模块的历史读取或写入函数来自动加载和保存用户主目录下名为 .python_history 的历史文件。以下代码通常应当在交互会话期间从用户的 PYTHONSTARTUP 文件自动执行。

```
import atexit
import os
import readline

histfile = os.path.join(os.path.expanduser("~"), ".python_history")
try:
    readline.read_history_file(histfile)
    # default history len is -1 (infinite), which may grow unruly
    readline.set_history_length(1000)
except FileNotFoundError:
    pass

atexit.register(readline.write_history_file, histfile)
```

此代码实际上会在 Python 运行于 交互模式时自动运行 (参见*Readline 配置*)。

以下示例实现了同样的目标，但是通过只添加新历史的方式来支持并发的交互会话。

```
import atexit
import os
import readline
histfile = os.path.join(os.path.expanduser("~"), ".python_history")

try:
    readline.read_history_file(histfile)
    h_len = readline.get_current_history_length()
except FileNotFoundError:
    open(histfile, 'wb').close()
    h_len = 0

def save(prev_h_len, histfile):
    new_h_len = readline.get_current_history_length()
    readline.set_history_length(1000)
    readline.append_history_file(new_h_len - prev_h_len, histfile)
atexit.register(save, h_len, histfile)
```

以下示例扩展了*code.InteractiveConsole* 类以支持历史保存/恢复。

```
import atexit
import code
import os
import readline

class HistoryConsole(code.InteractiveConsole):
    def __init__(self, locals=None, filename="<console>",
                 histfile=os.path.expanduser("~/.console-history")):
        code.InteractiveConsole.__init__(self, locals, filename)
        self.init_history(histfile)

    def init_history(self, histfile):
        readline.parse_and_bind("tab: complete")
        if hasattr(readline, "read_history_file"):
            try:
                readline.read_history_file(histfile)
            except FileNotFoundError:
                pass
            atexit.register(self.save_history, histfile)

    def save_history(self, histfile):
        readline.set_history_length(1000)
        readline.write_history_file(histfile)
```

6.8 `rlcompleter` --- GNU readline 的补全函数

源代码: Lib/rlcompleter.py

`rlcompeleter` 通过补全有效的 Python 标识符和关键字定义了一个适用于*readline* 模块的补全函数。

当此模块在具有可用的*readline* 模块的 Unix 平台被导入, 一个 `Completer` 实例将被自动创建并且它的 `complete()` 方法将设置为*readline* 的补全器.

示例:

```
>>> import rlcompleter
>>> import readline
>>> readline.parse_and_bind("tab: complete")
>>> readline. <TAB PRESSED>
readline.__doc__          readline.get_line_buffer(  readline.read_init_file(
readline.__file__         readline.insert_text(      readline.set_completer(
readline.__name__         readline.parse_and_bind(
>>> readline.
```

rlcompleter 模块是为了使用 Python 的 交互模式而设计的。除非 Python 是通过 `-S` 选项运行, 这个模块总是自动地被导入且配置 (参见*Readline 配置*)。

在没有*readline* 的平台, 此模块定义的 `Completer` 类仍然可以用于自定义行为.

6.8.1 Completer 对象

Completer 对象具有以下方法:

`Completer.`**`complete`**`(`*text*, *state*`)`
为 *text* 返回第 *state* 项补全。

如果指定的 *text* 不包含句点字符 (`'.'`), 它将根据当前*__main__*, *builtins* 和保留关键字（定义于*keyword* 模块）所定义的名称进行补全。

如果为带有句点的名称执行调用，它将尝试尽量求值直到最后一部分为止而不产生附带影响（函数不会被求值，但它可以生成对 `__getattr__()` 的调用），并通过*dir()* 函数来匹配剩余部分。在对表达式求值期间引发的任何异常都会被捕获、静默处理并返回*None*。

CHAPTER 7

二进制数据服务

本章介绍的模块提供了一些操作二进制数据的基本服务操作。有关二进制数据的其他操作，特别是与文件格式和网络协议有关的操作，将在相关章节中介绍。

下面描述的一些库*文本處理 (Text Processing) 服務* 也可以使用 ASCII 兼容的二进制格式（例如`re`）或所有二进制数据（例如`difflib`）。

另外，请参阅 Python 的内置二进制数据类型的文档*二进制序列类型 --- bytes, bytearray, memoryview* 。

7.1 `struct` --- 将字节串解读为打包的二进制数据

源代码： Lib/struct.py

此模块可以执行 Python 值和以 Python `bytes` 对象表示的 C 结构之间的转换。这可以被用来处理存储在文件中或是从网络连接等其他来源获取的二进制数据。它使用*格式字符串* 作为 C 结构布局的精简描述以及与 Python 值的双向转换。

备🄴： 默认情况下，打包给定 C 结构的结果会包含填充字节以使得所涉及的 C 类型保持正确的对齐；类似地，对齐在解包时也会被纳入考虑。选择此种行为的目的是使得被打包结构的字节能与相应 C 结构在内存中的布局完全一致。要处理平台独立的数据格式或省略隐式的填充字节，请使用 `standard` 大小和对齐而不是 `native` 大小和对齐：详情参见*字节顺序，大小和对齐方式*。

某些`struct` 的函数（以及`Struct` 的方法）接受一个 *buffer* 参数。这将指向实现了 bufferobjects 并提供只读或是可读写缓冲的对象。用于此目的的最常见类型为`bytes` 和`bytearray`，但许多其他可被视为字节数组的类型也实现了缓冲协议，因此它们无需额外从`bytes` 对象复制即可被读取或填充。

7.1.1 函数和异常

此模块定义了下列异常和函数：

exception struct.**error**
会在多种场合下被引发的异常；其参数为一个描述错误信息的字符串。

struct.**pack**(*format*, *v1*, *v2*, ...)
返回一个 bytes 对象，其中包含根据格式字符串 *format* 打包的值 *v1*, *v2*, ... 参数个数必须与格式字符串所要求的值完全匹配。

struct.**pack_into**(*format*, *buffer*, *offset*, *v1*, *v2*, ...)
根据格式字符串 *format* 打包 *v1*, *v2*, ... 等值并将打包的字节串写入可写缓冲区 *buffer* 从 *offset* 开始的位置。请注意 *offset* 是必需的参数。

struct.**unpack**(*format*, *buffer*)
根据格式字符串 *format* 从缓冲区 *buffer* 解包（假定是由 `pack(format, ...)` 打包）。结果为一个元组，即使其只包含一个条目。缓冲区的字节大小必须匹配格式所要求的大小，如*calcsize()* 所示。

struct.**unpack_from**(*format*, */*, *buffer*, *offset=0*)
对 *buffer* 从位置 *offset* 开始根据格式字符串 *format* 进行解包。结果为一个元组，即使其中只包含一个条目。缓冲区的字节大小从位置 *offset* 开始必须至少为*calcsize()* 显示的格式所要求的大小。

struct.**iter_unpack**(*format*, *buffer*)
根据格式字符串 *format* 以迭代方式从缓冲区 *buffer* 解包。此函数返回一个迭代器，它将从缓冲区读取相同大小的块直至其内容全部耗尽。缓冲区的字节大小必须整数倍于格式所要求的大小，如*calcsize()* 所示。

每次迭代将产生一个如格式字符串所指定的元组。

3.4 版新加入.

struct.**calcsize**(*format*)
返回与格式字符串 *format* 相对应的结构的大小（亦即 `pack(format, ...)` 所产生的字节串对象的大小）。

7.1.2 格式字符串

格式字符串是用来在打包和解包数据时指定预期布局的机制。它们使用指定被打包/解包数据类型的格式字符 进行构建。此外，还有一些特殊字符用来控制字节顺序，大小和对齐方式。

字节顺序，大小和对齐方式

默认情况下，C 类型以机器的本机格式和字节顺序表示，并在必要时通过跳过填充字节进行正确对齐（根据 C 编译器使用的规则）。

或者，根据下表，格式字符串的第一个字符可用于指示打包数据的字节顺序，大小和对齐方式：

字符	字节顺序	大小	对齐方式
`@`	按原字节	按原字节	按原字节
`=`	按原字节	标准	无
`<`	小端	标准	无
`>`	大端	标准	无
`!`	网络（= 大端）	标准	无

如果第一个字符不是其中之一，则假定为 `'@'` 。

本机字节顺序可能为大端或是小端，取决于主机系统的不同。例如，Intel x86 和 AMD64 (x86-64) 是小端的；Motorola 68000 和 PowerPC G5 是大端的；ARM 和 Intel Itanium 具有可切换的字节顺序（双端）。请使用 `sys.byteorder` 来检查你的系统字节顺序。

本机大小和对齐方式是使用 C 编译器的 `sizeof` 表达式来确定的。这总是会与本机字节顺序相绑定。

标准大小仅取决于格式字符；请参阅格式字符 部分中的表格。

请注意 `'@'` 和 `'='` 之间的区别：两个都使用本机字节顺序，但后者的大小和对齐方式是标准化的。

形式 `'!'` 代表网络字节顺序总是使用在 IETF RFC 1700 中所定义的大端序。

没有什么方式能指定非本机字节顺序（强制字节对调）；请正确选择使用 `'<'` 或 `'>'`。

F解：

(1) 填充只会在连续结构成员之间自动添加。填充不会添加到已编码结构的开头和末尾。

(2) 当使用非本机大小和对齐方式即'<', '>', '=', and '!' 时不会添加任何填充。

(3) 要将结构的末尾对齐到符合特定类型的对齐要求，请以该类型代码加重复计数的零作为格式结束。参见示例。

格式字符

格式字符具有以下含义；C 和 Python 值之间的按其指定类型的转换应当是相当明显的。'标准大小' 列是指当使用标准大小时以字节表示的已打包值大小；也就是当格式字符串以 `'<'`, `'>'`, `'!'` 或 `'='` 之一开头的情况。当使用本机大小时，已打包值的大小取决于具体的平台。

格式	C Type	Python 类型	标准大小	F解
`x`	填充字节	无		
`c`	`char`	长度为 1 的字节串	1	
`b`	`signed char`	整数	1	(1), (2)
`B`	`unsigned char`	整数	1	(2)
`?`	`_Bool`	bool	1	(1)
`h`	`short`	整数	2	(2)
`H`	`unsigned short`	整数	2	(2)
`i`	`int`	整数	4	(2)
`I`	`unsigned int`	整数	4	(2)
`l`	`long`	整数	4	(2)
`L`	`unsigned long`	整数	4	(2)
`q`	`long long`	整数	8	(2)
`Q`	`unsigned long long`	整数	8	(2)
`n`	`ssize_t`	整数		(3)
`N`	`size_t`	整数		(3)
`e`	(6)	float	2	(4)
`f`	`float`	float	4	(4)
`d`	`double`	float	8	(4)
`s`	`char[]`	字节串		
`p`	`char[]`	字节串		
`P`	`void *`	整数		(5)

3.3 版更變: 增加了对 `'n'` 和 `'N'` 格式的支持

3.6 版更變: 添加了对 `'e'` 格式的支持。

F解：

(1) '?' 转换码对应于 C99 定义的 _Bool 类型。如果此类型不可用，则使用 char 来模拟。在标准模式下，它总是以一个字节表示。

(2) 当尝试使用任何整数转换码打包一个非整数时，如果该非整数具有 __index__() 方法，则会在打包之前调用该方法将参数转换为一个整数。

3.2 版更變: 增加了针对非整数使用 __index__() 方法的特性。

(3) 'n' 和 'N' 转换码仅对本机大小可用（选择为默认或使用 '@' 字节顺序字符）。对于标准大小，你可以使用适合你的应用的任何其他整数格式。

(4) 对于 'f', 'd' 和 'e' 转换码，打包表示形式将使用 IEEE 754 binary32, binary64 或 binary16 格式 (分别对应于 'f', 'd' 或 'e')，无论平台使用何种浮点格式。

(5) 'P' 格式字符仅对本机字节顺序可用（选择为默认或使用 '@' 字节顺序字符）。字节顺序字符 '=' 选择使用基于主机系统的小端或大端排序。struct 模块不会将其解读为本机排序，因此 'P' 格式将不可用。

(6) IEEE 754 binary16 "半精度" 类型是在 IEEE 754 标准 的 2008 修订版中引入的。它包含一个符号位，5 个指数位和 11 个精度位（明确存储 10 位），可以完全精确地表示大致范围在 6.1e-05 和 6.5e+04 之间的数字。此类型并不被 C 编译器广泛支持：在一台典型的机器上，可以使用 unsigned short 进行存储，但不会被用于数学运算。请参阅维基百科页面 half-precision floating-point format 了解详情。

格式字符之前可以带有整数重复计数。例如，格式字符串 '4h' 的含义与 'hhhh' 完全相同。

格式之间的空白字符会被忽略；但是计数及其格式字符中不可有空白字符。

对于 's' 格式字符，计数会被解析为字节的长度，而不是像其他格式字符那样的重复计数；例如，'10s' 表示一个 10 字节的字节串，而 '10c' 表示 10 个字符。如果未给出计数，则默认值为 1。对于打包操作，字节串会被适当地截断或填充空字节以符合要求。对于解包操作，结果字节对象总是恰好具有指定数量的字节。作为特殊情况，'0s' 表示一个空字符串（而 '0c' 表示 0 个字符）。

当使用某一种整数格式 ('b', 'B', 'h', 'H', 'i', 'I', 'l', 'L', 'q', 'Q') 打包值 x 时，如果 x 在该格式的有效范围之外则将引发*struct.error*。

3.1 版更變: 在之前版本中，某些整数格式包装了超范围的值并会引发*DeprecationWarning* 而不是*struct.error*。

'p' 格式字符用于编码 "Pascal 字符串"，即存储在由计数指定的 固定长度字节中的可变长度短字符串。所存储的第一个字节为字符串长度或 255 中的较小值。之后是字符串对应的字节。如果传入*pack()* 的字符串过长（超过计数值减 1），则只有字符串前 count-1 个字节会被存储。如果字符串短于 count-1，则会填充空字节以使得恰好使用了 count 个字节。请注意对于*unpack()*，'p' 格式字符会消耗 count 个字节，但返回的字符串永远不会包含超过 255 个字节。

对于 '?' 格式字符，返回值为*True* 或*False*。在打包时将会使用参数对象的逻辑值。以本机或标准 bool 类型表示的 0 或 1 将被打包，任何非零值在解包时将为 True。

示例

備𝔼: 所有示例都假定使用一台大端机器的本机字节顺序、大小和对齐方式。

打包/解包三个整数的基础示例:

```
>>> from struct import *
>>> pack('hhl', 1, 2, 3)
b'\x00\x01\x00\x02\x00\x00\x00\x03'
>>> unpack('hhl', b'\x00\x01\x00\x02\x00\x00\x00\x03')
(1, 2, 3)
```

(下页继续)

(繼續上一頁)

```
>>> calcsize('hhl')
8
```

解包的字段可通过将它们赋值给变量或将结果包装为一个具名元组来命名:

```
>>> record = b'raymond   \x32\x12\x08\x01\x08'
>>> name, serialnum, school, gradelevel = unpack('<10sHHb', record)

>>> from collections import namedtuple
>>> Student = namedtuple('Student', 'name serialnum school gradelevel')
>>> Student._make(unpack('<10sHHb', record))
Student(name=b'raymond   ', serialnum=4658, school=264, gradelevel=8)
```

格式字符的顺序可能对大小产生影响，因为满足对齐要求所需的填充是不同的:

```
>>> pack('ci', b'*', 0x12131415)
b'*\x00\x00\x00\x12\x13\x14\x15'
>>> pack('ic', 0x12131415, b'*')
b'\x12\x13\x14\x15*'
>>> calcsize('ci')
8
>>> calcsize('ic')
5
```

以下格式 `'llh0l'` 指定在末尾有两个填充字节，假定 long 类型按 4 个字节的边界对齐:

```
>>> pack('llh0l', 1, 2, 3)
b'\x00\x00\x00\x01\x00\x00\x00\x02\x00\x03\x00\x00'
```

这仅当本机大小和对齐方式生效时才会起作用；标准大小和对齐方式并不会强制进行任何对齐。

也参考:

模块*array* 被打包为二进制存储的同质数据。

模組*xdrlib* 打包和解包 XDR 数据。

7.1.3 类

struct 模块还定义了以下类型:

class struct.**Struct**(*format*)

返回一个新的 Struct 对象，它会根据格式字符串 *format* 来写入和读取二进制数据。一次性地创建 Struct 对象并调用其方法相比使用同样的格式调用*struct* 函数更为高效，因为这样格式字符串只需被编译一次。

備🄵: 传递给*Struct* 和模块层级函数的已编译版最新格式字符串会被缓存，因此只使用少量格式字符串的程序无需担心重用单独的*Struct* 实例。

已编译的 Struct 对象支持以下方法和属性:

pack(*v1*, *v2*, ...)

等价于*pack()* 函数，使用了已编译的格式。(`len(result)` 将等于*size*。)

pack_into(*buffer*, *offset*, *v1*, *v2*, ...)

等价于*pack_into()* 函数，使用了已编译的格式。

unpack(*buffer*)
等价于*unpack()* 函数，使用了已编译的格式。缓冲区的字节大小必须等于*size*。

unpack_from(*buffer*, *offset=0*)
等价于*unpack_from()* 函数，使用了已编译的格式。缓冲区的字节大小从位置 *offset* 开始必须至少为*size*。

iter_unpack(*buffer*)
等价于*iter_unpack()* 函数，使用了已编译的格式。缓冲区的大小必须为*size* 的整数倍。

3.4 版新加入.

format
用于构造此 Struct 对象的格式字符串。

3.7 版更變: 格式字符串类型现在是*str* 而不再是*bytes*。

size
计算出对应于*format* 的结构大小（亦即*pack()* 方法所产生的字节串对象的大小）。

7.2 codecs --- 编解码器注册和相关基类

源代码： Lib/codecs.py

This module defines base classes for standard Python codecs (encoders and decoders) and provides access to the internal Python codec registry, which manages the codec and error handling lookup process. Most standard codecs are *text encodings*, which encode text to bytes (and decode bytes to text), but there are also codecs provided that encode text to text, and bytes to bytes. Custom codecs may encode and decode between arbitrary types, but some module features are restricted to be used specifically with *text encodings* or with codecs that encode to *bytes*.

该模块定义了以下用于使用任何编解码器进行编码和解码的函数：

codecs.**encode**(*obj*, *encoding='utf-8'*, *errors='strict'*)
使用为 *encoding* 注册的编解码器对 *obj* 进行编码。

可以给定 *Errors* 以设置所需要的错误处理方案。默认的错误处理方案 `'strict'` 表示编码错误将引发*ValueError* (或更特定编解码器相关的子类，例如*UnicodeEncodeError*)。请参阅编解码器基类了解有关编解码器错误处理的更多信息。

codecs.**decode**(*obj*, *encoding='utf-8'*, *errors='strict'*)
使用为 *encoding* 注册的编解码器对 *obj* 进行解码。

可以给定 *Errors* 以设置所需要的错误处理方案。默认的错误处理方案 `'strict'` 表示编码错误将引发*ValueError* (或更特定编解码器相关的子类，例如*UnicodeDecodeError*)。请参阅编解码器基类了解有关编解码器错误处理的更多信息。

每种编解码器的完整细节也可以直接查找获取：

codecs.**lookup**(*encoding*)
在 Python 编解码器注册表中查找编解码器信息，并返回一个*CodecInfo* 对象，其定义见下文。

首先将会在注册表缓存中查找编码，如果未找到，则会扫描注册的搜索函数列表。如果没有找到*CodecInfo* 对象，则将引发*LookupError*。否则，*CodecInfo* 对象将被存入缓存并返回给调用者。

class codecs.**CodecInfo**(*encode*, *decode*, *streamreader=None*, *streamwriter=None*, *incrementalencoder=None*, *incrementaldecoder=None*, *name=None*)
查找编解码器注册表所得到的编解码器细节信息。构造器参数将保存为同名的属性：

name
编码名称

encode
decode
无状态的编码和解码函数。它们必须是具有与 Codec 的*encode()* 和*decode()* 方法相同接口的函数或方法 (参见*Codec 接口*)。这些函数或方法应当工作于无状态的模式。

incrementalencoder
incrementaldecoder
增量式的编码器和解码器类或工厂函数。这些函数必须分别提供由基类*IncrementalEncoder* 和*IncrementalDecoder* 所定义的接口。增量式编解码器可以保持状态。

streamwriter
streamreader
流式写入器和读取器类或工厂函数。这些函数必须分别提供由基类*StreamWriter* 和*StreamReader* 所定义的接口。流式编解码器可以保持状态。

为了简化对各种编解码器组件的访问，本模块提供了以下附加函数，它们使用*lookup()* 来执行编解码器查找：

codecs.**getencoder**(*encoding*)
查找给定编码的编解码器并返回其编码器函数。

在编码无法找到时将引发*LookupError*。

codecs.**getdecoder**(*encoding*)
查找给定编码的编解码器并返回其解码器函数。

在编码无法找到时将引发*LookupError*。

codecs.**getincrementalencoder**(*encoding*)
查找给定编码的编解码器并返回其增量式编码器类或工厂函数。

在编码无法找到或编解码器不支持增量式编码器时将引发*LookupError*。

codecs.**getincrementaldecoder**(*encoding*)
查找给定编码的编解码器并返回其增量式解码器类或工厂函数。

在编码无法找到或编解码器不支持增量式解码器时将引发*LookupError*。

codecs.**getreader**(*encoding*)
查找给定编码的编解码器并返回其*StreamReader* 类或工厂函数。

在编码无法找到时将引发*LookupError*。

codecs.**getwriter**(*encoding*)
查找给定编码的编解码器并返回其*StreamWriter* 类或工厂函数。

在编码无法找到时将引发*LookupError*。

自定义编解码器的启用是通过注册适当的编解码器搜索函数：

codecs.**register**(*search_function*)
注册一个编解码器搜索函数。搜索函数预期接收一个参数，即全部以小写字母表示的编码格式名称，其中中连字符和空格会被转换为下划线，并返回一个*CodecInfo* 对象。在搜索函数无法找到给定编码格式的情况下，它应当返回 `None`。

3.9 版更變: 连字符和空格会被转换为下划线。

備𝔽: 搜索函数的注册目前是不可逆的，这在某些情况下可能导致问题，例如单元测试或模块重载等。

虽然内置的*open()* 和相关联的*io* 模块是操作已编码文本文件的推荐方式，但本模块也提供了额外的工具函数和类，允许在操作二进制文件时使用更多各类的编解码器：

codecs.**open**(*filename*, *mode='r'*, *encoding=None*, *errors='strict'*, *buffering=-1*)
使用给定的 *mode* 打开已编码的文件并返回一个*StreamReaderWriter* 的实例，提供透明的编码/解码。默认的文件模式为 `'r'`，表示以读取模式打开文件。

備𝔽: 下层的已编码文件总是以二进制模式打开。在读取和写入时不会自动执行 `'\n'` 的转换。*mode* 参数可以是内置*open()* 函数所接受的任意二进制模式；`'b'` 会被自动添加。

---

encoding 指定文件所要使用的编码格式。允许任何编码为字节串或从字节串解码的编码格式，而文件方法所支持的数据类型则取决于所使用的编解码器。

可以指定 *errors* 来定义错误处理方案。默认值 `'strict'` 表示在出现编码错误时引发*ValueError*。

buffering 的含义与内置*open()* 函数中的相同。默认值 -1 表示将使用默认的缓冲区大小。

codecs.**EncodedFile**(*file*, *data_encoding*, *file_encoding=None*, *errors='strict'*)
返回一个*StreamRecoder* 实例，它提供了 *file* 的透明转码包装版本。当包装版本被关闭时原始文件也会被关闭。

写入已包装文件的数据会根据给定的 *data_encoding* 解码，然后以使用 *file_encoding* 的字节形式写入原始文件。从原始文件读取的字节串将根据 *file_encoding* 解码，其结果将使用 *data_encoding* 进行编码。

如果 *file_encoding* 未给定，则默认为 *data_encoding*。

可以指定 *errors* 来定义错误处理方案。默认值 `'strict'` 表示在出现编码错误时引发*ValueError*。

codecs.**iterencode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)
使用增量式编码器通过迭代来编码由 *iterator* 所提供的输入。此函数属于*generator*。*errors* 参数（以及任何其他关键字参数）会被传递给增量式编码器。

此函数要求编解码器接受*str* 对象形式的文本进行编码。因此它不支持字节到字节的编码器，例如 `base64_codec`。

codecs.**iterdecode**(*iterator*, *encoding*, *errors='strict'*, ***kwargs*)
使用增量式解码器通过迭代来解码由 *iterator* 所提供的输入。此函数属于*generator*。*errors* 参数（以及任何其他关键字参数）会被传递给增量式解码器。

此函数要求编解码器接受*bytes* 对象进行解码。因此它不支持文本到文本的编码器，例如 `rot_13`，但是 `rot_13` 可以通过同样效果的*iterencode()* 来使用。

本模块还提供了以下常量，适用于读取和写入依赖于平台的文件：

codecs.**BOM**
codecs.**BOM_BE**
codecs.**BOM_LE**
codecs.**BOM_UTF8**
codecs.**BOM_UTF16**
codecs.**BOM_UTF16_BE**
codecs.**BOM_UTF16_LE**
codecs.**BOM_UTF32**
codecs.**BOM_UTF32_BE**
codecs.**BOM_UTF32_LE**
这些常量定义了多种字节序列，即一些编码格式的 Unicode 字节顺序标记（BOM）。它们在 UTF-16 和 UTF-32 数据流中被用以指明所使用的字节顺序，并在 UTF-8 中被用作 Unicode 签名。*BOM_UTF16* 是*BOM_UTF16_BE* 或*BOM_UTF16_LE*，具体取决于平台的本机字节顺序，*BOM* 是*BOM_UTF16* 的别名，*BOM_LE* 是*BOM_UTF16_LE* 的别名，*BOM_BE* 是*BOM_UTF16_BE* 的别名。其他序列则表示 UTF-8 和 UTF-32 编码格式中的 BOM。

7.2.1 编解码器基类

codecs 模块定义了一系列基类用来定义配合编解码器对象进行工作的接口，并且也可用作定制编解码器实现的基础。

每种编解码器必须定义四个接口以便用作 Python 中的编解码器：无状态编码器、无状态解码器、流读取器和流写入器。流读取器和写入器通常会重用无状态编码器/解码器来实现文件协议。编解码器作者还需要定义编解码器将如何处理编码和解码错误。

错误处理方案

To simplify and standardize error handling, codecs may implement different error handling schemes by accepting the *errors* string argument:

```
>>> 'German ß, ♬'.encode(encoding='ascii', errors='backslashreplace')
b'German \\xdf, \\u266c'
>>> 'German ß, ♬'.encode(encoding='ascii', errors='xmlcharrefreplace')
b'German &#223;, &#9836;'
```

The following error handlers can be used with all Python 标准编码 codecs:

值	含义
`'strict'`	Raise *UnicodeError* (or a subclass), this is the default. Implemented in *strict_errors()*.
`'ignore'`	忽略错误格式的数据并且不加进一步通知就继续执行。在*ignore_errors()* 中实现。
`'replace'`	Replace with a replacement marker. On encoding, use `?` (ASCII character). On decoding, use `\ufffd` (U+FFFD, the official REPLACEMENT CHARACTER). Implemented in *replace_errors()*.
`'backslashreplace'`	Replace with backslashed escape sequences. On encoding, use hexadecimal form of Unicode code point with formats `\xhh` `\uxxxx` `\Uxxxxxxxx`. On decoding, use hexadecimal form of byte value with format `\xhh`. Implemented in *backslashreplace_errors()*.
`'surrogateescape'`	在解码时，将字节替换为 `U+DC80` 至 `U+DCFF` 范围内的单个代理代码。当在编码数据时使用 `'surrogateescape'` 错误处理方案时，此代理将被转换回相同的字节。（请参阅 **PEP 383** 了解详情。）

The following error handlers are only applicable to encoding (within *text encodings*):

值	含义
`'xmlcharrefreplace'`	Replace with XML/HTML numeric character reference, which is a decimal form of Unicode code point with format `&#num;` Implemented in *xmlcharrefreplace_errors()*.
`'namereplace'`	Replace with `\N{...}` escape sequences, what appears in the braces is the Name property from Unicode Character Database. Implemented in *namereplace_errors()*.

此外，以下错误处理方案被专门用于指定的编解码器：

值	编解码器	含义
`'surrogatepass'`	utf-8, utf-16, utf-32, utf-16-be, utf-16-le, utf-32-be, utf-32-le	Allow encoding and decoding surrogate code point (`U+D800` - `U+DFFF`) as normal code point. Otherwise these codecs treat the presence of surrogate code point in *str* as an error.

3.1 版新加入: `'surrogateescape'` 和 `'surrogatepass'` 错误处理方案。

3.4 版更變: The 'surrogatepass' error handler now works with utf-16* and utf-32* codecs.

3.5 版新加入: 'namereplace' 错误处理方案。

3.5 版更變: The 'backslashreplace' error handler now works with decoding and translating.

允许的值集合可以通过注册新命名的错误处理方案来扩展:

codecs.**register_error**(*name*, *error_handler*)
在名称 *name* 之下注册错误处理函数 *error_handler*。当 *name* 被指定为错误形参时，*error_handler* 参数所指定的对象将在编码和解码期间发生错误的情况下被调用，

对于编码操作，将会调用 *error_handler* 并传入一个*UnicodeEncodeError* 实例，其中包含有关错误位置的信息。错误处理程序必须引发此异常或别的异常，或者也可以返回一个元组，其中包含输入的不可编码部分的替换对象，以及应当继续进行编码的位置。替换对象可以为*str* 或*bytes* 类型。如果替换对象为字节串，编码器将简单地将其复制到输出缓冲区。如果替换对象为字符串，编码器将对替换对象进行编码。对原始输入的编码操作会在指定位置继续进行。负的位置值将被视为相对于输入字符串的末尾。如果结果位置超出范围则将引发*IndexError*。

解码和转换的做法很相似，不同之处在于将把*UnicodeDecodeError* 或*UnicodeTranslateError* 传给处理程序，并且来自错误处理程序的替换对象将被直接放入输出。

之前注册的错误处理方案（包括标准错误处理方案）可通过名称进行查找:

codecs.**lookup_error**(*name*)
返回之前在名称 *name* 之下注册的错误处理方案。

在处理方案无法找到时将引发*LookupError*。

以下标准错误处理方案也可通过模块层级函数的方式来使用:

codecs.**strict_errors**(*exception*)
Implements the 'strict' error handling.

Each encoding or decoding error raises a *UnicodeError*.

codecs.**ignore_errors**(*exception*)
Implements the 'ignore' error handling.

Malformed data is ignored; encoding or decoding is continued without further notice.

codecs.**replace_errors**(*exception*)
Implements the 'replace' error handling.

Substitutes ? (ASCII character) for encoding errors or ￼ (U+FFFD, the official REPLACEMENT CHARACTER) for decoding errors.

codecs.**backslashreplace_errors**(*exception*)
Implements the 'backslashreplace' error handling.

Malformed data is replaced by a backslashed escape sequence. On encoding, use the hexadecimal form of Unicode code point with formats \xhh \uxxxx \Uxxxxxxxx. On decoding, use the hexadecimal form of byte value with format \xhh.

3.5 版更變: Works with decoding and translating.

codecs.**xmlcharrefreplace_errors**(*exception*)
Implements the 'xmlcharrefreplace' error handling (for encoding within *text encoding* only).

The unencodable character is replaced by an appropriate XML/HTML numeric character reference, which is a decimal form of Unicode code point with format &#num; .

codecs.**namereplace_errors**(*exception*)
Implements the 'namereplace' error handling (for encoding within *text encoding* only).

The unencodable character is replaced by a `\N{...}` escape sequence. The set of characters that appear in the braces is the Name property from Unicode Character Database. For example, the German lowercase letter `'ß'` will be converted to byte sequence `\N{LATIN SMALL LETTER SHARP S}`.

3.5 版新加入.

无状态的编码和解码

基本 `Codec` 类定义了这些方法，同时还定义了无状态编码器和解码器的函数接口：

`Codec.encode`(*input*, *errors='strict'*)

编码 *input* 对象并返回一个元组 (输出对象, 消耗长度)。例如，*text encoding* 会使用特定的字符集编码格式 (例如 `cp1252` 或 `iso-8859-1`) 将字符串转换为字节串对象。

errors 参数定义了要应用的错误处理方案。默认为 `'strict'` 处理方案。

此方法不一定会在 `Codec` 实例中保存状态。可使用必须保存状态的*StreamWriter* 作为编解码器以便高效地进行编码。

编码器必须能够处理零长度的输入并在此情况下返回输出对象类型的空对象。

`Codec.decode`(*input*, *errors='strict'*)

解码 *input* 对象并返回一个元组 (输出对象, 消耗长度)。例如，*text encoding* 的解码操作会使用特定的字符集编码格式将字节串对象转换为字符串对象。

对于文本编码格式和字节到字节编解码器，*input* 必须为一个字节串对象或提供了只读缓冲区接口的对象 -- 例如，缓冲区对象和映射到内存的文件。

errors 参数定义了要应用的错误处理方案。默认为 `'strict'` 处理方案。

此方法不一定会在 `Codec` 实例中保存状态。可使用必须保存状态的*StreamReader* 作为编解码器以便高效地进行解码。

解码器必须能够处理零长度的输入并在此情况下返回输出对象类型的空对象。

增量式的编码和解码

IncrementalEncoder 和*IncrementalDecoder* 类提供了增量式编码和解码的基本接口。对输入的编码/解码不是通过对无状态编码器/解码器的一次调用，而是通过对增量式编码器/解码器的*encode()*/*decode()* 方法的多次调用。增量式编码器/解码器会在方法调用期间跟踪编码/解码过程。

调用*encode()*/*decode()* 方法后的全部输出相当于将所有通过无状态编码器/解码器进行编码/解码的单个输入连接在一起所得到的输出。

IncrementalEncoder 对象

IncrementalEncoder 类用来对一个输入进行分步编码。它定义了以下方法，每个增量式编码器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**IncrementalEncoder**(*errors='strict'*)

IncrementalEncoder 实例的构造器。

所有增量式编码器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

IncrementalEncoder 可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。可用的值请参阅错误处理方案。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*IncrementalEncoder* 对象的生命期内在不同的错误处理策略之间进行切换。

encode(*object*, *final=False*)
编码 *object* (会将编码器的当前状态纳入考虑) 并返回已编码的结果对象。如果这是对*encode()* 的最终调用则 *final* 必须为真值（默认为假值）。

reset()
将编码器重置为初始状态。输出将被丢弃：调用 `.encode(object, final=True)`，在必要时传入一个空字节串或字符串，重置编码器并得到输出。

getstate()
返回编码器的当前状态，该值必须为一个整数。实现应当确保 `0` 是最常见的状态。(比整数更复杂的状态表示可以通过编组/选择状态并将结果字符串的字节数据编码为整数来转换为一个整数值)。

setstate(*state*)
将编码器的状态设为 *state*。*state* 必须为*getstate()* 所返回的一个编码器状态。

IncrementalDecoder 对象

IncrementalDecoder 类用来对一个输入进行分步解码。它定义了以下方法，每个增量式解码器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**IncrementalDecoder**(*errors='strict'*)
IncrementalDecoder 实例的构造器。

所有增量式解码器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

IncrementalDecoder 可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。可用的值请参阅错误处理方案。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*IncrementalDecoder* 对象的生命期内在不同的错误处理策略之间进行切换。

decode(*object*, *final=False*)
解码 *object* (会将解码器的当前状态纳入考虑) 并返回已解码的结果对象。如果这是对*decode()* 的最终调用则 *final* 必须为真值（默认为假值）。如果 *final* 为真值则解码器必须对输入进行完全解码并且必须刷新所有缓冲区。如果这无法做到（例如由于在输入结束时字节串序列不完整）则它必须像在无状态的情况下那样初始化错误处理（这可能引发一个异常）。

reset()
将解码器重置为初始状态。

getstate()
返回解码器的当前状态。这必须为一个二元组，第一项必须是包含尚未解码的输入的缓冲区。第二项必须为一个整数，可以表示附加状态信息。(实现应当确保 `0` 是最常见的附加状态信息。) 如果此附加状态信息为 `0` 则必须可以将解码器设为没有已缓冲输入并且以 `0` 作为附加状态信息，以便将先前已缓冲的输入馈送到解码器使其返回到先前的状态而不产生任何输出。(比整数更复杂的附加状态信息可以通过编组/选择状态信息并将结果字符串的字节数据编码为整数来转换为一个整数值。)

setstate(*state*)
将解码器的状态设为 *state*。*state* 必须为*getstate()* 所返回的一个解码器状态。

流式的编码和解码

StreamWriter 和*StreamReader* 类提供了一些泛用工作接口，可被用来非常方便地实现新的编码格式子模块。请参阅 `encodings.utf_8` 中的示例了解如何做到这一点。

StreamWriter 对象

StreamWriter 类是 `Codec` 的子类，它定义了以下方法，每个流式写入器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**StreamWriter**(*stream*, *errors='strict'*)

StreamWriter 实例的构造器。

所有流式写入器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

stream 参数必须为一个基于特定编解码器打开用于写入文本或二进制数据的文件类对象。

StreamWriter 可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。请参阅*错误处理方案* 了解下层的流式编解码器可支持的标准错误处理方案。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*StreamWriter* 对象的生命期内在不同的错误处理策略之间进行切换。

write(*object*)

将编码后的对象内容写入到流。

writelines(*list*)

Writes the concatenated iterable of strings to the stream (possibly by reusing the *write()* method). Infinite or very large iterables are not supported. The standard bytes-to-bytes codecs do not support this method.

reset()

重置用于保持内部状态的编解码器缓冲区。

调用此方法应当确保在干净的状态下放入输出数据，以允许直接添加新的干净数据而无须重新扫描整个流来恢复状态。

除了上述的方法，*StreamWriter* 还必须继承来自下层流的所有其他方法和属性。

StreamReader 对象

StreamReader 类是 `Codec` 的子类，它定义了以下方法，每个流式读取器都必须定义这些方法以便与 Python 编解码器注册表相兼容。

class codecs.**StreamReader**(*stream*, *errors='strict'*)

StreamReader 实例的构造器。

所有流式读取器必须提供此构造器接口。它们可以自由地添加额外的关键字参数，但只有在这里定义的参数才会被 Python 编解码器注册表所使用。

stream 参数必须为一个基于特定编解码器打开用于读取文本或二进制数据的文件类对象。

StreamReader 可以通过提供 *errors* 关键字参数来实现不同的错误处理方案。请参阅*错误处理方案* 了解下层的流式编解码器可支持的标准错误处理方案。

errors 参数将被赋值给一个同名的属性。通过对此属性赋值就可以在*StreamReader* 对象的生命期内在不同的错误处理策略之间进行切换。

errors 参数所允许的值集合可以使用*register_error()* 来扩展。

read(*size=-1*, *chars=-1*, *firstline=False*)
解码来自流的数据并返回结果对象。

chars 参数指明要返回的解码后码位或字节数量。*read()* 方法绝不会返回超出请求数量的数据，但如果可用数量不足，它可能返回少于请求数量的数据。

size 参数指明要读取并解码的已编码字节或码位的最大数量近似值。解码器可以适当地修改此设置。默认值 -1 表示尽可能多地读取并解码。此形参的目的是防止一次性解码过于巨大的文件。

firstline 旗标指明如果在后续行发生解码错误，则仅返回第一行就足够了。

此方法应当使用“贪婪”读取策略，这意味着它应当在编码格式定义和给定大小所允许的情况下尽可能多地读取数据，例如，如果在流上存在可选的编码结束或状态标记，这些内容也应当被读取。

readline(*size=None*, *keepends=True*)
从输入流读取一行并返回解码后的数据。

如果给定了 *size*，则将其作为 size 参数传递给流的*read()* 方法。

如果 *keepends* 为假值，则行结束符将从返回的行中去除。

readlines(*sizehint=None*, *keepends=True*)
从输入流读取所有行并将其作为一个行列表返回。

行结束符会使用编解码器的*decode()* 方法来实现，并且如果 *keepends* 为真值则会将其包含在列表条目中。

如果给定了 *sizehint*，则将其作为 *size* 参数传递给流的*read()* 方法。

reset()
重置用于保持内部状态的编解码器缓冲区。

请注意不应当对流进行重定位。使用此方法的主要目的是为了能够从解码错误中恢复。

除了上述的方法，*StreamReader* 还必须继承来自下层流的所有其他方法和属性。

StreamReaderWriter 对象

StreamReaderWriter 是一个方便的类，允许对同时工作于读取和写入模式的流进行包装。

其设计使得开发者可以使用*lookup()* 函数所返回的工厂函数来构造实例。

class codecs.**StreamReaderWriter**(*stream*, *Reader*, *Writer*, *errors='strict'*)
创建一个*StreamReaderWriter* 实例。*stream* 必须为一个文件类对象。*Reader* 和 *Writer* 必须为分别提供了*StreamReader* 和*StreamWriter* 接口的工厂函数或类。错误处理通过与流式读取器和写入器所定义的相同方式来完成。

StreamReaderWriter 实例定义了*StreamReader* 和*StreamWriter* 类的组合接口。它们还继承了来自下层流的所有其他方法和属性。

StreamRecoder 对象

StreamRecoder 将数据从一种编码格式转换为另一种，这对于处理不同编码环境的情况有时会很有用。

其设计使得开发者可以使用*lookup()* 函数所返回的工厂函数来构造实例。

class codecs.**StreamRecoder**(*stream*, *encode*, *decode*, *Reader*, *Writer*, *errors='strict'*)

创建一个实现了双向转换的*StreamRecoder* 实例: *encode* 和 *decode* 工作于前端—对代码可见的数据调用 read() 和 write()，而 *Reader* 和 *Writer* 工作于后端—*stream* 中的数据。

你可以使用这些对象来进行透明转码，例如从 Latin-1 转为 UTF-8 以及反向转换。

stream 参数必须为一个文件类对象。

encode 和 *decode* 参数必须遵循 Codec 接口。*Reader* 和 *Writer* 必须为分别提供了*StreamReader* 和*StreamWriter* 接口对象的工厂函数或类。

错误处理通过与流式读取器和写入器所定义的相同方式来完成。

StreamRecoder 实例定义了*StreamReader* 和*StreamWriter* 类的组合接口。它们还继承了来自下层流的所有其他方法和属性。

7.2.2 编码格式与 Unicode

Strings are stored internally as sequences of code points in range U+0000--U+10FFFF. (See **PEP 393** for more details about the implementation.) Once a string object is used outside of CPU and memory, endianness and how these arrays are stored as bytes become an issue. As with other codecs, serialising a string into a sequence of bytes is known as *encoding*, and recreating the string from the sequence of bytes is known as *decoding*. There are a variety of different text serialisation codecs, which are collectivity referred to as *text encodings*.

最简单的文本编码格式 (称为 'latin-1' 或 'iso-8859-1') 将码位 0--255 映射为字节值 0x0--0xff，这意味着包含 U+00FF 以上码位的字符串对象无法使用此编解码器进行编码。这样做将引发*UnicodeEncodeError*，其形式类似下面这样（不过详细的错误信息可能会有所不同）：UnicodeEncodeError: 'latin-1' codec can't encode character '\u1234' in position 3: ordinal not in range(256)。

还有另外一组编码格式（所谓的字符映射编码）会选择全部 Unicode 码位的不同子集并设定如何将这些码位映射为字节值 0x0--0xff。要查看这是如何实现的，只需简单地打开相应源码例如 encodings/cp1252.py (这是一个主要在 Windows 上使用的编码格式)。其中会有一个包含 256 个字符的字符串常量，指明每个字符所映射的字节值。

All of these encodings can only encode 256 of the 1114112 code points defined in Unicode. A simple and straightforward way that can store each Unicode code point, is to store each code point as four consecutive bytes. There are two possibilities: store the bytes in big endian or in little endian order. These two encodings are called UTF-32-BE and UTF-32-LE respectively. Their disadvantage is that if e.g. you use UTF-32-BE on a little endian machine you will always have to swap bytes on encoding and decoding. UTF-32 avoids this problem: bytes will always be in natural endianness. When these bytes are read by a CPU with a different endianness, then bytes have to be swapped though. To be able to detect the endianness of a UTF-16 or UTF-32 byte sequence, there's the so called BOM ("Byte Order Mark"). This is the Unicode character U+FEFF. This character can be prepended to every UTF-16 or UTF-32 byte sequence. The byte swapped version of this character (0xFFFE) is an illegal character that may not appear in a Unicode text. So when the first character in a UTF-16 or UTF-32 byte sequence appears to be a U+FFFE the bytes have to be swapped on decoding. Unfortunately the character U+FEFF had a second purpose as a ZERO WIDTH NO-BREAK SPACE: a character that has no width and doesn't allow a word to be split. It can e.g. be used to give hints to a ligature algorithm. With Unicode 4.0 using U+FEFF as a ZERO WIDTH NO-BREAK SPACE has been deprecated (with U+2060 (WORD JOINER) assuming this role). Nevertheless Unicode software still must be able to handle U+FEFF in both roles: as a BOM it's a device to determine the storage layout of the encoded bytes, and vanishes once the byte sequence has been decoded into a string; as a ZERO WIDTH NO-BREAK SPACE it's a normal character that will be decoded like any other.

There's another encoding that is able to encode the full range of Unicode characters: UTF-8. UTF-8 is an 8-bit encoding, which means there are no issues with byte order in UTF-8. Each byte in a UTF-8 byte sequence consists of two parts: marker bits (the most significant bits) and payload bits. The marker bits are a sequence of zero to four `1` bits followed by a `0` bit. Unicode characters are encoded like this (with x being payload bits, which when concatenated give the Unicode character):

范围	编码
`U-00000000` ... `U-0000007F`	0xxxxxxx
`U-00000080` ... `U-000007FF`	110xxxxx 10xxxxxx
`U-00000800` ... `U-0000FFFF`	1110xxxx 10xxxxxx 10xxxxxx
`U-00010000` ... `U-0010FFFF`	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Unicode 字符最不重要的一个位就是最右侧的二进制位 x。

由于 UTF-8 是一种 8 位编码格式，因此 BOM 是不必要的，并且已编码字符串中的任何 `U+FEFF` 字符（即使是作为第一个字符）都会被视为是 `ZERO WIDTH NO-BREAK SPACE`。

Without external information it's impossible to reliably determine which encoding was used for encoding a string. Each charmap encoding can decode any random byte sequence. However that's not possible with UTF-8, as UTF-8 byte sequences have a structure that doesn't allow arbitrary byte sequences. To increase the reliability with which a UTF-8 encoding can be detected, Microsoft invented a variant of UTF-8 (that Python calls `"utf-8-sig"`) for its Notepad program: Before any of the Unicode characters is written to the file, a UTF-8 encoded BOM (which looks like this as a byte sequence: `0xef`, `0xbb`, `0xbf`) is written. As it's rather improbable that any charmap encoded file starts with these byte values (which would e.g. map to

    LATIN SMALL LETTER I WITH DIAERESIS
    RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
    INVERTED QUESTION MARK

对于 iso-8859-1 编码格式来说），这提升了根据字节序列来正确猜测 `utf-8-sig` 编码格式的成功率。所以在这里 BOM 的作用并不是帮助确定生成字节序列所使用的字节顺序，而是作为帮助猜测编码格式的记号。在进行编码时 utf-8-sig 编解码器将把 `0xef`, `0xbb`, `0xbf` 作为头三个字节写入文件。在进行解码时 `utf-8-sig` 将跳过这三个字节，如果它们作为文件的头三个字节出现的话。在 UTF-8 中并不推荐使用 BOM，通常应当避免它们的出现。

7.2.3 标准编码

Python 自带了许多内置的编解码器，它们的实现或者是通过 C 函数，或者是通过映射表。以下表格是按名称排序的编解码器列表，并提供了一些常见别名以及编码格式通常针对的语言。别名和语言列表都不是详尽无遗的。请注意仅有大小写区别或使用连字符替代下划线的拼写形式也都是有效的别名；因此，`'utf-8'` 是 `'utf_8'` 编解码器的有效别名。

CPython implementation detail: 有些常见编码格式可以绕过编解码器查找机制来提升性能。这些优化机会对于 CPython 来说仅能通过一组有限的别名（大小写不敏感）来识别：utf-8, utf8, latin-1, latin1, iso-8859-1, iso8859-1, mbcs (Windows 专属), ascii, us-ascii, utf-16, utf16, utf-32, utf32, 也包括使用下划线替代连字符的的形式。使用这些编码格式的其他别名可能会导致更慢的执行速度。

3.6 版更變: 可识别针对 us-ascii 的优化机会。

许多字符集都支持相同的语言。它们在个别字符（例如是否支持 EURO SIGN 等）以及给字符所分配的码位方面存在差异。特别是对于欧洲语言来说，通常存在以下几种变体：

- 某个 ISO 8859 编码集
- 某个 Microsoft Windows 编码页，通常是派生自某个 8859 编码集，但会用附加的图形字符来替换控制字符。

- 某个 IBM EBCDIC 编码页
- 某个 IBM PC 编码页，通常会兼容 ASCII

编码	别名	语言
ascii	646, us-ascii	英语
big5	big5-tw, csbig5	繁体中文
big5hkscs	big5-hkscs, hkscs	繁体中文
cp037	IBM037, IBM039	英语
cp273	273, IBM273, csIBM273	德语 3.4 版新加入.
cp424	EBCDIC-CP-HE, IBM424	希伯来语
cp437	437, IBM437	英语
cp500	EBCDIC-CP-BE, EBCDIC-CP-CH, IBM500	西欧
cp720		阿拉伯语
cp737		希腊语
cp775	IBM775	波罗的海语言
cp850	850, IBM850	西欧
cp852	852, IBM852	中欧和东欧
cp855	855, IBM855	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
cp856		希伯来语
cp857	857, IBM857	土耳其语
cp858	858, IBM858	西欧
cp860	860, IBM860	葡萄牙语
cp861	861, CP-IS, IBM861	冰岛语
cp862	862, IBM862	希伯来语
cp863	863, IBM863	加拿大语
cp864	IBM864	阿拉伯语
cp865	865, IBM865	丹麦语/挪威语
cp866	866, IBM866	俄语
cp869	869, CP-GR, IBM869	希腊语
cp874		泰语
cp875		希腊语
cp932	932, ms932, mskanji, ms-kanji	日语
cp949	949, ms949, uhc	韩语
cp950	950, ms950	繁体中文
cp1006		乌尔都语
cp1026	ibm1026	土耳其语
cp1125	1125, ibm1125, cp866u, ruscii	乌克兰语 3.4 版新加入.
cp1140	ibm1140	西欧
cp1250	windows-1250	中欧和东欧
cp1251	windows-1251	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
cp1252	windows-1252	西欧
cp1253	windows-1253	希腊语
cp1254	windows-1254	土耳其语
cp1255	windows-1255	希伯来语
cp1256	windows-1256	阿拉伯语
cp1257	windows-1257	波罗的海语言

繼續下一頁

表 1 – 繼續上一頁

编码	别名	语言
cp1258	windows-1258	越南语
euc_jp	eucjp, ujis, u-jis	日语
euc_jis_2004	jisx0213, eucjis2004	日语
euc_jisx0213	eucjisx0213	日语
euc_kr	euckr, korean, ksc5601, ks_c-5601, ks_c-5601-1987, ksx1001, ks_x-1001	韩语
gb2312	chinese, csiso58gb231280, euc-cn, euccn, eucgb2312-cn, gb2312-1980, gb2312-80, iso-ir-58	简体中文
gbk	936, cp936, ms936	统一汉语
gb18030	gb18030-2000	统一汉语
hz	hzgb, hz-gb, hz-gb-2312	简体中文
iso2022_jp	csiso2022jp, iso2022jp, iso-2022-jp	日语
iso2022_jp_1	iso2022jp-1, iso-2022-jp-1	日语
iso2022_jp_2	iso2022jp-2, iso-2022-jp-2	日语，韩语，简体中文，西欧，希腊语
iso2022_jp_2004	iso2022jp-2004, iso-2022-jp-2004	日语
iso2022_jp_3	iso2022jp-3, iso-2022-jp-3	日语
iso2022_jp_ext	iso2022jp-ext, iso-2022-jp-ext	日语
iso2022_kr	csiso2022kr, iso2022kr, iso-2022-kr	韩语
latin_1	iso-8859-1, iso8859-1, 8859, cp819, latin, latin1, L1	西欧
iso8859_2	iso-8859-2, latin2, L2	中欧和东欧
iso8859_3	iso-8859-3, latin3, L3	世界语，马耳他语
iso8859_4	iso-8859-4, latin4, L4	波罗的海语言
iso8859_5	iso-8859-5, cyrillic	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
iso8859_6	iso-8859-6, arabic	阿拉伯语
iso8859_7	iso-8859-7, greek, greek8	希腊语
iso8859_8	iso-8859-8, hebrew	希伯来语
iso8859_9	iso-8859-9, latin5, L5	土耳其语
iso8859_10	iso-8859-10, latin6, L6	北欧语言
iso8859_11	iso-8859-11, thai	泰语
iso8859_13	iso-8859-13, latin7, L7	波罗的海语言
iso8859_14	iso-8859-14, latin8, L8	凯尔特语
iso8859_15	iso-8859-15, latin9, L9	西欧
iso8859_16	iso-8859-16, latin10, L10	东南欧
johab	cp1361, ms1361	韩语
koi8_r		俄语
koi8_t		塔吉克 3.5 版新加入.
koi8_u		乌克兰语
kz1048	kz_1048, strk1048_2002, rk1048	哈萨克语 3.5 版新加入.
mac_cyrillic	maccyrillic	保加利亚语，白俄罗斯语，马其顿语，俄语，塞尔维亚语
mac_greek	macgreek	希腊语

繼續下一頁

表 1 – 繼續上一頁

编码	别名	语言
mac_iceland	maciceland	冰岛语
mac_latin2	maclatin2, maccentraleurope, mac_centeuro	中欧和东欧
mac_roman	macroman, macintosh	西欧
mac_turkish	macturkish	土耳其语
ptcp154	csptcp154, pt154, cp154, cyrillic-asian	哈萨克语
shift_jis	csshiftjis, shiftjis, sjis, s_jis	日语
shift_jis_2004	shiftjis2004, sjis_2004, sjis2004	日语
shift_jisx0213	shiftjisx0213, sjisx0213, s_jisx0213	日语
utf_32	U32, utf32	所有语言
utf_32_be	UTF-32BE	所有语言
utf_32_le	UTF-32LE	所有语言
utf_16	U16, utf16	所有语言
utf_16_be	UTF-16BE	所有语言
utf_16_le	UTF-16LE	所有语言
utf_7	U7, unicode-1-1-utf-7	所有语言
utf_8	U8, UTF, utf8, cp65001	所有语言
utf_8_sig		所有语言

3.4 版更變: utf-16* 和 utf-32* 编码器将不再允许编码代理码位 (`U+D800--U+DFFF`)。utf-32* 解码器将不再解码与代理码位相对应的字节序列。

3.8 版更變: `cp65001` 现在是 `utf_8` 的一个别名。

7.2.4 Python 专属的编码格式

有一些预定义编解码器是 Python 专属的，因此它们在 Python 之外没有意义。这些编解码器按其所预期的输入和输出类型在下表中列出（请注意虽然文本编码是编解码器最常见的使用场景，但下层的编解码器架构支持任意数据转换而不仅是文本编码）。对于非对称编解码器，该列描述的含义是编码方向。

文字编码

以下编解码器提供了`str`到`bytes`的编码和*bytes-like object*到`str`的解码，类似于 Unicode 文本编码。

编码	别名	含义
idna		实现 **RFC 3490**，另请参阅*encodings.idna*。仅支持 `errors='strict'`。
mbcs	ansi, dbcs	Windows 专属：根据 ANSI 代码页（CP_ACP）对操作数进行编码。
oem		Windows 专属：根据 OEM 代码页（CP_OEMCP）对操作数进行编码。 3.6 版新加入.
palmos		PalmOS 3.5 的编码格式
punycode		实现 **RFC 3492**。不支持有状态编解码器。
raw_unicode_escape		Latin-1 编码格式附带对其他码位以 `\uXXXX` 和 `\UXXXXXXXX` 进行编码。现有反斜杠不会以任何方式转义。它被用于 Python 的 pickle 协议。
undefined		所有转换都将引发异常，甚至对空字符串也不例外。错误处理方案会被忽略。
unicode_escape		适合用于以 ASCII 编码的 Python 源代码中的 Unicode 字面值内容的编码格式，但引号不会被转义。对 Latin-1 源代码进行解码。请注意 Python 源代码实际上默认使用 UTF-8。

3.8 版更變: ”unicode_internal” 编解码器已被移除。

二进制转换

以下编解码器提供了二进制转换: *bytes-like object* 到*bytes* 的映射。它们不被*bytes.decode()* 所支持（该方法只生成*str* 类型的输出）。

编码	别名	含义	编码器/解码器
base64_codec[1]	base64, base_64	将操作数转换为多行 MIME base64 (结果总是包含一个末尾的 '\n') 3.4 版更變: 接受任意*bytes-like object* 作为输入用于编码和解码	*base64.encodebytes()* / *base64.decodebytes()*
bz2_codec	bz2	使用 bz2 压缩操作数	*bz2.compress()* / *bz2.decompress()*
hex_codec	hex	将操作数转换为十六进制表示，每个字节有两位数	*binascii.b2a_hex()* / *binascii.a2b_hex()*
quopri_codec	quopri, quoted-printable, quoted_printable	将操作数转换为 MIME 带引号的可打印数据	*quopri.encode()* 且 quotetabs=True / *quopri.decode()*
uu_codec	uu	使用 uuencode 转换操作数	*uu.encode()* / *uu.decode()*
zlib_codec	zip, zlib	使用 gzip 压缩操作数	*zlib.compress()* / *zlib.decompress()*

3.2 版新加入: 恢复二进制转换。

3.4 版更變: 恢复二进制转换的别名。

文字转换

以下编解码器提供了文本转换: *str* 到*str* 的映射。它不被*str.encode()* 所支持（该方法只生成*bytes* 类型的输出）。

编码	别名	含义
rot_13	rot13	返回操作数的凯撒密码加密结果

3.2 版新加入: 恢复 rot_13 文本转换。

3.4 版更變: 恢复 rot13 别名。

7.2.5 encodings.idna --- 应用程序中的国际化域名

此模块实现了 **RFC 3490** (应用程序中的国际化域名) 和 **RFC 3492** (Nameprep: 用于国际化域名 (IDN) 的 Stringprep 配置文件)。它是在 punycode 编码格式和*stringprep* 的基础上构建的。

If you need the IDNA 2008 standard from **RFC 5891** and **RFC 5895**, use the third-party idna module.

这些 RFC 共同定义了一个在域名中支持非 ASCII 字符的协议。一个包含非 ASCII 字符的域名 (例如 www.Alliancefrançaise.nu) 会被转换为兼容 ASCII 的编码格式 (简称 ACE，例如 www.xn--alliancefranaise-npb.nu)。随后此域名的 ACE 形式可以用于所有由于特定协议而不允许使用任意字符的场合，例如 DNS 查询，HTTP *Host* 字段等等。此转换是在应用中进行的；如有可能将对用户可见：应用应当透明地将 Unicode 域名标签转换为线上的 IDNA，并在 ACE 标签被呈现给用户之前将其转换回 Unicode。

Python 以多种方式支持这种转换: idna 编解码器执行 Unicode 和 ACE 之间的转换，基于在 **section 3.1 of RFC 3490** 中定义的分隔字符将输入字符串拆分为标签，再根据需要将每个标签转换为 ACE，相反地又会基于 . 分

[1] 除了字节类对象，'base64_codec' 也接受仅包含 ASCII 的*str* 实例用于解码

隔符将输入字节串拆分为标签，再将找到的任何 ACE 标签转换为 Unicode。此外，*socket* 模块可透明地将 Unicode 主机名转换为 ACE，以便应用在将它们传给 socket 模块时无须自行转换主机名。除此之外，许多包含以主机名作为函数参数的模块例如*http.client* 和*ftplib* 都接受 Unicode 主机名（并且*http.client* 也会在 *Host* 字段中透明地发送 IDNA 主机名，如果它需要发送该字段的话）。

当从线路接收主机名时（例如反向名称查找），到 Unicode 的转换不会自动被执行：希望向用户提供此种主机名的应用应当将它们解码为 Unicode。

encodings.idna 模块还实现了 nameprep 过程，该过程会对主机名执行特定的规范化操作，以实现国际域名的大小写不敏感特性与合并相似的字符。如果有需要可以直接使用 nameprep 函数。

encodings.idna.**nameprep**(*label*)
返回 *label* 经过名称处理操作的版本。该实现目前基于查询字符串，因此 AllowUnassigned 为真值。

encodings.idna.**ToASCII**(*label*)
将标签转换为 ASCII，规则定义见 **RFC 3490**。UseSTD3ASCIIRules 预设为假值。

encodings.idna.**ToUnicode**(*label*)
将标签转换为 Unicode，规则定义见 **RFC 3490**。

7.2.6 encodings.mbcs --- Windows ANSI 代码页

此模块实现 ANSI 代码页（CP_ACP）。

Availability: 仅 Windows 可用

3.3 版更變: 支持任何错误处理

3.2 版更變: 在 3.2 版之前，*errors* 参数会被忽略；总是会使用 'replace' 进行编码，并使用 'ignore' 进行解码。

7.2.7 encodings.utf_8_sig --- 带 BOM 签名的 UTF-8 编解码器

此模块实现了 UTF-8 编解码器的一个变种：在编码时将把 UTF-8 已编码 BOM 添加到 UTF-8 编码字节数据的开头。对于有状态编码器此操作只执行一次（当首次写入字节流时）。在解码时将跳过数据开头作为可选项的 UTF-8 已编码 BOM。

CHAPTER 8

数据类型

本章所描述的模块提供了许多专门的数据类型，如日期和时间、固定类型的数组、堆队列、双端队列、以及枚举。

Python 也提供一些内置数据类型，特别是，*dict*、*list*、*set*、*frozenset*、以及*tuple*。*str* 这个类是用来存储 Unicode 字符串的，而*bytes* 和*bytearray* 这两个类是用来存储二进制数据的。

本章包含以下模块的文档:

8.1 datetime --- 基本日期和时间类型

源代码： Lib/datetime.py

datetime 模块提供用于处理日期和时间的类。

在支持日期时间数学运算的同时，实现的关注点更着重于如何能够更有效地解析其属性用于格式化输出和数据操作。

也参考:

模块*calendar* 通用日历相关函数

模块*time* 时间的访问和转换

Module *zoneinfo* Concrete time zones representing the IANA time zone database.

dateutil 包 具有扩展时区和解析支持的第三方库。

8.1.1 感知型对象和简单型对象

日期和时间对象可以根据它们是否包含时区信息而分为“感知型”和“简单型”两类。

充分掌握应用性算法和政治性时间调整信息例如时区和夏令时的情况下，一个 **感知型**对象就能相对于其他感知型对象来精确定位自身时间点。感知型对象是用来表示一个没有解释空间的固定时间点。[1]

简单型对象没有包含足够多的信息来无歧义地相对于其他 date/time 对象来定位自身时间点。不论一个简单型对象所代表的是世界标准时间（UTC）、当地时间还是某个其他时区的时间完全取决于具体程序，就像一个特定数字所代表的是米、英里还是质量完全取决于具体程序一样。简单型对象更易于理解和使用，代价则是忽略了某些现实性考量。

对于要求感知型对象的应用，*datetime* 和*time* 对象具有一个可选的时区信息属性 `tzinfo`，它可被设为抽象类*tzinfo* 的子类的一个实例。这些*tzinfo* 对象会捕获与 UTC 时间的差值、时区名称以及夏令时是否生效等信息。

datetime 模块只提供了一个具体的*tzinfo* 类，即*timezone* 类。*timezone* 类可以表示具有相对于 UTC 的固定时差的简单时区，例如 UTC 本身或北美的 EST 和 EDT 时区等。支持时间的详细程度取决于具体的应用。世界各地的时间调整规则往往是政治性多于合理性，经常会发生变化，除了 UTC 之外并没有一个能适合所有应用的标准。

8.1.2 常量

The *datetime* module exports the following constants:

`datetime.`**`MINYEAR`**
: *date* 或者*datetime* 对象允许的最小年份。常量*MINYEAR* 是 `1` 。

`datetime.`**`MAXYEAR`**
: *date* 或*datetime* 对象允许最大的年份。常量*MAXYEAR* 是 `9999` 。

8.1.3 有效的类型

class `datetime.`**`date`**
: 一个理想化的简单型日期，它假设当今的公历在过去和未来永远有效。属性: *year*, *month*, and *day*。

class `datetime.`**`time`**
: 一个独立于任何特定日期的理想化时间，它假设每一天都恰好等于 24*60*60 秒。(这里没有“闰秒”的概念。）包含属性: *hour*, *minute*, *second*, *microsecond* 和*tzinfo*。

class `datetime.`**`datetime`**
: 日期和时间的结合。属性： *year*, *month*, *day*, *hour*, *minute*, *second*, *microsecond*, and *tzinfo*.

class `datetime.`**`timedelta`**
: 表示两个*date* 对象或者*time* 对象, 或者*datetime* 对象之间的时间间隔，精确到微秒。

class `datetime.`**`tzinfo`**
: 一个描述时区信息对象的抽象基类。用来给*datetime* 和*time* 类提供自定义的时间调整概念（例如处理时区和/或夏令时）。

class `datetime.`**`timezone`**
: 一个实现了*tzinfo* 抽象基类的子类，用于表示相对于世界标准时间（UTC）的偏移量。

  3.2 版新加入.

[1] 就是说如果我们忽略相对论效应的话。

这些类型的对象都是不可变的。

子类关系

```
object
    timedelta
    tzinfo
        timezone
    time
    date
        datetime
```

通用的特征属性

date, *datetime*, *time* 和*timezone* 类型共享这些通用特性:

- 这些类型的对象都是不可变的。
- 这些类型的对象是可哈希的，这意味着它们可被作为字典的键。
- 这些类型的对象支持通过*pickle* 模块进行高效的封存。

确定一个对象是感知型还是简单型

date 类型的对象都是简单型的。

time 或*datetime* 类型的对象可以是感知型或者简单型。

一个*datetime* 对象 *d* 在以下条件同时成立时将是感知型的:

1. `d.tzinfo` 不为 `None`
2. `d.tzinfo.utcoffset(d)` 不返回 `None`

在其他情况下，*d* 将是简单型的。

一个*time* 对象 *t* 在以下条件同时成立时将是感知型的:

1. `t.tzinfo` 不为 `None`
2. `t.tzinfo.utcoffset(None)` 不返回 `None`。

在其他情况下，*t* 将是简单型的。

感知型和简单型之间的区别不适用于*timedelta* 对象。

8.1.4 `timedelta` 类对象

timedelta 对象表示两个 date 或者 time 的时间间隔。

class datetime.**timedelta**(*days=0*, *seconds=0*, *microseconds=0*, *milliseconds=0*, *minutes=0*, *hours=0*, *weeks=0*)

所有参数都是可选的并且默认为 `0`。这些参数可以是整数或者浮点数，也可以是正数或者负数。

只有 *days*, *seconds* 和 *microseconds* 会存储在内部。参数单位的换算规则如下:

- 1 毫秒会转换成 1000 微秒。
- 1 分钟会转换成 60 秒。
- 1 小时会转换成 3600 秒。

- 1 星期会转换成 7 天。

并且 days, seconds, microseconds 会经标准化处理以保证表达方式的唯一性，即：

- `0 <= microseconds < 1000000`
- `0 <= seconds < 3600*24` (一天的秒数)
- `-999999999 <= days <= 999999999`

下面的例子演示了如何对 *days*, *seconds* 和 *microseconds* 以外的任意参数执行“合并”操作并标准化为以上三个结果属性:

```
>>> from datetime import timedelta
>>> delta = timedelta(
...     days=50,
...     seconds=27,
...     microseconds=10,
...     milliseconds=29000,
...     minutes=5,
...     hours=8,
...     weeks=2
... )
>>> # Only days, seconds, and microseconds remain
>>> delta
datetime.timedelta(days=64, seconds=29156, microseconds=10)
```

在有任何参数为浮点型并且 microseconds 值为小数的情况下，从所有参数中余下的微秒数将被合并，并使用四舍五入偶不入奇的规则将总计值舍入到最接近的整数微秒值。如果没有任何参数为浮点型的情况下，则转换和标准化过程将是完全精确的（不会丢失信息）。

如果标准化后的 days 数值超过了指定范围，将会抛出*`OverflowError`* 异常。

请注意对负数值进行标准化的结果可能会令人感到惊讶。例如:

```
>>> from datetime import timedelta
>>> d = timedelta(microseconds=-1)
>>> (d.days, d.seconds, d.microseconds)
(-1, 86399, 999999)
```

类属性：

`timedelta.`**`min`**
The most negative *`timedelta`* object, `timedelta(-999999999)`.

`timedelta.`**`max`**
The most positive *`timedelta`* object, `timedelta(days=999999999, hours=23, minutes=59, seconds=59, microseconds=999999)`.

`timedelta.`**`resolution`**
两个不相等的*`timedelta`* 类对象最小的间隔为 `timedelta(microseconds=1)`。

需要注意的是，因为标准化的缘故，`timedelta.max` > `-timedelta.min`，`-timedelta.max` 不可以表示一个*`timedelta`* 类对象。

实例属性（只读）：

属性	值
`days`	-999999999 至 999999999，含 999999999
`seconds`	0 至 86399，包含 86399
`microseconds`	0 至 999999，包含 999999

支持的运算：

运算	结果
`t1 = t2 + t3`	*t2* 和 *t3* 的和。运算后 *t1-t2 == t3* and *t1-t3 == t2* 必为真值。(1)
`t1 = t2 - t3`	*t2* 减 *t3* 的差。运算后 *t1 == t2 - t3* and *t2 == t1 + t3* 必为真值。(1)(6)
`t1 = t2 * i or t1 = i * t2`	乘以一个整数。运算后假如 `i != 0` 则 *t1* // i == *t2* 必为真值。
	In general, *t1* * i == *t1* * (i-1) + *t1* is true. (1)
`t1 = t2 * f or t1 = f * t2`	乘以一个浮点数，结果会被舍入到 timedelta 最接近的整数倍。精度使用四舍五偶入奇不入规则。
`f = t2 / t3`	总时间 *t2* 除以间隔单位 *t3* (3)。返回一个*float* 对象。
`t1 = t2 / f or t1 = t2 / i`	除以一个浮点数或整数。结果会被舍入到 timedelta 最接近的整数倍。精度使用四舍五偶入奇不入规则。
`t1 = t2 // i or t1 = t2 // t3`	计算底数，其余部分（如果有）将被丢弃。在第二种情况下，将返回整数。(3)
`t1 = t2 % t3`	余数为一个*timedelta* 对象。(3)
`q, r = divmod(t1, t2)`	通过: `q = t1 // t2` (3) and `r = t1 % t2` 计算出商和余数。q 是一个整数，r 是一个*timedelta* 对象。
`+t1`	返回一个相同数值的*timedelta* 对象。
`-t1`	等价于*timedelta*(*-t1.days*, *-t1.seconds*, *-t1.microseconds*), 和 *t1** -1. (1)(4)
`abs(t)`	当 `t.days >= 0``时等于 +\ *t*, 当 ``t.days < 0` 时 *-t* 。(2)
`str(t)`	返回一个形如 `[D day[s], ][H]H:MM:SS[.UUUUUU]` 的字符串，当 `t` 为负数的时候，D 也为负数。(5)
`repr(t)`	返回一个*timedelta* 对象的字符串表示形式，作为附带正规属性值的构造器调用。

F解：

(1) 结果正确，但可能会溢出。

(2) 结果正确，不会溢出。

(3) 除以 0 将会抛出异常*ZeroDivisionError* 。

(4) *-timedelta.max* 不是一个*timedelta* 类对象。

(5) *timedelta* 对象的字符串表示形式类似于其内部表示形式被规范化。对于负时间增量，这会导致一些不寻常的结果。例如:

```
>>> timedelta(hours=-5)
datetime.timedelta(days=-1, seconds=68400)
>>> print(_)
-1 day, 19:00:00
```

(6) 表达式 `t2 - t3` 通常与 `t2 + (-t3)` 是等价的，除非 t3 等于 `timedelta.max`; 在这种情况下前者会返回结果，而后者则会溢出。

除了上面列举的操作以外，*timedelta* 对象还支持与*date* 和*datetime* 对象进行特定的相加和相减运算（见下文）。

3.2 版更變: 现在已支持*timedelta* 对象与另一个*timedelta* 对象相整除或相除，包括求余运算和*divmod()* 函数。现在也支持*timedelta* 对象加上或乘以一个*float* 对象。

支持*timedelta* 对象之间进行比较，但其中有一些注意事项。

== 或 != 比较 总是返回一个*bool* 对象，无论被比较的对象是什么类型:

```
>>> from datetime import timedelta
>>> delta1 = timedelta(seconds=57)
>>> delta2 = timedelta(hours=25, seconds=2)
>>> delta2 != delta1
True
>>> delta2 == 5
False
```

对于所有其他比较 (例如 < 和 >)，当一个*timedelta*对象与其他类型的对象比较时，将引发*TypeError*:

```
>>> delta2 > delta1
True
>>> delta2 > 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'datetime.timedelta' and 'int'
```

在布尔运算中，*timedelta*对象当且仅当其不等于 timedelta(0) 时则会被视为真值。

实例方法:

timedelta.**total_seconds**()

返回时间间隔包含了多少秒。造价于 td / timedelta(seconds=1)。对于其它单位可以直接使用除法的形式 (例如 td / timedelta(microseconds=1))。

需要注意的是，时间间隔较大时，这个方法的结果中的微秒将会失真（大多数平台上大于 270 年视为一个较大的时间间隔）。

3.2 版新加入.

class:*timedelta* 用法示例

一个标准化的附加示例:

```
>>> # Components of another_year add up to exactly 365 days
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> another_year = timedelta(weeks=40, days=84, hours=23,
...                          minutes=50, seconds=600)
>>> year == another_year
True
>>> year.total_seconds()
31536000.0
```

*timedelta*算术运算的示例:

```
>>> from datetime import timedelta
>>> year = timedelta(days=365)
>>> ten_years = 10 * year
>>> ten_years
datetime.timedelta(days=3650)
>>> ten_years.days // 365
10
>>> nine_years = ten_years - year
>>> nine_years
datetime.timedelta(days=3285)
>>> three_years = nine_years // 3
```

(下页继续)

(繼續上一頁)

```
>>> three_years, three_years.days // 365
(datetime.timedelta(days=1095), 3)
```

8.1.5 date 对象

date 对象代表一个理想化历法中的日期（年、月和日），即当今的格列高利历向前后两个方向无限延伸。

公元 1 年 1 月 1 日是第 1 日，公元 1 年 1 月 2 日是第 2 日，依此类推。[2]

class datetime.**date**(*year*, *month*, *day*)

所有参数都是必要的。参数必须是在下面范围内的整数：

- `MINYEAR <= year <= MAXYEAR`
- `1 <= month <= 12`
- `1 <= 日期 <= 给定年月对应的天数`

如果参数不在这些范围内，则抛出*ValueError* 异常。

其它构造器，所有的类方法：

classmethod date.**today**()

返回当前的本地日期。

这等价于 `date.fromtimestamp(time.time())`。

classmethod date.**fromtimestamp**(*timestamp*)

返回对应于 POSIX 时间戳的当地时间，例如*time.time()* 返回的就是时间戳。

这可能引发*OverflowError*，如果时间戳数值超出所在平台 C `localtime()` 函数的支持范围的话，并且会在 `localtime()` 出错时引发*OSError*。通常该数值会被限制在 1970 年至 2038 年之间。请注意在时间戳概念包含闰秒的非 POSIX 系统上，闰秒会被*fromtimestamp()* 所忽略。

3.3 版更變: 引发*OverflowError* 而不是*ValueError*，如果时间戳数值超出所在平台 C `localtime()` 函数的支持范围的话，并会在 `localtime()` 出错时引发*OSError* 而不是*ValueError*。

classmethod date.**fromordinal**(*ordinal*)

返回对应于预期格列高利历序号的日期，其中公元 1 年 1 月 1 晶的序号为 1。

除非 `1 <= ordinal <= date.max.toordinal()` 否则会引发*ValueError*。对于任意日期 *d*，`date.fromordinal(d.toordinal()) == d`。

classmethod date.**fromisoformat**(*date_string*)

返回一个对应于以 `YYYY-MM-DD` 格式给出的 *date_string* 的*date* 对象：

```
>>> from datetime import date
>>> date.fromisoformat('2019-12-04')
datetime.date(2019, 12, 4)
```

这是*date.isoformat()* 的逆操作。它只支持 `YYYY-MM-DD` 格式。

3.7 版新加入.

classmethod date.**fromisocalendar**(*year*, *week*, *day*)

返回指定 year, week 和 day 所对应 ISO 历法日期的*date*。这是函数*date.isocalendar()* 的逆操作。

3.8 版新加入.

[2] 这与 Dershowitz 和 Reingold 所著 *Calendrical Calculations* 中“预期格列高利”历法的定义一致，它是适用于该书中所有运算的基础历法。请参阅该书了解在预期格利高利历序列与许多其他历法系统之间进行转换的算法。

类属性：

date.**min**
最小的日期 date(MINYEAR, 1, 1)。

date.**max**
最大的日期，date(MAXYEAR, 12, 31)。

date.**resolution**
两个日期对象的最小间隔，timedelta(days=1)。

实例属性（只读）：

date.**year**
在*MINYEAR*和*MAXYEAR*之间，包含边界。

date.**month**
1 至 12（含）

date.**day**
返回 1 到指定年月的天数间的数字。

支持的运算：

运算	结果
date2 = date1 + timedelta	*date2* 等于从 *date1* 减去 timedelta.days 天。(1)
date2 = date1 - timedelta	计算 *date2* 的值使得 date2 + timedelta == date1。(2)
timedelta = date1 - date2	(3)
date1 < date2	如果 *date1* 的时间在 *date2* 之前则认为 *date1* 小于 *date2* 。(4)

🄵解：

(1) 如果 timedelta.days > 0 则 *date2* 将在时间线上前进，如果 timedelta.days < 0 则将后退。操作完成后 date2 - date1 == timedelta.days。timedelta.seconds 和 timedelta.microseconds 会被忽略。如果 date2.year 将小于*MINYEAR* 或大于*MAXYEAR* 则会引发*OverflowError*。

(2) timedelta.seconds 和 timedelta.microseconds 会被忽略。

(3) 此值完全精确且不会溢出。操作完成后 timedelta.seconds 和 timedelta.microseconds 均为 0，并且 date2 + timedelta == date1。

(4) 换句话说，当且仅当 date1.toordinal() < date2.toordinal() 时 date1 < date2。日期比较会引发*TypeError*，如果比较目标不为*date* 对象的话。不过也可能会返回 NotImplemented，如果比较目标具有 timetuple() 属性的话。这个钩子给予其他日期对象类型实现混合类型比较的机会。否则，当*date* 对象与不同类型的对象比较时将会引发*TypeError*，除非是 == 或 != 比较。后两种情况将分别返回*False* 或*True*。

在布尔运算中，所有*date* 对象都会被视为真值。

实例方法：

date.**replace**(*year=self.year*, *month=self.month*, *day=self.day*)
返回一个具有同样值的日期，除非通过任何关键字参数给出了某些形参的新值。

示例：

```
>>> from datetime import date
>>> d = date(2002, 12, 31)
>>> d.replace(day=26)
datetime.date(2002, 12, 26)
```

date.**timetuple**()
返回一个*time.struct_time*，即*time.localtime()* 所返回的类型。

hours, minutes 和 seconds 值均为 0，且 DST 旗标值为 -1。

`d.timetuple()` 等价于:

```
time.struct_time((d.year, d.month, d.day, 0, 0, 0, d.weekday(), yday, -1))
```

其中 `yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1` 是当前年份中的日期序号，1 月 1 日的序号为 1。

date.**toordinal**()
返回日期的预期格列高利历序号，其中公元 1 年 1 月 1 日的序号为 1。对于任意*date* 对象 *d*，`date.fromordinal(d.toordinal()) == d`。

date.**weekday**()
返回一个整数代表星期几，星期一为 0，星期天为 6。例如，`date(2002, 12, 4).weekday() == 2`，表示的是星期三。参阅*isoweekday()*。

date.**isoweekday**()
返回一个整数代表星期几，星期一为 1，星期天为 7。例如: `date(2002, 12, 4).isoweekday() == 3`, 表示星期三。参见*weekday()*, *isocalendar()*。

date.**isocalendar**()
返回一个由三部分组成的*named tuple* 对象: `year`, `week` 和 `weekday`。

ISO 历法是一种被广泛使用的格列高利历。[3]

ISO 年由 52 或 53 个完整星期构成，每个星期开始于星期一结束于星期日。一个 ISO 年的第一个星期就是（格列高利）历法的一年中第一个包含星期四的星期。这被称为 1 号星期，这个星期四所在的 ISO 年与其所在的格列高利年相同。

例如，2004 年的第一天是星期四，因此 ISO 2004 年的第一个星期开始于 2003 年 12 月 29 日星期一，结束于 2004 年 1 月 4 日星期日:

```
>>> from datetime import date
>>> date(2003, 12, 29).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=1)
>>> date(2004, 1, 4).isocalendar()
datetime.IsoCalendarDate(year=2004, week=1, weekday=7)
```

3.9 版更變: 结果由元组改为*named tuple*。

date.**isoformat**()
返回一个以 ISO 8601 格式 `YYYY-MM-DD` 来表示日期的字符串:

```
>>> from datetime import date
>>> date(2002, 12, 4).isoformat()
'2002-12-04'
```

这是*date.fromisoformat()* 的逆操作。

date.**__str__**()
对于日期对象 *d*, `str(d)` 等价于 `d.isoformat()`。

date.**ctime**()
返回一个表示日期的字符串:

[3] 请参阅 R. H. van Gent 所著 ' ISO 8601 历法的数学指导 <https://www.staff.science.uu.nl/~gent0113/calendar/isocalendar.htm>'_ 以获取更完整的说明。

```
>>> from datetime import date
>>> date(2002, 12, 4).ctime()
'Wed Dec  4 00:00:00 2002'
```

d.ctime() 等效于:

```
time.ctime(time.mktime(d.timetuple()))
```

在原生 C ctime() 函数 (*time.ctime()* 会发起调用该函数，但*date.ctime()* 则不会) 遵循 C 标准的平台上。

date.**strftime**(*format*)

返回一个由显式格式字符串所指明的代表日期的字符串。表示时、分或秒的格式代码值将为 0。要获取格式指令的完整列表请参阅*strftime() 和 strptime() 的行为*。

date.**__format__**(*format*)

与*date.strftime()* 相同。此方法使得为*date* 对象指定以 格式化字符串字面值表示的格式化字符串以及使用*str.format()* 进行格式化成为可能。要获取格式指令的完整列表，请参阅*strftime() 和 strptime() 的行为*。

class:*date* 用法示例

计算距离特定事件天数的例子:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
>>> if my_birthday < today:
...     my_birthday = my_birthday.replace(year=today.year + 1)
>>> my_birthday
datetime.date(2008, 6, 24)
>>> time_to_birthday = abs(my_birthday - today)
>>> time_to_birthday.days
202
```

使用*date* 的更多例子:

```
>>> from datetime import date
>>> d = date.fromordinal(730920) # 730920th day after 1. 1. 0001
>>> d
datetime.date(2002, 3, 11)

>>> # Methods related to formatting string output
>>> d.isoformat()
'2002-03-11'
>>> d.strftime("%d/%m/%y")
'11/03/02'
>>> d.strftime("%A %d. %B %Y")
'Monday 11. March 2002'
>>> d.ctime()
'Mon Mar 11 00:00:00 2002'
```

(下页继续)

(繼續上一頁)

```
>>> 'The {1} is {0:%d}, the {2} is {0:%B}.'.format(d, "day", "month")
'The day is 11, the month is March.'

>>> # Methods for to extracting 'components' under different calendars
>>> t = d.timetuple()
>>> for i in t:
...     print(i)
2002                # year
3                   # month
11                  # day
0
0
0
0                   # weekday (0 = Monday)
70                  # 70th day in the year
-1
>>> ic = d.isocalendar()
>>> for i in ic:
...     print(i)
2002                # ISO year
11                  # ISO week number
1                   # ISO day number ( 1 = Monday )

>>> # A date object is immutable; all operations produce a new object
>>> d.replace(year=2005)
datetime.date(2005, 3, 11)
```

8.1.6 datetime 对象

datetime 对象是包含来自*date* 对象和*time* 对象的所有信息的单一对象。

与*date* 对象一样，*datetime* 假定当前的格列高利历向前后两个方向无限延伸；与*time* 对象一样，*datetime* 假定每一天恰好有 3600*24 秒。

构造器：

class datetime.**datetime**(*year*, *month*, *day*, *hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)

year, *month* 和 *day* 参数是必须的。*tzinfo* 可以是 None 或者是一个*tzinfo* 子类的实例。其余的参数必须是在下面范围内的整数：

- MINYEAR <= year <= MAXYEAR,
- 1 <= month <= 12,
- 1 <= day <= 指定年月的天数,
- 0 <= hour < 24,
- 0 <= minute < 60,
- 0 <= second < 60,
- 0 <= microsecond < 1000000,
- fold in [0, 1].

如果参数不在这些范围内，则抛出*ValueError* 异常。

3.6 版新加入: 增加了 fold 参数。

其它构造器，所有的类方法：

classmethod datetime.**today**()
返回表示当前地方时的 datetime 对象，其中*tzinfo* 为 None。

等价于:

```
datetime.fromtimestamp(time.time())
```

另请参阅*now()*, *fromtimestamp()*。

此方法的功能等价于*now()*，但是不带 tz 形参。

classmethod datetime.**now**(*tz=None*)
返回表示当前地方时的 date 和 time 对象。

如果可选参数 *tz* 为 None 或未指定，这就类似于*today()*，但该方法会在可能的情况下提供比通过*time.time()* 时间戳所获时间值更高的精度（例如，在提供了 C gettimeofday() 函数的平台上就可以做到这一点）。

如果 *tz* 不为 None，它必须是*tzinfo* 子类的一个实例，并且当前日期和时间将被转换到 *tz* 时区。

此函数可以替代*today()* 和*utcnow()*。

classmethod datetime.**utcnow**()
返回表示当前 UTC 时间的 date 和 time，其中*tzinfo* 为 None。

这类似于*now()*，但返回的是当前 UTC 日期和时间，类型为简单型*datetime* 对象。感知型的当前 UTC 日期时间可通过调用 datetime.now(timezone.utc) 来获得。另请参阅*now()*。

> **警告:** 由于简单型 datetime 对象会被许多 datetime 方法当作本地时间来处理，最好是使用感知型日期时间对象来表示 UTC 时间。因此，创建表示当前 UTC 时间的对象的推荐方式是通过调用 datetime.now(timezone.utc)。

classmethod datetime.**fromtimestamp**(*timestamp*, *tz=None*)
返回对应于 POSIX 时间戳例如*time.time()* 的返回值的本地日期和时间。如果可选参数 *tz* 为 None 或未指定，时间戳会被转换为所在平台的本地日期和时间，返回的*datetime* 对象将为天真型。

如果 *tz* 不为 None，它必须是*tzinfo* 子类的一个实例，并且时间戳将被转换到 *tz* 指定的时区。

fromtimestamp() 可能会引发*OverflowError*，如果时间戳数值超出所在平台 C localtime() 或 gmtime() 函数的支持范围的话，并会在 localtime() 或 gmtime() 报错时引发*OSError*。通常该数值会被限制在 1970 年至 2038 年之间。请注意在时间戳概念包含闰秒的非 POSIX 系统上，闰秒会被*fromtimestamp()* 所忽略，结果可能导致两个相差一秒的时间戳产生相同的*datetime* 对象。相比*utcfromtimestamp()* 更推荐使用此方法。

3.3 版更變: 引发*OverflowError* 而不是*ValueError*，如果时间戳数值超出所在平台 C localtime() 或 gmtime() 函数的支持范围的话。并会在 localtime() 或 gmtime() 出错时引发*OSError* 而不是*ValueError*。

3.6 版更變: *fromtimestamp()* 可能返回*fold* 值设为 1 的实例。

classmethod datetime.**utcfromtimestamp**(*timestamp*)
返回对应于 POSIX 时间戳的 UTC *datetime*，其中*tzinfo* 值为 None。（结果为简单型对象。）

这可能引发*OverflowError*，如果时间戳数值超出所在平台 C gmtime() 函数的支持范围的话，并会在 gmtime() 报错时引发*OSError*。通常该数值会被限制在 1970 至 2038 年之间。

要得到一个感知型*datetime* 对象，应调用*fromtimestamp()*:

```
datetime.fromtimestamp(timestamp, timezone.utc)
```

在 POSIX 兼容的平台上，它等价于以下表达式:

```
datetime(1970, 1, 1, tzinfo=timezone.utc) + timedelta(seconds=timestamp)
```

不同之处在于后一种形式总是支持完整年份范围：从*MINYEAR* 到*MAXYEAR* 的开区间。

> **警告:** 由于简单型 `datetime` 对象会被许多 `datetime` 方法当作本地时间来处理，最好是使用感知型日期时间对象来表示 UTC 时间。因此，创建表示特定 UTC 时间戳的日期时间对象的推荐方式是通过调用 `datetime.fromtimestamp(timestamp, tz=timezone.utc)`。

3.3 版更變: 引发*OverflowError* 而不是*ValueError*，如果时间戳数值超出所在平台 C `gmtime()` 函数的支持范围的话。并会在 `gmtime()` 出错时引发*OSError* 而不是*ValueError*。

classmethod datetime.**fromordinal**(*ordinal*)
返回对应于预期格列高利历序号的*datetime*，其中公元 1 年 1 月 1 日的序号为 1。除非 `1 <= ordinal <= datetime.max.toordinal()` 否则会引发*ValueError*。结果的 hour, minute, second 和 microsecond 值均为 0，并且*tzinfo* 值为 `None`。

classmethod datetime.**combine**(*date*, *time*, *tzinfo=self.tzinfo*)
返回一个新的*datetime* 对象，对象的日期部分等于给定的*date* 对象的值，而其时间部分等于给定的*time* 对象的值。如果提供了 *tzinfo* 参数，其值会被用来设置结果的*tzinfo* 属性，否则将使用 *time* 参数的*tzinfo* 属性。

对于任意*datetime* 对象 *d*, `d == datetime.combine(d.date(), d.time(), d.tzinfo)`。如果 date 是一个*datetime* 对象，它的时间部分和*tzinfo* 属性会被忽略。

3.6 版更變: 增加了 *tzinfo* 参数。

classmethod datetime.**fromisoformat**(*date_string*)
返回一个对应于*date.isoformat()* 和*datetime.isoformat()* 所提供的某一种 *date_string* 的*datetime* 对象。

特别地，此函数支持以下格式的字符串:

```
YYYY-MM-DD[*HH[:MM[:SS[.fff[fff]]]][+HH:MM[:SS[.ffffff]]]]
```

其中 `*` 可以匹配任意的单个字符。

> **警示:** 此函数并不支持解析任意 ISO 8601 字符串——它的目的只是作为*datetime.isoformat()* 的逆操作。在第三方包 dateutil 中提供了一个更完善的 ISO 8601 解析器 `dateutil.parser.isoparse`。

示例:

```
>>> from datetime import datetime
>>> datetime.fromisoformat('2011-11-04')
datetime.datetime(2011, 11, 4, 0, 0)
>>> datetime.fromisoformat('2011-11-04T00:05:23')
datetime.datetime(2011, 11, 4, 0, 5, 23)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283')
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000)
>>> datetime.fromisoformat('2011-11-04 00:05:23.283+00:00')
```

(下页继续)

(繼續上一頁)

```
datetime.datetime(2011, 11, 4, 0, 5, 23, 283000, tzinfo=datetime.timezone.utc)
>>> datetime.fromisoformat('2011-11-04T00:05:23+04:00')
datetime.datetime(2011, 11, 4, 0, 5, 23,
    tzinfo=datetime.timezone(datetime.timedelta(seconds=14400)))
```

3.7 版新加入.

classmethod datetime.**fromisocalendar**(*year*, *week*, *day*)
返回以 year, week 和 day 值指明的 ISO 历法日期所对应的*datetime*。该 datetime 对象的非日期部分将使用其标准默认值来填充。这是函数*datetime.isocalendar()* 的逆操作。

3.8 版新加入.

classmethod datetime.**strptime**(*date_string*, *format*)
返回一个对应于 *date_string*，根据 *format* 进行解析得到的*datetime* 对象。

这相当于:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

如果 date_string 和 format 无法被*time.strptime()* 解析或它返回一个不是时间元组的值，则将引发*ValueError*。要获取格式化指令的完整列表，请参阅*strftime() 和 strptime() 的行为*。

类属性:

datetime.**min**
最早的可表示*datetime*，datetime(MINYEAR, 1, 1, tzinfo=None)。

datetime.**max**
最晚的可表示*datetime*，datetime(MAXYEAR, 12, 31, 23, 59, 59, 999999, tzinfo=None)。

datetime.**resolution**
两个不相等的*datetime* 对象之间可能的最小间隔，timedelta(microseconds=1)。

实例属性（只读):

datetime.**year**
在*MINYEAR* 和*MAXYEAR* 之间，包含边界。

datetime.**month**
1 至 12（含）

datetime.**day**
返回 1 到指定年月的天数间的数字。

datetime.**hour**
取值范围是 range(24)。

datetime.**minute**
取值范围是 range(60)。

datetime.**second**
取值范围是 range(60)。

datetime.**microsecond**
取值范围是 range(1000000)。

datetime.**tzinfo**
作为 *tzinfo* 参数被传给*datetime* 构造器的对象，如果没有传入值则为 None。

datetime.**fold**
取值范围是 [0, 1]。用于在重复的时间段中消除边界时间歧义。(当夏令时结束时回拨时钟或由于政治原因导致当明时区的 UTC 时差减少就会出现重复的时间段。) 取值 0 (1) 表示两个时刻早于（晚于）所代表的同一边界时间。

3.6 版新加入.

支持的运算：

运算	结果
datetime2 = datetime1 + timedelta	(1)
datetime2 = datetime1 - timedelta	(2)
timedelta = datetime1 - datetime2	(3)
datetime1 < datetime2	比较*datetime* 与*datetime*。(4)

(1) datetime2 是从 datetime1 去掉了一段 timedelta 的结果，如果 timedelta.days > 0 则是在时间线上前进，如果 timedelta.days < 0 则是在时间线上后退。该结果具有与输入的 datetime 相同的*tzinfo* 属性，并且操作完成后 datetime2 - datetime1 == timedelta。如果 datetime2.year 将要小于*MINYEAR* 或大于*MAXYEAR* 则会引发*OverflowError*。请注意即使输入的是一个感知型对象，该方法也不会进行时区调整。

(2) 计算 datetime2 使得 datetime2 + timedelta == datetime1。与相加操作一样，结果具有与输入的 datetime 相同的*tzinfo* 属性，即使输入的是一个感知型对象，该方法也不会进行时区调整。

(3) 从一个*datetime* 减去一个*datetime* 仅对两个操作数均为简单型或均为感知型时有定义。如果一个是感知型而另一个是简单型，则会引发*TypeError*。

如果两个操作数都是简单型，或都是感知型并且具有相同的*tzinfo* 属性，则*tzinfo* 属性会被忽略，并且结果会是一个使得 datetime2 + t == datetime1 的*timedelta* 对象 *t*。在此情况下不会进行时区调整。

如果两个操作数都是感知型且具有不同的*tzinfo* 属性，a-b 操作的效果就如同 *a* 和 *b* 首先被转换为简单型 UTC 日期时间。结果将是 (a.replace(tzinfo=None) - a.utcoffset()) - (b.replace(tzinfo=None) - b.utcoffset())，除非具体实现绝对不溢出。

(4) 当 *datetime1* 的时间在 *datetime2* 之前则认为 *datetime1* 小于 *datetime2*。

如果比较的一方是简单型而另一方是感知型，则如果尝试进行顺序比较将引发*TypeError*。对于相等比较，简单型实例将永远不等于感知型实例。

如果两个比较方都是感知型，且具有相同的*tzinfo* 属性，则相同的*tzinfo* 属性会被忽略并对基本日期时间值进行比较。如果两个比较方都是感知型且具有不同的*tzinfo* 属性，则两个比较方将首先通过减去它们的 UTC 差值（使用 self.utcoffset() 获取）来进行调整。

3.3 版更變: 感知型和简单型*datetime* 实例之间的相等比较不会引发*TypeError*。

備F: 为了防止比较操作回退为默认的对象地址比较方式，datetime 比较通常会引发*TypeError*，如果比较目标不同样为*datetime* 对象的话。不过也可能会返回 NotImplemented，如果比较目标具有 timetuple() 属性的话。这个钩子给予其他种类的日期对象实现混合类型比较的机会。如果未实现，则当*datetime* 对象与不同类型比较时将会引发*TypeError*，除非是 == 或 != 比较。后两种情况将分别返回*False* 或*True*。

实例方法：

datetime.**date**()
返回具有同样 year, month 和 day 值的*date* 对象。

datetime.**time**()
返回具有同样 hour, minute, second, microsecond 和 fold 值的*time* 对象。*tzinfo* 值为 None。另请参见*timetz()* 方法。

3.6 版更變: fold 值会被复制给返回的*time* 对象。

datetime.**timetz**()
返回具有同样 hour, minute, second, microsecond, fold 和 tzinfo 属性性的*time* 对象。另请参见*time()* 方法。

3.6 版更變: fold 值会被复制给返回的*time* 对象。

datetime.**replace**(*year=self.year*, *month=self.month*, *day=self.day*, *hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, ***, *fold=0*)
返回一个具有同样属性值的 datetime，除非通过任何关键字参数为某些属性指定了新值。请注意可以通过指定 tzinfo=None 来从一个感知型 datetime 创建一个简单型 datetime 而不必转换日期和时间数据。

3.6 版新加入: 增加了 fold 参数。

datetime.**astimezone**(*tz=None*)
返回一个具有新的*tzinfo* 属性 *tz* 的*datetime* 对象，并会调整日期和时间数据使得结果对应的 UTC 时间与 *self* 相同，但为 *tz* 时区的本地时间。

如果给出了 *tz*，则它必须是一个*tzinfo* 子类的实例，并且其*utcoffset()* 和*dst()* 方法不可返回 None。如果 *self* 为简单型，它会被假定为基于系统时区表示的时间。

如果调用时不传入参数 (或传入 tz=None) 则将假定目标时区为系统的本地时区。转换后 datetime 实例的 .tzinfo 属性将被设为一个*timezone* 实例，时区名称和时差值将从 OS 获取。

如果 self.tzinfo 为 *tz*，self.astimezone(tz) 等于 *self*: 不会对日期或时间数据进行调整。否则结果为 *tz* 时区的本地时间，代表的 UTC 时间与 *self* 相同：在 astz = dt.astimezone(tz) 之后，astz - astz.utcoffset() 将具有与 dt - dt.utcoffset() 相同的日期和时间数据。

如果你只是想要附加一个时区对象 *tz* 到一个 datetime 对象 *dt* 而不调整日期和时间数据，请使用 dt.replace(tzinfo=tz)。如果你只想从一个感知型 datetime 对象 *dt* 移除时区对象，请使用 dt.replace(tzinfo=None)。

请注意默认的*tzinfo.fromutc()* 方法在*tzinfo* 的子类中可以被重载，从而影响*astimezone()* 的返回结果。如果忽略出错的情况，*astimezone()* 的行为就类似于:

```
def astimezone(self, tz):
    if self.tzinfo is tz:
        return self
    # Convert self to UTC, and attach the new time zone object.
    utc = (self - self.utcoffset()).replace(tzinfo=tz)
    # Convert from UTC to tz's local time.
    return tz.fromutc(utc)
```

3.3 版更變: *tz* 现在可以被省略。

3.6 版更變: *astimezone()* 方法可以由简单型实例调用，这将假定其表示本地时间。

datetime.**utcoffset**()
如果*tzinfo* 为 None，则返回 None，否则返回 self.tzinfo.utcoffset(self)，并且在后者不返回 None 或者一个幅度小于一天的*timedelta* 对象时将引发异常。

3.7 版更變: UTC 时差不再限制为一个整数分钟值。

datetime.**dst**()
如果*tzinfo* 为 None，则返回 None，否则返回 self.tzinfo.dst(self)，并且在后者不返回 None 或者一个幅度小于一天的*timedelta* 对象时将引发异常。

3.7 版更變: DST 差值不再限制为一个整数分钟值。

datetime.**tzname**()
如果*tzinfo*为 None，则返回 None，否则返回 self.tzinfo.tzname(self)，如果后者不返回 None 或者一个字符串对象则将引发异常。

datetime.**timetuple**()
返回一个*time.struct_time*，即*time.localtime()*所返回的类型。

d.timetuple() 等价于:

```
time.struct_time((d.year, d.month, d.day,
                  d.hour, d.minute, d.second,
                  d.weekday(), yday, dst))
```

其中 yday = d.toordinal() - date(d.year, 1, 1).toordinal() + 1 是日期在当前年份中的序号，起始序号 1 表示 1 月 1 日。结果的 tm_isdst 旗标的设定会依据*dst()*方法：如果*tzinfo*为 None 或*dst()*返回 None，则 tm_isdst 将设为 -1；否则如果*dst()*返回一个非零值，则 tm_isdst 将设为 1；在其他情况下 tm_isdst 将设为 0。

datetime.**utctimetuple**()
如果*datetime*实例 *d* 为简单型，这类似于 d.timetuple()，不同之处在于 tm_isdst 会强制设为 0，无论 d.dst() 返回什么结果。DST 对于 UTC 时间永远无效。

如果 *d* 为感知型，*d* 会通过减去 d.utcoffset() 来标准化为 UTC 时间，并返回该标准化时间所对应的*time.struct_time*。tm_isdst 会强制设为 0。请注意如果 *d*.year 为 MINYEAR 或 MAXYEAR 并且 UTC 调整超出一年的边界则可能引发*OverflowError*。

> **警告:** 由于简单型 datetime 对象会被许多 datetime 方法当作本地时间来处理，最好是使用感知型日期时间来表示 UTC 时间；因此，使用 utcfromtimetuple 可能会给出误导性的结果。如果你有一个表示 UTC 的简单型 datetime，请使用 datetime.replace(tzinfo=timezone.utc) 将其改为感知型，这样你才能使用*datetime.timetuple()*。

datetime.**toordinal**()
返回日期的预期格列高利历序号。与 self.date().toordinal() 相同。

datetime.**timestamp**()
返回对应于*datetime*实例的 POSIX 时间戳。此返回值是与*time.time()*返回值类似的*float*对象。

简单型*datetime*实例会假定为代表本地时间，并且此方法依赖于平台的 C mktime() 函数来执行转换。由于在许多平台上*datetime*支持的范围比 mktime() 更广，对于极其遥远的过去或未来此方法可能引发*OverflowError*。

对于感知型*datetime*实例，返回值的计算方式为:

```
(dt - datetime(1970, 1, 1, tzinfo=timezone.utc)).total_seconds()
```

3.3 版新加入.

3.6 版更變: *timestamp()*方法使用*fold*属性来消除重复间隔中的时间歧义。

備F: 没有一个方法能直接从表示 UTC 时间的简单型*datetime*实例获取 POSIX 时间戳。如果你的应用程序使用此惯例并且你的系统时区不是设为 UTC，你可以通过提供 tzinfo=timezone.utc 来获取 POSIX 时间戳:

```
timestamp = dt.replace(tzinfo=timezone.utc).timestamp()
```

或者通过直接计算时间戳:

```
timestamp = (dt - datetime(1970, 1, 1)) / timedelta(seconds=1)
```

---

datetime.**weekday**()
返回一个整数代表星期几，星期一为 0，星期天为 6。相当于 self.date().weekday()。另请参阅*isoweekday()*。

datetime.**isoweekday**()
返回一个整数代表星期几，星期一为 1，星期天为 7。相当于 self.date().isoweekday()。另请参阅*weekday()*, *isocalendar()*。

datetime.**isocalendar**()
返回一个由三部分组成的*named tuple*: year, week 和 weekday。等同于 self.date().isocalendar()。

datetime.**isoformat**(*sep='T'*, *timespec='auto'*)
返回一个以 ISO 8601 格式表示的日期和时间字符串：

- YYYY-MM-DDTHH:MM:SS.ffffff，如果*microsecond* 不为 0
- YYYY-MM-DDTHH:MM:SS，如果*microsecond* 为 0

如果*utcoffset()* 返回值不为 None，则添加一个字符串来给出 UTC 时差：

- YYYY-MM-DDTHH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]，如果*microsecond* 不为 0
- YYYY-MM-DDTHH:MM:SS+HH:MM[:SS[.ffffff]]，如果*microsecond* 为 0

示例:

```
>>> from datetime import datetime, timezone
>>> datetime(2019, 5, 18, 15, 17, 8, 132263).isoformat()
'2019-05-18T15:17:08.132263'
>>> datetime(2019, 5, 18, 15, 17, tzinfo=timezone.utc).isoformat()
'2019-05-18T15:17:00+00:00'
```

可选参数 *sep* (默认为 'T') 为单个分隔字符，会被放在结果的日期和时间两部分之间。例如:

```
>>> from datetime import tzinfo, timedelta, datetime
>>> class TZ(tzinfo):
...     """A time zone with an arbitrary, constant -06:39 offset."""
...     def utcoffset(self, dt):
...         return timedelta(hours=-6, minutes=-39)
...
>>> datetime(2002, 12, 25, tzinfo=TZ()).isoformat(' ')
'2002-12-25 00:00:00-06:39'
>>> datetime(2009, 11, 27, microsecond=100, tzinfo=TZ()).isoformat()
'2009-11-27T00:00:00.000100-06:39'
```

可选参数 *timespec* 要包含的额外时间组件值 (默认为 'auto')。它可以是以下值之一:

- 'auto': 如果*microsecond* 为 0 则与 'seconds' 相同，否则与 'microseconds' 相同。
- 'hours': 以两个数码的 HH 格式包含*hour*。
- 'minutes': 以 HH:MM 格式包含*hour* 和*minute*。
- 'seconds': 以 HH:MM:SS 格式包含*hour*, *minute* 和*second*。
- 'milliseconds': 包含完整时间，但将秒值的小数部分截断至微秒。格式为 HH:MM:SS.sss
- 'microseconds': 以 HH:MM:SS.ffffff 格式包含完整时间。

備𝔽: 排除掉的时间部分将被截断，而不是被舍入。

对于无效的 *timespec* 参数将引发*ValueError*:

```
>>> from datetime import datetime
>>> datetime.now().isoformat(timespec='minutes')
'2002-12-25T00:00'
>>> dt = datetime(2015, 1, 1, 12, 30, 59, 0)
>>> dt.isoformat(timespec='microseconds')
'2015-01-01T12:30:59.000000'
```

3.6 版新加入: 增加了 *timespec* 参数。

datetime.**__str__**()
对于*datetime* 实例 *d*，`str(d)` 等价于 `d.isoformat(' ')`。

datetime.**ctime**()
返回一个表示日期和时间的字符串:

```
>>> from datetime import datetime
>>> datetime(2002, 12, 4, 20, 30, 40).ctime()
'Wed Dec  4 20:30:40 2002'
```

输出字符串将 *并不*包括时区信息，无论输入的是感知型还是简单型。

`d.ctime()` 等效于:

```
time.ctime(time.mktime(d.timetuple()))
```

在原生 C `ctime()` 函数 (*time.ctime()* 会发起调用该函数，但*datetime.ctime()* 则不会) 遵循 C 标准的平台上。

datetime.**strftime**(*format*)
返回一个由显式格式字符串所指明的代表日期和时间的字符串，要获取格式指令的完整列表，请参阅*strftime() 和 strptime() 的行为*。

datetime.**__format__**(*format*)
与*datetime.strftime()* 相同。此方法使得为*datetime* 对象指定以 格式化字符串字面值表示的格式化字符串以及使用*str.format()* 进行格式化成为可能。要获取格式指令的完整列表，请参阅*strftime() 和 strptime() 的行为*。

用法示例: datetime

使用*datetime* 对象的例子:

```
>>> from datetime import datetime, date, time, timezone

>>> # Using datetime.combine()
>>> d = date(2005, 7, 14)
>>> t = time(12, 30)
>>> datetime.combine(d, t)
datetime.datetime(2005, 7, 14, 12, 30)

>>> # Using datetime.now()
>>> datetime.now()
datetime.datetime(2007, 12, 6, 16, 29, 43, 79043)   # GMT +1
```

(下页继续)

(繼續上一頁)

```
>>> datetime.now(timezone.utc)
datetime.datetime(2007, 12, 6, 15, 29, 43, 79060, tzinfo=datetime.timezone.utc)

>>> # Using datetime.strptime()
>>> dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
>>> dt
datetime.datetime(2006, 11, 21, 16, 30)

>>> # Using datetime.timetuple() to get tuple of all attributes
>>> tt = dt.timetuple()
>>> for it in tt:
...     print(it)
...
2006    # year
11      # month
21      # day
16      # hour
30      # minute
0       # second
1       # weekday (0 = Monday)
325     # number of days since 1st January
-1      # dst - method tzinfo.dst() returned None

>>> # Date in ISO format
>>> ic = dt.isocalendar()
>>> for it in ic:
...     print(it)
...
2006    # ISO year
47      # ISO week
2       # ISO weekday

>>> # Formatting a datetime
>>> dt.strftime("%A, %d. %B %Y %I:%M%p")
'Tuesday, 21. November 2006 04:30PM'
>>> 'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format(dt, "day",
↪"month", "time")
'The day is 21, the month is November, the time is 04:30PM.'
```

以下示例定义了一个*tzinfo* 子类，它捕获 Kabul, Afghanistan 时区的信息，该时区使用 +4 UTC 直到 1945 年，之后则使用 +4:30 UTC:

```
from datetime import timedelta, datetime, tzinfo, timezone

class KabulTz(tzinfo):
    # Kabul used +4 until 1945, when they moved to +4:30
    UTC_MOVE_DATE = datetime(1944, 12, 31, 20, tzinfo=timezone.utc)

    def utcoffset(self, dt):
        if dt.year < 1945:
            return timedelta(hours=4)
        elif (1945, 1, 1, 0, 0) <= dt.timetuple()[:5] < (1945, 1, 1, 0, 30):
            # An ambiguous ("imaginary") half-hour range representing
            # a 'fold' in time due to the shift from +4 to +4:30.
            # If dt falls in the imaginary range, use fold to decide how
            # to resolve. See PEP495.
```

(下页继续)

(繼續上一頁)

```
            return timedelta(hours=4, minutes=(30 if dt.fold else 0))
        else:
            return timedelta(hours=4, minutes=30)

    def fromutc(self, dt):
        # Follow same validations as in datetime.tzinfo
        if not isinstance(dt, datetime):
            raise TypeError("fromutc() requires a datetime argument")
        if dt.tzinfo is not self:
            raise ValueError("dt.tzinfo is not self")

        # A custom implementation is required for fromutc as
        # the input to this function is a datetime with utc values
        # but with a tzinfo set to self.
        # See datetime.astimezone or fromtimestamp.
        if dt.replace(tzinfo=timezone.utc) >= self.UTC_MOVE_DATE:
            return dt + timedelta(hours=4, minutes=30)
        else:
            return dt + timedelta(hours=4)

    def dst(self, dt):
        # Kabul does not observe daylight saving time.
        return timedelta(0)

    def tzname(self, dt):
        if dt >= self.UTC_MOVE_DATE:
            return "+04:30"
        return "+04"
```

上述 `KabulTz` 的用法:

```
>>> tz1 = KabulTz()

>>> # Datetime before the change
>>> dt1 = datetime(1900, 11, 21, 16, 30, tzinfo=tz1)
>>> print(dt1.utcoffset())
4:00:00

>>> # Datetime after the change
>>> dt2 = datetime(2006, 6, 14, 13, 0, tzinfo=tz1)
>>> print(dt2.utcoffset())
4:30:00

>>> # Convert datetime to another time zone
>>> dt3 = dt2.astimezone(timezone.utc)
>>> dt3
datetime.datetime(2006, 6, 14, 8, 30, tzinfo=datetime.timezone.utc)
>>> dt2
datetime.datetime(2006, 6, 14, 13, 0, tzinfo=KabulTz())
>>> dt2 == dt3
True
```

8.1.7 time 对象

一个*time*对象代表某日的（本地）时间，它独立于任何特定日期，并可通过*tzinfo*对象来调整。

class datetime.**time**(*hour=0*, *minute=0*, *second=0*, *microsecond=0*, *tzinfo=None*, *, *fold=0*)
所有参数都是可选的。*tzinfo* 可以是 None，或者是一个*tzinfo*子类的实例。其余的参数必须是在下面范围内的整数：

- 0 <= hour < 24,
- 0 <= minute < 60,
- 0 <= second < 60,
- 0 <= microsecond < 1000000,
- fold in [0, 1].

如果给出一个此范围以外的参数，则会引发*ValueError*。所有参数值默认为 0，只有 *tzinfo* 默认为*None*。

类属性：

time.**min**
早最的可表示*time*, time(0, 0, 0, 0)。

time.**max**
最晚的可表示*time*, time(23, 59, 59, 999999)。

time.**resolution**
两个不相等的*time*对象之间可能的最小间隔，timedelta(microseconds=1)，但是请注意*time*对象并不支持算术运算。

实例属性（只读）：

time.**hour**
取值范围是 range(24)。

time.**minute**
取值范围是 range(60)。

time.**second**
取值范围是 range(60)。

time.**microsecond**
取值范围是 range(1000000)。

time.**tzinfo**
作为 tzinfo 参数被传给*time*构造器的对象，如果没有传入值则为 None。

time.**fold**
取值范围是 [0, 1]。用于在重复的时间段中消除边界时间歧义。（当夏令时结束时回拨时钟或由于政治原因导致当明时区的 UTC 时差减少就会出现重复的时间段。）取值 0 (1) 表示两个时刻早于（晚于）所代表的同一边界时间。

3.6 版新加入.

*time*对象支持*time*与*time*的比较，当 *a* 时间在 *b* 之前时，则认为 *a* 小于 *b*。如果比较的一方是简单型而另一方是感知型，则如果尝试进行顺序比较将引发*TypeError*。对于相等比较，简单型实例将永远不等于感知型实例。

如果两个比较方都是感知型，且具有相同的*tzinfo*属性，相同的*tzinfo*属性会被忽略并对基本时间值进行比较。如果两个比较方都是感知型且具有不同的*tzinfo*属性，两个比较方将首先通过减去它们的 UTC 时差（从 self.utcoffset() 获取）来进行调整。为了防止将混合类型比较回退为基于对象地址的默认比

较，当*time* 对象与不同类型的对象比较时，将会引发*TypeError*，除非比较运算符是 == 或 !=。在后两种情况下将分别返回*False* 或*True*。

3.3 版更變: 感知型和简单型*time* 实例之间的相等比较不会引发*TypeError*。

在布尔运算时，*time* 对象总是被视为真值。

3.5 版更變: 在 Python 3.5 之前，如果一个*time* 对象代表 UTC 午夜零时则会被视为假值。此行为被认为容易引发困惑和错误，因此从 Python 3.5 起已被去除。详情参见 bpo-13936。

其他构造方法：

classmethod time.**fromisoformat**(*time_string*)

返回对应于*time.isoformat()* 所提供的某种 *time_string* 格式的*time*。特别地，此函数支持以下格式的字符串：

```
HH[:MM[:SS[.fff[fff]]]][+HH:MM[:SS[.ffffff]]]
```

> **警示:** 此方法 并不支持解析任意 ISO 8601 字符串。它的目的只是作为*time.isoformat()* 的逆操作。

示例:

```
>>> from datetime import time
>>> time.fromisoformat('04:23:01')
datetime.time(4, 23, 1)
>>> time.fromisoformat('04:23:01.000384')
datetime.time(4, 23, 1, 384)
>>> time.fromisoformat('04:23:01+04:00')
datetime.time(4, 23, 1, tzinfo=datetime.timezone(datetime.
↪timedelta(seconds=14400)))
```

3.7 版新加入.

实例方法：

time.**replace**(*hour=self.hour*, *minute=self.minute*, *second=self.second*, *microsecond=self.microsecond*, *tzinfo=self.tzinfo*, ***, *fold=0*)

返回一个具有同样属性值的*time*，除非通过任何关键字参数指定了某些属性值。请注意可以通过指定 tzinfo=None 从一个感知型*time* 创建一个简单型*time*，而不必转换时间数据。

3.6 版新加入: 增加了 fold 参数。

time.**isoformat**(*timespec='auto'*)

返回表示为下列 ISO 8601 格式之一的时间字符串：

- HH:MM:SS.ffffff，如果*microsecond* 不为 0
- HH:MM:SS，如果*microsecond* 为 0
- HH:MM:SS.ffffff+HH:MM[:SS[.ffffff]]，如果*utcoffset()* 不返回 None
- HH:MM:SS+HH:MM[:SS[.ffffff]]，如果*microsecond* 为 0 并且*utcoffset()* 不返回 None

可选参数 *timespec* 要包含的额外时间组件值 (默认为 'auto')。它可以是以下值之一：

- 'auto': 如果*microsecond* 为 0 则与 'seconds' 相同，否则与 'microseconds' 相同。
- 'hours': 以两个数码的 HH 格式包含*hour*。
- 'minutes': 以 HH:MM 格式包含*hour* 和*minute*。

- `'seconds'`: 以 `HH:MM:SS` 格式包含*hour*, *minute* 和*second*。
- `'milliseconds'`: 包含完整时间，但将秒值的小数部分截断至微秒。格式为 `HH:MM:SS.sss`
- `'microseconds'`: 以 `HH:MM:SS.ffffff` 格式包含完整时间。

備🄵: 排除掉的时间部分将被截断，而不是被舍入。

对于无效的 *timespec* 参数将引发*ValueError*。

示例:

```
>>> from datetime import time
>>> time(hour=12, minute=34, second=56, microsecond=123456).isoformat(timespec=
↪'minutes')
'12:34'
>>> dt = time(hour=12, minute=34, second=56, microsecond=0)
>>> dt.isoformat(timespec='microseconds')
'12:34:56.000000'
>>> dt.isoformat(timespec='auto')
'12:34:56'
```

3.6 版新加入: 增加了 *timespec* 参数。

`time.`**`__str__`**`()`
对于时间对象 *t*, `str(t)` 等价于 `t.isoformat()`。

`time.`**`strftime`**`(`*format*`)`
返回一个由显式格式字符串所指明的代表时间的字符串。要获取格式指令的完整列表，请参阅*strftime() 和 strptime() 的行为*。

`time.`**`__format__`**`(`*format*`)`
与*time.strftime()* 相同。此方法使得为*time* 对象指定以 格式化字符串字面值表示的格式化字符串以及使用*str.format()* 进行格式化成为可能。要获取格式指令的完整列表，请参阅*strftime() 和 strptime() 的行为*。

`time.`**`utcoffset`**`()`
如果*tzinfo* 为 `None`，则返回 `None`，否则返回 `self.tzinfo.utcoffset(None)`，并且在后者不返回 `None` 或一个幅度小于一天的 a *timedelta* 对象时将引发异常。

3.7 版更變: UTC 时差不再限制为一个整数分钟值。

`time.`**`dst`**`()`
如果*tzinfo* 为 `None`，则返回 `None`，否则返回 `self.tzinfo.dst(None)`，并且在后者不返回 `None` 或者一个幅度小于一天的*timedelta* 对象时将引发异常。

3.7 版更變: DST 差值不再限制为一个整数分钟值。

`time.`**`tzname`**`()`
如果*tzinfo* 为 `None`，则返回 `None`，否则返回 `self.tzinfo.tzname(None)`，如果后者不返回 `None` 或者一个字符串对象则将引发异常。

用法示例: time

使用*time*对象的例子:

```
>>> from datetime import time, tzinfo, timedelta
>>> class TZ1(tzinfo):
...     def utcoffset(self, dt):
...         return timedelta(hours=1)
...     def dst(self, dt):
...         return timedelta(0)
...     def tzname(self,dt):
...         return "+01:00"
...     def  __repr__(self):
...         return f"{self.__class__.__name__}()"
...
>>> t = time(12, 10, 30, tzinfo=TZ1())
>>> t
datetime.time(12, 10, 30, tzinfo=TZ1())
>>> t.isoformat()
'12:10:30+01:00'
>>> t.dst()
datetime.timedelta(0)
>>> t.tzname()
'+01:00'
>>> t.strftime("%H:%M:%S %Z")
'12:10:30 +01:00'
>>> 'The {} is {:%H:%M}.'.format("time", t)
'The time is 12:10.'
```

8.1.8 tzinfo 对象

class datetime.**tzinfo**

这是一个抽象基类，也就是说该类不应被直接实例化。请定义*tzinfo*的子类来捕获有关特定时区的信息。

*tzinfo*的（某个实体子类）的实例可以被传给*datetime*和*time*对象的构造器。这些对象会将它们的属性视为对应于本地时间，并且*tzinfo*对象支持展示本地时间与 UTC 的差值、时区名称以及 DST 差值的方法，都是与传给它们的日期或时间对象的相对值。

你需要派生一个实体子类，并且（至少）提供你使用*datetime*方法所需要的标准*tzinfo*方法的实现。*datetime*模块提供了*timezone*，这是*tzinfo*的一个简单实体子类，它能以与 UTC 的固定差值来表示不同的时区，例如 UTC 本身或北美的 EST 和 EDT。

对于封存操作的特殊要求：一个*tzinfo*子类必须具有可不带参数调用的 `__init__()` 方法，否则它虽然可以被封存，但可能无法再次解封。这是个技术性要求，在未来可能会被取消。

一个*tzinfo*的实体子类可能需要实现以下方法。具体需要实现的方法取决于感知型*datetime*对象如何使用它。如果有疑问，可以简单地全都实现。

tzinfo.**utcoffset**(*dt*)

将本地时间与 UTC 时差返回为一个*timedelta*对象，如果本地时区在 UTC 以东则为正值。如果本地时区在 UTC 以西则为负值。

这表示与 UTC 的总计时差；举例来说，如果一个*tzinfo*对象同时代表时区和 DST 调整，则*utcoffset()*应当返回两者的和。如果 UTC 时差不确定则返回 `None`。在其他情况下返回值必须为一个*timedelta*对象，其取值严格限制于 `-timedelta(hours=24)` 和 `timedelta(hours=24)` 之间（差值的幅度必须小于一天）。大多数*utcoffset()*的实现看起来可能像是以下两者之一:

```
return CONSTANT                 # fixed-offset class
return CONSTANT + self.dst(dt)  # daylight-aware class
```

如果*utcoffset()* 返回值不为 None，则*dst()* 也不应返回 None。

默认的*utcoffset()* 实现会引发*NotImplementedError*。

3.7 版更變: UTC 时差不再限制为一个整数分钟值。

tzinfo.**dst**(*dt*)

将夏令时（DST）调整返回为一个*timedelta* 对象，如果 DST 信息未知则返回 None。

如果 DST 未启用则返回 timedelta(0)。如果 DST 已启用则将差值作为一个*timedelta* 对象返回（参见*utcoffset()* 了解详情）。请注意 DST 差值如果可用，就会直接被加入*utcoffset()* 所返回的 UTC 时差，因此无需额外查询*dst()* 除非你希望单独获取 DST 信息。例如，*datetime.timetuple()* 会调用其*tzinfo* 属性的*dst()* 方法来确定应该如何设置 tm_isdst 旗标，而*tzinfo.fromutc()* 会调用*dst()* 来在跨越时区时处理 DST 的改变。

一个可以同时处理标准时和夏令时的*tzinfo* 子类的实例 *tz* 必须在此情形中保持一致：

```
tz.utcoffset(dt) - tz.dst(dt)
```

必须为具有同样的*tzinfo* 子类实例且 dt.tzinfo == tz 的每个*datetime* 对象 *dt* 返回同样的结果，此表达式会产生时区的“标准时差”，它不应取决于具体日期或时间，只取决于地理位置。*datetime.astimezone()* 的实现依赖此方法，但无法检测违反规则的情况；确保符合规则是程序员的责任。如果一个*tzinfo* 子类不能保证这一点，也许可以重载*tzinfo.fromutc()* 的默认实现以便在任何情况下与 astimezone() 正确配合。

大多数*dst()* 的实现可能会如以下两者之一：

```
def dst(self, dt):
    # a fixed-offset class:  doesn't account for DST
    return timedelta(0)
```

或者：

```
def dst(self, dt):
    # Code to set dston and dstoff to the time zone's DST
    # transition times based on the input dt.year, and expressed
    # in standard local time.

    if dston <= dt.replace(tzinfo=None) < dstoff:
        return timedelta(hours=1)
    else:
        return timedelta(0)
```

默认的*dst()* 实现会引发*NotImplementedError*。

3.7 版更變: DST 差值不再限制为一个整数分钟值。

tzinfo.**tzname**(*dt*)

将对应于*datetime* 对象 *dt* 的时区名称作为字符串返回。*datetime* 模块没有定义任何字符串名称相关内容，也不要求名称有任何特定含义。例如”GMT”, ”UTC”, ”-500”, ”-5:00”, ”EDT”, ”US/Eastern”, ”America/New York” 都是有效的返回值。如果字符串名称未知则返回 None。请注意这是一个方法而不是一个固定的字符串，这主要是因为某些*tzinfo* 子类可能需要根据所传入的特定 *dt* 值返回不同的名称，特别是当*tzinfo* 类要负责处理夏令时的时候。

默认的*tzname()* 实现会引发*NotImplementedError*。

这些方法会被*datetime*或*time*对象调用，用来与它们的同名方法相对应。*datetime*对象会将自身作为传入参数，而*time*对象会将 None 作为传入参数。这样*tzinfo*子类的方法应当准备好接受 *dt* 参数值为 None 或是*datetime*类的实例。

当传入 None 时，应当由类的设计者来决定最佳回应方式。例如，返回 None 适用于希望该类提示时间对象不参与*tzinfo*协议处理。让 utcoffset(None) 返回标准 UTC 时差也许会更有用处，因为并没有其他可用于发现标准时差的约定惯例。

当传入一个*datetime*对象来回应*datetime*方法时，dt.tzinfo 与 *self* 是同一对象。*tzinfo*方法可以依赖这一点，除非用户代码直接调用了*tzinfo*方法。此行为的目的是使得*tzinfo*方法将 *dt* 解读为本地时间，而不需要担心其他时区的相关对象。

还有一个额外的*tzinfo*方法，某个子类可能会希望重载它：

tzinfo.**fromutc**(*dt*)

此方法会由默认的*datetime.astimezone()*实现来调用。当被其调用时，dt.tzinfo 为 *self*，并且 *dt* 的日期和时间数据会被视为表示 UTC 时间，*fromutc()*的目标是调整日期和时间数据，返回一个等价的 datetime 来表示 *self* 的本地时间。

大多数*tzinfo*子类应该能够毫无问题地继承默认的*fromutc()*实现。它的健壮性足以处理固定差值的时区以及同时负责标准时和夏令时的时区，对于后者甚至还能处理 DST 转换时间在各个年份有变化的情况。一个默认*fromutc()*实现可能无法在所有情况下正确处理的例子是（与 UTC 的）标准时差取决于所经过的特定日期和时间，这种情况可能由于政治原因而出现。默认的 astimezone() 和*fromutc()*实现可能无法生成你希望的结果，如果这个结果恰好是跨越了标准时差发生改变的时刻当中的某个小时值的话。

忽略针对错误情况的代码，默认*fromutc()*实现的行为方式如下：

```
def fromutc(self, dt):
    # raise ValueError error if dt.tzinfo is not self
    dtoff = dt.utcoffset()
    dtdst = dt.dst()
    # raise ValueError if dtoff is None or dtdst is None
    delta = dtoff - dtdst  # this is self's standard offset
    if delta:
        dt += delta   # convert to standard local time
        dtdst = dt.dst()
        # raise ValueError if dtdst is None
    if dtdst:
        return dt + dtdst
    else:
        return dt
```

在以下 tzinfo_examples.py 文件中有一些*tzinfo*类的例子：

```
from datetime import tzinfo, timedelta, datetime

ZERO = timedelta(0)
HOUR = timedelta(hours=1)
SECOND = timedelta(seconds=1)

# A class capturing the platform's idea of local time.
# (May result in wrong values on historical times in
#  timezones where UTC offset and/or the DST rules had
#  changed in the past.)
import time as _time

STDOFFSET = timedelta(seconds = -_time.timezone)
if _time.daylight:
```

(下页继续)

(繼續上一頁)

```
    DSTOFFSET = timedelta(seconds = -_time.altzone)
else:
    DSTOFFSET = STDOFFSET

DSTDIFF = DSTOFFSET - STDOFFSET

class LocalTimezone(tzinfo):

    def fromutc(self, dt):
        assert dt.tzinfo is self
        stamp = (dt - datetime(1970, 1, 1, tzinfo=self)) // SECOND
        args = _time.localtime(stamp)[:6]
        dst_diff = DSTDIFF // SECOND
        # Detect fold
        fold = (args == _time.localtime(stamp - dst_diff))
        return datetime(*args, microsecond=dt.microsecond,
                        tzinfo=self, fold=fold)

    def utcoffset(self, dt):
        if self._isdst(dt):
            return DSTOFFSET
        else:
            return STDOFFSET

    def dst(self, dt):
        if self._isdst(dt):
            return DSTDIFF
        else:
            return ZERO

    def tzname(self, dt):
        return _time.tzname[self._isdst(dt)]

    def _isdst(self, dt):
        tt = (dt.year, dt.month, dt.day,
              dt.hour, dt.minute, dt.second,
              dt.weekday(), 0, 0)
        stamp = _time.mktime(tt)
        tt = _time.localtime(stamp)
        return tt.tm_isdst > 0

Local = LocalTimezone()


# A complete implementation of current DST rules for major US time zones.

def first_sunday_on_or_after(dt):
    days_to_go = 6 - dt.weekday()
    if days_to_go:
        dt += timedelta(days_to_go)
    return dt


# US DST Rules
#
# This is a simplified (i.e., wrong for a few cases) set of rules for US
```

(下页继续)

(繼續上一頁)

```
# DST start and end times. For a complete and up-to-date set of DST rules
# and timezone definitions, visit the Olson Database (or try pytz):
# http://www.twinsun.com/tz/tz-link.htm
# http://sourceforge.net/projects/pytz/ (might not be up-to-date)
#
# In the US, since 2007, DST starts at 2am (standard time) on the second
# Sunday in March, which is the first Sunday on or after Mar 8.
DSTSTART_2007 = datetime(1, 3, 8, 2)
# and ends at 2am (DST time) on the first Sunday of Nov.
DSTEND_2007 = datetime(1, 11, 1, 2)
# From 1987 to 2006, DST used to start at 2am (standard time) on the first
# Sunday in April and to end at 2am (DST time) on the last
# Sunday of October, which is the first Sunday on or after Oct 25.
DSTSTART_1987_2006 = datetime(1, 4, 1, 2)
DSTEND_1987_2006 = datetime(1, 10, 25, 2)
# From 1967 to 1986, DST used to start at 2am (standard time) on the last
# Sunday in April (the one on or after April 24) and to end at 2am (DST time)
# on the last Sunday of October, which is the first Sunday
# on or after Oct 25.
DSTSTART_1967_1986 = datetime(1, 4, 24, 2)
DSTEND_1967_1986 = DSTEND_1987_2006

def us_dst_range(year):
    # Find start and end times for US DST. For years before 1967, return
    # start = end for no DST.
    if 2006 < year:
        dststart, dstend = DSTSTART_2007, DSTEND_2007
    elif 1986 < year < 2007:
        dststart, dstend = DSTSTART_1987_2006, DSTEND_1987_2006
    elif 1966 < year < 1987:
        dststart, dstend = DSTSTART_1967_1986, DSTEND_1967_1986
    else:
        return (datetime(year, 1, 1), ) * 2

    start = first_sunday_on_or_after(dststart.replace(year=year))
    end = first_sunday_on_or_after(dstend.replace(year=year))
    return start, end


class USTimeZone(tzinfo):

    def __init__(self, hours, reprname, stdname, dstname):
        self.stdoffset = timedelta(hours=hours)
        self.reprname = reprname
        self.stdname = stdname
        self.dstname = dstname

    def __repr__(self):
        return self.reprname

    def tzname(self, dt):
        if self.dst(dt):
            return self.dstname
        else:
            return self.stdname
```

(下页继续)

(繼續上一頁)

```
    def utcoffset(self, dt):
        return self.stdoffset + self.dst(dt)

    def dst(self, dt):
        if dt is None or dt.tzinfo is None:
            # An exception may be sensible here, in one or both cases.
            # It depends on how you want to treat them.  The default
            # fromutc() implementation (called by the default astimezone()
            # implementation) passes a datetime with dt.tzinfo is self.
            return ZERO
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        # Can't compare naive to aware objects, so strip the timezone from
        # dt first.
        dt = dt.replace(tzinfo=None)
        if start + HOUR <= dt < end - HOUR:
            # DST is in effect.
            return HOUR
        if end - HOUR <= dt < end:
            # Fold (an ambiguous hour): use dt.fold to disambiguate.
            return ZERO if dt.fold else HOUR
        if start <= dt < start + HOUR:
            # Gap (a non-existent hour): reverse the fold rule.
            return HOUR if dt.fold else ZERO
        # DST is off.
        return ZERO

    def fromutc(self, dt):
        assert dt.tzinfo is self
        start, end = us_dst_range(dt.year)
        start = start.replace(tzinfo=self)
        end = end.replace(tzinfo=self)
        std_time = dt + self.stdoffset
        dst_time = std_time + HOUR
        if end <= dst_time < end + HOUR:
            # Repeated hour
            return std_time.replace(fold=1)
        if std_time < start or dst_time >= end:
            # Standard time
            return std_time
        if start <= std_time < end - HOUR:
            # Daylight saving time
            return dst_time


Eastern  = USTimeZone(-5, "Eastern",  "EST", "EDT")
Central  = USTimeZone(-6, "Central",  "CST", "CDT")
Mountain = USTimeZone(-7, "Mountain", "MST", "MDT")
Pacific  = USTimeZone(-8, "Pacific",  "PST", "PDT")
```

请注意同时负责标准时和夏令时的*tzinfo*子类在每年两次的 DST 转换点上会出现不可避免的微妙问题。具体而言，考虑美国东部时区 (UTC -0500)，它的 EDT 从三月的第二个星期天 1:59 (EST) 之后一分钟开始，并在十一月的第一天星期天 1:59 (EDT) 之后一分钟结束:

```
  UTC    3:MM  4:MM  5:MM  6:MM  7:MM  8:MM
  EST  22:MM 23:MM  0:MM  1:MM  2:MM  3:MM
```

(下页继续)

(繼續上一頁)

```
  EDT  23:MM  0:MM  1:MM  2:MM  3:MM  4:MM

start  22:MM 23:MM  0:MM  1:MM  3:MM  4:MM

  end  23:MM  0:MM  1:MM  1:MM  2:MM  3:MM
```

当 DST 开始时（即"start" 行），本地时钟从 1:59 跳到 3:00。形式为 2:MM 的时间值在那一天是没有意义的，因此在 DST 开始那一天 `astimezone(Eastern)` 不会输出包含 `hour == 2` 的结果。例如，在 2016 年春季时钟向前调整时，我们得到:

```
>>> from datetime import datetime, timezone
>>> from tzinfo_examples import HOUR, Eastern
>>> u0 = datetime(2016, 3, 13, 5, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname())
...
05:00:00 UTC = 00:00:00 EST
06:00:00 UTC = 01:00:00 EST
07:00:00 UTC = 03:00:00 EDT
08:00:00 UTC = 04:00:00 EDT
```

当 DST 结束时（见"end" 行），会有更糟糕的潜在问题：本地时间值中有一个小时是不可能没有歧义的：夏令时的最后一小时。即以北美东部时间表示当天夏令时结束时的形式为 5:MM UTC 的时间。本地时钟从 1:59 (夏令时) 再次跳回到 1:00 (标准时)。形式为 1:MM 的本地时间就是有歧义的。此时 `astimezone()` 是通过将两个相邻的 UTC 小时映射到两个相同的本地小时来模仿本地时钟的行为。在这个北美东部时间的示例中，形式为 5:MM 和 6:MM 的 UTC 时间在转换为北美东部时间时都将被映射到 1:MM，但前一个时间会将*fold*属性设为 0 而后一个时间会将其设为 1。例如，在 2016 年秋季时钟往回调整时，我们得到:

```
>>> u0 = datetime(2016, 11, 6, 4, tzinfo=timezone.utc)
>>> for i in range(4):
...     u = u0 + i*HOUR
...     t = u.astimezone(Eastern)
...     print(u.time(), 'UTC =', t.time(), t.tzname(), t.fold)
...
04:00:00 UTC = 00:00:00 EDT 0
05:00:00 UTC = 01:00:00 EDT 0
06:00:00 UTC = 01:00:00 EST 1
07:00:00 UTC = 02:00:00 EST 0
```

请注意不同的*datetime*实例仅通过*fold*属性值来加以区分，它们在比较时会被视为相等。

不允许时间显示存在歧义的应用需要显式地检查*fold*属性的值，或者避免使用混合式的*tzinfo*子类；当使用*timezone*或者任何其他固定差值的*tzinfo*子类例如仅表示 EST（固定差值 -5 小时）或仅表示 EDT（固定差值 -4 小时）的类时是不会有歧义的。

也参考:

zoneinfo *datetime* 模块有一个基本*timezone*类（用来处理任意与 UTC 的固定时差）及其*timezone.utc*属性（一个 UTC 时区实例）。

`zoneinfo` brings the *IANA timezone database* (also known as the Olson database) to Python, and its usage is recommended.

IANA 时区数据库 该时区数据库 (通常称为 tz, tzdata 或 zoneinfo) 包含大量代码和数据用来表示全球许多有代表性的地点的本地时间的历史信息。它会定期进行更新以反映各政治实体对时区边界、UTC 差值和夏令时规则的更改。

8.1.9 timezone 对象

timezone 类是*tzinfo* 的子类，它的每个实例都代表一个以与 UTC 的固定时差来定义的时区。

此类的对象不可被用于代表某些特殊地点的时区信息，这些地点在一年的不同日期会使用不同的时差，或是在历史上对民用时间进行过调整。

class datetime.**timezone**(*offset*, *name=None*)

offset 参数必须指定为一个*timedelta* 对象，表示本地时间与 UTC 的时差。它必须严格限制于 `-timedelta(hours=24)` 和 `timedelta(hours=24)` 之间，否则会引发*ValueError*。

name 参数是可选的。如果指定则必须为一个字符串，它将被用作*datetime.tzname()* 方法的返回值。

3.2 版新加入.

3.7 版更變: UTC 时差不再限制为一个整数分钟值。

timezone.**utcoffset**(*dt*)

返回当*timezone* 实例被构造时指定的固定值。

dt 参数会被忽略。返回值是一个*timedelta* 实例，其值等于本地时间与 UTC 之间的时差。

3.7 版更變: UTC 时差不再限制为一个整数分钟值。

timezone.**tzname**(*dt*)

返回当*timezone* 实例被构造时指定的固定值。

如果没有在构造器中提供 *name*，则 `tzname(dt)` 所返回的名称将根据 `offset` 值按以下规则生成。如果 *offset* 为 `timedelta(0)`，则名称为 "UTC"，否则为字符串 `UTC±HH:MM`，其中 ± 为 `offset` 的正负符号，HH 和 MM 分别为表示 `offset.hours` 和 `offset.minutes` 的两个数码。

3.6 版更變: 由 `offset=timedelta(0)` 生成的名称现在为简单的 *'UTC'* 而不再是 `'UTC+00:00'`。

timezone.**dst**(*dt*)

总是返回 `None`。

timezone.**fromutc**(*dt*)

返回 `dt + offset`。*dt* 参数必须为一个感知型*datetime* 实例，其中 `tzinfo` 值设为 `self`。

类属性:

timezone.**utc**

UTC 时区，`timezone(timedelta(0))`。

8.1.10 strftime() 和 strptime() 的行为

date, *datetime* 和*time* 对象都支持 `strftime(format)` 方法，可用来创建由一个显式格式字符串所控制的表示时间的字符串。

相反地，*datetime.strptime()* 类会根据表示日期和时间的字符串和相应的格式字符串来创建一个*datetime* 对象。

下表提供了 `strftime()` 与 `strptime()` 的高层级比较:

	strftime	strptime
用法	根据给定的格式将对象转换为字符串	将字符串解析为给定相应格式的*datetime* 对象
方法类型	实例方法	类方法
方法	*date*; *datetime*; *time*	*datetime*
签名	`strftime(format)`	`strptime(date_string, format)`

`strftime()` 和 `strptime()` Format Codes

以下列表显示了 1989 版 C 标准所要求的全部格式代码，它们在带有标准 C 实现的所有平台上均可用。

指令	意义	示例	F解
%a	当地工作日的缩写。	Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)	(1)
%A	本地化的星期中每日的完整名称。	Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)	(1)
%w	以十进制数显示的工作日，其中 0 表示星期日，6 表示星期六。	0, 1, ..., 6	
%d	补零后，以十进制数显示的月份中的一天。	01, 02, ..., 31	(9)
%b	当地月份的缩写。	Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)	(1)
%B	本地化的月份全名。	January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)	(1)
%m	补零后，以十进制数显示的月份。	01, 02, ..., 12	(9)
%y	补零后，以十进制数表示的，不带世纪的年份。	00, 01, ..., 99	(9)
%Y	十进制数表示的带世纪的年份。	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(2)
%H	以补零后的十进制数表示的小时（24 小时制）。	00, 01, ..., 23	(9)
%I	以补零后的十进制数表示的小时（12 小时制）。	01, 02, ..., 12	(9)
%p	本地化的 AM 或 PM 。	AM, PM (en_US); am, pm (de_DE)	(1), (3)
%M	补零后，以十进制数显示的分钟。	00, 01, ..., 59	(9)
%S	补零后，以十进制数显示的秒。	00, 01, ..., 59	(4), (9)
%f	Microsecond as a decimal number, zero-padded to 6 digits.	000000, 000001, ..., 999999	(5)
%z	UTC 偏移量，格式为 ±HHMM[SS[.ffffff]]（如果是简单型对象则为空字符串）。	(空), +0000, -0400, +1030, +063415, -030712.345216	(6)
%Z	时区名称（如果对象为简单型则为空字符串）。	(空), UTC, GMT	(6)

为了方便起见，还包括了 C89 标准不需要的其他一些指令。这些参数都对应于 ISO 8601 日期值。

指令	意义	示例	F解
%G	带有世纪的 ISO 8601 年份，表示包含大部分 ISO 星期 (%V) 的年份。	0001, 0002, ..., 2013, 2014, ..., 9998, 9999	(8)
%u	以十进制数显示的 ISO 8601 星期中的日序号，其中 1 表示星期一。	1, 2, ..., 7	
%V	以十进制数显示的 ISO 8601 星期，以星期一作为每周的第一天。第 01 周为包含 1 月 4 日的星期。	01, 02, ..., 53	(8), (9)

这些代码可能不是在所有平台上都可与 strftime() 方法配合使用。ISO 8601 年份和 ISO 8601 星期指令并不能与上面的年份和星期序号指令相互替代。调用 strptime() 时传入不完整或有歧义的 ISO 8601 指令将引发*ValueError*。

The full set of format codes supported varies across platforms, because Python calls the platform C library's strftime() function, and platform variations are common. To see the full set of format codes supported on your platform, consult the *strftime(3)* documentation. There are also differences between platforms in handling of unsupported format specifiers.

3.6 版新加入: 增加了 %G, %u 和 %V。

技术细节

总体而言，d.strftime(fmt) 类似于*time* 模块的 time.strftime(fmt, d.timetuple())，但是并非所有对象都支持 timetuple() 方法。

对于*datetime.strptime()* 类方法，默认值为 1900-01-01T00:00:00.000: 任何未在格式字符串中指定的部分都将从默认值中提取。[4]

使用 datetime.strptime(date_string, format) 等价于:

```
datetime(*(time.strptime(date_string, format)[0:6]))
```

除非格式中包含秒以下的部分或时区差值信息，它们在 datetime.strptime 中受支持但会被 time.strptime 所丢弃。

对于*time* 对象，年、月、日的格式代码不应被使用，因为*time* 对象没有这些值。如果它们被使用，则年份将被替换为 1900，而月和日将被替换为 1。

对于*date* 对象，时、分、秒和微秒的格式代码不应被使用，因为*date* 对象没有这些值。如果它们被使用，则它们都将被替换为 0。

出于相同的原因，对于包含当前区域设置字符集所无法表示的 Unicode 码位的格式字符串的处理方式也取决于具体平台。在某些平台上这样的码位会不加修改地原样输出，而在其他平台上 strftime 则可能引发*UnicodeError* 或只返回一个空字符串。

F解:

(1) 由于此格式依赖于当前区域设置，因此对具体输出值应当保持谨慎预期。字段顺序会发生改变（例如"month/day/year" 与"day/month/year"），并且输出可能包含使用区域设置所指定的默认编码格式的 Unicode 字符（例如如果当前区域为 ja_JP，则默认编码格式可能为 eucJP, SJIS 或 utf-8 中的一个；使用*locale.getlocale()* 可确定当前区域设置的编码格式）。

(2) strptime() 方法能够解析整个 [1, 9999] 范围内的年份，但 < 1000 的年份必须加零填充为 4 位数字宽度。

[4] 传入 datetime.strptime('Feb 29', '%b %d') 将导致错误，因为 1900 不是闰年。

3.2 版更變: 在之前的版本中，`strftime()` 方法只限于 >= 1900 的年份。

3.3 版更變: 在版本 3.2 中，`strftime()` 方法只限于 years >= 1000。

(3) 当与 `strptime()` 方法一起使用时，如果使用 `%I` 指令来解析小时，`%p` 指令只影响输出小时字段。

(4) 与*time* 模块不同的是，*datetime* 模块不支持闰秒。

(5) 当与 `strptime()` 方法一起使用时，`%f` 指令可接受一至六个数码及左边的零填充。`%f` 是对 C 标准中格式字符集的扩展（但单独在 datetime 对象中实现，因此它总是可用）。

(6) 对于简单型对象，`%z` and `%Z` 格式代码会被替换为空字符串。

对于一个感知型对象而言：

%z `utcoffset()` 会被转换为 `±HHMM[SS[.ffffff]]` 形式的字符串，其中 `HH` 为给出 UTC 时差的小时部分的 2 位数码字符串，`MM` 为给出 UTC 时差的分钟部分的 2 位数码字符串，`SS` 为给出 UTC 时差的秒部分的 2 位数码字符串，而 `ffffff` 为给出 UTC 时差的微秒部分的 6 位数码字符串。当时差为整数秒时 `ffffff` 部分将被省略，而当时差为整数分钟时 `ffffff` 和 `SS` 部分都将被省略。例如，如果 `utcoffset()` 返回 `timedelta(hours=-3, minutes=-30)`，则 `%z` 会被替换为字符串 `'-0330'`。

3.7 版更變: UTC 时差不再限制为一个整数分钟值。

3.7 版更變: 当提供 `%z` 指令给 `strptime()` 方法时，UTC 差值可以在时、分和秒之间使用冒号分隔符。例如，`'+01:00:00'` 将被解读为一小时的差值。此外，提供 `'Z'` 就相当于 `'+00:00'`。

%Z 在 `strftime()` 中，如果 `tzname()` 返回 `None` 则 `%Z` 会被替换为一个空字符串；在其他情况下 `%Z` 会被替换为返回值，该值必须为一个字符串。

`strptime()` 仅接受特定的 `%Z` 值:

1. 你的机器的区域设置可以是 `time.tzname` 中的任何值
2. 硬编码的值 `UTC` 和 `GMT`

这样生活在日本的人可用的值为 `JST`, `UTC` 和 `GMT`，但可能没有 `EST`。它将引发 `ValueError` 表示无效的值。

3.2 版更變: 当提供 `%z` 指令给 `strptime()` 方法时，将产生一个感知型*datetime* 对象。结果的 `tzinfo` 将被设为一个*timezone* 实例。

(7) 当与 `strptime()` 方法一起使用时，`%U` 和 `%W` 仅用于指定星期几和日历年份 (`%Y`) 的计算。

(8) 类似于 `%U` 和 `%W`，`%V` 仅用于在 `strptime()` 格式字符串中指定星期几和 ISO 年份 (`%G`) 的计算。还要注意 `%G` 和 `%Y` 是不可交换的。

(9) 当于 `strptime()` 方法一起使用时，前导的零在格式 `%d`, `%m`, `%H`, `%I`, `%M`, `%S`, `%J`, `%U`, `%W` 和 `%V` 中是可选的。格式 `%y` 不要求有前导的零。

𝔽解

8.2 zoneinfo --- IANA 时区支持

3.9 版新加入.

---

zoneinfo 模块根据 **PEP 615** 的最初说明提供了具体的时区实现来支持 IANA 时区数据库。按照默认设置，*zoneinfo* 会在可能的情况下使用系统的时区数据；如果系统时区数据不可用，该库将回退为使用 PyPI 上提供的 tzdata 第一方包。

也参考:

模块: *datetime* 提供*time* 和*datetime* 类型，*ZoneInfo* 类被设计为可配合这两个类型使用。

包 tzdata 由 CPython 核心开发者维护以通过 PyPI 提供时区数据的第一方包。

8.2.1 使用 ZoneInfo

ZoneInfo 是*datetime.tzinfo* 抽象基类的具体实现，其目标是通过构造器、*datetime.replace* 方法或*datetime.astimezone* 来与 tzinfo 建立关联:

```
>>> from zoneinfo import ZoneInfo
>>> from datetime import datetime, timedelta

>>> dt = datetime(2020, 10, 31, 12, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-10-31 12:00:00-07:00

>>> dt.tzname()
'PDT'
```

以此方式构造的日期时间对象可兼容日期时间运算并可在无需进一步干预的情况下处理夏令时转换:

```
>>> dt_add = dt + timedelta(days=1)

>>> print(dt_add)
2020-11-01 12:00:00-08:00

>>> dt_add.tzname()
'PST'
```

这些时区还支持在 **PEP 495** 中引入的*fold*。在可能导致时间歧义的时差转换中（例如夏令时到标准时的转换），当 fold=0 时会使用转换之前的时差，而当 fold=1 时则使用转换之后的时差，例如:

```
>>> dt = datetime(2020, 11, 1, 1, tzinfo=ZoneInfo("America/Los_Angeles"))
>>> print(dt)
2020-11-01 01:00:00-07:00

>>> print(dt.replace(fold=1))
2020-11-01 01:00:00-08:00
```

当执行来自另一时区的转换时，fold 将被设置为正确的值:

```
>>> from datetime import timezone
>>> LOS_ANGELES = ZoneInfo("America/Los_Angeles")
>>> dt_utc = datetime(2020, 11, 1, 8, tzinfo=timezone.utc)

>>> # Before the PDT -> PST transition
>>> print(dt_utc.astimezone(LOS_ANGELES))
2020-11-01 01:00:00-07:00

>>> # After the PDT -> PST transition
>>> print((dt_utc + timedelta(hours=1)).astimezone(LOS_ANGELES))
2020-11-01 01:00:00-08:00
```

8.2.2 数据源

zoneinfo 模块不直接提供时区数据，而是在可能的情况下从系统时区数据库或 PyPI 上的第一方包 tzdata 获取时区信息。某些系统，重要的一点是 Windows 系统也包括在内，并没有可用的 IANA 数据库，因此对于要保证获取时区信息的跨平台兼容性的项目，推荐对 tzdata 声明依赖。如果系统数据和 tzdata 均不可用，则所有对*ZoneInfo* 的调用都将引发*ZoneInfoNotFoundError*。

配置数据源

当 ZoneInfo(key) 被调用时，此构造器首先会在*TZPATH* 所指定的目录下搜索匹配 key 的文件，失败时则会在 tzdata 包中查找匹配。此行为可通过三种方式来配置:

1. 默认的*TZPATH* 未通过其他方式指定时可在编译时 进行配置。
2. *TZPATH* 可使用环境变量 进行配置。
3. 在运行时，搜索路径可使用*reset_tzpath()* 函数来修改。

编译时配置

默认的*TZPATH* 包括一些时区数据库的通用布署位置（Windows 除外，该系统没有时区数据的 “通用” 位置）。在 POSIX 系统中，下游分发者和从源码编译 Python 的开发者知道系统时区数据布署位置，他们可以通过指定编译时选项 TZPATH (或者更常见的是通过 configure 旗标 --with-tzpath) 来改变默认的时区路径，该选项应当是一个由*os.pathsep* 分隔的字符串。

在所有平台上，配置值会在*sysconfig.get_config_var()* 中以 TZPATH 键的形式提供。

环境配置

当初始化*TZPATH* 时（在导入时或不带参数调用*reset_tzpath()* 时），zoneinfo 模块将使用环境变量 PYTHONTZPATH，如果变量存在则会设置搜索路径。

PYTHONTZPATH

这是一个以*os.pathsep* 分隔的字符串，其中包含要使用的时区搜索路径。它必须仅由绝对路径而非相对路径组成。在 PYTHONTZPATH 中指定的相对路径部分将不会被使用，但在其他情况下当指定相对路径时的行为该实现是有定义的；CPython 将引发*InvalidTZPathWarning*，而其他实现可自由地忽略错误部分或是引发异常。

要设置让系统忽略系统数据并改用 tzdata 包，请设置 PYTHONTZPATH=""。

运行时配置

TZ 搜索路径也可在运行时使用*reset_tzpath()* 函数来配置。通常并不建议如此操作，不过在需要使用指定时区路径（或者需要禁止访问系统时区）的测试函数中使用它则是合理的。

8.2.3 ZoneInfo 类

class zoneinfo.**ZoneInfo**(*key*)

一个具体的*datetime.tzinfo* 子类，它代表一个由字符串 key 所指定的 IANA 时区。对主构造器的调用将总是返回可进行标识比较的对象；但是另一种方式，对所有的 key 值通过*ZoneInfo.clear_cache()* 禁止缓存失效，对以下断言将总是为真值:

```
a = ZoneInfo(key)
b = ZoneInfo(key)
assert a is b
```

key 必须采用相对的标准化 POSIX 路径的形式，其中没有对上一层级的引用。如果传入了不合要求的键则构造器将引发*ValueError*。

如果没有找到匹配 key 的文件，构造器将引发*ZoneInfoNotFoundError*。

ZoneInfo 类具有两个替代构造器:

classmethod ZoneInfo.**from_file**(*fobj*, */*, *key=None*)

基于一个返回字节串的文件类对象（例如一个以二进制模式打开的文件或是一个*io.BytesIO* 对象）构造 ZoneInfo 对象。不同于主构造器，此构造器总是会构造一个新对象。

key 形参设置时区名称以供 __str__() 和 __repr__() 使用。

由此构造器创建的对象不可被封存 (参见*pickling*)。

classmethod ZoneInfo.**no_cache**(*key*)

一个绕过构造器缓存的替代构造器。它与主构造器很相似，但每次调用都会返回一个新对象。此构造器在进行测试或演示时最为适用，但它也可以被用来创建具有不同缓存失效策略的系统。

由此构造器创建的对象在被解封时也会绕过反序列化进程的缓存。

> **警示:** 使用此构造器可以会以令人惊讶的方式改变日期时间对象的语义，只有在你确定你的需求时才使用它。

也可以使用以下的类方法:

classmethod ZoneInfo.**clear_cache**(*, *only_keys=None*)

一个可在 ZoneInfo 类上禁用缓存的方法。如果不传入参数，则会禁用所有缓存并且下次对每个键调用主构造器将返回一个新实例。

如果将一个键名称的可迭代对象传给 only_keys 形参，则将只有指定的键会被从缓存中移除。传给 only_keys 但在缓存中找不到的键会被忽略。

> **警告:** 发起调用此函数可能会以令人惊讶的方式改变使用 ZoneInfo 的日期时间对象的语义；这会修改进程范围内的全局状态并因此可能产生大范围的影响。只有在你确定你的需求时才使用它。

该类具有一个属性:

ZoneInfo.**key**

这是一个只读的*attribute*，它返回传给构造器的 key 的值，该值应为一个 IANA 时区数据库的查找键 (例如 America/New_York, Europe/Paris 或 Asia/Tokyo)。

对于不指定 key 形参而是基于文件构造时区，该属性将设为 None。

> **備🄵:** 尽管将这些信息暴露给最终用户是一种比较普通的做法，但是这些值被设计作为代表相关时区的主键而不一定是面向用户的元素。CLDR (Unicode 通用区域数据存储库) 之类的项目可被用来根据这些键获取更为用户友好的字符串。

字符串表示

当在*ZoneInfo* 对象上调用*str* 时返回的字符串表示默认会使用*ZoneInfo.key* 属性（参见该属性文档中的用法注释）：

```
>>> zone = ZoneInfo("Pacific/Kwajalein")
>>> str(zone)
'Pacific/Kwajalein'

>>> dt = datetime(2020, 4, 1, 3, 15, tzinfo=zone)
>>> f"{dt.isoformat()} [{dt.tzinfo}]"
'2020-04-01T03:15:00+12:00 [Pacific/Kwajalein]'
```

对于基于文件而非指定 key 形参所构建的对象，str 会回退为调用*repr()*。ZoneInfo 的 repr 是由具体实现定义的并且不一定会在不同版本间保持稳定，但它保证不会是一个有效的 ZoneInfo 键。

封存序列化

ZoneInfo 对象的序列化是基于键的，而不是序列化所有过渡数据，并且基于文件构造的 ZoneInfo 对象（即使是指定了 key 值的对象）不能被封存。

ZoneInfo 文件的行为取决于它的构造方式:

1. ZoneInfo(key): 当使用主构造器构造时，会基于键序列化一个 ZoneInfo 对象，而当反序列化时，反序列化过程会使用主构造器，因此预期它们与其他对同一时区的引用会是同一对象。例如，如果 europe_berlin_pkl 是一个包含基于 ZoneInfo("Europe/Berlin") 构建的封存数据的字符串，你可以预期出现以下的行为:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl)
   >>> a is b
   True
   ```

2. ZoneInfo.no_cache(key): 当通过绕过缓存的构造器构造时，ZoneInfo 对象也会基于键序列化，但当反序列化时，反序列化过程会使用绕过缓存的构造器。如果 europe_berlin_pkl_nc 是一个包含基于 ZoneInfo.no_cache("Europe/Berlin") 构造的封存数据的字符串，你可以预期出现以下的行为:

   ```
   >>> a = ZoneInfo("Europe/Berlin")
   >>> b = pickle.loads(europe_berlin_pkl_nc)
   >>> a is b
   False
   ```

3. ZoneInfo.from_file(fobj, /, key=None): 当通过文件构造时，ZoneInfo 对象会在封存时引发异常。如果最终用户想要封存通过文件构造的 ZoneInfo，则推荐他们使用包装类型或自定义序列化函数: 或者基于键序列化，或者存储文件对象的内容并将其序列化。

该序列化方法要求所需键的时区数据在序列化和反序列化中均可用，类似于在序列化和反序列化环境中都预期存在对类和函数的引用的方式。这还意味着在具有不同时区数据版本的环境中当解封被封存的 ZoneInfo 时并不会保证结果的一致性。

8.2.4 函数

zoneinfo.**available_timezones**()
获取一个包含可用 IANA 时区的在时区路径的任何位置均可用的全部有效键的集合。每次调用该函数时都会重新计算。

此函数仅包括规范时区名称而不包括“特殊”时区如位于 posix/ 和 right/ 目录下的时区或 posixrules 时区。

> **警示：** 此函数可能会打开大量的文件，因为确定时区路径上某个文件是否为有效时区的最佳方式是读取开头位置的“魔术字符串”。

備🄵： 这些值并不被设计用来对外公开给最终用户；对于面向用户的元素，应用程序应当使用 CLDR (Unicode 通用区域数据存储库) 之类来获取更为用户友好的字符串。另请参阅*ZoneInfo.key* 中的提示性说明。

---

zoneinfo.**reset_tzpath**(*to=None*)
设置或重置模块的时区搜索路径 (*TZPATH*)。当不带参数调用时，*TZPATH* 会被设为默认值。

调用 reset_tzpath 将不会使*ZoneInfo* 缓存失效，因而在缓存未命中的情况下对主 ZoneInfo 构造器的调用将只使用新的 TZPATH。

to 形参必须是由字符串或*os.PathLike* 组成的*sequence* 或而不是字符串，它们必须都是绝对路径。如果所传入的不是绝对路径则将引发*ValueError*。

8.2.5 全局变量

zoneinfo.**TZPATH**
一个表示时区搜索路径的只读序列 -- 当通过键构造 ZoneInfo 时，键会与 TZPATH 中的每个条目进行合并，并使用所找到的第一个文件。

TZPATH 可以只包含绝对路径，绝不包含相对路径，无论它是如何配置的。

zoneinfo.TZPATH 所指向的对象可能随着对*reset_tzpath()* 的调用而改变，因此推荐使用 zoneinfo.TZPATH 而不是从 zoneinfo 导入 TZPATH 或是将 zoneinfo.TZPATH 赋值给一个长期变量。

有关配置时区搜索路径的更多信息，请参阅配置数据源。

8.2.6 异常与警告

exception zoneinfo.**ZoneInfoNotFoundError**
当一个*ZoneInfo* 对象的构造由于在系统中找不到指定的键而失败时引发。这是*KeyError* 的一个子类。

exception zoneinfo.**InvalidTZPathWarning**
当 *PYTHONTZPATH* 包含将被过滤掉的无效组件，例如一个相对路径时引发。

8.3 calendar --- 日历相关函数

源代码： Lib/calendar.py

这个模块让你可以输出像 Unix **cal** 那样的日历，它还提供了其它与日历相关的实用函数。默认情况下，这些日历把星期一当作一周的第一天，星期天为一周的最后一天（按照欧洲惯例）。可以使用 *setfirstweekday()* 方法设置一周的第一天为星期天 (6) 或者其它任意一天。使用整数作为指定日期的参数。更多相关的函数，参见 *datetime* 和 *time* 模块。

在这个模块中定义的函数和类都基于一个理想化的日历，现行公历向过去和未来两个方向无限扩展。这与 Dershowitz 和 Reingold 的书" 历法计算" 中所有计算的基本日历 -- "proleptic Gregorian" 日历的定义相符合。ISO 8601 标准还规定了 0 和负数年份。0 年指公元前 1 年，-1 年指公元前 2 年，依此类推。

class calendar.**Calendar**(*firstweekday=0*)

创建一个 *Calendar* 对象。*firstweekday* 是一个整数，用于指定一周的第一天。0 是星期一（默认值），6 是星期天。

Calendar 对象提供了一些可被用于准备日历数据格式化的方法。这个类本身不执行任何格式化操作。这部分任务应由子类来完成。

Calendar 类的实例有下列方法：

iterweekdays()

返回一个迭代器，迭代器的内容为一星期的数字。迭代器的第一个值与 *firstweekday* 属性的值一至。

itermonthdates(*year*, *month*)

返回一个迭代器，迭代器的内容为 *year* 年 *month* 月 (1-12) 的日期。这个迭代器返回当月的所有日期 (*datetime.date* 对象)，日期包含了本月头尾用于组成完整一周的日期。

itermonthdays(*year*, *month*)

返回一个迭代器，迭代器的内容与 *itermonthdates()* 类似，为 *year* 年 *month* 月的日期，但不受 *datetime.date* 范围限制。返回的日期为当月每一天的日期对应的天数。对于不在当月的日期，显示为 0。

itermonthdays2(*year*, *month*)

返回一个迭代器，迭代器的内容与 *itermonthdates()* 类似为 *year* 年 *month* 月的日期，但不受 *datetime.date* 范围的限制。迭代器中的元素为一个由日期和代表星期几的数字组成的的元组。

itermonthdays3(*year*, *month*)

返回一个迭代器，迭代器的内容与 *itermonthdates()* 类似为 *year* 年 *month* 月的日期，但不受 *datetime.date* 范围的限制。迭代器的元素为一个由年，月，日组成的元组。

3.7 版新加入.

itermonthdays4(*year*, *month*)

返回一个迭代器，迭代器的内容与 *itermonthdates()* 类似为 *year* 年 *month* 月的日期，但不受 *datetime.date* 范围的限制。迭代器的元素为一个由年，月，日和代表星期几的数字组成的元组。

3.7 版新加入.

monthdatescalendar(*year*, *month*)

返回一个表示指定年月的周列表。周列表由七个 *datetime.date* 对象组成。

monthdays2calendar(*year*, *month*)

返回一个表示指定年月的周列表。周列表由七个代表日期的数字和代表周几的数字组成的二元元组。

monthdayscalendar(*year*, *month*)
返回一个表示指定年月的周列表。周列表由七个代表日期的数字组成。

yeardatescalendar(*year*, *width=3*)
返回可以用来格式化的指定年月的数据。返回的值是一个列表，列表是月份组成的行。每一行包含了最多 *width* 个月 (默认为 3)。每个月包含了 4 到 6 周，每周包含 1--7 天。每一天使用`datetime.date`对象。

yeardays2calendar(*year*, *width=3*)
返回可以用来模式化的指定年月的数据 (与`yeardatescalendar()`类似)。周列表的元素是由表示日期的数字和表示星期几的数字组成的元组。不在这个月的日子为 0。

yeardayscalendar(*year*, *width=3*)
返回可以用来模式化的指定年月的数据 (与`yeardatescalendar()`类似)。周列表的元素是表示日期的数字。不在这个月的日子为 0。

class calendar.**TextCalendar**(*firstweekday=0*)
可以使用这个类生成纯文本日历。

`TextCalendar`实例有以下方法：

formatmonth(*theyear*, *themonth*, *w=0*, *l=0*)
返回一个多行字符串来表示指定年月的日历。*w* 为日期的宽度，但始终保持日期居中。*l* 指定了每星期占用的行数。以上这些还依赖于构造器或者`setfirstweekday()`方法指定的周的第一天是哪一天。

prmonth(*theyear*, *themonth*, *w=0*, *l=0*)
与`formatmonth()`方法一样，返回一个月的日历。

formatyear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
返回一个多行字符串，这个字符串为一个 *m* 列日历。可选参数 *w*, *l*, 和 *c* 分别表示日期列数，周的行数，和月之间的间隔。同样，以上这些还依赖于构造器或者`setfirstweekday()`指定哪一天为一周的第一天。日历的第一年由平台依赖于使用的平台。

pryear(*theyear*, *w=2*, *l=1*, *c=6*, *m=3*)
与`formatyear()`方法一样，返回一整年的日历。

class calendar.**HTMLCalendar**(*firstweekday=0*)
可以使用这个类生成 HTML 日历。

`HTMLCalendar`实例有以下方法：

formatmonth(*theyear*, *themonth*, *withyear=True*)
返回一个 HTML 表格作为指定年月的日历。*withyear* 为真，则年份将会包含在表头，否则只显示月份。

formatyear(*theyear*, *width=3*)
返回一个 HTML 表格作为指定年份的日历。*width* (默认为 3) 用于规定每一行显示月份的数量。

formatyearpage(*theyear*, *width=3*, *css='calendar.css'*, *encoding=None*)
返回一个完整的 HTML 页面作为指定年份的日历。*width**(默认为 3) 用于规定每一行显示的月份数量。**css* 为层叠样式表的名字。如果不使用任何层叠样式表，可以使用`None`。*encoding* 为输出页面的编码 (默认为系统的默认编码)。

`HTMLCalendar`有以下属性，你可以重载它们来自定义应用日历的样式。

cssclasses
一个对应星期一到星期天的 CSS class 列表。默认列表为

```
cssclasses = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

可以向每天加入其它样式

```
cssclasses = ["mon text-bold", "tue", "wed", "thu", "fri", "sat", "sun red"]
```

需要注意的是，列表的长度必须为 7。

cssclass_noday
工作日的 CSS 类在上个月或下个月发生。

3.7 版新加入.

cssclasses_weekday_head
用于标题行中的工作日名称的 CSS 类列表。默认值与*cssclasses* 相同。

3.7 版新加入.

cssclass_month_head
月份的头 CSS 类（由 formatmonthname() 使用）。默认值为 "month" 。

3.7 版新加入.

cssclass_month
某个月的月历的 CSS 类（由*formatmonth()* 使用）。默认值为 "month" 。

3.7 版新加入.

cssclass_year
某年的年历的 CSS 类（由*formatyear()* 使用）。默认值为 "year" 。

3.7 版新加入.

cssclass_year_head
年历的 · 表头 CSS 类（由*formatyear()* 使用）。默认值为 "year" 。

3.7 版新加入.

需要注意的是，尽管上面命名的样式类都是单独出现的 (如：cssclass_month cssclass_noday)，但我们可以使用空格将样式类列表中的多个元素分隔开，例如：

```
"text-bold text-red"
```

下面是一个如何自定义 HTMLCalendar 的示例

```
class CustomHTMLCal(calendar.HTMLCalendar):
    cssclasses = [style + " text-nowrap" for style in
                  calendar.HTMLCalendar.cssclasses]
    cssclass_month_head = "text-center month-head"
    cssclass_month = "text-center month"
    cssclass_year = "text-italic lead"
```

class calendar.**LocaleTextCalendar**(*firstweekday=0*, *locale=None*)
这个子类*TextCalendar* 可以在构造函数中传递一个语言环境名称，并且返回月份和星期几的名称在特定语言环境中。如果此语言环境包含编码，则包含月份和工作日名称的所有字符串将作为 unicode 返回。

class calendar.**LocaleHTMLCalendar**(*firstweekday=0*, *locale=None*)
This subclass of *HTMLCalendar* can be passed a locale name in the constructor and will return month and weekday names in the specified locale. If this locale includes an encoding all strings containing month and weekday names will be returned as unicode.

備🄵: 这两个类的 formatweekday() 和 formatmonthname() 方法临时更改 dang 当前区域至给定 *locale* 。由于当前的区域设置是进程范围的设置，因此它们不是线程安全的。

对于简单的文本日历，这个模块提供了以下方法。

calendar.**setfirstweekday**(*weekday*)
设置每一周的开始 (0 表示星期一，6 表示星期天)。calendar 还提供了 MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY 和 SUNDAY 几个常量方便使用。例如，设置每周的第一天为星期天

```
import calendar
calendar.setfirstweekday(calendar.SUNDAY)
```

calendar.**firstweekday**()
返回当前设置的每星期的第一天的数值。

calendar.**isleap**(*year*)
如果 *year* 是闰年则返回 *True*，否则返回 *False*。

calendar.**leapdays**(*y1*, *y2*)
返回在范围 *y1* 至 *y2* （包含 y1 和 y2 ）之间的闰年的年数，其中 *y1* 和 *y2* 是年份。

此函数适用于跨越一个世纪变化的范围。

calendar.**weekday**(*year*, *month*, *day*)
返回某年（1970 -- ...），某月（1 -- 12 ），某日（1 -- 31 ）是星期几（0 是星期一）。

calendar.**weekheader**(*n*)
返回一个包含星期几的缩写名的头。*n* 指定星期几缩写的字符宽度。

calendar.**monthrange**(*year*, *month*)
返回指定 年份 的指定 月份 的第一天是星期几和这个月的天数。

calendar.**monthcalendar**(*year*, *month*)
返回表示一个月的日历的矩阵。每一行代表一周；此月份外的日子由零表示。每周从周一开始，除非使用 *setfirstweekday()* 改变设置。

calendar.**prmonth**(*theyear*, *themonth*, *w=0*, *l=0*)
打印由 *month()* 返回的一个月的日历。

calendar.**month**(*theyear*, *themonth*, *w=0*, *l=0*)
使用 *TextCalendar* 类的 formatmonth() 以多行字符串形式返回月份日历。

calendar.**prcal**(*year*, *w=0*, *l=0*, *c=6*, *m=3*)
打印由 *calendar()* 返回的整年的日历。

calendar.**calendar**(*year*, *w=2*, *l=1*, *c=6*, *m=3*)
使用 *TextCalendar* 类的 formatyear() 返回整年的 3 列的日历以多行字符串的形式。

calendar.**timegm**(*tuple*)
一个不相关但很好用的函数，它接受一个时间元组例如 *time* 模块中的 *gmtime()* 函数的返回并返回相应的 Unix 时间戳值，假定 1970 年开始计数，POSIX 编码。实际上，*time.gmtime()* 和 *timegm()* 是彼此相反的。

calendar 模块包含以下数据属性：

calendar.**day_name**
在当前语言环境下表示星期几的数组。

calendar.**day_abbr**
在当前语言环境下表示星期几缩写的数组。

calendar.**month_name**
在当前语言环境下表示一年中月份的数组。这遵循一月的月号为 1 的通常惯例，所以它的长度为 13 且 month_name[0] 是空字符串。

calendar.**month_abbr**
在当前语言环境下表示月份简写的数组。这遵循一月的月号为 1 的通常惯例，所以它的长度为 13 且 month_abbr[0] 是空字符串。

也参考:

模块*datetime* 为日期和时间提供与*time* 模块相似功能的面向对象接口。

模块*time* 底层时间相关函数。

8.4 collections --- 容器資料型態

原始碼： Lib/collections/__init__.py

這個模組實作了一些特🄵的容器資料型態，用來替代 Python 一般🄵建的容器，例如*dict* 、*list* 、*set* 和*tuple* 。

namedtuple()	用來建立一個欄位擁有名字的 tuple 子類🄵的函數
deque	一個類似 list 的容器，可以快速的在頭尾加入元素與取出元素。
ChainMap	一個像是 dict 的類🄵，用來🄵多個 mapping 建立單一的 view 。
Counter	dict 的子類🄵，用來計算可 hash 物件的數量。
OrderedDict	dict 的子類🄵，會記🄵物件被加入的順序。
defaultdict	dict 的子類🄵，在值不存在 dict 當中時會呼叫一個🄵生函式。
UserDict	封装了字典对象，简化了字典子类化
UserList	封装了列表对象，简化了列表子类化
UserString	封装了字符串对象，简化了字符串子类化

Deprecated since version 3.3, will be removed in version 3.10: 已将*容器抽象基类* 移至*collections.abc* 模块。为了保持向下兼容性，它们在 Python 3.9 版的这个模块中仍然存在。

8.4.1 ChainMap objects

3.3 版新加入.

一个*ChainMap* 类是为了将多个映射快速的链接到一起，这样它们就可以作为一个单元处理。它通常比创建一个新字典和多次调用*update()* 要快很多。

这个类可以用于模拟嵌套作用域，并且在模版化的时候比较有用。

class collections.**ChainMap**(**maps*)
一个*ChainMap* 将多个字典或者其他映射组合在一起，创建一个单独的可更新的视图。如果没有 *maps* 被指定，就提供一个默认的空字典，这样一个新链至少有一个映射。

底层映射被存储在一个列表中。这个列表是公开的，可以通过 *maps* 属性存取和更新。没有其他的状态。

搜索查询底层映射，直到一个键被找到。不同的是，写，更新和删除只操作第一个映射。

一个*ChainMap* 通过引用合并底层映射。所以，如果一个底层映射更新了，这些更改会反映到*ChainMap* 。

支持所有常用字典方法。另外还有一个 *maps* 属性 (attribute)，一个创建子上下文的方法 (method)，一个存取它们首个映射的属性 (property):

maps
一个可以更新的映射列表。这个列表是按照第一次搜索到最后一次搜索的顺序组织的。它是仅有的存储状态，可以被修改。列表最少包含一个映射。

new_child(*m=None*)
返回一个新的*ChainMap*类，包含了一个新映射 (map)，后面跟随当前实例的全部映射 (map)。如果 m 被指定，它就成为不同新的实例，就是在所有映射前加上 m，如果没有指定，就加上一个空字典，这样的话一个 d.new_child() 调用等价于 ChainMap({}, *d.maps) 。这个方法用于创建子上下文，不改变任何父映射的值。

3.4 版更變: 添加了 m 可选参数。

parents
属性返回一个新的*ChainMap*包含所有的当前实例的映射，除了第一个。这样可以在搜索的时候跳过第一个映射。使用的场景类似在*nested scopes*嵌套作用域中使用 nonlocal 关键词。用例也可以类比内建函数*super()*。一个 d.parents 的引用等价于 ChainMap(*d.maps[1:]) 。

注意，一个*ChainMap()*的迭代顺序是通过从后往前扫描所有映射来确定的:

```
>>> baseline = {'music': 'bach', 'art': 'rembrandt'}
>>> adjustments = {'art': 'van gogh', 'opera': 'carmen'}
>>> list(ChainMap(adjustments, baseline))
['music', 'art', 'opera']
```

这给出了与*dict.update()*调用序列相同的顺序，从最后一个映射开始:

```
>>> combined = baseline.copy()
>>> combined.update(adjustments)
>>> list(combined)
['music', 'art', 'opera']
```

3.9 版更變: 增加了对 | 和 |= 运算符的支持，相关说明见 **PEP 584**。

也参考:

- MultiContext class 在 Enthought CodeTools package 有支持写映射的选项。
- Django 中用于模板的 Context class 是只读的映射链。它还具有上下文推送和弹出特性，类似于*new_child()*方法和*parents*特征属性。
- Nested Contexts recipe 提供了是否对第一个映射或其他映射进行写和其他修改的选项。
- 一个 极简的只读版 Chainmap.

ChainMap 例子和方法

这一节提供了多个使用链映射的案例。

模拟 Python 内部 lookup 链的例子

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

让用户指定的命令行参数优先于环境变量，优先于默认值的例子

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}
```

(下页继续)

(繼續上一頁)

```
parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items() if v is not None}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

用*ChainMap*类模拟嵌套上下文的例子

```
c = ChainMap()          # Create root context
d = c.new_child()       # Create nested child context
e = c.new_child()       # Child of c, independent from d
e.maps[0]               # Current context dictionary -- like Python's locals()
e.maps[-1]              # Root context -- like Python's globals()
e.parents               # Enclosing context chain -- like Python's nonlocals

d['x'] = 1              # Set value in current context
d['x']                  # Get first key in the chain of contexts
del d['x']              # Delete from current context
list(d)                 # All nested values
k in d                  # Check all nested values
len(d)                  # Number of nested values
d.items()               # All nested items
dict(d)                 # Flatten into a regular dictionary
```

*ChainMap*类只更新链中的第一个映射，但 lookup 会搜索整个链。然而，如果需要深度写和删除，也可以很容易的通过定义一个子类来实现它

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'         # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
>>> d                            # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

8.4.2 Counter 物件

提供一個計數工具支援方便且快速的對應，舉例：

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

class collections.**Counter**([*iterable-or-mapping*])

一个*Counter*是一个*dict*的子类，用于计数可哈希对象。它是一个集合，元素像字典键 (key) 一样存储，它们的计数存储为值。计数可以是任何整数值，包括 0 和负数。*Counter*类有点像其他语言中的 bags 或 multisets。

被計數的元素來自一個 *iterable* 或是被其他的 *mapping* (or counter) 初始化。

```
>>> c = Counter()                           # a new, empty counter
>>> c = Counter('gallahad')                 # a new counter from an iterable
>>> c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
>>> c = Counter(cats=4, dogs=8)             # a new counter from keyword args
```

Counter 物件擁有一個字典的使用介面，除了遇到 Counter 中𠀀有的值時會回傳計數 0 取代*KeyError*這點不同。

```
>>> c = Counter(['eggs', 'ham'])
>>> c['bacon']                              # count of a missing element is zero
0
```

將一個值的計數設𠀀 0 𠀀不會真的從 counter 中𠀀除這個元素，使用 `del` 來𠀀除元素。

```
>>> c['sausage'] = 0                        # counter entry with a zero count
>>> del c['sausage']                        # del actually removes the entry
```

3.1 版新加入.

3.7 版更變: 作为*dict*的子类，*Counter*继承了记住插入顺序的功能。*Counter* 对象进行数学运算时同样会保持顺序。结果会先按每个元素在运算符左边的出现时间排序，然后再按其在运算符右边的出现时间排序。

Counter objects support additional methods beyond those available for all dictionaries:

elements()

返回一个迭代器，其中每个元素将重复出现计数值所指定次。元素会按首次出现的顺序返回。如果一个元素的计数值小于一，*elements()*将会忽略它。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

most_common([*n*])
返回一个列表，其中包含 *n* 个最常见的元素及出现次数，按常见程度由高到低排序。如果 *n* 被省略或为 None，*most_common()* 将返回计数器中的 所有元素。计数值相等的元素按首次出现的顺序排序：

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

subtract([*iterable-or-mapping*])
从 迭代对象或 映射对象减去元素。像*dict.update()* 但是是减去，而不是替换。输入和输出都可以是 0 或者负数。

```
>>> c = Counter(a=4, b=2, c=0, d=-2)
>>> d = Counter(a=1, b=2, c=3, d=4)
>>> c.subtract(d)
>>> c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

3.2 版新加入.

通常字典方法都可用于*Counter* 对象，除了有两个方法工作方式与字典并不相同。

fromkeys(*iterable*)
这个类方法没有在*Counter* 中实现。

update([*iterable-or-mapping*])
从 迭代对象计数元素或者从另一个 映射对象 (或计数器) 添加。像*dict.update()* 但是是加上，而不是替换。另外，迭代对象应该是序列元素，而不是一个 (key, value) 对。

Counter 对象的常用案例

```
sum(c.values())                 # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
+c                              # remove zero and negative counts
```

提供了几个数学操作，可以结合*Counter* 对象，以生产 multisets (计数器中大于 0 的元素)。加和减，结合计数器，通过加上或者减去元素的相应计数。交集和并集返回相应计数的最小或最大值。每种操作都可以接受带符号的计数，但是输出会忽略掉结果为零或者小于零的计数。

```
>>> c = Counter(a=3, b=1)
>>> d = Counter(a=1, b=2)
>>> c + d                       # add two counters together:  c[x] + d[x]
Counter({'a': 4, 'b': 3})
>>> c - d                       # subtract (keeping only positive counts)
Counter({'a': 2})
>>> c & d                       # intersection:  min(c[x], d[x])
Counter({'a': 1, 'b': 1})
>>> c | d                       # union:  max(c[x], d[x])
Counter({'a': 3, 'b': 2})
```

单目加和减（一元操作符）意思是从空计数器加或者减去。

```
>>> c = Counter(a=2, b=-4)
>>> +c
Counter({'a': 2})
>>> -c
Counter({'b': 4})
```

3.3 版新加入: 添加了对一元加，一元减和位置集合操作的支持。

備𝔽: 计数器主要是为了表达运行的正的计数而设计；但是，小心不要预先排除负数或者其他类型。为了帮助这些用例，这一节记录了最小范围和类型限制。

- *Counter* 类是一个字典的子类，不限制键和值。值用于表示计数，但你实际上 可以存储任何其他值。
- *most_common()* 方法在值需要排序的时候用。
- 原地操作比如 `c[key] += 1`，值类型只需要支持加和减。所以分数，小数，和十进制都可以用，负值也可以支持。这两个方法*update()* 和*subtract()* 的输入和输出也一样支持负数和 0。
- Multiset 多集合方法只为正值的使用情况设计。输入可以是负数或者 0，但只输出计数为正的值。没有类型限制，但值类型需要支持加，减和比较操作。
- *elements()* 方法要求正整数计数。忽略 0 和负数计数。

也参考:

- Bag class 在 Smalltalk。
- Wikipedia 链接 Multisets.
- C++ multisets 教程和例子。
- 数学操作和多集合用例，参考 *Knuth, Donald. The Art of Computer Programming Volume II, Section 4.6.3, Exercise 19* 。
- 在给定数量和集合元素枚举所有不同的多集合，参考*itertools.combinations_with_replacement()*

```
map(Counter, combinations_with_replacement('ABC', 2)) # --> AA AB AC BB BC CC
```

8.4.3 deque 对象

class collections.**deque**([*iterable*[, *maxlen*]])

返回一个新的双向队列对象，从左到右初始化 (用方法*append()*)，从 *iterable* （迭代对象) 数据创建。如果 *iterable* 没有指定，新队列为空。

Deque 队列是由栈或者 queue 队列生成的（发音是 “deck”,” double-ended queue” 的简称)。Deque 支持线程安全，内存高效添加 (append) 和弹出 (pop)，从两端都可以，两个方向的大概开销都是 O(1) 复杂度。

虽然*list* 对象也支持类似操作，不过这里优化了定长操作和 `pop(0)` 和 `insert(0, v)` 的开销。它们引起 O(n) 内存移动的操作，改变底层数据表达的大小和位置。

如果 *maxlen* 没有指定或者是 `None`，deques 可以增长到任意长度。否则，deque 就限定到指定最大长度。一旦限定长度的 deque 满了，当新项加入时，同样数量的项就从另一端弹出。限定长度 deque 提供类似 Unix filter `tail` 的功能。它们同样可以用与追踪最近的交换和其他数据池活动。

双向队列 (deque) 对象支持以下方法：

append(*x*)

添加 *x* 到右端。

appendleft(*x*)
添加 *x* 到左端。

clear()
移除所有元素，使其长度为 0.

copy()
创建一份浅拷贝。

3.5 版新加入.

count(*x*)
计算 deque 中元素等于 *x* 的个数。

3.2 版新加入.

extend(*iterable*)
扩展 deque 的右侧，通过添加 iterable 参数中的元素。

extendleft(*iterable*)
扩展 deque 的左侧，通过添加 iterable 参数中的元素。注意，左添加时，在结果中 iterable 参数中的顺序将被反过来添加。

index(*x*[, *start*[, *stop*]])
返回 *x* 在 deque 中的位置（在索引 *start* 之后，索引 *stop* 之前）。返回第一个匹配项，如果未找到则引发*ValueError*。

3.5 版新加入.

insert(*i*, *x*)
在位置 *i* 插入 *x* 。

如果插入会导致一个限长 deque 超出长度 *maxlen* 的话，就引发一个*IndexError*。

3.5 版新加入.

pop()
移去并且返回一个元素，deque 最右侧的那一个。如果没有元素的话，就引发一个*IndexError*。

popleft()
移去并且返回一个元素，deque 最左侧的那一个。如果没有元素的话，就引发*IndexError*。

remove(*value*)
移除找到的第一个 *value*。如果没有的话就引发*ValueError*。

reverse()
将 deque 逆序排列。返回 `None` 。

3.2 版新加入.

rotate(*n=1*)
向右循环移动 *n* 步。如果 *n* 是负数，就向左循环。

如果 deque 不是空的，向右循环移动一步就等价于 `d.appendleft(d.pop())` ，向左循环一步就等价于 `d.append(d.popleft())` 。

Deque 对象同样提供了一个只读属性:

maxlen
Deque 的最大尺寸，如果没有限定的话就是 `None` 。

3.1 版新加入.

除了以上操作，deque 还支持迭代、封存、`len(d)`、`reversed(d)`、`copy.copy(d)`、`copy.deepcopy(d)`、成员检测运算符 `in` 以及下标引用例如通过 `d[0]` 访问首个元素等。索引访问在两端的复杂度均为 O(1) 但在中间则会低至 O(n)。如需快速随机访问，请改用列表。

Deque 从版本 3.5 开始支持 `__add__()`，`__mul__()`，和 `__imul__()` 。

示例:

```
>>> from collections import deque
>>> d = deque('ghi')                 # make a new deque with three items
>>> for elem in d:                   # iterate over the deque's elements
...     print(elem.upper())
G
H
I

>>> d.append('j')                    # add a new entry to the right side
>>> d.appendleft('f')                # add a new entry to the left side
>>> d                                # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                          # return and remove the rightmost item
'j'
>>> d.popleft()                      # return and remove the leftmost item
'f'
>>> list(d)                          # list the contents of the deque
['g', 'h', 'i']
>>> d[0]                             # peek at leftmost item
'g'
>>> d[-1]                            # peek at rightmost item
'i'

>>> list(reversed(d))                # list the contents of a deque in reverse
['i', 'h', 'g']
>>> 'h' in d                         # search the deque
True
>>> d.extend('jkl')                  # add multiple elements at once
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])
>>> d.rotate(1)                      # right rotation
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)                     # left rotation
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))               # make a new deque in reverse order
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                        # empty the deque
>>> d.pop()                          # cannot pop from an empty deque
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in -toplevel-
        d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')              # extendleft() reverses the input order
>>> d
deque(['c', 'b', 'a'])
```

deque 用法

这一节展示了 deque 的多种用法。

限长 deque 提供了类似 Unix `tail` 过滤功能

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

另一个用法是维护一个近期添加元素的序列，通过从右边添加和从左边弹出

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # http://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

一个 轮询调度器 可以通过在*deque* 中放入迭代器来实现。值从当前迭代器的位置 0 被取出并暂存 (yield)。如果这个迭代器消耗完毕，就用*popleft()* 将其从对列中移去；否则，就通过*rotate()* 将它移到队列的末尾

```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # Remove an exhausted iterator.
            iterators.popleft()
```

rotate() 方法提供了一种方式来实现*deque* 切片和删除。例如，一个纯的 Python `del d[n]` 实现依赖于 `rotate()` 来定位要弹出的元素

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

要实现*deque* 切片，使用一个类似的方法，应用*rotate()* 将目标元素放到左边。通过*popleft()* 移去老的条目（entries），通过*extend()* 添加新的条目，然后反向 rotate。这个方法可以最小代价实现命令式的栈操作，诸如 `dup`, `drop`, `swap`, `over`, `pick`, `rot`, 和 `roll` 。

8.4.4 defaultdict 对象

class collections.**defaultdict**(*default_factory=None*, /[, ...])

返回一个新的类似字典的对象。*defaultdict* 是内置*dict* 类的子类。它重载了一个方法并添加了一个可写的实例变量。其余的功能与*dict* 类相同因而不在此文档中写明。

本对象包含一个名为*default_factory* 的属性，构造时，第一个参数用于为该属性提供初始值，默认为 None。所有其他参数（包括关键字参数）都相当于传递给*dict* 的构造函数。

defaultdict 对象除了支持标准*dict* 的操作，还支持以下方法作为扩展：

__missing__(*key*)

如果*default_factory* 属性为 None，则调用本方法会抛出*KeyError* 异常，附带参数 *key*。

如果*default_factory* 不为 None，则它会被（不带参数地）调用来为 *key* 提供一个默认值，这个值和 *key* 作为一对键值对被插入到字典中，并作为本方法的返回值返回。

如果调用*default_factory* 时抛出了异常，这个异常会原封不动地向外层传递。

在无法找到所需键值时，本方法会被*dict* 中的 __getitem__() 方法调用。无论本方法返回了值还是抛出了异常，都会被 __getitem__() 传递。

注意，*__missing__()* 不会被 __getitem__() 以外的其他方法调用。意味着 get() 会像正常的 dict 那样返回 None，而不是使用*default_factory*。

defaultdict 对象支持以下实例变量：

default_factory

本属性由*__missing__()* 方法来调用。如果构造对象时提供了第一个参数，则本属性会被初始化成那个参数，如果未提供第一个参数，则本属性为 None。

3.9 版更變: 增加了合并 (|) 与更新 (|=) 运算符，相关说明见 **PEP 584**。

defaultdict 例子

使用*list* 作为*default_factory*，很轻松地将（键-值对组成的）序列转换为（键-列表组成的）字典：

```
>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
>>> d = defaultdict(list)
>>> for k, v in s:
...     d[k].append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

当每个键第一次遇见时，它还没有在字典里面，所以自动创建该条目，即调用*default_factory* 方法，返回一个空的*list*。list.append() 操作添加值到这个新的列表里。当再次存取该键时，就正常操作，list.append() 添加另一个值到列表中。这个计数比它的等价方法*dict.setdefault()* 要快速和简单：

```
>>> d = {}
>>> for k, v in s:
...     d.setdefault(k, []).append(v)
...
>>> sorted(d.items())
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

设置*default_factory* 为*int*，使*defaultdict* 用于计数（类似其他语言中的 bag 或 multiset）：

```
>>> s = 'mississippi'
>>> d = defaultdict(int)
>>> for k in s:
...     d[k] += 1
...
>>> sorted(d.items())
[('i', 4), ('m', 1), ('p', 2), ('s', 4)]
```

当一个字母首次遇到时，它会查询失败，则*default_factory* 会调用*int()* 来提供一个整数 0 作为默认值。后续的自增操作建立起对每个字母的计数。

函数*int()* 总是返回 0，这是常数函数的特殊情况。一个更快和灵活的方法是使用 lambda 函数，可以提供任何常量值（不只是 0）：

```
>>> def constant_factory(value):
...     return lambda: value
>>> d = defaultdict(constant_factory('<missing>'))
>>> d.update(name='John', action='ran')
>>> '%(name)s %(action)s to %(object)s' % d
'John ran to <missing>'
```

设置*default_factory* 为*set* 使*defaultdict* 用于构建 set 集合：

```
>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
>>> d = defaultdict(set)
>>> for k, v in s:
...     d[k].add(v)
...
>>> sorted(d.items())
[('blue', {2, 4}), ('red', {1, 3})]
```

8.4.5 namedtuple() 命名元组的工厂函数

命名元组赋予每个位置一个含义，提供可读性和自文档性。它们可以用于任何普通元组，并添加了通过名字获取值的能力，通过索引值也是可以的。

collections.**namedtuple**(*typename*, *field_names*, *, *rename=False*, *defaults=None*, *module=None*)

返回一个新的元组子类，名为 *typename* 。这个新的子类用于创建类元组的对象，可以通过字段名来获取属性值，同样也可以通过索引和迭代获取值。子类实例同样有文档字符串（类名和字段名）另外一个有用的 `__repr__()` 方法，以 `name=value` 格式列明了元组内容。

field_names 是一个像 `['x', 'y']` 一样的字符串序列。另外 *field_names* 可以是一个纯字符串，用空白或逗号分隔开元素名，比如 `'x y'` 或者 `'x, y'` 。

任何有效的 Python 标识符都可以作为字段名，除了下划线开头的那些。有效标识符由字母，数字，下划线组成，但首字母不能是数字或下划线，另外不能是关键词*keyword* 比如 *class*, *for*, *return*, *global*, *pass*, 或 *raise* 。

如果 *rename* 为真，无效字段名会自动转换成位置名。比如 `['abc', 'def', 'ghi', 'abc']` 转换成 `['abc', '_1', 'ghi', '_3']` ，消除关键词 `def` 和重复字段名 `abc` 。

defaults 可以为 `None` 或者是一个默认值的*iterable* 。如果一个默认值域必须跟其他没有默认值的域在一起出现，*defaults* 就应用到最右边的参数。比如如果域名 `['x', 'y', 'z']` 和默认值 `(1, 2)` ，那么 `x` 就必须指定一个参数值，`y` 默认值 `1` ，`z` 默认值 `2` 。

如果 *module* 值有定义，命名元组的 `__module__` 属性值就被设置。

命名元组实例没有字典，所以它们要更轻量，并且占用更小内存。

要支持封存操作，应当将命名元组类赋值给一个匹配 *typename* 的变量。

3.1 版更變: 添加了对 *rename* 的支持。

3.6 版更變: *verbose* 和 *rename* 参数成为仅限关键字参数.

3.6 版更變: 添加了 *module* 参数。

3.7 版更變: 移除了 *verbose* 形参和 `_source` 属性。

3.7 版更變: 添加了 *defaults* 参数和 `_field_defaults` 属性。

```
>>> # Basic example
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(11, y=22)     # instantiate with positional or keyword arguments
>>> p[0] + p[1]             # indexable like the plain tuple (11, 22)
33
>>> x, y = p                # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y               # fields also accessible by name
33
>>> p                       # readable __repr__ with a name=value style
Point(x=11, y=22)
```

命名元组尤其有用于赋值*csv* *sqlite3* 模块返回的元组

```
EmployeeRecord = namedtuple('EmployeeRecord', 'name, age, title, department, paygrade
↪')

import csv
for emp in map(EmployeeRecord._make, csv.reader(open("employees.csv", "rb"))):
    print(emp.name, emp.title)

import sqlite3
conn = sqlite3.connect('/companydata')
cursor = conn.cursor()
cursor.execute('SELECT name, age, title, department, paygrade FROM employees')
for emp in map(EmployeeRecord._make, cursor.fetchall()):
    print(emp.name, emp.title)
```

除了继承元组的方法，命名元组还支持三个额外的方法和两个属性。为了防止字段名冲突，方法和属性以下划线开始。

classmethod somenamedtuple.**_make**(*iterable*)

类方法从存在的序列或迭代实例创建一个新实例。

```
>>> t = [11, 22]
>>> Point._make(t)
Point(x=11, y=22)
```

somenamedtuple.**_asdict**()

返回一个新的*dict*，它将字段名称映射到它们对应的值:

```
>>> p = Point(x=11, y=22)
>>> p._asdict()
{'x': 11, 'y': 22}
```

3.1 版更變: 返回一个*OrderedDict* 而不是*dict* 。

3.8 版更變: 返回一个常规*dict* 而不是*OrderedDict*。因为自 Python 3.7 起，常规字典已经保证有序。如果需要*OrderedDict* 的额外特性，推荐的解决方案是将结果转换为需要的类型: `OrderedDict(nt._asdict())`。

somenamedtuple.**_replace**(***kwargs*)
返回一个新的命名元组实例，并将指定域替换为新的值

```
>>> p = Point(x=11, y=22)
>>> p._replace(x=33)
Point(x=33, y=22)

>>> for partnum, record in inventory.items():
...     inventory[partnum] = record._replace(price=newprices[partnum],␣
↪timestamp=time.now())
```

somenamedtuple.**_fields**
字符串元组列出了字段名。用于提醒和从现有元组创建一个新的命名元组类型。

```
>>> p._fields            # view the field names
('x', 'y')

>>> Color = namedtuple('Color', 'red green blue')
>>> Pixel = namedtuple('Pixel', Point._fields + Color._fields)
>>> Pixel(11, 22, 128, 255, 0)
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

somenamedtuple.**_field_defaults**
字典将字段名称映射到默认值。

```
>>> Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
>>> Account._field_defaults
{'balance': 0}
>>> Account('premium')
Account(type='premium', balance=0)
```

要获取这个名字域的值，使用*getattr()* 函数:

```
>>> getattr(p, 'x')
11
```

转换一个字典到命名元组，使用 ** 两星操作符 (所述如 tut-unpacking-arguments):

```
>>> d = {'x': 11, 'y': 22}
>>> Point(**d)
Point(x=11, y=22)
```

因为一个命名元组是一个正常的 Python 类，它可以很容易的通过子类更改功能。这里是如何添加一个计算域和定宽输出打印格式:

```
>>> class Point(namedtuple('Point', ['x', 'y'])):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f' % (self.x, self.y, self.
↪hypot)
```

(下页继续)

(繼續上一頁)

```
>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

上面的子类设置 __slots__ 为一个空元组。通过阻止创建实例字典保持了较低的内存开销。

子类化对于添加和存储新的名字域是无效的。应当通过 *_fields* 创建一个新的命名元组来实现它:

```
>>> Point3D = namedtuple('Point3D', Point._fields + ('z',))
```

文档字符串可以自定义，通过直接赋值给 __doc__ 属性:

```
>>> Book = namedtuple('Book', ['id', 'title', 'authors'])
>>> Book.__doc__ += ': Hardcover book in active collection'
>>> Book.id.__doc__ = '13-digit ISBN'
>>> Book.title.__doc__ = 'Title of first printing'
>>> Book.authors.__doc__ = 'List of authors sorted by last name'
```

3.5 版更變: 文档字符串属性变成可写。

也参考:

- 请参阅 *typing.NamedTuple* ，以获取为命名元组添加类型提示的方法。它还使用 class 关键字提供了一种优雅的符号:

  ```
  class Component(NamedTuple):
      part_number: int
      weight: float
      description: Optional[str] = None
  ```

- 对于以字典为底层的可变域名，参考 *types.SimpleNamespace()* 。
- *dataclasses* 模块提供了一个装饰器和一些函数，用于自动将生成的特殊方法添加到用户定义的类中。

8.4.6 OrderedDict 对象

有序词典就像常规词典一样，但有一些与排序操作相关的额外功能。由于内置的 *dict* 类获得了记住插入顺序的能力（在 Python 3.7 中保证了这种新行为），它们变得不那么重要了。

一些与 *dict* 的不同仍然存在:

- 常规的 *dict* 被设计为非常擅长映射操作。跟踪插入顺序是次要的。
- *OrderedDict* 旨在擅长重新排序操作。空间效率、迭代速度和更新操作的性能是次要的。
- 算法上，*OrderedDict* 可以比 *dict* 更好地处理频繁的重新排序操作。这使其适用于跟踪最近的访问（例如在 LRU cache 中）。
- 对于 *OrderedDict* ，相等操作检查匹配顺序。
- *OrderedDict* 类的 popitem() 方法有不同的签名。它接受一个可选参数来指定弹出哪个元素。
- *OrderedDict* 类有一个 move_to_end() 方法，可以有效地将元素移动到任一端。
- Python 3.8 之前，*dict* 缺少 __reversed__() 方法。

class collections.**OrderedDict**([*items*])

返回一个*dict* 子类的实例，它具有专门用于重新排列字典顺序的方法。

3.1 版新加入.

popitem(*last=True*)

有序字典的*popitem()* 方法移除并返回一个 (key, value) 键值对。如果 *last* 值为真，则按 LIFO 后进先出的顺序返回键值对，否则就按 FIFO (first-in, first-out) 先进先出的顺序返回键值对。

move_to_end(*key*, *last=True*)

将现有 *key* 移动到有序字典的任一端。如果 *last* 为真值（默认）则将元素移至末尾；如果 *last* 为假值则将元素移至开头。如果 *key* 不存在则会触发*KeyError*:

```
>>> d = OrderedDict.fromkeys('abcde')
>>> d.move_to_end('b')
>>> ''.join(d.keys())
'acdeb'
>>> d.move_to_end('b', last=False)
>>> ''.join(d.keys())
'bacde'
```

3.2 版新加入.

相对于通常的映射方法，有序字典还另外提供了逆序迭代的支持，通过*reversed()* 。

OrderedDict 之间的相等测试是顺序敏感的，实现为 `list(od1.items())==list(od2.items())` 。*OrderedDict* 对象和其他的*Mapping* 的相等测试，是顺序敏感的字典测试。这允许*OrderedDict* 替换为任何字典可以使用的场所。

3.5 版更變: *OrderedDict* 的项 (item)，键 (key) 和值 (value) 视图 现在支持逆序迭代，通过*reversed()* 。

3.6 版更變: **PEP 468** 赞成将关键词参数的顺序保留，通过传递给*OrderedDict* 构造器和它的 `update()` 方法。

3.9 版更變: 增加了合并 (`|`) 与更新 (`|=`) 运算符，相关说明见 **PEP 584**。

OrderedDict 例子和用法

创建记住键值 最后插入顺序的有序字典变体很简单。如果新条目覆盖现有条目，则原始插入位置将更改并移至末尾:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

一个*OrderedDict* 对于实现*functools.lru_cache()* 的变体也很有用:

```
class LRU:

    def __init__(self, func, maxsize=128):
        self.func = func
        self.maxsize = maxsize
        self.cache = OrderedDict()

    def __call__(self, *args):
        if args in self.cache:
```

(下页继续)

(繼續上一頁)

```
            value = self.cache[args]
            self.cache.move_to_end(args)
            return value
        value = self.func(*args)
        if len(self.cache) >= self.maxsize:
            self.cache.popitem(False)
        self.cache[args] = value
        return value
```

8.4.7 UserDict 对象

UserDict 类是用作字典对象的外包装。对这个类的需求已部分由直接创建*dict* 的子类的功能所替代；不过，这个类处理起来更容易，因为底层的字典可以作为属性来访问。

class collections.**UserDict**([*initialdata*])

模拟字典的类。这个实例的内容保存在一个常规字典中，它可以通过*UserDict* 实例的*data* 属性来访问。如果提供了 *initialdata*，则*data* 会用其内容来初始化；请注意对 *initialdata* 的引用将不会被保留，以允许它被用于其他目的。

UserDict 实例提供了以下属性作为扩展方法和操作的支持:

data

一个真实的字典，用于保存*UserDict* 类的内容。

8.4.8 UserList 对象

这个类封装了列表对象。它是一个有用的基础类，对于你想自定义的类似列表的类，可以继承和覆盖现有的方法，也可以添加新的方法。这样我们可以对列表添加新的行为。

对这个类的需求已部分由直接创建*list* 的子类的功能所替代；不过，这个类处理起来更容易，因为底层的列表可以作为属性来访问。

class collections.**UserList**([*list*])

模拟一个列表。这个实例的内容被保存为一个正常列表，通过*UserList* 的*data* 属性存取。实例内容被初始化为一个 *list* 的 copy，默认为 `[]` 空列表。*list* 可以是迭代对象，比如一个 Python 列表，或者一个*UserList* 对象。

UserList 提供了以下属性作为可变序列的方法和操作的扩展:

data

一个*list* 对象用于存储*UserList* 的内容。

子类化的要求: *UserList* 的子类需要提供一个构造器，可以无参数调用，或者一个参数调用。返回一个新序列的列表操作需要创建一个实现类的实例。它假定了构造器可以以一个参数进行调用，这个参数是一个序列对象，作为数据源。

如果一个分离的类不希望依照这个需求，所有的特殊方法就必须重写；请参照源代码进行修改。

8.4.9 UserString 对象

UserString 类是用作字符串对象的外包装。对这个类的需求已部分由直接创建*str* 的子类的功能所替代；不过，这个类处理起来更容易，因为底层的字符串可以作为属性来访问。

class collections.**UserString**(*seq*)

模拟一个字符串对象。这个实例对象的内容保存为一个正常字符串，通过*UserString* 的*data* 属性存取。实例内容初始化设置为 *seq* 的 copy。*seq* 参数可以是任何可通过内建*str()* 函数转换为字符串的对象。

UserString 提供了以下属性作为字符串方法和操作的额外支持：

data

一个真正的*str* 对象用来存放*UserString* 类的内容。

3.5 版更變: 新方法 __getnewargs__, __rmod__, casefold, format_map, isprintable, 和 maketrans。

8.5 collections.abc --- 容器的抽象基类

3.3 版新加入: 该模块曾是*collections* 模块的组成部分。

源代码： Lib/_collections_abc.py

该模块定义了一些抽象基类，它们可用于判断一个具体类是否具有某一特定的接口；例如，这个类是否可哈希，或其是否为映射类。

3.9 版新加入: These abstract classes now support []. See *GenericAlias* 类型 and **PEP 585**.

8.5.1 容器抽象基类

这个容器模块提供了以下*ABCs*:

抽象基类	继承自	抽象方法	Mixin 方法
Container		`__contains__`	
Hashable		`__hash__`	
Iterable		`__iter__`	
Iterator	*Iterable*	`__next__`	`__iter__`
Reversible	*Iterable*	`__reversed__`	
Generator	*Iterator*	`send`, `throw`	`close`, `__iter__`, `__next__`
Sized		`__len__`	
Callable		`__call__`	
Collection	*Sized*, *Iterable*, *Container*	`__contains__`, `__iter__`, `__len__`	
Sequence	*Reversible*, *Collection*	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `index`, and `count`
MutableSequence	*Sequence*	`__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`	继承自*Sequence* 的方法，以及 `append`, `reverse`, `extend`, `pop`, `remove`，和 `__iadd__`
ByteString	*Sequence*	`__getitem__`, `__len__`	继承自*Sequence* 的方法
Set	*Collection*	`__contains__`, `__iter__`, `__len__`	`__le__`, `__lt__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`, `__and__`, `__or__`, `__sub__`, `__xor__`, and `isdisjoint`
MutableSet	*Set*	`__contains__`, `__iter__`, `__len__`, `add`, `discard`	继承自*Set* 的方法以及 `clear`, `pop`, `remove`, `__ior__`, `__iand__`, `__ixor__`，和 `__isub__`
Mapping	*Collection*	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `keys`, `items`, `values`, `get`, `__eq__`, and `__ne__`
MutableMapping	*Mapping*	`__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__`	继承自*Mapping* 的方法以及 `pop`, `popitem`, `clear`, `update`，和 `setdefault`
MappingView	*Sized*		`__len__`
ItemsView	*MappingView*, *Set*		`__contains__`, `__iter__`
KeysView	*MappingView*, *Set*		`__contains__`, `__iter__`
ValuesView	*MappingView*, *Collection*		`__contains__`, `__iter__`
Awaitable		`__await__`	
Coroutine	*Awaitable*	`send`, `throw`	`close`
AsyncIterable		`__aiter__`	
AsyncIterator	*AsyncIterable*	`__anext__`	`__aiter__`
AsyncGenerator	*AsyncIterator*	`asend`, `athrow`	`aclose`, `__aiter__`, `__anext__`

class collections.abc.**Container**
提供了 `__contains__()` 方法的抽象基类。

class collections.abc.**Hashable**

提供了 __hash__() 方法的抽象基类。

class collections.abc.**Sized**
提供了 __len__() 方法的抽象基类。

class collections.abc.**Callable**
提供了 __call__() 方法的抽象基类。

class collections.abc.**Iterable**
提供了 __iter__() 方法的抽象基类。

使用 isinstance(obj, Iterable) 可以检测一个类是否已经注册到了*Iterable* 或者实现了 __iter__() 函数，但是无法检测这个类是否能够使用 __getitem__() 方法进行迭代。检测一个对象是否是*iterable* 的唯一可信赖的方法是调用 iter(obj)。

class collections.abc.**Collection**
集合了 Sized 和 Iterable 类的抽象基类。

3.6 版新加入.

class collections.abc.**Iterator**
提供了*__iter__()* 和*__next__()* 方法的抽象基类。参见*iterator* 的定义。

class collections.abc.**Reversible**
为可迭代类提供了 __reversed__() 方法的抽象基类。

3.6 版新加入.

class collections.abc.**Generator**
生成器类，实现了 **PEP 342** 中定义的协议，继承并扩展了迭代器，提供了 send(), throw() 和 close() 方法。参见*generator* 的定义。

3.5 版新加入.

class collections.abc.**Sequence**
class collections.abc.**MutableSequence**
class collections.abc.**ByteString**
只读且可变的序列*sequences* 的抽象基类。

实现笔记：一些混入（Maxin）方法比如 __iter__(), __reversed__() 和 index() 会重复调用底层的 __getitem__() 方法。因此，如果实现的 __getitem__() 是常数级访问速度，那么相应的混入方法会有一个线性的表现；然而，如果底层方法是线性实现（例如链表），那么混入方法将会是平方级的表现，这也许就需要被重构了。

3.5 版更變: index() 方法支持 *stop* 和 *start* 参数。

class collections.abc.**Set**
class collections.abc.**MutableSet**
只读且可变的集合的抽象基类。

class collections.abc.**Mapping**
class collections.abc.**MutableMapping**
只读且可变的映射*mappings* 的抽象基类。

class collections.abc.**MappingView**
class collections.abc.**ItemsView**
class collections.abc.**KeysView**
class collections.abc.**ValuesView**
映射及其键和值的视图*views* 的抽象基类。

class collections.abc.**Awaitable**
为可等待对象*awaitable* 提供的类，可以被用于 await 表达式中。习惯上必须实现 __await__() 方法。

协程 对象和*Coroutine* ABC 的实例都是这个 ABC 的实例。

備𝔽: 在 CPython 里，基于生成器的协程（使用*types.coroutine()* 或*asyncio.coroutine()* 包装的生成器）都是 可等待对象，即使他们不含有 `__await__()` 方法。使用 `isinstance(gencoro, Awaitable)` 来检测他们会返回 `False`。要使用*inspect.isawaitable()* 来检测他们。

3.5 版新加入.

class collections.abc.**Coroutine**
用于协程兼容类的抽象基类。实现了如下定义在 coroutine-objects: 里的方法：`send()`，`throw()` 和 `close()`。通常的实现里还需要实现 `__await__()` 方法。所有的*Coroutine* 实例都必须是*Awaitable* 实例。参见*coroutine* 的定义。

備𝔽: 在 CPython 里，基于生成器的协程（使用*types.coroutine()* 或*asyncio.coroutine()* 包装的生成器）都是 可等待对象，即使他们不含有 `__await__()` 方法。使用 `isinstance(gencoro, Coroutine)` 来检测他们会返回 `False`。要使用*inspect.isawaitable()* 来检测他们。

3.5 版新加入.

class collections.abc.**AsyncIterable**
提供了 `__aiter__` 方法的抽象基类。参见*asynchronous iterable* 的定义。

3.5 版新加入.

class collections.abc.**AsyncIterator**
提供了 `__aiter__` 和 `__anext__` 方法的抽象基类。参见*asynchronous iterator* 的定义。

3.5 版新加入.

class collections.abc.**AsyncGenerator**
为异步生成器类提供的抽象基类，这些类实现了定义在 **PEP 525** 和 **PEP 492** 里的协议。

3.6 版新加入.

这些抽象基类让我们可以确定类和示例拥有某些特定的函数，例如：

```
size = None
if isinstance(myvar, collections.abc.Sized):
    size = len(myvar)
```

有些抽象基类也可以用作混入类（mixin），这可以更容易地开发支持容器 API 的类。例如，要写一个支持完整*Set* API 的类，只需要提供下面这三个方法：`__contains__()`, `__iter__()` 和 `__len__()`。抽象基类会补充上其余的方法，比如 `__and__()` 和 `isdisjoint()`:

```
class ListBasedSet(collections.abc.Set):
    ''' Alternate set implementation favoring space over speed
        and not requiring the set elements to be hashable. '''
    def __init__(self, iterable):
        self.elements = lst = []
        for value in iterable:
            if value not in lst:
                lst.append(value)

    def __iter__(self):
        return iter(self.elements)

    def __contains__(self, value):
```

(下页继续)

(繼續上一頁)

```
        return value in self.elements

    def __len__(self):
        return len(self.elements)

s1 = ListBasedSet('abcdef')
s2 = ListBasedSet('defghi')
overlap = s1 & s2            # The __and__() method is supported automatically
```

当把*Set* 和*MutableSet* 用作混入类时需注意：

(1) 由于某些集合操作会创建新集合，默认的混入方法需要一种从可迭代对象里创建新实例的方式。假定类构造器具有 `ClassName(iterable)` 形式的签名。这样它将执行一个名为 `_from_iterable()` 的内部类方法，该方法会调用 `cls(iterable)` 来产生一个新集合。如果*Set* 混入类在具有不同构造器签名的类中被使用，你将需要通过类方法或常规方法来重载 `_from_iterable()`，以便基于可迭代对象参数来构造新的实例。

(2) 重载比较符时时（想必是为了速度，因为其语义都是固定的），只需要重定义 `__le__()` 和 `__ge__()` 函数，然后其他的操作会自动跟进。

(3) 混入集合类*Set* 提供了一个 `_hash()` 方法为集合计算哈希值，然而，`__hash__()` 函数却没有被定义，因为并不是所有集合都是可哈希并且不可变的。为了使用混入类为集合添加哈希能力，可以同时继承*Set()* 和*Hashable()* 类，然后定义 `__hash__ = Set._hash`。

也参考:

- OrderedSet recipe 是基于*MutableSet* 构建的一个示例。
- 对于抽象基类，参见*abc* 模块和 **PEP 3119**。

8.6 heapq --- 堆積𝔽列 (heap queue) 演算法

原始碼： Lib/heapq.py

這個模組實作了堆積𝔽列 (heap queue) 演算法，亦被稱𝔽優先𝔽列 (priority queue) 演算法。

Heap（堆積）是一顆二元樹，樹上所有父節點的值都小於等於他的子節點的值。使用陣列實作，對於所有從 0 開始的 *k* 都滿足 `heap[k] <= heap[2*k+1]` 和 `heap[k] <= heap[2*k+2]`。𝔽了比較節點的值，不存在的元素被視𝔽無限大。heap 存在一個有趣的性質：樹上最小的元素永遠會在根節點 `heap[0]` 上。

下方的 API 跟一般教科書的 heap queue 演算法有兩個方面不同：第一，我們的索引從 0 開始計算，這會父節點與子節點之間的關𝔽𝔽生很微小的差𝔽，但更符合 Python 從 0 開始索引的設計。第二，我們的 pop 方法會回傳最小的元素而不是最大的元素（在教科書中被稱作"min heap"，而"max heap" 因𝔽他很適合做原地排序，所以更常出現在教科書中）。

這兩個特性使得把 heap 當作一個標准的 Python list 檢視時不會出現意外：`heap[0]` 是最小的物件，`heap.sort()` 能保持 heap 的性質不變！

建立一個 heap 可以使用 list 初始化𝔽 `[]`，或者使用函式*heapify()* 將一個已經有元素的 list 轉成一個 heap。

此模組提供下面的函式

`heapq.`**`heappush`**(*heap*, *item*)
把 *item* 放進 *heap*，𝔽保持 heap 性質不變。

heapq.**heappop**(*heap*)
從 *heap* 取出🄴回傳最小的元素，同時保持 heap 性質不變。如果 heap 是空的會🄴生 *IndexError* 錯誤。只存取最小元素但不取出可以使用 `heap[0]` 。

heapq.**heappushpop**(*heap*, *item*)
將 *item* 放入 heap ，接著從 *heap* 取出🄴回傳最小的元素。這個組合函式比呼叫 *heappush()* 之後呼叫 *heappop()* 更有效率。

heapq.**heapify**(*x*)
在🄴性時間🄴將 list *x* 轉🄴 heap，且過程不會申請額外記憶體。

heapq.**heapreplace**(*heap*, *item*)
從 *heap* 取出🄴回傳最小的元素，接著將新的 *item* 放進 heap。heap 的大小不會改變。如果 heap 是空的會🄴生 *IndexError* 錯誤。

這個一次完成的操作會比呼叫 *heappop()* 之後呼叫 *heappush()* 更有效率，🄴在維護 heap 的大小不變時更🄴適當，取出/放入的組合函式一定會從 heap 回傳一個元素🄴用 *item* 取代他。

函式的回傳值可能會大於被加入的 *item* 。如果這不是你期望發生的，可以考慮使用 *heappushpop()* 替代，他會回傳 heap 的最小值和 *item* 兩個當中比較小的那個，🄴將大的留在 heap 🄴。

這個模組也提供三個利用 heap 實作的一般用途函式

heapq.**merge**(**iterables*, *key=None*, *reverse=False*)
合🄴多個已排序的輸入🄴🄴生單一且已排序的輸出（舉例：合🄴來自多個 log 檔中有時間戳記的項目）。回傳一個 *iterator* 包含已經排序的值。

和 `sorted(itertools.chain(*iterables))` 類似但回傳值是一個 iterable ，不會一次把所有資料都放進記憶體中，🄴且假設每一個輸入都已經（由小到大）排序過了。

有兩個選用參數，指定時必須被當作關鍵字參數指定。

key 參數指定了一個 *key function* 引數，用來從每一個輸入的元素中🄴定一個比較的依據。預設的值是 `None` （直接比較元素）。

reverse 为一个布尔值。如果设为 `True`，则输入元素将按比较结果逆序进行合并。要达成与 `sorted(itertools.chain(*iterables), reverse=True)` 类似的行为，所有可迭代对象必须是已从大到小排序的。

3.5 版更變: 加入選用參數 *key* 和 *reverse* 。

heapq.**nlargest**(*n*, *iterable*, *key=None*)
从 *iterable* 所定义的数据集中返回前 *n* 个最大元素组成的列表。如果提供了 *key* 则其应指定一个单参数的函数，用于从 *iterable* 的每个元素中提取比较键 (例如 `key=str.lower`)。等价于: `sorted(iterable, key=key, reverse=True)[:n]`。

heapq.**nsmallest**(*n*, *iterable*, *key=None*)
从 *iterable* 所定义的数据集中返回前 *n* 个最小元素组成的列表。如果提供了 *key* 则其应指定一个单参数的函数，用于从 *iterable* 的每个元素中提取比较键 (例如 `key=str.lower`)。等价于: `sorted(iterable, key=key)[:n]`。

後兩個函式在 *n* 值比較小時有最好的表現。對於較大的 *n* 值，只用 *sorted()* 函式會更有效率。同樣地，當 `n==1` 時，使用🄴建函式 *min()* 和 *max()* 會有更好的效率。如果需要重🄴使用這些函式，可以考慮將 iterable 轉成真正的 heap 。

8.6.1 基礎範例

堆排序 可以通过将所有值推入堆中然后每次弹出一个最小值项来实现。

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

雖然類似 `sorted(iterable)` ，但跟*sorted()* 不同的是，這個實作不是 stable 的排序。

Heap 中的元素可以是 tuple 。這有利於將要比較的值（例如一個 task 的優先度）和主要資料放在一起排序。

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

8.6.2 優先🄵列 (Priority Queue) 實作細節

优先队列 是堆的常用场合，并且它的实现包含了多个挑战：

- 排序的穩定性：你如何將兩個擁有相同 priority 的 task 按照他們被加入的順序回傳。
- Tuple 的排序在某些情🄵下會壞掉，例如當 Tuple (priority, task) 的 priorities 相等且 tasks 🄵有一個預設的排序時。
- 當一個 heap 中 task 的 priority 改變時，你如何將它移到 heap 正確的位置上。
- 或者一個還🄵被解🄵的 task 需要被🄵除時，你要如何從🄵列中找到🄵🄵除指定的 task。

一個針對前兩個問題的解法是：儲存一個包含 priority 、entry count 和 task 三個元素的 tuple 。兩個 task 有相同 priority 時，entry count 會讓兩個 task 能根據加入的順序排序。因🄵🄵有任何兩個 task 擁有相同的 entry count ，所以永遠不會直接使用 task 做比較。

不可比较任务问题的另一种解决方案是创建一个忽略任务条目并且只比较优先级字段的包装器类：

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

剩下的問題可以藉由找到要🄴除的 task 🄵更改它的 priority 或者直接將它移除。尋找一個 task 可以使用一個 dictionary 指向🄵列當中的 entry 。

移除 entry 或更改它的 priority 更🄵困難，因🄴這會破壞 heap 的性質。所以一個可行的方案是將原本的 entry 做一個標記表示它已經被🄴除，🄵新增一個擁有新的 priority 的 entry 。

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

8.6.3 原理

Heap 是一個陣列對於所有從 0 開始的 index *k* 都存在性質 `a[k] <= a[2*k+1]` 和 `a[k] <= a[2*k+2]`。F了方便比較，不存在的元素被視F無限大。一個有趣的 heap 性質是 `a[0]` 永遠是最小的元素。

上面的特殊不变量是用来作为一场锦标赛的高效内存表示。下面的数字是 *k* 而不是 `a[k]`:

```
                                0

             1                                  2

      3              4                5               6

  7       8       9       10      11      12      13      14

15 16   17 18   19 20   21 22   23 24   25 26   27 28   29 30
```

在上面的树中，每个 *k* 单元都位于 `2*k+1` 和 `2*k+2` 之上。体育运动中我们经常见到二元锦标赛模式，每个胜者单元都位于另两个单元之上，并且我们可以沿着树形图向下追溯胜者所遇到的所有对手。但是，在许多采用这种锦标赛模式的计算机应用程序中，我们并不需要追溯胜者的历史。为了获得更高的内存利用效率，当一个胜者晋级时，我们会用较低层级的另一条目来替代它，因此规则变为一个单元和它之下的两个单元包含三个不同条目，上方单元“胜过”了两个下方单元。

如果此堆的不变性质始终受到保护，则序号 0 显然是总的赢家。删除它并找出“下一个”赢家的最简单算法方式是将某个输家（让我们假定是上图中的 30 号单元）移至 0 号位置，然后将这个新的 0 号沿树下行，不断进行值的交换，直到不变性质得到重建。这显然会是树中条目总数的对数。通过迭代所有条目，你将得到一个 O(n log n) 复杂度的排序。

此排序有一个很好的特性就是你可以在排序进行期间高效地插入新条目，前提是插入的条目不比你最近取出的 0 号元素“更好”。这在模拟上下文时特别有用，在这种情况下树保存的是所有传入事件，“胜出”条件是最小调度时间。当一个事件将其他事件排入执行计划时，它们的调试时间向未来方向延长，这样它们可方便地入堆。因此，堆结构很适宜用来实现调度器，我的 MIDI 音序器就是用的这个:-)。

用于实现调度器的各种结构都得到了充分的研究，堆是非常适宜的一种，因为它们的速度相当快，并且几乎是恒定的，最坏的情况与平均情况没有太大差别。虽然还存在其他总体而言更高效的实现方式，但其最坏的情况却可能非常糟糕。

堆在大磁盘排序中也非常有用。你应该已经了解大规模排序会有多个“运行轮次”（即预排序的序列，其大小通常与 CPU 内存容量相关），随后这些轮次会进入合并通道，轮次合并的组织往往非常巧妙[1]。非常重要的一点是初始排序应产生尽可能长的运行轮次。锦标赛模式是达成此目标的好办法。如果你使用全部有用内存来进行锦标赛，替换和安排恰好适合当前运行轮次的条目，你将可以对于随机输入生成两倍于内存大小的运行轮次，对于模糊排序的输入还会有更好的效果。

另外，如果你输出磁盘上的第 0 个条目并获得一个可能不适合当前锦标赛的输入（因为其值要“胜过”上一个输出值），它无法被放入堆中，因此堆的尺寸将缩小。被释放的内存可以被巧妙地立即重用以逐步构建第二个堆，其增长速度与第一个堆的缩减速度正好相同。当第一个堆完全消失时，你可以切换新堆并启动新的运行轮次。这样做既聪明又高效!

总之，堆是值得了解的有用内存结构。我在一些应用中用到了它们，并且认为保留一个'heap' 模块是很有意义的。:-)

𝔽解

8.7 `bisect` --- 陣列二分演算法 (Array bisection algorithm)

原始碼： Lib/bisect.py

這個模組維護一個已經排序過的 list，當我們每次做完插入後不需要再次排序整個 list。一個很長的 list 的比較操作很花費時間，𝔽了改進這點，這個模組是其中一個常用的方法。這個模組被命名𝔽*bisect* 來自他使用一個基礎的 bisection 演算法實作。模組的原始碼是這個演算法的一個完善的實作（邊界條件已經是正確的了）。

此模組提供下面的函式

`bisect.`**`bisect_left`**(*a*, *x*, *lo=0*, *hi=len(a)*)

在 *a* 當中找到一個位置，讓 *x* 插入後 *a* 仍然是排序好的。參數 *lo* 和 *hi* 用來指定 list 中應該被考慮的子區間，預設是考慮整個 list。如果 *a* 𝔽面已經有 *x* 出現，插入的位置會在所有 *x* 的前面（左邊）。回傳值可以被當作 `list.insert()` 的第一個參數，但列表 *a* 必須先排序過。

回傳的插入位置 *i* 將陣列 *a* 分𝔽兩半，使得 `all(val < x for val in a[lo:i])` 都在左側且 `all(val >= x for val in a[i:hi])` 都在右側。

`bisect.`**`bisect_right`**(*a*, *x*, *lo=0*, *hi=len(a)*)

`bisect.`**`bisect`**(*a*, *x*, *lo=0*, *hi=len(a)*)

類似*`bisect_left()`*，但回傳的插入位置會在所有 *a* 當中的 *x* 的後面（右邊）。

回傳的插入位置 *i* 將陣列 *a* 分𝔽兩半，使得 `all(val <= x for val in a[lo:i])` 都在左側且 `all(val > x for val in a[i:hi])` 都在右側。

[1] 当前时代的磁盘平衡算法与其说是巧妙，不如说是麻烦，这是由磁盘的寻址能力导致的结果。在无法寻址的设备例如大型磁带机上，情况则相当不同，开发者必须非常聪明地（极为提前地）确保每次磁带转动都尽可能地高效（就是说能够最好地加入到合并“进程”中）。有些磁带甚至能够反向读取，这也被用来避免倒带的耗时。请相信我，真正优秀的磁带机排序看起来是极其壮观的，排序从来都是一门伟大的艺术！:-)

bisect.**insort_left**(*a*, *x*, *lo=0*, *hi=len(a)*)
不破壞 *a* 的排序下插入 *x*，這等價於 `a.insert(bisect.bisect_left(a, x, lo, hi), x)`，注意 *a* 必須是已經排序過的 list。注意搜尋只需要 O(log n) 時間而插入需要很慢的 O(n) 時間，這使得插入操作主導了需要的花費時間。

bisect.**insort_right**(*a*, *x*, *lo=0*, *hi=len(a)*)
bisect.**insort**(*a*, *x*, *lo=0*, *hi=len(a)*)
類似*insort_left()*，但插入的位置會在所有 *a* 當中的 *x* 的後面（右邊）。

也參考:

SortedCollection recipe 使用 bisect 构造了一个功能完整的集合类，提供了直接的搜索方法和对用于搜索的 key 方法的支持。所有用于搜索的键都是预先计算的，以避免在搜索时对 key 方法的不必要调用。

8.7.1 搜索有序列表

上面的*bisect()*函数对于找到插入点是有用的，但在一般的搜索任务中可能会有点尴尬。下面 5 个函数展示了如何将其转变成有序列表中的标准查找函数

```
def index(a, x):
    'Locate the leftmost value exactly equal to x'
    i = bisect_left(a, x)
    if i != len(a) and a[i] == x:
        return i
    raise ValueError

def find_lt(a, x):
    'Find rightmost value less than x'
    i = bisect_left(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_le(a, x):
    'Find rightmost value less than or equal to x'
    i = bisect_right(a, x)
    if i:
        return a[i-1]
    raise ValueError

def find_gt(a, x):
    'Find leftmost value greater than x'
    i = bisect_right(a, x)
    if i != len(a):
        return a[i]
    raise ValueError

def find_ge(a, x):
    'Find leftmost item greater than or equal to x'
    i = bisect_left(a, x)
    if i != len(a):
        return a[i]
    raise ValueError
```

8.7.2 其他示例

函数*bisect()* 还可以用于数字表查询。这个例子是使用*bisect()* 从一个给定的考试成绩集合里，通过一个有序数字表，查出其对应的字母等级：90 分及以上是'A'，80 到 89 是'B'，以此类推

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

与*sorted()* 函数不同，对于*bisect()* 函数来说，*key* 或者 *reversed* 参数并没有什么意义。因为这会导致设计效率低下（连续调用 bisect 函数时，是不会" 记住" 过去查找过的键的）。

正相反，最好去搜索预先计算好的键列表，来查找相关记录的索引。

```
>>> data = [('red', 5), ('blue', 1), ('yellow', 8), ('black', 0)]
>>> data.sort(key=lambda r: r[1])
>>> keys = [r[1] for r in data]         # precomputed list of keys
>>> data[bisect_left(keys, 0)]
('black', 0)
>>> data[bisect_left(keys, 1)]
('blue', 1)
>>> data[bisect_left(keys, 5)]
('red', 5)
>>> data[bisect_left(keys, 8)]
('yellow', 8)
```

8.8 array --- 高效率的数值型態陣列

這個模組定義了一個物件型🄵，可以簡潔的表達一個包含基本數值的陣列：字元、整數、浮點數。陣列是一個非常類似 list 的序列型態，除了陣列會限制儲存的物件型🄵。在建立陣列時可以使用一個字元的 *type code* 來指定儲存的資料型🄵。下面是 type codes 的定義。

Type code	C Type	Python Type	最小所需的位元組	🄵解
'b'	signed char	int	1	
'B'	unsigned char	int	1	
'u'	wchar_t	Unicode character	2	(1)
'h'	signed short	int	2	
'H'	unsigned short	int	2	
'i'	signed int	int	2	
'I'	unsigned int	int	2	
'l'	signed long	int	4	
'L'	unsigned long	int	4	
'q'	signed long long	int	8	
'Q'	unsigned long long	int	8	
'f'	float	float	4	
'd'	double	float	8	

🄵解：

(1) 由于平台的不同它可能为 16 位或 32 位。

3.9 版更變: `array('u')` 现在使用 `wchar_t` 作为 C 类型而不再是已弃用的 `Py_UNICODE`。这个改变不会影响其行为，因为 `Py_UNICODE` 自 Python 3.3 起就是 `wchar_t` 的别名。

Deprecated since version 3.3, will be removed in version 4.0.

實際上數值的表示方法是被機器的架構所🄵定（更精准地🄵，被 C 的實作方法🄵定）。實際的大小可以透過 `itemsize` 屬性存取。

這個模組定義了下方的型🄵:

class `array.`**array**(*typecode*[, *initializer*])

一個新的陣列中的元素被 *typecode* 限制，🄵由選用的 *initializer* 參數初始化，*initializer* 必須是一個 list、*bytes-like object* 或包含適當型🄵變數的 iterable 。

如果指定一個 list 或 string ，新的陣列初始化時會傳入*fromlist()* 、*frombytes()* 或*fromunicode()* 方法（參照下方）將元素新增到其中。其他型態的變數則會傳入*extend()* 方法初始化。

引发一个审计事件 `array.__new__` 附带参数 `typecode, initializer`。

`array.`**typecodes**

一個包含所有可用的 type code 的字串。

数组对象支持普通的序列操作如索引、切片、拼接和重复等。当使用切片赋值时，所赋的值必须为具有相同类型码的数组对象；所有其他情况都将引发*TypeError*。数组对象也实现了缓冲区接口，可以用于所有支持字节类对象 的场合。

提供下方的資料物件與方法。

`array.`**typecode**

typecode 字元被用在建立陣列時。

`array.`**itemsize**

陣列當中的一個元素在🄵部需要的位元組 (bytes) 長度。

`array.`**append**(*x*)

新增一個元素 *x* 到陣列的最尾端。

`array.`**buffer_info**()

回傳一個 tuple `(address, length)` 表示當前的記憶體位置和陣列儲存元素的緩衝區記憶體長度。緩衝區的長度單位是 bytes ，🄵可以用 `array.buffer_info()[1] * array.itemsize` 計算得到。這偶爾會在底層操作需要記憶體位置的輸出輸入時很有用，例如 `ioctl()` 指令。只要陣列存在且🄵有使用任何更改長度的操作時，回傳的數值就有效。

備🄵: 當使用來自 C 或 C++ 程式碼（這是唯一使得這個資訊有效的途徑）的陣列物件時，更適當的做法是使用陣列物件支援的緩衝區介面。這個方法維護了向後兼容性，🄵應該在新的程式碼中避免。關於緩衝區介面的文件在 bufferobjects 。

`array.`**byteswap**()

”Byteswap” 所有陣列中的物件。這只有支援物件長度🄵 1、2、4 或 8 位元組的陣列，其他型🄵的值會導致*RuntimeError* 。這在從機器讀取位元順序不同的檔案時很有用。

`array.`**count**(*x*)

回傳 *x* 在陣列中出現了幾次。

`array.`**extend**(*iterable*)

從 *iterable* 中新增元素到陣列的尾端，如果 *iterable* 是另一個陣列，他必須有完全相同的 type code ，如果不同會🄵生*TypeError* 。如果 *iterable* 不是一個陣列，他必須可以被迭代 (iterable) 且其中的元素必須是可以被加入陣列中的正確型態。

array.**frombytes**(*s*)
從字串中新增元素。讀取時會將字串當作一個陣列，🄵面包含了 machine value（就像從檔案中使用*fromfile()* 方法讀出的資料）。

3.2 版新加入: 🄵了更明確，之前的 `fromstring()` 被更名🄵*frombytes()* 。

array.**fromfile**(*f*, *n*)
從*file object* *f* 讀取 *n* 個 machine value 類型的元素，接著將這些元素加入陣列的最尾端。如果只有少於 *n* 個有效的元素會🄵生*EOFError* 錯誤，但有效的元素仍然會被加入陣列中。

array.**fromlist**(*list*)
從 list 中新增元素。這等價於 `for x in list: a.append(x)` ，除了有型態錯誤🄵生時，陣列會保持原狀不會被更改。

array.**fromunicode**(*s*)
用給定的 unicode 字串擴展這個陣列。陣列必須是型態 `u` 的陣列；其他的型態會🄵生*ValueError* 錯誤。使用 `array.frombytes(unicodestring.encode(enc))` 來新增 Unicode 資料到一個其他型態的陣列。

array.**index**(*x*)
回傳最小的 *i* ，使得 *i* 是陣列中第一個 *x* 出現的索引值。

array.**insert**(*i*, *x*)
在位置 *i* 之前插入一個元素 *x* 。負數的索引值會從陣列尾端開始數。

array.**pop**([*i*])
移除🄵回傳陣列索引值 *i* 的元素。選擇性的參數 *i* 預設🄵 `-1` ，所以預設會🄵除🄵回傳最後一個元素。

array.**remove**(*x*)
從陣列中🄵除第一個出現的 *x* 。

array.**reverse**()
將整個陣列的元素按照順序逆轉。

array.**tobytes**()
將陣列轉🄵另一個 machine values 的陣列🄵回傳他的位元組表示（跟用*tofile()* 方法寫入檔案時的位元序列相同）。

3.2 版新加入: 🄵了明確性，過去的 `tostring()` 已更名🄵*tobytes()* 。

array.**tofile**(*f*)
將所有元素（以 machine code 的形式）寫入*file object* *f* 。

array.**tolist**()
不更改元素，將陣列轉🄵一般的 list 。

array.**tounicode**()
將陣列轉🄵一個字串。陣列的型態必須🄵 `u` 。其他型態的陣列會🄵生*ValueError* 錯誤。使用 `array.tobytes().decode(enc)` 將其他型態的陣列轉🄵字串。

当一个数组对象被打印或转换为字符串时，它会表示为 `array(typecode, initializer)`。如果数组为空则 *initializer* 会被省略，否则如果 *typecode* 为 `'u'` 则它是一个字符串，否则它是一个数字列表。使用*eval()* 保证能将字符串转换回具有相同类型和值的数组，只要*array* 类已通过 `from array import array` 被引入。例如:

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

也参考:

模組*struct* 將包含不同資料類型的二進位資料包裝與解開包裝。

模組*xdrlib* 將 External Data Representation (XDR) 的資料包裝與解開包裝，這用在一些遠端操作的系統 (remote procedure call systems) 。

NumPy NumPy 套件定義了另一個陣列型態

8.9 weakref --- 弱引用

源码： Lib/weakref.py

---

weakref 模块允许 Python 程序员创建对象的 *weak references* 。

在下文中，术语 *referent* 表示由弱引用引用的对象。

对对象的弱引用不能保证对象存活：当对像的引用只剩弱引用时，*garbage collection* 可以销毁引用并将其内存重用于其他内容。但是，在实际销毁对象之前，即使没有强引用，弱引用也一直能返回该对象。

弱引用的主要用途是实现保存大对象的高速缓存或映射，但又不希望大对象仅仅因为它出现在高速缓存或映射中而保持存活。

例如，如果您有许多大型二进制图像对象，则可能希望将名称与每个对象关联起来。如果您使用 Python 字典将名称映射到图像，或将图像映射到名称，则图像对象将保持活动状态，因为它们在字典中显示为值或键。*weakref* 模块提供的*WeakKeyDictionary* 和*WeakValueDictionary* 类可以替代 Python 字典，使用弱引用来构造映射，这些映射不会仅仅因为它们出现在映射对象中而使对象保持存活。例如，如果一个图像对象是*WeakValueDictionary* 中的值，那么当对该图像对象的剩余引用是弱映射对象所持有的弱引用时，垃圾回收可以回收该对象并将其在弱映射对象中相应的条目删除。

WeakKeyDictionary 和*WeakValueDictionary* 在它们的实现中使用弱引用，在弱引用上设置回调函数，当键或值被垃圾回收回收时通知弱字典。*WeakSet* 实现了*set* 接口，但像*WeakKeyDictionary* 一样，只持有其元素的弱引用。

finalize 提供了注册一个对象被垃圾收集时要调用的清理函数的方式。这比在原始弱引用上设置回调函数更简单，因为模块会自动确保对象被回收前终结器一直保持存活。

这些弱容器类型之一或者*finalize* 就是大多数程序所需要的 - 通常不需要直接创建自己的弱引用。*weakref* 模块暴露了低级机制，以便于高级用途。

并非所有对象都可以被弱引用；可以被弱引用的对象包括类实例，用 Python（而不是用 C）编写的函数，实例方法、集合、冻结集合，某些文件对象，生成器，类型对象，套接字，数组，双端队列，正则表达式模式对象以及代码对象等。

3.2 版更變: 添加了对 thread.lock，threading.Lock 和代码对象的支持。

几个内建类型如*list* 和*dict* 不直接支持弱引用，但可以通过子类化添加支持:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)   # this object is weak referenceable
```

CPython implementation detail: 其他内置类型例如*tuple* 和*int* 不支持弱引用，即使通过子类化也不支持。

Extension types can easily be made to support weak references; see weakref-support.

When `__slots__` are defined for a given type, weak reference support is disabled unless a `'__weakref__'` string is also present in the sequence of strings in the `__slots__` declaration. See __slots__ documentation for details.

class weakref.**ref**(*object*[, *callback*])
返回对 对象的弱引用。如果原始对象仍然存活，则可以通过调用引用对象来检索原始对象；如果引用的原始对象不再存在，则调用引用对象将得到*None*。如果提供了 回调而且值不是*None*，并且返回的弱引用对象仍然存活，则在对象即将终结时将调用回调; 弱引用对象将作为回调的唯一参数传递；指示物将不再可用。

许多弱引用也允许针对相同对象来构建。为每个弱引用注册的回调将按从最近注册的回调到最早注册的回调的顺序被调用。

回调所引发的异常将记录于标准错误输出，但无法被传播；它们会按与对象的 __del__() 方法所引发的异常相同的方式被处理。

如果 *object* 可哈希，则弱引用也为*hashable*。即使在 *object* 被删除之后它们仍将保持其哈希值。如果*hash()* 在 *object* 被删除之后才首次被调用，则该调用将引发*TypeError*。

弱引用支持相等检测，但不支持排序比较。如果被引用对象仍然存在，两个引用具有与它们的被引用对象一致的相等关系（无论 *callback* 是否相同）。如果删除了任一被引用对象，则仅在两个引用对象为同一对象时两者才相等。

这是一个可子类化的类型而非一个工厂函数。

__callback__
这个只读属性会返回当前关联到弱引用的回调。如果回调不存在或弱引用的被引用对象已不存在，则此属性的值为 None。

3.4 版更變: 添加了*__callback__* 属性。

weakref.**proxy**(*object*[, *callback*])
返回 *object* 的一个使用弱引用的代理。此函数支持在大多数上下文中使用代理，而不要求显式地对所使用的弱引用对象解除引用。返回的对象类型将为 ProxyType 或 CallableProxyType，具体取决于 *object* 是否可调用。Proxy 对象不是*hashable* 对象，无论被引用对象是否可哈希；这可避免与它们的基本可变性质相关的多种问题，并可防止它们被用作字典键。*callback* 与*ref()* 函数的同名形参含义相同。

3.8 版更變: 扩展代理对象所支持的运算符，包括矩阵乘法运算符 @ 和 @=。

weakref.**getweakrefcount**(*object*)
返回指向 *object* 的弱引用和代理的数量。

weakref.**getweakrefs**(*object*)
返回由指向 *object* 的所有弱引用和代理构成的列表。

class weakref.**WeakKeyDictionary**([*dict*])
弱引用键的映射类。当不再存在对键的强引用时，字典中的条目将被丢弃。这可被用来将额外数据关联到一个应用中其他部分所拥有的对象而无需在那些对象中添加属性。这对于重载了属性访问的对象来说特别有用。

3.9 版更變: 增加了对 | 和 |= 运算符的支持，相关说明见 **PEP 584**。

WeakKeyDictionary 对象具有一个额外方法可以直接公开内部引用。这些引用不保证在它们被使用时仍然保持“存活”，因此这些引用的调用结果需要在使用前进行检测。此方法可用于避免创建会导致垃圾回收器将保留键超出实际需要时长的引用。

WeakKeyDictionary.**keyrefs**()
返回包含对键的弱引用的可迭代对象。

class weakref.**WeakValueDictionary**([*dict*])
弱引用值的映射类。当不再存在对该值的强引用时，字典中的条目将被丢弃。

3.9 版更變: 增加了对 | 和 |= 运算符的支持，相关说明见 **PEP 584**。

WeakValueDictionary 对象具有一个额外方法，此方法存在与*WeakKeyDictionary* 对象的 keyrefs() 方法相同的问题。

WeakValueDictionary.**valuerefs**()
返回包含对值的弱引用的可迭代对象。

class weakref.**WeakSet**([*elements*])
保持对其元素弱引用的集合类。当不再有对某个元素的强引用时元素将被丢弃。

class weakref.**WeakMethod**(*method*)
一个模拟对绑定方法（即在类中定义并在实例中查找的方法）进行弱引用的自定义*ref* 子类。由于绑定方法是临时性的，标准弱引用无法保持它。*WeakMethod* 包含特别代码用来重新创建绑定方法，直到对象或初始函数被销毁:

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>
```

3.4 版新加入.

class weakref.**finalize**(*obj*, *func*, */*, **args*, ***kwargs*)
返回一个可调用的终结器对象，该对象将在 *obj* 作为垃圾回收时被调用。与普通的弱引用不同，终结器将总是存活，直到引用对象被回收，这极大地简化了生存期管理。

终结器总是被视为 *存活* 直到它被调用（显式调用或在垃圾回收时隐式调用），调用之后它将 *死亡*。调用存活的终结器将返回 `func(*arg, **kwargs)` 的求值结果，而调用死亡的终结器将返回*None*。

在垃圾收集期间由终结器回调所引发异常将显示于标准错误输出，但无法被传播。它们会按与对象的 `__del__()` 方法或弱引用的回调所引发异常相同的方式被处理。

当程序退出时，剩余的存活终结器会被调用，除非它们的*atexit* 属性已被设为假值。它们会按与创建时相反的顺序被调用。

终结器在*interpreter shutdown* 的后期绝不会发起调用其回调函数，此时模块全局变量很可能已被替换为*None*。

__call__()
如果 *self* 为存活状态则将其标记为已死亡，并返回调用 `func(*args, **kwargs)` 的结果。如果 *self* 已死亡则返回*None*。

detach()
如果 *self* 为存活状态则将其标记为已死亡，并返回元组 `(obj, func, args, kwargs)`。如果 *self* 已死亡则返*None*。

peek()
如果 *self* 为存活状态则返回元组 `(obj, func, args, kwargs)`。如果 *self* 已死亡则返回*None*。

alive
如果终结器为存活状态则该特征属性为真值，否则为假值。

atexit
一个可写的布尔型特征属性，默认为真值。当程序退出时，它会调用所有*atexit* 为真值的剩余存活终结器。它们会按与创建时相反的顺序被调用。

備𝔽: 很重要的一点是确保 *func*, *args* 和 *kwargs* 不拥有任何对 *obj* 的引用，无论是直接的或是间接的，否则的话 *obj* 将永远不会被作为垃圾回收。特别地，*func* 不应当是 *obj* 的一个绑定方法。

3.4 版新加入.

weakref.**ReferenceType**
弱引用对象的类型对象。

weakref.**ProxyType**
不可调用对象的代理的类型对象。

weakref.**CallableProxyType**
可调用对象的代理的类型对象。

weakref.**ProxyTypes**
包含所有代理的类型对象的序列。这可以用于更方便地检测一个对象是否是代理，而不必依赖于两种代理对象的名称。

也参考:

PEP 205 - 弱引用 此特性的提议和理由，包括早期实现的链接和其他语言中类似特性的相关信息。

8.9.1 弱引用对象

弱引用对象没有*ref.__callback__* 以外的方法和属性。一个弱引用对象如果存在，就允许通过调用它来获取引用:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

如果引用已不存在，则调用引用对象将返回*None*:

```
>>> del o, o2
>>> print(r())
None
```

检测一个弱引用对象是否仍然存在应该使用表达式 `ref() is not None`。通常，需要使用引用对象的应用代码应当遵循这样的模式:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
```

(下页继续)

(繼續上一頁)

```
    print("Object is still live!")
    o.do_something_useful()
```

使用单独的“存活”测试会在多线程应用中制造竞争条件；其他线程可能导致某个弱引用在该弱引用被调用前就失效；上述的写法在多线程应用和单线程应用中都是安全的。

特别版本的*ref* 对象可以通过子类化来创建。在*WeakValueDictionary* 的实现中就使用了这种方式来减少映射中每个条目的内存开销。这对于将附加信息关联到引用的情况最为适用，但也可以被用于在调用中插入额外处理来提取引用。

这个例子演示了如何将*ref* 的一个子类用于存储有关对象的附加信息并在引用被访问时影响其所返回的值:

```
import weakref

class ExtendedRef(weakref.ref):
    def __init__(self, ob, callback=None, /, **annotations):
        super().__init__(ob, callback)
        self.__counter = 0
        for k, v in annotations.items():
            setattr(self, k, v)

    def __call__(self):
        """Return a pair containing the referent and the number of
        times the reference has been called.
        """
        ob = super().__call__()
        if ob is not None:
            self.__counter += 1
            ob = (ob, self.__counter)
        return ob
```

8.9.2 示例

这个简单的例子演示了一个应用如何使用对象 ID 来提取之前出现过的对象。然后对象的 ID 可以在其它数据结构中使用，而无须强制对象保持存活，但处于存活状态的对象也仍然可以通过 ID 来提取。

```
import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]
```

8.9.3 终结器对象

使用*finalize*的主要好处在于它能更简便地注册回调函数，而无须保留所返回的终结器对象。例如

```
>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>
>>> del kenny
You killed Kenny!
```

终结器也可以被直接调用。但是终结器最多只能对回调函数发起一次调用。

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()                     # callback not called because finalizer dead
>>> del obj                 # callback not called because finalizer dead
```

你可以使用*detach()*方法来注销一个终结器。该方法将销毁终结器并返回其被创建时传给构造器的参数。

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<...Object object ...>, <function callback ...>, (1, 2), {'z': 3})
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

除非你将*atexit*属性设为*False*，否则终结器在程序退出时如果仍然存活就将被调用。例如

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

8.9.4 比较终结器与 __del__() 方法

假设我们想创建一个类，用它的实例来代表临时目录。当以下事件中的某一个发生时，这个目录应当与其内容一起被删除：

- 对象被作为垃圾回收，
- 对象的 remove() 方法被调用，或
- 程序退出。

我们可以尝试使用 __del__() 方法来实现这个类，如下所示：

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

从 Python 3.4 开始，__del__() 方法不会再阻止循环引用被作为垃圾回收，并且模块全局变量在*interpreter shutdown* 期间不会被强制设为*None*。因此这段代码在 CPython 上应该会正常运行而不会出现任何问题。

然而，__del__() 方法的处理会严重地受到具体实现的影响，因为它依赖于解释器垃圾回收实现方式的内部细节。

更健壮的替代方式可以是定义一个终结器，只引用它所需要的特定函数和对象，而不是获取对整个对象状态的访问权：

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.name)

    def remove(self):
        self._finalizer()

    @property
    def removed(self):
        return not self._finalizer.alive
```

像这样定义后，我们的终结器将只接受一个对其完成正确清理目录任务所需细节的引用。如果对象一直未被作为垃圾回收，终结器仍会在退出时被调用。

基于弱引用的终结器还具有另一项优势，就是它们可被用来为定义由第三方控制的类注册终结器，例如当一个模块被卸载时运行特定代码：

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the function body
weakref.finalize(sys.modules[__name__], unloading_module)
```

備𝔽: 如果当程序退出时你恰好在守护线程中创建终结器对象，则有可能该终结器不会在退出时被调用。但是，在一个守护线程中*atexit.register()*，`try: ... finally: ...` 和 `with: ...` 同样不能保证执行清理。

8.10 types --- 动态类型创建和内置类型名称

源代码: Lib/types.py

此模块定义了一些工具函数，用于协助动态创建新的类型。

它还为某些对象类型定义了名称，这些名称由标准 Python 解释器所使用，但并不像内置的*int* 或*str* 那样对外公开。

最后，它还额外提供了一些类型相关但重要程度不足以作为内置对象的工具类和函数。

8.10.1 动态类型创建

types.**new_class**(*name*, *bases=()*, *kwds=None*, *exec_body=None*)
使用适当的元类动态地创建一个类对象。

前三个参数是组成类定义头的部件：类名称，基类 (有序排列)，关键字参数 (例如 `metaclass`)。

exec_body 参数是一个回调函数，用于填充新创建类的命名空间。它应当接受类命名空间作为其唯一的参数并使用类内容直接更新命名空间。如果未提供回调函数，则它就等效于传入 `lambda ns: None`。

3.3 版新加入.

types.**prepare_class**(*name*, *bases=()*, *kwds=None*)
计算适当的元类并创建类命名空间。

参数是组成类定义头的部件：类名称，基类 (有序排列) 以及关键字参数 (例如 `metaclass`)。

返回值是一个 3 元组: `metaclass, namespace, kwds`

metaclass 是适当的元类，*namespace* 是预备好的类命名空间而 *kwds* 是所传入 *kwds* 参数移除每个 `'metaclass'` 条目后的已更新副本。如果未传入 *kwds* 参数，这将为一个空字典。

3.3 版新加入.

3.6 版更變: 所返回元组中 `namespace` 元素的默认值已被改变。现在当元类没有 `__prepare__` 方法时将会使用一个保留插入顺序的映射。

也参考:

metaclasses 这些函数所支持的类创建过程的完整细节

PEP 3115 - Python 3000 中的元类 引入 `__prepare__` 命名空间钩子

types.**resolve_bases**(*bases*)
动态地解析 MRO 条目，具体描述见 **PEP 560**。

此函数会在 *bases* 中查找不是*type* 的实例的项，并返回一个元组，其中每个具有 `__mro_entries__` 方法的此种对象对象将被替换为调用该方法解包后的结果。如果一个 *bases* 项是*type* 的实例，或它不具有 `__mro_entries__` 方法，则它将不加改变地被包含在返回的元组中。

3.7 版新加入.

也参考:

PEP 560 - 对类型模块和泛型类型的核心支持

8.10.2 标准解释器类型

此模块为许多类型提供了实现 Python 解释器所要求的名称。它刻意地避免了包含某些仅在处理过程中偶然出现的类型，例如 `listiterator` 类型。

此种名称的典型应用如*isinstance()* 或*issubclass()* 检测。

如果你要实例化这些类型中的任何一种，请注意其签名在不同 Python 版本之间可能出现变化。

以下类型有相应的标准名称定义:

types.**FunctionType**
types.**LambdaType**
用户自定义函数以及由 `lambda` 表达式所创建函数的类型。

引发一个审计事件 `function.__new__`，附带参数 `code`。

此审计事件只会被函数对象的直接实例化引发，而不会被普通编译所引发。

types.**GeneratorType**
generator 迭代器对象的类型，由生成器函数创建。

types.**CoroutineType**
coroutine 对象的类型，由 `async def` 函数创建。

3.5 版新加入.

types.**AsyncGeneratorType**
asynchronous generator 迭代器对象的类型，由异步生成器函数创建。

3.6 版新加入.

class types.**CodeType**(***kwargs*)
代码对象的类型，例如*compile()* 的返回值。

引发审计事件 `code.__new__` 附带参数 `code`, `filename`, `name`, `argcount`, `posonlyargcount`, `kwonlyargcount`, `nlocals`, `stacksize`, `flags`。

请注意被审计的参数可能与初始化代码所要求的名称或位置不相匹配。审计事件只会被代码对象的直接实例化引发，而不会被普通编译所引发。

replace(***kwargs*)
返回代码对象的一个副本，使用指定的新字段值。

3.8 版新加入.

types.**CellType**
单元对象的类型：这种对象被用作函数中自由变量的容器。

3.8 版新加入.

types.**MethodType**
用户自定义类实例方法的类型。

types.**BuiltinFunctionType**
types.**BuiltinMethodType**
内置函数例如*len()* 或*sys.exit()* 以及内置类方法的类型。(这里所说的“内置”是指“以 C 语言编写”。)

types.**WrapperDescriptorType**
某些内置数据类型和基类的方法的类型，例如 object.__init__() 或 object.__lt__()。

3.7 版新加入.

types.**MethodWrapperType**
某些内置数据类型和基类的 绑定方法的类型。例如 object().__str__ 所属的类型。

3.7 版新加入.

types.**MethodDescriptorType**
某些内置数据类型方法例如*str.join()* 的类型。

3.7 版新加入.

types.**ClassMethodDescriptorType**
某些内置数据类型 非绑定类方法例如 dict.__dict__['fromkeys'] 的类型。

3.7 版新加入.

class types.**ModuleType**(*name*, *doc=None*)
模块 的类型。构造器接受待创建模块的名称并以其*docstring* 作为可选参数。

備🄴: 如果你希望设置各种由导入控制的属性，请使用*importlib.util.module_from_spec()* 来创建一个新模块。

__doc__
模块的*docstring*。默认为 None。

__loader__
用于加载模块的*loader*。默认为 None。

此属性会匹配保存在 attr:*__spec__* 对象中的*importlib.machinery.ModuleSpec.loader*。

備🄴: A future version of Python may stop setting this attribute by default. To guard against this potential change, preferably read from the __spec__ attribute instead or use getattr(module, "__loader__", None) if you explicitly need to use this attribute.

3.4 版更變: 默认为 None。之前该属性为可选项。

__name__
模块的名称。应当能匹配*importlib.machinery.ModuleSpec.name*。

__package__
一个模块所属的*package*。如果模块为最高层级的（即不是任何特定包的组成部分）则该属性应设为 ''，否则它应设为特定包的名称 (如果模块本身也是一个包则名称可以为 __name__)。默认为 None。

此属性会匹配保存在 attr:*__spec__* 对象中的*importlib.machinery.ModuleSpec.parent*。

備🄴: A future version of Python may stop setting this attribute by default. To guard against this potential change, preferably read from the __spec__ attribute instead or use getattr(module, "__package__", None) if you explicitly need to use this attribute.

3.4 版更變: 默认为 None。之前该属性为可选项。

__spec__
模块的导入系统相关状态的记录。应当是一个*importlib.machinery.ModuleSpec* 的实例。

3.4 版新加入.

class types.**GenericAlias**(*t_origin*, *t_args*)
形参化泛型 的类型，例如 `list[int]`。

`t_origin` 应当是一个非形参化的泛型类，例如 `list`, `tuple` 或 `dict`。`t_args` 应当是一个形参化 `t_origin` 的*tuple* (长度可以为 1):

```
>>> from types import GenericAlias

>>> list[int] == GenericAlias(list, (int,))
True
>>> dict[str, int] == GenericAlias(dict, (str, int))
True
```

3.9 版新加入.

3.9.2 版更變: 此类型现在可以被子类化。

class types.**TracebackType**(*tb_next*, *tb_frame*, *tb_lasti*, *tb_lineno*)
回溯对象的类型，例如 `sys.exc_info()[2]` 中的对象。

请查看 语言参考了解可用属性和操作的细节，以及动态地创建回溯对象的指南。

types.**FrameType**
帧对象的类型，例如 `tb.tb_frame` 中的对象，其中 `tb` 是一个回溯对象。

请查看 语言参考了解可用属性和操作的细节。

types.**GetSetDescriptorType**
使用 `PyGetSetDef` 在扩展模块中定义的对象的类型，例如 `FrameType.f_locals` 或 `array.array.typecode`。此类型被用作对象属性的描述器；它的目的与*property* 类型相同，但专门针对在扩展模块中定义的类。

types.**MemberDescriptorType**
使用 `PyMemberDef` 在扩展模块中定义的对象的类型，例如 `datetime.timedelta.days`。此类型被用作使用标准转换函数的简单 C 数据成员的描述器；它的目的与*property* 类型相同，但专门针对在扩展模块中定义的类。

CPython implementation detail: 在 Python 的其它实现中，此类型可能与 `GetSetDescriptorType` 完全相同。

class types.**MappingProxyType**(*mapping*)
一个映射的只读代理。它提供了对映射条目的动态视图，这意味着当映射发生改变时，视图会反映这些改变。

3.3 版新加入.

3.9 版更變: 更新为支持 **PEP 584** 所新增的合并 (`|`) 运算符，它会简单地委托给下层的映射。

key in proxy
如果下层的映射中存在键 *key* 则返回 `True`，否则返回 `False`。

proxy[key]
返回下层的映射中以 *key* 为键的项。如果下层的映射中不存在键 *key* 则引发*KeyError*。

iter(proxy)
返回由下层映射的键为元素的迭代器。这是 `iter(proxy.keys())` 的快捷方式。

len(proxy)
返回下层映射中的项数。

copy()
返回下层映射的浅拷贝。

get(*key*[, *default*])
如果 *key* 存在于下层映射中则返回 *key* 的值，否则返回 *default*。如果 *default* 未给出则默认为 None，因而此方法绝不会引发*KeyError*。

items()
返回由下层映射的项 ((键, 值) 对) 组成的一个新视图。

keys()
返回由下层映射的键组成的一个新视图。

values()
返回由下层映射的值组成的一个新视图。

reversed(proxy)
返回一个包含下层映射的键的反向迭代器。

3.9 版新加入.

8.10.3 附加工具类和函数

class types.**SimpleNamespace**
一个简单的*object* 子类，提供了访问其命名空间的属性，以及一个有意义的 repr。

不同于*object*，对于 SimpleNamespace 你可以添加和移除属性。如果一个 SimpleNamespace 对象使用关键字参数进行初始化，这些参数会被直接加入下层命名空间。

此类型大致等价于以下代码:

```
class SimpleNamespace:
    def __init__(self, /, **kwargs):
        self.__dict__.update(kwargs)

    def __repr__(self):
        items = (f"{k}={v!r}" for k, v in self.__dict__.items())
        return "{}({})".format(type(self).__name__, ", ".join(items))

    def __eq__(self, other):
        if isinstance(self, SimpleNamespace) and isinstance(other,␣
↪SimpleNamespace):
            return self.__dict__ == other.__dict__
        return NotImplemented
```

SimpleNamespace 可被用于替代 class NS: pass。但是，对于结构化记录类型则应改用*namedtuple()*。

3.3 版新加入.

3.9 版更變: repr 中的属性顺序由字母顺序改为插入顺序 (类似 dict)。

types.**DynamicClassAttribute**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)
在类上访问 __getattr__ 的路由属性。

这是一个描述器，用于定义通过实例与通过类访问时具有不同行为的属性。当实例访问时保持正常行为，但当类访问属性时将被路由至类的 __getattr__ 方法；这是通过引发 AttributeError 来完成的。

这允许有在实例上激活的特性属性，同时又有在类上的同名虚拟属性 (一个例子请参见*enum.Enum*)。

3.4 版新加入.

8.10.4 协程工具函数

types.**coroutine**(*gen_func*)
此函数可将*generator* 函数转换为返回基于生成器的协程的*coroutine function*。基于生成器的协程仍然属于*generator iterator*，但同时又可被视为*coroutine* 对象兼*awaitable*。不过，它没有必要实现 `__await__()` 方法。

如果 *gen_func* 是一个生成器函数，它将被原地修改。

如果 *gen_func* 不是一个生成器函数，则它会被包装。如果它返回一个`collections.abc.Generator` 的实例，该实例将被包装在一个 *awaitable* 代理对象中。所有其他对象类型将被原样返回。

3.5 版新加入.

8.11 copy --- 浅层 (shallow) 和深层 (deep) 复制操作

源代码: Lib/copy.py

Python 的赋值语句不复制对象，而是创建目标和对象的绑定关系。对于自身可变，或包含可变项的集合，有时要生成副本用于改变操作，而不必改变原始对象。本模块提供了通用的浅层复制和深层复制操作，（如下所述）。

接口摘要：

copy.**copy**(*x*)
返回 *x* 的浅层复制。

copy.**deepcopy**(*x*[, *memo*])
返回 *x* 的深层复制。

exception copy.**Error**
针对模块特定错误引发。

浅层与深层复制的区别仅与复合对象（即包含列表或类的实例等其他对象的对象）相关：

- *浅层复制*构造一个新的复合对象，然后（在尽可能的范围内）将原始对象中找到的对象的*引用*插入其中。
- *深层复制*构造一个新的复合对象，然后，递归地将在原始对象里找到的对象的*副本*插入其中。

深度复制操作通常存在两个问题，而浅层复制操作并不存在这些问题：

- 递归对象 (直接或间接包含对自身引用的复合对象) 可能会导致递归循环。
- 由于深层复制会复制所有内容，因此可能会过多复制（例如本应该在副本之间共享的数据）。

deepcopy() 函数用以下方式避免了这些问题：

- 保留在当前复制过程中已复制的对象的”备忘录”（memo）字典；以及
- 允许用户定义的类重载复制操作或复制的组件集合。

This module does not copy types like module, method, stack trace, stack frame, file, socket, window, or any similar types. It does ”copy” functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the *pickle* module.

制作字典的浅层复制可以使用*dict.copy()* 方法，而制作列表的浅层复制可以通过赋值整个列表的切片完成，例如，`copied_list = original_list[:]`。

类可以使用与控制序列化（pickling）操作相同的接口来控制复制操作，关于这些方法的描述信息请参考*pickle* 模块。实际上，*copy* 模块使用的正是从*copyreg* 模块中注册的 pickle 函数。

想要给一个类定义它自己的拷贝操作实现，可以通过定义特殊方法 `__copy__()` 和 `__deepcopy__()`。调用前者以实现浅层拷贝操作，该方法不用传入额外参数。调用后者以实现深层拷贝操作；它应传入一个参数即 memo 字典。如果 `__deepcopy__()` 实现需要创建一个组件的深层拷贝，它应当调用*deepcopy()* 函数并以该组件作为第一个参数，而将 memo 字典作为第二个参数。

也参考：

模块*pickle* 讨论了支持对象状态检索和恢复的特殊方法。

8.12 pprint --- 数据美化输出

源代码： Lib/pprint.py

---

pprint 模块提供了“美化打印”任意 Python 数据结构的功能，这种美化形式可用作对解释器的输入。如果经格式化的结构包含非基本 Python 类型的对象，则其美化形式可能无法被加载。包含文件、套接字或类对象，以及许多其他不能用 Python 字面值来表示的对象都有可能导致这样的结果。

格式化后的形式会在可能的情况下以单行来表示对象，并在无法在允许宽度内容纳对象的情况下将其分为多行。如果你需要调整宽度限制则应显式地构造*PrettyPrinter* 对象。

字典在计算其显示形式前会先根据键来排序。

3.9 版更變: 添加了对美化打印*types.SimpleNamespace* 的支持。

pprint 模块定义了一个类：

class pprint.**PrettyPrinter**(*indent=1*, *width=80*, *depth=None*, *stream=None*, ***, *compact=False*, *sort_dicts=True*)

构造一个*PrettyPrinter* 实例。此构造器接受几个关键字形参。使用 *stream* 关键字可设置输出流；流对象使用的唯一方法是文件协议的 `write()` 方法。如果未指定此关键字，则*PrettyPrinter* 会选择 `sys.stdout`。每个递归层次的缩进量由 *indent* 指定；默认值为一。其他值可导致输出看起来有些怪异，，但可使得嵌套结构更易区分。可被打印的层级数量由 *depth* 控制；如果数据结构的层级被打印得过深，其所包含的下一层级会被替换为 `...`。在默认情况下，对被格式化对象的层级深度没有限制。希望的输出宽度可使用 *width* 形参来限制；默认值为 80 个字符。如果一个结构无法在限定宽度内被格式化，则将做到尽可能接近。如果 *compact* 为假值（默认）则长序列的每一项将被格式化为单独的行。如果 *compact* 为真值，则将在 *width* 可容纳的的情况下把尽可能多的项放入每个输出行。如果 *sort_dicts* 为真值（默认），字典将被格式化为按键排序，否则将按插入顺序显示。

3.4 版更變: 增加了 *compact* 形参。

3.8 版更變: 增加了 *sort_dicts* 形参。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff[:])
>>> pp = pprint.PrettyPrinter(indent=4)
>>> pp.pprint(stuff)
[   ['spam', 'eggs', 'lumberjack', 'knights', 'ni'],
    'spam',
    'eggs',
    'lumberjack',
    'knights',
    'ni']
```

(下页继续)

(繼續上一頁)

```
>>> pp = pprint.PrettyPrinter(width=41, compact=True)
>>> pp.pprint(stuff)
[['spam', 'eggs', 'lumberjack',
  'knights', 'ni'],
 'spam', 'eggs', 'lumberjack', 'knights',
 'ni']
>>> tup = ('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead',
... ('parrot', ('fresh fruit',)))))))
>>> pp = pprint.PrettyPrinter(depth=6)
>>> pp.pprint(tup)
('spam', ('eggs', ('lumberjack', ('knights', ('ni', ('dead', (...)))))))
```

pprint 模块还提供了一些快捷函数：

pprint.**pformat**(*object*, *indent=1*, *width=80*, *depth=None*, *, *compact=False*, *sort_dicts=True*)
object 的格式化表示作为字符串返回。*indent*, *width*, *depth*, *compact* 和 *sort_dicts* 将作为格式化形参被传入*PrettyPrinter* 构造器。

3.4 版更變: 增加了 *compact* 形参。

3.8 版更變: 增加了 *sort_dicts* 形参。

pprint.**pp**(*object*, **args*, *sort_dicts=False*, ***kwargs*)
打印 *object* 的格式化表示并附带一个换行符。如果 *sort_dicts* 为假值（默认），字典将按键的插入顺序显示，否则将按字典键排序。*args* 和 *kwargs* 将作为格式化形参被传给*pprint()*。

3.8 版新加入.

pprint.**pprint**(*object*, *stream=None*, *indent=1*, *width=80*, *depth=None*, *, *compact=False*, *sort_dicts=True*)
在 *stream* 上打打印 *object* 的格式化表示，并附带一个换行符。如果 *stream* 为 `None`，则使用 `sys.stdout`。这可以替代*print()* 函数在交互式解释器中使用以查看值（你甚至可以执行重新赋值 `print = pprint.pprint` 以在特定作用域中使用）。*indent*, *width*, *depth*, *compact* 和 *sort_dicts* 将作为格式化形参被传给*PrettyPrinter* 构造器。

3.4 版更變: 增加了 *compact* 形参。

3.8 版更變: 增加了 *sort_dicts* 形参。

```
>>> import pprint
>>> stuff = ['spam', 'eggs', 'lumberjack', 'knights', 'ni']
>>> stuff.insert(0, stuff)
>>> pprint.pprint(stuff)
[<Recursion on list with id=...>,
 'spam',
 'eggs',
 'lumberjack',
 'knights',
 'ni']
```

pprint.**isreadable**(*object*)
确定 *object* 的格式化表示是否“可读”，或是否可被用来通过*eval()* 重新构建对象的值。此函数对于递归对象总是返回 `False`。

```
>>> pprint.isreadable(stuff)
False
```

pprint.**isrecursive**(*object*)
确定 *object* 是否需要递归表示。

此外还定义了一个支持函数：

pprint.**saferepr**(*object*)
返回 *object* 的字符串表示，并为递归数据结构提供保护。如果 *object* 的表示形式公开了一个递归条目，该递归引用会被表示为 `<Recursion on typename with id=number>`。该表示因而不会进行其它格式化。

```
>>> pprint.saferepr(stuff)
"[<Recursion on list with id=...>, 'spam', 'eggs', 'lumberjack', 'knights', 'ni']"
```

8.12.1 PrettyPrinter 对象

PrettyPrinter 的实例具有下列方法：

PrettyPrinter.**pformat**(*object*)
返回 *object* 格式化表示。这会将传给*PrettyPrinter* 构造器的选项纳入考虑。

PrettyPrinter.**pprint**(*object*)
在所配置的流上打印 *object* 的格式化表示，并附加一个换行符。

下列方法提供了与同名函数相对应的实现。在实例上使用这些方法效率会更高一些，因为不需要创建新的*PrettyPrinter* 对象。

PrettyPrinter.**isreadable**(*object*)
确定对象的格式化表示是否“可读”，或者是否可使用*eval()* 重建对象值。请注意此方法对于递归对象将返回 `False`。如果设置了*PrettyPrinter* 的 *depth* 形参并且对象深度超出允许范围，此方法将返回 `False`。

PrettyPrinter.**isrecursive**(*object*)
确定对象是否需要递归表示。

此方法作为一个钩子提供，允许子类修改将对象转换为字符串的方式。默认实现使用*saferepr()* 实现的内部方式。

PrettyPrinter.**format**(*object*, *context*, *maxlevels*, *level*)
返回三个值：字符串形式的 *object* 已格式化版本，指明结果是否可读的旗标，以及指明是否检测到递归的旗标。第一个参数是要表示的对象。第二个是以对象*id()* 为键的字典，这些对象是当前表示上下文的一部分（影响 *object* 表示的直接和间接容器）；如果需要呈现一个已经在 *context* 中表示的对象，则第三个返回值应当为 `True`。对*format()* 方法的递归调用应当将容器的附加条目添加到此字典中。第三个参数 *maxlevels* 给出了对递归的请求限制；如果没有请求限制则其值将为 `0`。此参数应当不加修改地传给递归调用。第四个参数 *level* 给出于当前层级；传给递归调用的参数值应当小于当前调用的值。

8.12.2 示例

为了演示*pprint()* 函数及其形参的几种用法，让我们从 PyPI 获取关于某个项目的信息：

```
>>> import json
>>> import pprint
>>> from urllib.request import urlopen
>>> with urlopen('https://pypi.org/pypi/sampleproject/json') as resp:
...     project_info = json.load(resp)['info']
```

pprint() 以其基本形式显示了整个对象：

```
>>> pprint.pprint(project_info)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': ['Development Status :: 3 - Alpha',
                 'Intended Audience :: Developers',
                 'License :: OSI Approved :: MIT License',
                 'Programming Language :: Python :: 2',
                 'Programming Language :: Python :: 2.6',
                 'Programming Language :: Python :: 2.7',
                 'Programming Language :: Python :: 3',
                 'Programming Language :: Python :: 3.2',
                 'Programming Language :: Python :: 3.3',
                 'Programming Language :: Python :: 3.4',
                 'Topic :: Software Development :: Build Tools'],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {'last_day': -1, 'last_month': -1, 'last_week': -1},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {'Download': 'UNKNOWN',
                  'Homepage': 'https://github.com/pypa/sampleproject'},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

结果可以被限制到特定的 *depth* (更深层的内容将使用省略号):

```
>>> pprint.pprint(project_info, depth=1)
```

(下页继续)

(繼續上一頁)

```
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the project.\n'
                '\n'
                'The file should use UTF-8 encoding and be written using '
                'ReStructured Text. It\n'
                'will be used to generate the project webpage on PyPI, and '
                'should be written for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would include an overview of '
                'the project, basic\n'
                'usage examples, etc. Generally, including the project '
                'changelog in here is not\n'
                'a good idea, although a simple "What\'s New" section for the '
                'most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

此外，还可以设置建议的最大字符 *width*。如果一个对象无法被拆分，则将超出指定宽度:

```
>>> pprint.pprint(project_info, depth=1, width=60)
{'author': 'The Python Packaging Authority',
 'author_email': 'pypa-dev@googlegroups.com',
 'bugtrack_url': None,
 'classifiers': [...],
 'description': 'A sample Python project\n'
                '=======================\n'
                '\n'
                'This is the description file for the '
                'project.\n'
                '\n'
                'The file should use UTF-8 encoding and be '
```

(下页继续)

(繼續上一頁)

```
                'written using ReStructured Text. It\n'
                'will be used to generate the project '
                'webpage on PyPI, and should be written '
                'for\n'
                'that purpose.\n'
                '\n'
                'Typical contents for this file would '
                'include an overview of the project, '
                'basic\n'
                'usage examples, etc. Generally, including '
                'the project changelog in here is not\n'
                'a good idea, although a simple "What\'s '
                'New" section for the most recent version\n'
                'may be appropriate.',
 'description_content_type': None,
 'docs_url': None,
 'download_url': 'UNKNOWN',
 'downloads': {...},
 'home_page': 'https://github.com/pypa/sampleproject',
 'keywords': 'sample setuptools development',
 'license': 'MIT',
 'maintainer': None,
 'maintainer_email': None,
 'name': 'sampleproject',
 'package_url': 'https://pypi.org/project/sampleproject/',
 'platform': 'UNKNOWN',
 'project_url': 'https://pypi.org/project/sampleproject/',
 'project_urls': {...},
 'release_url': 'https://pypi.org/project/sampleproject/1.2.0/',
 'requires_dist': None,
 'requires_python': None,
 'summary': 'A sample Python project',
 'version': '1.2.0'}
```

8.13 reprlib --- 另一种 repr() 实现

源代码: Lib/reprlib.py

---

reprlib 模块提供了一种对象表示的产生方式，它会对结果字符串的大小进行限制。该方式被用于 Python 调试器，也适用于某些其他场景。

此模块提供了一个类、一个实例和一个函数：

class reprlib.**Repr**
: 该类提供了格式化服务适用于实现与内置*repr()* 相似的方法；其中附加了针对不同对象类型的大小限制，以避免生成超长的表示。

reprlib.**aRepr**
: 这是*Repr* 的一个实例，用于提供如下所述的*repr()* 函数。改变此对象的属性将会影响*repr()* 和 Python 调试器所使用的大小限制。

reprlib.**repr**(*obj*)
: 这是 aRepr 的*repr()* 方法。它会返回与同名内置函数所返回字符串相似的字符串，区别在于附带了对多数类型的大小限制。

在大小限制工具以外，此模块还提供了一个装饰器，用于检测对 `__repr__()` 的递归调用并改用一个占位符来替换。

`@reprlib.`**recursive_repr**(*fillvalue='...'*)
用于为 `__repr__()` 方法检测同一线程内部递归调用的装饰器。如果执行了递归调用，则会返回 *fillvalue*，否则执行正常的 `__repr__()` 调用。例如：

```
>>> from reprlib import recursive_repr
>>> class MyList(list):
...     @recursive_repr()
...     def __repr__(self):
...         return '<' + '|'.join(map(repr, self)) + '>'
...
>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

3.2 版新加入.

8.13.1 Repr 对象

Repr 实例对象包含一些属性可以用于为不同对象类型的表示提供大小限制，还包含一些方法可以格式化特定的对象类型。

`Repr.`**maxlevel**
创建递归表示形式的深度限制。默认为 `6`。

`Repr.`**maxdict**
`Repr.`**maxlist**
`Repr.`**maxtuple**
`Repr.`**maxset**
`Repr.`**maxfrozenset**
`Repr.`**maxdeque**
`Repr.`**maxarray**
代表命名对象类型的条目数量限制。对于*maxdict* 的默认值为 `4`，对于*maxarray* 为 `5`，对于其他则为 `6`。

`Repr.`**maxlong**
表示整数的最大字符数量。数码会从中间被丢弃。默认值为 `40`。

`Repr.`**maxstring**
表示字符串的字符数量限制。请注意字符源会使用字符串的“正常”表示形式：如果表示中需要用到转义序列，在缩短表示时它们可能会被破坏。默认值为 `30`。

`Repr.`**maxother**
此限制用于控制在*Repr* 对象上没有特定的格式化方法可用的对象类型的大小。它会以类似*maxstring* 的方式被应用。默认值为 `20`。

`Repr.`**repr**(*obj*)
内置*repr()* 的等价形式，它使用实例专属的格式化。

`Repr.`**repr1**(*obj*, *level*)
供*repr()* 使用的递归实现。此方法使用 *obj* 的类型来确定要调用哪个格式化方法，并传入 *obj* 和 *level*。类型专属的方法应当调用*repr1()* 来执行递归格式化，在递归调用中使用 `level - 1` 作为 *level* 的值。

Repr.**repr_TYPE**(*obj*, *level*)
特定类型的格式化方法会被实现为基于类型名称来命名的方法。在方法名称中，**TYPE** 会被替换为 `'_'.join(type(obj).__name__.split())`。对这些方法的分派会由*repr1()* 来处理。需要对值进行递归格式化的类型专属方法应当调用 `self.repr1(subobj, level - 1)`。

8.13.2 子类化 Repr 对象

通过*Repr.repr1()* 使用动态分派允许*Repr* 的子类添加对额外内置对象类型的支持，或是修改对已支持类型的处理。这个例子演示了如何添加对文件对象的特殊支持:

```
import reprlib
import sys

class MyRepr(reprlib.Repr):

    def repr_TextIOWrapper(self, obj, level):
        if obj.name in {'<stdin>', '<stdout>', '<stderr>'}:
            return obj.name
        return repr(obj)

aRepr = MyRepr()
print(aRepr.repr(sys.stdin))          # prints '<stdin>'
```

8.14 enum --- 对枚举的支持

3.4 版新加入.

源代码： Lib/enum.py

枚举是与多个唯一常量值绑定的一组符号名（即成员）。枚举中的成员可以进行身份比较，并且枚举自身也可迭代。

備🄴: 枚举成员名称的大小写

枚举表示的是常量，因此，建议枚举成员名称使用大写字母，本篇的示例将用此种风格。

8.14.1 模块内容

本模块定义了四个枚举类，用来定义名称与值的唯一组合: *Enum*、*IntEnum*、*Flag* 和*IntFlag*。此外，还定义了一个装饰器，*unique()*，和一个辅助类，*auto*。

class enum.**Enum**
创建枚举常量的基类。如需了解另一种构建语法，请参阅*Functional API* 小节。

class enum.**IntEnum**
创建*int* 子类枚举常量的基类。

class enum.**IntFlag**
创建可与位运算符搭配使用，又不失去*IntFlag* 成员资格的枚举常量的基类。*IntFlag* 成员也是*int* 的子类。

class enum.**Flag**
创建可与位运算符搭配使用，又不会失去*Flag* 成员资格的枚举常量的基类。

enum.**unique**()
确保一个名称只绑定一个值的 Enum 类装饰器。

class enum.**auto**
以合适的值代替 Enum 成员的实例。初始值默认从 1 开始。

3.6 版新加入: Flag, IntFlag, auto

8.14.2 创建 Enum

枚举是由 class 句法创建的，这种方式易读、易写。另一种创建方法请参阅*Functional API*。*Enum* 以如下定义枚举：

```
>>> from enum import Enum
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...
```

備🄴: Enum 成员值

成员值可以是*int*、*str* 等。若无需设定确切值，*auto* 实例可以自动为成员分配合适的值。将*auto* 与其他值混用时必须要慎重。

備🄴: 命名法

- 类 Color 是 *枚举*（或称为 *enum* ）。
- Color.RED、Color.GREEN 等属性是 *枚举成员*（或 *enum 成员*），也是常量。
- 枚举成员具有 *名称*和 *值* (例如 Color.RED 的名称为 RED，Color.BLUE 的值为 3 等等)

備🄴: 虽然 Enum 由 class 语法创建，但 Enum 并不是常规的 Python 类。详见*How are Enums different?*。

枚举成员的字符串表现形式更容易理解：

```
>>> print(Color.RED)
Color.RED
```

同时，它的 repr 包含更多信息：

```
>>> print(repr(Color.RED))
<Color.RED: 1>
```

枚举成员的 类型就是它所属的枚举：

```
>>> type(Color.RED)
<enum 'Color'>
>>> isinstance(Color.GREEN, Color)
```

(下页继续)

(繼續上一頁)

```
True
>>>
```

Enum 成员还包含 name 属性：

```
>>> print(Color.RED.name)
RED
```

枚举按定义的顺序进行迭代：

```
>>> class Shake(Enum):
...     VANILLA = 7
...     CHOCOLATE = 4
...     COOKIES = 9
...     MINT = 3
...
>>> for shake in Shake:
...     print(shake)
...
Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT
```

枚举成员可哈希，可用于字典和集合：

```
>>> apples = {}
>>> apples[Color.RED] = 'red delicious'
>>> apples[Color.GREEN] = 'granny smith'
>>> apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
True
```

8.14.3 枚举成员及其属性的编程访问

有时，要在程序中访问枚举成员（如，开发时不知道颜色的确切值，`Color.RED` 不适用的情况）。`Enum` 支持如下访问方式：

```
>>> Color(1)
<Color.RED: 1>
>>> Color(3)
<Color.BLUE: 3>
```

用 *name* 访问枚举成员时，可使用项目名称：

```
>>> Color['RED']
<Color.RED: 1>
>>> Color['GREEN']
<Color.GREEN: 2>
```

可访问枚举成员的 `name` 或 `value`：

```
>>> member = Color.RED
>>> member.name
'RED'
```

(下页继续)

(繼續上一頁)

```
>>> member.value
1
```

8.14.4 重复的枚举成员和值

两个枚举成员的名称不能相同：

```
>>> class Shape(Enum):
...     SQUARE = 2
...     SQUARE = 3
...
Traceback (most recent call last):
...
TypeError: Attempted to reuse key: 'SQUARE'
```

但是，两个枚举成员可以有相同的值。假设，成员 A 和 B 的值相同（先定义的是 A），则 B 是 A 的别名。按值查找 A 和 B 的值返回的是 A。按名称查找 B，返回的也是 A：

```
>>> class Shape(Enum):
...     SQUARE = 2
...     DIAMOND = 1
...     CIRCLE = 3
...     ALIAS_FOR_SQUARE = 2
...
>>> Shape.SQUARE
<Shape.SQUARE: 2>
>>> Shape.ALIAS_FOR_SQUARE
<Shape.SQUARE: 2>
>>> Shape(2)
<Shape.SQUARE: 2>
```

備🄵： 不支持创建与已定义属性（其他成员、方法等）同名的成员，也不支持创建与现有成员同名的属性。

8.14.5 确保唯一枚举值

默认情况下，枚举允许多个名称作为一个值的别名。如需禁用此行为，下述装饰器可以确保枚举中的值仅能只用一次：

@enum.unique

专用于枚举的 class 装饰器。它会搜索一个枚举的 __members__ 并收集所找到的任何别名；只要找到任何别名就会引发*ValueError* 并附带相关细节信息：

```
>>> from enum import Enum, unique
>>> @unique
... class Mistake(Enum):
...     ONE = 1
...     TWO = 2
...     THREE = 3
...     FOUR = 3
...
```

(下页继续)

(繼續上一頁)

```
Traceback (most recent call last):
...
ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

8.14.6 使用自动设定的值

如果确切的值不重要，你可以使用*auto*:

```
>>> from enum import Enum, auto
>>> class Color(Enum):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

值将由 _generate_next_value_() 来选择，该函数可以被重载:

```
>>> class AutoName(Enum):
...     def _generate_next_value_(name, start, count, last_values):
...         return name
...
>>> class Ordinal(AutoName):
...     NORTH = auto()
...     SOUTH = auto()
...     EAST = auto()
...     WEST = auto()
...
>>> list(Ordinal)
[<Ordinal.NORTH: 'NORTH'>, <Ordinal.SOUTH: 'SOUTH'>, <Ordinal.EAST: 'EAST'>, <Ordinal.
↪WEST: 'WEST'>]
```

備F: 默认 _generate_next_value_() 方法的目标是提供所给出的最后一个*int* 所在序列的下一个*int*，但这种行为方式属于实现细节并且可能发生改变。

備F: _generate_next_value_() 方法定义必须在任何其他成员之前。

8.14.7 迭代

对枚举成员的迭代不会给出别名:

```
>>> list(Shape)
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
```

特殊属性 __members__ 是一个从名称到成员的只读有序映射。它包含枚举中定义的所有名称，包括别名:

```
>>> for name, member in Shape.__members__.items():
...     name, member
```

(下页继续)

(繼續上一頁)

```
...
('SQUARE', <Shape.SQUARE: 2>)
('DIAMOND', <Shape.DIAMOND: 1>)
('CIRCLE', <Shape.CIRCLE: 3>)
('ALIAS_FOR_SQUARE', <Shape.SQUARE: 2>)
```

__members__ 属性可被用于对枚举成员进行详细的程序化访问。例如，找出所有别名:

```
>>> [name for name, member in Shape.__members__.items() if member.name != name]
['ALIAS_FOR_SQUARE']
```

8.14.8 比较

枚举成员是按标识号进行比较的:

```
>>> Color.RED is Color.RED
True
>>> Color.RED is Color.BLUE
False
>>> Color.RED is not Color.BLUE
True
```

枚举值之间的排序比较 不被支持。Enum 成员不属于整数 (另请参阅下文的*IntEnum*):

```
>>> Color.RED < Color.BLUE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Color' and 'Color'
```

相等比较的定义如下:

```
>>> Color.BLUE == Color.RED
False
>>> Color.BLUE != Color.RED
True
>>> Color.BLUE == Color.BLUE
True
```

与非枚举值的比较将总是不相等（同样地，*IntEnum* 被显式设计成不同的行为，参见下文）：

```
>>> Color.BLUE == 2
False
```

8.14.9 允许的枚举成员和属性

以上示例使用整数作为枚举值。使用整数相当简洁方便（并由*Functional API* 默认提供），但并不强制要求使用。在大部分用例中，开发者都关心枚举的实际值是什么。但如果值 确实重要，则枚举可以使用任意的值。

枚举属于 Python 的类，并可具有普通方法和特殊方法。如果我们有这样一个枚举:

```
>>> class Mood(Enum):
...     FUNKY = 1
...     HAPPY = 3
```

(下页继续)

(繼續上一頁)

```
...
...     def describe(self):
...         # self is the member here
...         return self.name, self.value
...
...     def __str__(self):
...         return 'my custom str! {0}'.format(self.value)
...
...     @classmethod
...     def favorite_mood(cls):
...         # cls here is the enumeration
...         return cls.HAPPY
...
```

那么:

```
>>> Mood.favorite_mood()
<Mood.HAPPY: 3>
>>> Mood.HAPPY.describe()
('HAPPY', 3)
>>> str(Mood.FUNKY)
'my custom str! 1'
```

对于允许内容的规则如下：以单下划线开头和结尾的名称是由枚举保留而不可使用；在枚举中定义的所有其他属性将成为该枚举的成员，例外项则包括特殊方法成员 (`__str__()`, `__add__()` 等)，描述符 (方法也属于描述符) 以及在 `_ignore_` 中列出的变量名。

注意：如果你的枚举定义了 `__new__()` 和/或 `__init__()` 那么给予枚举成员的任何值都会被传入这些方法。请参阅示例*Planet*。

8.14.10 受限的 Enum 子类化

一个新的*Enum*类必须基于一个 Enum 类，至多一个实体数据类型以及出于实际需要的任意多个基于*object*的 mixin 类。这些基类的顺序为:

```
class EnumName([mix-in, ...,] [data-type,] base-enum):
    pass
```

另外，仅当一个枚举未定义任何成员时才允许子类化该枚举。因此禁止这样的写法:

```
>>> class MoreColor(Color):
...     PINK = 17
...
Traceback (most recent call last):
...
TypeError: Cannot extend enumerations
```

但是允许这样的写法:

```
>>> class Foo(Enum):
...     def some_behavior(self):
...         pass
...
>>> class Bar(Foo):
...     HAPPY = 1
```

(下页继续)

(繼續上一頁)

```
...     SAD = 2
...
```

允许子类化定义了成员的枚举将会导致违反类型与实例的某些重要的不可变规则。在另一方面，允许在一组枚举之间共享某些通用行为也是有意义的。(请参阅示例*OrderedEnum* 。)

8.14.11 封存

枚举可以被封存与解封:

```
>>> from test.test_enum import Fruit
>>> from pickle import dumps, loads
>>> Fruit.TOMATO is loads(dumps(Fruit.TOMATO))
True
```

封存的常规限制同样适用：可封存枚举必须在模块的最高层级中定义，因为解封操作要求它们可以从该模块导入。

備🄵: 使用 pickle 协议版本 4 可以方便地封存嵌套在其他类中的枚举。

通过在枚举类中定义*__reduce_ex__()* 可以对 Enum 成员的封存/解封方式进行修改。

8.14.12 功能性 API

Enum 类属于可调用对象，它提供了以下功能性 API:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG')
>>> Animal
<enum 'Animal'>
>>> Animal.ANT
<Animal.ANT: 1>
>>> Animal.ANT.value
1
>>> list(Animal)
[<Animal.ANT: 1>, <Animal.BEE: 2>, <Animal.CAT: 3>, <Animal.DOG: 4>]
```

该 API 的主义类似于*namedtuple*。调用*Enum* 的第一个参数是枚举的名称。

第二个参数是枚举成员名称的 *来源*。它可以是一个用空格分隔的名称字符串、名称序列、键/值对 2 元组的序列，或者名称到值的映射（例如字典）。最后两种选项使得可以为枚举任意赋值；其他选项会自动以从 1 开始递增的整数赋值（使用 `start` 形参可指定不同的起始值）。返回值是一个派生自*Enum* 的新类。换句话说，以上对 `Animal` 的赋值就等价于:

```
>>> class Animal(Enum):
...     ANT = 1
...     BEE = 2
...     CAT = 3
...     DOG = 4
...
```

默认以 `1` 而以 `0` 作为起始数值的原因在于 `0` 的布尔值为 `False`，但所有枚举成员都应被求值为 `True`。

对使用功能性 API 创建的枚举执行封存可能会很麻烦，因为要使用帧堆栈的实现细节来尝试并找出枚举是在哪个模块中创建的（例如当你使用了另一个模块中的工具函数就可能失败，在 IronPython 或 Jython 上也可能无效）。解决办法是显式地指定模块名称，如下所示:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', module=__name__)
```

警告: 如果未提供 `module`，且 Enum 无法确定是哪个模块，新的 Enum 成员将不可被解封；为了让错误尽量靠近源头，封存将被禁用。

新的 pickle 协议版本 4 在某些情况下同样依赖于 *__qualname__* 被设为特定位置以便 pickle 能够找到相应的类。例如，类是否存在于全局作用域的 SomeData 类中:

```
>>> Animal = Enum('Animal', 'ANT BEE CAT DOG', qualname='SomeData.Animal')
```

完整的签名为:

```
Enum(value='NewEnumName', names=<...>, *, module='...', qualname='...', type=<mixed-
↪in class>, start=1)
```

值 将被新 Enum 类将记录为其名称的数据。

名称 Enum 的成员。这可以是一个空格或逗号分隔的字符串 (起始值将为 1，除非另行指定):

```
'RED GREEN BLUE' | 'RED,GREEN,BLUE' | 'RED, GREEN, BLUE'
```

或是一个名称的迭代器:

```
['RED', 'GREEN', 'BLUE']
```

或是一个 (名称, 值) 对的迭代器:

```
[('CYAN', 4), ('MAGENTA', 5), ('YELLOW', 6)]
```

或是一个映射:

```
{'CHARTREUSE': 7, 'SEA_GREEN': 11, 'ROSEMARY': 42}
```

模块 新 Enum 类所在模块的名称。

qualname 新 Enum 类在模块中的具体位置。

类型 要加入新 Enum 类的类型。

start 当只传入名称时要使用的起始数值。

3.5 版更變: 增加了 *start* 形参。

8.14.13 派生的枚举

IntEnum

所提供的第一个变种*Enum* 同时也是*int* 的一个子类。*IntEnum* 的成员可与整数进行比较；通过扩展，不同类型的整数枚举也可以相互进行比较:

```
>>> from enum import IntEnum
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Request(IntEnum):
...     POST = 1
...     GET = 2
...
>>> Shape == 1
False
>>> Shape.CIRCLE == 1
True
>>> Shape.CIRCLE == Request.POST
True
```

不过，它们仍然不可与标准*Enum* 枚举进行比较:

```
>>> class Shape(IntEnum):
...     CIRCLE = 1
...     SQUARE = 2
...
>>> class Color(Enum):
...     RED = 1
...     GREEN = 2
...
>>> Shape.CIRCLE == Color.RED
False
```

IntEnum 值在其他方面的行为都如你预期的一样类似于整数:

```
>>> int(Shape.CIRCLE)
1
>>> ['a', 'b', 'c'][Shape.CIRCLE]
'b'
>>> [i for i in range(Shape.SQUARE)]
[0, 1]
```

IntFlag

所提供的下一个*Enum* 的变种*IntFlag* 同样是基于*int* 的，不同之处在于*IntFlag* 成员可使用按位运算符 (&, |, ^, ~) 进行组合且结果仍然为*IntFlag* 成员。如果，正如名称所表明的，*IntFlag* 成员同时也是*int* 的子类，并能在任何使用*int* 的场合被使用。*IntFlag* 成员进行除按位运算以外的其他运算都将导致失去*IntFlag* 成员资格。

3.6 版新加入.

示例*IntFlag* 类:

```
>>> from enum import IntFlag
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...
>>> Perm.R | Perm.W
<Perm.R|W: 6>
>>> Perm.R + Perm.W
6
>>> RW = Perm.R | Perm.W
>>> Perm.R in RW
True
```

对于组合同样可以进行命名:

```
>>> class Perm(IntFlag):
...     R = 4
...     W = 2
...     X = 1
...     RWX = 7
>>> Perm.RWX
<Perm.RWX: 7>
>>> ~Perm.RWX
<Perm.-8: -8>
```

IntFlag 和*Enum* 的另一个重要区别在于如果没有设置任何旗标（值为 0），则其布尔值为*False*:

```
>>> Perm.R & Perm.X
<Perm.0: 0>
>>> bool(Perm.R & Perm.X)
False
```

由于*IntFlag* 成员同时也是*int* 的子类，因此它们可以相互组合:

```
>>> Perm.X | 8
<Perm.8|X: 9>
```

Flag

最后一个变种是*Flag*。与*IntFlag* 类似，*Flag* 成员可使用按位运算符 (&, |, ^, ~) 进行组合，与*IntFlag* 不同的是它们不可与任何其它*Flag* 枚举或*int* 进行组合或比较。虽然可以直接指定值，但推荐使用*auto* 作为值以便让*Flag* 选择适当的值。

3.6 版新加入.

与*IntFlag* 类似，如果*Flag* 成员的某种组合导致没有设置任何旗标，则其布尔值为*False*:

```
>>> from enum import Flag, auto
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.RED & Color.GREEN
<Color.0: 0>
```

(下页继续)

(繼續上一頁)

```
>>> bool(Color.RED & Color.GREEN)
False
```

单个旗标的值应当为二的乘方 (1, 2, 4, 8, ...)，旗标的组合则无此限制:

```
>>> class Color(Flag):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...     WHITE = RED | BLUE | GREEN
...
>>> Color.WHITE
<Color.WHITE: 7>
```

对"no flags set" 条件指定一个名称并不会改变其布尔值:

```
>>> class Color(Flag):
...     BLACK = 0
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.BLACK
<Color.BLACK: 0>
>>> bool(Color.BLACK)
False
```

備𝔽: 对于大多数新代码，强烈推荐使用*Enum* 和*Flag*，因为*IntEnum* 和*IntFlag* 打破了枚举的某些语义约定（例如可以同整数进行比较，并因而导致此行为被传递给其他无关的枚举）。*IntEnum* 和*IntFlag* 的使用应当仅限于*Enum* 和*Flag* 无法使用的场合；例如，当使用枚举替代整数常量时，或是与其他系统进行交互操作时。

其他事项

虽然*IntEnum* 是*enum* 模块的一部分，但要独立实现也应该相当容易:

```
class IntEnum(int, Enum):
    pass
```

这里演示了如何定义类似的派生枚举；例如一个混合了*str* 而不是*int* 的 `StrEnum`。

几条规则:

1. 当子类化*Enum* 时，在基类序列中的混合类型必须出现于*Enum* 本身之前，如以上*IntEnum* 的例子所示。
2. 虽然*Enum* 可以拥有任意类型的成员，不过一旦你混合了附加类型，则所有成员必须为相应类型的值，如在上面的例子中即为*int*。此限制不适用于仅添加方法而未指定另一数据类型的混合类。
3. 当混合了另一数据类型时，`value` 属性会 不同于枚举成员自身，但它们仍保持等价且比较结果也相等。
4. %-style formatting: *%s* 和 *%r* 会分别调用*Enum* 类的 `__str__()` 和 `__repr__()`；其他代码 (例如表示 IntEnum 的 *%i* 或 *%h*) 会将枚举成员视为对应的混合类型。

5. 格式化字符串字面值, *str.format()* 和*format()* 将使用混合类型的 __format__() 除非在子类中重载了 __str__() 或 __format__()，在这种情况下将使用被重载的方法或*Enum* 的方法。请使用!s 和!r 格式代码来强制使用*Enum* 类的 __str__() 和 __repr__() 方法。

8.14.14 何时使用 __new__() 与 __init__()

当你想要定制*Enum* 成员的实际值时必须使用 __new__()。任何其他修改可以用 __new__() 也可以用 __init__()，应优先使用 __init__()。

举例来说，如果你要向构造器传入多个条目，但只希望将其中一个作为值:

```
>>> class Coordinate(bytes, Enum):
...     """
...     Coordinate with binary codes that can be indexed by the int code.
...     """
...     def __new__(cls, value, label, unit):
...         obj = bytes.__new__(cls, [value])
...         obj._value_ = value
...         obj.label = label
...         obj.unit = unit
...         return obj
...     PX = (0, 'P.X', 'km')
...     PY = (1, 'P.Y', 'km')
...     VX = (2, 'V.X', 'km/s')
...     VY = (3, 'V.Y', 'km/s')
...

>>> print(Coordinate['PY'])
Coordinate.PY

>>> print(Coordinate(3))
Coordinate.VY
```

8.14.15 有趣的示例

虽然*Enum*, *IntEnum*, *IntFlag* 和*Flag* 预期可覆盖大多数应用场景，但它们无法覆盖全部。这里有一些不同类型枚举的方案，它们可以被直接使用，或是作为自行创建的参考示例。

省略值

在许多应用场景中人们都不关心枚举的实际值是什么。有几个方式可以定义此种类型的简单枚举:

- 使用*auto* 的实例作为值
- 使用*object* 的实例作为值
- 使用描述性的字符串作为值
- 使用元组作为值并用自定义的 __new__() 以一个*int* 值来替代该元组

使用以上任何一种方法均可向用户指明值并不重要，并且使人能够添加、移除或重排序成员而不必改变其余成员的数值。

无论你选择何种方法，你都应当提供一个*repr()* 并且它也需要隐藏（不重要的）值:

```
>>> class NoValue(Enum):
...     def __repr__(self):
...         return '<%s.%s>' % (self.__class__.__name__, self.name)
...
```

使用 auto

使用*auto* 的形式如下:

```
>>> class Color(NoValue):
...     RED = auto()
...     BLUE = auto()
...     GREEN = auto()
...
>>> Color.GREEN
<Color.GREEN>
```

使用 object

使用*object* 的形式如下:

```
>>> class Color(NoValue):
...     RED = object()
...     GREEN = object()
...     BLUE = object()
...
>>> Color.GREEN
<Color.GREEN>
```

使用描述性字符串

使用字符串作为值的形式如下:

```
>>> class Color(NoValue):
...     RED = 'stop'
...     GREEN = 'go'
...     BLUE = 'too fast!'
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
'go'
```

使用自定义的 __new__()

使用自动编号 __new__() 的形式如下:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls):
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
>>> class Color(AutoNumber):
...     RED = ()
...     GREEN = ()
...     BLUE = ()
...
>>> Color.GREEN
<Color.GREEN>
>>> Color.GREEN.value
2
```

要实现更通用的 AutoNumber，请添加 *args 到签名中:

```
>>> class AutoNumber(NoValue):
...     def __new__(cls, *args):      # this is the only change from above
...         value = len(cls.__members__) + 1
...         obj = object.__new__(cls)
...         obj._value_ = value
...         return obj
...
```

这样当你从 AutoNumber 继承时你将可以编写你自己的 __init__ 来处理任何附加参数:

```
>>> class Swatch(AutoNumber):
...     def __init__(self, pantone='unknown'):
...         self.pantone = pantone
...     AUBURN = '3497'
...     SEA_GREEN = '1246'
...     BLEACHED_CORAL = () # New color, no Pantone code yet!
...
>>> Swatch.SEA_GREEN
<Swatch.SEA_GREEN: 2>
>>> Swatch.SEA_GREEN.pantone
'1246'
>>> Swatch.BLEACHED_CORAL.pantone
'unknown'
```

備🄴: 如果定义了 __new__() 则它会在创建 Enum 成员期间被使用；随后它将被 Enum 的 __new__() 所替换，该方法会在类创建后被用来查找现有成员。

OrderedEnum

一个有序枚举，它不是基于*IntEnum*，因此保持了正常的*Enum*不变特性（例如不可与其他枚举进行比较）：

```
>>> class OrderedEnum(Enum):
...     def __ge__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value >= other.value
...         return NotImplemented
...     def __gt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value > other.value
...         return NotImplemented
...     def __le__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value <= other.value
...         return NotImplemented
...     def __lt__(self, other):
...         if self.__class__ is other.__class__:
...             return self.value < other.value
...         return NotImplemented
...
>>> class Grade(OrderedEnum):
...     A = 5
...     B = 4
...     C = 3
...     D = 2
...     F = 1
...
>>> Grade.C < Grade.A
True
```

DuplicateFreeEnum

如果发现重复的成员名称则将引发错误而不是创建别名:

```
>>> class DuplicateFreeEnum(Enum):
...     def __init__(self, *args):
...         cls = self.__class__
...         if any(self.value == e.value for e in cls):
...             a = self.name
...             e = cls(self.value).name
...             raise ValueError(
...                 "aliases not allowed in DuplicateFreeEnum:  %r --> %r"
...                 % (a, e))
...
>>> class Color(DuplicateFreeEnum):
...     RED = 1
...     GREEN = 2
...     BLUE = 3
...     GRENE = 2
...
Traceback (most recent call last):
...
ValueError: aliases not allowed in DuplicateFreeEnum:  'GRENE' --> 'GREEN'
```

備𝔽: 这个例子适用于子类化 Enum 来添加或改变禁用别名以及其他行为。如果需要的改变只是禁用别名，也可以选择使用*unique()* 装饰器。

Planet

如果定义了 __new__() 或 __init__() 则枚举成员的值将被传给这些方法:

```
>>> class Planet(Enum):
...     MERCURY = (3.303e+23, 2.4397e6)
...     VENUS   = (4.869e+24, 6.0518e6)
...     EARTH   = (5.976e+24, 6.37814e6)
...     MARS    = (6.421e+23, 3.3972e6)
...     JUPITER = (1.9e+27,   7.1492e7)
...     SATURN  = (5.688e+26, 6.0268e7)
...     URANUS  = (8.686e+25, 2.5559e7)
...     NEPTUNE = (1.024e+26, 2.4746e7)
...     def __init__(self, mass, radius):
...         self.mass = mass       # in kilograms
...         self.radius = radius   # in meters
...     @property
...     def surface_gravity(self):
...         # universal gravitational constant  (m3 kg-1 s-2)
...         G = 6.67300E-11
...         return G * self.mass / (self.radius * self.radius)
...
>>> Planet.EARTH.value
(5.976e+24, 6378140.0)
>>> Planet.EARTH.surface_gravity
9.802652743337129
```

TimePeriod

一个演示如何使用 _ignore_ 属性的例子:

```
>>> from datetime import timedelta
>>> class Period(timedelta, Enum):
...     "different lengths of time"
...     _ignore_ = 'Period i'
...     Period = vars()
...     for i in range(367):
...         Period['day_%d' % i] = i
...
>>> list(Period)[:2]
[<Period.day_0: datetime.timedelta(0)>, <Period.day_1: datetime.timedelta(days=1)>]
>>> list(Period)[-2:]
[<Period.day_365: datetime.timedelta(days=365)>, <Period.day_366: datetime.
↪timedelta(days=366)>]
```

8.14.16 各种枚举有何区别？

枚举具有自定义的元类，它会影响所派生枚举类及其实例（成员）的各个方面。

枚举类

EnumMeta 元类负责提供 __contains__(), __dir__(), __iter__() 及其他方法以允许用户通过*Enum*类来完成一般类做不到的事情，例如 *list(Color)* 或 *some_enum_var in Color*。EnumMeta 会负责确保最终*Enum*类中的各种其他方法是正确的 (例如 __new__(), *__getnewargs__()*, __str__() 和 __repr__())。

枚举成员（即实例）

有关枚举成员最有趣的特点是它们都是单例对象。EnumMeta 会在创建*Enum*类本身时将它们全部创建完成，然后准备好一个自定义的 __new__()，通过只返回现有的成员实例来确保不会再实例化新的对象。

细节要点

支持的 __dunder__ 名称

__members__ 是一个 member_name:member 条目的只读有序映射。它只在类上可用。

如果指定了 __new__()，它必须创建并返回枚举成员；相应地设定成员的 _value_ 也是一个很好的主意。一旦所有成员都创建完成它就不会再被使用。

支持的 _sunder_ 名称

- _name_ -- 成员的名称
- _value_ -- 成员的值；可以在 __new__ 中设置 / 修改
- _missing_ -- 当未发现某个值时所使用的查找函数；可被重载
- _ignore_ -- 一个名称列表，可以为*list*或*str*，它不会被转化为成员，并将从最终类中被移除
- _order_ -- 用于 Python 2/3 代码以确保成员顺序一致（类属性，在类创建期间会被移除）
- _generate_next_value_ -- 用于*Functional API*并通过*auto*为枚举成员获取适当的值；可被重载

3.6 版新加入: _missing_, _order_, _generate_next_value_

3.7 版新加入: _ignore_

用来帮助 Python 2 / Python 3 代码保持同步提供 _order_ 属性。它将与枚举的实际顺序进行对照检查，如果两者不匹配则会引发错误:

```
>>> class Color(Enum):
...     _order_ = 'RED GREEN BLUE'
...     RED = 1
...     BLUE = 3
...     GREEN = 2
...
Traceback (most recent call last):
...
TypeError: member order does not match _order_
```

備𝔽: 在 Python 2 代码中 _order_ 属性是必须的，因为定义顺序在被记录之前就会丢失。

_Private__names

Private names will be normal attributes in Python 3.11 instead of either an error or a member (depending on if the name ends with an underscore). Using these names in 3.9 and 3.10 will issue a *DeprecationWarning*.

Enum 成员类型

Enum 成员是其*Enum* 类的实例，一般通过 EnumClass.member 的形式来访问。在特定情况下它们也可通过 EnumClass.member.member 的形式来访问，但你绝对不应这样做，因为查找可能失败，或者更糟糕地返回你所查找的*Enum* 成员以外的对象（这也是成员应使用全大写名称的另一个好理由）：

```
>>> class FieldTypes(Enum):
...     name = 0
...     value = 1
...     size = 2
...
>>> FieldTypes.value.size
<FieldTypes.size: 2>
>>> FieldTypes.size.value
2
```

備𝔽: This behavior is deprecated and will be removed in 3.11.

3.5 版更變.

Enum 类和成员的布尔值

混合了非*Enum* 类型（例如*int*, *str* 等）的*Enum* 成员会按所混合类型的规则被求值；在其他情况下，所有成员都将被求值为*True*。要使你的自定义 Enum 的布尔值取决于成员的值，请在你的类中添加以下代码:

```
def __bool__(self):
    return bool(self.value)
```

Enum 类总是会被求值为*True*。

带有方法的 Enum 类

如果你为你的*Enum* 子类添加了额外的方法，如同上述的*Planet* 类一样，这些方法将在对成员执行*dir()* 时显示出来，但对类执行时则不会显示:

```
>>> dir(Planet)
['EARTH', 'JUPITER', 'MARS', 'MERCURY', 'NEPTUNE', 'SATURN', 'URANUS', 'VENUS', '__
↪class__', '__doc__', '__members__', '__module__']
>>> dir(Planet.EARTH)
['__class__', '__doc__', '__module__', 'name', 'surface_gravity', 'value']
```

组合 Flag 的成员

如果 Flag 成员的某种组合未被命名，则*repr()* 将包含所有已命名的旗标和值中所有已命名的旗标组合:

```
>>> class Color(Flag):
...     RED = auto()
...     GREEN = auto()
...     BLUE = auto()
...     MAGENTA = RED | BLUE
...     YELLOW = RED | GREEN
...     CYAN = GREEN | BLUE
...
>>> Color(3)  # named combination
<Color.YELLOW: 3>
>>> Color(7)      # not named combination
<Color.CYAN|MAGENTA|BLUE|YELLOW|GREEN|RED: 7>
```

備🄵: In 3.11 unnamed combinations of flags will only produce the canonical flag members (aka single-value flags). So `Color(7)` would produce something like `<Color.BLUE|GREEN|RED: 7>`.

8.15 graphlib --- 操作类似图的结构的功能

源代码: Lib/graphlib.py

class graphlib.**TopologicalSorter**(*graph=None*)

提供以拓扑方式对可哈希节点的图进行排序的功能。

拓扑排序是指图中顶点的线性排序，使得对于每条从顶点 u 到顶点 v 的有向边 u -> v，顶点 u 都排在顶点 v 之前。例如，图的顶点可以代表要执行的任务，而边代表某一个任务必须在另一个任务之前执行的约束条件；在这个例子中，拓扑排序只是任务的有效序列。完全拓扑排序当且仅当图不包含有向环，也就是说为有向无环图时，完全拓扑排序才是可能的。

如果提供了可选的 *graph* 参数则它必须为一个表示有向无环图的字典，其中的键为节点而值为包含图中该节点的所有上级节点（即具有指向键中的值的边的节点）的可迭代对象。额外的节点可以使用*add()* 方法添加到图中。

在通常情况下，对给定的图执行排序所需的步骤如下:

- 通过可选的初始图创建一个*TopologicalSorter* 的实例。
- 添加额外的节点到图中。
- 在图上调用*prepare()*。
- 当*is_active()* 为 True 时，迭代*get_ready()* 所返回的节点并加以处理。完成处理后在每个节点上调用*done()*。

在只需要对图中的节点进行立即排序并且不涉及并行性的情况下，可以直接使用便捷方法*TopologicalSorter.static_order()*:

```
>>> graph = {"D": {"B", "C"}, "C": {"A"}, "B": {"A"}}
>>> ts = TopologicalSorter(graph)
>>> tuple(ts.static_order())
('A', 'C', 'B', 'D')
```

这个类被设计用来在节点就绪时方便地支持对其并行处理。例如:

```
topological_sorter = TopologicalSorter()

# Add nodes to 'topological_sorter'...

topological_sorter.prepare()
while topological_sorter.is_active():
    for node in topological_sorter.get_ready():
        # Worker threads or processes take nodes to work on off the
        # 'task_queue' queue.
        task_queue.put(node)

    # When the work for a node is done, workers put the node in
    # 'finalized_tasks_queue' so we can get more nodes to work on.
    # The definition of 'is_active()' guarantees that, at this point, at
    # least one node has been placed on 'task_queue' that hasn't yet
    # been passed to 'done()', so this blocking 'get()' must (eventually)
    # succeed.  After calling 'done()', we loop back to call 'get_ready()'
    # again, so put newly freed nodes on 'task_queue' as soon as
    # logically possible.
    node = finalized_tasks_queue.get()
    topological_sorter.done(node)
```

add(*node*, **predecessors*)
将一个新节点及其上级节点添加到图中。*node* 以及 *predecessors* 中的所有元素都必须为可哈希对象。

如果附带相同的节点参数多次调用，则依赖项的集合将为所有被传入依赖项的并集。

可以添加不带依赖项的节点 (即不提供 *predecessors*) 或者重复提供依赖项。如果有先前未提供的节点包含在 *predecessors* 中则它将被自动添加到图中并且不带自己的上级节点。

如果在*prepare()* 之后被调用则会引发*ValueError*。

prepare()
将图标记为已完成并检查图中是否存在环。如何检测到任何环，则将引发*CycleError*，但*get_ready()* 仍可被用来获取尽可能多的节点直到环阻塞了操作过程。在调用此函数后，图将无法再修改，因此不能再使用*add()* 添加更多的节点。

is_active()
如果可以取得更多进展则返回 `True`，否则返回 `False`。如果环没有阻塞操作，并且还存在尚未被*TopologicalSorter.get_ready()* 返回的已就绪节点或者已标记为*TopologicalSorter.done()* 的节点数量少于已被*TopologicalSorter.get_ready()* 所返回的节点数量则还可以取得进展。

该类的 `__bool__()` 方法要使用此函数，因此除了:

```
if ts.is_active():
    ...
```

可能会简单地执行:

```
if ts:
    ...
```

如果之前未调用*prepare()* 就调用此函数则会引发*ValueError*。

done(**nodes*)
将*TopologicalSorter.get_ready()* 所返回的节点集合标记为已处理，解除对 *nodes* 中每个

节点的后续节点的阻塞以便在将来通过对*TopologicalSorter.get_ready()* 的调用来返回它们。

如果 *nodes* 中的任何节点已经被之前对该方法的调用标记为已处理或者如果未通过使用*TopologicalSorter.add()* 将一个节点添加到图中，如果未调用*prepare()* 即调用此方法或者如果节点尚未被*get_ready()* 所返回则将引发*ValueError*。

get_ready()
返回由所有已就绪节点组成的 tuple。初始状态下它将返回所有不带上级节点的节点，并且一旦通过调用*TopologicalSorter.done()* 将它们标记为已处理，之后的调用将返回所有上级节点已被处理的新节点。一旦无法再取得进展，则会返回空元组。

如果之前未调用*prepare()* 就调用此函数则会引发*ValueError*。

static_order()
返回一个迭代器，它将按照拓扑顺序来迭代所有节点。当使用此方法时，*prepare()* 和*done()* 不应被调用。此方法等价于:

```
def static_order(self):
    self.prepare()
    while self.is_active():
        node_group = self.get_ready()
        yield from node_group
        self.done(*node_group)
```

所返回的特定顺序可能取决于条目被插入图中的顺序。例如:

```
>>> ts = TopologicalSorter()
>>> ts.add(3, 2, 1)
>>> ts.add(1, 0)
>>> print([*ts.static_order()])
[2, 0, 1, 3]

>>> ts2 = TopologicalSorter()
>>> ts2.add(1, 0)
>>> ts2.add(3, 2, 1)
>>> print([*ts2.static_order()])
[0, 2, 1, 3]
```

这是由于实际上"0" 和"2" 在图中的级别相同（它们将在对*get_ready()* 的同一次调用中被返回）并且它们之间的顺序是由插入顺序决定的。

如果检测到任何环，则将引发*CycleError*。

3.9 版新加入.

8.15.1 异常

graphlib 模块定义了以下异常类:

exception graphlib.**CycleError**
ValueError 的子类，当特定的图中存在环时将由*TopologicalSorter.prepare()* 引发。如果存在多个环，则将只报告其中一个未定义的选项并将其包括在异常中。

检测到的环可以通过异常实例的 args 属性的第二个元素来访问，它由一个节点列表组成，其中的每个节点在图中都是列表中下一个节点的直接上级节点。在报告的列表中，开头和末尾的节点将是同一对象，以表明它是一个环。

CHAPTER 9

數值與數學模組

本章介绍的模块提供与数字和数学相关的函数和数据类型。*numbers* 模块定义了数字类型的抽象层次结构。*math* 和*cmath* 模块包含浮点数和复数的各种数学函数。*decimal* 模块支持使用任意精度算术的十进制数的精确表示。

本章记录以下模块：

9.1 numbers --- 数字的抽象基类

源代码： Lib/numbers.py

numbers 模块 (**PEP 3141**) 定义了数字抽象基类 的层级结构，其中逐级定义了更多操作。此模块中定义的类型都不可被实例化。

class numbers.**Number**
数字的层次结构的基础。如果你只想确认参数 *x* 是不是数字而不关心其类型，则使用 `isinstance(x, Number)`。

9.1.1 数字的层次

class numbers.**Complex**
这个类型的子类描述了复数并包括了适用于内置*complex* 类型的操作。这些操作有: 转换为*complex* 和*bool*, *real*, *imag*, +, -, *, /, **, *abs()*, *conjugate()*, == 以及 !=。除 - 和 != 之外所有操作都是抽象的。

real
抽象的。得到该数字的实数部分。

imag
抽象的。得到该数字的虚数部分。

abstractmethod conjugate()
抽象的。返回共轭复数。例如 (1+3j).conjugate() == (1-3j)。

class numbers.**Real**
相对于*Complex*，*Real* 加入了只有实数才能进行的操作。

简单的说，它们是：转化至*float*，*math.trunc()*、*round()*、*math.floor()*、*math.ceil()*、*divmod()*、//、%、<、<=、>、和 >=。

实数同样默认支持*complex()*、*real*、*imag* 和*conjugate()*。

class numbers.**Rational**
子类型*Real* 并加入*numerator* 和*denominator* 两种属性，这两种属性应该属于最低的级别。加入后，这默认支持*float()*。

numerator
抽象的。

denominator
抽象的。

class numbers.**Integral**
子类型*Rational* 还增加了到*int* 的转换操作。为*float()*, *numerator* 和*denominator* 提供了默认支持。为*pow()* 方法增加了求余和按位字符串运算的抽象方法: <<, >>, &, ^, |, ~。

9.1.2 类型接口注释。

实现者需要注意使相等的数字相等并拥有同样的值。当这两个数使用不同的扩展模块时，这其中的差异是很微妙的。例如，用*fractions.Fraction* 实现*hash()* 如下:

```
def __hash__(self):
    if self.denominator == 1:
        # Get integers right.
        return hash(self.numerator)
    # Expensive check, but definitely correct.
    if self == float(self):
        return hash(float(self))
    else:
        # Use tuple's hash to avoid a high collision rate on
        # simple fractions.
        return hash((self.numerator, self.denominator))
```

加入更多数字的 ABC

当然，这里有更多支持数字的 ABC，如果不加入这些，就将缺少层次感。你可以用如下方法在*Complex* 和*Real* 中加入 MyFoo:

```
class MyFoo(Complex): ...
MyFoo.register(Real)
```

实现算术运算

我们希望实现计算，因此，混合模式操作要么调用一个作者知道参数类型的实现，要么转变成为最接近的内置类型并对这个执行操作。对于子类*Integral*，这意味着 __add__() 和 __radd__() 必须用如下方式定义：

```
class MyIntegral(Integral):

    def __add__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(self, other)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(self, other)
        else:
            return NotImplemented

    def __radd__(self, other):
        if isinstance(other, MyIntegral):
            return do_my_adding_stuff(other, self)
        elif isinstance(other, OtherTypeIKnowAbout):
            return do_my_other_adding_stuff(other, self)
        elif isinstance(other, Integral):
            return int(other) + int(self)
        elif isinstance(other, Real):
            return float(other) + float(self)
        elif isinstance(other, Complex):
            return complex(other) + complex(self)
        else:
            return NotImplemented
```

Complex 有 5 种不同的混合类型的操作。我将上面提到的所有代码作为“模板”称作 MyIntegral 和 OtherTypeIKnowAbout。a 是*Complex* 的子类型 A 的实例 (a : A <: Complex)，同时 b : B <: Complex。我将要计算 a + b:

1. 如果 A 被定义成一个承认 b 的 __add__()，一切都没有问题。
2. 如果 A 转回成“模板”失败，它将返回一个属于 __add__() 的值，我们需要避免 B 定义了一个更加智能的 __radd__()，因此模板需要返回一个属于 __add__() 的*NotImplemented*。(或者 A 可能完全不实现 __add__()。)
3. 接着看 B 的 __radd__()。如果它承认 a，一切都没有问题。
4. 如果没有成功回退到模板，就没有更多的方法可以去尝试，因此这里将使用默认的实现。
5. 如果 B <: A，Python 在 A.__add__ 之前尝试 B.__radd__。这是可行的，是通过对 A 的认识实现的，因此这可以在交给*Complex* 处理之前处理这些实例。

如果 A <: Complex 和 B <: Real 没有共享任何资源，那么适当的共享操作涉及内置的*complex*，并且分别获得 __radd__()，因此 a+b == b+a。

由于对任何一直类型的大部分操作是十分相似的，可以定义一个帮助函数，即一个生成后续或相反的实例的生成器。例如，使用*fractions.Fraction* 如下：

```
def _operator_fallbacks(monomorphic_operator, fallback_operator):
    def forward(a, b):
        if isinstance(b, (int, Fraction)):
            return monomorphic_operator(a, b)
        elif isinstance(b, float):
            return fallback_operator(float(a), b)
```

(下页继续)

(繼續上一頁)

```
        elif isinstance(b, complex):
            return fallback_operator(complex(a), b)
        else:
            return NotImplemented
    forward.__name__ = '__' + fallback_operator.__name__ + '__'
    forward.__doc__ = monomorphic_operator.__doc__

    def reverse(b, a):
        if isinstance(a, Rational):
            # Includes ints.
            return monomorphic_operator(a, b)
        elif isinstance(a, numbers.Real):
            return fallback_operator(float(a), float(b))
        elif isinstance(a, numbers.Complex):
            return fallback_operator(complex(a), complex(b))
        else:
            return NotImplemented
    reverse.__name__ = '__r' + fallback_operator.__name__ + '__'
    reverse.__doc__ = monomorphic_operator.__doc__

    return forward, reverse

def _add(a, b):
    """a + b"""
    return Fraction(a.numerator * b.denominator +
                    b.numerator * a.denominator,
                    a.denominator * b.denominator)

__add__, __radd__ = _operator_fallbacks(_add, operator.add)

# ...
```

9.2 math --- 数学函数

该模块提供了对 C 标准定义的数学函数的访问。

这些函数不适用于复数；如果你需要计算复数，请使用*cmath*模块中的同名函数。将支持计算复数的函数区分开的目的，来自于大多数开发者并不愿意像数学家一样需要学习复数的概念。得到一个异常而不是一个复数结果使得开发者能够更早地监测到传递给这些函数的参数中包含复数，进而调查其产生的原因。

该模块提供了以下函数。除非另有明确说明，否则所有返回值均为浮点数。

9.2.1 数论与表示函数

math.**ceil**(*x*)
Return the ceiling of *x*, the smallest integer greater than or equal to *x*. If *x* is not a float, delegates to `x.__ceil__`, which should return an *Integral* value.

math.**comb**(*n*, *k*)
返回不重复且无顺序地从 *n* 项中选择 *k* 项的方式总数。

当 `k <= n` 时取值为 `n! / (k! * (n - k)!)`；当 `k > n` 时取值为零。

也称为二项式系数，因为它等价于表达式 `(1 + x) ** n` 的多项式展开中第 k 项的系数。

如果任一参数不为整数则会引发*TypeError*。如果任一参数为负数则会引发*ValueError*。

3.8 版新加入.

math.**copysign**(*x*, *y*)
返回一个基于 *x* 的绝对值和 *y* 的符号的浮点数。在支持带符号零的平台上，`copysign(1.0, -0.0)` 返回 *-1.0*.

math.**fabs**(*x*)
返回 *x* 的绝对值。

math.**factorial**(*x*)
以一个整数返回 *x* 的阶乘。如果 *x* 不是整数或为负数时则将引发*ValueError*。

3.9 版後已F用: 接受具有整数值的浮点数 (例如 `5.0`) 的行为已被弃用。

math.**floor**(*x*)
Return the floor of *x*, the largest integer less than or equal to *x*. If *x* is not a float, delegates to `x.__floor__`, which should return an *Integral* value.

math.**fmod**(*x*, *y*)
返回 `fmod(x, y)`，由平台 C 库定义。请注意，Python 表达式 `x % y` 可能不会返回相同的结果。C 标准的目的是 `fmod(x, y)` 完全（数学上；到无限精度）等于 `x - n*y` 对于某个整数 *n*，使得结果具有与 *x* 相同的符号和小于 `abs(y)` 的幅度。Python 的 `x % y` 返回带有 *y* 符号的结果，并且可能不能完全计算浮点参数。例如，`fmod(-1e-100, 1e100)` 是 `-1e-100`，但 Python 的 `-1e-100 % 1e100` 的结果是 `1e100-1e-100`，它不能完全表示为浮点数，并且取整为令人惊讶的 `1e100`。出于这个原因，函数*fmod()* 在使用浮点数时通常是首选，而 Python 的 `x % y` 在使用整数时是首选。

math.**frexp**(*x*)
以 `(m, e)` 对的形式返回 *x* 的尾数和指数。*m* 是一个浮点数，*e* 是一个整数，正好是 `x == m * 2**e`。如果 *x* 为零，则返回 `(0.0, 0)`，否则返回 `0.5 <= abs(m) < 1`。这用于以可移植方式“分离”浮点数的内部表示。

math.**fsum**(*iterable*)
返回迭代中的精确浮点值。通过跟踪多个中间部分和来避免精度损失:

```
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
```

该算法的准确性取决于 IEEE-754 算术保证和舍入模式为半偶的典型情况。在某些非 Windows 版本中，底层 C 库使用扩展精度添加，并且有时可能会使中间和加倍，导致它在最低有效位中关闭。

有关待进一步讨论和两种替代方法，参见 ASPN cookbook recipes for accurate floating point summation。

math.**gcd**(**integers*)
返回给定的整数参数的最大公约数。如果有一个参数非零，则返回值将是能同时整除所有参数的最大正整数。如果所有参数为零，则返回值为 `0`。不带参数的 `gcd()` 返回 `0`。

3.5 版新加入.

3.9 版更變: 添加了对任意数量的参数的支持。之前的版本只支持两个参数。

math.**isclose**(*a*, *b*, *, *rel_tol=1e-09*, *abs_tol=0.0*)
若 *a* 和 *b* 的值比较接近则返回 `True`，否则返回 `False`。

根据给定的绝对和相对容差确定两个值是否被认为是接近的。

rel_tol 是相对容差——它是 *a* 和 *b* 之间允许的最大差值，相对于 *a* 或 *b* 的较大绝对值。例如，要设置5%的容差，请传递 `rel_tol=0.05` 。默认容差为 `1e-09`，确保两个值在大约 9 位十进制数字内相同。*rel_tol* 必须大于零。

abs_tol 是最小绝对容差——对于接近零的比较很有用。*abs_tol* 必须至少为零。

如果没有错误发生，结果将是: `abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)` 。

IEEE 754 特殊值 `NaN` ，`inf` 和 `-inf` 将根据 IEEE 规则处理。具体来说，`NaN` 不被认为接近任何其他值，包括 `NaN` 。`inf` 和 `-inf` 只被认为接近自己。

3.5 版新加入.

也参考:

PEP 485 ——用于测试近似相等的函数

math.**isfinite**(*x*)
如果 *x* 既不是无穷大也不是 NaN，则返回 `True` ，否则返回 `False` 。(注意 `0.0` 被认为是有限的。)

3.2 版新加入.

math.**isinf**(*x*)
如果 *x* 是正或负无穷大，则返回 `True` ，否则返回 `False` 。

math.**isnan**(*x*)
如果 *x* 是 NaN（不是数字），则返回 `True` ，否则返回 `False` 。

math.**isqrt**(*n*)
返回非负整数 *n* 的整数平方根。这就是对 *n* 的实际平方根向下取整，或者相当于使得 $a^2 \leq n$ 的最大整数 *a*。

对于某些应用来说，可以更适合取值为使得 $n \leq a^2$ 的最小整数 *a* ，或者换句话说就是 *n* 的实际平方根向上取整。对于正数 *n*，这可以使用 `a = 1 + isqrt(n - 1)` 来计算。

3.8 版新加入.

math.**lcm**(**integers*)
返回给定的整数参数的最小公倍数。如果所有参数均非零，则返回值将是为所有参数的整数倍的最小正整数。如果参数之一为零，则返回值为 `0`。不带参数的 `lcm()` 返回 `1`。

3.9 版新加入.

math.**ldexp**(*x*, *i*)
返回 `x * (2**i)` 。这基本上是函数*`frexp()`* 的反函数。

math.**modf**(*x*)
返回 *x* 的小数和整数部分。两个结果都带有 *x* 的符号并且是浮点数。

math.**nextafter**(*x*, *y*)
返回 *x* 趋向于 *y* 的最接近的浮点数值。

如果 *x* 等于 *y* 则返回 *y*。

示例:

- `math.nextafter(x, math.inf)` 的方向朝上：趋向于正无穷。
- `math.nextafter(x, -math.inf)` 的方向朝下：趋向于负无穷。
- `math.nextafter(x, 0.0)` 趋向于零。
- `math.nextafter(x, math.copysign(math.inf, x))` 趋向于零的反方向。

另请参阅*math.ulp()*。

3.9 版新加入.

`math.`**`perm`**(*n*, *k=None*)
返回不重复且有顺序地从 *n* 项中选择 *k* 项的方式总数。

当 `k <= n` 时取值为 `n! / (n - k)!`；当 `k > n` 时取值为零。

如果 *k* 未指定或为 None，则 *k* 默认值为 *n* 并且函数将返回 `n!`。

如果任一参数不为整数则会引发*TypeError*。如果任一参数为负数则会引发*ValueError*。

3.8 版新加入.

`math.`**`prod`**(*iterable*, *, *start=1*)
计算输入的 *iterable* 中所有元素的积。积的默认 *start* 值为 `1`。

当可迭代对象为空时，返回起始值。此函数特别针对数字值使用，并会拒绝非数字类型。

3.8 版新加入.

`math.`**`remainder`**(*x*, *y*)
返回 IEEE 754 风格的 *x* 相对于 *y* 的余数。对于有限 *x* 和有限非零 *y*，这是差异 `x - n*y`，其中 `n` 是与商 `x / y` 的精确值最接近的整数。如果 `x / y` 恰好位于两个连续整数之间，则将最接近的 *偶数*用作 `n`。余数 `r = remainder(x, y)` 因此总是满足 `abs(r) <= 0.5 * abs(y)`。

特殊情况遵循 IEEE 754：特别是 `remainder(x, math.inf)` 对于任何有限 *x* 都是 *x*，而 `remainder(x, 0)` 和 `remainder(math.inf, x)` 引发*ValueError* 适用于任何非 NaN 的 *x*。如果余数运算的结果为零，则该零将具有与 *x* 相同的符号。

在使用 IEEE 754 二进制浮点的平台上，此操作的结果始终可以完全表示：不会引入舍入错误。

3.7 版新加入.

`math.`**`trunc`**(*x*)
Return *x* with the fractional part removed, leaving the integer part. This rounds toward 0: `trunc()` is equivalent to *floor()* for positive *x*, and equivalent to *ceil()* for negative *x*. If *x* is not a float, delegates to `x.__trunc__`, which should return an *Integral* value.

`math.`**`ulp`**(*x*)
返回浮点数 *x* 的最小有效比特位的值:

- 如果 *x* 是 NaN (非数字)，则返回 *x*。
- 如果 *x* 为负数，则返回 `ulp(-x)`。
- 如果 *x* 为正数，则返回 *x*。
- 如果 *x* 等于零，则返回 *去正规化的可表示最小正浮点数* (小于 *正规化的最小正浮点数*sys.float_info.min)。
- 如果 *x* 等于可表示最大正浮点数，则返回 *x* 的最低有效比特位的值，使得小于 *x* 的第一个浮点数为 `x - ulp(x)`。
- 在其他情况下 (*x* 是一个有限的正数)，则返回 *x* 的最低有效比特位的值，使得大于 *x* 的第一个浮点数为 `x + ulp(x)`。

ULP 即"Unit in the Last Place" 的缩写。

另请参阅*math.nextafter()* 和*sys.float_info.epsilon*。

3.9 版新加入.

注意*frexp()* 和*modf()* 具有与它们的 C 等价函数不同的调用/返回模式：它们采用单个参数并返回一对值，而不是通过' 输出形参' 返回它们的第二个返回参数（Python 中没有这样的东西）。

对于*ceil()*，*floor()* 和*modf()* 函数，请注意 所有足够大的浮点数都是精确整数。Python 浮点数通常不超过 53 位的精度（与平台 C double 类型相同），在这种情况下，任何浮点 *x* 与 `abs(x) >= 2**52` 必然没有小数位。

9.2.2 幂函数与对数函数

`math.`**`exp`**`(`*x*`)`
返回 *e* 次 *x* 幂，其中 *e* = 2.718281... 是自然对数的基数。这通常比 `math.e ** x` 或 `pow(math.e, x)` 更精确。

`math.`**`expm1`**`(`*x*`)`
返回 *e* 的 *x* 次幂，减 1。这里 *e* 是自然对数的基数。对于小浮点数 *x*，`exp(x) - 1` 中的减法可能导致 significant loss of precision；*expm1()* 函数提供了一种将此数量计算为全精度的方法:

```
>>> from math import exp, expm1
>>> exp(1e-5) - 1  # gives result accurate to 11 places
1.0000050000069649e-05
>>> expm1(1e-5)    # result accurate to full precision
1.0000050000166668e-05
```

3.2 版新加入.

`math.`**`log`**`(`*x*`[,` *base*`])`
使用一个参数，返回 *x* 的自然对数（底为 *e*）。

使用两个参数，返回给定的 *base* 的对数 *x*，计算为 `log(x)/log(base)`。

`math.`**`log1p`**`(`*x*`)`
返回 *1*+*x* 的自然对数（以 *e* 为底）。以对于接近零的 *x* 精确的方式计算结果。

`math.`**`log2`**`(`*x*`)`
返回 *x* 以 2 为底的对数。这通常比 `log(x, 2)` 更准确。

3.3 版新加入.

也参考:

int.bit_length() 返回表示二进制整数所需的位数，不包括符号和前导零。

`math.`**`log10`**`(`*x*`)`
返回 *x* 底为 10 的对数。这通常比 `log(x, 10)` 更准确。

`math.`**`pow`**`(`*x*`,` *y*`)`
将返回 `x` 的 `y` 次幂。特殊情况尽可能遵循 C99 标准的附录'F'。特别是，`pow(1.0, x)` 和 `pow(x, 0.0)` 总是返回 `1.0`，即使 `x` 是零或 NaN。如果 `x` 和 `y` 都是有限的，`x` 是负数，`y` 不是整数那么 `pow(x, y)` 是未定义的，并且引发*ValueError*。

与内置的 `**` 运算符不同，*math.pow()* 将其参数转换为*float* 类型。使用 `**` 或内置的*pow()* 函数来计算精确的整数幂。

`math.`**`sqrt`**`(`*x*`)`
返回 *x* 的平方根。

9.2.3 三角函数

math.**acos**(*x*)
返回以弧度为单位的 *x* 的反余弦值。结果范围在 0 到 pi 之间。

math.**asin**(*x*)
返回以弧度为单位的 *x* 的反正弦值。结果范围在 -pi/2 到 pi/2 之间。

math.**atan**(*x*)
返回以弧度为单位的 *x* 的反正切值。结果范围在 -pi/2 到 pi/2 之间。.

math.**atan2**(*y*, *x*)
以弧度为单位返回 atan(y / x) 。结果是在 -pi 和 pi 之间。从原点到点 (x, y) 的平面矢量使该角度与正 X 轴成正比。*atan2()* 的点的两个输入的符号都是已知的，因此它可以计算角度的正确象限。例如，atan(1) 和 atan2(1, 1) 都是 pi/4 ，但 atan2(-1, -1) 是 -3*pi/4 。

math.**cos**(*x*)
返回 *x* 弧度的余弦值。

math.**dist**(*p*, *q*)
返回 *p* 与 *q* 两点之间的欧几里得距离，以一个坐标序列（或可迭代对象）的形式给出。两个点必须具有相同的维度。

大致相当于：

```
sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
```

3.8 版新加入.

math.**hypot**(**coordinates*)
返回欧几里得范数，sqrt(sum(x**2 for x in coordinates))。这是从原点到坐标给定点的向量长度。

对于一个二维点 (x, y)，这等价于使用毕达哥拉斯定义 sqrt(x*x + y*y) 计算一个直角三角形的斜边。

3.8 版更變: 添加了对 n 维点的支持。之前的版本只支持二维点。

math.**sin**(*x*)
返回 *x* 弧度的正弦值。

math.**tan**(*x*)
返回 *x* 弧度的正切值。

9.2.4 角度转换

math.**degrees**(*x*)
将角度 *x* 从弧度转换为度数。

math.**radians**(*x*)
将角度 *x* 从度数转换为弧度。

9.2.5 双曲函数

双曲函数 是基于双曲线而非圆来对三角函数进行模拟。

math.**acosh**(*x*)
返回 *x* 的反双曲余弦值。

math.**asinh**(*x*)
返回 *x* 的反双曲正弦值。

math.**atanh**(*x*)
返回 *x* 的反双曲正切值。

math.**cosh**(*x*)
返回 *x* 的双曲余弦值。

math.**sinh**(*x*)
返回 *x* 的双曲正弦值。

math.**tanh**(*x*)
返回 *x* 的双曲正切值。

9.2.6 特殊函数

math.**erf**(*x*)
返回 *x* 处的 error function 。

erf() 函数可用于计算传统的统计函数，如 累积标准正态分布

```
def phi(x):
    'Cumulative distribution function for the standard normal distribution'
    return (1.0 + erf(x / sqrt(2.0))) / 2.0
```

3.2 版新加入.

math.**erfc**(*x*)
返回 *x* 处的互补误差函数。互补错误函数 定义为 `1.0 - erf(x)`。它用于 *x* 的大值，从其中减去一个会导致 有效位数损失。

3.2 版新加入.

math.**gamma**(*x*)
返回 *x* 处的 伽马函数 值。

3.2 版新加入.

math.**lgamma**(*x*)
返回 Gamma 函数在 *x* 绝对值的自然对数。

3.2 版新加入.

9.2.7 常数

`math.pi`
数学常数 π = 3.141592...，精确到可用精度。

`math.e`
数学常数 e = 2.718281...，精确到可用精度。

`math.tau`
数学常数 τ = 6.283185...，精确到可用精度。Tau 是一个圆周常数，等于 2π，圆的周长与半径之比。更多关于 Tau 的信息可参考 Vi Hart 的视频 Pi is (still) Wrong。吃两倍多的派来庆祝 Tau 日 吧!

3.6 版新加入.

`math.inf`
浮点正无穷大。(对于负无穷大，使用 `-math.inf` 。) 相当于 `float('inf')` 的输出。

3.5 版新加入.

`math.nan`
A floating-point "not a number" (NaN) value. Equivalent to the output of `float('nan')`. Due to the requirements of the IEEE-754 standard, `math.nan` and `float('nan')` are not considered to equal to any other numeric value, including themselves. To check whether a number is a NaN, use the *`isnan()`* function to test for NaNs instead of `is` or ==. Example:

```
>>> import math
>>> math.nan == math.nan
False
>>> float('nan') == float('nan')
False
>>> math.isnan(math.nan)
True
>>> math.isnan(float('nan'))
True
```

3.5 版新加入.

CPython implementation detail: *`math`* 模块主要包含围绕平台 C 数学库函数的简单包装器。特殊情况下的行为在适当情况下遵循 C99 标准的附录 F。当前的实现将引发*`ValueError`* 用于无效操作，如 `sqrt(-1.0)` 或 `log(0.0)` (其中 C99 附件 F 建议发出无效操作信号或被零除)，和*`OverflowError`* 用于溢出的结果 (例如，`exp(1000.0)`)。除非一个或多个输入参数是 NaN，否则不会从上述任何函数返回 NaN；在这种情况下，大多数函数将返回一个 NaN，但是 (再次遵循 C99 附件 F) 这个规则有一些例外，例如 `pow(float('nan'), 0.0)` 或 `hypot(float('nan'), float('inf'))` 。

请注意，Python 不会将显式 NaN 与静默 NaN 区分开来，并且显式 NaN 的行为仍未明确。典型的行为是将所有 NaN 视为静默的。

也参考:

***`cmath`* 模块** 这里很多函数的复数版本。

9.3 cmath --- 关于复数的数学函数

这一模块提供了一些关于复数的数学函数。该模块的函数的参数为整数、浮点数或复数。这些函数的参数也可为一个拥有 `__complex__()` 或 `__float__()` 方法的 Python 对象，这些方法分别用于将对象转换为复数和浮点数，这些函数作用于转换后的结果。

備𝔽: 在具有对于有符号零的硬件和系统级支持的平台上，涉及支割线的函数在支割线的 两侧都是连续的: 零的符号可用来区别支割线的一侧和另一侧。在不支持有符号零的平台上，连续性的规则见下文。

9.3.1 到极坐标和从极坐标的转换

使用 矩形坐标或 笛卡尔坐标在内部存储 Python 复数 `z`。这完全取决于它的 实部 `z.real` 和 虚部 `z.imag`。换句话说:

```
z == z.real + z.imag*1j
```

极坐标提供了另一种复数的表示方法。在极坐标中，一个复数 *z* 由模量 *r* 和相位角 *phi* 来定义。模量 *r* 是从 *z* 到坐标原点的距离，而相位角 *phi* 是以弧度为单位的，逆时针的，从正 X 轴到连接原点和 *z* 的线段间夹角的角度。

下面的函数可用于原生直角坐标与极坐标的相互转换。

cmath.**phase**(*x*)
: 将 *x* 的相位 (也称为 *x* 的 参数) 返回为一个浮点数。`phase(x)` 相当于 `math.atan2(x.imag, x.real)`。结果处于 [-π, π] 之间，以及这个操作的分支切断处于负实轴上，从上方连续。在支持有符号零的系统上（这包涵大多数当前的常用系统），这意味着结果的符号与 `x.imag` 的符号相同，即使 `x.imag` 的值是 0:

```
>>> phase(complex(-1.0, 0.0))
3.141592653589793
>>> phase(complex(-1.0, -0.0))
-3.141592653589793
```

備𝔽: 一个复数 *x* 的模数（绝对值）可以通过内置函数*abs()* 计算。没有单独的*cmath* 模块函数用于这个操作。

cmath.**polar**(*x*)
: 在极坐标中返回 *x* 的表达方式。返回一个数对 `(r, phi)`，*r* 是 *x* 的模数，*phi* 是 *x* 的相位角。`polar(x)` 相当于 `(abs(x), phase(x))`。

cmath.**rect**(*r*, *phi*)
: 通过极坐标的 *r* 和 *phi* 返回复数 *x*。相当于 `r * (math.cos(phi) + math.sin(phi)*1j)`。

9.3.2 幂函数与对数函数

cmath.**exp**(*x*)
返回 *e* 的 *x* 次方，*e* 是自然对数的底数。

cmath.**log**(*x*[, *base*])
返回给定 *base* 的 *x* 的对数。如果没有给定 *base*，返回 *x* 的自然对数。从 0 到 -∞ 存在一条支割线，沿负实轴之上连续。

cmath.**log10**(*x*)
返回底数为 10 的 *x* 的对数。它具有与*log()* 相同的支割线。

cmath.**sqrt**(*x*)
返回 *x* 的平方根。它具有与*log()* 相同的支割线。

9.3.3 三角函数

cmath.**acos**(*x*)
返回 *x* 的反余弦。这里有两条支割线：一条沿着实轴从 1 向右延伸到 ∞，从下面连续延伸。另外一条沿着实轴从 -1 向左延伸到 -∞，从上面连续延伸。

cmath.**asin**(*x*)
返回 *x* 的反正弦。它与*acos()* 有相同的支割线。

cmath.**atan**(*x*)
返回 *x* 的反正切。它具有两条支割线：一条沿着虚轴从 `1j` 延伸到 `∞j`，向右持续延伸。另一条是沿着虚轴从 `-1j` 延伸到 `-∞j` ，向左持续延伸。

cmath.**cos**(*x*)
返回 *x* 的余弦。

cmath.**sin**(*x*)
返回 *x* 的正弦。

cmath.**tan**(*x*)
返回 *x* 的正切。

9.3.4 双曲函数

cmath.**acosh**(*x*)
返回 *x* 的反双曲余弦。它有一条支割线沿着实轴从 1 到 -∞ 向左延伸，从上方持续延伸。

cmath.**asinh**(*x*)
返回 *x* 的反双曲正弦。它有两条支割线：一条沿着虚轴从 `1j` 向右持续延伸到 `∞j`。另一条是沿着虚轴从 `-1j` 向左持续延伸到 `-∞j`。

cmath.**atanh**(*x*)
返回 *x* 的反双曲正切。它有两条支割线：一条是沿着实轴从 `1` 延展到 ∞，从下面持续延展。另一条是沿着实轴从 `-1` 延展到 -∞，从上面持续延展。

cmath.**cosh**(*x*)
返回 *x* 的双曲余弦值。

cmath.**sinh**(*x*)
返回 *x* 的双曲正弦值。

cmath.**tanh**(*x*)
返回 *x* 的双曲正切值。

9.3.5 分类函数

cmath.**isfinite**(*x*)
如果 *x* 的实部和虚部都是有限的，则返回 `True`，否则返回 `False`。

3.2 版新加入.

cmath.**isinf**(*x*)
如果 *x* 的实部或者虚部是无穷大的，则返回 `True`，否则返回 `False`。

cmath.**isnan**(*x*)
如果 *x* 的实部或者虚部是 NaN，则返回 `True`，否则返回 `False`。

cmath.**isclose**(*a*, *b*, ***, *rel_tol=1e-09*, *abs_tol=0.0*)
若 *a* 和 *b* 的值比较接近则返回 `True`，否则返回 `False`。

根据给定的绝对和相对容差确定两个值是否被认为是接近的。

rel_tol 是相对容差——它是 *a* 和 *b* 之间允许的最大差值，相对于 *a* 或 *b* 的较大绝对值。例如，要设置5%的容差，请传递 `rel_tol=0.05`。默认容差为 `1e-09`，确保两个值在大约 9 位十进制数字内相同。*rel_tol* 必须大于零。

abs_tol 是最小绝对容差——对于接近零的比较很有用。*abs_tol* 必须至少为零。

如果没有错误发生，结果将是：`abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol)`。

IEEE 754 特殊值 `NaN`，`inf` 和 *-inf* 将根据 IEEE 规则处理。具体来说，`NaN` 不被认为接近任何其他值，包括 `NaN`。`inf` 和 `-inf` 只被认为接近自己。

3.5 版新加入.

也参考:

PEP 485 ——用于测试近似相等的函数

9.3.6 常数

cmath.**pi**
数学常数 π，作为一个浮点数。

cmath.**e**
数学常数 e，作为一个浮点数。

cmath.**tau**
数学常数 τ，作为一个浮点数。

3.6 版新加入.

cmath.**inf**
浮点正无穷大。相当于 `float('inf')`。

3.6 版新加入.

cmath.**infj**
具有零实部和正无穷虚部的复数。相当于 `complex(0.0, float('inf'))`。

3.6 版新加入.

cmath.**nan**
浮点“非数字”（NaN）值。相当于 `float('nan')`。

3.6 版新加入.

cmath.**nanj**
具有零实部和 NaN 虚部的复数。相当于 `complex(0.0, float('nan'))`。

3.6 版新加入.

请注意，函数的选择与模块*math* 中的函数选择相似，但不完全相同。拥有两个模块的原因是因为有些用户对复数不感兴趣，甚至根本不知道它们是什么。它们宁愿 `math.sqrt(-1)` 引发异常，也不想返回一个复数。另请注意，被*cmath* 定义的函数始终会返回一个复数，尽管答案可以表示为一个实数（在这种情况下，复数的虚数部分为零）。

关于支割线的注释：它们是沿着给定函数无法连续的曲线。它们是许多复变函数的必要特征。假设您需要使用复变函数进行计算，您将会了解支割线的概念。请参阅几乎所有关于复变函数的（不太基本）的书来获得启发。对于如何正确地基于数值目的来选择支割线的相关信息，一个良好的参考如下：

也参考:

Kahan, W: Branch cuts for complex elementary functions; or, Much ado about nothing's sign bit. In Iserles, A., and Powell, M. (eds.), The state of the art in numerical analysis. Clarendon Press (1987) pp165--211.

9.4 decimal --- 十进制定点和浮点运算

源码： Lib/decimal.py

---

decimal 模块为快速正确舍入的十进制浮点运算提供支持。与*float* 数据类型相比，它具有以下几个优点：

- Decimal "基于一个浮点模型，它是为人们设计的，并且必然具有最重要的指导原则——计算机必须提供与人们在学校学习的算法相同的算法。"——摘自十进制算术规范。
- Decimal 数字的表示是完全精确的。相比之下，`1.1` 和 `2.2` 这样的数字在二进制浮点中没有精确的表示。最终用户通常不希望 `1.1 + 2.2` 如二进制浮点数表示那样被显示为 `3.3000000000000003`。
- 精确性延续到算术中。在十进制浮点数中，`0.1 + 0.1 + 0.1 - 0.3` 恰好等于零。在二进制浮点数中，结果为 `5.5511151231257827e-017` 。虽然接近于零，但差异妨碍了可靠的相等性检验，并且差异可能会累积。因此，在具有严格相等不变量的会计应用程序中，decimal 是首选。
- 十进制模块包含有效位的概念，因此 `1.30 + 1.20` 的结果是 `2.50` 。保留尾随零以表示有效位。这是货币的惯用表示方法。乘法则沿用 "教科书 "中：保留被乘数中的所有数字的方法。例如，`1.3 * 1.2` 结果是 `1.56` 而 `1.30 * 1.20` 结果是 `1.5600` 。
- 与基于硬件的二进制浮点不同，十进制模块具有用户可更改的精度（默认为 28 位），可以与给定问题所需的一样大：

```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

- 二进制和十进制浮点都是根据已发布的标准实现的。虽然内置浮点类型只公开其功能的一小部分，但十进制模块公开了标准的所有必需部分。在需要时，程序员可以完全控制舍入和信号处理。这包括通过使用异常来阻止任何不精确操作来强制执行精确算术的选项。
- 十进制模块旨在支持 "无偏见，精确的非连续十进制算术（有时称为定点算术）和舍入浮点算术"。——摘自十进制算术规范。

模块设计以三个概念为中心：十进制数，算术上下文和信号。

十进制数是不可变的。它有一个符号，系数数字和一个指数。为了保持重要性，系数数字不会截断尾随零。十进制数也包括特殊值，例如 `Infinity` ，`-Infinity` ，和 `NaN` 。该标准还区分 `-0` 和 `+0` 。

算术的上下文是指定精度、舍入规则、指数限制、指示操作结果的标志以及确定符号是否被视为异常的陷阱启用器的环境。舍入选项包括*ROUND_CEILING* 、*ROUND_DOWN* 、*ROUND_FLOOR* 、*ROUND_HALF_DOWN*, *ROUND_HALF_EVEN* 、*ROUND_HALF_UP* 、*ROUND_UP* 以及*ROUND_05UP*.

信号是在计算过程中出现的异常条件组。根据应用程序的需要，信号可能会被忽略，被视为信息，或被视为异常。十进制模块中的信号有：*Clamped* 、*InvalidOperation* 、*DivisionByZero* 、*Inexact* 、*Rounded* 、*Subnormal* 、*Overflow* 、*Underflow* 以及*FloatOperation* 。

对于每个信号，都有一个标志和一个陷阱启动器。遇到信号时，其标志设置为 1 ，然后，如果陷阱启用器设置为 1 ，则引发异常。标志是粘性的，因此用户需要在监控计算之前重置它们。

也参考:

- IBM 的通用十进制算术规范，The General Decimal Arithmetic Specification.

9.4.1 快速入门教程

通常使用小数的开始是导入模块，使用*getcontext()* 查看当前上下文，并在必要时为精度、舍入或启用的陷阱设置新值:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[Overflow, DivisionByZero,
        InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

可以从整数、字符串、浮点数或元组构造十进制实例。从整数或浮点构造将执行该整数或浮点值的精确转换。十进制数包括特殊值，例如 `NaN` 代表“非数字”，正的和负的 `Infinity`，和 `-0`

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal(3.14)
Decimal('3.140000000000000124344978758017532527446746826171875')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.4142135623730951')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.4142135623730950488016887242')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

如果*FloatOperation* 信号被捕获，构造函数中的小数和浮点数的意外混合或排序比较会引发异常

```
>>> c = getcontext()
>>> c.traps[FloatOperation] = True
```

(下页继续)

(繼續上一頁)

```
>>> Decimal(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') < 3.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('3.5') == 3.5
True
```

3.3 版新加入.

新 Decimal 的重要性仅由输入的位数决定。上下文精度和舍入仅在算术运算期间发挥作用。

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

如果超出了 C 版本的内部限制，则构造一个十进制将引发*InvalidOperation*

```
>>> Decimal("1e9999999999999999999")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.InvalidOperation: [<class 'decimal.InvalidOperation'>]
```

3.3 版更變.

小数与 Python 的其余部分很好地交互。这是一个小的十进制浮点飞行杂技团：

```
>>> data = list(map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split()))
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
 Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.34
>>> round(a, 1)
Decimal('1.3')
>>> int(a)
1
>>> a * 5
```

(下页继续)

(繼續上一頁)

```
Decimal('6.70')
>>> a * b
Decimal('2.5058')
>>> c % a
Decimal('0.77')
```

Decimal 也可以使用一些数学函数：

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> Decimal('10').log10()
Decimal('1')
```

`quantize()` 方法将数字舍入为固定指数。此方法对于将结果舍入到固定的位置的货币应用程序非常有用：

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

如上所示，*getcontext()* 函数访问当前上下文并允许更改设置。这种方法满足大多数应用程序的需求。

对于更高级的工作，使用 Context() 构造函数创建备用上下文可能很有用。要使用备用活动，请使用*setcontext()* 函数。

根据标准，*decimal* 模块提供了两个现成的标准上下文*BasicContext* 和*ExtendedContext* 。前者对调试特别有用，因为许多陷阱都已启用：

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#143>", line 1, in -toplevel-
    Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

上下文还具有用于监视计算期间遇到的异常情况的信号标志。标志保持设置直到明确清除，因此最好通过使用 `clear_flags()` 方法清除每组受监控计算之前的标志。：

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
        capitals=1, clamp=0, flags=[Inexact, Rounded], traps=[])
```

flags 条目显示对 Pi 的有理逼近被舍入（超出上下文精度的数字被抛弃）并且结果是不精确的（一些丢弃的数字不为零）。

使用上下文的 traps 字段中的字典设置单个陷阱：

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
  File "<pyshell#112>", line 1, in -toplevel-
    Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

大多数程序仅在程序开始时调整当前上下文一次。并且，在许多应用程序中，数据在循环内单个强制转换为*Decimal* 。通过创建上下文集和小数，程序的大部分操作数据与其他 Python 数字类型没有区别。

9.4.2 Decimal 对象

class decimal.**Decimal**(*value="0"*, *context=None*)

根据 *value* 构造一个新的*Decimal* 对象。

value 可以是整数，字符串，元组，*float* ，或另一个*Decimal* 对象。如果没有给出 *value*，则返回 Decimal('0')。如果 *value* 是一个字符串，它应该在前导和尾随空格字符以及下划线被删除之后符合十进制数字字符串语法：

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

当上面出现 digit 时也允许其他十进制数码。其中包括来自各种其他语言系统的十进制数码（例如阿拉伯-印地语和天城文的数码）以及全宽数码 '\uff10' 到 '\uff19'。

如果 *value* 是一个*tuple* ，它应该有三个组件，一个符号（0 表示正数或 1 表示负数），一个数字的*tuple* 和整数指数。例如，Decimal((0, (1, 4, 1, 4), -3)) 返回 Decimal('1.414')。

如果 *value* 是*float* ，则二进制浮点值无损地转换为其精确的十进制等效值。此转换通常需要 53 位或更多位数的精度。例如，Decimal(float('1.1')) 转换为 "Decimal('1.100000000000000088817841970012523233890533447265625')"。

context 精度不会影响存储的位数。这完全由 *value* 中的位数决定。例如，Decimal('3.00000') 记录所有五个零，即使上下文精度只有三。

context 参数的目的是确定 *value* 是格式错误的字符串时该怎么做。如果上下文陷阱*InvalidOperation*，则引发异常；否则，构造函数返回一个新的 Decimal，其值为 NaN。

构造完成后，*Decimal* 对象是不可变的。

3.2 版更變: 现在允许构造函数的参数为*float* 实例。

3.3 版更變: *float* 参数在设置*FloatOperation* 陷阱时引发异常。默认情况下，陷阱已关闭。

3.6 版更變: 允许下划线进行分组，就像代码中的整数和浮点文字一样。

十进制浮点对象与其他内置数值类型共享许多属性，例如*float* 和*int* 。所有常用的数学运算和特殊方法都适用。同样，十进制对象可以复制、pickle、打印、用作字典键、用作集合元素、比较、排序和强制转换为另一种类型（例如*float* 或*int* ）。

算术对十进制对象和算术对整数和浮点数有一些小的差别。当余数运算符 % 应用于 Decimal 对象时，结果的符号是 *被除数*的符号，而不是除数的符号:

```
>>> (-7) % 4
1
>>> Decimal(-7) % Decimal(4)
Decimal('-3')
```

整数除法运算符 // 的行为类似，返回真商的整数部分（截断为零）而不是它的向下取整，以便保留通常的标识 `x == (x // y) * y + x % y`:

```
>>> -7 // 4
-2
>>> Decimal(-7) // Decimal(4)
Decimal('-1')
```

% 和 // 运算符实现了 `remainder` 和 `divide-integer` 操作（分别），如规范中所述。

十进制对象通常不能与浮点数或*fractions.Fraction* 实例在算术运算中结合使用：例如，尝试将*Decimal* 加到*float* ，将引发*TypeError*。但是，可以使用 Python 的比较运算符来比较*Decimal* 实例 x 和另一个数字 y 。这样可以避免在对不同类型的数字进行相等比较时混淆结果。

3.2 版更變: 现在完全支持*Decimal* 实例和其他数字类型之间的混合类型比较。

除了标准的数字属性，十进制浮点对象还有许多专门的方法：

adjusted()
在移出系数最右边的数字之后返回调整后的指数，直到只剩下前导数字：`Decimal('321e+5').adjusted()` 返回 7 。用于确定最高有效位相对于小数点的位置。

as_integer_ratio()
返回一对 `(n, d)` 整数，表示给定的*Decimal* 实例作为分数、最简形式项并带有正分母:

```
>>> Decimal('-3.14').as_integer_ratio()
(-157, 50)
```

转换是精确的。在 Infinity 上引发 OverflowError ，在 NaN 上引起 ValueError 。

3.6 版新加入.

as_tuple()
返回一个*named tuple* 表示的数字：`DecimalTuple(sign, digits, exponent)`。

canonical()
返回参数的规范编码。目前，一个*Decimal* 实例的编码始终是规范的，因此该操作返回其参数不变。

compare(*other*, *context=None*)
比较两个 Decimal 实例的值。*compare()* 返回一个 Decimal 实例，如果任一操作数是 NaN，那么结果是 NaN

```
a or b is a NaN  ==> Decimal('NaN')
a < b            ==> Decimal('-1')
a == b           ==> Decimal('0')
a > b            ==> Decimal('1')
```

compare_signal(*other*, *context=None*)
除了所有 NaN 信号之外，此操作与*compare()* 方法相同。也就是说，如果两个操作数都不是信令 NaN，那么任何静默的 NaN 操作数都被视为信令 NaN。

compare_total(*other*, *context=None*)
使用它们的抽象表示而不是它们的数值来比较两个操作数。类似于*compare()* 方法，但结果给出了一个总排序*Decimal* 实例。两个*Decimal* 实例具有相同的数值但不同的表示形式在此排序中比较不相等：

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

静默和发出信号的 NaN 也包括在总排序中。这个函数的结果是 `Decimal('0')` 如果两个操作数具有相同的表示，或是 `Decimal('-1')` 如果第一个操作数的总顺序低于第二个操作数，或是 `Decimal('1')` 如果第一个操作数在总顺序中高于第二个操作数。有关总排序的详细信息，请参阅规范。

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

compare_total_mag(*other*, *context=None*)
比较两个操作数使用它们的抽象表示而不是它们的值，如*compare_total()*，但忽略每个操作数的符号。`x.compare_total_mag(y)` 相当于 `x.copy_abs().compare_total(y.copy_abs())`。

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

conjugate()
只返回 self，这种方法只符合 Decimal 规范。

copy_abs()
返回参数的绝对值。此操作不受上下文影响并且是静默的：没有更改标志且不执行舍入。

copy_negate()
回到参数的否定。此操作不受上下文影响并且是静默的：没有标志更改且不执行舍入。

copy_sign(*other*, *context=None*)
返回第一个操作数的副本，其符号设置为与第二个操作数的符号相同。例如：

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

exp(*context=None*)
返回给定数字的（自然）指数函数 “e**x“的值。结果使用*ROUND_HALF_EVEN* 舍入模式正确舍入。

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

from_float(*f*)
將浮点数转换为十进制数的类方法。

注意，*Decimal.from_float(0.1)* 与 *Decimal('0.1')* 不同。由于 0.1 在二进制浮点中不能精确表示，因此该值存储为最接近的可表示值，即 *0x1.999999999999ap-4* 。十进制的等效值是 '0.1000000000000000055511151231257827021181583404541015625'。

備𝔽: 从 Python 3.2 开始，*Decimal* 实例也可以直接从*float* 构造。

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(float('-inf'))
Decimal('-Infinity')
```

3.1 版新加入.

fma(*other*, *third*, *context=None*)
混合乘法加法。返回 self*other+third ， 中间乘积 self*other 没有舍入。

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

is_canonical()
如果参数是规范的，则为返回*True*，否则为*False* 。目前，*Decimal* 实例总是规范的，所以这个操作总是返回*True* 。

is_finite()
如果参数是一个有限的数，则返回为*True* ； 如果参数为无穷大或 NaN ， 则返回为*False*。

is_infinite()
如果参数为正负无穷大，则返回为*True* ， 否则为*False* 。

is_nan()
如果参数为 NaN （无论是否静默），则返回为*True* ， 否则为*False* 。

is_normal(*context=None*)
如果参数是一个 标准的有限数则返回*True*。如果参数为零、次标准数、无穷大或 NaN 则返回*False*。

is_qnan()
如果参数为静默 NaN，返回*True*，否则返回*False*。

is_signed()
如果参数带有负号，则返回为*True*，否则返回*False*。注意，0 和 NaN 都可带有符号。

is_snan()
如果参数为显式 NaN，则返回*True*，否则返回*False*。

is_subnormal(*context=None*)
如果参数为次标准数，则返回*True*，否则返回*False*。

is_zero()
如果参数是 0（正负皆可），则返回*True*，否则返回*False*。

ln(*context=None*)
返回操作数的自然对数（以 e 为底）。结果是使用*ROUND_HALF_EVEN* 舍入模式正确舍入的。

log10(*context=None*)
返回操作数的以十为底的对数。结果是使用*ROUND_HALF_EVEN* 舍入模式正确舍入的。

logb(*context=None*)
对于一个非零数，返回其运算数的调整后指数作为一个*Decimal* 实例。如果运算数为零将返回 `Decimal('-Infinity')` 并且产生 the *DivisionByZero* 标志。如果运算数是无限大则返回 `Decimal('Infinity')` 。

logical_and(*other*, *context=None*)
logical_and() 是需要两个 逻辑运算数的逻辑运算（参考逻辑操作数 ）。按位输出两运算数的 and 运算的结果。

logical_invert(*context=None*)
logical_invert() 是一个逻辑运算。结果是操作数的按位求反。

logical_or(*other*, *context=None*)
logical_or() 是需要两个 *logical operands* 的逻辑运算（请参阅逻辑操作数 ）。结果是两个运算数的按位的 or 运算。

logical_xor(*other*, *context=None*)
logical_xor() 是需要两个 逻辑运算数的逻辑运算（参考逻辑操作数 ）。结果是按位输出的两运算数的异或运算。

max(*other*, *context=None*)
像 `max(self, other)` 一样，除了在返回之前应用上下文舍入规则并且用信号通知或忽略 NaN 值（取决于上下文以及它们是发信号还是安静）。

max_mag(*other*, *context=None*)
与*max()* 方法相似，但是操作数使用绝对值完成比较。

min(*other*, *context=None*)
像 `min(self, other)` 一样，除了在返回之前应用上下文舍入规则并且用信号通知或忽略 NaN 值（取决于上下文以及它们是发信号还是安静）。

min_mag(*other*, *context=None*)
与*min()* 方法相似，但是操作数使用绝对值完成比较。

next_minus(*context=None*)
返回小于给定操作数的上下文中可表示的最大数字（或者当前线程的上下文中的可表示的最大数字如果没有给定上下文）。

next_plus(*context=None*)
返回大于给定操作数的上下文中可表示的最小数字（或者当前线程的上下文中的可表示的最小数字如果没有给定上下文）。

next_toward(*other*, *context=None*)
如果两运算数不相等，返回在第二个操作数的方向上最接近第一个操作数的数。如果两操作数数值上相等，返回将符号设置为与第二个运算数相同的第一个运算数的拷贝。

normalize(*context=None*)
通过去除尾随的零并将所有结果等于 `Decimal('0')` 的转化为 `Decimal('0e0')` 来标准化数字。用于为等效类的属性生成规范值。比如，`Decimal('32.100')` 和 `Decimal('0.321000e+2')` 都被标准化为相同的值 `Decimal('32.1')`。

number_class(*context=None*)
返回一个字符串描述运算数的 *class* 。返回值是以下十个字符串中的一个。

- `"-Infinity"`，指示运算数为负无穷大。
- `"-Normal"`，指示该运算数是负正常数字。
- `"-Subnormal"`，指示该运算数是负的次标准数。
- `"-Zero"`，指示该运算数是负零。
- `"-Zero"`，指示该运算数是正零。
- `"+Subnormal"`，指示该运算数是正的次标准数。
- `"+Normal"`，指示该运算数是正的标准数。
- `"+Infinity"`，指示该运算数是正无穷。
- `"NaN"`，指示该运算数是肃静 NaN （非数字）。
- `"sNaN"`，指示该运算数是信号 NaN 。

quantize(*exp*, *rounding=None*, *context=None*)
返回的值等于舍入后的第一个运算数并且具有第二个操作数的指数。

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

与其他运算不同，如果量化运算后的系数长度大于精度，那么会发出一个*InvalidOperation*信号。这保证了除非有一个错误情况，量化指数恒等于右手运算数的指数。

与其他运算不同，量化永不信号下溢，即使结果不正常且不精确。

如果第二个运算数的指数大于第一个运算数的指数那或许需要舍入。在这种情况下，舍入模式由给定 `rounding` 参数决定，其余的由给定 `context` 参数决定；如果参数都未给定，使用当前线程上下文的舍入模式。

每当结果的指数大于 `Emax` 或小于 `Etiny` 就会返回错误。

radix()
返回 `Decimal(10)`，即*Decimal* 类进行所有算术运算所用的数制（基数）。这是为保持与规范描述的兼容性而加入的。

remainder_near(*other*, *context=None*)
返回 *self* 除以 *other* 的余数。这与 `self % other` 的区别在于所选择的余数要使其绝对值最小化。更准确地说，返回值为 `self - n * other` 其中 `n` 是最接近 `self / other` 的实际值的整数，并且如果两个整数与实际值的差相等则会选择其中的偶数。

如果结果为零则其符号将为 *self* 的符号。

```
>>> Decimal(18).remainder_near(Decimal(10))
Decimal('-2')
>>> Decimal(25).remainder_near(Decimal(10))
Decimal('5')
>>> Decimal(35).remainder_near(Decimal(10))
Decimal('-5')
```

rotate(*other*, *context=None*)
返回对第一个操作数的数码按第二个操作数所指定的数量进行轮转的结果。第二个操作数必须为 -precision 至 precision 精度范围内的整数。第二个操作数的绝对值给出要轮转的位数。如果第二个操作数为正值则向左轮转；否则向右轮转。如有必要第一个操作数的系数会在左侧填充零以达到 precision 所指定的长度。第一个操作数的符号和指数保持不变。

same_quantum(*other*, *context=None*)
检测自身与 other 是否具有相同的指数或是否均为 NaN。

此操作不受上下文影响且静默：不更改任何标志且不执行舍入。作为例外，如果无法准确转换第二个操作数，则 C 版本可能会引发 InvalidOperation。

scaleb(*other*, *context=None*)
返回第一个操作数使用第二个操作数对指数进行调整的结果。等价于返回第一个操作数乘以 10**other 的结果。第二个操作数必须为整数。

shift(*other*, *context=None*)
返回第一个操作数的数码按第二个操作数所指定的数量进行移位的结果。第二个操作数必须为 -precision 至 precision 范围内的整数。第二个操作数的绝对值给出要移动的位数。如果第二个操作数为正值则向左移位；否则向右移位。移入系数的数码为零。第一个操作数的符号和指数保持不变。

sqrt(*context=None*)
返回参数的平方根精确到完整精度。

to_eng_string(*context=None*)
转换为字符串，如果需要指数则会使用工程标注法。

工程标注法的指数是 3 的倍数。这会在十进制位的左边保留至多 3 个数码，并可能要求添加一至两个末尾零。

例如，此方法会将 Decimal('123E+1') 转换为 Decimal('1.23E+3')。

to_integral(*rounding=None*, *context=None*)
与*to_integral_value()* 方法相同。保留 to_integral 名称是为了与旧版本兼容。

to_integral_exact(*rounding=None*, *context=None*)
舍入到最接近的整数，发出信号*Inexact* 或者如果发生舍入则相应地发出信号*Rounded*。如果给出 rounding 形参则由其确定舍入模式，否则由给定的 context 来确定。如果没有给定任何形参则会使用当前上下文的舍入模式。

to_integral_value(*rounding=None*, *context=None*)
舍入到最接近的整数而不发出*Inexact* 或*Rounded* 信号。如果给出 *rounding* 则会应用其所指定的舍入模式；否则使用所提供的 *context* 或当前上下文的舍入方法。

逻辑操作数

logical_and(), logical_invert(), logical_or() 和 logical_xor() 方法期望其参数为 *逻辑操作数*。*逻辑操作数*是指数位与符号位均为零的*Decimal* 实例，并且其数字位均为 0 或 1。

9.4.3 上下文对象

上下文是算术运算所在的环境。它们管理精度、设置舍入规则、确定将哪些信号视为异常，并限制指数的范围。

每个线程都有自己的当前上下文，可使用*getcontext()* 和*setcontext()* 函数来读取或修改：

decimal.**getcontext**()
返回活动线程的当前上下文。

decimal.**setcontext**(*c*)
将活动线程的当前上下文设为 *c*。

你也可以使用 with 语句和*localcontext()* 函数来临时改变活动上下文。

decimal.**localcontext**(*ctx=None*)

返回一个上下文管理器，它将在进入 with 语句时将活动线程的当前上下文设为 *ctx* 的一个副本并在退出 with 语句时恢复之前的上下文。如果未指定上下文，则会使用当前上下文的一个副本。

例如，以下代码会将当前 decimal 精度设为 42 位，执行一个运算，然后自动恢复之前的上下文:

```
from decimal import localcontext

with localcontext() as ctx:
    ctx.prec = 42   # Perform a high precision calculation
    s = calculate_something()
s = +s  # Round the final result back to the default precision
```

新的上下文也可使用下述的*Context* 构造器来创建。此外，模块还提供了三种预设的上下文:

class decimal.**BasicContext**

这是由通用十进制算术规范描述所定义的标准上下文。精度设为九。舍入设为*ROUND_HALF_UP*。清除所有旗标。启用所有陷阱（视为异常），但*Inexact*, *Rounded* 和*Subnormal* 除外。

由于启用了许多陷阱，此上下文适用于进行调试。

class decimal.**ExtendedContext**

这是由通用十进制算术规范描述所定义的标准上下文。精度设为九。舍入设为*ROUND_HALF_EVEN*。清除所有旗标。不启用任何陷阱（因此在计算期间不会引发异常）。

由于禁用了陷阱，此上下文适用于希望结果值为 NaN 或 Infinity 而不是引发异常的应用。这允许应用在出现当其他情况下会中止程序的条件时仍能完成运行。

class decimal.**DefaultContext**

此上下文被*Context* 构造器用作新上下文的原型。改变一个字段（例如精度）的效果将是改变*Context* 构造器所创建的新上下文的默认值。

此上下文最适用于多线程环境。在线程开始前改变一个字段具有设置全系统默认值的效果。不推荐在线程开始后改变字段，因为这会要求线程同步避免竞争条件。

在单线程环境中，最好完全不使用此上下文。而是简单地电显式创建上下文，具体如下所述。

默认值为 prec=28, rounding=*ROUND_HALF_EVEN*，并为*Overflow*, *InvalidOperation* 和*DivisionByZero* 启用陷阱。

在已提供的三种上下文之外，还可以使用*Context* 构造器创建新的上下文。

class decimal.**Context**(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*)

创建一个新上下文。如果某个字段未指定或为*None*，则从*DefaultContext* 拷贝默认值。如果 *flags* 字段未指定或为*None*，则清空所有旗标。

prec 为一个 [1, *MAX_PREC*] 范围内的整数，用于设置该上下文中算术运算的精度。

rounding 选项应为*Rounding Modes* 小节中列出的常量之一。

traps 和 *flags* 字段列出要设置的任何信号。通常，新上下文应当只设置 traps 而让 flags 为空。

Emin 和 *Emax* 字段给定指数所允许的外部上限。*Emin* 必须在 [*MIN_EMIN*, 0] 范围内，*Emax* 在 [0, *MAX_EMAX*] 范围内。

capitals 字段为 0 或 1 (默认值)。如果设为 1，指数将附带打印大写的 E；其他情况则将使用小写的 e: Decimal('6.02e+23')。

clamp 字段为 0 (默认值) 或 1。如果设为 1，则*Decimal* 实例的指数 e 的表示范围在此上下文中将严格限制为 Emin - prec + 1 <= e <= Emax - prec + 1。如果 *clamp* 为 0 则将适用较弱的条件: *Decimal* 实例调整后的指数最大值为 Emax。当 *clamp* 为 1 时，一个较大的普通数值将在可能的情况

下减小其指数并为其系统添加相应数量的零，以便符合指数值限制；这可以保持数字值但会丢失有效末尾零的信息。例如:

```
>>> Context(prec=6, Emax=999, clamp=1).create_decimal('1.23e999')
Decimal('1.23000E+999')
```

clamp 值为 1 时即允许与在 IEEE 754 中描述的固定宽度十进制交换格式保持兼容性。

Context 类定义了几种通用方法以及大量直接在给定上下文中进行算术运算的方法。此外，对于上述的每种*Decimal* 方法（不包括 adjusted() 和 as_tuple() 方法）都有一个相应的*Context* 方法。例如，对于一个*Context* 的实例 C 和*Decimal* 的实例 x，C.exp(x) 就等价于 x.exp(context=C)。每个*Context* 方法都接受一个 Python 整数（即*int* 的实例）在任何接受 Decimal 的实例的地方使用。

clear_flags()
将所有旗标重置为 0。

clear_traps()
将所有陷阱重置为零 0。

3.3 版新加入.

copy()
返回上下文的一个副本。

copy_decimal(*num*)
返回 Decimal 实例 num 的一个副本。

create_decimal(*num*)
基于 *num* 创建一个新 Decimal 实例但使用 *self* 作为上下文。与*Decimal* 构造器不同，该上下文的精度、舍入方法、旗标和陷阱会被应用于转换过程。

此方法很有用处，因为常量往往被给予高于应用所需的精度。另一个好处在于立即执行舍入可以消除超出当前精度的数位所导致的意外效果。在下面的示例中，使用未舍入的输入意味着在总和中添加零会改变结果:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

此方法实现了 IBM 规格描述中的转换为数字操作。如果参数为字符串，则不允许有开头或末尾的空格或下划线。

create_decimal_from_float(*f*)
基于浮点数 *f* 创建一个新的 Decimal 实例，但会使用 *self* 作为上下文来执行舍入。与*Decimal.from_float()* 类方法不同，上下文的精度、舍入方法、旗标和陷阱会应用到转换中。

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(math.pi)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(math.pi)
Traceback (most recent call last):
    ...
decimal.Inexact: None
```

3.1 版新加入.

Etiny()
返回一个等于 `Emin - prec + 1` 的值即次标准化结果中的最小指数值。当发生向下溢出时，指数会设为*Etiny*。

Etop()
返回一个等于 `Emax - prec + 1` 的值。

使用 decimal 的通常方式是创建*Decimal* 实例然后对其应用算术运算, 这些运算发生在活动线程的当前上下文中。一种替代方式则是使用上下文的方法在特定上下文中进行计算。这些方法类似于*Decimal* 类的方法，在此仅简单地重新列出。

abs(*x*)
返回 *x* 的绝对值。

add(*x*, *y*)
返回 *x* 与 *y* 的和。

canonical(*x*)
返回相同的 Decimal 对象 *x*。

compare(*x*, *y*)
对 *x* 与 *y* 进行数值比较。

compare_signal(*x*, *y*)
对两个操作数进行数值比较。

compare_total(*x*, *y*)
对两个操作数使用其抽象表示进行比较。

compare_total_mag(*x*, *y*)
对两个操作数使用其抽象表示进行比较，忽略符号。

copy_abs(*x*)
返回 *x* 的副本，符号设为 0。

copy_negate(*x*)
返回 *x* 的副本，符号取反。

copy_sign(*x*, *y*)
从 *y* 拷贝符号至 *x*。

divide(*x*, *y*)
返回 *x* 除以 *y* 的结果。

divide_int(*x*, *y*)
返回 *x* 除以 *y* 的结果，截短为整数。

divmod(*x*, *y*)
两个数字相除并返回结果的整数部分。

exp(*x*)
返回 *e* ** *x*。

fma(*x*, *y*, *z*)
返回 *x* 乘以 *y* 再加 *z* 的结果。

is_canonical(*x*)
如果 *x* 是规范的则返回 `True`；否则返回 `False`。

is_finite(*x*)
如果 *x* 为有限的则返回 “True“；否则返回 `False`。

is_infinite(*x*)
如果 *x* 是无限的则返回 `True`；否则返回 `False`。

is_nan(*x*)
如果 *x* 是 qNaN 或 sNaN 则返回 `True`；否则返回 `False`。

is_normal(*x*)
如果 *x* 是标准数则返回 `True`；否则返回 `False`。

is_qnan(*x*)
如果 *x* 是静默 NaN 则返回 `True`；否则返回 `False`。

is_signed(*x*)
x 是负数则返回 `True`；否则返回 `False`。

is_snan(*x*)
如果 *x* 是显式 NaN 则返回 `True`；否则返回 `False`。

is_subnormal(*x*)
如果 *x* 是次标准数则返回 `True`；否则返回 `False`。

is_zero(*x*)
如果 *x* 为零则返回 `True`；否则返回 `False`。

ln(*x*)
返回 *x* 的自然对数（以 e 为底）。

log10(*x*)
返回 *x* 的以 10 为底的对数。

logb(*x*)
返回操作数的 MSD 等级的指数。

logical_and(*x*, *y*)
在操作数的每个数位间应用逻辑运算 *and*。

logical_invert(*x*)
反转 *x* 中的所有数位。

logical_or(*x*, *y*)
在操作数的每个数位间应用逻辑运算 *or*。

logical_xor(*x*, *y*)
在操作数的每个数位间应用逻辑运算 *xor*。

max(*x*, *y*)
对两个值执行数字比较并返回其中的最大值。

max_mag(*x*, *y*)
对两个值执行忽略正负号的数字比较。

min(*x*, *y*)
对两个值执行数字比较并返回其中的最小值。

min_mag(*x*, *y*)
对两个值执行忽略正负号的数字比较。

minus(*x*)
对应于 Python 中的单目前缀取负运算符执行取负操作。

multiply(*x*, *y*)
返回 *x* 和 *y* 的积。

next_minus(*x*)
返回小于 *x* 的最大数字表示形式。

next_plus(*x*)
返回大于 *x* 的最小数字表示形式。

next_toward(*x*, *y*)
返回 *x* 趋向于 *y* 的最接近的数字。

normalize(*x*)
将 *x* 改写为最简形式。

number_class(*x*)
返回 *x* 的类的表示。

plus(*x*)
对应于 Python 中的单目前缀取正运算符执行取正操作。此操作将应用上下文精度和舍入，因此它不是标识运算。

power(*x*, *y*, *modulo=None*)
返回 `x` 的 `y` 次方，如果给出了模数 `modulo` 则取其余数。

如为两个参数则计算 `x**y`。如果 `x` 为负值则 `y` 必须为整数。除非 `y` 为整数且结果为有限值并可在'precision' 位内精确表示否则结果将是不精确的。上下文的舍入模式将被使用。结果在 Python 版中总是会被正确地舍入。

`Decimal(0) ** Decimal(0)` 结果为 `InvalidOperation`，而如果 `InvalidOperation` 未被捕获，则结果为 `Decimal('NaN')`。

3.3 版更變: C 模块计算*power()* 时会使用已正确舍入的*exp()* 和*ln()* 函数。结果是经过良好定义的，但仅限于“几乎总是正确地舍入”。

带有三个参数时，计算 `(x**y) % modulo`。对于三个参数的形式，参数将会应用以下限制：

- 三个参数必须都是整数
- `y` 必须是非负数
- `x` 或 `y` 至少有一个不为零
- `modulo` 必须不为零且至多有'precision' 位

来自 `Context.power(x, y, modulo)` 的结果值等于使用无限精度计算 `(x**y) % modulo` 所得到的值，但其计算过程更高效。结果的指数为零，无论 `x`, `y` 和 `modulo` 的指数是多少。结果值总是完全精确的。

quantize(*x*, *y*)
返回的值等于 *x* (舍入后)，并且指数为 *y*。

radix()
恰好返回 10，因为这是 Decimal 对象:)

remainder(*x*, *y*)
返回整除所得到的余数。

结果的符号，如果不为零，则与原始除数的符号相同。

remainder_near(*x*, *y*)
返回 `x - y * n`，其中 *n* 为最接近 `x / y` 实际值的整数（如结果为 0 则其符号将与 *x* 的符号相同）。

rotate(*x*, *y*)
返回 *x* 翻转 *y* 次的副本。

same_quantum(*x*, *y*)
如果两个操作数具有相同的指数则返回 `True`。

scaleb(*x*, *y*)
返回第一个操作数添加第二个值的指数后的结果。

shift(*x*, *y*)
返回 *x* 变换 *y* 次的副本。

sqrt(*x*)
非负数基于上下文精度的平方根。

subtract(*x*, *y*)
返回 *x* 和 *y* 的差。

to_eng_string(*x*)
转换为字符串，如果需要指数则会使用工程标注法。

工程标注法的指数是 3 的倍数。这会在十进制位的左边保留至多 3 个数码，并可能要求添加一至两个末尾零。

to_integral_exact(*x*)
舍入到一个整数。

to_sci_string(*x*)
使用科学计数法将一个数字转换为字符串。

9.4.4 常数

本节中的常量仅与 C 模块相关。它们也被包含在纯 Python 版本以保持兼容性。

	32 位	64 位
decimal.**MAX_PREC**	425000000	999999999999999999
decimal.**MAX_EMAX**	425000000	999999999999999999
decimal.**MIN_EMIN**	-425000000	-999999999999999999
decimal.**MIN_ETINY**	-849999999	-1999999999999999997

decimal.**HAVE_THREADS**
该值为 True。已弃用，因为 Python 现在总是启用线程。

3.9 版後已𝔽用.

decimal.**HAVE_CONTEXTVAR**
默认值为 True。如果 Python 的编译带有 --without-decimal-contextvar 选项，则 C 版本会使用线程局部而不是协程局部上下文并且该值为 False。这在某些嵌套上下文场景中将稍快一些。

3.9 版新加入: 向下移植到 3.7 和 3.8。

9.4.5 舍入模式

decimal.**ROUND_CEILING**
舍入方向为 Infinity。

decimal.**ROUND_DOWN**
舍入方向为零。

decimal.**ROUND_FLOOR**
舍入方向为 -Infinity。

decimal.**ROUND_HALF_DOWN**
舍入到最接近的数，同样接近则舍入方向为零。

decimal.**ROUND_HALF_EVEN**
舍入到最接近的数，同样接近则舍入到最接近的偶数。

decimal.**ROUND_HALF_UP**
舍入到最接近的数，同样接近则舍入到零的反方向。

decimal.**ROUND_UP**
舍入到零的反方向。

decimal.**ROUND_05UP**
如果最后一位朝零的方向舍入后为 0 或 5 则舍入到零的反方向；否则舍入方向为零。

9.4.6 信号

信号代表在计算期间引发的条件。每个信号对应于一个上下文旗标和一个上下文陷阱启用器。

上下文旗标将在遇到特定条件时被设定。在完成计算之后，将为了获得信息而检测旗标（例如确定计算是否精确）。在检测旗标后，请确保在开始下一次计算之前清除所有旗标。

如果为信号设定了上下文的陷阱启用器，则条件会导致特定的 Python 异常被引发。举例来说，如果设定了*DivisionByZero* 陷阱，则当遇到此条件时就将引发*DivisionByZero* 异常。

class decimal.**Clamped**
修改一个指数以符合表示限制。

通常，限位将在一个指数超出上下文的 Emin 和 Emax 限制时发生。在可能的情况下，会通过给系数添加零来将指数缩减至符合限制。

class decimal.**DecimalException**
其他信号的基类，并且也是*ArithmeticError* 的一个子类。

class decimal.**DivisionByZero**
非无限数被零除的信号。

可在除法、取余队法或对一个数求负数次幂时发生。如果此信号未被陷阱捕获，则返回 Infinity 或 -Infinity 并且由对计算的输入来确定正负符号。

class decimal.**Inexact**
表明发生了舍入且结果是不精确的。

有非零数位在舍入期间被丢弃的信号。舍入结果将被返回。此信号旗标或陷阱被用于检测结果不精确的情况。

class decimal.**InvalidOperation**
执行了一个无效的操作。

表明请求了一个无意义的操作。如未被陷阱捕获则返回 NaN。可能的原因包括:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class decimal.**Overflow**
数值的溢出。

表明在发生舍入之后的指数大于 Emax。如果未被陷阱捕获，则结果将取决于舍入模式，或者向下舍入为最大的可表示有限数，或者向上舍入为 Infinity。无论哪种情况，都将引发*Inexact* 和*Rounded* 信号。

class decimal.**Rounded**
发生了舍入，但或许并没有信息丢失。

一旦舍入丢弃了数位就会发出此信号；即使被丢弃的数位是零 (例如将 5.00 舍入为 5.0)。如果未被陷阱捕获，则不经修改地返回结果。此信号用于检测有效位数的丢弃。

class decimal.**Subnormal**
在舍入之前指数低于 Emin。

当操作结果是次标准数（即指数过小）时就会发出此信号。如果未被陷阱捕获，则不经修改过返回结果。

class decimal.**Underflow**
数字向下溢出导致结果舍入到零。

当一个次标准数结果通过舍入转为零时就会发出此信号。同时还将引发*Inexact* 和*Subnormal* 信号。

class decimal.**FloatOperation**
为 float 和 Decimal 的混合启用更严格的语义。

如果信号未被捕获（默认），则在*Decimal* 构造器、*create_decimal()* 和所有比较运算中允许 float 和 Decimal 的混合。转换和比较都是完全精确的。发生的任何混合运算都将通过在上下文旗标中设置*FloatOperation* 来静默地记录。通过*from_float()* 或*create_decimal_from_float()* 进行显式转换则不会设置旗标。

在其他情况下（即信号被捕获），则只静默执行相等性比较和显式转换。所有其他混合运算都将引发*FloatOperation*。

以下表格总结了信号的层级结构:

```
exceptions.ArithmeticError(exceptions.Exception)
    DecimalException
        Clamped
        DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
        Inexact
            Overflow(Inexact, Rounded)
            Underflow(Inexact, Rounded, Subnormal)
        InvalidOperation
        Rounded
        Subnormal
        FloatOperation(DecimalException, exceptions.TypeError)
```

9.4.7 浮点数说明

通过提升精度来解决舍入错误

使用十进制浮点数可以消除十进制表示错误（即能够完全精确地表示 `0.1` 这样的数）；然而，某些运算在非零数位超出给定的精度时仍然可能导致舍入错误。

舍入错误的影响可能因接近相互抵销的加减运算被放大从而导致丢失有效位。Knuth 提供了两个指导性示例，其中出现了精度不足的浮点算术舍入，导致加法的交换律和分配律被打破：

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

decimal 模块则可以通过充分地扩展精度来避免有效位的丢失：

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

特殊的值

decimal 模块的数字系统提供了一些特殊的值，包括 `NaN`, `sNaN`, `-Infinity`, `Infinity` 以及两种零值 `+0` 和 `-0`。

无穷大可以使用 `Decimal('Infinity')` 来构建。它们也可以在不捕获*DivisionByZero* 信号捕获时通过除以零来产生。类似地，当不捕获*Overflow* 信号时，也可以通过舍入到超出最大可表示数字限制的方式产生无穷大的结果。

无穷大是有符号的（仿射）并可用于算术运算，它们会被当作极其巨大的不确定数字来处理。例如，无穷大加一个常量结果也将为无穷大。

某些不存在有效结果的运算将会返回 `NaN`，或者如果捕获了*InvalidOperation* 信号则会引发一个异常。例如，`0/0` 会返回 `NaN` 表示结果“不是一个数字”。这样的 `NaN` 是静默产生的，并且在产生之后参与其它计算时总是会得到 `NaN` 的结果。这种行为对于偶而缺少输入的各类计算都很有用处 --- 它允许在将特定结果标记为无效的同时让计算继续运行。

另一种变体形式是 sNaN，它在每次运算后会发出信号而不是保持静默。当对于无效结果需要中断计算进行特别处理时，这是一个很有用的返回值。

Python 中比较运算符的行为在涉及 NaN 时可能会令人有点惊讶。相等性检测在操作数中有静默型或信号型 NaN 时总是会返回*False* (即使是执行 Decimal('NaN')==Decimal('NaN'))，而不等性检测总是会返回*True*。当尝试使用 <, <=, > 或 >= 运算符中的任何一个来比较两个 Decimal 值时，如果运算数中有 NaN 则将引发*InvalidOperation* 信号，如果此信号未被捕获则将返回*False*。请注意通用十进制算术规范并未规定直接比较行为；这些涉及 NaN 的比较规则来自于 IEEE 854 标准 (见第 5.7 节表 3)。要确保严格符合标准，请改用 compare() 和 compare-signal() 方法。

有符号零值可以由向下溢出的运算产生。它们保留符号是为了让运算结果能以更高的精度传递。由于它们的大小为零，正零和负零会被视为相等，且它们的符号具有信息。

在这两个不相同但却相等的有符号零之外，还存在几种零的不同表示形式，它们的精度不同但值也都相等。这需要一些时间来逐渐适应。对于习惯了标准浮点表示形式的眼睛来说，以下运算返回等于零的值并不是显而易见的：

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000026')
```

9.4.8 使用线程

getcontext() 函数会为每个线程访问不同的*Context* 对象。具有单独线程上下文意味着线程可以修改上下文 (例如 getcontext().prec=10) 而不影响其他线程。

类似的*setcontext()* 会为当前上下文的目标自动赋值。

如果在调用*setcontext()* 之前调用了*getcontext()*，则*getcontext()* 将自动创建一个新的上下文在当前线程中使用。

新的上下文拷贝自一个名为 *DefaultContext* 的原型上下文。要控制默认值以便每个线程在应用运行期间都使用相同的值，可以直接修改 *DefaultContext* 对象。这应当在任何线程启动 之前完成以使得调用*getcontext()* 的线程之间不会产生竞争条件。例如:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
 . . .
```

9.4.9 例程

以下是一些用作工具函数的例程，它们演示了使用*Decimal*类的各种方式:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
    """Convert Decimal to a money formatted string.

    places:  required number of places after the decimal point
    curr:    optional currency symbol before the sign (may be blank)
    sep:     optional grouping separator (comma, period, space, or blank)
    dp:      decimal point indicator (comma or period)
             only specify as blank when places is zero
    pos:     optional sign for positive numbers: '+', space or blank
    neg:     optional sign for negative numbers: '-', '(', space or blank
    trailneg:optional trailing minus indicator:  '-', ')', space or blank

    >>> d = Decimal('-1234567.8901')
    >>> moneyfmt(d, curr='$')
    '-$1,234,567.89'
    >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
    '1.234.568-'
    >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
    '($1,234,567.89)'
    >>> moneyfmt(Decimal(123456789), sep=' ')
    '123 456 789.00'
    >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
    '<0.02>'

    """
    q = Decimal(10) ** -places      # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = list(map(str, digits))
    build, next = result.append, digits.pop
    if sign:
        build(trailneg)
    for i in range(places):
        build(next() if digits else '0')
    if places:
        build(dp)
    if not digits:
        build('0')
    i = 0
    while digits:
        build(next())
        i += 1
        if i == 3 and digits:
            i = 0
            build(sep)
    build(curr)
    build(neg if sign else pos)
    return ''.join(reversed(result))

def pi():
    """Compute Pi to the current precision.

    >>> print(pi())
```

(下页继续)

(繼續上一頁)

```
    3.14159265358979323846264338327

    """
    getcontext().prec += 2  # extra digits for intermediate steps
    three = Decimal(3)      # substitute "three=3.0" for regular floats
    lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
    while s != lasts:
        lasts = s
        n, na = n+na, na+8
        d, da = d+da, da+32
        t = (t * n) / d
        s += t
    getcontext().prec -= 2
    return +s               # unary plus applies the new precision

def exp(x):
    """Return e raised to the power of x.  Result type matches input type.

    >>> print(exp(Decimal(1)))
    2.718281828459045235360287471
    >>> print(exp(Decimal(2)))
    7.389056098930650227230427461
    >>> print(exp(2.0))
    7.38905609893
    >>> print(exp(2+0j))
    (7.38905609893+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s

def cos(x):
    """Return the cosine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(cos(Decimal('0.5')))
    0.8775825618903727161162815826
    >>> print(cos(0.5))
    0.87758256189
    >>> print(cos(0.5+0j))
    (0.87758256189+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
    while s != lasts:
```

(下页继续)

(繼續上一頁)

```
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s

def sin(x):
    """Return the sine of x as measured in radians.

    The Taylor series approximation works best for a small value of x.
    For larger values, first compute x = x % (2 * pi).

    >>> print(sin(Decimal('0.5')))
    0.47942553860420300027328793521
    >>> print(sin(0.5))
    0.479425538604
    >>> print(sin(0.5+0j))
    (0.479425538604+0j)

    """
    getcontext().prec += 2
    i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
    while s != lasts:
        lasts = s
        i += 2
        fact *= i * (i-1)
        num *= x * x
        sign *= -1
        s += num / fact * sign
    getcontext().prec -= 2
    return +s
```

9.4.10 Decimal 常见问题

Q. 总是输入 `decimal.Decimal('1234.5')` 是否过于笨拙。在使用交互解释器时有没有最小化输入量的方式?

A. 有些用户会将构造器简写为一个字母:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. 在带有两个十进制位的定点数应用中，有些输入值具有许多位，需要被舍入。另一些数则不应具有多余位，需要验证有效性。这种情况应该用什么方法?

A. 用 `quantize()` 方法舍入到固定数量的十进制位。如果设置了 *Inexact* 陷阱，它也适用于验证有效性:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')
```

```
>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')
```

```
>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')
```

```
>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
   ...
Inexact: None
```

Q. 当我使用两个有效位的输入时，我要如何在一个应用中保持有效位不变?

A. 某些运算例如与整数相加、相减和相乘将会自动保留固定的小数位数。其他运算，例如相除和非整数相乘则将会改变小数位数，需要再加上 `quantize()` 处理步骤：

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)     # And quantize division
Decimal('0.03')
```

在开发定点数应用时，更方便的做法是定义处理 `quantize()` 步骤的函数：

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)
```

```
>>> mul(a, b)                       # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. 表示同一个值有许多方式。数字 `200`, `200.000`, `2E2` 和 `02E+4` 的值都相同但有精度不同。是否有办法将它们转换为一个可识别的规范值?

A. `normalize()` 方法可将所有相同的值映射为统一表示形式：

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. 有些十进制值总是被打印为指数表示形式。是否有办法得到一个非指数表示形式?

A. 对于某些值来说，指数表示形式是表示系数中有效位的唯一办法。例如，将 `5.0E+3` 表示为 `5000` 可以让值保持恒定，但是无法显示原本的两位有效数字。

如果一个应用不必关心追踪有效位，则可以很容易地移除指数和末尾的零，丢弃有效位但让值保持不变：

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()
```

```
>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. 是否有办法将一个普通浮点数转换为*Decimal*?

A. 是的，任何二进制浮点数都可以精确地表示为 Decimal 值，但完全精确的转换可能需要比平常感觉更高的精度：

```
>>> Decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. 在一个复杂的计算中，我怎样才能保证不会得到由精度不足和舍入异常所导致的虚假结果。

A. 使用 decimal 模块可以很容易地检测结果。最好的做法是使用更高的精度和不同的舍入模式重新进行计算。明显不同的结果表明存在精度不足、舍入模式问题、不符合条件的输入或是结果不稳定的算法。

Q. 我发现上下文精度的应用只针对运算结果而不针对输入。在混合使用不同精度的值时有什么需要注意的吗?

A. 是的。原则上所有值都会被视为精确值，在这些值上进行的算术运算也是如此。只有结果会被舍入。对于输入来说其好处是“所输入即所得”。而其缺点则是如果你忘记了输入没有被舍入，结果看起来可能会很奇怪：

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

解决办法是提高精度或使用单目加法运算对输入执行强制舍入：

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

此外，还可以使用*Context.create_decimal()* 方法在创建输入时执行舍入：

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```

Q. CPython 实现对于巨大数字是否足够快速?

A. 是的。在 CPython 和 PyPy3 实现中，decimal 模块的 C/CFFI 版本集成了高速 libmpdec 库用于实现任意精度正确舍入的十进制浮点算术[1]。`libmpdec` 会对中等大小的数字使用 Karatsuba 乘法 而对非常巨大的数字使用数字原理变换。

必须要对任意精度算术适配上下文。`Emin` 和 `Emax` 应当总是设为最大值，`clamp` 应当总是设为 0 (默认值)。设置 `prec` 需要十分谨慎。

进行大数字算术的最便捷方式也是使用 `prec` 的最大值[2]：

[1] 3.3 版新加入.

[2] 3.9 版更變: 此方式现在适用于除了非整数乘方以外的所有精确结果。

```
>>> setcontext(Context(prec=MAX_PREC, Emax=MAX_EMAX, Emin=MIN_EMIN))
>>> x = Decimal(2) ** 256
>>> x / 128
Decimal('904625697166532776746648320380374280103671755200316906558262375061821325312')
```

对于不精确的结果，在 64 位平台上*MAX_PREC* 的值太大了，可用的内存将会不足:

```
>>> Decimal(1) / 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

在具有超量分配的系统上 (即 Linux)，一种更复杂的方式根据可用的 RAM 大小来调整 `prec`。假设你有 8GB 的 RAM 并期望同时有 10 个操作数，每个最多使用 500MB:

```
>>> import sys
>>>
>>> # Maximum number of digits for a single operand using 500MB in 8-byte words
>>> # with 19 digits per word (4-byte and 9 digits for the 32-bit build):
>>> maxdigits = 19 * ((500 * 1024**2) // 8)
>>>
>>> # Check that this works:
>>> c = Context(prec=maxdigits, Emax=MAX_EMAX, Emin=MIN_EMIN)
>>> c.traps[Inexact] = True
>>> setcontext(c)
>>>
>>> # Fill the available precision with nines:
>>> x = Decimal(0).logical_invert() * 9
>>> sys.getsizeof(x)
524288112
>>> x + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  decimal.Inexact: [<class 'decimal.Inexact'>]
```

总体而言（特别是在没有超量分配的系统上），如果期望所有计算都是精确的则推荐预估更严格的边界并设置*Inexact* 陷阱。

9.5 fractions --- 分数

源代码 Lib/fractions.py

---

fractions 模块支持分数运算。

分数实例可以由一对整数，一个分数，或者一个字符串构建而成。

class fractions.**Fraction**(*numerator=0*, *denominator=1*)
class fractions.**Fraction**(*other_fraction*)
class fractions.**Fraction**(*float*)
class fractions.**Fraction**(*decimal*)
class fractions.**Fraction**(*string*)

第一个版本要求 *numerator* 和 *denominator* 是*numbers.Rational* 的实例，并返回一个新的*Fraction* 实例，其值为 `numerator/denominator`。如果 *denominator* 为 `0` 将会引发*ZeroDivisionError*。第二个版本要求 *other_fraction* 是*numbers.Rational* 的实例，并返回一个*Fraction* 实例且与传入

值相等。下两个版本接受*float* 或*decimal.Decimal* 的实例，并返回一个*Fraction* 实例且与传入值完全相等。请注意由于二进制浮点数通常存在的问题 (参见 tut-fp-issues)，`Fraction(1.1)` 的参数并不会精确等于 11/10，因此 `Fraction(1.1)` 也 不会返回用户所期望的 `Fraction(11, 10)`。(请参阅下文中*limit_denominator()* 方法的文档。) 构造器的最后一个版本接受一个字符串或 unicode 实例。此实例的通常形式为:

```
[sign] numerator ['/' denominator]
```

其中的可选项 `sign` 可以为'+' 或'-' 并且 `numerator` 和 `denominator` (如果存在) 是十进制数码的字符串。此外，*float* 构造器所接受的任何表示一个有限值的字符串也都为*Fraction* 构造器所接受。不论哪种形式的输入字符串也都可以带有前缀和/或后缀的空格符。这里是一些示例:

```
>>> from fractions import Fraction
>>> Fraction(16, -10)
Fraction(-8, 5)
>>> Fraction(123)
Fraction(123, 1)
>>> Fraction()
Fraction(0, 1)
>>> Fraction('3/7')
Fraction(3, 7)
>>> Fraction(' -3/7 ')
Fraction(-3, 7)
>>> Fraction('1.414213 \t\n')
Fraction(1414213, 1000000)
>>> Fraction('-.125')
Fraction(-1, 8)
>>> Fraction('7e-6')
Fraction(7, 1000000)
>>> Fraction(2.25)
Fraction(9, 4)
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
>>> from decimal import Decimal
>>> Fraction(Decimal('1.1'))
Fraction(11, 10)
```

Fraction 类继承自抽象基类*numbers.Rational*，并实现了该类的所有方法和操作。*Fraction* 实例是可哈希的，并应当被视为不可变对象。此外，*Fraction* 还具有以下属性和方法:

3.2 版更變: *Fraction* 构造器现在接受*float* 和*decimal.Decimal* 实例。

3.9 版更變: 现在会使用*math.gcd()* 函数来正规化 *numerator* 和 *denominator*。*math.gcd()* 总是返回*int* 类型。在之前版本中，GCD 的类型取决于 *numerator* 和 *denominator* 的类型。

numerator
最简分数形式的分子。

denominator
最简分数形式的分母。

as_integer_ratio()
返回由两个整数组成的元组，两数之比等于该分数的值且其分母为正数。

3.8 版新加入.

from_float(*flt*)
此类方法可构造一个*Fraction* 来表示 *flt* 的精确值，该参数必须是一个*float*。请注意 `Fraction.from_float(0.3)` 的值并不等于 `Fraction(3, 10)`。

備𝔽: 从 Python 3.2 开始，在构造*Fraction* 实例时可以直接使用*float*。

from_decimal(*dec*)
此类方法可构造一个*Fraction* 来表示 *dec* 的精确值，该参数必须是一个*decimal.Decimal* 实例。

備𝔽: 从 Python 3.2 开始，在构造*Fraction* 实例时可以直接使用*decimal.Decimal* 实例。

limit_denominator(*max_denominator=1000000*)
找到并返回一个*Fraction* 使得其值最接近 `self` 并且分母不大于 max_denominator。此方法适用于找出给定浮点数的有理数近似值:

```
>>> from fractions import Fraction
>>> Fraction('3.1415926535897932').limit_denominator(1000)
Fraction(355, 113)
```

或是用来恢复被表示为一个浮点数的有理数:

```
>>> from math import pi, cos
>>> Fraction(cos(pi/3))
Fraction(4503599627370497, 9007199254740992)
>>> Fraction(cos(pi/3)).limit_denominator()
Fraction(1, 2)
>>> Fraction(1.1).limit_denominator()
Fraction(11, 10)
```

__floor__()
返回最大的*int* `<= self`。此方法也可通过*math.floor()* 函数来使用:

```
>>> from math import floor
>>> floor(Fraction(355, 113))
3
```

__ceil__()
返回最小的*int* `>= self`。此方法也可通过*math.ceil()* 函数来使用。

__round__()
__round__(*ndigits*)
第一个版本返回一个*int* 使得其值最接近 `self`，位值为二分之一时只对偶数舍入。第二个版本会将 `self` 舍入到最接近 `Fraction(1, 10**ndigits)` 的倍数（如果 `ndigits` 为负值则为逻辑运算），位值为二分之一时同样只对偶数舍入。此方法也可通过*round()* 函数来使用。

也参考:

***numbers* 模块** 构成数字塔的所有抽象基类。

9.6 random --- 生成伪随机数

源码： Lib/random.py

该模块实现了各种分布的伪随机数生成器。

对于整数，从范围中有统一的选择。对于序列，存在随机元素的统一选择、用于生成列表的随机排列的函数、以及用于随机抽样而无需替换的函数。

在实数轴上，有计算均匀、正态（高斯）、对数正态、负指数、伽马和贝塔分布的函数。为了生成角度分布，可以使用 von Mises 分布。

几乎所有模块函数都依赖于基本函数*random()*，它在半开放区间 [0.0,1.0) 内均匀生成随机浮点数。Python 使用 Mersenne Twister 作为核心生成器。它产生 53 位精度浮点数，周期为 2**19937-1，其在 C 中的底层实现既快又线程安全。Mersenne Twister 是现存最广泛测试的随机数发生器之一。但是，因为完全确定性，它不适用于所有目的，并且完全不适合加密目的。

这个模块提供的函数实际上是*random.Random*类的隐藏实例的绑定方法。你可以实例化自己的*Random*类实例以获取不共享状态的生成器。

如果你想使用自己设计的不同基础生成器，类*Random*也可以作为子类：在这种情况下，重载 `random()`、`seed()`、`getstate()` 以及 `setstate()` 方法。可选地，新生成器可以提供 `getrandbits()` 方法——这允许*randrange()*在任意大的范围内产生选择。

*random*模块还提供*SystemRandom*类，它使用系统函数*os.urandom()*从操作系统提供的源生成随机数。

> **警告：** 不应将此模块的伪随机生成器用于安全目的。有关安全性或加密用途，请参阅*secrets*模块。

也参考：

M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3--30 1998.

Complementary-Multiply-with-Carry recipe 用于兼容的替代随机数发生器，具有长周期和相对简单的更新操作。

9.6.1 簿记功能

`random.`**`seed`**`(`*a=None*, *version=2*`)`

初始化随机数生成器。

如果 *a* 被省略或为 `None`，则使用当前系统时间。如果操作系统提供随机源，则使用它们而不是系统时间（有关可用性的详细信息，请参阅*os.urandom()*函数）。

如果 *a* 是 int 类型，则直接使用。

对于版本 2（默认的），*str*、*bytes*或*bytearray*对象转换为*int*并使用它的所有位。

对于版本 1（用于从旧版本的 Python 再现随机序列），用于*str*和*bytes*的算法生成更窄的种子范围。

3.2 版更變: 已移至版本 2 方案，该方案使用字符串种子中的所有位。

3.9 版後已🄴用: 在将来，*seed* 必须是下列类型之一: *NoneType*, *int*, *float*, *str*, *bytes*或*bytearray*。

`random.`**`getstate`**`()`

返回捕获生成器当前内部状态的对象。这个对象可以传递给*setstate()*来恢复状态。

random.**setstate**(*state*)
state 应该是从之前调用*getstate()* 获得的，并且*setstate()* 将生成器的内部状态恢复到*getstate()* 被调用时的状态。

9.6.2 用于字节数据的函数

random.**randbytes**(*n*)
生成 *n* 个随机字节。

此方法不可用于生成安全凭据。那应当使用*secrets.token_bytes()*。

3.9 版新加入.

9.6.3 整数用函数

random.**randrange**(*stop*)
random.**randrange**(*start*, *stop*[, *step*])
从 `range(start, stop, step)` 返回一个随机选择的元素。这相当于 `choice(range(start, stop, step))`，但实际上并没有构建一个 range 对象。

位置参数模式匹配*range()* 。不应使用关键字参数，因为该函数可能以意外的方式使用它们。

3.2 版更變: *randrange()* 在生成均匀分布的值方面更为复杂。以前它使用了像 “int(random()*n)“这样的形式，它可以产生稍微不均匀的分布。

random.**randint**(*a*, *b*)
返回随机整数 *N* 满足 `a <= N <= b`。相当于 `randrange(a, b+1)`。

random.**getrandbits**(*k*)
返回具有 *k* 个随机比特位的非负 Python 整数。此方法随 MersenneTwister 生成器一起提供，其他一些生成器也可能将其作为 API 的可选部分提供。在可能的情况下，*getrandbits()* 会启用*randrange()* 来处理任意大的区间。

3.9 版更變: 此方法现在接受零作为 *k* 的值。

9.6.4 序列用函数

random.**choice**(*seq*)
从非空序列 *seq* 返回一个随机元素。如果 *seq* 为空，则引发*IndexError*。

random.**choices**(*population*, *weights=None*, *, *cum_weights=None*, *k=1*)
从 *population* 中选择替换，返回大小为 *k* 的元素列表。如果 *population* 为空，则引发*IndexError*。

如果指定了 *weight* 序列，则根据相对权重进行选择。或者，如果给出 *cum_weights* 序列，则根据累积权重（可能使用*itertools.accumulate()* 计算）进行选择。例如，相对权重 “[10, 5, 30, 5]“相当于累积权重 “[10, 15, 45, 50]“。在内部，相对权重在进行选择之前会转换为累积权重，因此提供累积权重可以节省工作量。

如果既未指定 *weight* 也未指定 *cum_weights* ，则以相等的概率进行选择。如果提供了权重序列，则它必须与 *population* 序列的长度相同。一个*TypeError* 指定了 *weights* 和 *cum_weights*。

weights 或 *cum_weights* 可使用*random()* 所返回的能与*float* 值进行相互运算的任何数字类型（包括 int、float、Fraction 但不包括 Decimal）。权重为负值的行为未有定义。如果权重为负值则将引发*ValueError*。

对于给定的种子，具有相等加权的*choices()* 函数通常产生与重复调用*choice()* 不同的序列。*choices()* 使用的算法使用浮点运算来实现内部一致性和速度。*choice()* 使用的算法默认为重复选择的整数运算，以避免因舍入误差引起的小偏差。

3.6 版新加入.

3.9 版更變: 如果所有权重均为负值则将引发*ValueError*。

random.**shuffle**(*x*[, *random*])
将序列 *x* 随机打乱位置。

可选参数 *random* 是一个 0 参数函数，在 [0.0, 1.0) 中返回随机浮点数；默认情况下，这是函数*random()* 。

要改变一个不可变的序列并返回一个新的打乱列表，请使用 “sample(x, k=len(x))“。

请注意，即使对于小的 `len(x)`，*x* 的排列总数也可以快速增长，大于大多数随机数生成器的周期。这意味着长序列的大多数排列永远不会产生。例如，长度为 2080 的序列是可以在 Mersenne Twister 随机数生成器的周期内拟合的最大序列。

Deprecated since version 3.9, will be removed in version 3.11: 可选形参 *random*。

random.**sample**(*population*, *k*, *, *counts=None*)
返回从总体序列或集合中选择的唯一元素的 *k* 长度列表。用于无重复的随机抽样。

返回包含来自总体的元素的新列表，同时保持原始总体不变。结果列表按选择顺序排列，因此所有子切片也将是有效的随机样本。这允许抽奖获奖者（样本）被划分为大奖和第二名获胜者（子切片）。

总体成员不必是*hashable* 或 unique 。如果总体包含重复，则每次出现都是样本中可能的选择。

重复的元素可以一个个地直接列出，或使用可选的仅限关键字形参 *counts* 来指定。例如，`sample(['red', 'blue'], counts=[4, 2], k=5)` 等价于 `sample(['red', 'red', 'red', 'red', 'blue', 'blue'], k=5)`。

要从一系列整数中选择样本，请使用*range()* 对象作为参数。对于从大量人群中采样，这种方法特别快速且节省空间：`sample(range(10000000), k=60)` 。

如果样本大小大于总体大小，则引发*ValueError* 。

3.9 版更變: 增加了 *counts* 形参。

3.9 版後已🄵用: 在将来，*population* 必须是一个序列。*set* 的实例将不再被支持。集合必须先转换为*list* 或*tuple*，最好是固定顺序以使抽样是可重现的。

9.6.5 实值分布

以下函数生成特定的实值分布。如常用数学实践中所使用的那样, 函数参数以分布方程中的相应变量命名; 大多数这些方程都可以在任何统计学教材中找到。

random.**random**()
返回 [0.0, 1.0) 范围内的下一个随机浮点数。

random.**uniform**(*a*, *b*)
返回一个随机浮点数 *N* ，当 `a <= b` 时 `a <= N <= b` ，当 `b < a` 时 `b <= N <= a` 。

取决于等式 `a + (b-a) * random()` 中的浮点舍入，终点 `b` 可以包括或不包括在该范围内。

random.**triangular**(*low*, *high*, *mode*)
返回一个随机浮点数 *N* ，使得 `low <= N <= high` 并在这些边界之间使用指定的 *mode* 。*low* 和 *high* 边界默认为零和一。*mode* 参数默认为边界之间的中点，给出对称分布。

random.**betavariate**(*alpha*, *beta*)
Beta 分布。参数的条件是 `alpha > 0` 和 `beta > 0`。返回值的范围介于 0 和 1 之间。

random.**expovariate**(*lambd*)
指数分布。*lambd* 是 1.0 除以所需的平均值，它应该是非零的。（该参数本应命名为 "lambda"，但这是 Python 中的保留字。）如果 *lambd* 为正，则返回值的范围为 0 到正无穷大；如果 *lambd* 为负，则返回值从负无穷大到 0。

random.**gammavariate**(*alpha*, *beta*)
Gamma 分布。（不是 gamma 函数!）参数的条件是 `alpha > 0` 和 `beta > 0`。

概率分布函数是:

```
          x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu*, *sigma*)
高斯分布。*mu* 是平均值，*sigma* 是标准差。这比下面定义的*normalvariate()* 函数略快。

多线程注意事项：当两个线程同时调用此方法时，它们有可能将获得相同的返回值。这可以通过三种办法来避免。1) 让每个线程使用不同的随机数生成器实例。2) 在所有调用外面加锁。3) 改用速度较慢但是线程安全的*normalvariate()* 函数。

random.**lognormvariate**(*mu*, *sigma*)
对数正态分布。如果你采用这个分布的自然对数，你将得到一个正态分布，平均值为 *mu* 和标准差为 *sigma* 。*mu* 可以是任何值，*sigma* 必须大于零。

random.**normalvariate**(*mu*, *sigma*)
正态分布。*mu* 是平均值，*sigma* 是标准差。

random.**vonmisesvariate**(*mu*, *kappa*)
冯 · 米塞斯分布。*mu* 是平均角度，以弧度表示，介于 0 和 2**pi* 之间，*kappa* 是浓度参数，必须大于或等于零。如果 *kappa* 等于零，则该分布在 0 到 2**pi* 的范围内减小到均匀的随机角度。

random.**paretovariate**(*alpha*)
帕累托分布。*alpha* 是形状参数。

random.**weibullvariate**(*alpha*, *beta*)
威布尔分布。*alpha* 是比例参数，*beta* 是形状参数。

9.6.6 替代生成器

class random.**Random**([*seed*])
该类实现了*random* 模块所用的默认伪随机数生成器。

3.9 版後已F用: 在将来，*seed* 必须是下列类型之一: NoneType, *int*, *float*, *str*, *bytes* 或*bytearray*。

class random.**SystemRandom**([*seed*])
使用*os.urandom()* 函数的类，用从操作系统提供的源生成随机数。这并非适用于所有系统。也不依赖于软件状态，序列不可重现。因此，*seed()* 方法没有效果而被忽略。*getstate()* 和*setstate()* 方法如果被调用则引发*NotImplementedError*。

9.6.7 关于再现性的说明

有时能够重现伪随机数生成器给出的序列是很有用处的。通过重用一个种子值，只要没有运行多线程，相同的序列就应当可在多次运行中重现。

大多数随机模块的算法和种子函数都会在 Python 版本中发生变化，但保证两个方面不会改变：

- 如果添加了新的播种方法，则将提供向后兼容的播种机。
- 当兼容的播种机被赋予相同的种子时，生成器的 `random()` 方法将继续产生相同的序列。

9.6.8 例子

基本示例:

```
>>> random()                            # Random float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                  # Random float:  2.5 <= x <= 10.0
3.1800146073117523

>>> expovariate(1 / 5)                  # Interval between arrivals averaging 5␣
↪seconds
5.148957571865031

>>> randrange(10)                       # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)                # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])     # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                       # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)   # Four samples without replacement
[40, 10, 50, 30]
```

模拟:

```
>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck
>>> # of 52 playing cards, and determine the proportion of cards
>>> # with a ten-value:  ten, jack, queen, or king.
>>> dealt = sample(['tens', 'low cards'], counts=[16, 36], k=20)
>>> dealt.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
```

(下页继续)

(繼續上一頁)

```
>>> def trial():
...     return choices('HT', cum_weights=(0.60, 1.00), k=7).count('H') >= 5
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> def trial():
...     return 2_500 <= sorted(choices(range(10_000), k=5))[2] < 7_500
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.7958
```

statistical bootstrapping 的示例，使用重新采样和替换来估计一个样本的均值的置信区间:

```
# http://statistics.about.com/od/Applications/a/Example-Of-Bootstrapping.htm
from statistics import fmean as mean
from random import choices

data = [41, 50, 29, 37, 81, 30, 73, 63, 20, 35, 68, 22, 60, 31, 95]
means = sorted(mean(choices(data, k=len(data))) for i in range(100))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[5]:.1f} to {means[94]:.1f}')
```

使用 重新采样排列测试 来确定统计学显著性或者使用 p-值 来观察药物与安慰剂的作用之间差异的示例:

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import fmean as mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10_000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

多服务器队列的到达时间和服务交付模拟:

```
from heapq import heapify, heapreplace
from random import expovariate, gauss
from statistics import mean, median, stdev

average_arrival_interval = 5.6
average_service_time = 15.0
stdev_service_time = 3.5
```

(下页继续)

(繼續上一頁)

```
num_servers = 3

waits = []
arrival_time = 0.0
servers = [0.0] * num_servers  # time when each server becomes available
heapify(servers)
for i in range(1_000_000):
    arrival_time += expovariate(1.0 / average_arrival_interval)
    next_server_available = servers[0]
    wait = max(0.0, next_server_available - arrival_time)
    waits.append(wait)
    service_duration = max(0.0, gauss(average_service_time, stdev_service_time))
    service_completed = arrival_time + wait + service_duration
    heapreplace(servers, service_completed)

print(f'Mean wait: {mean(waits):.1f}.  Stdev wait: {stdev(waits):.1f}.')
print(f'Median wait: {median(waits):.1f}.  Max wait: {max(waits):.1f}.')
```

也參考:

Statistics for Hackers Jake Vanderplas 撰写的视频教程，使用一些基本概念进行统计分析，包括模拟、抽样、改组和交叉验证。

Economics Simulation Peter Norvig 编写的市场模拟，显示了该模块提供的许多工具和分布的有效使用（高斯、均匀、样本、beta 变量、选择、三角和随机范围等）。

A Concrete Introduction to Probability (using Python) Peter Norvig 撰写的教程，涵盖了概率论基础知识，如何编写模拟，以及如何使用 Python 进行数据分析。

9.6.9 例程

默认的`random()` 返回在 *0.0 ≤ x < 1.0* 范围内 2^{-53} 的倍数。所有这些数值间隔相等并能精确表示为 Python 浮点数。但是在此间隔上有许多其他可表示浮点数是不可能的选择。例如，`0.05954861408025609` 就不是 2^{-53} 的整数倍。

以下规范程序采取了一种不同的方式。在间隔上的所有浮点数都是可能的选择。它们的尾数取值来自 $2^{52} \leq$ *尾数* $< 2^{53}$ 范围内整数的均匀分布。指数取值则来自几何分布，其中小于 *-53* 的指数的出现频率为下一个较大指数的一半。

```
from random import Random
from math import ldexp

class FullRandom(Random):

    def random(self):
        mantissa = 0x10_0000_0000_0000 | self.getrandbits(52)
        exponent = -53
        x = 0
        while not x:
            x = self.getrandbits(32)
            exponent += x.bit_length() - 32
        return ldexp(mantissa, exponent)
```

该类中所有的实值分布 都将使用新的方法:

```
>>> fr = FullRandom()
>>> fr.random()
0.05954861408025609
>>> fr.expovariate(0.25)
8.87925541791544
```

该规范程序在概念上等效于在 *0.0 ≤ x < 1.0* 范围内对所有 2^{-1074} 的倍数进行选择的算法。所有这样的数字间隔都相等，但大多必须向下舍入为最接近的 Python 浮点数表示形式。(2^{-1074} 这个数值是等于 `math.ulp(0.0)` 的未经正规化的最小正浮点数。)

也参考:

生成伪随机浮点数值 为 Allen B. Downey 所撰写的描述如何生成相比通过*`random()`* 正常生成的数值更细粒度浮点数的论文。

9.7 `statistics` --- 数學統計函式

3.4 版新加入.

Source code: Lib/statistics.py

该模块提供了用于计算数字 (*`Real`*-valued) 数据的数理统计量的函数。

此模块并不是诸如 NumPy， SciPy 等第三方库或者诸如 Minitab， SAS， Matlab 等针对专业统计学家的专有全功能统计软件包的竞品。此模块针对图形和科学计算器的水平。

除非明确注释，这些函数支持*`int`*， *`float`*， *`Decimal`* 和*`Fraction`*。当前不支持同其他类型（是否在数字塔中）的行为。混合类型的集合也是未定义的，并且依赖于实现。如果你输入的数据由混合类型组成，你应该能够使用*`map()`* 来确保一个一致的结果，比如: `map(float, input_data)`。

9.7.1 平均值與中央位置量數

這些函式計算來自一個母體或樣本的平均值或代表值。

`mean()`	數據的算術平均（平均值）。
`fmean()`	快速的，浮点算数平均数。
`geometric_mean()`	数据的几何平均数
`harmonic_mean()`	数据的调和均值
`median()`	數據的中位數（中間值）。
`median_low()`	數據中較小的中位數。
`median_high()`	數據中較大的中位數。
`median_grouped()`	分組數據的中位數或 50% 處。
`mode()`	离散的或标称的数据的单个众数（出现最多的值）。
`multimode()`	离散的或标称的数据的众数列表（出现最多的值）。
`quantiles()`	将数据以相等的概率分为多个间隔。

9.7.2 離度 (spread) 的測量

這些函式計算母體或樣本偏離平均值的程度。

pstdev()	數據的母體標准差
pvariance()	數據的母體變🄵數
stdev()	數據的樣本標准差
variance()	數據的樣本變🄵數

9.7.3 函式細節

注释：这些函数不需要对提供给它们的数据进行排序。但是，为了方便阅读，大多数例子展示的是已排序的序列。

statistics.**mean**(*data*)

返回 *data* 的样本算术平均数，形式为序列或迭代器。

算术平均数是数据之和与数据点个数的商。通常称作“平均数”，尽管它指示诸多数学平均数之一。它是数据的中心位置的度量。

若 *data* 为空，将会引发*StatisticsError*。

一些用法示例：

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

備🄵: The mean is strongly affected by outliers and is not necessarily a typical example of the data points. For a more robust, although less efficient, measure of central tendency, see *median()*.

样本均值给出了一个无偏向的真实总体均值的估计，因此当平均抽取所有可能的样本，`mean(sample)` 收敛于整个总体的真实均值。如果 *data* 代表整个总体而不是样本，那么 `mean(data)` 等同于计算真实整体均值 μ 。

---

statistics.**fmean**(*data*)

将 *data* 转换成浮点数并且计算算术平均数。

此函数的运行速度比*mean()* 函数快并且它总是返回一个*float*。*data* 可以为序列或可迭代对象。如果输入数据集为空，则会引发*StatisticsError*。

```
>>> fmean([3.5, 4.0, 5.25])
4.25
```

3.8 版新加入.

statistics.**geometric_mean**(*data*)
将 *data* 转换成浮点数并且计算几何平均数。

几何平均值使用值的乘积表示 数据的中心趋势或典型值（与使用它们的总和的算术平均值相反）。

如果输入数据集为空、包含零或包含负值则将引发*StatisticsError*。*data* 可以是序列或可迭代对象。

无需做出特殊努力即可获得准确的结果。（但是，将来或许会修改。）

```
>>> round(geometric_mean([54, 24, 36]), 1)
36.0
```

3.8 版新加入.

statistics.**harmonic_mean**(*data*)
返回 *data* 调和均值，该参数可以是序列或包含实数值的可迭代对象。

调和均值, 也叫次相反均值，所有数据的倒数的算术平均数*mean()* 的倒数。比如说，数据 *a* ， *b* ， *c* 的调和均值等于 `3/(1/a + 1/b + 1/c)` 。如果其中一个值为零，结果为零。

调和均值是一种均值类型，是数据中心位置的度量。它通常适合于求比率和比例的平均值，比如速率。

假设一辆车在 40 km/hr 的速度下行驶了 10 km ， 然后又以 60 km/hr 的速度行驶了 10 km 。车辆的平均速率是多少?

```
>>> harmonic_mean([40, 60])
48.0
```

假设一名投资者在三家公司各购买了等价值的股票，以 2.5，3 ， 10 的 P/E (价格/收益) 率。投资者投资组合的平均市盈率是多少?

```
>>> harmonic_mean([2.5, 3, 10])  # For an equal investment portfolio.
3.6
```

如果 *data* 为空或者任何一个元素的值小于零，会引发*StatisticsError* 。

当前算法在输入中遇到零时会提前退出。这意味着不会测试后续输入的有效性。（此行为将来可能会更改。）

3.6 版新加入.

statistics.**median**(*data*)
使用普通的 “取中间两数平均值” 方法返回数值数据的中位数（中间值）。如果 *data* 为空，则将引发*StatisticsError*。*data* 可以是序列或可迭代对象。

中位数是衡量中间位置的可靠方式，并且较少受到极端值的影响。当数据点的总数为奇数时，将返回中间数据点：

```
>>> median([1, 3, 5])
3
```

当数据点的总数为偶数时，中位数将通过对两个中间值求平均进行插值得出：

```
>>> median([1, 3, 5, 7])
4.0
```

这适用于当你的数据是离散的，并且你不介意中位数不是实际数据点的情况。

如果数据是有序的（支持排序操作）但不是数字（不支持加法），请考虑改用*median_low()* 或*median_high()*。

statistics.**median_low**(*data*)

返回数值数据的低中位数。如果 *data* 为空则将引发*StatisticsError*。*data* 可以是序列或可迭代对象。

低中位数一定是数据集的成员。当数据点总数为奇数时，将返回中间值。当其为偶数时，将返回两个中间值中较小的那个。

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

当你的数据是离散的，并且你希望中位数是一个实际数据点而非插值结果时可以使用低中位数。

statistics.**median_high**(*data*)

返回数据的高中位数。如果 *data* 为空则将引发*StatisticsError*。*data* 可以是序列或可迭代对象。

高中位数一定是数据集的成员。当数据点总数为奇数时，将返回中间值。当其为偶数时，将返回两个中间值中较大的那个。

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

当你的数据是离散的，并且你希望中位数是一个实际数据点而非插值结果时可以使用高中位数。

statistics.**median_grouped**(*data*, *interval=1*)

返回分组的连续数据的中位数，根据第 50 个百分点的位置使用插值来计算。如果 *data* 为空则将引发*StatisticsError*。*data* 可以是序列或可迭代对象。

```
>>> median_grouped([52, 52, 53, 54])
52.5
```

在下面的示例中，数据已经过舍入，这样每个值都代表数据分类的中间点，例如 1 是 0.5--1.5 分类的中间点，2 是 1.5--2.5 分类的中间点，3 是 2.5--3.5 的中间点等待。根据给定的数据，中间值应落在 3.5--4.5 分类之内，并可使用插值法来进行估算：

```
>>> median_grouped([1, 2, 2, 3, 4, 4, 4, 4, 4, 5])
3.7
```

可选参数 *interval* 表示分类间隔，默认值为 1。改变分类间隔自然会改变插件结果：

```
>>> median_grouped([1, 3, 3, 5, 7], interval=1)
3.25
>>> median_grouped([1, 3, 3, 5, 7], interval=2)
3.5
```

此函数不会检查数据点之间是否至少相隔 *interval* 的距离。

CPython implementation detail: 在某些情况下，*median_grouped()* 可以会将数据点强制转换为浮点数。此行为在未来有可能会发生改变。

也参考:

- ”Statistics for the Behavioral Sciences”, Frederick J Gravetter and Larry B Wallnau (8th Edition).
- Gnome Gnumeric 电子表格中的 SSMEDIAN 函数，包括 这篇讨论。

statistics.**mode**(*data*)

从离散或标称的 *data* 返回单个出现最多的数据点。此众数（如果存在）是最典型的值，并可用来度量中心的位置。

如果存在具有相同频率的多个众数，则返回在 *data* 中遇到的第一个。如果想要其中最小或最大的一个，请使用 `min(multimode(data))` 或 `max(multimode(data))`。如果输入的 *data* 为空，则会引发*StatisticsError*。

`mode` 将假定是离散数据并返回一个单一的值。这是通常的学校教学中标准的处理方式：

```
>>> mode([1, 1, 2, 3, 3, 3, 3, 4])
3
```

此众数的独特之处在于它是这个包中唯一还可应用于标称（非数字）数据的统计信息：

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

3.8 版更變: 现在会通过返回所遇到的第一个众数来处理多模数据集。之前它会在遇到超过一个的众数时引发*StatisticsError*。

statistics.**multimode**(*data*)

返回最频繁出现的值的列表，并按它们在 *data* 中首次出现的位置排序。如果存在多个众数则将返回一个以上的众数，或者如果 *data* 为空则将返回空列表：

```
>>> multimode('aabbbbccddddeeffffgg')
['b', 'd', 'f']
>>> multimode('')
[]
```

3.8 版新加入.

statistics.**pstdev**(*data*, *mu=None*)

返回总体标准差（总体方差的平方根）。请参阅*pvariance()* 了解参数和其他细节。

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

statistics.**pvariance**(*data*, *mu=None*)

返回非空序列或包含实数值的可迭代对象 *data* 的总体方差。方差或称相对于均值的二阶距，是对数据变化幅度（延展度或分散度）的度量。方差值较大表明数据的散布范围较大；方差值较小表明它紧密聚集于均值附近。

如果给出了可选的第二个参数 *mu*，它通常是 *data* 的均值。它也可以被用来计算相对于一个非均值点的二阶距。如果该参数省略或为 `None` (默认值)，则会自动进行算术均值的计算。

使用此函数可根据所有数值来计算方差。要根据一个样本来估算方差，通常*variance()* 函数是更好的选择。

如果 *data* 为空则会引发*StatisticsError*。

示例：

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

如果你已经计算过数据的平均值，你可以将其作为可选的第二个参数 *mu* 传入以避免重复计算：

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

同样也支持使用 Decimal 和 Fraction 值：

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

備𝔽： 当调用时附带完整的总体数据时，这将给出总体方差 σ^2。而当调用时只附带一个样本时，这将给出偏置样本方差 s^2，也被称为带有 N 个自由度的方差。

如果你通过某种方式知道了真实的总体平均值 μ，则可以使用此函数来计算一个样本的方差，并将已知的总体平均值作为第二个参数。假设数据点是总体的一个随机样本，则结果将为总体方差的无偏估计值。

---

statistics.**stdev**(*data*, *xbar=None*)

返回样本标准差（样本方差的平方根）。请参阅*variance()* 了解参数和其他细节。

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.**variance**(*data*, *xbar=None*)

返回包含至少两个实数值的可迭代对象 *data* 的样本方差。方差或称相对于均值的二阶矩，是对数据变化幅度（延展度或分散度）的度量。方差值较大表明数据的散布范围较大；方差值较小表明它紧密聚集于均值附近。

如果给出了可选的第二个参数 *xbar*，它应当是 *data* 的均值。如果该参数省略或为 None (默认值)，则会自动进行均值的计算。

当你的数据是总体数据的样本时请使用此函数。要根据整个总体数据来计算方差，请参见*pvariance()*。

如果 *data* 包含的值少于两个则会引发*StatisticsError*。

示例：

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

如果你已经计算过数据的平均值，你可以将其作为可选的第二个参数 *xbar* 传入以避免重复计算：

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

此函数不会试图检查你所传入的 *xbar* 是否为真实的平均值。使用任意值作为 *xbar* 可能导致无效或不可能的结果。

同样也支持使用 Decimal 和 Fraction 值：

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

備𝔽: 这是附带贝塞尔校正的样本方差 s^2，也称为具有 N-1 自由度的方差。假设数据点具有代表性（即为独立且均匀的分布），则结果应当是对总体方差的无偏估计。

如果你通过某种方式知道了真实的总体平均值 μ 则应当调用*pvariance()* 函数并将该值作为 *mu* 形参传入以得到一个样本的方差。

---

statistics.**quantiles**(*data*, *, *n=4*, *method='exclusive'*)

将 *data* 分隔为具有相等概率的 *n* 个连续区间。返回分隔这些区间的 `n - 1` 个分隔点的列表。

将 *n* 设为 4 以使用四分位（默认值）。将 *n* 设为 10 以使用十分位。将 *n* 设为 100 以使用百分位，即给出 99 个分隔点来将 *data* 分隔为 100 个大小相等的组。如果 *n* 小于 1 则将引发*StatisticsError*。

data 可以是包含样本数据的任意可迭代对象。为了获得有意义的结果，*data* 中数据点的数量应当大于 *n*。如果数据点的数量小于两个则将引发*StatisticsError*。

分隔点是通过对两个最接近的数据点进行线性插值得到的。例如，如果一个分隔点落在两个样本值 `100` 和 `112` 之间距离三分之一的位置，则分隔点的取值将为 `104`。

method 用于计算分位值，它会由于 *data* 是包含还是排除总体的最低和最高可能值而有所不同。

默认 *method* 是“唯一的”并且被用于在总体中数据采样这样可以有比样本中找到的更多的极端值。落在 *m* 个排序数据点的第 *i-th* 个以下的总体部分被计算为 `i / (m + 1)`。给定九个样本值，方法排序它们并且分配一下的百分位：10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% 。

将 *method* 设为”inclusive” 可用于描述总体数据或已明确知道包含有总体数据中最极端值的样本。*data* 中的最小值会被作为第 0 个百分位而最大值会被作为第 100 个百分位。总体数据里处于 *m* 个已排序数据点中 第 *i* 个以下的部分会以 `(i - 1) / (m - 1)` 来计算。给定 11 个样本值，该方法会对它们进行排序并赋予以下百分位: 0%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%。

```
# Decile cut points for empirically sampled data
>>> data = [105, 129, 87, 86, 111, 111, 89, 81, 108, 92, 110,
...         100, 75, 105, 103, 109, 76, 119, 99, 91, 103, 129,
...         106, 101, 84, 111, 74, 87, 86, 103, 103, 106, 86,
...         111, 75, 87, 102, 121, 111, 88, 89, 101, 106, 95,
...         103, 107, 101, 81, 109, 104]
>>> [round(q, 1) for q in quantiles(data, n=10)]
[81.0, 86.2, 89.0, 99.4, 102.5, 103.6, 106.0, 109.8, 111.0]
```

3.8 版新加入.

9.7.4 异常

只定义了一个异常：

exception statistics.**StatisticsError**
ValueError 的子类，表示统计相关的异常。

9.7.5 NormalDist 对象

NormalDist 工具可用于创建和操纵 随机变量 的正态分布。这个类将数据度量值的平均值和标准差作为单一实体来处理。

正态分布的概念来自于 中央极限定理 并且在统计学中有广泛的应用。

class statistics.**NormalDist**(*mu=0.0*, *sigma=1.0*)
返回一个新的 *NormalDist* 对象，其中 *mu* 代表 算术平均值 而 *sigma* 代表 标准差。

若 *sigma* 为负数，将会引发*StatisticsError*。

mean
一个只读特征属性，表示特定正态分布的 算术平均值。

median
一个只读特征属性，表示特定正态分布的 中位数。

mode
一个只读特征属性，表示特定正态分布的 众数。

stdev
一个只读特征属性，表示特定正态分布的 标准差。

variance
一个只读特征属性，表示特定正态分布的 方差。等于标准差的平方。

classmethod from_samples(*data*)
传入使用*fmean()* 和*stdev()* 基于 *data* 估算出的 *mu* 和 *sigma* 形参创建一个正态分布实例。

data 可以是任何*iterable* 并且应当包含能被转换为*float* 类型的值。如果 *data* 不包含至少两个元素，则会引发*StatisticsError*，因为估算中心值至少需要一个点而估算分散度至少需要两个点。

samples(*n*, *, *seed=None*)
对于给定的平均值和标准差生成 *n* 个随机样本。返回一个由*float* 值组成的*list*。

当给定 *seed* 时，创建一个新的底层随机数生成器实例。这适用于创建可重现的结果，即使对于多线程上下文也有效。

pdf(*x*)
使用 概率密度函数 (pdf)，计算一个随机变量 *X* 趋向于给定值 *x* 的相对可能性。在数学意义上，它是当 *dx* 趋向于零时比率 `P(x <= X < x+dx) / dx` 的极限。

相对可能性的计算方法是用一个狭窄区间内某个样本出现的概率除以区间的宽度（因此使用“密度”一词）。由于可能性是相对于其他点的，它的值可以大于 *1.0*。

cdf(*x*)
使用 累积分布函数 (cdf)，计算一个随机变量 *X* 小于等于 *x* 的概率。在数学上，它表示为 `P(X <= x)`。

inv_cdf(*p*)
计算反向累积分布函数，也称为 分位数函数 或 百分点 函数。在数学上，它表示为 `x : P(X <= x) = p`。

找出随机变量 *X* 的值 *x* 使得该变量小于等于该值的概率等于给定的概率 *p*。

overlap(*other*)
测量两个正态概率分布之间的一致性。返回介于 0.0 和 1.0 之间的值，给出 两个概率密度函数的重叠区域。

quantiles(*n=4*)
将指定正态分布划分为 *n* 个相等概率的连续分隔区。返回这些分隔区对应的 (n - 1) 个分隔点的列表。

将 *n* 设为 4 以使用四分位（默认值）。将 *n* 设为 10 以使用十分位。将 *n* 设为 100 以使用百分位，即给出 99 个分隔点来将正态分布分隔为 100 个大小相等的组。

zscore(*x*)
计算 标准分 即以高于或低于正态分布的平均值的标准差数值的形式来描述 *x*: `(x - mean) / stdev`.

3.9 版新加入.

NormalDist 的实例支持加上、减去、乘以或除以一个常量。这些运算被用于转换和缩放。例如:

```
>>> temperature_february = NormalDist(5, 2.5)             # Celsius
>>> temperature_february * (9/5) + 32                     # Fahrenheit
NormalDist(mu=41.0, sigma=4.5)
```

不允许一个常量除以*NormalDist* 的实例，因为结果将不是正态分布。

由于正态分布是由独立变量的累加效应产生的，因此允许表示为*NormalDist* 实例的 两组独立正态分布的随机变量相加和相减。例如:

```
>>> birth_weights = NormalDist.from_samples([2.5, 3.1, 2.1, 2.4, 2.7, 3.5])
>>> drug_effects = NormalDist(0.4, 0.15)
>>> combined = birth_weights + drug_effects
>>> round(combined.mean, 1)
3.1
>>> round(combined.stdev, 1)
0.5
```

3.8 版新加入.

NormalDist 示例和用法

NormalDist 适合用来解决经典概率问题。

举例来说，如果 SAT 考试的历史数据 显示分数呈平均值为 1060 且标准差为 195 的正态分布，则可以确定考试分数处于 1100 和 1200 之间的学生的百分比舍入到最接近的整数应为:

```
>>> sat = NormalDist(1060, 195)
>>> fraction = sat.cdf(1200 + 0.5) - sat.cdf(1100 - 0.5)
>>> round(fraction * 100.0, 1)
18.4
```

求 SAT 分数的 四分位 和 十分位:

```
>>> list(map(round, sat.quantiles()))
[928, 1060, 1192]
>>> list(map(round, sat.quantiles(n=10)))
[810, 896, 958, 1011, 1060, 1109, 1162, 1224, 1310]
```

为了估算一个不易解析的模型分布，*NormalDist* 可以生成用于 蒙特卡洛模拟 的输入样本:

```
>>> def model(x, y, z):
...     return (3*x + 7*x*y - 5*y) / (11 * z)
...
>>> n = 100_000
>>> X = NormalDist(10, 2.5).samples(n, seed=3652260728)
>>> Y = NormalDist(15, 1.75).samples(n, seed=4582495471)
>>> Z = NormalDist(50, 1.25).samples(n, seed=6582483453)
>>> quantiles(map(model, X, Y, Z))
[1.4591308524824727, 1.8035946855390597, 2.175091447274739]
```

当样本量较大并且成功试验的可能性接近 50% 时，正态分布可以被用来模拟 二项分布。

例如，一次开源会议有 750 名与会者和两个可分别容纳 500 人的会议厅。会上有一场关于 Python 的演讲和一场关于 Ruby 的演讲。在往届会议中，65% 的与会者更愿意去听关于 Python 的演讲。假定人群的偏好没有发生改变，那么 Python 演讲的会议厅不超出其容量上限的可能性是多少?

```
>>> n = 750             # Sample size
>>> p = 0.65            # Preference for Python
>>> q = 1.0 - p         # Preference for Ruby
>>> k = 500             # Room capacity

>>> # Approximation using the cumulative normal distribution
>>> from math import sqrt
>>> round(NormalDist(mu=n*p, sigma=sqrt(n*p*q)).cdf(k + 0.5), 4)
0.8402

>>> # Solution using the cumulative binomial distribution
>>> from math import comb, fsum
>>> round(fsum(comb(n, r) * p**r * q**(n-r) for r in range(k+1)), 4)
0.8402

>>> # Approximation using a simulation
>>> from random import seed, choices
>>> seed(8675309)
>>> def trial():
...     return choices(('Python', 'Ruby'), (p, q), k=n).count('Python')
>>> mean(trial() <= k for i in range(10_000))
0.8398
```

在机器学习问题中也经常会出现正态分布。

Wikipedia 上有一个 朴素贝叶斯分类器的好例子。挑战的问题是根据对多个正态分布的特征测量值包括身高、体重和足部尺码来预测一个人的性别。

我们得到了由八个人的测量值组成的训练数据集。假定这些测量值是正态分布的，因此我们用*NormalDist*来总结数据：

```
>>> height_male = NormalDist.from_samples([6, 5.92, 5.58, 5.92])
>>> height_female = NormalDist.from_samples([5, 5.5, 5.42, 5.75])
>>> weight_male = NormalDist.from_samples([180, 190, 170, 165])
>>> weight_female = NormalDist.from_samples([100, 150, 130, 150])
>>> foot_size_male = NormalDist.from_samples([12, 11, 12, 10])
>>> foot_size_female = NormalDist.from_samples([6, 8, 7, 9])
```

接下来，我们遇到一个特征测量值已知但性别未知的新人：

```
>>> ht = 6.0        # height
>>> wt = 130        # weight
```

(下页继续)

(繼續上一頁)

```
>>> fs = 8          # foot size
```

从是男是女各 50% 的 先验概率 出发，我们通过将该先验概率乘以给定性别的特征度量值的可能性累积值来计算后验概率：

```
>>> prior_male = 0.5
>>> prior_female = 0.5
>>> posterior_male = (prior_male * height_male.pdf(ht) *
...                   weight_male.pdf(wt) * foot_size_male.pdf(fs))

>>> posterior_female = (prior_female * height_female.pdf(ht) *
...                     weight_female.pdf(wt) * foot_size_female.pdf(fs))
```

最终预测值应为最大后验概率值。这种算法被称为 maximum a posteriori 或 MAP：

```
>>> 'male' if posterior_male > posterior_female else 'female'
'female'
```

CHAPTER 10

函式编程模組

本章里描述的模块提供了函数和类，以支持函数式编程风格和在可调用对象上的通用操作。

本章对下列模块进行说明：

10.1 itertools --- 为高效循环而创建迭代器的函数

本模块实现一系列*iterator*，这些迭代器受到 APL，Haskell 和 SML 的启发。为了适用于 Python，它们都被重新写过。

本模块标准化了一个快速、高效利用内存的核心工具集，这些工具本身或组合都很有用。它们一起形成了“迭代器代数”，这使得在纯 Python 中有可能创建简洁又高效的专用工具。

例如，SML 有一个制表工具：`tabulate(f)`，它可产生一个序列 `f(0), f(1), ...`。在 Python 中可以组合*`map()`* 和*`count()`* 实现：`map(f, count())`。

这些内置工具同时也能很好地与*`operator`* 模块中的高效函数配合使用。例如，我们可以将两个向量的点积映射到乘法运算符：`sum(map(operator.mul, vector1, vector2))`。

无穷迭代器：

迭代器	实参	结果	示例
`count()`	start, [step]	start, start+step, start+2*step, ...	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, ... plast, p0, p1, ...	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, ... 重复无限次或 n 次	`repeat(10, 3) --> 10 10 10`

根据最短输入序列长度停止的迭代器：

迭代器	实参	结果	示例
accumulate()	p [,func]	p0, p0+p1, p0+p1+p2, ...	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
chain()	p, q, ...	p0, p1, ... plast, q0, q1, ...	`chain('ABC', 'DEF') --> A B C D E F`
chain.from_iterable()	iterable	p0, p1, ... plast, q0, q1, ...	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
compress()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
dropwhile()	pred, seq	seq[n], seq[n+1], ... 从 pred 首次真值测试失败开始	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
filterfalse()	pred, seq	seq 中 pred(x) 为假值的元素，x 是 seq 中的元素。	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
groupby()	iterable[, key]	根据 key(v) 值分组的迭代器	
islice()	seq, [start,] stop [, step]	seq[start:stop:step] 中的元素	`islice('ABCDEFG', 2, None) --> C D E F G`
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ...	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
takewhile()	pred, seq	seq[0], seq[1], ..., 直到 pred 真值测试失败	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
tee()	it, n	it1, it2, ... itn 将一个迭代器拆分为 n 个迭代器	
zip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	`zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-`

排列组合迭代器：

迭代器	实参	结果
product()	p, q, ... [repeat=1]	笛卡尔积，相当于嵌套的 for 循环
permutations()	p[, r]	长度 r 元组，所有可能的排列，无重复元素
combinations()	p, r	长度 r 元组，有序，无重复元素
combinations_with_replacement()	p, r	长度 r 元组，有序，元素可重复

例子	结果
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

10.1.1 Itertool 函数

下列模块函数均创建并返回迭代器。有些迭代器不限制输出流长度，所以它们只应在能截断输出流的函数或循环中使用。

itertools.**accumulate**(*iterable*[, *func*, *, *initial=None*])

创建一个迭代器，返回累积汇总值或其他双目运算函数的累积结果值（通过可选的 *func* 参数指定）。

如果提供了 *func*，它应当为带有两个参数的函数。输入 *iterable* 的元素可以是能被 *func* 接受为参数的任意类型。（例如，对于默认的加法运算，元素可以是任何可相加的类型包括*Decimal* 或*Fraction*。）

通常，输出的元素数量与输入的可迭代对象是一致的。但是，如果提供了关键字参数 *initial*，则累加会以 *initial* 值开始，这样输出就比输入的可迭代对象多一个元素。

大致相当于：

```
def accumulate(iterable, func=operator.add, *, initial=None):
    'Return running totals'
    # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
    # accumulate([1,2,3,4,5], initial=100) --> 100 101 103 106 110 115
    # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
    it = iter(iterable)
    total = initial
    if initial is None:
        try:
            total = next(it)
        except StopIteration:
            return
    yield total
    for element in it:
        total = func(total, element)
        yield total
```

func 参数有几种用法。它可以被设为*min()* 最终得到一个最小值，或者设为*max()* 最终得到一个最大值，或设为*operator.mul()* 最终得到一个乘积。摊销表可通过累加利息和支付款项得到。给 iterable 设置初始值并只将参数 *func* 设为累加总数可以对一阶 递归关系 建模。

```
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> list(accumulate(data, operator.mul))     # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
>>> list(accumulate(data, max))              # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]

# Amortize a 5% loan of 1000 with 4 annual payments of 90
>>> cashflows = [1000, -90, -90, -90, -90]
>>> list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))
[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]

# Chaotic recurrence relation https://en.wikipedia.org/wiki/Logistic_map
>>> logistic_map = lambda x, _:  r * x * (1 - x)
>>> r = 3.8
>>> x0 = 0.4
>>> inputs = repeat(x0, 36)     # only the initial value is used
>>> [format(x, '.2f') for x in accumulate(inputs, logistic_map)]
['0.40', '0.91', '0.30', '0.81', '0.60', '0.92', '0.29', '0.79', '0.63',
 '0.88', '0.39', '0.90', '0.33', '0.84', '0.52', '0.95', '0.18', '0.57',
 '0.93', '0.25', '0.71', '0.79', '0.63', '0.88', '0.39', '0.91', '0.32',
 '0.83', '0.54', '0.95', '0.20', '0.60', '0.91', '0.30', '0.80', '0.60']
```

参考一个类似函数*functools.reduce()*，它只返回一个最终累积值。

3.2 版新加入.

3.3 版更變: 增加可选参数 *func* 。

3.8 版更變: 添加了可选的 *initial* 形参。

itertools.**chain**(**iterables*)
创建一个迭代器，它首先返回第一个可迭代对象中所有元素，接着返回下一个可迭代对象中所有元素，直到耗尽所有可迭代对象中的元素。可将多个序列处理为单个序列。大致相当于:

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

classmethod chain.**from_iterable**(*iterable*)
构建类似*chain()*迭代器的另一个选择。从一个单独的可迭代参数中得到链式输入，该参数是延迟计算的。大致相当于:

```
def from_iterable(iterables):
    # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

itertools.**combinations**(*iterable*, *r*)
返回由输入 *iterable* 中元素组成长度为 *r* 的子序列。

组合元组会以字典顺序根据所输入 *iterable* 的顺序发出。因此，如果所输入 *iterable* 是已排序的，组合元组也将按已排序的顺序生成。

即使元素的值相同，不同位置的元素也被认为是不同的。如果元素各自不同，那么每个组合中没有重复元素。

大致相当于:

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = list(range(r))
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

*combinations()*的代码可被改写为*permutations()*过滤后的子序列，（相对于元素在输入中的位置）元素不是有序的。

```
def combinations(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in permutations(range(n), r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

当 `0 <= r <= n` 时，返回项的个数是 `n! / r! / (n-r)!`；当 `r > n` 时，返回项个数为 0。

itertools.**combinations_with_replacement**(*iterable*, *r*)

返回由输入 *iterable* 中元素组成的长度为 *r* 的子序列，允许每个元素可重复出现。

组合元组会以字典顺序根据所输入 *iterable* 的顺序发出。因此，如果所输入 *iterable* 是已排序的，组合元组也将按已排序的顺序生成。

不同位置的元素是不同的，即使它们的值相同。因此如果输入中的元素都是不同的话，返回的组合中元素也都会不同。

大致相当于：

```
def combinations_with_replacement(iterable, r):
    # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
    pool = tuple(iterable)
    n = len(pool)
    if not n and r:
        return
    indices = [0] * r
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != n - 1:
                break
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

combinations_with_replacement() 的代码可被改写为 `production()` 过滤后的子序列，（相对于元素在输入中的位置）元素不是有序的。

```
def combinations_with_replacement(iterable, r):
    pool = tuple(iterable)
    n = len(pool)
    for indices in product(range(n), repeat=r):
        if sorted(indices) == list(indices):
            yield tuple(pool[i] for i in indices)
```

当 `n > 0` 时，返回项个数为 `(n+r-1)! / r! / (n-1)!`.

3.1 版新加入.

itertools.**compress**(*data*, *selectors*)

创建一个迭代器，它返回 *data* 中经 *selectors* 真值测试为 `True` 的元素。迭代器在两者较短的长度处停止。大致相当于：

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
    return (d for d, s in zip(data, selectors) if s)
```

3.1 版新加入.

itertools.**count**(*start=0*, *step=1*)
创建一个迭代器，它从 *start* 值开始，返回均匀间隔的值。常用于*map()* 中的实参来生成连续的数据点。此外，还用于*zip()* 来添加序列号。大致相当于：

```
def count(start=0, step=1):
    # count(10) --> 10 11 12 13 14 ...
    # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
    n = start
    while True:
        yield n
        n += step
```

当对浮点数计数时，替换为乘法代码有时精度会更好，例如：`(start + step * i for i in count())`。

3.1 版更變: 增加参数 *step*，允许非整型。

itertools.**cycle**(*iterable*)
创建一个迭代器，返回 *iterable* 中所有元素并保存一个副本。当取完 *iterable* 中所有元素，返回副本中的所有元素。无限重复。大致相当于：

```
def cycle(iterable):
    # cycle('ABCD') --> A B C D A B C D A B C D ...
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
              yield element
```

注意，该函数可能需要相当大的辅助空间（取决于 *iterable* 的长度）。

itertools.**dropwhile**(*predicate*, *iterable*)
创建一个迭代器，如果 *predicate* 为 true，迭代器丢弃这些元素，然后返回其他元素。注意，迭代器在 *predicate* 首次为 false 之前不会产生任何输出，所以可能需要一定长度的启动时间。大致相当于：

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

itertools.**filterfalse**(*predicate*, *iterable*)
创建一个迭代器，只返回 *iterable* 中 *predicate* 为 `False` 的元素。如果 *predicate* 是 `None`，返回真值测试为 false 的元素。大致相当于：

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.**groupby**(*iterable*, *key=None*)

创建一个迭代器，返回 *iterable* 中连续的键和组。*key* 是一个计算元素键值函数。如果未指定或为 None，*key* 缺省为恒等函数（identity function），返回元素不变。一般来说，*iterable* 需用同一个键值函数预先排序。

groupby() 操作类似于 Unix 中的 uniq。当每次 *key* 函数产生的键值改变时，迭代器会分组或生成一个新组（这就是为什么通常需要使用同一个键值函数先对数据进行排序）。这种行为与 SQL 的 GROUP BY 操作不同，SQL 的操作会忽略输入的顺序将相同键值的元素分在同组中。

返回的组本身也是一个迭代器，它与*groupby()* 共享底层的可迭代对象。因为源是共享的，当*groupby()* 对象向后迭代时，前一个组将消失。因此如果稍后还需要返回结果，可保存为列表：

```
groups = []
uniquekeys = []
data = sorted(data, key=keyfunc)
for k, g in groupby(data, keyfunc):
    groups.append(list(g))      # Store group iterator as a list
    uniquekeys.append(k)
```

groupby() 大致相当于：

```
class groupby:
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def __next__(self):
        self.id = object()
        while self.currkey == self.tgtkey:
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
        self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey, self.id))
    def _grouper(self, tgtkey, id):
        while self.id is id and self.currkey == tgtkey:
            yield self.currvalue
            try:
                self.currvalue = next(self.it)
            except StopIteration:
                return
            self.currkey = self.keyfunc(self.currvalue)
```

itertools.**islice**(*iterable*, *stop*)
itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

创建一个迭代器，返回从 *iterable* 里选中的元素。如果 *start* 不是 0，跳过 *iterable* 中的元素，直到到达 *start* 这个位置。之后迭代器连续返回元素，除非 *step* 设置的值很高导致被跳过。如果 *stop* 为 None，迭代器耗光为止；否则，在指定的位置停止。与普通的切片不同，*islice()* 不支持将 *start*，*stop*，或 *step* 设为负值。可用来从内部数据结构被压平的数据中提取相关字段（例如一个多行报告，它的名称字段出现在每三行上）。大致相当于：

```
def islice(iterable, *args):
    # islice('ABCDEFG', 2) --> A B
    # islice('ABCDEFG', 2, 4) --> C D
    # islice('ABCDEFG', 2, None) --> C D E F G
    # islice('ABCDEFG', 0, None, 2) --> A C E G
    s = slice(*args)
    start, stop, step = s.start or 0, s.stop or sys.maxsize, s.step or 1
    it = iter(range(start, stop, step))
    try:
        nexti = next(it)
    except StopIteration:
        # Consume *iterable* up to the *start* position.
        for i, element in zip(range(start), iterable):
            pass
        return
    try:
        for i, element in enumerate(iterable):
            if i == nexti:
                yield element
                nexti = next(it)
    except StopIteration:
        # Consume to *stop*.
        for i, element in zip(range(i + 1, stop), iterable):
            pass
```

如果 *start* 为 None，迭代从 0 开始。如果 *step* 为 None ，步长缺省为 1。

itertools.**permutations**(*iterable*, *r=None*)

连续返回由 *iterable* 元素生成长度为 *r* 的排列。

如果 *r* 未指定或为 None ， *r* 默认设置为 *iterable* 的长度，这种情况下，生成所有全长排列。

排列元组会以字典顺序根据所输入 *iterable* 的顺序发出。因此，如果所输入 *iterable* 是已排序的，组合元组也将按已排序的顺序生成。

即使元素的值相同，不同位置的元素也被认为是不同的。如果元素值都不同，每个排列中的元素值不会重复。

大致相当于：

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = list(range(n))
    cycles = list(range(n, n-r, -1))
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
```

(下页继续)

(繼續上一頁)

```
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

permutations() 的代码也可被改写为*product()* 的子序列，只要将含有重复元素（来自输入中同一位置的）的项排除。

```
def permutations(iterable, r=None):
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    for indices in product(range(n), repeat=r):
        if len(set(indices)) == r:
            yield tuple(pool[i] for i in indices)
```

当 `0 <= r <= n`，返回项个数为 `n! / (n-r)!`；当 `r > n`，返回项个数为 0。

itertools.**product**(**iterables*, *repeat=1*)

可迭代对象输入的笛卡儿积。

大致相当于生成器表达式中的嵌套循环。例如，`product(A, B)` 和 `((x,y) for x in A for y in B)` 返回结果一样。

嵌套循环像里程表那样循环变动，每次迭代时将最右侧的元素向后迭代。这种模式形成了一种字典序，因此如果输入的可迭代对象是已排序的，笛卡尔积元组依次序发出。

要计算可迭代对象自身的笛卡尔积，将可选参数 *repeat* 设定为要重复的次数。例如，`product(A, repeat=4)` 和 `product(A, A, A, A)` 是一样的。

该函数大致相当于下面的代码，只不过实际实现方案不会在内存中创建中间结果。

```
def product(*args, repeat=1):
    # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
    # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
    pools = [tuple(pool) for pool in args] * repeat
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

在*product()* 运行之前，它会完全耗尽输入的可迭代对象，在内存中保留值的临时池以生成结果积。相应地，它只适用于有限的输入。

itertools.**repeat**(*object*[, *times*])

创建一个迭代器，不断重复 *object* 。除非设定参数 *times* ，否则将无限重复。可用于*map()* 函数中的参数，被调用函数可得到一个不变参数。也可用于*zip()* 的参数以在元组记录中创建一个不变的部分。

大致相当于：

```
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
```

(下页继续)

(繼續上一頁)

```
        for i in range(times):
            yield object
```

repeat 最常见的用途就是在 *map* 或 *zip* 提供一个常量流：

```
>>> list(map(pow, range(10), repeat(2)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

itertools.**starmap**(*function*, *iterable*)

创建一个迭代器，使用从可迭代对象中获取的参数来计算该函数。当参数对应的形参已从一个单独可迭代对象组合为元组时（数据已被“预组对”）可用此函数代替*map()*。*map()* 与*starmap()* 之间的区别可以类比 function(a,b) 与 function(*c) 的区别。大致相当于:

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

itertools.**takewhile**(*predicate*, *iterable*)

创建一个迭代器，只要 predicate 为真就从可迭代对象中返回元素。大致相当于:

```
def takewhile(predicate, iterable):
    # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

itertools.**tee**(*iterable*, *n=2*)

从一个可迭代对象中返回 *n* 个独立的迭代器。

下面的 Python 代码能帮助解释 *tee* 做了什么（尽管实际的实现更复杂，而且仅使用了一个底层的 FIFO 队列）。

大致相当于:

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:             # when the local deque is empty
                try:
                    newval = next(it)   # fetch a new value and
                except StopIteration:
                    return
                for d in deques:        # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

一旦*tee()* 实施了一次分裂，原有的 *iterable* 不应再被使用；否则 tee 对象无法得知 *iterable* 可能已向后迭代。

tee 迭代器不是线程安全的。当同时使用由同一个*tee()* 调用所返回的迭代器时可能引发*RuntimeError*，即使原本的 *iterable* 是线程安全的。

该迭代工具可能需要相当大的辅助存储空间（这取决于要保存多少临时数据）。通常，如果一个迭代器在另一个迭代器开始之前就要使用大部份或全部数据，使用 *list()* 会比 *tee()* 更快。

itertools.**zip_longest**(**iterables*, *fillvalue=None*)

创建一个迭代器，从每个可迭代对象中收集元素。如果可迭代对象的长度未对齐，将根据 *fillvalue* 填充缺失值。迭代持续到耗光最长的可迭代对象。大致相当于：

```
def zip_longest(*args, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    iterators = [iter(it) for it in args]
    num_active = len(iterators)
    if not num_active:
        return
    while True:
        values = []
        for i, it in enumerate(iterators):
            try:
                value = next(it)
            except StopIteration:
                num_active -= 1
                if not num_active:
                    return
                iterators[i] = repeat(fillvalue)
                value = fillvalue
            values.append(value)
        yield tuple(values)
```

如果其中一个可迭代对象有无限长度，*zip_longest()* 函数应封装在限制调用次数的场景中（例如 *islice()* 或 *takewhile()*）。除非指定，*fillvalue* 默认为 None 。

10.1.2 itertools 配方

本节将展示如何使用现有的 itertools 作为基础构件来创建扩展的工具集。

基本上所有这些西方和许许多多其他的配方都可以通过 Python Package Index 上的 more-itertools 项目 来安装:

```
pip install more-itertools
```

扩展的工具提供了与底层工具集相同的高性能。保持了超棒的内存利用率，因为一次只处理一个元素，而不是将整个可迭代对象加载到内存。代码量保持得很小，以函数式风格将这些工具连接在一起，有助于消除临时变量。速度依然很快，因为倾向于使用“矢量化”构件来取代解释器开销大的 for 循环和*generator* 。

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def prepend(value, iterator):
    "Prepend a single value in front of an iterator"
    # prepend(1, [2, 3, 4]) -> 1 2 3 4
    return chain([value], iterator)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return map(function, count(start))

def tail(n, iterable):
    "Return an iterator over the last n items"
```

(下页继续)

(繼續上一頁)

```
    # tail(3, 'ABCDEFG') --> E F G
    return iter(collections.deque(iterable, maxlen=n))

def consume(iterator, n=None):
    "Advance the iterator n-steps ahead. If n is None, consume entirely."
    # Use functions that consume iterators at C speed.
    if n is None:
        # feed the entire iterator into a zero-length deque
        collections.deque(iterator, maxlen=0)
    else:
        # advance to the empty slice starting at position n
        next(islice(iterator, n, n), None)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)

def all_equal(iterable):
    "Returns True if all the elements are equal to each other"
    g = groupby(iterable)
    return next(g, True) and not next(g, False)

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(map(pred, iterable))

def pad_none(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(tuple(iterable), n))

def dotproduct(vec1, vec2):
    return sum(map(operator.mul, vec1, vec2))

def convolve(signal, kernel):
    # See:  https://betterexplained.com/articles/intuitive-convolution/
    # convolve(data, [0.25, 0.25, 0.25, 0.25]) --> Moving average (blur)
    # convolve(data, [1, -1]) --> 1st finite difference (1st derivative)
    # convolve(data, [1, -2, 1]) --> 2nd finite difference (2nd derivative)
    kernel = tuple(kernel)[::-1]
    n = len(kernel)
    window = collections.deque([0], maxlen=n) * n
    for x in chain(signal, repeat(0, n-1)):
        window.append(x)
        yield sum(map(operator.mul, kernel, window))

def flatten(list_of_lists):
    "Flatten one level of nesting"
    return chain.from_iterable(list_of_lists)
```

(下页继续)

(繼續上一頁)

```
def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.

    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return zip(a, b)

def grouper(iterable, n, fillvalue=None):
    "Collect data into fixed-length chunks or blocks"
    # grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return zip_longest(*args, fillvalue=fillvalue)

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    num_active = len(iterables)
    nexts = cycle(iter(it).__next__ for it in iterables)
    while num_active:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            # Remove the iterator we just exhausted from the cycle.
            num_active -= 1
            nexts = cycle(islice(nexts, num_active))

def partition(pred, iterable):
    "Use a predicate to partition entries into false entries and true entries"
    # partition(is_odd, range(10)) --> 0 2 4 6 8   and  1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(pred, t1), filter(pred, t2)

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def unique_everseen(iterable, key=None):
    "List unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') --> A B C D
    # unique_everseen('ABBCcAD', str.lower) --> A B C D
    seen = set()
    seen_add = seen.add
    if key is None:
        for element in filterfalse(seen.__contains__, iterable):
            seen_add(element)
            yield element
```

(下页继续)

(繼續上一頁)

```
    else:
        for element in iterable:
            k = key(element)
            if k not in seen:
                seen_add(k)
                yield element

def unique_justseen(iterable, key=None):
    "List unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
    # unique_justseen('ABBCcAD', str.lower) --> A B C A D
    return map(next, map(operator.itemgetter(1), groupby(iterable, key)))

def iter_except(func, exception, first=None):
    """ Call a function repeatedly until an exception is raised.

    Converts a call-until-exception interface to an iterator interface.
    Like builtins.iter(func, sentinel) but uses an exception instead
    of a sentinel to end the loop.

    Examples:
        iter_except(functools.partial(heappop, h), IndexError)    # priority queue␣
↪iterator
        iter_except(d.popitem, KeyError)                          # non-blocking dict␣
↪iterator
        iter_except(d.popleft, IndexError)                        # non-blocking deque␣
↪iterator
        iter_except(q.get_nowait, Queue.Empty)                    # loop over a␣
↪producer Queue
        iter_except(s.pop, KeyError)                              # non-blocking set␣
↪iterator

    """
    try:
        if first is not None:
            yield first()            # For database APIs needing an initial cast to␣
↪db.first()
        while True:
            yield func()
    except exception:
        pass

def first_true(iterable, default=False, pred=None):
    """Returns the first true value in the iterable.

    If no true value is found, returns *default*

    If *pred* is not None, returns the first item
    for which pred(item) is true.

    """
    # first_true([a,b,c], x) --> a or b or c or x
    # first_true([a,b], x, f) --> a if f(a) else b if f(b) else x
    return next(filter(pred, iterable), default)

def random_product(*args, repeat=1):
```

(下页继续)

(繼續上一頁)

```
    "Random selection from itertools.product(*args, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(map(random.choice, pools))

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Random selection from itertools.combinations_with_replacement(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.choices(range(n), k=r))
    return tuple(pool[i] for i in indices)

def nth_combination(iterable, r, index):
    "Equivalent to list(combinations(iterable, r))[index]"
    pool = tuple(iterable)
    n = len(pool)
    if r < 0 or r > n:
        raise ValueError
    c = 1
    k = min(r, n-r)
    for i in range(1, k+1):
        c = c * (n - k + i) // i
    if index < 0:
        index += c
    if index < 0 or index >= c:
        raise IndexError
    result = []
    while r:
        c, n, r = c*r//n, n-1, r-1
        while index >= c:
            index -= c
            c, n = c*(n-r)//n, n-1
        result.append(pool[-1-n])
    return tuple(result)
```

10.2 functools --- 高阶函数和可调用对象上的操作

源代码: Lib/functools.py

---

functools 模块应用于高阶函数，即参数或（和）返回值为其他函数的函数。通常来说，此模块的功能适用于所有可调用对象。

functools 模块定义了以下函数:

@functools.**cache**(*user_function*)
简单轻量级未绑定函数缓存。有时称为 "memoize"。

返回值与 lru_cache(maxsize=None) 相同，创建一个查找函数参数的字典的简单包装器。因为它不需要移出旧值，所以比带有大小限制的*lru_cache()* 更小更快。

例如:

```
@cache
def factorial(n):
    return n * factorial(n-1) if n else 1

>>> factorial(10)      # no previously cached result, makes 11 recursive calls
3628800
>>> factorial(5)       # just looks up cached value result
120
>>> factorial(12)      # makes two new recursive calls, the other 10 are cached
479001600
```

3.9 版新加入.

@functools.**cached_property**(*func*)
将一个类方法转换为特征属性，一次性计算该特征属性的值，然后将其缓存为实例生命周期内的普通属性。类似于*property()* 但增加了缓存功能。对于在其他情况下实际不可变的高计算资源消耗的实例特征属性来说该函数非常有用。

示例:

```
class DataSet:

    def __init__(self, sequence_of_numbers):
        self._data = tuple(sequence_of_numbers)

    @cached_property
    def stdev(self):
        return statistics.stdev(self._data)
```

cached_property() 的设定与*property()* 有所不同。常规的 property 会阻止属性写入，除非定义了 setter。与之相反，*cached_property* 则允许写入。

cached_property 装饰器仅在执行查找且不存在同名属性时才会运行。当运行时，*cached_property* 会写入同名的属性。后续的属性读取和写入操作会优先于 *cached_property* 方法，其行为就像普通的属性一样。

缓存的值可通过删除该属性来清空。这允许 *cached_property* 方法再次运行。

注意，这个装饰器会影响 **PEP 412** 键共享字典的操作。这意味着相应的字典实例可能占用比通常时更多的空间。

而且，这个装饰器要求每个实例上的 __dict__ 是可变的映射。这意味着它将不适用于某些类型，例如元类（因为类型实例上的 __dict__ 属性是类命名空间的只读代理），以及那些指定了 __slots__ 但未包括 __dict__ 作为所定义的空位之一的类（因为这样的类根本没有提供 __dict__ 属性）。

如果可变的映射不可用或者如果想要节省空间的键共享，可以通过在*cache()* 之上堆叠一个*property()* 来实现类似*cached_property()* 的效果:

```
class DataSet:
    def __init__(self, sequence_of_numbers):
        self._data = sequence_of_numbers

    @property
    @cache
    def stdev(self):
        return statistics.stdev(self._data)
```

3.8 版新加入.

functools.**cmp_to_key**(*func*)

将 (旧式的) 比较函数转换为新式的*key function* . 在类似于*sorted()* , *min()* , *max()* , *heapq.nlargest()* , *heapq.nsmallest()* , *itertools.groupby()* 等函数的 *key* 参数中使用。此函数主要用作将 Python 2 程序转换至新版的转换工具，以保持对比较函数的兼容。

比较函数意为一个可调用对象，该对象接受两个参数并比较它们，结果为小于则返回一个负数，相等则返回零，大于则返回一个正数。key function 则是一个接受一个参数，并返回另一个用以排序的值的可调用对象。

示例:

```
sorted(iterable, key=cmp_to_key(locale.strcoll))  # locale-aware sort order
```

有关排序示例和简要排序教程，请参阅 sortinghowto 。

3.2 版新加入.

@functools.**lru_cache**(*user_function*)

@functools.**lru_cache**(*maxsize=128*, *typed=False*)

一个为函数提供缓存功能的装饰器，缓存 *maxsize* 组传入参数，在下次以相同参数调用时直接返回上一次的结果。用以节约高开销或 I/O 函数的调用时间。

由于使用了字典存储缓存，所以该函数的固定参数和关键字参数必须是可哈希的。

不同模式的参数可能被视为不同从而产生多个缓存项，例如, *f(a=1, b=2)* 和 *f(b=2, a=1)* 因其参数顺序不同，可能会被缓存两次。

如果指定了 *user_function*，它必须是一个可调用对象。这允许 *lru_cache* 装饰器被直接应用于一个用户自定义函数，让 *maxsize* 保持其默认值 128:

```
@lru_cache
def count_vowels(sentence):
    sentence = sentence.casefold()
    return sum(sentence.count(vowel) for vowel in 'aeiou')
```

如果 *maxsize* 设为 None，LRU 特性将被禁用且缓存可无限增长。

如果 *typed* 设置为 true，不同类型的函数参数将被分别缓存。例如，f(3) 和 f(3.0) 将被视为不同而分别缓存。

被包装的函数配有一个 cache_parameters() 函数，该函数返回一个新的*dict* 用来显示 *maxsize* 和 *typed* 的值。这只是出于显示信息的目的。改变值没有任何效果。

为了衡量缓存的有效性以便调整 *maxsize* 形参，被装饰的函数带有一个 `cache_info()` 函数。当调用 `cache_info()` 函数时，返回一个具名元组，包含命中次数 *hits*，未命中次数 *misses* ，最大缓存数量 *maxsize* 和当前缓存大小 *currsize*。在多线程环境中，命中数与未命中数是不完全准确的。

该装饰器也提供了一个用于清理/使缓存失效的函数 `cache_clear()` 。

原始的未经装饰的函数可以通过 `__wrapped__` 属性访问。它可以用于检查、绕过缓存，或使用不同的缓存再次装饰原始函数。

LRU（最久未使用算法）缓存 在最近的调用是即将到来的调用的最佳预测值时性能最好（例如，新闻服务器上最热门文章倾向于每天更改）。缓存的大小限制可确保缓存不会在长期运行进程如网站服务器上无限制地增长。

一般来说，LRU 缓存只在当你想要重用之前计算的结果时使用。因此，用它缓存具有副作用的函数、需要在每次调用时创建不同、易变的对象的函数或者诸如 time () 或 random () 之类的不纯函数是没有意义的。

静态 Web 内容的 LRU 缓存示例:

```
@lru_cache(maxsize=32)
def get_pep(num):
    'Retrieve text of a Python Enhancement Proposal'
    resource = 'https://www.python.org/dev/peps/pep-%04d/' % num
    try:
        with urllib.request.urlopen(resource) as s:
            return s.read()
    except urllib.error.HTTPError:
        return 'Not Found'

>>> for n in 8, 290, 308, 320, 8, 218, 320, 279, 289, 320, 9991:
...     pep = get_pep(n)
...     print(n, len(pep))

>>> get_pep.cache_info()
CacheInfo(hits=3, misses=8, maxsize=32, currsize=8)
```

以下是使用缓存通过 动态规划 计算 斐波那契数列 的例子。

```
@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```

3.2 版新加入.

3.3 版更變: 添加 *typed* 选项。

3.8 版更變: 添加了 *user_function* 选项。

3.9 版新加入: 新增函数 `cache_parameters()`

`@functools.total_ordering`

给定一个声明一个或多个全比较排序方法的类，这个类装饰器实现剩余的方法。这减轻了指定所有可能的全比较操作的工作。

此类必须包含以下方法之一：__lt__() 、__le__()、__gt__() 或 __ge__()。另外，此类必须支持 __eq__() 方法。

例如:

```
@total_ordering
class Student:
    def _is_valid_operand(self, other):
        return (hasattr(other, "lastname") and
                hasattr(other, "firstname"))
    def __eq__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) ==
                (other.lastname.lower(), other.firstname.lower()))
    def __lt__(self, other):
        if not self._is_valid_operand(other):
            return NotImplemented
        return ((self.lastname.lower(), self.firstname.lower()) <
                (other.lastname.lower(), other.firstname.lower()))
```

備F: 虽然此装饰器使得创建具有良好行为的完全有序类型变得非常容易，但它 确实是以执行速度更缓慢和派生比较方法的堆栈回溯更复杂为代价的。如果性能基准测试表明这是特定应用的瓶颈所在，则改为实现全部六个富比较方法应该会轻松提升速度。

3.2 版新加入.

3.4 版更變: 现在已支持从未识别类型的下层比较函数返回 NotImplemented 异常。

functools.**partial**(*func*, */*, **args*, ***keywords*)

返回一个新的部分对象，当被调用时其行为类似于 *func* 附带位置参数 *args* 和关键字参数 *keywords* 被调用。如果为调用提供了更多的参数，它们会被附加到 *args*。如果提供了额外的关键字参数，它们会扩展并重载 *keywords*。大致等价于:

```
def partial(func, /, *args, **keywords):
    def newfunc(*fargs, **fkeywords):
        newkeywords = {**keywords, **fkeywords}
        return func(*args, *fargs, **newkeywords)
    newfunc.func = func
    newfunc.args = args
    newfunc.keywords = keywords
    return newfunc
```

partial() 会被“冻结了”一部分函数参数和/或关键字的部分函数应用所使用，从而得到一个具有简化签名的新对象。例如，*partial()* 可用来创建一个行为类似于 *int()* 函数的可调用对象，其中 *base* 参数默认为二:

```
>>> from functools import partial
>>> basetwo = partial(int, base=2)
>>> basetwo.__doc__ = 'Convert base 2 string to an int.'
>>> basetwo('10010')
18
```

class functools.**partialmethod**(*func*, */*, **args*, ***keywords*)

返回一个新的*partialmethod* 描述器，其行为类似*partial* 但它被设计用作方法定义而非直接用作可调用对象。

func 必须是一个*descriptor* 或可调用对象（同属两者的对象例如普通函数会被当作描述器来处理）。

当 *func* 是一个描述器（例如普通 Python 函数，`classmethod()`，`staticmethod()`，`abstractmethod()` 或其他`partialmethod` 的实例）时，对 `__get__` 的调用会被委托给底层的描述器，并会返回一个适当的部分对象 作为结果。

当 *func* 是一个非描述器类可调用对象时，则会动态创建一个适当的绑定方法。当用作方法时其行为类似普通 Python 函数：将会插入 *self* 参数作为第一个位置参数，其位置甚至会处于提供给`partialmethod` 构造器的 *args* 和 *keywords* 之前。

示例:

```
>>> class Cell:
...     def __init__(self):
...         self._alive = False
...     @property
...     def alive(self):
...         return self._alive
...     def set_state(self, state):
...         self._alive = bool(state)
...     set_alive = partialmethod(set_state, True)
...     set_dead = partialmethod(set_state, False)
...
>>> c = Cell()
>>> c.alive
False
>>> c.set_alive()
>>> c.alive
True
```

3.4 版新加入.

`functools.`**`reduce`**(*function*, *iterable*[, *initializer*])

将两个参数的 *function* 从左至右积累地应用到 *iterable* 的条目，以便将该可迭代对象缩减为单一的值。例如，`reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])` 是计算 `((((1+2)+3)+4)+5)` 的值。左边的参数 *x* 是积累值而右边的参数 *y* 则是来自 *iterable* 的更新值。如果存在可选项 *initializer*，它会被放在参与计算的可迭代对象的条目之前，并在可迭代对象为空时作为默认值。如果没有给出 *initializer* 并且 *iterable* 仅包含一个条目，则将返回第一项。

大致相当于:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

请参阅`itertools.accumulate()` 了解有关可产生所有中间值的迭代器。

`@functools.`**`singledispatch`**

将一个函数转换为单分派 *generic function*。

To define a generic function, decorate it with the `@singledispatch` decorator. When defining a function using `@singledispatch`, note that the dispatch happens on the type of the first argument:

```
>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)
```

To add overloaded implementations to the function, use the `register()` attribute of the generic function, which can be used as a decorator. For functions annotated with types, the decorator will infer the type of the first argument automatically:

```
>>> @fun.register
... def _(arg: int, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)
...
>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

对于不使用类型标注的代码，可以将适当的类型参数显式地传给装饰器本身:

```
>>> @fun.register(complex)
... def _(arg, verbose=False):
...     if verbose:
...         print("Better than complicated.", end=" ")
...     print(arg.real, arg.imag)
...
```

To enable registering *lambdas* and pre-existing functions, the `register()` attribute can also be used in a functional form:

```
>>> def nothing(arg, verbose=False):
...     print("Nothing.")
...
>>> fun.register(type(None), nothing)
```

The `register()` attribute returns the undecorated function. This enables decorator stacking, *pickling*, and the creation of unit tests for each variant independently:

```
>>> @fun.register(float)
... @fun.register(Decimal)
... def fun_num(arg, verbose=False):
...     if verbose:
...         print("Half of your number:", end=" ")
...     print(arg / 2)
...
>>> fun_num is fun
False
```

在调用时，泛型函数会根据第一个参数的类型进行分派:

```
>>> fun("Hello, world.")
Hello, world.
>>> fun("test.", verbose=True)
Let me just say, test.
>>> fun(42, verbose=True)
Strength in numbers, eh? 42
>>> fun(['spam', 'spam', 'eggs', 'spam'], verbose=True)
Enumerate this:
0 spam
1 spam
2 eggs
3 spam
>>> fun(None)
Nothing.
>>> fun(1.23)
0.615
```

Where there is no registered implementation for a specific type, its method resolution order is used to find a more generic implementation. The original function decorated with `@singledispatch` is registered for the base *`object`* type, which means it is used if no better implementation is found.

If an implementation is registered to an *abstract base class*, virtual subclasses of the base class will be dispatched to that implementation:

```
>>> from collections.abc import Mapping
>>> @fun.register
... def _(arg: Mapping, verbose=False):
...     if verbose:
...         print("Keys & Values")
...     for key, value in arg.items():
...         print(key, "=>", value)
...
>>> fun({"a": "b"})
a => b
```

To check which implementation the generic function will choose for a given type, use the `dispatch()` attribute:

```
>>> fun.dispatch(float)
<function fun_num at 0x1035a2840>
>>> fun.dispatch(dict)    # note: default implementation
<function fun at 0x103fe0000>
```

要访问所有忆注册实现，请使用只读的 `registry` 属性:

```
>>> fun.registry.keys()
dict_keys([<class 'NoneType'>, <class 'int'>, <class 'object'>,
           <class 'decimal.Decimal'>, <class 'list'>,
           <class 'float'>])
>>> fun.registry[float]
<function fun_num at 0x1035a2840>
>>> fun.registry[object]
<function fun at 0x103fe0000>
```

3.4 版新加入.

3.7 版更變: The `register()` attribute now supports using type annotations.

class functools.**singledispatchmethod**(*func*)

将一个方法转换为单分派 *generic function*。

To define a generic method, decorate it with the `@singledispatchmethod` decorator. When defining a function using `@singledispatchmethod`, note that the dispatch happens on the type of the first non-*self* or non-*cls* argument:

```
class Negator:
    @singledispatchmethod
    def neg(self, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    def _(self, arg: int):
        return -arg

    @neg.register
    def _(self, arg: bool):
        return not arg
```

`@singledispatchmethod` supports nesting with other decorators such as *@classmethod*. Note that to allow for `dispatcher.register`, `singledispatchmethod` must be the *outer most* decorator. Here is the `Negator` class with the `neg` methods bound to the class, rather than an instance of the class:

```
class Negator:
    @singledispatchmethod
    @classmethod
    def neg(cls, arg):
        raise NotImplementedError("Cannot negate a")

    @neg.register
    @classmethod
    def _(cls, arg: int):
        return -arg

    @neg.register
    @classmethod
    def _(cls, arg: bool):
        return not arg
```

The same pattern can be used for other similar decorators: *@staticmethod*, *@abstractmethod*, and others.

3.8 版新加入.

`functools.`**`update_wrapper`**(*wrapper*, *wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)

更新一个 *wrapper* 函数以使其类似于 *wrapped* 函数。可选参数为指明原函数的哪些属性要直接被赋值给 wrapper 函数的匹配属性的元组，并且这些 wrapper 函数的属性将使用原函数的对应属性来更新。这些参数的默认值是模块级常量 `WRAPPER_ASSIGNMENTS` (它将被赋值给 wrapper 函数的 `__module__`, `__name__`, `__qualname__`, `__annotations__` 和 `__doc__` 即文档字符串) 以及 `WRAPPER_UPDATES` (它将更新 wrapper 函数的 `__dict__` 即实例字典)。

为了允许出于内省和其他目的访问原始函数（例如绕过*lru_cache()* 之类的缓存装饰器），此函数会自动为 wrapper 添加一个指向被包装函数的 `__wrapped__` 属性。

此函数的主要目的是在*decorator* 函数中用来包装被装饰的函数并返回包装器。如果包装器函数未被更新，则被返回函数的元数据将反映包装器定义而不是原始函数定义，这通常没有什么用处。

update_wrapper() 可以与函数之外的可调用对象一同使用。在 *assigned* 或 *updated* 中命名的任何属性如果不存在于被包装对象则会被忽略（即该函数将不会尝试在包装器函数上设置它们）。如果包装器函数自身缺少在 *updated* 中命名的任何属性则仍将引发*AttributeError*。

3.2 版新加入: 自动添加 `__wrapped__` 属性。

3.2 版新加入: 默认拷贝 __annotations__ 属性。

3.2 版更變: 不存在的属性将不再触发*AttributeError*。

3.4 版更變: __wrapped__ 属性现在总是指向被包装的函数，即使该函数定义了 __wrapped__ 属性。(参见 bpo-17482)

@functools.**wraps**(*wrapped*, *assigned=WRAPPER_ASSIGNMENTS*, *updated=WRAPPER_UPDATES*)
这是一个便捷函数，用于在定义包装器函数时发起调用*update_wrapper()* 作为函数装饰器。它等价于 partial(update_wrapper, wrapped=wrapped, assigned=assigned, updated=updated)。例如:

```
>>> from functools import wraps
>>> def my_decorator(f):
...     @wraps(f)
...     def wrapper(*args, **kwds):
...         print('Calling decorated function')
...         return f(*args, **kwds)
...     return wrapper
...
>>> @my_decorator
... def example():
...     """Docstring"""
...     print('Called example function')
...
>>> example()
Calling decorated function
Called example function
>>> example.__name__
'example'
>>> example.__doc__
'Docstring'
```

如果不使用这个装饰器工厂函数，则 example 函数的名称将变为 'wrapper'，并且 example() 原本的文档字符串将会丢失。

10.2.1 partial 对象

partial 对象是由*partial()* 创建的可调用对象。它们具有三个只读属性:

partial.**func**
一个可调用对象或函数。对*partial* 对象的调用将被转发给*func* 并附带新的参数和关键字。

partial.**args**
最左边的位置参数将放置在提供给*partial* 对象调用的位置参数之前。

partial.**keywords**
当调用*partial* 对象时将要提供的关键字参数。

partial 对象与 function 对象的类似之处在于它们都是可调用、可弱引用的对象并可拥有属性。但两者也存在一些重要的区别。例如前者不会自动创建*__name__* 和 __doc__ 属性。而且，在类中定义的*partial* 对象的行为类似于静态方法，并且不会在实例属性查找期间转换为绑定方法。

10.3 operator --- 标准运算符替代函数

源代码: Lib/operator.py

---

operator 模块提供了一套与 Python 的内置运算符对应的高效率函数。例如，operator.add(x, y) 与表达式 x+y 相同。许多函数名与特殊方法名相同，只是没有双下划线。为了向后兼容性，也保留了许多包含双下划线的函数。为了表述清楚，建议使用没有双下划线的函数。

函数包含的种类有：对象的比较运算、逻辑运算、数学运算以及序列运算。

对象比较函数适用于所有的对象，函数名根据它们对应的比较运算符命名。

operator.**lt**(*a*, *b*)
operator.**le**(*a*, *b*)
operator.**eq**(*a*, *b*)
operator.**ne**(*a*, *b*)
operator.**ge**(*a*, *b*)
operator.**gt**(*a*, *b*)
operator.**__lt__**(*a*, *b*)
operator.**__le__**(*a*, *b*)
operator.**__eq__**(*a*, *b*)
operator.**__ne__**(*a*, *b*)
operator.**__ge__**(*a*, *b*)
operator.**__gt__**(*a*, *b*)
在 *a* 和 *b* 之间进行全比较。具体的，lt(a, b) 与 a < b 相同，le(a, b) 与 a <= b 相同，eq(a, b) 与 a == b 相同，ne(a, b) 与 a != b 相同，gt(a, b) 与 a > b 相同，ge(a, b)``与 ``a >= b 相同。注意这些函数可以返回任何值，无论它是否可当作布尔值。关于全比较的更多信息请参考 comparisons 。

逻辑运算通常也适用于所有对象，并且支持真值检测、标识检测和布尔运算：

operator.**not_**(*obj*)
operator.**__not__**(*obj*)
返回 not *obj* 的结果。(请注意对象实例并没有*__not__()* 方法；只有解释器核心可定义此操作。结果会受 __bool__() 和 __len__() 方法影响。)

operator.**truth**(*obj*)
如果 *obj* 为真值则返回*True*，否则返回*False*。这等价于使用*bool* 构造器。

operator.**is_**(*a*, *b*)
返回 a is b。检测对象标识。

operator.**is_not**(*a*, *b*)
返回 a is not b。检测对象标识。

数学和按位运算的种类是最多的：

operator.**abs**(*obj*)
operator.**__abs__**(*obj*)
返回 *obj* 的绝对值。

operator.**add**(*a*, *b*)
operator.**__add__**(*a*, *b*)
对于数字 *a* 和 *b*，返回 a + b。

operator.**and_**(*a*, *b*)
operator.**__and__**(*a*, *b*)
返回 *x* 和 *y* 按位与的结果。

operator.**floordiv**(*a*, *b*)
operator.**__floordiv__**(*a*, *b*)
返回 `a // b`。

operator.**index**(*a*)
operator.**__index__**(*a*)
返回 *a* 转换为整数的结果。等价于 `a.__index__()`。

operator.**inv**(*obj*)
operator.**invert**(*obj*)
operator.**__inv__**(*obj*)
operator.**__invert__**(*obj*)
返回数字 *obj* 按位取反的结果。这等价于 `~obj`。

operator.**lshift**(*a*, *b*)
operator.**__lshift__**(*a*, *b*)
返回 *a* 左移 *b* 位的结果。

operator.**mod**(*a*, *b*)
operator.**__mod__**(*a*, *b*)
返回 `a % b`。

operator.**mul**(*a*, *b*)
operator.**__mul__**(*a*, *b*)
对于数字 *a* 和 *b*，返回 `a * b`。

operator.**matmul**(*a*, *b*)
operator.**__matmul__**(*a*, *b*)
返回 `a @ b`。

3.5 版新加入.

operator.**neg**(*obj*)
operator.**__neg__**(*obj*)
返回 *obj* 取负的结果 (`-obj`)。

operator.**or_**(*a*, *b*)
operator.**__or__**(*a*, *b*)
返回 *a* 和 *b* 按位或的结果。

operator.**pos**(*obj*)
operator.**__pos__**(*obj*)
返回 *obj* 取正的结果 (`+obj`)。

operator.**pow**(*a*, *b*)
operator.**__pow__**(*a*, *b*)
对于数字 *a* 和 *b*，返回 `a ** b`。

operator.**rshift**(*a*, *b*)
operator.**__rshift__**(*a*, *b*)
返回 *a* 右移 *b* 位的结果。

operator.**sub**(*a*, *b*)
operator.**__sub__**(*a*, *b*)
返回 `a - b`。

operator.**truediv**(*a*, *b*)
operator.**__truediv__**(*a*, *b*)
返回 `a / b` 例如 2/3 将等于.66 而不是 0。这也被称为“真”除法。

operator.**xor**(*a*, *b*)

operator.**__xor__**(*a*, *b*)
返回 *a* 和 *b* 按位异或的结果。

适用于序列的操作（其中一些也适用于映射）包括：

operator.**concat**(*a*, *b*)
operator.**__concat__**(*a*, *b*)
对于序列 *a* 和 *b*，返回 `a + b`。

operator.**contains**(*a*, *b*)
operator.**__contains__**(*a*, *b*)
返回 `b in a` 检测的结果。请注意操作数是反序的。

operator.**countOf**(*a*, *b*)
返回 *b* 在 *a* 中的出现次数。

operator.**delitem**(*a*, *b*)
operator.**__delitem__**(*a*, *b*)
移除 *a* 中索引号为 *b* 的值。

operator.**getitem**(*a*, *b*)
operator.**__getitem__**(*a*, *b*)
返回 *a* 中索引为 *b* 的值。

operator.**indexOf**(*a*, *b*)
返回 *b* 在 *a* 中首次出现所在的索引号。

operator.**setitem**(*a*, *b*, *c*)
operator.**__setitem__**(*a*, *b*, *c*)
将 *a* 中索引号为 *b* 的值设为 *c*。

operator.**length_hint**(*obj*, *default=0*)
返回对象 *o* 的估计长度。首先尝试返回其实际长度，再使用 `object.__length_hint__()` 得出估计值，最后返回默认值。

3.4 版新加入.

operator 模块还定义了一些用于常规属性和条目查找的工具。这些工具适合用来编写快速字段提取器作为*map()*, *sorted()*, *itertools.groupby()* 或其他需要相应函数参数的函数的参数。

operator.**attrgetter**(*attr*)
operator.**attrgetter**(**attrs*)
返回一个可从操作数中获取 *attr* 的可调用对象。如果请求了一个以上的属性，则返回一个属性元组。属性名称还可包含点号。例如：

- 在 `f = attrgetter('name')` 之后，调用 `f(b)` 将返回 `b.name`。
- 在 `f = attrgetter('name', 'date')` 之后，调用 `f(b)` 将返回 `(b.name, b.date)`。
- 在 `f = attrgetter('name.first', 'name.last')` 之后，调用 `f(b)` 将返回 `(b.name.first, b.name.last)`。

等价于：

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
```

(下页继续)

(繼續上一頁)

```
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.**itemgetter**(*item*)
operator.**itemgetter**(**items*)

返回一个使用操作数的*__getitem__()* 方法从操作数中获取 *item* 的可调用对象。如果指定了多个条目，则返回一个查找值的元组。例如：

- 在 `f = itemgetter(2)` 之后，调用 `f(r)` 将返回 `r[2]`。
- 在 `g = itemgetter(2, 5, 3)` 之后，调用 `g(r)` 将返回 `(r[2], r[5], r[3])`。

等价于：

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

传入的条目可以为操作数的*__getitem__()* 所接受的任何类型。字典接受任意可哈希的值。列表、元组和字符串接受 index 或 slice 对象：

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1, 3, 5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2, None))('ABCDEFG')
'CDEFG'
>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

使用*itemgetter()* 从元组的记录中提取特定字段的例子：

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*, */*, **args*, ***kwargs*)

返回一个在操作数上调用 *name* 方法的可调用对象。如果给出额外的参数和/或关键字参数，它们也将被传给该方法。例如：

- 在 `f = methodcaller('name')` 之后，调用 `f(b)` 将返回 `b.name()`。

- 在 `f = methodcaller('name', 'foo', bar=1)` 之后，调用 `f(b)` 将返回 `b.name('foo', bar=1)`。

等价于:

```
def methodcaller(name, /, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

10.3.1 将运算符映射到函数

以下表格显示了抽象运算是如何对应于 Python 语法中的运算符和*operator*模块中的函数的。

运算	语法	函数
加法	`a + b`	`add(a, b)`
字符串拼接	`seq1 + seq2`	`concat(seq1, seq2)`
包含测试	`obj in seq`	`contains(seq, obj)`
除法	`a / b`	`truediv(a, b)`
除法	`a // b`	`floordiv(a, b)`
按位与	`a & b`	`and_(a, b)`
按位异或	`a ^ b`	`xor(a, b)`
按位取反	`~ a`	`invert(a)`
按位或	`a | b`	`or_(a, b)`
取幂	`a ** b`	`pow(a, b)`
标识	`a is b`	`is_(a, b)`
标识	`a is not b`	`is_not(a, b)`
索引赋值	`obj[k] = v`	`setitem(obj, k, v)`
索引删除	`del obj[k]`	`delitem(obj, k)`
索引取值	`obj[k]`	`getitem(obj, k)`
左移	`a << b`	`lshift(a, b)`
取模	`a % b`	`mod(a, b)`
乘法	`a * b`	`mul(a, b)`
矩阵乘法	`a @ b`	`matmul(a, b)`
取反（算术）	`- a`	`neg(a)`
取反（逻辑）	`not a`	`not_(a)`
正数	`+ a`	`pos(a)`
右移	`a >> b`	`rshift(a, b)`
切片赋值	`seq[i:j] = values`	`setitem(seq, slice(i, j), values)`
切片删除	`del seq[i:j]`	`delitem(seq, slice(i, j))`
切片取值	`seq[i:j]`	`getitem(seq, slice(i, j))`
字符串格式化	`s % obj`	`mod(s, obj)`
减法	`a - b`	`sub(a, b)`
真值测试	`obj`	`truth(obj)`
比较	`a < b`	`lt(a, b)`
比较	`a <= b`	`le(a, b)`
相等	`a == b`	`eq(a, b)`
不等	`a != b`	`ne(a, b)`
比较	`a >= b`	`ge(a, b)`
比较	`a > b`	`gt(a, b)`

10.3.2 原地运算符

许多运算都有 “原地” 版本。以下列出的是提供对原地运算符相比通常语法更底层访问的函数，例如*statement* `x += y` 相当于 `x = operator.iadd(x, y)`。换一种方式来讲就是 `z = operator.iadd(x, y)` 等价于语句块 `z = x; z += y`。

在这些例子中，请注意当调用一个原地方法时，运算和赋值是分成两个步骤来执行的。下面列出的原地函数只执行第一步即调用原地方法。第二步赋值则不加处理。

对于不可变的目标例如字符串、数字和元组，更新的值会被计算，但不会被再被赋值给输入变量：

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

对于可变的目标例如列表和字典，原地方法将执行更新，因此不需要后续赋值操作：

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

operator.**iadd**(*a*, *b*)
operator.**__iadd__**(*a*, *b*)
a = iadd(a, b) 等价于 a += b。

operator.**iand**(*a*, *b*)
operator.**__iand__**(*a*, *b*)
a = iand(a, b) 等价于 a &= b。

operator.**iconcat**(*a*, *b*)
operator.**__iconcat__**(*a*, *b*)
a = iconcat(a, b) 等价于 a += b 其中 *a* 和 *b* 为序列。

operator.**ifloordiv**(*a*, *b*)
operator.**__ifloordiv__**(*a*, *b*)
a = ifloordiv(a, b) 等价于 a //= b。

operator.**ilshift**(*a*, *b*)
operator.**__ilshift__**(*a*, *b*)
a = ilshift(a, b) 等价于 a <<= b。

operator.**imod**(*a*, *b*)
operator.**__imod__**(*a*, *b*)
a = imod(a, b) 等价于 a %= b。

operator.**imul**(*a*, *b*)
operator.**__imul__**(*a*, *b*)
a = imul(a, b) 等价于 a *= b。

operator.**imatmul**(*a*, *b*)
operator.**__imatmul__**(*a*, *b*)
a = imatmul(a, b) 等价于 a @= b。

3.5 版新加入.

operator.**ior**(*a*, *b*)

operator.**__ior__**(*a*, *b*)
    a = ior(a, b) 等价于 a |= b。

operator.**ipow**(*a*, *b*)
operator.**__ipow__**(*a*, *b*)
    a = ipow(a, b) 等价于 a **= b。

operator.**irshift**(*a*, *b*)
operator.**__irshift__**(*a*, *b*)
    a = irshift(a, b) 等价于 a >>= b。

operator.**isub**(*a*, *b*)
operator.**__isub__**(*a*, *b*)
    a = isub(a, b) 等价于 a -= b。

operator.**itruediv**(*a*, *b*)
operator.**__itruediv__**(*a*, *b*)
    a = itruediv(a, b) 等价于 a /= b。

operator.**ixor**(*a*, *b*)
operator.**__ixor__**(*a*, *b*)
    a = ixor(a, b) 等价于 a ^= b。

CHAPTER 11

檔案與目F存取

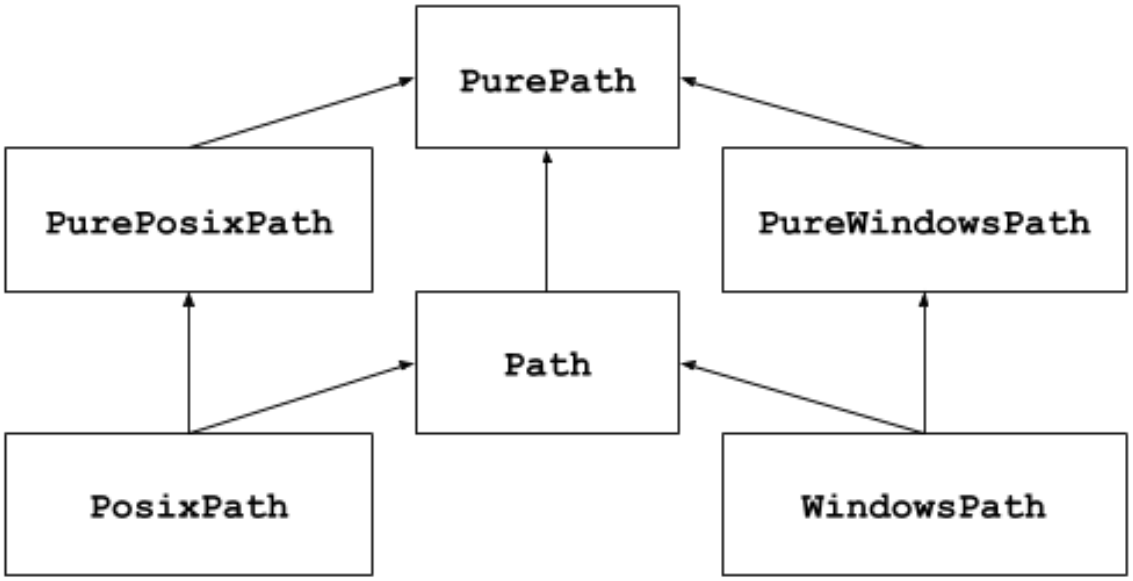
本章中描述的模块处理磁盘文件和目录。例如，有一些模块用于读取文件的属性，以可移植的方式操作路径以及创建临时文件。本章的完整模块列表如下：

11.1 pathlib --- 面向对象的文件系统路径

3.4 版新加入.

源代码 Lib/pathlib.py

该模块提供表示文件系统路径的类，其语义适用于不同的操作系统。路径类被分为提供纯计算操作而没有 I/O 的纯路径，以及从纯路径继承而来但提供 I/O 操作的具体路径。



如果以前从未用过此模块，或不确定哪个类适合完成任务，那要用的可能就是*Path*。它在运行代码的平台上实例化为具体路径。

在一些用例中纯路径很有用，例如：

1. 如果你想要在 Unix 设备上操作 Windows 路径（或者相反）。你不应在 Unix 上实例化一个*WindowsPath*，但是你可以实例化*PureWindowsPath*。
2. 你只想操作路径但不想实际访问操作系统。在这种情况下，实例化一个纯路径是有用的，因为它们没有任何访问操作系统的操作。

也参考：

PEP 428：pathlib 模块 -- 面向对象的文件系统路径。

也参考：

对于底层的路径字符串操作，你也可以使用*os.path* 模块。

11.1.1 基础使用

导入主类:

```
>>> from pathlib import Path
```

列出子目录:

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

列出当前目录树下的所有 Python 源代码文件:

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

在目录树中移动:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

查询路径的属性:

```
>>> q.exists()
True
>>> q.is_dir()
False
```

打开一个文件:

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 纯路径

纯路径对象提供了不实际访问文件系统的路径处理操作。有三种方式来访问这些类，也是不同的风格:

class pathlib.**PurePath**(**pathsegments*)

一个通用的类，代表当前系统的路径风格（实例化为*PurePosixPath*或者*PureWindowsPath*）：

```
>>> PurePath('setup.py')      # Running on a Unix machine
PurePosixPath('setup.py')
```

每一个 *pathsegments* 的元素可能是一个代表路径片段的字符串，一个返回字符串的实现了*os.PathLike*接口的对象，或者另一个路径对象:

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

当 *pathsegments* 为空的时候，假定为当前目录:

```
>>> PurePath()
PurePosixPath('.')
```

当给出一些绝对路径，最后一位将被当作锚（模仿*os.path.join()*的行为）：

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

但是，在 Windows 路径中，改变本地根目录并不会丢弃之前盘符的设置:

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

假斜线和单独的点都会被消除，但是双点（'..'）不会，以防改变符号链接的含义。

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

（一个很 naïve 的做法是让 `PurePosixPath('foo/../bar')` 等同于 `PurePosixPath('bar')`，如果 `foo` 是一个指向其他目录的符号链接那么这个做法就将出错）

纯路径对象实现了*os.PathLike* 接口，允许它们在任何接受此接口的地方使用。

3.6 版更變: 添加了*os.PathLike* 接口支持。

class pathlib.**PurePosixPath**(**pathsegments*)

一个*PurePath* 的子类，路径风格不同于 Windows 文件系统:

```
>>> PurePosixPath('/etc')
PurePosixPath('/etc')
```

pathsegments 参数的指定和*PurePath* 相同。

class pathlib.**PureWindowsPath**(**pathsegments*)

PurePath 的一个子类，路径风格为 Windows 文件系统路径:

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
```

pathsegments 参数的指定和*PurePath* 相同。

无论你正运行什么系统，你都可以实例化这些类，因为它们提供的操作不做任何系统调用。

通用性质

路径是不可变并可哈希的。相同风格的路径可以排序与比较。这些性质尊重对应风格的大小写转换语义:

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

不同风格的路径比较得到不等的结果并且无法被排序:

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
```

(下页继续)

(繼續上一頁)

```
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and 'PurePosixPath
↪'
```

运算符

斜杠 / 操作符有助于创建子路径，就像*os.path.join()* 一样:

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
```

文件对象可用于任何接受*os.PathLike* 接口实现的地方。

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

路径的字符串表示法为它自己原始的文件系统路径（以原生形式，例如在 Windows 下使用反斜杠）。你可以传递给任何需要字符串形式路径的函数。

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
>>> str(p)
'c:\\Program Files'
```

类似地，在路径上调用*bytes* 将原始文件系统路径作为字节对象给出，就像被*os.fsencode()* 编码一样:

```
>>> bytes(p)
b'/etc'
```

備F: 只推荐在 Unix 下调用*bytes*。在 Windows，unicode 形式是文件系统路径的规范表示法。

访问个别部分

为了访问路径独立的部分（组件），使用以下特征属性:

PurePath.**parts**

一个元组，可以访问路径的多个组件:

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')
```

(下页继续)

(繼續上一頁)

```
>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

（注意盘符和本地根目录是如何重组的）

方法和特征属性

纯路径提供以下方法和特征属性：

PurePath.**drive**
一个表示驱动器盘符或命名的字符串，如果存在:

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC 分享也被认作驱动器:

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

PurePath.**root**
一个表示（本地或全局）根的字符串，如果存在:

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

UNC 分享一样拥有根:

```
>>> PureWindowsPath('//host/share').root
'\\'
```

PurePath.**anchor**
驱动器和根的联合:

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

PurePath.**parents**
An immutable sequence providing access to the logical ancestors of the path:

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

PurePath.**parent**

此路径的逻辑父路径:

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

你不能超过一个 anchor 或空路径:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

備F: 这是一个单纯的词法操作，因此有以下行为:

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

如果要向上遍历任意文件系统路径，建议首先调用*Path.resolve()* 以解析符号链接并消除 ".." 部分。

PurePath.**name**

一个表示最后路径组件的字符串，排除了驱动器与根目录，如果存在的话:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

UNC 驱动器名不被考虑:

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

PurePath.**suffix**

最后一个组件的文件扩展名，如果存在:

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

PurePath.**suffixes**
路径的文件扩展名列表:

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

PurePath.**stem**
最后一个路径组件，除去后缀:

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.**as_posix**()
返回使用正斜杠（/）的路径字符串:

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.**as_uri**()
将路径表示为 file URL。如果并非绝对路径，抛出*ValueError*。

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = PureWindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

PurePath.**is_absolute**()
返回此路径是否为绝对路径。如果路径同时拥有驱动器符与根路径（如果风格允许）则将被认作绝对路径。

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.**is_relative_to**(**other*)

返回此路径是否相对于 *other* 的路径。

```
>>> p = PurePath('/etc/passwd')
>>> p.is_relative_to('/etc')
True
>>> p.is_relative_to('/usr')
False
```

3.9 版新加入.

PurePath.**is_reserved**()

在*PureWindowsPath*，如果路径是被 Windows 保留的则返回 `True`，否则 `False`。在*PurePosixPath*，总是返回 `False`。

```
>>> PureWindowsPath('nul').is_reserved()
True
>>> PurePosixPath('nul').is_reserved()
False
```

当保留路径上的文件系统被调用，则可能出现玄学失败或者意料之外的效应。

PurePath.**joinpath**(**other*)

调用此方法等同于将每个 *other* 参数中的项目连接在一起:

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

PurePath.**match**(*pattern*)

将此路径与提供的通配符风格的模式匹配。如果匹配成功则返回 `True`，否则返回 `False`。

如果 *pattern* 是相对的，则路径可以是相对路径或绝对路径，并且匹配是从右侧完成的:

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

如果 *pattern* 是绝对的，则路径必须是绝对的，并且路径必须完全匹配:

```
>>> PurePath('/a.py').match('/*.py')
True
>>> PurePath('a/b.py').match('/*.py')
False
```

与其他方法一样，是否大小写敏感遵循平台的默认规则:

```
>>> PurePosixPath('b.py').match('*.PY')
False
>>> PureWindowsPath('b.py').match('*.PY')
True
```

PurePath.**relative_to**(**other*)
计算此路径相对 *other* 表示路径的版本。如果不可计算，则抛出 ValueError:

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 694, in relative_to
    .format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not in the subpath of '/usr' OR one path is relative␣
↪and the other absolute.
```

注意：此函数是*PurePath* 的一部分并且适用于字符串。它不会检查或访问下层的文件结构。

PurePath.**with_name**(*name*)
返回一个新的路径并修改*name*。如果原本路径没有 name，ValueError 被抛出:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

PurePath.**with_stem**(*stem*)
返回一个带有修改后*stem* 的新路径。如果原路径没有名称，则会引发 ValueError:

```
>>> p = PureWindowsPath('c:/Downloads/draft.txt')
>>> p.with_stem('final')
PureWindowsPath('c:/Downloads/final.txt')
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_stem('lib')
PureWindowsPath('c:/Downloads/lib.gz')
>>> p = PureWindowsPath('c:/')
>>> p.with_stem('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 861, in with_stem
    return self.with_name(stem + self.suffix)
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 851, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

3.9 版新加入.

PurePath.**with_suffix**(*suffix*)
返回一个新的路径并修改*suffix*。如果原本的路径没有后缀，新的 *suffix* 则被追加以代替。如果 *suffix* 是空字符串，则原本的后缀被移除:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

11.1.3 具体路径

具体路径是纯路径的子类。除了后者提供的操作之外，它们还提供了对路径对象进行系统调用的方法。有三种方法可以实例化具体路径:

class pathlib.**Path**(**pathsegments*)
一个*PurePath*的子类，此类以当前系统的路径风格表示路径（实例化为*PosixPath*或*WindowsPath*）：

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments 参数的指定和*PurePath* 相同。

class pathlib.**PosixPath**(**pathsegments*)
一个*Path* 和*PurePosixPath* 的子类，此类表示一个非 Windows 文件系统的具体路径:

```
>>> PosixPath('/etc')
PosixPath('/etc')
```

pathsegments 参数的指定和*PurePath* 相同。

class pathlib.**WindowsPath**(**pathsegments*)
Path 和*PureWindowsPath* 的子类，从类表示一个 Windows 文件系统的具体路径:

```
>>> WindowsPath('c:/Program Files/')
WindowsPath('c:/Program Files')
```

pathsegments 参数的指定和*PurePath* 相同。

你只能实例化与当前系统风格相同的类（允许系统调用作用于不兼容的路径风格可能在应用程序中导致缺陷或失败）：

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
NotImplementedError: cannot instantiate 'WindowsPath' on your system
```

方法

除纯路径方法外，实体路径还提供以下方法。如果系统调用失败（例如因为路径不存在）这些方法中许多都会引发*OSError*。

3.8 版更變: 对于包含 OS 层级无法表示字符的路径，*exists()*, *is_dir()*, *is_file()*, *is_mount()*, *is_symlink()*, *is_block_device()*, *is_char_device()*, *is_fifo()*, *is_socket()* 现在将返回 `False` 而不是引发异常。

classmethod `Path.`**cwd**`()`
返回一个新的表示当前目录的路径对象（和*os.getcwd()* 返回的相同）：

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

classmethod `Path.`**home**`()`
返回一个表示当前用户家目录的新路径对象（和*os.path.expanduser()* 构造含 ~ 路径返回的相同）：

```
>>> Path.home()
PosixPath('/home/antoine')
```

Note that unlike *os.path.expanduser()*, on POSIX systems a *KeyError* or *RuntimeError* will be raised, and on Windows systems a *RuntimeError* will be raised if home directory can't be resolved.

3.5 版新加入.

`Path.`**stat**`()`
返回一个*os.stat_result* 对象，其中包含有关此路径的信息，例如*os.stat()*。结果会在每次调用此方法时重新搜索。

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

`Path.`**chmod**`(`*mode*`)`
改变文件的模式和权限，和*os.chmod()* 一样:

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

`Path.`**exists**`()`
此路径是否指向一个已存在的文件或目录:

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

備𝔽: 如果路径指向一个符号链接，*exists()* 返回此符号链接是否指向存在的文件或目录。

Path.**expanduser**()
返回展开了包含 ~ 和 ~user 的构造，就和*os.path.expanduser()* 一样:

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

Note that unlike *os.path.expanduser()*, on POSIX systems a *KeyError* or *RuntimeError* will be raised, and on Windows systems a *RuntimeError* will be raised if home directory can't be resolved.

3.5 版新加入.

Path.**glob**(*pattern*)
解析相对于此路径的通配符 *pattern*，产生所有匹配的文件:

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
```

"**" 模式表示 "此目录以及所有子目录，递归"。换句话说，它启用递归通配:

```
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

備𝔽: 在一个较大的目录树中使用"**" 模式可能会消耗非常多的时间。

引发一个审计事件 pathlib.Path.glob 附带参数 self, pattern。

Path.**group**()
返回拥有此文件的用户组。如果文件的 GID 无法在系统数据库中找到，将抛出*KeyError* 。

Path.**is_dir**()
如果路径指向一个目录（或者一个指向目录的符号链接）则返回 True，如果指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**is_file**()
如果路径指向一个正常的文件（或者一个指向正常文件的符号链接）则返回 True，如果指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**is_mount**()
如果路径是一个 挂载点 <*mount point*>：在文件系统中被其他不同的文件系统挂载的地点。在 POSIX 系统，此函数检查 *path* 的父级——path/.. 是否处于一个和 *path* 不同的设备中，或者 file:*path*/.. 和 *path* 是否指向相同设备的相同 i-node ——这能检测所有 Unix 以及 POSIX 变种上的挂载点。Windows 上未实现。

3.7 版新加入.

Path.**is_symlink**()
如果路径指向符号链接则返回 True，否则 False。

如果路径不存在也返回 False；其他错误（例如权限错误）被传播。

Path.**is_socket**()
如果路径指向一个 Unix socket 文件（或者指向 Unix socket 文件的符号链接）则返回 True，如果指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**is_fifo**()
如果路径指向一个先进先出存储（或者指向先进先出存储的符号链接）则返回 True ，指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**is_block_device**()
如果文件指向一个块设备（或者指向块设备的符号链接）则返回 True，指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**is_char_device**()
如果路径指向一个字符设备（或指向字符设备的符号链接）则返回 True，指向其他类型的文件则返回 False。

当路径不存在或者是一个破损的符号链接时也会返回 False；其他错误（例如权限错误）被传播。

Path.**iterdir**()
当路径指向一个目录时，产生该路径下的对象的路径:

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

The children are yielded in arbitrary order, and the special entries '.' and '..' are not included. If a file is removed from or added to the directory after creating the iterator, whether a path object for that file be included is unspecified.

Path.**lchmod**(*mode*)
就像*Path.chmod()* 但是如果路径指向符号链接则是修改符号链接的模式，而不是修改符号链接的目标。

Path.**lstat**()
就和*Path.stat()* 一样，但是如果路径指向符号链接，则是返回符号链接而不是目标的信息。

Path.**mkdir**(*mode=0o777*, *parents=False*, *exist_ok=False*)
新建给定路径的目录。如果给出了 *mode* ，它将与当前进程的 umask 值合并来决定文件模式和访问标志。如果路径已经存在，则抛出*FileExistsError*。

如果 *parents* 为真值，任何找不到的父目录都会伴随着此路径被创建；它们会以默认权限被创建，而不考虑 *mode* 设置（模仿 POSIX 的 mkdir -p 命令）。

如果 *parents* 为假值（默认），则找不到的父级目录会引发*FileNotFoundError*。

如果 *exist_ok* 为 false（默认），则在目标已存在的情况下抛出*FileExistsError*。

如果 *exist_ok* 为 true，则*FileExistsError*异常将被忽略（和 POSIX `mkdir -p` 命令行为相同），但是只有在最后一个路径组件不是现存的非目录文件时才生效。

3.5 版更變: *exist_ok* 形参被加入。

`Path.`**`open`**(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)

打开路径指向的文件，就像内置的*open()*函数所做的一样:

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

`Path.`**`owner`**()

返回拥有此文件的用户名。如果文件的 UID 无法在系统数据库中找到，则抛出*KeyError*。

`Path.`**`read_bytes`**()

以字节对象的形式返回路径指向的文件的二进制内容:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

3.5 版新加入.

`Path.`**`read_text`**(*encoding=None*, *errors=None*)

以字符串形式返回路径指向的文件的解码后文本内容。

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

文件先被打开然后关闭。有和*open()*一样的可选形参。

3.5 版新加入.

`Path.`**`readlink`**()

返回符号链接所指向的路径（即*os.readlink()*的返回值）:

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.readlink()
PosixPath('setup.py')
```

3.9 版新加入.

`Path.`**`rename`**(*target*)

将文件或目录重命名为给定的 *target*，并返回一个新的指向 *target* 的 Path 实例。在 Unix 上，如果 *target* 存在且为一个文件，如果用户有足够权限，则它将被静默地替换。*target* 可以是一个字符串或者另一个路径对象:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
```

(下页继续)

(繼續上一頁)

```
9
>>> target = Path('bar')
>>> p.rename(target)
PosixPath('bar')
>>> target.open().read()
'some text'
```

目标路径可能为绝对或相对路径。相对路径将被解释为相对于当前工作目录，而 不是相对于 Path 对象的目录。

3.8 版更變: 添加了返回值，返回新的 Path 实例。

Path.**replace**(*target*)

Rename this file or directory to the given *target*, and return a new Path instance pointing to *target*. If *target* points to an existing file or empty directory, it will be unconditionally replaced.

目标路径可能为绝对或相对路径。相对路径将被解释为相对于当前工作目录，而 不是相对于 Path 对象的目录。

3.8 版更變: 添加了返回值，返回新的 Path 实例。

Path.**resolve**(*strict=False*)

将路径绝对化，解析任何符号链接。返回新的路径对象:

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

”..”组件也将被消除（只有这一种方法这么做）:

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

如果路径不存在并且 *strict* 设为 True，则抛出*FileNotFoundError*。如果 *strict* 为 False，则路径将被尽可能地解析并且任何剩余部分都会被不检查是否存在地追加。如果在解析路径上发生无限循环，则抛出*RuntimeError*。

3.6 版新加入: 加入 *strict* 参数（3.6 之前的版本相当于 strict 值为 True）

Path.**rglob**(*pattern*)

这就像调用*Path.glob()* 并在给定的相对 *pattern* 前面添加了”**/”:

```
>>> sorted(Path().rglob("*.py"))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

引发一个审计事件 pathlib.Path.rglob 附带参数 self, pattern。

Path.**rmdir**()

移除此目录。此目录必须为空的。

Path.**samefile**(*other_path*)

返回此目录是否指向与可能是字符串或者另一个路径对象的 *other_path* 相同的文件。语义类似于*os.path.samefile()* 与*os.path.samestat()*。

如果两者都以同一原因无法访问，则抛出*OSError*。

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

3.5 版新加入.

Path.**symlink_to**(*target*, *target_is_directory=False*)
将此路径创建为指向 *target* 的符号链接。在 Windows 下，如果链接的目标是一个目录则 *target_is_directory* 必须为 true （默认为 False）。在 POSIX 下，*target_is_directory* 的值将被忽略。

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

備𝔽: 参数的顺序（link, target) 和*os.symlink()* 是相反的。

Path.**link_to**(*target*)
创建硬链接 *target* 指向此路径。

> **警告:** 此函数不是将此路径设为指向 *target* 的硬链接，这与函数和参数名的本义不同。参数顺序 (target, link) 与*Path.symlink_to()* 相反，而与*os.link()* 一致。

3.8 版新加入.

Path.**touch**(*mode=0o666*, *exist_ok=True*)
将给定的路径创建为文件。如果给出了 *mode* 它将与当前进程的 umask 值合并以确定文件的模式和访问标志。如果文件已经存在，则当 *exist_ok* 为 true 则函数仍会成功（并且将它的修改事件更新为当前事件），否则抛出*FileExistsError*。

Path.**unlink**(*missing_ok=False*)
移除此文件或符号链接。如果路径指向目录，则用*Path.rmdir()* 代替。

如果 *missing_ok* 为假值（默认），则如果路径不存在将会引发*FileNotFoundError*。

如果 *missing_ok* 为真值，则*FileNotFoundError* 异常将被忽略（和 POSIX rm -f 命令的行为相同）。

3.8 版更變: 增加了 *missing_ok* 形参。

Path.**write_bytes**(*data*)
将文件以二进制模式打开，写入 *data* 并关闭:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

一个同名的现存文件将被覆盖。

3.5 版新加入.

Path.**write_text**(*data*, *encoding=None*, *errors=None*)
将文件以文本模式打开，写入 *data* 并关闭:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

同名的现有文件会被覆盖。可选形参的含义与*open()* 的相同。

3.5 版新加入.

11.1.4 对应的 os 模块的工具

以下是一个映射了*os* 与*PurePath*/*Path* 对应相同的函数的表。

備🄴: 尽管*os.path.relpath()* 和*PurePath.relative_to()* 拥有相同的重叠的用例，但是它们语义相差很大，不能认为它们等价。

---

os 和 os.path	pathlib
os.path.abspath()	*Path.resolve()*
os.chmod()	*Path.chmod()*
os.mkdir()	*Path.mkdir()*
os.makedirs()	*Path.mkdir()*
os.rename()	*Path.rename()*
os.replace()	*Path.replace()*
os.rmdir()	*Path.rmdir()*
os.remove(), *os.unlink()*	*Path.unlink()*
os.getcwd()	*Path.cwd()*
os.path.exists()	*Path.exists()*
os.path.expanduser()	*Path.expanduser()* 和*Path.home()*
os.listdir()	*Path.iterdir()*
os.path.isdir()	*Path.is_dir()*
os.path.isfile()	*Path.is_file()*
os.path.islink()	*Path.is_symlink()*
os.link()	*Path.link_to()*
os.symlink()	*Path.symlink_to()*
os.readlink()	*Path.readlink()*
os.stat()	*Path.stat()*, *Path.owner()*, *Path.group()*
os.path.isabs()	*PurePath.is_absolute()*
os.path.join()	*PurePath.joinpath()*
os.path.basename()	*PurePath.name*
os.path.dirname()	*PurePath.parent*
os.path.samefile()	*Path.samefile()*
os.path.splitext()	*PurePath.suffix*

11.2 os.path --- 常用路径操作

Source code: Lib/posixpath.py (for POSIX) and Lib/ntpath.py (for Windows).

This module implements some useful functions on pathnames. To read or write files see *open()*, and for accessing the filesystem see the *os* module. The path parameters can be passed as strings, or bytes, or any object implementing the *os.PathLike* protocol.

与 unix shell 不同，Python 不执行任何 自动路径扩展。当应用程序需要类似 shell 的路径扩展时，可以显式调用诸如*expanduser()* 和*expandvars()* 之类的函数。（另请参见*glob* 模块。）

也参考:

pathlib 模块提供高级路径对象。

備𝔽: 所有这些函数都仅接受字节或字符串对象作为其参数。如果返回路径或文件名，则结果是相同类型的对象。

備𝔽: 由于不同的操作系统具有不同的路径名称约定，因此标准库中有此模块的几个版本。*os.path* 模块始终是适合 Python 运行的操作系统的路径模块，因此可用于本地路径。但是，如果操作的路径 总是以一种不同的格式显示，那么也可以分别导入和使用各个模块。它们都具有相同的接口：

- `posixpath` 用于 Unix 样式的路径
- `ntpath` 用于 Windows 路径

3.8 版更變: *exists()*、*lexists()*、*isdir()*、*isfile()*、*islink()* 和*ismount()* 现在遇到系统层面上不可表示的字符或字节的路径时，会返回 `False`，而不是抛出异常。

os.path.**abspath**(*path*)
: 返回路径 *path* 的绝对路径（标准化的）。在大多数平台上，这等同于用 `normpath(join(os.getcwd(), path))` 的方式调用*normpath()* 函数。

  3.6 版更變: 接受一个类路径对象。

os.path.**basename**(*path*)
: 返回路径 *path* 的基本名称。这是将 *path* 传入函数*split()* 之后，返回的一对值中的第二个元素。请注意，此函数的结果与 Unix **basename** 程序不同。**basename** 在 `'/foo/bar/'` 上返回 `'bar'`，而*basename()* 函数返回一个空字符串 (`''`)。

  3.6 版更變: 接受一个类路径对象。

os.path.**commonpath**(*paths*)
: 接受包含多个路径的序列 *paths*，返回 *paths* 的最长公共子路径。如果 *paths* 同时包含绝对路径和相对路径，或 *paths* 在不同的驱动器上，或 *paths* 为空，则抛出*ValueError* 异常。与*commonprefix()* 不同，本方法返回有效路径。

  可用性: Unix, Windows。

  3.5 版新加入.

  3.6 版更變: 接受一个类路径对象 序列。

os.path.**commonprefix**(*list*)
: 接受包含多个路径的 列表，返回所有路径的最长公共前缀（逐字符比较）。如果 列表为空，则返回空字符串 (`''`)。

備𠘨: 此函数是逐字符比较，因此可能返回无效路径。要获取有效路径，参见*commonpath()*。

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'

>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

3.6 版更變: 接受一个类路径对象。

os.path.**dirname**(*path*)
返回路径 *path* 的目录名称。这是将 *path* 传入函数*split()* 之后，返回的一对值中的第一个元素。

3.6 版更變: 接受一个类路径对象。

os.path.**exists**(*path*)
如果 *path* 指向一个已存在的路径或已打开的文件描述符，返回 True。对于失效的符号链接，返回 False。在某些平台上，如果使用*os.stat()* 查询到目标文件没有执行权限，即使 *path* 确实存在，本函数也可能返回 False。

3.3 版更變: *path* 现在可以是一个整数：如果该整数是一个已打开的文件描述符，返回 True，否则返回 False。

3.6 版更變: 接受一个类路径对象。

os.path.**lexists**(*path*)
如果 *path* 指向一个已存在的路径，返回 True。对于失效的符号链接，也返回 True。在缺失*os.lstat()* 的平台上等同于*exists()*。

3.6 版更變: 接受一个类路径对象。

os.path.**expanduser**(*path*)
在 Unix 和 Windows 上，将参数中开头部分的 ~ 或 ~user 替换为当前 用户 的家目录并返回。

在 Unix 上，开头的 ~ 会被环境变量 HOME 代替，如果变量未设置，则通过内置模块*pwd* 在 password 目录中查找当前用户的主目录。以 ~user 开头则直接在 password 目录中查找。

在 Windows 上，如果设置了 USERPROFILE，就使用这个变量，否则会将 HOMEPATH 和 HOMEDRIVE 结合在一起使用。以 ~user 开头则将上述方法生成路径的最后一截目录替换成 user。

如果展开路径失败，或者路径不是以波浪号开头，则路径将保持不变。

3.6 版更變: 接受一个类路径对象。

3.8 版更變: Windows 不再使用 HOME。

os.path.**expandvars**(*path*)
输入带有环境变量的路径作为参数，返回展开变量以后的路径。$name 或 ${name} 形式的子字符串被环境变量 *name* 的值替换。格式错误的变量名称和对不存在变量的引用保持不变。

在 Windows 上，除了 $name 和 ${name} 外，还可以展开 %name%。

3.6 版更變: 接受一个类路径对象。

os.path.**getatime**(*path*)
返回 *path* 的最后访问时间。返回值是一个浮点数，为纪元秒数（参见*time* 模块）。如果该文件不存在或不可访问，则抛出*OSError* 异常。

os.path.**getmtime**(*path*)
返回 *path* 的最后修改时间。返回值是一个浮点数，为纪元秒数（参见*time* 模块）。如果该文件不存在或不可访问，则抛出*OSError* 异常。

3.6 版更變: 接受一个类路径对象。

os.path.**getctime**(*path*)
返回 *path* 在系统中的 ctime，在有些系统（比如 Unix）上，它是元数据的最后修改时间，其他系统（比如 Windows）上，它是 *path* 的创建时间。返回值是一个数，为纪元秒数（参见*time* 模块）。如果该文件不存在或不可访问，则抛出*OSError* 异常。

3.6 版更變: 接受一个类路径对象。

os.path.**getsize**(*path*)
返回 *path* 的大小，以字节为单位。如果该文件不存在或不可访问，则抛出*OSError* 异常。

3.6 版更變: 接受一个类路径对象。

os.path.**isabs**(*path*)
如果 *path* 是一个绝对路径，则返回 `True`。在 Unix 上，它就是以斜杠开头，而在 Windows 上，它可以是去掉驱动器号后以斜杠（或反斜杠）开头。

3.6 版更變: 接受一个类路径对象。

os.path.**isfile**(*path*)
如果 *path* 是现有的 常规文件，则返回 `True`。本方法会跟踪符号链接，因此，对于同一路径，*islink()* 和*isfile()* 都可能为 `True`。

3.6 版更變: 接受一个类路径对象。

os.path.**isdir**(*path*)
如果 *path* 是现有的 目录，则返回 `True`。本方法会跟踪符号链接，因此，对于同一路径，*islink()* 和*isdir()* 都可能为 `True`。

3.6 版更變: 接受一个类路径对象。

os.path.**islink**(*path*)
如果 *path* 指向的现有 目录条目是一个符号链接，则返回 `True`。如果 Python 运行时不支持符号链接，则总是返回 `False`。

3.6 版更變: 接受一个类路径对象。

os.path.**ismount**(*path*)
如果路径 *path* 是 挂载点（文件系统中挂载其他文件系统的点），则返回 `True`。在 POSIX 上，该函数检查 *path* 的父目录 *path*`/..` 是否在与 *path* 不同的设备上，或者 *path*`/..` 和 *path* 是否指向同一设备上的同一 inode（这一检测挂载点的方法适用于所有 Unix 和 POSIX 变体）。本方法不能可靠地检测同一文件系统上的绑定挂载 (bind mount)。在 Windows 上，盘符和共享 UNC 始终是挂载点，对于任何其他路径，将调用 `GetVolumePathName` 来查看它是否与输入的路径不同。

3.4 版新加入: 支持在 Windows 上检测非根挂载点。

3.6 版更變: 接受一个类路径对象。

os.path.**join**(*path*, **paths*)
智能地拼接一个或多个路径部分。返回值是 *path* 和 **paths* 的所有成员的拼接，其中每个非空部分后面都紧跟一个目录分隔符，最后一个部分除外，这意味着如果最后一个部分为空，则结果将以分隔符结尾。如果某个部分为绝对路径，则之前的所有部分会被丢弃并从绝对路径部分开始继续拼接。

在 Windows 上，遇到绝对路径部分（例如 `r'\foo'`）时，不会重置盘符。如果某部分路径包含盘符，则会丢弃所有先前的部分，并重置盘符。请注意，由于每个驱动器都有一个“当前目录”，所以 `os.path.join("c:", "foo")` 表示驱动器 `C:` 上当前目录的相对路径 (`c:foo`)，而不是 `c:\foo`。

3.6 版更變: 接受一个类路径对象 用于 *path* 和 *paths* 。

os.path.**normcase**(*path*)
规范路径的大小写。在 Windows 上，将路径中的所有字符都转换为小写，并将正斜杠转换为反斜杠。在其他操作系统上返回原路径。

3.6 版更變: 接受一个类路径对象。

os.path.**normpath**(*path*)

通过折叠多余的分隔符和对上级目录的引用来标准化路径名，所以 A//B、A/B/、A/./B 和 A/foo/../B 都会转换成 A/B。这个字符串操作可能会改变带有符号链接的路径的含义。在 Windows 上，本方法将正斜杠转换为反斜杠。要规范大小写，请使用*normcase()*。

備🄵:

在 POSIX 系统上，根据 IEEE Std 1003.1 2013 Edition; 4.13 Pathname Resolution，如果一个路径名称以两个斜杠开始，则开始字符之后的第一个部分将以具体实现所定义的方式来解读，但是超过两个开始字符则将被视为单个字符。

3.6 版更變: 接受一个类路径对象。

---

os.path.**realpath**(*path*)

返回指定文件的规范路径，消除路径中存在的任何符号链接（如果操作系统支持）。

備🄵: 当发生符号链接循环时，返回的路径将是该循环的某个组成部分，但不能保证是哪个部分。

3.6 版更變: 接受一个类路径对象。

3.8 版更變: 在 Windows 上现在可以正确解析符号链接和交接点 (junction point)。

os.path.**relpath**(*path*, *start=os.curdir*)

返回从当前目录或可选的 *start* 目录至 *path* 的相对文件路径。这只是一个路径计算：不会访问文件系统来确认 *path* 或 *start* 是否存在或其性质。在 Windows 上，当 *path* 和 *start* 位于不同驱动器时将引发*ValueError*。

start 默认为*os.curdir*。

可用性: Unix, Windows。

3.6 版更變: 接受一个类路径对象。

os.path.**samefile**(*path1*, *path2*)

如果两个路径都指向相同的文件或目录，则返回 True。这由设备号和 inode 号确定，在任一路径上调用*os.stat()* 失败则抛出异常。

可用性: Unix, Windows。

3.2 版更變: 添加了 Windows 支持。

3.4 版更變: Windows 现在使用与其他所有平台相同的实现。

3.6 版更變: 接受一个类路径对象。

os.path.**sameopenfile**(*fp1*, *fp2*)

如果文件描述符 *fp1* 和 *fp2* 指向相同文件，则返回 True。

可用性: Unix, Windows。

3.2 版更變: 添加了 Windows 支持。

3.6 版更變: 接受一个类路径对象。

os.path.**samestat**(*stat1*, *stat2*)

如果 stat 元组 *stat1* 和 *stat2* 指向相同文件，则返回 True。这些 stat 元组可能是由*os.fstat()*、*os.lstat()* 或*os.stat()* 返回的。本函数实现了*samefile()* 和*sameopenfile()* 底层所使用的比较过程。

可用性: Unix, Windows。

3.4 版更變: 添加了 Windows 支持。

3.6 版更變: 接受一个类路径对象。

os.path.**split**(*path*)
将路径 *path* 拆分为一对，即 `(head, tail)`，其中，*tail* 是路径的最后一部分，而 *head* 里是除最后部分外的所有内容。*tail* 部分不会包含斜杠，如果 *path* 以斜杠结尾，则 *tail* 将为空。如果 *path* 中没有斜杠，*head* 将为空。如果 *path* 为空，则 *head* 和 *tail* 均为空。*head* 末尾的斜杠会被去掉，除非它是根目录（即它仅包含一个或多个斜杠）。在所有情况下，`join(head, tail)` 指向的位置都与 *path* 相同（但字符串可能不同）。另请参见函数*dirname()* 和*basename()*。

3.6 版更變: 接受一个类路径对象。

os.path.**splitdrive**(*path*)
将路径 *path* 拆分为一对，即 `(drive, tail)`，其中 *drive* 是挂载点或空字符串。在没有驱动器概念的系统上，*drive* 将始终为空字符串。在所有情况下，`drive + tail` 都与 *path* 相同。

在 Windows 上，本方法将路径拆分为驱动器/UNC 根节点和相对路径。

如果路径 path 包含盘符，则 drive 将包含冒号之前的所有内容包括冒号本身:

```
>>> splitdrive("c:/dir")
("c:", "/dir")
```

如果路径 path 包含 UNC 路径，则 drive 将包含主机名和 share，直至第四个分隔符但不包括该分隔符:

```
>>> splitdrive("//host/computer/dir")
("//host/computer", "/dir")
```

3.6 版更變: 接受一个类路径对象。

os.path.**splitext**(*path*)
将路径名称 *path* 拆分为 `(root, ext)` 对使得 `root + ext == path`，并且扩展名 *ext* 为空或以句点打头并最多只包含一个句点。

如果路径 path 不包含扩展名，则 *ext* 将为 `''`:

```
>>> splitext('bar')
('bar', '')
```

如果路径 path 包含扩展名，则 *ext* 将被设为该扩展名，包括打头的句点。请注意在其之前的句点将被忽略:

```
>>> splitext('foo.bar.exe')
('foo.bar', '.exe')
>>> splitext('/foo/bar.exe')
('/foo/bar', '.exe')
```

Leading periods of the last component of the path are considered to be part of the root:

```
>>> splitext('.cshrc')
('.cshrc', '')
>>> splitext('/foo/....jpg')
('/foo/....jpg', '')
```

3.6 版更變: 接受一个类路径对象。

os.path.**supports_unicode_filenames**
如果（在文件系统限制下）允许将任意 Unicode 字符串用作文件名，则为 `True`。

11.3 fileinput --- 迭代来自多个输入流的行

源代码: Lib/fileinput.py

此模块实现了一个辅助类和一些函数用来快速编写访问标准输入或文件列表的循环。如果你只想要读写一个文件请参阅*open()*。

典型用法为:

```
import fileinput
for line in fileinput.input():
    process(line)
```

此程序会迭代 sys.argv[1:] 中列出的所有文件内的行，如果列表为空则会使用 sys.stdin。如果有一个文件名为 '-'，它也会被替换为 sys.stdin 并且可选参数 *mode* 和 *openhook* 会被忽略。要指定替代文件列表，请将其作为第一个参数传给*input()*。也允许使用单个文件。

所有文件都默认以文本模式打开，但你可以通过在调用*input()* 或*FileInput* 时指定 *mode* 形参来重载此行为。如果在打开或读取文件时发生了 I/O 错误，将会引发*OSError*。

3.3 版更變: 原来会引发*IOError*；现在它是*OSError* 的别名。

如果 sys.stdin 被使用超过一次，则第二次之后的使用将不返回任何行，除非是被交互式的使用，或都是被显式地重置 (例如使用 sys.stdin.seek(0))。

空文件打开后将立即被关闭；它们在文件列表中会被注意到的唯一情况只有当最后打开的文件为空的时候。

反回的行不会对换行符做任何处理，这意味着文件中的最后一行可能不带换行符。

想要控制文件的打开方式，你可以通过将 *openhook* 形参传给*fileinput.input()* 或*FileInput()* 来提供一个打开钩子。此钩子必须为一个函数，它接受两个参数，*filename* 和 *mode*，并返回一个以相应模式打开的文件类对象。此模块已经提供了两个有用的钩子。

以下函数是此模块的初始接口：

fileinput.**input**(*files=None*, *inplace=False*, *backup=''*, ***, *mode='r'*, *openhook=None*)
: 创建一个*FileInput* 类的实例。该实例将被用作此模块中函数的全局状态，并且还将在迭代期间被返回使用。此函数的形参将被继续传递给*FileInput* 类的构造器。

  FileInput 实例可以在 with 语句中被用作上下文管理器。在这个例子中，*input* 在 with 语句结束后将会被关闭，即使发生了异常也是如此:

  ```
  with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
      for line in f:
          process(line)
  ```

  3.2 版更變: 可以被用作上下文管理器。

  3.8 版更變: 关键字形参 *mode* 和 *openhook* 现在是仅限关键字形参。

下列函数会使用*fileinput.input()* 所创建的全局状态；如果没有活动的状态，则会引发*RuntimeError*。

fileinput.**filename**()
: 返回当前被读取的文件名。在第一行被读取之前，返回 None。

fileinput.**fileno**()
: 返回以整数表示的当前文件 “文件描述符”。当未打开文件时（处在第一行和文件之间），返回 -1。

fileinput.**lineno**()
返回已被读取的累计行号。在第一行被读取之前，返回 0。在最后一个文件的最后一行被读取之后，返回该行的行号。

fileinput.**filelineno**()
返回当前文件中的行号。在第一行被读取之前，返回 0。在最后一个文件的最后一行被读取之后，返回此文件中该行的行号。

fileinput.**isfirstline**()
如果刚读取的行是其所在文件的第一行则返回 True，否则返回 False。

fileinput.**isstdin**()
如果最后读取的行来自 sys.stdin 则返回 True，否则返回 False。

fileinput.**nextfile**()
关闭当前文件以使下次迭代将从下一个文件（如果存在）读取第一行；不是从该文件读取的行将不会被计入累计行数。直到下一个文件的第一行被读取之后文件名才会改变。在第一行被读取之前，此函数将不会生效；它不能被用来跳过第一个文件。在最后一个文件的最后一行被读取之后，此函数将不再生效。

fileinput.**close**()
关闭序列。

此模块所提供的实现了序列行为的类同样也可用于子类化：

class fileinput.**FileInput**(*files=None*, *inplace=False*, *backup=''*, *, *mode='r'*, *openhook=None*)
类*FileInput* 是一个实现；它的方法*filename()*，*fileno()*，*lineno()*，*filelineno()*，*isfirstline()*，*isstdin()*，*nextfile()* 和*close()* 对应于此模块中具有相同名称的函数。此外它还有一个*readline()* 方法可返回下一个输入行，以及一个 __getitem__() 方法，该方法实现了序列行为。这种序列必须以严格的序列顺序来读写；随机读写和*readline()* 不可以被混用。

通过 *mode* 你可以指定要传给*open()* 的文件模式。它必须为 'r', 'rU', 'U' 和 'rb' 中的一个。

openhook 如果给出则必须为一个函数，它接受两个参数 *filename* 和 *mode*，并相应地返回一个打开的文件类对象。你不能同时使用 *inplace* 和 *openhook*。

FileInput 实例可以在 with 语句中被用作上下文管理器。在这个例子中，*input* 在 with 语句结束后将会被关闭，即使发生了异常也是如此：

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

3.2 版更變: 可以被用作上下文管理器。

3.4 版後已F用: 'rU' 和 'U' 模式。

3.8 版後已F用: 对 __getitem__() 方法的支持已弃用。

3.8 版更變: 关键字形参 *mode* 和 *openhook* 现在是仅限关键字形参。

可选的原地过滤: 如果传递了关键字参数 inplace=True 给*fileinput.input()* 或*FileInput* 构造器，则文件会被移至备份文件并将标准输出定向到输入文件（如果已存在与备份文件同名的文件，它将被静默地替换）。这使得编写一个能够原地重写其输入文件的过滤器成为可能。如果给出了 *backup* 形参 (通常形式为 backup='.<some extension>')，它将指定备份文件的扩展名，并且备份文件会被保留；默认情况下扩展名为 '.bak' 并且它会在输出文件关闭时被删除。在读取标准输入时原地过滤会被禁用。

此模块提供了以下两种打开文件钩子：

fileinput.**hook_compressed**(*filename*, *mode*)
使用*gzip* 和*bz2* 模块透明地打开 gzip 和 bzip2 压缩的文件（通过扩展名 '.gz' 和 '.bz2' 来识别）。如果文件扩展名不是 '.gz' 或 '.bz2'，文件会以正常方式打开（即使用*open()* 并且不带任何解压操作）。

使用示例: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed)`

`fileinput.`**`hook_encoded`**(*encoding*, *errors=None*)
返回一个通过*open()* 打开每个文件的钩子，使用给定的 *encoding* 和 *errors* 来读取文件。

使用示例: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`

3.6 版更變: 添加了可选的 *errors* 形参。

11.4 stat --- 解析 stat() 结果

源代码: Lib/stat.py

---

stat 模块定义了一些用于解析*os.stat()*, *os.fstat()* 和*os.lstat()* (如果它们存在) 输出结果的常量和函数。有关 `stat()`, `fstat()` 和 `lstat()` 调用的完整细节，请参阅你的系统文档。

3.4 版更變: stat 模块是通过 C 实现来支持的。

stat 模块定义了以下函数来检测特定文件类型:

`stat.`**`S_ISDIR`**(*mode*)
如果 mode 来自一个目录则返回非零值。

`stat.`**`S_ISCHR`**(*mode*)
如果 mode 来自一个字符特殊设备文件则返回非零值。

`stat.`**`S_ISBLK`**(*mode*)
如果 mode 来自一个块特殊设备文件则返回非零值。

`stat.`**`S_ISREG`**(*mode*)
如果 mode 来自一个常规文件则返回非零值。

`stat.`**`S_ISFIFO`**(*mode*)
如果 mode 来自一个 FIFO (命名管道) 则返回非零值。

`stat.`**`S_ISLNK`**(*mode*)
如果 mode 来自一个符号链接则返回非零值。

`stat.`**`S_ISSOCK`**(*mode*)
如果 mode 来自一个套接字则返回非零值。

`stat.`**`S_ISDOOR`**(*mode*)
如果 mode 来自一个门则返回非零值。

3.4 版新加入.

`stat.`**`S_ISPORT`**(*mode*)
如果 mode 来自一个事件端口则返回非零值。

3.4 版新加入.

`stat.`**`S_ISWHT`**(*mode*)
如果 mode 来自一个白输出则返回非零值。

3.4 版新加入.

定义了两个附加函数用于对文件模式进行更一般化的操作:

stat.**S_IMODE**(*mode*)
返回文件模式中可由*os.chmod()* 进行设置的部分 --- 即文件的 permission 位，加上 sticky 位、set-group-id 以及 set-user-id 位（在支持这些部分的系统上）。

stat.**S_IFMT**(*mode*)
返回文件模式中描述文件类型的部分（供上面的 S_IS*() 函数使用）。

通常，你应当使用 os.path.is*() 函数来检测文件的类型；这里提供的函数则适用于当你要对同一文件执行多项检测并且希望避免每项检测的 stat() 系统调用开销的情况。这些函数也适用于检测有关未被*os.path* 处理的信息，例如检测块和字符设备等。

示例:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
       calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.lstat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

另外还提供了一个附加的辅助函数用来将文件模式转换为人类易读的字符串:

stat.**filemode**(*mode*)
将文件模式转换为'-rwxrwxrwx' 形式的字符串。

3.3 版新加入.

3.4 版更變: 此函数支持*S_IFDOOR*, *S_IFPORT* and *S_IFWHT*。

以下所有变量是一些简单的符号索引，用于访问*os.stat()*, *os.fstat()* 或*os.lstat()* 所返回的 10 条目元组。

stat.**ST_MODE**
inode 保护模式。

stat.**ST_INO**
Inode 号

stat.**ST_DEV**
Inode 所在的设备。

stat.**ST_NLINK**
Inode 拥有的链接数量。

stat.**ST_UID**
所有者的用户 ID。

stat.**ST_GID**
所有者的用户组 ID。

stat.**ST_SIZE**
以字节为单位的普通文件大小；对于某些特殊文件则是所等待的数据量。

stat.**ST_ATIME**
上次访问的时间。

stat.**ST_MTIME**
上次修改的时间。

stat.**ST_CTIME**
操作系统所报告的"ctime"。在某些系统上（例如 Unix）是元数据的最后修改时间，而在其他系统上（例如 Windows）则是创建时间（请参阅系统平台的文档了解相关细节）。

对于“文件大小”的解析可因文件类型的不同而变化。对于普通文件就是文件的字节数。对于大部分种类的 Unix（特别包括 Linux）的 FIFO 和套接字来说，“大小”则是指在调用*os.stat()*, *os.fstat()* 或*os.lstat()* 时等待读取的字节数；这在某些时候很有用处，特别是在一个非阻塞的打开后轮询这些特殊文件中的一个时。其他字符和块设备的文件大小字段的含义还会有更多变化，具体取决于底层系统调用的实现方式。

以下变量定义了在*ST_MODE* 字段中使用的旗标。

使用上面的函数会比使用第一组旗标更容易移植：

stat.**S_IFSOCK**
套接字。

stat.**S_IFLNK**
符号链接。

stat.**S_IFREG**
普通文件。

stat.**S_IFBLK**
块设备。

stat.**S_IFDIR**
目录。

stat.**S_IFCHR**
字符设备。

stat.**S_IFIFO**
先进先出。

stat.**S_IFDOOR**
门。

3.4 版新加入.

stat.**S_IFPORT**
事件端口。

3.4 版新加入.

stat.**S_IFWHT**
白输出。

3.4 版新加入.

備𝔽: *S_IFDOOR*, *S_IFPORT* or *S_IFWHT* 等文件类型在不受系统平台支持时会被定义为 0。

以下旗标还可以*os.chmod()* 的在 *mode* 参数中使用：

stat.**S_ISUID**
设置 UID 位。

stat.**S_ISGID**
设置分组 ID 位。这个位有几种特殊用途。对于目录它表示该目录将使用 BSD 语义：在其中创建的文件将从目录继承其分组 ID，而不是从创建进程的有效分组 ID 继承，并且在其中创建的目录也将设置*S_ISGID* 位。对于没有设置分组执行位 (*S_IXGRP*) 的文件，设置分组 ID 位表示强制性文件/记录锁定 (另请参见*S_ENFMT*)。

stat.**S_ISVTX**
固定位。当对目录设置该位时则意味着此目录中的文件只能由文件所有者、目录所有者或特权进程来重命名或删除。

stat.**S_IRWXU**
文件所有者权限的掩码。

stat.**S_IRUSR**
所有者具有读取权限。

stat.**S_IWUSR**
所有者具有写入权限。

stat.**S_IXUSR**
所有者具有执行权限。

stat.**S_IRWXG**
组权限的掩码。

stat.**S_IRGRP**
组具有读取权限。

stat.**S_IWGRP**
组具有写入权限。

stat.**S_IXGRP**
组具有执行权限。

stat.**S_IRWXO**
其他人（不在组中）的权限掩码。

stat.**S_IROTH**
其他人具有读取权限。

stat.**S_IWOTH**
其他人具有写入权限。

stat.**S_IXOTH**
其他人具有执行权限。

stat.**S_ENFMT**
System V 执行文件锁定。此旗标是与*S_ISGID* 共享的：文件/记录锁定会针对未设置分组执行位 (*S_IXGRP*) 的文件强制执行。

stat.**S_IREAD**
Unix V7 中*S_IRUSR* 的同义词。

stat.**S_IWRITE**
Unix V7 中*S_IWUSR* 的同义词。

stat.**S_IEXEC**
Unix V7 中*S_IXUSR* 的同义词。

以下旗标可以在*os.chflags()* 的 *flags* 参数中使用：

stat.**UF_NODUMP**
不要转储文件。

stat.**UF_IMMUTABLE**
文件不能被更改。

stat.**UF_APPEND**
文件只能被附加。

stat.**UF_OPAQUE**
当通过联合堆栈查看时，目录是不透明的。

stat.**UF_NOUNLINK**
文件不能重命名或删除。

stat.**UF_COMPRESSED**
The file is stored compressed (macOS 10.6+).

stat.**UF_HIDDEN**
The file should not be displayed in a GUI (macOS 10.5+).

stat.**SF_ARCHIVED**
文件可能已存档。

stat.**SF_IMMUTABLE**
文件不能被更改。

stat.**SF_APPEND**
文件只能被附加。

stat.**SF_NOUNLINK**
文件不能重命名或删除。

stat.**SF_SNAPSHOT**
文件有一个快照文件

See the *BSD or macOS systems man page *chflags(2)* for more information.

在 Windows 上，以下文件属性常量可被用来检测*os.stat()* 所返回的 `st_file_attributes` 成员中的位。请参阅 Windows API 文档 了解有关这些常量含义的详情。

stat.**FILE_ATTRIBUTE_ARCHIVE**
stat.**FILE_ATTRIBUTE_COMPRESSED**
stat.**FILE_ATTRIBUTE_DEVICE**
stat.**FILE_ATTRIBUTE_DIRECTORY**
stat.**FILE_ATTRIBUTE_ENCRYPTED**
stat.**FILE_ATTRIBUTE_HIDDEN**
stat.**FILE_ATTRIBUTE_INTEGRITY_STREAM**
stat.**FILE_ATTRIBUTE_NORMAL**
stat.**FILE_ATTRIBUTE_NOT_CONTENT_INDEXED**
stat.**FILE_ATTRIBUTE_NO_SCRUB_DATA**
stat.**FILE_ATTRIBUTE_OFFLINE**
stat.**FILE_ATTRIBUTE_READONLY**
stat.**FILE_ATTRIBUTE_REPARSE_POINT**

stat.**FILE_ATTRIBUTE_SPARSE_FILE**
stat.**FILE_ATTRIBUTE_SYSTEM**
stat.**FILE_ATTRIBUTE_TEMPORARY**
stat.**FILE_ATTRIBUTE_VIRTUAL**
3.5 版新加入.

在 Windows 上，以下常量可被用来与*os.lstat()* 所返回的 st_reparse_tag 成员进行比较。这些是最主要的常量，而不是详尽的清单。

stat.**IO_REPARSE_TAG_SYMLINK**
stat.**IO_REPARSE_TAG_MOUNT_POINT**
stat.**IO_REPARSE_TAG_APPEXECLINK**
3.8 版新加入.

11.5 filecmp --- 文件及目录的比较

源代码: Lib/filecmp.py

---

filecmp 模块定义了用于比较文件及目录的函数，并且可以选取多种关于时间和准确性的折衷方案。对于文件的比较，另见*difflib* 模块。

filecmp 模块定义了如下函数:

filecmp.**cmp**(*f1*, *f2*, *shallow=True*)
比较名为 *f1* 和 *f2* 的文件，如果它们似乎相等则返回 True ，否则返回 False 。

如果 *shallow* 为真值且两个文件的*os.stat()* 签名信息（文件类型、大小和修改时间）一致，则文件会被视为相同。

在其他情况下，如果文件大小或内容不同则它们会被视为不同。

需要注意，没有外部程序被该函数调用，这赋予了该函数可移植性与效率。

该函数会缓存过去的比较及其结果，且在文件的*os.stat()* 信息变化后缓存条目失效。所有的缓存可以通过*clear_cache()* 清除。

filecmp.**cmpfiles**(*dir1*, *dir2*, *common*, *shallow=True*)
比较在两个目录 *dir1* 和 *dir2* 中，由 *common* 所确定名称的文件。

返回三组文件名列表：*match*, *mismatch*, *errors* 。*match* 含有相匹配的文件，*mismatch* 含有那些不匹配的，然后 *errors* 列出那些未被比较文件的名称。如果文件不存在于两目录中的任一个，或者用户缺少读取它们的权限，又或者因为其他的一些原因而无法比较，那么这些文件将会被列在 *errors* 中。

参数 *shallow* 具有同*filecmp.cmp()* 一致的含义与默认值。

例如，cmpfiles('a', 'b', ['c', 'd/e']) 将会比较 a/c 与 b/c 以及 a/d/e 与 b/d/e 。'c' 和 'd/e' 将会各自出现在返回的三个列表里的某一个列表中。

filecmp.**clear_cache**()
清除 filecmp 缓存。如果一个文件过快地修改，以至于超过底层文件系统记录修改时间的精度，那么该函数可能有助于比较该类文件。

3.4 版新加入.

11.5.1 dircmp 类

class filecmp.**dircmp**(*a*, *b*, *ignore=None*, *hide=None*)
创建一个用于比较目录 *a* 和 *b* 的新的目录比较对象。*ignore* 是需要忽略的文件名列表，且默认为*filecmp.DEFAULT_IGNORES* 。*hide* 是需要隐藏的文件名列表，且默认为 [os.curdir, os.pardir] 。

dircmp 类如*filecmp.cmp()* 中所描述的那样对文件进行 *shallow* 比较。

dircmp 类提供以下方法：

report()
将 *a* 与 *b* 之间的比较打印（到*sys.stdout* ）。

report_partial_closure()
打印 *a* 与 *b* 及共同直接子目录的比较结果。

report_full_closure()
打印 *a* 与 *b* 及共同子目录比较结果（递归地）。

dircmp 类提供了一些有趣的属性，用以得到关于参与比较的目录树的各种信息。

需要注意，通过 __getattr__() 钩子，所有的属性将会惰性求值，因此如果只使用那些计算简便的属性，将不会有速度损失。

left
目录 *a* 。

right
目录 *b* 。

left_list
经 *hide* 和 *ignore* 过滤，目录 *a* 中的文件与子目录。

right_list
经 *hide* 和 *ignore* 过滤，目录 *b* 中的文件与子目录。

common
同时存在于目录 *a* 和 *b* 中的文件和子目录。

left_only
仅在目录 *a* 中的文件和子目录。

right_only
仅在目录 *b* 中的文件和子目录。

common_dirs
同时存在于目录 *a* 和 *b* 中的子目录。

common_files
同时存在于目录 *a* 和 *b* 中的文件。

common_funny
在目录 *a* 和 *b* 中类型不同的名字，或者那些*os.stat()* 报告错误的名字。

same_files
在目录 *a* 和 *b* 中使用类的文件比较操作符相等的文件。

diff_files
在目录 *a* 和 *b* 中，根据类的文件比较操作符判定内容不等的文件。

funny_files
在目录 *a* 和 *b* 中无法比较的文件。

subdirs
一个将*common_dirs* 中名称映射为*dircmp* 对象的字典。

filecmp.**DEFAULT_IGNORES**
3.4 版新加入.

默认被*dircmp* 忽略的目录列表。

下面是一个简单的例子，使用 subdirs 属性递归搜索两个目录以显示公共差异文件：

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

11.6 tempfile --- 生成临时文件和目录

源代码： Lib/tempfile.py

该模块用于创建临时文件和目录，它可以跨平台使用。*TemporaryFile*、*NamedTemporaryFile*、*TemporaryDirectory* 和*SpooledTemporaryFile* 是带有自动清理功能的高级接口，可用作上下文管理器。*mkstemp()* 和*mkdtemp()* 是低级函数，使用完毕需手动清理。

所有由用户调用的函数和构造函数都带有参数，这些参数可以设置临时文件和临时目录的路径和名称。该模块生成的文件名包括一串随机字符，在公共的临时目录中，这些字符可以让创建文件更加安全。为了保持向后兼容性，参数的顺序有些奇怪。所以为了代码清晰，建议使用关键字参数。

这个模块定义了以下内容供用户调用：

tempfile.**TemporaryFile**(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)
返回一个*file-like object* （文件类对象）作为临时存储区域。创建该文件使用了与*mkstemp()* 相同的安全规则。它将在关闭后立即销毁（包括垃圾回收机制关闭该对象时）。在 Unix 下，该文件在目录中的条目根本不创建，或者创建文件后立即就被删除了，其他平台不支持此功能。您的代码不应依赖使用此功能创建的临时文件名称，因为它在文件系统中的名称可能是可见的，也可能是不可见的。

生成的对象可以用作上下文管理器（参见*示例*）。完成上下文或销毁临时文件对象后，临时文件将从文件系统中删除。

mode 参数默认值为 'w+b'，所以创建的文件不用关闭，就可以读取或写入。因为用的是二进制模式，所以无论存的是什么数据，它在所有平台上都表现一致。*buffering*、*encoding*、*errors* 和 *newline* 的含义与*open()* 中的相同。

参数 *dir*、*prefix* 和 *suffix* 的含义和默认值都与它们在*mkstemp()* 中的相同。

在 POSIX 平台上，它返回的对象是真实的文件对象。在其他平台上，它是一个文件类对象 (file-like object)，它的 file 属性是底层的真实文件对象。

如果可用，则使用*os.O_TMPFILE* 标志（仅限于 Linux，需要 3.11 及更高版本的内核）。

On platforms that are neither Posix nor Cygwin, TemporaryFile is an alias for NamedTemporaryFile.

引发一个 tempfile.mkstemp 审计事件，附带参数 fullpath。

3.5 版更變: 如果可用，现在用的是*os.O_TMPFILE* 标志。

3.8 版更變: 添加了 *errors* 参数。

tempfile.**NamedTemporaryFile**(*mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*, ***, *errors=None*)

This function operates exactly as *TemporaryFile()* does, except that the file is guaranteed to have a visible name in the file system (on Unix, the directory entry is not unlinked). That name can be retrieved from the name attribute of the returned file-like object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows). If *delete* is true (the default), the file is deleted as soon as it is closed. The returned object is always a file-like object whose file attribute is the underlying true file object. This file-like object can be used in a with statement, just like a normal file.

引发一个 tempfile.mkstemp 审计事件，附带参数 fullpath。

3.8 版更變: 添加了 *errors* 参数。

tempfile.**SpooledTemporaryFile**(*max_size=0*, *mode='w+b'*, *buffering=-1*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, ***, *errors=None*)

此函数执行的操作与*TemporaryFile()* 完全相同，但会将数据缓存在内存中，直到文件大小超过 *max_size*，或调用文件的 fileno() 方法为止，此时数据会被写入磁盘，并且写入操作与*TemporaryFile()* 相同。

此函数生成的文件对象有一个额外的方法——rollover()，可以忽略文件大小，让文件立即写入磁盘。

返回的对象是文件类对象 (file-like object)，它的 _file 属性是*io.BytesIO* 或*io.TextIOWrapper* 对象（取决于指定的是二进制模式还是文本模式）或真实的文件对象（取决于是否已调用 rollover()）。文件类对象可以像普通文件一样在 with 语句中使用。

3.3 版更變: 现在，文件的 truncate 方法可接受一个 size 参数。

3.8 版更變: 添加了 *errors* 参数。

tempfile.**TemporaryDirectory**(*suffix=None*, *prefix=None*, *dir=None*)

此函数会安全地创建一个临时目录，且使用与*mkdtemp()* 相同的规则。此函数返回的对象可用作上下文管理器（参见*示例*）。完成上下文或销毁临时目录对象后，新创建的临时目录及其所有内容将从文件系统中删除。

可以从返回对象的 name 属性中找到临时目录的名称。当返回的对象用作上下文管理器时，这个 name 会作为 with 语句中 as 子句的目标（如果有 as 的话）。

可以调用 cleanup() 方法来手动清理目录。

引发一个 tempfile.mkdtemp 审计事件，附带参数 fullpath。

3.2 版新加入.

tempfile.**mkstemp**(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

以最安全的方式创建一个临时文件。假设所在平台正确实现了*os.open()* 的*os.O_EXCL* 标志，则创建文件时不会有竞争的情况。该文件只能由创建者读写，如果所在平台用权限位来标记文件是否可执行，那么没有人有执行权。文件描述符不会过继给子进程。

与*TemporaryFile()* 不同，*mkstemp()* 用户用完临时文件后需要自行将其删除。

如果 *suffix* 不是 None 则文件名将以该后缀结尾，是 None 则没有后缀。*mkstemp()* 不会在文件名和后缀之间加点，如果需要加一个点号，请将其放在 *suffix* 的开头。

如果 *prefix* 不是 None，则文件名将以该前缀开头，是 None 则使用默认前缀。默认前缀是*gettempprefix()* 或*gettempprefixb()* 函数的返回值（自动调用合适的函数）。

如果 *dir* 不为 None，则在指定的目录创建文件，是 None 则使用默认目录。默认目录是从一个列表中选择出来的，这个列表不同平台不一样，但是用户可以设置 *TMPDIR*、*TEMP* 或 *TMP* 环境变量来设置目录的位置。因此，不能保证生成的临时文件路径很规范，比如，通过 os.popen() 将路径传递给外部命令时仍需要加引号。

如果 *suffix*、*prefix* 和 *dir* 中的任何一个不是 None，就要保证它们是同一数据类型。如果它们是 bytes，则返回的名称的类型就是 bytes 而不是 str。如果确实要用默认参数，但又想要返回值是 bytes 类型，请传入 suffix=b''。

如果指定了 *text* 且为真值，文件会以文本模式打开。否则，文件（默认）会以二进制模式打开。

mkstemp() 返回一个元组，元组中第一个元素是句柄，它是一个系统级句柄，指向一个打开的文件（等同于*os.open()* 的返回值），第二元素是该文件的绝对路径。

引发一个 tempfile.mkstemp 审计事件，附带参数 fullpath。

3.5 版更變: 现在，*suffix*、*prefix* 和 *dir* 可以以 bytes 类型按顺序提供，以获得 bytes 类型的返回值。之前只允许使用 str。*suffix* 和 *prefix* 现在可以接受 None，并且默认为 None 以使用合适的默认值。

3.6 版更變: *dir* 参数现在可接受一个路径类对象 (*path-like object*)。

tempfile.**mkdtemp**(*suffix=None*, *prefix=None*, *dir=None*)
以最安全的方式创建一个临时目录，创建该目录时不会有竞争的情况。该目录只能由创建者读取、写入和搜索。

mkdtemp() 用户用完临时目录后需要自行将其删除。

prefix、*suffix* 和 *dir* 的含义与它们在*mkstemp()* 中的相同。

mkdtemp() 返回新目录的绝对路径。

引发一个 tempfile.mkdtemp 审计事件，附带参数 fullpath。

3.5 版更變: 现在，*suffix*、*prefix* 和 *dir* 可以以 bytes 类型按顺序提供，以获得 bytes 类型的返回值。之前只允许使用 str。*suffix* 和 *prefix* 现在可以接受 None，并且默认为 None 以使用合适的默认值。

3.6 版更變: *dir* 参数现在可接受一个路径类对象 (*path-like object*)。

tempfile.**gettempdir**()
返回放置临时文件的目录的名称。这个方法的返回值就是本模块所有函数的 *dir* 参数的默认值。

Python 搜索标准目录列表，以找到调用者可以在其中创建文件的目录。这个列表是：

1. TMPDIR 环境变量指向的目录。
2. TEMP 环境变量指向的目录。
3. TMP 环境变量指向的目录。
4. 与平台相关的位置：
   - 在 Windows 上，依次为 C:\TEMP、C:\TMP、\TEMP 和 \TMP。
   - 在所有其他平台上，依次为 /tmp、/var/tmp 和 /usr/tmp。
5. 不得已时，使用当前工作目录。

搜索的结果会缓存起来，参见下面*tempdir* 的描述。

tempfile.**gettempdirb**()
与*gettempdir()* 相同，但返回值为字节类型。

3.5 版新加入.

tempfile.**gettempprefix**()
返回用于创建临时文件的文件名前缀，它不包含目录部分。

tempfile.**gettempprefixb**()
与*gettempprefix()* 相同，但返回值为字节类型。

3.5 版新加入.

本模块使用一个全局变量来存储由*gettempdir()* 返回的临时文件目录路径。可以直接给它赋值，这样可以覆盖自动选择的路径，但是不建议这样做。本模块中的所有函数都带有一个 *dir* 参数，该参数可用于指定目录，这是推荐的方法。

tempfile.**tempdir**
当设置为 None 以外的其他值时，此变量将决定本模块所有函数的 *dir* 参数的默认值。

如果在调用除*gettempprefix()* 外的上述任何函数时 tempdir 为 None (默认值) 则它会按照*gettempdir()* 中所描述的算法来初始化。

11.6.1 示例

以下是*tempfile* 模块典型用法的一些示例:

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.6.2 已弃用的函数和变量

创建临时文件有一种历史方法，首先使用*mktemp()* 函数生成一个文件名，然后使用该文件名创建文件。不幸的是，这是不安全的，因为在调用*mktemp()* 与随后尝试创建文件的进程之间的时间里，其他进程可能会使用该名称创建文件。解决方案是将两个步骤结合起来，立即创建文件。这个方案目前被*mkstemp()* 和上述其他函数所采用。

tempfile.**mktemp**(*suffix=''*, *prefix='tmp'*, *dir=None*)
2.3 版後已𝔽用: 使用*mkstemp()* 来代替。

返回一个绝对路径，这个路径指向的文件在调用本方法时不存在。*prefix*、*suffix* 和 *dir* 参数与*mkstemp()* 中的同名参数类似，不同之处在于不支持字节类型的文件名，不支持 `suffix=None` 和 `prefix=None`。

警告： 使用此功能可能会在程序中引入安全漏洞。当你开始使用本方法返回的文件执行任何操作时，可能有人已经捷足先登了。*mktemp()* 的功能可以很轻松地用*NamedTemporaryFile()* 代替，当然需要传递 `delete=False` 参数:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.7 `glob` --- Unix 风格路径名模式扩展

源代码: Lib/glob.py

The *glob* module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but `*`, `?`, and character ranges expressed with `[]` will be correctly matched. This is done by using the *os.scandir()* and *fnmatch.fnmatch()* functions in concert, and not by actually invoking a subshell.

Note that files beginning with a dot (`.`) can only be matched by patterns that also start with a dot, unlike *fnmatch.fnmatch()* or *pathlib.Path.glob()*. (For tilde and shell variable expansion, use *os.path.expanduser()* and *os.path.expandvars()*.)

对于字面值匹配，请将原字符用方括号括起来。例如，`'[?]'` 将匹配字符 `'?'`。

也参考:

pathlib 模块提供高级路径对象。

glob.**glob**(*pathname*, *, *recursive=False*)

返回匹配 *pathname* 的可能为空的路径名列表，其中的元素必须为包含路径信息的字符串。*pathname* 可以是绝对路径 (如 `/usr/src/Python-1.5/Makefile`) 或相对路径 (如 `../../Tools/*/*.gif`)，并且可包含 shell 风格的通配符。结果也将包含无效的符号链接（与在 shell 中一样）。结果是否排序取决于具体文件系统。如果某个符合条件的文件在调用此函数期间被移除或添加，是否包括该文件的路径是没有规定的。

如果 *recursive* 为真值，则模式"`**`" 将匹配目录中的任何文件以及零个或多个目录、子目录和符号链接。如果模式加了一个*os.sep* 或*os.altsep* 则将不匹配文件。

引发一个审计事件 `glob.glob` 附带参数 `pathname`, `recursive`。

備F: 在一个较大的目录树中使用"`**`" 模式可能会消耗非常多的时间。

3.5 版更變: 支持使用"`**`" 的递归 glob。

glob.**iglob**(*pathname*, *, *recursive=False*)
返回一个*iterator*，它会产生与*glob()* 相同的结果，但不会实际地同时保存它们。

引发一个审计事件 glob.glob 附带参数 pathname, recursive。

glob.**escape**(*pathname*)
转义所有特殊字符 ('?', '*' 和 '[')。这适用于当你想要匹配可能带有特殊字符的任意字符串字面值的情况。在 drive/UNC 共享点中的特殊字符不会被转义，例如在 Windows 上 escape('//?/c:/Quo vadis?.txt') 将返回 '//?/c:/Quo vadis[?].txt'。

3.4 版新加入.

例如，考虑一个包含以下内容的目录：文件 1.gif, 2.txt, card.gif 以及一个子目录 sub 其中只包含一个文件 3.txt. *glob()* 将产生如下结果。请注意路径的任何开头部分都将被保留。:

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

如果目录包含以 . 打头的文件，它们默认将不会被匹配。例如，考虑一个包含 card.gif 和 .card.gif 的目录:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

也参考:

模块*fnmatch* Shell 风格文件名（而非路径）扩展

11.8 fnmatch --- Unix 文件名模式匹配

源代码: Lib/fnmatch.py

此模块提供了 Unix shell 风格的通配符，它们 并不等同于正则表达式（关于后者的文档参见*re* 模块）。shell 风格通配符所使用的特殊字符如下：

模式	意义
*	匹配所有
?	匹配任何单个字符
[seq]	匹配 *seq* 中的任何字符
[!seq]	匹配任何不在 *seq* 中的字符

对于字面值匹配，请将原字符用方括号括起来。例如，'[?]' 将匹配字符 '?'。

注意文件名分隔符 (Unix 上为 '/') 不会被此模块特别对待。请参见*glob* 模块了解文件名扩展 (*glob* 使用*filter()* 来匹配文件名的各个部分)。类似地，以一个句点打头的文件名也不会被此模块特别对待，可以通过 * 和 ? 模式来匹配。

fnmatch.**fnmatch**(*filename*, *pattern*)
检测 *filename* 字符串是否匹配 *pattern* 字符串，返回*True* 或*False*。两个形参都会使用*os.path.normcase()* 进行大小写正规化。*fnmatchcase()* 可被用于执行大小写敏感的比较，无论这是否为所在操作系统的标准。

这个例子将打印当前目录下带有扩展名 .txt 的所有文件名:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

fnmatch.**fnmatchcase**(*filename*, *pattern*)
检测 *filename* 是否匹配 *pattern*，返回*True* 或*False*；此比较是大小写敏感的，并且不会应用*os.path.normcase()*。

fnmatch.**filter**(*names*, *pattern*)
基于可迭代对象 *names* 中匹配 *pattern* 的元素构造一个列表。它等价于 [n for n in names if fnmatch(n, pattern)]，但实现得更有效率。

fnmatch.**translate**(*pattern*)
返回 shell 风格 *pattern* 转换成的正则表达式以便用于*re.match()*。

示例:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```

也参考:

模块*glob* Unix shell 风格路径扩展。

11.9 linecache --- 随机读写文本行

源代码: Lib/linecache.py

---

linecache 模块允许从一个 Python 源文件中获取任意的行，并会尝试使用缓存进行内部优化，常应用于从单个文件读取多行的场合。此模块被*traceback* 模块用来提取源码行以便包含在格式化的回溯中。

tokenize.open() 函数被用于打开文件。此函数使用*tokenize.detect_encoding()* 来获取文件的编码格式；如果未指明编码格式，则默认编码为 UTF-8。

linecache 模块定义了下列函数:

linecache.**getline**(*filename*, *lineno*, *module_globals=None*)
从名为 *filename* 的文件中获取 *lineno* 行，此函数绝不会引发异常 --- 出现错误时它将返回 '' (所有找到的行都将包含换行符作为结束)。

如果找不到名为 *filename* 的文件，此函数会先在 *module_globals* 中检查 **PEP 302** `__loader__`。如果存在这样的加载器并且它定义了 `get_source` 方法，则由该方法来确定源行 (如果 `get_source()` 返回 `None`，则该函数返回 '')。最后，如果 *filename* 是一个相对路径文件名，则它会在模块搜索路径 `sys.path` 中按条目的相对位置进行查找。

linecache.**clearcache**()
清空缓存。如果你不再需要之前使用*getline()* 从文件读取的行即可使用此函数。

linecache.**checkcache**(*filename=None*)
检查缓存有效性。如果缓存中的文件在磁盘上发生了改变，而你需要更新后的版本即可使用此函数。如果省略了 *filename*，它会检查缓存中的所有条目。

linecache.**lazycache**(*filename*, *module_globals*)
捕获有关某个非基于文件的模块的足够细节信息，以允许稍后再通过*getline()* 来获取其中的行，即使当稍后调用时 *module_globals* 为 `None`。这可以避免在实际需要读取行之前执行 I/O，也不必始终保持模块全局变量。

3.5 版新加入.

示例:

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.10 shutil --- 高阶文件操作

源代码： Lib/shutil.py

---

shutil 模块提供了一系列对文件和文件集合的高阶操作。特别是提供了一些支持文件拷贝和删除的函数。对于单个文件的操作，请参阅*os* 模块。

警告： 即便是高阶文件拷贝函数 (*shutil.copy()*, *shutil.copy2()*) 也无法拷贝所有的文件元数据。

在 POSIX 平台上，这意味着将丢失文件所有者和组以及 ACL 数据。在 Mac OS 上，资源钩子和其他元数据不被使用。这意味着将丢失这些资源并且文件类型和创建者代码将不正确。在 Windows 上，将不会拷贝文件所有者、ACL 和替代数据流。

11.10.1 目录和文件操作

shutil.**copyfileobj**(*fsrc*, *fdst*[, *length*])
将文件类对象 *fsrc* 的内容拷贝到文件类对象 *fdst*。整数值 *length* 如果给出则为缓冲区大小。特别地，*length* 为负值表示拷贝数据时不对源数据进行分块循环处理；默认情况下会分块读取数据以避免不受控制的内存消耗。请注意如果 *fsrc* 对象的当前文件位置不为 0，则只有从当前文件位置到文件末尾的内容会被拷贝。

shutil.**copyfile**(*src*, *dst*, *, *follow_symlinks=True*)
将名为 *src* 的文件的内容（不包括元数据）拷贝到名为 *dst* 的文件并以尽可能高效的方式返回 *dst*。*src* 和 *dst* 均为路径类对象或以字符串形式给出的路径名。

dst 必须是完整的目标文件名；对于接受目标目录路径的拷贝请参见*copy()*。如果 *src* 和 *dst* 指定了同一个文件，则将引发*SameFileError*。

目标位置必须是可写的；否则将引发*OSError* 异常。如果 *dst* 已经存在，它将被替换。特殊文件如字符或块设备以及管道无法用此函数来拷贝。

如果 *follow_symlinks* 为假值且 *src* 为符号链接，则将创建一个新的符号链接而不是拷贝 *src* 所指向的文件。

引发一个审计事件 shutil.copyfile 附带参数 src, dst。

3.3 版更變: 曾经是引发*IOError* 而不是*OSError*。增加了 *follow_symlinks* 参数。现在是返回 *dst*。

3.4 版更變: 引发*SameFileError* 而不是*Error*。由于前者是后者的子类，此改变是向后兼容的。

3.8 版更變: 可能会在内部使用平台专属的快速拷贝系统调用以更高效地拷贝文件。参见依赖于具体平台的高效拷贝操作 一节。

exception shutil.**SameFileError**
此异常会在*copyfile()* 中的源和目标为同一文件时被引发。

3.4 版新加入.

shutil.**copymode**(*src*, *dst*, *, *follow_symlinks=True*)
从 *src* 拷贝权限位到 *dst*。文件的内容、所有者和分组将不受影响。*src* 和 *dst* 均为路径类对象或字符串形式的路径名。如果 *follow_symlinks* 为假值，并且 *src* 和 *dst* 均为符号链接，*copymode()* 将尝试修改 *dst* 本身的模式（而非它所指向的文件）。此功能并不是在所有平台上均可用；请参阅*copystat()* 了解详情。如果*copymode()* 无法修改本机平台上的符号链接，而它被要求这样做，它将不做任何操作即返回。

引发一个审计事件 shutil.copymode 附带参数 src, dst。

3.3 版更變: 加入 *follow_symlinks* 参数。

shutil.**copystat**(*src*, *dst*, *, *follow_symlinks=True*)
从 *src* 拷贝权限位、最近访问时间、最近修改时间以及旗标到 *dst*。在 Linux 上，*copystat()* 还会在可能的情况下拷贝 “扩展属性”。文件的内容、所有者和分组将不受影响。*src* 和 *dst* 均为路径类对象或字符串形式的路径名。

如果 *follow_symlinks* 为假值，并且 *src* 和 *dst* 均指向符号链接，*copystat()* 将作用于符号链接本身而非该符号链接所指向的文件—从 *src* 符号链接读取信息，并将信息写入 *dst* 符号链接。

備F: 并非所有平台者提供检查和修改符号链接的功能。Python 本身可以告诉你哪些功能是在本机上可用的。

- 如果 os.chmod in os.supports_follow_symlinks 为 True，则*copystat()* 可以修改符号链接的权限位。
- 如果 os.utime in os.supports_follow_symlinks 为 True，则*copystat()* 可以修改符号链接的最近访问和修改时间。
- 如果 os.chflags in os.supports_follow_symlinks 为 True，则*copystat()* 可以修改符号链接的旗标。(os.chflags 不是在所有平台上均可用。)

在此功能部分或全部不可用的平台上，当被要求修改一个符号链接时，*copystat()* 将尽量拷贝所有内容。*copystat()* 一定不会返回失败信息。

更多信息请参阅*os.supports_follow_symlinks*。

引发一个审计事件 shutil.copystat 附带参数 src, dst。

3.3 版更變: 添加了 *follow_symlinks* 参数并且支持 Linux 扩展属性。

shutil.**copy**(*src*, *dst*, *, *follow_symlinks=True*)

Copies the file *src* to the file or directory *dst*. *src* and *dst* should be *path-like objects* or strings. If *dst* specifies a directory, the file will be copied into *dst* using the base filename from *src*. If *dst* specifies a file that already exists, it will be replaced. Returns the path to the newly created file.

如果 *follow_symlinks* 为假值且 *src* 为符号链接，则 *dst* 也将被创建为符号链接。如果 *follow_symlinks* 为真值且 *src* 为符号链接，*dst* 将成为 *src* 所指向的文件的一个副本。

copy() 会拷贝文件数据和文件的权限模式 (参见*os.chmod()*)。其他元数据，例如文件的创建和修改时间不会被保留。要保留所有原有的元数据，请改用*copy2()* 。

引发一个审计事件 shutil.copyfile 附带参数 src, dst。

引发一个审计事件 shutil.copymode 附带参数 src, dst。

3.3 版更變: 添加了 *follow_symlinks* 参数。现在会返回新创建文件的路径。

3.8 版更變: 可能会在内部使用平台专属的快速拷贝系统调用以更高效地拷贝文件。参见依赖于具体平台的高效拷贝操作 一节。

shutil.**copy2**(*src*, *dst*, *, *follow_symlinks=True*)

类似于*copy()*，区别在于*copy2()* 还会尝试保留文件的元数据。

当 *follow_symlinks* 为假值且 *src* 为符号链接时，*copy2()* 会尝试将来自 *src* 符号链接的所有元数据拷贝到新创建的 *dst* 符号链接。但是，此功能不是在所有平台上均可用。在此功能部分或全部不可用的平台上，*copy2()* 将尽量保留所有元数据；*copy2()* 一定不会由于无法保留文件元数据而引发异常。

copy2() 会使用*copystat()* 来拷贝文件元数据。请参阅*copystat()* 了解有关修改符号链接元数据的平台支持的更多信息。

引发一个审计事件 shutil.copyfile 附带参数 src, dst。

引发一个审计事件 shutil.copystat 附带参数 src, dst。

3.3 版更變: 添加了 *follow_symlinks* 参数，还会尝试拷贝扩展文件系统属性（目前仅限 Linux）。现在会返回新创建文件的路径。

3.8 版更變: 可能会在内部使用平台专属的快速拷贝系统调用以更高效地拷贝文件。参见依赖于具体平台的高效拷贝操作 一节。

shutil.**ignore_patterns**(**patterns*)

这个工厂函数会创建一个函数，它可被用作*copytree()* 的 *ignore* 可调用对象参数，以忽略那些匹配所提供的 glob 风格的 *patterns* 之一的文件和目录。参见以下示例。

shutil.**copytree**(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)

Recursively copy an entire directory tree rooted at *src* to a directory named *dst* and return the destination directory. All intermediate directories needed to contain *dst* will also be created by default.

目录的权限和时间会通过*copystat()* 来拷贝，单个文件则会使用*copy2()* 来拷贝。

如果 *symlinks* 为真值，源目录树中的符号链接会在新目录树中表示为符号链接，并且原链接的元数据在平台允许的情况下也会被拷贝；如果为假值或省略，则会将被链接文件的内容和元数据拷贝到新目录树。

当 *symlinks* 为假值时，如果符号链接所指向的文件不存在，则会在拷贝进程的末尾将一个异常添加到*Error* 异常中的错误列表。如果你希望屏蔽此异常那就将可选的 *ignore_dangling_symlinks* 旗标设为真值。请注意此选项在不支持*os.symlink()* 的平台上将不起作用。

如果给出了 *ignore*，它必须是一个可调用对象，该对象将接受*copytree()* 所访问的目录以及*os.listdir()* 所返回的目录内容列表作为其参数。由于*copytree()* 是递归地被调用的，*ignore* 可调用

对象对于每个被拷贝目录都将被调用一次。该可调用对象必须返回一个相对于当前目录的目录和文件名序列（即其第二个参数的子集）；随后这些名称将在拷贝进程中被忽略。*ignore_patterns()* 可被用于创建这种基于 glob 风格模式来忽略特定名称的可调用对象。

如果发生了（一个或多个）异常，将引发一个附带原因列表的*Error*。

如果给出了 *copy_function*，它必须是一个将被用来拷贝每个文件的可调用对象。它在被调用时会将源路径和目标路径作为参数传入。默认情况下，*copy2()* 将被使用，但任何支持同样签名（与*copy()* 一致）都可以使用。

If *dirs_exist_ok* is false (the default) and *dst* already exists, a *FileExistsError* is raised. If *dirs_exist_ok* is true, the copying operation will continue if it encounters existing directories, and files within the *dst* tree will be overwritten by corresponding files from the *src* tree.

引发一个审计事件 `shutil.copytree` 附带参数 `src`, `dst`。

3.3 版更變: 当 *symlinks* 为假值时拷贝元数据。现在会返回 *dst*。

3.2 版更變: Added the *copy_function* argument to be able to provide a custom copy function. Added the *ignore_dangling_symlinks* argument to silence dangling symlinks errors when *symlinks* is false.

3.8 版更變: 可能会在内部使用平台专属的快速拷贝系统调用以更高效地拷贝文件。参见依赖于具体平台的高效拷贝操作 一节。

3.8 版新加入: *dirs_exist_ok* 形参。

`shutil.`**`rmtree`**(*path*, *ignore_errors=False*, *onerror=None*)

删除一个完整的目录树；*path* 必须指向一个目录（但不能是一个目录的符号链接）。如果 *ignore_errors* 为真值，删除失败导致的错误将被忽略；如果为假值或是省略，此类错误将通过调用由 *onerror* 所指定的处理程序来处理，或者如果此参数被省略则将引发一个异常。

備🄵: 在支持必要的基于 fd 的函数的平台上，默认会使用*rmtree()* 的可防御符号链接攻击的版本。在其他平台上，*rmtree()* 较易遭受符号链接攻击：给定适当的时间和环境，攻击者可以操纵文件系统中的符号链接来删除他们在其他情况下无法访问的文件。应用程序可以使用*rmtree.avoids_symlink_attacks* 函数属性来确定此类情况具体是哪一些。

如果提供了 *onerror*，它必须为接受三个形参的可调用对象: *function*, *path* 和 *excinfo*。

第一个形参 *function* 是引发异常的函数；它依赖于具体的平台和实现。第二个形参 *path* 将是传递给 *function* 的路径名。第三个形参 *excinfo* 将是由*sys.exc_info()* 所返回的异常信息。由 *onerror* 所引发的异常将不会被捕获。

引发一个审计事件 `shutil.rmtree` 附带参数 `path`。

3.3 版更變: 添加了一个防御符号链接攻击的版本，如果平台支持基于 fd 的函数就会被使用。

3.8 版更變: 在 Windows 上将不会再在移除连接之前删除目录连接中的内容。

`rmtree.`**`avoids_symlink_attacks`**

指明当前平台和实现是否提供防御符号链接攻击的*rmtree()* 版本。目前它仅在平台支持基于 fd 的目录访问函数时才返回真值。

3.3 版新加入.

`shutil.`**`move`**(*src*, *dst*, *copy_function=copy2*)

递归地将一个文件或目录 (*src*) 移至另一位置 (*dst*) 并返回目标位置。

如果目标是已存在的目录，则 *src* 会被移至该目录下。如果目标已存在但不是目录，它可能会被覆盖，具体取决于*os.rename()* 的语义。

如果目标是在当前文件系统中，则会使用*os.rename()*。在其他情况下，*src* 将被拷贝至 *dst*，使用的函数为 *copy_function*，然后目标会被移除。对于符号链接，则将在 *dst* 之下或以其本身为名称创建一个指向 *src* 目标的新符号链接，并且 *src* 将被移除。

如果给出了 *copy_function*，则它必须为接受两个参数 *src* 和 *dst* 的可调用对象，并将在*os.rename()* 无法使用时被用来将 *src* 拷贝到 *dst*。如果源是一个目录，则会调用*copytree()*，并向它传入 `copy_function()`。默认的 *copy_function* 是*copy2()*。使用*copy()* 作为 *copy_function* 允许在无法附带拷贝元数据时让移动操作成功执行，但其代价是不拷贝任何元数据。

引发一个审计事件 `shutil.move` 附带参数 `src`, `dst`。

3.3 版更變: 为异类文件系统添加了显式的符号链接处理，以便使它适应 GNU 的 **mv** 的行为。现在会返回 *dst*。

3.5 版更變: 增加了 *copy_function* 关键字参数。

3.8 版更變: 可能会在内部使用平台专属的快速拷贝系统调用以更高效地拷贝文件。参见依赖于具体平台的高效拷贝操作 一节。

3.9 版更變: 接受一个*path-like object* 作为 *src* 和 *dst*。

`shutil.`**`disk_usage`**(*path*)
返回给定路径的磁盘使用统计数据，形式为一个*named tuple*，其中包含 *total*, *used* 和 *free* 属性，分别表示总计、已使用和未使用空间的字节数。*path* 可以是一个文件或是一个目录。

3.3 版新加入.

3.8 版更變: 在 Windows 上，*path* 现在可以是一个文件或目录。

可用性: Unix, Windows。

`shutil.`**`chown`**(*path*, *user=None*, *group=None*)
修改给定 *path* 的所有者 *user* 和/或 *group*。

user 可以是一个系统用户名或 uid；*group* 同样如此。要求至少有一个参数。

另请参阅下层的函数*os.chown()*。

引发一个审计事件 `shutil.chown` 附带参数 `path`, `user`, `group`。

Availability: Unix.

3.3 版新加入.

`shutil.`**`which`**(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)
返回当给定的 *cmd* 被调用时将要运行的可执行文件的路径。如果没有 *cmd* 会被调用则返回 `None`。

mode 是一个传递给*os.access()* 的权限掩码，在默认情况下将确定文件是否存在并且为可执行文件。

当未指定 *path* 时，将会使用*os.environ()* 的结果，返回"PATH" 的值或回退为*os.defpath*。

在 Windows 上当前目录总是会被添加为 *path* 的第一项，无论你是否使用默认值或提供你自己的路径，这是命令行终端在查找可执行文件时所采用的行为方式。此外，当在 *path* 中查找 *cmd* 时，还会检查 `PATHEXT` 环境变量。例如，如果你调用 `shutil.which("python")`，*which()* 将搜索 `PATHEXT` 来确定它要在 *path* 目录中查找 `python.exe`。例如，在 Windows 上:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

3.3 版新加入.

3.8 版更變: 现在可以接受*bytes* 类型。如果 *cmd* 的类型为*bytes*，结果的类型也将为*bytes*。

exception shutil.**Error**
此异常会收集在多文件操作期间所引发的异常。对于*copytree()*，此异常参数将是一个由三元组 (*srcname*, *dstname*, *exception*) 构成的列表。

依赖于具体平台的高效拷贝操作

从 Python 3.8 开始，所有涉及文件拷贝的函数 (*copyfile()*, *copy()*, *copy2()*, *copytree()* 以及*move()*) 将会使用平台专属的"fast-copy" 系统调用以便更高效地拷贝文件 (参见 bpo-33671)。"fast-copy" 意味着拷贝操作将发生于内核之中，避免像在"`outfd.write(infd.read())`" 中那样使用 Python 用户空间的缓冲区。

在 macOS 上将会使用 fcopyfile 来拷贝文件内容（不含元数据）。

在 Linux 上将会使用*os.sendfile()*。

在 Windows 上*shutil.copyfile()* 将会使用更大的默认缓冲区（1 MiB 而非 64 KiB）并且会使用基于*memoryview()* 的*shutil.copyfileobj()* 变种形式。

如果快速拷贝操作失败并且没有数据被写入目标文件，则 shutil 将在内部静默地回退到使用效率较低的*copyfileobj()* 函数。

3.8 版更變.

copytree 示例

An example that uses the *ignore_patterns()* helper:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

这将会拷贝除 `.pyc` 文件和以 `tmp` 打头的文件或目录以外的所有条目.

另一个使用 *ignore* 参数来添加记录调用的例子:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

rmtree 示例

这个例子演示了如何在 Windows 上删除一个目录树，其中部分文件设置了只读属性位。它会使用 onerror 回调函数来清除只读属性位并再次尝试删除。任何后续的失败都将被传播。

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)
```

(下页继续)

(繼續上一頁)

```
shutil.rmtree(directory, onerror=remove_readonly)
```

11.10.2 归档操作

3.2 版新加入.

3.5 版更變: 添加了对 *xztar* 格式的支持。

本模块也提供了用于创建和读取压缩和归档文件的高层级工具。它们依赖于*zipfile* 和*tarfile* 模块。

shutil.**make_archive**(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

创建一个归档文件（例如 zip 或 tar）并返回其名称。

base_name 是要创建的文件名称，包括路径，去除任何特定格式的扩展名。*format* 是归档格式：为”zip”（如果*zlib* 模块可用），”tar”，”gztar”（如果*zlib* 模块可用），”bztar”（如果*bz2* 模块可用）或”xztar”（如果*lzma* 模块可用）中的一个。

root_dir 是一个目录，它将作为归档文件的根目录，归档中的所有路径都将是它的相对路径；例如，我们通常会在创建归档之前用 chdir 命令切换到 *root_dir*。

base_dir 是我们要执行归档的起始目录；也就是说 *base_dir* 将成为归档中所有文件和目录共有的路径前缀。*base_dir* 必须相对于 *root_dir* 给出。请参阅使用 *base_dir* 的归档程序示例 了解如何同时使用 *base_dir* 和 *root_dir*。

root_dir 和 *base_dir* 默认均为当前目录。

如果 *dry_run* 为真值，则不会创建归档文件，但将要被执行的操作会被记录到 *logger*。

owner 和 *group* 将在创建 tar 归档文件时被使用。默认会使用当前的所有者和分组。

logger 必须是一个兼容 **PEP 282** 的对象，通常为*logging.Logger* 的实例。

verbose 参数已不再使用并进入弃用状态。

引发一个审计事件 `shutil.make_archive` 并附带参数 `base_name`, `format`, `root_dir`, `base_dir`。

備F: 这个函数不是线程安全的。

3.8 版更變: 现在对于通过 `format="tar"` 创建的归档文件将使用新式的 pax (POSIX.1-2001) 格式而非旧式的 GNU 格式。

shutil.**get_archive_formats**()

返回支持的归档格式列表。所返回序列中的每个元素为一个元组 `(name, description)`。

默认情况下*shutil* 提供以下格式:

- *zip*: ZIP 文件（如果*zlib* 模块可用）。
- *tar*: 未压缩的 tar 文件。对于新归档文件将使用 POSIX.1-2001 pax 格式。
- *gztar*: gzip 压缩的 tar 文件（如果*zlib* 模块可用）。
- *bztar*: bzip2 压缩的 tar 文件（如果*bz2* 模块可用）。
- *xztar*: xz 压缩的 tar 文件（如果*lzma* 模块可用）。

你可以通过使用*register_archive_format()* 注册新的格式或为任何现有格式提供你自己的归档器。

shutil.**register_archive_format**(*name*, *function*[, *extra_args*[, *description*]])
为 *name* 格式注册一个归档器。

function 是将被用来解包归档文件的可调用对象。该可调用对象将接收要创建文件的 *base_name*，再加上要归档内容的 *base_dir* (其默认值为*os.curdir*)。更多参数会被作为关键字参数传入: *owner*, *group*, *dry_run* 和 *logger* (与向*make_archive()* 传入的参数一致)。

如果给出了 *extra_args*，则其应为一个 (name, value) 对的序列，将在归档器可调用对象被使用时作为附加的关键字参数。

description 由*get_archive_formats()* 使用，它将返回归档器的列表。默认值为一个空字符串。

shutil.**unregister_archive_format**(*name*)
从支持的格式中移除归档格式 *name*。

shutil.**unpack_archive**(*filename*[, *extract_dir*[, *format*]])
解包一个归档文件。*filename* 是归档文件的完整路径。

extract_dir 是归档文件解包的目标目录名称。如果未提供，则将使用当前工作目录。

format 是归档格式: 应为"zip", "tar", "gztar", "bztar" 或"xztar" 之一。或者任何通过*register_unpack_format()* 注册的其他格式。如果未提供，*unpack_archive()* 将使用归档文件的扩展名来检查是否注册了对应于该扩展名的解包器。在未找到任何解包器的情况下，将引发*ValueError*。

引发一个审计事件 shutil.unpack_archive 附带参数 filename, extract_dir, format。

> **警告:** Never extract archives from untrusted sources without prior inspection. It is possible that files are created outside of the path specified in the *extract_dir* argument, e.g. members that have absolute filenames starting with "/" or filenames with two dots "..".

3.7 版更變: 接受一个*path-like object* 作为 *filename* 和 *extract_dir*。

shutil.**register_unpack_format**(*name*, *extensions*, *function*[, *extra_args*[, *description*]])
注册一个解包格式。*name* 为格式名称而 *extensions* 为对应于该格式的扩展名列表，例如 Zip 文件的扩展名为 .zip。

function 是将被用来解包归档文件的可调用对象。该可调用对象将接受归档文件的路径，加上该归档文件要被解包的目标目录。

如果提供了 *extra_args*，则其应为一个 (name, value) 元组的序列，将被作为关键字参数传递给该可调用对象。

可以提供 *description* 来描述该格式，它将被*get_unpack_formats()* 返回。

shutil.**unregister_unpack_format**(*name*)
撤销注册一个解包格式。*name* 为格式的名称。

shutil.**get_unpack_formats**()
返回所有已注册的解包格式列表。所返回序列中的每个元素为一个元组 (name, extensions, description)。

默认情况下*shutil* 提供以下格式:

- *zip*: ZIP 文件（只有在相应模块可用时才能解包压缩文件）。
- *tar*: 未压缩的 tar 文件。
- *gztar*: gzip 压缩的 tar 文件（如果*zlib* 模块可用）。

- *bztar*: bzip2 压缩的 tar 文件（如果*bz2* 模块可用）。
- *xztar*: xz 压缩的 tar 文件（如果*lzma* 模块可用）。

你可以通过使用*register_unpack_format()* 注册新的格式或为任何现有格式提供你自己的解包器。

归档程序示例

在这个示例中，我们创建了一个 gzip 压缩的 tar 归档文件，其中包含用户的 .ssh 目录下的所有文件:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

结果归档文件中包含有:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

使用 *base_dir* 的归档程序示例

在这个例子中，与*上面的例子* 类似，我们演示了如何使用*make_archive()*，但这次是使用 *base_dir*。我们现在具有如下的目录结构:

```
$ tree tmp
tmp
└── root
    └── structure
        ├── content
        │   └── please_add.txt
        └── do_not_add.txt
```

在最终的归档中，应当会包括 please_add.txt，但不应当包括 do_not_add.txt。因此我们使用以下代码:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> make_archive(
...     archive_name,
...     'tar',
...     root_dir='tmp/root',
...     base_dir='structure/content',
... )
'/Users/tarek/my_archive.tar'
```

列出结果归档中的文件我们将会得到:

```
$ python -m tarfile -l /Users/tarek/myarchive.tar
structure/content/
structure/content/please_add.txt
```

11.10.3 查询输出终端的尺寸

shutil.**get_terminal_size**(*fallback=(columns, lines)*)

获取终端窗口的尺寸。

对于两个维度中的每一个，会分别检查环境变量 COLUMNS 和 LINES。如果定义了这些变量并且其值为正整数，则将使用这些值。

如果未定义 COLUMNS 或 LINES，这是通常的情况，则连接到*sys.__stdout__* 的终端将通过发起调用*os.get_terminal_size()* 被查询。

如果由于系统不支持查询，或是由于我们未连接到某个终端而导致查询终端尺寸不成功，则会使用在 fallback 形参中给出的值。fallback 默认为 (80, 24)，这是许多终端模拟器所使用的默认尺寸。

返回的值是一个*os.terminal_size* 类型的具名元组。

另请参阅: The Single UNIX Specification, Version 2, Other Environment Variables.

3.3 版新加入.

也参考:

模块*os* 操作系统接口，包括处理比 Python 文件对象 更低级别文件的功能。

模块*io* Python 的内置 I/O 库，包括抽象类和一些具体的类，如文件 I/O 。

内置函数*open()* 使用 Python 打开文件进行读写的标准方法。

CHAPTER 12

数据持久化

本章中描述的模块支持在磁盘上以持久形式存储 Python 数据。*pickle* 和*marshal* 模块可以将许多 Python 数据类型转换为字节流，然后从字节中重新创建对象。各种与 DBM 相关的模块支持一系列基于散列的文件格式，这些格式存储字符串到其他字符串的映射。

本章中描述的模块列表是：

12.1 pickle --- Python 对象序列化

源代码： Lib/pickle.py

模块*pickle* 实现了对一个 Python 对象结构的二进制序列化和反序列化。*"pickling"* 是将 Python 对象及其所拥有的层次结构转化为一个字节流的过程，而 *"unpickling"* 是相反的操作，会将（来自一个*binary file* 或者*bytes-like object* 的）字节流转化回一个对象层次结构。pickling（和 unpickling）也被称为“序列化”，“编组”[1] 或者“平面化”。而为了避免混乱，此处采用术语“封存 (pickling)”和“解封 (unpickling)”。

> **警告：** pickle 模块 **并不安全**。你只应该对你信任的数据进行 unpickle 操作。
>
> 构建恶意的 pickle 数据来 **在解封时执行任意代码**是可能的。绝对不要对不信任来源的数据和可能被篡改过的数据进行解封。
>
> 请考虑使用*hmac* 来对数据进行签名，确保数据没有被篡改。
>
> 在你处理不信任数据时，更安全的序列化格式如*json* 可能更为适合。参见*与 json 模块的比较* 。

[1] 不要把它与*marshal* 模块混淆。

12.1.1 与其他 Python 模块间的关系

与 marshal 间的关系

Python 有一个更原始的序列化模块称为*marshal*，但一般地*pickle* 应该是序列化 Python 对象时的首选。*marshal* 存在主要是为了支持 Python 的 .pyc 文件.

pickle 模块与*marshal* 在如下几方面显著地不同：

- *pickle* 模块会跟踪已被序列化的对象，所以该对象之后再次被引用时不会再次被序列化。*marshal* 不会这么做。

  这隐含了递归对象和共享对象。递归对象指包含对自己的引用的对象。这种对象并不会被 marshal 接受，并且实际上尝试 marshal 递归对象会让你的 Python 解释器崩溃。对象共享发生在对象层级中存在多处引用同一对象时。*pickle* 只会存储这些对象一次，并确保其他的引用指向同一个主副本。共享对象将保持共享，这可能对可变对象非常重要。

- *marshal* 不能被用于序列化用户定义类及其实例。*pickle* 能够透明地存储并保存类实例，然而此时类定义必须能够从与被存储时相同的模块被引入。

- 同样用于序列化的*marshal* 格式不保证数据能移植到不同的 Python 版本中。因为它的主要任务是支持 .pyc 文件，必要时会以破坏向后兼容的方式更改这种序列化格式，为此 Python 的实现者保留了更改格式的权利。*pickle* 序列化格式可以在不同版本的 Python 中实现向后兼容，前提是选择了合适的 pickle 协议。如果你的数据要在 Python 2 与 Python 3 之间跨越传递，封存和解封的代码在 2 和 3 之间也是不同的。

与 json 模块的比较

Pickle 协议和 JSON (JavaScript Object Notation) 间有着本质的不同：

- JSON 是一个文本序列化格式（它输出 unicode 文本，尽管在大多数时候它会接着以 utf-8 编码），而 pickle 是一个二进制序列化格式；
- JSON 是我们可以直观阅读的，而 pickle 不是；
- JSON 是可互操作的，在 Python 系统之外广泛使用，而 pickle 则是 Python 专用的；
- 默认情况下，JSON 只能表示 Python 内置类型的子集，不能表示自定义的类；但 pickle 可以表示大量的 Python 数据类型（可以合理使用 Python 的对象内省功能自动地表示大多数类型，复杂情况可以通过实现*specific object APIs* 来解决）。
- 不像 pickle，对一个不信任的 JSON 进行反序列化的操作本身不会造成任意代码执行漏洞。

也参考:

json 模块: 一个允许 JSON 序列化和反序列化的标准库模块

12.1.2 数据流格式

pickle 所使用的数据格式仅可用于 Python。这样做的好处是没有外部标准给该格式强加限制，比如 JSON 或 XDR（不能表示共享指针）标准；但这也意味着非 Python 程序可能无法重新读取 pickle 封存的 Python 对象。

默认情况下，*pickle* 格式使用相对紧凑的二进制来存储。如果需要让文件更小，可以高效地压缩 由 pickle 封存的数据。

pickletools 模块包含了相应的工具用于分析*pickle* 生成的数据流。*pickletools* 源码中包含了对 pickle 协议使用的操作码的大量注释。

当前共有 6 种不同的协议可用于封存操作。使用的协议版本越高，读取所生成 pickle 对象所需的 Python 版本就要越新。

- v0 版协议是原始的“人类可读”协议，并且向后兼容早期版本的 Python。
- v1 版协议是较早的二进制格式，它也与早期版本的 Python 兼容。
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of *new-style classes*. Refer to **PEP 307** for information about improvements brought by protocol 2.
- v3 版协议是在 Python 3.0 中引入的。它显式地支持*bytes* 字节对象，不能使用 Python 2.x 解封。这是 Python 3.0-3.7 的默认协议。
- v4 版协议添加于 Python 3.4。它支持存储非常大的对象，能存储更多种类的对象，还包括一些针对数据格式的优化。它是 Python 3.8 使用的默认协议。有关第 4 版协议带来改进的信息，请参阅 **PEP 3154**。
- 第 5 版协议是在 Python 3.8 中加入的。它增加了对带外数据的支持，并可加速带内数据处理。请参阅 **PEP 574** 了解第 5 版协议所带来的改进的详情。

備🄵: 序列化是一种比持久化更底层的概念，虽然*pickle* 读取和写入的是文件对象，但它不处理持久对象的命名问题，也不处理对持久对象的并发访问（甚至更复杂）的问题。*pickle* 模块可以将复杂对象转换为字节流，也可以将字节流转换为具有相同内部结构的对象。处理这些字节流最常见的做法是将它们写入文件，但它们也可以通过网络发送或存储在数据库中。*shelve* 模块提供了一个简单的接口，用于在 DBM 类型的数据库文件上封存和解封对象。

12.1.3 模块接口

要序列化某个包含层次结构的对象，只需调用*dumps()* 函数即可。同样，要反序列化数据流，可以调用*loads()* 函数。但是，如果要对序列化和反序列化加以更多的控制，可以分别创建*Pickler* 或*Unpickler* 对象。

pickle 模块包含了以下常量：

pickle.**HIGHEST_PROTOCOL**
整数，可用的最高协议版本。此值可以作为 协议值传递给*dump()* 和*dumps()* 函数，以及*Pickler* 的构造函数。

pickle.**DEFAULT_PROTOCOL**
整数，用于 pickle 数据的默认协议版本。它可能小于*HIGHEST_PROTOCOL*。当前默认协议是 v4，它在 Python 3.4 中首次引入，与之前的版本不兼容。

3.0 版更變: 默认协议版本是 3。

3.8 版更變: 默认协议版本是 4。

pickle 模块提供了以下方法，让封存过程更加方便：

pickle.**dump**(*obj*, *file*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)
将对象 *obj* 封存以后的对象写入已打开的*file object* *file*。它等同于 `Pickler(file, protocol).dump(obj)`。

参数 *file*、*protocol*、*fix_imports* 和 *buffer_callback* 的含义与它们在*Pickler* 的构造函数中的含义相同。

3.8 版更變: 加入了 *buffer_callback* 参数。

pickle.**dumps**(*obj*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)
将 *obj* 封存以后的对象作为*bytes* 类型直接返回，而不是将其写入到文件。

参数 *protocol*、*fix_imports* 和 *buffer_callback* 的含义与它们在*Pickler* 的构造函数中的含义相同。

3.8 版更變: 加入了 *buffer_callback* 参数。

pickle.**load**(*file*, *, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*, *buffers=None*)
从已打开的*file object* 文件中读取封存后的对象，重建其中特定对象的层次结构并返回。它相当于 `Unpickler(file).load()`。

Pickle 协议版本是自动检测出来的，所以不需要参数来指定协议。封存对象以外的其他字节将被忽略。

参数 *file*、*fix_imports*、*encoding*、*errors*、*strict* 和 *buffers* 的含义与它们在*Unpickler* 的构造函数中的含义相同。

3.8 版更變: 加入了 *buffers* 参数。

pickle.**loads**(*data*, */*, *, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*, *buffers=None*)
重建并返回一个对象的封存表示形式 *data* 的对象层级结构。*data* 必须为*bytes-like object*。

Pickle 协议版本是自动检测出来的，所以不需要参数来指定协议。封存对象以外的其他字节将被忽略。

Arguments *fix_imports*, *encoding*, *errors*, *strict* and *buffers* have the same meaning as in the *Unpickler* constructor.

3.8 版更變: 加入了 *buffers* 参数。

pickle 模块定义了以下 3 个异常：

exception pickle.**PickleError**
其他 pickle 异常的基类。它是*Exception* 的一个子类。

exception pickle.**PicklingError**
当*Pickler* 遇到无法解封的对象时抛出此错误。它是*PickleError* 的子类。

参考*可以被封存/解封的对象* 来了解哪些对象可以被封存。

exception pickle.**UnpicklingError**
当解封出错时抛出此异常，例如数据损坏或对象不安全。它是*PickleError* 的子类。

注意，解封时可能还会抛出其他异常，包括（但不限于）AttributeError、EOFError、ImportError 和 IndexError。

pickle 模块包含了 3 个类，*Pickler*、*Unpickler* 和*PickleBuffer*：

class pickle.**Pickler**(*file*, *protocol=None*, *, *fix_imports=True*, *buffer_callback=None*)
它接受一个二进制文件用于写入 pickle 数据流。

可选参数 *protocol* 是一个整数，告知 pickler 使用指定的协议，可选择的协议范围从 0 到*HIGHEST_PROTOCOL*。如果没有指定，这一参数默认值为*DEFAULT_PROTOCOL*。指定一个负数就相当于指定*HIGHEST_PROTOCOL*。

参数 *file* 必须有一个 write() 方法，该 write() 方法要能接收字节作为其唯一参数。因此，它可以是一个打开的磁盘文件（用于写入二进制内容），也可以是一个*io.BytesIO* 实例，也可以是满足这一接口的其他任何自定义对象。

如果 *fix_imports* 为 True 且 *protocol* 小于 3，pickle 将尝试将 Python 3 中的新名称映射到 Python 2 中的旧模块名称，因此 Python 2 也可以读取封存的数据流。

如果 *buffer_callback* 为 None（默认情况），缓冲区视图（buffer view）将会作为 pickle 流的一部分被序列化到 *file* 中。

如果 *buffer_callback* 不为 None，那它可以用缓冲区视图调用任意次。如果某次调用返回了 False 值（例如 None），则给定的缓冲区是*带外的*，否则缓冲区是带内的（例如保存在了 pickle 流里面）。

如果 *buffer_callback* 不是 None 且 *protocol* 是 None 或小于 5，就会出错。

3.8 版更變: 加入了 *buffer_callback* 参数。

dump(*obj*)
将 *obj* 封存后的内容写入已打开的文件对象，该文件对象已经在构造函数中指定。

persistent_id(*obj*)
默认无动作，子类继承重载时使用。

如果*persistent_id()* 返回 None，*obj* 会被照常 pickle。如果返回其他值，*Pickler* 会将这个函数的返回值作为 *obj* 的持久化 ID（Pickler 本应得到序列化数据流并将其写入文件，若此函数有返回值，则得到此函数的返回值并写入文件）。这个持久化 ID 的解释应当定义在*Unpickler.persistent_load()* 中（该方法定义还原对象的过程，并返回得到的对象）。注意，*persistent_id()* 的返回值本身不能拥有持久化 ID。

参阅持久化外部对象 获取详情和使用示例。

dispatch_table
Pickler 对象的 dispatch 表是*copyreg.pickle()* 中用到的 *reduction* 函数的注册。dispatch 表本身是一个 class 到其 reduction 函数的映射键值对。一个 reduction 函数只接受一个参数，就是其关联的 class，函数行为应当遵守*__reduce__()* 接口规范。

Pickler 对象默认并没有*dispatch_table* 属性，该对象默认使用*copyreg* 模块中定义的全局 dispatch 表。如果要为特定 Pickler 对象自定义序列化过程，可以将*dispatch_table* 属性设置为类字典对象（dict-like object）。另外，如果*Pickler* 的子类设置了*dispatch_table* 属性，则该子类的实例会使用这个表作为默认的 dispatch 表。

参阅*Dispatch 表* 获取使用示例。

3.3 版新加入.

reducer_override(*obj*)
可以在*Pickler* 的子类中定义的特殊 reducer。此方法的优先级高于*dispatch_table* 中的任何 reducer。它应该与*__reduce__()* 方法遵循相同的接口，它也可以返回 NotImplemented，这将使用*dispatch_table* 里注册的 reducer 来封存 obj。

参阅类型，函数和其他对象的自定义归约 获取详细的示例。

3.8 版新加入.

fast
已弃用。设为 True 则启用快速模式。快速模式禁用了“备忘录” (memo) 的使用，即不生成多余的 PUT 操作码来加快封存过程。不应将其与自指 (self-referential) 对象一起使用，否则将导致*Pickler* 无限递归。

如果需要进一步提高 pickle 的压缩率，请使用*pickletools.optimize()*。

class pickle.**Unpickler**(*file*, *, *fix_imports=True*, *encoding="ASCII"*, *errors="strict"*, *buffers=None*)
它接受一个二进制文件用于读取 pickle 数据流。

Pickle 协议版本是自动检测出来的，所以不需要参数来指定协议。

参数 *file* 必须有三个方法，read() 方法接受一个整数参数，readinto() 方法接受一个缓冲区作为参数，readline() 方法不需要参数，这与*io.BufferedIOBase* 里定义的接口是相同的。因此 *file* 可以是一个磁盘上用于二进制读取的文件，也可以是一个*io.BytesIO* 实例，也可以是满足这一接口的其他任何自定义对象。

可选的参数是 *fix_imports*, *encoding* 和 *errors*，用于控制由 Python 2 生成的 pickle 流的兼容性。如果 *fix_imports* 为 True，则 pickle 将尝试将旧的 Python 2 名称映射到 Python 3 中对应的新名称。*encoding* 和 *errors* 参数告诉 pickle 如何解码 Python 2 存储的 8 位字符串实例；这两个参数默认分别为'ASCII' 和'strict'。*encoding* 参数可置为'bytes' 来将这些 8 位字符串实例读取为字节对象。读取 NumPy array 和 Python 2 存储的*datetime*、*date* 和*time* 实例时，请使用 encoding='latin1'。

如果 *buffers* 为 None（默认值），则反序列化所需的所有数据都必须包含在 pickle 流中。这意味着在实例化*Pickler* 时（或调用*dump()* 或*dumps()* 时），参数 *buffer_callback* 为 None。

如果 *buffers* 不为 None，则每次 pickle 流引用带外 缓冲区视图时，消耗的对象都应该是可迭代的启用缓冲区的对象。这样的缓冲区应该按顺序地提供给 Pickler 对象的 *buffer_callback* 方法。

3.8 版更變: 加入了 *buffers* 参数。

load()
从构造函数中指定的文件对象里读取封存好的对象，重建其中特定对象的层次结构并返回。封存对象以外的其他字节将被忽略。

persistent_load(*pid*)
默认抛出*UnpicklingError* 异常。

如果定义了此方法，*persistent_load()* 应当返回持久化 ID *pid* 所指定的对象。如果遇到无效的持久化 ID，则应当引发*UnpicklingError*。

参阅持久化外部对象 获取详情和使用示例。

find_class(*module*, *name*)
如有必要，导入 *module* 模块并返回其中名叫 *name* 的对象，其中 *module* 和 *name* 参数都是*str* 对象。注意，不要被这个函数的名字迷惑，*find_class()* 同样可以用来导入函数。

子类可以重载此方法，来控制加载对象的类型和加载对象的方式，从而尽可能降低安全风险。参阅限制全局变量 获取更详细的信息。

引发一个审计事件 `pickle.find_class` 附带参数 `module`、`name`。

class pickle.**PickleBuffer**(*buffer*)
缓冲区的包装器 (wrapper)，缓冲区中包含着可封存的数据。*buffer* 必须是一个 buffer-providing 对象，比如*bytes-like object* 或多维数组。

PickleBuffer 本身就可以生成缓冲区对象，因此可以将其传递给需要缓冲区生成器的其他 API，比如*memoryview*。

PickleBuffer 对象只能用 pickle 版本 5 及以上协议进行序列化。它们符合带外序列化 的条件。

3.8 版新加入.

raw()
返回该缓冲区底层内存区域的*memoryview*。返回的对象是一维的、C 连续布局的 memoryview，格式为 `B` (无符号字节)。如果缓冲区既不是 C 连续布局也不是 Fortran 连续布局的，则抛出*BufferError* 异常。

release()
释放由 PickleBuffer 占用的底层缓冲区。

12.1.4 可以被封存/解封的对象

下列类型可以被封存：

- `None`, `True`, and `False`;
- integers, floating-point numbers, complex numbers;
- strings, bytes, bytearrays;
- tuples, lists, sets, and dictionaries containing only picklable objects;
- functions (built-in and user-defined) defined at the top level of a module (using `def`, not `lambda`);
- classes defined at the top level of a module;
- 某些类实例，这些类的*__dict__* 属性值或*__getstate__()* 函数的返回值可以被封存（详情参阅封存类实例 这一段）。

尝试封存不能被封存的对象会抛出*PicklingError*异常，异常发生时，可能有部分字节已经被写入指定文件中。尝试封存递归层级很深的对象时，可能会超出最大递归层级限制，此时会抛出*RecursionError*异常，可以通过*sys.setrecursionlimit()*调整递归层级，不过请谨慎使用这个函数，因为可能会导致解释器崩溃。

Note that functions (built-in and user-defined) are pickled by fully qualified name, not by value.[2] This means that only the function name is pickled, along with the name of the module the function is defined in. Neither the function's code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised.[3]

Similarly, classes are pickled by fully qualified name, so the same restrictions in the unpickling environment apply. Note that none of the class's code or data is pickled, so in the following example the class attribute `attr` is not restored in the unpickling environment:

```
class Foo:
    attr = 'A class attribute'

picklestring = pickle.dumps(Foo)
```

These restrictions are why picklable functions and classes must be defined at the top level of a module.

类似的，在封存类的实例时，其类体和类数据不会跟着实例一起被封存，只有实例数据会被封存。这样设计是有目的的，在将来修复类中的错误、给类增加方法之后，仍然可以载入原来版本类实例的封存数据来还原该实例。如果你准备长期使用一个对象，可能会同时存在较多版本的类体，可以为对象添加版本号，这样就可以通过类的*__setstate__()*方法将老版本转换成新版本。

12.1.5 封存类实例

在本节中，我们描述了可用于定义、自定义和控制如何封存和解封类实例的通用流程。

通常，使一个实例可被封存不需要附加任何代码。Pickle 默认会通过 Python 的内省机制获得实例的类及属性。而当实例解封时，它的 `__init__()` 方法通常 *不会* 被调用。其默认动作是：先创建一个未初始化的实例，然后还原其属性，下面的代码展示了这种行为的实现机制：

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def restore(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

类可以改变默认行为，只需定义以下一种或几种特殊方法：

object.**__getnewargs_ex__**()

对于使用第 2 版或更高版协议的 pickle，实现了*__getnewargs_ex__()*方法的类可以控制在解封时传给 `__new__()` 方法的参数。本方法必须返回一对 `(args, kwargs)` 用于构建对象，其中 *args* 是表示位置参数的 tuple，而 *kwargs* 是表示命名参数的 dict。它们会在解封时传递给 `__new__()` 方法。

如果类的 `__new__()` 方法只接受关键字参数，则应当实现这个方法。否则，为了兼容性，更推荐实现*__getnewargs__()*方法。

3.6 版更變: *__getnewargs_ex__()* 现在可用于第 2 和第 3 版协议。

[2] 这就是为什么 `lambda` 函数不可以被封存：所有的匿名函数都有同一个名字：`<lambda>`。
[3] 抛出的异常有可能是*ImportError*或*AttributeError*，也可能是其他异常。

object.**__getnewargs__**()
这个方法与上一个*__getnewargs_ex__()* 方法类似，但仅支持位置参数。它要求返回一个 tuple 类型的 args，用于解封时传递给 __new__() 方法。

如果定义了*__getnewargs_ex__()*，那么*__getnewargs__()* 就不会被调用。

3.6 版更變: 在 Python 3.6 前，第 2、3 版协议会调用*__getnewargs__()*，更高版本协议会调用*__getnewargs_ex__()*。

object.**__getstate__**()
类还可以进一步控制其实例的封存过程。如果类定义了*__getstate__()*，它就会被调用，其返回的对象是被当做实例内容来封存的，否则封存的是实例的 __dict__。如果*__getstate__()* 未定义，实例的*__dict__* 会被照常封存。

object.**__setstate__**(*state*)
当解封时，如果类定义了*__setstate__()*，就会在已解封状态下调用它。此时不要求实例的 state 对象必须是 dict。没有定义此方法的话，先前封存的 state 对象必须是 dict，且该 dict 内容会在解封时赋给新实例的 __dict__。

備𝔽: 如果*__getstate__()* 返回 False，那么在解封时就不会调用*__setstate__()* 方法。

参考*处理有状态的对象* 一段获取如何使用*__getstate__()* 和*__setstate__()* 方法的更多信息。

備𝔽: 在解封时，实例的某些方法例如 __getattr__(), __getattribute__() 或 __setattr__() 可能会被调用。由于这些方法可能要求某些内部不变量为真值，因此该类型应当实现 __new__() 以建立这样的不变量，因为当解封一个实例时 __init__() 并不会被调用。

可以看出，其实 pickle 并不直接调用上面的几个函数。事实上，这几个函数是复制协议的一部分，它们实现了*__reduce__()* 这一特殊接口。复制协议提供了统一的接口，用于在封存或复制对象的过程中取得所需数据。[4]

尽管这个协议功能很强，但是直接在类中实现*__reduce__()* 接口容易产生错误。因此，设计类时应当尽可能的使用高级接口（比如*__getnewargs_ex__()*、*__getstate__()* 和*__setstate__()*）。后面仍然可以看到直接实现*__reduce__()* 接口的状况，可能别无他法，可能为了获得更好的性能，或者两者皆有之。

object.**__reduce__**()
该接口当前定义如下。*__reduce__()* 方法不带任何参数，并且应返回字符串或最好返回一个元组（返回的对象通常称为“reduce 值”）。

如果返回字符串，该字符串会被当做一个全局变量的名称。它应该是对象相对于其模块的本地名称，pickle 模块会搜索模块命名空间来确定对象所属的模块。这种行为常在单例模式使用。

如果返回的是元组，则应当包含 2 到 6 个元素，可选元素可以省略或设置为 None。每个元素代表的意义如下：

- 一个可调用对象，该对象会在创建对象的最初版本时调用。
- 可调用对象的参数，是一个元组。如果可调用对象不接受参数，必须提供一个空元组。
- 可选元素，用于表示对象的状态，将被传给前述的*__setstate__()* 方法。如果对象没有此方法，则这个元素必须是字典类型，并会被添加至*__dict__* 属性中。
- 可选元素，一个返回连续项的迭代器（而不是序列）。这些项会被 obj.append(item) 逐个加入对象，或被 obj.extend(list_of_items) 批量加入对象。这个元素主要用于 list 的子类，

[4] *copy* 模块使用这一协议实现浅层 (shallow) 和深层 (deep) 复制操作。

也可以用于那些正确实现了 `append()` 和 `extend()` 方法的类。(具体是使用 `append()` 还是 `extend()` 取决于 pickle 协议版本以及待插入元素的项数，所以这两个方法必须同时被类支持。)

- 可选元素，一个返回连续键值对的迭代器(而不是序列)。这些键值对将会以 `obj[key] = value` 的方式存储于对象中。该元素主要用于 dict 子类，也可以用于那些实现了 `__setitem__()` 的类。
- 可选元素，一个带有 `(obj, state)` 签名的可调用对象。该可调用对象允许用户以编程方式控制特定对象的状态更新行为，而不是使用 `obj` 的静态 *`__setstate__()`* 方法。如果此处不是 `None`，则此可调用对象的优先级高于 `obj` 的 *`__setstate__()`*。

  3.8 版新加入: 新增了元组的第 6 项，可选元素 `(obj, state)`。

`object.`**`__reduce_ex__`**(*protocol*)

作为替代选项，也可以实现 *`__reduce_ex__()`* 方法。此方法的唯一不同之处在于它应接受一个整型参数用于指定协议版本。如果定义了这个函数，则会覆盖 *`__reduce__()`* 的行为。此外，*`__reduce__()`* 方法会自动成为扩展版方法的同义词。这个函数主要用于为以前的 Python 版本提供向后兼容的 reduce 值。

持久化外部对象

为了获取对象持久化的利益，*`pickle`* 模块支持引用已封存数据流之外的对象。这样的对象是通过一个持久化 ID 来引用的，它应当是一个由字母数字类字符组成的字符串 (对于第 0 版协议)[5] 或是一个任意对象 (用于任意新版协议)。

`pickle` 模块不提供对持久化 ID 的解析工作，它将解析工作分配给用户定义的方法，分别是 pickler 中的*`persistent_id()`* 方法和 unpickler 中的*`persistent_load()`* 方法。

要通过持久化 ID 将外部对象封存，必须在 pickler 中实现*`persistent_id()`* 方法，该方法接受需要被封存的对象作为参数，返回一个 `None` 或返回该对象的持久化 ID。如果返回 `None`，该对象会被按照默认方式封存为数据流。如果返回字符串形式的持久化 ID，则会封存这个字符串并加上一个标记，这样 unpickler 才能将其识别为持久化 ID。

要解封外部对象，Unpickler 必须实现*`persistent_load()`* 方法，接受一个持久化 ID 对象作为参数并返回一个引用的对象。

下面是一个全面的例子，展示了如何使用持久化 ID 来封存外部对象。

```
# Simple example presenting how persistent ID can be used to pickle
# external objects by reference.

import pickle
import sqlite3
from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
```

(下页继续)

[5] The limitation on alphanumeric characters is due to the fact that persistent IDs in protocol 0 are delimited by the newline character. Therefore if any kind of newline characters occurs in persistent IDs, the resulting pickled data will become unreadable.

(繼續上一頁)

```
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and return it.
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")


def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
        )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
```

(下页继续)

(繼續上一頁)

```
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # Load the records from the pickle data stream.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
    pprint.pprint(memos)


if __name__ == '__main__':
    main()
```

Dispatch 表

如果想对某些类进行自定义封存，而又不想在类中增加用于封存的代码，就可以创建带有特殊 dispatch 表的 pickler。

在*copyreg* 模块的 `copyreg.dispatch_table` 中定义了全局 dispatch 表。因此，可以使用 `copyreg.dispatch_table` 修改后的副本作为自有 dispatch 表。

例如

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

创建了一个带有自有 dispatch 表的*pickle.Pickler* 实例，它可以对 `SomeClass` 类进行特殊处理。另外，下列代码

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

does the same but all instances of `MyPickler` will by default share the private dispatch table. On the other hand, the code

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

modifies the global dispatch table shared by all users of the *copyreg* module.

处理有状态的对象

下面的示例展示了如何修改类在封存时的行为。其中 TextReader 类打开了一个文本文件，每次调用其 readline() 方法则返回行号和该行的字符。在封存这个 TextReader 的实例时，除了文件对象，其他属性都会被保存。当解封实例时，需要重新打开文件，然后从上次的位置开始继续读取。实现这些功能需要实现 *__setstate__()* 和 *__getstate__()* 方法。

```
class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

    def readline(self):
        self.lineno += 1
        line = self.file.readline()
        if not line:
            return None
        if line.endswith('\n'):
            line = line[:-1]
        return "%i: %s" % (self.lineno, line)

    def __getstate__(self):
        # Copy the object's state from self.__dict__ which contains
        # all our instance attributes. Always use the dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file
```

使用方法如下所示：

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

12.1.6 类型，函数和其他对象的自定义归约

3.8 版新加入.

有时，*dispatch_table* 可能不够灵活。特别是当我们想要基于对象类型以外的其他规则来对封存进行定制，或是当我们想要对函数和类的封存进行定制的时候。

对于那些情况，可能要基于*Pickler* 类进行子类化并实现*reducer_override()* 方法。此方法可返回任意的归约元组 (参见*__reduce__()*)。它也可以选择返回 NotImplemented 来回退到传统行为。

如果同时定义了*dispatch_table* 和*reducer_override()*，则*reducer_override()* 方法具有优先权。

備𝔽: 出于性能理由，可能不会为以下对象调用*reducer_override()*: None, True, False, 以及*int*, *float*, *bytes*, *str*, *dict*, *set*, *frozenset*, *list* 和*tuple* 的具体实例。

以下是一个简单的例子，其中我们允许封存并重新构建一个给定的类:

```
import io
import pickle

class MyClass:
    my_attribute = 1

class MyPickler(pickle.Pickler):
    def reducer_override(self, obj):
        """Custom reducer for MyClass."""
        if getattr(obj, "__name__", None) == "MyClass":
            return type, (obj.__name__, obj.__bases__,
                          {'my_attribute': obj.my_attribute})
        else:
            # For any other object, fallback to usual reduction
            return NotImplemented

f = io.BytesIO()
p = MyPickler(f)
p.dump(MyClass)

del MyClass

unpickled_class = pickle.loads(f.getvalue())

assert isinstance(unpickled_class, type)
assert unpickled_class.__name__ == "MyClass"
assert unpickled_class.my_attribute == 1
```

12.1.7 外部缓冲区

3.8 版新加入.

在某些场景中，*pickle* 模块会被用来传输海量的数据。因此，最小化内存复制次数以保证性能和节省资源是很重要的。但是*pickle* 模块的正常运作会将图类对象结构转换为字节序列流，因此在本质上就要从封存流中来回复制数据。

如果 *provider* (待传输对象类型的实现) 和 *consumer* (通信系统的实现) 都支持 pickle 第 5 版或更高版本所提供的外部传输功能，则此约束可以被撤销。

提供方 API

大的待封存数据对象必须实现协议 5 及以上版本专属的*__reduce_ex__()* 方法，该方法将为任意大的数据返回一个*PickleBuffer* 实例（而不是*bytes* 对象等）。

PickleBuffer 对象会 表明底层缓冲区可被用于外部数据传输。那些对象仍将保持与*pickle* 模块的正常用法兼容。但是，使用方也可以选择告知*pickle* 它们将自行处理那些缓冲区。

使用方 API

当序列化一个对象图时，通信系统可以启用对所生成*PickleBuffer* 对象的定制处理。

发送端需要传递 *buffer_callback* 参数到*Pickler* (或是到*dump()* 或*dumps()* 函数)，该回调函数将在封存对象图时附带每个所生成的*PickleBuffer* 被调用。由 *buffer_callback* 所累积的缓冲区的数据将不会被拷贝到 pickle 流，而是仅插入一个简单的标记。

接收端需要传递 *buffers* 参数到*Unpickler* (或是到*load()* 或*loads()* 函数)，其值是一个由缓冲区组成的可迭代对象，它会被传递给 *buffer_callback*。该可迭代对象应当按其被传递给 *buffer_callback* 时的顺序产生缓冲区。这些缓冲区将提供对象重构造器所期望的数据，对这些数据的封存产生了原本的*PickleBuffer* 对象。

在发送端和接受端之间，通信系统可以自由地实现它自己用于外部缓冲区的传输机制。潜在的优化包括使用共享内存或基于特定数据类型的压缩等。

示例

下面是一个小例子，在其中我们实现了一个*bytearray* 的子类，能够用于外部缓冲区封存:

```
class ZeroCopyByteArray(bytearray):

    def __reduce_ex__(self, protocol):
        if protocol >= 5:
            return type(self)._reconstruct, (PickleBuffer(self),), None
        else:
            # PickleBuffer is forbidden with pickle protocols <= 4.
            return type(self)._reconstruct, (bytearray(self),)

    @classmethod
    def _reconstruct(cls, obj):
        with memoryview(obj) as m:
            # Get a handle over the original buffer object
            obj = m.obj
            if type(obj) is cls:
                # Original buffer object is a ZeroCopyByteArray, return it
```

(下页继续)

(繼續上一頁)

```
                # as-is.
                return obj
            else:
                return cls(obj)
```

重构造器 (_reconstruct 类方法) 会在缓冲区的提供对象具有正确类型时返回该对象。在此小示例中这是模拟零拷贝行为的便捷方式。

在使用方，我们可以按通常方式封存那些对象，它们在反序列化时将提供原始对象的一个副本:

```
b = ZeroCopyByteArray(b"abc")
data = pickle.dumps(b, protocol=5)
new_b = pickle.loads(data)
print(b == new_b)  # True
print(b is new_b)  # False: a copy was made
```

但是如果我们传入 *buffer_callback* 然后在反序列化时给回累积的缓冲区，我们就能够取回原始对象:

```
b = ZeroCopyByteArray(b"abc")
buffers = []
data = pickle.dumps(b, protocol=5, buffer_callback=buffers.append)
new_b = pickle.loads(data, buffers=buffers)
print(b == new_b)  # True
print(b is new_b)  # True: no copy was made
```

这个例子受限于*bytearray* 会自行分配内存这一事实：你无法基于另一个对象的内存创建*bytearray* 的实例。但是，第三方数据类型例如 NumPy 数组则没有这种限制，允许在单独进程或系统间传输时使用零拷贝的封存（或是尽可能少地拷贝）。

也参考:

PEP 574 -- 带有外部数据缓冲区的 pickle 协议 5

12.1.8 限制全局变量

默认情况下，解封将会导入在 pickle 数据中找到的任何类或函数。对于许多应用来说，此行为是不可接受的，因为它会允许解封器导入并发起调用任意代码。只须考虑当这个手工构建的 pickle 数据流被加载时会做什么:

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

在这个例子里，解封器导入*os.system()* 函数然后应用字符串参数"echo hello world"。虽然这个例子不具攻击性，但是不难想象别人能够通过此方式对你的系统造成损害。

出于这样的理由，你可能会希望通过定制*Unpickler.find_class()* 来控制要解封的对象。与其名称所提示的不同，*Unpickler.find_class()* 会在执行对任何全局对象（例如一个类或一个函数）的请求时被调用。因此可以完全禁止全局对象或是将它们限制在一个安全的子集中。

下面的例子是一个解封器，它只允许某一些安全的来自*builtins* 模块的类被加载:

```
import builtins
import io
import pickle
```

(下页继续)

(繼續上一頁)

```
safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

A sample usage of our unpickler working as intended:

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

正如我们这个例子所显示的，对于允许解封的对象你必须要保持谨慎。因此如果要保证安全，你可以考虑其他选择例如*xmlrpc.client* 中的编组 API 或是第三方解决方案。

12.1.9 性能

较新版本的 pickle 协议（第 2 版或更高）具有针对某些常见特性和内置类型的高效二进制编码格式。此外，*pickle* 模块还拥有一个以 C 编写的透明优化器。

12.1.10 示例

对于最简单的代码，请使用*dump()* 和*load()* 函数。

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3+4j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

以下示例读取之前封存的数据。

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

也参考:

模块*copyreg* 为扩展类型提供 pickle 接口所需的构造函数。

模块*pickletools* 用于处理和分析已封存数据的工具。

模块*shelve* 带索引的数据库，用于存放对象，使用了*pickle* 模块。

模块*copy* 浅层 (shallow) 和深层 (deep) 复制对象操作

模块*marshal* 高效地序列化内置类型的数据。

F解

12.2 copyreg --- 注册配合 pickle 模块使用的函数

源代码: Lib/copyreg.py

---

copyreg 模块提供了可在封存特定对象时使用的一种定义函数方式。*pickle* 和*copy* 模块会在封存/拷贝特定对象时使用这些函数。此模块提供了非类对象构造器的相关配置信息。这样的构造器可以是工厂函数或类实例。

copyreg.**constructor**(*object*)
将 *object* 声明为一个有效的构造器。如果 *object* 是不可调用的（因而不是一个有效的构造器）则会引发*TypeError*。

copyreg.**pickle**(*type*, *function*, *constructor=None*)
声明该 *function* 应当被用作 *type* 类型对象的 “归约函数”。*function* 应当返回字符串或包含两到三个元素的元组。

The optional *constructor* parameter, if provided, is a callable object which can be used to reconstruct the object when called with the tuple of arguments returned by *function* at pickling time. A `TypeError` is raised if the *constructor* is not callable.

请查看`pickle`模块了解 *function* 和 *constructor* 所要求的接口的详情。请注意一个 pickler 对象或`pickle.Pickler`的子类的`dispatch_table`属性也可以被用来声明归约函数。

12.2.1 示例

以下示例将会显示如何注册一个封存函数，以及如何来使用它：

```
>>> import copyreg, copy, pickle
>>> class C:
...     def __init__(self, a):
...         self.a = a
...
>>> def pickle_c(c):
...     print("pickling a C instance...")
...     return C, (c.a,)
...
>>> copyreg.pickle(C, pickle_c)
>>> c = C(1)
>>> d = copy.copy(c)
pickling a C instance...
>>> p = pickle.dumps(c)
pickling a C instance...
```

12.3 `shelve` --- Python 对象持久化

源代码: Lib/shelve.py

"Shelf" 是一种持久化的类似字典的对象。与"dbm" 数据库的区别在于 Shelf 中的值（不是键！）实际上可以为任意 Python 对象 --- 即`pickle`模块能够处理的任何东西。这包括大部分类实例、递归数据类型，以及包含大量共享子对象的对象。键则为普通的字符串。

`shelve.`**`open`**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

打开一个持久化字典。filename 指定下层数据库的基准文件名。作为附带效果，会为 filename 添加一个扩展名并且可能创建更多的文件。默认情况下，下层数据库会以读写模式打开。可选的 *flag* 形参具有与`dbm.open()` *flag* 形参相同的含义。

默认会使用第 3 版 pickle 协议来序列化值。pickle 协议版本可通过 *protocol* 形参来指定。

由于 Python 语义的限制，Shelf 对象无法确定一个可变的持久化字典条目在何时被修改。默认情况下只有在被修改对象再赋值给 shelf 时才会写入该对象 (参见示例)。如果可选的 *writeback* 形参设为 `True`，则所有被访问的条目都将在内存中被缓存，并会在`sync()`和`close()`时被写入；这可以使得对持久化字典中可变条目的修改更方便，但是如果访问的条目很多，这会消耗大量内存作为缓存，并会使得关闭操作变得非常缓慢，因为所有被访问的条目都需要写回到字典（无法确定被访问的条目中哪个是可变的，也无法确定哪个被实际修改了）。

備🄵: 请不要依赖于 Shelf 的自动关闭功能；当你不再需要时应当总是显式地调用`close()`，或者使用`shelve.open()`作为上下文管理器:

```
with shelve.open('spam') as db:
    db['eggs'] = 'eggs'
```

警告： 由于*shelve*模块需要*pickle*的支持，因此从不可靠的来源载入 shelf 是不安全的。与 pickle 一样，载入 Shelf 时可以执行任意代码。

Shelf 对象支持字典所支持的大多数方法和运算（除了拷贝、构造器以及 | 和 |= 运算符）。这样就能方便地将基于字典的脚本转换为要求持久化存储的脚本。

额外支持的两个方法：

Shelf.**sync**()
: 如果 Shelf 打开时将 *writeback* 设为*True*则写回缓存中的所有条目。如果可行还会清空缓存并将持久化字典同步到磁盘。此方法会在使用*close()*关闭 Shelf 时自动被调用。

Shelf.**close**()
: 同步并关闭持久化 *dict* 对象。对已关闭 Shelf 的操作将失败并引发*ValueError*。

也参考：

持久化字典方案，使用了广泛支持的存储格式并具有原生字典的速度。

12.3.1 限制

- 可选择使用哪种数据库包 (例如*dbm.ndbm*或*dbm.gnu*) 取决于支持哪种接口。因此使用*dbm*直接打开数据库是不安全的。如果使用了*dbm*，数据库同样会（不幸地）受限于它 --- 这意味着存储在数据库中的（封存形式的）对象尺寸应当较小，并且在少数情况下键冲突有可能导致数据库拒绝更新。
- *shelve*模块不支持对 Shelf 对象的 *并发读/写访问*。（多个同时读取访问则是安全的。）当一个程序打开一个 shelve 对象来写入时，不应再有其他程序同时打开它来读取或写入。Unix 文件锁定可被用来解决此问题，但这在不同 Unix 版本上会存在差异，并且需要有关所用数据库实现的细节知识。

class shelve.**Shelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)
: *collections.abc.MutableMapping*的一个子类，它会将封存的值保存在 *dict* 对象中。

  默认会使用第 3 版 pickle 协议来序列化值。pickle 协议版本可通过 *protocol* 形参来指定。请参阅*pickle*文档来查看 pickle 协议的相关讨论。

  如果 *writeback* 形参为 `True`，对象将为所有访问过的条目保留缓存并在同步和关闭时将它们写回到 *dict*。这允许对可变的条目执行自然操作，但是会消耗更多内存并让同步和关闭花费更长时间。

  keyencoding 形参是在下层字典被使用之前用于编码键的编码格式。

  *Shelf*对象还可以被用作上下文管理器，在这种情况下它将在 `with` 语句块结束时自动被关闭。

  3.2 版更變: 添加了 *keyencoding* 形参；之前，键总是使用 UTF-8 编码。

  3.4 版更變: 添加了上下文管理器支持

class shelve.**BsdDbShelf**(*dict*, *protocol=None*, *writeback=False*, *keyencoding='utf-8'*)
: *Shelf*的一个子类，将 `first()`, `next()`, `previous()`, `last()` 和 `set_location()` 对外公开，在来自 pybsddb 的第三方 `bsddb` 模块中可用，但在其他数据库模块中不可用。传给构造器的 *dict* 对象必须支持这些方法。这通常是通过调用 `bsddb.hashopen()`, `bsddb.btopen()` 或 `bsddb.rnopen()` 之一来完成的。可选的 *protocol*, *writeback* 和 *keyencoding* 形参具有与*Shelf*类相同的含义。

class shelve.**DbfilenameShelf**(*filename*, *flag='c'*, *protocol=None*, *writeback=False*)

Shelf 的一个子类，它接受一个 *filename* 而非字典类对象。下层文件将使用*dbm.open()* 来打开。默认情况下，文件将以读写模式打开。可选的 *flag* 形参具有与*open()* 函数相同的含义。可选的 *protocol* 和 *writeback* 形参具有与*Shelf* 类相同的含义。

12.3.2 示例

对接口的总结如下 (`key` 为字符串，`data` 为任意对象):

```
import shelve

d = shelve.open(filename)  # open -- file may get suffix added by low-level
                           # library

d[key] = data              # store data at key (overwrites old data if
                           # using an existing key)
data = d[key]              # retrieve a COPY of data at key (raise KeyError
                           # if no such key)
del d[key]                 # delete data stored at key (raises KeyError
                           # if no such key)

flag = key in d            # true if the key exists
klist = list(d.keys())     # a list of all existing keys (slow!)

# as d was opened WITHOUT writeback=True, beware:
d['xx'] = [0, 1, 2]        # this works as expected, but...
d['xx'].append(3)          # *this doesn't!* -- d['xx'] is STILL [0, 1, 2]!

# having opened d without writeback=True, you need to code carefully:
temp = d['xx']             # extracts the copy
temp.append(5)             # mutates the copy
d['xx'] = temp             # stores the copy right back, to persist it

# or, d=shelve.open(filename,writeback=True) would let you just code
# d['xx'].append(5) and have it work as expected, BUT it would also
# consume more memory and make the d.close() operation slower.

d.close()                  # close it
```

也参考:

模块*dbm* dbm 风格数据库的泛型接口。

模块*pickle* *shelve* 所使用的对象序列化。

12.4 marshal --- 内部 Python 对象序列化

此模块包含一此能以二进制格式来读写 Python 值的函数。这种格式是 Python 专属的，但是独立于特定的机器架构（即你可以在一台 PC 上写入某个 Python 值，将文件传到一台 Sun 上并在那里读取它）。这种格式的细节有意不带文档说明；它可能在不同 Python 版本中发生改变（但这种情况极少发生）。[1]

[1] 此模块的名称来源于 Modula-3 (及其他语言) 的设计者所使用的术语，他们使用术语”marshal” 来表示以自包含的形式传输数据。严格地说，将数据从内部形式转换为外部形式 (例如用于 RPC 缓冲区) 称为”marshal” 而其逆过程则称为”unmarshal”。

这不是一个通用的“持久化”模块。对于通用的持久化以及通过 RPC 调用传递 Python 对象，请参阅*pickle* 和*shelve* 等模块。*marshal* 模块主要是为了支持读写 `.pyc` 文件形式“伪编译”代码的 Python 模块。因此，Python 维护者保留在必要时以不向下兼容的方式修改 marshal 格式的权利。如果你要序列化和反序列化 Python 对象，请改用*pickle* 模块 -- 其执行效率相当，版本独立性有保证，并且 pickle 还支持比 marshal 更多样的对象类型。

警告： *marshal* 模块对于错误或恶意构建的数据来说是不安全的。永远不要 unmarshal 来自不受信任的或未经验证的来源的数据。

不是所有 Python 对象类型都受支持；一般来说，此模块只能写入和读取不依赖于特定 Python 调用的对象。下列类型是受支持的：布尔值、整数、浮点数、复数、字符串、字节串、字节数组、元组、列表、集合、冻结集合、字典和代码对象，需要了解的一点是元组、列表、集合、冻结集合和字典只在其所包含的值也是这些值时才受支持。单例对象*None*, *Ellipsis* and *StopIteration* 也可以被 marshal 和 unmarshal。对于 *version* 低于 3 的格式，递归列表、集合和字典无法被写入（见下文）。

有些函数可以读/写文件，还有些函数可以操作字节类对象。

这个模块定义了以下函数：

`marshal.`**`dump`**(*value*, *file*[, *version*])
向打开的文件写入值。值必须为受支持的类型。文件必须为可写的*binary file*。

如果值具有（或所包含的对象具有）不受支持的类型，则会引发*ValueError* --- 但是将向文件写入垃圾数据。对象也将不能正确地通过*load()* 重新读取。

version 参数指明 `dump` 应当使用的数据格式（见下文）。

引发一个审计事件 `marshal.dumps`，附带参数 `value`, `version`。

`marshal.`**`load`**(*file*)
从打开的文件读取一个值并返回。如果读不到有效的值（例如由于数据为不同 Python 版本的不兼容 marshal 格式），则会引发*EOFError*, *ValueError* 或*TypeError*。文件必须为可读的*binary file*。

引发一个审计事件 `marshal.load`，没有附带参数。

備F: 如果通过*dump()* marshal 了一个包含不受支持类型的对象，*load()* 将为不可 marshal 的类型替换 `None`。

3.9.7 版更變: 使用此调用为每个代码对象引发一个 `code.__new__` 审计事件。现在它会为整个载入操作引发单个 `marshal.load` 事件。

`marshal.`**`dumps`**(*value*[, *version*])
返回将通过 `dump(value, file)` 被写入一个文件的字节串对象。值必须属于受支持的类型。如果值属于（或包含的对象属于）不受支持的类型则会引发*ValueError*。

version 参数指明 `dumps` 应当使用的数据类型（见下文）。

引发一个审计事件 `marshal.dumps`，附带参数 `value`, `version`。

`marshal.`**`loads`**(*bytes*)
将*bytes-like object* 转换为一个值。如果找不到有效的值，则会引发*EOFError*, *ValueError* 或*TypeError*。输入的额外字节串会被忽略。

引发一个审计事件 `marshal.loads`，附带参数 `bytes`。

3.9.7 版更變: 使用此调用为每个代码对象引发一个 `code.__new__` 审计事件。现在它会为整个载入操作引发单个 `marshal.loads` 事件。

此外，还定义了以下常量：

marshal.**version**
指明模块所使用的格式。第 0 版为历史格式，第 1 版为共享固化的字符串，第 2 版对浮点数使用二进制格式。第 3 版添加了对于对象实例化和递归的支持。目前使用的为第 4 版。

F解

12.5 dbm --- Unix "数据库" 接口

源代码: Lib/dbm/__init__.py

---

dbm 是一种泛用接口，针对各种 DBM 数据库 --- 包括*dbm.gnu* 或*dbm.ndbm*。如果未安装这些模块中的任何一种，则将使用*dbm.dumb* 模块中慢速但简单的实现。还有一个适用于 Oracle Berkeley DB 的 第三方接口。

exception dbm.**error**
一个元组，其中包含每个受支持的模块可引发的异常，另外还有一个名为*dbm.error* 的特殊异常作为第一项 --- 后者最在引发*dbm.error* 时被使用。

dbm.**whichdb**(*filename*)
此函数会猜测各种简单数据库模块中的哪一个是可用的 --- *dbm.gnu*, *dbm.ndbm* 还是*dbm.dumb* --- 应该被用来打开给定的文件。

返回下列值中的一个：如果文件由于不可读或不存在而无法打开则返回 None；如果文件的格式无法猜测则返回空字符串 ('')；或是包含所需模块名称的字符串，例如 'dbm.ndbm' 或 'dbm.gnu'。

dbm.**open**(*file*, *flag='r'*, *mode=0o666*)
打开数据库文件 *file* 并返回一个相应的对象。

如果数据库文件已存在，则使用*whichdb()* 函数来确定其类型和要使用的适当模块；如果文件不存在，则会使用上述可导入模块中的第一个。

可选的 *flag* 参数可以是：

值	意义
'r'	以只读方式打开现有数据库（默认）
'w'	以读写方式打开现有数据库
'c'	以读写方式打开数据库，如果不存在则创建它
'n'	始终创建一个新的空数据库，以读写方式打开

可选的 *mode* 参数是文件的 Unix 模式，仅在要创建数据库时才会被使用。其默认值为八进制数 0o666 (并将被当前的 umask 所修改)。

open() 所返回的对象支持与字典相同的基本功能；可以存储、获取和删除键及其对应的值，并可使用 in 运算符和 keys() 方法，以及 get() 和 setdefault()。

3.2 版更變: 现在 get() 和 setdefault() 在所有数据库模块中均可用。

3.8 版更變: 从只读数据库中删除键将引发数据库模块专属的错误而不是*KeyError*。

键和值总是被存储为字节串。这意味着当使用字符串时它们会在被存储之前隐式地转换至默认编码格式。

这些对象也支持在 with 语句中使用，当语句结束时将自动关闭它们。

3.4 版更變: 向*open()* 所返回的对象添加了上下文管理协议的原生支持。

以下示例记录了一些主机名和对应的标题，随后将数据库的内容打印出来。:

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value will raise an exception (most
    # likely a TypeError).
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

也参考:

模块*shelve* 存储非字符串数据的持久化模块。

以下部分描述了各个单独的子模块。

12.5.1 dbm.gnu --- GNU 对 dbm 的重解析

源代码: Lib/dbm/gnu.py

此模块与*dbm*模块很相似，但是改用 GNU 库 gdbm 来提供某些附加功能。请注意由*dbm.gnu*与*dbm.ndbm*所创建的文件格式是不兼容的。

*dbm.gnu*模块提供了对 GNU DBM 库的接口。dbm.gnu.gdbm 对象的行为类似于映射（字典），区别在于其键和值总是会在存储之前被转换为字节串。打印 gdbm 对象不会打印出键和值，并且 items() 和 values() 等方法也不受支持。

exception dbm.gnu.**error**
针对*dbm.gnu*专属错误例如 I/O 错误引发。*KeyError*的引发则针对一般映射错误例如指定了不正确的键。

dbm.gnu.**open**(*filename*[, *flag*[, *mode*]])
打开一个 gdbm 数据库并返回 gdbm 对象。*filename*参数为数据库文件名称。

可选的 *flag* 参数可以是:

值	意义
'r'	以只读方式打开现有数据库（默认）
'w'	以读写方式打开现有数据库
'c'	以读写方式打开数据库，如果不存在则创建它
'n'	始终创建一个新的空数据库，以读写方式打开

下列附加字符可被添加至旗标以控制数据库的打开方式：

值	意义
'f'	以快速模式打开数据库。写入数据库将不会同步。
's'	同步模式。这将导致数据库的更改立即写入文件。
'u'	不要鎖住資料庫

不是所有旗标都可用于所有版本的 gdbm。模块常量 open_flags 为包含受支持旗标字符的字符串。如果指定了无效的旗标则会引发*error*。

可选的 *mode* 参数是文件的 Unix 模式，仅在要创建数据库时才会被使用。其默认值为八进制数 0o666。

除了与字典类似的方法，gdbm 对象还有以下方法：

gdbm.**firstkey**()
: 使用此方法和*nextkey()* 方法可以循环遍历数据库中的每个键。遍历的顺序是按照 gdbm 的内部哈希值，而不会根据键的值排序。此方法将返回起始键。

gdbm.**nextkey**(*key*)
: 在遍历中返回 *key* 之后的的下一个键。以下代码将打印数据库 db 中的每个键，而不会在内存中创建一个包含所有键的列表：

```
k = db.firstkey()
while k is not None:
    print(k)
    k = db.nextkey(k)
```

gdbm.**reorganize**()
: 如果你进行了大量删除操作并且想要缩减 gdbm 文件所使用的空间，此例程将可重新组织数据库。除非使用此重组功能否则 gdbm 对象不会缩减数据库文件大小；在其他情况下，被删除的文件空间将会保留并在添加新的 (键, 值) 对时被重用。

gdbm.**sync**()
: 当以快速模式打开数据库时，此方法会将任何未写入数据强制写入磁盘。

gdbm.**close**()
: 关闭 gdbm 数据库。

12.5.2 dbm.ndbm --- 基于 ndbm 的接口

源代码: Lib/dbm/ndbm.py

---

dbm.ndbm 模块提供了对 Unix "(n)dbm" 库的接口。Dbm 对象的行为类似于映射（字典），区别在于其键和值总是被存储为字节串。打印 dbm 对象不会打印出键和值，并且 items() 和 values() 等方法也不受支持。

此模块可与" 经典 classic" ndbm 接口或 GNU GDBM 兼容接口一同使用。在 Unix 上，**configure** 脚本将尝试定位适当的头文件来简化此模块的构建。

exception dbm.ndbm.**error**
: 针对*dbm.ndbm* 专属错误例如 I/O 错误引发。*KeyError* 的引发则针对一般映射错误例如指定了不正确的键。

dbm.ndbm.**library**
: 所使用的 ndbm 实现库的名称。

dbm.ndbm.**open**(*filename*[, *flag*[, *mode*]])
打开一个 dbm 数据库并返回 ndbm 对象。*filename* 参数为数据库文件名称（不带 .dir 或 .pag 扩展名）。

可选的 *flag* 参数必须是下列值之一：

值	意义
'r'	以只读方式打开现有数据库（默认）
'w'	以读写方式打开现有数据库
'c'	以读写方式打开数据库，如果不存在则创建它
'n'	始终创建一个新的空数据库，以读写方式打开

可选的 *mode* 参数是文件的 Unix 模式，仅在要创建数据库时才会被使用。其默认值为八进制数 0o666 (并将被当前的 umask 所修改)。

除了与字典类似的方法，ndbm 对象还有以下方法：

ndbm.**close**()
关闭 ndbm 数据库。

12.5.3 dbm.dumb --- 便携式 DBM 实现

源代码: Lib/dbm/dumb.py

備🄵: *dbm.dumb* 模块的目的是在更健壮的模块不可用时作为*dbm* 模块的最终回退项。*dbm.dumb* 不是为高速运行而编写的，也不像其他数据库模块一样被经常使用。

---

dbm.dumb 模块提供了一个完全以 Python 编写的持久化字典类接口。不同于*dbm.gnu* 等其他模块，它不需要外部库。与其他持久化映射一样，它的键和值也总是被存储为字节串。

该模块定义以下内容：

exception dbm.dumb.**error**
针对*dbm.dumb* 专属错误例如 I/O 错误引发。*KeyError* 的引发则针对一般映射例如指定了不正确的键。

dbm.dumb.**open**(*filename*[, *flag*[, *mode*]])
打开一个 dumbdbm 数据库并返回 dumbdbm 对象。*filename* 参数为数据库文件的主名称（不带任何特定扩展名）。创建一个 dumbdbm 数据库时将创建多个带有 .dat 和 .dir 扩展名的文件。

可选的 *flag* 参数可以是：

值	意义
'r'	以只读方式打开现有数据库（默认）
'w'	以读写方式打开现有数据库
'c'	以读写方式打开数据库，如果不存在则创建它
'n'	始终创建一个新的空数据库，以读写方式打开

可选的 *mode* 参数是文件的 Unix 模式，仅在要创建数据库时才会被使用。其默认值为八进制数 0o666 (并将被当前的 umask 所修改)。

> **警告：** 当载入包含足够巨大/复杂条目的数据库时有可能导致 Python 解释器的崩溃，这是由于 Python AST 编译器有栈深度限制。

3.5 版更變: *open()* 在 flag 值为 `'n'` 时将总是创建一个新的数据库。

3.8 版更變: 附带 `'r'` 旗标打开的数据库现在将是只读的。附带 `'r'` 和 `'w'` 旗标的打开操作不会再创建数据库。

除了*collections.abc.MutableMapping* 类所提供的方法，dumbdbm 对象还提供了以下方法：

dumbdbm.**sync**()
同步磁盘上的目录和数据文件。此方法会由 `Shelve.sync()` 方法来调用。

dumbdbm.**close**()
关闭 dumbdbm 数据库。

12.6 sqlite3 --- SQLite 数据库 DB-API 2.0 接口模块

源代码： Lib/sqlite3/

SQLite 是一个 C 语言库，它可以提供一种轻量级的基于磁盘的数据库，这种数据库不需要独立的服务器进程，也允许需要使用一种非标准的 SQL 查询语言来访问它。一些应用程序可以使用 SQLite 作为内部数据存储。可以用它来创建一个应用程序原型，然后再迁移到更大的数据库，比如 PostgreSQL 或 Oracle。

The sqlite3 module was written by Gerhard Häring. It provides an SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**.

To use the module, start by creating a *Connection* object that represents the database. Here the data will be stored in the `example.db` file:

```
import sqlite3
con = sqlite3.connect('example.db')
```

The special path name `:memory:` can be provided to create a temporary database in RAM.

Once a *Connection* has been established, create a *Cursor* object and call its *execute()* method to perform SQL commands:

```
cur = con.cursor()

# Create table
cur.execute('''CREATE TABLE stocks
               (date text, trans text, symbol text, qty real, price real)''')

# Insert a row of data
cur.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")

# Save (commit) the changes
con.commit()

# We can also close the connection if we are done with it.
# Just be sure any changes have been committed or they will be lost.
con.close()
```

The saved data is persistent: it can be reloaded in a subsequent session even after restarting the Python interpreter:

```
import sqlite3
con = sqlite3.connect('example.db')
cur = con.cursor()
```

To retrieve data after executing a SELECT statement, either treat the cursor as an *iterator*, call the cursor's `fetchone()` method to retrieve a single matching row, or call `fetchall()` to get a list of the matching rows.

下面是一个使用迭代器形式的例子:

```
>>> for row in cur.execute('SELECT * FROM stocks ORDER BY price'):
        print(row)

('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
('2006-04-06', 'SELL', 'IBM', 500, 53.0)
('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

SQL operations usually need to use values from Python variables. However, beware of using Python's string operations to assemble queries, as they are vulnerable to SQL injection attacks (see the xkcd webcomic for a humorous example of what can go wrong):

```
# Never do this -- insecure!
symbol = 'RHAT'
cur.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)
```

Instead, use the DB-API's parameter substitution. To insert a variable into a query string, use a placeholder in the string, and substitute the actual values into the query by providing them as a `tuple` of values to the second argument of the cursor's `execute()` method. An SQL statement may use one of two kinds of placeholders: question marks (qmark style) or named placeholders (named style). For the qmark style, `parameters` must be a *sequence*. For the named style, it can be either a *sequence* or `dict` instance. The length of the *sequence* must match the number of placeholders, or a `ProgrammingError` is raised. If a `dict` is given, it must contain keys for all named parameters. Any extra items are ignored. Here's an example of both styles:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table lang (name, first_appeared)")

# This is the qmark style:
cur.execute("insert into lang values (?, ?)", ("C", 1972))

# The qmark style used with executemany():
lang_list = [
    ("Fortran", 1957),
    ("Python", 1991),
    ("Go", 2009),
]
cur.executemany("insert into lang values (?, ?)", lang_list)

# And this is the named style:
cur.execute("select * from lang where first_appeared=:year", {"year": 1972})
print(cur.fetchall())

con.close()
```

也参考:

https://www.sqlite.org SQLite 的主页；它的文档详细描述了它所支持的 SQL 方言的语法和可用的数据类型。

https://www.w3schools.com/sql/ 学习 SQL 语法的教程、参考和例子。

PEP 249 - DB-API 2.0 规范 Marc-André Lemburg 写的 PEP。

12.6.1 模块函数和常量

`sqlite3.`**`apilevel`**
String constant stating the supported DB-API level. Required by the DB-API. Hard-coded to `"2.0"`.

`sqlite3.`**`paramstyle`**
String constant stating the type of parameter marker formatting expected by the *`sqlite3`* module. Required by the DB-API. Hard-coded to `"qmark"`.

備𝔽: The *`sqlite3`* module supports both `qmark` and `numeric` DB-API parameter styles, because that is what the underlying SQLite library supports. However, the DB-API does not allow multiple values for the `paramstyle` attribute.

`sqlite3.`**`version`**
这个模块的版本号，是一个字符串。不是 SQLite 库的版本号。

`sqlite3.`**`version_info`**
这个模块的版本号，是一个由整数组成的元组。不是 SQLite 库的版本号。

`sqlite3.`**`sqlite_version`**
使用中的 SQLite 库的版本号，是一个字符串。

`sqlite3.`**`sqlite_version_info`**
使用中的 SQLite 库的版本号，是一个整数组成的元组。

`sqlite3.`**`threadsafety`**
Integer constant required by the DB-API, stating the level of thread safety the *`sqlite3`* module supports. Currently hard-coded to `1`, meaning *"Threads may share the module, but not connections."* However, this may not always be true. You can check the underlying SQLite library's compile-time threaded mode using the following query:

```
import sqlite3
con = sqlite3.connect(":memory:")
con.execute("""
    select * from pragma_compile_options
    where compile_options like 'THREADSAFE=%'
""").fetchall()
```

Note that the SQLITE_THREADSAFE levels do not match the DB-API 2.0 `threadsafety` levels.

`sqlite3.`**`PARSE_DECLTYPES`**
这个常量可以作为*`connect()`* 函数的 *detect_types* 参数。

设置这个参数后，*`sqlite3`* 模块将解析它返回的每一列申明的类型。它会申明的类型的第一个单词，比如“integer primary key”，它会解析出“integer”，再比如“number(10)”，它会解析出“number”。然后，它会在转换器字典里查找那个类型注册的转换器函数，并调用它。

`sqlite3.`**`PARSE_COLNAMES`**
这个常量可以作为*`connect()`* 函数的 *detect_types* 参数。

设置此参数可使得 SQLite 接口解析它所返回的每一列的列名。它将在其中查找形式为 [mytype] 的字符串，然后将'mytype' 确定为列的类型。它将尝试在转换器字典中查找'mytype' 条目，然后用找到的转换

器函数来返回值。在*Cursor.description* 中找到的列名并不包括类型，举例来说，如果你在你的 SQL 中使用了像 `'as "Expiration date [datetime]"'` 这样的写法，那么我们将解析出在第一个 `'['` 之前的所有内容并去除前导空格作为列名：即列名将为"Expiration date"。

sqlite3.**connect**(*database*[, *timeout*, *detect_types*, *isolation_level*, *check_same_thread*, *factory*, *cached_statements*, *uri*])

连接 SQLite 数据库 *database*。默认返回*Connection* 对象，除非使用了自定义的 *factory* 参数。

database 是准备打开的数据库文件的路径（绝对路径或相对于当前目录的相对路径），它是*path-like object*。你也可以用 `":memory:"` 在内存中打开一个数据库。

当一个数据库被多个连接访问的时候，如果其中一个进程修改这个数据库，在这个事务提交之前，这个 SQLite 数据库将会被一直锁定。*timeout* 参数指定了这个连接等待锁释放的超时时间，超时之后会引发一个异常。这个超时时间默认是 5.0（5 秒）。

isolation_level 参数，请查看*Connection* 对象的*isolation_level* 属性。

SQLite 原生只支持 5 种类型：TEXT，INTEGER，REAL，BLOB 和 NULL。如果你想用其它类型，你必须自己添加相应的支持。使用 *detect_types* 参数和模块级别的*register_converter()* 函数注册 ** 转换器 ** 可以简单的实现。

detect_types 默认为 0 (即关闭，不进行类型检测)，你可以将其设为任意的*PARSE_DECLTYPES* 和*PARSE_COLNAMES* 组合来启用类型检测。由于 SQLite 的行为，生成的字段类型 (例如 `max(data)`) 不能被检测，即使在设置了 *detect_types* 形参时也是如此。在此情况下，返回的类型为*str*。

默认情况下，*check_same_thread* 为*True*，只有当前的线程可以使用该连接。如果设置为*False*，则多个线程可以共享返回的连接。当多个线程使用同一个连接的时候，用户应该把写操作进行序列化，以避免数据损坏。

默认情况下，当调用 connect 方法的时候，*sqlite3* 模块使用了它的*Connection* 类。当然，你也可以创建*Connection* 类的子类，然后创建提供了 *factory* 参数的*connect()* 方法。

详情请查阅当前手册的*SQLite 与 Python 类型* 部分。

sqlite3 模块在内部使用语句缓存来避免 SQL 解析开销。如果要显式设置当前连接可以缓存的语句数，可以设置 *cached_statements* 参数。当前实现的默认值是缓存 100 条语句。

If *uri* is *True*, *database* is interpreted as a URI (Uniform Resource Identifier) with a file path and an optional query string. The scheme part *must* be `"file:"`. The path can be a relative or absolute file path. The query string allows us to pass parameters to SQLite. Some useful URI tricks include:

```
# Open a database in read-only mode.
con = sqlite3.connect("file:template.db?mode=ro", uri=True)

# Don't implicitly create a new database file if it does not already exist.
# Will raise sqlite3.OperationalError if unable to open a database file.
con = sqlite3.connect("file:nosuchdb.db?mode=rw", uri=True)

# Create a shared named in-memory database.
con1 = sqlite3.connect("file:mem1?mode=memory&cache=shared", uri=True)
con2 = sqlite3.connect("file:mem1?mode=memory&cache=shared", uri=True)
con1.executescript("create table t(t); insert into t values(28);")
rows = con2.execute("select * from t").fetchall()
```

More information about this feature, including a list of recognized parameters, can be found in the SQLite URI documentation.

引发一个审计事件 `sqlite3.connect`，附带参数 `database`。

3.4 版更變: 增加了 *uri* 参数。

3.7 版更變: *database* 现在可以是一个*path-like object* 对象了，不仅仅是字符串。

sqlite3.**register_converter**(*typename*, *callable*)
注册一个回调对象 *callable*，用来转换数据库中的字节串为自定的 Python 类型。所有类型为 *typename* 的数据库的值在转换时，都会调用这个回调对象。通过指定 *connect()* 函数的 *detect-types* 参数来设置类型检测的方式。注意，*typename* 与查询语句中的类型名进行匹配时不区分大小写。

sqlite3.**register_adapter**(*type*, *callable*)
注册一个回调对象 *callable*，用来转换自定义 Python 类型为一个 SQLite 支持的类型。这个回调对象 *callable* 仅接受一个 Python 值作为参数，而且必须返回以下某个类型的值：int，float，str 或 bytes。

sqlite3.**complete_statement**(*sql*)
如果字符串 *sql* 包含一个或多个完整的 SQL 语句（以分号结束）则返回 *True*。它不会验证 SQL 语法是否正确，仅会验证字符串字面上是否完整，以及是否以分号结束。

它可以用来构建一个 SQLite shell，下面是一个例子：

```
# A minimal SQLite shell for experiments

import sqlite3

con = sqlite3.connect(":memory:")
con.isolation_level = None
cur = con.cursor()

buffer = ""

print("Enter your SQL commands to execute in sqlite3.")
print("Enter a blank line to exit.")

while True:
    line = input()
    if line == "":
        break
    buffer += line
    if sqlite3.complete_statement(buffer):
        try:
            buffer = buffer.strip()
            cur.execute(buffer)

            if buffer.lstrip().upper().startswith("SELECT"):
                print(cur.fetchall())
        except sqlite3.Error as e:
            print("An error occurred:", e.args[0])
        buffer = ""

con.close()
```

sqlite3.**enable_callback_tracebacks**(*flag*)
默认情况下，您不会获得任何用户定义函数中的回溯消息，比如聚合，转换器，授权器回调等。如果要调试它们，可以设置 *flag* 参数为 `True` 并调用此函数。之后，回调中的回溯信息将会输出到 `sys.stderr`。再次使用 *False* 来禁用该功能。

12.6.2 连接对象（Connection）

class sqlite3.**Connection**

An SQLite database connection has the following attributes and methods:

isolation_level

获取或设置当前默认的隔离级别。表示自动提交模式的*None* 以及"DEFERRED", "IMMEDIATE" 或"EXCLUSIVE" 其中之一。详细描述请参阅控制事务。

in_transaction

如果是在活动事务中（还没有提交改变），返回*True*，否则，返回*False*。它是一个只读属性。

3.2 版新加入.

cursor(*factory=Cursor*)

这个方法接受一个可选参数 *factory*，如果要指定这个参数，它必须是一个可调用对象，而且必须返回*Cursor* 类的一个实例或者子类。

commit()

这个方法提交当前事务。如果没有调用这个方法，那么从上一次提交 commit() 以来所有的变化在其他数据库连接上都是不可见的。如果你往数据库里写了数据，但是又查询不到，请检查是否忘记了调用这个方法。

rollback()

这个方法回滚从上一次调用*commit()* 以来所有数据库的改变。

close()

关闭数据库连接。注意，它不会自动调用*commit()* 方法。如果在关闭数据库连接之前没有调用*commit()*，那么你的修改将会丢失!

execute(*sql*[, *parameters*])

Create a new *Cursor* object and call *execute()* on it with the given *sql* and *parameters*. Return the new cursor object.

executemany(*sql*[, *parameters*])

Create a new *Cursor* object and call *executemany()* on it with the given *sql* and *parameters*. Return the new cursor object.

executescript(*sql_script*)

Create a new *Cursor* object and call *executescript()* on it with the given *sql_script*. Return the new cursor object.

create_function(*name*, *num_params*, *func*, *, *deterministic=False*)

创建一个可以在 SQL 语句中使用的用户自定义函数，函数名为 *name*。*num_params* 为该函数所接受的形参个数（如果 *num_params* 为 -1，则该函数可接受任意数量的参数），*func* 是一个 Python 可调用对象，它将作为 SQL 函数被调用。如果 *deterministic* 为真值，则所创建的函数将被标记为 deterministic，这允许 SQLite 执行额外的优化。此旗标在 SQLite 3.8.3 或更高版本中受到支持，如果在旧版本中使用将引发*NotSupportedError*。

此函数可返回任何 SQLite 所支持的类型: bytes, str, int, float 和 None。

3.8 版更變: 增加了 *deterministic* 形参。

示例:

```
import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()
```

(下页继续)

(繼續上一頁)

```
con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", (b"foo",))
print(cur.fetchone()[0])

con.close()
```

create_aggregate(*name*, *num_params*, *aggregate_class*)

创建一个自定义的聚合函数。

参数中 *aggregate_class* 类必须实现两个方法：`step` 和 `finalize`。`step` 方法接受 *num_params* 个参数（如果 *num_params* 为 -1，那么这个函数可以接受任意数量的参数）；`finalize` 方法返回最终的聚合结果。

`finalize` 方法可以返回任何 SQLite 支持的类型：bytes，str，int，float 和 `None`。

示例:

```
import sqlite3

class MySum:
    def __init__(self):
        self.count = 0

    def step(self, value):
        self.count += value

    def finalize(self):
        return self.count

con = sqlite3.connect(":memory:")
con.create_aggregate("mysum", 1, MySum)
cur = con.cursor()
cur.execute("create table test(i)")
cur.execute("insert into test(i) values (1)")
cur.execute("insert into test(i) values (2)")
cur.execute("select mysum(i) from test")
print(cur.fetchone()[0])

con.close()
```

create_collation(*name*, *callable*)

使用 *name* 和 *callable* 创建排序规则。这个 *callable* 接受两个字符串对象，如果第一个小于第二个则返回 -1，如果两个相等则返回 0，如果第一个大于第二个则返回 1。注意，这是用来控制排序的（SQL 中的 ORDER BY），所以它不会影响其它的 SQL 操作。

注意，这个 *callable* 可调用对象会把它的参数作为 Python 字节串，通常会以 UTF-8 编码格式对它进行编码。

以下示例显示了使用“错误方式”进行排序的自定义排序规则:

```
import sqlite3

def collate_reverse(string1, string2):
    if string1 == string2:
        return 0
    elif string1 < string2:
```

(下页继续)

(繼續上一頁)

```
        return 1
    else:
        return -1

con = sqlite3.connect(":memory:")
con.create_collation("reverse", collate_reverse)

cur = con.cursor()
cur.execute("create table test(x)")
cur.executemany("insert into test(x) values (?)", [("a",), ("b",)])
cur.execute("select x from test order by x collate reverse")
for row in cur:
    print(row)
con.close()
```

要移除一个排序规则，需要调用 `create_collation` 并设置 callable 参数为 `None`。

```
con.create_collation("reverse", None)
```

interrupt()

可以从不同的线程调用这个方法来终止所有查询操作，这些查询操作可能正在连接上执行。此方法调用之后，查询将会终止，而且查询的调用者会获得一个异常。

set_authorizer(*authorizer_callback*)

此方法注册一个授权回调对象。每次在访问数据库中某个表的某一列的时候，这个回调对象将会被调用。如果要允许访问，则返回 `SQLITE_OK`，如果要终止整个 SQL 语句，则返回 `SQLITE_DENY`，如果这一列需要当做 NULL 值处理，则返回 `SQLITE_IGNORE`。这些常量可以在*sqlite3* 模块中找到。

回调的第一个参数表示要授权的操作类型。第二个和第三个参数将是参数或*None*，具体取决于第一个参数的值。第 4 个参数是数据库的名称（“main”，“temp” 等），如果需要的话。第 5 个参数是负责访问尝试的最内层触发器或视图的名称，或者如果此访问尝试直接来自输入 SQL 代码，则为*None*。

请参阅 SQLite 文档，了解第一个参数的可能值以及第二个和第三个参数的含义，具体取决于第一个参数。所有必需的常量都可以在*sqlite3* 模块中找到。

set_progress_handler(*handler*, *n*)

此例程注册回调。对 SQLite 虚拟机的每个多指令调用回调。如果要在长时间运行的操作期间从 SQLite 调用（例如更新用户界面），这非常有用。

如果要清除以前安装的任何进度处理程序，调用该方法时请将 *handler* 参数设置为*None*。

从处理函数返回非零值将终止当前正在执行的查询并导致它引发*OperationalError* 异常。

set_trace_callback(*trace_callback*)

为每个 SQLite 后端实际执行的 SQL 语句注册要调用的 *trace_callback*。

The only argument passed to the callback is the statement (as *str*) that is being executed. The return value of the callback is ignored. Note that the backend does not only run statements passed to the *Cursor.execute()* methods. Other sources include the *transaction management* of the sqlite3 module and the execution of triggers defined in the current database.

将传入的 *trace_callback* 设为*None* 将禁用跟踪回调。

備F: Exceptions raised in the trace callback are not propagated. As a development and debugging aid, use *enable_callback_tracebacks()* to enable printing tracebacks from exceptions raised in the trace

callback.

3.3 版新加入.

enable_load_extension(*enabled*)
此例程允许/禁止 SQLite 引擎从共享库加载 SQLite 扩展。SQLite 扩展可以定义新功能，聚合或全新的虚拟表实现。一个众所周知的扩展是与 SQLite 一起分发的全文搜索扩展。

默认情况下禁用可加载扩展。见[1].

3.2 版新加入.

```
import sqlite3

con = sqlite3.connect(":memory:")

# enable extension loading
con.enable_load_extension(True)

# Load the fulltext search extension
con.execute("select load_extension('./fts3.so')")

# alternatively you can load the extension using an API call:
# con.load_extension("./fts3.so")

# disable extension loading again
con.enable_load_extension(False)

# example from SQLite wiki
con.execute("create virtual table recipe using fts3(name, ingredients)")
con.executescript("""
    insert into recipe (name, ingredients) values ('broccoli stew', 'broccoli␣
↪peppers cheese tomatoes');
    insert into recipe (name, ingredients) values ('pumpkin stew', 'pumpkin␣
↪onions garlic celery');
    insert into recipe (name, ingredients) values ('broccoli pie', 'broccoli␣
↪cheese onions flour');
    insert into recipe (name, ingredients) values ('pumpkin pie', 'pumpkin␣
↪sugar flour butter');
    """)
for row in con.execute("select rowid, name, ingredients from recipe where␣
↪name match 'pie'"):
    print(row)

con.close()
```

load_extension(*path*)
This routine loads an SQLite extension from a shared library. You have to enable extension loading with *enable_load_extension()* before you can use this routine.

默认情况下禁用可加载扩展。见[1].

3.2 版新加入.

row_factory
您可以将此属性更改为可接受游标和原始行作为元组的可调用对象，并将返回实际结果行。这样，您可以实现更高级的返回结果的方法，例如返回一个可以按名称访问列的对象。

[1] The sqlite3 module is not built with loadable extension support by default, because some platforms (notably macOS) have SQLite libraries which are compiled without this feature. To get loadable extension support, you must pass `--enable-loadable-sqlite-extensions` to configure.

示例:

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone()["a"])

con.close()
```

如果返回一个元组是不够的，并且你想要对列进行基于名称的访问，你应该考虑将*row_factory*设置为高度优化的*sqlite3.Row* 类型。*Row* 提供基于索引和不区分大小写的基于名称的访问，几乎没有内存开销。它可能比您自己的基于字典的自定义方法甚至基于 db_row 的解决方案更好。

text_factory

Using this attribute you can control what objects are returned for the `TEXT` data type. By default, this attribute is set to *str* and the *sqlite3* module will return *str* objects for `TEXT`. If you want to return *bytes* instead, you can set it to *bytes*.

您还可以将其设置为接受单个 bytestring 参数的任何其他可调用对象，并返回结果对象。

请参阅以下示例代码以进行说明:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()

AUSTRIA = "Österreich"

# by default, rows are returned as str
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert row[0] == AUSTRIA

# but we can make sqlite3 always return bytestrings ...
con.text_factory = bytes
cur.execute("select ?", (AUSTRIA,))
row = cur.fetchone()
assert type(row[0]) is bytes
# the bytestrings will be encoded in UTF-8, unless you stored garbage in the
# database ...
assert row[0] == AUSTRIA.encode("utf-8")

# we can also implement a custom text_factory ...
# here we implement one that appends "foo" to all strings
con.text_factory = lambda x: x.decode("utf-8") + "foo"
cur.execute("select ?", ("bar",))
row = cur.fetchone()
assert row[0] == "barfoo"
```

(下页继续)

(繼續上一頁)

```
con.close()
```

total_changes
返回自打开数据库连接以来已修改，插入或删除的数据库行的总数。

iterdump()
返回以 SQL 文本格式转储数据库的迭代器。保存内存数据库以便以后恢复时很有用。此函数提供与 **sqlite3** shell 中的 .dump 命令相同的功能。

示例:

```
# Convert file existing_db.db to SQL dump file dump.sql
import sqlite3

con = sqlite3.connect('existing_db.db')
with open('dump.sql', 'w') as f:
    for line in con.iterdump():
        f.write('%s\n' % line)
con.close()
```

backup(*target*, ***, *pages=-1*, *progress=None*, *name="main"*, *sleep=0.250*)
This method makes a backup of an SQLite database even while it's being accessed by other clients, or concurrently by the same connection. The copy will be written into the mandatory argument *target*, that must be another *Connection* instance.

默认情况下，或者当 *pages* 为 0 或负整数时，整个数据库将在一个步骤中复制；否则该方法一次循环复制 *pages* 规定数量的页面。

如果指定了 *progress*，则它必须为 None 或一个将在每次迭代时附带三个整数参数执行的可调用对象，这三个参数分别是前一次迭代的状态 *status*，将要拷贝的剩余页数 *remaining* 以及总页数 *total*。

name 参数指定将被拷贝的数据库名称：它必须是一个字符串，其内容为表示主数据库的默认值 "main"，表示临时数据库的 "temp" 或是在 ATTACH DATABASE 语句的 AS 关键字之后指定表示附加数据库的名称。

sleep 参数指定在备份剩余页的连续尝试之间要休眠的秒数，可以指定为一个整数或一个浮点数值。

示例一，将现有数据库复制到另一个数据库中：

```
import sqlite3

def progress(status, remaining, total):
    print(f'Copied {total-remaining} of {total} pages...')

con = sqlite3.connect('existing_db.db')
bck = sqlite3.connect('backup.db')
with bck:
    con.backup(bck, pages=1, progress=progress)
bck.close()
con.close()
```

示例二，将现有数据库复制到临时副本中：

```
import sqlite3

source = sqlite3.connect('existing_db.db')
dest = sqlite3.connect(':memory:')
source.backup(dest)
```

可用性：SQLite 3.6.11 或以上版本

3.7 版新加入.

12.6.3 Cursor 对象

class sqlite3.**Cursor**

Cursor 游标实例具有以下属性和方法。

execute(*sql*[, *parameters*])

执行一条 SQL 语句。可以使用占位符 将值绑定到语句中。

execute() 将只执行一条单独的 SQL 语句。如果你尝试用它执行超过一条语句，将会引发*Warning*。如果你想要用一次调用执行多条 SQL 语句请使用*executescript()*。

executemany(*sql*, *seq_of_parameters*)

执行一条带形参的 SQL 命令，使用所有形参序列或在序列 *seq_of_parameters* 中找到的映射。*sqlite3* 模块还允许使用*iterator* 代替序列来产生形参。

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def __next__(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")

theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)

cur.execute("select c from characters")
print(cur.fetchall())

con.close()
```

这是一个使用生成器*generator* 的简短示例：

```
import sqlite3
import string

def char_generator():
    for c in string.ascii_lowercase:
        yield (c,)

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")
```

(下页继续)

(繼續上一頁)

```
cur.executemany("insert into characters(c) values (?)", char_generator())

cur.execute("select c from characters")
print(cur.fetchall())

con.close()
```

executescript(*sql_script*)

这是用于一次性执行多条 SQL 语句的非标准便捷方法。它会首先发出一条 COMMIT 语句，然后执行通过参数获得的 SQL 脚本。此方法会忽略 isolation_level；任何事件控制都必须被添加到 *sql_script*。

sql_script 可以是一个*str* 类的实例。

示例:

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently''s Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
    """)
con.close()
```

fetchone()

获取一个查询结果集的下一行，返回一个单独序列，或是在没有更多可用数据时返回*None*。

fetchmany(*size=cursor.arraysize*)

获取下一个多行查询结果集，返回一个列表。当没有更多可用行时将返回一个空列表。

每次调用获取的行数由 *size* 形参指定。如果没有给出该形参，则由 cursor 的 arraysize 决定要获取的行数。此方法将基于 size 形参值尝试获取指定数量的行。如果获取不到指定的行数，则可能返回较少的行。

请注意 *size* 形参会涉及到性能方面的考虑。为了获得优化的性能，通常最好是使用 arraysize 属性。如果使用 *size* 形参，则最好在从一个*fetchmany()* 调用到下一个调用之间保持相同的值。

fetchall()

获取一个查询结果的所有（剩余）行，返回一个列表。请注意 cursor 的 arraysize 属性会影响此操作的执行效率。当没有可用行时将返回一个空列表。

close()
立即关闭 cursor（而不是在当 __del__ 被调用的时候）。

从这一时刻起该 cursor 将不再可用，如果再尝试用该 cursor 执行任何操作将引发*ProgrammingError* 异常。

setinputsizes(*sizes*)
Required by the DB-API. Does nothing in *sqlite3*.

setoutputsize(*size*[, *column*])
Required by the DB-API. Does nothing in *sqlite3*.

rowcount
虽然*sqlite3* 模块的*Cursor* 类实现了此属性，但数据库引擎本身对于确定" 受影响行"/" 已选择行" 的支持并不完善。

对于*executemany()* 语句，修改行数会被汇总至*rowcount*。

根据 Python DB API 规格描述的要求，*rowcount* 属性" 当未在 cursor 上执行 executeXX() 或者上一次操作的 rowcount 不是由接口确定时为 -1"。这包括 SELECT 语句，因为我们无法确定一次查询将产生的行计数，而要等获取了所有行时才会知道。

在 SQLite 的 3.6.5 版之前，如果你执行 DELETE FROM table 时不附带任何条件，则*rowcount* 将被设为 0。

lastrowid
This read-only attribute provides the row id of the last inserted row. It is only updated after successful INSERT or REPLACE statements using the *execute()* method. For other statements, after *executemany()* or *executescript()*, or if the insertion failed, the value of lastrowid is left unchanged. The initial value of lastrowid is *None*.

備F: Inserts into WITHOUT ROWID tables are not recorded.

3.6 版更變: 增加了 REPLACE 语句的支持。

arraysize
用于控制*fetchmany()* 返回行数的可读取/写入属性。该属性的默认值为 1，表示每次调用将获取单独一行。

description
这个只读属性将提供上一次查询的列名称。为了与 Python DB API 保持兼容，它会为每个列返回一个 7 元组，每个元组的最后六个条目均为*None*。

对于没有任何匹配行的 SELECT 语句同样会设置该属性。

connection
这个只读属性将提供*Cursor* 对象所使用的 SQLite 数据库*Connection*。通过调用*con.cursor()* 创建的*Cursor* 对象所包含的*connection* 属性将指向 *con*:

```
>>> con = sqlite3.connect(":memory:")
>>> cur = con.cursor()
>>> cur.connection == con
True
```

12.6.4 行对象

class sqlite3.**Row**

一个*Row* 实例，该实例将作为用于*Connection* 对象的高度优化的*row_factory*。它的大部分行为都会模仿元组的特性。

它支持使用列名称的映射访问以及索引、迭代、文本表示、相等检测和*len()* 等操作。

如果两个*Row* 对象具有完全相同的列并且其成员均相等，则它们的比较结果为相等。

keys()

此方法会在一次查询之后立即返回一个列名称的列表，它是*Cursor.description* 中每个元组的第一个成员。

3.5 版更變: 添加了对切片操作的支持。

让我们假设我们如上面的例子所示初始化一个表:

```
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')
cur.execute("""insert into stocks
            values ('2006-01-05','BUY','RHAT',100,35.14)""")
con.commit()
cur.close()
```

现在我们将*Row* 插入:

```
>>> con.row_factory = sqlite3.Row
>>> cur = con.cursor()
>>> cur.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = cur.fetchone()
>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
...     print(member)
...
2006-01-05
BUY
RHAT
100.0
35.14
```

12.6.5 异常

exception sqlite3.**Warning**
Exception 的一个子类。

exception sqlite3.**Error**
此模块中其他异常的基类。它是*Exception* 的一个子类。

exception sqlite3.**DatabaseError**
针对数据库相关错误引发的异常。

exception sqlite3.**IntegrityError**
当数据库的关系一致性受到影响时引发的异常。例如外键检查失败等。它是*DatabaseError* 的子类。

exception sqlite3.**ProgrammingError**
编程错误引发的异常，例如表未找到或已存在，SQL 语句存在语法错误，指定的形参数量错误等。它是*DatabaseError* 的子类。

exception sqlite3.**OperationalError**
与数据库操作相关而不一定能受程序员掌控的错误引发的异常，例如发生非预期的连接中断，数据源名称未找到，事务无法被执行等。它是*DatabaseError* 的子类。

exception sqlite3.**NotSupportedError**
在使用了某个数据库不支持的方法或数据库 API 时引发的异常，例如在一个不支持事务或禁用了事务的连接上调用*rollback()* 方法等。它是*DatabaseError* 的子类。

12.6.6 SQLite 与 Python 类型

簡介

SQLite 原生支持如下的类型：NULL，INTEGER，REAL，TEXT，BLOB。

因此可以将以下 Python 类型发送到 SQLite 而不会出现任何问题：

Python 类型	SQLite 类型
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

这是 SQLite 类型默认转换为 Python 类型的方式：

SQLite 类型	Python 类型
NULL	*None*
INTEGER	*int*
REAL	*float*
TEXT	取决于*text_factory*，默认为*str*
BLOB	*bytes*

The type system of the *sqlite3* module is extensible in two ways: you can store additional Python types in an SQLite database via object adaptation, and you can let the *sqlite3* module convert SQLite types to different Python types via converters.

使用适配器将额外的 Python 类型保存在 SQLite 数据库中。

如上文所述，SQLite 只包含对有限类型集的原生支持。要让 SQLite 能使用其他 Python 类型，你必须将它们**适配**至 sqlite3 模块所支持的 SQLite 类型中的一种：NoneType, int, float, str, bytes。

有两种方式能让*sqlite3*模块将某个定制的 Python 类型适配为受支持的类型。

让对象自行适配

如果类是你自己编写的，这将是一个很好的方式。假设你有这样一个类:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
```

现在你可将这种点对象保存在一个 SQLite 列中。首先你必须选择一种受支持的类型用来表示点对象。让我们就用 str 并使用一个分号来分隔坐标值。然后你需要给你的类加一个方法 `__conform__(self, protocol)`，它必须返回转换后的值。形参 *protocol* 将为 `PrepareProtocol`。

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __conform__(self, protocol):
        if protocol is sqlite3.PrepareProtocol:
            return "%f;%f" % (self.x, self.y)

con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])

con.close()
```

注册可调用的适配器

另一种可能的做法是创建一个将该类型转换为字符串表示的函数并使用*register_adapter()*注册该函数。

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

def adapt_point(point):
    return "%f;%f" % (point.x, point.y)

sqlite3.register_adapter(Point, adapt_point)
```

(下页继续)

(繼續上一頁)

```
con = sqlite3.connect(":memory:")
cur = con.cursor()

p = Point(4.0, -3.2)
cur.execute("select ?", (p,))
print(cur.fetchone()[0])

con.close()
```

sqlite3 模块有两个适配器可用于 Python 的内置*datetime.date* 和*datetime.datetime* 类型。现在假设我们想要存储*datetime.datetime* 对象，但不是表示为 ISO 格式，而是表示为 Unix 时间戳。

```
import sqlite3
import datetime
import time

def adapt_datetime(ts):
    return time.mktime(ts.timetuple())

sqlite3.register_adapter(datetime.datetime, adapt_datetime)

con = sqlite3.connect(":memory:")
cur = con.cursor()

now = datetime.datetime.now()
cur.execute("select ?", (now,))
print(cur.fetchone()[0])

con.close()
```

将 SQLite 值转换为自定义 Python 类型

编写适配器让你可以将定制的 Python 类型发送给 SQLite。但要令它真正有用，我们需要实现从 Python 到 SQLite 再回到 Python 的双向转换。

输入转换器。

让我们回到 `Point` 类。我们以字符串形式在 SQLite 中存储了 x 和 y 坐标值。

首先，我们将定义一个转换器函数，它接受这样的字符串作为形参并根据该参数构造一个 `Point` 对象。

備F: 转换器函数在调用时 **总是**会附带一个*bytes* 对象，无论你将何种数据类型的值发给 SQLite。

```
def convert_point(s):
    x, y = map(float, s.split(b";"))
    return Point(x, y)
```

现在你需要让*sqlite3* 模块知道你从数据库中选取的其实是一个点对象。有两种方式都可以做到这件事：

- 隐式的声明类型
- 显式的通过列名

这两种方式会在*模块函数和常量* 一节中描述，相应条目为*PARSE_DECLTYPES* 和*PARSE_COLNAMES* 常量。

下面的示例说明了这两种方法。

```
import sqlite3

class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __repr__(self):
        return "(%f;%f)" % (self.x, self.y)

def adapt_point(point):
    return ("%f;%f" % (point.x, point.y)).encode('ascii')

def convert_point(s):
    x, y = list(map(float, s.split(b";")))
    return Point(x, y)

# Register the adapter
sqlite3.register_adapter(Point, adapt_point)

# Register the converter
sqlite3.register_converter("point", convert_point)

p = Point(4.0, -3.2)

#########################
# 1) Using declared types
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES)
cur = con.cursor()
cur.execute("create table test(p point)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute("select p from test")
print("with declared types:", cur.fetchone()[0])
cur.close()
con.close()

######################
# 1) Using column names
con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(p)")

cur.execute("insert into test(p) values (?)", (p,))
cur.execute('select p as "p [point]" from test')
print("with column names:", cur.fetchone()[0])
cur.close()
con.close()
```

默认适配器和转换器

对于 datetime 模块中的 date 和 datetime 类型已提供了默认的适配器。它们将会以 ISO 日期/ISO 时间戳的形式发给 SQLite。

默认转换器使用的注册名称是针对*datetime.date* 的"date" 和针对*datetime.datetime* 的"timestamp"。

通过这种方式，你可以在大多数情况下使用 Python 的 date/timestamp 对象而无须任何额外处理。适配器的格式还与实验性的 SQLite date/time 函数兼容。

下面的示例演示了这一点。

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_
↪COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print(today, "=>", row[0], type(row[0]))
print(now, "=>", row[1], type(row[1]))

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"
↪')
row = cur.fetchone()
print("current_date", row[0], type(row[0]))
print("current_timestamp", row[1], type(row[1]))

con.close()
```

如果存储在 SQLite 中的时间戳的小数位多于 6 个数字，则时间戳转换器会将该值截断至微秒精度。

備🄵: The default "timestamp" converter ignores UTC offsets in the database and always returns a naive *datetime.datetime* object. To preserve UTC offsets in timestamps, either leave converters disabled, or register an offset-aware converter with *register_converter()*.

12.6.7 控制事务

底层的 sqlite3 库默认会以 autocommit 模式运行，但 Python 的*sqlite3* 模块默认则不使用此模式。

autocommit 模式意味着修改数据库的操作会立即生效。BEGIN 或 SAVEPOINT 语句会禁用 autocommit 模式，而用于结束外层事务的 COMMIT, ROLLBACK 或 RELEASE 则会恢复 autocommit 模式。

Python 的*sqlite3* 模块默认会在数据修改语言 (DML) 类语句 (即 INSERT/UPDATE/DELETE/REPLACE) 之前隐式地执行一条 BEGIN 语句。

你可以控制*sqlite3* 隐式执行的 BEGIN 语句的种类，具体做法是通过将 *isolation_level* 形参传给*connect()* 调用，或者通过指定连接的 isolation_level 属性。如果你没有指定 *isolation_level*，将使用基本的 BEGIN，它等价于指定 DEFERRED。其他可能的值为 IMMEDIATE 和 EXCLUSIVE。

你可以禁用*sqlite3*模块的隐式事务管理，具体做法是将 isolation_level 设为 None。这将使得下层的 sqlite3 库采用 autocommit 模式。随后你可以通过在代码中显式地使用 BEGIN, ROLLBACK, SAVEPOINT 和 RELEASE 语句来完全控制事务状态。

请注意*executescript()* 会忽略 isolation_level；任何事务控制必要要显式地添加。

3.6 版更變: 以前*sqlite3* 会在 DDL 语句之前隐式地提交未完成事务。现在则不会再这样做。

12.6.8 有效使用 sqlite3

使用快捷方式

使用*Connection* 对象的非标准 execute(), executemany() 和 executescript() 方法，可以更简洁地编写代码，因为不必显式创建（通常是多余的）*Cursor* 对象。相反，*Cursor* 对象是隐式创建的，这些快捷方法返回游标对象。这样，只需对*Connection* 对象调用一次，就能直接执行 SELECT 语句并遍历对象。

```
import sqlite3

langs = [
    ("C++", 1985),
    ("Objective-C", 1984),
]

con = sqlite3.connect(":memory:")

# Create the table
con.execute("create table lang(name, first_appeared)")

# Fill the table
con.executemany("insert into lang(name, first_appeared) values (?, ?)", langs)

# Print the table contents
for row in con.execute("select name, first_appeared from lang"):
    print(row)

print("I just deleted", con.execute("delete from lang").rowcount, "rows")

# close is not a shortcut method and it's not called automatically,
# so the connection object should be closed manually
con.close()
```

通过名称而不是索引访问索引

*sqlite3*模块的一个有用功能是内置的*sqlite3.Row* 类，该类旨在用作行工厂。

该类的行装饰器可以用索引（如元组）和不区分大小写的名称访问：

```
import sqlite3

con = sqlite3.connect(":memory:")
con.row_factory = sqlite3.Row

cur = con.cursor()
cur.execute("select 'John' as name, 42 as age")
for row in cur:
```

(下页继续)

(繼續上一頁)

```
    assert row[0] == row["name"]
    assert row["name"] == row["nAmE"]
    assert row[1] == row["age"]
    assert row[1] == row["AgE"]

con.close()
```

使用连接作为上下文管理器

连接对象可以用来作为上下文管理器，它可以自动提交或者回滚事务。如果出现异常，事务会被回滚；否则，事务会被提交。

```
import sqlite3

con = sqlite3.connect(":memory:")
con.execute("create table lang (id integer primary key, name varchar unique)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("insert into lang(name) values (?)", ("Python",))

# con.rollback() is called after the with block finishes with an exception, the
# exception is still raised and must be caught
try:
    with con:
        con.execute("insert into lang(name) values (?)", ("Python",))
except sqlite3.IntegrityError:
    print("couldn't add Python twice")

# Connection object used as context manager only commits or rollbacks transactions,
# so the connection object should be closed manually
con.close()
```

🄵解

CHAPTER 13

資料壓縮與保存

本章中描述的模块支持 zlib、gzip、bzip2 和 lzma 数据压缩算法，以及创建 ZIP 和 tar 格式的归档文件。参见由 *shutil* 模块提供的*归档操作* 。

13.1 zlib --- 与 gzip 兼容的压缩

对于需要数据压缩的应用，此模块中的函数允许使用 zlib 库进行压缩和解压缩。zlib 库的项目主页是 https://www.zlib.net。已知此 Python 模块与 1.1.3 之前版本的 zlib 库存在不兼容；1.1.3 版则存在一个 安全缺陷，因此我们推荐使用 1.1.4 或更新的版本。

zlib 的函数有很多选项，一般需要按特定顺序使用。本文档没有覆盖全部的用法。更多详细信息请于 http://www.zlib.net/manual.html 参阅官方手册。

要读写 `.gz` 格式的文件，请参考*gzip* 模块。

此模块中可用的异常和函数如下：

exception zlib.**error**
在压缩或解压缩过程中发生错误时的异常。

zlib.**adler32**(*data*[, *value*])
计算 *data* 的 Adler-32 校验值。(Adler-32 校验的可靠性与 CRC32 基本相当，但比计算 CRC32 更高效。) 计算的结果是一个 32 位的整数。参数 *value* 是校验时的起始值，其默认值为 1。借助参数 *value* 可为分段的输入计算校验值。此算法没有加密强度，不应用于身份验证和数字签名。此算法的目的仅为验证数据的正确性，不适合作为通用散列算法。

3.0 版更變: The result is always unsigned. To generate the same numeric value when using Python 2 or earlier, use `adler32(data) & 0xffffffff`.

zlib.**compress**(*data*, */*, *level=-1*)
压缩 *data* 中的字节，返回含有已压缩内容的 bytes 对象。参数 *level* 为整数，可取值为 `0` 到 `9` 或 `-1`，用于指定压缩等级。`1` (Z_BEST_SPEED) 表示最快速度和最低压缩率，`9` (Z_BEST_COMPRESSION) 表示最慢速度和最高压缩率。`0` (Z_NO_COMPRESSION) 表示不压缩。参数默认值为 `-1`

(Z_DEFAULT_COMPRESSION)。Z_DEFAULT_COMPRESSION 是速度和压缩率之间的平衡 (一般相当于设压缩等级为 6)。函数发生错误时抛出*error* 异常。

3.6 版更變: 现在，*level* 可作为关键字参数。

zlib.**compressobj**(*level=-1*, *method=DEFLATED*, *wbits=MAX_WBITS*, *memLevel=DEF_MEM_LEVEL*, *strategy=Z_DEFAULT_STRATEGY*[, *zdict*])

返回一个压缩对象，用来压缩内存中难以容下的数据流。

参数 *level* 为压缩等级，是整数，可取值为 0 到 9 或 -1。1 (Z_BEST_SPEED) 表示最快速度和最低压缩率，9 (Z_BEST_COMPRESSION) 表示最慢速度和最高压缩率。0 (Z_NO_COMPRESSION) 表示不压缩。参数默认值为 -1 (Z_DEFAULT_COMPRESSION)。Z_DEFAULT_COMPRESSION 是速度和压缩率之间的平衡 (一般相当于设压缩等级为 6)。

method 表示压缩算法。现在只支持 DEFLATED 这个算法。

参数 *wbits* 指定压缩数据时所使用的历史缓冲区的大小 (窗口大小)，并指定压缩输出是否包含头部或尾部。参数的默认值是 15 (MAX_WBITS)。参数的值分为几个范围：

- +9 至 +15：窗口大小以二为底的对数。即这些值对应着 512 至 32768 的窗口大小。更大的值会提供更好的压缩，同时内存开销也会更大。压缩输出会包含 zlib 特定格式的头部和尾部。
- −9 至 −15：绝对值为窗口大小以二为底的对数。压缩输出仅包含压缩数据，没有头部和尾部。
- +25 至 +31 = 16 + (9 至 15)：后 4 个比特位为窗口大小以二为底的对数。压缩输出包含一个基本的 **gzip** 头部，并以校验和为尾部。

参数 *memLevel* 指定内部压缩操作时所占用内存大小。参数取 1 到 9。更大的值占用更多的内存，同时速度也更快输出也更小。

参数 *strategy* 用于调节压缩算法。可取值为 Z_DEFAULT_STRATEGY、Z_FILTERED、Z_HUFFMAN_ONLY、Z_RLE (zlib 1.2.0.1) 或 Z_FIXED (zlib 1.2.2.2)。

参数 *zdict* 指定预定义的压缩字典。它是一个字节序列 (如*bytes* 对象)，其中包含用户认为要压缩的数据中可能频繁出现的子序列。频率高的子序列应当放在字典的尾部。

3.3 版更變: 添加关键字参数 *zdict*。

zlib.**crc32**(*data*[, *value*])

计算 *data* 的 CRC (循环冗余校验) 值。计算的结果是一个 32 位的整数。参数 *value* 是校验时的起始值，其默认值为 0。借助参数 *value* 可为分段的输入计算校验值。此算法没有加密强度，不应用于身份验证和数字签名。此算法的目的仅为验证数据的正确性，不适合作为通用散列算法。

3.0 版更變: The result is always unsigned. To generate the same numeric value when using Python 2 or earlier, use `crc32(data) & 0xffffffff`.

zlib.**decompress**(*data*, */*, *wbits=MAX_WBITS*, *bufsize=DEF_BUF_SIZE*)

解压 *data* 中的字节，返回含有已解压内容的 bytes 对象。参数 *wbits* 取决于 *data* 的格式，具体参见下边的说明。*bufsize* 为输出缓冲区的起始大小。函数发生错误时抛出*error* 异常。

wbits 形参控制历史缓冲区的大小（或称“窗口大小”）以及所期望的头部和尾部格式。它类似于*compressobj()* 的形参，但可接受更大范围的值：

- +8 至 +15：窗口尺寸以二为底的对数。输入必须包含 zlib 头部和尾部。
- 0：根据 zlib 头部自动确定窗口大小。只从 zlib 1.2.3.5 版起受支持。
- −8 至 −15：使用 *wbits* 的绝对值作为窗口大小以二为底的对数。输入必须为原始数据流，没有头部和尾部。
- +24 至 +31 = 16 + (8 至 15)：使用后 4 个比特位作为窗口大小以二为底的对数。输入必须包括 gzip 头部和尾部。

- +40 至 +47 = 32 + (8 至 15)：使用后 4 个比特位作为窗口大小以二为底的对数，并且自动接受 zlib 或 gzip 格式。

当解压缩一个数据流时，窗口大小必须不小于用于压缩数据流的原始窗口大小；使用太小的值可能导致*error* 异常。默认 *wbits* 值对应于最大的窗口大小并且要求包括 zlib 头部和尾部。

bufsize 是用于存放解压数据的缓冲区初始大小。如果需要更大空间，缓冲区大小将按需增加，因此你不需要让这个值完全精确；对其进行调整仅会节省一点对 `malloc()` 的调用次数。

3.6 版更變: *wbits* 和 *bufsize* 可用作关键字参数。

`zlib.`**`decompressobj`**(*wbits=MAX_WBITS*[, *zdict*])
返回一个解压对象，用来解压无法被一次性放入内存的数据流。

wbits 形参控制历史缓冲区的大小（或称“窗口大小”）以及所期望的头部和尾部格式。它的含义与*对 decompress() 的描述* 相同。

zdict 形参指定指定一个预定义的压缩字典。如果提供了此形参，它必须与产生将解压数据的压缩器所使用的字典相同。

備𝔽: 如果 *zdict* 是一个可变对象 (例如*bytearray*)，则你不可在对*decompressobj()* 的调用和对解压器的 `decompress()` 方法的调用之间修改其内容。

3.3 版更變: 增加了 *zdict* 形参。

压缩对象支持以下方法：

`Compress.`**`compress`**(*data*)
压缩 *data* 并返回 bytes 对象，这个对象含有 *data* 的部分或全部内容的已压缩数据。所得的对象必须拼接在上一次调用*compress()* 方法所得数据的后面。缓冲区中可能留存部分输入以供下一次调用。

`Compress.`**`flush`**([*mode*])
压缩所有缓冲区的数据并返回已压缩的数据。参数 *mode* 可以传入的常量为：`Z_NO_FLUSH`、`Z_PARTIAL_FLUSH`、`Z_SYNC_FLUSH`、`Z_FULL_FLUSH`、`Z_BLOCK` (zlib 1.2.3.4) 或 `Z_FINISH`。默认值为 `Z_FINISH`。`Z_FINISH` 关闭已压缩数据流并不允许再压缩其他数据，`Z_FINISH` 以外的值皆允许这个对象继续压缩数据。调用*flush()* 方法并将 *mode* 设为 `Z_FINISH` 后会无法再次调用*compress()*，此时只能删除这个对象。

`Compress.`**`copy`**()
返回此压缩对象的一个拷贝。它可以用来高效压缩一系列拥有相同前缀的数据。

3.8 版更變: 添加了对压缩对象执行*copy.copy()* 和*copy.deepcopy()* 的支持。

解压缩对象支持以下方法：

`Decompress.`**`unused_data`**
一个 bytes 对象，其中包含压缩数据结束之后的任何字节数据。也就是说，它将为 `b""` 直到包含压缩数据的末尾字节可用。如果整个结果字节串都包含压缩数据，它将为一个空的 bytes 对象 `b""`。

`Decompress.`**`unconsumed_tail`**
一个 bytes 对象，其中包含未被上一次*decompress()* 调用所消耗的任何数据。此数据不能被 zlib 机制看到，因此你必须将其送回（可能要附带额外的数据拼接）到后续的*decompress()* 方法调用以获得正确的输出。

`Decompress.`**`eof`**
一个布尔值，指明是否已到达压缩数据流的末尾。

这使得区分正确构造的压缩数据流和不完整或被截断的压缩数据流成为可能。

3.3 版新加入.

Decompress.**decompress**(*data*, *max_length=0*)
解压缩 *data* 并返回 bytes 对象，其中包含对应于 *string* 中至少一部分数据的解压缩数据。此数据应当被拼接到之前任何对*decompress()* 方法的调用所产生的输出。部分输入数据可能会被保留在内部缓冲区以供后续处理。

如果可选的形参 *max_length* 非零则返回值将不会长于 *max_length*。这可能意味着不是所有已压缩输入都能被处理；并且未被消耗的数据将被保存在*unconsumed_tail* 属性中。如果要继续解压缩则这个字节串必须被传给对*decompress()* 的后续调用。如果 *max_length* 为零则整个输入都会被解压缩，并且*unconsumed_tail* 将为空。

3.6 版更變: *max_length* 可用作关键字参数。

Decompress.**flush**([*length*])
所有挂起的输入会被处理，并且返回包含剩余未压缩输出的 bytes 对象。在调用*flush()* 之后，*decompress()* 方法将无法被再次调用；唯一可行的操作是删除该对象。

可选的形参 *length* 设置输出缓冲区的初始大小。

Decompress.**copy**()
返回解压缩对象的一个拷贝。它可以用来在数据流的中途保存解压缩器的状态以便加快随机查找数据流后续位置的速度。

3.8 版更變: 添加了对解压缩对象执行*copy.copy()* 和*copy.deepcopy()* 的支持。

通过下列常量可获取模块所使用的 zlib 库的版本信息：

zlib.**ZLIB_VERSION**
构建此模块时所用的 zlib 库的版本字符串。它的值可能与运行时所加载的 zlib 不同。运行时加载的 zlib 库的版本字符串为*ZLIB_RUNTIME_VERSION*。

zlib.**ZLIB_RUNTIME_VERSION**
解释器所加载的 zlib 库的版本字符串。

3.3 版新加入.

也参考:

模块*gzip* 读写 **gzip** 格式的文件。

http://www.zlib.net zlib 库项目主页。

http://www.zlib.net/manual.html zlib 库用户手册。提供了库的许多功能的解释和用法。

13.2 gzip --- 对 gzip 格式的支持

源代码： Lib/gzip.py

此模块提供的简单接口帮助用户压缩和解压缩文件，功能类似于 GNU 应用程序 **gzip** 和 **gunzip**。

数据压缩由*zlib* 模块提供。

gzip 模块提供*GzipFile* 类和*open()*、*compress()*、*decompress()* 几个便利的函数。*GzipFile* 类可以读写 **gzip** 格式的文件，还能自动压缩和解压缩数据，这让操作压缩文件如同操作普通的*file object* 一样方便。

注意，此模块不支持部分可以被 **gzip** 和 **gunzip** 解压的格式，如利用 **compress** 或 **pack** 压缩所得的文件。

这个模块定义了以下内容：

gzip.**open**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)
以二进制方式或者文本方式打开一个 gzip 格式的压缩文件，返回一个*file object*。

filename 参数可以是一个实际的文件名 (一个 a *str* 对象或者*bytes* 对象), 或者是一个用来读写的已存在的文件对象。

mode 参数可以是二进制模式: `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'` or `'xb'`, 或者是文本模式 `'rt'`, `'at'`, `'wt'`, or `'xt'`。默认值是 `'rb'`。

The *compresslevel* argument is an integer from 0 to 9, as for the *GzipFile* constructor.

对于二进制模式，这个函数等价于*GzipFile* 构造器: `GzipFile(filename, mode, compresslevel)`。在这个例子中，*encoding*, *errors* 和 *newline* 三个参数一定不要设置。

对于文本模式，将会创建一个*GzipFile* 对象，并将它封装到一个*io.TextIOWrapper* 实例中，这个实例默认了指定编码，错误抓获行为和行。

3.3 版更變: 支持 *filename* 为一个文件对象，支持文本模式和 *encoding*, *errors* 和 *newline* 参数。

3.4 版更變: 支持 `'x'`, `'xb'` 和 “xt“ 三种模式。

3.6 版更變: 接受一个*path-like object*。

exception gzip.**BadGzipFile**
针对无效 gzip 文件引发的异常。它继承自*OSError*。针对无效 gzip 文件也可能引发*EOFError* 和*zlib.error*。

3.8 版新加入.

class gzip.**GzipFile**(*filename=None*, *mode=None*, *compresslevel=9*, *fileobj=None*, *mtime=None*)
GzipFile 类的构造器支持 `truncate()` 的异常, 与*file object* 的大多数方法非常相似。*fileobj* 和 *filename* 至少有一个不为空。

新的实例基于 *fileobj*，它可以是一个普通文件，一个*io.BytesIO* 对象，或者任何一个与文件相似的对象。当 *filename* 是一个文件对象时，它的默认值是 `None`。

当 *fileobj* 为 `None` 时，*filename* 参数只用于 **gzip** 文件头中，这个文件有可能包含未压缩文件的源文件名。如果文件可以被识别，默认 *fileobj* 的文件名；否则默认为空字符串，在这种情况下文件头将不包含源文件名。

mode 参数可以是 `'r'`, `'rb'`, `'a'`, `'ab'`, `'w'`, `'wb'`, `'x'` 或 `'xb'` 中的一个，具体取决于文件将被读取还是被写入。如果可识别则默认为 *fileobj* 的模式；否则默认为 `'rb'`。在未来的 Python 发布版中将不再使用 *fileobj* 的模式。最好总是指定 *mode* 为写入模式。

需要注意的是，文件默认使用二进制模式打开。如果要以文本模式打开文件一个压缩文件，请使用*open()* 方法 (或者使用*io.TextIOWrapper* 包装*GzipFile*)。

compresslevel 参数是一个从 `0` 到 `9` 的整数，用于控制压缩等级；`1` 最快但压缩比例最小，`9` 最慢但压缩比例最大。`0` 不压缩。默认为 `9`。

mtime 参数是一个可选的数字时间戳用于写入流的最后修改字段，。*mtime* 只在压缩模式中使用。如果省略或者值为 `None`，则使用当前时间。更多细节，详见*mtime* 属性。

调用*GzipFile* 的 `close()` 方法不会关闭 *fileobj*，因为你可以希望增加其它内容到已经压缩的数中。你可以将一个*io.BytesIO* 对象作为 *fileobj*，也可以使用*io.BytesIO* 的*getvalue()* 方法从内存缓存中恢复数据。

GzipFile 支持*io.BufferedIOBase* 类的接口, 包括迭代和 `with` 语句。只有 `truncate()` 方法没有实现。

GzipFile 还提供了以下的方法和属性:

peek(*n*)
在不移动文件指针的情况下读取 *n* 个未压缩字节。最多只有一个单独的读取流来服务这个方法调用。返回的字节数不一定刚好等于要求的数量。

備𝔽: 调用*peek()* 并没有改变*GzipFile* 的文件指针，它可能改变潜在文件对象 (例如: *GzipFile* 使用 *fileobj* 参数进行初始化)。

3.2 版新加入.

mtime
在解压的过程中，最后修改时间字段的值可能来自于这个属性，以整数的形式出现。在读取任何文件头信息前，初始值为 None。

所有 **gzip** 东方压缩流中必须包含时间戳这个字段。以便于像 **gunzip** 这样的程序可以使用时间戳。格式与*time.time()* 的返回值和*os.stat()* 对象的*st_mtime* 属性值一样。

3.1 版更變: 支持 with 语句，构造器参数 *mtime* 和*mtime* 属性。

3.2 版更變: 添加了对零填充和不可搜索文件的支持。

3.3 版更變: 实现*io.BufferedIOBase.read1()* 方法。

3.4 版更變: 支持 'x' and 'xb' 两种模式。

3.5 版更變: 支持写入任意*bytes-like objects*。*read()* 方法可以接受 “None“为参数。

3.6 版更變: 接受一个*path-like object*。

3.9 版後已𝔽用: 打开*GzipFile* 用于写入而不指定 *mode* 参数的做法已被弃用。

gzip.**compress**(*data*, *compresslevel=9*, *, *mtime=None*)
压缩 *data*，返回一个包含压缩数据的*bytes* 对象。*compresslevel* 和 *mtime* 的含义与上文中*GzipFile* 构造器的相同。

3.2 版新加入.

3.8 版更變: 添加了 *mtime* 形参用于可重复的输出。

gzip.**decompress**(*data*)
解压缩 *data*，返回一个包含未压缩数据的*bytes* 对象。

3.2 版新加入.

13.2.1 用法示例

读取压缩文件示例:

```
import gzip
with gzip.open('/home/joe/file.txt.gz', 'rb') as f:
    file_content = f.read()
```

创建 GZIP 文件示例:

```
import gzip
content = b"Lots of content here"
with gzip.open('/home/joe/file.txt.gz', 'wb') as f:
    f.write(content)
```

使用 GZIP 压缩已有的文件示例:

```
import gzip
import shutil
with open('/home/joe/file.txt', 'rb') as f_in:
    with gzip.open('/home/joe/file.txt.gz', 'wb') as f_out:
        shutil.copyfileobj(f_in, f_out)
```

使用 GZIP 压缩二进制字符串示例：

```
import gzip
s_in = b"Lots of content here"
s_out = gzip.compress(s_in)
```

也参考：

模块*zlib* 支持 **gzip** 格式所需要的基本压缩模块。

13.2.2 命令行界面

gzip 模块提供了简单的命令行界面用于压缩和解压缩文件。

在执行后*gzip* 模块会保留输入文件。

3.8 版更變: 添加一个带有用法说明的新命令行界面命令。默认情况下，当你要执行 CLI 时，默认压缩等级为 6。

命令行选项

file
如果 *file* 未指定，则从*sys.stdin* 读取。

--fast
指明最快速的压缩方法（较低压缩率）。

--best
指明最慢速的压缩方法（最高压缩率）。

-d, --decompress
解压缩给定的文件。

-h, --help
显示帮助消息。

13.3 bz2 --- 对 bzip2 压缩算法的支持

源代码： Lib/bz2.py

此模块提供了使用 bzip2 压缩算法压缩和解压数据的一套完整的接口。

bz2 模块包含：

- 用于读写压缩文件的*open()* 函数和*BZ2File* 类。
- 用于增量压缩和解压的*BZ2Compressor* 和*BZ2Decompressor* 类。
- 用于一次性压缩和解压的*compress()* 和*decompress()* 函数。

此模块中的所有类都能安全地从多个线程访问。

13.3.1 文件压缩和解压

bz2.**open**(*filename*, *mode='rb'*, *compresslevel=9*, *encoding=None*, *errors=None*, *newline=None*)
以二进制或文本模式打开 bzip2 压缩文件，返回一个*file object*。

和*BZ2File*的构造函数类似，*filename* 参数可以是一个实际的文件名（*str* 或*bytes* 对象），或是已有的可供读取或写入的文件对象。

mode 参数可设为二进制模式的 `'r'`、`'rb'`、`'w'`、`'wb'`、`'x'`、`'xb'`、`'a'` 或 `'ab'`，或者文本模式的 `'rt'`、`'wt'`、`'xt'` 或 `'at'`。默认是 `'rb'`。

compresslevel 参数是 1 到 9 的整数，和*BZ2File*的构造函数一样。

对于二进制模式，这个函数等价于*BZ2File* 构造器：`BZ2File(filename, mode, compresslevel=compresslevel)`。在这种情况下，不可提供 *encoding*, *errors* 和 *newline* 参数。

对于文本模式，将会创建一个*BZ2File*对象，并将它包装到一个*io.TextIOWrapper*实例中，此实例带有指定的编码格式、错误处理行为和行结束符。

3.3 版新加入.

3.4 版更變: 添加了 `'x'` (单独创建) 模式。

3.6 版更變: 接受一个类路径对象。

class bz2.**BZ2File**(*filename*, *mode='r'*, ***, *compresslevel=9*)
用二进制模式打开 bzip2 压缩文件。

如果 *filename* 是一个*str* 或*bytes* 对象，则打开名称对应的文件目录。否则的话，*filename* 应当是一个*file object*，它将被用来读取或写入压缩数据。

mode 参数可以是表示读取的 `'r'` (默认值)，表示覆写的 `'w'`，表示单独创建的 `'x'`，或表示添加的 `'a'`。这些模式还可分别以 `'rb'`, `'wb'`, `'xb'` 和 `'ab'` 的等价形式给出。

如果 *filename* 是一个文件对象（而不是实际的文件名），则 `'w'` 模式并不会截断文件，而是会等价于 `'a'`。

如果 *mode* 为 `'w'` 或 `'a'`，则 *compresslevel* 可以是 `1` 到 `9` 之间的整数，用于指定压缩等级: `1` 产生最低压缩率，而 `9` (默认值) 产生最高压缩率。

如果 *mode* 为 `'r'`，则输入文件可以为多个压缩流的拼接。

*BZ2File*提供了*io.BufferedIOBase*所指定的所有成员，但 `detach()` 和 `truncate()` 除外。并支持迭代和 `with` 语句。

*BZ2File*还提供了以下方法：

peek([*n*])
返回缓冲的数据而不前移文件位置。至少将返回一个字节的数据（除非为 EOF）。实际返回的字节数不确定。

備𝔽: 虽然调用*peek()* 不会改变*BZ2File*的文件位置，但它可能改变下层文件对象的位置（举例来说如果*BZ2File*是通过传入一个文件对象作为 *filename* 的话）。

3.3 版新加入.

3.1 版更變: 添加了对 `with` 语句的支持。

3.3 版更變: 添加了 `fileno()`, `readable()`, `seekable()`, `writable()`, `read1()` 和 `readinto()` 方法。

3.3 版更變: 添加了对 *filename* 使用*file object* 而非实际文件名的支持。

3.3 版更變: 添加了 `'a'` (append) 模式，以及对读取多数据流文件的支持。

3.4 版更變: 添加了 `'x'` (单独创建) 模式。

3.5 版更變: *read()* 方法现在接受 `None` 作为参数。

3.6 版更變: 接受一个类路径对象。

3.9 版更變: *buffering* 形参已被移除。它自 Python 3.0 起即被忽略并弃用。请传入一个打开文件对象来控制文件的打开方式。

compresslevel 形参成为仅限关键字参数。

13.3.2 增量压缩和解压

class bz2.**BZ2Compressor**(*compresslevel=9*)

创建一个新的压缩器对象。此对象可被用来执行增量数据压缩。对于一次性压缩，请改用*compress()* 函数。

如果给定 *compresslevel*，它必须为 `1` 至 `9` 之间的整数。默认值为 `9`。

compress(*data*)

向压缩器对象提供数据。在可能的情况下返回一段已压缩数据，否则返回空字节串。

当你已结束向压缩器提供数据时，请调用*flush()* 方法来完成压缩进程。

flush()

结束压缩进程，返回内部缓冲中剩余的压缩完成的数据。

调用此方法之后压缩器对象将不可再被使用。

class bz2.**BZ2Decompressor**

创建一个新的解压缩器对象。此对象可被用来执行增量数据解压缩。对于一次性解压缩，请改用*decompress()* 函数。

備𝔽: 这个类不会透明地处理包含多个已压缩数据流的输入，这不同于*decompress()* 和*BZ2File*。如果你需要通过*BZ2Decompressor* 来解压缩多个数据流输入，你必须为每个数据流都使用新的解压缩器。

---

decompress(*data*, *max_length=-1*)

解压缩 *data* (一个*bytes-like object*)，返回字节串形式的解压缩数据。某些 *data* 可以在内部被缓冲，以便用于后续的*decompress()* 调用。返回的数据应当与之前任何*decompress()* 调用的输出进行拼接。

如果 *max_length* 为非负数，将返回至多 *max_length* 个字节的解压缩数据。如果达到此限制并且可以产生后续输出，则*needs_input* 属性将被设为 `False`。在这种情况下，下一次*decompress()* 调用提供的 *data* 可以为 `b''` 以获取更多的输出。

如果所有输入数据都已被解压缩并返回（或是因为它少于 *max_length* 个字节，或是因为 *max_length* 为负数），则*needs_input* 属性将被设为 `True`。

在到达数据流末尾之后再尝试解压缩数据会引发 *EOFError*。在数据流末尾之后获取的任何数据都会被忽略并存储至*unused_data* 属性。

3.5 版更變: 添加了 *max_length* 形参。

eof
若达到了数据流的末尾标记则为 `True`。

3.3 版新加入.

unused_data
在压缩数据流的末尾之后获取的数据。

如果在达到数据流末尾之前访问此属性，其值将为 `b''`。

needs_input
如果在要求新的未解压缩输入之前*decompress()* 方法可以提供更多的解压缩数据则为 `False`。

3.5 版新加入.

13.3.3 一次性压缩或解压缩

`bz2.`**compress**(*data*, *compresslevel=9*)
压缩 *data*，此参数为一个字节类对象。

如果给定 *compresslevel*，它必须为 `1` 至 `9` 之间的整数。默认值为 `9`。

对于增量压缩，请改用*BZ2Compressor*。

`bz2.`**decompress**(*data*)
解压缩 *data*，此参数为一个字节类对象。

如果 *data* 是多个压缩数据流的拼接，则解压缩所有数据流。

对于增量解压缩，请改用*BZ2Decompressor*。

3.3 版更變: 支持了多数据流的输入。

13.3.4 用法示例

以下是*bz2* 模块典型用法的一些示例。

使用*compress()* 和*decompress()* 来显示往复式的压缩：

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> c = bz2.compress(data)
>>> len(data) / len(c)  # Data compression ratio
1.513595166163142
>>> d = bz2.decompress(c)
>>> data == d  # Check equality to original object after round-trip
True
```

使用*BZ2Compressor* 进行增量压缩：

```
>>> import bz2
>>> def gen_data(chunks=10, chunksize=1000):
...     """Yield incremental blocks of chunksize bytes."""
...     for _ in range(chunks):
...         yield b"z" * chunksize
...
>>> comp = bz2.BZ2Compressor()
>>> out = b""
>>> for chunk in gen_data():
...     # Provide data to the compressor object
...     out = out + comp.compress(chunk)
...
>>> # Finish the compression process.  Call this once you have
>>> # finished providing data to the compressor.
>>> out = out + comp.flush()
```

上面的示例使用了十分“非随机”的数据流（即 *b"z"* 块数据流）。随机数据的压缩率通常很差，而有序、重复的数据通常会产生很高的压缩率。

用二进制模式写入和读取 bzip2 压缩文件：

```
>>> import bz2
>>> data = b"""\
... Donec rhoncus quis sapien sit amet molestie. Fusce scelerisque vel augue
... nec ullamcorper. Nam rutrum pretium placerat. Aliquam vel tristique lorem,
... sit amet cursus ante. In interdum laoreet mi, sit amet ultrices purus
... pulvinar a. Nam gravida euismod magna, non varius justo tincidunt feugiat.
... Aliquam pharetra lacus non risus vehicula rutrum. Maecenas aliquam leo
... felis. Pellentesque semper nunc sit amet nibh ullamcorper, ac elementum
... dolor luctus. Curabitur lacinia mi ornare consectetur vestibulum."""
>>> with bz2.open("myfile.bz2", "wb") as f:
...     # Write compressed data to file
...     unused = f.write(data)
>>> with bz2.open("myfile.bz2", "rb") as f:
...     # Decompress data from file
...     content = f.read()
>>> content == data  # Check equality to original object after round-trip
True
```

13.4 lzma --- 用 LZMA 算法压缩

3.3 版新加入.

源代码： Lib/lzma.py

此模块提供了可以压缩和解压缩使用 LZMA 压缩算法的数据的类和便携函数。其中还包含支持 **xz** 工具所使用的 `.xz` 和旧式 `.lzma` 文件格式的文件接口，以及相应的原始压缩数据流。

此模块所提供的接口与*bz2*模块的非常类似。但是，请注意*LZMAFile* 不是线程安全的，这与*bz2.BZ2File* 不同，因此如果你需要在多个线程中使用单个*LZMAFile* 实例，则需要通过锁来保护它。

exception lzma.**LZMAError**

当在压缩或解压缩期间或是在初始化压缩器/解压缩器的状态期间发生错误时此异常会被引发。

13.4.1 读写压缩文件

lzma.**open**(*filename*, *mode="rb"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*, *encoding=None*, *errors=None*, *newline=None*)

以二进制或文本模式打开 LZMA 压缩文件，返回一个*file object*。

filename 参数可以是一个实际的文件名（以*str*, *bytes* 或*路径类* 对象的形式给出），在此情况下会打开指定名称的文件，或者可以是一个用于读写的现有文件对象。

mode 参数可以是二进制模式的 `"r"`, `"rb"`, `"w"`, `"wb"`, `"x"`, `"xb"`, `"a"` 或 `"ab"`，或者文本模式的 `"rt"`, `"wt"`, `"xt"` 或 `"at"`。默认值为 `"rb"`。

当打开一个文件用于读取时，*format* 和 *filters* 参数具有与*LZMADecompressor* 的参数相同的含义。在此情况下，*check* 和 *preset* 参数不应被使用。

当打开一个文件用于写入的，*format*, *check*, *preset* 和 *filters* 参数具有与*LZMACompressor* 的参数相同的含义。

对于二进制模式，这个函数等价于*LZMAFile* 构造器: `LZMAFile(filename, mode, ...)`。在这种情况下，不可提供 *encoding*, *errors* 和 *newline* 参数。

对于文本模式，将会创建一个*LZMAFile* 对象，并将它包装到一个*io.TextIOWrapper* 实例中，此实例带有指定的编码格式、错误处理行为和行结束符。

3.4 版更變: 增加了对 `"x"`, `"xb"` 和 `"xt"` 模式的支持。

3.6 版更變: 接受一个*类路径对象*。

class lzma.**LZMAFile**(*filename=None*, *mode="r"*, ***, *format=None*, *check=-1*, *preset=None*, *filters=None*)

以二进制模式打开一个 LZMA 压缩文件。

LZMAFile 可以包装在一个已打开的*file object* 中，或者是在给定名称的文件上直接操作。*filename* 参数指定所包装的文件对象，或是要打开的文件名称（类型为*str*, *bytes* 或*路径类* 对象）。如果是包装现有的文件对象，被包装的文件在*LZMAFile* 被关闭时将不会被关闭。

mode 参数可以是表示读取的 `"r"` (默认值)，表示覆写的 `"w"`，表示单独创建的 `"x"`，或表示添加的 `"a"`。这些模式还可以分别以 `"rb"`, `"wb"`, `"xb"` 和 `"ab"` 的等价形式给出。

如果 *filename* 是一个文件对象（而不是实际的文件名），则 `"w"` 模式并不会截断文件，而会等价于 `"a"`。

当打开一个文件用于读取时，输入文件可以为多个独立压缩流的拼接。它们会被作为单个逻辑流被透明地解码。

当打开一个文件用于读取时，*format* 和 *filters* 参数具有与*LZMADecompressor* 的参数相同的含义。在此情况下，*check* 和 *preset* 参数不应被使用。

当打开一个文件用于写入的，*format*, *check*, *preset* 和 *filters* 参数具有与*LZMACompressor* 的参数相同的含义。

LZMAFile 支持*io.BufferedIOBase* 所指定的所有成员，但 `detach()` 和 `truncate()` 除外。并支持迭代和 `with` 语句。

也提供以下方法：

peek(*size=-1*)

返回缓冲的数据而不前移文件位置。至少将返回一个字节的数据，除非已经到达 EOF。实际返回的字节数不确定（会忽略 *size* 参数）。

備🄴: 虽然调用*peek()* 不会改变*LZMAFile* 的文件位置，但它可能改变下层文件对象的位置（举例来说如果*LZMAFile* 是通过传入一个文件对象作为 *filename* 的话）。

3.4 版更變: 增加了对 "x" 和 "xb" 模式的支持。

3.5 版更變: *read()* 方法现在接受 None 作为参数。

3.6 版更變: 接受一个类路径对象。

13.4.2 在内存中压缩和解压缩数据

class lzma.**LZMACompressor**(*format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)

创建一个压缩器对象，此对象可被用来执行增量压缩。

压缩单个数据块的更便捷方式请参阅*compress()*。

format 参数指定应当使用哪种容器格式。可能的值有:

- **FORMAT_XZ: .xz 容器格式**。 这是默认格式。
- **FORMAT_ALONE: 传统的 .lzma 容器格式**。 这种格式相比 .xz 更为受限 -- 它不支持一致性检查或多重过滤器。
- **FORMAT_RAW: 原始数据流，不使用任何容器格式**。 这个格式描述器不支持一致性检查，并且要求你必须指定一个自定义的过滤器链（用于压缩和解压缩）。此外，以这种方式压缩的数据不可使用 FORMAT_AUTO 来解压缩 (参见*LZMADecompressor*)。

check 参数指定要包含在压缩数据中的一致性检查类型。这种检查在解压缩时使用，以确保数据没有被破坏。可能的值是:

- CHECK_NONE: 没有一致性检查。这是 FORMAT_ALONE 和 FORMAT_RAW 的默认值（也是唯一可接受的值）。
- CHECK_CRC32: 32 位循环冗余检查。
- CHECK_CRC64: 64 位循环冗余检查。这是 FORMAT_XZ 的默认值。
- CHECK_SHA256: 256 位安全哈希算法。

如果指定的检查不受支持，则会引发*LZMAError*。

压缩设置可被指定为一个预设的压缩等级（通过 *preset* 参数）或以自定义过滤器链来详细设置（通过 *filters* 参数）。

preset 参数（如果提供）应当为一个 0 到 9 (包括边界) 之间的整数，可以选择与常数 PRESET_EXTREME 进行 OR 运算。如果 *preset* 和 *filters* 均未给出，则默认行为是使用 PRESET_DEFAULT (预设等级 6)。更高的预设等级会产生更小的输出，但会使得压缩过程更缓慢。

備F: 除了更加 CPU 密集，使用更高的预设等级来压缩还需要更多的内存（并产生需要更多内存来解压缩的输出）。例如使用预设等级 9 时，一个*LZMACompressor* 对象的开销可以高达 800 MiB。出于这样的原因，通常最好是保持使用默认预设等级。

---

filters 参数（如果提供）应当指定一个过滤器链。详情参见指定自定义的过滤器链。

compress(*data*)

压缩 *data* (一个*bytes* object)，返回包含针对输入的至少一部分已压缩数据的*bytes* 对象。一部 *data* 可能会被放入内部缓冲区，以便用于后续的*compress()* 和*flush()* 调用。返回的数据应当与之前任何*compress()* 调用的输出进行拼接。

flush()

结束压缩进程，返回包含保存在压缩器的内部缓冲区中的任意数据的*bytes* 对象。

调用此方法之后压缩器将不可再被使用。

class lzma.**LZMADecompressor**(*format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)
创建一个压缩器对象，此对象可被用来执行增量解压缩。

一次性解压缩整个压缩数据流的更便捷方式请参阅*decompress()*。

format 参数指定应当被使用的容器格式。默认值为 FORMAT_AUTO，它可以解压缩 .xz 和 .lzma 文件。其他可能的值为 FORMAT_XZ, FORMAT_ALONE 和 FORMAT_RAW。

memlimit 参数指定解压缩器可以使用的内存上限（字节数）。当使用此参数时，如果不可能在给定内存上限之内解压缩输入数据则解压缩将失败并引发*LZMAError*。

filters 参数指定用于创建被解压缩数据流的过滤器链。此参数在 *format* 为 FORMAT_RAW 时要求提供，但对于其他格式不应使用。有关过滤器链的更多信息请参阅指定自定义的过滤器链。

備𝔽: 这个类不会透明地处理包含多个已压缩数据流的输入，这不同于*decompress()* 和*LZMAFile*。要通过*LZMADecompressor* 来解压缩多个数据流输入，你必须为每个数据流都创建一个新的解压缩器。

---

decompress(*data*, *max_length=-1*)
解压缩 *data* (一个*bytes-like object*)，返回字节串形式的解压缩数据。某些 *data* 可以在内部被缓冲，以便用于后续的*decompress()* 调用。返回的数据应当与之前任何*decompress()* 调用的输出进行拼接。

如果 *max_length* 为非负数，将返回至多 *max_length* 个字节的解压缩数据。如果达到此限制并且可以产生后续输出，则*needs_input* 属性将被设为 False。在这种情况下，下一次*decompress()* 调用提供的 *data* 可以为 b'' 以获取更多的输出。

如果所有输入数据都已被解压缩并返回（或是因为它少于 *max_length* 个字节，或是因为 *max_length* 为负数），则*needs_input* 属性将被设为 True。

在到达数据流末尾之后再尝试解压缩数据会引发 *EOFError*。在数据流末尾之后获取的任何数据都会被忽略并存储至*unused_data* 属性。

3.5 版更變: 添加了 *max_length* 形参。

check
输入流使用的一致性检查的 ID。这可能为 CHECK_UNKNOWN 直到已解压了足够的输入数据来确定它所使用的一致性检查。

eof
若达到了数据流末尾标识符则为 True。

unused_data
压缩数据流的末尾还有数据。

在达到数据流末尾之前，这个值将为 b""。

needs_input
如果在要求新的未解压缩输入之前*decompress()* 方法可以提供更多的解压缩数据则为 False。

3.5 版新加入.

lzma.**compress**(*data*, *format=FORMAT_XZ*, *check=-1*, *preset=None*, *filters=None*)
压缩 *data* (一个*bytes* 对象)，返回包含压缩数据的*bytes* 对象。

参见上文的*LZMACompressor* 了解有关 *format*, *check*, *preset* 和 *filters* 参数的说明。

lzma.**decompress**(*data*, *format=FORMAT_AUTO*, *memlimit=None*, *filters=None*)
解压缩 *data* (一个*bytes* 对象)，返回包含解压缩数据的*bytes* 对象。

如果 *data* 是多个单独压缩数据流的拼接，则解压缩所有相应数据流，并返回结果的拼接。

参见上文的*LZMADecompressor* 了解有关 *format*, *memlimit* 和 *filters* 参数的说明。

13.4.3 杂项

lzma.**is_check_supported**(*check*)
如果本系统支持给定的一致性检查则返回 True。

CHECK_NONE 和 CHECK_CRC32 总是受支持。CHECK_CRC64 和 CHECK_SHA256 或许不可用，如果你正在使用基于受限制特性集编译的 **liblzma** 版本的话。

13.4.4 指定自定义的过滤器链

过滤器链描述符是由字典组成的序列，其中每个字典包含单个过滤器的 ID 和选项。每个字典必须包含键 "id"，并可能包含额外的键用来指定基于过滤器的选项。有效的过滤器 ID 如下：

- **压缩过滤器：**
  - FILTER_LZMA1 (配合 FORMAT_ALONE 使用)
  - FILTER_LZMA2 (配合 FORMAT_XZ 和 FORMAT_RAW 使用)
- **Delta 过滤器：**
  - FILTER_DELTA
- **Branch-Call-Jump (BCJ) 过滤器：**
  - FILTER_X86
  - FILTER_IA64
  - FILTER_ARM
  - FILTER_ARMTHUMB
  - FILTER_POWERPC
  - FILTER_SPARC

一个过滤器链最多可由 4 个过滤器组成，并且不能为空。过滤器链中的最后一个过滤器必须为压缩过滤器，其他过滤器必须为 Delta 或 BCJ 过滤器。

压缩过滤器支持下列选项（指定为表示过滤器的字典中的附加条目）：

- preset: 压缩预设选项，用于作为未显式指定的选项的默认值的来源。
- dict_size: 以字节表示的字典大小。这应当在 4 KiB 和 1.5 GiB 之间（包含边界）。
- lc: 字面值上下文的比特数。
- lp: 字面值位置的比特数。总计值 lc + lp 必须不大于 4。
- pb: 位置的比特数；必须不大于 4。
- mode: MODE_FAST 或 MODE_NORMAL。
- nice_len: 对于一个匹配应当被视为“适宜长度”的值。这应当小于或等于 273。
- mf: 要使用的匹配查找器 -- MF_HC3, MF_HC4, MF_BT2, MF_BT3 或 MF_BT4。
- depth: 匹配查找器使用的最大查找深度。0 (默认值) 表示基于其他过滤器选项自动选择。

Delta 过滤器保存字节数据之间的差值，在特定环境下可产生更具重复性的输入。它支持一个 `dist` 选项，指明要减去的字节之间的差值大小。默认值为 1，即相邻字节之间的差值。

BCJ 过滤器主要作用于机器码。它们会转换机器码内的相对分支、调用和跳转以使用绝对寻址，其目标是提升冗余度以供压缩器利用。这些过滤器支持一个 `start_offset` 选项，指明应当被映射到输入数据开头的地址。默认值为 0。

13.4.5 示例

在已压缩的数据中读取:

```
import lzma
with lzma.open("file.xz") as f:
    file_content = f.read()
```

创建一个压缩文件:

```
import lzma
data = b"Insert Data Here"
with lzma.open("file.xz", "w") as f:
    f.write(data)
```

在内存中压缩文件:

```
import lzma
data_in = b"Insert Data Here"
data_out = lzma.compress(data_in)
```

增量压缩:

```
import lzma
lzc = lzma.LZMACompressor()
out1 = lzc.compress(b"Some data\n")
out2 = lzc.compress(b"Another piece of data\n")
out3 = lzc.compress(b"Even more data\n")
out4 = lzc.flush()
# Concatenate all the partial results:
result = b"".join([out1, out2, out3, out4])
```

写入已压缩数据到已打开的文件:

```
import lzma
with open("file.xz", "wb") as f:
    f.write(b"This data will not be compressed\n")
    with lzma.open(f, "w") as lzf:
        lzf.write(b"This *will* be compressed\n")
    f.write(b"Not compressed\n")
```

使用自定义过滤器链创建一个已压缩文件:

```
import lzma
my_filters = [
    {"id": lzma.FILTER_DELTA, "dist": 5},
    {"id": lzma.FILTER_LZMA2, "preset": 7 | lzma.PRESET_EXTREME},
]
with lzma.open("file.xz", "w", filters=my_filters) as f:
    f.write(b"blah blah blah")
```

13.5 zipfile --- 使用 ZIP 存档

源代码： Lib/zipfile.py

ZIP 文件格式是一个常用的归档与压缩标准。这个模块提供了创建、读取、写入、添加及列出 ZIP 文件的工具。任何对此模块的进阶使用都将需要理解此格式，其定义参见 PKZIP 应用程序笔记。

此模块目前不能处理分卷 ZIP 文件。它可以处理使用 ZIP64 扩展（超过 4 GB 的 ZIP 文件）的 ZIP 文件。它支持解密 ZIP 归档中的加密文件，但是目前不能创建一个加密的文件。解密非常慢，因为它是使用原生 Python 而不是 C 实现的。

这个模块定义了以下内容：

exception zipfile.**BadZipFile**
为损坏的 ZIP 文件抛出的错误。

3.2 版新加入.

exception zipfile.**BadZipfile**
BadZipFile 的别名，与旧版本 Python 保持兼容性。

3.2 版後已🄴用.

exception zipfile.**LargeZipFile**
当 ZIP 文件需要 ZIP64 功能但是未启用时会抛出此错误。

class zipfile.**ZipFile**
用于读写 ZIP 文件的类。欲了解构造函数的描述，参阅段落*ZipFile 对象*。

class zipfile.**Path**
用于 zip 文件的兼容 pathlib 的包装器。详情参见*Path 对象*。

3.8 版新加入.

class zipfile.**PyZipFile**
用于创建包含 Python 库的 ZIP 归档的类。

class zipfile.**ZipInfo**(*filename='NoName'*, *date_time=(1980, 1, 1, 0, 0, 0)*)
用于表示档案内一个成员信息的类。此类的实例会由 *ZipFile* 对象的 *getinfo()* 和 *infolist()* 方法返回。大多数 *zipfile* 模块的用户都不必创建它们，只需使用此模块所创建的实例。*filename* 应当是档案成员的全名，*date_time* 应当是包含六个字段的描述最近修改时间的元组；这些字段的描述请参阅*ZipInfo 对象*。

zipfile.**is_zipfile**(*filename*)
根据文件的 Magic Number，如果 *filename* 是一个有效的 ZIP 文件则返回 `True`，否则返回 `False`。*filename* 也可能是一个文件或类文件对象。

3.1 版更變: 支持文件或类文件对象。

zipfile.**ZIP_STORED**
未被压缩的归档成员的数字常数。

zipfile.**ZIP_DEFLATED**
常用的 ZIP 压缩方法的数字常数。需要 *zlib* 模块。

zipfile.**ZIP_BZIP2**
BZIP2 压缩方法的数字常数。需要 *bz2* 模块。

3.3 版新加入.

zipfile.**ZIP_LZMA**
LZMA 压缩方法的数字常数。需要*lzma* 模块。

3.3 版新加入.

備🄵: ZIP 文件格式规范包括自 2001 年以来对 bzip2 压缩的支持，以及自 2006 年以来对 LZMA 压缩的支持。但是，一些工具（包括较旧的 Python 版本）不支持这些压缩方法，并且可能拒绝完全处理 ZIP 文件，或者无法提取单个文件。

也参考:

PKZIP 应用程序笔记 Phil Katz 编写的 ZIP 文件格式文档，此格式和使用的算法的创建者。

Info-ZIP 主页 有关 Info-ZIP 项目的 ZIP 存档程序和开发库的信息。

13.5.1 ZipFile 对象

class zipfile.**ZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *compresslevel=None*, ***, *strict_timestamps=True*)
打开一个 ZIP 文件，*file* 为一个指向文件的路径（字符串），一个类文件对象或者一个*path-like object*。

形参 *mode* 应当为 `'r'` 来读取一个存在的文件，`'w'` 来截断并写入新的文件，`'a'` 来添加到一个存在的文件，或者 `'x'` 来仅新建并写入新的文件。如果 *mode* 为 `'x'` 并且 *file* 指向已经存在的文件，则抛出*FileExistsError*。如果 *mode* 为 `'a'` 且 *file* 为已存在的文件，则格外的文件将被加入。如果 *file* 不指向 ZIP 文件，之后一个新的 ZIP 归档将被追加为此文件。这是为了将 ZIP 归档添加到另一个文件（例如 `python.exe`）。如果 *mode* 为 `'a'` 并且文件不存在，则会新建。如果 *mode* 为 `'r'` 或 `'a'`，则文件应当可定位。

compression 是在写入归档时要使用的 ZIP 压缩方法，应为*ZIP_STORED*, *ZIP_DEFLATED*, *ZIP_BZIP2* 或*ZIP_LZMA*；不可识别的值将导致引发*NotImplementedError*。如果指定了*ZIP_DEFLATED*, *ZIP_BZIP2* 或*ZIP_LZMA* 但相应的模块 (*zlib*, *bz2* 或*lzma*) 不可用，则会引发*RuntimeError*。默认值为*ZIP_STORED*。

如果 *allowZip64* 为 `True` (默认值) 则当 zipfile 大于 4 GiB 时 zipfile 将创建使用 ZIP64 扩展的 ZIP 文件。如果该参数为 `false` 则当 ZIP 文件需要 ZIP64 扩展时*zipfile* 将引发异常。

compresslevel 形参控制在将文件写入归档时要使用的压缩等级。当使用*ZIP_STORED* 或*ZIP_LZMA* 时无压缩效果。当使用*ZIP_DEFLATED* 时接受整数 `0` 至 `9` (更多信息参见*zlib*)。当使用*ZIP_BZIP2* 时接受整数 `1` 至 `9` (更多信息参见*bz2*)。

strict_timestamps 参数在设为 `False` 时允许压缩早于 1980-01-01 的文件，代价时会将时间戳设为 1980-01-01。类似的行为也会对晚于 2107-12-31 的文件发生，时间戳也会被设为该上限值。

如果创建文件时使用 `'w'`, `'x'` 或 `'a'` 模式并且未向归档添加任何文件就执行了*closed*，则会将适当的空归档 ZIP 结构写入文件。

ZipFile 也是一个上下文管理器，因此支持 `with` 语句。在这个示例中，*myzip* 将在 `with` 语句块执行完成之后被关闭 --- 即使是发生了异常:

```
with ZipFile('spam.zip', 'w') as myzip:
    myzip.write('eggs.txt')
```

3.2 版新加入: 添加了将*ZipFile* 用作上下文管理员的功能。

3.3 版更變: 添加了对*bzip2* 和*lzma* 压缩的支持。

3.4 版更變: 默认启用 ZIP64 扩展。

3.5 版更變: 添加了对不可查找数据流的支持。并添加了对 `'x'` 模式的支持。

3.6 版更變: 在此之前，对于不可识别的压缩值将引发普通的*RuntimeError*。

3.6.2 版更變: *file* 形参接受一个*path-like object*。

3.7 版更變: 添加了 *compresslevel* 形参。

3.8 版新加入: *strict_timestamps* 仅限关键字参数

ZipFile.**close**()
关闭归档文件。你必须在退出程序之前调用*close()* 否则将不会写入关键记录数据。

ZipFile.**getinfo**(*name*)
返回一个*ZipInfo* 对象，其中包含有关归档成员 *name* 的信息。针对一个目前并不包含于归档中的名称调用*getinfo()* 将会引发*KeyError*。

ZipFile.**infolist**()
返回一个列表，其中包含每个归档成员的*ZipInfo* 对象。如果是打开一个现有归档则这些对象的排列顺序与它们对应条目在磁盘上的实际 ZIP 文件中的顺序一致。

ZipFile.**namelist**()
返回按名称排序的归档成员列表。

ZipFile.**open**(*name*, *mode='r'*, *pwd=None*, *, *force_zip64=False*)
以二进制文件类对象的形式访问一个归档成员。*name* 可以是归档内某个文件的名称也可以是某个*ZipInfo* 对象。如果包含了 *mode* 形参，则它必须为 `'r'` (默认值) 或 `'w'`。*pwd* 为用于解密已加密 ZIP 文件的密码。

open() 也是一个上下文管理器，因此支持 `with` 语句:

```
with ZipFile('spam.zip') as myzip:
    with myzip.open('eggs.txt') as myfile:
        print(myfile.read())
```

如果 *mode* 为 `'r'` 则文件类对象 (`ZipExtFile`) 将为只读并且提供下列方法: *read()*, *readline()*, *readlines()*, *seek()*, *tell()*, __iter__(), *__next__()*。这些对象可独立于 ZipFile 进行操作。

如果 `mode='w'` 则返回一个可写入的文件句柄，它将支持*write()* 方法。当一个可写入的文件句柄被打开时，尝试读写 ZIP 文件中的其他文件将会引发*ValueError*。

当写入一个文件时，如果文件大小不能预先确定但是可能超过 2 GiB，可传入 `force_zip64=True` 以确保标头格式能够支持超大文件。如果文件大小可以预先确定，则在构造*ZipInfo* 对象时应设置*file_size*，并将其用作 *name* 形参。

備F: *open()*, *read()* 和*extract()* 方法可接受文件名或*ZipInfo* 对象。当尝试读取一个包含重复名称成员的 ZIP 文件时你将发现此功能很有好处。

3.6 版更變: 移除了对 `mode='U'` 的支持。请使用*io.TextIOWrapper* 以在*universal newlines* 模式中读取已压缩的文本文件。

3.6 版更變: *ZipFile.open()* can now be used to write files into the archive with the `mode='w'` option.

3.6 版更變: 在已关闭的 ZipFile 上调用*open()* 将引发*ValueError*。在之前的版本中则会引发*RuntimeError*。

ZipFile.**extract**(*member*, *path=None*, *pwd=None*)
从归档中提取出一个成员放入当前工作目录；*member* 必须为成员的完整名称或*ZipInfo* 对象。成员的文件信息会尽可能精确地被提取。*path* 指定一个要提取到的不同目录。*member* 可以是一个文件名或*ZipInfo* 对象。*pwd* 是用于解密文件的密码。

返回所创建的经正规化的路径（对应于目录或新文件）。

備𝔽: 如果一个成员文件名为绝对路径，则将去掉驱动器/UNC 共享点和前导的（反）斜杠，例如: `///foo/bar` 在 Unix 上将变为 `foo/bar`，而 `C:\foo\bar` 在 Windows 上将变为 `foo\bar`。并且一个成员文件名中的所有 `".."` 都将被移除，例如: `../../foo../../ba..r` 将变为 `foo../ba..r`。在 Windows 上非法字符 (`:`, `<`, `>`, `|`, `"`, `?`, and `*`) 会被替换为下划线 (_)。

3.6 版更變: 在已关闭的 ZipFile 上调用*extract()* 将引发*ValueError*。在之前的版本中则将引发*RuntimeError*。

3.6.2 版更變: *path* 形参接受一个*path-like object*。

ZipFile.**extractall**(*path=None*, *members=None*, *pwd=None*)
从归档中提取出所有成员放入当前工作目录。*path* 指定一个要提取到的不同目录。*members* 为可选项且必须为*namelist()* 所返回列表的一个子集。*pwd* 是用于解密文件的密码。

警告: 绝不要未经预先检验就从不可靠的源中提取归档文件。这样有可能在 *path* 之外创建文件，例如某些成员具有以 `"/"` 开始的文件名或带有两个点号 `".."` 的文件名。此模块会尝试防止这种情况。参见*extract()* 的注释。

3.6 版更變: 在已关闭的 ZipFile 上调用*extractall()* 将引发*ValueError*。在之前的版本中则将引发*RuntimeError*。

3.6.2 版更變: *path* 形参接受一个*path-like object*。

ZipFile.**printdir**()
将归档的目录表打印到 `sys.stdout`。

ZipFile.**setpassword**(*pwd*)
设置 *pwd* 为用于提取已加密文件的默认密码。

ZipFile.**read**(*name*, *pwd=None*)
返回归档中文件 *name* 的字节数据。*name* 是归档中文件的名称，或是一个*ZipInfo* 对象。归档必须以读取或追加方式打开。*pwd* 为用于已加密文件的密码，并且如果指定该参数则它将覆盖通过*setpassword()* 设置的默认密码。on a ZipFile that uses a compression method 在使用*ZIP_STORED*，*ZIP_DEFLATED*, *ZIP_BZIP2* 或*ZIP_LZMA* 以外的压缩方法的 ZipFile 上调用*read()* 将引发*NotImplementedError*。如果相应的压缩模块不可用也会引发错误。

3.6 版更變: 在已关闭的 ZipFile 上调用*read()* 将引发*ValueError*。在之前的版本中则会引发*RuntimeError*。

ZipFile.**testzip**()
读取归档中的所有文件并检查它们的 CRC 和文件头。返回第一个已损坏文件的名称，在其他情况下则返回 `None`。

3.6 版更變: 在已关闭的 ZipFile 上调用*testzip()* 将引发*ValueError*。在之前的版本中则将引发*RuntimeError*。

ZipFile.**write**(*filename*, *arcname=None*, *compress_type=None*, *compresslevel=None*)
将名为 *filename* 的文件写入归档，给予的归档名为 *arcname* (默认情况下将与 *filename* 一致，但是不带驱动器盘符并会移除开头的路径分隔符)。*compress_type* 如果给出，它将覆盖作为构造器 *compression* 形参对于新条目所给出的值。类似地，*compresslevel* 如果给出也将覆盖构造器。归档必须使用 `'w'`, `'x'` 或 `'a'` 模式打开。

備𝔽: 归档名称应当是基于归档根目录的相对路径，也就是说，它们不应以路径分隔符开头。

備𝔽: 如果 arcname (或 filename，如果 arcname 未给出) 包含一个空字节，则归档中该文件的名称将在空字节位置被截断。

備𝔽: 文件名开头有一个斜杠可能导致存档文件无法在 Windows 系统上的某些 zip 程序中打开。

3.6 版更變: 在使用 'r' 模式创建的 ZipFile 或已关闭的 ZipFile 上调用*write()* 将引发*ValueError*。在之前的版本中则会引发*RuntimeError*。

ZipFile.**writestr**(*zinfo_or_arcname*, *data*, *compress_type=None*, *compresslevel=None*)
将一个文件写入归档。内容为 *data*，它可以是一个*str* 或*bytes* 的实例；如果是*str*，则会先使用 UTF-8 进行编码。*zinfo_or_arcname* 可以是它在归档中将被给予的名称，或者是*ZipInfo* 的实例。如果它是一个实例，则至少必须给定文件名、日期和时间。如果它是一个名称，则日期和时间会被设为当前日期和时间。归档必须以 'w', 'x' 或 'a' 模式打开。

如果给定了 *compress_type*，它将会覆盖作为新条目构造器的 *compression* 形参或在 *zinfo_or_arcname* (如果是一个*ZipInfo* 实例) 中所给出的值。类似地，如果给定了 *compresslevel*，它将会覆盖构造器。

備𝔽: 当传入一个*ZipInfo* 实例作为 *zinfo_or_arcname* 形参时，所使用的压缩方法将为在给定的*ZipInfo* 实例的 *compress_type* 成员中指定的方法。默认情况下，*ZipInfo* 构造器将将此成员设为*ZIP_STORED*。

3.2 版更變: *compress_type* 参数。

3.6 版更變: 在使用 'r' 模式创建的 ZipFile 或已关闭的 ZipFile 上调用*writestr()* 将引发*ValueError*。在之前的版本中则会引发*RuntimeError*。

以下数据属性也是可用的:

ZipFile.**filename**
ZIP 文件的名称。

ZipFile.**debug**
要使用的调试输出等级。这可以设为从 0 (默认无输出) 到 3 (最多输出) 的值。调试信息会被写入 sys.stdout。

ZipFile.**comment**
关联到 ZIP 文件的*bytes* 对象形式的说明。如果将说明赋给以 'w', 'x' 或 'a' 模式创建的*ZipFile* 实例，它的长度不应超过 65535 字节。超过此长度的说明将被截断。

13.5.2 Path 对象

class zipfile.**Path**(*root*, *at=''*)
根据 root zipfile (它可以是一个*ZipFile* 实例或适合传给*ZipFile* 构造器的 file) 构造一个 Path 对象。

at 指定此 Path 在 zipfile 中的位置，例如'dir/file.txt', 'dir/' 或''。默认为空字符串，即指定跟目录。

Path 对象会公开*pathlib.Path* 对象的下列特性:

Path 对象可以使用 / 操作符进行遍历。

Path.**name**
最终的路径组成部分。

Path.**open**(*mode='r'*, ***, *pwd*, ****)
在当前路径上发起调用*ZipFile.open()*。允许通过支持的模式打开用于读取或写入文本或二进制数据: 'r', 'w', 'rb', 'wb'。当以文本模式打开时位置和关键字参数会被传给*io.TextIOWrapper*，在其他情况下则会被忽略。pwd 是要传给*ZipFile.open()* 的 pwd 形参。

3.9 版更變: 增加了对以文本和二进制模式打开的支持。现在默认为文本模式。

Path.**iterdir**()
枚举当前目录的子目录。

Path.**is_dir**()
如果当前上下文引用了一个目录则返回 True。

Path.**is_file**()
如果当前上下文引用了一个文件则返回 True。

Path.**exists**()
如果当前上下文引用了 zip 文件内的一个文件或目录则返回 True。

Path.**read_text**(***, ****)
读取当前文件为 unicode 文本。位置和关键字参数会被传递给*io.TextIOWrapper* (buffer 除外，它将由上下文确定)。

Path.**read_bytes**()
读取当前文件为字节串。

13.5.3 PyZipFile 对象

PyZipFile 构造器接受与*ZipFile* 构造器相同的形参，以及一个额外的形参 *optimize*。

class zipfile.**PyZipFile**(*file*, *mode='r'*, *compression=ZIP_STORED*, *allowZip64=True*, *optimize=-1*)
3.2 版新加入: *optimize* 形参。

3.4 版更變: 默认启用 ZIP64 扩展。

实例在*ZipFile* 对象所具有的方法以外还附加了一个方法:

writepy(*pathname*, *basename=''*, *filterfunc=None*)
查找 *.py 文件并将相应的文件添加到归档。

如果*PyZipFile* 的 *optimize* 形参未给定或为 -1，则相应的文件为 *.pyc 文件，并在必要时进行编译。

如果*PyZipFile* 的 *optimize* 形参为 0, 1 或 2，则限具有相应优化级别 (参见*compile()*) 的文件会被添加到归档，并在必要时进行编译。

如果 *pathname* 是文件，则文件名必须以 .py 为后缀，并且只有 (相应的 *.pyc) 文件会被添加到最高层级（不带路径信息）。如果 *pathname* 不是以 .py 为后缀的文件，则将引发*RuntimeError*。如果它是目录，并且该目录不是一个包目录，则所有的 *.pyc 文件会被添加到最高层级。如果目录是一个包目录，则所有的 *.pyc 会被添加到包名所表示的文件路径下，并且如果有任何子目录为包目录，则会以排好的顺序递归地添加这些目录。

basename 仅限在内部使用。

如果给定 *filterfunc*，则它必须是一个接受单个字符串参数的函数。在将其添加到归档之前它将被传入每个路径（包括每个单独的完整路径）。如果 *filterfunc* 返回假值，则路径将不会被添加，而如果它是一个目录则其内容将被忽略。例如，如果我们的测试文件全都位于 test 目录或以字符串 test_ 打头，则我们可以使用一个 *filterfunc* 来排除它们:

```
>>> zf = PyZipFile('myprog.zip')
>>> def notests(s):
...     fn = os.path.basename(s)
...     return (not (fn == 'test' or fn.startswith('test_')))
>>> zf.writepy('myprog', filterfunc=notests)
```

writepy() 方法会产生带有这样一些文件名的归档:

```
string.pyc                      # Top level name
test/__init__.pyc               # Package directory
test/testall.pyc                # Module test.testall
test/bogus/__init__.pyc         # Subpackage directory
test/bogus/myfile.pyc           # Submodule test.bogus.myfile
```

3.4 版新加入: *filterfunc* 形参。

3.6.2 版更變: *pathname* 形参接受一个*path-like object*。

3.7 版更變: 递归排序目录条目。

13.5.4 ZipInfo 对象

ZipInfo 类的实例会通过*getinfo()* 和*ZipFile* 对象的*infolist()* 方法返回。每个对象将存储关于 ZIP 归档的一个成员的信息。

有一个类方法可以为文件系统文件创建*ZipInfo* 实例:

classmethod ZipInfo.**from_file**(*filename*, *arcname=None*, *, *strict_timestamps=True*)
为文件系统中的文件构造一个*ZipInfo* 实例，并准备将其添加到一个 zip 文件。

filename 应为文件系统中某个文件或目录的路径。

如果指定了 *arcname*，它会被用作归档中的名称。如果未指定 *arcname*，则所用名称与 *filename* 相同，但将去除任何驱动器盘符和打头的路径分隔符。

strict_timestamps 参数在设为 `False` 时允许压缩早于 1980-01-01 的文件，代价时会将时间戳设为 1980-01-01。类似的行为也会对晚于 2107-12-31 的文件发生，时间戳也会被设为该上限值。

3.6 版新加入.

3.6.2 版更變: *filename* 形参接受一个*path-like object*。

3.8 版新加入: *strict_timestamps* 仅限关键字参数

实例具有下列方法和属性:

ZipInfo.**is_dir**()
如果此归档成员是一个目录则返回 `True`。

这会使用条目的名称：目录应当总是以 / 结尾。

3.6 版新加入.

ZipInfo.**filename**
归档中的文件名称。

ZipInfo.**date_time**
上次修改存档成员的时间和日期。这是六个值的元组：

索引	值
0	Year (>= 1980)
1	月（1 为基数）
2	月份中的日期（1 为基数）
3	小时（0 为基数）
4	分钟（0 为基数）
5	秒（0 为基数）

備🄵: ZIP 文件格式不支持 1980 年以前的时间戳。

ZipInfo.**compress_type**
归档成员的压缩类型。

ZipInfo.**comment**
bytes 对象形式的单个归档成员的注释。

ZipInfo.**extra**
扩展字段数据。PKZIP Application Note 包含一些保存于该*bytes* 对象中的内部结构的注释。

ZipInfo.**create_system**
创建 ZIP 归档所用的系统。

ZipInfo.**create_version**
创建 ZIP 归档所用的 PKZIP 版本。

ZipInfo.**extract_version**
需要用来提取归档的 PKZIP 版本。

ZipInfo.**reserved**
必须为零。

ZipInfo.**flag_bits**
ZIP 标志位。

ZipInfo.**volume**
文件头的分卷号。

ZipInfo.**internal_attr**
内部属性。

ZipInfo.**external_attr**
外部文件属性。

ZipInfo.**header_offset**
文件头的字节偏移量。

ZipInfo.**CRC**
未压缩文件的 CRC-32。

ZipInfo.**compress_size**
已压缩数据的大小。

ZipInfo.**file_size**
未压缩文件的大小。

13.5.5 命令行界面

zipfile 模块提供了简单的命令行接口用于与 ZIP 归档的交互。

如果你想要创建一个新的 ZIP 归档，请在*-c* 选项后指定其名称然后列出应当被包含的文件名:

```
$ python -m zipfile -c monty.zip spam.txt eggs.txt
```

传入一个目录也是可接受的:

```
$ python -m zipfile -c monty.zip life-of-brian_1979/
```

如果你想要将一个 ZIP 归档提取到指定的目录，请使用*-e* 选项:

```
$ python -m zipfile -e monty.zip target-dir/
```

要获取一个 ZIP 归档中的文件列表，请使用*-l* 选项:

```
$ python -m zipfile -l monty.zip
```

命令行选项

-l <zipfile>
--list <zipfile>
列出一个 zipfile 中的文件名。

-c <zipfile> <source1> ... <sourceN>
--create <zipfile> <source1> ... <sourceN>
基于源文件创建 zipfile。

-e <zipfile> <output_dir>
--extract <zipfile> <output_dir>
将 zipfile 提取到目标目录中。

-t <zipfile>
--test <zipfile>
检测 zipfile 是否有效。

13.5.6 解压缩的障碍

zipfile 模块的提取操作可能会由于下面列出的障碍而失败。

由于文件本身

解压缩可能由于不正确的密码 / CRC 校验和 / ZIP 格式或不受支持的压缩 / 解密方法而失败。

文件系统限制

超出特定文件系统上的限制可能会导致解压缩失败。例如目录条目所允许的字符、文件名的长度、路径名的长度、单个文件的大小以及文件的数量等等。

资源限制

缺乏内存或磁盘空间将会导致解压缩失败。例如，作用于 zipfile 库的解压缩炸弹 (即 ZIP bomb) 就可能造成磁盘空间耗尽。

中断

在解压缩期间中断执行，例如按下 ctrl-C 或杀死解压缩进程可能会导致归档文件的解压缩不完整。

提取的默认行为

不了解提取的默认行为可能导致不符合期望的解压缩结果。例如，当提取相同归档两次时，它会不经询问地覆盖文件。

13.6 `tarfile` --- 读写 tar 归档文件

源代码: Lib/tarfile.py

---

tarfile 模块可以用来读写 tar 归档，包括使用 gzip, bz2 和 lzma 压缩的归档。请使用*zipfile* 模块来读写 `.zip` 文件，或者使用*shutil* 的高层级函数。

一些事实和数字:

- 读写*gzip*, *bz2* 和*lzma* 解压的归档要求相应的模块可用。
- 支持读取 / 写入 POSIX.1-1988 (ustar) 格式。
- 对 GNU tar 格式的读/写支持，包括 *longname* 和 *longlink* 扩展，对所有种类 *sparse* 扩展的只读支持，包括 sparse 文件的恢复。
- 对 POSIX.1-2001 (pax) 格式的读/写支持。
- 处理目录、正常文件、硬链接、符号链接、fifo 管道、字符设备和块设备，并且能够获取和恢复文件信息例如时间戳、访问权限和所有者等。

3.3 版更變: 添加了对*lzma* 压缩的支持。

`tarfile.`**`open`**(*name=None*, *mode='r'*, *fileobj=None*, *bufsize=10240*, ***kwargs*)

针对路径名 *name* 返回*TarFile* 对象。有关*TarFile* 对象以及所允许的关键字参数的详细信息请参阅*TarFile 对象*。

mode 必须是 `'filemode[:compression]'` 形式的字符串，其默认值为 `'r'`。以下是模式组合的完整列表:

模式	动作
`'r' or 'r:*'`	打开和读取使用透明压缩（推荐）。
`'r:'`	打开和读取不使用压缩。
`'r:gz'`	打开和读取使用 gzip 压缩。
`'r:bz2'`	打开和读取使用 bzip2 压缩。
`'r:xz'`	打开和读取使用 lzma 压缩。
`'x'` 或 `'x:'`	创建 tarfile 不进行压缩。如果文件已经存在，则抛出*FileExistsError* 异常。
`'x:gz'`	使用 gzip 压缩创建 tarfile。如果文件已经存在，则抛出*FileExistsError* 异常。
`'x:bz2'`	使用 bzip2 压缩创建 tarfile。如果文件已经存在，则抛出*FileExistsError* 异常。
`'x:xz'`	使用 lzma 压缩创建 tarfile。如果文件已经存在，则抛出*FileExistsError* 异常。
`'a' or 'a:'`	打开以便在没有压缩的情况下追加。如果文件不存在，则创建该文件。
`'w' or 'w:'`	打开用于未压缩的写入。
`'w:gz'`	打开用于 gzip 压缩的写入。
`'w:bz2'`	打开用于 bzip2 压缩的写入。
`'w:xz'`	打开用于 lzma 压缩的写入。

请注意 `'a:gz'`, `'a:bz2'` 或 `'a:xz'` 是不可能的组合。如果 *mode* 不适用于打开特定（压缩的）文件用于读取，则会引发*ReadError*。请使用 *mode* `'r'` 来避免这种情况。如果某种压缩方法不受支持，则会引发*CompressionError*。

如果指定了 *fileobj*，它会被用作对应于 *name* 的以二进制模式打开的*file object* 的替代。它会被设定为处在位置 0。

对于 `'w:gz'`, `'r:gz'`, `'w:bz2'`, `'r:bz2'`, `'x:gz'`, `'x:bz2'` 等模式，*tarfile.open()* 接受关键字参数 *compresslevel* (默认值为 9) 来指定文件的压缩等级。

对于 `'w:xz'` 和 `'x:xz'` 模式，*tarfile.open()* 接受关键字参数 *preset* 来指定文件的压缩等级。

针对特殊的目的，还存在第二种 *mode* 格式: `'filemode|[compression]'`。*tarfile.open()* 将返回一个将其数据作为数据块流来处理的*TarFile* 对象。对此文件将不能执行随机查找。如果给定了 *fileobj*，它可以是任何具有 `read()` 或 `write()` 方法 (由 *mode* 确定) 的对象。*bufsize* 指定块大小，默认值为 `20 * 512` 字节。可与此格式组合使用的有 `sys.stdin`, 套接字*file object* 或磁带设备等。但是，对于这样的*TarFile* 对象存在不允许随机访问的限制，参见*示例*。目前可用的模式如下:

模式	动作
`'r|*'`	打开 tar 块的 流以进行透明压缩读取。
`'r|'`	打开一个未压缩的 tar 块的 *stream* 用于读取。
`'r|gz'`	打开一个 gzip 压缩的 *stream* 用于读取。
`'r|bz2'`	打开一个 bzip2 压缩的 *stream* 用于读取。
`'r|xz'`	打开一个 lzma 压缩 *stream* 用于读取。
`'w|'`	打开一个未压缩的 *stream* 用于写入。
`'w|gz'`	打开一个 gzip 压缩的 *stream* 用于写入。
`'w|bz2'`	打开一个 bzip2 压缩的 *stream* 用于写入。
`'w|xz'`	打开一个 lzma 压缩的 *stream* 用于写入。

3.5 版更變: 添加了 `'x'` (仅创建) 模式。

3.6 版更變: *name* 形参接受一个*path-like object*。

class tarfile.**TarFile**

用于读取和写入 tar 归档的类。请不要直接使用这个类：而要使用*tarfile.open()*。参见*TarFile 对象*。

tarfile.**is_tarfile**(*name*)
如果 *name* 是一个*tarfile* 能读取的 tar 归档文件则返回*True*。*name* 可以为*str*，文件或文件类对象。

3.9 版更變: 支持文件或类文件对象。

tarfile 模块定义了以下异常:

exception tarfile.**TarError**
所有*tarfile* 异常的基类。

exception tarfile.**ReadError**
当一个不能被*tarfile* 模块处理或者因某种原因而无效的 tar 归档被打开时将被引发。

exception tarfile.**CompressionError**
当一个压缩方法不受支持或者当数据无法被正确解码时将被引发。

exception tarfile.**StreamError**
当达到流式*TarFile* 对象的典型限制时将被引发。

exception tarfile.**ExtractError**
当使用*TarFile.extract()* 时针对 *non-fatal* 所引发的异常，但是仅限 TarFile.errorlevel==2。

exception tarfile.**HeaderError**
如果获取的缓冲区无效则会由*TarInfo.frombuf()* 引发的异常。

以下常量在模块层级上可用:

tarfile.**ENCODING**
默认的字符编码格式: 在 Windows 上为 'utf-8'，其他系统上则为*sys.getfilesystemencoding()* 所返回的值。

以下常量各自定义了一个*tarfile* 模块能够创建的 tar 归档格式。相关细节请参阅*受支持的 tar 格式* 小节。

tarfile.**USTAR_FORMAT**
POSIX.1-1988 (ustar) 格式。

tarfile.**GNU_FORMAT**
GNU tar 格式。

tarfile.**PAX_FORMAT**
POSIX.1-2001 (pax) 格式。

tarfile.**DEFAULT_FORMAT**
用于创建归档的默认格式。目前为*PAX_FORMAT*。

3.8 版更變: 新归档的默认格式已更改为*PAX_FORMAT* 而不再是*GNU_FORMAT*。

也参考:

模块*zipfile* *zipfile* 标准模块的文档。

归档操作 标准*shutil* 模块所提供的高层级归档工具的文档。

GNU tar manual, Basic Tar Format 针对 tar 归档文件的文档，包含 GNU tar 扩展。

13.6.1 TarFile 对象

TarFile 对象提供了一个 tar 归档的接口。一个 tar 归档就是数据块的序列。一个归档成员（被保存文件）是由一个标头块加多个数据块组成的。一个文件可以在一个 tar 归档中多次被保存。每个归档成员都由一个*TarInfo* 对象来代表，详情参见*TarInfo 对象*。

TarFile 对象可在 `with` 语句中作为上下文管理器使用。当语句块结束时它将自动被关闭。请注意在发生异常事件时被打开用于写入的归档将不会被终结；只有内部使用的文件对象将被关闭。相关用例请参见*示例*。

3.2 版新加入: 添加了对上下文管理器协议的支持。

class tarfile.**TarFile**(*name=None*, *mode='r'*, *fileobj=None*, *format=DEFAULT_FORMAT*, *tarinfo=TarInfo*, *dereference=False*, *ignore_zeros=False*, *encoding=ENCODING*, *errors='surrogateescape'*, *pax_headers=None*, *debug=0*, *errorlevel=0*)

下列所有参数都是可选项并且也可作为实例属性来访问。

name 是归档的路径名称。*name* 可以是一个*path-like object*。如果给定了 *fileobj* 则它可以被省略。在此情况下，如果对象的 `name` 属性存在则它会被使用。

mode 可以为 `'r'` 表示从现有归档读取，`'a'` 表示将数据追加到现有文件，`'w'` 表示创建新文件覆盖现有文件，或者 `'x'` 表示仅在文件不存在时创建新文件。

如果给定了 *fileobj*，它会被用于读取或写入数据。如果可以被确定，则 *mode* 会被 *fileobj* 的模式所覆盖。*fileobj* 的使用将从位置 0 开始。

備F: 当*TarFile* 被关闭时，*fileobj* 不会被关闭。

---

format 控制用于写入的归档格式。它必须为在模块层级定义的常量*USTAR_FORMAT*, *GNU_FORMAT* 或*PAX_FORMAT* 中的一个。当读取时，格式将被自动检测，即使单个归档中存在不同的格式。

tarinfo 参数可以被用来将默认的*TarInfo* 类替换为另一个。

如果 *dereference* 为*False*，则会将符号链接和硬链接添加到归档中。如果为*True*，则会将目标文件的内容添加到归档中。在不支持符号链接的系统上参数将不起作用。

如果 *ignore_zeros* 为*False*，则会将空的数据块当作归档的末尾来处理。如果为*True*，则会跳过空的（和无效的）数据块并尝试获取尽可能多的成员。此参数仅适用于读取拼接的或损坏的归档。

debug 可设为从 `0` (无调试消息) 到 `3` (全部调试消息)。消息会被写入到 `sys.stderr`。

如果 *errorlevel* 为 `0`，则当使用*TarFile.extract()* 时会忽略所有错误。无论何种情况，当启用调试时它们都将被显示为调试输出的错误消息。如果为 `1`，则所有 *fatal* 错误会被作为*OSError* 异常被引发。如果为 `2`，则所有 *non-fatal* 错误也会被作为*TarError* 异常被引发。

encoding 和 *errors* 参数定义了读取或写入归档所使用的字符编码格式以及要如何处理转换错误。默认设置将适用于大多数用户。要深入了解详情可参阅*Unicode 问题* 小节。

可选的 *pax_headers* 参数是字符串的字典，如果 *format* 为*PAX_FORMAT* 它将被作为 pax 全局标头被添加。

3.2 版更變: 使用 `'surrogateescape'` 作为 *errors* 参数的默认值。

3.5 版更變: 添加了 `'x'` (仅创建) 模式。

3.6 版更變: *name* 形参接受一个*path-like object*。

classmethod TarFile.**open**(...)

作为替代的构造器。*tarfile.open()* 函数实际上是这个类方法的快捷方式。

TarFile.**getmember**(*name*)

返回成员 *name* 的*TarInfo* 对象。如果 *name* 在归档中找不到，则会引发*KeyError*。

備𝔽: 如果一个成员在归档中出现超过一次，它的最后一次出现会被视为是最新的版本。

TarFile.**getmembers**()
以*TarInfo* 对象列表的形式返回归档的成员。列表的顺序与归档中成员的顺序一致。

TarFile.**getnames**()
以名称列表的形式返回成员。它的顺序与*getmembers()* 所返回列表的顺序一致。

TarFile.**list**(*verbose=True*, ***, *members=None*)
将内容清单打印到 `sys.stdout`。如果 *verbose* 为*False*，则将只打印成员名称。如果为*True*，则输出将类似于 **`ls -l`** 的输出效果。如果给定了可选的 *members*，它必须为*getmembers()* 所返回的列表的一个子集。

3.5 版更變: 添加了 *members* 形参。

TarFile.**next**()
当*TarFile* 被打开用于读取时，以*TarInfo* 对象的形式返回归档的下一个成员。如果不再有可用对象则返回*None*。

TarFile.**extractall**(*path="."*, *members=None*, ***, *numeric_owner=False*)
将归档中的所有成员提取到当前工作目录或 *path* 目录。如果给定了可选的 *members*，则它必须为*getmembers()* 所返回的列表的一个子集。字典信息例如所有者、修改时间和权限会在所有成员提取完毕后被设置。这样做是为了避免两个问题：目录的修改时间会在每当在其中创建文件时被重置。并且如果目录的权限不允许写入，提取文件到目录的操作将失败。

如果 *numeric_owner* 为*True*，则将使用来自 tarfile 的 uid 和 gid 数值来设置被提取文件的所有者/用户组。在其他情况下，则会使用来自 tarfile 的名称值。

警告: 绝不要未经预先检验就从不可靠的源中提取归档文件。这样有可能在 *path* 之外创建文件，例如某些成员具有以 `"/"` 开始的绝对路径文件名或带有两个点号 `".."` 的文件名。

3.5 版更變: 添加了 *numeric_owner* 形参。

3.6 版更變: *path* 形参接受一个*path-like object*。

TarFile.**extract**(*member*, *path=""*, *set_attrs=True*, ***, *numeric_owner=False*)
从归档中提取出一个成员放入当前工作目录，将使用其完整名称。成员的文件信息会尽可能精确地被提取。*member* 可以是一个文件名或*TarInfo* 对象。你可以使用 *path* 指定一个不同的目录。*path* 可以是一个*path-like object*。将会设置文件属性 (owner, mtime, mode) 除非 *set_attrs* 为假值。

如果 *numeric_owner* 为*True*，则将使用来自 tarfile 的 uid 和 gid 数值来设置被提取文件的所有者/用户组。在其他情况下，则会使用来自 tarfile 的名称值。

備𝔽: *extract()* 方法不会处理某些提取问题。在大多数情况下你应当考虑使用*extractall()* 方法。

警告: 查看*extractall()* 的警告信息。

3.2 版更變: 添加了 *set_attrs* 形参。

3.5 版更變: 添加了 *numeric_owner* 形参。

3.6 版更變: *path* 形参接受一个*path-like object*。

TarFile.**extractfile**(*member*)
将归档中的一个成员提取为文件对象。*member* 可以是一个文件名或*TarInfo* 对象。如果 *member* 是一个常规文件或链接，则会返回一个*io.BufferedReader* 对象。对于所有其他现有成员，则都将返回*None*。如果 *member* 未在归档中出现，则会引发*KeyError*。

3.3 版更變: 返回一个*io.BufferedReader* 对象。

TarFile.**add**(*name*, *arcname=None*, *recursive=True*, ***, *filter=None*)
将文件 *name* 添加到归档。*name* 可以为任意类型的文件（目录、fifo、符号链接等等）。如果给出 *arcname* 则它将为归档中的文件指定一个替代名称。默认情况下会递归地添加目录。这可以通过将 *recursive* 设为*False* 来避免。递归操作会按排序顺序添加条目。如果给定了 *filter*，它应当为一个接受*TarInfo* 对象并返回已修改*TarInfo* 对象的函数。如果它返回*None* 则*TarInfo* 对象将从归档中被排除。具体示例参见示例。

3.2 版更變: 添加了 *filter* 形参。

3.7 版更變: 递归操作按排序顺序添加条目。

TarFile.**addfile**(*tarinfo*, *fileobj=None*)
将*TarInfo* 对象 *tarinfo* 添加到归档。如果给定了 *fileobj*，它应当是一个*binary file*，并会从中读取 `tarinfo.size` 个字节添加到归档。你可以直接创建*TarInfo* 对象，或是使用*gettarinfo()* 来创建。

TarFile.**gettarinfo**(*name=None*, *arcname=None*, *fileobj=None*)
基于*os.stat()* 的结果或者现有文件的相同数据创建一个*TarInfo*。文件或者是命名为 *name*，或者是使用文件描述符指定为一个*file object* *fileobj*。*name* 可以是一个*path-like object*。如果给定了 *arcname*，则它将为归档中的文件指定一个替代名称，在其他情况下，名称将从 *fileobj* 的*name* 属性或 *name* 参数获取。名称应当是一个文本字符串。

你可以在使用*addfile()* 添加*TarInfo* 的某些属性之前修改它们。如果文件对象不是从文件开头进行定位的普通文件对象，*size* 之类的属性就可能需要修改。例如*GzipFile* 之类的文件就属于这种情况。*name* 也可以被修改，在这种情况下 *arcname* 可以是一个占位字符串。

3.6 版更變: *name* 形参接受一个*path-like object*。

TarFile.**close**()
关闭*TarFile*。在写入模式下，会向归档添加两个表示结束的零数据块。

TarFile.**pax_headers**
一个包含 pax 全局标头的键值对的字典。

13.6.2 TarInfo 对象

TarInfo 对象代表*TarFile* 中的一个文件。除了会存储所有必要的文件属性（例如文件类型、大小、时间、权限、所有者等），它还提供了一些确定文件类型的有用方法。此对象 并不包含文件数据本身。

TarInfo 对象可通过*TarFile* 的方法 `getmember()`, `getmembers()` 和 `gettarinfo()` 返回。

class tarfile.**TarInfo**(*name=""*)
创建一个*TarInfo* 对象。

classmethod TarInfo.**frombuf**(*buf*, *encoding*, *errors*)
基于字符串缓冲区 *buf* 创建并返回一个*TarInfo* 对象。

如果缓冲区无效则会引发*HeaderError*。

classmethod TarInfo.**fromtarfile**(*tarfile*)
从*TarFile* 对象 *tarfile* 读取下一个成员并将其作为*TarInfo* 对象返回。

TarInfo.**tobuf**(*format=DEFAULT_FORMAT*, *encoding=ENCODING*, *errors='surrogateescape'*)
基于*TarInfo* 对象创建一个字符串缓冲区。有关参数的信息请参见*TarFile* 类的构造器。

3.2 版更變: 使用 'surrogateescape' 作为 *errors* 参数的默认值。

TarInfo 对象具有以下公有数据属性:

TarInfo.**name**
归档成员的名称。

TarInfo.**size**
以字节表示的大小。

TarInfo.**mtime**
上次修改的时间。

TarInfo.**mode**
权限位。

TarInfo.**type**
文件类型。*type* 通常为以下常量之一: REGTYPE, AREGTYPE, LNKTYPE, SYMTYPE, DIRTYPE, FIFOTYPE, CONTTYPE, CHRTYPE, BLKTYPE, GNUTYPE_SPARSE。要更方便地确定一个*TarInfo*对象的类型，请使用下述的 is*() 方法。

TarInfo.**linkname**
目标文件名的名称，该属性仅在类型为 LNKTYPE 和 SYMTYPE 的*TarInfo*对象中存在。

TarInfo.**uid**
最初保存该成员的用户的用户 ID。

TarInfo.**gid**
最初保存该成员的用户的分组 ID。

TarInfo.**uname**
用户名。

TarInfo.**gname**
分组名。

TarInfo.**pax_headers**
一个包含所关联的 pax 扩展标头的键值对的字典。

TarInfo 对象还提供了一些便捷查询方法:

TarInfo.**isfile**()
如果 Tarinfo 对象为普通文件则返回*True*。

TarInfo.**isreg**()
与*isfile()* 相同。

TarInfo.**isdir**()
如果为目录则返回*True*。

TarInfo.**issym**()
如果为符号链接则返回*True*。

TarInfo.**islnk**()
如果为硬链接则返回*True*。

TarInfo.**ischr**()
如果为字符设备则返回*True*。

TarInfo.**isblk**()
如果为块设备则返回*True*。

TarInfo.**isfifo**()
如果为 FIFO 则返回*True*。.

TarInfo.**isdev**()
如果为字符设备、块设备或 FIFO 之一则返回*True*。

13.6.3 命令行界面

3.4 版新加入.

tarfile 模块提供了简单的命令行接口以便与 tar 归档进行交互。

如果你想要创建一个新的 tar 归档，请在*-c* 选项后指定其名称然后列出应当被包含的文件名:

```
$ python -m tarfile -c monty.tar  spam.txt eggs.txt
```

传入一个字典也是可接受的:

```
$ python -m tarfile -c monty.tar life-of-brian_1979/
```

如果你想要将一个 tar 归档提取到指定的目录，请使用*-e* 选项:

```
$ python -m tarfile -e monty.tar
```

你也可以通过传入目录名称将一个 tar 归档提取到不同的目录:

```
$ python -m tarfile -e monty.tar  other-dir/
```

要获取一个 tar 归档中文件的列表，请使用*-l* 选项:

```
$ python -m tarfile -l monty.tar
```

命令行选项

-l <tarfile>
--list <tarfile>
列出一个 tarfile 中的文件名。

-c <tarfile> <source1> ... <sourceN>
--create <tarfile> <source1> ... <sourceN>
基于源文件创建 tarfile。

-e <tarfile> [<output_dir>]
--extract <tarfile> [<output_dir>]
如果未指定 *output_dir* 则会将 tarfile 提取到当前目录。

-t <tarfile>
--test <tarfile>
检测 tarfile 是否有效。

-v, --verbose
更详细地输出结果。

13.6.4 示例

如何将整个 tar 归档提取到当前工作目录:

```
import tarfile
tar = tarfile.open("sample.tar.gz")
tar.extractall()
tar.close()
```

如何通过*TarFile.extractall()* 使用生成器函数而非列表来提取一个 tar 归档的子集:

```
import os
import tarfile

def py_files(members):
    for tarinfo in members:
        if os.path.splitext(tarinfo.name)[1] == ".py":
            yield tarinfo

tar = tarfile.open("sample.tar.gz")
tar.extractall(members=py_files(tar))
tar.close()
```

如何基于一个文件名列表创建未压缩的 tar 归档:

```
import tarfile
tar = tarfile.open("sample.tar", "w")
for name in ["foo", "bar", "quux"]:
    tar.add(name)
tar.close()
```

使用 `with` 语句的同一个示例:

```
import tarfile
with tarfile.open("sample.tar", "w") as tar:
    for name in ["foo", "bar", "quux"]:
        tar.add(name)
```

如何读取一个 gzip 压缩的 tar 归档并显示一些成员信息:

```
import tarfile
tar = tarfile.open("sample.tar.gz", "r:gz")
for tarinfo in tar:
    print(tarinfo.name, "is", tarinfo.size, "bytes in size and is ", end="")
    if tarinfo.isreg():
        print("a regular file.")
    elif tarinfo.isdir():
        print("a directory.")
    else:
        print("something else.")
tar.close()
```

如何创建一个归档并使用*TarFile.add()* 中的 *filter* 形参来重置用户信息:

```
import tarfile
def reset(tarinfo):
    tarinfo.uid = tarinfo.gid = 0
```

(下页继续)

(繼續上一頁)

```
    tarinfo.uname = tarinfo.gname = "root"
    return tarinfo
tar = tarfile.open("sample.tar.gz", "w:gz")
tar.add("foo", filter=reset)
tar.close()
```

13.6.5 受支持的 tar 格式

通过*tarfile* 模块可以创建三种 tar 格式:

- The POSIX.1-1988 ustar 格式 (*USTAR_FORMAT*)。它支持最多 256 个字符的文件名长度和最多 100 个字符的链接名长度。文件大小上限为 8 GiB。这是一种老旧但广受支持的格式。
- GNU tar 格式 (*GNU_FORMAT*)。它支持长文件名和链接名、大于 8 GiB 的文件以及稀疏文件。它是 GNU/Linux 系统上的事实标准。*tarfile* 完全支持针对长名称的 GNU tar 扩展，稀疏文件支持则限制为只读。
- POSIX.1-2001 pax 格式 (*PAX_FORMAT*)。它是几乎无限制的最灵活格式。它支持长文件名和链接名，大文件以及使用便捷方式存储路径名。现代的 tar 实现，包括 GNU tar, bsdtar/libarchive 和 star，完全支持扩展 *pax* 特性；某些老旧或不维护的库可能不受支持，但应当会将 *pax* 归档视为广受支持的 *ustar* 格式。这是目前新建归档的默认格式。

  它扩展了现有的 *ustar* 格式，包括用于无法以其他方式存储的附加标头。存在两种形式的 pax 标头：扩展标头只影响后续的文件标头，全局标头则适用于完整归档并会影响所有后续的文件。为了便于移植，在 pax 标头中的所有数据均以 *UTF-8* 编码。

还有一些 tar 格式的其他变种，它们可以被读取但不能被创建:

- 古老的 V7 格式。这是来自 Unix 第七版的第一个 tar 格式，它只存储常规文件和目录。名称长度不能超过 100 个字符，并且没有用户/分组名信息。某些归档在带有非 ASCII 字符字段的情况下会产生计算错误的标头校验和。
- SunOS tar 扩展格式。此格式是 POSIX.1-2001 pax 格式的一个变种，但并不保持兼容。

13.6.6 Unicode 问题

最初 tar 格式被设计用来在磁带机上生成备份，主要关注于保存文件系统信息。现在 tar 归档通常用于文件分发和在网络上交换归档。最初格式（它是所有其他格式的基础）的一个问题是它没有支持不同字符编码格式的概念。例如，一个在 *UTF-8* 系统上创建的普通 tar 归档如果包含非 *ASCII* 字符则将无法在 *Latin-1* 系统上被正确读取。文本元数据（例如文件名，链接名，用户/分组名）将变为损坏状态。不幸的是，没有什么办法能够自动检测一个归档的编码格式。pax 格式被设计用来解决这个问题。它使用通用字符编码格式 *UTF-8* 来存储非 ASCII 元数据。

在*tarfile* 中字符转换的细节由*TarFile* 类的 *encoding* 和 *errors* 关键字参数控制。

encoding 定义了用于归档中元数据的字符编码格式。默认值为*sys.getfilesystemencoding()* 或是回退选项 `'ascii'`。根据归档是被读取还是被写入，元数据必须被解码或编码。如果没有正确设置 *encoding*，转换可能会失败。

errors 参数定义了不能被转换的字符将如何处理。可能的取值在*错误处理方案* 小节列出。默认方案为 `'surrogateescape'`，它也被 Python 用于文件系统调用，参见*文件名，命令行参数，以及环境变量。* 。

对于*PAX_FORMAT* 归档（默认格式），*encoding* 通常是不必要的，因为所有元数据都使用 *UTF-8* 来存储。*encoding* 仅在解码二进制 pax 标头或存储带有替代字符的字符串等少数场景下会被使用。

CHAPTER 14

檔案格式

本章中描述的模块解析各种不是标记语言且与电子邮件无关的杂项文件格式。

14.1 csv --- CSV 文件读写

源代码： Lib/csv.py

CSV (Comma Separated Values) 格式是电子表格和数据库中最常见的输入、输出文件格式。在 **RFC 4180** 规范推出的很多年前，CSV 格式就已经被开始使用了，由于当时并没有合理的标准，不同应用程序读写的数据会存在细微的差别。这种差别让处理多个来源的 CSV 文件变得困难。但尽管分隔符会变化，此类文件的大致格式是相似的，所以编写一个单独的模块以高效处理此类数据，将程序员从读写数据的繁琐细节中解放出来是有可能的。

csv 模块实现了 CSV 格式表单数据的读写。其提供了诸如“以兼容 Excel 的方式输出数据文件”或“读取 Excel 程序输出的数据文件”的功能，程序员无需知道 Excel 所采用 CSV 格式的细节。此模块同样可以用于定义其他应用程序可用的 CSV 格式或定义特定需求的 CSV 格式。

csv 模块中的*reader* 类和*writer* 类可用于读写序列化的数据。也可使用*DictReader* 类和*DictWriter* 类以字典的形式读写数据。

也参考：

该实现在“Python 增强提议” - PEP *305* (CSV 文件 API) 中被提出《Python 增强提议》提出了对 Python 的这一补充。

14.1.1 模块内容

csv 模块定义了以下函数：

csv.**reader**(*csvfile*, *dialect='excel'*, ***fmtparams*)
返回一个 reader 对象，该对象将逐行遍历 *csvfile*。*csvfile* 可以是任何对象，只要这个对象支持*iterator* 协议并在每次调用 `__next__()` 方法时都返回字符串，文件对象 和列表对象均适用。如果 *csvfile* 是文件对象，则打开它时应使用 `newline=''`。[1] 可选参数 *dialect* 是用于不同的 CSV 变种的特定参数组。它可以是*Dialect* 类的子类的实例，也可以是*list_dialects()* 函数返回的字符串之一。另一个可选关键字参数 *fmtparams* 可以覆写当前变种格式中的单个格式设置。有关变种和格式设置参数的完整详细信息，请参见变种与格式参数 部分。

csv 文件的每一行都读取为一个由字符串组成的列表。除非指定了 `QUOTE_NONNUMERIC` 格式选项（在这种情况下，未加引号的字段会转换为浮点数），否则不会执行自动数据类型转换。

一个简短的用法示例：

```
>>> import csv
>>> with open('eggs.csv', newline='') as csvfile:
...     spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')
...     for row in spamreader:
...         print(', '.join(row))
Spam, Spam, Spam, Spam, Spam, Baked Beans
Spam, Lovely Spam, Wonderful Spam
```

csv.**writer**(*csvfile*, *dialect='excel'*, ***fmtparams*)
返回一个 writer 对象，该对象负责将用户的数据在给定的文件类对象上转换为带分隔符的字符串。*csvfile* 可以是任何具有 `write()` 方法的对象。如果 *csvfile* 是一个文件对象，则打开它时应使用 `newline=''`[1]。可以给出可选的 *dialect* 形参用来定义一组特定 CSV 变种专属的形参。它可以是*Dialect* 类的某个子类的实例或是*list_dialects()* 函数所返回的字符串之一。还可以给出另一个可选的 *fmtparams* 关键字参数来覆盖当前变种中的单个格式化形参。有关各个变种和格式化形参的完整细节，请参阅变种与格式参数 部分。为了尽量简化与实现 DB API 的模块之间的接口，*None* 值会被当作空字符串写入。虽然这个转换是不可逆的，但它可以简化 SQL NULL 数据值到 CSV 文件的转储而无需预处理从 `cursor.fetch*` 调用返回的数据。在被写入之前所有其他非字符串数据都会先用*str()* 来转转为字符串。

一个简短的用法示例：

```
import csv
with open('eggs.csv', 'w', newline='') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=' ',
                            quotechar='|', quoting=csv.QUOTE_MINIMAL)
    spamwriter.writerow(['Spam'] * 5 + ['Baked Beans'])
    spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

csv.**register_dialect**(*name*[, *dialect*[, ***fmtparams*]])
将 *dialect* 与 *name* 关联起来。*name* 必须是字符串。变种的指定可以通过传入一个*Dialect* 的子类，或通过 *fmtparams* 关键字参数，或是两者同时传入，此时关键字参数会覆盖 dialect 形参。有关变种和格式化形参的完整细节，请参阅变种与格式参数 部分。

csv.**unregister_dialect**(*name*)
从变种注册表中删除 *name* 对应的变种。如果 *name* 不是已注册的变种名称，则抛出*Error* 异常。

csv.**get_dialect**(*name*)
返回 *name* 对应的变种。如果 *name* 不是已注册的变种名称，则抛出*Error* 异常。该函数返回的是不可变的*Dialect* 对象。

[1] 如果没有指定 `newline=''`，则嵌入引号中的换行符将无法正确解析，并且在写入时，使用 `\r\n` 换行的平台会有多余的 `\r` 写入。由于 csv 模块会执行自己的（通用）换行符处理，因此指定 `newline=''` 应该总是安全的。

csv.**list_dialects**()
返回所有已注册变种的名称。

csv.**field_size_limit**([*new_limit*])
返回解析器当前允许的最大字段大小。如果指定了 *new_limit*，则它将成为新的最大字段大小。

csv 模块定义了以下类：

class csv.**DictReader**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, **args*, ***kwds*)
创建一个对象，该对象在操作上类似于常规 reader，但是将每行中的信息映射到一个*dict*，该 dict 的键由 *fieldnames* 可选参数给出。

fieldnames 参数是一个*sequence*。如果省略 *fieldnames*，则文件 *f* 第一行中的值将用作字段名。无论字段名是如何确定的，字典都将保留其原始顺序。

如果某一行中的字段多于字段名，则剩余数据会被放入一个列表，并与 *restkey* 所指定的字段名 (默认为 None) 一起保存。如果某个非空白行的字段少于字段名，则缺失的值会使用 *restval* 的值来填充 (默认为 None)。

所有其他可选或关键字参数都传递给底层的*reader* 实例。

3.6 版更變: 返回的行现在的类型是 OrderedDict。

3.8 版更變: 现在，返回的行是*dict* 类型。

一个简短的用法示例：

```
>>> import csv
>>> with open('names.csv', newline='') as csvfile:
...     reader = csv.DictReader(csvfile)
...     for row in reader:
...         print(row['first_name'], row['last_name'])
...
Eric Idle
John Cleese

>>> print(row)
{'first_name': 'John', 'last_name': 'Cleese'}
```

class csv.**DictWriter**(*f*, *fieldnames*, *restval=''*, *extrasaction='raise'*, *dialect='excel'*, **args*, ***kwds*)
创建一个对象，该对象在操作上类似常规 writer，但会将字典映射到输出行。*fieldnames* 参数是由键组成的*序列*，它指定字典中值的顺序，这些值会按指定顺序传递给 writerow() 方法并写入文件 *f*。如果字典缺少 *fieldnames* 中的键，则可选参数 *restval* 用于指定要写入的值。如果传递给 writerow() 方法的字典的某些键在 *fieldnames* 中找不到，则可选参数 *extrasaction* 用于指定要执行的操作。如果将其设置为默认值 'raise'，则会引发*ValueError*。如果将其设置为 'ignore'，则字典中的其他键值将被忽略。所有其他可选或关键字参数都传递给底层的*writer* 实例。

注意，与*DictReader* 类不同，*DictWriter* 类的 *fieldnames* 参数不是可选参数。

一个简短的用法示例：

```
import csv

with open('names.csv', 'w', newline='') as csvfile:
    fieldnames = ['first_name', 'last_name']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'first_name': 'Baked', 'last_name': 'Beans'})
    writer.writerow({'first_name': 'Lovely', 'last_name': 'Spam'})
    writer.writerow({'first_name': 'Wonderful', 'last_name': 'Spam'})
```

class csv.**Dialect**

Dialect 类是一个容器类，其属性包含有如何处理双引号、空白符、分隔符等的信息。由于缺少严格的 CSV 规格描述，不同的应用程序会产生略有差别的 CSV 数据。*Dialect* 实例定义了*reader* 和*writer* 实例将具有怎样的行为。

所有可用的*Dialect* 名称会由*list_dialects()* 返回，并且它们可由特定的*reader* 和*writer* 类通过它们的初始化函数 (__init__) 来注册，例如:

```
import csv

with open('students.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, dialect='unix')
                                 ^^^^^^^^^^^^^^
```

class csv.**excel**

excel 类定义了 Excel 生成的 CSV 文件的常规属性。它在变种注册表中的名称是 'excel'。

class csv.**excel_tab**

excel_tab 类定义了 Excel 生成的、制表符分隔的 CSV 文件的常规属性。它在变种注册表中的名称是 'excel-tab'。

class csv.**unix_dialect**

unix_dialect 类定义了在 UNIX 系统上生成的 CSV 文件的常规属性，即使用 '\n' 作为换行符，且所有字段都有引号包围。它在变种注册表中的名称是 'unix'。

3.2 版新加入.

class csv.**Sniffer**

Sniffer 类用于推断 CSV 文件的格式。

Sniffer 类提供了两个方法:

sniff(*sample*, *delimiters=None*)

分析给定的 *sample* 并返回一个*Dialect* 子类，该子类中包含了分析出的格式参数。如果给出可选的 *delimiters* 参数，则该参数会被解释为字符串，该字符串包含了可能的有效定界符。

has_header(*sample*)

分析示例文本（假定为 CSV 格式），如果第一行很可能是一系列列标题，则返回*True*。

使用*Sniffer* 的示例:

```
with open('example.csv', newline='') as csvfile:
    dialect = csv.Sniffer().sniff(csvfile.read(1024))
    csvfile.seek(0)
    reader = csv.reader(csvfile, dialect)
    # ... process CSV file contents here ...
```

csv 模块定义了以下常量:

csv.**QUOTE_ALL**

指示*writer* 对象给所有字段加上引号。

csv.**QUOTE_MINIMAL**

指示*writer* 对象仅为包含特殊字符（例如 定界符、引号字符或 行结束符中的任何字符）的字段加上引号。

csv.**QUOTE_NONNUMERIC**

指示*writer* 对象为所有非数字字段加上引号。

指示 reader 将所有未用引号引出的字段转换为 *float* 类型。

csv.**QUOTE_NONE**
指示*writer*对象不使用引号引出字段。当 定界符出现在输出数据中时，其前面应该有 转义符。如果未设置 转义符，则遇到任何需要转义的字符时，writer 都会抛出*Error*异常。

指示*reader*不对引号字符进行特殊处理。

*csv*模块定义了以下异常：

exception csv.**Error**
该异常可能由任何发生错误的函数抛出。

14.1.2 变种与格式参数

为了更容易指定输入和输出记录的格式，特定的一组格式参数组合为一个 dialect（变种）。一个 dialect 是一个*Dialect*类的子类，它具有一组特定的方法和一个 validate() 方法。创建*reader*或*writer*对象时，程序员可以将某个字符串或*Dialect*类的子类指定为 dialect 参数。要想补充或覆盖 *dialect* 参数，程序员还可以单独指定某些格式参数，这些参数的名称与下面*Dialect*类定义的属性相同。

Dialect 类支持以下属性：

Dialect.**delimiter**
一个用于分隔字段的单字符，默认为 ','。

Dialect.**doublequote**
控制出现在字段中的 引号字符本身应如何被引出。当该属性为*True*时，双写引号字符。如果该属性为*False*，则在 引号字符的前面放置 转义符。默认值为*True*。

在输出时，如果 *doublequote* 是*False*，且 转义符未指定，且在字段中发现 引号字符时，会抛出*Error*异常。

Dialect.**escapechar**
一个用于 writer 的单字符，用来在 *quoting* 设置为*QUOTE_NONE*的情况下转义 定界符，在 *doublequote* 设置为*False*的情况下转义 引号字符。在读取时，*escapechar* 去除了其后所跟字符的任何特殊含义。该属性默认为*None*，表示禁用转义。

Dialect.**lineterminator**
放在*writer*产生的行的结尾，默认为 '\r\n'。

備F: *reader* 经过硬编码，会识别 '\r' 或 '\n' 作为行尾，并忽略 *lineterminator*。未来可能会更改这一行为。

Dialect.**quotechar**
一个单字符，用于包住含有特殊字符的字段，特殊字符如 定界符或 引号字符或换行符。默认为 '"'。

Dialect.**quoting**
控制 writer 何时生成引号，以及 reader 何时识别引号。该属性可以等于任何 QUOTE_* 常量（参见模块内容 段落），默认为*QUOTE_MINIMAL*。

Dialect.**skipinitialspace**
如果为*True*，则忽略 定界符之后的空格。默认值为*False*。

Dialect.**strict**
如果为 True，则在输入错误的 CSV 时抛出*Error*异常。默认值为 False。

14.1.3 Reader 对象

Reader 对象（*DictReader* 实例和*reader()* 函数返回的对象）具有以下公开方法：

csvreader.**__next__**()
: 返回 reader 的可迭代对象的下一行，它可以是一个列表（如果对象是由*reader()* 返回）或字典（如果是一个*DictReader* 实例），根据当前*Dialect* 来解析。通常你应当以 next(reader) 的形式来调用它。

Reader 对象具有以下公开属性：

csvreader.**dialect**
: 变种描述，只读，供解析器使用。

csvreader.**line_num**
: 源迭代器已经读取了的行数。它与返回的记录数不同，因为记录可能跨越多行。

DictReader 对象具有以下公开属性：

csvreader.**fieldnames**
: 字段名称。如果在创建对象时未传入字段名称，则首次访问时或从文件中读取第一条记录时会初始化此属性。

14.1.4 Writer 对象

Writer 对象（*DictWriter* 实例和*writer()* 函数返回的对象）具有下面的公开方法。对于 Writer 对象，行必须是（一组可迭代的）字符串或数字。对于*DictWriter* 对象，行必须是一个字典，这个字典将字段名映射为字符串或数字（数字要先经过*str()* 转换类型）。请注意，输出的复数会有括号包围。这样其他程序读取 CSV 文件时可能会有一些问题（假设它们完全支持复数）。

csvwriter.**writerow**(*row*)
: 将 *row* 形参写入到 writer 的文件对象，根据当前*Dialect* 进行格式化。返回对下层文件对象的 *write* 方法的调用的返回值。

  3.5 版更變: 开始支持任意类型的迭代器。

csvwriter.**writerows**(*rows*)
: 将 *rows**（即能迭代出多个上述 **row* 对象的迭代器）中的所有元素写入 writer 的文件对象，并根据当前设置的变种进行格式化。

Writer 对象具有以下公开属性：

csvwriter.**dialect**
: 变种描述，只读，供 writer 使用。

DictWriter 对象具有以下公开方法：

DictWriter.**writeheader**()
: 在 writer 的文件对象中，写入一行字段名称（字段名称在构造函数中指定），并根据当前设置的变种进行格式化。本方法的返回值就是内部使用的*csvwriter.writerow()* 方法的返回值。

  3.2 版新加入.

  3.8 版更變: 现在*writeheader()* 也返回其内部使用的*csvwriter.writerow()* 方法的返回值。

14.1.5 示例

读取 CSV 文件最简单的一个例子:

```
import csv
with open('some.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

读取其他格式的文件:

```
import csv
with open('passwd', newline='') as f:
    reader = csv.reader(f, delimiter=':', quoting=csv.QUOTE_NONE)
    for row in reader:
        print(row)
```

相应最简单的写入示例是:

```
import csv
with open('some.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerows(someiterable)
```

由于使用*open()* 来读取 CSV 文件，因此默认情况下，将使用系统默认编码来解码文件并转换为 unicode（请参阅*locale.getpreferredencoding()*）。要使用其他编码来解码文件，请使用 open 的 `encoding` 参数:

```
import csv
with open('some.csv', newline='', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

这同样适用于写入非系统默认编码的内容：打开输出文件时，指定 encoding 参数。

注册一个新的变种:

```
import csv
csv.register_dialect('unixpwd', delimiter=':', quoting=csv.QUOTE_NONE)
with open('passwd', newline='') as f:
    reader = csv.reader(f, 'unixpwd')
```

Reader 的更高级用法——捕获并报告错误:

```
import csv, sys
filename = 'some.csv'
with open(filename, newline='') as f:
    reader = csv.reader(f)
    try:
        for row in reader:
            print(row)
    except csv.Error as e:
        sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

尽管该模块不直接支持解析字符串，但仍可如下轻松完成:

```
import csv
for row in csv.reader(['one,two,three']):
    print(row)
```

𝔽解

14.2 configparser --- 配置文件解析器

源代码: Lib/configparser.py

此模块提供了它实现一种基本配置语言*ConfigParser* 类，这种语言所提供的结构与 Microsoft Windows INI 文件的类似。你可以使用这种语言来编写能够由最终用户来自定义的 Python 程序。

備𝔽: 这个库 并不能够解析或写入在 Windows Registry 扩展版本 INI 语法中所使用的值-类型前缀。

也参考:

模块*shlex* 支持创建可被用作应用配置文件的替代的 Unix 终端式微语言。

模块*json* json 模块实现了一个 JavaScript 语法的子集，它也可被用于这种目的。

14.2.1 快速起步

让我们准备一个非常基本的配置文件，它看起来是这样的:

```
[DEFAULT]
ServerAliveInterval = 45
Compression = yes
CompressionLevel = 9
ForwardX11 = yes

[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

INI 文件的结构描述见*以下章节*。总的来说，这种文件由多个节组成，每个节包含多个带有值的键。*configparser* 类可以读取和写入这种文件。让我们先通过程序方式来创建上述的配置文件。

```
>>> import configparser
>>> config = configparser.ConfigParser()
>>> config['DEFAULT'] = {'ServerAliveInterval': '45',
...                      'Compression': 'yes',
...                      'CompressionLevel': '9'}
>>> config['bitbucket.org'] = {}
>>> config['bitbucket.org']['User'] = 'hg'
>>> config['topsecret.server.com'] = {}
>>> topsecret = config['topsecret.server.com']
```

(下页继续)

(繼續上一頁)

```
>>> topsecret['Port'] = '50022'     # mutates the parser
>>> topsecret['ForwardX11'] = 'no'  # same here
>>> config['DEFAULT']['ForwardX11'] = 'yes'
>>> with open('example.ini', 'w') as configfile:
...   config.write(configfile)
...
```

如你所见，我们可以把配置解析器当作一个字典来处理。两者确实存在差异，将在后文说明，但是其行为非常接近于字典所具有一般行为。

现在我们已经创建并保存了一个配置文件，让我们再将它读取出来并探究其中包含的数据。

```
>>> config = configparser.ConfigParser()
>>> config.sections()
[]
>>> config.read('example.ini')
['example.ini']
>>> config.sections()
['bitbucket.org', 'topsecret.server.com']
>>> 'bitbucket.org' in config
True
>>> 'bytebong.com' in config
False
>>> config['bitbucket.org']['User']
'hg'
>>> config['DEFAULT']['Compression']
'yes'
>>> topsecret = config['topsecret.server.com']
>>> topsecret['ForwardX11']
'no'
>>> topsecret['Port']
'50022'
>>> for key in config['bitbucket.org']:
...     print(key)
user
compressionlevel
serveraliveinterval
compression
forwardx11
>>> config['bitbucket.org']['ForwardX11']
'yes'
```

正如我们在上面所看到的，相关的 API 相当直观。唯一有些神奇的地方是 `DEFAULT` 小节，它为所有其他小节提供了默认值[1]。还要注意小节中的键大小写不敏感并且会存储为小写形式[1]。

[1] 配置解析器允许重度定制。如果你有兴趣改变脚注说明中所介绍的行为，请参阅*Customizing Parser Behaviour* 一节。

14.2.2 支持的数据类型

配置解析器并不会猜测配置文件中值的类型，而总是将它们在内部存储为字符串。这意味着如果你需要其他数据类型，你应当自己来转换:

```
>>> int(topsecret['Port'])
50022
>>> float(topsecret['CompressionLevel'])
9.0
```

由于这种任务十分常用，配置解析器提供了一系列便捷的获取方法来处理整数、浮点数和布尔值。最后一个类型的处理最为有趣，因为简单地将值传给 `bool()` 是没有用的，`bool('False')` 仍然会是 `True`。为解决这个问题配置解析器还提供了*getboolean()*。这个方法对大小写不敏感并可识别 `'yes'`/`'no'`, `'on'`/`'off'`, `'true'`/`'false'` 和 `'1'`/`'0'`[1] 等布尔值。例如:

```
>>> topsecret.getboolean('ForwardX11')
False
>>> config['bitbucket.org'].getboolean('ForwardX11')
True
>>> config.getboolean('bitbucket.org', 'Compression')
True
```

除了*getboolean()*，配置解析器还提供了同类的*getint()* 和*getfloat()* 方法。你可以注册你自己的转换器并或是定制已提供的转换器。[1]

14.2.3 回退值

与字典类似，你可以使用某个小节的 `get()` 方法来提供回退值:

```
>>> topsecret.get('Port')
'50022'
>>> topsecret.get('CompressionLevel')
'9'
>>> topsecret.get('Cipher')
>>> topsecret.get('Cipher', '3des-cbc')
'3des-cbc'
```

请注意默认值会优先于回退值。例如，在我们的示例中 `'CompressionLevel'` 键仅在 `'DEFAULT'` 小节被指定。如果你尝试在 `'topsecret.server.com'` 小节获取它，我们将总是获取到默认值，即使我们指定了一个回退值:

```
>>> topsecret.get('CompressionLevel', '3')
'9'
```

还需要注意的一点是解析器层级的 `get()` 方法提供了自定义的更复杂接口，它被维护用于向下兼容。当使用此方法时，回退值可以通过 `fallback` 仅限关键字参数来提供:

```
>>> config.get('bitbucket.org', 'monster',
...            fallback='No such things as monsters')
'No such things as monsters'
```

同样的 `fallback` 参数也可在*getint()*, *getfloat()* 和*getboolean()* 方法中使用，例如:

```
>>> 'BatchMode' in topsecret
False
```

(下页继续)

(繼續上一頁)

```
>>> topsecret.getboolean('BatchMode', fallback=True)
True
>>> config['DEFAULT']['BatchMode'] = 'no'
>>> topsecret.getboolean('BatchMode', fallback=True)
False
```

14.2.4 受支持的 INI 文件结构

A configuration file consists of sections, each led by a `[section]` header, followed by key/value entries separated by a specific string (= or `:` by default[1]). By default, section names are case sensitive but keys are not[1]. Leading and trailing whitespace is removed from keys and values. Values can be omitted if the parser is configured to allow it[1], in which case the key/value delimiter may also be left out. Values can also span multiple lines, as long as they are indented deeper than the first line of the value. Depending on the parser's mode, blank lines may be treated as parts of multiline values or ignored.

By default, a valid section name can be any string that does not contain '\n' or ']'. To change this, see *ConfigParser.SECTCRE*.

配置文件可以包含注释，要带有指定字符前缀 (默认为 `#` 和 `;`[1])。注释可以单独出现于原本的空白行，并可使用缩进。[1]

例如:

```
[Simple Values]
key=value
spaces in keys=allowed
spaces in values=allowed as well
spaces around the delimiter = obviously
you can also use : to delimit keys from values

[All Values Are Strings]
values like this: 1000000
or this: 3.14159265359
are they treated as numbers? : no
integers, floats and booleans are held as: strings
can use the API to get converted values directly: true

[Multiline Values]
chorus: I'm a lumberjack, and I'm okay
    I sleep all night and I work all day

[No Values]
key_without_value
empty string value here =

[You can use comments]
# like this
; or this

# By default only in an empty line.
# Inline comments can be harmful because they prevent users
# from using the delimiting characters as parts of values.
# That being said, this can be customized.

    [Sections Can Be Indented]
```

(下页继续)

(繼續上一頁)

```
        can_values_be_as_well = True
        does_that_mean_anything_special = False
        purpose = formatting for readability
        multiline_values = are
            handled just fine as
            long as they are indented
            deeper than the first line
            of a value
        # Did I mention we can indent comments, too?
```

14.2.5 值的插值

在核心功能之上，*ConfigParser* 还支持插值。这意味着值可以在被 get() 调用返回之前进行预处理。

class configparser.**BasicInterpolation**

默认实现由*ConfigParser* 来使用。它允许值包含引用了相同小节中其他值或者特殊的默认小节中的值的格式字符串[1]。额外的默认值可以在初始化时提供。

例如:

```
[Paths]
home_dir: /Users
my_dir: %(home_dir)s/lumberjack
my_pictures: %(my_dir)s/Pictures

[Escape]
# use a %% to escape the % sign (% is the only character that needs to be␣
↪escaped):
gain: 80%%
```

在上面的例子里，*ConfigParser* 的 *interpolation* 设为 BasicInterpolation()，这会将 %(home_dir)s 求解为 home_dir 的值 (在这里是 /Users)。%(my_dir)s 的将被实际求解为 /Users/lumberjack。所有插值都是按需进行的，这样引用链中使用的键不必以任何特定顺序在配置文件中指明。

当 interpolation 设为 None 时，解析器会简单地返回 %(my_dir)s/Pictures 作为 my_pictures 的值，并返回 %(home_dir)s/lumberjack 作为 my_dir 的值。

class configparser.**ExtendedInterpolation**

一个用于插值的替代处理程序实现了更高级的语法，它被用于 zc.buildout 中的实例。扩展插值使用 ${section:option} 来表示来自外部小节的值。插值可以跨越多个层级。为了方便使用，section: 部分可被省略，插值会默认作用于当前小节（可能会从特殊小节获取默认值）。

例如，上面使用基本插值描述的配置，使用扩展插值将是这个样子:

```
[Paths]
home_dir: /Users
my_dir: ${home_dir}/lumberjack
my_pictures: ${my_dir}/Pictures

[Escape]
# use a $$ to escape the $ sign ($ is the only character that needs to be␣
↪escaped):
cost: $$80
```

来自其他小节的值也可以被获取:

```
[Common]
home_dir: /Users
library_dir: /Library
system_dir: /System
macports_dir: /opt/local

[Frameworks]
Python: 3.2
path: ${Common:system_dir}/Library/Frameworks/

[Arthur]
nickname: Two Sheds
last_name: Jackson
my_dir: ${Common:home_dir}/twosheds
my_pictures: ${my_dir}/Pictures
python_dir: ${Frameworks:path}/Python/Versions/${Frameworks:Python}
```

14.2.6 映射协议访问

3.2 版新加入.

映射协议访问这个通用名称是指允许以字典的方式来使用自定义对象的功能。在*configparser* 中，映射接口的实现使用了 `parser['section']['option']` 标记法。

`parser['section']` 专门为解析器中的小节数据返回一个代理。这意味着其中的值不会被拷贝，而是在需要时从原始解析器中获取。更为重要的是，当值在小节代理上被修改时，它们其实是在原始解析器中发生了改变。

configparser 对象的行为会尽可能地接近真正的字典。映射接口是完整而且遵循*MutableMapping* ABC 规范的。但是，还是有一些差异应当被纳入考虑:

- 默认情况下，小节中的所有键是以大小写不敏感的方式来访问的[1]。例如 `for option in parser["section"]` 只会产生 `optionxform` 形式的选项键名称。也就是说默认使用小写字母键名。与此同时，对于一个包含键 `'a'` 的小节，以下两个表达式均将返回 `True`:

```
"a" in parser["section"]
"A" in parser["section"]
```

- 所有小节也包括 `DEFAULTSECT`，这意味着对一个小节执行 `.clear()` 可能无法使得该小节显示为空。这是因为默认值是无法从小节中被删除的（因为从技术上说它们并不在那里）。如果它们在小节中被覆盖，删除将导致默认值重新变为可见。尝试删除默认值将会引发*KeyError*。
- `DEFAULTSECT` 无法从解析器中被移除:
  - 尝试删除将引发*ValueError*，
  - `parser.clear()` 会保留其原状，
  - `parser.popitem()` 绝不会将其返回。
- `parser.get(section, option, **kwargs)` - 第二个参数 **并非**回退值。但是请注意小节层级的 `get()` 方法可同时兼容映射协议和经典配置解析器 API。
- `parser.items()` 兼容映射协议（返回 *section_name*, *section_proxy* 对的列表，包括 DEFAULTSECT）。但是，此方法也可以带参数发起调用: `parser.items(section, raw, vars)`。这种调用形式返回指定 `section` 的 *option*, *value* 对的列表，将展开所有插值（除非提供了 `raw=True` 选项）。

映射协议是在现有的传统 API 之上实现的，以便重载原始接口的子类仍然具有符合预期的有效映射。

14.2.7 定制解析器行为

INI 格式的变种数量几乎和使用此格式的应用一样多。*configparser* 花费了很大力气来为尽量大范围的可用 INI 样式提供支持。默认的可用功能主要由历史状况来确定，你很可能会想要定制某些特性。

改变特定配置解析器行为的最常见方式是使用 `__init__()` 选项:

- *defaults*，默认值: `None`

  此选项接受一个键值对的字典，它将被首先放入 `DEFAULT` 小节。这实现了一种优雅的方式来支持简洁的配置文件，它不必指定与已记录的默认值相同的值。

  提示：如果你想要为特定的小节指定默认的值，请在读取实际文件之前使用 `read_dict()`。

- *dict_type*，默认值: *dict*

  此选项主要影响映射协议的行为和写入配置文件的外观。使用标准字典时，每个小节是按照它们被加入解析器的顺序保存的。在小节内的选项也是如此。

  还有其他替换的字典类型可以使用，例如在写回数据时对小节和选项进行排序。

  请注意：存在其他方式只用一次操作来添加键值对的集合。当你在这些操作中使用一个常规字典时，键将按顺序进行排列。例如:

```
>>> parser = configparser.ConfigParser()
>>> parser.read_dict({'section1': {'key1': 'value1',
...                                'key2': 'value2',
...                                'key3': 'value3'},
...                   'section2': {'keyA': 'valueA',
...                                'keyB': 'valueB',
...                                'keyC': 'valueC'},
...                   'section3': {'foo': 'x',
...                                'bar': 'y',
...                                'baz': 'z'}
... })
>>> parser.sections()
['section1', 'section2', 'section3']
>>> [option for option in parser['section3']]
['foo', 'bar', 'baz']
```

- *allow_no_value*，默认值: `False`

  已知某些配置文件会包括不带值的设置，但其在其他方面均符合*configparser* 所支持的语法。构造器的 *allow_no_value* 形参可用于指明应当接受这样的值:

```
>>> import configparser

>>> sample_config = """
... [mysqld]
...   user = mysql
...   pid-file = /var/run/mysqld/mysqld.pid
...   skip-external-locking
...   old_passwords = 1
...   skip-bdb
...   # we don't need ACID today
...   skip-innodb
... """
>>> config = configparser.ConfigParser(allow_no_value=True)
>>> config.read_string(sample_config)
```

(下页继续)

(繼續上一頁)

```
>>> # Settings with values are treated as before:
>>> config["mysqld"]["user"]
'mysql'

>>> # Settings without values provide None:
>>> config["mysqld"]["skip-bdb"]

>>> # Settings which aren't specified still raise an error:
>>> config["mysqld"]["does-not-exist"]
Traceback (most recent call last):
  ...
KeyError: 'does-not-exist'
```

- *delimiters*，默认值: `('=', ':')`

  分隔符是用于在小节内分隔键和值的子字符串。在一行中首次出现的分隔子字符串会被视为一个分隔符。这意味着值可以包含分隔符（但键不可以）。

  另请参见*ConfigParser.write()* 的 *space_around_delimiters* 参数。

- *comment_prefixes*，默认值: `('#', ';')`

- *inline_comment_prefixes*，默认值: `None`

  注释前缀是配置文件中用于标示一条有效注释的开头的字符串。*comment_prefixes* 仅用在被视为空白的行（可以缩进）之前而 *inline_comment_prefixes* 可用在每个有效值之后（例如小节名称、选项以及空白的行）。默认情况下禁用行内注释，并且 `'#'` 和 `';'` 都被用作完整行注释的前缀。

  3.2 版更變: 在之前的*configparser* 版本中行为匹配 `comment_prefixes=('#',';')` 和 `inline_comment_prefixes=(';',)`。

  请注意配置解析器不支持对注释前缀的转义，因此使用 *inline_comment_prefixes* 可能妨碍用户将被用作注释前缀的字符指定为可选值。当有疑问时，请避免设置 *inline_comment_prefixes*。在许多情况下，在多行值的一行开头存储注释前缀字符的唯一方式是进行前缀插值，例如:

```
>>> from configparser import ConfigParser, ExtendedInterpolation
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> # the default BasicInterpolation could be used as well
>>> parser.read_string("""
... [DEFAULT]
... hash = #
...
... [hashes]
... shebang =
...   ${hash}!/usr/bin/env python
...   ${hash} -*- coding: utf-8 -*-
...
... extensions =
...   enabled_extension
...   another_extension
...   #disabled_by_comment
...   yet_another_extension
...
... interpolation not necessary = if # is not at line start
... even in multiline values = line #1
...   line #2
...   line #3
... """)
```

(下页继续)

(繼續上一頁)

```
>>> print(parser['hashes']['shebang'])

#!/usr/bin/env python
# -*- coding: utf-8 -*-
>>> print(parser['hashes']['extensions'])

enabled_extension
another_extension
yet_another_extension
>>> print(parser['hashes']['interpolation not necessary'])
if # is not at line start
>>> print(parser['hashes']['even in multiline values'])
line #1
line #2
line #3
```

- *strict*，默认值: `True`

  当设为 `True` 时，解析器在从单一源读取 (使用 `read_file()`, `read_string()` 或 `read_dict()`) 期间将不允许任何小节或选项出现重复。推荐在新的应用中使用严格解析器。

  3.2 版更變: 在之前的*configparser* 版本中行为匹配 `strict=False`。

- *empty_lines_in_values*，默认值: `True`

  在配置解析器中，值可以包含多行，只要它们的缩进级别低于它们所对应的键。默认情况下解析器还会将空行视为值的一部分。于此同时，键本身也可以任意缩进以提升可读性。因此，当配置文件变得非常庞大而复杂时，用户很容易失去对文件结构的掌控。例如:

```
[Section]
key = multiline
  value with a gotcha

 this = is still a part of the multiline value of 'key'
```

  在用户查看时这可能会特别有问题，如果她是使用比例字体来编辑文件的话。这就是为什么当你的应用不需要带有空行的值时，你应该考虑禁用它们。这将使得空行每次都会作为键之间的分隔。在上面的示例中，空行产生了两个键，`key` 和 `this`。

- *default_section*，默认值: `configparser.DEFAULTSECT` (即: `"DEFAULT"`)

  允许设置一个保存默认值的特殊节在其他节或插值等目的中使用的惯例是这个库所拥有的一个强大概念，使得用户能够创建复杂的声明性配置。这种特殊节通常称为 `"DEFAULT"` 但也可以被定制为指向任何其他有效的节名称。一些典型的值包括: `"general"` 或 `"common"`。所提供的名称在从任意节读取的时候被用于识别默认的节，而且也会在将配置写回文件时被使用。它的当前值可以使用 `parser_instance.default_section` 属性来获取，并且可以在运行时被修改（即将文件从一种格式转换为另一种格式）。

- *interpolation*，默认值: `configparser.BasicInterpolation`

  插值行为可以用通过提供 *interpolation* 参数提供自定义处理程序的方式来定制。`None` 可用来完全禁用插值，`ExtendedInterpolation()` 提供了一种更高级的变体形式，它的设计受到了 `zc.buildout` 的启发。有关该主题的更多信息请参见专门的文档章节。*RawConfigParser* 具有默认的值 `None`。

- *converters*，默认值: 不设置

  配置解析器提供了可选的值获取方法用来执行类型转换。默认实现包括*getint()*, *getfloat()* 以及*getboolean()*。如果还需要其他获取方法，用户可以在子类中定义它们，或者传入一个字典，其中每个键都是一个转换器的名称而每个值都是一个实现了特定转换的可调用对象。例如，传

入 {'decimal': decimal.Decimal} 将对解释器对象和所有节代理添加 getdecimal()。换句话说，可以同时编写 parser_instance.getdecimal('section', 'key', fallback=0) 和 parser_instance['section'].getdecimal('key', 0)。

如果转换器需要访问解析器的状态，可以在配置解析器子类上作为一个方法来实现。如果该方法的名称是以 get 打头的，它将在所有节代理上以兼容字典的形式提供（参见上面的 getdecimal() 示例）。

更多高级定制选项可通过重载这些解析器属性的默认值来达成。默认值是在类中定义的，因此它们可以通过子类或属性赋值来重载。

ConfigParser.**BOOLEAN_STATES**

默认情况下当使用*getboolean()* 时，配置解析器会将下列值视为 True: '1', 'yes', 'true', 'on' 而将下列值视为 False: '0', 'no', 'false', 'off'。你可以通过指定一个自定义的字符串键及其对应的布尔值字典来重载此行为。例如:

```
>>> custom = configparser.ConfigParser()
>>> custom['section1'] = {'funky': 'nope'}
>>> custom['section1'].getboolean('funky')
Traceback (most recent call last):
...
ValueError: Not a boolean: nope
>>> custom.BOOLEAN_STATES = {'sure': True, 'nope': False}
>>> custom['section1'].getboolean('funky')
False
```

其他典型的布尔值对包括 accept/reject 或 enabled/disabled。

ConfigParser.**optionxform**(*option*)

这个方法会转换每次 read, get, 或 set 操作的选项名称。默认会将名称转换为小写形式。这也意味着当一个配置文件被写入时，所有键都将为小写形式。如果此行为不合适则要重载此方法。例如:

```
>>> config = """
... [Section1]
... Key = Value
...
... [Section2]
... AnotherKey = Value
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> list(typical['Section1'].keys())
['key']
>>> list(typical['Section2'].keys())
['anotherkey']
>>> custom = configparser.RawConfigParser()
>>> custom.optionxform = lambda option: option
>>> custom.read_string(config)
>>> list(custom['Section1'].keys())
['Key']
>>> list(custom['Section2'].keys())
['AnotherKey']
```

備🄵: optionxform 函数会将选项名称转换为规范形式。这应该是一个幂等函数：如果名称已经为规范形式，则应不加修改地将其返回。

ConfigParser.**SECTCRE**

一个已编译正则表达式会被用来解析节标头。默认将 [section] 匹配到名称 "section"。空格会被

视为节名称的一部分，因此 `[ larch ]` 将被读取为一个名称为 `" larch "` 的节。如果此行为不合适则要重载此属性。例如:

```
>>> import re
>>> config = """
... [Section 1]
... option = value
...
... [  Section 2  ]
... another = val
... """
>>> typical = configparser.ConfigParser()
>>> typical.read_string(config)
>>> typical.sections()
['Section 1', '  Section 2  ']
>>> custom = configparser.ConfigParser()
>>> custom.SECTCRE = re.compile(r"\[ *(?P<header>[^]]+?) *\]")
>>> custom.read_string(config)
>>> custom.sections()
['Section 1', 'Section 2']
```

備🄴: 虽然 ConfigParser 对象也使用 `OPTCRE` 属性来识别选项行，但并不推荐重载它，因为这会与构造器选项 *allow_no_value* 和 *delimiters* 产生冲突。

14.2.8 旧式 API 示例

主要出于向下兼容性的考虑，*configparser* 还提供了一种采用显式 `get`/`set` 方法的旧式 API。虽然以下介绍的方法存在有效的用例，但对于新项目仍建议采用映射协议访问。旧式 API 在多数时候都更复杂、更底层并且完全违反直觉。

一个写入配置文件的示例:

```
import configparser

config = configparser.RawConfigParser()

# Please note that using RawConfigParser's set functions, you can assign
# non-string values to keys internally, but will receive an error when
# attempting to write to a file or when you get it in non-raw mode. Setting
# values using the mapping protocol or ConfigParser's set() does not allow
# such assignments to take place.
config.add_section('Section1')
config.set('Section1', 'an_int', '15')
config.set('Section1', 'a_bool', 'true')
config.set('Section1', 'a_float', '3.1415')
config.set('Section1', 'baz', 'fun')
config.set('Section1', 'bar', 'Python')
config.set('Section1', 'foo', '%(bar)s is %(baz)s!')

# Writing our configuration file to 'example.cfg'
with open('example.cfg', 'w') as configfile:
    config.write(configfile)
```

一个再次读取配置文件的示例:

```
import configparser

config = configparser.RawConfigParser()
config.read('example.cfg')

# getfloat() raises an exception if the value is not a float
# getint() and getboolean() also do this for their respective types
a_float = config.getfloat('Section1', 'a_float')
an_int = config.getint('Section1', 'an_int')
print(a_float + an_int)

# Notice that the next output does not interpolate '%(bar)s' or '%(baz)s'.
# This is because we are using a RawConfigParser().
if config.getboolean('Section1', 'a_bool'):
    print(config.get('Section1', 'foo'))
```

要获取插值，请使用*ConfigParser*:

```
import configparser

cfg = configparser.ConfigParser()
cfg.read('example.cfg')

# Set the optional *raw* argument of get() to True if you wish to disable
# interpolation in a single get operation.
print(cfg.get('Section1', 'foo', raw=False))  # -> "Python is fun!"
print(cfg.get('Section1', 'foo', raw=True))   # -> "%(bar)s is %(baz)s!"

# The optional *vars* argument is a dict with members that will take
# precedence in interpolation.
print(cfg.get('Section1', 'foo', vars={'bar': 'Documentation',
                                       'baz': 'evil'}))

# The optional *fallback* argument can be used to provide a fallback value
print(cfg.get('Section1', 'foo'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'foo', fallback='Monty is not.'))
      # -> "Python is fun!"

print(cfg.get('Section1', 'monster', fallback='No such things as monsters.'))
      # -> "No such things as monsters."

# A bare print(cfg.get('Section1', 'monster')) would raise NoOptionError
# but we can also use:

print(cfg.get('Section1', 'monster', fallback=None))
      # -> None
```

默认值在两种类型的 ConfigParser 中均可用。它们将在当某个选项未在别处定义时被用于插值。

```
import configparser

# New instance with 'bar' and 'baz' defaulting to 'Life' and 'hard' each
config = configparser.ConfigParser({'bar': 'Life', 'baz': 'hard'})
config.read('example.cfg')

```

(下页继续)

(繼續上一頁)

```
print(config.get('Section1', 'foo'))      # -> "Python is fun!"
config.remove_option('Section1', 'bar')
config.remove_option('Section1', 'baz')
print(config.get('Section1', 'foo'))      # -> "Life is hard!"
```

14.2.9 ConfigParser 对象

class configparser.**ConfigParser**(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*, *interpolation=BasicInterpolation()*, *converters={}*)

主配置解析器。当给定 *defaults* 时，它会被初始化为包含固有默认值的字典。当给定 *dict_type* 时，它将被用来创建包含节、节中的选项以及默认值的字典。

当给定 *delimiters* 时，它会被用作分隔键与值的子字符串的集合。当给定 *comment_prefixes* 时，它将被用作在否则为空行的注释的前缀子字符串的集合。注释可以被缩进。当给定 *inline_comment_prefixes* 时，它将被用作非空行的注释的前缀子字符串的集合。

当 *strict* 为 `True` (默认值) 时，解析器在从单个源（文件、字符串或字典）读取时将不允许任何节或选项出现重复，否则会引发*DuplicateSectionError* 或*DuplicateOptionError*。当 *empty_lines_in_values* 为 `False` (默认值: `True`) 时，每个空行均表示一个选项的结束。在其他情况下，一个多行选项内部的空行会被保留为值的一部分。当 *allow_no_value* 为 `True` (默认值: `False`) 时，将接受没有值的选项；此种选项的值将为 `None` 并且它们会以不带末尾分隔符的形式被序列化。

当给定 *default_section* 时，它将指定存放其他节的默认值和用于插值的特殊节的名称 (通常命名为 `"DEFAULT"`)。该值可通过使用 `default_section` 实例属性在运行时被读取或修改。

插值行为可通过给出 *interpolation* 参数提供自定义处理程序的方式来定制。`None` 可用来完全禁用插值，`ExtendedInterpolation()` 提供了一种更高级的变体形式，它的设计受到了 `zc.buildout` 的启发。有关该主题的更多信息请参见专门的文档章节。

插值中使用的所有选项名称将像任何其他选项名称引用一样通过*optionxform()* 方法来传递。例如，使用*optionxform()* 的默认实现（它会将选项名称转换为小写形式）时，值 `foo %(bar)s` 和 `foo %(BAR)s` 是等价的。

当给定 *converters* 时，它应当为一个字典，其中每个键代表一个类型转换器的名称而每个值则为实现从字符串到目标数据类型的转换的可调用对象。每个转换器会获得其在解析器对象和节代理上对应的 `get*()` 方法。

3.1 版更變: 默认的 *dict_type* 为*collections.OrderedDict*。

3.2 版更變: 添加了 *allow_no_value*, *delimiters*, *comment_prefixes*, *strict*, *empty_lines_in_values*, *default_section* 以及 *interpolation*。

3.5 版更變: 添加了 *converters* 参数。

3.7 版更變: *defaults* 参数会通过*read_dict()* 来读取，提供全解析器范围内一致的行为：非字符串类型的键和值会被隐式地转换为字符串。

3.8 版更變: 默认的 *dict_type* 为*dict*，因为它现在会保留插入顺序。

defaults()
返回包含实例范围内默认值的字典。

sections()
返回可用节的列表；*default section* 不包括在该列表中。

add_section(*section*)
向实例添加一个名为 *section* 的节。如果给定名称的节已存在，将会引发*DuplicateSectionError*。如果传入了 *default section* 名称，则会引发*ValueError*。节名称必须为字符串；如果不是则会引发*TypeError*。

3.2 版更變: 非字符串的节名称将引发*TypeError*。

has_section(*section*)
指明相应名称的 *section* 是否存在于配置中。*default section* 不包含在内。

options(*section*)
返回指定 *section* 中可用选项的列表。

has_option(*section*, *option*)
如果给定的 *section* 存在并且包含给定的 *option* 则返回*True*；否则返回*False*。如果指定的 *section* 为*None* 或空字符串，则会使用 DEFAULT。

read(*filenames*, *encoding=None*)
尝试读取并解析一个包含文件名的可迭代对象，返回一个被成功解析的文件名列表。

如果 *filenames* 为字符串、*bytes* 对象或*path-like object*，它会被当作单个文件来处理。如果 *filenames* 中名称对应的某个文件无法被打开，该文件将被忽略。这样的设计使得你可以指定包含多个潜在配置文件位置的可迭代对象（例如当前目录、用户家目录以及某个系统级目录），存在于该可迭代对象中的所有配置文件都将被读取。

如果名称对应的文件全都不存在，则*ConfigParser* 实例将包含一个空数据集。一个要求从文件加载初始值的应用应当在调用*read()* 来获取任何可选文件之前使用*read_file()* 来加载所要求的一个或多个文件:

```
import configparser, os

config = configparser.ConfigParser()
config.read_file(open('defaults.cfg'))
config.read(['site.cfg', os.path.expanduser('~/.myapp.cfg')],
            encoding='cp1250')
```

3.2 版新加入: *encoding* 形参。在之前的版本中，所有文件都将使用*open()* 的默认编码格式来读取。

3.6.1 版新加入: *filenames* 形参接受一个*path-like object*。

3.7 版新加入: *filenames* 形参接受一个*bytes* 对象。

read_file(*f*, *source=None*)
从 *f* 读取并解析配置数据，它必须是一个产生 Unicode 字符串的可迭代对象（例如以文本模式打开的文件）。

可选参数 *source* 指定要读取的文件名称。如果未给出并且 *f* 具有 `name` 属性，则该属性会被用作 *source*；默认值为 `'<???>'`。

3.2 版新加入: 替代*readfp()*。

read_string(*string*, *source='<string>'*)
从字符串中解析配置数据。

可选参数 *source* 指定一个所传入字符串的上下文专属名称。如果未给出，则会使用 `'<string>'`。这通常应为一个文件系统路径或 URL。

3.2 版新加入.

read_dict(*dictionary*, *source='<dict>'*)
从任意一个提供了类似于字典的 `items()` 方法的对象加载配置。键为节名称，值为包含节中所

出现的键和值的字典。如果所用的字典类型会保留顺序，则节和其中的键将按顺序加入。值会被自动转换为字符串。

可选参数 *source* 指定一个所传入字曲的上下文专属名称。如果未给出，则会使用 `<dict>`。

此方法可被用于在解析器之间拷贝状态。

3.2 版新加入.

get(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
获取指定名称的 *section* 的一个 *option* 的值。如果提供了 *vars*，则它必须为一个字典。*option* 的查找顺序为 *vars**（如果有提供）、**section* 以及 *DEFAULTSECT*。如果未找到该键并且提供了 *fallback*，则它会被用作回退值。可以提供 `None` 作为 *fallback* 值。

所有 `'%'` 插值会在返回值中被展开，除非 *raw* 参数为真值。插值键所使用的值会按与选项相同的方式来查找。

3.2 版更變: *raw*, *vars* 和 *fallback* 都是仅限关键字参数，以防止用户试图使用第三个参数作业为 *fallback* 回退值（特别是在使用映射协议的时候）。

getint(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
将在指定 *section* 中的 *option* 强制转换为整数的便捷方法。参见*get()* 获取对于 *raw*, *vars* 和 *fallback* 的解释。

getfloat(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
将在指定 *section* 中的 *option* 强制转换为浮点数的便捷方法。参见*get()* 获取对于 *raw*, *vars* 和 *fallback* 的解释。

getboolean(*section*, *option*, *, *raw=False*, *vars=None*[, *fallback*])
将在指定 *section* 中的 *option* 强制转换为布尔值的便捷方法。请注意选项所接受的值为 `'1'`, `'yes'`, `'true'` 和 `'on'`，它们会使得此方法返回 `True`，以及 `'0'`, `'no'`, `'false'` 和 `'off'`，它们会使得此方法返回 `False`。这些字符串值会以对大小写不敏感的方式被检测。任何其他值都将导致引发*ValueError*。参见*get()* 获取对于 *raw*, *vars* 和 *fallback* 的解释。

items(*raw=False*, *vars=None*)
items(*section*, *raw=False*, *vars=None*)
当未给出 *section* 时，将返回由 *section_name*, *section_proxy* 对组成的列表，包括 DEFAULTSECT。

在其他情况下，将返回给定的 *section* 中的 option 的 *name*, *value* 对组成的列表。可选参数具有与*get()* 方法的参数相同的含义。

3.8 版更變: *vars* 中的条目将不在结果中出现。之前的行为混淆了实际的解析器选项和为插值提供的变量。

set(*section*, *option*, *value*)
如果给定的节存在，则将所给出的选项设为指定的值；在其他情况下将引发*NoSectionError*。*option* 和 *value* 必须为字符串；如果不是则将引发*TypeError*。

write(*fileobject*, *space_around_delimiters=True*)
将配置的表示形式写入指定的*file object*，该对象必须以文本模式打开（接受字符串）。此表示形式可由将来的*read()* 调用进行解析。如果 *space_around_delimiters* 为真值，键和值之前的分隔符两边将加上空格。

備𝔽: 原始配置文件中的注释在写回配置时不会被保留。具体哪些会被当作注释，取决于为 *comment_prefix* 和 *inline_comment_prefix* 所指定的值。

---

remove_option(*section*, *option*)
将指定的 *option* 从指定的 *section* 中移除。如果指定的节不存在则会引发*NoSectionError*。如果要移除的选项存在则返回*True*；在其他情况下将返回*False*。

remove_section(*section*)
从配置中移除指定的 *section*。如果指定的节确实存在则返回 `True`。在其他情况下将返回 `False`。

optionxform(*option*)
将选项名 *option* 转换为输入文件中的形式或客户端代码所传入的应当在内部结构中使用的形式。默认实现将返回 *option* 的小写形式版本；子类可以重载此行为，或者客户端代码也可以在实例上设置一个具有此名称的属性来影响此行为。

你不需要子类化解析器来使用此方法，你也可以在一个实例上设置它，或使用一个接受字符串参数并返回字符串的函数。例如将它设为 `str` 将使得选项名称变得大小写敏感:

```
cfgparser = ConfigParser()
cfgparser.optionxform = str
```

请注意当读取配置文件时，选项名称两边的空格将在调用*optionxform()* 之前被去除。

readfp(*fp*, *filename=None*)
3.2 版後已🄴用: 使用*read_file()* 来代替。

3.2 版更變: *readfp()* 现在将在 *fp* 上执行迭代而不是调用 `fp.readline()`。

对于调用*readfp()* 时传入不支持迭代的参数的现有代码，可以在文件类对象外使用以下生成器作为包装器:

```
def readline_generator(fp):
    line = fp.readline()
    while line:
        yield line
        line = fp.readline()
```

不再使用`parser.readfp(fp)` 而是改用`parser.read_file(readline_generator(fp))`。

configparser.**MAX_INTERPOLATION_DEPTH**
当 *raw* 形参为假值时 `get()` 所采用的递归插值的最大深度。这只在使用默认的 *interpolation* 时会起作用。

14.2.10 RawConfigParser 对象

class configparser.**RawConfigParser**(*defaults=None*, *dict_type=dict*, *allow_no_value=False*, ***, *delimiters=('=', ':')*, *comment_prefixes=('#', ';')*, *inline_comment_prefixes=None*, *strict=True*, *empty_lines_in_values=True*, *default_section=configparser.DEFAULTSECT*[, *interpolation*])
旧式*ConfigParser*。它默认禁用插值并且允许通过不安全的 `add_section` 和 `set` 方法以及旧式 `defaults=` 关键字参数处理来设置非字符串的节名、选项名和值。

3.8 版更變: 默认的 *dict_type* 为*dict*，因为它现在会保留插入顺序。

備🄴: 考虑改用*ConfigParser*，它会检查内部保存的值的类型。如果你不想要插值，你可以使用 `ConfigParser(interpolation=None)`。

---

add_section(*section*)
向实例添加一个名为 *section* 的节。如果给定名称的节已存在，将会引发*DuplicateSectionError*。如果传入了 *default section* 名称，则会引发*ValueError*。

不检查 *section* 以允许用户创建以非字符串命名的节。此行为已不受支持并可能导致内部错误。

set(*section*, *option*, *value*)
如果给定的节存在，则将给定的选项设为指定的值；在其他情况下将引发*NoSectionError*。虽然可能使用*RawConfigParser*(或使用*ConfigParser* 并将 *raw* 形参设为真值) 以便实现非字符串值的 *internal* 存储，但是完整功能（包括插值和输出到文件）只能使用字符串值来实现。

此方法允许用户在内部将非字符串值赋给键。此行为已不受支持并会在尝试写入到文件或在非原始模式下获取数据时导致错误。**请使用映射协议 API**，它不允许出现这样的赋值。

14.2.11 异常

exception configparser.**Error**
所有其他*configparser* 异常的基类。

exception configparser.**NoSectionError**
当找不到指定节时引发的异常。

exception configparser.**DuplicateSectionError**
当调用 add_section() 时传入已存在的节名称，或者在严格解析器中当单个输入文件、字符串或字典内出现重复的节时引发的异常。

3.2 版新加入: 将可选的 source 和 lineno 属性和参数添加到 __init__()。

exception configparser.**DuplicateOptionError**
当单个选项在从单个文件、字符串或字典读取时出现两次时引发的异常。这会捕获拼写错误和大小写敏感相关的错误，例如一个字典可能包含两个键分别代表同一个大小写不敏感的配置键。

exception configparser.**NoOptionError**
当指定的选项未在指定的节中被找到时引发的异常。

exception configparser.**InterpolationError**
当执行字符串插值发生问题时所引发的异常的基类。

exception configparser.**InterpolationDepthError**
当字符串插值由于迭代次数超出*MAX_INTERPOLATION_DEPTH* 而无法完成所引发的异常。为*InterpolationError* 的子类。

exception configparser.**InterpolationMissingOptionError**
当从某个值引用的选项并不存在时引发的异常。为*InterpolationError* 的子类。

exception configparser.**InterpolationSyntaxError**
当将要执行替换的源文本不符合要求的语法时引发的异常。为*InterpolationError* 的子类。

exception configparser.**MissingSectionHeaderError**
当尝试解析一个不带节标头的文件时引发的异常。

exception configparser.**ParsingError**
当尝试解析一个文件而发生错误时引发的异常。

3.2 版更變: filename 属性和 __init__() 参数被重命名为 source 以保持一致性。

𝔽解

14.3 netrc --- netrc 文件处理

源代码: Lib/netrc.py

---

netrc 类解析并封装了 Unix 的 **ftp** 程序和其他 FTP 客户端所使用的 netrc 文件格式。

class netrc.**netrc**([*file*])
netrc 的实例或其子类的实例会被用来封装来自 netrc 文件的数据。如果有初始化参数，它将指明要解析的文件。如果未给出参数，则位于用户家目录的 .netrc 文件 -- 即*os.path.expanduser()* 所确定的文件 -- 将会被读取。在其他情况下，则将引发*FileNotFoundError* 异常。解析错误将引发*NetrcParseError* 并附带诊断信息，包括文件名、行号以及终止令牌。如果在 POSIX 系统上未指明参数，则当 .netrc 文件中有密码时，如果文件归属或权限不安全（归属的用户不是运行进程的用户，或者可供任何其他用户读取或写入）将引发*NetrcParseError*。这实现了与 ftp 和其他使用 .netrc 的程序同等的安全行为。

3.4 版更變: 添加了 POSIX 权限检查。

3.7 版更變: 当未将 *file* 作为参数传入时会使用*os.path.expanduser()* 来查找 .netrc 文件的位置。

exception netrc.**NetrcParseError**
当在源文本中遇到语法错误时由*netrc* 类引发的异常。此异常的实例提供了三个有用属性: msg 为错误的文本说明，filename 为源文件的名称，而 lineno 给出了错误所在的行号。

14.3.1 netrc 对象

netrc 实例具有下列方法:

netrc.**authenticators**(*host*)
针对 *host* 的身份验证者返回一个 3 元组 (login, account, password)。如果 netrc 文件不包含针对给定主机的条目，则返回关联到'default' 条目的元组。如果匹配的主机或默认条目均不可用，则返回 None。

netrc.**__repr__**()
将类数据以 netrc 文件的格式转储为一个字符串。(这会丢弃注释并可能重排条目顺序。)

netrc 的实例具有一些公共实例变量:

netrc.**hosts**
将主机名映射到 (login, account, password) 元组的字典。如果存在'default' 条目，则会表示为使用该名称的伪主机。

netrc.**macros**
将宏名称映射到字符串列表的字典。

備𝔽: 密码会被限制为 ASCII 字符集的一个子集。所有 ASCII 标点符号均可用作密码，但是要注意空白符和非打印字符不允许用作密码。这是.netrc 文件解析方式带来的限制，在未来可能会被解除。

14.4 plistlib --- 生成与解析 Apple .plist 文件

源代码: Lib/plistlib.py

此模块提供了可读写 Apple "property list" 文件的接口，它主要用于 macOS 和 iOS 系统。此模块同时支持二进制和 XML plist 文件。

property list (`.plist`) 文件格式是一种简单的序列化格式，它支持一些基本对象类型，例如字典、列表、数字和字符串等。通常使用一个字典作为最高层级对象。

要写入和解析 plist 文件，请使用*dump()* 和*load()* 函数。

要以字节串对象形式操作 plist 数据，请使用*dumps()* 和*loads()*。

值可以为字符串、整数、浮点数、布尔值、元组、列表、字典（但只允许用字符串作为键）、*bytes*、*bytearray* 或*datetime.datetime* 对象。

3.4 版更變: 新版 API，旧版 API 已被弃用。添加了对二进制 plist 格式的支持。

3.8 版更變: 添加了在二进制 plist 中读写*UID* 令牌的支持，例如用于 NSKeyedArchiver 和 NSKeyedUnarchiver。

3.9 版更變: 旧 API 已被移除。

也参考:

PList 指南页面 针对该文件格式的 Apple 文档。

这个模块定义了以下函数:

plistlib.**load**(*fp*, *, *fmt=None*, *dict_type=dict*)
读取 plist 文件。*fp* 应当可读并且为二进制文件对象。返回已解包的根对象（通常是一个字典）。

fmt 为文件的格式，有效的值如下:

- *None*: 自动检测文件格式
- *FMT_XML*: XML 文件格式
- *FMT_BINARY*: 二进制 plist 格式

dict_type 为字典用来从 plist 文件读取的类型。

FMT_XML 格式的 XML 数据会使用来自*xml.parsers.expat* 的 Expat 解析器 -- 请参阅其文档了解错误格式 XML 可能引发的异常。未知元素将被 plist 解析器直接略过。

当文件无法被解析时二进制格式的解析器将引发 `InvalidFileException`。

3.4 版新加入.

plistlib.**loads**(*data*, *, *fmt=None*, *dict_type=dict*)
从一个 bytes 对象加载 plist。参阅*load()* 获取相应关键字参数的说明。

3.4 版新加入.

plistlib.**dump**(*value*, *fp*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*)
将 *value* 写入 plist 文件。*Fp* 应当可写并且为二进制文件对象。

fmt 参数指定 plist 文件的格式，可以是以下值之一:

- *FMT_XML*: XML 格式的 plist 文件
- *FMT_BINARY*: 二进制格式的 plist 文件

当 *sort_keys* 为真值（默认）时字典的键将经过排序再写入 plist，否则将按字典的迭代顺序写入。

当 *skipkeys* 为假值（默认）时该函数将在字典的键不为字符串时引发*TypeError*，否则将跳过这样的键。

如果对象是不受支持的类型或者是包含不受支持类型的对象的容器则将引发*TypeError*。

对于无法在（二进制）plist 文件中表示的整数值，将会引发*OverflowError*。

3.4 版新加入.

plistlib.**dumps**(*value*, *, *fmt=FMT_XML*, *sort_keys=True*, *skipkeys=False*)
将 *value* 以 plist 格式字节串对象的形式返回。参阅*dump()* 的文档获取此函数的关键字参数的说明。

3.4 版新加入.

可以使用以下的类:

class plistlib.**UID**(*data*)
包装一个*int*。该类将在读取或写入 NSKeyedArchiver 编码的数据时被使用，其中包含 UID（参见 PList 指南）。

It has one attribute, `data`, which can be used to retrieve the int value of the UID. `data` must be in the range `0 <= data < 2**64`.

3.8 版新加入.

可以使用以下的常量:

plistlib.**FMT_XML**
用于 plist 文件的 XML 格式。

3.4 版新加入.

plistlib.**FMT_BINARY**
用于 plist 文件的二进制格式。

3.4 版新加入.

14.4.1 示例

生成一个 plist:

```
pl = dict(
    aString = "Doodah",
    aList = ["A", "B", 12, 32.1, [1, 2, 3]],
    aFloat = 0.1,
    anInt = 728,
    aDict = dict(
        anotherString = "<hello & hi there!>",
        aThirdString = "M\xe4ssig, Ma\xdf",
        aTrueValue = True,
        aFalseValue = False,
    ),
    someData = b"<binary gunk>",
    someMoreData = b"<lots of binary gunk>" * 10,
    aDate = datetime.datetime.fromtimestamp(time.mktime(time.gmtime())),
)
with open(fileName, 'wb') as fp:
    dump(pl, fp)
```

解析一个 plist:

```
with open(fileName, 'rb') as fp:
    pl = load(fp)
print(pl["aKey"])
```

CHAPTER 15

加密服务

本章中描述的模块实现了加密性质的各种算法。它们可以在安装时自行选择。在 Unix 系统上，*crypt* 模块也可以使用。以下是为内容概要：

15.1 hashlib --- 安全哈希与消息摘要

源码： Lib/hashlib.py

这个模块针对不同的安全哈希和消息摘要算法实现了一个通用的接口。包括 FIPS 的 SHA1, SHA224, SHA256, SHA384, and SHA512 (定义于 FIPS 180-2) 算法，以及 RSA 的 MD5 算法 (定义于 Internet **RFC 1321**)。术语“安全哈希”和“消息摘要”是可互换的，较旧的算法被称为消息摘要，现代术语是安全哈希。

備🄴： 如果你想找到 adler32 或 crc32 哈希函数，它们在*zlib* 模块中。

警告： 有些算法已知存在哈希碰撞弱点，请参考最后的“另请参阅”段。

15.1.1 哈希算法

每种类型的 *hash* 都有一个构造器方法。它们都返回一个具有相同的简单接口的 hash 对象。例如，使用 use `sha256()` 创建一个 SHA-256 hash 对象。你可以使用 `update()` 方法向这个对象输入*字节类对象* (通常是*bytes*)。在任何时候你都可以使用 `digest()` 或 `hexdigest()` 方法获得到目前为止输入这个对象的拼接数据的 *digest*。

備🄴： 为了更好的多线程性能，在对象创建或者更新时，若数据大于 2047 字节则 Python 的*GIL* 会被释放。

備🄵: 向 update() 输入字符串对象是不被支持的，因为哈希基于字节而非字符。

此模块中总是可用的哈希算法构造器有 sha1(), sha224(), sha256(), sha384(), sha512(), *blake2b()* 和*blake2s()*。md5() 通常也是可用的，但如果你在使用少见的"FIPS 兼容" 的 Python 编译版本则可能会找不到它。此外还可能有一些附加的算法，具体取决于你的平台上的 Python 所使用的 OpenSSL 库。在大部分平台上可用的还有 sha3_224(), sha3_256(), sha3_384(), sha3_512(), shake_128(), shake_256() 等等。·

3.6 版新加入: SHA3 (Keccak) 和 SHAKE 构造器 sha3_224(), sha3_256(), sha3_384(), sha3_512(), shake_128(), shake_256().

3.6 版新加入: 添加了*blake2b()* 和*blake2s()* 。 3.9 版更變: 所有 hashlib 的构造器都接受仅限关键字参数 *usedforsecurity* 且其默认值为 True。设为假值即允许在受限的环境中使用不安全且阻塞的哈希算法。False 表示此哈希算法不可用于安全场景，例如用作非加密的单向压缩函数。

现在 hashlib 会使用 OpenSSL 1.1.1 或更新版本的 SHA3 和 SHAKE。

例如，如果想获取字节串 b'Nobody inspects the spammish repetition' 的摘要:

```
>>> import hashlib
>>> m = hashlib.sha256()
>>> m.update(b"Nobody inspects")
>>> m.update(b" the spammish repetition")
>>> m.digest()
b'\x03\x1e\xdd}Ae\x15\x93\xc5\xfe\\\x00o\xa5u+7\xfd\xdf\xf7\xbcN\x84:\xa6\xaf\x0c\x95\
↪x0fK\x94\x06'
>>> m.digest_size
32
>>> m.block_size
64
```

更简要的写法:

```
>>> hashlib.sha224(b"Nobody inspects the spammish repetition").hexdigest()
'a4337bc45a8fc544c03f52dc550cd6e1e87021bc896588bd79e901e2'
```

hashlib.**new**(*name*[, *data*], *, *usedforsecurity=True*)
一个接受所希望的算法对应的字符串 *name* 作为第一个形参的通用构造器。它还允许访问上面列出的哈希算法以及你的 OpenSSL 库可能提供的任何其他算法。同名的构造器要比*new()* 更快所以应当优先使用。

使用*new()* 并附带由 OpenSSL 所提供了算法:

```
>>> h = hashlib.new('sha256')
>>> h.update(b"Nobody inspects the spammish repetition")
>>> h.hexdigest()
'031edd7d41651593c5fe5c006fa5752b37fddff7bc4e843aa6af0c950f4b9406'
```

Hashlib 提供下列常量属性:

hashlib.**algorithms_guaranteed**
一个集合，其中包含此模块在所有平台上都保证支持的哈希算法的名称。请注意'md5' 也在此清单中，虽然某些上游厂商提供了一个怪异的排除了此算法的"FIPS 兼容" Python 编译版本。

3.2 版新加入.

hashlib.**algorithms_available**
一个集合，其中包含在所运行的 Python 解释器上可用的哈希算法的名称。将这些名称传给*new()* 时将

可被识别。*algorithms_guaranteed* 将总是它的一个子集。同样的算法在此集合中可能以不同的名称出现多次（这是 OpenSSL 的原因）。

3.2 版新加入.

下列值会以构造器所返回的哈希对象的常量属性的形式被提供:

hash.**digest_size**
以字节表示的结果哈希对象的大小。

hash.**block_size**
以字节表示的哈希算法的内部块大小。

hash 对象具有以下属性:

hash.**name**
此哈希对象的规范名称，总是为小写形式并且总是可以作为*new()* 的形参用来创建另一个此类型的哈希对象。

3.4 版更變: 该属性名称自被引入起即存在于 CPython 中，但在 Python 3.4 之前并未正式指明，因此可能不存在于某些平台上。

哈希对象具有下列方法:

hash.**update**(*data*)
用*bytes-like object* 来更新哈希对象。重复调用相当于单次调用并传入所有参数的拼接结果: `m.update(a); m.update(b)` 等价于 `m.update(a+b)`。

3.1 版更變: 当使用 OpenSSL 提供的哈希算法在大于 2047 字节的数据上执行哈希更新时 Python GIL 会被释放以允许其他线程运行。

hash.**digest**()
返回当前已传给*update()* 方法的数据摘要。这是一个大小为*digest_size* 的字节串对象，字节串中可包含 0 至 255 的完整取值范围。

hash.**hexdigest**()
类似于*digest()* 但摘要会以两倍长度字符串对象的形式返回，其中仅包含十六进制数码。这可以被用于在电子邮件或其他非二进制环境中安全地交换数据值。

hash.**copy**()
返回哈希对象的副本（“克隆”）。这可被用来高效地计算共享相同初始子串的数据的摘要。

15.1.2 SHAKE 可变长度摘要

`shake_128()` 和 `shake_256()` 算法提供安全的 length_in_bits//2 至 128 或 256 位可变长度摘要。为此，它们的摘要需指定一个长度。SHAKE 算法不限制最大长度。

shake.**digest**(*length*)
返回当前已传给 `update()` 方法的数据摘要。这是一个大小为 *length* 的字节串对象，字节串中可包含 0 to 255 的完整取值范围。

shake.**hexdigest**(*length*)
类似于*digest()* 但摘要会以两倍长度字符串对象的形式返回，其中仅包含十六进制数码。这可以被用于在电子邮件或其他非二进制环境中安全地交换数据值。

15.1.3 密钥派生

密钥派生和密钥延展算法被设计用于安全密码哈希。`sha1(password)` 这样的简单算法无法防御暴力攻击。好的密码哈希函数必须可以微调、放慢步调，并且包含 加盐。

`hashlib.`**`pbkdf2_hmac`**(*hash_name*, *password*, *salt*, *iterations*, *dklen=None*)

此函数提供 PKCS#5 基于密码的密钥派生函数 2。它使用 HMAC 作为伪随机函数。

字符串 *hash_name* 是要求用于 HMAC 的哈希摘要算法的名称，例如'sha1' 或'sha256'。*password* 和 *salt* 会以字节串缓冲区的形式被解析。应用和库应当将 *password* 限制在合理长度 (例如 1024)。*salt* 应当为适当来源例如*os.urandom()* 的大约 16 个或更多的字节串数据。

iterations 数值应当基于哈希算法和算力来选择。在 2013 年时，建议至少为 100,000 次 SHA-256 迭代。

dklen 为派生密钥的长度。如果 *dklen* 为 `None` 则会使用哈希算法 *hash_name* 的摘要大小，例如 SHA-512 为 64。

```
>>> import hashlib
>>> dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 100000)
>>> dk.hex()
'0394a2ede332c9a13eb82e9b24631604c31df978b4e2f0fbd2c549944f9d79a5'
```

3.4 版新加入.

備🄴: 随同 OpenSSL 提供了一个快速的 *pbkdf2_hmac* 实现。Python 实现是使用*hmac* 的内联版本。它的速度大约要慢上三倍并且不会释放 GIL。

`hashlib.`**`scrypt`**(*password*, ***, *salt*, *n*, *r*, *p*, *maxmem=0*, *dklen=64*)

此函数提供基于密码加密的密钥派生函数，其定义参见 **RFC 7914**。

password 和 *salt* 必须为字节类对象。应用和库应当将 *password* 限制在合理长度 (例如 1024)。*salt* 应当为适当来源例如*os.urandom()* 的大约 16 个或更多的字节串数据。

n 为 CPU/内存开销因子，*r* 为块大小，*p* 为并行化因子，*maxmem* 为内存限制 (OpenSSL 1.1.0 默认为 32 MiB)。*dklen* 为派生密钥的长度。

可用性: OpenSSL 1.1+。

3.6 版新加入.

15.1.4 BLAKE2

BLAKE2 是在 **RFC 7693** 中定义的加密哈希函数，它有两种形式:

- **BLAKE2b**，针对 64 位平台进行优化，并会生成长度介于 1 和 64 字节之间任意大小的摘要。
- **BLAKE2s**，针对 8 至 32 位平台进行优化，并会生成长度介于 1 和 32 字节之间任意大小的摘要。

BLAKE2 支持 **keyed mode** (HMAC 的更快速更简单的替代), **salted hashing**, **personalization** 和 **tree hashing**.

此模块的哈希对象遵循标准库*hashlib* 对象的 API。

创建哈希对象

新哈希对象可通过调用构造器函数来创建:

hashlib.**blake2b**(*data=b''*, ***, *digest_size=64*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*)

hashlib.**blake2s**(*data=b''*, ***, *digest_size=32*, *key=b''*, *salt=b''*, *person=b''*, *fanout=1*, *depth=1*, *leaf_size=0*, *node_offset=0*, *node_depth=0*, *inner_size=0*, *last_node=False*, *usedforsecurity=True*)

这些函数返回用于计算 BLAKE2b 或 BLAKE2s 的相应的哈希对象。它们接受下列可选通用形参:

- *data*: 要哈希的初始数据块，它必须为*bytes-like object*。它只能作为位置参数传入。
- *digest_size*: 以字节数表示的输出摘要大小。
- *key*: 用于密钥哈希的密钥（对于 BLAKE2b 最长 64 字节，对于 BLAKE2s 最长 32 字节）。
- *salt*: 用于随机哈希的盐值（对于 BLAKE2b 最长 16 字节，对于 BLAKE2s 最长 8 字节）。
- *person*: 个性化字符串（对于 BLAKE2b 最长 16 字节，对于 BLAKE2s 最长 8 字节）。

下表显示了常规参数的限制（以字节为单位）:

Hash	目标长度	长度（键）	长度（盐）	长度（个人）
BLAKE2b	64	64	16	16
BLAKE2s	32	32	8	8

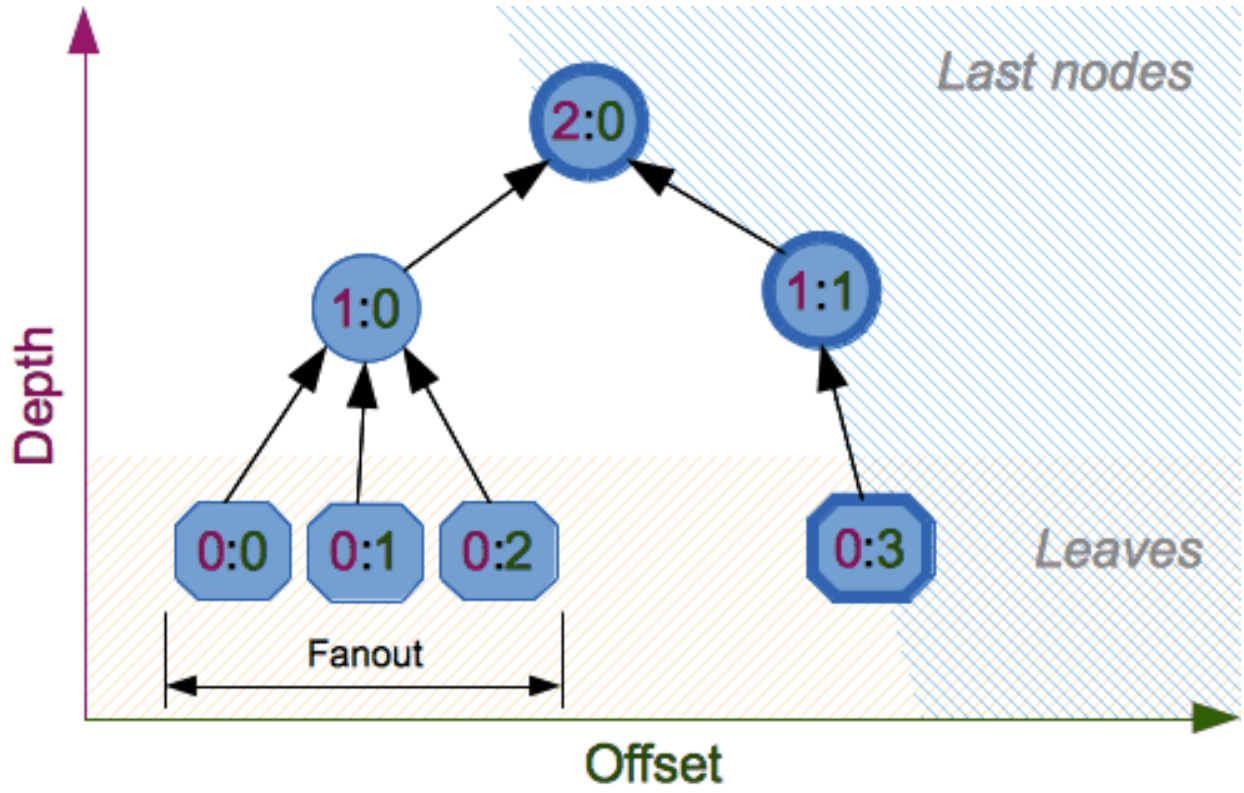
備🄵: BLAKE2 规格描述为盐值和个性化形参定义了固定的长度，但是为了方便起见，此实现接受指定在长度以内的任意大小的字节串。如果形参长度小于指定值，它将以零值进行填充，因此举例来说，`b'salt'` 和 `b'salt\x00'` 为相同的值 (*key* 的情况则并非如此。)

如下面的模块*constants* 所描述，这些是可用的大小取值。

构造器函数还接受下列树形哈希形参:

- *fanout*: 扇出值 (0 至 255，如无限制即为 0，连续模式下为 1)。
- *depth*: 树的最大深度 (1 至 255，如无限制则为 255，连续模式下为 1)。
- *leaf_size*: maximal byte length of leaf (0 to `2**32-1`, 0 if unlimited or in sequential mode).
- *node_offset*: node offset (0 to `2**64-1` for BLAKE2b, 0 to `2**48-1` for BLAKE2s, 0 for the first, leftmost, leaf, or in sequential mode).
- *node_depth*: 节点深度 (0 至 255，对于叶子或在连续模式下则为 0)。
- *inner_size*: 内部摘要大小 (对于 BLAKE2b 为 0 至 64，对于 BLAKE2s 为 0 至 32，连续模式下则为 0)。
- *last_node*: 一个布尔值，指明所处理的节点是否为最后一个 (连续模式下则为 *False*)。

请参阅 BLAKE2 规格描述 第 2.10 节获取有关树形哈希的完整说明。



常数

blake2b.**SALT_SIZE**

blake2s.**SALT_SIZE**

盐值长度（构造器所接受的最大长度）。

blake2b.**PERSON_SIZE**

blake2s.**PERSON_SIZE**

个性化字符串长度（构造器所接受的最大长度）。

blake2b.**MAX_KEY_SIZE**

blake2s.**MAX_KEY_SIZE**

最大密钥长度。

blake2b.**MAX_DIGEST_SIZE**

blake2s.**MAX_DIGEST_SIZE**

哈希函数可输出的最大摘要长度。

示例

简单哈希

要计算某个数据的哈希值，你应该首先通过调用适当的构造器函数 (*blake2b()* 或 *blake2s()*) 来构造一个哈希对象，然后通过在该对象上调用 `update()` 来更新目标数据，最后通过调用 `digest()` (或针对十六进制编码字符串的 `hexdigest()`) 来获取该对象的摘要。

```
>>> from hashlib import blake2b
>>> h = blake2b()
>>> h.update(b'Hello world')
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

作为快捷方式，你可以直接以位置参数的形式向构造器传入第一个数据块来直接更新:

```
>>> from hashlib import blake2b
>>> blake2b(b'Hello world').hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

你可以多次调用 *hash.update()* 至你所想要的任意次数以迭代地更新哈希值:

```
>>> from hashlib import blake2b
>>> items = [b'Hello', b' ', b'world']
>>> h = blake2b()
>>> for item in items:
...     h.update(item)
>>> h.hexdigest()

↪'6ff843ba685842aa82031d3f53c48b66326df7639a63d128974c5c14f31a0f33343a8c65551134ed1ae0f2b0dd2bb495dc
↪'
```

使用不同的摘要大小

BLAKE2 具有可配置的摘要大小，对于 BLAKE2b 最多 64 字节，对于 BLAKE2s 最多 32 字节。例如，要使用 BLAKE2b 来替代 SHA-1 而不改变输出大小，我们可以让 BLAKE2b 产生 20 个字节的摘要:

```
>>> from hashlib import blake2b
>>> h = blake2b(digest_size=20)
>>> h.update(b'Replacing SHA1 with the more secure function')
>>> h.hexdigest()
'd24f26cf8de66472d58d4e1b1774b4c9158b1f4c'
>>> h.digest_size
20
>>> len(h.digest())
20
```

不同摘要大小的哈希对象具有完全不同的输出（较短哈希值 *并非* 较长哈希值的前缀）；即使输出长度相同，BLAKE2b 和 BLAKE2s 也会产生不同的输出:

```
>>> from hashlib import blake2b, blake2s
>>> blake2b(digest_size=10).hexdigest()
'6fa1d8fcfd719046d762'
>>> blake2b(digest_size=11).hexdigest()
'eb6ec15daf9546254f0809'
>>> blake2s(digest_size=10).hexdigest()
'1bf21a98c78a1c376ae9'
>>> blake2s(digest_size=11).hexdigest()
'567004bf96e4a25773ebf4'
```

密钥哈希

Keyed hashing can be used for authentication as a faster and simpler replacement for Hash-based message authentication code (HMAC). BLAKE2 can be securely used in prefix-MAC mode thanks to the indifferentiability property inherited from BLAKE.

这个例子演示了如何使用密钥 `b'pseudorandom key'` 来为 `b'message data'` 获取一个（十六进制编码的）128 位验证代码:

```
>>> from hashlib import blake2b
>>> h = blake2b(key=b'pseudorandom key', digest_size=16)
>>> h.update(b'message data')
>>> h.hexdigest()
'3d363ff7401e02026f4a4687d4863ced'
```

作为实际的例子，一个 Web 应用可为发送给用户的 cookies 进行对称签名，并在之后对其进行验证以确保它们没有被篡改:

```
>>> from hashlib import blake2b
>>> from hmac import compare_digest
>>>
>>> SECRET_KEY = b'pseudorandomly generated server secret key'
>>> AUTH_SIZE = 16
>>>
>>> def sign(cookie):
...     h = blake2b(digest_size=AUTH_SIZE, key=SECRET_KEY)
...     h.update(cookie)
...     return h.hexdigest().encode('utf-8')
>>>
>>> def verify(cookie, sig):
...     good_sig = sign(cookie)
...     return compare_digest(good_sig, sig)
>>>
>>> cookie = b'user-alice'
>>> sig = sign(cookie)
>>> print("{0},{1}".format(cookie.decode('utf-8'), sig))
user-alice,b'43b3c982cf697e0c5ab22172d1ca7421'
>>> verify(cookie, sig)
True
>>> verify(b'user-bob', sig)
False
>>> verify(cookie, b'0102030405060708090a0b0c0d0e0f00')
False
```

即使存在原生的密钥哈希模式，BLAKE2 也同样可在*hmac* 模块的 HMAC 构造过程中使用:

```
>>> import hmac, hashlib
>>> m = hmac.new(b'secret key', digestmod=hashlib.blake2s)
>>> m.update(b'message')
>>> m.hexdigest()
'e3c8102868d28b5ff85fc35dda07329970d1a01e273c37481326fe0c861c8142'
```

随机哈希

用户可通过设置 *salt* 形参来为哈希函数引入随机化。随机哈希适用于防止对数字签名中使用的哈希函数进行碰撞攻击。

> 随机哈希被设计用来处理当一方（消息准备者）要生成由另一方（消息签名者）进行签名的全部或部分消息的情况。如果消息准备者能够找到加密哈希函数的碰撞现象（即两条消息产生相同的哈希值），则他们就可以准备将产生相同哈希值和数字签名但却具有不同结果的有意义的消息版本（例如向某个账户转入 $1,000,000 而不是 $10）。加密哈希函数的设计都是以防碰撞性能为其主要目标之一的，但是当前针对加密哈希函数的集中攻击可能导致特定加密哈希函数所提供的防碰撞性能低于预期。随机哈希为签名者提供了额外的保护，可以降低准备者在数字签名生成过程中使得两条或更多条消息最终产生相同哈希值的可能性 --- 即使为特定哈希函数找到碰撞现象是可行的。但是，当消息的所有部分均由签名者准备时，使用随机哈希可能降低数字签名所提供的安全性。
>
> (NIST SP-800-106 "数字签名的随机哈希")

在 BLAKE2 中，盐值会在初始化期间作为对哈希函数的一次性输入而不是对每个压缩函数的输入来处理。

警告： 使用 BLAKE2 或任何其他通用加密哈希函数例如 SHA-256 进行 *加盐哈希* (或纯哈希) 并不适用于哈希密码。请参阅 BLAKE2 FAQ 了解更多信息。

```
>>> import os
>>> from hashlib import blake2b
>>> msg = b'some message'
>>> # Calculate the first hash with a random salt.
>>> salt1 = os.urandom(blake2b.SALT_SIZE)
>>> h1 = blake2b(salt=salt1)
>>> h1.update(msg)
>>> # Calculate the second hash with a different random salt.
>>> salt2 = os.urandom(blake2b.SALT_SIZE)
>>> h2 = blake2b(salt=salt2)
>>> h2.update(msg)
>>> # The digests are different.
>>> h1.digest() != h2.digest()
True
```

个性化

出于不同的目的强制让哈希函数为相同的输入生成不同的摘要有时也是有用的。正如 Skein 哈希函数的作者所言:

> 我们建议所有应用设计者慎重考虑这种做法；我们已看到有许多协议在协议的某一部分中计算出来的哈希值在另一个完全不同的部分中也可以被使用，因为两次哈希计算是针对类似或相关的数据进行的，这样攻击者可以强制应用为相同的输入生成哈希值。个性化协议中所使用的每个哈希函数将有效地阻止这种类型的攻击。
>
> (Skein 哈希函数族, p. 21)

BLAKE2 可通过向 *person* 参数传入字节串来进行个性化:

```
>>> from hashlib import blake2b
>>> FILES_HASH_PERSON = b'MyApp Files Hash'
>>> BLOCK_HASH_PERSON = b'MyApp Block Hash'
>>> h = blake2b(digest_size=32, person=FILES_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'20d9cd024d4fb086aae819a1432dd2466de12947831b75c5a30cf2676095d3b4'
>>> h = blake2b(digest_size=32, person=BLOCK_HASH_PERSON)
>>> h.update(b'the same content')
>>> h.hexdigest()
'cf68fb5761b9c44e7878bfb2c4c9aea52264a80b75005e65619778de59f383a3'
```

个性化配合密钥模式也可被用来从单个密钥派生出多个不同密钥。

```
>>> from hashlib import blake2s
>>> from base64 import b64decode, b64encode
>>> orig_key = b64decode(b'Rm5EPJai72qcK3RGBpW3vPNfZy5OZothY+kHY6h21KM=')
>>> enc_key = blake2s(key=orig_key, person=b'kEncrypt').digest()
>>> mac_key = blake2s(key=orig_key, person=b'kMAC').digest()
>>> print(b64encode(enc_key).decode('utf-8'))
rbPb15S/Z9t+agffno5wuhB77VbRi6F9Iv2qIxU7WHw=
>>> print(b64encode(mac_key).decode('utf-8'))
G9GtHFE1YluXY1zWPlYk1e/nWfu0WSEb0KRcjhDeP/o=
```

树形模式

以下是对包含两个叶子节点的最小树进行哈希的例子:

```
  10
 /  \
00  01
```

这个例子使用 64 字节内部摘要，返回 32 字节最终摘要:

```
>>> from hashlib import blake2b
>>>
>>> FANOUT = 2
>>> DEPTH = 2
>>> LEAF_SIZE = 4096
>>> INNER_SIZE = 64
>>>
>>> buf = bytearray(6000)
```

(下页继续)

(繼續上一頁)

```
>>>
>>> # Left leaf
... h00 = blake2b(buf[0:LEAF_SIZE], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=0, last_node=False)
>>> # Right leaf
... h01 = blake2b(buf[LEAF_SIZE:], fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=1, node_depth=0, last_node=True)
>>> # Root node
... h10 = blake2b(digest_size=32, fanout=FANOUT, depth=DEPTH,
...               leaf_size=LEAF_SIZE, inner_size=INNER_SIZE,
...               node_offset=0, node_depth=1, last_node=True)
>>> h10.update(h00.digest())
>>> h10.update(h01.digest())
>>> h10.hexdigest()
'3ad2a9b37c6070e374c7a8c508fe20ca86b6ed54e286e93a0318e95e881db5aa'
```

开发人员

BLAKE2 是由 *Jean-Philippe Aumasson*, *Samuel Neves*, *Zooko Wilcox-O'Hearn* 和 *Christian Winnerlein* 基于 *Jean-Philippe Aumasson*, *Luca Henzen*, *Willi Meier* 和 *Raphael C.-W. Phan* 所创造的 SHA-3 入围方案 BLAKE 进行设计的。

它使用的核心算法来自由 *Daniel J. Bernstein* 所设计的 ChaCha 加密。

stdlib 实现是基于 pyblake2 模块的。它由 *Dmitry Chestnykh* 在 *Samuel Neves* 所编写的 C 实现的基础上编写。此文档拷贝自 pyblake2 并由 *Dmitry Chestnykh* 撰写。

C 代码由 *Christian Heimes* 针对 Python 进行了部分的重写。

以下公共领域贡献同时适用于 C 哈希函数实现、扩展代码和本文档:

> 在法律许可的范围内，作者已将此软件的全部版权以及关联和邻接权利贡献到全球公共领域。此软件的发布不附带任何担保。
>
> 你应该已收到此软件附带的 CC0 公共领域专属证书的副本。如果没有，请参阅 https://creativecommons.org/publicdomain/zero/1.0/。

根据创意分享公共领域贡献 1.0 通用规范，下列人士为此项目的开发提供了帮助或对公共领域的修改作出了贡献:

- *Alexandr Sokolovskiy*

也参考:

模块*hmac* 使用哈希运算来生成消息验证代码的模块。

模块*base64* 针对非二进制环境对二进制哈希值进行编辑的另一种方式。

https://blake2.net BLAKE2 官方网站

https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf 有关安全哈希算法的 FIPS 180-2 出版物。

https://en.wikipedia.org/wiki/Cryptographic_hash_function#Cryptographic_hash_algorithms 包含关于哪些算法存在已知问题以及对其使用所造成的影响的信息的 Wikipedia 文章。

https://www.ietf.org/rfc/rfc2898.txt PKCS #5: 基于密码的加密规范描述 2.0 版

15.2 hmac --- 基于密钥的消息验证

源代码: Lib/hmac.py

此模块实现了 HMAC 算法，算法的描述参见 **RFC 2104**。

hmac.**new**(*key*, *msg=None*, *digestmod=''*)
返回一个新的 hmac 对象。*key* 是一个指定密钥的 bytes 或 bytearray 对象。如果提供了 *msg*，将会调用 `update(msg)` 方法。*digestmod* 为 HMAC 对象所用的摘要名称、摘要构造器或模块。它可以是适用于*hashlib.new()* 的任何名称。虽然该参数位置靠后，但它却是必须的。

3.4 版更變: 形参 *key* 可以为 bytes 或 bytearray 对象。形参 *msg* 可以为*hashlib* 所支持的任意类型。形参 *digestmod* 可以为某种哈希算法的名称。

Deprecated since version 3.4, removed in version 3.8: MD5 作为 *digestmod* 的隐式默认摘要已被弃用。digestmod 形参现在是必须的。请将其作为关键字参数传入以避免当你没有初始 msg 时将导致的麻烦。

hmac.**digest**(*key*, *msg*, *digest*)
基于给定密钥 *key* 和 *digest* 返回 *msg* 的摘要。此函数等价于 `HMAC(key, msg, digest).digest()`，但使用了优化的 C 或内联实现，对放入内存的消息能处理得更快。形参 *key*, *msg* 和 *digest* 具有与*new()* 中相同的含义。

作为 CPython 的实现细节，优化的 C 实现仅当 *digest* 为字符串并且是一个 OpenSSL 所支持的摘要算法的名称时才会被使用。

3.7 版新加入.

HMAC 对象具有下列方法:

HMAC.**update**(*msg*)
用 *msg* 来更新 hmac 对象。重复调用相当于单次调用并传入所有参数的拼接结果: `m.update(a); m.update(b)` 等价于 `m.update(a + b)`。

3.4 版更變: 形参 *msg* 可以为*hashlib* 所支持的任何类型。

HMAC.**digest**()
返回当前已传给*update()* 方法的字节串数据的摘要。这个字节串数据的长度将与传给构造器的摘要的长度 *digest_size* 相同。它可以包含非 ASCII 的字节，包括 NUL 字节。

> **警告:** 在验证例程运行期间将*digest()* 的输出与外部提供的摘要进行比较时，建议使用*compare_digest()* 函数而不是 == 运算符以减少定时攻击防御力的不足。

HMAC.**hexdigest**()
类似于*digest()* 但摘要会以两倍长度字符串的形式返回，其中仅包含十六进制数码。这可以被用于在电子邮件或其他非二进制环境中安全地交换数据值。

> **警告:** 在验证例程运行期间将*hexdigest()* 的输出与外部提供的摘要进行比较时，建议使用*compare_digest()* 函数而不是 == 运算符以减少定时攻击防御力的不足。

HMAC.**copy**()
返回 hmac 对象的副本（“克隆）。这可被用来高效地计算共享相同初始子串的数据的摘要。

hash 对象具有以下属性:

HMAC.**digest_size**
以字节表示的结果 HMAC 摘要的大小。

HMAC.**block_size**
以字节表示的哈希算法的内部块大小。

3.4 版新加入.

HMAC.**name**
HMAC 的规范名称，总是为小写形式，例如 hmac-md5。

3.4 版新加入.

3.9 版後已🄵用: 未写入文档的属性 HMAC.digest_cons, HMAC.inner 和 HMAC.outer 属于内部实现细节，将在 Python 3.10 中被移除。

这个模块还提供了下列辅助函数:

hmac.**compare_digest**(*a*, *b*)
返回 a == b。此函数使用一种经专门设计的方式通过避免基于内容的短路行为来防止定时分析，使得它适合处理密码。*a* 和 *b* 必须为相同的类型：或者是*str* (仅限 ASCII 字符，如*HMAC.hexdigest()* 的返回值)，或者是*bytes-like object*。

備🄵: 如果 *a* 和 *b* 具有不同的长度，或者如果发生了错误，定时攻击在理论上可以获取有关 *a* 和 *b* 的类型和长度信息—但不能获取它们的值。

3.3 版新加入.

3.9 版更變: 此函数在可能的情况下会在内部使用 OpenSSL 的 CRYPTO_memcmp()。

也參考:

模块*hashlib* 提供安全哈希函数的 Python 模块。

15.3 secrets --- 🄵生用於管理機密的安全亂數

3.6 版新加入.

原始碼: Lib/secrets.py

---

secrets 模組可用於🄵生高加密🄵度的亂數，適合用來管理諸如密碼、帳號認證、安全性權杖（security tokens）這類資料，以及管理其他相關的機密資料。

尤其應優先使用*secrets* 作🄵預設來替代*random* 模組中的預設🄵亂數🄵生器（pseudo-random number generator），該模組被設計用於建模和模擬，而非用於安全性和加密。

也參考:

PEP 506

15.3.1 亂數

The *secrets* module provides access to the most secure source of randomness that your operating system provides.

class secrets.**SystemRandom**
一個用來🄵生亂數的類🄵，用的是作業系統提供的最高品質來源。請參🄵*random.SystemRandom* 以獲取更多細節。

secrets.**choice**(*sequence*)
從一非空序列中，回傳一個隨機選取的元素。

secrets.**randbelow**(*n*)
回傳一個 [0, *n*) 範圍之🄵的隨機整數。

secrets.**randbits**(*k*)
回傳一個具 *k* 個隨機位元的整數。

15.3.2 🄵生權杖（token）

secrets 模組提供了一些🄵生安全性權杖的函式，適合用於諸如重設密碼、難以猜測的 URL，或類似的應用。

secrets.**token_bytes**([*nbytes=None*])
回傳一個隨機位元組字串，其中含有 *nbytes* 位元組的數字。如果 *nbytes* 🄵 None 或未提供，則會使用一合理預設值。

```
>>> token_bytes(16)
b'\xebr\x17D*t\xae\xd4\xe3S\xb6\xe2\xebP1\x8b'
```

secrets.**token_hex**([*nbytes=None*])
回傳一以十六進位表示的隨機字串。字串具有 *nbytes* 個隨機位元組，每個位元組會轉成兩個十六進位的數字。如果 *nbytes* 🄵 None 或未提供，則會使用一個合理的預設值。

```
>>> token_hex(16)
'f9bf78b9a18ce6d46a0cd2b0b86df9da'
```

secrets.**token_urlsafe**([*nbytes=None*])
回傳一個 URL 安全的隨機文本字串，包含 *nbytes* 個隨機位元組。文本將使用 Base64 編碼，因此平均來🄵每個位元組會對應到約 1.3 個字元。如果 *nbytes* 🄵 None 或未提供，則會使用一個合理的預設值。

```
>>> token_urlsafe(16)
'Drmhze6EPcv0fN_81Bj-nA'
```

權杖應當使用多少個位元組？

🄵了在面對 暴力攻擊 時能保證安全，權杖必須具有足🄵的隨機性。不幸的是，對隨機性是否足🄵的標準，會隨著電腦越來越🄵大🄵能🄵在更短時間🄵進行更多猜測而不斷提高。在 2015 年時，人們認🄵 32 位元組（256 位元）的隨機性對於*secrets* 模組所預期的一般使用場景來🄵是足🄵的。

對於想自行管理權杖長度的使用者，你可以對各種 token_* 函式明白地指定*int* 引數（argument）來指定權杖要使用的隨機性程度。該引數以位元組數來表示要使用的隨機性程度。

否則，如未提供引數，或者如果引數🄵 None，則 token_* 函式則會使用一個合理的預設值。

備🄵： 該預設值可能在任何時候被改變，包括在維護版本更新的時候。

15.3.3 其他函式

secrets.**compare_digest**(*a*, *b*)
如果字串 *a* 與 *b* 相等則回傳 True，否則回傳 False，這樣的處理方式可降低 時序攻擊 的風險。請參🄵*hmac.compare_digest()* 以了解更多細節。

15.3.4 應用技巧和典範實務（best practices）

本節展示了一些使用*secrets* 來管理基本安全等級的應用技巧和典範實務。

🄵生八個字元長的字母數字密碼：

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
password = ''.join(secrets.choice(alphabet) for i in range(8))
```

備🄴： 應用程式不能 以可復原的格式存儲密碼，無論是用純文本還是經過加密。它們應當先加鹽（salt），再使用高加密🄵度的單向（不可逆）雜🄵函數來🄵生雜🄵值。

🄵生十個字元長的字母數字密碼，其中包含至少一個小寫字母，至少一個大寫字母以及至少三個數字：

```
import string
import secrets
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(secrets.choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

🄵生 XKCD 風格的 passphrase：

```
import secrets
# On standard Linux systems, use a convenient dictionary file.
# Other platforms may need to provide their own word-list.
with open('/usr/share/dict/words') as f:
    words = [word.strip() for word in f]
    password = ' '.join(secrets.choice(words) for i in range(4))
```

🄵生難以猜測的暫時性 URL，🄵含回復密碼時所用的一個安全性權杖：

```
import secrets
url = 'https://example.com/reset=' + secrets.token_urlsafe()
```

CHAPTER 16

通用操作系统服务

本章中描述的各模块提供了在（几乎）所有的操作系统上可用的操作系统特性的接口，例如文件和时钟。这些接口通常以 Unix 或 C 接口为参照对象设计，不过在大多数其他系统上也可用。这里有一个概述：

16.1 os --- 多种操作系统接口

源代码： Lib/os.py

本模块提供了一种使用与操作系统相关的功能的便捷式途径。如果你只是想读写一个文件，请参阅*open()*，如果你想操作文件路径，请参阅*os.path* 模块，如果你想读取通过命令行给出的所有文件中的所有行，请参阅*fileinput* 模块。为了创建临时文件和目录，请参阅*tempfile* 模块，对于高级文件和目录处理，请参阅*shutil* 模块。

关于这些函数的可用性的说明：

- Python 中所有依赖于操作系统的内置模块的设计都是这样，只要不同的操作系统某一相同的功能可用，它就使用相同的接口。例如，函数 `os.stat(path)` 以相同的格式返回关于 *path* 的状态信息（该格式源于 POSIX 接口）。
- 特定于某一操作系统的扩展通过操作*os* 模块也是可用的，但是使用它们当然是对可移植性的一种威胁。
- 所有接受路径或文件名的函数都同时支持字节串和字符串对象，并在返回路径或文件名时使用相应类型的对象作为结果。
- 在 VxWorks 系统上，os.fork, os.execv 和 os.spawn*p* 不被支持。

備🄴： 如果使用无效或无法访问的文件名与路径，或者其他类型正确但操作系统不接受的参数，此模块的所有函数都抛出*OSError* （或者它的子类）。

exception `os.`**error**
内建的*OSError* 异常的一个别名。

os.**name**
导入的依赖特定操作系统的模块的名称。以下名称目前已注册: 'posix', 'nt', 'java'.

也参考:

sys.platform 有更详细的描述. *os.uname()* 只给出系统提供的版本信息。

platform 模块对系统的标识有更详细的检查。

16.1.1 文件名，命令行参数，以及环境变量。

在 Python 中，使用字符串类型表示文件名、命令行参数和环境变量。在某些系统上，在将这些字符串传递给操作系统之前，必须将这些字符串解码为字节。Python 使用文件系统编码来执行此转换（请参阅*sys.getfilesystemencoding()* ）。

3.1 版更變: 在某些系统上，使用文件系统编码进行转换可能会失败。在这种情况下，Python 会使用*代理转义编码错误处理器*，这意味着在解码时，不可解码的字节被 Unicode 字符 U+DCxx 替换，并且这些字节在编码时再次转换为原始字节。

文件系统编码必须保证成功解码小于 128 的所有字节。如果文件系统编码无法提供此保证，API 函数可能会引发 UnicodeErrors 。

16.1.2 进程参数

这些函数和数据项提供了操作当前进程和用户的信息。

os.**ctermid**()
返回与进程控制终端对应的文件名。

可用性: Unix。

os.**environ**
A *mapping* object where keys and values are strings that represent the process environment. For example, `environ['HOME']` is the pathname of your home directory (on some platforms), and is equivalent to `getenv("HOME")` in C.

This mapping is captured the first time the *os* module is imported, typically during Python startup as part of processing `site.py`. Changes to the environment made after this time are not reflected in *os.environ*, except for changes made by modifying *os.environ* directly.

该映射除了可以用于查询环境外，还能用于修改环境。当该映射被修改时，将自动调用*putenv()*。

在 Unix 系统上，键和值会使用*sys.getfilesystemencoding()* 和 'surrogateescape' 的错误处理。如果你想使用其他的编码，使用*environb*。

備🄵: Calling *putenv()* directly does not change *os.environ*, so it's better to modify *os.environ*.

備🄵: 在某些平台上，包括 FreeBSD 和 macOS，设置 `environ` 可能导致内存泄漏。请参阅 `putenv()` 的系统文档。

You can delete items in this mapping to unset environment variables. *unsetenv()* will be called automatically when an item is deleted from *os.environ*, and when one of the `pop()` or `clear()` methods is called.

3.9 版更變: 已更新并支持了 **PEP 584** 的合并 (|) 和更新 (|=) 运算符。

os.**environb**
Bytes version of *environ*: a *mapping* object where both keys and values are *bytes* objects representing the process environment. *environ* and *environb* are synchronized (modifying *environb* updates *environ*, and vice versa).

只有在*supports_bytes_environ* 为 True 的时候*environb* 才是可用的。

3.2 版新加入.

3.9 版更變: 已更新并支持了 **PEP 584** 的合并 (|) 和更新 (|=) 运算符。

os.**chdir**(*path*)
os.**fchdir**(*fd*)
os.**getcwd**()
以上函数请参阅文件和目录 。

os.**fsencode**(*filename*)
将路径类 *filename* 编码为文件系统编码，使用 'surrogateescape' 错误回调方法，在 Windows 上会使用 'strict'，*bytes* 类型则原样返回。

fsdecode() 是此函数的逆向函数。

3.2 版新加入.

3.6 版更變: 增加对实现了*os.PathLike* 接口的对象的支持。

os.**fsdecode**(*filename*)
将路径类 *filename* 从文件系统编码方式解码，编码方式应使用的是 'surrogateescape' 错误回调方法，在 Windows 上应使用的是 'strict' 方法。*str* 字符串则原样返回。

fsencode() 是此函数的逆向函数。

3.2 版新加入.

3.6 版更變: 增加对实现了*os.PathLike* 接口的对象的支持。

os.**fspath**(*path*)
返回路径的文件系统表示。

如果传入的是*str* 或*bytes* 类型的字符串，将原样返回。否则*__fspath__()* 将被调用，如果得到的是一个*str* 或*bytes* 类型的对象，那就返回这个值。其他所有情况则会抛出*TypeError* 异常。

3.6 版新加入.

class os.**PathLike**
某些对象用于表示文件系统中的路径（如*pathlib.PurePath* 对象），本类是这些对象的抽象基类。

3.6 版新加入.

abstractmethod **__fspath__**()
返回当前对象的文件系统表示。

这个方法只应该返回一个*str* 字符串或*bytes* 字节串，请优先选择*str* 字符串。

os.**getenv**(*key*, *default=None*)
Return the value of the environment variable *key* if it exists, or *default* if it doesn't. *key*, *default* and the result are str. Note that since *getenv()* uses *os.environ*, the mapping of *getenv()* is similarly also captured on import, and the function may not reflect future environment changes.

在 Unix 系统上，键和值会使用*sys.getfilesystemencoding()* 和 "'surrogateescape'" 错误处理进行解码。如果你想使用其他的编码，使用*os.getenvb()*。

可用性: 大部分的 Unix 系统，Windows。

os.**getenvb**(*key*, *default=None*)
Return the value of the environment variable *key* if it exists, or *default* if it doesn't. *key*, *default* and the result are bytes. Note that since *getenvb()* uses *os.environb*, the mapping of *getenvb()* is similarly also captured on import, and the function may not reflect future environment changes.

getenvb() 仅在*supports_bytes_environ* 为 True 时可用。

可用性: 大部分的 Unix 系统。

3.2 版新加入.

os.**get_exec_path**(*env=None*)
返回将用于搜索可执行文件的目录列表，与在外壳程序中启动一个进程时相似。指定的 *env* 应为用于搜索 PATH 的环境变量字典。默认情况下，当 *env* 为 None 时，将会使用*environ* 。

3.2 版新加入.

os.**getegid**()
返回当前进程的有效组 ID。对应当前进程执行文件的"set id" 位。

可用性: Unix。

os.**geteuid**()
返回当前进程的有效用户 ID。

可用性: Unix。

os.**getgid**()
返回当前进程的实际组 ID。

可用性: Unix。

os.**getgrouplist**(*user*, *group*)
Return list of group ids that *user* belongs to. If *group* is not in the list, it is included; typically, *group* is specified as the group ID field from the password record for *user*, because that group ID will otherwise be potentially omitted.

可用性: Unix。

3.3 版新加入.

os.**getgroups**()
返回当前进程对应的组 ID 列表

可用性: Unix。

備𝔽: 在 macOS 中，*getgroups()* 会和其他 Unix 平台有所不同。如果 Python 解释器是在 10.5 或更早版本中部署的，则*getgroups()* 会返回与当前用户进程相关联的有效组 ID 列表；该列表受限于系统预定义的条目数量，通常为 16，并且在适当的权限下还可通过调用*setgroups()* 来修改。如果是在高于 10.5 的版本中部署的，则*getgroups()* 会返回与进程的有效用户 ID 相关联的当前组访问列表；组访问列表可能会在进程的生命周期之内发生改变，它不会受对*setgroups()* 的调用影响，且其长度也不被限制为 16。部署目标值 MACOSX_DEPLOYMENT_TARGET 可以通过*sysconfig.get_config_var()* 来获取。

---

os.**getlogin**()
返回通过控制终端进程进行登录的用户名。在多数情况下，使用*getpass.getuser()* 会更有效，因为后者会通过检查环境变量 LOGNAME 或 USERNAME 来查找用户，再由 pwd.getpwuid(os.getuid())[0] 来获取当前用户 ID 的登录名。

可用性: Unix, Windows。

os.**getpgid**(*pid*)
根据进程 id *pid* 返回进程的组 ID 列表。如果 *pid* 为 0，则返回当前进程的进程组 ID 列表

可用性: Unix。

os.**getpgrp**()
返回当时进程组的 ID

可用性: Unix。

os.**getpid**()
返回当前进程 ID

os.**getppid**()
返回父进程 ID。当父进程已经结束，在 Unix 中返回的 ID 是初始进程 (1) 中的一个，在 Windows 中仍然是同一个进程 ID，该进程 ID 有可能已经被进行进程所占用。

可用性: Unix, Windows。

3.2 版更變: 添加 WIndows 的支持。

os.**getpriority**(*which*, *who*)
获取程序调度优先级。*which* 参数值可以是*PRIO_PROCESS*，*PRIO_PGRP*，或*PRIO_USER* 中的一个，*who* 是相对于 *which* (*PRIO_PROCESS* 的进程标识符，*PRIO_PGRP* 的进程组标识符和*PRIO_USER* 的用户 ID)。当 *who* 为 0 时（分别）表示调用的进程，调用进程的进程组或调用进程所属的真实用户 ID。

可用性: Unix。

3.3 版新加入.

os.**PRIO_PROCESS**
os.**PRIO_PGRP**
os.**PRIO_USER**
函数*getpriority()* 和*setpriority()* 的参数。

可用性: Unix。

3.3 版新加入.

os.**getresuid**()
返回一个由 (ruid, euid, suid) 所组成的元组，分别表示当前进程的真实用户 ID，有效用户 ID 和甲暫存用户 ID。

可用性: Unix。

3.2 版新加入.

os.**getresgid**()
返回一个由 (rgid, egid, sgid) 所组成的元组，分别表示当前进程的真实组 ID，有效组 ID 和暫存组 ID。

可用性: Unix。

3.2 版新加入.

os.**getuid**()
返回当前进程的真实用户 ID。

可用性: Unix。

os.**initgroups**(*username*, *gid*)
调用系统 initgroups()，使用指定用户所在的所有值来初始化组访问列表，包括指定的组 ID。

可用性: Unix。

3.2 版新加入.

os.**putenv**(*key*, *value*)
将名为 *key* 的环境变量值设置为 *value*。该变量名修改会影响由*os.system()*，*popen()* ，*fork()* 和*execv()* 发起的子进程。

Assignments to items in *os.environ* are automatically translated into corresponding calls to *putenv()*; however, calls to *putenv()* don't update *os.environ*, so it is actually preferable to assign to items of *os.environ*. This also applies to *getenv()* and *getenvb()*, which respectively use *os.environ* and *os.environb* in their implementations.

備🄵: 在某些平台上，包括 FreeBSD 和 macOS，设置 `environ` 可能导致内存泄漏。请参阅 `putenv()` 的系统文档。

引发一个审计事件 `os.putenv`，附带参数 `key`, `value`。

3.9 版更變: 该函数现在总是可用。

`os.`**`setegid`**`(`*egid*`)`
设置当前进程的有效组 ID。

可用性: Unix。

`os.`**`seteuid`**`(`*euid*`)`
设置当前进程的有效用户 ID。

可用性: Unix。

`os.`**`setgid`**`(`*gid*`)`
设置当前进程的组 ID。

可用性: Unix。

`os.`**`setgroups`**`(`*groups*`)`
将 *group* 参数值设置为与当进程相关联的附加组 ID 列表。*group* 参数必须为一个序列，每个元素应为每个组的数字 ID。该操作通常只适用于超级用户。

可用性: Unix。

備🄵: 在 macOS 中，*groups* 的长度不能超过系统定义的最大有效组 ID 数量，通常为 16。对于未返回与调用 setgroups() 产生的相同组列表的情况，请参阅*getgroups()* 的文档。

`os.`**`setpgrp`**`()`
根据已实现的版本（如果有）来调用系统 `setpgrp()` 或 `setpgrp(0, 0)`。相关说明，请参考 Unix 手册。

可用性: Unix。

`os.`**`setpgid`**`(`*pid*, *pgrp*`)`
使用系统调用 `setpgid()`，将 *pid* 对应进程的组 ID 设置为 *pgrp*。相关说明，请参考 Unix 手册。

可用性: Unix。

`os.`**`setpriority`**`(`*which*, *who*, *priority*`)`
设置程序调度优先级。*which* 的值为*PRIO_PROCESS*, *PRIO_PGRP* 或*PRIO_USER* 之一，而 *who* 会相对于 *which* (*PRIO_PROCESS* 的进程标识符, *PRIO_PGRP* 的进程组标识符和*PRIO_USER* 的用户 ID) 被解析。*who* 值为零 (分别) 表示调用进程，调用进程的进程组或调用进程的真实用户 ID。*priority* 是范围在 -20 至 19 的值。默认优先级为 0；较小的优先级数值会更优先被调度。

可用性: Unix。

3.3 版新加入.

`os.`**`setregid`**`(`*rgid*, *egid*`)`
设置当前进程的真实和有效组 ID。

可用性: Unix。

os.**setresgid**(*rgid*, *egid*, *sgid*)
设置当前进程的真实，有效和暂存组 ID。

可用性: Unix。

3.2 版新加入.

os.**setresuid**(*ruid*, *euid*, *suid*)
设置当前进程的真实，有效和暂存用户 ID。

可用性: Unix。

3.2 版新加入.

os.**setreuid**(*ruid*, *euid*)
设置当前进程的真实和有效用户 ID。

可用性: Unix。

os.**getsid**(*pid*)
调用系统调用 getsid()。相关说明，请参考 Unix 手册。

可用性: Unix。

os.**setsid**()
使用系统调用 getsid()。相关说明，请参考 Unix 手册。

可用性: Unix。

os.**setuid**(*uid*)
设置当前进程的用户 ID。

可用性: Unix。

os.**strerror**(*code*)
根据 *code* 中的错误码返回错误消息。在某些平台上当给出未知错误码时 strerror() 将返回 NULL 并会引发*ValueError*。

os.**supports_bytes_environ**
如果操作系统上原生环境类型是字节型则为 True (例如在 Windows 上为 False)。

3.2 版新加入.

os.**umask**(*mask*)
设定当前数值掩码并返回之前的掩码。

os.**uname**()
返回当前操作系统的识别信息。返回值是一个有 5 个属性的对象：

- sysname - 操作系统名
- nodename - 机器在网络上的名称（需要先设定）
- release - 操作系统发行信息
- version - 操作系统版本信息
- machine - 硬件标识符

为了向后兼容，该对象也是可迭代的，像是一个按照 sysname，nodename，release，version，和 machine 顺序组成的元组。

有些系统会将 nodename 截短为 8 个字符或截短至前缀部分；获取主机名的一个更好方式是*socket.gethostname()* 或甚至可以用 socket.gethostbyaddr(socket.gethostname())。

可用性: 较新的 Unix 版本。

3.3 版更變: 返回结果的类型由元组变成一个类似元组的对象，同时具有命名的属性。

os.**unsetenv**(*key*)
取消设置（删除）名为 *key* 的环境变量。变量名的改变会影响由*os.system()*，*popen()*，*fork()* 和*execv()* 触发的子进程。

Deletion of items in *os.environ* is automatically translated into a corresponding call to *unsetenv()*; however, calls to *unsetenv()* don't update *os.environ*, so it is actually preferable to delete items of *os.environ*.

引发一个审计事件 os.unsetenv，附带参数 key。

3.9 版更變: 该函数现在总是可用，并且在 Windows 上也可用。

16.1.3 创建文件对象

这些函数创建新的*file objects* 。（参见*open()* 以获取打开文件描述符的相关信息。）

os.**fdopen**(*fd*, **args*, ***kwargs*)
返回打开文件描述符 *fd* 对应文件的对象。类似内建*open()* 函数，二者接受同样的参数。不同之处在于*fdopen()* 第一个参数应该为整数。

16.1.4 文件描述符操作

这些函数对文件描述符所引用的 I/O 流进行操作。

文件描述符是一些小的整数，对应于当前进程所打开的文件。例如，标准输入的文件描述符通常是 0，标准输出是 1，标准错误是 2。之后被进程打开的文件的文件描述符会被依次指定为 3，4，5 等。“文件描述符”这个词有点误导性，在 Unix 平台中套接字和管道也被文件描述符所引用。

当需要时，可以用*fileno()* 可以获得*file object* 所对应的文件描述符。需要注意的是，直接使用文件描述符会绕过文件对象的方法，会忽略如数据内部缓冲等情况。

os.**close**(*fd*)
关闭文件描述符 *fd*。

> **備𝔽:** 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。若要关闭由内建函数*open()*、*popen()* 或*fdopen()* 返回的” 文件对象”，则应使用其相应的*close()* 方法。

os.**closerange**(*fd_low*, *fd_high*)
关闭从 *fd_low* （包括）到 *fd_high* （排除）间的文件描述符，并忽略错误。类似（但快于）：

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.**copy_file_range**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*)
从文件描述符 *src* 复制 *count* 字节，从偏移量 *offset_src* 开始读取，到文件描述符 *dst*，从偏移量 *offset_dst* 开始写入。如果 *offset_src* 为 None，则 *src* 将从当前位置开始读取；*offset_dst* 同理。*src* 和 *dst* 指向的文件必须处于相同的文件系统，否则将会抛出一个*errno* 被设为*errno.EXDEV* 的*OSError* 。

此复制的完成没有额外的从内核到用户空间再回到内核的数据转移花费。另外，一些文件系统可能实现额外的优化。完成复制就如同打开两个二进制文件一样。

返回值是复制的字节的数目。这可能低于需求的数目。

Availability: Linux kernel >= 4.5 或 glibc >= 2.27。

3.8 版新加入.

os.**device_encoding**(*fd*)
如果连接到终端，则返回一个与 *fd* 关联的设备描述字符，否则返回*None*。

os.**dup**(*fd*)
返回一个文件描述符 *fd* 的副本。该文件描述符的副本是不可继承的。

在 Windows 中，当复制一个标准流（0: stdin, 1: stdout, 2: stderr）时，新的文件描述符是可继承的。

3.4 版更變: 新的文件描述符现在是不可继承的。

os.**dup2**(*fd*, *fd2*, *inheritable=True*)
把文件描述符 *fd* 复制为 *fd2*，必要时先关闭后者。返回 *fd2*。新的文件描述符默认是可继承的，除非在 *inheritable* 为 `False` 时，是不可继承的。

3.4 版更變: 添加可选参数 *inheritable*。

3.7 版更變: 成功时返回 *fd2*，以过去的版本中，总是返回 `None`。

os.**fchmod**(*fd*, *mode*)
将 *fd* 指定文件的权限状态修改为 *mode*。可以参考*chmod()* 中列出 *mode* 的可用值。从 Python 3.3 开始，这相当于 `os.chmod(fd, mode)`。

引发一个审计事件 `os.chmod`，附带参数 `path`、`mode`、`dir_fd`。

可用性: Unix。

os.**fchown**(*fd*, *uid*, *gid*)
分别将 *fd* 指定文件的所有者和组 ID 修改为 *uid* 和 *gid* 的值。若不想变更其中的某个 ID，可将相应值设为 -1。参考*chown()*。从 Python 3.3 开始，这相当于 `os.chown(fd, uid, gid)`。

引发一个审计事件 `os.chown`，附带参数 `path`、`uid`、`gid`、`dir_fd`。

可用性: Unix。

os.**fdatasync**(*fd*)
强制将文件描述符 *fd* 指定文件写入磁盘。不强制更新元数据。

可用性: Unix。

備F: 该功能在 MacOS 中不可用。

---

os.**fpathconf**(*fd*, *name*)
返回与打开的文件有关的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX.1，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 `pathconf_names` 字典中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 `pathconf_names`，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

从 Python 3.3 起，此功能等价于 `os.pathconf(fd, name)`。

可用性: Unix。

os.**fstat**(*fd*)
获取文件描述符 *fd* 的状态. 返回一个*stat_result* 对象。

从 Python 3.3 起，此功能等价于 `os.stat(fd)`。

也参考:

stat() 函数。

`os.`**`fstatvfs`**(*fd*)
返回文件系统的信息，该文件系统是文件描述符 *fd* 指向的文件所在的文件系统，与*statvfs()* 一样。从 Python 3.3 开始，它等效于 `os.statvfs(fd)`。

可用性: Unix。

`os.`**`fsync`**(*fd*)
强制将文件描述符 *fd* 指向的文件写入磁盘。在 Unix，这将调用原生 `fsync()` 函数；在 Windows，则是 MS `_commit()` 函数。

如果要写入的是缓冲区内的 Python 文件对象 *f*，请先执行 `f.flush()`，然后执行 `os.fsync(f.fileno())`，以确保与 *f* 关联的所有内部缓冲区都写入磁盘。

可用性: Unix, Windows。

`os.`**`ftruncate`**(*fd*, *length*)
截断文件描述符 *fd* 指向的文件，以使其最大为 *length* 字节。从 Python 3.3 开始，它等效于 `os.truncate(fd, length)`。

引发一个审计事件 `os.truncate`，附带参数 `fd`, `length`。

可用性: Unix, Windows。

3.5 版更變: 添加了 Windows 支持

`os.`**`get_blocking`**(*fd*)
获取文件描述符的阻塞模式：如果设置了*O_NONBLOCK* 标志位，返回 `False`，如果该标志位被清除，返回 `True`。

参见*set_blocking()* 和*socket.socket.setblocking()*。

可用性: Unix。

3.5 版新加入.

`os.`**`isatty`**(*fd*)
如果文件描述符 *fd* 打开且已连接至 tty 设备（或类 tty 设备），返回 `True`，否则返回 `False`。

`os.`**`lockf`**(*fd*, *cmd*, *len*)
在打开的文件描述符上，使用、测试或删除 POSIX 锁。*fd* 是一个打开的文件描述符。*cmd* 指定要进行的操作，它们是*F_LOCK*、*F_TLOCK*、*F_ULOCK* 或*F_TEST* 中的一个。*len* 指定哪部分文件需要锁定。

引发一个审计事件 `os.lockf`，附带参数 `fd`、`cmd`、`len`。

可用性: Unix。

3.3 版新加入.

`os.`**`F_LOCK`**
`os.`**`F_TLOCK`**
`os.`**`F_ULOCK`**
`os.`**`F_TEST`**
标志位，用于指定*lockf()* 进行哪一种操作。

可用性: Unix。

3.3 版新加入.

os.**lseek**(*fd*, *pos*, *how*)
將文件描述符 *fd* 的当前位置设置为 *pos*，位置的计算方式 *how* 如下：设置为*SEEK_SET* 或 0 表示从文件开头计算，设置为*SEEK_CUR* 或 1 表示从文件当前位置计算，设置为*SEEK_END* 或 2 表示文件末尾计算。返回新指针位置，这个位置是从文件开头计算的，单位是字节。

os.**SEEK_SET**
os.**SEEK_CUR**
os.**SEEK_END**
lseek() 函数的参数，它们的值分别为 0、1 和 2。

3.3 版新加入: 某些操作系统可能支持其他值，例如 os.SEEK_HOLE 或 os.SEEK_DATA。

os.**open**(*path*, *flags*, *mode=0o777*, *, *dir_fd=None*)
打开文件 *path*，根据 *flags* 设置各种标志位，并根据 *mode* 设置其权限状态。当计算 *mode* 时，会首先根据当前 umask 值将部分权限去除。本方法返回新文件的描述符。新的文件描述符是不可继承 的。

有关 flag 和 mode 取值的说明，请参见 C 运行时文档。标志位常量（如*O_RDONLY* 和*O_WRONLY*）在*os* 模块中定义。特别地，在 Windows 上需要添加*O_BINARY* 才能以二进制模式打开文件。

本函数带有 *dir_fd* 参数，支持基于目录描述符的相对路径。

引发一个审计事件 open，附带参数 path、mode、flags。

3.4 版更變: 新的文件描述符现在是不可继承的。

備𝔽: 本函数适用于底层的 I/O。常规用途请使用内置函数*open()*，该函数的 read() 和 write() 方法（及其他方法）会返回文件对象。要将文件描述符包装在文件对象中，请使用*fdopen()*。

3.3 版新加入: *dir_fd* 参数。

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

3.6 版更變: 接受一个类路径对象。

以下常量是*open()* 函数 *flags* 参数的选项。可以用按位或运算符 | 将它们组合使用。部分常量并非在所有平台上都可用。有关其可用性和用法的说明，请参阅 *open(2)* 手册（Unix 上）或 MSDN （Windows 上）。

os.**O_RDONLY**
os.**O_WRONLY**
os.**O_RDWR**
os.**O_APPEND**
os.**O_CREAT**
os.**O_EXCL**
os.**O_TRUNC**
上述常量在 Unix 和 Windows 上均可用。

os.**O_DSYNC**
os.**O_RSYNC**
os.**O_SYNC**
os.**O_NDELAY**
os.**O_NONBLOCK**
os.**O_NOCTTY**
os.**O_CLOEXEC**
这个常数仅在 Unix 系统中可用。

3.3 版更變: 增加*O_CLOEXEC* 常量。

os.**O_BINARY**
os.**O_NOINHERIT**

os.**O_SHORT_LIVED**
os.**O_TEMPORARY**
os.**O_RANDOM**
os.**O_SEQUENTIAL**
os.**O_TEXT**
这个常数仅在 Windows 系统中可用。

os.**O_ASYNC**
os.**O_DIRECT**
os.**O_DIRECTORY**
os.**O_NOFOLLOW**
os.**O_NOATIME**
os.**O_PATH**
os.**O_TMPFILE**
os.**O_SHLOCK**
os.**O_EXLOCK**
上述常量是扩展常量，如果 C 库未定义它们，则不存在。

3.4 版更變: 在支持的系统上增加*O_PATH*。增加*O_TMPFILE*，仅在 Linux Kernel 3.11 或更高版本可用。

os.**openpty**()
打开一对新的伪终端，返回一对文件描述符（主，从），分别为 pty 和 tty。新的文件描述符是不可继承的。对于（稍微）轻量一些的方法，请使用*pty*模块。

可用性: 某些 Unix。

3.4 版更變: 新的文件描述符不再可继承。

os.**pipe**()
创建一个管道，返回一对分别用于读取和写入的文件描述符（r, w）。新的文件描述符是不可继承的。

可用性: Unix, Windows。

3.4 版更變: 新的文件描述符不再可继承。

os.**pipe2**(*flags*)
创建带有 *flags* 标志位的管道。可通过对以下一个或多个值进行“或”运算来构造这些 *flags*: *O_NONBLOCK*、*O_CLOEXEC*。返回一对分别用于读取和写入的文件描述符（r, w）。

可用性: 某些 Unix。

3.3 版新加入.

os.**posix_fallocate**(*fd*, *offset*, *len*)
确保为 *fd* 指向的文件分配了足够的磁盘空间，该空间从偏移量 *offset* 开始，到 *len* 字节为止。

可用性: Unix。

3.3 版新加入.

os.**posix_fadvise**(*fd*, *offset*, *len*, *advice*)
声明即将以特定模式访问数据，使内核可以提前进行优化。数据范围是从 *fd* 所指向文件的 *offset* 开始，持续 *len* 个字节。*advice* 的取值是如下之一: *POSIX_FADV_NORMAL*, *POSIX_FADV_SEQUENTIAL*, *POSIX_FADV_RANDOM*, *POSIX_FADV_NOREUSE*, *POSIX_FADV_WILLNEED* 或*POSIX_FADV_DONTNEED*。

可用性: Unix。

3.3 版新加入.

os.**POSIX_FADV_NORMAL**
os.**POSIX_FADV_SEQUENTIAL**
os.**POSIX_FADV_RANDOM**

os.**POSIX_FADV_NOREUSE**
os.**POSIX_FADV_WILLNEED**
os.**POSIX_FADV_DONTNEED**
用于*posix_fadvise()* 的 *advice* 参数的标志位，指定可能使用的访问模式。

可用性: Unix。

3.3 版新加入.

os.**pread**(*fd*, *n*, *offset*)
从文件描述符 *fd* 所指向文件的偏移位置 *offset* 开始，读取至多 *n* 个字节，而保持文件偏移量不变。

返回所读取字节的字节串 (bytestring)。如果到达了 *fd* 指向的文件末尾，则返回空字节对象。

可用性: Unix。

3.3 版新加入.

os.**preadv**(*fd*, *buffers*, *offset*, *flags=0*)
从文件描述符 *fd* 所指向文件的偏移位置 *offset* 开始，将数据读取至可变字节类对象 缓冲区 *buffers* 中，保持文件偏移量不变。将数据依次存放到每个缓冲区中，填满一个后继续存放到序列中的下一个缓冲区，来保存其余数据。

flags 参数可以由零个或多个标志位进行按位或运算来得到：

- *RWF_HIPRI*
- *RWF_NOWAIT*

返回实际读取的字节总数，该总数可以小于所有对象的总容量。

操作系统可能对允许使用的缓冲区数量有限制（使用*sysconf()* 获取 `'SC_IOV_MAX'` 值）。

本方法结合了*os.readv()* 和*os.pread()* 的功能。

可用性：Linux 2.6.30 或更高版本，FreeBSD 6.0 或更高版本，OpenBSD 2.7 或更高版本，AIX 7.1 或更高版本。使用标志位需要 Linux 4.6 或更高版本。

3.7 版新加入.

os.**RWF_NOWAIT**
不要等待无法立即获得的数据。如果指定了此标志，那么当需要从后备存储器中读取数据，或等待文件锁时，系统调用将立即返回。

如果成功读取数据，则返回读取的字节数。如果未读取到数据，则返回 `-1`，并将错误码 errno 置为*errno.EAGAIN*。

可用性：Linux 4.14 或更高版本。

3.7 版新加入.

os.**RWF_HIPRI**
高优先级读/写。允许基于块的文件系统对设备进行轮询，这样可以降低延迟，但可能会占用更多资源。

目前在 Linux 上，此功能仅在使用*O_DIRECT* 标志打开的文件描述符上可用。

可用性：Linux 4.6 或更高版本。

3.7 版新加入.

os.**pwrite**(*fd*, *str*, *offset*)
将 *str* 中的字节串 (bytestring) 写入文件描述符 *fd* 的偏移位置 *offset* 处，保持文件偏移量不变。

返回实际写入的字节数。

可用性: Unix。

3.3 版新加入.

os.**pwritev**(*fd*, *buffers*, *offset*, *flags=0*)
将缓冲区 *buffers* 的内容写入文件描述符 *fd* 的偏移位置 *offset* 处，保持文件偏移量不变。缓冲区 *buffers* 必须是由字节类对象 组成的序列。缓冲区以数组顺序处理。先写入第一个缓冲区的全部内容，再写入第二个缓冲区，照此继续。

flags 参数可以由零个或多个标志位进行按位或运算来得到：

- *RWF_DSYNC*
- *RWF_SYNC*

返回实际写入的字节总数。

操作系统可能对允许使用的缓冲区数量有限制（使用*sysconf()* 获取 'SC_IOV_MAX' 值）。

本方法结合了*os.writev()* 和*os.pwrite()* 的功能。

可用性：Linux 2.6.30 或更高版本，FreeBSD 6.0 或更高版本，OpenBSD 2.7 或更高版本，AIX 7.1 或更高版本。使用标志位需要 Linux 4.7 或更高版本。

3.7 版新加入.

os.**RWF_DSYNC**
提供立即写入功能，等效于*O_DSYNC* open(2) 标志。该标志仅作用于系统调用写入的数据。

可用性：Linux 4.7 或更高版本。

3.7 版新加入.

os.**RWF_SYNC**
提供立即写入功能，等效于*O_SYNC* open(2) 标志。该标志仅作用于系统调用写入的数据。

可用性：Linux 4.7 或更高版本。

3.7 版新加入.

os.**read**(*fd*, *n*)
从文件描述符 *fd* 中读取至多 *n* 个字节。

返回所读取字节的字节串 (bytestring)。如果到达了 *fd* 指向的文件末尾，则返回空字节对象。

備🄵: 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。若要读取由内建函数*open()*、*popen()*、*fdopen()* 或*sys.stdin* 返回的" 文件对象"，则应使用其相应的 read() 或 readline() 方法。

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

os.**sendfile**(*out_fd*, *in_fd*, *offset*, *count*)
os.**sendfile**(*out_fd*, *in_fd*, *offset*, *count*, *headers=()*, *trailers=()*, *flags=0*)
将文件描述符 *in_fd* 中的 *count* 字节复制到文件描述符 *out_fd* 的偏移位置 *offset* 处。返回复制的字节数，如果到达 EOF，返回 0。

定义了*sendfile()* 的所有平台均支持第一种函数用法。

在 Linux 上，将 *offset* 设置为 None，则从 *in_fd* 的当前位置开始读取，并更新 *in_fd* 的位置。

第二种情况可以被用于 macOS 和 FreeBSD，其中 *headers* 和 *trailers* 是任意的缓冲区序列，它们会在写入来自 *in_fd* 的数据之前被写入。它的返回内容与第一种情况相同。

在 macOS 和 FreeBSD 上，传入 0 值作为 *count* 将指定持续发送直至到达 *in_fd* 的末尾。

所有平台都支持将套接字作为 *out_fd* 文件描述符，有些平台也支持其他类型（如常规文件或管道）。

跨平台应用程序不应使用 *headers*、*trailers* 和 *flags* 参数。

可用性: Unix。

備🄵: 有关`sendfile()`的高级封装，参见`socket.socket.sendfile()`。

3.3 版新加入.

3.9 版更變: *out* 和 *in* 参数被重命名为 *out_fd* 和 *in_fd*。

os.**set_blocking**(*fd*, *blocking*)
设置指定文件描述符的阻塞模式：如果 blocking 为 `False`，则为该描述符设置`O_NONBLOCK` 标志位，反之则清除该标志位。

参见`get_blocking()` 和`socket.socket.setblocking()`。

可用性: Unix。

3.5 版新加入.

os.**SF_NODISKIO**
os.**SF_MNOWAIT**
os.**SF_SYNC**
`sendfile()` 函数的参数（假设当前实现支持这些参数）。

可用性: Unix。

3.3 版新加入.

os.**readv**(*fd*, *buffers*)
从文件描述符 *fd* 将数据读取至多个可变的字节类对象 缓冲区 *buffers* 中。将数据依次存放到每个缓冲区中，填满一个后继续存放到序列中的下一个缓冲区，来保存其余数据。

返回实际读取的字节总数，该总数可以小于所有对象的总容量。

操作系统可能对允许使用的缓冲区数量有限制（使用`sysconf()` 获取 `'SC_IOV_MAX'` 值）。

可用性: Unix。

3.3 版新加入.

os.**tcgetpgrp**(*fd*)
返回与 *fd* 指定的终端相关联的进程组（*fd* 是由`os.open()` 返回的已打开的文件描述符）。

可用性: Unix。

os.**tcsetpgrp**(*fd*, *pg*)
设置与 *fd* 指定的终端相关联的进程组为 *pg*（**fd* 是由`os.open()` 返回的已打开的文件描述符）。

可用性: Unix。

os.**ttyname**(*fd*)
返回一个字符串，该字符串表示与文件描述符 *fd* 关联的终端。如果 *fd* 没有与终端关联，则抛出异常。

可用性: Unix。

os.**write**(*fd*, *str*)
将 *str* 中的字节串 (bytestring) 写入文件描述符 *fd*。

返回实际写入的字节数。

備𝔽: 该功能适用于低级 I/O 操作，必须用于*os.open()* 或*pipe()* 返回的文件描述符。若要写入由内建函数*open()*、*popen()*、*fdopen()*、*sys.stdout* 或*sys.stderr* 返回的" 文件对象"，则应使用其相应的 `write()` 方法。

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

`os.`**`writev`**(*fd*, *buffers*)
将缓冲区 *buffers* 的内容写入文件描述符 *fd*。缓冲区 *buffers* 必须是由字节类对象 组成的序列。缓冲区以数组顺序处理。先写入第一个缓冲区的全部内容，再写入第二个缓冲区，照此继续。

返回实际写入的字节总数。

操作系统可能对允许使用的缓冲区数量有限制（使用*sysconf()* 获取 `'SC_IOV_MAX'` 值）。

可用性: Unix。

3.3 版新加入.

查询终端的尺寸

3.3 版新加入.

`os.`**`get_terminal_size`**(*fd=STDOUT_FILENO*)
返回终端窗口的尺寸，格式为 `(columns, lines)`，它是类型为*terminal_size* 的元组。

可选参数 `fd` （默认为 `STDOUT_FILENO` 或标准输出）指定应查询的文件描述符。

如果文件描述符未连接到终端，则抛出*OSError* 异常。

shutil.get_terminal_size() 是供常规使用的高阶函数，`os.get_terminal_size` 是其底层的实现。

可用性: Unix, Windows。

`class` `os.`**`terminal_size`**
元组的子类，存储终端窗口尺寸 `(columns, lines)`。

`columns`
终端窗口的宽度，单位为字符。

`lines`
终端窗口的高度，单位为字符。

文件描述符的继承

3.4 版新加入.

每个文件描述符都有一个"inheritable"（可继承）标志位，该标志位控制了文件描述符是否可以由子进程继承。从 Python 3.4 开始，由 Python 创建的文件描述符默认是不可继承的。

在 UNIX 上，执行新程序时，不可继承的文件描述符在子进程中是关闭的，其他文件描述符将被继承。

在 Windows 上，不可继承的句柄和文件描述符在子进程中是关闭的，但标准流（文件描述符 0、1 和 2 即标准输入、标准输出和标准错误）是始终继承的。如果使用*spawn** 函数，所有可继承的句柄和文件描述符都将被继承。如果使用*subprocess* 模块，将关闭除标准流以外的所有文件描述符，并且仅当 *close_fds* 参数为 `False` 时才继承可继承的句柄。

os.**get_inheritable**(*fd*)
获取指定文件描述符的“可继承”标志位（为布尔值）。

os.**set_inheritable**(*fd*, *inheritable*)
设置指定文件描述符的“可继承”标志位。

os.**get_handle_inheritable**(*handle*)
获取指定句柄的“可继承”标志位（为布尔值）。

可用性: Windows。

os.**set_handle_inheritable**(*handle*, *inheritable*)
设置指定句柄的“可继承”标志位。

可用性: Windows。

16.1.5 文件和目录

在某些 Unix 平台上，许多函数支持以下一项或多项功能：

- **指定文件描述符为参数：**通常在*os* 模块中提供给函数的 *path* 参数必须是表示文件路径的字符串，但是，某些函数现在可以接受其 *path* 参数为打开文件描述符，该函数将对描述符指向的文件进行操作。（对于 POSIX 系统，Python 将调用以 `f` 开头的函数变体（如调用 `fchdir` 而不是 `chdir`）。）

  可以用*os.supports_fd* 检查某个函数在你的平台上是否支持将 *path* 参数指定为文件描述符。如果不支持，使用该功能将抛出*NotImplementedError* 异常。

  如果该函数还支持 *dir_fd* 或 *follow_symlinks* 参数，那么用文件描述符作为 *path* 后就不能再指定上述参数了。

- **基于目录描述符的相对路径：**如果 *dir_fd* 不是 `None`，它就应该是一个指向目录的文件描述符，这时待操作的 path 应该是相对路径，相对路径是相对于前述目录的。如果 path 是绝对路径，则 *dir_fd* 将被忽略。（对于 POSIX 系统，Python 将调用该函数的变体，变体以 `at` 结尾，可能以 `f` 开头（如调用 `faccessat` 而不是 `access`）。

  可以用*os.supports_dir_fd* 检查某个函数在你的平台上是否支持 *dir_fd*。如果不支持，使用该功能将抛出*NotImplementedError* 异常。

- **不跟踪符号链接：**如果 *follow_symlinks* 为 `False`，并且待操作路径的最后一个元素是符号链接，则该函数将在符号链接本身而不是链接所指向的文件上操作。（对于 POSIX 系统，Python 将调用该函数的 `l...` 变体。）

  可以用*os.supports_follow_symlinks* 检查某个函数在你的平台上是否支持 *follow_symlinks*。如果不支持，使用该功能将抛出*NotImplementedError* 异常。

os.**access**(*path*, *mode*, *, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)
使用实际用户 ID/用户组 ID 测试对 *path* 的访问。请注意，大多数测试操作将使用有效用户 ID/用户组 ID，因此可以在 suid/sgid 环境中运用此例程，来测试调用用户是否具有对 *path* 的指定访问权限。*mode* 为*F_OK* 时用于测试 *path* 是否存在，也可以对*R_OK*、*W_OK* 和*X_OK* 中的一个或多个进行“或”运算来测试指定权限。允许访问则返回*True*，否则返回*False*。更多信息请参见 Unix 手册页 *access(2)*。

本函数支持指定基于目录描述符的相对路径 和不跟踪符号链接。

如果 *effective_ids* 为 `True`，*access()* 将使用有效用户 ID/用户组 ID 而非实际用户 ID/用户组 ID 进行访问检查。您的平台可能不支持 *effective_ids*，您可以使用*os.supports_effective_ids* 检查它是否可用。如果不可用，使用它时会抛出*NotImplementedError* 异常。

備F: 使用*access()* 来检查用户是否具有某项权限（如打开文件的权限），然后再使用*open()* 打开文件，这样做存在一个安全漏洞，因为用户可能会在检查和打开文件之间的时间里做其他操作。推荐使用*EAFP* 技术。如:

```
if os.access("myfile", os.R_OK):
    with open("myfile") as fp:
        return fp.read()
return "some default data"
```

最好写成:

```
try:
    fp = open("myfile")
except PermissionError:
    return "some default data"
else:
    with fp:
        return fp.read()
```

備🄵: 即使*access()* 指示 I/O 操作会成功，但实际操作仍可能失败，尤其是对网络文件系统的操作，其权限语义可能超出常规的 POSIX 权限位模型。

3.3 版更變: 添加 *dir_fd*、*effective_ids* 和 *follow_symlinks* 参数。

3.6 版更變: 接受一个类路径对象。

os.**F_OK**
os.**R_OK**
os.**W_OK**
os.**X_OK**
作为*access()* 的 *mode* 参数的可选值，分别测试 *path* 的存在性、可读性、可写性和可执行性。

os.**chdir**(*path*)
将当前工作目录更改为 *path*。

本函数支持指定文件描述符为参数。其中，描述符必须指向打开的目录，不能是打开的文件。

本函数可以抛出*OSError* 及其子类的异常，如*FileNotFoundError*、*PermissionError* 和*NotADirectoryError* 异常。

引发一个审计事件 os.chdir，附带参数 path。

3.3 版新加入: 在某些平台上新增支持将 *path* 参数指定为文件描述符。

3.6 版更變: 接受一个类路径对象。

os.**chflags**(*path*, *flags*, *, *follow_symlinks=True*)
将 *path* 的 flags 设置为其他由数字表示的 *flags*。*flags* 可以用以下值按位或组合起来（以下值在*stat* 模块中定义）:

- *stat.UF_NODUMP*
- *stat.UF_IMMUTABLE*
- *stat.UF_APPEND*
- *stat.UF_OPAQUE*
- *stat.UF_NOUNLINK*
- *stat.UF_COMPRESSED*
- *stat.UF_HIDDEN*

- *stat.SF_ARCHIVED*
- *stat.SF_IMMUTABLE*
- *stat.SF_APPEND*
- *stat.SF_NOUNLINK*
- *stat.SF_SNAPSHOT*

本函数支持不跟踪符号链接。

引发一个审计事件 os.chflags，附带参数 path、flags。

可用性: Unix。

3.3 版新加入: *follow_symlinks* 参数。

3.6 版更變: 接受一个类路径对象。

os.**chmod**(*path*, *mode*, *, *dir_fd=None*, *follow_symlinks=True*)

将 *path* 的 mode 更改为其他由数字表示的 *mode*。*mode* 可以用以下值之一，也可以将它们按位或组合起来（以下值在*stat* 模块中定义）：

- *stat.S_ISUID*
- *stat.S_ISGID*
- *stat.S_ENFMT*
- *stat.S_ISVTX*
- *stat.S_IREAD*
- *stat.S_IWRITE*
- *stat.S_IEXEC*
- *stat.S_IRWXU*
- *stat.S_IRUSR*
- *stat.S_IWUSR*
- *stat.S_IXUSR*
- *stat.S_IRWXG*
- *stat.S_IRGRP*
- *stat.S_IWGRP*
- *stat.S_IXGRP*
- *stat.S_IRWXO*
- *stat.S_IROTH*
- *stat.S_IWOTH*
- *stat.S_IXOTH*

本函数支持指定文件描述符、指定基于目录描述符的相对路径 和不跟踪符号链接。

備F: 尽管 Windows 支持*chmod()*，但只能用它设置文件的只读标志（stat.S_IWRITE 和 stat.S_IREAD 常量或对应的整数值）。所有其他标志位都会被忽略。

引发一个审计事件 os.chmod，附带参数 path、mode、dir_fd。

3.3 版新加入: 添加了指定 *path* 为文件描述符的支持，以及 *dir_fd* 和 *follow_symlinks* 参数。

3.6 版更變: 接受一个类路径对象。

os.**chown**(*path*, *uid*, *gid*, *, *dir_fd=None*, *follow_symlinks=True*)
将 *path* 的用户和组 ID 分别修改为数字形式的 *uid* 和 *gid*。若要使其中某个 ID 保持不变，请将其置为 -1。

本函数支持指定文件描述符、指定基于目录描述符的相对路径 和不跟踪符号链接。

参见更高阶的函数*shutil.chown()*，除了数字 ID 之外，它也接受名称。

引发一个审计事件 os.chown，附带参数 path、uid、gid、dir_fd。

可用性: Unix。

3.3 版新加入: 添加了指定 *path* 为文件描述符的支持，以及 *dir_fd* 和 *follow_symlinks* 参数。

3.6 版更變: 支持类路径对象。

os.**chroot**(*path*)
将当前进程的根目录更改为 *path*。

可用性: Unix。

3.6 版更變: 接受一个类路径对象。

os.**fchdir**(*fd*)
将当前工作目录更改为文件描述符 *fd* 指向的目录。fd 必须指向打开的目录而非文件。从 Python 3.3 开始，它等效于 os.chdir(fd)。

引发一个审计事件 os.chdir，附带参数 path。

可用性: Unix。

os.**getcwd**()
返回表示当前工作目录的字符串。

os.**getcwdb**()
返回表示当前工作目录的字节串 (bytestring)。

3.8 版更變: 在 Windows 上，本函数现在会使用 UTF-8 编码格式而不是 ANSI 代码页：请参看 **PEP 529** 了解具体原因。该函数在 Windows 上不再被弃用。

os.**lchflags**(*path*, *flags*)
将 *path* 的 flags 设置为其他由数字表示的 *flags*，与*chflags()* 类似，但不跟踪符号链接。从 Python 3.3 开始，它等效于 os.chflags(path, flags, follow_symlinks=False)。

引发一个审计事件 os.chflags，附带参数 path、flags。

可用性: Unix。

3.6 版更變: 接受一个类路径对象。

os.**lchmod**(*path*, *mode*)
将 *path* 的权限状态修改为 *mode*。如果 path 是符号链接，则影响符号链接本身而非链接目标。可以参考*chmod()* 中列出 *mode* 的可用值。从 Python 3.3 开始，它等效于 os.chmod(path, mode, follow_symlinks=False)。

引发一个审计事件 os.chmod，附带参数 path、mode、dir_fd。

可用性: Unix。

3.6 版更變: 接受一个类路径对象。

os.**lchown**(*path*, *uid*, *gid*)
将 *path* 的用户和组 ID 分别修改为数字形式的 *uid* 和 *gid*，本函数不跟踪符号链接。从 Python 3.3 开始，它等效于 os.chown(path, uid, gid, follow_symlinks=False)。

引发一个审计事件 os.chown，附带参数 path、uid、gid、dir_fd。

可用性: Unix。

3.6 版更變: 接受一个类路径对象。

os.**link**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*, *follow_symlinks=True*)
创建一个指向 *src* 的硬链接，名为 *dst*。

本函数支持将 *src_dir_fd* 和 *dst_dir_fd* 中的一个或两个指定为基于目录描述符的相对路径，支持不跟踪符号链接。

引发一个审计事件 os.link 附带参数 src、dst、src_dir_fd、dst_dir_fd。

可用性: Unix, Windows。

3.2 版更變: 添加了对 Windows 的支持。

3.3 版新加入: 添加 *src_dir_fd*、*dst_dir_fd* 和 *follow_symlinks* 参数。

3.6 版更變: 接受一个类路径对象 作为 *src* 和 *dst*。

os.**listdir**(*path='.'*)
返回一个包含由 *path* 指定目录中条目名称组成的列表。该列表按任意顺序排列，并且不包括特殊条目 '.' 和 '..'，即使它们存在于目录中。如果有文件在调用此函数期间在被移除或添加到目录中，是否要包括该文件的名称并没有规定。

path 可以是类路径对象。如果 *path* 是（直接传入或通过*PathLike* 接口间接传入）bytes 类型，则返回的文件名也将是 bytes 类型，其他情况下是 str 类型。

本函数也支持指定文件描述符为参数，其中描述符必须指向目录。

引发一个审计事件 os.listdir，附带参数 path。

備F: 要将 str 类型的文件名编码为 bytes，请使用*fsencode()*。

也参考:

scandir() 函数返回目录内文件名的同时，也返回文件属性信息，它在某些具体情况下能提供更好的性能。

3.2 版更變: *path* 变为可选参数。

3.3 版新加入: 新增支持将 *path* 参数指定为打开的文件描述符。

3.6 版更變: 接受一个类路径对象。

os.**lstat**(*path*, *, *dir_fd=None*)
在给定路径上执行本函数，其操作相当于 lstat() 系统调用，类似于*stat()* 但不跟踪符号链接。返回值是*stat_result* 对象。

在不支持符号链接的平台上，本函数是*stat()* 的别名。

从 Python 3.3 起，此功能等价于 os.stat(path, dir_fd=dir_fd, follow_symlinks=False)。

本函数支持基于目录描述符的相对路径。

也参考:

stat() 函数。

3.2 版更變: 添加对 Windows 6.0 (Vista) 符号链接的支持。

3.3 版更變: 添加了 *dir_fd* 参数。

3.6 版更變: 接受一个类路径对象。

3.8 版更變: 目前在 Windows 上，遇到表示另一个路径的重解析点（即名称代理，包括符号链接和目录结点），本函数将打开它。其他种类的重解析点由`stat()`交由操作系统解析。

os.**mkdir**(*path*, *mode=0o777*, *, *dir_fd=None*)
创建一个名为 *path* 的目录，应用以数字表示的权限模式 *mode*。

If the directory already exists, `FileExistsError` is raised. If a parent directory in the path does not exist, `FileNotFoundError` is raised.

某些系统会忽略 *mode*。如果没有忽略它，那么将首先从它中减去当前的 umask 值。如果除最后 9 位（即 *mode* 八进制的最后 3 位）之外，还设置了其他位，则其他位的含义取决于各个平台。在某些平台上，它们会被忽略，应显式调用`chmod()`进行设置。

本函数支持基于目录描述符的相对路径。

如果需要创建临时目录，请参阅`tempfile`模块中的`tempfile.mkdtemp()`函数。

引发一个审计事件 os.mkdir，附带参数 path、mode、dir_fd。

3.3 版新加入: *dir_fd* 参数。

3.6 版更變: 接受一个类路径对象。

os.**makedirs**(*name*, *mode=0o777*, *exist_ok=False*)
递归目录创建函数。与`mkdir()`类似，但会自动创建到达最后一级目录所需要的中间目录。

mode 参数会传递给`mkdir()`，用来创建最后一级目录，对于该参数的解释，请参阅*mkdir() 中的描述*。要设置某些新建的父目录的权限，可以在调用`makedirs()`之前设置 umask。现有父目录的权限不会更改。

如果 *exist_ok* 为 False (默认值)，则如果目标目录已存在将引发`FileExistsError`。

備🄵: 如果要创建的路径元素包含`pardir` (如 UNIX 系统中的"..") `makedirs()` 将无法明确目标。

本函数能正确处理 UNC 路径。

引发一个审计事件 os.mkdir，附带参数 path、mode、dir_fd。

3.2 版新加入: *exist_ok* 参数。

3.4.1 版更變: 在 Python 3.4.1 以前，如果 *exist_ok* 为 True，且目录已存在，且 *mode* 与现有目录的权限不匹配，`makedirs()` 仍会抛出错误。由于无法安全地实现此行为，因此在 Python 3.4.1 中将该行为删除。请参阅 bpo-21082。

3.6 版更變: 接受一个类路径对象。

3.7 版更變: *mode* 参数不再影响新创建的中间目录的权限。

os.**mkfifo**(*path*, *mode=0o666*, *, *dir_fd=None*)
创建一个名为 *path* 的 FIFO（命名管道，一种先进先出队列），具有以数字表示的权限状态 *mode*。将从 mode 中首先减去当前的 umask 值。

本函数支持基于目录描述符的相对路径。

FIFO 是可以像常规文件一样访问的管道。FIFO 如果没有被删除（如使用`os.unlink()`），会一直存在。通常，FIFO 用作“客户端”和“服务器”进程之间的汇合点：服务器打开 FIFO 进行读取，而客户端打开 FIFO 进行写入。请注意，`mkfifo()` 不会打开 FIFO --- 它只是创建汇合点。

可用性: Unix。

3.3 版新加入: *dir_fd* 参数。

3.6 版更變: 接受一个类路径对象。

os.**mknod**(*path*, *mode=0o600*, *device=0*, *, *dir_fd=None*)
创建一个名为 *path* 的文件系统节点（文件，设备专用文件或命名管道）。*mode* 指定权限和节点类型，方法是将权限与下列节点类型 stat.S_IFREG、stat.S_IFCHR、stat.S_IFBLK 和 stat.S_IFIFO 之一（按位或）组合（这些常量可以在*stat* 模块中找到）。对于 stat.S_IFCHR 和 stat.S_IFBLK，*device* 参数指定了新创建的设备专用文件（可能会用到*os.makedev()*），否则该参数将被忽略。

本函数支持基于目录描述符的相对路径。

可用性: Unix。

3.3 版新加入: *dir_fd* 参数。

3.6 版更變: 接受一个类路径对象。

os.**major**(*device*)
提取主设备号，提取自原始设备号（通常是 stat 中的 st_dev 或 st_rdev 字段）。

os.**minor**(*device*)
提取次设备号，提取自原始设备号（通常是 stat 中的 st_dev 或 st_rdev 字段）。

os.**makedev**(*major*, *minor*)
将主设备号和次设备号组合成原始设备号。

os.**pathconf**(*path*, *name*)
返回所给名称的文件有关的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX.1，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 pathconf_names 字典中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 pathconf_names，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

本函数支持指定文件描述符为参数。

可用性: Unix。

3.6 版更變: 接受一个类路径对象。

os.**pathconf_names**
字典，表示映射关系，为*pathconf()* 和*fpathconf()* 可接受名称与操作系统为这些名称定义的整数值之间的映射。这可用于判断系统已定义了哪些名称。

可用性: Unix。

os.**readlink**(*path*, *, *dir_fd=None*)
返回一个字符串，为符号链接指向的实际路径。其结果可以是绝对或相对路径。如果是相对路径，则可用 os.path.join(os.path.dirname(path), result) 转换为绝对路径。

如果 *path* 是字符串对象（直接传入或通过*PathLike* 接口间接传入），则结果也将是字符串对象，且此类调用可能会引发 UnicodeDecodeError。如果 *path* 是字节对象（直接传入或间接传入），则结果将会是字节对象。

本函数支持基于目录描述符的相对路径。

当尝试解析的路径可能含有链接时，请改用*realpath()* 以正确处理递归和平台差异。

可用性: Unix, Windows。

3.2 版更變: 添加对 Windows 6.0 (Vista) 符号链接的支持。

3.3 版新加入: *dir_fd* 参数。

3.6 版更變: 在 Unix 上可以接受一个类路径对象。

3.8 版更變: 在 Windows 上接受类路径对象 和字节对象。

3.8 版更變: 增加了对目录链接的支持，且返回值改为了“替换路径”的形式（通常带有 \\?\ 前缀），而不是先前那样返回可选的”print name” 字段。

os.**remove**(*path*, *, *dir_fd=None*)

移除（删除）文件 *path*。如果 *path* 是目录，则会引发*IsADirectoryError*。请使用*rmdir()* 来删除目录。如果文件不存在，则会引发*FileNotFoundError*。

本函数支持基于目录描述符的相对路径。

在 Windows 上，尝试删除正在使用的文件会抛出异常。而在 Unix 上，虽然该文件的条目会被删除，但分配给文件的存储空间仍然不可用，直到原始文件不再使用为止。

本函数在语义上与*unlink()* 相同。

引发一个审计事件 os.remove，附带参数 path、dir_fd。

3.3 版新加入: *dir_fd* 参数。

3.6 版更變: 接受一个类路径对象。

os.**removedirs**(*name*)

递归删除目录。工作方式类似于*rmdir()*，不同之处在于，如果成功删除了末尾一级目录，*removedirs()* 会尝试依次删除 *path* 中提到的每个父目录，直到抛出错误为止（但该错误会被忽略，因为这通常表示父目录不是空目录）。例如，os.removedirs('foo/bar/baz') 将首先删除目录 'foo/bar/baz'，然后如果 'foo/bar' 和 'foo' 为空，则继续删除它们。如果无法成功删除末尾一级目录，则抛出*OSError* 异常。

引发一个审计事件 os.remove，附带参数 path、dir_fd。

3.6 版更變: 接受一个类路径对象。

os.**rename**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

将文件或目录 *src* 重命名为 *dst*。如果 *dst* 已存在，则下列情况下将会操作失败，并抛出*OSError* 的子类:

在 Windows 上，如果 *dst* 已存在，则抛出*FileExistsError* 异常。

在 Unix 上，如果 *src* 是文件而 *dst* 是目录，将抛出*IsADirectoryError* 异常，反之则抛出*NotADirectoryError* 异常。如果两者都是目录且 *dst* 为空，则 *dst* 将被静默替换。如果 *dst* 是非空目录，则抛出*OSError* 异常。如果两者都是文件，则在用户具有权限的情况下，将对 *dst* 进行静默替换。如果 *src* 和 *dst* 在不同的文件系统上，则本操作在某些 Unix 分支上可能会失败。如果成功，重命名操作将是一个原子操作（这是 POSIX 的要求）。

本函数支持将 *src_dir_fd* 和 *dst_dir_fd* 中的一个或两个指定为基于目录描述符的相对路径。

如果需要在不同平台上都能替换目标，请使用*replace()*。

引发一个审计事件 os.rename 附带参数 src、dst、src_dir_fd、dst_dir_fd。

3.3 版新加入: *src_dir_fd* 和 *dst_dir_fd* 参数。

3.6 版更變: 接受一个类路径对象 作为 *src* 和 *dst*。

os.**renames**(*old*, *new*)

递归重命名目录或文件。工作方式类似*rename()*，除了会首先创建新路径所需的中间目录。重命名后，将调用*removedirs()* 删除旧路径中不需要的目录。

備🄵: 如果用户没有权限删除末级的目录或文件，则本函数可能会无法建立新的目录结构。

引发一个审计事件 os.rename 附带参数 src、dst、src_dir_fd、dst_dir_fd。

3.6 版更變: 接受一个类路径对象 作为 *old* 和 *new*。

os.**replace**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)
Rename the file or directory *src* to *dst*. If *dst* is a non-empty directory, *OSError* will be raised. If *dst* exists and is a file, it will be replaced silently if the user has permission. The operation may fail if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

本函数支持将 *src_dir_fd* 和 *dst_dir_fd* 中的一个或两个指定为基于目录描述符的相对路径。

引发一个审计事件 os.rename 附带参数 src、dst、src_dir_fd、dst_dir_fd。

3.3 版新加入.

3.6 版更變: 接受一个类路径对象 作为 *src* 和 *dst*。

os.**rmdir**(*path*, *, *dir_fd=None*)
移除（删除）目录 *path*。如果目录不存在或不为空，则会分别抛出*FileNotFoundError* 或*OSError* 异常。要删除整个目录树，可以使用*shutil.rmtree()*。

本函数支持基于目录描述符的相对路径。

引发一个审计事件 os.rmdir，附带参数 path、dir_fd。

3.3 版新加入: *dir_fd* 参数。

3.6 版更變: 接受一个类路径对象。

os.**scandir**(*path='.'*)
返回一个*os.DirEntry* 对象的迭代器，它们对应于由 *path* 指定目录中的条目。这些条目会以任意顺序生成，并且不包括特殊条目 '.' 和 '..'。如果有文件在迭代器创建之后在目录中被移除或添加，是否要包括该文件对应的条目并没有规定。

如果需要文件类型或文件属性信息，使用*scandir()* 代替*listdir()* 可以大大提高这部分代码的性能，因为如果操作系统在扫描目录时返回的是*os.DirEntry* 对象，则该对象包含了这些信息。所有*os.DirEntry* 的方法都可能执行一次系统调用，但是*is_dir()* 和*is_file()* 通常只在有符号链接时才执行一次系统调用。*os.DirEntry.stat()* 在 Unix 上始终需要一次系统调用，而在 Windows 上只在有符号链接时才需要。

path 可以是类路径对象。如果 *path* 是（直接传入或通过*PathLike* 接口间接传入的）bytes 类型，那么每个*os.DirEntry* 的*name* 和*path* 属性将是 bytes 类型，其他情况下是 str 类型。

本函数也支持指定文件描述符为参数，其中描述符必须指向目录。

引发一个审计事件 os.scandir，附带参数 path。

scandir() 迭代器支持上下文管理 协议，并具有以下方法:

scandir.**close**()
关闭迭代器并释放占用的资源。

当迭代器迭代完毕，或垃圾回收，或迭代过程出错时，将自动调用本方法。但仍建议显式调用它或使用 with 语句。

3.6 版新加入.

下面的例子演示了*scandir()* 的简单用法，用来显示给定 *path* 中所有不以 '.' 开头的文件（不包括目录）。entry.is_file() 通常不会增加一次额外的系统调用:

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

備𝔽: 在基于 Unix 的系统上，*scandir()* 使用系统的 opendir() 和 readdir() 函数。在 Windows 上，它使用 Win32 FindFirstFileW 和 FindNextFileW 函数。

3.5 版新加入.

3.6 版新加入: 添加了对上下文管理 协议和*close()* 方法的支持。如果*scandir()* 迭代器没有迭代完毕且没有显式关闭，其析构函数将发出*ResourceWarning* 警告。

本函数接受一个类路径对象。

3.7 版更變: 在 Unix 上新增支持指定文件描述符为参数。

class os.**DirEntry**

由*scandir()* 生成的对象，用于显示目录内某个条目的文件路径和其他文件属性。

scandir() 将在不进行额外系统调用的情况下，提供尽可能多的此类信息。每次进行 `stat()` 或 `lstat()` 系统调用时，`os.DirEntry` 对象会将结果缓存下来。

`os.DirEntry` 实例不适合存储在长期存在的数据结构中，如果你知道文件元数据已更改，或者自调用*scandir()* 以来已经经过了很长时间，请调用 `os.stat(entry.path)` 来获取最新信息。

因为 `os.DirEntry` 方法可以进行系统调用，所以它也可能抛出*OSError* 异常。如需精确定位错误，可以逐个调用 `os.DirEntry` 中的方法来捕获*OSError*，并适当处理。

为了能直接用作类路径对象，`os.DirEntry` 实现了*PathLike* 接口。

`os.DirEntry` 实例所包含的属性和方法如下:

name

本条目的基本文件名，是根据*scandir()* 的 *path* 参数得出的相对路径。

如果*scandir()* 的 *path* 参数是 `bytes` 类型，则*name* 属性也是 `bytes` 类型，否则为 `str`。使用*fsdecode()* 解码 byte 类型的文件名。

path

本条目的完整路径:等效于`os.path.join(scandir_path, entry.name)`,其中 *scandir_path* 就是*scandir()* 的 *path* 参数。仅当*scandir()* 的 *path* 参数为绝对路径时，本路径才是绝对路径。如果*scandir()* 的 *path* 参数是文件描述符，则*path* 属性与上述*name* 属性相同。

如果*scandir()* 的 *path* 参数是 `bytes` 类型，则*path* 属性也是 `bytes` 类型，否则为 `str`。使用*fsdecode()* 解码 byte 类型的文件名。

inode()

返回本条目的索引节点号 (inode number)。

这一结果是缓存在 `os.DirEntry` 对象中的，请调用 `os.stat(entry.path, follow_symlinks=False).st_ino` 来获取最新信息。

一开始没有缓存时，在 Windows 上需要一次系统调用，但在 Unix 上不需要。

is_dir(*, *follow_symlinks=True*)

如果本条目是目录，或是指向目录的符号链接，则返回 `True`。如果本条目是文件，或指向任何其他类型的文件，或该目录不再存在，则返回 `False`。

如果 *follow_symlinks* 是 `False`，那么仅当本条目为目录时返回 `True`（不跟踪符号链接），如果本条目是任何类型的文件，或该文件不再存在，则返回 `False`。

这一结果是缓存在 `os.DirEntry` 对象中的，且 *follow_symlinks* 为 `True` 和 `False` 时的缓存是分开的。请调用*os.stat()* 和*stat.S_ISDIR()* 来获取最新信息。

一开始没有缓存时，大多数情况下不需要系统调用。特别是对于非符号链接，Windows 和 Unix 都不需要系统调用，除非某些 Unix 文件系统（如网络文件系统）返回了 `dirent.d_type ==`

DT_UNKNOWN。如果本条目是符号链接，则需要一次系统调用来跟踪它（除非 *follow_symlinks* 为 False）。

本方法可能抛出*OSError* 异常，如*PermissionError* 异常，但*FileNotFoundError* 异常会被内部捕获且不会抛出。

is_file(*, *follow_symlinks=True*)
如果本条目是文件，或是指向文件的符号链接，则返回 True。如果本条目是目录，或指向目录，或指向其他非文件条目，或该文件不再存在，则返回 False。

如果 *follow_symlinks* 是 False，那么仅当本条目为文件时返回 True（不跟踪符号链接），如果本条目是目录或其他非文件条目，或该文件不再存在，则返回 False。

这一结果是缓存在 os.DirEntry 对象中的。缓存、系统调用、异常抛出都与*is_dir()* 一致。

is_symlink()
如果本条目是符号链接（即使是断开的链接），返回 True。如果是目录或任何类型的文件，或本条目不再存在，返回 False。

这一结果是缓存在 os.DirEntry 对象中的，请调用*os.path.islink()* 来获取最新信息。

一开始没有缓存时，大多数情况下不需要系统调用。其实 Windows 和 Unix 都不需要系统调用，除非某些 Unix 文件系统（如网络文件系统）返回了 dirent.d_type == DT_UNKNOWN。

本方法可能抛出*OSError* 异常，如*PermissionError* 异常，但*FileNotFoundError* 异常会被内部捕获且不会抛出。

stat(*, *follow_symlinks=True*)
返回本条目对应的*stat_result* 对象。本方法默认会跟踪符号链接，要获取符号链接本身的 stat，请添加 follow_symlinks=False 参数。

在 Unix 上，本方法需要一次系统调用。在 Windows 上，仅在 *follow_symlinks* 为 True 且该条目是一个重解析点（如符号链接或目录结点）时，才需要一次系统调用。

在 Windows 上，*stat_result* 的 st_ino、st_dev 和 st_nlink 属性总是为零。请调用*os.stat()* 以获得这些属性。

这一结果是缓存在 os.DirEntry 对象中的，且 *follow_symlinks* 为 True 和 False 时的缓存是分开的。请调用*os.stat()* 来获取最新信息。

注意，os.DirEntry 和*pathlib.Path* 的几个属性和方法之间存在很好的对应关系。具体来说是 name 属性，以及 is_dir()、is_file()、is_symlink() 和 stat() 方法，在两个类中具有相同的含义。

3.5 版新加入.

3.6 版更變: 添加了对*PathLike* 接口的支持。在 Windows 上添加了对*bytes* 类型路径的支持。

os.**stat**(*path*, *, *dir_fd=None*, *follow_symlinks=True*)
获取文件或文件描述符的状态。在所给路径上执行等效于 stat() 系统调用的操作。*path* 可以是字符串类型，或（直接传入或通过*PathLike* 接口间接传入的）bytes 类型，或打开的文件描述符。返回一个*stat_result* 对象。

本函数默认会跟踪符号链接，要获取符号链接本身的 stat，请添加 follow_symlinks=False 参数，或使用*lstat()*。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

在 Windows 上，传入 follow_symlinks=False 将禁用所有名称代理重解析点，其中包括符号链接和目录结点。其他类型的重解析点将直接打开，比如不像链接的或系统无法跟踪的重解析点。当多个链接形成一个链时，本方法可能会返回原始链接的 stat，无法完整遍历到非链接的对象。在这种情况下，要获取最终路径的 stat，请使用*os.path.realpath()* 函数尽可能地解析路径，并在解析结果上调用*lstat()*。这不适用于空链接或交接点，否则会抛出异常。

示例:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

也参考:

fstat() 和*lstat()* 函数。

3.3 版新加入: 增加 *dir_fd* 和 *follow_symlinks* 参数，可指定文件描述符代替路径。

3.6 版更變: 接受一个类路径对象。

3.8 版更變: 在 Windows 上，本方法将跟踪系统能解析的所有重解析点，并且传入 `follow_symlinks=False` 会停止跟踪所有名称代理重解析点。现在，如果操作系统遇到无法跟踪的重解析点，*stat* 将返回原始路径的信息，就像已指定 `follow_symlinks=False` 一样，而不会抛出异常。

class os.**stat_result**

本对象的属性大致对应于 `stat` 结构体成员，主要作为*os.stat()*、*os.fstat()* 和*os.lstat()* 的返回值。

属性:

st_mode
文件模式：包括文件类型和文件模式位（即权限位）。

st_ino
与平台有关，但如果不为零，则根据 `st_dev` 值唯一地标识文件。通常:

- 在 Unix 上该值表示索引节点号 (inode number)。
- 在 Windows 上该值表示 文件索引号 。

st_dev
该文件所在设备的标识符。

st_nlink
硬链接的数量。

st_uid
文件所有者的用户 ID。

st_gid
文件所有者的用户组 ID。

st_size
文件大小（以字节为单位），文件可以是常规文件或符号链接。符号链接的大小是它包含的路径的长度，不包括末尾的空字节。

时间戳:

st_atime
最近的访问时间，以秒为单位。

st_mtime
最近的修改时间，以秒为单位。

st_ctime
取决于平台：

- 在 Unix 上表示最近的元数据更改时间，
- 在 Windows 上表示创建时间，以秒为单位。

st_atime_ns
最近的访问时间，以纳秒表示，为整数。

st_mtime_ns
最近的修改时间，以纳秒表示，为整数。

st_ctime_ns
取决于平台：

- 在 Unix 上表示最近的元数据更改时间，
- 在 Windows 上表示创建时间，以纳秒表示，为整数。

備F：*st_atime*、*st_mtime* 和*st_ctime* 属性的确切含义和分辨率取决于操作系统和文件系统。例如，在使用 FAT 或 FAT32 文件系统的 Windows 上，*st_mtime* 有 2 秒的分辨率，而*st_atime* 仅有 1 天的分辨率。详细信息请参阅操作系统文档。

类似地，尽管*st_atime_ns*、*st_mtime_ns* 和*st_ctime_ns* 始终以纳秒表示，但许多系统并不提供纳秒精度。在确实提供纳秒精度的系统上，用于存储*st_atime*、*st_mtime* 和*st_ctime* 的浮点对象无法保留所有精度，因此不够精确。如果需要确切的时间戳，则应始终使用*st_atime_ns*、*st_mtime_ns* 和*st_ctime_ns*。

在某些 Unix 系统上（如 Linux 上），以下属性可能也可用：

st_blocks
为文件分配的字节块数，每块 512 字节。文件是稀疏文件时，它可能小于*st_size*/512。

st_blksize
“首选的”块大小，用于提高文件系统 I/O 效率。写入文件时块大小太小可能会导致读取-修改-重写效率低下。

st_rdev
设备类型（如果是 inode 设备）。

st_flags
用户定义的文件标志位。

在其他 Unix 系统上（如 FreeBSD 上），以下属性可能可用（但仅当 root 使用它们时才被填充）：

st_gen
文件生成号。

st_birthtime
文件创建时间。

在 Solaris 及其衍生版本上，以下属性可能也可用：

st_fstype
文件所在文件系统的类型的唯一标识，为字符串。

On macOS systems, the following attributes may also be available:

st_rsize
文件的实际大小。

st_creator
文件的创建者。

st_type
文件类型。

在 Windows 系统上，以下属性也可用：

st_file_attributes
Windows 文件属性：dwFileAttributes，由 GetFileInformationByHandle() 返回的 BY_HANDLE_FILE_INFORMATION 结构体的成员之一。请参阅*stat* 模块中的 FILE_ATTRIBUTE_* 常量。

st_reparse_tag
当*st_file_attributes* 存在 FILE_ATTRIBUTE_REPARSE_POINT 集合时，本字段包含重解析点类型标记。请参阅*stat* 模块中的 IO_REPARSE_TAG_* 常量。

标准模块*stat* 中定义了函数和常量，这些函数和常量可用于从 stat 结构体中提取信息。（在 Windows 上，某些项填充的是虚值。）

为了向后兼容，一个*stat_result* 实例还可以作为至少包含 10 个整数的元组访问，以提供 stat 结构中最重要（和可移植）的成员，整数顺序为*st_mode*, *st_ino*, *st_dev*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, *st_ctime*。某些实现可能在末尾还有更多项。为了与旧版 Python 兼容，以元组形式访问*stat_result* 始终返回整数。

3.3 版新加入: 添加了*st_atime_ns*、*st_mtime_ns* 和*st_ctime_ns* 成员。

3.5 版新加入: 在 Windows 上添加了*st_file_attributes* 成员。

3.5 版更變: 在 Windows 上，如果可用，会返回文件索引作为*st_ino* 的值。

3.7 版新加入: 在 Solaris 及其衍生版本上添加了*st_fstype* 成员。

3.8 版新加入: 在 Windows 上添加了*st_reparse_tag* 成员。

3.8 版更變: 在 Windows 上，*st_mode* 成员现在可以根据需要将特殊文件标识为 S_IFCHR、S_IFIFO 或 S_IFBLK。

os.**statvfs**(*path*)
在所给的路径上执行 statvfs() 系统调用。返回值是一个对象，其属性描述了所给路径上的文件系统，并且与 statvfs 结构体的成员相对应，即：f_bsize, f_frsize, f_blocks, f_bfree, f_bavail, f_files, f_ffree, f_favail, f_flag, f_namemax, f_fsid。

为 f_flag 属性位定义了两个模块级常量：如果存在 ST_RDONLY 位，则文件系统以只读挂载；如果存在 ST_NOSUID 位，则文件系统禁用或不支持 setuid/setgid 位。

为基于 GNU/glibc 的系统还定义了额外的模块级常量。它们是 ST_NODEV（禁止访问设备专用文件），ST_NOEXEC（禁止执行程序），ST_SYNCHRONOUS（写入后立即同步），ST_MANDLOCK（允许文件系统上的强制锁定），ST_WRITE（写入文件/目录/符号链接），ST_APPEND（仅追加文件），ST_IMMUTABLE（不可变文件），ST_NOATIME（不更新访问时间），ST_NODIRATIME（不更新目录访问时间），ST_RELATIME（相对于 mtime/ctime 更新访问时间）。

本函数支持指定文件描述符为参数。

可用性: Unix。

3.2 版更變: 添加了 ST_RDONLY 和 ST_NOSUID 常量。

3.3 版新加入: 新增支持将 *path* 参数指定为打开的文件描述符。

3.4 版更變: 添加了 ST_NODEV、ST_NOEXEC、ST_SYNCHRONOUS、ST_MANDLOCK、ST_WRITE、ST_APPEND、ST_IMMUTABLE、ST_NOATIME、ST_NODIRATIME 和 ST_RELATIME 常量。

3.6 版更變: 接受一个类路径对象。

3.7 版新加入: 添加了 f_fsid。

os.**supports_dir_fd**
一个*set* 对象，指示*os* 模块中的哪些函数接受一个打开的文件描述符作为 *dir_fd* 参数。不同平台提供的功能不同，且 Python 用于实现 *dir_fd* 参数的底层函数并非在 Python 支持的所有平台上都可用。考虑到一致性，支持 *dir_fd* 的函数始终允许指定描述符，但如果在底层不支持时调用了该函数，则会抛出异常。(在所有平台上始终支持将 *dir_fd* 指定为 None。)

要检查某个函数是否接受打开的文件描述符作为 *dir_fd* 参数，请在 supports_dir_fd 前使用 in 运算符。例如，如果*os.stat()* 在当前平台上接受打开的文件描述符作为 *dir_fd* 参数，则此表达式的计算结果为 True:

```
os.stat in os.supports_dir_fd
```

目前 *dir_fd* 参数仅在 Unix 平台上有效，在 Windows 上均无效。

3.3 版新加入.

os.**supports_effective_ids**
一个*set* 对象，指示*os.access()* 是否允许在当前平台上将其 *effective_ids* 参数指定为 True。(所有平台都支持将 *effective_ids* 指定为 False。) 如果当前平台支持，则集合将包含*os.access()*，否则集合为空。

如果当前平台上的*os.access()* 支持 effective_ids=True，则此表达式的计算结果为 True:

```
os.access in os.supports_effective_ids
```

目前仅 Unix 平台支持 *effective_ids*，Windows 不支持。

3.3 版新加入.

os.**supports_fd**
一个*set* 对象，指示在当前平台上*os* 模块中的哪些函数接受一个打开的文件描述符作为 *path* 参数。不同平台提供的功能不同，且 Python 所使用到的底层函数 (用于实现接受描述符作为 *path*) 并非在 Python 支持的所有平台上都可用。

要判断某个函数是否接受打开的文件描述符作为 *path* 参数，请在 supports_fd 前使用 in 运算符。例如，如果*os.chdir()* 在当前平台上接受打开的文件描述符作为 *path* 参数，则此表达式的计算结果为 True:

```
os.chdir in os.supports_fd
```

3.3 版新加入.

os.**supports_follow_symlinks**
一个*set* 对象，指示在当前平台上*os* 模块中的哪些函数的 *follow_symlinks* 参数可指定为 False。不同平台提供的功能不同，且 Python 用于实现 *follow_symlinks* 的底层函数并非在 Python 支持的所有平台上都可用。考虑到一致性，支持 *follow_symlinks* 的函数始终允许将其指定为 False，但如果在底层不支持时调用了该函数，则会抛出异常。(在所有平台上始终支持将 *follow_symlinks* 指定为 True。)

要检查某个函数的 *follow_symlinks* 参数是否可以指定为 False，请在 supports_follow_symlinks 前使用 in 运算符。例如，如果在当前平台上调用*os.stat()* 时可以指定 follow_symlinks=False，则此表达式的计算结果为 True:

```
os.stat in os.supports_follow_symlinks
```

3.3 版新加入.

os.**symlink**(*src*, *dst*, *target_is_directory=False*, ***, *dir_fd=None*)
创建一个指向 *src* 的符号链接，名为 *dst*。

在 Windows 上，符号链接可以表示文件或目录两种类型，并且不会动态改变类型。如果目标存在，则新建链接的类型将与目标一致。否则，如果 *target_is_directory* 为 `True`，则符号链接将创建为目录链接，为 `False`（默认）将创建为文件链接。在非 Windows 平台上，*target_is_directory* 被忽略。

本函数支持基于目录描述符的相对路径。

備🄵: 在 Windows 10 或更高版本上，如果启用了开发人员模式，非特权帐户可以创建符号链接。如果开发人员模式不可用/未启用，则需要 *SeCreateSymbolicLinkPrivilege* 权限，或者该进程必须以管理员身份运行。

当本函数由非特权账户调用时，抛出 *OSError* 异常。

引发一个审计事件 `os.symlink`，附带参数 `src`、`dst`、`dir_fd`。

可用性: Unix, Windows。

3.2 版更變: 添加对 Windows 6.0 (Vista) 符号链接的支持。

3.3 版新加入: 添加了 *dir_fd* 参数，现在在非 Windows 平台上允许 *target_is_directory* 参数。

3.6 版更變: 接受一个类路径对象 作为 *src* 和 *dst*。

3.8 版更變: 针对启用了开发人员模式的 Windows，添加了非特权账户创建符号链接的支持。

`os.`**`sync`**`()`
强制将所有内容写入磁盘。

可用性: Unix。

3.3 版新加入.

`os.`**`truncate`**`(`*path*, *length*`)`
截断 *path* 对应的文件，以使其最大为 *length* 字节。

本函数支持指定文件描述符为参数。

引发一个审计事件 `os.truncate`，附带参数 `path`, `length`。

可用性: Unix, Windows。

3.3 版新加入.

3.5 版更變: 添加了 Windows 支持

3.6 版更變: 接受一个类路径对象。

`os.`**`unlink`**`(`*path*, *, *dir_fd=None*`)`
移除（删除）文件 *path*。该函数在语义上与 *remove()* 相同，`unlink` 是其传统的 Unix 名称。请参阅 *remove()* 的文档以获取更多信息。

引发一个审计事件 `os.remove`，附带参数 `path`、`dir_fd`。

3.3 版新加入: *dir_fd* 参数。

3.6 版更變: 接受一个类路径对象。

`os.`**`utime`**`(`*path*, *times=None*, **[*, *ns*`]`, *dir_fd=None*, *follow_symlinks=True*`)`
设置文件 *path* 的访问时间和修改时间。

utime() 有 *times* 和 *ns* 两个可选参数，它们指定了设置给 *path* 的时间，用法如下：

- 如果指定 *ns*，它必须是一个 `(atime_ns, mtime_ns)` 形式的二元组，其中每个成员都是一个表示纳秒的整数。

- 如果 *times* 不为 None，则它必须是 (atime, mtime) 形式的二元组，其中每个成员都是一个表示秒的 int 或 float。
- 如果 *times* 为 None 且未指定 *ns*，则相当于指定 ns=(atime_ns, mtime_ns)，其中两个时间均为当前时间。

同时为 *times* 和 *ns* 指定元组会出错。

注意，根据操作系统记录访问时间和修改时间的分辨率，后续的*stat()* 调用可能不会返回此处设置的确切时间。请参阅*stat()*。保留精确时间的最佳方法是使用*os.stat()* 结果对象中的 *st_atime_ns* 和 *st_mtime_ns* 字段，并将 *ns* 参数设置为 *utime*。

本函数支持指定文件描述符、指定基于目录描述符的相对路径 和不跟踪符号链接。

引发一个审计事件 os.utime，附带参数 path、times、ns、dir_fd。

3.3 版新加入: 新增支持将 *path* 参数指定为打开的文件描述符，以及支持 *dir_fd*、*follow_symlinks* 和 *ns* 参数。

3.6 版更變: 接受一个类路径对象。

os.**walk**(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)

生成目录树中的文件名，方式是按上-> 下或下-> 上顺序浏览目录树。对于以 *top* 为根的目录树中的每个目录（包括 *top* 本身），它都会生成一个三元组 (dirpath, dirnames, filenames)。

dirpath 是表示目录路径的字符串。*dirnames* 是 *dirpath* 中子目录名称组成的列表 (excluding '.' and '..')。*filenames* 是 *dirpath* 中非目录文件名称组成的列表。请注意列表中的名称不带路径部分。要获取 *dirpath* 中文件或目录的完整路径（以 *top* 打头），请执行 os.path.join(dirpath, name)。列表是否排序取决于具体文件系统。如果有文件或列表生成期间被移除或添加到 *dirpath* 目录中，是否要包括该文件的名称并没有规定。

如果可选参数 *topdown* 为 True 或未指定，则在所有子目录的三元组之前生成父目录的三元组（目录是自上而下生成的）。如果 *topdown* 为 False，则在所有子目录的三元组生成之后再生成父目录的三元组（目录是自下而上生成的）。无论 *topdown* 为何值，在生成目录及其子目录的元组之前，都将检索全部子目录列表。

当 *topdown* 为 True 时，调用者可以就地修改 *dirnames* 列表（也许用到了 del 或切片），而*walk()* 将仅仅递归到仍保留在 *dirnames* 中的子目录内。这可用于减少搜索、加入特定的访问顺序，甚至可在继续*walk()* 之前告知*walk()* 由调用者新建或重命名的目录的信息。当 *topdown* 为 False 时，修改 *dirnames* 对 walk 的行为没有影响，因为在自下而上模式中，*dirnames* 中的目录是在 *dirpath* 本身之前生成的。

默认将忽略*scandir()* 调用中的错误。如果指定了可选参数 *onerror*，它应该是一个函数。出错时它会被调用，参数是一个*OSError* 实例。它可以报告错误然后继续遍历，或者抛出异常然后中止遍历。注意，可以从异常对象的 filename 属性中获取出错的文件名。

walk() 默认不会递归进指向目录的符号链接。可以在支持符号链接的系统上将 *followlinks* 设置为 True，以访问符号链接指向的目录。

備𝔽: 注意，如果链接指向自身的父目录，则将 *followlinks* 设置为 True 可能导致无限递归。*walk()* 不会记录它已经访问过的目录。

備𝔽: 如果传入的是相对路径，请不要在恢复*walk()* 之间更改当前工作目录。*walk()* 不会更改当前目录，并假定其调用者也不会更改当前目录。

下面的示例遍历起始目录内所有子目录，打印每个目录内的文件占用的字节数，CVS 子目录不会被遍历:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

在下一个示例（*shutil.rmtree()* 的简单实现）中，必须使树自下而上遍历，因为*rmdir()* 只允许在目录为空时删除目录:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

引发一个审计事件 `os.walk`，附带参数 `top`, `topdown`, `onerror`, `followlinks`。

3.5 版更變: 现在，本函数调用的是*os.scandir()* 而不是*os.listdir()*，从而减少了调用*os.stat()* 的次数而变得更快。

3.6 版更變: 接受一个类路径对象。

`os.`**`fwalk`**(*top='.'*, *topdown=True*, *onerror=None*, ***, *follow_symlinks=False*, *dir_fd=None*)

本方法的行为与*walk()* 完全一样，除了它产生的是 4 元组 `(dirpath, dirnames, filenames, dirfd)`，并且它支持 `dir_fd`。

dirpath、*dirnames* 和 *filenames* 与*walk()* 输出的相同，*dirfd* 是指向目录 *dirpath* 的文件描述符。

本函数始终支持基于目录描述符的相对路径 和不跟踪符号链接。但是请注意，与其他函数不同，*fwalk()* 的 *follow_symlinks* 的默认值为 `False`。

備𝔽: 由于*fwalk()* 会生成文件描述符，而它们仅在下一个迭代步骤前有效，因此如果要将描述符保留更久，则应复制它们（比如使用*dup()*）。

下面的示例遍历起始目录内所有子目录，打印每个目录内的文件占用的字节数，CVS 子目录不会被遍历:

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
    print(root, "consumes", end="")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end="")
    print("bytes in", len(files), "non-directory files")
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories
```

在下一个示例中，必须使树自下而上遍历，因为*rmdir()* 只允许在目录为空时删除目录:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

引发一个审计事件 `os.fwalk`，附带参数 `top`, `topdown`, `onerror`, `follow_symlinks`, `dir_fd`。

可用性: Unix。

3.3 版新加入.

3.6 版更變: 接受一个类路径对象。

3.7 版更變: 添加了对 *bytes* 类型路径的支持。

os.**memfd_create**(*name*[, *flags=os.MFD_CLOEXEC*])

创建一个匿名文件，返回指向该文件的文件描述符。*flags* 必须是系统上可用的 `os.MFD_*` 常量之一（或将它们按位“或”组合起来）。新文件描述符默认是不可继承的。

name 提供的名称会被用作文件名，并且 `/proc/self/fd/` 目录中相应符号链接的目标将显示为该名称。显示的名称始终以 `memfd:` 为前缀，并且仅用于调试目的。名称不会影响文件描述符的行为，因此多个文件可以有相同的名称，不会有副作用。

可用性：Linux 3.17 或更高版本，且装有 glibc 2.27 或更高版本。

3.8 版新加入.

os.**MFD_CLOEXEC**
os.**MFD_ALLOW_SEALING**
os.**MFD_HUGETLB**
os.**MFD_HUGE_SHIFT**
os.**MFD_HUGE_MASK**
os.**MFD_HUGE_64KB**
os.**MFD_HUGE_512KB**
os.**MFD_HUGE_1MB**
os.**MFD_HUGE_2MB**
os.**MFD_HUGE_8MB**
os.**MFD_HUGE_16MB**
os.**MFD_HUGE_32MB**
os.**MFD_HUGE_256MB**
os.**MFD_HUGE_512MB**
os.**MFD_HUGE_1GB**
os.**MFD_HUGE_2GB**
os.**MFD_HUGE_16GB**

以上标志位可以传递给 *memfd_create()*。

可用性：Linux 3.17 或更高版本，且装有 glibc 2.27 或更高版本。`MFD_HUGE*` 标志仅在 Linux 4.14 及以上可用。

3.8 版新加入.

Linux 扩展属性

3.3 版新加入.

这些函数仅在 Linux 上可用。

os.**getxattr**(*path*, *attribute*, *, *follow_symlinks=True*)
返回 *path* 的扩展文件系统属性 *attribute* 的值。*attribute* 可以是 bytes 或 str （直接传入或通过*PathLike* 接口间接传入）。如果是 str，则使用文件系统编码来编码字符串。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

引发一个审计事件 os.getxattr，附带参数 path、attribute。

3.6 版更變: 接受一个类路径对象 作为 *path* 和 *attribute*。

os.**listxattr**(*path=None*, *, *follow_symlinks=True*)
返回一个列表，包含 *path* 的所有扩展文件系统属性。列表中的属性都表示为字符串，它们是根据文件系统编码解码出来的。如果 *path* 为 None，则*listxattr()* 将检查当前目录。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

引发一个审计事件 os.listxattr，附带参数 path。

3.6 版更變: 接受一个类路径对象。

os.**removexattr**(*path*, *attribute*, *, *follow_symlinks=True*)
从 *path* 中删除扩展文件系统属性 *attribute*。*attribute* 应该是 bytes 或 str （直接传入或通过*PathLike* 接口间接传入）。如果是 str，则使用文件系统编码来编码字符串。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

引发一个审计事件 os.removexattr，附带参数 path、attribute。

3.6 版更變: 接受一个类路径对象 作为 *path* 和 *attribute*。

os.**setxattr**(*path*, *attribute*, *value*, *flags=0*, *, *follow_symlinks=True*)
将 *path* 的扩展文件系统属性 *attribute* 设为 *value*。*attribute* 必须是没有内嵌 NUL 的字节串或字符串（直接或通过*PathLike* 间接传入）。如果是字符串，则使用文件系统编码格式进行编码。*flags* 可以为*XATTR_REPLACE* 或*XATTR_CREATE*。如果指定*XATTR_REPLACE* 而该属性不存在，则将引发 ENODATA。如果指定*XATTR_CREATE* 而该属性已存在，则将不创建该属性并将引发 EEXISTS。

本函数支持指定文件描述符为参数 和不跟踪符号链接。

備𝔽: Linux kernel 2.6.39 以下版本的一个 bug 导致在某些文件系统上，flags 参数会被忽略。

引发一个审计事件 os.setxattr，附带参数 path、attribute、value、flags。

3.6 版更變: 接受一个类路径对象 作为 *path* 和 *attribute*。

os.**XATTR_SIZE_MAX**
一条扩展属性的值的最大大小。在当前的 Linux 上是 64 KiB。

os.**XATTR_CREATE**
这是*setxattr()* 的 flags 参数的可取值，它表示该操作必须创建一个属性。

os.**XATTR_REPLACE**
这是*setxattr()* 的 flags 参数的可取值，它表示该操作必须替换现有属性。

16.1.6 进程管理

下列函数可用于创建和管理进程。

所有*exec**函数都接受一个参数列表，用来给新程序加载到它的进程中。在所有情况下，传递给新程序的第一个参数是程序本身的名称，而不是用户在命令行上输入的参数。对于 C 程序员来说，这就是传递给 `main()` 函数的 `argv[0]`。例如，`os.execv('/bin/echo', ['foo', 'bar'])` 只会在标准输出上打印 `bar`，而 `foo` 会被忽略。

`os.`**`abort`**`()`
发送 `SIGABRT` 信号到当前进程。在 Unix 上，默认行为是生成一个核心转储。在 Windows 上，该进程立即返回退出代码 `3`。请注意，使用*signal.signal()*可以为 `SIGABRT` 注册 Python 信号处理程序，而调用本函数将不会调用按前述方法注册的程序。

`os.`**`add_dll_directory`**`(`*path*`)`
将路径添加到 DLL 搜索路径。

This search path is used when resolving dependencies for imported extension modules (the module itself is resolved through *sys.path*), and also by *ctypes*.

要移除目录，可以在返回的对象上调用 **close()**，也可以在 `with` 语句内使用本方法。

参阅 Microsoft 文档 获取如何加载 DLL 的信息。

引发一个审计事件 `os.add_dll_directory`，附带参数 `path`。

可用性: Windows。

3.8 版新加入: 早期版本的 CPython 解析 DLL 时用的是当前进程的默认行为。这会导致不一致，比如不是每次都会去搜索 `PATH` 和当前工作目录，且系统函数（如 `AddDllDirectory`）失效。

在 3.8 中，DLL 的两种主要加载方式现在可以显式覆盖进程的行为，以确保一致性。请参阅 移植说明了解如何更新你的库。

`os.`**`execl`**`(`*path*, *arg0*, *arg1*, *...*`)`
`os.`**`execle`**`(`*path*, *arg0*, *arg1*, *...*, *env*`)`
`os.`**`execlp`**`(`*file*, *arg0*, *arg1*, *...*`)`
`os.`**`execlpe`**`(`*file*, *arg0*, *arg1*, *...*, *env*`)`
`os.`**`execv`**`(`*path*, *args*`)`
`os.`**`execve`**`(`*path*, *args*, *env*`)`
`os.`**`execvp`**`(`*file*, *args*`)`
`os.`**`execvpe`**`(`*file*, *args*, *env*`)`
这些函数都将执行一个新程序，以替换当前进程。它们没有返回值。在 Unix 上，新程序会加载到当前进程中，且进程号与调用者相同。过程中的错误会被报告为*OSError*异常。

当前进程会被立即替换。打开的文件对象和描述符都不会刷新，因此如果这些文件上可能缓冲了数据，则应在调用*exec**函数之前使用 `sys.stdout.flush()` 或*os.fsync()*刷新它们。

*exec**函数的”l” 和”v” 变体不同在于命令行参数的传递方式。如果在编码时固定了参数数量，则”l” 变体可能是最方便的，各参数作为 `execl*()` 函数的附加参数传入即可。当参数数量可变时，”v” 变体更方便，参数以列表或元组的形式作为 *args* 参数传递。在这两种情况下，子进程的第一个参数都应该是即将运行的命令名称，但这不是强制性的。

结尾包含”p” 的变体（*execlp()*、*execlpe()*、*execvp()* 和*execvpe()*）将使用 `PATH` 环境变量来查找程序 *file*。当环境被替换时（使用下一段讨论的*exec*e* 变体之一），`PATH` 变量将来自于新环境。其他变体*execl()*、*execle()*、*execv()* 和*execve()* 不使用 `PATH` 变量来查找程序，因此 *path* 必须包含正确的绝对或相对路径。

对于*execle()*、*execlpe()*、*execve()* 和*execvpe()*（都以”e” 结尾），*env* 参数是一个映射，用于定义新进程的环境变量（代替当前进程的环境变量）。而函数*execl()*、*execlp()*、*execv()* 和*execvp()* 会将当前进程的环境变量过继给新进程。

某些平台上的*execve()* 可以将 *path* 指定为打开的文件描述符。当前平台可能不支持此功能，可以使用*os.supports_fd* 检查它是否支持。如果不可用，则使用它会抛出*NotImplementedError* 异常。

引发一个审计事件 `os.exec`，附带参数 `path`、`args`、`env`。

可用性: Unix, Windows。

3.3 版新加入: 新增支持将*execve()* 的 *path* 参数指定为打开的文件描述符。

3.6 版更變: 接受一个类路径对象。

`os._exit`(*n*)
以状态码 *n* 退出进程，不会调用清理处理程序，不会刷新 stdio，等等。

備𝔽: 退出的标准方法是使用 `sys.exit(n)`。而*_exit()* 通常只应在*fork()* 出的子进程中使用。

以下是已定义的退出代码，可以用于*_exit()*，尽管它们不是必需的。这些退出代码通常用于 Python 编写的系统程序，例如邮件服务器的外部命令传递程序。

備𝔼: 其中部分退出代码在部分 Unix 平台上可能不可用，因为平台间存在差异。如果底层平台定义了这些常量，那上层也会定义。

`os.EX_OK`
退出代码，表示未发生任何错误。

可用性: Unix。

`os.EX_USAGE`
退出代码，表示命令使用不正确，如给出的参数数量有误。

可用性: Unix。

`os.EX_DATAERR`
退出代码，表示输入数据不正确。

可用性: Unix。

`os.EX_NOINPUT`
退出代码，表示某个输入文件不存在或不可读。

可用性: Unix。

`os.EX_NOUSER`
退出代码，表示指定的用户不存在。

可用性: Unix。

`os.EX_NOHOST`
退出代码，表示指定的主机不存在。

可用性: Unix。

`os.EX_UNAVAILABLE`
退出代码，表示所需的服务不可用。

可用性: Unix。

`os.EX_SOFTWARE`
退出代码，表示检测到内部软件错误。

可用性: Unix。

os.**EX_OSERR**
退出代码，表示检测到操作系统错误，例如无法 fork 或创建管道。

可用性: Unix。

os.**EX_OSFILE**
退出代码，表示某些系统文件不存在、无法打开或发生其他错误。

可用性: Unix。

os.**EX_CANTCREAT**
退出代码，表示无法创建用户指定的输出文件。

可用性: Unix。

os.**EX_IOERR**
退出代码，表示对某些文件进行读写时发生错误。

可用性: Unix。

os.**EX_TEMPFAIL**
退出代码，表示发生了暂时性故障。它可能并非意味着真正的错误，例如在可重试的情况下无法建立网络连接。

可用性: Unix。

os.**EX_PROTOCOL**
退出代码，表示协议交换是非法的、无效的或无法解读的。

可用性: Unix。

os.**EX_NOPERM**
退出代码，表示没有足够的权限执行该操作（但不适用于文件系统问题）。

可用性: Unix。

os.**EX_CONFIG**
退出代码，表示发生某种配置错误。

可用性: Unix。

os.**EX_NOTFOUND**
退出代码，表示的内容类似于“找不到条目”。

可用性: Unix。

os.**fork**()
Fork 出一个子进程。在子进程中返回 `0`，在父进程中返回子进程的进程号。如果发生错误，则抛出*OSError* 异常。

注意，当从线程中使用 `fork()` 时，某些平台（包括 FreeBSD <= 6.3 和 Cygwin）存在已知问题。

引发一个审计事件 `os.fork`，没有附带参数。

3.8 版更變: 不再支持在子解释器中调用 `fork()`（将抛出*RuntimeError* 异常）。

> **警告:** 有关 SSL 模块与 fork() 结合的应用，请参阅*ssl*。

可用性: Unix。

os.**forkpty**()
Fork 出一个子进程，使用新的伪终端作为子进程的控制终端。返回一对 `(pid, fd)`，其中 *pid* 在子进

程中为 0，这是父进程中新子进程的进程号，而 *fd* 是伪终端主设备的文件描述符。对于更便于移植的方法，请使用*pty* 模块。如果发生错误，则抛出*OSError* 异常。

引发一个审计事件 os.forkpty，没有附带参数。

3.8 版更變: 不再支持在子解释器中调用 forkpty()（将抛出*RuntimeError* 异常）。

可用性: 某些 Unix。

os.**kill**(*pid*, *sig*)
将信号 *sig* 发送至进程 *pid*。特定平台上可用的信号常量定义在*signal* 模块中。

Windows：*signal.CTRL_C_EVENT* 和*signal.CTRL_BREAK_EVENT* 信号是特殊信号，只能发送给共享同一个控制台窗口的控制台进程，如某些子进程。*sig* 取任何其他值将导致该进程被 TerminateProcess API 无条件终止，且退出代码为 *sig*。Windows 版本的*kill()* 还需要传入待结束进程的句柄。

另请参阅*signal.pthread_kill()*。

引发一个审计事件 os.kill，附带参数 pid、sig。

3.2 版新加入: Windows 支持。

os.**killpg**(*pgid*, *sig*)
将信号 *sig* 发送给进程组 *pgid*。

引发一个审计事件 os.killpg，附带参数 pgid、sig。

可用性: Unix。

os.**nice**(*increment*)
将进程的优先级（nice 值）增加 *increment*，返回新的 nice 值。

可用性: Unix。

os.**pidfd_open**(*pid*, *flags=0*)
返回一个文件描述符，它指向进程 *pid*。该描述符可用于管理进程，避免出现竞争和信号。*flags* 参数提供给将来扩展使用，当前没有定义标志值。

更多详细信息请参阅 *pidfd_open(2)* 手册页。

可用性: Linux 5.3+。

3.9 版新加入.

os.**plock**(*op*)
将程序段锁定到内存中。*op* 的值（定义在 <sys/lock.h> 中）决定了哪些段被锁定。

可用性: Unix。

os.**popen**(*cmd*, *mode='r'*, *buffering=-1*)
打开一个管道，它通往 / 接受自命令 *cmd*。返回值是连接到管道的文件对象，根据 *mode* 是 'r'（默认）还是 'w' 决定该对象可以读取还是写入。*buffering* 参数与内置函数*open()* 相应的参数含义相同。返回的文件对象只能读写文本字符串，不能是字节类型。

如果子进程成功退出，则 close 方法返回*None*。如果发生错误，则返回子进程的返回码。在 POSIX 系统上，如果返回码为正，则它就是进程返回值左移一个字节后的值。如果返回码为负，则进程是被信号终止的，返回码取反后就是该信号。(例如，如果子进程被终止，则返回值可能是 - signal.SIGKILL。) 在 Windows 系统上，返回值包含子进程的返回码（有符号整数）。

在 Unix 上，*waitstatus_to_exitcode()* 可以将 close 方法的返回值（即退出状态，不能是 None）转换为退出码。在 Windows 上，close 方法的结果直接就是退出码（或 None）。

本方法是使用*subprocess.Popen* 实现的，如需更强大的方法来管理和沟通子进程，请参阅该类的文档。

os.**posix_spawn**(*path*, *argv*, *env*, *, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

包装 `posix_spawn()` C 库 API，使其可以从 Python 调用。

大多数用户应使用*subprocess.run()* 代替*posix_spawn()*。

仅位置参数 (Positional-only arguments) *path*、*args* 和 *env* 与*execve()* 中的类似。

path 形参是可执行文件的路径，*path* 中应当包含目录。使用*posix_spawnp()* 可传入不带目录的可执行文件。

file_actions 参数可以是由元组组成的序列，序列描述了对子进程中指定文件描述符采取的操作，这些操作会在 C 库实现的 `fork()` 和 `exec()` 步骤间完成。每个元组的第一个元素必须是下面列出的三个类型指示符之一，用于描述元组剩余的元素：

os.**POSIX_SPAWN_OPEN**
(`os.POSIX_SPAWN_OPEN`, *fd*, *path*, *flags*, *mode*)

执行 `os.dup2(os.open(path, flags, mode), fd)`。

os.**POSIX_SPAWN_CLOSE**
(`os.POSIX_SPAWN_CLOSE`, *fd*)

执行 `os.close(fd)`。

os.**POSIX_SPAWN_DUP2**
(`os.POSIX_SPAWN_DUP2`, *fd*, *new_fd*)

执行 `os.dup2(fd, new_fd)`。

这些元组对应于 C 库 `posix_spawn_file_actions_addopen()`，`posix_spawn_file_actions_addclose()` 和 `posix_spawn_file_actions_adddup2()` API 调用，它们为调用 `posix_spawn()` 自身做准备。

setpgroup 参数将子进程的进程组设置为指定值。如果指定值为 0，则子进程的进程组 ID 将与其进程 ID 相同。如果未设置 *setpgroup* 值，则子进程将继承父进程的进程组 ID。本参数对应于 C 库 `POSIX_SPAWN_SETPGROUP` 标志。

如果 *resetids* 参数为 `True`，则会将子进程的有效用户 ID 和有效组 ID 重置为父进程的实际用户 ID 和实际组 ID。如果该参数为 `False`，则子进程保留父进程的有效用户 ID 和有效组 ID。无论哪种情况，若在可执行文件上启用了“设置用户 ID”和“设置组 ID”权限位，它们将覆盖有效用户 ID 和有效组 ID 的设置。本参数对应于 C 库 `POSIX_SPAWN_RESETIDS` 标志。

如果 *setsid* 参数为 `True`，它将为 *posix_spawn* 新建一个会话 ID。*setsid* 需要 `POSIX_SPAWN_SETSID` 或 `POSIX_SPAWN_SETSID_NP` 标志，否则会抛出*NotImplementedError* 异常。

setsigmask 参数将信号掩码设置为指定的信号集合。如果未使用该参数，则子进程将继承父进程的信号掩码。本参数对应于 C 库 `POSIX_SPAWN_SETSIGMASK` 标志。

sigdef 参数将集合中所有信号的操作全部重置为默认。本参数对应于 C 库 `POSIX_SPAWN_SETSIGDEF` 标志。

scheduler 参数必须是一个元组，其中包含调度器策略（可选）以及携带了调度器参数的*sched_param* 实例。在调度器策略所在位置为 `None` 表示未提供该值。本参数是 C 库 `POSIX_SPAWN_SETSCHEDPARAM` 和 `POSIX_SPAWN_SETSCHEDULER` 标志的组合。

引发一个审计事件 `os.posix_spawn`，附带参数 `path`、`argv`、`env`。

3.8 版新加入.

可用性: Unix。

os.**posix_spawnp**(*path*, *argv*, *env*, *, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

包装 `posix_spawnp()` C 库 API，使其可以从 Python 调用。

与*posix_spawn()* 相似，但是系统会在 PATH 环境变量指定的目录列表中搜索可执行文件 *executable*（与 execvp(3) 相同）。

引发一个审计事件 os.posix_spawn，附带参数 path、argv、env。

3.8 版新加入.

可用性: 请参阅*posix_spawn()* 文档。

os.**register_at_fork**(*, *before=None*, *after_in_parent=None*, *after_in_child=None*)

注册可调用对象，在使用*os.fork()* 或类似的进程克隆 API 派生新的子进程时，这些对象会运行。参数是可选的，且为仅关键字 (Keyword-only) 参数。每个参数指定一个不同的调用点。

- *before* 是一个函数，在 fork 子进程前调用。
- *after_in_parent* 是一个函数，在 fork 子进程后从父进程调用。
- *after_in_child* 是一个函数，从子进程中调用。

只有希望控制权回到 Python 解释器时，才进行这些调用。典型的子进程 启动时不会触发它们，因为子进程不会重新进入解释器。

在注册的函数中，用于 fork 前运行的函数将按与注册相反的顺序调用。用于 fork 后（从父进程或子进程）运行的函数按注册顺序调用。

注意，第三方 C 代码的 fork() 调用可能不会调用这些函数，除非它显式调用了 PyOS_BeforeFork()、PyOS_AfterFork_Parent() 和 PyOS_AfterFork_Child()。

函数注册后无法注销。

可用性: Unix。

3.7 版新加入.

os.**spawnl**(*mode*, *path*, *...*)
os.**spawnle**(*mode*, *path*, *...*, *env*)
os.**spawnlp**(*mode*, *file*, *...*)
os.**spawnlpe**(*mode*, *file*, *...*, *env*)
os.**spawnv**(*mode*, *path*, *args*)
os.**spawnve**(*mode*, *path*, *args*, *env*)
os.**spawnvp**(*mode*, *file*, *args*)
os.**spawnvpe**(*mode*, *file*, *args*, *env*)

在新进程中执行程序 *path*。

（注意，*subprocess* 模块提供了更强大的工具来生成新进程并跟踪执行结果，使用该模块比使用这些函数更好。尤其应当检查使用 *subprocess* 模块替换旧函数 部分。）

mode 为*P_NOWAIT* 时，本函数返回新进程的进程号。*mode* 为*P_WAIT* 时，如果进程正常退出，返回退出代码，如果被终止，返回 -signal，其中 *signal* 是终止进程的信号。在 Windows 上，进程号实际上是进程句柄，因此可以与*waitpid()* 函数一起使用。

注意在 VxWorks 上，新进程被终止时，本函数不会返回 -signal，而是会抛出 OSError 异常。

*spawn** 函数的"l" 和"v" 变体不同在于命令行参数的传递方式。如果在编码时固定了参数数量，则"l" 变体可能是最方便的，各参数作为 spawnl*() 函数的附加参数传入即可。当参数数量可变时，"v" 变体更方便，参数以列表或元组的形式作为 *args* 参数传递。在这两种情况下，子进程的第一个参数都必须是即将运行的命令名称。

结尾包含第二个"p" 的变体（*spawnlp()*、*spawnlpe()*、*spawnvp()* 和*spawnvpe()*）将使用 PATH 环境变量来查找程序 *file*。当环境被替换时（使用下一段讨论的*spawn*e* 变体之一），PATH 变量将来自于新环境。其他变体*spawnl()*、*spawnle()*、*spawnv()* 和*spawnve()* 不使用 PATH 变量来查找程序，因此 *path* 必须包含正确的绝对或相对路径。

对于*spawnle()*、*spawnlpe()*、*spawnve()* 和*spawnvpe()* （都以"e" 结尾），*env* 参数是一个映射，用于定义新进程的环境变量（代替当前进程的环境变量）。而函数*spawnl()*、*spawnlp()*、*spawnv()* 和*spawnvp()* 会将当前进程的环境变量过继给新进程。注意，*env* 字典中的键和值必须是字符串。无效的键或值将导致函数出错，返回值为 127。

例如，以下对*spawnlp()* 和*spawnvpe()* 的调用是等效的:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

引发一个审计事件 os.spawn，附带参数 mode、path、args、env。

可用性: Unix, Windows。*spawnlp()*、*spawnlpe()*、*spawnvp()* 和*spawnvpe()* 在 Windows 上不可用。*spawnle()* 和*spawnve()* 在 Windows 上不是线程安全的，建议使用*subprocess* 模块替代。

3.6 版更變: 接受一个类路径对象。

os.P_NOWAIT

os.P_NOWAITO

*spawn** 系列函数的 *mode* 参数的可取值。如果给出这些值中的任何一个，则 spawn*() 函数将在创建新进程后立即返回，且返回值为进程号。

可用性: Unix, Windows。

os.P_WAIT

*spawn** 系列函数的 *mode* 参数的可取值。如果将 *mode* 指定为该值，则 spawn*() 函数将在新进程运行完毕后返回，运行成功则返回进程的退出代码，被信号终止则返回 -signal。

可用性: Unix, Windows。

os.P_DETACH

os.P_OVERLAY

*spawn** 系列函数的 *mode* 参数的可取值。它们比上面列出的值可移植性差。*P_DETACH* 与*P_NOWAIT* 相似，但是新进程会与父进程的控制台脱离。使用*P_OVERLAY* 则会替换当前进程，*spawn** 函数将不会返回。

可用性: Windows。

os.startfile(*path*[, *operation*])

使用已关联的应用程序打开文件。

当 *operation* 未指定或指定为 'open' 时，这类似于在 Windows 资源管理器中双击文件，或在交互式命令行中将文件名作为 **start** 命令的参数：通过扩展名相关联的应用程序（如果有）打开文件。

当指定另一个 *operation* 时，它必须是一个 "命令动词" ("command verb")，该词指定对文件执行的操作。Microsoft 文档中的常用动词有 'print' 和 'edit' （用于文件），以及 'explore' 和 'find' （用于目录）。

关联的应用程序启动后*startfile()* 就会立即返回。本函数没有等待应用程序关闭的选项，也没有办法检索应用程序的退出状态。*path* 形参是基于当前目录的相对路径。如果要使用绝对路径，请确保第一个字符不是斜杠 ('/')；如果是斜杠的话则底层的 Win32 ShellExecute() 函数将失效。请使用*os.path.normpath()* 函数来确保路径已针对 Win32 正确编码。

为了减少解释器的启动开销，直到第一次调用本函数后，才解析 Win32 ShellExecute() 函数。如果无法解析该函数 ，则抛出*NotImplementedError* 异常。

引发一个审计事件 os.startfile，附带参数 path、operation。

可用性: Windows。

os.**system**(*command*)
在子外壳程序中执行此命令（一个字符串）。这是通过调用标准 C 函数 system() 来实现的，并受到同样的限制。对*sys.stdin* 的更改等不会反映在所执行命令的环境中。如果 *command* 生成了任何输出，它将被发送到解释器的标准输出流。C 标准没有指明这个 C 函数返回值的含义，因此这个 Python 函数的返回值取决于具体系统。

在 Unix 上，返回值为进程的退出状态，以针对*wait()* 而指定的格式进行编码。

在 Windows 上，返回值是运行 *command* 后系统 Shell 返回的值。该 Shell 由 Windows 环境变量 COMSPEC: 给出：通常是 **cmd.exe**，它会返回命令的退出状态。在使用非原生 Shell 的系统上，请查阅 Shell 的文档。

subprocess 模块提供了更强大的工具来生成新进程并跟踪执行结果，使用该模块比使用本函数更好。参阅*subprocess* 文档中的*使用 subprocess 模块替换旧函数* 部分以获取有用的帮助。

在 Unix 上，*waitstatus_to_exitcode()* 可以将返回值（即退出状态）转换为退出码。在 Windows 上，返回值就是退出码。

引发一个*审计事件* os.system，附带参数 command。

可用性: Unix, Windows。

os.**times**()
返回当前的全局进程时间。返回值是一个有 5 个属性的对象：

- user - user time
- system - system time
- children_user - user time of all child processes
- children_system - system time of all child processes
- elapsed - elapsed real time since a fixed point in the past

For backwards compatibility, this object also behaves like a five-tuple containing user, system, children_user, children_system, and elapsed in that order.

See the Unix manual page *times(2)* and *times(3)* manual page on Unix or the GetProcessTimes MSDN on Windows. On Windows, only user and system are known; the other attributes are zero.

可用性: Unix, Windows。

3.3 版更變: 返回结果的类型由元组变成一个类似元组的对象，同时具有命名的属性。

os.**wait**()
等待子进程执行完毕，返回一个元组，包含其 pid 和退出状态指示：一个 16 位数字，其低字节是终止该进程的信号编号，高字节是退出状态码（信号编号为零的情况下），如果生成了核心文件，则低字节的高位会置位。

可以使用*waitstatus_to_exitcode()* 来将退出状态转换为退出码。

可用性: Unix。

也参考:

waitpid() 可以等待特定的子进程执行完毕，且支持更多选项。

os.**waitid**(*idtype*, *id*, *options*)
等待一个或多个子进程执行完毕。*idtype* 可以是*P_PID*, *P_PGID*, *P_ALL*, 或*P_PIDFD* （Linux 可用）。*id* 指定要等待的 pid。*options* 是由*WEXITED*、*WSTOPPED* 或*WCONTINUED* 中的一个或多个进行或运算构造的，且额外可以与*WNOHANG* 或*WNOWAIT* 进行或运算。返回值是一个对象，对应着 siginfo_t 结构体中的数据，即：si_pid, si_uid, si_signo, si_status, si_code 或 None（如果指定了*WNOHANG* 且没有子进程处于等待状态）。

可用性: Unix。

3.3 版新加入.

os.**P_PID**
os.**P_PGID**
os.**P_ALL**
waitid() 的 *idtype* 参数的可取值。它们影响 *id* 的解释方式。

可用性: Unix。

3.3 版新加入.

os.**P_PIDFD**
这是仅 Linux 上存在的一种 *idtype*，它表示 *id* 是指向一个进程的文件描述符。

可用性: Linux 5.4+

3.9 版新加入.

os.**WEXITED**
os.**WSTOPPED**
os.**WNOWAIT**
用于*waitid()* 的 *options* 参数的标志位，指定要等待的子进程信号。

可用性: Unix。

3.3 版新加入.

os.**CLD_EXITED**
os.**CLD_KILLED**
os.**CLD_DUMPED**
os.**CLD_TRAPPED**
os.**CLD_STOPPED**
os.**CLD_CONTINUED**
waitid() 返回的结果中，si_code 的可取值。

可用性: Unix。

3.3 版新加入.

3.9 版更變: 添加了*CLD_KILLED* 和*CLD_STOPPED* 值。

os.**waitpid**(*pid*, *options*)
本函数的细节在 Unix 和 Windows 上有不同之处。

在 Unix 上：等待进程号为 *pid* 的子进程执行完毕，返回一个元组，内含其进程 ID 和退出状态指示（编码与*wait()* 相同）。调用的语义受整数 *options* 的影响，常规操作下该值应为 0。

如果 *pid* 大于 0，则*waitpid()* 会获取该指定进程的状态信息。如果 *pid* 为 0，则获取当前进程所在进程组中的所有子进程的状态。如果 *pid* 为 -1，则获取当前进程的子进程状态。如果 *pid* 小于 -1，则获取进程组 -pid （*pid* 的绝对值）中所有进程的状态。

当系统调用返回 -1 时，将抛出带有错误码的*OSError* 异常。

在 Windows 上：等待句柄为 *pid* 的进程执行完毕，返回一个元组，内含 *pid* 以及左移 8 位后的退出状态码（移位简化了跨平台使用本函数）。小于或等于 0 的 *pid* 在 Windows 上没有特殊含义，且会抛出异常。整数值 *options* 无效。*pid* 可以指向任何 ID 已知的进程，不一定是子进程。调用*spawn** 函数时传入*P_NOWAIT* 将返回合适的进程句柄。

可以使用*waitstatus_to_exitcode()* 来将退出状态转换为退出码。

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此函数现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

os.**wait3**(*options*)
与*waitpid()* 相似，差别在于没有进程 ID 参数，且返回一个 3 元组，其中包括子进程 ID，退出状态指示和资源使用信息。关于资源使用信息的详情，请参考*resource*.*getrusage()*。option 参数与传入*waitpid()* 和*wait4()* 的相同。

可以使用*waitstatus_to_exitcode()* 来将退出状态转换为退出码。

可用性: Unix。

os.**wait4**(*pid*, *options*)
与*waitpid()* 相似，差别在本方法返回一个 3 元组，其中包括子进程 ID，退出状态指示和资源使用信息。关于资源使用信息的详情，请参考*resource*.*getrusage()*。*wait4()* 的参数与*waitpid()* 的参数相同。

可以使用*waitstatus_to_exitcode()* 来将退出状态转换为退出码。

可用性: Unix。

os.**waitstatus_to_exitcode**(*status*)
将等待状态转换为退出码。

在 Unix 上：

- 如果进程正常退出（当 `WIFEXITED(status)` 为真值），则返回进程退出状态 (返回 `WEXITSTATUS(status)`): 结果值大于等于 0。
- 如果进程被信号终止（当 `WIFSIGNALED(status)` 为真值），则返回 `-signum` 其中 *signum* 为导致进程终止的信号数值 (返回 `-WTERMSIG(status)`): 结果值小于 0。
- 否则将抛出*ValueError* 异常。

在 Windows 上，返回 *status* 右移 8 位的结果。

在 Unix 上，如果进程正被追踪或*waitpid()* 附带*WUNTRACED* 选项被调用，则调用者必须先检查 `WIFSTOPPED(status)` 是否为真值。如果 `WIFSTOPPED(status)` 为真值则此函数不可被调用。

也参考:

WIFEXITED(), *WEXITSTATUS()*, *WIFSIGNALED()*, *WTERMSIG()*, *WIFSTOPPED()*, *WSTOPSIG()* 函数。

3.9 版新加入.

os.**WNOHANG**
用于*waitpid()* 的选项，如果没有立即可用的子进程状态，则立即返回。在这种情况下，函数返回 `(0, 0)`。

可用性: Unix。

os.**WCONTINUED**
被任务控制 (job control) 停止的子进程，如果上次报告状态后已恢复运行，则此选项将报告这些子进程。

可用性: 部分 Unix 系统。

os.**WUNTRACED**
已停止的子进程，如果自停止以来尚未报告其当前状态，则此选项将报告这些子进程。

可用性: Unix。

下列函数采用进程状态码作为参数，状态码由*system()*、*wait()* 或*waitpid()* 返回。它们可用于确定进程上发生的操作。

os.**WCOREDUMP**(*status*)
如果为该进程生成了核心转储，返回 `True`，否则返回 `False`。

此函数应当仅在*WIFSIGNALED()* 为真值时使用。

可用性: Unix。

os.**WIFCONTINUED**(*status*)
如果一个已停止的子进程通过传送*SIGCONT* 获得恢复（如果该进程是从任务控制停止后再继续的）则返回 True，否则返回 False。

参见*WCONTINUED* 选项。

可用性: Unix。

os.**WIFSTOPPED**(*status*)
如果进程是通过传送一个信号来停止的则返回 True，否则返回 False。

WIFSTOPPED() 只有在当*waitpid()* 调用是通过使用*WUNTRACED* 选项来完成或者当该进程正被追踪时 (参见 *ptrace(2)*) 才返回 True。

可用性: Unix。

os.**WIFSIGNALED**(*status*)
如果进程是通过一个信号来终止的则返回 True ，否则返回 False。

可用性: Unix。

os.**WIFEXITED**(*status*)
如果进程正常终止退出则返回 True，也就是说通过调用 exit() 或 _exit()，或者通过从 main() 返回；在其他情况下则返回 False。

可用性: Unix。

os.**WEXITSTATUS**(*status*)
返回进程退出状态。

此函数应当仅在*WIFEXITED()* 为真值时使用。

可用性: Unix。

os.**WSTOPSIG**(*status*)
返回导致进程停止的信号。

此函数应当仅在*WIFSTOPPED()* 为真值时使用。

可用性: Unix。

os.**WTERMSIG**(*status*)
返回导致进程终止的信号的编号。

此函数应当仅在*WIFSIGNALED()* 为真值时使用。

可用性: Unix。

16.1.7 调度器接口

这些函数控制操作系统如何为进程分配 CPU 时间。它们仅在某些 Unix 平台上可用。更多细节信息请查阅你所用 Unix 的指南页面。

3.3 版新加入.

以下调度策略如果被操作系统支持就会对外公开。

os.**SCHED_OTHER**
默认调度策略。

os.**SCHED_BATCH**
用于 CPU 密集型进程的调度策略，它会尽量为计算机中的其余任务保留交互性。

os.**SCHED_IDLE**
用于极低优先级的后台任务的调度策略。

os.**SCHED_SPORADIC**
用于偶发型服务程序的调度策略。

os.**SCHED_FIFO**
先进先出的调度策略。

os.**SCHED_RR**
循环式的调度策略。

os.**SCHED_RESET_ON_FORK**
此旗标可与任何其他调度策略进行 OR 运算。当带有此旗标的进程设置分叉时，其子进程的调度策略和优先级会被重置为默认值。

class os.**sched_param**(*sched_priority*)
这个类表示在*sched_setparam()*, *sched_setscheduler()* 和*sched_getparam()* 中使用的可修改调度形参。它属于不可变对象。

目前它只有一个可能的形参：

sched_priority
一个调度策略的调度优先级。

os.**sched_get_priority_min**(*policy*)
获取 *policy* 的最小优先级数值。*policy* 是以上调度策略常量之一。

os.**sched_get_priority_max**(*policy*)
获取 *policy* 的最高优先级数值。*policy* 是以上调度策略常量之一。

os.**sched_setscheduler**(*pid*, *policy*, *param*)
设置 PID 为 *pid* 的进程的调度策略。*pid* 为 0 指的是调用本方法的进程。*policy* 是以上调度策略常量之一。*param* 是一个*sched_param* 实例。

os.**sched_getscheduler**(*pid*)
返回 PID 为 *pid* 的进程的调度策略。*pid* 为 0 指的是调用本方法的进程。返回的结果是以上调度策略常量之一。

os.**sched_setparam**(*pid*, *param*)
设置 PID 为 *pid* 的进程的调度参数。*pid* 为 0 表示调用方过程。*param* 是一个*sched_param* 实例。

os.**sched_getparam**(*pid*)
返回 PID 为 *pid* 的进程的调度参数为一个*sched_param* 实例。*pid* 为 0 指的是调用本方法的进程。

os.**sched_rr_get_interval**(*pid*)
返回 PID 为 *pid* 的进程在时间片轮转调度下的时间片长度（单位为秒）。*pid* 为 0 指的是调用本方法的进程。

os.**sched_yield**()
自愿放弃 CPU。

os.**sched_setaffinity**(*pid*, *mask*)
将 PID 为 *pid* 的进程（为零则为当前进程）限制到一组 CPU 上。*mask* 是整数的可迭代对象，表示应将进程限制在其中的一组 CPU。

os.**sched_getaffinity**(*pid*)
返回 PID 为 *pid* 的进程（为零则为当前进程）被限制到的那一组 CPU。

16.1.8 其他系统信息

os.**confstr**(*name*)
返回字符串格式的系统配置信息。*name* 指定要查找的配置名称，它可以是字符串，是一个系统已定义的名称，这些名称定义在不同标准（POSIX，Unix 95，Unix 98 等）中。一些平台还定义了额外的其他名称。当前操作系统已定义的名称在 `confstr_names` 字典的键中给出。对于未包含在该映射中的配置名称，也可以传递一个整数作为 *name*。

如果 *name* 指定的配置值未定义，返回 `None`。

如果 *name* 是一个字符串且不是已定义的名称，将抛出*ValueError* 异常。如果当前系统不支持 *name* 指定的配置名称，即使该名称存在于 `confstr_names`，也会抛出*OSError* 异常，错误码为*errno.EINVAL*。

可用性: Unix。

os.**confstr_names**
字典，表示映射关系，为*confstr()* 可接受名称与操作系统为这些名称定义的整数值之间的映射。这可用于判断系统已定义了哪些名称。

可用性: Unix。

os.**cpu_count**()
返回系统的 CPU 数量。不确定则返回 `None`。

该数量不同于当前进程可以使用的 CPU 数量。可用的 CPU 数量可以由 `len(os.sched_getaffinity(0))` 方法获得。

3.4 版新加入.

os.**getloadavg**()
返回系统运行队列中最近 1、5 和 15 分钟内的平均进程数。无法获得平均负载则抛出*OSError* 异常。

可用性: Unix。

os.**sysconf**(*name*)
返回整数格式的系统配置信息。如果 *name* 指定的配置值未定义，返回 `-1`。对*confstr()* 的 *name* 参数的注释在此处也适用。当前已知的配置名称在 `sysconf_names` 字典中提供。

可用性: Unix。

os.**sysconf_names**
字典，表示映射关系，为*sysconf()* 可接受名称与操作系统为这些名称定义的整数值之间的映射。这可用于判断系统已定义了哪些名称。

可用性: Unix。

以下数据值用于支持对路径本身的操作。所有平台都有定义。

对路径的高级操作在*os.path* 模块中定义。

os.**curdir**
操作系统用来表示当前目录的常量字符串。在 Windows 和 POSIX 上是 `'.'`。在*os.path* 中也可用。

os.**pardir**
操作系统用来表示父目录的常量字符串。在 Windows 和 POSIX 上是 `'..'`。在*os.path* 中也可用。

os.**sep**
操作系统用来分隔路径不同部分的字符。在 POSIX 上是 `'/'`，在 Windows 上是是 `'\\'`。注意，仅了解它不足以能解析或连接路径，请使用*os.path.split()* 和*os.path.join()*，但它有时是有用的。在*os.path* 中也可用。

os.**altsep**
操作系统用来分隔路径不同部分的替代字符。如果仅存在一个分隔符，则为 None。在 sep 是反斜杠的 Windows 系统上，该值被设为 '/'。在*os.path* 中也可用。

os.**extsep**
分隔基本文件名与扩展名的字符，如 os.py 中的 '.'。在*os.path* 中也可用。

os.**pathsep**
操作系统通常用于分隔搜索路径（如 PATH）中不同部分的字符，如 POSIX 上是 ':'，Windows 上是 ';'。在*os.path* 中也可用。

os.**defpath**
在环境变量没有 'PATH' 键的情况下，*exec*p** and *spawn*p** 使用的默认搜索路径。在*os.path* 中也可用。

os.**linesep**
当前平台用于分隔（或终止）行的字符串。它可以是单个字符，如 POSIX 上是 '\n'，也可以是多个字符，如 Windows 上是 '\r\n'。在写入以文本模式（默认模式）打开的文件时，请不要使用 *os.linesep* 作为行终止符，请在所有平台上都使用一个 '\n' 代替。

os.**devnull**
空设备的文件路径。如 POSIX 上为 '/dev/null'，Windows 上为 'nul'。在*os.path* 中也可用。

os.**RTLD_LAZY**
os.**RTLD_NOW**
os.**RTLD_GLOBAL**
os.**RTLD_LOCAL**
os.**RTLD_NODELETE**
os.**RTLD_NOLOAD**
os.**RTLD_DEEPBIND**
setdlopenflags() 和*getdlopenflags()* 函数所使用的标志。请参阅 Unix 手册页 *dlopen(3)* 获取不同标志的含义。

3.3 版新加入.

16.1.9 随机数

os.**getrandom**(*size*, *flags=0*)
获得最多为 *size* 的随机字节。本函数返回的字节数可能少于请求的字节数。

这些字节可用于为用户空间的随机数生成器提供种子，或用于加密目的。

getrandom() 依赖于从设备驱动程序和其他环境噪声源收集的熵。不必要地读取大量数据将对使用 /dev/random 和 /dev/urandom 设备的其他用户产生负面影响。

flags 参数是一个位掩码，可以是零个或多个下列值以或运算组合：*os.GRND_RANDOM* 和*GRND_NONBLOCK*。

另请参阅 Linux getrandom() 手册页 。

可用性：Linux 3.17 或更高版本。

3.6 版新加入.

os.**urandom**(*size*)
Return a bytestring of *size* random bytes suitable for cryptographic use.

本函数从系统指定的随机源获取随机字节。对于加密应用程序，返回的数据应有足够的不可预测性，尽管其确切的品质取决于操作系统的实现。

在 Linux 上，如果 getrandom() 系统调用可用，它将以阻塞模式使用：阻塞直到系统的 urandom 熵池初始化完毕（内核收集了 128 位熵）。原理请参阅 **PEP 524**。在 Linux 上，*getrandom()* 可以以非阻塞模式（使用*GRND_NONBLOCK* 标志）获取随机字节，或者轮询直到系统的 urandom 熵池初始化完毕。

在类 Unix 系统上，随机字节是从 /dev/urandom 设备读取的。如果 /dev/urandom 设备不可用或不可读，则抛出*NotImplementedError* 异常。

在 Windows 上将使用 CryptGenRandom()。

也参考:

secrets 模块提供了更高级的功能。所在平台会提供随机数生成器，有关其易于使用的接口，请参阅*random.SystemRandom*。

3.6.0 版更變: 在 Linux 上，getrandom() 现在以阻塞模式使用，以提高安全性。

3.5.2 版更變: 在 Linux 上，如果 getrandom() 系统调用阻塞（urandom 熵池尚未初始化完毕），则退回一步读取 /dev/urandom。

3.5 版更變: 在 Linux 3.17 和更高版本上，现在使用 getrandom() 系统调用（如果可用）。在 OpenBSD 5.6 和更高版本上，现在使用 getentropy() C 函数。这些函数避免了使用内部文件描述符。

os.**GRND_NONBLOCK**

默认情况下，从 /dev/random 读取时，如果没有可用的随机字节，则*getrandom()* 会阻塞；从 /dev/urandom 读取时，如果熵池尚未初始化，则会阻塞。

如果设置了*GRND_NONBLOCK* 标志，则这些情况下*getrandom()* 不会阻塞，而是立即抛出*BlockingIOError* 异常。

3.6 版新加入.

os.**GRND_RANDOM**

如果设置了此标志位，那么将从 /dev/random 池而不是 /dev/urandom 池中提取随机字节。

3.6 版新加入.

16.2 io --- 处理流的核心工具

源代码: Lib/io.py

16.2.1 總覽

io 模块提供了 Python 用于处理各种 I/O 类型的主要工具。三种主要的 I/O 类型分别为: *文本 I/O*, *二进制 I/O* 和 *原始 I/O*。这些是泛型类型，有很多种后端存储可以用在他们上面。一个隶属于任何这些类型的具体对象被称作*file object*。其他同类的术语还有 *流*和 *类文件对象*。

独立于其类别，每个具体流对象也将具有各种功能：它可以是只读，只写或读写。它还可以允许任意随机访问（向前或向后寻找任何位置），或仅允许顺序访问（例如在套接字或管道的情况下）。

所有流对提供给它们的数据类型都很敏感。例如将*str* 对象给二进制流的 write() 方法会引发*TypeError*。将*bytes* 对象提供给文本流的 write() 方法也是如此。

3.3 版更變: 由于*IOError* 现在是*OSError* 的别名，因此用于引发*IOError* 的操作现在会引发*OSError* 。

文本 I/O

文本 I/O 预期并生成*str* 对象。这意味着，无论何时后台存储是由字节组成的（例如在文件的情况下），数据的编码和解码都是透明的，并且可以选择转换特定于平台的换行符。

创建文本流的最简单方法是使用*open()*，可以选择指定编码：

```
f = open("myfile.txt", "r", encoding="utf-8")
```

内存中文本流也可以作为*StringIO* 对象使用：

```
f = io.StringIO("some initial text data")
```

TextIOBase 的文档中详细描述了文本流的 API

二进制 I/O

二进制 I/O（也称为缓冲 I/O）预期*bytes-like objects* 并生成*bytes* 对象。不执行编码、解码或换行转换。这种类型的流可以用于所有类型的非文本数据，并且还可以在需要手动控制文本数据的处理时使用。

创建二进制流的最简单方法是使用*open()*，并在模式字符串中指定 `'b'`：

```
f = open("myfile.jpg", "rb")
```

内存中二进制流也可以作为*BytesIO* 对象使用：

```
f = io.BytesIO(b"some initial binary data: \x00\x01")
```

BufferedIOBase 的文档中详细描述了二进制流的 API

其他库模块可以提供额外的方式来创建文本或二进制流。参见*socket.socket.makefile()* 的示例。

原始 I/O

原始 I/O（也称为 *非缓冲 I/O*）通常用作二进制和文本流的低级构建块。用户代码直接操作原始流的用法非常罕见。不过，可以通过在禁用缓冲的情况下以二进制模式打开文件来创建原始流：

```
f = open("myfile.jpg", "rb", buffering=0)
```

RawIOBase 的文档中详细描述了原始流的 API

16.2.2 高阶模块接口

io.**DEFAULT_BUFFER_SIZE**
包含模块缓冲 I/O 类使用的默认缓冲区大小的整数。（如果可能）*open()* 将使用文件的 blksize（由*os.stat()* 获得）。

io.**open**(*file*, *mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *closefd=True*, *opener=None*)
这是内置的*open()* 函数的别名。

`open` 附带参数 `path`、`mode`、`flags` 会引发审计事件。

io.**open_code**(*path*)
以 'rb' 模式打开提供的文件。如果目的是将文件内容做为可执行代码，则应使用此函数。

path 应当为*str* 类型并且是一个绝对路径。

此函数的行为可以由对 PyFile_SetOpenCodeHook() 的先期调用所重载。但是，如果 path 为*str* 类型并且是一个绝对路径，open_code(path) 的行为应当总是与 open(path, 'rb') 一致。重载此行为的目的是为了给文件附加额外的验证或预处理。

3.8 版新加入.

exception io.**BlockingIOError**
这是内置的*BlockingIOError* 异常的兼容性别名。

exception io.**UnsupportedOperation**
在流上调用不支持的操作时引发的继承*OSError* 和*ValueError* 的异常。

也参考:

sys 包含标准 IO 流: *sys.stdin*, *sys.stdout* 和*sys.stderr* 。

16.2.3 类的层次结构

I/O 流被安排为按类的层次结构实现。首先是抽象基类 (ABC)，用于指定流的各种类别，然后是提供标准流实现的具体类。

> **備F:** 抽象基类还提供某些方法的默认实现，以帮助实现具体的流类。例如*BufferedIOBase* 提供了 readinto() 和*readline()* 的未优化实现。

I/O 层次结构的顶部是抽象基类*IOBase* 。它定义了流的基本接口。但是请注意，对流的读取和写入之间没有分离。如果实现不支持指定的操作，则会引发*UnsupportedOperation* 。

抽象基类*RawIOBase* 是*IOBase* 的子类。它负责将字节读取和写入流中。*RawIOBase* 的子类*FileIO* 提供计算机文件系统中文件的接口。

抽象基类*BufferedIOBase* 继承了*IOBase* ，处理原始二进制流（*RawIOBase* ）上的缓冲。其子类*BufferedWriter* 、*BufferedReader* 和*BufferedRWPair* 分别缓冲可读、可写以及可读写的原始二进制流。*BufferedRandom* 提供了带缓冲的可随机访问流接口。*BufferedIOBase* 的另一个子类*BytesIO* 是内存中字节流。

抽象基类*TextIOBase* 继承了*IOBase* 。它处理可表示文本的流，并处理字符串的编码和解码。类*TextIOWrapper* 继承了*TextIOBase* ，是原始缓冲流（*BufferedIOBase* ）的缓冲文本接口。最后，*StringIO* 是文本的内存流。

参数名不是规范的一部分，只有*open()* 的参数才用作关键字参数。

下表总结了抽象基类提供的*io* 模块:

抽象基类	继承	抽象方法	Mixin 方法和属性
IOBase		fileno, seek, 和 truncate	close, closed, __enter__, __exit__, flush, isatty, __iter__, __next__, readable, readline, readlines, seekable, tell, writable 和 writelines
RawIOBase	*IOBase*	readinto 和 write	继承 *IOBase* 方法, read, 和 readall
BufferedIOBase	*IOBase*	detach, read, read1, 和 write	继承 *IOBase* 方法, readinto, 和 readinto1
TextIOBase	*IOBase*	detach, read, readline, 和 write	继承 *IOBase* 方法, encoding, errors, 和 newlines

I/O 基类

class io.**IOBase**

The abstract base class for all I/O classes.

此类为许多方法提供了空的抽象实现，派生类可以有选择地重写。默认实现代表一个无法读取、写入或查找的文件。

尽管 *IOBase* 没有声明 read() 或 write()，因为它们的签名会有所不同，但是实现和客户端应该将这些方法视为接口的一部分。此外，当调用不支持的操作时可能会引发 *ValueError*（或 *UnsupportedOperation*）。

从文件读取或写入文件的二进制数据的基本类型为 *bytes*。其他*bytes-like objects* 也可以作为方法参数。文本 I/O 类使用 *str* 数据。

请注意，在关闭的流上调用任何方法（甚至查询）都是未定义的（undefined）。在这种情况下，实现可能会引发 *ValueError*。

IOBase（及其子类）支持迭代器协议，这意味着可以迭代 *IOBase* 对象以产生流中的行。根据流是二进制流（产生字节）还是文本流（产生字符串），行的定义略有不同。请参见下面的 *readline()*。

IOBase 也是一个上下文管理器，因此支持 with 语句。在这个示例中，*file* 将在 with 语句块执行完成之后被关闭 --- 即使是发生了异常:

```
with open('spam.txt', 'w') as file:
    file.write('Spam and eggs!')
```

IOBase 提供以下数据属性和方法:

close()

刷新并关闭此流。如果文件已经关闭，则此方法无效。文件关闭后，对文件的任何操作（例如读取或写入）都会引发 *ValueError*。

为方便起见，允许多次调用此方法。但是，只有第一个调用才会生效。

closed

如果流已关闭，则返回 True。

fileno()

返回流的底层文件描述符（整数）---如果存在。如果 IO 对象不使用文件描述符，则会引发 *OSError*。

flush()
刷新流的写入缓冲区（如果适用）。这对只读和非阻塞流不起作用。

isatty()
如果流是交互式的（即连接到终端/tty 设备），则返回 True 。

readable()
如果可以读取流，则返回 True 。否则为 False ，且 read() 将引发*OSError* 错误。

readline(*size=-1*, */*)
从流中读取并返回一行。如果指定了 *size*，将至多读取 *size* 个字节。

对于二进制文件行结束符总是 b'\n'；对于文本文件，可以用将 *newline* 参数传给*open()* 的方式来选择要识别的行结束符。

readlines(*hint=-1*, */*)
从流中读取并返回包含多行的列表。可以指定 *hint* 来控制要读取的行数：如果（以字节/字符数表示的）所有行的总大小超出了 *hint* 则将不会读取更多的行。

0 或更小的 *hint* 值以及 None，会被视为没有 hint。

请注意使用 for line in file: ... 就足够对文件对象进行迭代了，可以不必调用 file.readlines()。

seek(*offset*, *whence=SEEK_SET*, */*)
将流位置修改到给定的字节 *offset*。*offset* 将相对于由 *whence* 指定的位置进行解析。*whence* 的默认值为 SEEK_SET。*whence* 的可用值有：

- SEEK_SET 或 0 -- 流的开头（默认值）；*offset* 应为零或正值
- SEEK_CUR or 1 -- 当前流位置；*offset* 可以为负值
- SEEK_END or 2 -- 流的末尾；*offset* 通常为负值

返回新的绝对位置。

3.1 版新加入: SEEK_* 常量.

3.3 版新加入: 某些操作系统还可支持其他的值，例如 os.SEEK_HOLE 或 os.SEEK_DATA。特定文件的可用值还会取决于它是以文本还是二进制模式打开。

seekable()
如果流支持随机访问则返回 True。如为 False，则*seek()*, *tell()* 和*truncate()* 将引发*OSError*。

tell()
返回当前流的位置。

truncate(*size=None*, */*)
将流的大小调整为给定的 *size* 个字节（如果未指定 *size* 则调整至当前位置）。当前的流位置不变。这个调整操作可扩展或减小当前文件大小。在扩展的情况下，新文件区域的内容取决于具体平台（在大多数系统上，额外的字节会填充为零）。返回新的文件大小。

3.5 版更變: 现在 Windows 在扩展时将文件填充为零。

writable()
如果流支持写入则返回 True。如为 False，则 write() 和*truncate()* 将引发*OSError*。

writelines(*lines*, */*)
将行列表写入到流。不会添加行分隔符，因此通常所提供的每一行都带有末尾行分隔符。

__del__()
为对象销毁进行准备。*IOBase* 提供了此方法的默认实现，该实现会调用实例的*close()* 方法。

class io.**RawIOBase**
Base class for raw binary streams. It inherits *IOBase*.

原始二进制流通常会提供对下层 OS 设备或 API 的低层级访问，而不是尝试将其封装到高层级的基元中（此功能是在更高层级的缓冲二进制流和文本流中实现的，将在下文中描述）。

RawIOBase 在*IOBase* 的现有成员以外还提供了下列方法:

read(*size=-1*, /)
从对象中读取 *size* 个字节并将其返回。作为一个便捷选项，如果 *size* 未指定或为 -1，则返回所有字节直到 EOF。在其他情况下，仅会执行一次系统调用。如果操作系统调用返回字节数少于 *size* 则此方法也可能返回少于 *size* 个字节。

如果返回 0 个字节而 *size* 不为零 0，这表明到达文件末尾。如果处于非阻塞模式并且没有更多字节可用，则返回 None。

默认实现会转至*readall()* 和*readinto()*。

readall()
从流中读取并返回所有字节直到 EOF，如有必要将对流执行多次调用。

readinto(*b*, /)
将字节数据读入预先分配的可写*bytes-like object* *b*，并返回所读取的字节数。例如，*b* 可以是一个*bytearray*。如果对象处理非阻塞模式并且没有更多字节可用，则返回 None。

write(*b*, /)
将给定的*bytes-like object* *b* 写入到下层的原始流，并返回所写入的字节数。这可以少于 *b* 的总字节数，具体取决于下层原始流的设定，特别是如果它处于非阻塞模式的话。如果原始流设为非阻塞并且不能真正向其写入单个字节时则返回 None。调用者可以在此方法返回后释放或改变 *b*，因此该实现应该仅在方法调用期间访问 *b*。

class io.**BufferedIOBase**
Base class for binary streams that support some kind of buffering. It inherits *IOBase*.

与*RawIOBase* 的主要差别在于*read()*, *readinto()* 和*write()* 等方法将（分别）尝试按照要求读取尽可能多的输入或是耗尽所有给定的输出，其代价是可能会执行一次以上的系统调用。

除此之外，那些方法还可能引发*BlockingIOError*，如果下层的原始数据流处于非阻塞模式并且无法接受或给出足够数据的话；不同于对应的*RawIOBase* 方法，它们将永远不会返回 None。

并且，*read()* 方法也没有转向*readinto()* 的默认实现。

典型的*BufferedIOBase* 实现不应当继承自*RawIOBase* 实现，而要包装一个该实现，正如*BufferedWriter* 和*BufferedReader* 所做的那样。

BufferedIOBase 在*IOBase* 的现有成员以外还提供或重载了下列数据属性和方法:

raw
由*BufferedIOBase* 处理的下层原始流 (*RawIOBase* 的实例)。它不是*BufferedIOBase* API 的组成部分并且不存在于某些实现中。

detach()
从缓冲区分离出下层原始流并将其返回。

在原始流被分离之后，缓冲区将处于不可用的状态。

某些缓冲区例如*BytesIO* 并无可从此方法返回的单独原始流的概念。它们将会引发*UnsupportedOperation*。

3.1 版新加入.

read(*size=-1*, /)
读取并返回最多 *size* 个字节。如果此参数被省略、为 None 或为负值，则读取并返回所有数据直到 EOF。如果流已经到达 EOF 则返回一个空的*bytes* 对象。

如果此参数为正值，并且下层原始流不可交互，则可能发起多个原始读取以满足字节计数（直至先遇到 EOF）。但对于可交互原始流，则将至多发起一个原始读取，并且简短的结果并不意味着已到达 EOF。

BlockingIOError 会在下层原始流不处于阻塞模式，并且当前没有可用数据时被引发。

read1(*size=-1*, */*)
通过至多一次对下层流的*read()* (或*readinto()*) 方法的调用读取并返回至多 *size* 个字节。这适用于在*BufferedIOBase* 对象之上实现你自己的缓冲区的情况。

如果 *size* 为 -1 (默认值)，则返回任意数量的字节（多于零字节，除非已到达 EOF）。

readinto(*b*, */*)
将字节数据读入预先分配的可写*bytes-like object* *b* 并返回所读取的字节数。例如，*b* 可以是一个*bytearray*。

类似于*read()*，可能对下层原始流发起多次读取，除非后者为交互式。

BlockingIOError 会在下层原始流不处于阻塞模式，并且当前没有可用数据时被引发。

readinto1(*b*, */*)
将字节数据读入预先分配的可写*bytes-like object* *b*，其中至多使用一次对下层原始流*read()* (或*readinto()*) 方法的调用。返回所读取的字节数。

BlockingIOError 会在下层原始流不处于阻塞模式，并且当前没有可用数据时被引发。

3.5 版新加入.

write(*b*, */*)
写入给定的*bytes-like object* *b*，并返回写入的字节数 (总是等于 *b* 的字节长度，因为如果写入失败则会引发*OSError*)。根据具体实现的不同，这些字节可能被实际写入下层流，或是出于运行效率和冗余等考虑而暂存于缓冲区。

当处于非阻塞模式时，如果需要将数据写入原始流但它无法在不阻塞的情况下接受所有数据则将引发*BlockingIOError*。

调用者可能会在此方法返回后释放或改变 *b*，因此该实现应当仅在方法调用期间访问 *b*。

原始文件 I/O

class io.**FileIO**(*name*, *mode='r'*, *closefd=True*, *opener=None*)
代表一个包含字节数据的 OS 层级文件的原始二进制流。它继承自*RawIOBase*。

name 可以是以下两项之一：

- 代表将被打开的文件路径的字符串或*bytes* 对象。在此情况下 closefd 必须为 `True` (默认值) 否则将会引发异常。
- 代表一个现有 OS 层级文件描述符的号码的整数，作为结果的*FileIO* 对象将可访问该文件。当 FileIO 对象被关闭时此 fd 也将被关闭，除非 *closefd* 设为 `False`。

mode 可以为 `'r'`, `'w'`, `'x'` 或 `'a'` 分别表示读取（默认模式）、写入、独占新建或添加。如果以写入或添加模式打开的文件不存在将自动新建；当以写入模式打开时文件将先清空。以新建模式打开时如果文件已存在则将引发*FileExistsError*。以新建模式打开文件也意味着要写入，因此该模式的行为与 `'w'` 类似。在模式中附带 `'+'` 将允许同时读取和写入。

该类的 `read()` (当附带正值参数调用时), `readinto()` 和 `write()` 方法将只执行一次系统调用。

可以通过传入一个可调用对象作为 *opener* 来使用自定义文件打开器。然后通过调用 *opener* 并传入 (*name*, *flags*) 来获取文件对象所对应的下层文件描述符。*opener* 必须返回一个打开文件描述符（传入*os.open* 作为 *opener* 的结果在功能上将与传入 `None` 类似）。

新创建的文件是不可继承的。

有关 opener 参数的示例，请参见内置函数*open()* 。

3.3 版更變: 增加了 *opener* 参数。增加了 `'x'` 模式。

3.4 版更變: 文件现在禁止继承。

FileIO 在继承自*RawIOBase* 和*IOBase* 的现有成员以外还提供了以下数据属性和方法:

mode
构造函数中给定的模式。

name
文件名。当构造函数中没有给定名称时，这是文件的文件描述符。

缓冲流

相比原始 I/O，缓冲 I/O 流提供了针对 I/O 设备的更高层级接口。

class io.**BytesIO**(*initial_bytes=b''*)
一个使用内在字节缓冲区的二进制流。它继承自*BufferedIOBase*。在*close()* 方法被调用时将会丢弃缓冲区。

可选参数 *initial_bytes* 是一个包含初始数据的*bytes-like object*。

BytesIO 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重载了下列方法:

getbuffer()
返回一个对应于缓冲区内容的可读写视图而不必拷贝其数据。此外，改变视图将透明地更新缓冲区内容:

```
>>> b = io.BytesIO(b"abcdef")
>>> view = b.getbuffer()
>>> view[2:4] = b"56"
>>> b.getvalue()
b'ab56ef'
```

> **備🄴:** 只要视图保持存在，*BytesIO* 对象就无法被改变大小或关闭。

3.2 版新加入.

getvalue()
返回包含整个缓冲区内容的*bytes*。

read1(*size=-1*, */*)
在*BytesIO* 中，这与*read()* 相同。

3.7 版更變: *size* 参数现在是可选的。

readinto1(*b*, */*)
在*BytesIO* 中，这与*readinto()* 相同。

3.5 版新加入.

class io.**BufferedReader**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
一个提供对可读、不可查找的*RawIOBase* 原始二进制流的高层级访问的缓冲二进制流。它继承自*BufferedIOBase*。

当从此对象读取数据时，可能会从下层原始流请求更大量的数据，并存放到内部缓冲区中。接下来可以在后续读取时直接返回缓冲数据。

根据给定的可读 *raw* 流和 *buffer_size* 创建*BufferedReader* 的构造器。如果省略 *buffer_size*，则会使用*DEFAULT_BUFFER_SIZE*。

BufferedReader 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重载了下列方法:

peek(*size=0*, */*)
从流返回字节数据而不前移位置。完成此调用将至多读取一次原始流。返回的字节数量可能少于或多于请求的数量。

read(*size=-1*, */*)
读取并返回 *size* 个字节，如果 *size* 未给定或为负值，则读取至 EOF 或是在非阻塞模式下读取调用将会阻塞。

read1(*size=-1*, */*)
在原始流上通过单次调用读取并返回至多 *size* 个字节。如果至少缓冲了一个字节，则只返回缓冲的字节。在其他情况下，将执行一次原始流读取。

3.7 版更變: *size* 参数现在是可选的。

class io.**BufferedWriter**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
一个提供对可写、不可查找的*RawIOBase* 原始二进制流的高层级访问的缓冲二进制流。它继承自*BufferedIOBase*。

当写入到此对象时，数据通常会被放入到内部缓冲区中。缓冲区将在满足某些条件的情况下被写到下层的*RawIOBase* 对象，包括:

- 当缓冲区对于所有挂起数据而言太小时;
- 当*flush()* 被调用时
- 当（为*BufferedRandom* 对象）请求 seek() 时;
- 当*BufferedWriter* 对象被关闭或销毁时。

该构造器会为给定的可写 *raw* 流创建一个*BufferedWriter*。如果未给定 *buffer_size*，则使用默认的*DEFAULT_BUFFER_SIZE*。

BufferedWriter 在继承自*BufferedIOBase* 和*IOBase* 的成员以外还提供或重载了下列方法:

flush()
将缓冲区中保存的字节数据强制放入原始流。如果原始流发生阻塞则应当引发*BlockingIOError*。

write(*b*, */*)
写入*bytes-like object* *b* 并返回写入的字节数。当处于非阻塞模式时，如果缓冲区需要被写入但原始流发生阻塞则将引发*BlockingIOError*。

class io.**BufferedRandom**(*raw*, *buffer_size=DEFAULT_BUFFER_SIZE*)
一个提供对不可查找的*RawIOBase* 原始二进制流的高层级访问的缓冲二进制流。它继承自*BufferedReader* 和*BufferedWriter*。

该构造器会为在第一个参数中给定的可查找原始流创建一个读取器和写入器。如果省略 *buffer_size* 则使用默认的*DEFAULT_BUFFER_SIZE*。

BufferedRandom 能做到*BufferedReader* 或*BufferedWriter* 所能做的任何事。此外，还会确保实现 seek() 和 tell()。

class io.**BufferedRWPair**(*reader*, *writer*, *buffer_size=DEFAULT_BUFFER_SIZE*, */*)
一个提供对两个不可查找的*RawIOBase* 原始二进制流的高层级访问的缓冲二进制流 --- 一个可读，另一个可写。它继承自*BufferedIOBase*。

reader 和 *writer* 分别是可读和可写的*RawIOBase* 对象。如果省略 *buffer_size* 则使用默认的*DEFAULT_BUFFER_SIZE*。

BufferedRWPair 实现了*BufferedIOBase* 的所有方法，但*detach()* 除外，调用该方法将引发*UnsupportedOperation*。

警告： *BufferedRWPair* 不会尝试同步访问其下层的原始流。你不应当将传给它与读取器和写入器相同的对象；而要改用*BufferedRandom*。

文本 I/O

class io.**TextIOBase**

Base class for text streams. This class provides a character and line based interface to stream I/O. It inherits *IOBase*.

TextIOBase 在来自*IOBase* 的成员以外还提供或重载了以下数据属性和方法:

encoding

用于将流的字节串解码为字符串以及将字符串编码为字节串的编码格式名称。

errors

解码器或编码器的错误设置。

newlines

一个字符串、字符串元组或者 None，表示目前已经转写的新行。根据具体实现和初始构造器旗标的不同，此属性或许会不可用。

buffer

由*TextIOBase* 处理的下层二进制缓冲区（为一个*BufferedIOBase* 的实例）。它不是*TextIOBase* API 的组成部分并且不存在于某些实现中。

detach()

从*TextIOBase* 分离出下层二进制缓冲区并将其返回。

在下层缓冲区被分离后，*TextIOBase* 将处于不可用的状态。

某些*TextIOBase* 的实现，例如*StringIO* 可能并无下层缓冲区的概念，因此调用此方法将引发*UnsupportedOperation*。

3.1 版新加入.

read(*size=-1*, */*)

从流中读取至多 *size* 个字符并以单个*str* 的形式返回。如果 *size* 为负值或 None，则读取至 EOF。

readline(*size=-1*, */*)

读取至换行符或 EOF 并返回单个 str。如果流已经到达 EOF，则将返回一个空字符串。

如果指定了 *size*，最多将读取 *size* 个字符。

seek(*offset*, *whence=SEEK_SET*, */*)

将流位置改为给定的偏移位置 *offset*。具体行为取决于 *whence* 形参。*whence* 的默认值为 SEEK_SET。

- SEEK_SET 或 0: 从流的开始位置起查找（默认值）；*offset* 必须为*TextIOBase.tell()* 所返回的数值或为零。任何其他 *offset* 值都将导致未定义的行为。
- SEEK_CUR 或 1: ”查找” 到当前位置；*offset* 必须为零，表示无操作（所有其他值均不受支持）。
- SEEK_END 或 2: 查找到流的末尾；*offset* 必须为零（所有其他值均不受支持）。

以不透明数字形式返回新的绝对位置。

3.1 版新加入: SEEK_* 常量.

tell()
以不透明数字形式返回当前流的位置。该数字通常并不代表下层二进制存储中对应的字节数。

write(*s*, /)
将字符串 *s* 写入到流并返回写入的字符数。

class io.**TextIOWrapper**(*buffer*, *encoding=None*, *errors=None*, *newline=None*, *line_buffering=False*, *write_through=False*)
一个提供对*BufferedIOBase* 缓冲二进制流的高层级访问的缓冲文本流。它继承自*TextIOBase*。

encoding 给出流被解码或编码时将使用的编码格式。它默认为*locale.getpreferredencoding(False)*。

errors 是一个可选的字符串，它指明编码格式和编码格式错误的处理方式。传入 'strict' 将在出现编码格式错误时引发*ValueError* (默认值 None 具有相同的效果)，传入 'ignore' 将忽略错误。(请注意忽略编码格式错误会导致数据丢失。) 'replace' 会在出现错误数据时插入一个替换标记 (例如 '?')。'backslashreplace' 将把错误数据替换为一个反斜杠转义序列。在写入时，还可以使用 'xmlcharrefreplace' (替换为适当的 XML 字符引用) 或 'namereplace' (替换为 \N{...} 转义序列)。任何其他通过*codecs.register_error()* 注册的错误处理方式名称也可以被接受。

newline 控制行结束符处理方式。它可以为 None, '', '\n', '\r' 和 '\r\n'。其工作原理如下:

- 当从流读取输入时，如果 *newline* 为 None，则将启用*universal newlines* 模式。输入中的行结束符可以为 '\n', '\r' 或 '\r\n'，在返回给调用者之前它们会被统一转写为 '\n'。如果 *newline* 为 ''，也会启用通用换行模式，但行结束符会不加转写即返回给调用者。如果 *newline* 具有任何其他合法的值，则输入行将仅由给定的字符串结束，并且行结束符会不加转写即返回给调用者。
- 将输出写入流时，如果 *newline* 为 None，则写入的任何 '\n' 字符都将转换为系统默认行分隔符*os.linesep*。如果 *newline* 是 '' 或 '\n'，则不进行翻译。如果 *newline* 是任何其他合法值，则写入的任何 '\n' 字符将被转换为给定的字符串。

如果 *line_buffering* 为 True，则当一个写入调用包含换行符或回车时将会应用 flush()。

如果 *write_through* 为 True，对 write() 的调用会确保不被缓冲：在*TextIOWrapper* 对象上写入的任何数据会立即交给其下层的 *buffer* 来处理。

3.3 版更變: 已添加 *write_through* 参数

3.3 版更變: 默认的 *encoding* 现在将为 locale.getpreferredencoding(False) 而非 locale.getpreferredencoding()。不要使用*locale.setlocale()* 来临时改变区域编码格式，要使用当前区域编码格式而不是用户的首选编码格式。

TextIOWrapper 在继承自*TextIOBase* 和*IOBase* 的现有成员以外还提供了以下数据属性和方法:

line_buffering
是否启用行缓冲。

write_through
写入是否要立即传给下层的二进制缓冲。

3.7 版新加入.

reconfigure(**, encoding][, errors][, newline][, line_buffering][, write_through]*)
使用 *encoding*, *errors*, *newline*, *line_buffering* 和 *write_through* 的新设置来重新配置此文本流。

未指定的形参将保留当前设定，例外情况是当指定了 *encoding* 但未指定 *errors* 时将会使用 errors='strict'。

如果已经有数据从流中被读取则将无法再改变编码格式或行结束符。另一方面，在写入数据之后再改变编码格式则是可以的。

此方法会在设置新的形参之前执行隐式的流刷新。

3.7 版新加入.

class io.**StringIO**(*initial_value=''*, *newline='\n'*)
一个使用内存文本缓冲的文本流。它继承自*TextIOBase*。

当*close()* 方法被调用时将会丢弃文本缓冲区。

缓冲区的初始值可通过提供 *initial_value* 来设置。如果启用了行结束符转写，换行将以*write()* 所用的方式被编码。数据流位置将被设为缓冲区的开头。

newline 参数的规则与*TextIOWrapper* 所用的一致，不同之处在于当将输出写入到流时，如果 *newline* 为 None，则在所有平台上换行符都会被写入为 \n。

StringIO 在继承自*TextIOBase* 和*IOBase* 的现有成员以外还提供了以下方法:

getvalue()
返回一个包含缓冲区全部内容的 str。换行符会以与*read()* 相同的方式被编码，但是流的位置不会被改变。

用法示例:

```
import io

output = io.StringIO()
output.write('First line.\n')
print('Second line.', file=output)

# Retrieve file contents -- this will be
# 'First line.\nSecond line.\n'
contents = output.getvalue()

# Close object and discard memory buffer --
# .getvalue() will now raise an exception.
output.close()
```

class io.**IncrementalNewlineDecoder**
用于在*universal newlines* 模式下解码换行符的辅助编解码器。它继承自*codecs.IncrementalDecoder*。

16.2.4 性能

本节讨论所提供的具体 I/O 实现的性能。

二进制 I/O

即使在用户请求单个字节时，也只读取和写入大块数据。通过该方法，缓冲 I/O 隐藏了操作系统调用和执行无缓冲 I/O 例程时的任何低效性。增益取决于操作系统和执行的 I/O 类型。例如，在某些现代操作系统上(例如 Linux)，无缓冲磁盘 I/O 可以与缓冲 I/O 一样快。但最重要的是，无论平台和支持设备如何，缓冲 I/O 都能提供可预测的性能。因此，对于二进制数据，应首选使用缓冲的 I/O 而不是未缓冲的 I/O 。

文本 I/O

二进制存储（如文件）上的文本 I/O 比同一存储上的二进制 I/O 慢得多，因为它需要使用字符编解码器在 Unicode 和二进制数据之间进行转换。这在处理大量文本数据（如大型日志文件）时会变得非常明显。此外，由于使用的重构算法 `TextIOWrapper.tell()` 和 `TextIOWrapper.seek()` 都相当慢。

StringIO 是原生的内存 Unicode 容器，速度与*BytesIO* 相似。

多线程

FileIO 对象是线程安全的，只要它们封装的操作系统调用（比如 Unix 下的 `read(2)`）也是线程安全的。

二进制缓冲对象（例如*BufferedReader*, *BufferedWriter*, *BufferedRandom* 和*BufferedRWPair*）使用锁来保护其内部结构；因此，可以安全地一次从多个线程中调用它们。

TextIOWrapper 对象不再是线程安全的。

可重入性

二进制缓冲对象（*BufferedReader*，*BufferedWriter*，*BufferedRandom* 和*BufferedRWPair* 的实例）不是可重入的。虽然在正常情况下不会发生可重入调用，但仍可能会在*signal* 处理程序执行 I/O 时产生。如果线程尝试重入已经访问的缓冲对象，则会引发*RuntimeError*。注意，这并不禁止其他线程进入缓冲对象。

上面的内容隐式地扩展到文本文件中，因为*open()* 函数将把缓冲对象封装在*TextIOWrapper* 中。这包括标准流，因而也会影响内置的*print()* 函数。

16.3 `time` --- 时间的访问和转换

该模块提供了各种与时间相关的函数。相关功能还可以参阅*datetime* 和*calendar* 模块。

尽管所有平台皆可使用此模块，但模块内的函数并非所有平台都可用。此模块中定义的大多数函数的实现都是调用其所在平台的 C 语言库的同名函数。因为这些函数的语义可能因平台而异，所以使用时最好查阅对应平台的相关文档。

下面是一些术语和惯例的解释.

- *epoch* 是时间开始的点，其值取决于平台。对于 Unix，epoch 是 1970 年 1 月 1 日 00:00:00（UTC）。要找出给定平台上的 epoch，请查看 `time.gmtime(0)`。
- 术语 *纪元秒数* 是指自 epoch（纪元）时间点以来经过的总秒数，通常不包括 闰秒。在所有符合 POSIX 标准的平台上，闰秒都不会记录在总秒数中。
- 此模块中的函数可能无法处理纪元之前或遥远未来的日期和时间。“遥远未来”的定义由对应的 C 语言库决定；对于 32 位系统，它通常是指 2038 年及以后。
- 函数*strptime()* 在接收到 `%y` 格式代码时可以解析使用 2 位数表示的年份。当解析 2 位数年份时，函数会按照 POSIX 和 ISO C 标准进行年份转换：数值 69--99 被映射为 1969--1999；数值 0--68 被映射为 2000--2068。
- UTC 是协调世界时（Coordinated Universal Time）的缩写。它以前也被称为格林威治标准时间（GMT）。使用 UTC 而不是 CUT 作为缩写是英语与法语（Temps Universel Coordonné）之间妥协的结果，不是什么低级错误。

- DST 是夏令时（Daylight Saving Time）的缩写，在一年的某一段时间中将当地时间调整（通常）一小时。DST 的规则非常神奇（由当地法律确定），并且每年的起止时间都不同。C 语言库中有一个表格，记录了各地的夏令时规则（实际上，为了灵活性，C 语言库通常是从某个系统文件中读取这张表）。从这个角度而言，这张表是夏令时规则的唯一权威真理。
- 由于平台限制，各种实时函数的精度可能低于其值或参数所要求（或给定）的精度。例如，在大多数 Unix 系统上，时钟频率仅为每秒 50 或 100 次。
- 另一方面，*time()* 和*sleep()* 的精度优于它们的 Unix 等价物：时间表示为浮点数，*time()* 返回最准确的时间（使用 Unix `gettimeofday()` 如果可用），并且*sleep()* 将接受非零分数的时间（Unix `select()` 用于实现此功能，如果可用）。
- 时间值由*gmtime()*，*localtime()* 和*strptime()* 返回，并被*asctime()*，*mktime()* 和*strftime()* 接受，是一个 9 个整数的序列。*gmtime()*，*localtime()* 和*strptime()* 的返回值还提供各个字段的属性名称。

  请参阅*struct_time* 以获取这些对象的描述。

  3.3 版更變: 在平台支持相应的 `struct tm` 成员时，*struct_time* 类型被扩展提供 `tm_gmtoff` 和 `tm_zone` 属性。

  3.6 版更變: *struct_time* 的属性 `tm_gmtoff` 和 `tm_zone` 现在可在所有平台上使用。
- 使用以下函数在时间表示之间进行转换：

从	到	使用
自纪元以来的秒数	UTC 的*struct_time*	*gmtime()*
自纪元以来的秒数	本地时间的*struct_time*	*localtime()*
UTC 的*struct_time*	自纪元以来的秒数	*calendar.timegm()*
本地时间的*struct_time*	自纪元以来的秒数	*mktime()*

16.3.1 函数

`time.`**`asctime`**`([t])`

转换由*gmtime()* 或*localtime()* 所返回的表示时间的元组或*struct_time* 为以下形式的字符串: `'Sun Jun 20 23:21:05 1993'`。日期字段的长度为两个字符，如果日期只有一个数字则会以零填充，例如: `'Wed Jun 9 04:26:40 1993'`。

如果未提供 *t*，则会使用*localtime()* 所返回的当前时间。*asctime()* 不会使用区域设置信息。

備𝔽: 与同名的 C 函数不同，*asctime()* 不添加尾随换行符。

`time.`**`pthread_getcpuclockid`**`(thread_id)`

返回指定的 *thread_id* 的特定于线程的 CPU 时间时钟的 *clk_id* 。

使用*threading.Thread* 对象的*threading.get_ident()* 或*ident* 属性为 *thread_id* 获取合适的值。

警告: 传递无效的或过期的 *thread_id* 可能会导致未定义的行为，例如段错误。

可用性： Unix（有关详细信息，请参见 *pthread_getcpuclockid(3)* 的手册页）。

3.7 版新加入.

time.**clock_getres**(*clk_id*)
返回指定时钟 *clk_id* 的分辨率（精度）。有关 *clk_id* 的可接受值列表，请参阅*Clock ID 常量* 。

Availability: Unix.

3.3 版新加入.

time.**clock_gettime**(*clk_id*) → float
返回指定 *clk_id* 时钟的时间。有关 *clk_id* 的可接受值列表，请参阅*Clock ID 常量* 。

Availability: Unix.

3.3 版新加入.

time.**clock_gettime_ns**(*clk_id*) → int
与*clock_gettime()* 相似，但返回时间为纳秒。

Availability: Unix.

3.7 版新加入.

time.**clock_settime**(*clk_id*, *time: float*)
设置指定 *clk_id* 时钟的时间。目前，*CLOCK_REALTIME* 是 *clk_id* 唯一可接受的值。

Availability: Unix.

3.3 版新加入.

time.**clock_settime_ns**(*clk_id*, *time: int*)
与*clock_settime()* 相似，但设置时间为纳秒。

Availability: Unix.

3.7 版新加入.

time.**ctime**([*secs*])
转换以距离初始纪元的秒数表示的时间为以下形式的字符串: `'Sun Jun 20 23:21:05 1993'` 代表本地时间。日期字段的长度为两个字符，如果日期只有一个数字则会以零填充，例如: `'Wed Jun 9 04:26:40 1993'`。

如果 *secs* 未提供或为*None*，则使用*time()* 所返回的当前时间。`ctime(secs)` 等价于 `asctime(localtime(secs))`。*ctime()* 不会使用区域设置信息。

time.**get_clock_info**(*name*)
获取有关指定时钟的信息作为命名空间对象。支持的时钟名称和读取其值的相应函数是:

- `'monotonic'`: *time.monotonic()*
- `'perf_counter'`: *time.perf_counter()*
- `'process_time'`: *time.process_time()*
- `'thread_time'`: *time.thread_time()*
- `'time'`: *time.time()*

结果具有以下属性:

- *adjustable*： 如果时钟可以自动更改（例如通过 NTP 守护程序）或由系统管理员手动更改，则为 `True` ，否则为 `False` 。
- *implementation*： 用于获取时钟值的基础 C 函数的名称。有关可能的值，请参阅*Clock ID 常量* 。
- *monotonic*： 如果时钟不能倒退，则为 `True` ，否则为 `False` 。
- *resolution*： 以秒为单位的时钟分辨率（*float* ）

3.3 版新加入.

time.**gmtime**([*secs*])
将以自 epoch 开始的秒数表示的时间转换为 UTC 的*struct_time*，其中 dst 标志始终为零。如果未提供 *secs* 或为*None*，则使用*time()* 所返回的当前时间。一秒以内的小数将被忽略。有关*struct_time* 对象的说明请参见上文。有关此函数的逆操作请参阅*calendar.timegm()*。

time.**localtime**([*secs*])
与*gmtime()* 相似但转换为当地时间。如果未提供 *secs* 或为*None*，则使用由*time()* 返回的当前时间。当 DST 适用于给定时间时，dst 标志设置为 `1`。

localtime() may raise *OverflowError*, if the timestamp is outside the range of values supported by the platform C `localtime()` or `gmtime()` functions, and *OSError* on `localtime()` or `gmtime()` failure. It's common for this to be restricted to years between 1970 and 2038.

time.**mktime**(*t*)
这是*localtime()* 的反函数。它的参数是*struct_time* 或者完整的 9 元组（因为需要 dst 标志；如果它是未知的则使用 `-1` 作为 dst 标志），它表示 *local* 的时间，而不是 UTC。它返回一个浮点数，以便与*time()* 兼容。如果输入值不能表示为有效时间，则*OverflowError* 或*ValueError* 将被引发（这取决于 Python 或底层 C 库是否捕获到无效值）。它可以生成时间的最早日期取决于平台。

time.**monotonic**() → float
（以小数表示的秒为单位）返回一个单调时钟的值，即不能倒退的时钟。该时钟不受系统时钟更新的影响。返回值的参考点未被定义，因此只有两次调用之间的差值才是有效的。

3.3 版新加入.

3.5 版更變: 该功能现在始终可用且始终在系统范围内。

time.**monotonic_ns**() → int
与*monotonic()* 相似，但是返回时间为纳秒数。

3.7 版新加入.

time.**perf_counter**() → float
（以小数表示的秒为单位）返回一个性能计数器的值，即用于测量较短持续时间的具有最高有效精度的时钟。它会包括睡眠状态所消耗的时间并且作用于全系统范围。返回值的参考点未被定义，因此只有两次调用之间的差值才是有效的。

3.3 版新加入.

time.**perf_counter_ns**() → int
与*perf_counter()* 相似，但是返回时间为纳秒。

3.7 版新加入.

time.**process_time**() → float
（以小数表示的秒为单位）返回当前进程的系统和用户 CPU 时间的总计值。它不包括睡眠状态所消耗的时间。根据定义它只作用于进程范围。返回值的参考点未被定义，因此只有两次调用之间的差值才是有效的。

3.3 版新加入.

time.**process_time_ns**() → int
与*process_time()* 相似，但是返回时间为纳秒。

3.7 版新加入.

time.**sleep**(*secs*)
调用该方法的线程将被暂停执行 *secs* 秒。参数可以是浮点数，以表示更为精确的睡眠时长。由于任何捕获到的信号都会终止*sleep()* 引发的该睡眠过程并开始执行信号的处理例程，因此实际的暂停时长可能小于请求的时长；此外，由于系统需要调度其他活动，实际暂停时长也可能比请求的时间长。

3.5 版更變: 现在，即使该睡眠过程被信号中断，该函数也会保证调用它的线程至少会睡眠 *secs* 秒。信号处理例程抛出异常的情况除外。（欲了解我们做出这次改变的原因，请参见 **PEP 475**）

time.**strftime**(*format*[, *t*])

转换一个元组或*struct_time*表示的由*gmtime()*或*localtime()*返回的时间到由 *format* 参数指定的字符串。如果未提供 *t*，则使用由*localtime()*返回的当前时间。*format* 必须是一个字符串。如果 *t* 中的任何字段超出允许范围，则引发*ValueError*。

0 是时间元组中任何位置的合法参数；如果它通常是非法的，则该值被强制改为正确的值。

以下指令可以嵌入 *format* 字符串中。它们显示时没有可选的字段宽度和精度规范，并被*strftime()*结果中的指示字符替换：

指令	意义	F解
%a	本地化的缩写星期中每日的名称。	
%A	本地化的星期中每日的完整名称。	
%b	本地化的月缩写名称。	
%B	本地化的月完整名称。	
%c	本地化的适当日期和时间表示。	
%d	十进制数 [01,31] 表示的月中日。	
%H	十进制数 [00,23] 表示的小时（24 小时制）。	
%I	十进制数 [01,12] 表示的小时（12 小时制）。	
%j	十进制数 [001,366] 表示的年中日。	
%m	十进制数 [01,12] 表示的月。	
%M	十进制数 [00,59] 表示的分钟。	
%p	本地化的 AM 或 PM 。	(1)
%S	十进制数 [00,61] 表示的秒。	(2)
%U	十进制数 [00,53] 表示的一年中的周数（星期日作为一周的第一天）。在第一个星期日之前的新年中的所有日子都被认为是在第 0 周。	(3)
%w	十进制数 [0(星期日),6] 表示的周中日。	
%W	十进制数 [00,53] 表示的一年中的周数（星期一作为一周的第一天）。在第一个星期一之前的新年中的所有日子被认为是在第 0 周。	(3)
%x	本地化的适当日期表示。	
%X	本地化的适当时间表示。	
%y	十进制数 [00,99] 表示的没有世纪的年份。	
%Y	十进制数表示的带世纪的年份。	
%z	Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].[1]	
%Z	Time zone name (no characters if no time zone exists). Deprecated.[1]	
%%	字面的 '%' 字符。	

F解：

(1) 当与*strptime()*函数一起使用时，如果使用 %I 指令来解析小时，%p 指令只影响输出小时字段。

(2) 范围真的是 0 到 61；值 60 在表示 leap seconds 的时间戳中有效，并且由于历史原因支持值 61。

(3) 当与*strptime()*函数一起使用时，%U 和 %W 仅用于指定星期几和年份的计算。

Here is an example, a format for dates compatible with that specified in the **RFC 2822** Internet email standard.[1]

[1] The use of %Z is now deprecated, but the %z escape that expands to the preferred hour/minute offset is not supported by all ANSI C libraries. Also, a strict reading of the original 1982 **RFC 822** standard calls for a two-digit year (%y rather than %Y), but practice moved to 4-digit years long before the year 2000. After that, **RFC 822** became obsolete and the 4-digit year has been first recommended by **RFC 1123** and then mandated by **RFC 2822**.

```
>>> from time import gmtime, strftime
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Thu, 28 Jun 2001 14:17:15 +0000'
```

某些平台可能支持其他指令，但只有此处列出的指令具有 ANSI C 标准化的含义。要查看平台支持的完整格式代码集，请参阅 *strftime(3)* 文档。

在某些平台上，可选的字段宽度和精度规范可以按照以下顺序紧跟在指令的初始 '%' 之后；这也不可移植。字段宽度通常为 2，除了 %j，它是 3。

time.**strptime**(*string*[, *format*])

根据格式解析表示时间的字符串。返回值为一个被*gmtime()* 或*localtime()* 返回的*struct_time* 。

format 参数使用与*strftime()* 相同的指令。它默认为匹配*ctime()* 所返回的格式 "%a %b %d %H:%M:%S %Y"` 。如果 *string* 不能根据 *format* 来解析，或者解析后它有多余的数据，则会引发*ValueError*。当无法推断出更准确的值时，用于填充任何缺失数据的默认值是 (1900, 1, 1, 0, 0, 0, 0, 1, -1) 。*string* 和 *format* 都必须为字符串。

例如:

```
>>> import time
>>> time.strptime("30 Nov 00", "%d %b %y")
time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0,
                 tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
```

支持 %Z 指令是基于 tzname 中包含的值以及 daylight 是否为真。因此，它是特定于平台的，除了识别始终已知的 UTC 和 GMT （并且被认为是非夏令时时区）。

仅支持文档中指定的指令。因为每个平台都实现了 strftime()，它有时会提供比列出的指令更多的指令。但是 strptime() 独立于任何平台，因此不一定支持所有未记录为支持的可用指令。

class time.**struct_time**

返回的时间值序列的类型为*gmtime()* 、*localtime()* 和*strptime()* 。它是一个带有*named tuple* 接口的对象：可以通过索引和属性名访问值。存在以下值:

索引	属性	值
0	tm_year	（例如，1993）
1	tm_mon	range [1, 12]
2	tm_mday	range [1, 31]
3	tm_hour	range [0, 23]
4	tm_min	range [0, 59]
5	tm_sec	range [0, 61]；见*strftime()* 介绍中的 **(2)**
6	tm_wday	range [0, 6]，周一为 0
7	tm_yday	range [1, 366]
8	tm_isdst	0, 1 或 -1；如下所示
N/A	tm_zone	时区名称的缩写
N/A	tm_gmtoff	以秒为单位的 UTC 以东偏离

请注意，与 C 结构不同，月份值是 [1,12] 的范围，而不是 [0,11] 。

在调用*mktime()* 时，tm_isdst 可以在夏令时生效时设置为 1，而在夏令时不生效时设置为 0。值-1 表示这是未知的，并且通常会导致填写正确的状态。

当一个长度不正确的元组被传递给期望*struct_time* 的函数，或者具有错误类型的元素时，会引发*TypeError* 。

time.**time**() → float
返回以浮点数表示的从*epoch* 开始的秒数的时间值。epoch 的具体日期和 leap seconds 的处理取决于平台。在 Windows 和大多数 Unix 系统中，epoch 是 1970 年 1 月 1 日 00:00:00 (UTC)，并且闰秒将不计入从 epoch 开始的秒数。这通常被称为 Unix 时间。要了解给定平台上 epoch 的具体定义，请查看 `gmtime(0)`。

请注意，即使时间总是作为浮点数返回，但并非所有系统都提供高于 1 秒的精度。虽然此函数通常返回非递减值，但如果在两次调用之间设置了系统时钟，则它可以返回比先前调用更低的值。

返回的数字*time()* 可以通过将其传递给*gmtime()* 函数或转换为 UTC 中更常见的时间格式（即年、月、日、小时等）或通过将它传递给*localtime()* 函数获得本地时间。在这两种情况下都返回一个*struct_time* 对象，日历日期组件可以从中作为属性访问。

time.**thread_time**() → float
（以小数表示的秒为单位）返回当前线程的系统和用户 CPU 时间的总计值。它不包括睡眠状态所消耗的时间。根据定义它只作用于线程范围。返回值的参考点未被定义，因此只有两次调用之间的差值才是有效的。

可用性： Windows、Linux、Unix 系统支持 `CLOCK_THREAD_CPUTIME_ID`。

3.7 版新加入.

time.**thread_time_ns**() → int
与*thread_time()* 相似，但返回纳秒时间。

3.7 版新加入.

time.**time_ns**() → int
与*time()* 相似，但返回时间为用整数表示的自*epoch* 以来所经过的纳秒数。

3.7 版新加入.

time.**tzset**()
重置库例程使用的时间转换规则。环境变量 `TZ` 指定如何完成。它还将设置变量 `tzname`（来自 `TZ` 环境变量），`timezone`（UTC 的西部非 DST 秒），`altzone`（UTC 以西的 DST 秒）和 `daylight`（如果此时区没有任何夏令时规则则为 0，如果有夏令时适用的时间，无论过去、现在或未来，则为非零）。

Availability: Unix.

備F: 虽然在很多情况下，更改 `TZ` 环境变量而不调用*tzset()* 可能会影响函数的输出，例如*localtime()*，不应该依赖此行为。

`TZ` 不应该包含空格。

`TZ` 环境变量的标准格式是（为了清晰起见，添加了空格）：

```
std offset [dst [offset [,start[/time], end[/time]]]]
```

组件的位置是：

std 和 dst 三个或更多字母数字，给出时区缩写。这些将传到 time.tzname

offset 偏移量的形式为：`± hh[:mm[:ss]]`。这表示添加到达 UTC 的本地时间的值。如果前面有'-'，则时区位于本初子午线的东边；否则，在它是西边。如果 dst 之后没有偏移，则假设夏令时比标准时间提前一小时。

start[/time], end[/time] 指示何时更改为 DST 和从 DST 返回。开始日期和结束日期的格式为以下之一：

Jn Julian 日 *n*（1 <= *n* <= 365）。闰日不计算在内，因此在所有年份中，2 月 28 日是第 59 天，3 月 1 日是第 60 天。

n 从零开始的 Julian 日（0 <= *n* <= 365）。闰日计入，可以引用 2 月 29 日。

Mm.n.d 一年中 *m* 月的第 *n* 周（1 <= *n* <= 5，1 <= *m* <= 12，第 5 周表示“可能在 *m* 月第 4 周或第 5 周出现的最后第 *d* 日”）的第 *d* 天（0 <= *d* <= 6）。第 1 周是第 *d* 天发生的第一周。第 0 天是星期天。

`time` 的格式与 `offset` 的格式相同，但不允许使用前导符号（'-' 或'+'）。如果没有给出时间，则默认值为 02:00:00。

```
>>> os.environ['TZ'] = 'EST+05EDT,M4.1.0,M10.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'02:07:36 05/08/03 EDT'
>>> os.environ['TZ'] = 'AEST-10AEDT-11,M10.5.0,M3.5.0'
>>> time.tzset()
>>> time.strftime('%X %x %Z')
'16:08:12 05/08/03 AEST'
```

在许多 Unix 系统（包括 *BSD，Linux，Solaris 和 Darwin 上），使用系统的区域信息（*tzfile(5)*）数据库来指定时区规则会更方便。为此，将 `TZ` 环境变量设置为所需时区数据文件的路径，相对于系统'zoneinfo' 时区数据库的根目录，通常位于 `/usr/share/zoneinfo`。例如，`'US/Eastern'`、`'Australia/Melbourne'`、`'Egypt'` 或 `'Europe/Amsterdam'`。

```
>>> os.environ['TZ'] = 'US/Eastern'
>>> time.tzset()
>>> time.tzname
('EST', 'EDT')
>>> os.environ['TZ'] = 'Egypt'
>>> time.tzset()
>>> time.tzname
('EET', 'EEST')
```

16.3.2 Clock ID 常量

这些常量用作*clock_getres()* 和*clock_gettime()* 的参数。

`time.`**CLOCK_BOOTTIME**
与*CLOCK_MONOTONIC* 相同，除了它还包括系统暂停的任何时间。

这允许应用程序获得一个暂停感知的单调时钟，而不必处理*CLOCK_REALTIME* 的复杂性，如果使用 `settimeofday()` 或类似的时间更改时间可能会有不连续性。

可用性: Linux 2.6.39 或更新

3.7 版新加入.

`time.`**CLOCK_HIGHRES**
Solaris OS 有一个 `CLOCK_HIGHRES` 计时器，试图使用最佳硬件源，并可能提供接近纳秒的分辨率。`CLOCK_HIGHRES` 是不可调节的高分辨率时钟。

可用性: Solaris.

3.3 版新加入.

`time.`**CLOCK_MONOTONIC**
无法设置的时钟，表示自某些未指定的起点以来的单调时间。

Availability: Unix.

3.3 版新加入.

time.**CLOCK_MONOTONIC_RAW**
类似于*CLOCK_MONOTONIC*，但可以访问不受 NTP 调整影响的原始硬件时间。

可用性: Linux 2.6.28 和更新版本，macOS 10.12 和更新版本。

3.3 版新加入.

time.**CLOCK_PROCESS_CPUTIME_ID**
来自 CPU 的高分辨率每进程计时器。

Availability: Unix.

3.3 版新加入.

time.**CLOCK_PROF**
来自 CPU 的高分辨率每进程计时器。

可用性: FreeBSD，NetBSD 7 或更新，OpenBSD.

3.7 版新加入.

time.**CLOCK_TAI**
国际原子时间

该系统必须有一个当前闰秒表以便能给出正确的回答。PTP 或 NTP 软件可以用来维护闰秒表。

可用性: Linux。

3.9 版新加入.

time.**CLOCK_THREAD_CPUTIME_ID**
特定于线程的 CPU 时钟。

Availability: Unix.

3.3 版新加入.

time.**CLOCK_UPTIME**
该时间的绝对值是系统运行且未暂停的时间，提供准确的正常运行时间测量，包括绝对值和间隔值。

可用性: FreeBSD，OpenBSD 5.5 或更新。

3.7 版新加入.

time.**CLOCK_UPTIME_RAW**
单调递增的时钟，记录从一个任意起点开始的时间，不受频率或时间调整的影响，并且当系统休眠时将不会递增。

可用性: macOS 10.12 和更新版本。

3.8 版新加入.

以下常量是唯一可以发送到*clock_settime()*的参数。

time.**CLOCK_REALTIME**
系统范围的实时时钟。设置此时钟需要适当的权限。

Availability: Unix.

3.3 版新加入.

16.3.3 时区常量

time.**altzone**
本地 DST 时区的偏移量，以 UTC 为单位的秒数，如果已定义。如果当地 DST 时区在 UTC 以东（如在西欧，包括英国），则是负数。只有当 daylight 非零时才使用它。见下面的注释。

time.**daylight**
如果定义了 DST 时区，则为非零。见下面的注释。

time.**timezone**
本地（非 DST）时区的偏移量，UTC 以西的秒数（西欧大部分地区为负，美国为正，英国为零）。见下面的注释。

time.**tzname**
两个字符串的元组：第一个是本地非 DST 时区的名称，第二个是本地 DST 时区的名称。如果未定义 DST 时区，则不应使用第二个字符串。见下面的注释。

備🄵： 对于上述时区常量（*altzone*、*daylight*、*timezone* 和 *tzname*），该值由模块加载时有效的时区规则确定，或者最后一次 *tzset()* 被调用时，并且在过去的时间可能不正确。建议使用来自 *localtime()* 结果的 tm_gmtoff 和 tm_zone 来获取时区信息。

也参考：

模块 *datetime* 更多面向对象的日期和时间接口。

模块 *locale* 国际化服务。区域设置会影响 *strftime()* 和 *strptime()* 中许多格式说明符的解析。

模块 *calendar* 一般日历相关功能。这个模块的 timegm() 是函数 *gmtime()* 的反函数。

🄵解

16.4 argparse --- 命令行选项、参数和子命令解析器

3.2 版新加入.

源代码： Lib/argparse.py

教程

此页面包含该 API 的参考信息。有关 Python 命令行解析更细致的介绍，请参阅 argparse 教程。

argparse 模块可以让人轻松编写用户友好的命令行接口。程序定义它需要的参数，然后 *argparse* 将弄清如何从 *sys.argv* 解析出那些参数。*argparse* 模块还会自动生成帮助和使用手册，并在用户给程序传入无效参数时报出错误信息。

16.4.1 示例

以下代码是一个 Python 程序，它获取一个整数列表并计算总和或者最大值：

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

假设上面的 Python 代码保存在名为 `prog.py` 的文件中，它可以在命令行运行并提供有用的帮助消息：

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

当使用适当的参数运行时，它会输出命令行传入整数的总和或者最大值：

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

如果传入无效参数，则会报出错误：

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

以下部分将引导你完成这个示例。

创建一个解析器

使用*argparse* 的第一步是创建一个*ArgumentParser* 对象：

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

ArgumentParser 对象包含将命令行解析成 Python 数据类型所需的全部信息。

添加参数

给一个*ArgumentParser* 添加程序参数信息是通过调用*add_argument()* 方法完成的。通常，这些调用指定*ArgumentParser* 如何获取命令行字符串并将其转换为对象。这些信息在*parse_args()* 调用时被存储和使用。例如：

```
>>> parser.add_argument('integers', metavar='N', type=int, nargs='+',
...                     help='an integer for the accumulator')
>>> parser.add_argument('--sum', dest='accumulate', action='store_const',
...                     const=sum, default=max,
...                     help='sum the integers (default: find the max)')
```

稍后，调用*parse_args()* 将返回一个具有 integers 和 accumulate 两个属性的对象。integers 属性将是一个包含一个或多个整数的列表，而 accumulate 属性当命令行中指定了 --sum 参数时将是*sum()* 函数，否则则是*max()* 函数。

解析参数

ArgumentParser 通过*parse_args()* 方法解析参数。它将检查命令行，把每个参数转换为适当的类型然后调用相应的操作。在大多数情况下，这意味着一个简单的*Namespace* 对象将从命令行参数中解析出的属性构建：

```
>>> parser.parse_args(['--sum', '7', '-1', '42'])
Namespace(accumulate=<built-in function sum>, integers=[7, -1, 42])
```

在脚本中，通常*parse_args()* 会被不带参数调用，而*ArgumentParser* 将自动从*sys.argv* 中确定命令行参数。

16.4.2 ArgumentParser 对象

class argparse.**ArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *parents=[]*, *formatter_class=argparse.HelpFormatter*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*, *allow_abbrev=True*, *exit_on_error=True*)

创建一个新的*ArgumentParser* 对象。所有的参数都应当作为关键字参数传入。每个参数在下面都有它更详细的描述，但简而言之，它们是：

- *prog* - 程序的名称（默认：sys.argv[0]）
- *usage* - 描述程序用途的字符串（默认值：从添加到解析器的参数生成）
- *description* - 在参数帮助文档之前显示的文本（默认值：无）
- *epilog* - 在参数帮助文档之后显示的文本（默认值：无）
- *parents* - 一个*ArgumentParser* 对象的列表，它们的参数也应包含在内
- *formatter_class* - 用于自定义帮助文档输出格式的类
- *prefix_chars* - 可选参数的前缀字符集合（默认值：'-'）
- *fromfile_prefix_chars* - 当需要从文件中读取其他参数时，用于标识文件名的前缀字符集合（默认值：None）
- *argument_default* - 参数的全局默认值（默认值：None）
- *conflict_handler* - 解决冲突选项的策略（通常是不必要的）

- *add_help* - 为解析器添加一个 `-h/--help` 选项（默认值：`True`）
- *allow_abbrev* - 如果缩写是无歧义的，则允许缩写长选项（默认值：`True`）
- *exit_on_error* - 决定当错误发生时是否让 ArgumentParser 附带错误信息退出。(默认值: `True`)

3.5 版更變: 添加 *allow_abbrev* 参数。

3.8 版更變: 在之前的版本中，*allow_abbrev* 还会禁用短旗标分组，例如 `-vv` 表示为 `-v -v`。

3.9 版更變: 添加了 *exit_on_error* 形参。

以下部分描述这些参数如何使用。

prog

默认情况下，*ArgumentParser* 对象使用 `sys.argv[0]` 来确定如何在帮助消息中显示程序名称。这一默认值几乎总是可取的，因为它将使帮助消息与从命令行调用此程序的方式相匹配。例如，对于有如下代码的名为 `myprogram.py` 的文件：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

该程序的帮助信息将显示 `myprogram.py` 作为程序名称（无论程序从何处被调用）：

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
$ cd ..
$ python subdir/myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

要更改这样的默认行为，可以使用 `prog=` 参数为*ArgumentParser* 提供另一个值：

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.print_help()
usage: myprogram [-h]

optional arguments:
 -h, --help  show this help message and exit
```

需要注意的是，无论是从 `sys.argv[0]` 或是从 `prog=` 参数确定的程序名称，都可以在帮助消息里通过 `%(prog)s` 格式串来引用。

```
>>> parser = argparse.ArgumentParser(prog='myprogram')
>>> parser.add_argument('--foo', help='foo of the %(prog)s program')
>>> parser.print_help()
usage: myprogram [-h] [--foo FOO]

```

(下页继续)

(繼續上一頁)

```
optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo of the myprogram program
```

usage

默认情况下，*ArgumentParser* 根据它包含的参数来构建用法消息：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

可以通过 `usage=` 关键字参数覆盖这一默认消息：

```
>>> parser = argparse.ArgumentParser(prog='PROG', usage='%(prog)s [options]')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [options]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

在用法消息中可以使用 `%(prog)s` 格式说明符来填入程序名称。

description

大多数对*ArgumentParser* 构造方法的调用都会使用 `description=` 关键字参数。这个参数简要描述这个程序做什么以及怎么做。在帮助消息中，这个描述会显示在命令行用法字符串和各种参数的帮助消息之间:

```
>>> parser = argparse.ArgumentParser(description='A foo that bars')
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit
```

在默认情况下，description 将被换行以便适应给定的空间。如果想改变这种行为，见*formatter_class* 参数。

epilog

一些程序喜欢在 description 参数后显示额外的对程序的描述。这种文字能够通过给*ArgumentParser*:: 提供 epilog= 参数而被指定。

```
>>> parser = argparse.ArgumentParser(
...     description='A foo that bars',
...     epilog="And that's how you'd foo a bar")
>>> parser.print_help()
usage: argparse.py [-h]

A foo that bars

optional arguments:
 -h, --help  show this help message and exit

And that's how you'd foo a bar
```

和*description* 参数一样，epilog= text 在默认情况下会换行，但是这种行为能够被调整通过提供*formatter_class* 参数给 ArgumentParse.

parents

有些时候，少数解析器会使用同一系列参数。单个解析器能够通过提供 parents= 参数给*ArgumentParser* 而使用相同的参数而不是重复这些参数的定义。parents= 参数使用*ArgumentParser* 对象的列表，从它们那里收集所有的位置和可选的行为，然后将这写行为加到正在构建的*ArgumentParser* 对象。

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--parent', type=int)

>>> foo_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> foo_parser.add_argument('foo')
>>> foo_parser.parse_args(['--parent', '2', 'XXX'])
Namespace(foo='XXX', parent=2)

>>> bar_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> bar_parser.add_argument('--bar')
>>> bar_parser.parse_args(['--bar', 'YYY'])
Namespace(bar='YYY', parent=None)
```

请注意大多数父解析器会指定 add_help=False . 否则，ArgumentParse 将会看到两个 -h/--help 选项（一个在父参数中一个在子参数中）并且产生一个错误。

備F: 你在通过 “parents=“ 传递解析器之前必须完全初始化它们。如果你在子解析器之后改变父解析器，这些改变将不会反映在子解析器上。

---

formatter_class

ArgumentParser 对象允许通过指定备用格式化类来自定义帮助格式。目前，有四种这样的类。

class argparse.**RawDescriptionHelpFormatter**
class argparse.**RawTextHelpFormatter**
class argparse.**ArgumentDefaultsHelpFormatter**
class argparse.**MetavarTypeHelpFormatter**

RawDescriptionHelpFormatter 和*RawTextHelpFormatter* 在正文的描述和展示上给与了更多的控制。*ArgumentParser* 对象会将*description* 和*epilog* 的文字在命令行中自动换行。

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     description='''this description
...         was indented weird
...             but that is okay''',
...     epilog='''
...             likewise for this epilog whose whitespace will
...         be cleaned up and whose words will be wrapped
...         across a couple lines''')
>>> parser.print_help()
usage: PROG [-h]

this description was indented weird but that is okay

optional arguments:
 -h, --help  show this help message and exit

likewise for this epilog whose whitespace will be cleaned up and whose words
will be wrapped across a couple lines
```

传*RawDescriptionHelpFormatter* 给 formatter_class= 表示*description* 和*epilog* 已经被正确的格式化了，不能在命令行中被自动换行:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.RawDescriptionHelpFormatter,
...     description=textwrap.dedent('''\
...         Please do not mess up this text!
...         --------------------------------
...             I have indented it
...             exactly the way
...             I want it
...         '''))
>>> parser.print_help()
usage: PROG [-h]

Please do not mess up this text!
--------------------------------
   I have indented it
   exactly the way
   I want it

optional arguments:
 -h, --help  show this help message and exit
```

RawTextHelpFormatter 保留所有种类文字的空格，包括参数的描述。然而，多重的新行会被替换成一行。如果你想保留多重的空白行，可以在新行之间加空格。

ArgumentDefaultsHelpFormatter 自动添加默认的值的信息到每一个帮助信息的参数中:

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.ArgumentDefaultsHelpFormatter)
>>> parser.add_argument('--foo', type=int, default=42, help='FOO!')
>>> parser.add_argument('bar', nargs='*', default=[1, 2, 3], help='BAR!')
>>> parser.print_help()
usage: PROG [-h] [--foo FOO] [bar ...]

positional arguments:
 bar         BAR! (default: [1, 2, 3])

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   FOO! (default: 42)
```

MetavarTypeHelpFormatter 为它的值在每一个参数中使用*type* 的参数名当作它的显示名（而不是使用通常的格式*dest*):

```
>>> parser = argparse.ArgumentParser(
...     prog='PROG',
...     formatter_class=argparse.MetavarTypeHelpFormatter)
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', type=float)
>>> parser.print_help()
usage: PROG [-h] [--foo int] float

positional arguments:
  float

optional arguments:
  -h, --help  show this help message and exit
  --foo int
```

prefix_chars

许多命令行会使用 - 当作前缀，比如 -f/--foo。如果解析器需要支持不同的或者额外的字符，比如像 +f 或者 /foo 的选项，可以在参数解析构建器中使用 prefix_chars= 参数。

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='-+')
>>> parser.add_argument('+f')
>>> parser.add_argument('++bar')
>>> parser.parse_args('+f X ++bar Y'.split())
Namespace(bar='Y', f='X')
```

prefix_chars= 参数默认使用 '-'。提供一组不包括 - 的字符将导致 -f/--foo 选项不被允许。

fromfile_prefix_chars

Sometimes, for example when dealing with a particularly long argument list, it may make sense to keep the list of arguments in a file rather than typing it out at the command line. If the `fromfile_prefix_chars=` argument is given to the *ArgumentParser* constructor, then arguments that start with any of the specified characters will be treated as files, and will be replaced by the arguments they contain. For example:

```
>>> with open('args.txt', 'w') as fp:
...     fp.write('-f\nbar')
>>> parser = argparse.ArgumentParser(fromfile_prefix_chars='@')
>>> parser.add_argument('-f')
>>> parser.parse_args(['-f', 'foo', '@args.txt'])
Namespace(f='bar')
```

从文件读取的参数在默认情况下必须一个一行（但是可参见*convert_arg_line_to_args()*）并且它们被视为与命令行上的原始文件引用参数位于同一位置。所以在以上例子中，`['-f', 'foo', '@args.txt']` 的表示和 `['-f', 'foo', '-f', 'bar']` 的表示相同。

`fromfile_prefix_chars=` 参数默认为 `None`，意味着参数不会被当作文件对待。

argument_default

一般情况下，参数默认会通过设置一个默认到*add_argument()* 或者调用带一组指定键值对的*ArgumentParser.set_defaults()* 方法。但是有些时候，为参数指定一个普遍适用的解析器会更有用。这能够通过传输 `argument_default=` 关键词参数给*ArgumentParser* 来完成。举个栗子，要全局禁止在*parse_args()* 中创建属性，我们提供 `argument_default=SUPPRESS`:

```
>>> parser = argparse.ArgumentParser(argument_default=argparse.SUPPRESS)
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar', nargs='?')
>>> parser.parse_args(['--foo', '1', 'BAR'])
Namespace(bar='BAR', foo='1')
>>> parser.parse_args([])
Namespace()
```

allow_abbrev

正常情况下，当你向*ArgumentParser* 的*parse_args()* 方法传入一个参数列表时，它会*recognizes abbreviations*。

这个特性可以设置 `allow_abbrev` 为 `False` 来关闭:

```
>>> parser = argparse.ArgumentParser(prog='PROG', allow_abbrev=False)
>>> parser.add_argument('--foobar', action='store_true')
>>> parser.add_argument('--foonley', action='store_false')
>>> parser.parse_args(['--foon'])
usage: PROG [-h] [--foobar] [--foonley]
PROG: error: unrecognized arguments: --foon
```

3.5 版新加入.

conflict_handler

ArgumentParser 对象不允许在相同选项字符串下有两种行为。默认情况下，*ArgumentParser* 对象会产生一个异常如果去创建一个正在使用的选项字符串参数。

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
Traceback (most recent call last):
 ..
ArgumentError: argument --foo: conflicting option string(s): --foo
```

有些时候（例如：使用*parents*），重写旧的有相同选项字符串的参数会更有用。为了产生这种行为，'resolve' 值可以提供给*ArgumentParser* 的 conflict_handler= 参数：

```
>>> parser = argparse.ArgumentParser(prog='PROG', conflict_handler='resolve')
>>> parser.add_argument('-f', '--foo', help='old foo help')
>>> parser.add_argument('--foo', help='new foo help')
>>> parser.print_help()
usage: PROG [-h] [-f FOO] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 -f FOO      old foo help
 --foo FOO   new foo help
```

注意*ArgumentParser* 对象只能移除一个行为如果它所有的选项字符串都被重写。所以，在上面的例子中，旧的 -f/--foo 行为回合 -f 行为保持一样，因为只有 --foo 选项字符串被重写。

add_help

默认情况下，ArgumentParser 对象添加一个简单的显示解析器帮助信息的选项。举个栗子，考虑一个名为 myprogram.py 的文件包含如下代码：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--foo', help='foo help')
args = parser.parse_args()
```

如果 -h or --help 在命令行中被提供，参数解析器帮助信息会打印：

```
$ python myprogram.py --help
usage: myprogram.py [-h] [--foo FOO]

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO   foo help
```

有时候可能会需要关闭额外的帮助信息。这可以通过在*ArgumentParser* 中设置 add_help= 参数为 False 来实现。

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> parser.add_argument('--foo', help='foo help')
>>> parser.print_help()
usage: PROG [--foo FOO]
```

(下页继续)

(繼續上一頁)

```
optional arguments:
 --foo FOO  foo help
```

帮助选项一般为 -h/--help。如果 prefix_chars= 被指定并且没有包含 - 字符，在这种情况下，-h --help 不是有效的选项。此时，prefix_chars 的第一个字符将用作帮助选项的前缀。

```
>>> parser = argparse.ArgumentParser(prog='PROG', prefix_chars='+/')
>>> parser.print_help()
usage: PROG [+h]

optional arguments:
  +h, ++help  show this help message and exit
```

exit_on_error

正常情况下，当你向*ArgumentParser* 的*parse_args()* 方法传入一个无效的参数列表时，它将会退出并发出错误信息。

如果用户想要手动捕获错误，可通过将 exit_on_error 设为 False 来启用该特性:

```
>>> parser = argparse.ArgumentParser(exit_on_error=False)
>>> parser.add_argument('--integers', type=int)
_StoreAction(option_strings=['--integers'], dest='integers', nargs=None, const=None,␣
↪default=None, type=<class 'int'>, choices=None, help=None, metavar=None)
>>> try:
...     parser.parse_args('--integers a'.split())
... except argparse.ArgumentError:
...     print('Catching an argumentError')
...
Catching an argumentError
```

3.9 版新加入.

16.4.3 add_argument() 方法

ArgumentParser.**add_argument**(*name or flags...*[, *action*][, *nargs*][, *const*][, *default*][, *type*][, *choices*][, *required*][, *help*][, *metavar*][, *dest*])

定义单个的命令行参数应当如何解析。每个形参都在下面有它自己更多的描述，长话短说有:

- *name or flags* - 一个命名或者一个选项字符串的列表，例如 foo 或 -f, --foo。
- *action* - 当参数在命令行中出现时使用的动作基本类型。
- *nargs* - 命令行参数应当消耗的数目。
- *const* - 被一些*action* 和*nargs* 选择所需求的常数。
- *default* - 当参数未在命令行中出现并且也不存在于命名空间对象时所产生的值。
- *type* - 命令行参数应当被转换成的类型。
- *choices* - 可用的参数的容器。
- *required* - 此命令行选项是否可省略（仅选项可用）。
- *help* - 一个此选项作用的简单描述。
- *metavar* - 在使用方法消息中使用的参数值示例。

- *dest* - 被添加到*parse_args()* 所返回对象上的属性名。

以下部分描述这些参数如何使用。

name or flags

add_argument() 方法必须知道它是否是一个选项，例如 `-f` 或 `--foo`，或是一个位置参数，例如一组文件名。第一个传递给*add_argument()* 的参数必须是一系列 flags 或者是一个简单的参数名。例如，可以选项可以被这样创建:

```
>>> parser.add_argument('-f', '--foo')
```

而位置参数可以这么创建:

```
>>> parser.add_argument('bar')
```

当*parse_args()* 被调用，选项会以 `-` 前缀识别，剩下的参数则会被假定为位置参数:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-f', '--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args(['BAR'])
Namespace(bar='BAR', foo=None)
>>> parser.parse_args(['BAR', '--foo', 'FOO'])
Namespace(bar='BAR', foo='FOO')
>>> parser.parse_args(['--foo', 'FOO'])
usage: PROG [-h] [-f FOO] bar
PROG: error: the following arguments are required: bar
```

action

ArgumentParser 对象将命令行参数与动作相关联。这些动作可以做与它们相关联的命令行参数的任何事，尽管大多数动作只是简单的向*parse_args()* 返回的对象上添加属性。`action` 命名参数指定了这个命令行参数应当如何处理。供应的动作有:

- `'store'` - 存储参数的值。这是默认的动作。例如:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo')
  >>> parser.parse_args('--foo 1'.split())
  Namespace(foo='1')
  ```

- `'store_const'` - 存储被*const* 命名参数指定的值。`'store_const'` 动作通常用在选项中来指定一些标志。例如:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', action='store_const', const=42)
  >>> parser.parse_args(['--foo'])
  Namespace(foo=42)
  ```

- `'store_true'` and `'store_false'` - 这些是 `'store_const'` 分别用作存储 `True` 和 `False` 值的特殊用例。另外，它们的默认值分别为 `False` 和 `True`。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('--bar', action='store_false')
>>> parser.add_argument('--baz', action='store_false')
>>> parser.parse_args('--foo --bar'.split())
Namespace(foo=True, bar=False, baz=True)
```

- `'append'` - 存储一个列表，并且将每个参数值追加到列表中。在允许多次使用选项时很有用。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='append')
>>> parser.parse_args('--foo 1 --foo 2'.split())
Namespace(foo=['1', '2'])
```

- `'append_const'` - 这存储一个列表，并将*const* 命名参数指定的值追加到列表中。(注意*const* 命名参数默认为 `None`。) "append_const" 动作一般在多个参数需要在同一列表中存储常数时会有用。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--str', dest='types', action='append_const', const=str)
>>> parser.add_argument('--int', dest='types', action='append_const', const=int)
>>> parser.parse_args('--str --int'.split())
Namespace(types=[<class 'str'>, <class 'int'>])
```

- `'count'` - 计算一个关键字参数出现的数目或次数。例如，对于一个增长的详情等级来说有用:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--verbose', '-v', action='count', default=0)
>>> parser.parse_args(['-vvv'])
Namespace(verbose=3)
```

  请注意，*default* 将为 `None`，除非显式地设为 *0*。

- `'help'` - 打印所有当前解析器中的选项和参数的完整帮助信息，然后退出。默认情况下，一个 help 动作会被自动加入解析器。关于输出是如何创建的，参与*ArgumentParser*。

- `'version'` - 期望有一个 `version=` 命名参数在*add_argument()* 调用中，并打印版本信息并在调用后退出:

```
>>> import argparse
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--version', action='version', version='%(prog)s 2.0')
>>> parser.parse_args(['--version'])
PROG 2.0
```

- `'extend'` - 这会存储一个列表，并将每个参数值加入到列表中。示例用法:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument("--foo", action="extend", nargs="+", type=str)
>>> parser.parse_args(["--foo", "f1", "--foo", "f2", "f3", "f4"])
Namespace(foo=['f1', 'f2', 'f3', 'f4'])
```

  3.8 版新加入.

你还可以通过传递一个 Action 子类或实现相同接口的其他对象来指定任意操作。`BooleanOptionalAction` 在 `argparse` 中可用并会添加对布尔型操作例如 `--foo` 和 `--no-foo` 的支持:

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=argparse.BooleanOptionalAction)
>>> parser.parse_args(['--no-foo'])
Namespace(foo=False)
```

3.9 版新加入.

创建自定义动作的推荐方式是扩展*Action*，重载 `__call__` 方法以及可选的 `__init__` 和 `format_usage` 方法。

一个自定义动作的例子:

```
>>> class FooAction(argparse.Action):
...     def __init__(self, option_strings, dest, nargs=None, **kwargs):
...         if nargs is not None:
...             raise ValueError("nargs not allowed")
...         super().__init__(option_strings, dest, **kwargs)
...     def __call__(self, parser, namespace, values, option_string=None):
...         print('%r %r %r' % (namespace, values, option_string))
...         setattr(namespace, self.dest, values)
...
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action=FooAction)
>>> parser.add_argument('bar', action=FooAction)
>>> args = parser.parse_args('1 --foo 2'.split())
Namespace(bar=None, foo=None) '1' None
Namespace(bar='1', foo=None) '2' '--foo'
>>> args
Namespace(bar='1', foo='2')
```

更多描述，见*Action*。

nargs

ArgumentParser 对象通常关联一个单独的命令行参数到一个单独的被执行的动作。`nargs` 命名参数关联不同数目的命令行参数到单一动作。支持的值有:

- `N` （一个整数）。命令行中的 `N` 个参数会被聚集到一个列表中。例如:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs=2)
  >>> parser.add_argument('bar', nargs=1)
  >>> parser.parse_args('c --foo a b'.split())
  Namespace(bar=['c'], foo=['a', 'b'])
  ```

  注意 `nargs=1` 会产生一个单元素列表。这和默认的元素本身是不同的。

- `'?'`。如果可能的话，会从命令行中消耗一个参数，并产生一个单一项。如果当前没有命令行参数，则会产生*default* 值。注意，对于选项，有另外的用例 - 选项字符串出现但没有跟随命令行参数，则会产生*const* 值。一些说用用例:

  ```
  >>> parser = argparse.ArgumentParser()
  >>> parser.add_argument('--foo', nargs='?', const='c', default='d')
  >>> parser.add_argument('bar', nargs='?', default='d')
  >>> parser.parse_args(['XX', '--foo', 'YY'])
  Namespace(bar='XX', foo='YY')
  ```

(下页继续)

(繼續上一頁)

```
>>> parser.parse_args(['XX', '--foo'])
Namespace(bar='XX', foo='c')
>>> parser.parse_args([])
Namespace(bar='d', foo='d')
```

nargs='?' 的一个更普遍用法是允许可选的输入或输出文件:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', nargs='?', type=argparse.FileType('r'),
...                     default=sys.stdin)
>>> parser.add_argument('outfile', nargs='?', type=argparse.FileType('w'),
...                     default=sys.stdout)
>>> parser.parse_args(['input.txt', 'output.txt'])
Namespace(infile=<_io.TextIOWrapper name='input.txt' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='output.txt' encoding='UTF-8'>)
>>> parser.parse_args([])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>,
          outfile=<_io.TextIOWrapper name='<stdout>' encoding='UTF-8'>)
```

- '*'。所有当前命令行参数被聚集到一个列表中。注意通过 nargs='*' 来实现多个位置参数通常没有意义，但是多个选项是可能的。例如:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', nargs='*')
>>> parser.add_argument('--bar', nargs='*')
>>> parser.add_argument('baz', nargs='*')
>>> parser.parse_args('a b --foo x y --bar 1 2'.split())
Namespace(bar=['1', '2'], baz=['a', 'b'], foo=['x', 'y'])
```

- '+'。和 '*' 类似，所有当前命令行参数被聚集到一个列表中。另外，当前没有至少一个命令行参数时会产生一个错误信息。例如:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('foo', nargs='+')
>>> parser.parse_args(['a', 'b'])
Namespace(foo=['a', 'b'])
>>> parser.parse_args([])
usage: PROG [-h] foo [foo ...]
PROG: error: the following arguments are required: foo
```

如果不提供 nargs 命名参数，则消耗参数的数目将被*action* 决定。通常这意味着单一项目（非列表）消耗单一命令行参数。

const

add_argument() 的 “const“ 参数用于保存不从命令行中读取但被各种*ArgumentParser* 动作需求的常数值。最常用的两例为:

- 当*add_argument()* 通过 action='store_const' 或 action='append_const 调用时。这些动作将 const 值添加到*parse_args()* 返回的对象的属性中。在*action* 的描述中查看案例。
- 当*add_argument()* 通过选项（例如 -f 或 --foo）调用并且 nargs='?' 时。这会创建一个可以跟随零个或一个命令行参数的选项。当解析命令行时，如果选项后没有参数，则将用 const 代替。在*nargs* 描述中查看案例。

对 'store_const' 和 'append_const' 动作，const 命名参数必须给出。对其他动作，默认为 None。

default

所有选项和一些位置参数可能在命令行中被忽略。*add_argument()* 的命名参数 default，默认值为 None，指定了在命令行参数未出现时应当使用的值。对于选项，default 值在选项未在命令行中出现时使用:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args(['--foo', '2'])
Namespace(foo='2')
>>> parser.parse_args([])
Namespace(foo=42)
```

如果目标命名空间已经有一个属性集，则 *default* 动作不会覆盖它:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=42)
>>> parser.parse_args([], namespace=argparse.Namespace(foo=101))
Namespace(foo=101)
```

如果 default 值是一个字符串，解析器解析此值就像一个命令行参数。特别是，在将属性设置在*Namespace* 的返回值之前，解析器应用任何提供的*type* 转换参数。否则解析器使用原值:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--length', default='10', type=int)
>>> parser.add_argument('--width', default=10.5, type=int)
>>> parser.parse_args()
Namespace(length=10, width=10.5)
```

对于*nargs* 等于 ? 或 * 的位置参数，default 值在没有命令行参数出现时使用。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', nargs='?', default=42)
>>> parser.parse_args(['a'])
Namespace(foo='a')
>>> parser.parse_args([])
Namespace(foo=42)
```

提供 default=argparse.SUPPRESS 导致命令行参数未出现时没有属性被添加:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default=argparse.SUPPRESS)
>>> parser.parse_args([])
Namespace()
>>> parser.parse_args(['--foo', '1'])
Namespace(foo='1')
```

type

默认情况下，解析器会将命令行参数当作简单字符串读入。然而，命令行字符串经常应当被解读为其他类型，例如*float* 或*int*。*add_argument()* 的 type 关键字允许执行任何必要的类型检查和类型转换。

如果*type* 关键字使用了*default* 关键字，则类型转换器仅会在默认值为字符串时被应用。

传给 type 的参数可以是任何接受单个字符串的可调用对象。如果函数引发了 ArgumentTypeError，*TypeError* 或*ValueError*，异常会被捕获并显示经过良好格式化的错误消息。其他异常类型则不会被处理。

普通内置类型和函数可被用作类型转换器:

```
import argparse
import pathlib

parser = argparse.ArgumentParser()
parser.add_argument('count', type=int)
parser.add_argument('distance', type=float)
parser.add_argument('street', type=ascii)
parser.add_argument('code_point', type=ord)
parser.add_argument('source_file', type=open)
parser.add_argument('dest_file', type=argparse.FileType('w', encoding='latin-1'))
parser.add_argument('datapath', type=pathlib.Path)
```

用户自定义的函数也可以被使用:

```
>>> def hyphenated(string):
...     return '-'.join([word[:4] for word in string.casefold().split()])
...
>>> parser = argparse.ArgumentParser()
>>> _ = parser.add_argument('short_title', type=hyphenated)
>>> parser.parse_args(['"The Tale of Two Cities"'])
Namespace(short_title='"the-tale-of-two-citi')
```

不建议将*bool()* 函数用作类型转换器。它所做的只是将空字符串转为 False 而将非空字符串转为 True。这通常不是用户所想要的。

通常，type 关键字是仅应被用于只会引发上述三种被支持的异常的简单转换的便捷选项。任何具有更复杂错误处理或资源管理的转换都应当在参数被解析后由下游代码来完成。

For example, JSON or YAML conversions have complex error cases that require better reporting than can be given by the type keyword. A *JSONDecodeError* would not be well formatted and a FileNotFound exception would not be handled at all.

即使*FileType* 在用于 type 关键字时也存在限制。如果一个参数使用了 *FileType* 并且有一个后续参数出错，则将报告一个错误但文件并不会被自动关闭。在此情况下，更好的做法是等待直到解析器运行完毕再使用 with 语句来管理文件。

对于简单地检查一组固定值的类型检查器，请考虑改用*choices* 关键字。

choices

某些命令行参数应当从一组受限值中选择。这可通过将一个容器对象作为 *choices* 关键字参数传给*add_argument()* 来处理。当执行命令行解析时，参数值将被检查，如果参数不是可接受的值之一就将显示错误消息:

```
>>> parser = argparse.ArgumentParser(prog='game.py')
>>> parser.add_argument('move', choices=['rock', 'paper', 'scissors'])
>>> parser.parse_args(['rock'])
Namespace(move='rock')
>>> parser.parse_args(['fire'])
usage: game.py [-h] {rock,paper,scissors}
game.py: error: argument move: invalid choice: 'fire' (choose from 'rock',
'paper', 'scissors')
```

请注意 *choices* 容器包含的内容会在执行任意*type* 转换之后被检查，因此 *choices* 容器中对象的类型应当与指定的*type* 相匹配:

```
>>> parser = argparse.ArgumentParser(prog='doors.py')
>>> parser.add_argument('door', type=int, choices=range(1, 4))
>>> print(parser.parse_args(['3']))
Namespace(door=3)
>>> parser.parse_args(['4'])
usage: doors.py [-h] {1,2,3}
doors.py: error: argument door: invalid choice: 4 (choose from 1, 2, 3)
```

任何容器都可作为 *choices* 值传入，因此*list* 对象，*set* 对象以及自定义容器都是受支持的。

不建议使用*enum.Enum*，因为要控制其在用法、帮助和错误消息中的外观是很困难的。

已格式化的选项会覆盖默认的 *metavar*，该值一般是派生自 *dest*。这通常就是你所需要的，因为用户永远不会看到 *dest* 形参。如果不想要这样的显示（或许因为有很多选择），只需指定一个显式的*metavar*。

required

通常，*argparse* 模块会认为 `-f` 和 `--bar` 等旗标是指明 可选的参数，它们总是可以在命令行中被忽略。要让一个选项成为 必需的，则可以将 `True` 作为 `required=` 关键字参数传给*add_argument()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', required=True)
>>> parser.parse_args(['--foo', 'BAR'])
Namespace(foo='BAR')
>>> parser.parse_args([])
usage: [-h] --foo FOO
: error: the following arguments are required: --foo
```

如这个例子所示，如果一个选项被标记为 `required`，则当该选项未在命令行中出现时，*parse_args()* 将会报告一个错误。

備F: 必需的选项通常被认为是不适宜的，因为用户会预期 *options* 都是 可选的，因此在可能的情况下应当避免使用它们。

---

help

help 值是一个包含参数简短描述的字符串。当用户请求帮助时（一般是通过在命令行中使用 -h 或 --help 的方式），这些 help 描述将随每个参数一同显示:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', action='store_true',
...                     help='foo the bars before frobbling')
>>> parser.add_argument('bar', nargs='+',
...                     help='one of the bars to be frobbled')
>>> parser.parse_args(['-h'])
usage: frobble [-h] [--foo] bar [bar ...]

positional arguments:
 bar     one of the bars to be frobbled

optional arguments:
 -h, --help  show this help message and exit
 --foo   foo the bars before frobbling
```

help 字符串可包括各种格式描述符以避免重复使用程序名称或参数*default* 等文本。有效的描述符包括程序名称 %(prog)s 和传给*add_argument()* 的大部分关键字参数，例如 %(default)s, %(type)s 等等:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('bar', nargs='?', type=int, default=42,
...                     help='the bar to %(prog)s (default: %(default)s)')
>>> parser.print_help()
usage: frobble [-h] [bar]

positional arguments:
 bar     the bar to frobble (default: 42)

optional arguments:
 -h, --help  show this help message and exit
```

由于帮助字符串支持%-formatting，如果你希望在帮助字符串中显示 % 字面值，你必须将其转义为 %%。

argparse 支持静默特定选项的帮助，具体做法是将 help 的值设为 argparse.SUPPRESS:

```
>>> parser = argparse.ArgumentParser(prog='frobble')
>>> parser.add_argument('--foo', help=argparse.SUPPRESS)
>>> parser.print_help()
usage: frobble [-h]

optional arguments:
  -h, --help  show this help message and exit
```

metavar

当*ArgumentParser*生成帮助消息时，它需要用某种方式来引用每个预期的参数。默认情况下，ArgumentParser 对象使用*dest*值作为每个对象的"name"。默认情况下，对于位置参数动作，*dest*值将被直接使用，而对于可选参数动作，*dest*值将被转为大写形式。因此，一个位置参数 `dest='bar'` 的引用形式将为 `bar`。一个带有单独命令行参数的可选参数 `--foo` 的引用形式将为 `FOO`。示例如下:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('bar')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo FOO] bar

positional arguments:
 bar

optional arguments:
 -h, --help  show this help message and exit
 --foo FOO
```

可以使用 `metavar` 来指定一个替代名称:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', metavar='YYY')
>>> parser.add_argument('bar', metavar='XXX')
>>> parser.parse_args('X --foo Y'.split())
Namespace(bar='X', foo='Y')
>>> parser.print_help()
usage:  [-h] [--foo YYY] XXX

positional arguments:
 XXX

optional arguments:
 -h, --help  show this help message and exit
 --foo YYY
```

请注意 `metavar` 仅改变 显示的名称 - *parse_args()* 对象的属性名称仍然会由*dest*值确定。

不同的 `nargs` 值可能导致 metavar 被多次使用。提供一个元组给 `metavar` 即为每个参数指定不同的显示信息:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', nargs=2)
>>> parser.add_argument('--foo', nargs=2, metavar=('bar', 'baz'))
>>> parser.print_help()
usage: PROG [-h] [-x X X] [--foo bar baz]

optional arguments:
 -h, --help     show this help message and exit
 -x X X
 --foo bar baz
```

dest

大多数*ArgumentParser* 动作会添加一些值作为*parse_args()* 所返回对象的一个属性。该属性的名称由*add_argument()* 的 dest 关键字参数确定。对于位置参数动作，dest 通常会作为*add_argument()* 的第一个参数提供:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('bar')
>>> parser.parse_args(['XXX'])
Namespace(bar='XXX')
```

对于可选参数动作，dest 的值通常取自选项字符串。*ArgumentParser* 会通过接受第一个长选项字符串并去掉开头的 -- 字符串来生成 dest 的值。如果没有提供长选项字符串，则 dest 将通过接受第一个短选项字符串并去掉开头的 - 字符来获得。任何内部的 - 字符都将被转换为 _ 字符以确保字符串是有效的属性名称。下面的例子显示了这种行为:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('-f', '--foo-bar', '--foo')
>>> parser.add_argument('-x', '-y')
>>> parser.parse_args('-f 1 -x 2'.split())
Namespace(foo_bar='1', x='2')
>>> parser.parse_args('--foo 1 -y 2'.split())
Namespace(foo_bar='1', x='2')
```

dest 允许提供自定义属性名称:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', dest='bar')
>>> parser.parse_args('--foo XXX'.split())
Namespace(bar='XXX')
```

Action 类

Action 类实现了 Action API，它是一个返回可调用对象的可调用对象，返回的可调用对象可处理来自命令行的参数。任何遵循此 API 的对象均可作为 action 形参传给 add_argument()。

class argparse.**Action**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Action 对象会被 ArgumentParser 用来表示解析从命令行中的一个或多个字符串中解析出单个参数所必须的信息。Action 类必须接受两个位置参数以及传给*ArgumentParser.add_argument()* 的任何关键字参数，除了 action 本身。

Action 的实例（或作为 or return value of any callable to the action 形参的任何可调用对象的返回值）应当定义"dest", "option_strings", "default", "type", "required", "help" 等属性。确保这些属性被定义的最容易方式是调用 Action.__init__。

Action 的实例应当为可调用对象，因此所有子类都必须重载 __call__ 方法，该方法应当接受四个形参:

- parser - 包含此动作的 ArgumentParser 对象。
- namespace - 将由*parse_args()* 返回的*Namespace* 对象。大多数动作会使用*setattr()* 为此对象添加属性。
- values - 已关联的命令行参数，并提供相应的类型转换。类型转换由*add_argument()* 的*type* 关键字参数来指定。
- option_string - 被用来发起调用此动作的选项字符串。option_string 参数是可选的，且此参数在动作关联到位置参数时将被略去。

__call__ 方法可以执行任意动作，但通常将基于 dest 和 values 来设置 namespace 的属性。

动作子类可定义 format_usage 方法，该方法不带参数，所返回的字符串将被用于打印程序的用法说明。如果未提供此方法，则将使用适当的默认值。

16.4.4 parse_args() 方法

ArgumentParser.**parse_args**(*args=None*, *namespace=None*)

将参数字符串转换为对象并将其设为命名空间的属性。返回带有成员的命名空间。

之前对*add_argument()* 的调用决定了哪些对象被创建以及它们如何被赋值。请参阅*add_argument()* 的文档了解详情。

- *args* - 要解析的字符串列表。默认值是从*sys.argv* 获取。
- *namespace* - 用于获取属性的对象。默认值是一个新的空*Namespace* 对象。

选项值语法

parse_args() 方法支持多种指定选项值的方式（如果它接受选项的话）。在最简单的情况下，选项和它的值是作为两个单独参数传入的:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('--foo')
>>> parser.parse_args(['-x', 'X'])
Namespace(foo=None, x='X')
>>> parser.parse_args(['--foo', 'FOO'])
Namespace(foo='FOO', x=None)
```

对于长选项（名称长度超过一个字符的选项），选项和值也可以作为单个命令行参数传入，使用 = 分隔它们即可:

```
>>> parser.parse_args(['--foo=FOO'])
Namespace(foo='FOO', x=None)
```

对于短选项（长度只有一个字符的选项），选项和它的值可以拼接在一起:

```
>>> parser.parse_args(['-xX'])
Namespace(foo=None, x='X')
```

有些短选项可以使用单个 - 前缀来进行合并，如果仅有最后一个选项（或没有任何选项）需要值的话:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x', action='store_true')
>>> parser.add_argument('-y', action='store_true')
>>> parser.add_argument('-z')
>>> parser.parse_args(['-xyzZ'])
Namespace(x=True, y=True, z='Z')
```

无效的参数

在解析命令行时，*parse_args()* 会检测多种错误，包括有歧义的选项、无效的类型、无效的选项、错误的位置参数个数等等。当遇到这种错误时，它将退出并打印出错误文本同时附带用法消息:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', type=int)
>>> parser.add_argument('bar', nargs='?')

>>> # invalid type
>>> parser.parse_args(['--foo', 'spam'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: argument --foo: invalid int value: 'spam'

>>> # invalid option
>>> parser.parse_args(['--bar'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: no such option: --bar

>>> # wrong number of arguments
>>> parser.parse_args(['spam', 'badger'])
usage: PROG [-h] [--foo FOO] [bar]
PROG: error: extra arguments found: badger
```

包含 - 的参数

parse_args() 方法会在用户明显出错时尝试给出错误信息，但某些情况本身就存在歧义。例如，命令行参数 `-1` 可能是尝试指定一个选项也可能是尝试提供一个位置参数。*parse_args()* 方法在此会谨慎行事: 位置参数只有在它们看起来像负数并且解析器中没有任何选项看起来像负数时才能以 - 打头。:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-x')
>>> parser.add_argument('foo', nargs='?')

>>> # no negative number options, so -1 is a positional argument
>>> parser.parse_args(['-x', '-1'])
Namespace(foo=None, x='-1')

>>> # no negative number options, so -1 and -5 are positional arguments
>>> parser.parse_args(['-x', '-1', '-5'])
Namespace(foo='-5', x='-1')

>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-1', dest='one')
>>> parser.add_argument('foo', nargs='?')

>>> # negative number options present, so -1 is an option
>>> parser.parse_args(['-1', 'X'])
Namespace(foo=None, one='X')

>>> # negative number options present, so -2 is an option
>>> parser.parse_args(['-2'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: no such option: -2

>>> # negative number options present, so both -1s are options
```

(下页继续)

(繼續上一頁)

```
>>> parser.parse_args(['-1', '-1'])
usage: PROG [-h] [-1 ONE] [foo]
PROG: error: argument -1: expected one argument
```

如果你有必须以 - 打头的位置参数并且看起来不像负数，你可以插入伪参数 '--' 以告诉*parse_args()* 在那之后的内容是一个位置参数:

```
>>> parser.parse_args(['--', '-f'])
Namespace(foo='-f', one=None)
```

参数缩写（前缀匹配）

parse_args() 方法在默认情况下 允许将长选项缩写为前缀，如果缩写无歧义（即前缀与一个特定选项相匹配）的话:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-bacon')
>>> parser.add_argument('-badger')
>>> parser.parse_args('-bac MMM'.split())
Namespace(bacon='MMM', badger=None)
>>> parser.parse_args('-bad WOOD'.split())
Namespace(bacon=None, badger='WOOD')
>>> parser.parse_args('-ba BA'.split())
usage: PROG [-h] [-bacon BACON] [-badger BADGER]
PROG: error: ambiguous option: -ba could match -badger, -bacon
```

可产生一个以上选项的参数会引发错误。此特定可通过将*allow_abbrev* 设为 `False` 来禁用。

在 sys.argv 以外

有时在*sys.argv* 以外用 ArgumentParser 解析参数也是有用的。这可以通过将一个字符串列表传给*parse_args()* 来实现。它适用于在交互提示符下进行检测:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument(
...     'integers', metavar='int', type=int, choices=range(10),
...     nargs='+', help='an integer in the range 0..9')
>>> parser.add_argument(
...     '--sum', dest='accumulate', action='store_const', const=sum,
...     default=max, help='sum the integers (default: find the max)')
>>> parser.parse_args(['1', '2', '3', '4'])
Namespace(accumulate=<built-in function max>, integers=[1, 2, 3, 4])
>>> parser.parse_args(['1', '2', '3', '4', '--sum'])
Namespace(accumulate=<built-in function sum>, integers=[1, 2, 3, 4])
```

命名空间对象

class argparse.**Namespace**

由*parse_args()* 默认使用的简单类，可创建一个存放属性的对象并将其返回。

这个类被有意做得很简单，只是一个具有可读字符串表示形式的*object*。如果你更喜欢类似字典的属性视图，你可以使用标准 Python 中惯常的*vars()*:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> args = parser.parse_args(['--foo', 'BAR'])
>>> vars(args)
{'foo': 'BAR'}
```

另一个用处是让*ArgumentParser* 为一个已存在对象而不是为一个新的*Namespace* 对象的属性赋值。这可以通过指定 namespace= 关键字参数来实现:

```
>>> class C:
...     pass
...
>>> c = C()
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.parse_args(args=['--foo', 'BAR'], namespace=c)
>>> c.foo
'BAR'
```

16.4.5 其它实用工具

子命令

ArgumentParser.**add_subparsers**([*title*][, *description*][, *prog*][, *parser_class*][, *action*][, *option_string*][, *dest*][, *required*][, *help*][, *metavar*])

许多程序都会将其功能拆分为一系列子命令，例如，svn 程序包含的子命令有 svn checkout, svn update 和 svn commit。当一个程序能执行需要多组不同种类命令行参数时这种拆分功能的方式是一个非常好的主意。*ArgumentParser* 通过*add_subparsers()* 方法支持创建这样的子命令。*add_subparsers()* 方法通常不带参数地调用并返回一个特殊的动作对象。这种对象只有一个方法 add_parser()，它接受一个命令名称和任意多个*ArgumentParser* 构造器参数，并返回一个可以通常方式进行修改的*ArgumentParser* 对象。

形参的描述

- title - 输出帮助的子解析器分组的标题；如果提供了描述则默认为”subcommands”，否则使用位置参数的标题
- description - 输出帮助中对子解析器的描述，默认为 None
- prog - 将与子命令帮助一同显示的用法信息，默认为程序名称和子解析器参数之前的任何位置参数。
- parser_class - 将被用于创建子解析器实例的类，默认为当前解析器类（例如 ArgumentParser）
- *action* - 当此参数在命令行中出现时要执行动作的基本类型
- *dest* - 将被用于保存子命令名称的属性名；默认为 None 即不保存任何值
- *required* - 是否必须要提供子命令，默认为 False (在 3.7 中新增)
- *help* - 在输出帮助中的子解析器分组帮助信息，默认为 None

- *metavar* - 帮助信息中表示可用子命令的字符串；默认为 None 并以 {cmd1, cmd2, ..} 的形式表示子命令

一些使用示例:

```
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', action='store_true', help='foo help')
>>> subparsers = parser.add_subparsers(help='sub-command help')
>>>
>>> # create the parser for the "a" command
>>> parser_a = subparsers.add_parser('a', help='a help')
>>> parser_a.add_argument('bar', type=int, help='bar help')
>>>
>>> # create the parser for the "b" command
>>> parser_b = subparsers.add_parser('b', help='b help')
>>> parser_b.add_argument('--baz', choices='XYZ', help='baz help')
>>>
>>> # parse some argument lists
>>> parser.parse_args(['a', '12'])
Namespace(bar=12, foo=False)
>>> parser.parse_args(['--foo', 'b', '--baz', 'Z'])
Namespace(baz='Z', foo=True)
```

请注意*parse_args()* 返回的对象将只包含主解析器和由命令行所选择的子解析器的属性（而没有任何其他子解析器）。因此在上面的例子中，当指定了 a 命令时，将只存在 foo 和 bar 属性，而当指定了 b 命令时，则只存在 foo 和 baz 属性。

类似地，当从一个子解析器请求帮助消息时，只有该特定解析器的帮助消息会被打印出来。帮助消息将不包括父解析器或同级解析器的消息。（每个子解析器命令一条帮助消息，但是，也可以像上面那样通过提供 help= 参数给 add_parser() 来给出。）

```
>>> parser.parse_args(['--help'])
usage: PROG [-h] [--foo] {a,b} ...

positional arguments:
  {a,b}   sub-command help
    a     a help
    b     b help

optional arguments:
  -h, --help  show this help message and exit
  --foo   foo help

>>> parser.parse_args(['a', '--help'])
usage: PROG a [-h] bar

positional arguments:
  bar     bar help

optional arguments:
  -h, --help  show this help message and exit

>>> parser.parse_args(['b', '--help'])
usage: PROG b [-h] [--baz {X,Y,Z}]

optional arguments:
  -h, --help     show this help message and exit
  --baz {X,Y,Z}  baz help
```

add_subparsers() 方法也支持 title 和 description 关键字参数。当两者都存在时，子解析器的命令将出现在输出帮助消息中它们自己的分组内。例如:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(title='subcommands',
...                                    description='valid subcommands',
...                                    help='additional help')
>>> subparsers.add_parser('foo')
>>> subparsers.add_parser('bar')
>>> parser.parse_args(['-h'])
usage:  [-h] {foo,bar} ...

optional arguments:
  -h, --help  show this help message and exit

subcommands:
  valid subcommands

  {foo,bar}   additional help
```

此外，add_parser 还支持附加的 aliases 参数，它允许多个字符串指向同一子解析器。这个例子类似于 svn，将别名 co 设为 checkout 的缩写形式:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>> checkout = subparsers.add_parser('checkout', aliases=['co'])
>>> checkout.add_argument('foo')
>>> parser.parse_args(['co', 'bar'])
Namespace(foo='bar')
```

一个特别有效的处理子命令的方式是将*add_subparsers()* 方法与对*set_defaults()* 的调用结合起来使用，这样每个子解析器就能知道应当执行哪个 Python 函数。例如:

```
>>> # sub-command functions
>>> def foo(args):
...     print(args.x * args.y)
...
>>> def bar(args):
...     print('((%s))' % args.z)
...
>>> # create the top-level parser
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers()
>>>
>>> # create the parser for the "foo" command
>>> parser_foo = subparsers.add_parser('foo')
>>> parser_foo.add_argument('-x', type=int, default=1)
>>> parser_foo.add_argument('y', type=float)
>>> parser_foo.set_defaults(func=foo)
>>>
>>> # create the parser for the "bar" command
>>> parser_bar = subparsers.add_parser('bar')
>>> parser_bar.add_argument('z')
>>> parser_bar.set_defaults(func=bar)
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('foo 1 -x 2'.split())
>>> args.func(args)
```

(下页继续)

(繼續上一頁)

```
2.0
>>>
>>> # parse the args and call whatever function was selected
>>> args = parser.parse_args('bar XYZYX'.split())
>>> args.func(args)
((XYZYX))
```

通过这种方式，你可以在参数解析结束后让*parse_args()* 执行调用适当函数的任务。像这样将函数关联到动作通常是你处理每个子解析器的不同动作的最简便方式。但是，如果有必要检查被发起调用的子解析器的名称，则*add_subparsers()* 调用的 dest 关键字参数将可实现:

```
>>> parser = argparse.ArgumentParser()
>>> subparsers = parser.add_subparsers(dest='subparser_name')
>>> subparser1 = subparsers.add_parser('1')
>>> subparser1.add_argument('-x')
>>> subparser2 = subparsers.add_parser('2')
>>> subparser2.add_argument('y')
>>> parser.parse_args(['2', 'frobble'])
Namespace(subparser_name='2', y='frobble')
```

3.7 版更變: 新增 *required* 关键字参数。

FileType 对象

class argparse.**FileType**(*mode='r'*, *bufsize=-1*, *encoding=None*, *errors=None*)

FileType 工厂类用于创建可作为*ArgumentParser.add_argument()* 的 type 参数传入的对象。以*FileType* 对象作为其类型的参数将使用命令行参数以所请求模式、缓冲区大小、编码格式和错误处理方式打开文件（请参阅*open()* 函数了解详情）:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--raw', type=argparse.FileType('wb', 0))
>>> parser.add_argument('out', type=argparse.FileType('w', encoding='UTF-8'))
>>> parser.parse_args(['--raw', 'raw.dat', 'file.txt'])
Namespace(out=<_io.TextIOWrapper name='file.txt' mode='w' encoding='UTF-8'>, raw=
↪<_io.FileIO name='raw.dat' mode='wb'>)
```

FileType 对象能理解伪参数 '-' 并会自动将其转换为 sys.stdin 用于可读的*FileType* 对象，或是 sys.stdout 用于可写的*FileType* 对象:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('infile', type=argparse.FileType('r'))
>>> parser.parse_args(['-'])
Namespace(infile=<_io.TextIOWrapper name='<stdin>' encoding='UTF-8'>)
```

3.4 版新加入: *encodings* 和 *errors* 关键字参数。

参数组

ArgumentParser.**add_argument_group**(*title=None*, *description=None*)

在默认情况下，*ArgumentParser* 会在显示帮助消息时将命令行参数分为“位置参数”和“可选参数”两组。当存在比默认更好的参数分组概念时，可以使用*add_argument_group()* 方法来创建适当的分组:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group = parser.add_argument_group('group')
>>> group.add_argument('--foo', help='foo help')
>>> group.add_argument('bar', help='bar help')
>>> parser.print_help()
usage: PROG [--foo FOO] bar

group:
  bar    bar help
  --foo FOO  foo help
```

add_argument_group() 方法返回一个具有*add_argument()* 方法的参数分组对象，这与常规的*ArgumentParser* 一样。当一个参数被加入分组时，解析器会将它视为一个正常的参数，但是会在不同的帮助消息分组中显示该参数。*add_argument_group()* 方法接受 *title* 和 *description* 参数，它们可被用来定制显示内容:

```
>>> parser = argparse.ArgumentParser(prog='PROG', add_help=False)
>>> group1 = parser.add_argument_group('group1', 'group1 description')
>>> group1.add_argument('foo', help='foo help')
>>> group2 = parser.add_argument_group('group2', 'group2 description')
>>> group2.add_argument('--bar', help='bar help')
>>> parser.print_help()
usage: PROG [--bar BAR] foo

group1:
  group1 description

  foo    foo help

group2:
  group2 description

  --bar BAR  bar help
```

请注意任意不在你的自定义分组中的参数最终都将回到通常的“位置参数”和“可选参数”分组中。

互斥

ArgumentParser.**add_mutually_exclusive_group**(*required=False*)

创建一个互斥组。*argparse* 将会确保互斥组中只有一个参数在命令行中可用:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group()
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args(['--foo'])
Namespace(bar=True, foo=True)
>>> parser.parse_args(['--bar'])
Namespace(bar=False, foo=False)
```

(下页继续)

(繼續上一頁)

```
>>> parser.parse_args(['--foo', '--bar'])
usage: PROG [-h] [--foo | --bar]
PROG: error: argument --bar: not allowed with argument --foo
```

add_mutually_exclusive_group() 方法也接受一个 *required* 参数，表示在互斥组中至少有一个参数是需要的:

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> group = parser.add_mutually_exclusive_group(required=True)
>>> group.add_argument('--foo', action='store_true')
>>> group.add_argument('--bar', action='store_false')
>>> parser.parse_args([])
usage: PROG [-h] (--foo | --bar)
PROG: error: one of the arguments --foo --bar is required
```

注意，目前互斥参数组不支持*add_argument_group()* 的 *title* 和 *description* 参数。

解析器默认值

ArgumentParser.**set_defaults**(***kwargs*)

在大多数时候，*parse_args()* 所返回对象的属性将完全通过检查命令行参数和参数动作来确定。*set_defaults()* 则允许加入一些无须任何命令行检查的额外属性:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('foo', type=int)
>>> parser.set_defaults(bar=42, baz='badger')
>>> parser.parse_args(['736'])
Namespace(bar=42, baz='badger', foo=736)
```

请注意解析器层级的默认值总是会覆盖参数层级的默认值:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='bar')
>>> parser.set_defaults(foo='spam')
>>> parser.parse_args([])
Namespace(foo='spam')
```

解析器层级默认值在需要多解析器时会特别有用。请参阅*add_subparsers()* 方法了解此类型的一个示例。

ArgumentParser.**get_default**(*dest*)

获取一个命名空间属性的默认值，该值是由*add_argument()* 或*set_defaults()* 设置的:

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', default='badger')
>>> parser.get_default('foo')
'badger'
```

打印帮助

在大多数典型应用中，*parse_args()* 将负责任何用法和错误消息的格式化和打印。但是，也可使用某些其他格式化方法:

ArgumentParser.**print_usage**(*file=None*)
打印一段简短描述，说明应当如何在命令行中发起调用*ArgumentParser*。如果 *file* 为 None，则默认使用*sys.stdout*。

ArgumentParser.**print_help**(*file=None*)
打印一条帮助消息，包括程序用法和通过*ArgumentParser* 注册的相关参数信息。如果 *file* 为 None，则默认使用*sys.stdout*。

还存在这些方法的几个变化形式，它们只返回字符串而不打印消息:

ArgumentParser.**format_usage**()
返回一个包含简短描述的字符串，说明应当如何在命令行中发起调用*ArgumentParser*。

ArgumentParser.**format_help**()
反回一个包含帮助消息的字符串，包括程序用法和通过*ArgumentParser* 注册的相关参数信息。

部分解析

ArgumentParser.**parse_known_args**(*args=None*, *namespace=None*)

有时一个脚本可能只解析部分命令行参数，而将其余的参数继续传递给另一个脚本或程序。在这种情况下，*parse_known_args()* 方法会很有用处。它的作用方式很类似*parse_args()* 但区别在于当存在额外参数时它不会产生错误。而是会返回一个由两个条目构成的元组，其中包含带成员的命名空间和剩余参数字符串的列表。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo', action='store_true')
>>> parser.add_argument('bar')
>>> parser.parse_known_args(['--foo', '--badger', 'BAR', 'spam'])
(Namespace(bar='BAR', foo=True), ['--badger', 'spam'])
```

警告: 前缀匹配 规则应用于 parse_known_args()。一个选项即使只是已知选项的前缀部分解析器也能识别该选项，不会将其放入剩余参数列表。

自定义文件解析

ArgumentParser.**convert_arg_line_to_args**(*arg_line*)
从文件读取的参数（见*ArgumentParser* 的 *fromfile_prefix_chars* 关键字参数）将是一行读取一个参数。*convert_arg_line_to_args()* 可被重载以使用更复杂的读取方式。

此方法接受从参数文件读取的字符串形式的单个参数 *arg_line*。它返回从该字符串解析出的参数列表。此方法将在每次按顺序从参数文件读取一行时被调用一次。

此方法的一个有用的重载是将每个以空格分隔的单词视为一个参数。下面的例子演示了如何实现此重载:

```
class MyArgumentParser(argparse.ArgumentParser):
    def convert_arg_line_to_args(self, arg_line):
        return arg_line.split()
```

退出方法

ArgumentParser.**exit**(*status=0*, *message=None*)

此方法将终结程序，退出时附带指定的 *status*，并且如果给出了 *message* 则会在退出前将其打印输出。用户可重载此方法以不同方式来处理这些步骤:

```
class ErrorCatchingArgumentParser(argparse.ArgumentParser):
    def exit(self, status=0, message=None):
        if status:
            raise Exception(f'Exiting because of an error: {message}')
        exit(status)
```

ArgumentParser.**error**(*message*)

此方法将向标准错误打印包括 *message* 的用法消息并附带状态码 2 终结程序。

混合解析

ArgumentParser.**parse_intermixed_args**(*args=None*, *namespace=None*)

ArgumentParser.**parse_known_intermixed_args**(*args=None*, *namespace=None*)

许多 Unix 命令允许用户混用可选参数与位置参数。*parse_intermixed_args()* 和*parse_known_intermixed_args()* 方法均支持这种解析风格。

这些解析器并不支持所有的 argparse 特性，并且当未支持的特性被使用时将会引发异常。特别地，子解析器，argparse.REMAINDER 以及同时包括可选与位置参数的互斥分组是不受支持的。

下面的例子显示了*parse_known_args()* 与*parse_intermixed_args()* 之间的差异：前者会将 ['2', '3'] 返回为未解析的参数，而后者会将所有位置参数收集至 rest 中。

```
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument('--foo')
>>> parser.add_argument('cmd')
>>> parser.add_argument('rest', nargs='*', type=int)
>>> parser.parse_known_args('doit 1 --foo bar 2 3'.split())
(Namespace(cmd='doit', foo='bar', rest=[1]), ['2', '3'])
>>> parser.parse_intermixed_args('doit 1 --foo bar 2 3'.split())
Namespace(cmd='doit', foo='bar', rest=[1, 2, 3])
```

parse_known_intermixed_args() 返回由两个条目组成的元组，其中包含带成员的命名空间以及剩余参数字符串列表。当存在任何剩余的未解析参数字符串时*parse_intermixed_args()* 将引发一个错误。

3.7 版新加入.

16.4.6 升级 optparse 代码

起初，*argparse* 曾经尝试通过*optparse* 来维持兼容性。但是，*optparse* 很难透明地扩展，特别是那些为支持新的 nargs= 描述方式和更好的用法消息所需的修改。当 When most everything in *optparse* 中几乎所有内容都已被复制粘贴或打上补丁时，维持向下兼容看来已是不切实际的。

argparse 模块在许多方面对标准库的*optparse* 模块进行了增强，包括:

- 处理位置参数。
- 支持子命令。
- 允许替代选项前缀例如 + 和 /。
- 处理零个或多个以及一个或多个风格的参数。

- 生成更具信息量的用法消息。
- 提供用于定制 type 和 action 的更为简单的接口。

从*optparse* 到*argparse* 的部分升级路径:

- 将 所 有*optparse.OptionParser.add_option()* 调 用 替 换 为*ArgumentParser.add_argument()* 调用。
- 将 (options, args) = parser.parse_args() 替换为 args = parser.parse_args() 并为位置参数添加额外的*ArgumentParser.add_argument()* 调用。请注意之前所谓的 options 在*argparse* 上下文中被称为 args。
- 通过使用*parse_intermixed_args()* 而非*parse_args()* 来替换*optparse.OptionParser.disable_interspersed_args()*。
- 将回调动作和 callback_* 关键字参数替换为 type 或 action 参数。
- 将 type 关键字参数字符串名称替换为相应的类型对象（例如 int, float, complex 等）。
- 将 optparse.Values 替 换 为*Namespace* 并 将 optparse.OptionError 和 optparse.OptionValueError 替换为 ArgumentError。
- 将隐式参数字符串例如使用标准 Python 字典语法的 %default 或 %prog 替换为格式字符串，即 %(default)s 和 %(prog)s。
- 将 OptionParser 构 造 器 version 参 数 替 换 为 对 parser.add_argument('--version', action='version', version='<the version>') 的调用。

16.5 getopt --- C 风格的命令行选项解析器

源代码: Lib/getopt.py

備𝔽: *getopt* 模块是一个命令行选项解析器，其 API 设计会让 C getopt() 函数的用户感到熟悉。不熟悉 C getopt() 函数或者希望写更少代码并获得更完善帮助和错误消息的用户应当考虑改用*argparse* 模块。

此模块可协助脚本解析 sys.argv 中的命令行参数。它支持与 Unix getopt() 函数相同的惯例（包括形式如'-' 与'--' 的参数的特殊含义）。也能通过可选的第三个参数来使用与 GNU 软件所支持形式相类似的长选项。

此模块提供了两个函数和一个异常:

getopt.**getopt**(*args*, *shortopts*, *longopts=[]*)
解析命令行选项与形参列表。*args* 为要解析的参数列表，不包含最开头的对正在运行的程序的引用。通常这意味着 sys.argv[1:]。*shortopts* 为脚本所要识别的字母选项，包含要求后缀一个冒号 (':'；即与 Unix getopt() 所用的格式相同) 的选项。

備𝔽: 与 GNU getopt() 不同，在非选项参数之后，所有后续参数都会被视为非选项。这类似于非 GNU Unix 系统的运作方式。

如果指定了 *longopts*，则必须为一个由应当被支持的长选项名称组成的列表。开头的 '--' 字符不应被包括在选项名称中。要求参数的长选项后应当带一个等号 ('=')。可选参数不被支持。如果想仅接受长选项，则 *shortopts* 应为一个空字符串。命令行中的长选项只要提供了恰好能匹配可接受选项之一的选项名称前缀即可被识别。举例来说，如果 *longopts* 为 ['foo', 'frob']，则选项 --fo 将匹配为 --foo，但 --f 将不能得到唯一匹配，因此将引发*GetoptError*。

返回值由两个元素组成：第一个是 (option, value) 对的列表；第二个是在去除该选项列表后余下的程序参数列表（这也就是 *args* 的尾部切片）。每个被返回的选项与值对的第一个元素是选项，短选项前缀一个连字符 (例如 '-x')，长选项则前缀两个连字符 (例如 '--long-option')，第二个元素是选项参数，如果选项不带参数则为空字符串。列表中选项的排列顺序与它们被解析的顺序相同，因此允许多次出现。长选项与短选项可以混用。

getopt.**gnu_getopt**(*args*, *shortopts*, *longopts=[]*)

此函数与*getopt()* 类似，区别在于它默认使用 GNU 风格的扫描模式。这意味着选项和非选项参数可能会混在一起。*getopt()* 函数将在遇到非选项参数时立即停止处理选项。

如果选项字符串的第一个字符为 '+'，或者如果设置了环境变量 POSIXLY_CORRECT，则选项处理会在遇到非选项参数时立即停止。

exception getopt.**GetoptError**

This is raised 当参数列表中出现不可识别的选项或者当一个需要参数的选项未带参数时将引发此异常。此异常的参数是一个指明错误原因的字符串。对于长选项，将一个参数传给不需要参数的选项也将导致引发此异常。msg 和 opt 属性会给出错误消息和关联的选项；如果没有关联到异常的特定选项，则 opt 将为空字符串。

exception getopt.**error**

GetoptError 的别名；用于向后兼容。

一个仅使用 Unix 风格选项的例子:

```
>>> import getopt
>>> args = '-a -b -cfoo -d bar a1 a2'.split()
>>> args
['-a', '-b', '-cfoo', '-d', 'bar', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'abc:d:')
>>> optlist
[('-a', ''), ('-b', ''), ('-c', 'foo'), ('-d', 'bar')]
>>> args
['a1', 'a2']
```

使用长选项名也同样容易:

```
>>> s = '--condition=foo --testing --output-file abc.def -x a1 a2'
>>> args = s.split()
>>> args
['--condition=foo', '--testing', '--output-file', 'abc.def', '-x', 'a1', 'a2']
>>> optlist, args = getopt.getopt(args, 'x', [
...     'condition=', 'output-file=', 'testing'])
>>> optlist
[('--condition', 'foo'), ('--testing', ''), ('--output-file', 'abc.def'), ('-x', '')]
>>> args
['a1', 'a2']
```

在脚本中，典型的用法类似这样:

```
import getopt, sys

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "ho:v", ["help", "output="])
    except getopt.GetoptError as err:
        # print help information and exit:
        print(err)  # will print something like "option -a not recognized"
        usage()
        sys.exit(2)
```

(下页继续)

(繼續上一頁)

```
    output = None
    verbose = False
    for o, a in opts:
        if o == "-v":
            verbose = True
        elif o in ("-h", "--help"):
            usage()
            sys.exit()
        elif o in ("-o", "--output"):
            output = a
        else:
            assert False, "unhandled option"
    # ...

if __name__ == "__main__":
    main()
```

请注意通过*argparse* 模块可以使用更少的代码并附带更详细的帮助与错误消息生成等价的命令行接口:

```
import argparse

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-o', '--output')
    parser.add_argument('-v', dest='verbose', action='store_true')
    args = parser.parse_args()
    # ... do something with args.output ...
    # ... do something with args.verbose ..
```

也参考:

模块*argparse* 替代的命令行选项和参数解析库。

16.6 `logging` --- Python 的日志记录工具

源代码: Lib/logging/__init__.py

Important

此页面仅包含 API 参考信息。教程信息和更多高级用法的讨论，请参阅

- 基础教程
- 进阶教程
- 日志操作手册

这个模块为应用与库实现了灵活的事件日志系统的函数与类。

使用标准库提供的 logging API 最主要的好处是，所有的 Python 模块都可能参与日志输出，包括你自己的日志消息和第三方模块的日志消息。

这个模块提供许多强大而灵活的功能。如果对 logging 不太熟悉，掌握它最好的方式就是查看它对应的教程（详见右侧的链接）。

该模块定义的基础类和函数都列在下面。

- 记录器暴露了应用程序代码直接使用的接口。
- 处理器将日志记录（由记录器创建）发送到适当的目标。
- 过滤器提供了更细粒度的功能，用于确定要输出的日志记录。
- 格式器指定最终输出中日志记录的样式。

16.6.1 记录器对象

记录器有以下的属性和方法。注意 永远不要直接实例化记录器，应当通过模块级别的函数 `logging.getLogger(name)` 。多次使用相同的名字调用*getLogger()* 会一直返回相同的 Logger 对象的引用。

`name` 一般是句点分割的层级值, 像 "foo.bar.baz" (尽管也可以只是普通的 `foo`)。层次结构列表中位于下方的记录器是列表中较高位置的记录器的子级。例如，有个名叫 `foo` 的记录器，而名字是 `foo.bar`，`foo.bar.baz`，和 `foo.bam` 的记录器都是 `foo` 的子级。记录器的名字分级类似 Python 包的层级，如果使用建议的结构 `logging.getLogger(__name__)` 在每个模块的基础上组织记录器，则与之完全相同。这是因为在模块里，`__name__` 是该模块在 Python 包命名空间中的名字。

class logging.**Logger**

propagate
如果这个属性为真，记录到这个记录器的事件除了会发送到此记录器的所有处理程序外，还会传递给更高级别（祖先）记录器的处理器，此外任何关联到这个记录器的处理器。消息会直接传递给祖先记录器的处理器——不考虑祖先记录器的级别和过滤器。

如果为假，记录消息将不会传递给当前记录器的祖先记录器的处理器。

Spelling it out with an example: If the propagate attribute of the logger named `A.B.C` evaluates to true, any event logged to `A.B.C` via a method call such as `logging.getLogger('A.B.C').error(...)` will [subject to passing that logger's level and filter settings] be passed in turn to any handlers attached to loggers named `A.B`, `A` and the root logger, after first being passed to any handlers attached to `A.B.C`. If any logger in the chain `A.B.C`, `A.B`, `A` has its `propagate` attribute set to false, then that is the last logger whose handlers are offered the event to handle, and propagation stops at that point.

构造器将这个属性初始化为 `True`。

備🄵: 如果将处理器附加到记录器 和其一个或多个祖先记录器，它可能发出多次相同的记录。通常，不需要将处理器附加到一个以上的记录器上——如果把它附加到记录器层次结构中最高的适当记录器上，则它将看到所有后代记录器记录的所有事件，前提是它们的传播设置保留为 `True`。仅将处理器附加到根记录器，通过传播来处理其余部分这种方案是比较常用的。

---

setLevel(*level*)
给记录器设置阈值为 *level* 。日志等级小于 *level* 会被忽略。严重性为 *level* 或更高的日志消息将由该记录器的任何一个或多个处理器发出，除非将处理器的级别设置为比 *level* 更高的级别。

创建记录器时，级别默认设置为 `NOTSET` （当记录器是根记录器时，将处理所有消息；如果记录器不是根记录器，则将委托给父级）。请注意，根记录器的默认级别为 `WARNING` 。

委派给父级的意思是如果记录器的级别设置为 NOTSET，将遍历其祖先记录器，直到找到级别不是 NOTSET 的记录器，或者到根记录器为止。

如果发现某个父级的级别不是 NOTSET ，那么该父级的级别将被视为发起搜索的记录器的有效级别，并用于确定如何处理日志事件。

如果搜索到达根记录器，并且其级别为 NOTSET，则将处理所有消息。否则，将使用根记录器的级别作为有效级别。

参见日志级别 级别列表。

3.2 版更變: 现在 *level* 参数可以接受形如'INFO' 的级别字符串表示形式，以代替形如 INFO 的整数常量。但是请注意，级别在内部存储为整数，并且*getEffectiveLevel()* 和*isEnabledFor()* 等方法的传入/返回值也为整数。

isEnabledFor(*level*)
指示此记录器是否将处理级别为 *level* 的消息。此方法首先检查由 logging.disable(level) 设置的模块级的级别，然后检查由*getEffectiveLevel()* 确定的记录器的有效级别。

getEffectiveLevel()
指示此记录器的有效级别。如果通过*setLevel()* 设置了除 NOTSET 以外的值，则返回该值。否则，将层次结构遍历到根，直到找到除 NOTSET 以外的其他值，然后返回该值。返回的值是一个整数，通常为 logging.DEBUG、logging.INFO 等等。

getChild(*suffix*)
返回由后缀确定的该记录器的后代记录器。因此，logging.getLogger('abc').getChild('def.ghi') 与 logging.getLogger('abc.def.ghi') 将返回相同的记录器。这是一个便捷方法，当使用如 __name__ 而不是字符串字面值命名父记录器时很有用。

3.2 版新加入.

debug(*msg*, **args*, ***kwargs*)
在此记录器上记录 DEBUG 级别的消息。*msg* 是消息格式字符串，而 *args* 是用于字符串格式化操作合并到 *msg* 的参数。（注意，这里的意思是可以在格式字符串中使用关键字，及单个字典参数。）未提供 *args* 时，不会对 *msg* 执行%格式化操作。

kwargs 中会检查四个关键字参数：*exc_info* ，*stack_info* ，*stacklevel* 和 *extra* 。

如果 *exc_info* 的求值结果不为 false，则它将异常信息添加到日志消息中。如果提供了异常元组（按照*sys.exc_info()* 返回的格式）或异常实例，则它将被使用；否则，调用*sys.exc_info()* 以获取异常信息。

第二个可选关键字参数是 *stack_info*，默认为 False。如果为 True，则将堆栈信息添加到日志消息中，包括实际的日志调用。请注意，这与通过指定 *exc_info* 显示的堆栈信息不同：前者是从堆栈底部到当前线程中的日志记录调用的堆栈帧，而后者是在搜索异常处理程序时，跟踪异常而打开的堆栈帧的信息。

您可以独立于 *exc_info* 来指定 *stack_info*，例如，即使在未引发任何异常的情况下，也可以显示如何到达代码中的特定点。堆栈帧在标题行之后打印：

```
Stack (most recent call last):
```

这模仿了显示异常帧时所使用的 Traceback (most recent call last): 。

第三个可选关键字参数是 *stacklevel* ，默认为 1 。如果大于 1 ，则在为日志记录事件创建的*LogRecord* 中计算行号和函数名时，将跳过相应数量的堆栈帧。可以在记录帮助器时使用它，以便记录的函数名称，文件名和行号不是帮助器的函数/方法的信息，而是其调用方的信息。此参数是*warnings* 模块中的同名等效参数。

第四个关键字参数是 *extra* ，传递一个字典，该字典用于填充为日志记录事件创建的、带有用户自定义属性的*LogRecord* 中的 __dict__ 。然后可以按照需求使用这些自定义属性。例如，可以将它们合并到已记录的消息中：

```
FORMAT = '%(asctime)s %(clientip)-15s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
```

(下页继续)

(繼續上一頁)

```
logger = logging.getLogger('tcpserver')
logger.warning('Protocol problem: %s', 'connection reset', extra=d)
```

输出类似于

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection␣
↪reset
```

extra 中传入的字典的键不应与日志系统使用的键冲突。(有关日志系统使用哪些键的更多信息，请参见*Formatter* 的文档。)

如果在已记录的消息中使用这些属性，则需要格外小心。例如，在上面的示例中，*Formatter* 已设置了格式字符串，其在*LogRecord* 的属性字典中键值为 “clientip” 和 “user”。如果缺少这些内容，则将不会记录该消息，因为会引发字符串格式化异常。因此，在这种情况下，您始终需要使用 *extra* 字典传递这些键。

尽管这可能很烦人，但此功能旨在用于特殊情况，例如在多个上下文中执行相同代码的多线程服务器，并且出现的有趣条件取决于此上下文（例如在上面的示例中就是远程客户端 IP 地址和已验证用户名)。在这种情况下，很可能将专门的*Formatter* 与特定的*Handler* 一起使用。

3.2 版更變: 增加了 *stack_info* 参数。

3.5 版更變: *exc_info* 参数现在可以接受异常实例。

3.8 版更變: 增加了 *stacklevel* 参数。

info(*msg*, **args*, ***kwargs*)
在此记录器上记录 INFO 级别的消息。参数解释同*debug()*。

warning(*msg*, **args*, ***kwargs*)
在此记录器上记录 WARNING 级别的消息。参数解释同*debug()*。

備F: 有一个功能上与 warning 一致的方法 warn。由于 warn 已被弃用，请勿再用——改为使用 warning。

---

error(*msg*, **args*, ***kwargs*)
在此记录器上记录 ERROR 级别的消息。参数解释同*debug()*。

critical(*msg*, **args*, ***kwargs*)
在此记录器上记录 CRITICAL 级别的消息。参数解释同*debug()*。

log(*level*, *msg*, **args*, ***kwargs*)
在此记录器上记录 *level* 整数代表的级别的消息。参数解释同*debug()*。

exception(*msg*, **args*, ***kwargs*)
在此记录器上记录 ERROR 级别的消息。参数解释同*debug()*。异常信息将添加到日志消息中。仅应从异常处理程序中调用此方法。

addFilter(*filter*)
将指定的过滤器 *filter* 添加到此记录器。

removeFilter(*filter*)
从此记录器中删除指定的过滤器 *filter*。

filter(*record*)
将此记录器的过滤器应用于记录，如果记录能被处理则返回 True。过滤器会被依次使用，直到其中一个返回假值为止。如果它们都不返回假值，则记录将被处理（传递给处理器)。如果返回任一为假值，则不会对该记录做进一步处理。

addHandler(*hdlr*)
将指定的处理器 *hdlr* 添加到此记录器。

removeHandler(*hdlr*)
从此记录器中删除指定的处理器 *hdlr*。

findCaller(*stack_info=False*, *stacklevel=1*)
查找调用源的文件名和行号，以文件名，行号，函数名称和堆栈信息 4 元素元组的形式返回。堆栈信息将返回 None，除非 *stack_info* 为 True。

stacklevel 参数用于调用 *debug()* 和其他 API。如果大于 1，则多余部分将用于跳过堆栈帧，然后再确定要返回的值。当从帮助器/包装器代码调用日志记录 API 时，这通常很有用，以便事件日志中的信息不是来自帮助器/包装器代码，而是来自调用它的代码。

handle(*record*)
通过将记录传递给与此记录器及其祖先关联的所有处理器来处理（直到某个 *propagate* 值为 false）。此方法用于从套接字接收的未序列化的以及在本地创建的记录。使用 *filter()* 进行记录器级别过滤。

makeRecord(*name*, *level*, *fn*, *lno*, *msg*, *args*, *exc_info*, *func=None*, *extra=None*, *sinfo=None*)
这是一种工厂方法，可以在子类中对其进行重写以创建专门的 *LogRecord* 实例。

hasHandlers()
检查此记录器是否配置了任何处理器。通过在此记录器及其记录器层次结构中的父级中查找处理器完成此操作。如果找到处理器则返回 True，否则返回 False。只要找到 “propagate” 属性设置为假值的记录器，该方法就会停止搜索层次结构——其将是最后一个检查处理器是否存在的记录器。

3.2 版新加入.

3.7 版更變: 现在可以对处理器进行序列化和反序列化。

16.6.2 日志级别

日志记录级别的数值在下表中给出。如果你想要定义自己的级别，并且需要它们具有相对于预定义级别的特定值，那么这你可能对以下内容感兴趣。如果你定义具有相同数值的级别，它将覆盖预定义的值；预定义的名称将失效。

级别	数值
CRITICAL	50
ERROR	40
WARNING	30
INFO	20
DEBUG	10
NOTSET	0

16.6.3 处理器对象

Handler 有以下属性和方法。注意不要直接实例化*Handler*；这个类用来派生其他更有用的子类。但是，子类的 __init__() 方法需要调用*Handler.__init__()*。

class logging.**Handler**

__init__(*level=NOTSET*)
初始化*Handler*实例时，需要设置它的级别，将过滤列表置为空，并且创建锁（通过*createLock()*）来序列化对 I/O 的访问。

createLock()
初始化一个线程锁，用来序列化对底层的 I/O 功能的访问，底层的 I/O 功能可能不是线程安全的。

acquire()
获取由*createLock()*创建的线程锁。

release()
释放由*acquire()*获取的线程锁。

setLevel(*level*)
给处理器设置阈值为 *level*。日志级别小于 *level* 将被忽略。创建处理器时，日志级别被设置为 NOTSET（所有的消息都会被处理）。

参见日志级别 级别列表。

3.2 版更變: *level* 形参现在接受像'INFO' 这样的字符串形式的级别表达方式，也可以使用像 INFO 这样的整数常量。

setFormatter(*fmt*)
将此处理器的*Formatter*设置为 *fmt*。

addFilter(*filter*)
将指定的过滤器 *filter* 添加到此处理器。

removeFilter(*filter*)
从此处理器中删除指定的过滤器 *filter*。

filter(*record*)
将此处理器的过滤器应用于记录，在要处理记录时返回 True。依次查询过滤器，直到其中一个返回假值为止。如果它们都不返回假值，则将发出记录。如果返回一个假值，则处理器将不会发出记录。

flush()
确保所有日志记录从缓存输出。此版本不执行任何操作，并且应由子类实现。

close()
回收处理器使用的所有资源。此版本不输出，但从内部处理器列表中删除处理器，内部处理器在*shutdown()*被调用时关闭。子类应确保从重写的*close()*方法中调用此方法。

handle(*record*)
经已添加到处理器的过滤器过滤后，有条件地发出指定的日志记录。用获取/释放 I/O 线程锁包装了记录的实际发出行为。

handleError(*record*)
调用*emit()*期间遇到异常时，应从处理器中调用此方法。如果模块级属性 raiseExceptions 是 False，则异常将被静默忽略。这是大多数情况下日志系统需要的——大多数用户不会关心日志系统中的错误，他们对应用程序错误更感兴趣。但是，你可以根据需要将其替换为自定义处理器。指定的记录是发生异常时正在处理的记录。（raiseExceptions 的默认值是 True，因为这在开发过程中是比较有用的）。

format(*record*)
如果设置了格式器则用其对记录进行格式化。否则，使用模块的默认格式器。

emit(*record*)
执行实际记录给定日志记录所需的操作。这个版本应由子类实现，因此这里直接引发*NotImplementedError*异常。

有关作为标准随附的处理器列表，请参见*logging.handlers*。

16.6.4 格式器对象

*Formatter*对象拥有以下的属性和方法。一般情况下，它们负责将*LogRecord*转换为可由人或外部系统解释的字符串。基础的*Formatter*允许指定格式字符串。如果未提供任何值，则使用默认值`'%(message)s'`，它仅将消息包括在日志记录调用中。要在格式化输出中包含其他信息（如时间戳），请阅读下文。

格式器可以使用格式化字符串来初始化，该字符串利用*LogRecord*的属性——例如上述默认值，用户的消息和参数预先格式化为*LogRecord*的 *message* 属性后被使用。此格式字符串包含标准的 Python %-s 样式映射键。有关字符串格式的更多信息，请参见*printf 风格的字符串格式化*。

*LogRecord 属性*一节中给出了*LogRecord*中有用的映射键。

class logging.**Formatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*)
返回*Formatter*类的新实例。实例将使用整个消息的格式字符串以及消息的日期/时间部分的格式字符串进行初始化。如果未指定 *fmt*，则使用`'%(message)s'`。如果未指定 *datefmt*，则使用*formatTime()*文档中描述的格式。

style 形参可以是'%', '{' 或'$' 之一，它决定格式字符串将如何与数据进行合并：使用%-formatting, *str.format()* 或是*string.Template*。这仅适用于格式字符串 *fmt* (例如 `'%(message)s'` 或 `{message}`)，不适用于传递给 `Logger.debug` 的实际日志消息等；请参阅 formatting-styles 了解有关在日志消息中使用 {- 和 $-formatting 的更多详情。

3.2 版更變: 加入了 *style* 形参。

3.8 版更變: 加入 *validate* 参数。不正确或不匹配的样式和格式将引发 `ValueError` 错误。例如: `logging.Formatter('%(asctime)s - %(message)s', style='{')`。

format(*record*)
记录的属性字典用作字符串格式化操作的参数。返回结果字符串。在格式化字典之前，需要执行几个准备步骤。使用 *msg % args* 计算记录的 *message* 属性。如果格式化字符串包含`'(asctime)'`，则调用*formatTime()*来格式化事件时间。如果有异常信息，则使用*formatException()*将其格式化并附加到消息中。请注意，格式化的异常信息缓存在属性 *exc_text* 中。这很有用，因为可以对异常信息进行序列化并通过网络发送，但是如果您有不止一个定制了异常信息格式的*Formatter*子类，则应格外小心。在这种情况下，您必须在格式器完成格式化后清除缓存的值，以便下一个处理事件的格式器不使用缓存的值，而是重新计算它。

如果栈信息可用，它将被添加在异常信息之后，如有必要请使用*formatStack()*来转换它。

formatTime(*record*, *datefmt=None*)
此方法应由想要使用格式化时间的格式器中的*format()*调用。可以在格式器中重写此方法以提供任何特定要求，但是基本行为如下：如果指定了 *datefmt*（字符串），则将其用于*time.strftime()*来格式化记录的创建时间。否则，使用格式'%Y-%m-%d %H:%M:%S,uuu'，其中 uuu 部分是毫秒值，其他字母根据*time.strftime()*文档。这种时间格式的示例为 `2003-01-23 00:29:50,411`。返回结果字符串。

此函数使用一个用户可配置函数将创建时间转换为元组。默认情况下，使用*time.localtime()*；要为特定格式化程序实例更改此项，请将实例的 `converter` 属性设为具有与*time.localtime()*或*time.gmtime()*相同签名的函数。要为所有格式化程序更改此项，例如当你希望所有日志时间都显示为 GMT，请在 `Formatter` 类中设置 `converter` 属性。

3.3 版更變: 在之前版本中，默认格式是被硬编码的，例如这个例子: `2010-09-06 22:38:15,292` 其中逗号之前的部分由 strptime 格式字符串 (`'%Y-%m-%d %H:%M:%S'`) 处理，而逗号之后的部分为毫秒值。因为 strptime 没有表示毫秒的占位符，毫秒值使用了另外的格式字符串来添加 `'%s,%03d'` --- 这两个格式字符串代码都是硬编码在该方法中的。经过修改，这些字符串被定义为类层级的属性，当需要时可以在实例层级上被重载。属性的名称为 `default_time_format` (用于 strptime 格式字符串) 和 `default_msec_format` (用于添加毫秒值)。

3.9 版更變: `default_msec_format` 可以为 `None`。

formatException(*exc_info*)
将指定的异常信息（由*sys.exc_info()* 返回的标准异常元组）格式化为字符串。默认实现只是使用了*traceback.print_exception()*。结果字符串将被返回。

formatStack(*stack_info*)
将指定的堆栈信息（由*traceback.print_stack()* 返回的字符串，但移除末尾的换行符）格式化为字符串。默认实现只是返回输入值。

16.6.5 过滤器对象

`Filters` 可被 `Handlers` 和 `Loggers` 用来实现比按层级提供更复杂的过滤操作。基本过滤器类只允许低于日志记录器层级结构中低于特定层级的事件。例如，一个用'A.B' 初始化的过滤器将允许'A.B', 'A.B.C', 'A.B.C.D', 'A.B.D' 等日志记录器所记录的事件。但'A.BB', 'B.A.B' 等则不允许。如果用空字符串初始化，则所有事件都会通过。

class logging.**Filter**(*name=''*)
返回一个*Filter* 类的实例。如果指定了 *name*，则它将被用来为日志记录器命名，该类及其子类将通过该过滤器允许指定事件通过。如果 *name* 为空字符串，则允许所有事件通过。

filter(*record*)
是否要记录指定的记录？返回零表示否，非零表示是。如果认为合适，则可以通过此方法就地修改记录。

请注意关联到处理器的过滤器会在事件由处理器发出之前被查询，而关联到日志记录器的过滤器则会在有事件被记录的的任何时候（使用*debug()*, *info()* 等等）在将事件发送给处理器之前被查询。这意味着由后代日志记录器生成的事件将不会被父代日志记录器的过滤器设置所过滤，除非该过滤器也已被应用于后代日志记录器。

你实际上不需要子类化 `Filter`：你可以传入任何一个包含有相同语义的 `filter` 方法的实例。

3.2 版更變: 你不需要创建专门的 `Filter` 类，或使用具有 `filter` 方法的其他类：你可以使用一个函数（或其他可调用对象）作为过滤器。过滤逻辑将检查过滤器对象是否具有 `filter` 属性：如果有，就会将它当作是 `Filter` 并调用它的*filter()* 方法。在其他情况下，则会将它当作是可调用对象并将记录作为唯一的形参进行调用。返回值应当与*filter()* 的返回值相一致。

尽管过滤器主要被用来构造比层级更复杂的规则以过滤记录，但它们可以查看由它们关联的处理器或记录器所处理的每条记录：当你想要执行统计特定记录器或处理器共处理了多少条记录，或是在所处理的*LogRecord* 中添加、修改或移除属性这样的任务时该特性将很有用处。显然改变 LogRecord 时需要相当小心，但将上下文信息注入日志确实是被允许的 (参见 filters-contextual)。

16.6.6 LogRecord 属性

LogRecord 实例是每当有日志被记录时由*Logger* 自动创建的，并且可通过*makeLogRecord()* 手动创建（例如根据从网络接收的已封存事件创建）。

class logging.**LogRecord**(*name*, *level*, *pathname*, *lineno*, *msg*, *args*, *exc_info*, *func=None*, *sinfo=None*)

包含与被记录的事件相关的所有信息。

主要信息是在 msg 和 args 中传递的，它们使用 msg % args 组合到一起以创建记录的 message 字段。

参数

- **name** -- 用于记录由此 LogRecord 所表示事件的记录器名称。请注意此名称将始终为该值，即使它可能是由附加到不同（祖先）日志记录器的处理器所发出的。
- **level** -- 以数字表示的日志记录事件层级（如 DEBUG, INFO 等）。请注意这会转换为 LogRecord 的 两个属性: levelno 为数字值而 levelname 为对应的层级名称。
- **pathname** -- 进行日志记录调用的文件的完整路径名。
- **lineno** -- 记录调用所在源文件中的行号。
- **msg** -- 事件描述消息，可能为带有可变数据占位符的格式字符串。
- **args** -- 要合并到 *msg* 参数以获得事件描述的可变数据。
- **exc_info** -- 包含当前异常信息的异常元组，或者如果没有可用异常信息则为 None。
- **func** -- 发起调用日志记录调用的函数或方法名称。
- **sinfo** -- 一个文本字符串，表示当前线程中从堆栈底部直到日志记录调用的堆栈信息。

getMessage()

在将*LogRecord* 实例与任何用户提供的参数合并之后，返回此实例的消息。如果用户提供给日志记录调用的消息参数不是字符串，则会在其上调用*str()* 以将它转换为字符串。此方法允许将用户定义的类用作消息，类的 __str__ 方法可以返回要使用的实际格式字符串。

3.2 版更變: 通过提供用于创建记录的工厂方法已使得*LogRecord* 的创建更易于配置。该工厂方法可使用*getLogRecordFactory()* 和*setLogRecordFactory()* （在此可查看工厂方法的签名）来设置。

在创建时可使用此功能将你自己的值注入*LogRecord*。你可以使用以下模式:

```
old_factory = logging.getLogRecordFactory()

def record_factory(*args, **kwargs):
    record = old_factory(*args, **kwargs)
    record.custom_attribute = 0xdecafbad
    return record

logging.setLogRecordFactory(record_factory)
```

通过此模式，多个工厂方法可以被链接起来，并且只要它们不重载彼此的属性或是在无意中覆盖了上面列出的标准属性，就不会发生意外。

16.6.7 LogRecord 属性

LogRecord 具有许多属性，它们大多数来自于传递给构造器的形参。（请注意 LogRecord 构造器形参与 LogRecord 属性的名称并不总是完全彼此对应的。）这些属性可被用于将来自记录的数据合并到格式字符串中。下面的表格（按字母顺序）列出了属性名称、它们的含义以及相应的%-style 格式字符串内占位符。

如果是使用 {}-格式化（*str.format()*），你可以将 `{attrname}` 用作格式字符串内的占位符。如果是使用 $-格式化（*string.Template*），则会使用 `${attrname}` 的形式。当然在这两种情况下，都应当将 `attrname` 替换为你想要使用的实际属性名称。

在 {}-格式化的情况下，你可以在属性名称之后放置指定的格式化旗标，并用冒号来分隔两者。例如，占位符 `{msecs:03d}` 会将毫秒值 `4` 格式化为 `004`。请参看*str.format()* 文档了解你所能使用的选项的完整细节。

属性名称	格式	描述
args	此属性不需要用户进行格式化。	合并到 `msg` 以产生 `message` 的包含参数的元组，或是其中的值将被用于合并的字典（当只有一个参数且其类型为字典时）。
asctime	`%(asctime)s`	表示*LogRecord* 何时被创建的供人查看时间值。默认形式为'2003-07-08 16:49:45,896'（逗号之后的数字为时间的毫秒部分）。
created	`%(created)f`	*LogRecord* 被创建的时间（即*time.time()* 的返回值）。
exc_info	此属性不需要用户进行格式化。	异常元组（例如 `sys.exc_info`）或者如未发生异常则为 `None`。
filename	`%(filename)s`	`pathname` 的文件名部分。
funcName	`%(funcName)s`	函数名包括调用日志记录.
levelname	`%(levelname)s`	消息文本记录级别（`'DEBUG'`, `'INFO'`, `'WARNING'`, `'ERROR'`, `'CRITICAL'`）。
levelno	`%(levelno)s`	消息数字的记录级别 (`DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`).
lineno	`%(lineno)d`	发出日志记录调用所在的源行号（如果可用）。
message	`%(message)s`	记入日志的消息，即 `msg % args` 的结果。这是在发起调用*Formatter.format()* 时设置的。
module	`%(module)s`	模块 (`filename` 的名称部分)。
msecs	`%(msecs)d`	*LogRecord* 被创建的时间的毫秒部分。
msg	此属性不需要用户进行格式化。	在原始日志记录调用中传入的格式字符串。与 `args` 合并以产生 `message`，或是一个任意对象 (参见 arbitrary-object-messages)。
name	`%(name)s`	用于记录调用的日志记录器名称。
pathname	`%(pathname)s`	发出日志记录调用的源文件的完整路径名（如果可用）。
process	`%(process)d`	进程 ID（如果可用）
processName	`%(processName)s`	进程名（如果可用）
relativeCreated	`%(relativeCreated)d`	以毫秒数表示的 LogRecord 被创建的时间，即相对于 logging 模块被加载时间的差值。
stack_info	此属性不需要用户进行格式化。	当前线程中从堆栈底部起向上直到包括日志记录调用并引发创建当前记录堆栈帧创建的堆栈帧信息（如果可用）。
thread	`%(thread)d`	线程 ID（如果可用）
threadName	`%(threadName)s`	线程名（如果可用）

3.1 版更變: 添加了 *processName*

16.6.8 LoggerAdapter 对象

LoggerAdapter 实例会被用来方便地将上下文信息传入日志记录调用。要获取用法示例，请参阅 添加上下文信息到你的日志记录输出部分。

class logging.**LoggerAdapter**(*logger*, *extra*)
返回一个*LoggerAdapter* 的实例，该实例的初始化使用了下层的*Logger* 实例和一个字典类对象。

process(*msg*, *kwargs*)
修改传递给日志记录调用的消息和/或关键字参数以便插入上下文信息。此实现接受以 *extra* 形式传给构造器的对象并使用'extra' 键名将其加入 *kwargs*。返回值为一个 (*msg*, *kwargs*) 元组，其包含（可能经过修改的）传入参数。

在上述方法之外，*LoggerAdapter* 还支持*Logger* 的下列方法: *debug()*，*info()*，*warning()*，*error()*，*exception()*，*critical()*，*log()*，*isEnabledFor()*，*getEffectiveLevel()*，*setLevel()* 以及*hasHandlers()*。这些方法具有与它们在*Logger* 中的对应方法相同的签名，因此你可以互换使用这两种类型的实例。

3.2 版更變: *isEnabledFor()*，*getEffectiveLevel()*，*setLevel()* 和*hasHandlers()* 方法已被添加到*LoggerAdapter*。这些方法会委托给下层的日志记录器。

3.6 版更變: 增加了 manager 属性和 _log() 方法，它们会委托给下层的日志记录器并允许适配器嵌套。

16.6.9 线程安全

logging 模块的目标是使客户端不必执行任何特殊操作即可确保线程安全。它通过使用线程锁来达成这个目标；用一个锁来序列化对模块共享数据的访问，并且每个处理程序也会创建一个锁来序列化对其下层 I/O 的访问。

如果你要使用*signal* 模块来实现异步信号处理程序，则可能无法在这些处理程序中使用 logging。这是因为*threading* 模块中的锁实现并非总是可重入的，所以无法从此类信号处理程序发起调用。

16.6.10 模块级函数

在上述的类之外，还有一些模块级的函数。

logging.**getLogger**(*name=None*)
返回具有指定 name 的日志记录器，或者当 name 为 None 时返回层级结构中的根日志记录器。如果指定了 name，它通常是以点号分隔的带层级结构的名称，如 *'a'*、*'a.b'* 或 *'a.b.c.d'*。这些名称的选择完全取决于使用 logging 的开发者。

所有用给定的 name 对该函数的调用都将返回相同的日志记录器实例。这意味着日志记录器实例不需要在应用的各部分间传递。

logging.**getLoggerClass**()
返回标准的*Logger* 类，或是最近传给*setLoggerClass()* 的类。此函数可以从一个新的类定义中调用，以确保安装自定义的*Logger* 类不会撤销其他代码已经应用的自定义操作。例如:

```
class MyLogger(logging.getLoggerClass()):
    # ... override behaviour here
```

logging.**getLogRecordFactory**()
返回一个被用来创建*LogRecord* 的可调用对象。

3.2 版新加入: 此函数与*setLogRecordFactory()* 一起提供，以允许开发者对表示日志记录事件的*LogRecord* 的构造有更好的控制。

请参阅*setLogRecordFactory()* 了解有关如何调用该工厂方法的更多信息。

logging.**debug**(*msg*, **args*, ***kwargs*)

在根日志记录器上记录一条 DEBUG 级别的消息。*msg* 是消息格式字符串，而 *args* 是要使用字符串格式化运算符合并到 *msg* 的参数。（请注意这意味着你可以在格式字符串中使用关键字以及单个字典参数。）

在 *kwargs* 中有三个关键字参数会被检查: *exc_info* 参数如果不为假值则会将异常信息添加到日志记录消息。如果提供了异常元组（为*sys.exc_info()* 的返回值格式）或异常实例则它会被使用；在其他情况下，会调用*sys.exc_info()* 以获取异常信息。

第二个可选关键字参数是 *stack_info*，默认为 False。如果为 True，则将堆栈信息添加到日志消息中，包括实际的日志调用。请注意，这与通过指定 *exc_info* 显示的堆栈信息不同：前者是从堆栈底部到当前线程中的日志记录调用的堆栈帧，而后者是在搜索异常处理程序时，跟踪异常而打开的堆栈帧的信息。

您可以独立于 *exc_info* 来指定 *stack_info*，例如，即使在未引发任何异常的情况下，也可以显示如何到达代码中的特定点。堆栈帧在标题行之后打印：

```
Stack (most recent call last):
```

这模仿了显示异常帧时所使用的 Traceback (most recent call last): 。

第三个可选关键字参数是 *extra*，它可被用来传递一个字典，该字典会被用来填充为日志记录事件创建并附带用户自定义属性的 LogRecord 的 __dict__。之后将可按你的需要使用这些自定义属性。例如，可以将它们合并到已记录的消息中。举例来说:

```
FORMAT = '%(asctime)s %(clientip)-15s %(user)-8s %(message)s'
logging.basicConfig(format=FORMAT)
d = {'clientip': '192.168.0.1', 'user': 'fbloggs'}
logging.warning('Protocol problem: %s', 'connection reset', extra=d)
```

应当会打印出这样的内容:

```
2006-02-08 22:20:02,165 192.168.0.1 fbloggs  Protocol problem: connection reset
```

extra 中传入的字典的键不应与日志系统使用的键冲突。（有关日志系统使用哪些键的更多信息，请参见*Formatter* 的文档。）

如果你选择在已记录的消息中使用这些属性，则需要格外小心。例如在上面的示例中，*Formatter* 已设置了格式字符串，其在 LogRecord 的属性字典中应有'clientip' 和'user'。如果缺少这些属性，消息将不被记录，因为会引发字符串格式化异常，你始终需要传入带有这些键的 *extra* 字典。

尽管这可能很烦人，但此功能旨在用于特殊情况，例如在多个上下文中执行相同代码的多线程服务器，并且出现的有趣条件取决于此上下文（例如在上面的示例中就是远程客户端 IP 地址和已验证用户名）。在这种情况下，很可能将专门的*Formatter* 与特定的*Handler* 一起使用。

3.2 版更變: 增加了 *stack_info* 参数。

logging.**info**(*msg*, **args*, ***kwargs*)

在根日志记录器上记录一条 INFO 级别的消息。参数解释同*debug()*。

logging.**warning**(*msg*, **args*, ***kwargs*)

在根日志记录器上记录一条 WARNING 级别的消息。参数解释同*debug()*。

備F: 有一个已过时方法 warn 其功能与 warning 一致。由于 warn 已被弃用，请不要使用它——而是改用 warning。

---

logging.**error**(*msg*, **args*, ***kwargs*)

在根日志记录器上记录一条 ERROR 级别的消息。参数解释同*debug()*。

logging.**critical**(*msg*, **args*, ***kwargs*)

在根日志记录器上记录一条 CRITICAL 级别的消息。参数解释同*debug()*。

logging.**exception**(*msg*, **args*, ***kwargs*)
在根日志记录器上记录一条 ERROR 级别的消息。参数解释同*debug()*。异常信息将被添加到日志消息中。此函数应当仅从异常处理程序中调用。

logging.**log**(*level*, *msg*, **args*, ***kwargs*)
在根日志记录器上记录一条 *level* 级别的消息。其他参数解释同*debug()*。

logging.**disable**(*level=CRITICAL*)
为所有日志记录器提供重载的级别 *level*，其优先级高于日志记录器自己的级别。当需要临时限制整个应用程序中的日志记录输出时，此功能会很有用。它的效果是禁用所有重要程度为 *level* 及以下的日志记录调用，因此如果你附带 INFO 值调用它，则所有 INFO 和 DEBUG 事件就会被丢弃，而重要程度为 WARNING 以及上的事件将根据日志记录器的当前有效级别来处理。如果 `logging.disable(logging.NOTSET)` 被调用，它将移除这个重载的级别，因此日志记录输出会再次取决于单个日志记录器的有效级别。

请注意如果你定义了任何高于 CRITICAL 的自定义日志级别（并不建议这样做），你就将无法沿用 *level* 形参的默认值，而必须显式地提供适当的值。

3.7 版更變: *level* 形参默认级别为 CRITICAL。请参阅 bpo-28524 了解此项改变的更多细节。

logging.**addLevelName**(*level*, *levelName*)
在一个内部字典中关联级别 *level* 与文本 *levelName*，该字典会被用来将数字级别映射为文本表示形式，例如在*Formatter* 格式化消息的时候。此函数也可被用来定义你自己的级别。唯一的限制是自定义的所有级别必须使用此函数来注册，级别值必须为正整数并且其应随严重程度而递增。

備𝔽: 如果你考虑要定义你自己的级别，请参阅 custom-levels 部分。

---

logging.**getLevelName**(*level*)
返回日志记录级别 *level* 的字符串表示。

如果 *level* 为预定义的级别 CRITICAL, ERROR, WARNING, INFO 或 DEBUG 之一则你会得到相应的字符串。如果你使用*addLevelName()* 将级别关联到名称则返回你为 *level* 所关联的名称。如果传入了与已定义级别相对应的数字值，则返回对应的字符串表示。

level 形参也接受级别的字符串表示例如'INFO'。在这种情况下，此函数将返回级别所对应的数字值。

如果未传入可匹配的数字或字符串值，则返回字符串'Level %s' % level。

備𝔽: 级别在内部以整数表示（因为它们在日志记录逻辑中需要进行比较）。此函数被用于在整数级别与通过 `%(levelname)s` 格式描述符方式在格式化日志输出中显示的级别名称之间进行相互的转换(参见*LogRecord 属性*)。

3.4 版更變: 在早于 3.4 的 Python 版本中，此函数也可传入一个字符串形式的级别名称，并将返回对应的级别数字值。此未记入文档的行为被视为是一个错误，并在 Python 3.4 中被移除，但又在 3.4.2 中被恢复以保持向下兼容性。

logging.**makeLogRecord**(*attrdict*)
创建并返回一个新的*LogRecord* 实例，实例属性由 *attrdict* 定义。此函数适用于接受一个通过套接字传输的封存好的*LogRecord* 属性字典，并在接收端将其重建为一个*LogRecord* 实例。

logging.**basicConfig**(***kwargs*)
通过使用默认的*Formatter* 创建一个*StreamHandler* 并将其加入根日志记录器来为日志记录系统执行基本配置。如果没有为根日志记录器定义处理器则*debug()*, *info()*, *warning()*, *error()* 和*critical()* 等函数将自动调用*basicConfig()*。

如果根日志记录器已配置了处理器则此函数将不执行任何操作，除非关键字参数 *force* 被设为 True。

備𝔽: 此函数应当在其他线程启动之前从主线程被调用。在 2.7.1 和 3.2 之前的 Python 版本中，如果此函数从多个线程被调用，一个处理器（在极少的情况下）有可能被多次加入根日志记录器，导致非预期的结果例如日志中的消息出现重复。

支持以下关键字参数。

格式	描述
filename	使用指定的文件名而不是 StreamHandler 创建 FileHandler。
filemode	如果指定了 *filename*，则用此模式 打开该文件。默认模式为 `'a'`。
format	使用指定的格式字符串作为处理器。默认为属性以冒号分隔的 `levelname`, `name` 和 `message`。
datefmt	使用指定的日期/时间格式，与*`time.strftime()`* 所接受的格式相同。
style	如果指定了 *format*，将为格式字符串使用此风格。`'%'`, `'{'` 或 `'$'` 分别对应于*printf 风格*, *`str.format()`* 或*`string.Template`*。默认为 `'%'`。
level	设置根记录器级别去指定*level*.
stream	使用指定的流初始化 StreamHandler。请注意此参数与 *filename* 是不兼容的 - 如果两者同时存在，则会引发 `ValueError`。
handlers	如果指定，这应为一个包含要加入根日志记录器的已创建处理器的可迭代对象。任何尚未设置格式描述符的处理器将被设置为在此函数中创建的默认格式描述符。请注意此参数与 *filename* 或 *stream* 不兼容——如果两者同时存在，则会引发 `ValueError`。
force	如果将此关键字参数指定为 true，则在执行其他参数指定的配置之前，将移除并关闭附加到根记录器的所有现有处理器。
encoding	如果此关键字参数与 *filename* 一同被指定，则该值会在创建 FileHandler 时被使用，因而也会在打开输出文件时被使用。
errors	如果此关键字参数与 *filename* 一同被指定，则该值会在创建 FileHandler 时被使用，因而也会在打开输出文件时被使用。如果未指定，则会使用值'backslashreplace'。请注意如果指定为 `None`，它将被原样传给 func:*open*，这意味着将会与传入'errors' 一样处理。

3.2 版更變: 增加了 *style* 参数。

3.3 版更變: 增加了 *handlers* 参数。增加了额外的检查来捕获指定不兼容参数的情况 (例如同时指定 *handlers* 与 *stream* 或 *filename*，或者同时指定 *stream* 与 *filename*)。

3.8 版更變: 增加了 *force* 参数。

3.9 版更變: 增加了 *encoding* 和 *errors* 参数。

`logging.`**`shutdown`**`()`
通过刷新和关闭所有处理程序来通知日志记录系统执行有序停止。此函数应当在应用退出时被调用并且在此调用之后不应再使用日志记录系统。

当 logging 模块被导入时，它会将此函数注册为退出处理程序 (参见*`atexit`*)，因此通常不需要手动执行该操作。

`logging.`**`setLoggerClass`**`(`*`klass`*`)`
通知日志记录系统在实例化日志记录器时使用 *klass* 类。该类应当定义 `__init__()` 使其只要求一个 name 参数，并且 `__init__()` 应当调用 `Logger.__init__()`。此函数通常会在需要使用自定义日志记录器行为的应用程序实例化任何日志记录器之前被调用。在此调用之后，在任何其他时刻都不要使用该子类来直接实例化日志记录器：请继续使用*`logging.getLogger()`* API 来获取你的日志记录器。

`logging.`**`setLogRecordFactory`**`(`*`factory`*`)`
设置一个用来创建*`LogRecord`* 的可调用对象。

参数 `factory` -- 用来实例化日志记录的工厂可调用对象。

3.2 版新加入: 此函数与*getLogRecordFactory()* 一起提供，以便允许开发者对如何构造表示日志记录事件的*LogRecord* 有更好的控制。

可调用对象 factory 具有如下签名:

```
factory(name, level, fn, lno, msg, args, exc_info, func=None, sinfo=None,
**kwargs)
```

name 日志记录器名称

level 日志记录级别（数字）。

fn 进行日志记录调用的文件的完整路径名。

lno 记录调用所在文件中的行号。

msg 日志消息。

args 日志记录消息的参数。

exc_info 异常元组，或 `None` 。

func 调用日志记录调用的函数或方法的名称。

sinfo 与*traceback.print_stack()* 所提供的类似的栈回溯信息，显示调用的层级结构。

kwargs 其他关键字参数。

16.6.11 模块级属性

logging.**lastResort**

通过此属性提供的“最后处理者”。这是一个以 `WARNING` 级别写入到 `sys.stderr` 的*StreamHandler*，用于在没有任何日志记录配置的情况下处理日志记录事件。最终结果就是将消息打印到 `sys.stderr`，这会替代先前形式为”no handlers could be found for logger XYZ” 的错误消息。如果出于某种原因你需要先前的行为，可将 `lastResort` 设为 `None`。

3.2 版新加入.

16.6.12 与警告模块集成

captureWarnings() 函数可用来将*logging* 和*warnings* 模块集成。

logging.**captureWarnings**(*capture*)

此函数用于打开和关闭日志系统对警告的捕获。

如果 *capture* 是 `True`，则*warnings* 模块发出的警告将重定向到日志记录系统。具体来说，将使用*warnings.formatwarning()* 格式化警告信息，并将结果字符串使用 `WARNING` 等级记录到名为 `'py.warnings'` 的记录器中。

如果 *capture* 是 `False`，则将停止将警告重定向到日志记录系统，并且将警告重定向到其原始目标（即在 `captureWarnings(True)` 调用之前的有效目标）。

也参考:

***logging.config* 模块** 日志记录模块的配置 API 。

***logging.handlers* 模块** 日志记录模块附带的有用处理器。

PEP 282 - Logging 系统 该提案描述了 Python 标准库中包含的这个特性。

Original Python logging package 这是该*logging* 包的原始来源。该站点提供的软件包版本适用于 Python 1.5.2、2.1.x 和 2.2.x，它们不被*logging* 包含在标准库中。

16.7 logging.config --- 日志记录配置

源代码： Lib/logging/config.py

Important

此页面仅包含参考信息。有关教程，请参阅

- 基础教程
- 进阶教程
- 日志操作手册

这一节描述了用于配置 logging 模块的 API。

16.7.1 配置函数

下列函数可配置 logging 模块。它们位于*logging.config* 模块中。它们的使用是可选的 --- 要配置 logging 模块你可以使用这些函数，也可以通过调用主 API (在*logging* 本身定义) 并定义在*logging* 或*logging.handlers* 中声明的处理程序。

logging.config.**dictConfig**(*config*)

从一个字典获取日志记录配置。字典的内容描述见下文的配置字典架构。

如果在配置期间遇到错误，此函数将引发*ValueError*，*TypeError*，*AttributeError* 或*ImportError* 并附带适当的描述性消息。下面是将会引发错误的（可能不完整的）条件列表:

- `level` 不是字符串或者不是对应于实际日志记录级别的字符串。
- `propagate` 值不是布尔类型。
- id 没有对应的目标。
- 在增量调用期间发现不存在的处理程序 id。
- 无效的日志记录器名称。
- 无法解析为内部或外部对象。

解析由 `DictConfigurator` 类执行，该类的构造器可传入用于配置的字典，并且具有 `configure()` 方法。*logging.config* 模块具有可调用属性 `dictConfigClass`，其初始值设为 `DictConfigurator`。你可以使用你自己的适当实现来替换 `dictConfigClass` 的值。

dictConfig() 会调用 `dictConfigClass` 并传入指定的字典，然后在所返回的对象上调用 `configure()` 方法以使配置生效:

```
def dictConfig(config):
    dictConfigClass(config).configure()
```

例如，DictConfigurator 的子类可以在它自己的 __init__() 中调用 DictConfigurator.__init__()，然后设置可以在后续 configure() 调用中使用的自定义前缀。dictConfigClass 将被绑定到这个新的子类，然后就可以与在默认的未定制状态下完全相同的方式调用*dictConfig()*。

3.2 版新加入.

logging.config.**fileConfig**(*fname*, *defaults=None*, *disable_existing_loggers=True*)

从一个*configparser* 格式文件中读取日志记录配置。文件格式应当与配置文件格式 中的描述一致。此函数可在应用程序中被多次调用，以允许最终用户在多个预设配置中进行选择（如果开发者提供了展示选项并加载选定配置的机制）。

参数

- **fname** -- 一个文件名，或一个文件类对象，或是一个派生自*RawConfigParser* 的实例。如果传入了一个派生自 RawConfigParser 的实例，它会被原样使用。否则，将会实例化一个 Configparser，并且它会从作为 fname 传入的对象中读取配置。如果存在*readline()* 方法，则它会被当作一个文件类对象并使用*read_file()* 来读取；在其它情况下，它会被当作一个文件名并传递给*read()*。
- **defaults** -- 要传递给 ConfigParser 的默认值可在此参数中指定。
- **disable_existing_loggers** -- 如果指定为 False，则当执行此调用时已存在的日志记录器会保持启用。默认值为 True 因为这将以向下兼容方式启用旧行为。此行为是禁用任何现有的非根日志记录器除非它们或它们的上级在日志记录配置中被显式地命名。

3.4 版更變: 现在接受*RawConfigParser* 子类的实例作为 fname 的值。这有助于:

- 使用一个配置文件，其中日志记录配置只是全部应用程序配置的一部分。
- 使用从一个文件读取的配置，它随后会在被传给 fileConfig 之前由使用配置的应用程序来修改（例如基于命令行参数或运行时环境的其他部分）。

logging.config.**listen**(*port=DEFAULT_LOGGING_CONFIG_PORT*, *verify=None*)

在指定的端口上启动套接字服务器，并监听新的配置。如果未指定端口，则会使用模块默认的 DEFAULT_LOGGING_CONFIG_PORT。日志记录配置将作为适合由*dictConfig()* 或*fileConfig()* 进行处理的文件来发送。返回一个*Thread* 实例，你可以在该实例上调用*start()* 来启动服务器，对该服务器你可以在适当的时候执行*join()*。要停止该服务器，请调用*stopListening()*。

如果指定 verify 参数，则它应当是一个可调用对象，该对象应当验证通过套接字接收的字节数据是否有效且应被处理。这可以通过对通过套接字发送的内容进行加密和/或签名来完成，这样 verify 可调用对象就能执行签名验证和/或解密。verify 可调用对象的调用会附带一个参数——通过套接字接收的字节数据——并应当返回要处理的字节数据，或者返回 None 来指明这些字节数据应当被丢弃。返回的字节数据可以与传入的字节数据相同（例如在只执行验证的时候），或者也可以完全不同（例如在可能执行了解密的时候）。

要将配置发送到套接字，请读取配置文件并将其作为字节序列发送到套接字，字节序列要以使用 struct.pack('>L', n) 打包为二进制格式的四字节长度的字符串打头。

備𝔽: Because portions of the configuration are passed through *eval()*, use of this function may open its users to a security risk. While the function only binds to a socket on localhost, and so does not accept connections from remote machines, there are scenarios where untrusted code could be run under the account of the process which calls *listen()*. Specifically, if the process calling *listen()* runs on a multi-user machine where users cannot trust each other, then a malicious user could arrange to run essentially arbitrary code in a victim user's process, simply by connecting to the victim's *listen()* socket and sending a configuration which runs whatever code the attacker wants to have executed in the victim's process. This is especially easy to do if the default port is used, but not hard even if a different port is used. To avoid the risk of this happening, use the verify argument to *listen()* to prevent unrecognised configurations from being applied.

3.4 版更變: 添加了 verify 参数。

備F: 如果你希望将配置发送给未禁用现有日志记录器的监听器，你将需要使用 JSON 格式的配置，该格式将使用*dictConfig()* 进行配置。此方法允许你在你发送的配置中将 disable_existing_loggers 指定为 False。

---

logging.config.**stopListening**()

停止通过对*listen()* 的调用所创建的监听服务器。此函数的调用通常会先于在*listen()* 的返回值上调用 join()。

16.7.2 Security considerations

The logging configuration functionality tries to offer convenience, and in part this is done by offering the ability to convert text in configuration files into Python objects used in logging configuration - for example, as described in 用户定义对象. However, these same mechanisms (importing callables from user-defined modules and calling them with parameters from the configuration) could be used to invoke any code you like, and for this reason you should treat configuration files from untrusted sources with *extreme caution* and satisfy yourself that nothing bad can happen if you load them, before actually loading them.

16.7.3 配置字典架构

描述日志记录配置需要列出要创建的不同对象及它们之间的连接；例如，你可以创建一个名为'console' 的处理程序，然后名为'startup' 的日志记录器将可以把它的消息发送给'console' 处理程序。这些对象并不仅限于*logging* 模块所提供的对象，因为你还可以编写你自己的格式化或处理程序类。这些类的形参可能还需要包括 sys.stderr 这样的外部对象。描述这些对象和连接的语法会在下面的对象连接 中定义。

字典架构细节

传给*dictConfig()* 的字典必须包含以下的键:

- *version* - 应设为代表架构版本的整数值。目前唯一有效的值是 1，使用此键可允许架构在继续演化的同时保持向下兼容性。

所有其他键都是可选项，但如存在它们将根据下面的描述来解读。在下面提到'configuring dict' 的所有情况下，都将检查它的特殊键 '()' 以确定是否需要自定义实例化。如果需要，则会使用下面用户定义对象 所描述的机制来创建一个实例；否则，会使用上下文来确定要实例化的对象。

- *formatters* - 对应的值将是一个字典，其中每个键是一个格式器 ID 而每个值则是一个描述如何配置相应*Formatter* 实例的字典。

  将在配置字典中搜索键 format 和 datefmt (默认值均为 None) 并且这些键会被用于构造*Formatter* 实例。

  3.8 版更變: 一个 validate 键 (默认值为 True) 可被添加到配置字典的 formatters 部分，这会被用来验证格式的有效性。

- *filters* - 对应的值将是一个字典，其中每个键是一个过滤器 ID 而每个值则是一个描述如何配置相应 Filter 实例的字典。

  将在配置字典中搜索键 name (默认值为空字符串) 并且该键会被用于构造*logging.Filter* 实例。

- *handlers* - 对应的值将是一个字典，其中每个键是一个处理程序 ID 而每个值则是一个描述如何配置相应 Handler 实例的字典。

  将在配置字典中搜索下列键:

  - `class` (强制)。这是处理程序类的完整限定名称。
  - `level` (可选)。处理程序的级别。
  - `formatter` (可选)。处理程序所对应格式化器的 ID。
  - `filters` (可选)。由处理程序所对应过滤器的 ID 组成的列表。

  所有 *其他*键会被作为关键字参数传递给处理程序类的构造器。例如，给定如下配置:

```
handlers:
  console:
    class : logging.StreamHandler
    formatter: brief
    level   : INFO
    filters: [allow_foo]
    stream  : ext://sys.stdout
  file:
    class : logging.handlers.RotatingFileHandler
    formatter: precise
    filename: logconfig.log
    maxBytes: 1024
    backupCount: 3
```

  ID 为 `console` 的处理程序会被实例化为*`logging.StreamHandler`*，并使用 `sys.stdout` 作为下层流。ID 为 `file` 的处理程序会被实例化为*`logging.handlers.RotatingFileHandler`*，并附带关键字参数 `filename='logconfig.log', maxBytes=1024, backupCount=3`。

- *loggers* - 对应的值将是一个字典，其中每个键是一个日志记录器名称而每个值则是一个描述如何配置相应 Logger 实例的字典。

  将在配置字典中搜索下列键:

  - `level` (可选)。日志记录器的级别。
  - `propagate` (可选)。日志记录器的传播设置。
  - `filters` (可选)。由日志记录器对应过滤器的 ID 组成的列表。
  - `handlers` (可选)。由日志记录器对应处理程序的 ID 组成的列表。

  指定的日志记录器将根据指定的级别、传播、过滤器和处理程序来配置。

- *root* - 这将成为根日志记录器对应的配置。配置的处理方式将与所有日志记录器一致，除了 `propagate` 设置将不可用之外。

- *incremental* - 配置是否要被解读为在现有配置上新增。该值默认为 `False`，这意味着指定的配置将以与当前*`fileConfig()`* API 所使用的相同语义来替代现有的配置。

  如果指定的值为 `True`，配置会按照增量配置 部分所描述的方式来处理。

- *disable_existing_loggers* - 是否要禁用任何现有的非根日志记录器。该设置对应于*`fileConfig()`* 中的同名形参。如果省略，则此形参默认为 `True`。如果 *incremental* 为 `True` 则该省会被忽略。

增量配置

为增量配置提供完全的灵活性是很困难的。例如，由于过滤器和格式化器这样的对象是匿名的，一旦完成配置，在增加配置时就不可能引用这些匿名对象。

此外，一旦完成了配置，在运行时任意改变日志记录器、处理程序、过滤器、格式化器的对象图就不是很有必要；日志记录器和处理程序的详细程度只需通过设置级别即可实现控制（对于日志记录器则可设置传播旗标）。在多线程环境中以安全的方式任意改变对象图也许会导致问题；虽然并非不可能，但这样做的好处不足以抵销其所增加的实现复杂度。

这样，当配置字典的 `incremental` 键存在且为 `True` 时，系统将完全忽略任何 `formatters` 和 `filters` 条目，并仅会处理 `handlers` 条目中的 `level` 设置，以及 `loggers` 和 `root` 条目中的 `level` 和 `propagate` 设置。

使用配置字典中的值可让配置以封存字典对象的形式通过线路传送给套接字监听器。这样，长时间运行的应用程序的日志记录的详细程度可随时间改变而无须停止并重新启动应用程序。

对象连接

该架构描述了一组日志记录对象——日志记录器、处理程序、格式化器、过滤器——它们在对象图中彼此连接。因此，该架构需要能表示对象之间的连接。例如，在配置完成后，一个特定的日志记录器关联到了一个特定的处理程序。出于讨论的目的，我们可以说该日志记录器代表两者间连接的源头，而处理程序则代表对应的目标。当然在已配置对象中这是由包含对处理程序的引用的日志记录器来代表的。在配置字典中，这是通过给每个目标对象一个 ID 来无歧义地标识它，然后在源头对象中使用该 ID 来实现的。

因此，举例来说，考虑以下 YAML 代码段:

```
formatters:
  brief:
    # configuration for formatter with id 'brief' goes here
  precise:
    # configuration for formatter with id 'precise' goes here
handlers:
  h1: #This is an id
   # configuration of handler with id 'h1' goes here
   formatter: brief
  h2: #This is another id
   # configuration of handler with id 'h2' goes here
   formatter: precise
loggers:
  foo.bar.baz:
    # other configuration for logger 'foo.bar.baz'
    handlers: [h1, h2]
```

（注：这里使用 YAML 是因为它的可读性比表示字典的等价 Python 源码形式更好。）

日志记录器 ID 就是日志记录器的名称，它会在程序中被用来获取对日志记录器的引用，例如 `foo.bar.baz`。格式化器和过滤器的 ID 可以是任意字符串值 (例如上面的 `brief`, `precise`) 并且它们是瞬态的，因为它们仅对处理配置字典有意义并会被用来确定对象之间的连接，而当配置调用完成时不会在任何地方保留。

上面的代码片段指明名为 `foo.bar.baz` 的日志记录器应当关联到两个处理程序，它们的 ID 是 `h1` 和 `h2`。`h1` 的格式化器的 ID 是 `brief`，而 `h2` 的格式化器的 ID 是 `precise`。

用户定义对象

此架构支持用户定义对象作为处理程序、过滤器和格式化器。（日志记录器的不同实例不需要具有不同类型，因此这个配置架构并不支持用户定义日志记录器类。）

要配置的对象是由字典描述的，其中包含它们的配置详情。在某些地方，日志记录系统将能够从上下文中推断出如何实例化一个对象，但是当要实例化一个用户自定义对象时，系统将不知道要如何做。为了提供用户自定义对象实例化的完全灵活性，用户需要提供一个‘工厂’函数——即在调用时传入配置字典并返回实例化对象的可调用对象。这是用一个通过特殊键 `'()'` 来访问的工厂函数的绝对导入路径来标示的。下面是一个实际的例子:

```
formatters:
  brief:
    format: '%(message)s'
  default:
    format: '%(asctime)s %(levelname)-8s %(name)-15s %(message)s'
    datefmt: '%Y-%m-%d %H:%M:%S'
  custom:
      (): my.package.customFormatterFactory
      bar: baz
      spam: 99.9
      answer: 42
```

上面的 YAML 代码片段定义了三个格式化器。第一个的 ID 为 `brief`，是带有特殊格式字符串的标准*logging.Formatter* 实例。第二个的 ID 为 `default`，具有更长的格式同时还显式地定义了时间格式，并将最终实例化一个带有这两个格式字符串的*logging.Formatter*。以 Python 源代码形式显示的 `brief` 和 `default` 格式化器分别具有下列配置子字典:

```
{
  'format' : '%(message)s'
}
```

和:

```
{
  'format' : '%(asctime)s %(levelname)-8s %(name)-15s %(message)s',
  'datefmt' : '%Y-%m-%d %H:%M:%S'
}
```

并且由于这些字典不包含特殊键 `'()'`，实例化方式是从上下文中推断出来的：结果会创建标准的*logging.Formatter* 实例。第三个格式器的 ID 为 `custom`，对应配置子字典为:

```
{
  '()' : 'my.package.customFormatterFactory',
  'bar' : 'baz',
  'spam' : 99.9,
  'answer' : 42
}
```

并且它包含特殊键 `'()'`，这意味着需要用户自定义实例化方式。在此情况下，将使用指定的工厂可调用对象。如果它本身就是一个可调用对象则将被直接使用——否则如果你指定了一个字符串（如这个例子所示）则将使用正常的导入机制来定位实例的可调用对象。调用该可调用对象将传入配置子字典中 **剩余的**条目作为关键字参数。在上面的例子中，调用将预期返回 ID 为 `custom` 的格式化器:

```
my.package.customFormatterFactory(bar='baz', spam=99.9, answer=42)
```

将 `'()'` 用作特殊键是因为它不是一个有效的关键字形参名称，这样就不会与调用中使用的关键字参数发生冲突。`'()'` 还被用作表明对应值为可调用对象的助记符。

访问外部对象

有时一个配置需要引用配置以外的对象，例如 sys.stderr。如果配置字典是使用 Python 代码构造的，这会很直观，但是当配置是通过文本文件（例如 JSON, YAML）提供的时候就会引发问题。在一个文本文件中，没有将 sys.stderr 与字符串字面值 'sys.stderr' 区分开来的标准方式。为了实现这种区分，配置系统会在字符串值中查找规定的特殊前缀并对其做特殊处理。例如，如果在配置中将字符串字面值 'ext://sys.stderr' 作为一个值来提供，则 ext:// 将被去除而该值的剩余部分将使用正常导入机制来处理。

此类前缀的处理方式类似于协议处理：存在一种通用机制来查找与正则表达式 ^(?P<prefix>[a-z]+)://(?P<suffix>.*)$ 相匹配的前缀，如果识别出了 prefix，则 suffix 会以与前缀相对应的方式来处理并且处理的结果将替代原字符串值。如果未识别出前缀，则原字符串将保持不变。

访问内部对象

除了外部对象，有时还需要引用配置中的对象。这将由配置系统针对它所了解的内容隐式地完成。例如，在日志记录器或处理程序中表示 level 的字符串值 'DEBUG' 将被自动转换为值 logging.DEBUG，而 handlers, filters 和 formatter 条目将接受一个对象 ID 并解析为适当的目标对象。

但是，对于*logging* 模块所不了解的用户自定义对象则需要一种更通用的机制。例如，考虑*logging.handlers.MemoryHandler*，它接受一个 target 参数即其所委托的另一个处理程序。由于系统已经知道存在该类，因而在配置中，给定的 target 只需为相应目标处理程序的的对象 ID 即可，而系统将根据该 ID 解析出处理程序。但是，如果用户定义了一个具有 alternate 处理程序的 my.package.MyHandler，则配置程序将不知道 alternate 指向的是一个处理程序。为了应对这种情况，通用解析系统允许用户指定:

```
handlers:
  file:
    # configuration of file handler goes here

  custom:
    (): my.package.MyHandler
    alternate: cfg://handlers.file
```

字符串字面值 'cfg://handlers.file' 将按照与 ext:// 前缀类似的方式被解析为结果字符串，但查找操作是在配置自身而不是在导入命名空间中进行。该机制允许按点号或按索引来访问，与 str.format 所提供的方式类似。这样，给定以下代码段:

```
handlers:
  email:
    class: logging.handlers.SMTPHandler
    mailhost: localhost
    fromaddr: my_app@domain.tld
    toaddrs:
      - support_team@domain.tld
      - dev_team@domain.tld
    subject: Houston, we have a problem.
```

在该配置中，字符串 'cfg://handlers' 将解析为带有 handlers 键的字典，字符串 'cfg://handlers.email 将解析为 handlers 字典中带有 email 键的字典，依此类推。字符串 'cfg://handlers.email.toaddrs[1] 将解析为 'dev_team.domain.tld' 而字符串 'cfg://handlers.email.toaddrs[0]' 将解析为值 'support_team@domain.tld'。subject 值可以使用 'cfg://handlers.email.subject' 或者等价的 'cfg://handlers.email[subject]' 来访问。后一种形式仅在键包含空格或非字母数字类字符的情况下才需要使用。如果一个索引仅由十进制数码构成，则将尝试使用相应的整数值来访问，如果有必要则将回退为字符串值。

给定字符串 cfg://handlers.myhandler.mykey.123，这将解析为

config_dict['handlers']['myhandler']['mykey']['123']。 如 果 字 符 串 被 指 定 为 cfg://handlers.myhandler.mykey[123]， 系 统 将 尝 试 从 config_dict['handlers']['myhandler']['mykey'][123] 中提取值，并在尝试失败时回退为 config_dict['handlers']['myhandler']['mykey']['123']。

导入解析与定制导入器

导入解析默认使用内置的*__import__()* 函数来执行导入。你可能想要将其替换为你自己的导入机制：如果是这样的话，你可以替换 DictConfigurator 或其超类 BaseConfigurator 类的 importer 属性。但是你必须小心谨慎，因为函数是从类中通过描述器方式来访问的。如果你使用 Python 可调用对象来执行导入，并且你希望在类层级而不是在实例层级上定义它，则你需要用*staticmethod()* 来装饰它。例如:

```
from importlib import import_module
from logging.config import BaseConfigurator

BaseConfigurator.importer = staticmethod(import_module)
```

如果你是在一个配置器的 实例上设置导入可调用对象则你不需要用*staticmethod()* 来装饰。

16.7.4 配置文件格式

fileConfig() 所能理解的配置文件格式是基于*configparser* 功能的。该文件必须包含 [loggers]，[handlers] 和 [formatters] 等小节，它们通过名称来标识文件中定义的每种类型的实体。对于每个这样的实体，都有单独的小节来标识实体的配置方式。因此，对于 [loggers] 小节中名为 log01 的日志记录器，相应的配置详情保存在 [logger_log01] 小节中。类似地，对于 [handlers] 小节中名为 hand01 的处理程序，其配置将保存在名为 [handler_hand01] 的小节中，而对于 [formatters] 小节中名为 form01 的格式化器，其配置将在名为 [formatter_form01] 的小节中指定。根日志记录器的配置必须在名为 [logger_root] 的小节中指定。

備𝔽: *fileConfig()* API 比*dictConfig()* API 更旧因而没有提供涵盖日志记录特定方面的功能。例如，你无法配置*Filter* 对象，该对象使用*fileConfig()* 提供超出简单整数级别的消息过滤功能。如果你想要在你的日志记录配置中包含*Filter* 的实例，你将必须使用*dictConfig()*。请注意未来还将向*dictConfig()* 添加对配置功能的强化，因此值得考虑在方便的时候转换到这个新 API。

在文件中这些小节的例子如下所示。

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

根日志记录器必须指定一个级别和一个处理程序列表。根日志小节的例子如下所示。

```
[logger_root]
level=NOTSET
handlers=hand01
```

level 条目可以为 DEBUG, INFO, WARNING, ERROR, CRITICAL 或 NOTSET 之一。其中 NOTSET 仅适用于根日志记录器，表示将会记录所有消息。级别值会在 logging 包命名空间的上下文中通过*eval()* 来得出。

handlers 条目是以逗号分隔的处理程序名称列表，它必须出现于 [handlers] 小节并且在配置文件中有相应的小节。

对于根日志记录器以外的日志记录器，还需要某些附加信息。下面的例子演示了这些信息。

```
[logger_parser]
level=DEBUG
handlers=hand01
propagate=1
qualname=compiler.parser
```

level 和 handlers 条目的解释方式与根日志记录器的一致，不同之处在于如果一个非根日志记录器的级别被指定为 NOTSET，则系统会咨询更高层级的日志记录器来确定该日志记录器的有效级别。propagate 条目设为 1 表示消息必须从此日志记录器传播到更高层级的处理程序，设为 0 表示消息 **不会**传播到更高层级的处理程序。qualname 条目是日志记录器的层级通道名称，也就是应用程序获取日志记录器所用的名称。

指定处理程序配置的小节说明如下。

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

class 条目指明处理程序的类（由 logging 包命名空间中的*eval()* 来确定）。level 会以与日志记录器相同的方式来解读，NOTSET 会被视为表示‘记录一切消息’。

formatter 条目指明此处理程序的格式化器的键名称。如为空白，则会使用默认的格式化器 (logging._defaultFormatter)。如果指定了名称，则它必须出现于 [formatters] 小节并且在配置文件中有相应的小节。

args 条目，当在 logging 包命名空间的上下文中执行*eval()* 时将是传给处理程序类构造器的参数列表。请参阅相应处理程序的构造器说明或者下面的示例，以了解典型的条目是如何构造的。如果未提供，则默认为 ()。

可选的 kwargs 条目，当在 logging 包命名空间的上下文中执行*eval()* 时将是传给处理程序的构造器的关键字参数字典。如果未提供，则默认为 {}。

```
[handler_hand02]
class=FileHandler
level=DEBUG
formatter=form02
args=('python.log', 'w')

[handler_hand03]
class=handlers.SocketHandler
level=INFO
formatter=form03
args=('localhost', handlers.DEFAULT_TCP_LOGGING_PORT)

[handler_hand04]
class=handlers.DatagramHandler
level=WARN
formatter=form04
args=('localhost', handlers.DEFAULT_UDP_LOGGING_PORT)
```

(下页继续)

(繼續上一頁)

```
[handler_hand05]
class=handlers.SysLogHandler
level=ERROR
formatter=form05
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)

[handler_hand06]
class=handlers.NTEventLogHandler
level=CRITICAL
formatter=form06
args=('Python Application', '', 'Application')

[handler_hand07]
class=handlers.SMTPHandler
level=WARN
formatter=form07
args=('localhost', 'from@abc', ['user1@abc', 'user2@xyz'], 'Logger Subject')
kwargs={'timeout': 10.0}

[handler_hand08]
class=handlers.MemoryHandler
level=NOTSET
formatter=form08
target=
args=(10, ERROR)

[handler_hand09]
class=handlers.HTTPHandler
level=NOTSET
formatter=form09
args=('localhost:9022', '/log', 'GET')
kwargs={'secure': True}
```

指定格式化器配置的小节说明如下。

```
[formatter_form01]
format=F1 %(asctime)s %(levelname)s %(message)s
datefmt=
class=logging.Formatter
```

`format` 是整个格式字符串，而 `datefmt` 条目则是兼容 `strftime()` 的日期/时间格式字符串。如果为空，此包将替换任何接近于日期格式字符串 `'%Y-%m-%d %H:%M:%S'` 的内容。此格式还指定了毫秒，并使用逗号分隔符将其附加到结果当中。此格式的时间示例如 `2003-01-23 00:29:50,411`。

`class` 条目是可选的。它指明格式化器类的名称（形式为带点号的模块名加类名。）此选项适用于实例化 *Formatter* 的子类。*Formatter* 的子类可通过扩展或收缩格式来显示异常回溯信息。

備𝔼: 由于如上所述使用了 *eval()*，因此使用 *listen()* 通过套接字来发送和接收配置会导致潜在的安全风险。此风险仅限于相互间没有信任的多个用户在同一台机器上运行代码的情况；请参阅 *listen()* 了解更多信息。

也参考:

模块 *logging* 日志记录模块的 API 参考。

模块*logging.handlers* 日志记录模块附带的有用处理程序。

16.8 logging.handlers --- 日志处理程序

源代码: Lib/logging/handlers.py

Important

此页面仅包含参考信息。有关教程，请参阅

- 基础教程
- 进阶教程
- 日志记录操作手册

这个包提供了以下有用的处理程序。请注意有三个处理程序类 (*StreamHandler*, *FileHandler* 和*NullHandler*) 实际上是在*logging* 模块本身定义的，但其文档与其他处理程序一同记录在此。

16.8.1 StreamHandler

StreamHandler 类位于核心*logging* 包，它可将日志记录输出发送到数据流例如 *sys.stdout*, *sys.stderr* 或任何文件类对象（或者更精确地说，任何支持 `write()` 和 `flush()` 方法的对象）。

class logging.**StreamHandler**(*stream=None*)

返回一个新的*StreamHandler* 类。如果指定了 *stream*，则实例将用它作为日志记录输出；在其他情况下将使用 *sys.stderr*。

emit(*record*)

如果指定了一个格式化器，它会被用来格式化记录。随后记录会被写入到*terminator* 之后的流中。如果存在异常信息，则会使用*traceback.print_exception()* 来格式化并添加到流中。

flush()

通过调用流的*flush()* 方法来刷新它。请注意 `close()` 方法是继承自*Handler* 的所以没有输出，因此有时可能需要显式地调用*flush()*。

setStream(*stream*)

将实例的流设为指定值，如果两者不一致的话。旧的流会在设置新的流之前被刷新。

参数 **stream** -- 处理程序应当使用的流。

傳回 旧的流，如果流已被改变的话，如果未被改变则为 *None*。

3.7 版新加入.

terminator

当将已格式化的记录写入到流时被用作终止符的字符串。默认值为 `'\n'`。

如果你不希望以换行符终止，你可以将处理程序类实例的 `terminator` 属性设为空字符串。

在较早的版本中，终止符被硬编码为 `'\n'`。

3.2 版新加入.

16.8.2 FileHandler

FileHandler 类位于核心*logging* 包，它可将日志记录输出到磁盘文件中。它从*StreamHandler* 继承了输出功能。

class logging.**FileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

返回一个*FileHandler* 类的新实例。将打开指定的文件并将其用作日志记录流。如果未指定 *mode*，则会使用 `'a'`。如果 *encoding* 不为 `None`，则会将其用作打开文件的编码格式。如果 *delay* 为真值，则文件打开会被推迟至第一次调用*emit()* 时。默认情况下，文件会无限增长。如果指定了 *errors*，它会被用于确定编码格式错误的处理方式。

3.6 版更變: 除了字符串值，也接受*Path* 对象作为 *filename* 参数。

3.9 版更變: 增加了 *errors* 形参。

close()

关闭文件。

emit(*record*)

将记录输出到文件。

16.8.3 NullHandler

3.1 版新加入.

NullHandler 类位于核心*logging* 包，它不执行任何格式化或输出。它实际上是一个供库开发者使用的‘无操作’处理程序。

class logging.**NullHandler**

返回一个*NullHandler* 类的新实例。

emit(*record*)

此方法不执行任何操作。

handle(*record*)

此方法不执行任何操作。

createLock()

此方法会对锁返回 `None`，因为没有下层 I/O 的访问需要被序列化。

请参阅 library-config 了解有关如何使用*NullHandler* 的更多信息。

16.8.4 WatchedFileHandler

WatchedFileHandler 类位于*logging.handlers* 模块，这个 `FileHandler` 用于监视它所写入日志记录的文件。如果文件发生变化，它会被关闭并使用文件名重新打开。

发生文件更改可能是由于使用了执行文件轮换的程序例如 *newsyslog* 和 *logrotate*。这个处理程序在 Unix/Linux 下使用，它会监视文件来查看自上次发出数据后是否有更改。(如果文件的设备或 inode 发生变化就认为已被更改。) 如果文件被更改，则会关闭旧文件流，并再打开文件以获得新文件流。

这个处理程序不适合在 Windows 下使用，因为在 Windows 下打开的日志文件无法被移动或改名——日志记录会使用排他的锁来打开文件——因此这样的处理程序是没有必要的。此外，*ST_INO* 在 Windows 下不受支持；*stat()* 将总是为该值返回零。

class logging.handlers.**WatchedFileHandler**(*filename*, *mode='a'*, *encoding=None*, *delay=False*, *errors=None*)

返回一个*WatchedFileHandler* 类的新实例。将打开指定的文件并将其用作日志记录流。如果未指定 *mode*，则会使用 `'a'`。如果 *encoding* 不为 `None`，则会将其用作打开文件的编码格式。如果 *delay*

为真值，则文件打开会被推迟至第一次调用*emit()* 时。默认情况下，文件会无限增长。如果提供了 *errors*，它会被用于确定编码格式错误的处理方式。

3.6 版更變: 除了字符串值，也接受*Path* 对象作为 *filename* 参数。

3.9 版更變: 增加了 *errors* 形参。

reopenIfNeeded()
检查文件是否已更改。如果已更改，则会刷新并关闭现有流然后重新打开文件，这通常是将记录输出到文件的先导操作。

3.6 版新加入.

emit(*record*)
将记录输出到文件，但如果文件已更改则会先调用*reopenIfNeeded()* 来重新打开它。

16.8.5 BaseRotatingHandler

BaseRotatingHandler 类位于*logging.handlers* 模块中，它是轮换文件处理程序类*RotatingFileHandler* 和*TimedRotatingFileHandler* 的基类。你不需要实例化此类，但它具有你可能需要重载的属性和方法。

class logging.handlers.**BaseRotatingHandler**(*filename*, *mode*, *encoding=None*, *delay=False*, *errors=None*)
类的形参与 FileHandler 的相同。其属性有:

namer
如果此属性被设为一个可调用对象，则*rotation_filename()* 方法会委托给该可调用对象。传给该可调用对象的形参与传给*rotation_filename()* 的相同。

備F: namer 函数会在轮换期间被多次调用，因此它应当尽可能的简单快速。它还应当对给定的输入每次都返回相同的输出，否则轮换行为可能无法按预期工作。

It's also worth noting that care should be taken when using a namer to preserve certain attributes in the filename which are used during rotation. For example, *RotatingFileHandler* expects to have a set of log files whose names contain successive integers, so that rotation works as expected, and *TimedRotatingFileHandler* deletes old log files (based on the backupCount parameter passed to the handler's initializer) by determining the oldest files to delete. For this to happen, the filenames should be sortable using the date/time portion of the filename, and a namer needs to respect this. (If a namer is wanted that doesn't respect this scheme, it will need to be used in a subclass of *TimedRotatingFileHandler* which overrides the *getFilesToDelete()* method to fit in with the custom naming scheme.)

3.3 版新加入.

rotator
如果此属性被设为一个可调用对象，则*rotate()* 方法会委托给该可调用对象。传给该可调用对象的形参与传给*rotate()* 的相同。

3.3 版新加入.

rotation_filename(*default_name*)
当轮换时修改日志文件的文件名。

提供该属性以便可以提供自定义文件名。

默认实现会调用处理程序的'namer' 属性，如果它是可调用对象的话，并传给它默认的名称。如果该属性不是可调用对象 (默认值为 None)，则将名称原样返回。

参数 default_name -- 日志文件的默认名称。

3.3 版新加入.

rotate(*source*, *dest*)
当执行轮换时，轮换当前日志。

默认实现会调用处理程序的'rotator' 属性，如果它是可调用对象的话，并传给它 source 和 dest 参数。如果该属性不是可调用对象 (默认值为 None)，则将源简单地重命名为目标。

参数

- **source** -- 源文件名。这通常为基本文件名，例如'test.log'。
- **dest** -- 目标文件名。这通常是源被轮换后的名称，例如'test.log.1'。

3.3 版新加入.

该属性存在的理由是让你不必进行子类化——你可以使用与*RotatingFileHandler*和*TimedRotatingFileHandler*的实例相同的可调用对象。如果 namer 或 rotator 可调用对象引发了异常，将会按照与 emit() 调用期间的任何其他异常相同的方式来处理，例如通过处理程序的 handleError() 方法。

如果你需要对轮换进程执行更多的修改，你可以重载这些方法。

请参阅 cookbook-rotator-namer 获取具体示例。

16.8.6 RotatingFileHandler

*RotatingFileHandler*类位于*logging.handlers*模块，它支持磁盘日志文件的轮换。

class logging.handlers.**RotatingFileHandler**(*filename*, *mode='a'*, *maxBytes=0*, *backupCount=0*, *encoding=None*, *delay=False*, *errors=None*)
返回一个*RotatingFileHandler*类的新实例。将打开指定的文件并将其用作日志记录流。如果未指定 *mode*，则会使用 'a'。如果 *encoding* 不为 None，则会将其用作打开文件的编码格式。如果 *delay* 为真值，则文件打开会被推迟至第一次调用*emit()*。默认情况下，文件会无限增长。如果提供了 *errors*，它会被用于确定编码格式错误的处理方式。

你可以使用 *maxBytes* 和 *backupCount* 值来允许文件以预定的大小执行 *rollover*。当即将超出预定大小时，将关闭旧文件并打开一个新文件用于输出。只要当前日志文件长度接近 *maxBytes* 就会发生轮换；但是如果 *maxBytes* 或 *backupCount* 两者之一的值为零，就不会发生轮换，因此你通常要设置 *backupCount* 至少为 1，而 *maxBytes* 不能为零。当 *backupCount* 为非零值时，系统将通过为原文件名添加扩展名'.1', '.2' 等来保存旧日志文件。例如，当 *backupCount* 为 5 而基本文件名为 app.log 时，你将得到 app.log, app.log.1, app.log.2 直至 app.log.5。当前被写入的文件总是 app.log。当此文件写满时，它会被关闭并重户名为 app.log.1，而如果文件 app.log.1, app.log.2 等存在，则它们会被分别重命名为 app.log.2, app.log.3 等等。

3.6 版更變: 除了字符串值，也接受*Path*对象作为 *filename* 参数。

3.9 版更變: 增加了 *errors* 形参。

doRollover()
执行上文所描述的轮换。

emit(*record*)
将记录输出到文件，以适应上文所描述的轮换。

16.8.7 TimedRotatingFileHandler

TimedRotatingFileHandler 类位于*logging.handlers* 模块，它支持基于特定时间间隔的磁盘日志文件轮换。

class logging.handlers.**TimedRotatingFileHandler**(*filename*, *when='h'*, *interval=1*, *backupCount=0*, *encoding=None*, *delay=False*, *utc=False*, *atTime=None*, *errors=None*)

返回一个新的*TimedRotatingFileHandler* 类实例。指定的文件会被打开并用作日志记录的流。对于轮换操作它还会设置文件名前缀。轮换的发生是基于 *when* 和 *interval* 的积。

你可以使用 *when* 来指定 *interval* 的类型。可能的值列表如下。请注意它们不是大小写敏感的。

值	间隔类型	如果/如何使用 *atTime*
`'S'`	秒	忽略
`'M'`	分钟	忽略
`'H'`	小时	忽略
`'D'`	天	忽略
`'W0'-'W6'`	工作日 (0= 星期一)	用于计算初始轮换时间
`'midnight'`	如果未指定 *atTime* 则在午夜执行轮换，否则将使用 *atTime*。	用于计算初始轮换时间

当使用基于星期的轮换时，星期一为'W0'，星期二为'W1'，以此类推直至星期日为'W6'。在这种情况下，传入的 *interval* 值不会被使用。

系统将通过为文件名添加扩展名来保存旧日志文件。扩展名是基于日期和时间的，根据轮换间隔的长短使用 strftime 格式 `%Y-%m-%d_%H-%M-%S` 或是其中有变动的部分。

当首次计算下次轮换的时间时（即当处理程序被创建时），现有日志文件的上次被修改时间或者当前时间会被用来计算下次轮换的发生时间。

如果 *utc* 参数为真值，将使用 UTC 时间；否则会使用本地时间。

如果 *backupCount* 不为零，则最多将保留 *backupCount* 个文件，而如果当轮换发生时创建了更多的文件，则最旧的文件会被删除。删除逻辑使用间隔时间来确定要删除的文件，因此改变间隔时间可能导致旧文件被继续保留。

如果 *delay* 为真值，则会将文件打开延迟到首次调用*emit()* 的时候。

如果 *atTime* 不为 `None`，则它必须是一个 `datetime.time` 的实例，该实例指定轮换在一天内的发生时间，用于轮换被设为“在午夜”或“在每星期的某一天”之类的情况。请注意在这些情况下，*atTime* 值实际上会被用于计算*初始*轮换，而后续轮换将会通过正常的间隔时间计算来得出。

如果指定了 *errors*，它会被用来确定编码错误的处理方式。

備𝔽: 初始轮换时间的计算是在处理程序被初始化时执行的。后续轮换时间的计算则仅在轮换发生时执行，而只有当提交输出时轮换才会发生。如果不记住这一点，你就可能会感到困惑。例如，如果设置时间间隔为“每分钟”，那并不意味着你总会看到（文件名中）带有间隔一分钟时间的日志文件；如果在应用程序执行期间，日志记录输出的生成频率高于每分钟一次，那么你可以预期看到间隔一分钟时间的日志文件。另一方面，如果（假设）日志记录消息每五分钟才输出一次，那么文件时间将会存在对应于没有输出（因而没有轮换）的缺失。

3.4 版更變: 添加了 *atTime* 形参。

3.6 版更變: 除了字符串值，也接受*Path* 对象作为 *filename* 参数。

3.9 版更變: 增加了 *errors* 形参。

doRollover()
执行上文所描述的轮换。

emit(*record*)
将记录输出到文件，以适应上文所描述的轮换。

getFilesToDelete()
Returns a list of filenames which should be deleted as part of rollover. These are the absolute paths of the oldest backup log files written by the handler.

16.8.8 SocketHandler

SocketHandler 类位于*logging.handlers* 模块，它会将日志记录输出发送到网络套接字。基类所使用的是 TCP 套接字。

class logging.handlers.**SocketHandler**(*host*, *port*)
返回一个*SocketHandler* 类的新实例，该实例旨在与使用 *host* 与 *port* 给定地址的远程主机进行通信。

3.4 版更變: 如果 `port` 指定为 `None`，会使用 `host` 中的值来创建一个 Unix 域套接字——在其他情况下，则会创建一个 TCP 套接字。

close()
关闭套接字。

emit()
对记录的属性字典执行封存并以二进制格式将其写入套接字。如果套接字存在错误，则静默地丢弃数据包。如果连接在此之前丢失，则重新建立连接。要在接收端将记录解封并输出到*LogRecord*，请使用*makeLogRecord()* 函数。

handleError()
处理在*emit()* 期间发生的错误。最可能的原因是连接丢失。关闭套接字以便我们能在下次事件时重新尝试。

makeSocket()
这是一个工厂方法，它允许子类定义它们想要的套接字的准确类型。默认实现会创建一个 TCP 套接字 (*socket.SOCK_STREAM*)。

makePickle(*record*)
将记录的属性字典封存为带有长度前缀的二进制格式，并将其返回以准备通过套接字进行传输。此操作在细节上相当于:

```
data = pickle.dumps(record_attr_dict, 1)
datalen = struct.pack('>L', len(data))
return datalen + data
```

请注意封存操作不是绝对安全的。如果你关心安全问题，你可能会想要重载此方法以实现更安全的机制。例如，你可以使用 HMAC 对封存对象进行签名然后在接收端验证它们，或者你也可以在接收端禁用全局对象的解封操作。

send(*packet*)
将封存后的字节串 *packet* 发送到套接字。所发送字节串的格式与*makePickle()* 文档中的描述一致。

此函数允许部分发送，这可能会在网络繁忙时发生。

createSocket()
尝试创建一个套接字；失败时将使用指数化回退算法处理。在失败初次发生时，处理程序将丢弃它正尝试发送的消息。当后续消息交由同一实例处理时，它将不会尝试连接直到经过一段时间以

后。默认形参设置为初始延迟一秒，如果在延迟之后连接仍然无法建立，处理程序将每次把延迟翻倍直至达到 30 秒的最大值。

此行为由下列处理程序属性控制:

- `retryStart` (初始延迟，默认为 1.0 秒)。
- `retryFactor` (倍数，默认为 2.0)。
- `retryMax` (最大延迟，默认为 30.0 秒)。

这意味着如果远程监听器在处理程序被使用 之后启动，你可能会丢失消息（因为处理程序在延迟结束之前甚至不会尝试连接，而在延迟期间静默地丢弃消息）。

16.8.9 DatagramHandler

DatagramHandler 类位于*logging.handlers* 模块，它继承自*SocketHandler*，支持通过 UDP 套接字发送日志记录消息。

class logging.handlers.**DatagramHandler**(*host*, *port*)

返回一个*DatagramHandler* 类的新实例，该实例旨在与使用 *host* 与 *port* 给定地址的远程主机进行通信。

3.4 版更變: 如果 `port` 指定为 `None`，会使用 `host` 中的值来创建一个 Unix 域套接字——在其他情况下，则会创建一个 UDP 套接字。

emit()

对记录的属性字典执行封存并以二进制格式将其写入套接字。如果套接字存在错误，则静默地丢弃数据包。要在接收端将记录解封并输出到*LogRecord*，请使用*makeLogRecord()* 函数。

makeSocket()

SocketHandler 的工厂方法会在此被重载以创建一个 UDP 套接字 (*socket.SOCK_DGRAM*)。

send(*s*)

将封存后的字节串发送到套接字。所发送字节串的格式与*SocketHandler.makePickle()* 文档中的描述一致。

16.8.10 SysLogHandler

SysLogHandler 类位于*logging.handlers* 模块，它支持将日志记录消息发送到远程或本地 Unix syslog。

class logging.handlers.**SysLogHandler**(*address=('localhost', SYSLOG_UDP_PORT)*, *facility=LOG_USER*, *socktype=socket.SOCK_DGRAM*)

返回一个*SysLogHandler* 类的新实例用来与通过 *address* 以 `(host, port)` 元组形式给出地址的远程 Unix 机器进行通讯。如果未指定 *address*，则使用 `('localhost', 514)`。该地址会被用于打开套接字。提供 `(host, port)` 元组的一种替代方式是提供字符串形式的地址，例如'/dev/log'。在这种情况下，会使用 Unix 域套接字将消息发送到 syslog。如果未指定 *facility*，则使用 `LOG_USER`。打开的套接字类型取决于 *socktype* 参数，该参数的默认值为*socket.SOCK_DGRAM* 即打开一个 UDP 套接字。要打开一个 TCP 套接字（用来配合较新的 syslog 守护程序例如 rsyslog 使用），请指定值为*socket.SOCK_STREAM*。

请注意如果你的服务器不是在 UDP 端口 514 上进行侦听，则*SysLogHandler* 可能无法正常工作。在这种情况下，请检查你应当为域套接字所使用的地址——它依赖于具体的系统。例如，在 Linux 上通常为'/dev/log' 而在 OS/X 上则为'/var/run/syslog'。你需要检查你的系统平台并使用适当的地址（如果你的应用程序需要在多个平台上运行则可能需要在运行时进行这样的检查）。在 Windows 上，你大概必须要使用 UDP 选项。

3.2 版更變: 添加了 *socktype*。

close()
关闭连接远程主机的套接字。

emit(*record*)
记录会被格式化，然后发送到 syslog 服务器。如果存在异常信息，则它 不会被发送到服务器。

3.2.1 版更變: (参见: bpo-12168。) 在较早的版本中，发送至 syslog 守护程序的消息总是以一个 NUL 字节结束，因为守护程序的早期版本期望接收一个以 NUL 结束的消息——即使它不包含于对应的规范说明 (**RFC 5424**)。这些守护程序的较新版本不再期望接收 NUL 字节，如果其存在则会将其去除，而最新的守护程序（更紧密地遵循 RFC 5424）会将 NUL 字节作为消息的一部分传递出去。

为了在面对所有这些不同守护程序行为时能够更方便地处理 syslog 消息，通过使用类层级属性 `append_nul`，添加 NUL 字节的操作已被作为可配置项。该属性默认为 `True` (保留现有行为) 但可在 `SysLogHandler` 实例上设为 `False` 以便让实例 不会添加 NUL 结束符。

3.3 版更變: (参见: bpo-12419。) 在较早的版本中，没有"ident" 或"tag" 前缀工具可以用来标识消息的来源。现在则可以使用一个类层级属性来设置它，该属性默认为 `""` 表示保留现有行为，但可在 `SysLogHandler` 实例上重载以便让实例不会为所处理的每条消息添加标识。请注意所提供的标识必须为文本而非字节串，并且它会被原封不动地添加到消息中。

encodePriority(*facility*, *priority*)
将功能和优先级编码为一个整数。你可以传入字符串或者整数——如果传入的是字符串，则会使用内部的映射字典将其转换为整数。

符号 `LOG_` 的值在*`SysLogHandler`* 中定义并且是 `sys/syslog.h` 头文件中所定义值的镜像。

优先级

名称（字符串）	符号值
`alert`	LOG_ALERT
`crit` 或 `critical`	LOG_CRIT
`debug`	LOG_DEBUG
`emerg` 或 `panic`	LOG_EMERG
`err` 或 `error`	LOG_ERR
`info`	LOG_INFO
`notice`	LOG_NOTICE
`warn` 或 `warning`	LOG_WARNING

设备

名称（字符串）	符号值
auth	LOG_AUTH
authpriv	LOG_AUTHPRIV
cron	LOG_CRON
daemon	LOG_DAEMON
ftp	LOG_FTP
kern	LOG_KERN
lpr	LOG_LPR
mail	LOG_MAIL
news	LOG_NEWS
syslog	LOG_SYSLOG
user	LOG_USER
uucp	LOG_UUCP
local0	LOG_LOCAL0
local1	LOG_LOCAL1
local2	LOG_LOCAL2
local3	LOG_LOCAL3
local4	LOG_LOCAL4
local5	LOG_LOCAL5
local6	LOG_LOCAL6
local7	LOG_LOCAL7

mapPriority(*levelname*)
将日志记录级别名称映射到 syslog 优先级名称。如果你使用自定义级别，或者如果默认算法不适合你的需要，你可能需要重载此方法。默认算法将 DEBUG, INFO, WARNING, ERROR 和 CRITICAL 映射到等价的 syslog 名称，并将所有其他级别名称映射到'warning'。

16.8.11 NTEventLogHandler

NTEventLogHandler 类位于 *logging.handlers* 模块，它支持将日志记录消息发送到本地 Windows NT, Windows 2000 或 Windows XP 事件日志。在你使用它之前，你需要安装 Mark Hammond 的 Python Win32 扩展。

class logging.handlers.**NTEventLogHandler**(*appname*, *dllname=None*, *logtype='Application'*)
返回一个 *NTEventLogHandler* 类的新实例。*appname* 用来定义出现在事件日志中的应用名称。将使用此名称创建适当的注册表条目。*dllname* 应当给出要包含在日志中的消息定义的.dll 或.exe 的完整限定路径名称（如未指定则会使用 'win32service.pyd' ——此文件随 Win32 扩展安装且包含一些基本的消息定义占位符。请注意使用这些占位符将使你的事件日志变得很大，因为整个消息源都会被放入日志。如果你希望有较小的日志，你必须自行传入包含你想要在事件日志中使用的消息定义的.dll 或.exe 名称）。*logtype* 为 'Application', 'System' 或 'Security' 之一，且默认值为 'Application'。

close()
这时，你就可以从注册表中移除作为事件日志条目来源的应用名称。但是，如果你这样做，你将无法如你所预期的那样在事件日志查看器中看到这些事件——它必须能访问注册表来获取.dll 名称。当前版本并不会这样做。

emit(*record*)
确定消息 ID，事件类别和事件类型，然后将消息记录到 NT 事件日志中。

getEventCategory(*record*)
返回记录的事件类别。如果你希望指定你自己的类别就要重载此方法。此版本将返回 0。

getEventType(*record*)
返回记录的事件类型。如果你希望指定你自己的类型就要重载此方法。此版本将使用处理程序的 typemap 属性来执行映射，该属性在 `__init__()` 被设置为一个字典，其中包含 `DEBUG`, `INFO`, `WARNING`, `ERROR` 和 `CRITICAL` 的映射。如果你使用你自己的级别，你将需要重载此方法或者在处理程序的 *typemap* 属性中放置一个合适的字典。

getMessageID(*record*)
返回记录的消息 ID。如果你使用你自己的消息，你可以通过将 *msg* 传给日志记录器作为 ID 而非格式字符串实现此功能。然后，你可以在这里使用字典查找来获取消息 ID。此版本将返回 1，是 `win32service.pyd` 中的基本消息 ID。

16.8.12 SMTPHandler

SMTPHandler 类位于 *logging.handlers* 模块，它支持将日志记录消息通过 SMTP 发送到一个电子邮件地址。

class logging.handlers.**SMTPHandler**(*mailhost*, *fromaddr*, *toaddrs*, *subject*, *credentials=None*, *secure=None*, *timeout=1.0*)
返回一个 *SMTPHandler* 类的新实例。该实例使用电子邮件的发件人、收件人地址和主题行进行初始化。*toaddrs* 应当为字符串列表。要指定一个非标准 SMTP 端口，请使用 (host, port) 元组格式作为 *mailhost* 参数。如果你使用一个字符串，则会使用标准 SMTP 端口。如果你的 SMTP 服务器要求验证，你可以指定一个 (username, password) 元组作为 *credentials* 参数。

要指定使用安全协议 (TLS)，请传入一个元组作为 *secure* 参数。这将仅在提供了验证凭据时才能被使用。元组应当或是一个空元组，或是一个包含密钥文件名的单值元组，或是一个包含密钥文件和证书文件的 2 值元组。(此元组会被传给 *smtplib.SMTP.starttls()* 方法。)

可以使用 *timeout* 参数为与 SMTP 服务器的通信指定超时限制。

3.3 版新加入: 增加了 *timeout* 参数。

emit(*record*)
对记录执行格式化并将其发送到指定的地址。

getSubject(*record*)
如果你想要指定一个基于记录的主题行，请重载此方法。

16.8.13 MemoryHandler

MemoryHandler 类位于 *logging.handlers* 模块，它支持在内存中缓冲日志记录，并定期将其刷新到 *target* 处理程序中。刷新会在缓冲区满的时候，或是在遇到特定或更高严重程度事件的时候发生。

MemoryHandler 是更通用的 *BufferingHandler* 的子类，后者属于抽象类。它会在内存中缓冲日志记录。当每条记录被添加到缓冲区时，会通过调用 `shouldFlush()` 来检查缓冲区是否应当刷新。如果应当刷新，则要使用 `flush()` 来执行刷新。

class logging.handlers.**BufferingHandler**(*capacity*)
使用指定容量的缓冲区初始化处理程序。这里，*capacity* 是指缓冲的日志记录数量。

emit(*record*)
将记录添加到缓冲区。如果 *shouldFlush()* 返回真值，则会调用 *flush()* 来处理缓冲区。

flush()
你可以重载此方法来实现自定义的刷新行为。此版本只是将缓冲区清空。

shouldFlush(*record*)
如果缓冲区容量已满则返回 `True`。可以重载此方法以实现自定义的刷新策略。

class logging.handlers.**MemoryHandler**(*capacity*, *flushLevel=ERROR*, *target=None*, *flushOnClose=True*)

返回一个*MemoryHandler* 类的新实例。该实例使用 *capacity* 指定的缓冲区大小（要缓冲的记录数量）来初始化。如果 *flushLevel* 未指定，则使用 ERROR。如果未指定 *target*，则需要在此处理程序执行任何实际操作之前使用*setTarget()* 来设置目标。如果 *flushOnClose* 指定为 False，则当处理程序被关闭时 不会刷新缓冲区。如果未指定或指定为 True，则当处理程序被关闭时将会发生之前的缓冲区刷新行为。

3.6 版更變: 增加了 *flushOnClose* 形参。

close()

调用*flush()*，设置目标为 None 并清空缓冲区。

flush()

对于*MemoryHandler*，刷新是指将缓冲的记录发送到目标，如果存在目标的话。当此行为发生时缓冲区也将被清空。如果你想要不同的行为请重载此方法。

setTarget(*target*)

设置此处理程序的目标处理程序。

shouldFlush(*record*)

检测缓冲区是否已满或是有记录为 *flushLevel* 或更高级别。

16.8.14 HTTPHandler

HTTPHandler 类位于*logging.handlers* 模块，它支持使用 GET 或 POST 语义将日志记录消息发送到 Web 服务器。

class logging.handlers.**HTTPHandler**(*host*, *url*, *method='GET'*, *secure=False*, *credentials=None*, *context=None*)

返回一个*HTTPHandler* 类的新实例。*host* 可以为 host:port 的形式，如果你需要使用指定端口号的话。如果没有指定 *method*，则会使用 GET。如果 *secure* 为真值，则将使用 HTTPS 连接。*context* 形参可以设为一个*ssl.SSLContext* 实例以配置用于 HTTPS 连接的 SSL 设置。如果指定了 *credentials*，它应当为包含 userid 和 password 的二元组，该元组将被放入使用 Basic 验证的 HTTP 'Authorization' 标头中。如果你指定了 credentials，你还应当指定 secure=True 这样你的 userid 和 password 就不会以明文在线路上传输。

3.5 版更變: 增加了 *context* 形参。

mapLogRecord(*record*)

基于 record 提供一个字典，它将被执行 URL 编码并发送至 Web 服务器。默认实现仅返回 record.__dict__。在只需将*LogRecord* 的某个子集发送至 Web 服务器，或者需要对发送至服务器的内容进行更多定制时可以重载此方法。

emit(*record*)

将记录以经 URL 编码的形式发送至 Web 服务器。会使用*mapLogRecord()* 方法来将要发送的记录转换为字典。

備𝔽: 由于记录发送至 Web 服务器所需的预处理与通用的格式化操作不同，使用*setFormatter()* 来指定一个*Formatter* 用于*HTTPHandler* 是没有效果的。此处理程序不会调用*format()*，而是调用*mapLogRecord()* 然后再调用*urllib.parse.urlencode()* 来以适合发送至 Web 服务器的形式对字典进行编码。

16.8.15 QueueHandler

3.2 版新加入.

QueueHandler 类位于*logging.handlers* 模块，它支持将日志记录消息发送到一个队列，例如在*queue* 或*multiprocessing* 模块中实现的队列。

配合*QueueListener* 类使用，*QueueHandler* 可用来使处理程序在与执行日志记录的线程不同的线程上完成工作。这对 Web 应用程序以及其他服务于客户端的线程需要尽可能快地响应的服务应用程序来说很重要，任何潜在的慢速操作（例如通过*SMTPHandler* 发送邮件）都要在单独的线程上完成。

class logging.handlers.**QueueHandler**(*queue*)

返回一个*QueueHandler* 类的新实例。该实例使用队列来初始化以向其发送消息。*queue* 可以为任何队列类对象；它由*enqueue()* 方法来使用，该方法需要知道如何向其发送消息。队列 *不要求*具有任务跟踪 API，这意味着你可以为 *queue* 使用*SimpleQueue* 实例。

emit(*record*)

将准备 LogRecord 的结果排入队列。如果发生了异常（例如由于有界队列已满），则会调用*handleError()* 方法来处理错误。这可能导致记录被静默地丢弃 (如果 logging.raiseExceptions 为 False) 或者消息被打印到 sys.stderr (如果 logging.raiseExceptions 为 True)。

prepare(*record*)

准备用于队列的记录。此方法返回的对象会被排入队列。

基本实现会格式化记录以合并消息、参数以及可能存在的异常信息。它还会从记录中原地移除无法封存的条目。

如果你想要将记录转换为 dict 或 JSON 字符串，或者发送记录被修改后的副本而让初始记录保持原样，则你可能会想要重载此方法。

enqueue(*record*)

使用 put_nowait() 将记录排入队列；如果你想要使用阻塞行为，或超时设置，或自定义的队列实现，则你可能会想要重载此方法。

16.8.16 QueueListener

3.2 版新加入.

QueueListener 类位于*logging.handlers* 模块，它支持从一个队列接收日志记录消息，例如在*queue* 或*multiprocessing* 模块中实现的队列。消息是在内部线程中从队列接收并在同一线程上传递到一个或多个处理程序进行处理的。尽管*QueueListener* 本身并不是一个处理程序，但由于它要与*QueueHandler* 配合工作，因此也在此处介绍。

配合*QueueHandler* 类使用，*QueueListener* 可用来使处理程序在与执行日志记录的线程不同的线程上完成工作。这对 Web 应用程序以及其他服务与客户端的线程需要尽可能快地响应的服务应用程序来说很重要，任何潜在的慢速动作（例如通过*SMTPHandler* 发送邮件）都要在单独的线程上完成。

class logging.handlers.**QueueListener**(*queue*, **handlers*, *respect_handler_level=False*)

返回一个*QueueListener* 类的新实例。该实例初始化时要传入一个队列以向其发送消息，还要传入一个处理程序列表用来处理放置在队列中的条目。队列可以是任何队列类对象；它会被原样传给*dequeue()* 方法，该方法需要知道如何从其获取消息。队列 *不要求*具有任务跟踪 API（但如提供则会使用它），这意味着你可以为 *queue* 使用*SimpleQueue* 实例。

如果 respect_handler_level 为 True，则在决定是否将消息传递给处理程序之前会遵循处理程序的级别（与消息的级别进行比较）；在其他情况下，其行为与之前的 Python 版本一致——总是将每条消息传递给每个处理程序。

3.5 版更變: 增加了 respect_handler_level 参数。

dequeue(*block*)
从队列移出一条记录并将其返回，可以选择阻塞。

基本实现使用 `get()`。如果你想要使用超时设置或自定义的队列实现，则你可能会想要重载此方法。

prepare(*record*)
准备一条要处理的记录。

该实现只是返回传入的记录。如果你想要对记录执行任何自定义的 marshal 操作或在将其传给处理程序之前进行调整，则你可能会想要重载此方法。

handle(*record*)
处理一条记录。

此方法简单地循环遍历处理程序，向它们提供要处理的记录。实际传给处理程序的对象就是从*prepare()* 返回的对象。

start()
启动监听器。

此方法启动一个后台线程来监视 LogRecords 队列以进行处理。

stop()
停止监听器。

此方法要求线程终止，然后等待它完成终止操作。请注意在你的应用程序退出之前如果你没有调用此方法，则可能会有一些记录在留在队列中，它们将不会被处理。

enqueue_sentinel()
将一个标记写入队列以通知监听器退出。此实现会使用 `put_nowait()`。如果你想要使得超时设置或自定义的队列实现，则你可能会想要重载此方法。

3.3 版新加入.

也参考:

模块*logging* 日志记录模块的 API 参考。

模块*logging.config* 日志记录模块的配置 API 。

16.9 getpass --- 便携式密码输入工具

源代码: Lib/getpass.py

---

getpass 模块提供了两个函数:

getpass.**getpass**(*prompt='Password: '*, *stream=None*)
提示用户输入一个密码且不会回显。用户会看到字符串 *prompt* 作为提示，其默认值为 `'Password: '`。在 Unix 上，如有必要提示会使用替换错误句柄写入到文件类对象 *stream*。*stream* 默认指向控制终端 (`/dev/tty`)，如果不可用则指向 `sys.stderr` (此参数在 Windows 上会被忽略)。

如果回显自由输入不可用则 getpass() 将回退为打印一条警告消息到 *stream* 并且从 `sys.stdin` 读取同时发出*GetPassWarning*。

備🄵: 如果你从 IDLE 内部调用 getpass，输入可能是在你启动 IDLE 的终端中而非在 IDLE 窗口本身中完成。

---

exception getpass.**GetPassWarning**
一个当密码输入可能被回显时发出的*UserWarning* 子类。

getpass.**getuser**()
返回用户的“登录名称”。

此函数会按顺序检查环境变量 LOGNAME, USER, LNAME 和 USERNAME，并返回其中第一个被设置为非空字符串的值。如果均未设置，则在支持*pwd* 模块的系统上将返回来自密码数据库的登录名，否则将引发一个异常。

通常情况下，此函数应优先于*os.getlogin()* 使用。

16.10 curses --- 终端字符单元显示的处理

curses 模块提供了 curses 库的接口，这是可移植高级终端处理的事实标准。

虽然 curses 在 Unix 环境中使用最为广泛，但也有适用于 Windows，DOS 以及其他可能的系统的版本。此扩展模块旨在匹配 ncurses 的 API，这是一个部署在 Linux 和 Unix 的 BSD 变体上的开源 curses 库。

備🄵: 每当文档提到 **字符**时，它可以被指定为一个整数，一个单字符 Unicode 字符串或者一个单字节的字节字符串。

每当此文档提到 **字符串**时，它可以被指定为一个 Unicode 字符串或者一个字节字符串。

備🄵: 从 5.4 版本开始，ncurses 库使用 nl_langinfo 函数来决定如何解释非 ASCII 数据。这意味着你需要在程序中调用*locale.setlocale()* 函数，并使用一种系统中可用的编码方法来编码 Unicode 字符串。这个例子使用了系统默认的编码:

```
import locale
locale.setlocale(locale.LC_ALL, '')
code = locale.getpreferredencoding()
```

然后使用 *code* 作为*str.encode()* 调用的编码。

也参考:

模块*curses.ascii* 在 ASCII 字符上工作的工具，无论你的区域设置是什么。

模块*curses.panel* 为 curses 窗口添加深度的面板栈扩展。

模块*curses.textpad* 用于使 curses 支持 **Emacs** 式绑定的可编辑文本部件。

curses-howto 关于配合 Python 使用 curses 的教学材料，由 Andrew Kuchling 和 Eric Raymond 撰写。

Python 源码发布包的 Tools/demo/ 目录包含了一些使用此模块所提供的 curses 绑定的示例程序。

16.10.1 函数

curses 模块定义了以下异常：

exception curses.**error**
当 curses 库中函数返回一个错误时引发的异常。

備𝔽: 只要一个函数或方法的 *x* 或 *y* 参数是可选项，它们会默认为当前光标位置。而当 *attr* 是可选项时，它会默认为 A_NORMAL。

---

curses 模块定义了以下函数：

curses.**baudrate**()
以每秒比特数为单位返回终端输出速度。在软件终端模拟器上它将具有一个固定的最高值。此函数出于历史原因被包括；在以前，它被用于写输出循环以提供时间延迟，并偶尔根据线路速度来改变接口。

curses.**beep**()
发出短促的提醒声音。

curses.**can_change_color**()
根据程序员能否改变终端显示的颜色返回 True 或 False。

curses.**cbreak**()
进入 cbreak 模式。在 cbreak 模式（有时也称为“稀有”模式）通常的 tty 行缓冲会被关闭并且字符可以被一个一个地读取。但是，与原始模式不同，特殊字符（中断、退出、挂起和流程控制）会在 tty 驱动和调用程序上保留其效果。首先调用*raw()* 然后调用*cbreak()* 会将终端置于 cbreak 模式。

curses.**color_content**(*color_number*)
返回颜色值 *color_number* 中红、绿和蓝（RGB）分量的强度，此强度值必须介于 0 和 COLORS - 1 之间。返回一个 3 元组，其中包含给定颜色的 R,G,B 值，它们必须介于 0 (无分量) 和 1000 (最大分量) 之间。

curses.**color_pair**(*pair_number*)
返回用于以指定颜色对显示文本的属性值。仅支持前 256 个颜色对。该属性值可与 A_STANDOUT, A_REVERSE 以及其他 A_* 属性组合使用。*pair_number()* 是此函数的对应操作。

curses.**curs_set**(*visibility*)
设置光标状态。*visibility* 可设为 0, 1 或 2 表示不可见、正常与高度可见。如果终端支持所请求的可见性，则返回之前的光标状态；否则会引发异常。在许多终端上，“正常可见”模式为下划线光标而“高度可见”模式为方块形光标。

curses.**def_prog_mode**()
将当前终端模式保存为”program” 模式，即正在运行的程序使用 curses 的模式。（与其相对的是”shell” 模式，即程序不使用 curses。）对*reset_prog_mode()* 的后续调用将恢复此模式。

curses.**def_shell_mode**()
将当前终端模式保存为”shell” 模式，即正在运行的程序不使用 curses 的模式。（与其相对的是”program” 模式，即程序使用功能。）对*reset_shell_mode()* 的后续调用将恢复此模式。

curses.**delay_output**(*ms*)
在输出中插入 *ms* 毫秒的暂停。

curses.**doupdate**()
更新物理屏幕。curses 库会保留两个数据结构，一个代表当前物理屏幕的内容以及一个虚拟屏幕代表需要的后续状态。*doupdate()* 整体更新物理屏幕以匹配虚拟屏幕。

虚拟屏幕可以通过在写入操作例如在一个窗口上执行*addstr()* 之后调用*noutrefresh()* 来刷新。普通的*refresh()* 调用只是简单的 noutrefresh() 加 doupdate()；如果你需要更新多个窗口，

你可以通过在所有窗口上发出 noutrefresh() 调用再加单次 doupdate() 来提升性能并可减少屏幕闪烁。

curses.**echo**()
进入 echo 模式。在 echo 模式下，输入的每个字符都会在输入后回显到屏幕上。

curses.**endwin**()
撤销库的初始化，使终端返回正常状态。

curses.**erasechar**()
将用户的当前擦除字符以单字节字节串对象的形式返回。在 Unix 操作系统下这是 curses 程序用来控制 tty 的属性，而不是由 curses 库本身来设置的。

curses.**filter**()
如果要使用*filter()* 例程，它必须在调用*initscr()* 之前被调用。其效果是在这些调用期间，LINES 会被设为 1；clear，cup，cud，cud1，cuu1，cuu，vpa 等功能会被禁用；而 home 字符串会被设为 cr 的值。其影响是光标会被限制在当前行内，屏幕刷新也是如此。这可被用于启用单字符模式的行编辑而不触及屏幕的其余部分。

curses.**flash**()
闪烁屏幕。也就是将其改为反显并在很短的时间内将其改回原状。有些人更喜欢这样的‘视觉响铃’而非*beep()* 所产生的听觉提醒信号。

curses.**flushinp**()
刷新所有输入缓冲区。这会丢弃任何已被用户输入但尚未被程序处理的预输入内容。

curses.**getmouse**()
在*getch()* 返回 KEY_MOUSE 以发出鼠标事件信号之后，应当调用此方法来获取加入队列的鼠标事件，事件以一个 5 元组 (id, x, y, z, bstate) 来表示。*id* 为用于区分多个设置的 ID 值，*x*, *y*, *z* 为事件的坐标。(*z* 目前未被使用。) *bstate* 为一个整数值，其各比特位将设置用于表示事件类型，并将是下列常量中一个或多个按位 OR 的结果，其中 *n* 是以 1 至 4 表示的键号: BUTTONn_PRESSED, BUTTONn_RELEASED, BUTTONn_CLICKED, BUTTONn_DOUBLE_CLICKED, BUTTONn_TRIPLE_CLICKED, BUTTON_SHIFT, BUTTON_CTRL, BUTTON_ALT。

curses.**getsyx**()
将当前虚拟屏幕光标的坐标作为元组 (y, x) 返回。如果*leaveok* 当前为 True，则返回 (-1, -1)。

curses.**getwin**(*file*)
读取由之前的 putwin() 调用存放在文件中的窗口相关数据。该例程随后将使用该数据创建并初始化一个新窗口，并返回该新窗口对象。

curses.**has_colors**()
如果终端能显示彩色则返回 True；否则返回 False。

curses.**has_ic**()
如果终端具有插入和删除字符的功能则返回 True。此函数仅是出于历史原因而被包括的，因为所有现代软件终端模拟器都具有这些功能。

curses.**has_il**()
如果终端具有插入和删除字符功能，或者能够使用滚动区域来模拟这些功能则返回 True。此函数仅是出于历史原因而被包括的，因为所有现代软件终端模拟器都具有这些功能。

curses.**has_key**(*ch*)
接受一个键值 *ch*，并在当前终端类型能识别出具有该值的键时返回 True。

curses.**halfdelay**(*tenths*)
用于半延迟模式，与 cbreak 模式的类似之处是用户所键入的字符会立即对程序可用。但是，在阻塞 *tenths* 个十分之一秒之后，如果还未输入任何内容则将引发异常。*tenths* 值必须为 1 和 255 之间的数字。使用*nocbreak()* 可退出半延迟模式。

curses.**init_color**(*color_number*, *r*, *g*, *b*)
更改某个颜色的定义，接受要更改的颜色编号以及三个 RGB 值（表示红绿蓝三分量的强度）。*color_number* 值必须为 0 和 *COLORS - 1* 之间的数字。*r*, *g*, *b* 值必须为 0 和 1000 之间的数字。当使用*init_color()* 时，出现在屏幕上的对应颜色会立即按照新定义来更改。此函数在大多数终端上都是无操作的；它仅会在*can_change_color()* 返回 True 时生效。

curses.**init_pair**(*pair_number*, *fg*, *bg*)
更改某个颜色对的定义。它接受三个参数：要更改的颜色对编号，前景色编号和背景色编号。*pair_number* 值必须为 1 和 COLOR_PAIRS - 1 之间的数字（并且 0 号颜色对固定为黑底白字而无法更改）。*fg* 和 *bg* 参数必须为 0 和 COLORS - 1 之间的数字，或者在调用*use_default_colors()* 之后则为 -1。如果颜色对之前已被初始化，则屏幕会被刷新使得出现在屏幕上的该颜色会立即按照新定义来更改。

curses.**initscr**()
初始化库。返回代表整个屏幕的窗口 对象。

備F: 如果打开终端时发生错误，则下层的 curses 库可能会导致解释器退出。

---

curses.**is_term_resized**(*nlines*, *ncols*)
如果*resize_term()* 会修改窗口结构则返回 True，否则返回 False。

curses.**isendwin**()
如果*endwin()* 已经被调用（即 curses 库已经被撤销初始化则返回 True。

curses.**keyname**(*k*)
将编号为 *k* 的键名称作为字节串对象返回。生成可打印 ASCII 字符的键名称就是键所对应的字符。Ctrl-键组合的键名称则是一个两字节的字节串对象，它由插入符 (b'^') 加对应的可打印 ASCII 字符组成。Alt-键组合 (128--255) 的键名称则是由前缀 b'M-' 加对应的可打印 ASCII 字符组成的字节串对象。

curses.**killchar**()
将用户的当前行删除字符以单字节字节串对象的形式返回。在 Unix 操作系统下这是 curses 程序用来控制 tty 的属性，而不是由 curses 库本身来设置的。

curses.**longname**()
返回一个字节串对象，其中包含描述当前终端的 terminfo 长名称字段。详细描述的最大长度为 128 个字符。它仅在调用*initscr()* 之后才会被定义。

curses.**meta**(*flag*)
如果 *flag* 为 True，则允许输入 8 比特位的字符。如果 *flag* 为 False，则只允许 7 比特位的字符。

curses.**mouseinterval**(*interval*)
以毫秒为单位设置能够被识别为点击的按下和释放事件之间可以间隔的最长时间，并返回之前的间隔值。默认值为 200 毫秒，即五分之一秒。

curses.**mousemask**(*mousemask*)
设置要报告的鼠标事件，并返回一个元组 (availmask, oldmask)。*availmask* 表明指定的鼠标事件中哪些可以被报告；当完全失败时将返回 0。*oldmask* 是给定窗口的鼠标事件之前的掩码值。如果从未调用此函数，则不会报告任何鼠标事件。

curses.**napms**(*ms*)
休眠 *ms* 毫秒。

curses.**newpad**(*nlines*, *ncols*)
创建并返回一个指向具有给定行数和列数新的面板数据结构的指针。将面板作为窗口对象返回。

面板类似于窗口，区别在于它不受屏幕大小的限制，并且不必与屏幕的特定部分相关联。面板可以在需要使用大窗口时使用，并且每次只需将窗口的一部分放在屏幕上。面板不会发生自动刷新（例如由于滚动或输入回显）。面板的*refresh()* 和*noutrefresh()* 方法需要 6 个参数来指定面板要显示的部分以及要用于显示的屏幕位置。这些参数是 *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*；*p* 参数表示要显示的面板区域的左上角而 *s* 参数定义了要显示的面板区域在屏幕上的剪切框。

curses.**newwin**(*nlines*, *ncols*)
curses.**newwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
返回一个新的窗口，其左上角位于 (begin_y, begin_x)，并且其高度/宽度为 *nlines*/*ncols*。

默认情况下，窗口将从指定位置扩展到屏幕的右下角。

curses.**nl**()
进入 newline 模式。此模式会在输入时将回车转换为换行符，并在输出时将换行符转换为回车加换行。newline 模式会在初始时启用。

curses.**nocbreak**()
退出 cbreak 模式。返回具有行缓冲的正常"cooked" 模式。

curses.**noecho**()
退出 echo 模式。关闭输入字符的回显。

curses.**nonl**()
退出 newline 模式。停止在输入时将回车转换为换行，并停止在输出时从换行到换行/回车的底层转换（但这不会改变 addch('\n') 的行为，此行为总是在虚拟屏幕上执行相当于回车加换行的操作）。当停止转换时，curses 有时能使纵向移动加快一些；并且，它将能够在输入时检测回车键。

curses.**noqiflush**()
当使用 noqiflush() 例程时，与 INTR, QUIT 和 SUSP 字符相关联的输入和输出队列的正常刷新将不会被执行。如果你希望在处理程序退出后还能继续输出，就像没有发生过中断一样，你可能会想要在信号处理程序中调用 noqiflush()。

curses.**noraw**()
退出 raw 模式。返回具有行缓冲的正常"cooked" 模式。

curses.**pair_content**(*pair_number*)
返回包含对应于所请求颜色对的元组 (fg, bg)。*pair_number* 的值必须在 0 和 COLOR_PAIRS - 1 之间。

curses.**pair_number**(*attr*)
返回通过属性值 *attr* 所设置的颜色对的编号。*color_pair()* 是此函数的对应操作。

curses.**putp**(*str*)
等价于 tputs(str, 1, putchar)；为当前终端发出指定 terminfo 功能的值。请注意*putp()* 的输出总是前往标准输出。

curses.**qiflush**([*flag*])
如果 *flag* 为 False，则效果与调用*noqiflush()* 相同。如果 *flag* 为 True 或未提供参数，则在读取这些控制字符时队列将被刷新。

curses.**raw**()
进入 raw 模式。在 raw 模式下，正常的行缓冲和对中断、退出、挂起和流程控制键的处理会被关闭；字符会被逐个地提交给 curses 输入函数。

curses.**reset_prog_mode**()
将终端恢复到"program" 模式，如之前由*def_prog_mode()* 所保存的一样。

curses.**reset_shell_mode**()
将终端恢复到"shell" 模式，如之前由*def_shell_mode()* 所保存的一样。

curses.**resetty**()
将终端模式恢复到最后一次调用*savetty()* 时的状态。

curses.**resize_term**(*nlines*, *ncols*)
由*resizeterm()* 用来执行大部分工作的后端函数；当调整窗口大小时，*resize_term()* 会以空白填充扩展区域。调用方应用程序应当以适当的数据填充这些区域。resize_term() 函数会尝试调整所有窗口的大小。但是，由于面板的调用约定，在不与应用程序进行额外交互的情况下是无法调整其大小的。

curses.**resizeterm**(*nlines*, *ncols*)
将标准窗口和当前窗口的大小调整为指定的尺寸，并调整由 curses 库所使用的记录窗口尺寸的其他记录数据（特别是 SIGWINCH 处理程序）。

curses.**savetty**()
将终端模式的当前状态保存在缓冲区中，可供*resetty()* 使用。

curses.**get_escdelay**()
提取通过*set_escdelay()* 设置的值。

3.9 版新加入.

curses.**set_escdelay**(*ms*)
设置读取一个转义字符后要等待的毫秒数，以区分在键盘上输入的单个转义字符与通过光标和功能键发送的转义序列。

3.9 版新加入.

curses.**get_tabsize**()
提取通过*set_tabsize()* 设置的值。

3.9 版新加入.

curses.**set_tabsize**(*size*)
设置 curses 库在将制表符添加到窗口时将制表符转换为空格所使用的列数。

3.9 版新加入.

curses.**setsyx**(*y*, *x*)
将虚拟屏幕光标设置到 *y*, *x*。如果 *y* 和 *x* 均为 `-1`，则*leaveok* 将设为 `True`。

curses.**setupterm**(*term=None*, *fd=-1*)
初始化终端。*term* 为给出终端名称的字符串或为 `None`；如果省略或为 `None`，则将使用 `TERM` 环境变量的值。*fd* 是任何初始化序列将被发送到的文件描述符；如未指定或为 `-1`，则将使用 `sys.stdout` 的文件描述符。

curses.**start_color**()
如果程序员想要使用颜色，则必须在任何其他颜色操作例程被调用之前调用它。在*initscr()* 之后立即调用此例程是一个很好的做法。

start_color() 会初始化八种基本颜色（黑、红、绿、黄、蓝、品、青和白）以及*curses* 模块中的两个全局变量 `COLORS` 和 `COLOR_PAIRS`，其中包含终端可支持的颜色和颜色对的最大数量。它还会将终端中的颜色恢复为终端刚启动时的值。

curses.**termattrs**()
返回终端所支持的所有视频属性逻辑 OR 的值。此信息适用于当 curses 程序需要对屏幕外观进行完全控制的情况。

curses.**termname**()
将环境变量 `TERM` 的值截短至 14 个字节，作为字节串对象返回。

curses.**tigetflag**(*capname*)
将与 terminfo 功能名称 *capname* 相对应的布尔功能值以整数形式返回。如果 *capname* 不是一个布尔功能则返回 `-1`，如果其被取消或不存在于终端描述中则返回 `0`。

curses.**tigetnum**(*capname*)
将与 terminfo 功能名称 *capname* 相对应的数字功能值以整数形式返回。如果 *capname* 不是一个数字功能则返回 `-2`，如果其被取消或不存在于终端描述中则返回 `-1`。

curses.**tigetstr**(*capname*)
将与 terminfo 功能名称 *capname* 相对应的字符串功能值以字节串对象形式返回。如果 *capname* 不是一个 terminfo ” 字符串功能” 或者如果其被取消或不存在于终端描述中则返回 `None`。

curses.**tparm**(*str*[, ...])
使用提供的形参初始化字节串对象 *str*，其中 *str* 应当是从 terminfo 数据库获取的参数化字符串。例如 `tparm(tigetstr("cup"), 5, 3)` 的结果可能为 `b'\033[6;4H'`，实际结果将取决于终端类型。

curses.**typeahead**(*fd*)
指定将被用于预输入检查的文件描述符 *fd*。如果 *fd* 为 `-1`，则不执行预输入检查。

curses 库会在更新屏幕时通过定期查找预输入来执行" 断行优化"。如果找到了输入，并且输入是来自于 tty，则会将当前更新推迟至 refresh 或 doupdate 再次被调用的时候，以便允许更快地响应预先输入的命令。此函数允许为预输入检查指定其他的文件描述符。

curses.**unctrl**(*ch*)
返回一个字节串对象作为字符 *ch* 的可打印表示形式。控制字符会表示为一个变换符加相应的字符，例如 `b'^C'`。可打印字符则会保持原样。

curses.**ungetch**(*ch*)
推送 *ch* 以便让下一个*getch()* 返回该字符。

備🄵: 在 `getch()` 被调用之前只能推送一个 *ch*。

---

curses.**update_lines_cols**()
更新 `LINES` 和 `COLS`。适用于检测屏幕大小的手动调整。

3.5 版新加入.

curses.**unget_wch**(*ch*)
推送 *ch* 以便让下一个*get_wch()* 返回该字符。

備🄵: 在 `get_wch()` 被调用之前只能推送一个 *ch*。

3.3 版新加入.

curses.**ungetmouse**(*id*, *x*, *y*, *z*, *bstate*)
将 `KEY_MOUSE` 事件推送到输入队列，将其与给定的状态数据进行关联。

curses.**use_env**(*flag*)
如果使用此函数，则应当在调用*initscr()* 或 newterm 之前调用它。当 *flag* 为 `False` 时，将会使用在 terminfo 数据库中指定的行和列的值，即使设置了环境变量 `LINES` 和 `COLUMNS` (默认使用)，或者如果 curses 是在窗口中运行（在此情况下如果未设置 `LINES` 和 `COLUMNS` 则默认行为将是使用窗口大小）。

curses.**use_default_colors**()
允许在支持此特性的终端上使用默认的颜色值。使用此函数可在你的应用程序中支持透明效果。默认颜色会被赋给颜色编号 `-1`。举例来说，在调用此函数后，`init_pair(x, curses.COLOR_RED, -1)` 会将颜色对 *x* 初始化为红色前景和默认颜色背景。

curses.**wrapper**(*func*, */*, **args*, ***kwargs*)
初始化 curses 并调用另一个可调用对象 *func*，该对象应当为你的使用 curses 的应用程序的其余部分。如果应用程序引发了异常，此函数将在重新引发异常并生成回溯信息之前将终端恢复到正常状态。随后可调用对象 *func* 会被传入主窗口'stdscr' 作为其第一个参数，再带上其他所有传给 `wrapper()` 的参数。在调用 *func* 之前，`wrapper()` 会启用 cbreak 模式，关闭回显，启用终端键盘，并在终端具有颜色支持的情况下初始化颜色。在退出时（无论是正常退出还是异常退出）它会恢复 cooked 模式，打开回显，并禁用终端键盘。

16.10.2 Window 对象

Window 对象会由上面的*initscr()* 和*newwin()* 返回，它具有以下方法和属性:

window.**addch**(*ch*[, *attr*])
window.**addch**(*y*, *x*, *ch*[, *attr*])
将带有属性 *attr* 的字符 *ch* 绘制到 (y, x)，覆盖之前在该位置上绘制的任何字符。默认情况下，字符的位置和属性均为窗口对象的当前设置。

備𝔽: 在窗口、子窗口或面板之外写入会引发*curses.error*。尝试在窗口、子窗口或面板的右下角写入将在字符被打印之后导致异常被引发。

---

window.**addnstr**(*str*, *n*[, *attr*])
window.**addnstr**(*y*, *x*, *str*, *n*[, *attr*])
将带有属性 *attr* 的字符串 *str* 中的至多 *n* 个字符绘制到 (y, x)，覆盖之前在屏幕上的任何内容。

window.**addstr**(*str*[, *attr*])
window.**addstr**(*y*, *x*, *str*[, *attr*])
将带有属性 *attr* 的字符串 *str* 绘制到 (y, x)，覆盖之前在屏幕上的任何内容。

備𝔽:

- 在窗口、子窗口或面板之外写入会引发*curses.error*。尝试在窗口、子窗口或面板的右下角写入将在字符串被打印之后导致异常被引发。
- 此 Python 模块的后端 ncurses 中的一个缺陷 会在调整窗口大小时导致段错误。此缺陷已在 ncurses-6.1-20190511 中被修复。如果你必须使用较早版本的 ncurses，则你只要在调用*addstr()* 时不传入嵌入了换行符的 *str* 即可避免触发此错误。请为每一行分别调用*addstr()*。

---

window.**attroff**(*attr*)
从应用于写入到当前窗口的"background" 集中移除属性 *attr*。

window.**attron**(*attr*)
向应用于写入到当前窗口的"background" 集中添加属性 *attr*。

window.**attrset**(*attr*)
将"background" 属性集设为 *attr*。该集合初始时为 0 (无属性)。

window.**bkgd**(*ch*[, *attr*])
将窗口 background 特征属性设为带有属性 *attr* 的字符 *ch*。随后此修改将应用于放置到该窗口中的每个字符。

- 窗口中每个字符的属性会被修改为新的 background 属性。
- 不论之前的 background 字符出现在哪里，它都会被修改为新的 background 字符。

window.**bkgdset**(*ch*[, *attr*])
设置窗口的背景。窗口的背景由字符和属性的任意组合构成。背景的属性部分会与写入窗口的所有非空白字符合并（即 OR 运算）。背景和字符和属性部分均会与空白字符合并。背景将成为字符的特征属性并在任何滚动与插入/删除行/字符操作中与字符一起移动。

window.**border**([*ls*[, *rs*[, *ts*[, *bs*[, *tl*[, *tr*[, *bl*[, *br*]]]]]]]])
在窗口边缘绘制边框。每个参数指定用于边界特定部分的字符; 请参阅下表了解更多详情。

備𝔽: 任何形参的值为 0 都将导致该形参使用默认字符。关键字形参 不可被使用。默认字符在下表中列出:

参数	描述	默认值
ls	左侧	ACS_VLINE
rs	右侧	ACS_VLINE
ts	顶部	ACS_HLINE
bs	底部	ACS_HLINE
tl	左上角	ACS_ULCORNER
tr	右上角	ACS_URCORNER
bl	左下角	ACS_LLCORNER
br	右下角	ACS_LRCORNER

window.**box**([*vertch*, *horch*])
类似于*border()*，但 *ls* 和 *rs* 均为 *vertch* 而 *ts* 和 *bs* 均为 *horch*。此函数总是会使用默认的转角字符。

window.**chgat**(*attr*)
window.**chgat**(*num*, *attr*)
window.**chgat**(*y*, *x*, *attr*)
window.**chgat**(*y*, *x*, *num*, *attr*)
在当前光标位置或是在所提供的位置（y, x）设置 *num* 个字符的属性。如果 *num* 未给出或为 -1，则将属性设置到所有字符上直至行尾。如果提供了位置（y, x）则此函数会将光标移至该位置。修改过的行将使用*touchline()* 方法处理以便下次窗口刷新时内容会重新显示。

window.**clear**()
类似于*erase()*，但还会导致在下次调用*refresh()* 时整个窗口被重新绘制。

window.**clearok**(*flag*)
如果 *flag* 为 True，则在下次调用*refresh()* 时将完全清除窗口。

window.**clrtobot**()
从光标位置开始擦除直至窗口末端：光标以下的所有行都会被删除，然后会执行*clrtoeol()* 的等效操作。

window.**clrtoeol**()
从光标位置开始擦除直至行尾。

window.**cursyncup**()
更新窗口所有上级窗口的当前光标位置以反映窗口的当前光标位置。

window.**delch**([*y*, *x*])
删除位于（y, x）的任何字符。

window.**deleteln**()
删除在光标之下的行。所有后续的行都会上移一行。

window.**derwin**(*begin_y*, *begin_x*)
window.**derwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
”derive window” 的缩写，*derwin()* 与调用*subwin()* 等效，不同之处在于 *begin_y* 和 *begin_x* 是想对于窗口的初始位置，而不是相对于整个屏幕。返回代表所派生窗口的窗口对象。

window.**echochar**(*ch*[, *attr*])
使用属性 *attr* 添加字符 *ch*，并立即在窗口上调用*refresh()*。

window.**enclose**(*y*, *x*)
检测给定的相对屏幕的字符-单元格坐标是否被给定的窗口所包围，返回 True 或 False。它适用于确定是哪个屏幕窗口子集包围着某个鼠标事件的位置。

window.**encoding**
用于编码方法参数（Unicode 字符串和字符）的编码格式。encoding 属性是在创建子窗口时从父窗口继承的，例如通过*window.subwin()*。默认情况下，会使用当前区域的编码格式 (参见*locale.getpreferredencoding()*)。

3.3 版新加入.

window.**erase**()
清空窗口。

window.**getbegyx**()
返回左上角坐标的元组 (y, x)。

window.**getbkgd**()
返回给定窗口的当前背景字符/属性对。

window.**getch**([*y*, *x*])
获取一个字符。请注意所返回的整数 不一定要在 ASCII 范围以内：功能键、小键盘键等等是由大于 255 的数字表示的。在无延迟模式下，如果没有输入则返回 -1，在其他情况下都会等待直至有键被按下。

window.**get_wch**([*y*, *x*])
获取一个宽字符。对于大多数键都是返回一个字符，对于功能键、小键盘键和其他特殊键则是返回一个整数。在无延迟模式下，如果没有输入则引发一个异常。

3.3 版新加入.

window.**getkey**([*y*, *x*])
获取一个字符，返回一个字符串而不是像*getch()* 那样返回一个整数。功能键、小键盘键和其他特殊键则是返回一个包含键名的多字节字符串。在无延迟模式下，如果没有输入则引发一个异常。

window.**getmaxyx**()
返回窗口高度和宽度的元组 (y, x)。

window.**getparyx**()
将此窗口相对于父窗口的起始坐标作为元组 (y, x) 返回。如果此窗口没有父窗口则返回 (-1, -1)。

window.**getstr**()
window.**getstr**(*n*)
window.**getstr**(*y*, *x*)
window.**getstr**(*y*, *x*, *n*)
从用户读取一个字节串对象，附带基本的行编辑功能。

window.**getyx**()
返回当前光标相对于窗口左上角的位置的元组 (y, x)。

window.**hline**(*ch*, *n*)
window.**hline**(*y*, *x*, *ch*, *n*)
显示一条起始于 (y, x) 长度为 *n* 个字符 *ch* 的水平线。

window.**idcok**(*flag*)
如果 *flag* 为 False，curses 将不再考虑使用终端的硬件插入/删除字符功能；如果 *flag* 为 True，则会启用字符插入和删除。当 curses 首次初始化时，默认会启用字符插入/删除。

window.**idlok**(*flag*)
如果 *flag* 为 True，*curses* 将尝试使用硬件行编辑功能。否则，行插入/删除会被禁用。

window.**immedok**(*flag*)
如果 *flag* 为 True，窗口图像中的任何改变都会自动导致窗口被刷新；你不必再自己调用*refresh()*。但是，这可能会由于重复调用 wrefresh 而显著降低性能。此选项默认被禁用。

window.**inch**([*y*, *x*])
返回窗口中给定位置上的字符。下面的 8 个比特位是字符本身，上面的比特位则为属性。

window.**insch**(*ch*[, *attr*])
window.**insch**(*y*, *x*, *ch*[, *attr*])
将带有属性 *attr* 的字符 *ch* 绘制到 (y, x)，将该行从位置 *x* 开始右移一个字符。

window.**insdelln**(*nlines*)
在指定窗口的当前行上方插入 *nlines* 行。下面的 *nlines* 行将丢失。对于 *nlines* 为负值的情况，则从光标下方的行开始删除 *nlines* 行，并将其余的行向上移动。下面的 *nlines* 行会被清空。当前光标位置将保持不变。

window.**insertln**()
在光标下方插入一个空行。所有后续的行都会下移一行。

window.**insnstr**(*str*, *n*[, *attr*])
window.**insnstr**(*y*, *x*, *str*, *n*[, *attr*])
在光标下方的字符之前插入一个至多为 *n* 个字符的字符串（字符数量将与该行相匹配）。如果 *n* 为零或负数，则插入整个字符串。光标右边的所有字符将被右移，该行右端的字符将丢失。光标位置将保持不变（在移到可能指定的 *y*, *x* 之后）。

window.**insstr**(*str*[, *attr*])
window.**insstr**(*y*, *x*, *str*[, *attr*])
在光标下方的字符之前插入一个字符串（字符数量将与该行相匹配）。光标右边的所有字符将被右移，该行右端的字符将丢失。光标位置将保持不变（在移到可能指定的 *y*, *x* 之后）。

window.**instr**([*n*])
window.**instr**(*y*, *x*[, *n*])
返回从窗口的当前光标位置，或者指定的 *y*, *x* 开始提取的字符所对应的字节串对象。属性会从字符中去除。如果指定了 *n*，*instr()* 将返回长度至多为 *n* 个字符的字符串（不包括末尾的 NUL）。

window.**is_linetouched**(*line*)
如果指定的行自上次调用*refresh()* 后发生了改变则返回 `True`；否则返回 `False`。如果 *line* 对于给定的窗口不可用则会引发*curses.error* 异常。

window.**is_wintouched**()
如果指定的窗口自上次调用*refresh()* 后发生了改变则返回 `True`；否则返回 `False`。

window.**keypad**(*flag*)
如果 *flag* 为 `True`，则某些键（小键盘键、功能键等）生成的转义序列将由*curses* 来解析。如果 *flag* 为 `False`，转义序列将保持在输入流中的原样。

window.**leaveok**(*flag*)
如果 *flag* 为 `True`，则在更新时光标将停留在原地，而不是在“光标位置”。这将可以减少光标的移动。在可能的情况下光标将变为不可见。

如果 *flag* 为 `False`，光标在更新后将总是位于“光标位置”。

window.**move**(*new_y*, *new_x*)
将光标移至 `(new_y, new_x)`。

window.**mvderwin**(*y*, *x*)
让窗口在其父窗口内移动。窗口相对于屏幕的参数不会被更改。此例程用于在屏幕的相同物理位置显示父窗口的不同部分。

window.**mvwin**(*new_y*, *new_x*)
移动窗口以使其左上角位于 `(new_y, new_x)`。

window.**nodelay**(*flag*)
如果 *flag* 为 `True`，则*getch()* 将为非阻塞的。

window.**notimeout**(*flag*)
如果 *flag* 为 `True`，则转义序列将不会发生超时。

如果 *flag* 为 `False`，则在几毫秒之后，转义序列将不会被解析，并将保持在输入流中的原样。

window.**noutrefresh**()
标记为刷新但保持等待。此函数会更新代表预期窗口状态的数据结构，但并不强制更新物理屏幕。要完成后者，请调用*doupdate()*。

window.**overlay**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
将窗口覆盖在 *destwin* 上方。窗口的大小不必相同，只有重叠的区域会被复制。此复制是非破坏性的，这意味着当前背景字符不会覆盖掉 *destwin* 的旧内容。

为了获得对被复制区域的细粒度控制，可以使用*overlay()* 的第二种形式。*sminrow* 和 *smincol* 是源窗口的左上角坐标，而其他变量则在目标窗口中标记出一个矩形。

window.**overwrite**(*destwin*[, *sminrow*, *smincol*, *dminrow*, *dmincol*, *dmaxrow*, *dmaxcol*])
将窗口覆盖在 *destwin* 上方。窗口的大小不必相同，此时只有重叠的区域会被复制。此复制是破坏性的，这意味着当前背景字符会覆盖掉 *destwin* 的旧内容。

为了获得对被复制区域的细粒度控制，可以使用*overwrite()* 的第二种形式。*sminrow* 和 *smincol* 是源窗口的左上角坐标，而其他变量则在目标窗口中标记出一个矩形。

window.**putwin**(*file*)
将关联到窗口的所有数据写入到所提供的文件对象。此信息可在以后使用*getwin()* 函数来提取。

window.**redrawln**(*beg*, *num*)
指明从 *beg* 行开始的 *num* 个屏幕行已被破坏并且应当在下次*refresh()* 调用时完全重绘。

window.**redrawwin**()
触碰整个窗口，以使其在下次*refresh()* 调用时完全重绘。

window.**refresh**([*pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol*])
立即更新显示（将实际屏幕与之前的绘制/删除方法进行同步）。

6 个可选参数仅在窗口为使用*newpad()* 创建的面板时可被指定。需要额外的形参来指定所涉及到的是面板和屏幕的哪一部分。*pminrow* 和 *pmincol* 指定要在面板中显示的矩形的左上角。*sminrow*, *smincol*, *smaxrow* 和 *smaxcol* 指定要在屏幕中显示的矩形的边。要在面板中显示的矩形的右下角是根据屏幕坐标计算出来的，由于矩形的大小必须相同。两个矩形都必须完全包含在其各自的结构之内。负的 *pminrow*, *pmincol*, *sminrow* 或 *smincol* 值会被视为将它们设为零值。

window.**resize**(*nlines*, *ncols*)
为 curses 窗口重新分配存储空间以将其尺寸调整为指定的值。如果任一维度的尺寸大于当前值，则窗口的数据将以具有合并了当前背景渲染（由*bkgdset()* 设置）的空白来填充。

window.**scroll**([*lines=1*])
将屏幕或滚动区域向上滚动 *lines* 行。

window.**scrollok**(*flag*)
控制当一个窗口的光标移出窗口或滚动区域边缘时会发生什么，这可能是在底端行执行换行操作，或者在最后一行输入最后一个字符导致的结果。如果 *flag* 为 `False`，光标会留在底端行。如果 *flag* 为 `True`，窗口会向上滚动一行。请注意为了在终端上获得实际的滚动效果，还需要调用*idlok()*。

window.**setscrreg**(*top*, *bottom*)
设置从 *top* 行至 *bottom* 行的滚动区域。所有滚动操作将在此区域中进行。

window.**standend**()
关闭 standout 属性。在某些终端上此操作会有关闭所有属性的副作用。

window.**standout**()
启用属性 *A_STANDOUT*。

window.**subpad**(*begin_y*, *begin_x*)
window.**subpad**(*nlines*, *ncols*, *begin_y*, *begin_x*)
返回一个子窗口，其左上角位于 `(begin_y, begin_x)`，并且其宽度/高度为 *ncols*/*nlines*。

window.**subwin**(*begin_y*, *begin_x*)
window.**subwin**(*nlines*, *ncols*, *begin_y*, *begin_x*)
返回一个子窗口，其左上角位于 `(begin_y, begin_x)`，并且其宽度/高度为 *ncols*/*nlines*。

默认情况下，子窗口将从指定位置扩展到窗口的右下角。

window.**syncdown**()
触碰已在上级窗口上被触碰的每个位置。此例程由*refresh()* 调用，因此几乎从不需要手动调用。

window.**syncok**(*flag*)
如果 *flag* 为 True，则*syncup()* 会在窗口发生改变的任何时候自动被调用。

window.**syncup**()
触碰已在窗口中被改变的此窗口的各个上级窗口中的所有位置。

window.**timeout**(*delay*)
为窗口设置阻塞或非阻塞读取行为。如果 *delay* 为负值，则会使用阻塞读取（这将无限期地等待输入）。如果 *delay* 为零，则会使用非阻塞读取，并且当没有输入在等待时*getch()* 将返回 -1。如果 *delay* 为正值，则*getch()* 将阻塞 *delay* 毫秒，并且当此延时结束时仍无输入将返回 -1。

window.**touchline**(*start*, *count*[, *changed*])
假定从行 *start* 开始的 *count* 行已被更改。如果提供了 *changed*，它将指明是将受影响的行标记为已更改 (*changed*=True) 还是未更改 (*changed*=False)。

window.**touchwin**()
假定整个窗口已被更改，其目的是用于绘制优化。

window.**untouchwin**()
将自上次调用*refresh()* 以来窗口中的所有行标记为未改变。

window.**vline**(*ch*, *n*)
window.**vline**(*y*, *x*, *ch*, *n*)
显示一条起始于 (y, x) 长度为 *n* 个字符 *ch* 的垂直线。

16.10.3 常量

curses 模块定义了以下数据成员:

curses.**ERR**
一些返回整数的 curses 例程，例如*getch()*，在失败时将返回*ERR*。

curses.**OK**
一些返回整数的 curses 例程，例如*napms()*，在成功时将返回*OK*。

curses.**version**
一个代表当前模块版本的字节串对象。也作 __version__。

curses.**ncurses_version**
一个具名元组，它包含构成 ncurses 库版本号的三个数字: *major*, *minor* 和 *patch*。三个值均为整数。三个值也可通过名称来访问，因此 curses.ncurses_version[0] 等价于 curses.ncurses_version.major，依此类推。

可用性：如果使用了 ncurses 库。

3.8 版新加入.

有些常量可用于指定字符单元属性。实际可用的常量取决于具体的系统。

属性	含义
A_ALTCHARSET	备用字符集模式
A_BLINK	闪烁模式
A_BOLD	粗体模式
A_DIM	暗淡模式
A_INVIS	不可见或空白模式
A_ITALIC	斜体模式
A_NORMAL	正常属性
A_PROTECT	保护模式
A_REVERSE	反转背景色和前景色
A_STANDOUT	突出模式
A_UNDERLINE	下划线模式
A_HORIZONTAL	水平突出显示
A_LEFT	左高亮
A_LOW	底部高亮
A_RIGHT	右高亮
A_TOP	顶部高亮
A_VERTICAL	垂直突出显示
A_CHARTEXT	用于提取字符的位掩码

3.7 版新加入: A_ITALIC was added.

有几个常量可用于提取某些方法返回的相应属性。

位掩码	含义
A_ATTRIBUTES	用于提取属性的位掩码
A_CHARTEXT	用于提取字符的位掩码
A_COLOR	用于提取颜色对字段信息的位掩码

键由名称以 KEY_ 开头的整数常量引用。确切的可用键取决于系统。

关键常数	键
KEY_MIN	最小键值
KEY_BREAK	中断键（不可靠）
KEY_DOWN	向下箭头
KEY_UP	向上箭头
KEY_LEFT	向左箭头
KEY_RIGHT	向右箭头
KEY_HOME	Home 键 (上 + 左箭头)
KEY_BACKSPACE	退格（不可靠）
KEY_F0	功能键。支持至多 64 个功能键。
KEY_Fn	功能键 *n* 的值
KEY_DL	删除行
KEY_IL	插入行
KEY_DC	删除字符
KEY_IC	插入字符或进入插入模式
KEY_EIC	退出插入字符模式
KEY_CLEAR	清空屏幕
KEY_EOS	清空至屏幕底部
KEY_EOL	清空至行尾

繼續下一頁

表 1 – 繼續上一頁

关键常数	键
KEY_SF	向前滚动 1 行
KEY_SR	向后滚动 1 行 (反转)
KEY_NPAGE	下一页
KEY_PPAGE	上一页
KEY_STAB	设置制表符
KEY_CTAB	清除制表符
KEY_CATAB	清除所有制表符
KEY_ENTER	回车或发送 (不可靠)
KEY_SRESET	软 (部分) 重置 (不可靠)
KEY_RESET	重置或硬重置 (不可靠)
KEY_PRINT	打印
KEY_LL	Home 向下或到底 (左下)
KEY_A1	键盘的左上角
KEY_A3	键盘的右上角
KEY_B2	键盘的中心
KEY_C1	键盘左下方
KEY_C3	键盘右下方
KEY_BTAB	回退制表符
KEY_BEG	Beg (开始)
KEY_CANCEL	取消
KEY_CLOSE	关闭
KEY_COMMAND	Cmd (命令行)
KEY_COPY	复制
KEY_CREATE	创建
KEY_END	End
KEY_EXIT	退出
KEY_FIND	查找
KEY_HELP	帮助
KEY_MARK	标记
KEY_MESSAGE	消息
KEY_MOVE	移动
KEY_NEXT	下一个
KEY_OPEN	打开
KEY_OPTIONS	选项
KEY_PREVIOUS	Prev (上一个)
KEY_REDO	重做
KEY_REFERENCE	Ref (引用)
KEY_REFRESH	刷新
KEY_REPLACE	替换
KEY_RESTART	重启
KEY_RESUME	恢复
KEY_SAVE	保存
KEY_SBEG	Shift + Beg (开始)
KEY_SCANCEL	Shift + Cancel
KEY_SCOMMAND	Shift + Command
KEY_SCOPY	Shift + Copy
KEY_SCREATE	Shift + Create
KEY_SDC	Shift + 删除字符
KEY_SDL	Shift + 删除行

繼續下一頁

表 1 – 繼續上一頁

关键常数	键
KEY_SELECT	选择
KEY_SEND	Shift + End
KEY_SEOL	Shift + 清空行
KEY_SEXIT	Shift + 退出
KEY_SFIND	Shift + 查找
KEY_SHELP	Shift + 帮助
KEY_SHOME	Shift + Home
KEY_SIC	Shift + 输入
KEY_SLEFT	Shift + 向左箭头
KEY_SMESSAGE	Shift + 消息
KEY_SMOVE	Shift + 移动
KEY_SNEXT	Shift + 下一个
KEY_SOPTIONS	Shift + 选项
KEY_SPREVIOUS	Shift + 上一个
KEY_SPRINT	Shift + 打印
KEY_SREDO	Shift + 重做
KEY_SREPLACE	Shift + 替换
KEY_SRIGHT	Shift + 向右箭头
KEY_SRSUME	Shift + 恢复
KEY_SSAVE	Shift + 保存
KEY_SSUSPEND	Shift + 挂起
KEY_SUNDO	Shift + 撤销
KEY_SUSPEND	挂起
KEY_UNDO	撤销操作
KEY_MOUSE	鼠标事件已发生
KEY_RESIZE	终端大小改变事件
KEY_MAX	最大键值

在 VT100 及其软件仿真（例如 X 终端仿真器）上，通常至少有四个功能键（KEY_F1, KEY_F2, KEY_F3, KEY_F4 ）可用，并且箭头键以明显的方式映射到 KEY_UP, KEY_DOWN, KEY_LEFT 和 KEY_RIGHT 。如果您的机器有一个 PC 键盘，可以安全地使用箭头键和十二个功能键（旧的 PC 键盘可能只有十个功能键）；此外，以下键盘映射是标准的：

键帽	常数
Insert	KEY_IC
Delete	KEY_DC
Home	KEY_HOME
End	KEY_END
Page Up	KEY_PPAGE
Page Down	KEY_NPAGE

下表列出了替代字符集中的字符。这些字符继承自 VT100 终端，在 X 终端等软件模拟器上通常均为可用。当没有可用的图形时，curses 会回退为粗糙的可打印 ASCII 近似符号。

備F: 只有在调用 *initscr()* 之后才能使用它们

ACS 代码	含义
ACS_BBSS	右上角的别名
ACS_BLOCK	实心方块
ACS_BOARD	正方形
ACS_BSBS	水平线的别名
ACS_BSSB	左上角的别名
ACS_BSSS	顶部 T 型的别名
ACS_BTEE	底部 T 型
ACS_BULLET	正方形
ACS_CKBOARD	棋盘（点刻）
ACS_DARROW	向下箭头
ACS_DEGREE	等级符
ACS_DIAMOND	菱形
ACS_GEQUAL	大于或等于
ACS_HLINE	水平线
ACS_LANTERN	灯形符号
ACS_LARROW	向左箭头
ACS_LEQUAL	小于或等于
ACS_LLCORNER	左下角
ACS_LRCORNER	右下角
ACS_LTEE	左侧 T 型
ACS_NEQUAL	不等号
ACS_PI	字母 π
ACS_PLMINUS	正负号
ACS_PLUS	加号
ACS_RARROW	向右箭头
ACS_RTEE	右侧 T 型
ACS_S1	扫描线 1
ACS_S3	扫描线 3
ACS_S7	扫描线 7
ACS_S9	扫描线 9
ACS_SBBS	右下角的别名
ACS_SBSB	垂直线的别名
ACS_SBSS	右侧 T 型的别名
ACS_SSBB	左下角的别名
ACS_SSBS	底部 T 型的别名
ACS_SSSB	左侧 T 型的别名
ACS_SSSS	交叉或大加号的替代名称
ACS_STERLING	英镑
ACS_TTEE	顶部 T 型
ACS_UARROW	向上箭头
ACS_ULCORNER	左上角
ACS_URCORNER	右上角
ACS_VLINE	垂线

下表列出了预定义的颜色：

常数	颜色
COLOR_BLACK	黑色
COLOR_BLUE	蓝色
COLOR_CYAN	青色（浅绿蓝色）
COLOR_GREEN	绿色
COLOR_MAGENTA	洋红色（紫红色）
COLOR_RED	红色
COLOR_WHITE	白色
COLOR_YELLOW	黄色

16.11 curses.textpad --- 用于 curses 程序的文本输入控件

curses.textpad 模块提供了一个 *Textbox* 类，该类在 curses 窗口中处理基本的文本编辑，支持一组与 Emacs 类似的键绑定（因此这也适用于 Netscape Navigator, BBedit 6.x, FrameMaker 和许多其他程序）。该模块还提供了一个绘制矩形的函数，适用于容纳文本框或其他目的。

curses.textpad 模块定义了以下函数:

curses.textpad.**rectangle**(*win*, *uly*, *ulx*, *lry*, *lrx*)
绘制一个矩形。第一个参数必须为窗口对象；其余参数均为相对于该窗口的坐标值。第二和第三个参数为要绘制的矩形的左上角的 y 和 x 坐标值；第四和第五个参数为其右下角的 y 和 x 坐标值。将会使用 VT100/IBM PC 形式的字符在可用的终端上（包括 xterm 和大多数其他软件终端模拟器）绘制矩形。在其他情况下则将使用 ASCII 横杠、竖线和加号绘制。

16.11.1 文本框对象

你可以通过如下方式实例化一个 *Textbox*:

class curses.textpad.**Textbox**(*win*)
返回一个文本框控件对象。*win* 参数必须是一个 curses 窗口 对象，文本框将被包含在其中。文本框的编辑光标在初始时位于包含窗口的左上角，坐标值为 (0, 0)。实例的 *stripspaces* 旗标初始时为启用。

Textbox 对象具有以下方法:

edit([*validator*])
这是你通常将使用的入口点。它接受编辑按键直到键入了一个终止按键。如果提供了 *validator*，它必须是一个函数。它将在每次按键时被调用并传入相应的按键作作为形参；命令发送将在结果上执行。此方法会以字符串形式返回窗口内容；是否包括窗口中的空白将受到 *stripspaces* 属性的影响。

do_command(*ch*)
处理单个按键命令。以下是支持的特殊按键:

按键	动作
Control-A	转到窗口的左边缘。
Control-B	光标向左，如果可能，包含前一行。
Control-D	删除光标下的字符。
Control-E	前往右边缘（stripspaces 关闭时）或者行尾（stripspaces 启用时）。
Control-F	向右移动光标，适当时换行到下一行。
Control-G	终止，返回窗口内容。
Control-H	向后删除字符。
Control-J	如果窗口是 1 行则终止，否则插入换行符。
Control-K	如果行为空，则删除它，否则清除到行尾。
Control-L	刷新屏幕。
Control-N	光标向下; 向下移动一行。
Control-O	在光标位置插入一个空行。
Control-P	光标向上; 向上移动一行。

如果光标位于无法移动的边缘，则移动操作不执行任何操作。在可能的情况下，支持以下同义词:

常数	按键
KEY_LEFT	Control-B
KEY_RIGHT	Control-F
KEY_UP	Control-P
KEY_DOWN	Control-N
KEY_BACKSPACE	Control-h

所有其他按键将被视为插入给定字符并右移的命令（带有自动折行）。

gather()
以字符串形式返回窗口内容；是否包括窗口中的空白将受到*stripspaces*成员的影响。

stripspaces
此属性是控制窗口中空白解读方式的旗标。当启用时，每一行的末尾空白会被忽略；任何将光标定位至末尾空白的光标动作都将改为前往该行末尾，并且在收集窗口内容时将去除末尾空白。

16.12 curses.ascii --- 用于 ASCII 字符的工具

curses.ascii 模块提供了一些 ASCII 字符的名称常量以及在各种 ASCII 字符类中执行成员检测的函数。所提供的控制字符常量如下:

名称	含义
NUL	
SOH	标题开始，控制台中断
STX	文本开始
ETX	文本结束
EOT	传输结束
ENQ	查询，附带 ACK 流量控制
ACK	确认
BEL	蜂鸣器
BS	退格

繼續下一頁

表 3 – 繼續上一頁

名称	含义
TAB	制表符
HT	TAB 的别名：”水平制表符”
LF	换行
NL	LF 的别名：”新行”
VT	垂直制表符
FF	换页
CR	回车
SO	Shift-out，开始替换字符集
SI	Shift-in，恢复默认字符集
DLE	Data-link escape，数据链接转义
DC1	XON，用于流程控制
DC2	Device control 2，块模式流程控制
DC3	XOFF，用于流程控制
DC4	设备控制 4
NAK	否定确认
SYN	同步空闲
ETB	末端传输块
CAN	取消
EM	媒体结束
SUB	替换
ESC	退出
FS	文件分隔符
GS	组分隔符
RS	Record separator，块模式终止符
US	单位分隔符
SP	空格
DEL	删除

请注意其中有许多在现今已经没有实际作用。这些助记符是来源于数字计算机之前的电传打印机规范。

此模块提供了下列函数，对应于标准 C 库中的函数:

curses.ascii.**isalnum**(*c*)
检测 ASCII 字母数字类字符；它等价于 `isalpha(c) 或 isdigit(c)`。

curses.ascii.**isalpha**(*c*)
检测 ASCII 字母类字符；它等价于 `isupper(c) or islower(c)`。

curses.ascii.**isascii**(*c*)
检测字符值是否在 7 位 ASCII 集范围内。

curses.ascii.**isblank**(*c*)
检测 ASCII 空白字符；包括空格或水平制表符。

curses.ascii.**iscntrl**(*c*)
检测 ASCII 控制字符（在 0x00 到 0x1f 或 0x7f 范围内）。

curses.ascii.**isdigit**(*c*)
检测 ASCII 十进制数码，即 `'0'` 至 `'9'`。它等价于 `c in string.digits`。

curses.ascii.**isgraph**(*c*)
检测任意 ASCII 可打印字符，不包括空白符。

curses.ascii.**islower**(*c*)
检测 ASCII 小写字母字符。

curses.ascii.**isprint**(*c*)
检测任意 ASCII 可打印字符，包括空白符。

curses.ascii.**ispunct**(*c*)
检测任意 ASCII 可打印字符，不包括空白符或字母数字类字符。

curses.ascii.**isspace**(*c*)
检测 ASCII 空白字符；包括空格，换行，回车，进纸，水平制表和垂直制表。

curses.ascii.**isupper**(*c*)
检测 ASCII 大写字母字符。

curses.ascii.**isxdigit**(*c*)
检测 ASCII 十六进制数码。这等价于 `c in string.hexdigits`。

curses.ascii.**isctrl**(*c*)
检测 ASCII 控制字符（码位值 0 至 31）。

curses.ascii.**ismeta**(*c*)
检测非 ASCII 字符（码位值 0x80 及以上）。

这些函数接受整数或单字符字符串；当参数为字符串时，会先使用内置函数*ord()* 进行转换。

请注意所有这些函数都是检测根据你传入的字符串的字符所生成的码位值；它们实际上完全不会知晓本机的字符编码格式。

以下两个函数接受单字符字符串或整数形式的字节值；它们会返回相同类型的值。

curses.ascii.**ascii**(*c*)
返回对应于 *c* 的下个 7 比特位的 ASCII 值。

curses.ascii.**ctrl**(*c*)
返回对应于给定字符的控制字符（字符比特值会与 0x1f 进行按位与运算）。

curses.ascii.**alt**(*c*)
返回对应于给定 ASCII 字符的 8 比特位字符（字符比特值会与 0x80 进行按位或运算）。

以下函数接受单字符字符串或整数值；它会返回一个字符串。

curses.ascii.**unctrl**(*c*)
返回 ASCII 字符 *c* 的字符串表示形式。如果 *c* 是可打印字符，则字符串为字符本身。如果该字符是控制字符 (0x00--0x1f) 则字符串由一个插入符 (`'^'`) 加相应的大写字母组成。如果该字符是 ASCII 删除符 (0x7f) 则字符串为 `'^?'`。如果该字符设置了元比特位 (0x80)，元比特位会被去除，应用以上规则后将在结果之前添加 `'!'`。

curses.ascii.**controlnames**
一个 33 元素的字符串数据，其中按从 0 (NUL) 到 0x1f (US) 的顺序包含了三十二个 ASCII 控制字符的 ASCII 助记符，另加空格符的助记符 `SP`。

16.13 `curses.panel` --- curses 的面板栈扩展

面板是具有添加深度功能的窗口，因此它们可以从上至下堆叠为栈，只有显示每个窗口的可见部分会显示出来。面板可以在栈中被添加、上移或下移，也可以被移除。

16.13.1 函数

curses.panel 模块定义了以下函数:

curses.panel.**bottom_panel**()
返回面板栈中的底部面板。

curses.panel.**new_panel**(*win*)
返回一个面板对象，将其与给定的窗口 *win* 相关联。请注意你必须显式地保持所返回的面板对象。如果你不这样做，面板对象会被垃圾回收并从面板栈中被移除。

curses.panel.**top_panel**()
返回面板栈中的顶部面板。

curses.panel.**update_panels**()
在面板栈发生改变后更新虚拟屏幕。这不会调用*curses.doupdate()*，因此你不必自己执行此操作。

16.13.2 Panel 对象

Panel 对象，如上面*new_panel()* 所返回的对象，是带有栈顺序的多个窗口。总是会有一个窗口与确定内容的面板相关联，面板方法会负责窗口在面板栈中的深度。

Panel 对象具有以下方法:

Panel.**above**()
返回当前面板之上的面板。

Panel.**below**()
返回当前面板之下的面板。

Panel.**bottom**()
将面板推至栈底部。

Panel.**hidden**()
如果面板被隐藏（不可见）则返回 `True`，否则返回 `False`。

Panel.**hide**()
隐藏面板。这不会删除对象，它只是让窗口在屏幕上不可见。

Panel.**move**(*y*, *x*)
将面板移至屏幕坐标 `(y, x)`。

Panel.**replace**(*win*)
将与面板相关联的窗口改为窗口 *win*。

Panel.**set_userptr**(*obj*)
将面板的用户指向设为 *obj*。这被用来将任意数据与面板相关联，数据可以是任何 Python 对象。

Panel.**show**()
显示面板（面板可能已被隐藏）。

Panel.**top**()
将面板推至栈顶部。

Panel.**userptr**()
返回面板的用户指针。这可以是任何 Python 对象。

Panel.**window**()
返回与面板相关联的窗口对象。

16.14 platform --- 获取底层平台的标识数据

源代码： Lib/platform.py

備𝔽： 特定平台按字母顺序排列，Linux 包括在 Unix 小节之中。

16.14.1 跨平台

platform.**architecture**(*executable=sys.executable*, *bits=''*, *linkage=''*)
查询给定的可执行文件（默认为 Python 解释器二进制码文件）来获取各种架构信息。

返回一个元素 (bits, linkage)，其中包含可执行文件所使用的位架构和链接格式信息。这两个值均以字符串形式返回。

无法确定的值将返回为形参预设所给出的值。如果给出的位数为 ''，则会使用 sizeof(pointer) (或者当 Python 版本 < 1.5.2 时为 sizeof(long)) 作为所支持的指针大小的提示。

此函数依赖于系统的 file 命令来执行实际的操作。这在几乎所有 Unix 平台和某些非 Unix 平台上只有当可执行文件指向 Python 解释器时才可用。当以上要求不满足时将会使用合理的默认值。

備𝔽： 在 macOS (也许还有其他平台) 上，可执行文件可能是包含多种架构的通用文件。

要获取当前解释器的“64 位性”，更可靠的做法是查询*sys.maxsize* 属性:

```
is_64bits = sys.maxsize > 2**32
```

---

platform.**machine**()
返回机器类型，例如 'i386'。如果该值无法确定则会返回一个空字符串。

platform.**node**()
返回计算机的网络名称（可能不是完整限定名称!）。如果该值无法确定则会返回一个空字符串。

platform.**platform**(*aliased=0*, *terse=0*)
返回一个标识底层平台的字符串，其中带有尽可能多的有用信息。

输出信息的目标是“人类易读”而非机器易解析。它在不同平台上可能看起来不一致，这是有意为之的。

如果 *aliased* 为真值，此函数将使用各种平台不同与其通常名称的别名来报告系统名称，例如 SunOS 将被报告为 Solaris。*system_alias()* 函数将被用于实现此功能。

将 *terse* 设为真值将导致此函数只返回标识平台所必须的最小量信息。

3.8 版更變: 在 macOS 上，此函数现在会在*mac_ver()* 返回的发布版字符串非空时使用它，以便获取 macOS 版本而非 darwin 版本。

platform.**processor**()
返回（真实的）处理器名称，例如 'amdk6'。

如果该值无法确定则将返回空字符串。请注意许多平台都不提供此信息或是简单地返回与*machine()* 相同的值。NetBSD 则会提供此信息。

platform.**python_build**()
返回一个元组 (buildno, builddate)，以字符串表示的 Python 编译代码和日期。

platform.**python_compiler**()
返回一个标识用于编译 Python 的编译器的的字符串。

platform.**python_branch**()
返回一个标识 Python 实现的 SCM 分支的字符串。

platform.**python_implementation**()
返回一个标识 Python 实现的字符串。可能的返回值有: 'CPython', 'IronPython', 'Jython', 'PyPy'。

platform.**python_revision**()
返回一个标识 Python 实现的 SCM 修订版的字符串。

platform.**python_version**()
将 Python 版本以字符串 `'major.minor.patchlevel'` 形式返回。

请注意此返回值不同于 Python `sys.version`，它将总是包括 patchlevel (默认为 0)。

platform.**python_version_tuple**()
将 Python 版本以字符串元组 `(major, minor, patchlevel)` 形式返回。

请注意此返回值不同于 Python `sys.version`，它将总是包括 patchlevel (默认为 `'0'`)。

platform.**release**()
Returns the system's release, e.g. `'2.2.0'` or `'NT'`. An empty string is returned if the value cannot be determined.

platform.**system**()
返回系统平台/OS 的名称，例如 `'Linux'`, `'Darwin'`, `'Java'`, `'Windows'`。如果该值无法确定则将返回一个空字符串。

platform.**system_alias**(*system*, *release*, *version*)
返回别名为某些系统所使用的常见营销名称的 `(system, release, version)`。它还会在可能导致混淆的情况下对信息进行一些重排序操作。

platform.**version**()
返回系统的发布版本信息，例如 `'#3 on degas'`。如果该值无法确定则将返回一个空字符串。

platform.**uname**()
具有高可移植性的 uname 接口。返回包含六个属性的*namedtuple()*: *system*, *node*, *release*, *version*, *machine* 和*processor*。

请注意此函数添加的第六个属性 (*processor*) 并不存在于*os.uname()* 的结果中。并且前两个属性的属性名称也不一致；*os.uname()* 是将它们称为 sysname 和 nodename。

无法确定的条目会被设为 `''`。

3.3 版更變: Result changed from a tuple to a *namedtuple()*.

16.14.2 Java 平台

platform.**java_ver**(*release=''*, *vendor=''*, *vminfo=('', '', '')*, *osinfo=('', '', '')*)
Jython 的版本接口

返回一个元组 `(release, vendor, vminfo, osinfo)`，其中 *vminfo* 为元组 `(vm_name, vm_release, vm_vendor)` 而 *osinfo* 为元组 `(os_name, os_version, os_arch)`。无法确定的值将设为由形参所给出的默认值 (默认均为 `''`)。

16.14.3 Windows 平台

platform.**win32_ver**(*release=''*, *version=''*, *csd=''*, *ptype=''*)
Get additional version information from the Windows Registry and return a tuple `(release, version, csd, ptype)` referring to OS release, version number, CSD level (service pack) and OS type (multi/single processor). Values which cannot be determined are set to the defaults given as parameters (which all default to an empty string).

一点提示: *ptype* 在单个处理器的 NT 机器上为 `'Uniprocessor Free'` 而在多个处理器的机器上为 `'Multiprocessor Free'`。*'Free'* 是指该 OS 版本没有调试代码。它还可能显示 *'Checked'* 表示该 OS 版本使用了调试代码，即检测参数、范围等的代码。

platform.**win32_edition**()
Returns a string representing the current Windows edition, or `None` if the value cannot be determined. Possible values include but are not limited to `'Enterprise'`, `'IoTUAP'`, `'ServerStandard'`, and `'nanoserver'`.

3.8 版新加入.

platform.**win32_is_iot**()
如果*win32_edition()* 返回的 Windows 版本被识别为 IoT 版则返回 `True`。

3.8 版新加入.

16.14.4 macOS Platform

platform.**mac_ver**(*release=''*, *versioninfo=('', '', '')*, *machine=''*)
Get macOS version information and return it as tuple `(release, versioninfo, machine)` with *versioninfo* being a tuple `(version, dev_stage, non_release_version)`.

无法确定的条目会被设为 `''`。所有元组条目均为字符串。

16.14.5 Unix 平台

platform.**libc_ver**(*executable=sys.executable*, *lib=''*, *version=''*, *chunksize=16384*)
尝试确定可执行文件（默认为 Python 解释器）所链接到的 libc 版本。返回一个字符串元组 `(lib, version)`，当查找失败时其默认值将设为给定的形参值。

请注意此函数对于不同 libc 版本向可执行文件添加符号的方式有深层的关联，可能仅适用于使用 **gcc** 编译出来的可执行文件。

文件将按 *chunksize* 个字节的分块来读取和扫描。

16.15 errno --- 标准 errno 系统符号

This module makes available standard `errno` system symbols. The value of each symbol is the corresponding integer value. The names and descriptions are borrowed from `linux/include/errno.h`, which should be all-inclusive.

errno.**errorcode**
提供从 errno 值到底层系统中字符串名称的映射的字典。例如，`errno.errorcode[errno.EPERM]` 映射为 `'EPERM'` 。

如果要将数字的错误代码转换为错误信息，请使用*os.strerror()*。

在下面的列表中，当前平台上没有使用的符号没有被本模块定义。已定义的符号的具体列表可参见 errno.errorcode.keys()。可用的符号包括：

errno.**EPERM**
Operation not permitted. This error is mapped to the exception *PermissionError*.

errno.**ENOENT**
No such file or directory. This error is mapped to the exception *FileNotFoundError*.

errno.**ESRCH**
No such process. This error is mapped to the exception *ProcessLookupError*.

errno.**EINTR**
Interrupted system call. This error is mapped to the exception *InterruptedError*.

errno.**EIO**
I/O 错误

errno.**ENXIO**
无此设备或地址

errno.**E2BIG**
参数列表过长

errno.**ENOEXEC**
执行格式错误

errno.**EBADF**
错误的文件号

errno.**ECHILD**
No child processes. This error is mapped to the exception *ChildProcessError*.

errno.**EAGAIN**
Try again. This error is mapped to the exception *BlockingIOError*.

errno.**ENOMEM**
内存不足

errno.**EACCES**
Permission denied. This error is mapped to the exception *PermissionError*.

errno.**EFAULT**
错误的地址

errno.**ENOTBLK**
需要块设备

errno.**EBUSY**
设备或资源忙

errno.**EEXIST**
File exists. This error is mapped to the exception *FileExistsError*.

errno.**EXDEV**
跨设备链接

errno.**ENODEV**
无此设备

errno.**ENOTDIR**
Not a directory. This error is mapped to the exception *NotADirectoryError*.

errno.**EISDIR**
Is a directory. This error is mapped to the exception *IsADirectoryError*.

errno.**EINVAL**
无效的参数

errno.**ENFILE**
文件表溢出

errno.**EMFILE**
打开的文件过多

errno.**ENOTTY**
不是打字机

errno.**ETXTBSY**
文本文件忙

errno.**EFBIG**
文件过大

errno.**ENOSPC**
设备已无可用空间

errno.**ESPIPE**
非法查找

errno.**EROFS**
只读文件系统

errno.**EMLINK**
链接过多

errno.**EPIPE**
Broken pipe. This error is mapped to the exception *BrokenPipeError*.

errno.**EDOM**
数学参数超出函数范围

errno.**ERANGE**
数学运算结果无法表示

errno.**EDEADLK**
将发生资源死锁

errno.**ENAMETOOLONG**
文件名过长

errno.**ENOLCK**
没有可用的记录锁

errno.**ENOSYS**
功能未实现

errno.**ENOTEMPTY**
目录非空

errno.**ELOOP**
遇到过多的符号链接

errno.**EWOULDBLOCK**
Operation would block. This error is mapped to the exception *BlockingIOError*.

errno.**ENOMSG**
没有所需类型的消息

errno.**EIDRM**
标识符被移除

errno.**ECHRNG**
信道编号超出范围

errno.**EL2NSYNC**
级别 2 未同步

errno.**EL3HLT**
级别 3 已停止

errno.**EL3RST**
级别 3 重置

errno.**ELNRNG**
链接编号超出范围

errno.**EUNATCH**
未附加协议驱动

errno.**ENOCSI**
没有可用的 CSI 结构

errno.**EL2HLT**
级别 2 已停止

errno.**EBADE**
无效的交换

errno.**EBADR**
无效的请求描述符

errno.**EXFULL**
交换已满

errno.**ENOANO**
没有阳极

errno.**EBADRQC**
无效的请求码 ·

errno.**EBADSLT**
无效的槽位

errno.**EDEADLOCK**
文件锁定死锁错误

errno.**EBFONT**
错误的字体文件格式

errno.**ENOSTR**
设备不是流

errno.**ENODATA**
没有可用的数据

errno.**ETIME**
计时器已到期

errno.**ENOSR**
流资源不足

errno.**ENONET**
机器不在网络上

errno.**ENOPKG**
包未安装

errno.**EREMOTE**
对象是远程的

errno.**ENOLINK**
链接已被切断

errno.**EADV**
广告错误

errno.**ESRMNT**
挂载错误

errno.**ECOMM**
发送时通讯错误

errno.**EPROTO**
协议错误

errno.**EMULTIHOP**
已尝试多跳

errno.**EDOTDOT**
RFS 专属错误

errno.**EBADMSG**
非数据消息

errno.**EOVERFLOW**
值相对于已定义数据类型过大

errno.**ENOTUNIQ**
名称在网络上不唯一

errno.**EBADFD**
文件描述符处于错误状态

errno.**EREMCHG**
远端地址已改变

errno.**ELIBACC**
无法访问所需的共享库

errno.**ELIBBAD**
访问已损坏的共享库

errno.**ELIBSCN**
a.out 中的.lib 部分已损坏

errno.**ELIBMAX**
尝试链接过多的共享库

errno.**ELIBEXEC**
无法直接执行共享库

errno.**EILSEQ**
非法字节序列

errno.**ERESTART**
已中断系统调用需要重启

errno.**ESTRPIPE**
流管道错误

errno.**EUSERS**
用户过多

errno.**ENOTSOCK**
在非套接字上执行套接字操作

errno.**EDESTADDRREQ**
需要目标地址

errno.**EMSGSIZE**
消息过长

errno.**EPROTOTYPE**
套接字的协议类型错误

errno.**ENOPROTOOPT**
协议不可用

errno.**EPROTONOSUPPORT**
协议不受支持

errno.**ESOCKTNOSUPPORT**
套接字类型不受支持

errno.**EOPNOTSUPP**
操作在传输端点上不受支持

errno.**EPFNOSUPPORT**
协议族不受支持

errno.**EAFNOSUPPORT**
地址族不受协议支持

errno.**EADDRINUSE**
地址已被使用

errno.**EADDRNOTAVAIL**
无法分配要求的地址

errno.**ENETDOWN**
网络已断开

errno.**ENETUNREACH**
网络不可达

errno.**ENETRESET**
网络因重置而断开连接

errno.**ECONNABORTED**
Software caused connection abort. This error is mapped to the exception *ConnectionAbortedError*.

errno.**ECONNRESET**
Connection reset by peer. This error is mapped to the exception *ConnectionResetError*.

errno.**ENOBUFS**
没有可用的缓冲区空间

errno.**EISCONN**
传输端点已连接

errno.**ENOTCONN**
传输端点未连接

errno.**ESHUTDOWN**
Cannot send after transport endpoint shutdown. This error is mapped to the exception *BrokenPipeError*.

errno.**ETOOMANYREFS**
引用过多：无法拼接

errno.**ETIMEDOUT**
Connection timed out. This error is mapped to the exception *TimeoutError*.

errno.**ECONNREFUSED**
Connection refused. This error is mapped to the exception *ConnectionRefusedError*.

errno.**EHOSTDOWN**
主机已关闭

errno.**EHOSTUNREACH**
没有到主机的路由

errno.**EALREADY**
Operation already in progress. This error is mapped to the exception *BlockingIOError*.

errno.**EINPROGRESS**
Operation now in progress. This error is mapped to the exception *BlockingIOError*.

errno.**ESTALE**
过期的 NFS 文件句柄

errno.**EUCLEAN**
结构需要清理

errno.**ENOTNAM**
不是 XENIX 命名类型文件

errno.**ENAVAIL**
没有可用的 XENIX 信标

errno.**EISNAM**
是命名类型文件

errno.**EREMOTEIO**
远程 I/O 错误

errno.**EDQUOT**
超出配额

16.16 ctypes --- Python 的外部函数库

ctypes 是 Python 的外部函数库。它提供了与 C 兼容的数据类型，并允许调用 DLL 或共享库中的函数。可使用该模块以纯 Python 形式对这些库进行封装。

16.16.1 ctypes 教程

Note: The code samples in this tutorial use *doctest* to make sure that they actually work. Since some code samples behave differently under Linux, Windows, or macOS, they contain doctest directives in comments.

注意：部分示例代码引用了 ctypes *c_int* 类型。在 `sizeof(long) == sizeof(int)` 的平台上此类型是*c_long* 的一个别名。所以，在程序输出*c_long* 而不是你期望的*c_int* 时不必感到迷惑 --- 它们实际上是同一种类型。

载入动态连接库

ctypes 导出了 *cdll* 对象，在 Windows 系统中还导出了 *windll* 和 *oledll* 对象用于载入动态连接库。

通过操作这些对象的属性，你可以载入外部的动态链接库。*cdll* 载入按标准的 `cdecl` 调用协议导出的函数，而 *windll* 导入的库按 `stdcall` 调用协议调用其中的函数。*oledll* 也按 `stdcall` 调用协议调用其中的函数，并假定该函数返回的是 Windows `HRESULT` 错误代码，并当函数调用失败时，自动根据该代码甩出一个*OSError* 异常。

3.3 版更變: 原来在 Windows 下抛出的异常类型*WindowsError* 现在是*OSError* 的一个别名。

这是一些 Windows 下的例子。注意：`msvcrt` 是微软 C 标准库，包含了大部分 C 标准函数，这些函数都是以 cdecl 调用协议进行调用的。

```
>>> from ctypes import *
>>> print(windll.kernel32)
<WinDLL 'kernel32', handle ... at ...>
>>> print(cdll.msvcrt)
<CDLL 'msvcrt', handle ... at ...>
>>> libc = cdll.msvcrt
>>>
```

Windows 会自动添加通常的 `.dll` 文件扩展名。

備F: 通过 `cdll.msvcrt` 调用的标准 C 函数，可能会导致调用一个过时的，与当前 Python 所不兼容的函数。因此，请尽量使用标准的 Python 函数，而不要使用 `msvcrt` 模块。

在 Linux 下，必须使用 包含文件扩展名的文件名来导入共享库。因此不能简单使用对象属性的方式来导入库。因此，你可以使用方法 `LoadLibrary()`，或构造 CDLL 对象来导入库。

```
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle ... at ...>
>>> libc = CDLL("libc.so.6")
>>> libc
<CDLL 'libc.so.6', handle ... at ...>
>>>
```

操作导入的动态链接库中的函数

通过操作 dll 对象的属性来操作这些函数。

```
>>> from ctypes import *
>>> libc.printf
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.GetModuleHandleA)
<_FuncPtr object at 0x...>
>>> print(windll.kernel32.MyOwnFunction)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 239, in __getattr__
    func = _StdcallFuncPtr(name, self)
AttributeError: function 'MyOwnFunction' not found
>>>
```

注意：Win32 系统的动态库，比如 `kernel32` 和 `user32`，通常会同时导出同一个函数的 ANSI 版本和 UNICODE 版本。UNICODE 版本通常会在名字最后以 `W` 结尾，而 ANSI 版本的则以 `A` 结尾。win32 的 `GetModuleHandle` 函数会根据一个模块名返回一个 *模块句柄*，该函数暨同时包含这样的两个版本的原型函数，并通过宏 UNICODE 是否定义，来决定宏 `GetModuleHandle` 导出的是哪个具体函数。

```
/* ANSI version */
HMODULE GetModuleHandleA(LPCSTR lpModuleName);
/* UNICODE version */
HMODULE GetModuleHandleW(LPCWSTR lpModuleName);
```

windll 不会通过这样的魔法手段来帮你决定选择哪一种函数，你必须显式的调用 `GetModuleHandleA` 或 `GetModuleHandleW`，并分别使用字节对象或字符串对象作参数。

有时候，dlls 的导出的函数名不符合 Python 的标识符规范，比如 `"???2@YAPAXI@Z"`。此时，你必须使用*getattr()* 方法来获得该函数。

```
>>> getattr(cdll.msvcrt, "???2@YAPAXI@Z")
<_FuncPtr object at 0x...>
>>>
```

Windows 下，有些 dll 导出的函数没有函数名，而是通过其顺序号调用。对此类函数，你也可以通过 dll 对象的数值索引来操作这些函数。

```
>>> cdll.kernel32[1]
<_FuncPtr object at 0x...>
>>> cdll.kernel32[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ctypes.py", line 310, in __getitem__
    func = _StdcallFuncPtr(name, self)
AttributeError: function ordinal 0 not found
>>>
```

调用函数

你可以貌似是调用其它 Python 函数那样直接调用这些函数。在这个例子中，我们调用了 `time()` 函数，该函数返回一个系统时间戳（从 Unix 时间起点到现在的秒数），而 “GetModuleHandleA()“ 函数返回一个 win32 模块句柄。

此函数中调用的两个函数都使用了空指针（用 `None` 作为空指针）：

```
>>> print(libc.time(None))
1150640792
>>> print(hex(windll.kernel32.GetModuleHandleA(None)))
0x1d000000
>>>
```

如果你用 `cdecl` 调用方式调用 `stdcall` 约定的函数，则会甩出一个异常*ValueError*。反之亦然。

```
>>> cdll.kernel32.GetModuleHandleA(None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with not enough arguments (4 bytes missing)
>>>

>>> windll.msvcrt.printf(b"spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Procedure probably called with too many arguments (4 bytes in excess)
>>>
```

你必须阅读这些库的头文件或说明文档来确定它们的调用协议。

在 Windows 中，*ctypes* 使用 win32 结构化异常处理来防止由于在调用函数时使用非法参数导致的程序崩溃。

```
>>> windll.kernel32.GetModuleHandleA(32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: exception: access violation reading 0x00000020
>>>
```

然而，总有许多办法，通过调用*ctypes* 使得 Python 程序崩溃。因此，你必须小心使用。*faulthandler* 模块可以用于帮助诊断程序崩溃的原因。(比如由于错误的 C 库函数调用导致的段错误)。

`None`，整型，字节对象和（UNICODE）字符串是仅有的可以直接作为函数参数使用的四种 Python 本地数据类型。*None*‘ 作为 C 的空指针 (`NULL`)，字节和字符串类型作为一个指向其保存数据的内存块指针（`char *` 或 `wchar_t *`)。Python 的整型则作为平台默认的 C 的 `int` 类型，他们的数值被截断以适应 C 类型的整型长度。

在我们开始调用函数前，我们必须先了解作为函数参数的*ctypes* 数据类型。

基础数据类型

ctypes 定义了一些和 C 兼容的基本数据类型：

ctypes 类型	C 类型	Python 数据类型
c_bool	_Bool	bool (1)
c_char	char	单字符字节串对象
c_wchar	wchar_t	单字符字符串
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 或 long long	int
c_ulonglong	unsigned __int64 或 unsigned long long	int
c_size_t	size_t	int
c_ssize_t	ssize_t 或 Py_ssize_t	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	字节串对象或 None
c_wchar_p	wchar_t * (NUL terminated)	字符串或 None
c_void_p	void *	int 或 None

(1) 构造函数接受任何具有真值的对象。

所有这些类型都可以通过使用正确类型和值的可选初始值调用它们来创建:

```
>>> c_int()
c_long(0)
>>> c_wchar_p("Hello, World")
c_wchar_p(140018365411392)
>>> c_ushort(-3)
c_ushort(65533)
>>>
```

由于这些类型是可变的，它们的值也可以在以后更改:

```
>>> i = c_int(42)
>>> print(i)
c_long(42)
>>> print(i.value)
42
>>> i.value = -99
>>> print(i.value)
-99
>>>
```

当给指针类型的对象*c_char_p*, *c_wchar_p* 和*c_void_p* 等赋值时，将改变它们所指向的 *内存地址*，而不是它们所指向的内存区域的 *内容* (这是理所当然的，因为 Python 的 bytes 对象是不可变的):

```
>>> s = "Hello, World"
>>> c_s = c_wchar_p(s)
>>> print(c_s)
c_wchar_p(139966785747344)
>>> print(c_s.value)
Hello World
>>> c_s.value = "Hi, there"
>>> print(c_s)              # the memory location has changed
c_wchar_p(139966783348904)
>>> print(c_s.value)
Hi, there
>>> print(s)                # first object is unchanged
Hello, World
>>>
```

但你要注意不能将它们传递给会改变指针所指内存的函数。如果你需要可改变的内存块，ctypes 提供了*create_string_buffer()* 函数，它提供多种方式创建这种内存块。当前的内存块内容可以通过 raw 属性存取，如果你希望将它作为 NUL 结束的字符串，请使用 value 属性:

```
>>> from ctypes import *
>>> p = create_string_buffer(3)            # create a 3 byte buffer, initialized to␣
↪NUL bytes
>>> print(sizeof(p), repr(p.raw))
3 b'\x00\x00\x00'
>>> p = create_string_buffer(b"Hello")     # create a buffer containing a NUL␣
↪terminated string
>>> print(sizeof(p), repr(p.raw))
6 b'Hello\x00'
>>> print(repr(p.value))
b'Hello'
>>> p = create_string_buffer(b"Hello", 10) # create a 10 byte buffer
>>> print(sizeof(p), repr(p.raw))
10 b'Hello\x00\x00\x00\x00\x00'
>>> p.value = b"Hi"
>>> print(sizeof(p), repr(p.raw))
10 b'Hi\x00lo\x00\x00\x00\x00\x00'
>>>
```

create_string_buffer() 函数替代以前的 ctypes 版本中的 c_buffer() 函数 (仍然可当作别名使用) 和 c_string() 函数。*create_unicode_buffer()* 函数创建包含 unicode 字符的可变内存块，与之对应的 C 语言类型是 wchar_t。

调用函数，继续

注意 printf 将打印到真正标准输出设备，而 * 不是 * *sys.stdout*，因此这些实例只能在控制台提示符下工作，而不能在 *IDLE* 或 *PythonWin* 中运行。

```
>>> printf = libc.printf
>>> printf(b"Hello, %s\n", b"World!")
Hello, World!
14
>>> printf(b"Hello, %S\n", "World!")
Hello, World!
14
>>> printf(b"%d bottles of beer\n", 42)
42 bottles of beer
```

(下页继续)

(繼續上一頁)

```
19
>>> printf(b"%f bottles of beer\n", 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: Don't know how to convert parameter 2
>>>
```

正如前面所提到过的，除了整数、字符串以及字节串之外，所有的 Python 类型都必须使用它们对应的*ctypes*类型包装，才能够被正确地转换为所需的 C 语言类型。

```
>>> printf(b"An int %d, a double %f\n", 1234, c_double(3.14))
An int 1234, a double 3.140000
31
>>>
```

使用自定义的数据类型调用函数

你也可以通过自定义*ctypes*参数转换方式来允许自定义类型作为参数。*ctypes*会寻找 _as_parameter_ 属性并使用它作为函数参数。当然，它必须是数字、字符串或者二进制字符串:

```
>>> class Bottles:
...     def __init__(self, number):
...         self._as_parameter_ = number
...
>>> bottles = Bottles(42)
>>> printf(b"%d bottles of beer\n", bottles)
42 bottles of beer
19
>>>
```

如果你不想把实例的数据存储到 _as_parameter_ 属性。可以通过定义*property*函数计算出这个属性。

指定必选参数的类型 (函数原型)

可以通过设置 argtypes 属性的方法指定从 DLL 中导出函数的必选参数类型。

argtypes 必须是一个 C 数据类型的序列 (这里的 printf 可能不是个好例子，因为它是变长参数，而且每个参数的类型依赖于格式化字符串，不过尝试这个功能也很方便):

```
>>> printf.argtypes = [c_char_p, c_char_p, c_int, c_double]
>>> printf(b"String '%s', Int %d, Double %f\n", b"Hi", 10, 2.2)
String 'Hi', Int 10, Double 2.200000
37
>>>
```

指定数据类型可以防止不合理的参数传递（就像 C 函数的原型），并且会自动尝试将参数转换为需要的类型:

```
>>> printf(b"%d %d %d", 1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: wrong type
>>> printf(b"%s %d %f\n", b"X", 2, 3)
X 2 3.000000
```

(下页继续)

(繼續上一頁)

```
13
>>>
```

如果你想通过自定义类型传递参数给函数，必须实现 `from_param()` 类方法，才能够将此自定义类型用于 `argtypes` 序列。`from_param()` 类方法接受一个 Python 对象作为函数输入，它应该进行类型检查或者其他必要的操作以保证接收到的对象是合法的，然后返回这个对象，或者它的 `_as_parameter_` 属性，或者其他你想要传递给 C 函数的参数。这里也一样，返回的结果必须是整型、字符串、二进制字符串、*ctypes* 类型，或者一个具有 `_as_parameter_` 属性的对象。

返回类型

默认情况下都会假定函数返回 C `int` 类型。其他返回类型可以通过设置函数对象的 `restype` 属性来指定。

这是个更高级的例子，它调用了 `strchr` 函数，这个函数接收一个字符串指针以及一个字符作为参数，返回另一个字符串指针。

```
>>> strchr = libc.strchr
>>> strchr(b"abcdef", ord("d"))
8059983
>>> strchr.restype = c_char_p    # c_char_p is a pointer to a string
>>> strchr(b"abcdef", ord("d"))
b'def'
>>> print(strchr(b"abcdef", ord("x")))
None
>>>
```

如果希望避免上述的 `ord("x")` 调用，可以设置 `argtypes` 属性，第二个参数就会将单字符的 Python 二进制字符对象转换为 C 字符:

```
>>> strchr.restype = c_char_p
>>> strchr.argtypes = [c_char_p, c_char]
>>> strchr(b"abcdef", b"d")
'def'
>>> strchr(b"abcdef", b"def")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ArgumentError: argument 2: exceptions.TypeError: one character string expected
>>> print(strchr(b"abcdef", b"x"))
None
>>> strchr(b"abcdef", b"d")
'def'
>>>
```

如果外部函数返回了一个整数，你也可以使用要给可调用的 Python 对象（比如函数或者类）作为 `restype` 属性的值。将会以 C 函数返回的 整数对象作为参数调用这个可调用对象，执行后的结果作为最终函数返回值。这在错误返回值校验和自动抛出异常等方面比较有用。

```
>>> GetModuleHandle = windll.kernel32.GetModuleHandleA
>>> def ValidHandle(value):
...     if value == 0:
...         raise WinError()
...     return value
...
>>>
>>> GetModuleHandle.restype = ValidHandle
```

(下页继续)

(繼續上一頁)

```
>>> GetModuleHandle(None)
486539264
>>> GetModuleHandle("something silly")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in ValidHandle
OSError: [Errno 126] The specified module could not be found.
>>>
```

WinError 函数可以调用 Windows 的 FormatMessage() API 获取错误码的字符串说明，然后返回一个异常。WinError 接收一个可选的错误码作为参数，如果没有的话，它将调用*GetLastError()* 获取错误码。

请注意，使用 errcheck 属性可以实现更强大的错误检查手段；详情请见参考手册。

传递指针（或以引用方式传递形参）

有时候 C 函数接口可能由于要往某个地址写入值，或者数据太大不适合作为值传递，从而希望接收一个 指针作为数据参数类型。这和 *传递参数引用*类似。

ctypes 暴露了*byref()* 函数用于通过引用传递参数，使用*pointer()* 函数也能达到同样的效果，只不过*pointer()* 需要更多步骤，因为它要先构造一个真实指针对象。所以在 Python 代码本身不需要使用这个指针对象的情况下，使用*byref()* 效率更高。

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer(b'\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b"1 3.14 Hello", b"%d %f %s",
...             byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
>>>
```

结构体和联合

结构体和联合必须继承自*ctypes* 模块中的*Structure* 和*Union* 。子类必须定义 _fields_ 属性。_fields_ 是一个二元组列表，二元组中包含 *field name* 和 *field type* 。

type 字段必须是一个*ctypes* 类型，比如*c_int*，或者其他*ctypes* 类型: 结构体、联合、数组、指针。

这是一个简单的 POINT 结构体，它包含名称为 *x* 和 *y* 的两个变量，还展示了如何通过构造函数初始化结构体。

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = [("x", c_int),
...                 ("y", c_int)]
...
>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10 20
>>> point = POINT(y=5)
>>> print(point.x, point.y)
```

(下页继续)

(繼續上一頁)

```
0 5
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: too many initializers
>>>
```

当然，你可以构造更复杂的结构体。一个结构体可以通过设置 type 字段包含其他结构体或者自身。

这是以一个 RECT 结构体，他包含了两个 POINT ，分别叫 *upperleft* 和 *lowerright*:

```
>>> class RECT(Structure):
...     _fields_ = [("upperleft", POINT),
...                 ("lowerright", POINT)]
...
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0 5
>>> print(rc.lowerright.x, rc.lowerright.y)
0 0
>>>
```

嵌套结构体可以通过几种方式构造初始化:

```
>>> r = RECT(POINT(1, 2), POINT(3, 4))
>>> r = RECT((1, 2), (3, 4))
```

可以通过 类获取字段*descriptor* ，它能提供很多有用的调试信息。

```
>>> print(POINT.x)
<Field type=c_long, ofs=0, size=4>
>>> print(POINT.y)
<Field type=c_long, ofs=4, size=4>
>>>
```

警告： *ctypes* 不支持带位域的结构体、联合以值的方式传给函数。这可能在 32 位 x86 平台上可以正常工作，但是对于一般情况，这种行为是未定义的。带位域的结构体、联合应该总是通过指针传递给函数。

结构体/联合字段对齐及字节顺序

默认情况下，结构体和联合的字段与 C 的字节对齐是一样的。也可以在定义子类的时候指定类的 `_pack_` 属性来覆盖这种行为。它必须设置为一个正整数，表示字段的最大对齐字节。这和 MSVC 中的 `#pragma pack(n)` 功能一样。

ctypes 中的结构体和联合使用的是本地字节序。要使用非本地字节序，可以使用*BigEndianStructure*, *LittleEndianStructure*, `BigEndianUnion`, and `LittleEndianUnion` 作为基类。这些类不能包含指针字段。

结构体和联合中的位域

结构体和联合中是可以包含位域字段的。位域只能用于整型字段，位长度通过 _fields_ 中的第三个参数指定:

```
>>> class Int(Structure):
...     _fields_ = [("first_16", c_int, 16),
...                 ("second_16", c_int, 16)]
...
>>> print(Int.first_16)
<Field type=c_long, ofs=0:0, bits=16>
>>> print(Int.second_16)
<Field type=c_long, ofs=0:16, bits=16>
>>>
```

数组

数组是一个序列，包含指定个数元素，且必须类型相同。

创建数组类型的推荐方式是使用一个类型乘以一个正数:

```
TenPointsArrayType = POINT * 10
```

下面是一个构造的数据案例，结构体中包含了 4 个 POINT 和一些其他东西。

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class MyStruct(Structure):
...     _fields_ = [("a", c_int),
...                 ("b", c_float),
...                 ("point_array", POINT * 4)]
>>>
>>> print(len(MyStruct().point_array))
4
>>>
```

和平常一样，通过调用它创建实例:

```
arr = TenPointsArrayType()
for pt in arr:
    print(pt.x, pt.y)
```

以上代码会打印几行 `0 0` ，因为数组内容被初始化为 0.

也能通过指定正确类型的数据来初始化:

```
>>> from ctypes import *
>>> TenIntegers = c_int * 10
>>> ii = TenIntegers(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(ii)
<c_long_Array_10 object at 0x...>
>>> for i in ii: print(i, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>>
```

指针

可以将*ctypes* 类型数据传入*pointer()* 函数创建指针:

```
>>> from ctypes import *
>>> i = c_int(42)
>>> pi = pointer(i)
>>>
```

指针实例拥有*contents* 属性，它返回指针指向的真实对象，如上面的 `i` 对象:

```
>>> pi.contents
c_long(42)
>>>
```

注意*ctypes* 并没有 OOR （返回原始对象），每次访问这个属性时都会构造返回一个新的相同对象:

```
>>> pi.contents is i
False
>>> pi.contents is pi.contents
False
>>>
```

将这个指针的 contents 属性赋值为另一个*c_int* 实例将会导致该指针指向该实例的内存地址:

```
>>> i = c_int(99)
>>> pi.contents = i
>>> pi.contents
c_long(99)
>>>
```

指针对象也可以通过整数下标进行访问:

```
>>> pi[0]
99
>>>
```

通过整数下标赋值可以改变指针所指向的真实内容:

```
>>> print(i)
c_long(99)
>>> pi[0] = 22
>>> print(i)
c_long(22)
>>>
```

使用 0 以外的索引也是合法的，但是你必须确保知道自己为什么这么做，就像 C 语言中: 你可以访问或者修改任意内存内容。通常只会在函数接收指针是才会使用这种特性，而且你 知道这个指针指向的是一个数组而不是单个值。

内部细节, *pointer()* 函数不只是创建了一个指针实例，它首先创建了一个指针 类型。这是通过调用*POINTER()* 函数实现的，它接收*ctypes* 类型为参数，返回一个新的类型:

```
>>> PI = POINTER(c_int)
>>> PI
<class 'ctypes.LP_c_long'>
>>> PI(42)
```

(下页继续)

(繼續上一頁)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected c_long instead of int
>>> PI(c_int(42))
<ctypes.LP_c_long object at 0x...>
>>>
```

无参调用指针类型可以创建一个 NULL 指针。NULL 指针的布尔值是 False

```
>>> null_ptr = POINTER(c_int)()
>>> print(bool(null_ptr))
False
>>>
```

解引用指针的时候，*ctypes* 会帮你检测是否指针为 NULL (但是解引用无效的非 NULL 指针仍会导致 Python 崩溃):

```
>>> null_ptr[0]
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>

>>> null_ptr[0] = 1234
Traceback (most recent call last):
    ....
ValueError: NULL pointer access
>>>
```

类型转换

通常情况下, ctypes 具有严格的类型检查。这代表着, 如果在函数 argtypes 中或者结构体定义成员中有 POINTER(c_int) 类型，只有相同类型的实例才会被接受。也有一些例外。比如，你可以传递兼容的数组实例给指针类型。所以，对于 POINTER(c_int) ，ctypes 也可以接受 c_int 类型的数组:

```
>>> class Bar(Structure):
...     _fields_ = [("count", c_int), ("values", POINTER(c_int))]
...
>>> bar = Bar()
>>> bar.values = (c_int * 3)(1, 2, 3)
>>> bar.count = 3
>>> for i in range(bar.count):
...     print(bar.values[i])
...
1
2
3
>>>
```

另外，如果一个函数 argtypes 列表中的参数显式的定义为指针类型 (如 POINTER(c_int))，指针所指向的类型 (这个例子中是 c_int) 也可以传递给函数。ctypes 会自动调用对应的*byref()* 转换。

可以给指针内容赋值为 None 将其设置为 Null

```
>>> bar.values = None
>>>
```

有时候你拥有一个不兼容的类型。在 C 中，你可以将一个类型强制转换为另一个。*ctypes* 中的 a *cast()* 函数提供了相同的功能。上面的结构体 `Bar` 的 `value` 字段接收 `POINTER(c_int)` 指针或者*c_int* 数组，但是不能接受其他类型的实例:

```
>>> bar.values = (c_byte * 4)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incompatible types, c_byte_Array_4 instance instead of LP_c_long instance
>>>
```

这种情况下, 需要手动使用*cast()* 函数。

cast() 函数可以将一个指针实例强制转换为另一种 ctypes 类型。*cast()* 接收两个参数，一个 ctypes 指针对象或者可以被转换为指针的其他类型对象，和一个 ctypes 指针类型。返回第二个类型的一个实例，该返回实例和第一个参数指向同一片内存空间:

```
>>> a = (c_byte * 4)()
>>> cast(a, POINTER(c_int))
<ctypes.LP_c_long object at ...>
>>>
```

所以*cast()* 可以用来给结构体 `Bar` 的 `values` 字段赋值:

```
>>> bar = Bar()
>>> bar.values = cast((c_byte * 4)(), POINTER(c_int))
>>> print(bar.values[0])
0
>>>
```

不完整类型

不完整类型即还没有定义成员的结构体、联合或者数组。在 C 中，它们通常用于前置声明，然后在后面定义:

```
struct cell; /* forward declaration */

struct cell {
    char *name;
    struct cell *next;
};
```

直接翻译成 ctypes 的代码如下，但是这行不通:

```
>>> class cell(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("next", POINTER(cell))]
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in cell
NameError: name 'cell' is not defined
>>>
```

因为新的 `cell` 类在 class 语句结束之前还没有完成定义。在*ctypes* 中，我们可以先定义 `cell` 类，在 class 语句结束之后再设置 `_fields_` 属性:

```
>>> from ctypes import *
>>> class cell(Structure):
...     pass
...
>>> cell._fields_ = [("name", c_char_p),
...                  ("next", POINTER(cell))]
>>>
```

让我们试试。我们定义两个 cell 实例，让它们互相指向对方，然后通过指针链式访问几次:

```
>>> c1 = cell()
>>> c1.name = b"foo"
>>> c2 = cell()
>>> c2.name = b"bar"
>>> c1.next = pointer(c2)
>>> c2.next = pointer(c1)
>>> p = c1
>>> for i in range(8):
...     print(p.name, end=" ")
...     p = p.next[0]
...
foo bar foo bar foo bar foo bar
>>>
```

回调函数

ctypes 允许创建一个指向 Python 可调用对象的 C 函数。它们有时候被称为 *回调函数*。

首先，你必须为回调函数创建一个类，这个类知道调用约定，包括返回值类型以及函数接收的参数类型及个数。

CFUNCTYPE() 工厂函数使用 cdecl 调用约定创建回调函数类型。在 Windows 上，*WINFUNCTYPE()* 工厂函数使用 stdcall 调用约定为回调函数创建类型。

这些工厂函数的第一个参数是返回值类型，回调函数的参数类型作为剩余参数。

这里展示一个使用 C 标准库函数 qsort() 的例子，它使用一个回调函数对数据进行排序。qsort() 将用来给整数数组排序:

```
>>> IntArray5 = c_int * 5
>>> ia = IntArray5(5, 1, 7, 33, 99)
>>> qsort = libc.qsort
>>> qsort.restype = None
>>>
```

qsort() 必须接收的参数，一个指向待排序数据的指针，元素个数，每个元素的大小，以及一个指向排序函数的指针，即回调函数。然后回调函数接收两个元素的指针，如果第一个元素小于第二个，则返回一个负整数，如果相等则返回 0，否则返回一个正整数。

所以，我们的回调函数要接收两个整数指针，返回一个整数。首先我们创建回调函数的 类型

```
>>> CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>>
```

首先，这是一个简单的回调，它会显示传入的值:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return 0
...
>>> cmp_func = CMPFUNC(py_cmp_func)
>>>
```

结果:

```
>>> qsort(ia, len(ia), sizeof(c_int), cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 5 7
py_cmp_func 1 7
>>>
```

现在我们可以比较两个元素并返回有用的结果了:

```
>>> def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>>
>>> qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

我们可以轻易地验证，现在数组是有序的了:

```
>>> for i in ia: print(i, end=" ")
...
1 5 7 33 99
>>>
```

这些工厂函数可以当作装饰器工厂，所以可以这样写:

```
>>> @CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
... def py_cmp_func(a, b):
...     print("py_cmp_func", a[0], b[0])
...     return a[0] - b[0]
...
>>> qsort(ia, len(ia), sizeof(c_int), py_cmp_func)
py_cmp_func 5 1
py_cmp_func 33 99
py_cmp_func 7 33
py_cmp_func 1 7
py_cmp_func 5 7
>>>
```

備🄴: 请确保你维持的*CFUNCTYPE()* 对象的引用周期与它们在 C 代码中的使用期一样长。*ctypes* 不会确保这一点，如果不这样做，它们可能会被垃圾回收，导致程序在执行回调函数时发生崩溃。

注意，如果回调函数在 Python 之外的另外一个线程使用 (比如，外部代码调用这个回调函数)，ctypes 会在每一次调用上创建一个虚拟 Python 线程。这个行为在大多数情况下是合理的，但也意味着如果有数据使用*threading.local* 方式存储，将无法访问，就算它们是在同一个 C 线程中调用的。

访问 dll 的导出变量

一些动态链接库不仅仅导出函数，也会导出变量。一个例子就是 Python 库本身的 `Py_OptimizeFlag`，根据启动选项 `-O` 、`-OO` 的不同，它是值可能为 0、1、2 的整型。

ctypes 可以通过 `in_dll()` 类方法访问这类变量。*pythonapi* 是用于访问 Python C 接口的预定义符号:

```
>>> opt_flag = c_int.in_dll(pythonapi, "Py_OptimizeFlag")
>>> print(opt_flag)
c_long(0)
>>>
```

如果解释器使用 `-O` 选项启动，这个例子会打印 `c_long(1)`，如果使用 `-OO` 启动，则会打印 `c_long(2)`。

一个扩展例子, 同时也展示了使用指针访问 Python 导出的 `PyImport_FrozenModules` 指针对象。

对文档中这个值的解释说明

> 该指针被初始化为指向 `struct _frozen` 数组，以 `NULL` 或者 0 作为结束标记。当一个冻结模块被导入，首先要在这个表中搜索。第三方库可以以此来提供动态创建的冻结模块集合。

这足以证明修改这个指针是很有用的。为了让实例大小不至于太长，这里只展示如何使用*ctypes* 读取这个表:

```
>>> from ctypes import *
>>>
>>> class struct_frozen(Structure):
...     _fields_ = [("name", c_char_p),
...                 ("code", POINTER(c_ubyte)),
...                 ("size", c_int)]
...
>>>
```

我们定义了 `struct _frozen` 数据类型，接着就可以获取这张表的指针了:

```
>>> FrozenTable = POINTER(struct_frozen)
>>> table = FrozenTable.in_dll(pythonapi, "PyImport_FrozenModules")
>>>
```

由于 `table` 是指向 `struct_frozen` 数组的 指针，我们可以遍历它，只不过需要自己判断循环是否结束，因为指针本身并不包含长度。它早晚会因为访问到野指针或者什么的把自己搞崩溃，所以我们最好在遇到 `NULL` 后就让它退出循环:

```
>>> for item in table:
...     if item.name is None:
...         break
...     print(item.name.decode("ascii"), item.size)
...
_frozen_importlib 31764
_frozen_importlib_external 41499
__hello__ 161
__phello__ -161
```

(下页继续)

(繼續上一頁)

```
__phello__.spam 161
>>>
```

Python 的冻结模块和冻结包 (由负 size 成员表示) 并不是广为人知的事情，它们仅仅用于实验。例如，可以使用 import __hello__ 尝试一下这个功能。

意外

ctypes 也有自己的边界，有时候会发生一些意想不到的事情。

比如下面的例子:

```
>>> from ctypes import *
>>> class POINT(Structure):
...     _fields_ = ("x", c_int), ("y", c_int)
...
>>> class RECT(Structure):
...     _fields_ = ("a", POINT), ("b", POINT)
...
>>> p1 = POINT(1, 2)
>>> p2 = POINT(3, 4)
>>> rc = RECT(p1, p2)
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
1 2 3 4
>>> # now swap the two points
>>> rc.a, rc.b = rc.b, rc.a
>>> print(rc.a.x, rc.a.y, rc.b.x, rc.b.y)
3 4 3 4
>>>
```

嗯。我们预想应该打印 3 4 1 2 。但是为什么呢? 这是 rc.a, rc.b = rc.b, rc.a 这行代码展开后的步骤:

```
>>> temp0, temp1 = rc.b, rc.a
>>> rc.a = temp0
>>> rc.b = temp1
>>>
```

注意 temp0 和 temp1 对象始终引用了对象 rc 的内容。然后执行 rc.a = temp0 会把 temp0 的内容拷贝到 rc 的空间。这也改变了 temp1 的内容。最终导致赋值语句 rc.b = temp1 没有产生预想的效果。

记住，访问被包含在结构体、联合、数组中的对象并不会将其 复制出来，而是得到了一个代理对象，它是对根对象的内部内容的一层包装。

下面是另一个可能和预期有偏差的例子:

```
>>> s = c_char_p()
>>> s.value = b"abc def ghi"
>>> s.value
b'abc def ghi'
>>> s.value is s.value
False
>>>
```

備F: 使用*c_char_p* 实例化的对象只能将其值设置为 bytes 或者整数。

为什么这里打印了 `False`？ctypes 实例是一些内存块加上一些用于访问这些内存块的*descriptor* 组成。将 Python 对象存储在内存块并不会存储对象本身，而是存储了对象的 内容。每次访问对象的内容都会构造一个新的 Python 对象。

变长数据类型

ctypes 对变长数组和结构体提供了一些支持。

The *resize()* function can be used to resize the memory buffer of an existing ctypes object. The function takes the object as first argument, and the requested size in bytes as the second argument. The memory block cannot be made smaller than the natural memory block specified by the objects type, a *ValueError* is raised if this is tried:

```
>>> short_array = (c_short * 4)()
>>> print(sizeof(short_array))
8
>>> resize(short_array, 4)
Traceback (most recent call last):
    ...
ValueError: minimum size is 8
>>> resize(short_array, 32)
>>> sizeof(short_array)
32
>>> sizeof(type(short_array))
8
>>>
```

这非常好，但是要怎么访问数组中额外的元素呢？因为数组类型已经定义包含 4 个元素，导致我们访问新增元素时会产生以下错误:

```
>>> short_array[:]
[0, 0, 0, 0]
>>> short_array[7]
Traceback (most recent call last):
    ...
IndexError: invalid index
>>>
```

使用*ctypes* 访问变长数据类型的一个可行方法是利用 Python 的动态特性，根据具体情况，在知道这个数据的大小后，(重新) 指定这个数据的类型。

16.16.2 ctypes 参考手册

寻找动态链接库

在编译型语言中，动态链接库会在编译、链接或者程序运行时访问。

The purpose of the `find_library()` function is to locate a library in a way similar to what the compiler or runtime loader does (on platforms with several versions of a shared library the most recent should be loaded), while the ctypes library loaders act like when a program is run, and call the runtime loader directly.

`ctypes.util` 模块提供了一个函数，可以帮助确定需要加载的库。

`ctypes.util.`**`find_library`**`(`*`name`*`)`
: 尝试寻找一个库然后返回其路径名，*name* 是库名称, 且去除了 *lib* 等前缀和 `.so` 、`.dylib` 、版本号等后缀 (这是 posix 连接器 `-l` 选项使用的格式)。如果没有找到对应的库，则返回 `None` 。

确切的功能取决于系统。

在 Linux 上，`find_library()` 会尝试运行外部程序 (`/sbin/ldconfig`, `gcc`, `objdump` 以及 `ld`) 来寻找库文件。返回库文件的文件名。

3.6 版更變: 在 Linux 上，如果其他方式找不到的话，会使用环境变量 `LD_LIBRARY_PATH` 搜索动态链接库。

这是一些例子:

```
>>> from ctypes.util import find_library
>>> find_library("m")
'libm.so.6'
>>> find_library("c")
'libc.so.6'
>>> find_library("bz2")
'libbz2.so.1.0'
>>>
```

On macOS, `find_library()` tries several predefined naming schemes and paths to locate the library, and returns a full pathname if successful:

```
>>> from ctypes.util import find_library
>>> find_library("c")
'/usr/lib/libc.dylib'
>>> find_library("m")
'/usr/lib/libm.dylib'
>>> find_library("bz2")
'/usr/lib/libbz2.dylib'
>>> find_library("AGL")
'/System/Library/Frameworks/AGL.framework/AGL'
>>>
```

在 Windows 上，`find_library()` 在系统路径中搜索，然后返回全路径，但是如果没有预定义的命名方案，`find_library("c")` 调用会返回 `None`

使用 *ctypes* 包装动态链接库，更好的方式可能是在开发的时候就确定名称，然后硬编码到包装模块中去，而不是在运行时使用 `find_library()` 寻找库。

加载动态链接库

有很多方式可以将动态链接库加载到 Python 进程。其中之一是实例化以下类的其中一个:

class ctypes.**CDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

此类的实例即已加载的动态链接库。库中的函数使用标准 C 调用约定，并假定返回 `int` 。

在 Windows 上创建 *CDLL* 实例可能会失败，即使 DLL 名称确实存在。当某个被加载 DLL 所依赖的 DLL 未找到时，将引发 *OSError* 错误并附带消息 *"[WinError 126] The specified module could not be found"*. 此错误消息不包含缺失 DLL 的名称，因为 Windows API 并不会返回此类信息，这使得此错误难以诊断。要解决此错误并确定是哪一个 DLL 未找到，你需要找出所依赖的 DLL 列表并使用 Windows 调试与跟踪工具确定是哪一个未找到。

也参考:

Microsoft DUMPBIN 工具 -- 一个用于查找 DLL 依赖的工具。

class ctypes.**OleDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

仅 Windows : 此类的实例即加载好的动态链接库，其中的函数使用 `stdcall` 调用约定，并且假定返回

windows 指定的*HRESULT* 返回码。*HRESULT* 的值包含的信息说明函数调用成功还是失败，以及额外错误码。如果返回值表示失败，会自动抛出*OSError* 异常。

3.3 版更變: 以前是引发*WindowsError*。

class ctypes.**WinDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*, *use_errno=False*, *use_last_error=False*, *winmode=None*)

仅 Windows: 此类的实例即加载好的动态链接库，其中的函数使用 stdcall 调用约定，并假定默认返回 int 。

调用动态库导出的函数之前，Python 会释放*global interpreter lock* ，并在调用后重新获取。

class ctypes.**PyDLL**(*name*, *mode=DEFAULT_MODE*, *handle=None*)

这个类实例的行为与*CDLL* 类似，只不过 不会在调用函数的时候释放 GIL 锁，且调用结束后会检查 Python 错误码。如果错误码被设置，会抛出一个 Python 异常。

所以，它只在直接调用 Python C 接口函数的时候有用。

通过使用至少一个参数（共享库的路径名）调用它们，可以实例化所有这些类。也可以传入一个已加载的动态链接库作为 handler 参数，其他情况会调用系统底层的 dlopen 或 LoadLibrary 函数将库加载到进程，并获取其句柄。

mode 可以指定库加载方式。详情请参见 *dlopen(3)* 手册页。在 Windows 上，会忽略 *mode* ，在 posix 系统上，总是会加上 RTLD_NOW ，且无法配置。

use_errno 参数如果设置为 true，可以启用 ctypes 的机制，通过一种安全的方法获取系统的*errno* 错误码。*ctypes* 维护了一个线程局部变量，它是系统*errno* 的一份拷贝；如果调用了使用 use_errno=True 创建的外部函数，*errno* 的值会与 ctypes 自己拷贝的那一份进行交换，函数执行完后立即再交换一次。

The function *ctypes.get_errno()* returns the value of the ctypes private copy, and the function *ctypes.set_errno()* changes the ctypes private copy to a new value and returns the former value.

use_last_error 参数如果设置为 true，可以在 Windows 上启用相同的策略，它是通过 Windows API 函数*GetLastError()* 和 SetLastError() 管理的。*ctypes.get_last_error()* 和*ctypes.set_last_error()* 可用于获取和设置 ctypes 自己维护的 windows 错误码拷贝。

winmode 参数用于在 Windows 平台上指定库的加载方式 (因为 *mode* 会被忽略)。他接受任何与 Win32 API 的 LoadLibraryEx 的标志兼容的值作为参数。省略时，默认设置使用最安全的 DLL 加载的标志，以避免 DLL 劫持等问题。传入 DLL 的全路径是保证正确加载库及其依赖最安全的方法。

3.8 版更變: 增加了 *winmode* 参数。

ctypes.**RTLD_GLOBAL**

用于 *mode* 参数的标识值。在此标识不可用的系统上，它被定义为整数 0。

ctypes.**RTLD_LOCAL**

Flag to use as *mode* parameter. On platforms where this is not available, it is the same as *RTLD_GLOBAL*.

ctypes.**DEFAULT_MODE**

加载动态链接库的默认模式。在 OSX 10.3 上，它是 *RTLD_GLOBAL* ，其余系统上是 *RTLD_LOCAL* 。

这些类的实例没有共用方法。动态链接库的导出函数可以通过属性或者索引的方式访问。注意，通过属性的方式访问会缓存这个函数，因而每次访问它时返回的都是同一个对象。另一方面，通过索引访问，每次都会返回一个新的对象:

```
>>> from ctypes import CDLL
>>> libc = CDLL("libc.so.6")  # On Linux
>>> libc.time == libc.time
True
>>> libc['time'] == libc['time']
False
```

还有下面这些属性可用，他们的名称以下划线开头，以避免和导出函数重名:

PyDLL.**_handle**
用于访问库的系统句柄。

PyDLL.**_name**
传入构造函数的库名称。

共享库也可以通用使用一个预制对象来加载，这种对象是*LibraryLoader* 类的实例，具体做法或是通过调用 LoadLibrary() 方法，或是通过将库作为加载器实例的属性来提取。

class ctypes.**LibraryLoader**(*dlltype*)
加载共享库的类。*dlltype* 应当为*CDLL*, *PyDLL*, *WinDLL* 或*OleDLL* 类型之一。

__getattr__() 具有特殊的行为：它允许通过将一个共享库作为库加载器实例的属性进行访问来加载它。加载结果将被缓存，因此重复的属性访问每次都会返回相同的库。

LoadLibrary(*name*)
加载一个共享库到进程中并将其返回。此方法总是返回一个新的库实例。

可用的预制库加载器有如下这些:

ctypes.**cdll**
创建*CDLL* 实例。

ctypes.**windll**
仅限 Windows：创建*WinDLL* 实例.

ctypes.**oledll**
仅限 Windows：创建*OleDLL* 实例。

ctypes.**pydll**
创建*PyDLL* 实例。

要直接访问 C Python api，可以使用一个现成的 Python 共享库对象:

ctypes.**pythonapi**
一个*PyDLL* 的实例，它将 Python C API 函数作为属性公开。请注意所有这些函数都应返回 C int，当然这也不是绝对的，因此你必须分配正确的 restype 属性以使用这些函数。

引发一个审计事件 ctypes.dlopen，附带参数 name。

引发一个审计事件 ctypes.dlsym，附带参数 library, name。

引发一个审计事件 ctypes.dlsym/handle，附带参数 handle, name。

外部函数

正如之前小节的说明，外部函数可作为被加载共享库的属性来访问。用此方式创建的函数对象默认接受任意数量的参数，接受任意 ctypes 数据实例作为参数，并且返回库加载器所指定的默认结果类型。它们是一个私有类的实例:

class ctypes.**_FuncPtr**
C 可调用外部函数的基类。

外部函数的实例也是兼容 C 的数据类型；它们代表 C 函数指针。

此行为可通过对外部函数对象的特殊属性赋值来自定义。

restype
赋值为一个 ctypes 类型来指定外部函数的结果类型。使用 None 表示 void，即不返回任何结果的函数。

赋值为一个不为 ctypes 类型的可调用 Python 对象也是可以的，在此情况下函数应返回 C int，该可调用对象将附带此整数被调用，以允许进一步的处理或错误检测。这种用法已被弃用，为了更

灵活的后续处理或错误检测请使用一个 ctypes 数据类型作为*restype* 并将*errcheck* 属性赋值为一个可调用对象。

argtypes
赋值为一个 ctypes 类型的元组来指定函数所接受的参数类型。使用 stdcall 调用规范的函数只能附带与此元组长度相同数量的参数进行调用；使用 C 调用规范的函数还可接受额外的未指明参数。

当外部函数被调用时，每个实际参数都会被传给*argtypes* 元组中条目的 from_param() 类方法，此方法允许将实际参数适配为此外部函数所接受的对象。例如，*argtypes* 元组中的*c_char_p* 条目将使用 ctypes 约定规则把作为参数传入的字符串转换为字节串对象。

新增：现在可以将不是 ctypes 类型的条目放入 argtypes，但每个条目都必须具有 from_param() 方法用于返回可作为参数的值（整数、字符串、ctypes 实例）。这样就允许定义可将自定义对象适配为函数形参的适配器。

errcheck
将一个 Python 函数或其他可调用对象赋值给此属性。该可调用对象将附带三个及以上的参数被调用。

callable(*result*, *func*, *arguments*)
result 是外部函数返回的结果，由 restype 属性指明。

func 是外部函数对象本身，这样就允许重新使用相同的可调用对象来对多个函数进行检查或后续处理。

arguments 是一个包含最初传递给函数调用的形参的元组，这样就允许对所用参数的行为进行特别处理。

此函数所返回的对象将会由外部函数调用返回，但它还可以在外部函数调用失败时检查结果并引发异常。

exception ctypes.**ArgumentError**
此异常会在外部函数无法对某个传入参数执行转换时被引发。

引发一个审计事件 ctypes.seh_exception 并附带参数 code。

引发一个审计事件 ctypes.call_function，附带参数 func_pointer, arguments。

函数原型

外部函数也可通过实例化函数原型来创建。函数原型类似于 C 中的函数原型；它们在不定义具体实现的情况下描述了一个函数（返回类型、参数类型、调用约定）。工厂函数必须使用函数所需要的结果类型和参数类型来调用，并可被用作装饰器工厂函数，在此情况下可以通过 @wrapper 语法应用于函数。请参阅回调函数了解有关示例。

ctypes.**CFUNCTYPE**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)
返回的函数原型会创建使用标准 C 调用约定的函数。该函数在调用过程中将释放 GIL。如果 *use_errno* 设为真值，则在调用之前和之后系统*errno* 变量的 ctypes 私有副本会与真正的*errno* 值进行交换；*use_last_error* 会为 Windows 错误码执行同样的操作。

ctypes.**WINFUNCTYPE**(*restype*, **argtypes*, *use_errno=False*, *use_last_error=False*)
Windows only: The returned function prototype creates functions that use the stdcall calling convention. The function will release the GIL during the call. *use_errno* and *use_last_error* have the same meaning as above.

ctypes.**PYFUNCTYPE**(*restype*, **argtypes*)
返回的函数原型会创建使用 Python 调用约定的函数。该函数在调用过程中将 不会释放 GIL。

这些工厂函数所创建的函数原型可通过不同的方式来实例化，具体取决于调用中的类型与数量:

prototype(*address*)
在指定地址上返回一个外部函数，地址值必须为整数。

prototype(*callable*)
基于 Python *callable* 创建一个 C 可调用函数（回调函数）。

prototype(*func_spec*[, *paramflags*])
返回由一个共享库导出的外部函数。*func_spec* 必须为一个 2 元组 `(name_or_ordinal, library)`。第一项是字符串形式的所导出函数名称，或小整数形式的所导出函数序号。第二项是该共享库实例。

prototype(*vtbl_index*, *name*[, *paramflags*[, *iid*]])
返回将调用一个 COM 方法的外部函数。*vtbl_index* 虚拟函数表中的索引。*name* 是 COM 方法的名称。*iid* 是可选的指向接口标识符的指针，它被用于扩展的错误报告。

COM 方法使用特殊的调用约定：除了在 `argtypes` 元组中指定的形参，它们还要求一个指向 COM 接口的指针作为第一个参数。

可选的 *paramflags* 形参会创建相比上述特性具有更多功能的外部函数包装器。

paramflags 必须为一个与 `argtypes` 长度相同的元组。

此元组中的每一项都包含有关形参的更多信息，它必须为包含一个、两个或更多条目的元组。

第一项是包含形参指令旗标组合的整数。

1 指定函数的一个输入形参。

2 输出形参。外部函数会填入一个值。

4 默认为整数零值的输入形参。

可选的第二项是字符串形式的形参名称。如果指定此项，则可以使用该形参名称来调用外部函数。

可选的第三项是该形参的默认值。

这个例子演示了如何包装 Windows 的 `MessageBoxW` 函数以使其支持默认形参和已命名参数。相应 windows 头文件的 C 声明是这样的:

```
WINUSERAPI int WINAPI
MessageBoxW(
    HWND hWnd,
    LPCWSTR lpText,
    LPCWSTR lpCaption,
    UINT uType);
```

这是使用*ctypes* 的包装:

```
>>> from ctypes import c_int, WINFUNCTYPE, windll
>>> from ctypes.wintypes import HWND, LPCWSTR, UINT
>>> prototype = WINFUNCTYPE(c_int, HWND, LPCWSTR, LPCWSTR, UINT)
>>> paramflags = (1, "hwnd", 0), (1, "text", "Hi"), (1, "caption", "Hello from ctypes
↪"), (1, "flags", 0)
>>> MessageBox = prototype(("MessageBoxW", windll.user32), paramflags)
```

现在 `MessageBox` 外部函数可以通过以下方式来调用:

```
>>> MessageBox()
>>> MessageBox(text="Spam, spam, spam")
>>> MessageBox(flags=2, text="foo bar")
```

第二个例子演示了输出形参。这个 win32 GetWindowRect 函数通过将指定窗口的维度拷贝至调用者必须提供的 RECT 结构体来提取这些值。这是相应的 C 声明:

```
WINUSERAPI BOOL WINAPI
GetWindowRect(
     HWND hWnd,
     LPRECT lpRect);
```

这是使用*ctypes*的包装:

```
>>> from ctypes import POINTER, WINFUNCTYPE, windll, WinError
>>> from ctypes.wintypes import BOOL, HWND, RECT
>>> prototype = WINFUNCTYPE(BOOL, HWND, POINTER(RECT))
>>> paramflags = (1, "hwnd"), (2, "lprect")
>>> GetWindowRect = prototype(("GetWindowRect", windll.user32), paramflags)
>>>
```

带有输出形参的函数如果输出形参存在单一值则会自动返回该值，或是当输出形参存在多个值时返回包含这些值的元组，因此当 GetWindowRect 被调用时现在将返回一个 RECT 实例。

输出形参可以与 errcheck 协议相结合以执行进一步的输出处理与错误检查。Win32 GetWindowRect API 函数返回一个 BOOL 来表示成功或失败，因此此函数可执行错误检查，并在 API 调用失败时引发异常:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     return args
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

如果 errcheck 不加更改地返回它所接收的参数元组，则*ctypes*会继续对输出形参执行常规处理。如果你希望返回一个窗口坐标的元组而非 RECT 实例，你可以从函数中提取这些字段并返回它们，常规处理将不会再执行:

```
>>> def errcheck(result, func, args):
...     if not result:
...         raise WinError()
...     rc = args[1]
...     return rc.left, rc.top, rc.bottom, rc.right
...
>>> GetWindowRect.errcheck = errcheck
>>>
```

工具函数

ctypes.**addressof**(*obj*)
: 以整数形式返回内存缓冲区地址。*obj* 必须为一个 ctypes 类型的实例。

  引发一个审计事件 ctypes.addressof，附带参数 obj。

ctypes.**alignment**(*obj_or_type*)
: 返回一个 ctypes 类型的对齐要求。*obj_or_type* 必须为一个 ctypes 类型或实例。

ctypes.**byref**(*obj*[, *offset*])
: 返回指向 *obj* 的轻量指针，该对象必须为一个 ctypes 类型的实例。*offset* 默认值为零，且必须为一个将被添加到内部指针值的整数。

byref(obj, offset) 对应于这段 C 代码:

```
(((char *)&obj) + offset)
```

返回的对象只能被用作外部函数调用形参。它的行为类似于 pointer(obj)，但构造起来要快很多。

ctypes.**cast**(*obj*, *type*)
此函数类似于 C 的强制转换运算符。它返回一个 *type* 的新实例，该实例指向与 *obj* 相同的内存块。*type* 必须为指针类型，而 *obj* 必须为可以被作为指针来解读的对象。

ctypes.**create_string_buffer**(*init_or_size*, *size=None*)
此函数会创建一个可变的字符缓冲区。返回的对象是一个*c_char* 的 ctypes 数组。

init_or_size 必须是一个指明数组大小的整数，或者是一个将被用来初始化数组条目的字节串对象。

如果将一个字节串对象指定为第一个参数，则将使缓冲区大小比其长度多一项以便数组的最后一项为一个 NUL 终结符。可以传入一个整数作为第二个参数以允许在不使用字节串长度的情况下指定数组大小。

引发一个审计事件 ctypes.create_string_buffer，附带参数 init, size。

ctypes.**create_unicode_buffer**(*init_or_size*, *size=None*)
此函数会创建一个可变的 unicode 字符缓冲区。返回的对象是一个*c_wchar* 的 ctypes 数组。

init_or_size 必须是一个指明数组大小的整数，或者是一个将被用来初始化数组条目的字符串。

如果将一个字符串指定为第一个参数，则将使缓冲区大小比其长度多一项以便数组的最后一项为一个 NUL 终结符。可以传入一个整数作为第二个参数以允许在不使用字符串长度的情况下指定数组大小。

引发一个审计事件 ctypes.create_unicode_buffer，附带参数 init, size。

ctypes.**DllCanUnloadNow**()
仅限 Windows：此函数是一个允许使用 ctypes 实现进程内 COM 服务的钩子。它将由 _ctypes 扩展 dll 所导出的 DllCanUnloadNow 函数来调用。

ctypes.**DllGetClassObject**()
仅限 Windows：此函数是一个允许使用 ctypes 实现进程内 COM 服务的钩子。它将由 _ctypes 扩展 dll 所导出的 DllGetClassObject 函数来调用。

ctypes.util.**find_library**(*name*)
尝试寻找一个库并返回路径名称。*name* 是库名称并且不带任何前缀如 lib 以及后缀如 .so，.dylib 或版本号（形式与 posix 链接器选项 -l 所用的一致）。如果找不到库，则返回 None。

确切的功能取决于系统。

ctypes.util.**find_msvcrt**()
仅限 Windows：返回 Python 以及扩展模块所使用的 VC 运行时库的文件名。如果无法确定库名称，则返回 None。

如果你需要通过调用 free(void *) 来释放内存，例如某个扩展模块所分配的内存，重要的一点是你应当使用分配内存的库中的函数。

ctypes.**FormatError**([*code*])
仅限 Windows：返回错误码 *code* 的文本描述。如果未指定错误码，则会通过调用 Windows api 函数 GetLastError 来获得最新的错误码。

ctypes.**GetLastError**()
仅限 Windows：返回 Windows 在调用线程中设置的最新错误码。此函数会直接调用 Windows *GetLastError()* 函数，它并不返回错误码的 ctypes 私有副本。

ctypes.**get_errno**()
返回调用线程中系统*errno* 变量的 ctypes 私有副本的当前值。

引发一个审计事件 ctypes.get_errno，不附带任何参数。

ctypes.**get_last_error**()
仅限 Windows：返回调用线程中系统 LastError 变量的 ctypes 私有副本的当前值。

引发一个审计事件 ctypes.get_last_error，不附带任何参数。

ctypes.**memmove**(*dst*, *src*, *count*)
与标准 C memmove 库函数相同：将 *count* 个字节从 *src* 拷贝到 *dst*。*dst* 和 *src* 必须为整数或可被转换为指针的 ctypes 实例。

ctypes.**memset**(*dst*, *c*, *count*)
与标准 C memset 库函数相同：将位于地址 *dst* 的内存块用 *count* 个字节的 *c* 值填充。*dst* 必须为指定地址的整数或 ctypes 实例。

ctypes.**POINTER**(*type*)
这个工厂函数创建并返回一个新的 ctypes 指针类型。指针类型会被缓存并在内部重用，因此重复调用此函数耗费不大。*type* 必须为 ctypes 类型。

ctypes.**pointer**(*obj*)
此函数会创建一个新的指向 *obj* 的指针实例。返回的对象类型为 POINTER(type(obj))。

注意：如果你只是想向外部函数调用传递一个对象指针，你应当使用更为快速的 byref(obj)。

ctypes.**resize**(*obj*, *size*)
此函数可改变 *obj* 的内部内存缓冲区大小，其参数必须为 ctypes 类型的实例。没有可能将缓冲区设为小于对象类型的本机大小值，该值由 sizeof(type(obj)) 给出，但将缓冲区加大则是可能的。

ctypes.**set_errno**(*value*)
设置调用线程中系统*errno* 变量的 ctypes 私有副本的当前值为 *value* 并返回原来的值。

引发一个审计事件 ctypes.set_errno 附带参数 errno。

ctypes.**set_last_error**(*value*)
仅限 Windows：设置调用线程中系统 LastError 变量的 ctypes 私有副本的当前值为 *value* 并返回原来的值。

引发一个审计事件 ctypes.set_last_error，附带参数 error。

ctypes.**sizeof**(*obj_or_type*)
返回 ctypes 类型或实例的内存缓冲区以字节表示的大小。其功能与 C sizeof 运算符相同。

ctypes.**string_at**(*address*, *size=-1*)
此函数返回从内存地址 *address* 开始的以字节串表示的 C 字符串。如果指定了 size，则将其用作长度，否则将假定字符串以零值结尾。

引发一个审计事件 ctypes.string_at，附带参数 address, size。

ctypes.**WinError**(*code=None*, *descr=None*)
仅限 Windows：此函数可能是 ctypes 中名字起得最差的函数。它会创建一个 OSError 的实例。如果未指定 *code*，则会调用 GetLastError 来确定错误码。如果未指定 *descr*，则会调用*FormatError()* 来获取错误的文本描述。

3.3 版更變: 以前是会创建一个*WindowsError* 的实例。

ctypes.**wstring_at**(*address*, *size=-1*)
此函数返回从内存地址 *address* 开始的以字符串表示的宽字节字符串。如果指定了 *size*，则将其用作字符串中的字符数量，否则将假定字符串以零值结尾。

引发一个审计事件 ctypes.wstring_at，附带参数 address, size。

数据类型

class ctypes.**_CData**

这个非公有类是所有 ctypes 数据类型的共同基类。另外，所有 ctypes 类型的实例都包含一个存放 C 兼容数据的内存块；该内存块的地址可由*addressof()* 辅助函数返回。还有一个实例变量被公开为*_objects*；此变量包含其他在内存块包含指针的情况下需要保持存活的 Python 对象。

ctypes 数据类型的通用方法，它们都是类方法（严谨地说，它们是*metaclass* 的方法）：

from_buffer(*source*[, *offset*])

此方法返回一个共享 *source* 对象缓冲区的 ctypes 实例。*source* 对象必须支持可写缓冲区接口。可选的 *offset* 形参指定以字节表示的源缓冲区内偏移量；默认值为零。如果源缓冲区不够大则会引发*ValueError*。

引发一个审计事件 ctypes.cdata/buffer 附带参数 pointer, size, offset。

from_buffer_copy(*source*[, *offset*])

此方法创建一个 ctypes 实例，从 *source* 对象缓冲区拷贝缓冲区，该对象必须是可读的。可选的 *offset* 形参指定以字节表示的源缓冲区内偏移量；默认值为零。如果源缓冲区不够大则会引发*ValueError*。

引发一个审计事件 ctypes.cdata/buffer 附带参数 pointer, size, offset。

from_address(*address*)

此方法会使用 *address* 所指定的内存返回一个 ctypes 类型的实例，该参数必须为一个整数。

引发一个审计事件 ctypes.cdata，附带参数 address。

from_param(*obj*)

此方法会将 *obj* 适配为一个 ctypes 类型。它调用时会在当该类型存在于外部函数的 argtypes 元组时传入外部函数调用所使用的实际对象；它必须返回一个可被用作函数调用参数的对象。

所有 ctypes 数据类型都带有这个类方法的默认实现，它通常会返回 *obj*，如果该对象是此类型的实例的话。某些类型也能接受其他对象。

in_dll(*library*, *name*)

此方法返回一个由共享库导出的 ctypes 类型。*name* 为导出数据的符号名称，*library* 为所加载的共享库。

ctypes 数据类型的通用实例变量:

_b_base_

有时 ctypes 数据实例并不拥有它们所包含的内存块，它们只是共享了某个基对象的部分内存块。*_b_base_* 只读成员是拥有内存块的根 ctypes 对象。

_b_needsfree_

这个只读变量在 ctypes 数据实例自身已分配了内存块时为真值，否则为假值。

_objects

这个成员或者为 None，或者为一个包含需要保持存活以使内存块的内存保持有效的 Python 对象的字典。这个对象只是出于调试目的而对外公开；绝对不要修改此字典的内容。

基础数据类型

class ctypes.**_SimpleCData**
这个非公有类是所有基本 ctypes 数据类型的基类。它在这里被提及是因为它包含基本 ctypes 数据类型共有的属性。*_SimpleCData* 是*_CData* 的子类，因此继承了其方法和属性。非指针及不包含指针的 ctypes 数据类型现在将可以被封存。

实例拥有一个属性:

value
这个属性包含实例的实际值。对于整数和指针类型，它是一个整数，对于字符类型，它是一个单字符字符串对象或字符串，对于字符指针类型，它是一个 Python 字节串对象或字符串。

当从 ctypes 实例提取 `value` 属性时，通常每次会返回一个新的对象。*ctypes* 并 没有实现原始对象返回，它总是会构造一个新的对象。所有其他 ctypes 对象实例也同样如此。

基本数据类型当作为外部函数调用结果被返回或者作为结构字段成员或数组项被提取时，会透明地转换为原生 Python 类型。换句话说，如果某个外部函数具有*c_char_p* 的 `restype`，你将总是得到一个 Python 字节串对象，而 不是一个*c_char_p* 实例。

基本数据类型的子类并 没有继续此行为。因此，如果一个外部函数的 `restype` 是*c_void_p* 的一个子类，你将从函数调用得到一个该子类的实例。当然，你可以通过访问 `value` 属性来获取指针的值。

这些是基本 ctypes 数据类型:

class ctypes.**c_byte**
代表 C `signed char` 数据类型，并将值解读为一个小整数。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_char**
代表 C `char` 数据类型，并将值解读为单个字符。该构造器接受一个可选的字符串初始化器，字符串的长度必须恰好为一个字符。

class ctypes.**c_char_p**
当指向一个以零为结束符的字符串时代表 C `char *` 数据类型。对于通用字符指针来说也可能指向二进制数据，必须要使用 `POINTER(c_char)`。该构造器接受一个整数地址，或者一个字节串对象。

class ctypes.**c_double**
代表 C `double` 数据类型。该构造器接受一个可选的浮点数初始化器。

class ctypes.**c_longdouble**
代表 C `long double` 数据类型。该构造器接受一个可选的浮点数初始化器。在 `sizeof(long double) == sizeof(double)` 的平台上它是*c_double* 的一个别名。

class ctypes.**c_float**
代表 C `float` 数据类型。该构造器接受一个可选的浮点数初始化器。

class ctypes.**c_int**
代表 C `signed int` 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。在 `sizeof(int) == sizeof(long)` 的平台上它是*c_long* 的一个别名。

class ctypes.**c_int8**
代表 C 8 位 `signed int` 数据类型。通常是*c_byte* 的一个别名。

class ctypes.**c_int16**
代表 C 16 位 `signed int` 数据类型。通常是*c_short* 的一个别名。

class ctypes.**c_int32**
代表 C 32 位 `signed int` 数据类型。通常是*c_int* 的一个别名。

class ctypes.**c_int64**
代表 C 64 位 `signed int` 数据类型。通常是*c_longlong* 的一个别名。

class ctypes.**c_long**
代表 C `signed long` 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_longlong**
代表 C `signed long long` 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_short**
代表 C `signed short` 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_size_t**
代表 C `size_t` 数据类型。

class ctypes.**c_ssize_t**
代表 C `ssize_t` 数据类型。

3.2 版新加入.

class ctypes.**c_ubyte**
代表 C `unsigned char` 数据类型，它将值解读为一个小整数。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_uint**
代表 C `unsigned int` 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。在 `sizeof(int) == sizeof(long)` 的平台上它是*c_ulong* 的一个别名。

class ctypes.**c_uint8**
代表 C 8 位 `unsigned int` 数据类型。通常是*c_ubyte* 的一个别名。

class ctypes.**c_uint16**
代表 C 16 位 `unsigned int` 数据类型。通常是*c_ushort* 的一个别名。

class ctypes.**c_uint32**
代表 C 32 位 `unsigned int` 数据类型。通常是*c_uint* 的一个别名。

class ctypes.**c_uint64**
代表 C 64 位 `unsigned int` 数据类型。通常是*c_ulonglong* 的一个别名。

class ctypes.**c_ulong**
代表 C `unsigned long` 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_ulonglong**
代表 C `unsigned long long` 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_ushort**
代表 C `unsigned short` 数据类型。该构造器接受一个可选的整数初始化器；不会执行溢出检查。

class ctypes.**c_void_p**
代表 C `void *` 类型。该值被表示为整数形式。该构造器接受一个可选的整数初始化器。

class ctypes.**c_wchar**
代表 C `wchar_t` 数据类型，并将值解读为一单个字符的 unicode 字符串。该构造器接受一个可选的字符串初始化器，字符串的长度必须恰好为一个字符。

class ctypes.**c_wchar_p**
代表 C `wchar_t *` 数据类型，它必须为指向以零为结束符的宽字符串的指针。该构造器接受一个整数地址或者一个字符串。

class ctypes.**c_bool**
代表 C `bool` 数据类型 (更准确地说是 C99 `_Bool`)。它的值可以为 `True` 或 `False`，并且该构造器接受任何具有逻辑值的对象。

class ctypes.**HRESULT**
Windows 专属：代表一个 HRESULT 值，它包含某个函数或方法调用的成功或错误信息。

class ctypes.**py_object**
代表 C PyObject * 数据类型。不带参数地调用此构造器将创建一个 NULL PyObject * 指针。

ctypes.wintypes 模块提供了其他许多 Windows 专属的数据类型，例如 HWND, WPARAM 或 DWORD。还定义了一些有用的结构体例如 MSG 或 RECT。

结构化数据类型

class ctypes.**Union**(**args*, ***kw*)
本机字节序的联合所对应的抽象基类。

class ctypes.**BigEndianStructure**(**args*, ***kw*)
大端字节序的结构体所对应的抽象基类。

class ctypes.**LittleEndianStructure**(**args*, ***kw*)
小端字节序的结构体所对应的抽象基类。

非本机字节序的结构体不能包含指针类型字段，或任何其他包含指针类型字段的数据类型。

class ctypes.**Structure**(**args*, ***kw*)
本机字节序的结构体所对应的抽象基类。

实际的结构体和联合类型必须通过子类化这些类型之一来创建，并且至少要定义一个*_fields_* 类变量。*ctypes* 将创建*descriptor*，它允许通过直接属性访问来读取和写入字段。这些是

fields
一个定义结构体字段的序列。其中的条目必须为 2 元组或 3 元组。元组的第一项是字段名称，第二项指明字段类型；它可以是任何 ctypes 数据类型。

对于整数类型字段例如*c_int*，可以给定第三个可选项。它必须是一个定义字段比特位宽度的小正整数。

字段名称在一个结构体或联合中必须唯一。不会检查这个唯一性，但当名称出现重复时将只有一个字段可被访问。

可以在定义 Structure 子类的类语句之后再定义*_fields_* 类变量，这将允许创建直接或间接引用其自身的数据类型:

```
class List(Structure):
    pass
List._fields_ = [("pnext", POINTER(List)),
                 ...
                ]
```

但是，*_fields_* 类变量必须在类型第一次被使用（创建实例，调用*sizeof()* 等等）之前进行定义。在此之后对*_fields_* 类变量赋值将会引发 AttributeError。

可以定义结构体类型的子类，它们会继承基类的字段再加上在子类中定义的任何*_fields_*。

pack
一个可选的小整数，它允许覆盖实体中结构体字段的对齐方式。当*_fields_* 被赋值时必须已经定义了*_pack_*，否则它将没有效果。

anonymous
一个可选的序列，它会列出未命名（匿名）字段的名称。当*_fields_* 被赋值时必须已经定义了*_anonymous_*，否则它将没有效果。

在此变量中列出的字段必须为结构体或联合类型字段。*ctypes* 将在结构体类型中创建描述器以允许直接访问嵌套字段，而无需创建对应的结构体或联合字段。

以下是一个示例类型（Windows）：

```
class _U(Union):
    _fields_ = [("lptdesc", POINTER(TYPEDESC)),
                ("lpadesc", POINTER(ARRAYDESC)),
                ("hreftype", HREFTYPE)]

class TYPEDESC(Structure):
    _anonymous_ = ("u",)
    _fields_ = [("u", _U),
                ("vt", VARTYPE)]
```

TYPEDESC 结构体描述了一个 COM 数据类型，vt 字段指明哪个联合字段是有效的。由于 u 字段被定义为匿名字段，现在可以直接从 TYPEDESC 实例访问成员。td.lptdesc 和 td.u.lptdesc 是等价的，但前者速度更快，因为它不需要创建临时的联合实例：

```
td = TYPEDESC()
td.vt = VT_PTR
td.lptdesc = POINTER(some_type)
td.u.lptdesc = POINTER(some_type)
```

可以定义结构体的子类，它们会继承基类的字段。如果子类定义具有单独的*_fields_* 变量，在其中指定的字段会被添加到基类的字段中。

结构体和联合的构造器均可接受位置和关键字参数。位置参数用于按照*_fields_* 中的出现顺序来初始化成员字段。构造器中的关键字参数会被解读为属性赋值，因此它们将以相应的名称来初始化*_fields_*，或为不存在于*_fields_* 中的名称创建新的属性。

数组与指针

class ctypes.**Array**(**args*)
数组的抽象基类。

The recommended way to create concrete array types is by multiplying any *ctypes* data type with a non-negative integer. Alternatively, you can subclass this type and define *_length_* and *_type_* class variables. Array elements can be read and written using standard subscript and slice accesses; for slice reads, the resulting object is *not* itself an *Array*.

length
一个指明数组中元素数量的正整数。超出范围的抽取会导致*IndexError*。该值将由*len()* 返回。

type
指明数组中每个元素的类型。

Array 子类构造器可接受位置参数，用来按顺序初始化元素。

class ctypes.**_Pointer**
私有对象，指针的抽象基类。

实际的指针类型是通过调用*POINTER()* 并附带其将指向的类型来创建的；这会由*pointer()* 自动完成。

如果一个指针指向的是数组，则其元素可使用标准的抽取和切片方式来读写。指针对象没有长度，因此*len()* 将引发*TypeError*。抽取负值将会从指针之前的内存中读取（与 C 一样），并且超出范围的抽取将可能因非法访问而导致崩溃（视你的运气而定）。

type
指明所指向的类型。

contents
返回指针所指向的对象。对此属性赋值会使指针改为指向所赋值的对象。

CHAPTER 17

并发执行

本章中描述的模块支持并发执行代码。适当的工具选择取决于要执行的任务（CPU 密集型或 IO 密集型）和偏好的开发风格（事件驱动的协作式多任务或抢占式多任务处理）。这是一个概述：

17.1 threading --- 基于线程的并行

源代码: Lib/threading.py

这个模块在较低级的模块 *_thread* 基础上建立较高级的线程接口。参见： *queue* 模块。

3.7 版更變: 这个模块曾经为可选项，但现在总是可用。

備🄴: 虽然他们没有在下面列出，这个模块仍然支持 Python 2.x 系列的这个模块下以 `camelCase` （驼峰法）命名的方法和函数。

CPython implementation detail: 在 CPython 中，由于存在*全局解释器锁*，同一时刻只有一个线程可以执行 Python 代码（虽然某些性能导向的库可能会去除此限制）。如果你想让你的应用更好地利用多核心计算机的计算资源，推荐你使用 *multiprocessing* 或 *concurrent.futures.ProcessPoolExecutor*。但是，如果你想要同时运行多个 I/O 密集型任务，则多线程仍然是一个合适的模型。

这个模块定义了以下函数：

`threading.`**`active_count`**`()`
返回当前存活的 *Thread* 对象的数量。返回值与 *enumerate()* 所返回的列表长度一致。

`threading.`**`current_thread`**`()`
返回当前对应调用者的控制线程的 *Thread* 对象。如果调用者的控制线程不是利用 *threading* 创建，会返回一个功能受限的虚拟线程对象。

`threading.`**`excepthook`**`(`*args*, */*`)`
处理由 *Thread.run()* 引发的未捕获异常。

args 参数具有以下属性：

- *exc_type*: 异常类型
- *exc_value*: 异常值，可以是 None.
- *exc_traceback*: 异常回溯，可以是 None.
- *thread*: 引发异常的线程，可以为 None。

如果 *exc_type* 为*SystemExit*，则异常会被静默地忽略。在其他情况下，异常将被打印到*sys.stderr*。

如果此函数引发了异常，则会调用*sys.excepthook()* 来处理它。

threading.excepthook() 可以被重载以控制由*Thread.run()* 引发的未捕获异常的处理方式。

使用定制钩子存放 *exc_value* 可能会创建引用循环。它应当在不再需要异常时被显式地清空以打破引用循环。

使用定制钩子存放 *thread* 可能会在它设为被终结对象时将其重生。请避免在定制钩子完成后存放 *thread* 以避免对象的重生。

也参考:

sys.excepthook() 处理未捕获的异常。

3.8 版新加入.

threading.**get_ident**()
返回当前线程的“线程标识符”。它是一个非零的整数。它的值没有直接含义，主要是用作 magic cookie，比如作为含有线程相关数据的字典的索引。线程标识符可能会在线程退出，新线程创建时被复用。

3.3 版新加入.

threading.**get_native_id**()
返回内核分配给当前线程的原生集成线程 ID。这是一个非负整数。它的值可被用来在整个系统中唯一地标识这个特定线程（直到线程终结，在那之后该值可能会被 OS 回收再利用）。

可用性: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX。

3.8 版新加入.

threading.**enumerate**()
返回当前所有存活的*Thread* 对象的列表。该列表包括守护线程以及*current_thread()* 创建的空线程。它不包括已终结的和尚未开始的线程。但是，主线程将总是结果的一部分，即使是在已终结的时候。

threading.**main_thread**()
返回主*Thread* 对象。一般情况下，主线程是 Python 解释器开始时创建的线程。

3.4 版新加入.

threading.**settrace**(*func*)
为所有*threading* 模块开始的线程设置追踪函数。在每个线程的*run()* 方法被调用前，*func* 会被传递给*sys.settrace()* 。

threading.**setprofile**(*func*)
为所有*threading* 模块开始的线程设置性能测试函数。在每个线程的*run()* 方法被调用前，*func* 会被传递给*sys.setprofile()* 。

threading.**stack_size**([*size*])
返回创建线程时用的堆栈大小。可选参数 *size* 指定之后新建的线程的堆栈大小，而且一定要是 0（根据平台或者默认配置）或者最小是 32,768(32KiB) 的一个正整数。如果 *size* 没有指定，默认是 0。如果不支持改变线程堆栈大小，会抛出*RuntimeError* 错误。如果指定的堆栈大小不合法，会抛出*ValueError* 错误并且不会修改堆栈大小。32KiB 是当前最小的能保证解释器有足够堆栈空间的堆栈大小。需要注意的是部分平台对于堆栈大小会有特定的限制，例如要求大于 32KiB 的堆栈大小或者需要根据系统内

存页面的整数倍进行分配 - 应当查阅平台文档有关详细信息（4KiB 页面比较普遍，在没有更具体信息的情况下，建议的方法是使用 4096 的倍数作为堆栈大小）。

适用于: Windows，具有 POSIX 线程的系统。

这个模块同时定义了以下常量：

threading.**TIMEOUT_MAX**
阻塞函数（*Lock.acquire()*, *RLock.acquire()*, *Condition.wait()*, ...）中形参 *timeout* 允许的最大值。传入超过这个值的 timeout 会抛出*OverflowError* 异常。

3.2 版新加入.

这个模块定义了许多类，详见以下部分。

该模块的设计基于 Java 的线程模型。但是，在 Java 里面，锁和条件变量是每个对象的基础特性，而在 Python 里面，这些被独立成了单独的对象。Python 的*Thread* 类只是 Java 的 Thread 类的一个子集；目前还没有优先级，没有线程组，线程还不能被销毁、停止、暂停、恢复或中断。Java 的 Thread 类的静态方法在实现时会映射为模块级函数。

下述方法的执行都是原子性的。

17.1.1 线程本地数据

线程本地数据是特定线程的数据。管理线程本地数据，只需要创建一个*local* （或者一个子类型）的实例并在实例中储存属性：

```
mydata = threading.local()
mydata.x = 1
```

在不同的线程中，实例的值会不同。

class threading.**local**
一个代表线程本地数据的类。

更多相关细节和大量示例，参见 _threading_local 模块的文档。

17.1.2 线程对象

The *Thread* class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the *run()* method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, *only* override the __init__() and *run()* methods of this class.

当线程对象一但被创建，其活动一定会因调用线程的*start()* 方法开始。这会在独立的控制线程调用*run()* 方法。

一旦线程活动开始，该线程会被认为是' 存活的' 。当它的*run()* 方法终结了（不管是正常的还是抛出未被处理的异常），就不是' 存活的'。*is_alive()* 方法用于检查线程是否存活。

其他线程可以调用一个线程的*join()* 方法。这会阻塞调用该方法的线程，直到被调用*join()* 方法的线程终结。

线程有名字。名字可以传递给构造函数，也可以通过*name* 属性读取或者修改。

如果*run()* 方法引发了异常，则会调用*threading.excepthook()* 来处理它。在默认情况下，*threading.excepthook()* 会静默地忽略*SystemExit*。

一个线程可以被标记成一个“守护线程”。这个标识的意义是，当剩下的线程都是守护线程时，整个 Python 程序将会退出。初始值继承于创建线程。这个标识可以通过*daemon* 特征属性或者 *daemon* 构造器参数来设置。

備🄵: 守护线程在程序关闭时会突然关闭。他们的资源（例如已经打开的文档，数据库事务等等）可能没有被正确释放。如果你想你的线程正常停止，设置他们成为非守护模式并且使用合适的信号机制，例如: *Event*。

有个” 主线程” 对象；这对应 Python 程序里面初始的控制线程。它不是一个守护线程。

” 虚拟线程对象” 是可以被创建的。这些是对应于“外部线程”的线程对象，它们是在线程模块外部启动的控制线程，例如直接来自 C 代码。虚拟线程对象功能受限；他们总是被认为是存活的和守护模式，不能被*join()* 。因为无法检测外来线程的终结，它们永远不会被删除。

class threading.**Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *, *daemon=None*)

应当始终使用关键字参数调用此构造函数。参数如下:

group 应该为 None；为了日后扩展 ThreadGroup 类实现而保留。

target 是用于*run()* 方法调用的可调用对象。默认是 None，表示不需要调用任何方法。

name 是线程名称。默认情况下，由”Thread-*N*” 格式构成一个唯一的名称，其中 *N* 是小的十进制数。

args 是用于调用目标函数的参数元组。默认是 ()。

kwargs 是用于调用目标函数的关键字参数字典。默认是 {}。

如果不是 None，*daemon* 参数将显式地设置该线程是否为守护模式。如果是 None (默认值)，线程将继承当前线程的守护模式属性。

如果子类型重载了构造函数，它一定要确保在做任何事前，先发起调用基类构造器 (Thread.__init__())。

3.3 版更變: 加入 *daemon* 参数。

start()

开始线程活动。

它在一个线程里最多只能被调用一次。它安排对象的*run()* 方法在一个独立的控制进程中调用。

如果同一个线程对象中调用这个方法的次数大于一次，会抛出*RuntimeError* 。

run()

代表线程活动的方法。

你可以在子类型里重载这个方法。标准的*run()* 方法会对作为 *target* 参数传递给该对象构造器的可调用对象（如果存在）发起调用，并附带从 *args* 和 *kwargs* 参数分别获取的位置和关键字参数。

join(*timeout=None*)

等待，直到线程终结。这会阻塞调用这个方法的线程，直到被调用*join()* 的线程终结 -- 不管是正常终结还是抛出未处理异常 -- 或者直到发生超时，超时选项是可选的。

当 *timeout* 参数存在而且不是 None 时，它应该是一个用于指定操作超时的以秒为单位的浮点数(或者分数)。因为*join()* 总是返回 None ，所以你一定要在*join()* 后调用*is_alive()* 才能判断是否发生超时 -- 如果线程仍然存活，则*join()* 超时。

当 *timeout* 参数不存在或者是 None ，这个操作会阻塞直到线程终结。

一个线程可以被*join()* 很多次。

如果尝试加入当前线程会导致死锁，*join()* 会引起*RuntimeError* 异常。如果尝试*join()* 一个尚未开始的线程，也会抛出相同的异常。

name
只用于识别的字符串。它没有语义。多个线程可以赋予相同的名称。初始名称由构造函数设置。

getName()
setName()
旧的*name* 取值/设值 API；请改为直接以特征属性方式使用它。

ident
这个线程的'线程标识符'，如果线程尚未开始则为 None 。这是个非零整数。参见*get_ident()* 函数。当一个线程退出而另外一个线程被创建，线程标识符会被复用。即使线程退出后，仍可得到标识符。

native_id
此线程的原生集成线程 ID。这是一个非负整数，或者如果线程还未启动则为 None。请参阅*get_native_id()* 函数。这表示线程 ID (TID) 已被 OS (内核) 赋值给线程。它的值可能被用来在全系统范围内唯一地标识这个特定线程（直到线程终结，在那之后该值可能会被 OS 回收再利用）。

備F: 类似于进程 ID，线程 ID 的有效期（全系统范围内保证唯一）将从线程被创建开始直到线程被终结。

可用性: 需要*get_native_id()* 函数。

3.8 版新加入.

is_alive()
返回线程是否存活。

当*run()* 方法刚开始直到*run()* 方法刚结束，这个方法返回 True 。模块函数*enumerate()* 返回包含所有存活线程的列表。

daemon
一个表示这个线程是（True）否（False）守护线程的布尔值。一定要在调用*start()* 前设置好，不然会抛出*RuntimeError* 。初始值继承于创建线程；主线程不是守护线程，因此主线程创建的所有线程默认都是*daemon* = False。

当没有存活的非守护线程时，整个 Python 程序才会退出。

isDaemon()
setDaemon()
旧的*daemon* 取值/设值 API；请改为直接以特性属性方式使用它。

17.1.3 锁对象

原始锁是一个在锁定时不属于特定线程的同步基元组件。在 Python 中，它是能用的最低级的同步基元组件，由*_thread* 扩展模块直接实现。

原始锁处于"锁定"或者"非锁定"两种状态之一。它被创建时为非锁定状态。它有两个基本方法，*acquire()* 和*release()* 。当状态为非锁定时，*acquire()* 将状态改为锁定并立即返回。当状态是锁定时，*acquire()* 将阻塞至其他线程调用*release()* 将其改为非锁定状态，然后*acquire()* 调用重置其为锁定状态并返回。*release()* 只在锁定状态下调用；它将状态改为非锁定并立即返回。如果尝试释放一个非锁定的锁，则会引发*RuntimeError* 异常。

锁同样支持上下文管理协议。

当多个线程在*acquire()* 等待状态转变为未锁定被阻塞，然后*release()* 重置状态为未锁定时，只有一个线程能继续执行；至于哪个等待线程继续执行没有定义，并且会根据实现而不同。

所有方法的执行都是原子性的。

class threading.**Lock**
实现原始锁对象的类。一旦一个线程获得一个锁，会阻塞随后尝试获得锁的线程，直到它被释放；任何线程都可以释放它。

需要注意的是 Lock 其实是一个工厂函数，返回平台支持的具体锁类中最有效的版本的实例。

acquire(*blocking=True*, *timeout=-1*)
可以阻塞或非阻塞地获得锁。

当调用时参数 *blocking* 设置为 True （缺省值），阻塞直到锁被释放，然后将锁锁定并返回 True 。

在参数 *blocking* 被设置为 False 的情况下调用，将不会发生阻塞。如果调用时 *blocking* 设为 True 会阻塞，并立即返回 False ； 否则，将锁锁定并返回 True。

当浮点型 *timeout* 参数被设置为正值调用时，只要无法获得锁，将最多阻塞 *timeout* 设定的秒数。*timeout* 参数被设置为 -1 时将无限等待。当 *blocking* 为 false 时，*timeout* 指定的值将被忽略。

如果成功获得锁，则返回 True，否则返回 False (例如发生 超时的时候)。

3.2 版更變: 新的 *timeout* 形参。

3.2 版更變: 现在如果底层线程实现支持，则可以通过 POSIX 上的信号中断锁的获取。

release()
释放一个锁。这个方法可以在任何线程中调用，不单指获得锁的线程。

当锁被锁定，将它重置为未锁定，并返回。如果其他线程正在等待这个锁解锁而被阻塞，只允许其中一个允许。

在未锁定的锁调用时，会引发*RuntimeError* 异常。

没有返回值。

locked()
如果获得了锁则返回真值。

17.1.4 递归锁对象

重入锁是一个可以被同一个线程多次获取的同步基元组件。在内部，它在基元锁的锁定/非锁定状态上附加了” 所属线程” 和” 递归等级” 的概念。在锁定状态下，某些线程拥有锁；在非锁定状态下，没有线程拥有它。

若要锁定锁，线程调用其*acquire()* 方法；一旦线程拥有了锁，方法将返回。若要解锁，线程调用*release()* 方法。*acquire()*/*release()* 对可以嵌套；只有最终*release()* (最外面一对的*release()*) 将锁解开，才能让其他线程继续处理*acquire()* 阻塞。

递归锁也支持上下文管理协议。

class threading.**RLock**
此类实现了重入锁对象。重入锁必须由获取它的线程释放。一旦线程获得了重入锁，同一个线程再次获取它将不阻塞；线程必须在每次获取它时释放一次。

需要注意的是 RLock 其实是一个工厂函数，返回平台支持的具体递归锁类中最有效的版本的实例。

acquire(*blocking=True*, *timeout=-1*)
可以阻塞或非阻塞地获得锁。

当无参数调用时：如果这个线程已经拥有锁，递归级别增加一，并立即返回。否则，如果其他线程拥有该锁，则阻塞至该锁解锁。一旦锁被解锁 (不属于任何线程)，则抢夺所有权，设置递归等级为一，并返回。如果多个线程被阻塞，等待锁被解锁，一次只有一个线程能抢到锁的所有权。在这种情况下，没有返回值。

当发起调用时将 *blocking* 参数设为真值，则执行与无参数调用时一样的操作，然后返回 `True`。

当发起调用时将 *blocking* 参数设为假值，则不进行阻塞。如果一个无参数调用将要阻塞，则立即返回 `False`；在其他情况下，执行与无参数调用时一样的操作，然后返回 `True`。

当发起调用时将浮点数的 *timeout* 参数设为正值时，只要无法获得锁，将最多阻塞 *timeout* 所指定的秒数。如果已经获得锁则返回 `True`，如果超时则返回假值。

3.2 版更變: 新的 *timeout* 形参。

release()
释放锁，自减递归等级。如果减到零，则将锁重置为非锁定状态 (不被任何线程拥有)，并且，如果其他线程正被阻塞着等待锁被解锁，则仅允许其中一个线程继续。如果自减后，递归等级仍然不是零，则锁保持锁定，仍由调用线程拥有。

只有当前线程拥有锁才能调用这个方法。如果锁被释放后调用这个方法，会引起*RuntimeError*异常。

没有返回值。

17.1.5 条件对象

条件变量总是与某种类型的锁对象相关联，锁对象可以通过传入获得，或者在缺省的情况下自动创建。当多个条件变量需要共享同一个锁时，传入一个锁很有用。锁是条件对象的一部分，你不必单独地跟踪它。

条件变量遵循上下文管理协议：使用 `with` 语句会在它包围的代码块内获取关联的锁。*acquire()* 和*release()* 方法也能调用关联锁的相关方法。

其它方法必须在持有关联的锁的情况下调用。*wait()* 方法释放锁，然后阻塞直到其它线程调用*notify()* 方法或*notify_all()* 方法唤醒它。一旦被唤醒，*wait()* 方法重新获取锁并返回。它也可以指定超时时间。

The *notify()* method wakes up one of the threads waiting for the condition variable, if any are waiting. The *notify_all()* method wakes up all threads waiting for the condition variable.

注意：*notify()* 方法和*notify_all()* 方法并不会释放锁，这意味着被唤醒的线程不会立即从它们的*wait()* 方法调用中返回，而是会在调用了*notify()* 方法或*notify_all()* 方法的线程最终放弃了锁的所有权后返回。

使用条件变量的典型编程风格是将锁用于同步某些共享状态的权限，那些对状态的某些特定改变感兴趣的线程，它们重复调用*wait()* 方法，直到看到所期望的改变发生；而对于修改状态的线程，它们将当前状态改变为可能是等待者所期待的新状态后，调用*notify()* 方法或者*notify_all()* 方法。例如，下面的代码是一个通用的无限缓冲区容量的生产者-消费者情形：

```
# Consume one item
with cv:
    while not an_item_is_available():
        cv.wait()
    get_an_available_item()

# Produce one item
with cv:
    make_an_item_available()
    cv.notify()
```

使用 `while` 循环检查所要求的条件成立与否是有必要的，因为*wait()* 方法可能要经过不确定长度的时间后才会返回，而此时导致*notify()* 方法调用的那个条件可能已经不再成立。这是多线程编程所固有的问题。*wait_for()* 方法可自动化条件检查，并简化超时计算。

```
# Consume an item
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

选择*notify()* 还是*notify_all()* ，取决于一次状态改变是只能被一个还是能被多个等待线程所用。例如在一个典型的生产者-消费者情形中，添加一个项目到缓冲区只需唤醒一个消费者线程。

class threading.**Condition**(*lock=None*)

实现条件变量对象的类。一个条件变量对象允许一个或多个线程在被其它线程所通知之前进行等待。

如果给出了非 None 的 *lock* 参数，则它必须为*Lock* 或者*RLock* 对象，并且它将被用作底层锁。否则，将会创建新的*RLock* 对象，并将其用作底层锁。

3.3 版更變: 从工厂函数变为类。

acquire(**args*)

请求底层锁。此方法调用底层锁的相应方法，返回值是底层锁相应方法的返回值。

release()

释放底层锁。此方法调用底层锁的相应方法。没有返回值。

wait(*timeout=None*)

等待直到被通知或发生超时。如果线程在调用此方法时没有获得锁，将会引发*RuntimeError* 异常。

这个方法释放底层锁，然后阻塞，直到在另外一个线程中调用同一个条件变量的*notify()* 或*notify_all()* 唤醒它，或者直到可选的超时发生。一旦被唤醒或者超时，它重新获得锁并返回。

当提供了 *timeout* 参数且不是 None 时，它应该是一个浮点数，代表操作的超时时间，以秒为单位（可以为小数）。

当底层锁是个*RLock* ，不会使用它的*release()* 方法释放锁，因为当它被递归多次获取时，实际上可能无法解锁。相反，使用了*RLock* 类的内部接口，即使多次递归获取它也能解锁它。然后，在重新获取锁时，使用另一个内部接口来恢复递归级别。

返回 True ，除非提供的 *timeout* 过期，这种情况下返回 False。

3.2 版更變: 很明显，方法总是返回 None。

wait_for(*predicate*, *timeout=None*)

等待，直到条件计算为真。*predicate* 应该是一个可调用对象而且它的返回值可被解释为一个布尔值。可以提供 *timeout* 参数给出最大等待时间。

这个实用方法会重复地调用*wait()* 直到满足判断式或者发生超时。返回值是判断式最后一个返回值，而且如果方法发生超时会返回 False 。

忽略超时功能，调用此方法大致相当于编写:

```
while not predicate():
    cv.wait()
```

因此，规则同样适用于*wait()* ： 锁必须在被调用时保持获取，并在返回时重新获取。随着锁定执行判断式。

3.2 版新加入.

notify(*n=1*)

默认唤醒一个等待这个条件的线程。如果调用线程在没有获得锁的情况下调用这个方法，会引发*RuntimeError* 异常。

这个方法唤醒最多 *n* 个正在等待这个条件变量的线程；如果没有线程在等待，这是一个空操作。

当前实现中，如果至少有 *n* 个线程正在等待，准确唤醒 *n* 个线程。但是依赖这个行为并不安全。未来，优化的实现有时会唤醒超过 *n* 个线程。

注意：被唤醒的线程并没有真正恢复到它调用的*wait()*，直到它可以重新获得锁。因为*notify()*不释放锁，其调用者才应该这样做。

notify_all()
唤醒所有正在等待这个条件的线程。这个方法行为与*notify()*相似，但并不只唤醒单一线程，而是唤醒所有等待线程。如果调用线程在调用这个方法时没有获得锁，会引发*RuntimeError*异常。

17.1.6 信号量对象

这是计算机科学史上最古老的同步原语之一，早期的荷兰科学家 Edsger W. Dijkstra 发明了它。（他使用名称 `P()` 和 `V()` 而不是*acquire()*和*release()*）。

一个信号量管理一个内部计数器，该计数器因*acquire()*方法的调用而递减，因*release()*方法的调用而递增。计数器的值永远不会小于零；当*acquire()*方法发现计数器为零时，将会阻塞，直到其它线程调用*release()*方法。

信号量对象也支持上下文管理协议。

class threading.**Semaphore**(*value=1*)
该类实现信号量对象。信号量对象管理一个原子性的计数器，代表*release()*方法的调用次数减去*acquire()*的调用次数再加上一个初始值。如果需要，*acquire()*方法将会阻塞直到可以返回而不会使得计数器变成负数。在没有显式给出 *value* 的值时，默认为 1。

可选参数 *value* 赋予内部计数器初始值，默认值为 `1`。如果 *value* 被赋予小于 0 的值，将会引发*ValueError*异常。

3.3 版更變: 从工厂函数变为类。

acquire(*blocking=True*, *timeout=None*)
获取一个信号量。

在不带参数的情况下调用时：

- 如果在进入时内部计数器的值大于零，则将其减一并立即返回 `True`.
- 如果在进入时内部计数器的值为零，则将会阻塞直到被对*release()*的调用唤醒。一旦被唤醒（并且计数器的值大于 0），则将计数器减 1 并返回 `True`。每次对*release()*的调用将只唤醒一个线程。线程被唤醒的次序是不可确定的。

当发起调用时将 *blocking* 设为假值，则不进行阻塞。如果一个无参数调用将要阻塞，则立即返回 `False`；在其他情况下，执行与无参数调用时一样的操作，然后返回 `True`。

当发起调用时如果 *timeout* 不为 `None`，则它将阻塞最多 *timeout* 秒。请求在此时段时未能成功完成获取则将返回 `False`。在其他情况下返回 `True`。

3.2 版更變: 新的 *timeout* 形参。

release(*n=1*)
释放一个信号量，将内部计数器的值增加 *n*。当进入时值为零且有其他线程正在等待它再次变为大于零时，则唤醒那 *n* 个线程。

3.9 版更變: 增加了 *n* 形参以一次性释放多个等待线程。

class threading.**BoundedSemaphore**(*value=1*)
该类实现有界信号量。有界信号量通过检查以确保它当前的值不会超过初始值。如果超过了初始值，将会引发*ValueError*异常。在大多情况下，信号量用于保护数量有限的资源。如果信号量被释放的次数过多，则表明出现了错误。没有指定时，*value* 的值默认为 1。

3.3 版更變: 从工厂函数变为类。

Semaphore 例子

信号量通常用于保护数量有限的资源，例如数据库服务器。在资源数量固定的任何情况下，都应该使用有界信号量。在生成任何工作线程前，应该在主线程中初始化信号量。

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
```

工作线程生成后，当需要连接服务器时，这些线程将调用信号量的 acquire 和 release 方法：

```
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

使用有界信号量能减少这种编程错误：信号量的释放次数多于其请求次数。

17.1.7 事件对象

这是线程之间通信的最简单机制之一：一个线程发出事件信号，而其他线程等待该信号。

一个事件对象管理一个内部标识，调用*set()* 方法可将其设置为 true ，调用*clear()* 方法可将其设置为 false ，调用*wait()* 方法将进入阻塞直到标识为 true 。

class threading.**Event**

实现事件对象的类。事件对象管理一个内部标识，调用*set()* 方法可将其设置为 true。调用*clear()* 方法可将其设置为 false 。调用*wait()* 方法将进入阻塞直到标识为 true。这个标识初始时为 false 。

3.3 版更變: 从工厂函数变为类。

is_set()

当且仅当内部标识为 true 时返回 `True` 。

set()

将内部标识设置为 true 。所有正在等待这个事件的线程将被唤醒。当标识为 true 时，调用*wait()* 方法的线程不会被被阻塞。

clear()

将内部标识设置为 false 。之后调用*wait()* 方法的线程将会被阻塞，直到调用*set()* 方法将内部标识再次设置为 true 。

wait(*timeout=None*)

阻塞线程直到内部变量为 true 。如果调用时内部标识为 true，将立即返回。否则将阻塞线程，直到调用*set()* 方法将标识设置为 true 或者发生可选的超时。

当提供了 timeout 参数且不是 `None` 时，它应该是一个浮点数，代表操作的超时时间，以秒为单位（可以为小数）。

当且仅当内部旗标在等待调用之前或者等待开始之后被设为真值时此方法将返回 `True`，也就是说，它将总是返回 `True` 除非设定了超时且操作发生了超时。

3.1 版更變: 很明显，方法总是返回 `None`。

17.1.8 定时器对象

此类表示一个操作应该在等待一定的时间之后运行 --- 相当于一个定时器。*Timer* 类是*Thread* 类的子类，因此可以像一个自定义线程一样工作。

与线程一样，通过调用 start() 方法启动定时器。而*cancel()* 方法可以停止计时器（在计时结束前），定时器在执行其操作之前等待的时间间隔可能与用户指定的时间间隔不完全相同。

例如:

```
def hello():
    print("hello, world")

t = Timer(30.0, hello)
t.start()  # after 30 seconds, "hello, world" will be printed
```

class threading.**Timer**(*interval*, *function*, *args=None*, *kwargs=None*)

创建一个定时器，在经过 *interval* 秒的间隔事件后，将会用参数 *args* 和关键字参数 *kwargs* 调用 *function*。如果 *args* 为 None （默认值），则会使用一个空列表。如果 *kwargs* 为 None （默认值），则会使用一个空字典。

3.3 版更變: 从工厂函数变为类。

cancel()

停止定时器并取消执行计时器将要执行的操作。仅当计时器仍处于等待状态时有效。

17.1.9 栅栏对象

3.2 版新加入.

栅栏类提供一个简单的同步原语，用于应对固定数量的线程需要彼此相互等待的情况。线程调用*wait()* 方法后将阻塞，直到所有线程都调用了*wait()* 方法。此时所有线程将被同时释放。

栅栏对象可以被多次使用，但进程的数量不能改变。

这是一个使用简便的方法实现客户端进程与服务端进程同步的例子:

```
b = Barrier(2, timeout=5)

def server():
    start_server()
    b.wait()
    while True:
        connection = accept_connection()
        process_server_connection(connection)

def client():
    b.wait()
    while True:
        connection = make_connection()
        process_client_connection(connection)
```

class threading.**Barrier**(*parties*, *action=None*, *timeout=None*)

创建一个需要 *parties* 个线程的栅栏对象。如果提供了可调用的 *action* 参数，它会在所有线程被释放时在其中一个线程中自动调用。*timeout* 是默认的超时时间，如果没有在*wait()* 方法中指定超时时间的话。

wait(*timeout=None*)
冲出栅栏。当栅栏中所有线程都已经调用了这个函数，它们将同时被释放。如果提供了 *timeout* 参数，这里的 *timeout* 参数优先于创建栅栏对象时提供的 *timeout* 参数。

函数返回值是一个整数，取值范围在 0 到 *parties* -- 1，在每个线程中的返回值不相同。可用于从所有线程中选择唯一的一个线程执行一些特别的工作。例如:

```
i = barrier.wait()
if i == 0:
    # Only one thread needs to print this
    print("passed the barrier")
```

如果创建栅栏对象时在构造函数中提供了 *action* 参数，它将在其中一个线程释放前被调用。如果此调用引发了异常，栅栏对象将进入损坏态。

如果发生了超时，栅栏对象将进入破损态。

如果栅栏对象进入破损态，或重置栅栏时仍有线程等待释放，将会引发*BrokenBarrierError*异常。

reset()
重置栅栏为默认的初始态。如果栅栏中仍有线程等待释放，这些线程将会收到*BrokenBarrierError*异常。

请注意使用此函数时，如果存在状态未知的其他线程，则可能需要执行外部同步。如果栅栏已损坏则最好将其废弃并新建一个。

abort()
使栅栏处于损坏状态。这将导致任何现有和未来对*wait()*的调用失败并引发*BrokenBarrierError*。例如可以在需要中止某个线程时使用此方法，以避免应用程序的死锁。

更好的方式是：创建栅栏时提供一个合理的超时时间，来自动避免某个线程出错。

parties
冲出栅栏所需要的线程数量。

n_waiting
当前时刻正在栅栏中阻塞的线程数量。

broken
一个布尔值，值为 `True` 表明栅栏为破损态。

exception threading.**BrokenBarrierError**
异常类，是*RuntimeError*异常的子类，在*Barrier*对象重置时仍有线程阻塞时和对象进入破损态时被引发。

17.1.10 在 with 语句中使用锁、条件和信号量

这个模块提供的带有 `acquire()` 和 `release()` 方法的对象，可以被用作 `with` 语句的上下文管理器。当进入语句块时 `acquire()` 方法会被调用，退出语句块时 `release()` 会被调用。因此，以下片段:

```
with some_lock:
    # do something...
```

相当于:

```
some_lock.acquire()
try:
    # do something...
finally:
    some_lock.release()
```

现在*Lock* 、*RLock* 、*Condition* 、*Semaphore* 和*BoundedSemaphore* 对象可以用作 with 语句的上下文管理器。

17.2 multiprocessing --- 基于进程的并行

源代码 Lib/multiprocessing/

17.2.1 简介

multiprocessing 是一个支持使用与*threading* 模块类似的 API 来产生进程的包。*multiprocessing* 包同时提供了本地和远程并发操作，通过使用子进程而非线程有效地绕过了*全局解释器锁*。因此，*multiprocessing* 模块允许程序员充分利用给定机器上的多个处理器。它在 Unix 和 Windows 上均可运行。

multiprocessing 模块还引入了在*threading* 模块中没有的 API。一个主要的例子就是*Pool* 对象，它提供了一种快捷的方法，赋予函数并行化处理一系列输入值的能力，可以将输入数据分配给不同进程处理（数据并行）。下面的例子演示了在模块中定义此类函数的常见做法，以便子进程可以成功导入该模块。这个数据并行的基本例子使用了*Pool* ，

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

将在标准输出中打印

```
[1, 4, 9]
```

Process 类

在*multiprocessing* 中，通过创建一个*Process* 对象然后调用它的 start() 方法来生成进程。*Process* 和*threading.Thread* API 相同。一个简单的多进程程序示例是:

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
```

(下页继续)

(繼續上一頁)

```
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

要显示所涉及的各个进程 ID，这是一个扩展示例:

```
from multiprocessing import Process
import os

def info(title):
    print(title)
    print('module name:', __name__)
    print('parent process:', os.getppid())
    print('process id:', os.getpid())

def f(name):
    info('function f')
    print('hello', name)

if __name__ == '__main__':
    info('main line')
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

关于为什么 `if __name__ == '__main__'` 部分是必需的解释，请参见*编程指导*。

上下文和启动方法

根据不同的平台，*multiprocessing* 支持三种启动进程的方法。这些 *启动方法*有

spawn 父进程会启动一个全新的 python 解释器进程。子进程将只继承那些运行进程对象的*run()* 方法所必需的资源。特别地，来自父进程的非必需文件描述符和句柄将不会被继承。使用此方法启动进程相比使用 *fork* 或 *forkserver* 要慢上许多。

可在 Unix 和 Windows 上使用。Windows 上的默认设置。

fork 父进程使用*os.fork()* 来产生 Python 解释器分叉。子进程在开始时实际上与父进程相同。父进程的所有资源都由子进程继承。请注意，安全分叉多线程进程是棘手的。

只存在于 Unix。Unix 中的默认值。

forkserver 程序启动并选择 *forkserver* 启动方法时，将启动服务器进程。从那时起，每当需要一个新进程时，父进程就会连接到服务器并请求它分叉一个新进程。分叉服务器进程是单线程的，因此使用*os.fork()* 是安全的。没有不必要的资源被继承。

可在 Unix 平台上使用，支持通过 Unix 管道传递文件描述符。

3.8 版更變: 对于 macOS，*spawn* 启动方式是默认方式。因为 *fork* 可能导致 subprocess 崩溃，被认为是不安全的，查看 bpo-33725 。

3.4 版更變: 在所有 unix 平台上添加支持了 *spawn* ，并且为一些 unix 平台添加了 *forkserver* 。在 Windows 上子进程不再继承所有可继承的父进程句柄。

在 Unix 上通过 *spawn* 和 *forkserver* 方式启动多进程会同时启动一个 *资源追踪进程*，负责追踪当前程序的进程产生的、并且不再被使用的命名系统资源 (如命名信号量以及*SharedMemory* 对象)。当所有进程退出后，资源追踪会负责释放这些仍被追踪的的对象。通常情况下是不会有这种对象的，但是假如一个子进程被某个信号杀死，就可能存在这一类资源的“泄露”情况。(泄露的信号量以及共享内存不会被释放，直到下一次系

统重启，对于这两类资源来说，这是一个比较大的问题，因为操作系统允许的命名信号量的数量是有限的，而共享内存也会占据主内存的一片空间）

要选择一个启动方法，你应该在主模块的 `if __name__ == '__main__'` 子句中调用*set_start_method()*。例如：

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    mp.set_start_method('spawn')
    q = mp.Queue()
    p = mp.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

在程序中*set_start_method()*不应该被多次调用。

或者，你可以使用*get_context()*来获取上下文对象。上下文对象与 multiprocessing 模块具有相同的 API，并允许在同一程序中使用多种启动方法。：

```
import multiprocessing as mp

def foo(q):
    q.put('hello')

if __name__ == '__main__':
    ctx = mp.get_context('spawn')
    q = ctx.Queue()
    p = ctx.Process(target=foo, args=(q,))
    p.start()
    print(q.get())
    p.join()
```

请注意，对象在不同上下文创建的进程间可能并不兼容。特别是，使用 *fork* 上下文创建的锁不能传递给使用 *spawn* 或 *forkserver* 启动方法启动的进程。

想要使用特定启动方法的库应该使用*get_context()*以避免干扰库用户的选择。

警告： `'spawn'` 和 `'forkserver'` 启动方法当前不能在 Unix 上和“冻结的”可执行内容一同使用（例如，有类似 **PyInstaller** 和 **cx_Freeze** 的包产生的二进制文件）。`'fork'` 启动方法可以使用。

在进程之间交换对象

multiprocessing 支持进程之间的两种通信通道：

队列

Queue 类是一个近似*queue.Queue*的克隆。例如：

```
from multiprocessing import Process, Queue

def f(q):
```

(下页继续)

(繼續上一頁)

```
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

队列是线程和进程安全的。

管道

Pipe() 函数返回一个由管道连接的连接对象，默认情况下是双工（双向）。例如:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

返回的两个连接对象*Pipe()* 表示管道的两端。每个连接对象都有 `send()` 和 `recv()` 方法（相互之间的）。请注意，如果两个进程（或线程）同时尝试读取或写入管道的 同一端，则管道中的数据可能会损坏。当然，在不同进程中同时使用管道的不同端的情况下不存在损坏的风险。

进程间同步

对于所有在*threading* 存在的同步原语，*multiprocessing* 中都有类似的等价物。例如，可以使用锁来确保一次只有一个进程打印到标准输出:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

不使用锁的情况下，来自于多进程的输出很容易产生混淆。

进程间共享状态

如上所述，在进行并发编程时，通常最好尽量避免使用共享状态。使用多个进程时尤其如此。

但是，如果你真的需要使用一些共享数据，那么*multiprocessing* 提供了两种方法。

共享内存

可以使用*Value* 或*Array* 将数据存储在共享内存映射中。例如，以下代码:

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
        a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))

    p = Process(target=f, args=(num, arr))
    p.start()
    p.join()

    print(num.value)
    print(arr[:])
```

将打印

```
3.1415927
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

创建 num 和 arr 时使用的 'd' 和 'i' 参数是*array* 模块使用的类型的 typecode： 'd' 表示双精度浮点数，'i' 表示有符号整数。这些共享对象将是进程和线程安全的。

为了更灵活地使用共享内存，可以使用*multiprocessing.sharedctypes* 模块，该模块支持创建从共享内存分配的任意 ctypes 对象。

服务进程

由*Manager()* 返回的管理器对象控制一个服务进程，该进程保存 Python 对象并允许其他进程使用代理操作它们。

Manager() 返回的管理器支持类型: *list* 、*dict* 、*Namespace* 、*Lock* 、*RLock* 、*Semaphore* 、*BoundedSemaphore* 、*Condition* 、*Event* 、*Barrier* 、*Queue* 、*Value* 和*Array* 。例如

```
from multiprocessing import Process, Manager

def f(d, l):
    d[1] = '1'
    d['2'] = 2
    d[0.25] = None
    l.reverse()

if __name__ == '__main__':
    with Manager() as manager:
        d = manager.dict()
        l = manager.list(range(10))
```

(下页继续)

(繼續上一頁)

```
        p = Process(target=f, args=(d, l))
        p.start()
        p.join()

        print(d)
        print(l)
```

将打印

```
{0.25: None, 1: '1', '2': 2}
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

使用服务进程的管理器比使用共享内存对象更灵活，因为它们可以支持任意对象类型。此外，单个管理器可以通过网络由不同计算机上的进程共享。但是，它们比使用共享内存慢。

使用工作进程

Pool 类表示一个工作进程池。它具有允许以几种不同方式将任务分配到工作进程的方法。

例如:

```
from multiprocessing import Pool, TimeoutError
import time
import os

def f(x):
    return x*x

if __name__ == '__main__':
    # start 4 worker processes
    with Pool(processes=4) as pool:

        # print "[0, 1, 4,..., 81]"
        print(pool.map(f, range(10)))

        # print same numbers in arbitrary order
        for i in pool.imap_unordered(f, range(10)):
            print(i)

        # evaluate "f(20)" asynchronously
        res = pool.apply_async(f, (20,))      # runs in *only* one process
        print(res.get(timeout=1))             # prints "400"

        # evaluate "os.getpid()" asynchronously
        res = pool.apply_async(os.getpid, ()) # runs in *only* one process
        print(res.get(timeout=1))             # prints the PID of that process

        # launching multiple evaluations asynchronously *may* use more processes
        multiple_results = [pool.apply_async(os.getpid, ()) for i in range(4)]
        print([res.get(timeout=1) for res in multiple_results])

        # make a single worker sleep for 10 secs
        res = pool.apply_async(time.sleep, (10,))
        try:
            print(res.get(timeout=1))
        except TimeoutError:
```

(下页继续)

(繼續上一頁)

```
            print("We lacked patience and got a multiprocessing.TimeoutError")

        print("For the moment, the pool remains available for more work")

    # exiting the 'with'-block has stopped the pool
    print("Now the pool is closed and no longer available")
```

请注意，进程池的方法只能由创建它的进程使用。

備𝔽: 这个包中的功能要求子进程可以导入 `__main__` 模块。虽然这在*编程指导* 中有描述，但还是需要提前说明一下。这意味着一些示例在交互式解释器中不起作用，比如*multiprocessing.pool.Pool* 示例。例如:

```
>>> from multiprocessing import Pool
>>> p = Pool(5)
>>> def f(x):
...     return x*x
...
>>> with p:
...   p.map(f, [1,2,3])
Process PoolWorker-1:
Process PoolWorker-2:
Process PoolWorker-3:
Traceback (most recent call last):
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
AttributeError: 'module' object has no attribute 'f'
```

(如果尝试执行上面的代码，它会以一种半随机的方式将三个完整的堆栈内容交替输出，然后你只能以某种方式停止父进程。)

17.2.2 参考

multiprocessing 包主要复制了*threading* 模块的 API。

Process 和异常

class multiprocessing.**Process**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, ***, *daemon=None*)

进程对象表示在单独进程中运行的活动。*Process* 类拥有和*threading.Thread* 等价的大部分方法。

应始终使用关键字参数调用构造函数。*group* 应该始终是 None；它仅用于兼容*threading.Thread*。*target* 是由*run()* 方法调用的可调用对象。它默认为 None，意味着什么都没有被调用。*name* 是进程名称（有关详细信息，请参阅*name*）。*args* 是目标调用的参数元组。*kwargs* 是目标调用的关键字参数字典。如果提供，则键参数 *daemon* 将进程*daemon* 标志设置为 True 或 False。如果是 None（默认值），则该标志将从创建的进程继承。

默认情况下，不会将任何参数传递给 *target*。

如果子类重写构造函数，它必须确保它在对进程执行任何其他操作之前调用基类构造函数（Process.__init__()）。

3.3 版更變: 加入 *daemon* 参数。

run()
表示进程活动的方法。

你可以在子类中重载此方法。标准*run()* 方法调用传递给对象构造函数的可调用对象作为目标参数（如果有），分别从 *args* 和 *kwargs* 参数中获取顺序和关键字参数。

start()
启动进程活动。

这个方法每个进程对象最多只能调用一次。它会将对象的*run()* 方法安排在一个单独的进程中调用。

join([*timeout*])
如果可选参数 *timeout* 是 None （默认值），则该方法将阻塞，直到调用*join()* 方法的进程终止。如果 *timeout* 是一个正数，它最多会阻塞 *timeout* 秒。请注意，如果进程终止或方法超时，则该方法返回 None 。检查进程的*exitcode* 以确定它是否终止。

一个进程可以被 join 多次。

进程无法 join 自身，因为这会导致死锁。尝试在启动进程之前 join 进程是错误的。

name
进程的名称。该名称是一个字符串，仅用于识别目的。它没有语义。可以为多个进程指定相同的名称。

初始名称由构造器设定。如果没有为构造器提供显式名称，则会构造一个形式为'Process-N_1:N_2:...:N_k' 的名称，其中每个 N_k 是其父亲的第 N 个孩子。

is_alive()
返回进程是否还活着。

粗略地说，从*start()* 方法返回到子进程终止之前，进程对象仍处于活动状态。

daemon
进程的守护标志，一个布尔值。这必须在*start()* 被调用之前设置。

初始值继承自创建进程。

当进程退出时，它会尝试终止其所有守护进程子进程。

请注意，不允许在守护进程中创建子进程。这是因为当守护进程由于父进程退出而中断时，其子进程会变成孤儿进程。另外，这些 **不是** Unix 守护进程或服务，它们是正常进程，如果非守护进程已经退出，它们将被终止（并且不被合并）。

除了*threading.Thread* API ， *Process* 对象还支持以下属性和方法：

pid
返回进程 ID。在生成该进程之前，这将是 None 。

exitcode
The child's exit code. This will be None if the process has not yet terminated.

If the child's *run()* method returned normally, the exit code will be 0. If it terminated via *sys.exit()* with an integer argument *N*, the exit code will be *N*.

If the child terminated due to an exception not caught within *run()*, the exit code will be 1. If it was terminated by signal *N*, the exit code will be the negative value *-N*.

authkey
进程的身份验证密钥（字节字符串）。

当*multiprocessing* 初始化时，主进程使用*os.urandom()* 分配一个随机字符串。

当创建*Process* 对象时，它将继承其父进程的身份验证密钥，尽管可以通过将*authkey* 设置为另一个字节字符串来更改。

参见认证密码 。

sentinel
系统对象的数字句柄，当进程结束时将变为”ready” 。

如果要使用*multiprocessing.connection.wait()* 一次等待多个事件，可以使用此值。否则调用*join()* 更简单。

在 Windows 上，这是一个操作系统句柄，可以与 WaitForSingleObject 和 WaitForMultipleObjects 系列 API 调用一起使用。在 Unix 上，这是一个文件描述符，可以使用来自*select* 模块的原语。

3.3 版新加入.

terminate()
终止进程。在 Unix 上，这是使用 SIGTERM 信号完成的；在 Windows 上使用 TerminateProcess() 。请注意，不会执行退出处理程序和 finally 子句等。

请注意，进程的后代进程将不会被终止——它们将简单地变成孤立的。

> **警告:** 如果在关联进程使用管道或队列时使用此方法，则管道或队列可能会损坏，并可能无法被其他进程使用。类似地，如果进程已获得锁或信号量等，则终止它可能导致其他进程死锁。

kill()
与*terminate()* 相同，但在 Unix 上使用 SIGKILL 信号。

3.7 版新加入.

close()
关闭*Process* 对象，释放与之关联的所有资源。如果底层进程仍在运行，则会引发*ValueError* 。一旦*close()* 成功返回，*Process* 对象的大多数其他方法和属性将引发*ValueError* 。

3.7 版新加入.

注意*start()* 、*join()* 、*is_alive()* 、*terminate()* 和*exitcode* 方法只能由创建进程对象的进程调用。

Process 一些方法的示例用法：

```
>>> import multiprocessing, time, signal
>>> p = multiprocessing.Process(target=time.sleep, args=(1000,))
>>> print(p, p.is_alive())
<Process ... initial> False
>>> p.start()
>>> print(p, p.is_alive())
<Process ... started> True
>>> p.terminate()
>>> time.sleep(0.1)
>>> print(p, p.is_alive())
<Process ... stopped exitcode=-SIGTERM> False
>>> p.exitcode == -signal.SIGTERM
True
```

exception multiprocessing.**ProcessError**
所有*multiprocessing* 异常的基类。

exception multiprocessing.**BufferTooShort**
当提供的缓冲区对象太小而无法读取消息时，Connection.recv_bytes_into() 引发的异常。

如果 e 是一个*BufferTooShort* 实例，那么 e.args[0] 将把消息作为字节字符串给出。

exception multiprocessing.**AuthenticationError**
出现身份验证错误时引发。

exception multiprocessing.**TimeoutError**
有超时的方法超时时引发。

管道和队列

使用多进程时，一般使用消息机制实现进程间通信，尽可能避免使用同步原语，例如锁。

消息机制包含：*Pipe()* (可以用于在两个进程间传递消息)，以及队列 (能够在多个生产者和消费者之间通信)。

Queue, *SimpleQueue* 以及*JoinableQueue* 都是多生产者，多消费者，并且实现了 FIFO 的队列类型，其表现与标准库中的*queue.Queue* 类相似。不同之处在于*Queue* 缺少标准库的*queue.Queue* 从 Python 2.5 开始引入的*task_done()* 和*join()* 方法。

如果你使用了*JoinableQueue* ，那么你 **必须** 对每个已经移出队列的任务调用*JoinableQueue.task_done()*。不然的话用于统计未完成任务的信号量最终会溢出并抛出异常。

另外还可以通过使用一个管理器对象创建一个共享队列，详见管理器 。

備🄴: *multiprocessing* 使用了普通的*queue.Empty* 和*queue.Full* 异常去表示超时。你需要从*queue* 中导入它们，因为它们并不在*multiprocessing* 的命名空间中。

備🄴: 当一个对象被放入一个队列中时，这个对象首先会被一个后台线程用 pickle 序列化，并将序列化后的数据通过一个底层管道的管道传递到队列中。这种做法会有点让人惊讶，但一般不会出现什么问题。如果它们确实妨碍了你，你可以使用一个由管理器*manager* 创建的队列替换它。

(1) 将一个对象放入一个空队列后，可能需要极小的延迟，队列的方法*empty()* 才会返回*False* 。而*get_nowait()* 可以不抛出*queue.Empty* 直接返回。

(2) 如果有多个进程同时将对象放入队列，那么在队列的另一端接受到的对象可能是无序的。但是由同一个进程放入的多个对象的顺序在另一端输出时总是一样的。

警告: 如果一个进程在尝试使用*Queue* 期间被*Process.terminate()* 或*os.kill()* 调用终止了，那么队列中的数据很可能被破坏。这可能导致其他进程在尝试使用该队列时发生异常。

警告: 正如刚才提到的，如果一个子进程将一些对象放进队列中 (并且它没有用*JoinableQueue.cancel_join_thread* 方法)，那么这个进程在所有缓冲区的对象被刷新进管道之前，是不会终止的。

这意味着，除非你确定所有放入队列中的对象都已经被消费了，否则如果你试图等待这个进程，你可能会陷入死锁中。相似地，如果该子进程不是后台进程，那么父进程可能在试图等待所有非后台进程退出时挂起。

注意用管理器创建的队列不存在这个问题，详见编程指导 。

该示例 展示了如何使用队列实现进程间通信。

multiprocessing.**Pipe**([*duplex*])
返回一对*Connection* 对象 (conn1, conn2) ，分别表示管道的两端。

如果 *duplex* 被置为 `True` (默认值)，那么该管道是双向的。如果 *duplex* 被置为 `False` ，那么该管道是单向的，即 `conn1` 只能用于接收消息，而 `conn2` 仅能用于发送消息。

class multiprocessing.**Queue**([*maxsize*])
返回一个使用一个管道和少量锁和信号量实现的共享队列实例。当一个进程将一个对象放进队列中时，一个写入线程会启动并将对象从缓冲区写入管道中。

一旦超时，将抛出标准库*queue* 模块中常见的异常*queue.Empty* 和*queue.Full*。

除了*task_done()* 和*join()* 之外，*Queue* 实现了标准库类*queue.Queue* 中所有的方法。

qsize()
返回队列的大致长度。由于多线程或者多进程的上下文，这个数字是不可靠的。

Note that this may raise *NotImplementedError* on Unix platforms like macOS where `sem_getvalue()` is not implemented.

empty()
如果队列是空的，返回 `True` ，反之返回 `False` 。由于多线程或多进程的环境，该状态是不可靠的。

full()
如果队列是满的，返回 `True` ，反之返回 `False` 。由于多线程或多进程的环境，该状态是不可靠的。

put(*obj*[, *block*[, *timeout*]])
将 obj 放入队列。如果可选参数 *block* 是 `True` (默认值) 而且 *timeout* 是 `None` (默认值), 将会阻塞当前进程，直到有空的缓冲槽。如果 *timeout* 是正数，将会在阻塞了最多 *timeout* 秒之后还是没有可用的缓冲槽时抛出*queue.Full* 异常。反之 (*block* 是 `False` 时)，仅当有可用缓冲槽时才放入对象，否则抛出*queue.Full* 异常 (在这种情形下 *timeout* 参数会被忽略)。

3.8 版更變: 如果队列已经关闭，会抛出*ValueError* 而不是*AssertionError* 。

put_nowait(*obj*)
相当于 `put(obj, False)`。

get([*block*[, *timeout*]])
从队列中取出并返回对象。如果可选参数 *block* 是 `True` (默认值) 而且 *timeout* 是 `None` (默认值), 将会阻塞当前进程，直到队列中出现可用的对象。如果 *timeout* 是正数，将会在阻塞了最多 *timeout* 秒之后还是没有可用的对象时抛出*queue.Empty* 异常。反之 (*block* 是 `False` 时)，仅当有可用对象能够取出时返回，否则抛出*queue.Empty* 异常 (在这种情形下 *timeout* 参数会被忽略)。

3.8 版更變: 如果队列已经关闭，会抛出*ValueError* 而不是*OSError* 。

get_nowait()
相当于 `get(False)`。

multiprocessing.Queue 类有一些在*queue.Queue* 类中没有出现的方法。这些方法在大多数情形下并不是必须的。

close()
指示当前进程将不会再往队列中放入对象。一旦所有缓冲区中的数据被写入管道之后，后台的线程会退出。这个方法在队列被 gc 回收时会自动调用。

join_thread()
等待后台线程。这个方法仅在调用了*close()* 方法之后可用。这会阻塞当前进程，直到后台线程退出，确保所有缓冲区中的数据都被写入管道中。

默认情况下，如果一个不是队列创建者的进程试图退出，它会尝试等待这个队列的后台线程。这个进程可以使用*cancel_join_thread()* 让*join_thread()* 方法什么都不做直接跳过。

cancel_join_thread()
防止*join_thread()* 方法阻塞当前进程。具体而言，这防止进程退出时自动等待后台线程退出。详见*join_thread()*。

可能这个方法称为"allow_exit_without_flush() "会更好。这有可能会导致正在排队进入队列的数据丢失，大多数情况下你不需要用到这个方法，仅当你不关心底层管道中可能丢失的数据，只是希望进程能够马上退出时使用。

備𝔽: 该类的功能依赖于宿主操作系统具有可用的共享信号量实现。否则该类将被禁用，任何试图实例化一个*Queue* 对象的操作都会抛出*ImportError* 异常，更多信息详见 bpo-3770 。后续说明的任何专用队列对象亦如此。

---

class multiprocessing.**SimpleQueue**
这是一个简化的*Queue* 类的实现，很像带锁的*Pipe* 。

close()
关闭队列：释放内部资源。

队列在被关闭后就不可再被使用。例如不可再调用*get()*, *put()* 和*empty()* 等方法。

3.9 版新加入.

empty()
如果队列为空返回 True ，否则返回 False 。

get()
从队列中移出并返回一个对象。

put(*item*)
将 *item* 放入队列。

class multiprocessing.**JoinableQueue**([*maxsize*])
JoinableQueue 类是*Queue* 的子类，额外添加了*task_done()* 和*join()* 方法。

task_done()
指出之前进入队列的任务已经完成。由队列的消费者进程使用。对于每次调用*get()* 获取的任务，执行完成后调用*task_done()* 告诉队列该任务已经处理完成。

如果*join()* 方法正在阻塞之中，该方法会在所有对象都被处理完的时候返回 (即对之前使用*put()* 放进队列中的所有对象都已经返回了对应的*task_done()*) 。

如果被调用的次数多于放入队列中的项目数量，将引发*ValueError* 异常。

join()
阻塞至队列中所有的元素都被接收和处理完毕。

当条目添加到队列的时候，未完成任务的计数就会增加。每当消费者进程调用*task_done()* 表示这个条目已经被回收，该条目所有工作已经完成，未完成计数就会减少。当未完成计数降到零的时候，join() 阻塞被解除。

杂项

multiprocessing.**active_children**()
返回当前进程存活的子进程的列表。

调用该方法有“等待”已经结束的进程的副作用。

multiprocessing.**cpu_count**()
返回系统的 CPU 数量。

该数量不同于当前进程可以使用的 CPU 数量。可用的 CPU 数量可以由 `len(os.sched_getaffinity(0))` 方法获得。

When the number of CPUs cannot be determined a *NotImplementedError* is raised.

也参考:

os.cpu_count()

multiprocessing.**current_process**()
返回与当前进程相对应的*Process* 对象。

和*threading.current_thread()* 相同。

multiprocessing.**parent_process**()
返回父进程*Process* 对象，和父进程调用*current_process()* 返回的对象一样。如果一个进程已经是主进程，parent_process 会返回 None.

3.8 版新加入.

multiprocessing.**freeze_support**()
为使用了*multiprocessing* 的程序，提供冻结以产生 Windows 可执行文件的支持。(在 **py2exe**, **PyInstaller** 和 **cx_Freeze** 上测试通过)

需要在 main 模块的 `if __name__ == '__main__'` 该行之后马上调用该函数。例如:

```
from multiprocessing import Process, freeze_support

def f():
    print('hello world!')

if __name__ == '__main__':
    freeze_support()
    Process(target=f).start()
```

如果没有调用 `freeze_support()` 在尝试运行被冻结的可执行文件时会抛出*RuntimeError* 异常。

对 `freeze_support()` 的调用在非 Windows 平台上是无效的。如果该模块在 Windows 平台的 Python 解释器中正常运行 (该程序没有被冻结)，调用 “freeze_support()“ 也是无效的。

multiprocessing.**get_all_start_methods**()
返回支持的启动方法的列表，该列表的首项即为默认选项。可能的启动方法有 `'fork'` ，`'spawn'` 和 “'forkserver'“ 。在 Windows 中，只有 `'spawn'` 是可用的。Unix 平台总是支持 `'fork'` 和 `'spawn'` ，且 `'fork'` 是默认值。

3.4 版新加入.

multiprocessing.**get_context**(*method=None*)
返回一个 Context 对象。该对象具有和*multiprocessing* 模块相同的 API。

如果 *method* 设置成 None 那么将返回默认上下文对象。否则 *method* 应该是 `'fork'`, `'spawn'`, `'forkserver'` 。如果指定的启动方法不存在，将抛出*ValueError* 异常。

3.4 版新加入.

multiprocessing.**get_start_method**(*allow_none=False*)
返回启动进程时使用的启动方法名。

如果启动方法已经固定，并且 *allow_none* 被设置成 False ，那么启动方法将被固定为默认的启动方法，并且返回其方法名。如果启动方法没有设定，并且 *allow_none* 被设置成 True ，那么将返回 `None` 。

The return value can be `'fork'`, `'spawn'`, `'forkserver'` or `None`. `'fork'` is the default on Unix, while `'spawn'` is the default on Windows and macOS.

3.8 版更變: 对于 macOS，*spawn* 启动方式是默认方式。因为 *fork* 可能导致 subprocess 崩溃，被认为是不安全的，查看 bpo-33725 。

3.4 版新加入.

multiprocessing.**set_executable**(*executable*)
Set the path of the Python interpreter to use when starting a child process. (By default *sys.executable* is used). Embedders will probably need to do some thing like

```
set_executable(os.path.join(sys.exec_prefix, 'pythonw.exe'))
```

以使他们可以创建子进程。

3.4 版更變: 现在在 Unix 平台上使用 `'spawn'` 启动方法时支持调用该方法。

multiprocessing.**set_start_method**(*method*)
设置启动子进程的方法。*method* 可以是 `'fork'` , `'spawn'` 或者 `'forkserver'` 。

注意这最多只能调用一次，并且需要藏在 main 模块中，由 `if __name__ == '__main__'` 进行保护。

3.4 版新加入.

備🄵: *multiprocessing* 并没有包含类似*threading.active_count()* , *threading.enumerate()* , *threading.settrace()* , *threading.setprofile()*, *threading.Timer* , 或者*threading.local* 的方法和类。

连接（Connection）对象

Connection 对象允许收发可以序列化的对象或字符串。它们可以看作面向消息的连接套接字。

通常使用*Pipe* 创建 Connection 对象。详见：*监听器及客户端*.

class multiprocessing.connection.**Connection**

send(*obj*)
将一个对象发送到连接的另一端，可以用*recv()* 读取。

发送的对象必须是可以序列化的，过大的对象 (接近 32MiB+ ，这个值取决于操作系统) 有可能引发*ValueError* 异常。

recv()
返回一个由另一端使用*send()* 发送的对象。该方法会一直阻塞直到接收到对象。如果对端关闭了连接或者没有东西可接收，将抛出*EOFError* 异常。

fileno()
返回由连接对象使用的描述符或者句柄。

close()
关闭连接对象。

当连接对象被垃圾回收时会自动调用。

poll([*timeout*])
返回连接对象中是否有可以读取的数据。

如果未指定 *timeout*，此方法会马上返回。如果 *timeout* 是一个数字，则指定了最大阻塞的秒数。如果 *timeout* 是 None ，那么将一直等待，不会超时。

注意通过使用*multiprocessing.connection.wait()* 可以一次轮询多个连接对象。

send_bytes(*buffer*[, *offset*[, *size*]])
从一个*bytes-like object* （字节类对象）对象中取出字节数组并作为一条完整消息发送。

如果由 *offset* 给定了在 *buffer* 中读取数据的位置。如果给定了 *size*，那么将会从缓冲区中读取多个字节。过大的缓冲区 (接近 32MiB+，此值依赖于操作系统) 有可能引发*ValueError* 异常。

recv_bytes([*maxlength*])
以字符串形式返回一条从连接对象另一端发送过来的字节数据。此方法在接收到数据前将一直阻塞。如果连接对象被对端关闭或者没有数据可读取，将抛出*EOFError* 异常。

如果给定了 *maxlength* 并且消息长于 *maxlength* 那么将抛出*OSError* 并且该连接对象将不再可读。

3.3 版更變: 曾经该函数抛出*IOError*，现在这是*OSError* 的别名。

recv_bytes_into(*buffer*[, *offset*])
将一条完整的字节数据消息读入 *buffer* 中并返回消息的字节数。此方法在接收到数据前将一直阻塞。如果连接对象被对端关闭或者没有数据可读取，将抛出*EOFError* 异常。

buffer must be a writable *bytes-like object*. If *offset* is given then the message will be written into the buffer from that position. Offset must be a non-negative integer less than the length of *buffer* (in bytes).

如果缓冲区太小，则将引发 BufferTooShort 异常，并且完整的消息将会存放在异常实例 e 的 e.args[0] 中。

3.3 版更變: 现在连接对象自身可以通过*Connection.send()* 和*Connection.recv()* 在进程之间传递。

3.3 版新加入: 连接对象现已支持上下文管理协议 -- 参见 see 上下文管理器类型 。*__enter__()* 返回连接对象，*__exit__()* 会调用*close()* 。

例如:

```
>>> from multiprocessing import Pipe
>>> a, b = Pipe()
>>> a.send([1, 'hello', None])
>>> b.recv()
[1, 'hello', None]
>>> b.send_bytes(b'thank you')
>>> a.recv_bytes()
b'thank you'
>>> import array
>>> arr1 = array.array('i', range(5))
>>> arr2 = array.array('i', [0] * 10)
>>> a.send_bytes(arr1)
>>> count = b.recv_bytes_into(arr2)
>>> assert count == len(arr1) * arr1.itemsize
>>> arr2
array('i', [0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

警告： *Connection.recv()* 方法会自动解封它收到的数据，除非你能够信任发送消息的进程，否则此处可能有安全风险。

因此，除非连接对象是由 Pipe() 产生的，否则你应该仅在使用了某种认证手段之后才使用*recv()* 和*send()* 方法。参考*认证密码*。

警告： 如果一个进程在试图读写管道时被终止了，那么管道中的数据很可能是不完整的，因为此时可能无法确定消息的边界。

同步原语

通常来说同步原语在多进程环境中并不像它们在多线程环境中那么必要。参考*threading* 模块的文档。

注意可以使用管理器对象创建同步原语，参考*管理器* 。

class multiprocessing.**Barrier**(*parties*[, *action*[, *timeout*]])
类似*threading.Barrier* 的栅栏对象。

3.3 版新加入.

class multiprocessing.**BoundedSemaphore**([*value*])
非常类似*threading.BoundedSemaphore* 的有界信号量对象。

一个小小的不同在于，它的 acquire 方法的第一个参数名是和*Lock.acquire()* 一样的 *block* 。

備🄵: On macOS, this is indistinguishable from *Semaphore* because sem_getvalue() is not implemented on that platform.

---

class multiprocessing.**Condition**([*lock*])
条件变量：*threading.Condition* 的别名。

指定的 *lock* 参数应该是*multiprocessing* 模块中的*Lock* 或者*RLock* 对象。

3.3 版更變: 新增了*wait_for()* 方法。

class multiprocessing.**Event**
A clone of *threading.Event*.

class multiprocessing.**Lock**
原始锁（非递归锁）对象，类似于*threading.Lock* 。一旦一个进程或者线程拿到了锁，后续的任何其他进程或线程的其他请求都会被阻塞直到锁被释放。任何进程或线程都可以释放锁。除非另有说明，否则*multiprocessing.Lock* 用于进程或者线程的概念和行为都和*threading.Lock* 一致。

注意*Lock* 实际上是一个工厂函数。它返回由默认上下文初始化的 multiprocessing.synchronize.Lock 对象。

Lock supports the *context manager* protocol and thus may be used in with statements.

acquire(*block=True*, *timeout=None*)
获得锁，阻塞或非阻塞的。

如果 *block* 参数被设为 True（默认值），对该方法的调用在锁处于释放状态之前都会阻塞，然后将锁设置为锁住状态并返回 True 。需要注意的是第一个参数名与*threading.Lock.acquire()* 的不同。

如果 *block* 参数被设置成 `False` ，方法的调用将不会阻塞。如果锁当前处于锁住状态，将返回 `False` ；否则将锁设置成锁住状态，并返回 `True` 。

当 *timeout* 是一个正浮点数时，会在等待锁的过程中最多阻塞等待 *timeout* 秒，当 *timeout* 是负数时，效果和 *timeout* 为 0 时一样，当 *timeout* 是 `None` （默认值）时，等待时间是无限长。需要注意的是，对于 *timeout* 参数是负数和 `None` 的情况，其行为与*threading.Lock.acquire()* 是不一样的。当 *block* 参数为 `False` 时，*timeout* 并没有实际用处，会直接忽略。否则，函数会在拿到锁后返回 `True` 或者超时没拿到锁后返回 `False` 。

release()
释放锁，可以在任何进程、线程使用，并不限于锁的拥有者。

当尝试释放一个没有被持有的锁时，会抛出*ValueError* 异常，除此之外其行为与*threading.Lock.release()* 一样。

class multiprocessing.**RLock**
递归锁对象: 类似于*threading.RLock* 。递归锁必须由持有线程、进程亲自释放。如果某个进程或者线程拿到了递归锁，这个进程或者线程可以再次拿到这个锁而不需要等待。但是这个进程或者线程的拿锁操作和释放锁操作的次数必须相同。

注意*RLock* 是一个工厂函数，调用后返回一个使用默认 context 初始化的 `multiprocessing.synchronize.RLock` 实例。

RLock 支持*context manager*，所以可在 `with` 语句内使用。

acquire(*block=True*, *timeout=None*)
获得锁，阻塞或非阻塞的。

当 *block* 参数设置为 `True` 时，会一直阻塞直到锁处于空闲状态（没有被任何进程、线程拥有），除非当前进程或线程已经拥有了这把锁。然后当前进程/线程会持有这把锁（在锁没有其他持有者的情况下），锁内的递归等级加一，并返回 `True` . 注意，这个函数第一个参数的行为和*threading.RLock.acquire()* 的实现有几个不同点，包括参数名本身。

当 *block* 参数是 `False` , 将不会阻塞，如果此时锁被其他进程或者线程持有，当前进程、线程获取锁操作失败，锁的递归等级也不会改变，函数返回 `False` , 如果当前锁已经处于释放状态，则当前进程、线程则会拿到锁，并且锁内的递归等级加一，函数返回 `True` 。

timeout 参数的使用方法及行为与*Lock.acquire()* 一样。但是要注意 *timeout* 的其中一些行为和*threading.RLock.acquire()* 中实现的行为是不同的。

release()
释放锁，使锁内的递归等级减一。如果释放后锁内的递归等级降低为 0，则会重置锁的状态为释放状态（即没有被任何进程、线程持有），重置后如果有有其他进程和线程在等待这把锁，他们中的一个会获得这个锁而继续运行。如果释放后锁内的递归等级还没到达 0，则这个锁仍将保持未释放状态且当前进程和线程仍然是持有者。

只有当前进程或线程是锁的持有者时，才允许调用这个方法。如果当前进程或线程不是这个锁的拥有者，或者这个锁处于已释放的状态(即没有任何拥有者)，调用这个方法会抛出*AssertionError* 异常。注意这里抛出的异常类型和*threading.RLock.release()* 中实现的行为不一样。

class multiprocessing.**Semaphore**([*value*])
一种信号量对象: 类似于*threading.Semaphore*.

一个小小的不同在于，它的 `acquire` 方法的第一个参数名是和*Lock.acquire()* 一样的 *block* 。

備F: On macOS, `sem_timedwait` is unsupported, so calling `acquire()` with a timeout will emulate that function's behavior using a sleeping loop.

備𝔽: 假如信号 SIGINT 是来自于 Ctrl-C，并且主线程被 BoundedSemaphore.acquire(), *Lock.acquire()*, *RLock.acquire()*, Semaphore.acquire(), Condition.acquire() 或 Condition.wait() 阻塞，则调用会立即中断同时抛出*KeyboardInterrupt* 异常。

这和*threading* 的行为不同，此模块中当执行对应的阻塞调用时，SIGINT 会被忽略。

備𝔽: 这个包的某些功能依赖于宿主机系统的共享信号量的实现，如果系统没有这个特性，multiprocessing.synchronize 会被禁用，尝试导入这个模块会引发*ImportError* 异常，详细信息请查看 bpo-3770 。

共享 ctypes 对象

在共享内存上创建可被子进程继承的共享对象时是可行的。

multiprocessing.**Value**(*typecode_or_type*, **args*, *lock=True*)
返回一个从共享内存上创建的*ctypes* 对象。默认情况下返回的对象实际上是经过了同步器包装过的。可以通过*Value* 的 *value* 属性访问这个对象本身。

typecode_or_type 指明了返回的对象类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。**args* 会透传给这个类的构造函数。

如果 *lock* 参数是 True （默认值），将会新建一个递归锁用于同步对于此值的访问操作。如果 *lock* 是*Lock* 或者*RLock* 对象，那么这个传入的锁将会用于同步对这个值的访问操作，如果 *lock* 是 False，那么对这个对象的访问将没有锁保护，也就是说这个变量不是进程安全的。

诸如 += 这类的操作会引发独立的读操作和写操作，也就是说这类操作符并不具有原子性。所以，如果你想让递增共享变量的操作具有原子性，仅仅以这样的方式并不能达到要求:

```
counter.value += 1
```

共享对象内部关联的锁是递归锁 (默认情况下就是) 的情况下，你可以采用这种方式

```
with counter.get_lock():
    counter.value += 1
```

注意 *lock* 只能是命名参数。

multiprocessing.**Array**(*typecode_or_type*, *size_or_initializer*, ***, *lock=True*)
从共享内存中申请并返回一个具有 ctypes 类型的数组对象。默认情况下返回值实际上是被同步器包装过的数组对象。

typecode_or_type 指明了返回的数组中的元素类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。如果 *size_or_initializer* 是一个整数，那就会当做数组的长度，并且整个数组的内存会初始化为 0。否则，如果 *size_or_initializer* 会被当成一个序列用于初始化数组中的每一个元素，并且会根据元素个数自动判断数组的长度。

如果 *lock* 为 True (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个*Lock* 或*RLock* 对象则该对象将被用于同步对值的访问。如果 *lock* 为 False 则对返回对象的访问将不会自动得到锁的保护，也就是说它不是“进程安全的”。

请注意 *lock* 是一个仅限关键字参数。

请注意*ctypes.c_char* 的数组具有 *value* 和 *raw* 属性，允许被用来保存和提取字符串。

multiprocessing.sharedctypes 模块

multiprocessing.sharedctypes 模块提供了一些函数，用于分配来自共享内存的、可被子进程继承的*ctypes* 对象。

備F: 虽然可以将指针存储在共享内存中，但请记住它所引用的是特定进程地址空间中的位置。而且，指针很可能在第二个进程的上下文中无效，尝试从第二个进程对指针进行解引用可能会导致崩溃。

---

multiprocessing.sharedctypes.**RawArray**(*typecode_or_type*, *size_or_initializer*)
从共享内存中申请并返回一个 ctypes 数组。

typecode_or_type 指明了返回的数组中的元素类型: 它可能是一个 ctypes 类型或者*array* 模块中使用的类型字符。如果 *size_or_initializer* 是一个整数，那就会当做数组的长度，并且整个数组的内存会初始化为 0。否则，如果 *size_or_initializer* 会被当成一个序列用于初始化数组中的每一个元素，并且会根据元素个数自动判断数组的长度。

注意对元素的访问、赋值操作可能是非原子操作 - 使用*Array()* , 从而借助其中的锁保证操作的原子性。

multiprocessing.sharedctypes.**RawValue**(*typecode_or_type*, **args*)
从共享内存中申请并返回一个 ctypes 对象。

typecode_or_type 指明了返回的对象类型: 它可能是一个 ctypes 类型或者*array* 模块中每个类型对应的单字符长度的字符串。**args* 会透传给这个类的构造函数。

注意对 value 的访问、赋值操作可能是非原子操作 - 使用*Value()* ，从而借助其中的锁保证操作的原子性。

请注意*ctypes.c_char* 的数组具有 *value* 和 *raw* 属性，允许被用来保存和提取字符串 - 请查看*ctypes* 文档。

multiprocessing.sharedctypes.**Array**(*typecode_or_type*, *size_or_initializer*, *, *lock=True*)
返回一个纯 ctypes 数组, 或者在此之上经过同步器包装过的进程安全的对象，这取决于 *lock* 参数的值，除此之外，和*RawArray()* 一样。

如果 *lock* 为 `True` (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个 `Lock` 或 `RLock` 对象则该对象将被用于同步对值的访问。如果 *lock* 为 `False` 则对返回对象的访问将不会自动得到锁的保护，也就是说它不是“进程安全的”。

注意 *lock* 只能是命名参数。

multiprocessing.sharedctypes.**Value**(*typecode_or_type*, **args*, *lock=True*)
返回一个纯 ctypes 数组, 或者在此之上经过同步器包装过的进程安全的对象，这取决于 *lock* 参数的值，除此之外，和 `RawArray()` 一样。

如果 *lock* 为 `True` (默认值) 则将创建一个新的锁对象用于同步对值的访问。如果 *lock* 为一个 `Lock` 或 `RLock` 对象则该对象将被用于同步对值的访问。如果 *lock* 为 `False` 则对返回对象的访问将不会自动得到锁的保护，也就是说它不是“进程安全的”。

注意 *lock* 只能是命名参数。

multiprocessing.sharedctypes.**copy**(*obj*)
从共享内存中申请一片空间将 ctypes 对象 *obj* 过来，然后返回一个新的 ctypes 对象。

multiprocessing.sharedctypes.**synchronized**(*obj*[, *lock*])
将一个 ctypes 对象包装为进程安全的对象并返回，使用 *lock* 同步对于它的操作。如果 *lock* 是 `None` (默认值)，则会自动创建一个*multiprocessing.RLock* 对象。

同步器包装后的对象会在原有对象基础上额外增加两个方法: `get_obj()` 返回被包装的对象，`get_lock()` 返回内部用于同步的锁。

需要注意的是，访问包装后的 ctypes 对象会比直接访问原来的纯 ctypes 对象慢得多。

3.5 版更變: 同步器包装后的对象支持*context manager* 协议。

下面的表格对比了创建普通 ctypes 对象和基于共享内存上创建共享 ctypes 对象的语法。(表格中的 `MyStruct` 是*`ctypes.Structure`* 的子类)

ctypes	使用类型的共享 ctypes	使用 typecode 的共享 ctypes
c_double(2.4)	RawValue(c_double, 2.4)	RawValue('d', 2.4)
MyStruct(4, 6)	RawValue(MyStruct, 4, 6)	
(c_short * 7)()	RawArray(c_short, 7)	RawArray('h', 7)
(c_int * 3)(9, 2, 8)	RawArray(c_int, (9, 2, 8))	RawArray('i', (9, 2, 8))

下面是一个在子进程中修改多个 ctypes 对象的例子。

```
from multiprocessing import Process, Lock
from multiprocessing.sharedctypes import Value, Array
from ctypes import Structure, c_double

class Point(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

def modify(n, x, s, A):
    n.value **= 2
    x.value **= 2
    s.value = s.value.upper()
    for a in A:
        a.x **= 2
        a.y **= 2

if __name__ == '__main__':
    lock = Lock()

    n = Value('i', 7)
    x = Value(c_double, 1.0/3.0, lock=False)
    s = Array('c', b'hello world', lock=lock)
    A = Array(Point, [(1.875,-6.25), (-5.75,2.0), (2.375,9.5)], lock=lock)

    p = Process(target=modify, args=(n, x, s, A))
    p.start()
    p.join()

    print(n.value)
    print(x.value)
    print(s.value)
    print([(a.x, a.y) for a in A])
```

输出如下

```
49
0.1111111111111111
HELLO WORLD
[(3.515625, 39.0625), (33.0625, 4.0), (5.640625, 90.25)]
```

管理器

管理器提供了一种创建共享数据的方法，从而可以在不同进程中共享，甚至可以通过网络跨机器共享数据。管理器维护一个用于管理 共享对象的服务。其他进程可以通过代理访问这些共享对象。

multiprocessing.**Manager**()
返回一个已启动的*SyncManager* 管理器对象，这个对象可以用于在不同进程中共享数据。返回的管理器对象对应了一个已经启动的子进程，并且拥有一系列方法可以用于创建共享对象、返回对应的代理。

当管理器被垃圾回收或者父进程退出时，管理器进程会立即退出。管理器类定义在*multiprocessing.managers* 模块:

class multiprocessing.managers.**BaseManager**([*address*[, *authkey*]])
创建一个 BaseManager 对象。

一旦创建，应该及时调用*start()* 或者 get_server().serve_forever() 以确保管理器对象对应的管理进程已经启动。

address 是管理器服务进程监听的地址。如果 *address* 是 None，则允许和任意主机的请求建立连接。

authkey 是认证标识，用于检查连接服务进程的请求合法性。如果 *authkey* 是 None，则会使用 current_process().authkey，否则，就使用 *authkey*，需要保证它必须是 byte 类型的字符串。

start([*initializer*[, *initargs*]])
为管理器开启一个子进程，如果 *initializer* 不是 None，子进程在启动时将会调用 initializer(*initargs)。

get_server()
返回一个 Server 对象，它是管理器在后台控制的真实的服务。Server 对象拥有 serve_forever() 方法。

```
>>> from multiprocessing.managers import BaseManager
>>> manager = BaseManager(address=('', 50000), authkey=b'abc')
>>> server = manager.get_server()
>>> server.serve_forever()
```

Server 额外拥有一个*address* 属性。

connect()
将本地管理器对象连接到一个远程管理器进程:

```
>>> from multiprocessing.managers import BaseManager
>>> m = BaseManager(address=('127.0.0.1', 50000), authkey=b'abc')
>>> m.connect()
```

shutdown()
停止管理器的进程。这个方法只能用于已经使用*start()* 启动的服务进程。

它可以被多次调用。

register(*typeid*[, *callable*[, *proxytype*[, *exposed*[, *method_to_typeid*[, *create_method*]]]]])
一个 classmethod，可以将一个类型或者可调用对象注册到管理器类。

typeid 是一种" 类型标识符"，用于唯一表示某种共享对象类型，必须是一个字符串。

callable 是一个用来为此类型标识符创建对象的可调用对象。如果一个管理器实例将使用*connect()* 方法连接到服务器，或者 *create_method* 参数为 False，那么这里可留下 None。

proxytype 是*BaseProxy* 的子类，可以根据 *typeid* 为共享对象创建一个代理，如果是 None，则会自动创建一个代理类。

exposed 是一个函数名组成的序列，用来指明只有这些方法可以使用*BaseProxy._callmethod()* 代理。(如果 *exposed* 是 None, 则会在 proxytype._exposed_ 存在的情况下转而使用它) 当暴露的方法列表没有指定的时候，共享对象的所有“公共方法”都会被代理。(这里的“公共方法”是指所有拥有 __call__() 方法并且不是以 '_' 开头的属性)

method_to_typeid 是一个映射，用来指定那些应该返回代理对象的暴露方法所返回的类型。(如果 *method_to_typeid* 是 None, 则 proxytype._method_to_typeid_ 会在存在的情况下被使用) 如果方法名称不在这个映射中或者映射是 None , 则方法返回的对象会是一个值拷贝。

create_method 指明，是否要创建一个以 *typeid* 命名并返回一个代理对象的方法，这个函数会被服务进程用于创建共享对象，默认为 True 。

BaseManager 实例也有一个只读属性。

address
管理器所用的地址。

3.3 版更變: 管理器对象支持上下文管理协议 - 查看上下文管理器类型 。*__enter__()* 启动服务进程（如果它还没有启动）并且返回管理器对象，*__exit__()* 会调用*shutdown()* 。

在之前的版本中，如果管理器服务进程没有启动，*__enter__()* 不会负责启动它。

class multiprocessing.managers.**SyncManager**
BaseManager 的子类，可用于进程的同步。这个类型的对象使用*multiprocessing.Manager()* 创建。

它拥有一系列方法，可以为大部分常用数据类型创建并返回代理对象 代理，用于进程间同步。甚至包括共享列表和字典。

Barrier(*parties*[, *action*[, *timeout*]])
创建一个共享的*threading.Barrier* 对象并返回它的代理。

3.3 版新加入.

BoundedSemaphore([*value*])
创建一个共享的*threading.BoundedSemaphore* 对象并返回它的代理。

Condition([*lock*])
创建一个共享的*threading.Condition* 对象并返回它的代理。

如果提供了 *lock* 参数，那它必须是*threading.Lock* 或*threading.RLock* 的代理对象。

3.3 版更變: 新增了*wait_for()* 方法。

Event()
创建一个共享的*threading.Event* 对象并返回它的代理。

Lock()
创建一个共享的*threading.Lock* 对象并返回它的代理。

Namespace()
创建一个共享的*Namespace* 对象并返回它的代理。

Queue([*maxsize*])
创建一个共享的*queue.Queue* 对象并返回它的代理。

RLock()
创建一个共享的*threading.RLock* 对象并返回它的代理。

Semaphore([*value*])
创建一个共享的*threading.Semaphore* 对象并返回它的代理。

Array(*typecode*, *sequence*)
创建一个数组并返回它的代理。

Value(*typecode*, *value*)
创建一个具有可写 value 属性的对象并返回它的代理。

dict()
dict(*mapping*)
dict(*sequence*)
创建一个共享的*dict* 对象并返回它的代理。

list()
list(*sequence*)
创建一个共享的*list* 对象并返回它的代理。

3.6 版更變: 共享对象能够嵌套。例如, 共享的容器对象如共享列表，可以包含另一个共享对象，他们全都会在*SyncManager* 中进行管理和同步。

class multiprocessing.managers.**Namespace**
一个可以注册到*SyncManager* 的类型。

命名空间对象没有公共方法，但是拥有可写的属性。直接 print 会显示所有属性的值。

值得一提的是，当对命名空间对象使用代理的时候，访问所有名称以 '_' 开头的属性都只是代理器上的属性，而不是命名空间对象的属性。

```
>>> manager = multiprocessing.Manager()
>>> Global = manager.Namespace()
>>> Global.x = 10
>>> Global.y = 'hello'
>>> Global._z = 12.3    # this is an attribute of the proxy
>>> print(Global)
Namespace(x=10, y='hello')
```

自定义管理器

要创建一个自定义的管理器，需要新建一个*BaseManager* 的子类，然后使用这个管理器类上的*register()* 类方法将新类型或者可调用方法注册上去。例如:

```
from multiprocessing.managers import BaseManager

class MathsClass:
    def add(self, x, y):
        return x + y
    def mul(self, x, y):
        return x * y

class MyManager(BaseManager):
    pass

MyManager.register('Maths', MathsClass)

if __name__ == '__main__':
    with MyManager() as manager:
        maths = manager.Maths()
        print(maths.add(4, 3))         # prints 7
        print(maths.mul(7, 8))         # prints 56
```

使用远程管理器

可以将管理器服务运行在一台机器上，然后使用客户端从其他机器上访问。(假设它们的防火墙允许)
运行下面的代码可以启动一个服务，此付包含了一个共享队列，允许远程客户端访问:

```
>>> from multiprocessing.managers import BaseManager
>>> from queue import Queue
>>> queue = Queue()
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue', callable=lambda:queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

远程客户端可以通过下面的方式访问服务:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.put('hello')
```

也可以通过下面的方式:

```
>>> from multiprocessing.managers import BaseManager
>>> class QueueManager(BaseManager): pass
>>> QueueManager.register('get_queue')
>>> m = QueueManager(address=('foo.bar.org', 50000), authkey=b'abracadabra')
>>> m.connect()
>>> queue = m.get_queue()
>>> queue.get()
'hello'
```

本地进程也可以访问这个队列，利用上面的客户端代码通过远程方式访问:

```
>>> from multiprocessing import Process, Queue
>>> from multiprocessing.managers import BaseManager
>>> class Worker(Process):
...     def __init__(self, q):
...         self.q = q
...         super().__init__()
...     def run(self):
...         self.q.put('local hello')
...
>>> queue = Queue()
>>> w = Worker(queue)
>>> w.start()
>>> class QueueManager(BaseManager): pass
...
>>> QueueManager.register('get_queue', callable=lambda: queue)
>>> m = QueueManager(address=('', 50000), authkey=b'abracadabra')
>>> s = m.get_server()
>>> s.serve_forever()
```

代理对象

代理是一个 指向其他共享对象的对象，这个对象 (很可能) 在另外一个进程中。共享对象也可以说是代理 指涉的对象。多个代理对象可能指向同一个指涉对象。

代理对象代理了指涉对象的一系列方法调用 (虽然并不是指涉对象的每个方法都有必要被代理)。通过这种方式，代理的使用方法可以和它的指涉对象一样:

```
>>> from multiprocessing import Manager
>>> manager = Manager()
>>> l = manager.list([i*i for i in range(10)])
>>> print(l)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> print(repr(l))
<ListProxy object, typeid 'list' at 0x...>
>>> l[4]
16
>>> l[2:5]
[4, 9, 16]
```

注意，对代理使用*str()* 函数会返回指涉对象的字符串表示，但是*repr()* 却会返回代理本身的内部字符串表示。

被代理的对象很重要的一点是必须可以被序列化，这样才能允许他们在进程间传递。因此，指涉对象可以包含*代理对象* 。这允许管理器中列表、字典或者其他*代理对象* 对象之间的嵌套。

```
>>> a = manager.list()
>>> b = manager.list()
>>> a.append(b)         # referent of a now contains referent of b
>>> print(a, b)
[<ListProxy object, typeid 'list' at ...>] []
>>> b.append('hello')
>>> print(a[0], b)
['hello'] ['hello']
```

类似地，字典和列表代理也可以相互嵌套:

```
>>> l_outer = manager.list([ manager.dict() for i in range(2) ])
>>> d_first_inner = l_outer[0]
>>> d_first_inner['a'] = 1
>>> d_first_inner['b'] = 2
>>> l_outer[1]['c'] = 3
>>> l_outer[1]['z'] = 26
>>> print(l_outer[0])
{'a': 1, 'b': 2}
>>> print(l_outer[1])
{'c': 3, 'z': 26}
```

如果指涉对象包含了普通*list* 或*dict* 对象，对这些内部可变对象的修改不会通过管理器传播，因为代理无法得知被包含的值什么时候被修改了。但是把存放在容器代理中的值本身是会通过管理器传播的（会触发代理对象中的 __setitem__ ）从而有效修改这些对象，所以可以把修改过的值重新赋值给容器代理:

```
# create a list proxy and append a mutable object (a dictionary)
lproxy = manager.list()
lproxy.append({})
# now mutate the dictionary
d = lproxy[0]
d['a'] = 1
```

(下页继续)

(繼續上一頁)

```
d['b'] = 2
# at this point, the changes to d are not yet synced, but by
# updating the dictionary, the proxy is notified of the change
lproxy[0] = d
```

在大多是使用情形下，这种实现方式并不比嵌套*代理对象* 方便，但是依然演示了对于同步的一种控制级别。

備𝔽: *multiprocessing* 中的代理类并没有提供任何对于代理值比较的支持。所以，我们会得到如下结果:

```
>>> manager.list([1,2,3]) == [1,2,3]
False
```

当需要比较值的时候，应该替换为使用指涉对象的拷贝。

---

class multiprocessing.managers.**BaseProxy**

代理对象是*BaseProxy* 派生类的实例。

_callmethod(*methodname*[, *args*[, *kwds*]])

调用指涉对象的方法并返回结果。

如果 `proxy` 是一个代理且其指涉的是 `obj`，那么下面的表达式:

```
proxy._callmethod(methodname, args, kwds)
```

相当于求取以下表达式的值:

```
getattr(obj, methodname)(*args, **kwds)
```

于管理器进程。

返回结果会是一个值拷贝或者一个新的共享对象的代理 - 见函数*BaseManager.register()* 中关于参数 *method_to_typeid* 的文档。

如果这个调用熬出了异常，则这个异常会被*_callmethod()* 透传出来。如果是管理器进程本身抛出的一些其他异常，则会被*_callmethod()* 转换为 `RemoteError` 异常重新抛出。

特别注意，如果 *methodname* 没有 暴露出来，将会引发一个异常。

_callmethod() 的一个使用示例:

```
>>> l = manager.list(range(10))
>>> l._callmethod('__len__')
10
>>> l._callmethod('__getitem__', (slice(2, 7),)) # equivalent to l[2:7]
[2, 3, 4, 5, 6]
>>> l._callmethod('__getitem__', (20,))          # equivalent to l[20]
Traceback (most recent call last):
...
IndexError: list index out of range
```

_getvalue()

返回指涉对象的一份拷贝。

如果指涉对象无法序列化，则会抛出一个异常。

__repr__()

返回代理对象的内部字符串表示。

__str__()
返回指涉对象的内部字符串表示。

清理

代理对象使用了一个弱引用回调函数，当它被垃圾回收时，会将自己从拥有此指涉对象的管理器上反注册，当共享对象没有被任何代理器引用时，会被管理器进程删除。

进程池

可以创建一个进程池，它将使用*Pool* 类执行提交给它的任务。

class multiprocessing.pool.**Pool**([*processes*[, *initializer*[, *initargs*[, *maxtasksperchild*[, *context*]]]])

一个进程池对象，它控制可以提交作业的工作进程池。它支持带有超时和回调的异步结果，以及一个并行的 map 实现。

processes 是要使用的工作进程数目。如果 *processes* 为 None，则使用*os.cpu_count()* 返回的值。

如果 *initializer* 不为 None，则每个工作进程将会在启动时调用 initializer(*initargs)。

maxtasksperchild 是一个工作进程在它退出或被一个新的工作进程代替之前能完成的任务数量，为了释放未使用的资源。默认的 *maxtasksperchild* 是 None，意味着工作进程寿与池齐。

context 可被用于指定启动的工作进程的上下文。通常一个进程池是使用函数 multiprocessing.Pool() 或者一个上下文对象的*Pool()* 方法创建的。在这两种情况下，*context* 都是适当设置的。

注意，进程池对象的方法只有创建它的进程能够调用。

> **警告：** *multiprocessing.pool* 对象具有需要正确管理的内部资源（像任何其他资源一样），具体方式是将进程池用作上下文管理器，或者手动调用*close()* 和*terminate()*。未做此类操作将导致进程在终结阶段挂起。
>
> 请注意依赖垃圾回收器来销毁进程池是 **不正确**的做法，因为 CPython 并不保证会调用进程池终结程序（请参阅 object.__del__() 来了解详情）。

3.2 版新加入: *maxtasksperchild*

3.4 版新加入: *context*

備F: 通常来说，*Pool* 中的 Worker 进程的生命周期和进程池的工作队列一样长。一些其他系统中（如 Apache, mod_wsgi 等）也可以发现另一种模式，他们会让工作进程在完成一些任务后退出，清理、释放资源，然后启动一个新的进程代替旧的工作进程。*Pool* 的 *maxtasksperchild* 参数给用户提供了这种能力。

---

apply(*func*[, *args*[, *kwds*]])
使用 *args* 参数以及 *kwds* 命名参数调用 *func*，它会返回结果前阻塞。这种情况下，*apply_async()* 更适合并行化工作。另外 *func* 只会在一个进程池中的一个工作进程中执行。

apply_async(*func*[, *args*[, *kwds*[, *callback*[, *error_callback*]]]])
apply() 方法的一个变种，返回一个*AsyncResult* 对象。

如果指定了 *callback*，它必须是一个接受单个参数的可调用对象。当执行成功时，*callback* 会被用于处理执行后的返回结果，否则，调用 *error_callback*。

如果指定了 *error_callback*，它必须是一个接受单个参数的可调用对象。当目标函数执行失败时，会将抛出的异常对象作为参数传递给 *error_callback* 执行。

回调函数应该立即执行完成，否则会阻塞负责处理结果的线程。

map(*func*, *iterable*[, *chunksize*])
内置`map()`函数的并行版本（但它只支持一个 *iterable* 参数，对于多个可迭代对象请参阅`starmap()`）。它会保持阻塞直到获得结果。

这个方法会将可迭代对象分割为许多块，然后提交给进程池。可以将 *chunksize* 设置为一个正整数从而（近似）指定每个块的大小可以。

注意对于很长的迭代对象，可能消耗很多内存。可以考虑使用`imap()`或`imap_unordered()`并且显示指定 *chunksize* 以提升效率。

map_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])
`map()`方法的一个变种，返回一个`AsyncResult`对象。

如果指定了 *callback*，它必须是一个接受单个参数的可调用对象。当执行成功时，*callback* 会被用于处理执行后的返回结果，否则，调用 *error_callback*。

如果指定了 *error_callback*，它必须是一个接受单个参数的可调用对象。当目标函数执行失败时，会将抛出的异常对象作为参数传递给 *error_callback* 执行。

回调函数应该立即执行完成，否则会阻塞负责处理结果的线程。

imap(*func*, *iterable*[, *chunksize*])
`map()`的延迟执行版本。

chunksize 参数的作用和`map()`方法的一样。对于很长的迭代器，给 *chunksize* 设置一个很大的值会比默认值 `1` **极大**地加快执行速度。

同样，如果 *chunksize* 是 `1`，那么`imap()`方法所返回的迭代器的 `next()` 方法拥有一个可选的 *timeout* 参数：如果无法在 *timeout* 秒内执行得到结果，则 “next(timeout)“ 会抛出`multiprocessing.TimeoutError`异常。

imap_unordered(*func*, *iterable*[, *chunksize*])
和`imap()`相同，只不过通过迭代器返回的结果是任意的。（当进程池中只有一个工作进程的时候，返回结果的顺序才能认为是” 有序” 的）

starmap(*func*, *iterable*[, *chunksize*])
Like `map()` except that the elements of the *iterable* are expected to be iterables that are unpacked as arguments.

比如可迭代对象 `[(1,2), (3, 4)]` 会转化为等价于 `[func(1,2), func(3,4)]` 的调用。

3.3 版新加入.

starmap_async(*func*, *iterable*[, *chunksize*[, *callback*[, *error_callback*]]])
相当于`starmap()`与`map_async()`的结合，迭代 *iterable* 的每一项，解包作为 *func* 的参数并执行，返回用于获取结果的对象。

3.3 版新加入.

close()
阻止后续任务提交到进程池，当所有任务执行完成后，工作进程会退出。

terminate()
不必等待未完成的任务，立即停止工作进程。当进程池对象被垃圾回收时，会立即调用`terminate()`。

join()
等待工作进程结束。调用`join()`前必须先调用`close()`或者`terminate()`。

3.3 版新加入: 进程池对象现在支持上下文管理器协议 - 参见上下文管理器类型 。*__enter__()* 返回进程池对象, *__exit__()* 会调用*terminate()* 。

class multiprocessing.pool.**AsyncResult**

Pool.apply_async() 和*Pool.map_async()* 返回对象所属的类。

get([*timeout*])

用于获取执行结果。如果 *timeout* 不是 None 并且在 *timeout* 秒内仍然没有执行完得到结果，则抛出*multiprocessing.TimeoutError* 异常。如果远程调用发生异常，这个异常会通过*get()* 重新抛出。

wait([*timeout*])

阻塞，直到返回结果，或者 *timeout* 秒后超时。

ready()

返回执行状态，是否已经完成。

successful()

判断调用是否已经完成并且未引发异常。如果还未获得结果则将引发*ValueError*。

3.7 版更變: 如果没有执行完，会抛出*ValueError* 异常而不是*AssertionError* 。

下面的例子演示了进程池的用法:

```
from multiprocessing import Pool
import time

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(processes=4) as pool:          # start 4 worker processes
        result = pool.apply_async(f, (10,)) # evaluate "f(10)" asynchronously in a␣
↪single process
        print(result.get(timeout=1))         # prints "100" unless your computer is␣
↪*very* slow

        print(pool.map(f, range(10)))        # prints "[0, 1, 4,..., 81]"

        it = pool.imap(f, range(10))
        print(next(it))                      # prints "0"
        print(next(it))                      # prints "1"
        print(it.next(timeout=1))            # prints "4" unless your computer is␣
↪*very* slow

        result = pool.apply_async(time.sleep, (10,))
        print(result.get(timeout=1))         # raises multiprocessing.TimeoutError
```

监听器及客户端

通常情况下，进程间通过队列或者*Pipe()* 返回的*Connection* 传递消息。

不过，*multiprocessing.connection* 模块其实提供了一些更灵活的特性。最基础的用法是通过它抽象出来的高级 API 来操作 socket 或者 Windows 命名管道。也提供一些高级用法，如通过*hmac* 模块来支持 摘要认证，以及同时监听多个管道连接。

multiprocessing.connection.**deliver_challenge**(*connection*, *authkey*)

发送一个随机生成的消息到另一端，并等待回复。

如果收到的回复与使用 *authkey* 作为键生成的信息摘要匹配成功，就会发送一个欢迎信息给管道另一端。否则抛出*AuthenticationError* 异常。

multiprocessing.connection.**answer_challenge**(*connection*, *authkey*)
接收一条信息，使用 *authkey* 作为键计算信息摘要，然后将摘要发送回去。

如果没有收到欢迎消息，就抛出*AuthenticationError* 异常。

multiprocessing.connection.**Client**(*address*[, *family*[, *authkey*]])
尝试使用 *address* 地址上的监听器建立一个连接，返回*Connection* 。

连接的类型取决于 *family* 参数，但是通常可以省略，因为可以通过 *address* 的格式推导出来。(查看地址格式)

如果提供了 *authkey* 参数并且不是 None，那它必须是一个字符串并且会被当做基于 HMAC 认证的密钥。如果 *authkey* 是 None 则不会有认证行为。认证失败抛出*AuthenticationError* 异常，请查看 See 认证密码 。

class multiprocessing.connection.**Listener**([*address*[, *family*[, *backlog*[, *authkey*]]]])
可以监听连接请求，是对于绑定套接字或者 Windows 命名管道的封装。

address 是监听器对象中的绑定套接字或命名管道使用的地址。

備🄵: 如果使用'0.0.0.0' 作为监听地址，那么在 Windows 上这个地址无法建立连接。想要建立一个可连接的端点，应该使用'127.0.0.1' 。

---

family 是套接字 (或者命名管道) 使用的类型。它可以是以下一种: `'AF_INET'` (TCP 套接字类型), `'AF_UNIX'` (Unix 域套接字) 或者 `'AF_PIPE'` (Windows 命名管道)。其中只有第一个保证各平台可用。如果 *family* 是 `None` , 那么 family 会根据 *address* 的格式自动推导出来。如果 *address* 也是 `None` , 则取默认值。默认值为可用类型中速度最快的。见地址格式 。注意，如果 *family* 是 `'AF_UNIX'` 而 address 是 "None" , 套接字会在一个*tempfile.mkstemp()* 创建的私有临时目录中创建。

如果监听器对象使用了套接字，*backlog* (默认值为 1) 会在套接字绑定后传递给它的*listen()* 方法。

如果提供了 *authkey* 参数并且不是 None，那它必须是一个字符串并且会被当做基于 HMAC 认证的密钥。如果 *authkey* 是 None 则不会有认证行为。认证失败抛出*AuthenticationError* 异常，请查看 See 认证密码 。

accept()
接受一个连接并返回一个*Connection* 对象，其连接到的监听器对象已绑定套接字或者命名管道。如果已经尝试过认证并且失败了，则会抛出*AuthenticationError* 异常。

close()
关闭监听器对象上的绑定套接字或者命名管道。此函数会在监听器被垃圾回收后自动调用。不过仍然建议显式调用函数关闭。

监听器对象拥有下列只读属性:

address
监听器对象使用的地址。

last_accepted
最后一个连接所使用的地址。如果没有的话就是 `None` 。

3.3 版新加入: 监听器对象现在支持了上下文管理协议 - 见上下文管理器类型 。*__enter__()* 返回一个监听器对象, *__exit__()* 会调用*close()* 。

multiprocessing.connection.**wait**(*object_list*, *timeout=None*)
一直等待直到 *object_list* 中某个对象处于就绪状态。返回 *object_list* 中处于就绪状态的对象。如果 *timeout* 是一个浮点型，该方法会最多阻塞这么多秒。如果 *timeout* 是 `None` ，则会允许阻塞的事件没有限制。timeout 为负数的情况下和为 0 的情况相同。

对于 Unix 和 Windows，满足下列条件的对象可以出现在 *object_list* 中

- 可读的*Connection* 对象；
- 一个已连接并且可读的*socket.socket* 对象；或者
- *Process* 对象中的*sentinel* 属性。

当一个连接或者套接字对象拥有有效的数据可被读取的时候，或者另一端关闭后，这个对象就处于就绪状态。

Unix: `wait(object_list, timeout)` 和 `select.select(object_list, [], [], timeout)` 几乎相同。差别在于，如果*select.select()* 被信号中断，它会抛出一个附带错误号为 `EINTR` 的*OSError* 异常，而*wait()* 不会。

Windows: *object_list* 中的元素必须是一个表示为整数的可等待的句柄 (按照 Win32 函数 `WaitForMultipleObjects()` 的文档中所定义) 或者一个拥有 `fileno()` 方法的对象，这个对象返回一个套接字句柄或者管道句柄。(注意管道和套接字两种句柄 **不是**可等待的句柄)

3.3 版新加入.

示例

下面的服务代码创建了一个使用 `'secret password'` 作为认证密码的监听器。它会等待连接然后发送一些数据给客户端:

```
from multiprocessing.connection import Listener
from array import array

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'

with Listener(address, authkey=b'secret password') as listener:
    with listener.accept() as conn:
        print('connection accepted from', listener.last_accepted)

        conn.send([2.25, None, 'junk', float])

        conn.send_bytes(b'hello')

        conn.send_bytes(array('i', [42, 1729]))
```

下面的代码连接到服务然后从服务器上 j 接收一些数据:

```
from multiprocessing.connection import Client
from array import array

address = ('localhost', 6000)

with Client(address, authkey=b'secret password') as conn:
    print(conn.recv())                  # => [2.25, None, 'junk', float]

    print(conn.recv_bytes())            # => 'hello'

    arr = array('i', [0, 0, 0, 0, 0])
    print(conn.recv_bytes_into(arr))    # => 8
    print(arr)                          # => array('i', [42, 1729, 0, 0, 0])
```

下面的代码使用了*wait()*，以便在同时等待多个进程发来消息。

```
import time, random
from multiprocessing import Process, Pipe, current_process
```

(下页继续)

(繼續上一頁)

```
from multiprocessing.connection import wait

def foo(w):
    for i in range(10):
        w.send((i, current_process().name))
    w.close()

if __name__ == '__main__':
    readers = []

    for i in range(4):
        r, w = Pipe(duplex=False)
        readers.append(r)
        p = Process(target=foo, args=(w,))
        p.start()
        # We close the writable end of the pipe now to be sure that
        # p is the only process which owns a handle for it.  This
        # ensures that when p closes its handle for the writable end,
        # wait() will promptly report the readable end as being ready.
        w.close()

    while readers:
        for r in wait(readers):
            try:
                msg = r.recv()
            except EOFError:
                readers.remove(r)
            else:
                print(msg)
```

地址格式

- `'AF_INET'` 地址是 `(hostname, port)` 形式的元组类型，其中 *hostname* 是一个字符串，*port* 是整数。
- `'AF_UNIX'` 地址是文件系统上文件名的字符串。
- `'AF_PIPE'` 地址是一个 `r'\.\pipe{PipeName}'` 形式的字符串。要使用*Client()* 来连接到远程计算机上一个名为 *ServerName* 的命名管道，则应当改用 `r'\`*ServerName*`\pipe{PipeName}'` 形式的地址。

注意，使用两个反斜线开头的字符串默认被当做 `'AF_PIPE'` 地址而不是 `'AF_UNIX'` 。

认证密码

当使用*Connection.recv* 接收数据时，数据会自动被反序列化。不幸的是，对于一个不可信的数据源发来的数据，反序列化是存在安全风险的。所以*Listener* 和*Client()* 之间使用*hmac* 模块进行摘要认证。

认证密钥是一个 byte 类型的字符串，可以认为是和密码一样的东西，连接建立好后，双方都会要求另一方证明知道认证密钥。（这个证明过程不会通过连接发送密钥）

如果要求认证但是没有指定认证密钥，则会使用 `current_process().authkey` 的返回值 (参见*Process*)。这个值会被当前进程所创建的任何*Process* 对象自动继承。这意味着（默认情况下）当一个多进程程序的所有进程在彼此之间建立连接的时候，会共享同一个认证密钥。

os.urandom() 也可以用来生成合适的认证密钥。

日志

当前模块也提供了一些对 logging 的支持。注意，*logging* 模块本身并没有使用进程间共享的锁，所以来自于多个进程的日志可能（具体取决于使用的日志 handler 类型）相互覆盖或者混杂。

multiprocessing.**get_logger**()
返回*multiprocessing* 使用的 logger，必要的话会创建一个新的。

如果创建的首个 logger 日志级别为 `logging.NOTSET` 并且没有默认 handler。通过这个 logger 打印的消息不会传递到根 logger。

注意在 Windows 上，子进程只会继承父进程 logger 的日志级别 - 对于 logger 的其他自定义项不会继承。

multiprocessing.**log_to_stderr**(*level=None*)
This function performs a call to *get_logger()* but in addition to returning the logger created by get_logger, it adds a handler which sends output to *sys.stderr* using format `'[%(levelname)s/%(processName)s] %(message)s'`. You can modify `levelname` of the logger by passing a `level` argument.

下面是一个在交互式解释器中打开日志功能的例子:

```
>>> import multiprocessing, logging
>>> logger = multiprocessing.log_to_stderr()
>>> logger.setLevel(logging.INFO)
>>> logger.warning('doomed')
[WARNING/MainProcess] doomed
>>> m = multiprocessing.Manager()
[INFO/SyncManager-...] child process calling self.run()
[INFO/SyncManager-...] created temp directory /.../pymp-...
[INFO/SyncManager-...] manager serving at '/.../listener-...'
>>> del m
[INFO/MainProcess] sending shutdown message to manager
[INFO/SyncManager-...] manager exiting with exitcode 0
```

要查看日志等级的完整列表，见*logging* 模块。

multiprocessing.dummy 模块

multiprocessing.dummy 复制了*multiprocessing* 的 API，不过是在*threading* 模块之上包装了一层。

特别地，*multiprocessing.dummy* 所提供的 `Pool` 函数会返回一个*ThreadPool* 的实例，该类是*Pool* 的子类，它支持所有相同的方法调用但会使用一个工作线程池而非工作进程池。

class multiprocessing.pool.**ThreadPool**([*processes*[, *initializer*[, *initargs*]]])
一个线程池对象，用来控制可向其提交任务的工作线程池。*ThreadPool* 实例与*Pool* 实例是完全接口兼容的，并且它们的资源也必须被正确地管理，或者是将线程池作为上下文管理器来使用，或者是通过手动调用*close()* 和*terminate()*。

processes 是要使用的工作线程数目。如果 *processes* 为 `None`，则使用*os.cpu_count()* 返回的值。

如果 *initializer* 不为 `None`，则每个工作进程将会在启动时调用 `initializer(*initargs)`。

不同于*Pool*，*maxtasksperchild* 和 *context* 不可被提供。

> 備F: *ThreadPool* 具有与*Pool* 相同的接口，它围绕一个进程池进行设计并且先于*concurrent.futures* 模块的引入。因此，它继承了一些对于基于线程的池来说没有意义的操作，并且它具有自己的用于表示异步任务状态的类型*AsyncResult*，该类型不为任何其他库所知。

用户通常应该倾向于使用*concurrent.futures.ThreadPoolExecutor*，它拥有从一开始就围绕线程进行设计的更简单接口，并且返回与许多其他库相兼容的*concurrent.futures.Future*实例，包括*asyncio*库。

17.2.3 编程指导

使用*multiprocessing*时，应遵循一些指导原则和习惯用法。

所有 start 方法

下面这些适用于所有 start 方法。

避免共享状态

应该尽可能避免在进程间传递大量数据，越少越好。

最好坚持使用队列或者管道进行进程间通信，而不是底层的同步原语。

可序列化

保证所代理的方法的参数是可以序列化的。

代理的线程安全性

不要在多线程中同时使用一个代理对象，除非你用锁保护它。

（而在不同进程中使用 相同的代理对象却没有问题。）

使用 Join 避免僵尸进程

在 Unix 上，如果一个进程执行完成但是没有被 join，就会变成僵尸进程。一般来说，僵尸进程不会很多，因为每次新启动进程（或者*active_children()*被调用）时，所有已执行完成且没有被 join 的进程都会自动被 join，而且对一个执行完的进程调用*Process.is_alive*也会 join 这个进程。尽管如此，对自己启动的进程显式调用 join 依然是最佳实践。

继承优于序列化、反序列化

当使用 *spawn* 或者 *forkserver* 的启动方式时，*multiprocessing*中的许多类型都必须是可序列化的，这样子进程才能使用它们。但是通常我们都应该避免使用管道和队列发送共享对象到另外一个进程，而是重新组织代码，对于其他进程创建出来的共享对象，让那些需要访问这些对象的子进程可以直接将这些对象从父进程继承过来。

避免杀死进程

通过*Process.terminate*停止一个进程很容易导致这个进程正在使用的共享资源（如锁、信号量、管道和队列）损坏或者变得不可用，无法在其他进程中继续使用。

所以，最好只对那些从来不使用共享资源的进程调用*Process.terminate*。

Join 使用队列的进程

记住，往队列放入数据的进程会一直等待直到队列中所有项被”feeder” 线程传给底层管道。（子进程可以调用队列的*Queue.cancel_join_thread*方法禁止这种行为）

这意味着，任何使用队列的时候，你都要确保在进程 join 之前，所有存放到队列中的项将会被其他进程、线程完全消费。否则不能保证这个写过队列的进程可以正常终止。记住非精灵进程会自动 join 。

下面是一个会导致死锁的例子:

```
from multiprocessing import Process, Queue

def f(q):
    q.put('X' * 1000000)

if __name__ == '__main__':
    queue = Queue()
    p = Process(target=f, args=(queue,))
    p.start()
    p.join()                    # this deadlocks
    obj = queue.get()
```

交换最后两行可以修复这个问题（或者直接删掉 `p.join()`）。

显式传递资源给子进程

在 Unix 上，使用 *fork* 方式启动的子进程可以使用父进程中全局创建的共享资源。不过，最好是显式将资源对象通过参数的形式传递给子进程。

除了（部分原因）让代码兼容 Windows 以及其他的进程启动方式外，这种形式还保证了在子进程生命期这个对象是不会被父进程垃圾回收的。如果父进程中的某些对象被垃圾回收会导致资源释放，这就变得很重要。

所以对于实例：

```
from multiprocessing import Process, Lock

def f():
    ... do something using "lock" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f).start()
```

应当重写成这样：

```
from multiprocessing import Process, Lock

def f(l):
    ... do something using "l" ...

if __name__ == '__main__':
    lock = Lock()
    for i in range(10):
        Process(target=f, args=(lock,)).start()
```

谨防将*sys.stdin*数据替换为“类似文件的对象”

multiprocessing 原本会无条件地这样调用：

```
os.close(sys.stdin.fileno())
```

在 `multiprocessing.Process._bootstrap()` 方法中——这会导致与“进程中的进程”相关的一些问题。这已经被修改成了：

```
sys.stdin.close()
sys.stdin = open(os.open(os.devnull, os.O_RDONLY), closefd=False)
```

它解决了进程相互冲突导致文件描述符错误的根本问题，但是对使用带缓冲的“文件类对象”替换*sys.stdin()* 作为输出的应用程序造成了潜在的危险。如果多个进程调用了此文件类对象的*close()* 方法，会导致相同的数据多次刷写到此对象，损坏数据。

如果你写入文件类对象并实现了自己的缓存，可以在每次追加缓存数据时记录当前进程 id，从而将其变成 fork 安全的，当发现进程 id 变化后舍弃之前的缓存，例如:

```
@property
def cache(self):
    pid = os.getpid()
    if pid != self._pid:
        self._pid = pid
        self._cache = []
    return self._cache
```

需要更多信息，请查看 bpo-5155, bpo-5313 以及 bpo-5331

spawn 和 *forkserver* 启动方式

相对于 *fork* 启动方式，有一些额外的限制。

更依赖序列化

Process.__init__() 的所有参数都必须可序列化。同样的，当你继承*Process* 时，需要保证当调用*Process.start* 方法时，实例可以被序列化。

全局变量

记住，如果子进程中的代码尝试访问一个全局变量，它所看到的值（如果有）可能和父进程中执行*Process.start* 那一刻的值不一样。

当全局变量知识模块级别的常量时，是不会有问题的。

安全导入主模块

确保主模块可以被新启动的 Python 解释器安全导入而不会引发什么副作用（比如又启动了一个子进程）

例如，使用 *spawn* 或 *forkserver* 启动方式执行下面的模块，会引发*RuntimeError* 异常而失败。

```
from multiprocessing import Process

def foo():
    print('hello')

p = Process(target=foo)
p.start()
```

应该通过下面的方法使用 if __name__ == '__main__': ，从而保护程序” 入口点”:

```
from multiprocessing import Process, freeze_support, set_start_method

def foo():
    print('hello')

if __name__ == '__main__':
    freeze_support()
    set_start_method('spawn')
    p = Process(target=foo)
    p.start()
```

（如果程序将正常运行而不是冻结，则可以省略 `freeze_support()` 行）

这允许新启动的 Python 解释器安全导入模块然后运行模块中的 `foo()` 函数。

如果主模块中创建了进程池或者管理器，这个规则也适用。

17.2.4 示例

创建和使用自定义管理器、代理的示例:

```
from multiprocessing import freeze_support
from multiprocessing.managers import BaseManager, BaseProxy
import operator

##

class Foo:
    def f(self):
        print('you called Foo.f()')
    def g(self):
        print('you called Foo.g()')
    def _h(self):
        print('you called Foo._h()')

# A simple generator function
def baz():
    for i in range(10):
        yield i*i

# Proxy type for generator objects
class GeneratorProxy(BaseProxy):
    _exposed_ = ['__next__']
    def __iter__(self):
        return self
    def __next__(self):
        return self._callmethod('__next__')

# Function to return the operator module
def get_operator_module():
    return operator

##

class MyManager(BaseManager):
    pass

# register the Foo class; make `f()` and `g()` accessible via proxy
MyManager.register('Foo1', Foo)

# register the Foo class; make `g()` and `_h()` accessible via proxy
MyManager.register('Foo2', Foo, exposed=('g', '_h'))

# register the generator function baz; use `GeneratorProxy` to make proxies
MyManager.register('baz', baz, proxytype=GeneratorProxy)

# register get_operator_module(); make public functions accessible via proxy
MyManager.register('operator', get_operator_module)
```

(下页继续)

(繼續上一頁)

```
##

def test():
    manager = MyManager()
    manager.start()

    print('-' * 20)

    f1 = manager.Foo1()
    f1.f()
    f1.g()
    assert not hasattr(f1, '_h')
    assert sorted(f1._exposed_) == sorted(['f', 'g'])

    print('-' * 20)

    f2 = manager.Foo2()
    f2.g()
    f2._h()
    assert not hasattr(f2, 'f')
    assert sorted(f2._exposed_) == sorted(['g', '_h'])

    print('-' * 20)

    it = manager.baz()
    for i in it:
        print('<%d>' % i, end=' ')
    print()

    print('-' * 20)

    op = manager.operator()
    print('op.add(23, 45) =', op.add(23, 45))
    print('op.pow(2, 94) =', op.pow(2, 94))
    print('op._exposed_ =', op._exposed_)

##

if __name__ == '__main__':
    freeze_support()
    test()
```

使用*Pool*:

```
import multiprocessing
import time
import random
import sys

#
# Functions used by test code
#

def calculate(func, args):
    result = func(*args)
```

(下页继续)

(繼續上一頁)

```
    return '%s says that %s%s = %s' % (
        multiprocessing.current_process().name,
        func.__name__, args, result
        )

def calculatestar(args):
    return calculate(*args)

def mul(a, b):
    time.sleep(0.5 * random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5 * random.random())
    return a + b

def f(x):
    return 1.0 / (x - 5.0)

def pow3(x):
    return x ** 3

def noop(x):
    pass

#
# Test code
#

def test():
    PROCESSES = 4
    print('Creating pool with %d processes\n' % PROCESSES)

    with multiprocessing.Pool(PROCESSES) as pool:
        #
        # Tests
        #

        TASKS = [(mul, (i, 7)) for i in range(10)] + \
                [(plus, (i, 8)) for i in range(10)]

        results = [pool.apply_async(calculate, t) for t in TASKS]
        imap_it = pool.imap(calculatestar, TASKS)
        imap_unordered_it = pool.imap_unordered(calculatestar, TASKS)

        print('Ordered results using pool.apply_async():')
        for r in results:
            print('\t', r.get())
        print()

        print('Ordered results using pool.imap():')
        for x in imap_it:
            print('\t', x)
        print()

        print('Unordered results using pool.imap_unordered():')
```

(下页继续)

(繼續上一頁)

```
        for x in imap_unordered_it:
            print('\t', x)
        print()

        print('Ordered results using pool.map() --- will block till complete:')
        for x in pool.map(calculatestar, TASKS):
            print('\t', x)
        print()

        #
        # Test error handling
        #

        print('Testing error handling:')

        try:
            print(pool.apply(f, (5,)))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.apply()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(pool.map(f, list(range(10))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from pool.map()')
        else:
            raise AssertionError('expected ZeroDivisionError')

        try:
            print(list(pool.imap(f, list(range(10)))))
        except ZeroDivisionError:
            print('\tGot ZeroDivisionError as expected from list(pool.imap())')
        else:
            raise AssertionError('expected ZeroDivisionError')

        it = pool.imap(f, list(range(10)))
        for i in range(10):
            try:
                x = next(it)
            except ZeroDivisionError:
                if i == 5:
                    pass
            except StopIteration:
                break
            else:
                if i == 5:
                    raise AssertionError('expected ZeroDivisionError')

        assert i == 9
        print('\tGot ZeroDivisionError as expected from IMapIterator.next()')
        print()

        #
        # Testing timeouts
        #
```

(下页继续)

(繼續上一頁)

```
        print('Testing ApplyResult.get() with timeout:', end=' ')
        res = pool.apply_async(calculate, TASKS[0])
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % res.get(0.02))
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()

        print('Testing IMapIterator.next() with timeout:', end=' ')
        it = pool.imap(calculatestar, TASKS)
        while 1:
            sys.stdout.flush()
            try:
                sys.stdout.write('\n\t%s' % it.next(0.02))
            except StopIteration:
                break
            except multiprocessing.TimeoutError:
                sys.stdout.write('.')
        print()
        print()


if __name__ == '__main__':
    multiprocessing.freeze_support()
    test()
```

一个演示如何使用队列来向一组工作进程提供任务并收集结果的例子：

```
import time
import random

from multiprocessing import Process, Queue, current_process, freeze_support

#
# Function run by worker processes
#

def worker(input, output):
    for func, args in iter(input.get, 'STOP'):
        result = calculate(func, args)
        output.put(result)

#
# Function used to calculate result
#

def calculate(func, args):
    result = func(*args)
    return '%s says that %s%s = %s' % \
        (current_process().name, func.__name__, args, result)
```

(下页继续)

(繼續上一頁)

```
#
# Functions referenced by tasks
#

def mul(a, b):
    time.sleep(0.5*random.random())
    return a * b

def plus(a, b):
    time.sleep(0.5*random.random())
    return a + b

#
#
#

def test():
    NUMBER_OF_PROCESSES = 4
    TASKS1 = [(mul, (i, 7)) for i in range(20)]
    TASKS2 = [(plus, (i, 8)) for i in range(10)]

    # Create queues
    task_queue = Queue()
    done_queue = Queue()

    # Submit tasks
    for task in TASKS1:
        task_queue.put(task)

    # Start worker processes
    for i in range(NUMBER_OF_PROCESSES):
        Process(target=worker, args=(task_queue, done_queue)).start()

    # Get and print results
    print('Unordered results:')
    for i in range(len(TASKS1)):
        print('\t', done_queue.get())

    # Add more tasks using `put()`
    for task in TASKS2:
        task_queue.put(task)

    # Get and print some more results
    for i in range(len(TASKS2)):
        print('\t', done_queue.get())

    # Tell child processes to stop
    for i in range(NUMBER_OF_PROCESSES):
        task_queue.put('STOP')


if __name__ == '__main__':
    freeze_support()
    test()
```

17.3 multiprocessing.shared_memory --- 可从进程直接访问的共享内存

源代码: Lib/multiprocessing/shared_memory.py

3.8 版新加入.

该模块提供了一个*SharedMemory* 类，用于分配和管理多核或对称多处理器（SMP）机器上进程间的共享内存。为了协助管理不同进程间的共享内存生命周期，multiprocessing.managers 模块也提供了一个*BaseManager* 的子类：SharedMemoryManager。

本模块中，共享内存是指"System V 类型" 的共享内存块（虽然实现方式可能不完全一致）而不是 "分布式共享内存"。这种类型的的共享内存允许不同进程读写一片公共（或者共享）的易失性存储区域。一般来说，进程被限制只能访问属于自己进程空间的内存，但是共享内存允许跨进程共享数据，从而避免通过进程间发送消息的形式传递数据。与通过磁盘、套接字或者其他要求序列化、反序列化和复制数据的共享形式相比，直接通过内存共享数据拥有更出色的性能。

class multiprocessing.shared_memory.**SharedMemory**(*name=None*, *create=False*, *size=0*)

创建一个新的共享内存块或者连接到一片已经存在的共享内存块。每个共享内存块都被指定了一个全局唯一的名称。通过这种方式，进程可以使用一个特定的名字创建共享内存区块，然后其他进程使用同样的名字连接到这个共享内存块。

作为一种跨进程共享数据的方式，共享内存块的寿命可能超过创建它的原始进程。一个共享内存块可能同时被多个进程使用，当一个进程不再需要访问这个共享内存块的时候，应该调用*close()* 方法。当一个共享内存块不被任何进程使用的时候，应该调用*unlink()* 方法以保证必要的清理。

name 是共享内存的唯一名称，字符串类型。如果创建一个新共享内存块的时候，名称指定为 None (默认值)，将会随机产生一个新名称。

create 指定创建一个新的共享内存块 (True) 还是连接到已存在的共享内存块 (False) 。

如果是新创建共享内存块则 *size* 用于指定块的大小为多少字节。由于某些平台是以内存页大小为最小单位来分配内存的，最终得到的内存块大小可能大于或等于要求的大小。如果是连接到已经存在的共享内存块，size 参数会被忽略。

close()

关闭实例对于共享内存的访问连接。所有实例确认自己不再需要使用共享内存的时候都应该调用 close() ，以保证必要的资源清理。调用 close() 并不会销毁共享内存区域。

unlink()

请求销毁底层的共享内存块。为了执行必要的资源清理，在所有使用这个共享内存块的进程中，unlink() 应该调用一次（且只能调用一次）。发出此销毁请求后，共享内存块可能会、也可能不会立即销毁，且此行为在不同操作系统之间可能不同。调用 unlink() 后再尝试访问其中的数据可能导致内存错误。注意：最后一个关闭共享内存访问权限的进程可以以任意顺序调用 unlink() 和*close()*。

buf

共享内存块内容的 memoryview 。

name

共享内存块的唯一标识，只读属性。

size

共享内存块的字节大小，只读属性。

以下示例展示了*SharedMemory* 底层的用法:

```
>>> from multiprocessing import shared_memory
>>> shm_a = shared_memory.SharedMemory(create=True, size=10)
>>> type(shm_a.buf)
<class 'memoryview'>
>>> buffer = shm_a.buf
>>> len(buffer)
10
>>> buffer[:4] = bytearray([22, 33, 44, 55])  # Modify multiple at once
>>> buffer[4] = 100                           # Modify single byte at a time
>>> # Attach to an existing shared memory block
>>> shm_b = shared_memory.SharedMemory(shm_a.name)
>>> import array
>>> array.array('b', shm_b.buf[:5])  # Copy the data into a new array.array
array('b', [22, 33, 44, 55, 100])
>>> shm_b.buf[:5] = b'howdy'  # Modify via shm_b using bytes
>>> bytes(shm_a.buf[:5])      # Access via shm_a
b'howdy'
>>> shm_b.close()   # Close each SharedMemory instance
>>> shm_a.close()
>>> shm_a.unlink()  # Call unlink only once to release the shared memory
```

以下示例展示了一个现实中的例子，使用*SharedMemory* 类和 NumPy arrays 结合，从两个 Python shell 中访问同一个 `numpy.ndarray`：

```
>>> # In the first Python interactive shell
>>> import numpy as np
>>> a = np.array([1, 1, 2, 3, 5, 8])  # Start with an existing NumPy array
>>> from multiprocessing import shared_memory
>>> shm = shared_memory.SharedMemory(create=True, size=a.nbytes)
>>> # Now create a NumPy array backed by shared memory
>>> b = np.ndarray(a.shape, dtype=a.dtype, buffer=shm.buf)
>>> b[:] = a[:]  # Copy the original data into shared memory
>>> b
array([1, 1, 2, 3, 5, 8])
>>> type(b)
<class 'numpy.ndarray'>
>>> type(a)
<class 'numpy.ndarray'>
>>> shm.name  # We did not specify a name so one was chosen for us
'psm_21467_46075'

>>> # In either the same shell or a new Python shell on the same machine
>>> import numpy as np
>>> from multiprocessing import shared_memory
>>> # Attach to the existing shared memory block
>>> existing_shm = shared_memory.SharedMemory(name='psm_21467_46075')
>>> # Note that a.shape is (6,) and a.dtype is np.int64 in this example
>>> c = np.ndarray((6,), dtype=np.int64, buffer=existing_shm.buf)
>>> c
array([1, 1, 2, 3, 5, 8])
>>> c[-1] = 888
>>> c
array([  1,   1,   2,   3,   5, 888])

>>> # Back in the first Python interactive shell, b reflects this change
>>> b
array([  1,   1,   2,   3,   5, 888])
```

(下页继续)

(繼續上一頁)

```
>>> # Clean up from within the second Python shell
>>> del c  # Unnecessary; merely emphasizing the array is no longer used
>>> existing_shm.close()

>>> # Clean up from within the first Python shell
>>> del b  # Unnecessary; merely emphasizing the array is no longer used
>>> shm.close()
>>> shm.unlink()  # Free and release the shared memory block at the very end
```

class multiprocessing.managers.**SharedMemoryManager**([*address*[, *authkey*]])

BaseManager 的子类，可用于管理跨进程的共享内存块。

调用*SharedMemoryManager* 实例上的*start()* 方法会启动一个新进程。这个新进程的唯一目的就是管理所有由它创建的共享内存块的生命周期。想要释放此进程管理的所有共享内存块，可以调用实例的*shutdown()* 方法。这会触发执行它管理的所有*SharedMemory* 对象的 SharedMemory.unlink() 方法，然后停止这个进程。通过 SharedMemoryManager 创建 SharedMemory 实例，我们可以避免手动跟踪和释放共享内存资源。

这个类提供了创建和返回*SharedMemory* 实例的方法，以及以共享内存为基础创建一个列表类对象 (*ShareableList*) 的方法。

有关继承的可选输入参数 *address* 和 *authkey* 以及他们如何用于从进程连接已经存在的 SharedMemoryManager 服务，参见*multiprocessing.managers.BaseManager* 。

SharedMemory(*size*)

使用 size 参数，创建一个新的指定字节大小的*SharedMemory* 对象并返回。

ShareableList(*sequence*)

创建并返回一个新的*ShareableList* 对象，通过输入参数 sequence 初始化。

下面的案例展示了 SharedMemoryManager 的基本机制:

```
>>> from multiprocessing.managers import SharedMemoryManager
>>> smm = SharedMemoryManager()
>>> smm.start()  # Start the process that manages the shared memory blocks
>>> sl = smm.ShareableList(range(4))
>>> sl
ShareableList([0, 1, 2, 3], name='psm_6572_7512')
>>> raw_shm = smm.SharedMemory(size=128)
>>> another_sl = smm.ShareableList('alpha')
>>> another_sl
ShareableList(['a', 'l', 'p', 'h', 'a'], name='psm_6572_12221')
>>> smm.shutdown()  # Calls unlink() on sl, raw_shm, and another_sl
```

以下案例展示了 SharedMemoryManager 对象的一种可能更方便的使用方式，通过 with 语句来保证所有共享内存块在使用完后被释放。

```
>>> with SharedMemoryManager() as smm:
...     sl = smm.ShareableList(range(2000))
...     # Divide the work among two processes, storing partial results in sl
...     p1 = Process(target=do_work, args=(sl, 0, 1000))
...     p2 = Process(target=do_work, args=(sl, 1000, 2000))
...     p1.start()
...     p2.start()  # A multiprocessing.Pool might be more efficient
...     p1.join()
...     p2.join()   # Wait for all work to complete in both processes
...     total_result = sum(sl)  # Consolidate the partial results now in sl
```

在 with 语句中使用 SharedMemoryManager 对象的时候，使用这个管理器创建的共享内存块会在 with 语句代码块结束后被释放。

class multiprocessing.shared_memory.**ShareableList**(*sequence=None*, *, *name=None*)

提供一个可修改的类 list 对象，其中所有值都存放在共享内存块中。这限制了可被存储在其中的值只能是 int, float, bool, str（每条数据小于 10M）, bytes（每条数据小于 10M）以及 None 这些内置类型。它另一个显著区别于内置 list 类型的地方在于它的长度无法修改（比如，没有 append, insert 等操作）且不支持通过切片操作动态创建新的*ShareableList* 实例。

sequence 会被用来为一个新的 ShareableList 填充值。设为 None 则会基于唯一的共享内存名称关联到已经存在的 ShareableList。

name 是所请求的共享内存的唯一名称，与*SharedMemory* 的定义中所描述的一致。当关联到现有的 ShareableList 时，则指明其共享内存块的唯一名称并将 sequence 设为 None。

count(*value*)

返回 value 出现的次数。

index(*value*)

返回 value 首次出现的位置，如果 value 不存在, 则抛出*ValueError* 异常。

format

包含由所有当前存储值所使用的*struct* 打包格式的只读属性。

shm

存储了值的*SharedMemory* 实例。

下面的例子演示了*ShareableList* 实例的基本用法:

```
>>> from multiprocessing import shared_memory
>>> a = shared_memory.ShareableList(['howdy', b'HoWdY', -273.154, 100, None, True,␣
↪42])
>>> [ type(entry) for entry in a ]
[<class 'str'>, <class 'bytes'>, <class 'float'>, <class 'int'>, <class 'NoneType'>,
↪<class 'bool'>, <class 'int'>]
>>> a[2]
-273.154
>>> a[2] = -78.5
>>> a[2]
-78.5
>>> a[2] = 'dry ice'  # Changing data types is supported as well
>>> a[2]
'dry ice'
>>> a[2] = 'larger than previously allocated storage space'
Traceback (most recent call last):
  ...
ValueError: exceeds available storage for existing str
>>> a[2]
'dry ice'
>>> len(a)
7
>>> a.index(42)
6
>>> a.count(b'howdy')
0
>>> a.count(b'HoWdY')
1
>>> a.shm.close()
>>> a.shm.unlink()
>>> del a  # Use of a ShareableList after call to unlink() is unsupported
```

下面的例子演示了一个、两个或多个进程如何通过提供下层的共享内存块名称来访问同一个*ShareableList*:

```
>>> b = shared_memory.ShareableList(range(5))          # In a first process
>>> c = shared_memory.ShareableList(name=b.shm.name)   # In a second process
>>> c
ShareableList([0, 1, 2, 3, 4], name='...')
>>> c[-1] = -999
>>> b[-1]
-999
>>> b.shm.close()
>>> c.shm.close()
>>> c.shm.unlink()
```

17.4 concurrent 包

目前，此包中只有一个模块:

- *concurrent.futures* ——启动并行任务

17.5 concurrent.futures --- 启动并行任务

3.2 版新加入.

源码: Lib/concurrent/futures/thread.py 和 Lib/concurrent/futures/process.py

---

concurrent.futures 模块提供异步执行回调高层接口。

异步执行可以由*ThreadPoolExecutor* 使用线程或由*ProcessPoolExecutor* 使用单独的进程来实现。两者都是实现抽像类*Executor* 定义的接口。

17.5.1 Executor 对象

class concurrent.futures.**Executor**

抽象类提供异步执行调用方法。要通过它的子类调用，而不是直接调用。

submit(*fn*, */*, **args*, ***kwargs*)

Schedules the callable, *fn*, to be executed as `fn(*args, **kwargs)` and returns a *Future* object representing the execution of the callable.

```
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(pow, 323, 1235)
    print(future.result())
```

map(*func*, **iterables*, *timeout=None*, *chunksize=1*)

类似于*map(func, *iterables)* 函数，除了以下两点:

- *iterables* 是立即执行而不是延迟执行的;
- *func* 是异步执行的，对 *func* 的多个调用可以并发执行。

如果*__next__()* 已被调用且返回的结果在对*Executor.map()* 的原始调用经过 *timeout* 秒后还不可用，则已返回的迭代器将引发*concurrent.futures.TimeoutError*。*timeout* 可以为 int 或 float 类型。如果 *timeout* 未指定或为 `None`，则不限制等待时间。

如果 *func* 调用引发一个异常，当从迭代器中取回它的值时这个异常将被引发。

使用*ProcessPoolExecutor* 时，这个方法会将 *iterables* 分割任务块并作为独立的任务并提交到执行池中。这些块的大概数量可以由 *chunksize* 指定正整数设置。对很长的迭代器来说，使用大的 *chunksize* 值比默认值 1 能显著地提高性能。*chunksize* 对*ThreadPoolExecutor* 没有效果。

3.5 版更變: 加入 *chunksize* 参数。

shutdown(*wait=True*, *, *cancel_futures=False*)

当待执行的 future 对象完成执行后向执行者发送信号，它就会释放正在使用的任何资源。在关闭后调用*Executor.submit()* 和*Executor.map()* 将会引发*RuntimeError*。

如果 *wait* 为 `True` 则此方法只有在所有待执行的 future 对象完成执行且释放已分配的资源后才会返回。如果 *wait* 为 `False`，方法立即返回，所有待执行的 future 对象完成执行后会释放已分配的资源。不管 *wait* 的值是什么，整个 Python 程序将等到所有待执行的 future 对象完成执行后才退出。

如果 *cancel_futures* 为 `True`，此方法将取消所有执行器还未开始运行的挂起的 Future。任何已完成或正在运行的 Future 将不会被取消，无论 *cancel_futures* 的值是什么?

如果 *cancel_futures* 和 *wait* 均为 `True`，则执行器已开始运行的所有 Future 将在此方法返回之前完成。其余的 Future 会被取消。

如果使用 `with` 语句，你就可以避免显式调用这个方法，它将会停止*Executor* (就好像*Executor.shutdown()* 调用时 *wait* 设为 `True` 一样等待):

```
import shutil
with ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

3.9 版更變: 增加了 *cancel_futures*。

17.5.2 ThreadPoolExecutor

ThreadPoolExecutor 是*Executor* 的子类，它使用线程池来异步执行调用。

当回调已关联了一个*Future* 然后再等待另一个*Future* 的结果时就会发生死锁情况。例如:

```
import time
def wait_on_b():
    time.sleep(5)
    print(b.result())  # b will never complete because it is waiting on a.
    return 5

def wait_on_a():
    time.sleep(5)
    print(a.result())  # a will never complete because it is waiting on b.
    return 6
```

(下页继续)

(繼續上一頁)

```
executor = ThreadPoolExecutor(max_workers=2)
a = executor.submit(wait_on_b)
b = executor.submit(wait_on_a)
```

And:

```
def wait_on_future():
    f = executor.submit(pow, 5, 2)
    # This will never complete because there is only one worker thread and
    # it is executing this function.
    print(f.result())

executor = ThreadPoolExecutor(max_workers=1)
executor.submit(wait_on_future)
```

class concurrent.futures.**ThreadPoolExecutor**(*max_workers=None*, *thread_name_prefix=''*, *initializer=None*, *initargs=()*)

Executor 的一个子类，使用最多 *max_workers* 个线程的线程池来异步执行调用。

initializer 是在每个工作者线程开始处调用的一个可选可调用对象。*initargs* 是传递给初始化器的元组参数。任何向池提交更多工作的尝试，*initializer* 都将引发一个异常，当前所有等待的工作都会引发一个*BrokenThreadPool*。

3.5 版更變: 如果 *max_workers* 为 None 或没有指定，将默认为机器处理器的个数，假如*ThreadPoolExecutor* 侧重于 I/O 操作而不是 CPU 运算，那么可以乘以 5，同时工作线程的数量可以比*ProcessPoolExecutor* 的数量高。

3.6 版新加入: 添加 *thread_name_prefix* 参数允许用户控制由线程池创建的*threading.Thread* 工作线程名称以方便调试。

3.7 版更變: 加入 *initializer* 和 *initargs* 参数。

3.8 版更變: *max_workers* 的默认值已改为 `min(32, os.cpu_count() + 4)`。这个默认值会保留至少 5 个工作线程用于 I/O 密集型任务。对于那些释放了 GIL 的 CPU 密集型任务，它最多会使用 32 个 CPU 核心。这样能够避免在多核机器上不知不觉地使用大量资源。

现在 ThreadPoolExecutor 在启动 *max_workers* 个工作线程之前也会重用空闲的工作线程。

ThreadPoolExecutor 例子

```
import concurrent.futures
import urllib.request

URLS = ['http://www.foxnews.com/',
        'http://www.cnn.com/',
        'http://europe.wsj.com/',
        'http://www.bbc.co.uk/',
        'http://some-made-up-domain.com/']

# Retrieve a single page and report the URL and contents
def load_url(url, timeout):
    with urllib.request.urlopen(url, timeout=timeout) as conn:
        return conn.read()

# We can use a with statement to ensure threads are cleaned up promptly
with concurrent.futures.ThreadPoolExecutor(max_workers=5) as executor:
```

(下页继续)

(繼續上一頁)

```
    # Start the load operations and mark each future with its URL
    future_to_url = {executor.submit(load_url, url, 60): url for url in URLS}
    for future in concurrent.futures.as_completed(future_to_url):
        url = future_to_url[future]
        try:
            data = future.result()
        except Exception as exc:
            print('%r generated an exception: %s' % (url, exc))
        else:
            print('%r page is %d bytes' % (url, len(data)))
```

17.5.3 ProcessPoolExecutor

ProcessPoolExecutor 类是*Executor* 的子类，它使用进程池来异步地执行调用。*ProcessPoolExecutor* 会使用*multiprocessing* 模块，这允许它绕过*全局解释器锁* 但也意味着只可以处理和返回可封存的对象。

__main__ 模块必须可以被工作者子进程导入。这意味着*ProcessPoolExecutor* 不可以工作在交互式解释器中。

从提交给*ProcessPoolExecutor* 的回调中调用*Executor* 或*Future* 方法会导致死锁。

class concurrent.futures.**ProcessPoolExecutor**(*max_workers=None*, *mp_context=None*, *initializer=None*, *initargs=()*)

异步地执行调用的*Executor* 子类使用最多具有 *max_workers* 个进程的进程池。如果 *max_workers* 为 None 或未给出，它将默认为机器的处理器个数。如果 *max_workers* 小于等于 0，则将引发*ValueError*。在 Windows 上，*max_workers* 必须小于等于 61，否则将引发*ValueError*。如果 *max_workers* 为 None，则所选择的默认值最多为 61，即使存在更多的处理器。*mp_context* 可以是一个多进程上下文或是 None。它将被用来启动工作进程。如果 *mp_context* 为 None 或未给出，则将使用默认的多进程上下文。

initializer 是一个可选的可调用对象，它会在每个工作进程启动时被调用；*initargs* 是传给 initializer 的参数元组。如果 *initializer* 引发了异常，则所有当前在等待的任务以及任何向进程池提交更多任务的尝试都将引发*BrokenProcessPool*。

3.3 版更變: 如果其中一个工作进程被突然终止，BrokenProcessPool 就会马上触发。可预计的行为没有定义，但执行器上的操作或它的 future 对象会被冻结或死锁。

3.7 版更變: 添加 *mp_context* 参数允许用户控制由进程池创建给工作者进程的开始方法。

加入 *initializer* 和 *initargs* 参数。

ProcessPoolExecutor 例子

```
import concurrent.futures
import math

PRIMES = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def is_prime(n):
```

(下页继续)

(繼續上一頁)

```
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    sqrt_n = int(math.floor(math.sqrt(n)))
    for i in range(3, sqrt_n + 1, 2):
        if n % i == 0:
            return False
    return True

def main():
    with concurrent.futures.ProcessPoolExecutor() as executor:
        for number, prime in zip(PRIMES, executor.map(is_prime, PRIMES)):
            print('%d is prime: %s' % (number, prime))

if __name__ == '__main__':
    main()
```

17.5.4 Future 对象

Future 类将可调用对象封装为异步执行。*Future* 实例由*Executor.submit()* 创建。

class concurrent.futures.**Future**

将可调用对象封装为异步执行。*Future* 实例由*Executor.submit()* 创建，除非测试，不应直接创建。

cancel()

尝试取消调用。如果调用正在执行或已结束运行不能被取消则该方法将返回 `False`，否则调用会被取消并且该方法将返回 `True`。

cancelled()

如果调用成功取消返回 `True`。

running()

如果调用正在执行而且不能被取消那么返回 “True“。

done()

如果调用已被取消或正常结束那么返回 `True`。

result(*timeout=None*)

返回调用返回的值。如果调用还没完成那么这个方法将等待 *timeout* 秒。如果在 *timeout* 秒内没有执行完成，*concurrent.futures.TimeoutError* 将会被触发。*timeout* 可以是整数或浮点数。如果 *timeout* 没有指定或为 `None`，那么等待时间就没有限制。

如果 futrue 在完成前被取消则*CancelledError* 将被触发。

如果调用引发了一个异常，这个方法也会引发同样的异常。

exception(*timeout=None*)

返回由调用引发的异常。如果调用还没完成那么这个方法将等待 *timeout* 秒。如果在 *timeout* 秒内没有执行完成，*concurrent.futures.TimeoutError* 将会被触发。*timeout* 可以是整数或浮点数。如果 *timeout* 没有指定或为 `None`，那么等待时间就没有限制。

如果 futrue 在完成前被取消则*CancelledError* 将被触发。

如果调用正常完成那么返回 None。

add_done_callback(*fn*)
附加可调用 *fn* 到 future 对象。当 future 对象被取消或完成运行时，将会调用 *fn*，而这个 future 对象将作为它唯一的参数。

加入的可调用对象总被属于添加它们的进程中的线程按加入的顺序调用。如果可调用对象引发一个*Exception* 子类，它会被记录下来并被忽略掉。如果可调用对象引发一个*BaseException* 子类，这个行为没有定义。

如果 future 对象已经完成或已取消，*fn* 会被立即调用。

下面这些*Future* 方法用于单元测试和*Executor* 实现。

set_running_or_notify_cancel()
这个方法只可以在执行关联*Future* 工作之前由*Executor* 实现调用或由单测试调用。

如果这个方法返回 False 那么*Future* 已被取消，即*Future.cancel()* 已被调用并返回 True 。等待*Future* 完成 (即通过*as_completed()* 或*wait()*) 的线程将被唤醒。

如果这个方法返回 True 那么*Future* 不会被取消并已将它变为正在运行状态，也就是说调用*Future.running()* 时将返回 *True*。

这个方法只可以被调用一次并且不能在调用*Future.set_result()* 或*Future.set_exception()* 之后再调用。

set_result(*result*)
设置将*Future* 关联工作的结果给 *result* 。

这个方法只可以由*Executor* 实现和单元测试使用。

3.8 版更變: 如果*Future* 已经完成则此方法会引发*concurrent.futures.InvalidStateError*。

set_exception(*exception*)
设置*Future* 关联工作的结果给*Exception* *exception* 。

这个方法只可以由*Executor* 实现和单元测试使用。

3.8 版更變: 如果*Future* 已经完成则此方法会引发*concurrent.futures.InvalidStateError*。

17.5.5 模块函数

concurrent.futures.**wait**(*fs*, *timeout=None*, *return_when=ALL_COMPLETED*)
Wait for the *Future* instances (possibly created by different *Executor* instances) given by *fs* to complete. Duplicate futures given to *fs* are removed and will be returned only once. Returns a named 2-tuple of sets. The first set, named done, contains the futures that completed (finished or cancelled futures) before the wait completed. The second set, named not_done, contains the futures that did not complete (pending or running futures).

timeout 可以用来控制返回前最大的等待秒数。*timeout* 可以为 int 或 float 类型。如果 *timeout* 未指定或为 None ，则不限制等待时间。

return_when 指定此函数应在何时返回。它必须为以下常数之一:

常数	描述
FIRST_COMPLETED	函数将在任意可等待对象结束或取消时返回。
FIRST_EXCEPTION	函数将在任意可等待对象因引发异常而结束时返回。当没有引发任何异常时它就相当于 ALL_COMPLETED。
ALL_COMPLETED	函数将在所有可等待对象结束或取消时返回。

concurrent.futures.**as_completed**(*fs*, *timeout=None*)
返回一个包含 *fs* 所指定的*Future* 实例（可能由不同的*Executor* 实例创建）的迭代器，这些实例会在完成时生成 future 对象（包括正常结束或被取消的 future 对象）。任何由 *fs* 所指定的重复 future 对象将只被返回一次。任何在*as_completed()* 被调用之前完成的 future 对象将优先被生成。如果*__next__()* 被调用并且在对*as_completed()* 的原始调用 *timeout* 秒之后结果仍不可用，则返回的迭代器将引发*concurrent.futures.TimeoutError*。*timeout* 可以为整数或浮点数。如果 *timeout* 未指定或为 None，则不限制等待时间。

也参考:

PEP 3148 -- futures - 异步执行指令。 该提案描述了 Python 标准库中包含的这个特性。

17.5.6 Exception 类

exception concurrent.futures.**CancelledError**
future 对象被取消时会触发。

exception concurrent.futures.**TimeoutError**
future 对象执行超出给定的超时数值时引发。

exception concurrent.futures.**BrokenExecutor**
当执行器被某些原因中断而且不能用来提交或执行新任务时就会被引发派生于*RuntimeError* 的异常类。

3.7 版新加入.

exception concurrent.futures.**InvalidStateError**
当某个操作在一个当前状态所不允许的 future 上执行时将被引发。

3.8 版新加入.

exception concurrent.futures.thread.**BrokenThreadPool**
当 ThreadPoolExecutor 中的其中一个工作者初始化失败时会引发派生于*BrokenExecutor* 的异常类。

3.7 版新加入.

exception concurrent.futures.process.**BrokenProcessPool**
当 ThreadPoolExecutor 中的其中一个工作者不完整终止时 (比如，被外部杀死) 会引发派生于*BrokenExecutor* (原名*RuntimeError*) 的异常类。

3.3 版新加入.

17.6 subprocess --- 子进程管理

源代码: Lib/subprocess.py

---

subprocess 模块允许你生成新的进程，连接它们的输入、输出、错误管道，并且获取它们的返回码。此模块打算代替一些老旧的模块与功能：

```
os.system
os.spawn*
```

在下面的段落中，你可以找到关于*subprocess* 模块如何代替这些模块和功能的相关信息。

也参考:

PEP 324 -- 提出 subprocess 模块的 PEP

17.6.1 使用 subprocess 模块

推荐的调用子进程的方式是在任何它支持的用例中使用*run()* 函数。对于更进阶的用例，也可以使用底层的*Popen* 接口。

run() 函数是在 Python 3.5 被添加的；如果你需要与旧版本保持兼容，查看*较旧的高阶 API* 段落。

subprocess.**run**(*args*, ***, *stdin=None*, *input=None*, *stdout=None*, *stderr=None*, *capture_output=False*, *shell=False*, *cwd=None*, *timeout=None*, *check=False*, *encoding=None*, *errors=None*, *text=None*, *env=None*, *universal_newlines=None*, ***other_popen_kwargs*)

运行被 *arg* 描述的指令。等待指令完成，然后返回一个*CompletedProcess* 实例。

以上显示的参数仅仅是最简单的一些，下面*常用参数* 描述（因此在缩写签名中使用仅关键字标示）。完整的函数头和*Popen* 的构造函数一样，此函数接受的大多数参数都被传递给该接口。（*timeout*, *input*, *check* 和 *capture_output* 除外）。

如果 *capture_output* 设为 true，stdout 和 stderr 将会被捕获。在使用时，内置的*Popen* 对象将自动用 `stdout=PIPE` 和 `stderr=PIPE` 创建。*stdout* 和 *stderr* 参数不应当与 *capture_output* 同时提供。如果你希望捕获并将两个流合并在一起，使用 `stdout=PIPE` 和 `stderr=STDOUT` 来代替 *capture_output*。

timeout 参数将被传递给*Popen.communicate()*。如果发生超时，子进程将被杀死并等待。*TimeoutExpired* 异常将在子进程中断后被抛出。

input 参数将被传递给*Popen.communicate()* 以及子进程的标准输入。如果使用此参数，它必须是一个字节序列。如果指定了 *encoding* 或 *errors* 或者将 *text* 设置为 `True`，那么也可以是一个字符串。当使用此参数时，在创建内部*Popen* 对象时将自动带上 `stdin=PIPE`，并且不能再手动指定 *stdin* 参数。

如果 *check* 设为 True, 并且进程以非零状态码退出, 一个*CalledProcessError* 异常将被抛出. 这个异常的属性将设置为参数, 退出码, 以及标准输出和标准错误, 如果被捕获到.

如果 *encoding* 或者 *error* 被指定, 或者 *text* 被设为 True, 标准输入, 标准输出和标准错误的文件对象将通过指定的 *encoding* 和 *errors* 以文本模式打开, 否则以默认的*io.TextIOWrapper* 打开. *universal_newline* 参数等同于 *text* 并且提供了向后兼容性. 默认情况下, 文件对象是以二进制模式打开的.

如果 *env* 不是 `None`, 它必须是一个字典, 为新的进程设置环境变量; 它用于替换继承的当前进程的环境的默认行为. 它将直接被传递给*Popen*.

例如:

```
>>> subprocess.run(["ls", "-l"])  # doesn't capture output
CompletedProcess(args=['ls', '-l'], returncode=0)

>>> subprocess.run("exit 1", shell=True, check=True)
Traceback (most recent call last):
  ...
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1

>>> subprocess.run(["ls", "-l", "/dev/null"], capture_output=True)
CompletedProcess(args=['ls', '-l', '/dev/null'], returncode=0,
stdout=b'crw-rw-rw- 1 root root 1, 3 Jan 23 16:23 /dev/null\n', stderr=b'')
```

3.5 版新加入.

3.6 版更變: 添加了 *encoding* 和 *errors* 形参.

3.7 版更變: 添加了 *text* 形参, 作为 *universal_newlines* 的一个更好理解的别名. 添加了 *capture_output* 形参.

class subprocess.**CompletedProcess**

run() 的返回值, 代表一个进程已经结束.

args
被用作启动进程的参数. 可能是一个列表或字符串.

returncode
子进程的退出状态码. 通常来说, 一个为 0 的退出码表示进程运行正常.

一个负值 -N 表示子进程被信号 N 中断 (仅 POSIX).

stdout
从子进程捕获到的标准输出. 一个字节序列, 或一个字符串, 如果*run()* 是设置了 *encoding*, *errors* 或者 text=True 来运行的. 如果未有捕获, 则为 None.

如果你通过 stderr=subprocess.STDOUT 运行进程，标准输入和标准错误将被组合在这个属性中，并且*stderr* 将为 None。

stderr
捕获到的子进程的标准错误. 一个字节序列, 或者一个字符串, 如果*run()* 是设置了参数 *encoding*, *errors* 或者 text=True 运行的. 如果未有捕获, 则为 None.

check_returncode()
如果*returncode* 非零, 抛出*CalledProcessError*.

3.5 版新加入.

subprocess.**DEVNULL**
可被*Popen* 的 *stdin*, *stdout* 或者 *stderr* 参数使用的特殊值, 表示使用特殊文件*os.devnull*.

3.3 版新加入.

subprocess.**PIPE**
可被*Popen* 的 *stdin*, *stdout* 或者 *stderr* 参数使用的特殊值, 表示打开标准流的管道. 常用于*Popen.communicate()*.

subprocess.**STDOUT**
可被*Popen* 的 *stdin* ， *stdout* 或者 *stderr* 参数使用的特殊值，表示标准错误与标准输出使用同一句柄。

exception subprocess.**SubprocessError**
此模块的其他异常的基类。

3.3 版新加入.

exception subprocess.**TimeoutExpired**
SubprocessError 的子类，等待子进程的过程中发生超时时被抛出。

cmd
用于创建子进程的指令。

timeout
超时秒数。

output
子进程的输出，如果被*run()* 或*check_output()* 捕获。否则为 None。

stdout
对 output 的别名，对应的有*stderr*。

stderr
子进程的标准错误输出，如果被*run()* 捕获。否则为 None。

3.3 版新加入.

3.5 版更變: 添加了 *stdout* 和 *stderr* 属性。

exception subprocess.**CalledProcessError**
Subclass of *SubprocessError*, raised when a process run by *check_call()*, *check_output()*, or *run()* (with check=True) returns a non-zero exit status.

returncode
子进程的退出状态。如果程序由一个信号终止，这将会被设为一个负的信号码。

cmd
用于创建子进程的指令。

output
子进程的输出，如果被*run()* 或*check_output()* 捕获。否则为 None。

stdout
对 output 的别名，对应的有*stderr*。

stderr
子进程的标准错误输出，如果被*run()* 捕获。否则为 None。

3.5 版更變: 添加了 *stdout* 和 *stderr* 属性。

常用参数

为了支持丰富的使用案例，*Popen* 的构造函数（以及方便的函数）接受大量可选的参数。对于大多数典型的用例，许多参数可以被安全地留以它们的默认值。通常需要的参数有：

args 被所有调用需要，应当为一个字符串，或者一个程序参数序列。提供一个参数序列通常更好，它可以更小心地使用参数中的转义字符以及引用（例如允许文件名中的空格）。如果传递一个简单的字符串，则 *shell* 参数必须为*True* （见下文）或者该字符串中将被运行的程序名必须用简单的命名而不指定任何参数。

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are *PIPE*, *DEVNULL*, an existing file descriptor (a positive integer), an existing file object with a valid file descriptor, and None. *PIPE* indicates that a new pipe to the child should be created. *DEVNULL* indicates that the special file *os.devnull* will be used. With the default settings of None, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the child process should be captured into the same file handle as for *stdout*.

如果 *encoding* 或 *errors* 被指定，或者 *text* （也名为 *universal_newlines*）为真，则文件对象 *stdin* 、*stdout* 与 *stderr* 将会使用在此次调用中指定的 *encoding* 和 *errors* 以文本模式打开或者为默认的*io.TextIOWrapper*。

当构造函数的 *newline* 参数为 None 时。对于 *stdin*，输入的换行符 '\n' 将被转换为默认的换行符*os.linesep*。对于 *stdout* 和 *stderr*，所有输出的换行符都被转换为 '\n'。更多信息，查看*io.TextIOWrapper* 类的文档。

如果文本模式未被使用，*stdin*，*stdout* 和 *stderr* 将会以二进制流模式打开。没有编码与换行符转换发生。

3.6 版新加入: 添加了 *encoding* 和 *errors* 形参。

3.7 版新加入: 添加了 *text* 形参作为 *universal_newlines* 的别名。

備🄴: 文件对象*Popen.stdin* 、*Popen.stdout* 和*Popen.stderr* 的换行符属性不会被*Popen.communicate()* 方法更新。

如果 *shell* 设为 True,，则使用 shell 执行指定的指令。如果您主要使用 Python 增强的控制流（它比大多数系统 shell 提供的强大），并且仍然希望方便地使用其他 shell 功能，如 shell 管道、文

件通配符、环境变量展开以及 ~ 展开到用户家目录，这将非常有用。但是，注意 Python 自己也实现了许多类似 shell 的特性（例如*glob*, *fnmatch*, *os.walk()*, *os.path.expandvars()*, *os.path.expanduser()* 和*shutil*）。

3.3 版更變: 当 *universal_newline* 被设为 `True`，则类使用*locale.getpreferredencoding(False)* 编码来代替 `locale.getpreferredencoding()`。关于它们的区别的更多信息，见*io.TextIOWrapper*。

備𝔽: 在使用 `shell=True` 之前，请阅读*Security Considerations* 段落。

这些选项以及所有其他选项在*Popen* 构造函数文档中有更详细的描述。

Popen 构造函数

此模块的底层的进程创建与管理由*Popen* 类处理。它提供了很大的灵活性，因此开发者能够处理未被便利函数覆盖的不常见用例。

class subprocess.**Popen**(*args*, *bufsize=-1*, *executable=None*, *stdin=None*, *stdout=None*, *stderr=None*, *preexec_fn=None*, *close_fds=True*, *shell=False*, *cwd=None*, *env=None*, *universal_newlines=None*, *startupinfo=None*, *creationflags=0*, *restore_signals=True*, *start_new_session=False*, *pass_fds=()*, ***, *group=None*, *extra_groups=None*, *user=None*, *umask=-1*, *encoding=None*, *errors=None*, *text=None*)

在一个新的进程中执行子程序。在 POSIX，此类使用类似于*os.execvp()* 的行为来执行子程序。在 Windows，此类使用了 Windows `CreateProcess()` 函数。*Popen* 的参数如下：

args 应当是一个程序参数的序列或者是一个单独的字符串或*path-like object*。默认情况下，如果 *args* 是序列则要运行的程序为 *args* 中的第一项。如果 *args* 是字符串，则其解读依赖于具体平台，如下所述。请查看 *shell* 和 *executable* 参数了解其与默认行为的其他差异。除非另有说明，否则推荐以序列形式传入 *args*。

向外部函数传入序列形式参数的一个例子如下：

```
Popen(["/usr/bin/git", "commit", "-m", "Fixes a bug."])
```

在 POSIX，如果 *args* 是一个字符串，此字符串被作为将被执行的程序的命名或路径解释。但是，只有在不传递任何参数给程序的情况下才能这么做。

備𝔽: 将 shell 命令拆分为参数序列的方式可能并不很直观，特别是在复杂的情况下。*shlex.split()* 可以演示如何确定 *args* 适当的拆分形式：

```
>>> import shlex, subprocess
>>> command_line = input()
/bin/vikings -input eggs.txt -output "spam spam.txt" -cmd "echo '$MONEY'"
>>> args = shlex.split(command_line)
>>> print(args)
['/bin/vikings', '-input', 'eggs.txt', '-output', 'spam spam.txt', '-cmd', "echo '
↪$MONEY'"]
>>> p = subprocess.Popen(args) # Success!
```

特别注意，由 shell 中的空格分隔的选项（例如 *-input*）和参数（例如 *eggs.txt* ）位于分开的列表元素中，而在需要时使用引号或反斜杠转义的参数在 shell （例如包含空格的文件名或上面显示的 *echo* 命令）是单独的列表元素。

在 Windows，如果 *args* 是一个序列，他将通过一个在*在 Windows 上将参数列表转换为一个字符串* 描述的方式被转换为一个字符串。这是因为底层的 `CreateProcess()` 只处理字符串。

3.6 版更變: 在 POSIX 上如果 *shell* 为 `False` 并且序列包含路径类对象则 *args* 形参可以接受一个*path-like object*。

3.8 版更變: 如果在 Windows 上 *shell* 为 `False` 并且序列包含字节串和路径类对象则 *args* 形参可以接受一个*path-like object*。

参数 *shell*（默认为 `False`）指定是否使用 shell 执行程序。如果 *shell* 为 `True`，更推荐将 *args* 作为字符串传递而非序列。

在 POSIX，当 `shell=True`，shell 默认为 `/bin/sh`。如果 *args* 是一个字符串，此字符串指定将通过 shell 执行的命令。这意味着字符串的格式必须和在命令提示符中所输入的完全相同。这包括，例如，引号和反斜杠转义包含空格的文件名。如果 *args* 是一个序列，第一项指定了命令，另外的项目将作为传递给 shell （而非命令）的参数对待。也就是说，*Popen* 等同于:

```
Popen(['/bin/sh', '-c', args[0], args[1], ...])
```

在 Windows，使用 `shell=True`，环境变量 `COMSPEC` 指定了默认 shell。在 Windows 你唯一需要指定 `shell=True` 的情况是你想要执行内置在 shell 中的命令（例如 **`dir`** 或者 **`copy`**）。在运行一个批处理文件或者基于控制台的可执行文件时，不需要 `shell=True`。

備🄵: 在使用 `shell=True` 之前，请阅读*Security Considerations* 段落。

---

bufsize 将在*open()* 函数创建了 stdin/stdout/stderr 管道文件对象时作为对应的参数供应:

- `0` 表示不使用缓冲区（读取与写入是一个系统调用并且可以返回短内容）
- `1` 表示行缓冲（只有 `universal_newlines=True` 时才有用，例如，在文本模式中）
- 任何其他正值表示使用一个约为对应大小的缓冲区
- 负的 *bufsize* （默认）表示使用系统默认的 io.DEFAULT_BUFFER_SIZE。

3.3.1 版更變: *bufsize* 现在默认为 -1 来启用缓冲，以符合大多数代码所期望的行为。在 Python 3.2.4 和 3.3.1 之前的版本中，它错误地将默认值设为了为 `0`，这是无缓冲的并且允许短读取。这是无意的，并且与大多数代码所期望的 Python 2 的行为不一致。

executable 参数指定一个要执行的替换程序。这很少需要。当 `shell=True`，*executable* 替换 *args* 指定运行的程序。但是，原始的 *args* 仍然被传递给程序。大多数程序将被 *args* 指定的程序作为命令名对待，这可以与实际运行的程序不同。在 POSIX，*args* 名作为实际调用程序中可执行文件的显示名称，例如 **`ps`**。如果 `shell=True`，在 POSIX，*executable* 参数指定用于替换默认 shell `/bin/sh` 的 shell。

3.6 版更變: 在 POSIX 上 *executable* 形参可以接受一个*path-like object*。

3.8 版更變: 在 Windows 上 *executable* 形参可以接受一个字节串和*path-like object*。

stdin, *stdout* and *stderr* specify the executed program's standard input, standard output and standard error file handles, respectively. Valid values are *PIPE*, *DEVNULL*, an existing file descriptor (a positive integer), an existing *file object* with a valid file descriptor, and `None`. *PIPE* indicates that a new pipe to the child should be created. *DEVNULL* indicates that the special file *os.devnull* will be used. With the default settings of `None`, no redirection will occur; the child's file handles will be inherited from the parent. Additionally, *stderr* can be *STDOUT*, which indicates that the stderr data from the applications should be captured into the same file handle as for stdout.

如果 *preexec_fn* 被设为一个可调用对象，此对象将在子进程刚创建时被调用。（仅 POSIX）

> **警告:** *preexec_fn* 形参在应用程序中存在多线程时是不安全的。子进程在调用前可能死锁。如果你必须使用它，保持警惕！最小化你调用的库的数量。

備𝔽: 如果你需要修改子进程环境，使用 *env* 形参而非在 *preexec_fn* 中进行。*start_new_session* 形参可以代替之前常用的 *preexec_fn* 来在子进程中调用 os.setsid()。

3.8 版更變: *preexec_fn* 形参在子解释器中已不再受支持。在子解释器中使用此形参将引发`RuntimeError`。这个新限制可能会影响部署在 mod_wsgi, uWSGI 和其他嵌入式环境中的应用。

如果 *close_fds* 为真，所有文件描述符除了 `0`, `1`, `2` 之外都会在子进程执行前关闭。而当 *close_fds* 为假时，文件描述符遵守它们继承的标志，如文件描述符的继承 所述。

在 Windows，如果 *close_fds* 为真，则子进程不会继承任何句柄，除非在 `STARTUPINFO.lpAttributeList` 的 `handle_list` 的键中显式传递，或者通过标准句柄重定向传递。

3.2 版更變: *close_fds* 的默认值已经从`False`修改为上述值。

3.7 版更變: 在 Windows，当重定向标准句柄时 *close_fds* 的默认值从`False`变为`True`。现在重定向标准句柄时有可能设置 *close_fds* 为`True`。（标准句柄指三个 stdio 的句柄）

pass_fds 是一个可选的在父子进程间保持打开的文件描述符序列。提供任何 *pass_fds* 将强制 *close_fds* 为`True`。（仅 POSIX）

3.2 版更變: 加入了 *pass_fds* 形参。

如果 *cwd* 不为 `None`，此函数在执行子进程前会将当前工作目录改为 *cwd*。*cwd* 可以是一个字符串、字节串或路径类对象 。特别地，当可执行文件的路径为相对路径时，此函数会相对于 *cwd* 来查找 *executable* (或 *args* 中的第一个条目)。

3.6 版更變: 在 POSIX 上 *cwd* 形参接受一个*path-like object*。

3.7 版更變: 在 Windows 上 *cwd* 形参接受一个*path-like object*。

3.8 版更變: 在 Windows 上 *cwd* 形参接受一个字节串对象。

如果 *restore_signals* 为 true（默认值），则 Python 设置为 SIG_IGN 的所有信号将在 exec 之前的子进程中恢复为 SIG_DFL。目前，这包括 SIGPIPE，SIGXFZ 和 SIGXFSZ 信号。（仅 POSIX）

3.2 版更變: *restore_signals* 被加入。

如果 *start_new_session* 为 true，则 setsid() 系统调用将在子进程执行之前被执行。（仅 POSIX）

3.2 版更變: *start_new_session* 被添加。

如果 *group* 不为 `None`，则 setregid() 系统调用将于子进程执行之前在下级进程中进行。如果所提供的值为一个字符串，将通过`grp.getgrnam()`来查找它，并将使用 `gr_gid` 中的值。如果该值为一个整数，它将被原样传递。（POSIX 专属）

可用性: POSIX

3.9 版新加入.

如果 *extra_groups* 不为 `None`，则 setgroups() 系统调用将于子进程之前在下级进程中进行。在 *extra_groups* 中提供的字符串将通过`grp.getgrnam()`来查找，并将使用 `gr_gid` 中的值。整数值将被原样传递。（POSIX 专属）

可用性: POSIX

3.9 版新加入.

如果 *user* 不为 `None`，则 setreuid() 系统调用将于子进程执行之前在下级进程中进行。如果所提供的值为一个字符串，将通过`pwd.getpwnam()`来查找它，并将使用 `pw_uid` 中的值。如果该值为一个整数，它将被原样传递。（POSIX 专属）

可用性: POSIX

3.9 版新加入.

如果 *umask* 不为负值，则 umask() 系统调用将在子进程执行之前在下级进程中进行。

可用性: POSIX

3.9 版新加入.

如果 *env* 不为 `None`，则必须为一个为新进程定义了环境变量的字典；这些用于替换继承的当前进程环境的默认行为。

備𝔽: 如果指定，*env* 必须提供所有被子进程需求的变量。在 Windows，为了运行一个 side-by-side assembly ，指定的 *env* **必须**包含一个有效的 `SystemRoot`。

如果 *encoding* 或 *errors* 被指定，或者 *text* 为 true，则文件对象 *stdin*, *stdout* 和 *stderr* 将会以指定的编码和 *errors* 以文本模式打开，如同常用参数 所述。*universal_newlines* 参数等同于 *text* 并且提供向后兼容性。默认情况下，文件对象都以二进制模式打开。

3.6 版新加入: *encoding* 和 *errors* 被添加。

3.7 版新加入: *text* 作为 *universal_newlines* 的一个更具可读性的别名被添加。

如果给出，*startupinfo* 将是一个将被传递给底层的 `CreateProcess` 函数的*STARTUPINFO* 对象。*creationflags*，如果给出，可以是一个或多个以下标志之一:

- *CREATE_NEW_CONSOLE*
- *CREATE_NEW_PROCESS_GROUP*
- *ABOVE_NORMAL_PRIORITY_CLASS*
- *BELOW_NORMAL_PRIORITY_CLASS*
- *HIGH_PRIORITY_CLASS*
- *IDLE_PRIORITY_CLASS*
- *NORMAL_PRIORITY_CLASS*
- *REALTIME_PRIORITY_CLASS*
- *CREATE_NO_WINDOW*
- *DETACHED_PROCESS*
- *CREATE_DEFAULT_ERROR_MODE*
- *CREATE_BREAKAWAY_FROM_JOB*

Popen 对象支持通过 `with` 语句作为上下文管理器，在退出时关闭文件描述符并等待进程:

```
with Popen(["ifconfig"], stdout=PIPE) as proc:
    log.write(proc.stdout.read())
```

引发一个审计事件 `subprocess.Popen`，附带参数 `executable`, `args`, `cwd`, `env`。

3.2 版更變: 添加了上下文管理器支持。

3.6 版更變: 现在，如果 Popen 析构时子进程仍然在运行，则析构器会发送一个*ResourceWarning* 警告。

3.8 版更變: 在某些情况下 Popen 可以使用*os.posix_spawn()* 以获得更好的性能。在适用于 Linux 的 Windows 子系统和 QEMU 用户模拟器上，使用*os.posix_spawn()* 的 Popen 构造器不再会因找不到程序等错误而引发异常，而是上下级进程失败并返回一个非零的*returncode*。

异常

在子进程中抛出的异常，在新的进程开始执行前，将会被再次在父进程中抛出。

被引发的最一般异常是*OSError*。例如这会在尝试执行一个不存在的文件时发生。应用程序应当为*OSError*异常做好准备。请注意，如果 shell=True，则*OSError* 仅会在未找到选定的 shell 本身时被引发。要确定 shell 是否未找到所请求的应用程序，必须检查来自子进程的返回码或输出。

如果*Popen* 调用时有无效的参数，则一个*ValueError* 将被抛出。

check_call() 与*check_output()* 在调用的进程返回非零退出码时将抛出*CalledProcessError*。

所有接受 *timeout* 形参的函数与方法，例如*call()* 和*Popen.communicate()* 将会在进程退出前超时到期时抛出*TimeoutExpired*。

此模块中定义的异常都继承自*SubprocessError*。

3.3 版新加入: 基类*SubprocessError* 被添加。

17.6.2 安全考量

不像一些其他的 popen 功能，此实现绝不会隐式调用一个系统 shell。这意味着任何字符，包括 shell 元字符，可以安全地被传递给子进程。如果 shell 被明确地调用，通过 shell=True 设置，则确保所有空白字符和元字符被恰当地包裹在引号内以避免 shell 注入 漏洞就由应用程序负责了。

当使用 shell=True，*shlex.quote()* 函数可以作为在将被用于构造 shell 指令的字符串中转义空白字符以及 shell 元字符的方案。

17.6.3 Popen 对象

Popen 类的实例拥有以下方法：

Popen.**poll**()
检查子进程是否已被终止。设置并返回*returncode* 属性。否则返回 None。

Popen.**wait**(*timeout=None*)
等待子进程被终止。设置并返回*returncode* 属性。

如果进程在 *timeout* 秒后未中断，抛出一个*TimeoutExpired* 异常，可以安全地捕获此异常并重新等待。

備🄴: 当 stdout=PIPE 或者 stderr=PIPE 并且子进程产生了足以阻塞 OS 管道缓冲区接收更多数据的输出到管道时，将会发生死锁。当使用管道时用*Popen.communicate()* 来规避它。

備🄴: 此函数使用了一个 busy loop （非阻塞调用以及短睡眠）实现。使用*asyncio* 模块进行异步等待：参阅*asyncio.create_subprocess_exec*。

3.3 版更變: *timeout* 被添加

Popen.**communicate**(*input=None*, *timeout=None*)
与进程交互：将数据发送到 stdin。从 stdout 和 stderr 读取数据，直到抵达文件结尾。等待进程终止并设置*returncode* 属性。可选的 *input* 参数应为要发送到下级进程的数据，或者如果没有要发送到下级进程的数据则为 None。如果流是以文本模式打开的，则 *input* 必须为字符串。在其他情况下，它必须为字节串。

communicate() 返回一个 (stdout_data, stderr_data) 元组。如果文件以文本模式打开则为字符串；否则字节。

注意如果你想要向进程的 stdin 传输数据，你需要通过 stdin=PIPE 创建此 Popen 对象。类似的，要从结果元组获取任何非 None 值，你同样需要设置 stdout=PIPE 或者 stderr=PIPE。

如果进程在 *timeout* 秒后未终止，一个*TimeoutExpired* 异常将被抛出。捕获此异常并重新等待将不会丢失任何输出。

如果超时到期，子进程不会被杀死，所以为了正确清理一个行为良好的应用程序应该杀死子进程并完成通讯。

```
proc = subprocess.Popen(...)
try:
    outs, errs = proc.communicate(timeout=15)
except TimeoutExpired:
    proc.kill()
    outs, errs = proc.communicate()
```

備🄵: 内存里数据读取是缓冲的，所以如果数据尺寸过大或无限，不要使用此方法。

3.3 版更變: *timeout* 被添加

Popen.**send_signal**(*signal*)
将信号 *signal* 发送给子进程。

如果进程已完成则不做任何操作。

備 🄵: 在 Windows，SIGTERM 是一个*terminate()* 的别名。CTRL_C_EVENT 和 CTRL_BREAK_EVENT 可以被发送给以包含 CREATE_NEW_PROCESS 的 *creationflags* 形参启动的进程。

Popen.**terminate**()
停止子进程。在 POSIX 操作系统上，此方法会发送 SIGTERM 给子进程。在 Windows 上则会调用 Win32 API 函数 TerminateProcess() 来停止子进程。

Popen.**kill**()
杀死子进程。在 POSIX 操作系统上，此函数会发送 SIGKILL 给子进程。在 Windows 上*kill()* 则是*terminate()* 的别名。

以下属性也是可用的：

Popen.**args**
args 参数传递给*Popen* -- 一个程序参数的序列或者一个简单字符串。

3.3 版新加入.

Popen.**stdin**
如果 *stdin* 参数为*PIPE*，此属性是一个类似*open()* 返回的可写的流对象。如果 *encoding* 或 *errors* 参数被指定或者 *universal_newlines* 参数为 True，则此流是一个文本流，否则是字节流。如果 *stdin* 参数非*PIPE*，此属性为 None。

Popen.**stdout**
如果 *stdout* 参数是*PIPE*，此属性是一个类似*open()* 返回的可读流。从流中读取子进程提供的输出。如果 *encoding* 或 *errors* 参数被指定或者 *universal_newlines* 参数为 True，此流为文本流，否则为字节流。如果 *stdout* 参数非*PIPE*，此属性为 None。

Popen.**stderr**
如果 *stderr* 参数是*PIPE*，此属性是一个类似*open()* 返回的可读流。从流中读取子进程提供的输出。如果 *encoding* 或 *errors* 参数被指定或者 *universal_newlines* 参数为 True，此流为文本流，否则为字节流。如果 *stderr* 参数非*PIPE*，此属性为 None。

警告: 使用*communicate()* 而非*.stdin.write*，*.stdout.read* 或者*.stderr.read* 来避免由于任意其他 OS 管道缓冲区被子进程填满阻塞而导致的死锁。

Popen.**pid**
子进程的进程号。

注意如果你设置了 *shell* 参数为 True，则这是生成的子 shell 的进程号。

Popen.**returncode**
此进程的退出码，由*poll()* 和*wait()* 设置（以及直接由*communicate()* 设置）。一个 None 值表示此进程仍未结束。

一个负值 -N 表示子进程被信号 N 中断 (仅 POSIX).

17.6.4 Windows Popen 助手

STARTUPINFO 类和以下常数仅在 Windows 有效。

class subprocess.**STARTUPINFO**(*, *dwFlags=0*, *hStdInput=None*, *hStdOutput=None*, *hStdError=None*, *wShowWindow=0*, *lpAttributeList=None*)
在*Popen* 创建时部分支持 Windows 的 STARTUPINFO 结构。接下来的属性仅能通过关键词参数设置。

3.7 版更變: 仅关键词参数支持被加入。

dwFlags
一个位域，用于确定属性*STARTUPINFO* 是否在进程创建窗口时使用。

```
si = subprocess.STARTUPINFO()
si.dwFlags = subprocess.STARTF_USESTDHANDLES | subprocess.STARTF_USESHOWWINDOW
```

hStdInput
如果*dwFlags* 被指定为*STARTF_USESTDHANDLES*，此值是进程的标准输入句柄，如果*STARTF_USESTDHANDLES* 未指定，则默认的标准输入是键盘缓冲区。

hStdOutput
如果*dwFlags* 指定为*STARTF_USESTDHANDLES*，此属性是进程的标准输出句柄。除此之外，此此属性将被忽略并且默认标准输出是控制台窗口缓冲区。

hStdError
如果*dwFlags* 被指定为*STARTF_USESTDHANDLES*，则此属性是进程的标准错误句柄。除此之外，此属性将被忽略并且默认标准错误为控制台窗口的缓冲区。

wShowWindow
如果*dwFlags* 指定了*STARTF_USESHOWWINDOW*，此属性可为能被指定为函数 ShowWindow 的 nCmdShow 的形参的任意值，除了 SW_SHOWDEFAULT。如此之外，此属性被忽略。

SW_HIDE 被提供给此属性。它在*Popen* 由 shell=True 调用时使用。

lpAttributeList
STARTUPINFOEX 给出的用于进程创建的额外属性字典，参阅 UpdateProcThreadAttribute。

支持的属性：

handle_list 将被继承的句柄的序列。如果非空，*close_fds* 必须为 true。

当传递给*Popen* 构造函数时，这些句柄必须暂时地能被*os.set_handle_inheritable()* 继承，否则*OSError* 将以 Windows error `ERROR_INVALID_PARAMETER` (87) 抛出。

> **警告：** 在多线程进程中，请谨慎使用，以便在将此功能与对继承所有句柄的其他进程创建函数（例如*os.system()*）的并发调用相结合时，避免泄漏标记为可继承的句柄。这也应用于临时性创建可继承句柄的标准句柄重定向。

3.7 版新加入.

Windows 常数

subprocess 模块曝出以下常数。

`subprocess.`**STD_INPUT_HANDLE**
标准输入设备，这是控制台输入缓冲区 `CONIN$`。

`subprocess.`**STD_OUTPUT_HANDLE**
标准输出设备。最初，这是活动控制台屏幕缓冲区 `CONOUT$`。

`subprocess.`**STD_ERROR_HANDLE**
标准错误设备。最初，这是活动控制台屏幕缓冲区 `CONOUT$`。

`subprocess.`**SW_HIDE**
隐藏窗口。另一个窗口将被激活。

`subprocess.`**STARTF_USESTDHANDLES**
指明*STARTUPINFO.hStdInput*, *STARTUPINFO.hStdOutput* 和*STARTUPINFO.hStdError* 属性包含额外的信息。

`subprocess.`**STARTF_USESHOWWINDOW**
指明*STARTUPINFO.wShowWindow* 属性包含额外的信息。

`subprocess.`**CREATE_NEW_CONSOLE**
新的进程将有新的控制台，而不是继承父进程的（默认）控制台。

`subprocess.`**CREATE_NEW_PROCESS_GROUP**
用于指明将创建一个新的进程组的*Popen* `creationflags` 形参。这个旗标对于在子进程上使用*os.kill()* 来说是必须的。

如果指定了*CREATE_NEW_CONSOLE* 则这个旗标会被忽略。

`subprocess.`**ABOVE_NORMAL_PRIORITY_CLASS**
用于指明一个新进程将具有高于平均的优先级的*Popen* `creationflags` 形参。

3.7 版新加入.

`subprocess.`**BELOW_NORMAL_PRIORITY_CLASS**
用于指明一个新进程将具有低于平均的优先级的*Popen* `creationflags` 形参。

3.7 版新加入.

`subprocess.`**HIGH_PRIORITY_CLASS**
用于指明一个新进程将具有高优先级的*Popen* `creationflags` 形参。

3.7 版新加入.

`subprocess.`**IDLE_PRIORITY_CLASS**
用于指明一个新进程将具有空闲（最低）优先级的*Popen* `creationflags` 形参。

3.7 版新加入.

subprocess.**NORMAL_PRIORITY_CLASS**
用于指明一个新进程将具有正常（默认）优先级的*Popen* creationflags 形参。

3.7 版新加入.

subprocess.**REALTIME_PRIORITY_CLASS**
用于指明一个新进程将具有实时优先级的*Popen* creationflags 形参。你应当几乎永远不使用 REALTIME_PRIORITY_CLASS，因为这会中断管理鼠标输入、键盘输入以及后台磁盘刷新的系统线程。这个类只适用于直接与硬件“对话”，或者执行短暂任务具有受限中断的应用。

3.7 版新加入.

subprocess.**CREATE_NO_WINDOW**
指明一个新进程将不会创建窗口的*Popen* creationflags 形参。

3.7 版新加入.

subprocess.**DETACHED_PROCESS**
指明一个新进程将不会继承其父控制台的*Popen* creationflags 形参。这个值不能与 CREATE_NEW_CONSOLE 一同使用。

3.7 版新加入.

subprocess.**CREATE_DEFAULT_ERROR_MODE**
指明一个新进程不会继承调用方进程的错误模式的*Popen* creationflags 形参。新进程会转为采用默认的错误模式。这个特性特别适用于运行时禁用硬错误的多线程 shell 应用。

3.7 版新加入.

subprocess.**CREATE_BREAKAWAY_FROM_JOB**
指明一个新进程不会关联到任务的*Popen* creationflags 形参。

3.7 版新加入.

17.6.5 较旧的高阶 API

在 Python 3.5 之前，这三个函数组成了 subprocess 的高阶 API。现在你可以在许多情况下使用*run()*，但有大量现在代码仍会调用这些函数。

subprocess.**call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)
运行由 *args* 所描述的命令。等待命令完成，然后返回*returncode* 属性。

需要捕获 stdout 或 stderr 的代码应当改用*run()*:

```
run(...).returncode
```

要屏蔽 stdout 或 stderr，可提供*DEVNULL* 这个值。

上面显示的参数只是常见的一些。完整的函数签名与*Popen* 构造器的相同——此函数会将所提供的 *timeout* 之外的全部参数直接传递给目标接口。

備🄵: 请不要在此函数中使用 stdout=PIPE 或 stderr=PIPE。如果子进程向管道生成了足以填满 OS 管理缓冲区的输出而管道还未被读取时它将会阻塞。

3.3 版更變: *timeout* 被添加

subprocess.**check_call**(*args*, *, *stdin=None*, *stdout=None*, *stderr=None*, *shell=False*, *cwd=None*, *timeout=None*, ***other_popen_kwargs*)

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise *CalledProcessError*. The *CalledProcessError* object will have the return code in the *returncode* attribute. If *check_call()* was unable to start the process it will propagate the exception that was raised.

需要捕获 stdout 或 stderr 的代码应当改用*run()*:

```
run(..., check=True)
```

要屏蔽 stdout 或 stderr，可提供*DEVNULL* 这个值。

上面显示的参数只是常见的一些。完整的函数签名与*Popen* 构造器的相同——此函数会将所提供的 *timeout* 之外的全部参数直接传递给目标接口。

備𝔽: 请不要在此函数中使用 `stdout=PIPE` 或 `stderr=PIPE`。如果子进程向管道生成了足以填满 OS 管理缓冲区的输出而管道还未被读取时它将会阻塞。

3.3 版更變: *timeout* 被添加

subprocess.**check_output**(*args*, *, *stdin=None*, *stderr=None*, *shell=False*, *cwd=None*, *encoding=None*, *errors=None*, *universal_newlines=None*, *timeout=None*, *text=None*, ***other_popen_kwargs*)

附带参数运行命令并返回其输出。

如果返回码非零则会引发*CalledProcessError*。*CalledProcessError* 对象将在*returncode* 属性中保存返回码并在*output* 属性中保存所有输出。

这相当于:

```
run(..., check=True, stdout=PIPE).stdout
```

上面显示的参数只是常见的一些。完整的函数签名与*run()* 的大致相同——大部分参数会通过该接口直接传递。存在一个与*run()* 行为不同的 API 差异：传递 `input=None` 的行为将与 `input=b''` (或 `input=''`，具体取决于其他参数) 一样而不是使用父对象的标准输入文件处理。

默认情况下，此函数将把数据返回为已编码的字节串。输出数据的实际编码格式将取决于发起调用的命令，因此解码为文本的操作往往需要在应用程序层级上进行处理。

此行为可以通过设置 *text*, *encoding*, *errors* 或将 *universal_newlines* 设为 `True` 来重载，具体描述见常用参数 和*run()*。

要在结果中同时捕获标准错误，请使用 `stderr=subprocess.STDOUT`:

```
>>> subprocess.check_output(
...     "ls non_existent_file; exit 0",
...     stderr=subprocess.STDOUT,
...     shell=True)
'ls: non_existent_file: No such file or directory\n'
```

3.1 版新加入.

3.3 版更變: *timeout* 被添加

3.4 版更變: 增加了对 *input* 关键字参数的支持。

3.6 版更變: 增加了 *encoding* 和 *errors*。详情参见*run()*。

3.7 版新加入: *text* 作为 *universal_newlines* 的一个更具可读性的别名被添加。

17.6.6 使用 subprocess 模块替换旧函数

在这一节中，”a 改为 b” 意味着 b 可以被用作 a 的替代。

備𝔽: 在这一节中的所有”a” 函数会在找不到被执行的程序时（差不多）静默地失败；”b” 替代函数则会改为引发*OSError*。

此外，在使用*check_output()* 时如果替代函数所请求的操作产生了非零返回值则将失败并引发*CalledProcessError*。操作的输出仍能以所引发异常的*output* 属性的方式被访问。

在下列例子中，我们假定相关的函数都已从*subprocess* 模块中导入了。

替代 /bin/sh shell 命令替换

```
output=$(mycmd myarg)
```

改为:

```
output = check_output(["mycmd", "myarg"])
```

替代 shell 管道

```
output=$(dmesg | grep hda)
```

改为:

```
p1 = Popen(["dmesg"], stdout=PIPE)
p2 = Popen(["grep", "hda"], stdin=p1.stdout, stdout=PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
```

启动 p2 之后再执行 `p1.stdout.close()` 调用很重要，这是为了让 p1 能在 p2 先于 p1 退出时接收到 SIGPIPE。

另外，对于受信任的输入，shell 本身的管道支持仍然可被直接使用:

```
output=$(dmesg | grep hda)
```

改为:

```
output=check_output("dmesg | grep hda", shell=True)
```

替代 os.system()

```
sts = os.system("mycmd" + " myarg")
# becomes
retcode = call("mycmd" + " myarg", shell=True)
```

𝔽解:

- 通过 shell 来调用程序通常是不必要的。

- *call()* 返回值的编码方式与*os.system()* 的不同。
- *os.system()* 函数在命令运行期间会忽略 SIGINT 和 SIGQUIT 信号，但调用方必须在使用*subprocess* 模块时分别执行此操作。

一个更现实的例子如下所示:

```
try:
    retcode = call("mycmd" + " myarg", shell=True)
    if retcode < 0:
        print("Child was terminated by signal", -retcode, file=sys.stderr)
    else:
        print("Child returned", retcode, file=sys.stderr)
except OSError as e:
    print("Execution failed:", e, file=sys.stderr)
```

替代 os.spawn 函数族

P_NOWAIT 示例:

```
pid = os.spawnlp(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg")
==>
pid = Popen(["/bin/mycmd", "myarg"]).pid
```

P_WAIT 示例:

```
retcode = os.spawnlp(os.P_WAIT, "/bin/mycmd", "mycmd", "myarg")
==>
retcode = call(["/bin/mycmd", "myarg"])
```

Vector 示例:

```
os.spawnvp(os.P_NOWAIT, path, args)
==>
Popen([path] + args[1:])
```

Environment 示例:

```
os.spawnlpe(os.P_NOWAIT, "/bin/mycmd", "mycmd", "myarg", env)
==>
Popen(["/bin/mycmd", "myarg"], env={"PATH": "/usr/bin"})
```

替代 os.popen(), os.popen2(), os.popen3()

```
(child_stdin, child_stdout) = os.popen2(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdin, child_stdout) = (p.stdin, p.stdout)
```

```
(child_stdin,
 child_stdout,
 child_stderr) = os.popen3(cmd, mode, bufsize)
==>
```

(下页继续)

(繼續上一頁)

```
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=PIPE, close_fds=True)
(child_stdin,
 child_stdout,
 child_stderr) = (p.stdin, p.stdout, p.stderr)
```

```
(child_stdin, child_stdout_and_stderr) = os.popen4(cmd, mode, bufsize)
==>
p = Popen(cmd, shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, stderr=STDOUT, close_fds=True)
(child_stdin, child_stdout_and_stderr) = (p.stdin, p.stdout)
```

返回码以如下方式处理转写:

```
pipe = os.popen(cmd, 'w')
...
rc = pipe.close()
if rc is not None and rc >> 8:
    print("There were some errors")
==>
process = Popen(cmd, stdin=PIPE)
...
process.stdin.close()
if process.wait() != 0:
    print("There were some errors")
```

来自 popen2 模块的替代函数

備𝔽: 如果 popen2 函数的 cmd 参数是一个字符串，命令会通过 /bin/sh 来执行。如果是一个列表，命令会被直接执行。

```
(child_stdout, child_stdin) = popen2.popen2("somestring", bufsize, mode)
==>
p = Popen("somestring", shell=True, bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

```
(child_stdout, child_stdin) = popen2.popen2(["mycmd", "myarg"], bufsize, mode)
==>
p = Popen(["mycmd", "myarg"], bufsize=bufsize,
          stdin=PIPE, stdout=PIPE, close_fds=True)
(child_stdout, child_stdin) = (p.stdout, p.stdin)
```

`popen2.Popen3` 和 `popen2.Popen4` 基本上类似于*`subprocess.Popen`*，不同之处在于:

- *`Popen`* 如果执行失败会引发一个异常。
- *capturestderr* 参数被替换为 *stderr* 参数。
- 必须指定 `stdin=PIPE` 和 `stdout=PIPE`。
- popen2 默认会关闭所有文件描述符，但对于*`Popen`* 你必须指明 `close_fds=True` 以才能在所有平台或较旧的 Python 版本中确保此行为。

17.6.7 旧式的 Shell 发起函数

此模块还提供了以下来自 2.x `commands` 模块的旧版函数。这些操作会隐式地发起调用系统 shell 并且上文所描述的有关安全与异常处理一致性保证都不适用于这些函数。

`subprocess.`**`getstatusoutput`**(*cmd*)
返回在 shell 中执行 *cmd* 产生的 `(exitcode, output)`。

在 shell 中以 `Popen.check_output()` 执行字符串 *cmd* 并返回一个 2 元组 `(exitcode, output)`。会使用当前区域设置的编码格式；请参阅*常用参数* 中的说明来了解详情。

末尾的一个换行符会从输出中被去除。命令的退出码可被解读为子进程的返回码。例如:

```
>>> subprocess.getstatusoutput('ls /bin/ls')
(0, '/bin/ls')
>>> subprocess.getstatusoutput('cat /bin/junk')
(1, 'cat: /bin/junk: No such file or directory')
>>> subprocess.getstatusoutput('/bin/junk')
(127, 'sh: /bin/junk: not found')
>>> subprocess.getstatusoutput('/bin/kill $$')
(-15, '')
```

可用性: POSIX 和 Windows。

3.3.4 版更變: 添加了 Windows 支持。

此函数现在返回 (exitcode, output) 而不是像 Python 3.3.3 及更早的版本那样返回 (status, output)。exitcode 的值与*`returncode`* 相同。

`subprocess.`**`getoutput`**(*cmd*)
返回在 shell 中执行 *cmd* 产生的输出（stdout 和 stderr）。

类似于*`getstatusoutput()`*，但退出码会被忽略并且返回值为包含命令输出的字符串。例如:

```
>>> subprocess.getoutput('ls /bin/ls')
'/bin/ls'
```

可用性: POSIX 和 Windows。

3.3.4 版更變: 添加了 Windows 支持

17.6.8 🄵解

在 Windows 上将参数列表转换为一个字符串

在 Windows 上，*args* 序列会被转换为可使用以下规则来解析的字符串（对应于 MS C 运行时所使用的规则）：

1. 参数以空白符分隔，即空格符或制表符。
2. 用双引号标示的字符串会被解读为单个参数，而不再考虑其中的空白符。一个参数可以嵌套用引号标示的字符串。
3. 带有一个反斜杠前缀的双引号会被解读为双引号字面值。
4. 反斜杠会按字面值解读，除非它是作为双引号的前缀。
5. 如果反斜杠被作为双引号的前缀，则每个反斜杠对会被解读为一个反斜杠字面值。如果反斜杠数量为奇数，则最后一个反斜杠会如规则 3 所描述的那样转义下一个双引号。

也参考:

shlex 此模块提供了用于解析和转义命令行的函数。

17.7 sched --- 事件调度器

源码： Lib/sched.py

---

sched 模块定义了一个实现通用事件调度程序的类：

class sched.**scheduler**(*timefunc=time.monotonic*, *delayfunc=time.sleep*)

scheduler 类定义了一个调度事件的通用接口。它需要两个函数来实际处理 “外部世界” ——*timefunc* 应当不带参数地调用，并返回一个数字（“时间”，可以为任意单位）。*delayfunc* 函数应当带一个参数调用，与 *timefunc* 的输出相兼容，并且应当延迟其所指定的时间单位。每个事件运行后还将调用 *delayfunc* 并传入参数 `0` 以允许其他线程有机会在多线程应用中运行。

3.3 版更變: *timefunc* 和 *delayfunc* 参数是可选的。

3.3 版更變: *scheduler* 类可以安全的在多线程环境中使用。

示例：

```
>>> import sched, time
>>> s = sched.scheduler(time.time, time.sleep)
>>> def print_time(a='default'):
...     print("From print_time", time.time(), a)
...
>>> def print_some_times():
...     print(time.time())
...     s.enter(10, 1, print_time)
...     s.enter(5, 2, print_time, argument=('positional',))
...     s.enter(5, 1, print_time, kwargs={'a': 'keyword'})
...     s.run()
...     print(time.time())
...
>>> print_some_times()
930343690.257
From print_time 930343695.274 positional
From print_time 930343695.275 keyword
From print_time 930343700.273 default
930343700.276
```

17.7.1 调度器对象

scheduler 实例拥有以下方法和属性：

scheduler.**enterabs**(*time*, *priority*, *action*, *argument=()*, *kwargs={}*)

安排一个新事件。*time* 参数应该有一个数字类型兼容的返回值，与传递给构造函数的 *timefunc* 函数的返回值兼容。计划在相同 *time* 的事件将按其 *priority* 的顺序执行。数字越小表示优先级越高。

执行事件意为执行 `action(*argument, **kwargs)`。*argument* 是包含有 *action* 的位置参数的序列。*kwargs* 是包含 *action* 的关键字参数的字典。

返回值是一个事件，可用于以后取消事件（参见*cancel()*）。

3.3 版更變: *argument* 参数是可选的。

3.3 版更變: 添加了 *kwargs* 形参。

scheduler.**enter**(*delay*, *priority*, *action*, *argument=()*, *kwargs={}*)
安排延后 *delay* 时间单位的事件。除了相对时间，其他参数、效果和返回值与*enterabs()* 的相同。

3.3 版更變: *argument* 参数是可选的。

3.3 版更變: 添加了 *kwargs* 形参。

scheduler.**cancel**(*event*)
从队列中删除事件。如果 *event* 不是当前队列中的事件，则此方法将引发*ValueError*。

scheduler.**empty**()
如果事件队列为空则返回 True。

scheduler.**run**(*blocking=True*)
运行所有预定事件。此方法将等待（使用传递给构造函数的 delayfunc() 函数）进行下一个事件，然后执行它，依此类推，直到没有更多的计划事件。

如果 *blocking* 为 false，则执行由于最快到期（如果有）的预定事件，然后在调度程序中返回下一个预定调用的截止时间（如果有）。

action 或 *delayfunc* 都可以引发异常。在任何一种情况下，调度程序都将保持一致状态并传播异常。如果 *action* 引发异常，则在将来调用*run()* 时不会尝试该事件。

如果一系列事件的运行时间大于下一个事件发生前的可用时间，那么调度程序将完全落后。没有事件会被丢弃；调用代码负责取消不再相关的事件。

3.3 版更變: 添加了 *blocking* 形参。

scheduler.**queue**
只读属性按照将要运行的顺序返回即将发生的事件列表。每个事件都显示为*named tuple* ，包含以下字段：time、priority、action、argument、kwargs。

17.8 queue --- 一个同步的队列类

源代码: Lib/queue.py

---

queue 模块实现了多生产者、多消费者队列。这特别适用于消息必须安全地在多线程间交换的线程编程。模块中的*Queue* 类实现了所有所需的锁定语义。

模块实现了三种类型的队列，它们的区别仅仅是条目取回的顺序。在 FIFO 队列中，先添加的任务先取回。在 LIFO 队列中，最近被添加的条目先取回 (操作类似一个堆栈)。优先级队列中，条目将保持排序 (使用*heapq* 模块) 并且最小值的条目第一个返回。

在内部，这三个类型的队列使用锁来临时阻塞竞争线程；然而，它们并未被设计用于线程的重入性处理。

此外，模块实现了一个" 简单的" FIFO 队列类型，*SimpleQueue* ，这个特殊实现为小功能在交换中提供额外的保障。

queue 模块定义了下列类和异常：

class queue.**Queue**(*maxsize=0*)
Constructor for a FIFO queue. *maxsize* is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If *maxsize* is less than or equal to zero, the queue size is infinite.

class queue.**LifoQueue**(*maxsize=0*)
LIFO 队列构造函数。*maxsize* 是个整数，用于设置可以放入队列中的项目数的上限。当达到这个大小的时候，插入操作将阻塞至队列中的项目被消费掉。如果 *maxsize* 小于等于零，队列尺寸为无限大。

class queue.**PriorityQueue**(*maxsize=0*)
优先级队列构造函数。*maxsize* 是个整数，用于设置可以放入队列中的项目数的上限。当达到这个大小的时候，插入操作将阻塞至队列中的项目被消费掉。如果 *maxsize* 小于等于零，队列尺寸为无限大。

最小值先被取出 (最小值条目是由 `sorted(list(entries))[0]` 返回的条目)。条目的典型模式是一个以下形式的元组：`(priority_number, data)` 。

如果 *data* 元素没有可比性，数据将被包装在一个类中，忽略数据值，仅仅比较优先级数字：

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

class queue.**SimpleQueue**
无界的 FIFO 队列构造函数。简单的队列，缺少任务跟踪等高级功能。

3.7 版新加入.

exception queue.**Empty**
对空的*Queue* 对象，调用非阻塞的*get()* (or *get_nowait()*) 时，引发的异常。

exception queue.**Full**
对满的*Queue* 对象，调用非阻塞的*put()* (or *put_nowait()*) 时，引发的异常。

17.8.1 Queue 对象

队列对象 (*Queue*, *LifoQueue*, 或者*PriorityQueue*) 提供下列描述的公共方法。

Queue.**qsize**()
返回队列的大致大小。注意，qsize() > 0 不保证后续的 get() 不被阻塞，qsize() < maxsize 也不保证 put() 不被阻塞。

Queue.**empty**()
如果队列为空，返回 `True` ，否则返回 `False` 。如果 empty() 返回 `True` ，不保证后续调用的 put() 不被阻塞。类似的，如果 empty() 返回 `False` ，也不保证后续调用的 get() 不被阻塞。

Queue.**full**()
如果队列是满的返回 `True` ，否则返回 `False` 。如果 full() 返回 `True` 不保证后续调用的 get() 不被阻塞。类似的，如果 full() 返回 `False` 也不保证后续调用的 put() 不被阻塞。

Queue.**put**(*item*, *block=True*, *timeout=None*)
将 *item* 放入队列。如果可选参数 *block* 是 true 并且 *timeout* 是 `None` (默认)，则在必要时阻塞至有空闲插槽可用。如果 *timeout* 是个正数，将最多阻塞 *timeout* 秒，如果在这段时间没有可用的空闲插槽，将引发*Full* 异常。反之 (*block* 是 false)，如果空闲插槽立即可用，则把 *item* 放入队列，否则引发*Full* 异常 (在这种情况下，*timeout* 将被忽略)。

Queue.**put_nowait**(*item*)
Equivalent to `put(item, block=False)`.

Queue.**get**(*block=True*, *timeout=None*)
从队列中移除并返回一个项目。如果可选参数 *block* 是 true 并且 *timeout* 是 `None` (默认值)，则在必要时阻塞至项目可得到。如果 *timeout* 是个正数，将最多阻塞 *timeout* 秒，如果在这段时间内项目不能得到，将引发*Empty* 异常。反之 (*block* 是 false) , 如果一个项目立即可得到，则返回一个项目，否则引发*Empty* 异常 (这种情况下，*timeout* 将被忽略)。

POSIX 系统 3.0 之前，以及所有版本的 Windows 系统中，如果 *block* 是 true 并且 *timeout* 是 None，这个操作将进入基础锁的不间断等待。这意味着，没有异常能发生，尤其是 SIGINT 将不会触发*KeyboardInterrupt* 异常。

Queue.**get_nowait**()
相当于 get(False)。

提供了两个方法，用于支持跟踪排队的任务是否被守护的消费者线程完整的处理。

Queue.**task_done**()
表示前面排队的任务已经被完成。被队列的消费者线程使用。每个*get()* 被用于获取一个任务，后续调用*task_done()* 告诉队列，该任务的处理已经完成。

如果*join()* 当前正在阻塞，在所有条目都被处理后，将解除阻塞(意味着每个*put()* 进队列的条目的*task_done()* 都被收到)。

如果被调用的次数多于放入队列中的项目数量，将引发*ValueError* 异常。

Queue.**join**()
阻塞至队列中所有的元素都被接收和处理完毕。

当条目添加到队列的时候，未完成任务的计数就会增加。每当消费者线程调用*task_done()* 表示这个条目已经被回收，该条目所有工作已经完成，未完成计数就会减少。当未完成计数降到零的时候，*join()* 阻塞被解除。

如何等待排队的任务被完成的示例：

```
import threading, queue

q = queue.Queue()

def worker():
    while True:
        item = q.get()
        print(f'Working on {item}')
        print(f'Finished {item}')
        q.task_done()

# turn-on the worker thread
threading.Thread(target=worker, daemon=True).start()

# send thirty task requests to the worker
for item in range(30):
    q.put(item)
print('All task requests sent\n', end='')

# block until all tasks are done
q.join()
print('All work completed')
```

17.8.2 SimpleQueue 对象

SimpleQueue 对象提供下列描述的公共方法。

SimpleQueue.**qsize**()
返回队列的大致大小。注意，qsize() > 0 不保证后续的 get() 不被阻塞。

SimpleQueue.**empty**()
如果队列为空，返回 True ，否则返回 False 。如果 empty() 返回 False ，不保证后续调用的 get() 不被阻塞。

SimpleQueue.**put**(*item*, *block=True*, *timeout=None*)
将 *item* 放入队列。此方法永不阻塞，始终成功（除了潜在的低级错误，例如内存分配失败）。可选参数 *block* 和 *timeout* 仅仅是为了保持*Queue.put()* 的兼容性而提供，其值被忽略。

CPython implementation detail: This method has a C implementation which is reentrant. That is, a `put()` or `get()` call can be interrupted by another `put()` call in the same thread without deadlocking or corrupting internal state inside the queue. This makes it appropriate for use in destructors such as `__del__` methods or *weakref* callbacks.

SimpleQueue.**put_nowait**(*item*)
Equivalent to `put(item, block=False)`, provided for compatibility with *Queue.put_nowait()*.

SimpleQueue.**get**(*block=True*, *timeout=None*)
从队列中移除并返回一个项目。如果可选参数 *block* 是 true 并且 *timeout* 是 None (默认值)，则在必要时阻塞至项目可得到。如果 *timeout* 是个正数，将最多阻塞 *timeout* 秒，如果在这段时间内项目不能得到，将引发*Empty* 异常。反之 (*block* 是 false) , 如果一个项目立即可得到，则返回一个项目，否则引发*Empty* 异常 (这种情况下，*timeout* 将被忽略)。

SimpleQueue.**get_nowait**()
相当于 `get(False)` 。

也参考:

类***multiprocessing.Queue*** 一个用于多进程上下文的队列类（而不是多线程）。

collections.deque 是无界队列的一个替代实现，具有快速的不需要锁并且支持索引的原子化*append()* 和*popleft()* 操作。

17.9 contextvars --- 上下文变量

本模块提供了相关 API 用于管理、存储和访问上下文相关的状态。*ContextVar* 类用于声明 上下文变量并与其一起使用。函数*copy_context()* 和类*Context* 用于管理当前上下文和异步框架中。

在多并发环境中，有状态上下文管理器应该使用上下文变量，而不是*threading.local()* 来防止他们的状态意外泄露到其他代码。

更多信息参见 **PEP 567** 。

3.7 版新加入.

17.9.1 上下文变量

class contextvars.**ContextVar**(*name*[, *, *default*])
此类用于声明一个新的上下文变量，如:

```
var: ContextVar[int] = ContextVar('var', default=42)
```

name 参数用于内省和调试，必需。

调用*ContextVar.get()* 时，如果上下文中没有找到此变量的值，则返回可选的仅命名参数 *default* 。

重要： 上下文变量应该在顶级模块中创建，且永远不要在闭包中创建。*Context* 对象拥有对上下文变量的强引用，这可以让上下文变量被垃圾收集器正确回收。

name
上下文变量的名称，只读属性。

3.7.1 版新加入.

get([*default*])
返回当前上下文中此上下文变量的值。

如果当前上下文中此变量没有值，则此方法会:

- 如果提供了得话，返回传入的 *default* 值；或者
- 返回上下文变量本身的默认值，如果创建此上下文变量时提供了默认值；或者
- 抛出*LookupError* 异常。

set(*value*)
调用此方法设置上下文变量在当前上下文中的值。

必选参数 *value* 是上下文变量的新值。

返回一个*Token* 对象，可通过*ContextVar.reset()* 方法将上下文变量还原为之前某个状态。

reset(*token*)
将上下文变量重置为调用*ContextVar.set()* 之前、创建 *token* 时候的状态。

例如:

```
var = ContextVar('var')

token = var.set('new value')
# code that uses 'var'; var.get() returns 'new value'.
var.reset(token)

# After the reset call the var has no value again, so
# var.get() would raise a LookupError.
```

class contextvars.**Token**
ContextVar.set() 方法返回 *Token* 对象。此对象可以传递给*ContextVar.reset()* 方法用于将上下文变量还原为调用 *set* 前的状态。

var
只读属性。指向创建此 token 的*ContextVar* 对象。

old_value
一个只读属性。会被设为在创建此令牌的*ContextVar.set()* 方法调用之前该变量所具有的值。如果调用之前变量没有设置值，则它指向*Token.MISSING* 。

MISSING
Token.old_value 会用到的一个标记对象。

17.9.2 手动上下文管理

contextvars.**copy_context**()
返回当前上下文中*Context* 对象的拷贝。

以下代码片段会获取当前上下文的拷贝并打印设置到其中的所有变量及其值:

```
ctx: Context = copy_context()
print(list(ctx.items()))
```

此函数复杂度为 O(1)，也就是说对于只包含几个上下文变量和很多上下文变量的情况，他们是一样快的。

class contextvars.**Context**
ContextVars 中所有值的映射。

Context() 创建一个不包含任何值的空上下文。如果要获取当前上下文的拷贝，使用*copy_context()* 函数。

Context 实现了*collections.abc.Mapping* 接口。

run(*callable*, **args*, ***kwargs*)
按照 *run* 方法中的参数在上下文对象中执行 callable(*args, **kwargs) 代码。返回执行结果，如果发生异常，则将异常透传出来。

callable 对上下文变量所做的任何修改都会保留在上下文对象中:

```
var = ContextVar('var')
var.set('spam')

def main():
    # 'var' was set to 'spam' before
    # calling 'copy_context()' and 'ctx.run(main)', so:
    # var.get() == ctx[var] == 'spam'

    var.set('ham')

    # Now, after setting 'var' to 'ham':
    # var.get() == ctx[var] == 'ham'

ctx = copy_context()

# Any changes that the 'main' function makes to 'var'
# will be contained in 'ctx'.
ctx.run(main)

# The 'main()' function was run in the 'ctx' context,
# so changes to 'var' are contained in it:
# ctx[var] == 'ham'

# However, outside of 'ctx', 'var' is still set to 'spam':
# var.get() == 'spam'
```

当在多个系统线程或者递归调用同一个上下文对象的此方法，抛出*RuntimeError* 异常。

copy()
返回此上下文对象的浅拷贝。

var in context
如果 *context* 中含有名称为 *var* 的变量，返回 `True` ，否则返回 `False` 。

context[var]
返回名称为 *var* 的*ContextVar* 变量。如果上下文对象中不包含这个变量，则抛出*KeyError* 异常。

get(*var*[, *default*])
如果 *var* 在上下文对象中具有值则返回 *var* 的值。在其他情况下返回 *default*。如果未给出 *default* 则返回 `None`。

iter(context)
返回一个存储在上下文对象中的变量的迭代器。

len(proxy)
返回上下文对象中所设的变量的数量。

keys()
返回上下文对象中的所有变量的列表。

values()
返回上下文对象中所有变量值的列表。

items()
返回包含上下文对象中所有变量及其值的 2 元组的列表。

17.9.3 asyncio 支持

上下文变量在*asyncio* 中有原生的支持并且无需任何额外配置即可被使用。例如，以下是一个简单的回显服务器，它使用上下文变量来让远程客户端的地址在处理该客户端的 Task 中可用:

```
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr')

def render_goodbye():
    # The address of the currently handled client can be accessed
    # without passing it explicitly to this function.

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\n'.encode()

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # In any code that we call is now possible to get
    # client's address by calling 'client_addr_var.get()'.

    while True:
        line = await reader.readline()
        print(line)
        if not line.strip():
            break
```

(下页继续)

(繼續上一頁)

```
        writer.write(line)

    writer.write(render_goodbye())
    writer.close()

async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())

# To test it you can use telnet:
#     telnet 127.0.0.1 8081
```

以下是上述某些服务的支持模块:

17.10 _thread --- 底层多线程 API

该模块提供了操作多个线程（也被称为 *轻量级进程* 或 *任务*）的底层原语——多个控制线程共享全局数据空间。为了处理同步问题，也提供了简单的锁机制（也称为 *互斥锁* 或 *二进制信号*）。*threading* 模块基于该模块提供了更易用的高级多线程 API。

3.7 版更變: 这个模块曾经是可选的，但现在总是可用的。

这个模块定义了以下常量和函数:

exception _thread.**error**
发生线程相关错误时抛出。

3.3 版更變: 现在是内建异常*RuntimeError* 的别名。

_thread.**LockType**
锁对象的类型。

_thread.**start_new_thread**(*function*, *args*[, *kwargs*])
开启一个新线程并返回其标识。线程执行函数 *function* 并附带参数列表 *args* (必须是元组)。可选的 *kwargs* 参数指定一个关键字参数字典。

当函数返回时，线程会静默地退出。

当函数因某个未处理异常而终结时，*sys.unraisablehook()* 会被调用以处理异常。钩子参数的 *object* 属性为 *function*。在默认情况下，会打印堆栈回溯然后该线程将退出（但其他线程会继续运行）。

当函数引发*SystemExit* 异常时，它会被静默地忽略。

3.8 版更變: 现在会使用*sys.unraisablehook()* 来处理未处理的异常。

_thread.**interrupt_main**()
模拟一个*signal.SIGINT* 信号到达主线程的效果。线程可以使用这个函数来中断主线程。

如果 Python 没有处理*signal.SIGINT* (将它设为*signal.SIG_DFL* 或*signal.SIG_IGN*)，此函数将不做任何事。

_thread.**exit**()
抛出*SystemExit* 异常。如果没有捕获的话，这个异常会使线程退出。

_thread.**allocate_lock**()
返回一个新的锁对象。锁中的方法在后面描述。初始情况下锁处于解锁状态。

_thread.**get_ident**()
返回当前线程的“线程描述符”。它是一个非零的整型数。它的值没有什么含义，主要是作为 magic cookie 使用，比如作为含有线程相关数据的字典的索引。线程描述符可能会在线程退出，新线程创建时复用。

_thread.**get_native_id**()
返回内核分配给当前线程的原生集成线程 ID。这是一个非负整数。它的值可被用来在整个系统中唯一地标识这个特定线程（直到线程终结，在那之后该值可能会被 OS 回收再利用）。

可用性: Windows, FreeBSD, Linux, macOS, OpenBSD, NetBSD, AIX。

3.8 版新加入.

_thread.**stack_size**([*size*])
返回新建线程时使用的堆栈大小。可选参数 *size* 指定之后新建的线程的堆栈大小，而且一定要是 0（根据平台或者默认配置）或者最小是 32,768(32KiB) 的一个正整数。如果 *size* 没有指定，默认是 0。如果不支持改变线程堆栈大小，会抛出*RuntimeError* 错误。如果指定的堆栈大小不合法，会抛出*ValueError* 错误并且不会修改堆栈大小。32KiB 是当前最小的能保证解释器足够堆栈空间的堆栈大小。需要注意的是部分平台对于堆栈大小会有特定的限制，例如要求大于 32KiB 的堆栈大小或者需要根据系统内存页面的整数倍进行分配 - 应当查阅平台文档有关详细信息（4KiB 页面比较普遍，在没有更具体信息的情况下，建议的方法是使用 4096 的倍数作为堆栈大小）

可用性: Windows，具有 POSIX 线程的系统。

_thread.**TIMEOUT_MAX**
Lock.acquire() 方法中 *timeout* 参数允许的最大值。传入超过这个值的 timeout 会抛出*OverflowError* 异常。

3.2 版新加入.

锁对象有以下方法：

lock.**acquire**(*waitflag=1*, *timeout=-1*)
没有任何可选参数时，该方法无条件申请获得锁，有必要的话会等待其他线程释放锁（同时只有一个线程能获得锁——这正是锁存在的原因）。

如果传入了整型参数 *waitflag*，具体的行为取决于传入的值：如果是 0 的话，只会在能够立刻获取到锁时才获取，不会等待，如果是非零的话，会像之前提到的一样，无条件获取锁。

如果传入正浮点数参数 *timeout*，相当于指定了返回之前等待得最大秒数。如果传入负的 *timeout*，相当于无限期等待。如果 *waitflag* 是 0 的话，不能指定 *timeout*。

如果成功获取到所会返回 True，否则返回 False。

3.2 版更變: *timeout* 形参是新增的。

3.2 版更變: 现在获取锁的操作可以被 POSIX 信号中断。

lock.**release**()
释放锁。锁必须已经被获取过，但不一定是同一个线程获取的。

lock.**locked**()
返回锁的状态：如果已被某个线程获取，返回 True，否则返回 False。

除了这些方法之外，锁对象也可以通过 with 语句使用，例如：

```
import _thread

a_lock = _thread.allocate_lock()

with a_lock:
    print("a_lock is locked while this executes")
```

注意事项：

- 线程与中断奇怪地交互：*KeyboardInterrupt* 异常可能会被任意一个线程捕获。（如果*signal* 模块可用的话，中断总是会进入主线程。）
- 调用*sys.exit()* 或是抛出*SystemExit* 异常等效于调用*_thread.exit()*。
- 不可能中断锁的 `acquire()` 方法——*KeyboardInterrupt* 一场会在锁获取到之后发生。
- 当主线程退出时，由系统决定其他线程是否存活。在大多数系统中，这些线程会直接被杀掉，不会执行 `try ... finally` 语句，也不会执行对象析构函数。
- 当主线程退出时，不会进行正常的清理工作（除非使用了 `try ... finally` 语句），标准 I/O 文件也不会刷新。

CHAPTER 18

网络和进程间通信

本章介绍的模块提供了网络和进程间通信的机制。

某些模块仅适用于同一台机器上的两个进程，例如*signal* 和*mmap* 。其他模块支持两个或多个进程可用于跨机器通信的网络协议。

本章中描述的模块列表是：

18.1 asyncio --- 异步 I/O

Hello World!

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')

asyncio.run(main())
```

asyncio 是用来编写 **并发**代码的库，使用 **async/await** 语法。

asyncio 被用作多个提供高性能 Python 异步框架的基础，包括网络和网站服务，数据库连接库，分布式任务队列等等。

asyncio 往往是构建 IO 密集型和高层级 **结构化**网络代码的最佳选择。

asyncio 提供一组 **高层级** API 用于：

- 并发地运行 *Python* 协程 并对其执行过程实现完全控制；
- 执行网络 *IO* 和 *IPC*；

- 控制子进程;
- 通过队列 实现分布式任务;
- 同步 并发代码;

此外，还有一些 **低层级** API 以支持 库和框架的开发者实现:

- 创建和管理事件循环，以提供异步 API 用于网络化, 运行子进程，处理OS 信号 等等;
- 使用*transports* 实现高效率协议;
- 通过 async/await 语法桥接 基于回调的库和代码。

参考引用

18.1.1 协程与任务

本节将简述用于协程与任务的高层级 API。

协程

Coroutines declared with the async/await syntax is the preferred way of writing asyncio applications. For example, the following snippet of code prints ”hello”, waits 1 second, and then prints ”world”:

```
>>> import asyncio

>>> async def main():
...     print('hello')
...     await asyncio.sleep(1)
...     print('world')
```

(下页继续)

(繼續上一頁)

```
>>> asyncio.run(main())
hello
world
```

注意：简单地调用一个协程并不会使其被调度执行

```
>>> main()
<coroutine object main at 0x1053bb7c8>
```

要真正运行一个协程，asyncio 提供了三种主要机制:

- *asyncio.run()* 函数用来运行最高层级的入口点"main()" 函数 (参见上面的示例。)
- 等待一个协程。以下代码段会在等待 1 秒后打印"hello"，然后 再次等待 2 秒后打印"world":

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')

    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())
```

预期的输出:

```
started at 17:13:52
hello
world
finished at 17:13:55
```

- *asyncio.create_task()* 函数用来并发运行作为 asyncio 任务 的多个协程。

  让我们修改以上示例，并发运行两个 `say_after` 协程:

```
async def main():
    task1 = asyncio.create_task(
        say_after(1, 'hello'))

    task2 = asyncio.create_task(
        say_after(2, 'world'))

    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2
```

(下页继续)

(繼續上一頁)

```
    print(f"finished at {time.strftime('%X')}")
```

注意，预期的输出显示代码段的运行时间比之前快了 1 秒:

```
started at 17:14:32
hello
world
finished at 17:14:34
```

可等待对象

如果一个对象可以在 `await` 语句中使用，那么它就是 **可等待**对象。许多 asyncio API 都被设计为接受可等待对象。

可等待对象有三种主要类型: **协程**, **任务**和 **Future**.

协程

Python 协程属于 可等待对象，因此可以在其他协程中被等待:

```
import asyncio

async def nested():
    return 42

async def main():
    # Nothing happens if we just call "nested()".
    # A coroutine object is created but not awaited,
    # so it *won't run at all*.
    nested()

    # Let's do it differently now and await it:
    print(await nested())  # will print "42".

asyncio.run(main())
```

重要: 在本文档中" 协程" 可用来表示两个紧密关联的概念:

- *协程函数*: 定义形式为 `async def` 的函数;
- *协程对象*: 调用 *协程函数*所返回的对象。

---

asyncio 也支持旧式的基于生成器的 协程。

任务

任务被用来“并行的”调度协程

当一个协程通过*asyncio.create_task()* 等函数被封装为一个 任务，该协程会被自动调度执行:

```
import asyncio

async def nested():
    return 42

async def main():
    # Schedule nested() to run soon concurrently
    # with "main()".
    task = asyncio.create_task(nested())

    # "task" can now be used to cancel "nested()", or
    # can simply be awaited to wait until it is complete:
    await task

asyncio.run(main())
```

Future 对象

Future 是一种特殊的 **低层级**可等待对象，表示一个异步操作的 **最终结果**。

当一个 Future 对象 被等待，这意味着协程将保持等待直到该 Future 对象在其他地方操作完毕。

在 asyncio 中需要 Future 对象以便允许通过 async/await 使用基于回调的代码。

通常情况下 **没有必要**在应用层级的代码中创建 Future 对象。

Future 对象有时会由库和某些 asyncio API 暴露给用户，用作可等待对象:

```
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

一个很好的返回对象的低层级函数的示例是*loop.run_in_executor()*。

运行 asyncio 程序

asyncio.**run**(*coro*, *, *debug=False*)

执行*coroutine* *coro* 并返回结果。

此函数会运行传入的协程，负责管理 asyncio 事件循环，终结异步生成器，并关闭线程池。

当有其他 asyncio 事件循环在同一线程中运行时，此函数不能被调用。

如果 *debug* 为 `True`，事件循环将以调试模式运行。

此函数总是会创建一个新的事件循环并在结束时关闭之。它应当被用作 asyncio 程序的主入口点，理想情况下应当只被调用一次。

示例:

```
async def main():
    await asyncio.sleep(1)
    print('hello')

asyncio.run(main())
```

3.7 版新加入.

3.9 版更變: 更新为使用*loop.shutdown_default_executor()*。

備🄵: `asyncio.run()` 的源代码可以在 Lib/asyncio/runners.py 中找到。

创建任务

`asyncio.`**`create_task`**(*coro*, *, *name=None*)

将 *coro* 协程 封装为一个*Task* 并调度其执行。返回 Task 对象。

name 不为 `None`，它将使用*Task.set_name()* 来设为任务的名称。

该任务会在*get_running_loop()* 返回的循环中执行，如果当前线程没有在运行的循环则会引发*RuntimeError*。

重要: Save a reference to the result of this function, to avoid a task disappearing mid execution.

3.7 版新加入.

3.8 版更變: 添加了 `name` 形参。

休眠

`coroutine` `asyncio.`**`sleep`**(*delay*, *result=None*, *, *loop=None*)

阻塞 *delay* 指定的秒数。

如果指定了 *result*，则当协程完成时将其返回给调用者。

`sleep()` 总是会挂起当前任务，以允许其他任务运行。

将 delay 设为 0 将提供一个经优化的路径以允许其他任务运行。这可供长期间运行的函数使用以避免在函数调用的全过程中阻塞事件循环。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。 以下协程示例运行 5 秒，每秒显示一次当前日期:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
        print(datetime.datetime.now())
        if (loop.time() + 1.0) >= end_time:
            break
```

(下页继续)

(繼續上一頁)

```
        await asyncio.sleep(1)

asyncio.run(display_date())
```

并发运行任务

awaitable asyncio.**gather**(**aws*, *loop=None*, *return_exceptions=False*)

并发运行 *aws* 序列中的可等待对象。

如果 *aws* 中的某个可等待对象为协程，它将自动被作为一个任务调度。

如果所有可等待对象都成功完成，结果将是一个由所有返回值聚合而成的列表。结果值的顺序与 *aws* 中可等待对象的顺序一致。

如果 *return_exceptions* 为 `False` (默认)，所引发的首个异常会立即传播给等待 `gather()` 的任务。*aws* 序列中的其他可等待对象 **不会被取消**并将继续运行。

如果 *return_exceptions* 为 `True`，异常会和成功的结果一样处理，并聚合至结果列表。

如果 `gather()` 被取消，所有被提交 (尚未完成) 的可等待对象也会 被取消。

如果 *aws* 序列中的任一 Task 或 Future 对象 被取消，它将被当作引发了*CancelledError* 一样处理 -- 在此情况下 `gather()` 调用 **不会**被取消。这是为了防止一个已提交的 Task/Future 被取消导致其他 Tasks/Future 也被取消。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。 示例:

```
import asyncio

async def factorial(name, number):
    f = 1
    for i in range(2, number + 1):
        print(f"Task {name}: Compute factorial({number}), currently i={i}...")
        await asyncio.sleep(1)
        f *= i
    print(f"Task {name}: factorial({number}) = {f}")
    return f

async def main():
    # Schedule three calls *concurrently*:
    L = await asyncio.gather(
        factorial("A", 2),
        factorial("B", 3),
        factorial("C", 4),
    )
    print(L)

asyncio.run(main())

# Expected output:
#
#     Task A: Compute factorial(2), currently i=2...
#     Task B: Compute factorial(3), currently i=2...
#     Task C: Compute factorial(4), currently i=2...
#     Task A: factorial(2) = 2
#     Task B: Compute factorial(3), currently i=3...
#     Task C: Compute factorial(4), currently i=3...
```

(下页继续)

(繼續上一頁)

```
#      Task B: factorial(3) = 6
#      Task C: Compute factorial(4), currently i=4...
#      Task C: factorial(4) = 24
#      [2, 6, 24]
```

備🄵: 如果 *return_exceptions* 为 False，则在 gather() 被标记为已完成后取消它将不会取消任何已提交的可等待对象。例如，在将一个异常传播给调用者之后，gather 可被标记为已完成，因此，在从 gather 捕获一个（由可等待对象所引发的）异常之后调用 `gather.cancel()` 将不会取消任何其他可等待对象。

3.7 版更變: 如果 *gather* 本身被取消，则无论 *return_exceptions* 取值为何，消息都会被传播。

屏蔽取消操作

awaitable asyncio.**shield**(*aw*, *, *loop=None*)

保护一个可等待对象 防止其被取消。

如果 *aw* 是一个协程，它将自动被作为任务调度。

以下语句:

```
res = await shield(something())
```

相当于:

```
res = await something()
```

不同之处在于如果包含它的协程被取消，在 `something()` 中运行的任务不会被取消。从 `something()` 的角度看来，取消操作并没有发生。然而其调用者已被取消，因此"await" 表达式仍然会引发*CancelledError*。

如果通过其他方式取消 `something()` (例如在其内部操作) 则 `shield()` 也会取消。

如果希望完全忽略取消操作 (不推荐) 则 `shield()` 函数需要配合一个 try/except 代码段，如下所示:

```
try:
    res = await shield(something())
except CancelledError:
    res = None
```

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。

超时

coroutine asyncio.**wait_for**(*aw*, *timeout*, *, *loop=None*)

等待 *aw* 可等待对象 完成，指定 timeout 秒数后超时。

如果 *aw* 是一个协程，它将自动被作为任务调度。

timeout 可以为 `None`，也可以为 float 或 int 型数值表示的等待秒数。如果 *timeout* 为 `None`，则等待直到完成。

如果发生超时，任务将取消并引发*asyncio.TimeoutError*.

要避免任务取消，可以加上*shield()*。

此函数将等待直到 Future 确实被取消，所以总等待时间可能超过 *timeout*。如果在取消期间发生了异常，异常将会被传播。

如果等待被取消，则 *aw* 指定的对象也会被取消。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。 示例:

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except asyncio.TimeoutError:
        print('timeout!')

asyncio.run(main())

# Expected output:
#
#     timeout!
```

3.7 版更變: 当 *aw* 因超时被取消，`wait_for` 会等待 *aw* 被取消。之前版本则将立即引发*asyncio.TimeoutError*。

简单等待

coroutine asyncio.**wait**(*aws*, *, *loop=None*, *timeout=None*, *return_when=ALL_COMPLETED*)

并发地运行 *aws* 可迭代对象中的可等待对象 并进入阻塞状态直到满足 *return_when* 所指定的条件。

aws 可迭代对象必须不为空。

返回两个 Task/Future 集合: `(done, pending)`。

用法:

```
done, pending = await asyncio.wait(aws)
```

如指定 *timeout* (float 或 int 类型) 则它将被用于控制返回之前等待的最长秒数。

请注意此函数不会引发*asyncio.TimeoutError*。当超时发生时，未完成的 Future 或 Task 将在指定秒数后被返回。

return_when 指定此函数应在何时返回。它必须为以下常数之一:

常数	描述
FIRST_COMPLETED	函数将在任意可等待对象结束或取消时返回。
FIRST_EXCEPTION	函数将在任意可等待对象因引发异常而结束时返回。当没有引发任何异常时它就相当于 ALL_COMPLETED。
ALL_COMPLETED	函数将在所有可等待对象结束或取消时返回。

与*wait_for()* 不同，`wait()` 在超时发生时不会取消可等待对象。

3.8 版後已F用: 如果 *aws* 中的某个可等待对象为协程，它将自动被作为任务调度。直接向 `wait()` 传入协程对象已弃用，因为这会导致令人迷惑的行为。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。

備🄵: `wait()` 会自动以任务的形式调度协程，之后将以 `(done, pending)` 集合形式返回显式创建的任务对象。因此以下代码并不会有预期的行为:

```
async def foo():
    return 42

coro = foo()
done, pending = await asyncio.wait({coro})

if coro in done:
    # This branch will never be run!
```

以上代码段的修正方法如下:

```
async def foo():
    return 42

task = asyncio.create_task(foo())
done, pending = await asyncio.wait({task})

if task in done:
    # Everything will work as expected now.
```

Deprecated since version 3.8, will be removed in version 3.11: 直接向 `wait()` 传入协程对象的方式已弃用。

asyncio.**as_completed**(*aws*, *, *loop=None*, *timeout=None*)

并发地运行 *aws* 可迭代对象中的可等待对象。返回一个协程的迭代器。所返回的每个协程可被等待以从剩余的可等待对象的可迭代对象中获得最早的下一个结果。

如果在所有 Future 对象完成前发生超时则将引发*asyncio.TimeoutError*。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。

示例:

```
for coro in as_completed(aws):
    earliest_result = await coro
    # ...
```

在线程中运行

coroutine asyncio.**to_thread**(*func*, */*, **args*, ***kwargs*)

在不同的线程中异步地运行函数 *func*。

向此函数提供的任何 *args 和 **kwargs 会被直接传给 *func*。并且，当前*contextvars.Context* 会被传播，允许在不同的线程中访问来自事件循环的上下文变量。

返回一个可被等待以获取 *func* 的最终结果的协程。

这个协程函数主要是用于执行在其他情况下会阻塞事件循环的 IO 密集型函数/方法。例如:

```
def blocking_io():
    print(f"start blocking_io at {time.strftime('%X')}")
    # Note that time.sleep() can be replaced with any blocking
```

(下页继续)

(繼續上一頁)

```
    # IO-bound operation, such as file operations.
    time.sleep(1)
    print(f"blocking_io complete at {time.strftime('%X')}")

async def main():
    print(f"started main at {time.strftime('%X')}")

    await asyncio.gather(
        asyncio.to_thread(blocking_io),
        asyncio.sleep(1))

    print(f"finished main at {time.strftime('%X')}")


asyncio.run(main())

# Expected output:
#
# started main at 19:50:53
# start blocking_io at 19:50:53
# blocking_io complete at 19:50:54
# finished main at 19:50:54
```

在任何协程中直接调用 *blocking_io()* 将会在调用期间阻塞事件循环，导致额外的 1 秒运行时间。而通过改用 *asyncio.to_thread()*，我们可以在不同的线程中运行它从而不会阻塞事件循环。

備F: 由于*GIL* 的存在，*asyncio.to_thread()* 通常只能被用来将 IO 密集型函数变为非阻塞的。但是，对于会释放 GIL 的扩展模块或无此限制的替代性 Python 实现来说，*asyncio.to_thread()* 也可被用于 CPU 密集型函数。

3.9 版新加入.

跨线程调度

asyncio.**run_coroutine_threadsafe**(*coro*, *loop*)

向指定事件循环提交一个协程。(线程安全)

返回一个*concurrent.futures.Future* 以等待来自其他 OS 线程的结果。

此函数应该从另一个 OS 线程中调用，而非事件循环运行所在线程。示例:

```
# Create a coroutine
coro = asyncio.sleep(1, result=3)

# Submit the coroutine to a given loop
future = asyncio.run_coroutine_threadsafe(coro, loop)

# Wait for the result with an optional timeout argument
assert future.result(timeout) == 3
```

如果在协程内产生了异常，将会通知返回的 Future 对象。它也可被用来取消事件循环中的任务:

```
try:
    result = future.result(timeout)
```

(下页继续)

(繼續上一頁)

```
except asyncio.TimeoutError:
    print('The coroutine took too long, cancelling the task...')
    future.cancel()
except Exception as exc:
    print(f'The coroutine raised an exception: {exc!r}')
else:
    print(f'The coroutine returned: {result!r}')
```

查看并发和多线程 章节的文档。

不同与其他 asyncio 函数，此函数要求显式地传入 *loop* 参数。

3.5.1 版新加入.

内省

asyncio.**current_task**(*loop=None*)
返回当前运行的*Task* 实例，如果没有正在运行的任务则返回 None。

如果 *loop* 为 None 则会使用*get_running_loop()* 获取当前事件循环。

3.7 版新加入.

asyncio.**all_tasks**(*loop=None*)
返回事件循环所运行的未完成的*Task* 对象的集合。

如果 *loop* 为 None，则会使用*get_running_loop()* 获取当前事件循环。

3.7 版新加入.

Task 对象

class asyncio.**Task**(*coro*, *, *loop=None*, *name=None*)
一个与*Future* 类似 的对象，可运行 Python 协程。非线程安全。

Task 对象被用来在事件循环中运行协程。如果一个协程在等待一个 Future 对象，Task 对象会挂起该协程的执行并等待该 Future 对象完成。当该 Future 对象 完成，被打包的协程将恢复执行。

事件循环使用协同日程调度: 一个事件循环每次运行一个 Task 对象。而一个 Task 对象会等待一个 Future 对象完成，该事件循环会运行其他 Task、回调或执行 IO 操作。

使用高层级的*asyncio.create_task()* 函数来创建 Task 对象，也可用低层级的*loop.create_task()* 或*ensure_future()* 函数。不建议手动实例化 Task 对象。

要取消一个正在运行的 Task 对象可使用*cancel()* 方法。调用此方法将使该 Task 对象抛出一个*CancelledError* 异常给打包的协程。如果取消期间一个协程正在等待一个 Future 对象，该 Future 对象也将被取消。

cancelled() 可被用来检测 Task 对象是否被取消。如果打包的协程没有抑制*CancelledError* 异常并且确实被取消，该方法将返回 True。

asyncio.Task 从*Future* 继承了其除*Future.set_result()* 和*Future.set_exception()* 以外的所有 API。

Task 对象支持*contextvars* 模块。当一个 Task 对象被创建，它将复制当前上下文，然后在复制的上下文中运行其协程。

3.7 版更變: 加入对*contextvars* 模块的支持。

3.8 版更變: 添加了 name 形参。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。

cancel(*msg=None*)
请求取消 Task 对象。

这将安排在下一轮事件循环中抛出一个*CancelledError* 异常给被封包的协程。

协程在之后有机会进行清理甚至使用 `try` ... ... `except CancelledError` ... `finally` 代码块抑制异常来拒绝请求。不同于*Future.cancel()*，*Task.cancel()* 不保证 Task 会被取消，虽然抑制完全取消并不常见，也很不鼓励这样做。

3.9 版更變: 增加了 `msg` 形参。 以下示例演示了协程是如何侦听取消请求的:

```
async def cancel_me():
    print('cancel_me(): before sleep')

    try:
        # Wait for 1 hour
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        print('cancel_me(): cancel sleep')
        raise
    finally:
        print('cancel_me(): after sleep')

async def main():
    # Create a "cancel_me" Task
    task = asyncio.create_task(cancel_me())

    # Wait for 1 second
    await asyncio.sleep(1)

    task.cancel()
    try:
        await task
    except asyncio.CancelledError:
        print("main(): cancel_me is cancelled now")

asyncio.run(main())

# Expected output:
#
#     cancel_me(): before sleep
#     cancel_me(): cancel sleep
#     cancel_me(): after sleep
#     main(): cancel_me is cancelled now
```

cancelled()
如果 Task 对象 被取消则返回 `True`。

当使用*cancel()* 发出取消请求时 Task 会被取消，其封包的协程将传播被抛入的*CancelledError* 异常。

done()
如果 Task 对象 已完成则返回 `True`。

当 Task 所封包的协程返回一个值、引发一个异常或 Task 本身被取消时，则会被认为 已完成。

result()
返回 Task 的结果。

如果 Task 对象 *已完成*，其封包的协程的结果会被返回 (或者当协程引发异常时，该异常会被重新引发。)

如果 Task 对象 *被取消*，此方法会引发一个*CancelledError* 异常。

如果 Task 对象的结果还不可用，此方法会引发一个*InvalidStateError* 异常。

exception()
返回 Task 对象的异常。

如果所封包的协程引发了一个异常，该异常将被返回。如果所封包的协程正常返回则该方法将返回 None。

如果 Task 对象 *被取消*，此方法会引发一个*CancelledError* 异常。

如果 Task 对象尚未 *完成*，此方法将引发一个*InvalidStateError* 异常。

add_done_callback(*callback*, *, *context=None*)
添加一个回调，将在 Task 对象 *完成*时被运行。

此方法应该仅在低层级的基于回调的代码中使用。

要了解更多细节请查看*Future.add_done_callback()* 的文档。

remove_done_callback(*callback*)
从回调列表中移除 *callback* 指定的回调。

此方法应该仅在低层级的基于回调的代码中使用。

要了解更多细节请查看*Future.remove_done_callback()* 的文档。

get_stack(*, *limit=None*)
返回此 Task 对象的栈框架列表。

如果所封包的协程未完成，这将返回其挂起所在的栈。如果协程已成功完成或被取消，这将返回一个空列表。如果协程被一个异常终止，这将返回回溯框架列表。

框架总是从按从旧到新排序。

每个被挂起的协程只返回一个栈框架。

可选的 *limit* 参数指定返回框架的数量上限；默认返回所有框架。返回列表的顺序要看是返回一个栈还是一个回溯：栈返回最新的框架，回溯返回最旧的框架。(这与 traceback 模块的行为保持一致。)

print_stack(*, *limit=None*, *file=None*)
打印此 Task 对象的栈或回溯。

此方法产生的输出类似于 traceback 模块通过*get_stack()* 所获取的框架。

limit 参数会直接传递给*get_stack()*。

file 参数是输出所写入的 I/O 流；默认情况下输出会写入*sys.stderr*。

get_coro()
返回由*Task* 包装的协程对象。

3.8 版新加入.

get_name()
返回 Task 的名称。

如果没有一个 Task 名称被显式地赋值，默认的 asyncio Task 实现会在实例化期间生成一个默认名称。

3.8 版新加入.

set_name(*value*)
设置 Task 的名称。

value 参数可以为任意对象，它随后会被转换为字符串。

在默认的 Task 实现中，名称将在任务对象的*repr()* 输出中可见。

3.8 版新加入.

基于生成器的协程

備🄵: Support for generator-based coroutines is **deprecated** and is removed in Python 3.11.

基于生成器的协程是 async/await 语法的前身。它们是使用 `yield from` 语句创建的 Python 生成器，可以等待 Future 和其他协程。

基于生成器的协程应该使用*@asyncio.coroutine* 装饰，虽然这并非强制。

@asyncio.**coroutine**
用来标记基于生成器的协程的装饰器。

此装饰器使得旧式的基于生成器的协程能与 async/await 代码相兼容:

```
@asyncio.coroutine
def old_style_coroutine():
    yield from asyncio.sleep(1)

async def main():
    await old_style_coroutine()
```

此装饰器不应该被用于 `async def` 协程。

Deprecated since version 3.8, will be removed in version 3.11: 请改用 `async def`。

asyncio.**iscoroutine**(*obj*)
如果 *obj* 是一个协程对象 则返回 `True`。

此方法不同于*inspect.iscoroutine()* 因为它对基于生成器的协程返回 `True`。

asyncio.**iscoroutinefunction**(*func*)
如果 *func* 是一个协程函数 则返回 `True`。

此方法不同于*inspect.iscoroutinefunction()* 因为它对以*@coroutine* 装饰的基于生成器的协程函数返回 `True`。

18.1.2 流

源码: Lib/asyncio/streams.py

流是用于处理网络连接的支持 async/await 的高层级原语。流允许发送和接收数据，而不需要使用回调或低级协议和传输。

下面是一个使用 asyncio streams 编写的 TCP echo 客户端示例:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_echo_client('Hello World!'))
```

参见下面的*Examples* 部分。

Stream 函数

下面的高级 asyncio 函数可以用来创建和处理流:

coroutine asyncio.**open_connection**(*host=None*, *port=None*, ***, *loop=None*, *limit=None*, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*)

建立网络连接并返回一对 `(reader, writer)` 对象。

返回的 *reader* 和 *writer* 对象是*StreamReader* 和*StreamWriter* 类的实例。

loop 参数是可选的，当从协程中等待该函数时，总是可以自动确定。

limit 确定返回的*StreamReader* 实例使用的缓冲区大小限制。默认情况下，*limit* 设置为 64 KiB 。

其余的参数直接传递到*loop.create_connection()* 。

3.7 版新加入: *ssl_handshake_timeout* 形参。

3.8 版新加入: Added *happy_eyeballs_delay* and *interleave* parameters.

coroutine asyncio.**start_server**(*client_connected_cb*, *host=None*, *port=None*, ***, *loop=None*, *limit=None*, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

启动套接字服务。

当一个新的客户端连接被建立时，回调函数 *client_connected_cb* 会被调用。该函数会接收到一对参数 `(reader, writer)` ，reader 是类*StreamReader* 的实例，而 writer 是类*StreamWriter* 的实例。

client_connected_cb 即可以是普通的可调用对象也可以是一个协程函数; 如果它是一个协程函数，它将自动作为*Task* 被调度。

loop 参数是可选的。当在一个协程中 await 该方法时，该参数始终可以自动确定。

limit 确定返回的*StreamReader* 实例使用的缓冲区大小限制。默认情况下，*limit* 设置为 64 KiB 。

余下的参数将会直接传递给*loop.create_server()*.

3.7 版新加入: The *ssl_handshake_timeout* and *start_serving* parameters.

Unix 套接字

coroutine asyncio.**open_unix_connection**(*path=None*, ***, *loop=None*, *limit=None*, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

建立一个 Unix 套接字连接并返回 `(reader, writer)` 这对返回值。

与*open_connection()* 相似，但是是在 Unix 套接字上的操作。

请看文档*loop.create_unix_connection()*.

可用性: Unix。

3.7 版新加入: *ssl_handshake_timeout* 形参。

3.7 版更變: *path* 现在是一个*path-like object*

coroutine asyncio.**start_unix_server**(*client_connected_cb*, *path=None*, ***, *loop=None*, *limit=None*, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

启动一个 Unix 套接字服务。

与*start_server()* 相似，但是是在 Unix 套接字上的操作。

请看文档*loop.create_unix_server()*.

可用性: Unix。

3.7 版新加入: The *ssl_handshake_timeout* and *start_serving* parameters.

3.7 版更變: *path* 形参现在可以是*path-like object* 对象。

StreamReader

class asyncio.**StreamReader**

这个类表示一个读取器对象，该对象提供 api 以便于从 IO 流中读取数据。

不推荐直接实例化 *StreamReader* 对象，建议使用*open_connection()* 和*start_server()* 来获取 *StreamReader* 实例。

coroutine read(*n=-1*)

至多读取 *n* 个 byte。如果没有设置 *n*，则自动置为 `-1`，-1 时表示读至 EOF 并返回所有读取的 byte。

如果读到 EOF，且内部缓冲区为空，则返回一个空的 `bytes` 对象。

coroutine readline()

读取一行，其中“行”指的是以 `\n` 结尾的字节序列。

如果读到 EOF 而没有找到 `\n`，该方法返回部分读取的数据。

如果读到 EOF，且内部缓冲区为空，则返回一个空的 `bytes` 对象。

coroutine readexactly(*n*)

精确读取 *n* 个 bytes，不会超过也不能少于。

如果在读取完 *n* 个 byte 之前读取到 EOF，则会引发*IncompleteReadError* 异常。使用*IncompleteReadError.partial* 属性来获取到达流结束之前读取的 bytes 字符串。

coroutine readuntil(*separator=b'\n'*)
从流中读取数据直至遇到 *separator*

成功后，数据和指定的 separator 将从内部缓冲区中删除 (或者说被消费掉)。返回的数据将包括在末尾的指定 separator。

如果读取的数据量超过了配置的流限制，将引发*LimitOverrunError* 异常，数据将留在内部缓冲区中并可以再次读取。

如果在找到完整的 separator 之前到达 EOF，则会引发*IncompleteReadError* 异常，并重置内部缓冲区。*IncompleteReadError.partial* 属性可能包含指定 separator 的一部分。

3.5.2 版新加入.

at_eof()
如果缓冲区为空并且 `feed_eof()` 被调用，则返回 `True` 。

StreamWriter

class asyncio.**StreamWriter**
这个类表示一个写入器对象，该对象提供 api 以便于写数据至 IO 流中。

不建议直接实例化 *StreamWriter*；而应改用*open_connection()* 和*start_server()*。

write(*data*)
此方法会尝试立即将 *data* 写入到下层的套接字。如果写入失败，数据会被排入内部写缓冲队列直到可以被发送。

此方法应当与 `drain()` 方法一起使用:

```
stream.write(data)
await stream.drain()
```

writelines(*data*)
此方法会立即尝试将一个字节串列表（或任何可迭代对象）写入到下层的套接字。如果写入失败，数据会被排入内部写缓冲队列直到可以被发送。

此方法应当与 `drain()` 方法一起使用:

```
stream.writelines(lines)
await stream.drain()
```

close()
此方法会关闭流以及下层的套接字。

此方法应与 `wait_closed()` 方法一起使用:

```
stream.close()
await stream.wait_closed()
```

can_write_eof()
如果下层的传输支持*write_eof()* 方法则返回 “True“，否则返回 `False`。

write_eof()
在已缓冲的写入数据被刷新后关闭流的写入端。

transport
返回下层的 asyncio 传输。

get_extra_info(*name*, *default=None*)
访问可选的传输信息；详情参见*BaseTransport.get_extra_info()*。

coroutine drain()
等待直到可以适当地恢复写入到流。示例:

```
writer.write(data)
await writer.drain()
```

这是一个与下层的 IO 写缓冲区进行交互的流程控制方法。当缓冲区大小达到最高水位（最大上限）时，*drain()* 会阻塞直到缓冲区大小减少至最低水位以便恢复写入。当没有要等待的数据时，*drain()* 会立即返回。

is_closing()
如果流已被关闭或正在被关闭则返回 `True`。

3.7 版新加入.

coroutine wait_closed()
等待直到流被关闭。

应当在*close()* 之后被调用以便等待直到下层的连接被关闭。

3.7 版新加入.

示例

使用流的 TCP 回显客户端

使用*asyncio.open_connection()* 函数的 TCP 回显客户端:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')

    print('Close the connection')
    writer.close()

asyncio.run(tcp_echo_client('Hello World!'))
```

也参考:

使用低层级*loop.create_connection()* 方法的*TCP 回显客户端协议* 示例。

使用流的 TCP 回显服务器

TCP 回显服务器使用*asyncio.start_server()* 函数:

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addrs = ', '.join(str(sock.getsockname()) for sock in server.sockets)
    print(f'Serving on {addrs}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

也参考:

使用*loop.create_server()* 方法的*TCP 回显服务器协议* 示例。

获取 HTTP 标头

查询命令行传入 URL 的 HTTP 标头的简单示例:

```
import asyncio
import urllib.parse
import sys

async def print_http_headers(url):
    url = urllib.parse.urlsplit(url)
    if url.scheme == 'https':
        reader, writer = await asyncio.open_connection(
            url.hostname, 443, ssl=True)
    else:
        reader, writer = await asyncio.open_connection(
            url.hostname, 80)

    query = (
        f"HEAD {url.path or '/'} HTTP/1.0\r\n"
        f"Host: {url.hostname}\r\n"
```

(下页继续)

(繼續上一頁)

```
        f"\r\n"
    )

    writer.write(query.encode('latin-1'))
    while True:
        line = await reader.readline()
        if not line:
            break

        line = line.decode('latin1').rstrip()
        if line:
            print(f'HTTP header> {line}')

    # Ignore the body, close the socket
    writer.close()

url = sys.argv[1]
asyncio.run(print_http_headers(url))
```

用法:

```
python example.py http://example.com/path/page.html
```

或使用 HTTPS:

```
python example.py https://example.com/path/page.html
```

注册一个打开的套接字以等待使用流的数据

使用*open_connection()* 函数实现等待直到套接字接收到数据的协程:

```
import asyncio
import socket

async def wait_for_data():
    # Get a reference to the current event loop because
    # we want to access low-level APIs.
    loop = asyncio.get_running_loop()

    # Create a pair of connected sockets.
    rsock, wsock = socket.socketpair()

    # Register the open socket to wait for data.
    reader, writer = await asyncio.open_connection(sock=rsock)

    # Simulate the reception of data from the network
    loop.call_soon(wsock.send, 'abc'.encode())

    # Wait for data
    data = await reader.read(100)

    # Got data, we are done: close the socket
    print("Received:", data.decode())
    writer.close()
```

(下页继续)

(繼續上一頁)

```
    # Close the second socket
    wsock.close()

asyncio.run(wait_for_data())
```

也參考:

使用低层级协议以及*loop.create_connection()* 方法的注册一个打开的套接字以等待使用协议的数据示例。

使用低层级的*loop.add_reader()* 方法来监视文件描述符的监视文件描述符以读取事件 示例。

18.1.3 同步原语

源代码: Lib/asyncio/locks.py

---

asyncio 同步原语被设计为与*threading* 模块的类似，但有两个关键注意事项:

- asyncio 原语不是线程安全的，因此它们不应被用于 OS 线程同步 (而应当使用*threading*);
- 这些同步原语的方法不接受 *timeout* 参数；请使用*asyncio.wait_for()* 函数来执行带有超时的操作。

asyncio 具有下列基本同步原语:

- *Lock*
- *Event*
- *Condition*
- *Semaphore*
- *BoundedSemaphore*

Lock

class asyncio.**Lock**(*, *loop=None*)

实现一个用于 asyncio 任务的互斥锁。非线程安全。

asyncio 锁可被用来保证对共享资源的独占访问。

使用 Lock 的推荐方式是通过 `async with` 语句:

```
lock = asyncio.Lock()

# ... later
async with lock:
    # access shared state
```

这等价于:

```
lock = asyncio.Lock()

# ... later
await lock.acquire()
try:
    # access shared state
finally:
    lock.release()
```

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。

coroutine acquire()
获取锁。

此方法会等待直至锁为 *unlocked*，将其设为 *locked* 并返回 `True`。

当有一个以上的协程在*acquire()* 中被阻塞则会等待解锁，最终只有一个协程会被执行。

锁的获取是 公平的: 被执行的协程将是第一个开始等待锁的协程。

release()
释放锁。

当锁为 *locked* 时，将其设为 *unlocked* 并返回。

如果锁为 *unlocked*，则会引发*RuntimeError*。

locked()
如果锁为 *locked* 则返回 `True`。

Event

class asyncio.**Event**(*, *loop=None*)
事件对象。该对象不是线程安全的。

asyncio 事件可被用来通知多个 asyncio 任务已经有事件发生。

Event 对象会管理一个内部旗标，可通过*set()* 方法将其设为 *true* 并通过*clear()* 方法将其重设为 *false*。*wait()* 方法会阻塞直至该旗标被设为 *true*。该旗标初始时会被设为 *false*。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。 示例:

```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task
```

(下页继续)

(繼續上一頁)

```
asyncio.run(main())
```

coroutine wait()
等待直至事件被设置。

如果事件已被设置，则立即返回 `True`。否则将阻塞直至另一个任务调用*set()*。

set()
设置事件。

所有等待事件被设置的任务将被立即唤醒。

clear()
清空（取消设置）事件。

通过*wait()* 进行等待的任务现在将会阻塞直至*set()* 方法被再次调用。

is_set()
如果事件已被设置则返回 `True`。

Condition

class asyncio.**Condition**(*lock=None*, *, *loop=None*)
条件对象。该对象不是线程安全的。

asyncio 条件原语可被任务用于等待某个事件发生，然后获取对共享资源的独占访问。

在本质上，Condition 对象合并了*Event* 和*Lock* 的功能。多个 Condition 对象有可能共享一个 Lock，这允许关注于共享资源的特定状态的不同任务实现对共享资源的协同独占访问。

可选的 *lock* 参数必须为*Lock* 对象或 `None`。在后一种情况下会自动创建一个新的 Lock 对象。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。

使用 Condition 的推荐方式是通过 `async with` 语句:

```
cond = asyncio.Condition()

# ... later
async with cond:
    await cond.wait()
```

这等价于:

```
cond = asyncio.Condition()

# ... later
await cond.acquire()
try:
    await cond.wait()
finally:
    cond.release()
```

coroutine acquire()
获取下层的锁。

此方法会等待直至下层的锁为 *unlocked*，将其设为 *locked* 并返回 returns `True`。

notify(*n=1*)
唤醒最多 *n* 个正在等待此条件的任务（默认为 1 个）。如果没有任务正在等待则此方法为空操作。

锁必须在此方法被调用前被获取并在随后被快速释放。如果通过一个 *unlocked* 锁调用则会引发*RuntimeError*。

locked()
如果下层的锁已被获取则返回 `True`。

notify_all()
唤醒所有正在等待此条件的任务。

此方法的行为类似于*notify()*，但会唤醒所有正在等待的任务。

锁必须在此方法被调用前被获取并在随后被快速释放。如果通过一个 *unlocked* 锁调用则会引发*RuntimeError*。

release()
释放下层的锁。

当在未锁定的锁上发起调用时，会引发*RuntimeError*。

coroutine wait()
等待直至收到通知。

当此方法被调用时如果调用方任务未获得锁，则会引发*RuntimeError*。

这个方法会释放下层的锁，然后保持阻塞直到被*notify()* 或*notify_all()* 调用所唤醒。一旦被唤醒，Condition 会重新获取它的锁并且此方法将返回 `True`。

coroutine wait_for(*predicate*)
等待直到目标值变为 *true*。

目标必须为一个可调用对象，其结果将被解读为一个布尔值。最终的值将为返回值。

Semaphore

class asyncio.**Semaphore**(*value=1*, *, *loop=None*)
信号量对象。该对象不是线程安全的。

信号量会管理一个内部计数器，该计数器会随每次*acquire()* 调用递减并随每次*release()* 调用递增。计数器的值永远不会降到零以下；当*acquire()* 发现其值为零时，它将保持阻塞直到有某个任务调用了*release()*。

可选的 *value* 参数用来为内部计数器赋初始值 (默认值为 `1`)。如果给定的值小于 `0` 则会引发*ValueError*。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。

使用 Semaphore 的推荐方式是通过 `async with` 语句。:

```
sem = asyncio.Semaphore(10)

# ... later
async with sem:
    # work with shared resource
```

这等价于:

```
sem = asyncio.Semaphore(10)

# ... later
await sem.acquire()
try:
    # work with shared resource
finally:
    sem.release()
```

coroutine acquire()
获取一个信号量。

如果内部计数器的值大于零，则将其减一并立即返回 True。如果其值为零，则会等待直到*release()* 并调用并返回 True。

locked()
如果信号量对象无法被立即获取则返回 True。

release()
释放一个信号量对象，将内部计数器的值加一。可以唤醒一个正在等待获取信号量对象的任务。

不同于*BoundedSemaphore*，*Semaphore* 允许执行的 release() 调用多于 acquire() 调用。

BoundedSemaphore

class asyncio.**BoundedSemaphore**(*value=1*, *, *loop=None*)
绑定的信号量对象。该对象不是线程安全的。

BoundedSemaphore 是特殊版本的*Semaphore*，如果在*release()* 中内部计数器值增加到初始 *value* 以上它将引发一个*ValueError*。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。

3.9 版更變: 使用 await lock 或 yield from lock 以及/或者 with 语句 (with await lock, with (yield from lock)) 来获取锁的操作已被移除。请改用 async with lock。

18.1.4 子进程

源代码: Lib/asyncio/subprocess.py, Lib/asyncio/base_subprocess.py

---

本节介绍了用于创建和管理子进程的高层级 async/await asyncio API。

下面的例子演示了如何用 asyncio 运行一个 shell 命令并获取其结果:

```
import asyncio

async def run(cmd):
    proc = await asyncio.create_subprocess_shell(
        cmd,
        stdout=asyncio.subprocess.PIPE,
        stderr=asyncio.subprocess.PIPE)

    stdout, stderr = await proc.communicate()
```

(下页继续)

(繼續上一頁)

```
    print(f'[{cmd!r} exited with {proc.returncode}]')
    if stdout:
        print(f'[stdout]\n{stdout.decode()}')
    if stderr:
        print(f'[stderr]\n{stderr.decode()}')

asyncio.run(run('ls /zzz'))
```

将打印:

```
['ls /zzz' exited with 1]
[stderr]
ls: /zzz: No such file or directory
```

由于所有 asyncio 子进程函数都是异步的并且 asyncio 提供了许多工具用来配合这些函数使用，因此并行地执行和监视多个子进程十分容易。要修改上面的例子来同时运行多个命令确实是非常简单的:

```
async def main():
    await asyncio.gather(
        run('ls /zzz'),
        run('sleep 1; echo "hello"'))

asyncio.run(main())
```

另请参阅*Examples* 小节。

创建子进程

coroutine asyncio.**create_subprocess_exec**(*program*, **args*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

创建一个子进程。

limit 参数为 `Process.stdout` 和 `Process.stderr` 设置*StreamReader* 包装器的缓冲区上限（如果将*subprocess.PIPE* 传给了 *stdout* 和 *stderr* 参数）。

返回一个*Process* 实例。

有关其他形参的说明请查阅*loop.subprocess_exec()* 的文档。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。

coroutine asyncio.**create_subprocess_shell**(*cmd*, *stdin=None*, *stdout=None*, *stderr=None*, *loop=None*, *limit=None*, ***kwds*)

运行 *cmd* shell 命令。

limit 参数为 `Process.stdout` 和 `Process.stderr` 设置*StreamReader* 包装器的缓冲区上限（如果将*subprocess.PIPE* 传给了 *stdout* 和 *stderr* 参数）。

返回一个*Process* 实例。

有关其他形参的说明请查阅*loop.subprocess_shell()* 的文档。

重要: 应用程序要负责确保正确地转义所有空白字符和特殊字符以防止 shell 注入 漏洞。*shlex.quote()* 函数可以被用来正确地转义字符串中可以被用来构造 shell 命令的空白字符和特殊 shell 字符。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。

備🄴: 如果使用了*ProactorEventLoop* 则子进程将在 Windows 中可用。详情参见*Windows 上的子进程支持*。

也参考:

asyncio 还有下列低层级 API 可配合子进程使用: *loop.subprocess_exec()*, *loop.subprocess_shell()*, *loop.connect_read_pipe()*, *loop.connect_write_pipe()* 以及*子进程传输* 和*子进程协议*。

常数

asyncio.subprocess.**PIPE**
可以被传递给 *stdin*, *stdout* 或 *stderr* 形参。

如果 *PIPE* 被传递给 *stdin* 参数，则*Process.stdin* 属性将会指向一个 StreamWriter 实例。

如果 *PIPE* 被传递给 *stdout* 或 *stderr* 参数，则*Process.stdout* 和*Process.stderr* 属性将会指向 StreamReader 实例。

asyncio.subprocess.**STDOUT**
可以用作 *stderr* 参数的特殊值，表示标准错误应当被重定向到标准输出。

asyncio.subprocess.**DEVNULL**
可以用作 *stdin*, *stdout* 或 *stderr* 参数来处理创建函数的特殊值。它表示将为相应的子进程流使用特殊文件*os.devnull*。

与子进程交互

create_subprocess_exec() 和*create_subprocess_shell()* 函数都返回 *Process* 类的实例。*Process* 是一个高层级包装器，它允许与子进程通信并监视其完成情况。

class asyncio.subprocess.**Process**
一个用于包装 create_subprocess_exec() and create_subprocess_shell() 函数创建的 OS 进程的对象。

这个类被设计为具有与*subprocess.Popen* 类相似的 API，但两者有一些重要的差异:

- 不同于 Popen，Process 实例没有与*poll()* 方法等价的方法;
- *communicate()* 和*wait()* 方法没有 *timeout* 形参；要使用 wait_for() 函数;
- *Process.wait()* 方法是异步的，而*subprocess.Popen.wait()* 方法则被实现为阻塞型忙循环;
- *universal_newlines* 形参不被支持。

这个类*不是线程安全的*。

请参阅*子进程和线程* 部分。

coroutine wait()
等待子进程终结。

设置并返回*returncode* 属性。

備F: 当使用 stdout=PIPE 或 stderr=PIPE 并且子进程产生了足以阻塞 OS 管道缓冲区等待接收更多的数据的输出时，此方法会发生死锁。当使用管道时请使用*communicate()* 方法来避免这种情况。

coroutine communicate(*input=None*)
与进程交互:

1. 发送数据到 *stdin* (如果 *input* 不为 None);
2. 从 *stdout* 和 *stderr* 读取数据，直至到达 EOF;
3. 等待进程终结。

可选的 *input* 参数为将被发送到子进程的数据 (*bytes* 对象)。

返回一个元组 (stdout_data, stderr_data)。

如果在将 *input* 写入到 *stdin* 时引发了*BrokenPipeError* 或*ConnectionResetError* 异常，异常会被忽略。此条件会在进程先于所有数据被写入到 *stdin* 之前退出时发生。

如果想要将数据发送到进程的 *stdin*，则创建进程时必须使用 stdin=PIPE。类似地，要在结果元组中获得任何不为 None 的值，则创建进程时必须使用 stdout=PIPE 和/或 stderr=PIPE 参数。

注意，数据读取在内存中是带缓冲的，因此如果数据量过大或不受则不要使用此方法。

send_signal(*signal*)
将信号 *signal* 发送给子进程。

備 F: 在 Windows 上，SIGTERM 是*terminate()* 的别名。CTRL_C_EVENT 和 CTRL_BREAK_EVENT 可被发送给创建时设置了 *creationflags* 形参且其中包括 CREATE_NEW_PROCESS_GROUP 的进程。

terminate()
停止子进程。

在 POSIX 系统中此方法会发送*signal.SIGTERM* 给子进程。

在 Windows 上会调用 Win32 API 函数 TerminateProcess() 以停止子进程。

kill()
杀掉子进程。

在 POSIX 系统中此方法会发送 SIGKILL 给子进程。

在 Windows 上此方法是*terminate()* 的别名。

stdin
标准输入流 (StreamWriter) 或者如果进程创建时设置了 stdin=None 则为 None。

stdout
标准输出流 (StreamReader) 或者如果进程创建时设置了 stdout=None 则为 None。

stderr
标准错误流 (StreamReader) 或者如果进程创建时设置了 stderr=None 则为 None。

> **警告：** Use the *communicate()* method rather than *process.stdin.write()*, *await process.stdout.read()* or *await process.stderr.read()*. This avoids deadlocks due to streams pausing reading or writing and blocking the child process.

pid
进程标识号（PID）。

注意对于由 Note that for processes created by the `create_subprocess_shell()` 函数所创建的进程，这个属性将是所生成的 shell 的 PID。

returncode
当进程退出时返回其代号。

`None` 值表示进程尚未终止。

一个负值 `-N` 表示子进程被信号 `N` 中断 (仅 POSIX).

子进程和线程

标准 asyncio 事件循环默认支持从不同线程中运行子进程。

在 Windows 上子进程（默认）只由*ProactorEventLoop* 提供，*SelectorEventLoop* 没有子进程支持。

在 UNIX 上会使用 *child watchers* 来让子进程结束等待，详情请参阅进程监视器。

3.8 版更變: UNIX 对于从不同线程中无限制地生成子进程会切换为使用*ThreadedChildWatcher*。

使用 *不活动的*当前子监视器生成子进程将引发*RuntimeError*。

请注意其他的事件循环实现可能有其本身的限制；请查看它们各自的文档。

也参考:

asyncio 中的并发和多线程 章节。

示例

一个使用*Process* 类来控制子进程并用*StreamReader* 类来从其标准输出读取信息的示例。

这个子进程是由*create_subprocess_exec()* 函数创建的:

```
import asyncio
import sys

async def get_date():
    code = 'import datetime; print(datetime.datetime.now())'

    # Create the subprocess; redirect the standard output
    # into a pipe.
    proc = await asyncio.create_subprocess_exec(
        sys.executable, '-c', code,
        stdout=asyncio.subprocess.PIPE)

    # Read one line of output.
    data = await proc.stdout.readline()
    line = data.decode('ascii').rstrip()
```

(下页继续)

(繼續上一頁)

```
    # Wait for the subprocess exit.
    await proc.wait()
    return line

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

另请参阅使用低层级 API 编写的相同示例。

18.1.5 队列集

源代码: Lib/asyncio/queues.py

---

asyncio 队列被设计成与*queue* 模块类似。尽管 asyncio 队列不是线程安全的，但是他们是被设计专用于 async/await 代码。

注意 asyncio 的队列没有 *timeout* 形参；请使用*asyncio.wait_for()* 函数为队列添加超时操作。

参见下面的*Examples* 部分

队列

class asyncio.**Queue**(*maxsize=0*, *, *loop=None*)

先进，先出（FIFO）队列

如果 *maxsize* 小于等于零，则队列尺寸是无限的。如果是大于 0 的整数，则当队列达到 *maxsize* 时，`await put()` 将阻塞至某个元素被*get()* 取出。

不像标准库中的并发型*queue* ，队列的尺寸一直是已知的，可以通过调用*qsize()* 方法返回。

Deprecated since version 3.8, will be removed in version 3.10: *loop* 形参。

这个类*不是线程安全的*。

maxsize

队列中可存放的元素数量。

empty()

如果队列为空返回 `True` ，否则返回 `False` 。

full()

如果有*maxsize* 个条目在队列中，则返回 `True` 。

如果队列用 `maxsize=0` （默认）初始化，则*full()* 永远不会返回 `True` 。

coroutine get()

从队列中删除并返回一个元素。如果队列为空，则等待，直到队列中有元素。

get_nowait()

立即返回一个队列中的元素，如果队列内有值，否则引发异常*QueueEmpty* 。

coroutine join()

阻塞至队列中所有的元素都被接收和处理完毕。

当条目添加到队列的时候，未完成任务的计数就会增加。每当消费协程调用*task_done()* 表示这个条目已经被回收，该条目所有工作已经完成，未完成计数就会减少。当未完成计数降到零的时候，*join()* 阻塞被解除。

coroutine put(*item*)
添加一个元素进队列。如果队列满了，在添加元素之前，会一直等待空闲插槽可用。

put_nowait(*item*)
不阻塞的放一个元素入队列。

如果没有立即可用的空闲槽，引发*QueueFull* 异常。

qsize()
返回队列用的元素数量。

task_done()
表明前面排队的任务已经完成，即 get 出来的元素相关操作已经完成。

由队列使用者控制。每个*get()* 用于获取一个任务，任务最后调用*task_done()* 告诉队列，这个任务已经完成。

如果*join()* 当前正在阻塞，在所有条目都被处理后，将解除阻塞 (意味着每个*put()* 进队列的条目的*task_done()* 都被收到)。

如果被调用的次数多于放入队列中的项目数量，将引发*ValueError* 。

优先级队列

class asyncio.**PriorityQueue**
Queue 的变体；按优先级顺序取出条目 (最小的先取出)。

条目通常是 (priority_number, data) 形式的元组。

后进先出队列

class asyncio.**LifoQueue**
Queue 的变体，先取出最近添加的条目（后进，先出）。

异常

exception asyncio.**QueueEmpty**
当队列为空的时候，调用*get_nowait()* 方法而引发这个异常。

exception asyncio.**QueueFull**
当队列中条目数量已经达到它的 *maxsize* 的时候，调用*put_nowait()* 方法而引发的异常。

示例

队列能被用于多个的并发任务的工作量分配：

```
import asyncio
import random
import time


async def worker(name, queue):
    while True:
        # Get a "work item" out of the queue.
        sleep_for = await queue.get()
```

(下页继续)

(繼續上一頁)

```
        # Sleep for the "sleep_for" seconds.
        await asyncio.sleep(sleep_for)

        # Notify the queue that the "work item" has been processed.
        queue.task_done()

        print(f'{name} has slept for {sleep_for:.2f} seconds')


async def main():
    # Create a queue that we will use to store our "workload".
    queue = asyncio.Queue()

    # Generate random timings and put them into the queue.
    total_sleep_time = 0
    for _ in range(20):
        sleep_for = random.uniform(0.05, 1.0)
        total_sleep_time += sleep_for
        queue.put_nowait(sleep_for)

    # Create three worker tasks to process the queue concurrently.
    tasks = []
    for i in range(3):
        task = asyncio.create_task(worker(f'worker-{i}', queue))
        tasks.append(task)

    # Wait until the queue is fully processed.
    started_at = time.monotonic()
    await queue.join()
    total_slept_for = time.monotonic() - started_at

    # Cancel our worker tasks.
    for task in tasks:
        task.cancel()
    # Wait until all worker tasks are cancelled.
    await asyncio.gather(*tasks, return_exceptions=True)

    print('====')
    print(f'3 workers slept in parallel for {total_slept_for:.2f} seconds')
    print(f'total expected sleep time: {total_sleep_time:.2f} seconds')


asyncio.run(main())
```

18.1.6 异常

源代码: Lib/asyncio/exceptions.py

---

exception asyncio.**TimeoutError**
该操作已超过规定的截止日期。

> **重要:** 这个异常与内置*TimeoutError*异常不同。

exception asyncio.**CancelledError**
该操作已被取消。

取消 asyncio 任务时，可以捕获此异常以执行自定义操作。在几乎所有情况下，都必须重新引发异常。

3.8 版更變: *CancelledError* 现在是*BaseException* 的子类。

exception asyncio.**InvalidStateError**
Task 或*Future* 的内部状态无效。

在为已设置结果值的未来对象设置结果值等情况下，可以引发此问题。

exception asyncio.**SendfileNotAvailableError**
”sendfile” 系统调用不适用于给定的套接字或文件类型。

子类*RuntimeError* 。

exception asyncio.**IncompleteReadError**
请求的读取操作未完全完成。

由*asyncio stream APIs* 提出

此异常是*EOFError* 的子类。

expected
预期字节的总数（*int* ）。

partial
到达流结束之前读取的*bytes* 字符串。

exception asyncio.**LimitOverrunError**
在查找分隔符时达到缓冲区大小限制。

由*asyncio stream APIs* 提出

consumed
要消耗的字节总数。

18.1.7 事件循环

源代码: Lib/asyncio/events.py, Lib/asyncio/base_events.py

前言

事件循环是每个 asyncio 应用的核心。事件循环会运行异步任务和回调，执行网络 IO 操作，以及运行子进程。

应用开发者通常应当使用高层级的 asyncio 函数，例如*asyncio.run()*，应当很少有必要引用循环对象或调用其方法。本节所针对的主要是低层级代码、库和框架的编写者，他们需要更细致地控制事件循环行为。

获取事件循环

以下低层级函数可被用于获取、设置或创建事件循环:

asyncio.**get_running_loop**()
返回当前 OS 线程中正在运行的事件循环。

如果没有正在运行的事件循环则会引发*RuntimeError*。此函数只能由协程或回调来调用。

3.7 版新加入.

asyncio.**get_event_loop**()
获取当前事件循环。

如果当前 OS 线程没有设置当前事件循环，该 OS 线程为主线程，并且*set_event_loop()* 还没有被调用，则 asyncio 将创建一个新的事件循环并将其设为当前事件循环。

由于此函数具有相当复杂的行为（特别是在使用了自定义事件循环策略的时候），更推荐在协程和回调中使用*get_running_loop()* 函数而非*get_event_loop()*。

应该考虑使用*asyncio.run()* 函数而非使用低层级函数来手动创建和关闭事件循环。

asyncio.**set_event_loop**(*loop*)
将 *loop* 设置为当前 OS 线程的当前事件循环。

asyncio.**new_event_loop**()
Create and return a new event loop object.

请注意*get_event_loop()*, *set_event_loop()* 以及*new_event_loop()* 函数的行为可以通过设置自定义事件循环策略 来改变。

内容

本文档包含下列小节:

- 事件循环方法集 章节是事件循环 APIs 的参考文档;
- 回调处理 章节是从调度方法例如*loop.call_soon()* 和*loop.call_later()* 中返回*Handle* 和*TimerHandle* 实例的文档。
- *Server Objects* 章节记录了从事件循环方法返回的类型，比如*loop.create_server()*；
- *Event Loop Implementations* 章节记录了*SelectorEventLoop* 和*ProactorEventLoop* 类;
- *Examples* 章节展示了如何使用某些事件循环 API。

事件循环方法集

事件循环有下列 **低级** APIs:

- 运行和停止循环
- 安排回调
- 调度延迟回调
- 创建 *Future* 和 *Task*
- 打开网络连接

运行和停止循环

`loop.`**`run_until_complete`**`(`*`future`*`)`
运行直到 *future* (`Future` 的实例) 被完成。

如果参数是*coroutine object* ，将被隐式调度为`asyncio.Task` 来运行。

返回 Future 的结果或者引发相关异常。

`loop.`**`run_forever`**`()`
运行事件循环直到`stop()` 被调用。

如果`stop()` 在调用`run_forever()` 之前被调用，循环将轮询一次 I/O 选择器并设置超时为零，再运行所有已加入计划任务的回调来响应 I/O 事件（以及已加入计划任务的事件），然后退出。

如果`stop()` 在`run_forever()` 运行期间被调用，循环将运行当前批次的回调然后退出。请注意在此情况下由回调加入计划任务的新回调将不会运行；它们将会在下次`run_forever()` 或`run_until_complete()` 被调用时运行。

`loop.`**`stop`**`()`
停止事件循环。

`loop.`**`is_running`**`()`
返回 `True` 如果事件循环当前正在运行。

`loop.`**`is_closed`**`()`
如果事件循环已经被关闭，返回 `True` 。

`loop.`**`close`**`()`
关闭事件循环。

当这个函数被调用的时候，循环必须处于非运行状态。pending 状态的回调将被丢弃。

此方法清除所有的队列并立即关闭执行器，不会等待执行器完成。

这个方法是幂等的和不可逆的。事件循环关闭后，不应调用其他方法。

`coroutine` `loop.`**`shutdown_asyncgens`**`()`
安排所有当前打开的*asynchronous generator* 对象通过 `aclose()` 调用来关闭。在调用此方法后，如果有

新的异步生成器被迭代事件循环将会发出警告。这应当被用来可靠地完成所有已加入计划任务的异步生成器。

请注意当使用*asyncio.run()* 时不必调用此函数。

示例:

```
try:
    loop.run_forever()
finally:
    loop.run_until_complete(loop.shutdown_asyncgens())
    loop.close()
```

3.6 版新加入.

coroutine loop.**shutdown_default_executor**()
安排默认执行器的关闭并等待它合并 ThreadPoolExecutor 中的所有线程。在调用此方法后，如果在使用默认执行器期间调用了*loop.run_in_executor()* 则将会引发*RuntimeError*。

请注意当使用*asyncio.run()* 时不必调用此函数。

3.9 版新加入.

安排回调

loop.**call_soon**(*callback*, **args*, *context=None*)
安排 *callback* *callback* 在事件循环的下一次迭代时附带 *args* 参数被调用。

回调按其注册顺序被调用。每个回调仅被调用一次。

可选的仅关键字型参数 *context* 允许为要运行的 *callback* 指定一个自定义*contextvars.Context* 。如果没有提供 *context* ，则使用当前上下文。

返回一个能用来取消回调的*asyncio.Handle* 实例。

这个方法不是线程安全的。

loop.**call_soon_threadsafe**(*callback*, **args*, *context=None*)
call_soon() 的线程安全变体。必须被用于安排 *来自其他线程*的回调。

如果在已被关闭的循环上调用则会引发*RuntimeError*。这可能会在主应用程序被关闭时在二级线程上发生。

查看*并发和多线程* 章节的文档。

3.7 版更變: 仅用于关键字形参的参数 *context* 已经被添加。请参阅：**PEP 567** 查看更多细节。

備🄵: 大多数*asyncio* 的调度函数不让传递关键字参数。为此，请使用*functools.partial()* ：

```
# will schedule "print("Hello", flush=True)"
loop.call_soon(
    functools.partial(print, "Hello", flush=True))
```

使用 partial 对象通常比使用 lambda 更方便，asyncio 在调试和错误消息中能更好的呈现 partial 对象。

调度延迟回调

事件循环提供安排调度函数在将来某个时刻调用的机制。事件循环使用单调时钟来跟踪时间。

`loop.call_later`(*delay*, *callback*, **args*, *context=None*)
安排 *callback* 在给定的 *delay* 秒（可以是 int 或者 float）后被调用。

返回一个`asyncio.TimerHandle`实例，该实例能用于取消回调。

callback 只被调用一次。如果两个回调被安排在同样的时间点，执行顺序未限定。

可选的位置参数 *args* 在被调用的时候传递给 *callback* 。如果你想把关键字参数传递给 *callback* ，请使用`functools.partial()` 。

可选的仅关键字型参数 *context* 允许为要运行的 *callback* 指定一个自定义`contextvars.Context` 。如果没有提供 *context* ，则使用当前上下文。

3.7 版更變: 仅用于关键字形参的参数 *context* 已经被添加。请参阅： **PEP 567** 查看更多细节。

3.8 版更變: 在 Python 3.7 和更早版本的默认事件循环实现中，*delay* 不能超过一天。这在 Python 3.8 中已被修复。

`loop.call_at`(*when*, *callback*, **args*, *context=None*)
安排 *callback* 在给定的绝对时间戳 *when* (int 或 float) 被调用，使用与`loop.time()` 同样的时间参考。

这个函数的行为与`call_later()` 相同。

返回一个`asyncio.TimerHandle`实例，该实例能用于取消回调。

3.7 版更變: 仅用于关键字形参的参数 *context* 已经被添加。请参阅： **PEP 567** 查看更多细节。

3.8 版更變: 在 Python 3.7 和更早版本的默认事件循环实现中，*when* 和当前时间相差不能超过一天。在这 Python 3.8 中已被修复。

`loop.time`()
根据时间循环内部的单调时钟，返回当前时间为一个`float` 值。

備🄵: 3.8 版更變: 在 Python 3.7 和更早版本中超时 (相对的 *delay* 或绝对的 *when*) 不能超过一天。这在 Python 3.8 中已被修复。

也参考:

`asyncio.sleep()` 函数

创建 Future 和 Task

`loop.create_future`()
创建一个附加到事件循环中的`asyncio.Future`对象。

这是在 asyncio 中创建 Futures 的首选方式。这让第三方事件循环可以提供 Future 对象的替代实现 (更好的性能或者功能)。

3.5.2 版新加入.

`loop.create_task`(*coro*, ***, *name=None*)
安排一个协程 的执行。返回一个`Task`对象。

三方的事件循环可以使用它们自己定义的`Task`类的子类来实现互操作性。这个例子里，返回值的类型是`Task`的子类。

如果提供了 *name* 参数且不为 `None`，它会使用`Task.set_name()` 来设为任务的名称。

3.8 版更變: 添加了 name 形参。

loop.**set_task_factory**(*factory*)
设置一个任务工厂，它将由*loop.create_task()* 来使用。

如果 *factory* 为 None 则将设置默认的任务工厂。在其他情况下，*factory* 必须为一个 可调用对象且签名匹配 (loop, coro)，其中 *loop* 是对活动事件循环的引用，而 *coro* 是一个协程对象。该可调用对象必须返回一个兼容*asyncio.Future* 的对象。

loop.**get_task_factory**()
返回一个任务工厂，或者如果是使用默认值则返回 None。

打开网络连接

coroutine loop.**create_connection**(*protocol_factory*, *host=None*, *port=None*, ***, *ssl=None*, *family=0*, *proto=0*, *flags=0*, *sock=None*, *local_addr=None*, *server_hostname=None*, *ssl_handshake_timeout=None*, *happy_eyeballs_delay=None*, *interleave=None*)
打开一个流式传输连接，连接到由 *host* 和 *port* 指定的地址。

套接字族可以是*AF_INET* 或*AF_INET6*，具体取决于 *host* (或 *family* 参数，如果有提供的话)。

套接字类型将为*SOCK_STREAM*。

protocol_factory 必须为一个返回*asyncio 协议* 实现的可调用对象。

这个方法会尝试在后台创建连接。当创建成功，返回 (transport, protocol) 组合。

基本操作的时间顺序如下:

1. 创建连接并为其创建一个传输。
2. 不带参数地调用 *protocol_factory* 并预期返回一个协议 实例。
3. 协议实例通过调用其*connection_made()* 方法与传输进行配对。
4. 成功时返回一个 (transport, protocol) 元组。

创建的传输是一个具体实现相关的双向流。

其他参数:

- *ssl*: 如果给定该参数且不为假值，则会创建一个 SSL/TLS 传输（默认创建一个纯 TCP 传输）。如果 *ssl* 是一个*ssl.SSLContext* 对象，则会使用此上下文来创建传输对象；如果 *ssl* 为*True*，则会使用从*ssl.create_default_context()* 返回的默认上下文。

  也参考:

  SSL/TLS 安全事项

- *server_hostname* 设置或重载目标服务器的证书将要匹配的主机名。应当只在 *ssl* 不为 None 时传入。默认情况下会使用 *host* 参数的值。如果 *host* 为空那就没有默认值，你必须为 *server_hostname* 传入一个值。如果 *server_hostname* 为空字符串，则主机名匹配会被禁用（这是一个严重的安全风险，使得潜在的中间人攻击成为可能）。

- *family*，*proto*，*flags* 是可选的地址族，协议和标志，通过传递给 getaddrinfo() 来解析 *host*。如果给出，这些应该都是来自*socket* 模块相应的常量的整数。

- 如果给出 *happy_eyeballs_delay*，它将为此链接启用 Happy Eyeballs。该函数应当为一个表示在开始下一个并行尝试之前要等待连接尝试完成的秒数的浮点数。这也就是在 **RFC 8305** 中定义的" 连接尝试延迟"。该 RFC 所推荐的合理默认值为 0.25 (250 毫秒)。

- *interleave* 控制当主机名解析为多个 IP 地址时的地址重排序。如果该参数为 `0` 或未指定，则不会进行重排序，这些地址会按*getaddrinfo()* 所返回的顺序进行尝试。如果指定了一个正整数，这些地址会按地址族交错排列，而指定的整数会被解读为 **RFC 8305** 所定义的" 首个地址族计数"。如果 *happy_eyeballs_delay* 未指定则默认值为 `0`，否则为 `1`。
- 如果给出 *sock*，它应当是一个已存在、已连接并将被传输所使用的*socket.socket* 对象。如果给出了 *sock*，则 *host*, *port*, *family*, *proto*, *flags*, *happy_eyeballs_delay*, *interleave* 和 *local_addr* 都不应当被指定。
- 如果给出 *local_addr*，它应当是一个用来在本地绑定套接字的 `(local_host, local_port)` 元组。*local_host* 和 *local_port* 会使用 `getaddrinfo()` 来查找，这与 *host* 和 *port* 类似。
- *ssl_handshake_timeout* 是（用于 TLS 连接的）在放弃连接之前要等待 TLS 握手完成的秒数。如果参数为 `None` 则使用 (默认的) `60.0`。

3.8 版新加入: 增加了 *happy_eyeballs_delay* 和 *interleave* 形参。

Happy Eyeballs 算法：成功使用双栈主机。当服务器的 IPv4 路径和协议工作正常，但服务器的 IPv6 路径和协议工作不正常时，双栈客户端应用程序相比单独 IPv4 客户端会感受到明显的连接延迟。这是不可接受的因为它会导致双栈客户端糟糕的用户体验。此文档指明了减少这种用户可见延迟的算法要求并提供了具体的算法。

详情参见: https://tools.ietf.org/html/rfc6555

3.7 版新加入: *ssl_handshake_timeout* 形参。

3.6 版更變: 套接字选项 `TCP_NODELAY` 默认已为所有 TCP 连接进行了设置。

3.5 版更變: *ProactorEventLoop* 类中添加 SSL/TLS 支持。

也参考:

open_connection() 函数是一个高层级的替代 API。它返回一对 (*StreamReader*, *StreamWriter*)，可在 async/await 代码中直接使用。

coroutine loop.**create_datagram_endpoint**(*protocol_factory*, *local_addr=None*, *remote_addr=None*, ***, *family=0*, *proto=0*, *flags=0*, *reuse_address=None*, *reuse_port=None*, *allow_broadcast=None*, *sock=None*)

備𝔽: 形参 *reuse_address* 已不再受支持，因为使用 `SO_REUSEADDR` 会对 UDP 造成显著的安全问题。显式地传入 `reuse_address=True` 将会引发异常。

当具有不同 UID 的多个进程将套接字赋给具有 `SO_REUSEADDR` 的相同 UDP 套接字地址时，传入的数据包可能会在套接字间随机分配。

对于受支持的平台，*reuse_port* 可以被用作类似功能的替代。通过 *reuse_port* 将改用 `SO_REUSEPORT`，它能够防止具有不同 UID 的进程将套接字赋给相同的套接字地址。

创建一个数据报连接。

套接字族可以是*AF_INET*, *AF_INET6* 或*AF_UNIX*，具体取决于 *host* (或 *family* 参数，如果有提供的话)。

socket 类型将是*SOCK_DGRAM* 。

protocol_factory 必须为一个返回协议 实现的可调用对象。

成功时返回一个 `(transport, protocol)` 元组。

其他参数:

- 如果给出 *local_addr*，它应当是一个用来在本地绑定套接字的 `(local_host, local_port)` 元组。*local_host* 和 *local_port* 是使用*getaddrinfo()* 来查找的。
- *remote_addr*，如果指定的话，就是一个 `(remote_host, remote_port)` 元组，用于同一个远程地址连接。*remote_host* 和 *remote_port* 是使用*getaddrinfo()* 来查找的。
- *family*, *proto*, *flags* 是可选的地址族，协议和标志，其会被传递给*getaddrinfo()* 来完成 *host* 的解析。如果要指定的话，这些都应该是来自于*socket* 模块的对应常量。
- *reuse_port* 告知内核，只要在创建时都设置了这个旗标，就允许此端点绑定到其他现有端点所绑定的相同端口上。这个选项在 Windows 和某些 Unix 上不受支持。如果 `SO_REUSEPORT` 常量未定义则此功能就是不受支持的。
- *allow_broadcast* 告知内核允许此端点向广播地址发送消息。
- *sock* 可选择通过指定此值用于使用一个预先存在的，已经处于连接状态的*socket.socket* 对象，并将其提供给此传输对象使用。如果指定了这个值，*local_addr* 和 *remote_addr* 就应该被忽略 (必须为*None*)。

参见*UDP echo 客户端协议* 和*UDP echo 服务端协议* 的例子。

3.4.4 版更變: 添加了 *family*, *proto*, *flags*, *reuse_address*, *reuse_port*, *allow_broadcast* 和 *sock* 等参数。

3.8.1 版更變: 出于安全考虑，*reuse_address* 形参已不再受支持。

3.8 版更變: 添加 WIndows 的支持。

coroutine loop.**create_unix_connection**(*protocol_factory*, *path=None*, ***, *ssl=None*, *sock=None*, *server_hostname=None*, *ssl_handshake_timeout=None*)

创建 Unix 连接

套接字族将为*AF_UNIX*；套接字类型将为*SOCK_STREAM*。

成功时返回一个 `(transport, protocol)` 元组。

path 是所要求的 Unix 域套接字的名字，除非指定了 *sock* 形参。抽象的 Unix 套接字，*str*，*bytes* 和*Path* 路径都是受支持的。

请查看*loop.create_connection()* 方法的文档了解有关此方法的参数的信息。

可用性: Unix。

3.7 版新加入: *ssl_handshake_timeout* 形参。

3.7 版更變: *path* 形参现在可以是*path-like object* 对象。

创建网络服务

coroutine loop.**create_server**(*protocol_factory*, *host=None*, *port=None*, ***, *family=socket.AF_UNSPEC*, *flags=socket.AI_PASSIVE*, *sock=None*, *backlog=100*, *ssl=None*, *reuse_address=None*, *reuse_port=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

创建 TCP 服务 (socket 类型*SOCK_STREAM*) 监听 *host* 地址的 *port* 端口。

返回一个*Server* 对象。

参数:

- *protocol_factory* 必须为一个返回*协议* 实现的可调用对象。
- *host* 形参可被设为几种类型，它确定了服务器所应监听的位置:
  - 如果 *host* 是一个字符串，则 TCP 服务器会被绑定到 *host* 所指明的单一网络接口。

  - 如果 *host* 是一个字符串序列，则 TCP 服务器会被绑定到序列所指明的所有网络接口。
  - 如果 *host* 是一个空字符串或 `None`，则会应用所有接口并将返回包含多个套接字的列表（通常是一个 IPv4 的加一个 IPv6 的）。
- The *port* parameter can be set to specify which port the server should listen on. If `0` or `None` (the default), a random unused port will be selected (note that if *host* resolves to multiple network interfaces, a different random port will be selected for each interface).
- *family* 可被设为*socket.AF_INET* 或*AF_INET6* 以强制此套接字使用 IPv4 或 IPv6。如果未设定，则 *family* 将通过主机名称来确定 (默认为 `AF_UNSPEC`)。
- *flags* 是用于*getaddrinfo()* 的位掩码。
- 可以选择指定 *sock* 以便使用预先存在的套接字对象。如果指定了此参数，则不可再指定 *host* 和 *port*。
- *backlog* 是传递给*listen()* 的最大排队连接的数量（默认为 100）。
- *ssl* 可被设置为一个*SSLContext* 实例以在所接受的连接上启用 TLS。
- *reuse_address* 告知内核要重用一个处于 `TIME_WAIT` 状态的本地套接字，而不是等待其自然超时失效。如果未指定此参数则在 Unix 上将自动设置为 `True`。
- *reuse_port* 告知内核，只要在创建的时候都设置了这个标志，就允许此端点绑定到其它端点列表所绑定的同样的端口上。这个选项在 Windows 上是不支持的。
- *ssl_handshake_timeout* 是（用于 TLS 服务器的）在放弃连接之前要等待 TLS 握手完成的秒数。如果参数为 (默认值) `None` 则为 `60.0` 秒。
- *start_serving* 设置成 `True` (默认值) 会导致创建 server 并立即开始接受连接。设置成 `False`，用户需要等待*Server.start_serving()* 或者*Server.serve_forever()* 以使 server 开始接受连接。

3.7 版新加入: 增加了 *ssl_handshake_timeout* 和 *start_serving* 形参。

3.6 版更變: 套接字选项 `TCP_NODELAY` 默认已为所有 TCP 连接进行了设置。

3.5 版更變: *ProactorEventLoop* 类中添加 SSL/TLS 支持。

3.5.1 版更變: *host* 形参可以是一个字符串的序列。

也參考:

start_server() 函数是一个高层级的替代 API，它返回一对*StreamReader* 和*StreamWriter*，可在 async/await 代码中使用。

coroutine loop.**create_unix_server**(*protocol_factory*, *path=None*, ***, *sock=None*, *backlog=100*, *ssl=None*, *ssl_handshake_timeout=None*, *start_serving=True*)

与*loop.create_server()* 类似但是专用于*AF_UNIX* 套接字族。

path 是必要的 Unix 域套接字名称，除非提供了 *sock* 参数。抽象的 Unix 套接字, *str*, *bytes* 和*Path* 路径都是受支持的。

请查看*loop.create_server()* 方法的文档了解有关此方法的参数的信息。

可用性: Unix。

3.7 版新加入: The *ssl_handshake_timeout* and *start_serving* parameters.

3.7 版更變: *path* 形参现在可以是*Path* 对象。

coroutine loop.**connect_accepted_socket**(*protocol_factory*, *sock*, ***, *ssl=None*, *ssl_handshake_timeout=None*)

将已被接受的连接包装成一个传输/协议对。

此方法可被服务器用来接受 asyncio 以外的连接，但是使用 asyncio 来处理它们。

参数：

- *protocol_factory* 必须为一个返回协议 实现的可调用对象。
- *sock* 是一个预先存在的套接字对象，它是由`socket.accept` 返回的。
- *ssl* 可被设置为一个`SSLContext` 以在接受的连接上启用 SSL。
- *ssl_handshake_timeout* 是 (为一个 SSL 连接) 在中止连接前，等待 SSL 握手完成的时间【单位秒】。如果为 `None` (缺省) 则是 `60.0` 秒。

返回一个 `(transport, protocol)` 对。

3.7 版新加入: *ssl_handshake_timeout* 形参。

3.5.3 版新加入.

传输文件

coroutine loop.**sendfile**(*transport*, *file*, *offset=0*, *count=None*, *, *fallback=True*)

将 *file* 通过 *transport* 发送。返回所发送的字节总数。

如果可用的话，该方法将使用高性能的`os.sendfile()`。

file 必须是个二进制模式打开的常规文件对象。

offset 指明从何处开始读取文件。如果指定了 *count*，它是要传输的字节总数而不再一直发送文件直至抵达 EOF。文件位置总是会被更新，即使此方法引发了错误，并可以使用`file.tell()` 来获取实际发送的字节总数。

fallback 设为 `True` 会使得 asyncio 在平台不支持 sendfile 系统调用时手动读取并发送文件（例如 Windows 或 Unix 上的 SSL 套接字）。

如果系统不支持 *sendfile* 系统调用且 *fallback* 为 `False` 则会引发`SendfileNotAvailableError`。

3.7 版新加入.

TLS 升级

coroutine loop.**start_tls**(*transport*, *protocol*, *sslcontext*, *, *server_side=False*, *server_hostname=None*, *ssl_handshake_timeout=None*)

将现有基于传输的连接升级到 TLS。

返回一个新的传输实例，其中 *protocol* 必须在 *await* 之后立即开始使用。传给 *start_tls* 方法的 *transport* 实例应永远不会被再次使用。

参数：

- *transport* 和 *protocol* 实例的方法与`create_server()` 和`create_connection()` 所返回的类似。
- *sslcontext*： 一个已经配置好的`SSLContext` 实例。
- 当服务端连接已升级时 (如`create_server()` 所创建的对象) *server_side* 会传入 `True`。
- *server_hostname*： 设置或者覆盖目标服务器证书中相对应的主机名。
- *ssl_handshake_timeout* 是（用于 TLS 连接的）在放弃连接之前要等待 TLS 握手完成的秒数。如果参数为 `None` 则使用 (默认的) `60.0`。

3.7 版新加入.

监控文件描述符

`loop.`**`add_reader`**(*fd*, *callback*, **args*)
开始监视 *fd* 文件描述符以获取读取的可用性，一旦 *fd* 可用于读取，使用指定的参数调用 *callback* 。

`loop.`**`remove_reader`**(*fd*)
停止对文件描述符 *fd* 读取可用性的监视。

`loop.`**`add_writer`**(*fd*, *callback*, **args*)
开始监视 *fd* 文件描述符的写入可用性，一旦 *fd* 可用于写入，使用指定的参数调用 *callback* 。

使用*functools.partial() 传递关键字参数* 给 *callback*.

`loop.`**`remove_writer`**(*fd*)
停止对文件描述符 *fd* 的写入可用性监视。

另请查看*平台支持* 一节了解以上方法的某些限制。

直接使用 socket 对象

通常，使用基于传输的 API 的协议实现，例如*loop.create_connection()* 和*loop.create_server()* 比直接使用套接字的实现更快。但是，在某些应用场景下性能并不非常重要，直接使用*socket* 对象会更方便。

coroutine `loop.`**`sock_recv`**(*sock*, *nbytes*)
从 *sock* 接收至多 *nbytes*。*socket.recv()* 的异步版本。

返回接收到的数据【bytes 对象类型】。

sock 必须是个非阻塞 socket。

3.7 版更變: 虽然这个方法总是被记录为协程方法，但它在 Python 3.7 之前的发行版中会返回一个*Future*。从 Python 3.7 开始它则是一个 `async def` 方法。

coroutine `loop.`**`sock_recv_into`**(*sock*, *buf*)
从 *sock* 接收数据放入 *buf* 缓冲区。模仿了阻塞型的*socket.recv_into()* 方法。

返回写入缓冲区的字节数。

sock 必须是个非阻塞 socket。

3.7 版新加入.

coroutine `loop.`**`sock_sendall`**(*sock*, *data*)
将 *data* 发送到 *sock* 套接字。*socket.sendall()* 的异步版本。

此方法会持续发送数据到套接字直至 *data* 中的所有数据发送完毕或是有错误发生。当成功时会返回 `None`。当发生错误时，会引发一个异常。此外，没有办法能确定有多少数据或是否有数据被连接的接收方成功处理。

sock 必须是个非阻塞 socket。

3.7 版更變: 虽然这个方法一直被标记为协程方法。但是，Python 3.7 之前，该方法返回*Future* ，从 Python 3.7 开始，这个方法是 `async def` 方法。

coroutine `loop.`**`sock_connect`**(*sock*, *address*)
将 *sock* 连接到位于 *address* 的远程套接字。

socket.connect() 的异步版本。

sock 必须是个非阻塞 socket。

3.5.2 版更變: address 不再需要被解析。sock_connect 将尝试检查 *address* 是否已通过调用 *socket.inet_pton()* 被解析。如果没有，则将使用 *loop.getaddrinfo()* 来解析 *address*。

也参考:

loop.create_connection() 和 *asyncio.open_connection()* 。

coroutine loop.**sock_accept**(*sock*)
接受一个连接。模仿了阻塞型的 *socket.accept()* 方法。

此 *scoket* 必须绑定到一个地址上并且监听连接。返回值是一个 (conn, address) 对，其中 *conn* 是一个 新 * 的套接字对象，用于在此连接上收发数据，*address* 是连接的另一端的套接字所绑定的地址。

sock 必须是个非阻塞 socket。

3.7 版更變: 虽然这个方法一直被标记为协程方法。但是，Python 3.7 之前，该方法返回 *Future* ，从 Python 3.7 开始，这个方法是 async def 方法。

也参考:

loop.create_server() 和 *start_server()*。

coroutine loop.**sock_sendfile**(*sock*, *file*, *offset=0*, *count=None*, *, *fallback=True*)
在可能的情况下使用高性能的 *os.sendfile* 发送文件。返回所发送的字节总数。

socket.sendfile() 的异步版本。

sock 必须为非阻塞型的 *socket.SOCK_STREAM socket*。

file 必须是个用二进制方式打开的常规文件对象。

offset 指明从何处开始读取文件。如果指定了 *count*，它是要传输的字节总数而不再一直发送文件直至抵达 EOF。文件位置总是会被更新，即使此方法引发了错误，并可以使用 *file.tell()* 来获取实际发送的字节总数。

当 *fallback* 被设为 True 时，会使用 asyncio 在平台不支持 sendfile 系统调用时手动读取并发送文件（例如 Windows 或 Unix 上的 SSL 套接字）。

如果系统不支持 *sendfile* 并且 *fallback* 为 False ，引发 *SendfileNotAvailableError* 异常。

sock 必须是个非阻塞 socket。

3.7 版新加入.

DNS

coroutine loop.**getaddrinfo**(*host*, *port*, *, *family=0*, *type=0*, *proto=0*, *flags=0*)
异步版的 *socket.getaddrinfo()* 。

coroutine loop.**getnameinfo**(*sockaddr*, *flags=0*)
异步版的 *socket.getnameinfo()* 。

3.7 版更變: *getaddrinfo* 和 *getnameinfo* 方法一直被标记返回一个协程，但是 Python 3.7 之前，实际返回的是 *asyncio.Future* 对象。从 Python 3.7 开始，这两个方法是协程。

使用管道

coroutine loop.**connect_read_pipe**(*protocol_factory*, *pipe*)
在事件循环中注册 *pipe* 的读取端。

protocol_factory 必须为一个返回*asyncio 协议* 实现的可调用对象。

pipe 是个*类似文件型对象*.

返回一对 (transport, protocol)，其中 *transport* 支持*ReadTransport* 接口而 *protocol* 是由 *protocol_factory* 所实例化的对象。

使用*SelectorEventLoop* 事件循环，*pipe* 被设置为非阻塞模式。

coroutine loop.**connect_write_pipe**(*protocol_factory*, *pipe*)
在事件循环中注册 *pipe* 的写入端。

protocol_factory 必须为一个返回*asyncio 协议* 实现的可调用对象。

pipe 是个*类似文件型对象*.

返回一对 (transport, protocol)，其中 *transport* 支持*WriteTransport* 接口而 *protocol* 是由 *protocol_factory* 所实例化的对象。

使用*SelectorEventLoop* 事件循环，*pipe* 被设置为非阻塞模式。

備🄴: 在 Windows 中*SelectorEventLoop* 不支持上述方法。对于 Windows 请改用*ProactorEventLoop*。

也参考:

loop.subprocess_exec() 和*loop.subprocess_shell()* 方法。

Unix 信号

loop.**add_signal_handler**(*signum*, *callback*, **args*)
设置 *callback* 作为 *signum* 信号的处理程序。

此回调将与该事件循环中其他加入队列的回调和可运行协程一起由 *loop* 发起调用。不同与使用*signal.signal()* 注册的信号处理程序，使用此函数注册的回调可以与事件循环进行交互。

如果信号数字非法或者不可捕获，就抛出一个*ValueError* 。如果建立处理器的过程中出现问题，会抛出一个*RuntimeError* 。

使用*functools.partial() 传递关键字参数* 给 *callback*.

和*signal.signal()* 一样，这个函数只能在主线程中调用。

loop.**remove_signal_handler**(*sig*)
移除 *sig* 信号的处理程序。

如果信号处理程序被移除则返回 True，否则如果给定信号未设置处理程序则返回 False。

可用性: Unix。

也参考:

signal 模块。

在线程或者进程池中执行代码。

awaitable loop.**run_in_executor**(*executor*, *func*, **args*)
安排在指定的执行器中调用 *func* 。

The *executor* argument should be an *concurrent.futures.Executor* instance. The default executor is used if *executor* is None.

示例:

```
import asyncio
import concurrent.futures

def blocking_io():
    # File operations (such as logging) can block the
    # event loop: run them in a thread pool.
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    # CPU-bound operations will block the event loop:
    # in general it is preferable to run them in a
    # process pool.
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    ## Options:

    # 1. Run in the default loop's executor:
    result = await loop.run_in_executor(
        None, blocking_io)
    print('default thread pool', result)

    # 2. Run in a custom thread pool:
    with concurrent.futures.ThreadPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, blocking_io)
        print('custom thread pool', result)

    # 3. Run in a custom process pool:
    with concurrent.futures.ProcessPoolExecutor() as pool:
        result = await loop.run_in_executor(
            pool, cpu_bound)
        print('custom process pool', result)

asyncio.run(main())
```

这个方法返回一个*asyncio.Future* 对象。

使用*functools.partial() 传递关键字参数* 给 *func* 。

3.5.3 版更變: *loop.run_in_executor()* 不会再配置它所创建的线程池执行器的 max_workers，而是将其留给线程池执行器 (*ThreadPoolExecutor*) 来设置默认值。

loop.**set_default_executor**(*executor*)
将 *executor* 设为*run_in_executor()* 所使用的默认执行器。*executor* 应当是*ThreadPoolExecutor* 的实例。

3.8 版後已🄴用: 使用不是 Using an executor that is not an instance of *ThreadPoolExecutor* 实例的执行器的做法已被弃用并将在 Python 3.9 中引起错误。

executor 必须是个*concurrent.futures.ThreadPoolExecutor* 的实例。

错误处理 API

允许自定义事件循环中如何去处理异常。

`loop.`**`set_exception_handler`**`(`*handler*`)`
将 *handler* 设置为新的事件循环异常处理器。

如果 *handler* 为 `None`，将设置默认的异常处理程序。在其他情况下，*handler* 必须是一个可调用对象且签名匹配 `(loop, context)`，其中 `loop` 是对活动事件循环的引用，而 `context` 是一个包含异常详情的 `dict` (请查看*call_exception_handler()* 文档来获取关于上下文的更多信息)。

`loop.`**`get_exception_handler`**`()`
返回当前的异常处理器，如果没有设置异常处理器，则返回 `None` 。

3.5.2 版新加入.

`loop.`**`default_exception_handler`**`(`*context*`)`
默认的异常处理器。

此方法会在发生异常且未设置异常处理程序时被调用。此方法也可以由想要具有不同于默认处理程序的行为的自定义异常处理程序来调用。

context 参数和*call_exception_handler()* 中的同名参数完全相同。

`loop.`**`call_exception_handler`**`(`*context*`)`
调用当前事件循环异常处理器。

context 是个包含下列键的 `dict` 对象 (未来版本的 Python 可能会引入新键):

- 'message': 错误消息;
- 'exception' （可选）: 异常对象;
- 'future' （可选）: *asyncio.Future* 实例。
- 'task' (optional): *asyncio.Task* instance;
- 'handle' （可选）: *asyncio.Handle* 实例;
- 'protocol' （可选）: *Protocol* 实例;
- 'transport' （可选）: *Transport* 实例;
- 'socket' (optional): *socket.socket* instance;
- **'asyncgen' (optional): Asynchronous generator that caused** the exception.

備 🄴: 此方法不应在子类化的事件循环中被重载。对于自定义的异常处理，请使用*set_exception_handler()* 方法。

开启调试模式

`loop.`**`get_debug`**`()`
获取事件循环调试模式状态（*bool*）。

如果环境变量 `PYTHONASYNCIODEBUG` 是一个非空字符串，就返回 `True` ，否则就返回 `False` 。

`loop.`**`set_debug`**`(`*enabled: bool*`)`
设置事件循环的调试模式。

3.7 版更變: 现在也可以通过新的*Python 开发模式* 来启用调试模式。

也参考:

debug mode of asyncio.

运行子进程

本小节所描述的方法都是低层级的。在常规 async/await 代码中请考虑改用高层级的*asyncio.create_subprocess_shell()* 和*asyncio.create_subprocess_exec()* 便捷函数。

備 🄵: On Windows, the default event loop *ProactorEventLoop* supports subprocesses, whereas *SelectorEventLoop* does not. See *Subprocess Support on Windows* for details.

---

coroutine `loop.`**`subprocess_exec`**`(`*protocol_factory*, **args*, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*`)`
用 *args* 指定的一个或者多个字符串型参数创建一个子进程。

args 必须是个由下列形式的字符串组成的列表：

- *str*；
- 或者由*文件系统编码格式* 编码的*bytes*。

第一个字符串指定可执行程序，其余的字符串指定其参数。所有字符串参数共同组成了程序的 `argv`。

此方法类似于调用标准库*subprocess.Popen* 类，设置 `shell=False` 并将字符串列表作为第一个参数传入；但是，*Popen* 只接受一个单独的字符串列表参数，而 *subprocess_exec* 接受多个字符串参数。

protocol_factory 必须为一个返回*asyncio.SubprocessProtocol* 类的子类的可调用对象。

其他参数：

- *stdin* 可以是以下对象之一：
  - 一个文件类对象，表示要使用*connect_write_pipe()* 连接到子进程的标准输入流的管道
  - *subprocess.PIPE* 常量（默认），将创建并连接一个新的管道。
  - `None` 值，这将使得子进程继承来自此进程的文件描述符
  - *subprocess.DEVNULL* 常量，这表示将使用特殊的*os.devnull* 文件
- *stdout* 可以是以下对象之一：
  - 一个文件类对象，表示要使用*connect_write_pipe()* 连接到子进程的标准输出流的管道
  - *subprocess.PIPE* 常量（默认），将创建并连接一个新的管道。
  - `None` 值，这将使得子进程继承来自此进程的文件描述符
  - *subprocess.DEVNULL* 常量，这表示将使用特殊的*os.devnull* 文件

- *stderr* 可以是以下对象之一:
  - 一个文件类对象，表示要使用*connect_write_pipe()* 连接到子进程的标准错误流的管道
  - *subprocess.PIPE* 常量（默认），将创建并连接一个新的管道。
  - None 值，这将使得子进程继承来自此进程的文件描述符
  - *subprocess.DEVNULL* 常量，这表示将使用特殊的*os.devnull* 文件
  - *subprocess.STDOUT* 常量，将把标准错误流连接到进程的标准输出流
- 所有其他关键字参数会被不加解释地传给*subprocess.Popen*，除了 *bufsize*, *universal_newlines*, *shell*, *text*, *encoding* 和 *errors*，它们都不应当被指定。

  asyncio 子进程 API 不支持将流解码为文本。可以使用*bytes.decode()* 来将从流返回的字节串转换为文本。

其他参数的文档，请参阅*subprocess.Popen* 类的构造函数。

返回一对 (transport, protocol)，其中 *transport* 来自*asyncio.SubprocessTransport* 基类而 *protocol* 是由 *protocol_factory* 所实例化的对象。

coroutine loop.**subprocess_shell**(*protocol_factory*, *cmd*, ***, *stdin=subprocess.PIPE*, *stdout=subprocess.PIPE*, *stderr=subprocess.PIPE*, ***kwargs*)

基于 *cmd* 创建一个子进程，该参数可以是一个*str* 或者按*文件系统编码格式* 编码得到的*bytes*，使用平台的"shell" 语法。

这类似与用 shell=True 调用标准库的*subprocess.Popen* 类。

protocol_factory 必须为一个返回*SubprocessProtocol* 类的子类的可调用对象。

请参阅*subprocess_exec()* 了解有关其余参数的详情。

返回一对 (transport, protocol)，其中 *transport* 来自*SubprocessTransport* 基类而 *protocol* 是由 *protocol_factory* 所实例化的对象。

備🄴: 应用程序要负责确保正确地转义所有空白字符和特殊字符以防止 shell 注入 漏洞。*shlex.quote()* 函数可以被用来正确地转义字符串中可能被用来构造 shell 命令的空白字符和特殊字符。

回调处理

class asyncio.**Handle**

由*loop.call_soon()*, *loop.call_soon_threadsafe()* 所返回的回调包装器对象。

cancel()

取消回调。如果此回调已被取消或已被执行，此方法将没有任何效果。

cancelled()

如果此回调已被取消则返回 True。

3.7 版新加入.

class asyncio.**TimerHandle**

由*loop.call_later()* 和*loop.call_at()* 所返回的回调包装器对象。

这个类是*Handle* 的子类。

when()

返回加入计划任务的回调时间，以*float* 值表示的秒数。

时间值是一个绝对时间戳，使用与*loop.time()* 相同的时间引用。

3.7 版新加入.

Server 对象

Server 对象可使用*loop.create_server()*, *loop.create_unix_server()*, *start_server()* 和*start_unix_server()* 等函数来创建。

请不要直接实例化该类。

class asyncio.**Server**

Server 对象是异步上下文管理器。当用于 `async with` 语句时，异步上下文管理器可以确保 Server 对象被关闭，并且在 `async with` 语句完成后，不接受新的连接。

```
srv = await loop.create_server(...)

async with srv:
    # some code

# At this point, srv is closed and no longer accepts new connections.
```

3.7 版更變: Python3.7 开始，Server 对象是一个异步上下文管理器。

close()

停止服务：关闭监听的套接字并且设置*sockets* 属性为 `None` 。

用于表示已经连进来的客户端连接会保持打开的状态。

服务器是被异步关闭的，使用*wait_closed()* 协程来等待服务器关闭。

get_loop()

返回与服务器对象相关联的事件循环。

3.7 版新加入.

coroutine start_serving()

开始接受连接。

这个方法是幂等的【相同参数重复执行，能获得相同的结果】，所以此方法能在服务已经运行的时候调用。

传给*loop.create_server()* 和*asyncio.start_server()* 的 *start_serving* 仅限关键字形参允许创建不接受初始连接的 Server 对象。在此情况下可以使用 `Server.start_serving()` 或*Server.serve_forever()* 让 Server 对象开始接受连接。

3.7 版新加入.

coroutine serve_forever()

开始接受连接，直到协程被取消。`serve_forever` 任务的取消将导致服务器被关闭。

如果服务器已经在接受连接了，这个方法可以被调用。每个 *Server* 对象，仅能有一个 `serve_forever` 任务。

示例:

```
async def client_connected(reader, writer):
    # Communicate with the client with
    # reader/writer streams.  For example:
    await reader.readline()

async def main(host, port):
    srv = await asyncio.start_server(
```

(下页继续)

(繼續上一頁)

```
        client_connected, host, port)
    await srv.serve_forever()

asyncio.run(main('127.0.0.1', 0))
```

3.7 版新加入.

is_serving()
如果服务器正在接受新连接的状态，返回 `True` 。

3.7 版新加入.

coroutine wait_closed()
等待*close()* 方法执行完毕。

sockets
服务器监听的*socket.socket* 对象列表。

3.7 版更變: 在 Python 3.7 之前 `Server.sockets` 会直接返回内部的服务器套接字列表。在 3.7 版则会返回该列表的副本。

事件循环实现

asyncio 带有两种不同的事件循环实现: *SelectorEventLoop* 和*ProactorEventLoop*。

默认情况下 asyncio 被配置为在 Unix 上使用*SelectorEventLoop* 而在 Windows 上使用*ProactorEventLoop*。

class asyncio.**SelectorEventLoop**
基于*selectors* 模块的事件循环。

使用给定平台中最高效的可用 *selector*。也可以手动配置要使用的特定 selector:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

可用性: Unix, Windows。

class asyncio.**ProactorEventLoop**
用"I/O Completion Ports" (IOCP) 构建的专为 Windows 的事件循环。

可用性: Windows。

也参考:

有关 I/O 完成端口的 MSDN 文档。

class asyncio.**AbstractEventLoop**
asyncio 兼容事件循环的抽象基类。

事件循环方法 一节列出了 `AbstractEventLoop` 的替代实现应当定义的所有方法。

示例

请注意本节中的所有示例都 **有意地**演示了如何使用低层级的事件循环 API，例如*loop.run_forever()* 和*loop.call_soon()*。现代的 asyncio 应用很少需要以这样的方式编写；请考虑使用高层级的函数例如*asyncio.run()*。

call_soon() 的 Hello World 示例。

一个使用*loop.call_soon()* 方法来安排回调的示例。回调会显示 "Hello World" 然后停止事件循环:

```
import asyncio

def hello_world(loop):
    """A callback to print 'Hello World' and stop the event loop"""
    print('Hello World')
    loop.stop()

loop = asyncio.get_event_loop()

# Schedule a call to hello_world()
loop.call_soon(hello_world, loop)

# Blocking call interrupted by loop.stop()
try:
    loop.run_forever()
finally:
    loop.close()
```

也参考:

一个类似的*Hello World* 示例，使用协程和*run()* 函数创建。

使用 call_later() 来展示当前的日期

一个每秒刷新显示当前日期的示例。回调使用*loop.call_later()* 方法在 5 秒后将自身重新加入计划日程，然后停止事件循环:

```
import asyncio
import datetime

def display_date(end_time, loop):
    print(datetime.datetime.now())
    if (loop.time() + 1.0) < end_time:
        loop.call_later(1, display_date, end_time, loop)
    else:
        loop.stop()

loop = asyncio.get_event_loop()

# Schedule the first call to display_date()
end_time = loop.time() + 5.0
loop.call_soon(display_date, end_time, loop)

# Blocking call interrupted by loop.stop()
try:
```

(下页继续)

(繼續上一頁)

```
    loop.run_forever()
finally:
    loop.close()
```

也参考:

一个类似的*current date* 示例，使用协程和*run()* 函数创建。

监控一个文件描述符的读事件

使用*loop.add_reader()* 方法，等到文件描述符收到一些数据，然后关闭事件循环:

```
import asyncio
from socket import socketpair

# Create a pair of connected file descriptors
rsock, wsock = socketpair()

loop = asyncio.get_event_loop()

def reader():
    data = rsock.recv(100)
    print("Received:", data.decode())

    # We are done: unregister the file descriptor
    loop.remove_reader(rsock)

    # Stop the event loop
    loop.stop()

# Register the file descriptor for read event
loop.add_reader(rsock, reader)

# Simulate the reception of data from the network
loop.call_soon(wsock.send, 'abc'.encode())

try:
    # Run the event loop
    loop.run_forever()
finally:
    # We are done. Close sockets and the event loop.
    rsock.close()
    wsock.close()
    loop.close()
```

也参考:

- 一个类似的示例，使用传输、协议和*loop.create_connection()* 方法创建。
- 另一个类似的示例，使用了高层级的*asyncio.open_connection()* 函数和流。

为 SIGINT 和 SIGTERM 设置信号处理器

(这个 signals 示例只适用于 Unix。)

使用*loop.add_signal_handler()* 方法为信号 SIGINT 和 SIGTERM 注册处理程序:

```
import asyncio
import functools
import os
import signal

def ask_exit(signame, loop):
    print("got signal %s: exit" % signame)
    loop.stop()

async def main():
    loop = asyncio.get_running_loop()

    for signame in {'SIGINT', 'SIGTERM'}:
        loop.add_signal_handler(
            getattr(signal, signame),
            functools.partial(ask_exit, signame, loop))

    await asyncio.sleep(3600)

print("Event loop running for 1 hour, press Ctrl+C to interrupt.")
print(f"pid {os.getpid()}: send SIGINT or SIGTERM to exit.")

asyncio.run(main())
```

18.1.8 Futures

源代码: Lib/asyncio/futures.py, Lib/asyncio/base_futures.py

Future 对象用来链接 **底层回调式代码**和高层异步/等待式代码。

Future 函数

asyncio.**isfuture**(*obj*)
如果 *obj* 为下面任意对象，返回 True:

- 一个*asyncio.Future* 类的实例，
- 一个*asyncio.Task* 类的实例，
- 带有 _asyncio_future_blocking 属性的类似 Future 的对象。

3.5 版新加入.

asyncio.**ensure_future**(*obj*, *, *loop=None*)
返回:

- *obj* 参数会是保持原样，如果 *obj* 是*Future*、*Task* 或类似 Future 的对象 (*isfuture()* 用于测试。)
- 封装了 *obj* 的*Task* 对象，如果 *obj* 是一个协程 (使用*iscoroutine()* 进行检测)；在此情况下该协程将通过 ensure_future() 加入执行计划。

- 等待 *obj* 的*Task* 对象，如果 *obj* 是一个可等待对象 (*inspect.isawaitable()* 用于测试)

如果 *obj* 不是上述对象会引发一个*TypeError* 异常。

重要: 查看*create_task()* 函数，它是创建新任务的首选途径。

Save a reference to the result of this function, to avoid a task disappearing mid execution.

3.5.1 版更變: 这个函数接受任意*awaitable* 对象。

asyncio.**wrap_future**(*future*, *, *loop=None*)
将一个*concurrent.futures.Future* 对象封装到*asyncio.Future* 对象中。

Future 对象

class asyncio.**Future**(*, *loop=None*)
一个 Future 代表一个异步运算的最终结果。线程不安全。

Future 是一个*awaitable* 对象。协程可以等待 Future 对象直到它们有结果或异常集合或被取消。

通常 Future 用于支持底层回调式代码 (例如在协议实现中使用 asyncio *transports*) 与高层异步/等待式代码交互。

经验告诉我们永远不要面向用户的接口暴露 Future 对象，同时建议使用*loop.create_future()* 来创建 Future 对象。这种方法可以让 Future 对象使用其它的事件循环实现，它可以注入自己的优化实现。

3.7 版更變: 加入对*contextvars* 模块的支持。

result()
返回 Future 的结果。

如果 Future 状态为 完成，并由*set_result()* 方法设置一个结果，则返回这个结果。

如果 Future 状态为 完成，并由*set_exception()* 方法设置一个异常，那么这个方法会引发异常。

如果 Future 已 取消，方法会引发一个*CancelledError* 异常。

如果 Future 的结果还不可用，此方法会引发一个*InvalidStateError* 异常。

set_result(*result*)
将 Future 标记为 完成并设置结果。

如果 Future 已经 完成则抛出一个*InvalidStateError* 错误。

set_exception(*exception*)
将 Future 标记为 完成并设置一个异常。

如果 Future 已经 完成则抛出一个*InvalidStateError* 错误。

done()
如果 Future 为已 完成则返回 True 。

如果 Future 为 取消或调用*set_result()* 设置了结果或调用*set_exception()* 设置了异常，那么它就是 完成。

cancelled()
如果 Future 已 取消则返回 True

这个方法通常在设置结果或异常前用来检查 Future 是否已 取消。

```
if not fut.cancelled():
    fut.set_result(42)
```

add_done_callback(*callback*, *, *context=None*)
添加一个在 Future 完成时运行的回调函数。

调用 *callback* 时，Future 对象是它的唯一参数。

如果调用这个方法时 Future 已经 完成，回调函数会被*loop.call_soon()* 调度。

可选键值类的参数 *context* 允许 *callback* 运行在一个指定的自定义*contextvars.Context* 对象中。如果没有提供 *context* ，则使用当前上下文。

可以用*functools.partial()* 给回调函数传递参数，例如:

```
# Call 'print("Future:", fut)' when "fut" is done.
fut.add_done_callback(
    functools.partial(print, "Future:"))
```

3.7 版更變: 加入键值类形参 *context*。请参阅 **PEP 567** 查看更多细节。

remove_done_callback(*callback*)
从回调列表中移除 *callback* 。

返回被移除的回调函数的数量，通常为 1，除非一个回调函数被添加多次。

cancel(*msg=None*)
取消 Future 并调度回调函数。

如果 Future 已经 完成或 取消，返回 `False` 。否则将 Future 状态改为 取消并在调度回调函数后返回 `True` 。

3.9 版更變: 增加了 `msg` 形参。

exception()
返回 Future 已设置的异常。

只有 Future 在 完成时才返回异常（或者 `None` ，如果没有设置异常）。

如果 Future 已 取消，方法会引发一个*CancelledError* 异常。

如果 Future 还没 完成，这个方法会引发一个*InvalidStateError* 异常。

get_loop()
返回 Future 对象已绑定的事件循环。

3.7 版新加入.

这个例子创建一个 Future 对象，创建和调度一个异步任务去设置 Future 结果，然后等待其结果:

```
async def set_after(fut, delay, value):
    # Sleep for *delay* seconds.
    await asyncio.sleep(delay)

    # Set *value* as a result of *fut* Future.
    fut.set_result(value)

async def main():
    # Get the current event loop.
    loop = asyncio.get_running_loop()

    # Create a new Future object.
```

(下页继续)

(繼續上一頁)

```
    fut = loop.create_future()

    # Run "set_after()" coroutine in a parallel Task.
    # We are using the low-level "loop.create_task()" API here because
    # we already have a reference to the event loop at hand.
    # Otherwise we could have just used "asyncio.create_task()".
    loop.create_task(
        set_after(fut, 1, '... world'))

    print('hello ...')

    # Wait until *fut* has a result (1 second) and print it.
    print(await fut)

asyncio.run(main())
```

重要： 该 Future 对象是为了模仿*concurrent.futures.Future* 类。主要差异包含：

- 与 asyncio 的 Future 不同，*concurrent.futures.Future* 实例不是可等待对象。
- *asyncio.Future.result()* 和*asyncio.Future.exception()* 不接受 *timeout* 参数。
- Future 没有完成时*asyncio.Future.result()* 和*asyncio.Future.exception()* 抛出一个*InvalidStateError* 异常。
- 使用*asyncio.Future.add_done_callback()* 注册的回调函数不会立即调用，而是被*loop.call_soon()* 调度。
- asyncio Future 不能兼容*concurrent.futures.wait()* 和*concurrent.futures.as_completed()* 函数。
- *asyncio.Future.cancel()* 接受一个可选的 `msg` 参数，但 `concurrent.futures.cancel()` 无此参数。

18.1.9 传输和协议

前言

传输和协议会被像*loop.create_connection()* 这类 **底层**事件循环接口使用。它们使用基于回调的编程风格支持网络或 IPC 协议（如 HTTP）的高性能实现。

基本上，传输和协议应只在库和框架上使用，而不应该在高层的异步应用中使用它们。

本文档包含*Transports* 和*Protocols* 。

概述

在最顶层，传输只关心 **怎样**传送字节内容，而协议决定传送 **哪些**字节内容 (还要在一定程度上考虑何时)。

也可以这样说：从传输的角度来看，传输是套接字 (或类似的 I/O 终端) 的抽象，而协议是应用程序的抽象。

换另一种说法，传输和协议一起定义网络 I/0 和进程间 I/O 的抽象接口。

传输对象和协议对象总是一对一关系：协议调用传输方法来发送数据，而传输在接收到数据时调用协议方法传递数据。

大部分面向连接的事件循环方法 (如*loop.create_connection()*) 通常接受 *protocol_factory* 参数为接收到的链接创建 协议对象，并用 传输对象来表示。这些方法一般会返回 `(transport, protocol)` 元组。

内容

本文档包含下列小节:

- 传输 部分记载异步 IO *BaseTransport* 、*ReadTransport* 、*WriteTransport* 、*Transport* 、*DatagramTransport* 和*SubprocessTransport* 等类。
- The *Protocols* section documents asyncio *BaseProtocol*, *Protocol*, *BufferedProtocol*, *DatagramProtocol*, and *SubprocessProtocol* classes.
- 例子 部分展示怎样使用传输、协议和底层事件循环接口。

传输

源码: Lib/asyncio/transports.py

传输属于*asyncio* 模块中的类，用来抽象各种通信通道。

传输对象总是由异步 *IO* 事件循环 实例化。

异步 IO 实现 TCP、UDP、SSL 和子进程管道的传输。传输上可用的方法由传输的类型决定。

传输类属于线程不安全 。

传输层级

class asyncio.**BaseTransport**
所有传输的基类。包含所有异步 IO 传输共用的方法。

class asyncio.**WriteTransport**(*BaseTransport*)
只写链接的基础传输。

WriteTransport 类的实例由*loop.connect_write_pipe()* 事件循环方法返回，也被子进程相关的方法如*loop.subprocess_exec()* 使用。

class asyncio.**ReadTransport**(*BaseTransport*)
只读链接的基础传输。

ReadTransport 类的实例由*loop.connect_read_pipe()* 事件循环方法返回，也被子进程相关的方法如*loop.subprocess_exec()* 使用。

class asyncio.**Transport**(*WriteTransport*, *ReadTransport*)
接口代表一个双向传输，如 TCP 链接。

用户不用直接实例化传输；调用一个功能函数，给它传递协议工厂和其它需要的信息就可以创建传输和协议。

传输类实例由如*loop.create_connection()* 、*loop.create_unix_connection()* 、*loop.create_server()* 、*loop.sendfile()* 等这类事件循环方法使用或返回。

class asyncio.**DatagramTransport**(*BaseTransport*)
数据报 (UDP) 传输链接。

DatagramTransport 类实例由事件循环方法*loop.create_datagram_endpoint()* 返回。

class asyncio.**SubprocessTransport**(*BaseTransport*)
表示父进程和子进程之间连接的抽象。

SubprocessTransport 类的实例由事件循环方法*loop.subprocess_shell()* 和*loop.subprocess_exec()* 返回。

基础传输

BaseTransport.**close**()
关闭传输。

如果传输具有发送数据缓冲区，将会异步发送已缓存的数据。在所有已缓存的数据都已处理后，就会将*None* 作为协议*protocol.connection_lost()* 方法的参数并进行调用。在这之后，传输不再接收任何数据。

BaseTransport.**is_closing**()
返回 `True` ，如果传输正在关闭或已经关闭。。

BaseTransport.**get_extra_info**(*name*, *default=None*)
返回传输或它使用的相关资源信息。

name 是表示要获取传输特定信息的字符串。

default 是在信息不可用或传输不支持第三方事件循环实现或当前平台查询时返回的值。

例如下面代码尝试获取传输相关套接字对象:

```
sock = transport.get_extra_info('socket')
if sock is not None:
    print(sock.getsockopt(...))
```

传输可查询信息类别:

- 套接字:
  - `'peername'`: 套接字链接时的远端地址，*socket.socket.getpeername()* 方法的结果 (出错时为 `None`)
  - `'socket'`: *socket.socket* 实例
  - `'sockname'`: 套接字本地地址，*socket.socket.getsockname()* 方法的结果
- SSL 套接字
  - `'compression'`: 用字符串指定压缩算法，或者链接没有压缩时为 `None` ；*ssl.SSLSocket.compression()* 的结果。
  - `'cipher'`: 一个三值的元组，包含使用密码的名称，定义使用的 SSL 协议的版本，使用的加密位数。*ssl.SSLSocket.cipher()* 的结果。

  - 'peercert': 远端凭证; *ssl.SSLSocket.getpeercert()* 结果。
  - 'sslcontext': *ssl.SSLContext* 实例
  - 'ssl_object': *ssl.SSLObject* 或*ssl.SSLSocket* 实例
- 管道:
  - 'pipe': 管道对象
- 子进程:
  - 'subprocess': *subprocess.Popen* 实例

BaseTransport.**set_protocol**(*protocol*)
设置一个新协议。

只有两种协议都写明支持切换才能完成切换协议。

BaseTransport.**get_protocol**()
返回当前协议。

只读传输

ReadTransport.**is_reading**()
如果传输接收到新数据时返回 True 。

3.7 版新加入.

ReadTransport.**pause_reading**()
暂停传输的接收端。*protocol.data_received()* 方法将不会收到数据，除非*resume_reading()* 被调用。

3.7 版更變: 这个方法幂等的，它可以在传输已经暂停或关闭时调用。

ReadTransport.**resume_reading**()
恢复接收端。如果有数据可读取时，协议方法*protocol.data_received()* 将再次被调用。

3.7 版更變: 这个方法幂等的，它可以在传输已经准备好读取数据时调用。

只写传输

WriteTransport.**abort**()
立即关闭传输，不会等待已提交的操作处理完毕。已缓存的数据将会丢失。不会接收更多数据。最终*None* 将作为协议的*protocol.connection_lost()* 方法的参数被调用。

WriteTransport.**can_write_eof**()
如果传输支持*write_eof()* 返回*True* 否则返回*False* 。

WriteTransport.**get_write_buffer_size**()
返回传输使用输出缓冲区的当前大小。

WriteTransport.**get_write_buffer_limits**()
获取写入流控制 *high* 和 *low* 高低标记位。返回元组 (low, high) ，*low* 和 *high* 为正字节数。

使用*set_write_buffer_limits()* 设置限制。

3.4.2 版新加入.

WriteTransport.**set_write_buffer_limits**(*high=None*, *low=None*)
设置写入流控制 *high* 和 *low* 高低标记位。

这两个值（以字节数表示）控制何时调用协议的*protocol.pause_writing()* 和*protocol.resume_writing()* 方法。如果指明，则低水位必须小于或等于高水位。*high* 和 *low* 都不能为负值。

pause_writing() 会在缓冲区尺寸大于或等于 *high* 值时被调用。如果写入已经被暂停，*resume_writing()* 会在缓冲区尺寸小于或等于 *low* 值时被调用。

默认值是实现专属的。如果只给出了高水位值，则低水位值默认为一个小于或等于高水位值的实现传属值。将 *high* 设为零会强制将 *low* 也设为零，并使得*pause_writing()* 在缓冲区变为非空的任何时刻被调用。将 *low* 设为零会使得*resume_writing()* 在缓冲区为空时只被调用一次。对于上下限都使用零值通常是不够优化的，因为它减少了并发执行 I/O 和计算的机会。

可使用*get_write_buffer_limits()* 来获取上下限值。

WriteTransport.**write**(*data*)
将一些 *data* 字节串写入传输。

此方法不会阻塞；它会缓冲数据并安排其被异步地发出。

WriteTransport.**writelines**(*list_of_data*)
将数据字节串的列表（或任意可迭代对象）写入传输。这在功能上等价于在可迭代对象产生的每个元素上调用*write()*，但其实现可能更为高效。

WriteTransport.**write_eof**()
在刷新所有已缓冲数据之后关闭传输的写入端。数据仍可以被接收。

如果传输（例如 SSL）不支持半关闭的连接，此方法会引发*NotImplementedError*。

数据报传输

DatagramTransport.**sendto**(*data*, *addr=None*)
将 *data* 字节串发送到 *addr* (基于传输的目标地址) 所给定的远端对等方。如果 *addr* 为*None*，则将数据发送到传输创建时给定的目标地址。

此方法不会阻塞；它会缓冲数据并安排其被异步地发出。

DatagramTransport.**abort**()
立即关闭传输，不会等待已提交的操作执行完毕。已缓存的数据将会丢失。不会接收更多的数据。协议的*protocol.connection_lost()* 方法最终将附带*None* 作为参数被调用。

子进程传输

SubprocessTransport.**get_pid**()
将子进程的进程 ID 以整数形式返回。

SubprocessTransport.**get_pipe_transport**(*fd*)
返回对应于整数文件描述符 *fd* 的通信管道的传输:

- 0: 标准输入 (*stdin*) 的可读流式传输，如果子进程创建时未设置 `stdin=PIPE` 则为*None*
- 1: 标准输出 (*stdout*) 的可写流式传输，如果子进程创建时未设置 `stdout=PIPE` 则为*None*
- 2: 标准错误 (*stderr*) 的可写流式传输，如果子进程创建时未设置 `stderr=PIPE` 则为*None*
- 其他 *fd*: *None*

SubprocessTransport.**get_returncode**()
返回整数形式的进程返回码，或者如果还未返回则为*None*，这类似于*subprocess.Popen.returncode*属性。

SubprocessTransport.**kill**()
杀死子进程。

在 POSIX 系统中，函数会发送 SIGKILL 到子进程。在 Windows 中，此方法是*terminate()* 的别名。

另请参见*subprocess.Popen.kill()*。

SubprocessTransport.**send_signal**(*signal*)
发送 *signal* 编号到子进程，与*subprocess.Popen.send_signal()* 一样。

SubprocessTransport.**terminate**()
停止子进程。

在 POSIX 系统中，此方法会发送 SIGTERM 到子进程。在 Windows 中，则会调用 Windows API 函数 TerminateProcess() 来停止子进程。

另请参见*subprocess.Popen.terminate()*。

SubprocessTransport.**close**()
通过调用*kill()* 方法来杀死子进程。

如果子进程尚未返回，并关闭 *stdin*, *stdout* 和 *stderr* 管道的传输。

协议

源码: Lib/asyncio/protocols.py

---

asyncio 提供了一组抽象基类，它们应当被用于实现网络协议。这些类被设计为与*传输* 配合使用。

抽象基础协议类的子类可以实现其中的部分或全部方法。所有这些方法都是回调：它们由传输或特定事件调用，例如当数据被接收的时候。基础协议方法应当由相应的传输来调用。

基础协议

class asyncio.**BaseProtocol**
带有所有协议的共享方法的基础协议。

class asyncio.**Protocol**(*BaseProtocol*)
用于实现流式协议（TCP, Unix 套接字等等）的基类。

class asyncio.**BufferedProtocol**(*BaseProtocol*)
用于实现可对接收缓冲区进行手动控制的流式协议的基类。

class asyncio.**DatagramProtocol**(*BaseProtocol*)
用于实现数据报（UDP）协议的基类。

class asyncio.**SubprocessProtocol**(*BaseProtocol*)
用于实现与子进程通信（单向管道）的协议的基类。

基础协议

所有 asyncio 协议均可实现基础协议回调。

连接回调

连接回调会在所有协议上被调用，每个成功的连接将恰好调用一次。所有其他协议回调只能在以下两个方法之间被调用。

BaseProtocol.**connection_made**(*transport*)
连接建立时被调用。

transport 参数是代表连接的传输。此协议负责将引用保存至对应的传输。

BaseProtocol.**connection_lost**(*exc*)
连接丢失或关闭时将被调用。

方法的参数是一个异常对象或为*None*。后者意味着收到了常规的 EOF，或者连接被连接的一端取消或关闭。

流程控制回调

流程控制回调可由传输来调用以暂停或恢复协议所执行的写入操作。

请查看*set_write_buffer_limits()* 方法的文档了解详情。

BaseProtocol.**pause_writing**()
当传输的缓冲区升至高水位以上时将被调用。

BaseProtocol.**resume_writing**()
当传输的缓冲区降到低水位以下时将被调用。

如果缓冲区大小等于高水位值，则*pause_writing()* 不会被调用：缓冲区大小必须要高于该值。

相反地，*resume_writing()* 会在缓冲区大小等于或小于低水位值时被调用。这些结束条件对于当两个水位取零值时也能确保符合预期的行为是很重要的。

流式协议

事件方法，例如*loop.create_server()*，*loop.create_unix_server()*，*loop.create_connection()*，*loop.create_unix_connection()*，*loop.connect_accepted_socket()*，*loop.connect_read_pipe()* 和*loop.connect_write_pipe()* 都接受返回流式协议的工厂。

Protocol.**data_received**(*data*)
当收到数据时被调用。*data* 为包含入站数据的非空字节串对象。

数据是否会被缓冲、分块或重组取决于具体传输。通常，你不应依赖于特定的语义而应使你的解析具有通用性和灵活性。但是，数据总是要以正确的顺序被接收。

此方法在连接打开期间可以被调用任意次数。

但是，*protocol.eof_received()* 最多只会被调用一次。一旦 *eof_received()* 被调用，`data_received()` 就不会再被调用。

Protocol.**eof_received**()
当发出信号的另一端不再继续发送数据时（例如通过调用*transport.write_eof()*，如果另一端也使用 asyncio 的话）被调用。

此方法可能返回假值 (包括 None)，在此情况下传输将会自行关闭。相反地，如果此方法返回真值，将以所用的协议来确定是否要关闭传输。由于默认实现是返回 None，因此它会隐式地关闭连接。

某些传输，包括 SSL 在内，并不支持半关闭的连接，在此情况下从该方法返回真值将导致连接被关闭。

状态机：

```
start -> connection_made
    [-> data_received]*
    [-> eof_received]?
-> connection_lost -> end
```

缓冲流协议

3.7 版新加入.

带缓冲的协议可与任何支持流式协议 的事件循环方法配合使用。

BufferedProtocol 实现允许显式手动分配和控制接收缓冲区。随后事件循环可以使用协议提供的缓冲区来避免不必要的数据复制。这对于接收大量数据的协议来说会有明显的性能提升。复杂的协议实现能显著地减少缓冲区分配的数量。

以下回调是在*BufferedProtocol* 实例上被调用的:

BufferedProtocol.**get_buffer**(*sizehint*)
调用后会分配新的接收缓冲区。

sizehint 是推荐的返回缓冲区最小尺寸。返回小于或大于 *sizehint* 推荐尺寸的缓冲区也是可接受的。当设为 -1 时，缓冲区尺寸可以是任意的。返回尺寸为零的缓冲区则是错误的。

get_buffer() 必须返回一个实现了 缓冲区协议的对象。

BufferedProtocol.**buffer_updated**(*nbytes*)
用接收的数据更新缓冲区时被调用。

nbytes 是被写入到缓冲区的字节总数。

BufferedProtocol.**eof_received**()
请查看*protocol.eof_received()* 方法的文档。

在连接期间*get_buffer()* 可以被调用任意次数。但是，*protocol.eof_received()* 最多只能被调用一次，如果被调用，则在此之后*get_buffer()* 和*buffer_updated()* 不能再被调用。

状态机：

```
start -> connection_made
    [-> get_buffer
        [-> buffer_updated]?
    ]*
    [-> eof_received]?
-> connection_lost -> end
```

数据报协议

数据报协议实例应当由传递给*loop.create_datagram_endpoint()* 方法的协议工厂来构造。

DatagramProtocol.**datagram_received**(*data*, *addr*)
当接收到数据报时被调用。*data* 是包含传入数据的字节串对象。*addr* 是发送数据的对等端地址；实际的格式取决于具体传输。

DatagramProtocol.**error_received**(*exc*)
当前一个发送或接收操作引发*OSError* 时被调用。*exc* 是*OSError* 的实例。

此方法会在当传输（例如 UDP）检测到无法将数据报传给接收方等极少数情况下被调用。而在大多数情况下，无法送达的数据报将被静默地丢弃。

備🄵: 在 BSD 系统（macOS, FreeBSD 等等）上，数据报协议不支持流控制，因为没有可靠的方式来检测因写入多过包所导致的发送失败。

套接字总是显示为'ready' 且多余的包会被丢弃。有一定的可能性会引发*OSError* 并设置 errno 为*errno.ENOBUFS*；如果此异常被引发，它将被报告给*DatagramProtocol.error_received()*，在其他情况下则会被忽略。

子进程协议

子进程协议实例应当由传递给*loop.subprocess_exec()* 和*loop.subprocess_shell()* 方法的协议工厂函数来构造。

SubprocessProtocol.**pipe_data_received**(*fd*, *data*)
当子进程向其 stdout 或 stderr 管道写入数据时被调用。

fd 是以整数表示的管道文件描述符。

data 是包含已接收数据的非空字节串对象。

SubprocessProtocol.**pipe_connection_lost**(*fd*, *exc*)
与子进程通信的其中一个管道关闭时被调用。

fd 以整数表示的已关闭文件描述符。

SubprocessProtocol.**process_exited**()
子进程退出时被调用。

示例

TCP 回显服务器

使用*loop.create_server()* 方法创建 TCP 回显服务器，发回已接收的数据，并关闭连接:

```
import asyncio


class EchoServerProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        peername = transport.get_extra_info('peername')
        print('Connection from {}'.format(peername))
        self.transport = transport
```

(下页继续)

(繼續上一頁)

```
    def data_received(self, data):
        message = data.decode()
        print('Data received: {!r}'.format(message))

        print('Send: {!r}'.format(message))
        self.transport.write(data)

        print('Close the client socket')
        self.transport.close()


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    server = await loop.create_server(
        lambda: EchoServerProtocol(),
        '127.0.0.1', 8888)

    async with server:
        await server.serve_forever()


asyncio.run(main())
```

也參考:

使用流的 *TCP* 回显服务器 示例，使用了高层级的*asyncio.start_server()* 函数。

TCP 回显客户端

使用*loop.create_connection()* 方法的 TCP 回显客户端，发送数据并等待，直到连接被关闭:

```
import asyncio


class EchoClientProtocol(asyncio.Protocol):
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        transport.write(self.message.encode())
        print('Data sent: {!r}'.format(self.message))

    def data_received(self, data):
        print('Data received: {!r}'.format(data.decode()))

    def connection_lost(self, exc):
        print('The server closed the connection')
        self.on_con_lost.set_result(True)
```

(下页继续)

(繼續上一頁)

```
async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = 'Hello World!'

    transport, protocol = await loop.create_connection(
        lambda: EchoClientProtocol(message, on_con_lost),
        '127.0.0.1', 8888)

    # Wait until the protocol signals that the connection
    # is lost and close the transport.
    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

也參考:

使用流的 TCP 回显客户端 示例，使用了高层级的*asyncio.open_connection()* 函数。

UDP 回显服务器

使用*loop.create_datagram_endpoint()* 方法的 UDP 回显服务器，发回已接收的数据:

```
import asyncio


class EchoServerProtocol:
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        message = data.decode()
        print('Received %r from %s' % (message, addr))
        print('Send %r to %s' % (message, addr))
        self.transport.sendto(data, addr)


async def main():
    print("Starting UDP server")

    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    # One protocol instance will be created to serve all
    # client requests.
    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoServerProtocol(),
```

(下页继续)

(繼續上一頁)

```
        local_addr=('127.0.0.1', 9999))

    try:
        await asyncio.sleep(3600)  # Serve for 1 hour.
    finally:
        transport.close()


asyncio.run(main())
```

UDP 回显客户端

使用*loop.create_datagram_endpoint()* 方法的 UDP 回显客户端，发送数据并在收到回应时关闭传输:

```
import asyncio


class EchoClientProtocol:
    def __init__(self, message, on_con_lost):
        self.message = message
        self.on_con_lost = on_con_lost
        self.transport = None

    def connection_made(self, transport):
        self.transport = transport
        print('Send:', self.message)
        self.transport.sendto(self.message.encode())

    def datagram_received(self, data, addr):
        print("Received:", data.decode())

        print("Close the socket")
        self.transport.close()

    def error_received(self, exc):
        print('Error received:', exc)

    def connection_lost(self, exc):
        print("Connection closed")
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    on_con_lost = loop.create_future()
    message = "Hello World!"

    transport, protocol = await loop.create_datagram_endpoint(
        lambda: EchoClientProtocol(message, on_con_lost),
        remote_addr=('127.0.0.1', 9999))
```

(下页继续)

(繼續上一頁)

```
    try:
        await on_con_lost
    finally:
        transport.close()


asyncio.run(main())
```

链接已存在的套接字

附带一个协议使用*loop.create_connection()* 方法，等待直到套接字接收数据:

```
import asyncio
import socket


class MyProtocol(asyncio.Protocol):

    def __init__(self, on_con_lost):
        self.transport = None
        self.on_con_lost = on_con_lost

    def connection_made(self, transport):
        self.transport = transport

    def data_received(self, data):
        print("Received:", data.decode())

        # We are done: close the transport;
        # connection_lost() will be called automatically.
        self.transport.close()

    def connection_lost(self, exc):
        # The socket has been closed
        self.on_con_lost.set_result(True)


async def main():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()
    on_con_lost = loop.create_future()

    # Create a pair of connected sockets
    rsock, wsock = socket.socketpair()

    # Register the socket to wait for data.
    transport, protocol = await loop.create_connection(
        lambda: MyProtocol(on_con_lost), sock=rsock)

    # Simulate the reception of data from the network.
    loop.call_soon(wsock.send, 'abc'.encode())

    try:
        await protocol.on_con_lost
```

(下页继续)

(繼續上一頁)

```
    finally:
        transport.close()
        wsock.close()

asyncio.run(main())
```

也參考:

使用低层级的*loop.add_reader()* 方法来注册一个 FD 的监视文件描述符以读取事件 示例。

使用在协程中通过*open_connection()* 函数创建的高层级流的注册一个打开的套接字以等待使用流的数据 示例。

loop.subprocess_exec() 与 SubprocessProtocol

一个使用子进程协议来获取子进程的输出并等待子进程退出的示例。

这个子进程是由*loop.subprocess_exec()* 方法创建的:

```
import asyncio
import sys

class DateProtocol(asyncio.SubprocessProtocol):
    def __init__(self, exit_future):
        self.exit_future = exit_future
        self.output = bytearray()

    def pipe_data_received(self, fd, data):
        self.output.extend(data)

    def process_exited(self):
        self.exit_future.set_result(True)

async def get_date():
    # Get a reference to the event loop as we plan to use
    # low-level APIs.
    loop = asyncio.get_running_loop()

    code = 'import datetime; print(datetime.datetime.now())'
    exit_future = asyncio.Future(loop=loop)

    # Create the subprocess controlled by DateProtocol;
    # redirect the standard output into a pipe.
    transport, protocol = await loop.subprocess_exec(
        lambda: DateProtocol(exit_future),
        sys.executable, '-c', code,
        stdin=None, stderr=None)

    # Wait for the subprocess exit using the process_exited()
    # method of the protocol.
    await exit_future

    # Close the stdout pipe.
    transport.close()

    # Read the output which was collected by the
```

(下页继续)

(繼續上一頁)

```
    # pipe_data_received() method of the protocol.
    data = bytes(protocol.output)
    return data.decode('ascii').rstrip()

date = asyncio.run(get_date())
print(f"Current date: {date}")
```

另请参阅使用高层级 API 编写的相同示例。

18.1.10 策略

事件循环策略是各个进程的全局对象，它控制事件循环的管理。每个事件循环都有一个默认策略，可以使用策略 API 更改和定制该策略。

策略定义了 “上下文” 的概念，每个上下文管理一个单独的事件循环。默认策略将 *context* 定义为当前线程。

通过使用自定义事件循环策略，可以自定义*get_event_loop()* 、*set_event_loop()* 和*new_event_loop()* 函数的行为。

策略对象应该实现*AbstractEventLoopPolicy* 抽象基类中定义的 API。

获取和设置策略

可以使用下面函数获取和设置当前进程的策略:

asyncio.**get_event_loop_policy**()
: 返回当前进程域的策略。

asyncio.**set_event_loop_policy**(*policy*)
: 将 *policy* 设置为当前进程域策略。

  如果 *policy* 设为 None 将恢复默认策略。

策略对象

抽象事件循环策略基类定义如下:

class asyncio.**AbstractEventLoopPolicy**
: 异步策略的抽象基类。

  get_event_loop()
  : 为当前上下文获取事件循环。

    返回一个实现*AbstractEventLoop* 接口的事件循环对象。

    该方法永远不应返回 None。

    3.6 版更變.

  set_event_loop(*loop*)
  : 将当前上下文的事件循环设置为 *loop* 。

  new_event_loop()
  : 创建并返回一个新的事件循环对象。

    该方法永远不应返回 None。

get_child_watcher()
获取子进程监视器对象。

返回一个实现*AbstractChildWatcher* 接口的监视器对象。

该函数仅支持 Unix。

set_child_watcher(*watcher*)
将当前子进程监视器设置为 *watcher* 。

该函数仅支持 Unix。

asyncio 附带下列内置策略:

class asyncio.**DefaultEventLoopPolicy**
默认的 asyncio 策略。在 Unix 上使用*SelectorEventLoop* 而在 Windows 上使用*ProactorEventLoop*。

不需要手动安装默认策略。asyncio 已配置成自动使用默认策略。

3.8 版更變: 在 Windows 上，现在默认会使用*ProactorEventLoop*。

class asyncio.**WindowsSelectorEventLoopPolicy**
一个使用*SelectorEventLoop* 事件循环实现的替代事件循环策略。

可用性: Windows。

class asyncio.**WindowsProactorEventLoopPolicy**
使用*ProactorEventLoop* 事件循环实现的另一种事件循环策略。

可用性: Windows。

进程监视器

进程监视器允许定制事件循环如何监视 Unix 子进程。具体来说，事件循环需要知道子进程何时退出。

在 asyncio 中子进程由*create_subprocess_exec()* 和*loop.subprocess_exec()* 函数创建。

asyncio 定义了*AbstractChildWatcher* 抽象基类，子监视器必须要实现它，并具有四种不同实现: *ThreadedChildWatcher* (已配置为默认使用), *MultiLoopChildWatcher*, *SafeChildWatcher* 和*FastChildWatcher*。

请参阅子进程和线程 部分。

以下两个函数可用于自定义子进程监视器实现，它将被 asyncio 事件循环使用:

asyncio.**get_child_watcher**()
返回当前策略的当前子监视器。

asyncio.**set_child_watcher**(*watcher*)
将当前策略的子监视器设置为 *watcher* 。*watcher* 必须实现*AbstractChildWatcher* 基类定义的方法。

備 🄵: 第三方事件循环实现可能不支持自定义子监视器。对于这样的事件循环，禁止使用*set_child_watcher()* 或不起作用。

---

class asyncio.**AbstractChildWatcher**

add_child_handler(*pid*, *callback*, **args*)
注册一个新的子处理回调函数。

安排 `callback(pid, returncode, *args)` 在进程的 PID 与 *pid* 相等时调用。指定另一个同进程的回调函数替换之前的回调处理函数。

回调函数 *callback* 必须是线程安全。

remove_child_handler(*pid*)
删除进程 PID 与 *pid* 相等的进程的处理函数。

处理函数成功删除时返回 `True`，没有删除时返回 `False`。

attach_loop(*loop*)
给一个事件循环绑定监视器。

如果监视器之前已绑定另一个事件循环，那么在绑定新循环前会先解绑原来的事件循环。

注意：循环有可能是 `None`。

is_active()
如果监视器已准备好使用则返回 `True`。

使用 *不活动的*当前子监视器生成子进程将引发*RuntimeError*。

3.8 版新加入.

close()
关闭监视器。

必须调用这个方法以确保相关资源会被清理。

class asyncio.**ThreadedChildWatcher**
此实现会为每个生成的子进程启动一具新的等待线程。

即使是当 asyncio 事件循环运行在非主 OS 线程上时它也能可靠地工作。

当处理大量子进程时不存在显著的开销 (每次子进程结束时为 *O(1)*)，但当每个进程启动一个线程时则需要额外的内存。

此监视器会默认被使用。

3.8 版新加入.

class asyncio.**MultiLoopChildWatcher**
此实现会在实例化时注册一个 `SIGCHLD` 信号处理程序。这可能会破坏为 `SIGCHLD` 信号安装自定义处理程序的第三方代码。

此监视器会在收到 `SIGCHLD` 信号时通过显式地轮询每个进程来避免干扰其他代码生成的进程。

该监视器一旦被安装就不会限制从不同线程运行子进程。

该解决方案是安全的，但在处理大量进程时会有显著的开销 (每收到一个 `SIGCHLD` 时为 *O(n)*)。

3.8 版新加入.

class asyncio.**SafeChildWatcher**
该实现会使用主线程中的活动事件循环来处理 `SIGCHLD` 信号。如果主线程没有正在运行的事件循环，则其他线程无法生成子进程 (会引发*RuntimeError*)。

此监视器会在收到 `SIGCHLD` 信号时通过显式地轮询每个进程来避免干扰其他代码生成的进程。

该解决方案与*MultiLoopChildWatcher* 同样安全并同样具有 *O(N)* 复杂度，但需要主线程有正在运行的事件循环才能工作。

class asyncio.**FastChildWatcher**
这种实现直接调用 `os.waitpid(-1)` 来获取所有已结束的进程，可能会中断其它代码🄵生进程并等待它们结束。

在处理大量子监视器时没有明显的开销 (*O(1)* 每次子监视器结束)。

该解决方案需要主线程有正在运行的事件循环才能工作，这与*SafeChildWatcher*一样。

class asyncio.**PidfdChildWatcher**

这个实现会轮询处理文件描述符 (pidfds) 以等待子进程终结。在某些方面，*PidfdChildWatcher* 是一个“理想的”子进程监视器实现。它不需要使用信号或线程，不会介入任何在事件循环以外发起的进程，并能随事件循环发起的子进程数量进行线性伸缩。其主要缺点在于 pidfds 是 Linux 专属的，并且仅在较近版本的核心（5.3+）上可用。

3.9 版新加入.

自定义策略

要实现一个新的事件循环策略，建议子类化*DefaultEventLoopPolicy*并重写需要定制行为的方法，例如:

```
class MyEventLoopPolicy(asyncio.DefaultEventLoopPolicy):

    def get_event_loop(self):
        """Get the event loop.

        This may be None or an instance of EventLoop.
        """
        loop = super().get_event_loop()
        # Do something with loop ...
        return loop

asyncio.set_event_loop_policy(MyEventLoopPolicy())
```

18.1.11 平台支持

*asyncio*模块被设计为可移植的，但由于平台的底层架构和功能，一些平台存在细微的差异和限制。

所有平台

- *loop.add_reader()*和*loop.add_writer()*不能用来监视文件 I/O。

Windows

源代码: Lib/asyncio/proactor_events.py, Lib/asyncio/windows_events.py, Lib/asyncio/windows_utils.py

---

3.8 版更變: 在 Windows 上，*ProactorEventLoop*现在是默认的事件循环。

Windows 上的所有事件循环都不支持以下方法:

- 不支持*loop.create_unix_connection()*和*loop.create_unix_server()*。*socket.AF_UNIX*套接字相关参数仅限于 Unix。
- 不支持*loop.add_signal_handler()*和*loop.remove_signal_handler()*。

*SelectorEventLoop*有下列限制:

- *SelectSelector*只被用于等待套接字事件：它支持套接字且最多支持 512 个套接字。

- *loop.add_reader()* 和*loop.add_writer()* 只接受套接字处理回调函数 (如管道、文件描述符等都不支持)。
- 因为不支持管道，所以*loop.connect_read_pipe()* 和*loop.connect_write_pipe()* 方法没有实现。
- 不支持*Subprocesses*，也就是*loop.subprocess_exec()* 和*loop.subprocess_shell()* 方法没有实现。

ProactorEventLoop 有下列限制:

- 不支持*loop.add_reader()* 和*loop.add_writer()* 方法。

Windows 上单调时钟的分辨率大约为 15.6 毫秒。最佳的分辨率是 0.5 毫秒。分辨率依赖于具体的硬件 (HPET) 和 Windows 的设置。

Windows 的子进程支持

在 Windows 上，默认的事件循环*ProactorEventLoop* 支持子进程，而*SelectorEventLoop* 则不支持。

也不支持*policy.set_child_watcher()* 函数，*ProactorEventLoop* 有不同的机制来监视子进程。

macOS

完整支持流行的 macOS 版本。

macOS <= 10.8

在 macOS 10.6, 10.7 和 10.8 上，默认的事件循环使用*selectors.KqueueSelector*，在这些版本上它并不支持字符设备。可以手工配置*SelectorEventLoop* 来使用*SelectSelector* 或*PollSelector* 以在这些较老版本的 macOS 上支持字符设备。例如:

```
import asyncio
import selectors

selector = selectors.SelectSelector()
loop = asyncio.SelectorEventLoop(selector)
asyncio.set_event_loop(loop)
```

18.1.12 高层级 API 索引

这个页面列举了所有能用于 async/wait 的高层级 asyncio API 集。

Tasks

运行异步程序，创建 Task 对象，等待多件事运行超时的公共集。

run()	创建事件循环，运行一个协程，关闭事件循环。
create_task()	启动一个 asyncio 的 Task 对象。
await *sleep()*	休眠几秒。
await *gather()*	并发执行所有事件的调度和等待。
await *wait_for()*	有超时控制的运行。
await *shield()*	屏蔽取消操作
await *wait()*	完成情况的监控器
current_task()	返回当前 Task 对象
all_tasks()	返回事件循环中所有的 task 对象。
Task	Task 对象
to_thread()	在不同的 OS 线程中异步地运行一个函数。
run_coroutine_threadsafe()	从其他 OS 线程中调度一个协程。
for in *as_completed()*	用 for 循环监控完成情况。

示例

- 使用 *asyncio.gather()* 并行运行.
- 使用 *asyncio.wait_for()* 强制超时.
- 撤销协程.
- *asyncio.sleep()* 的用法.
- 请主要参阅协程与任务文档.

队列集

队列集被用于多个异步 Task 对象的运行调度，实现连接池以及发布/订阅模式。

Queue	先进先出队列
PriorityQueue	优先级队列。
LifoQueue	后进先出队列。

示例

- 使用 *asyncio.Queue* 在多个并发任务间分配工作量.
- 请参阅队列集文档.

子进程集

用于生成子进程和运行 shell 命令的工具包。

await *create_subprocess_exec()*	创建一个子进程。
await *create_subprocess_shell()*	运行一个 shell 命令。

示例

- 执行一个 *shell* 命令.
- 请参阅子进程 *APIs* 相关文档.

流

用于网络 IO 处理的高级 API 集。

await *open_connection()*	建立一个 TCP 连接。
await *open_unix_connection()*	建立一个 Unix socket 连接。
await *start_server()*	启动 TCP 服务。
await *start_unix_server()*	启动一个 Unix socket 服务。
StreamReader	接收网络数据的高级 async/await 对象。
StreamWriter	发送网络数据的高级 async/await 对象。

示例

- *TCP* 客户端样例.
- 请参阅*streams APIs* 文档。

同步

能被用于 Task 对象集的，类似线程的同步基元组件。

Lock	互斥锁。
Event	事件对象。
Condition	条件对象
Semaphore	信号量
BoundedSemaphore	有界的信号量。

示例

- *asyncio.Event* 的用法.
- 请参阅 asyncio 文档*synchronization primitives*.

异常

asyncio.TimeoutError	类似*wait_for()* 等函数在超时时候被引发。请注意 asyncio.TimeoutError 与内建异常*TimeoutError* 无关。
asyncio.CancelledError	当一个 Task 对象被取消的时候被引发。请参阅*Task.cancel()*。

示例

- *在取消请求发生的运行代码中如何处理 CancelledError 异常*.
- 请参阅完整的*asyncio 专用异常* 列表.

18.1.13 低层级 API 索引

本页列出所有低层级的 asyncio API。

获取事件循环

asyncio.get_running_loop()	获取当前运行的事件循环 **首选**函数。
asyncio.get_event_loop()	获得一个事件循环实例 (当前或通过策略)。
asyncio.set_event_loop()	通过当前策略将事件循环设置当前事件循环。
asyncio.new_event_loop()	创建一个新的事件循环。

示例

- *使用 asyncio.get_running_loop()*。

事件循环方法集

查阅*事件循环方法* 相关的主要文档段落。

生命周期

loop.run_until_complete()	运行一个期程/任务/可等待对象直到完成。
loop.run_forever()	一直运行事件循环。
loop.stop()	停止事件循环。
loop.close()	关闭事件循环。
loop.is_running()	返回 `True`，如果事件循环正在运行。
loop.is_closed()	返回 `True`，如果事件循环已经被关闭。
`await` *loop.shutdown_asyncgens()*	关闭异步生成器。

调试

loop.set_debug()	开启或禁用调试模式。
loop.get_debug()	获取当前测试模式。

调度回调函数

loop.call_soon()	尽快调用回调。
loop.call_soon_threadsafe()	*loop.call_soon()* 方法线程安全的变体。
loop.call_later()	在给定时间 之后调用回调函数。
loop.call_at()	在 指定时间调用回调函数。

线程/进程池

await *loop.run_in_executor()*	在*concurrent.futures* 执行器中运行一个独占 CPU 或其它阻塞函数。
loop.set_default_executor()	设置*loop.run_in_executor()* 默认执行器。

任务与期程

loop.create_future()	创建一个*Future* 对象。
loop.create_task()	将协程当作*Task* 一样调度。
loop.set_task_factory()	设置*loop.create_task()* 使用的工厂，它将用来创建*Tasks* 。
loop.get_task_factory()	获取*loop.create_task()* 使用的工厂，它用来创建*Tasks* 。

DNS

await *loop.getaddrinfo()*	异步版的*socket.getaddrinfo()* 。
await *loop.getnameinfo()*	异步版的*socket.getnameinfo()* 。

网络和 IPC

await *loop.create_connection()*	打开一个 TCP 链接。
await *loop.create_server()*	创建一个 TCP 服务。
await *loop.create_unix_connection()*	打开一个 Unix socket 连接。
await *loop.create_unix_server()*	创建一个 Unix socket 服务。
await *loop.connect_accepted_socket()*	将*socket* 包装成 `(transport, protocol)` 对。
await *loop.create_datagram_endpoint()*	打开一个数据报 (UDP) 连接。
await *loop.sendfile()*	通过传输通道发送一个文件。
await *loop.start_tls()*	将一个已建立的链接升级到 TLS。
await *loop.connect_read_pipe()*	将管道读取端包装成 `(transport, protocol)` 对。
await *loop.connect_write_pipe()*	将管道写入端包装成 `(transport, protocol)` 对。

套接字

await *loop.sock_recv()*	从*socket* 接收数据。
await *loop.sock_recv_into()*	从*socket* 接收数据到一个缓冲区中。
await *loop.sock_sendall()*	发送数据到*socket* 。
await *loop.sock_connect()*	链接 await *loop.sock_connect()* 。
await *loop.sock_accept()*	接受一个*socket* 链接。
await *loop.sock_sendfile()*	利用*socket* 发送一个文件。
loop.add_reader()	开始对一个文件描述符的可读性的监视。
loop.remove_reader()	停止对一个文件描述符的可读性的监视。
loop.add_writer()	开始对一个文件描述符的可写性的监视。
loop.remove_writer()	停止对一个文件描述符的可写性的监视。

Unix 信号

loop.add_signal_handler()	给*signal* 添加一个处理回调函数。
loop.remove_signal_handler()	删除*signal* 的处理回调函数。

子进程

loop.subprocess_exec()	衍生一个子进程
loop.subprocess_shell()	从终端命令衍生一个子进程。

错误处理

loop.call_exception_handler()	调用异常处理器。
loop.set_exception_handler()	设置一个新的异常处理器。
loop.get_exception_handler()	获取当前异常处理器。
loop.default_exception_handler()	默认异常处理器实现。

示例

- 使用 *asyncio.get_event_loop()* 和 *loop.run_forever()*.
- 使用 *loop.call_later()*.
- 使用 loop.create_connection() 实现*echo* 客户端.
- 使用 loop.create_connection() 去链接 *socket*.
- 使用 *add_reader()* 监听 *FD(*文件描述符*)* 的读取事件.
- 使用 *loop.add_signal_handler()*.
- 使用 *loop.add_signal_handler()*。

传输

所有传输都实现以下方法:

transport.close()	关闭传输。
transport.is_closing()	返回 `True`，如果传输正在关闭或已经关闭。
transport.get_extra_info()	请求传输的相关信息。
transport.set_protocol()	设置一个新协议。
transport.get_protocol()	返回当前协议。

传输可以接收数据 (TCP 和 Unix 链接，管道等)。它通过*loop.create_connection()*，*loop.create_unix_connection()*，*loop.connect_read_pipe()* 等方法返回。

读取传输

transport.is_reading()	返回 `True`，如果传输正在接收。
transport.pause_reading()	暂停接收。
transport.resume_reading()	继续接收。

传输可以发送数据 (TCP 和 Unix 链接，管道等)。它通过*loop.create_connection()*，*loop.create_unix_connection()*，*loop.connect_write_pipe()* 等方法返回。

写入传输

transport.write()	向传输写入数据。
transport.write()	向传输写入缓冲。
transport.can_write_eof()	返回*True*，如果传输支持发送 EOF。
transport.write_eof()	在冲洗已缓冲的数据后关闭传输和发送 EOF。
transport.abort()	立即关闭传输。
transport.get_write_buffer_size()	返回写入流控制的高位标记位和低位标记位。
transport.set_write_buffer_limits()	设置新的写入流控制的高位标记位和低位标记位。

由*loop.create_datagram_endpoint()* 返回的传输:

数据报传输

transport.sendto()	发送数据到远程链接端。
transport.abort()	立即关闭传输。

基于子进程的底层抽象传输，它由*loop.subprocess_exec()* 和*loop.subprocess_shell()* 返回:

子进程传输

transport.get_pid()	返回子进程的进程 ID。
transport.get_pipe_transport()	返回请求通信管道 (*stdin*, *stdout*, 或 *stderr*) 的传输。
transport.get_returncode()	返回子进程的返回代号。
transport.kill()	杀死子进程。
transport.send_signal()	发送一个信号到子进程。
transport.terminate()	停止子进程。
transport.close()	杀死子进程并关闭所有管道。

协议

协议类可以由下面 **回调方法**实现：

callback *connection_made()*	链接建立时被调用。
callback *connection_lost()*	链接丢失或关闭时被调用。
callback *pause_writing()*	传输的缓冲区超过高位标记位时被调用。
callback *resume_writing()*	传输的缓冲区传送到低位标记位时被调用。

流协议 (TCP, Unix 套接字, 管道)

callback *data_received()*	接收到数据时被调用。
callback *eof_received()*	接收到 EOF 时被调用。

缓冲流协议

callback *get_buffer()*	调用后会分配新的接收缓冲区。
callback *buffer_updated()*	用接收的数据更新缓冲区时被调用。
callback *eof_received()*	接收到 EOF 时被调用。

数据报协议

callback *datagram_received()*	接收到数据报时被调用。
callback *error_received()*	前一个发送或接收操作引发*OSError* 时被调用。

子进程协议

callback *pipe_data_received()*	子进程向 *stdout* 或 *stderr* 管道写入数据时被调用。
callback *pipe_connection_lost()*	与子进程通信的其中一个管道关闭时被调用。
callback *process_exited()*	子进程退出时被调用。

事件循环策略

策略是改变*asyncio.get_event_loop()* 这类函数行为的一个底层机制。更多细节可以查阅策略部分。

访问策略

asyncio.get_event_loop_policy()	返回当前进程域的策略。
asyncio.set_event_loop_policy()	设置一个新的进程域策略。
AbstractEventLoopPolicy	策略对象的基类。

18.1.14 用 asyncio 开发

异步编程与传统的“顺序”编程不同。

本页列出常见的错误和陷阱，并解释如何避免它们。

Debug 模式

默认情况下，asyncio 以生产模式运行。为了简化开发，asyncio 还有一种 *debug 模式 * 。

有几种方法可以启用异步调试模式:

- 将 `PYTHONASYNCIODEBUG` 环境变量设置为 `1` 。
- 使用*Python 开发模式*。
- 将 `debug=True` 传递给*asyncio.run()* 。
- 调用*loop.set_debug()* 。

除了启用调试模式外，还要考虑:

- 将*asyncio logger* 的日志级别设置为 `logging.DEBUG`，例如，下面的代码片段可以在应用程序启动时运行:

```
logging.basicConfig(level=logging.DEBUG)
```

- 配置*warnings* 模块以显示*ResourceWarning* 警告。一种方法是使用 `-W default` 命令行选项。

启用调试模式时:

- asyncio 检查未被等待的协程 并记录他们；这将消除“被遗忘的等待”问题。
- 许多非线程安全的异步 APIs (例如*loop.call_soon()* 和*loop.call_at()* 方法)，如果从错误的线程调用，则会引发异常。
- 如果执行 I/O 操作花费的时间太长，则记录 I/O 选择器的执行时间。
- 执行时间超过 100 毫秒的回调将会载入日志。属性 `loop.slow_callback_duration` 可用于设置以秒为单位的最小执行持续时间，这被视为“缓慢”。

并发性和多线程

事件循环在线程中运行 (通常是主线程)，并在其线程中执行所有回调和任务。当一个任务在事件循环中运行时，没有其他任务可以在同一个线程中运行。当一个任务执行一个 `await` 表达式时，正在运行的任务被挂起，事件循环执行下一个任务。

要调度来自另一 OS 线程的*callback*，应该使用*loop.call_soon_threadsafe()* 方法。例如:

```
loop.call_soon_threadsafe(callback, *args)
```

几乎所有异步对象都不是线程安全的，这通常不是问题，除非在任务或回调函数之外有代码可以使用它们。如果需要这样的代码来调用低级异步 API，应该使用*loop.call_soon_threadsafe()* 方法，例如:

```
loop.call_soon_threadsafe(fut.cancel)
```

要从不同的 OS 线程调度一个协程对象，应该使用*run_coroutine_threadsafe()* 函数。它返回一个*concurrent.futures.Future* 。查询结果:

```
async def coro_func():
    return await asyncio.sleep(1, 42)

# Later in another OS thread:

future = asyncio.run_coroutine_threadsafe(coro_func(), loop)
# Wait for the result:
result = future.result()
```

为了能够处理信号和执行子进程，事件循环必须运行于主线程中。

方法*loop.run_in_executor()* 可以和*concurrent.futures.ThreadPoolExecutor* 一起使用，用于在一个不同的操作系统线程中执行阻塞代码，并避免阻塞运行事件循环的那个操作系统线程。

目前没有什么办法能直接从另一个进程 (例如通过*multiprocessing* 启动的进程) 安排协程或回调。事件循环方法 小节列出了可以从管道读取并监视文件描述符而不会阻塞事件循环的 API。此外，asyncio 的子进程 API 提供了一种启动进程并从事件循环与其通信的办法。最后，之前提到的*loop.run_in_executor()* 方法也可配合*concurrent.futures.ProcessPoolExecutor* 使用以在另一个进程中执行代码。

运行阻塞的代码

不应该直接调用阻塞 (CPU 绑定) 代码。例如，如果一个函数执行 1 秒的 CPU 密集型计算，那么所有并发异步任务和 IO 操作都将延迟 1 秒。

可以用执行器在不同的线程甚至不同的进程中运行任务，以避免使用事件循环阻塞 OS 线程。请参阅*loop.run_in_executor()* 方法了解详情。

日志

asyncio 使用*logging* 模块，所有日志记录都是通过 `"asyncio"` logger 执行的。

默认日志级别是 `logging.INFO` 。可以很容易地调整:

```
logging.getLogger("asyncio").setLevel(logging.WARNING)
```

检测 never-awaited 协同程序

当协程函数被调用而不是被等待时 (即执行 `coro()` 而不是 `await coro()`) 或者协程没有通过*asyncio.create_task()* 被排入计划日程，asyncio 将会发出一条*RuntimeWarning*:

```
import asyncio

async def test():
    print("never scheduled")

async def main():
    test()

asyncio.run(main())
```

输出:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
  test()
```

调试模式的输出:

```
test.py:7: RuntimeWarning: coroutine 'test' was never awaited
Coroutine created at (most recent call last)
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

  < .. >

  File "../t.py", line 7, in main
    test()
  test()
```

通常的修复方法是等待协程或者调用*asyncio.create_task()* 函数:

```
async def main():
    await test()
```

检测就再也没异常

如果调用*Future.set_exception()*，但不等待 Future 对象，将异常传播到用户代码。在这种情况下，当 Future 对象被垃圾收集时，asyncio 将发出一条日志消息。

未处理异常的例子:

```
import asyncio

async def bug():
    raise Exception("not consumed")

async def main():
    asyncio.create_task(bug())

asyncio.run(main())
```

输出:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
  exception=Exception('not consumed')>

Traceback (most recent call last):
  File "test.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

激活调试模式 以获取任务创建处的跟踪信息:

```
asyncio.run(main(), debug=True)
```

调试模式的输出:

```
Task exception was never retrieved
future: <Task finished coro=<bug() done, defined at test.py:3>
    exception=Exception('not consumed') created at asyncio/tasks.py:321>

source_traceback: Object created at (most recent call last):
  File "../t.py", line 9, in <module>
    asyncio.run(main(), debug=True)

< .. >

Traceback (most recent call last):
  File "../t.py", line 4, in bug
    raise Exception("not consumed")
Exception: not consumed
```

備🄵: asyncio 的源代码可以在 Lib/asyncio/ 中找到。

18.2 socket --- 底层网络接口

源代码: Lib/socket.py

这个模块提供了访问 BSD 套接字的接口。在所有现代 Unix 系统、Windows、macOS 和其他一些平台上可用。

備🄵: 一些行为可能因平台不同而异，因为调用的是操作系统的套接字 API。

这个 Python 接口是用 Python 的面向对象风格对 Unix 系统调用和套接字库接口的直译：函数 `socket()` 返回一个 套接字对象，其方法是对各种套接字系统调用的实现。形参类型一般与 C 接口相比更高级：例如在 Python 文件 `read()` 和 `write()` 操作中，接收操作的缓冲区分配是自动的，发送操作的缓冲区长度是隐式的。

也参考:

模块*socketserver* 用于简化网络服务端编写的类。

模块*ssl* 套接字对象的 TLS/SSL 封装。

18.2.1 套接字协议族

根据系统以及构建选项，此模块提供了各种套接字协议簇。

特定的套接字对象需要的地址格式将根据此套接字对象被创建时指定的地址族被自动选择。套接字地址表示如下：

- 一个绑定在文件系统节点上的*AF_UNIX* 套接字的地址表示为一个字符串，使用文件系统字符编码和 `'surrogateescape'` 错误回调方法（see **PEP 383**）。一个地址在 Linux 的抽象命名空间被返回为带有初始的 null 字节的*字节类对象*；注意在这个命名空间种的套接字可能与普通文件系统套接字通信，所以打算运行在 Linux 上的程序可能需要解决两种地址类型。当传递为参数时，一个字符串或字节类对象可以用于任一类型的地址。

  3.3 版更變: 之前，*AF_UNIX* 套接字路径被假设使用 UTF-8 编码。

  3.5 版更變: 现在支持可写的*字节类对象*。

- 一对 `(host, port)` 被用于*AF_INET* 地址族，*host* 是一个表示为互联网域名表示法之内的主机名或者一个 IPv4 地址的字符串，例如 `'daring.cwi.nl'` 或 `'100.50.200.5'`，*port* 是一个整数。

  - 对于 IPv4 地址，有两种可接受的特殊形式被用来代替一个主机地址：`''` 代表 `INADDR_ANY`，用来绑定到所有接口；字符串 `'<broadcast>'` 代表 `INADDR_BROADCAST`。此行为不兼容 IPv6，因此，如果你的 Python 程序打算支持 IPv6，则可能需要避开这些。

- 对于*AF_INET6* 地址族，使用一个四元组 `(host, port, flowinfo, scope_id)`，其中 *flowinfo* 和 *scope_id* 代表了 C 库 `struct sockaddr_in6` 中的 `sin6_flowinfo` 和 `sin6_scope_id` 成员。对于*socket* 模块中的方法，*flowinfo* 和 *scope_id* 可以被省略，只为了向后兼容。注意，省略 *scope_id* 可能会导致操作带有领域 (Scope) 的 IPv6 地址时出错。

  3.7 版更變: 对于多播地址（其 *scope_id* 起作用），地址中可以不包含 `%scope_id`（或 `zone id`）部分，这部分是多余的，可以放心省略（推荐）。

- `AF_NETLINK` 套接字由一对 `(pid, groups)` 表示。

- 指定 `AF_TIPC` 地址族可以使用仅 Linux 支持的 TIPC 协议。TIPC 是一种开放的、非基于 IP 的网络协议，旨在用于集群计算环境。其地址用元组表示，其中的字段取决于地址类型。一般元组形式为 `(addr_type, v1, v2, v3 [, scope])`，其中：

  - *addr_type* 取 `TIPC_ADDR_NAMESEQ`、`TIPC_ADDR_NAME` 或 `TIPC_ADDR_ID` 中的一个。
  - *scope* 取 `TIPC_ZONE_SCOPE`、`TIPC_CLUSTER_SCOPE` 和 `TIPC_NODE_SCOPE` 中的一个。
  - 如果 *addr_type* 为 `TIPC_ADDR_NAME`，那么 *v1* 是服务器类型，*v2* 是端口标识符，*v3* 应为 0。

    如果 *addr_type* 为 `TIPC_ADDR_NAMESEQ`，那么 *v1* 是服务器类型，*v2* 是端口号下限，而 *v3* 是端口号上限。

    如果 *addr_type* 为 `TIPC_ADDR_ID`，那么 *v1* 是节点 (node)，*v2* 是 ref，*v3* 应为 0。

- *AF_CAN* 地址族使用元组 `(interface, )`，其中 *interface* 是表示网络接口名称的字符串，如 `'can0'`。网络接口名 `''` 可以用于接收本族所有网络接口的数据包。

  - *CAN_ISOTP* 协议接受一个元组 `(interface, rx_addr, tx_addr)`，其中两个额外参数都是无符号长整数，都表示 CAN 标识符（标准或扩展标识符）。
  - *CAN_J1939* 协议接受一个元组 `(interface, name, pgn, addr)`，其中额外参数有：表示 ECU 名称的 64 位无符号整数，表示参数组号 (Parameter Group Number, PGN) 的 32 位无符号整数，以及表示地址的 8 位整数。

- `PF_SYSTEM` 协议簇的 `SYSPROTO_CONTROL` 协议接受一个字符串或元组 `(id, unit)`。其中字符串是内核控件的名称，该控件使用动态分配的 ID。而如果 ID 和内核控件的单元 (unit) 编号都已知，或使用了已注册的 ID，可以采用元组。

  3.3 版新加入.

- AF_BLUETOOTH 支持以下协议和地址格式：
  - BTPROTO_L2CAP 接受 (bdaddr, psm)，其中 bdaddr 为字符串格式的蓝牙地址，psm 是一个整数。
  - BTPROTO_RFCOMM 接受 (bdaddr, channel)，其中 bdaddr 为字符串格式的蓝牙地址，channel 是一个整数。
  - BTPROTO_HCI 接受 (device_id,)，其中 device_id 为整数或字符串，它表示接口对应的蓝牙地址（具体取决于你的系统，NetBSD 和 DragonFlyBSD 需要蓝牙地址字符串，其他系统需要整数）。

    3.2 版更變: 添加了对 NetBSD 和 DragonFlyBSD 的支持。
  - BTPROTO_SCO 接受 bdaddr，其中 bdaddr 是 *bytes* 对象，其中含有字符串格式的蓝牙地址（如 b'12:23:34:45:56:67'），FreeBSD 不支持此协议。
- *AF_ALG* 是一个仅 Linux 可用的、基于套接字的接口，用于连接内核加密算法。算法套接字可用包括 2 至 4 个元素的元组来配置 (type, name [, feat [, mask]])，其中：
  - *type* 是表示算法类型的字符串，如 aead、hash、skcipher 或 rng。
  - *name* 是表示算法类型和操作模式的字符串，如 sha256、hmac(sha256)、cbc(aes) 或 drbg_nopr_ctr_aes256。
  - *feat* 和 *mask* 是无符号 32 位整数。

  Availability: Linux 2.6.38, some algorithm types require more recent Kernels.

  3.6 版新加入.
- *AF_VSOCK* 用于支持虚拟机与宿主机之间的通讯。该套接字用 (CID, port) 元组表示，其中 Context ID (CID) 和 port 都是整数。

  Availability: Linux >= 4.8 QEMU >= 2.8 ESX >= 4.0 ESX Workstation >= 6.5.

  3.7 版新加入.
- *AF_PACKET* 是一个底层接口，直接连接至网卡。数据包使用元组 (ifname, proto[, pkttype[, hatype[, addr]]]) 表示，其中：
  - *ifname* - 指定设备名称的字符串。
  - *proto* - 一个用网络字节序表示的整数，指定以太网协议版本号。
  - *pkttype* - 指定数据包类型的整数（可选）：
    - PACKET_HOST （默认）- 寻址到本地主机的数据包。
    - PACKET_BROADCAST - 物理层广播的数据包。
    - PACKET_MULTIHOST - 发送到物理层多播地址的数据包。
    - PACKET_OTHERHOST - 被（处于混杂模式的）网卡驱动捕获的、发送到其他主机的数据包。
    - PACKET_OUTGOING - 来自本地主机的、回环到一个套接字的数据包。
  - *hatype* - 可选整数，指定 ARP 硬件地址类型。
  - *addr* - 可选的类字节串对象，用于指定硬件物理地址，其解释取决于各设备。

    Availability: Linux >= 2.2.
- *AF_QIPCRTR* 是一个仅 Linux 可用的、基于套接字的接口，用于与高通平台中协处理器上运行的服务进行通信。该地址簇用一个 (node, port) 元组表示，其中 *node* 和 *port* 为非负整数。

  Availability: Linux >= 4.7.

3.8 版新加入.

- IPPROTO_UDPLITE 是一种 UDP 的变体，允许指定数据包的哪一部分计算入校验码内。它添加了两个可以修改的套接字选项。self.setsockopt(IPPROTO_UDPLITE, UDPLITE_SEND_CSCOV, length) 修改传出数据包的哪一部分计算入校验码内，而 self.setsockopt(IPPROTO_UDPLITE, UDPLITE_RECV_CSCOV, length) 将过滤掉计算入校验码的数据太少的数据包。在这两种情况下，length 都应在 range(8, 2**16, 8) 范围内。

  对于 IPv4，应使用 socket(AF_INET, SOCK_DGRAM, IPPROTO_UDPLITE) 来构造这样的套接字；对于 IPv6，应使用 socket(AF_INET6, SOCK_DGRAM, IPPROTO_UDPLITE) 来构造这样的套接字。

  Availability: Linux >= 2.6.20, FreeBSD >= 10.1-RELEASE

  3.9 版新加入.

如果你在 IPv4/v6 套接字地址的 *host* 部分中使用了一个主机名，此程序可能会表现不确定行为，因为 Python 使用 DNS 解析返回的第一个地址。套接字地址在实际的 IPv4/v6 中以不同方式解析，根据 DNS 解析和/或 host 配置。为了确定行为，在 *host* 部分中使用数字的地址。

所有的错误都抛出异常。对于无效的参数类型和内存溢出异常情况可能抛出普通异常；从 Python 3.3 开始，与套接字或地址语义有关的错误抛出*OSError* 或它的子类之一（常用*socket.error*）。

可以用*setblocking()* 设置非阻塞模式。一个基于超时的 generalization 通过*settimeout()* 支持。

18.2.2 模块内容

socket 模块包含下列元素。

异常

exception socket.**error**
一个被弃用的*OSError* 的别名。

3.3 版更變: 根据 **PEP 3151**，这个类是*OSError* 的别名。

exception socket.**herror**
OSError 的子类，本异常通常表示与地址相关的错误，比如那些在 POSIX C API 中使用了 *h_errno* 的函数，包括*gethostbyname_ex()* 和*gethostbyaddr()*。附带的值是一对 (h_errno, string)，代表库调用返回的错误。*h_errno* 是一个数字，而 *string* 表示 *h_errno* 的描述，它们由 C 函数 hstrerror() 返回。

3.3 版更變: 此类是*OSError* 的子类。

exception socket.**gaierror**
OSError 的子类，本异常来自*getaddrinfo()* 和*getnameinfo()*，表示与地址相关的错误。附带的值是一对 (error, string)，代表库调用返回的错误。*string* 表示 *error* 的描述，它由 C 函数 gai_strerror() 返回。数字值 *error* 与本模块中定义的 EAI_* 常量之一匹配。

3.3 版更變: 此类是*OSError* 的子类。

exception socket.**timeout**
OSError 的子类，当套接字发生超时，且事先已调用过*settimeout()*（或隐式地通过*setdefaulttimeout()*）启用了超时，则会抛出此异常。附带的值是一个字符串，其值总是”timed out”。

3.3 版更變: 此类是*OSError* 的子类。

常数

AF_* 和 SOCK_* 常量现在都在 `AddressFamily` 和 `SocketKind` 这两个*IntEnum*集合内。

3.4 版新加入.

socket.**AF_UNIX**
socket.**AF_INET**
socket.**AF_INET6**
这些常量表示地址（和协议）簇，用于 `socket()` 的第一个参数。如果*AF_UNIX* 常量未定义，即表示不支持该协议。不同系统可能会有更多其他常量可用。

socket.**SOCK_STREAM**
socket.**SOCK_DGRAM**
socket.**SOCK_RAW**
socket.**SOCK_RDM**
socket.**SOCK_SEQPACKET**
这些常量表示套接字类型，用于 `socket()` 的第二个参数。不同系统可能会有更多其他常量可用。(一般只有*SOCK_STREAM* 和*SOCK_DGRAM* 可用)

socket.**SOCK_CLOEXEC**
socket.**SOCK_NONBLOCK**
这两个常量（如果已定义）可以与上述套接字类型结合使用，允许你设置这些原子性相关的 flags （从而避免可能的竞争条件和单独调用的需要）。

也参考:

Secure File Descriptor Handling （安全地处理文件描述符） 提供了更详尽的解释。

可用性： Linux >= 2.6.27。

3.2 版新加入.

SO_*
socket.**SOMAXCONN**
MSG_*
SOL_*
SCM_*
IPPROTO_*
IPPORT_*
INADDR_*
IP_*
IPV6_*
EAI_*
AI_*
NI_*
TCP_*
此列表内的许多常量，记载在 Unix 文档中的套接字和/或 IP 协议部分，同时也定义在本 socket 模块中。它们通常用于套接字对象的 `setsockopt()` 和 `getsockopt()` 方法的参数中。在大多数情况下，仅那些在 Unix 头文件中有定义的符号会在本模块中定义，部分符号提供了默认值。

3.6 版更變: 添加了 `SO_DOMAIN`, `SO_PROTOCOL`, `SO_PEERSEC`, `SO_PASSSEC`, `TCP_USER_TIMEOUT`, `TCP_CONGESTION`。

3.6.5 版更變: 在 Windows 上，如果 Windows 运行时支持，则 `TCP_FASTOPEN`、`TCP_KEEPCNT` 可用。

3.7 版更變: 添加了 `TCP_NOTSENT_LOWAT`。

在 Windows 上，如果 Windows 运行时支持，则 `TCP_KEEPIDLE`、`TCP_KEEPINTVL` 可用。

socket.**AF_CAN**

socket.**PF_CAN**
SOL_CAN_*
CAN_*
此列表内的许多常量，记载在 Linux 文档中，同时也定义在本 socket 模块中。

可用性：Linux >= 2.6.25。

3.3 版新加入.

socket.**CAN_BCM**
CAN_BCM_*
CAN 协议簇内的 CAN_BCM 是广播管理器（Bbroadcast Manager -- BCM）协议，广播管理器常量在 Linux 文档中有所记载，在本 socket 模块中也有定义。

可用性：Linux >= 2.6.25。

備𝔽：CAN_BCM_CAN_FD_FRAME 旗标仅在 Linux >= 4.8 时可用。

3.4 版新加入.

socket.**CAN_RAW_FD_FRAMES**
在 CAN_RAW 套接字中启用 CAN FD 支持，默认是禁用的。它使应用程序可以发送 CAN 和 CAN FD 帧。但是，从套接字读取时，也必须同时接受 CAN 和 CAN FD 帧。

此常量在 Linux 文档中有所记载。

可用性：Linux >= 3.6。

3.5 版新加入.

socket.**CAN_RAW_JOIN_FILTERS**
加入已应用的 CAN 过滤器，这样只有与所有 CAN 过滤器匹配的 CAN 帧才能传递到用户空间。

此常量在 Linux 文档中有所记载。

可用性：Linux >= 4.1。

3.9 版新加入.

socket.**CAN_ISOTP**
CAN 协议簇中的 CAN_ISOTP 就是 ISO-TP (ISO 15765-2) 协议。ISO-TP 常量在 Linux 文档中有所记载。

可用性：Linux >= 2.6.25。

3.7 版新加入.

socket.**CAN_J1939**
CAN 协议族中的 CAN_J1939 即 SAE J1939 协议。J1939 常量记录在 Linux 文档中。

可用性：Linux >= 5.4。

3.9 版新加入.

socket.**AF_PACKET**
socket.**PF_PACKET**
PACKET_*
此列表内的许多常量，记载在 Linux 文档中，同时也定义在本 socket 模块中。

可用性：Linux >= 2.2。

socket.**AF_RDS**
socket.**PF_RDS**
socket.**SOL_RDS**

RDS_*
此列表内的许多常量，记载在 Linux 文档中，同时也定义在本 socket 模块中。

可用性：Linux >= 2.6.30。

3.3 版新加入.

socket.**SIO_RCVALL**
socket.**SIO_KEEPALIVE_VALS**
socket.**SIO_LOOPBACK_FAST_PATH**
RCVALL_*
Windows 的 WSAIoctl() 的常量。这些常量用于套接字对象的*ioctl()* 方法的参数。

3.6 版更變: 添加了 SIO_LOOPBACK_FAST_PATH。

TIPC_*
TIPC 相关常量，与 C socket API 导出的常量一致。更多信息请参阅 TIPC 文档。

socket.**AF_ALG**
socket.**SOL_ALG**
ALG_*
用于 Linux 内核加密算法的常量。

可用性：Linux >= 2.6.38。

3.6 版新加入.

socket.**AF_VSOCK**
socket.**IOCTL_VM_SOCKETS_GET_LOCAL_CID**
VMADDR*
SO_VM*
用于 Linux 宿主机/虚拟机通讯的常量。

可用性：Linux >= 4.8。

3.7 版新加入.

socket.**AF_LINK**
Availability: BSD, macOS.

3.4 版新加入.

socket.**has_ipv6**
本常量为一个布尔值，该值指示当前平台是否支持 IPv6。

socket.**BDADDR_ANY**
socket.**BDADDR_LOCAL**
这些是字符串常量，包含蓝牙地址，这些地址具有特殊含义。例如，当用 BTPROTO_RFCOMM 指定绑定套接字时，*BDADDR_ANY* 表示“任何地址”。

socket.**HCI_FILTER**
socket.**HCI_TIME_STAMP**
socket.**HCI_DATA_DIR**
与 BTPROTO_HCI 一起使用。*HCI_FILTER* 在 NetBSD 或 DragonFlyBSD 上不可用。*HCI_TIME_STAMP* 和*HCI_DATA_DIR* 在 FreeBSD、NetBSD 或 DragonFlyBSD 上不可用。

socket.**AF_QIPCRTR**
高通 IPC 路由协议的常数，用于与提供远程处理器的服务进行通信。

可用性：Linux >= 4.7。

函数

创建套接字

下列函数都能创建套接字对象.

class socket.**socket**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

使用给定的地址族、套接字类型和协议号创建一个新的套接字。地址族应为*AF_INET* (默认值), *AF_INET6*, *AF_UNIX*, *AF_CAN*, *AF_PACKET* 或*AF_RDS* 之一。套接字类型应为*SOCK_STREAM* (默认值), *SOCK_DGRAM*, *SOCK_RAW* 或其他可能的 SOCK_ 常量之一。协议号通常为零并且可以省略，或在协议族为*AF_CAN* 的情况下，协议应为 CAN_RAW, *CAN_BCM*, *CAN_ISOTP* 或*CAN_J1939* 之一。

如果指定了 *fileno*，那么将从这一指定的文件描述符中自动检测 *family*、*type* 和 *proto* 的值。如果调用本函数时显式指定了 *family*、*type* 或 *proto* 参数，可以覆盖自动检测的值。这只会影响 Python 表示诸如*socket.getpeername()* 一类函数的返回值的方式，而不影响实际的操作系统资源。与*socket.fromfd()* 不同，*fileno* 将返回原先的套接字，而不是复制出新的套接字。这有助于在分离的套接字上调用*socket.close()* 来关闭它。

新创建的套接字是不可继承的。

引发一个审计事件 socket.__new__ 附带参数 self、family、type、protocol。

3.3 版更變: 添加了 AF_CAN 簇。添加了 AF_RDS 簇。

3.4 版更變: 添加了 CAN_BCM 协议。

3.4 版更變: 返回的套接字现在是不可继承的。

3.7 版更變: 添加了 CAN_ISOTP 协议。

3.7 版更變: 当将*SOCK_NONBLOCK* 或*SOCK_CLOEXEC* 标志位应用于 *type* 上时，它们会被清除，且*socket.type* 反映不出它们。但它们仍将传递给底层系统的 *socket()* 调用。因此，

```
sock = socket.socket(
    socket.AF_INET,
    socket.SOCK_STREAM | socket.SOCK_NONBLOCK)
```

仍将在支持 SOCK_NONBLOCK 的系统上创建一个非阻塞的套接字，但是 sock.type 会被置为 socket.SOCK_STREAM。

3.9 版更變: 添加了 CAN_J1939 协议。

socket.**socketpair**([*family*[, *type*[, *proto*]]])

构建一对已连接的套接字对象，使用给定的地址簇、套接字类型和协议号。地址簇、套接字类型和协议号与上述 socket() 函数相同。默认地址簇为*AF_UNIX* （需要当前平台支持，不支持则默认为*AF_INET* ）。

新创建的套接字都是不可继承的。

3.2 版更變: 现在，返回的套接字对象支持全部套接字 API，而不是全部 API 的一个子集。

3.4 版更變: 返回的套接字现在都是不可继承的。

3.5 版更變: 添加了 Windows 支持。

socket.**create_connection**(*address*[, *timeout*[, *source_address*]])

连接到一个 TCP 服务，该服务正在侦听 Internet *address* （用二元组 (host, port) 表示）。连接后返回套接字对象。这是比*socket.connect()* 更高级的函数：如果 *host* 是非数字主机名，它将尝试从*AF_INET* 和*AF_INET6* 解析它，然后依次尝试连接到所有可能的地址，直到连接成功。这使得编写兼容 IPv4 和 IPv6 的客户端变得容易。

传入可选参数 *timeout* 可以在套接字实例上设置超时（在尝试连接前）。如果未提供 *timeout*，则使用由*getdefaulttimeout()* 返回的全局默认超时设置。

如果提供了 *source_address*，它必须为二元组 (host, port)，以便套接字在连接之前绑定为其源地址。如果 host 或 port 分别为" 或 0，则使用操作系统默认行为。

3.2 版更變: 添加了 *source_address*。

socket.**create_server**(*address*, *, *family=AF_INET*, *backlog=None*, *reuse_port=False*, *dualstack_ipv6=False*)

便捷函数，创建绑定到 *address* （二元组 (host, port) ）的 TCP 套接字，返回套接字对象。

family 应设置为*AF_INET* 或*AF_INET6*。*backlog* 是传递给*socket.listen()* 的队列大小，当它为 0 则表示默认的合理值。*reuse_port* 表示是否设置 SO_REUSEPORT 套接字选项。

如果 *dualstack_ipv6* 为 true 且平台支持，则套接字能接受 IPv4 和 IPv6 连接，否则将抛出*ValueError* 异常。大多数 POSIX 平台和 Windows 应该支持此功能。启用此功能后，*socket.getpeername()* 在进行 IPv4 连接时返回的地址将是一个（映射到 IPv4 的）IPv6 地址。在默认启用该功能的平台上（如 Linux），如果 *dualstack_ipv6* 为 false，即显式禁用此功能。该参数可以与*has_dualstack_ipv6()* 结合使用：

```
import socket

addr = ("", 8080)  # all interfaces, port 8080
if socket.has_dualstack_ipv6():
    s = socket.create_server(addr, family=socket.AF_INET6, dualstack_ipv6=True)
else:
    s = socket.create_server(addr)
```

備F: 在 POSIX 平台上，设置 SO_REUSEADDR 套接字选项是为了立即重用以前绑定在同一 *address* 上并保持 TIME_WAIT 状态的套接字。

3.8 版新加入.

socket.**has_dualstack_ipv6**()

如果平台支持创建 IPv4 和 IPv6 连接都可以处理的 TCP 套接字，则返回 True。

3.8 版新加入.

socket.**fromfd**(*fd*, *family*, *type*, *proto=0*)

复制文件描述符 *fd* （一个由文件对象的 fileno() 方法返回的整数），然后从结果中构建一个套接字对象。地址簇、套接字类型和协议号与上述 socket() 函数相同。文件描述符应指向一个套接字，但不会专门去检查——如果文件描述符是无效的，则对该对象的后续操作可能会失败。本函数很少用到，但是在将套接字作为标准输入或输出传递给程序（如 Unix inet 守护程序启动的服务器）时，可以使用本函数获取或设置套接字选项。套接字将处于阻塞模式。

新创建的套接字是不可继承的。

3.4 版更變: 返回的套接字现在是不可继承的。

socket.**fromshare**(*data*)

根据*socket.share()* 方法获得的数据实例化套接字。套接字将处于阻塞模式。

可用性: Windows。

3.3 版新加入.

socket.**SocketType**

这是一个 Python 类型对象，表示套接字对象的类型。它等同于 type(socket(...))。

其他功能

socket 模块还提供多种网络相关服务：

socket.**close**(*fd*)
关闭一个套接字文件描述符。它类似于*os.close()*，但专用于套接字。在某些平台上（特别是在 Windows 上），*os.close()* 对套接字文件描述符无效。

3.7 版新加入.

socket.**getaddrinfo**(*host*, *port*, *family=0*, *type=0*, *proto=0*, *flags=0*)
将 *host*/*port* 参数转换为 5 元组的序列，其中包含创建（连接到某服务的）套接字所需的所有参数。*host* 是域名，是字符串格式的 IPv4/v6 地址或 `None`。*port* 是字符串格式的服务名称，如 `'http'`、端口号（数字）或 `None`。传入 `None` 作为 *host* 和 *port* 的值，相当于将 `NULL` 传递给底层 C API。

可以指定 *family*、*type* 和 *proto* 参数，以缩小返回的地址列表。向这些参数分别传入 0 表示保留全部结果范围。*flags* 参数可以是 `AI_*` 常量中的一个或多个，它会影响结果的计算和返回。例如，`AI_NUMERICHOST` 会禁用域名解析，此时如果 *host* 是域名，则会抛出错误。

本函数返回一个列表，其中的 5 元组具有以下结构：

```
(family, type, proto, canonname, sockaddr)
```

在这些元组中，*family*, *type*, *proto* 都是整数且其作用是被传入 `socket()` 函数。如果 `AI_CANONNAME` 是 *flags* 参数的一部分则 *canonname* 将是表示 *host* 规范名称的字符串；否则 *canonname* 将为空。*sockaddr* 是一个描述套接字地址的元组，其具体格式取决于返回的 *family* (对于*AF_INET* 为 `(address, port)` 2 元组，对于*AF_INET6* 则为 `(address, port, flowinfo, scope_id)` 4 元组)，其作用是被传入*socket.connect()* 方法。

引发一个审计事件 `socket.getaddrinfo` 附带参数 `host`、`port`、`family`、`type`、`protocol`。

下面的示例获取了 TCP 连接地址信息，假设该连接通过 80 端口连接至 `example.org`（如果系统未启用 IPv6，则结果可能会不同）：

```
>>> socket.getaddrinfo("example.org", 80, proto=socket.IPPROTO_TCP)
[(<AddressFamily.AF_INET6: 10>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
 (<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
 6, '', ('93.184.216.34', 80))]
```

3.2 版更變: 现在可以使用关键字参数的形式来传递参数。

3.7 版更變: 对于 IPv6 多播地址，表示地址的字符串将不包含 `%scope_id` 部分。

socket.**getfqdn**([*name*])
Return a fully qualified domain name for *name*. If *name* is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by *gethostbyaddr()* is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available and *name* was provided, it is returned unchanged. If *name* was empty or equal to `'0.0.0.0'`, the hostname from *gethostname()* is returned.

socket.**gethostbyname**(*hostname*)
将主机名转换为 IPv4 地址格式。IPv4 地址以字符串格式返回，如 `'100.50.200.5'`。如果主机名本身是 IPv4 地址，则原样返回。更完整的接口请参考*gethostbyname_ex()*。*gethostbyname()* 不支持 IPv6 名称解析，应使用*getaddrinfo()* 来支持 IPv4/v6 双协议栈。

引发一个审计事件 `socket.gethostbyname`，附带参数 `hostname`。

socket.**gethostbyname_ex**(*hostname*)
Translate a host name to IPv4 address format, extended interface. Return a triple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the host's primary host name, *aliaslist* is a (possibly empty) list of alternative

host names for the same address, and *ipaddrlist* is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). *gethostbyname_ex()* does not support IPv6 name resolution, and *getaddrinfo()* should be used instead for IPv4/v6 dual stack support.

引发一个审计事件 socket.gethostbyname，附带参数 hostname。

socket.**gethostname**()
返回一个字符串，包含当前正在运行 Python 解释器的机器的主机名。

引发一个审计事件 socket.gethostname，没有附带参数。

注意：*gethostname()* 并不总是返回全限定域名，必要的话请使用*getfqdn()*。

socket.**gethostbyaddr**(*ip_address*)
返回三元组 (hostname, aliaslist, ipaddrlist)，其中 *hostname* 是响应给定 *ip_address* 的主要主机名，*aliaslist* 是相同地址的其他可用主机名的列表（可能为空），而 *ipaddrlist* 是 IPv4/v6 地址列表，包含相同主机名、相同接口的不同地址（很可能仅包含一个地址）。要查询全限定域名，请使用函数*getfqdn()*。*gethostbyaddr()* 支持 IPv4 和 IPv6。

引发一个审计事件 socket.gethostbyaddr，附带参数 ip_address。

socket.**getnameinfo**(*sockaddr*, *flags*)
将套接字地址 *sockaddr* 转换为二元组 (host, port)。*host* 的形式可能是全限定域名，或是由数字表示的地址，具体取决于 *flags* 的设置。同样，*port* 可以包含字符串格式的端口名称或数字格式的端口号。

对于 IPv6 地址，如果 *sockaddr* 包含有意义的 *scope_id*，则 %scope_id 会被附加到主机部分。这种情况通常发生在多播地址上。

关于 *flags* 的更多信息可参阅 *getnameinfo(3)*。

引发一个审计事件 socket.getnameinfo，附带参数 sockaddr。

socket.**getprotobyname**(*protocolname*)
将 Internet 协议名称（如 'icmp'）转换为常量，该常量适用于 socket() 函数的第三个（可选）参数。通常只有在"raw" 模式 (*SOCK_RAW*) 中打开的套接字才需要使用该常量。在正常的套接字模式下，协议省略或为零时，会自动选择正确的协议。

socket.**getservbyname**(*servicename*[, *protocolname*])
将 Internet 服务名称和协议名称转换为该服务的端口号。协议名称是可选的，如果提供的话应为 'tcp' 或 'udp'，否则将匹配出所有协议。

引发一个审计事件 socket.getservbyname，附带参数 servicename、protocolname。

socket.**getservbyport**(*port*[, *protocolname*])
将 Internet 端口号和协议名称转换为该服务的服务名称。协议名称是可选的，如果提供的话应为 'tcp' 或 'udp'，否则将匹配出所有协议。

引发一个审计事件 socket.getservbyport，附带参数 port、protocolname。

socket.**ntohl**(*x*)
将 32 位正整数从网络字节序转换为主机字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 4 字节交换操作。

socket.**ntohs**(*x*)
将 16 位正整数从网络字节序转换为主机字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 2 字节交换操作。

3.7 版後已F用: 如果 *x* 不符合 16 位无符号整数，但符合 C 语言的正整数，则它会被静默截断为 16 位无符号整数。此静默截断功能已弃用，未来版本的 Python 将对此抛出异常。

socket.**htonl**(*x*)
将 32 位正整数从主机字节序转换为网络字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 4 字节交换操作。

socket.**htons**(*x*)
将 16 位正整数从主机字节序转换为网络字节序。在主机字节序与网络字节序相同的计算机上，这是一个空操作。字节序不同将执行 2 字节交换操作。

3.7 版後已🄴用: 如果 *x* 不符合 16 位无符号整数，但符合 C 语言的正整数，则它会被静默截断为 16 位无符号整数。此静默截断功能已弃用，未来版本的 Python 将对此抛出异常。

socket.**inet_aton**(*ip_string*)
将 IPv4 地址从点分十进制字符串格式（如'123.45.67.89'）转换为 32 位压缩二进制格式，转换后为字节对象，长度为四个字符。与那些使用标准 C 库，且需要 `struct in_addr` 类型的对象的程序交换信息时，本函数很有用。该类型即本函数返回的 32 位压缩二进制的 C 类型。

inet_aton() 也接受句点数少于三的字符串，详情请参阅 Unix 手册 *inet(3)*。

如果传入本函数的 IPv4 地址字符串无效，则抛出*OSError*。注意，具体什么样的地址有效取决于 `inet_aton()` 的底层 C 实现。

inet_aton() 不支持 IPv6，在 IPv4/v6 双协议栈下应使用*inet_pton()* 来代替。

socket.**inet_ntoa**(*packed_ip*)
将 32 位压缩 IPv4 地址（一个类字节对象，长 4 个字节）转换为标准的点分十进制字符串形式（如'123.45.67.89'）。与那些使用标准 C 库，且需要 `struct in_addr` 类型的对象的程序交换信息时，本函数很有用。该类型即本函数参数中的 32 位压缩二进制数据的 C 类型。

如果传入本函数的字节序列长度不是 4 个字节，则抛出*OSError*。*inet_ntoa()* 不支持 IPv6，在 IPv4/v6 双协议栈下应使用*inet_ntop()* 来代替。

3.5 版更變: 现在支持可写的字节类对象。

socket.**inet_pton**(*address_family*, *ip_string*)
将特定地址簇的 IP 地址（字符串）转换为压缩二进制格式。当库或网络协议需要接受 `struct in_addr` 类型的对象（类似*inet_aton()*）或 `struct in6_addr` 类型的对象时，*inet_pton()* 很有用。

目前 *address_family* 支持*AF_INET* 和*AF_INET6*。如果 IP 地址字符串 *ip_string* 无效，则抛出*OSError*。注意，具体什么地址有效取决于 *address_family* 的值和 `inet_pton()` 的底层实现。

可用性：Unix（可能非所有平台都可用）、Windows。

3.4 版更變: 添加了 Windows 支持

socket.**inet_ntop**(*address_family*, *packed_ip*)
将压缩 IP 地址（一个类字节对象，数个字节长）转换为标准的、特定地址簇的字符串形式（如 `'7.10.0.5'` 或 `'5aef:2b::8'`）。当库或网络协议返回 `struct in_addr` 类型的对象（类似*inet_ntoa()*）或 `struct in6_addr` 类型的对象时，*inet_ntop()* 很有用。

目前 *address_family* 支持*AF_INET* 和*AF_INET6*。如果字节对象 *packed_ip* 与指定的地址簇长度不符，则抛出*ValueError*。针对*inet_ntop()* 调用的错误则抛出*OSError*。

可用性：Unix（可能非所有平台都可用）、Windows。

3.4 版更變: 添加了 Windows 支持

3.5 版更變: 现在支持可写的字节类对象。

socket.**CMSG_LEN**(*length*)
返回给定 *length* 所关联数据的辅助数据项的总长度（不带尾部填充）。此值通常用作*recvmsg()* 接收一个辅助数据项的缓冲区大小，但是 **RFC 3542** 要求可移植应用程序使用*CMSG_SPACE()*，以此将尾部填充的空间计入，即使该项在缓冲区的最后。如果 *length* 超出允许范围，则抛出*OverflowError*。

可用性：大多数 Unix 平台，其他平台也可能可用。

3.3 版新加入.

socket.**CMSG_SPACE**(*length*)
返回*recvmsg()* 所需的缓冲区大小，以接收给定 *length* 所关联数据的辅助数据项，带有尾部填充。接收多个项目所需的缓冲区空间是关联数据长度的*CMSG_SPACE()* 值的总和。如果 *length* 超出允许范围，则抛出*OverflowError*。

请注意，某些系统可能支持辅助数据，但不提供本函数。还需注意，如果使用本函数的结果来设置缓冲区大小，可能无法精确限制可接收的辅助数据量，因为可能会有其他数据写入尾部填充区域。

可用性：大多数 Unix 平台，其他平台也可能可用。

3.3 版新加入.

socket.**getdefaulttimeout**()
返回用于新套接字对象的默认超时（以秒为单位的浮点数）。值 None 表示新套接字对象没有超时。首次导入 socket 模块时，默认值为 None。

socket.**setdefaulttimeout**(*timeout*)
设置用于新套接字对象的默认超时（以秒为单位的浮点数）。首次导入 socket 模块时，默认值为 None。可能的取值及其各自的含义请参阅*settimeout()*。

socket.**sethostname**(*name*)
将计算机的主机名设置为 *name*。如果权限不足将抛出*OSError*。

引发一个审计事件 socket.sethostname，附带参数 name。

Availability: Unix.

3.3 版新加入.

socket.**if_nameindex**()
返回一个列表，包含网络接口（网卡）信息二元组（整数索引，名称字符串）。系统调用失败则抛出*OSError*。

可用性: Unix, Windows。

3.3 版新加入.

3.8 版更變: 添加了 Windows 支持。

備𝔽: 在 Windows 中网络接口在不同上下文中具有不同的名称（所有名称见对应示例）：

- UUID: {FB605B73-AAC2-49A6-9A2F-25416AEA0573}
- 名称: ethernet_32770
- 友好名称: vEthernet (nat)
- 描述: Hyper-V Virtual Ethernet Adapter

此函数返回列表中第二种形式的名称，在此示例中为 ethernet_32770。

---

socket.**if_nametoindex**(*if_name*)
返回网络接口名称相对应的索引号。如果没有所给名称的接口，则抛出*OSError*。

可用性: Unix, Windows。

3.3 版新加入.

3.8 版更變: 添加了 Windows 支持。

也参考:

”Interface name” 为*if_nameindex()* 中所描述的名称。

socket.**if_indextoname**(*if_index*)
返回网络接口索引号相对应的接口名称。如果没有所给索引号的接口，则抛出*OSError*。

可用性: Unix, Windows。

3.3 版新加入.

3.8 版更變: 添加了 Windows 支持。

也參考:

”Interface name” 为*if_nameindex()* 中所描述的名称。

socket.**send_fds**(*sock*, *buffers*, *fds*[, *flags*[, *address*]])
将文件描述符列表 *fds* 通过一个*AF_UNIX* 套接字 *sock* 进行发送。*fds* 形参是由文件描述符构成的序列。请查看 sendmsg() 获取这些形参的文档。

可用性: 支持*sendmsg()* 和 SCM_RIGHTS 机制的 Unix 系统。

3.9 版新加入.

socket.**recv_fds**(*sock*, *bufsize*, *maxfds*[, *flags*])
接收至多 *maxfds* 个来自*AF_UNIX* 套接字 *sock* 的文件描述符。返回 (msg, list(fds), flags, addr)。请查看 recvmsg() 获取有些形参的文档。

可用性: 支持*recvmsg()* 和 SCM_RIGHTS 机制的 Unix 系统。

3.9 版新加入.

備🄵: 位于文件描述符列表末尾的任何被截断整数。

18.2.3 套接字对象

套接字对象具有以下方法。除了*makefile()*，其他都与套接字专用的 Unix 系统调用相对应。

3.2 版更變: 添加了对上下文管理器 协议的支持。退出上下文管理器与调用*close()* 等效。

socket.**accept**()
接受一个连接。此 socket 必须绑定到一个地址上并且监听连接。返回值是一个 (conn, address) 对，其中 *conn* 是一个 新的套接字对象，用于在此连接上收发数据，*address* 是连接另一端的套接字所绑定的地址。

新创建的套接字是不可继承的。

3.4 版更變: 该套接字现在是不可继承的。

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**bind**(*address*)
将套接字绑定到 *address*。套接字必须尚未绑定。(*address* 的格式取决于地址簇——参见上文)

引发一个审计事件 socket.bind，附带参数 self、address。

socket.**close**()
将套接字标记为关闭。当*makefile()* 创建的所有文件对象都关闭时，底层系统资源（如文件描述符）也将关闭。一旦上述情况发生，将来对套接字对象的所有操作都会失败。对端将接收不到任何数据（清空队列数据后）。

垃圾回收时，套接字会自动关闭，但建议显式*close()* 它们，或在它们周围使用 with 语句。

3.6 版更變: 现在，如果底层的 close() 调用出错，会抛出*OSError*。

備🄵: *close()* 释放与连接相关联的资源，但不一定立即关闭连接。如果需要及时关闭连接，请在调用*close()* 之前调用*shutdown()*。

socket.**connect**(*address*)
连接到 *address* 处的远程套接字。（*address* 的格式取决于地址簇——参见上文）

如果连接被信号中断，则本方法将等待，直到连接完成。如果信号处理程序未抛出异常，且套接字阻塞中或已超时，则在超时后抛出*socket.timeout*。对于非阻塞套接字，如果连接被信号中断，则本方法将抛出*InterruptedError* 异常（或信号处理程序抛出的异常）。

引发一个审计事件 socket.connect，附带参数 self、address。

3.5 版更變: 本方法现在将等待，直到连接完成，而不是在以下情况抛出*InterruptedError* 异常。该情况为，连接被信号中断，信号处理程序未抛出异常，且套接字阻塞中或已超时（具体解释请参阅 **PEP 475** ）。

socket.**connect_ex**(*address*)
类似于 connect(address)，但是对于 C 级别的 connect() 调用返回的错误，本函数将返回错误指示器，而不是抛出异常（对于其他问题，如 “找不到主机”，仍然可以抛出异常）。如果操作成功，则错误指示器为 0，否则为 errno 变量的值。这对支持如异步连接很有用。

引发一个审计事件 socket.connect，附带参数 self、address。

socket.**detach**()
将套接字对象置于关闭状态，而底层的文件描述符实际并不关闭。返回该文件描述符，使其可以重新用于其他目的。

3.2 版新加入.

socket.**dup**()
创建套接字的副本。

新创建的套接字是不可继承的。

3.4 版更變: 该套接字现在是不可继承的。

socket.**fileno**()
返回套接字的文件描述符（一个小整数），失败返回 -1。配合*select.select()* 使用很有用。

在 Windows 下，此方法返回的小整数在允许使用文件描述符的地方无法使用（如*os.fdopen()* ）。Unix 无此限制。

socket.**get_inheritable**()
获取套接字文件描述符或套接字句柄的可继承标志：如果子进程可以继承套接字则为 True，否则为 False。

3.4 版新加入.

socket.**getpeername**()
返回套接字连接到的远程地址。举例而言，这可以用于查找远程 IPv4/v6 套接字的端口号。（返回的地址格式取决于地址簇——参见上文。）部分系统不支持此函数。

socket.**getsockname**()
返回套接字本身的地址。举例而言，这可以用于查找 IPv4/v6 套接字的端口号。（返回的地址格式取决于地址簇——参见上文。）

socket.**getsockopt**(*level*, *optname*[, *buflen*])
返回指定套接字选项的值（参阅 Unix 手册页 *getsockopt(2)* ）。所需的符号常量（SO_* 等）已定义在本模块中。如果未指定 *buflen*，则认为该选项值为整数，由本函数返回该整数值。如果指定 *buflen*，则它定义了用于存放选项值的缓冲区的最大长度，且该缓冲区将作为字节对象返回。对缓冲区的解码工作由调用者自行完成（针对编码为字节串的 C 结构，其解码方法请参阅可选的内置模块*struct* ）。

socket.**getblocking**()
如果套接字处于阻塞模式，返回 True，非阻塞模式返回 False。

这相当于检测 socket.gettimeout() == 0。

3.7 版新加入.

socket.**gettimeout**()
返回套接字操作相关的超时秒数（浮点数），未设置超时则返回 None。它反映最后一次调用*setblocking()* 或*settimeout()* 后的设置。

socket.**ioctl**(*control*, *option*)

平台 Windows

ioctl() 方法是 WSAIoctl 系统接口的有限接口。请参考 Win32 文档 以获取更多信息。

在其他平台上，可以使用通用的*fcntl.fcntl()* 和*fcntl.ioctl()* 函数，它们接受套接字对象作为第一个参数。

当前仅支持以下控制码：SIO_RCVALL、SIO_KEEPALIVE_VALS 和 SIO_LOOPBACK_FAST_PATH。

3.6 版更變: 添加了 SIO_LOOPBACK_FAST_PATH。

socket.**listen**([*backlog*])
启动一个服务器用于接受连接。如果指定 *backlog*，则它最低为 0（小于 0 会被置为 0），它指定系统允许暂未 accept 的连接数，超过后将拒绝新连接。未指定则自动设为合理的默认值。

3.5 版更變: *backlog* 参数现在是可选的。

socket.**makefile**(*mode='r'*, *buffering=None*, ***, *encoding=None*, *errors=None*, *newline=None*)
返回与套接字关联的文件对象。返回的对象的具体类型取决于*makefile()* 的参数。这些参数的解释方式与内置的*open()* 函数相同，其中 *mode* 的值仅支持 'r'（默认），'w' 和 'b'。

套接字必须处于阻塞模式，它可以有超时，但是如果发生超时，文件对象的内部缓冲区可能会以不一致的状态结尾。

关闭*makefile()* 返回的文件对象不会关闭原始套接字，除非所有其他文件对象都已关闭且在套接字对象上调用了*socket.close()*。

備𝔽: 在 Windows 上，由*makefile()* 创建的文件类对象无法作为带文件描述符的文件对象使用，如无法作为*subprocess.Popen()* 的流参数。

---

socket.**recv**(*bufsize*[, *flags*])
从套接字接收数据。返回值是一个字节对象，表示接收到的数据。*bufsize* 指定一次接收的最大数据量。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。

備𝔽: 为了最佳匹配硬件和网络的实际情况，*bufsize* 的值应为 2 的相对较小的幂，如 4096。

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**recvfrom**(*bufsize*[, *flags*])
从套接字接收数据。返回值是一对 (bytes, address)，其中 *bytes* 是字节对象，表示接收到的数据，*address* 是发送端套接字的地址。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。(*address* 的格式取决于地址簇——参见上文)

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

3.7 版更變: 对于多播 IPv6 地址，*address* 的第一项不会再包含 `%scope_id` 部分。要获得完整的 IPv6 地址请使用*getnameinfo()*。

socket.**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])

从套接字接收普通数据（至多 *bufsize* 字节）和辅助数据。*ancbufsize* 参数设置用于接收辅助数据的内部缓冲区的大小（以字节为单位），默认为 0，表示不接收辅助数据。可以使用*CMSG_SPACE()*或*CMSG_LEN()* 计算辅助数据缓冲区的合适大小，无法放入缓冲区的项目可能会被截断或丢弃。*flags* 参数默认为 0，其含义与*recv()* 中的相同。

返回值是一个四元组：`(data, ancdata, msg_flags, address)`。*data* 项是一个*bytes* 对象，用于保存接收到的非辅助数据。*ancdata* 项是零个或多个元组 `(cmsg_level, cmsg_type, cmsg_data)` 组成的列表，表示接收到的辅助数据（控制消息）：*cmsg_level* 和 *cmsg_type* 是分别表示协议级别和协议类型的整数，而 *cmsg_data* 是保存相关数据的*bytes* 对象。*msg_flags* 项由各种标志按位或组成，表示接收消息的情况，详细信息请参阅系统文档。如果接收端套接字断开连接，则 *address* 是发送端套接字的地址（如果有），否则该值无指定。

某些系统上可以利用*AF_UNIX* 套接字通过*sendmsg()* 和*recvmsg()* 在进程之间传递文件描述符。使用此功能时（通常仅限于*SOCK_STREAM* 套接字），*recvmsg()* 将在其辅助数据中返回以下格式的项 `(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds)`，其中 *fds* 是一个*bytes* 对象，是新文件描述符表示为原生 C `int` 类型的二进制数组。如果*recvmsg()* 在系统调用返回后抛出异常，它将首先关闭此机制接收到的所有文件描述符。

对于仅接收到一部分的辅助数据项，一些系统没有指示其截断长度。如果某个项目可能超出了缓冲区的末尾，*recvmsg()* 将发出*RuntimeWarning*，并返回其在缓冲区内的部分，前提是该对象被截断于关联数据开始后。

在支持 `SCM_RIGHTS` 机制的系统上，下方的函数将最多接收 *maxfds* 个文件描述符，返回消息数据和包含描述符的列表（同时忽略意外情况，如接收到无关的控制消息）。另请参阅*sendmsg()*。

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.CMSG_LEN(maxfds * fds.
↪itemsize))
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS:
            # Append data, ignoring any truncated integers at the end.
            fds.frombytes(cmsg_data[:len(cmsg_data) - (len(cmsg_data) % fds.
↪itemsize)])
    return msg, list(fds)
```

可用性：大多数 Unix 平台，其他平台也可能可用。

3.3 版新加入.

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**recvmsg_into**(*buffers*[, *ancbufsize*[, *flags*]])

从套接字接收普通数据和辅助数据，其行为与*recvmsg()* 相同，但将非辅助数据分散到一系列缓冲区中，而不是返回新的字节对象。*buffers* 参数必须是可迭代对象，它迭代出可供写入的缓冲区（如*bytearray* 对象），这些缓冲区将被连续的非辅助数据块填充，直到数据全部写完或缓冲区用完为止。在允许使用的缓冲区数量上，操作系统可能会有限制（*sysconf()* 的 `SC_IOV_MAX` 值）。*ancbufsize* 和 *flags* 参数的含义与*recvmsg()* 中的相同。

返回值为四元组：`(nbytes, ancdata, msg_flags, address)`，其中 *nbytes* 是写入缓冲区的非辅助数据的字节总数，而 *ancdata*、*msg_flags* 和 *address* 与*recvmsg()* 中的相同。

示例:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'little lamb---')]
```

可用性：大多数 Unix 平台，其他平台也可能可用。

3.3 版新加入.

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])
从套接字接收数据，将其写入 *buffer* 而不是创建新的字节串。返回值是一对 `(nbytes, address)`，其中 *nbytes* 是收到的字节数，*address* 是发送端套接字的地址。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。(*address* 的格式取决于地址簇——参见上文)

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])
从套接字接收至多 *nbytes* 个字节，将其写入缓冲区而不是创建新的字节串。如果 *nbytes* 未指定（或指定为 0），则接收至所给缓冲区的最大可用大小。返回接收到的字节数。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。

socket.**send**(*bytes*[, *flags*])
发送数据给套接字。本套接字必须已连接到远程套接字。可选参数 *flags* 的含义与上述 *recv()* 中的相同。本方法返回已发送的字节数。应用程序要负责检查所有数据是否已发送，如果仅传输了部分数据，程序需要自行尝试传输其余数据。有关该主题的更多信息，请参考 socket-howto。

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**sendall**(*bytes*[, *flags*])
发送数据给套接字。本套接字必须已连接到远程套接字。可选参数 *flags* 的含义与上述 *recv()* 中的相同。与 *send()* 不同，本方法持续从 *bytes* 发送数据，直到所有数据都已发送或发生错误为止。成功后会返回 `None`。出错后会抛出一个异常，此时并没有办法确定成功发送了多少数据。

3.5 版更變: 每次成功发送数据后，套接字超时不再重置。现在，套接字超时是发送所有数据的最大总持续时间。

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**sendto**(*bytes*, *address*)
socket.**sendto**(*bytes*, *flags*, *address*)
发送数据给套接字。本套接字不应连接到远程套接字，而应由 *address* 指定目标套接字。可选参数 *flags* 的含义与上述 *recv()* 中的相同。本方法返回已发送的字节数。(*address* 的格式取决于地址簇——参见上文。)

引发一个审计事件 `socket.sendto`，附带参数 `self`、`address`。

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发 *InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**sendmsg**(*buffers*[, *ancdata*[, *flags*[, *address*]]])
将普通数据和辅助数据发送给套接字，将从一系列缓冲区中收集非辅助数据，并将其拼接为一条消息。*buffers* 参数指定的非辅助数据应为可迭代的字节类对象 (如 *bytes* 对象)，在允许使用的缓冲区数量上，操作系统可能会有限制 (*sysconf()* 的 `SC_IOV_MAX` 值)。*ancdata* 参数指定的辅助数据 (控制

消息）应为可迭代对象，迭代出零个或多个 (cmsg_level, cmsg_type, cmsg_data) 元组，其中 *cmsg_level* 和 *cmsg_type* 是分别指定协议级别和协议类型的整数，而 *cmsg_data* 是保存相关数据的字节类对象。请注意，某些系统（特别是没有*CMSG_SPACE()* 的系统）可能每次调用仅支持发送一条控制消息。*flags* 参数默认为 0，与*send()* 中的含义相同。如果 *address* 指定为除 None 以外的值，它将作为消息的目标地址。返回值是已发送的非辅助数据的字节数。

在支持 SCM_RIGHTS 机制的系统上，下方的函数通过一个*AF_UNIX* 套接字来发送文件描述符列表 *fds*。另请参阅*recvmsg()*。

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SCM_RIGHTS, array.
↪array("i", fds))])
```

可用性：大多数 Unix 平台，其他平台也可能可用。

引发一个审计事件 socket.sendmsg，附带参数 self、address。

3.3 版新加入.

3.5 版更變: 如果系统调用被中断，但信号处理程序没有触发异常，此方法现在会重试系统调用，而不是触发*InterruptedError* 异常 (原因详见 **PEP 475**)。

socket.**sendmsg_afalg**([*msg*], *, *op*[, *iv*[, *assoclen*[, *flags*]]])
为*AF_ALG* 套接字定制的*sendmsg()* 版本。可为*AF_ALG* 套接字设置模式、IV、AEAD 关联数据的长度和标志位。

可用性：Linux >= 2.6.38。

3.6 版新加入.

socket.**sendfile**(*file*, *offset=0*, *count=None*)
使用高性能的*os.sendfile* 发送文件，直到达到文件的 EOF 为止，返回已发送的字节总数。*file* 必须是一个以二进制模式打开的常规文件对象。如果*os.sendfile* 不可用（如 Windows）或 *file* 不是常规文件，将使用*send()* 代替。*offset* 指示从哪里开始读取文件。如果指定了 *count*，它确定了要发送的字节总数，而不会持续发送直到达到文件的 EOF。返回时或发生错误时，文件位置将更新，在这种情况下，*file.tell()* 可用于确定已发送的字节数。套接字必须为*SOCK_STREAM* 类型。不支持非阻塞的套接字。

3.5 版新加入.

socket.**set_inheritable**(*inheritable*)
设置套接字文件描述符或套接字句柄的可继承标志。

3.4 版新加入.

socket.**setblocking**(*flag*)
设置套接字为阻塞或非阻塞模式：如果 *flag* 为 false，则将套接字设置为非阻塞，否则设置为阻塞。

本方法是某些*settimeout()* 调用的简写：

- sock.setblocking(True) 相当于 sock.settimeout(None)
- sock.setblocking(False) 相当于 sock.settimeout(0.0)

3.7 版更變: 本方法不再对*socket.type* 属性设置*SOCK_NONBLOCK* 标志。

socket.**settimeout**(*value*)
为阻塞套接字的操作设置超时。*value* 参数可以是非负浮点数，表示秒，也可以是 None。如果赋为一个非零值，那么如果在操作完成前超过了超时时间 *value*，后续的套接字操作将抛出*timeout* 异常。如果赋为 0，则套接字将处于非阻塞模式。如果指定为 None，则套接字将处于阻塞模式。

更多信息请查阅关于套接字超时的说明。

3.7 版更變: 本方法不再修改*socket.type* 属性的*SOCK_NONBLOCK* 标志。

socket.**setsockopt**(*level*, *optname*, *value: int*)

socket.**setsockopt**(*level*, *optname*, *value: buffer*)

socket.**setsockopt**(*level*, *optname*, *None*, *optlen: int*)

设置给定套接字选项的值（参阅 Unix 手册页 *setsockopt(2)*）。所需的符号常量（SO_* 等）已定义在本*socket* 模块中。该值可以是整数、None 或表示缓冲区的字节类对象。在后一种情况下，由调用者确保字节串中包含正确的数据位（关于将 C 结构体编码为字节串的方法，请参阅可选的内置模块*struct*）。当 *value* 设置为 None 时，必须设置 *optlen* 参数。这相当于调用 setsockopt() C 函数时使用了 optval=NULL 和 optlen=optlen 参数。

3.5 版更變: 现在支持可写的字节类对象。

3.6 版更變: 添加了 setsockopt(level, optname, None, optlen: int) 调用形式。

socket.**shutdown**(*how*)

关闭一半或全部的连接。如果 *how* 为 SHUT_RD，则后续不再允许接收。如果 *how* 为 SHUT_WR，则后续不再允许发送。如果 *how* 为 SHUT_RDWR，则后续的发送和接收都不允许。

socket.**share**(*process_id*)

复制套接字，并准备将其与目标进程共享。目标进程必须以 *process_id* 形式提供。然后可以利用某种形式的进程间通信，将返回的字节对象传递给目标进程，还可以使用*fromshare()* 在新进程中重新创建套接字。一旦本方法调用完毕，就可以安全地将套接字关闭，因为操作系统已经为目标进程复制了该套接字。

可用性: Windows。

3.3 版新加入.

注意此处没有 read() 或 write() 方法，请使用不带 *flags* 参数的*recv()* 和*send()* 来替代。

套接字对象还具有以下（只读）属性，这些属性与传入*socket* 构造函数的值相对应。

socket.**family**

套接字的协议簇。

socket.**type**

套接字的类型。

socket.**proto**

套接字的协议。

18.2.4 关于套接字超时的说明

一个套接字对象可以处于以下三种模式之一：阻塞、非阻塞或超时。套接字默认以阻塞模式创建，但是可以调用*setdefaulttimeout()* 来更改。

- 在 *blocking mode*（阻塞模式）中，操作将阻塞，直到操作完成或系统返回错误（如连接超时）。
- 在 *non-blocking mode*（非阻塞模式）中，如果操作无法立即完成，则操作将失败（不幸的是，不同系统返回的错误不同）：位于*select* 中的函数可用于了解套接字何时以及是否可以读取或写入。
- 在 *timeout mode*（超时模式）下，如果无法在指定的超时内完成操作（抛出*timeout* 异常），或如果系统返回错误，则操作将失败。

備𝔽: 在操作系统层面上，超时模式下的套接字在内部都设置为非阻塞模式。同时，阻塞和超时模式在文件描述符和套接字对象之间共享，这些描述符和对象均应指向同一个网络端点。如果，比如你决定使用套接字的*fileno()*，这一实现细节可能导致明显的结果。

超时与 connect 方法

connect() 操作也受超时设置的约束，通常建议在调用*connect()* 之前调用*settimeout()*，或将超时参数直接传递给*create_connection()*。但是，无论 Python 套接字超时设置如何，系统网络栈都有可能返回自带的连接超时错误。

超时与 accept 方法

如果*getdefaulttimeout()* 的值不是*None*，则*accept()* 方法返回的套接字将继承该超时值。若是 None，返回的套接字行为取决于侦听套接字的设置：

- 如果侦听套接字处于 *阻塞模式*或 *超时模式*，则*accept()* 返回的套接字处于 *阻塞模式*；
- 如果侦听套接字处于 *非阻塞模式*，那么*accept()* 返回的套接字是阻塞还是非阻塞取决于操作系统。如果要确保跨平台时的正确行为，建议手动覆盖此设置。

18.2.5 示例

以下是 4 个使用 TCP/IP 协议的最小示例程序：一台服务器，它将收到的所有数据原样返回（仅服务于一个客户端），还有一个使用该服务器的客户端。注意，服务器必须按序执行 socket()，*bind()*，*listen()*，*accept()* （可能需要重复执行*accept()* 以服务多个客户端），而客户端仅需要按序执行 socket()，*connect()*。还须注意，服务器不在侦听套接字上发送*sendall()*/*recv()*，而是在*accept()* 返回的新套接字上发送。

前两个示例仅支持 IPv4。

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data: break
            conn.sendall(data)
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

下两个例子与上两个很像，但是同时支持 IPv4 和 IPv6。服务端将监听第一个可用的地址族（它本应同时监听两个）。在大多数支持 IPv6 的系统上，IPv6 将有优先权并且服务端可能不会接受 IPv4 流量。客户端将尝试连接到作为名称解析结果被返回的所有地址，并将流量发送给连接成功的第一个地址。

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
with conn:
    print('Connected by', addr)
    while True:
        data = conn.recv(1024)
        if not data: break
        conn.send(data)
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
```

(下页继续)

(繼續上一頁)

```
    sys.exit(1)
with s:
    s.sendall(b'Hello, world')
    data = s.recv(1024)
print('Received', repr(data))
```

下面的例子演示了如何在 Windows 上使用原始套接字编写一个非常简单的网络嗅探器。这个例子需要管理员权限来修改接口:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

下面的例子演示了如何使用 socket 接口与采用原始套接字协议的 CAN 网络进行通信。要改为通过广播管理器协议来使用 CAN，则要用以下方式打开一个 socket:

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

在绑定 (CAN_RAW) 或连接 (*CAN_BCM*) socket 之后，你将可以在 socket 对象上正常使用*socket.send()* 以及*socket.recv()* 操作（及同类操作）。

最后一个例子可能需要特别的权限:

```
import socket
import struct


# CAN frame packing/unpacking (see 'struct can_frame' in <linux/can.h>)

can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])
```

(下页继续)

(繼續上一頁)

```
# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW)
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_can_frame(cf))

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

多次运行一个示例，且每次执行之间等待时间过短，可能导致这个错误:

```
OSError: [Errno 98] Address already in use
```

这是因为前一次运行使套接字处于 TIME_WAIT 状态，无法立即重用。

要防止这种情况，需要设置一个*socket* 标志 socket.SO_REUSEADDR:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

SO_REUSEADDR 标志告诉内核将处于 TIME_WAIT 状态的本地套接字重新使用，而不必等到固有的超时到期。

也参考:

关于套接字编程（C 语言）的介绍，请参阅以下文章:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*，作者 Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*，作者 Samuel J. Leffler et al,

两篇文章都在 UNIX 开发者手册，补充文档 1（第 PS1:7 和 PS1:8 节）中。那些特定于平台的参考资料，它们包含与套接字有关的各种系统调用，也是套接字语义细节的宝贵信息来源。对于 Unix，请参考手册页。对于 Windows，请参阅 WinSock（或 Winsock 2）规范。如果需要支持 IPv6 的 API，读者可能希望参考 **RFC 3493**，标题为 Basic Socket Interface Extensions for IPv6。

18.3 ssl --- 套接字对象的 TLS/SSL 包装器

源代码: Lib/ssl.py

This module provides access to Transport Layer Security (often known as "Secure Sockets Layer") encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, macOS, and probably additional platforms, as long as OpenSSL is installed on that platform.

備🄵: 某些行为可能具有平台依赖，因为调用是根据操作系统的嵌套字 API。不同版本的 Open SSL 也会引起差异：例如 Open SSL 版本 1.0.1 自带 TLSv1.1 和 TLSv1.2

> **警告:** 在阅读安全考量 前不要使用此模块。这样做可能会导致虚假的安全感，因为 ssl 模块的默认设置不一定适合你的应用程序。

本文档记录"ssl" 模块的对象和函数；更多关于 TLS,SSL, 和证书的信息，请参阅下方的 "详情" 选项

本模块提供了一个类*ssl.SSLSocket*，它派生自*socket.socket* 类型，并提供类似套接字的包装器，也能够对通过带 SSL 套接字的数据进行加密和解密。它支持一些额外方法例如 `getpeercert()`，该方法可从连接的另一端获取证书，还有 `cipher()`，该方法可获取安全连接所使用的密码。

对于更复杂的应用程序，*ssl.SSLContext* 类有助于管理设置项和证书，进而可以被使用*SSLContext.wrap_socket()* 方法创建的 SSL 套接字继承。

3.5.3 版更變: 更新以支持和 OpenSSL 1.1.0 的链接

3.6 版更變: OpenSSL 0.9.8、1.0.0 和 1.0.1 已过时，将不再被支持。在 ssl 模块未来的版本中，最低需要 OpenSSL 1.0.2 或 1.1.0。

18.3.1 方法、常量和异常处理

套接字创建

从 Python 3.2 和 2.7.9 开始，建议使用*SSLContext* 实例的*SSLContext.wrap_socket()* 来将套接字包装为*SSLSocket* 对象。辅助函数*create_default_context()* 会返回一个新的带有安全默认设置的上下文。旧的*wrap_socket()* 函数已被弃用，因为它效率较差并且不支持服务器名称提示（SNI）和主机匹配。

客户端套接字实例，采用默认上下文和 IPv4/IPv6 双栈:

```
import socket
import ssl

hostname = 'www.python.org'
context = ssl.create_default_context()

with socket.create_connection((hostname, 443)) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

客户端套接字示例，带有自定义上下文和 IPv4:

```
hostname = 'www.python.org'
# PROTOCOL_TLS_CLIENT requires valid cert chain and hostname
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('path/to/cabundle.pem')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

服务器套接字实例，在 localhost 上监听 IPv4:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain('/path/to/certchain.pem', '/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
    sock.bind(('127.0.0.1', 8443))
    sock.listen(5)
    with context.wrap_socket(sock, server_side=True) as ssock:
        conn, addr = ssock.accept()
        ...
```

上下文创建

便捷函数，可以帮助创建*SSLContext* 对象，用于常见的目的。

ssl.**create_default_context**(*purpose=Purpose.SERVER_AUTH*, *cafile=None*, *capath=None*, *cadata=None*)

返回一个新的*SSLContext* 对象，使用给定 *purpose* 的默认设置。该设置由*ssl* 模块选择，并且通常是代表一个比直接调用*SSLContext* 构造器时更高的安全等级。

cafile, *capath*, *cadata* 代表用于进行证书核验的可选受信任 CA 证书，与*SSLContext.load_verify_locations()* 的一致。如果三个参数均为*None*，此函数可以转而选择信任系统的默认 CA 证书。

设置为: *PROTOCOL_TLS*, *OP_NO_SSLv2* 和*OP_NO_SSLv3*，带有不含 RC4 及未认证的高强度加密密码套件。传入*SERVER_AUTH* 作为 *purpose* 会将*verify_mode* 设为*CERT_REQUIRED* 并加载 CA 证书 (若给出 *cafile*, *capath* 或 *cadata* 之一) 或用*SSLContext.load_default_certs()* 加载默认 CA 证书。

当*keylog_filename* 受支持并且设置了环境变量 SSLKEYLOGFILE 时，*create_default_context()* 会启用密钥日志记录。

備🄴: 协议、选项、密码和其他设置可随时更改为更具约束性的值而无须事先弃用。这些值代表了兼容性和安全性之间的合理平衡。

如果你的应用需要特定的设置，你应当创建一个*SSLContext* 并自行应用设置。

備🄴: 如果你发现当某些较旧的客户端或服务器尝试与用此函数创建的*SSLContext* 进行连接时收到了报错提示"Protocol or cipher suite mismatch"，这可能是因为它们只支持 SSL3.0 而它被此函数用*OP_NO_SSLv3* 排除掉了。SSL3.0 被广泛认为 完全不可用。如果你仍希望继续使用此函数但仍允许 SSL 3.0 连接，你可以使用以下代码重新启用它们:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options &= ~ssl.OP_NO_SSLv3
```

3.4 版新加入.

3.4.4 版更變: RC4 被从默认密码字符串中丢弃。

3.6 版更變: ChaCha20/Poly1305 被添加到默认密码字符串中。

3DES 被从默认密码字符串中丢弃。

3.8 版更變: 增加了对密钥日志记录至 `SSLKEYLOGFILE` 的支持。

异常

exception ssl.**SSLError**

引发此异常以提示来自下层 SSL 实现（目前由 OpenSSL 库提供）的错误。它表示在下层网络连接之上叠加的高层级加密和验证层存在某种问题。此错误是*OSError* 的一个子类型。*SSLError* 实例的错误和消息是由 OpenSSL 库提供的。

3.3 版更變: *SSLError* 曾经是*socket.error* 的一个子类型。

library

一个字符串形式的助记符，用来指明发生错误的 OpenSSL 子模块，例如 `SSL`, `PEM` 或 `X509`。可能的取值范围依赖于 OpenSSL 的版本。

3.3 版新加入.

reason

一个字符串形式的助记符，用来指明发生错误的原因，例如 `CERTIFICATE_VERIFY_FAILED`。可能的取值范围依赖于 OpenSSL 的版本。

3.3 版新加入.

exception ssl.**SSLZeroReturnError**

SSLError 的子类，当尝试读取或写入且 SSL 连接已被完全关闭时会被引发。请注意这并不意味着下层的传输（读取 TCP）已被关闭。

3.3 版新加入.

exception ssl.**SSLWantReadError**

SSLError 的子类，当尝试读取或写入数据，但在请求被满足之前还需要在下层的 TCP 传输上接收更多数据时会被非阻塞型 *SSL* 套接字 引发。

3.3 版新加入.

exception ssl.**SSLWantWriteError**

SSLError 的子类，当尝试读取或写入数据，但在请求被满足之前还需要在下层的 TCP 传输上发送更多数据时会被非阻塞型 *SSL* 套接字 引发。

3.3 版新加入.

exception ssl.**SSLSyscallError**

SSLError 的子类，当尝试在 SSL 套接字上执行操作时遇到系统错误时会被引发。不幸的是，没有简单的方式能检查原始 errno 编号。

3.3 版新加入.

exception ssl.**SSLEOFError**

SSLError 的子类，当 SSL 连接被突然终止时会被引发。通常，当遇到此错误时你不应再尝试重用下层的传输。

3.3 版新加入.

exception ssl.**SSLCertVerificationError**
SSLError 的子类，当证书验证失败时会被引发。

3.7 版新加入.

verify_code
一个数字形式的错误编号，用于表示验证错误。

verify_message
用于表示验证错误的人类可读的字符串。

exception ssl.**CertificateError**
SSLCertVerificationError 的别名。

3.7 版更變: 此异常现在是*SSLCertVerificationError* 的别名。

随机生成

ssl.**RAND_bytes**(*num*)
返回 *num* 个高加密强度伪随机字节数据。如果 PRNG 未使用足够的数据作为随机种子或者如果当前 RAND 方法不支持该操作则会引发*SSLError*。*RAND_status()* 可被用来检查 PRNG 的状态而*RAND_add()* 可被用来为 PRNG 设置随机种子。

对于几乎所有应用程序都更推荐使用*os.urandom()*。

请阅读维基百科文章 Cryptographically secure pseudorandom number generator (CSPRNG) 以了解对于高加密强度生成器的具体要求。

3.3 版新加入.

ssl.**RAND_pseudo_bytes**(*num*)
返回 (bytes, is_cryptographic): bytes 是 *num* 个伪随机字节数据，如果所生成的字节数据为高加密强度则 is_cryptographic 为 `True`。如果当前 RAND 方法不支持此操作则会引发*SSLError*。

所生成的伪随机字节序列如果具有足够的长度则将会具有唯一性，并是并非不可预测。它们可被用于非加密目的以及加密协议中的特定目的，但通常不可被用于密钥生成等目的。

对于几乎所有应用程序都更推荐使用*os.urandom()*。

3.3 版新加入.

3.6 版後已🄵用: OpenSSL 已弃用了*ssl.RAND_pseudo_bytes()*，请改用*ssl.RAND_bytes()*。

ssl.**RAND_status**()
如果 SSL 伪随机数生成器已使用‘足够的’随机性作为种子则返回 `True`，否则返回 `False`。你可以使用*ssl.RAND_egd()* 和*ssl.RAND_add()* 来增加伪随机数生成器的随机性。

ssl.**RAND_egd**(*path*)
如果你在某处运行了一个熵收集守护程序（EGD），且 *path* 是向其打开的套接字连接路径名，此函数将从该套接字读取 256 个字节的随机性数据，并将其添加到 SSL 伪随机数生成器以增加所生成密钥的安全性。此操作通常只在没有更好随机性源的系统上才是必要的。

请查看 http://egd.sourceforge.net/ 或 http://prngd.sourceforge.net/ 来了解有关熵收集守护程序源的信息。

可用性: 对于 LibreSSL 和 OpenSSL > 1.1.0 不可用。

ssl.**RAND_add**(*bytes*, *entropy*)
将给定的 *bytes* 混合到 SSL 伪随机数生成器中。形参 *entropy* (float 类型) 是数据所包含的熵的下界 (因此你可以总是使用 `0.0`)。请查看 **RFC 1750** 了解有关熵源的更多信息。

3.5 版更變: 现在接受可写的字节类对象。

证书处理

ssl.**match_hostname**(*cert*, *hostname*)
验证 *cert* (使用*SSLSocket.getpeercert()* 所返回的已解码格式) 是否匹配给定的 *hostname*。所应用的规则是在 **RFC 2818**, **RFC 5280** 和 **RFC 6125** 中描述的检查 HTTPS 服务器身份的规则。除了 HTTPS，此函数还应当适用于各种基于 SSL 协议的服务器身份检查操作，例如 FTPS, IMAPS, POPS 等等。

失败时引发*CertificateError*。成功时此函数无返回值:

```
>>> cert = {'subject': ((('commonName', 'example.com'),),)}
>>> ssl.match_hostname(cert, "example.com")
>>> ssl.match_hostname(cert, "example.org")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/py3k/Lib/ssl.py", line 130, in match_hostname
ssl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

3.2 版新加入.

3.3.3 版更變: 此函数现在遵循 **RFC 6125**, 6.4.3 小节，它不会匹配多个通配符 (例如 `*.*.com` 或 `*a*.example.org`) 也不匹配国际化域名 (IDN) 片段内部的通配符。IDN A 标签例如 `www*.xn--pthon-kva.org` 仍然受支持，但 `x*.python.org` 不再能匹配 `xn--tda.python.org`。

3.5 版更變: 现在支持匹配存在于证书的 subjectAltName 字段中的 IP 地址。

3.7 版更變: 此函数不再被用于 TLS 连接。主机匹配现在是由 OpenSSL 执行的。

允许位于段的最左端且为唯一字符的通配符。部分通配符例如 `www*.example.com` 已不再受支持。

3.7 版後已F用.

ssl.**cert_time_to_seconds**(*cert_time*)
返回距离 Unix 纪元零时的秒数，给定的 `cert_time` 字符串代表来自证书的”notBefore” 或”notAfter” 日期值，采用 `"%b %d %H:%M:%S %Y %Z"` strptime 格式（C 区域）。

以下为示例代码:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")
>>> timestamp
1515144883
>>> from datetime import datetime
>>> print(datetime.utcfromtimestamp(timestamp))
2018-01-05 09:34:43
```

”notBefore” 或”notAfter” 日期值必须使用 GMT (**RFC 5280**)。

3.5 版更變: 将输入时间解读为 UTC 时间，基于输入字符串中指明的’GMT’ 时区。在之前使用的是本地时区。返回一个整数（不带输入格式中秒的分数部分）

ssl.**get_server_certificate**(*addr*, *ssl_version=PROTOCOL_TLS*, *ca_certs=None*)
带 SSL 保护的服务器的地址 `addr` 以 (*hostname*, *port-number*) 对的形式给出，获取服务器的证书，并将其以 PEM 编码字符串的形式返回。如果指定了 `ssl_version`，则使用该版本的 SSL 协议尝试连接服务器。如果指定了 `ca_certs`，它应当是一个包含根证书列表的文件，使用与*SSLContext.wrap_socket()* 中同名形参一致的格式。该调用将尝试根据指定的根证书集来验证服务器证书，如果验证失败则该调用也将失败。

3.3 版更變: 此函数现在是 IPv6 兼容的。-compatible.

3.5 版更變: 默认的 *ssl_version* 从*PROTOCOL_SSLv3* 改为*PROTOCOL_TLS* 以保证与现代服务器的最大兼容性。

ssl.**DER_cert_to_PEM_cert**(*DER_cert_bytes*)
根据给定的 DER 编码字节块形式的证书，返回同一证书的 PEM 编码字符串版本。

ssl.**PEM_cert_to_DER_cert**(*PEM_cert_string*)
根据给定的 ASCII PEM 字符串形式的证书，返回同一证书的 DER 编码字节序列。

ssl.**get_default_verify_paths**()
返回包含 OpenSSL 的默认 cafile 和 capath 的路径的命名元组。此路径与*SSLContext.set_default_verify_paths()* 所使用的相同。返回值是一个*named tuple* `DefaultVerifyPaths`:

- `cafile` - 解析出的 cafile 路径或者如果文件不存在则为 `None`,
- `capath` - 解析出的 capath 路径或者如果目录不存在则为 `None`,
- `openssl_cafile_env` - 指向一个 cafile 的 OpenSSL 环境键,
- `openssl_cafile` - 一个 cafile 的硬编码路径,
- `openssl_capath_env` - 指向一个 capath 的 OpenSSL 环境键,
- `openssl_capath` - 一个 capath 目录的硬编码路径

可用性: LibreSSL 会忽略环境变量 `openssl_cafile_env` 和 `openssl_capath_env`。

3.4 版新加入.

ssl.**enum_certificates**(*store_name*)
从 Windows 的系统证书库中检索证书。*store_name* 可以是 `CA`, `ROOT` 或 `MY` 中的一个。Windows 也可能会提供额外的证书库。

此函数返回一个包含 (cert_bytes, encoding_type, trust) 元组的列表。encoding_type 指明 cert_bytes 的编码格式。它可以为 `x509_asn` 以表示 X.509 ASN.1 数据或是 `pkcs_7_asn` 以表示 PKCS#7 ASN.1 数据。trust 以 OIDS 集合的形式指明证书的目的，或者如果证书对于所有目的都可以信任则为 `True`。

示例:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

可用性: Windows。

3.4 版新加入.

ssl.**enum_crls**(*store_name*)
Windows 的系统证书库中检索 CRL。*store_name* 可以是 `CA`, `ROOT` 或 `MY` 中的一个。Windows 也可能会提供额外的证书库。

此函数返回一个包含 (cert_bytes, encoding_type, trust) 元组的列表。encoding_type 指明 cert_bytes 的编码格式。它可以为 `x509_asn` 以表示 X.509 ASN.1 数据或是 `pkcs_7_asn` 以表示 PKCS#7 ASN.1 数据。

可用性: Windows。

3.4 版新加入.

ssl.**wrap_socket**(*sock*, *keyfile=None*, *certfile=None*, *server_side=False*, *cert_reqs=CERT_NONE*, *ssl_version=PROTOCOL_TLS*, *ca_certs=None*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *ciphers=None*)
接受一个*socket.socket* 的实例 `sock`，并返回一个*ssl.SSLSocket* 的实例，该类型是*socket.socket* 的子类型，它将下层的套接字包装在一个 SSL 上下文中。`sock` 必须是一个*SOCK_STREAM* 套接字；其他套接字类型不被支持。

在内部，该函数会创建一个*SSLContext*，其协议版本为 *ssl_version* 且*SSLContext.options* 设为 *cert_reqs*。如果设置了 *keyfile*, *certfile*, *ca_certs* 或 *ciphers* 等形参，则参数值

会被传给*SSLContext.load_cert_chain()*，*SSLContext.load_verify_locations()* 以及*SSLContext.set_ciphers()*。

参数 *server_side*，*do_handshake_on_connect* 和 *suppress_ragged_eofs* 具有与*SSLContext.wrap_socket()* 相同的含义。

3.7 版後已F用: 从 Python 3.2 和 2.7.9 开始，建议使用*SSLContext.wrap_socket()* 来代替*wrap_socket()*。模块级函数的功能受限并且将创建不安全的客户端套接字，不带服务器名称提示或主机名匹配。

常量

所有常量现在都是*enum.IntEnum* 或*enum.IntFlag* 多项集的成员。

3.6 版新加入.

ssl.**CERT_NONE**

SSLContext.verify_mode 或*wrap_socket()* 的 cert_reqs 形参可能的取值。*PROTOCOL_TLS_CLIENT* 除外，这是默认的模式。对于客户端套接字，几乎任何证书都是可接受的。验证错误例如不受信任或过期的证书错误会被忽略并且不会中止 TLS/SSL 握手。

在服务器模式下，不会从客户端请求任何证书，因此客户端不会发送任何用于客户端证书身份验证的证书。

参见下文对于*安全考量* 的讨论。

ssl.**CERT_OPTIONAL**

SSLContext.verify_mode 或*wrap_socket()* 的 cert_reqs 形参可能的取值。*CERT_OPTIONAL* 具有与*CERT_REQUIRED* 相同的含义。对于客户端套接字推荐改用*CERT_REQUIRED*。

在服务器模式下，客户端证书请求会被发送给客户端。客户端可以忽略请求也可以发送一个证书以执行 TLS 客户端证书身份验证。如果客户端选择发送证书，则将对其执行验证。任何验证错误都将立即中止 TLS 握手。

使用此设置要求将一组有效的 CA 证书传递给*SSLContext.load_verify_locations()* 或是作为*wrap_socket()* 的 ca_certs 形参值。

ssl.**CERT_REQUIRED**

SSLContext.verify_mode 或*wrap_socket()* 的 cert_reqs 形参可能的取值。在此模式下，需要从套接字连接的另一端获取证书；如果未提供证书或验证失败则将引发*SSLError*。此模式 **不能**在客户端模式下对证书进行验证，因为它不会匹配主机名。*check_hostname* 也必须被启用以验证证书的真实性。*PROTOCOL_TLS_CLIENT* 会使用*CERT_REQUIRED* 并默认启用*check_hostname*。

对于服务器套接字，此模式会提供强制性的 TLS 客户端证书验证。客户端证书请求会被发送给客户端并且客户端必须提供有效且受信任的证书。

使用此设置要求将一组有效的 CA 证书传递给*SSLContext.load_verify_locations()* 或是作为*wrap_socket()* 的 ca_certs 形参值。

class ssl.**VerifyMode**

CERT_* 常量的*enum.IntEnum* 多项集。

3.6 版新加入.

ssl.**VERIFY_DEFAULT**

SSLContext.verify_flags 可能的取值。在此模式下，证书吊销列表（CRL）并不会被检查。OpenSSL 默认不要求也不验证 CRL。

3.4 版新加入.

ssl.**VERIFY_CRL_CHECK_LEAF**
SSLContext.verify_flags 可能的取值。在此模式下，只会检查对等证书而不检查任何中间 CA 证书。此模式要求提供由对等证书颁发者（其直接上级 CA）签名的有效 CRL。如果未使用*SSLContext.load_verify_locations* 加载正确的 CRL，则验证将失败。

3.4 版新加入.

ssl.**VERIFY_CRL_CHECK_CHAIN**
SSLContext.verify_flags 可能的取值。在此模式下，会检查对等证书链中所有证书的 CRL。

3.4 版新加入.

ssl.**VERIFY_X509_STRICT**
SSLContext.verify_flags 可能的取值，用于禁用已损坏 X.509 证书的绕过操作。

3.4 版新加入.

ssl.**VERIFY_X509_TRUSTED_FIRST**
SSLContext.verify_flags 可能的取值。它指示 OpenSSL 在构建用于验证某个证书的信任链时首选受信任的证书。此旗标将默认被启用。

3.4.4 版新加入.

class ssl.**VerifyFlags**
VERIFY_* 常量的*enum.IntFlag* 多项集。

3.6 版新加入.

ssl.**PROTOCOL_TLS**
选择客户端和服务器均支持的最高协议版本。此选项名称并不准确，实际上"SSL" 和"TLS" 协议均可被选择。

3.6 版新加入.

ssl.**PROTOCOL_TLS_CLIENT**
像*PROTOCOL_TLS* 一样地自动协商最高协议版本，但是只支持客户端*SSLSocket* 连接。此协议默认会启用*CERT_REQUIRED* 和*check_hostname*。

3.6 版新加入.

ssl.**PROTOCOL_TLS_SERVER**
像*PROTOCOL_TLS* 一样地自动协商最高协议版本，但是只支持服务器*SSLSocket* 连接。

3.6 版新加入.

ssl.**PROTOCOL_SSLv23**
PROTOCOL_TLS 的别名。

3.6 版後已🄵用: 请改用*PROTOCOL_TLS*。

ssl.**PROTOCOL_SSLv2**
选择 SSL 版本 2 作为通道加密协议。

如果 OpenSSL 编译时使用了 OPENSSL_NO_SSL2 旗标则此协议将不可用。

> **警告:** SSL 版本 2 并不安全。极不建议使用它。

3.6 版後已🄵用: OpenSSL 已经移除了对 SSLv2 的支持。

ssl.**PROTOCOL_SSLv3**
选择 SSL 版本 3 作为通道加密协议。

如果 OpenSSL 编译时使用了 OPENSSL_NO_SSLv3 旗标则此协议将不可用。

> **警告：** SSL 版本 3 并不安全。极不建议使用它。

3.6 版後已🄵用: OpenSSL 已经弃用了所有带有特定版本号的协议。请改用默认协议*PROTOCOL_TLS* 并附带*OP_NO_SSLv3* 等旗标。

ssl.**PROTOCOL_TLSv1**
选择 TLS 版本 1.0 作为通道加密协议。

3.6 版後已🄵用: OpenSSL 已经弃用了所有带有特定版本号的协议。请改用默认协议*PROTOCOL_TLS* 并附带*OP_NO_SSLv3* 等旗标。

ssl.**PROTOCOL_TLSv1_1**
选择 TLS 版本 1.1 作为通道加密协议。仅适用于 openssl 版本 1.0.1+。

3.4 版新加入.

3.6 版後已🄵用: OpenSSL 已经弃用了所有带有特定版本号的协议。请改用默认协议*PROTOCOL_TLS* 并附带*OP_NO_SSLv3* 等旗标。

ssl.**PROTOCOL_TLSv1_2**
选译 TLS 版本 1.2 作为通道加密协议。这是最新的版本，也应是能提供最大保护的最佳选择，如果通信双方都支持它的话。仅适用于 openssl 版本 1.0.1+。

3.4 版新加入.

3.6 版後已🄵用: OpenSSL 已经弃用了所有带有特定版本号的协议。请改用默认协议*PROTOCOL_TLS* 并附带*OP_NO_SSLv3* 等旗标。

ssl.**OP_ALL**
对存在于其他 SSL 实现中的各种缺陷启用绕过操作。默认会设置此选项。没有必要设置与 OpenSSL 的 SSL_OP_ALL 常量同名的旗标。

3.2 版新加入.

ssl.**OP_NO_SSLv2**
阻止 SSLv2 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 SSLv2 作为协议版本。

3.2 版新加入.

3.6 版後已🄵用: SSLv2 已被弃用

ssl.**OP_NO_SSLv3**
阻止 SSLv3 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 SSLv3 作为协议版本。

3.2 版新加入.

3.6 版後已🄵用: SSLv3 已被弃用

ssl.**OP_NO_TLSv1**
阻止 TLSv1 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 TLSv1 作为协议版本。

3.2 版新加入.

3.7 版後已🄵用: 此选项自 OpenSSL 1.1.0 起已被弃用，请改用新的*SSLContext.minimum_version* 和*SSLContext.maximum_version*。

ssl.**OP_NO_TLSv1_1**
阻止 TLSv1.1 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 TLSv1.1 作为协议版本。仅适用于 openssl 版本 1.0.1+。

3.4 版新加入.

3.7 版後已𝔽用: 此选项自 OpenSSL 1.1.0 起已被弃用。

ssl.**OP_NO_TLSv1_2**
阻止 TLSv1.2 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 TLSv1.2 作为协议版本。仅适用于 openssl 版本 1.0.1+。

3.4 版新加入.

3.7 版後已𝔽用: 此选项自 OpenSSL 1.1.0 起已被弃用。

ssl.**OP_NO_TLSv1_3**
阻止 TLSv1.3 连接。此选项仅可与*PROTOCOL_TLS* 结合使用。它会阻止对等方选择 TLSv1.3 作为协议版本。TLS 1.3 适用于 OpenSSL 1.1.1 或更新的版本。当 Python 编译是基于较旧版本的 OpenSSL 时，该旗标默认为 *0*。

3.7 版新加入.

3.7 版後已𝔽用: 此选项自 OpenSSL 1.1.0 起已被弃用。它被添加到 2.7.15, 3.6.3 和 3.7.0 是为了向下兼容 OpenSSL 1.0.2。

ssl.**OP_NO_RENEGOTIATION**
禁用所有 TLSv1.2 和更早版本的重协商操作。不发送 HelloRequest 消息，并忽略通过 ClientHello 发起的重协商请求。

此选项仅适用于 OpenSSL 1.1.0h 及更新的版本。

3.7 版新加入.

ssl.**OP_CIPHER_SERVER_PREFERENCE**
使用服务器的密码顺序首选项，而不是客户端的首选项。此选项在客户端套接字和 SSLv2 服务器套接字上无效。

3.3 版新加入.

ssl.**OP_SINGLE_DH_USE**
阻止对于单独的 SSL 会话重用相同的 DH 密钥。这会提升前向保密性但需要更多的计算资源。此选项仅适用于服务器套接字。

3.3 版新加入.

ssl.**OP_SINGLE_ECDH_USE**
阻止对于单独的 SSL 会话重用相同的 ECDH 密钥。这会提升前向保密性但需要更多的计算资源。此选项仅适用于服务器套接字。

3.3 版新加入.

ssl.**OP_ENABLE_MIDDLEBOX_COMPAT**
在 TLS 1.3 握手中发送虚拟更改密码规格（CCS）消息以使得 TLS 1.3 连接看起来更像是 TLS 1.2 连接。

此选项仅适用于 OpenSSL 1.1.1 及更新的版本。

3.8 版新加入.

ssl.**OP_NO_COMPRESSION**
在 SSL 通道上禁用压缩。这适用于应用协议支持自己的压缩方案的情况。

此选项仅适用于 OpenSSL 1.0.0 及更新的版本。

3.3 版新加入.

class ssl.**Options**
OP_* 常量的*enum.IntFlag* 多项集。

ssl.**OP_NO_TICKET**
阻止客户端请求会话凭据。

3.6 版新加入.

ssl.**OP_IGNORE_UNEXPECTED_EOF**
忽略 TLS 连接的意外关闭。

此选项仅适用于 OpenSSL 3.0.0 及更新的版本。

3.10 版新加入.

ssl.**HAS_ALPN**
OpenSSL 库是否具有对 **RFC 7301** 中描述的 *应用层协议协商* TLS 扩展的内置支持。

3.5 版新加入.

ssl.**HAS_NEVER_CHECK_COMMON_NAME**
OpenSSL 库是否具有对不检测目标通用名称的内置支持且*SSLContext.hostname_checks_common_name* 为可写状态。

3.7 版新加入.

ssl.**HAS_ECDH**
OpenSSL 库是否具有对基于椭圆曲线的 Diffie-Hellman 密钥交换的内置支持。此常量应当为真值，除非发布者明确地禁用了此功能。

3.3 版新加入.

ssl.**HAS_SNI**
OpenSSL 库是否具有对 *服务器名称提示扩展*（在 **RFC 6066** 中定义）的内置支持。

3.2 版新加入.

ssl.**HAS_NPN**
OpenSSL 库是否具有对 *应用层协议协商* 中描述的 *下一协议协商*的内置支持。当此常量为真值时，你可以使用*SSLContext.set_npn_protocols()* 方法来公告你想要支持的协议。

3.3 版新加入.

ssl.**HAS_SSLv2**
OpenSSL 库是否具有对 SSL 2.0 协议的内置支持。

3.7 版新加入.

ssl.**HAS_SSLv3**
OpenSSL 库是否具有对 SSL 3.0 协议的内置支持。

3.7 版新加入.

ssl.**HAS_TLSv1**
OpenSSL 库是否具有对 TLS 1.0 协议的内置支持。

3.7 版新加入.

ssl.**HAS_TLSv1_1**
OpenSSL 库是否具有对 TLS 1.1 协议的内置支持。

3.7 版新加入.

ssl.**HAS_TLSv1_2**
OpenSSL 库是否具有对 TLS 1.2 协议的内置支持。

3.7 版新加入.

ssl.**HAS_TLSv1_3**
OpenSSL 库是否具有对 TLS 1.3 协议的内置支持。

3.7 版新加入.

ssl.**CHANNEL_BINDING_TYPES**
受支持的 TLS 通道绑定类型组成的列表。此列表中的字符串可被用作传给*SSLSocket.get_channel_binding()* 的参数。

3.3 版新加入.

ssl.**OPENSSL_VERSION**
解释器所加载的 OpenSSL 库的版本字符串:

```
>>> ssl.OPENSSL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

3.2 版新加入.

ssl.**OPENSSL_VERSION_INFO**
代表 OpenSSL 库的版本信息的五个整数所组成的元组:

```
>>> ssl.OPENSSL_VERSION_INFO
(1, 0, 2, 11, 15)
```

3.2 版新加入.

ssl.**OPENSSL_VERSION_NUMBER**
OpenSSL 库的原始版本号，以单个整数表示:

```
>>> ssl.OPENSSL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSSL_VERSION_NUMBER)
'0x100020bf'
```

3.2 版新加入.

ssl.**ALERT_DESCRIPTION_HANDSHAKE_FAILURE**
ssl.**ALERT_DESCRIPTION_INTERNAL_ERROR**
ALERT_DESCRIPTION_*
来自 **RFC 5246** 等文档的警报描述。IANA TLS Alert Registry 中包含了这个列表及对定义其含义的 RFC 引用。

被用作*SSLContext.set_servername_callback()* 中的回调函数的返回值。

3.4 版新加入.

class ssl.**AlertDescription**
ALERT_DESCRIPTION_* 常量的*enum.IntEnum* 多项集。

3.6 版新加入.

Purpose.**SERVER_AUTH**
create_default_context() 和*SSLContext.load_default_certs()* 的选项值。这个值表明此上下文可以被用来验证 Web 服务器（因此，它将被用来创建客户端套接字）。

3.4 版新加入.

Purpose.**CLIENT_AUTH**
create_default_context() 和*SSLContext.load_default_certs()* 的选项值。这个值表明此上下文可以被用来验证 Web 客户端（因此，它将被用来创建服务器端套接字）。

3.4 版新加入.

class ssl.**SSLErrorNumber**
SSL_ERROR_* 常量的*enum.IntEnum* 多项集。

3.6 版新加入.

class ssl.**TLSVersion**
SSLContext.maximum_version 和*SSLContext.minimum_version* 中的 SSL 和 TLS 版本的*enum.IntEnum* 多项集。

3.7 版新加入.

TLSVersion.**MINIMUM_SUPPORTED**

TLSVersion.**MAXIMUM_SUPPORTED**
受支持的最低和最高 SSL 或 TLS 版本。这些常量被称为魔术常量。它们的值并不反映可用的最低和最高 TLS/SSL 版本。

TLSVersion.**SSLv3**

TLSVersion.**TLSv1**

TLSVersion.**TLSv1_1**

TLSVersion.**TLSv1_2**

TLSVersion.**TLSv1_3**
SSL 3.0 至 TLS 1.3。

18.3.2 SSL 套接字

class ssl.**SSLSocket**(*socket.socket*)
SSL 套接字提供了套接字对象 的下列方法:

- *accept()*
- *bind()*
- *close()*
- *connect()*
- *detach()*
- *fileno()*
- *getpeername()*, *getsockname()*
- *getsockopt()*, *setsockopt()*
- *gettimeout()*, *settimeout()*, *setblocking()*
- *listen()*
- *makefile()*
- *recv()*, *recv_into()* (but passing a non-zero `flags` argument is not allowed)
- *send()*, *sendall()* (with the same limitation)
- *sendfile()* (but *os.sendfile* will be used for plain-text sockets only, else *send()* will be used)
- *shutdown()*

但是，由于 SSL（和 TLS）协议在 TCP 之上具有自己的框架，因此 SSL 套接字抽象在某些方面可能与常规的 OS 层级套接字存在差异。特别是要查看非阻塞型套接字说明。

SSLSocket 的实例必须使用*SSLContext.wrap_socket()* 方法来创建。

3.5 版更變: 新增了 `sendfile()` 方法。

3.5 版更變: shutdown() 不会在每次接收或发送字节数据后重置套接字超时。现在套接字超时为关闭的最大总持续时间。

3.6 版後已𝔽用: 直接创建*SSLSocket* 实例的做法已被弃用，请使用*SSLContext.wrap_socket()* 来包装套接字。

3.7 版更變: *SSLSocket* 的实例必须使用*wrap_socket()* 来创建。在较早的版本中，直接创建实例是可能的。但这从未被记入文档或是被正式支持。

SSL 套接字还具有下列方法和属性:

SSLSocket.**read**(*len=1024*, *buffer=None*)

从 SSL 套接字读取至多 *len* 个字节的数据并将结果作为 bytes 实例返回。如果指定了 *buffer*，则改为读取到缓冲区，并返回所读取的字节数。

如果套接字为非阻塞型 则会引发*SSLWantReadError* 或*SSLWantWriteError* 且读取将阻塞。

由于在任何时候重新协商都是可能的，因此调用*read()* 也可能导致写入操作。

3.5 版更變: 套接字超时在每次接收或发送字节数据后不会再被重置。现在套接字超时为读取至多 *len* 个字节数据的最大总持续时间。

3.6 版後已𝔽用: 请使用 recv() 来代替*read()*。

SSLSocket.**write**(*buf*)

将 *buf* 写入到 SSL 套接字并返回所写入的字节数。*buf* 参数必须为支持缓冲区接口的对象。

如果套接字为非阻塞型 则会引发*SSLWantReadError* 或*SSLWantWriteError* 且读取将阻塞。

由于在任何时候重新协商都是可能的，因此调用*write()* 也可能导致读取操作。

3.5 版更變: 套接字超时在每次接收或发送字节数据后不会再被重置。现在套接字超时为写入 *buf* 的最大总持续时间。

3.6 版後已𝔽用: 请使用 send() 来代替*write()*。

備𝔽: *read()* 和*write()* 方法是读写未加密的应用级数据，并将其解密/加密为带加密的线路级数据的低层级方法。这些方法需要有激活的 SSL 连接，即握手已完成而*SSLSocket.unwrap()* 尚未被调用。

通常你应当使用套接字 API 方法例如*recv()* 和*send()* 来代替这些方法。

SSLSocket.**do_handshake**()

执行 SSL 设置握手。

3.4 版更變: 当套接字的*context* 的*check_hostname* 属性为真值时此握手方法还会执行*match_hostname()*。

3.5 版更變: 套接字超时在每次接收或发送字节数据时不会再被重置。现在套接字超时为握手的最大总持续时间。

3.7 版更變: 主机名或 IP 地址会在握手期间由 OpenSSL 进行匹配。函数*match_hostname()* 将不再被使用。在 OpenSSL 拒绝主机名和 IP 地址的情况下，握手将提前被中止并向对等方发送 TLS 警告消息。

SSLSocket.**getpeercert**(*binary_form=False*)

如果连接另一端的对等方没有证书，则返回 None。如果 SSL 握手还未完成，则会引发*ValueError*。

如果 binary_form 形参为*False*，并且从对等方接收到了证书，此方法将返回一个*dict* 实例。如果证书未通过验证，则字典将为空。如果证书通过验证，它将返回由多个密钥组成的字典，其中包括 subject (证书颁发给的主体) 和 issuer (颁发证书的主体)。如果证书包含一个 *Subject Alternative Name* 扩展的实例 (see **RFC 3280**)，则字典中还将有一个 subjectAltName 键。

subject 和 issuer 字段都是包含在证书中相应字段的数据结构中给出的相对专有名称（RDN）序列的元组，每个 RDN 均为 name-value 对的序列。这里是一个实际的示例:

```
{'issuer': ((('countryName', 'IL'),),
            (('organizationName', 'StartCom Ltd.'),),
            (('organizationalUnitName',
              'Secure Digital Certificate Signing'),),
            (('commonName',
              'StartCom Class 2 Primary Intermediate Server CA'),)),
 'notAfter': 'Nov 22 08:15:19 2013 GMT',
 'notBefore': 'Nov 21 03:09:52 2011 GMT',
 'serialNumber': '95F0',
 'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
             (('commonName', '*.eff.org'),),
             (('emailAddress', 'hostmaster@eff.org'),)),
 'subjectAltName': (('DNS', '*.eff.org'), ('DNS', 'eff.org')),
 'version': 3}
```

備𝔽: 要验证特定服务的证书，你可以使用*match_hostname()* 函数。

如果 binary_form 形参为*True*，并且提供了证书，此方法会将整个证书的 DER 编码形式作为字节序列返回，或者如果对等方未提供证书则返回*None*。对等方是否提供证书取决于 SSL 套接字的角色:

- 对于客户端 SSL 套接字，服务器将总是提供证书，无论是否需要进行验证;
- 对于服务器 SSL 套接字，客户端将仅在服务器要求时才提供证书；因此如果你使用了*CERT_NONE* (而不是*CERT_OPTIONAL* 或*CERT_REQUIRED*) 则*getpeercert()* 将返回*None*。

3.2 版更變: 返回的字典包括额外的条目例如 issuer 和 notBefore。

3.4 版更變: 如果握手未完成则会引发*ValueError*。返回的字典包括额外的 X509v3 扩展条目例如 crlDistributionPoints, caIssuers 和 OCSP URI。

3.9 版更變: IPv6 地址字符串不再附带末尾换行符。

SSLSocket.**cipher**()
返回由三个值组成的元组，其中包含所使用的密码名称，定义其使用方式的 SSL 协议版本，以及所使用的加密比特位数。如果尚未建立连接，则返回 None。

SSLSocket.**shared_ciphers**()
返回在握手期间由客户端共享的密码列表。所返回列表的每个条目都是由三个值组成的元组，其中包括密码名称，定义其使用方式的 SSL 协议版本，以及密码所使用的加密比特位数。如果尚未建立连接或套接字为客户端套接字则*shared_ciphers()* 将返回 None。

3.5 版新加入.

SSLSocket.**compression**()
以字符串形式返回所使用的压缩算法，或者如果连接没有使用压缩则返回 None。

如果高层级的协议支持自己的压缩机制，你可以使用*OP_NO_COMPRESSION* 来禁用 SSL 层级的压缩。

3.3 版新加入.

SSLSocket.**get_channel_binding**(*cb_type="tls-unique"*)
为当前连接获取字节串形式的通道绑定数据。如果尚未连接或握手尚未完成则返回 None。

cb_type 形参允许选择需要的通道绑定类型。有效的通道绑定类型在*CHANNEL_BINDING_TYPES* 列表中列出。目前只支持由 **RFC 5929** 所定义的'tls-unique' 通道绑定。如果请求了一个不受支持的通道绑定类型则将引发*ValueError*。

3.3 版新加入.

SSLSocket.**selected_alpn_protocol**()
返回在 TLS 握手期间所选择的协议。如果*SSLContext.set_alpn_protocols()* 未被调用，如果另一方不支持 ALPN，如果此套接字不支持任何客户端所用的协议，或者如果握手尚未发生，则将返回 None。

3.5 版新加入.

SSLSocket.**selected_npn_protocol**()
返回在 Return the higher-level protocol that was selected during the TLS/SSL 握手期间所选择的高层级协议。如果*SSLContext.set_npn_protocols()* 未被调用，或者如果另一方不支持 NPN，或者如果握手尚未发生，则将返回 None。

3.3 版新加入.

SSLSocket.**unwrap**()
执行 SSL 关闭握手，这会从下层的套接字中移除 TLS 层，并返回下层的套接字对象。这可被用来通过一个连接将加密操作转为非加密。返回的套接字应当总是被用于同连接另一方的进一步通信，而不是原始的套接字。

SSLSocket.**verify_client_post_handshake**()
向一个 TLS 1.3 客户端请求握手后身份验证(PHA)。只有在初始 TLS 握手之后且双方都启用了 PHA 的情况下才能为服务器端套接字的 TLS 1.3 连接启用 PHA，参见*SSLContext.post_handshake_auth*。

此方法不会立即执行证书交换。服务器端会在下一次写入事件期间发送 CertificateRequest 并期待客户端在下一次读取事件期间附带证书进行响应。

如果有任何前置条件未被满足（例如非 TLS 1.3，PHA 未启用），则会引发*SSLError*。

備 𝔽: 仅在 OpenSSL 1.1.1 且 TLS 1.3 被启用时可用。没有 TLS 1.3 支持，此方法将引发*NotImplementedError*。

3.8 版新加入.

SSLSocket.**version**()
Return the actual SSL protocol version negotiated by the connection as a string, or None if no secure connection is established. As of this writing, possible return values include "SSLv2", "SSLv3", "TLSv1", "TLSv1.1" and "TLSv1.2". Recent OpenSSL versions may define more return values.

3.5 版新加入.

SSLSocket.**pending**()
返回在连接上等待被读取的已解密字节数。

SSLSocket.**context**
此 SSL 套接字所联结的*SSLContext* 对象。如果 SSL 套接字是使用已弃用的*wrap_socket()* 函数(而非*SSLContext.wrap_socket()*) 创建的，则这将是为此 SSL 套接字创建的自定义上下文对象。

3.2 版新加入.

SSLSocket.**server_side**
一个布尔值，对于服务器端套接字为 True 而对于客户端套接字则为 False。

3.2 版新加入.

SSLSocket.**server_hostname**
服务器的主机名: *str* 类型，对于服务器端套接字或者如果构造器中未指定主机名则为 None。

3.2 版新加入.

3.7 版更變: 现在该属性将始终为 ASCII 文本。当 server_hostname 为一个国际化域名（IDN）时，该属性现在会保存为 A 标签形式 ("xn--pythn-mua.org") 而非 U 标签形式 ("pythön.org")。

SSLSocket.**session**
用于 SSL 连接的*SSLSession*。该会话将在执行 TLS 握手后对客户端和服务器端套接字可用。对于客户端套接字该会话可以在调用*do_handshake()* 之前被设置以重用一个会话。

3.6 版新加入.

SSLSocket.**session_reused**
3.6 版新加入.

18.3.3 SSL 上下文

3.2 版新加入.

SSL 上下文可保存各种比单独 SSL 连接寿命更长的数据，例如 SSL 配置选项，证书和私钥等。它还可为服务器端套接字管理缓存，以加快来自相同客户端的重复连接。

class ssl.**SSLContext**(*protocol=PROTOCOL_TLS*)
创建一个新的 SSL 上下文。你可以传入 *protocol*，它必须为此模块中定义的 PROTOCOL_* 常量之一。该形参指定要使用哪个 SSL 协议版本。通常，服务器会选择一个特定的协议版本，而客户端必须适应服务器的选择。大多数版本都不能与其他版本互操作。如果未指定，则默认值为*PROTOCOL_TLS*；它提供了与其他版本的最大兼容性。

这个表显示了客户端（横向）的哪个版本能够连接服务器（纵向）的哪个版本。

客户端 / 服务器	SSLv2	SSLv3	TLS[3]	TLSv1	TLSv1.1	TLSv1.2
SSLv2	是	否	否[1]	否	否	否
SSLv3	否	是	否[2]	否	否	否
TLS (SSLv23)[3]	否[1]	否[2]	是	是	是	是
TLSv1	否	否	是	是	否	否
TLSv1.1	否	否	是	否	是	否
TLSv1.2	否	否	是	否	否	是

脚注

也参考:

create_default_context() 让*ssl* 为特定目标选择安全设置。

3.6 版更變: 上下文会使用安全默认值来创建。默认设置的选项有*OP_NO_COMPRESSION*, *OP_CIPHER_SERVER_PREFERENCE*, *OP_SINGLE_DH_USE*, *OP_SINGLE_ECDH_USE*, *OP_NO_SSLv2* (except for *PROTOCOL_SSLv2*) 和*OP_NO_SSLv3* (except for *PROTOCOL_SSLv3*)。初始密码集列表只包含 HIGH 密码，不包含 NULL 密码和 MD5 密码 (*PROTOCOL_SSLv2* 除外)。

SSLContext 对象具有以下方法和属性:

SSLContext.**cert_store_stats**()
获取以字典表示的有关已加载的 X.509 证书数量，被标记为 CA 证书的 X.509 证书数量以及证书吊销列表的统计信息。

具有一个 CA 证书和一个其他证书的上下文示例:

[3] TLS 1.3 协议在 OpenSSL >= 1.1.1 中设置*PROTOCOL_TLS* 时可用。没有专门针对 TLS 1.3 的 PROTOCOL 常量。
[1] *SSLContext* 默认设置*OP_NO_SSLv2* 以禁用 SSLv2。
[2] *SSLContext* 默认设置*OP_NO_SSLv3* 以禁用 SSLv3。

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

3.4 版新加入.

SSLContext.**load_cert_chain**(*certfile*, *keyfile=None*, *password=None*)

Load a private key and the corresponding certificate. The *certfile* string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate's authenticity. The *keyfile* string, if present, must point to a file containing the private key. Otherwise the private key will be taken from *certfile* as well. See the discussion of 证书 for more information on how the certificate is stored in the *certfile*.

password 参数可以是一个函数，调用时将得到用于解密私钥的密码。它在私钥被加密且需要密码时才会被调用。它调用时将不带任何参数，并且应当返回一个字符串、字节串或字节数组。如果返回值是一个字符串，在用它解密私钥之前它将以 UTF-8 进行编码。或者也可以直接将字符串、字节串或字节数组值作为 *password* 参数提供。如果私钥未被加密且不需要密码则它将被忽略。

如果未指定 *password* 参数且需要一个密码，将会使用 OpenSSL 内置的密码提示机制来交互式地提示用户输入密码。

如果私钥不能匹配证书则会引发*SSLError*。

3.3 版更變: 新增可选参数 *password*。

SSLContext.**load_default_certs**(*purpose=Purpose.SERVER_AUTH*)

Load a set of default "certification authority" (CA) certificates from default locations. On Windows it loads CA certs from the `CA` and `ROOT` system stores. On all systems it calls *SSLContext.set_default_verify_paths()*. In the future the method may load CA certificates from other locations, too.

purpose 旗标指明要加载哪一类 CA 证书。默认设置*Purpose.SERVER_AUTH* 加载被标记且被信任用于 TLS Web 服务器验证（客户端套接字）的证书。*Purpose.CLIENT_AUTH* 则加载用于在服务器端进行客户端证书验证的 CA 证书。

3.4 版新加入.

SSLContext.**load_verify_locations**(*cafile=None*, *capath=None*, *cadata=None*)

当*verify_mode* 不为*CERT_NONE* 时加载一组用于验证其他对等方证书的" 证书颁发机构" (CA) 证书。必须至少指定 *cafile* 或 *capath* 中的一个。

此方法还可加载 PEM 或 DER 格式的证书吊销列表 (CRL)，为此必须正确配置*SSLContext.verify_flags*。

如果存在 *cafile* 字符串，它应为 PEM 格式的级联 CA 证书文件的路径。请参阅证书 中的讨论来了解有关如何处理此文件中的证书的更多信息。

如果存在 *capath* 字符串，它应为包含多个 PEM 格式的 CA 证书的目录的路径，并遵循 OpenSSL 专属布局。

如果存在 *cadata* 对象，它应为一个或多个 PEM 编码的证书的 ASCII 字符串或者 DER 编码的证书的*bytes-like object*。与 *capath* 一样 PEM 编码的证书之外的多余行会被忽略，但至少要有一个证书。

3.4 版更變: 新增可选参数 *cadata*

SSLContext.**get_ca_certs**(*binary_form=False*)

获取已离开法人" 证书颁发机构" (CA) 证书列表。如果 `binary_form` 形参为*False* 则每个列表条目都是一个类似于*SSLSocket.getpeercert()* 输出的字典。在其他情况下此方法将返回一个 DER 编码的证书的列表。返回的列表不包含来自 *capath* 的证书，除非 SSL 连接请求并加载了一个证书。

備𝔽: capath 目录中的证书不会被加载，除非它们已至少被使用过一次。

3.4 版新加入.

SSLContext.**get_ciphers**()

获取已启用密码的列表。该列表将按密码的优先级排序。参见*SSLContext.set_ciphers()*。

示例:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()  # OpenSSL 1.0.x
[{'alg_bits': 256,
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'id': 50380848,
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 256},
 {'alg_bits': 128,
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'id': 50380847,
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1/SSLv3',
  'strength_bits': 128}]
```

在 OpenSSL 1.1 及更新的版本中密码字典会包含额外的字段:

```
>>> ctx.get_ciphers()  # OpenSSL 1.1+
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(256) Mac=AEAD',
  'digest': None,
  'id': 50380848,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES256-GCM-SHA384',
  'protocol': 'TLSv1.2',
  'strength_bits': 256,
  'symmetric': 'aes-256-gcm'},
 {'aead': True,
  'alg_bits': 128,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH     Au=RSA  '
                 'Enc=AESGCM(128) Mac=AEAD',
  'digest': None,
  'id': 50380847,
  'kea': 'kx-ecdhe',
  'name': 'ECDHE-RSA-AES128-GCM-SHA256',
  'protocol': 'TLSv1.2',
  'strength_bits': 128,
  'symmetric': 'aes-128-gcm'}]
```

可用性: OpenSSL 1.0.2+。

3.6 版新加入.

SSLContext.**set_default_verify_paths**()
从构建 OpenSSL 库时定义的文件系统路径中加载一组默认的" 证书颁发机构" (CA) 证书。不幸的是，没有一种简单的方式能知道此方法是否执行成功：如果未找到任何证书也不会返回错误。不过，当 OpenSSL 库是作为操作系统的一部分被提供时，它的配置应当是正确的。

SSLContext.**set_ciphers**(*ciphers*)
为使用此上下文创建的套接字设置可用密码。它应当为 OpenSSL 密码列表格式 的字符串。如果没有可被选择的密码（由于编译时选项或其他配置禁止使用所指定的任何密码），则将引发*SSLError*。

備🄵: 在连接后，SSL 套接字的*SSLSocket.cipher()* 方法将给出当前所选择的密码。

在默认情况下 OpenSSL 1.1.1 会启用 TLS 1.3 密码套件。该套件不能通过*set_ciphers()* 来禁用。

SSLContext.**set_alpn_protocols**(*protocols*)
指定在 SSL/TLS 握手期间套接字应当通告的协议。它应为由 ASCII 字符串组成的列表，例如 `['http/1.1', 'spdy/2']`，按首选顺序排列。协议的选择将在握手期间发生，并依据 **RFC 7301** 来执行。在握手成功后，*SSLSocket.selected_alpn_protocol()* 方法将返回已达成一致的协议。

如果*HAS_ALPN* 为 `False` 则此方法将引发*NotImplementedError*。

当双方都支持 ALPN 但不能就协议达成一致时 OpenSSL 1.1.0 至 1.1.0e 将中止并引发*SSLError*。1.1.0f+ 的行为类似于 1.0.2，*SSLSocket.selected_alpn_protocol()* 返回 None。

3.5 版新加入.

SSLContext.**set_npn_protocols**(*protocols*)
指定在 Specify which protocols the socket should advertise during the SSL/TLS 握手期间套接字应当通告的协议。它应为由字符串组成的列表，例如 `['http/1.1', 'spdy/2']`，按首选顺序排列。协议的选择将在握手期间发生，并将依据 应用层协议协商 来执行。在握手成功后，*SSLSocket.selected_npn_protocol()* 方法将返回已达成一致的协议。

如果*HAS_NPN* 为 `False` 则此方法将引发*NotImplementedError*。

3.3 版新加入.

SSLContext.**sni_callback**
注册一个回调函数，当 TLS 客户端指定了一个服务器名称提示时，该回调函数将在 SSL/TLS 服务器接收到 TLS Client Hello 握手消息后被调用。服务器名称提示机制的定义见 **RFC 6066** section 3 - Server Name Indication。

每个 `SSLContext` 只能设置一个回调。如果 *sni_callback* 被设置为 `None` 则会禁用回调。对该函数的后续调用将禁用之前注册的回调。

此回调函数将附带三个参数来调用；第一个参数是*ssl.SSLSocket*，第二个参数是代表客户端准备与之通信的服务器的字符串 (或者如果 TLS Client Hello 不包含服务器名称则为*None*) 而第三个参数是原来的*SSLContext*。服务器名称参数为文本形式。对于国际化域名，服务器名称是一个 IDN A 标签 (`"xn--pythn-mua.org"`)。

此回调的一个典型用法是将*ssl.SSLSocket* 的*SSLSocket.context* 属性修改为一个*SSLContext* 类型的新对象，该对象代表与服务器相匹配的证书链。

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like *SSLSocket.selected_alpn_protocol()* and *SSLSocket.context*. The *SSLSocket.getpeercert()*, *SSLSocket.cipher()* and *SSLSocket.compression()* methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not return meaningful values nor can they be called safely.

sni_callback 函数必须返回 `None` 以允许 TLS 协商继续进行。如果想要 TLS 失败，则可以返回常量*ALERT_DESCRIPTION_**。其他返回值将导致 TLS 的致命错误*ALERT_DESCRIPTION_INTERNAL_ERROR*。

如果从 *sni_callback* 函数引发了异常，则 TLS 连接将终止并发出 TLS 致命警告消息*ALERT_DESCRIPTION_HANDSHAKE_FAILURE*。

如果 OpenSSL library 库在构建时定义了 OPENSSL_NO_TLSEXT 则此方法将返回*NotImplementedError*。

3.7 版新加入.

SSLContext.**set_servername_callback**(*server_name_callback*)

这是被保留用于向下兼容的旧式 API。在可能的情况下，你应当改用*sni_callback*。给出的 *server_name_callback* 类似于 *sni_callback*，不同之处在于当服务器主机名是 IDN 编码的国际化域名时，*server_name_callback* 会接收到一个已编码的 U 标签 ("pythön.org")。

如果发生了服务器名称解码错误。TLS 连接将终止并向客户端发出*ALERT_DESCRIPTION_INTERNAL_ERROR* 最严重 TLS 警告消息。

3.4 版新加入.

SSLContext.**load_dh_params**(*dhfile*)

加载密钥生成参数用于 Diffie-Hellman (DH) 密钥交换。使用 DH 密钥交换能以消耗（服务器和客户端的）计算资源为代价提升前向保密性。*dhfile* 参数应当为指向一个包含 PEM 格式的 DH 形参的文件的路径。

此设置不会应用于客户端套接字。你还可以使用*OP_SINGLE_DH_USE* 选项来进一步提升安全性。

3.3 版新加入.

SSLContext.**set_ecdh_curve**(*curve_name*)

为基于椭圆曲线的 Elliptic Curve-based Diffie-Hellman (ECDH) 密钥交换设置曲线名称。ECDH 显著快于常规 DH 同时据信同样安全。*curve_name* 形参应为描述某个知名椭圆曲线的字符串，例如受到广泛支持的曲线 prime256v1。

此设置不会应用于客户端套接字。你还可以使用*OP_SINGLE_ECDH_USE* 选项来进一步提升安全性。

如果*HAS_ECDH* 为 False 则此方法将不可用。

3.3 版新加入.

也参考:

SSL/TLS 与完美的前向保密性 Vincent Bernat。

SSLContext.**wrap_socket**(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)

包装一个现有的 Python 套接字 *sock* 并返回一个*SSLContext.sslsocket_class* 的实例 (默认为*SSLSocket*)。返回的 SSL 套接字会绑定上下文、设置以及证书。*sock* 必须是一个*SOCK_STREAM* 套接字；其他套接字类型不被支持。

形参 server_side 是一个布尔值，它标明希望从该套接字获得服务器端行为还是客户端行为。

对于客户端套接字，上下文的构造会延迟执行；如果下层的套接字尚未连接，上下文的构造将在对套接字调用 connect() 之后执行。对于服务器端套接字，如果套接字没有远端对等方，它会被视为一个监听套接字，并且服务器端 SSL 包装操作会在通过 accept() 方法所接受的客户端连接上自动执行。此方法可能会引发*SSLError*。

在客户端连接上，可选形参 *server_hostname* 指定所要连接的服务的主机名。这允许单个服务器托管具有单独证书的多个基于 SSL 的服务，很类似于 HTTP 虚拟主机。如果 *server_side* 为真值则指定 *server_hostname* 将引发*ValueError*。

形参 do_handshake_on_connect 指明是否要在调用 socket.connect() 之后自动执行 SSL 握手，还是要通过发起调用*SSLSocket.do_handshake()* 方法让应用程序显式地调用它。显式地调用*SSLSocket.do_handshake()* 可给予程序对握手中所涉及的套接字 I/O 阻塞行为的控制。

形参 suppress_ragged_eofs 指明 SSLSocket.recv() 方法应当如何从连接的另一端发送非预期的 EOF 信号。如果指定为*True* (默认值)，它将返回正常的 EOF (空字节串对象) 来响应从下层套接字引发的非预期的 EOF 错误；如果指定为*False*，它将向调用方引发异常。

session，参见*session*。

3.5 版更變: 总是允许传送 server_hostname，即使 OpenSSL 没有 SNI。

3.6 版更變: 增加了 *session* 参数。

3.7 版更變: 此方法返回*SSLContext.sslsocket_class* 的实例而非硬编码的*SSLSocket*。

SSLContext.**sslsocket_class**
SSLContext.wrap_socket() 的返回类型，默认为*SSLSocket*。该属性可以在类实例上被重载以便返回自定义的*SSLSocket* 的子类。

3.7 版新加入.

SSLContext.**wrap_bio**(*incoming*, *outgoing*, *server_side=False*, *server_hostname=None*, *session=None*)
包装 BIO 对象 *incoming* 和 *outgoing* 并返回一个*SSLContext.sslobject_class* (默认为*SSLObject*) 的实例。SSL 例程将从 BIO 中读取输入数据并将数据写入到 outgoing BIO。

server_side, *server_hostname* 和 *session* 形参具有与*SSLContext.wrap_socket()* 中相同的含义。

3.6 版更變: 增加了 *session* 参数。

3.7 版更變: 此方法返回*SSLContext.sslobject_class* 的实例则非硬编码的*SSLObject*。

SSLContext.**sslobject_class**
SSLContext.wrap_bio() 的返回类型，默认为*SSLObject*。该属性可以在类实例上被重载以便返回自定义的*SSLObject* 的子类。

3.7 版新加入.

SSLContext.**session_stats**()
获取由此上下文所创建或管理的 SSL 会话的相关统计信息。返回将每个 信息片 的名称映射到其数字值的字典。例如，以下是自上下文被创建以来会话缓存中命中和未命中的总数:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.**check_hostname**
是否要将匹配*SSLSocket.do_handshake()* 中对等方证书的主机名。该上下文的*verify_mode* 必须被设为*CERT_OPTIONAL* 或*CERT_REQUIRED*，并且你必须将 *server_hostname* 传给*wrap_socket()* 以便匹配主机名。启用主机名检查会自动将*verify_mode* 从*CERT_NONE* 设为*CERT_REQUIRED*。只要启用了主机名检查就无法将其设回*CERT_NONE*。*PROTOCOL_TLS_CLIENT* 协议默认启用主机名检查。对于其他协议，则必须显式地启用主机名检查。

示例:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

3.4 版新加入.

3.7 版更變: 现在当主机名检查被启用且*verify_mode* 为*CERT_NONE* 时*verify_mode* 会自动更改为*CERT_REQUIRED*。在之前版本中同样的操作将失败并引发*ValueError*。

備𝔽: 此特性要求 OpenSSL 0.9.8f 或更新的版本。

SSLContext.**keylog_filename**
每当生成或接收到密钥时，将 TLS 密钥写入到一个密钥日志文件。密钥日志文件的设计仅适用于调试目的。文件的格式由 NSS 指明并为许多流量分析工具例如 Wireshark 所使用。日志文件会以追加模式打开。写入操作会在线程之间同步，但不会在进程之间同步。

3.8 版新加入.

備𝔽: 此特性要求 OpenSSL 1.1.1 或更新的版本。

SSLContext.**maximum_version**
一个代表所支持的最高 TLS 版本的*TLSVersion* 枚举成员。该值默认为*TLSVersion.MAXIMUM_SUPPORTED*。这个属性对于*PROTOCOL_TLS*, *PROTOCOL_TLS_CLIENT* 和*PROTOCOL_TLS_SERVER* 以外的其他协议来说都是只读的。

maximum_version, *minimum_version* 和*SSLContext.options* 等属性都会影响上下文所支持的 SSL 和 TLS 版本。这个实现不会阻止无效的组合。例如一个*options* 为*OP_NO_TLSv1_2* 而*maximum_version* 设为*TLSVersion.TLSv1_2* 的上下文将无法建立 TLS 1.2 连接。

備𝔽: 除非 ssl 模块使用 OpenSSL 1.1.0g 或更新的版本编译否则这个属性将不可用。

3.7 版新加入.

SSLContext.**minimum_version**
与*SSLContext.maximum_version* 类似，区别在于它是所支持的最低版本或为*TLSVersion.MINIMUM_SUPPORTED*。

備𝔽: 除非 ssl 模块使用 OpenSSL 1.1.0g 或更新的版本编译否则这个属性将不可用。

3.7 版新加入.

SSLContext.**num_tickets**
控制 TLS_PROTOCOL_SERVER 上下文的 TLS 1.3 会话凭据数量。这个设置不会影响 TLS 1.0 到 1.2 连接。

備𝔽: 除非 ssl 模块使用 OpenSSL 1.1.1 或更新的版本编译否则这个属性将不可用。

3.8 版新加入.

SSLContext.**options**
一个代表此上下文中所启用的 SSL 选项集的整数。默认值为*OP_ALL*，但你也可以通过在选项间进行 OR 运算来指定其他选项例如*OP_NO_SSLv2*。

備𝔽: 对于 0.9.8m 之前的 OpenSSL 版本，只能设置选项，而不能清除它们。尝试（通过重置相应比特

位）清除选项将会引发*ValueError*。

3.6 版更變: *SSLContext.options* 返回*Options* 旗标:

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

SSLContext.**post_handshake_auth**
启用 TLS 1.3 握手后客户端身份验证。握手后验证默认是被禁用的，服务器只能在初始握手期间请求 TLS 客户端证书。当启用时，服务器可以在握手之后的任何时候请求 TLS 客户端证书。

当在客户端套接字上启用时，客户端会向服务器发信号说明它支持握手后身份验证。

当在服务器端套接字上启用时，*SSLContext.verify_mode* 也必须被设为*CERT_OPTIONAL* 或*CERT_REQUIRED*。实际的客户端证书交换会被延迟直至*SSLSocket.verify_client_post_handshake()* 被调用并执行了一些 I/O 操作后再进行。

備𝔽: 仅在 OpenSSL 1.1.1 且 TLS 1.3 被启用时可用。如果没有 TLS 1.3 支持，该属性值将为 None 且不可被更改

3.8 版新加入.

SSLContext.**protocol**
构造上下文时所选择的协议版本。这个属性是只读的。

SSLContext.**hostname_checks_common_name**
在没有目标替代名称扩展的情况下*check_hostname* 是否要回退为验证证书的通用名称（默认为真值）。

備𝔽: 仅在 OpenSSL 1.1.0 或更新的版本上可写。

3.7 版新加入.

3.9.3 版更變: 此旗标在 OpenSSL 1.1.1k 之前的版本上不起作用。Python 3.8.9, 3.9.3, 和 3.10 包含了针对之前版本的变通处理。

SSLContext.**verify_flags**
证书验证操作的旗标。你可以通过对*VERIFY_CRL_CHECK_LEAF* 等值执行 OR 运算来设置组合旗标。在默认情况下 OpenSSL 不会要求也不会验证证书吊销列表（CRL）。仅在 openssl 版本 0.9.8+ 上可用。

3.4 版新加入.

3.6 版更變: *SSLContext.verify_flags* 返回*VerifyFlags* 旗标:

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.**verify_mode**
是否要尝试验证其他对等方的证书以及如果验证失败应采取何种行为。该属性值必须为*CERT_NONE*, *CERT_OPTIONAL* 或*CERT_REQUIRED* 之一。

3.6 版更變: *SSLContext.verify_mode* 返回*VerifyMode* 枚举:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

18.3.4 证书

总的来说证书是公钥/私钥系统的一个组成部分。在这个系统中，每个 *主体* (可能是一台机器、一个人或者一个组织) 都会分配到唯一的包含两部分的加密密钥。一部分密钥是公开的，称为 *公钥*；另一部分密钥是保密的，称为 *私钥*。这两个部分是互相关联的，就是说如果你用其中一个部分来加密一条消息，你将能用并且 **只能**用另一个部分来解密它。

在一个证书中包含有两个主体的相关信息。它包含 *目标方* 的名称和目标方的公钥。它还包含由第二个主体 *颁发方* 所发布的声明：目标方的身份与他们所宣称的一致，包含的公钥也确实是目标方的公钥。颁发方的声明使用颁发方的私钥进行签名，该私钥的内容只有颁发方自己才知道。但是，任何人都可以找到颁发方的公钥，用它来解密这个声明，并将其与证书中的其他信息进行比较来验证颁发方声明的真实性。证书还包含有关其有效期限的信息。这被表示为两个字段，即"notBefore" 和"notAfter"。

在 Python 中应用证书时，客户端或服务器可以用证书来证明自己的身份。还可以要求网络连接的另一方提供证书，提供的证书可以用于验证以满足客户端或服务器的验证要求。如果验证失败，连接尝试可被设置为引发一个异常。验证是由下层的 OpenSSL 框架来自动执行的；应用程序本身不必关注其内部的机制。但是应用程序通常需要提供一组证书以允许此过程的发生。

Python 使用文件来包含证书。它们应当采用"PEM" 格式 (参见 **RFC 1422**)，这是一种带有头部行和尾部行的 base-64 编码包装形式:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

证书链

包含证书的 Python 文件可以包含一系列的证书，有时被称为 *证书链*。这个证书链应当以" 作为" 客户端或服务器的主体的专属证书打头，然后是证书颁发方的证书，然后是 *上述* 证书的颁发方的证书，证书链就这样不断上溯直到你得到一个 *自签名* 的证书，即具有相同目标方和颁发方的证书，有时也称为 *根证书*。在证书文件中这些证书应当被拼接为一体。例如，假设我们有一个包含三个证书的证书链，以我们的服务器证书打头，然后是为我们的服务器证书签名的证书颁发机构的证书，最后是为证书颁发机构的证书颁发证书的机构的根证书:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

CA 证书

如果你想要求对连接的另一方的证书进行验证，你必须提供一个"CA 证书" 文件，其中包含了你愿意信任的每个颁发方的证书链。同样地，这个文件的内容就是这些证书链拼接在一起的结果。为了进行验证，Python 将使用它在文件中找到的第一个匹配的证书链。可以通过调用 *SSLContext.load_default_certs()* 来使用系统平台的证书文件，这可以由 *create_default_context()* 自动完成。

合并的密钥和证书

私钥往往与证书存储在相同的文件中；在此情况下，只需要将 certfile 形参传给*SSLContext.load_cert_chain()* 和*wrap_socket()*。如果私钥是与证书一起存储的，则它应当放在证书链的第一个证书之前:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

自签名证书

如果你准备创建一个提供 SSL 加密连接服务的服务器，你需要为该服务获取一份证书。有许多方式可以获取合适的证书，例如从证书颁发机构购买。另一种常见做法是生成自签名证书。生成自签名证书的最简单方式是使用 OpenSSL 软件包，代码如下所示:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
.............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

自签名证书的缺点在于它是它自身的根证书，因此不会存在于别人的已知（且信任的）根证书缓存当中。

18.3.5 示例

检测 SSL 支持

要检测一个 Python 安装版中是否带有 SSL 支持，用户代码应当使用以下例程:

```
try:
    import ssl
except ImportError:
    pass
else:
    ...  # do something that requires SSL support
```

客户端操作

这个例子创建了一个 SSL 上下文并使用客户端套接字的推荐安全设置，包括自动证书验证:

```
>>> context = ssl.create_default_context()
```

如果你喜欢自行调整安全设置，你可能需要从头创建一个上下文（但是请请注意避免不正确的设置）：

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(这段代码假定你的操作系统将所有 CA 证书打包存放于 `/etc/ssl/certs/ca-bundle.crt`；如果不是这样，你将收到报错信息，必须修改此位置)

PROTOCOL_TLS_CLIENT 协议配置用于证书验证和主机名验证的上下文。*verify_mode* 设为*CERT_REQUIRED* 而*check_hostname* 设为 `True`。所有其他协议都会使用不安全的默认值创建 SSL 上下文。

当你使用此上下文去连接服务器时，*CERT_REQUIRED* 和*check_hostname* 会验证服务器证书；它将确认服务器证书使用了某个 CA 证书进行签名，检查签名是否正确，并验证其他属性例如主机名的有效性和身份真实性:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                            server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

你可以随后获取该证书:

```
>>> cert = conn.getpeercert()
```

可视化检查显示证书能够证明目标服务 (即 HTTPS 主机 `www.python.org`) 的身份:

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt
↪',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': ((('businessCategory', 'Private Organization'),),
             (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
             (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
             (('serialNumber', '3359300'),),
             (('streetAddress', '16 Allen Rd'),),
             (('postalCode', '03894-4801'),),
             (('countryName', 'US'),),
             (('stateOrProvinceName', 'NH'),),
             (('localityName', 'Wolfeboro'),),
             (('organizationName', 'Python Software Foundation'),),
             (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
```

(下页继续)

(繼續上一頁)

```
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

现在 SSL 通道已建立并已验证了证书，你可以继续与服务器对话了:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
 b'X-Cache-Hits: 11',
 b'Vary: Cookie',
 b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
 b'Connection: close',
 b'',
 b'']
```

参见下文对于安全考量 的讨论。

服务器端操作

对于服务器操作，通常你需要在文件中存放服务器证书和私钥各一份。你将首先创建一个包含密钥和证书的上下文，这样客户端就能检查你的身份真实性。然后你将打开一个套接字，将其绑定到一个端口，在其上调用 `listen()`，并开始等待客户端连接:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

bindsocket = socket.socket()
bindsocket.bind(('myaddr.example.com', 10023))
bindsocket.listen(5)
```

当有客户端连接时，你将在套接字上调用 `accept()` 以从另一端获取新的套接字，并使用上下文的*`SSLContext.wrap_socket()`* 方法来为连接创建一个服务器端 SSL 套接字:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

随后你将从 connstream 读取数据并对其进行处理，直至你结束与客户端的会话（或客户端结束与你的会话）：

```
def deal_with_client(connstream):
    data = connstream.recv(1024)
    # empty data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.recv(1024)
    # finished with client
```

并返回至监听新的客户端连接（当然，真正的服务器应当会在单独的线程中处理每个客户端连接，或者将套接字设为非阻塞模式 并使用事件循环）。

18.3.6 关于非阻塞套接字的说明

在非阻塞模式下 SSL 套接字的行为与常规套接字略有不同。当使用非阻塞模式时，你需要注意下面这些事情:

- 如果一个 I/O 操作会阻塞，大多数*SSLSocket* 方法都将引发*SSLWantWriteError* 或*SSLWantReadError* 而非*BlockingIOError*。如果有必要在下层套接字上执行读取操作将引发*SSLWantReadError*，在下层套接字上执行写入操作则将引发*SSLWantWriteError*。请注意尝试 写入到 SSL 套接字可能需要先从下层套接字 读取，而尝试从 SSL 套接字 读取则可能需要先向下层套接字 写入。

  3.5 版更變: 在较早的 Python 版本中，SSLSocket.send() 方法会返回零值而非引发*SSLWantWriteError* 或*SSLWantReadError*。

- 调用*select()* 将告诉你可以从 OS 层级的套接字读取 (或向其写入)，但这并不意味着在上面的 SSL 层有足够的数据。例如，可能只有部分 SSL 帧已经到达。因此，你必须准备好处理 SSLSocket.recv() 和 SSLSocket.send() 失败的情况，并在再次调用*select()* 之后重新尝试。

- 相反地，由于 SSL 层具有自己的帧机制，一个 SSL 套接字可能仍有可读取的数据而*select()* 并不知道这一点。因此，你应当先调用 SSLSocket.recv() 取走所有潜在的可用数据，然后只在必要时对*select()* 调用执行阻塞。

  (当然，类似的保留规则在使用其他原语例如*poll()*，或*selectors* 模块中的原语时也适用)

- SSL 握手本身将是非阻塞的: *SSLSocket.do_handshake()* 方法必须不断重试直至其成功返回。下面是一个使用*select()* 来等待套接字就绪的简短例子:

  ```
  while True:
      try:
          sock.do_handshake()
          break
  ```

(下页继续)

(繼續上一頁)

```
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
    except ssl.SSLWantWriteError:
        select.select([], [sock], [])
```

也參考:

asyncio 模块支持*非阻塞 SSL 套接字* 并提供了更高层级的 API。它会使用*selectors* 模块来轮询事件并处理*SSLWantWriteError*, *SSLWantReadError* 和*BlockingIOError* 等异常。它还会异步地执行 SSL 握手。

18.3.7 内存 BIO 支持

3.5 版新加入.

自从 SSL 模块在 Python 2.6 起被引入之后，*SSLSocket* 类提供了两个互相关联但彼此独立的功能分块:

- SSL 协议处理
- 网络 IO

网络 IO API 与*socket.socket* 所提供的功能一致，*SSLSocket* 也是从那里继承而来的。这允许 SSL 套接字被用作常规套接字的替代，使得向现有应用程序添加 SSL 支持变得非常容易。

将 SSL 协议处理与网络 IO 结合使用通常都能运行良好，但在某些情况下则不能。此情况的一个例子是 async IO 框架，该框架要使用不同的 IO 多路复用模型而非 (基于就绪状态的) " 在文件描述器上执行选择/轮询" 模型，该模型是*socket.socket* 和内部 OpenSSL 套接字 IO 例程正常运行的假设前提。这种情况在该模型效率不高的 Windows 平台上最为常见。为此还提供了一个*SSLSocket* 的简化形式，称为*SSLObject*。

class ssl.**SSLObject**

SSLSocket 的简化形式，表示一个不包含任何网络 IO 方法的 SSL 协议实例。这个类通常由想要通过内存缓冲区为 SSL 实现异步 IO 的框架作者来使用。

这个类在低层级 SSL 对象上实现了一个接口，与 OpenSSL 所实现的类似。此对象会捕获 SSL 连接的状态但其本身不提供任何网络 IO。IO 需要通过单独的"BIO" 对象来执行，该对象是 OpenSSL 的 IO 抽象层。

这个类没有公有构造器。*SSLObject* 实例必须使用*wrap_bio()* 方法来创建。此方法将创建*SSLObject* 实例并将其绑定到一个 BIO 对。其中 *incoming* BIO 用来将数据从 Python 传递到 SSL 协议实例，而 *outgoing* BIO 用来进行数据反向传递。

可以使用以下方法:

- *context*
- *server_side*
- *server_hostname*
- *session*
- *session_reused*
- *read()*
- *write()*
- *getpeercert()*
- *selected_alpn_protocol()*
- *selected_npn_protocol()*

- *cipher()*
- *shared_ciphers()*
- *compression()*
- *pending()*
- *do_handshake()*
- *verify_client_post_handshake()*
- *unwrap()*
- *get_channel_binding()*
- *version()*

与*SSLSocket* 相比，此对象缺少下列特性:

- 任何形式的网络 IO; `recv()` 和 `send()` 仅对下层的*MemoryBIO* 缓冲区执行读取和写入。
- 不存在 *do_handshake_on_connect* 机制。你必须总是手动调用*do_handshake()* 来开始握手操作。
- 不存在对 *suppress_ragged_eofs* 的处理。所有违反协议的文件结束条件将通过*SSLEOFError* 异常来报告。
- 方法*unwrap()* 的调用不返回任何东西，不会如 SSL 套接字那样返回下层的套接字。
- *server_name_callback* 回调被传给*SSLContext.set_servername_callback()* 时将获得一个*SSLObject* 实例而非*SSLSocket* 实例作为其第一个形参。

有关*SSLObject* 用法的一些说明:

- 在*SSLObject* 上的所有 IO 都是*非阻塞的*。这意味着例如*read()* 在其需要比 incoming BIO 可用的更多数据时将会引发*SSLWantReadError*。
- 不存在模块层级的 `wrap_bio()` 调用，就像*wrap_socket()* 那样。*SSLObject* 总是通过*SSLContext* 来创建。

3.7 版更變: *SSLObject* 的实例必须使用*wrap_bio()* 来创建。在较早的版本中，直接创建实例是可能的。但这从未被记入文档或是被正式支持。

SSLObject 会使用内存缓冲区与外部世界通信。*MemoryBIO* 类提供了可被用于此目的的内存缓冲区。它包装了一个 OpenSSL 内存 BIO (Basic IO) 对象:

class ssl.**MemoryBIO**

一个可被用来在 Python 和 SSL 协议实例之间传递数据的内存缓冲区。

pending

返回当前存在于内存缓冲区的字节数。

eof

一个表明内存 BIO 目前是否位于文件末尾的布尔值。

read(*n=-1*)

从内存缓冲区读取至多 *n* 个字节。如果 *n* 未指定或为负值，则返回全部字节数据。

write(*buf*)

将字节数据从 *buf* 写入到内存 BIO。*buf* 参数必须为支持缓冲区协议的对象。

返回值为写入的字节数，它总是与 *buf* 的长度相等。

write_eof()

将一个 EOF 标记写入到内存 BIO。在此方法被调用以后，再调用*write()* 将是非法的。属性*eof* will 在缓冲区当前的所有数据都被读取之后将变为真值。

18.3.8 SSL 会话

3.6 版新加入.

class ssl.**SSLSession**

session 所使用的会话对象。

id

time

timeout

ticket_lifetime_hint

has_ticket

18.3.9 安全考量

最佳默认值

针对 **客户端使用**，如果你对于安全策略没有任何特殊要求，则强烈推荐你使用*create_default_context()* 函数来创建你的 SSL 上下文。它将加载系统的受信任 CA 证书，启用证书验证和主机名检查，并尝试合理地选择安全的协议和密码设置。

例如，以下演示了你应当如何使用*smtplib.SMTP* 类来创建指向一个 SMTP 服务器的受信任且安全的连接:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

如果连接需要客户端证书，可使用*SSLContext.load_cert_chain()* 来添加。

作为对比，如果你通过自行调用*SSLContext* 构造器来创建 SSL 上下文，它默认将不会启用证书验证和主机名检查。如果你这样做，请阅读下面的段落以达到良好的安全级别。

手动设置

验证证书

当直接调用*SSLContext* 构造器时，默认会使用*CERT_NONE*。由于它不会验证对等方的身份真实性，因此是不安全的，特别是在客户端模式下，大多数时候你都希望能保证你所连接的服务器的身份真实性。因此，当处于客户端模式时，强烈推荐使用*CERT_REQUIRED*。但是，光这样还不够；你还必须检查服务器证书，这可以通过调用*SSLSocket.getpeercert()* 来获取并匹配目标服务。对于许多协议和应用来说，服务可通过主机名来标识；在此情况下，可以使用*match_hostname()* 函数。这种通用检测会在*SSLContext.check_hostname* 被启用时自动执行。

3.7 版更變: 主机名匹配现在是由 OpenSSL 来执行的。Python 不会再使用*match_hostname()*。

在服务器模式下，如果你想要使用 SSL 层来验证客户端（而不是使用更高层级的验证机制），你也必须要指定*CERT_REQUIRED* 并以类似方式检查客户端证书。

协议版本

SSL 版本 2 和 3 被认为是不安全的因而使用它们会有风险。如果你想要客户端和服务器之间有最大的兼容性，推荐使用*PROTOCOL_TLS_CLIENT* 或*PROTOCOL_TLS_SERVER* 作为协议版本。SSLv2 和 SSLv3 默认会被禁用。

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.options |= ssl.OP_NO_TLSv1
>>> client_context.options |= ssl.OP_NO_TLSv1_1
```

前面创建的 SSL 上下文将只允许 TLSv1.2 及更新版本（如果你的系统支持）的服务器连接。*PROTOCOL_TLS_CLIENT* 默认会使用证书验证和主机名检查。你必须将证书加载到上下文中。

密码选择

如果你有更高级的安全要求，也可以通过*SSLContext.set_ciphers()* 方法在协商 SSL 会话时对所启用的加密进行微调。从 Python 3.2.3 开始，ssl 默认会禁用某些较弱的加密，但你还可能希望进一步限制加密选项。请确保仔细阅读 OpenSSL 文档中有关 加密列表格式 的部分。如果你想要检查给定的加密列表启用了哪些加密，可以使用*SSLContext.get_ciphers()* 或所在系统的 `openssl ciphers` 命令。

多进程

如果使用此模块作为多进程应用的一部分（例如使用*multiprocessing* 或*concurrent.futures* 模块），请注意 OpenSSL 的内部随机数字生成器并不能正确处理分支进程。应用程序必须修改父进程的 PRNG 状态，如果它们要使用任何包含*os.fork()* 的 SSL 特性的话。任何对*RAND_add()*, *RAND_bytes()* 或*RAND_pseudo_bytes()* 都可以做到这一点。

18.3.10 TLS 1.3

3.7 版新加入.

Python 通过 OpenSSL 1.1.1 提供了临时性和实验性的 TLS 1.3 支持。这个新协议的行为与之前版本的 TLS/SSL 略有不同。某些新的 TLS 1.3 特性暂时还不可用。

- TLS 1.3 使用一组不同的加密套件集。默认情况下所有 AES-GCM 和 ChaCha20 加密套件都会被启用。*SSLContext.set_ciphers()* 方法还不能启用或禁用任何 TLS 1.3 加密，但*SSLContext.get_ciphers()* 会返回它们。
- 会话凭据不再会作为初始握手的组成部分被发送而是以不同的方式来处理。*SSLSocket.session* 和*SSLSession* 与 TLS 1.3 不兼容。
- 客户端证书在初始握手期间也不会再被验证。服务器可以在任何时候请求证书。客户端会在它们从服务器发送或接收应用数据时处理证书请求。
- 早期数据、延迟的 TLS 客户端证书请求、签名算法配置和密钥重生成等 TLS 1.3 特性尚未被支持。

18.3.11 LibreSSL 支持

LibreSSL 是 OpenSSL 1.0.1 的一个分支。ssl 模块包含对 LibreSSL 的有限支持。当 ssl 模块使用 LibreSSL 进行编译时某些特性将不可用。

- LibreSSL >= 2.6.1 不再支持 NPN。*SSLContext.set_npn_protocols()* 和*SSLSocket.selected_npn_protocol()* 方法将不可用。
- *SSLContext.set_default_verify_paths()* 会忽略环境变量 SSL_CERT_FILE 和 SSL_CERT_PATH，虽然*get_default_verify_paths()* 仍然支持它们。

也参考:

Class *socket.socket* 下层*socket* 类的文档

SSL/TLS 高强度加密：概述 Apache HTTP Server 文档介绍

RFC 1422: 因特网电子邮件的隐私加强：第二部分：基于证书的密钥管理 Steve Kent

RFC 4086: 确保安全的随机性要求 Donald E., Jeffrey I. Schiller

RFC 5280: 互联网 X.509 公钥基础架构证书和证书吊销列表 (CRL) 配置文件 D. Cooper

RFC 5246: 传输层安全性 (TLS) 协议版本 1.2 T. Dierks et. al.

RFC 6066: 传输层安全性 (TLS) 的扩展 D. Eastlake

IANA TLS: 传输层安全性 (TLS) 的参数 IANA

RFC 7525: 传输层安全性 (TLS) 和数据报传输层安全性 (DTLS) 的安全使用建议 IETF

Mozilla 的服务器端 TLS 建议 Mozilla

18.4 select --- 等待 I/O 完成

该模块提供了对 select() 和 poll() 函数的访问，这些函数在大多数操作系统中是可用的。在 Solaris 及其衍生版本上可用 devpoll()，在 Linux 2.5+ 上可用 epoll()，在大多数 BSD 上可用 kqueue()。注意，在 Windows 上，本模块仅适用于套接字；在其他操作系统上，本模块也适用于其他文件类型（特别地，在 Unix 上也适用于管道）。本模块不能用于常规文件，不能检测出（自上次读取文件后）文件是否有新数据写入。

備🄴: *selectors* 模块是在*select* 模块原型的基础上进行高级且高效的 I/O 复用。推荐用户改用*selectors* 模块，除非用户希望对 OS 级的函数原型进行精确控制。

该模块定义以下内容：

exception select.**error**
一个被弃用的*OSError* 的别名。

3.3 版更變: 根据 **PEP 3151**，这个类是*OSError* 的别名。

select.**devpoll**()
（仅支持 Solaris 及其衍生版本）返回一个 /dev/poll 轮询对象，请参阅下方*/dev/poll 轮询对象* 获取 devpoll 对象所支持的方法。

devpoll() 对象与实例化时允许的文件描述符数量有关，如果在程序中降低了此数值，devpoll() 调用将失败。如果程序提高了此数值，devpoll() 可能会返回一个不完整的活动文件描述符列表。

新的文件描述符是不可继承的。

3.3 版新加入.

3.4 版更變: 新的文件描述符现在是不可继承的。

select.**epoll**(*sizehint=-1*, *flags=0*)
（仅支持 Linux 2.5.44 或更高版本）返回一个 edge poll 对象，该对象可作为 I/O 事件的边缘触发或水平触发接口。

sizehint 指示 epoll 预计需要注册的事件数。它必须为正数，或为 *-1* 以使用默认值。它仅在 epoll_create1() 不可用的旧系统上会被用到，其他情况下它没有任何作用（尽管仍会检查其值）。

flags 已经弃用且完全被忽略。但是，如果提供该值，则它必须是 0 或 select.EPOLL_CLOEXEC，否则会抛出 OSError 异常。

请参阅下方边缘触发和水平触发的轮询 *(epoll)* 对象 获取 epoll 对象所支持的方法。

epoll 对象支持上下文管理器：当在 with 语句中使用时，新建的文件描述符会在运行至语句块结束时自动关闭。

新的文件描述符是不可继承的。

3.3 版更變: 增加了 *flags* 参数。

3.4 版更變: 增加了对 with 语句的支持。新的文件描述符现在是不可继承的。

3.4 版後已🄵用: *flags* 参数。现在默认采用 select.EPOLL_CLOEXEC 标志。使用*os.set_inheritable()* 来让文件描述符可继承。

select.**poll**()
（部分操作系统不支持）返回一个 poll 对象，该对象支持注册和注销文件描述符，支持对描述符进行轮询以获取 I/O 事件。请参阅下方*Poll* 对象 获取 poll 对象所支持的方法。

select.**kqueue**()
（仅支持 BSD）返回一个内核队列对象，请参阅下方*Kqueue* 对象 获取 kqueue 对象所支持的方法。

新的文件描述符是不可继承的。

3.4 版更變: 新的文件描述符现在是不可继承的。

select.**kevent**(*ident*, *filter=KQ_FILTER_READ*, *flags=KQ_EV_ADD*, *fflags=0*, *data=0*, *udata=0*)
（仅支持 BSD）返回一个内核事件对象，请参阅下方*Kevent* 对象 获取 kevent 对象所支持的方法。

select.**select**(*rlist*, *wlist*, *xlist*[, *timeout*])
这是一个明白直观的 Unix select() 系统调用接口。前三个参数是由‘可等待对象’组成的序列：可以是代表文件描述符的整数，或是带有名为*fileno()* 的返回这样的整数的无形参方法的对象:

- *rlist*：等待，直到可以开始读取
- *wlist*：等待，直到可以开始写入
- *xlist*：等待“异常情况”（请参阅当前系统的手册，以获取哪些情况称为异常情况）

允许空的可迭代对象，但是否接受三个空的可迭代对象则取决于具体平台。（已知在 Unix 上可行但在 Windows 上不可行。）可选的 *timeout* 参数以一个浮点数表示超时秒数。当省略 *timeout* 参数时该函数将阻塞直到至少有一个文件描述符准备就绪。超时值为零表示执行轮询且永不阻塞。

返回值是三个列表，包含已就绪对象，返回的三个列表是前三个参数的子集。当超时时间已到且没有文件描述符就绪时，返回三个空列表。

可迭代对象中可接受的对象类型有 Python 文件对象 (例如 sys.stdin 以及*open()* 或*os.popen()* 所返回的对象)，由*socket.socket()* 返回的套接字对象等。你也可以自定义一个 *wrapper* 类，只要它具有适当的*fileno()* 方法（该方法要确实返回一个文件描述符，而不能只是一个随机整数）。

> **備𝔽:** Windows 上不接受文件对象，但接受套接字。在 Windows 上，底层的 `select()` 函数由 WinSock 库提供，且不处理不是源自 WinSock 的文件描述符。

3.5 版更變: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出 *InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

`select.`**`PIPE_BUF`**
当一个管道已经被 *select()*、*poll()* 或本模块中的某个接口报告为可写入时，可以在不阻塞该管道的情况下写入的最小字节数。它不适用于套接字等其他类型的文件类对象。

POSIX 上须保证该值不小于 512。

可用性: Unix

3.2 版新加入.

18.4.1 /dev/poll 轮询对象

Solaris 及其衍生版本具备 `/dev/poll`。`select()` 复杂度为 O（最高文件描述符），`poll()` 为 O（文件描述符数量），而 `/dev/poll` 为 O（活动的文件描述符）。

`/dev/poll` 的行为与标准 `poll()` 对象十分类似。

`devpoll.`**`close`**`()`
关闭轮询对象的文件描述符。

3.4 版新加入.

`devpoll.`**`closed`**
如果轮询对象已关闭，则返回 `True`。

3.4 版新加入.

`devpoll.`**`fileno`**`()`
返回轮询对象的文件描述符对应的数字。

3.4 版新加入.

`devpoll.`**`register`**`(`*fd*[, *eventmask*]`)`
在轮询对象中注册文件描述符。这样，将来调用 *poll()* 方法时将检查文件描述符是否有未处理的 I/O 事件。*fd* 可以是整数，也可以是带有 *fileno()* 方法的对象（该方法返回一个整数）。文件对象已经实现了 `fileno()`，因此它们也可以用作参数。

eventmask 是可选的位掩码，用于指定要检查的事件类型。这些常量与 `poll()` 对象所用的相同。本参数的默认值是常量 `POLLIN`、`POLLPRI` 和 `POLLOUT` 的组合。

> **警告:** 注册已注册过的文件描述符不会报错，但是结果是不确定的。正确的操作是先注销或直接修改它。与 `poll()` 相比，这是一个重要的区别。

`devpoll.`**`modify`**`(`*fd*[, *eventmask*]`)`
此方法先执行 *unregister()* 后执行 *register()*。直接执行此操作效率（稍微）高一些。

`devpoll.`**`unregister`**`(`*fd*`)`
删除轮询对象正在跟踪的某个文件描述符。与 *register()* 方法类似，*fd* 可以是整数，也可以是带有 *fileno()* 方法的对象（该方法返回一个整数）。

尝试删除从未注册过的文件描述符将被安全地忽略。

devpoll.**poll**([*timeout*])
轮询已注册的文件描述符的集合，并返回一个列表，列表可能为空，也可能有多个 (fd, event) 二元组，其中包含了要报告事件或错误的描述符。*fd* 是文件描述符，*event* 是一个位掩码，表示该描述符所报告的事件 --- POLLIN 表示可以读取，POLLOUT 表示该描述符可以写入，依此类推。空列表表示调用超时，没有任何文件描述符报告事件。如果指定了 *timeout*，它将指定系统等待事件时，等待多长时间后返回（以毫秒为单位）。如果 *timeout* 为空，-1 或 *None*，则本调用将阻塞，直到轮询对象发生事件为止。

3.5 版更變: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出 *InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

18.4.2 边缘触发和水平触发的轮询 (epoll) 对象

https://linux.die.net/man/4/epoll

eventmask

常数	意义
EPOLLIN	可读
EPOLLOUT	可写
EPOLLPRI	紧急数据读取
EPOLLERR	在关联的文件描述符上有错误情况发生
EPOLLHUP	关联的文件描述符已挂起
EPOLLET	设置触发方式为边缘触发，默认为水平触发
EPOLLONESHOT	设置 one-shot 模式。触发一次事件后，该描述符会在轮询对象内部被禁用。
EPOLLEXCLUSIVE	当已关联的描述符发生事件时，仅唤醒一个 epoll 对象。默认（如果未设置此标志）是唤醒所有轮询该描述符的 epoll 对象。
EPOLLRDHUP	流套接字的对侧关闭了连接或关闭了写入到一半的连接。
EPOLLRDNORM	等同于 EPOLLIN
EPOLLRDBAND	可以读取优先数据带。
EPOLLWRNORM	等同于 EPOLLOUT
EPOLLWRBAND	可以写入优先级数据。
EPOLLMSG	忽略

3.6 版新加入: 增加了 EPOLLEXCLUSIVE。仅支持 Linux Kernel 4.5 或更高版本。

epoll.**close**()
关闭用于控制 epoll 对象的文件描述符。

epoll.**closed**
如果 epoll 对象已关闭，则返回 True。

epoll.**fileno**()
返回文件描述符对应的数字，该描述符用于控制 epoll 对象。

epoll.**fromfd**(*fd*)
根据给定的文件描述符创建 epoll 对象。

epoll.**register**(*fd*[, *eventmask*])
在 epoll 对象中注册一个文件描述符。

epoll.**modify**(*fd*, *eventmask*)
修改一个已注册的文件描述符。

epoll.**unregister**(*fd*)
从 epoll 对象中删除一个已注册的文件描述符。

3.9 版更變: 此方法不会再忽略*EBADF* 错误。

epoll.**poll**(*timeout=None*, *maxevents=-1*)
等待事件发生，timeout 是浮点数，单位为秒。

3.5 版更變: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出*InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

18.4.3 Poll 对象

大多数 Unix 系统支持 `poll()` 系统调用，为服务器提供了更好的可伸缩性，使服务器可以同时服务于大量客户端。`poll()` 的伸缩性更好，因为该调用内部仅列出所关注的文件描述符，而 `select()` 会构造一个 bitmap，在其中将所关注的描述符所对应的 bit 打开，然后重新遍历整个 bitmap。因此 `select()` 复杂度是 O（最高文件描述符），而 `poll()` 是 O（文件描述符数量）。

poll.**register**(*fd*[, *eventmask*])
在轮询对象中注册文件描述符。这样，将来调用*poll()* 方法时将检查文件描述符是否有未处理的 I/O 事件。*fd* 可以是整数，也可以是带有*fileno()* 方法的对象（该方法返回一个整数）。文件对象已经实现了 `fileno()`，因此它们也可以用作参数。

eventmask 是可选的位掩码，用于指定要检查的事件类型，它可以是常量 `POLLIN`、`POLLPRI` 和 `POLLOUT` 的组合，如下表所述。如果未指定本参数，默认将会检查所有 3 种类型的事件。

常数	意义
`POLLIN`	有要读取的数据
`POLLPRI`	有紧急数据需要读取
`POLLOUT`	准备输出：写不会阻塞
`POLLERR`	某种错误条件
`POLLHUP`	挂起
`POLLRDHUP`	流套接字的对侧关闭了连接，或关闭了写入到一半的连接
`POLLNVAL`	无效的请求：描述符未打开

注册已注册过的文件描述符不会报错，且等同于只注册一次该描述符。

poll.**modify**(*fd*, *eventmask*)
修改一个已注册的文件描述符，等同于 `register(fd, eventmask)`。尝试修改未注册的文件描述符会抛出*OSError* 异常，错误码为 `ENOENT`。

poll.**unregister**(*fd*)
删除轮询对象正在跟踪的某个文件描述符。与*register()* 方法类似，*fd* 可以是整数，也可以是带有*fileno()* 方法的对象（该方法返回一个整数）。

尝试删除从未注册过的文件描述符会抛出*KeyError* 异常。

poll.**poll**([*timeout*])
轮询已注册的文件描述符的集合，并返回一个列表，列表可能为空，也可能有多个 `(fd, event)` 二元组，其中包含了要报告事件或错误的描述符。*fd* 是文件描述符，*event* 是一个位掩码，表示该描述符所报告的事件 --- `POLLIN` 表示可以读取，`POLLOUT` 表示该描述符可以写入，依此类推。空列表表示调用超时，没有任何文件描述符报告事件。如果指定了 *timeout*，它将指定系统等待事件时，等待多长时间后返回（以毫秒为单位）。如果 *timeout* 为空、负数或*None*，则本调用将阻塞，直到轮询对象发生事件为止。

3.5 版更變: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出*InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

18.4.4 Kqueue 对象

kqueue.**close**()
关闭用于控制 kqueue 对象的文件描述符。

kqueue.**closed**
如果 kqueue 对象已关闭，则返回 True。

kqueue.**fileno**()
返回文件描述符对应的数字，该描述符用于控制 epoll 对象。

kqueue.**fromfd**(*fd*)
根据给定的文件描述符创建 kqueue 对象。

kqueue.**control**(*changelist*, *max_events*[, *timeout*]) → eventlist
Kevent 的低级接口

- changelist 必须是一个可迭代对象，迭代出 kevent 对象，否则置为 None。
- max_events 必须是 0 或一个正整数。
- timeout 单位为秒（一般为浮点数），默认为 None，即永不超时。

3.5 版更變: 现在，当本函数被信号中断时，重试超时将从头开始计时，不会抛出 *InterruptedError* 异常。除非信号处理程序抛出异常（相关原理请参阅 **PEP 475**）。

18.4.5 Kevent 对象

https://www.freebsd.org/cgi/man.cgi?query=kqueue&sektion=2

kevent.**ident**
用于区分事件的标识值。其解释取决于筛选器，但该值通常是文件描述符。在构造函数中，该标识值可以是整数或带有 *fileno()* 方法的对象。kevent 在内部存储整数。

kevent.**filter**
内核筛选器的名称。

常数	意义
KQ_FILTER_READ	获取描述符，并在有数据可读时返回
KQ_FILTER_WRITE	获取描述符，并在有数据可写时返回
KQ_FILTER_AIO	AIO 请求
KQ_FILTER_VNODE	当在 *fflag* 中监视的一个或多个请求事件发生时返回
KQ_FILTER_PROC	监视进程 ID 上的事件
KQ_FILTER_NETDEV	Watch for events on a network device [not available on macOS]
KQ_FILTER_SIGNAL	每当监视的信号传递到进程时返回
KQ_FILTER_TIMER	建立一个任意的计时器

kevent.**flags**
筛选器操作。

常数	意义
KQ_EV_ADD	添加或修改事件
KQ_EV_DELETE	从队列中删除事件
KQ_EV_ENABLE	Permitscontrol() 返回事件
KQ_EV_DISABLE	禁用事件
KQ_EV_ONESHOT	在第一次发生后删除事件
KQ_EV_CLEAR	检索事件后重置状态
KQ_EV_SYSFLAGS	内部事件
KQ_EV_FLAG1	内部事件
KQ_EV_EOF	筛选特定 EOF 条件
KQ_EV_ERROR	请参阅返回值

kevent.**fflags**
筛选特定标志。

KQ_FILTER_READ 和 KQ_FILTER_WRITE 筛选标志：

常数	意义
KQ_NOTE_LOWAT	套接字缓冲区的低水线

KQ_FILTER_VNODE 筛选标志：

常数	意义
KQ_NOTE_DELETE	已调用 *unlink()*
KQ_NOTE_WRITE	发生写入
KQ_NOTE_EXTEND	文件已扩展
KQ_NOTE_ATTRIB	属性已更改
KQ_NOTE_LINK	链接计数已更改
KQ_NOTE_RENAME	文件已重命名
KQ_NOTE_REVOKE	对文件的访问权限已被撤销

KQ_FILTER_PROC filter flags:

常数	意义
KQ_NOTE_EXIT	进程已退出
KQ_NOTE_FORK	该进程调用了 *fork()*
KQ_NOTE_EXEC	进程已执行新进程
KQ_NOTE_PCTRLMASK	内部筛选器标志
KQ_NOTE_PDATAMASK	内部筛选器标志
KQ_NOTE_TRACK	跨 *fork()* 执行进程
KQ_NOTE_CHILD	在 *NOTE_TRACK* 的子进程上返回
KQ_NOTE_TRACKERR	无法附加到子对象

KQ_FILTER_NETDEV filter flags (not available on macOS):

常数	意义
KQ_NOTE_LINKUP	链接已建立
KQ_NOTE_LINKDOWN	链接已断开
KQ_NOTE_LINKINV	链接状态无效

kevent.**data**
　筛选特定数据。

kevent.**udata**
　用户自定义值。

18.5 selectors --- 高级 I/O 复用库

3.4 版新加入.

源码: Lib/selectors.py

18.5.1 簡介

此模块允许高层级且高效率的 I/O 复用，它建立在*select* 模块原型的基础之上。推荐用户改用此模块，除非他们希望对所使用的 OS 层级原型进行精确控制。

它定义了一个*BaseSelector* 抽象基类，以及多个实际的实现 (*KqueueSelector*, *EpollSelector*...)，它们可被用于在多个文件对象上等待 I/O 就绪通知。在下文中，" 文件对象" 是指任何具有 fileno() 方法的对象，或是一个原始文件描述器。参见*file object*。

DefaultSelector 是一个指向当前平台上可用的最高效实现的别名：这应为大多数用户的默认选择。

備E: 受支持的文件对象类型取决于具体平台：在 Windows 上，支持套接字但不支持管道，而在 Unix 上两者均受支持（某些其他类型也可能受支持，例如 fifo 或特殊文件设备等）。

也参考:

select 低层级的 I/O 多路复用模块。

18.5.2 类

类的层次结构:

```
BaseSelector
+-- SelectSelector
+-- PollSelector
+-- EpollSelector
+-- DevpollSelector
+-- KqueueSelector
```

下文中，*events* 一个位掩码，指明哪些 I/O 事件要在给定的文件对象上执行等待。它可以是以下模块级常量的组合:

常量	含意
EVENT_READ	可读
EVENT_WRITE	可写

class selectors.**SelectorKey**

SelectorKey 是一个*namedtuple*，用来将文件对象关联到其下层的文件描述器、选定事件掩码和附加数据等。它会被某些*BaseSelector* 方法返回。

fileobj
已注册的文件对象。

fd
下层的文件描述器。

events
必须在此文件对象上被等待的事件。

data
可选的关联到此文件对象的不透明数据：例如，这可被用来存储各个客户端的会话 ID。

class selectors.**BaseSelector**

一个*BaseSelector*，用来在多个文件对象上等待 I/O 事件就绪。它支持文件流注册、注销，以及在这些流上等待 I/O 事件的方法。它是一个抽象基类，因此不能被实例化。请改用*DefaultSelector*，或者*SelectSelector*，*KqueueSelector* 等。如果你想要指明使用某个实现，并且你的平台支持它的话。*BaseSelector* 及其具体实现支持*context manager* 协议。

abstractmethod register(*fileobj*, *events*, *data=None*)
注册一个用于选择的文件对象，在其上监视 I/O 事件。

fileobj 是要监视的文件对象。它可以是整数形式的文件描述符或者具有 `fileno()` 方法的对象。*events* 是要监视的事件的位掩码。*data* 是一个不透明对象。

这将返回一个新的*SelectorKey* 实例，或在出现无效事件掩码或文件描述符时引发*ValueError*，或在文件对象已被注册时引发*KeyError*。

abstractmethod unregister(*fileobj*)
注销对一个文件对象的选择，移除对它的监视。在文件对象被关闭之前应当先将其注销。

fileobj 必须是之前已注册的文件对象。

这将返回已关联的*SelectorKey* 实例，或者如果 *fileobj* 未注册则会引发*KeyError*。It will raise *ValueError* 如果 *fileobj* 无效（例如它没有 `fileno()` 方法或其 `fileno()` 方法返回无效值）。

modify(*fileobj*, *events*, *data=None*)
更改已注册文件对象所监视的事件或所附带的数据。

这等价于 `BaseSelector.unregister(fileobj)()` 加 `BaseSelector.register(fileobj, events, data)()`，区别在于它可以被更高效地实现。

这将返回一个新的*SelectorKey* 实例，或在出现无效事件掩码或文件描述符时引发*ValueError*，或在文件对象未被注册时引发*KeyError*。

abstractmethod select(*timeout=None*)
等待直到有已注册的文件对象就绪，或是超过时限。

如果 `timeout > 0`，这指定以秒数表示的最大等待时间。如果 `timeout <= 0`，调用将不会阻塞，并将报告当前就绪的文件对象。如果 *timeout* 为 `None`，调用将阻塞直到某个被监视的文件对象就绪。

这将返回由 `(key, events)` 元组构成的列表，每项各表示一个就绪的文件对象。

key 是对应于就绪文件对象的*SelectorKey* 实例。*events* 是在此文件对象上等待的事件位掩码。

備🄵: 如果当前进程收到一个信号，此方法可在任何文件对象就绪之前或超出时限时返回：在此情况下，将返回一个空列表。

3.5 版更變: 现在当被某个信号中断时，如果信号处理程序没有引发异常，选择器会用重新计算的超时值进行重试（请查看 **PEP 475** 其理由），而不是在超时之前返回空的事件列表。

close()
关闭选择器。

必须调用这个方法以确保下层资源会被释放。选择器被关闭后将不可再使用。

get_key(*fileobj*)
返回关联到某个已注册文件对象的键。

此方法将返回关联到文件对象的*SelectorKey* 实例，或在文件对象未注册时引发*KeyError*。

abstractmethod get_map()
返回从文件对象到选择器键的映射。

这将返回一个将已注册文件对象映射到与其相关联的*SelectorKey* 实例的*Mapping* 实例。

class selectors.**DefaultSelector**
默认的选择器类，使用当前平台上可用的最高效实现。这应为大多数用户的默认选择。

class selectors.**SelectSelector**
基于*select.select()* 的选择器。

class selectors.**PollSelector**
基于*select.poll()* 的选择器。

class selectors.**EpollSelector**
基于*select.epoll()* 的选择器。

fileno()
此方法将返回由下层*select.epoll()* 对象所使用的文件描述符。

class selectors.**DevpollSelector**
基于*select.devpoll()* 的选择器。

fileno()
此方法将返回由下层*select.devpoll()* 对象所使用的文件描述符。

3.5 版新加入.

class selectors.**KqueueSelector**
基于*select.kqueue()* 的选择器。

fileno()
此方法将返回由下层*select.kqueue()* 对象所使用的文件描述符。

18.5.3 示例

下面是一个简单的回显服务器实现:

```
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)
```

(下页继续)

(繼續上一頁)

```
def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

18.6 signal --- 设置异步事件处理程序

该模块提供了在 Python 中使用信号处理程序的机制。

18.6.1 一般规则

signal.signal() 函数允许定义在接收到信号时执行的自定义处理程序。少量的默认处理程序已经设置：*SIGPIPE* 被忽略（因此管道和套接字上的写入错误可以报告为普通的 Python 异常）以及如果父进程没有更改*SIGINT*，则其会被翻译成*KeyboardInterrupt* 异常。

一旦设置，特定信号的处理程序将保持安装，直到它被显式重置（Python 模拟 BSD 样式接口而不管底层实现），但*SIGCHLD* 的处理程序除外，它遵循底层实现。

执行 Python 信号处理程序

Python 信号处理程序不会在低级（C ）信号处理程序中执行。相反，低级信号处理程序设置一个标志，告诉*virtual machine* 稍后执行相应的 Python 信号处理程序（例如在下一个*bytecode* 指令）。这会导致：

- 捕获同步错误是没有意义的，例如*SIGFPE* 或*SIGSEGV*，它们是由 C 代码中的无效操作引起的。Python 将从信号处理程序返回到 C 代码，这可能会再次引发相同的信号，导致 Python 显然的挂起。从 Python 3.3 开始，你可以使用*faulthandler* 模块来报告同步错误。
- 纯 C 中实现的长时间运行的计算（例如在大量文本上的正则表达式匹配）可以在任意时间内不间断地运行，而不管接收到任何信号。计算完成后将调用 Python 信号处理程序。
- If the handler raises an exception, it will be raised ”out of thin air” in the main thread. See the *note below* for a discussion.

信号与线程

Python 信号处理程序总是会在主 Python 主解释器的主线程中执行，即使信号是在另一个线程中接收的。这意味着信号不能被用作线程间通信的手段。你可以改用*threading* 模块中的同步原语。

此外，只有主解释器的主线程才被允许设置新的信号处理程序。

18.6.2 模块内容

3.5 版更變: 信号（SIG* ），处理程序（*SIG_DFL* ，*SIG_IGN*）和 sigmask（*SIG_BLOCK* ，*SIG_UNBLOCK* ，*SIG_SETMASK* ）下面列出的相关常量变成了*enums* 。*getsignal()* ，*pthread_sigmask()* ，*sigpending()* 和*sigwait()* 函数返回人类可读的*enums* 。

在*signal* 模块中定义的变量是：

signal.**SIG_DFL**
这是两种标准信号处理选项之一；它只会执行信号的默认函数。例如，在大多数系统上，对于 SIGQUIT 的默认操作是转储核心并退出，而对于*SIGCHLD* 的默认操作是简单地忽略它。

signal.**SIG_IGN**
这是另一个标准信号处理程序，它将简单地忽略给定的信号。

signal.**SIGABRT**
来自 *abort(3)* 的中止信号。

signal.**SIGALRM**
来自 *alarm(2)* 的计时器信号。

可用性: Unix。

signal.**SIGBREAK**
来自键盘的中断 (CTRL + BREAK)。

可用性: Windows。

signal.**SIGBUS**
总线错误 (非法的内存访问)。

可用性: Unix。

signal.**SIGCHLD**
子进程被停止或终结。

可用性: Unix。

signal.**SIGCLD**
SIGCHLD 的别名。

signal.**SIGCONT**
如果进程当前已停止则继续执行它

可用性: Unix。

signal.**SIGFPE**
浮点异常。例如除以零。

也参考:

当除法或求余运算的第二个参数为零时会引发*ZeroDivisionError* 。

signal.**SIGHUP**
在控制终端上检测到挂起或控制进程的终止。

可用性: Unix。

signal.**SIGILL**
非法指令。

signal.**SIGINT**
来自键盘的中断 (CTRL + C)。

默认的动作是引发*KeyboardInterrupt*。

signal.**SIGKILL**
终止信号。

它不能被捕获、阻塞或忽略。

可用性: Unix。

signal.**SIGPIPE**
损坏的管道：写入到没有读取器的管道。

默认的动作是忽略此信号。

可用性: Unix。

signal.**SIGSEGV**
段错误：无效的内存引用。

signal.**SIGTERM**
终结信号。

signal.**SIGUSR1**
用户自定义信号 1。

可用性: Unix。

signal.**SIGUSR2**
用户自定义信号 2。

可用性: Unix。

signal.**SIGWINCH**
窗口调整大小信号。

可用性: Unix。

SIG*
所有信号编号都是符号化定义的。例如，挂起信号被定义为*signal.SIGHUP*；变量的名称与 C 程序中使用的名称相同，具体见 <signal.h>。'signal()' 的 Unix 手册页面列出了现有的信号 (在某些系统上这是 *signal(2)*，在其他系统中此列表则是在 *signal(7)* 中)。请注意并非所有系统都会定义相同的信号名称集；只有系统所定义的名称才会由此模块来定义。

signal.**CTRL_C_EVENT**
对应于 Ctrl+C 击键事件的信号。此信号只能用于*os.kill()* 。

可用性: Windows。

3.2 版新加入.

signal.**CTRL_BREAK_EVENT**
对应于 Ctrl+Break 击键事件的信号。此信号只能用于*os.kill()* 。

可用性: Windows。

3.2 版新加入.

signal.**NSIG**
比最高信号数多一。

signal.**ITIMER_REAL**
实时递减间隔计时器，并在到期时发送*SIGALRM* 。

signal.**ITIMER_VIRTUAL**
仅在进程执行时递减间隔计时器，并在到期时发送 SIGVTALRM 。

signal.**ITIMER_PROF**
当进程执行时以及当系统替进程执行时都会减小间隔计时器。这个计时器与 ITIMER_VIRTUAL 相配结，通常被用于分析应用程序在用户和内核空间中花费的时间。SIGPROF 会在超期时被发送。

signal.**SIG_BLOCK**
pthread_sigmask() 的 *how* 形参的一个可能的值，表明信号将会被阻塞。

3.3 版新加入.

signal.**SIG_UNBLOCK**
pthread_sigmask() 的 *how* 形参的是个可能的值，表明信号将被解除阻塞。

3.3 版新加入.

signal.**SIG_SETMASK**
pthread_sigmask() 的 *how* 形参的一个可能的值，表明信号掩码将要被替换。

3.3 版新加入.

signal 模块定义了一个异常:

exception signal.**ItimerError**
作为来自下层*setitimer()* 或*getitimer()* 实现错误的信号被引发。如果将无效的定时器或负的时间值传给*setitimer()* 就导致这个错误。此错误是*OSError* 的子类型。

3.3 版新加入: 此错误是*IOError* 的子类型，现在则是*OSError* 的别名。

signal 模块定义了以下函数:

signal.**alarm**(*time*)
如果 *time* 值非零，则此函数将要求将一个*SIGALRM* 信号在 *time* 秒内发往进程。任何在之前排入计划的警报都会被取消（在任何时刻都只能有一个警报被排入计划）。后续的返回值将是任何之前设置的警报被传入之前的秒数。如果 *time* 值为零，则不会将任何警报排入计划，并且任何已排入计划的警报都会被取消。如果返回值为零，则目前没有任何警报被排入计划。

可用性: Unix。更多信息请参见手册页面 *alarm(2)*。

signal.**getsignal**(*signalnum*)
返回当前用于信号 *signalnum* 的信号处理程序。返回值可以是一个 Python 可调用对象，或是特殊值*signal.SIG_IGN*, *signal.SIG_DFL* 或*None* 之一。在这里，*signal.SIG_IGN* 表示信号在之前被忽略，*signal.SIG_DFL* 表示之前在使用默认的信号处理方式，而 None 表示之前的信号处理程序未由 Python 安装。

signal.**strsignal**(*signalnum*)
返回信号 *signalnum* 的系统描述，例如"Interrupt", "Segmentation fault" 等等。如果信号无法被识别则返回*None*。

3.8 版新加入.

signal.**valid_signals**()
返回本平台上的有效信号编号集。这可能会少于 range(1, NSIG)，如果某些信号被系统保留作为内部使用的话。

3.8 版新加入.

signal.**pause**()
使进程休眠直至接收到一个信号；然后将会调用适当的处理程序。返回空值。

可用性: Unix。更多信息请参见手册页面 *signal(2)*。

另请参阅*sigwait()*, *sigwaitinfo()*, *sigtimedwait()* 和*sigpending()*。

signal.**raise_signal**(*signum*)
向调用方进程发送一个信号。返回空值。

3.8 版新加入.

signal.**pidfd_send_signal**(*pidfd*, *sig*, *siginfo=None*, *flags=0*)
发送信号 *sig* 到文件描述符 *pidfd* 所指向的进程。Python 目前不支持 *siginfo* 形参；它必须为 None。提供 *flags* 参数是为了将来扩展；当前未定义旗标值。

更多信息请参阅 *pidfd_send_signal(2)* 手册页面。

可用性: Linux 5.1+

3.9 版新加入.

signal.**pthread_kill**(*thread_id*, *signalnum*)
将信号 *signalnum* 发送至与调用者在同一进程中另一线程 *thread_id*。目标线程可被用于执行任何代码（Python 或其它）。但是，如果目标线程是在执行 Python 解释器，则 Python 信号处理程序将由主解释器的主线程来执行。因此，将信号发送给特定 Python 线程的唯一作用在于强制让一个正在运行的系统调用失败并抛出*InterruptedError*。

使用*threading.get_ident()* 或*threading.Thread* 对象的*ident* 属性为 *thread_id* 获取合适的值。

如果 *signalnum* 为 0，则不会发送信号，但仍然会执行错误检测；这可被用来检测目标线程是否仍在运行。

引发一个审计事件 signal.pthread_kill，附带参数 thread_id, signalnum。

可用性: Unix。更多信息请参见手册页面 *pthread_kill(3)*。

另请参阅*os.kill()*。

3.3 版新加入.

signal.**pthread_sigmask**(*how*, *mask*)
获取和/或修改调用方线程的信号掩码。信号掩码是一组传送过程目前为调用者而阻塞的信号集。返回旧的信号掩码作为一组信号。

该调用的行为取决于 *how* 的值，具体见下。

- *SIG_BLOCK*: 被阻塞信号集是当前集与 *mask* 参数的并集。
- *SIG_UNBLOCK*: *mask* 中的信号会从当前已阻塞信号集中被移除。允许尝试取消对一个非阻塞信号的阻塞。
- *SIG_SETMASK*: 已阻塞信号集会被设为 *mask* 参数的值。

mask 是一个信号编号集合(例如{*signal.SIGINT*, *signal.SIGTERM*})。请使用*valid_signals()* 表示包含所有信号的完全掩码。

例如，signal.pthread_sigmask(signal.SIG_BLOCK, []) 会读取调用方线程的信号掩码。

SIGKILL 和 SIGSTOP 不能被阻塞。

可用性: Unix。更多信息请参见手册页面 *sigprocmask(3)* 和 *pthread_sigmask(3)*。

另请参阅*pause()*, *sigpending()* 和*sigwait()*。

3.3 版新加入.

signal.**setitimer**(*which*, *seconds*, *interval=0.0*)
设置由 *which* 指明的给定间隔计时器 (*signal.ITIMER_REAL*, *signal.ITIMER_VIRTUAL* 或 *signal.ITIMER_PROF* 之一) 在 *seconds* 秒 (接受浮点数值，为与 *alarm()* 之差) 之后开始并在每 *interval* 秒间隔时 (如果 *interval* 不为零) 启动。由 *which* 指明的间隔计时器可通过将 *seconds* 设为零来清空。

当一个间隔计时器启动时，会有信号发送至进程。所发送的具体信号取决于所使用的计时器；*signal.ITIMER_REAL* 将发送 *SIGALRM*, *signal.ITIMER_VIRTUAL* 将发送 SIGVTALRM, 而 *signal.ITIMER_PROF* 将发送 SIGPROF.

原有的值会以元组: (delay, interval) 的形式被返回。

尝试传入无效的计时器将导致 *ItimerError*。

可用性: Unix。

signal.**getitimer**(*which*)
返回由 *which* 指明的给定间隔计时器当前的值。

可用性: Unix。

signal.**set_wakeup_fd**(*fd*, *, *warn_on_full_buffer=True*)
将唤醒文件描述符设为 *fd*。当接收到信号时，会将信号编号以单个字节的形式写入 fd。这可被其它库用来唤醒一次 poll 或 select 调用，以允许该信号被完全地处理。

原有的唤醒 fd 会被返回（或者如果未启用文件描述符唤醒则返回 -1）。如果 *fd* 为 -1，文件描述符唤醒会被禁用。如果不为 -1，则 *fd* 必须为非阻塞型。需要由库来负责在重新调用 poll 或 select 之前从 *fd* 移除任何字节数据。

当启用线程用时，此函数只能从主解释器的主线程 被调用；尝试从另一线程调用它将导致 *ValueError* 异常被引发。

使用此函数有两种通常的方式。在两种方式下，当有信号到达时你都是用 fd 来唤醒，但之后它们在确定达到的一个或多个信号 *which* 时存在差异。

在第一种方式下，我们从 fd 的缓冲区读取数据，这些字节值会给你信号编号。这种方式很简单，但在少数情况下会发生问题：通常 fd 将有缓冲区空间大小限制，如果信号到达得太多且太快，缓冲区可能会爆满，有些信号可能丢失。如果你使用此方式，则你应当设置 warn_on_full_buffer=True，当信号丢失时这至少能将警告消息打印到 stderr。

在第二种方式下，我们 只会将唤醒 fd 用于唤醒，而忽略实际的字节值。在此情况下，我们所关心的只有 fd 的缓冲区为空还是不为空；爆满的缓冲区完全不会导致问题。如果你使用此方式，则你应当设置 warn_on_full_buffer=False，这样你的用户就不会被虚假的警告消息所迷惑。

3.5 版更變: 在 Windows 上，此函数现在也支持套接字处理。

3.7 版更變: 添加了 warn_on_full_buffer 形参。

signal.**siginterrupt**(*signalnum*, *flag*)
更改系统调用重启行为：如果 *flag* 为 *False*，系统调用将在被信号 *signalnum* 中断时重启，否则系统调用将被中断。返回空值。

可用性: Unix。更多信息请参见手册页面 *siginterrupt(3)*。

请注意用 *signal()* 安装信号处理程序将重启行为重置为可通过显式调用 siginterrupt() 并为给定信号的 *flag* 设置真值来实现中断。

signal.**signal**(*signalnum*, *handler*)
将信号 *signalnum* 的处理程序设为函数 *handler*。*handler* 可以为接受两个参数（见下）的 Python 可调用对象，或者为特殊值 *signal.SIG_IGN* 或 *signal.SIG_DFL* 之一。之前的信号处理程序将被返回（参见上文 *getsignal()* 的描述）。（更多信息请参阅 Unix 手册页面 *signal(2)*。）

当启用线程用时，此函数只能从主解释器的主线程被调用；尝试从另一线程调用它将导致*ValueError*异常被引发。

handler 将附带两个参数调用：信号编号和当前堆栈帧 (None 或一个帧对象；有关帧对象的描述请参阅类型层级结构描述或者参阅*inspect* 模块中的属性描述)。

在 Windows 上，*signal()* 调用只能附带*SIGABRT*, *SIGFPE*, *SIGILL*, *SIGINT*, *SIGSEGV*, *SIGTERM* 或*SIGBREAK*。任何其他值都将引发*ValueError*。请注意不是所有系统都定义了同样的信号名称集合；如果一个信号名称未被定义为 SIG* 模块层级常量则将引发*AttributeError*。

signal.**sigpending**()
检查正在等待传送给调用方线程的信号集合（即在阻塞期间被引发的信号）。返回正在等待的信号集合。

可用性: Unix。更多信息请参见手册页面 *sigpending(2)*。

另请参阅*pause()*, *pthread_sigmask()* 和*sigwait()*。

3.3 版新加入.

signal.**sigwait**(*sigset*)
挂起调用方线程的执行直到信号集合 *sigset* 中指定的信号之一被传送。此函数会接受该信号（将其从等待信号列表中移除），并返回信号编号。

可用性: Unix。更多信息请参见手册页面 *sigwait(3)*。

另请参阅*pause()*, *pthread_sigmask()*, *sigpending()*, *sigwaitinfo()* 和*sigtimedwait()*。

3.3 版新加入.

signal.**sigwaitinfo**(*sigset*)
挂起调用方线程的执行直到信号集合 *sigset* 中指定的信号之一被传送。此函数会接受该信号并将其从等待信号列表中移除。如果 *sigset* 中的信号之一已经在等待调用方线程，此函数将立即返回并附带有关该信号的信息。被传送信号的信号处理程序不会被调用。如果该函数被某个不在 *sigset* 中的信号中断则会引发*InterruptedError*。

返回值是一个代表 siginfo_t 结构体所包含数据的对象，具体为: si_signo, si_code, si_errno, si_pid, si_uid, si_status, si_band。

可用性: Unix。更多信息请参见手册页面 *sigwaitinfo(2)*。

另请参阅*pause()*, *sigwait()* 和*sigtimedwait()*。

3.3 版新加入.

3.5 版更變: 当被某个不在 *sigset* 中的信号中断时本函数将进行重试并且信号处理程序不会引发异常（请参阅 **PEP 475** 了解其理由）。

signal.**sigtimedwait**(*sigset*, *timeout*)
类似于*sigwaitinfo()*，但会接受一个额外的 *timeout* 参数来指定超时限制。如果将 *timeout* 指定为 0，则会执行轮询。如果发生超时则返回*None*。

可用性: Unix。更多信息请参见手册页面 *sigtimedwait(2)*。

另请参阅*pause()*, *sigwait()* 和*sigwaitinfo()*。

3.3 版新加入.

3.5 版更變: 现在当此函数被某个不在 *sigset* 中的信号中断时将以计算出的 *timeout* 进行重试并且信号处理程序不会引发异常（请参阅 **PEP 475** 了解其理由）。

18.6.3 示例

这是一个最小示例程序。它使用*alarm()* 函数来限制等待打开一个文件所花费的时间；这在文件为无法开启的串行设备时会很有用处，此情况通常会导致*os.open()* 无限期地挂起。解决办法是在打开文件之前设置 5 秒钟的 alarm；如果操作耗时过长，将会发送 alarm 信号，并且处理程序会引发一个异常。

```
import signal, os

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    raise OSError("Couldn't open device!")

# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```

18.6.4 对于 SIGPIPE 的说明

将你的程序用管道输出到工具例如 *head(1)* 将会导致*SIGPIPE* 信号在其标准输出的接收方提前关闭时被发送到你的进程。这将引发一个异常例如 `BrokenPipeError: [Errno 32] Broken pipe`。要处理这种情况，请对你的入口点进行包装以捕获此异常，如下所示:

```
import os
import sys

def main():
    try:
        # simulate large output (your code replaces this loop)
        for x in range(10000):
            print("y")
        # flush output here to force SIGPIPE to be triggered
        # while inside this try block.
        sys.stdout.flush()
    except BrokenPipeError:
        # Python flushes standard streams on exit; redirect remaining output
        # to devnull to avoid another BrokenPipeError at shutdown
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python exits with error code 1 on EPIPE

if __name__ == '__main__':
    main()
```

Do not set *SIGPIPE*'s disposition to *SIG_DFL* in order to avoid *BrokenPipeError*. Doing that would cause your program to exit unexpectedly whenever any socket connection is interrupted while your program is still writing to it.

18.6.5 Note on Signal Handlers and Exceptions

If a signal handler raises an exception, the exception will be propagated to the main thread and may be raised after any *bytecode* instruction. Most notably, a *KeyboardInterrupt* may appear at any point during execution. Most Python code, including the standard library, cannot be made robust against this, and so a *KeyboardInterrupt* (or any other exception resulting from a signal handler) may on rare occasions put the program in an unexpected state.

To illustrate this issue, consider the following code:

```
class SpamContext:
    def __init__(self):
        self.lock = threading.Lock()

    def __enter__(self):
        # If KeyboardInterrupt occurs here, everything is fine
        self.lock.acquire()
        # If KeyboardInterrupt occcurs here, __exit__ will not be called
        ...
        # KeyboardInterrupt could occur just before the function returns

    def __exit__(self, exc_type, exc_val, exc_tb):
        ...
        self.lock.release()
```

For many programs, especially those that merely want to exit on *KeyboardInterrupt*, this is not a problem, but applications that are complex or require high reliability should avoid raising exceptions from signal handlers. They should also avoid catching *KeyboardInterrupt* as a means of gracefully shutting down. Instead, they should install their own *SIGINT* handler. Below is an example of an HTTP server that avoids *KeyboardInterrupt*:

```
import signal
import socket
from selectors import DefaultSelector, EVENT_READ
from http.server import HTTPServer, SimpleHTTPRequestHandler

interrupt_read, interrupt_write = socket.socketpair()

def handler(signum, frame):
    print('Signal handler called with signal', signum)
    interrupt_write.send(b'\0')
signal.signal(signal.SIGINT, handler)

def serve_forever(httpd):
    sel = DefaultSelector()
    sel.register(interrupt_read, EVENT_READ)
    sel.register(httpd, EVENT_READ)

    while True:
        for key, _ in sel.select():
            if key.fileobj == interrupt_read:
                interrupt_read.recv(1)
                return
            if key.fileobj == httpd:
                httpd.handle_request()

print("Serving on port 8000")
httpd = HTTPServer(('', 8000), SimpleHTTPRequestHandler)
serve_forever(httpd)
print("Shutdown...")
```

18.7 mmap --- 内存映射文件支持

内存映射文件对象的行为既像*bytearray* 又像文件对象。你可以在大部分接受*bytearray* 的地方使用 mmap 对象；例如，你可以使用*re* 模块来搜索一个内存映射文件。你也可以通过执行 `obj[index] = 97` 来修改单个字节，或者通过对切片赋值来修改一个子序列: `obj[i1:i2] = b'...'`。你还可以在文件的当前位置开始读取和写入数据，并使用 `seek()` 前往另一个位置。

内存映射文件是由*mmap* 构造函数创建的，其在 Unix 和 Windows 上是不同的。无论哪种情况，你都必须为一个打开的文件提供文件描述符以进行更新。如果你希望映射一个已有的 Python 文件对象，请使用该对象的 `fileno()` 方法来获取 *fileno* 参数的正确值。否则，你可以使用*os.open()* 函数来打开这个文件，这会直接返回一个文件描述符（结束时仍然需要关闭该文件）。

備🄴: 如果要为可写的缓冲文件创建内存映射，则应当首先*flush()* 该文件。这确保了对缓冲区的本地修改在内存映射中可用。

对于 Unix 和 Windows 版本的构造函数，可以将 *access* 指定为可选的关键字参数。*access* 接受以下四个值之一: `ACCESS_READ` ，`ACCESS_WRITE` 或 `ACCESS_COPY` 分别指定只读，直写或写时复制内存，或 `ACCESS_DEFAULT` 推迟到 *prot* 。*access* 可以在 Unix 和 Windows 上使用。如果未指定 *access* ，则 Windows mmap 返回直写映射。这三种访问类型的初始内存值均取自指定的文件。向 `ACCESS_READ` 内存映射赋值会引发*TypeError* 异常。向 `ACCESS_WRITE` 内存映射赋值会影响内存和底层的文件。向 `ACCESS_COPY` 内存映射赋值会影响内存，但不会更新底层的文件。

3.7 版更變: 添加了 `ACCESS_DEFAULT` 常量。

要映射匿名内存，应将 -1 作为 fileno 和 length 一起传递。

class mmap.**mmap**(*fileno*, *length*, *tagname=None*, *access=ACCESS_DEFAULT*[, *offset*])

(Windows 版本) 映射被文件句柄 *fileno* 指定的文件的 *length* 个字节，并创建一个 mmap 对象。如果 *length* 大于当前文件大小，则文件将扩展为包含 *length* 个字节。如果 *length* 为 `0`，则映射的最大长度为当前文件大小。如果文件为空，Windows 会引发异常（你无法在 Windows 上创建空映射）。

如果 *tagname* 被指定且不是 `None` ，则是为映射提供标签名称的字符串。Windows 允许你对同一文件拥有许多不同的映射。如果指定现有标签的名称，则会打开该标签，否则将创建该名称的新标签。如果省略此参数或设置为 `None` ，则创建的映射不带名称。避免使用 tag 参数将有助于使代码在 Unix 和 Windows 之间可移植。

offset 可以被指定为非负整数偏移量。mmap 引用将相对于从文件开头的偏移。*offset* 默认为 0。*offeset* 必须是 `ALLOCATIONGRANULARITY` 的倍数。

引发一个审计事件 `mmap.__new__` 附带参数 `fileno`, `length`, `access`, `offset`。

class mmap.**mmap**(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE|PROT_READ*, *access=ACCESS_DEFAULT*[, *offset*])

(Unix 版本) 映射文件描述符 *fileno* 指定的文件的 *length* 个字节，并返回一个 mmap 对象。如果 *length* 为 `0` ，则当调用*mmap* 时，映射的最大长度将为文件的当前大小。

flags 指明映射的性质。`MAP_PRIVATE` 会创建私有的写入时拷贝映射，因此对 mmap 对象内容的修改将为该进程所私有，而 `MAP_SHARED` 会创建与其他映射同一文件区域的进程所共享的映射。默认值为 `MAP_SHARED`。

如果指明了 *prot*，它将给出所需的内存保护方式；最有用的两个值是 `PROT_READ` 和 `PROT_WRITE`，分别指明页面为可读或可写。*prot* 默认为 `PROT_READ | PROT_WRITE`。

可以指定 *access* 作为替代 *flags* 和 *prot* 的可选关键字形参。同时指定 *flags*, *prot* 和 *access* 将导致错误。请参阅上文中 *access* 的描述了解有关如何使用此形参的信息。

offset 可以被指定为非负整数偏移量。mmap 引用将相对于从文件开头的偏移。*offset* 默认为 0。*offset* 必须是 ALLOCATIONGRANULARITY 的倍数，它在 Unix 系统上等价于 PAGESIZE。

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with physical backing store on macOS and OpenVMS.

这个例子演示了使用*mmap* 的简单方式:

```
import mmap

# write a simple example file
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    mm.close()
```

mmap 也可以在 with 语句中被用作上下文管理器:

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

3.2 版新加入: 上下文管理器支持。

下面的例子演示了如何创建一个匿名映射并在父进程和子进程之间交换数据。:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

引发一个审计事件 mmap.__new__ 附带参数 fileno, length, access, offset。

映射内存的文件对象支持以下方法:

close()

关闭 mmap。后续调用该对象的其他方法将导致引发 ValueError 异常。此方法将不会关闭打开的文件。

closed
如果文件已关闭则返回 `True`。

3.2 版新加入.

find(*sub*[, *start*[, *end*]])
返回子序列 *sub* 在对象内被找到的最小索引号，使得 *sub* 被包含在 [*start*, *end*] 范围中。可选参数 *start* 和 *end* 会被解读为切片表示法。如果未找到则返回 `-1`。

3.5 版更變: 现在支持可写的字节类对象。

flush([*offset*[, *size*]])
将对文件的内存副本的修改刷新至磁盘。如果不使用此调用则无法保证在对象被销毁前将修改写回存储。如果指定了 *offset* 和 *size*，则只将对指定范围内字节的修改刷新至磁盘；在其他情况下，映射的全部范围都会被刷新。*offset* 必须为 `PAGESIZE` 或 `ALLOCATIONGRANULARITY` 的倍数。

返回 `None` 以表示成功。当调用失败时将引发异常。

3.8 版更變: 在之前版本中，成功时将返回非零值；在 Windows 下当发生错误时将返回零。在 Unix 下成功时将返回零值；当发生错误时将引发异常。

madvise(*option*[, *start*[, *length*]])
将有关内存区域的建议 *option* 发送至内核，从 *start* 开始扩展 *length* 个字节。*option* 必须为系统中可用的*MADV_* 常量* 之一。如果省略 *start* 和 *length*，则会包含整个映射。在某些系统中（包括 Linux），*start* 必须为 `PAGESIZE` 的倍数。

可用性: 具有 `madvise()` 系统调用的系统。

3.8 版新加入.

move(*dest*, *src*, *count*)
将从偏移量 *src* 开始的 *count* 个字节拷贝到目标索引号 *dest*。如果 mmap 创建时设置了 `ACCESS_READ`，则调用 move 将引发*TypeError* 异常。

read([*n*])
返回一个*bytes*，其中包含从当前文件位置开始的至多 *n* 个字节。如果参数省略，为 `None` 或负数，则返回从当前文件位置开始直至映射结尾的所有字节。文件位置会被更新为返回字节数据之后的位置。

3.3 版更變: 参数可被省略或为 `None`。

read_byte()
将当前文件位置上的一个字节以整数形式返回，并让文件位置前进 1。

readline()
返回一个单独的行，从当前文件位置开始直到下一个换行符。文件位置会被更新为返回字节数据之后的位置。

resize(*newsize*)
改变映射以及下层文件的大小，如果存在的话。如果 mmap 创建时设置了 `ACCESS_READ` 或 `ACCESS_COPY`，则改变映射大小将引发*TypeError* 异常。

rfind(*sub*[, *start*[, *end*]])
返回子序列 *sub* 在对象内被找到的最大索引号，使得 *sub* 被包含在 [*start*, *end*] 范围中。可选参数 *start* 和 *end* 会被解读为切片表示法。如果未找到则返回 `-1`。

3.5 版更變: 现在支持可写的字节类对象。

seek(*pos*[, *whence*])
设置文件的当前位置。*whence* 参数为可选项并且默认为 os.SEEK_SET 或 0 (绝对文件定位)；其他值还有 os.SEEK_CUR 或 1 (相对当前位置查找) 和 os.SEEK_END 或 2 (相对文件末尾查找)。

size()
返回文件的长度，该数值可以大于内存映射区域的大小。

tell()
返回文件指针的当前位置。

write(*bytes*)
将 *bytes* 中的字节数据写入文件指针当前位置的内存并返回写入的字节总数 (一定不小于 len(bytes)，因为如果写入失败，将会引发*ValueError*)。在字节数据被写入后文件位置将会更新。如果 mmap 创建时设置了 ACCESS_READ，则向其写入将引发*TypeError* 异常。

3.5 版更變: 现在支持可写的字节类对象。

3.6 版更變: 现在会返回写入的字节总数。

write_byte(*byte*)
将整数值 *byte* 写入文件指针当前位置的内存；文件位置前进 1。如果 mmap 创建时设置了 ACCESS_READ，则向其写入将引发*TypeError* 异常。

18.7.1 MADV_* 常量

mmap.**MADV_NORMAL**
mmap.**MADV_RANDOM**
mmap.**MADV_SEQUENTIAL**
mmap.**MADV_WILLNEED**
mmap.**MADV_DONTNEED**
mmap.**MADV_REMOVE**
mmap.**MADV_DONTFORK**
mmap.**MADV_DOFORK**
mmap.**MADV_HWPOISON**
mmap.**MADV_MERGEABLE**
mmap.**MADV_UNMERGEABLE**
mmap.**MADV_SOFT_OFFLINE**
mmap.**MADV_HUGEPAGE**
mmap.**MADV_NOHUGEPAGE**
mmap.**MADV_DONTDUMP**
mmap.**MADV_DODUMP**
mmap.**MADV_FREE**
mmap.**MADV_NOSYNC**
mmap.**MADV_AUTOSYNC**
mmap.**MADV_NOCORE**
mmap.**MADV_CORE**
mmap.**MADV_PROTECT**
这些选项可被传给*mmap.madvise()*。不是每个选项都存在于每个系统中。

可用性: 具有 madvise() 系统调用的系统。

3.8 版新加入.

CHAPTER 19

互联网数据处理

本章介绍了支持处理互联网上常用数据格式的模块。

19.1 email --- 电子邮件与 MIME 处理包

源代码: Lib/email/__init__.py

---

email 包是一个用于管理电子邮件消息的库。它 并非被设计为执行向 SMTP (**RFC 2821**), NNTP 或其他服务器发送电子邮件消息的操作；这些是*smtplib* 和*nntplib* 等模块的功能。*email* 包试图尽可能地遵循 RFC，支持 **RFC 5322** 和 **RFC 6532**，以及与 MIME 相关的各个 RFC 例如 **RFC 2045**, **RFC 2046**, **RFC 2047**, **RFC 2183** 和 **RFC 2231**。

email 包的总体结构可以分为三个主要组件，另外还有第四个组件用于控制其他组件的行为。

这个包的中心组件是代表电子邮件消息的 “对象模型”。应用程序主要通过在*message* 子模块中定义的对象模型接口与这个包进行交互。应用程序可以使用此 API 来询问有关现有电子邮件的问题、构造新的电子邮件，或者添加或移除自身也使用相同对象模型接口的电子邮件子组件。也就是说，遵循电子邮件消息及其 MIME 子组件的性质，电子邮件对象模型是所有提供*EmailMessage* API 的对象所构成的树状结构。

这个包的另外两个主要组件是*parser* 和*generator*。parser 接受电子邮件消息的序列化版本（字节流）并将其转换为*EmailMessage* 对象树。generator 接受*EmailMessage* 并将其转回序列化的字节流。(parser 和 generator 还能处理文本字符流，但不建议这种用法，因为这很容易导致某种形式的无效消息。

控制组件是*policy* 模块。每一个*EmailMessage*、每一个*generator* 和每一个*parser* 都有一个相关联的*policy* 对象来控制其行为。通常应用程序只有在*EmailMessage* 被创建时才需要指明控制策略，或者通过直接实例代*EmailMessage* 来新建电子邮件，或者通过使用*parser* 来解析输入流。但是策略也可以在使用*generator* 序列化消息时被更改。例如，这允许从磁盘解析通用电子邮件消息，而在将消息发送到电子邮件服务器时使用标准 SMTP 设置对其进行序列化。

email 包会尽量地对应用程序隐藏各种控制类 RFC 的细节。从概念上讲应用程序应当能够将电子邮件消息视为 Unicode 文本和二进制附件的结构化树，而不必担心在序列化时要如何表示它们。但在实际中，经常有必要至少了解一部分控制类 MIME 消息及其结构的规划，特别是 MIME ” 内容类型” 的名称和性质以及它们是如何标识多部分文档的。在大多数情况下这些知识应当仅对于更复杂的应用程序来说才是必需的，并且即便

在那时它也应当仅是特定的高层级结构，而不是如何表示这些结构的细节信息。由于 MIME 内容类型被广泛应用于现代因特网软件（而非只是电子邮件），因此这对许多程序员来说将是很熟悉的概念。

以下小节描述了*email* 包的具体功能。我们会从*message* 对象模型开始，它是应用程序将要使用的主要接口，之后是*parser* 和*generator* 组件。然后我们会介绍*policy* 控制组件，它将完成对这个库的主要组件的处理。

接下来的三个小节会介绍这个包可能引发的异常以及*parser* 可能检测到的缺陷（即与 RFC 不相符）。然后我们会介绍*headerregistry* 和*contentmanager* 子组件，它们分别提供了用于更精细地操纵标题和载荷的工具。这两个组件除了包含使用与生成非简单消息的相关特性，还记录了它们的可扩展性 API，这将是高级应用程序所感兴趣的内容。

在此之后是一组使用之前小节所介绍的 API 的基本部分的示例。

前面的内容是 email 包的现代（对 Unicode 支持良好）API。从*Message* 类开始的其余小节则介绍了旧式*compat32* API，它会更直接地处理如何表示电子邮件消息的细节。*compat32* API 不会向应用程序隐藏 RFC 的相关细节，但对于需要进行此种层级操作的应用程序来说将是很有用的工具。此文档对于因向下兼容理由而仍然使用*compat32* API 的应用程序也是很适合的。

3.6 版更變: 文档经过重新组织和撰写以鼓励使用新的*EmailMessage*/*EmailPolicy* API。

email 包文档的内容:

19.1.1 email.message: 表示一封电子邮件信息

源代码: Lib/email/message.py

3.6 版新加入:[1]

位于*email* 包的中心的类就是*EmailMessage* 类。这个类导入自*email.message* 模块。它是*email* 对象模型的基类。*EmailMessage* 为设置和查询头字段内容、访问信息体的内容、以及创建和修改结构化信息提供了核心功能。

一份电子邮件信息由 *标头*和 *载荷*（又被称为 *内容*）组成。标头遵循 **RFC 5322** 或者 **RFC 6532** 风格的字段名和值，字段名和字段值之间由一个冒号隔开。这个冒号既不属于字段名，也不属于字段值。信息的载荷可能是一段简单的文字消息，也可能是一个二进制的对象，更可能是由多个拥有各自标头和载荷的子信息组成的结构化子信息序列。对于后者类型的载荷，信息的 MIME 类型将会被指明为诸如 *multipart/** 或 *message/rfc822* 的类型。

EmailMessage 对象所提供的抽象概念模型是一个头字段组成的有序字典加一个代表 **RFC 5322** 标准的信息体的 *载荷*。载荷有可能是一系列子 EmailMessage 对象的列表。你除了可以通过一般的字典方法来访问头字段名和值，还可以使用特制方法来访问头的特定字段（比如说 MIME 内容类型字段）、操纵载荷、生成信息的序列化版本、递归遍历对象树。

EmailMessage 的类字典接口的字典索引是头字段名，头字段名必须是 ASCII 值。字典值是带有一些附加方法的字符串。虽然头字段的存储和获取都是保留其原始大小写的，但是字段名的匹配是大小写不敏感的。与真正的字典不同，键与键之间不但存在顺序关系，还可以重复。我们提供了额外的方法来处理含有重复键的头。

*载荷*是多样的。对于简单的信息对象，它是字符串或字节对象；对于诸如 *multipart/** 和 *message/rfc822* 信息对象的 MIME 容器文档，它是一个*EmailMessage* 对象列表。

class email.message.**EmailMessage**(*policy=default*)

如果指定了 *policy*，消息将由这个 *policy* 所指定的规则来更新和序列化信息的表达。如果没有指定 *policy*，其将默认使用*default* 策略。这个策略遵循电子邮件的 RFC 标准，除了行终止符号（RFC 要求使用 \r\n，此策略使用 Python 标准的 \n 行终止符）。请前往*policy* 的文档获取更多信息。

[1] 原先在 3.4 版本中以*provisional module* 添加。过时的文档被移动至*email.message.Message: 使用 compat32 API 来表示电子邮件消息*。

as_string(*unixfrom=False*, *maxheaderlen=None*, *policy=None*)
以一段扁平的字符串的形式返回整个信息对象。若可选的 *unixform* 参数为真，返回的字符串会包含信封头。*unixform* 的默认值是 `False` 。为了保持与基类*Message* 的兼容性，*maxheaderlen* 是被接受的，但是其默认值是 `None` 。这个默认值表示行长度由策略的 `max_line_length` 属性所控制。从信息实例所获取到的策略可以通过 *policy* 参数重写。这样可以对该方法所产生的输出进行略微的控制，因为指定的 *policy* 会被传递到*Generator* 当中。

扁平化信息可能会对*EmailMessage* 做出修改。这是因为为了完成向字符串的转换，一些内容需要使用默认值填入（举个例子，MIME 边界字段可能会被生成或被修改）。

请注意，这个方法是为了便利而提供，不一定是适合你的应用程序的最理想的序列化信息的方法。这在你处理多封信息的时候尤甚。如果你需要使用更加灵活的 API 来序列化信息，请参见*email.generator.Generator* 。同时请注意，当*utf8* 属性为其默认值 `False` 的时候，本方法将限制其行为为生成以“7 bit clean”方式序列化的信息。

3.6 版更變: *maxheaderlen* 没有被指定时的默认行为从默认为 0 修改为默认为策略的 *max_line_length* 值。

__str__()
与 `as_string(policy=self.policy.clone(utf8=True))` 等价。这将让 `str(msg)` 产生的字符串包含人类可读的的序列化信息内容。

3.4 版更變: 本方法开始使用 `utf8=True` ，而非*as_string()* 的直接替身。使用 `utf8=True` 会产生类似于 **RFC 6531** 的信息表达。

as_bytes(*unixfrom=False*, *policy=None*)
以字节串对象的形式返回整个扁平化后的消息。当可选的 *unixfrom* 为真值时，返回的字符串会包含信封标头。*unixfrom* 的默认值为 `False`。*policy* 参数可被用于重载从消息实例获取的默认 policy。这可被用来控制该方法所产生的部分格式效果，因为指定的 *policy* 将被传递给*BytesGenerator*。

扁平化信息可能会对*EmailMessage* 做出修改。这是因为为了完成向字符串的转换，一些内容需要使用默认值填入（举个例子，MIME 边界字段可能会被生成或被修改）。

请注意，这个方法是为了便利而提供，不一定是适合你的应用程序的最理想的序列化信息的方法。这在你处理多封信息的时候尤甚。如果你需要使用更加灵活的 API 来序列化信息，请参见*email.generator.BytesGenerator* 。

__bytes__()
与*as_bytes()* 等价。这将让 `bytes(msg)` 产生一个包含序列化信息内容的字节序列对象。

is_multipart()
如果该信息的载荷是一个子*EmailMessage* 对象列表，返回 `True` ；否则返回 `False` 。在*is_multipart()* 返回 `True` 的场合下，载荷应当是一个字符串对象（有可能是一个使用了内容传输编码进行编码的二进制载荷）。请注意，*is_multipart()* 返回 `True` 不意味着 `msg.get_content_maintype() == 'multipart'` 也会返回 `True` 。举个例子，`is_multipart` 在*EmailMessage* 是 `message/rfc822` 类型的信息的情况下，其返回值也是 `True` 。

set_unixfrom(*unixfrom*)
将信息的信封头设置为 *unixform* ，这应当是一个字符串。（在*mboxMessage* 中有关于这个头的一段简短介绍。）

get_unixfrom()
返回消息的信封头。如果信封头从未被设置过，默认返回 `None` 。

以下方法实现了对信息的头字段进行访问的类映射接口。请留意，只是类映射接口，这与平常的映射接口（比如说字典映射）有一些语义上的不同。举个例子，在一个字典当中，键之间不可重复，但是信息头字段是可以重复的。不光如此，在字典当中调用*keys()* 方法返回的结果，其顺序没有保证；但是在一个*EmailMessage* 对象当中，返回的头字段永远以其在原信息当中出现的顺序，或以其加入信息的顺序为序。任何删了后又重新加回去的头字段总是添加在当时列表的末尾。

这些语义上的不同是刻意而为之的，是出于在绝大多数常见使用情景中都方便的初衷下设计的。

还请留意，无论在什么情况下，消息当中的任何信封头字段都不会包含在映射接口当中。

__len__()
返回头字段的总数，重复的也计算在内。

__contains__(*name*)
如果消息对象中有一个名为 *name* 的字段，其返回值为 `True` 。匹配无视大小写差异，*name* 也不包含末尾的的冒号。`in` 操作符的实现中用到了这个方法，比如说：

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)
返回头字段名对应的字段值。*name* 不含冒号分隔符。如果字段未找到，返回 `None` 。*KeyError* 异常永不抛出。

请注意，如果对应名字的字段找到了多个，具体返回哪个字段值是未定义的。请使用*get_all()* 方法获取当前匹配字段名的所有字段值。

使用标准策略（非 `compat32`）时，返回值是*email.headerregistry.BaseHeader* 的某个子类的一个实例。

__setitem__(*name*, *val*)
在信息头中添加名为 *name* 值为 *val* 的字段。这个字段会被添加在已有字段列表的结尾处。

请注意，这个方法 既不会覆盖 也不会删除任何字段名重名的已有字段。如果你确实想保证新字段是整个信息头当中唯一拥有 *name* 字段名的字段，你需要先把旧字段删除。例如：

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

如果 `policy` 明确要求某些字段是唯一的（至少标准策略就有这么做），对这些字段在已有同名字段的情况下仍然调用此方法尝试为字段名赋值会引发*ValueError* 异常。这是为了一致性而刻意设计出的行为，不过我们随时可能会突然觉得“还是在这种情况下自动把旧字段删除比较好吧”而把这个行为改掉，所以不要以为这是特性而依赖这个行为。

__delitem__(*name*)
删除信息头当中字段名匹配 *name* 的所有字段。如果匹配指定名称的字段没有找到，也不会抛出任何异常。

keys()
以列表形式返回消息头中所有的字段名。

values()
以列表形式返回消息头中所有的字段值。

items()
以二元元组的列表形式返回消息头中所有的字段名和字段值。

get(*name*, *failobj=None*)
返回对应字段名的字段值。这个方法与*__getitem__()* 是一样的，只不过如果对应字段名的字段没有找到，该方法会返回 *failobj* 。这个参数是可选的（默认值为 `None` ）。

以下是一些与头有关的更多有用方法：

get_all(*name*, *failobj=None*)
返回字段名为 *name* 的所有字段值的列表。如果信息内不存在匹配的字段，返回 *failobj*（其默认值为 `None` ）。

add_header(*_name*, *_value*, ***_params*)
高级头字段设定。这个方法与*__setitem__()* 类似，不过你可以使用关键字参数为字段提供附加参数。*_name* 是字段名，*_value* 是字段 主值。

对于关键字参数字典 *_params* 的每个键值对而言，它的键被用作参数的名字，其中下划线被替换为短横杠（毕竟短横杠不是合法的 Python 标识符）。一般来讲，参数以 键 =" 值" 的方式添加，除非值是 None 。要真的是这样的话，只有键会被添加。

如果值含有非 ASCII 字符，你可以将值写成 (CHARSET, LANGUAGE, VALUE) 形式的三元组，这样你可以人为控制字符的字符集和语言。CHARSET 是一个字符串，它为你的值的编码命名；LANGUAGE 一般可以直接设为 None ，也可以直接设为空字符串（其他可能取值参见:rfc‘2231‘）；VALUE 是一个字符串值，其包含非 ASCII 的码点。如果你没有使用三元组，你的字符串又含有非 ASCII 字符，那么它就会使用 **RFC 2231** 中，CHARSET 为 utf-8 ，LANGUAGE 为 None 的格式编码。

例如：

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

会添加一个形如下文的头字段：

```
Content-Disposition: attachment; filename="bud.gif"
```

带有非 ASCII 字符的拓展接口：

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

replace_header(*_name*, *_value*)
替换头字段。只会替换掉信息内找到的第一个字段名匹配 *_name* 的字段值。字段的顺序不变，原字段名的大小写也不变。如果没有找到匹配的字段，抛出*KeyError* 异常。

get_content_type()
返回信息的内容类型，其形如 *maintype/subtype* ，强制全小写。如果信息的 *Content-Type* 头字段不存在则返回*get_default_type()* 的返回值；如果信息的 *Content-Type* 头字段无效则返回 text/plain 。

（根据 **RFC 2045** 所述，信息永远都有一个默认类型，所以*get_content_type()* 一定会返回一个值。**RFC 2045** 定义信息的默认类型为 *text/plain* 或 *message/rfc822* ，其中后者仅出现在消息头位于一个 *multipart/digest* 容器中的场合中。如果消息头的 *Content-Type* 字段所指定的类型是无效的，**RFC 2045** 令其默认类型为 *text/plain* 。）

get_content_maintype()
返回信息的主要内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *maintype* 部分。

get_content_subtype()
返回信息的子内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *subtype* 部分。

get_default_type()
返回默认的内容类型。绝大多数的信息，其默认内容类型都是 *text/plain* 。作为 *multipart/digest* 容器内子部分的信息除外，它们的默认内容类型是 *message/rfc822* 。

set_default_type(*ctype*)
设置默认的内容类型。尽管并非强制，但是 *ctype* 仍应当是 *text/plain* 或 *message/rfc822* 二者取一。默认内容类型并不存储在 *Content-Type* 头字段当中，所以设置此项的唯一作用就是决定当 *Content-Type* 头字段在信息中不存在时，get_content_type 方法的返回值。

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)
在 *Content-Type* 头字段当中设置一个参数。如果该参数已于字段中存在，将其旧值替换为 *value* 。如果 *header* 是 Content-Type （默认值），并且该头字段于信息中尚未存在，则会先添加该字

段，将其值设置为 *text/plain* ，并附加参数值。可选的 *header* 可以让你指定 *Content-Type* 之外的另一个头字段。

如果值包含非 ASCII 字符，其字符集和语言可以通过可选参数 *charset* 和 *language* 显式指定。可选参数 *language* 指定 **RFC 2231** 当中的语言，其默认值是空字符串。*charset* 和 *language* 都应当字符串。默认使用的是 utf8 *charset* ， *language* 为 None 。

如果 *replace* 为 False （默认值），该头字段会被移动到所有头字段列表的末尾。如果 *replace* 为 True ，字段会被原地更新。

于*EmailMessage* 对象而言，*requote* 参数已被弃用。

请注意，头字段已有的参数值可以通过头字段的 params 属性来访问（举例: msg['Content-Type'].params['charset'] ）。

3.4 版更變: 添加了 replace 关键字。

del_param(*param*, *header='content-type'*, *requote=True*)
从 *Content-Type* 头字段中完全移去给定的参数。头字段会被原地重写，重写后的字段不含参数和值。可选的 *header* 可以让你指定 *Content-Type* 之外的另一个字段。

于*EmailMessage* 对象而言，*requote* 参数已被弃用。

get_filename(*failobj=None*)
返回信息头当中 *Content-Disposition* 字段当中名为 filename 的参数值。如果该字段当中没有此参数，该方法会退而寻找 *Content-Type* 字段当中的 name 参数值。如果这个也没有找到，或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照*email.utils.unquote()* 方法去除引号。

get_boundary(*failobj=None*)
返回信息头当中 *Content-Type* 字段当中名为 boundary 的参数值。如果字段当中没有此参数，或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照*email.utils.unquote()* 方法去除引号。

set_boundary(*boundary*)
将 *Content-Type* 头字段的 boundary 参数设置为 *boundary* 。*set_boundary()* 方法永远都会在必要的时候为 *boundary* 添加引号。如果信息对象中没有 *Content-Type* 头字段，抛出*HeaderParseError* 异常。

请注意使用这个方法与直接删除旧的 *Content-Type* 头字段然后使用*add_header()* 方法添加一个带有新边界值参数的 *Content-Type* 头字段有细微差距。*set_boundary()* 方法会保留 *Content-Type* 头字段在原信息头当中的位置。

get_content_charset(*failobj=None*)
返回 *Content-Type* 头字段中的 charset 参数，强制小写。如果字段当中没有此参数，或者这个字段压根不存在，返回 *failobj* 。

get_charsets(*failobj=None*)
返回一个包含了信息内所有字符集名字的列表。如果信息是 *multipart* 类型的，那么列表当中的每一项都对应其载荷的子部分的字符集名字。否则，该列表是一个长度为 1 的列表。

列表当中的每一项都是一个字符串，其值为对应子部分的 *Content-Type* 头字段的 charset 参数值。如果该子部分没有此头字段，或者没有此参数，或者其主要 MIME 类型并非 *text* ，那么列表中的那一项即为 *failobj* 。

is_attachment()
如果信息头当中存在一个名为 *Content-Disposition* 的字段，且该字段的值为 attachment（大小写无关），返回 True 。否则，返回 False 。

3.4.2 版更變: 为了与*is_multipart()* 方法一致，is_attachment 现在是一个方法，不再是属性了。

get_content_disposition()
如果信息的 *Content-Disposition* 头字段存在，返回其字段值；否则返回 None 。返回的值均为小写，不包含参数。如果信息遵循 **RFC 2183** 标准，则此方法的返回值只可能在 *inline* 、*attachment* 和 None 之间选择。

3.5 版新加入.

下列方法与信息内容（载荷）之访问与操控有关。

walk()
walk() 方法是一个多功能生成器。它可以被用来以深度优先顺序遍历信息对象树的所有部分和子部分。一般而言，*walk()* 会被用作 for 循环的迭代器，每一次迭代都返回其下一个子部分。

以下例子会打印出一封具有多部分结构之信息的每个部分的 MIME 类型。

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

walk 会遍历所有*is_multipart()* 方法返回 True 的部分之子部分，哪怕 msg.get_content_maintype() == 'multipart' 返回的是 False 。使用 _structure 除错帮助函数可以帮助我们在下面这个例子当中看清楚这一点：

```
>>> from email.iterators import _structure
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

在这里，message 的部分并非 multiparts ，但是它们真的包含子部分！is_multipart() 返回 True ， walk 也深入进这些子部分中。

get_body(*preferencelist=('related', 'html', 'plain')*)
返回信息的 MIME 部分。这个部分是最可能成为信息体的部分。

preferencelist 必须是一个字符串序列，其内容从 related 、html 和 plain 这三者组成的集合中选取。这个序列代表着返回的部分的内容类型之偏好。

在 get_body 方法被调用的对象上寻找匹配的候选者。

如果 related 未包括在 *preferencelist* 中，可考虑将所遇到的任意相关的根部分（或根部分的子部分）在该（子）部分与一个首选项相匹配时作为候选项。

当遇到一个 multipart/related 时，将检查 start 形参并且如果找到了一个匹配 *Content-ID* 的部分，在查找候选匹配时只考虑它。在其他情况下则只考虑 multipart/related 的第一个（默认的根）部分。

如果一个部分具有 *Content-Disposition* 标头，则当标头值为 inline 时将只考虑将该部分作为候选匹配。

如果没有任何候选部分匹配 *preferencelist* 中的任何首选项，则返回 None。

注: (1) 对于大多数应用来说有意义的 *preferencelist* 组合仅有 ('plain',), ('html', 'plain') 以及默认的 ('related', 'html', 'plain')。(2) 由于匹配是从调用 get_body 的对象开始的，因此在 multipart/related 上调用 get_body 将返回对象本身，除非 *preferencelist* 具有非默认值。(3) 未指定 *Content-Type* 或者 *Content-Type* 标头无效的消息（或消息部分）将被当作具有 text/plain 类型来处理，这有时可能导致 get_body 返回非预期的结果。

iter_attachments()
返回包含所有不是候选"body" 部分的消息的直接子部分的迭代器。也就是说，跳过首次出现的每个 text/plain, text/html, multipart/related 或 multipart/alternative (除非通过 *Content-Disposition: attachment* 将它们显式地标记为附件)，并返回所有的其余部分。当直接应用于 multipart/related 时，将返回包含除根部分之外所有相关部分的迭代器（即由 start 形参所指向的部分，或者当没有 start 形参或 start 形参不能匹配任何部分的 *Content-ID* 时则为第一部分）。当直接应用于 multipart/alternative 或非 multipart 时，将返回一个空迭代器。

iter_parts()
返回包含消息的所有直接子部分的迭代器，对于非 multipart 将为空对象。(另请参阅*walk()*。)

get_content(**args*, *content_manager=None*, ***kw*)
调用 *content_manager* 的*get_content()* 方法，将自身作为消息对象传入，并将其他参数或关键字作为额外参数传入。如果未指定 *content_manager*，则会使用当前*policy* 所指定的 content_manager。

set_content(**args*, *content_manager=None*, ***kw*)
调用 *content_manager* 的*set_content()* 方法，将自身作为消息传入，并将其他参数或关键字作为额外参数传入。如果未指定 *content_manager*，则会使用当前*policy* 所指定的 content_manager。

make_related(*boundary=None*)
将非 multipart 消息转换为 multipart/related 消息，将任何现有的 *Content-* 标头和载荷移入 multipart 的（新加）首部分。如果指定了 *boundary*，会用它作为 multipart 中的分界字符串，否则会在必要时自动创建分界（例如当消息被序列化时）。

make_alternative(*boundary=None*)
将非 multipart 或 multipart/related 转换为 multipart/alternative，将任何现有的 *Content-* 标头和载荷移入 multipart 的（新加）首部分。如果指定了 *boundary*，会用它作为 multipart 中的分界字符串，否则会在必要时自动创建分界（例如当消息被序列化时）。

make_mixed(*boundary=None*)
将非 multipart, multipart/related 或 multipart-alternative 转换为 multipart/mixed，将任何现有的 *Content-* 标头和载荷移入 multipart 的（新加）首部分。如果指定了 *boundary*，会用它作为 multipart 中的分界字符串，否则会在必要时自动创建分界（例如当消息被序列化时）。

add_related(**args*, *content_manager=None*, ***kw*)
如果消息为 multipart/related，则创建一个新的消息对象，将所有参数传给其*set_content()* 方法，并将其*attach()* 到 multipart。如果消息为非 multipart，则先调用*make_related()* 然后再继续上述步骤。如果消息为任何其他类型的 multipart，则会引

发*TypeError*。如果未指定 *content_manager*，则使用当前*policy* 所指定的 content_manager。如果添加的部分没有 *Content-Disposition* 标头，则会添加一个值为 inline 的标头。

add_alternative(**args*, *content_manager=None*, ***kw*)
如果消息为 multipart/alternative，则创建一个新的消息对象，将所有参数传给其*set_content()* 方法，并将其*attach()* 到 multipart。如果消息为非 multipart 或 multipart/related，则先调用*make_alternative()* 然后再继续上述步骤。如果消息为任何其他类型的 multipart，则会引发*TypeError*。如果未指定 *content_manager*，则会使用当前*policy* 所指定的 content_manager。

add_attachment(**args*, *content_manager=None*, ***kw*)
如果消息为 multipart/mixed，则创建一个新的消息对象，将所有参数传给其*set_content()* 方法，并将其*attach()* 到 multipart。如果消息为非 multipart，multipart/related 或 multipart/alternative，则先调用*make_mixed()* 然后再继续上述步骤。如果未指定 *content_manager*，则使用当前*policy* 所指定的 content_manager。如果添加的部分没有 *Content-Disposition* 标头，则会添加一个值为 attachment 的标头。此方法对于显式附件 (*Content-Disposition: attachment*) 和 inline 附件 (*Content-Disposition: inline*) 均可使用，只须向 content_manager 传入适当的选项即可。

clear()
移除所有载荷和所有标头。

clear_content()
移除载荷以及所有 Content- 标头，其他标头不加改变并且保持其原有顺序。

EmailMessage 对象具有下列实例属性:

preamble
MIME 文档格式在标头之后的空白行以及第一个多部分的分界字符串之间允许添加一些文本，通常，此文本在支持 MIME 的邮件阅读器中永远不可见，因为它处在标准 MIME 保护范围之外。但是，当查看消息的原始文本，或当在不支持 MIME 的阅读器中查看消息时，此文本会变得可见。

preamble 属性包含 MIME 文档开头部分的这些处于保护范围之外的文本。当*Parser* 在标头之后及第一个分界字符串之前发现一些文本时，它会将这些文本赋值给消息的 *preamble* 属性。当*Generator* 写出 MIME 消息的纯文本表示形式时，如果它发现消息具有 *preamble* 属性，它将在标头及第一个分界之间区域写出这些文本。请参阅*email.parser* 和*email.generator* 了解更多细节。

请注意如果消息对象没有前导文本，则 *preamble* 属性将为 None。

epilogue
epilogue 属性的作用方式与 *preamble* 相同，区别在于它包含在最后一个分界及消息结尾之间出现的文本。与*preamble* 类似，如果没有附加文本，则此属性将为 None。

defects
defects 属性包含在解析消息时发现的所有问题的列表。请参阅*email.errors* 了解可能的解析缺陷的详细描述。

class email.message.**MIMEPart**(*policy=default*)
这个类代表 MIME 消息的子部分。它与*EmailMessage* 相似，不同之处在于当*set_content()* 被调用时不会添加 *MIME-Version* 标头，因为子部分不需要有它们自己的 *MIME-Version* 标头。

脚注

19.1.2 email.parser: 解析电子邮件信息

源代码: Lib/email/parser.py

使用以下两种方法的其中一种以创建消息对象结构：直接创建一个*EmailMessage* 对象，使用字典接口添加消息头，并且使用*set_content()* 和其他相关方法添加消息负载；或者通过解析一个电子邮件消息的序列化表达来创建消息对象结构。

email 包提供了一个可以理解包含 MIME 文档在内的绝大多数电子邮件文档结构的标准语法分析程序。你可以传递给语法分析程序一个字节串、字符串或者文件对象，语法分析程序会返回给你对应于该对象结构的根*EmailMessage* 实例。对于简单的、非 MIME 的消息，这个根对象的负载很可能就是一个包含了该消息文字内容的字符串。对于 MIME 消息，调用根对象的*is_multipart()* 方法会返回 `True`，其子项可以通过负载操纵方法来进行访问，例如*get_body()*、*iter_parts()* 还有*walk()*。

事实上你可以使用的语法分析程序接口有两种: *Parser* API 和增量式的*FeedParser* API。当你的全部消息内容都在内存当中，或者整个消息都保存在文件系统内的一个文件当中的时候，*Parser* API 非常有用。当你从可能会为了等待更多输入而阻塞的数据流当中读取消息（比如从套接字当中读取电子邮件消息）的时候，*FeedParser* 会更合适。*FeedParser* 会增量读取并解析消息，并且只有在你关闭语法分析程序的时候才会返回根对象。

请注意，语法分析程序可以进行有限的拓展，你当然也可以完全从零开始实现你自己的语法分析程序。将*email* 包中内置的语法分析程序和*EmailMessage* 类连接起来的所有逻辑代码都包含在 `policy` 类当中，所以如有必要，自定义的语法分析程序可以通过实现自定义的对应 `policy` 方法来创建对应的消息对象树。

FeedParser API

从 `email.feedparser` 模块导入的*BytesFeedParser* 类提供了一个适合于增量解析电子邮件消息的 API，比如说在从一个可能会阻塞（例如套接字）的源当中读取消息文字的场合中它就会变得很有用。当然，*BytesFeedParser* 也可以用来解析一个已经完整包含在一个*bytes-like object* 、字符串或文件内的电子邮件消息，但是在这些场合下使用*BytesParser* API 可能会更加方便。这两个语法分析程序 API 的语义和最终结果是一致的。

BytesFeedParser 的 API 十分简洁易懂：你创建一个语法分析程序的实例，向它不断输入大量的字节直到尽头，然后关闭这个语法分析程序就可以拿到根消息对象了。在处理符合标准的消息的时候*BytesFeedParser* 非常准确；在处理不符合标准的消息的时候它做的也不差，但这视消息损坏的程度而定。它会向消息对象的*defects* 属性中写入它从消息中找到的问题列表。关于它能找到的所有问题类型的列表，详见*email.errors* 模块。

这里是*BytesFeedParser* 的 API：

class email.parser.**BytesFeedParser**(*_factory=None*, ***, *policy=policy.compat32*)

创建一个*BytesFeedParser* 实例。可选的 *_factory* 参数是一个不带参数的可调用对象；如果没有被指定，就会使用 *policy* 参数的*message_factory* 属性。每当需要一个新的消息对象的时候，*_factory* 都会被调用。

如果指定了 *policy* 参数，它就会使用这个参数所指定的规则来更新消息的表达方式。如果没有设定 *policy* 参数，它就会使用*compat32* 策略。这个策略维持了对 Python 3.2 版本的 email 包的后向兼容性，并且使用*Message* 作为默认的工厂。其他策略使用*EmailMessage* 作为默认的 *_factory*。关于 *policy* 还会控制什么，参见*policy* 的文档。

注：**一定要指定 policy 关键字**。在未来版本的 Python 当中，它的默认值会变成*email.policy.default* 。

3.2 版新加入.

3.3 版更變: 添加了 *policy* 关键字。

3.6 版更變: *_factory* 默认为策略 `message_factory`。

feed(*data*)
向语法分析程序输入更多数据。*data* 应当是一个包含一行或多行内容的*bytes-like object*。行内容可以是不完整的，语法分析程序会妥善的将这些不完整的行缝合在一起。每一行可以使用以下三种常见的终止符号的其中一种：回车符、换行符或回车符加换行符（三者甚至可以混合使用）。

close()
完成之前输入的所有数据的解析并返回根消息对象。如果在这个方法被调用之后仍然调用*feed()*方法，结果是未定义的。

class email.parser.**FeedParser**(*_factory=None*, *, *policy=policy.compat32*)
行为跟*BytesFeedParser*类一致，只不过向*feed()*方法输入的内容必须是字符串。它的实用性有限，因为这种消息只有在其只含有 ASCII 文字，或者 `utf8` 被设置为 `True` 且没有二进制格式的附件的时候，才会有效。

3.3 版更變: 添加了 *policy* 关键字。

Parser API

*BytesParser*类从*email.parser*模块导入，当消息的完整内容包含在一个*bytes-like object* 或文件中时它提供了可用于解析消息的 API。*email.parser*模块还提供了*Parser*用来解析字符串，以及只用来解析消息头的*BytesHeaderParser*和*HeaderParser*，如果你只对消息头感兴趣就可以使用后两者。在这种场合下*BytesHeaderParser*和*HeaderParser*速度非常快，因为它们并不会尝试解析消息体，而是将载荷设为原始数据。

class email.parser.**BytesParser**(*_class=None*, *, *policy=policy.compat32*)
创建一个*BytesParser*实例。*_class* 和 *policy* 参数在含义和语义上与*BytesFeedParser*的 *_factory* 和 *policy* 参数一致。

注:**一定要指定 policy 关键字**。在未来版本的 Python 当中,它的默认值会变成*email.policy.default*。

3.3 版更變: 移除了在 2.4 版本中被弃用的 *strict* 参数。新增了 *policy* 关键字。

3.6 版更變: *_class* 默认为策略 `message_factory`。

parse(*fp*, *headersonly=False*)
从二进制的类文件对象 *fp* 中读取全部数据、解析其字节内容、并返回消息对象。*fp* 必须同时支持*readline()*方法和 `read()` 方法。

fp 内包含的字节内容必须是一块遵循 **RFC 5322** （如果 `utf8` 为 `True`，则为 **RFC 6532** ）格式风格的消息头和消息头延续行，并可能紧跟一个信封头。头块要么以数据结束，要么以一个空行为终止。跟着头块的是消息体（消息体内可能包含 MIME 编码的子部分，这也包括 *Content-Transfer-Encoding* 字段为 `8bit` 的子部分）。

可选的 *headersonly* 指示了是否应当在读取完消息头后就终止。默认值为 `False` ，意味着它会解析整个文件的全部内容。

parsebytes(*bytes*, *headersonly=False*)
与*parse()*方法类似，只不过它要求输入为一个*bytes-like object* 而不是类文件对象。于一个*bytes-like object* 调用此方法相当于先将这些字节包装于一个*BytesIO*实例中，然后调用*parse()*方法。

可选的 *headersonly* 与*parse()*方法中的 *headersonly* 是一致的。

3.2 版新加入.

class email.parser.**BytesHeaderParser**(*_class=None*, ***, *policy=policy.compat32*)
除了 *headersonly* 默认为 `True`，其他与*BytesParser* 类完全一样。

3.3 版新加入.

class email.parser.**Parser**(*_class=None*, ***, *policy=policy.compat32*)
这个类与*BytesParser* 一样，但是处理字符串输入。

3.3 版更變: 移除了 *strict* 参数。添加了 *policy* 关键字。

3.6 版更變: *_class* 默认为策略 `message_factory`。

parse(*fp*, *headersonly=False*)
从文本模式的文件类对象 *fp* 读取所有数据，解析所读取的文本，并返回根消息对象。*fp* 必须同时支持文件类对象上的*readline()* 和*read()* 方法。

除了文字模式的要求外，这个方法跟*BytesParser.parse()* 的运行方式一致。

parsestr(*text*, *headersonly=False*)
与*parse()* 方法类似，只不过它要求输入为一个字符串而不是类文件对象。于一个字符串对象调用此方法相当于先将 *text* 包装于一个*StringIO* 实例中，然后调用*parse()* 方法。

可选的 *headersonly* 与*parse()* 方法中的 *headersonly* 是一致的。

class email.parser.**HeaderParser**(*_class=None*, ***, *policy=policy.compat32*)
除了 *headersonly* 默认为 `True` ，其他与*Parser* 类完全一样。

考虑到从一个字符串或一个文件对象中创建一个消息对象是非常常见的任务，我们提供了四个方便的函数。它们于顶层*email* 包命名空间内可用。

email.**message_from_bytes**(*s*, *_class=None*, ***, *policy=policy.compat32*)
从一个*bytes-like object* 中返回消息对象。这与 `BytesParser().parsebytes(s)` 等价。可选的 *_class* 和 *policy* 参数与*BytesParser* 类的构造函数的参数含义一致。

3.2 版新加入.

3.3 版更變: 移除了 *strict* 参数。添加了 *policy* 关键字。

email.**message_from_binary_file**(*fp*, *_class=None*, ***, *policy=policy.compat32*)
从打开的二进制*file object* 中返回消息对象。这与 `BytesParser().parse(fp)` 等价。*_class* 和 *policy* 参数与*BytesParser* 类的构造函数的参数含义一致。

3.2 版新加入.

3.3 版更變: 移除了 *strict* 参数。添加了 *policy* 关键字。

email.**message_from_string**(*s*, *_class=None*, ***, *policy=policy.compat32*)
从一个字符串中返回消息对象。这与 `Parser().parsestr(s)` 等价。*_class* 和 *policy* 参数与*Parser* 类的构造函数的参数含义一致。

3.3 版更變: 移除了 *strict* 参数。添加了 *policy* 关键字。

email.**message_from_file**(*fp*, *_class=None*, ***, *policy=policy.compat32*)
从一个打开的*file object* 中返回消息对象。这与 `Parser().parse(fp)` 等价。*_class* 和 *policy* 参数与*Parser* 类的构造函数的参数含义一致。

3.3 版更變: 移除了 *strict* 参数。添加了 *policy* 关键字。

3.6 版更變: *_class* 默认为策略 `message_factory`。

这里是一个展示了你如何在 Python 交互式命令行中使用*message_from_bytes()* 的例子:

```
>>> import email
>>> msg = email.message_from_bytes(myBytes)
```

附加说明

在解析语义的时候请注意：

- 大多数非 *multipart* 类型的消息都会被解析为一个带有字符串负载的消息对象。这些对象在调用*is_multipart()* 的时候会返回 False，调用*iter_parts()* 的时候会产生一个空列表。
- 所有 *multipart* 类型的消息都会被解析成一个容器消息对象。该对象的负载是一个子消息对象列表。外层的容器消息在调用*is_multipart()* 的时候会返回 True，在调用*iter_parts()* 的时候会产生一个子部分列表。
- 大多数内容类型为 *message/** （例如 *message/delivery-status* 和 *message/rfc822* ）的消息也会被解析为一个负载是长度为 1 的列表的容器对象。在它们身上调用*is_multipart()* 方法会返回 True，调用*iter_parts()* 所产生的单个元素会是一个子消息对象。
- 一些不遵循标准的消息在其内部关于它是否为 *multipart* 类型前后不一。这些消息可能在消息头的 *Content-Type* 字段中写明为 *multipart* ，但它们的*is_multipart()* 方法的返回值可能是 False 。如果这种消息被*FeedParser* 类解析，它们的 *defects* 属性列表当中会有一个 MultipartInvariantViolationDefect 类的实例。关于更多信息，详见*email.errors* 。

19.1.3 email.generator: 生成 MIME 文档

源代码: Lib/email/generator.py

最普通的一种任务是生成由消息对象结构体表示的电子邮件消息的扁平（序列化）版本。如果你想通过*smtplib.SMTP.sendmail()* 或*nntplib* 模块来发送你的消息或是将消息打印到控制台就将需要这样做。接受一个消息对象结构体并生成其序列化表示就是这些生成器类的工作。

与*email.parser* 模块一样，你并不会受限于已捆绑生成器的功能；你可以自己从头写一个。不过，已捆绑生成器知道如何以符合标准的方式来生成大多数电子邮件，应该能够很好地处理 MIME 和非 MIME 电子邮件消息，并且被设计为面向字节的解析和生成操作是互逆的，它假定两者都使用同样的非转换型*policy*。也就是说，通过*BytesParser* 类来解析序列化字节流然后再使用*BytesGenerator* 来重新生成序列化字节流应当得到与输入相同的结果[1]。（而另一方面，在由程序所构造的*EmailMessage* 上使用生成器可能导致对默认填入的*EmailMessage* 对象的改变。。）

可以使用*Generator* 类将消息扁平化为文本（而非二进制数据）的序列化表示形式，但是由于 Unicode 无法直接表示二进制数据，因此消息有必要被转换为仅包含 ASCII 字符的数据，这将使用标准电子邮件 RFC 内容传输编码格式技术来编码电子邮件消息以便通过非“8 比特位兼容”的信道来传输。

为了适应 SMIME 签名消息的可重现处理过程，*Generator* 禁用了针对 multipart/signed 类型的消息部分及所有子部分的标头折叠。

class email.generator.**BytesGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

返回一个*BytesGenerator* 对象，该对象将把提供给*flatten()* 方法的任何消息或者提供给*write()* 方法的任何经过代理转义编码的文本写入到*file-like object* *outfp*。*outfp* 必须支持接受二进制数据的 write 方法。

如果可选的 *mangle_from_* 为 True，则会将一个 > 字符放到恰好以字符串 "From " 打头，即开头文本为 From 加一个空格的的行前面。*mangle_from_* 默认为 *policy* 的*mangle_from_* 设置值 (对于*compat32* 策略为 True，对于所有其他策略则为 False)。*mangle_from_* 被设计为在当消息以 unix mbox 格式存储时使用 (参见*mailbox* 和 WHY THE CONTENT-LENGTH FORMAT IS BAD)。

[1] 此语句假定你使用了正确的 unixfrom 设置，并且没有用于自动调整的 policy 设置调用（例如 refold_source 必须为 none，这 不是默认值）。这也不是 100% 为真，因为如果消息不遵循 RFC 标准则有时实际原始文本的信息会在解析错误恢复时丢失。它的目标是在可能的情况下修复这些后续边缘情况。

如果 *maxheaderlen* 不为 None，则重新折叠任何长于 *maxheaderlen* 的标头行，或者如果为 0，则不重新包装任何标头。如果 *manheaderlen* 为 None (默认值)，则根据 *policy* 设置包装标头和其他消息行。

如果指定了 *policy*，则使用该策略来控制消息的生成。如果 *policy* 为 None (默认值)，则使用与传递给 flatten 的*Message* 或*EmailMessage* 对象相关联的策略来控制消息的生成。请参阅*email.policy* 了解有关 *policy* 所控制内容的详情。

3.2 版新加入.

3.3 版更變: 添加了 *policy* 关键字。

3.6 版更變: *mangle_from_* 和 *maxheaderlen* 形参的默认行为是遵循策略。

flatten(*msg*, *unixfrom=False*, *linesep=None*)

将将以 *msg* 为根的消息对象结构体的文本表示形式打印到创建*BytesGenerator* 实例时指定的输出文件。

如果*policy* 选项*cte_type* 为 8bit (默认值)，则会将未被修改的原始已解析消息中的任何标头拷贝到输出，其中会重新生成与原始数据相同的高比特位组字节数据，并保留具有它们的任何消息体部分的非 ASCII *Content-Transfer-Encoding*。如果 cte_type 为 7bit，则会根据需要使用兼容 ASCII 的 *Content-Transfer-Encoding* 来转换高比特位组字节数据。也就是说，将具有非 ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) 的部分转换为兼容 ASCII 的 *Content-Transfer-Encoding*，并使用 MIME unknown-8bit 字符集来编码标头中不符合 RFC 的非 ASCII 字节数据，以使其符合 RFC。

如果 *unixfrom* 为 True，则会在根消息对象的第一个 **RFC 5322** 标头之前打印 Unix mailbox 格式 (参见*mailbox*) 所使用的封包标头分隔符。如果根对象没有封包标头，则会创建一个标准标头。默认值为 False。请注意对于子部分来说，不会打印任何封包标头。

如果 *linesep* 不为 None，则会将其用作扁平化消息的所有行之间的分隔符。如果 *linesep* 为 None (默认值)，则使用在 *policy* 中指定的值。

clone(*fp*)

返回此*BytesGenerator* 实例的独立克隆，具有完全相同的选项设置，而 *fp* 为新的 *outfp*。

write(*s*)

使用 ASCII 编解码器和 surrogateescape 错误处理程序编码 *s*，并将其传递给传入到*BytesGenerator* 的构造器的 *outfp* 的 *write* 方法。

作为一个便捷工具，*EmailMessage* 提供了*as_bytes()* 和 bytes(aMessage) (即*__bytes__()*) 等方法，它们简单地生成一个消息对象的序列化二进制表示形式。更多细节请参阅*email.message*。

因为字符串无法表示二进制数据，*Generator* 类必须将任何消息中扁平化的任何二进制数据转换为兼容 ASCII 的格式，具体将其转换为兼容 ASCII 的 *Content-Transfer_Encoding*。使用电子邮件 RFC 的术语，你可以将其视作*Generator* 序列化为不" 支持 8 比特" 的 I/O 流。换句话说，大部分应用程序将需要使用*BytesGenerator*，而非*Generator*。

class email.generator.**Generator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, ***, *policy=None*)

返回一个*Generator*，它将把提供给*flatten()* 方法的任何消息，或者提供给*write()* 方法的任何文本写入到*file-like object* *outfp*。*outfp* 必须支持接受字符串数据的 write 方法。

如果可选的 *mangle_from_* 为 True，则会将一个 > 字符放到恰好以字符串 "From " 打头，即开头文本为 From 加一个空格的的行前面。*mangle_from_* 默认为 *policy* 的*mangle_from_* 设置值 (对于*compat32* 策略为 True，对于所有其他策略则为 False)。*mangle_from_* 被设计为在当消息以 unix mbox 格式存储时使用 (参见*mailbox* 和 WHY THE CONTENT-LENGTH FORMAT IS BAD)。

如果 *maxheaderlen* 不为 None，则重新折叠任何长于 *maxheaderlen* 的标头行，或者如果为 0，则不重新包装任何标头。如果 *manheaderlen* 为 None (默认值)，则根据 *policy* 设置包装标头和其他消息行。

如果指定了 *policy*，则使用该策略来控制消息的生成。如果 *policy* 为 `None` (默认值)，则使用与传递给 `flatten` 的*Message* 或*EmailMessage* 对象相关联的策略来控制消息的生成。请参阅*email.policy* 了解有关 *policy* 所控制内容的详情。

3.3 版更變: 添加了 *policy* 关键字。

3.6 版更變: *mangle_from_* 和 *maxheaderlen* 形参的默认行为是遵循策略。

flatten(*msg*, *unixfrom=False*, *linesep=None*)

将以 *msg* 为根的消息对象结构体的文本表示形式打印到当*Generator* 实例被创建时所指定的输出文件。

如果*policy* 选项*cte_type* 为 `8bit`，则视同选项被设为 `7bit` 来生成消息。（这是必需的，因为字符串无法表示非 ASCII 字节数据。）将使用兼容 ASCII 的 *Content-Transfer-Encoding* 按需转换任何具有高比特位组的字节数据。也就是说，将具有非 ASCII *Content-Transfer-Encoding* (*Content-Transfer-Encoding: 8bit*) 的部分转换为兼容 ASCII 的 *Content-Transfer-Encoding*，并使用 MIME `unknown-8bit` 字符集来编码标头中不符合 RFC 的非 ASCII 字节数据，以使其符合 RFC。

如果 *unixfrom* 为 `True`，则会在根消息对象的第一个 **RFC 5322** 标头之前打印 Unix mailbox 格式 (参见*mailbox*) 所使用的封包标头分隔符。如果根对象没有封包标头，则会创建一个标准标头。默认值为 `False`。请注意对于子部分来说，不会打印任何封包标头。

如果 *linesep* 不为 `None`，则会将其用作扁平化消息的所有行之间的分隔符。如果 *linesep* 为 `None` (默认值)，则使用在 *policy* 中指定的值。

3.2 版更變: 添加了对重编码 `8bit` 消息体的支持，以及 *linesep* 参数。

clone(*fp*)

返回此*Generator* 实例的独立克隆，具有完全相同的选项设置，而 *fp* 为新的 *outfp*。

write(*s*)

将 *s* 写入到传给*Generator* 的构造器的 *outfp* 的 *write* 方法。这足够为*Generator* 实际提供可用于*print()* 函数的文件类 API。

作为一个便捷工具，*EmailMessage* 提供了*as_string()* 和 `str(aMessage)` (即*__str__()*) 等方法，它们简单地生成一个消息对象的已格式化字符串表示形式。更多细节请参阅*email.message*。

email.generator 模块还提供了一个派生类*DecodedGenerator*，它类似于*Generator* 基类，不同之处在于非 *text* 部分不会被序列化，而是被表示为基于模板并填写了有关该部分的信息的字符串输出流的形式。

class email.generator.**DecodedGenerator**(*outfp*, *mangle_from_=None*, *maxheaderlen=None*, *fmt=None*, ***, *policy=None*)

行为类似于*Generator*，不同之处在于对传给*Generator.flatten()* 的消息的任何子部分，如果该子部分的主类型为 *text*，则打印该子部分的已解码载荷，而如果其主类型不为 *text*，则不直接打印它而是使用来自该部分的信息填入字符串 *fmt* 并将填写完成的字符串打印出来。

要填入 *fmt*，则执行 `fmt % part_info`，其中 `part_info` 是由下列键和值组成的字典:

- `type` -- 非 *text* 部分的完整 MIME 类型
- `maintype` -- 非 *text* 部分的主 MIME 类型
- `subtype` -- 非 *text* 部分的子 MIME 类型
- `filename` -- 非 *text* 部分的文件名
- `description` -- 与非 *text* 部分相关联的描述
- `encoding` -- 非 *text* 部分的内容转换编码格式

如果 *fmt* 为 `None`，则使用下列默认 *fmt*:

”[忽略消息的非文本 (%(type)s) 部分，文件名%(filename)s]”

可选的 *_mangle_from_* 和 *maxheaderlen* 与*Generator* 基类的相同。

🄵解

19.1.4 email.policy: Policy 对象

3.3 版新加入.

源代码: Lib/email/policy.py

---

email 的主要焦点是按照各种电子邮件和 MIME RFC 的描述来处理电子邮件消息。但是电子邮件消息的基本格式（一个由名称加冒号加值的标头字段构成的区块，后面再加一个空白行和任意的‘消息体’）是在电子邮件领域以外也获得应用的格式。这些应用的规则有些与主要电子邮件 RFC 十分接近，有些则很不相同。即使是操作电子邮件，有时也可能需要打破严格的 RFC 规则，例如生成可与某些并不遵循标准的电子邮件服务器互联的电子邮件，或者是实现希望应用某些破坏标准的操作方式的扩展。

Policy 对象给予 email 包处理这些不同用例的灵活性。

Policy 对象封装了一组属性和方法用来在使用期间控制 email 包中各个组件的行为。*Policy* 实例可以被传给 email 包中的多个类和方法以更改它们的默认行为。可设置的值及其默认值如下所述。

在 email 包中的所有类会使用一个默认的策略。对于所有*parser* 类及相关的便捷函数，还有对于*Message* 类来说，它是*Compat32* 策略，通过其对应的预定义实例*compat32* 来使用。这个策略提供了与 Python3.3 版之前的 email 包的完全向下兼容性（在某些情况下，也包括对缺陷的兼容性）。

传给*EmailMessage* 的 *policy* 关键字的默认值是*EmailPolicy* 策略，表示为其预定义的实例*default*。

在创建*Message* 或*EmailMessage* 对象时，它需要一个策略。如果消息是由*parser* 创建的，则传给该解析器的策略将是它所创建的消息所使用的策略。如果消息是由程序创建的，则该策略可以在创建它的时候指定。当消息被传递给*generator* 时，生成器默认会使用来自该消息的策略，但你也可以将指定的策略传递给生成器，这将覆盖存储在消息对象上的策略。

email.parser 类和解析器便捷函数的 *policy* 关键字的默认值在未来的 Python 版本中 **将会改变**。因此在调用任何*parser* 模块所描述的类和函数时你应当 **总是显式地指定你想要使用的策略**。

本文档的第一部分介绍了*Policy* 的特性，它是一个*abstract base class*，定义了所有策略对象包括*compat32* 的共有特性。这些特性包括一些由 email 包内部调用的特定钩子方法，自定义策略可以重载这些方法以获得不同行为。第二部分描述了实体类*EmailPolicy* 和*Compat32*，它们分别实现了提供标准行为和向下兼容行为与特性的钩子。

Policy 实例是不可变的，但它们可以被克隆，接受与类构造器一致的关键字参数并返回一个新的*Policy* 实例，新实例是原实例的副本，但具有被改变的指定属性。

例如，以下代码可以被用来从一个 Unix 系统的磁盘文件中读取电子邮件消息并将其传递给系统的 `sendmail` 程序:

```
>>> from email import message_from_binary_file
>>> from email.generator import BytesGenerator
>>> from email import policy
>>> from subprocess import Popen, PIPE
>>> with open('mymsg.txt', 'rb') as f:
...     msg = message_from_binary_file(f, policy=policy.default)
>>> p = Popen(['sendmail', msg['To'].addresses[0]], stdin=PIPE)
>>> g = BytesGenerator(p.stdin, policy=msg.policy.clone(linesep='\r\n'))
>>> g.flatten(msg)
```

(下页继续)

(繼續上一頁)

```
>>> p.stdin.close()
>>> rc = p.wait()
```

这里我们让*BytesGenerator* 在创建要送入 sendmail's stdin 的二进制字串时使用符合 RFC 的行分隔字符，默认的策略将会使用 \n 行分隔符。

某些 email 包的方法接受一个 *policy* 关键字参数，允许为该方法重载策略。例如，以下代码使用了来自之前示例的 *msg* 对象的*as_bytes()* 方法并使用其运行所在平台的本机行分隔符将消息写入一个文件:

```
>>> import os
>>> with open('converted.txt', 'wb') as f:
...     f.write(msg.as_bytes(policy=msg.policy.clone(linesep=os.linesep)))
17
```

Policy 对象也可使用加法运算符进行组合来产生一个新策略对象，其设置是被加总对象的非默认值的组合:

```
>>> compat_SMTP = policy.compat32.clone(linesep='\r\n')
>>> compat_strict = policy.compat32.clone(raise_on_defect=True)
>>> compat_strict_SMTP = compat_SMTP + compat_strict
```

此运算不满足交换律；也就是说对象的添加顺序很重要。见以下演示:

```
>>> policy100 = policy.compat32.clone(max_line_length=100)
>>> policy80 = policy.compat32.clone(max_line_length=80)
>>> apolicy = policy100 + policy80
>>> apolicy.max_line_length
80
>>> apolicy = policy80 + policy100
>>> apolicy.max_line_length
100
```

class email.policy.**Policy**(***kw*)

这是所有策略类的*abstract base class*。它提供了一些简单方法的默认实现，以及不可变特征属性，*clone()* 方法以及构造器语义的实现。

可以向策略类的构造器传入各种关键字参数。可以指定的参数是该类的任何非方法特征属性，以及实体类的任何额外非方法特征属性。在构造器中指定的值将覆盖相应属性的默认值。

这个类定义了下列特征属性，因此下列值可以被传给任何策略类的构造器:

max_line_length

序列化输出中任何行的最大长度，不计入行字符的末尾。默认值为 78，基于 **RFC 5322**。值为 0 或*None* 表示完全没有行包装。

linesep

用来在序列化输出中确定行的字符串。默认值为 \n 因为这是 Python 所使用的内部行结束符规范，但 RFC 的要求是 \r\n。

cte_type

控制可能要求使用的内容传输编码格式类型。可能的值包括:

7bit	所有数据必须为" 纯 7 比特位" (仅 ASCII)。这意味着在必要情况下数据将使用可打印引用形式或 base64 编码格式进行编码。
8bit	数据不会被限制为纯 7 比特位。标头中的数据仍要求仅 ASCII 因此将被编码（参阅下文的*fold_binary()* 和*utf8* 了解例外情况），但消息体部分可能使用 8bit CTE。

cte_type 值为 8bit 仅适用于 BytesGenerator 而非 Generator，因为字符串不能包含二进

制数据。如果 Generator 运行于指定了 cte_type=8bit 的策略，它的行为将与 cte_type 为 7bit 相同。

raise_on_defect
如为*True*，则遇到的任何缺陷都将引发错误。如为*False* (默认值)，则缺陷将被传递给*register_defect()* 方法。

mangle_from_
如为*True*，则消息体中以 *"From "* 开头的行会通过在其前面放一个 > 来进行转义。当消息被生成器执行序列化时会使用此形参。默认值 t: *False*。

3.5 版新加入: *mangle_from_* 形参。

message_factory
用来构造新的空消息对象的工厂函数。在构建消息时由解析器使用。默认为 None，在此情况下会使用*Message*。

3.6 版新加入.

下列*Policy* 方法是由使用 email 库的代码来调用以创建具有自室外设置的策略实例:

clone(***kw*)
返回一个新的*Policy* 实例，其属性与当前实例具有相同的值，除非是那些由关键字参数给出了新值的属性。

其余的*Policy* 方法是由 email 包代码来调用的，而不应当被使用 email 包的应用程序所调用。自定义的策略必须实现所有这些方法。

handle_defect(*obj*, *defect*)
处理在 *obj* 上发现的 *defect*。当 email 包调用此方法时，*defect* 将总是 Defect 的一个子类。

默认实现会检查*raise_on_defect* 旗标。如果其为 True，则 *defect* 会被作为异常来引发。如果其为 False (默认值)，则 *obj* 和 *defect* 会被传递给*register_defect()*。

register_defect(*obj*, *defect*)
在 *obj* 上注册一个 *defect*。在 email 包中，*defect* 将总是 Defect 的一个子类。

默认实现会调用 *obj* 的 defects 属性的 append 方法。当 email 包调用*handle_defect* 时，*obj* 通常将具有一个带 append 方法的 defects 属性。配合 email 包使用的自定义对象类型（例如自定义的 Message 对象）也应当提供这样的属性，否则在被解析消息中的缺陷将引发非预期的错误。

header_max_count(*name*)
返回名为 *name* 的标头的最大允许数量。

当添加一个标头到*EmailMessage* 或*Message* 对象时被调用。如果返回值不为 0 或 None，并且已有的名称为 *name* 的标头数量大于等于所返回的值，则会引发*ValueError*。

由于 Message.__setitem__ 的默认行为是将值添加到标头列表，因此很容易不知情地创建重复的标头。此方法允许在程序中限制可以被添加到 Message 中的特定标头的实例数量。(解析器不会考虑此限制，它将忠实地产生被解析消息中存在的任意数量的标头。)

默认实现对于所有标头名称都返回 None。

header_source_parse(*sourcelines*)
email 包调用此方法时将传入一个字符串列表，其中每个字符串以在被解析源中找到的行分隔符结束。第一行包括字段标头名称和分隔符。源中的所有空白符都会被保留。此方法应当返回 (name, value) 元组以保存至 Message 中来代表被解析的标头。

如果一个实现希望保持与现有 email 包策略的兼容性，则 *name* 应当为保留大小写形式的名称（所有字符直至':' 分隔符），而 *value* 应当为展开后的值（移除所有行分隔符，但空白符保持不变），并移除开头的空白符。

sourcelines 可以包含经替代转义的二进制数据。

此方法没有默认实现

header_store_parse(*name*, *value*)
当一个应用通过程序代码修改 Message (而不是由解析器创建 Message) 时，email 包会调用此方法并附带应用程序所提供的名称和值。此方法应当返回 (name, value) 元组以保存至 Message 中用来表示标头。

如果一个实现希望保持与现有 email 包策略的兼容性，则 *name* 和 *value* 应当为字符串或字符串的子类，它们不会修改在参数中传入的内容。

此方法没有默认实现

header_fetch_parse(*name*, *value*)
当标头被应用程序所请求时，email 包会调用此方法并附带当前保存在 Message 中的 *name* 和 *value*，并且无论此方法返回什么它都会被回传给应用程序作为被提取标头的值。请注意可能会有多个相同名称的标头被保存在 Message 中；此方法会将指定标头的名称和值返回给应用程序。

value 可能包含经替代转义的二进制数据。此方法所返回的值应当没有经替代转义的二进制数据。

此方法没有默认实现

fold(*name*, *value*)
email 包调用此方法时会附带当前保存在 Message 中的给定标头的 *name* 和 *value*。此方法应当返回一个代表该标头的（根据策略设置）通过处理 *name* 和 *value* 并在适当位置插入 *linesep* 字符来正确地“折叠”的字符串。请参阅 **RFC 5322** 了解有关折叠电子邮件标头的规则的讨论。

value 可能包含经替代转义的二进制数据。此方法所返回的字符串应当没有经替代转义的二进制数据。

fold_binary(*name*, *value*)
与 *fold()* 类似，不同之处在于返回的值应当为字节串对象而非字符串。

value 可能包含经替代转义的二进制数据。这些数据可以在被返回的字节串对象中被转换回二进制数据。

class email.policy.**EmailPolicy**(***kw*)
这个实体 *Policy* 提供了完全遵循当前电子邮件 RFC 的行为。这包括 (但不限于) **RFC 5322**, **RFC 2047** 以及当前的各种 MIME RFC。

此策略添加了新的标头解析和折叠算法。标头不是简单的字符串，而是带有依赖于字段类型的属性的 str 的子类。这个解析和折叠算法完整实现了 **RFC 2047** 和 **RFC 5322**。

message_factory 属性的默认值为 *EmailMessage*。

除了上面列出的适用于所有策略的可设置属性，此策略还添加了下列额外属性:

3.6 版新加入:[1]

utf8
如为 False，则遵循 **RFC 5322**，通过编码为“已编码字”来支持标头中的非 ASCII 字符。如为 True，则遵循 **RFC 6532** 并对标头使用 utf-8 编码格式。以此方式格式化的消息可被传递给支持 SMTPUTF8 扩展 (**RFC 6531**) 的 SMTP 服务器。

refold_source
如果 Message 对象中标头的值源自 *parser* (而非由程序设置)，此属性会表明当将消息转换回序列化形式时是否应当由生成器来重新折叠该值。可能的值如下:

[1] 最初在 3.3 中作为暂定特性 添加。

none	所有源值使用原始折叠
long	具有任何长度超过 max_line_length 的行的源值将被折叠
all	所有值会被重新折叠。

默认值为 long。

header_factory
该可调用对象接受两个参数，name 和 value，其中 name 为标头字段名而 value 为展开后的标头字段值，并返回一个表示该标头的字符串子类。已提供的默认 header_factory (参见*headerregistry*) 支持对各种地址和日期 **RFC 5322** 标头字段类型及主要 MIME 标头字段类型的自定义解析。未来还将添加对其他自定义解析的支持。

content_manager
此对象至少有两个方法: get_content 和 set_content。当一个*EmailMessage* 对象的*get_content()* 或*set_content()* 方法被调用时，它会调用此对象的相应方法，将消息对象作为其第一个参数，并将传给它的任何参数或关键字作为附加参数传入。默认情况下 content_manager 会被设为*raw_data_manager*。

3.4 版新加入.

这个类提供了下列对*Policy* 的抽象方法的具体实现:

header_max_count(*name*)
返回用来表示具有给定名称的标头的专用类的*max_count* 属性的值。

header_source_parse(*sourcelines*)
此名称会被作为到':' 止的所有内容来解析。该值是通过从第一行的剩余部分去除前导空格，再将所有后续行连接到一起，并去除所有末尾回车符或换行符来确定的。

header_store_parse(*name*, *value*)
name 将会被原样返回。如果输入值具有 name 属性并可在忽略大小写的情况下匹配 *name*，则 value 也会被原样返回。在其他情况下 *name* 和 *value* 会被传递给 header_factory，并将结果标头对象作为值返回。在此情况下如果输入值包含 CR 或 LF 字符则会引发 ValueError。

header_fetch_parse(*name*, *value*)
如果值具有 name 属性，它会被原样返回。在其他情况下 *name* 和移除了所有 CR 和 LF 字符的 *value* 会被传递给 header_factory，并返回结果标头对象。任何经替代转义的字节串会被转换为 unicode 未知字符字形。

fold(*name*, *value*)
标头折叠是由*refold_source* 策略设置来控制的。当且仅当一个值没有 name 属性（具有 name 属性就意味着它是某种标头对象）它才会被当作是 "源值"。如果一个原值需要按照策略来重新折叠，则会通过将 *name* 和去除了所有 CR 和 LF 字符的 *value* 传递给 header_factory 来将其转换为标头对象。标头对象的折叠是通过调用其 fold 方法并附带当前策略来完成的。

源值会使用*splitlines()* 来拆分成多行。如果该值不被重新折叠，则会使用策略中的 linesep 重新合并这些行并将其返回。例外的是包含非 ascii 二进制数据的行。在此情况下无论 refold_source 如何设置该值都会被重新折叠，这会导致二进制数据使用 unknown-8bit 字符集进行 CTE 编码。

fold_binary(*name*, *value*)
如果*cte_type* 为 7bit 则与*fold()* 类似，不同之处在于返回的值是字节串。

如果*cte_type* 为 8bit，则将非 ASCII 二进制数据转换回字节串。带有二进制数据的标头不会被重新折叠，无论 refold_header 设置如何，因为无法知晓该二进制数据是由单字节字符还是多字节字符组成的。

以下*EmailPolicy* 的实例提供了适用于特定应用领域的默认值。请注意在未来这些实例（特别是 HTTP 实例）的行为可能会被调整以便更严格地遵循与其领域相关的 RFC。

email.policy.**default**
一个未改变任何默认值的 EmailPolicy 实例。此策略使用标准的 Python \n 行结束符而非遵循 RFC 的 \r\n。

email.policy.**SMTP**
适用于按照符合电子邮件 RFC 的方式来序列化消息。与 default 类似，但 linesep 被设为遵循 RFC 的 \r\n。

email.policy.**SMTPUTF8**
与 SMTP 类似但是*utf8* 为 True。适用于在不使用标头内已编码字的情况下对消息进行序列化。如果发送方或接收方地址具有非 ASCII 字符则应当只被用于 SMTP 传输 (*smtplib.SMTP.send_message()* 方法会自动如此处理)。

email.policy.**HTTP**
适用于序列化标头以在 HTTP 通信中使用。与 SMTP 类似但是 max_line_length 被设为 None (无限制)。

email.policy.**strict**
便捷实例。与 default 类似但是 raise_on_defect 被设为 True。这样可以允许通过以下写法来严格地设置任何策略:

```
somepolicy + policy.strict
```

因为所有这些*EmailPolicies*，email 包的高效 API 相比 Python 3.2 API 发生了以下几方面变化:

- 在*Message* 中设置标头将使得该标头被解析并创建一个标头对象。
- 从*Message* 提取标头将使得该标头被解析并创建和返回一个标头对象。
- 任何标头对象或任何由于策略设置而被重新折叠的标头都会使用一种完全实现了 RFC 折叠算法的算法来进行折叠，包括知道在休息需要并允许已编码字。

从应用程序的视角来看，这意味着任何通过*EmailMessage* 获得的标头都是具有附加属性的标头对象，其字符串值都是该标头的完全解码后的 unicode 值。类似地，可以使用 unicode 对象为一个标头赋予新的值，或创建一个新的标头对象，并且该策略将负责把该 unicode 字符串转换为正确的 RFC 已编码形式。

标头对象及其属性的描述见*headerregistry*。

class email.policy.**Compat32**(**kw*)
这个实体*Policy* 为向下兼容策略。它复制了 Python 3.2 中 email 包的行为。*policy* 模块还定义了该类的一个实例*compat32*，用来作为默认策略。因此 email 包的默认行为会保持与 Python 3.2 的兼容性。

下列属性具有与*Policy* 默认值不同的值:

mangle_from_
默认值为 True。

这个类提供了下列对*Policy* 的抽象方法的具体实现:

header_source_parse(*sourcelines*)
此名称会被作为到':' 止的所有内容来解析。该值是通过从第一行的剩余部分去除前导空格，再将所有后续行连接到一起，并去除所有末尾回车符或换行符来确定的。

header_store_parse(*name*, *value*)
name 和 value 会被原样返回。

header_fetch_parse(*name*, *value*)
如果 value 包含二进制数据，则会使用 unknown-8bit 字符集来将其转换为*Header* 对象。在其他情况下它会被原样返回。

fold(*name*, *value*)
标头会使用*Header* 折叠算法进行折叠，该算法保留 value 中现有的换行，并将每个结果行的长

度折叠至 max_line_length。非 ASCII 二进制数据会使用 unknown-8bit 字符串进行 CTE 编码。

fold_binary(*name*, *value*)
标头会使用*Header* 折叠算法进行折叠，该算法保留 value 中现有的换行，并将每个结果行的长度折叠至 max_line_length。如果 cte_type 为 7bit，则非 ascii 二进制数据会使用 unknown-8bit 字符集进行 CTE 编码。在其他情况下则会使用原始的源标头，这将保留其现有的换行和所包含的任何（不符合 RFC 的）二进制数据。

email.policy.**compat32**
Compat32 的实例，提供与 Python 3.2 中的 email 包行为的向下兼容性。

脚注

19.1.5 email.errors: 异常和缺陷类

源代码: Lib/email/errors.py

以下异常类在*email.errors* 模块中定义：

exception email.errors.**MessageError**
这是*email* 包可以引发的所有异常的基类。它源自标准异常*Exception* 类，这个类没有定义其他方法。

exception email.errors.**MessageParseError**
这是由*Parser* 类引发的异常的基类。它派生自*MessageError*。*headerregistry* 使用的解析器也在内部使用这个类。

exception email.errors.**HeaderParseError**
在解析消息的 **RFC 5322** 标头时，某些错误条件下会触发，此类派生自*MessageParseError*。如果在调用方法时内容类型未知，则*set_boundary()* 方法将引发此错误。当尝试创建一个看起来包含嵌入式标头的标头时*Header* 可能会针对某些 base64 解码错误引发此错误（也就是说，应该是一个没有前导空格并且看起来像标题的延续行）。

exception email.errors.**BoundaryError**
已弃用和不再使用的。

exception email.errors.**MultipartConversionError**
当使用 add_payload() 将有效负载添加到*Message* 对象时，有效负载已经是一个标量，而消息的 *Content-Type* 主类型不是 *multipart* 或者缺少时触发该异常。*MultipartConversionError* 多重继承自*MessageError* 和内置的*TypeError*。

由于 Message.add_payload() 已被弃用，此异常实际上极少会被引发。但是如果在派生自*MIMENonMultipart* 的类 (例如*MIMEImage*) 的实例上调用*attach()* 方法也可以引发此异常。

以下是*FeedParser* 在解析消息时可发现的缺陷列表。请注意这些缺陷会在问题被发现时加入到消息中，因此举例来说，如果某条嵌套在 *multipart/alternative* 中的消息具有错误的标头，该嵌套消息对象就会有一条缺陷，但外层消息对象则没有。

所有缺陷类都是 email.errors.MessageDefect 的子类。

- NoBoundaryInMultipartDefect -- 一条消息宣称有多个部分，但却没有 *boundary* 形参。
- StartBoundaryNotFoundDefect -- 在 *Content-Type* 标头中宣称的开始边界无法被找到。
- CloseBoundaryNotFoundDefect -- 找到了开始边界，但相应的结束边界无法被找到。

3.3 版新加入.

- `FirstHeaderLineIsContinuationDefect` -- 消息以一个继续行作为其第一个标头行。
- `MisplacedEnvelopeHeaderDefect` - 在标头块中间发现了一个"Unix From" 标头。
- `MissingHeaderBodySeparatorDefect` - 在解析没有前缀空格但又不包含':' 的标头期间找到一行内容。解析将假定该行表示消息体的第一行以继续执行。

  3.3 版新加入.
- `MalformedHeaderDefect` -- 找到一个缺失了冒号或格式错误的标头。

  3.3 版後已🄵用: 此缺陷在近几个 Python 版本中已不再使用。
- `MultipartInvariantViolationDefect` -- 一条消息宣称为 *multipart*，但无法找到任何子部分。请注意当一条消息有此缺陷时，其*is_multipart()* 方法可能返回 `False`，即使其内容类型宣称为 *multipart*。
- `InvalidBase64PaddingDefect` -- 当解码一个 base64 编码的字节分块时，填充的数据不正确。虽然添加了足够的填充数据以执行解码，但作为结果的已解码字节串可能无效。
- `InvalidBase64CharactersDefect` -- 当解码一个 base64 编码的字节分块时，遇到了 base64 字符表以外的字符。这些字符会被忽略，但作为结果的已解码字节串可能无效。
- `InvalidBase64LengthDefect` -- 当解码一个 base64 编码的字节分块时，非填充 base64 字符的数量无效（比 4 的倍数多 1）。已编码分块会保持原样。

19.1.6 email.headerregistry: 自定义标头对象

源代码: Lib/email/headerregistry.py

3.6 版新加入:[1]

标头是由*str* 的自定义子类来表示的。用于表示给定标头的特定类则由创建标头时生效的*policy* 的*header_factory* 确定。这一节记录了 email 包为处理兼容 **RFC 5322** 的电子邮件消息所实现的特定 `header_factory`，它不仅为各种标头类型提供了自定义的标头对象，还为应用程序提供了添加其自定义标头类型的扩展机制。

当使用派生自*EmailPolicy* 的任何策略对象时，所有标头都通过*HeaderRegistry* 产生并且以*BaseHeader* 作为其最后一个基类。每个标头类都有一个由该标头类型确定的附加基类。例如，许多标头都以*UnstructuredHeader* 类作为其另一个基类。一个标头专用的第二个类是由标头名称使用存储在*HeaderRegistry* 中的查找表来确定的。所有这些都针对典型应用程序进行透明的管理，但也为修改默认行为提供了接口，以便由更复杂的应用使用。

以下各节首先记录了标头基类及其属性，然后是用于修改*HeaderRegistry* 行为的 API，最后是用于表示从结构化标头解析的数据的支持类。

class email.headerregistry.**BaseHeader**(*name*, *value*)

name 和 *value* 会从*header_factory* 调用传递给 `BaseHeader`。任何标头对象的字符串值都是完成解码为 unicode 的 *value*。

这个基类定义了下列只读属性:

name

标头的名称（字段在':' 之前的部分）。这就是 *name* 的*header_factory* 调用所传递的值；也就是说会保持大小写形式。

[1] 最初在 3.3 中作为暂定模块 添加

defects
一个包含 HeaderDefect 实例的元组，这些实例报告了在解析期间发现的任何 RFC 合规性问题。email 包会尝试尽可能地检测合规性问题。请参阅*errors* 模块了解可能被报告的缺陷类型的相关讨论。

max_count
此类型标头可具有相同 name 的最大数量。None 值表示无限制。此属性的 BaseHeader 值为 None；专用的标头类预期将根据需要重载这个值。

BaseHeader 还提供了以下方法，它由 email 库代码调用，通常不应当由应用程序来调用。

fold(*, *policy*)
返回一个字符串，其中包含用来根据 *policy* 正确地折叠标头的*linesep* 字符。*cte_type* 为 8bit 时将被作为 7bit 来处理，因为标头不能包含任意二进制数据。如果*utf8* 为 False，则非 ASCII 数据将根据 **RFC 2047** 来编码。

BaseHeader 本身不能被用于创建标头对象。它定义了一个与每个专用标头相配合的协议以便生成标头对象。具体来说，BaseHeader 要求专用类提供一个名为 parse 的*classmethod()*。此方法的调用形式如下:

```
parse(string, kwds)
```

kwds 是包含了一个预初始化键 defects 的字典。defects 是一个空列表。parse 方法应当将任何已检测到的缺陷添加到此列表中。在返回时，kwds 字典 *必须*至少包含 decoded 和 defects 等键的值。decoded 应当是标头的字符串值（即完全解码为 unicode 的标头值）。parse 方法应当假定 *string* 可能包含 content-transfer-encoded 部分，但也应当正确地处理全部有效的 unicode 字符以便它能解析未经编码的标头值。

随后 BaseHeader 的 __new__ 会创建标头实例，并调用其 init 方法。专属类如果想要设置 BaseHeader 自身所提供的属性之外的附加属性，只需提供一个 init 方法。这样的 init 看起来应该是这样:

```
def init(self, /, *args, **kw):
    self._myattr = kw.pop('myattr')
    super().init(*args, **kw)
```

也就是说，专属类放入 kwds 字典的任何额外内容都应当被移除和处理，并且 kw (和 args) 的剩余内容会被传递给 BaseHeader init 方法。

class email.headerregistry.**UnstructuredHeader**
" 非结构化" 标头是 **RFC 5322** 中默认的标头类型。任何没有指定语法的标头都会被视为是非结构化的。非结构化标头的经典例子是 *Subject* 标头。

在 **RFC 5322** 中，非结构化标头是指一段以 ASCII 字符集表示的任意文本。但是 **RFC 2047** 具有一个 **RFC 5322** 兼容机制用来将标头值中的非 ASCII 文本编码为 ASCII 字符。当包含已编码字的 *value* 被传递给构造器时，UnstructuredHeader 解析器会按照非结构化文本的 **RFC 2047** 规则将此类已编码字转换为 unicode。解析器会使用启发式机制来尝试解码一些不合规的已编码字。在此种情况下各类缺陷，例如已编码字或未编码文本中的无效字符问题等缺陷将会被注册。

此标头类型未提供附加属性。

class email.headerregistry.**DateHeader**
RFC 5322 为电子邮件标头内的日期指定了非常明确的格式。DateHeader 解析器会识别该日期格式，并且也能识别间或出现的一些“不规范”变种形式。

这个标头类型提供了以下附加属性。

datetime
如果标头值能被识别为某一种有效的日期形式，此属性将包含一个代表该日期的*datetime* 实例。如果输入日期的时区被指定为 -0000 (表示为 UTC 但不包含源时区的相关信息)，则*datetime*

将为简单型*datetime*。如果找到了特定的时区时差值 (包括 *+0000*)，则*datetime* 将包含使用*datetime.timezone* 来记录时区时差时的感知型 datetime。

标头的 decoded 值是由按照 **RFC 5322** 对 datetime 进行格式化来确定的；也就是说，它会被设为:

```
email.utils.format_datetime(self.datetime)
```

当创建 DateHeader 时，*value* 可以为*datetime* 实例。例如这意味着以下代码是有效的并能实现人们预期的行为:

```
msg['Date'] = datetime(2011, 7, 15, 21)
```

因为这是个简单型 datetime 它将被解读为 UTC 时间戳，并且结果值的时区将为 -0000。使用来自*utils* 模块的*localtime()* 函数会更有用:

```
msg['Date'] = utils.localtime()
```

这个例子将日期标头设为使用当前时区时差值的当前时间和日期。

class email.headerregistry.**AddressHeader**

地址标头是最复杂的结构化标头类型之一。AddressHeader 类提供了适合任何地址标头的泛用型接口。

这个标头类型提供了以下附加属性。

groups

编码了在标头值中找到的地址和分组的*Group* 对象的元组。非分组成员的地址在此列表中表示为*display_name* 为 None 的单地址 Groups。

addresses

编码了来自标头值的所有单独地址的*Address* 对象的元组。如果标头值包含任何分组，则来自分组的单个地址将包含在该分组出现在值中的点上列出（也就是说，地址列表会被“展平”为一维列表）。

标头的 decoded 值将为所有已编码字解码为 unicode 的结果。*idna* 编码的域名也将被解码为 unicode。decoded 值是通过对 groups 属性的元素的*str* 值使用 ', ' 进行*join* 来设置的。

可以使用*Address* 与*Group* 对象的任意组合的列表来设置一个地址标头的值。display_name 为 None 的 Group 对象将被解读为单独地址，这允许一个地址列表可以附带通过使用从源标头的 groups 属性获取的列表而保留原分组。

class email.headerregistry.**SingleAddressHeader**

AddressHeader 的子类，添加了一个额外的属性:

address

由标头值编码的单个地址。如果标头值实际上包含一个以上的地址（这在默认*policy* 下将违反 RFC），则访问此属性将导致*ValueError*。

上述类中许多还具有一个 Unique 变体 (例如 UniqueUnstructuredHeader)。其唯一差别是在 Unique 变体中*max_count* 被设为 1。

class email.headerregistry.**MIMEVersionHeader**

实际上 *MIME-Version* 标头只有一个有效的值，即 1.0。为了将来的扩展，这个标头类还支持其他的有效版本号。如果一个版本号是 **RFC 2045** 的有效值，则标头对象的以下属性将具有不为 None 的值:

version

字符串形式的版本号。任何空格和/或注释都会被移除。

major

整数形式的主版本号

minor
整数形式的次版本号

class email.headerregistry.**ParameterizedMIMEHeader**
MIME 标头都以前缀'Content-' 打头。每个特定标头都具有特定的值，其描述在该标头的类之中。有些也可以接受一个具有通用格式的补充形参形表。这个类被用作所有接受形参的 MIME 标头的基类。

params
一个将形参名映射到形参值的字典。

class email.headerregistry.**ContentTypeHeader**
处理 *Content-Type* 标头的*ParameterizedMIMEHeader* 类。

content_type
maintype/subtype 形式的内容类型字符串。

maintype

subtype

class email.headerregistry.**ContentDispositionHeader**
处理 *Content-Disposition* 标头的*ParameterizedMIMEHeader* 类。

content_disposition
inline 和 attachment 是仅有的常用有效值。

class email.headerregistry.**ContentTransferEncoding**
处理 *Content-Transfer-Encoding* 标头。

cte
可用的有效值为 7bit, 8bit, base64 和 quoted-printable。更多信息请参阅 **RFC 2045**。

class email.headerregistry.**HeaderRegistry**(*base_class=BaseHeader*, *default_class=UnstructuredHeader*, *use_default_map=True*)
这是由*EmailPolicy* 在默认情况下使用的工厂函数。HeaderRegistry 会使用 *base_class* 和从它所保存的注册表中获取的专用类来构建用于动态地创建标头实例的类。当给定的标头名称未在注册表中出现时，则会使用由 *default_class* 所指定的类作为专用类。当 *use_default_map* 为 True (默认值) 时，则会在初始化期间把将标头名称与类的标准映射拷贝到注册表中。*base_class* 始终会是所生成类的 __bases__ 列表中的最后一个类。

默认的映射有:

subject UniqueUnstructuredHeader

date UniqueDateHeader

resent-date DateHeader

orig-date UniqueDateHeader

sender UniqueSingleAddressHeader

resent-sender SingleAddressHeader

到 UniqueAddressHeader

resent-to AddressHeader

cc UniqueAddressHeader

resent-cc AddressHeader

bcc UniqueAddressHeader

resent-bcc AddressHeader

从 UniqueAddressHeader

resent-from AddressHeader

reply-to UniqueAddressHeader

mime-version MIMEVersionHeader

content-type ContentTypeHeader

content-disposition ContentDispositionHeader

content-transfer-encoding ContentTransferEncodingHeader

message-id MessageIDHeader

`HeaderRegistry` 具有下列方法:

map_to_type(*self*, *name*, *cls*)
name 是要映射的标头名称。它将在注册表中被转换为小写形式。*cls* 是要与 *base_class* 一起被用来创建用于实例化与 *name* 相匹配的标头的类的专用类。

__getitem__(*name*)
构造并返回一个类来处理 *name* 标头的创建。

__call__(*name*, *value*)
从注册表获得与 *name* 相关联的专用标头 (如果 *name* 未在注册表中出现则使用 *default_class*) 并将其与 *base_class* 相组合以产生类，调用被构造类的构造器，传入相同的参数列表，并最终返回由此创建的类实例。

以下的类是用于表示从结构化标头解析的数据的类，并且通常会由应用程序使用以构造结构化的值并赋给特定的标头。

class email.headerregistry.**Address**(*display_name=''*, *username=''*, *domain=''*, *addr_spec=None*)
用于表示电子邮件地址的类。地址的一般形式为:

```
[display_name] <username@domain>
```

或是:

```
username@domain
```

其中每个部分都必须符合在 **RFC 5322** 中阐述的特定语法规则。

为了方便起见可以指定 *addr_spec* 来替代 *username* 和 *domain*，在此情况下 *username* 和 *domain* 将从 *addr_spec* 中解析。*addr_spec* 应当是一个正确地引用了 RFC 的字符串；如果它不是 `Address` 则将引发错误。Unicode 字符也允许使用并将在序列化时被正确地编码。但是，根据 RFC，地址的 username 部分 不允许有 unicode。

display_name
地址的显示名称部分（如果有的话）并去除所有引用项。如果地址没有显示名称，则此属性将为空字符串。

username
地址的 `username` 部分，去除所有引用项。

domain
地址的 `domain` 部分。

addr_spec
地址的 `username@domain` 部分，经过正确引用处理以作为纯地址使用（上面显示的第二种形式）。此属性不可变。

__str__()
对象的 str 值是根据 **RFC 5322** 规则进行引用处理的地址，但不带任何非 ASCII 字符的 Content Transfer Encoding。

为了支持 SMTP (**RFC 5321**)，Address 会处理一种特殊情况：如果 username 和 domain 均为空字符串 (或为 None)，则 Address 的字符串值为 <>。

class email.headerregistry.**Group**(*display_name=None*, *addresses=None*)
用于表示地址组的类。地址组的一般形式为:

```
display_name: [address-list];
```

作为处理由组和单个地址混合构成的列表的便捷方式，Group 也可以通过将 *display_name* 设为 None 以用来表示不是某个组的一部分的单独地址并提供单独地址的列表作为 *addresses*。

display_name
组的 display_name。如果其为 None 并且恰好有一个 Address 在 addresses 中，则 Group 表示一个不在某个组中的单独地址。

addresses
一个可能为空的表示组中地址的包含*Address* 对象的元组。

__str__()
Group 的 str 值会根据 **RFC 5322** 进行格式化，但不带任何非 ASCII 字符的 Content Transfer Encoding。如果 display_name 为空值且只有一个单独 Address 在 addresses 列表中，则 str 值将与该单独 Address 的 str 相同。

脚注

19.1.7 email.contentmanager: 管理 MIME 内容

源代码: Lib/email/contentmanager.py

3.6 版新加入:[1]

class email.contentmanager.**ContentManager**
内容管理器的基类。提供注册 MIME 内容和其他表示形式间转换器的标准注册机制，以及 get_content 和 set_content 发送方法。

get_content(*msg*, **args*, ***kw*)
基于 *msg* 的 mimetype 查找处理函数（参见下一段），调用该函数，传递所有参数，并返回调用的结果。预期的效果是处理程序将从 *msg* 中提取有效载荷并返回编码了有关被提取数据信息的对象。

要找到处理程序，将在注册表中查找以下键，找到第一个键即停止:

- 表示完整 MIME 类型的字符串 (maintype/subtype)
- 表示 maintype 的字符串
- 空字符串

如果这些键都没有产生处理程序，则为完整 MIME 类型引发一个*KeyError*。

set_content(*msg*, *obj*, **args*, ***kw*)
如果 maintype 为 multipart，则引发*TypeError*；否则基于 *obj* 的类型（参见下一段）查找处理函数，在 *msg* 上调用*clear_content()*，并调用处理函数，传递所有参数。预期的效果是

[1] 最初在 3.4 中作为暂定模块 添加

处理程序将转换 *obj* 并存入 *msg*，并可能对 *msg* 进行其他更改，例如添加各种 MIME 标头来编码需要用来解释所存储数据的信息。

要找到处理程序，将获取 *obj* 的类型 (`typ = type(obj)`)，并在注册表中查找以下键，找到第一个键即停止:

- 类型本身 (`typ`)
- 类型的完整限定名称 (`typ.__module__ + '.' + typ.__qualname__`)。
- 类型的 qualname (`typ.__qualname__`)
- 类型的 name (`typ.__name__`)。

如果未匹配到上述的任何一项，则在*MRO* (`typ.__mro__`) 中为每个类型重复上述的所有检测。最后，如果没有其他键产生处理程序，则为 `None` 键检测处理程序。如果也没有 `None` 的处理程序，则为该类型的完整限定名称引发*KeyError*。

并会添加一个 *MIME-Version* 标头，如果没有的话 (另请参见*MIMEPart*)。

add_get_handler(*key*, *handler*)
将 *handler* 函数记录为 *key* 的处理程序。对于可能的 *key* 键，请参阅*get_content()*。

add_set_handler(*typekey*, *handler*)
将 *handler* 记录为当一个匹配 *typekey* 的类型对象被传递给*set_content()* 时所要调用的函数。对于可能的 *typekey* 值，请参阅*set_content()*。

内容管理器实例

目前 email 包只提供了一个实体内容管理器*raw_data_manager*，不过在未来可能会添加更多。*raw_data_manager* 是由*EmailPolicy* 及其衍生工具所提供的*content_manager*。

email.contentmanager.**raw_data_manager**
这个内容管理器仅提供了超出*Message* 本身提供内容的最小接口：它只处理文本、原始字节串和*Message* 对象。不过相比基础 API，它具有显著的优势：在文本部分上执行 `get_content` 将返回一个 unicode 字符串而无需由应用程序来手动解码，`set_content` 为控制添加到一个部分的标头和控制内容传输编码格式提供了丰富的选项集合，并且它还启用了多种 `add_` 方法，从而简化了多部分消息的创建过程。

email.contentmanager.**get_content**(*msg*, *errors='replace'*)
将指定部分的有效载荷作为字符串（对于 `text` 部分），*EmailMessage* 对象（对于 `message/rfc822` 部分）或 `bytes` 对象（对于所有其他非多部分类型）返回。如果是在 `multipart` 上调用则会引发*KeyError*。如果指定部分是一个 `text` 部分并且指明了 *errors*，则会在将载荷解码为 unicode 时将其用作错误处理程序。默认的错误处理程序是 `replace`。

email.contentmanager.**set_content**(*msg*, *<'str'>*, *subtype="plain"*, *charset='utf-8'*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
email.contentmanager.**set_content**(*msg*, *<'bytes'>*, *maintype*, *subtype*, *cte="base64"*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
email.contentmanager.**set_content**(*msg*, *<'EmailMessage'>*, *cte=None*, *disposition=None*, *filename=None*, *cid=None*, *params=None*, *headers=None*)
向 *msg* 添加标头和有效载荷:

添加一个带有 `maintype/subtype` 值的 *Content-Type* 标头。

- 对于 `str`，将 MIME `maintype` 设为 `text`，如果指定了子类型 *subtype* 则设为指定值，否则设为 `plain`。
- 对于 `bytes`，将使用指定的 *maintype* 和 *subtype*，如果未指定则会引发*TypeError*。

- 对于*EmailMessage*对象，将 maintype 设为 message，并将指定的 subtype 设为 *subtype*，如果未指定则设为 rfc822。如果 *subtype* 为 partial，则引发一个错误（必须使用 bytes 对象来构造 message/partial 部分）。

如果提供了 *charset* (这只对 str 适用)，则使用指定的字符集将字符串编码为字节串。默认值为 utf-8。如果指定的 *charset* 是某个标准 MIME 字符集名称的已知别名，则会改用该标准字符集。

如果设置了 *cte*，则使用指定的内容传输编码格式对有效载荷进行编码，并将 *Content-Transfer-Encoding* 标头设为该值。可能的 *cte* 值有 quoted-printable, base64, 7bit, 8bit 和 binary。如果输入无法以指定的编码格式被编码 (例如，对于包含非 ASCII 值的输入指定 *cte* 值为 7bit)，则会引发*ValueError*。

- 对于 str 对象，如果 *cte* 未设置则会使用启发方式来确定最紧凑的编码格式。
- 对于*EmailMessage*，按照 **RFC 2046**，如果为 *subtype* rfc822 请求的 *cte* 为 quoted-printable 或 base64，而为 7bit 以外的任何 *cte* 为 *subtype* external-body 则会引发一个错误。对于 message/rfc822，如果 *cte* 未指定则会使用 8bit。对于所有其他 *subtype* 值，都会使用 7bit。

備𝔽： *cte* 值为 binary 实际上还不能正确工作。由 set_content 所修改的 EmailMessage 对象是正确的，但*BytesGenerator*不会正确地将其序列化。

如果设置了 *disposition*，它会被用作 *Content-Disposition* 标头的值。如果未设置，并且指定了 *filename*，则添加值为 attachment 的标头。如果未设置 *disposition* 并且也未指定 *filename*，则不添加标头。*disposition* 的有效值仅有 attachment 和 inline。

如果设置了 *filename*，则将其用作 *Content-Disposition* 标头的 filename 参数的值。

如果设置了 *cid*，则添加一个 *Content-ID* 标头并将其值设为 *cid*。

如果设置了 *params*，则迭代其 items 方法并使用输出的 (key, value) 结果对在 *Content-Type* 标头上设置附加参数。

如果设置了 *headers* 并且为 headername: headervalue 形式的字符串的列表或为 header 对象的列表（通过一个 name 属性与字符串相区分），则将标头添加到 *msg*。

脚注

19.1.8 email: 示例

以下是一些如何使用*email*包来读取、写入和发送简单电子邮件以及更复杂的 MIME 邮件的示例。

首先，让我们看看如何创建和发送简单的文本消息（文本内容和地址都可能包含 unicode 字符）：

```
# Import smtplib for the actual sending function
import smtplib

# Import the email modules we'll need
from email.message import EmailMessage

# Open the plain text file whose name is in textfile for reading.
with open(textfile) as fp:
    # Create a text/plain message
    msg = EmailMessage()
    msg.set_content(fp.read())

# me == the sender's email address
```

(下页继续)

(繼續上一頁)

```
# you == the recipient's email address
msg['Subject'] = f'The contents of {textfile}'
msg['From'] = me
msg['To'] = you

# Send the message via our own SMTP server.
s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

解析 **RFC 822** 标题可以通过使用*parser* 模块中的类来轻松完成：

```
# Import the email modules we'll need
from email.parser import BytesParser, Parser
from email.policy import default

# If the e-mail headers are in a file, uncomment these two lines:
# with open(messagefile, 'rb') as fp:
#     headers = BytesParser(policy=default).parse(fp)

#  Or for parsing headers in a string (this is an uncommon operation), use:
headers = Parser(policy=default).parsestr(
        'From: Foo Bar <user@example.com>\n'
        'To: <someone_else@example.com>\n'
        'Subject: Test message\n'
        '\n'
        'Body would go here\n')

#  Now the header items can be accessed as a dictionary:
print('To: {}'.format(headers['to']))
print('From: {}'.format(headers['from']))
print('Subject: {}'.format(headers['subject']))

# You can also access the parts of the addresses:
print('Recipient username: {}'.format(headers['to'].addresses[0].username))
print('Sender name: {}'.format(headers['from'].addresses[0].display_name))
```

以下是如何发送包含可能在目录中的一系列家庭照片的 MIME 消息示例：

```
# Import smtplib for the actual sending function
import smtplib

# And imghdr to find the types of our images
import imghdr

# Here are the email package modules we'll need
from email.message import EmailMessage

# Create the container email message.
msg = EmailMessage()
msg['Subject'] = 'Our family reunion'
# me == the sender's email address
# family = the list of all recipients' email addresses
msg['From'] = me
msg['To'] = ', '.join(family)
msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'
```

(下页继续)

(繼續上一頁)

```
# Open the files in binary mode.  Use imghdr to figure out the
# MIME subtype for each specific image.
for file in pngfiles:
    with open(file, 'rb') as fp:
        img_data = fp.read()
    msg.add_attachment(img_data, maintype='image',
                                 subtype=imghdr.what(None, img_data))

# Send the email via our own SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

以下是如何将目录的全部内容作为电子邮件消息发送的示例:[1]

```
#!/usr/bin/env python3

"""Send the contents of a directory as a MIME message."""

import os
import smtplib
# For guessing MIME type based on file name extension
import mimetypes

from argparse import ArgumentParser

from email.message import EmailMessage
from email.policy import SMTP


def main():
    parser = ArgumentParser(description="""\
Send the contents of a directory as a MIME message.
Unless the -o option is given, the email is sent by forwarding to your local
SMTP server, which then does the normal delivery process.  Your local machine
must be running an SMTP server.
""")
    parser.add_argument('-d', '--directory',
                        help="""Mail the contents of the specified directory,
                        otherwise use the current directory.  Only the regular
                        files in the directory are sent, and we don't recurse to
                        subdirectories.""")
    parser.add_argument('-o', '--output',
                        metavar='FILE',
                        help="""Print the composed message to FILE instead of
                        sending the message to the SMTP server.""")
    parser.add_argument('-s', '--sender', required=True,
                        help='The value of the From: header (required)')
    parser.add_argument('-r', '--recipient', required=True,
                        action='append', metavar='RECIPIENT',
                        default=[], dest='recipients',
                        help='A To: header value (at least one required)')
    args = parser.parse_args()
    directory = args.directory
    if not directory:
```

(下页继续)

[1] 感谢 Matthew Dixon Cowles 提供最初的灵感和示例。

(繼續上一頁)

```
        directory = '.'
    # Create the message
    msg = EmailMessage()
    msg['Subject'] = f'Contents of directory {os.path.abspath(directory)}'
    msg['To'] = ', '.join(args.recipients)
    msg['From'] = args.sender
    msg.preamble = 'You will not see this in a MIME-aware mail reader.\n'

    for filename in os.listdir(directory):
        path = os.path.join(directory, filename)
        if not os.path.isfile(path):
            continue
        # Guess the content type based on the file's extension.  Encoding
        # will be ignored, although we should check for simple things like
        # gzip'd or compressed files.
        ctype, encoding = mimetypes.guess_type(path)
        if ctype is None or encoding is not None:
            # No guess could be made, or the file is encoded (compressed), so
            # use a generic bag-of-bits type.
            ctype = 'application/octet-stream'
        maintype, subtype = ctype.split('/', 1)
        with open(path, 'rb') as fp:
            msg.add_attachment(fp.read(),
                               maintype=maintype,
                               subtype=subtype,
                               filename=filename)
    # Now send or store the message
    if args.output:
        with open(args.output, 'wb') as fp:
            fp.write(msg.as_bytes(policy=SMTP))
    else:
        with smtplib.SMTP('localhost') as s:
            s.send_message(msg)


if __name__ == '__main__':
    main()
```

以下是如何将上述 MIME 消息解压缩到文件目录中的示例：

```
#!/usr/bin/env python3

"""Unpack a MIME message into a directory of files."""

import os
import email
import mimetypes

from email.policy import default

from argparse import ArgumentParser


def main():
    parser = ArgumentParser(description="""\
Unpack a MIME message into a directory of files.
```

(下页继续)

(繼續上一頁)

```
""")
    parser.add_argument('-d', '--directory', required=True,
                        help="""Unpack the MIME message into the named
                        directory, which will be created if it doesn't already
                        exist.""")
    parser.add_argument('msgfile')
    args = parser.parse_args()

    with open(args.msgfile, 'rb') as fp:
        msg = email.message_from_binary_file(fp, policy=default)

    try:
        os.mkdir(args.directory)
    except FileExistsError:
        pass

    counter = 1
    for part in msg.walk():
        # multipart/* are just containers
        if part.get_content_maintype() == 'multipart':
            continue
        # Applications should really sanitize the given filename so that an
        # email message can't be used to overwrite important files
        filename = part.get_filename()
        if not filename:
            ext = mimetypes.guess_extension(part.get_content_type())
            if not ext:
                # Use a generic bag-of-bits extension
                ext = '.bin'
            filename = f'part-{counter:03d}{ext}'
        counter += 1
        with open(os.path.join(args.directory, filename), 'wb') as fp:
            fp.write(part.get_payload(decode=True))


if __name__ == '__main__':
    main()
```

以下是如何使用备用纯文本版本创建 HTML 消息的示例。为了让事情变得更有趣，我们在 html 部分中包含了一个相关的图像，我们保存了一份我们要发送的内容到硬盘中，然后发送它。

```
#!/usr/bin/env python3

import smtplib

from email.message import EmailMessage
from email.headerregistry import Address
from email.utils import make_msgid

# Create the base text message.
msg = EmailMessage()
msg['Subject'] = "Ayons asperges pour le déjeuner"
msg['From'] = Address("Pepé Le Pew", "pepe", "example.com")
msg['To'] = (Address("Penelope Pussycat", "penelope", "example.com"),
             Address("Fabrette Pussycat", "fabrette", "example.com"))
msg.set_content("""\
```

(下页继续)

(繼續上一頁)

```
Salut!

Cela ressemble à un excellent recipie[1] déjeuner.

[1] http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718

--Pepé
""")

# Add the html version.  This converts the message into a multipart/alternative
# container, with the original text message as the first part and the new html
# message as the second part.
asparagus_cid = make_msgid()
msg.add_alternative("""\
<html>
  <head></head>
  <body>
    <p>Salut!</p>
    <p>Cela ressemble à un excellent
        <a href="http://www.yummly.com/recipe/Roasted-Asparagus-Epicurious-203718">
            recipie
        </a> déjeuner.
    </p>
    <img src="cid:{asparagus_cid}" />
  </body>
</html>
""".format(asparagus_cid=asparagus_cid[1:-1]), subtype='html')
# note that we needed to peel the <> off the msgid for use in the html.

# Now add the related image to the html part.
with open("roasted-asparagus.jpg", 'rb') as img:
    msg.get_payload()[1].add_related(img.read(), 'image', 'jpeg',
                                     cid=asparagus_cid)

# Make a local copy of what we are going to send.
with open('outgoing.msg', 'wb') as f:
    f.write(bytes(msg))

# Send the message via local SMTP server.
with smtplib.SMTP('localhost') as s:
    s.send_message(msg)
```

如果我们发送最后一个示例中的消息，这是我们可以处理它的一种方法：

```
import os
import sys
import tempfile
import mimetypes
import webbrowser

# Import the email modules we'll need
from email import policy
from email.parser import BytesParser

# An imaginary module that would make this work and be safe.
from imaginary import magic_html_parser
```

(下页继续)

(繼續上一頁)

```
# In a real program you'd get the filename from the arguments.
with open('outgoing.msg', 'rb') as fp:
    msg = BytesParser(policy=policy.default).parse(fp)

# Now the header items can be accessed as a dictionary, and any non-ASCII will
# be converted to unicode:
print('To:', msg['to'])
print('From:', msg['from'])
print('Subject:', msg['subject'])

# If we want to print a preview of the message content, we can extract whatever
# the least formatted payload is and print the first three lines.  Of course,
# if the message has no plain text part printing the first three lines of html
# is probably useless, but this is just a conceptual example.
simplest = msg.get_body(preferencelist=('plain', 'html'))
print()
print(''.join(simplest.get_content().splitlines(keepends=True)[:3]))

ans = input("View full message?")
if ans.lower()[0] == 'n':
    sys.exit()

# We can extract the richest alternative in order to display it:
richest = msg.get_body()
partfiles = {}
if richest['content-type'].maintype == 'text':
    if richest['content-type'].subtype == 'plain':
        for line in richest.get_content().splitlines():
            print(line)
        sys.exit()
    elif richest['content-type'].subtype == 'html':
        body = richest
    else:
        print("Don't know how to display {}".format(richest.get_content_type()))
        sys.exit()
elif richest['content-type'].content_type == 'multipart/related':
    body = richest.get_body(preferencelist=('html'))
    for part in richest.iter_attachments():
        fn = part.get_filename()
        if fn:
            extension = os.path.splitext(part.get_filename())[1]
        else:
            extension = mimetypes.guess_extension(part.get_content_type())
        with tempfile.NamedTemporaryFile(suffix=extension, delete=False) as f:
            f.write(part.get_content())
            # again strip the <> to go from email form of cid to html form.
            partfiles[part['content-id'][1:-1]] = f.name
else:
    print("Don't know how to display {}".format(richest.get_content_type()))
    sys.exit()
with tempfile.NamedTemporaryFile(mode='w', delete=False) as f:
    # The magic_html_parser has to rewrite the href="cid:...." attributes to
    # point to the filenames in partfiles.  It also has to do a safety-sanitize
    # of the html.  It could be written using html.parser.
    f.write(magic_html_parser(body.get_content(), partfiles))
```

(下页继续)

(繼續上一頁)

```
webbrowser.open(f.name)
os.remove(f.name)
for fn in partfiles.values():
    os.remove(fn)

# Of course, there are lots of email messages that could break this simple
# minded program, but it will handle the most common ones.
```

直到输出提示，上面的输出是：

```
To: Penelope Pussycat <penelope@example.com>, Fabrette Pussycat <fabrette@example.com>
From: Pepé Le Pew <pepe@example.com>
Subject: Ayons asperges pour le déjeuner

Salut!

Cela ressemble à un excellent recipie[1] déjeuner.
```

脚注

旧式 API:

19.1.9 email.message.Message: 使用 compat32 API 来表示电子邮件消息

Message 类与*EmailMessage* 类非常相似，但没有该类所添加的方法，并且某些方法的默认行为也略有不同。我们还在这里记录了一些虽然被*EmailMessage* 类所支持但并不推荐的方法，除非你是在处理旧有代码。

在其他情况下这两个类的理念和结构都是相同的。

本文档描述了默认 (对于*Message*) 策略*Compat32* 之下的行为。如果你要使用其他策略，你应当改用*EmailMessage* 类。

电子邮件消息由多个 标头和一个 载荷组成。标头必须为 **RFC 5322** 风格的名称和值，其中字典名和值由冒号分隔。冒号不是字段名或字段值的组成部分。载荷可以是简单的文本消息，或是二进制对象，或是多个子消息的结构化序列，每个子消息都有自己的标头集合和自己的载荷。后一种类型的载荷是由具有 *multipart/** 或 *message/rfc822* 等 MIME 类型的消息来指明的。

Message 对象所提供了概念化模型是由标头组成的有序字典，加上用于访问标头中的特殊信息以及访问载荷的额外方法，以便能生成消息的序列化版本，并递归地遍历对象树。请注意重复的标头是受支持的，但必须使用特殊的方法来访问它们。

Message 伪字典以标头名作为索引，标头名必须为 ASCII 值。字典的值为应当只包含 ASCII 字符的字符串；对于非 ASCII 输入有一些特殊处理，但这并不总能产生正确的结果。标头以保留原大小写的形式存储和返回，但字段名称匹配对大小写不敏感。还可能会有一个单独的封包标头，也称 *Unix-From* 标头或 `From_` 标头。载荷对于简单消息对象的情况是一个字符串或字节串，对于 MIME 容器文档的情况 (例如 *multipart/** 和 *message/rfc822*) 则是一个*Message* 对象。

以下是*Message* 类的方法:

class email.message.**Message**(*policy=compat32*)

如果指定了 *policy* (它必须为*policy* 类的实例) 则使用它所设置的规则来更新和序列化消息的表示形式。如果未设置 *policy*，则使用*compat32* 策略，该策略会保持对 Python 3.2 版 email 包的向下兼容性。更多信息请参阅*policy* 文档。

3.3 版更變: 增加了 *policy* 关键字参数。

as_string(*unixfrom=False*, *maxheaderlen=0*, *policy=None*)
以展平的字符串形式返回整个消息对象。或可选的 *unixfrom* 为真值，返回的字符串会包括封包标头。*unixfrom* 的默认值是 `False`。出于保持向下兼容性的原因，*maxheaderlen* 的默认值是 `0`，因此如果你想要不同的值你必须显式地重载它（在策略中为 *max_line_length* 指定的值将被此方法忽略）。*policy* 参数可被用于覆盖从消息实例获取的默认策略。这可以用来对该方法所输出的格式进行一些控制，因为指定的 *policy* 将被传递给 `Generator`。

如果需要填充默认值以完成对字符串的转换则展平消息可能触发对*Message* 的修改（例如，MIME 边界可能会被生成或被修改）。

请注意此方法是出于便捷原因提供的，可能无法总是以你想要的方式格式化消息。例如，在默认情况下它不会按 unix mbox 格式的要求对以 `From` 打头的行执行调整。为了获得更高灵活性，请实例化一个*Generator* 实例并直接使用其*flatten()* 方法。例如:

```
from io import StringIO
from email.generator import Generator
fp = StringIO()
g = Generator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

如果消息对象包含未按照 RFC 标准进行编码的二进制数据，则这些不合规数据将被 unicode "unknown character" 码位值所替代。（另请参阅*as_bytes()* 和*BytesGenerator*。）

3.4 版更變: 增加了 *policy* 关键字参数。

__str__()
与*as_string()* 等价。这将让 `str(msg)` 产生一个包含已格式化消息的字符号。

as_bytes(*unixfrom=False*, *policy=None*)
以字节串对象的形式返回整个扁平化后的消息。当可选的 *unixfrom* 为真值时，返回的字符串会包括封包标头。*unixfrom* 的默认值为 `False`。*policy* 参数可被用于重载从消息实例获取的默认策略。这可被用来控制该方法所产生的部分格式化效果，因为指定的 *policy* 将被传递给 `BytesGenerator`。

如果需要填充默认值以完成对字符串的转换则展平消息可能触发对*Message* 的修改（例如，MIME 边界可能会被生成或被修改）。

请注意此方法是出于便捷原因提供的，可能无法总是以你想要的方式格式化消息。例如，在默认情况下它不会按 unix mbox 格式的要求对以 `From` 打头的行执行调整。为了获得更高灵活性，请实例化一个*BytesGenerator* 实例并直接使用其*flatten()* 方法。例如:

```
from io import BytesIO
from email.generator import BytesGenerator
fp = BytesIO()
g = BytesGenerator(fp, mangle_from_=True, maxheaderlen=60)
g.flatten(msg)
text = fp.getvalue()
```

3.4 版新加入.

__bytes__()
与*as_bytes()* 等价。这将让 `bytes(msg)` 产生一个包含已格式化消息的字节串对象。

3.4 版新加入.

is_multipart()
如果该消息的载荷是一个子*Message* 对象列表则返回 `True`，否则返回 `False`。当*is_multipart()* 返回 `False` 时，载荷应当是一个字符串对象（有可能是一个 CTE 编码的二进制载荷）。（请注意*is_multipart()* 返回 `True` 并不意味着"msg.get_content_maintype() ==

'multipart'" 将返回 True。例如，is_multipart 在*Message* 类型为 message/rfc822 时也将返回 True。)

set_unixfrom(*unixfrom*)
将消息的封包标头设为 *unixfrom*，这应当是一个字符串。

get_unixfrom()
返回消息的信封头。如果信封头从未被设置过，默认返回 None 。

attach(*payload*)
将给定的 *payload* 添加到当前载荷中，当前载荷在该调用之前必须为 None 或是一个*Message* 对象列表。在调用之后，此载荷将总是一个*Message* 对象列表。如果你想将此载荷设为一个标量对象（如字符串），请改用*set_payload()*。

这是一个过时的方法。在 EmailMessage 类上它的功能已被*set_content()* 及相应的 make 和 add 方法所替代。

get_payload(*i=None*, *decode=False*)
返回当前的载荷，它在*is_multipart()* 为 True 时将是一个*Message* 对象列表，在*is_multipart()* 为 False 时则是一个字符串。如果该载荷是一个列表且你修改了这个列表对象，那么你就是原地修改了消息的载荷。

传入可选参数 *i* 时，如果*is_multipart()* 为 True，*get_payload()* 将返回载荷从零开始计数的第 *i* 个元素。如果 *i* 小于 0 或大于等于载荷中的条目数则将引发*IndexError*。如果载荷是一个字符串 (即*is_multipart()* 为 False) 且给出了 *i*，则会引发*TypeError*。

可选的 *decode* 是一个指明载荷是否应根据 *Content-Transfer-Encoding* 标头被解码的旗标。当其值为 True 且消息没有多个部分时，如果此标头值为 quoted-printable 或 base64 则载荷将被解码。如果使用了其他编码格式，或者找不到 *Content-Transfer-Encoding* 标头时，载荷将被原样返回（不编码）。在所有情况下返回值都是二进制数据。如果消息有多个部分且 *decode* 旗标为 True，则将返回 None。如果载荷为 base64 但内容不完全正确（如缺少填充符、存在 base64 字母表以外的字符等），则将在消息的缺陷属性中添加适当的缺陷值 (分别为 InvalidBase64PaddingDefect 或 InvalidBase64CharactersDefect)。

当 *decode* 为 False (默认值) 时消息体会作为字符串返回而不解码 *Content-Transfer-Encoding*。但是，对于 *Content-Transfer-Encoding* 为 8bit 的情况，会尝试使用 *Content-Type* 标头指定的 charset 来解码原始字节串，并使用 replace 错误处理程序。如果未指定 charset，或者如果指定的 charset 未被 email 包所识别，则会使用默认的 ASCII 字符集来解码消息体。

这是一个过时的方法。在 EmailMessage 类上它的功能已被*get_content()* 和*iter_parts()* 方法所替代。

set_payload(*payload*, *charset=None*)
将整个消息对象的载荷设为 *payload*。客户端要负责确保载荷的不变性。可选的 *charset* 用于设置消息的默认字符集；详情请参阅*set_charset()*。

这是一个过时的方法。在 EmailMessage 类上它的功能已被*set_content()* 方法所替代。

set_charset(*charset*)
将载荷的字符集设为 *charset*，它可以是*Charset* 实例 (参见*email.charset*)、字符集名称字符串或 None。如果是字符串，它将被转换为一个*Charset* 实例。如果 *charset* 是 None，charset 形参将从 *Content-Type* 标头中被删除（消息将不会进行其他修改）。任何其他值都将导致*TypeError*。

如果 *MIME-Version* 标头不存在则将被添加。如果 *Content-Type* 标头不存在，则将添加一个值为 *text/plain* 的该标头。无论 *Content-Type* 标头是否已存在，其 charset 形参都将被设为 *charset.output_charset*。如果 *charset.input_charset* 和 *charset.output_charset* 不同，则载荷将被重编码为 *output_charset*。如果 *Content-Transfer-Encoding* 标头不存在，则载荷将在必要时使用指定的*Charset* 来转换编码，并将添加一个具有相应

值的标头。如果 *Content-Transfer-Encoding* 标头已存在，则会假定载荷已使用该 *Content-Transfer-Encoding* 进行正确编码并不会再被修改。

这是一个过时的方法。在 EmailMessage 类上它的功能已被 email.emailmessage.EmailMessage.set_content() 方法的 *charset* 形参所替代。

get_charset()
返回与消息的载荷相关联的*Charset* 实例。

这是一个过时的方法。在 EmailMessage 类上它将总是返回 None。

以下方法实现了用于访问消息的 **RFC 2822** 标头的类映射接口。请注意这些方法和普通映射（例如字典）接口之间存在一些语义上的不同。举例来说，在一个字典中不能有重复的键，但消息标头则可能有重复。并且，在字典中由*keys()* 返回的键的顺序是没有保证的，但在*Message* 对象中，标头总是会按它们在原始消息中的出现或后继加入顺序返回。任何已删除再重新加入的标头总是会添加到标头列表的末尾。

这些语义上的差异是有意为之且其目的是为了提供最大的便利性。

请注意在任何情况下，消息当中的任何封包标头都不会包含在映射接口当中。

在由字节串生成的模型中，任何包含非 ASCII 字节数据（违反 RFC）的标头值当通过此接口来获取时，将被表示为使用 *unknown-8bit* 字符集的*Header* 对象。

__len__()
返回标头的总数，包括重复项。

__contains__(*name*)
如果消息对象中有一个名为 *name* 的字段则返回 True。匹配操作对大小写不敏感并且 *name* 不应包括末尾的冒号。用于 in 运算符，例如:

```
if 'message-id' in myMessage:
   print('Message-ID:', myMessage['message-id'])
```

__getitem__(*name*)
返回指定名称标头字段的值。*name* 不应包括作为字段分隔符的冒号。如果标头未找到，则返回 None；*KeyError* 永远不会被引发。

请注意如果指定名称的字段在消息标头中多次出现，具体将返回哪个字段值是未定义的。请使用*get_all()* 方法来获取所有指定名称标头的值。

__setitem__(*name*, *val*)
将具有字段名 *name* 和值 *val* 的标头添加到消息中。字段会被添加到消息的现有字段的末尾。

请注意，这个方法 既不会覆盖 也不会删除任何字段名重名的已有字段。如果你确实想保证新字段是整个信息头当中唯一拥有 *name* 字段名的字段，你需要先把旧字段删除。例如:

```
del msg['subject']
msg['subject'] = 'Python roolz!'
```

__delitem__(*name*)
删除信息头当中字段名匹配 *name* 的所有字段。如果匹配指定名称的字段没有找到，也不会抛出任何异常。

keys()
以列表形式返回消息头中所有的字段名。

values()
以列表形式返回消息头中所有的字段值。

items()
以二元元组的列表形式返回消息头中所有的字段名和字段值。

get(*name*, *failobj=None*)
返回指定名称标头字段的值。这与*__getitem__()* 是一样的，不同之处在于如果指定名称标头未找到则会返回可选的 *failobj* (默认为 None)。

以下是一些有用的附加方法:

get_all(*name*, *failobj=None*)
返回字段名为 *name* 的所有字段值的列表。如果信息内不存在匹配的字段，返回 *failobj*（其默认值为 None ）。

add_header(*_name*, *_value*, ***_params*)
高级头字段设定。这个方法与*__setitem__()* 类似，不过你可以使用关键字参数为字段提供附加参数。*_name* 是字段名，*_value* 是字段 主值。

对于关键字参数字典 *_params* 中的每一项，其键会被当作形参名，并执行下划线和连字符间的转换（因为连字符不是合法的 Python 标识符）。通常，形参将以 key="value" 的形式添加，除非值为 None，在这种情况下将只添加键。如果值包含非 ASCII 字符，可将其指定为格式为 (CHARSET, LANGUAGE, VALUE) 的三元组，其中 CHARSET 为要用来编码值的字符集名称字符串，LANGUAGE 通常可设为 None 或空字符串（请参阅 **RFC 2231** 了解其他可能的取值），而 VALUE 为包含非 ASCII 码位的字符串值。如果不是传入一个三元组且值包含非 ASCII 字符，则会自动以 **RFC 2231** 格式使用 CHARSET 为 utf-8 和 LANGUAGE 为 None 对其进行编码。

以下是为示例代码:

```
msg.add_header('Content-Disposition', 'attachment', filename='bud.gif')
```

会添加一个形如下文的头字段:

```
Content-Disposition: attachment; filename="bud.gif"
```

使用非 ASCII 字符的示例代码:

```
msg.add_header('Content-Disposition', 'attachment',
               filename=('iso-8859-1', '', 'Fußballer.ppt'))
```

它的输出结果为

```
Content-Disposition: attachment; filename*="iso-8859-1''Fu%DFballer.ppt"
```

replace_header(*_name*, *_value*)
替换一个标头。将替换在匹配 *_name* 的消息中找到的第一个标头，标头顺序和字段名大小写保持不变。如果未找到匹配的标头，则会引发*KeyError*。

get_content_type()
返回消息的内容类型。返回的字符串会强制转换为 *maintype/subtype* 的全小写形式。如果消息中没有 *Content-Type* 标头则将返回由*get_default_type()* 给出的默认类型。因为根据 **RFC 2045**，消息总是要有一个默认类型，所以*get_content_type()* 将总是返回一个值。

RFC 2045 将消息的默认类型定义为 *text/plain*，除非它是出现在 *multipart/digest* 容器内，在这种情况下其类型应为 *message/rfc822*。如果 *Content-Type* 标头指定了无效的类型，**RFC 2045** 规定其默认类型应为 *text/plain*。

get_content_maintype()
返回信息的主要内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *maintype* 部分。

get_content_subtype()
返回信息的子内容类型。准确来说，此方法返回的是*get_content_type()* 方法所返回的形如 *maintype/subtype* 的字符串当中的 *subtype* 部分。

get_default_type()
返回默认的内容类型。绝大多数的信息，其默认内容类型都是 *text/plain* 。作为 *multipart/digest* 容器内子部分的信息除外，它们的默认内容类型是 *message/rfc822* 。

set_default_type(*ctype*)
设置默认的内容类型。*ctype* 应当为 *text/plain* 或者 *message/rfc822*，尽管这并非强制。默认的内容类型不会存储在 *Content-Type* 标头中。

get_params(*failobj=None*, *header='content-type'*, *unquote=True*)
将消息的 *Content-Type* 形参作为列表返回。所返回列表的元素为以 '=' 号拆分出的键/值对 2 元组。'=' 左侧的为键，右侧的为值。如果形参值中没有 '=' 号，否则该将值如*get_param()* 描述并且在可选 *unquote* 为 True (默认值) 时会被取消转义。

可选的 *failobj* 是在没有 *Content-Type* 标头时要返回的对象。可选的 *header* 是要替代 *Content-Type* 被搜索的标头。

这是一个过时的方法。在 EmailMessage 类上它的功能已被标头访问方法所返回的单独标头对象的 *params* 特征属性所替代。

get_param(*param*, *failobj=None*, *header='content-type'*, *unquote=True*)
将 *Content-Type* 标头的形参 *param* 作为字符串返回。如果消息没有 *Content-Type* 标头或者没有这样的形参，则返回 *failobj* (默认为 None)。

如果给出可选的 *header*，它会指定要替代 *Content-Type* 来使用的消息标头。

形参的键总是以大小写不敏感的方式来比较的。返回值可以是一个字符串，或者如果形参以 **RFC 2231** 编码则是一个 3 元组。当为 3 元组时，值中的元素采用 (CHARSET, LANGUAGE, VALUE) 的形式。请注意 CHARSET 和 LANGUAGE 都可以为 None，在此情况下你应当将 VALUE 当作以 us-ascii 字符集来编码。你可以总是忽略 LANGUAGE。

如果你的应用不关心形参是否以 **RFC 2231** 来编码，你可以通过调用*email.utils.collapse_rfc2231_value()* 来展平形参值，传入来自*get_param()* 的返回值。当值为元组时这将返回一个经适当编码的 Unicode 字符串，否则返回未经转换的原字符串。例如:

```
rawparam = msg.get_param('foo')
param = email.utils.collapse_rfc2231_value(rawparam)
```

无论在哪种情况下，形参值（或为返回的字符串，或为 3 元组形式的 VALUE 条目）总是未经转换的，除非 *unquote* 被设为 False。

这是一个过时的方法。在 EmailMessage 类上它的功能已被标头访问方法所返回的单独标头对象的 *params* 特征属性所替代。

set_param(*param*, *value*, *header='Content-Type'*, *requote=True*, *charset=None*, *language=''*, *replace=False*)
在 *Content-Type* 标头中设置一个形参。如果该形参已存在于标头中，它的值将被替换为 *value*。如果此消息还未定义 *Content-Type* 标头，它将被设为 *text/plain* 且新的形参值将按 **RFC 2045** 的要求添加。

可选的 *header* 指定一个 *Content-Type* 的替代标头，并且所有形参将根据需要被转换，除非可选的 *requote* 为 False (默认为 True)。

如果指定了可选的 *charset*，形参将按照 **RFC 2231** 来编码。可选的 *language* 指定了 RFC 2231 的语言，默认为空字符串。*charset* 和 *language* 都应为字符串。

如果 *replace* 为 False （默认值），该头字段会被移动到所有头字段的末尾。如果 *replace* 为 True ，字段会被原地更新。

3.4 版更變: 添加了 replace 关键字。

del_param(*param*, *header='content-type'*, *requote=True*)
从 *Content-Type* 标头中完全移除给定的形参。标头将被原地重写并不带该形参或它的值。所有

的值将根据需要被转换，除非 *requote* 为 False（默认为 True）。可选的 *header* 指定 *Content-Type* 的一个替代项。

set_type(*type*, *header='Content-Type'*, *requote=True*)
设置 *Content-Type* 标头的主类型和子类型。*type* 必须为 *maintype/subtype* 形式的字符串，否则会引发*ValueError*。

此方法可替换 *Content-Type* 标头，并保持所有形参不变。如果 *requote* 为 False，这会保持原有标头引用转换不变，否则形参将被引用转换（默认行为）。

可以在 *header* 参数中指定一个替代标头。当 *Content-Type* 标头被设置时也会添加一个 *MIME-Version* 标头。

这是一个过时的方法。在 EmailMessage 类上它的功能已被 make_ 和 add_ 方法所替代。

get_filename(*failobj=None*)
返回信息头当中 *Content-Disposition* 字段当中名为 filename 的参数值。如果该字段当中没有此参数，该方法会退而寻找 *Content-Type* 字段当中的 name 参数值。如果这个也没有找到，或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照*email.utils.unquote()* 方法去除引号。

get_boundary(*failobj=None*)
返回信息头当中 *Content-Type* 字段当中名为 boundary 的参数值。如果字段当中没有此参数，或者这些个字段压根就不存在，返回 *failobj* 。返回的字符串永远按照*email.utils.unquote()* 方法去除引号。

set_boundary(*boundary*)
将 *Content-Type* 头字段的 boundary 参数设置为 *boundary* 。*set_boundary()* 方法永远都会在必要的时候为 *boundary* 添加引号。如果信息对象中没有 *Content-Type* 头字段，抛出*HeaderParseError* 异常。

请注意使用这个方法与删除旧的 *Content-Type* 标头并通过*add_header()* 添加一个带有新边界的新标头有细微的差异，因为*set_boundary()* 会保留 *Content-Type* 标头在原标头列表中的顺序。但是，它 不会保留原 *Content-Type* 标头中可能存在的任何连续的行。

get_content_charset(*failobj=None*)
返回 *Content-Type* 头字段中的 charset 参数，强制小写。如果字段当中没有此参数，或者这个字段压根不存在，返回 *failobj* 。

请注意此方法不同于*get_charset()*，后者会返回*Charset* 实例作为消息体的默认编码格式。

get_charsets(*failobj=None*)
返回一个包含了信息内所有字符集名字的列表。如果信息是 *multipart* 类型的，那么列表当中的每一项都对应其负载的子部分的字符集名字。否则，该列表是一个长度为 1 的列表。

列表中的每一项都是字符串，它们是其所表示的子部分的 *Content-Type* 标头中 charset 形参的值。但是，如果该子部分没有 *Content-Type* 标头，或没有 charset 形参，或者主 MIME 类型不是 *text*，则所返回列表中的对应项将为 *failobj*。

get_content_disposition()
如果信息的 *Content-Disposition* 头字段存在，返回其字段值；否则返回 None 。返回的值均为小写，不包含参数。如果信息遵循 **RFC 2183** 标准，则返回值只可能在 *inline* 、*attachment* 和 None 之间选择。

3.5 版新加入.

walk()
walk() 方法是一个多功能生成器。它可以被用来以深度优先顺序遍历信息对象树的所有部分和子部分。一般而言，*walk()* 会被用作 for 循环的迭代器，每一次迭代都返回其下一个子部分。

以下例子会打印出一封具有多部分结构之信息的每个部分的 MIME 类型。

```
>>> for part in msg.walk():
...     print(part.get_content_type())
multipart/report
text/plain
message/delivery-status
text/plain
text/plain
message/rfc822
text/plain
```

walk 会遍历所有*is_multipart()* 方法返回 True 的部分之子部分，哪怕 msg.get_content_maintype() == 'multipart' 返回的是 False 。使用 _structure 除错帮助函数可以帮助我们在下面这个例子当中看清楚这一点：

```
>>> for part in msg.walk():
...     print(part.get_content_maintype() == 'multipart',
...           part.is_multipart())
True True
False False
False True
False False
False False
False True
False False
>>> _structure(msg)
multipart/report
    text/plain
    message/delivery-status
        text/plain
        text/plain
    message/rfc822
        text/plain
```

在这里，message 的部分并非 multiparts ，但是它们真的包含子部分！is_multipart() 返回 True ， walk 也深入进这些子部分中。

Message 对象也可以包含两个可选的实例属性，它们可被用于生成纯文本的 MIME 消息。

preamble
MIME 文档格式在标头之后的空白行以及第一个多部分的分界字符串之间允许添加一些文本，通常，此文本在支持 MIME 的邮件阅读器中永远不可见，因为它处在标准 MIME 防护范围之外。但是，当查看消息的原始文本，或当在不支持 MIME 的阅读器中查看消息时，此文本会变得可见。

preamble 属性包含 MIME 文档开头部分的这些处于保护范围之外的文本。当*Parser* 在标头之后及第一个分界字符串之前发现一些文本时，它会将这些文本赋值给消息的 *preamble* 属性。当*Generator* 写出 MIME 消息的纯文本表示形式时，如果它发现消息具有 *preamble* 属性，它将在标头及第一个分界之间区域写出这些文本。请参阅*email.parser* 和*email.generator* 了解更多细节。

请注意如果消息对象没有前导文本，则 *preamble* 属性将为 None。

epilogue
epilogue 属性的作用方式与 *preamble* 属性相同，区别在于它包含出现于最后一个分界与消息结尾之间的文本。

你不需要将 epilogue 设为空字符串以便让*Generator* 在文件末尾打印一个换行符。

defects
defects 属性包含在解析消息时发现的所有问题的列表。请参阅*email.errors* 了解可能的解析缺

陷的详细描述。

19.1.10 email.mime: 从头创建电子邮件和 MIME 对象

源代码: Lib/email/mime/

此模块是旧版 (Compat32) 电子邮件 API 的组成部分。它的功能在新版 API 中被*contentmanager* 部分替代，但在某些应用中这些类仍可能有用，即使是在非旧版代码中。

通常，你是通过传递一个文件或一些文本到解析器来获得消息对象结构体的，解析器会解析文本并返回根消息对象。不过你也可以从头开始构建一个完整的消息结构体，甚至是手动构建单独的*Message* 对象。实际上，你也可以接受一个现有的结构体并添加新的*Message* 对象并移动它们。这为切片和分割 MIME 消息提供了非常方便的接口。

你可以通过创建*Message* 实例并手动添加附件和所有适当的标头来创建一个新的对象结构体。不过对于 MIME 消息来说，*email* 包提供了一些便捷子类来让事情变得更容易。

这些类列示如下:

class email.mime.base.**MIMEBase**(*_maintype*, *_subtype*, *, *policy=compat32*, ***_params*)
模块: email.mime.base

这是*Message* 的所有 MIME 专属子类。通常你不会创建专门的*MIMEBase* 实例，尽管你可以这样做。*MIMEBase* 主要被提供用来作为更具体的 MIME 感知子类的便捷基类。

_maintype 是 *Content-Type* 的主类型 (例如 *text* 或 *image*)，而 *_subtype* 是 *Content-Type* 的次类型 (例如 *plain* 或 *gif*)。*_params* 是一个形参键/值字典并会被直接传递给*Message.add_header*。

如果指定了 *policy* (默认为*compat32* 策略)，它将被传递给*Message*。

MIMEBase 类总是会添加一个 *Content-Type* 标头 (基于 *_maintype*, *_subtype* 和 *_params*)，以及一个 *MIME-Version* 标头 (总是设为 1.0)。

3.6 版更變: 添加了 *policy* 仅限关键字形参。

class email.mime.nonmultipart.**MIMENonMultipart**
模块: email.mime.nonmultipart

MIMEBase 的子类，这是用于非 *multipart* MIME 消息的中间基类。这个类的主要目标是避免使用*attach()* 方法，该方法仅对 *multipart* 消息有意义。如果*attach()* 被调用，则会引发*MultipartConversionError* 异常。

class email.mime.multipart.**MIMEMultipart**(*_subtype='mixed'*, *boundary=None*, *_subparts=None*, *, *policy=compat32*, ***_params*)
模块: email.mime.multipart

MIMEBase 的子类，这是用于 *multipart* MIME 消息的中间基类。可选的 *_subtype* 默认为 *mixed*，但可被用来指定消息的子类型。将会在消息对象中添加一个:mimetype:*multipart/_subtype* 的 *Content-Type* 标头。并还将添加一个 *MIME-Version* 标头。

可选的 *boundary* 是多部分边界字符串。当为 None (默认值) 时，则会在必要时（例如当消息被序列化时）计算边界。

_subparts 是载荷初始子部分的序列。此序列必须可以被转换为列表。你总是可以使用*Message.attach* 方法将新的子部分附加到消息中。

可选的 *policy* 参数默认为*compat32*。

用于 *Content-Type* 标头的附加形参会从关键字参数中获取，或者传入到 *_params* 参数，该参数是一个关键字的字典。

3.6 版更變: 添加了 *policy* 仅限关键字形参。

class email.mime.application.**MIMEApplication**(*_data*, *_subtype='octet-stream'*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

模块: email.mime.application

MIMENonMultipart 的子类，*MIMEApplication* 类被用来表示主类型为 *application* 的 MIME 消息。*_data* 是包含原始字节数据的字符串。可选的 *_subtype* 指定 MIME 子类型并默认为 *octet-stream*。

可选的 *_encoder* 是一个可调用对象（即函数），它将执行实际的数据编码以便传输。这个可调用对象接受一个参数，该参数是*MIMEApplication* 的实例。它应当使用*get_payload()* 和*set_payload()* 来将载荷改为已编码形式。它还应根据需要将任何 *Content-Transfer-Encoding* 或其他标头添加到消息对象中。默认编码格式为 base64。请参阅*email.encoders* 模块来查看内置编码器列表。

可选的 *policy* 参数默认为*compat32*。

_params 会被直接传递给基类的构造器。

3.6 版更變: 添加了 *policy* 仅限关键字形参。

class email.mime.audio.**MIMEAudio**(*_audiodata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

模块: email.mime.audio

MIMENonMultipart 的子类，*MIMEAudio* 类被用来创建主类型为 *audio* 的 MIME 消息。*_audiodata* 是包含原始音频数据的字符串。如果此数据可由标准 Python 模块*sndhdr* 来解码，则其子类型将被自动包括在 *Content-Type* 标头中。在其他情况下你可以通过 *_subtype* 参数显式地指定音频子类型。如果无法猜测出主类型并且未给出 *_subtype*，则会引发*TypeError*。

可选的 *_encoder* 是一个可调用对象（即函数），它将执行实际的音频数据编码以便传输。这个可调用对象接受一个参数，该参数是*MIMEAudio* 的实例。它应当使用*get_payload()* 和*set_payload()* 来将载荷改为已编码形式。它还应根据需要将任何 *Content-Transfer-Encoding* 或其他标头添加到消息对象中。默认编码格式为 base64。请参阅*email.encoders* 模块来查看内置编码器列表。

可选的 *policy* 参数默认为*compat32*。

_params 会被直接传递给基类的构造器。

3.6 版更變: 添加了 *policy* 仅限关键字形参。

class email.mime.image.**MIMEImage**(*_imagedata*, *_subtype=None*, *_encoder=email.encoders.encode_base64*, ***, *policy=compat32*, ***_params*)

模块: email.mime.image

MIMENonMultipart 的子类，*MIMEImage* 类被用来创建主类型为 *image* 的 MIME 消息对象。*_imagedata* 是包含原始图像数据的字符串。如果此数据可由标准 Python 模块*imghdr* 来解码，则其子类型将被自动包括在 *Content-Type* 标头中。在其他情况下你可以通过 *_subtype* 参数显式地指定图像子类型。如果无法猜测出主类型并且未给出 *_subtype*，则会引发*TypeError*。

可选的 *_encoder* 是一个可调用对象（即函数），它将执行实际的图像数据编码以便传输。这个可调用对象接受一个参数，该参数是*MIMEImage* 的实例。它应当使用*get_payload()* 和*set_payload()* 来将载荷改为已编码形式。它还应根据需要将任何 *Content-Transfer-Encoding* 或其他标头添加到消息对象中。默认编码格式为 base64。请参阅*email.encoders* 模块来查看内置编码器列表。

可选的 *policy* 参数默认为*compat32*。

_params 会被直接传递给*MIMEBase* 构造器。

3.6 版更變: 添加了 *policy* 仅限关键字形参。

class email.mime.message.**MIMEMessage**(*_msg*, *_subtype='rfc822'*, ***, *policy=compat32*)
模块: email.mime.message

MIMENonMultipart 的子类，*MIMEMessage* 类被用来创建主类型为 *message* 的 MIME 对象。*_msg* 将被用作载荷，并且必须为*Message* 类（或其子类）的实例，否则会引发*TypeError*。

可选的 *_subtype* 设置消息的子类型；它的默认值为 *rfc822*。

可选的 *policy* 参数默认为*compat32*。

3.6 版更變: 添加了 *policy* 仅限关键字形参。

class email.mime.text.**MIMEText**(*_text*, *_subtype='plain'*, *_charset=None*, ***, *policy=compat32*)
模块: email.mime.text

MIMENonMultipart 的子类，*MIMEText* 类被用来创建主类型为 *text* 的 MIME 对象。*_text* 是用作载荷的字符串。*_subtype* 指定子类型并且默认为 *plain*。*_charset* 是文本的字符集并会作为参数传递给*MIMENonMultipart* 构造器；如果该字符串仅包含 ascii 码位则其默认值为 us-ascii，否则为 utf-8。*_charset* 形参接受一个字符串或是一个*Charset* 实例。

除非 *_charset* 参数被显式地设为 None，否则所创建的 MIMEText 对象将同时具有附带 charset 形参的 *Content-Type* 标头，以及 *Content-Transfer-Encoding* 标头。这意味着后续的 set_payload 调用将不再产生已编码的载荷，即使它在 set_payload 命令中被传入。你可以通过删除 Content-Transfer-Encoding 标头来“重置”此行为，在此之后的 set_payload 调用将自动编码新的载荷（并添加新的 *Content-Transfer-Encoding* 标头）。

可选的 *policy* 参数默认为*compat32*。

3.5 版更變: *_charset* 也可接受*Charset* 实例。

3.6 版更變: 添加了 *policy* 仅限关键字形参。

19.1.11 email.header: 国际化标头

源代码: Lib/email/header.py

此模块是旧式 (Compat32) email API 的一部分。在当前的 API 中标头的编码和解码是由*EmailMessage* 类的字典型 API 来透明地处理的。除了在旧有代码中使用，此模块在需要完全控制当编码标头时所使用的字符集时也很有用处。

本节中的其余文本是此模块的原始文档。

RFC 2822 是描述电子邮件消息格式的基础标准。它派生自更早的 **RFC 822** 标准，该标准在大多数电子邮件仅由 ASCII 字符组成时已被广泛使用。**RFC 2822** 所描述的规范假定电子邮件都只包含 7 位 ASCII 字符。

当然，随着电子邮件在全球部署，它已经变得国际化了，例如电子邮件消息中现在可以使用特定语言的专属字符集。这个基础标准仍然要求电子邮件消息只使用 7 位 ASCII 字符来进行传输，为此编写了大量 RFC 来描述如何将包含非 ASCII 字符的电子邮件编码为符合 **RFC 2822** 的格式。这些 RFC 包括 **RFC 2045**, **RFC 2046**, **RFC 2047** 和 **RFC 2231**。*email* 包在其*email.header* 和*email.charset* 模块中支持了这些标准。

如果你想在你的电子邮件标头中包括非 ASCII 字符，比如说是在 *Subject* 或 *To* 字段中，你应当使用*Header* 类并将*Message* 对象中的字段赋值为*Header* 的实例而不是使用字符串作为字段值。请从*email.header* 模块导入*Header* 类。例如:

```
>>> from email.message import Message
>>> from email.header import Header
>>> msg = Message()
>>> h = Header('p\xf6stal', 'iso-8859-1')
```

(下页继续)

(繼續上一頁)

```
>>> msg['Subject'] = h
>>> msg.as_string()
'Subject: =?iso-8859-1?q?p=F6stal?=\n\n'
```

是否注意到这里我们是如何希望 *Subject* 字段包含非 ASCII 字符的？我们通过创建一个*Header* 实例并传入字节串编码所用的字符集来做到这一点。当后续的*Message* 实例被展平时，*Subject* 字段会正确地按 **RFC 2047** 来编码。可感知 MIME 的电子邮件阅读器将会使用嵌入的 ISO-8859-1 字符来显示此标头。

以下是*Header* 类描述:

class email.header.**Header**(*s=None*, *charset=None*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*, *errors='strict'*)

创建符合 MIME 要求的标头，其中可包含不同字符集的字符串。

可选的 *s* 是初始标头值。如果为 None (默认值)，则表示初始标头值未设置。你可以在稍后使用*append()* 方法调用向标头添加新值。*s* 可以是*bytes* 或*str* 的实例，注意参阅*append()* 文档了解相关语义。

可选的 *charset* 用于两种目的：它的含义与*append()* 方法的 *charset* 参数相同。它还会为所有省略了 *charset* 参数的后续*append()* 调用设置默认字符集。如果 *charset* 在构造器中未提供（默认设置），则会将 us-ascii 字符集用作 *s* 的初始字符集以及后续*append()* 调用的默认字符集。

通过 *maxlinelen* 可以显式指定最大行长度。要将第一行拆分为更短的值 (以适应未被包括在 to account for the field header which isn't included in *s* 中的字段标头，例如 *Subject*)，则将字段名称作为 *header_name* 传入。*maxlinelen* 默认值为 76，而 *header_name* 默认值为 None，表示不考虑拆分超长标头的第一行。

可选的 *continuation_ws* 必须为符合 **RFC 2822** 的折叠用空白符，通常是空格符或硬制表符。这个字符将被加缀至连续行的开头。*continuation_ws* 默认为一个空格符。

可选的 *errors* 会被直接传递给*append()* 方法。

append(*s*, *charset=None*, *errors='strict'*)

将字符串 *s* 添加到 MIME 标头。

如果给出可选的 *charset*，它应当是一个*Charset* 实例 (参见*email.charset*) 或字符集名称，该参数将被转换为一个*Charset* 实例。如果为 None (默认值) 则表示会使用构造器中给出的 *charset*。

s 可以是*bytes* 或*str* 的实例。如果它是*bytes* 的实例，则 *charset* 为该字节串的编码格式，如果字节串无法用该字符集来解码则将引发*UnicodeError*。

如果 *s* 是*str* 的实例，则 *charset* 是用来指定字符串中字符字符集的提示。

在这两种情况下，当使用 **RFC 2047** 规则产生符合 **RFC 2822** 的标头时，将使用指定字符集的输出编解码器来编码字符串。如果字符串无法使用该输出编解码器来编码，则将引发 UnicodeError。

可选的 *errors* 会在 *s* 为字节串时被作为 errors 参数传递给 decode 调用。

encode(*splitchars=';, \t'*, *maxlinelen=None*, *linesep='\n'*)

将消息标头编码为符合 RFC 的格式，可能会对过长的行采取折行并将非 ASCII 部分以 base64 或 quoted-printable 编码格式进行封装。

可选的 *splitchars* 是一个字符串，其中包含应在正常的标头折行处理期间由拆分算法赋予额外权重的字符。这是对于 **RFC 2822** 中' 更高层级语法拆分' 的很粗略的支持：在拆分期间会首选在 splitchar 之前的拆分点，字符的优先级是基于它们在字符串中的出现顺序。字符串中可包含空格和制表符以指明当其他拆分字符未在被拆分行中出现时是否要将某个字符作为优先于另一个字符的首选拆分点。拆分字符不会影响以 **RFC 2047** 编码的行。

如果给出 *maxlinelen*，它将覆盖实例的最大行长度值。

linesep 指定用来分隔已折叠标头行的字符。它默认为 Python 应用程序代码中最常用的值 (\n)，但也可以指定为 \r\n 以便产生带有符合 RFC 的行分隔符的标头。

3.2 版更變: 增加了 *linesep* 参数。

Header 类还提供了一些方法以支持标准运算符和内置函数。

__str__()
以字符串形式返回*Header* 的近似表示，使用不受限制的行长度。所有部分都会使用指定编码格式转换为 unicode 并适当地连接起来。任何带有 'unknown-8bit' 字符集的部分都会使用 'replace' 错误处理程序解码为 ASCII。

3.2 版更變: 增加对 'unknown-8bit' 字符集的处理。

__eq__(*other*)
这个方法允许你对两个*Header* 实例进行相等比较。

__ne__(*other*)
这个方法允许你对两个*Header* 实例进行不等比较。

email.header 模块还提供了下列便捷函数。

email.header.**decode_header**(*header*)
在不转换字符集的情况下对消息标头值进行解码。*header* 为标头值。

这个函数返回一个 (decoded_string, charset) 对的列表，其中包含标头的每个已解码部分。对于标头的未编码部分 *charset* 为 None，在其他情况下则为一个包含已编码字符串中所指定字符集名称的小写字符串。

以下是为示例代码:

```
>>> from email.header import decode_header
>>> decode_header('=?iso-8859-1?q?p=F6stal?=')
[(b'p\xf6stal', 'iso-8859-1')]
```

email.header.**make_header**(*decoded_seq*, *maxlinelen=None*, *header_name=None*, *continuation_ws=' '*)
基于*decode_header()* 所返回的数据对序列创建一个*Header* 实例。

decode_header() 接受一个标头值字符串并返回格式为 (decoded_string, charset) 的数据对序列，其中 *charset* 是字符集名称。

这个函数接受这样的数据对序列并返回一个*Header* 实例。可选的 *maxlinelen*, *header_name* 和 *continuation_ws* 与*Header* 构造器中的含义相同。

19.1.12 email.charset: 表示字符集

源代码: Lib/email/charset.py

此模块是旧版 (Compat32) email API 的组成部分。在新版 API 中只会使用其中的别名表。

本段落中的剩余文本是该模块的原始文档。

此模块提供了一个*Charset* 类用来表示电子邮件消息中的字符集和字符集转换操作，以及一个字符集注册表和几个用于操作此注册表的便捷方法。*Charset* 的实例在*email* 包的其他几个模块中也有使用。

请从*email.charset* 模块导入这个类。

class email.charset.**Charset**(*input_charset=DEFAULT_CHARSET*)
将字符集映射到其 email 特征属性。

这个类提供了特定字符集对于电子邮件的要求的相关信息。考虑到适用编解码器的可用性，它还为字符集之间的转换提供了一些便捷例程。在给定字符集的情况下，它将尽可能地以符合 RFC 的方式在电子邮件消息中提供有关如何使用该字符集的信息。

特定字符集当在电子邮件标头或消息体中使用时必须以 quoted-printable 或 base64 来编码。某些字符集则必须被立即转换，不允许在电子邮件中使用。

可选的 *input_charset* 说明如下；它总是会被强制转为小写。在进行别名正规化后它还会被用来查询字符集注册表以找出用于该字符集的标头编码格式、消息体编码格式和输出转换编解码器。举例来说，如果 *input_charset* 为 `iso-8859-1`，则标头和消息体将会使用 quoted-printable 来编码并且不需要输出转换编解码器。如果 *input_charset* 为 `euc-jp`，则标头将使用 base64 来编码，消息体将不会被编码，但输出文本将从 `euc-jp` 字符集转换为 `iso-2022-jp` 字符集。

Charset 实例具有下列数据属性:

input_charset
指定的初始字符集。通用别名会被转换为它们的 官方电子邮件名称 (例如 `latin_1` 会被转换为 `iso-8859-1`)。默认值为 7 位 `us-ascii`。

header_encoding
If the character set must be encoded before it can be used in an email header, this attribute will be set to `charset.QP` (for quoted-printable), `charset.BASE64` (for base64 encoding), or `charset.SHORTEST` for the shortest of QP or BASE64 encoding. Otherwise, it will be `None`.

body_encoding
Same as *header_encoding*, but describes the encoding for the mail message's body, which indeed may be different than the header encoding. `charset.SHORTEST` is not allowed for *body_encoding*.

output_charset
某些字符集在用于电子邮件标头或消息体之前必须被转换。如果 *input_charset* 是这些字符集之一，该属性将包含输出将要转换的字符集名称。在其他情况下，该属性将为 `None`。

input_codec
用于将 *input_charset* 转换为 Unicode 的 Python 编解码器名称。如果不需要任何转换编解码器，该属性将为 `None`。

output_codec
用于将 Unicode 转换为 *output_charset* 的 Python 编解码器名称。如果不需要任何转换编解码器，该属性将具有与 *input_codec* 相同的值。

Charset 实例还有下列方法:

get_body_encoding()
返回用于消息体编码的内容转换编码格式。

根据所使用的编码格式返回 `quoted-printable` 或 `base64`，或是返回一个函数，在这种情况下你应当调用该函数并附带一个参数，即被编码的消息对象。该函数应当自行将 *Content-Transfer-Encoding* 标头设为适当的值。

如果 *body_encoding* 为 `QP` 则返回字符串 `quoted-printable`，如果 *body_encoding* 为 `BASE64` 则返回字符串 `base64`，并在其他情况下返回字符串 `7bit`。

get_output_charset()
返回输出字符集。

如果 *output_charset* 属性不为 `None` 则返回该属性，否则返回 *input_charset*。

header_encode(*string*)
对字符串 *string* 执行标头编码。

编码格式的类型 (base64 或 quoted-printable) 将取决于 *header_encoding* 属性。

header_encode_lines(*string*, *maxlengths*)
通过先将 *string* 转换为字节串来对其执行标头编码。

这类似于 *header_encode()*，区别是字符串会被调整至参数 *maxlengths* 所给出的最大行长度，它应当是一个迭代器：该迭代器返回的每个元素将提供下一个最大行长度。

body_encode(*string*)
对字符串 *string* 执行消息体编码。

编码格式的类型 (base64 或 quoted-printable) 将取决于 *body_encoding* 属性。

Charset 类还提供了一些方法以支持标准运算和内置函数。

__str__()
将 *input_charset* 以强制转为小写的字符串形式返回。__repr__() 是*__str__()* 的别名。

__eq__(*other*)
这个方法允许你对两个*Charset* 实例进行相等比较。

__ne__(*other*)
这个方法允许你对两个*Charset* 实例进行相等比较。

email.charset 模块还提供了下列函数用于向全局字符集、别名以及编解码器注册表添加新条目:

email.charset.**add_charset**(*charset*, *header_enc=None*, *body_enc=None*, *output_charset=None*)
向全局注册表添加字符特征属性。

charset 是输入字符集，它必须为某个字符集的正规名称。

Optional *header_enc* and *body_enc* is either charset.QP for quoted-printable, charset.BASE64 for base64 encoding, charset.SHORTEST for the shortest of quoted-printable or base64 encoding, or None for no encoding. SHORTEST is only valid for *header_enc*. The default is None for no encoding.

可选的 *output_charset* 是输出所应当采用的字符集。当 Charset.convert() 方法被调用时将会执行从输入字符集到输出字符集的转换。默认情况下输出字符集将与输入字符集相同。

input_charset 和 *output_charset* 都必须在模块的字符集-编解码器映射中具有 Unicode 编解码器条目；使用*add_codec()* 可添加本模块还不知道的编解码器。请参阅*codecs* 模块的文档来了解更多信息。

全局字符集注册表保存在模块全局字典 CHARSETS 中。

email.charset.**add_alias**(*alias*, *canonical*)
添加一个字符集别名。*alias* 为特定的别名，例如 latin-1。*canonical* 是字符集的正规名称，例如 iso-8859-1。

全局字符集注册表保存在模块全局字典 ALIASES 中。

email.charset.**add_codec**(*charset*, *codecname*)
添加在给定字符集的字符和 Unicode 之间建立映射的编解码器。

charset 是某个字符集的正规名称。*codecname* 是某个 Python 编解码器的名称，可以被用来作为*str* 的*encode()* 方法的第二个参数。

19.1.13 email.encoders: 编码器

源代码: Lib/email/encoders.py

此模块是旧版 (Compat32) email API 的组成部分。在新版 API 中将由*set_content()* 方法的 *cte* 形参提供该功能。

此模块在 Python 3 中已弃用。这里提供的函数不应被显式地调用，因为*MIMEText* 类会在类实例化期间使用 *_subtype* 和 *_charset* 值来设置内容类型和 CTE 标头。

本段落中的剩余文本是该模块的原始文档。

当创建全新的*Message* 对象时，你经常需要对载荷编码以便通过兼容的邮件服务器进行传输。对于包含二进制数据的 *image/** 和 *text/** 类型的消息来说尤其如此。

email 包在其 encoders 模块中提供了一些方便的编码器。这些编码器实际上由*MIMEAudio* 和*MIMEImage* 类构造器所使用以提供默认编码格式。所有编码器函数都只接受一个参数，即要编码的消息对象。它们通常会提取有效载荷，对其进行编码，并将载荷重置为这种新编码的值。它们还应当相应地设置 *Content-Transfer-Encoding* 标头。

请注意，这些函数对于多段消息没有意义。它们必须应用到各个单独的段上面，而不是整体。如果直接传递一个多段类型的消息，会产生一个*TypeError* 错误。

下面是提供的编码函数：

email.encoders.**encode_quopri**(*msg*)
将有效数据编码为经转换的可打印形式，并将 *Content-Transfer-Encoding* 标头设置为 quoted-printable[1]。当大多数实际的数据是普通的可打印数据但包含少量不可打印的字符时，这是一个很好的编码。

email.encoders.**encode_base64**(*msg*)
将有效载荷编码为 base64 形式，并将 *Content-Transfer-Encoding* 标头设为 base64。当你的载荷主要包含不可打印数据时这是一种很好用的编码格式，因为它比 quoted-printable 更紧凑。base64 编码格式的缺点是它会使文本变成人类不可读的形式。

email.encoders.**encode_7or8bit**(*msg*)
此函数并不实际改变消息的有效载荷，但它会基于载荷数据将 *Content-Transfer-Encoding* 标头相应地设为 7bit 或 8bit。

email.encoders.**encode_noop**(*msg*)
此函数什么都不会做；它甚至不会设置 *Content-Transfer-Encoding* 标头。

𝔽解

19.1.14 email.utils: 其他工具

源代码: Lib/email/utils.py

---

email.utils 模块提供如下几个工具

email.utils.**localtime**(*dt=None*)
以感知型 datetime 对象返回当地时间。如果调用时参数为空，则返回当前时间。否则 *dt* 参数应该是一个*datetime* 实例，并根据系统时区数据库转换为当地时区。如果 *dt* 是简单型的 (即 dt.tzinfo 是 None)，则假定为当地时间。在这种情况下，为正值或零的 *isdst* 会使 localtime 假定夏季时间（例如，夏令时）对指定时间生效或不生效。负值 *isdst* 会使 localtime 预测夏季时间对指定时间是否生效。

3.3 版新加入.

email.utils.**make_msgid**(*idstring=None*, *domain=None*)
返回一个适合作为兼容 **RFC 2822** 的 *Message-ID* 标头的字符串。可选参数 *idstring* 可传入一字符串以增强该消息 ID 的唯一性。可选参数 *domain* 可用于提供消息 ID 中字符'@' 之后的部分，其默认值是本机的主机名。正常情况下无需覆盖此默认值，但在特定情况下覆盖默认值可能会有用，比如构建一个分布式系统，在多台主机上采用一致的域名。

3.2 版更變: 增加了关键字 *domain*

下列函数是旧（Compat32）电子邮件 API 的一部分。新 API 提供的解析和格式化在标头解析机制中已经被自动完成，故在使用新 API 时没有必要直接使用它们函数。

[1] 请注意使用*encode_quopri()* 编码格式还会对数据中的所有制表符和空格符进行编码。

email.utils.**quote**(*str*)
返回一个新的字符串，*str* 中的反斜杠被替换为两个反斜杠，并且双引号被替换为反斜杠加双引号。

email.utils.**unquote**(*str*)
返回 *str* 被去除引用后的字符串。如果 *str* 开头和结尾均是双引号，则这对双引号被去除。类似地，如果 *str* 开头和结尾都是尖角括号，这对尖角括号会被去除。

email.utils.**parseaddr**(*address*)
将地址（应为诸如 *To* 或者 *Cc* 之类包含地址的字段值）解析为构成之的 真实名字和 电子邮件地址部分。返回包含这两个信息的一个元组；如若解析失败，则返回一个二元组 `('', '')`。

email.utils.**formataddr**(*pair*, *charset='utf-8'*)
是*parseaddr()* 的逆操作，接受一个（真实名字，电子邮件地址）的二元组，并返回适合于 *To* or *Cc* 标头的字符串。如果第一个元素为假性值，则第二个元素将被原样返回。

可选地，如果指定 *charset*，则被视为一符合 **RFC 2047** 的编码字符集，用于编码 真实名字中的非 ASCII 字符。可以是一个*str* 类的实例，或者一个*Charset* 类。默认为 `utf-8` 。

3.3 版更變: 添加了 *charset* 选项。

email.utils.**getaddresses**(*fieldvalues*)
该方法返回一个形似 `parseaddr()` 返回的二元组的列表。*fieldvalues* 是一个序列，包含了形似*Message.get_all* 返回值的标头字段值。获取了一消息的所有收件人的简单示例如下：

```
from email.utils import getaddresses

tos = msg.get_all('to', [])
ccs = msg.get_all('cc', [])
resent_tos = msg.get_all('resent-to', [])
resent_ccs = msg.get_all('resent-cc', [])
all_recipients = getaddresses(tos + ccs + resent_tos + resent_ccs)
```

email.utils.**parsedate**(*date*)
尝试根据 **RFC 2822** 的规则解析一个日期。然而，有些寄信人不严格遵守这一格式，所以这种情况下*parsedate()* 会尝试猜测其形式。*date* 是一个字符串包含了一个形如 `"Mon, 20 Nov 1995 19:12:08 -0500"` 的 **RFC 2822** 格式日期。如果日期解析成功，*parsedate()* 将返回一个九元组，可直接传递给*time.mktime()*；否则返回 `None`。注意返回的元组中下标为 6、7、8 的部分是无用的。

email.utils.**parsedate_tz**(*date*)
执行与*parsedate()* 相同的功能，但会返回 `None` 或是一个 10 元组；前 9 个元素构成一个可以直接传给*time.mktime()* 的元组，而第十个元素则是该日期的时区与 UTC (格林威治平均时 GMT 的正式名称)[1] 的时差。如果输入字符串不带时区，则所返回元组的最后一个元素将为 `0`，这表示 UTC。请注意结果元组的索引号 6, 7 和 8 是不可用的。

email.utils.**parsedate_to_datetime**(*date*)
format_datetime() 的逆操作。执行与*parsedate()* 相同的功能，但会在成功时返回一个*datetime*。如果输入日期的时区值为 `-0000`，则 `datetime` 将为一个简单型 `datetime`，而如果日期符合 RFC 标准则它将代表一个 UTC 时间，但是并不指明日期所在消息的实际源时区。如果输入日期具有任何其他有效的时区差值，则 `datetime` 将为一个感知型 `datetime` 并与*timezone tzinfo* 相对应。

3.3 版新加入.

email.utils.**mktime_tz**(*tuple*)
将*parsedate_tz()* 所返回的 10 元组转换为一个 UTC 时间戳（相距 Epoch 纪元初始的秒数）。如果元组中的时区项为 `None`，则视为当地时间。

email.utils.**formatdate**(*timeval=None*, *localtime=False*, *usegmt=False*)
返回 **RFC 2822** 标准的日期字符串，例如:

[1] 请注意时区时差的符号与同一时区的 `time.timezone` 变量的符号相反；后者遵循 POSIX 标准而此模块遵循 **RFC 2822**。

```
Fri, 09 Nov 2001 01:08:47 -0000
```

可选的 *timeval* 如果给出，则是一个可被*time.gmtime()* 和*time.localtime()* 接受的浮点数时间值，否则会使得当前时间。

可选的 *localtime* 是一个旗标，当为 `True` 时，将会解析 *timeval*，并返回一个相对于当地时区而非 UTC 的日期值，并会适当地考虑夏令时。默认值 `False` 表示使用 UTC。

可选的 *usegmt* 是一个旗标，当为 `True` 时，将会输出一个日期字符串，其中时区表示为 ascii 字符串 `GMT` 而非数字形式的 `-0000`。这对某些协议（例如 HTTP）来说是必要的。这仅在 *localtime* 为 `False` 时应用。默认值为 `False`。

email.utils.**format_datetime**(*dt*, *usegmt=False*)

类似于 `formatdate`，但输入的是一个*datetime* 实例。如果实例是一个简单型 datetime，它会被视为" 不带源时区信息的 UTC"，并且使用传统的 `-0000` 作为时区。如果实例是一个感知型 `datetime`，则会使用数字形式的时区时差。如果实例是感知型且时区时差为零，则 *usegmt* 可能会被设为 `True`，在这种情况下将使用字符串 `GMT` 而非数字形式的时区时差。这提供了一种生成符合标准 HTTP 日期标头的方式。

3.3 版新加入.

email.utils.**decode_rfc2231**(*s*)

根据 **RFC 2231** 解码字符串 *s*。

email.utils.**encode_rfc2231**(*s*, *charset=None*, *language=None*)

根据 **RFC 2231** 对字符串 *s* 进行编码。可选的 *charset* 和 *language* 如果给出，则为指明要使用的字符集名称和语言名称。如果两者均未给出，则会原样返回 *s*。如果给出 *charset* 但未给出 *language*，则会使用空字符串作为 *language* 值来对字符串进行编码。

email.utils.**collapse_rfc2231_value**(*value*, *errors='replace'*, *fallback_charset='us-ascii'*)

当以 **RFC 2231** 格式来编码标头形参时，*Message.get_param* 可能返回一个包含字符集、语言和值的 3 元组。*collapse_rfc2231_value()* 会将此返回为一个 unicode 字符串。可选的 *errors* 会被传递给*str* 的*encode()* 方法的 *errors* 参数；它的默认值为 `'replace'`。可选的 *fallback_charset* 指定当 **RFC 2231** 标头中的字符集无法被 Python 识别时要使用的字符集；它的默认值为 `'us-ascii'`。

为方便起见，如果传给*collapse_rfc2231_value()* 的 *value* 不是一个元组，则应为一个字符串并会将其原样返回。

email.utils.**decode_params**(*params*)

根据 **RFC 2231** 解码参数列表。*params* 是一个包含 `(content-type, string-value)` 形式的元素的 2 元组的序列。

备注

19.1.15 email.iterators: 迭代器

源代码: Lib/email/iterators.py

通过*Message.walk* 方法来迭代消息对象树是相当容易的。*email.iterators* 模块提供了一些适用于消息对象树的高层级迭代器。

email.iterators.**body_line_iterator**(*msg*, *decode=False*)

此函数会迭代 *msg* 的所有子部分中的所有载荷，逐行返回字符串载荷。它会跳过所有子部分的标头，并且它也会跳过任何包含不为 Python 字符串的载荷的子部分。这基本上等价于使用*readline()* 从一个文件读取消息的纯文本表示形式，并跳过所有中间的标头。

可选的 *decode* 会被传递给*Message.get_payload*。

email.iterators.**typed_subpart_iterator**(*msg*, *maintype='text'*, *subtype=None*)
此函数会迭代 *msg* 的所有子部分，只返回其中与 *maintype* 和 *subtype* 所指定的 MIME 类型相匹配的子部分。

请注意 *subtype* 是可选项；如果省略，则仅使用主类型来进行子部分 MIME 类型的匹配。*maintype* 也是可选项；它的默认值为 `text`。

因此，在默认情况下`typed_subpart_iterator()` 会返回每一个 MIME 类型为 `text/*` 的子部分。

增加了以下函数作为有用的调试工具。它 不应当被视为该包所支持的公共接口的组成部分。

email.iterators.**_structure**(*msg*, *fp=None*, *level=0*, *include_default=False*)
打印消息对象结构的内容类型的缩进表示形式。例如:

```
>>> msg = email.message_from_file(somefile)
>>> _structure(msg)
multipart/mixed
    text/plain
    text/plain
    multipart/digest
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
        message/rfc822
            text/plain
    text/plain
```

可选项 *fp* 是一个作为打印输出目标的文件类对象。它必须适用于 Python 的`print()` 函数。*level* 是供内部使用的。*include_default* 如果为真值，则会同时打印默认类型。

也参考:

***smtplib* 模块** SMTP (简单邮件传输协议) 客户端

***poplib* 模块** POP (邮局协议) 客户端

***imaplib* 模块** IMAP (互联网消息访问协议) 客户端

***nntplib* 模块** NNTP (网络新闻传输协议) 客户端

***mailbox* 模块** 创建、读取和管理使用保存在磁盘中的多种标准格式的消息集的工具。

***smtpd* 模块** SMTP 服务器框架 (主要适用于测试)

19.2 `json` --- JSON 编码和解码器

源代码: Lib/json/__init__.py

JSON (JavaScript Object Notation), specified by **RFC 7159** (which obsoletes **RFC 4627**) and by ECMA-404, is a lightweight data interchange format inspired by JavaScript object literal syntax (although it is not a strict subset of JavaScript[1]).

[1] 正如 RFC 7159 的勘误表 所说明的，JSON 允许以字符串表示字面值字符 U+2028 (LINE SEPARATOR) 和 U+2029 (PARAGRAPH SEPARATOR)，而 JavaScript (在 ECMAScript 5.1 版中) 不允许。

警告： Be cautious when parsing JSON data from untrusted sources. A malicious JSON string may cause the decoder to consume considerable CPU and memory resources. Limiting the size of data to be parsed is recommended.

json 提供了与标准库*marshal* 和*pickle* 相似的 API 接口。

对基本的 Python 对象层次结构进行编码：

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
>>> print(json.dumps("\"foo\bar"))
"\"foo\bar"
>>> print(json.dumps('\u1234'))
"\u1234"
>>> print(json.dumps('\\'))
"\\"
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump(['streaming API'], io)
>>> io.getvalue()
'["streaming API"]'
```

紧凑编码：

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
```

美化輸出：

```
>>> import json
>>> print(json.dumps({'4': 5, '6': 7}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

JSON 解码：

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar"')
'"foo\x08ar'
>>> from io import StringIO
>>> io = StringIO('["streaming API"]')
>>> json.load(io)
['streaming API']
```

特殊 JSON 对象解码：

```
>>> import json
>>> def as_complex(dct):
...     if '__complex__' in dct:
```

(下页继续)

(繼續上一頁)

```
...         return complex(dct['real'], dct['imag'])
...     return dct
...
>>> json.loads('{"__complex__": true, "real": 1, "imag": 2}',
...     object_hook=as_complex)
(1+2j)
>>> import decimal
>>> json.loads('1.1', parse_float=decimal.Decimal)
Decimal('1.1')
```

扩展*JSONEncoder*:

```
>>> import json
>>> class ComplexEncoder(json.JSONEncoder):
...     def default(self, obj):
...         if isinstance(obj, complex):
...             return [obj.real, obj.imag]
...         # Let the base class default method raise the TypeError
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(2 + 1j, cls=ComplexEncoder)
'[2.0, 1.0]'
>>> ComplexEncoder().encode(2 + 1j)
'[2.0, 1.0]'
>>> list(ComplexEncoder().iterencode(2 + 1j))
['[2.0', ', 1.0', ']']
```

从命令行使用*json.tool* 来验证并美化输出:

```
$ echo '{"json":"obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

详细文档请参见命令行界面。

備𝔽: JSON 是 YAML 1.2 的一个子集。由该模块的默认设置生成的 JSON （尤其是默认的“分隔符”设置值）也是 YAML 1.0 and 1.1 的一个子集。因此该模块也能够用于序列化为 YAML。

備𝔽: 这个模块的编码器和解码器默认保护输入和输出的顺序。仅当底层的容器未排序时才会失去顺序。

19.2.1 基本使用

json.**dump**(*obj*, *fp*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

使用这个转换表 将 *obj* 序列化为 JSON 格式化流形式的 *fp* (支持 `.write()` 的*file-like object*)。

如果 *skipkeys* 是 true（默认为 `False`），那么那些不是基本对象（包括*str*、*int*、*float*、*bool*、None）的字典的键会被跳过；否则引发一个*TypeError*。

json 模块始终产生*str* 对象而非*bytes* 对象。因此，`fp.write()` 必须支持*str* 输入。

如果 *ensure_ascii* 是 true （即默认值），输出保证将所有输入的非 ASCII 字符转义。如果 *ensure_ascii* 是 false，这些字符会原样输出。

If *check_circular* is false (default: `True`), then the circular reference check for container types will be skipped and a circular reference will result in an *RecursionError* (or worse).

如果 *allow_nan* 是 false（默认为 `True`），那么在对严格 JSON 规格范围外的*float* 类型值（`nan`、`inf` 和 `-inf`）进行序列化时会引发一个*ValueError*。如果 *allow_nan* 是 true，则使用它们的 JavaScript 等价形式（`NaN`、`Infinity` 和 `-Infinity`）。

如果 *indent* 是一个非负整数或者字符串，那么 JSON 数组元素和对象成员会被美化输出为该值指定的缩进等级。如果缩进等级为零、负数或者 `""`，则只会添加换行符。`None` (默认值) 选择最紧凑的表达。使用一个正整数会让每一层缩进同样数量的空格。如果 *indent* 是一个字符串 (比如 `"\t"`)，那个字符串会被用于缩进每一层。

3.2 版更變: 允许使用字符串作为 *indent* 而不再仅仅是整数。

当指定时，*separators* 应当是一个 `(item_separator, key_separator)` 元组。当 *indent* 为 `None` 时，默认值取 `(', ', ': ')`，否则取 `(',', ': ')`。为了得到最紧凑的 JSON 表达式，你应该指定其为 `(',', ':')` 以消除空白字符。

3.4 版更變: 现当 *indent* 不是 `None` 时，采用 `(',', ': ')` 作为默认值。

当 *default* 被指定时，其应该是一个函数，每当某个对象无法被序列化时它会被调用。它应该返回该对象的一个可以被 JSON 编码的版本或者引发一个*TypeError*。如果没有被指定，则会直接引发*TypeError*。

如果 *sort_keys* 是 true（默认为 `False`），那么字典的输出会以键的顺序排序。

为了使用一个自定义的*JSONEncoder* 子类（比如：覆盖了 `default()` 方法来序列化额外的类型），通过 *cls* 关键字参数来指定；否则将使用*JSONEncoder*。

3.6 版更變: 所有的可选参数现在是*keyword-only* 的了。

備🄵: 与*pickle* 和*marshal* 不同，JSON 不是一个具有框架的协议，所以尝试多次使用同一个 *fp* 调用*dump()* 来序列化多个对象会产生一个不合规的 JSON 文件。

---

json.**dumps**(*obj*, *, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *cls=None*, *indent=None*, *separators=None*, *default=None*, *sort_keys=False*, ***kw*)

使用这个转换表 将 *obj* 序列化为 JSON 格式的*str*。其参数的含义与*dump()* 中的相同。

備🄵: JSON 中的键-值对中的键永远是*str* 类型的。当一个对象被转化为 JSON 时，字典中所有的键都会被强制转换为字符串。这所造成的结果是字典被转换为 JSON 然后转换回字典时可能和原来的不相等。换句话说，如果 x 具有非字符串的键，则有 `loads(dumps(x)) != x`。

---

json.**load**(*fp*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

使用这个转换表 将 *fp* (一个支持 `.read()` 并包含一个 JSON 文档的*text file* 或者*binary file*) 反序列化为一个 Python 对象。

object_hook 是一个可选的函数，它会被调用于每一个解码出的对象字面量（即一个`dict`）。*object_hook* 的返回值会取代原本的`dict`。这一特性能够被用于实现自定义解码器（如 JSON-RPC 的类型提示)。

object_pairs_hook 是一个可选的函数，它会被调用于每一个有序列表对解码出的对象字面量。*object_pairs_hook* 的返回值将会取代原本的`dict` 。这一特性能够被用于实现自定义解码器。如果 *object_hook* 也被定义，*object_pairs_hook* 优先。

3.1 版更變: 添加了对 *object_pairs_hook* 的支持。

parse_float ，如果指定，将与每个要解码 JSON 浮点数的字符串一同调用。默认状态下，相当于`float(num_str)` 。可以用于对 JSON 浮点数使用其它数据类型和语法分析程序（比如`decimal.Decimal`)。

parse_int ，如果指定，将与每个要解码 JSON 整数的字符串一同调用。默认状态下，相当于`int(num_str)` 。可以用于对 JSON 整数使用其它数据类型和语法分析程序（比如`float`)。

3.9.14 版更變: The default *parse_int* of `int()` now limits the maximum length of the integer string via the interpreter's *integer string conversion length limitation* to help avoid denial of service attacks.

parse_constant ，如果指定，将要与以下字符串中的一个一同调用: `'-Infinity'` ，`'Infinity'` ，`'NaN'` 。如果遇到无效的 JSON 数字则可以使用它引发异常。

3.1 版更變: *parse_constant* 不再调用'null' ，'true' ，'false' 。

要使用自定义的`JSONDecoder` 子类，用 `cls` 指定他；否则使用`JSONDecoder` 。额外的关键词参数会通过类的构造函数传递。

如果反序列化的数据不是有效 JSON 文档，引发`JSONDecodeError` 错误。

3.6 版更變: 所有的可选参数现在是*keyword-only* 的了。

3.6 版更變: *fp* 现在可以是*binary file* 。输入编码应当是 UTF-8 ，UTF-16 或者 UTF-32 。

json.**loads**(*s*, *, *cls=None*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *object_pairs_hook=None*, ***kw*)

使用这个转换表 将 *s* (一个包含 JSON 文档的`str`, `bytes` 或`bytearray` 实例) 反序列化为 Python 对象。

其他参数的含义与`load()` 中的相同。

如果反序列化的数据不是有效 JSON 文档，引发`JSONDecodeError` 错误。

3.6 版更變: *s* 现在可以为`bytes` 或`bytearray` 类型。输入编码应为 UTF-8, UTF-16 或 UTF-32。

3.9 版更變: 关键字参数 *encoding* 已被移除。

19.2.2 编码器和解码器

class json.**JSONDecoder**(*, *object_hook=None*, *parse_float=None*, *parse_int=None*, *parse_constant=None*, *strict=True*, *object_pairs_hook=None*)

简单的 JSON 解码器。

默认情况下，解码执行以下翻译:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

它还将“NaN”、“Infinity”和“-Infinity”理解为它们对应的“float”值，这超出了 JSON 规范。

如果指定了 *object_hook*，它将被调用并传入每个已解码 JSON 对象的结果，并且其返回值将被用来替代给定的`dict`。它可被用于提供自定义的反序列化操作（例如支持 JSON-RPC 类提示）。

如果指定了 *object_pairs_hook* 则它将被调用并传入以对照值有序列表进行解码的每个 JSON 对象的结果。*object_pairs_hook* 的结果值将被用来替代`dict`。这一特性可被用于实现自定义解码器。如果还定义了 *object_hook*，则 *object_pairs_hook* 的优先级更高。

3.1 版更變: 添加了对 *object_pairs_hook* 的支持。

parse_float，如果指定，将与每个要解码 JSON 浮点数的字符串一同调用。默认状态下，相当于`float(num_str)`。可以用于对 JSON 浮点数使用其它数据类型和语法分析程序（比如`decimal.Decimal`）。

parse_int，如果指定，将与每个要解码 JSON 整数的字符串一同调用。默认状态下，相当于`int(num_str)`。可以用于对 JSON 整数使用其它数据类型和语法分析程序（比如`float`）。

parse_constant，如果指定，将要与以下字符串中的一个一同调用：`'-Infinity'`，`'Infinity'`，`'NaN'`。如果遇到无效的 JSON 数字则可以使用它引发异常。

如果 *strict* 为 false（默认为`True`），那么控制字符将被允许在字符串内。在此上下文中的控制字符编码在范围 0--31 内的字符，包括`'\t'`(制表符)，`'\n'`，`'\r'`和`'\0'`。

如果反序列化的数据不是有效 JSON 文档，引发`JSONDecodeError`错误。

3.6 版更變: 所有形参现在都是仅限关键字参数。

decode(*s*)
返回 *s* 的 Python 表示形式（包含一个 JSON 文档的`str`实例）。

如果给定的 JSON 文档无效则将引发`JSONDecodeError`。

raw_decode(*s*)
从 *s* 中解码出 JSON 文档（以 JSON 文档开头的一个`str`对象）并返回一个 Python 表示形式为 2 元组以及指明该文档在 *s* 中结束位置的序号。

这可以用于从一个字符串解码 JSON 文档，该字符串的末尾可能有无关的数据。

class json.**JSONEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)
用于 Python 数据结构的可扩展 JSON 编码器。

默认支持以下对象和类型：

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int 和 float 派生的枚举	number
True	true
False	false
None	null

3.4 版更變: 添加了对 int 和 float 派生的枚举类的支持

为了将其拓展至识别其他对象，需要子类化并实现*default()* 方法于另一种返回 o 的可序列化对象的方法如果可行，否则它应该调用超类实现（来引发*TypeError* ）。

如果 *skipkeys* 为假值（默认），则当尝试对非*str*, *int*, *float* 或 None 的键进行编码时将会引发*TypeError*。如果 *skipkeys* 为真值，这些条目将被直接跳过。

如果 *ensure_ascii* 是 true （即默认值），输出保证将所有输入的非 ASCII 字符转义。如果 *ensure_ascii* 是 false，这些字符会原样输出。

If *check_circular* is true (the default), then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an *RecursionError*). Otherwise, no such check takes place.

如果 *allow_nan* 为 true （默认），那么 NaN ， Infinity ， 和 -Infinity 进行编码。此行为不符合 JSON 规范，但与大多数的基于 Javascript 的编码器和解码器一致。否则，它将是一个*ValueError* 来编码这些浮点数。

如果 *sort_keys* 为 true （默认为：False ），那么字典的输出是按照键排序；这对回归测试很有用，以确保可以每天比较 JSON 序列化。

如果 *indent* 是一个非负整数或者字符串，那么 JSON 数组元素和对象成员会被美化输出为该值指定的缩进等级。如果缩进等级为零、负数或者 ""，则只会添加换行符。None (默认值) 选择最紧凑的表达。使用一个正整数会让每一层缩进同样数量的空格。如果 *indent* 是一个字符串 (比如 "\t")，那个字符串会被用于缩进每一层。

3.2 版更變: 允许使用字符串作为 *indent* 而不再仅仅是整数。

当指定时，*separators* 应当是一个 (item_separator, key_separator) 元组。当 *indent* 为 None 时，默认值取 (', ', ': ')，否则取 (',', ': ')。为了得到最紧凑的 JSON 表达式，你应该指定其为 (',', ':') 以消除空白字符。

3.4 版更變: 现当 *indent* 不是 None 时，采用 (',', ': ') 作为默认值。

当 *default* 被指定时，其应该是一个函数，每当某个对象无法被序列化时它会被调用。它应该返回该对象的一个可以被 JSON 编码的版本或者引发一个*TypeError*。如果没有被指定，则会直接引发*TypeError*。

3.6 版更變: 所有形参现在都是仅限关键字参数。

default(*o*)

在子类中实现这种方法使其返回 *o* 的可序列化对象，或者调用基础实现（引发*TypeError* ）。

例如，为了支持任意的迭代器，你可以这样来实现*default()*:

```
def default(self, o):
   try:
       iterable = iter(o)
   except TypeError:
```

(下页继续)

(繼續上一頁)

```
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return json.JSONEncoder.default(self, o)
```

encode(*o*)
返回 Python *o* 数据结构的 JSON 字符串表达方式。例如:

```
>>> json.JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

iterencode(*o*)
编码给定对象 *o* ，并且让每个可用的字符串表达方式。例如:

```
for chunk in json.JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

19.2.3 例外

exception json.**JSONDecodeError**(*msg*, *doc*, *pos*)
拥有以下附加属性的*ValueError* 的子类:

msg
未格式化的错误消息。

doc
正在解析的 JSON 文档。

pos
The start index of *doc* where parsing failed.

lineno
The line corresponding to *pos*.

colno
The column corresponding to *pos*.

3.5 版新加入.

19.2.4 标准符合性和互操作性

The JSON format is specified by **RFC 7159** and by ECMA-404. This section details this module's level of compliance with the RFC. For simplicity, *JSONEncoder* and *JSONDecoder* subclasses, and parameters other than those explicitly mentioned, are not considered.

此模块不严格遵循于 RFC ，它实现了一些扩展是有效的 Javascript 但不是有效的 JSON。尤其是:

- 无限和 NaN 数值是被接受并输出;
- 对象内的重复名称是接受的，并且仅使用最后一对属性-值对的值。

自从 RFC 允许符合 RFC 的语法分析程序接收不符合 RFC 的输入文本以来，这个模块的解串器在默认状态下默认符合 RFC 。

字符编码

RFC 要求使用 UTF-8，UTF-16，或 UTF-32 之一来表示 JSON，为了最大互通性推荐使用 UTF-8。

RFC 允许，尽管不是必须的，这个模块的序列化默认设置为 *ensure_ascii=True*，这样消除输出以便结果字符串至容纳 ASCII 字符。

ensure_ascii 参数以外，此模块是严格的按照在 Python 对象和*Unicode strings* 间的转换定义的，并且因此不能直接解决字符编码的问题。

RFC 禁止添加字符顺序标记（BOM）在 JSON 文本的开头，这个模块的序列化器不添加 BOM 标记在它的输出上。RFC，准许 JSON 反序列化器忽略它们输入中的初始 BOM 标记，但不要求。此模块的反序列化器引发*ValueError* 当存在初始 BOM 标记。

RFC 不会明确禁止包含字节序列的 JSON 字符串这不对应有效的 Unicode 字符（比如不成对的 UTF-16 的替代物），但是它确实指出它们可能会导致互操作性问题。默认下，模块对这样的序列接受和输出（当在原始*str* 存在时）代码点。

Infinite 和 NaN 数值

RFC 不允许 infinite 或者 NaN 数值的表达方式。尽管这样，默认情况下，此模块接受并且输出 `Infinity`，`-Infinity`，和 `NaN` 好像它们是有效的 JSON 数字字面值

```
>>> # Neither of these calls raises an exception, but the results are not valid JSON
>>> json.dumps(float('-inf'))
'-Infinity'
>>> json.dumps(float('nan'))
'NaN'
>>> # Same when deserializing
>>> json.loads('-Infinity')
-inf
>>> json.loads('NaN')
nan
```

序列化器中，*allow_nan* 参数可用于替代这个行为。反序列化器中，*parse_constant* 参数，可用于替代这个行为。

对象中的重复名称

RFC 具体说明了在 JSON 对象里的名字应该是唯一的，但没有规定如何处理 JSON 对象中的重复名称。默认下，此模块不引发异常；作为替代，对于给定名它将忽略除姓-值对之外的所有对:

```
>>> weird_json = '{"x": 1, "x": 2, "x": 3}'
>>> json.loads(weird_json)
{'x': 3}
```

The *object_pairs_hook* parameter can be used to alter this behavior.

顶级非对象，非数组值

过时的 **RFC 4627** 指定的旧版本 JSON 要求 JSON 文本顶级值必须是 JSON 对象或数组（Python *dict* 或*list*），并且不能是 JSON null 值，布尔值，数值或者字符串值。**RFC 7159** 移除这个限制，此模块没有并且从未在序列化器和反序列化器中实现这个限制。

无论如何，为了最大化地获取互操作性，你可能希望自己遵守该原则。

实现限制

一些 JSON 反序列化器的实现应该在以下方面做出限制：

- 可接受的 JSON 文本大小
- 嵌套 JSON 对象和数组的最高水平
- JSON 数字的范围和精度
- JSON 字符串的内容和最大长度

此模块不强制执行任何上述限制，除了相关的 Python 数据类型本身或者 Python 解释器本身的限制以外。

当序列化为 JSON ，在应用中当心此类限制这可能破坏你的 JSON 。特别是，通常将 JSON 数字反序列化为 IEEE 754 双精度数字，从而受到该表示方式的范围和精度限制。这是特别相关的，当序列化非常大的 Python *int* 值时，或者当序列化”exotic” 数值类型的实例时比如*decimal.Decimal* 。

19.2.5 命令行界面

源代码： Lib/json/tool.py

The *json.tool* module provides a simple command line interface to validate and pretty-print JSON objects.

如果未指定可选的 `infile` 和 `outfile` 参数，则将分别使用*sys.stdin* 和*sys.stdout*:

```
$ echo '{"json": "obj"}' | python -m json.tool
{
    "json": "obj"
}
$ echo '{1.2:3.4}' | python -m json.tool
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
```

3.5 版更變: 输出现在将与输入顺序保持一致。请使用*--sort-keys* 选项来将输出按照键的字母顺序排序。

命令行选项

infile

要被验证或美化打印的 JSON 文件：

```
$ python -m json.tool mp_films.json
[
    {
        "title": "And Now for Something Completely Different",
        "year": 1971
    },
    {
```

(下页继续)

(繼續上一頁)

```
        "title": "Monty Python and the Holy Grail",
        "year": 1975
    }
]
```

如果 *infile* 未指定，则从*sys.stdin* 读取。

outfile
将 *infile* 输出写入到给定的 *outfile*。在其他情况下写入到*sys.stdout*。

--sort-keys
将字典输出按照键的字母顺序排序。

3.5 版新加入.

--no-ensure-ascii
禁用非 ASCII 字符的转义，详情参见*json.dumps()*。

3.9 版新加入.

--json-lines
将每个输入行解析为单独的 JSON 对象。

3.8 版新加入.

--indent, --tab, --no-indent, --compact
用于空白符控制的互斥选项。

3.9 版新加入.

-h, --help
显示帮助消息。

F解

19.3 mailbox --- 操作多种格式的邮箱

源代码： Lib/mailbox.py

本模块定义了两个类，*Mailbox* 和*Message*，用于访问和操作磁盘中的邮箱及其所包含的电子邮件。*Mailbox* 提供了类似字典的从键到消息的映射。*Message* 为*email.message* 模块的*Message* 类增加了特定格式专属的状态和行为。支持的邮箱格式有 Maildir, mbox, MH, Babyl 以及 MMDF。

也参考：

模块*email* 表示和操作邮件消息。

19.3.1 Mailbox 对象

class mailbox.**Mailbox**

一个邮箱，它可以被检视和修改。

Mailbox 类定义了一个接口并且它不应被实例化。而是应该让格式专属的子类继承*Mailbox* 并且你的代码应当实例化一个特定的子类。

Mailbox 接口类似于字典，其中每个小键都有对应的消息。键是由*Mailbox* 实例发出的，它们将由实例来使用并且只对该*Mailbox* 实例有意义。键会持续标识一条消息，即使对应的消息已被修改，例如被另一条消息所替代。

可以使用 set 型方法*add()* 将消息添加到*Mailbox* 并可以使用 del 语句或 set 型方法*remove()* 和*discard()* 将其移除。

Mailbox 接口语义在某些值得注意的方面与字典语义有所不同。每次请求消息时，都会基于邮箱的当前状态生成一个新的表示形式（通常为*Message* 实例）。类似地，当向*Mailbox* 实例添加消息时，所提供的消息表示形式的内容将被复制。无论在哪种情况下*Mailbox* 实例都不会保留对消息表示形式的引用。

默认的*Mailbox* 迭代器会迭代消息表示形式，而不像默认的字典迭代器那样迭代键。此外，在迭代期间修改邮箱是安全且有明确定义的。在创建迭代器之后被添加到邮箱的消息将对该迭代不可见。在迭代器产出消息之前被从邮箱移除的消息将被静默地跳过，但是使用来自迭代器的键也有可能导致*KeyError* 异常，如果对应的消息后来被移除的话。

> **警告:** 在修改可能同时被其他某个进程修改的邮箱时要非常小心。用于此种任务的最安全邮箱格式是 Maildir；请尽量避免使用 mbox 之类的单文件格式进行并发写入。如果你正在修改一个邮箱，你 *必须* 在读取文件中的任何消息或者执行添加或删除消息等修改操作 *之前* 通过调用*lock()* 以及*unlock()* 方法来锁定它。如果未锁定邮箱则将导致丢失消息或损坏整个邮箱的风险。

Mailbox 实例具有下列方法:

add(*message*)

将 *message* 添加到邮箱并返回分配给它的键。

形参 *message* 可以是*Message* 实例、*email.message.Message* 实例、字符串、字节串或文件类对象（应当以二进制模式打开）。如果 *message* 是适当的格式专属*Message* 子类的实例（举例来说，如果它是一个*mboxMessage* 实例而这是一个*mbox* 实例），将使用其格式专属的信息。在其他情况下，则会使用合理的默认值作为格式专属的信息。

3.2 版更變: 增加了对二进制输入的支持。

remove(*key*)
__delitem__(*key*)
discard(*key*)

从邮箱中删除对应于 *key* 的消息。

当消息不存在时，如果此方法是作为*remove()* 或*__delitem__()* 调用则会引发*KeyError* 异常，而如果此方法是作为*discard()* 调用则不会引发异常。如果下层邮箱格式支持来自其他进程的并发修改则*discard()* 的行为可能是更为适合的。

__setitem__(*key*, *message*)

将 *key* 所对应的消息替换为 *message*。如果没有与 *key* 所对应的消息则会引发*KeyError* 异常。

与*add()* 一样，形参 *message* 可以是*Message* 实例、*email.message.Message* 实例、字符串、字节串或文件类对象（应当以二进制模式打开）。如果 *message* 是适当的格式专属*Message* 子类的实例（举例来说，如果它是一个*mboxMessage* 实例而这是一个*mbox* 实例），将使用其格式专属的信息。在其他情况下，当前与 *key* 所对应的消息的格式专属信息则会保持不变。

iterkeys()
keys()
如果通过*iterkeys()* 调用则返回一个迭代所有键的迭代器，或者如果通过*keys()* 调用则返回一个键列表。

itervalues()
__iter__()
values()
如果通过*itervalues()* 或*__iter__()* 调用则返回一个迭代所有消息的表示形式的迭代器，或者如果通过*values()* 调用则返回一个由这些表示形式组成的列表。消息会被表示为适当的格式专属*Message* 子类的实例，除非当*Mailbox* 实例被初始化时指定了自定义的消息工厂函数。

備F: *__iter__()* 的行为与字典不同，后者是对键进行迭代。
---

iteritems()
items()
如果通过*iteritems()* 调用则返回一个迭代 (*key*, *message*) 对的迭代器，其中 *key* 为键而 *message* 为消息的表示形式，或者如果通过*items()* 调用则返回一个由这种键值对组成的列表。消息会被表示为适当的格式专属*Message* 子类的实例，除非当*Mailbox* 实例被初始化时指定了自定义的消息工厂函数。

get(*key*, *default=None*)
__getitem__(*key*)
返回对应于 *key* 的消息的表示形式。当对应的消息不存在时，如果通过*get()* 调用则返回 *default* 而如果通过*__getitem__()* 调用此方法则会引发*KeyError* 异常。消息会被表示为适当的格式专属*Message* 子类的实例，除非当*Mailbox* 实例被初始化时指定了自定义的消息工厂函数。

get_message(*key*)
将对应于 *key* 的消息的表示形式作为适当的格式专属*Message* 子类的实例返回，或者如果对应的消息不存在则会引发*KeyError* 异常。

get_bytes(*key*)
返回对应于 *key* 的消息的字节表示形式，或者如果对应的消息不存在则会引发*KeyError* 异常。

3.2 版新加入.

get_string(*key*)
返回对应于 *key* 的消息的字符串表示形式，或者如果对应的消息不存在则会引发*KeyError* 异常。消息是通过*email.message.Message* 处理来将其转换为纯 7bit 表示形式的。

get_file(*key*)
返回对应于 *key* 的消息的文件类表示形式，或者如果对应的消息不存在则会引发*KeyError* 异常。文件类对象的行为相当于以二进制模式打开。当不再需要此文件时应当将其关闭。

3.2 版更變: 此文件对象实际上是二进制文件；之前它被不正确地以文本模式返回。并且，此文件类对象现在还支持上下文管理协议：你可以使用 `with` 语句来自动关闭它。

備F: 不同于其他消息表示形式，文件类表示形式并不一定独立于创建它们的*Mailbox* 实例或下层的邮箱。每个子类都会提供更具体的文档。

__contains__(*key*)
如果 *key* 有对应的消息则返回 `True`，否则返回 `False`。

__len__()
返回邮箱中消息的数量。

clear()
从邮箱中删除所有消息。

pop(*key*, *default=None*)
返回对应于 *key* 的消息的表示形式并删除该消息。如果对应的消息不存在则返回 *default*。消息会被表示为适当的格式专属*Message* 子类的实例，除非当*Mailbox* 实例被初始化时指定了自定义的消息工厂函数。

popitem()
返回一个任意的 (*key*, *message*) 对，其中 *key* 为键而 *message* 为消息的表示形式，并删除对应的消息。如果邮箱为空，则会引发*KeyError* 异常。消息会被表示为适当的格式专属*Message* 子类的实例，除非当*Mailbox* 实例被初始化时指定了自定义的消息工厂函数。

update(*arg*)
形参 *arg* 应当是 *key* 到 *message* 的映射或 (*key*, *message*) 对的可迭代对象。用来更新邮箱以使得对于每个给定的 *key* 和 *message*，与 *key* 相对应的消息会被设为 *message*，就像通过使用*__setitem__()* 一样。类似于*__setitem__()*，每个 *key* 都必须在邮箱中有一个对应的消息否则将会引发*KeyError* 异常，因此在通常情况下将 *arg* 设为*Mailbox* 实例是不正确的。

備🄵: 与字典不同，关键字参数是不受支持的。
---

flush()
将所有待定的更改写入到文件系统。对于某些*Mailbox* 子类来说，更改总是被立即写入因而*flush()* 并不会做任何事，但您仍然应当养成调用此方法的习惯。

lock()
在邮箱上获取一个独占式咨询锁以使其他进程知道不能修改它。如果锁无法被获取则会引发*ExternalClashError*。所使用的具体锁机制取决于邮箱的格式。在对邮箱内容进行任何修改之前你应当 总是锁定它。

unlock()
释放邮箱上的锁，如果存在的话。

close()
刷新邮箱，如果必要则将其解锁。并关闭所有打开的文件。对于某些*Mailbox* 子类来说，此方法并不会做任何事。

Maildir

class mailbox.**Maildir**(*dirname*, *factory=None*, *create=True*)
Mailbox 的一个子类，用于 Maildir 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*MaildirMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

如果 *create* 为 `True` 且 *dirname* 路径存在，它将被视为已有的 maildir 而无需尝试验证其目录布局。

使用 *dirname* 这个名称而不使用 *path* 是出于历史原因。

Maildir 是一种基于目录的邮箱格式，它是针对 qmail 邮件传输代理而发明的，现在也得到了其他程序的广泛支持。Maildir 邮箱中的消息存储在一个公共目录结构中的单独文件内。这样的设计允许 Maildir 邮箱被多个彼此无关的程序访问和修改而不会导致数据损坏，因此文件锁定操作是不必要的。

Maildir 邮箱包含三个子目录，分别是: `tmp`, `new` 和 `cur`。消息会不时地在 `tmp` 子目录中创建然后移至 `new` 子目录来结束投递。后续电子邮件客户端可能将消息移至 `cur` 子目录并将有关消息状态的信息存储在附带到其文件名的特殊”info” 小节中。

Courier 电子邮件传输代理所引入的文件夹风格也是受支持的。主邮箱中任何子目录只要其名称的第一个字符是 '.' 就会被视为文件夹。文件夹名称会被*Maildir* 表示为不带前缀 '.' 的形式。每个文件夹自身都是一个 Maildir 邮箱但不应包含其他文件夹。逻辑嵌套关系是使用 '.' 来划定层级，例如”Archived.2005.07”。

備F: Maildir 规范要求使用在特定消息文件名中使用冒号 (':')。但是，某些操作系统不允许将此字符用于文件名，如果你希望在这些操作系统上使用类似 Maildir 的格式，你应当指定改用另一个字符。叹号 ('!') 是一个受欢迎的选择。例如:

```
import mailbox
mailbox.Maildir.colon = '!'
```

colon 属性也可以在每个实例上分别设置。

Maildir 实例具有*Mailbox* 的所有方法及下列附加方法:

list_folders()
返回所有文件夹名称的列表。

get_folder(*folder*)
返回表示名称为 *folder* 的文件夹的*Maildir* 实例。如果文件夹不存在则会引发*NoSuchMailboxError* 异常。

add_folder(*folder*)
创建名称为 *folder* 的文件夹并返回表示它的*Maildir* 实例。

remove_folder(*folder*)
删除名称为 *folder* 的文件夹。如果文件夹包含任何消息，则将引发*NotEmptyError* 异常且该文件夹将不会被删除。

clean()
从邮箱中删除最近 36 小时内未被访问过的临时文件。Maildir 规范要求邮件阅读程序应当时常进行此操作。

Maildir 所实现的某些*Mailbox* 方法值得进行特别的说明:

add(*message*)
__setitem__(*key*, *message*)
update(*arg*)

> **警告:** 这些方法会基于当前进程 ID 来生成唯一文件名。当使用多线程时，可能发生未被检测到的名称冲突并导致邮箱损坏，除非是对线程进行协调以避免使用这些方法同时操作同一个邮箱。

flush()
对 Maildir 邮箱的所有更改都会立即被应用，所以此方法并不会做任何事情。

lock()
unlock()
Maildir 邮箱不支持（或要求）锁定操作，所以此方法并不会做任何事情。

close()
Maildir 实例不保留任何打开的文件并且下层的邮箱不支持锁定操作，所以此方法不会做任何事情。

get_file(*key*)
根据主机平台的不同，当返回的文件保持打开状态时可能无法修改或移除下层的消息。

也参考:

Courier 上的 maildir 指南页面 该格式的规格说明。描述了用于支持文件夹的通用扩展。

使用 maildir 格式 Maildir 发明者对它的说明。包括已更新的名称创建方案和"info" 语义的相关细节。

mbox

class mailbox.**mbox**(*path*, *factory=None*, *create=True*)
Mailbox 的子类，用于 mbox 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*mboxMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

mbox 格式是在 Unix 系统上存储电子邮件的经典格式。mbox 邮箱中的所有消息都存储在一个单独文件中，每条消息的开头由前五个字符为"From " 的行来指明。

还有一些 mbox 格式变种对原始格式中发现的缺点做了改进，*mbox* 只实现原始格式，有时被称为 *mboxo*。这意味着当存储消息时 *Content-Length* 标头如果存在则会被忽略并且消息体中出现于行开头的任何"From " 会被转换为">From "，但是当读取消息时">From " 则不会被转换为"From "。

mbox 所实现的某些*Mailbox* 方法值得进行特别的说明:

get_file(*key*)
在*mbox* 实例上调用 `flush()` 或 `close()` 之后再使用文件可能产生无法预料的结果或者引发异常。

lock()
unlock()
使用三种锁机制 --- dot 锁，以及可能情况下的 `flock()` 和 `lockf()` 系统调用。

也参考:

tin 上的 mbox 指南页面 该格式的规格说明，包括有关锁的详情。

在 Unix 上配置 Netscape Mail: 为何 Content-Length 格式是不好的 使用原始 mbox 格式而非其变种的一些理由。

"mbox" 是由多个彼此不兼容的邮箱格式构成的家族 有关 mbox 变种的历史。

MH

class mailbox.**MH**(*path*, *factory=None*, *create=True*)
Mailbox 的子类，用于 MH 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*MHMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

MH 是一种基于目录的邮箱格式，它是针对 MH Message Handling System 电子邮件用户代理而发明的。在 MH 邮箱的每条消息都放在单独文件中。MH 邮箱中除了邮件消息还可以包含其他 MH 邮箱 (称为*文件夹*)。文件夹可以无限嵌套。MH 邮箱还支持 *序列*，这是一种命名列表，用来对消息进行逻辑分组而不必将其移入子文件夹。序列是在每个文件夹中名为 `.mh_sequences` 的文件内定义的。

MH 类可以操作 MH 邮箱，但它并不试图模拟 **mh** 的所有行为。特别地，它并不会修改 `context` 或 `.mh_profile` 文件也不会受其影响，这两个文件是 **mh** 用来存储状态和配置数据的。

MH 实例具有*Mailbox* 的所有方法及下列附加方法:

list_folders()
返回所有文件夹名称的列表。

get_folder(*folder*)
返回表示名称为 *folder* 的文件夹的*MH* 实例。如果文件夹不存在则会引发*NoSuchMailboxError* 异常。

add_folder(*folder*)
创建名称为 *folder* 的文件夹并返回表示它的*MH* 实例。

remove_folder(*folder*)
删除名称为 *folder* 的文件夹。如果文件夹包含任何消息，则将引发*NotEmptyError* 异常且该文件夹将不会被删除。

get_sequences()
返回映射到键列表的序列名称字典。如果不存在任何序列，则返回空字典。

set_sequences(*sequences*)
根据由映射到键列表的名称组成的字典 *sequences* 来重新定义邮箱中的序列，该字典与*get_sequences()* 返回值的形式一样。

pack()
根据需要重命名邮箱中的消息以消除序号中的空缺。序列列表中的条目会做相应的修改。

備F: 已发送的键会因此操作而失效并且不应当被继续使用。

MH 所实现的某些*Mailbox* 方法值得进行特别的说明:

remove(*key*)
__delitem__(*key*)
discard(*key*)
这些方法会立即删除消息。通过在名称前加缀一个冒号作为消息删除标记的 MH 惯例不会被使用。

lock()
unlock()
使用三种锁机制 --- dot 锁，以及可能情况下下 `flock()` 和 `lockf()` 系统调用。对于 MH 邮箱来说，锁定邮箱意味着锁定 `.mh_sequences` 文件，并且仅在执行会影响单独消息文件的操作期间锁定单独消息文件。

get_file(*key*)
根据主机平台的不同，当返回的文件保持打开状态时可能无法移除下层的消息。

flush()
对 MH 邮箱的所有更改都会立即被应用，所以此方法并不会做任何事情。

close()
MH 实例不保留任何打开的文件，所以此方法等价于*unlock()*。

也参考:

nmh - Message Handling System **nmh** 的主页，这是原始 **mh** 的更新版本。

MH & nmh: Email for Users & Programmers 使用 GPL 许可证的介绍 **mh** 与 **nmh** 的图书，包含有关该邮箱格式的各种信息。

Babyl

class mailbox.**Babyl**(*path*, *factory=None*, *create=True*)

Mailbox 的子类，用于 Babyl 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*BabylMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

Babyl 是 Rmail 电子邮箱用户代理所使用单文件邮箱格式，包括在 Emacs 中。每条消息的开头由一个包含 Control-Underscore (`'\037'`) 和 Control-L (`'\014'`) 这两个字符的行来指明。消息的结束由下一条消息的开头来指明，或者当为最后一条消息时则由一个包含 Control-Underscore (`'\037'`) 字符的行来指明。

Babyl 邮箱中的消息带有两组标头：原始标头和所谓的可见标头。可见标头通常为原始标头经过重格式化和删减以更易读的子集。Babyl 邮箱中的每条消息都附带了一个 标签列表，即记录消息相关额外信息的短字符串，邮箱中所有的用户定义标签列表会存储于 Babyl 的选项部分。

Babyl 实例具有*Mailbox* 的所有方法及下列附加方法:

get_labels()

返回邮箱中使用的所有用户定义标签名称的列表。

備🄴: 邮箱中存在哪些标签会通过检查实际的消息而非查询 Babyl 选项部分的标签列表，但 Babyl 选项部分会在邮箱被修改时更新。

Babyl 所实现的某些*Mailbox* 方法使得进行特别的说明:

get_file(*key*)

在 Babyl 邮箱中，消息的标头并不是与消息体存储在一起的。要生成文件类表示形式，标头和消息体会被一起拷贝到一个*io.BytesIO* 实例中，它具有与文件相似的 API。因此，文件类对象实际上独立于下层邮箱，但与字符串表达形式相比并不会更节省内存。

lock()

unlock()

使用三种锁机制 --- dot 锁，以及可能情况下的 `flock()` 和 `lockf()` 系统调用。

也参考:

Format of Version 5 Babyl Files Babyl 格式的规格说明。

Reading Mail with Rmail Rmail 的帮助手册，包含了有关 Babyl 语义的一些信息。

MMDF

class mailbox.**MMDF**(*path*, *factory=None*, *create=True*)

Mailbox 的子类，用于 MMDF 格式的邮箱。形参 *factory* 是一个可调用对象，它接受一个文件类消息表示形式（其行为相当于以二进制模式打开）并返回一个自定义的表示形式。如果 *factory* 为 `None`，则会使用*MMDFMessage* 作为默认的消息表示形式。如果 *create* 为 `True`，则当邮箱不存在时会创建它。

MMDF 是一种专用于电子邮件传输代理 Multichannel Memorandum Distribution Facility 的单文件邮箱格式。每条消息使用与 mbox 消息相同的形式，但其前后各有包含四个 Control-A (`'\001'`) 字符的行。与 mbox 格式一样，每条消息的开头由一个前五个字符为”From ” 的行来指明，但当存储消息时额外出现的”From ” 不会被转换为”>From ” 因为附加的消息分隔符可防止将这些内容误认为是后续消息的开头。

MMDF 所实现的某些*Mailbox* 方法值得进行特别的说明:

get_file(*key*)

在*MMDF* 实例上调用 `flush()` 或 `close()` 之后再使用文件可能产生无法预料的结果或者引发异常。

lock()
unlock()
使用三种锁机制 --- dot 锁，以及可能情况下的 `flock()` 和 `lockf()` 系统调用。

也参考:

tin 上的 mmdf 指南页面 MMDF 格式的规格说明，来自新闻阅读器 tin 的文档。

MMDF 一篇描述 Multichannel Memorandum Distribution Facility 的维基百科文章。

19.3.2 Message 对象

class mailbox.**Message**(*message=None*)
email.message 模块的*Message* 的子类。*mailbox.Message* 的子类添加了特定邮箱格式专属的状态和行为。

如果省略了 *message*，则新实例会以默认的空状态被创建。如果 *message* 是一个*email.message.Message* 实例，其内容会被拷贝；此外，如果 *message* 是一个*Message* 实例，则任何格式专属信息会尽可能地被拷贝。如果 *message* 是一个字符串、字节串或文件，则它应当包含兼容 **RFC 2822** 的消息，该消息会被读取和解析。文档应当以二进制模式打开，但文本模式的文件也会被接受以向下兼容。

各个子类所提供的格式专属状态和行为各有不同，但总的来说只有那些不仅限于特定邮箱的特性才会被支持（虽然这些特性可能专属于特定邮箱格式）。例如，例如，单文件邮箱格式的文件偏移量和基于目录的邮箱格式的文件名都不会被保留，因为它们都仅适用于对应的原始邮箱。但消息是否已被用户读取或标记为重要等状态则会被保留，因为它们适用于消息本身。

不要求用*Message* 实例来表示使用*Mailbox* 实例所提取到的消息。在某些情况下，生成*Message* 表示形式所需的时间和内存空间可能是不可接受的。对于此类情况，*Mailbox* 实例还提供了字符串和文件类表示形式，并可在初始化*Mailbox* 实例时指定自定义的消息工厂函数。

MaildirMessage

class mailbox.**MaildirMessage**(*message=None*)
具有 Maildir 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

通常，邮件用户代理应用程序会在用户第一次打开并关闭邮箱之后将 `new` 子目录中的所有消息移至 `cur` 子目录，将这些消息记录为旧消息，无论它们是否真的已被阅读。`cur` 下的每条消息都有一个”info” 部分被添加到其文件名中以存储有关其状态的信息。（某些邮件阅读器还会把”info” 部分也添加到 `new` 下的消息中。）”info” 部分可以采用两种形式之一：它可能包含”2,” 后面跟一个经标准化的旗标列表（例如”2,FR”）或者它可能包含”1,” 后面跟所谓的实验性信息。Maildir 消息的标准旗标如下:

旗标	意义	解释
D	草稿	正在撰写中
F	已标记	已被标记为重要
P	已检视	转发，重新发送或退回
R	已回复	回复给
S	已查看	已阅读
T	已删除	标记为可被删除

MaildirMessage 实例提供以下方法:

get_subdir()
返回”new” (如果消息应当被存储在 `new` 子目录下) 或者”cur” (如果消息应当被存储在 `cur` 子目录下)。

> **備𝔽:** 消息通常会在其邮箱被访问后被从 `new` 移至 `cur`，无论该消息是否已被阅读。如果 msg.get_flags()“ 中的 `"S"` 为 ``True` 则说明消息 `msg` 已被阅读。

set_subdir(*subdir*)
设置消息应当被存储到的子目录。形参 *subdir* 必须为”new” 或”cur”。

get_flags()
返回一个指明当前所设旗标的字符串。如果消息符合标准的 Maildir 格式，则结果为零或按字母顺序各自出现一次的 `'D'`, `'F'`, `'P'`, `'R'`, `'S'` 和 `'T'` 的拼接。如果未设任何旗标或者如果”info” 包含实验性语义则返回空字符串。

set_flags(*flags*)
设置由 *flags* 所指定的旗标并重置所有其它旗标。

add_flag(*flag*)
设置由 *flag* 所指明的旗标而不改变其他旗标。要一次性添加一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。当前”info” 会被覆盖，无论它是否只包含实验性信息而非旗标。

remove_flag(*flag*)
重置由 *flag* 所指明的旗标而不改变其他旗标。要一次性移除一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。如果”info” 包含实验性信息而非旗标，则当前的”info” 不会被修改。

get_date()
以表示 Unix 纪元秒数的浮点数形式返回消息的发送日期。

set_date(*date*)
将消息的发送日期设为 *date*，一个表示 Unix 纪元秒数的浮点数。

get_info()
返回一个包含消息的”info” 的字符串。这适用于访问和修改实验性的”info” (即不是由旗标组成的列表)。

set_info(*info*)
将”info” 设为 *info*，这应当是一个字符串。

当一个*MaildirMessage* 实例基于*mboxMessage* 或*MMDFMessage* 实例被创建时，将会忽略 *Status* 和 *X-Status* 标头并进行下列转换:

结果状态	*mboxMessage* 或*MMDFMessage* 状态
”cur” 子目录	O 旗标
F 旗标	F 旗标
R 旗标	A 旗标
S 旗标	R 旗标
T 旗标	D 旗标

当一个*MaildirMessage* 实例基于*MHMessage* 实例被创建时，将进行下列转换:

结果状态	*MHMessage* 状态
”cur” 子目录	”unseen” 序列
”cur” 子目录和 S 旗标	非”unseen” 序列
F 旗标	”flagged” 序列
R 旗标	”replied” 序列

当一个*MaildirMessage* 实例基于*BabylMessage* 实例被创建时，将进行下列转换:

结果状态	*BabylMessage* 状态
”cur” 子目录	”unseen” 标签
”cur” 子目录和 S 旗标	非”unseen” 标签
P 旗标	”forwarded” 或”resent” 标签
R 旗标	”answered” 标签
T 旗标	”deleted” 标签

mboxMessage

class mailbox.**mboxMessage**(*message=None*)

具有 mbox 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

mbox 邮箱中的消息会一起存储在单个文件中。发件人的信封地址和发送时间通常存储在指明每条消息的起始的以”From ” 打头的行中，不过在 mbox 的各种实现之间此数据的确切格式具有相当大的差异。指明消息状态的各种旗标，例如是否已读或标记为重要等等通常存储在 *Status* 和 *X-Status* 标头中。

传统的 mbox 消息旗标如下:

旗标	意义	解释
R	已阅读	已阅读
O	旧消息	之前已经过 MUA 检测
D	已删除	标记为可被删除
F	已标记	已被标记为重要
A	已回复	回复给

”R” 和”O” 旗标存储在 *Status* 标头中，而”D”, ”F” 和”A” 旗标存储在 *X-Status* 标头中。旗标和标头通常会按上述顺序显示。

mboxMessage 实例提供了下列方法:

get_from()

返回一个表示在 mbox 邮箱中标记消息起始的”From ” 行的字符串。开头的”From ” 和末尾的换行符会被去除。

set_from(*from_*, *time_=None*)

将”From ” 行设为 *from_*，这应当被指定为不带开头的”From ” 或末尾的换行符。为方便起见，可以指定 *time_* 并将经过适当的格式化再添加到 *from_*。如果指定了 *time_*，它应当是一个*time.struct_time* 实例，适合传入*time.strftime()* 的元组或者 `True`(以使用*time.gmtime()*)。

get_flags()

返回一个指明当前所设旗标的字符串。如果消息符合规范格式，则结果为零或各自出现一次的 `'R'`, `'O'`, `'D'`, `'F'` 和 `'A'` 按上述顺序的拼接。

set_flags(*flags*)

设置由 *flags* 所指明的旗标并重启所有其他旗标。形参 *flags* 应当为零或各自出现多次的 `'R'`, `'O'`, `'D'`, `'F'` 和 `'A'` 按任意顺序的拼接。

add_flag(*flag*)

设置由 *flag* 所指明的旗标而不改变其他旗标。要一次性添加一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

remove_flag(*flag*)

重置由 *flag* 所指明的旗标而不改变其他旗标。要一次性移除一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

当一个*mboxMessage* 实例基于*MaildirMessage* 实例被创建时，”From ” 行会基于*MaildirMessage* 实例的发送时间被生成，并进行下列转换:

结果状态	*MaildirMessage* 状态
R 旗标	S 旗标
O 旗标	”cur” 子目录
D 旗标	T 旗标
F 旗标	F 旗标
A 旗标	R 旗标

当一个*mboxMessage* 实例基于*MHMessage* 实例被创建时，将进行下列转换:

结果状态	*MHMessage* 状态
R 旗标和 O 旗标	非”unseen” 序列
O 旗标	”unseen” 序列
F 旗标	”flagged” 序列
A 旗标	”replied” 序列

当一个*mboxMessage* 实例基于*BabylMessage* 实例被创建时，将进行下列转换:

结果状态	*BabylMessage* 状态
R 旗标和 O 旗标	非”unseen” 标签
O 旗标	”unseen” 标签
D 旗标	”deleted” 标签
A 旗标	”answered” 标签

当一个*Message* 实例基于*MMDFMessage* 实例被创建时，”From ” 行会被拷贝并直接对应所有旗标。

结果状态	*MMDFMessage* 状态
R 旗标	R 旗标
O 旗标	O 旗标
D 旗标	D 旗标
F 旗标	F 旗标
A 旗标	A 旗标

MHMessage

class mailbox.**MHMessage**(*message=None*)

具有 MH 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

MH 消息不支持传统意义上的标记或旗标，但它们支持序列，即对任意消息的逻辑分组。某些邮件阅读程序 (但不包括标准 **mh** 和 **nmh**) 以与其他格式使用旗标类似的方式来使用序列，如下所示:

序列	解释
unseen	未阅读，但之前已经过 MUA 检测
已回复	回复给
已标记	已被标记为重要

MHMessage 实例提供了下列方法:

get_sequences()
返回一个包含此消息的序列的名称的列表。

set_sequences(*sequences*)
设置包含此消息的序列的列表。

add_sequence(*sequence*)
将 *sequence* 添加到包含此消息的序列的列表。

remove_sequence(*sequence*)
将 *sequence* 从包含此消息的序列的列表中移除。

当一个*MHMessage* 实例基于*MaildirMessage* 实例被创建时，将进行下列转换:

结果状态	*MaildirMessage* 状态
"unseen" 序列	非 S 旗标
"replied" 序列	R 旗标
"flagged" 序列	F 旗标

当一个*MHMessage* 实例基于*mboxMessage* 或*MMDFMessage* 实例被创建时，将会忽略 *Status* 和 *X-Status* 标头并进行下列转换:

结果状态	*mboxMessage* 或*MMDFMessage* 状态
"unseen" 序列	非 R 旗标
"replied" 序列	A 旗标
"flagged" 序列	F 旗标

当一个*MHMessage* 实例基于*BabylMessage* 实例被创建时，将进入下列转换:

结果状态	*BabylMessage* 状态
"unseen" 序列	"unseen" 标签
"replied" 序列	"answered" 标签

BabylMessage

class mailbox.**BabylMessage**(*message=None*)
具有 Babyl 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

某些消息标签被称为 *属性*，根据惯例被定义为具有特殊的含义。这些属性如下所示:

标签	解释
unseen	未阅读，但之前已经过 MUA 检测
deleted	标记为可被删除
filed	复制到另一个文件或邮箱
answered	回复给
forwarded	已转发
edited	已被用户修改
resent	已重发

默认情况下，Rmail 只显示可见标头。不过*BabylMessage* 类会使用原始标头因为它们更完整。如果需要可以显式地访问可见标头。

BabylMessage 实例提供了下列方法:

get_labels()
返回邮件上的标签列表。

set_labels(*labels*)
将消息上的标签列表设置为 *labels* 。

add_label(*label*)
将 *label* 添加到消息上的标签列表中。

remove_label(*label*)
从消息上的标签列表中删除 *label* 。

get_visible()
返回一个*Message* 实例，其标头为消息的可见标头而其消息体为空。

set_visible(*visible*)
将消息的可见标头设为与 *message* 中的标头一致。形参 *visible* 应当是一个*Message* 实例，*email.message.Message* 实例，字符串或文件类对象（且应当以文本模式打开）。

update_visible()
当一个*BabylMessage* 实例的原始标头被修改时，可见标头不会自动进行对应修改。此方法将按以下方式更新可见标头：每个具有对应原始标头的可见标头会被设为原始标头的值，每个没有对应原始标头的可见标头会被移除，而任何存在于原始标头但不存在于可见标头中的 *Date*, *From*, *Reply-To*, *To*, *CC* 和 *Subject* 会被添加至可见标头。

当一个*BabylMessage* 实例基于*MaildirMessage* 实例被创建时，将进行下列转换:

结果状态	*MaildirMessage* 状态
”unseen” 标签	非 S 旗标
”deleted” 标签	T 旗标
”answered” 标签	R 旗标
”forwarded” 标签	P 旗标

当一个*BabylMessage* 实例基于*mboxMessage* 或*MMDFMessage* 实例被创建时，将会忽略 *Status* 和 *X-Status* 标头并进入下列转换:

结果状态	*mboxMessage* 或*MMDFMessage* 状态
”unseen” 标签	非 R 旗标
”deleted” 标签	D 旗标
”answered” 标签	A 旗标

当一个*BabylMessage* 实例基于*MHMessage* 实例被创建时，将进入下列转换:

结果状态	*MHMessage* 状态
”unseen” 标签	”unseen” 序列
”answered” 标签	”replied” 序列

MMDFMessage

class mailbox.**MMDFMessage**(*message=None*)

具有 MMDF 专属行为的消息。形参 *message* 的含义与*Message* 构造器一致。

与 mbox 邮箱中的消息类似，MMDF 消息会与将发件人的地址和发送日期作为以”From ” 打头的初始行一起存储。同样地，指明消息状态的旗标通常存储在 *Status* 和 *X-Status* 标头中。

传统的 MMDF 消息旗标与 mbox 消息的类似，如下所示:

旗标	意义	解释
R	已阅读	已阅读
O	旧消息	之前已经过 MUA 检测
D	已删除	标记为可被删除
F	已标记	已被标记为重要
A	已回复	回复给

”R” 和”O” 旗标存储在 *Status* 标头中，而”D”, ”F” 和”A” 旗标存储在 *X-Status* 标头中。旗标和标头通常会按上述顺序显示。

MMDFMessage 实例提供了下列方法，与*mboxMessage* 所提供的类似:

get_from()

返回一个表示在 mbox 邮箱中标记消息起始的”From ” 行的字符串。开头的”From ” 和末尾的换行符会被去除。

set_from(*from_*, *time_=None*)

将”From ” 行设为 *from_*，这应当被指定为不带开头的”From ” 或末尾的换行符。为方便起见，可以指定 *time_* 并将经过适当的格式化再添加到 *from_*。如果指定了 *time_*，它应当是一个*time.struct_time* 实例，适合传入*time.strftime()* 的元组或者 `True`(以使用*time.gmtime()*)。

get_flags()

返回一个指明当前所设旗标的字符串。如果消息符合规范格式，则结果为零或各自出现一次的 `'R'`, `'O'`, `'D'`, `'F'` 和 `'A'` 按上述顺序的拼接。

set_flags(*flags*)

设置由 *flags* 所指明的旗标并重启所有其他旗标。形参 *flags* 应当为零或各自出现多次的 `'R'`, `'O'`, `'D'`, `'F'` 和 `'A'` 按任意顺序的拼接。

add_flag(*flag*)

设置由 *flag* 所指明的旗标而不改变其他旗标。要一次性添加一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

remove_flag(*flag*)

重置由 *flag* 所指明的旗标而不改变其他旗标。要一次性移除一个以上的旗标，*flag* 可以为包含一个以上字符的字符串。

当一个*MMDFMessage* 实例基于*MaildirMessage* 实例被创建时，”From ” 行会基于*MaildirMessage* 实例的发送日期被生成，并进入下列转换:

结果状态	*MaildirMessage* 状态
R 旗标	S 旗标
O 旗标	”cur” 子目录
D 旗标	T 旗标
F 旗标	F 旗标
A 旗标	R 旗标

当一个*MMDFMessage* 实例基于*MHMessage* 实例被创建时，将进行下列转换:

结果状态	*MHMessage* 状态
R 旗标和 O 旗标	非”unseen” 序列
O 旗标	”unseen” 序列
F 旗标	”flagged” 序列
A 旗标	”replied” 序列

当一个*MMDFMessage* 实例基于*BabylMessage* 实例被创建时，将进行下列转换:

结果状态	*BabylMessage* 状态
R 旗标和 O 旗标	非”unseen” 标签
O 旗标	”unseen” 标签
D 旗标	”deleted” 标签
A 旗标	”answered” 标签

当一个*MMDFMessage* 实例基于*mboxMessage* 实例被创建时，”From ” 行会被拷贝并直接对应所有旗标:

结果状态	*mboxMessage* 状态
R 旗标	R 旗标
O 旗标	O 旗标
D 旗标	D 旗标
F 旗标	F 旗标
A 旗标	A 旗标

19.3.3 异常

mailbox 模块中定义了下列异常类:

exception mailbox.**Error**
所有其他模块专属异常的基类。

exception mailbox.**NoSuchMailboxError**
在期望获得一个邮箱但未找到时被引发，例如当使用不存在的路径来实例化一个*Mailbox* 子类时 (且将 *create* 形参设为 `False`)，或是当打开一个不存在的路径时。

exception mailbox.**NotEmptyError**
在期望一个邮箱为空但不为空时被引发，例如当删除一个包含消息的文件夹时。

exception mailbox.**ExternalClashError**
在某些邮箱相关条件超出了程序控制范围导致其无法继续运行时被引发，例如当要获取的锁已被另一个程序获取时，或是当要生成的唯一性文件名已存在时。

exception mailbox.**FormatError**
在某个文件中的数据无法被解析时被引发，例如当一个*MH* 实例尝试读取已损坏的 `.mh_sequences` 文件时。

19.3.4 示例

一个打印指定邮箱中所有消息的主题的简单示例:

```
import mailbox
for message in mailbox.mbox('~/mbox'):
    subject = message['subject']       # Could possibly be None.
    if subject and 'python' in subject.lower():
        print(subject)
```

要将所有邮件从 Babyl 邮箱拷贝到 MH 邮箱，请转换所有可转换的格式专属信息:

```
import mailbox
destination = mailbox.MH('~/Mail')
destination.lock()
for message in mailbox.Babyl('~/RMAIL'):
    destination.add(mailbox.MHMessage(message))
destination.flush()
destination.unlock()
```

这个示例将来自多个邮件列表的邮件分类放入不同的邮箱，小心避免由于其他程序的并发修改导致的邮件损坏，由于程序中断导致的邮件丢失，或是由于邮箱中消息格式错误导致的意外终止:

```
import mailbox
import email.errors

list_names = ('python-list', 'python-dev', 'python-bugs')

boxes = {name: mailbox.mbox('~/email/%s' % name) for name in list_names}
inbox = mailbox.Maildir('~/Maildir', factory=None)

for key in inbox.iterkeys():
    try:
        message = inbox[key]
    except email.errors.MessageParseError:
        continue                # The message is malformed. Just leave it.

    for name in list_names:
        list_id = message['list-id']
        if list_id and name in list_id:
            # Get mailbox to use
            box = boxes[name]

            # Write copy to disk before removing original.
            # If there's a crash, you might duplicate a message, but
            # that's better than losing a message completely.
            box.lock()
            box.add(message)
            box.flush()
            box.unlock()

            # Remove original message
            inbox.lock()
            inbox.discard(key)
            inbox.flush()
            inbox.unlock()
            break               # Found destination, so stop looking.
```

(下页继续)

(繼續上一頁)

```
for box in boxes.itervalues():
    box.close()
```

19.4 mimetypes --- 映射文件名到 MIME 类型

源代码: Lib/mimetypes.py

---

mimetypes 模块可以在文件名或 URL 和关联到文件扩展名的 MIME 类型之间执行转换。所提供的转换包括从文件名到 MIME 类型和从 MIME 类型到文件扩展名；后一种转换不支持编码格式。

该模块提供了一个类和一些便捷函数。这些函数是该模块通常的接口，但某些应用程序可能也会希望使用类。

下列函数提供了此模块的主要接口。如果此模块尚未被初始化，它们将会调用*init()*，如果它们依赖于*init()* 所设置的信息的话。

mimetypes.**guess_type**(*url*, *strict=True*)

根据 *url* 给出的文件名、路径或 URL 来猜测文件的类型，URL 可以为字符串或*path-like object*。

返回值是一个元组 (type, encoding) 其中 *type* 在无法猜测（后缀不存在或者未知）时为 None，或者为 'type/subtype' 形式的字符串，可以作为 MIME *content-type* 标头。

encoding 在无编码格式时为 None，或者为程序所用的编码格式 (例如 **compress** 或 **gzip**)。它可以作为 *Content-Encoding* 标头，但 **不可**作为 *Content-Transfer-Encoding* 标头。映射是表格驱动的。编码格式前缀对大小写敏感；类型前缀会先以大小写敏感方式检测再以大小写不敏感方式检测。

可选的 *strict* 参数是一个旗标，指明要将已知 MIME 类型限制在 IANA 已注册 的官方类型之内。当 *strict* 为 True 时（默认值），则仅支持 IANA 类型；当 *strict* 为 False 时，则还支持某些附加的非标准但常用的 MIME 类型。

3.8 版更變: 增加了*path-like object* 作为 url 的支持。

mimetypes.**guess_all_extensions**(*type*, *strict=True*)

根据由 *type* 给出的文件 MIME 类型猜测其扩展名。返回值是由所有可能的文件扩展名组成的字符串列表，包括开头的点号 ('.')。这些扩展名不保证能关联到任何特定的数据流，但是将会由*guess_type()* 映射到 MIME 类型 *type*。

可选的 *strict* 参数具有与*guess_type()* 函数一致的含义。

mimetypes.**guess_extension**(*type*, *strict=True*)

根据由 *type* 给出的文件 MIME 类型猜测其扩展名。返回值是一个表示文件扩展名的字符串，包括开头的点号 ('.')。该扩展名不保证能关联到任何特定的数据流，但是将会由*guess_type()* 映射到 MIME 类型 *type*。如果不能猜测出 *type* 的扩展名，则将返回 None。

可选的 *strict* 参数具有与*guess_type()* 函数一致的含义。

有一些附加函数和数据项可被用于控制此模块的行为。

mimetypes.**init**(*files=None*)

初始化内部数据结构。*files* 如果给出则必须是一个文件名序列，它应当被用于协助默认的类型映射。如果省略则要使用的文件名会从*knownfiles* 中获取；在 Windows 上，将会载入当前注册表设置。在 *files* 或*knownfiles* 中指定的每个文件名的优先级将高于在它之前的文件名。*init()* 允许被重复调用。

为 *files* 指定一个空列表将防止应用系统默认选项：将只保留来自内置列表的常用值。

如果 *files* 为 None 则内部数据结构会完全重建为其初始默认值。这是一个稳定操作并将在多次调用时产生相同的结果。

3.2 版更變: 在之前版本中，Windows 注册表设置会被忽略。

mimetypes.**read_mime_types**(*filename*)
载入在文件 *filename* 中给定的类型映射，如果文件存在的话。返回的类型映射会是一个字典，其中的键值对为文件扩展名包括开头的点号 ('.') 与 'type/subtype' 形式的字符串。如果文件 *filename* 不存在或无法被读取，则返回 None。

mimetypes.**add_type**(*type*, *ext*, *strict=True*)
添加一个从 MIME 类型 *type* 到扩展名 *ext* 的映射。当扩展名已知时，新类型将替代旧类型。当类型已知时，扩展名将被添加到已知扩展名列表。

当 *strict* 为 True 时（默认值），映射将被添加到官方 MIME 类型，否则添加到非标准类型。

mimetypes.**inited**
指明全局数据结构是否已被初始化的旗标。这会由*init()* 设为 True。

mimetypes.**knownfiles**
通常安装的类型映射文件名列表。这些文件一般被命名为 mime.types 并会由不同的包安装在不同的位置。

mimetypes.**suffix_map**
将后缀映射到其他后缀的字典。它被用来允许识别已编码的文件，其编码格式和类型是由相同的扩展名来指明的。例如，.tgz 扩展名被映射到 .tar.gz 以允许编码格式和类型被分别识别。

mimetypes.**encodings_map**
映射文件扩展名到编码格式类型的字典。

mimetypes.**types_map**
映射文件扩展名到 MIME 类型的字典。

mimetypes.**common_types**
映射文件扩展名到非标准但常见的 MIME 类型的字典。

此模块一个使用示例:

```
>>> import mimetypes
>>> mimetypes.init()
>>> mimetypes.knownfiles
['/etc/mime.types', '/etc/httpd/mime.types', ... ]
>>> mimetypes.suffix_map['.tgz']
'.tar.gz'
>>> mimetypes.encodings_map['.gz']
'gzip'
>>> mimetypes.types_map['.tgz']
'application/x-tar-gz'
```

19.4.1 MimeTypes 对象

MimeTypes 类可以被用于那些需要多个 MIME 类型数据库的应用程序；它提供了与*mimetypes* 模块所提供的类似接口。

class mimetypes.**MimeTypes**(*filenames=()*, *strict=True*)
这个类表示 MIME 类型数据库。默认情况下，它提供了对与此模块其余部分一致的数据库的访问权限。这个初始数据库是此模块所提供数据库的一个副本，并可以通过使用*read()* 或*readfp()* 方法将附加的 mime.types 样式文载入到数据库中来进行扩展。如果不需要默认数据的话这个映射字典也可以在载入附加数据之前先被清空。

可选的 *filenames* 形参可被用来让附加文件被载入到默认数据库“之上”。

suffix_map
将后缀映射到其他后缀的字典。它被用来允许识别已编码的文件，其编码格式和类型是由相同的扩展名来指明的。例如，`.tgz` 扩展名被映射到 `.tar.gz` 以允许编码格式和类型被分别识别。这是在模块中定义的全局*suffix_map* 的一个副本。

encodings_map
映射文件扩展名到编码格式类型的字典。这是在模块中定义的全局*encodings_map* 的一个副本。

types_map
包含两个字典的元组，将文件扩展名映射到 MIME 类型：第一个字典针对非标准类型而第二个字典针对标准类型。它们会由*common_types* 和*types_map* 来初始化。

types_map_inv
包含两个字典的元组，将 MIME 类型映射到文件扩展名列表：第一个字典针对非标准类型而第二个字典针对标准类型。它们会由*common_types* 和*types_map* 来初始化。

guess_extension(*type*, *strict=True*)
类似于*guess_extension()* 函数，使用存储的表作为对象的一部分。

guess_type(*url*, *strict=True*)
类似于*guess_type()* 函数，使用存储的表作为对象的一部分。

guess_all_extensions(*type*, *strict=True*)
类似于*guess_all_extensions()* 函数，使用存储的表作为对象的一部分。

read(*filename*, *strict=True*)
从名称为 *filename* 的文件载入 MIME 信息。此方法使用*readfp()* 来解析文件。

如果 *strict* 为 `True`，信息将被添加到标准类型列表，否则添加到非标准类型列表。

readfp(*fp*, *strict=True*)
从打开的文件 *fp* 载入 MIME 类型信息。文件必须具有标准 `mime.types` 文件的格式。

如果 *strict* 为 `True`，信息将被添加到标准类型列表，否则添加到非标准类型列表。

read_windows_registry(*strict=True*)
从 Windows 注册表载入 MIME 类型信息。

可用性: Windows。

如果 *strict* 为 `True`，信息将被添加到标准类型列表，否则添加到非标准类型列表。

3.2 版新加入.

19.5 base64 ——Base16、Base32、Base64、Base85 資料編碼

源代码： Lib/base64.py

這個模組提供將二進位資料編碼成可顯示 ASCII 字元以及解碼回原始資料的功能，包括了**RFC 3548**中的 Base16、Base32、Base64 等編碼方式，以及標准 Ascii85、Base85 編碼等。

RFC 3548標准編碼可以使電子郵件、URL，或是 HTTP POST 🄴容等傳輸管道安全地傳遞資料。這些編碼演算法與 **uuencode** 🄴不相同。

此模块提供了两个接口。新的接口提供了从类字节对象 到 ASCII 字节*bytes* 的编码，以及将 ASCII 的类字节对象 或字符串解码到*bytes* 的操作。此模块支持定义在 **RFC 3548** 中的所有 base-64 字母表（普通的、URL 安全的和文件系统安全的）。

旧的接口不提供从字符串的解码操作，但提供了操作文件对象 的编码和解码函数。旧接口只支持标准的 Base64 字母表，并且按照 **RFC 2045** 的规范每 76 个字符增加一个换行符。注意：如果你需要支持 **RFC 2045**，那么使用*email* 模块可能更加合适。

3.3 版更變: 新的接口提供的解码函数现在已经支持只包含 ASCII 的 Unicode 字符串。

3.4 版更變: 所有类字节对象 现在已经被所有编码和解码函数接受。添加了对 Ascii85/Base85 的支持。

新的接口提供：

base64.**b64encode**(*s*, *altchars=None*)
对*bytes-like object* *s* 进行 Base64 编码，并返回编码后的*bytes*。

可选项 *altchars* 必须是一个长 2 字节的*bytes-like object*，它指定了用于替换 + 和 / 的字符。这允许应用程序生成 URL 或文件系统安全的 Base64 字符串。默认值是 None，使用标准 Base64 字母表。

base64.**b64decode**(*s*, *altchars=None*, *validate=False*)
解码 Base64 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的*bytes*。

可选项 *altchars* 必须是一个长 2 字节的*bytes-like object*，它指定了用于替换 + 和 / 的字符。

如果 *s* 被不正确地填写，一个*binascii.Error* 错误将被抛出。

如果 *validate* 值为 False （默认情况），则在填充检查前，将丢弃既不在标准 base-64 字母表之中也不在备用字母表中的字符。如果 *validate* 为 True，这些非 base64 字符将导致*binascii.Error*。

base64.**standard_b64encode**(*s*)
编码*bytes-like object* *s*，使用标准 Base64 字母表并返回编码过的*bytes*。

base64.**standard_b64decode**(*s*)
解码*bytes-like object* 或 ASCII 字符串 *s*，使用标准 Base64 字母表并返回编码过的*bytes*。

base64.**urlsafe_b64encode**(*s*)
编码*bytes-like object* *s*，使用 URL 与文件系统安全的字母表，使用 - 以及 _ 代替标准 Base64 字母表中的 + 和 / 。返回编码过的*bytes*。结果中可能包含 =。

base64.**urlsafe_b64decode**(*s*)
解码*bytes-like object* 或 ASCII 字符串 *s*，使用 URL 与文件系统安全的字母表，使用 - 以及 _ 代替标准 Base64 字母表中的 + 和 / 。返回解码过的*bytes*

base64.**b32encode**(*s*)
用 Base32 编码*bytes-like object* *s* 并返回编码过的*bytes*

base64.**b32decode**(*s*, *casefold=False*, *map01=None*)
解码 Base32 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的*bytes*。

可选的 *casefold* 是一个指定小写字幕是否可接受为输入的标志。为了安全考虑，默认值为 False。

RFC 3548 允许将字母 0(zero) 映射为字母 O(oh)，并可以选择是否将字母 1(one) 映射为 I(eye) 或 L(el)。可选参数 *map01* 不是 None 时，它的值指定 1 的映射目标 (I 或 l)，当 *map01* 非 None 时，0 总是被映射为 O。为了安全考虑，默认值被设为 None，如果是这样，0 和 1 不允许被作为输入。

如果 *s* 被错误地填写或输入中存在字母表之外的字符，将抛出*binascii.Error*。

base64.**b16encode**(*s*)
用 Base16 编码*bytes-like object* *s* 并返回编码过的*bytes*

base64.**b16decode**(*s*, *casefold=False*)
解码 Base16 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的*bytes*。

可选的 *casefold* 是一个指定小写字幕是否可接受为输入的标志。为了安全考虑，默认值为 False。

如果 *s* 被错误地填写或输入中存在字母表之外的字符，将抛出*binascii.Error*。

base64.**a85encode**(*b*, *, *foldspaces=False*, *wrapcol=0*, *pad=False*, *adobe=False*)
用 Ascii85 编码*bytes-like object* *s* 并返回编码过的`bytes`

foldspaces 是一个可选的标志，使用特殊的短序列'y' 代替'btoa' 提供的 4 个连续空格 (ASCII 0x20)。这个特性不被" 标准" Ascii85 编码支持。

wrapcol 控制了输出是否包含换行符 (`b'\n'`). 如果该值非零, 则每一行只有该值所限制的字符长度.

pad 控制在编码之前输入是否填充为 4 的倍数。请注意，`btoa` 实现总是填充。

adobe 控制编码后的字节序列是否要加上 <~ 和 ~>，这是 Adobe 实现所使用的。

3.4 版新加入.

base64.**a85decode**(*b*, *, *foldspaces=False*, *adobe=False*, *ignorechars=b' \t\n\r\v'*)
解码 Ascii85 编码过的*bytes-like object* 或 ASCII 字符串 *s* 并返回解码过的`bytes`。

foldspaces 旗标指明是否应接受'y' 短序列作为 4 个连续空格 (ASCII 0x20) 的快捷方式。此特性不被" 标准" Ascii85 编码格式所支持。

adobe 控制输入序列是否为 Adobe Ascii85 格式 (即附加 <~ 和 ~>)。

ignorechars 应当是一个*bytes-like object* 或 ASCII 字符串，其中包含要从输入中忽略的字符。这应当只包含空白字符，并且默认包含 ASCII 中所有的空白字符。

3.4 版新加入.

base64.**b85encode**(*b*, *pad=False*)
用 base85（如 git 风格的二进制 diff 数据所用格式）编码*bytes-like object* *b* 并返回编码后的`bytes`。

如果 *pad* 为真值，输入将以 `b'\0'` 填充以使其编码前长度为 4 字节的倍数。

3.4 版新加入.

base64.**b85decode**(*b*)
解码 base85 编码过的*bytes-like object* 或 ASCII 字符串 *b* 并返回解码过的`bytes`。如有必要，填充会被隐式地移除。

3.4 版新加入.

旧式接口:

base64.**decode**(*input*, *output*)
解码二进制 *input* 文件的内容并将结果二进制数据写入 *output* 文件。*input* 和 *output* 必须为文件对象. *input* 将被读取直至 `input.readline()` 返回空字节串对象。

base64.**decodebytes**(*s*)
解码*bytes-like object* *s*，该对象必须包含一行或多行 base64 编码的数据，并返回已解码的`bytes`。

3.1 版新加入.

base64.**encode**(*input*, *output*)
编码二进制 *input* 文件的内容并将经 base64 编码的数据写入 *output* 文件。*input* 和 *output* 必须为文件对象。*input* 将被读取直到 `input.read()` 返回空字节串对象。`encode()` 会在每输出 76 个字节之后插入一个换行符 (`b'\n'`)，并会确保输出总是以换行符来结束，如 **RFC 2045** (MIME) 所规定的那样。

base64.**encodebytes**(*s*)
编码*bytes-like object* *s*，其中可以包含任意二进制数据，并返回包含经 base64 编码数据的`bytes`，每输出 76 个字节之后将带一个换行符 (`b'\n'`)，并会确保在末尾也有一个换行符，如 **RFC 2045** (MIME) 所规定的那样。

3.1 版新加入.

此模块的一个使用示例:

```
>>> import base64
>>> encoded = base64.b64encode(b'data to be encoded')
>>> encoded
b'ZGF0YSB0byBiZSBlbmNvZGVk'
>>> data = base64.b64decode(encoded)
>>> data
b'data to be encoded'
```

也参考:

模块*binascii* 支持模块，包含 ASCII 到二进制和二进制到 ASCII 转换。

RFC 1521 - MIME (Multipurpose Internet Mail Extensions) 第一部分：规定并描述因特网消息体的格式的机制。
第 5.2 节，“Base64 内容转换编码格式”提供了 base64 编码格式的定义。

19.6 binhex --- 对 binhex4 文件进行编码和解码

源代码: Lib/binhex.py

3.9 版後已🄴用.

此模块以 binhe4 格式对文件进行编码和解码，该格式允许 Macintosh 文件以 ASCII 格式表示。仅处理数据分支。

binhex 模块定义了以下功能:

binhex.**binhex**(*input*, *output*)
将带有文件名 *输入* 的二进制文件转换为 binhex 文件 *输出*。输出参数可以是文件名或类文件对象（`write()` 和 `close()` 方法的任何对象）。

binhex.**hexbin**(*input*, *output*)
解码 binhex 文件输入。*输入* 可以是支持 `read()` 和 `close()` 方法的文件名或类文件对象。生成的文件将写入名为 *output* 的文件，除非参数为 `None`，在这种情况下，从 binhex 文件中读取输出文件名。

还定义了以下异常:

exception binhex.**Error**
当无法使用 binhex 格式编码某些内容时（例如，文件名太长而无法放入文件名字段中），或者输入未正确编码的 binhex 数据时，会引发异常。

也参考:

模块*binascii* 支持模块，包含 ASCII 到二进制和二进制到 ASCII 转换。

19.6.1 🄵解

还有一个替代的、功能更强大的编码器和解码器接口，详细信息请参见源代码。

如果您在非 Macintosh 平台上编码或解码文本文件，它们仍将使用旧的 Macintosh 换行符约定（回车符作为行尾）。

19.7 binascii --- 二进制和 ASCII 码互转

binascii 模块包含很多在二进制和二进制表示的各种 ASCII 码之间转换的方法。通常情况不会直接使用这些函数，而是使用像*uu*，*base64*，或*binhex* 这样的封装模块。为了执行效率高，*binascii* 模块含有许多用 C 写的低级函数，这些底层函数被一些高级模块所使用。

備𝔽: a2b_* 函数接受只含有 ASCII 码的 Unicode 字符串。其他函数只接受*字节类对象*（例如*bytes*，*bytearray* 和其他支持缓冲区协议的对象）。

3.3 版更變: ASCII-only unicode strings are now accepted by the a2b_* functions.

binascii 模块定义了以下函数：

binascii.**a2b_uu**(*string*)
将单行 uu 编码数据转换成二进制数据并返回。uu 编码每行的数据通常包含 45 个（二进制）字节，最后一行除外。每行数据后面可能跟有空格。

binascii.**b2a_uu**(*data*, *, *backtick=False*)
将二进制数据转换为 ASCII 编码字符，返回值是转换后的行数据，包括换行符。*data* 的长度最多为 45。如果 *backtick* 为 ture，则零由 `'`' 而不是空格表示。

3.7 版更變: 增加 *backtick* 形参。

binascii.**a2b_base64**(*string*)
将 base64 数据块转换成二进制并以二进制数据形式返回。一次可以传递多行数据。

binascii.**b2a_base64**(*data*, *, *newline=True*)
将二进制数据转换为一行用 base64 编码的 ASCII 字符串。返回值是转换后的行数据，如果 *newline* 为 true，则返回值包括换行符。该函数的输出符合：rfc：*3548*。

3.6 版更變: 增加 *newline* 形参。

binascii.**a2b_qp**(*data*, *header=False*)
将一个引号可打印的数据块转换成二进制数据并返回。一次可以转换多行。如果可选参数 *header* 存在且为 true，则数据中的下划线将被解码成空格。

binascii.**b2a_qp**(*data*, *quotetabs=False*, *istext=True*, *header=False*)
将二进制数据转换为一行或多行带引号可打印编码的 ASCII 字符串。返回值是转换后的行数据。如果可选参数 *quotetabs* 存在且为真值，则对所有制表符和空格进行编码。如果可选参数 *istext* 存在且为真值，则不对新行进行编码，但将对尾随空格进行编码。如果可选参数 *header* 存在且为 true，则空格将被编码为下划线 **RFC 1522**。如果可选参数 *header* 存在且为假值，则也会对换行符进行编码; 不进行换行转换编码可能会破坏二进制数据流。

binascii.**a2b_hqx**(*string*)
将 binhex4 格式的 ASCII 数据不进行 RLE 解压缩直接转换为二进制数据。该字符串应包含完整数量的二进制字节，或者（在 binhex4 数据最后部分）剩余位为零。

3.9 版後已𝔽用.

binascii.**rledecode_hqx**(*data*)
根据 binhex4 标准对数据执行 RLE 解压缩。该算法在一个字节的数据后使用 0x90 作为重复指示符，然后计数。计数 0 指定字节值 0x90 。该例程返回解压缩的数据，输入数据以孤立的重复指示符结束的情况下，将引发*Incomplete* 异常。

3.2 版更變: 仅接受 bytestring 或 bytearray 对象作为输入。

3.9 版後已𝔽用.

binascii.**rlecode_hqx**(*data*)
在 *data* 上执行 binhex4 游程编码压缩并返回结果。

3.9 版後已𝔽用.

binascii.**b2a_hqx**(*data*)
执行 hexbin4 类型二进制到 ASCII 码的转换并返回结果字符串。输入数据应经过 RLE 编码，且数据长度可被 3 整除（除了最后一个片段）。

3.9 版後已𝔽用.

binascii.**crc_hqx**(*data*, *value*)
以 *value* 作为初始 CRC 计算 *data* 的 16 位 CRC 值，返回其结果。这里使用 CRC-CCITT 生成多项式 $x^{16} + x^{12} + x^5 + 1$，通常表示为 0x1021。该 CRC 被用于 binhex4 格式。

binascii.**crc32**(*data*[, *value*])
Compute CRC-32, the unsigned 32-bit checksum of *data*, starting with an initial CRC of *value*. The default initial CRC is zero. The algorithm is consistent with the ZIP file checksum. Since the algorithm is designed for use as a checksum algorithm, it is not suitable for use as a general hash algorithm. Use as follows:

```
print(binascii.crc32(b"hello world"))
# Or, in two pieces:
crc = binascii.crc32(b"hello")
crc = binascii.crc32(b" world", crc)
print('crc32 = {:#010x}'.format(crc))
```

3.0 版更變: The result is always unsigned. To generate the same numeric value when using Python 2 or earlier, use `crc32(data) & 0xffffffff`.

binascii.**b2a_hex**(*data*[, *sep*[, *bytes_per_sep=1*]])
binascii.**hexlify**(*data*[, *sep*[, *bytes_per_sep=1*]])
返回二进制数据 *data* 的十六进制表示形式。*data* 的每个字节都被转换为相应的 2 位十六进制表示形式。因此返回的字节对象的长度是 *data* 的两倍。

使用：*bytes.hex()* 方法也可以方便地实现相似的功能（但仅返回文本字符串）。

如果指定了 *sep*，它必须为单字符 str 或 bytes 对象。它将被插入每个 *bytes_per_sep* 输入字节之后。分隔符位置默认从输出的右端开始计数，如果你希望从左端开始计数，请提供一个负的 *bytes_per_sep* 值。

```
>>> import binascii
>>> binascii.b2a_hex(b'\xb9\x01\xef')
b'b901ef'
>>> binascii.hexlify(b'\xb9\x01\xef', '-')
b'b9-01-ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b'_', 2)
b'b9_01ef'
>>> binascii.b2a_hex(b'\xb9\x01\xef', b' ', -2)
b'b901 ef'
```

3.8 版更變: 添加了 *sep* 和 *bytes_per_sep* 形参。

binascii.**a2b_hex**(*hexstr*)
binascii.**unhexlify**(*hexstr*)
返回由十六进制字符串 *hexstr* 表示的二进制数据。此函数功能与*b2a_hex()* 相反。*hexstr* 必须包含偶数个十六进制数字（可以是大写或小写），否则会引发*Error* 异常。

使用：*bytes.fromhex()* 类方法也实现相似的功能（仅接受文本字符串参数，不限制其中的空白字符）。

exception binascii.**Error**
通常是因为编程错误引发的异常。

exception binascii.**Incomplete**
数据不完整引发的异常。通常不是编程错误导致的，可以通过读取更多的数据并再次尝试来处理该异常。

也参考:

模块*base64* 支持在 16，32，64，85 进制中进行符合 RFC 协议的 base64 样式编码。

模块*binhex* 支持在 Macintosh 上使用的 binhex 格式。

模块*uu* 支持在 Unix 上使用的 UU 编码。

模块*quopri* 支持在 MIME 版本电子邮件中使用引号可打印编码。

19.8 quopri --- 编码与解码经过 MIME 转码的可打印数据

源代码: Lib/quopri.py

此模块会执行转换后可打印的传输编码与解码，具体定义见 **RFC 1521**: "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies"。转换后可打印的编码格式被设计用于只包含相对较少的不可打印字符的数据；如果存在大量这样的字符，通过*base64*模块所提供的 base64 编码方案会更为紧凑，例如当发送图片文件时。

quopri.**decode**(*input*, *output*, *header=False*)
解码 *input* 文件的内容并将已解码二进制数据结果写入 *output* 文件。*input* 和 *output* 必须为二进制文件对象。如果提供了可选参数 *header* 且为真值，下划线将被解码为空格。此函数可用于解码 "Q" 编码的头数据，具体描述见 **RFC 1522**: "MIME (Multipurpose Internet Mail Extensions) Part Two: Message Header Extensions for Non-ASCII Text"。

quopri.**encode**(*input*, *output*, *quotetabs*, *header=False*)
编码 *input* 文件的内容并将转换后可打印的数据结果写入 *output* 文件。*input* 和 *output* 必须为二进制文件对象. *quotetabs* 是一个非可选的旗标，它控制是否要编码内嵌的空格与制表符；当为真值时将编码此类内嵌空白符，当为假值时则保持原样不进行编码。请注意出现在行尾的空格与制表符总是会被编码，具体描述见 **RFC 1521**。*header* 旗标控制空格符是否要编码为下划线，具体描述见 **RFC 1522**。

quopri.**decodestring**(*s*, *header=False*)
类似*decode()*，区别在于它接受一个源*bytes*并返回对应的已解码*bytes*。

quopri.**encodestring**(*s*, *quotetabs=False*, *header=False*)
类型*encode()*，区别在于它接受一个源*bytes*并返回对应的已编码*bytes*。在默认情况下，它会发送 False 值给*encode()*函数的 *quotetabs* 形参。

也参考:

模块*base64* 编码与解码 MIME base64 数据

CHAPTER 20

结构化标记处理工具

Python 支持各种模块，以处理各种形式的结构化数据标记。这包括使用标准通用标记语言（SGML）和超文本标记语言（HTML）的模块，以及使用可扩展标记语言（XML）的几个接口。

20.1 html --- 超文本标记语言支持

源码： Lib/html/__init__.py

该模块定义了操作 HTML 的工具。

html.**escape**(*s*, *quote=True*)
将字符串 *s* 中的字符 “&“ 、< 和 > 转换为安全的 HTML 序列。如果需要在 HTML 中显示可能包含此类字符的文本，请使用此选项。如果可选的标志 *quote* 为真值，则字符 (") 和 (') 也被转换；这有助于包含在由引号分隔的 HTML 属性中，如 <a href="...">。

3.2 版新加入.

html.**unescape**(*s*)
将字符串 *s* 中的所有命名和数字字符引用 (例如 >, >, >) 转换为相应的 Unicode 字符。此函数使用 HTML 5 标准为有效和无效字符引用定义的规则，以及*HTML 5 命名字符引用列表*。

3.4 版新加入.

---

html 包中的子模块是：

- *html.parser* ——具有宽松解析模式的 HTML / XHTML 解析器
- *html.entities* -- HTML 实体定义

20.2 html.parser --- 简单的 HTML 和 XHTML 解析器

源代码： Lib/html/parser.py

这个模块定义了一个*HTMLParser* 类，为 HTML（超文本标记语言）和 XHTML 文本文件解析提供基础。

class html.parser.**HTMLParser**(*, *convert_charrefs=True*)

创建一个能解析无效标记的解析器实例。

如果 *convert_charrefs* 为 True (默认值)，则所有字符引用 (script/style 元素中的除外) 都会自动转换为相应的 Unicode 字符。

一个*HTMLParser* 类的实例用来接受 HTML 数据，并在标记开始、标记结束、文本、注释和其他元素标记出现的时候调用对应的方法。要实现具体的行为，请使用*HTMLParser* 的子类并重载其方法。

这个解析器不检查结束标记是否与开始标记匹配，也不会因外层元素完毕而隐式关闭了的元素引发结束标记处理。

3.4 版更變: *convert_charrefs* 关键字参数被添加。

3.5 版更變: *convert_charrefs* 参数的默认值现在为 True。

20.2.1 HTML 解析器的示例程序

下面是简单的 HTML 解析器的一个基本示例，使用*HTMLParser* 类，当遇到开始标记、结束标记以及数据的时候将内容打印出来。

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Encountered a start tag:", tag)

    def handle_endtag(self, tag):
        print("Encountered an end tag :", tag)

    def handle_data(self, data):
        print("Encountered some data  :", data)

parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

输出是:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

20.2.2 HTMLParser 方法

HTMLParser 实例有下列方法：

HTMLParser.**feed**(*data*)
填充一些文本到解析器中。如果包含完整的元素，则被处理；如果数据不完整，将被缓冲直到更多的数据被填充，或者*close()* 被调用。*data* 必须为*str* 类型。

HTMLParser.**close**()
如同后面跟着一个文件结束标记一样，强制处理所有缓冲数据。这个方法能被派生类重新定义，用于在输入的末尾定义附加处理，但是重定义的版本应当始终调用基类*HTMLParser* 的*close()* 方法。

HTMLParser.**reset**()
重置实例。丢失所有未处理的数据。在实例化阶段被隐式调用。

HTMLParser.**getpos**()
返回当前行号和偏移值。

HTMLParser.**get_starttag_text**()
返回最近打开的开始标记中的文本。结构化处理时通常应该不需要这个，但在处理“已部署”的 HTML 或是在以最小改变来重新生成输入时可能会有用处（例如可以保留属性间的空格等）。

下列方法将在遇到数据或者标记元素的时候被调用。他们需要在子类中重载。基类的实现中没有任何实际操作（除了*handle_startendtag()* ）：

HTMLParser.**handle_starttag**(*tag*, *attrs*)
This method is called to handle the start tag of an element (e.g. `<div id="main">`).

tag 参数是小写的标记名。*attrs* 参数是一个 `(name, value)` 形式的列表，包含了所有在标记的 <> 括号中找到的属性。*name* 转换为小写，*value* 的引号被去除，字符和实体引用都会被替换。

实例中，对于标签 `<A HREF="https://www.cwi.nl/">`，这个方法将以下列形式被调用 `handle_starttag('a', [('href', 'https://www.cwi.nl/')])` 。

html.entities 中的所有实体引用，会被替换为属性值。

HTMLParser.**handle_endtag**(*tag*)
此方法被用来处理元素的结束标记（例如：`</div>` ）。

tag 参数是小写的标签名。

HTMLParser.**handle_startendtag**(*tag*, *attrs*)
类似于*handle_starttag()*，只是在解析器遇到 XHTML 样式的空标记时被调用（`<img ... />`）。这个方法能被需要这种特殊词法信息的子类重载；默认实现仅简单调用*handle_starttag()* 和*handle_endtag()* 。

HTMLParser.**handle_data**(*data*)
这个方法被用来处理任意数据（例如：文本节点和 `<script>...</script>` 以及 `<style>...</style>` 中的内容）。

HTMLParser.**handle_entityref**(*name*)
这个方法被用于处理 `&name;` 形式的命名字符引用（例如 `>`），其中 *name* 是通用的实体引用（例如：`'gt'`）。如果 *convert_charrefs* 为 `True`，该方法永远不会被调用。

HTMLParser.**handle_charref**(*name*)
这个方法被用来处理 `&#NNN;` 和 `&#xNNN;` 形式的十进制和十六进制字符引用。例如，`>` 等效的十进制形式为 `>` ，而十六进制形式为 `>` ；在这种情况下，方法将收到 `'62'` 或 `'x3E'` 。如果 *convert_charrefs* 为 `True` ，则该方法永远不会被调用。

HTMLParser.**handle_comment**(*data*)
这个方法在遇到注释的时候被调用（例如：`<!--comment-->` ）。

例如，`<!-- comment -->` 这个注释会用 `' comment '` 作为参数调用此方法。

Internet Explorer 条件注释（condcoms）的内容也被发送到这个方法，因此，对于 `<!--[if IE 9]>IE9-specific content<![endif]-->`，这个方法将接收到 `'[if IE 9]>IE9-specific content<![endif]'`。

`HTMLParser.`**`handle_decl`**`(`*decl*`)`
这个方法用来处理 HTML doctype 申明（例如 `<!DOCTYPE html>`）。

decl 形参为 `<!...>` 标记中的所有内容（例如：`'DOCTYPE html'`）。

`HTMLParser.`**`handle_pi`**`(`*data*`)`
此方法在遇到处理指令的时候被调用。*data* 形参将包含整个处理指令。例如，对于处理指令 `<?proc color='red'>`，这个方法将以 `handle_pi("proc color='red'")` 形式被调用。它旨在被派生类重载；基类实现中无任何实际操作。

備🄵: *`HTMLParser`* 类使用 SGML 语法规则处理指令。使用 `'?'` 结尾的 XHTML 处理指令将导致 `'?'` 包含在 *data* 中。

`HTMLParser.`**`unknown_decl`**`(`*data*`)`
当解析器读到无法识别的声明时，此方法被调用。

data 形参为 `<![...]>` 标记中的所有内容。某些时候对派生类的重载很有用。基类实现中无任何实际操作。

20.2.3 示例

下面的类实现了一个解析器，用于更多示例的演示:

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)

    def handle_endtag(self, tag):
        print("End tag  :", tag)

    def handle_data(self, data):
        print("Data     :", data)

    def handle_comment(self, data):
        print("Comment  :", data)

    def handle_entityref(self, name):
        c = chr(name2codepoint[name])
        print("Named ent:", c)

    def handle_charref(self, name):
        if name.startswith('x'):
            c = chr(int(name[1:], 16))
        else:
            c = chr(int(name))
        print("Num ent  :", c)
```

(下页继续)

(繼續上一頁)

```
    def handle_decl(self, data):
        print("Decl     :", data)

parser = MyHTMLParser()
```

解析一个文档类型声明:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/
↪html4/strict.dtd"
```

解析一个具有一些属性和标题的元素:

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

`script` 和 `style` 元素中的内容原样返回，无需进一步解析:

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style

>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

解析注释:

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

解析命名或数字形式的字符引用，并把他们转换到正确的字符（注意：这 3 种转义都是 `'>'`）：

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

填充不完整的块给*feed()* 执行，*handle_data()* 可能会多次调用（除非 *convert_charrefs* 被设置为 `True`）：

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
...
Start tag: span
Data     : buff
Data     : ered
Data     : text
End tag  : span
```

解析无效的 HTML (例如：未引用的属性）也能正常运行:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```

20.3 html.entities --- HTML 一般实体的定义

源码： Lib/html/entities.py

该模块定义了四个词典，*html5*、*name2codepoint*、*codepoint2name*、以及*entitydefs*。

html.entities.**html5**
: 将 HTML5 命名字符引用[1] 映射到等效的 Unicode 字符的字典，例如 `html5['gt;'] == '>'`。请注意，尾随的分号包含在名称中（例如 `'gt;'`），但是即使没有分号，一些名称也会被标准接受，在这种情况下，名称出现时带有和不带有 `';'`。另见*html.unescape()*。

  3.3 版新加入.

html.entities.**entitydefs**
: 将 XHTML 1.0 实体定义映射到 ISO Latin-1 中的替换文本的字典。

html.entities.**name2codepoint**
: 将 HTML 实体名称映射到 Unicode 代码点的字典。

html.entities.**codepoint2name**
: 将 Unicode 代码点映射到 HTML 实体名称的字典。

🄵解

20.4 XML 處理模組

源码： Lib/xml/

用于处理 XML 的 Python 接口分组在 `xml` 包中。

[1] See https://html.spec.whatwg.org/multipage/syntax.html#named-character-references

警告： XML 模块对于错误或恶意构造的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞* 和*defusedxml 包* 部分。

值得注意的是*xml* 包中的模块要求至少有一个 SAX 兼容的 XML 解析器可用。Expat 解析器包含在 Python 中，因此*xml.parsers.expat* 模块将始终可用。

xml.dom 和*xml.sax* 包的文档是 DOM 和 SAX 接口的 Python 绑定的定义。

XML 处理子模块包括:

- *xml.etree.ElementTree*：ElementTree API，一个简单而轻量级的 XML 处理器
- *xml.dom*：DOM API 定义
- *xml.dom.minidom*：最小的 DOM 实现
- *xml.dom.pulldom*：支持构建部分 DOM 树
- *xml.sax*：SAX2 基类和便利函数
- *xml.parsers.expat*：Expat 解析器绑定

20.4.1 XML 漏洞

XML 处理模块对于恶意构造的数据是不安全的。攻击者可能滥用 XML 功能来执行拒绝服务攻击、访问本地文件、生成与其它计算机的网络连接或绕过防火墙。

下表概述了已知的攻击以及各种模块是否容易受到攻击。

种类	sax	etree	minidom	pulldom	xmlrpc
billion laughs	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
quadratic blowup	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)	**Vulnerable** (1)
external entity expansion	Safe (5)	Safe (2)	Safe (3)	Safe (5)	安全 (4)
DTD retrieval	Safe (5)	安全	安全	Safe (5)	安全
decompression bomb	安全	安全	安全	安全	**易受攻击**

1. Expat 2.4.1 and newer is not vulnerable to the ”billion laughs” and ”quadratic blowup” vulnerabilities. Items still listed as vulnerable due to potential reliance on system-provided libraries. Check `pyexpat.EXPAT_VERSION`.
2. *xml.etree.ElementTree* 不会扩展外部实体并在实体发生时引发 `ParserError`。
3. *xml.dom.minidom* 不会扩展外部实体，只是简单地返回未扩展的实体。
4. `xmlrpclib` 不扩展外部实体并省略它们。
5. 从 Python 3.7.1 开始，默认情况下不再处理外部通用实体。

billion laughs / exponential entity expansion （狂笑/递归实体扩展） Billion Laughs 攻击 -- 也称为递归实体扩展 -- 使用多级嵌套实体。每个实体多次引用另一个实体，最终实体定义包含一个小字符串。指数级扩展导致几千 GB 的文本，并消耗大量内存和 CPU 时间。

quadratic blowup entity expansion（二次爆炸实体扩展） 二次爆炸攻击类似于 Billion Laughs 攻击，它也滥用实体扩展。它不是嵌套实体，而是一遍又一遍地重复一个具有几千个字符的大型实体。攻击不如递归情况有效，但它避免触发禁止深度嵌套实体的解析器对策。

external entity expansion 实体声明可以包含的不仅仅是替换文本。它们还可以指向外部资源或本地文件。XML 解析器访问资源并将内容嵌入到 XML 文档中。

DTD retrieval Python 的一些 XML 库*xml.dom.pulldom* 从远程或本地位置检索文档类型定义。该功能与外部实体扩展问题具有相似的含义。

decompression bomb Decompression bombs（解压炸弹，又名 ZIP bomb）适用于所有可以解析压缩 XML 流（例如 gzip 压缩的 HTTP 流或 LZMA 压缩的文件）的 XML 库。对于攻击者来说，它可以将传输的数据量减少三个量级或更多。

PyPI 上 defusedxml 的文档包含有关所有已知攻击向量的更多信息以及示例和参考。

20.4.2 `defusedxml` 包

defusedxml 是一个纯 Python 软件包，它修改了所有标准库 XML 解析器的子类，可以防止任何潜在的恶意操作。对于解析不受信任的 XML 数据的任何服务器代码，建议使用此程序包。该软件包还提供了有关更多 XML 漏洞（如 XPath 注入）的示例漏洞和扩展文档。

20.5 `xml.etree.ElementTree` --- ElementTree XML API

源代码： Lib/xml/etree/ElementTree.py

---

xml.etree.ElementTree 模块实现了一个简单高效的 API，用于解析和创建 XML 数据。

3.3 版更變: 此模块将在可能的情况下使用快速实现。

3.3 版後已🄵用: `xml.etree.cElementTree` 模块已被弃用。

> **警告：** *xml.etree.ElementTree* 模块对于恶意构建的数据是不安全的。如果需要解析不可信或未经身份验证的数据，请参见*XML 漏洞* 。

20.5.1 教程

这是一个使用*xml.etree.ElementTree* （简称 `ET` ）的简短教程。目标是演示模块的一些构建块和基本概念。

XML 树和元素

XML 是一种继承性的分层数据格式，最自然的表示方法是使用树。为此，`ET` 有两个类 -- *ElementTree* 将整个 XML 文档表示为一个树，*Element* 表示该树中的单个节点。与整个文档的交互（读写文件）通常在*ElementTree* 级别完成。与单个 XML 元素及其子元素的交互是在*Element* 级别完成的。

解析 XML

我们将使用以下 XML 文档作为本节的示例数据：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank>1</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank>4</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank>68</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

可以通过从文件中读取来导入此数据：

```
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```

或直接从字符串中解析：

```
root = ET.fromstring(country_data_as_string)
```

fromstring() 将 XML 从字符串直接解析为*Element*，该元素是已解析树的根元素。其他解析函数可能会创建一个*ElementTree*。确切信息请查阅文档。

作为*Element*，root 具有标签和属性字典：

```
>>> root.tag
'data'
>>> root.attrib
{}
```

还有可以迭代的子节点：

```
>>> for child in root:
...     print(child.tag, child.attrib)
...
country {'name': 'Liechtenstein'}
country {'name': 'Singapore'}
country {'name': 'Panama'}
```

子级是可以嵌套的，我们可以通过索引访问特定的子级节点：

```
>>> root[0][1].text
'2008'
```

備𝔽: 并非 XML 输入的所有元素都将作为解析树的元素结束。目前，此模块跳过输入中的任何 XML 注释、处理指令和文档类型声明。然而，使用这个模块的 API 而不是从 XML 文本解析构建的树可以包含注释和处理指令，生成 XML 输出时同样包含这些注释和处理指令。可以通过将自定义*TreeBuilder* 实例传递给*XMLParser* 构造器来访问文档类型声明。

用于非阻塞解析的拉取 API

此模块所提供了大多数解析函数都要求在返回任何结果之前一次性读取整个文档。可以使用*XMLParser* 并以增量方式添加数据，但这是在回调目标上调用方法的推送式 API。有时用户真正想要的是能够以增量方式解析 XML 而无需阻塞操作，同时享受完整的已构造*Element* 对象。

针对此需求的最强大工具是*XMLPullParser*。它不要求通过阻塞式读取来获得 XML 数据，而是通过执行*XMLPullParser.feed()* 调用来增量式地添加数据。要获得已解析的 XML 元素，应调用*XMLPullParser.read_events()*。下面是一个示例:

```
>>> parser = ET.XMLPullParser(['start', 'end'])
>>> parser.feed('<mytag>sometext')
>>> list(parser.read_events())
[('start', <Element 'mytag' at 0x7fa66db2be58>)]
>>> parser.feed(' more text</mytag>')
>>> for event, elem in parser.read_events():
...     print(event)
...     print(elem.tag, 'text=', elem.text)
...
end
```

常见的用例是针对以非阻塞方式进行的应用程序，其中 XML 是从套接字接收或从某些存储设备增量式读取的。在这些用例中，阻塞式读取是不可接受的。

因为其非常灵活，*XMLPullParser* 在更简单的用例中使用起来可能并不方便。如果你不介意你的应用程序在读取 XML 数据时造成阻塞但仍希望具有增量解析能力，可以考虑*iterparse()*。它在你读取大型 XML 文档并且不希望将它完全放去内存时会很适用。

查找感兴趣的元素

Element 有一些很有效的方法，可帮助递归遍历其下的所有子树（包括子级，子级的子级，等等）。例如*Element.iter()*:

```
>>> for neighbor in root.iter('neighbor'):
...     print(neighbor.attrib)
...
{'name': 'Austria', 'direction': 'E'}
{'name': 'Switzerland', 'direction': 'W'}
{'name': 'Malaysia', 'direction': 'N'}
{'name': 'Costa Rica', 'direction': 'W'}
{'name': 'Colombia', 'direction': 'E'}
```

Element.findall() 仅查找当前元素的直接子元素中带有指定标签的元素。*Element.find()* 找带有特定标签的 第一个子级，然后可以用*Element.text* 访问元素的文本内容。*Element.get* 访问元素的属性:

```
>>> for country in root.findall('country'):
...     rank = country.find('rank').text
...     name = country.get('name')
...     print(name, rank)
...
Liechtenstein 1
Singapore 4
Panama 68
```

通过使用*XPath*，可以更精确地指定要查找的元素。

修改 XML 文件

ElementTree 提供了一种构建 XML 文档并将其写入文件的简单方法。调用*ElementTree.write()* 方法就可以实现。

创建后可以直接操作*Element* 对象。例如：使用*Element.text* 修改文本字段，使用*Element.set()* 方法添加和修改属性，以及使用*Element.append()* 添加新的子元素。

假设我们要为每个国家/地区的中添加一个排名，并在排名元素中添加一个 `updated` 属性：

```
>>> for rank in root.iter('rank'):
...     new_rank = int(rank.text) + 1
...     rank.text = str(new_rank)
...     rank.set('updated', 'yes')
...
>>> tree.write('output.xml')
```

生成的 XML 现在看起来像这样：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
    <country name="Panama">
        <rank updated="yes">69</rank>
        <year>2011</year>
        <gdppc>13600</gdppc>
        <neighbor name="Costa Rica" direction="W"/>
        <neighbor name="Colombia" direction="E"/>
    </country>
</data>
```

可以使用*Element.remove()* 删除元素。假设我们要删除排名高于 50 的所有国家/地区：

```
>>> for country in root.findall('country'):
...     # using root.findall() to avoid removal during traversal
...     rank = int(country.find('rank').text)
...     if rank > 50:
...         root.remove(country)
...
>>> tree.write('output.xml')
```

请注意在迭代时进行并发修改可能会导致问题，就像在迭代并修改 Python 列表或字典时那样。因此，这个示例先通过 `root.findall()` 收集了所有匹配的元素，在此之后再对匹配项列表进行迭代。

生成的 XML 现在看起来像这样：

```
<?xml version="1.0"?>
<data>
    <country name="Liechtenstein">
        <rank updated="yes">2</rank>
        <year>2008</year>
        <gdppc>141100</gdppc>
        <neighbor name="Austria" direction="E"/>
        <neighbor name="Switzerland" direction="W"/>
    </country>
    <country name="Singapore">
        <rank updated="yes">5</rank>
        <year>2011</year>
        <gdppc>59900</gdppc>
        <neighbor name="Malaysia" direction="N"/>
    </country>
</data>
```

构建 XML 文档

SubElement() 函数还提供了一种便捷方法来为给定元素创建新的子元素：

```
>>> a = ET.Element('a')
>>> b = ET.SubElement(a, 'b')
>>> c = ET.SubElement(a, 'c')
>>> d = ET.SubElement(c, 'd')
>>> ET.dump(a)
<a><b /><c><d /></c></a>
```

解析带有命名空间的 XML

如果 XML 输入带有 命名空间，则具有前缀的 `prefix:sometag` 形式的标记和属性将被扩展为 `{uri}sometag`，其中 *prefix* 会被完整 *URI* 所替换。并且，如果存在 默认命名空间，则完整 URI 会被添加到所有未加前缀的标记之前。

下面的 XML 示例包含两个命名空间，一个具有前缀”fictional” 而另一个则作为默认命名空间：

```
<?xml version="1.0"?>
<actors xmlns:fictional="http://characters.example.com"
        xmlns="http://people.example.com">
    <actor>
        <name>John Cleese</name>
```

(下页继续)

(繼續上一頁)

```
        <fictional:character>Lancelot</fictional:character>
        <fictional:character>Archie Leach</fictional:character>
    </actor>
    <actor>
        <name>Eric Idle</name>
        <fictional:character>Sir Robin</fictional:character>
        <fictional:character>Gunther</fictional:character>
        <fictional:character>Commander Clement</fictional:character>
    </actor>
</actors>
```

搜索和探查这个 XML 示例的一种方式是手动为*find()* 或*findall()* 的 xpath 中的每个标记或属性添加 URI:

```
root = fromstring(xml_text)
for actor in root.findall('{http://people.example.com}actor'):
    name = actor.find('{http://people.example.com}name')
    print(name.text)
    for char in actor.findall('{http://characters.example.com}character'):
        print(' |-->', char.text)
```

一种更好的方式是搜索带命名空间的 XML 示例创建一个字典来存放你自己的前缀并在搜索函数中使用它们:

```
ns = {'real_person': 'http://people.example.com',
      'role': 'http://characters.example.com'}

for actor in root.findall('real_person:actor', ns):
    name = actor.find('real_person:name', ns)
    print(name.text)
    for char in actor.findall('role:character', ns):
        print(' |-->', char.text)
```

这两种方式都会输出:

```
John Cleese
 |--> Lancelot
 |--> Archie Leach
Eric Idle
 |--> Sir Robin
 |--> Gunther
 |--> Commander Clement
```

20.5.2 XPath 支持

此模块提供了对 XPath 表达式 的有限支持用于在树中定位元素。其目标是支持一个简化语法的较小子集；完整的 XPath 引擎超出了此模块的适用范围。

示例

下面是一个演示此模块的部分 XPath 功能的例子。我们将使用来自*解析 XML* 小节的 `countrydata` XML 文档:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(countrydata)

# Top-level elements
root.findall(".")

# All 'neighbor' grand-children of 'country' children of the top-level
# elements
root.findall("./country/neighbor")

# Nodes with name='Singapore' that have a 'year' child
root.findall(".//year/..[@name='Singapore']")

# 'year' nodes that are children of nodes with name='Singapore'
root.findall(".//*[@name='Singapore']/year")

# All 'neighbor' nodes that are the second child of their parent
root.findall(".//neighbor[2]")
```

对于带有命名空间的 XML，应使用通常的限定 `{namespace}tag` 标记法:

```
# All dublin-core "title" tags in the document
root.findall(".//{http://purl.org/dc/elements/1.1/}title")
```

支持的 XPath 语法

语法	含义
`tag`	选择具有给定标记的所有子元素。例如，`spam` 是选择名为 `spam` 的所有子元素，而 `spam/egg` 是在名为 `spam` 的子元素中选择所有名为 `egg` 的孙元素，`{*}spam` 是在任意（或无）命名空间中选择名为 `spam` 的标记，而 `{}*` 是只选择不在一个命名空间中的标记。 3.8 版更變: 增加了对星号通配符的支持。
`*`	选择所有子元素，包括注释和处理说明。例如 `*/egg` 选择所有名为 `egg` 的孙元素。
`.`	选择当前节点。这在路径的开头非常有用，用于指示它是相对路径。
`//`	选择所有子元素在当前元素的所有下级中选择所有下级元素。例如，`.//egg` 是在整个树中选择所有 `egg` 元素。
`..`	选择父元素。如果路径试图前往起始元素的上级（元素的 `find` 被调用）则返回 `None`。
`[@attrib]`	选择具有给定属性的所有元素。
`[@attrib='value']`	选择给定属性具有给定值的所有元素。该值不能包含引号。
`[tag]`	选择所有包含 `tag` 子元素的元素。只支持直系子元素。
`[.='text']`	选择完整文本内容等于 `text` 的所有元素（包括后代）。 3.7 版新加入.
`[tag='text']`	选择所有包含名为 `tag` 的子元素的元素，这些子元素（包括后代）的完整文本内容等于给定的 `text` 。
`[position]`	选择位于给定位置的所有元素。位置可以是一个整数 (1 表示首位)，表达式 `last()` (表示末位)，或者相对于末位的位置 (例如 `last()-1`)。

谓词（方括号内的表达式）之前必须带有标签名称，星号或其他谓词。`position` 谓词前必须有标签名称。

20.5.3 参考引用

函数

xml.etree.ElementTree.**canonicalize**(*xml_data=None*, *, *out=None*, *from_file=None*, ***options*)

C14N 2.0 转换功能。.

规整化是标准化 XML 输出的一种方式，该方式允许按字节比较和数字签名。它降低了 XML 序列化器所具有的自由度并改为生成更受约束的 XML 表示形式。主要限制涉及命名空间声明的位置、属性的顺序和可忽略的空白符等。

此函数接受一个 XML 数字字符串 (*xml_data*) 或文件路径或者文件类对象 (*from_file*) 作为输入，将其转换为规整形式，并在提供了 *out* 文件（类）对象的情况下将其写到该对象的话，或者如果未提供则将其作为文本字符串返回。输出文件接受文本而非字节数据。因此它应当以使用 `utf-8` 编码格式的文本模式来打开。

典型使用:

```
xml_data = "<root>...</root>"
print(canonicalize(xml_data))

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(xml_data, out=out_file)

with open("c14n_output.xml", mode='w', encoding='utf-8') as out_file:
    canonicalize(from_file="inputfile.xml", out=out_file)
```

配置选项 *options* 如下:

- *with_comments*: 设为真值以包括注释 (默认为假值)
- ***strip_text*: 设为真值以去除文本内容前后的空白符**（默认值：否）
- ***rewrite_prefixes*: 设为真值以替换带有"n{number}" 前缀的命名空间**（默认值：否）
- ***qname_aware_tags*: 一组可感知限定名称的标记名称，其中的前缀** 应当在文本内容中被替换 (默认为空值)
- ***qname_aware_attrs*: 一组可感知限定名称的属性名称，其中的前缀** 应当在文本内容中被替换 (默认为空值)
- *exclude_attrs*: 一组不应当被序列化的属性名称
- *exclude_tags*: 一组不应当被序列化的标记名称

在上面的选项列表中，" 一组" 是指任意多项集或包含字符串的可迭代对象，排序是不必要的。

3.8 版新加入.

`xml.etree.ElementTree.`**`Comment`**(*text=None*)
注释元素工厂函数。这个工厂函数可创建一个特殊元素，它将被标准序列化器当作 XML 注释来进行序列化。注释字串可以是字节串或是 Unicode 字符串。*text* 是包含注释字串的字符串。返回一个表示注释的元素实例。

请注意*XMLParser* 会跳过输入中的注释而不会为其创建注释对象。*ElementTree* 将只在当使用某个*Element* 方法向树插入了注释节点时才会包含注释节点。

`xml.etree.ElementTree.`**`dump`**(*elem*)
将一个元素树或元素结构体写入到 sys.stdout。此函数应当只被用于调试。

实际输出格式是依赖于具体实现的。在这个版本中，它将以普通 XML 文件的格式写入。

elem 是一个元素树或单独元素。

3.8 版更變: *dump()* 函数现在会保留用户指定的属性顺序。

`xml.etree.ElementTree.`**`fromstring`**(*text*, *parser=None*)
根据一个字符串常量解析 XML 的节。与*XML()* 类似。*text* 是包含 XML 数据的字符串。*parser* 是可选的解析器实例。如果未给出，则会使用标准*XMLParser* 解析器。返回一个*Element* 实例。

`xml.etree.ElementTree.`**`fromstringlist`**(*sequence*, *parser=None*)
根据一个字符串片段序列解析 XML 文档。*sequence* 是包含 XML 数据片段的列表或其他序列对象。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个*Element* 实例。

3.2 版新加入.

`xml.etree.ElementTree.`**`indent`**(*tree*, *space=" "*, *level=0*)
添加空格到子树来实现树的缩进效果。这可以被用来生成美化打印的 XML 输出。*tree* 可以为 Element 或 ElementTree。*space* 是对应将被插入的每个缩进层级的空格字符串，默认为两个空格符。要对已缩进的树的部分子树进行缩进，请传入初始缩进层级作为 *level*。

3.9 版新加入.

`xml.etree.ElementTree.`**`iselement`**(*element*)
检测一个对象是否为有效的元素对象。*element* 是一个元素实例。如果对象是一个元素对象则返回 `True`。

`xml.etree.ElementTree.`**`iterparse`**(*source*, *events=None*, *parser=None*)
以增量方式将一个 XML 节解析为元素树，并向用户报告执行情况。*source* 是包含 XML 数据的文件名或*file object*。*events* 是要报告的事件序列。所支持的事件字符串有 `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` 和 `"end-ns"` (”ns” 事件用于获取详细的命名空间信息)。如果 *events* 被省略，则只有 `"end"` 事件会被报告。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解

析器。*parser* 必须为*XMLParser* 的子类并且只能使用默认的*TreeBuilder* 作为目标。返回一个提供 `(event, elem)` 对的*iterator*。

请注意虽然*iterparse()* 是以增量方式构建树，但它会对 *source*（或其所指定的文件）发出阻塞式读取。因此，它不适用于不可执行阻塞式读取的应用。对于完全非阻塞式的解析，请参看*XMLPullParser*。

備𝔽: *iterparse()* 只会确保当发出"start" 事件时看到了开始标记的">" 字符，因而在这个点上属性已被定义，但文本容和末尾属性还未被定义。这同样适用于元素的下级；它们可能存在也可能不存在。

如果你需要已完全填充的元素，请改为查找"end" 事件。

3.4 版後已𝔽用: *parser* 参数。

3.8 版更變: 增加了 `comment` 和 `pi` 事件。

`xml.etree.ElementTree.`**`parse`**(*source*, *parser=None*)
将一个 XML 的节解析为元素树。*source* 是包含 XML 数据的文件名或文件对象。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个*ElementTree* 实例。

`xml.etree.ElementTree.`**`ProcessingInstruction`**(*target*, *text=None*)
PI 元素工厂函数。这个工厂函数可创建一个特殊元素，它将被当作 XML 处理指令来进行序列化。*target* 是包含 PI 目标的字符串。*text* 如果给出则是包含 PI 内容的字符串。返回一个表示处理指令的元素实例。

请注意*XMLParser* 会跳过输入中的处理指令而不会为其创建注释对象。*ElementTree* 将只在当使用某个*Element* 方法向树插入了处理指令节点时才会包含处理指令节点。

`xml.etree.ElementTree.`**`register_namespace`**(*prefix*, *uri*)
注册一个命名空间前缀。这个注册表是全局的，并且任何对应给定前缀或命名空间 URI 的现有映射都会被移除。*prefix* 是命名空间前缀。*uri* 是命名空间 URI。如果可能的话，这个命名空间中的标记和属性将附带给定的前缀来进行序列化。

3.2 版新加入.

`xml.etree.ElementTree.`**`SubElement`**(*parent*, *tag*, *attrib={}*, ***extra*)
子元素工厂函数。这个函数会创建一个元素实例，并将其添加到现有的元素。

元素名、属性名和属性值可以是字节串或 Unicode 字符串。*parent* 是父元素。*tag* 是子元素名。*attrib* 是一个可选的字典，其中包含元素属性。*extra* 包含额外的属性，以关键字参数形式给出。返回一个元素实例。

`xml.etree.ElementTree.`**`tostring`**(*element*, *encoding="us-ascii"*, *method="xml"*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)
生成一个 XML 元素的字符串表示形式，包括所有子元素。*element* 是一个*Element* 实例。*encoding*[1] 是输出编码格式（默认为 US-ASCII）。请使用 `encoding="unicode"` 来生成 Unicode 字符串（否则生成字节串）。*method* 是 `"xml"`, `"html"` 或 `"text"` (默认为 `"xml"`)。*xml_declaration*, *default_namespace* 和 *short_empty_elements* 具有与*ElementTree.write()* 中一致的含义。返回一个包含 XML 数据（可选）已编码的字符串。

3.4 版新加入: *short_empty_elements* 形参。

3.8 版新加入: *xml_declaration* 和 *default_namespace* 形参。

3.8 版更變: *tostring()* 函数现在会保留用户指定的属性顺序。

`xml.etree.ElementTree.`**`tostringlist`**(*element*, *encoding="us-ascii"*, *method="xml"*, ***, *xml_declaration=None*, *default_namespace=None*, *short_empty_elements=True*)

[1] 包括在 XML 输出中的编码格式字符串应当符合适当的标准。例如"UTF-8" 是有效的，但"UTF8" 是无效的。请参阅 https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 和 https://www.iana.org/assignments/character-sets/character-sets.xhtml。

生成一个 XML 元素的字符串表示形式，包括所有子元素。*element* 是一个*Element* 实例。*encoding*[1] 是输出编码格式（默认为 US-ASCII）。请使用 `encoding="unicode"` 来生成 Unicode 字符串（否则生成字节串）。*method* 是 `"xml"`, `"html"` 或 `"text"` (默认为 `"xml"`)。*xml_declaration*, *default_namespace* 和 *short_empty_elements* 具有与*ElementTree.write()* 中一致的含义。返回一个包含 XML 数据（可选）已编码字符串的列表。它并不保证任何特定的序列，除了 `b"".join(tostringlist(element)) == tostring(element)`。

3.2 版新加入.

3.4 版新加入: *short_empty_elements* 形参。

3.8 版新加入: *xml_declaration* 和 *default_namespace* 形参。

3.8 版更變: *tostringlist()* 函数现在会保留用户指定的属性顺序。

`xml.etree.ElementTree.`**`XML`**(*text*, *parser=None*)

根据一个字符串常量解析 XML 的节。此函数可被用于在 Python 代码中嵌入 “XML 字面值”。*text* 是包含 XML 数据的字符串。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个*Element* 实例。

`xml.etree.ElementTree.`**`XMLID`**(*text*, *parser=None*)

根据一个字符串常量解析 XML 的节，并且还将返回一个将元素的 id:s 映射到元素的字典。*text* 是包含 XML 数据的字符串。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回一个包含*Element* 实例和字典的元组。

20.5.4 XInclude 支持

此模块通过 `xml.etree.ElementInclude` 辅助模块提供了对 XInclude 指令 的有限支持，这个模块可被用来根据元素树的信息在其中插入子树和文本字符串。

示例

以下是一个演示 XInclude 模块用法的例子。要在当前文本中包括一个 XML 文档，请使用 `{http://www.w3.org/2001/XInclude}include` 元素并将 **parse** 属性设为 `"xml"`，并使用 **href** 属性来指定要包括的文档。

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="source.xml" parse="xml" />
</document>
```

默认情况下，**href** 属性会被当作文件名来处理。你可以使用自定义加载器来重载此行为。还要注意标准辅助器不支持 XPointer 语法。

要处理这个文件，请正常加载它，并将根元素传给*xml.etree.ElementTree* 模块:

```
from xml.etree import ElementTree, ElementInclude

tree = ElementTree.parse("document.xml")
root = tree.getroot()

ElementInclude.include(root)
```

ElementInclude 模块使用来自 **source.xml** 文档的根元素替代 `{http://www.w3.org/2001/XInclude}include` 元素。结果看起来大概是这样:

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <para>This is a paragraph.</para>
</document>
```

如果省略了 **parse** 属性，它会取默认的"xml"。要求有 href 属性。

要包括文本文档，请使用 {http://www.w3.org/2001/XInclude}include 元素，并将 **parse** 属性设为"text":

```
<?xml version="1.0"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) <xi:include href="year.txt" parse="text" />.
</document>
```

结果可能如下所示：

```
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  Copyright (c) 2003.
</document>
```

20.5.5 参考引用

函数

xml.etree.ElementInclude.**default_loader**(*href*, *parse*, *encoding=None*)
默认的加载器。这个默认的加载器会从磁盘读取所包括的资源。*href* 是一个 URL。*parse* 是"xml" 或"text" 表示解析模式。*encoding* 是可选的文本编码格式。如果未给出，则编码格式为 utf-8。返回已扩展的资源。如果解析模式为 "xml"，则它是一个 ElementTree 实例。如果解析模式为"text"，则它是一个 Unicode 字符串。如果加载器失败，它可以返回 None 或者引发异常。

xml.etree.ElementInclude.**include**(*elem*, *loader=None*, *base_url=None*, *max_depth=6*)
这个函数会扩展 XInclude 指令。*elem* 是根元素。*loader* 是可选的资源加载器。如果省略，则它默认为*default_loader()*。如果给出，则它应当是一个实现了与*default_loader()* 相同的接口的可调用对象。*base_url* 是原文件的基准 URL，用于求解相对的包括文件引用。*max_depth* 是递归包括的最大数量。此限制是为了降低恶意内容爆破的风险。传入一个负值可禁用此限制。

返回已扩展的资源。如果解析模式为 "xml"，则它是一个 ElementTree 实例。如果解析模式为"text"，则它是一个 Unicode 字符串。如果加载器失败，它可以返回 None 或者引发异常。

3.9 版新加入: *base_url* 和 *max_depth* 形参。

元素对象

class xml.etree.ElementTree.**Element**(*tag*, *attrib={}*, ***extra*)
元素类。这个类定义了 Element 接口，并提供了这个接口的引用实现。

元素名、属性名和属性值可以是字节串或 Unicode 字符串。*tag* 是元素名。*attrib* 是一个可选的字典，其中包含元素属性。*extra* 包含额外的属性，以关键字参数形式给出。

tag
一个标识此元素意味着何种数据的字符串 (换句话说，元素类型)。

text
tail
这些属性可被用于存放与元素相关联的额外数据。它们的值通常为字符串但也可以是任何应用专属的对象。如果元素是基于 XML 文件创建的，*text* 属性会存放元素的开始标记及其第一个子元素

或结束标记之间的文本，或者为 None，而 *tail* 属性会存放元素的结束标记及下一个标记之间的文本，或者为 None。对于 XML 数据

```
<a><b>1<c>2<d/>3</c></b>4</a>
```

a 元素的 *text* 和 *tail* 属性均为 None，*b* 元素的 *text* 为 "1" 而 *tail* 为 "4"，*c* 元素的 *text* 为 "2" 而 *tail* 为 None，*d* 元素的 *text* 为 None 而 *tail* 为 "3"。

要获取一个元素的内部文本，请参阅*itertext()*，例如 "".join(element.itertext())。

应用程序可以将任意对象存入这些属性。

attrib
一个包含元素属性的字典。请注意虽然 *attrib* 值总是一个真正可变的 Python 字典，但 ElementTree 实现可以选择其他内部表示形式，并只在有需要时才创建字典。为了发挥这种实现的优势，请在任何可能情况下使用下列字典方法。

以下字典类方法作用于元素属性。

clear()
重设一个元素。此方法会移除所有子元素，清空所有属性，并将 text 和 tail 属性设为 None。

get(*key*, *default=None*)
获取名为 *key* 的元素属性。

返回属性的值，或者如果属性未找到则返回 *default*。

items()
将元素属性以 (name, value) 对序列的形式返回。所返回属性的顺序任意。

keys()
将元素属性名称以列表的形式返回。所返回名称的顺序任意。

set(*key*, *value*)
将元素的 *key* 属性设为 *value*。

以下方法作用于元素的下级（子元素）。

append(*subelement*)
将元素 *subelement* 添加到此元素的子元素内部列表。如果 *subelement* 不是一个*Element* 则会引发*TypeError*。

extend(*subelements*)
使用具有零个或多个元素的序列对象添加 *subelements*。如果某个子元素不是*Element* 则会引发*TypeError*。

3.2 版新加入.

find(*match*, *namespaces=None*)
查找第一个匹配 *match* 的子元素。*match* 可以是一个标记名称或者路径。返回一个元素实例或 None。*namespaces* 是可选的从命名空间前缀到完整名称的映射。传入 '' 作为前缀可将表达式中所有无前缀的标记名称移动到给定的命名空间。

findall(*match*, *namespaces=None*)
根据标记名称或者路径 查找所有匹配的子元素。返回一个包含所有匹配元素按文档顺序排序的列表。*namespaces* 是可选的从命名空间前缀到完整名称的映射。传入 '' 作为前缀可将表达式中所有无前缀的标记名称移动到给定的命名空间。

findtext(*match*, *default=None*, *namespaces=None*)
查找第一个匹配 *match* 的子元素的文本。*match* 可以是一个标记名称或者路径。反回第一个匹配的元素的文本内容，或者如果元素未找到则返回 *default*。请注意如果匹配的元素没有文本内容则会返回一个空字符串。*namespaces* 是可选的从命名空间前缀到完整名称的映射。传入 '' 作为前缀可将表达式中所有无前缀的标记名称移动到给定的命名空间。

insert(*index*, *subelement*)
将 *subelement* 插入到此元素的给定位置中。如果 *subelement* 不是一个*Element* 则会引发*TypeError*。

iter(*tag=None*)
创建一个以当前元素为根元素的树的*iterator*。该迭代器将以文档（深度优先）顺序迭代此元素及其所有下级元素。如果 *tag* 不为 None 或 '*'，则迭代器只返回标记为 *tag* 的元素。如果树结构在迭代期间被修改，则结果是未定义的。

3.2 版新加入.

iterfind(*match*, *namespaces=None*)
根据标记名称或者路径查找所有匹配的子元素。返回一个按文档顺序产生所有匹配元素的可迭代对象。*namespaces* 是可选的从命名空间前缀到完整名称的映射。

3.2 版新加入.

itertext()
创建一个文本迭代器。该迭代器将按文档顺序遍历此元素及其所有子元素，并返回所有内部文本。

3.2 版新加入.

makeelement(*tag*, *attrib*)
创建一个与此元素类型相同的新元素对象。请不要调用此方法，而应改用*SubElement()* 工厂函数。

remove(*subelement*)
从元素中移除 *subelement*。与 find* 方法不同的是此方法会基于实例的标识来比较元素，而不是基于标记的值或内容。

Element 对象还支持下列序列类型方法以配合子元素使用: __delitem__(), __getitem__(), __setitem__(), __len__()。

注意：不带子元素的元素将被检测为 False。此行为将在未来的版本中发生变化。请改用 len(elem) 或 elem is None 进行检测。

```
element = root.find('foo')

if not element:  # careful!
    print("element not found, or element has no subelements")

if element is None:
    print("element not found")
```

在 Python 3.8 之前，元素的 XML 属性的序列化顺序会通过按其名称排序来强制使其可被预期。由于现在字典已保证是有序的，这个强制重排序在 Python 3.8 中已被移除以保留原本由用户代码解析或创建的属性顺序。

通常，用户代码应当尽量不依赖于特定的属性顺序，因为 XML 信息设定 明确地排除了用属性顺序转递信息的做法。代码应当准备好处理任何输入顺序。对于要求确定性的 XML 输出的情况，例如加密签名或检测数据集等，可以通过规范化*canonicalize()* 函数来进行传统的序列化。

对于规范化输出不可用但仍然要求输出特定属性顺序的情况，代码应当设法直接按要求的顺序来创建属性，以避免代码阅读者产生不匹配的感觉。如果这一点是难以做到的，可以在序列化之前应用以下写法来强制实现顺序不依赖于元素的创建:

```
def reorder_attributes(root):
    for el in root.iter():
        attrib = el.attrib
        if len(attrib) > 1:
```

(下页继续)

(繼續上一頁)

```
            # adjust attribute order, e.g. by sorting
            attribs = sorted(attrib.items())
            attrib.clear()
            attrib.update(attribs)
```

ElementTree 对象

class xml.etree.ElementTree.**ElementTree**(*element=None*, *file=None*)

ElementTree 包装器类。这个类表示一个完整的元素层级结构，并添加了一些对于标准 XML 序列化的额外支持。

element 是根元素。如果给出 XML *file* 则将使用其内容来初始化树结构。

_setroot(*element*)

替换该树结构的根元素。这将丢弃该树结构的当前内容，并将其替换为给定的元素。请小心使用。*element* 是一个元素实例。

find(*match*, *namespaces=None*)

与*Element.find()* 类似，从树的根节点开始。

findall(*match*, *namespaces=None*)

与*Element.findall()* 类似，从树的根节点开始。

findtext(*match*, *default=None*, *namespaces=None*)

与*Element.findtext()* 类似，从树的根节点开始。

getroot()

返回这个树的根元素。

iter(*tag=None*)

创建并返回根元素的树结构迭代器。该迭代器会以节顺序遍历这个树的所有元素。*tag* 是要查找的标记（默认返回所有元素）。

iterfind(*match*, *namespaces=None*)

与*Element.iterfind()* 类似，从树的根节点开始。

3.2 版新加入.

parse(*source*, *parser=None*)

将一个外部 XML 节载入到此元素树。*source* 是一个文件名或*file object*。*parser* 是可选的解析器实例。如果未给出，则会使用标准的*XMLParser* 解析器。返回该节的根元素。

write(*file*, *encoding="us-ascii"*, *xml_declaration=None*, *default_namespace=None*, *method="xml"*, *, *short_empty_elements=True*)

将元素树以 XML 格式写入到文件。*file* 为文件名，或是以写入模式打开的*file object*。*encoding*[1] 为输出编码格式 (默认为 US-ASCII)。*xml_declaration* 控制是否要将 XML 声明添加到文件中。使用 `False` 表示从不添加，`True` 表示总是添加，`None` 表示仅在非 US-ASCII 或 UTF-8 或 Unicode 时添加 (默认为 `None`)。*default_namespace* 设置默认 XML 命名空间 (用于"xmlns")。*method* 为 `"xml"`, `"html"` 或 `"text"` (默认为 `"xml"`)。仅限关键字形参 *short_empty_elements* 控制不包含内容的元素的格式。如为 `True` (默认值)，它们会被输出为单个自结束标记，否则它们会被输出为一对开始/结束标记。

输出是一个字符串 (*str*) 或字节串 (*bytes*)。由 *encoding* 参数来控制。如果 *encoding* 为 `"unicode"`，则输出是一个字符串；否则为字节串；请注意这可能与 *file* 的类型相冲突，如果它是一个打开的*file object* 的话；请确保你不会试图写入字符串到二进制流或者反向操作。

3.4 版新加入: *short_empty_elements* 形参。

3.8 版更變: *write()* 方法现在会保留用户指定的属性顺序。

这是将要被操作的 XML 文件:

```
<html>
    <head>
        <title>Example page</title>
    </head>
    <body>
        <p>Moved to <a href="http://example.org/">example.org</a>
        or <a href="http://example.com/">example.com</a>.</p>
    </body>
</html>
```

修改第一段中的每个链接的"target" 属性的示例:

```
>>> from xml.etree.ElementTree import ElementTree
>>> tree = ElementTree()
>>> tree.parse("index.xhtml")
<Element 'html' at 0xb77e6fac>
>>> p = tree.find("body/p")      # Finds first occurrence of tag p in body
>>> p
<Element 'p' at 0xb77ec26c>
>>> links = list(p.iter("a"))    # Returns list of all links
>>> links
[<Element 'a' at 0xb77ec2ac>, <Element 'a' at 0xb77ec1cc>]
>>> for i in links:             # Iterates through all found links
...     i.attrib["target"] = "blank"
>>> tree.write("output.xhtml")
```

QName 对象

class xml.etree.ElementTree.**QName**(*text_or_uri*, *tag=None*)

QName 包装器。这可被用来包装 QName 属性值，以便在输出中获得适当的命名空间处理。*text_or_uri* 是一个包含 QName 值的字符串，其形式为 {uri}local，或者如果给出了 tag 参数，则为 QName 的 URI 部分。如果给出了 *tag*，则第一个参数会被解读为 URI，而这个参数会被解读为本地名称。*QName* 实例是不透明的。

TreeBuilder 对象

class xml.etree.ElementTree.**TreeBuilder**(*element_factory=None*, ***, *comment_factory=None*, *pi_factory=None*, *insert_comments=False*, *insert_pis=False*)

通用元素结构构建器。此构建器会将包含 start, data, end, comment 和 pi 方法调用的序列转换为格式良好的元素结构。你可以通过这个类使用一个自定义 XML 解析器或其他 XML 类格式的解析器来构建元素结构。

如果给出 *element_factory*，它必须为接受两个位置参数的可调用对象：一个标记和一个属性字典。它预期会返回一个新的元素实例。

如果给出 *comment_factory* 和 *pi_factory* 函数，它们的行为应当像*Comment()* 和*ProcessingInstruction()* 函数一样创建注释和处理指令。如果未给出，则将使用默认工厂函数。当 *insert_comments* 和/或 *insert_pis* 为真值时，如果 comments/pis 在根元素之中（但不在其之外）出现则它们将被插入到树中。

close()

刷新构建器缓存，并返回最高层级的文档元素。返回一个*Element* 实例。

data(*data*)
将文本添加到当前元素。*data* 为要添加的文本。这应当是一个字节串或 Unicode 字符串。

end(*tag*)
关闭当前元素。*tag* 是元素名称。返回已关闭的元素。

start(*tag*, *attrs*)
打开一个新元素。*tag* 是元素名称。*attrs* 是包含元素属性的字典。返回打开的元素。

comment(*text*)
使用给定的 *text* 创建一条注释。如果 `insert_comments` 为真值，这还会将其添加到树结构中。

3.8 版新加入.

pi(*target*, *text*)
使用给定的 *target* 名称和 *text* 创建一条注释。如果 `insert_pis` 为真值，这还会将其添加到树结构中。

3.8 版新加入.

此外，自定义的*TreeBuilder* 对象还提供了以下方法:

doctype(*name*, *pubid*, *system*)
处理一条 doctype 声明。*name* 为 doctype 名称。*pubid* 为公有标识。*system* 为系统标识。此方法不存在于默认的*TreeBuilder* 类中。

3.2 版新加入.

start_ns(*prefix*, *uri*)
在定义了 `start()` 回调的打开元素的该回调被调用之前，当解析器遇到新的命名空间声明时都会被调用。*prefix* 对于默认命名空间为 `''` 或者在其他情况下为被声明的命名空间前缀名称。*uri* 是命名空间 URI。

3.8 版新加入.

end_ns(*prefix*)
在声明了命名空间前缀映射的元素的 `end()` 回调之后被调用，附带超出作用域的 *prefix* 的名称。

3.8 版新加入.

class xml.etree.ElementTree.**C14NWriterTarget**(*write*, *, *with_comments=False*, *strip_text=False*, *rewrite_prefixes=False*, *qname_aware_tags=None*, *qname_aware_attrs=None*, *exclude_attrs=None*, *exclude_tags=None*)
C14N 2.0 写入器。其参数与*canonicalize()* 函数的相同。这个类并不会构建树结构而是使用 *write* 函数将回调事件直接转换为序列化形式。

3.8 版新加入.

XMLParser 对象

class xml.etree.ElementTree.**XMLParser**(*, *target=None*, *encoding=None*)
这个类是此模块的低层级构建单元。它使用*xml.parsers.expat* 来实现高效、基于事件的 XML 解析。它可以通过*feed()* 方法增量式地收受 XML 数据，并且解析事件会被转换为推送式 API ——通过在 *target* 对象上发起对回调的调用。如果省略 *target*，则会使用标准的*TreeBuilder*。如果给出了 *encoding*[1] ，该值将覆盖在 XML 文件中指定的编码格式。

3.8 版更變: 所有形参现在都是仅限关键字形参。*html* 参数不再受支持。

close()
结束向解析器提供数据。返回调用在构造期间传入的 *target* 的 `close()` 方法的结果；在默认情况下，这是最高层级的文档元素。

feed(*data*)
将数据送入解析器。*data* 是编码后的数据。

XMLParser.feed() 会为每个打开的标记调用 *target* 的 `start(tag, attrs_dict)` 方法，为每个关闭的标记调用它的 `end(tag)` 方法，并通过 `data(data)` 方法来处理数据。有关更多受支持的回调方法，请参阅 *TreeBuilder* 类。*XMLParser.close()* 会调用 *target* 的 `close()` 方法。*XMLParser* 不仅仅可被用来构建树结构。下面是一个统计 XML 文件最大深度的示例:

```
>>> from xml.etree.ElementTree import XMLParser
>>> class MaxDepth:                     # The target object of the parser
...     maxDepth = 0
...     depth = 0
...     def start(self, tag, attrib):   # Called for each opening tag.
...         self.depth += 1
...         if self.depth > self.maxDepth:
...             self.maxDepth = self.depth
...     def end(self, tag):             # Called for each closing tag.
...         self.depth -= 1
...     def data(self, data):
...         pass            # We do not need to do anything with data.
...     def close(self):    # Called when all data has been parsed.
...         return self.maxDepth
...
>>> target = MaxDepth()
>>> parser = XMLParser(target=target)
>>> exampleXml = """
... <a>
...   <b>
...   </b>
...   <b>
...     <c>
...       <d>
...       </d>
...     </c>
...   </b>
... </a>"""
>>> parser.feed(exampleXml)
>>> parser.close()
4
```

XMLPullParser 对象

class xml.etree.ElementTree.**XMLPullParser**(*events=None*)
适用于非阻塞应用程序的拉取式解析器。它的输入侧 API 与 *XMLParser* 的类似，但不是向回调目标推送调用，*XMLPullParser* 会收集一个解析事件的内部列表并让用户来读取它。*events* 是要报告的事件序列。受支持的事件字符串有 `"start"`, `"end"`, `"comment"`, `"pi"`, `"start-ns"` 和 `"end-ns"` ("ns" 事件被用于获取详细的命名空间信息)。如果 *events* 被省略，则只报告 `"end"` 事件。

feed(*data*)
将给定的字节数据送入解析器。

close()
通知解析器数据流已终结。不同于 *XMLParser.close()*，此方法总是返回 *None*。当解析器被关闭时任何还未被获取的事件仍可通过 *read_events()* 被读取。

read_events()
返回包含在送入解析器的数据中遇到的事件的迭代器。此迭代器会产生 (event, elem) 对，其中 *event* 是代表事件类型的字符串 (例如 "end") 而 *elem* 是遇到的*Element* 对象，或者以下的其他上下文值。

- start, end: 当前元素。
- comment, pi: 当前注释 / 处理指令
- start-ns: 一个指定所声明命名空间映射的元组 (prefix, uri)。
- end-ns: *None* (这可能在未来版本中改变)

在之前对*read_events()* 的调用中提供的事件将不会被再次产生。事件仅当它们从迭代器中被取出时才会在内部队列中被消费，因此多个读取方对获取自*read_events()* 的迭代器进行平行迭代将产生无法预料的结果。

備🄵: *XMLPullParser* 只会确保当发出"start" 事件时看到了开始标记的">" 字符，因而在这个点上属性已被定义，但文本内容和末尾属性还未被定义。这同样适用于元素的下级；它们可能存在也可能不存在。

如果你需要已完全填充的元素，请改为查找"end" 事件。

3.4 版新加入.

3.8 版更變: 增加了 comment 和 pi 事件。

异常

class xml.etree.ElementTree.**ParseError**
XML 解析器错误，由此模块中的多个解析方法在解析失败时引发。此异常的实例的字符串表示将包含用户友好的错误消息。此外，它将具有下列可用属性:

code
来自外部解析器的数字错误代码。请参阅*xml.parsers.expat* 的文档查看错误代码列表及它们的含义。

position
一个包含 *line*, *column* 数值的元组，指明错误发生的位置。

🄵解

20.6 xml.dom --- 文档对象模型 API

源代码: Lib/xml/dom/__init__.py

文档对象模型 "DOM" 是一个来自万维网联盟（W3C）的跨语言 API，用于访问和修改 XML 文档。DOM 的实现将 XML 文档以树结构表示，或者允许客户端代码从头构建这样的结构。然后它会通过一组提供通用接口的对象赋予对结构的访问权。

DOM 特别适用于进行随机访问的应用。SAX 仅允许你每次查看文档的一小部分。如果你正在查看一个 SAX 元素，你将不能访问其他元素。如果你正在查看一个文本节点，你将不能访问包含它的元素。当你编写一个 SAX 应用时，你需要在你自己的代码的某个地方记住你的程序在文档中的位置。SAX 不会帮你做这件事。并且，如果你想要在 XML 文档中向前查看，你是绝对办不到的。

有些应用程序在不能访问树的事件驱动模型中是根本无法编写的。当然你可以在 SAX 事件中自行构建某种树，但是 DOM 可以使你避免编写这样的代码。DOM 是针对 XML 数据的标准树表示形式。

文档对象模型是由 W3C 分阶段定义的，在其术语中称为“层级”。Python 中该 API 的映射大致是基于 DOM 第 2 层级的建议。

DOM 应用程序通常从将某些 XML 解析为 DOM 开始。此操作如何实现完全未被 DOM 第 1 层级所涉及，而第 2 层级也只提供了有限的改进：有一个 `DOMImplementation` 对象类，它提供对 `Document` 创建方法的访问，但却没有办法以不依赖具体实现的方式访问 XML 读取器/解析器/文档创建器。也没有当不存在 `Document` 对象的情况下访问这些方法的定义良好的方式。在 Python 中，每个 DOM 实现将提供一个函数 *`getDOMImplementation()`*。DOM 第 3 层级增加了一个载入/存储规格说明，它定义了与读取器的接口，但这在 Python 标准库中尚不可用。

一旦你得到了 DOM 文档对象，你就可以通过 XML 文档的属性和方法访问它的各个部分。这些属性定义在 DOM 规格说明当中；参考指南的这一部分描述了 Python 对此规格说明的解读。

W3C 提供的规格说明定义了适用于 Java, ECMAScript 和 OMG IDL 的 DOM API。这里定义的 Python 映射很大程度上是基于此规格说明的 IDL 版本，但并不要求严格映射（但具体实现可以自由地支持对 IDL 的严格映射）。请参阅一致性 一节查看有关映射要求的详细讨论。

也参考:

文档对象模型 (DOM) 第 2 层级规格说明 被 Python DOM API 作为基础的 W3C 建议。

文档对象模型 (DOM) 第 1 层级规格说明 被 *`xml.dom.minidom`* 所支持的 W3C 针对 DOM 的建议。

Python Language Mapping Specification 此文档指明了从 OMG IDL 到 Python 的映射。

20.6.1 模块内容

`xml.dom` 包含下列函数:

xml.dom.**registerDOMImplementation**(*name*, *factory*)
注册 *factory* 函数并使用名称 *name*。该工厂函数应当返回一个实现了 `DOMImplementation` 接口的对象。该工厂函数可每次都返回相同对象，或每次调用都返回新的对象，视具体实现的要求而定（例如该实现是否支持某些定制功能）。

xml.dom.**getDOMImplementation**(*name=None*, *features=()*)
返回一个适当的 DOM 实现。*name* 是通用名称、DOM 实现的模块名称或者 `None`。如果它不为 `None`，则会导入相应模块并在导入成功时返回一个 `DOMImplementation` 对象。如果没有给出名称，并且如果设置了 `PYTHON_DOM` 环境变量，此变量会被用来查找相应的实现。

如果未给出 name，此函数会检查可用的实现来查找具有所需特性集的一个。如果找不到任何实现，则会引发 *`ImportError`*。features 集必须是包含 `(feature, version)` 对的序列，它会被传给可用的 `DOMImplementation` 对象上的 `hasFeature()` 方法。

还提供了一些便捷常量:

xml.dom.**EMPTY_NAMESPACE**
该值用于指明没有命名空间被关联到 DOM 中的某个节点。它通常被作为某个节点的 `namespaceURI`，或者被用作某个命名空间专属方法的 *namespaceURI* 参数。

xml.dom.**XML_NAMESPACE**
关联到保留前缀 `xml` 的命名空间 URI，如 XML 中的命名空间 （第 4 节）所定义的。

xml.dom.**XMLNS_NAMESPACE**
命名空间声明的命名空间 URI，如 文档对象模型 (DOM) 第 2 层级核心规格说明 (第 1.1.8 节) 所定义的。

xml.dom.**XHTML_NAMESPACE**
XHTML 命名空间的 URI，如 XHTML 1.0: 扩展超文本标记语言 (第 3.1.1 节) 所定义的。

此外，*xml.dom* 还包含一个基本 Node 类和一些 DOM 异常类。此模块提供的 Node 类未实现 DOM 规格描述所定义的任何方法和属性；实际的 DOM 实现必须提供它们。提供 Node 类作为此模块的一部分并没有提供用于实际的 Node 对象的 nodeType 属性的常量；它们是位于类内而不是位于模块层级以符合 DOM 规格描述。

20.6.2 DOM 中的对象

DOM 的权威文档是来自 W3C 的 DOM 规格描述。

请注意，DOM 属性也可以作为节点而不是简单的字符串进行操作。然而，必须这样做的情况相当少见，所以这种用法还没有被写入文档。

接口	部件	目的
DOMImplementation	*DOMImplementation* 对象	底层实现的接口。
Node	节点对象	文档中大多数对象的基本接口。
NodeList	节点列表对象	节点序列的接口。
DocumentType	文档类型对象	有关处理文档所需声明的信息。
Document	*Document* 对象	表示整个文档的对象。
Element	元素对象	文档层次结构中的元素节点。
Attr	*Attr* 对象	元素节点上的属性值节点。
Comment	注释对象	源文档中注释的表示形式。
Text	*Text* 和 *CDATASection* 对象	包含文档中文本内容的节点。
ProcessingInstruction	*ProcessingInstruction* 对象	处理指令表示形式。

描述在 Python 中使用 DOM 定义的异常的小节。

DOMImplementation 对象

DOMImplementation 接口提供了一种让应用程序确定他们所使用的 DOM 中某一特性可用性的方式。DOM 第 2 级还添加了使用 DOMImplementation 来创建新的 Document 和 DocumentType 对象的能力。

DOMImplementation.**hasFeature**(*feature*, *version*)
如果字符串对 *feature* 和 *version* 所标识的特性已被实现则返回 True。

DOMImplementation.**createDocument**(*namespaceUri*, *qualifiedName*, *doctype*)
返回一个新的 Document 对象 (DOM 的根节点)，包含一个具有给定 *namespaceUri* 和 *qualifiedName* 的下级 Element 对象。*doctype* 必须为由 *createDocumentType()* 创建的 DocumentType 对象，或者为 None。在 Python DOM API 中，前两个参数也可为 None 以表示不要创建任何下级 Element。

DOMImplementation.**createDocumentType**(*qualifiedName*, *publicId*, *systemId*)
返回一个新的封装了给定 *qualifiedName*, *publicId* 和 *systemId* 字符串的 DocumentType 对象，它表示包含在 XML 文档类型声明中的信息。

节点对象

XML 文档的所有组成部分都是 Node 的子类。

Node.**nodeType**
一个代表节点类型的整数。类型符号常量在 Node 对象上: ELEMENT_NODE, ATTRIBUTE_NODE, TEXT_NODE, CDATA_SECTION_NODE, ENTITY_NODE, PROCESSING_INSTRUCTION_NODE, COMMENT_NODE, DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE。这是个只读属性。

Node.**parentNode**
当前节点的上级，或者对于文档节点则为 None。该值总是一个 Node 对象或者 None。对于 Element 节点，这将为上级元素，但对于根元素例外，在此情况下它将为 Document 对象。对于 Attr 节点，它将总是为 None。这是个只读属性。

Node.**attributes**
属性对象的 NamedNodeMap。这仅对元素才有实际值；其它对象会为该属性提供 None 值。这是个只读属性。

Node.**previousSibling**
在此节点之前具有相同上级的相邻节点。例如结束标记紧接在在 *self* 元素的开始标记之前的元素。当然，XML 文档并非只是由元素组成，因此之前相邻节点可以是文本、注释或者其他内容。如果此节点是上级的第一个子节点，则该属性将为 None。这是一个只读属性。

Node.**nextSibling**
在此节点之后具有相同上级的相邻节点。另请参见*previousSibling*。如果此节点是上级的最后一个子节点，则该属性将为 None。这是一个只读属性。

Node.**childNodes**
包含在此节点中的节点列表。这是一个只读属性。

Node.**firstChild**
节点的第一个下级，如果有的话，否则为 None。这是个只读属性。

Node.**lastChild**
节点的最后一个下级，如果有的话，否则为 None。这是个只读属性。

Node.**localName**
tagName 在冒号之后的部分，如果有冒号的话，否则为整个 tagName。该值为一个字符串。

Node.**prefix**
tagName 在冒号之前的部分，如果有冒号的话，否则为空字符串。该值为一个字符串或者为 None。

Node.**namespaceURI**
关联到元素名称的命名空间。这将是一个字符串或为 None。这是个只读属性。

Node.**nodeName**
这对于每种节点类型具有不同的含义；请查看 DOM 规格说明来了解详情。你总是可以从其他特征属性例如元素的 tagName 特征属性或属性的 name 特征属性获取你能从这里获取的信息。对于所有节点类型，这个属性的值都将是一个字符串或为 None。这是一个只读属性。

Node.**nodeValue**
这对于每种节点类型具有不同的含义；请查看 DOM 规格说明来了解详情。具体情况与*nodeName* 的类似。该值是一个字符串或为 None。

Node.**hasAttributes**()
如果该节点具有任何属性则返回 True。

Node.**hasChildNodes**()
如果该节点具有任何子节点则返回 True。

Node.**isSameNode**(*other*)
如果 *other* 指向的节点就是此节点则返回 True。这对于使用了任何代理架构的 DOM 实现来说特别有用（因为多个对象可能指向相同节点）。

備𝔽: 这是基于已提议的 DOM 第 3 等级 API，目前尚处于“起草”阶段，但这个特定接口看来并不存在争议。来自 W3C 的修改将不会影响 Python DOM 接口中的这个方法（不过针对它的任何新 W3C API 也将受到支持）。

Node.**appendChild**(*newChild*)
在子节点列表末尾添加一个新的子节点，返回 *newChild*。如果节点已存在于树结构中，它将先被移除。

Node.**insertBefore**(*newChild*, *refChild*)
在现有的子节点之前插入一个新的子节点。它必须属于 *refChild* 是这个节点的子节点的情况；如果不是，则会引发*ValueError*。*newChild* 会被返回。如果 *refChild* 为 None，它会将 *newChild* 插入到子节点列表的末尾。

Node.**removeChild**(*oldChild*)
移除一个子节点。*oldChild* 必须是这个节点的子节点；如果不是，则会引发*ValueError*。成功时 *oldChild* 会被返回。如果 *oldChild* 将不再被继续使用，则将调用它的 unlink() 方法。

Node.**replaceChild**(*newChild*, *oldChild*)
将一个现有节点替换为新的节点。这必须属于 *oldChild* 是该节点的子节点的情况；如果不是，则会引发*ValueError*。

Node.**normalize**()
合并相邻的文本节点以便将所有文本段存储为单个 Text 实例。这可以简化许多应用程序处理来自 DOM 树文本的操作。

Node.**cloneNode**(*deep*)
克隆此节点。设置 *deep* 表示也克隆所有子节点。此方法将返回克隆的节点。

节点列表对象

NodeList 代表一个节点列表。在 DOM 核心建议中这些对象有两种使用方式: 由 Element 对象提供作为其子节点列表，以及由 Node 的 getElementsByTagName() 和 getElementsByTagNameNS() 方法通过此接口返回对象来表示查询结果。

DOM 第 2 层级建议为这些对象定义一个方法和一个属性:

NodeList.**item**(*i*)
从序列中返回第 *i* 项，如果序列不为空的话，否则返回 None。索引号 *i* 不允许小于零或大于等于序列的长度。

NodeList.**length**
序列中的节点数量。

此外，Python DOM 接口还要求提供一些额外支持来允许将 NodeList 对象用作 Python 序列。所有 NodeList 实现都必须包括对 __len__() 和 __getitem__() 的支持；这样 NodeList 就允许使用 for 语句进行迭代并能正确地支持*len()* 内置函数。

如果一个 DOM 实现支持文档的修改，则 NodeList 实现还必须支持 __setitem__() 和 __delitem__() 方法。

文档类型对象

有关一个文档所声明的标注和实体的信息（包括解析器所使用并能提供信息的外部子集）可以从 DocumentType 对象获取。文档的 DocumentType 可从 Document 对象的 doctype 属性中获取；如果一个文档没有 DOCTYPE 声明，则该文档的 doctype 属性将被设为 None 而非此接口的一个实例。

DocumentType 是 Node 是专门化，并增加了下列属性:

DocumentType.**publicId**
: 文档类型定义的外部子集的公有标识。这将为一个字符串或者为 None。

DocumentType.**systemId**
: 文档类型定义的外部子集的系统标识。这将为一个字符串形式的 URI，或者为 None。

DocumentType.**internalSubset**
: 一个给出来自文档的完整内部子集的字符串。这不包括子集外面的圆括号。如果文档没有内部子集，则应为 None。

DocumentType.**name**
: DOCTYPE 声明中给出的根元素名称，如果有的话。

DocumentType.**entities**
: 这是给出外部实体定义的 NamedNodeMap。对于多次定义的实体名称，则只提供第一次的定义（其他的会按照 XML 建议被忽略）。这可能为 None，如果解析器未提供此信息，或者如果未定义任何实体的话。

DocumentType.**notations**
: 这是给出标注定义的 NamedNodeMap。对于多次定义的标注，则只提供第一次的定义（其他的会按照 XML 建议被忽略）。这可能为 None，如果解析器未提供此信息，或者如果未定义任何标注的话。

Document 对象

Document 代表一个完整的 XML 文档，包括其组成元素、属性、处理指令和注释等。请记住它会继承来自 Node 的属性。

Document.**documentElement**
: 文档唯一的根元素。

Document.**createElement**(*tagName*)
: 创建并返回一个新的元素节点。当元素被创建时不会被插入到文档中。你需要通过某个其他方法例如 insertBefore() 或 appendChild() 来显示地插入它。

Document.**createElementNS**(*namespaceURI*, *tagName*)
: 创建并返回一个新的带有命名空间的元素。*tagName* 可以带有前缀。当元素被创建时不会被插入到文档中。你需要通过某个其他方法例如 insertBefore() 或 appendChild() 来显式地插入它。

Document.**createTextNode**(*data*)
: 创建并返回一个包含作为形参被传入的数据的文本节点。与其他创建方法一样，此方法不会将节点插入到树中。

Document.**createComment**(*data*)
: 创建并返回一个包含作为形参被传入的数据的注释节点。与其他创建方法一样，此方法不会将节点插入到树中。

Document.**createProcessingInstruction**(*target*, *data*)
: 创建并返回一个包含作为形参被传入的 *target* 和 *data* 的处理指令节点。与其他创建方法一样，此方法不会将节点插入到树中。

Document.**createAttribute**(*name*)
创建并返回一个属性节点。此方法不会将属性节点关联到任何特定的元素。你必须在正确的 Element 对象上使用 setAttributeNode() 来使用新创建的属性实例。

Document.**createAttributeNS**(*namespaceURI*, *qualifiedName*)
创建并返回一个带有命名空间的属性节点。*tagName* 可以带有前缀。此方法不会将属性节点关联到任何特定的元素。你必须在正确的 Element 对象上使用 setAttributeNode() 来使用新创建的属性实例。

Document.**getElementsByTagName**(*tagName*)
搜索全部具有特定元素类型名称的后继元素（直接下级、下级的下级等等）。

Document.**getElementsByTagNameNS**(*namespaceURI*, *localName*)
搜索全部具有特定命名空间 URI 和 localname 的后继元素（直接下级、下级的下级等等）。localname 是命名空间在前缀之后的部分。

元素对象

Element 是 Node 的子类，因此会继承该类的全部属性。

Element.**tagName**
元素类型名称。在使用命名空间的文档中它可能包含冒号。该值是一个字符串。

Element.**getElementsByTagName**(*tagName*)
与 Document 类中的对应方法相同。

Element.**getElementsByTagNameNS**(*namespaceURI*, *localName*)
与 Document 类中的对应方法相同。

Element.**hasAttribute**(*name*)
如果元素带有名称为 *name* 的属性则返回 True。

Element.**hasAttributeNS**(*namespaceURI*, *localName*)
如果元素带有名称为 *namespaceURI* 加 *localName* 的属性则返回 True。

Element.**getAttribute**(*name*)
将名称为 *name* 的属性的值作为字符串返回。如果指定属性不存在，则返回空字符串，就像该属性没有对应的值一样。

Element.**getAttributeNode**(*attrname*)
返回名称为 *attrname* 的属性对应的 Attr 节点。

Element.**getAttributeNS**(*namespaceURI*, *localName*)
将名称为 *namespaceURI* 加 *localName* 的属性的值作为字符串返回。如果指定属性不存在，则返回空字符串，就像该属性没有对应的值一样。

Element.**getAttributeNodeNS**(*namespaceURI*, *localName*)
将给定 *namespaceURI* 加 *localName* 的属性的值作为节点返回。

Element.**removeAttribute**(*name*)
移除指定名称的节点。如果没有匹配的属性，则会引发*NotFoundErr*。

Element.**removeAttributeNode**(*oldAttr*)
从属性列表中移除并返回 *oldAttr*，如果该属性存在的话。如果 *oldAttr* 不存在，则会引发*NotFoundErr*。

Element.**removeAttributeNS**(*namespaceURI*, *localName*)
移除指定名称的属性。请注意它是使用 localName 而不是 qname。如果没有匹配的属性也不会引发异常。

Element.**setAttribute**(*name*, *value*)
将属性值设为指定的字符串。

Element.**setAttributeNode**(*newAttr*)
将一个新的属性节点添加到元素，当匹配到 name 属性时如有必要会替换现有的属性。如果发生了替换，将返回原有属性节点。如果 *newAttr* 已经被使用，则会引发*InuseAttributeErr*。

Element.**setAttributeNodeNS**(*newAttr*)
将一个新的属性节点添加到元素，当匹配到 namespaceURI 和 localName 属性时如有必要会替换现有的属性。如果发生了替换，将返回原有属性节点。如果 *newAttr* 已经被使用，则会引发*InuseAttributeErr*。

Element.**setAttributeNS**(*namespaceURI*, *qname*, *value*)
将属性值设为 *namespaceURI* 和 *qname* 所给出的字符串。请注意 qname 是整个属性名称。这与上面的方法不同。

Attr 对象

Attr 继承自 Node，因此会继承其全部属性。

Attr.**name**
属性名称。在使用命名空间的文档中可能会包括冒号。

Attr.**localName**
名称在冒号之后的部分，如果有的话，否则为完整名称。这是个只读属性。

Attr.**prefix**
名称在冒号之前的部分，如果有冒号的话，否则为空字符串。

Attr.**value**
属性的文本值。这与 nodeValue 属性同义。

NamedNodeMap 对象

NamedNodeMap 不是继承自 Node。

NamedNodeMap.**length**
属性列表的长度。

NamedNodeMap.**item**(*index*)
返回特定带有索引号的属性。获取属性的顺序是强制规定的，但在 DOM 的生命期内会保持一致。其中每一项均为属性节点。可使用 value 属性获取其值。

还有一些试验性方法给予这个类更多的映射行为。你可以使用这些方法或者使用 Element 对象上标准化的 getAttribute*() 方法族。

注释对象

Comment 代表 XML 文档中的注释。它是 Node 的子类，但不能拥有下级节点。

Comment.**data**
注释的内容是一个字符串。该属性包含在开头 <!-- 和末尾 --> 之间的所有字符，但不包括这两个符号。

Text 和 CDATASection 对象

`Text` 接口代表 XML 文档中的文本。如果解析器和 DOM 实现支持 DOM 的 XML 扩展，则包裹在 CDATA 标记的节中的部分会被存储到 `CDATASection` 对象中。这两个接口很相似，但是提供了不同的 `nodeType` 属性值。

这些接口扩展了 `Node` 接口。它们不能拥有下级节点。

`Text`.**data**
字符串形式的文本节点内容。

備𝔽: `CDATASection` 节点的使用并不表示该节点代表一个完整的 CDATA 标记节，只是表示该节点的内容是 CDATA 节的一部分。单个 CDATA 节可以由文档树中的多个节点来表示。没有什么办法能确定两个相邻的 `CDATASection` 节点是否代表不同的 CDATA 标记节。

ProcessingInstruction 对象

代表 XML 文档中的处理指令。它继承自 `Node` 接口并且不能拥有下级节点。

`ProcessingInstruction`.**target**
到第一个空格符为止的处理指令内容。这是个只读属性。

`ProcessingInstruction`.**data**
在第一个空格符之后的处理指令内容。

异常

DOM 第 2 层级推荐定义一个异常*DOMException*，以及多个变量用来允许应用程序确定发生了何种错误。*DOMException* 实例带有*code* 属性用来提供特定异常所对应的值。

Python DOM 接口提供了一些常量，但还扩展了异常集以使 DOM 所定义的每个异常代码都存在特定的异常。接口的具体实现必须引发正确的特定异常，它们各自带有正确的*code* 属性值。

exception `xml.dom`.**DOMException**
所有特定 DOM 异常所使用的异常基类。该异常类不可被直接实例化。

exception `xml.dom`.**DomstringSizeErr**
当指定范围的文本不能适配一个字符串时被引发。此异常在 Python DOM 实现中尚不可用，但可从不是以 Python 编写的 DOM 实现中接收。

exception `xml.dom`.**HierarchyRequestErr**
当尝试插入一个节点但该节点类型不被允许时被引发。

exception `xml.dom`.**IndexSizeErr**
当一个方法的索引或大小参数为负值或超出允许的值范围时被引发。

exception `xml.dom`.**InuseAttributeErr**
当尝试插入一个 `Attr` 节点但该节点已存在于文档中的某处时被引发。

exception `xml.dom`.**InvalidAccessErr**
当某个参数或操作在底层对象中不受支持时被引发。

exception `xml.dom`.**InvalidCharacterErr**
当某个字符串参数包含的字符在使用它的上下文中不被 XML 1.0 标准建议所允许时引发。例如，尝试创建一个元素类型名称中带有空格的 `Element` 节点将导致此错误被引发。

exception xml.dom.**InvalidModificationErr**
当尝试修改某个节点的类型时被引发。

exception xml.dom.**InvalidStateErr**
当尝试使用未定义或不再可用的的对象时被引发。

exception xml.dom.**NamespaceErr**
如果试图以 XML 中的命名空间 建议所不允许的方式修改任何对象，则会引发此异常。

exception xml.dom.**NotFoundErr**
当某个节点不存在于被引用的上下文中时引发的异常。例如，`NamedNodeMap.removeNamedItem()` 将在所传入的节点不在于于映射中时引发此异常。

exception xml.dom.**NotSupportedErr**
当具体实现不支持所请求的对象类型或操作时被引发。

exception xml.dom.**NoDataAllowedErr**
当为某个不支持数据的节点指定数据时被引发。

exception xml.dom.**NoModificationAllowedErr**
当尝试修改某个不允许修改的对象（例如只读节点）时被引发。

exception xml.dom.**SyntaxErr**
当指定了无效或非法的字符串时被引发。

exception xml.dom.**WrongDocumentErr**
当将某个节点插入非其当前所属的另一个文档，并且具体实现不支持从一个文档向一个文档迁移节点时被引发。

DOM 建议映射中针对上述异常而定义的异常代码如下表所示:

常量	异常
DOMSTRING_SIZE_ERR	*DomstringSizeErr*
HIERARCHY_REQUEST_ERR	*HierarchyRequestErr*
INDEX_SIZE_ERR	*IndexSizeErr*
INUSE_ATTRIBUTE_ERR	*InuseAttributeErr*
INVALID_ACCESS_ERR	*InvalidAccessErr*
INVALID_CHARACTER_ERR	*InvalidCharacterErr*
INVALID_MODIFICATION_ERR	*InvalidModificationErr*
INVALID_STATE_ERR	*InvalidStateErr*
NAMESPACE_ERR	*NamespaceErr*
NOT_FOUND_ERR	*NotFoundErr*
NOT_SUPPORTED_ERR	*NotSupportedErr*
NO_DATA_ALLOWED_ERR	*NoDataAllowedErr*
NO_MODIFICATION_ALLOWED_ERR	*NoModificationAllowedErr*
SYNTAX_ERR	*SyntaxErr*
WRONG_DOCUMENT_ERR	*WrongDocumentErr*

20.6.3 一致性

本节描述了 Python DOM API、W3C DOM 建议以及 Python 的 OMG IDL 映射之间的一致性要求和关系。

类型映射

将根据下表，将 DOM 规范中使用的 IDL 类型映射为 Python 类型。

IDL 类型	Python Type
`boolean`	`bool` 或 `int`
`int`	`int`
`long int`	`int`
`unsigned int`	`int`
`DOMString`	`str` 或 `bytes`
`null`	`None`

访问器方法

从 OMG IDL 到 Python 的映射以类似于 Java 映射的方式定义了针对 IDL `attribute` 声明的访问器函数。映射以下 IDL 声明

```
readonly attribute string someValue;
         attribute string anotherValue;
```

会产生三个访问器函数: `someValue` 的"get" 方法 (`_get_someValue()`)，以及 `anotherValue` 的"get" 和"set" 方法 (`_get_anotherValue()` 和 `_set_anotherValue()`)。特别地，该映射不要求 IDL 属性像普通 Python 属性那样可访问: `object.someValue` 并非必须可用，并可能引发*AttributeError*。

但是，Python DOM API 则 确实要求普通属性访问可用。这意味着由 Python IDL 解译器生成的典型代理有可能会不可用，如果 DOM 对象是通过 CORBA 来访问则在客户端可能需要有包装对象。虽然这确实要求为 CORBA DOM 客户端进行额外的考虑，但具有从 Python 通过 CORBA 使用 DOM 经验的实现并不会认为这是个问题。已经声明了 `readonly` 的属性不必在所有 DOM 实现中限制写入访问。

在 Python DOM API 中，访问器函数不是必须的。如果提供，则它们应当采用由 Python IDL 映射所定义的形式，但这些方法会被认为不必要，因为这些属性可以从 Python 直接访问。永远都不要为 `readonly` 属性提供"set" 访问器。

IDL 定义没有完全体现 W3C DOM API 的要求，如特定对象的概念，又如 `getElementsByTagName()` 的返回值为"live" 等。Python DOM API 并不强制具体实现执行这些要求。

20.7 `xml.dom.minidom` --- 最小化的 DOM 实现

源代码: Lib/xml/dom/minidom.py

---

xml.dom.minidom 是文档对象模型接口的最小化实现，具有与其他语言类似的 API。它的目标是比完整 DOM 更简单并且更为小巧。对于 DOM 还不十分熟悉的用户则应当考虑改用*xml.etree.ElementTree* 模块来进行 XML 处理。

警告： *xml.dom.minidom* 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

DOM 应用程序通常会从将某个 XML 解析为 DOM 开始。使用*xml.dom.minidom* 时，这是通过各种解析函数来完成的:

```
from xml.dom.minidom import parse, parseString

dom1 = parse('c:\\temp\\mydata.xml')  # parse an XML file by name

datasource = open('c:\\temp\\mydata.xml')
dom2 = parse(datasource)  # parse an open file

dom3 = parseString('<myxml>Some data<empty/> some more data</myxml>')
```

parse() 函数可接受一个文件名或者打开的文件对象。

xml.dom.minidom.**parse**(*filename_or_file*, *parser=None*, *bufsize=None*)
根据给定的输入返回一个 Document。*filename_or_file* 可以是一个文件名，或是一个文件类对象。如果给定 *parser* 则它必须是一个 SAX2 解析器对象。此函数将修改解析器的处理程序并激活命名空间支持；其他解析器配置（例如设置一个实体求解器）必须已经提前完成。

如果你将 XML 存放为字符串形式，则可以改用*parseString()* 函数:

xml.dom.minidom.**parseString**(*string*, *parser=None*)
返回一个代表 *string* 的 Document。此方法会为指定字符串创建一个*io.StringIO* 对象并将其传递给*parse()*。

两个函数均返回一个代表文档内容的 Document 对象。object representing the content of the document.

parse() 和*parseString()* 函数所做的是将 XML 解析器连接到一个"DOM 构建器"，它可以从任意 SAX 解析器接收解析事件并将其转换为 DOM 树结构。这两个函数的名称可能有些误导性，但在学习此接口时是很容易掌握的。文档解析操作将在这两个函数返回之前完成；简单地说这两个函数本身并不提供解析器实现。

你也可以通过在一个"DOM 实现" 对象上调用方法来创建 Document。此对象可通过调用*xml.dom* 包或者*xml.dom.minidom* 模块中的 getDOMImplementation() 函数来获取。一旦你获得了一个 Document，你就可以向它添加子节点来填充 DOM:

```
from xml.dom.minidom import getDOMImplementation

impl = getDOMImplementation()

newdoc = impl.createDocument(None, "some_tag", None)
top_element = newdoc.documentElement
text = newdoc.createTextNode('Some textual content.')
top_element.appendChild(text)
```

一旦你得到了 DOM 文档对象，你就可以通过其属性和方法访问对应 XML 文档的各个部分。这些属性定义在 DOM 规格说明当中；文档对象的主要特征属性是 documentElement。它给出了 XML 文档中的主元素：即包含了所有其他元素的元素。以下是一个示例程序:

```
dom3 = parseString("<myxml>Some data</myxml>")
assert dom3.documentElement.tagName == "myxml"
```

当你完成对一个 DOM 树的处理时，你可以选择调用 unlink() 方法以鼓励尽早清除不再需要的对象。unlink() 是*xml.dom.minidom* 针对 DOM API 的专属扩展，它会将特定节点及其下级标记为不再有用。在其他情况下，Python 的垃圾回收器将负责最终处理树结构中的对象。

也参考:

文档对象模型 (DOM) 第 1 层级规格说明 被*xml.dom.minidom*所支持的 W3C 针对 DOM 的建议。

20.7.1 DOM 对象

Python 的 DOM API 定义被作为*xml.dom* 模块文档的一部分给出。这一节列出了该 API 和*xml.dom.minidom* 之间的差异。

Node.**unlink**()
破坏 DOM 的内部引用以便它能在没有循环 GC 的 Python 版本上垃圾回收器回收。即使在循环 GC 可用的时候，使用此方法也可让大量内存更快变为可用，因此当 DOM 对象不再被需要时尽早调用它们的这个方法是很好的做法。此方法只须在 Document 对象上调用，但也可以在下级节点上调用以丢弃该节点的下级节点。

你可以通过使用 with 语句来避免显式调用此方法。以下代码会在 with 代码块退出时自动取消链接 *dom*:

```
with xml.dom.minidom.parse(datasource) as dom:
    ... # Work with dom.
```

Node.**writexml**(*writer*, *indent=""*, *addindent=""*, *newl=""*, *encoding=None*, *standalone=None*)
将 XML 写入到写入器对象。写入器接受文本而非字节串作为输入，它应当具有与文件对象接口相匹配的 write() 方法。*indent* 形参是当前节点的缩进层级。*addindent* 形参是用于当前节点的下级节点的缩进量。*newl* 形参指定用于一行结束的字符串。

对于 Document 节点，可以使用附加的关键字参数 *encoding* 来指定 XML 标头的编码格式字段。

类似地，显式指明 *standalone* 参数将会使单独的文档声明被添加到 XML 文档的开头部分。如果将该值设为 *True*，则会添加 *standalone="yes"*，否则将为 *"no"*。未指明该参数将使文档声明被省略。

3.8 版更變: *writexml()* 方法现在会保留用户指定的属性顺序。

3.9 版更變: The *standalone* parameter was added.

Node.**toxml**(*encoding=None*, *standalone=None*)
返回一个包含 XML DOM 节点所代表的 XML 的字符串或字节串。

带有显式的 *encoding*[1] 参数时，结果为使用指定编码格式的字节串。没有 *encoding* 参数时，结果为 Unicode 字符串，并且结果字符串中的 XML 声明将不指定编码格式。使用 UTF-8 以外的编码格式对此字符串进行编码通常是不正确的，因为 UTF-8 是 XML 的默认编码格式。

standalone 参数的行为与*writexml()* 中的完全一致。

3.8 版更變: *toxml()* 方法现在会保留用户指定的属性顺序。

3.9 版更變: The *standalone* parameter was added.

Node.**toprettyxml**(*indent="\t"*, *newl="\n"*, *encoding=None*, *standalone=None*)
返回文档的美化打印版本。*indent* 指定缩进字符串并默认为制表符；*newl* 指定标示每行结束的字符串并默认为 \n。

encoding 参数的行为类似于*toxml()* 的对应参数。

standalone 参数的行为与*writexml()* 中的完全一致。

3.8 版更變: *toprettyxml()* 方法现在会保留用户指定的属性顺序。

3.9 版更變: The *standalone* parameter was added.

[1] 包括在 XML 输出中的编码格式名称应当遵循适当的标准。例如，"UTF-8" 是有效的，但"UTF8" 在 XML 文档的声明中是无效的，即使 Python 接受其作为编码格式名称。详情参见 https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 和 https://www.iana.org/assignments/character-sets/character-sets.xhtml。

20.7.2 DOM 示例

此示例程序是个相当实际的简单程序示例。在这个特定情况中，我们没有过多地利用 DOM 的灵活性。

```
import xml.dom.minidom

document = """\
<slideshow>
<title>Demo slideshow</title>
<slide><title>Slide title</title>
<point>This is a demo</point>
<point>Of a program for processing slides</point>
</slide>

<slide><title>Another demo slide</title>
<point>It is important</point>
<point>To have more than</point>
<point>one slide</point>
</slide>
</slideshow>
"""

dom = xml.dom.minidom.parseString(document)

def getText(nodelist):
    rc = []
    for node in nodelist:
        if node.nodeType == node.TEXT_NODE:
            rc.append(node.data)
    return ''.join(rc)

def handleSlideshow(slideshow):
    print("<html>")
    handleSlideshowTitle(slideshow.getElementsByTagName("title")[0])
    slides = slideshow.getElementsByTagName("slide")
    handleToc(slides)
    handleSlides(slides)
    print("</html>")

def handleSlides(slides):
    for slide in slides:
        handleSlide(slide)

def handleSlide(slide):
    handleSlideTitle(slide.getElementsByTagName("title")[0])
    handlePoints(slide.getElementsByTagName("point"))

def handleSlideshowTitle(title):
    print("<title>%s</title>" % getText(title.childNodes))

def handleSlideTitle(title):
    print("<h2>%s</h2>" % getText(title.childNodes))

def handlePoints(points):
    print("<ul>")
    for point in points:
        handlePoint(point)
    print("</ul>")
```

(下页继续)

(繼續上一頁)

```
def handlePoint(point):
    print("<li>%s</li>" % getText(point.childNodes))

def handleToc(slides):
    for slide in slides:
        title = slide.getElementsByTagName("title")[0]
        print("<p>%s</p>" % getText(title.childNodes))

handleSlideshow(dom)
```

20.7.3 minidom 和 DOM 标准

xml.dom.minidom 模块实际上是兼容 DOM 1.0 的 DOM 并带有部分 DOM 2 特性（主要是命名空间特性）。

Python 中 DOM 接口的用法十分直观。会应用下列映射规则:

- 接口是通过实例对象来访问的。应用程序不应实例化这些类本身；它们应当使用 `Document` 对象提供的创建器函数。派生的接口支持上级接口的所有操作（和属性），并添加了新的操作。
- 操作以方法的形式使用。因由 DOM 只使用 `in` 形参，参数是以正常顺序传入的（从左至右）。不存在可选参数。`void` 操作返回 `None`。
- IDL 属性会映射到实例属性。为了兼容针对 Python 的 OMG IDL 语言映射，属性 `foo` 也可通过访问器方法 `_get_foo()` 和 `_set_foo()` 来访问。`readonly` 属性不可被修改；运行时并不强制要求这一点。
- `short int`, `unsigned int`, `unsigned long long` 和 `boolean` 类型都会映射为 Python 整数类型。
- `DOMString` 类型会映射为 Python 字符串。*xml.dom.minidom* 支持字节串或字符串，但通常是产生字符串。`DOMString` 类型的值也可以为 `None`，W3C 的 DOM 规格说明允许其具有 IDL `null` 值。
- `const` 声明会映射为它们各自的作用域内的变量 (例如 `xml.dom.minidom.Node.PROCESSING_INSTRUCTION_NODE`)；它们不可被修改。
- `DOMException` 目前不被*xml.dom.minidom* 所支持。*xml.dom.minidom* 会改为使用标准 Python 异常例如*TypeError* 和*AttributeError*。
- `NodeList` 对象是使用 Python 内置列表类型来实现的。这些对象提供了 DOM 规格说明中定义的接口，但在较早版本的 Python 中它们不支持官方 API。相比在 W3C 建议中定义的接口，它们要更加的”Pythonic”。

下列接口未在*xml.dom.minidom* 中实现:

- `DOMTimeStamp`
- `EntityReference`

这些接口所反映的 XML 文档信息对于大多数 DOM 用户来说没有什么帮助。

F解

20.8 xml.dom.pulldom --- 支持构建部分 DOM 树

源代码: Lib/xml/dom/pulldom.py

---

xml.dom.pulldom 模块提供了一个“拉取解析器”，它能在必要时被用于产生文件的可访问 DOM 的片段。其基本概念包括从输入的 XML 流拉取“事件”并处理它们。与同样地同时应用了事件驱动处理模型加回调函数的 SAX 不同，拉取解析器的用户要负责显式地从流拉取事件，并循环遍历这些事件直到处理结束或者发生了错误条件。

警告: *xml.dom.pulldom* 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

3.7.1 版更變: SAX 解析器默认不再处理一般外部实体以提升在默认情况下的安全性。要启用外部实体处理，请传入一个自定义的解析器实例:

```
from xml.dom.pulldom import parse
from xml.sax import make_parser
from xml.sax.handler import feature_external_ges

parser = make_parser()
parser.setFeature(feature_external_ges, True)
parse(filename, parser=parser)
```

示例:

```
from xml.dom import pulldom

doc = pulldom.parse('sales_items.xml')
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'item':
        if int(node.getAttribute('price')) > 50:
            doc.expandNode(node)
            print(node.toxml())
```

event 是一个常量，可以取下列值之一:

- START_ELEMENT
- END_ELEMENT
- COMMENT
- START_DOCUMENT
- END_DOCUMENT
- CHARACTERS
- PROCESSING_INSTRUCTION
- IGNORABLE_WHITESPACE

node 是一个 xml.dom.minidom.Document, xml.dom.minidom.Element 或 xml.dom.minidom.Text 类型的对象。

由于文档是被当作“展平”的事件流来处理的，文档“树”会被隐式地遍历并且无论所需元素在树中的深度如何都会被找到。换句话说，不需要考虑层级问题，例如文档节点的递归搜索等，但是如果元素的内容很重要，则有必要保留一些上下文相关的状态（例如记住任意给定点在文档中的位置）或者使用*DOMEventStream.expandNode()* 方法并切换到 DOM 相关的处理过程。

class xml.dom.pulldom.**PullDom**(*documentFactory=None*)
xml.sax.handler.ContentHandler 的子类。

class xml.dom.pulldom.**SAX2DOM**(*documentFactory=None*)
xml.sax.handler.ContentHandler 的子类。

xml.dom.pulldom.**parse**(*stream_or_string*, *parser=None*, *bufsize=None*)
基于给定的输入返回一个*DOMEventStream*。*stream_or_string* 可以是一个文件名，或是一个文件类对象。*parser* 如果给出，则必须是一个*XMLReader* 对象。此函数将改变解析器的文档处理程序并激活命名空间支持；其他解析器配置（例如设置实体解析器）必须在之前已完成。

如果你将 XML 存放为字符串，则可以改用*parseString()* 函数:

xml.dom.pulldom.**parseString**(*string*, *parser=None*)
返回一个*DOMEventStream* 来表示 (Unicode) *string*。

xml.dom.pulldom.**default_bufsize**
将 *bufsize* 形参的默认值设为*parse()*。

此变量的值可在调用*parse()* 之前修改并使新值生效。

20.8.1 DOMEventStream 对象

class xml.dom.pulldom.**DOMEventStream**(*stream*, *parser*, *bufsize*)
3.8 版後已🄴用: 对 序列协议的支持已被弃用。

getEvent()
返回一个元组，其中包含 *event* 和 xml.dom.minidom.Document 形式的当前 *node* 如果 event 等于 START_DOCUMENT，包含 xml.dom.minidom.Element 如果 event 等于 START_ELEMENT 或 END_ELEMENT 或者 xml.dom.minidom.Text 如果 event 等于 CHARACTERS。当前 node 不包含有关其子节点的信息，除非*expandNode()* 被调用。

expandNode(*node*)
将 *node* 的所有子节点扩展到 *node* 中。例如:

```
from xml.dom import pulldom

xml = '<html><title>Foo</title> <p>Some text <div>and more</div></p> </html>'
doc = pulldom.parseString(xml)
for event, node in doc:
    if event == pulldom.START_ELEMENT and node.tagName == 'p':
        # Following statement only prints '<p/>'
        print(node.toxml())
        doc.expandNode(node)
        # Following statement prints node with all its children '<p>Some text
↪<div>and more</div></p>'
        print(node.toxml())
```

reset()

20.9 xml.sax --- 支持 SAX2 解析器

源代码: Lib/xml/sax/__init__.py

---

xml.sax 包提供多个模块，它们在 Python 上实现了用于 XML (SAX) 接口的简单 API。这个包本身为 SAX API 用户提供了一些最常用的 SAX 异常和便捷函数。

> **警告:** *xml.sax* 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

3.7.1 版更變: SAX 解析器默认不会再处理通用外部实体以便提升安全性。在此之前，解析器会创建网络连接来获取远程文件或是从 DTD 和实体文件系统中加载本地文件。此特性可通过在解析器对象上调用*setFeature()* 对象并传入参数*feature_external_ges* 来重新启用。

可用的便捷函数如下所列:

xml.sax.**make_parser**(*parser_list=[]*)
创建并返回一个 SAX *XMLReader* 对象。将返回第一个被找到的解析器。如果提供了 *parser_list*，它必须为一个包含字符串的可迭代对象，这些字符串指定了具有名为 `create_parser()` 函数的模块。在 *parser_list* 中列出的模块将在默认解析器列表中的模块之前被使用。

3.8 版更變: *parser_list* 参数可以是任意可迭代对象，而不一定是列表。

xml.sax.**parse**(*filename_or_stream*, *handler*, *error_handler=handler.ErrorHandler()*)
创建一个 SAX 解析器并用它来解析文档。用于传入文档的 *filename_or_stream* 可以是一个文件名或文件对象。*handler* 形参必须是一个 SAX *ContentHandler* 实例。如果给出了 *error_handler*，则它必须是一个 SAX *ErrorHandler* 实例；如果省略，则对于任何错误都将引发*SAXParseException*。此函数没有返回值；所有操作必须由传入的 *handler* 来完成。

xml.sax.**parseString**(*string*, *handler*, *error_handler=handler.ErrorHandler()*)
类似于*parse()*，但解析对象是作为形参传入的缓冲区 *string*。*string* 必须为*str* 实例或者*bytes-like object*。

3.5 版更變: 增加了对*str* 实例的支持。

典型的 SAX 应用程序会使用三种对象：读取器、处理句柄和输入源。“读取器”在此上下文中与解析器同义，即某个从输入源读取字节或字符，并产生事件序列的代码段。事件随后将被分发给处理句柄对象，即由读取器发起调用处理句柄上的某个方法。因此 SAX 应用程序必须获取一个读取器对象，创建或打开输入源，创建处理句柄，并一起连接到这些对象。作为准备工作的最后一步，将调用读取器来解析输入内容。在解析过程中，会根据来自输入数据的结构化和语义化事件来调用处理句柄对象上的方法。

就这些对象而言，只有接口部分是需要关注的；它们通常不是由应用程序本身来实例化。由于 Python 没有显式的接口标记法，它们的正式引入形式是类，但应用程序可能会使用并非从已提供的类继承而来的实现。*InputSource*, *Locator*, Attributes, AttributesNS 以及*XMLReader* 接口是在*xml.sax.xmlreader* 模块中定义的。处理句柄接口是在*xml.sax.handler* 中定义的。为了方便起见，*InputSource* (它往往会被直接实例化) 和处理句柄类也可以从*xml.sax* 获得。这些接口的描述见下文。

除了这些类，*xml.sax* 还提供了如下异常类。

exception xml.sax.**SAXException**(*msg*, *exception=None*)
封装某个 XML 错误或警告。这个类可以包含来自 XML 解析器或应用程序的基本错误或警告信息：它可以被子类化以提供额外的功能或是添加本地化信息。请注意虽然在*ErrorHandler* 接口中定义的处理句柄可以接收该异常的实例，但是并不要求实际引发该异常 --- 它也可以被用作信息的容器。

当实例化时，*msg* 应当是适合人类阅读的错误描述。如果给出了可选的 *exception* 形参，它应当为 `None` 或者解析代码所捕获的异常并会被作为信息传递出去。

这是其他 SAX 异常类的基类。

exception xml.sax.**SAXParseException**(*msg*, *exception*, *locator*)
SAXException 的子类，针对解析错误引发。这个类的实例会被传递给 SAX *ErrorHandler* 接口的方法来提供关于解析错误的信息。这个类支持 SAX *Locator* 接口以及*SAXException* 接口。

exception xml.sax.**SAXNotRecognizedException**(*msg*, *exception=None*)
SAXException 的子类，当 SAX *XMLReader* 遇到不可识别的特性或属性时引发。SAX 应用程序和扩展可能会出于类似目的而使用这个类。

exception xml.sax.**SAXNotSupportedException**(*msg*, *exception=None*)
SAXException 的子类，当 SAX *XMLReader* 被要求启用某个不受支持的特性，或者将某个属性设为具体实现不支持的值时引发。SAX 应用程序和扩展可能会出于类似目的而使用这个类。

也参考:

SAX: The Simple API for XML 这个网站是 SAX API 定义的焦点。它提供了一个 Java 实现以及在线文档。还包括其他实现的链接和历史信息。

***xml.sax.handler* 模块** 应用程序所提供对象的接口定义。

***xml.sax.saxutils* 模块** 可在 SAX 应用程序中使用的便捷函数。

***xml.sax.xmlreader* 模块** 解析器所提供对象的接口定义。

20.9.1 SAXException 对象

SAXException 异常类支持下列方法:

SAXException.**getMessage**()
返回描述错误条件的适合人类阅读的消息。

SAXException.**getException**()
返回一个封装的异常对象或者 None。

20.10 xml.sax.handler --- SAX 处理句柄的基类

源代码: Lib/xml/sax/handler.py

SAX API 定类了四种处理句柄：内容句柄、DTD 句柄、错误句柄以及实体解析器。应用程序通常只需要实现他们感兴趣的事件对应的接口；他们可以在单个对象或多个对象中实现这些接口。处理句柄的实现应当继承自在*xml.sax.handler* 模块中提供的基类，以便所有方法都能获得默认的实现。

class xml.sax.handler.**ContentHandler**
这是 SAX 中的主回调接口，也是对应用程序来说最重要的一个接口。此接口中事件的顺序反映了文档中信息的顺序。

class xml.sax.handler.**DTDHandler**
处理 DTD 事件。

这个接口仅指定了基本解析（未解析的实体和属性）所需的那些 DTD 事件。

class xml.sax.handler.**EntityResolver**
用于解析实体的基本接口。如果你创建了实现此接口的对象，然后用你的解析器注册该对象，该解析器将调用你的对象中的方法来解析所有外部实体。

class xml.sax.handler.**ErrorHandler**
解析器用来向应用程序表示错误和警告的接口。这个对象的方法控制错误是要立即转换为异常还是以某种其他该来来处理。

除了这些类，*xml.sax.handler* 还提供了表示特性和属性名称的符号常量。

xml.sax.handler.**feature_namespaces**

值: "http://xml.org/sax/features/namespaces"
true: 执行命名空间处理。
false: 可选择不执行命名空间处理 (这意味着 namespace-prefixes; default)。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_namespace_prefixes**

值: "http://xml.org/sax/features/namespace-prefixes"
true: 报告原始的带前缀名称和用于命名空间声明的属性。
false: 不报告用于命名空间声明的属性，可选择不报告原始的带前缀名称（默认）。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_string_interning**

值: "http://xml.org/sax/features/string-interning"
true: 所有元素名称、前缀、属性名称、命名空间 URI 以及本地名称都使用内置的 intern 函数进行内化。
false: 名称不要求被内化，但也可以被内化（默认）。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_validation**

值: "http://xml.org/sax/features/validation"
true: 报告所有的验证错误（包括 external-general-entities 和 external-parameter-entities）。
false: 不报告验证错误。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_external_ges**

值: "http://xml.org/sax/features/external-general-entities"
true: 包括所有的外部通用（文本）实体。
false: 不包括外部通用实体。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**feature_external_pes**

值: "http://xml.org/sax/features/external-parameter-entities"
true: 包括所有的外部参数实体，也包括外部 DTD 子集。
false: 不包括任何外部参数实体，也不包括外部 DTD 子集。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**all_features**
全部特性列表。

xml.sax.handler.**property_lexical_handler**

值: "http://xml.org/sax/properties/lexical-handler"
数据类型: xml.sax.sax2lib.LexicalHandler (在 Python 2 中不受支持)
描述: 可选的扩展处理句柄，用于注释等词法事件。
访问: 读/写

xml.sax.handler.**property_declaration_handler**

值: "http://xml.org/sax/properties/declaration-handler"
数据类型: xml.sax.sax2lib.DeclHandler (在 Python 2 中不受支持)
描述: 可选的扩展处理句柄，用于标注和未解析实体以外的 DTD 相关事件。
访问: 读/写

xml.sax.handler.**property_dom_node**

值: "http://xml.org/sax/properties/dom-node"
数据类型: org.w3c.dom.Node (在 Python 2 中不受支持)
描述: 在解析时，如果这是一个 DOM 迭代器则为当前被访问的 DOM 节点；不在解析时，则将根 DOM 节点用于迭代。
access: (解析) 只读; (不解析) 读/写

xml.sax.handler.**property_xml_string**

值: "http://xml.org/sax/properties/xml-string"
data type: Bytes
描述: 作为当前事件来源的字符串字面值。
访问: 只读

xml.sax.handler.**all_properties**
已知属性名称列表。

20.10.1 ContentHandler 对象

用户应当子类化*ContentHandler* 来支持他们的应用程序。以下方法会由解析器在输入文档的适当事件上调用:

ContentHandler.**setDocumentLocator**(*locator*)
由解析器调用来给予应用程序一个定位器以确定文档事件来自何处。

强烈建议（虽然不是绝对的要求）SAX 解析器提供一个定位器：如果提供的话，它必须在发起调用 DocumentHandler 接口的任何其他方法之前通过发起调用此方法来提供定位器。

定位器允许应用程序确定任何文档相关事件的结束位置，即使解析器没有报告错误。通常，应用程序将使用这些信息来报告它自己的错误（例如未匹配到应用程序业务规则的字符内容）。定位器所返回的信息可能不足以与搜索引擎配合使用。

请注意定位器只有在发起调用此接口中的事件时才会返回正确的信息。应用程序不应试图在其他任何时刻使用它。

ContentHandler.**startDocument**()
接收一个文档开始的通知。

SAX 解析器将只发起调用这个方法一次，并且会在调用这个接口或 DTDHandler 中的任何其他方法之前 (*setDocumentLocator()* 除外)。

ContentHandler.**endDocument**()
接收一个文档结束的通知。

SAX 解析器将只发起调用这个方法一次，并且它将是在解析过程中最后发起调用的方法。解析器在（因不可恢复的错误）放弃解析或到达输入的终点之前不应发起调用这个方法。

ContentHandler.**startPrefixMapping**(*prefix*, *uri*)
开始一个前缀 URI 命名空间映射的范围。

来自此事件的信息对于一般命名空间处理来说是不必要的：当 `feature_namespaces` 特性被启用时（默认）SAX XML 读取器将自动为元素和属性名称替换前缀。

但是也存在一些情况，当应用程序需要在字符数据或属性值中使用前缀，而它们无法被安全地自动扩展；*startPrefixMapping()* 和*endPrefixMapping()* 事件会向应用程序提供信息以便在这些上下文内部扩展前缀，如果有必要的话。

请注意*startPrefixMapping()* 和*endPrefixMapping()* 事件并不保证能够相对彼此被正确地嵌套：所有*startPrefixMapping()* 事件都将在对应的*startElement()* 事件之前发生，而所有*endPrefixMapping()* 事件都将在对应的*endElement()* 事件之后发生，但它们的并不保证一致。

ContentHandler.**endPrefixMapping**(*prefix*)
结束一个前缀 URI 映射的范围。

请参看*startPrefixMapping()* 了解详情。此事件将总是会在对应的*endElement()* 事件之后发生，但*endPrefixMapping()* 事件的顺序则并没有保证。

ContentHandler.**startElement**(*name*, *attrs*)
在非命令空间模式下指示一个元素的开始。

name 形参包含字符串形式的元素类型原始 XML 1.0 名称而 *attrs* 形参存放包含元素属性的 `Attributes` 接口对象 (参见*Attributes 接口*)。作为 *attrs* 传入的对象可能被解析器所重用；维持一个对它的引用不是保持属性副本的可靠方式。要保持这些属性的一个副本，请使用 *attrs* 对象的*copy()* 方法。

ContentHandler.**endElement**(*name*)
在非命名空间模式下指示一个元素的结束。

name 形参包含元素类型的名称，与*startElement()* 事件的一样。

ContentHandler.**startElementNS**(*name*, *qname*, *attrs*)
在命名空间模式下指示一个元素的开始。

name 形参包含以 `(uri, localname)` 元组表示的元素类型名称，*qname* 形参包含源文档中使用的原始 XML 1.0 名称，而 *attrs* 形参存放包含元素属性的 `AttributesNS` 接口实例 (参见*AttributesNS 接口*)。如果没有命名空间被关联到元素，则 *name* 的 *uri* 部分将为 `None`。作为 *attrs* 传入的对象可能被解析器所重用；维持一个对它的引用不是保持属性副本的可靠方式。要保持这些属性的一个副本，请使用 *attrs* 对象的*copy()* 方法。

解析器可将 *qname* 形参设为 `None`，除非 `feature_namespace_prefixes` 特性已被激活。

ContentHandler.**endElementNS**(*name*, *qname*)
在命名空间模式下指示一个元素的结束。

name 形参包含元素类型的名称，与*startElementNS()* 方法的一样，*qname* 形参也是类似的。

ContentHandler.**characters**(*content*)
接收字符数据的通知。

解析器将调用此方法来报告每一个字符数据分块。SAX 解析器可以将所有连续字符数据返回为一个单独分块，或者将其拆成几个分块；但是，在任意单个事件中的所有字符都必须来自同一个外部实体以便定位器提供有用的信息。

content 可以是一个字符串或字节串实例；`expat` 读取器模块总是会产生字符串。

> **備𝔽:** Python XML 特别关注小组所提供的早期 SAX 1 接口针对此方法使用了一个更类似于 Java 的接口。由于 Python 所使用的大多数解析器都没有利用老式的接口，因而选择了更简单的签名来替代它。要将旧代码转换为新接口，请使用 *content* 而不要通过旧的 *offset* 和 *length* 形参来对内容进行切片。

ContentHandler.**ignorableWhitespace**(*whitespace*)
接收元素内容中可忽略空白符的通知。

验证解析器必须使用此方法来报告每个可忽略的空白符分块（参见 W3C XML 1.0 建议第 2.10 节）：非验证解析器如果能够解析并使用内容模型的话也可以使用此方法。

SAX 解析器可以将所有连续字符数据返回为一个单独分块，或者将其拆成几个分块；但是，在任意单个事件中的所有字符都必须来自同一个外部实体以便定位器提供有用的信息。

ContentHandler.**processingInstruction**(*target*, *data*)
接受一条处理指令的通知。

解析器将为已找到的每条处理指令发起调用该方法一次：请注意处理指令可能出现在主文档元素之前或之后。

SAX 解析器绝不应当使用此方法来报告 XML 声明（XML 1.0 第 2.8 节）或文本声明（XML 1.0 第 4.3.1 节）。

ContentHandler.**skippedEntity**(*name*)
接收一个已跳过实体的通知。

解析器将为每个已跳过实体发起调用此方法一次。非验证处理程序可能会跳过未看到声明的实体（例如，由于实体是在一个外部 because, for example, the entity was declared in an external DTD 子集中声明的）。所有处理程序都可以跳过外部实体，具体取决于 `feature_external_ges` 和 `feature_external_pes` 属性的值。

20.10.2 DTDHandler 对象

DTDHandler 实例提供了下列方法:

DTDHandler.**notationDecl**(*name*, *publicId*, *systemId*)
处理标注声明事件。

DTDHandler.**unparsedEntityDecl**(*name*, *publicId*, *systemId*, *ndata*)
处理未解析的实体声明事件。

20.10.3 EntityResolver 对象

EntityResolver.**resolveEntity**(*publicId*, *systemId*)
求解一个实体的系统标识符并返回一个字符串形式的系统标识符作为读取源，或是一个 InputSource 作为读取源。默认的实现会返回 *systemId*。

20.10.4 ErrorHandler 对象

带有这个接口的对象被用于接收来自*XMLReader*的错误和警告信息。如果你创建了一个实现这个接口的对象，然后用你的*XMLReader*注册这个对象，则解析器将调用你的对象中的这个方法来报告所有的警告和错误。有三个可用的错误级别：警告、（或许）可恢复的错误和不可恢复的错误。所有方法都接受SAXParseException作为唯一的形参。错误和警告可以通过引发所传入的异常对象来转换为异常。

ErrorHandler.**error**(*exception*)
当解析器遇到一个可恢复的错误时调用。如果此方法没有引发异常，则解析可能会继续，但是应用程序不能预期获得更多的文档信息。允许解析器继续可能会允许在输入文档中发现额外的错误。

ErrorHandler.**fatalError**(*exception*)
当解析器遇到一个不可恢复的错误时调用；在此方法返回时解析应当终止。

ErrorHandler.**warning**(*exception*)
当解析器向应用程序提供次要警告信息时调用。在此方法返回时解析应当继续，并且文档信息将继续被传递给应用程序。在此方法中引发异常将导致解析结束。

20.11 xml.sax.saxutils --- SAX 工具集

源代码: Lib/xml/sax/saxutils.py

---

*xml.sax.saxutils*模块包含一些在创建 SAX 应用程序时十分有用的类和函数，它们可以被直接使用，或者是作为基类使用。

xml.sax.saxutils.**escape**(*data*, *entities={}*)
对数据字符串中的'&'，'<'和'>'进行转义。

你可以通过传入一个字典作为可选的*entities*形参来对其他字符串数据进行转义。字典的键和值必须为字符串；每个键将被替换为其所对应的值。字符'&'，'<'和'>'总是会被转义，即使提供了*entities*。

xml.sax.saxutils.**unescape**(*data*, *entities={}*)
对字符串数据中的'&'，'<'和'>'进行反转义。

你可以通过传入一个字典作为可选的*entities*形参来对其他数据字符串进行转义。字典的键和值必须都为字符串；每个键将被替换为所对应的值。'&amp'，'<'和'>'将总是保持不被转义，即使提供了*entities*。

xml.sax.saxutils.**quoteattr**(*data*, *entities={}*)
类似于*escape()*，但还会对*data*进行处理以将其用作属性值。返回值是*data*加上任何额外要求的替换的带引号版本。*quoteattr()*将基于*data*的内容选择一个引号字符，以尽量避免在字符串中编码任何引号字符。如果单双引号字符在*data*中都存在，则双引号字符将被编码并且*data*将使用双引号来标记。结果字符串可被直接用作属性值:

```
>>> print("<element attr=%s>" % quoteattr("ab ' cd \" ef"))
<element attr="ab ' cd &quot; ef">
```

此函数适用于为 HTML 或任何使用引用实体语法的 SGML 生成属性值。

class xml.sax.saxutils.**XMLGenerator**(*out=None*, *encoding='iso-8859-1'*, *short_empty_elements=False*)
这个类通过将 SAX 事件写回到 XML 文档来实现*ContentHandler*接口。换句话说，使用*XMLGenerator*作为内容处理程序将重新产生所解析的原始文档。*out*应当为一个文件类对象，它默认将为*sys.stdout*。*encoding*为输出流的编码格式，它默认将为'iso-8859-1'。*short_empty_elements*控制不包含内容的元素的格式化：如为False(默认值)则它们会以开始/结束标记对的形式被发送，如果设为True则它们会以单个自结束标记的形式被发送。

3.2 版新加入: *short_empty_elements* 形参。

class xml.sax.saxutils.**XMLFilterBase**(*base*)
这个类被设计用来分隔*XMLReader* 和客户端应用的事件处理程序。在默认情况下，它除了将请求传送给读取器并将事件传送给处理程序之外什么都不做，但其子类可以重载特定的方法以在传送它们的时候修改事件流或配置请求。

xml.sax.saxutils.**prepare_input_source**(*source*, *base=''*)
此函数接受一个输入源和一个可选的基准 URL 并返回一个经过完整解析可供读取的*InputSource*。输入源的给出形式可以为字符串、文件类对象或*InputSource* 对象；解析器将使用此函数来针对它们的 parse() 方法实现多态 *source* 参数。

20.12 xml.sax.xmlreader --- 用于 XML 解析器的接口

源代码: Lib/xml/sax/xmlreader.py

SAX 解析器实现了*XMLReader* 接口。它们是在一个 Python 模块中实现的，该模块必须提供一个 create_parser() 函数。该函数由*xml.sax.make_parser()* 不带参数地发起调用来创建新的解析器对象。

class xml.sax.xmlreader.**XMLReader**
可由 SAX 解析器继承的基类。

class xml.sax.xmlreader.**IncrementalParser**
在某些情况下，最好不要一次性地解析输入源，而是在可用的时候分块送入。请注意读取器通常不会读取整个文件，它同样也是分块读取的；并且 parse() 在处理完整个文档之前不会返回。所以如果不希望 parse() 出现阻塞行为则应当使用这些接口。

当解析器被实例化时它已准备好立即开始接受来自 feed 方法的数据。在通过调用 close 方法结束解析时 reset 方法也必须被调用以使解析器准备好接受新的数据，无论它是来自于 feed 还是使用 parse 方法。

请注意这些方法 不可在解析期间被调用，即在 parse 被调用之后及其返回之前。

默认情况下，该类还使用 IncrementalParser 接口的 feed, close 和 reset 方法来实现 XMLReader 接口的 parse 方法以方便 SAX 2.0 驱动的编写者。

class xml.sax.xmlreader.**Locator**
用于关联一个 SAX 事件与一个文档位置的接口。定位器对象只有在调用 DocumentHandler 的方法期间才会返回有效的结果；在其他任何时候，结果都是不可预测的。如果信息不可用，这些方法可能返回 None。

class xml.sax.xmlreader.**InputSource**(*system_id=None*)
XMLReader 读取实体所需信息的封装。

这个类可能包括了关于公有标识符、系统标识符、字节流（可能带有字符编码格式信息）和/或一个实体的字符流的信息。

应用程序将创建这个类的对象以便在*XMLReader.parse()* 方法中使用或是用于从 EntityResolver.resolveEntity 返回值。

InputSource 属于应用程序，*XMLReader* 不能修改从应用程序传递给它的*InputSource* 对象，但它可以创建副本并进行修改。

class xml.sax.xmlreader.**AttributesImpl**(*attrs*)
这是 Attributes 接口（参见*Attributes 接口* 一节）的具体实现。这是一个 startElement() 调用中的元素属性的字典类对象。除了最有用处的字典操作，它还支持接口所描述的一些其他方法。该类的对象应当由读取器来实例化；*attrs* 必须为包含从属性名到属性值的映射的字典类对象。

class xml.sax.xmlreader.**AttributesNSImpl**(*attrs*, *qnames*)
可感知命名空间的*AttributesImpl* 变体形式，它将被传递给 startElementNS()。它派生自*AttributesImpl*，但会将属性名称解读为 *namespaceURI* 和 *localname* 二元组。此外，它还提供了一些期望接收在原始文档中出现的限定名称的方法。这个类实现了 AttributesNS 接口（参见*AttributesNS 接口* 一节）。

20.12.1 XMLReader 对象

XMLReader 接口支持下列方法:

XMLReader.**parse**(*source*)
处理输入源，产生 SAX 事件。*source* 对象可以是一个系统标识符（标识输入源的字符串 -- 通常为文件名或 URL），*pathlib.Path* 或*路径类* 对象，或者是*InputSource* 对象。当*parse()* 返回时，输入会被全部处理完成，解析器对象可以被丢弃或重置。

3.5 版更變: 增加了对字符流的支持。

3.8 版更變: 增加了对路径类对象的支持。

XMLReader.**getContentHandler**()
返回当前的*ContentHandler*。

XMLReader.**setContentHandler**(*handler*)
设置当前的*ContentHandler*。如果没有设置*ContentHandler*，内容事件将被丢弃。

XMLReader.**getDTDHandler**()
返回当前的*DTDHandler*。

XMLReader.**setDTDHandler**(*handler*)
设置当前的*DTDHandler*。如果没有设置*DTDHandler*，DTD 事件将被德育。

XMLReader.**getEntityResolver**()
返回当前的*EntityResolver*。

XMLReader.**setEntityResolver**(*handler*)
设置当前的*EntityResolver*。如果没有设置*EntityResolver*，尝试解析一个外部实体将导致打开该实体的系统标识符，并且如果它不可用则操作将失败。

XMLReader.**getErrorHandler**()
返回当前的*ErrorHandler*。

XMLReader.**setErrorHandler**(*handler*)
设置当前的错误处理句柄。如果没有设置*ErrorHandler*，错误将作为异常被引发，并将打印警告信息。

XMLReader.**setLocale**(*locale*)
允许应用程序为错误和警告设置语言区域。

SAX 解析器不要求为错误和警告提供本地化信息；但是如果它们无法支持所请求的语言区域，则必须引发一个 SAX 异常。应用程序可以在解析的中途请求更改语言区域。

XMLReader.**getFeature**(*featurename*)
返回 *featurename* 特性的当前设置。如果特性无法被识别，则会引发 SAXNotRecognizedException。在*xml.sax.handler* 模块中列出了常见的特性名称。

XMLReader.**setFeature**(*featurename*, *value*)
将 *featurename* 设为 *value*。如果特性无法被识别，则会引发 SAXNotRecognizedException。如果特性或其设置不被解析器所支持，则会引发 *SAXNotSupportedException*。

XMLReader.**getProperty**(*propertyname*)
返回 *propertyname* 属性的当前设置。如果属性无法被识别，则会引发 SAXNotRecognizedException。在*xml.sax.handler* 模块中列出了常见的属性名称。

XMLReader.**setProperty**(*propertyname*, *value*)
将 *propertyname* 设为 *value*。如果属性无法被识别，则会引发 SAXNotRecognizedException。如果属性或其设置不被解析器所支持，则会引发 *SAXNotSupportedException*。

20.12.2 IncrementalParser 对象

IncrementalParser 的实例额外提供了下列方法:

IncrementalParser.**feed**(*data*)
处理 *data* 的一个分块。

IncrementalParser.**close**()
确定文档的结尾。这将检查只能在结尾处检查的格式是否良好的条件，发起调用处理程序，并可能会清理在解析期间分配的资源。

IncrementalParser.**reset**()
此方法会在调用 close 来重置解析器以便其准备好解析新的文档之后被调用。在 close 之后未调用 reset 即调用 parse 或 feed 的结果是未定义的。

20.12.3 Locator 对象

Locator 的实例提供了下列方法:

Locator.**getColumnNumber**()
返回当前事件开始位置的列号。

Locator.**getLineNumber**()
返回当前事件开始位置的行号。

Locator.**getPublicId**()
返回当前事件的公有标识符。

Locator.**getSystemId**()
返回当前事件的系统标识符。

20.12.4 InputSource 对象

InputSource.**setPublicId**(*id*)
设置该*InputSource* 的公有标识符。

InputSource.**getPublicId**()
返回此*InputSource* 的公有标识符。

InputSource.**setSystemId**(*id*)
设置此*InputSource* 的系统标识符。

InputSource.**getSystemId**()
返回此*InputSource* 的系统标识符。

InputSource.**setEncoding**(*encoding*)
设置此*InputSource* 的字符编码格式。

编码格式必须是 XML 编码声明可接受的字符串（参见 XML 建议规范第 4.3.3 节）。

如果*InputSource* 还包含一个字符流则*InputSource* 的 encoding 属性会被忽略。

InputSource.**getEncoding**()
获取此 InputSource 的字符编码格式。

InputSource.**setByteStream**(*bytefile*)
设置此输入源的字节流（为*binary file* 对象）。

如果还指定了一个字符流被则 SAX 解析器会忽略此设置，但它将优先使用字节流而不是自己打开一个 URI 连接。

如果应用程序知道字节流的字符编码格式，它应当使用 setEncoding 方法来设置它。

InputSource.**getByteStream**()
获取此输入源的字节流。

getEncoding 方法将返回该字节流的字符编码格式，如果未知则返回 None。

InputSource.**setCharacterStream**(*charfile*)
设置此输入源的字符流 (为*text file* 对象)。

如果指定了一个字符流，SAX 解析器将忽略任何字节流并且不会尝试打开一个指向系统标识符的 URI 连接。

InputSource.**getCharacterStream**()
获取此输入源的字符流。

20.12.5 Attributes 接口

Attributes 对象实现了一部分映射协议，包括 copy(), get(), __contains__(), items(), keys() 和 values() 等方法。还提供了下列方法:

Attributes.**getLength**()
返回属性的数量。

Attributes.**getNames**()
返回属性的名称。

Attributes.**getType**(*name*)
返回属性 *name* 的类型，通常为 'CDATA'。

Attributes.**getValue**(*name*)
返回属性 *name* 的值。

20.12.6 AttributesNS 接口

此接口是 Attributes 接口（参见*Attributes 接口* 章节）的一个子类型。那个接口所支持的所有方法在 AttributesNS 对象上也都可用。

下列方法也是可用的:

AttributesNS.**getValueByQName**(*name*)
返回一个限定名称的值。

AttributesNS.**getNameByQName**(*name*)
返回限定名称 *name* 的 (namespace, localname) 对。

AttributesNS.**getQNameByName**(*name*)
返回 (namespace, localname) 对的限定名称。

AttributesNS.**getQNames**()
返回所有属性的限定名称。

20.13 xml.parsers.expat --- 使用 Expat 的快速 XML 解析

警告： pyexpat 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

xml.parsers.expat 模块是针对 Expat 非验证 XML 解析器的 Python 接口。此模块提供了一个扩展类型 xmlparser，它代表一个 XML 解析器的当前状态。在创建一个 xmlparser 对象之后，该对象的各个属性可被设置为相应的处理句柄函数。随后当将一个 XML 文档送入解析器时，就会为该 XML 文档中的字符数据和标记调用处理句柄函数。

此模块使用 pyexpat 模块来提供对 Expat 解析器的访问。直接使用 pyexpat 模块的方式已被弃用。

此模块提供了一个异常和一个类型对象:

exception xml.parsers.expat.**ExpatError**
此异常会在 Expat 报错时被引发。请参阅*ExpatError 异常* 一节了解有关解读 Expat 错误的更多信息。

exception xml.parsers.expat.**error**
ExpatError 的别名。

xml.parsers.expat.**XMLParserType**
来自*ParserCreate()* 函数的返回值的类型。

xml.parsers.expat 模块包含两个函数:

xml.parsers.expat.**ErrorString**(*errno*)
返回给定错误号 *errno* 的解释性字符串。

xml.parsers.expat.**ParserCreate**(*encoding=None*, *namespace_separator=None*)
创建并返回一个新的 xmlparser 对象。如果指定了 *encoding*，它必须为指定 XML 数据所使用的编码格式名称的字符串。Expat 支持的编码格式没有 Python 那样多，而且它的编码格式库也不能被扩展；它支持 UTF-8, UTF-16, ISO-8859-1 (Latin1) 和 ASCII。如果给出了 *encoding*[1] 则它将覆盖隐式或显式指定的文档编码格式。

可以选择让 Expat 为你做 XML 命名空间处理，这是通过提供 *namespace_separator* 值来启用的。该值必须是一个单字符的字符串；如果字符串的长度不合法则将引发*ValueError* (None 被视为等同于省略)。当命名空间处理被启用时，属于特定命名空间的元素类型名称和属性名称将被展开。传递给 The element name passed to the 元素处理句柄 StartElementHandler 和 EndElementHandler 的元素名称将为命名空间 URI，命名空间分隔符和名称的本地部分的拼接。如果命名空间分隔符是一个零字节 (chr(0)) 则命名空间 URI 和本地部分将被直接拼接而不带任何分隔符。

举例来说，如果 *namespace_separator* 被设为空格符 (' ') 并对以下文档进行解析:

```
<?xml version="1.0"?>
<root xmlns    = "http://default-namespace.org/"
      xmlns:py = "http://www.python.org/ns/">
  <py:elem1 />
  <elem2 xmlns="" />
</root>
```

[1] 包括在 XML 输出中的编码格式字符串应当符合适当的标准。例如"UTF-8" 是有效的，但"UTF8" 是无效的。请参阅 https://www.w3.org/TR/2006/REC-xml11-20060816/#NT-EncodingDecl 和 https://www.iana.org/assignments/character-sets/character-sets.xhtml。

StartElementHandler 将为每个元素获取以下字符串:

```
http://default-namespace.org/ root
http://www.python.org/ns/ elem1
elem2
```

由于 pyexpat 所使用的 Expat 库的限制，被返回的 xmlparser 实例只能被用来解析单个 XML 文档。请为每个文档调用 ParserCreate 来提供单独的解析器实例。

也参考:

The Expat XML Parser Expat 项目的主页。

20.13.1 XMLParser 对象

xmlparser 对象具有以下方法:

xmlparser.**Parse**(*data*[, *isfinal*])
解析字符串 *data* 的内容，调用适当的处理函数来处理解析后的数据。在对此方法的最后一次调用时 *isfinal* 必须为真值；它允许以片段形式解析单个文件，而不是提交多个文件。*data* 在任何时候都可以为空字符串。

xmlparser.**ParseFile**(*file*)
解析从对象 *file* 读取的 XML 数据。*file* 仅需提供 read(nbytes) 方法，当没有更多数据可读时将返回空字符串。

xmlparser.**SetBase**(*base*)
设置要用于解析声明中的系统标识符的相对 URI 的基准。解析相对标识符的任务会留给应用程序进行：这个值将作为 *base* 参数传递给*ExternalEntityRefHandler()*, *NotationDeclHandler()* 和*UnparsedEntityDeclHandler()* 函数。

xmlparser.**GetBase**()
返回包含之前调用*SetBase()* 所设置的基准位置的字符串，或者如果未调用*SetBase()* 则返回 None。

xmlparser.**GetInputContext**()
将生成当前事件的输入数据以字符串形式返回。数据为包含文本的实体的编码格式。如果被调用时未激活事件处理句柄，则返回值将为 None。

xmlparser.**ExternalEntityParserCreate**(*context*[, *encoding*])
创建一个“子”解析器，可被用来解析由父解析器解析的内容所引用的外部解析实体。*context* 形参应当是传递给*ExternalEntityRefHandler()* 处理函数的字符串，具体如下所述。子解析器创建时*ordered_attributes* 和*specified_attributes* 会被设为此解析器的值。

xmlparser.**SetParamEntityParsing**(*flag*)
控制参数实体（包括外部 DTD 子集）的解析。可能的 *flag* 值有 XML_PARAM_ENTITY_PARSING_NEVER, XML_PARAM_ENTITY_PARSING_UNLESS_STANDALONE 和 XML_PARAM_ENTITY_PARSING_ALWAYS。如果该旗标设置成功则返回真值。

xmlparser.**UseForeignDTD**([*flag*])
调用时将 *flag* 设为真值（默认）将导致 Expat 调用*ExternalEntityRefHandler* 时将所有参数设为*None* 以允许加载替代的 DTD。如果文档不包含文档类型声明，*ExternalEntityRefHandler* 仍然会被调用，但*StartDoctypeDeclHandler* 和*EndDoctypeDeclHandler* 将不会被调用。

为 *flag* 传入假值将将撤消之前传入真值的调用，除此之外没有其他影响。

此方法只能在调用*Parse()* 或*ParseFile()* 方法之前被调用；在已调用过这两个方法之后调用它会导致引发*ExpatError* 且*code* 属性被设为 errors.codes[errors.XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING]。

xmlparser 对象具有下列属性:

xmlparser.**buffer_size**
当*buffer_text* 为真值时所使用的缓冲区大小。可以通过将此属性赋一个新的整数值来设置一个新的缓冲区大小。当大小发生改变时，缓冲区将被刷新。

xmlparser.**buffer_text**
将此属性设为真值会使得 xmlparser 对象缓冲 Expat 所返回的文本内容以尽可能地避免多次调用*CharacterDataHandler()* 回调。这可以显著地提升性能，因为 Expat 通常会将字符数据在每个行结束的位置上进行分块。此属性默认为假值，但可以在任何时候被更改。

xmlparser.**buffer_used**
当*buffer_text* 被启用时，缓冲区中存储的字节数。这些字节数据表示以 UTF-8 编码的文本。当*buffer_text* 为假值时此属性没有任何实际意义。

xmlparser.**ordered_attributes**
将该属性设为非零整数会使得各个属性被报告为列表而非字典。各个属性会按照在文档文本中的出现顺序显示。对于每个属性，将显示两个列表条目：属性名和属性值。（该模块的较旧版本也使用了此格式。）默认情况下，该属性为假值；它可以在任何时候被更改。

xmlparser.**specified_attributes**
如果设为非零整数，解析器将只报告在文档实例中指明的属性而不报告来自属性声明的属性。设置此属性的应用程序需要特别小心地使用从声明中获得的附加信息以符合 XML 处理程序的行为标准。默认情况下，该属性为假值；它可以在任何时候被更改。

下列属性包含与 xmlparser 对象遇到的最近发生的错误有关联的值，并且一旦对 Parse() 或 ParseFile() 的调用引发了*xml.parsers.expat.ExpatError* 异常就将只包含正确的值。

xmlparser.**ErrorByteIndex**
错误发生位置的字节索引号。

xmlparser.**ErrorCode**
指明问题的的数字代码。该值可被传给*ErrorString()* 函数，或是与在 errors 对象中定义的常量之一进行比较。

xmlparser.**ErrorColumnNumber**
错误发生位置的列号。

xmlparser.**ErrorLineNumber**
错误发生位置的行号。

下列属性包含 xmlparser 对象中关联到当前解析位置的值。在回调报告解析事件期间它们将指示生成事件的字符序列的第一个字符的位置。当在回调的外部被调用时，所指示的位置将恰好位于最后的解析事件之后（无论是否存在关联的回调）。

xmlparser.**CurrentByteIndex**
解析器输入的当前字节索引号。

xmlparser.**CurrentColumnNumber**
解析器输入的当前列号。

xmlparser.**CurrentLineNumber**
解析器输入的当前行号。

可被设置的处理句柄列表。要在一个 xmlparser 对象 *o* 上设置处理句柄，请使用 o.handlername = func。*handlername* 必须从下面的列表中获取，而 *func* 必须为接受正确数量参数的可调用对象。所有参数均为字符串，除非另外指明。

xmlparser.**XmlDeclHandler**(*version*, *encoding*, *standalone*)
当解析 XML 声明时被调用。XML 声明是 XML 建议适用版本、文档文本的编码格式，以及可选的“独立”声明的（可选）声明。*version* 和 *encoding* 将为字符串，而 *standalone* 在文档被声明为独立时将为 1，

在文档被声明为非独立时将为 0，或者在 standalone 短语被省略时则为 -1。这仅适用于 Expat 的 1.95.0 或更新版本。

xmlparser.**StartDoctypeDeclHandler**(*doctypeName*, *systemId*, *publicId*, *has_internal_subset*)
当 Expat 开始解析文档类型声明 (<!DOCTYPE ...) 时被调用。*doctypeName* 会完全按所显示的被提供。*systemId* 和 *publicId* 形参给出所指定的系统和公有标识符，如果被省略则为 None。如果文档包含内部文档声明子集则 *has_internal_subset* 将为真值。这要求 Expat 1.2 或更新的版本。

xmlparser.**EndDoctypeDeclHandler**()
当 Expat 完成解析文档类型声明时被调用。这要求 Expat 1.2 或更新版本。

xmlparser.**ElementDeclHandler**(*name*, *model*)
为每个元素类型声明调用一次。*name* 为元素类型名称，而 *model* 为内容模型的表示形式。

xmlparser.**AttlistDeclHandler**(*elname*, *attname*, *type*, *default*, *required*)
为一个元素类型的每个已声明属性执行调用。如果一个属性列表声明声明了三个属性，这个处理句柄会被调用三次，每个属性一次。*elname* 是声明所适用的元素的名称而 *attname* 是已声明的属性的名称。属性类型是作为 *type* 传入的字符串；可能的值有 'CDATA', 'ID', 'IDREF', ... *default* 给出了当属性未被文档实例所指明时该属性的默认值，或是为 None，如果没有默认值 (#IMPLIED 值) 的话。如果属性必须在文档实例中给出，则 *required* 将为真值。这要求 Expat 1.95.0 或更新的版本。

xmlparser.**StartElementHandler**(*name*, *attributes*)
在每个元素开始时被调用。*name* 是包含元素名称的字符串，而 *attributes* 是元素的属性。如果*ordered_attributes* 为真值，则属性为列表形式 (完整描述参见*ordered_attributes*)。否则为将名称映射到值的字典。

xmlparser.**EndElementHandler**(*name*)
在每个元素结束时被调用。

xmlparser.**ProcessingInstructionHandler**(*target*, *data*)
在每次处理指令时调用。

xmlparser.**CharacterDataHandler**(*data*)
针对字符数据调用。此方法将被用于普通字符数据、CDATA 标记内容以及可忽略的空白符。需要区分这几种情况的应用程序可以使用*StartCdataSectionHandler*, *EndCdataSectionHandler* 和*ElementDeclHandler* 回调来收集必要的信息。

xmlparser.**UnparsedEntityDeclHandler**(*entityName*, *base*, *systemId*, *publicId*, *notationName*)
针对未解析（NDATA）实体声明调用。此方法仅存在于 Expat 库的 1. 2 版；对于更新的版本，请改用*EntityDeclHandler*。（下层 Expat 库中的对应函数已被声明为过时。）

xmlparser.**EntityDeclHandler**(*entityName*, *is_parameter_entity*, *value*, *base*, *systemId*, *publicId*, *notationName*)
针对所有实体声明被调用。对于形参和内部实体，*value* 将为给出实体的声明内容的字符串；对于外部实体将为 None。*notationName* 形参对于已解析实体将为 None，对于未解析实体则为标注的名称。如果实体为形参实体则 *is_parameter_entity* 将为真值而如果为普通实体则为假值（大多数应用程序只需要关注普通实体）。此方法仅从 1.95.0 版 Expat 库开始才可用。

xmlparser.**NotationDeclHandler**(*notationName*, *base*, *systemId*, *publicId*)
针对标注声明被调用。*notationName*, *base*, *systemId* 和 *publicId* 如果给出则均应为字符串。如果省略公有标识符，则 *publicId* 将为 None。

xmlparser.**StartNamespaceDeclHandler**(*prefix*, *uri*)
当一个元素包含命名空间声明时被调用。命名空间声明会在为声明所在的元素调用*StartElementHandler* 之前被处理。

xmlparser.**EndNamespaceDeclHandler**(*prefix*)
当到达包含命名空间声明的元素的关闭标记时被调用。此方法会按照调用*StartNamespaceDeclHandler* 以指明每个命名空间作用域的开始的逆顺序为元素上的每个命名空间声明调用一次。对这个处理句柄的调用是在相应的*EndElementHandler* 之后针对元素的结束而进行的。

xmlparser.**CommentHandler**(*data*)
针对注释被调用。*data* 是注释的文本，不包括开头的 '<!--' 和末尾的 '-->'。

xmlparser.**StartCdataSectionHandler**()
在一个 CDATA 节的开头被调用。需要此方法和*EndCdataSectionHandler* 以便能够标识 CDATA 节的语法开始和结束。

xmlparser.**EndCdataSectionHandler**()
在一个 CDATA 节的末尾被调用。

xmlparser.**DefaultHandler**(*data*)
针对 XML 文档中没有指定适用处理句柄的任何字符被调用。这包括了所有属于可被报告的结构的一部分，但未提供处理句柄的字符。

xmlparser.**DefaultHandlerExpand**(*data*)
这与*DefaultHandler()* 相同，但不会抑制内部实体的扩展。实体引用将不会被传递给默认处理句柄。

xmlparser.**NotStandaloneHandler**()
当 XML 文档未被声明为独立文档时被调用。这种情况发生在出现外部子集或对参数实体的引用，但 XML 声明没有在 XML 声明中将 standalone 设为 yes 的时候。如果这个处理句柄返回 0，那么解析器将引发 XML_ERROR_NOT_STANDALONE 错误。如果这个处理句柄没有被设置，那么解析器就不会为这个条件引发任何异常。

xmlparser.**ExternalEntityRefHandler**(*context*, *base*, *systemId*, *publicId*)
为对外部实体的引用执行调用。*base* 为当前的基准，由之前对*SetBase()* 的调用设置。公有和系统标识符 *systemId* 和 *publicId* 如果给出则🄵为字符串；如果公有标识符未给出，则 *publicId* 将为 None。*context* 是仅根据以下说明来使用的不透明值。

对于要解析的外部实体，这个处理句柄必须被实现。它负责使用 ExternalEntityParserCreate(context) 来创建子解析器，通过适当的回调将其初始化，并对实体进行解析。这个处理句柄应当返回一个整数；如果它返回 0，则解析器将引发 XML_ERROR_EXTERNAL_ENTITY_HANDLING 错误，否则解析将会继续。

如果未提供这个处理句柄，外部实体会由*DefaultHandler* 回调来报告，如果提供了该回调的话。

20.13.2 ExpatError 异常

ExpatError 异常包含几个有趣的属性:

ExpatError.**code**
Expat 对于指定错误的内部错误号。*errors.messages* 字典会将这些错误号映射到 Expat 的错误消息。例如:

```
from xml.parsers.expat import ParserCreate, ExpatError, errors

p = ParserCreate()
try:
    p.Parse(some_xml_document)
except ExpatError as err:
    print("Error:", errors.messages[err.code])
```

errors 模块也提供了一些错误消息常量和一个将这些消息映射回错误码的字典*codes*，参见下文。

ExpatError.**lineno**
检测到错误所在的行号。首行的行号为 1。

ExpatError.**offset**
错误发生在行中的字符偏移量。首列的列号为 0。

20.13.3 示例

以下程序定义了三个处理句柄，会简单地打印出它们的参数。:

```
import xml.parsers.expat

# 3 handler functions
def start_element(name, attrs):
    print('Start element:', name, attrs)
def end_element(name):
    print('End element:', name)
def char_data(data):
    print('Character data:', repr(data))

p = xml.parsers.expat.ParserCreate()

p.StartElementHandler = start_element
p.EndElementHandler = end_element
p.CharacterDataHandler = char_data

p.Parse("""<?xml version="1.0"?>
<parent id="top"><child1 name="paul">Text goes here</child1>
<child2 name="fred">More text</child2>
</parent>""", 1)
```

来自这个程序的输出是:

```
Start element: parent {'id': 'top'}
Start element: child1 {'name': 'paul'}
Character data: 'Text goes here'
End element: child1
Character data: '\n'
Start element: child2 {'name': 'fred'}
Character data: 'More text'
End element: child2
Character data: '\n'
End element: parent
```

20.13.4 内容模型描述

内容模型是使用嵌套的元组来描述的。每个元素包含四个值：类型、限定符、名称和一个子元组。子元组就是附加的内容模型描述。

前两个字段的值是在*xml.parsers.expat.model* 模块中定义的常量。这些常量可分为两组：模型类型组和限定符组。

模型类型组中的常量有:

xml.parsers.expat.model.**XML_CTYPE_ANY**
: 模型名称所指定的元素被声明为具有 ANY 内容模型。

xml.parsers.expat.model.**XML_CTYPE_CHOICE**
: 命名元素允许从几个选项中选择；这被用于内容模型例如 (A | B | C)。

xml.parsers.expat.model.**XML_CTYPE_EMPTY**
: 被声明为 EMPTY 的元素具有此模型类型。

xml.parsers.expat.model.**XML_CTYPE_MIXED**

xml.parsers.expat.model.**XML_CTYPE_NAME**

xml.parsers.expat.model.**XML_CTYPE_SEQ**
代表彼此相连的一系列模型的模型用此模型类型来指明。这被用于 (A, B, C) 这样的模型。

限定符组中的常量有:

xml.parsers.expat.model.**XML_CQUANT_NONE**
未给出限定符，这样它可以只出现一次，例如 A。

xml.parsers.expat.model.**XML_CQUANT_OPT**
可选的模型：它可以出现一次或完全不出现，例如 A?。

xml.parsers.expat.model.**XML_CQUANT_PLUS**
模型必须出现一次或多次 (例如 A+)。

xml.parsers.expat.model.**XML_CQUANT_REP**
模型必须出现零次或多次，例如 A*。

20.13.5 Expat 错误常量

下列常量是在*xml.parsers.expat.errors* 模块中提供的。这些常量在有错误发生时解读被引发的 ExpatError 异常对象的某些属性时很有用处。出于保持向下兼容性的理由，这些常量的值是错误 消息而不是数字形式的错误 代码，为此你可以将它的*code* 属性和 errors.codes[errors.XML_ERROR_*CONSTANT_NAME*] 进行比较。

errors 模块具有以下属性:

xml.parsers.expat.errors.**codes**
将字符串描述映射到其错误代码的字典。

3.2 版新加入.

xml.parsers.expat.errors.**messages**
将数字形式的错误代码映射到其字符串描述的字典。

3.2 版新加入.

xml.parsers.expat.errors.**XML_ERROR_ASYNC_ENTITY**

xml.parsers.expat.errors.**XML_ERROR_ATTRIBUTE_EXTERNAL_ENTITY_REF**
属性值中指向一个外部实体而非内部实体的实体引用。

xml.parsers.expat.errors.**XML_ERROR_BAD_CHAR_REF**
指向一个在 XML 不合法的字符的字符引用 (例如，字符 0 或'&#0;')。

xml.parsers.expat.errors.**XML_ERROR_BINARY_ENTITY_REF**
指向一个使用标注声明，因而无法被解析的实体的实体引用。

xml.parsers.expat.errors.**XML_ERROR_DUPLICATE_ATTRIBUTE**
一个属性在一个开始标记中被使用超过一次。

xml.parsers.expat.errors.**XML_ERROR_INCORRECT_ENCODING**

xml.parsers.expat.errors.**XML_ERROR_INVALID_TOKEN**
当一个输入字节无法被正确分配给一个字符时引发；例如，在 UTF-8 输入流中的 NUL 字节 (值为 0).

xml.parsers.expat.errors.**XML_ERROR_JUNK_AFTER_DOC_ELEMENT**
在文档元素之后出现空白符以外的内容。

xml.parsers.expat.errors.**XML_ERROR_MISPLACED_XML_PI**
在输入数据开始位置以外的地方发现 XML 声明。

xml.parsers.expat.errors.**XML_ERROR_NO_ELEMENTS**
文档不包含任何元素（XML 要求所有文档都包含恰好一个最高层级元素）。

xml.parsers.expat.errors.**XML_ERROR_NO_MEMORY**
Expat 无法在内部分配内存。

xml.parsers.expat.errors.**XML_ERROR_PARAM_ENTITY_REF**
在不被允许的位置发现一个参数实体引用。

xml.parsers.expat.errors.**XML_ERROR_PARTIAL_CHAR**
在输入中发出一个不完整的字符。

xml.parsers.expat.errors.**XML_ERROR_RECURSIVE_ENTITY_REF**
一个实体引用包含了对同一实体的另一个引用；可能是通过不同的名称，并可能是间接的引用。

xml.parsers.expat.errors.**XML_ERROR_SYNTAX**
遇到了某个未指明的语法错误。

xml.parsers.expat.errors.**XML_ERROR_TAG_MISMATCH**
一个结束标记不能匹配到最内层的未关闭开始标记。

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_TOKEN**
某些记号（例如开始标记）在流结束或遇到下一个记号之前还未关闭。

xml.parsers.expat.errors.**XML_ERROR_UNDEFINED_ENTITY**
对一个未定义的实体进行了引用。

xml.parsers.expat.errors.**XML_ERROR_UNKNOWN_ENCODING**
文档编码格式不被 Expat 所支持。

xml.parsers.expat.errors.**XML_ERROR_UNCLOSED_CDATA_SECTION**
一个 CDATA 标记节还未关闭。

xml.parsers.expat.errors.**XML_ERROR_EXTERNAL_ENTITY_HANDLING**

xml.parsers.expat.errors.**XML_ERROR_NOT_STANDALONE**
解析器确定文档不是“独立的”但它却在 XML 声明中声明自己是独立的，并且 `NotStandaloneHandler` 被设置为返回 `0`。

xml.parsers.expat.errors.**XML_ERROR_UNEXPECTED_STATE**

xml.parsers.expat.errors.**XML_ERROR_ENTITY_DECLARED_IN_PE**

xml.parsers.expat.errors.**XML_ERROR_FEATURE_REQUIRES_XML_DTD**
请求了一个需要已编译 DTD 支持的操作，但 Expat 被配置为不带 DTD 支持。此错误应当绝对不会被*xml.parsers.expat* 模块的标准构建版本所报告。

xml.parsers.expat.errors.**XML_ERROR_CANT_CHANGE_FEATURE_ONCE_PARSING**
在解析开始之后请求一个只能在解析开始之前执行的行为改变。此错误（目前）只能由 `UseForeignDTD()` 所引发。

xml.parsers.expat.errors.**XML_ERROR_UNBOUND_PREFIX**
当命名空间处理被启用时发现一个未声明的前缀。

xml.parsers.expat.errors.**XML_ERROR_UNDECLARING_PREFIX**
文档试图移除与某个前缀相关联的命名空间声明。

xml.parsers.expat.errors.**XML_ERROR_INCOMPLETE_PE**
一个参数实体包含不完整的标记。

xml.parsers.expat.errors.**XML_ERROR_XML_DECL**
文档完全未包含任何文档元素。

xml.parsers.expat.errors.**XML_ERROR_TEXT_DECL**
解析一个外部实体中的文本声明时出现错误。

xml.parsers.expat.errors.**XML_ERROR_PUBLICID**
在公有 id 中发现不被允许的字符。

xml.parsers.expat.errors.**XML_ERROR_SUSPENDED**
在挂起的解析器上请求执行操作，但未获得允许。这包括提供额外输入或停止解析器的尝试。

xml.parsers.expat.errors.**XML_ERROR_NOT_SUSPENDED**
在解析器未被挂起的时候执行恢复解析器的尝试。

xml.parsers.expat.errors.**XML_ERROR_ABORTED**
此错误不应当被报告给 Python 应用程序。

xml.parsers.expat.errors.**XML_ERROR_FINISHED**
在一个已经完成解析输入的解析器上请求执行操作，但未获得允许。这包括提供额外输入或停止解析器的尝试。

xml.parsers.expat.errors.**XML_ERROR_SUSPEND_PE**

🄵解

CHAPTER 21

互联网协议和支持

本章介绍的模块实现了互联网协议并支持相关技术。它们都是用 Python 实现的。这些模块中的大多数都需要存在依赖于系统的模块*socket*，目前大多数流行平台都支持它。这是一个概述：

21.1 webbrowser --- 方便的 Web 浏览器控制器

源码： Lib/webbrowser.py

---

webbrowser 模块提供了一个高级接口，允许向用户显示基于 Web 的文档。在大多数情况下，只需从该模块调用*open()* 函数就可以了。

在 Unix 下，图形浏览器在 X11 下是首选，但如果图形浏览器不可用或 X11 显示不可用，则将使用文本模式浏览器。如果使用文本模式浏览器，则调用进程将阻塞，直到用户退出浏览器。

如果存在环境变量 BROWSER，则将其解释为*os.pathsep* 分隔的浏览器列表，以便在平台默认值之前尝试。当列表部分的值包含字符串 %s 时，它被解释为一个文字浏览器命令行，用于替换 %s 的参数 URL；如果该部分不包含 %s，则它只被解释为要启动的浏览器的名称。[1]

对于非 Unix 平台，或者当 Unix 上有远程浏览器时，控制过程不会等待用户完成浏览器，而是允许远程浏览器在显示界面上维护自己的窗口。如果 Unix 上没有远程浏览器，控制进程将启动一个新的浏览器并等待。

脚本 **webbrowser** 可以用作模块的命令行界面。它接受一个 URL 作为参数。还接受以下可选参数：-n 如果可能，在新的浏览器窗口中打开 URL；-t 在新的浏览器页面（“标签”）中打开 URL。这些选择当然是相互排斥的。用法示例：

```
python -m webbrowser -t "https://www.python.org"
```

定义了以下异常：

exception webbrowser.**Error**
发生浏览器控件错误时引发异常。

[1] 这里命名的不带完整路径的可执行文件将在 PATH 环境变量给出的目录中搜索。

定义了以下函数：

webbrowser.**open**(*url*, *new=0*, *autoraise=True*)
使用默认浏览器显示 *url*。如果 *new* 为 0，则尽可能在同一浏览器窗口中打开 *url*。如果 *new* 为 1，则尽可能打开新的浏览器窗口。如果 *new* 为 2，则尽可能打开新的浏览器页面（“标签”）。如果 *autoraise* 为 “True”，则会尽可能置前窗口（请注意，在许多窗口管理器下，无论此变量的设置如何，都会置前窗口）。

请注意，在某些平台上，尝试使用此函数打开文件名，可能会起作用并启动操作系统的关联程序。但是，这种方式不被支持也不可移植。

使用 `url` 参数会引发*auditing event* `webbrowser.open` 。

webbrowser.**open_new**(*url*)
如果可能，在默认浏览器的新窗口中打开 *url*，否则，在唯一的浏览器窗口中打开 *url*。

webbrowser.**open_new_tab**(*url*)
如果可能，在默认浏览器的新页面（“标签”）中打开 *url*，否则等效于*open_new()*。

webbrowser.**get**(*using=None*)
返回浏览器类型为 *using* 指定的控制器对象。如果 *using* 为 `None`，则返回适用于调用者环境的默认浏览器的控制器。

webbrowser.**register**(*name*, *constructor*, *instance=None*, *, *preferred=False*)
注册 *name* 浏览器类型。注册浏览器类型后，*get()* 函数可以返回该浏览器类型的控制器。如果没有提供 *instance*，或者为 `None`，*constructor* 将在没有参数的情况下被调用，以在需要时创建实例。如果提供了 *instance*，则永远不会调用 *constructor*，并且可能是 `None`。

将 *preferred* 设置为 `True` 使得这个浏览器成为*get()* 不带参数调用的首选结果。否则，只有在您计划设置 `BROWSER` 变量，或使用与您声明的处理程序的名称相匹配的非空参数调用*get()* 时，此入口点才有用。

3.7 版更變: 添加了仅关键字参数 *preferred*。

预定义了许多浏览器类型。此表给出了可以传递给*get()* 函数的类型名称以及控制器类的相应实例化，这些都在此模块中定义。

类型名	类名	F解
'mozilla'	Mozilla('mozilla')	
'firefox'	Mozilla('mozilla')	
'netscape'	Mozilla('netscape')	
'galeon'	Galeon('galeon')	
'epiphany'	Galeon('epiphany')	
'skipstone'	BackgroundBrowser('skipstone')	
'kfmclient'	Konqueror()	(1)
'konqueror'	Konqueror()	(1)
'kfm'	Konqueror()	(1)
'mosaic'	BackgroundBrowser('mosaic')	
'opera'	Opera()	
'grail'	Grail()	
'links'	GenericBrowser('links')	
'elinks'	Elinks('elinks')	
'lynx'	GenericBrowser('lynx')	
'w3m'	GenericBrowser('w3m')	
'windows-default'	WindowsDefault	(2)
'macosx'	MacOSXOSAScript('default')	(3)
'safari'	MacOSXOSAScript('safari')	(3)
'google-chrome'	Chrome('google-chrome')	
'chrome'	Chrome('chrome')	
'chromium'	Chromium('chromium')	
'chromium-browser'	Chromium('chromium-browser')	

F解:

(1) “Konqueror” 是 Unix 的 KDE 桌面环境的文件管理器，只有在 KDE 运行时才有意义。一些可靠地检测 KDE 的方法会很好；仅检查 KDEDIR 变量是不够的。另请注意，KDE 2 的 **konqueror** 命令，会使用名称 “kfm” ---此实现选择运行的 Konqueror 的最佳策略。

(2) 仅限 Windows 平台。

(3) Only on macOS platform.

3.3 版新加入: 添加了对 Chrome/Chromium 的支持。

以下是一些简单的例子:

```
url = 'https://docs.python.org/'

# Open URL in a new tab, if a browser window is already open.
webbrowser.open_new_tab(url)

# Open URL in new window, raising the window if possible.
webbrowser.open_new(url)
```

21.1.1 浏览器控制器对象

浏览器控制器提供三个与模块级便捷函数相同的方法：

controller.**open**(*url*, *new=0*, *autoraise=True*)
使用此控制器处理的浏览器显示 *url*。如果 *new* 为 1，则尽可能打开新的浏览器窗口。如果 *new* 为 2，则尽可能打开新的浏览器页面（“标签”）。

controller.**open_new**(*url*)
如果可能，在此控制器处理的浏览器的新窗口中打开 *url* ，否则，在唯一的浏览器窗口中打开 *url* 。别名*open_new()*。

controller.**open_new_tab**(*url*)
如果可能，在此控制器处理的浏览器的新页面（“标签”）中打开 *url*，否则等效于*open_new()*。

🄵解

21.2 wsgiref --- WSGI 工具和引用的实现

Web 服务器网关接口（WSGI）是 Web 服务器软件和用 Python 编写的 Web 应用程序之间的标准接口。具有标准接口能够让支持 WSGI 的应用程序与多种不同的 Web 服务器配合使用。

只有 Web 服务器和编程框架的开发者才需要了解 WSGI 设计的每个细节和边界情况。你不需要了解 WSGI 的每个细节而只需要安装一个 WSGI 应用程序或编写使用现有框架的 Web 应用程序。

wsgiref 是一个 WSGI 规范的参考实现，可被用于将 WSGI 支持添加到 Web 服务器或框架中。它提供了操作 WSGI 环境变量和响应标头的工具，实现 WSGI 服务器的基类，一个可部署 WSGI 应用程序的演示性 HTTP 服务器，以及一个用于检查 WSGI 服务器和应用程序是否符合 WSGI 规范的验证工具 (**PEP 3333**)。

参看 wsgi.readthedocs.io 获取有关 WSGI 的更多信息，以及教程和其他资源的链接。

21.2.1 wsgiref.util -- WSGI 环境工具

本模块提供了多种工具配合 WSGI 环境使用。WSGI 环境就是一个包含 **PEP 3333** 所描述的 HTTP 请求变量的目录。所有接受 *environ* 形参的函数都会预期被得到一个符合 WSGI 规范的目录；请参阅 **PEP 3333** 来了解相关规范的细节。

wsgiref.util.**guess_scheme**(*environ*)
返回对于 wsgi.url_scheme 应为”http” 还是”https” 的猜测，具体方式是在 *environ* 中检查 HTTPS 环境变量。返回值是一个字符串。

此函数适用于创建一个包装了 CGI 或 CGI 类协议例如 FastCGI 的网关。通常，提供这种协议的服务器将包括一个 HTTPS 变量并会在通过 SSL 接收请求时将其值设为”1”, ”yes” 或”on”。这样，此函数会在找到上述值时返回”https”，否则返回”http”。

wsgiref.util.**request_uri**(*environ*, *include_query=True*)
使用 **PEP 3333** 的”URL Reconstruction” 一节中的算法返回完整的请求 URL，可能包括查询字符串。如果 *include_query* 为假值，则结果 URI 中将不包括查询字符串。

wsgiref.util.**application_uri**(*environ*)
类似于*request_uri()*，区别在于 PATH_INFO 和 QUERY_STRING 变量会被忽略。结果为请求所指定的应用程序对象的基准 URI。

wsgiref.util.**shift_path_info**(*environ*)
将单个名称从 PATH_INFO 变换为 SCRIPT_NAME 并返回该名称。*environ* 目录将被原地 修改；如果你需要保留原始 PATH_INFO 或 SCRIPT_NAME 不变请使用一个副本。

如果 PATH_INFO 中没有剩余的路径节，则返回 None。

通常，此例程被用来处理请求 URI 路径的每个部分，比如说将路径当作是一系列字典键。此例程会修改传入的环境以使其适合发起调用位于目标 URI 上的其他 WSGI 应用程序，如果有一个 WSGI 应用程序位于 /foo，而请求 URI 路径为 /foo/bar/baz，且位于 /foo 的 WSGI 应用程序调用了*shift_path_info()*，它将获得字符串"bar"，而环境将被更新以适合传递给位于 /foo/bar 的 WSGI 应用程序。也就是说，SCRIPT_NAME 将由 /foo 变为 /foo/bar，而 PATH_INFO 将由 /bar/baz 变为 /baz。

当 PATH_INFO 只是"/" 时，此例程会返回一个空字符串并在 SCRIPT_NAME 末尾添加一个斜杠，虽然空的路径节通常会被忽略，并且 SCRIPT_NAME 通常也不以斜杠作为结束。此行为是有意为之的，用来确保应用程序在使用此例程执行对象遍历时能区分以 /x 结束的和以 /x/ 结束的 URI。

wsgiref.util.**setup_testing_defaults**(*environ*)
以简短的默认值更新 *environ* 用于测试目的。

此全程会添加 WSGI 所需要的各种参数，包括 HTTP_HOST, SERVER_NAME, SERVER_PORT, REQUEST_METHOD, SCRIPT_NAME, PATH_INFO 以及 **PEP 3333** 中定义的所有 wsgi.* 变量。它只提供默认值，而不会替换这些变量的现有设置。

此例程的目的是让 WSGI 服务器的单元测试以及应用程序设置测试环境更为容易。它不应该被实际的 WSGI 服务器或应用程序所使用，因为它用的是假数据！

用法示例：

```
from wsgiref.util import setup_testing_defaults
from wsgiref.simple_server import make_server

# A relatively simple WSGI application. It's going to print out the
# environment dictionary after being updated by setup_testing_defaults
def simple_app(environ, start_response):
    setup_testing_defaults(environ)

    status = '200 OK'
    headers = [('Content-type', 'text/plain; charset=utf-8')]

    start_response(status, headers)

    ret = [("%s: %s\n" % (key, value)).encode("utf-8")
           for key, value in environ.items()]
    return ret

with make_server('', 8000, simple_app) as httpd:
    print("Serving on port 8000...")
    httpd.serve_forever()
```

除了上述的环境函数，*wsgiref.util* 模块还提供了以下辅助工具：

wsgiref.util.**is_hop_by_hop**(*header_name*)
如果'header_name' 是 **RFC 2616** 所定义的 HTTP/1.1 "Hop-by-Hop" 标头则返回 True。

class wsgiref.util.**FileWrapper**(*filelike*, *blksize=8192*)
一个将文件类对象转换为*iterator* 的包装器。结果对象同时支持 __getitem__() 和 __iter__() 迭代形式，以便兼容 Python 2.1 和 Jython。当对象被迭代时，可选的 *blksize* 形参将被反复地传给 文件类对象的 read() 方法以获取字节串并输出。当 read() 返回空字节串时，迭代将结束并不可再恢复。

如果 *filelike* 具有 `close()` 方法，返回的对象也将具有 `close()` 方法，并且它将在被调用时发起调用 *filelike* 对象的 `close()` 方法。

用法示例：

```
from io import StringIO
from wsgiref.util import FileWrapper

# We're using a StringIO-buffer for as the file-like object
filelike = StringIO("This is an example file-like object"*10)
wrapper = FileWrapper(filelike, blksize=5)

for chunk in wrapper:
    print(chunk)
```

3.8 版後已🄴用: 对 序列协议的支持已被弃用。

21.2.2 `wsgiref.headers` -- WSGI 响应标头工具

此模块提供了一个单独的类*Headers*，可以方便地使用一个映射类接口操作 WSGI 响应标头。

class wsgiref.headers.**Headers**([*headers*])

创建一个包装了 *headers* 的映射类对象，它必须为如 **PEP 3333** 所描述的由标头名称/值元组构成的列表。*headers* 的默认值为一个空列表。

Headers 对象支持典型的映射操作包括 `__getitem__()`, `get()`, `__setitem__()`, `setdefault()`, `__delitem__()` 和 `__contains__()`。对于以上各个方法，映射的键是标头名称（大小写不敏感），而值是关联到该标头名称的第一个值。设置一个标头会移除该标头的任何现有值，再将一个新值添加到所包装的标头列表末尾。标头的现有顺序通常会保持不变，只在所包装的列表末尾添加新的标头。

与字典不同，当你试图获取或删除所包装的标头列表中不存在的键时*Headers* 对象不会引发错误。获取一个不存在的标头只是返回 `None`，而删除一个不存在的标头则没有任何影响。

Headers 对象还支持 `keys()`, `values()` 以及 `items()` 方法。如果存在具有多个值的键则 `keys()` 和 `items()` 所返回的列表可能包括多个相同的键。对*Headers* 对象执行 `len()` 的结果与 `items()` 的长度相同，也与所包装的标头列表的长度相同。实际上，`items()` 方法只是返回所包装的标头列表的一个副本。

在*Headers* 对象上调用 `bytes()` 将返回适用于作为 HTTP 响应标头来传输的已格式化字节串。每个标头附带以逗号加空格分隔的值放置在一行中。每一行都以回车符加换行符结束，且该字节串会以一个空行作为结束。

除了映射接口和格式化特性，*Headers* 对象还具有下列方法用来查询和添加多值标头，以及添加具有 MIME 参数的标头:

get_all(*name*)

返回包含指定标头的所有值的列表。

返回的列表项将按它们在原始标头列表中的出现或被添加到实例中的顺序排序，并可能包含重复项。任何被删除并重新插入的字段总是会被添加到标头列表末尾。如果给定名称的字段不存在，则返回一个空列表。

add_header(*name*, *value*, ***_params*)

添加一个（可能有多个值）标头，带有通过关键字参数指明的可选的 MIME 参数。

name 是要添加的标头字段。可以使用关键字参数来为标头字段设置 MIME 参数。每个参数必须为字符串或 `None`。参数名中的下划线会被转换为连字符，因为连字符不可在 Python 标识符中出现，但许多 MIME 参数名都包括连字符。如果参数值为字符串，它会以 `name="value"` 的形式被添

加到标头值参数中。如果为 None，则只会添加参数名。(这适用于没有值的 MIME 参数。) 示例用法:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

以上代码将添加一个这样的标头:

```
Content-Disposition: attachment; filename="bud.gif"
```

3.5 版更變: *headers* 形参是可选的。

21.2.3 wsgiref.simple_server -- 一个简单的 WSGI HTTP 服务器

此模块实现了一个简单的 HTTP 服务器 (基于*http.server*) 来发布 WSGI 应用程序。每个服务器实例会在给定的主机名和端口号上发布一个 WSGI 应用程序。如果你想在一个主机名和端口号上发布多个应用程序，你应当创建一个通过解析 PATH_INFO 来选择每个请求要发起调用哪个应用程序的 WSGI 应用程序。(例如，使用*wsgiref.util* 中的 shift_path_info() 函数。)

wsgiref.simple_server.**make_server**(*host*, *port*, *app*, *server_class=WSGIServer*, *handler_class=WSGIRequestHandler*)

创建一个新的 WSGI 服务器并在 *host* 和 *port* 上进行监听，接受对 *app* 的连接。返回值是所提供的 *server_class* 的实例，并将使用指定的 *handler_class* 来处理请求。*app* 必须是一个如 **PEP 3333** 所定义的 WSGI 应用程序对象。

用法示例:

```
from wsgiref.simple_server import make_server, demo_app

with make_server('', 8000, demo_app) as httpd:
    print("Serving HTTP on port 8000...")

    # Respond to requests until process is killed
    httpd.serve_forever()

    # Alternative: serve one request, then exit
    httpd.handle_request()
```

wsgiref.simple_server.**demo_app**(*environ*, *start_response*)

此函数是一个小巧但完整的 WSGI 应用程序，它返回一个包含消息"Hello world!" 以及 *environ* 形参中提供的键/值对的文本页面。它适用于验证 WSGI 服务器 (例如*wsgiref.simple_server*) 是否能够正确地运行一个简单的 WSGI 应用程序。

class wsgiref.simple_server.**WSGIServer**(*server_address*, *RequestHandlerClass*)

创建一个*WSGIServer* 实例。*server_address* 应当是一个 (host,port) 元组，而 *RequestHandlerClass* 应当是*http.server.BaseHTTPRequestHandler* 的子类，它将被用来处理请求。

你通常不需要调用此构造器，因为*make_server()* 函数能为你处理所有的细节。

WSGIServer 是*http.server.HTTPServer* 的子类，因此它所有的方法 (例如 serve_forever() 和 handle_request()) 都是可用的。*WSGIServer* 还提供了以下 WSGI 专属的方法:

set_app(*application*)

将可调用对象 *application* 设为将要接受请求的 WSGI 应用程序。

get_app()

返回当前设置的应用程序可调用对象。

但是，你通常不需要使用这些附加的方法，因为*set_app()* 通常会由*make_server()* 来调用，而*get_app()* 主要是针对请求处理句柄实例的。

class wsgiref.simple_server.**WSGIRequestHandler**(*request*, *client_address*, *server*)
为给定的 *request* 创建一个 HTTP 处理句柄 (例如套接字)，*client_address* (一个 `(host,port)` 元组)，以及 *server* (*WSGIServer* 实例)。

你不需要直接创建该类的实例；它们会根据*WSGIServer* 对象的需要自动创建。但是，你可以子类化该类并将其作为 *handler_class* 提供给*make_server()* 函数。一些可能在子类中重载的相关方法:

get_environ()
返回包含针对一个请求的 WSGI 环境的字典。默认实现会拷贝*WSGIServer* 对象的 base_environ 字典属性的内容并添加从 HTTP 请求获取的各种标头。对此方法的每个调用应当返回一个 **PEP 3333** 所指明的包含所有相关 CGI 环境变量的新字典。

get_stderr()
返回应被用作 wsgi.errors 流的对象。默认实现将只返回 sys.stderr。

handle()
处理 HTTP 请求。默认的实现会使用*wsgiref.handlers* 类创建一个处理句柄实例来实现实际的 WSGI 应用程序接口。

21.2.4 wsgiref.validate --- WSGI 一致性检查器

当创建新的 WSGI 应用程序对象、框架、服务器或中间件时，使用*wsgiref.validate* 来验证新代码的一致性是很有用的。此模块提供了一个创建 WSGI 应用程序对象的函数来验证 WSGI 服务器或网关与 WSGI 应用程序对象之间的通信，以便检查双方的协议一致性。

请注意这个工具并不保证完全符合 **PEP 3333**；此模块没有报告错误并不一定表示不存在错误。但是，如果此模块确实报告了错误，那么几乎可以肯定服务器或应用程序不是 100% 符合要求的。

此模块是基于 Ian Bicking 的"Python Paste" 库的 paste.lint 模块。

wsgiref.validate.**validator**(*application*)
包装 *application* 并返回一个新的 WSGI 应用程序对象。返回的应用程序将转发所有请求到原始的 *application*，并将检查 *application* 和发起调用它的服务器是否都符合 WSGI 规范和 **RFC 2616**。

任何被检测到的不一致性都会导致引发*AssertionError*；但是请注意，如何对待这些错误取决于具体服务器。例如，*wsgiref.simple_server* 和其他基于*wsgiref.handlers* 的服务器（它们没有重载错误处理方法来做其他操作）将简单地输出一条消息报告发生了错误，并将回溯转给 sys.stderr 或者其他错误数据流。

这个包装器也可能会使用*warnings* 模块生成输出来指明存在问题但实际上未被 **PEP 3333** 所禁止的行为。除非它们被 Python 命令行选项或*warnings* API 所屏蔽，否则任何此类警告都将被写入到 sys.stderr (不是 wsgi.errors，除非它们恰好是同一个对象)。

用法示例:

```
from wsgiref.validate import validator
from wsgiref.simple_server import make_server

# Our callable object which is intentionally not compliant to the
# standard, so the validator is going to break
def simple_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain')]  # HTTP Headers
    start_response(status, headers)

    # This is going to break because we need to return a list, and
    # the validator is going to inform us
    return b"Hello World"
```

(下页继续)

(繼續上一頁)

```
# This is the application wrapped in a validator
validator_app = validator(simple_app)

with make_server('', 8000, validator_app) as httpd:
    print("Listening on port 8000....")
    httpd.serve_forever()
```

21.2.5 wsgiref.handlers -- 服务器/网关基类

此模块提供了用于实现 WSGI 服务器和网关的处理句柄基类。这些基类可处理大部分与 WSGI 应用程序通信的工作，只要给予它们一个带有输入、输出和错误流的 CGI 类环境。

class wsgiref.handlers.**CGIHandler**

通过 sys.stdin, sys.stdout, sys.stderr 和 os.environ 发起基于 CGI 的调用。这在当你有一个 WSGI 应用程序并想将其作为 CGI 脚本运行时很有用处。只需发起调用 CGIHandler().run(app)，其中 app 是你想要发起调用的 WSGI 应用程序。

该类是*BaseCGIHandler* 的子类，它将设置 wsgi.run_once 为真值，wsgi.multithread 为假值，wsgi.multiprocess 为真值，并总是使用*sys* 和*os* 来获取所需的 CGI 流和环境。

class wsgiref.handlers.**IISCGIHandler**

CGIHandler 的一个专门化替代，用于在 Microsoft 的 IIS Web 服务器上部署，无需设置 config allowPathInfo 选项 (IIS>=7) 或 metabase allowPathInfoForScriptMappings (IIS<7)。

默认情况下，IIS 给出的 PATH_INFO 会与前面的 SCRIPT_NAME 重复，导致想要实现路由的 WSGI 应用程序出现问题。这个处理句柄会去除任何这样的重复路径。

IIS 可被配置为传递正确的 PATH_INFO，但这会导致另一个 PATH_TRANSLATED 出错的问题。幸运的是这个变量很少被使用并且不被 WSGI 所保证。但是在 IIS<7 上，这个设置只能在 vhost 层级上进行，影响到所有其他脚本映射，其中许多在受 PATH_TRANSLATED 问题影响时都会中断运行。因此 IIS<7 的部署几乎从不附带这样的修正（即使 IIS7 也很少使用它，因为它仍然不带 UI）。

CGI 代码没有办法确定该选项是否已设置，因此提供了一个单独的处理句柄类。它的用法与*CGIHandler* 相同，即通过调用 IISCGIHandler().run(app)，其中 app 是你想要发起调用的 WSGI 应用程序。

3.2 版新加入.

class wsgiref.handlers.**BaseCGIHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

类似于*CGIHandler*，但是改用*sys* 和*os* 模块，CGI 环境和 I/O 流会被显式地指定。*multithread* 和 *multiprocess* 值被用来为处理句柄实例所运行的任何应用程序设置 wsgi.multithread 和 wsgi.multiprocess 旗标。

该类是*SimpleHandler* 的子类，旨在用于 HTTP "origin servers" 以外的软件。如果你在编写一个网关协议实现（例如 CGI, FastCGI, SCGI 等等），它使用 Status: 标头来发布 HTTP 状态，你可能会想要子类化该类而不是*SimpleHandler*。

class wsgiref.handlers.**SimpleHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

类似于*BaseCGIHandler*，但被设计用于 HTTP 原始服务器。如果你在编写一个 HTTP 服务器实现，你可能会想要子类化该类而不是*BaseCGIHandler*。

该类是*BaseHandler* 的子类。它重载了 __init__(), get_stdin(), get_stderr(), add_cgi_vars(), _write() 和 _flush() 方法以支持通过构造器显式地设置环境和流。所提供的环境和流存储在 stdin, stdout, stderr 和 environ 属性中。

stdout 的*write()* 方法应该完整写入每个数据块，与*io.BufferedIOBase* 一样。

class wsgiref.handlers.**BaseHandler**

这是适用于运行 WSGI 应用程序的抽象基类。每个实例将处理一个单独的 HTTP 请求，不过在原则上你也可以创建一个可针对多个请求重用的子类。

BaseHandler 实例只有一个方法提供给外部使用:

run(*app*)

运行指定的 WSGI 应用程序 *app*。

所有的*BaseHandler* 其他方法都是在运行应用程序过程中由该方法发起调用的，因此主要是为了允许定制运行过程。

以下方法必须在子类中被重载:

_write(*data*)

缓冲字节数据 *data* 以便传输给客户端。如果此方法真的传输了数据也是可以的；*BaseHandler* 只有在底层系统真有这样的区分时才会区分写入和刷新操作以提高效率。

_flush()

强制将缓冲的数据传输给客户端。如果此方法无任何操作也是可以的（例如，*_write()* 实际上已发送了数据）。

get_stdin()

返回一个适合被用作当前被处理的请求的 wsgi.input 的输入流对象。

get_stderr()

返回一个适合被用作当前所处理请求的 wsgi.errors 的输出流对象。

add_cgi_vars()

将当前请求的 CGI 变量插入到 environ 属性。

以下是一些你可能会想要重载的其他方法。但这只是个简略的列表，它不包括每个可被重载的方法。你应当在在尝试创建自定义的*BaseHandler* 子类之前参阅文档字符串和源代码来了解更多信息。

用于自定义 WSGI 环境的属性和方法:

wsgi_multithread

用于 wsgi.multithread 环境变量的值。它在*BaseHandler* 中默认为真值，但在其他子类中可能有不同的默认值（或是由构造器来设置）。

wsgi_multiprocess

用于 wsgi.multiprocess 环境变量的值。它在*BaseHandler* 中默认为真值，但在其他子类中可能有不同的默认值（或是由构造器来设置）。

wsgi_run_once

用于 wsgi.run_once 环境变量的值。它在*BaseHandler* 中默认为假值，但在*CGIHandler* 中默认为真值。

os_environ

要包括在每个请求的 WSGI 环境中的默认环境变量。在默认情况下，这是当*wsgiref.handlers* 被导入时的 os.environ 的一个副本，但也可以在类或实例层级上创建它们自己的子类。请注意该字典应当被当作是只读的，因为默认值会在多个类和实际之间共享。

server_software

如果设置了*origin_server* 属性，该属性的值会被用来设置默认的 SERVER_SOFTWARE WSGI 环境变量，并且还会被用来设置 HTTP 响应中默认的 Server: 标头。它会被不是 It is ignored for handlers (such as *BaseCGIHandler* and *CGIHandler*) that are not HTTP origin servers.

3.3 版更變: 名称"Python" 会被替换为实现专属的名称如"CPython", "Jython" 等等。

get_scheme()
返回当前请求所使用的 URL 方案。默认的实现会使用来自*wsgiref.util* 的 guess_scheme() 函数根据当前请求的 environ 变量来猜测方案应该是"http" 还是"https"。

setup_environ()
将 environ 属性设为填充了完整内容的 WSGI 环境。默认的实现会使用上述的所有方法和属性，加上*get_stdin()*, *get_stderr()* 和*add_cgi_vars()* 方法以及*wsgi_file_wrapper* 属性。它还会插入一个 SERVER_SOFTWARE 键，如果该键还不存在的话，只要*origin_server* 属性为真值并且设置了*server_software* 属性。

用于自定义异常处理的方法和属性:

log_exception(*exc_info*)
将 *exc_info* 元素记录到服务器日志。*exc_info* 是一个 (type, value, traceback) 元组。默认的实现会简单地将回溯信息写入请求的 wsgi.errors 流并刷新它。子类可以重载此方法来修改格式或重设输出目标，通过邮件将回溯信息发给管理员，或执行任何其他符合要求的动作。

traceback_limit
要包括在默认*log_exception()* 方法的回溯信息输出中的最大帧数。如果为 None，则会包括所有的帧。

error_output(*environ*, *start_response*)
此方法是一个为用户生成错误页面的 WSGI 应用程序。它仅当将标头发送给客户端之前发生错误时会被发起调用。

此方法可使用 sys.exc_info() 来访问当前错误信息，并应当在调用 *start_response* 时将该信息传递给它（如 **PEP 3333** 的"Error Handling" 部分所描述的）。

默认的实现只是使用*error_status*, *error_headers*, 和*error_body* 属性来生成一个输出页面。子类可以重载此方法来产生更动态化的错误输出。

但是请注意，从安全角度看来是不建议将诊断信息暴露给任何用户的；理想的做法是你应当通过特别处理来启用诊断输出，因此默认的实现并不包括这样的内容。

error_status
用于错误响应的 HTTP 状态。这应当是一个在 **PEP 3333** 中定义的字符串；它默认为代码 500 以相应的消息。

error_headers
用于错误响应的 HTTP 标头。这应当是由 WSGI 响应标头 ((name, value) 元组) 构成的列表，如 **PEP 3333** 所定义的。默认的列表只是将内容类型设为 text/plain。

error_body
错误响应体。这应当是一个 HTTP 响应体字节串。它默认为纯文本"A server error occurred. Please contact the administrator."

用于 **PEP 3333** 的" 可选的平台专属文件处理" 特性的方法和属性:

wsgi_file_wrapper
一个 wsgi.file_wrapper 工厂对象，或为 None。该属性的默认值是*wsgiref.util.FileWrapper* 类。

sendfile()
重载以实现平台专属的文件传输。此方法仅在应用程序的返回值是由*wsgi_file_wrapper* 属性指定的类的实例时会被调用。如果它能够成功地传输文件则应当返回真值，以使得默认的传输代码将不会被执行。此方法的默认实现只会返回假值。

杂项方法和属性:

origin_server
该属性在处理句柄的*_write()* 和*_flush()* 被用于同客户端直接通信而不是通过需要 HTTP 状态为某种特殊 Status: 标头的 CGI 类网关协议时应当被设为真值

该属性在*BaseHandler* 中默认为真值，但在*BaseCGIHandler* 和*CGIHandler* 中则为假值。

http_version
如果*origin_server* 为真值，则该字符串属性会被用来设置给客户端的响应的 HTTP 版本。它的默认值为 `"1.0"`。

`wsgiref.handlers.`**read_environ**`()`
将来自 `os.environ` 的 CGI 变量转码为 **PEP 3333** "bytes in unicode" 字符串，返回一个新的字典。此函数被*CGIHandler* 和*IISCGIHandler* 用来替代直接使用 `os.environ`，后者不一定在所有使用 Python 3 的平台和 Web 服务器上都符合 WSGI 标准 -- 特别是当 OS 的实际环境为 Unicode 时 (例如 Windows)，或者当环境为字节数据，但被 Python 用来解码它的系统编码格式不是 ISO-8859-1 时 (例如使用 UTF-8 的 Unix 系统)。

如果你要实现自己的基于 CGI 的处理句柄，你可能会想要使用此例程而不是简单地从 `os.environ` 直接拷贝值。

3.2 版新加入.

21.2.6 示例

这是一个可运行的"Hello World" WSGI 应用程序:

```
from wsgiref.simple_server import make_server

# Every WSGI application must have an application object - a callable
# object that accepts two arguments. For that purpose, we're going to
# use a function (note that you're not limited to a function, you can
# use a class for example). The first argument passed to the function
# is a dictionary containing CGI-style environment variables and the
# second variable is the callable object.
def hello_world_app(environ, start_response):
    status = '200 OK'  # HTTP Status
    headers = [('Content-type', 'text/plain; charset=utf-8')]  # HTTP Headers
    start_response(status, headers)

    # The returned object is going to be printed
    return [b"Hello World"]

with make_server('', 8000, hello_world_app) as httpd:
    print("Serving on port 8000...")

    # Serve until process is killed
    httpd.serve_forever()
```

一个发布当前目录的 WSGI 应用程序示例，接受通过命令行指定可选的目录和端口号 (默认值: 8000):

```
#!/usr/bin/env python3
'''
Small wsgiref based web server. Takes a path to serve from and an
optional port number (defaults to 8000), then tries to serve files.
Mime types are guessed from the file names, 404 errors are raised
if the file is not found. Used for the make serve target in Doc.
'''
import sys
import os
import mimetypes
from wsgiref import simple_server, util
```

(下页继续)

(繼續上一頁)

```
def app(environ, respond):

    fn = os.path.join(path, environ['PATH_INFO'][1:])
    if '.' not in fn.split(os.path.sep)[-1]:
        fn = os.path.join(fn, 'index.html')
    type = mimetypes.guess_type(fn)[0]

    if os.path.exists(fn):
        respond('200 OK', [('Content-Type', type)])
        return util.FileWrapper(open(fn, "rb"))
    else:
        respond('404 Not Found', [('Content-Type', 'text/plain')])
        return [b'not found']

if __name__ == '__main__':
    path = sys.argv[1] if len(sys.argv) > 1 else os.getcwd()
    port = int(sys.argv[2]) if len(sys.argv) > 2 else 8000
    httpd = simple_server.make_server('', port, app)
    print("Serving {} on port {}, control-C to stop".format(path, port))
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print("Shutting down.")
        httpd.server_close()
```

21.3 `urllib` --- URL 处理模块

源代码: Lib/urllib/

`urllib` 是一个收集了多个涉及 URL 的模块的包：

- *`urllib.request`* 打开和读取 URL
- *`urllib.error`* 包含*`urllib.request`* 抛出的异常
- *`urllib.parse`* 用于解析 URL
- *`urllib.robotparser`* 用于解析 `robots.txt` 文件

21.4 `urllib.request` --- 用于打开 URL 的可扩展库

源码： Lib/urllib/request.py

`urllib.request` 模块定义了适用于在各种复杂情况下打开 URL（主要为 HTTP）的函数和类 --- 例如基本认证、摘要认证、重定向、cookies 及其它。

也参考:

对于更高级别的 HTTP 客户端接口，建议使用 Requests 。

`urllib.request` 模块定义了以下函数：

urllib.request.**urlopen**(*url*, *data=None*[, *timeout*], *, *cafile=None*, *capath=None*, *cadefault=False*, *context=None*)

打开统一资源定位符 *url*，可以是一个字符串或一个*Request* 对象。

data 必须是一个对象，用于给出要发送到服务器的附加数据，若不需要发送数据则为 None。详情请参阅*Request* 。

urllib.request 模块采用 HTTP/1.1 协议，并且在其 HTTP 请求中包含 Connection:close 头部信息。

timeout 为可选参数，用于指定阻塞操作（如连接尝试）的超时时间，单位为秒。如未指定，将使用全局默认超时参数）。本参数实际仅对 HTTP、HTTPS 和 FTP 连接有效。

如果给定了 *context* 参数，则必须是一个*ssl.SSLContext* 实例，用于描述各种 SSL 参数。更多详情请参阅*HTTPSConnection* 。

cafile 和 *capath* 为可选参数，用于为 HTTPS 请求指定一组受信 CA 证书。*cafile* 应指向包含 CA 证书的单个文件，*capath* 则应指向哈希证书文件的目录。更多信息可参阅*ssl.SSLContext.load_verify_locations()* 。

参数 *cadefault* 将被忽略。

本函数总会返回一个对象，该对象可作为*context manager* 使用，带有 *url*、*headers* 和 *status* 属性。有关这些属性的更多详细信息，请参阅*urllib.response.addinfourl* 。

对于 HTTP 和 HTTPS 的 URL 而言，本函数将返回一个稍经修改的*http.client.HTTPResponse* 对象。除了上述 3 个新的方法之外，还有 msg 属性包含了与*reason* 属性相同的信息---服务器返回的原因描述文字，而不是*HTTPResponse* 的文档所述的响应头部信息。

对于 FTP、文件、数据的 URL，以及由传统的*URLopener* 和*FancyURLopener* 类处理的请求，本函数将返回一个*urllib.response.addinfourl* 对象。

协议发生错误时，将会引发*URLError* 。

请注意，如果没有处理函数对请求进行处理，则有可能会返回 None 。尽管默认安装的全局*OpenerDirector* 会用*UnknownHandler* 来确保不会发生这种情况。

此外，如果检测到设置了代理（比如设置了 http_proxy 之类的环境变量），默认会安装*ProxyHandler* 并确保通过代理处理请求。

Python 2.6 以下版本中留存的 urllib.urlopen 函数已停止使用了；*urllib.request.urlopen()* 对应于传统的 urllib2.urlopen 。对代理服务的处理是通过将字典参数传给 urllib.urlopen 来完成的，可以用*ProxyHandler* 对象获取到代理处理函数。

默认会触发一条审计事件 urllib.Request ，参数 fullurl 、data 、headers 、method 均取自请求对象。

3.2 版更變: 增加了 *cafile* 与 *capath*。

3.2 版更變: 只要条件允许（指*ssl.HAS_SNI* 为真），现在能够支持 HTTPS 虚拟主机。

3.2 版新加入: *data* 可以是一个可迭代对象。

3.3 版更變: 增加了 *cadefault*。

3.4.3 版更變: 增加了 *context*。

3.6 版後已𝔽用: *cafile* 、*capath* 和 *cadefault* 已废弃，转而推荐使用 *context*。请改用*ssl.SSLContext.load_cert_chain()* 或让*ssl.create_default_context()* 选取系统信任的 CA 证书。

urllib.request.**install_opener**(*opener*)

安装一个*OpenerDirector* 实例，作为默认的全局打开函数。仅当 urlopen 用到该打开函数时才需要安装；否则，只需调用*OpenerDirector.open()* 而不是*urlopen()*。代码不会检查是否真的属于*OpenerDirector* 类，所有具备适当接口的类都能适用。

urllib.request.**build_opener**([*handler*, ...])
返回一个*OpenerDirector*实例，以给定顺序把处理函数串联起来。处理函数可以是*BaseHandler*的实例，也可以是*BaseHandler*的子类（这时构造函数必须允许不带任何参数的调用）。以下类的实例将位于处理函数之前，除非处理函数已包含这些类、其实例或其子类：*ProxyHandler*（如果检测到代理设置）、*UnknownHandler*、*HTTPHandler*、*HTTPDefaultErrorHandler*、*HTTPRedirectHandler*、*FTPHandler*、*FileHandler*、*HTTPErrorProcessor*。

若 Python 安装时已带了 SSL 支持（指可以导入*ssl*模块），则还会加入*HTTPSHandler*。

A *BaseHandler* subclass may also change its `handler_order` attribute to modify its position in the handlers list.

urllib.request.**pathname2url**(*path*)
将路径名 *path* 从路径本地写法转换为 URL 路径部分所采用的格式。本函数不会生成完整的 URL。返回值将会用*quote()*函数加以编码。

urllib.request.**url2pathname**(*path*)
从百分号编码的 URL 中将 *path* 部分转换为本地路径的写法。本函数不接受完整的 URL，并利用*unquote()*函数对 *path* 进行解码。

urllib.request.**getproxies**()
This helper function returns a dictionary of scheme to proxy server URL mappings. It scans the environment for variables named `<scheme>_proxy`, in a case insensitive approach, for all operating systems first, and when it cannot find it, looks for proxy information from System Configuration for macOS and Windows Systems Registry for Windows. If both lowercase and uppercase environment variables exist (and disagree), lowercase is preferred.

備F: 如果存在环境变量 `REQUEST_METHOD`，通常表示脚本运行于 CGI 环境中，则环境变量 `HTTP_PROXY`（大写的 `_PROXY`）将会被忽略。这是因其可以由客户端用 HTTP 头部信息 "Proxy:" 注入。若要在 CGI 环境中使用 HTTP 代理，请显式使用 `ProxyHandler，或确保变量名称为小写（或至少是 `_proxy 后缀）。

提供了以下类：

class urllib.request.**Request**(*url*, *data=None*, *headers={}*, *origin_req_host=None*, *unverifiable=False*, *method=None*)
URL 请求对象的抽象类。

url 应为包含合法 URL 的字符串。

data 必须是一个对象，用于给定发往服务器的附加数据，若无需此类数据则为 `None`。目前唯一用到 *data* 的只有 HTTP 请求。支持的对象类型包括字节串、类文件对象和可遍历的类字节串对象。如果没有提供 `Content-Length` 和 `Transfer-Encoding` 头部字段，*HTTPHandler* 会根据 *data* 的类型设置这些头部字段。`Content-Length` 将用于发送字节对象，而 **RFC 7230** 第 3.3.1 节中定义的 `Transfer-Encoding: chunked` 将用于发送文件和其他可遍历对象。

对于 HTTP POST 请求方法而言，*data* 应该是标准 *application/x-www-form-urlencoded* 格式的缓冲区。*urllib.parse.urlencode()* 函数的参数为映射对象或二元组序列，并返回一个该编码格式的 ASCII 字符串。在用作 *data* 参数之前，应将其编码为字节串。

headers should be a dictionary, and will be treated as if *add_header()* was called with each key and value as arguments. This is often used to "spoof" the `User-Agent` header value, which is used by a browser to identify itself -- some HTTP servers only allow requests coming from common browsers as opposed to scripts. For example, Mozilla Firefox may identify itself as `"Mozilla/5.0 (X11; U; Linux i686) Gecko/20071127 Firefox/2.0.0.11"`, while *urllib*'s default user agent string is `"Python-urllib/2.6"` (on Python 2.6). All header keys are sent in camel case.

如果给出了 *data* 参数，则应当包含合适的 `Content-Type` 头部信息。若未提供且 *data* 不是 None，则会把 `Content-Type: application/x-www-form-urlencoded` 加入作为默认值。

接下来的两个参数，只对第三方 HTTP cookie 的处理才有用：

origin_req_host 应为发起初始会话的请求主机，定义参见 **RFC 2965** 。默认指为 “http.cookiejar.request_host(self)“ 。这是用户发起初始请求的主机名或 IP 地址。假设请求是针对 HTML 文档中的图片数据发起的，则本属性应为对包含图像的页面发起请求的主机。

unverifiable 应该标示出请求是否无法验证，定义参见 **RFC 2965** 。默认值为 `False` 。所谓无法验证的请求，是指用户没有机会对请求的 URL 做验证。例如，如果请求是针对 HTML 文档中的图像，用户没有机会去许可能自动读取图像，则本参数应为 True。

method 应为字符串，标示要采用的 HTTP 请求方法（例如 `'HEAD'` ）。如果给出本参数，其值会存储在*method* 属性中，并由*get_method()* 使用。如果 *data* 为 “None“ 则默认值为 `'GET'` ，否则为 `'POST'`。子类可以设置*method* 属性来标示不同的默认请求方法。

備𝔽: 如果 data 对象无法分多次传递其内容（比如文件或只能生成一次内容的可迭代对象）并且由于 HTTP 重定向或身份验证而发生请求重试行为，则该请求不会正常工作。*data* 是紧挨着头部信息发送给 HTTP 服务器的。现有库不支持 HTTP 100-continue 的征询。

3.3 版更變: Request 类增加了*Request.method* 参数。

3.4 版更變: 默认*Request.method* 可以在类中标明。

3.6 版更變: 如果给出了 `Content-Length` ，且 *data* 既不为 `None` 也不是字节串对象，则不会触发错误。而会退而求其次采用分块传输的编码格式。

class urllib.request.**OpenerDirector**
OpenerDirector 类通过串接在一起的*BaseHandler* 打开 URL，并负责管理 handler 链及从错误中恢复。

class urllib.request.**BaseHandler**
这是所有已注册 handler 的基类，只做了简单的注册机制。

class urllib.request.**HTTPDefaultErrorHandler**
为 HTTP 错误响应定义的默认 handler，所有出错响应都会转为*HTTPError* 异常。

class urllib.request.**HTTPRedirectHandler**
一个用于处理重定向的类。

class urllib.request.**HTTPCookieProcessor**(*cookiejar=None*)
一个用于处理 HTTP Cookies 的类。

class urllib.request.**ProxyHandler**(*proxies=None*)
Cause requests to go through a proxy. If *proxies* is given, it must be a dictionary mapping protocol names to URLs of proxies. The default is to read the list of proxies from the environment variables `<protocol>_proxy`. If no proxy environment variables are set, then in a Windows environment proxy settings are obtained from the registry's Internet Settings section, and in a macOS environment proxy information is retrieved from the System Configuration Framework.

若要禁用自动检测出来的代理，请传入空的字典对象。

环境变量 `no_proxy` 可用于指定不必通过代理访问的主机；应为逗号分隔的主机名后缀列表，可加上 `:port` ，例如 `cern.ch,ncsa.uiuc.edu,some.host:8080` 。

備𝔽: 如果设置了 `REQUEST_METHOD` 变量，则会忽略 `HTTP_PROXY` ；参阅*getproxies()* 文档。

---

class urllib.request.**HTTPPasswordMgr**
维护 `(realm, uri) -> (user, password)` 映射数据库。

class urllib.request.**HTTPPasswordMgrWithDefaultRealm**
维护 (realm, uri) -> (user, password) 映射数据库。realm 为 None 视作全匹配，若没有其他合适的安全区域就会检索它。

class urllib.request.**HTTPPasswordMgrWithPriorAuth**
HTTPPasswordMgrWithDefaultRealm 的一个变体，也带有 uri -> is_authenticated 映射数据库。可被 BasicAuth 处理函数用于确定立即发送身份认证凭据的时机，而不是先等待 401 响应。

3.5 版新加入.

class urllib.request.**AbstractBasicAuthHandler**(*password_mgr=None*)
这是一个帮助完成 HTTP 身份认证的混合类，对远程主机和代理都适用。参数 *password_mgr* 应与*HTTPPasswordMgr* 兼容；关于必须支持哪些接口，请参阅*HTTPPasswordMgr 对象* 对象的章节。如果 *password_mgr* 还提供 is_authenticated 和 update_authenticated 方法（请参阅*HTTPPasswordMgrWithPriorAuth 对象* 对象），则 handler 将对给定 URI 用到 is_authenticated 的结果，来确定是否随请求发送身份认证凭据。如果该 URI 的 is_authenticated 返回 True，则发送凭据。如果 is_authenticated 为 False，则不发送凭据，然后若收到 401 响应，则使用身份认证凭据重新发送请求。如果身份认证成功，则调用 update_authenticated 设置该 URI 的 is_authenticated 为 True，这样后续对该 URI 或其所有父 URI 的请求将自动包含该身份认证凭据。

3.5 版新加入: 增加了对 is_authenticated 的支持。

class urllib.request.**HTTPBasicAuthHandler**(*password_mgr=None*)
处理远程主机的身份认证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。如果给出错误的身份认证方式，HTTPBasicAuthHandler 将会触发*ValueError* 。

class urllib.request.**ProxyBasicAuthHandler**(*password_mgr=None*)
处理有代理服务时的身份认证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。

class urllib.request.**AbstractDigestAuthHandler**(*password_mgr=None*)
这是一个帮助完成 HTTP 身份认证的混合类，对远程主机和代理都适用。参数 *password_mgr* 应与*HTTPPasswordMgr* 兼容；关于必须支持哪些接口，请参阅*HTTPPasswordMgr 对象* 的章节。

class urllib.request.**HTTPDigestAuthHandler**(*password_mgr=None*)
处理远程主机的身份认证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。如果同时添加了 digest 身份认证 handler 和 basic 身份认证 handler，则会首先尝试 digest 身份认证。如果 digest 身份认证再返回 40x 响应，会再发送到 basic 身份验证 handler 进行处理。如果给出 Digest 和 Basic 之外的身份认证方式，本 handler 方法将会触发*ValueError* 。

3.3 版更變: 碰到不支持的认证方式时，将会触发*ValueError* 。

class urllib.request.**ProxyDigestAuthHandler**(*password_mgr=None*)
处理有代理服务时的身份认证。*password_mgr* 应与*HTTPPasswordMgr* 兼容；有关哪些接口是必须支持的，请参阅*HTTPPasswordMgr 对象* 章节。

class urllib.request.**HTTPHandler**
用于打开 HTTP URL 的 handler 类。

class urllib.request.**HTTPSHandler**(*debuglevel=0*, *context=None*, *check_hostname=None*)
用于打开 HTTPS URL 的 handler 类。*context* 和 *check_hostname* 的含义与*http.client.HTTPSConnection* 的一样。

3.2 版更變: 添加 *context* 和 *check_hostname* 参数。

class urllib.request.**FileHandler**
打开本地文件。

class urllib.request.**DataHandler**
打开数据 URL。

3.4 版新加入.

class urllib.request.**FTPHandler**
打开 FTP URL。

class urllib.request.**CacheFTPHandler**
打开 FTP URL，并将打开的 FTP 连接存入缓存，以便最大程度减少延迟。

class urllib.request.**UnknownHandler**
处理所有未知类型 URL 的兜底类。

class urllib.request.**HTTPErrorProcessor**
处理出错的 HTTP 响应。

21.4.1 Request 对象

以下方法介绍了*Request* 的公开接口，因此子类可以覆盖所有这些方法。这里还定义了几个公开属性，客户端可以利用这些属性了解经过解析的请求。

Request.**full_url**
传给构造函数的原始 URL。

3.4 版更變.

Request.full_url 是一个带有 setter、getter 和 deleter 的属性。读取*full_url* 属性将会返回附带片段（fragment）的初始请求 URL。

Request.**type**
URI 方式。

Request.**host**
URI 权限，通常是整个主机，但也有可能带有冒号分隔的端口号。

Request.**origin_req_host**
请求的原始主机，不含端口。

Request.**selector**
URI 路径。若*Request* 使用代理，selector 将会是传给代理的完整 URL。

Request.**data**
请求的数据体，未给出则为 None 。

3.4 版更變: 现在如果修改*Request.data* 的值，则会删除之前设置或计算过的 “Content-Length” 头部信息。

Request.**unverifiable**
布尔值，标识本请求是否属于 **RFC 2965** 中定义的无法验证的情况。

Request.**method**
要采用的 HTTP 请求方法。默认为*None*，表示*get_method()* 将对方法进行正常处理。设置本值可以覆盖*get_method()* 中的默认处理过程，设置方式可以是在*Request* 的子类中给出默认值，也可以通过 *method* 参数给*Request* 构造函数传入一个值。

3.3 版新加入.

3.4 版更變: 现在可以在子类中设置默认值；而之前只能通过构造函数的实参进行设置。

Request.**get_method**()
返回表示 HTTP 请求方法的字符串。如果*Request.method*不为 None，则返回其值。否则若*Request.data*为则返回 'GET'，不为 None 则返回 'POST'。只对 HTTP 请求有效。

3.3 版更變: 现在 get_method 会兼顾*Request.method*的值。

Request.**add_header**(*key*, *val*)
Add another header to the request. Headers are currently ignored by all handlers except HTTP handlers, where they are added to the list of headers sent to the server. Note that there cannot be more than one header with the same name, and later calls will overwrite previous calls in case the *key* collides. Currently, this is no loss of HTTP functionality, since all headers which have meaning when used more than once have a (header-specific) way of gaining the same functionality using only one header. Note that headers added using this method are also added to redirected requests.

Request.**add_unredirected_header**(*key*, *header*)
添加一项不会被加入重定向请求的头部信息。

Request.**has_header**(*header*)
返回本实例是否带有命名头部信息（对常规数据和非重定向数据都会检测）。

Request.**remove_header**(*header*)
从本请求实例中移除指定命名的头部信息（对常规数据和非重定向数据都会检测）。

3.4 版新加入.

Request.**get_full_url**()
返回构造器中给定的 URL。

3.4 版更變.

返回*Request.full_url*

Request.**set_proxy**(*host*, *type*)
连接代理服务器，为当前请求做准备。*host* 和 *type* 将会取代本实例中的对应值，selector 将会是构造函数中给出的初始 URL。

Request.**get_header**(*header_name*, *default=None*)
返回给定头部信息的数据。如果该头部信息不存在，返回默认值。

Request.**header_items**()
返回头部信息，形式为（名称, 数据）的元组列表。

3.4 版更變: 自 3.3 起已弃用的下列方法已被删除：add_data、has_data、get_data、get_type、get_host、get_selector、get_origin_req_host 和 is_unverifiable 。

21.4.2 OpenerDirector 对象

OpenerDirector 实例有以下方法：

OpenerDirector.**add_handler**(*handler*)
handler 应为*BaseHandler*的实例。将检索以下类型的方法，并将其添加到对应的处理链中（注意 HTTP 错误是特殊情况）。请注意，下文中的 *protocol* 应替换为要处理的实际协议，例如 http_response() 将是 HTTP 协议响应处理函数。并且 *type* 也应替换为实际的 HTTP 代码，例如 http_error_404() 将处理 HTTP 404 错误。

- <protocol>_open() —表明该 handler 知道如何打开 *protocol* 协议的 URL。

  更多信息请参阅*BaseHandler.<protocol>_open()* 。

- http_error_<type>() —表明该 handler 知道如何处理代码为 *type* 的 HTTP 错误。

  更多信息请参阅*BaseHandler.http_error_<nnn>()* 。

- `<protocol>_error()` —表明该 handler 知道如何处理来自协议为 *protocol*（非 `http`）的错误。
- `<protocol>_request()` —表明该 handler 知道如何预处理协议为 *protocol* 的请求。

  更多信息请参阅*BaseHandler.<protocol>_request()* 。
- `<protocol>_response()` —表明该 handler 知道如何后处理协议为 *protocol* 的响应。

  更多信息请参阅*BaseHandler.<protocol>_response()* 。

`OpenerDirector.`**`open`**(*url*, *data*=*None*[, *timeout*])
Open the given *url* (which can be a request object or a string), optionally passing the given *data*. Arguments, return values and exceptions raised are the same as those of *urlopen()* (which simply calls the *open()* method on the currently installed global *OpenerDirector*). The optional *timeout* parameter specifies a timeout in seconds for blocking operations like the connection attempt (if not specified, the global default timeout setting will be used). The timeout feature actually works only for HTTP, HTTPS and FTP connections.

`OpenerDirector.`**`error`**(*proto*, **args*)
处理给定协议的错误。将用给定的参数（协议相关）调用已注册的给定协议的错误处理程序。HTTP 协议是特殊情况，采用 HTTP 响应码来确定具体的错误处理程序；请参考 handler 类的 `http_error_<type>()` 方法。

返回值和异常均与*urlopen()* 相同。

OpenerDirector 对象分 3 个阶段打开 URL：

每个阶段中调用这些方法的次序取决于 handler 实例的顺序。

1. 每个具有 `_request()` 这类方法的 handler 都会调用本方法对请求进行预处理。
2. 调用具有 `_open()` 这类方法的 handler 来处理请求。当 handler 返回非*None* 值（即响应）或引发异常（通常是*URLError*）时，本阶段结束。本阶段能够传播异常。

   事实上，以上算法首先会尝试名为 `default_open()` 的。如果这些方法全都返回*None*，则会对名为 `_open()` 的这类方法重复过程。如果又都返回*None*，则再对名为 `unknown_open()` 的方法重复过程。

   请注意，这些方法的代码可能会调用*OpenerDirector* 父实例的*open()* 和*error()* 方法。
3. 每个具有 `_response()` 这类方法的 handler 都会调用这些方法，以对响应进行后处理。

21.4.3 BaseHandler 对象

BaseHandler 对象提供了一些直接可用的方法，以及其他一些可供派生类使用的方法。以下是可供直接使用的方法：

`BaseHandler.`**`add_parent`**(*director*)
将 director 加为父 OpenerDirector。

`BaseHandler.`**`close`**()
移除所有父 OpenerDirector。

以下属性和方法仅供*BaseHandler* 的子类使用：

備𝔽： 以下约定已被采纳：定义 `<protocol>_request()` 或 `<protocol>_response()` 方法的子类应命名为 `*Processor`；所有其他子类都应命名为 `*Handler` 。

`BaseHandler.`**`parent`**
一个可用的*OpenerDirector*，可用于以其他协议打开 URI，或处理错误。

BaseHandler.**default_open**(*req*)
本方法在*BaseHandler* 中 未予定义，但其子类若要捕获所有 URL 则应进行定义。

This method, if implemented, will be called by the parent *OpenerDirector*. It should return a file-like object as described in the return value of the *open()* method of *OpenerDirector*, or None. It should raise *URLError*, unless a truly exceptional thing happens (for example, *MemoryError* should not be mapped to URLError).

本方法将会在所有协议的 open 方法之前被调用。

BaseHandler.<protocol>_open(req)
本方法在*BaseHandler* 中 未予定义，但其子类若要处理给定协议的 URL 则应进行定义。

若定义了本方法，将会被父*OpenerDirector* 对象调用。返回值和 default_open() 的一样。

BaseHandler.**unknown_open**(*req*)
本方法在*BaseHandler* 中 未予定义，但其子类若要捕获并打开所有未注册 handler 的 URL，则应进行定义。

若实现了本方法，将会被*parent* 属性指向的父*OpenerDirector* 调用。返回值和*default_open()* 的一样。

BaseHandler.**http_error_default**(*req*, *fp*, *code*, *msg*, *hdrs*)
本方法在*BaseHandler* 中 未予定义，但其子类若要为所有未定义 handler 的 HTTP 错误提供一个兜底方法，则应进行重写。*OpenerDirector* 会自动调用本方法，获取错误信息，而通常在其他时候不应去调用。

req 会是一个*Request* 对象，*fp* 是一个带有 HTTP 错误体的文件类对象，*code* 是三位数的错误码，*msg* 是供用户阅读的解释信息，*hdrs* 则是一个包含出错头部信息的字典对象。

返回值和触发的异常应与*urlopen()* 的相同。

BaseHandler.http_error_<nnn>(req, fp, code, msg, hdrs)
nnn 应为三位数的 HTTP 错误码。本方法在*BaseHandler* 中也未予定义，但当子类的实例发生代码为 *nnn* 的 HTTP 错误时，若方法存在则会被调用。

子类应该重写本方法，以便能处理相应的 HTTP 错误。

参数、返回值和触发的异常应与 http_error_default() 相同。

BaseHandler.<protocol>_request(req)
本方法在*BaseHandler* 中 未予定义，但其子类若要对给定协议的请求进行预处理，则应进行定义。

若实现了本方法，将会被父*OpenerDirector* 调用。*req* 将为*Request* 对象。返回值应为*Request* 对象。

BaseHandler.<protocol>_response(req, response)
本方法在*BaseHandler* 中 未予定义，但其子类若要对给定协议的请求进行后处理，则应进行定义。

若实现了本方法，将会被父*OpenerDirector* 调用。*req* 将为*Request* 对象。*response* 应实现与*urlopen()* 返回值相同的接口。返回值应实现与*urlopen()* 返回值相同的接口。

21.4.4 HTTPRedirectHandler 对象

備🄴: 某些 HTTP 重定向操作需要本模块的客户端代码提供的功能。这时会触发*HTTPError*。有关各种重定向代码的确切含义，请参阅 **RFC 2616** 。

如果给到 HTTPRedirectHandler 的重定向 URL 不是 HTTP、HTTPS 或 FTP URL，则出于安全考虑将触发 HTTPError 异常。

HTTPRedirectHandler.**redirect_request**(*req*, *fp*, *code*, *msg*, *hdrs*, *newurl*)
返回*Request* 或 None 对象作为重定向行为的响应。当服务器接收到重定向请求时，http_error_30*() 方法的默认实现代码将会调用本方法。如果确实应该发生重定向，则返回一个新的*Request* 对象，使得 http_error_30*() 能重定向至 *newurl*。否则，若没有 handler 会处理此 URL，则会引发*HTTPError*；或者本方法不能处理但或许会有其他 handler 会处理，则返回 None 。

備🄵: 本方法的默认实现代码并未严格遵循 **RFC 2616**，即 POST 请求的 301 和 302 响应不得在未经用户确认的情况下自动进行重定向。现实情况下，浏览器确实允许自动重定向这些响应，将 POST 更改为 GET ，于是默认实现代码就复现了这种处理方式。

HTTPRedirectHandler.**http_error_301**(*req*, *fp*, *code*, *msg*, *hdrs*)
重定向到 Location: 或 URI: URL。这个方法会在得到 HTTP 'moved permanently' 响应时由上级*OpenerDirector* 来调用。

HTTPRedirectHandler.**http_error_302**(*req*, *fp*, *code*, *msg*, *hdrs*)
与*http_error_301()* 相同，不过是发生 "found" 响应时的调用。

HTTPRedirectHandler.**http_error_303**(*req*, *fp*, *code*, *msg*, *hdrs*)
与*http_error_301()* 相同，不过是发生 "see other" 响应时的调用。

HTTPRedirectHandler.**http_error_307**(*req*, *fp*, *code*, *msg*, *hdrs*)
与*http_error_301()* 相同，不过是发生 "temporary redirect" 响应时的调用。

21.4.5 HTTPCookieProcessor 对象

HTTPCookieProcessor 的实例具备一个属性:

HTTPCookieProcessor.**cookiejar**
cookie 存放在*http.cookiejar.CookieJar* 中。

21.4.6 ProxyHandler 对象

ProxyHandler.<protocol>_open(request)
ProxyHandler 将为每种 *protocol* 准备一个 _open() 方法，在构造函数给出的 *proxies* 字典中包含对应的代理服务器信息。通过调用 request.set_proxy()，本方法将把请求转为通过代理服务器，并会调用 handler 链中的下一个 handler 来完成对应的协议处理。

21.4.7 HTTPPasswordMgr 对象

以下方法*HTTPPasswordMgr* 和*HTTPPasswordMgrWithDefaultRealm* 对象均有提供。

HTTPPasswordMgr.**add_password**(*realm*, *uri*, *user*, *passwd*)
uri 可以是单个 URI，也可以是 URI 列表。*realm*、*user* 和 *passwd* 必须是字符串。这使得在为 *realm* 和超级 URI 进行身份认证时，`(user, passwd)` 可用作认证令牌。

HTTPPasswordMgr.**find_user_password**(*realm*, *authuri*)
为给定 realm 和 URI 获取用户名和密码。如果没有匹配的用户名和密码，本方法将会返回 `(None, None)`。

对于*HTTPPasswordMgrWithDefaultRealm* 对象，如果给定 *realm* 没有匹配的用户名和密码，将搜索 realm `None`。

21.4.8 HTTPPasswordMgrWithPriorAuth 对象

这是*HTTPPasswordMgrWithDefaultRealm* 的扩展，以便对那些需要一直发送认证凭证的 URI 进行跟踪。

HTTPPasswordMgrWithPriorAuth.**add_password**(*realm*, *uri*, *user*, *passwd*, *is_authenticated=False*)
realm、*uri*、*user*、*passwd* 的含义与*HTTPPasswordMgr.add_password()* 的相同。*is_authenticated* 为给定 URI 或 URI 列表设置 `is_authenticated` 标志的初始值。如果 *is_authenticated* 设为 `True`，则会忽略 *realm*。

HTTPPasswordMgrWithPriorAuth.**find_user_password**(*realm*, *authuri*)
与*HTTPPasswordMgrWithDefaultRealm* 对象的相同。

HTTPPasswordMgrWithPriorAuth.**update_authenticated**(*self*, *uri*, *is_authenticated=False*)
更新给定 *uri* 或 URI 列表的 `is_authenticated` 标志。

HTTPPasswordMgrWithPriorAuth.**is_authenticated**(*self*, *authuri*)
返回给定 URI `is_authenticated` 标志的当前状态。

21.4.9 AbstractBasicAuthHandler 对象

AbstractBasicAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)
通过获取用户名和密码并重新尝试请求，以处理身份认证请求。*authreq* 应该是请求中包含 realm 的头部信息名称，*host* 指定了需要进行身份认证的 URL 和路径，*req* 应为 (已失败的) *Request* 对象，*headers* 应该是出错的头部信息。

host 要么是一个认证信息（例如 `"python.org"`），要么是一个包含认证信息的 URL（如 “”http://python.org/”“ ）。不论是哪种格式，认证信息中都不能包含用户信息（因此，`"python.org"` 和 `"python.org:80"` 没问题，而 `"joe:password@python.org"` 则不行）。

21.4.10 HTTPBasicAuthHandler 对象

HTTPBasicAuthHandler.**http_error_401**(*req*, *fp*, *code*, *msg*, *hdrs*)
如果可用的话，请用身份认证信息重试请求。

21.4.11 ProxyBasicAuthHandler 对象

ProxyBasicAuthHandler.**http_error_407**(*req*, *fp*, *code*, *msg*, *hdrs*)
如果可用的话，请用身份认证信息重试请求。

21.4.12 AbstractDigestAuthHandler 对象

AbstractDigestAuthHandler.**http_error_auth_reqed**(*authreq*, *host*, *req*, *headers*)
authreq 应为请求中有关 realm 的头部信息名称，*host* 应为需要进行身份认证的主机，*req* 应为（已失败的）*Request* 对象，*headers* 则应为出错的头部信息。

21.4.13 HTTPDigestAuthHandler 对象

HTTPDigestAuthHandler.**http_error_401**(*req*, *fp*, *code*, *msg*, *hdrs*)
如果可用的话，请用身份认证信息重试请求。

21.4.14 ProxyDigestAuthHandler 对象

ProxyDigestAuthHandler.**http_error_407**(*req*, *fp*, *code*, *msg*, *hdrs*)
如果可用的话，请用身份认证信息重试请求。

21.4.15 HTTPHandler 对象

HTTPHandler.**http_open**(*req*)
发送 HTTP 请求，根据 `req.has_data()` 的结果，可能是 GET 或 POST 格式。

21.4.16 HTTPSHandler 对象

HTTPSHandler.**https_open**(*req*)
发送 HTTPS 请求，根据 `req.has_data()` 的结果，可能是 GET 或 POST 格式。

21.4.17 FileHandler 对象

FileHandler.**file_open**(*req*)
若无主机名或主机名为 `'localhost'`，则打开本地文件。

3.2 版更變: 本方法仅适用于本地主机名。如果给出的是远程主机名，将会触发*URLError*。

21.4.18 DataHandler 对象

DataHandler.**data_open**(*req*)
读取内含数据的 URL。这种 URL 本身包含了经过编码的数据。**RFC 2397** 中给出了数据 URL 的语法定义。目前的代码库将忽略经过 base64 编码的数据 URL 中的空白符，因此 URL 可以放入任何源码文件中。如果数据 URL 的 base64 编码尾部缺少填充，即使某些浏览器不介意，但目前的代码库仍会引发*ValueError*。

21.4.19 FTPHandler 对象

FTPHandler.**ftp_open**(*req*)
打开由 *req* 给出的 FTP 文件。登录时的用户名和密码总是为空。

21.4.20 CacheFTPHandler 对象

CacheFTPHandler 对象即为加入以下方法的*FTPHandler* 对象：

CacheFTPHandler.**setTimeout**(*t*)
设置连接超时为 *t* 秒。

CacheFTPHandler.**setMaxConns**(*m*)
设置已缓存的最大连接数为 *m* 。

21.4.21 UnknownHandler 对象

UnknownHandler.**unknown_open**()
触发*URLError* 异常。

21.4.22 HTTPErrorProcessor 对象

HTTPErrorProcessor.**http_response**(*request*, *response*)
处理出错的 HTTP 响应。

对于 200 错误码，响应对象将立即返回。

对于 200 以外的错误码，只通过*OpenerDirector.error()* 将任务传给 handler 的 http_error_<type>() 方法。如果最终没有 handler 处理错误，*HTTPDefaultErrorHandler* 将触发*HTTPError*。

HTTPErrorProcessor.**https_response**(*request*, *response*)
HTTPS 出错响应的处理。

与*http_response()* 方法相同。

21.4.23 示例

urllib-howto 中给出了更多的示例。

以下示例将读取 python.org 主页并显示前 300 个字节的内容：

```
>>> import urllib.request
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(300))
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">\n\n\n<html
xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">\n\n<head>\n
<meta http-equiv="content-type" content="text/html; charset=utf-8" />\n
<title>Python Programming '
```

请注意，urlopen 将返回字节串对象。这是因为 urlopen 无法自动确定由 HTTP 服务器收到的字节流的编码。通常，只要能确定或猜出编码格式，就应将返回的字节串解码为字符串。

下述 W3C 文档 https://www.w3.org/International/O-charset列出了可用于指明（X）HTML 或 XML 文档编码信息的多种方案。

python.org 网站已在 meta 标签中指明，采用的是 *utf-8* 编码，因此这里将用同样的格式对字节串进行解码。

```
>>> with urllib.request.urlopen('http://www.python.org/') as f:
...     print(f.read(100).decode('utf-8'))
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

不用*context manager* 方法也能获得同样的结果：

```
>>> import urllib.request
>>> f = urllib.request.urlopen('http://www.python.org/')
>>> print(f.read(100).decode('utf-8'))
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtm
```

以下示例将会把数据流发送给某 CGI 的 stdin，并读取返回数据。请注意，该示例只能工作于 Python 装有 SSL 支持的环境。

```
>>> import urllib.request
>>> req = urllib.request.Request(url='https://localhost/cgi-bin/test.cgi',
...                       data=b'This data is passed to stdin of the CGI')
>>> with urllib.request.urlopen(req) as f:
...     print(f.read().decode('utf-8'))
...
Got Data: "This data is passed to stdin of the CGI"
```

上述示例中的 CGI 代码如下所示：

```
#!/usr/bin/env python
import sys
data = sys.stdin.read()
print('Content-type: text/plain\n\nGot Data: "%s"' % data)
```

下面是利用*Request* 发送 “PUT“ 请求的示例：

```
import urllib.request
DATA = b'some data'
req = urllib.request.Request(url='http://localhost:8080', data=DATA,method='PUT')
with urllib.request.urlopen(req) as f:
    pass
print(f.status)
print(f.reason)
```

基本 HTTP 认证示例：

```
import urllib.request
# Create an OpenerDirector with support for Basic HTTP Authentication...
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(realm='PDQ Application',
                          uri='https://mahler:8092/site-updates.py',
                          user='klem',
```

(下页继续)

(繼續上一頁)

```
                          passwd='kadidd!ehopper')
opener = urllib.request.build_opener(auth_handler)
# ...and install it globally so it can be used with urlopen.
urllib.request.install_opener(opener)
urllib.request.urlopen('http://www.example.com/login.html')
```

build_opener() 默认提供了很多现成的 handler，包括*ProxyHandler*。默认情况下，*ProxyHandler* 会使用名为 <scheme>_proxy 的环境变量，其中的 <scheme> 是相关的 URL 协议。例如，可以读取 http_proxy 环境变量来获取 HTTP 代理的 URL。

以下示例将默认的*ProxyHandler* 替换为自己的 handler，由程序提供代理 URL，并利用*ProxyBasicAuthHandler* 加入代理认证的支持：

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')

opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
# This time, rather than install the OpenerDirector, we use it directly:
opener.open('http://www.example.com/login.html')
```

添加 HTTP 头部信息：

可利用*Request* 构造函数的 *headers* 参数，或者是：

```
import urllib.request
req = urllib.request.Request('http://www.example.com/')
req.add_header('Referer', 'http://www.python.org/')
# Customize the default User-Agent header value:
req.add_header('User-Agent', 'urllib-example/0.1 (Contact: . . .)')
r = urllib.request.urlopen(req)
```

OpenerDirector 自动会在每个*Request* 中加入一项 *User-Agent* 头部信息。若要修改，请参见以下语句：

```
import urllib.request
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
opener.open('http://www.example.com/')
```

另请记得，当*Request* 传给*urlopen()* （或*OpenerDirector.open()*）时，会加入一些标准的头部信息（*Content-Length* 、*Content-Type* 和 *Host*）。

以下会话示例用 GET 方法读取包含参数的 URL。

```
>>> import urllib.request
>>> import urllib.parse
>>> params = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> url = "http://www.musi-cal.com/cgi-bin/query?%s" % params
>>> with urllib.request.urlopen(url) as f:
...     print(f.read().decode('utf-8'))
...
```

以下示例换用 POST 方法。请注意 urlencode 输出结果先被编码为字节串 data，再送入 urlopen。

```
>>> import urllib.request
>>> import urllib.parse
```

(下页继续)

(繼續上一頁)

```
>>> data = urllib.parse.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> data = data.encode('ascii')
>>> with urllib.request.urlopen("http://requestb.in/xrbl82xr", data) as f:
...     print(f.read().decode('utf-8'))
...
```

以下示例显式指定了 HTTP 代理，以覆盖环境变量中的设置：

```
>>> import urllib.request
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.request.FancyURLopener(proxies)
>>> with opener.open("http://www.python.org") as f:
...     f.read().decode('utf-8')
...
```

以下示例根本不用代理，也覆盖了环境变量中的设置：

```
>>> import urllib.request
>>> opener = urllib.request.FancyURLopener({})
>>> with opener.open("http://www.python.org/") as f:
...     f.read().decode('utf-8')
...
```

21.4.24 沿袭的接口

以下函数和类是由 Python 2 模块 `urllib``（相对早于 ``urllib2` ）移植过来的。将来某个时候可能会停用。

`urllib.request.`**`urlretrieve`**(*url*, *filename=None*, *reporthook=None*, *data=None*)

将 URL 形式的网络对象复制为本地文件。如果 URL 指向本地文件，则必须提供文件名才会执行复制。返回值为元组 `(filename, headers)`，其中 *filename* 是保存网络对象的本地文件名，*headers* 是由*`urlopen()`* 返回的远程对象 `info()` 方法的调用结果。可能触发的异常与*`urlopen()`* 相同。

第二个参数指定文件的保存位置（若未给出，则会是名称随机生成的临时文件）。第三个参数是个可调用对象，在建立网络连接时将会调用一次，之后每次读完数据块后会调用一次。该可调用对象将会传入 3 个参数：已传输的块数、块的大小（以字节为单位）和文件总的大小。如果面对的是老旧 FTP 服务器，文件大小参数可能会是 `-1`，这些服务器响应读取请求时不会返回文件大小。

以下例子演示了大部分常用场景：

```
>>> import urllib.request
>>> local_filename, headers = urllib.request.urlretrieve('http://python.org/')
>>> html = open(local_filename)
>>> html.close()
```

如果 *url* 使用 `http:` 方式的标识符，则可能给出可选的 *data* 参数来指定一个 `POST` 请求 (通常的请求类型为 `GET`)。*data* 参数必须是标准 *`application/x-www-form-urlencoded`* 格式的字节串对象；参见*`urllib.parse.urlencode()`* 函数。

`urlretrieve()` 在检测到可用数据少于预期的大小（即由 *Content-Length* 标头所报告的大小）时将引发 `ContentTooShortError`。例如，这可能会在下载中断时发生。

Content-Length 会被视为大小的下限：如果存在更多的可用数据，urlretrieve 会读取更多的数据，但是如果可用数据少于该值，则会引发异常。

在这种情况下你仍然能够获取已下载的数据，它会存放在异常实例的 `content` 属性中。

如果未提供 *Content-Length* 标头，urlretrieve 就无法检查它所下载的数据大小，只是简单地返回它。在这种情况下你只能假定下载是成功的。

urllib.request.**urlcleanup**()
清理之前调用*urlretrieve()* 时可能留下的临时文件。

class urllib.request.**URLopener**(*proxies=None*, ***x509*)
3.3 版後已𝔽用.

用于打开和读取 URL 的基类。除非你需要支持使用 http:, ftp: 或 file: 以外的方式来打开对象，那你也许可以使用*FancyURLopener*。

在默认情况下，*URLopener* 类会发送一个内容为 urllib/VVV 的 *User-Agent* 标头，其中 *VVV* 是*urllib* 的版本号。应用程序可以通过子类化*URLopener* 或*FancyURLopener* 并在子类定义中将类属性*version* 设为适当的字符串值来定义自己的 *User-Agent* 标头。

可选的 *proxies* 形参应当是一个将方式名称映射到代理 URL 的字典，如为空字典则会完全关闭代理。它的默认值为 None，在这种情况下如果存在环境代理设置则将使用它，正如上文*urlopen()* 的定义中所描述的。

一些归属于 *x509* 的额外关键字形参可在使用 https: 方式时被用于客户端的验证。支持通过关键字 *key_file* 和 *cert_file* 来提供 SSL 密钥和证书；对客户端验证的支持需要提供这两个形参。

如果服务器返回错误代码则*URLopener* 对象将引发*OSError* 异常。

open(*fullurl*, *data=None*)
使用适当的协议打开 *fullurl*。此方法会设置缓存和代理信息，然后调用适当的打开方法并附带其输入参数。如果方式无法被识别，则会调用*open_unknown()*。*data* 参数的含义与*urlopen()* 中的 *data* 参数相同。

此方法总是会使用*quote()* 来对 *fullurl* 进行转码。

open_unknown(*fullurl*, *data=None*)
用于打开未知 URL 类型的可重载接口。

retrieve(*url*, *filename=None*, *reporthook=None*, *data=None*)
提取 *url* 的内容并将其存放到 *filename* 中。返回值为元组，由一个本地文件名和一个包含响应标头 (对于远程 URL) 的*email.message.Message* 对象或者 None (对于本地 URL)。之后调用方必须打开并读取 *filename* 的内容。如果 *filename* 未给出并且 URL 指向一个本地文件，则会返回输入文件名。如果 URL 非本地并且 *filename* 未给出，则文件名为带有与输入 URL 的路径末尾部分后缀相匹配的后缀的*tempfile.mktemp()* 的输出。如果给出了 *reporthook*，它必须为接受三个数字形参的函数：数据分块编号、读入分块的最大数据量和下载的总数据量 (未知则为 -1)。它将在开始时和从网络读取每个数据分块之后被调用。对于本地 URL *reporthook* 会被忽略。

如果 *url* 使用 http: 方式的标识符，则可能给出可选的 *data* 参数来指定一个 POST 请求 (通常的请求类型为 GET)。*data* 参数必须为标准的 *application/x-www-form-urlencoded* 格式；参见*urllib.parse.urlencode()* 函数。

version
指明打开器对象的用户代理名称的变量。以便让*urllib* 告诉服务器它是某个特定的用户代理，请在子类中将其作为类变量来设置或是在调用基类构造器之前在构造器中设置。

class urllib.request.**FancyURLopener**(...)
3.3 版後已𝔽用.

FancyURLopener 子类化了*URLopener* 以提供对以下 HTTP 响应代码的默认处理: 301, 302, 303, 307 和 401。对于上述的 30x 响应代码，会使用 *Location* 标头来获取实际 URL。对于 401 响应代码 (authentication required)，则会执行基本 HTTP 验证。对于 30x 响应代码，递归层数会受 *maxtries* 属性值的限制，该值默认为 10。

对于所有其他响应代码，会调用 http_error_default() 方法，你可以在子类中重载此方法来正确地处理错误。

備𝔽: 根据 RFC 2616 的说明，对 POST 请求的 301 和 302 响应不可在未经用户确认的情况下自动进行重定向。在现实情况下，浏览器确实允许自动重定义这些响应，将 POST 更改为 GET，于是*urllib*就会复现这种行为。

传给此构造器的形参与*URLopener* 的相同。

備𝔽: 当执行基本验证时，*FancyURLopener* 实例会调用其*prompt_user_passwd()* 方法。默认的实现将向用户询问控制终端所要求的信息。如有必要子类可以重载此方法来支持更适当的行为。

FancyURLopener 类附带了一个额外方法，它应当被重载以提供适当的行为:

prompt_user_passwd(*host*, *realm*)
返回指定的安全体系下在给定的主机上验证用户所需的信息。返回的值应当是一个元组 (user, password)，它可被用于基本验证。

该实现会在终端上提示此信息；应用程序应当重载此方法以使用本地环境下适当的交互模型。

21.4.25 urllib.request 的限制

- 目前，仅支持下列协议: HTTP (0.9 和 1.0 版), FTP, 本地文件, 以及数据 URL。

  3.4 版更變: 增加了对数据 URL 的支持。
- *urlretrieve()* 的缓存特性已被禁用，等待有人有时间去正确地解决过期时间标头的处理问题。
- 应当有一个函数来查询特定 URL 是否在缓存中。
- 为了保持向下兼容性，如果某个 URL 看起来是指向本地文件但该文件无法被打开，则该 URL 会使用 FTP 协议来重新解读。这有时可能会导致令人迷惑的错误消息。
- *urlopen()* 和*urlretrieve()* 方法在等待网络连接建立时会导致任意长的延迟。这意味着在不使用线程的情况下，搭建一个可交互的网络客户端是非常困难的。
- 由*urlopen()* 或*urlretrieve()* 返回的数据就是服务器所返回的原始数据。这可以是二进制数据（如图片）、纯文本或 HTML 代码等。HTTP 协议在响应标头中提供了类型信息，这可以通过读取 *Content-Type* 标头来查看。如果返回的数据是 HTML，你可以使用*html.parser* 模块来解析它。
- 处理 FTP 协议的代码无法区分文件和目录。这在尝试读取指向不可访问的 URL 时可能导致意外的行为。如果 URL 以一个 / 结束，它会被认为指向一个目录并将作相应的处理。但是如果读取一个文件的尝试导致了 550 错误（表示 URL 无法找到或不可访问，这常常是由于权限原因），则该路径会被视为一个目录以便处理 URL 是指定一个目录但略去了末尾 / 的情况。这在你尝试获取一个因其设置了读取权限因而无法访问的文件时会造成误导性的结果；FTP 代码将尝试读取它，因 550 错误而失败，然后又为这个不可读取的文件执行目录列表操作。如果需要细粒度的控制，请考虑使用*ftplib* 模块，子类化*FancyURLopener*，或是修改 *_urlopener* 来满足你的需求。

21.5 urllib.response --- urllib 使用的 Response 类

urllib.response 模块定义了一些函数和提供最小化文件类接口包括 read() 和 readline() 等的类。此模块定义的函数会由*urllib.request* 模块在内部使用。典型的响应对象是一个*urllib.response.addinfourl* 实例:

class urllib.response.**addinfourl**

url
已读取资源的 URL，通常用于确定是否进行了重定向。

headers
以*EmailMessage* 实例的形式返回响应的标头。

status
3.9 版新加入.

由服务器返回的状态码。

geturl()
3.9 版後已🄵用: 已弃用，建议改用*url*。

info()
3.9 版後已🄵用: 已弃用，建议改用*headers*。

code
3.9 版後已🄵用: 已弃用，建议改用*status*。

getstatus()
3.9 版後已🄵用: 已弃用，建议改用*status*。

21.6 urllib.parse 用于解析 URL

源代码: Lib/urllib/parse.py

该模块定义了一个标准接口，用于 URL 字符串按组件 (协议、网络位置、路径等) 分解，或将组件组合回 URL 字符串，并将” 相对 URL ” 转换为给定” 基础 URL ” 的绝对 URL。

The module has been designed to match the Internet RFC on Relative Uniform Resource Locators. It supports the following URL schemes: file, ftp, gopher, hdl, http, https, imap, mailto, mms, news, nntp, prospero, rsync, rtsp, rtspu, sftp, shttp, sip, sips, snews, svn, svn+ssh, telnet, wais, ws, wss.

urllib.parse 模块定义的函数可分为两个主要门类: URL 解析和 URL 转码。这些函数将在以下各节中详细说明。

21.6.1 URL 解析

URL 解析功能可以将一个 URL 字符串分割成其组件，或者将 URL 组件组合成一个 URL 字符串。

`urllib.parse.`**`urlparse`**(*urlstring*, *scheme=''*, *allow_fragments=True*)

将一个 URL 解析为六个部分，返回一个包含 6 项的*named tuple*。这对应于 URL 的主要结构: `scheme://netloc/path;parameters?query#fragment`。每个元组项均为字符串，可能为空字符串。这些部分不会再被拆分为更小的部分（例如，netloc 将为单个字符串），并且% 转义不会被扩展。上面显示的分隔符不会出现在结果中，只有 *path* 部分的开头斜杠例外，它如果存在则会被保留。例如:

```
>>> from urllib.parse import urlparse
>>> urlparse("scheme://netloc/path;parameters?query#fragment")
ParseResult(scheme='scheme', netloc='netloc', path='/path;parameters', params='',
            query='query', fragment='fragment')
>>> o = urlparse("http://docs.python.org:80/3/library/urllib.parse.html?"
...              "highlight=params#url-parsing")
>>> o
ParseResult(scheme='http', netloc='docs.python.org:80',
            path='/3/library/urllib.parse.html', params='',
            query='highlight=params', fragment='url-parsing')
>>> o.scheme
'http'
>>> o.netloc
'docs.python.org:80'
>>> o.hostname
'docs.python.org'
>>> o.port
80
>>> o._replace(fragment="").geturl()
'http://docs.python.org:80/3/library/urllib.parse.html?highlight=params'
```

根据 **RFC 1808** 中的语法规范，urlparse 仅在 netloc 前面正确地附带了'//' 的情况下才会识别它。否则输入会被当作是一个相对 URL 因而以路径的组成部分开头。

```
>>> from urllib.parse import urlparse
>>> urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('www.cwi.nl/%7Eguido/Python.html')
ParseResult(scheme='', netloc='', path='www.cwi.nl/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> urlparse('help/Python.html')
ParseResult(scheme='', netloc='', path='help/Python.html', params='',
            query='', fragment='')
```

scheme 参数给出了默认的协议，只有在 URL 未指定协议的情况下才会被使用。它应该是与 *urlstring* 相同的类型（文本或字节串），除此之外默认值 `''` 也总是被允许，并会在适当情况下自动转换为 `b''`。

如果 *allow_fragments* 参数为假值，则片段标识符不会被识别。它们会被解析为路径、参数或查询部分，在返回值中 `fragment` 会被设为空字符串。

返回值是一个*named tuple*，这意味着它的条目可以通过索引或作为命名属性来访问，这些属性是:

属性	索引	值	值（如果不存在）
scheme	0	URL 协议	*scheme* 参数
netloc	1	网络位置部分	空字符串
path	2	分层路径	空字符串
params	3	No longer used	always an empty string
query	4	查询组件	空字符串
fragment	5	片段识别	空字符串
username		用户名	*None*
password		密码	*None*
hostname		主机名（小写）	*None*
port		端口号为整数（如果存在）	*None*

如果在 URL 中指定了无效的端口，读取 port 属性将引发*ValueError*。有关结果对象的更多信息请参阅结构化解析结果 一节。

在 netloc 属性中不匹配的方括号将引发*ValueError*。

如果 netloc 属性中的字符在 NFKC 规范化下（如 IDNA 编码格式所使用的）被分解成 /, ?, #, @ 或 : 则将引发*ValueError*。如果在解析之前 URL 就被分解，则不会引发错误。

与所有具名元组的情况一样，该子类还有一些特别有用的附加方法和属性。其中一个方法是 _replace()。_replace() 方法将返回一个新的 ParseResult 对象来将指定字段替换为新的值。

```
>>> from urllib.parse import urlparse
>>> u = urlparse('//www.cwi.nl:80/%7Eguido/Python.html')
>>> u
ParseResult(scheme='', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
>>> u._replace(scheme='http')
ParseResult(scheme='http', netloc='www.cwi.nl:80', path='/%7Eguido/Python.html',
            params='', query='', fragment='')
```

3.2 版更變: 添加了 IPv6 URL 解析功能。

3.3 版更變: The fragment is now parsed for all URL schemes (unless *allow_fragment* is false), in accordance with **RFC 3986**. Previously, a whitelist of schemes that support fragments existed.

3.6 版更變: 超范围的端口号现在会引发*ValueError*，而不是返回*None*。

3.8 版更變: 在 NFKC 规范化下会影响 netloc 解析的字符现在将引发*ValueError*。

urllib.parse.**parse_qs**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

解析以字符串参数形式（类型为 *application/x-www-form-urlencoded* 的数据）给出的查询字符串。返回字典形式的数据。结果字典的键为唯一的查询变量名而值为每个变量名对应的值列表。

可选参数 *keep_blank_values* 是一个旗标，指明是否要将以百分号转码的空值作为空字符串处理。真值表示空值应当被保留作为空字符串。默认的假值表示空值会被忽略并将其视作未包括的值。

可选参数 *strict_parsing* 是一个旗标，指明要如何处理解析错误。如为假值（默认），错误会被静默地忽略。如为真值，错误会引发*ValueError* 异常。

可选的 *encoding* 和 *errors* 形参指定如何将以百分号编码的序列解码为 Unicode 字符，即作为*bytes.decode()* 方法所接受的数据。

可选参数 *max_num_fields* 是要读取的最大字段数量的。如果设置，则如果读取的字段超过 *max_num_fields* 会引发*ValueError*。

可选参数 *separator* 是用来分隔查询参数的符号。默认为 &。

使用*urllib.parse.urlencode()* 函数 (并将 doseq 形参设为 True) 将这样的字典转换为查询字符串。.

3.2 版更變: 增加了 *encoding* 和 *errors* 形参。

3.8 版更變: 增加了 *max_num_fields* 形参。

3.9.2 版更變: Added *separator* parameter with the default value of &. Python versions earlier than Python 3.9.2 allowed using both ; and & as query parameter separator. This has been changed to allow only a single separator key, with & as the default separator.

urllib.parse.**parse_qsl**(*qs*, *keep_blank_values=False*, *strict_parsing=False*, *encoding='utf-8'*, *errors='replace'*, *max_num_fields=None*, *separator='&'*)

解析以字符串参数形式（类型为 *application/x-www-form-urlencoded* 的数据）给出的查询字符串。数据以字段名和字段值对列表的形式返回。

可选参数 *keep_blank_values* 是一个旗标，指明是否要将以百分号转码的空值作为空字符串处理。真值表示空值应当被保留作为空字符串。默认的假值表示空值会被忽略并将其视作未包括的值。

可选参数 *strict_parsing* 是一个旗标，指明要如何处理解析错误。如为假值（默认），错误会被静默地忽略。如为真值，错误会引发*ValueError* 异常。

可选的 *encoding* 和 *errors* 形参指定如何将以百分号编码的序列解码为 Unicode 字符，即作为*bytes.decode()* 方法所接受的数据。

可选参数 *max_num_fields* 是要读取的最大字段数量的。如果设置，则如果读取的字段超过 *max_num_fields* 会引发*ValueError*。

可选参数 *separator* 是用来分隔查询参数的符号。默认为 &。

使用*urllib.parse.urlencode()* 函数将这样的名值对列表转换为查询字符串。

3.2 版更變: 增加了 *encoding* 和 *errors* 形参。

3.8 版更變: 增加了 *max_num_fields* 形参。

3.9.2 版更變: Added *separator* parameter with the default value of &. Python versions earlier than Python 3.9.2 allowed using both ; and & as query parameter separator. This has been changed to allow only a single separator key, with & as the default separator.

urllib.parse.**urlunparse**(*parts*)

根据 urlparse() 所返回的元组来构造一个 URL。*parts* 参数可以是任何包含六个条目的可迭代对象。构造的结果可能是略有不同但保持等价的 URL，如果被解析的 URL 原本包含不必要的分隔符（例如，带有空查询的 ?；RFC 已声明这是等价的）。

urllib.parse.**urlsplit**(*urlstring*, *scheme=''*, *allow_fragments=True*)

此函数类似于*urlparse()*，但不会拆分来自 URL 的参数。此函数通常应当在需要允许将参数应用到 URL 的 *path* 部分的每个分节的较新的 URL 语法的情况下 (参见 **RFC 2396**) 被用来代替*urlparse()*。需要使用一个拆分函数来拆分路径分节和参数。此函数将返回包含 5 个条目的*named tuple*:

```
(addressing scheme, network location, path, query, fragment identifier).
```

返回值是一个*named tuple*，它的条目可以通过索引或作为命名属性来访问:

属性	索引	值	值（如果不存在）
scheme	0	URL 协议	*scheme* 参数
netloc	1	网络位置部分	空字符串
path	2	分层路径	空字符串
query	3	查询组件	空字符串
fragment	4	片段识别	空字符串
username		用户名	*None*
password		密码	*None*
hostname		主机名（小写）	*None*
port		端口号为整数（如果存在）	*None*

如果在 URL 中指定了无效的端口，读取 port 属性将引发*ValueError*。有关结果对象的更多信息请参阅结构化解析结果 一节。

在 netloc 属性中不匹配的方括号将引发*ValueError*。

如果 netloc 属性中的字符在 NFKC 规范化下（如 IDNA 编码格式所使用的）被分解成 /, ?, #, @ 或 :，则将引发*ValueError*。如果在解析之前 URL 就被分解，则不会引发错误。

依据更新 RFC 3986 的 WHATWG spec，ASCII 换行符 \n, \r 和制表符 \t 等字符会从 URL 中被去除。

3.6 版更變: 超范围的端口号现在会引发*ValueError*，而不是返回*None*。

3.8 版更變: 在 NFKC 规范化下会影响 netloc 解析的字符现在将引发*ValueError*。

3.9.5 版更變: ASCII 换行符和制表符会从 URL 中被去除。

urllib.parse.**urlunsplit**(*parts*)
将*urlsplit()* 所返回的元组中的元素合并为一个字符串形式的完整 URL。*parts* 参数可以是任何包含五个条目的可迭代对象。其结果可能是略有不同但保持等价的 URL，如果被解析的 URL 原本包含不必要的分隔符（例如，带有空查询的?；RFC 已声明这是等价的）。

urllib.parse.**urljoin**(*base*, *url*, *allow_fragments=True*)
通过合并一个" 基准 URL" (*base*) 和另一个 URL (*url*) 来构造一个完整 ("absolute") URL。在非正式情况下，这将使用基准 URL 的各部分，特别是地址协议、网络位置和 (一部分) 路径来提供相对 URL 中缺失的部分。例如:

```
>>> from urllib.parse import urljoin
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html', 'FAQ.html')
'http://www.cwi.nl/%7Eguido/FAQ.html'
```

allow_fragments 参数具有与*urlparse()* 中的对应参数一致的含义与默认值。

備𝔽: 如果 *url* 为绝对 URL (即以 // 或 scheme:// 打头)，则 *url* 的主机名和/或协议将出现在结果中。例如:

```
>>> urljoin('http://www.cwi.nl/%7Eguido/Python.html',
...         '//www.python.org/%7Eguido')
'http://www.python.org/%7Eguido'
```

如果你不想要那样的行为，请使用*urlsplit()* 和*urlunsplit()* 对 *url* 进行预处理，移除可能存在的 *scheme* 和 *netloc* 部分。

3.5 版更變: 更新行为以匹配 **RFC 3986** 中定义的语义。

urllib.parse.**urldefrag**(*url*)
如果 *url* 包含片段标识符，则返回不带片段标识符的 *url* 修改版本。如果 *url* 中没有片段标识符，则返回未经修改的 *url* 和一个空字符串。

返回值是一个*named tuple*，它的条目可以通过索引或作为命名属性来访问:

属性	索引	值	值（如果不存在）
url	0	不带片段的 URL	空字符串
fragment	1	片段识别	空字符串

请参阅结构化解析结果 一节了解有关结果对象的更多信息。

3.2 版更變: 结果为已构造好的对象而不是一个简单的 2 元组。-tuple.

urllib.parse.**unwrap**(*url*)
从已包装的 URL (即被格式化为 <URL:scheme://host/path>, <scheme://host/path>, URL:scheme://host/path 或 scheme://host/path 的字符串) 中提取 URL。如果 *url* 不是一个已包装的 URL，它将被原样返回。

21.6.2 解析 ASCII 编码字节

这些 URL 解析函数最初设计只用于操作字符串。但在实践中，它也能够操作经过正确转码和编码的 ASCII 字节序列形式的 URL。相应地，此模块中的 URL 解析函数既可以操作*str* 对象也可以操作*bytes* 和*bytearray* 对象。

如果传入*str* 数据，结果将只包含*str* 数据。如果传入*bytes* 或*bytearray* 数据，则结果也将只包含*bytes* 数据。

试图在单个函数调用中混用*str* 数据和*bytes* 或*bytearray* 数据将导致引发*TypeError*，而试图传入非 ASCII 字节值则将引发*UnicodeDecodeError*。

为了支持结果对象在*str* 和*bytes* 之间方便地转换，所有来自 URL 解析函数的返回值都会提供 encode() 方法 (当结果包含*str* 数据) 或 decode() 方法 (当结果包含*bytes* 数据)。这些方法的签名与*str* 和*bytes* 的对应方法相匹配 (不同之处在于其默认编码格式是 'ascii' 而非 'utf-8')。每个方法会输出包含相应类型的*bytes* 数据 (对于 encode() 方法) 或*str* 数据 (对于 decode() 方法) 的值。

对于某些需要在有可能不正确地转码的包含非 ASCII 数据的 URL 上进行操作的应用程序来说，在发起调用 URL 解析方法之前必须自行将字节串解码为字符。

在本节中描述的行为仅适用于 URL 解析函数。URL 转码函数在产生和消耗字节序列时使用它们自己的规则，详情参见单独 URL 转码函数的文档。

3.2 版更變: URL 解析函数现在接受 ASCII 编码的字节序列

21.6.3 结构化解析结果

urlparse(), *urlsplit()* 和*urldefrag()* 函数的结果对象是*tuple* 类型的子类。这些子类中增加了在那些函数的文档中列出的属性，之前小节中描述的编码和解码支持，以及一个附加方法:

urllib.parse.SplitResult.**geturl**()
以字符串形式返回原始 URL 的重合并版本。这可能与原始 URL 有所不同，例如协议的名称可能被正规化为小写字母、空的组成部分可能被丢弃。特别地，空的参数、查询和片段标识符将会被移除。

对于*urldefrag()* 的结果，只有空的片段标识符会被移除。对于*urlsplit()* 和*urlparse()* 的结果，所有被记录的改变都会被应用到此方法所返回的 URL 上。

如果是通过原始的解析方法传回则此方法的结果会保持不变:

```
>>> from urllib.parse import urlsplit
>>> url = 'HTTP://www.Python.org/doc/#'
>>> r1 = urlsplit(url)
>>> r1.geturl()
'http://www.Python.org/doc/'
>>> r2 = urlsplit(r1.geturl())
>>> r2.geturl()
'http://www.Python.org/doc/'
```

下面的类提供了当在*str* 对象上操作时对结构化解析结果的实现:

class urllib.parse.**DefragResult**(*url*, *fragment*)
用于*urldefrag()* 结果的实体类，包含有*str* 数据。encode() 方法会返回一个*DefragResultBytes* 实例。

3.2 版新加入.

class urllib.parse.**ParseResult**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)
用于*urlparse()* 结果的实体类，包含有*str* 数据。encode() 方法会返回一个*ParseResultBytes* 实例。

class urllib.parse.**SplitResult**(*scheme*, *netloc*, *path*, *query*, *fragment*)
用于*urlsplit()* 结果的实体类，包含有*str* 数据。encode() 方法会返回一个*SplitResultBytes* 实例。

下面的类提供了当在*bytes* 或*bytearray* 对象上操作时对解析结果的实现:

class urllib.parse.**DefragResultBytes**(*url*, *fragment*)
用于*urldefrag()* 结果的实体类，包含有*bytes* 数据。decode() 方法会返回一个*DefragResult* 实例。

3.2 版新加入.

class urllib.parse.**ParseResultBytes**(*scheme*, *netloc*, *path*, *params*, *query*, *fragment*)
用于*urlparse()* 结果的实体类，包含有*bytes* 数据。decode() 方法会返回一个*ParseResult* 实例。

3.2 版新加入.

class urllib.parse.**SplitResultBytes**(*scheme*, *netloc*, *path*, *query*, *fragment*)
用于*urlsplit()* 结果的实体类，包含有*bytes* 数据。decode() 方法会返回一个*SplitResult* 实例。

3.2 版新加入.

21.6.4 URL 转码

URL 转码函数的功能是接收程序数据并通过对特殊字符进行转码并正确编码非 ASCII 文本来将其转为可以安全地用作 URL 组成部分的形式。它们还支持逆转此操作以便从作为 URL 组成部分的内容中重建原始数据，如果上述的 URL 解析函数还未覆盖此功能的话。

urllib.parse.**quote**(*string*, *safe='/'*, *encoding=None*, *errors=None*)
使用 %xx 转义符替换 *string* 中的特殊字符。字母、数字和 '_.-~' 等字符一定不会被转码。在默认情况下，此函数只对 URL 的路径部分进行转码。可选的 *safe* 形参额外指定不应被转码的 ASCII 字符 --- 其默认值为 '/'。

string 可以是*str* 或*bytes* 对象。

3.7 版更變: 从 **RFC 2396** 迁移到 **RFC 3986** 以转码 URL 字符串。"~" 现在已被包括在非保留字符集中。

可选的 *encoding* 和 *errors* 形参指明如何处理非 ASCII 字符，与*str.encode()* 方法所接受的值一样。*encoding* 默认为 'utf-8'。*errors* 默认为 'strict'，表示不受支持的字符将引发*UnicodeEncodeError*。如果 *string* 为*bytes* 则不可提供 *encoding* 和 *errors*，否则将引发*TypeError*。

请注意 quote(string, safe, encoding, errors) 等价于 quote_from_bytes(string.encode(encoding, errors), safe)。

例如: quote('/El Niño/') 将产生 '/El%20Ni%C3%B1o/'.

urllib.parse.**quote_plus**(*string*, *safe=''*, *encoding=None*, *errors=None*)
类似于*quote()*，但还会使用加号来替换空格，如在构建放入 URL 的查询字符串时对于转码 HTML 表单值时所要求的那样。原始字符串中的加号会被转义，除非它们已包括在 *safe* 中。它也不会将 *safe* 的默认值设为 '/'。

例如: quote_plus('/El Niño/') 将产生 '%2FEl+Ni%C3%B1o%2F'。

urllib.parse.**quote_from_bytes**(*bytes*, *safe='/'*)
类似于*quote()*，但是接受*bytes* 对象而非*str*，并且不执行从字符串到字节串的编码。

例如: quote_from_bytes(b'a&\xef') 产生 'a%26%EF'。

urllib.parse.**unquote**(*string*, *encoding='utf-8'*, *errors='replace'*)
将 %xx 转义符替换为其单字符等价物。可选的 *encoding* 和 *errors* 形参指定如何将以百分号编码的序列解码为 Unicode 字符，即*bytes.decode()* 方法所接受的数据。

string 可以是*str* 或*bytes* 对象。

encoding 默认为 'utf-8'。*errors* 默认为 'replace'，表示无效的序列将被替换为占位字符。

例如: unquote('/El%20Ni%C3%B1o/') 将产生 '/El Niño/'。

3.9 版更變: *string* 形参支持 bytes 和 str 对象（之前仅支持 str）。

urllib.parse.**unquote_plus**(*string*, *encoding='utf-8'*, *errors='replace'*)
类似于*unquote()*，但还会将加号替换为空格，如反转码表单值所要求的。

string 必须为*str*。

例如: unquote_plus('/El+Ni%C3%B1o/') 将产生 '/El Niño/'。

urllib.parse.**unquote_to_bytes**(*string*)
将 %xx 转义符替换为其单八位等价物，并返回一个*bytes* 对象。

string 可以是*str* 或*bytes* 对象。

如果它是*str*，则 *string* 中未转义的非 ASCII 字符会被编码为 UTF-8 字节串。

例如: unquote_to_bytes('a%26%EF') y 将产生 b'a&\xef'。

urllib.parse.**urlencode**(*query*, *doseq=False*, *safe=''*, *encoding=None*, *errors=None*, *quote_via=quote_plus*)
将一个包含有*str* 或*bytes* 对象的映射对象或二元组序列转换为以百分号编码的 ASCII 文本字符串。如果所产生的字符串要被用作*urlopen()* 函数的 POST 操作的 *data*，则它应当被编码为字节串，否则它将导致*TypeError*。

结果字符串是一系列 key=value 对，由 '&' 字符进行分隔，其中 *key* 和 *value* 都已使用 *quote_via* 函数转码。在默认情况下，会使用*quote_plus()* 来转码值，这意味着空格会被转码为 '+' 字符而'/' 字符会被转码为 %2F，即遵循 GET 请求的标准 (application/x-www-form-urlencoded)。另一个可以作为 *quote_via* 传入的替代函数是*quote()*，它将把空格转码为 %20 并且不编码'/' 字符。为了最大程序地控制要转码的内容，请使用 quote 并指定 *safe* 的值。

当使用二元组序列作为 *query* 参数时，每个元组的第一个元素为键而第二个元素为值。值元素本身也可以为一个序列，在那种情况下，如果可选的形参 *doseq* 的值为 True，则每个键的值序列元素生成单个 key=value 对（以 '&' 分隔）。被编码的字符串中的参数顺序将与序列中的形参元素顺序相匹配。

safe, *encoding* 和 *errors* 形参会被传递给 *quote_via* (*encoding* 和 *errors* 形参仅在查询元素为*str* 时会被传递)。

为了反向执行这个编码过程，此模块提供了*parse_qs()* 和*parse_qsl()* 来将查询字符串解析为 Python 数据结构。

请参考*urllib 示例* 来了解如何使用*urllib.parse.urlencode()* 方法来生成 URL 的查询字符串或 POST 请求的数据。

3.2 版更變: 查询支持字节和字符串对象。

3.5 版新加入: *quote_via* 参数.

也参考:

WHATWG - URL 现有标准 定义 URL、域名、IP 地址、application/x-www-form-urlencoded 格式及其 API 的工作组。

RFC 3986 - 统一资源标识符 这是当前的标准 (STD66)。任何对于 urllib.parse 模块的修改都必须遵循该标准。某些偏离也可能会出现，这大都是出于向下兼容的目的以及特定的经常存在于各主要浏览器上的实际解析需求。

RFC 2732 - URL 中的 IPv6 Addresses 地址显示格式。 这指明了 IPv6 URL 的解析要求。

RFC 2396 - 统一资源标识符（URI）：通用语法 描述统一资源名称 (URN) 和统一资源定位符 (URL) 通用语义要求的文档。

RFC 2368 - mailto URL 模式。 mailto URL 模式的解析要求。

RFC 1808 - 相对统一资源定位符 这个请求注释包括联结绝对和相对 URL 的规则，其中包括大量控制边界情况处理的” 异常示例”。

RFC 1738 - 统一资源定位符 (URL) 这指明了绝对 URL 的正式语义和句法。

21.7 urllib.error --- urllib.request 引发的异常类

源代码： Lib/urllib/error.py

---

urllib.error 模块为*urllib.request* 所引发的异常定义了异常类。基础异常类是*URLError*。

下列异常会被*urllib.error* 按需引发：

exception urllib.error.**URLError**
处理程序在遇到问题时会引发此异常（或其派生的异常）。它是*OSError* 的一个子类。

reason
此错误的原因。它可以是一个消息字符串或另一个异常实例。

3.3 版更變: *URLError* 已被设为*OSError* 而不是*IOError* 的子类。

exception urllib.error.**HTTPError**
虽然是一个异常（*URLError* 的一个子类），*HTTPError* 也可以作为一个非异常的文件类返回值（与*urlopen()* 返回的对象相同）。这适用于处理特殊 HTTP 错误例如作为认证请求的时候。

code
一个 HTTP 状态码，具体定义见 **RFC 2616**。这个数字的值对应于存放在*http.server.BaseHTTPRequestHandler.responses* 代码字典中的某个值。

reason
这通常是一个解释本次错误原因的字符串。

headers
导致*HTTPError* 的特定 HTTP 请求的 HTTP 响应头。

3.4 版新加入.

exception urllib.error.**ContentTooShortError**(*msg*, *content*)
此异常会在*urlretrieve()* 函数检测到已下载的数据量小于期待的数据量（由 *Content-Length* 头给定）时被引发。content 属性中将存放已下载（可能被截断）的数据。

21.8 urllib.robotparser --- robots.txt 语法分析程序

源代码： Lib/urllib/robotparser.py

此模块提供了一个单独的类*RobotFileParser*, 它可以回答关于某个特定用户代理是否能在 Web 站点获取发布 robots.txt 文件的 URL 的问题。有关 robots.txt 文件结构的更多细节请参阅 http://www.robotstxt.org/orig.html。

class urllib.robotparser.**RobotFileParser**(*url=''*)
这个类提供了一些可以读取、解析和回答关于 *url* 上的 robots.txt 文件的问题的方法。

set_url(*url*)
设置指向 robots.txt 文件的 URL。

read()
读取 robots.txt URL 并将其输入解析器。

parse(*lines*)
解析行参数。

can_fetch(*useragent*, *url*)
如果允许 *useragent* 按照被解析 robots.txt 文件中的规则来获取 *url* 则返回 True。

mtime()
返回最近一次获取 robots.txt 文件的时间。这适用于需要定期检查 robots.txt 文件更新情况的长时间运行的网页爬虫。

modified()
将最近一次获取 robots.txt 文件的时间设置为当前时间。

crawl_delay(*useragent*)
为指定的 *useragent* 从 robots.txt 返回 Crawl-delay 形参。如果此形参不存在或不适用于指定的 *useragent* 或者此形参的 robots.txt 条目存在语法错误，则返回 None。

3.6 版新加入.

request_rate(*useragent*)
以*named tuple* RequestRate(requests, seconds) 的形式从 robots.txt 返回 Request-rate 形参的内容。如果此形参不存在或不适用于指定的 *useragent* 或者此形参的 robots.txt 条目存在语法错误，则返回 None。

3.6 版新加入.

site_maps()
以*list()* 的形式从 robots.txt 返回 Sitemap 形参的内容。如果此形参不存在或者此形参的 robots.txt 条目存在语法错误，则返回 None。

3.8 版新加入.

下面的例子演示了*RobotFileParser* 类的基本用法:

```
>>> import urllib.robotparser
>>> rp = urllib.robotparser.RobotFileParser()
>>> rp.set_url("http://www.musi-cal.com/robots.txt")
>>> rp.read()
>>> rrate = rp.request_rate("*")
>>> rrate.requests
3
>>> rrate.seconds
20
>>> rp.crawl_delay("*")
6
>>> rp.can_fetch("*", "http://www.musi-cal.com/cgi-bin/search?city=San+Francisco")
False
>>> rp.can_fetch("*", "http://www.musi-cal.com/")
True
```

21.9 `http` --- HTTP 模块

源代码: Lib/http/__init__.py

---

http 是一个包，它收集了多个用于处理超文本传输协议的模块:

- *http.client* 是一个低层级的 HTTP 协议客户端；对于高层级的 URL 访问请使用*urllib.request*
- *http.server* 包含基于*socketserver* 的基本 HTTP 服务类
- *http.cookies* 包含一些有用来实现通过 cookies 进行状态管理的工具
- *http.cookiejar* 提供了 cookies 的持久化

http 也是一个通过*http.HTTPStatus* 枚举定义了一些 HTTP 状态码以及相关联消息的模块

class http.**HTTPStatus**

3.5 版新加入.

enum.IntEnum 的子类，它定义了组 HTTP 状态码，原理短语以及用英语书写的长描述文本。

用法:

```
>>> from http import HTTPStatus
>>> HTTPStatus.OK
<HTTPStatus.OK: 200>
>>> HTTPStatus.OK == 200
True
>>> HTTPStatus.OK.value
200
>>> HTTPStatus.OK.phrase
'OK'
>>> HTTPStatus.OK.description
'Request fulfilled, document follows'
>>> list(HTTPStatus)
[<HTTPStatus.CONTINUE: 100>, <HTTPStatus.SWITCHING_PROTOCOLS: 101>, ...]
```

21.9.1 HTTP 状态码

已支持并且已在*http.HTTPStatus* IANA 注册 的状态码有：

状态码	映射名	详情
100	CONTINUE	HTTP/1.1 **RFC 7231**, 6.2.1 节
101	SWITCHING_PROTOCOLS	HTTP/1.1 RFC 7231, 6.2.2 节
102	PROCESSING	WebDAV RFC 2518, 10.1 节
103	EARLY_HINTS	用于指定提示 **RFC 8297** 的 HTTP 状态码
200	OK	HTTP/1.1 RFC 7231, 6.3.1 节
201	CREATED	HTTP/1.1 RFC 7231, 6.3.2 节
202	ACCEPTED	HTTP/1.1 RFC 7231, 6.3.3 节
203	NON_AUTHORITATIVE_INFORMATION	HTTP/1.1 RFC 7231, 6.3.4 节
204	NO_CONTENT	HTTP/1.1 RFC 7231, 6.3.5 节
205	RESET_CONTENT	HTTP/1.1 RFC 7231, 6.3.6 节
206	PARTIAL_CONTENT	HTTP/1.1 RFC 7233, 4.1 节
207	MULTI_STATUS	WebDAV RFC 4918, 11.1 节
208	ALREADY_REPORTED	WebDAV Binding Extensions RFC 5842, 7.1 节
226	IM_USED	Delta Encoding in HTTP RFC 3229, 10.4.1 节
300	MULTIPLE_CHOICES：有多种资源可选择	HTTP/1.1 RFC 7231, 6.4.1 节
301	MOVED_PERMANENTLY：永久移动	HTTP/1.1 RFC 7231, 6.4.2 节
302	FOUND：临时移动	HTTP/1.1 RFC 7231, 6.4.3 节
303	SEE_OTHER：已经移动	HTTP/1.1 RFC 7231, 6.4.4 节
304	NOT_MODIFIED：没有修改	HTTP/1.1 RFC 7232, 4.1 节
305	USE_PROXY：使用代理	HTTP/1.1 RFC 7231, 6.4.5 节
307	TEMPORARY_REDIRECT：临时重定向	HTTP/1.1 **RFC 7231**, 6.4.7 节
308	PERMANENT_REDIRECT：永久重定向	Permanent Redirect **RFC 7238**, Section 3 (Experi
400	BAD_REQUEST：错误请求	HTTP/1.1 RFC 7231, 6.5.1 节
401	UNAUTHORIZED：未授权	HTTP/1.1 Authentication RFC 7235, 3.1 节
402	PAYMENT_REQUIRED：保留，将来使用	HTTP/1.1 RFC 7231, 6.5.2 节
403	FORBIDDEN：禁止	HTTP/1.1 RFC 7231, 6.5.3 节
404	NOT_FOUND：没有找到	HTTP/1.1 RFC 7231, 6.5.4 节
405	METHOD_NOT_ALLOWED：该请求方法不允许	HTTP/1.1 RFC 7231, 6.5.5 节
406	NOT_ACCEPTABLE：不可接受	HTTP/1.1 RFC 7231, 6.5.6 节
407	PROXY_AUTHENTICATION_REQUIRED：要求使用代理验证身份	HTTP/1.1 Authentication RFC 7235, 3.1 节
408	REQUEST_TIMEOUT：请求超时	HTTP/1.1 **RFC 7231**, 6.5.7 节
409	CONFLICT：冲突	HTTP/1.1 RFC 7231, 6.5.8 节
410	GONE：已经不在了	HTTP/1.1 RFC 7231, 6.5.9 节
411	LENGTH_REQUIRED：长度要求	HTTP/1.1 RFC 7231, 6.5.10 节
412	PRECONDITION_FAILED：前提条件错误	HTTP/1.1 RFC 7232, 4.2 节
413	REQUEST_ENTITY_TOO_LARGE：请求体太大了	HTTP/1.1 RFC 7231, 6.5.11 节
414	REQUEST_URI_TOO_LONG：请求 URI 太长了	HTTP/1.1 RFC 7231, 6.5.12 节
415	UNSUPPORTED_MEDIA_TYPE：不支持的媒体格式	HTTP/1.1 RFC 7231, 6.5.13 节
416	REQUESTED_RANGE_NOT_SATISFIABLE	HTTP/1.1 Range Requests RFC 7233, 4.4 节
417	EXPECTATION_FAILED：期望失败	HTTP/1.1 RFC 7231, 6.5.14 节
418	IM_A_TEAPOT	HTCPCP/1.0 **RFC 2324**, Section 2.3.2
421	MISDIRECTED_REQUEST	HTTP/2 **RFC 7540**, 9.1.2 节
422	UNPROCESSABLE_ENTITY：可加工实体	WebDAV RFC 4918, 11.2 节
423	LOCKED：锁着	WebDAV RFC 4918, 11.3 节
424	FAILED_DEPENDENCY：失败的依赖	WebDAV RFC 4918, 11.4 节
425	TOO_EARLY	使用 HTTP **RFC 8470** 中的早期数据

繼

表 1 – 繼續上一頁

状态码	映射名	详情
426	UPGRADE_REQUIRED：升级需要	HTTP/1.1 RFC 7231, 6.5.15 节
428	PRECONDITION_REQUIRED：先决条件要求	Additional HTTP Status Codes RFC 6585
429	TOO_MANY_REQUESTS：太多的请求	Additional HTTP Status Codes RFC 6585
431	REQUEST_HEADER_FIELDS_TOO_LARGE：请求头太大	Additional HTTP Status Codes RFC 6585
451	UNAVAILABLE_FOR_LEGAL_REASONS	HTTP 状态码用于报告法律障碍 **RFC 7725**
500	INTERNAL_SERVER_ERROR：内部服务错误	HTTP/1.1 RFC 7231, 6.6.1 节
501	NOT_IMPLEMENTED：不可执行	HTTP/1.1 RFC 7231, 6.6.2 节
502	BAD_GATEWAY：无效网关	HTTP/1.1 RFC 7231, 6.6.3 节
503	SERVICE_UNAVAILABLE：服务不可用	HTTP/1.1 RFC 7231, 6.6.4 节
504	GATEWAY_TIMEOUT：网关超时	HTTP/1.1 RFC 7231, 6.6.5 节
505	HTTP_VERSION_NOT_SUPPORTED：HTTP 版本不支持	HTTP/1.1 RFC 7231, 6.6.6 节
506	VARIANT_ALSO_NEGOTIATES：服务器存在内部配置错误	透明内容协商在：HTTP **RFC 2295**, 8.1 节（实
507	INSUFFICIENT_STORAGE：存储不足	WebDAV RFC 4918, 11.5 节
508	LOOP_DETECTED：循环检测	WebDAV Binding Extensions RFC 5842, 7.2 节
510	NOT_EXTENDED：不扩展	WebDAV Binding Extensions RFC 5842, 7.2 节
511	NETWORK_AUTHENTICATION_REQUIRED：要求网络身份验证	Additional HTTP Status Codes **RFC 6585**, 6 节

为了保持向后兼容性，枚举值也以常量形式出现在*http.client* 模块中，。枚举名等于常量名 (例如 `http.HTTPStatus.OK` 也可以是 `http.client.OK`)。

3.7 版更變: 添加了 `421 MISDIRECTED_REQUEST` 状态码。

3.8 版新加入: 添加了 `451 UNAVAILABLE_FOR_LEGAL_REASONS` 状态码。

3.9 版新加入: 增加了 `103 EARLY_HINTS`, `418 IM_A_TEAPOT` 和 `425 TOO_EARLY` 状态码

21.10 http.client --- HTTP 协议客户端

源代码： Lib/http/client.py

这个模块定义了实现 HTTP 和 HTTPS 协议客户端的类。它通常不直接使用 --- 模块*urllib.request* 用它来处理使用 HTTP 和 HTTPS 的 URL。

也参考:

对于更高级别的 HTTP 客户端接口，建议使用 Requests 。

備𝔽: HTTPS 支持仅在编译 Python 时启用了 SSL 支持的情况下（通过*ssl* 模块）可用。

该模块支持以下类：

class http.client.**HTTPConnection**(*host*, *port=None*[, *timeout*], *source_address=None*, *blocksize=8192*)

HTTPConnection 的实例代表与 HTTP 的一个连接事务。它的实例化应当传入一个主机和可选的端口号。如果没有传入端口号，如果主机字符串的形式为 主机：端口则会从中提取端口，否则将使用默认的 HTTP 端口（80）。如果给出了可选的 *timeout* 参数，则阻塞操作（例如连接尝试）将在指定的秒数之后超时（如果未给出，则使用全局默认超时设置）。可选的 *source_address* 参数可以为一个 (主机, 端口) 元组，用作进行 HTTP 连接的源地址。可选的 *blocksize* 参数可以字节为单位设置缓冲区的大小，用来发送文件类消息体。

举个例子，以下调用都是创建连接到同一主机和端口的服务器的实例：

```
>>> h1 = http.client.HTTPConnection('www.python.org')
>>> h2 = http.client.HTTPConnection('www.python.org:80')
>>> h3 = http.client.HTTPConnection('www.python.org', 80)
>>> h4 = http.client.HTTPConnection('www.python.org', 80, timeout=10)
```

3.2 版更變: 添加了 *source_address*。

3.4 版更變: 删除了 *strict* 参数，不再支持 HTTP 0.9 风格的“简单响应”。

3.7 版更變: 添加了 *blocksize* 参数。

class http.client.**HTTPSConnection**(*host*, *port=None*, *key_file=None*, *cert_file=None*[, *timeout*], *source_address=None*, *, *context=None*, *check_hostname=None*, *blocksize=8192*)

HTTPConnection 的子类，使用 SSL 与安全服务器进行通信。默认端口为 443。如果指定了 *context*，它必须为一个描述 SSL 各选项的*ssl.SSLContext* 实例。

请参阅安全考量 了解有关最佳实践的更多信息。

3.2 版更變: 添加了 *source_address*, *context* 和 *check_hostname*。

3.2 版更變: 这个类目前会在可能的情况下（即如果*ssl.HAS_SNI* 为真值）支持 HTTPS 虚拟主机。

3.4 版更變: 删除了 *strict* 参数，不再支持 HTTP 0.9 风格的“简单响应”。

3.4.3 版更變: 目前这个类在默认情况下会执行所有必要的证书和主机检查。要回复到先前的非验证行为，可以将 ssl._create_unverified_context() 传递给 *context* 参数。

3.8 版更變: 该类现在对于默认的 *context* 或在传入 *cert_file* 并附带自定义 *context* 时会启用 TLS 1.3 *ssl.SSLContext.post_handshake_auth*。

3.6 版後已🄵用: *key_file* 和 *cert_file* 已弃用并转而推荐 *context*。请改用*ssl.SSLContext.load_cert_chain()* 或让*ssl.create_default_context()* 为你选择系统所信任的 CA 证书。

check_hostname 参数也已弃用；应当改用 *context* 的*ssl.SSLContext.check_hostname* 属性。

class http.client.**HTTPResponse**(*sock*, *debuglevel=0*, *method=None*, *url=None*)

在成功连接后返回类的实例，而不是由用户直接实例化。

3.4 版更變: 删除了 *strict* 参数，不再支持 HTTP 0.9 风格的“简单响应”。

这个模块定义了以下函数:

http.client.**parse_headers**(*fp*)

从文件指针 *fp* 中解析头部信息，该文件代表 HTTP 请求/响应。该文件必须是 BufferedIOBase reader 对象（即不是文本），并且必须提供符合 **RFC 2822** 格式的头部。

该函数返回 http.client.HTTPMessage 的实例，带有头部各个字段，但不带正文数据（与*HTTPResponse.msg* 和*http.server.BaseHTTPRequestHandler.headers* 一样）。返回之后，文件指针 *fp* 已为读取 HTTP 正文做好准备了。

備🄵: *parse_headers()* 不会解析 HTTP 消息的开始行；只会解析各 Name: value 行。文件必须为读取这些字段做好准备，所以在调用该函数之前，第一行应该已经被读取过了。

下列异常可以适当地被引发:

exception http.client.**HTTPException**

此模块中其他异常的基类。它是*Exception* 的一个子类。

exception http.client.**NotConnected**

HTTPException 的一个子类。

exception http.client.**InvalidURL**
HTTPException 的一个子类，如果给出了一个非数字或为空值的端口就会被引发。

exception http.client.**UnknownProtocol**
HTTPException 的一个子类。

exception http.client.**UnknownTransferEncoding**
HTTPException 的一个子类。

exception http.client.**UnimplementedFileMode**
HTTPException 的一个子类。

exception http.client.**IncompleteRead**
HTTPException 的一个子类。

exception http.client.**ImproperConnectionState**
HTTPException 的一个子类。

exception http.client.**CannotSendRequest**
ImproperConnectionState 的一个子类。

exception http.client.**CannotSendHeader**
ImproperConnectionState 的一个子类。

exception http.client.**ResponseNotReady**
ImproperConnectionState 的一个子类。

exception http.client.**BadStatusLine**
HTTPException 的一个子类。如果服务器反馈了一个我们不理解的 HTTP 状态码就会被引发。

exception http.client.**LineTooLong**
HTTPException 的一个子类。如果在 HTTP 协议中从服务器接收到过长的行就会被引发。

exception http.client.**RemoteDisconnected**
ConnectionResetError 和*BadStatusLine* 的一个子类。当尝试读取响应时的结果是未从连接读取到数据时由*HTTPConnection.getresponse()* 引发，表明远端已关闭连接。

3.5 版新加入: 在此之前引发的异常为*BadStatusLine*('')。

此模块中定义的常量为:

http.client.**HTTP_PORT**
HTTP 协议默认的端口号 (总是 80)。

http.client.**HTTPS_PORT**
HTTPS 协议默认的端口号 (总是 443)。

http.client.**responses**
这个字典把 HTTP 1.1 状态码映射到 W3C 名称。

例如: http.client.responses[http.client.NOT_FOUND] 是 'NOT FOUND （未发现）。

本模块中可用的 HTTP 状态码常量可以参见*HTTP 状态码* 。

21.10.1 HTTPConnection 对象

HTTPConnection 实例拥有以下方法：

HTTPConnection.**request**(*method*, *url*, *body=None*, *headers={}*, *, *encode_chunked=False*)
这会使用 HTTP 请求方法 *method* 和选择器 *url* 向服务器发送请求。

如果给定 *body*，那么给定的数据会在信息头完成之后发送。它可能是一个*str* 、一个*bytes-like object* 、一个打开的*file object*，或者*bytes* 迭代器。如果 *body* 是字符串，它会按 HTTP 默认的 ISO-8859-1 编码；如果是一个字节类对象，它会按原样发送；如果是*file object* ，文件的内容会被发送，这个文件对象应该支持 `read()` 方法。如果这个文件对象是一个*io.TextIOBase* 实例，`read()` 方法返回的数据会按 ISO-8859-1 编码，否则 `read()` 方法返回的数据会按原样发送；如果 *body* 是一个迭代器，迭代器中的元素会被发送，直到迭代器耗尽。

headers 参数应是额外的随请求发送的 HTTP 信息头的字典。

如果 *headers* 既不包含 Content-Length 也没有 Transfer-Encoding，但存在请求正文，那么这些头字段中的一个会自动设定。如果 *body* 是 `None`，那么对于要求正文的方法 (`PUT`，`POST`，和 `PATCH`)，Content-Length 头会被设为 `0`。如果 *body* 是字符串或者类似字节的对象，并且也不是文件，Content-Length 头会设为正文的长度。任何其他类型的 *body* （一般是文件或迭代器）会按块编码，这时会自动设定 Transfer-Encoding 头以代替 Content-Length。

在 *headers* 中指定 Transfer-Encoding 时，*encode_chunked* 是唯一相关的参数。如果 *encode_chunked* 为 `False`，HTTPConnection 对象会假定所有的编码都由调用代码处理。如果为 `True`，正文会按块编码。

備𝔽: HTTP 协议在 1.1 版中添加了块传输编码。除非明确知道 HTTP 服务器可以处理 HTTP 1.1，调用者要么必须指定 Content-Length，要么必须传入*str* 或字节类对象，注意该对象不能是表达 body 的文件。

3.2 版新加入: *body* 现在可以是可迭代对象了。

3.6 版更變: 如果 Content-Length 和 Transfer-Encoding 都没有在 *headers* 中设置，文件和可迭代的 *body* 对象现在会按块编码。添加了 *encode_chunked* 参数。不会尝试去确定文件对象的 Content-Length。

HTTPConnection.**getresponse**()
应当在发送一个请求从服务器获取响应时被调用。返回一个*HTTPResponse* 的实例。

備𝔽: 请注意你必须在读取了整个响应之后才能向服务器发送新的请求。

3.5 版更變: 如果引发了*ConnectionError* 或其子类，*HTTPConnection* 对象将在发送新的请求时准备好重新连接。

HTTPConnection.**set_debuglevel**(*level*)
设置调试等级。默认的调试等级为 `0`，意味着不会打印调试输出。任何大于 `0` 的值将使得所有当前定义的调试输出被打印到 stdout。`debuglevel` 会被传给任何新创建的*HTTPResponse* 对象。

3.1 版新加入.

HTTPConnection.**set_tunnel**(*host*, *port=None*, *headers=None*)
为 HTTP 连接隧道设置主机和端口。这将允许通过代理服务器运行连接。

host 和 port 参数指明隧道连接的位置（即 CONNECT 请求所包含的地址，而 不是代理服务器的地址）。

headers 参数应为一个随 CONNECT 请求发送的额外 HTTP 标头的映射。

例如，要通过一个运行于本机 8080 端口的 HTTPS 代理服务器隧道，我们应当向*HTTPSConnection* 构造器传入代理的地址，并将我们最终想要访问的主机地址传给*set_tunnel()* 方法:

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("localhost", 8080)
>>> conn.set_tunnel("www.python.org")
>>> conn.request("HEAD","/index.html")
```

3.2 版新加入.

HTTPConnection.**connect**()
当对象被创建后连接到指定的服务器。默认情况下，如果客户端还未建立连接，此函数会在发送请求时自动被调用。

HTTPConnection.**close**()
关闭到服务器的连接。

HTTPConnection.**blocksize**
用于发送文件类消息体的缓冲区大小。

3.7 版新加入.

作为对使用上述 request() 方法的替代同，你也可以通过使用下面的四个函数，分步骤发送请的请求。

HTTPConnection.**putrequest**(*method*, *url*, *skip_host=False*, *skip_accept_encoding=False*)
应为连接服务器之后首先调用的函数。将向服务器发送一行数据，包含 *method* 字符串、*url* 字符串和 HTTP 版本（HTTP/1.1）。若要禁止自动发送 Host: 或 Accept-Encoding: 头部信息（比如需要接受其他编码格式的内容），请将 *skip_host* 或 *skip_accept_encoding* 设为非 False 值。

HTTPConnection.**putheader**(*header*, *argument*[, ...])
向服务器发送一个 **RFC 822** 格式的头部。将向服务器发送一行由头、冒号和空格以及第一个参数组成的数据。如果还给出了其他参数，将在后续行中发送，每行由一个制表符和一个参数组成。

HTTPConnection.**endheaders**(*message_body=None*, *, *encode_chunked=False*)
向服务器发送一个空行，表示头部文件结束。可选的 *message_body* 参数可用于传入一个与请求相关的消息体。

如果 *encode_chunked* 为 True，则对 *message_body* 的每次迭代结果将依照 **RFC 7230** 3.3.1 节的规范进行分块编码。数据如何编码取决于 *message_body* 的类型。如果 *message_body* 实现了 buffer 接口，编码将生成一个数据块。如果 *message_body* 是*collections.abc.Iterable*，则 *message_body* 的每次迭代都会产生一个块。如果 *message_body* 为*file object*，那么每次调用 .read() 都会产生一个数据块。在 *message_body* 结束后，本方法立即会自动标记块编码数据的结束。

備F: 由于分块编码的规范要求，迭代器本身产生的空块将被分块编码器忽略。这是为了避免目标服务器因错误编码而过早终止对请求的读取。

3.6 版新加入: 支持分块编码。加入了 *encode_chunked* 参数。

HTTPConnection.**send**(*data*)
发送数据到服务器。本函数只应在调用*endheaders()* 方法之后且调用*getresponse()* 之前直接调用。

21.10.2 HTTPResponse 对象

HTTPResponse 实例封装了来自服务器的 HTTP 响应。通过它可以访问请求头和响应体。响应是可迭代对象，可在 with 语句中使用。

3.5 版更變: 现在已实现了*io.BufferedIOBase* 接口，并且支持所有的读取操作。

HTTPResponse.**read**([*amt*])
读取并返回响应体，或后续 *amt* 个字节。

HTTPResponse.**readinto**(*b*)
读取响应体的后续 len(b) 个字节到缓冲区 *b*。返回读取的字节数。

3.3 版新加入.

HTTPResponse.**getheader**(*name*, *default=None*)
返回头部信息中的 *name* 值，如果没有与 *name* 匹配的字段，则返回 *default*。如果名为 *name* 的字段不止一个，则返回所有字段，中间用', ' 连接。如果'default' 不是单个字符串，而是其他可迭代对象，则其元素同样以逗号连接并返回。

HTTPResponse.**getheaders**()
返回 (header, value) 元组构成的列表。

HTTPResponse.**fileno**()
返回底层套接字的 `fileno`。

HTTPResponse.**msg**
包含响应头的 `http.client.HTTPMessage` 实例。`http.client.HTTPMessage` 是*email.message* 的子类。

HTTPResponse.**version**
服务器使用的 HTTP 协议版本。10 代表 HTTP/1.0，11 代表 HTTP/1.1。

HTTPResponse.**url**
已读取资源的 URL，通常用于确定是否进行了重定向。

HTTPResponse.**headers**
响应的头部信息，形式为*email.message.EmailMessage* 的实例。

HTTPResponse.**status**
由服务器返回的状态码。

HTTPResponse.**reason**
服务器返回的原因短语。

HTTPResponse.**debuglevel**
一个调试钩子。如果*debuglevel* 大于零，状态信息将在读取和解析响应数据时打印输出到 stdout。

HTTPResponse.**closed**
如果流被关闭，则为 “True“。

HTTPResponse.**geturl**()
3.9 版後已🄵用: 已弃用，建议用*url*。

HTTPResponse.**info**()
3.9 版後已🄵用: 已弃用，建议用*headers*。

HTTPResponse.**getstatus**()
3.9 版後已🄵用: 已弃用，建议用*status* 。

21.10.3 示例

下面是使用 GET 方法的会话示例：

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> print(r1.status, r1.reason)
200 OK
>>> data1 = r1.read()  # This will return entire content.
>>> # The following example demonstrates reading data in chunks.
>>> conn.request("GET", "/")
>>> r1 = conn.getresponse()
>>> while chunk := r1.read(200):
...     print(repr(chunk))
b'<!doctype html>\n<!--[if"...
...
>>> # Example of an invalid request
>>> conn = http.client.HTTPSConnection("docs.python.org")
>>> conn.request("GET", "/parrot.spam")
>>> r2 = conn.getresponse()
>>> print(r2.status, r2.reason)
404 Not Found
>>> data2 = r2.read()
>>> conn.close()
```

以下是使用 HEAD 方法的会话示例。请注意，HEAD 方法从不返回任何数据。

```
>>> import http.client
>>> conn = http.client.HTTPSConnection("www.python.org")
>>> conn.request("HEAD", "/")
>>> res = conn.getresponse()
>>> print(res.status, res.reason)
200 OK
>>> data = res.read()
>>> print(len(data))
0
>>> data == b''
True
```

下面是用 POST 发送请求的会话示例：

```
>>> import http.client, urllib.parse
>>> params = urllib.parse.urlencode({'@number': 12524, '@type': 'issue', '@action':
↪'show'})
>>> headers = {"Content-type": "application/x-www-form-urlencoded",
...            "Accept": "text/plain"}
>>> conn = http.client.HTTPConnection("bugs.python.org")
>>> conn.request("POST", "", params, headers)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
302 Found
>>> data = response.read()
>>> data
b'Redirecting to <a href="http://bugs.python.org/issue12524">http://bugs.python.org/
↪issue12524</a>'
>>> conn.close()
```

在客户端，HTTP PUT 请求与 POST 请求非常相似。区别只在于服务器端，HTTP 服务器将允许通过 PUT 请求创建资源。应该注意的是，自定义的 HTTP 方法也可以在*urllib.request.Request* 中通过设置适当的方法属性来进行处理。下面是一个会话示例，演示了如何利用 http.client 发送 PUT 请求。

```
>>> # This creates an HTTP message
>>> # with the content of BODY as the enclosed representation
>>> # for the resource http://localhost:8080/file
...
>>> import http.client
>>> BODY = "***filecontents***"
>>> conn = http.client.HTTPConnection("localhost", 8080)
>>> conn.request("PUT", "/file", BODY)
>>> response = conn.getresponse()
>>> print(response.status, response.reason)
200, OK
```

21.10.4 HTTPMessage 对象

http.client.HTTPMessage 的实例存有 HTTP 响应的头部信息。利用*email.message.Message* 类实现。

21.11 ftplib --- FTP 协议客户端

源代码： Lib/ftplib.py

本模块定义了*FTP* 类和一些相关项目。*FTP* 类实现了 FTP 协议的客户端。可以用该类编写 Python 程序，执行各种自动化的 FTP 任务，如镜像其他 FTP 服务器。*urllib.request* 模块也用它来处理使用了 FTP 的 URL。关于 FTP（文件传输协议）的详情请参阅 Internet **RFC 959**。

默认编码为 UTF-8，遵循 **RFC 2640**。

以下是使用*ftplib* 模块的会话示例:

```
>>> from ftplib import FTP
>>> ftp = FTP('ftp.us.debian.org')  # connect to host, default port
>>> ftp.login()                     # user anonymous, passwd anonymous@
'230 Login successful.'
>>> ftp.cwd('debian')               # change into "debian" directory
'250 Directory successfully changed.'
>>> ftp.retrlines('LIST')           # list directory contents
-rw-rw-r--    1 1176     1176         1063 Jun 15 10:18 README
...
drwxr-sr-x    5 1176     1176         4096 Dec 19  2000 pool
drwxr-sr-x    4 1176     1176         4096 Nov 17  2008 project
drwxr-xr-x    3 1176     1176         4096 Oct 10  2012 tools
'226 Directory send OK.'
>>> with open('README', 'wb') as fp:
>>>     ftp.retrbinary('RETR README', fp.write)
'226 Transfer complete.'
>>> ftp.quit()
'221 Goodbye.'
```

这个模块定义了以下内容:

class ftplib.**FTP**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *timeout=None*, *source_address=None*, *, *encoding='utf-8'*)

返回一个*FTP* 类的新实例。当传入 *host* 时，将调用 `connect(host)` 方法。当传入 *user* 时，将额外调用 `login(user, passwd, acct)` 方法（其中 *passwd* 和 *acct* 若没有传入则默认为空字符串）。可选参数 *timeout* 指定阻塞操作（如连接尝试）的超时（以秒为单位，如果未指定超时，将使用全局默认超时设置）。*source_address* 是一个 2 元组 `(host, port)`，套接字在连接前绑定它，作为其源地址。*encoding* 参数指定目录和文件名的编码。

FTP 类支持 `with` 语句，例如：

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
...     ftp.login()
...     ftp.dir()
...
'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
>>>
```

3.2 版更變: 支持了 `with` 语句。

3.3 版更變: 添加了 *source_address* 参数。

3.9 版更變: 如果 *timeout* 参数设置为 0，创建非阻塞套接字时，它将引发*ValueError* 来阻止该操作。添加了 *encoding* 参数，且为了遵循 **RFC 2640**，该参数默认值从 Latin-1 改为了 UTF-8。

class ftplib.**FTP_TLS**(*host=''*, *user=''*, *passwd=''*, *acct=''*, *keyfile=None*, *certfile=None*, *context=None*, *timeout=None*, *source_address=None*, *, *encoding='utf-8'*)

一个*FTP* 的子类，它为 FTP 添加了 TLS 支持，如 **RFC 4217** 所述。它将像通常一样连接到 21 端口，暗中保护在身份验证前的 FTP 控制连接。而保护数据连接需要用户明确调用*prot_p()* 方法。*context* 是一个*ssl.SSLContext* 对象，该对象可以将 SSL 配置选项、证书和私钥打包放入一个单独的（可以长久存在的）结构中。请阅读安全考量 以获取最佳实践。

keyfile 和 *certfile* 是可以代替 *context* 的传统方案，它们可以分别指向 PEM 格式的私钥和证书链文件，用于进行 SSL 连接。

3.2 版新加入.

3.3 版更變: 添加了 *source_address* 参数。

3.4 版更變: 本类现在支持通过*ssl.SSLContext.check_hostname* 和 服务器名称提示（参阅*ssl.HAS_SNI*）进行主机名检查。

3.6 版後已F用: *keyfile* 和 *certfile* 已弃用并转而推荐 *context*。请改用*ssl.SSLContext.load_cert_chain()* 或让*ssl.create_default_context()* 为你选择系统所信任的 CA 证书。

3.9 版更變: 如果 *timeout* 参数设置为 0，创建非阻塞套接字时，它将引发*ValueError* 来阻止该操作。添加了 *encoding* 参数，且为了遵循 **RFC 2640**，该参数默认值从 Latin-1 改为了 UTF-8。

以下是使用*FTP_TLS* 类的会话示例：

```
>>> ftps = FTP_TLS('ftp.pureftpd.org')
>>> ftps.login()
'230 Anonymous user logged in'
>>> ftps.prot_p()
'200 Data protection level set to "private"'
>>> ftps.nlst()
['6jack', 'OpenBSD', 'antilink', 'blogbench', 'bsdcam', 'clockspeed', 'djbdns-jedi
↪', 'docs', 'eaccelerator-jedi', 'favicon.ico', 'francotone', 'fugu', 'ignore',
↪'libpuzzle', 'metalog', 'minidentd', 'misc', 'mysql-udf-global-user-variables',
↪'php-jenkins-hash', 'php-skein-hash', 'php-webdav', 'phpaudit', 'phpbench',
↪'pincaster', 'ping', 'posto', 'pub', 'public', 'public_keys', 'pure-ftpd',
↪'qscan', 'qtc', 'sharedance', 'skycache', 'sound', 'tmp', 'ucarp']
```

(下页继续)

(繼續上一頁)

exception ftplib.**error_reply**
从服务器收到意外答复时，将引发本异常。

exception ftplib.**error_temp**
收到表示临时错误的错误代码（响应代码在 400--499 范围内）时，将引发本异常。

exception ftplib.**error_perm**
收到表示永久性错误的错误代码（响应代码在 500--599 范围内）时，将引发本异常。

exception ftplib.**error_proto**
从服务器收到不符合 FTP 响应规范的答复，比如以数字 1--5 开头时，将引发本异常。

ftplib.**all_errors**
所有异常的集合（一个元组），由于 FTP 连接出现问题（并非调用者的编码错误），*FTP* 实例的方法可能会引发这些异常。该集合包括上面列出的四个异常以及*OSError* 和*EOFError*。

也参考:

***netrc* 模块** .netrc 文件格式解析器。FTP 客户端在响应用户之前，通常使用 .netrc 文件来加载用户认证信息。

21.11.1 FTP 对象

一些方法可以按照两种方式来使用：一种处理文本文件，另一种处理二进制文件。方法名称与相应的命令相同，文本版中命令后面跟着 lines，二进制版中命令后面跟着 binary。

FTP 实例具有下列方法:

FTP.**set_debuglevel**(*level*)
设置实例的调试级别，它控制着调试信息的数量。默认值 0 不产生调试信息。值 1 产生中等数量的调试信息，通常每个请求产生一行。大于或等于 2 的值产生的调试信息最多，FTP 控制连接上发送和接收的每一行都将被记录下来。

FTP.**connect**(*host=''*, *port=0*, *timeout=None*, *source_address=None*)
连接到给定的主机和端口。默认端口号由 FTP 协议规范规定，为 21。偶尔才需要指定其他端口号。每个实例只应调用一次本函数，如果在创建实例时就传入了 host，则根本不应调用它。所有其他方法只能在建立连接后使用。可选参数 *timeout* 指定连接尝试的超时（以秒为单位）。如果没有传入 *timeout*，将使用全局默认超时设置。*source_address* 是一个 2 元组 (host, port)，套接字在连接前绑定它，作为其源地址。

引发一个审计事件 ftplib.connect，附带参数 self, host, port。

3.3 版更變: 添加了 *source_address* 参数。

FTP.**getwelcome**()
返回服务器发送的欢迎消息，作为连接开始的回复。（该消息有时包含与用户有关的免责声明或帮助信息。）

FTP.**login**(*user='anonymous'*, *passwd=''*, *acct=''*)
以 *user* 的身份登录。*passwd* 和 *acct* 是可选参数，默认为空字符串。如果没有指定 *user*，则默认为 'anonymous'。如果 *user* 为 'anonymous'，那么默认的 *passwd* 是 'anonymous@'。连接建立后，每个实例只应调用一次本函数；如果在创建实例时传入了 host 和 user，则完全不应该调用本函数。在客户端登录后，才允许执行大多数 FTP 命令。*acct* 参数提供记账信息 ("accounting information")；仅少数系统实现了该特性。

FTP.**abort**()
中止正在进行的文件传输。本方法并不总是有效，但值得一试。

FTP.**sendcmd**(*cmd*)
将一条简单的命令字符串发送到服务器，返回响应的字符串。

引发一个审计事件 `ftplib.sendcmd`，附带参数 `self`, `cmd`。

FTP.**voidcmd**(*cmd*)
将一条简单的命令字符串发送到服务器，并处理响应内容。如果收到的响应代码表示的是成功（代码范围 200--299），则不返回任何内容。否则将引发*error_reply*。

引发一个审计事件 `ftplib.sendcmd`，附带参数 `self`, `cmd`。

FTP.**retrbinary**(*cmd*, *callback*, *blocksize=8192*, *rest=None*)
以二进制传输模式下载文件。*cmd* 应为恰当的 `RETR` 命令：`'RETR 文件名'`。*callback* 函数会在收到每个数据块时调用，传入的参数是表示数据块的一个字节。为执行实际传输，创建了底层套接字对象，可选参数 *blocksize* 指定了读取该对象时的最大块大小（这也是传入 *callback* 的数据块的最大大小）。已经选择了合理的默认值。*rest* 的含义与*transfercmd()* 方法中的含义相同。

FTP.**retrlines**(*cmd*, *callback=None*)
按照初始化时的 *encoding* 参数指定的编码，获取文件或目录列表。*cmd* 应是恰当的 `RETR` 命令（参阅*retrbinary()*），也可以是诸如 `LIST` 或 `NLST` 之类的命令（通常就只是字符串 `'LIST'`）。`LIST` 获取文件列表以及那些文件的信息。`NLST` 获取文件名称列表。*callback* 函数会在每一行都调用，参数就是包含一行的字符串，删除了尾部的 CRLF。默认的 *callback* 会把行打印到 `sys.stdout`。

FTP.**set_pasv**(*val*)
如果 *val* 为 true，则打开“被动”模式，否则禁用被动模式。默认下被动模式是打开的。

FTP.**storbinary**(*cmd*, *fp*, *blocksize=8192*, *callback=None*, *rest=None*)
以二进制传输模式存储文件。*cmd* 应为恰当的 `STOR` 命令: `"STOR filename"`。*fp* 是一个文件对象 (以二进制模式打开)，将使用它的 `read()` 方法读取它，用于提供要存储的数据，直到遇到 EOF，读取时的块大小为 *blocksize*。参数 *blocksize* 的默认值为 8192。可选参数 *callback* 是单参数函数，在每个数据块发送后都会以该数据块作为参数来调用它。*rest* 的含义与*transfercmd()* 方法中的含义相同。

3.2 版更變: 添加了 *rest* 参数。

FTP.**storlines**(*cmd*, *fp*, *callback=None*)
以文本行模式存储文件。*cmd* 应为恰当的 `STOR` 命令 (请参阅*storbinary()*)。*fp* 是一个文件对象 (以二进制模式打开)，将使用它的*readline()* 方法读取它的每一行，用于提供要存储的数据，直到遇到 EOF。可选参数 *callback* 是单参数函数，在每行发送后都会以该行作为参数来调用它。

FTP.**transfercmd**(*cmd*, *rest=None*)
在 FTP 数据连接上开始传输数据。如果传输处于活动状态，传输命令由 *cmd* 指定，需发送 `EPRT` 或 `PORT` 命令，然后接受连接 (accept)。如果服务器是被动服务器，需发送 `EPSV` 或 `PASV` 命令，连接到服务器 (connect)，然后启动传输命令。两种方式都将返回用于连接的套接字。

如果传入了可选参数 *rest*，则一条 `REST` 命令会被发送到服务器，并以 *rest* 作为参数。*rest* 通常表示请求文件中的字节偏移量，它告诉服务器重新开始发送文件的字节，从请求的偏移量处开始，跳过起始字节。但是请注意，*transfercmd()* 方法会将 *rest* 转换为字符串，但是不检查字符串的内容，转换用的编码是在初始化时指定的 *encoding* 参数。如果服务器无法识别 `REST` 命令，将引发*error_reply* 异常。如果发生这种情况，只需不带 *rest* 参数调用*transfercmd()*。

FTP.**ntransfercmd**(*cmd*, *rest=None*)
类似于*transfercmd()*，但返回一个元组，包括数据连接和数据的预计大小。如果预计大小无法计算，则返回的预计大小为 `None`。*cmd* 和 *rest* 的含义与*transfercmd()* 中的相同。

FTP.**mlsd**(*path=""*, *facts=[]*)
使用 `MLSD` 命令以标准格式列出目录内容 (**RFC 3659**)。如果省略 *path* 则使用当前目录。*facts* 是字符串列表，表示所需的信息类型（如 `["type", "size", "perm"]`）。返回一个生成器对象，每个在 path 中找到的文件都将在该对象中生成两个元素的元组。第一个元素是文件名，第二个元素是该文件的 facts 的字典。该字典的内容受 *facts* 参数限制，但不能保证服务器会返回所有请求的 facts。

3.3 版新加入.

FTP.**nlst**(*argument*[, ...])
返回一个文件名列表，文件名由 NLST 命令返回。可选参数 *argument* 是待列出的目录（默认为当前服务器目录）。可以使用多个参数，将非标准选项传递给 NLST 命令。

備𝔽: 如果目标服务器支持相关命令，那么*mlsd()* 提供的 API 更好。

FTP.**dir**(*argument*[, ...])
生成目录列表，即 LIST 命令所返回的结果，并将其打印到标准输出。可选参数 *argument* 是待列出的目录（默认为当前服务器目录）。可以使用多个参数，将非标准选项传递给 LIST 命令。如果最后一个参数是一个函数，它将被用作 *callback* 函数，与*retrlines()* 中的相同，默认将打印到 sys.stdout。本方法返回 None。

備𝔽: 如果目标服务器支持相关命令，那么*mlsd()* 提供的 API 更好。

FTP.**rename**(*fromname*, *toname*)
将服务器上的文件 *fromname* 重命名为 *toname*。

FTP.**delete**(*filename*)
将服务器上名为 *filename* 的文件删除。如果删除成功，返回响应文本，如果删除失败，在权限错误时引发*error_perm*，在其他错误时引发*error_reply*。

FTP.**cwd**(*pathname*)
设置服务器端的当前目录。

FTP.**mkd**(*pathname*)
在服务器上创建一个新目录。

FTP.**pwd**()
返回服务器上当前目录的路径。

FTP.**rmd**(*dirname*)
将服务器上名为 *dirname* 的目录删除。

FTP.**size**(*filename*)
请求服务器上名为 *filename* 的文件大小。成功后以整数返回文件大小，未成功则返回 None。注意，SIZE 不是标准命令，但通常许多服务器的实现都支持该命令。

FTP.**quit**()
向服务器发送 QUIT 命令并关闭连接。这是关闭一个连接的“礼貌”方式，但是如果服务器对 QUIT 命令的响应带有错误消息则这会引发一个异常。这意味着对*close()* 方法的调用，它将使得*FTP* 实例对后继调用无效（见下文）。

FTP.**close**()
单方面关闭连接。这不该被应用于已经关闭的连接，例如成功调用*quit()* 之后的连接。在此调用之后*FTP* 实例不应被继续使用（在调用*close()* 或*quit()* 之后你不能通过再次发起调用*login()* 方法重新打开连接）。

21.11.2 FTP_TLS 对象

FTP_TLS 类继承自*FTP*，它定义了下述其他对象：

FTP_TLS.**ssl_version**
欲采用的 SSL 版本（默认为*ssl.PROTOCOL_SSLv23*）。

FTP_TLS.**auth**()
通过使用 TLS 或 SSL 来设置一个安全控制连接，具体取决于*ssl_version* 属性是如何设置的。

3.4 版更變: 此方法现在支持使用*ssl.SSLContext.check_hostname* 和服务器名称指示 (参见*ssl.HAS_SNI*) 进行主机名检查。

FTP_TLS.**ccc**()
将控制通道回复为纯文本。这适用于发挥知道如何使用非安全 FTP 处理 NAT 而无需打开固定端口的防火墙的优势。

3.3 版新加入.

FTP_TLS.**prot_p**()
设置加密数据连接。

FTP_TLS.**prot_c**()
设置明文数据连接。

21.12 poplib --- POP3 协议客户端

源代码： Lib/poplib.py

本模块定义了一个*POP3* 类，该类封装了到 POP3 服务器的连接过程，并实现了 **RFC 1939** 中定义的协议。*POP3* 类同时支持 **RFC 1939** 中最小的和可选的命令集。*POP3* 类还支持在 **RFC 2595** 中引入的 STLS 命令，用于在已建立的连接上启用加密通信。

本模块额外提供一个*POP3_SSL* 类，在连接到 POP3 服务器时，该类为使用 SSL 作为底层协议层提供了支持。

注意，尽管 POP3 具有广泛的支持，但它已经过时。POP3 服务器的实现质量差异很大，而且大多很糟糕。如果邮件服务器支持 IMAP，则最好使用*imaplib.IMAP4* 类，因为 IMAP 服务器一般实现得更好。

poplib 模块提供了两个类：

class poplib.**POP3**(*host*, *port=POP3_PORT*[, *timeout*])
本类实现实际的 POP3 协议。实例初始化时，连接就会建立。如果省略 *port*，则使用标准 POP3 端口 (110)。可选参数 *timeout* 指定连接尝试的超时时间（以秒为单位，如果未指定超时，将使用全局默认超时设置）。

引发一个审计事件 poplib.connect，附带参数 self, host, port。

引发一个审计事件 poplib.putline，附带参数 self, line。

3.9 版更變: 如果 *timeout* 参数设置为 0，创建非阻塞套接字时，它将引发*ValueError* 来阻止该操作。

class poplib.**POP3_SSL**(*host*, *port=POP3_SSL_PORT*, *keyfile=None*, *certfile=None*, *timeout=None*, *context=None*)
一个*POP3* 的子类，它使用经 SSL 加密的套接字连接到服务器。如果端口 *port* 未指定，则使用 995，它是标准的 POP3-over-SSL 端口。*timeout* 的作用与*POP3* 构造函数中的相同。*context* 是一个可选的*ssl.SSLContext* 对象，该对象可以将 SSL 配置选项、证书和私钥打包放入一个单独的（可以长久存在的）结构中。请阅读安全考量 以获取最佳实践。

keyfile 和 *certfile* 是可以代替 *context* 的传统方案，它们可以分别指向 PEM 格式的私钥和证书链文件，用于进行 SSL 连接。

引发一个审计事件 `poplib.connect`，附带参数 `self`, `host`, `port`。

引发一个审计事件 `poplib.putline`，附带参数 `self`, `line`。

3.2 版更變: 添加了 *context* 参数。

3.4 版更變: 本类现在支持使用 *ssl.SSLContext.check_hostname* 和 服务器名称指示（参阅 *ssl.HAS_SNI*）进行主机名检查。

3.6 版後已𝔽用: *keyfile* 和 *certfile* 已弃用并转而推荐 *context*。请改用 *ssl.SSLContext.load_cert_chain()* 或让 *ssl.create_default_context()* 为你选择系统所信任的 CA 证书。

3.9 版更變: 如果 *timeout* 参数设置为 0，创建非阻塞套接字时，它将引发 *ValueError* 来阻止该操作。

定义了一个异常，它是作为 *poplib* 模块的属性定义的：

exception poplib.**error_proto**
此模块的所有错误都将引发本异常（来自 *socket* 模块的错误不会被捕获）。异常的原因将以字符串的形式传递给构造函数。

也参考:

***imaplib* 模块** 标准的 Python IMAP 模块。

有关 Fetchmail 的常见问题 `fetchmail` POP/IMAP 客户端的“常见问题”收集了 POP3 服务器之间的差异和 RFC 不兼容的信息，如果要编写基于 POP 协议的应用程序，这可能会很有用。

21.12.1 POP3 对象

所有 POP3 命令均以同名的方法表示，小写，大多数方法返回的是服务器发送的响应文本。

POP3 实例具有下列方法：

POP3.**set_debuglevel**(*level*)
设置实例的调试级别，它控制着调试信息的数量。默认值 `0` 不产生调试信息。值 `1` 产生中等数量的调试信息，通常每个请求产生一行。大于或等于 `2` 的值产生的调试信息最多，FTP 控制连接上发送和接收的每一行都将被记录下来。

POP3.**getwelcome**()
返回 POP3 服务器发送的问候语字符串。

POP3.**capa**()
查询服务器支持的功能，这些功能在 **RFC 2449** 中有说明。返回一个 `{'name': ['param'...]}` 形式的字典。

3.4 版新加入.

POP3.**user**(*username*)
发送 user 命令，返回的响应应该指示需要一个密码。

POP3.**pass_**(*password*)
发送密码，响应包括邮件数和邮箱大小。注意：在调用 `quit()` 前，服务器上的邮箱都是锁定的。

POP3.**apop**(*user*, *secret*)
使用更安全的 APOP 身份验证登录到 POP3 服务器。

POP3.**rpop**(*user*)
使用 RPOP 身份验证（类似于 Unix r-命令）登录到 POP3 服务器。

POP3.**stat**()
获取邮箱状态。结果为 2 个整数组成的元组：`(message count, mailbox size)`。

POP3.**list**([*which*])
请求消息列表，结果的形式为 `(response, ['mesg_num octets', ...], octets)`。如果设置了 *which*，它表示需要列出的消息。

POP3.**retr**(*which*)
检索编号为 *which* 的整条消息，并设置其已读标志位。结果的形式为 `(response, ['line', ...], octets)`。

POP3.**dele**(*which*)
将编号为 *which* 的消息标记为待删除。在多数服务器上，删除操作直到 QUIT 才会实际执行（主要例外是 Eudora QPOP，它在断开连接时执行删除，故意违反了 RFC）。

POP3.**rset**()
移除邮箱中的所有待删除标记。

POP3.**noop**()
什么都不做。可以用于保持活动状态。

POP3.**quit**()
登出：提交更改，解除邮箱锁定，断开连接。

POP3.**top**(*which*, *howmuch*)
检索编号为 *which* 的消息，范围是消息头加上消息头往后数 *howmuch* 行。结果的形式为 `(response, ['line', ...], octets)`。

本方法使用 POP3 TOP 命令，不同于 RETR 命令，它不设置邮件的已读标志位。不幸的是，TOP 在 RFC 中说明不清晰，且在小众的服务器软件中往往不可用。信任并使用它之前，请先手动对目标 POP3 服务器测试本方法。

POP3.**uidl**(*which=None*)
返回消息摘要（唯一 ID）列表。如果指定了 *which*，那么结果将包含那条消息的唯一 ID，形式为 `'response mesgnum uid`，如果未指定，那么结果为列表 `(response, ['mesgnum uid', ...], octets)`。

POP3.**utf8**()
尝试切换至 UTF-8 模式。成功则返回服务器的响应，失败则引发*error_proto* 异常。在 **RFC 6856** 中有说明。

3.5 版新加入.

POP3.**stls**(*context=None*)
在活动连接上开启 TLS 会话，在 **RFC 2595** 中有说明。仅在用户身份验证前允许这样做。

context 参数是一个*ssl.SSLContext* 对象，该对象可以将 SSL 配置选项、证书和私钥打包放入一个单独的（可以长久存在的）结构中。请阅读*安全考量* 以获取最佳实践。

此方法支持通过*ssl.SSLContext.check_hostname* 和 服务器名称指示 (参见*ssl.HAS_SNI*) 进行主机名检查。

3.4 版新加入.

POP3_SSL 实例没有额外方法。该子类的接口与其父类的相同。

21.12.2 POP3 示例

以下是一个最短示例（不带错误检查），该示例将打开邮箱，检索并打印所有消息:

```
import getpass, poplib

M = poplib.POP3('localhost')
M.user(getpass.getuser())
M.pass_(getpass.getpass())
numMessages = len(M.list()[1])
for i in range(numMessages):
    for j in M.retr(i+1)[1]:
        print(j)
```

模块的最后有一段测试，其中包含的用法示例更加广泛。

21.13 imaplib --- IMAP4 协议客户端

源代码： Lib/imaplib.py

本模块定义了三个类：*IMAP4* 、*IMAP4_SSL* 和*IMAP4_stream* 。这三个类封装了与 IMAP4 服务器的连接并实现了 **RFC 2060** 当中定义的大多数 IMAP4rev1 客户端协议。其与 IMAP4（**RFC 1730** ）服务器后向兼容，但是 STATUS 指令在 IMAP4 中不支持。

imaplib 模块提供了三个类，其中*IMAP4* 是基类:

class imaplib.**IMAP4**(*host=''*, *port=IMAP4_PORT*, *timeout=None*)

这个类实现了实际的 IMAP4 协议。当其实例被实例化时会创建连接并确定协议版本 (IMAP4 或 IMAP4rev1)。如果未指明 *host*，则会使用 '' (本地主机)。如果省略 *port*，则会使用标准 IMAP4 端口 (143)。可选的 *timeout* 形参指定连接尝试的超时秒数。如果未指定超时或为 None，则会使用全局默认的套接字超时。

IMAP4 类支持 with 语句。当这样使用时，IMAP4 LOGOUT 命令会在 with 语句退出时自动发出。例如:

```
>>> from imaplib import IMAP4
>>> with IMAP4("domain.org") as M:
...     M.noop()
...
('OK', [b'Nothing Accomplished. d25if65hy903weo.87'])
```

3.5 版更變: 支持了 with 语句。

3.9 版更變: 添加了可选的 *timeout* 形参。

有三个异常被定义为*IMAP4* 类的属性:

exception IMAP4.**error**

任何错误都将引发该异常。异常的原因会以字符串的形式传递给构造器。

exception IMAP4.**abort**

IMAP4 服务器错误会导致引发该异常。这是*IMAP4.error* 的子类。请注意关闭此实例并实例化一个新实例通常将会允许从该异常中恢复。

exception IMAP4.**readonly**

当一个可写邮箱的状态被服务器修改时会引发此异常。此异常是*IMAP4.error* 的子类。某个其他客户端现在会具有写入权限，将需要重新打开该邮箱以重新获得写入权限。

另外还有一个针对安全连接的子类:

class imaplib.**IMAP4_SSL**(*host=''*, *port=IMAP4_SSL_PORT*, *keyfile=None*, *certfile=None*, *ssl_context=None*, *timeout=None*)

这是一个派生自*IMAP4*的子类，它使用经 SSL 加密的套接字进行连接 (为了使用这个类你需要编译时附带 SSL 支持的 socket 模块)。如果未指定 *host*，则会使用 `''` (本地主机)。如果省略了 *port*，则会使用标准的 IMAP4-over-SSL 端口 (993)。*ssl_context* 是一个*ssl.SSLContext*对象，它允许将 SSL 配置选项、证书和私钥打包放入一个单独的 (可以长久存在的) 结构体中。请阅读安全考量以获取最佳实践。

keyfile 和 *certfile* 是 *ssl_context* 的旧式替代品——它们可以指向 PEM 格式的私钥和证书链文件用于 SSL 连接。请注意 *keyfile*/*certfile* 形参不能与 *ssl_context* 共存，如果 *keyfile*/*certfile* 与 *ssl_context* 一同被提供则会引发*ValueError*。

可选的 *timeout* 形参指明连接尝试的超时秒数。如果参数未给出或为 None，则会使用全局默认的套接字超时设置。

3.3 版更變: 增加了 *ssl_context* 形参。

3.4 版更變: 本类现在支持通过*ssl.SSLContext.check_hostname*和服务器名称提示（参阅*ssl.HAS_SNI*）进行主机名检查。

3.6 版後已𝔼用: *keyfile* 和 *certfile* 已弃用并转而推荐 *ssl_context*。请改用*ssl.SSLContext.load_cert_chain()*，或让*ssl.create_default_context()*为你选择系统所信任的 CA 证书。

3.9 版更變: 添加了可选的 *timeout* 形参。

第二个子类允许由子进程所创建的连接:

class imaplib.**IMAP4_stream**(*command*)

这是一个派生自*IMAP4*的子类，它可以连接 `stdin/stdout` 文件描述符，此种文件是通过向 `subprocess.Popen()` 传入 *command* 来创建的。

定义了下列工具函数:

imaplib.**Internaldate2tuple**(*datestr*)

解析一个 IMAP4 `INTERNALDATE` 字符串并返回对应的本地时间。返回值是一个*time.struct_time*元组或者如果字符串格式错误则为 `None`。

imaplib.**Int2AP**(*num*)

将一个整数转换为使用字符集 [`A` .. `P`] 的字节串表示形式。

imaplib.**ParseFlags**(*flagstr*)

将一个 IMAP4 `FLAGS` 响应转换为包含单独旗标的元组。

imaplib.**Time2Internaldate**(*date_time*)

将 *date_time* 转换为 IMAP4 `INTERNALDATE` 表示形式。返回值是以下形式的字符串: `"DD-Mmm-YYYY HH:MM:SS +HHMM"` (包括双引号)。*date_time* 参数可以是一个代表距离纪元起始的秒数 (如*time.time()*的返回值) 的数字 (整数或浮点数)，一个代表本地时间的 9 元组，一个*time.struct_time*实例 (如*time.localtime()*的返回值)，一个感知型的*datetime.datetime*实例，或一个双引号字符串。在最后一种情况下，它会被假定已经具有正确的格式。

请注意 IMAP4 消息编号会随邮箱的改变而改变；特别是在使用 `EXPUNGE` 命令执行删除后剩余的消息会被重新编号。因此高度建议通过 UID 命令来改用 UID。

模块的最后有一段测试，其中包含的用法示例更加广泛。

也参考:

描述该协议的文档，实现该协议的服务器源代码，由华盛顿大学 IMAP 信息中心提供 (**源代码**) https://github.com/uw-imap/imap (**不再维护**)。

21.13.1 IMAP4 对象

所有 IMAP4rev1 命令都表示为同名的方法，可以为大写或小写形式。

命令的所有参数都会被转换为字符串，只有 AUTHENTICATE 例外，而, and the last argument to APPEND 的最后一个参数会被作为 IMAP4 字面值传入。如有必要 (字符串包含 IMAP4 协议中的敏感字符并且未加圆括号或双引号) 每个字符串都会被转码。但是，LOGIN 命令的 *password* 参数总是会被转码。如果你想让某个参数字符串免于被转码 (例如: STORE 的 *flags* 参数) 则要为该字符串加上圆括号 (例如: r'(\Deleted)')。

每条命令均返回一个元组: (type, [data, ...]) 其中 *type* 通常为 'OK' 或 'NO'，而 *data* 为来自命令响应的文本，或为来自命令的规定结果。每个 *data* 均为 bytes 或者元组。如果为元组，则其第一部分是响应的标头，而第二部分将包含数据 (例如: 'literal' 值)。

以下命令的 *message_set* 选项为指定要操作的一条或多条消息的字符串。它可以是一个简单的消息编号 ('1')，一段消息编号区间 ('2:4')，或者一组以逗号分隔的非连续区间 ('1:3,6:9')。区间可以包含一个星号来表示无限的上界 ('3:*')。

IMAP4 实例具有下列方法:

IMAP4.**append**(*mailbox*, *flags*, *date_time*, *message*)
将 *message* 添加到指定的邮箱。

IMAP4.**authenticate**(*mechanism*, *authobject*)
认证命令 --- 要求对响应进行处理。

mechanism 指明要使用哪种认证机制——它应当在实例变量 capabilities 中以 AUTH=mechanism 的形式出现。

authobject 必须是一个可调用对象:

```
data = authobject(response)
```

它将被调用以便处理服务器连续响应；传给它的 *response* 参数将为 bytes 类型。它应当返回 base64 编码的 bytes 数据并发送给服务器。或者在客户端中止响应时返回 None 并应改为发送 *。

3.5 版更變: 字符串形式的用户名和密码现在会被执行 utf-8 编码而不限于 ASCII 字符。

IMAP4.**check**()
为服务器上的邮箱设置检查点。

IMAP4.**close**()
关闭当前选定的邮箱。已删除的消息会从可写邮箱中被移除。在 LOGOUT 之前建议执行此命令。

IMAP4.**copy**(*message_set*, *new_mailbox*)
将 *message_set* 消息拷贝到 *new_mailbox* 的末尾。

IMAP4.**create**(*mailbox*)
新建名为 *mailbox* 新邮箱。

IMAP4.**delete**(*mailbox*)
删除名为 *mailbox* 的旧邮箱。

IMAP4.**deleteacl**(*mailbox*, *who*)
删除邮箱上某人的 ACL (移除任何权限)。

IMAP4.**enable**(*capability*)
启用 *capability* (参见 **RFC 5161**)。大多数功能都不需要被启用。目前只有 UTF8=ACCEPT 功能受到支持 (参见 **RFC 6855**)。

3.5 版新加入: *enable()* 方法本身，以及 **RFC 6855** 支持。

IMAP4.**expunge**()
从选定的邮箱中永久移除被删除的条目。为每条被删除的消息各生成一个 EXPUNGE 响应。返回包含按接收时间排序的 EXPUNGE 消息编号的列表。

IMAP4.**fetch**(*message_set*, *message_parts*)
获取消息（的各个部分）。*message_parts* 应为加圆标号的消息部分名称字符串，例如: "(UID BODY[TEXT])"。返回的数据是由消息部分封包和数据组成的元组。

IMAP4.**getacl**(*mailbox*)
获取 *mailbox* 的 ACL。此方法是非标准的，但是被 Cyrus 服务器所支持。

IMAP4.**getannotation**(*mailbox*, *entry*, *attribute*)
提取 *mailbox* 的特定 ANNOTATION。此方法是非标准的，但是被 Cyrus 服务器所支持。

IMAP4.**getquota**(*root*)
获取 quota *root* 的资源使用和限制。此方法是 rfc2087 定义的 IMAP4 QUOTA 扩展的组成部分。

IMAP4.**getquotaroot**(*mailbox*)
获取指定 *mailbox* 的 quota roots 列表。此方法是 rfc2087 定义的 IMAP4 QUOTA 扩展的组成部分。

IMAP4.**list**([*directory*[, *pattern*]])
列出 *directory* 中与 *pattern* 相匹配的邮箱名称。*directory* 默认为最高层级的电邮文件夹，而 *pattern* 默认为匹配任何文本。返回的数据包含 LIST 响应列表。

IMAP4.**login**(*user*, *password*)
使用纯文本密码标识客户。*password* 将被转码。

IMAP4.**login_cram_md5**(*user*, *password*)
在标识用户以保护密码时强制使用 CRAM-MD5 认证。将只在服务器 CAPABILITY 响应包含 AUTH=CRAM-MD5 阶段时才有效。

IMAP4.**logout**()
关闭对服务器的连接。返回服务器 BYE 响应。

3.8 版更變: 此方法不会再忽略静默的任意异常。

IMAP4.**lsub**(*directory=""*, *pattern='*'*)
列出 directory 中抽取的与 pattern 相匹配的邮箱。*directory* 默认为最高层级目录而 *pattern* 默认为匹配任何邮箱。返回的数据为消息部分封包和数据的元组。

IMAP4.**myrights**(*mailbox*)
显示某个邮箱的本人 ACL (即本人在邮箱中的权限)。

IMAP4.**namespace**()
返回 **RFC 2342** 中定义的 IMAP 命名空间。

IMAP4.**noop**()
将 NOOP 发送给服务器。

IMAP4.**open**(*host*, *port*, *timeout=None*)
打开连接 *host* 上 *port* 的套接字。可选的 *timeout* 形参指定连接尝试的超时秒数。如果 timeout 未给出或为 None，则会使用全局默认的套接字超时。另外请注意如果 *timeout* 形参被设为零，它将引发*ValueError* 以拒绝创建非阻塞套接字。此方法会由*IMAP4* 构造器隐式地调用。此方法所建立的连接对象将在*IMAP4.read()*, *IMAP4.readline()*, *IMAP4.send()* 和*IMAP4.shutdown()* 等方法中被使用。你可以重载此方法。

引发一个审计事件 imaplib.open，附带参数 self, host, port。

3.9 版更變: 加入 *timeout* 参数。

IMAP4.**partial**(*message_num*, *message_part*, *start*, *length*)
获取消息被截断的部分。返回的数据是由消息部分封包和数据组成的元组。

IMAP4.**proxyauth**(*user*)
作为 *user* 进行认证。允许经权限的管理员通过代理进入任意用户的邮箱。

IMAP4.**read**(*size*)
从远程服务器读取 *size* 字节。你可以重载此方法。

IMAP4.**readline**()
从远程服务器读取一行。你可以重载此方法。

IMAP4.**recent**()
提示服务器进行更新。如果没有新消息则返回的数据为 None，否则为 RECENT 响应的值。

IMAP4.**rename**(*oldmailbox*, *newmailbox*)
将名为 *oldmailbox* 的邮箱重命名为 *newmailbox*。

IMAP4.**response**(*code*)
如果收到响应 *code* 则返回其数据，否则返回 None。返回给定的代码，而不是普通的类型。

IMAP4.**search**(*charset*, *criterion*[, ...])
在邮箱中搜索匹配的消息。*charset* 可以为 None，在这种情况下在发给服务器的请求中将不指定 CHARSET。IMAP 协议要求至少指定一个标准；当服务器返回错误时将会引发异常。*charset* 为 None 对应使用*enable()* 命令启用了 UTF8=ACCEPT 功能的情况。

示例:

```
# M is a connected IMAP4 instance...
typ, msgnums = M.search(None, 'FROM', '"LDJ"')

# or:
typ, msgnums = M.search(None, '(FROM "LDJ")')
```

IMAP4.**select**(*mailbox='INBOX'*, *readonly=False*)
选择一个邮箱。返回的数据是 *mailbox* 中消息的数量 (EXISTS 响应)。默认的 *mailbox* 为 'INBOX'。如果设置了 *readonly* 旗标，则不允许修改该邮箱。

IMAP4.**send**(*data*)
将 data 发送给远程服务器。请可以重载此方法。

引发一个审计事件 imaplib.send，附带参数 self, data。

IMAP4.**setacl**(*mailbox*, *who*, *what*)
发送 *mailbox* 的 ACL。此方法是非标准的，但是被 Cyrus 服务器所支持。

IMAP4.**setannotation**(*mailbox*, *entry*, *attribute*[, ...])
设置 *mailbox* 的 ANNOTATION。此方法是非标准的，但是被 Cyrus 服务器所支持。

IMAP4.**setquota**(*root*, *limits*)
设置 quota *root* 的资源限制为 *limits*。此方法是 rfc2087 定义的 IMAP4 QUOTA 扩展的组成部分。

IMAP4.**shutdown**()
关闭在 open 中建立的连接。此方法会由*IMAP4.logout()* 隐式地调用。你可以重载此方法。

IMAP4.**socket**()
返回用于连接服务器的套接字实例。

IMAP4.**sort**(*sort_criteria*, *charset*, *search_criterion*[, ...])
sort 命令是 search 的变化形式，带有结果排序语句。返回的数据包含以空格分隔的匹配消息编号列表。

排序命令在 *search_criterion* 参数之前还有两个参数；一个带圆括号的 *sort_criteria* 列表，和搜索的 *charset*。请注意不同于 search，搜索的 *charset* 参数是强制性的。还有一个 uid sort 命令与 sort 对应，如同 uid search 与 search 对应一样。sort 命令首先在邮箱中搜索匹配给定搜索条件的消息，使用 charset 参数来解读搜索条件中的字符串。然后它将返回所匹配消息的编号。

这是一个 IMAP4rev1 扩展命令。

IMAP4.**starttls**(*ssl_context=None*)
发送一个 STARTTLS 命令。*ssl_context* 参数是可选的并且应为一个*ssl.SSLContext* 对象。这将在 IMAP 连接上启用加密。请阅读*安全考量* 来了解最佳实践。

3.2 版新加入.

3.4 版更變: 此方法现在支持使用*ssl.SSLContext.check_hostname* 和服务器名称指示 (参见*ssl.HAS_SNI*) 进行主机名检查。

IMAP4.**status**(*mailbox*, *names*)
针对 *mailbox* 请求指定的状态条件。

IMAP4.**store**(*message_set*, *command*, *flag_list*)
改变邮箱中消息的旗标处理。*command* 由 **RFC 2060** 的 6.4.6 小节指明，应为"FLAGS", "+FLAGS" 或"-FLAGS" 之一，并可选择附带".SILENT" 后缀。

例如，要在所有消息上设置删除旗标:

```
typ, data = M.search(None, 'ALL')
for num in data[0].split():
   M.store(num, '+FLAGS', '\\Deleted')
M.expunge()
```

備F: 创建包含']' 的旗标 (例如: "[test]") 会违反 **RFC 3501** (IMAP 协议)。但是，imaplib 在历史上曾经允许创建这样的标签，并且流行的 IMAP 服务器如 Gmail 都会接受并生成这样的旗标。有些非 Python 程序也会创建这样的旗标。虽然它违反 RFC 并且 IMAP 客户端和服务器应当严格遵守规范，但是 imaplib 出于向下兼容的理由仍然继续允许创建这样的标签，并且在 Python 3.6 中会在其被服务器所发送时处理它们，因为这能提升实际的兼容性。

IMAP4.**subscribe**(*mailbox*)
订阅新邮箱。

IMAP4.**thread**(*threading_algorithm*, *charset*, *search_criterion*[, ...])
thread 命令是 search 的变化形式，带有针对结果的消息串句法。返回的数据包含以空格分隔的消息串成员列表。

消息串成员由零个或多个消息编号组成，以空格分隔，标示了连续的上下级关系。

thread 命令在 *search_criterion* 参数之前还有两个参数；一个 *threading_algorithm*，以及搜索使用的 *charset*。请注意不同于 search，搜索使用的 *charset* 参数是强制性的。还有一个 uid thread 命令与 thread 对应，如同 uid search 与 search 对应一个。thread 命令首先在邮箱中搜索匹配给定搜索条件的消息，使用 charset 参数来解读搜索条件中的字符串。然后它将按照指定的消息串算法返回所匹配的消息串。

这是一个 IMAP4rev1 扩展命令。

IMAP4.**uid**(*command*, *arg*[, ...])
执行 command arg 并附带用 UID 所标识的消息，而不是用消息编号。返回与命令对应的响应。必须至少提供一个参数；如果不提供任何参数，服务器将返回错误并引发异常。

IMAP4.**unsubscribe**(*mailbox*)
取消订阅原有邮箱。

IMAP4.**unselect**()
imaplib.IMAP4.unselect() 会释放关联到选定邮箱的服务器资源并将服务器返回到已认证状态。此命令会执行与*imaplib.IMAP4.close()* 相同的动作，区别在于它不会从当前选定邮箱中永久性地移除消息。

3.9 版新加入.

IMAP4.**xatom**(*name*[, ...])
允许服务器在 CAPABILITY 响应中通知简单的扩展命令。

在*IMAP4* 的实例上定义了下列属性:

IMAP4.**PROTOCOL_VERSION**
在服务器的 CAPABILITY 响应中最新的受支持协议。

IMAP4.**debug**
控制调试输出的整数值。初始值会从模块变量 Debug 中获取。大于三的值表示将追踪每一条命令。

IMAP4.**utf8_enabled**
通常为 False 的布尔值，但也可以被设为 True，如果成功地为 UTF8=ACCEPT 功能发送了*enable()* 命令的话。

3.5 版新加入.

21.13.2 IMAP4 示例

以下是一个最短示例（不带错误检查），该示例将打开邮箱，检索并打印所有消息:

```
import getpass, imaplib

M = imaplib.IMAP4()
M.login(getpass.getuser(), getpass.getpass())
M.select()
typ, data = M.search(None, 'ALL')
for num in data[0].split():
    typ, data = M.fetch(num, '(RFC822)')
    print('Message %s\n%s\n' % (num, data[0][1]))
M.close()
M.logout()
```

21.14 smtplib ---SMTP 协议客户端

源代码： Lib/smtplib.py

---

smtplib 模块定义了一个 SMTP 客户端会话对象，该对象可将邮件发送到 Internet 上带有 SMTP 或 ESMTP 接收程序的计算机。关于 SMTP 和 ESMTP 操作的详情请参阅 **RFC 821** (简单邮件传输协议) 和 **RFC 1869** (SMTP 服务扩展)。

class smtplib.**SMTP**(*host=''*, *port=0*, *local_hostname=None*[, *timeout*], *source_address=None*)
一个*SMTP* 实例就是一个封装好的 SMTP 连接。该实例具有的方法支持所有 SMTP 和 ESMTP 操作。如果传入了可选参数 host 和 port，那么将在初始化时使用这些参数调用 SMTP *connect()* 方法。如果传入了 *local_hostname*，它将在 HELO/EHLO 命令中被用作本地主机的 FQDN。否则将使用*socket.getfqdn()* 找到本地主机名。如果*connect()* 返回了成功码以外的内容，则引发*SMTPConnectError* 异常。可选参数 *timeout* 指定阻塞操作（如连接尝试）的超时（以秒为单位，如果未指定超时，将使用全局默认超时设置）。到达超时时长后会引发*socket.timeout* 异常。可选参数 source_address 允许在有多张网卡的计算机中绑定到某些特定的源地址，和/或绑定到某些特定的源 TCP 端口。在连接前，套接字需要绑定一个 2 元组 (host, port) 作为其源地址。如果省略，或者是主机为 '' 和/或端口为 0，则将使用操作系统默认行为。

正常使用时，只需要初始化或 connect 方法，*sendmail()* 方法，再加上*SMTP.quit()* 方法即可。下文包括了一个示例。

SMTP 类支持 with 语句。当这样使用时，with 语句一退出就会自动发出 SMTP QUIT 命令。例如:

```
>>> from smtplib import SMTP
>>> with SMTP("domain.org") as smtp:
...     smtp.noop()
...
(250, b'Ok')
>>>
```

引发一个审计事件 smtplib.send，附带参数 self, data。

3.3 版更變: 支持了 with 语句。

3.3 版更變: 添加了 source_address 参数。

3.5 版新加入: 现在已支持 SMTPUTF8 扩展 (**RFC 6531**)。

3.9 版更變: 如果 *timeout* 形参被设为零，则它将引发*ValueError* 来阻止创建非阻塞的套接字

class smtplib.**SMTP_SSL**(*host=''*, *port=0*, *local_hostname=None*, *keyfile=None*, *certfile=None*[, *timeout*], *context=None*, *source_address=None*)

SMTP_SSL 实例与*SMTP* 实例的行为完全相同。在开始连接就需要 SSL，且 starttls() 不适合的情况下，应该使用*SMTP_SSL*。如果未指定 *host*，则使用 localhost。如果 *port* 为 0，则使用标准 SMTP-over-SSL 端口（465）。可选参数 *local_hostname*、*timeout* 和 *source_address* 的含义与*SMTP* 类中的相同。可选参数 *context* 是一个*SSLContext* 对象，可以从多个方面配置安全连接。请阅读安全考量 以获取最佳实践。

keyfile 和 *certfile* 是 *context* 的传统替代物，它们可以指向 PEM 格式的私钥和证书链文件用于 SSL 连接。

3.3 版更變: 增加了 *context*。

3.3 版更變: 添加了 source_address 参数。

3.4 版更變: 本类现在支持通过*ssl.SSLContext.check_hostname* 和 服务器名称提示（参阅*ssl.HAS_SNI*）进行主机名检查。

3.6 版後已🄵用: *keyfile* 和 *certfile* 已弃用并转而推荐 *context*。请改用*ssl.SSLContext.load_cert_chain()* 或让*ssl.create_default_context()* 为你选择系统所信任的 CA 证书。

3.9 版更變: 如果 *timeout* 形参被设为零，则它将引发*ValueError* 来阻止创建非阻塞的套接字

class smtplib.**LMTP**(*host=''*, *port=LMTP_PORT*, *local_hostname=None*, *source_address=None*[, *timeout*])

LMTP 协议与 ESMTP 非常相似，它很大程度上基于标准的 SMTP 客户端。将 Unix 套接字用于 LMTP 是很常见的，因此 connect() 方法支持 Unix 套接字，也支持常规的 host:port 服务器。可选参数 local_hostname 和 source_address 的含义与*SMTP* 类中的相同。要指定 Unix 套接字，*host* 必须使用绝对路径，以'/' 开头。

支持使用常规的 SMTP 机制来进行认证。当使用 Unix 套接字时，LMTP 通常不支持或要求任何认证，但你的情况可能会有所不同。

3.9 版更變: 添加了可选的 *timeout* 形参。

同样地定义了一组精心选择的异常:

exception smtplib.**SMTPException**

OSError 的子类，它是本模块提供的所有其他异常的基类。

3.4 版更變: SMTPException 已成为*OSError* 的子类

exception smtplib.**SMTPServerDisconnected**

当服务器意外断开连接，或在*SMTP* 实例连接到服务器之前尝试使用它时将引发此异常。

exception smtplib.**SMTPResponseException**
包括 SMTP 错误代码的所有异常的基类。这些异常会在 SMTP 服务器返回错误代码时在实例中生成。错误代码存放在错误的 smtp_code 属性中，并且 smtp_error 属性会被设为错误消息。

exception smtplib.**SMTPSenderRefused**
发送方地址被拒绝。除了在所有*SMTPResponseException* 异常上设置的属性，还会将'sender' 设为代表拒绝方 SMTP 服务器的字符串。

exception smtplib.**SMTPRecipientsRefused**
所有接收方地址被拒绝。每个接收方的错误可通过属性 recipients 来访问，该属性是一个字典，其元素顺序与*SMTP.sendmail()* 所返回的一致。

exception smtplib.**SMTPDataError**
SMTP 服务器拒绝接收消息数据。

exception smtplib.**SMTPConnectError**
在建立与服务器的连接期间发生了错误。

exception smtplib.**SMTPHeloError**
服务器拒绝了我们的 HELO 消息。

exception smtplib.**SMTPNotSupportedError**
尝试的命令或选项不被服务器所支持。

3.5 版新加入.

exception smtplib.**SMTPAuthenticationError**
SMTP 认证出现问题。最大的可能是服务器不接受所提供的用户名/密码组合。

也参考:

RFC 821 - 简单邮件传输协议 SMTP 的协议定义。该文件涵盖了 SMTP 的模型、操作程序和协议细节。

RFC 1869 - SMTP 服务扩展 定义了 SMTP 的 ESMTP 扩展。这描述了一个用新命令扩展 SMTP 的框架，支持动态发现服务器所提供的命令，并定义了一些额外的命令。

21.14.1 SMTP 对象

一个*SMTP* 实例拥有以下方法：

SMTP.**set_debuglevel**(*level*)
设置调试输出级别。如果 *level* 的值为 1 或 True ，就会产生连接的调试信息，以及所有发送和接收服务器的信息。如果 *level* 的值为 2 ，则这些信息会被加上时间戳。

3.5 版更變: 添调试级别 2 。

SMTP.**docmd**(*cmd*, *args=''*)
向服务器发送一条命令 *cmd* 。可选的参数 *args* 被简单地串联到命令中，用一个空格隔开。

这将返回一个由数字响应代码和实际响应行组成的 2 元组（多行响应被连接成一个长行）。

在正常操作中，应该没有必要明确地调用这个方法。它被用来实现其他方法，对于测试私有扩展可能很有用。

如果在等待回复的过程中，与服务器的连接丢失，*SMTPServerDisconnected* 将被触发。

SMTP.**connect**(*host='localhost'*, *port=0*)
连接到某个主机的某个端口。默认是连接到 localhost 的标准 SMTP 端口（25）上。如果主机名以冒号 (':') 结尾，后跟数字，则该后缀将被删除，且数字将视作要使用的端口号。如果在实例化时指定了 host，则构造函数会自动调用本方法。返回包含响应码和响应消息的 2 元组，它们由服务器在其连接响应中发送。

触发一个*auditing event* smtplib.connect，其参数为 self ， host ， port 。

SMTP.**helo**(*name=''*)
使用 HELO 向 SMTP 服务器表明自己的身份。hostname 参数默认为本地主机的完全合格域名。服务器返回的消息被存储为对象的 helo_resp 属性。

在正常操作中，应该没有必要明确调用这个方法。它将在必要时被*sendmail()* 隐式调用。

SMTP.**ehlo**(*name=''*)
使用 EHLO 向 ESMTP 服务器标识自身。hostname 参数默认为 localhost 的标准域名。使用*has_extn()* 来检查响应中的 ESMTP 选项，并将它们保存起来。还给一些信息性的属性赋值：服务器返回的消息存储为 ehlo_resp 属性；根据服务器是否支持 ESMTP，将 does_esmtp 设置为 true 或 false；而 esmtp_features 是一个字典，包含该服务器支持的 SMTP 服务扩展的名称及参数（如果有参数）。

除非你想在发送邮件前使用*has_extn()* ，否则应该没有必要明确调用这个方法。它将在必要时被*sendmail()* 隐式调用。

SMTP.**ehlo_or_helo_if_needed**()
如果这个会话中没有先前的 EHLO 或 HELO 命令，该方法会调用*ehlo()* 和/或*helo()* 。它首先尝试 ESMTP EHLO 。

SMTPHeloError 服务器没有正确回复 HELO 问候。

SMTP.**has_extn**(*name*)
如果 *name* 在服务器返回的 SMTP 服务扩展集合中，返回*True* ，否则为*False* 。大小写被忽略。

SMTP.**verify**(*address*)
使用 SMTP VRFY 检查此服务器上的某个地址是否有效。如果用户地址有效则返回一个由代码 250 和完整 **RFC 822** 地址（包括人名）组成的元组。否则返回 400 或更大的 SMTP 错误代码以及一个错误字符串。

備𝔽: 许多网站都禁用 SMTP VRFY 以阻止垃圾邮件。

SMTP.**login**(*user*, *password*, ***, *initial_response_ok=True*)
登录到一个需要认证的 SMTP 服务器。参数是用于认证的用户名和密码。如果会话在之前没有执行过 EHLO 或 HELO 命令，此方法会先尝试 ESMTP EHLO。如果认证成功则此方法将正常返回，否则可能引发以下异常:

SMTPHeloError 服务器没有正确回复 HELO 问候。

SMTPAuthenticationError 服务器不接受所提供的用户名/密码组合。

SMTPNotSupportedError 服务器不支持 AUTH 命令。

SMTPException 未找到适当的认证方法。

smtplib 所支持的每种认证方法只要被服务器声明支持就会被依次尝试。请参阅*auth()* 获取受支持的认证方法列表。*initial_response_ok* 会被传递给*auth()* 。

可选的关键字参数 *initial_response_ok* 对于支持它的认证方法，是否可以与 AUTH 命令一起发送 **RFC 4954** 中所规定的“初始响应”，而不是要求回复/响应。

3.5 版更變: 可能会引发*SMTPNotSupportedError*，并添加 *initial_response_ok* 形参。

SMTP.**auth**(*mechanism*, *authobject*, ***, *initial_response_ok=True*)
为指定的认证机制 *mechanism* 发送 SMTP AUTH 命令，并通过 *authobject* 处理回复响应。

mechanism 指定要使用何种认证机制作为 AUTH 命令的参数；可用的值是在 esmtp_features 的 auth 元素中列出的内容。

authobject 必须是接受一个可选的单独参数的可调用对象:

data = authobject(challenge=None)

如果可选的关键字参数 *initial_response_ok* 为真值，则将先不带参数地调用 `authobject()`。它可以返回 **RFC 4954** " 初始响应" ASCII `str`，其内容将被编码并使用下述的 `AUTH` 命令来发送。如果 `authobject()` 不支持初始响应（例如由于要求一个回复），它应当将 `None` 作为附带 `challenge=None` 调用的返回值。如果 *initial_response_ok* 为假值，则 `authobject()` 将不会附带 `None` 被首先调用。

如果初始响应检测返回了 `None`，或者如果 *initial_response_ok* 为假值，则将调用 `authobject()` 来处理服务器的回复响应；它所传递的 *challenge* 参数将为一个 `bytes`。它应当返回用 base64 进行编码的 ASCII `str` *data* 并发送给服务器。

`SMTP` 类提供的 `authobjects` 针对 `CRAM-MD5`, `PLAIN` 和 `LOGIN` 等机制；它们的名称分别是 `SMTP.auth_cram_md5`, `SMTP.auth_plain` 和 `SMTP.auth_login`。它们都要求将 `user` 和 `password` 这两个 `SMTP` 实例属性设为适当的值。

用户代码通常不需要直接调用 `auth`，而是调用*`login()`* 方法，它将按上述顺序依次尝试上述每一种机制。`auth` 被公开以便辅助实现*`smtplib`* 没有（或尚未）直接支持的认证方法。

3.5 版新加入.

SMTP.**starttls**(*keyfile=None*, *certfile=None*, *context=None*)

将 SMTP 连接设为 TLS (传输层安全) 模式。后续的所有 SMTP 命令都将被加密。你应当随即再次调用*`ehlo()`*。

如果提供了 *keyfile* 和 *certfile*，它们会被用来创建*`ssl.SSLContext`*。

可选的 *context* 形参是一个*`ssl.SSLContext`* 对象；它是使用密钥文件和证书的替代方式，如果指定了该形参则 *keyfile* 和 *certfile* 都应为 `None`。

如果这个会话中没有先前的 `EHLO` or `HELO` 命令，该方法会首先尝试 ESMTP `EHLO`。

3.6 版後已🄵用: *keyfile* 和 *certfile* 已弃用并转而推荐 *context*。请改用*`ssl.SSLContext.load_cert_chain()`* 或让*`ssl.create_default_context()`* 为你选择系统所信任的 CA 证书。

`SMTPHeloError` 服务器没有正确回复 `HELO` 问候。

`SMTPNotSupportedError` 服务器不支持 STARTTLS 扩展。

`RuntimeError` SSL/TLS 支持在你的 Python 解释器上不可用。

3.3 版更變: 增加了 *context*。

3.4 版更變: 此方法现在支持使用 `SSLContext.check_hostname` 和服务器名称指示符 (参见*`HAS_SNI`*) 进行主机名检查。

3.5 版更變: 因缺少 STARTTLS 支持而引发的错误现在是*`SMTPNotSupportedError`* 子类而不是*`SMTPException`* 基类。

SMTP.**sendmail**(*from_addr*, *to_addrs*, *msg*, *mail_options=()*, *rcpt_options=()*)

发送邮件。必要参数是一个 **RFC 822** 发件地址字符串，一个 **RFC 822** 收件地址字符串列表（裸字符串将被视为含有 1 个地址的列表），以及一个消息字符串。调用者可以将 ESMTP 选项列表（如 `8bitmime`）作为 *mail_options* 传入，用于 `MAIL FROM` 命令。需要与所有 `RCPT` 命令一起使用的 ESMTP 选项（如 `DSN` 命令）可以作为 *rcpt_options* 传入。（如果需要对不同的收件人使用不同的 ESMTP 选项，则必须使用底层的方法来发送消息，如 `mail()`, `rcpt()` 和 `data()`。）

備🄵: *from_addr* 和 *to_addrs* 形参被用来构造传输代理所使用的消息封包。`sendmail` 不会以任何方式修改消息标头。

---

msg 可以是一个包含 ASCII 范围内字符的字符串，或是一个字节串。字符串会使用 ascii 编解码器编码为字节串，并且单独的 `\r` 和 `\n` 字符会被转换为 `\r\n` 字符序列。字节串则不会被修改。

如果在此之前本会话没有执行过 EHLO 或 HELO 命令，此方法会先尝试 ESMTP EHLO。如果服务器执行了 ESMTP，消息大小和每个指定的选项将被传递给它（如果指定的选项属于服务器声明的特性集）。如果 EHLO 失败，则将尝试 HELO 并屏蔽 ESMTP 选项。

如果邮件被至少一个接收方接受则此方法将正常返回。在其他情况下它将引发异常。也就是说，如果此方法没有引发异常，则应当会有人收到你的邮件。如果此方法没有引发异常，它将返回一个字典，其中的条目对应每个拒绝的接收方。每个条目均包含由服务器发送的 SMTP 错误代码和相应错误消息所组成的元组。

如果 SMTPUTF8 包括在 *mail_options* 中，并且被服务器所支持，则 *from_addr* 和 *to_addrs* 可能包含非 ASCII 字符。

此方法可能引发以下异常:

SMTPRecipientsRefused 所有收件人都被拒绝。无人收到邮件。该异常的 recipients 属性是一个字典，其中有被拒绝收件人的信息（类似于至少有一个收件人接受邮件时所返回的信息）。

SMTPHeloError 服务器没有正确回复 HELO 问候。

SMTPSenderRefused 服务器不接受 *from_addr*。

SMTPDataError 服务器回复了一个意外的错误代码（而不是拒绝收件人）。

SMTPNotSupportedError 在 *mail_options* 中给出了 SMTPUTF8 但是不被服务器所支持。

除非另有说明，即使在引发异常之后连接仍将被打开。

3.2 版更變: *msg* 可以为字节串。

3.5 版更變: 增加了 SMTPUTF8 支持，并且如果指定了 SMTPUTF8 但是不被服务器所支持则可能会引发*SMTPNotSupportedError*。

SMTP.**send_message**(*msg*, *from_addr=None*, *to_addrs=None*, *mail_options=()*, *rcpt_options=()*)
本方法是一种快捷方法，用于带着消息调用*sendmail()*，消息由*email.message.Message* 对象表示。参数的含义与*sendmail()* 中的相同，除了 *msg*，它是一个 Message 对象。

如果 *from_addr* 为 None 或 *to_addrs* 为 None，那么 “send_message“将根据 **RFC 5322**，从 *msg* 头部提取地址填充下列参数：如果头部存在 *Sender* 字段，则用它填充 *from_addr*，不存在则用 *From* 字段填充 *from_addr*。*to_addrs* 组合了 *msg* 中的 *To*, *Cc* 和 *Bcc* 字段的值（字段存在的情况下）。如果一组 *Resent-** 头部恰好出现在 message 中，那么就忽略常规的头部，改用 *Resent-** 头部。如果 message 包含多组 *Resent-** 头部，则引发*ValueError*，因为无法明确检测出哪一组 *Resent-* 头部是最新的。

send_message 使用*BytesGenerator* 来序列化 *msg*，且将 \r\n 作为 *linesep*，并调用*sendmail()* 来传输序列化后的结果。无论 *from_addr* 和 *to_addrs* 的值为何，send_message 都不会传输 *msg* 中可能出现的 *Bcc* 或 *Resent-Bcc* 头部。如果 *from_addr* 和 *to_addrs* 中的某个地址包含非 ASCII 字符，且服务器没有声明支持 SMTPUTF8，则引发 SMTPNotSupported 错误。如果服务器支持，则 Message 将按新克隆的*policy* 进行序列化，其中的*utf8* 属性被设置为 True，且 SMTPUTF8 和 BODY=8BITMIME 被添加到 *mail_options* 中。

3.2 版新加入.

3.5 版新加入: 支持国际化地址 (SMTPUTF8)。

SMTP.**quit**()
终结 SMTP 会话并关闭连接。返回 SMTP QUIT 命令的结果。

与标准 SMTP/ESMTP 命令 HELP, RSET, NOOP, MAIL, RCPT 和 DATA 对应的低层级方法也是受支持的。通常不需要直接调用这些方法，因此它们没有被写入本文档。相关细节请参看模块代码。

21.14.2 SMTP 示例

这个例子提示用户输入消息封包所需的地址 ('To' 和'From' 地址)，以及所要封包的消息。请注意包括在消息中的标头必须包括在输入的消息中；这个例子不对 **RFC 822** 标头进行任何处理。特别地，'To' 和'From' 地址必须显式地包括在消息标头中。

```
import smtplib

def prompt(prompt):
    return input(prompt).strip()

fromaddr = prompt("From: ")
toaddrs  = prompt("To: ").split()
print("Enter message, end with ^D (Unix) or ^Z (Windows):")

# Add the From: and To: headers at the start!
msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
while True:
    try:
        line = input()
    except EOFError:
        break
    if not line:
        break
    msg = msg + line

print("Message length is", len(msg))

server = smtplib.SMTP('localhost')
server.set_debuglevel(1)
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

備🄵: 通常，你将需要使用*email* 包的特性来构造电子邮件消息，然后你可以通过*send_message()* 来发送它，参见*email: 示例*。

21.15 uuid --- RFC 4122 定义的 UUID 对象

Source code: Lib/uuid.py

这个模块提供了不可变的*UUID* 对象 (*UUID* 类) 和*uuid1()*, *uuid3()*, *uuid4()*, *uuid5()* 等函数用于生成 **RFC 4122** 所定义的第 1, 3, 4 和 5 版 UUID。

如果你想要的只是一个唯一的 ID，你可能应该调用*uuid1()* 或*uuid4()*。注意*uuid1()* 可能会损害隐私，因为它创建了一个包含计算机网络地址的 UUID。*uuid4()* 可以创建一个随机 UUID。

根据底层平台的支持，*uuid1()* 可能会也可能不会返回一个" 安全的" UUID。安全的 UUID 是使用同步方法生成的，确保没有两个进程可以获得相同的 UUID。所有*UUID* 的实例都有一个 is_safe 属性，使用这个枚举来传递关于 UUID 安全的任何信息:

class uuid.**SafeUUID**
    3.7 版新加入.

safe
该 UUID 是由平台以多进程安全的方式生成的。

unsafe
UUID 不是以多进程安全的方式生成的。

unknown
该平台不提供 UUID 是否安全生成的信息。

class uuid.**UUID**(*hex=None*, *bytes=None*, *bytes_le=None*, *fields=None*, *int=None*, *version=None*, *, *is_safe=SafeUUID.unknown*)
用一串 32 位十六进制数字、一串大端序 16 个字节作为 *bytes* 参数、一串 16 个小端序字节作为 *bytes_le* 参数、一个由六个整数组成的元组（32 位 *time_low*，16 位 *time_mid*，16 位 *time_hi_version*，8 位 *clock_seq_hi_variant*，8 位 *clock_seq_low*，48 位 *node*）作为 *fields* 参数，或者一个 128 位整数作为 *int* 参数创建一个 UUID。当给出一串十六进制数字时，大括号、连字符和 URN 前缀都是可选的。例如，这些表达式都产生相同的 UUID:

```
UUID('{12345678-1234-5678-1234-567812345678}')
UUID('12345678123456781234567812345678')
UUID('urn:uuid:12345678-1234-5678-1234-567812345678')
UUID(bytes=b'\x12\x34\x56\x78'*4)
UUID(bytes_le=b'\x78\x56\x34\x12\x34\x12\x78\x56' +
              b'\x12\x34\x56\x78\x12\x34\x56\x78')
UUID(fields=(0x12345678, 0x1234, 0x5678, 0x12, 0x34, 0x567812345678))
UUID(int=0x12345678123456781234567812345678)
```

必须给出 *hex*、*bytes*、*bytes_le*、*fields* 或 *int* 中的唯一一个。*version* 参数是可选的；如果给定，产生的 UUID 将根据 **RFC 4122** 设置其变体和版本号，覆盖给定的 *hex*、*bytes*、*bytes_le*、*fields* 或 *int* 中的位。

UUID 对象的比较是通过比较它们的*UUID.int* 属性进行的。与非 UUID 对象的比较会引发*TypeError*。

`str(uuid)` 返回一个 `12345678-1234-5678-1234-567812345678` 形式的字符串，其中 32 位十六进制数字代表 UUID。

UUID 实例有这些只读的属性:

UUID.**bytes**
UUID 是一个 16 字节的字符串（包含 6 个整数字段，大端字节顺序）。

UUID.**bytes_le**
UUID 是一个 16 字节的字符串（其中 *time_low*、*time_mid* 和 *time_hi_version* 为小端字节顺序）。

UUID.**fields**
以元组形式存放的 UUID 的 6 个整数域，有六个单独的属性和两个派生属性:

域	意义
time_low	UUID 的前 32 位
time_mid	接前一域的 16 位
time_hi_version	接前一域的 16 位
clock_seq_hi_variant	接前一域的 8 位
clock_seq_low	接前一域的 8 位
node	UUID 的最后 48 位
time	UUID 的总长 60 位的时间戳
clock_seq	14 位的序列号

UUID.**hex**
The UUID as a 32-character lowercase hexadecimal string.

UUID.**int**
UUID 是一个 128 位的整数。

UUID.**urn**
在 **RFC 4122** 中定义的 URN 形式的 UUID。

UUID.**variant**
UUID 的变体，它决定了 UUID 的内部布局。这将是*RESERVED_NCS*，*RFC_4122*，*RESERVED_MICROSOFT* 或*RESERVED_FUTURE* 中的一个。

UUID.**version**
UUID 版本号（1 到 5，只有当变体为*RFC_4122* 时才有意义）。

UUID.**is_safe**
一个*SafeUUID* 的枚举，表示平台是否以多进程安全的方式生成 UUID。

3.7 版新加入.

uuid 模块定义了以下函数:

uuid.**getnode**()
获取 48 位正整数形式的硬件地址。第一次运行时，它可能会启动一个单独的程序，这可能会相当慢。如果所有获取硬件地址的尝试都失败了，我们会按照 **RFC 4122** 中的建议，选择一个随机的 48 位数字，其多播位 (第一个八进制数的最小有效位) 设置为 1。“硬件地址” 是指一个网络接口的 MAC 地址。在一台有多个网络接口的机器上，普遍管理的 MAC 地址 (即第一个八位数的第二个最小有效位是 未设置的) 将比本地管理的 MAC 地址优先，但没有其他排序保证。

3.7 版更變: 普遍管理的 MAC 地址优于本地管理的 MAC 地址，因为前者保证是全球唯一的，而后者则不是。

uuid.**uuid1**(*node=None*, *clock_seq=None*)
根据主机 ID、序列号和当前时间生成一个 UUID。如果没有给出 *node*，则使用*getnode()* 来获取硬件地址。如果给出了 *clock_seq*，它将被用作序列号；否则将选择一个随机的 14 比特位序列号。

uuid.**uuid3**(*namespace*, *name*)
根据命名空间标识符（这是一个 UUID）和名称（这是一个字符串）的 MD5 哈希值，生成一个 UUID。

uuid.**uuid4**()
生成一个随机的 UUID。

uuid.**uuid5**(*namespace*, *name*)
根据命名空间标识符（这是一个 UUID）和名称（这是一个字符串）的 SHA-1 哈希值生成一个 UUID。

uuid 模块定义了以下命名空间标识符，供*uuid3()* 或*uuid5()* 使用。

uuid.**NAMESPACE_DNS**
当指定这个命名空间时，*name* 字符串是一个完全限定的域名。

uuid.**NAMESPACE_URL**
当指定这个命名空间时，*name* 字符串是一个 URL。

uuid.**NAMESPACE_OID**
当指定这个命名空间时，*name* 字符串是一个 ISO OID。

uuid.**NAMESPACE_X500**
当指定这个命名空间时，*name* 字符串是 DER 或文本输出格式的 X.500 DN。

uuid 模块为 `variant` 属性的可能值定义了以下常量:

uuid.**RESERVED_NCS**
为 NCS 兼容性保留。

uuid.**RFC_4122**
指定 **RFC 4122** 中给出的 UUID 布局。

uuid.**RESERVED_MICROSOFT**
为微软的兼容性保留。

uuid.**RESERVED_FUTURE**
保留给未来的定义。

也参考:

RFC 4122 - 通用唯一标识符（UUID）URN 命名空间 本规范定义了 UUID 的统一资源名称空间，UUID 的内部格式，以及生成 UUID 的方法。

21.15.1 示例

下面是一些*uuid*模块的典型使用例子:

```
>>> import uuid

>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```

21.16 socketserver --- 用于网络服务器的框架

源代码: Lib/socketserver.py

---

socketserver 模块简化了编写网络服务器的任务。

该模块具有四个基础实体服务器类:

class socketserver.**TCPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server. If *bind_and_activate* is true, the constructor automatically attempts to invoke *server_bind()* and *server_activate()*. The other parameters are passed to the *BaseServer* base class.

class socketserver.**UDPServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
该类使用数据包，即一系列离散的信息分包，它们可能会无序地到达或在传输中丢失。该类的形参与*TCPServer* 的相同。

class socketserver.**UnixStreamServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
class socketserver.**UnixDatagramServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
这两个更常用的类与 TCP 和 UDP 类相似，但使用 Unix 域套接字；它们在非 Unix 系统平台上不可用。它们的形参与*TCPServer* 的相同。

这四个类会 同步地处理请求；每个请求必须完成才能开始下一个请求。这就不适用于每个请求要耗费很长时间来完成的情况，或者因为它需要大量的计算，又或者它返回了大量的数据而客户端处理起来很缓慢。解决方案是创建单独的进程或线程来处理每个请求；*ForkingMixIn* 和*ThreadingMixIn* 混合类可以被用于支持异步行为。

创建一个服务器需要分几个步骤进行。首先，你必须通过子类化*BaseRequestHandler* 类并重载其*handle()* 方法来创建一个请求处理句柄类；这个方法将处理传入的请求。其次，你必须实例化某个服务器类，将服务器地址和请求处理句柄类传给它。建议在 `with` 语句中使用该服务器。然后再调用服务器对象的*handle_request()* 或*serve_forever()* 方法来处理一个或多个请求。最后，调用*server_close()* 来关闭套接字（除非你使用了 `with` 语句）。

当从*ThreadingMixIn* 继承线程连接行为时，你应当显式地声明你希望在突然关机时你的线程采取何种行为。*ThreadingMixIn* 类定义了一个属性 *daemon_threads*，它指明服务器是否应当等待线程终止。如果你希望线程能自主行动你应当显式地设置这个旗标；默认值为*False*，表示 Python 将不会在*ThreadingMixIn* 所创建的所有线程都退出之前退出。

服务器类具有同样的外部方法和属性，无论它们使用哪种网络协议。

21.16.1 服务器创建的说明

在继承图中有五个类，其中四个代表四种类型的同步服务器:

```
+------------+
| BaseServer |
+------------+
      |
      v
+-----------+         +------------------+
| TCPServer |-------->| UnixStreamServer |
+-----------+         +------------------+
      |
```

(下页继续)

(繼續上一頁)

```
      v
+-----------+         +---------------------+
| UDPServer |-------->| UnixDatagramServer |
+-----------+         +---------------------+
```

请注意*UnixDatagramServer*派生自*UDPServer*，而不是*UnixStreamServer* --- IP 和 Unix 流服务器的唯一区别是地址族，它会在两种 Unix 服务器类中简单地重复。

class socketserver.**ForkingMixIn**
class socketserver.**ThreadingMixIn**

每种服务器类型的分叉和线程版本都可以使用这些混合类来创建。例如，*ThreadingUDPServer*的创建方式如下:

```
class ThreadingUDPServer(ThreadingMixIn, UDPServer):
    pass
```

混合类先出现，因为它重载了*UDPServer*中定义的一个方法。设置各种属性也会改变下层服务器机制的行为。

*ForkingMixIn*和下文提及的分叉类仅在支持*fork()*的 POSIX 系统平台上可用。

socketserver.ForkingMixIn.server_close() 会等待直到所有子进程完成，除非 socketserver.ForkingMixIn.block_on_close 属性为假值。

socketserver.ThreadingMixIn.server_close() 会等待直到所有非守护类线程完成，除非 socketserver.ThreadingMixIn.block_on_close 属性为假值。请将 ThreadingMixIn.daemon_threads 设为 True 来使用守护类线程以便不等待线完成。

3.7 版更變: socketserver.ForkingMixIn.server_close() 和 socketserver.ThreadingMixIn.server_close() 现在会等待直到所有子进程和非守护类线程完成。请新增一个 socketserver.ForkingMixIn.block_on_close 类属性来选择 3.7 版之前的行为。

class socketserver.**ForkingTCPServer**
class socketserver.**ForkingUDPServer**
class socketserver.**ThreadingTCPServer**
class socketserver.**ThreadingUDPServer**

这些类都是使用混合类来预定义的。

要实现一个服务，你必须从*BaseRequestHandler*派生一个类并重定义其*handle()*方法。然后你可以通过组合某种服务器类型与你的请求处理句柄类来运行各种版本的服务。请求处理句柄类对于数据报和流服务必须是不相同的。这可以通过使用处理句柄子类*StreamRequestHandler*或*DatagramRequestHandler*来隐藏。

当然，你仍然需要动点脑筋！举例来说，如果服务包含可能被不同请求所修改的内存状态则使用分叉服务器是没有意义的，因为在子进程中的修改将永远不会触及保存在父进程中的初始状态并传递到各个子进程。在这种情况下，你可以使用线程服务器，但你可能必须使用锁来保护共享数据的一致性。

另一方面，如果你是在编写一个所有数据保存在外部（例如文件系统）的 HTTP 服务器，同步类实际上将在正在处理某个请求的时候“失聪” -- 如果某个客户端在接收它所请求的所有数据时很缓慢这可能会是非常长的时间。这时线程或分叉服务器会更为适用。

在某些情况下，合适的做法是同步地处理请求的一部分，但根据请求数据在分叉的子进程中完成处理。这可以通过使用一个同步服务器并在请求处理句柄类*handle()*中进行显式分叉来实现。

还有一种可以在既不支持线程也不支持*fork()*的环境（或者对于本服务来说这两者开销过大或是不适用）中处理多个同时请求的方式是维护一个显式的部分完成的请求表并使用*selectors*来决定接下来要处理哪个请求（或者是否要处理一个新传入的请求）。这对于流服务来说特别重要，因为每个客户端可能会连接很长的时间（如果不能使用线程或子进程）。请参阅*asyncore*来了解另一种管理方式。

21.16.2 Server 对象

class socketserver.**BaseServer**(*server_address*, *RequestHandlerClass*)

这是本模块中所有 Server 对象的超类。它定义了下文给出的接口，但没有实现大部分的方法，它们应在子类中实现。两个形参存储在对应的*server_address* 和*RequestHandlerClass* 属性中。

fileno()

返回服务器正在监听的套接字的以整数表示的文件描述符。此函数最常被传递给*selectors*，以允许在同一进程中监控多个服务器。

handle_request()

处理单个请求。此函数会依次调用下列方法: *get_request()*, *verify_request()* 和*process_request()*。如果用户提供的处理句柄类的*handle()* 方法引发了异常，则将调用服务器的*handle_error()* 方法。如果在*timeout* 秒内未接收到请求，将会调用*handle_timeout()* 并将返回*handle_request()*。

serve_forever(*poll_interval=0.5*)

对请求进行处理直至收到显式的*shutdown()* 请求。每隔 *poll_interval* 秒对 shutdown 进行轮询。忽略*timeout* 属性。它还会调用*service_actions()*，这可被子类或混合类用来提供某个给定服务的专属操作。例如，*ForkingMixIn* 类使用*service_actions()* 来清理僵尸子进程。

3.3 版更變: 将 service_actions 调用添加到 serve_forever 方法。

service_actions()

此方法会在 the *serve_forever()* 循环中被调用。此方法可被子类或混合类所重载以执行某个给定服务的专属操作，例如清理操作。

3.3 版新加入.

shutdown()

通知*serve_forever()* 循环停止并等待它完成。*shutdown()* 必须在*serve_forever()* 运行于不同线程时被调用否则它将发生死锁。

server_close()

清理服务器。可以被重载。

address_family

服务器套接字所属的协议族。常见的例子有*socket.AF_INET* 和*socket.AF_UNIX*。

RequestHandlerClass

用户提供的请求处理句柄类；将为每个请求创建该类的实例。

server_address

The address on which the server is listening. The format of addresses varies depending on the protocol family; see the documentation for the *socket* module for details. For Internet protocols, this is a tuple containing a string giving the address, and an integer port number: `('127.0.0.1', 80)`, for example.

socket

将由服务器用于监听入站请求的套接字对象。

服务器类支持下列类变量:

allow_reuse_address

服务器是否要允许地址的重用。默认值为*False*，并可在子类中设置以改变策略。

request_queue_size

请求队列的长度。如果处理单个请求要花费很长的时间，则当服务器正忙时到达的任何请求都会被加入队列，最多加入*request_queue_size* 个请求。一旦队列被加满，来自客户端的更多请求将收到"Connection denied" 错误。默认值为 5，但可在子类中重载。

socket_type
服务器使用的套接字类型；常见的有*socket.SOCK_STREAM* 和*socket.SOCK_DGRAM* 这两个值。

timeout
超时限制，以秒数表示，或者如果不限制超时则为*None*。如果在超时限制期间没有收到*handle_request()*，则会调用*handle_timeout()* 方法。

有多个服务器方法可被服务器基类的子类例如*TCPServer* 所重载；这些方法对服务器对象的外部用户来说并无用处。

finish_request(*request*, *client_address*)
通过实例化*RequestHandlerClass* 并调用其*handle()* 方法来实际处理请求。

get_request()
必须接受来自套接字的请求，并返回一个 2 元组，其中包含用来与客户端通信的 *new* 套接字对象，以及客户端的地址。

handle_error(*request*, *client_address*)
此函数会在*RequestHandlerClass* 实例的*handle()* 方法引发异常时被调用。默认行为是将回溯信息打印到标准错误并继续处理其他请求。

3.6 版更變: 现在只针对派生自*Exception* 类的异常调用此方法。

handle_timeout()
此函数会在*timeout* 属性被设为*None* 以外的值并且在超出时限之后仍未收到请求时被调用。分叉服务器的默认行为是收集任何已退出的子进程状态，而在线程服务器中此方法则不做任何操作。

process_request(*request*, *client_address*)
调用*finish_request()* 来创建*RequestHandlerClass* 的实例。如果需要，此函数可创建一个新的进程或线程来处理请求；*ForkingMixIn* 和*ThreadingMixIn* 类能完成此任务。

server_activate()
由服务器的构造器调用以激活服务器。TCP 服务器的默认行为只是在服务器的套接字上发起调用*listen()*。可以被重载。

server_bind()
由服务器的构造器调用以将套接字绑定到所需的地址。可以被重载。

verify_request(*request*, *client_address*)
必须返回一个布尔值；如果值为*True*，请求将被处理。而如果值为*False*，请求将被拒绝。此函数可被重载以实现服务器的访问控制。默认实现总是返回*True*。

3.6 版更變: 添加了对*context manager* 协议的支持。退出上下文管理器与调用*server_close()* 等效。

21.16.3 请求处理句柄对象

class socketserver.**BaseRequestHandler**
这是所有请求处理句柄对象的超类。它定义了下文列出的接口。一个实体请求处理句柄子类必须定义新的*handle()* 方法，并可重载任何其他方法。对于每个请求都会创建一个新的子类的实例。

setup()
会在*handle()* 方法之前被调用以执行任何必要的初始化操作。默认实现不执行任何操作。

handle()
此函数必须执行为请求提供服务所需的全部操作。默认实现不执行任何操作。它有几个可用的实例属性；请求为 `self.request`；客户端地址为 `self.client_address`；服务器实例为 `self.server`，如果它需要访问特定服务器信息的话。

针对数据报或流服务的 self.request 类型是不同的。对于流服务，self.request 是一个套接字对象；对于数据报服务，self.request 是一对字符串与套接字。

finish()
在*handle()* 方法之后调用以执行任何需要的清理操作。默认实现不执行任何操作。如果*setup()* 引发了异常，此函数将不会被调用。

class socketserver.**StreamRequestHandler**
class socketserver.**DatagramRequestHandler**
BaseRequestHandler 子类重载了*setup()* 和*finish()* 方法，并提供了 self.rfile 和 self.wfile 属性。self.rfile 和 self.wfile 属性可以被分别读取或写入，以获取请求数据或将数据返回给客户端。

这两个类的 rfile 属性都支持*io.BufferedIOBase* 可读接口，并且 DatagramRequestHandler.wfile 还支持*io.BufferedIOBase* 可写接口。

3.6 版更變: StreamRequestHandler.wfile 也支持*io.BufferedIOBase* 可写接口。

21.16.4 示例

socketserver.TCPServer 示例

以下是服务端:

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    with socketserver.TCPServer((HOST, PORT), MyTCPHandler) as server:
        # Activate the server; this will keep running until you
        # interrupt the program with Ctrl-C
        server.serve_forever()
```

一个使用流（通过提供标准文件接口来简化通信的文件类对象）的替代请求处理句柄类:

```
class MyTCPHandler(socketserver.StreamRequestHandler):

    def handle(self):
```

(下页继续)

(繼續上一頁)

```
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # Likewise, self.wfile is a file-like object used to write back
        # to the client
        self.wfile.write(self.data.upper())
```

区别在于第二个处理句柄的 `readline()` 调用将多次调用 `recv()` 直至遇到一个换行符，而第一个处理句柄的单次 `recv()` 调用只是返回在一次 `sendall()` 调用中由客户端发送的内容。

以下是客户端:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# Create a socket (SOCK_STREAM means a TCP socket)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    # Connect to server and send data
    sock.connect((HOST, PORT))
    sock.sendall(bytes(data + "\n", "utf-8"))

    # Receive data from the server and shut down
    received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

这个示例程序的输出应该是像这样的:

服务器:

```
$ python TCPServer.py
127.0.0.1 wrote:
b'hello world with TCP'
127.0.0.1 wrote:
b'python is nice'
```

客户端:

```
$ python TCPClient.py hello world with TCP
Sent:     hello world with TCP
Received: HELLO WORLD WITH TCP
$ python TCPClient.py python is nice
Sent:     python is nice
Received: PYTHON IS NICE
```

socketserver.UDPServer 示例

以下是服务端:

```
import socketserver

class MyUDPHandler(socketserver.BaseRequestHandler):
    """
    This class works similar to the TCP handler class, except that
    self.request consists of a pair of data and client socket, and since
    there is no connection the client address must be given explicitly
    when sending data back via sendto().
    """

    def handle(self):
        data = self.request[0].strip()
        socket = self.request[1]
        print("{} wrote:".format(self.client_address[0]))
        print(data)
        socket.sendto(data.upper(), self.client_address)

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999
    with socketserver.UDPServer((HOST, PORT), MyUDPHandler) as server:
        server.serve_forever()
```

以下是客户端:

```
import socket
import sys

HOST, PORT = "localhost", 9999
data = " ".join(sys.argv[1:])

# SOCK_DGRAM is the socket type to use for UDP sockets
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# As you can see, there is no connect() call; UDP has no connections.
# Instead, data is directly sent to the recipient via sendto().
sock.sendto(bytes(data + "\n", "utf-8"), (HOST, PORT))
received = str(sock.recv(1024), "utf-8")

print("Sent:     {}".format(data))
print("Received: {}".format(received))
```

这个示例程序的输出应该是与 TCP 服务器示例相一致的。

异步混合类

要构建异步处理句柄，请使用*ThreadingMixIn* 和*ForkingMixIn* 类。

ThreadingMixIn 类的示例:

```
import socket
import threading
import socketserver

class ThreadedTCPRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        data = str(self.request.recv(1024), 'ascii')
        cur_thread = threading.current_thread()
        response = bytes("{}: {}".format(cur_thread.name, data), 'ascii')
        self.request.sendall(response)

class ThreadedTCPServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
    pass

def client(ip, port, message):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        sock.sendall(bytes(message, 'ascii'))
        response = str(sock.recv(1024), 'ascii')
        print("Received: {}".format(response))

if __name__ == "__main__":
    # Port 0 means to select an arbitrary unused port
    HOST, PORT = "localhost", 0

    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    with server:
        ip, port = server.server_address

        # Start a thread with the server -- that thread will then start one
        # more thread for each request
        server_thread = threading.Thread(target=server.serve_forever)
        # Exit the server thread when the main thread terminates
        server_thread.daemon = True
        server_thread.start()
        print("Server loop running in thread:", server_thread.name)

        client(ip, port, "Hello World 1")
        client(ip, port, "Hello World 2")
        client(ip, port, "Hello World 3")

        server.shutdown()
```

这个示例程序的输出应该是像这样的:

```
$ python ThreadedTCPServer.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
```

ForkingMixIn 类的使用方式是相同的，区别在于服务器将为每个请求产生一个新的进程。仅在支持*fork()*

的 POSIX 系统平台上可用。

21.17 http.server --- HTTP 服务器

源代码： Lib/http/server.py

这个模块定义了实现 HTTP 服务器（Web 服务器）的类。

> **警告：** *http.server* is not recommended for production. It only implements *basic security checks*.

HTTPServer 是*socketserver.TCPServer* 的一个子类。它会创建和侦听 HTTP 套接字，并将请求调度给处理程序。用于创建和运行服务器的代码看起来像这样：

```
def run(server_class=HTTPServer, handler_class=BaseHTTPRequestHandler):
    server_address = ('', 8000)
    httpd = server_class(server_address, handler_class)
    httpd.serve_forever()
```

class http.server.**HTTPServer**(*server_address*, *RequestHandlerClass*)
该类基于*TCPServer* 类，并会将服务器地址存入名为 server_name 和 server_port 的实例变量中。服务器可被处理程序通过 server 实例变量访问。

class http.server.**ThreadingHTTPServer**(*server_address*, *RequestHandlerClass*)
这个类与 HTTPServer 相似，但会通过使用*ThreadingMixIn* 来以线程处理请求。这适用于处理 Web 浏览器预开启套接字，以便*HTTPServer* 无限期地执行等待。

3.7 版新加入.

HTTPServer 和*ThreadingHTTPServer* 必须在实例化时给定一个 *RequestHandlerClass*，此模块提供了该对象的三种不同形式：

class http.server.**BaseHTTPRequestHandler**(*request*, *client_address*, *server*)
这个类用于处理到达服务器的 HTTP 请求。它本身无法响应任何实际的 HTTP 请求；它必须被子类化以处理每个请求方法（例如 GET 或 POST）。*BaseHTTPRequestHandler* 提供了许多供子类使用的类和实例变量以及方法。

这个处理程序将解析请求和标头，然后调用特定请求类型对应的方法。方法名称将根据请求来构造。例如，对于请求方法 SPAM，将不带参数地调用 do_SPAM() 方法。所有相关信息会被保存在该处理程序的实际变量中。子类不需要重载或扩展 __init__() 方法。

BaseHTTPRequestHandler 具有下列实例变量:

client_address
包含 (host, port) 形式的指向客户端地址的元组。

server
包含服务器实例。

close_connection
应当在*handle_one_request()* 返回之前设定的布尔值，指明是否要期待另一个请求，还是应当关闭连接。

requestline
包含 HTTP 请求行的字符串表示。末尾的 CRLF 会被去除。该属性应当

由*handle_one_request()* 来设定。如果无有效请求行被处理，则它应当被设为空字符串。

command
包含具体的命令（请求类型）。例如 `'GET'`。

path
包含请求路径。如果 URL 的查询部分存在，`path` 会包含这个查询部分。使用 **RFC 3986** 的术语来说，在这里，`path` 包含 `hier-part` 和 `query` 。

request_version
包含请求的版本字符串。例如 `'HTTP/1.0'`。

headers
存放由*MessageClass* 类变量所指定的类的实例。该实例会解析并管理 HTTP 请求中的标头。*http.client* 中的*parse_headers()* 函数将被用来解析标头并且它需要 HTTP 请求提供一个有效的 **RFC 2822** 风格的标头。

rfile
一个*io.BufferedIOBase* 输入流，准备从可选的输入数据的开头进行读取。

wfile
包含用于写入响应并发回给客户端的输出流。在写入流时必须正确遵守 HTTP 协议以便成功地实现与 HTTP 客户端的互操作。

3.6 版更變: 这是一个*io.BufferedIOBase* 流。

BaseHTTPRequestHandler 具有下列属性:

server_version
指定服务器软件版本。你可能会想要重载该属性。该属性的格式为多个以空格分隔的字符串，其中每个字符串的形式为 name[/version]。例如 `'BaseHTTP/0.2'`。

sys_version
包含 Python 系统版本，采用*version_string* 方法和*server_version* 类变量所支持的形式。例如 `'Python/1.4'`。

error_message_format
指定应当被*send_error()* 方法用来构建发给客户端的错误响应的格式字符串。该字符串应使用来自*responses* 的变量根据传给*send_error()* 的状态码来填充默认值。

error_content_type
指定发送给客户端的错误响应的 Content-Type HTTP 标头。默认值为 `'text/html'`。

protocol_version
该属性指定在响应中使用的 HTTP 协议版本。如果设为 `'HTTP/1.1'`，服务器将允许 HTTP 永久连接；但是，这样你的服务器 *必须* 在发给客户端的所有响应中包括一个准确的 `Content-Length` 标头 (使用*send_header()*)。为了向下兼容，该设置默认为 `'HTTP/1.0'`。

MessageClass
指定一个*email.message.Message* 这样的类来解析 HTTP 标头。通常该属性不会被重载，其默认值为 `http.client.HTTPMessage`。

responses
该属性包含一个整数错误代码与由短消息和长消息组成的二元组的映射。例如，`{code: (shortmessage, longmessage)}`。*shortmessage* 通常是作为消息响应中的 *message* 键，而 *longmessage* 则是作为 *explain* 键。该属性会被*send_response_only()* 和*send_error()* 方法所使用。

BaseHTTPRequestHandler 实例具有下列方法:

handle()
调用*handle_one_request()* 一次（或者如果启用了永久连接则为多次）来处理传入的 HTTP 请求。你应该完全不需要重载它；而是要实现适当的 `do_*()` 方法。

handle_one_request()
此方法将解析并将请求分配给适当的 `do_*()` 方法。你应该完全不需要重载它。

handle_expect_100()
当一个符合 HTTP/1.1 标准的服务器收到 `Expect: 100-continue` 请求标头时它会以 `100 Continue` 加 `200 OK` 的标头作为响应。如果服务器不希望客户端继续则可以重载此方法引发一个错误。例如服务器可以选择发送 `417 Expectation Failed` 作为响应标头并 `return False`。

3.2 版新加入.

send_error(*code*, *message=None*, *explain=None*)
发送并记录回复给客户端的完整错误信息。数字形式的 *code* 指明 HTTP 错误代码，可选的 *message* 为简短的易于人类阅读的错误描述。*explain* 参数可被用于提供更详细的错误信息；它将使用*error_message_format* 属性来进行格式化并在一组完整的标头之后作为响应体被发送。*responses* 属性存放了 *message* 和 *explain* 的默认值，它们将在未提供时被使用；对于未知代码两者的默认值均为字符串 `???`。如果方法为 HEAD 或响应代码是下列值之一则响应体将为空: `1xx`, `204 No Content`, `205 Reset Content`, `304 Not Modified`。

3.4 版更變: 错误响应包括一个 Content-Length 标头。增加了 *explain* 参数。

send_response(*code*, *message=None*)
将一个响应标头添加到标头缓冲区并记录被接受的请求。HTTP 响应行会被写入到内部缓冲区，后面是 *Server* 和 *Date* 标头。这两个标头的值将分别通过*version_string()* 和*date_time_string()* 方法获取。如果服务器不打算使用*send_header()* 方法发送任何其他标头，则*send_response()* 后面应该跟一个*end_headers()* 调用。

3.3 版更變: 标头会被存储到内部缓冲区并且需要显式地调用*end_headers()*。

send_header(*keyword*, *value*)
将 HTTP 标头添加到内部缓冲区，它将在*end_headers()* 或*flush_headers()* 被发起调用时写入输出流。*keyword* 应当指定标头关键字，并以 *value* 指定其值。请注意，在 send_header 调用结束之后，必须调用*end_headers()* 以便完成操作。

3.2 版更變: 标头将被存入内部缓冲区。

send_response_only(*code*, *message=None*)
只发送响应标头，用于当 `100 Continue` 响应被服务器发送给客户端的场合。标头不会被缓冲而是直接发送到输出流。如果未指定 *message*，则会发送与响应 *code* 相对应的 HTTP 消息。

3.2 版新加入.

end_headers()
将一个空行（指明响应中 HTTP 标头的结束）添加到标头缓冲区并调用*flush_headers()*。

3.2 版更變: 已缓冲的标头会被写入到输出流。

flush_headers()
最终将标头发送到输出流并清空内部标头缓冲区。

3.3 版新加入.

log_request(*code='-'*, *size='-'*)
记录一次被接受（成功）的请求。*code* 应当指定与请求相关联的 HTTP 代码。如果请求的大小可用，则它应当作为 *size* 形参传入。

log_error(...)
当请求无法完成时记录一次错误。默认情况下，它会将消息传给*log_message()*，因此它接受同样的参数 (*format* 和一些额外的值)。

log_message(*format*, ...)
将任意一条消息记录到 sys.stderr。此方法通常会被重载以创建自定义的错误日志记录机制。*format* 参数是标准 printf 风格的格式字符串，其中会将传给*log_message()* 的额外参数用作格式化操作的输入。每条消息日志记录的开头都会加上客户端 IP 地址和当前日期时间。

version_string()
返回服务器软件的版本字符串。该值为*server_version* 与*sys_version* 属性的组合。

date_time_string(*timestamp=None*)
返回由 *timestamp* 所给定的日期和时间（参数应为 None 或为*time.time()* 所返回的格式），格式化为一个消息标头。如果省略 *timestamp*，则会使用当前日期和时间。

结果看起来像 'Sun, 06 Nov 1994 08:49:37 GMT'。

log_date_time_string()
返回当前的日期和时间，为日志格式化

address_string()
返回客户端的地址

3.3 版更變: 在之前版本中，会执行一次名称查找。为了避免名称解析的时延，现在将总是返回 IP 地址。

class http.server.**SimpleHTTPRequestHandler**(*request*, *client_address*, *server*, *directory=None*)
这个类会为目录 *directory* 及以下的文件提供发布服务，或者如果未提供 *directory* 则为当前目录，直接将目录结构映射到 HTTP 请求。

3.7 版新加入: *directory* 形参。

3.9 版更變: *directory* 形参接受一个*path-like object*。

诸如解析请求之类的大量工作都是由基类*BaseHTTPRequestHandler* 完成的。本类实现了*do_GET()* 和*do_HEAD()* 函数。

以下是*SimpleHTTPRequestHandler* 的类属性。

server_version
这会是 "SimpleHTTP/" + __version__，其中 __version__ 定义于模块级别。

extensions_map
将后缀映射为 MIME 类型的字典，其中包含了覆盖系统默认值的自定义映射关系。不区分大小写，因此字典键只应为小写值。

3.9 版更變: 此字典不再填充默认的系统映射，而只包含覆盖值。

SimpleHTTPRequestHandler 类定义了以下方法：

do_HEAD()
本方法为 'HEAD' 请求提供服务：它将发送等同于 GET 请求的头文件。关于合法头部信息的更完整解释，请参阅*do_GET()* 方法。

do_GET()
这一请求通过把其解释为当前工作目录的相对路径来映射到本地文件

如果请求被映射到目录，则会依次检查该目录是否存在 index.html 或 index.htm 文件。若存在则返回文件内容；否则会调用 list_directory() 方法生成目录列表。本方法将利用*os.listdir()* 扫描目录，如果*listdir()* 失败，则返回 404 出错应答。

如果请求被映射到文件，则会打开该文件。打开文件时的任何*OSError* 异常都会被映射为 404，'File not found' 错误。如果请求中带有 'If-Modified-Since' 标头，而在此时间点之

后文件未作修改，则会发送 304，'Not Modified' 的响应。否则会调用 guess_type() 方法猜测内容的类型，该方法会反过来用到 *extensions_map* 变量，并返回文件内容。

将会输出 'Content-type:' 头部信息，带上猜出的内容类型，然后是 'Content-Length:' 头部信息，带有文件的大小，以及 'Last-Modified:' 头部信息，带有文件的修改时间。

后面是一个空行，标志着头部信息的结束，然后输出文件的内容。如果文件的 MIME 类型以 text/ 开头，文件将以文本模式打开；否则将使用二进制模式。

用法示例请参阅*http.server* 模块中的*test()* 函数的实现。

3.7 版更變: 为 'If-Modified-Since' 头部信息提供支持。

SimpleHTTPRequestHandler 类的用法可如下所示，以便创建一个非常简单的 Web 服务，为相对于当前目录的文件提供服务：

```
import http.server
import socketserver

PORT = 8000

Handler = http.server.SimpleHTTPRequestHandler

with socketserver.TCPServer(("", PORT), Handler) as httpd:
    print("serving at port", PORT)
    httpd.serve_forever()
```

http.server can also be invoked directly using the -m switch of the interpreter. Similar to the previous example, this serves files relative to the current directory:

```
python -m http.server
```

The server listens to port 8000 by default. The default can be overridden by passing the desired port number as an argument:

```
python -m http.server 9000
```

By default, the server binds itself to all interfaces. The option -b/--bind specifies a specific address to which it should bind. Both IPv4 and IPv6 addresses are supported. For example, the following command causes the server to bind to localhost only:

```
python -m http.server --bind 127.0.0.1
```

3.4 版新加入: 引入了 --bind 参数。

3.8 版新加入: 为了支持 IPv6 改进了 --bind 参数。

By default, the server uses the current directory. The option -d/--directory specifies a directory to which it should serve the files. For example, the following command uses a specific directory:

```
python -m http.server --directory /tmp/
```

3.7 版新加入: --directory argument was introduced.

class http.server.**CGIHTTPRequestHandler**(*request*, *client_address*, *server*)

该类可为当前及以下目录中的文件或输出 CGI 脚本提供服务。注意，把 HTTP 分层结构映射到本地目录结构，这与*SimpleHTTPRequestHandler* 完全一样。

備𝔽: 由*CGIHTTPRequestHandler*类运行的 CGI 脚本不能进行重定向操作（HTTP 代码 302），因为在执行 CGI 脚本之前会发送代码 200（接下来就输出脚本）。这样状态码就冲突了。

然而，如果这个类猜测它是一个 CGI 脚本，那么就会运行该 CGI 脚本，而不是作为文件提供出去。只会识别基于目录的 CGI ——另有一种常用的服务器设置，即标识 CGI 脚本是通过特殊的扩展名。

如果请求指向 cgi_directories 以下的路径，do_GET() 和 do_HEAD() 函数已作修改，不是给出文件，而是运行 CGI 脚本并输出结果。

CGIHTTPRequestHandler 定义了以下数据成员：

cgi_directories
默认为 ['/cgi-bin', '/htbin']，视作 CGI 脚本所在目录。

CGIHTTPRequestHandler 定义了以下方法：

do_POST()
本方法服务于 'POST' 请求，仅用于 CGI 脚本。如果试图向非 CGI 网址发送 POST 请求，则会输出错误 501：Can only POST to CGI scripts”。

请注意，为了保证安全性，CGI 脚本将以用户 nobody 的 UID 运行。CGI 脚本运行错误将被转换为错误 403。

通过在命令行传入 --cgi 参数，可以启用*CGIHTTPRequestHandler*：

```
python -m http.server --cgi
```

21.17.1 Security Considerations

SimpleHTTPRequestHandler will follow symbolic links when handling requests, this makes it possible for files outside of the specified directory to be served.

21.18 http.cookies --- HTTP 状态管理

源代码: Lib/http/cookies.py

---

http.cookies 模块定义的类将 cookie 的概念抽象了出来，这是一种 HTTP 状态的管理机制。它既支持简单的纯字符串形式的 cookie，也为任何可序列化数据类型的 cookie 提供抽象。

以前，该模块严格套用 **RFC 2109** 和 **RFC 2068** 规范中描述的解析规则。后来人们发现，MSIE 3.0 并不遵循这些规范中的字符规则，而且目前许多浏览器和服务器在处理 cookie 时也放宽了解析规则。因此，这里用到的解析规则没有那么严格。

字符集*string.ascii_letters* 、*string.digits* 和 !#$%&'*+-.^_`|~: 给出了本模块允许出现在 cookie 名称中的有效字符集（如*key*）。

3.3 版更變: “:”字符可用于有效的 cookie 名称。

備𝔽: 当遇到无效 cookie 时会触发*CookieError*，所以若 cookie 数据来自浏览器，一定要做好应对无效数据的准备，并在解析时捕获*CookieError*。

---

exception http.cookies.**CookieError**
出现异常的原因，可能是不符合 **RFC 2109**： 属性不正确、*Set-Cookie* 头部信息不正确等等。

class http.cookies.**BaseCookie**([*input*])
类似字典的对象，字典键为字符串，字典值是*Morsel* 实例。请注意，在将键值关联时，首先会把值转换为包含键和值的*Morsel* 对象。

若给出 *input* ，将会传给*load()* 方法。

class http.cookies.**SimpleCookie**([*input*])
该类派生于*BaseCookie*，并覆盖了 value_decode() 和 value_encode() 方法。SimpleCookie 允许用字符串作为 cookie 值。在设置值时，SimpleCookie 将调用内置的*str()* 将其转换为字符串。从 HTTP 接收到的值将作为字符串保存。

也参考:

***http.cookiejar* 模块** 处理网络 客户端的 HTTP cookie。*http.cookiejar* 和*http.cookies* 模块相互没有依赖关系。

RFC 2109 - HTTP 状态管理机制 这是本模块实现的状态管理规范。

21.18.1 Cookie 对象

BaseCookie.**value_decode**(*val*)
由字符串返回元组 (real_value, coded_value)。real_value 可为任意类型。*BaseCookie* 中的此方法未实现任何解码工作——只为能被子类重写。

BaseCookie.**value_encode**(*val*)
返回元组 (real_value, coded_value)。*val* 可为任意类型，coded_value 则会转换为字符串。*BaseCookie* 中的此方法未实现任何编码工作——只为能被子类重写。

通常在 *value_decode* 的取值范围内，*value_encode()* 和*value_decode()* 应为可互逆操作。

BaseCookie.**output**(*attrs=None*, *header='Set-Cookie:'*, *sep='\r\n'*)
返回可作为 HTTP 标头信息发送的字符串表示。*attrs* 和 *header* 会传给每个*Morsel* 的*output()* 方法。*sep* 用来将标头连接在一起，默认为 '\r\n' (CRLF) 组合。

BaseCookie.**js_output**(*attrs=None*)
返回一段可供嵌入的 JavaScript 代码，若在支持 JavaScript 的浏览器上运行，其作用如同发送 HTTP 头部信息一样。

attrs 的含义与*output()* 的相同。

BaseCookie.**load**(*rawdata*)
若 *rawdata* 为字符串，则会作为 HTTP_COOKIE 进行解析，并将找到的值添加为*Morsel*。如果是字典值，则等价于:

```
for k, v in rawdata.items():
    cookie[k] = v
```

21.18.2 Morsel 对象

class http.cookies.**Morsel**
对键/值对的抽象，带有 **RFC 2109** 的部分属性。

morsel 对象类似于字典，键的组成是常量——均为有效的 **RFC 2109** 属性，包括：

- expires
- path
- comment
- domain
- max-age
- secure
- version
- httponly
- samesite

httponly 属性指明了该 cookie 仅在 HTTP 请求中传输，且不能通过 JavaScript 访问。这是为了减轻某些跨站脚本攻击的危害。

samesite 属性指明了浏览器不得与跨站请求一起发送该 cookie。这有助于减轻 CSRF 攻击的危害。此属性的有效值为“Strict”和“Lax”。

键不区分大小写，默认值为 ''。

3.5 版更變: 现在，__eq__() 会同时考虑*key* 和*value* 。

3.7 版更變: Attributes *key*, *value* and *coded_value* are read-only. Use *set()* for setting them.

3.8 版更變: 增加对 samesite 属性的支持。

Morsel.**value**
Cookie 的值。

Morsel.**coded_value**
编码后的 cookie 值——也即要发送的内容。

Morsel.**key**
cookie 名称

Morsel.**set**(*key*, *value*, *coded_value*)
设置 *key*、*value* 和 *coded_value* 属性。

Morsel.**isReservedKey**(*K*)
K 是否属于*Morsel* 的键。

Morsel.**output**(*attrs=None*, *header='Set-Cookie:'*)
返回 morsel 的字符串形式，适用于作为 HTTP 头部信息进行发送。默认包含所有属性，除非给出 *attrs* 属性列表。*header* 默认为 "Set-Cookie:"。

Morsel.**js_output**(*attrs=None*)
返回一段可供嵌入的 JavaScript 代码，若在支持 JavaScript 的浏览器上运行，其作用如同发送 HTTP 头部信息一样。

attrs 的含义与*output()* 的相同。

Morsel.**OutputString**(*attrs=None*)
返回 morsel 的字符串形式，不含 HTTP 或 JavaScript 数据。

attrs 的含义与*output()* 的相同。

Morsel.**update**(*values*)
用字典 *values* 中的值更新 morsel 字典中的值。若有 *values* 字典中的键不是有效的 **RFC 2109** 属性，则会触发错误。

3.5 版更變: 无效键会触发错误。

Morsel.**copy**(*value*)
返回 morsel 对象的浅表复制副本。

3.5 版更變: 返回一个 morsel 对象，而非字典。

Morsel.**setdefault**(*key*, *value=None*)
若 key 不是有效的 **RFC 2109** 属性则触发错误，否则与*dict.setdefault()* 相同。

21.18.3 示例

以下例子演示了*http.cookies* 模块的用法。

```
>>> from http import cookies
>>> C = cookies.SimpleCookie()
>>> C["fig"] = "newton"
>>> C["sugar"] = "wafer"
>>> print(C) # generate HTTP headers
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> print(C.output()) # same thing
Set-Cookie: fig=newton
Set-Cookie: sugar=wafer
>>> C = cookies.SimpleCookie()
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> print(C.output(header="Cookie:"))
Cookie: rocky=road; Path=/cookie
>>> print(C.output(attrs=[], header="Cookie:"))
Cookie: rocky=road
>>> C = cookies.SimpleCookie()
>>> C.load("chips=ahoy; vienna=finger") # load from a string (HTTP header)
>>> print(C)
Set-Cookie: chips=ahoy
Set-Cookie: vienna=finger
>>> C = cookies.SimpleCookie()
>>> C.load('keebler="E=everybody; L=\\"Loves\\"; fudge=\\012;";')
>>> print(C)
Set-Cookie: keebler="E=everybody; L=\"Loves\"; fudge=\012;"
>>> C = cookies.SimpleCookie()
>>> C["oreo"] = "doublestuff"
>>> C["oreo"]["path"] = "/"
>>> print(C)
Set-Cookie: oreo=doublestuff; Path=/
>>> C = cookies.SimpleCookie()
>>> C["twix"] = "none for you"
>>> C["twix"].value
'none for you'
```

(下页继续)

(繼續上一頁)

```
>>> C = cookies.SimpleCookie()
>>> C["number"] = 7 # equivalent to C["number"] = str(7)
>>> C["string"] = "seven"
>>> C["number"].value
'7'
>>> C["string"].value
'seven'
>>> print(C)
Set-Cookie: number=7
Set-Cookie: string=seven
```

21.19 http.cookiejar ——HTTP 客户端的 Cookie 处理

源代码： Lib/http/cookiejar.py

---

http.cookiejar 模块定义了用于自动处理 HTTP cookie 的类。这对访问需要小段数据——*cookies* 的网站很有用，这些数据由 Web 服务器的 HTTP 响应在客户端计算机上设置，然后在以后的 HTTP 请求中返回给服务器。

常规的 Netscape Cookie 协议和 **RFC 2965** 定义的协议都可以处理。RFC 2965 处理默认情况下处于关闭状态。**RFC 2109** cookie 被解析为 Netscape cookie，随后根据有效的'policy' 被视为 Netscape 或 RFC 2965 cookie。请注意，Internet 上的大多数 cookie 是 Netscape cookie。*http.cookiejar* 尝试遵循事实上的 Netscape cookie 协议（与原始 Netscape 规范中所设定的协议大不相同），包括注意 `max-age` 和 `port` RFC 2965 引入的 cookie 属性。

備𝔽： 在 *Set-Cookie* 和 *Set-Cookie2* 头中找到的各种命名参数通常指 *attributes*。为了不与 Python 属性相混淆，模块文档使用 *cookie-attribute* 代替。

此模块定义了以下异常：

exception http.cookiejar.**LoadError**
FileCookieJar 实例在从文件加载 cookies 出错时抛出这个异常。*LoadError* 是 *OSError* 的一个子类。

3.3 版更變: LoadError 成为 *OSError* 的子类而不是 *IOError*。

提供了以下类：

class http.cookiejar.**CookieJar**(*policy=None*)
policy 是实现了 *CookiePolicy* 接口的一个对象。

CookieJar 类储存 HTTP cookies。它从 HTTP 请求提取 cookies，并在 HTTP 响应中返回它们。*CookieJar* 实例在必要时自动处理包含 cookie 的到期情况。子类还负责储存和从文件或数据库中查找 cookies。

class http.cookiejar.**FileCookieJar**(*filename*, *delayload=None*, *policy=None*)
policy 是实现了 *CookiePolicy* 接口的一个对象。对于其他参数，参考相应属性的文档。

一个可以从硬盘中文件加载或保存 cookie 的 *CookieJar*。Cookies 不会在 *load()* 或 *revert()* 方法调用前从命名的文件中加载。子类的文档位于段落 *FileCookieJar 的子类及其与 Web 浏览器的协同*。

3.8 版更變: 文件名形参支持 *path-like object*。

class http.cookiejar.**CookiePolicy**
此类负责确定是否应从服务器接受每个 cookie 或将其返回给服务器。

class http.cookiejar.**DefaultCookiePolicy**(*blocked_domains=None*, *allowed_domains=None*, *netscape=True*, *rfc2965=False*, *rfc2109_as_netscape=None*, *hide_cookie2=False*, *strict_domain=False*, *strict_rfc2965_unverifiable=True*, *strict_ns_unverifiable=False*, *strict_ns_domain=DefaultCookiePolicy.DomainLiberal*, *strict_ns_set_initial_dollar=False*, *strict_ns_set_path=False*, *secure_protocols=("https", "wss")*)
构造参数只能以关键字参数传递，*blocked_domains* 是一个我们既不会接受也不会返回 cookie 的域名序列。*allowed_domains* 如果不是*None*，则是仅有的我们会接受或返回的域名序列。*secure_protocols* 是可以添加安全 cookies 的协议序列。默认将 *https* 和 *wss*（安全 WebSocket）考虑为安全协议。对于其他参数，参考*CookiePolicy* 和*DefaultCookiePolicy* 对象的文档。

DefaultCookiePolicy 实现了 Netscape 和 **RFC 2965** cookies 的标准接受 / 拒绝规则。默认情况下，**RFC 2109** cookies（即在 *Set-Cookie* 头中收到的 cookie-attribute 版本为 1 的 cookies）将按照 RFC 2965 规则处理。然而，如果 RFC 2965 的处理被关闭，或者*rfc2109_as_netscape* 为 `True`，*Cookie* 实例的 `version` 属性设置将被为 0，RFC 2109 cookies *CookieJar* 实例将" 降级" 为 Netscape cookies。*DefaultCookiePolicy* 也提供一些参数以允许一些策略微调。

class http.cookiejar.**Cookie**
这个类代表 Netscape、**RFC 2109** 和 **RFC 2965** 的 cookie 。我们不希望*http.cookiejar* 的用户构建他们自己的*Cookie* 实例。相反，如果有必要，请在一个*CookieJar* 实例上调用 `make_cookies()` 。

也参考:

模块*urllib.request* URL 打开带有自动的 cookie 处理。

模块*http.cookies* HTTP cookie 类，主要是对服务端代码有用。*http.cookiejar* 和*http.cookies* 模块不相互依赖。

https://curl.se/rfc/cookie_spec.html 原始 Netscape cookie 协议的规范。虽然这仍然是主流协议，但所有主要浏览器（以及*http.cookiejar*）实现的"Netscape cookie 协议" 与 "cookie_spec.html"中描述的协议仅有几分相似之处。

RFC 2109 - HTTP 状态管理机制 被 **RFC 2965** 所取代。使用 *Set-Cookie* version=1 。

RFC 2965 - HTTP 状态管理机制 修正了错误的 Netscape 协议。使用 *Set-Cookie2* 来代替 *Set-Cookie* 。没有广泛被使用。

http://kristol.org/cookie/errata.html 未完成的:rfc:*2965* 勘误表。

HTTP 状态管理使用方法

21.19.1 CookieJar 和 FileCookieJar 对象

CookieJar 对象支持*iterator* 协议，用于迭代包含的*Cookie* 对象。

CookieJar 有以下方法:

CookieJar.**add_cookie_header**(*request*)
在 *request* 中添加正确的 *Cookie* 头。

如果策略允许（即 `rfc2965` 和 `hide_cookie2` 属性在*CookieJar* 的*CookiePolicy* 实例中分别为 True 和 False），*Cookie2* 标头也会在适当时候添加。

如*urllib.request* 所记载的，*request* 对象（通常是一个*urllib.request.Request* 实例）必须支持 get_full_url()，get_host()，get_type()，unverifiable()，has_header()，get_header()，header_items()，add_unredirected_header() 和 origin_req_host 属性。

3.3 版更變: *request* 对象需要 origin_req_host 属性。对已废弃的方法 get_origin_req_host() 的依赖已被移除。

CookieJar.**extract_cookies**(*response*, *request*)
从 HTTP *response* 中提取 cookie，并在政策允许的情况下，将它们存储在*CookieJar* 中。

CookieJar 将在 *response* 参数中寻找允许的 *Set-Cookie* 和 *Set-Cookie2* 头信息，并适当地存储 cookies（须经*CookiePolicy.set_ok()* 方法批准）。

response 对象（通常是调用*urllib.request.urlopen()* 或类似方法的结果）应该支持 info() 方法，它返回*email.message.Message* 实例。

如*urllib.request* 的文档所说，*request* 对象（通常是一个*urllib.request.Request* 实例）必须支持 get_full_url()，get_host()，unverifiable() 和 origin_req_host 属性。该请求用于设置 cookie-attributes 的默认值，以及检查 cookie 是否允许被设置。

3.3 版更變: *request* 对象需要 origin_req_host 属性。对已废弃的方法 get_origin_req_host() 的依赖已被移除。

CookieJar.**set_policy**(*policy*)
设置要使用的*CookiePolicy* 实例。

CookieJar.**make_cookies**(*response*, *request*)
返回从 *response* 对象中提取的*Cookie* 对象的序列。

关于 *response* 和 *request* 参数所需的接口，请参见*extract_cookies()* 的文档。

CookieJar.**set_cookie_if_ok**(*cookie*, *request*)
如果策略规定可以这样做，就设置一个*Cookie* 。

CookieJar.**set_cookie**(*cookie*)
设置一个*Cookie*，不需要检查策略是否应该被设置。

CookieJar.**clear**([*domain*[, *path*[, *name*]]])
清除一些 cookie。

如果调用时没有参数，则清除所有的 cookie。如果给定一个参数，只有属于该 *domain* 的 cookies 将被删除。如果给定两个参数，那么属于指定的 *domain* 和 URL *path* 的 cookie 将被删除。如果给定三个参数，那么属于指定的 *domain* 、*path* 和 *name* 的 cookie 将被删除

如果不存在匹配的 cookie，则会引发*KeyError*。

CookieJar.**clear_session_cookies**()
丢弃所有 session cookie。

丢弃所有 discard 属性为真值的已包含 cookie（通常是因为它们没有 max-age 或 expires cookie 属性，或者显式的 discard cookie 属性）。对于交互式浏览器，会话的结束通常对应于关闭浏览器窗口。

请注意 save() 方法并不会保存会话的 cookie，除非你通过传入一个真值给 *ignore_discard* 参数来提出明确的要求。

FileCookieJar 实现了下列附加方法:

FileCookieJar.**save**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
将 cookie 保存到文件。

基类会引发*NotImplementedError*。子类可以继续不实现该方法。

filename 为要用来保存 cookie 的文件名称。如果未指定 *filename*，则会使用 self.filename (该属性默认为传给构造器的值，如果有传入的话)；如果 self.filename 为*None*，则会引发*ValueError*。

ignore_discard: 即使设定了丢弃 cookie 仍然保存它们。*ignore_expires*: 即使 cookie 已超期仍然保存它们

文件如果已存在则会被覆盖，这将清除其所包含的全部 cookie。已保存的 cookie 可以使用*load()* 或*revert()* 方法来恢复。

FileCookieJar.**load**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
从文件加载 cookie。

旧的 cookie 将被保留，除非是被新加载的 cookie 所覆盖。

其参数与*save()* 的相同。

指定的文件必须为该类所能理解的格式，否则将引发*LoadError*。也可能会引发*OSError*，例如当文件不存在的时候。

3.3 版更變: 过去触发的*IOError*，现在是*OSError* 的别名。

FileCookieJar.**revert**(*filename=None*, *ignore_discard=False*, *ignore_expires=False*)
清除所有 cookie 并从保存的文件重新加载 cookie。

revert() 可以引发与*load()* 相同的异常。如果执行失败，对象的状态将不会被改变。

FileCookieJar 实例具有下列公有属性:

FileCookieJar.**filename**
默认的保存 cookie 的文件的文件名。该属性可以被赋值。

FileCookieJar.**delayload**
如为真值，则惰性地从磁盘加载 cookie。该属性不应当被赋值。这只是一个提示，因为它只会影响性能，而不会影响行为（除非磁盘中的 cookie 被改变了）。*CookieJar* 对象可能会忽略它。任何包括在标准库中的*FileCookieJar* 类都不会惰性地加载 cookie。

21.19.2 FileCookieJar 的子类及其与 Web 浏览器的协同

提供了以下*CookieJar* 子类用于读取和写入。

class http.cookiejar.**MozillaCookieJar**(*filename*, *delayload=None*, *policy=None*)
一个能够以 Mozilla `cookies.txt` 文件格式（该格式也被 Lynx 和 Netscape 浏览器所使用）从磁盘加载和存储 cookie 的*FileCookieJar*。

備𝔼: 这会丢失有关 **RFC 2965** cookie 的信息，以及有关较新或非标准的 cookie 属性例如 `port`。

> **警告:** 在存储之前备份你的 cookie，如果你的 cookie 丢失/损坏会造成麻烦的话（有一些微妙的因素可能导致文件在加载/保存的往返过程中发生细微的变化）。

还要注意在 Mozilla 运行期间保存的 cookie 将可能被 Mozilla 清除。

class http.cookiejar.**LWPCookieJar**(*filename*, *delayload=None*, *policy=None*)
一个能够以 libwww-perl 库的 `Set-Cookie3` 文件格式从磁盘加载和存储 cookie 的*FileCookieJar*。这适用于当你想以人类可读的文件来保存 cookie 的情况。

3.8 版更變: 文件名形参支持*path-like object*。

21.19.3 CookiePolicy 对象

实现了*CookiePolicy* 接口的对象具有下列方法:

CookiePolicy.**set_ok**(*cookie*, *request*)
返回指明是否应当从服务器接受 cookie 的布尔值。

cookie 是一个*Cookie* 实例。*request* 是一个实现了由*CookieJar.extract_cookies()* 的文档所定义的接口的对象。

CookiePolicy.**return_ok**(*cookie*, *request*)
返回指明是否应当将 cookie 返回给服务器的布尔值。

cookie 是一个*Cookie* 实例。*request* 是一个实现了*CookieJar.add_cookie_header()* 的文档所定义的接口的对象。

CookiePolicy.**domain_return_ok**(*domain*, *request*)
对于给定的 cookie 域如果不应当返回 cookie 则返回 False。

此方法是一种优化操作。它消除了检查每个具有特定域的 cookie 的必要性（这可能会涉及读取许多文件）。从*domain_return_ok()* 和*path_return_ok()* 返回真值并将所有工作留给*return_ok()*。

如果*domain_return_ok()* 为 cookie 域返回真值，则会为 cookie 路径调用*path_return_ok()*。在其他情况下，则不会为该 cookie 域调用*path_return_ok()* 和*return_ok()*。如果*path_return_ok()* 返回真值，则会调用*return_ok()* 并附带*Cookie* 对象本身以进行全面检查。在其他情况下，都永远不会为该 cookie 路径调用*return_ok()*。

请注意*domain_return_ok()* 会针对每个 *cookie* 域被调用，而非只针对 *request* 域。例如，该函数会针对 ".example.com" 和 "www.example.com" 被调用，如果 request 域为 "www.example.com" 的话。对于*path_return_ok()* 也是如此。

request 参数与*return_ok()* 的文档所说明的一致。

CookiePolicy.**path_return_ok**(*path*, *request*)
对于给定的 cookie 路径如果不应当返回 cookie 返回 False。

请参阅*domain_return_ok()* 的文档。

除了实现上述方法，*CookiePolicy* 接口的实现还必须提供下列属性，指明应当使用哪种协议以及如何使用。所有这些属性都可以被赋值。

CookiePolicy.**netscape**
实现 Netscape 协议。

CookiePolicy.**rfc2965**
实现 **RFC 2965** 协议。

CookiePolicy.**hide_cookie2**
不要向请求添加 *Cookie2* 标头（此标头是提示服务器请求方能识别 **RFC 2965** cookie）。

定义*CookiePolicy* 类的最适用方式是通过子类化*DefaultCookiePolicy* 并重载部分或全部上述的方法。*CookiePolicy* 本身可被用作' 空策略' 以允许设置和接收所有的 cookie（但这没有什么用处）。

21.19.4 DefaultCookiePolicy 对象

实现接收和返回 cookie 的标准规则。

RFC 2965 和 Netscape cookie 均被涵盖。RFC 2965 处理默认关闭。

提供自定义策略的最容易方式是重载此类并在你重载的实现中添加你自己的额外检查之前调用其方法:

```
import http.cookiejar
class MyCookiePolicy(http.cookiejar.DefaultCookiePolicy):
    def set_ok(self, cookie, request):
        if not http.cookiejar.DefaultCookiePolicy.set_ok(self, cookie, request):
            return False
        if i_dont_want_to_store_this_cookie(cookie):
            return False
        return True
```

在实现*CookiePolicy* 接口所要求的特性之外，该类还允许你阻止和允许特定的域设置和接收 cookie。还有一些严格性开关允许你将相当宽松的 Netscape 协议规则收紧一点（代价是可能会阻止某些无害的 cookie）。

A domain blacklist and whitelist is provided (both off by default). Only domains not in the blacklist and present in the whitelist (if the whitelist is active) participate in cookie setting and returning. Use the *blocked_domains* constructor argument, and `blocked_domains()` and `set_blocked_domains()` methods (and the corresponding argument and methods for *allowed_domains*). If you set a whitelist, you can turn it off again by setting it to *None*.

Domains in block or allow lists that do not start with a dot must equal the cookie domain to be matched. For example, `"example.com"` matches a blacklist entry of `"example.com"`, but `"www.example.com"` does not. Domains that do start with a dot are matched by more specific domains too. For example, both `"www.example.com"` and `"www.coyote.example.com"` match `".example.com"` (but `"example.com"` itself does not). IP addresses are an exception, and must match exactly. For example, if blocked_domains contains `"192.168.1.2"` and `".168.1.2"`, 192.168.1.2 is blocked, but 193.168.1.2 is not.

DefaultCookiePolicy 实现了下列附加方法:

`DefaultCookiePolicy.`**`blocked_domains`**`()`
返回被阻止域的序列（元组类型）。

`DefaultCookiePolicy.`**`set_blocked_domains`**`(`*blocked_domains*`)`
设置被阻止域的序列。

`DefaultCookiePolicy.`**`is_blocked`**`(`*domain*`)`
Return whether *domain* is on the blacklist for setting or receiving cookies.

`DefaultCookiePolicy.`**`allowed_domains`**`()`
返回*None*，或者被允许域的序列（元组类型）。

`DefaultCookiePolicy.`**`set_allowed_domains`**`(`*allowed_domains*`)`
设置被允许域的序列，或者为*None*。

`DefaultCookiePolicy.`**`is_not_allowed`**`(`*domain*`)`
Return whether *domain* is not on the whitelist for setting or receiving cookies.

DefaultCookiePolicy 实例具有下列属性，它们都是基于同名的构造器参数来初始化的，并且都可以被赋值。

`DefaultCookiePolicy.`**`rfc2109_as_netscape`**
如为真值，则请求*CookieJar* 实例将 **RFC 2109** cookie (即在带有 version 值为 1 的 cookie 属性的 *Set-Cookie* 标头中接收到的 cookie) 降级为 Netscape cookie: 即将*Cookie* 实例的 version 属性设为 0。默认值为*None*，在此情况下 RFC 2109 cookie 仅在 s are downgraded if and only if **RFC 2965** 处理被关闭时才会被降级。因此，RFC 2109 cookie 默认会被降级。

通用严格性开关:

DefaultCookiePolicy.**strict_domain**
不允许网站设置带国家码顶级域的包含两部分的域名例如 .co.uk, .gov.uk, .co.nz 等。此开关尚未十分完善，并不保证有效!

RFC 2965 协议严格性开关:

DefaultCookiePolicy.**strict_rfc2965_unverifiable**
遵循针对不可验证事务的 **RFC 2965** 规则（不可验证事务通常是由重定向或请求发布在其它网站的图片导致的）。如果该属性为假值，则 永远不会基于可验证性而阻止 cookie。

Netscape 协议严格性开关:

DefaultCookiePolicy.**strict_ns_unverifiable**
即便是对 Netscape cookie 也要应用 **RFC 2965** 规则。

DefaultCookiePolicy.**strict_ns_domain**
指明针对 Netscape cookie 的域匹配规则的严格程度。可接受的值见下文。

DefaultCookiePolicy.**strict_ns_set_initial_dollar**
忽略 Set-Cookie 中的 cookie: 即名称前缀为 '{TX-PL-LABEL}#x27; 的标头。

DefaultCookiePolicy.**strict_ns_set_path**
不允许设置路径与请求 URL 路径不匹配的 cookie。

strict_ns_domain 是一组旗标。其值是通过或运算来构造的（例如, DomainStrictNoDots|DomainStrictNonDomain 表示同时设置两个旗标）。

DefaultCookiePolicy.**DomainStrictNoDots**
当设置 cookie 是，'host prefix' 不可包含点号（例如 www.foo.bar.com 不能为 .bar.com 设置 cookie，因为 www.foo 包含了一个点号）。

DefaultCookiePolicy.**DomainStrictNonDomain**
没有显式指明 Cookies that did not explicitly specify a domain cookie 属性的 cookie 只能被返回给与设置 cookie 的域相同的域（例如 spam.example.com 不会是来自 example.com 的返回 cookie，如果该域名没有 domain cookie 属性的话）。

DefaultCookiePolicy.**DomainRFC2965Match**
当设置 cookie 时，要求完整的 **RFC 2965** 域匹配。

下列属性是为方便使用而提供的，是上述旗标的几种最常用组合:

DefaultCookiePolicy.**DomainLiberal**
等价于 0 (即所有上述 Netscape 域严格性旗标均停用)。

DefaultCookiePolicy.**DomainStrict**
等价于 DomainStrictNoDots|DomainStrictNonDomain。

21.19.5 Cookie 对象

Cookie 实例具有与各种 cookie 标准中定义的标准 cookie 属性大致对应的 Python 属性。这并非一一对应，因为存在复杂的赋默认值的规则，因为 max-age 和 expires cookie 属性包含相同信息，也因为 **RFC 2109** cookie 可以被*http.cookiejar* 从第 1 版' 降级为' 第 0 版 (Netscape) cookie。

对这些属性的赋值在*CookiePolicy* 方法的极少数情况以外应该都是不必要的。该类不会强制内部一致性，因此如果这样做则你应当清楚自己在做什么。

Cookie.**version**
整数或*None*。Netscape cookie 的*version* 值为 0。**RFC 2965** 和 **RFC 2109** cookie 的 version cookie 属性值为 1。但是，请注意*http.cookiejar* 可以将 RFC 2109 cookie ' 降级' 为 Netscape cookie，在此情况下*version* 值也为 0。

Cookie.**name**
Cookie 名称（一个字符串）。

Cookie.**value**
Cookie 值（一个字符串），或为*None*。

Cookie.**port**
代表一个端口或一组端口（例如'80' 或'80,8080'）的字符串，或为*None*。

Cookie.**path**
Cookie 路径 (字符串类型，例如 `'/acme/rocket_launchers'`)。

Cookie.**secure**
如果 cookie 应当只能通过安全连接返回则为 `True`。

Cookie.**expires**
整数类型的过期时间，以距离 Unix 纪元的秒数表示，或者为*None*。另请参阅*is_expired()* 方法。

Cookie.**discard**
如果是会话 cookie 则为 `True`。

Cookie.**comment**
来自服务器的解释此 cookie 功能的字符串形式的注释，或者为*None*。

Cookie.**comment_url**
链接到来自服务器的解释此 cookie 功能的注释的 URL，或者为*None*。

Cookie.**rfc2109**
如果 cookie 是作为 **RFC 2109** cookie 被接收（即该 cookie 是在 *Set-Cookie* 标头中送达，且该标头的 Version cookie 属性的值为 1）则为 `True`。之所以要提供该属性是因为*http.cookiejar* 可能会从 RFC 2109 cookies ' 降级' 为 Netscape cookie，在此情况下*version* 值为 0。

Cookie.**port_specified**
如果服务器显式地指定了一个端口或一组端口（在 *Set-Cookie* / *Set-Cookie2* 标头中）则为 `True`。

Cookie.**domain_specified**
如果服务器显式地指定了一个域则为 `True`。

Cookie.**domain_initial_dot**
该属性为 `True` 表示服务器显式地指定了以一个点号 (`'.'`) 打头的域。

Cookie 可能还有额外的非标准 cookie 属性。这些属性可以通过下列方法来访问:

Cookie.**has_nonstandard_attr**(*name*)
如果 cookie 具有相应名称的 cookie 属性则返回 `True`。

Cookie.**get_nonstandard_attr**(*name*, *default=None*)
如果 cookie 具有相应名称的 cookie 属性，则返回其值。否则，返回 *default*。

Cookie.**set_nonstandard_attr**(*name*, *value*)
设置指定名称的 cookie 属性的值。

Cookie 类还定义了下列方法:

Cookie.**is_expired**(*now=None*)
如果 cookie 传入了服务器请求其所应过期的时间则为 `True`。如果给出 *now* 值（距离 Unix 纪元的秒数），则返回在指定的时间 cookie 是否已过期。

21.19.6 示例

第一个例子显示了*http.cookiejar*的最常见用法:

```
import http.cookiejar, urllib.request
cj = http.cookiejar.CookieJar()
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

这个例子演示了如何使用你的 Netscape, Mozilla 或 Lynx cookie 打开一个 URL (假定 cookie 文件位置采用 Unix/Netscape 惯例):

```
import os, http.cookiejar, urllib.request
cj = http.cookiejar.MozillaCookieJar()
cj.load(os.path.join(os.path.expanduser("~"), ".netscape", "cookies.txt"))
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

下一个例子演示了*DefaultCookiePolicy*的使用。启用 **RFC 2965** cookie，在设置和返回 Netscape cookie 时更严格地限制域，以及阻止某些域设置 cookie 或返回它们:

```
import urllib.request
from http.cookiejar import CookieJar, DefaultCookiePolicy
policy = DefaultCookiePolicy(
    rfc2965=True, strict_ns_domain=Policy.DomainStrict,
    blocked_domains=["ads.net", ".ads.net"])
cj = CookieJar(policy)
opener = urllib.request.build_opener(urllib.request.HTTPCookieProcessor(cj))
r = opener.open("http://example.com/")
```

21.20 xmlrpc --- XMLRPC 服务端与客户端模块

XML-RPC 是一种远程过程调用方法，它使用通过 HTTP 传递的 XML 作为载体。有了它，客户端可以在远程服务器上调用带参数的方法（服务器以 URI 命名）并获取结构化的数据。

`xmlrpc` 是一个集合了 XML-RPC 服务端与客户端实现模块的包。这些模块是:

- *xmlrpc.client*
- *xmlrpc.server*

21.21 xmlrpc.client --- XML-RPC 客户端访问

源代码: Lib/xmlrpc/client.py

XML-RPC 是一种远程过程调用方法，它以使用 HTTP(S) 传递的 XML 作为载体。通过它，客户端可以在远程服务器（服务器以 URI 指明）上调用带参数的方法并获取结构化的数据。本模块支持编写 XML-RPC 客户端代码；它会处理在通用 Python 对象和 XML 之间进行在线翻译的所有细节。

警告: *xmlrpc.client* 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

3.5 版更變: 对于 HTTPS URI，现在*xmlrpc.client* 默认会执行所有必要的证书和主机名检查。

class xmlrpc.client.**ServerProxy**(*uri*, *transport=None*, *encoding=None*, *verbose=False*, *allow_none=False*, *use_datetime=False*, *use_builtin_types=False*, ***, *headers=()*, *context=None*)

ServerProxy 实例是管理与远程 XML-RPC 服务器通信的对象。要求的第一个参数为 URI (统一资源定位符)，通常就是服务器的 URL。可选的第二个参数为传输工厂实例；在默认情况下对于 https: URL 是一个内部 `SafeTransport` 实例，在其他情况下则是一个内部 HTTP `Transport` 实例。可选的第三个参数为编码格式，默认为 UTF-8。可选的第四个参数为调试旗标。

下列形参控制所返回代理实例的使用。如果 *allow_none* 为真值，则 Python 常量 `None` 将被转写为 XML；默认行为是针对 `None` 引发*TypeError*。这是对 XML-RPC 规格的一个常用扩展，但并不被所有客户端和服务器所支持；请参阅 http://ontosys.com/xml-rpc/extensions.php 了解详情。*use_builtin_types* 旗标可被用来将日期/时间值表示为*datetime.datetime* 对象而将二进制数据表示为*bytes* 对象；此旗标默认为假值。*datetime.datetime*, *bytes* 和*bytearray* 对象可以被传给调用操作。*headers* 形参为可选的随每次请求发送的 HTTP 标头序列，其形式为包含代表标头名称和值的 2 元组的序列 (例如 *[('Header-Name', 'value')]*)。已不再适用的 *use_datetime* 旗标与 *use_builtin_types* 类似但它只针对日期/时间值。

3.3 版更變: 增加了 *use_builtin_types* 旗标。

3.8 版更變: 增加了 *headers* 形参。

HTTP 和 HTTPS 传输均支持用于 HTTP 基本身份验证的 URL 语法扩展: `http://user:pass@host:port/path`。`user:pass` 部分将以 base64 编码为 HTTP 'Authorization' 标头，并在发起调用 XML-RPC 方法时作为连接过程的一部分发送给远程服务器。你只需要在远程服务器要求基本身份验证账号和密码时使用此语法。如果提供了 HTTPS URL，*context* 可以为*ssl.SSLContext* 并配置有下层 HTTPS 连接的 SSL 设置。

返回的实例是一个代理对象，具有可被用来在远程服务器上发起相应 RPC 调用的方法。如果远程服务器支持内省 API，则也可使用该代理对象在远程服务器上查询它所支持的方法（服务发现）并获取其他服务器相关的元数据

适用的类型（即可通过 XML 生成 marshall 对象），包括如下类型（除了已说明的例外，它们会被反 marshall 为同样的 Python 类型）：

XML-RPC 类型	Python 数据类型
`boolean`	*bool*
`int`, `i1`, `i2`, `i4`, `i8` 或者 `biginteger`	*int* 的范围从 -2147483648 到 2147483647。值将获得 `<int>` 标志。
`double` 或 `float`	*float*。值将获得 `<double>` 标志。
`string`	*str*
`array`	*list* 或*tuple* 包含整合元素。数组以*lists* 形式返回。
`struct`	*dict*。键必须为字符串，值可以为任何适用的类型。可以传入用户自定义类的对象；只有其*__dict__* 属性会被传输。
`dateTime.iso8601`	*DateTime* 或*datetime.datetime*。返回的类型取决于 *use_builtin_types* 和 *use_datetime* 标志的值。
`base64`	*Binary*, *bytes* 或*bytearray*。返回的类型取决于 *use_builtin_types* 标志的值。
`nil`	`None` 常量。仅当 *allow_none* 为 true 时才允许传递。
`bigdecimal`	*decimal.Decimal*. 仅返回类型。

这是 This is the full set of data types supported by XML-RPC 所支持数据类型的完整集合。方法调用也可能引发一个特殊的*Fault* 实例，用来提示 XML-RPC 服务器错误，或是用*ProtocolError* 来提示 HTTP/HTTPS 传输层中的错误。*Fault* 和*ProtocolError* 都派生自名为 `Error` 的基类。请注意 xmlrpc client 模块目前不可 marshal 内置类型的子类的实例。

当传入字符串时，XML 中的特殊字符如 <, > 和 & 将被自动转义。但是，调用方有责任确保字符串中没有

XML 中不允许的字符，例如 ASCII 值在 0 和 31 之间的控制字符（当然，制表、换行和回车除外）；不这样做将导致 XML-RPC 请求的 XML 格式不正确。如果你必须通过 XML-RPC 传入任意字节数据，请使用*bytes* 或*bytearray* 类或者下文描述的*Binary* 包装器类。

`Server` 被保留作为*ServerProxy* 的别名用于向下兼容。新的代码应当使用*ServerProxy*。

3.5 版更變: 增加了 *context* 参数。

3.6 版更變: 增加了对带有前缀的类型标签的支持 (例如 `ex:nil`)。增加了对反 marshall 被 Apache XML-RPC 实现用于表示数值的附加类型的支持: `i1`, `i2`, `i8`, `biginteger`, `float` 和 `bigdecimal`。请参阅 http://ws.apache.org/xmlrpc/types.html 了解详情。

也参考:

XML-RPC HOWTO 以多种语言对 XML-RPC 操作和客户端软件进行了很好的说明。包含 XML-RPC 客户端开发者所需知道的几乎任何事情。

XML-RPC Introspection 描述了用于内省的 XML-RPC 协议扩展。

XML-RPC Specification 官方规范说明。

21.21.1 ServerProxy 对象

ServerProxy 实例有一个方法与 XML-RPC 服务器所接受的每个远程过程调用相对应。调用该方法会执行一个 RPC，通过名称和参数签名来调度（例如同一个方法名可通过多个参数签名来重载）。RPC 结束时返回一个值，它可以是适用类型的返回数据或是表示错误的*Fault* 或*ProtocolError* 对象。

支持 XML 内省 API 的服务器还支持一些以保留的 `system` 属性分组的通用方法:

`ServerProxy.system.`**`listMethods`**`()`
此方法返回一个字符串列表，每个字符串都各自对应 XML-RPC 服务器所支持的（非系统）方法。

`ServerProxy.system.`**`methodSignature`**`(`*name*`)`
此方法接受一个形参，即某个由 XML-RPC 服务器所实现的方法名称。它返回一个由此方法可能的签名组成的数组。一个签名就是一个类型数组。这些类型中的第一个是方法的的返回类型，其余的均为形参。

由于允许多个签名（即重载），此方法是返回一个签名列表而非一个单例。

签名本身被限制为一个方法所期望的最高层级形参。举例来说如果一个方法期望有一个结构体数组作为形参，并返回一个字符串，则其签名就是”string, array”。如果它期望有三个整数并返回一个字符串，则其签名是”string, int, int, int”。

如果方法没有定义任何签名，则将返回一个非数组值。在 Python 中这意味着返回值的类型为列表以外的类型。

`ServerProxy.system.`**`methodHelp`**`(`*name*`)`
此方法接受一个形参，即 XML-RPC 服务器所实现的某个方法的名称。它返回描述相应方法用法的文档字符串。如果没有可用的文档字符串，则返回空字符串。文档字符串可以包含 HTML 标记。

3.5 版更變: *ServerProxy* 的实例支持*context manager* 协议用于关闭下层传输。

以下是一个可运行的示例。服务器端代码:

```
from xmlrpc.server import SimpleXMLRPCServer

def is_even(n):
    return n % 2 == 0

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
```

(下页继续)

(繼續上一頁)

```
server.register_function(is_even, "is_even")
server.serve_forever()
```

前述服务器的客户端代码:

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:8000/") as proxy:
    print("3 is even: %s" % str(proxy.is_even(3)))
    print("100 is even: %s" % str(proxy.is_even(100)))
```

21.21.2 日期時間物件

class xmlrpc.client.**DateTime**

该类的初始化可以使用距离 Unix 纪元的秒数、时间元组、ISO 8601 时间/日期字符串或*datetime.datetime* 实例。它具有下列方法，主要是为 marshall 和反 marshall 代码的内部使用提供支持:

decode(*string*)

接受一个字符串作为实例的新时间值。

encode(*out*)

将此*DateTime* 条目的 XML-RPC 编码格式写入到 *out* 流对象。

它还通过富比较和 __repr__() 方法来支持某些 Python 内置操作。

以下是一个可运行的示例。服务器端代码:

```
import datetime
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def today():
    today = datetime.datetime.today()
    return xmlrpc.client.DateTime(today)

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(today, "today")
server.serve_forever()
```

前述服务器的客户端代码:

```
import xmlrpc.client
import datetime

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print("Today: %s" % converted.strftime("%d.%m.%Y, %H:%M"))
```

21.21.3 Binary 对象

class xmlrpc.client.**Binary**
该类的初始化可以使用字节数据（可包括 NUL）。对*Binary* 对象的初始访问是由一个属性来提供的:

data
被*Binary* 实例封装的二进制数据。该数据以*bytes* 对象的形式提供。

Binary 对象具有下列方法，支持这些方法主要是供 marshall 和反 marshall 代码在内部使用:

decode(*bytes*)
接受一个 base64 *bytes* 对象并将其解码为实例的新数据。

encode(*out*)
将此二进制条目的 XML-RPC base 64 编码格式写入到 *out* 流对象。

被编码数据将依据 **RFC 2045 第 6.8 节** 每 76 个字符换行一次，这是撰写 XML-RPC 规范说明时 base64 规范的事实标准。

它还通过 __eq__() 和 __ne__() 方法来支持某些 Python 内置运算符。

该二进制对象的示例用法。我们将通过 XMLRPC 来传输一张图片:

```
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpc.client.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(python_logo, 'python_logo')

server.serve_forever()
```

客户端会获取图片并将其保存为一个文件:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)
```

21.21.4 Fault 对象

class xmlrpc.client.**Fault**
Fault 对象封装了 XML-RPC fault 标签的内容。Fault 对象具有下列属性:

faultCode
A string indicating the fault type.

faultString
一个包含与 fault 相关联的诊断消息的字符串。

在接下来的示例中我们将通过返回一个复数类型的值来故意引发一个*Fault*。服务器端代码:

```
from xmlrpc.server import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x, y):
    return x+y+0j

server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_function(add, 'add')

server.serve_forever()
```

前述服务器的客户端代码:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpc.client.Fault as err:
    print("A fault occurred")
    print("Fault code: %d" % err.faultCode)
    print("Fault string: %s" % err.faultString)
```

21.21.5 ProtocolError 对象

class xmlrpc.client.**ProtocolError**

ProtocolError 对象描述了下层传输层中的协议错误（例如当 URI 所指定的服务器不存在时的 404 'not found' 错误）。它具有下列属性:

url

触发错误的 URI 或 URL。

errcode

错误代码。

errmsg

错误消息或诊断字符串。

headers

一个包含触发错误的 HTTP/HTTPS 请求的标头的字典。

在接下来的示例中我们将通过提供一个无效的 URI 来故意引发一个*ProtocolError*:

```
import xmlrpc.client

# create a ServerProxy with a URI that doesn't respond to XMLRPC requests
proxy = xmlrpc.client.ServerProxy("http://google.com/")

try:
    proxy.some_method()
except xmlrpc.client.ProtocolError as err:
    print("A protocol error occurred")
    print("URL: %s" % err.url)
    print("HTTP/HTTPS headers: %s" % err.headers)
```

(下页继续)

(繼續上一頁)

```
    print("Error code: %d" % err.errcode)
    print("Error message: %s" % err.errmsg)
```

21.21.6 MultiCall 对象

MultiCall 对象提供了一种将对远程服务器的多个调用封装为一个单独请求的方式[1]。

class xmlrpc.client.**MultiCall**(*server*)

创建一个用于盒式方法调用的对象。*server* 是调用的最终目标。可以对结果对象发起调用，但它们将立即返回 None，并只在*MultiCall* 对象中存储调用名称和形参。调用该对象本身会导致所有已存储的调用作为一个单独的 system.multicall 请求被发送。此调用的结果是一个*generator*；迭代这个生成器会产生各个结果。

以下是该类的用法示例。服务器端代码:

```
from xmlrpc.server import SimpleXMLRPCServer

def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    return x // y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(("localhost", 8000))
print("Listening on port 8000...")
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

前述服务器的客户端代码:

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")
multicall = xmlrpc.client.MultiCall(proxy)
multicall.add(7, 3)
multicall.subtract(7, 3)
multicall.multiply(7, 3)
multicall.divide(7, 3)
result = multicall()

print("7+3=%d, 7-3=%d, 7*3=%d, 7//3=%d" % tuple(result))
```

[1] 此做法被首次提及是在 a discussion on xmlrpc.com。

21.21.7 便捷函数

xmlrpc.client.**dumps**(*params*, *methodname=None*, *methodresponse=None*, *encoding=None*, *allow_none=False*)

请 *params* 转换为一个 XML-RPC 请求。或者当 *methodresponse* 为真值时则转换为一个请求。*params* 可以是一个参数元组或是一个*Fault* 异常类的实例。如果 *methodresponse* 为真值，只有单独的值可以被返回，这意味着 *params* 的长度必须为 1。如果提供了 *encoding*，则在生成的 XML 会使用该编码格式；默认的编码格式为 UTF-8。Python 的*None* 值不可在标准 XML-RPC 中使用；要通过扩展来允许使用它，请为 *allow_none* 提供一个真值。

xmlrpc.client.**loads**(*data*, *use_datetime=False*, *use_builtin_types=False*)

将一个 XML-RPC 请求或响应转换为 Python 对象 `(params, methodname)`。*params* 是一个参数元组；*methodname* 是一个字符串，或者如果数据包没有提供方法名则为 `None`。如果 XML-RPC 数据包是代表一个故障条件，则此函数将引发一个*Fault* 异常。*use_builtin_types* 旗标可被用于将日期/时间值表示为*datetime.datetime* 对象并将二进制数据表示为*bytes* 对象；此旗标默认为假值。

已过时的 *use_datetime* 旗标与 *use_builtin_types* 类似但只作用于日期/时间值。

3.3 版更變: 增加了 *use_builtin_types* 旗标。

21.21.8 客户端用法的示例

```
# simple test program (from the XML-RPC specification)
from xmlrpc.client import ServerProxy, Error

# server = ServerProxy("http://localhost:8000") # local server
with ServerProxy("http://betty.userland.com") as proxy:

    print(proxy)

    try:
        print(proxy.examples.getStateName(41))
    except Error as v:
        print("ERROR", v)
```

要通过 HTTP 代理访问一个 XML-RPC 服务器，你必须自行定义一个传输。下面的例子演示了具体做法:

```
import http.client
import xmlrpc.client

class ProxiedTransport(xmlrpc.client.Transport):

    def set_proxy(self, host, port=None, headers=None):
        self.proxy = host, port
        self.proxy_headers = headers

    def make_connection(self, host):
        connection = http.client.HTTPConnection(*self.proxy)
        connection.set_tunnel(host, headers=self.proxy_headers)
        self._connection = host, connection
        return connection

transport = ProxiedTransport()
transport.set_proxy('proxy-server', 8080)
server = xmlrpc.client.ServerProxy('http://betty.userland.com', transport=transport)
print(server.examples.getStateName(41))
```

21.21.9 客户端与服务器用法的示例

参见*SimpleXMLRPCServer 示例*。

F解

21.22 xmlrpc.server --- 基本 XML-RPC 服务器

源代码: Lib/xmlrpc/server.py

---

xmlrpc.server 模块为以 Python 编写 XML-RPC 服务器提供了一个基本服务器框架。服务器可以是独立的，使用*SimpleXMLRPCServer*，或是嵌入某个 CGI 环境中，使用*CGIXMLRPCRequestHandler*。

警告: *xmlrpc.server* 模块对于恶意构建的数据是不安全的。如果你需要解析不受信任或未经身份验证的数据，请参阅*XML 漏洞*。

class xmlrpc.server.**SimpleXMLRPCServer**(*addr*, *requestHandler=SimpleXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=False*)

创建一个新的服务器实例。这个类提供了一些用来注册可以被 XML-RPC 协议所调用的函数的方法。*requestHandler* 形参应该是一个用于请求处理句柄实例的工厂函数；它默认为*SimpleXMLRPCRequestHandler*。*addr* 和 *requestHandler* 形参会被传给*socketserver.TCPServer* 构造器。如果 *logRequests* 为真值（默认），请求将被记录到日志；将此形参设为假值将关闭日志记录。*allow_none* 和 *encoding* 形参会被传给*xmlrpc.client* 并控制将从服务器返回的 XML-RPC 响应。*bind_and_activate* 形参控制 `server_bind()` 和 `server_activate()` 是否会被构造器立即调用；它默认为真值。将其设为假值将允许代码在地址被绑定之前操作 *allow_reuse_address* 类变量。*use_builtin_types* 形参会被传给*loads()* 函数并控制当收到日期/时间值或二进制数据时将处理哪些类型；它默认为假值。

3.3 版更變: 增加了 *use_builtin_types* 旗标。

class xmlrpc.server.**CGIXMLRPCRequestHandler**(*allow_none=False*, *encoding=None*, *use_builtin_types=False*)

创建一个新的实例来处理 CGI 环境中的 XML-RPC 请求。*allow_none* 和 *encoding* 形参会被传递给*xmlrpc.client* 并控制将要从服务器返回的 XML-RPC 响应。*use_builtin_types* 形参会被传递给*loads()* 函数并控制当接收到日期/时间值或二进制数据时要处理哪种类型；该形参默认为假值。

3.3 版更變: 增加了 *use_builtin_types* 旗标。

class xmlrpc.server.**SimpleXMLRPCRequestHandler**

创建一个新的请求处理句柄实例。该请求处理句柄支持 `POST` 请求并会修改日志记录操作以便使用传递给*SimpleXMLRPCServer* 形参的 *logRequests* 形参。

21.22.1 SimpleXMLRPCServer 对象

SimpleXMLRPCServer 类是基于*socketserver.TCPServer* 并提供了一个创建简单、独立的 XML-RPC 服务器的方式。

SimpleXMLRPCServer.**register_function**(*function=None*, *name=None*)
注册一个可以响应 XML-RPC 请求的函数。如果给出了 *name*，则它将成为与 *function* 相关联的方法名，否则将使用 `function.__name__`。*name* 是一个字符串，并可能包含不可在 Python 标识符中出现的字符，包括句点符。

此方法也可被用作装饰器。当被用作装饰器时，*name* 只能被指定为以 *name* 注册的 *function* 的一个关键字参数。如果没有指定 *name*，则将使用 `function.__name__`。

3.7 版更變: *register_function()* 可被用作装饰器。

SimpleXMLRPCServer.**register_instance**(*instance*, *allow_dotted_names=False*)
注册一个被用来公开未使用*register_function()* 注册的方法名的对象。如果 *instance* 包含 `_dispatch()` 方法，它将被调用并附带被请求的方法名和来自请求的形参。它的 API 是 `def _dispatch(self, method, params)` (请注意 *params* 并不表示变量参数列表)。如果它调用了一个下层函数来执行任务，该函数将以 `func(*params)` 的形式被调用，即扩展了形参列表。来自 `_dispatch()` 的返回值将作为结果被返回给客户端。如果 *instance* 不包含 `_dispatch()` 方法，则会在其中搜索与被请求的方法名相匹配的属性。

如果可选的 *allow_dotted_names* 参数为真值且该实例没有 `_dispatch()` 方法，则如果被请求的方法名包含句点符，会单独搜索该方法名的每个组成部分，其效果就是执行了简单的分层级搜索。通过搜索找到的值将随即附带来自请求的形参被调用，并且返回值会被回传给客户端。

> **警告:** 启用 *allow_dotted_names* 选项将允许入侵者访问你的模块的全局变量并可能允许入侵者在你的机器上执行任意代码。仅可在安全、封闭的网络中使用此选项。

SimpleXMLRPCServer.**register_introspection_functions**()
注册 XML-RPC 内省函数 `system.listMethods`, `system.methodHelp` 和 `system.methodSignature`。

SimpleXMLRPCServer.**register_multicall_functions**()
注册 XML-RPC 多调用函数 system.multicall。

SimpleXMLRPCRequestHandler.**rpc_paths**
一个必须为元组类型的属性值，其中列出所接收 XML-RPC 请求的有效路径部分。发送到其他路径的请求将导致 404 "no such page" HTTP 错误。如果此元组为空，则所有路径都将被视为有效。默认值为 `('/', '/RPC2')`。

SimpleXMLRPCServer 示例

服务器端代码:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000),
```

(下页继续)

(繼續上一頁)

```
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name
    def adder_function(x, y):
        return x + y
    server.register_function(adder_function, 'add')

    # Register an instance; all the methods of the instance are
    # published as XML-RPC methods (in this case, just 'mul').
    class MyFuncs:
        def mul(self, x, y):
            return x * y

    server.register_instance(MyFuncs())

    # Run the server's main loop
    server.serve_forever()
```

以下客户端代码将调用上述服务器所提供的方法:

```
import xmlrpc.client

s = xmlrpc.client.ServerProxy('http://localhost:8000')
print(s.pow(2,3))  # Returns 2**3 = 8
print(s.add(2,3))  # Returns 5
print(s.mul(5,2))  # Returns 5*2 = 10

# Print list of available methods
print(s.system.listMethods())
```

`register_function()` 也可被用作装饰器。上述服务器端示例可以通过装饰器方式来注册函数:

```
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()

    # Register pow() function; this will use the value of
    # pow.__name__ as the name, which is just 'pow'.
    server.register_function(pow)

    # Register a function under a different name, using
    # register_function as a decorator. *name* can only be given
    # as a keyword argument.
    @server.register_function(name='add')
    def adder_function(x, y):
```

(下页继续)

(繼續上一頁)

```
        return x + y

    # Register a function under function.__name__.
    @server.register_function
    def mul(x, y):
        return x * y

    server.serve_forever()
```

以下包括在 Lib/xmlrpc/server.py 模块中的例子演示了一个允许带点号名称并注册有多调用函数的服务器。

警告: 启用 *allow_dotted_names* 选项将允许入侵者访问你的模块的全局变量并可能允许入侵者在你的机器上执行任意代码。仅可在安全、封闭的网络中使用此示例。

```
import datetime

class ExampleService:
    def getData(self):
        return '42'

    class currentTime:
        @staticmethod
        def getCurrentTime():
            return datetime.datetime.now()

with SimpleXMLRPCServer(("localhost", 8000)) as server:
    server.register_function(pow)
    server.register_function(lambda x,y: x+y, 'add')
    server.register_instance(ExampleService(), allow_dotted_names=True)
    server.register_multicall_functions()
    print('Serving XML-RPC on localhost port 8000')
    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nKeyboard interrupt received, exiting.")
        sys.exit(0)
```

这个 ExampleService 演示程序可通过命令行发起调用:

```
python -m xmlrpc.server
```

可与上述服务器进行交互的客户端包括在 *Lib/xmlrpc/client.py* 中:

```
server = ServerProxy("http://localhost:8000")

try:
    print(server.currentTime.getCurrentTime())
except Error as v:
    print("ERROR", v)

multi = MultiCall(server)
multi.getData()
multi.pow(2,9)
```

(下页继续)

(繼續上一頁)

```
multi.add(1,2)
try:
    for response in multi():
        print(response)
except Error as v:
    print("ERROR", v)
```

这个可与示例 XMLRPC 服务器进行交互的客户端的启动方式如下:

```
python -m xmlrpc.client
```

21.22.2 CGIXMLRPCRequestHandler

CGIXMLRPCRequestHandler 类可被用来处理发送给 Python CGI 脚本的 XML-RPC 请求。

CGIXMLRPCRequestHandler.**register_function**(*function=None*, *name=None*)
注册一个可以响应 XML-RPC 请求的函数。如果给出了 *name*，则它将成为与 *function* 相关联的方法名，否则将使用 `function.__name__`。*name* 是一个字符串，并可能包含不可在 Python 标识符中出现的字符，包括句点符。

此方法也可被用作装饰器。当被用作装饰器时，*name* 只能被指定为以 *name* 注册的 *function* 的一个关键字参数。如果没有指定 *name*，则将使用 `function.__name__`。

3.7 版更變: *register_function()* 可被用作装饰器。

CGIXMLRPCRequestHandler.**register_instance**(*instance*)
注册一个对象用来公开未使用*register_function()* 进行注册的方法名。如果实例包含 `_dispatch()` 方法，它会附带所请求的方法名和来自请求的形参被调用；返回值会作为结果被返回给客户端。如果实例不包含 `_dispatch()` 方法，则在其中搜索与所请求方法名相匹配的属性；如果所请求方法名包含句点，则会分别搜索方法名的每个部分，其效果就是执行了简单的层级搜索。搜索找到的值将附带来自请求的形参被调用，其返回值会被返回给客户端。

CGIXMLRPCRequestHandler.**register_introspection_functions**()
注册 XML-RPC 内海函数 `system.listMethods`, `system.methodHelp` 和 `system.methodSignature`。

CGIXMLRPCRequestHandler.**register_multicall_functions**()
注册 XML-RPC 多调用函数 `system.multicall`。

CGIXMLRPCRequestHandler.**handle_request**(*request_text=None*)
处理一个 XML-RPC 请求。如果给出了 *request_text*，它应当是 HTTP 服务器所提供的 POST 数据，否则将使用 stdin 的内容。

示例:

```
class MyFuncs:
    def mul(self, x, y):
        return x * y


handler = CGIXMLRPCRequestHandler()
handler.register_function(pow)
handler.register_function(lambda x,y: x+y, 'add')
handler.register_introspection_functions()
handler.register_instance(MyFuncs())
handler.handle_request()
```

21.22.3 XMLRPC 服务器文档

这些类扩展了上面的类以发布响应 HTTP GET 请求的 HTML 文档。服务器可以是独立的，使用*DocXMLRPCServer*，或是嵌入某个 CGI 环境中，使用*DocCGIXMLRPCRequestHandler*。

class xmlrpc.server.**DocXMLRPCServer**(*addr*, *requestHandler=DocXMLRPCRequestHandler*, *logRequests=True*, *allow_none=False*, *encoding=None*, *bind_and_activate=True*, *use_builtin_types=True*)

创建一个新的服务器实例。所有形参的含义与*SimpleXMLRPCServer* 的相同；*requestHandler* 默认为*DocXMLRPCRequestHandler*。

3.3 版更變: 增加了 *use_builtin_types* 旗标。

class xmlrpc.server.**DocCGIXMLRPCRequestHandler**

创建一个新的实例来处理 CGI 环境中的 XML-RPC 请求。

class xmlrpc.server.**DocXMLRPCRequestHandler**

创建一个新的请求处理句柄实例。该请求处理句柄支持 XML-RPC POST 请求、文档 GET 请求并会修改日志记录操作以便使用传递给*DocXMLRPCServer* 构造器形参的 *logRequests* 形参。

21.22.4 DocXMLRPCServer 对象

DocXMLRPCServer 类派生自*SimpleXMLRPCServer* 并提供了一种创建自动记录文档的、独立的 XML-RPC 服务器的方式。HTTP POST 请求将作为 XML-RPC 方法调用来处理。HTTP GET 请求将通过生成 pydoc 风格的 HTML 文档来处理。这将允许服务器自己提供基于 Web 的文档。

DocXMLRPCServer.**set_server_title**(*server_title*)

设置所生成 HTML 文档要使用的标题。此标题将在 HTML ”title” 元素中使用。

DocXMLRPCServer.**set_server_name**(*server_name*)

设置所生成 HTML 文档要使用的名称。此名称将出现在所生成文档顶部的”h1” 元素中。

DocXMLRPCServer.**set_server_documentation**(*server_documentation*)

设置所生成 Set the description used in the generated HTML 文档要使用的描述。此描述将显示为文档中的一个段落，位于服务器名称之下。

21.22.5 DocCGIXMLRPCRequestHandler

DocCGIXMLRPCRequestHandler 类派生自*CGIXMLRPCRequestHandler* 并提供了一种创建自动记录文档的 XML-RPC CGI 脚本的方式。HTTP POST 请求将作为 XML-RPC 方法调用来处理。HTTP GET 请求将通过生成 pydoc 风格的 HTML 文档来处理。这将允许服务器自己提供基于 Web 的文档。

DocCGIXMLRPCRequestHandler.**set_server_title**(*server_title*)

设置所生成 HTML 文档要使用的标题。此标题将在 HTML ”title” 元素中使用。

DocCGIXMLRPCRequestHandler.**set_server_name**(*server_name*)

设置所生成 HTML 文档要使用的名称。此名称将出现在所生成文档顶部的”h1” 元素中。

DocCGIXMLRPCRequestHandler.**set_server_documentation**(*server_documentation*)

设置所生成 Set the description used in the generated HTML 文档要使用的描述。此描述将显示为文档中的一个段落，位于服务器名称之下。

21.23 ipaddress --- IPv4/IPv6 操作库

源代码： Lib/ipaddress.py

---

ipaddress 提供了创建、处理和操作 IPv4 和 IPv6 地址和网络的功能。

该模块中的函数和类可以直接处理与 IP 地址相关的各种任务，包括检查两个主机是否在同一个子网中，遍历某个子网中的所有主机，检查一个字符串是否是一个有效的 IP 地址或网络定义等等。

这是完整的模块 API 参考—若要查看概述，请见 ipaddress-howto.

3.3 版新加入.

21.23.1 方便的工厂函数

ipaddress 模块提供来工厂函数来方便地创建 IP 地址，网络和接口：

ipaddress.**ip_address**(*address*)
Return an *IPv4Address* or *IPv6Address* object depending on the IP address passed as argument. Either IPv4 or IPv6 addresses may be supplied; integers less than `2**32` will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

```
>>> ipaddress.ip_address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.ip_address('2001:db8::')
IPv6Address('2001:db8::')
```

ipaddress.**ip_network**(*address*, *strict=True*)
Return an *IPv4Network* or *IPv6Network* object depending on the IP address passed as argument. *address* is a string or integer representing the IP network. Either IPv4 or IPv6 networks may be supplied; integers less than `2**32` will be considered to be IPv4 by default. *strict* is passed to *IPv4Network* or *IPv6Network* constructor. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address, or if the network has host bits set.

```
>>> ipaddress.ip_network('192.168.0.0/28')
IPv4Network('192.168.0.0/28')
```

ipaddress.**ip_interface**(*address*)
Return an *IPv4Interface* or *IPv6Interface* object depending on the IP address passed as argument. *address* is a string or integer representing the IP address. Either IPv4 or IPv6 addresses may be supplied; integers less than `2**32` will be considered to be IPv4 by default. A *ValueError* is raised if *address* does not represent a valid IPv4 or IPv6 address.

这些方便的函数的一个缺点是需要同时处理 IPv4 和 IPv6 格式，这意味着提供的错误信息并不精准，因为函数不知道是打算采用 IPv4 还是 IPv6 格式。更详细的错误报告可以通过直接调用相应版本的类构造函数来获得。

21.23.2 IP 地址

地址对象

IPv4Address 和*IPv6Address* 对象有很多共同的属性。一些只对 IPv6 地址有意义的属性也在*IPv4Address* 对象实现，以便更容易编写正确处理两种 IP 版本的代码。地址对象是可哈希的*hashable*，所以它们可以作为字典中的键来使用。

class ipaddress.**IPv4Address**(*address*)

构造一个 IPv4 地址。如果 *address* 不是一个有效的 IPv4 地址，会抛出*AddressValueError* 。

以下是有效的 IPv4 地址：

1. 以十进制小数点表示的字符串，由四个十进制整数组成，范围为 0--255，用点隔开 (例如 `192.168.0.1`)。每个整数代表地址中的八位（一个字节）。不允许使用前导零，以免与八进制表示产生歧义。
2. 一个 32 位可容纳的整数。
3. 一个长度为 4 的封装在*bytes* 对象中的整数 (高位优先)。

```
>>> ipaddress.IPv4Address('192.168.0.1')
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(3232235521)
IPv4Address('192.168.0.1')
>>> ipaddress.IPv4Address(b'\xC0\xA8\x00\x01')
IPv4Address('192.168.0.1')
```

3.8 版更變: 前导零可被接受，即使是在可能与八进制表示混淆的情况下也会被接受。

3.10 版更變: 前导零不再被接受，并且会被视作错误。IPv4 地址字符串现在严格按照 glibc 的*inet_pton()* 函数进行解析。

3.9.5 版更變: 上述变更也在自 3.9.5 版本起的 Python 3.9 当中被包含。

3.8.12 版更變: The above change was also included in Python 3.8 starting with version 3.8.12.

version

合适的版本号：IPv4 为 `4` ，IPv6 为 `6` 。

max_prefixlen

在该版本的地址表示中，比特数的总数：IPv4 为 `32` ；IPv6 为 `128` 。

前缀定义了地址中的前导位数量，通过比较来确定一个地址是否是网络的一部分。

compressed

exploded

以点符号分割十进制表示的字符串。表示中不包括前导零。

由于 IPv4 没有为八位数设为零的地址定义速记符号，这两个属性始终与 IPv4 地址的 “str(addr)“相同。暴露这些属性使得编写能够处理 IPv4 和 IPv6 地址的显示代码变得更加容易。

packed

这个地址的二进制表示——一个适当长度的*bytes* 对象（最高的八位在最前）。对于 IPv4 来说是 4 字节，对于 IPv6 来说是 16 字节。

reverse_pointer

IP 地址的反向 DNS PTR 记录的名称，例如:

```
>>> ipaddress.ip_address("127.0.0.1").reverse_pointer
'1.0.0.127.in-addr.arpa'
>>> ipaddress.ip_address("2001:db8::1").reverse_pointer
'1.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.8.b.d.0.1.0.0.2.ip6.arpa'
```

这是可用于执行 PTR 查询的名称，而不是已解析的主机名本身。

3.5 版新加入.

is_multicast
如果该地址被保留用作多播用途，返回 `True` 。关于多播地址，请参见 **RFC 3171** （IPv4）和 **RFC 2373** （IPv6）。

is_private
如果该地址被分配至私有网络，返回 `True` 。关于公共网络，请参见 iana-ipv4-special-registry （针对 IPv4）和 iana-ipv6-special-registry （针对 IPv6）。

is_global
如果该地址被分配至公共网络，返回 `True` 。关于公共网络，请参见 iana-ipv4-special-registry （针对 IPv4）和 iana-ipv6-special-registry （针对 IPv6）。

3.4 版新加入.

is_unspecified
当地址未指定时为 “True“ 。参见 **RFC 5735** (IPv4) 或 **RFC 2373** (IPv6).

is_reserved
如果该地址属于互联网工程任务组（IETF）所规定的其他保留地址，返回 `True` 。

is_loopback
如果该地址为一个回环地址，返回 `True` 。关于回环地址，请见 **RFC 3330** （IPv4）和 **RFC 2373** （IPv6）。

is_link_local
如果该地址被保留用于本地链接则为 `True`。参见 **RFC 3927**。

`IPv4Address.__format__`(*fmt*)
返回一个 IP 地址的字符串表示，由一个明确的格式字符串控制。*fmt* 可以是以下之一: `'s'`，默认选项，相当于*`str()`*，`'b'` 用于零填充的二进制字符串，`'X'` 或者 `'x'` 用于大写或小写的十六进制表示，或者 `'n'` 相当于 `'b'` 用于 IPv4 地址和 `'x'` 用于 IPv6 地址。对于二进制和十六进制表示法，可以使用形式指定器 `'#'` 和分组选项 `'_'`。`__format__` 被 `format`、`str.format` 和 f 字符串使用。

```
>>> format(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> '{:#b}'.format(ipaddress.IPv4Address('192.168.0.1'))
'0b11000000101010000000000000000001'
>>> f'{ipaddress.IPv6Address("2001:db8::1000"):s}'
'2001:db8::1000'
>>> format(ipaddress.IPv6Address('2001:db8::1000'), '_X')
'2001_0DB8_0000_0000_0000_0000_0000_1000'
>>> '{:#_n}'.format(ipaddress.IPv6Address('2001:db8::1000'))
'0x2001_0db8_0000_0000_0000_0000_0000_1000'
```

3.9 版新加入.

class `ipaddress.`**IPv6Address**(*address*)
构造一个 IPv6 地址。如果 *address* 不是一个有效的 IPv6 地址，会抛出*`AddressValueError`* 。

以下是有效的 IPv6 地址:

1. 一个由八组四个 16 进制数字组成的字符串，每组展示为 16 位. 以冒号分隔. 这可以描述为 分解 (普通书写). 此字符可以被 压缩 (速记书写) . 详见:RFC:*4291* . 例如, `"0000:0000:0000:0000:0000:0abc:0007:0def"` 可以被精简为 `"::abc:7:def"`.

   可选择的是，该字符串也可以有一个范围区 ID，用后缀 “%scope_id“表示。如果存在，范围 ID 必须是非空的，并且不能包含 “%“。详见:RFC:*4007*。例如，`fe80::1234%1``可以识别节点第一条链路上的地址``fe80::1234`

2. 适合 128 位的整数.

3. 一个打包在长度为 16 字节的大端序*bytes* 对象中的整数。

```
>>> ipaddress.IPv6Address('2001:db8::1000')
IPv6Address('2001:db8::1000')
>>> ipaddress.IPv6Address('ff02::5678%1')
IPv6Address('ff02::5678%1')
```

compressed

地址表示的简短形式，省略了组中的前导零，完全由零组成的最长的组序列被折叠成一个空组。

这也是 “str(addr)“对 IPv6 地址返回的值。

exploded

地址的长形式表示，包括所有前导零和完全由零组成的组。

关于以下属性和方法，请参见*IPv4Address* 类的相应文档。

packed

reverse_pointer

version

max_prefixlen

is_multicast

is_private

is_global

is_unspecified

is_reserved

is_loopback

is_link_local
3.4 版新加入: is_global

is_site_local
如果地址被保留用于本地站点则为 `True`。请注意本地站点地址空间已被 **RFC 3879** 弃用。请使用*is_private* 来检测此地址是否位于 **RFC 4193** 所定义的本地唯一地址空间中。

ipv4_mapped
映射 IPv4 的地址 (起始为 `::FFFF/96`), 这一属性记录为嵌入 IPv4 地址. 其他的任何地址, 这一属性为 `None`.

scope_id
对于:RFC:*4007*“定义的作用域地址，此属性以字符串的形式确定地址所属的作用域的特定区域。当没有指定作用域时，该属性将是 *“None“*。

sixtofour
对于看起来是 6to4 的地址（以 “2002::/16“开头），如 **RFC 3056** 所定义的，此属性将返回嵌入的 IPv4 地址。对于任何其他地址，该属性将是 “None“。

teredo
对于看起来是:RFC:*4380*“定义的 *Teredo* 地址（以 *“2001::/32“*开头）的地址，此属性将返回嵌入式 *IP* 地址对 *“(server, client)“*。对于任何其他地址，该属性将是 “None“。

IPv6Address.**__format__**(*fmt*)
请参考 *IPv4Address* 中对应的方法文档。

3.9 版新加入.

转换字符串和整数

与网络模块互操作像套接字模块, 地址必须转换为字符串或整数. 这是使用 *str()* 和 *int()* 内置函数:

```
>>> str(ipaddress.IPv4Address('192.168.0.1'))
'192.168.0.1'
>>> int(ipaddress.IPv4Address('192.168.0.1'))
3232235521
>>> str(ipaddress.IPv6Address('::1'))
'::1'
>>> int(ipaddress.IPv6Address('::1'))
1
```

请注意，IPv6 范围内的地址被转换为没有范围区域 ID 的整数。

运算符

地址对象支持一些运算符。除非另有说明，运算符只能在兼容对象之间应用（即 IPv4 与 IPv4，IPv6 与 IPv6）。

比较运算符

地址对象可以用通常的一组比较运算符进行比较。具有不同范围区域 ID 的相同 IPv6 地址是不平等的。一些例子:

```
>>> IPv4Address('127.0.0.2') > IPv4Address('127.0.0.1')
True
>>> IPv4Address('127.0.0.2') == IPv4Address('127.0.0.1')
False
>>> IPv4Address('127.0.0.2') != IPv4Address('127.0.0.1')
True
>>> IPv6Address('fe80::1234') == IPv6Address('fe80::1234%1')
False
>>> IPv6Address('fe80::1234%1') != IPv6Address('fe80::1234%2')
True
```

算术运算符

整数可以被添加到地址对象或从地址对象中减去。一些例子:

```
>>> IPv4Address('127.0.0.2') + 3
IPv4Address('127.0.0.5')
>>> IPv4Address('127.0.0.2') - 3
IPv4Address('126.255.255.255')
>>> IPv4Address('255.255.255.255') + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ipaddress.AddressValueError: 4294967296 (>= 2**32) is not permitted as an IPv4 address
```

21.23.3 IP 网络的定义

IPv4Network 和*IPv6Network* 对象提供了一个定义和检查 IP 网络定义的机制。一个网络定义由一个 *掩码* 和一个 *网络地址* 组成，因此定义了一个 IP 地址的范围，当用掩码屏蔽（二进制 AND）时，等于网络地址。例如，一个带有掩码 “255.255.255.0“和网络地址 “192.168.1.0“的网络定义由包括 “192.168.1.0“到 “192.168.1.255“的 IP 地址组成。

前缀、网络掩码和主机掩码

有几种相等的方法来指定 IP 网络掩码。*前缀* `/<nbits>``是一个符号，表示在网络掩码中设置了多少个高阶位。一个 * 网络掩码 * 是一个设置了一定数量高阶位的 IP 地址。因此，前缀/24“等同于 IPv4 中的网络掩码 “255.255.255.0“或 IPv6 中的网络掩码 “ffff:ff00::。此外，* 主机掩码 * 是 * 网络掩码 * 的逻辑取反，有时被用来表示网络掩码（例如在 Cisco 访问控制列表中）。在 IPv4 中，相当于主机掩码``0.0.0.255``的是/24“ 。

网络对象

所有由地址对象实现的属性也由网络对象实现。此外，网络对象还实现了额外的属性。所有这些在*IPv4Network* 和*IPv6Network* 之间是共同的，所以为了避免重复，它们只在*IPv4Network* 中记录。网络对象是*hashable*，所以它们可以作为字典中的键使用。

class ipaddress.**IPv4Network**(*address*, *strict=True*)

构建一个 IPv4 网络定义。*address* 可以是以下之一：

1. 一个由 IP 地址和可选掩码组成的字符串，用斜线 (/) 分开。IP 地址是网络地址，掩码可以是一个单一的数字，这意味着它是一个 *前缀*，或者是一个 IPv4 地址的字符串表示。如果是后者，如果掩码以非零字段开始，则被解释为 *网络掩码*，如果以零字段开始，则被解释为 *主机掩码*，唯一的例外是全零的掩码被视为 *网络掩码*。如果没有提供掩码，它就被认为是 /32。

   例如，以下的 * 地址 * 描述是等同的：192.168.1.0/24，192.168.1.0/255.255.255.0``和``192.168.1.0/0.0.0.255

2. 一个可转化为 32 位的整数。这相当于一个单地址的网络，网络地址是 *address*，掩码是 “/32“。

3. 一个整数被打包成一个长度为 4 的大端序*bytes* 对象。其解释类似于一个整数 *address*。

4. 一个地址描述和一个网络掩码的双元组，其中地址描述是一个字符串，一个 32 位的整数，一个 4 字节的打包整数，或一个现有的 IPv4Address 对象；而网络掩码是一个代表前缀长度的整数 (例如 24) 或一个代表前缀掩码的字符串 (例如 255.255.255.0)。

如果 *address* 不是一个有效的 IPv4 地址则会引发*AddressValueError*。如果掩码不是有效的 IPv4 地址则会引发*NetmaskValueError*。

如果 *strict* 是 `True`，并且在提供的地址中设置了主机位，那么*ValueError* 将被触发。否则，主机位将被屏蔽掉，以确定适当的网络地址。

除非另有说明，如果参数的 IP 版本与 `self` 不兼容，所有接受其他网络/地址对象的网络方法都将引发*TypeError*。

3.5 版更變: 为 *address* 构造函数参数添加了双元组形式。

version

max_prefixlen
请参考*IPv4Address* 中的相应属性文档。

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local
如果这些属性对网络地址和广播地址都是真实的，那么它们对整个网络来说就是真实的。

network_address
网络的网络地址。网络地址和前缀长度一起唯一地定义了一个网络。

broadcast_address
网络的广播地址。发送到广播地址的数据包应该被网络上的每台主机所接收。

hostmask
主机掩码，作为一个*IPv4Address* 对象。

netmask
网络掩码，作为一个*IPv4Address* 对象。

with_prefixlen

compressed

exploded
网络的字符串表示，其中掩码为前缀符号。

`with_prefixlen` 和 `compressed` 总是与 `str(network)` 相同，`exploded` 使用分解形式的网络地址。

with_netmask
网络的字符串表示，掩码用 net mask 符号表示。

with_hostmask
网络的字符串表示，其中的掩码为主机掩码符号。

num_addresses
网络中的地址总数。

prefixlen
网络前缀的长度，以比特为单位。

hosts()
返回一个网络中可用主机的迭代器。可用的主机是属于该网络的所有 IP 地址，除了网络地址本身

和网络广播地址。对于掩码长度为 31 的网络，网络地址和网络广播地址也包括在结果中。掩码为 32 的网络将返回一个包含单一主机地址的列表。

```
>>> list(ip_network('192.0.2.0/29').hosts())
[IPv4Address('192.0.2.1'), IPv4Address('192.0.2.2'),
 IPv4Address('192.0.2.3'), IPv4Address('192.0.2.4'),
 IPv4Address('192.0.2.5'), IPv4Address('192.0.2.6')]
>>> list(ip_network('192.0.2.0/31').hosts())
[IPv4Address('192.0.2.0'), IPv4Address('192.0.2.1')]
>>> list(ip_network('192.0.2.1/32').hosts())
[IPv4Address('192.0.2.1')]
```

overlaps(*other*)
如果这个网络部分或全部包含在 *other* 中，或者 *other* 全部包含在这个网络中，则为 “True“。

address_exclude(*network*)
计算从这个网络中移除给定的 *network* 后产生的网络定义。返回一个网络对象的迭代器。如果 *network* 不完全包含在这个网络中则会引发*ValueError*。

```
>>> n1 = ip_network('192.0.2.0/28')
>>> n2 = ip_network('192.0.2.1/32')
>>> list(n1.address_exclude(n2))
[IPv4Network('192.0.2.8/29'), IPv4Network('192.0.2.4/30'),
 IPv4Network('192.0.2.2/31'), IPv4Network('192.0.2.0/32')]
```

subnets(*prefixlen_diff=1*, *new_prefix=None*)
根据参数值，加入的子网构成当前的网络定义。*prefixlen_diff* 是我们的前缀长度应该增加的数量。*new_prefix* 是所需的子网的新前缀；它必须大于我们的前缀。必须设置 *prefixlen_diff* 和 *new_prefix* 中的一个，且只有一个。返回一个网络对象的迭代器。

```
>>> list(ip_network('192.0.2.0/24').subnets())
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
>>> list(ip_network('192.0.2.0/24').subnets(prefixlen_diff=2))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=26))
[IPv4Network('192.0.2.0/26'), IPv4Network('192.0.2.64/26'),
 IPv4Network('192.0.2.128/26'), IPv4Network('192.0.2.192/26')]
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=23))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise ValueError('new prefix must be longer')
ValueError: new prefix must be longer
>>> list(ip_network('192.0.2.0/24').subnets(new_prefix=25))
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/25')]
```

supernet(*prefixlen_diff=1*, *new_prefix=None*)
包含这个网络定义的超级网，取决于参数值。*prefixlen_diff* 是我们的前缀长度应该减少的数量。*new_prefix* 是超级网的新前缀；它必须比我们的前缀小。必须设置 *prefixlen_diff* 和 *new_prefix* 中的一个，而且只有一个。返回一个单一的网络对象。

```
>>> ip_network('192.0.2.0/24').supernet()
IPv4Network('192.0.2.0/23')
>>> ip_network('192.0.2.0/24').supernet(prefixlen_diff=2)
IPv4Network('192.0.0.0/22')
>>> ip_network('192.0.2.0/24').supernet(new_prefix=20)
IPv4Network('192.0.0.0/20')
```

subnet_of(*other*)
如果这个网络是 *other* 的子网，则返回 “True“。

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> b.subnet_of(a)
True
```

3.7 版新加入.

supernet_of(*other*)
如果这个网络是 *other* 的超网，则返回 “True“。

```
>>> a = ip_network('192.168.1.0/24')
>>> b = ip_network('192.168.1.128/30')
>>> a.supernet_of(b)
True
```

3.7 版新加入.

compare_networks(*other*)
将这个网络与 *ohter* 网络进行比较。在这个比较中，只考虑网络地址；不考虑主机位。返回是 “-1“ 、0``或``1。

```
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.2/32'))
-1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.0/32'))
1
>>> ip_network('192.0.2.1/32').compare_networks(ip_network('192.0.2.1/32'))
0
```

3.7 版後已F用: 它使用与”<”、”==” 和”>” 相同的排序和比较算法。

class ipaddress.**IPv6Network**(*address*, *strict=True*)
构建一个 IPv6 网络定义。*address* 可以是以下之一：

1. 一个由 IP 地址和可选前缀长度组成的字符串，用斜线（/）分开。IP 地址是网络地址，前缀长度必须是一个数字，即 *prefix*。如果没有提供前缀长度，就认为是 “/128“。

   Note that currently expanded netmasks are not supported. That means 2001:db00::0/24 is a valid argument while 2001:db00::0/ffff:ff00:: is not.

2. 一个适合 128 位的整数。这相当于一个单地址网络，网络地址是 *address*，掩码是 “/128“。

3. 一个整数被打包成一个长度为 16 的大端序*bytes* 对象。其解释类似于一个整数的 *address*。

4. 一个地址描述和一个网络掩码的双元组，其中地址描述是一个字符串，一个 128 位的整数，一个 16 字节的打包整数，或者一个现有的 IPv6Address 对象；而网络掩码是一个代表前缀长度的整数。

如果 *address* 不是一个有效的 IPv6 地址则会引发*AddressValueError*。如果掩码对 IPv6 地址无效则会引发*NetmaskValueError*。

如果 *strict* 是 True，并且在提供的地址中设置了主机位，那么*ValueError* 将被触发。否则，主机位将被屏蔽掉，以确定适当的网络地址。

3.5 版更變: 为 *address* 构造函数参数添加了双元组形式。

version

max_prefixlen

is_multicast

is_private

is_unspecified

is_reserved

is_loopback

is_link_local

network_address

broadcast_address

hostmask

netmask

with_prefixlen

compressed

exploded

with_netmask

with_hostmask

num_addresses

prefixlen

hosts()
返回一个网络中可用主机的迭代器。可用的主机是属于该网络的所有 IP 地址，除了 Subnet-Router 任播的地址。对于掩码长度为 127 的网络，子网-路由器任播地址也包括在结果中。掩码为 128 的网络将返回一个包含单一主机地址的列表。

overlaps(*other*)

address_exclude(*network*)

subnets(*prefixlen_diff=1*, *new_prefix=None*)

supernet(*prefixlen_diff=1*, *new_prefix=None*)

subnet_of(*other*)

supernet_of(*other*)

compare_networks(*other*)
请参考*IPv4Network* 中的相应属性文档。

is_site_local
如果这些属性对网络地址和广播地址都是真的，那么对整个网络来说就是真的。

运算符

网络对象支持一些操作符。除非另有说明，操作符只能在兼容的对象之间应用（例如，IPv4 与 IPv4，IPv6 与 IPv6）。

逻辑操作符

网络对象可以用常规的逻辑运算符集进行比较。网络对象首先按网络地址排序，然后按网络掩码排序。

迭代

网络对象可以被迭代，以列出属于该网络的所有地址。对于迭代，所有主机都会被返回，包括不可用的主机（对于可用的主机，使用*hosts()* 方法）。一个例子:

```
>>> for addr in IPv4Network('192.0.2.0/28'):
...     addr
...
IPv4Address('192.0.2.0')
IPv4Address('192.0.2.1')
IPv4Address('192.0.2.2')
IPv4Address('192.0.2.3')
IPv4Address('192.0.2.4')
IPv4Address('192.0.2.5')
IPv4Address('192.0.2.6')
IPv4Address('192.0.2.7')
IPv4Address('192.0.2.8')
IPv4Address('192.0.2.9')
IPv4Address('192.0.2.10')
IPv4Address('192.0.2.11')
IPv4Address('192.0.2.12')
IPv4Address('192.0.2.13')
IPv4Address('192.0.2.14')
IPv4Address('192.0.2.15')
```

作为地址容器的网络

网络对象可以作为地址的容器。一些例子:

```
>>> IPv4Network('192.0.2.0/28')[0]
IPv4Address('192.0.2.0')
>>> IPv4Network('192.0.2.0/28')[15]
IPv4Address('192.0.2.15')
>>> IPv4Address('192.0.2.6') in IPv4Network('192.0.2.0/28')
True
>>> IPv4Address('192.0.3.6') in IPv4Network('192.0.2.0/28')
False
```

21.23.4 接口对象

接口对象是*hashable* 的，所以它们可以作为字典中的键使用。

class ipaddress.**IPv4Interface**(*address*)

构建一个 IPv4 接口。*address* 的含义与*IPv4Network* 构造器中的一样，不同之处在于任意主机地址总是会被接受。

IPv4Interface 是*IPv4Address* 的一个子类，所以它继承了该类的所有属性。此外，还有以下属性可用:

ip
地址（*IPv4Address*）没有网络信息。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.ip
IPv4Address('192.0.2.5')
```

network
该接口所属的网络（*IPv4Network*）。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.network
IPv4Network('192.0.2.0/24')
```

with_prefixlen
用前缀符号表示的接口与掩码的字符串。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_prefixlen
'192.0.2.5/24'
```

with_netmask
以网络为掩码的接口的字符串表示。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_netmask
'192.0.2.5/255.255.255.0'
```

with_hostmask
以主机为掩码的接口的字符串表示。

```
>>> interface = IPv4Interface('192.0.2.5/24')
>>> interface.with_hostmask
'192.0.2.5/0.0.0.255'
```

class ipaddress.**IPv6Interface**(*address*)
构建一个 IPv6 接口。*address* 的含义与*IPv6Network* 构造器中的一样，不同之处在于任意主机地址总是会被接受。

IPv6Interface 是*IPv6Address* 的一个子类，所以它继承了该类的所有属性。此外，还有以下属性可用：

ip

network

with_prefixlen

with_netmask

with_hostmask
请参考*IPv4Interface* 中的相应属性文档。

运算符

接口对象支持一些运算符。除非另有说明，运算符只能在兼容的对象之间应用（即 IPv4 与 IPv4，IPv6 与 IPv6）。

逻辑操作符

接口对象可以用通常的逻辑运算符集进行比较。

对于相等比较（==``和!=“），IP 地址和网络都必须是相同的对象才会相等。一个接口不会与任何地址或网络对象相等。

对于排序 (<、> 等)，规则是不同的。具有相同 IP 版本的接口和地址对象可以被比较，而地址对象总是在接口对象之前排序。两个接口对象首先通过它们的网络进行比较，如果它们是相同的，则通过它们的 IP 地址进行比较。

21.23.5 其他模块级别函数

该模块还提供以下模块级函数：

ipaddress.**v4_int_to_packed**(*address*)
以网络（大端序）顺序将一个地址表示为 4 个打包的字节。*address* 是一个 IPv4 IP 地址的整数表示。如果整数是负数或太大而不满足 IPv4 IP 地址要求，会触发一个*ValueError*。

```
>>> ipaddress.ip_address(3221225985)
IPv4Address('192.0.2.1')
>>> ipaddress.v4_int_to_packed(3221225985)
b'\xc0\x00\x02\x01'
```

ipaddress.**v6_int_to_packed**(*address*)
以网络（大端序）顺序将一个地址表示为 4 个打包的字节。*address* 是一个 IPv6 IP 地址的整数表示。如果整数是负数或太大而不满足 IPv6 IP 地址要求，会触发一个*ValueError*。

ipaddress.**summarize_address_range**(*first*, *last*)
给出第一个和最后一个 IP 地址，返回总结的网络范围的迭代器。*first* 是范围内的第一个*IPv4Address*或*IPv6Address*，*last* 是范围内的最后一个*IPv4Address*或*IPv6Address*。如果 *first* 或 *last* 不是 IP 地址或不是同一版本则会引发*TypeError*。如果 *last* 不大于 *first*，或者 *first* 的地址版本不是 4 或 6 则会引发*ValueError*。

```
>>> [ipaddr for ipaddr in ipaddress.summarize_address_range(
...     ipaddress.IPv4Address('192.0.2.0'),
...     ipaddress.IPv4Address('192.0.2.130'))]
[IPv4Network('192.0.2.0/25'), IPv4Network('192.0.2.128/31'), IPv4Network('192.0.2.
↪130/32')]
```

ipaddress.**collapse_addresses**(*addresses*)
返回一个*IPv4Network*或*IPv6Network*对象的迭代器。*addresses* 是一个*IPv4Network*或*IPv6Network*对象的迭代器。如果 *addresses* 包含混合版本的对象则会引发*TypeError*。

```
>>> [ipaddr for ipaddr in
... ipaddress.collapse_addresses([ipaddress.IPv4Network('192.0.2.0/25'),
... ipaddress.IPv4Network('192.0.2.128/25')])]
[IPv4Network('192.0.2.0/24')]
```

ipaddress.**get_mixed_type_key**(*obj*)
返回一个适合在网络和地址之间进行排序的键。地址和网络对象在默认情况下是不可排序的；它们在本质上是不同的，所以表达式:

```
IPv4Address('192.0.2.0') <= IPv4Network('192.0.2.0/24')
```

是没有意义的。然而，有些时候，你可能希望让*ipaddress*对这些进行排序。如果你需要这样做，你可以使用这个函数作为*sorted()*的 *key* 参数。

obj 是一个网络或地址对象。

21.23.6 自定义异常

为了支持来自类构造函数的更具体的错误报告，模块定义了以下异常：

exception ipaddress.**AddressValueError**(*ValueError*)
与地址有关的任何数值错误。

exception ipaddress.**NetmaskValueError**(*ValueError*)
与网络掩码有关的任何数值错误

CHAPTER 22

多媒体服务

本章描述的模块实现了主要用于多媒体应用的各种算法或接口。它们可在安装时自行决定。这是一个概述：

22.1 wave --- 读写 WAV 格式文件

源代码: Lib/wave.py

wave 模块提供了一个处理 WAV 声音格式的便利接口。它不支持压缩/解压，但是支持单声道/立体声。

wave 模块定义了以下函数和异常:

wave.**open**(*file*, *mode=None*)

如果 *file* 是一个字符串，打开对应文件名的文件。否则就把它作为文件类对象来处理。*mode* 可以为以下值：

'rb' 只读模式。

'wb' 只写模式。

注意不支持同时读写 WAV 文件。

mode 设为 'rb' 时返回一个 Wave_read 对象，而 *mode* 设为 'wb' 时返回一个 Wave_write 对象。如果省略 *mode* 并指定 *file* 来传入一个文件类对象，则 file.mode 会被用作 *mode* 的默认值。

如果操作的是文件对象，当使用 wave 对象的 close() 方法时，并不会真正关闭文件对象，这需要调用者负责来关闭文件对象。

open() 函数可以在 with 语句中使用。当 with 阻塞结束时，*Wave_read.close()* 或 *Wave_write.close()* 方法会被调用。

3.4 版更變: 添加了对不可搜索文件的支持。

exception wave.**Error**

当不符合 WAV 格式或无法操作时引发的错误。

22.1.1 Wave_read 对象

由*open()* 返回的 Wave_read 对象，有以下几种方法:

Wave_read.**close**()
　关闭*wave* 打开的数据流并使对象不可用。当对象销毁时会自动调用。

Wave_read.**getnchannels**()
　返回声道数量（1 为单声道，2 为立体声）

Wave_read.**getsampwidth**()
　返回采样字节长度。

Wave_read.**getframerate**()
　返回采样频率。

Wave_read.**getnframes**()
　返回音频总帧数。

Wave_read.**getcomptype**()
　返回压缩类型（只支持 'NONE' 类型）

Wave_read.**getcompname**()
　getcomptype() 的通俗版本。使用 'not compressed' 代替 'NONE'。

Wave_read.**getparams**()
　返回一个*namedtuple()* (nchannels, sampwidth, framerate, nframes, comptype, compname)，与 get*() 方法的输出相同。

Wave_read.**readframes**(*n*)
　读取并返回以*bytes* 对象表示的最多 *n* 帧音频。

Wave_read.**rewind**()
　设置当前文件指针位置。

后面两个方法是为了和*aifc* 保持兼容，实际不做任何事情。

Wave_read.**getmarkers**()
　返回 None。

Wave_read.**getmark**(*id*)
　引发错误异常。

以下两个方法都使用指针，具体实现由其底层决定。

Wave_read.**setpos**(*pos*)
　设置文件指针到指定位置。

Wave_read.**tell**()
　返回当前文件指针位置。

22.1.2 Wave_write 对象

对于可查找的输出流，wave 头将自动更新以反映实际写入的帧数。对于不可查找的流，当写入第一帧时 *nframes* 值必须准确。获取准确的 *nframes* 值可以通过调用*setnframes()* 或*setparams()* 并附带*close()* 被调用之前将要写入的帧数，然后使用*writeframesraw()* 来写入帧数据，或者通过调用*writeframes()* 并附带所有要写入的帧。在后一种情况下*writeframes()* 将计算数据中的帧数并在写入帧数据之前相应地设置 *nframes*。

由*open()* 返回的 Wave_write 对象，有以下几种方法:

3.4 版更變: 添加了对不可搜索文件的支持。

Wave_write.**close**()
确保 *nframes* 是正确的，并在文件被*wave* 打开时关闭它。此方法会在对象收集时被调用。如果输出流不可查找且 *nframes* 与实际写入的帧数不匹配时引发异常。

Wave_write.**setnchannels**(*n*)
设置声道数。

Wave_write.**setsampwidth**(*n*)
设置采样字节长度为 *n*。

Wave_write.**setframerate**(*n*)
设置采样频率为 *n*。

3.2 版更變: 对此方法的非整数输入会被舍入到最接近的整数。

Wave_write.**setnframes**(*n*)
设置总帧数为 *n*。如果与之后实际写入的帧数不一致此值将会被更改（如果输出流不可查找则此更改尝试将引发错误）。

Wave_write.**setcomptype**(*type*, *name*)
设置压缩格式。目前只支持 NONE 即无压缩格式。

Wave_write.**setparams**(*tuple*)
tuple 应该是 (nchannels, sampwidth, framerate, nframes, comptype, compname)，每项的值应可用于 set*() 方法。设置所有形参。

Wave_write.**tell**()
返回当前文件指针，其指针含义和*Wave_read.tell()* 以及*Wave_read.setpos()* 是一致的。

Wave_write.**writeframesraw**(*data*)
写入音频数据但不更新 *nframes*。

3.4 版更變: 现在可接受任意*bytes-like object*。

Wave_write.**writeframes**(*data*)
写入音频帧并确保 *nframes* 是正确的。如果输出流不可查找且在 *data* 被写入之后写入的总帧数与之前设定的 has been written does not match the previously set value for *nframes* 值不匹配将会引发错误。

3.4 版更變: 现在可接受任意*bytes-like object*。

注意在调用 writeframes() 或 writeframesraw() 之后再设置任何格式参数是无效的，而且任何这样的尝试将引发*wave.Error*。

22.2 colorsys --- 颜色系统间的转换

源代码： Lib/colorsys.py

---

colorsys 模块定义了计算机显示器所用的 RGB (Red Green Blue) 色彩空间与三种其他色彩坐标系统 YIQ, HLS (Hue Lightness Saturation) 和 HSV (Hue Saturation Value) 表示的颜色值之间的双向转换。所有这些色彩空间的坐标都使用浮点数值来表示。在 YIQ 空间中，Y 坐标取值为 0 和 1 之间，而 I 和 Q 坐标均可以为正数或负数。在所有其他空间中，坐标取值均为 0 和 1 之间。

也参考:

有关色彩空间的更多信息可访问 https://poynton.ca/ColorFAQ.html 和 https://www.cambridgeincolour.com/tutorials/color-spaces.htm。

colorsys 模块定义了如下函数:

colorsys.**rgb_to_yiq**(*r*, *g*, *b*)
把颜色从 RGB 值转为 YIQ 值。

colorsys.**yiq_to_rgb**(*y*, *i*, *q*)
把颜色从 YIQ 值转为 RGB 值。

colorsys.**rgb_to_hls**(*r*, *g*, *b*)
把颜色从 RGB 值转为 HLS 值。

colorsys.**hls_to_rgb**(*h*, *l*, *s*)
把颜色从 HLS 值转为 RGB 值。

colorsys.**rgb_to_hsv**(*r*, *g*, *b*)
把颜色从 RGB 值转为 HSV 值。

colorsys.**hsv_to_rgb**(*h*, *s*, *v*)
把颜色从 HSV 值转为 RGB 值。

示例:

```
>>> import colorsys
>>> colorsys.rgb_to_hsv(0.2, 0.4, 0.4)
(0.5, 0.5, 0.4)
>>> colorsys.hsv_to_rgb(0.5, 0.5, 0.4)
(0.2, 0.4, 0.4)
```

CHAPTER 23

國際化

本章中介绍的模块通过提供选择要在程序信息中使用的语言的机制或通过定制输出以匹配本地约定来帮助你编写不依赖于语言和区域设置的软件。

本章中描述的模块列表是:

23.1 gettext --- 多语种国际化服务

源代码: Lib/gettext.py

gettext 模块为 Python 模块和应用程序提供国际化 (Internationalization, I18N) 和本地化 (Localization, L10N) 服务。它同时支持 GNU **gettext** 消息编目 API 和更高级的、基于类的 API，后者可能更适合于 Python 文件。下方描述的接口允许用户使用一种自然语言编写模块和应用程序消息，并提供翻译后的消息编目，以便在不同的自然语言下运行。

同时还给出一些本地化 Python 模块及应用程序的小技巧。

23.1.1 GNU gettext API

模块*gettext* 定义了下列 API，这与 **gettext** API 类似。如果你使用该 API，将会对整个应用程序产生全局的影响。如果你的应用程序支持多语种，而语言选择取决于用户的语言环境设置，这通常正是你所想要的。而如果你正在本地化某个 Python 模块，或者你的应用程序需要在运行时切换语言，相反你或许想用基于类的 API。

gettext.**bindtextdomain**(*domain*, *localedir=None*)

将 *domain* 绑定到本地目录 *localedir*。更具体地来说，模块*gettext* 将使用路径 (在 Unix 系统中): *localedir*/*language*/LC_MESSAGES/*domain*.mo 查找二进制 .mo 文件，此处对应地查找 *language* 的位置是环境变量 LANGUAGE, LC_ALL, LC_MESSAGES 和 LANG 中。

如果遗漏了 *localedir* 或者设置为 None，那么将返回当前 *domain* 所绑定的值[1]

gettext.**bind_textdomain_codeset**(*domain*, *codeset=None*)
将 *domain* 绑定到 *codeset*，修改*lgettext()*，*ldgettext()*，*lngettext()* 和*ldngettext()* 函数返回的字节串的字符编码。如果省略了 *codeset*，则返回当前绑定的编码集。

Deprecated since version 3.8, will be removed in version 3.10.

gettext.**textdomain**(*domain=None*)
修改或查询当前的全局域。如果 *domain* 为 None，则返回当前的全局域，不为 None 则将全局域设置为 *domain*，并返回它。

gettext.**gettext**(*message*)
返回 *message* 的本地化翻译，依据包括当前的全局域、语言和语言环境目录。本函数在本地命名空间中通常有别名 _()（参考下面的示例）。

gettext.**dgettext**(*domain*, *message*)
与*gettext()* 类似，但在指定的 *domain* 中查找 message。

gettext.**ngettext**(*singular*, *plural*, *n*)
与*gettext()* 类似，但考虑了复数形式。如果找到了翻译，则将 *n* 代入复数公式，然后返回得出的消息（某些语言具有两种以上的复数形式）。如果未找到翻译，则 *n* 为 1 时返回 *singular*，为其他数时返回 *plural*。

复数公式取自编目头文件。它是 C 或 Python 表达式，有一个自变量 *n*，该表达式计算的是所需复数形式在编目中的索引号。关于在 .po 文件中使用的确切语法和各种语言的公式，请参阅 GNU gettext 文档 。

gettext.**dngettext**(*domain*, *singular*, *plural*, *n*)
与*ngettext()* 类似，但在指定的 *domain* 中查找 message。

gettext.**pgettext**(*context*, *message*)

gettext.**dpgettext**(*domain*, *context*, *message*)

gettext.**npgettext**(*context*, *singular*, *plural*, *n*)

gettext.**dnpgettext**(*domain*, *context*, *singular*, *plural*, *n*)
与前缀中没有 p 的相应函数类似（即*gettext()*，*dgettext()*，*ngettext()*，*dngettext()* ），但是仅翻译给定的 message *context*。

3.8 版新加入.

gettext.**lgettext**(*message*)

gettext.**ldgettext**(*domain*, *message*)

gettext.**lngettext**(*singular*, *plural*, *n*)

gettext.**ldngettext**(*domain*, *singular*, *plural*, *n*)
与前缀中没有 l 的相应函数等效（*gettext()*，*dgettext()*，*ngettext()* 和*dngettext()* ），但是如果没有用*bind_textdomain_codeset()* 显式设置其他编码，则返回的翻译将以首选系统编码来编码字节串。

> **警告：** 在 Python 3 中应避免使用这些函数，因为它们返回的是编码后的字节串。最好使用返回 Unicode 字符串的其他方法，因为大多数 Python 应用程序都希望将人类可读的文本作为字符串而不是字节来处理。此外，如果翻译后的字符串存在编码问题，则可能会意外出现与 Unicode 相关的异常。

[1] 不同系统的默认语言环境目录是不同的；比如在 RedHat Linux 上是 /usr/share/locale，在 Solaris 上是 /usr/lib/locale。*gettext* 模块不会支持这些基于不同系统的默认值；而它的默认值为 *sys.base_prefix*/share/locale（请参阅*sys.base_prefix*）。基于上述原因，最好每次都在应用程序的开头使用明确的绝对路径来调用*bindtextdomain()*。

Deprecated since version 3.8, will be removed in version 3.10.

注意，GNU **gettext** 还定义了 dcgettext() 方法，但它被认为不实用，因此目前没有实现它。

这是该 API 的典型用法示例:

```
import gettext
gettext.bindtextdomain('myapplication', '/path/to/my/language/directory')
gettext.textdomain('myapplication')
_ = gettext.gettext
# ...
print(_('This is a translatable string.'))
```

23.1.2 基于类的 API

与 GNU **gettext** API 相比，*gettext* 模块的基于类的 API 提供了更多的灵活性和更强的便利性。这是本地化 Python 应用程序和模块的推荐方法。gettext 定义了一个*GNUTranslations* 类，该类实现了 GNU .mo 格式文件的解析，并且具有用于返回字符串的方法。本类的实例也可以将自身作为函数 _() 安装到内建命名空间中。

gettext.**find**(*domain*, *localedir=None*, *languages=None*, *all=False*)

本函数实现了标准的 .mo 文件搜索算法。它接受一个 *domain*，它与*textdomain()* 接受的域相同。可选参数 *localedir* 与*bindtextdomain()* 中的相同。可选参数 *languages* 是多条字符串的列表，其中每条字符串都是一种语言代码。

如果没有传入 *localedir*，则使用默认的系统语言环境目录。[2] 如果没有传入 *languages*，则搜索以下环境变量：LANGUAGE、LC_ALL、LC_MESSAGES 和 LANG。从这些变量返回的第一个非空值将用作 *languages* 变量。环境变量应包含一个语言列表，由冒号分隔，该列表会被按冒号拆分，以产生所需的语言代码字符串列表。

find() 将扩展并规范化 language，然后遍历它们，搜索由这些组件构建的现有文件:

localedir/*language*/LC_MESSAGES/*domain*.mo

find() 返回找到类似的第一个文件名。如果找不到这样的文件，则返回 None。如果传入了 *all*，它将返回一个列表，包含所有文件名，并按它们在语言列表或环境变量中出现的顺序排列。

gettext.**translation**(*domain*, *localedir=None*, *languages=None*, *class_=None*, *fallback=False*, *codeset=None*)

根据 *domain*、*localedir* 和 *languages*，返回 *Translations 实例，首先应将前述参数传入*find()* 以获取关联的 .mo 文件路径的列表。名字与 .mo 文件名相同的实例将被缓存。如果传入 *class_*，它将是实际被实例化的类，否则实例化*GNUTranslations*。类的构造函数必须只接受一个文件对象 参数。如果传入 *codeset*，那么在*lgettext()* 和*lngettext()* 方法中，对翻译后的字符串进行编码的字符集将被改变。

如果找到多个文件，后找到的文件将用作先前文件的替补。为了设置替补，将使用*copy.copy()* 从缓存中克隆每个 translation 对象。实际的实例数据仍在缓存中共享。

如果 .mo 文件未找到，且 *fallback* 为 false（默认值），则本函数引发*OSError* 异常，如果 *fallback* 为 true，则返回一个*NullTranslations* 实例。

3.3 版更變: *IOError* 代替*OSError* 被引发。

Deprecated since version 3.8, will be removed in version 3.10: *codeset* 参数。

gettext.**install**(*domain*, *localedir=None*, *codeset=None*, *names=None*)

根据传入*translation()* 函数的 *domain*、*localedir* 和 *codeset*，在 Python 内建命名空间中安装 _() 函数。

[2] 参阅上方*bindtextdomain()* 的脚注。

names 参数的信息请参阅 translation 对象的*install()* 方法的描述。

如下所示，通常将字符串包括在 _() 函数的调用中，以标记应用程序中待翻译的字符串，就像这样:

```
print(_('This string will be translated.'))
```

为了方便，一般将 _() 函数安装在 Python 内建命名空间中，以便在应用程序的所有模块中轻松访问它。

Deprecated since version 3.8, will be removed in version 3.10: *codeset* 参数。

NullTranslations 类

translation 类实际实现的是，将原始源文件消息字符串转换为已翻译的消息字符串。所有 translation 类使用的基类为*NullTranslations*，它提供了基本的接口，可用于编写自己定制的 translation 类。以下是 NullTranslations 的方法:

class gettext.**NullTranslations**(*fp=None*)

接受一个可选参数文件对象 *fp*，该参数会被基类忽略。初始化由派生类设置的"protected"（受保护的）实例变量 *_info* 和 *_charset*，与 *_fallback* 类似，但它是通过*add_fallback()* 来设置的。如果 *fp* 不为 None，就会调用 self._parse(fp)。

_parse(*fp*)

在基类中没有操作，本方法接受文件对象 *fp*，从该文件读取数据，用来初始化消息编目。如果你手头的消息编目文件的格式不受支持，则应重写本方法来解析你的格式。

add_fallback(*fallback*)

添加 *fallback* 为当前 translation 对象的替补对象。如果 translation 对象无法为指定消息提供翻译，则应向替补查询。

gettext(*message*)

如果设置了替补，则转发 gettext() 给替补。否则返回 *message*。在派生类中被重写。

ngettext(*singular*, *plural*, *n*)

如果设置了替补，则转发 ngettext() 给替补。否则，*n* 为 1 时返回 *singular*，为其他时返回 *plural*。在派生类中被重写。

pgettext(*context*, *message*)

如果设置了替补，则转发*pgettext()* 给替补。否则返回已翻译的消息。在派生类中被重写。

3.8 版新加入.

npgettext(*context*, *singular*, *plural*, *n*)

如果设置了替补，则转发*npgettext()* 给替补。否则返回已翻译的消息。在派生类中被重写。

3.8 版新加入.

lgettext(*message*)

lngettext(*singular*, *plural*, *n*)

与*gettext()* 和*ngettext()* 等效，但是如果没有用*set_output_charset()* 显式设置编码，则返回的翻译将以首选系统编码来编码字节串。在派生类中被重写。

> **警告:** 应避免在 Python 3 中使用这些方法。请参阅*lgettext()* 函数的警告。

Deprecated since version 3.8, will be removed in version 3.10.

info()

返回"protected"（受保护的） _info 变量，它是一个字典，包含在消息编目文件中找到的元数据。

charset()
返回消息编目文件的编码。

output_charset()
返回由*lgettext()* 和*lngettext()* 翻译出的消息的编码。

Deprecated since version 3.8, will be removed in version 3.10.

set_output_charset(*charset*)
更改翻译出的消息的编码。

Deprecated since version 3.8, will be removed in version 3.10.

install(*names=None*)
本方法将*gettext()* 安装至内建命名空间，并绑定为 _。

如果传入 *names* 参数，该参数必须是一个序列，包含除 _() 外其他要安装在内建命名空间中的函数的名称。支持的名称有 'gettext'，'ngettext'，'pgettext'，'npgettext'，'lgettext' 和 'lngettext'。

注意，这仅仅是使 _() 函数在应用程序中生效的一种方法，尽管也是最方便的方法。由于它会影响整个应用程序全局，特别是内建命名空间，因此已经本地化的模块不应该安装 _()，而是应该用下列代码使 _() 在各自模块中生效:

```
import gettext
t = gettext.translation('mymodule', ...)
_ = t.gettext
```

这只将 _() 放在其模块的全局命名空间中，所以只影响其模块内的调用。

3.8 版更變: 添加了 'pgettext' 和 'npgettext'。

GNUTranslations 类

gettext 模块提供了一个派生自*NullTranslations* 的其他类: *GNUTranslations*。该类重写了 _parse()，同时能以大端序和小端序格式读取 GNU **gettext** 格式的 .mo 文件。

GNUTranslations 从翻译编目中解析出可选的元数据。GNU **gettext** 约定，将元数据包括在空字符串的翻译中。该元数据采用 **RFC 822** 样式的 key: value 键值对，且应该包含 Project-Id-Version 键。如果找到 Content-Type 键，那么将用 charset 属性去初始化"protected"（受保护的）_charset 实例变量，而该变量在未找到键的情况下默认为 None。如果指定了字符编码，那么从编目中读取的所有消息 ID 和消息字符串都将使用此编码转换为 Unicode，若未指定编码，则假定编码为 ASCII。

由于消息 ID 也是作为 Unicode 字符串读取的，因此所有 *gettext() 方法都假定消息 ID 为 Unicode 字符串，而不是字节串。

整个键/值对集合将被放入一个字典，并设置为"protected"（受保护的）_info 实例变量。

如果 .mo 文件的魔法值 (magic number) 无效，或遇到意外的主版本号，或在读取文件时发生其他问题，则实例化*GNUTranslations* 类会引发*OSError*。

class gettext.**GNUTranslations**
下列方法是根据基类实现重写的:

gettext(*message*)
在编目中查找 *message* ID，并以 Unicode 字符串形式返回相应的消息字符串。如果在编目中没有 *message* ID 条目，且配置了替补，则查找请求将被转发到替补的*gettext()* 方法。否则，返回 *message* ID。

ngettext(*singular*, *plural*, *n*)
查找消息 ID 的复数形式。*singular* 用作消息 ID，用于在编目中查找，同时 *n* 用于确定使用哪种复数形式。返回的消息字符串是 Unicode 字符串。

如果在编目中没有找到消息 ID，且配置了替补，则查找请求将被转发到替补的*ngettext()* 方法。否则，当 *n* 为 1 时返回 *singular*，其他情况返回 *plural*。

例如：

```
n = len(os.listdir('.'))
cat = GNUTranslations(somefile)
message = cat.ngettext(
    'There is %(num)d file in this directory',
    'There are %(num)d files in this directory',
    n) % {'num': n}
```

pgettext(*context*, *message*)
在编目中查找 *context* 和 *message* ID，并以 Unicode 字符串形式返回相应的消息字符串。如果在编目中没有 *message* ID 和 *context* 条目，且配置了替补，则查找请求将被转发到替补的*pgettext()* 方法。否则，返回 *message* ID。

3.8 版新加入.

npgettext(*context*, *singular*, *plural*, *n*)
查找消息 ID 的复数形式。*singular* 用作消息 ID，用于在编目中查找，同时 *n* 用于确定使用哪种复数形式。

如果在编目中没有找到 *context* 对应的消息 ID，且配置了替补，则查找请求将被转发到替补的*npgettext()* 方法。否则，当 *n* 为 1 时返回 *singular*，其他情况返回 *plural*。

3.8 版新加入.

lgettext(*message*)

lngettext(*singular*, *plural*, *n*)
与*gettext()* 和*ngettext()* 等效，但是如果没有用*set_output_charset()* 显式设置编码，则返回的翻译将以首选系统编码来编码字节串。

> **警告：** 应避免在 Python 3 中使用这些方法。请参阅*lgettext()* 函数的警告。

Deprecated since version 3.8, will be removed in version 3.10.

Solaris 消息编目支持

Solaris 操作系统定义了自己的二进制 `.mo` 文件格式，但由于找不到该格式的文档，因此目前不支持该格式。

编目构造器

GNOME 用的 *gettext* 模块是 James Henstridge 写的版本，但该版本的 API 略有不同。它文档中的用法是:

```
import gettext
cat = gettext.Catalog(domain, localedir)
_ = cat.gettext
print(_('hello world'))
```

为了与本模块的旧版本兼容，`Catalog()` 函数是上述*translation()* 函数的别名。

本模块与 Henstridge 的模块有一个区别：他的编目对象支持通过映射 API 进行访问，但是该特性似乎从未使用过，因此目前不支持该特性。

23.1.3 国际化 (I18N) 你的程序和模块

国际化 (I18N) 是指使程序可切换多种语言的操作。本地化 (L10N) 是指程序的适配能力，一旦程序被国际化，就能适配当地的语言和文化习惯。为了向 Python 程序提供不同语言的消息，需要执行以下步骤:

1. 准备程序或模块，将可翻译的字符串特别标记起来
2. 在已标记的文件上运行一套工具，用来生成原始消息编目
3. 创建消息编目的不同语言的翻译
4. 使用*gettext* 模块，以便正确翻译消息字符串

为了准备代码以达成 I18N，需要巡视文件中的所有字符串。所有要翻译的字符串都应包括在 `_('...')` 内，以此打上标记——也就是调用 `_()` 函数。例如:

```
filename = 'mylog.txt'
message = _('writing a log message')
with open(filename, 'w') as fp:
    fp.write(message)
```

在这个例子中，字符串 `'writing a log message'` 被标记为待翻译，而字符串 `'mylog.txt'` 和 `'w'` 没有被标记。

有一些工具可以将待翻译的字符串提取出来。原版的 GNU **gettext** 仅支持 C 或 C++ 源代码，但其扩展版 **xgettext** 可以扫描多种语言的代码（包括 Python），来找出标记为可翻译的字符串。Babel 是一个 Python 国际化库，其中包含一个 `pybabel` 脚本，用于提取并编译消息编目。François Pinard 写的称为 **xpot** 的程序也能完成类似的工作，可从他的 po-utils 软件包 中获取。

(Python 还包括了这些程序的纯 Python 版本，称为 **pygettext.py** 和 **msgfmt.py**，某些 Python 发行版已经安装了它们。**pygettext.py** 类似于 **xgettext**，但只能理解 Python 源代码，无法处理诸如 C 或 C++ 的其他编程语言。**pygettext.py** 支持的命令行界面类似于 **xgettext**，查看其详细用法请运行 `pygettext.py --help`。**msgfmt.py** 与 GNU **msgfmt** 是二进制兼容的。有了这两个程序，可以不需要 GNU **gettext** 包来国际化 Python 应用程序。)

xgettext、**pygettext** 或类似工具生成的 `.po` 文件就是消息编目。它们是结构化的人类可读文件，包含源代码中所有被标记的字符串，以及这些字符串的翻译的占位符。

然后把这些 .po 文件的副本交给各个人工译者，他们为所支持的每种自然语言编写翻译。译者以 < 语言名称>.po 文件的形式发送回翻译完的某个语言的版本，将该文件用 **msgfmt** 程序编译为机器可读的 .mo 二进制编目文件。*gettext* 模块使用 .mo 文件在运行时进行实际的翻译处理。

如何在代码中使用*gettext* 模块取决于国际化单个模块还是整个应用程序。接下来的两节将讨论每种情况。

本地化你的模块

如果要本地化模块，则切忌进行全局性的更改，如更改内建命名空间。不应使用 GNU **gettext** API，而应使用基于类的 API。

假设你的模块叫做"spam"，并且该模块的各种自然语言翻译 .mo 文件存放于 /usr/share/locale，为 GNU **gettext** 格式。以下内容应放在模块顶部:

```
import gettext
t = gettext.translation('spam', '/usr/share/locale')
_ = t.gettext
```

本地化你的应用程序

如果正在本地化应用程序，可以将 _() 函数全局安装到内建命名空间中，通常在应用程序的主文件中安装。这将使某个应用程序的所有文件都能使用 _('...')，而不必在每个文件中显式安装它。

最简单的情况，就只需将以下代码添加到应用程序的主程序文件中:

```
import gettext
gettext.install('myapplication')
```

如果需要设置语言环境目录，可以将其传递给*install()* 函数:

```
import gettext
gettext.install('myapplication', '/usr/share/locale')
```

即时更改语言

如果程序需要同时支持多种语言，则可能需要创建多个翻译实例，然后在它们之间进行显式切换，如下所示:

```
import gettext

lang1 = gettext.translation('myapplication', languages=['en'])
lang2 = gettext.translation('myapplication', languages=['fr'])
lang3 = gettext.translation('myapplication', languages=['de'])

# start by using language1
lang1.install()

# ... time goes by, user selects language 2
lang2.install()

# ... more time goes by, user selects language 3
lang3.install()
```

延迟翻译

在大多数代码中，字符串会在编写位置进行翻译。但偶尔需要将字符串标记为待翻译，实际翻译却推迟到后面。一个典型的例子是:

```
animals = ['mollusk',
           'albatross',
           'rat',
           'penguin',
           'python', ]
# ...
for a in animals:
    print(a)
```

此处希望将 animals 列表中的字符串标记为可翻译，但不希望在打印之前对它们进行翻译。

这是处理该情况的一种方式:

```
def _(message): return message

animals = [_('mollusk'),
           _('albatross'),
           _('rat'),
           _('penguin'),
           _('python'), ]

del _

# ...
for a in animals:
    print(_(a))
```

该方法之所以可行，是因为 _() 的虚定义只是简单地返回了原本的字符串。并且该虚定义将临时覆盖内建命名空间中 _() 的定义（直到 del 命令）。即使先前在本地命名空间中已经有了 _() 的定义也请注意。

注意，第二次使用 _() 不会认为"a" 可以传递给 **gettext** 程序去翻译，因为该参数不是字符串文字。

解决该问题的另一种方法是下面这个例子:

```
def N_(message): return message

animals = [N_('mollusk'),
           N_('albatross'),
           N_('rat'),
           N_('penguin'),
           N_('python'), ]

# ...
for a in animals:
    print(_(a))
```

这种情况下标记可翻译的字符串使用的是函数 N_()，该函数不会与 _() 的任何定义冲突。但是，需要教会消息提取程序去寻找用 N_() 标记的可翻译字符串。**xgettext**, **pygettext**, pybabel extract 和 **xpot** 都支持此功能，使用 -k 命令行开关即可。这里选择 N_() 为名称完全是任意的，它也能轻易改为 MarkThisStringForTranslation()。

23.1.4 致谢

以下人员为创建此模块贡献了代码、反馈、设计建议、早期实现和宝贵的经验：

- Peter Funk
- James Henstridge
- Juan David Ibáñez Palomar
- Marc-André Lemburg
- Martin von Löwis
- François Pinard
- Barry Warsaw
- Gustavo Niemeyer

🄵解

23.2 `locale` --- 国际化服务

源代码： Lib/locale.py

---

locale 模块开通了 POSIX 本地化数据库和功能的访问。POSIX 本地化机制让程序员能够为应用程序处理某些本地化的问题，而不需要去了解运行软件的每个国家的全部文化特点。

locale 模块是在 `_locale` 模块的基础上实现的，而 _locale 则又用到了 ANSI C 语言实现的本地化功能。

locale 模块定义了以下异常和函数：

exception `locale.`**Error**
: 当传给*setlocale()* 的 locale 无法识别时，会触发异常。

`locale.`**setlocale**(*category*, *locale=None*)
: 如果给定了 *locale* 而不是 `None`，*setlocale()* 会修改 *category* 的 locale 设置。可用的类别会在下面的数据描述中列出。*locale* 可以是一个字符串，也可以是两个字符串（语言代码和编码）组成的可迭代对象。若为可迭代对象，则会用地区别名引擎转换为一个地区名称。若为空字符串则指明采用用户的默认设置。如果 locale 设置修改失败，会触发*Error* 异常。如果成功则返回新的 locale 设置。

  如果省略 *locale* 或为 `None`，将返回 *category* 但当前设置。

  setlocale() 在大多数系统上都不是线程安全的。应用程序通常会如下调用：

  ```
  import locale
  locale.setlocale(locale.LC_ALL, '')
  ```

  这会把所有类别的 locale 都设为用户的默认设置（通常在 `LANG` 环境变量中指定）。如果后续 locale 没有改动，则使用多线程应该不会产生问题。

`locale.`**localeconv**()
: 以字典的形式返回本地约定的数据库。此字典具有以下字符串作为键：

类别	键	含义
LC_NUMERIC	'decimal_point'	小数点字符。
	'grouping'	数字列表，指定 'thousands_sep' 应该出现的位置。如果列表以*CHAR_MAX* 结束，则不会作分组。如果列表以 0 结束，则重复使用最后的分组大小。
	'thousands_sep'	组之间使用的字符。
LC_MONETARY	'int_curr_symbol'	国际货币符号。
	'currency_symbol'	当地货币符号。
	'p_cs_precedes/n_cs_precedes'	货币符号是否在值之前（对于正值或负值）。
	'p_sep_by_space/n_sep_by_space'	货币符号是否通过空格与值分隔（对于正值或负值）。
	'mon_decimal_point'	用于货币金额的小数点。
	'frac_digits'	货币值的本地格式中使用的小数位数。
	'int_frac_digits'	货币价值的国际格式中使用的小数位数。
	'mon_thousands_sep'	用于货币值的组分隔符。
	'mon_grouping'	相当于 'grouping'，用于货币价值。
	'positive_sign'	用于标注正货币价值的符号。
	'negative_sign'	用于注释负货币价值的符号。
	'p_sign_posn/n_sign_posn'	符号的位置（对于正值或负值），见下文。

可以将所有数值设置为*CHAR_MAX*，以指示此语言环境中未指定任何值。

下面给出了 'p_sign_posn' 和 'n_sign_posn' 的可能值。

值	解释
0	被括号括起来的货币和金额。
1	该标志应位于值和货币符号之前。
2	该标志应位于值和货币符号之后。
3	标志应该紧跟在值之前。
4	标志应该紧跟值项。
CHAR_MAX	此语言环境中未指定任何内容。

该函数将 LC_CTYPE 地区设为 LC_NUMERIC 地区，若地区不同且数字或货币字符为非 ASCII，则设置为 LC_MONETARY 地区。这个临时的改变会影响到其他线程。

3.7 版更變: 现在在某些情况下，该函数会将 LC_CTYPE 地区设为 LC_NUMERIC 地区。

locale.**nl_langinfo**(*option*)

以字符串形式返回一些地区相关的信息。本函数并非在所有系统都可用，而且可用的 option 在不同平台上也可能不同。可填的参数值为数值，在 locale 模块中提供了对应的符号常量。

nl_langinfo() 函数可接受以下值。大部分含义都取自 GNU C 库。

locale.**CODESET**

获取一个字符串，代表所选地区采用的字符编码名称。

locale.**D_T_FMT**

获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示日期和时间。

locale.**D_FMT**
获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示日期。

locale.**T_FMT**
获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示时间。

locale.**T_FMT_AMPM**
获取一个字符串，可用作*time.strftime()* 的格式串，以便以 am/pm 的格式表示时间。

DAY_1 ... DAY_7
获取一周中第 n 天的名称。

備🄵: 这里遵循美国惯例，即 DAY_1 是星期天，而不是国际惯例（ISO 8601），即星期一是一周的第一天。

ABDAY_1 ... ABDAY_7
获取一周中第 n 天的缩写名称。

MON_1 ... MON_12
获取第 n 个月的名称。

ABMON_1 ... ABMON_12
获取第 n 个月的缩写名称。

locale.**RADIXCHAR**
获取小数点字符（小数点、小数逗号等）。

locale.**THOUSEP**
获取千位数（三位数一组）的分隔符。

locale.**YESEXPR**
获取一个可供 regex 函数使用的正则表达式，用于识别需要回答是或否的问题的肯定回答。

備🄵: 该表达式的语法适用于 C 库的 regex() 函数，可能与*re* 中的语法不一样。

---

locale.**NOEXPR**
获取一个可供 regex(3) 函数使用的正则表达式，用于识别需要回答是或否的问题的否定回答。

locale.**CRNCYSTR**
获取货币符号，如果符号应位于数字之前，则在其前面加上“-”；如果符号应位于数字之后，则前面加“+”；如果符号应取代小数点字符，则前面加“.”。

locale.**ERA**
获取一个字符串，代表当前地区使用的纪元。

大多数地区都没有定义该值。定义了该值的一个案例日本。日本传统的日期表示方法中，包含了当时天皇统治朝代的名称。

通常没有必要直接使用该值。在格式串中指定 E 符号，会让*time.strftime()* 函数启用此信息。返回字符串的格式并没有定义，因此不得假定各个系统都能理解。

locale.**ERA_D_T_FMT**
获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示带纪元的日期和时间。

locale.**ERA_D_FMT**
获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示带纪元的日期。

locale.**ERA_T_FMT**
获取一个字符串，可用作*time.strftime()* 的格式串，以便以地区特定格式表示带纪元的时间。

locale.**ALT_DIGITS**
获取 0 到 99 的表示法，最多不超过 100 个值。

locale.**getdefaultlocale**([*envvars*])
尝试确定默认的地区设置，并以 (language code, encoding) 元组的形式返回。

根据 POSIX 的规范，未调用 setlocale(LC_ALL, '') 的程序采用可移植的 'C' 区域设置运行。调用 setlocale(LC_ALL, '') 则可采用 LANG 变量定义的默认区域。由于不想干扰当前的区域设置，因此就以上述方式进行了模拟。

为了维持与其他平台的兼容性，不仅需要检测 LANG 变量，还需要检测 envvars 参数给出的变量列表。首先发现的定义将被采用。*envvars* 默认为 GNU gettext 采用的搜索路径；必须包含 'LANG' 变量。GNU gettext 的搜索路径依次包含了 'LC_ALL'、'LC_CTYPE'、'LANG' 和 'LANGUAGE'。

除了 'C' 之外，语言代码对应 **RFC 1766** 标准。如果 语言代码和 编码不能确定，可为 None。

locale.**getlocale**(*category=LC_CTYPE*)
以列表的形式返回指定地区类别的当前设置，结果包括 语言代码、编码。*category* 可以是 LC_* 之一，*LC_ALL* 除外。默认为*LC_CTYPE*。

除了 'C' 之外，语言代码对应 **RFC 1766** 标准。如果 语言代码和 编码不能确定，可为 None。

locale.**getpreferredencoding**(*do_setlocale=True*)
Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

某些系统必须调用*setlocale()* 才能获取用户偏好，所以本函数不是线程安全的。如果不需要或不希望调用 setlocale，*do_setlocale* 应设为 False。

On Android or in the UTF-8 mode (-X utf8 option), always return 'UTF-8', the locale and the *do_setlocale* argument are ignored.

3.7 版更變: The function now always returns UTF-8 on Android or if the UTF-8 mode is enabled.

locale.**normalize**(*localename*)
为给定的区域名称返回标准代码。返回的区域代码已经格式化，可供*setlocale()* 使用。如果标准化操作失败，则返回原名称。

如果给出的编码无法识别，则本函数默认采用区域代码的默认编码，这正类似于*setlocale()*。

locale.**resetlocale**(*category=LC_ALL*)
将 *category* 的区域设为默认值。

默认设置通过调用*getdefaultlocale()* 确定。*category* 默认为*LC_ALL*。

locale.**strcoll**(*string1*, *string2*)
根据当前的*LC_COLLATE* 设置，对两个字符串进行比较。与其他比较函数一样，根据 *string1* 位于 *string2* 之前、之后或是相同，返回负值、、正值或者 0。

locale.**strxfrm**(*string*)
将字符串转换为可用于本地化比较的字符串。例如 strxfrm(s1) < strxfrm(s2) 相当于 strcoll(s1, s2) < 0。在重复比较同一个字符串时，可能会用到本函数，比如整理字符串列表时。

locale.**format_string**(*format*, *val*, *grouping=False*, *monetary=False*)
根据当前的*LC_NUMERIC* 设置，对数字 *val* 进行格式化。格式将遵循 % 运算符的约定。浮点值的小数点会按需修改。若 *grouping* 为 True，则还会考虑分组。

若 *monetary* 为 True，则会用到货币千位分隔符和分组字符串。

格式化符的处理类似 `format % val`，但会考虑到当前的区域设置。

3.7 版更變: 增加了关键字参数 *monetary* 。

`locale.`**`format`**(*format*, *val*, *grouping=False*, *monetary=False*)
请注意，本函数的工作原理与*format_string()* 类似，但只对 `%char` 格式符起作用。比如 `'%f'` 和 `'%.0f'` 都是有效的格式符，但 `'%f KiB'` 则不是。

若想用到全部的格式化串，请采用*format_string()*。

3.7 版後已🄴用: 请改用*format_string()* 。

`locale.`**`currency`**(*val*, *symbol=True*, *grouping=False*, *international=False*)
根据当前的*LC_MONETARY* 设置，对数字 *val* 进行格式化。

如果 *symbol* 为 True（默认值），则返回的字符串将包含货币符号。如果 *grouping* 为 True（非默认值），则会进行分组。如果 *international* 为 True（非默认值），将采用国际货币符号。

请注意，本函数对区域“C”无效，所以必须先通过*setlocale()* 设置一个区域。

`locale.`**`str`**(*float*)
对浮点数进行格式化，格式要求与内置函数 `str(float)` 相同，但会考虑小数点。

`locale.`**`delocalize`**(*string*)
根据*LC_NUMERIC* 的设置，将字符串转换为格式化后的数字字符串。

3.5 版新加入.

`locale.`**`atof`**(*string*, *func=float*)
Converts a string to a number, following the *LC_NUMERIC* settings, by calling *func* on the result of calling *delocalize()* on *string*.

`locale.`**`atoi`**(*string*)
按照*LC_NUMERIC* 的约定，将字符串转换为整数。

`locale.`**`LC_CTYPE`**
字符型函数的区域类别。根据该类别的设置，模块*string* 中处理大小写的函数会改变操作方式。

`locale.`**`LC_COLLATE`**
字符串排序会用到的区域类别。将会影响*locale* 模块的*strcoll()* 和*strxfrm()* 函数。

`locale.`**`LC_TIME`**
格式化时间时会用到的区域类别。*time.strftime()* 函数会参考这些约定。

`locale.`**`LC_MONETARY`**
格式化货币值时会用到的区域类别。可用值可由*localeconv()* 函数获取。

`locale.`**`LC_MESSAGES`**
显示消息时用到的区域类别。目前 Python 不支持应用定制的本地化消息。由操作系统显示的消息，比如由*os.strerror()* 返回的消息可能会受到该类别的影响。

`locale.`**`LC_NUMERIC`**
格式化数字时用到的区域类别。*locale* 模块的*format()* 、*atoi()* 、*atof()* 和*str()* 函数会受到该类别的影响。其他所有数字格式化操作不受影响。

`locale.`**`LC_ALL`**
混合所有的区域设置。如果在区域改动时使用该标志，将尝试设置所有类别的区域参数。只要有任何一个类别设置失败，就不会修改任何类别。在使用此标志获取区域设置时，会返回一个代表所有类别设置的字符串。之后可用此字符串恢复设置。

`locale.`**`CHAR_MAX`**
一个符号常量，*localeconv()* 返回多个值时将会用到。

示例:

```
>>> import locale
>>> loc = locale.getlocale()  # get current locale
# use German locale; name might vary with platform
>>> locale.setlocale(locale.LC_ALL, 'de_DE')
>>> locale.strcoll('f\xe4n', 'foo')  # compare a string containing an umlaut
>>> locale.setlocale(locale.LC_ALL, '')    # use user's preferred locale
>>> locale.setlocale(locale.LC_ALL, 'C')   # use default (C) locale
>>> locale.setlocale(locale.LC_ALL, loc)   # restore saved locale
```

23.2.1 背景、细节、提示、技巧和注意事项

The C standard defines the locale as a program-wide property that may be relatively expensive to change. On top of that, some implementation are broken in such a way that frequent locale changes may cause core dumps. This makes the locale somewhat painful to use correctly.

当程序第一次启动时，无论用户偏好定义成什么，区域值都是 C。不过有一个例外，就是在启动时修改*LC_CTYPE* 类别，设置当前区域编码为用户偏好编码。程序必须调用 `setlocale(LC_ALL, '')` 明确表示用户偏好区域将设为其他类别。

若要从库程序中调用*setlocale()*，通常这不是个好主意，因为副作用是会影响整个程序。保存和恢复区域设置也几乎一样糟糕：不仅代价高昂，而且会影响到恢复之前运行的其他线程。

如果是要编写通用模块，需要有一种不受区域设置影响的操作方式（比如某些用到*time.strftime()* 的格式），将不得不寻找一种不用标准库的方案。更好的办法是说服自己，可以采纳区域设置。只有在万不得已的情况下，才能用文档标注出模块与非 C 区域设置不兼容。

根据区域设置进行数字运算的唯一方法，就是采用本模块定义的函数：*atof()* 、*atoi()* 、*format()* 、*str()*。

无法根据区域设置进行大小写转换和字符分类。对于（Unicode）文本字符串来说，这些操作都是根据字符值进行的；而对于字节符串来说，转换和分类则是根据字节的 ASCII 值进行的，高位被置位的字节（即非 ASCII 字节）永远不会被转换或被视作字母或空白符之类。

23.2.2 针对扩展程序编写人员和嵌入 Python 运行的程序

除了要查询当前区域，扩展模块不应去调用*setlocale()*。但由于返回值只能用于恢复设置，所以也没什么用（也许只能用于确认是否为 C）。

当 Python 代码利用*locale* 模块修改区域设置时，也会影响到嵌入 Python 运行的应用程序。如果嵌入运行的程序不希望发生这种情况，则应从 `config.c` 文件的内置模块表中删除 `_locale` 扩展模块（所有操作均是由它完成的），并确保 `_locale` 模块不能成为一个共享库。

23.2.3 访问信息目录

`locale.`**`gettext`**(*msg*)

`locale.`**`dgettext`**(*domain*, *msg*)

`locale.`**`dcgettext`**(*domain*, *msg*, *category*)

`locale.`**`textdomain`**(*domain*)

`locale.`**`bindtextdomain`**(*domain*, *dir*)

在提供 gettext 接口的系统中，locale 模块暴露出了 C 库的接口。它由 `gettext()` 、`dgettext()` 、`dcgettext()` 、`textdomain()` 、`bindtextdomain()` 和 `bind_textdomain_codeset()` 等函数组成。他们与*gettext* 模块中的同名函数类似，但采用了 C 库二进制格式的消息目录，以及 C 库的搜索算法来查找消息目录。

Python 应用程序通常不需要调用这些函数，而应改用*gettext*。这条规则的一个已知例外，是与附加 C 库链接的应用程序，他们在内部调用了 `gettext()` 或 `dcgettext()`。对于这些应用程序，可能有必要绑定文本域，以便库可以准确找到他们的信息目录。

CHAPTER 24

程式框架

本章中描述的模块是很大程度上决定程序结构的框架。目前，这里描述的模块都面向编写命令行接口。

本章描述的完整模块列表如下：

24.1 `turtle` --- 龜圖學

源码： Lib/turtle.py

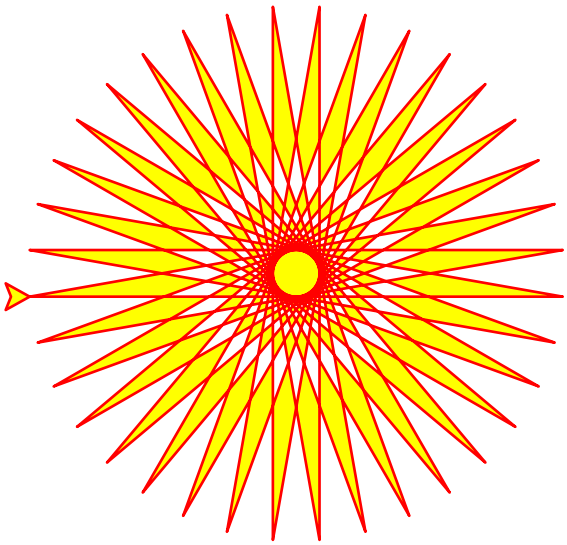
24.1.1 介紹

海龟绘图很适合用来引导孩子学习编程。最初来自于 Wally Feurzeig, Seymour Papert 和 Cynthia Solomon 于 1967 年所创造的 Logo 编程语言。

想像一下，一🄵機器龜在 x-y 平面上從 (0, 0) 出發。在 `import turtle` 之後，給它命令 `turtle.forward(15)`，然後它就會移動 (在螢幕上！) 15 個單位像素，方向是朝著其正面對的方向。給它命令 `turtle.right(25)`，它就會在原地順時針旋轉 25 度。

Turtle star

龜可以使用重🄵簡單動作之程式來畫出🄵雜的形狀。



```
from turtle import *
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

藉由結合這些類似的命令，F雜的形狀和圖形可以輕易被畫出來。

turtle 模块是基于 Python 标准发行版 2.5 以来的同名模块重新编写并进行了功能扩展。

新模块尽量保持了原模块的特点，并且 (几乎)100% 与其兼容。这就意味着初学编程者能够以交互方式使用模块的所有命令、类和方法——运行 IDLE 时注意加 `-n` 参数。

turtle 模块提供面向对象和面向过程两种形式的海龟绘图基本组件。由于它使用*tkinter* 实现基本图形界面，因此需要安装了 Tk 支持的 Python 版本。

面向对象的接口主要使用“2+2”个类：

1. *TurtleScreen* 类定义图形窗口作为绘图海龟的运动场。它的构造器需要一个 `tkinter.Canvas` 或*ScrolledCanvas* 作为参数。应在*turtle* 作为某个程序的一部分的时候使用。

   Screen() 函数返回一个*TurtleScreen* 子类的单例对象。此函数应在*turtle* 作为独立绘图工具时使用。作为一个单例对象，其所属的类是不可被继承的。

   TurtleScreen/Screen 的所有方法还存在对应的函数，即作为面向过程的接口组成部分。

2. *RawTurtle* (别名: *RawPen*) 类定义海龟对象在*TurtleScreen* 上绘图。它的构造器需要一个 Canvas, ScrolledCanvas 或 TurtleScreen 作为参数，以指定 RawTurtle 对象在哪里绘图。

   从 RawTurtle 派生出子类*Turtle* (别名: `Pen`)，该类对象在*Screen* 实例上绘图，如果实例不存在则会自动创建。

   RawTurtle/Turtle 的所有方法也存在对应的函数，即作为面向过程的接口组成部分。

过程式接口提供与*Screen*和*Turtle*类的方法相对应的函数。函数名与对应的方法名相同。当 Screen 类的方法对应函数被调用时会自动创建一个 Screen 对象。当 Turtle 类的方法对应函数被调用时会自动创建一个(匿名的) Turtle 对象。

如果屏幕上需要有多个海龟，就必须使用面向对象的接口。

備🄴: 以下文档给出了函数的参数列表。对于方法来说当然还有额外的第一个参数 *self*，这里省略了。

24.1.2 可用的 Turtle 和 Screen 方法概览

Turtle 方法

海龟动作

移动和绘制

forward() | *fd()* 前进
backward() | *bk()* | *back()* 后退
right() | *rt()* 右转
left() | *lt()* 左转
goto() | *setpos()* | *setposition()* 前往/定位
setx() 设置 x 坐标
sety() 设置 y 坐标
setheading() | *seth()* 设置朝向
home() 返回原点
circle() 画圆
dot() 画点
stamp() 印章
clearstamp() 清除印章
clearstamps() 清除多个印章
undo() 撤消
speed() 速度

获取海龟的状态

position() | *pos()* 位置
towards() 目标方向
xcor() x 坐标
ycor() y 坐标
heading() 朝向
distance() 距离

设置与度量单位

degrees() 角度
radians() 弧度

画笔控制

绘图状态

pendown() | *pd()* | *down()* 画笔落下
penup() | *pu()* | *up()* 画笔抬起

pensize() | *width()* 画笔粗细
pen() 画笔
isdown() 画笔是否落下

颜色控制

color() 颜色
pencolor() 画笔颜色
fillcolor() 填充颜色

填充

filling() 是否填充
begin_fill() 开始填充
end_fill() 结束填充

更多绘图控制

reset() 重置
clear() 清空
write() 书写

海龟状态

可见性

showturtle() | *st()* 显示海龟
hideturtle() | *ht()* 隐藏海龟
isvisible() 是否可见

外观

shape() 形状
resizemode() 大小调整模式
shapesize() | *turtlesize()* 形状大小
shearfactor() 剪切因子
settiltangle() 设置倾角
tiltangle() 倾角
tilt() 倾斜
shapetransform() 变形
get_shapepoly() 获取形状多边形

使用事件

onclick() 当鼠标点击
onrelease() 当鼠标释放
ondrag() 当鼠标拖动

特殊海龟方法

begin_poly() 开始记录多边形
end_poly() 结束记录多边形
get_poly() 获取多边形
clone() 克隆
getturtle() | *getpen()* 获取海龟画笔
getscreen() 获取屏幕
setundobuffer() 设置撤消缓冲区

undobufferentries() 撤消缓冲区条目数

TurtleScreen/Screen 方法

窗口控制

bgcolor() 背景颜色
bgpic() 背景图片
clearscreen()
resetscreen()
screensize() 屏幕大小
setworldcoordinates() 设置世界坐标系

动画控制

delay() 延迟
tracer() 追踪
update() 更新

使用屏幕事件

listen() 监听
onkey() | *onkeyrelease()* 当键盘按下并释放
onkeypress() 当键盘按下
onclick() | *onscreenclick()* 当点击屏幕
ontimer() 当达到定时
mainloop() | *done()* 主循环

设置与特殊方法

mode() 模式
colormode() 颜色模式
getcanvas() 获取画布
getshapes() 获取形状
register_shape() | *addshape()* 添加形状
turtles() 所有海龟
window_height() 窗口高度
window_width() 窗口宽度

输入方法

textinput() 文本输入
numinput() 数字输入

Screen 专有方法

bye() 退出
exitonclick() 当点击时退出
setup() 设置
title() 标题

24.1.3 RawTurtle/Turtle 方法和对应函数

本节中的大部分示例都使用 Turtle 类的一个实例，命名为 `turtle`。

海龟动作

`turtle.`**`forward`**(*distance*)
`turtle.`**`fd`**(*distance*)

参数 `distance` -- 一个数值 (整型或浮点型)

海龟前进 *distance* 指定的距离，方向为海龟的朝向。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.forward(25)
>>> turtle.position()
(25.00,0.00)
>>> turtle.forward(-75)
>>> turtle.position()
(-50.00,0.00)
```

`turtle.`**`back`**(*distance*)
`turtle.`**`bk`**(*distance*)
`turtle.`**`backward`**(*distance*)

参数 `distance` -- 一个数值

海龟后退 *distance* 指定的距离，方向与海龟的朝向相反。不改变海龟的朝向。

```
>>> turtle.position()
(0.00,0.00)
>>> turtle.backward(30)
>>> turtle.position()
(-30.00,0.00)
```

`turtle.`**`right`**(*angle*)
`turtle.`**`rt`**(*angle*)

参数 `angle` -- 一个数值 (整型或浮点型)

海龟右转 *angle* 个单位。(单位默认为角度，但可通过*degrees()* 和*radians()* 函数改变设置。) 角度的正负由海龟模式确定，参见*mode()*。

```
>>> turtle.heading()
22.0
>>> turtle.right(45)
>>> turtle.heading()
337.0
```

`turtle.`**`left`**(*angle*)
`turtle.`**`lt`**(*angle*)

参数 `angle` -- 一个数值 (整型或浮点型)

海龟左转 *angle* 个单位。(单位默认为角度，但可通过*degrees()* 和*radians()* 函数改变设置。) 角度的正负由海龟模式确定，参见*mode()*。

```
>>> turtle.heading()
22.0
>>> turtle.left(45)
>>> turtle.heading()
67.0
```

turtle.**goto**(*x*, *y=None*)
turtle.**setpos**(*x*, *y=None*)
turtle.**setposition**(*x*, *y=None*)

參數

- **x** -- 一个数值或数值对/向量
- **y** -- 一个数值或 None

如果 *y* 为 None，*x* 应为一个表示坐标的数值对或 *Vec2D* 类对象 (例如 *pos()* 返回的对象).

海龟移动到一个绝对坐标。如果画笔已落下将会画线。不改变海龟的朝向。

```
>>> tp = turtle.pos()
>>> tp
(0.00,0.00)
>>> turtle.setpos(60,30)
>>> turtle.pos()
(60.00,30.00)
>>> turtle.setpos((20,80))
>>> turtle.pos()
(20.00,80.00)
>>> turtle.setpos(tp)
>>> turtle.pos()
(0.00,0.00)
```

turtle.**setx**(*x*)

參數 x -- 一个数值 (整型或浮点型)

设置海龟的横坐标为 *x*，纵坐标保持不变。

```
>>> turtle.position()
(0.00,240.00)
>>> turtle.setx(10)
>>> turtle.position()
(10.00,240.00)
```

turtle.**sety**(*y*)

參數 y -- 一个数值 (整型或浮点型)

设置海龟的纵坐标为 *y*，横坐标保持不变。

```
>>> turtle.position()
(0.00,40.00)
>>> turtle.sety(-10)
>>> turtle.position()
(0.00,-10.00)
```

turtle.**setheading**(*to_angle*)
turtle.**seth**(*to_angle*)

參數 to_angle -- 一个数值 (整型或浮点型)

设置海龟的朝向为 *to_angle*。以下是以角度表示的几个常用方向：

标准模式	logo 模式
0 - 东	0 - 北
90 - 北	90 - 东
180 - 西	180 - 南
270 - 南	270 - 西

```
>>> turtle.setheading(90)
>>> turtle.heading()
90.0
```

turtle.**home**()

海龟移至初始坐标 (0,0)，并设置朝向为初始方向 (由海龟模式确定，参见*mode()*)。

```
>>> turtle.heading()
90.0
>>> turtle.position()
(0.00,-10.00)
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**circle**(*radius*, *extent=None*, *steps=None*)

參數

- **radius** -- 一个数值
- **extent** -- 一个数值 (或 None)
- **steps** -- 一个整型数 (或 None)

绘制一个 *radius* 指定半径的圆。圆心在海龟左边 *radius* 个单位；*extent* 为一个夹角，用来决定绘制圆的一部分。如未指定 *extent** 则绘制整个圆。如果 **extent* 不是完整圆周，则以当前画笔位置为一个端点绘制圆弧。如果 *radius* 为正值则朝逆时针方向绘制圆弧，否则朝顺时针方向。最终海龟的朝向会依据 *extent* 的值而改变。

圆实际是以其内切正多边形来近似表示的，其边的数量由 *steps* 指定。如果未指定边数则会自动确定。此方法也可用来绘制正多边形。

```
>>> turtle.home()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(50)
>>> turtle.position()
(-0.00,0.00)
>>> turtle.heading()
0.0
>>> turtle.circle(120, 180)  # draw a semicircle
>>> turtle.position()
(0.00,240.00)
>>> turtle.heading()
180.0
```

turtle.**dot**(*size=None*, **color*)

参数

- **size** -- 一个整型数 >= 1 (如果指定)
- **color** -- 一个颜色字符串或颜色数值元组

绘制一个直径为 *size*，颜色为 *color* 的圆点。如果 *size* 未指定，则直径取 pensize+4 和 2*pensize 中的较大值。

```
>>> turtle.home()
>>> turtle.dot()
>>> turtle.fd(50); turtle.dot(20, "blue"); turtle.fd(50)
>>> turtle.position()
(100.00,-0.00)
>>> turtle.heading()
0.0
```

turtle.**stamp**()

在海龟当前位置印制一个海龟形状。返回该印章的 stamp_id，印章可以通过调用 `clearstamp(stamp_id)` 来删除。

```
>>> turtle.color("blue")
>>> turtle.stamp()
11
>>> turtle.fd(50)
```

turtle.**clearstamp**(*stampid*)

参数 stampid -- 一个整型数，必须是之前*stamp()* 调用的返回值

删除 *stampid* 指定的印章。

```
>>> turtle.position()
(150.00,-0.00)
>>> turtle.color("blue")
>>> astamp = turtle.stamp()
>>> turtle.fd(50)
>>> turtle.position()
(200.00,-0.00)
>>> turtle.clearstamp(astamp)
>>> turtle.position()
(200.00,-0.00)
```

turtle.**clearstamps**(*n=None*)

参数 n -- 一个整型数 (或 None)

删除全部或前/后 *n* 个海龟印章。如果 *n* 为 None 则删除全部印章，如果 $n > 0$ 则删除前 *n* 个印章，否则如果 $n < 0$ 则删除后 *n* 个印章。

```
>>> for i in range(8):
...     turtle.stamp(); turtle.fd(30)
13
14
15
16
17
18
```

(下页继续)

(繼續上一頁)

```
19
20
>>> turtle.clearstamps(2)
>>> turtle.clearstamps(-2)
>>> turtle.clearstamps()
```

turtle.**undo**()

撤消 (或连续撤消) 最近的一个 (或多个) 海龟动作。可撤消的次数由撤消缓冲区的大小决定。

```
>>> for i in range(4):
...     turtle.fd(50); turtle.lt(80)
...
>>> for i in range(8):
...     turtle.undo()
```

turtle.**speed**(*speed=None*)

参数 **speed** -- 一个 0..10 范围内的整型数或速度字符串 (见下)

设置海龟移动的速度为 0..10 表示的整型数值。如未指定参数则返回当前速度。

如果输入数值大于 10 或小于 0.5 则速度设为 0。速度字符串与速度值的对应关系如下:

- ”fastest”: 0 最快
- ”fast”: 10 快
- ”normal”: 6 正常
- ”slow”: 3 慢
- ”slowest”: 1 最慢

速度值从 1 到 10，画线和海龟转向的动画效果逐级加快。

注意: *speed* = 0 表示 没有动画效果。forward/back 将使海龟向前/向后跳跃，同样的 left/right 将使海龟立即改变朝向。

```
>>> turtle.speed()
3
>>> turtle.speed('normal')
>>> turtle.speed()
6
>>> turtle.speed(9)
>>> turtle.speed()
9
```

获取海龟的状态

turtle.**position**()
turtle.**pos**()

返回海龟当前的坐标 (x,y) (为*Vec2D* 矢量类对象)。

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

参数

- **x** -- 一个数值或数值对/矢量，或一个海龟实例
- **y** -- 一个数值——如果 *x* 是一个数值，否则为 None

返回从海龟位置到由 (x,y)、矢量或另一海龟所确定位置的连线的夹角。此数值依赖于海龟的初始朝向，这又取决于”standard”/”world” 或”logo” 模式设置。

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

turtle.**xcor**()
返回海龟的 x 坐标。

```
>>> turtle.home()
>>> turtle.left(50)
>>> turtle.forward(100)
>>> turtle.pos()
(64.28,76.60)
>>> print(round(turtle.xcor(), 5))
64.27876
```

turtle.**ycor**()
返回海龟的 y 坐标。

```
>>> turtle.home()
>>> turtle.left(60)
>>> turtle.forward(100)
>>> print(turtle.pos())
(50.00,86.60)
>>> print(round(turtle.ycor(), 5))
86.60254
```

turtle.**heading**()
返回海龟当前的朝向 (数值依赖于海龟模式参见*mode()*)。

```
>>> turtle.home()
>>> turtle.left(67)
>>> turtle.heading()
67.0
```

turtle.**distance**(*x*, *y=None*)

参数

- **x** -- 一个数值或数值对/矢量，或一个海龟实例
- **y** -- 一个数值——如果 *x* 是一个数值，否则为 None

返回从海龟位置到由 (x,y)，适量或另一海龟对应位置的单位距离。

```
>>> turtle.home()
>>> turtle.distance(30,40)
50.0
>>> turtle.distance((30,40))
50.0
>>> joe = Turtle()
>>> joe.forward(77)
>>> turtle.distance(joe)
77.0
```

度量单位设置

turtle.**degrees**(*fullcircle=360.0*)

参数 fullcircle -- 一个数值

设置角度的度量单位，即设置一个圆周为多少” 度”。默认值为 360 度。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0

Change angle measurement unit to grad (also known as gon,
grade, or gradian and equals 1/100-th of the right angle.)
>>> turtle.degrees(400.0)
>>> turtle.heading()
100.0
>>> turtle.degrees(360)
>>> turtle.heading()
90.0
```

turtle.**radians**()

设置角度的度量单位为弧度。其值等于 `degrees(2*math.pi)`。

```
>>> turtle.home()
>>> turtle.left(90)
>>> turtle.heading()
90.0
>>> turtle.radians()
>>> turtle.heading()
1.5707963267948966
```

画笔控制

绘图状态

turtle.**pendown**()
turtle.**pd**()
turtle.**down**()

画笔落下 -- 移动时将画线。

turtle.**penup**()
turtle.**pu**()
turtle.**up**()

画笔抬起 -- 移动时不画线。

turtle.**pensize**(*width=None*)
turtle.**width**(*width=None*)

参数 width -- 一个正数值

设置线条的粗细为 *width* 或返回该值。如果 resizemode 设为”auto” 并且 turtleshape 为多边形，该多边形也以同样组细的线条绘制。如未指定参数，则返回当前的 pensize。

```
>>> turtle.pensize()
1
>>> turtle.pensize(10)   # from here on lines of width 10 are drawn
```

turtle.**pen**(*pen=None*, ***pendict*)

参数

- **pen** -- 一个包含部分或全部下列键的字典
- **pendict** -- 一个或多个以下列键为关键字的关键字参数

返回或设置画笔的属性，以一个包含以下键值对的” 画笔字典” 表示:

- ”shown”: True/False
- ”pendown”: True/False
- ”pencolor”: 颜色字符串或颜色元组
- ”fillcolor”: 颜色字符串或颜色元组
- ”pensize”: 正数值
- ”speed”: 0..10 范围内的数值
- ”resizemode”: ”auto” 或”user” 或”noresize”
- ”stretchfactor”: (正数值, 正数值)
- ”outline”: 正数值
- ”tilt”: 数值

此字典可作为后续调用*pen()* 时的参数，以恢复之前的画笔状态。另外还可将这些属性作为关键词参数提交。使用此方式可以用一条语句设置画笔的多个属性。

```
>>> turtle.pen(fillcolor="black", pencolor="red", pensize=10)
>>> sorted(turtle.pen().items())
[('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'),
 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'),
 ('shearfactor', 0.0), ('shown', True), ('speed', 9),
 ('stretchfactor', (1.0, 1.0)), ('tilt', 0.0)]
>>> penstate=turtle.pen()
>>> turtle.color("yellow", "")
>>> turtle.penup()
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow')]
>>> turtle.pen(penstate, fillcolor="green")
>>> sorted(turtle.pen().items())[:3]
[('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red')]
```

turtle.**isdown**()

如果画笔落下返回 `True`，如果画笔抬起返回 `False`。

```
>>> turtle.penup()
>>> turtle.isdown()
False
>>> turtle.pendown()
>>> turtle.isdown()
True
```

颜色控制

turtle.**pencolor**(**args*)

返回或设置画笔颜色。

允许以下四种输入格式:

pencolor() 返回以颜色描述字符串或元组 (见示例) 表示的当前画笔颜色。可用作其他 color/pencolor/fillcolor 调用的输入。

pencolor(colorstring) 设置画笔颜色为 *colorstring* 指定的 Tk 颜色描述字符串，例如 `"red"`、`"yellow"` 或 `"#33cc8c"`。

pencolor((r, g, b)) 设置画笔颜色为以 *r*, *g*, *b* 元组表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode，colormode 的值为 1.0 或 255 (参见*colormode()*)。

pencolor(r, g, b) 设置画笔颜色为以 *r*, *g*, *b* 表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode。

如果 turtleshape 为多边形，该多边形轮廓也以新设置的画笔颜色绘制。

```
>>> colormode()
1.0
>>> turtle.pencolor()
'red'
>>> turtle.pencolor("brown")
>>> turtle.pencolor()
'brown'
>>> tup = (0.2, 0.8, 0.55)
>>> turtle.pencolor(tup)
>>> turtle.pencolor()
(0.2, 0.8, 0.5490196078431373)
>>> colormode(255)
>>> turtle.pencolor()
(51.0, 204.0, 140.0)
>>> turtle.pencolor('#32c18f')
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

turtle.**fillcolor**(**args*)

返回或设置填充颜色。

允许以下四种输入格式:

fillcolor() 返回以颜色描述字符串或元组 (见示例) 表示的当前填充颜色。可用作其他 color/pencolor/fillcolor 调用的输入。

fillcolor(colorstring) 设置填充颜色为 *colorstring* 指定的 Tk 颜色描述字符串，例如 `"red"`、`"yellow"` 或 `"#33cc8c"`。

fillcolor((r, g, b)) 设置填充颜色为以 *r*, *g*, *b* 元组表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode，colormode 的值为 1.0 或 255 (参见*colormode()*)。

fillcolor(r, g, b) 设置填充颜色为 *r*, *g*, *b* 表示的 RGB 颜色。*r*, *g*, *b* 的取值范围应为 0..colormode。

如果 turtleshape 为多边形，该多边形内部也以新设置的填充颜色填充。

```
>>> turtle.fillcolor("violet")
>>> turtle.fillcolor()
'violet'
>>> turtle.pencolor()
(50.0, 193.0, 143.0)
```

(下页继续)

(繼續上一頁)

```
>>> turtle.fillcolor((50, 193, 143))  # Integers, not floats
>>> turtle.fillcolor()
(50.0, 193.0, 143.0)
>>> turtle.fillcolor('#ffffff')
>>> turtle.fillcolor()
(255.0, 255.0, 255.0)
```

turtle.**color**(**args*)

返回或设置画笔颜色和填充颜色。

允许多种输入格式。使用如下 0 至 3 个参数:

color() 返回以一对颜色描述字符串或元组表示的当前画笔颜色和填充颜色，两者可分别由*pencolor()* 和*fillcolor()* 返回。

color(colorstring),color((r,g,b)),color(r,g,b) 输入格式与*pencolor()* 相同，同时设置填充颜色和画笔颜色为指定的值。

color(colorstring1, colorstring2),color((r1,g1,b1), (r2,g2,b2)) 相当于 pencolor(colorstring1) 加 fillcolor(colorstring2)，使用其他输入格式的方法也与之类似。

如果 turtleshape 为多边形，该多边形轮廓与填充也使用新设置的颜色。

```
>>> turtle.color("red", "green")
>>> turtle.color()
('red', 'green')
>>> color("#285078", "#a0c8f0")
>>> color()
((40.0, 80.0, 120.0), (160.0, 200.0, 240.0))
```

另参见: Screen 方法*colormode()*。

填充

turtle.**filling**()

返回填充状态 (填充为 True，否则为 False)。

```
>>> turtle.begin_fill()
>>> if turtle.filling():
...     turtle.pensize(5)
... else:
...     turtle.pensize(3)
```

turtle.**begin_fill**()

在绘制要填充的形状之前调用。

turtle.**end_fill**()

填充上次调用*begin_fill()* 之后绘制的形状。

自相交多边形或多个形状间的重叠区域是否填充取决于操作系统的图形引擎、重叠的类型以及重叠的层数。例如上面的 Turtle 多芒星可能会全部填充为黄色，也可能会有一些白色区域。

```
>>> turtle.color("black", "red")
>>> turtle.begin_fill()
>>> turtle.circle(80)
>>> turtle.end_fill()
```

更多绘图控制

turtle.**reset**()
从屏幕中删除海龟的绘图，海龟回到原点并设置所有变量为默认值。

```
>>> turtle.goto(0,-22)
>>> turtle.left(100)
>>> turtle.position()
(0.00,-22.00)
>>> turtle.heading()
100.0
>>> turtle.reset()
>>> turtle.position()
(0.00,0.00)
>>> turtle.heading()
0.0
```

turtle.**clear**()
从屏幕中删除指定海龟的绘图。不移动海龟。海龟的状态和位置以及其他海龟的绘图不受影响。

turtle.**write**(*arg*, *move=False*, *align="left"*, *font=("Arial", 8, "normal")*)

参数

- **arg** -- 要书写到 TurtleScreen 的对象
- **move** -- True/False
- **align** -- 字符串"left", "center" 或"right"
- **font** -- 一个三元组 (fontname, fontsize, fonttype)

基于 *align* ("left", "center" 或"right") 并使用给定的字体将文本——*arg* 的字符串表示形式——写到当前海龟位置。如果 *move* 为真值，画笔会移至文本的右下角。默认情况下 *move* 为 `False`。

```
>>> turtle.write("Home = ", True, align="center")
>>> turtle.write((0,0), True)
```

海龟状态

可见性

turtle.**hideturtle**()
turtle.**ht**()
使海龟不可见。当你绘制复杂图形时这是个好主意，因为隐藏海龟可显著加快绘制速度。

```
>>> turtle.hideturtle()
```

turtle.**showturtle**()
turtle.**st**()
使海龟可见。

```
>>> turtle.showturtle()
```

turtle.**isvisible**()
如果海龟显示返回 `True`，如果海龟隐藏返回 `False`。

```
>>> turtle.hideturtle()
>>> turtle.isvisible()
False
>>> turtle.showturtle()
>>> turtle.isvisible()
True
```

外观

turtle.**shape**(*name=None*)

> **参数 name** -- 一个有效的形状名字符串

设置海龟形状为 *name* 指定的形状名，如未指定形状名则返回当前的形状名。*name* 指定的形状名应存在于 TurtleScreen 的 shape 字典中。多边形的形状初始时有以下几种: ”arrow”, ”turtle”, ”circle”, ”square”, ”triangle”, ”classic”。要了解如何处理形状请参看 Screen 方法*register_shape()*。

```
>>> turtle.shape()
'classic'
>>> turtle.shape("turtle")
>>> turtle.shape()
'turtle'
```

turtle.**resizemode**(*rmode=None*)

> **参数 rmode** -- 字符串”auto”, ”user”, ”noresize” 其中之一

设置大小调整模式为以下值之一: ”auto”, ”user”, ”noresize”。如未指定 *rmode* 则返回当前的大小调整模式。不同的大小调整模式的效果如下:

- ”auto”: 根据画笔粗细值调整海龟的外观。
- ”user”: 根据拉伸因子和轮廓宽度 (outline) 值调整海龟的外观，两者是由*shapesize()* 设置的。
- ”noresize”: 不调整海龟的外观大小。

`resizemode("user")` 会由*shapesize()* 带参数使用时被调用。

```
>>> turtle.resizemode()
'noresize'
>>> turtle.resizemode("auto")
>>> turtle.resizemode()
'auto'
```

turtle.**shapesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)
turtle.**turtlesize**(*stretch_wid=None*, *stretch_len=None*, *outline=None*)

> **参数**
>
> - **stretch_wid** -- 正数值
> - **stretch_len** -- 正数值
> - **outline** -- 正数值

返回或设置画笔的属性 x/y-拉伸因子和/或轮廓。设置大小调整模式为”user”。当且仅当大小调整模式设为”user” 时海龟会基于其拉伸因子调整外观: *stretch_wid* 为垂直于其朝向的宽度拉伸因子，*stretch_len* 为平等于其朝向的长度拉伸因子，决定形状轮廓线的粗细。

```
>>> turtle.shapesize()
(1.0, 1.0, 1)
>>> turtle.resizemode("user")
>>> turtle.shapesize(5, 5, 12)
>>> turtle.shapesize()
(5, 5, 12)
>>> turtle.shapesize(outline=8)
>>> turtle.shapesize()
(5, 5, 8)
```

turtle.**shearfactor**(*shear=None*)

参数 shear -- 数值 (可选)

设置或返回当前的剪切因子。根据 share 指定的剪切因子即剪切角度的切线来剪切海龟形状。不改变海龟的朝向 (移动方向)。如未指定 shear 参数: 返回当前的剪切因子即剪切角度的切线，与海龟朝向平行的线条将被剪切。

```
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.shearfactor(0.5)
>>> turtle.shearfactor()
0.5
```

turtle.**tilt**(*angle*)

参数 angle -- 一个数值

海龟形状自其当前的倾角转动 *angle* 指定的角度，但 不改变海龟的朝向 (移动方向)。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(30)
>>> turtle.fd(50)
>>> turtle.tilt(30)
>>> turtle.fd(50)
```

turtle.**settiltangle**(*angle*)

参数 angle -- 一个数值

旋转海龟形状使其指向 *angle* 指定的方向，忽略其当前的倾角，不改变海龟的朝向 (移动方向)。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.settiltangle(45)
>>> turtle.fd(50)
>>> turtle.settiltangle(-45)
>>> turtle.fd(50)
```

3.1 版後已🄵用.

turtle.**tiltangle**(*angle=None*)

参数 angle -- 一个数值 (可选)

设置或返回当前的倾角。如果指定 angle 则旋转海龟形状使其指向 angle 指定的方向，忽略其当前的倾角。不改变海龟的朝向 (移动方向)。如果未指定 angle: 返回当前的倾角，即海龟形状的方向和海龟朝向 (移动方向) 之间的夹角。

```
>>> turtle.reset()
>>> turtle.shape("circle")
>>> turtle.shapesize(5,2)
>>> turtle.tilt(45)
>>> turtle.tiltangle()
45.0
```

turtle.**shapetransform**(*t11=None*, *t12=None*, *t21=None*, *t22=None*)

参数

- **t11** -- 一个数值 (可选)
- **t12** -- 一个数值 (可选)
- **t21** -- 一个数值 (可选)
- **t12** -- 一个数值 (可选)

设置或返回海龟形状的当前变形矩阵。

如未指定任何矩阵元素，则返回以 4 元素元组表示的变形矩阵。否则就根据设置指定元素的矩阵来改变海龟形状，矩阵第一排的值为 t11, t12 而第二排的值为 t21, t22。行列式 t11 * t22 - t12 * t21 必须不为零，否则会引发错误。根据指定矩阵修改拉伸因子 stretchfactor, 剪切因子 shearfactor 和倾角 tiltangle。

```
>>> turtle = Turtle()
>>> turtle.shape("square")
>>> turtle.shapesize(4,2)
>>> turtle.shearfactor(-0.5)
>>> turtle.shapetransform()
(4.0, -1.0, -0.0, 2.0)
```

turtle.**get_shapepoly**()

返回以坐标值对元组表示的当前形状多边形。这可以用于定义一个新形状或一个复合形状的多个组成部分。

```
>>> turtle.shape("square")
>>> turtle.shapetransform(4, -1, 0, 2)
>>> turtle.get_shapepoly()
((50, -20), (30, 20), (-50, 20), (-30, -20))
```

使用事件

turtle.**onclick**(*fun*, *btn=1*, *add=None*)

参数

- **fun** -- 一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** -- 鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** -- `True` 或 `False` -- 如为 `True` 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到鼠标点击此海龟事件。如果 *fun* 值为 `None`，则移除现有的绑定。以下为使用匿名海龟即过程式的示例:

```
>>> def turn(x, y):
...     left(180)
...
```

(下页继续)

(繼續上一頁)

```
>>> onclick(turn)  # Now clicking into the turtle will turn it.
>>> onclick(None)  # event-binding will be removed
```

turtle.**onrelease**(*fun*, *btn=1*, *add=None*)

參數

- **fun** -- 一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** -- 鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** -- `True` 或 `False` -- 如为 `True` 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到在此海龟上释放鼠标按键事件。如果 *fun* 值为 `None`，则移除现有的绑定。

```
>>> class MyTurtle(Turtle):
...     def glow(self,x,y):
...         self.fillcolor("red")
...     def unglow(self,x,y):
...         self.fillcolor("")
...
>>> turtle = MyTurtle()
>>> turtle.onclick(turtle.glow)      # clicking on turtle turns fillcolor red,
>>> turtle.onrelease(turtle.unglow) # releasing turns it to transparent.
```

turtle.**ondrag**(*fun*, *btn=1*, *add=None*)

參數

- **fun** -- 一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** -- 鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** -- `True` 或 `False` -- 如为 `True` 则将添加一个新绑定，否则将取代先前的绑定

将 *fun* 指定的函数绑定到在此海龟上移动鼠标事件。如果 *fun* 值为 `None`，则移除现有的绑定。

注: 在海龟上移动鼠标事件之前应先发生在此海龟上点击鼠标事件。

```
>>> turtle.ondrag(turtle.goto)
```

在此之后点击并拖动海龟可在屏幕上手绘线条 (如果画笔为落下)。

特殊海龟方法

turtle.**begin_poly**()
开始记录多边形的顶点。当前海龟位置为多边形的第一个顶点。

turtle.**end_poly**()
停止记录多边形的顶点。当前海龟位置为多边形的最后一个顶点。它将连线到第一个顶点。

turtle.**get_poly**()
返回最新记录的多边形。

```
>>> turtle.home()
>>> turtle.begin_poly()
>>> turtle.fd(100)
>>> turtle.left(20)
>>> turtle.fd(30)
>>> turtle.left(60)
```

(下页继续)

(繼續上一頁)

```
>>> turtle.fd(50)
>>> turtle.end_poly()
>>> p = turtle.get_poly()
>>> register_shape("myFavouriteShape", p)
```

turtle.**clone**()
创建并返回海龟的克隆体，具有相同的位置、朝向和海龟属性。

```
>>> mick = Turtle()
>>> joe = mick.clone()
```

turtle.**getturtle**()
turtle.**getpen**()
返回海龟对象自身。唯一合理的用法: 作为一个函数来返回" 匿名海龟":

```
>>> pet = getturtle()
>>> pet.fd(50)
>>> pet
<turtle.Turtle object at 0x...>
```

turtle.**getscreen**()
返回作为海龟绘图场所的*TurtleScreen* 类对象。该对象将可调用 TurtleScreen 方法。

```
>>> ts = turtle.getscreen()
>>> ts
<turtle._Screen object at 0x...>
>>> ts.bgcolor("pink")
```

turtle.**setundobuffer**(*size*)

参数 size -- 一个整型数值或 None

设置或禁用撤销缓冲区。如果 *size* 为整数，则开辟一个给定大小的空撤销缓冲区。*size* 给出了可以通过*undo()* 方法/函数撤销海龟动作的最大次数。如果 *size* 为 None，则禁用撤销缓冲区。

```
>>> turtle.setundobuffer(42)
```

turtle.**undobufferentries**()
返回撤销缓冲区里的条目数。

```
>>> while undobufferentries():
...     undo()
```

复合形状

要使用由多个不同颜色多边形构成的复合海龟形状，你必须明确地使用辅助类*Shape*，具体步骤如下:

1. 创建一个空 Shape 对象，类型为"compound"。
2. 按照需要使用 addcomponent() 方法向此对象添加多个部件。

   例如:

```
>>> s = Shape("compound")
>>> poly1 = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s.addcomponent(poly1, "red", "blue")
```

(下页继续)

(繼續上一頁)

```
>>> poly2 = ((0,0),(10,-5),(-10,-5))
>>> s.addcomponent(poly2, "blue", "red")
```

3. 接下来将 Shape 对象添加到 Screen 对象的形状列表并使用它:

```
>>> register_shape("myshape", s)
>>> shape("myshape")
```

備F: *Shape* 类在*register_shape()* 方法的内部以多种方式使用。应用程序编写者 只有在使用上述的复合形状时才需要处理 Shape 类。

24.1.4 TurtleScreen/Screen 方法及对应函数

本节中的大部分示例都使用 TurtleScreen 类的一个实例，命名为 `screen`。

窗口控制

`turtle.`**`bgcolor`**(**args*)

> **参数 `args`** -- 一个颜色字符串或三个取值范围 0..colormode 内的数值或一个取值范围相同的数值 3 元组

设置或返回 TurtleScreen 的背景颜色。

```
>>> screen.bgcolor("orange")
>>> screen.bgcolor()
'orange'
>>> screen.bgcolor("#800080")
>>> screen.bgcolor()
(128.0, 0.0, 128.0)
```

`turtle.`**`bgpic`**(*picname=None*)

> **参数 `picname`** -- 一个字符串, gif-文件名, `"nopic"`, 或 `None`

设置背景图片或返回当前背景图片名称。如果 *picname* 为一个文件名，则将相应图片设为背景。如果 *picname* 为 `"nopic"`，则删除当前背景图片。如果 *picname* 为 `None`，则返回当前背景图片文件名。:

```
>>> screen.bgpic()
'nopic'
>>> screen.bgpic("landscape.gif")
>>> screen.bgpic()
"landscape.gif"
```

`turtle.`**`clear`**()

> 備F: 此 TurtleScreen 方法作为全局函数时只有一个名字 `clearscreen`。全局函数 `clear` 所对应的是 Turtle 方法 `clear`。

turtle.**clearscreen**()
从中删除所有海龟的全部绘图。将已清空的 TurtleScreen 重置为初始状态: 白色背景，无背景片，无事件绑定并启用追踪。

turtle.**reset**()

備𝔽: 此 TurtleScreen 方法作为全局函数时只有一个名字 `resetscreen`。全局函数 `reset` 所对应的是 Turtle 方法 `reset`。

---

turtle.**resetscreen**()
重置屏幕上的所有海龟为其初始状态。

turtle.**screensize**(*canvwidth=None*, *canvheight=None*, *bg=None*)

参数

- **canvwidth** -- 正整型数，以像素表示画布的新宽度值
- **canvheight** -- 正整型数，以像素表示画面的新高度值
- **bg** -- 颜色字符串或颜色元组，新的背景颜色

如未指定任何参数，则返回当前的 (canvaswidth, canvasheight)。否则改变作为海龟绘图场所的画布大小。不改变绘图窗口。要观察画布的隐藏区域，可以使用滚动条。通过此方法可以令之前绘制于画布之外的图形变为可见。

```
>>> screen.screensize()
(400, 300)
>>> screen.screensize(2000,1500)
>>> screen.screensize()
(2000, 1500)
```

也可以用来寻找意外逃走的海龟;-)

turtle.**setworldcoordinates**(*llx*, *lly*, *urx*, *ury*)

参数

- **llx** -- 一个数值, 画布左下角的 x-坐标
- **lly** -- 一个数值, 画布左下角的 y-坐标
- **urx** -- 一个数值, 画面右上角的 x-坐标
- **ury** -- 一个数值, 画布右上角的 y-坐标

设置用户自定义坐标系并在必要时切换模式为”world”。这会执行一次 `screen.reset()`。如果”world”模式已激活，则所有图形将根据新的坐标系重绘。

注意: 在用户自定义坐标系中，角度可能显得扭曲。

```
>>> screen.reset()
>>> screen.setworldcoordinates(-50,-7.5,50,7.5)
>>> for _ in range(72):
...     left(10)
...
>>> for _ in range(8):
...     left(45); fd(2)   # a regular octagon
```

动画控制

turtle.**delay**(*delay=None*)

参数 delay -- 正整型数

设置或返回以毫秒数表示的延迟值 *delay*。(这约等于连续两次画布刷新的间隔时间。) 绘图延迟越长，动画速度越慢。

可选参数:

```
>>> screen.delay()
10
>>> screen.delay(5)
>>> screen.delay()
5
```

turtle.**tracer**(*n=None*, *delay=None*)

参数

- **n** -- 非负整型数
- **delay** -- 非负整型数

启用/禁用海龟动画并设置刷新图形的延迟时间。如果指定 *n* 值，则只有每第 n 次屏幕刷新会实际执行。(可被用来加速复杂图形的绘制。) 如果调用时不带参数，则返回当前保存的 n 值。第二个参数设置延迟值 (参见*delay()*)。

```
>>> screen.tracer(8, 25)
>>> dist = 2
>>> for i in range(200):
...     fd(dist)
...     rt(90)
...     dist += 2
```

turtle.**update**()

执行一次 TurtleScreen 刷新。在禁用追踪时使用。

另参见 RawTurtle/Turtle 方法*speed()*。

使用屏幕事件

turtle.**listen**(*xdummy=None*, *ydummy=None*)

设置焦点到 TurtleScreen (以便接收按键事件)。使用两个 Dummy 参数以便能够传递*listen()* 给 onclick 方法。

turtle.**onkey**(*fun*, *key*)
turtle.**onkeyrelease**(*fun*, *key*)

参数

- **fun** -- 一个无参数的函数或 None
- **key** -- 一个字符串: 键 (例如"a") 或键标 (例如"space")

绑定 *fun* 指定的函数到按键释放事件。如果 *fun* 值为 None，则移除事件绑定。注: 为了能够注册按键事件，TurtleScreen 必须得到焦点。(参见 method *listen()* 方法。)

```
>>> def f():
...     fd(50)
...     lt(60)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onkeypress**(*fun*, *key=None*)

參數

- **fun** -- 一个无参数的函数或 None
- **key** -- 一个字符串: 键 (例如"a") 或键标 (例如"space")

绑定 *fun* 指定的函数到指定键的按下事件。如未指定键则绑定到任意键的按下事件。注: 为了能够注册按键事件，必须得到焦点。(参见 *listen()* 方法。)

```
>>> def f():
...     fd(50)
...
>>> screen.onkey(f, "Up")
>>> screen.listen()
```

turtle.**onclick**(*fun*, *btn=1*, *add=None*)
turtle.**onscreenclick**(*fun*, *btn=1*, *add=None*)

參數

- **fun** -- 一个函数，调用时将传入两个参数表示在画布上点击的坐标。
- **btn** -- 鼠标按钮编号，默认值为 1 (鼠标左键)
- **add** -- True 或 False -- 如为 True 则将添加一个新绑定，否则将取代先前的绑定

绑定 *fun* 指定的函数到鼠标点击屏幕事件。如果 *fun* 值为 None，则移除现有的绑定。

以下示例使用一个 TurtleScreen 实例 screen 和一个 Turtle 实例 turtle:

```
>>> screen.onclick(turtle.goto) # Subsequently clicking into the TurtleScreen will
>>>                             # make the turtle move to the clicked point.
>>> screen.onclick(None)        # remove event binding again
```

備𝔽: 此 TurtleScreen 方法作为全局函数时只有一个名字 onscreenclick。全局函数 onclick 所对应的是 Turtle 方法 onclick。

---

turtle.**ontimer**(*fun*, *t=0*)

參數

- **fun** -- 一个无参数的函数
- **t** -- 一个数值 >= 0

安装一个计时器，在 *t* 毫秒后调用 *fun* 函数。

```
>>> running = True
>>> def f():
...     if running:
...         fd(50)
```

(下页继续)

(繼續上一頁)

```
...         lt(60)
...         screen.ontimer(f, 250)
>>> f()   ### makes the turtle march around
>>> running = False
```

turtle.**mainloop**()
turtle.**done**()

开始事件循环 - 调用 Tkinter 的 mainloop 函数。必须作为一个海龟绘图程序的结束语句。如果一个脚本是在以 -n 模式 (无子进程) 启动的 IDLE 中运行时 不可使用 - 用于实现海龟绘图的交互功能。:

```
>>> screen.mainloop()
```

输入方法

turtle.**textinput**(*title*, *prompt*)

参数

- **title** -- 字符串
- **prompt** -- 字符串

弹出一个对话框窗口用来输入一个字符串。形参 title 为对话框窗口的标题，prompt 为一条文本，通常用来提示要输入什么信息。返回输入的字符串。如果对话框被取消则返回 None。:

```
>>> screen.textinput("NIM", "Name of first player:")
```

turtle.**numinput**(*title*, *prompt*, *default=None*, *minval=None*, *maxval=None*)

参数

- **title** -- 字符串
- **prompt** -- 字符串
- **default** -- 数值 (可选)
- **minval** -- 数值 (可选)
- **maxval** -- 数值 (可选)

弹出一个对话框窗口用来输入一个数值。title 为对话框窗口的标题，prompt 为一条文本，通常用来描述要输入的数值信息。default: 默认值, minval: 可输入的最小值, maxval: 可输入的最大值。输入数值的必须在指定的 minval .. maxval 范围之内，否则将给出一条提示，对话框保持打开等待修改。返回输入的数值。如果对话框被取消则返回 None。:

```
>>> screen.numinput("Poker", "Your stakes:", 1000, minval=10, maxval=10000)
```

设置与特殊方法

turtle.**mode**(*mode=None*)

参数 mode -- 字符串”standard”, ”logo” 或”world” 其中之一

设置海龟模式 (”standard”, ”logo” 或”world”) 并执行重置。如未指定模式则返回当前的模式。

”standard” 模式与旧的*turtle* 兼容。”logo” 模式与大部分 Logo 海龟绘图兼容。”world” 模式使用用户自定义的” 世界坐标系”。**注意**: 在此模式下，如果 x/y 单位比率不等于 1 则角度会显得扭曲。

模式	初始海龟朝向	正数角度
”standard”	朝右 (东)	逆时针
”logo”	朝上 (北)	顺时针

```
>>> mode("logo")   # resets turtle heading to north
>>> mode()
'logo'
```

turtle.**colormode**(*cmode=None*)

参数 cmode -- 数值 1.0 或 255 其中之一

返回颜色模式或将其设为 1.0 或 255。构成颜色三元组的 *r*, *g*, *b* 数值必须在 0..*cmode* 范围之内。

```
>>> screen.colormode(1)
>>> turtle.pencolor(240, 160, 80)
Traceback (most recent call last):
     ...
TurtleGraphicsError: bad color sequence: (240, 160, 80)
>>> screen.colormode()
1.0
>>> screen.colormode(255)
>>> screen.colormode()
255
>>> turtle.pencolor(240,160,80)
```

turtle.**getcanvas**()

返回此 TurtleScreen 的 Canvas 对象。供了解 Tkinter 的 Canvas 对象内部机理的人士使用。

```
>>> cv = screen.getcanvas()
>>> cv
<turtle.ScrolledCanvas object ...>
```

turtle.**getshapes**()

返回所有当前可用海龟形状的列表。

```
>>> screen.getshapes()
['arrow', 'blank', 'circle', ..., 'turtle']
```

turtle.**register_shape**(*name*, *shape=None*)
turtle.**addshape**(*name*, *shape=None*)

调用此函数有三种不同方式:

(1) *name* 为一个 gif 文件的文件名，*shape* 为 None: 安装相应的图像形状。:

```
>>> screen.register_shape("turtle.gif")
```

備𝔽: 当海龟转向时图像形状 不会转动，因此无法显示海龟的朝向!

(2) *name* 为指定的字符串，*shape* 为由坐标值对构成的元组: 安装相应的多边形形状。

```
>>> screen.register_shape("triangle", ((5,-3), (0,5), (-5,-3)))
```

(3) *name* 为指定的字符串，为一个 (复合) *Shape* 类对象: 安装相应的复合形状。

将一个海龟形状加入 TurtleScreen 的形状列表。只有这样注册过的形状才能通过执行 `shape(shapename)` 命令来使用。

`turtle.`**`turtles`**`()`
返回屏幕上的海龟列表。

```
>>> for turtle in screen.turtles():
...     turtle.color("red")
```

`turtle.`**`window_height`**`()`
返回海龟窗口的高度。:

```
>>> screen.window_height()
480
```

`turtle.`**`window_width`**`()`
返回海龟窗口的宽度。:

```
>>> screen.window_width()
640
```

Screen 专有方法, 而非继承自 TurtleScreen

`turtle.`**`bye`**`()`
关闭海龟绘图窗口。

`turtle.`**`exitonclick`**`()`
将 `bye()` 方法绑定到 Screen 上的鼠标点击事件。

如果配置字典中"using_IDLE" 的值为 `False` (默认值) 则同时进入主事件循环。注: 如果启动 IDLE 时使用了 `-n` 开关 (无子进程)，`turtle.cfg` 中此数值应设为 `True`。在此情况下 IDLE 本身的主事件循环同样会作用于客户脚本。

`turtle.`**`setup`**`(`*width=_CFG["width"], height=_CFG["height"], startx=_CFG["leftright"], starty=_CFG["topbottom"]*`)`
设置主窗口的大小和位置。默认参数值保存在配置字典中，可通过 `turtle.cfg` 文件进行修改。

参数

- **`width`** -- 如为一个整型数值，表示大小为多少像素，如为一个浮点数值，则表示屏幕的占比；默认为屏幕的 50%
- **`height`** -- 如为一个整型数值，表示高度为多少像素，如为一个浮点数值，则表示屏幕的占比；默认为屏幕的 75%
- **`startx`** -- 如为正值，表示初始位置距离屏幕左边缘多少像素，负值表示距离右边缘，`None` 表示窗口水平居中
- **`starty`** -- 如为正值，表示初始位置距离屏幕上边缘多少像素，负值表示距离下边缘，`None` 表示窗口垂直居中

```
>>> screen.setup (width=200, height=200, startx=0, starty=0)
>>>               # sets window to 200x200 pixels, in upper left of screen
>>> screen.setup(width=.75, height=0.5, startx=None, starty=None)
>>>               # sets window to 75% of screen by 50% of screen and centers
```

turtle.**title**(*titlestring*)

参数 titlestring -- 一个字符串，显示为海龟绘图窗口的标题栏文本

设置海龟窗口标题为 *titlestring* 指定的文本。

```
>>> screen.title("Welcome to the turtle zoo!")
```

24.1.5 公共类

class turtle.**RawTurtle**(*canvas*)
class turtle.**RawPen**(*canvas*)

参数 canvas -- 一个 tkinter.Canvas，*ScrolledCanvas* 或 *TurtleScreen* 类对象

创建一个海龟。海龟对象具有"Turtle/RawTurtle 方法" 一节所述的全部方法。

class turtle.**Turtle**
RawTurtle 的子类，具有相同的接口，但其绘图场所为默认的 *Screen* 类对象，在首次使用时自动创建。

class turtle.**TurtleScreen**(*cv*)

参数 cv -- 一个 tkinter.Canvas 类对象

提供面向屏幕的方法例如 setbg() 等。说明见上文。

class turtle.**Screen**
TurtleScreen 的子类，*增加了四个方法*.

class turtle.**ScrolledCanvas**(*master*)

参数 master -- 可容纳 ScrolledCanvas 的 Tkinter 部件，即添加了滚动条的 Tkinter-canvas

由 Screen 类使用，使其能够自动提供一个 ScrolledCanvas 作为海龟的绘图场所。

class turtle.**Shape**(*type_*, *data*)

参数 type_ -- 字符串"polygon", "image", "compound" 其中之一

实现形状的数据结构。(type_, data) 必须遵循以下定义:

*type_*	*data*
"polygon"	一个多边形元组，即由坐标值对构成的元组
"image"	一个图片 (此形式仅限内部使用!)
"compound"	None (复合形状必须使用 *addcomponent()* 方法来构建)

addcomponent(*poly*, *fill*, *outline=None*)

参数

- **poly** -- 一个多边形，即由数值对构成的元组
- **fill** -- 一种颜色，将用来填充 *poly* 指定的多边形
- **outline** -- 一种颜色，用于多边形的轮廓 (如有指定)

示例:

```
>>> poly = ((0,0),(10,-5),(0,10),(-10,-5))
>>> s = Shape("compound")
>>> s.addcomponent(poly, "red", "blue")
>>> # ... add more components and then use register_shape()
```

参见复合形状。

class turtle.**Vec2D**(*x*, *y*)

一个二维矢量类，用来作为实现海龟绘图的辅助类。也可能在海龟绘图程序中使用。派生自元组，因此矢量也属于元组!

提供的运算 (*a*, *b* 为矢量, *k* 为数值):

- `a + b` 矢量加法
- `a - b` 矢量减法
- `a * b` 内积
- `k * a` 和 `a * k` 与标量相乘
- `abs(a)` a 的绝对值
- `a.rotate(angle)` 旋转

24.1.6 帮助与配置

如何使用帮助

Screen 和 Turtle 类的公用方法以文档字符串提供了详细的文档。因此可以利用 Python 帮助工具获取这些在线帮助信息:

- 当使用 IDLE 时，输入函数/方法调用将弹出工具提示显示其签名和文档字符串的头几行。
- 对文法或函数调用*help()* 将显示其文档字符串:

```
>>> help(Screen.bgcolor)
Help on method bgcolor in module turtle:

bgcolor(self, *args) unbound turtle.Screen method
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.


      >>> screen.bgcolor("orange")
      >>> screen.bgcolor()
      "orange"
      >>> screen.bgcolor(0.5,0,0.5)
      >>> screen.bgcolor()
      "#800080"

>>> help(Turtle.penup)
Help on method penup in module turtle:

penup(self) unbound turtle.Turtle method
    Pull the pen up -- no drawing when moving.
```

(下页继续)

(繼續上一頁)

```
    Aliases: penup | pu | up

    No argument

    >>> turtle.penup()
```

- 方法对应函数的文档字符串的形式会有一些修改:

```
>>> help(bgcolor)
Help on function bgcolor in module turtle:

bgcolor(*args)
    Set or return backgroundcolor of the TurtleScreen.

    Arguments (if given): a color string or three numbers
    in the range 0..colormode or a 3-tuple of such numbers.

    Example::

      >>> bgcolor("orange")
      >>> bgcolor()
      "orange"
      >>> bgcolor(0.5,0,0.5)
      >>> bgcolor()
      "#800080"

>>> help(penup)
Help on function penup in module turtle:

penup()
    Pull the pen up -- no drawing when moving.

    Aliases: penup | pu | up

    No argument

    Example:
    >>> penup()
```

这些修改版文档字符串是在导入时与方法对应函数的定义一起自动生成的。

文档字符串翻译为不同的语言

可使用工具创建一个字典，键为方法名，值为 Screen 和 Turtle 类公共方法的文档字符串。

turtle.**write_docstringdict**(*filename="turtle_docstringdict"*)

> **参数 filename** -- 一个字符串，表示文件名
>
> 创建文档字符串字典并将其写入 filename 指定的 Python 脚本文件。此函数必须显示地调用 (海龟绘图类并不使用此函数)。文档字符串字典将被写入到 Python 脚本文件 *filename*.py。该文件可作为模板用来将文档字符串翻译为不同语言。

如果你 (或你的学生) 想使用本国语言版本的*turtle* 在线帮助，你必须翻译文档字符串并保存结果文件，例如 turtle_docstringdict_german.py.

如果你在 turtle.cfg 文件中加入了相应的条目，此字典将在导入模块时被读取并替代原有的英文版文档字符串。

在撰写本文档时已经有了德语和意大利语版的文档字符串字典。(更多需求请联系 glingl@aon.at)

如何配置 Screen 和 Turtle

内置的默认配置是模仿旧 turtle 模块的外观和行为，以便尽可能地与其保持兼容。

如果你想使用不同的配置，以便更好地反映此模块的特性或是更适合你的需求，例如在课堂中使用，你可以准备一个配置文件 `turtle.cfg`，该文件将在导入模块时被读取并根据其中的设定修改模块配置。

内置的配置对应以下的 turtle.cfg:

```
width = 0.5
height = 0.75
leftright = None
topbottom = None
canvwidth = 400
canvheight = 300
mode = standard
colormode = 1.0
delay = 10
undobuffersize = 1000
shape = classic
pencolor = black
fillcolor = black
resizemode = noresize
visible = True
language = english
exampleturtle = turtle
examplescreen = screen
title = Python Turtle Graphics
using_IDLE = False
```

选定条目的简短说明:

- 开头的四行对应 `Screen.setup()` 方法的参数。
- 第 5 和 6 行对应 `Screen.screensize()` 方法的参数。
- *shape* 可以是任何内置形状，即: arrow, turtle 等。更多信息可用 `help(shape)` 查看。
- 如果你想使用无填充色 (即令海龟变透明)，你必须写 `fillcolor = ""` (但 cfg 文件中所有非空字符串都不可加引号)。
- 如果你想令海龟反映其状态，你必须使用 `resizemode = auto`。
- 如果你设置语言例如 `language = italian` 则文档字符串字典 `turtle_docstringdict_italian.py` 将在导入模块时被加载 (如果导入路径即*turtle* 的目录中存在此文件。
- *exampleturtle* 和 *examplescreen* 条目定义了相应对象在文档字符串中显示的名称。方法文档字符串转换为函数文档字符串时将从文档字符串中删去这些名称。
- *using_IDLE*: 如果你经常使用 IDLE 并启用其 -n 开关 (" 无子进程") 则应将此项设为 `True`，这将阻止*exitonclick()* 进入主事件循环。

`turtle.cfg` 文件可以保存于*turtle* 所在目录，当前工作目录也可以有一个同名文件。后者会重载覆盖前者的设置。

`Lib/turtledemo` 目录中也有一个 `turtle.cfg` 文件。你可以将其作为示例进行研究，并在运行演示时查看其作用效果 (但最好不要在演示查看器中运行)。

24.1.7 turtledemo --- 演示脚本集

turtledemo 包汇集了一组演示脚本。这些脚本可以通过以下命令打开所提供的演示查看器运行和查看:

```
python -m turtledemo
```

此外，你也可以单独运行其中的演示脚本。例如，:

```
python -m turtledemo.bytedesign
```

turtledemo 包目录中的内容:

- 一个演示查看器 __main__.py，可用来查看脚本的源码并即时运行。
- 多个脚本文件，演示*turtle* 模块的不同特性。所有示例可通过 Examples 菜单打开。也可以单独运行每个脚本。
- 一个 turtle.cfg 文件，作为说明如何编写并使用模块配置文件的示例模板。

演示脚本清单如下:

名称	描述	相关特性
bytedesign	复杂的传统海龟绘图模式	tracer(), delay, update()
chaos	绘制 Verhulst 动态模型，演示通过计算机的运算可能会生成令人惊叹的结果	世界坐标系
clock	绘制模拟时钟显示本机的当前时间	海龟作为表针, ontimer
colormixer	试验 r, g, b 颜色模式	ondrag() 当鼠标拖动
forest	绘制 3 棵广度优先树	随机化
fractalcurves	绘制 Hilbert & Koch 曲线	递归
lindenmayer	文化数学 (印度装饰艺术)	L-系统
minimal_hanoi	汉诺塔	矩形海龟作为汉诺盘 (shape, shapesize)
nim	玩经典的“尼姆”游戏，开始时有三堆小棒，与电脑对战。	海龟作为小棒，事件驱动 (鼠标, 键盘)
paint	超极简主义绘画程序	onclick() 当鼠标点击
peace	初级技巧	海龟: 外观与动画
penrose	非周期性地使用风筝和飞镖形状铺满平面	stamp() 印章
planet_and_moon	模拟引力系统	复合开关, Vec2D 类
round_dance	两两相对并不断旋转舞蹈的海龟	复合形状, clone shapesize, tilt, get_shapepoly, update
sorting_animate	动态演示不同的排序方法	简单对齐, 随机化
tree	一棵 (图形化的) 广度优先树 (使用生成器)	clone() 克隆
two_canvases	简单设计	两块画布上的海龟
wikipedia	一个来自介绍海龟绘图的维基百科文章的图案	clone(), undo()
yinyang	另一个初级示例	circle() 画圆

祝你玩得开心!

24.1.8 Python 2.6 之后的变化

- `Turtle.tracer()`, `Turtle.window_width()` 和 `Turtle.window_height()` 方法已被去除。具有这些名称和功能的方法现在只限于 `Screen` 类的方法。但其对应的函数仍然可用。(实际上在 Python 2.6 中这些方法就已经只是从对应的 `TurtleScreen`/`Screen` 类的方法复制而来。)
- `Turtle.fill()` 方法已被去除。`begin_fill()` 和 `end_fill()` 的行为则有细微改变: 现在每个填充过程必须以一个 `end_fill()` 调用来结束。
- 新增了一个 `Turtle.filling()` 方法。该方法返回一个布尔值: 如果填充过程正在进行为 `True`，否则为 `False`。此行为相当于 Python 2.6 中不带参数的 `fill()` 调用。

24.1.9 Python 3.0 之后的变化

- 新增了 `Turtle.shearfactor()`, `Turtle.shapetransform()` 和 `Turtle.get_shapepoly()` 方法。这样就可以使用所有标准线性变换来调整海龟形状。`Turtle.tiltangle()` 的功能已被加强: 现在可被用来获取或设置倾角。`Turtle.settiltangle()` 已弃用。
- 新增了 `Screen.onkeypress()` 方法作为对 `Screen.onkey()` 的补充，实际就是将行为绑定到 keyrelease 事件。后者相应增加了一个别名: `Screen.onkeyrelease()`。
- 新增了 `Screen.mainloop()` 方法。这样当仅需使用 Screen 和 Turtle 对象时不需要再额外导入 `mainloop()`。
- 新增了两个方法 `Screen.textinput()` 和 `Screen.numinput()`。用来弹出对话框接受输入并分别返回字符串和数值。
- 两个新的示例脚本 `tdemo_nim.py` 和 `tdemo_round_dance.py` 被加入到 `Lib/turtledemo` 目录中。

24.2 cmd --- 支持面向行的命令解释器

源代码: Lib/cmd.py

Cmd 类提供简单框架用于编写面向行的命令解释器。这些通常对测试工具，管理工具和原型有用，这些工具随后将被包含在更复杂的接口中。

class cmd.**Cmd**(*completekey='tab'*, *stdin=None*, *stdout=None*)

一个*Cmd* 实例或子类实例是面向行的解释器框架结构。实例化*Cmd* 本身是没有充分理由的，它作为自定义解释器类的超类是非常有用的为了继承*Cmd* 的方法并且封装动作方法。

可选参数 *completekey* 是完成键的*readline* 名称；默认是 `Tab` 。如果 *completekey* 不是*None* 并且*readline* 是可用的，命令完成会自动完成。

可选参数 *stdin* 和 *stdout* 指定了 Cmd 实例或子类实例将用于输入和输出的输入和输出文件对象。如果没有指定，他们将默认为*sys.stdin* 和*sys.stdout* 。

如果你想要使用一个给定的 *stdin* ，确保将实例的*use_rawinput* 属性设置为 `False` ，否则 *stdin* 将被忽略。

24.2.1 Cmd 对象

Cmd 实例有下列方法：

Cmd.**cmdloop**(*intro=None*)
反复发出提示，接受输入，从收到的输入中解析出一个初始前缀，并分派给操作方法，将其余的行作为参数传递给它们。

可选参数是在第一个提示之前发布的横幅或介绍字符串（这将覆盖*intro* 类属性）。

如果*readline* 继承模块被加载，输入将自动继承类似 **bash** 的历史列表编辑（例如，Control-P 滚动回到最后一个命令，Control-N 转到下一个命令，以 Control-F 非破坏性的方式向右 Control-B 移动光标，破坏性地等）。

输入的文件结束符被作为字符串传回 `'EOF'` 。

解释器实例将会识别命令名称 `foo` 当且仅当它有方法 `do_foo()` 。有一个特殊情况，分派始于字符 `'?'` 的行到方法 `do_help()` 。另一种特殊情况，分派始于字符 `'!'` 的行到方法 `do_shell()` （如果定义了这个方法）

这个方法将返回当*postcmd()* 方法返回一个真值。参数 *stop* 到*postcmd()* 是命令对应的返回值 `do_*()` 的方法。

如果激活了完成，全部命令将会自动完成，并且通过调用 `complete_foo()` 参数 *text* , *line*, *begidx* , 和 *endidx* 完成全部命令参数。*text* 是我们试图匹配的字符串前缀，所有返回的匹配项必须以它为开头。*line* 是删除了前导空格的当前的输入行，*begidx* 和 *endidx* 是前缀文本的开始和结束索引。，可以用于根据参数位置提供不同的完成。

所有*Cmd* 的子类继承一个预定义 `do_help()` 。这个方法使用参数 `'bar'` 调用，调用对应的方法 `help_bar()` ，如果不存在，打印 `do_bar()` 的文档字符串，如果可用。没有参数的情况下，`do_help()` 方法会列出所有可用的帮助主题（即所有具有相应的 `help_*()` 方法或命令的文档字符串），也会列举所有未被记录的命令。

Cmd.**onecmd**(*str*)
解释该参数，就好像它是为响应提示而键入的一样。这可能会被覆盖，但通常不应该被覆盖; 请参阅: *precmd()* 和*postcmd()* 方法，用于执行有用的挂钩。返回值是一个标志，指示解释器对命令的解释是否应该停止。如果命令 *str* 有一个 `do_*()` 方法，则返回该方法的返回值，否则返回*default()* 方法的返回值。

Cmd.**emptyline**()
在响应提示输入空行时调用的方法。如果此方法未被覆盖，则重复输入的最后一个非空命令。

Cmd.**default**(*line*)
当命令前缀不能被识别的时候在输入行调用的方法。如果此方法未被覆盖，它将输出一个错误信息并返回。

Cmd.**completedefault**(*text*, *line*, *begidx*, *endidx*)
当没有特定于命令的 `complete_*()` 方法可用时，调用此方法完成输入行。默认情况下，它返回一个空列表。

Cmd.**precmd**(*line*)
钩方法在命令行 *line* 被解释之前执行，但是在输入提示被生成和发出后。这个方法是一个在*Cmd* 中的存根；它的存在是为了被子类覆盖。返回值被用作*onecmd()* 方法执行的命令；*precmd()* 的实现或许会重写命令或者简单的返回 *line* 不变。

Cmd.**postcmd**(*stop*, *line*)
钩方法只在命令调度完成后执行。这个方法是一个在*Cmd* 中的存根；它的存在是为了子类被覆盖。*line* 是被执行的命令行，*stop* 是一个表示在调用*postcmd()* 之后是否终止执行的标志；这将作为*onecmd()* 方法的返回值。这个方法的返回值被用作与 *stop* 相关联的内部标志的新值；返回 false 将导致解释继续。

Cmd.**preloop**()
钩方法当*cmdloop()* 被调用时执行一次。方法是一个在*Cmd* 中的存根；它的存在是为了被子类覆盖。

Cmd.**postloop**()
钩方法在*cmdloop()* 即将返回时执行一次。这个方法是一个在*Cmd* 中的存根；它的存在是为了被子类覆盖。

Instances of *Cmd* subclasses have some public instance variables:

Cmd.**prompt**
发出提示以请求输入。

Cmd.**identchars**
接受命令前缀的字符串。

Cmd.**lastcmd**
看到最后一个非空命令前缀。

Cmd.**cmdqueue**
排队的输入行列表。当需要新的输入时，在*cmdloop()* 中检查 cmdqueue 列表；如果它不是空的，它的元素将被按顺序处理，就像在提示符处输入一样。

Cmd.**intro**
要作为简介或横幅发出的字符串。可以通过给*cmdloop()* 方法一个参数来覆盖它。

Cmd.**doc_header**
如果帮助输出具有记录命令的段落，则发出头文件。

Cmd.**misc_header**
如果帮助输出其他帮助主题的部分（即与 `do_*()` 方法没有关联的 `help_*()` 方法），则发出头文件。

Cmd.**undoc_header**
如果帮助输出未被记录命令的部分（即与 `help_*()` 方法没有关联的 `do_*()` 方法），则发出头文件。

Cmd.**ruler**
用于在帮助信息标题的下方绘制分隔符的字符，如果为空，则不绘制标尺线。这个字符默认是 `'='` 。

Cmd.**use_rawinput**
这是一个标志，默认为 true 。如果为 true ，，*cmdloop()* 使用*input()* 先是提示并且阅读下一个命令；如果为 false ， `sys.stdout.write()` 和 `sys.stdin.readline()` 被使用。（这意味着解释器将会自动支持类似于 **Emacs** 的行编辑和命令历史记录按键操作，通过导入*readline* 在支持它的系统上。）

24.2.2 Cmd 例子

The *cmd* module is mainly useful for building custom shells that let a user work with a program interactively.

这部分提供了一个简单的例子来介绍如何使用一部分在*turtle* 模块中的命令构建一个 shell 。

基础的 turtle 命令比如*forward()* 被添加进一个*Cmd* 子类，方法名为 `do_forward()` 。参数被转换成数字并且分发至 turtle 模组中。docstring 是 shell 提供的帮助实用程序。

例子也包含使用*precmd()* 方法实现基础的记录和回放的功能，这个方法负责将输入转换为小写并且将命令写入文件。`do_playback()` 方法读取文件并添加记录命令至 `cmdqueue` 用于即时回放:

```
import cmd, sys
from turtle import *

class TurtleShell(cmd.Cmd):
    intro = 'Welcome to the turtle shell.   Type help or ? to list commands.\n'
    prompt = '(turtle) '
```

(下页继续)

(繼續上一頁)

```
    file = None

    # ----- basic turtle commands -----
    def do_forward(self, arg):
        'Move the turtle forward by the specified distance:  FORWARD 10'
        forward(*parse(arg))
    def do_right(self, arg):
        'Turn turtle right by given number of degrees:  RIGHT 20'
        right(*parse(arg))
    def do_left(self, arg):
        'Turn turtle left by given number of degrees:  LEFT 90'
        left(*parse(arg))
    def do_goto(self, arg):
        'Move turtle to an absolute position with changing orientation.  GOTO 100 200'
        goto(*parse(arg))
    def do_home(self, arg):
        'Return turtle to the home position:  HOME'
        home()
    def do_circle(self, arg):
        'Draw circle with given radius an options extent and steps:  CIRCLE 50'
        circle(*parse(arg))
    def do_position(self, arg):
        'Print the current turtle position:  POSITION'
        print('Current position is %d %d\n' % position())
    def do_heading(self, arg):
        'Print the current turtle heading in degrees:  HEADING'
        print('Current heading is %d\n' % (heading(),))
    def do_color(self, arg):
        'Set the color:  COLOR BLUE'
        color(arg.lower())
    def do_undo(self, arg):
        'Undo (repeatedly) the last turtle action(s):  UNDO'
    def do_reset(self, arg):
        'Clear the screen and return turtle to center:  RESET'
        reset()
    def do_bye(self, arg):
        'Stop recording, close the turtle window, and exit:  BYE'
        print('Thank you for using Turtle')
        self.close()
        bye()
        return True

    # ----- record and playback -----
    def do_record(self, arg):
        'Save future commands to filename:  RECORD rose.cmd'
        self.file = open(arg, 'w')
    def do_playback(self, arg):
        'Playback commands from a file:  PLAYBACK rose.cmd'
        self.close()
        with open(arg) as f:
            self.cmdqueue.extend(f.read().splitlines())
    def precmd(self, line):
        line = line.lower()
        if self.file and 'playback' not in line:
            print(line, file=self.file)
        return line
```

(下页继续)

(繼續上一頁)

```
    def close(self):
        if self.file:
            self.file.close()
            self.file = None

def parse(arg):
    'Convert a series of zero or more numbers to an argument tuple'
    return tuple(map(int, arg.split()))

if __name__ == '__main__':
    TurtleShell().cmdloop()
```

这是一个示例会话，其中 turtle shell 显示帮助功能，使用空行重复命令，以及简单的记录和回放功能：

```
Welcome to the turtle shell.   Type help or ? to list commands.

(turtle) ?

Documented commands (type help <topic>):
========================================
bye     color    goto     home  playback  record  right
circle  forward  heading  left  position  reset   undo

(turtle) help forward
Move the turtle forward by the specified distance:  FORWARD 10
(turtle) record spiral.cmd
(turtle) position
Current position is 0 0

(turtle) heading
Current heading is 0

(turtle) reset
(turtle) circle 20
(turtle) right 30
(turtle) circle 40
(turtle) right 30
(turtle) circle 60
(turtle) right 30
(turtle) circle 80
(turtle) right 30
(turtle) circle 100
(turtle) right 30
(turtle) circle 120
(turtle) right 30
(turtle) circle 120
(turtle) heading
Current heading is 180

(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 100
(turtle)
(turtle) right 90
(turtle) forward 400
```

(下页继续)

(繼續上一頁)

```
(turtle) right 90
(turtle) forward 500
(turtle) right 90
(turtle) forward 400
(turtle) right 90
(turtle) forward 300
(turtle) playback spiral.cmd
Current position is 0 0

Current heading is 0

Current heading is 180

(turtle) bye
Thank you for using Turtle
```

24.3 shlex ——简单的词义分析

源代码： Lib/shlex.py

---

shlex 类可用于编写类似 Unix shell 的简单词义分析程序。通常可用于编写“迷你语言”（如 Python 应用程序的运行控制文件）或解析带引号的字符串。

shlex 模块中定义了以下函数：

shlex.**split**(*s*, *comments=False*, *posix=True*)
用类似 shell 的语法拆分字符串 *s*。如果 *comments* 为*False* (默认值)，则不会解析给定字符串中的注释 (*commenters* 属性的*shlex* 实例设为空字符串)。本函数默认工作于 POSIX 模式下，但若 *posix* 参数为 False，则采用非 POSIX 模式。

備🄵： Since the *split()* function instantiates a *shlex* instance, passing `None` for *s* will read the string to split from standard input.

3.9 版後已🄵用: 在以后的 Python 版本中，给 *s* 传入 `None` 将触发异常。

shlex.**join**(*split_command*)
将列表 *split_command* 中的单词串联起来，返回一个字符串。本函数是*split()* 的逆运算。

```
>>> from shlex import join
>>> print(join(['echo', '-n', 'Multiple words']))
echo -n 'Multiple words'
```

为防止注入漏洞，返回值是经过 shell 转义的（参见*quote()* ）。

3.8 版新加入.

shlex.**quote**(*s*)
返回经过 shell 转义的字符串 *s* 。返回值为字符串，可以安全地作为 shell 命令行中的单词使用，可用于不能使用列表的场合。

以下用法是不安全的：

```
>>> filename = 'somefile; rm -rf ~'
>>> command = 'ls -l {}'.format(filename)
>>> print(command)  # executed by a shell: boom!
ls -l somefile; rm -rf ~
```

用*quote()* 可以堵住这种安全漏洞：

```
>>> from shlex import quote
>>> command = 'ls -l {}'.format(quote(filename))
>>> print(command)
ls -l 'somefile; rm -rf ~'
>>> remote_command = 'ssh home {}'.format(quote(command))
>>> print(remote_command)
ssh home 'ls -l '"'"'somefile; rm -rf ~'"'"''
```

这种包装方式兼容于 UNIX shell 和*split()* 。

```
>>> from shlex import split
>>> remote_command = split(remote_command)
>>> remote_command
['ssh', 'home', "ls -l 'somefile; rm -rf ~'"]
>>> command = split(remote_command[-1])
>>> command
['ls', '-l', 'somefile; rm -rf ~']
```

3.3 版新加入.

shlex 模块中定义了以下类：

class shlex.**shlex**(*instream=None*, *infile=None*, *posix=False*, *punctuation_chars=False*)

shlex 及其子类的实例是一种词义分析器对象。利用初始化参数可指定从哪里读取字符。初始化参数必须是具备*read()* 和*readline()* 方法的文件/流对象，或者是一个字符串。如果没有给出初始化参数，则会从 sys.stdin 获取输入。第二个可选参数是个文件名字符串，用于设置*infile* 属性的初始值。如果 *instream* 参数被省略或等于 sys.stdin，则第二个参数默认为"stdin"。*posix* 参数定义了操作的模式：若 *posix* 不为真值（默认），则*shlex* 实例将工作于兼容模式。若运行于 POSIX 模式下，则*shlex* 会尽可能地应用 POSIX shell 解析规则。*punctuation_chars* 参数提供了一种使行为更接近于真正的 shell 解析的方式。该参数可接受多种值：默认值、False、保持 Python 3.5 及更早版本的行为。如果设为 True，则会改变对字符 ();<>|& 的解析方式：这些字符将作为独立的形符被返回（视作标点符号）。如果设为非空字符串，则这些字符将被用作标点符号。出现在 *punctuation_chars* 中的*wordchars* 属性中的任何字符都会从*wordchars* 中被删除。请参阅*改进的 shell 兼容性* 了解详情。*punctuation_chars* 只能在创建*shlex* 实例时设置，以后不能再作修改。

3.6 版更變: 加入 *punctuation_chars* 参数。

也参考:

***configparser* 模块** 配置文件解析器，类似于 Windows 的 .ini 文件。

24.3.1 shlex 对象

shlex 实例具备以下方法：

shlex.**get_token**()
返回一个单词。如果所有单词已用*push_token()* 堆叠在一起了，则从堆中弹出一个单词。否则就从输入流中读取一个。如果读取时遇到文件结束符，则会返回 `eof`（在非 POSIX 模式下为空字符串 `""`，在 POSIX 模式下为 ``None`）。

shlex.**push_token**(*str*)
将参数值压入单词堆栈。

shlex.**read_token**()
读取一个原单词。忽略堆栈，且不解释源请求。（通常没什么用，只是为了完整起见。）

shlex.**sourcehook**(*filename*)
当*shlex* 检测到源请求（见下面的*source*），本方法被赋予以下单词作为参数，并应返回一个由文件名和打开的文件对象组成的元组。

通常本方法会先移除参数中的引号。如果结果为绝对路径名，或者之前没有有效的源请求，或者之前的源请求是一个流对象（比如 `sys.stdin`），那么结果将不做处理。否则，如果结果是相对路径名，那么前面将会加上目录部分，目录名来自于源堆栈中前一个文件名（类似于 C 预处理器对 `#include "file.h"` 的处理方式）。

结果被视为一个文件名，并作为元组的第一部分返回，元组的第二部分以此为基础调用*open()* 获得。（注意：这与实例初始化中的参数顺序相反！）

此钩子函数是公开的，可用于实现路径搜索、添加文件扩展名或黑入其他命名空间。没有对应的“关闭”钩子函数，但 shlex 实例在返回 EOF 时会调用源输入流的*close()* 方法。

若要更明确地控制源堆栈，请采用*push_source()* 和*pop_source()* 方法。

shlex.**push_source**(*newstream*, *newfile=None*)
将输入源流压入输入堆栈。如果指定了文件名参数，以后错误信息中将会用到。这与*sourcehook()* 内部使用的方法相同。

shlex.**pop_source**()
从输入堆栈中弹出最后一条输入源。当遇到输入流的 EOF 时，内部也使用同一方法。

shlex.**error_leader**(*infile=None*, *lineno=None*)
本方法生成一条错误信息的首部，以 Unix C 编译器错误标签的形式；格式为 “"%s", line %d: ’，其中 ``%s` 被替换为当前源文件的名称，`%d` 被替换为当前输入行号（可用可选参数覆盖）。

这是个快捷函数，旨在鼓励*shlex* 用户以标准的、可解析的格式生成错误信息，以便 Emacs 和其他 Unix 工具理解。

shlex 子类的实例有一些公共实例变量，这些变量可以控制词义分析，也可用于调试。

shlex.**commenters**
将被视为注释起始字符串。从注释起始字符串到行尾的所有字符都将被忽略。默认情况下只包括 `'#'`。

shlex.**wordchars**
可连成多字符单词的字符串。默认包含所有 ASCII 字母数字和下划线。在 POSIX 模式下，Latin-1 字符集的重音字符也被包括在内。如果*punctuation_chars* 不为空，则可出现在文件名规范和命令行参数中的 `~-./*?=` 字符也将包含在内，任何 `punctuation_chars` 中的字符将从 `wordchars` 中移除。如果*whitespace_split* 设为 `True`，则本规则无效。

shlex.**whitespace**
将被视为空白符并跳过的字符。空白符是单词的边界。默认包含空格、制表符、换行符和回车符。

shlex.**escape**
将视为转义字符。仅适用于 POSIX 模式，默认只包含 `'\'`。

shlex.**quotes**
将视为引号的字符。单词中的字符将会累至再次遇到同样的引号（因此，不同的引号会像在 shell 中一样相互包含。）默认包含 ASCII 单引号和双引号。

shlex.**escapedquotes**
quotes 中的字符将会解析*escape* 定义的转义字符。这只在 POSIX 模式下使用，默认只包含 `'"'`。

shlex.**whitespace_split**
若为 `True`，则只根据空白符拆分单词。这很有用，比如用*shlex* 解析命令行，用类似 shell 参数的方式读取各个单词。当与*punctuation_chars* 一起使用时，将根据空白符和这些字符拆分单词。

3.8 版更變: *punctuation_chars* 属性已与*whitespace_split* 属性兼容。

shlex.**infile**
当前输入的文件名，可能是在类实例化时设置的，或者是由后来的源请求堆栈生成的。在构建错误信息时可能会用到本属性。

shlex.**instream**
shlex 实例正从中读取字符的输入流。

shlex.**source**
本属性默认值为 `None`。如果给定一个字符串，则会识别为包含请求，类似于各种 shell 中的 `source` 关键字。也就是说，紧随其后的单词将作为文件名打开，作为输入流，直至遇到 EOF 后调用流的*close()* 方法，然后原输入流仍变回输入源。Source 请求可以在词义堆栈中嵌套任意深度。

shlex.**debug**
如果本属性为大于 `1` 的数字，则*shlex* 实例会把动作进度详细地输出出来。若需用到本属性，可阅读源代码来了解细节。

shlex.**lineno**
源的行数（到目前为止读到的换行符数量加 1）。

shlex.**token**
单词缓冲区。在捕获异常时可能会用到。

shlex.**eof**
用于确定文件结束的单词。在非 POSIX 模式下，将设为空字符串 `''`，在 POSIX 模式下被设为 `None`。

shlex.**punctuation_chars**
只读属性。表示应视作标点符号的字符。标点符号将作为单个单词返回。然而，请注意不会进行语义有效性检查：比如“»>”可能会作为一个单词返回，虽然 shell 可能无法识别。

3.6 版新加入.

24.3.2 解析规则

在非 POSIX 模式下时，*shlex* 会试图遵守以下规则：

- 不识别单词中的引号（`Do"Not"Separate` 解析为一个单词 `Do"Not"Separate`）。
- 不识别转义字符；
- 引号包裹的字符保留字面意思。
- 成对的引号会将单词分离（`"Do"Separate` 解析为 `"Do"` 和 `Separate`）。
- 如果*whitespace_split* 为 `False`，则未声明为单词字符、空白或引号的字符将作为单字符标记返回。若为 `True`，则*shlex* 只根据空白符拆分单词。
- EOF 用空字符串（`''`）表示；
- 空字符串无法解析，即便是加了引号。

在 POSIX 模式时，*shlex* 将尝试遵守以下解析规则：

- 引号会被剔除，且不会拆分单词（`"Do"Not"Separate"` 将解析为单个单词 `DoNotSeparate`）。
- 未加引号包裹的转义字符（如 `'\'`）保留后一个字符的字面意思；
- 引号中的字符不属于*escapedquotes*（例如，`"'"`），则保留引号中所有字符的字面值。
- 若引号包裹的字符属于*escapedquotes*（例如 `'"'`），则保留引号中所有字符的字面意思，属于*escape* 中的字符除外。仅当后跟后半个引号或转义字符本身时，转义字符才保留其特殊含义。否则，转义字符将视作普通字符。
- EOF 用*None* 表示。
- 允许出现引号包裹的空字符串（`''`）。

24.3.3 改进的 shell 兼容性

3.6 版新加入.

shlex 类提供了与常见 Unix shell（如 `bash`、`dash` 和 “sh“）的解析兼容性。为了充分利用这种兼容性，请在构造函数中设定 `punctuation_chars` 参数。该参数默认为 `False`，维持 3.6 以下版本的行为。如果设为 `True`，则会改变对 `();<>|&` 字符的解析方式：这些字符都将视为单个标记返回。虽然不算是完整的 shell 解析程序（考虑到 shell 的多样性，超出了标准库的范围），但确实能比其他方式更容易进行命令行的处理。以下代码段演示了两者的差异：

```
>>> import shlex
>>> text = "a && b; c && d || e; f >'abc'; (def \"ghi\")"
>>> s = shlex.shlex(text, posix=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b;', 'c', '&&', 'd', '||', 'e;', 'f', '>abc;', '(def', 'ghi)']
>>> s = shlex.shlex(text, posix=True, punctuation_chars=True)
>>> s.whitespace_split = True
>>> list(s)
['a', '&&', 'b', ';', 'c', '&&', 'd', '||', 'e', ';', 'f', '>', 'abc', ';',
'(', 'def', 'ghi', ')']
```

当然，返回的词法单元对 shell 无效，需要对返回的词法单元自行进行错误检查。

punctuation_chars 参数可以不传入 `True`，而是传入包含特定字符的字符串，用于确定由哪些字符构成标点符号。例如：

```
>>> import shlex
>>> s = shlex.shlex("a && b || c", punctuation_chars="|")
>>> list(s)
['a', '&', '&', 'b', '||', 'c']
```

備𝔼: 如果指定了 `punctuation_chars`，则*wordchars* 属性的参数会是 `~-./*?=`。因为这些字符可以出现在文件名（包括通配符）和命令行参数中（如 `--color=auto`）。因此：

```
>>> import shlex
>>> s = shlex.shlex('~/a && b-c --color=auto || d *.py?',
...                 punctuation_chars=True)
>>> list(s)
['~/a', '&&', 'b-c', '--color=auto', '||', 'd', '*.py?']
```

不过为了尽可能接近于 shell，建议在使用*punctuation_chars*时始终使用 posix 和*whitespace_split*，这将完全否定*wordchars*。

为了达到最佳效果，punctuation_chars 应与 posix=True 一起设置。（注意 posix=False 是*shlex*的默认设置）。

CHAPTER 25

以 Tk 打造 GUI

Tcl/Tk 集成到 Python 中已经有一些年头了。Python 程序员可以通过*tkinter* 包和它的扩展，*tkinter.tix* 模块和*tkinter.ttk* 模块，来使用这套鲁棒的、平台无关的窗口工具集。

tkinter 包是使用面向对象方式对 Tcl/Tk 进行的一层薄包装。使用*tkinter*，你不需要写 Tcl 代码，但你将需要参阅 Tk 文档，有时还需要参阅 Tcl 文档。*tkinter* 是一组包装器，它将 Tk 的可视化部件实现为相应的 Python 类。

tkinter 的主要特点是速度很快，并且通常直接附带在 Python 中。虽然它的官方文档做得不好，但还是有许多可用的资源，包括：在线参考、教程、入门书等等。*tkinter* 还有众所周知的较过时的外观界面，这在 Tk 8.5 中已得到很大改进。无论如何，你还可以考虑许多其他的 GUI 库。Python wiki 例出了一些替代性的 GUI 框架和工具。

25.1 `tkinter` --- Tcl/Tk 的 Python 接口

源代码: Lib/tkinter/__init__.py

The *tkinter* package (”Tk interface”) is the standard Python interface to the Tcl/Tk GUI toolkit. Both Tk and *tkinter* are available on most Unix platforms, including macOS, as well as on Windows systems.

在命令行中运行 `python -m tkinter`，应该会弹出一个 Tk 界面的窗口，表明*tkinter* 包已经正确安装，而且告诉你 Tcl/Tk 的版本号，通过这个版本号，你就可以参考对应的 Tcl/Tk 文档了。

也参考:

- **TkDocs** Extensive tutorial on creating user interfaces with Tkinter. Explains key concepts, and illustrates recommended approaches using the modern API.
- **Tkinter 8.5 参考：一个 Python 图形用户界面** Reference documentation for Tkinter 8.5 detailing available classes, methods, and options.

Tcl/Tk Resources:

- **Tk 命令** Comprehensive reference to each of the underlying Tcl/Tk commands used by Tkinter.

- **Tcl/Tk Home Page** Additional documentation, and links to Tcl/Tk core development.

Books:

- **Modern Tkinter for Busy Python Developers** By Mark Roseman. (ISBN 978-1999149567)
- **Python and Tkinter Programming** By Alan Moore. (ISBN 978-1788835886)
- **使用 Python 编程** By Mark Lutz; has excellent coverage of Tkinter. (ISBN 978-0596158101)
- **Tcl and the Tk Toolkit (2nd edition)** By John Ousterhout, inventor of Tcl/Tk, and Ken Jones; does not cover Tkinter. (ISBN 978-0321336330)

25.1.1 Tkinter 模块

Support for Tkinter is spread across several modules. Most applications will need the main *tkinter* module, as well as the *tkinter.ttk* module, which provides the modern themed widget set and API:

```
from tkinter import *
from tkinter import ttk
```

class tkinter.**Tk**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=True*, *sync=False*, *use=None*)

Construct a toplevel Tk widget, which is usually the main window of an application, and initialize a Tcl interpreter for this widget. Each instance has its own associated Tcl interpreter.

The *Tk* class is typically instantiated using all default values. However, the following keyword arguments are currently recognized:

screenName When given (as a string), sets the DISPLAY environment variable. (X11 only)

baseName Name of the profile file. By default, *baseName* is derived from the program name (sys.argv[0]).

className Name of the widget class. Used as a profile file and also as the name with which Tcl is invoked (*argv0* in *interp*).

useTk If True, initialize the Tk subsystem. The *tkinter.Tcl()* function sets this to False.

sync If True, execute all X server commands synchronously, so that errors are reported immediately. Can be used for debugging. (X11 only)

use Specifies the *id* of the window in which to embed the application, instead of it being created as an independent toplevel window. *id* must be specified in the same way as the value for the -use option for toplevel widgets (that is, it has a form like that returned by winfo_id()).

Note that on some platforms this will only work correctly if *id* refers to a Tk frame or toplevel that has its -container option enabled.

Tk reads and interprets profile files, named *.className*.tcl and *.baseName*.tcl, into the Tcl interpreter and calls *exec()* on the contents of *.className*.py and *.baseName*.py. The path for the profile files is the HOME environment variable or, if that isn't defined, then *os.curdir*.

tk

The Tk application object created by instantiating *Tk*. This provides access to the Tcl interpreter. Each widget that is attached the same instance of *Tk* has the same value for its *tk* attribute.

master

The widget object that contains this widget. For *Tk*, the *master* is *None* because it is the main window. The terms *master* and *parent* are similar and sometimes used interchangeably as argument names; however, calling winfo_parent() returns a string of the widget name whereas *master* returns the object. *parent*/*child* reflects the tree-like relationship while *master*/*slave* reflects the container structure.

children
The immediate descendants of this widget as a *dict* with the child widget names as the keys and the child instance objects as the values.

tkinter.**Tcl**(*screenName=None*, *baseName=None*, *className='Tk'*, *useTk=False*)
Tcl() 函数是一个工厂函数，它创建的对象与*Tk* 类创建的对象非常相似，只是它不初始化 Tk 子系统。在不想创建或无法创建（如没有 X Server 的 Unix/Linux 系统）额外的顶层窗口的环境中驱动 Tcl 解释器时，这一点非常有用。由*Tcl()* 对象创建的对象可以通过调用其 `loadtk()` 方法来创建顶层窗口（并初始化 Tk 子系统）。

The modules that provide Tk support include:

tkinter Main Tkinter module.

tkinter.colorchooser 让用户选择颜色的对话框。

tkinter.commondialog 在此处列出的其他模块中定义的对话框的基类。

tkinter.filedialog 允许用户指定文件的通用对话框，用于打开或保存文件。

tkinter.font 帮助操作字体的工具。

tkinter.messagebox 访问标准的 Tk 对话框。

tkinter.scrolledtext 内置纵向滚动条的文本组件。

tkinter.simpledialog 基础对话框和便捷功能。

tkinter.ttk Themed widget set introduced in Tk 8.5, providing modern alternatives for many of the classic widgets in the main *tkinter* module.

Additional modules:

_tkinter A binary module that contains the low-level interface to Tcl/Tk. It is automatically imported by the main *tkinter* module, and should never be used directly by application programmers. It is usually a shared library (or DLL), but might in some cases be statically linked with the Python interpreter.

idlelib Python's Integrated Development and Learning Environment (IDLE). Based on *tkinter*.

tkinter.constants Symbolic constants that can be used in place of strings when passing various parameters to Tkinter calls. Automatically imported by the main *tkinter* module.

tkinter.dnd (experimental) Drag-and-drop support for *tkinter*. This will become deprecated when it is replaced with the Tk DND.

tkinter.tix (deprecated) An older third-party Tcl/Tk package that adds several new widgets. Better alternatives for most can be found in *tkinter.ttk*.

turtle Tk 窗口中的海龟绘图库。

25.1.2 Tkinter 拾遗

本节不应作为 Tk 或 Tkinter 的详尽教程。而只是一个补充，提供一些系统指南。

致谢：

- Tk 是 John Ousterhout 在伯克利大学时写的。
- Tkinter 是由 Steen Lumholt 和 Guido van Rossum 编写的。
- 本拾遗由弗吉尼亚大学的 Matt Conway 撰写。
- HTML 渲染和一些自由编辑功能，是由 Ken Manheimer 根据 FrameMaker 创建的。
- Fredrik Lundh 认真研究并修改了类的接口描述，使其与 Tk 4.2 保持一致。

- Mike Clarkson 将文档转换为 LaTeX 格式，并编写了参考手册的用户界面章节。

本节内容的用法

本节分为两部分：前半部分（大致）涵盖了背景材料，后半部分可以作为手头的参考手册。

当试图回答“如何才能怎么怎么”这种问题时，通常最好是弄清楚如何直接在 Tk 中实现，然后转换回相应的*tkinter*调用。Python 程序员通常可查看 Tk 文档来猜测正确的 Python 命令。这意味着要用好 Tkinter，必须对 Tk 有一定的了解。本文无法完成这个任务，所以最好的做法就是给出目前最好的文档。下面给出一些小提示：

- 作者强烈建议用户先拿到一份 Tk 手册。具体来说，`manN` 目录下的 man 文档是最有用的。`man3` 的 man 文档描述了 Tk 库的 C 接口，因此对脚本编写人员没有什么特别的帮助。
- Addison-Wesley 出版了一本名为《Tcl 和 Tk 工具包》的书，作者是 John Ousterhout（ISBN 0-201-63337-X），对于新手来说，这是一本很好的 Tcl 和 Tk 介绍书籍。该书不算详尽，很多细节还是要看 man 手册。
- 对大多数人而言，`tkinter/__init__.py` 是最后一招，但其他手段都无效时也不失为一个好去处。

简单的 Hello World 程序

```
import tkinter as tk

class Application(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def create_widgets(self):
        self.hi_there = tk.Button(self)
        self.hi_there["text"] = "Hello World\n(click me)"
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack(side="top")

        self.quit = tk.Button(self, text="QUIT", fg="red",
                              command=self.master.destroy)
        self.quit.pack(side="bottom")

    def say_hi(self):
        print("hi there, everyone!")

root = tk.Tk()
app = Application(master=root)
app.mainloop()
```

25.1.3 Tcl/Tk 速览

类的层次结构看起来很复杂，但在实际操作中，应用程序编写人员几乎总是查看最底层的类。

F解：

- 提供这些类，是为了能在同一个命名空间下将某些功能组织在一起。并不意味着可以独立对其进行实例化。
- *Tk* 类在同一个应用程序中仅需作一次实例化。应用程序编程人员不需要显式进行实例化，只要有其他任何类被实例化，系统就会创建一个。
- `Widget` 类不是用来实例化的，它仅用于继承以便生成“真正”的部件（C++ 中称为“抽象类”）。

若要充分利用这些参考资料，有时需要知道如何阅读 Tk 的短文，以及如何识别 Tk 命令的各个部分。（参见*将简单的 Tk 映射到 Tkinter* 一节，了解以下内容在`tkinter` 中的对应部分）。

Tk 脚本就是 Tcl 程序。像所有其他的 Tcl 程序一样，Tk 脚本只是由空格分隔的单词列表。一个 Tk 部件只是它的 *class*，*options* 用于进行配置，*actions* 让它执行有用的动作。

要在 Tk 中制作一个部件，总是采用如下格式的命令：

```
classCommand newPathname options
```

classCommand 表示要制作何种部件（按钮、标签、菜单.....）。

newPathname 该组件的新名字。Tk 中的所有名字都必须唯一。为了帮助实现这一点，Tk 中的部件都用 路径命名，就像文件系统中的文件一样。顶层的部件，即 根，名为 `.`` （句点），而子部件则由更多的句点分隔。比如部件名可能会是 ``.myApp.controlPanel.okButton 。

options 配置部件的外观，有时也能配置行为。这些选项以标志和值的列表形式出现。标志前带一个 “-”，就像 Unix shell 命令的标志一样，如果值超过一个单词，就用引号括起来。

例如：

```
button    .fred    -fg red -text "hi there"
   ^        ^      \_____________________/
   |        |                 |
 class     new              options
command   widget  (-opt val -opt val ...)
```

一旦创建成功，部件的路径名就成了一条新的命令。这个新的 部件命令是程序员让新部件执行某些 *action* 的句柄。在 C 语言中可表示为 someAction(fred, someOptions)，在 C++ 中可表示为 fred.someAction(someOptions)，而在 Tk 中写作：

```
.fred someAction someOptions
```

请注意，对象名 `.fred` 是以句点开头的。

如您所料，*someAction* 的可用值取决于部件的类别。如果 fred 为按钮，则 *.fred disable* 有效（fred 会变灰），而当 fred 为标签时则无效（Tk 不支持标签的禁用）。

someOptions 的合法值取决于动作。有些动作不需要参数，比如 `disable`，其他动作如文本输入框的 `delete` 命令则需用参数指定要删除的文本范围。

25.1.4 将简单的 Tk 映射到 Tkinter

Tk 中的类命令对应于 Tkinter 中的类构造函数：

```
button .fred                =====>  fred = Button()
```

父对象是隐含在创建时给定的新名字中的。在 Tkinter 中，父对象是显式指定的：

```
button .panel.fred          =====>  fred = Button(panel)
```

Tk 中的配置项是以连字符标签列表的形式给出的，后跟着参数值。在 Tkinter 中，选项指定为实例构造函数中的关键字参数，在配置调用时指定为关键字 args，或在已有实例中指定为实例索引，以字典的形式。参见可选配置项 一节的选项设置部分。

```
button .fred -fg red        =====>  fred = Button(panel, fg="red")
.fred configure -fg red     =====>  fred["fg"] = red
                            OR ==>  fred.config(fg="red")
```

在 Tk 中, 要在某个部件上执行动作, 要用部件名作为命令, 并后面附上动作名称, 可能还会带有参数（option）。在 Tkinter 中，则调用类实例的方法来执行部件的动作。部件能够执行的动作（方法）列在 `tkinter/__init__.py` 中。

```
.fred invoke                =====>  fred.invoke()
```

若要将部件交给打包器（geometry manager），需带上可选参数去调用 pack。在 Tkinter 中，Pack 类拥有全部这些功能，pack 命令的各种形式都以方法的形式实现。*tkinter* 中的所有部件都是从 Packer 继承而来的，因此继承了所有打包方法。关于 Form geometry manager 的更多信息，请参见*tkinter.tix* 模块的文档。

```
pack .fred -side left       =====>  fred.pack(side="left")
```

25.1.5 Tk 和 Tkinter 如何关联

自上而下：

（Python）应用程序位于这里 Python 应用程序进行了*tkinter* 调用。

tkinter （Python 包） 此调用（比如创建一个按钮部件）在*tkinter* 包中实现，它是用 Python 编写的。该 Python 函数将解析命令和参数，并转换为貌似 Tk 脚本的格式，而不像是 Python 脚本。

_tkinter（C 语言） 这些命令及参数将被传给 `_tkinter` 扩展模块中的 C 函数——注意下划线。

Tk 部件（C 和 Tcl） 该 C 函数能调用其他的 C 模块，包括构成 Tk 库的 C 函数。Tk 是用 C 和一些 Tcl 实现的。Tk 部件的 Tcl 部分用于将某些默认行为绑定到部件上，并在导入 Python *tkinter* 包时执行一次。（该阶段对用户而言永不可见）。

Tk（C 语言） Tk 部件的 Tk 语言部分实现了最终的映射

Xlib（C 语言） Xlib 库在屏幕上绘制图形。

25.1.6 快速参考

可选配置项

配置参数可以控制组件颜色和边框宽度等。可通过三种方式进行设置:

在对象创建时，使用关键字参数

```
fred = Button(self, fg="red", bg="blue")
```

在对象创建后，将参数名用作字典索引

```
fred["fg"] = "red"
fred["bg"] = "blue"
```

利用 config() 方法修改对象的属性

```
fred.config(fg="red", bg="blue")
```

关于这些参数及其表现的完整解释，请参阅 Tk 手册中有关组件的 man 帮助页。

请注意，man 手册页列出了每个部件的“标准选项”和“组件特有选项”。前者是很多组件通用的选项列表，后者是该组件特有的选项。标准选项在 *options(3)* man 手册中有文档。

本文没有区分标准选项和部件特有选项。有些选项不适用于某类组件。组件是否对某选项做出响应，取决于组件的类别；按钮组件有一个 `command` 选项，而标签组件就没有。

组件支持的选项在其手册中有列出，也可在运行时调用 `config()` 方法（不带参数）查看，或者通过调用组件的 `keys()` 方法进行查询。这些调用的返回值为字典，字典的键是字符串格式的选项名（比如 `'relief'`），字典的值为五元组。

有些选项，比如 `bg` 是全名通用选项的同义词（`bg` 是“background”的简写）。向 `config()` 方法传入选项的简称将返回一个二元组，而不是五元组。传回的二元组将包含同义词的全名和“真正的”选项（比如 `('bg', 'background')`）。

索引	意义	示例
0	选项名称	`'relief'`
1	数据库查找的选项名称	`'relief'`
2	数据库查找的选项类	`'Relief'`
3	默认值	`'raised'`
4	当前值	`'groove'`

示例:

```
>>> print(fred.config())
{'relief': ('relief', 'relief', 'Relief', 'raised', 'groove')}
```

当然，输出的字典将包含所有可用选项及其值。这里只是举个例子。

包装器

包装器是 Tk 的形状管理机制之一。形状管理器用于指定部件在容器内的相对位置——彼此的 *宿主关系*。与更为麻烦的 *定位器* 相比（不太常用，这里不做介绍），包装器可接受定性的相对关系——*上面*、*左边*、*填充* 等，并确定精确的放置坐标。

主部件 的大小都由其内部的“从属部件”的大小决定。包装器用于控制从属部件在主部件中出现的位置。可以把部件包入框架，再把框架包入其他框架中，搭建出所需的布局。此外，只要完成了包装，组件的布局就会进行动态调整，以适应布局参数的变化。

请注意，只有用形状管理器指定几何形状后，部件才会显示出来。忘记设置形状参数是新手常犯的错误，惊讶于创建完部件却啥都没出现。部件只有在应用了类似于打包器的 `pack()` 方法之后才会显示在屏幕上。

调用 pack() 方法时可以给出由关键字/参数值组成的键值对，以便控制组件在其容器中出现的位置，以及主程序窗口大小变动时的行为。下面是一些例子：

```
fred.pack()                     # defaults to side = "top"
fred.pack(side="left")
fred.pack(expand=1)
```

包装器的参数

关于包装器及其可接受的参数，更多信息请参阅 man 手册和 John Ousterhout 书中的第 183 页。

anchor 锚点类型。表示包装器要放置的每个从属组件的位置。

expand 布尔型，`0` 或 `1`。

fill 合法值为：`'x'`、`'y'`、`'both'`、`'none'`。

ipadx 和 ipady 距离值，指定从属部件每一侧的内边距。

padx 和 pady 距离值，指定从属部件的外边距。

side 合法值为：`'left'`、`'right'`、`'top'`、`'bottom'`。

部件与变量的关联

通过一些特定参数，某些组件（如文本输入组件）的当前设置可直接与应用程序的变量关联。这些参数包括 `variable`、`textvariable`、`onvalue`、`offvalue`、`value`。这种关联是双向的：只要这些变量因任何原因发生变化，其关联的部件就会更新以反映新的参数值。

不幸的是，在目前 *tkinter* 的实现代码中，不可能通过 `variable` 或 `textvariable` 参数将任意 Python 变量移交给组件。变量只有是 *tkinter* 中定义的 Variable 类的子类，才能生效。

已经定义了很多有用的 Variable 子类：`StringVar`、`IntVar`、`DoubleVar` 和 `BooleanVar`。调用 `get()` 方法可以读取这些变量的当前值；调用 `set()` 方法则可改变变量值。只要遵循这种用法，组件就会保持跟踪变量的值，而不需要更多的干预。

例如：

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master):
        super().__init__(master)
        self.pack()
```

(下页继续)

(繼續上一頁)

```
        self.entrythingy = tk.Entry()
        self.entrythingy.pack()

        # Create the application variable.
        self.contents = tk.StringVar()
        # Set it to some value.
        self.contents.set("this is a variable")
        # Tell the entry widget to watch this variable.
        self.entrythingy["textvariable"] = self.contents

        # Define a callback for when the user hits return.
        # It prints the current value of the variable.
        self.entrythingy.bind('<Key-Return>',
                              self.print_contents)

    def print_contents(self, event):
        print("Hi. The current entry content is:",
              self.contents.get())

root = tk.Tk()
myapp = App(root)
myapp.mainloop()
```

窗口管理器

Tk 有个实用命令 `wm`，用于与窗口管理器进行交互。`wm` 命令的参数可用于控制标题、位置、图标之类的东西。在*tkinter* 中，这些命令已被实现为 `Wm` 类的方法。顶层部件是 `Wm` 类的子类，所以可以直接调用 `Wm` 的这些方法。

要获得指定部件所在的顶层窗口，通常只要引用该部件的主窗口即可。当然，如果该部件是包装在框架内的，那么主窗口不代表就是顶层窗口。为了获得任意组件所在的顶层窗口，可以调用 `_root()` 方法。该方法以下划线开头，表明其为 Python 实现的代码，而非 Tk 提供的某个接口。

以下是一些典型用法：

```
import tkinter as tk

class App(tk.Frame):
    def __init__(self, master=None):
        super().__init__(master)
        self.pack()

# create the application
myapp = App()

#
# here are method calls to the window manager class
#
myapp.master.title("My Do-Nothing Application")
myapp.master.maxsize(1000, 400)

# start the program
myapp.mainloop()
```

Tk 参数的数据类型

anchor 合法值是罗盘的方位点：`"n"`、`"ne"`、`"e"`、`"se"`、`"s"`、`"sw"`、`"w"`、`"nw"` 和 `"center"`

位图 内置已命名的位图有八个：`'error'`、`'gray25'`、`'gray50'`、`'hourglass'`、`'info'`、`'questhead'`、`'question'`、`'warning'`。若要指定位图的文件名，请给出完整路径，前面加一个 `@`，比如 `"@/usr/contrib/bitmap/gumby.bit"`。

布尔值 可以传入整数 0 或 1，或是字符串 `"yes"` 或 `"no"`。

callback -- 回调 指任何无需调用参数的 Python 函数。例如：

```
def print_it():
    print("hi there")
fred["command"] = print_it
```

color Colors can be given as the names of X colors in the rgb.txt file, or as strings representing RGB values in 4 bit: `"#RGB"`, 8 bit: `"#RRGGBB"`, 12 bit: `"#RRRGGGBBB"`, or 16 bit: `"#RRRRGGGGBBBB"` ranges, where R,G,B here represent any legal hex digit. See page 160 of Ousterhout's book for details.

cursor 可采用 `cursorfont.h` 中的标准光标名称，去掉 `XC_` 前缀。比如要获取一个手形光标（`XC_hand2`），可以用字符串 `"hand2"`。也可以指定自己的位图和掩码文件作为光标。参见 Ousterhout 书中的第 179 页。

distance 屏幕距离可以用像素或绝对距离来指定。像素是数字，绝对距离是字符串，后面的字符表示单位：`c` 是厘米，`i` 是英寸，`m` 是毫米，`p` 则表示打印机的点数。例如，3.5 英寸可表示为 `"3.5i"`。

font Tk 采用一串名称的格式表示字体，例如 `{courier 10 bold}`。正数的字体大小以点为单位，负数的大小以像素为单位。

geometry 这是一个 `widthxheight` 形式的字符串，其中宽度和高度对于大多数部件来说是以像素为单位的（对于显示文本的部件来说是以字符为单位的）。例如：fred["geometry"] = "200x100"。

justify 合法的值为字符串：`"left"`、`"center"`、`"right"` 和 `"fill"`。

region 这是包含四个元素的字符串，以空格分隔，每个元素是表示一个合法的距离值（见上文）。例如：`"2 3 4 5"`、`"3i 2i 4.5i 2i"` 和 `"3c 2c 4c 10.43c"` 都是合法的区域值。

relief 决定了组件的边框样式。合法值包括：`"raised"`、`"sunken"`、`"flat"`、`"groove"` 和 `"ridge"`。

scrollcommand 这几乎就是带滚动条部件的 `set()` 方法，但也可是任一只有一个参数的部件方法。

wrap 只能是以下值之一：`"none"`、`"char"`、`"word"`。

绑定和事件

部件命令中的 bind 方法可觉察某些事件，并在事件发生时触发一个回调函数。bind 方法的形式是：

```
def bind(self, sequence, func, add=''):
```

这里：

序列 是一个表示事件的目标种类的字符串。（详情请看 bind 的手册页和 John Outsterhout 的书的第 201 页。）

func 是带有一个参数的 Python 函数，发生事件时将会调用。传入的参数为一个 Event 实例。（以这种方式部署的函数通常称为 *回调函数*。）

add 可选项，`''` 或 `'+'`。传入空字符串表示本次绑定将替换与此事件关联的其他所有绑定。传递 `'+'` 则意味着加入此事件类型已绑定函数的列表中。

例如：

```
def turn_red(self, event):
    event.widget["activeforeground"] = "red"

self.button.bind("<Enter>", self.turn_red)
```

请注意，在 `turn_red()` 回调函数中如何访问事件的 widget 字段。该字段包含了捕获 X 事件的控件。下表列出了事件可供访问的其他字段，及其在 Tk 中的表示方式，这在查看 Tk 手册时很有用处。

Tk	Tkinter 事件字段	Tk	Tkinter 事件字段
%f	焦点	%A	字符
%h	height	%E	send_event
%k	keycode	%K	keysym
%s	状况	%N	keysym_num
%t	time	%T	类型
%w	宽度	%W	widget
%x	x	%X	x_root
%y	y	%Y	y_root

index 参数

很多控件都需要传入 index 参数。该参数用于指明 Text 控件中的位置，或指明 Entry 控件中的字符，或指明 Menu 控件中的菜单项。

Entry 控件的索引（index、view index 等） Entry 控件带有 index 属性，指向显示文本中的字符位置。这些 *tkinter* 函数可用于访问文本控件中的这些特定位置：

Text 控件的索引 Text 控件的索引语法非常复杂，最好还是在 Tk 手册中查看。

Menu 索引（menu.invoke()、menu.entryconfig() 等） 菜单的某些属性和方法可以操纵特定的菜单项。只要属性或参数需要用到菜单索引，就可用以下方式传入：

- 一个整数，指的是控件项的数字位置，从顶部开始计数，从 0 开始；
- 字符串 `"active"`，指的是当前光标所在的菜单；
- 字符串 `"last"`，指的是上一个菜单项；
- 带有 `@` 前缀的整数，比如 `@6`，这里的整数解释为菜单坐标系中的 y 像素坐标。
- 表示没有任何菜单条目的字符串 `"none"` 经常与 menu.activate() 一同被用来停用所有条目，以及
- 与菜单项的文本标签进行模式匹配的文本串，从菜单顶部扫描到底部。请注意，此索引类型是在其他所有索引类型之后才会考虑的，这意味着文本标签为 `last`、`active` 或 `none` 的菜单项匹配成功后，可能会视为这些单词文字本身。

图片

通过 `tkinter.Image` 的各种子类可以创建相应格式的图片：

- `BitmapImage` 对应 XBM 格式的图片。
- `PhotoImage` 对应 PGM、PPM、GIF 和 PNG 格式的图片。后者自 Tk 8.6 开始支持。

这两种图片可通过 `file` 或 `data` 属性创建的（也可能由其他属性创建）。

然后可在某些支持 `image` 属性的控件中（如标签、按钮、菜单）使用图片对象。这时，Tk 不会保留对图片对象的引用。当图片对象的最后一个 Python 引用被删除时，图片数据也会删除，并且 Tk 会在用到图片对象的地方显示一个空白框。

也参考:

Pillow 包增加了对 BMP、JPEG、TIFF 和 WebP 等格式的支持。

25.1.7 文件句柄

Tk 允许为文件操作注册和注销一个回调函数，当对文件描述符进行 I/O 时，Tk 的主循环会调用该回调函数。每个文件描述符只能注册一个处理程序。示例代码如下:

```
import tkinter
widget = tkinter.Tk()
mask = tkinter.READABLE | tkinter.WRITABLE
widget.tk.createfilehandler(file, mask, callback)
...
widget.tk.deletefilehandler(file)
```

在 Windows 系统中不可用。

由于不知道可读取多少字节，你可能不希望使用*BufferedIOBase* 或*TextIOBase* 的*read()* 或*readline()* 方法，因为这些方法必须读取预定数量的字节。对于套接字，可使用*recv()* 或*recvfrom()* 方法；对于其他文件，可使用原始读取方法或 `os.read(file.fileno(), maxbytecount)`。

`Widget.tk.`**`createfilehandler`**(*file*, *mask*, *func*)

注册文件处理程序的回调函数 *func*。*file* 参数可以是具备*fileno()* 方法的对象（例如文件或套接字对象），也可以是整数文件描述符。*mask* 参数是下述三个常量的逻辑“或”组合。回调函数将用以下格式调用:

```
callback(file, mask)
```

`Widget.tk.`**`deletefilehandler`**(*file*)

注销文件处理函数。

`tkinter.`**`READABLE`**
`tkinter.`**`WRITABLE`**
`tkinter.`**`EXCEPTION`**

Constants used in the *mask* arguments.

25.2 `tkinter.colorchooser` --- 颜色选择对话框

源代码: Lib/tkinter/colorchooser.py

---

tkinter.colorchooser 模块提供了*Chooser* 类作为原生颜色选择对话框的接口。`Chooser` 实现了一个模式颜色选择对话框窗口。`Chooser` 类继承自*Dialog* 类。

`class` `tkinter.colorchooser.`**`Chooser`**(*master=None*, ***options*)

`tkinter.colorchooser.`**`askcolor`**(*color=None*, ***options*)

创建一个颜色选择对话框。调用此方法将显示相应窗口，等待用户进行选择，并将选择的颜色 (或 `None`) 返回给调用者。

也参考:

模块*tkinter.commondialog* Tkinter 标准对话框模块

25.3 tkinter.font --- Tkinter 字体封装

源代码: Lib/tkinter/font.py

---

tkinter.font 模块提供用于创建和使用命名字体的*Font* 类。

不同的字体粗细和倾斜是:

```
tkinter.font.NORMAL
tkinter.font.BOLD
tkinter.font.ITALIC
tkinter.font.ROMAN
```

class tkinter.font.**Font**(*root=None*, *font=None*, *name=None*, *exists=False*, ***options*)

Font 类表示命名字体。*Font* 实例具有唯一的名称，可以通过其族、大小和样式配置进行指定。命名字体是 Tk 将字体创建和标识为单个对象的方法，而不是通过每次出现时的属性来指定字体。

参数:

font - 字体指示符元组 (family, size, options)
name - 唯一的字体名
exists - 指向现有命名字体（如果有）

其他关键字选项（如果指定了 *font*，则忽略）:

family - 字体系列，例如 Courier，Times
size - 字体大小
如果 *size* 为正数，则解释为以磅为单位的大小。
如果 *size* 是负数，则将其绝对值
解释为以像素为单位的大小。
weight - 字体强调 (NORMAL, BOLD)（普通，加粗）
slant - ROMAN, ITALIC（正体，斜体）
underline - 字体下划线（0 - 无下划线，1 - 有下划线）
overstrike - 字体删除线（0 - 无删除线，1 - 有删除线）

actual(*option=None*, *displayof=None*)
返回字体的属性。

cget(*option*)
检索字体的某一个属性值。

config(***options*)
修改字体的某一个属性值。

copy()
返回当前字体的新实例。

measure(*text*, *displayof=None*)
返回以当前字体格式化时文本将在指定显示上占用的空间量。如果未指定显示，则假定为主应用程序窗口。

metrics(**options*, ***kw*)
返回特定字体的数据。选项包括:

***ascent* - 基线和最高点之间的距离**（在该字体中的一个字符可以占用的空间中）

***descent* - 基线和最低点之间的距离**（在该字体中的一个字符可以占用的空间中）

***linespace* - 所需最小垂直间距（在两个** 该字体的字符间，使得这两个字符在垂直方向上不重叠）。

fixed - 如果该字体宽度被固定则为 1，否则为 0。

`tkinter.font.`**`families`**(*root=None*, *displayof=None*)
返回不同的字体系列。

`tkinter.font.`**`names`**(*root=None*)
返回定义字体的名字。

`tkinter.font.`**`nametofont`**(*name*)
返回一个*Font* 类，代表一个 tk 命名的字体。

25.4 Tkinter 对话框

25.4.1 `tkinter.simpledialog` --- 标准 Tkinter 输入对话框

源码: Lib/tkinter/simpledialog.py

---

tkinter.simpledialog 模块包含了一些便捷类和函数，用于创建简单的模态对话框，从用户那里读取一个值。

`tkinter.simpledialog.`**`askfloat`**(*title*, *prompt*, ***kw*)
`tkinter.simpledialog.`**`askinteger`**(*title*, *prompt*, ***kw*)
`tkinter.simpledialog.`**`askstring`**(*title*, *prompt*, ***kw*)
以上三个函数提供给用户输入期望值的类型的对话框.

`class` `tkinter.simpledialog.`**`Dialog`**(*parent*, *title=None*)
自定义对话框的基类.

`body`(*master*)
可以覆盖，用于构建对话框的界面，并返回初始焦点所在的部件。

`buttonbox`()
加入 OK 和 Cancel 按钮的默认行为. 重写自定义按钮布局.

25.4.2 `tkinter.filedialog` --- 文件选择对话框.

源码: Lib/tkinter/filedialog.py

---

tkinter.filedialog 模块提供了用于创建文件/目录选择窗口的类和工厂函数。

原生加载/保存对话框.

以下类和函数提供了文件对话窗口，这些窗口带有原生外观，具备可定制行为的配置项。这些关键字参数适用于以下类和函数：

parent ——对话框下方的窗口

title ——窗口的标题

initialdir ——对话框的启动目录

initialfile ——打开对话框时选中的文件

filetypes ——（标签，匹配模式）元组构成的列表，允许使用“*”通配符

defaultextension ——默认的扩展名，用于加到文件名后面（保存对话框）。

multiple ——为 True 则允许多选

** 静态工厂函数 **

调用以下函数时，会创建一个模态的、原生外观的对话框，等待用户选取，然后将选中值或 None 返回给调用者。

tkinter.filedialog.**askopenfile**(*mode="r"*, ***options*)
tkinter.filedialog.**askopenfiles**(*mode="r"*, ***options*)
上述两个函数创建了*Open* 对话框，并以只读模式返回打开的文件对象。

tkinter.filedialog.**asksaveasfile**(*mode="w"*, ***options*)
创建*SaveAs* 对话框并返回一个写入模式打开的文件对象。

tkinter.filedialog.**askopenfilename**(***options*)
tkinter.filedialog.**askopenfilenames**(***options*)
以上两个函数创建了*Open* 对话框，并返回选中的文件名，对应着已存在的文件。

tkinter.filedialog.**asksaveasfilename**(***options*)
创建*SaveAs* 对话框，并返回选中的文件名。

tkinter.filedialog.**askdirectory**(***options*)

提示用户选择一个目录.
其他关键字参数：
mustexist ——确定是否必须为已存在的目录。

class tkinter.filedialog.**Open**(*master=None*, ***options*)
class tkinter.filedialog.**SaveAs**(*master=None*, ***options*)
上述两个类提供了用于保存和加载文件的原生对话窗口。

** 便捷类 **

以下类用于从头开始创建文件/目录窗口。不会模仿当前系统的原生外观。

class tkinter.filedialog.**Directory**(*master=None*, ***options*)
创建对话框，提示用户选择一个目录。

備🄴: 为了实现自定义的事件处理和行为，应继承 *FileDialog* 类。

---

class tkinter.filedialog.**FileDialog**(*master*, *title=None*)
创建一个简单的文件选择对话框。

cancel_command(*event=None*)
触发对话窗口的终止。

dirs_double_event(*event*)
目录双击事件的处理程序。

dirs_select_event(*event*)
目录单击事件的处理程序。

files_double_event(*event*)
文件双击事件的处理程序。

files_select_event(*event*)
文件单击事件的处理程序。

filter_command(*event=None*)
以目录过滤文件.

get_filter()
获取当前使用的文件过滤器.

get_selection()
获取当前选择的项目.

go(*dir_or_file=os.curdir*, *pattern="*"*, *default=""*, *key=None*)
渲染对话和启动事件循环,

ok_event(*event*)
退出对话回到当前选择.

quit(*how=None*)
退出对话回到文件名, 如果有的话.

set_filter(*dir*, *pat*)
设置文件过滤器.

set_selection(*file*)
将当前选中文件更新为 *file*。

class tkinter.filedialog.**LoadFileDialog**(*master*, *title=None*)
FileDialog 的一个子类，为选择已有文件创建对话窗口。

ok_command()
检测有否给出文件，以及选中的文件是否存在。

class tkinter.filedialog.**SaveFileDialog**(*master*, *title=None*)
FileDialog 的一个子类，创建用于选择目标文件的对话窗口。

ok_command()
检测选中文件是否为目录。如果选中已存在文件，则需要进行确认。

25.4.3 tkinter.commondialog --- 对话窗口模板

源码: Lib/tkinter/commondialog.py

---

tkinter.commondialog 模块提供了 *Dialog* 类，是其他模块定义的对话框的基类。

class tkinter.commondialog.**Dialog**(*master=None*, ***options*)

show(*color=None*, ***options*)
渲染对话窗口.

也参考:

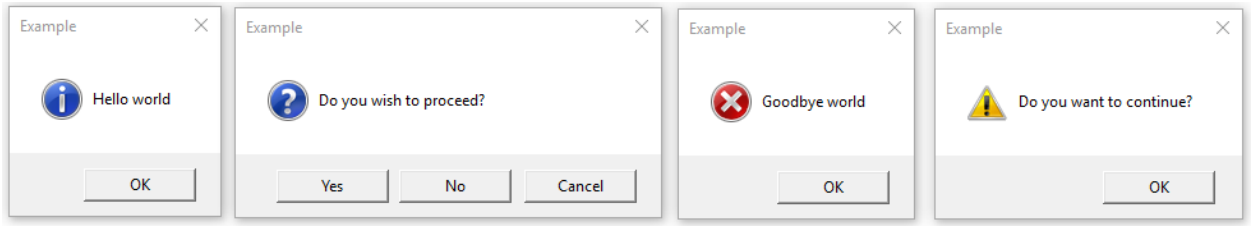
模块*tkinter.messagebox*, tut-files

25.5 tkinter.messagebox --- Tkinter 消息提示

源代码: Lib/tkinter/messagebox.py

---

tkinter.messagebox 模块提供了一个模板基类以及多个常用配置的便捷方法。消息框为模式窗口并将基于用户的选择返回 (True, False, OK, None, Yes, No) 的一个子集。常用消息框风格和布局包括但不限于:



class tkinter.messagebox.**Message**(*master=None*, ***options*)
创建一个默认信息消息框。

信息消息框

tkinter.messagebox.**showinfo**(*title=None*, *message=None*, ***options*)

警告消息框

tkinter.messagebox.**showwarning**(*title=None*, *message=None*, ***options*)
tkinter.messagebox.**showerror**(*title=None*, *message=None*, ***options*)

疑问消息框

tkinter.messagebox.**askquestion**(*title=None*, *message=None*, ***options*)
tkinter.messagebox.**askokcancel**(*title=None*, *message=None*, ***options*)
tkinter.messagebox.**askretrycancel**(*title=None*, *message=None*, ***options*)
tkinter.messagebox.**askyesno**(*title=None*, *message=None*, ***options*)
tkinter.messagebox.**askyesnocancel**(*title=None*, *message=None*, ***options*)

25.6 tkinter.scrolledtext --- 滚动文字控件

源代码: Lib/tkinter/scrolledtext.py

---

tkinter.scrolledtext 模块提供了一个同名的类，实现了带有垂直滚动条并被配置为可以“正常运作”的文本控件。使用*ScrolledText* 类会比直接配置一个文本控件附加滚动条要简单得多。

文本控件与滚动条打包在一个 Frame 中，Grid 方法和 Pack 方法的布局管理器从 Frame 对象中获得。这允许*ScrolledText* 控件可以直接用于实现大多数正常的布局管理行为。

如果需要更具体的控制，可以使用以下属性:

class tkinter.scrolledtext.**ScrolledText**(*master=None*, ***kw*)

frame
围绕文本和滚动条控件的框架。

vbar
滚动条控件。

25.7 tkinter.dnd --- 拖放操作支持

源代码: Lib/tkinter/dnd.py

備🄴: 此模块是实验性的且在为 Tk DND 所替代后将被弃用。

---

tkinter.dnd 模块为单个应用内部的对象提供了在同一窗口中或多个窗口间的拖放操作支持。要将对象设为可拖放，你必须为其创建启动拖放进程的事件绑定。通常，你要将 ButtonPress 事件绑定到你所编写的回调函数 (参见绑定和事件)。该函数应当调用 *dnd_start()*，其中'source' 为要拖动的对象，而'event' 为发起调用的事件（你的回调函数的参数）。

目标对象的选择方式如下:

1. 从顶至底地在鼠标之下的区域中搜索目标控件
   - 目标控件应当具有一个指向可调用对象的 *dnd_accept* 属性
   - 如果 *dnd_accept* 不存在或是返回 None，则将转至父控件中搜索
   - 如果目标控件未找到，则目标对象为 None
2. 调用 *<old_target>.dnd_leave(source, event)*
3. 调用 *<new_target>.dnd_enter(source, event)*
4. 调用 *<target>.dnd_commit(source, event)* 来通知释放
5. 调用 *<source>.dnd_end(target, event)* 来表明拖放的结束

class tkinter.dnd.**DndHandler**(*source*, *event*)
DndHandler 类处理拖放事件，在事件控件的根对象上跟踪 Motion 和 ButtonRelease 事件。

cancel(*event=None*)
取消拖放进程。

finish(*event*, *commit=0*)
执行结束播放函数。

on_motion(*event*)
在执行拖动期间为目标对象检查鼠标之下的区域。

on_release(*event*)
当释放模式被触发时表明拖动的结束。

tkinter.dnd.**dnd_start**(*source*, *event*)
用于拖放进程的工厂函数。

也参考:

绑定和事件

25.8 tkinter.ttk --- Tk 主题部件

源代码: Lib/tkinter/ttk.py

---

tkinter.ttk 模块自 Tk 8.5 开始引入，可用于访问 Tk 风格的控件包。如果 Python 未基于 Tk 8.5 编译，只要安装了 *Tile* 仍可访问本模块。前一种采用 Tk 8.5 的方式带有更多好处，比如在 X11 系统下支持反锯齿字体渲染和透明窗口（需要 X11 中的窗口组管理器）。

tkinter.ttk 的基本设计思路，就是尽可能地把控件的行为代码与实现其外观的代码分离开来。

也参考:

Tk 控件风格 一份文档介绍 Tk 支持的主题

25.8.1 使用 Ttk

开始使用 Ttk, 导入模块:

```
from tkinter import ttk
```

重写基础 Tk 控件, 导入应跟随 Tk 导入:

```
from tkinter import *
from tkinter.ttk import *
```

这段代码会让以下几个 tkinter.ttk 控件（Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale 和 Scrollbar）自动替换掉 Tk 的对应控件。

使用新控件的直接好处，是拥有更好的跨平台的外观，但新旧控件并不完全兼容。主要区别在于，Ttk 组件不再包含“fg”、“bg”等与样式相关的参数。而是用 ttk.Style 类来定义更美观的样式效果。

也参考:

将现有的应用程序转换为 Tile 控件，可参考 Converting existing applications to use Tile widgets 。 此文介绍使用新控件时的常见差别（使用 Tcl ）。

25.8.2 Ttk 控件

ttk 中有 18 种部件，其中十二种已存在于 tkinter 中: Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale, Scrollbar 以及*Spinbox*。另外六种是新增的: *Combobox*, *Notebook*, *Progressbar*, Separator, Sizegrip 以及*Treeview*。它们全都是*Widget* 的子类。

ttk 控件可以改善应用程序的外观。如上所述，修改样式的代码与 tk 控件存在差异。

Tk 代码:

```
l1 = tkinter.Label(text="Test", fg="black", bg="white")
l2 = tkinter.Label(text="Test", fg="black", bg="white")
```

Ttk 代码:

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
l2 = ttk.Label(text="Test", style="BW.TLabel")
```

有关*TtkStyling* 的更多信息，请参阅`Style` 类文档。

25.8.3 控件

`ttk.Widget` 定义了由 Tk 带主题部件所支持的标准选项和方法，它们不应该被直接实例化。

标准选项

所有 `ttk` 小部件接受以下选项：

选项	描述
类	指定窗口类。在查询选项数据库中窗口的其他选项时，使用该类，确定窗口的默认绑定标签，以及选择控件的默认布局和样式。此选项仅为只读，并且只能在创建窗口时指定。
光标	指定要用于控件的鼠标光标。如果设置为空字符串（默认值），则为父控件继承光标.
takefocus	确定窗口是否在键盘遍历期间接受焦点。返回 0 或 1，返回空字符串。如果返回 0，则表示在键盘遍历期间应该跳过该窗口。如果为 1，则表示只要可以查看窗口就应该接收输入焦点。并且空字符串意味着遍历脚本决定是否关注窗口。
风格	可用于指定自定义控件样式。

可滚动控件选项

控件支持以下选项使用滚动条控制.

选项	描述
xscroll 命令	用于与水平滚动条通讯. 当视图在控件的窗口改变, 控件将会基于 scroll 命令生成 Tcl 命令. 通常该参数由滚动条的 `Scrollbar.set()` 方法组成。当窗口中的可见内容发生变化时，将会刷新滚动条的状态。
yscroll 命令	用于与垂直滚动条通讯. 更多信息请参考上面的信息.

标签选项

以下选项支持标签, 按钮已及其他类按钮的控件.

选项	描述
文本	指定显示在控件内的文本。
文本变量	指定一个变量名，其值将用于设置 text 参数。
下划线	设置文本字符串中带下划线字符的索引（基于 0）。下划线字符用于激活快捷键。
图片	指定一个用于显示的图像。这是一个由 1 个或多个元素组成的列表。第一个元素是默认的图像名称。列表的其余部分是由`Style.map()`定义的“状态/值对”的序列，指定控件在某状态或状态组合时要采用的图像。列表中的所有图像应具备相同的尺寸。
compound	指定同时存在 text 和 image 参数时，应如何显示文本和对应的图片。合法的值包括： • text: 只显示文本 • image: 只显示图片 • top, bottom, left, right: 分别显示图片的上, 下, 左, 右的文本. • none: 默认. 如果设置显示图片, 否则文本.
宽度	如果值大于零，指定文本标签留下多少空间，单位是字符数；如果值小于零，则指定最小宽度。如果等于零或未指定，则使用文本标签本身的宽度。

兼容性选项

选项	描述
状况	可以设为“normal”或“disabled”，以便控制“禁用”状态标志位。本参数只允许写入：用以改变控件的状态，但`Widget.state()`方法不影响本参数。

控件状态

控件状态是无关状态标志的位图.

标志	描述
活动	鼠标光标经过控件并按下鼠标按钮，将引发动作。
禁用	在程序控制下控件是禁用的
焦点	控件有键盘焦点
按压	控件已被按下。
选择	勾选或单选框之类的控件，表示启用、选中状态。
背景	Windows 和 Mac 系统的窗口具有“激活”或后台的概念。后台窗口的控件会设置 *foreground* 参数，而前台窗口的控件则会清除此状态。
只读	控件不允许用户修改。
alternate	控件的备选显式格式。
无效的	控件的值是无效的

所谓的控件状态，就是一串状态名称的组合，可在某个名称前加上感叹号，表示该状态位是关闭的。

ttk.Widget

除了以下方法之外，ttk.Widget 还支持 tkinter.Widget.cget() 和 tkinter.Widget.configure() 方法。

class tkinter.ttk.**Widget**

identify(*x*, *y*)
返回位于 *x y* 的控件名称，如果该坐标点不属于任何控件，则返回空字符串。

x 和 *y* 是控件内的相对坐标，单位是像素。

instate(*statespec*, *callback=None*, **args*, ***kw*)
检测控件的状态。如果没有设置回调函数，那么当控件状态符合 *statespec* 时返回 True，否则返回 False。如果指定了回调函数，那么当控件状态匹配 *statespec* 时将会调用回调函数，且带上参数 args。

state(*statespec=None*)
修改或查询控件的状态。如果给出了 *statespec*，则会设置控件的状态，并返回一个新的 *statespec*，表明哪些标志做过改动。如果未给出 *statespec*，则返回当前启用的状态标志。

statespec 通常是列表或元组类型。

25.8.4 组合框

ttk.Combobox 控件是文本框和下拉列表的组合体。该控件是 Entry 的子类。

除了从*Widget* 继承的 Widget.cget()、Widget.configure()、*Widget.identify()*、*Widget.instate()* 和*Widget.state()* 方法，以及从 Entry 继承的 Entry.bbox()、Entry.delete()、Entry.icursor()、Entry.index()、Entry.insert()、Entry.selection()、Entry.xview() 方法，ttk.Combobox 还自带了其他几个方法。

选项

控件可设置以下参数：

选项	描述
exportselection	布尔值，如果设为 True，则控件的选中文字将关联为窗口管理器的选中文字（可由 Misc.selection_get 返回）。
justify	指定文本在控件中的对齐方式。可为 left、center、right 之一。
height	设置下拉列表框的高度。
postcommand	一条代码，（可用 Misc.register 进行注册），在显示之前将被调用。可用于选取要显示的值。
状况	normal、readonly 或 disabled。在 readonly 状态下，数据不能直接编辑，用户只能从下拉列表中选取。在 normal 状态下，可直接编辑文本框。在 disabled 状态下，无法做任何交互。
文本变量	设置一个变量名，其值与控件的值关联。每当该变量对应的值发生变动时，控件值就会更新，反之亦然。参见 tkinter.StringVar。
值	设置显示于下拉列表中的值。
宽度	设置为整数值，表示输入窗口的应有宽度，单位是字符单位（控件字体的平均字符宽度）。

虚拟事件

当用户从下拉列表中选择某个元素时，控件会生成一条 **«ComboboxSelected»** 虚拟事件。

ttk.Combobox

class tkinter.ttk.**Combobox**

current(*newindex=None*)
如果给出了 *newindex*，则把控件值设为 *newindex* 位置的元素值。否则，返回当前值的索引，当前值未在列表中则返回 -1。

get()
返回控件的当前值。

set(*value*)
设置控件的值为 *value* 。

25.8.5 Spinbox

`ttk.Spinbox` 控件是 `ttk.Entry` 的扩展，带有增减箭头。可用于数字或字符串列表。这是 `Entry` 的子类。

除了从*`Widget`* 继承的 `Widget.cget()`、`Widget.configure()`、`Widget.identified()`、*`Widget.instate()`* 和*`Widget.state()`* 方法，以及从 `Entry` 继承的 `Entry. bbox()`、`Entry.delete()`、`Entry.icursor()`、`Entry.index()`、`Entry.insert()`、`Entry.xview()` 方法，控件还自带了其他一些方法，在 `ttk.Spinbox` 中都有介绍。

选项

控件可设置以下参数：

选项	描述
从	浮点值。即递减按钮要递减的最小值。作为参数使用时必须写成 `from_`，因为 `from` 是 Python 关键字。
到	浮点值。即递增按钮能够到达的最大值。
增加	浮点值。指定递增/递减按钮每次的修改量。默认值为 1.0。
值	字符串或浮点值构成的序列。如若给出，则递增/递减会在此序列元素间循环，而不是增减数值。
wrap	布尔值。若为 `True`，则递增和递减按钮会由 `to` 值循环至 `from` 值，或由 `from` 值循环至 `to` 值。
格式	字符串。指定递增/递减按钮的数字格式。必须以“%W.Pf”的格式给出，W 是填充的宽度，P 是小数精度，% 和 f 就是本身的含义。
命令	Python 回调函数。只要递增或递减按钮按下之后，就会进行不带参数的调用。

虚拟事件

用户若按下 <Up>，则控件会生成 **«Increment»** 虚拟事件，若按下 <Down> 则会生成 **«Decrement»** 事件。

ttk.Spinbox

class tkinter.ttk.**Spinbox**

get()
返回 spinbox 当前值

set(*value*)
设置 spinbox 的值为 * 值 *.

25.8.6 笔记本

Ttk Notebook 控件管理着多个窗口的集合，每次显示其中的一个。每个子窗口都与某个 tab 关联，可供用户选中以改变当前显示的窗口。

选项

控件可设置以下参数：

选项	描述
height	如若给出且大于 0，则指定子窗口面板的应有高度（不含内部缩进）。否则会采用所有子窗口面板的最大高度。
padding	指定在控件外部添加的留白。padding 是最多包含四个值的列表，指定左顶右底的空间。如果给出的元素少于四个，底部值默认为顶部值，右侧值默认为左侧值，顶部值默认为左侧值。
宽度	若给出且大于 0，则设置面板的应有宽度（不含内部 padding）。否则将采用所有子窗口面板的最大宽度。

Tab 选项

Tab 特有属性如下：

选项	描述
状况	可为 normal、disabled 或 disabled 之一。若为 disabled 则不能选中。若为 hidden 则不会显示。
sticky	指定子窗口在面板内的定位方式。应为包含零个或多个 n、s、e 、w 字符的字符串。每个字母表示子窗口应紧靠的方向（北、南、东或西），正如 `grid()` 位置管理器所述。
padding	指定控件和面板之间的留白空间。格式与本控件的 padding 属性相同。
文本	指定显示在 tab 上的文本。
图片	指定显示在 tab 上的图片。参见 *`Widget`* 的 image 属性。
compound	当文本和图片同时存在时，指定图片相对于文本的显示位置。合法的属性值参见 *Label Options* 。
下划线	指定下划线在文本字符串中的索引（基于 0）。如果调用过了 *`Notebook.enable_traversal()`*，带下划线的字符将用于激活快捷键。

Tab 标识

The tab_id present in several methods of `ttk.Notebook` may take any of the following forms:

- 位于 0 和 tab 总数之间的整数值。
- 子窗口的名称
- 以“@x,y”形式给出的位置，唯一标识了 tab 页。
- 字符串“current”，标识当前选中的 Tab。
- 字符串字面值”end”，它返回标签页的数量 (仅适用于*`Notebook.index()`*)

虚拟事件

当选中一个新 tab 页之后，控件会生成一条 **«NotebookTabChanged»** 虚拟事件。

ttk.Notebook

class tkinter.ttk.**Notebook**

add(*child*, ***kw*)
添加一个新 tab 页。

如果窗口是由 Notebook 管理但处于隐藏状态，则会恢复到之前的位置。

可用属性请参见*Tab Options* 。

forget(*tab_id*)
删除 *tab_id* 指定的 tab 页，移除与其窗口的关联。

hide(*tab_id*)
隐藏 *tab_id* 指定的 tab 页。

tab 页不会显示出来，但关联的窗口仍接受 Notebook 的管理，其配置属性会继续保留。隐藏的 tab 页可由*`add()`* 恢复。

identify(*x*, *y*)
返回 tab 页内位置为 *x*、*y* 的控件名称，若不存在则返回空字符串。

index(*tab_id*)
返回 *tab_id* 指定 tab 页的索引值，如果 *tab_id* 为 end 则返回 tab 页的总数。

insert(*pos*, *child*, ***kw*)
在指定位置插入一个面板控件。

pos 可为字符串“end”、整数索引值或子窗口名称。如果 *child* 已由 Notebook 管理，则移至指定位置。

可用属性请参见*Tab Options* 。

select(*tab_id=None*)
选中 *tab_id* 指定 tab。

显示关联的子窗口，之前选中的窗口将取消映射关系。如果省略 *tab_id* ，则返回当前选中面板的控件名称。

tab(*tab_id*, *option=None*, ***kw*)
查询或修改 *tab_id* 指定 tab 的属性。

如果未给出 *kw*，则返回由 tab 属性组成的字典。如果指定了 *option*，则返回其值。否则，设置属性值。

tabs()
返回 Notebook 管理的窗口列表。

enable_traversal()
为包含 Notebook 的顶层窗口启用键盘遍历。

这将为包含 Notebook 的顶层窗口增加如下键盘绑定关系：

- `Control-Tab`：将当前 tab 加入选中列表。
- `Shift-Control-Tab`：选中当前 tab 之前的页。
- `Alt-K`：这里 *K* 是任意 tab 页的快捷键（带下划线）字符，将会直接选中该 tab。

一个顶层窗口中可为多个 Notebook 启用键盘遍历，包括嵌套的 Notebook 。但仅当所有面板都将所在 Notebook 作为父控件时，键盘遍历才会生效。

25.8.7 Progressbar

`ttk.Progressbar` 控件可为长时间操作显示状态。可工作于两种模式：1）确定模式，显示相对完成进度；2）不确定模式，显示动画让用户知道工作正在进行中。

选项

控件可设置以下参数：

选项	描述
orient	horizontal 或 vertical。指定进度条的显示方向。
length	指定进度条长轴的长度（横向为宽度，纵向则为高度）。
模式	determinate 或 indeterminate。
maximum	设定最大值。默认为 100。
值	进度条的当前值。在 determinate 模式下代表已完成的工作量。在 indeterminate 模式下，解释为 *maximum* 的模；也就是说，当本值增至 *maximum* 时，进度条完成了一个“周期”。
variable	与属性值关联的变量名。若给出，则当变量值变化时会自动设为进度条的值。
phase	只读属性。只要值大于 0 且在 determinate 模式下小于最大值，控件就会定期增大该属性值。当前主题可利用本属性提供额外的动画效果。

ttk.Progressbar

class tkinter.ttk.**Progressbar**

start(*interval=None*)
启动自动递增模式：安排一个循环的定时器事件，每隔 *interval* 毫秒调用一次`Progressbar.step()`。*interval* 可省略，默认为 50 毫秒。

step(*amount=None*)
将进度条的值增加 *amount*。

amount 可省略，默认为 1.0。

stop()
停止自增模式：取消所有由*Progressbar.start()* 启动的循环定时器事件。

25.8.8 Separator

ttk.Separator 控件用于显示横向或纵向的分隔条。

除由 ttk.Widget 继承而来的方法外，没有定义其他方法。

选项

属性如下：

选项	描述
orient	horizontal 或 vertical。指定分隔条的方向。

25.8.9 Sizegrip

ttk.Sizegrip 控件允许用户通过按下并拖动控制柄来调整内部顶层窗口的大小。

除由 ttk.Widget 继承的之外，没有其他属性和方法。

与平台相关的注意事项

- On macOS, toplevel windows automatically include a built-in size grip by default. Adding a Sizegrip is harmless, since the built-in grip will just mask the widget.

Bug

- 假如内部的顶层窗口位置是相对于屏幕的右侧或底部进行设置的，那么 Sizegrip 控件将不会改变窗口的大小。
- 仅支持东南方向的缩放。

25.8.10 Treeview

ttk.Treeview 控件可将多项内容分层级显示。每个数据项抖带有一个文本标签、一张图片（可选）和一个数据列表（可选）。这些数据值将在树标签后面分列显示。

数据值的显示顺序可用属性 displaycolumns 进行控制。树控件还可以显示列标题。数据列可通过数字或名称进行访问，各列的名称在属性 columns 中列出。参阅*Column Identifiers*。

每个数据项都由唯一名称进行标识。如果调用者未提供数据项的 ID，树控件会自动生成。根有且只有一个，名为 {}。根本身不会显示出来；其子项将显示在顶层。

每个数据项均带有 tag 列表，可用于绑定事件及控制外观。

Treeview 组件支持水平和垂直滚动，滚动时会依据*Scrollable Widget Options* 描述的属性和*Treeview.xview()* 和*Treeview.yview()* 方法。

选项

控件可设置以下参数：

选项	描述
columns	列标识的列表，指定列的数量和名称。
displaycolumns	列标识的列表（索引可为符号或整数），指定要显示的数据列及显示顺序，或者字符串 “#all”。
height	指定可见的行数。注意：所需宽度由各列宽度之和决定。
padding	指定控件内部的留白。为不超过四个元素的长度列表。
selectmode	控制内部类如何进行选中项的管理。可为 extended、browse 或 none。若设为 extended（默认），则可选中多个项。若为 browse ，则每次只能选中一项。若为 none，则无法修改选中项。 请注意，代码和 tag 绑定可自由进行选中操作，不受本属性的限制。
show	由 0 个或下列值组成的列表，指定要显示树的哪些元素。 • tree： 在 #0 列显示树的文本标签。 • headings： 显示标题行。 默认为 “tree headings”，显示所有元素。 ** 注意 **： 第 #0 列一定是指 tree 列，即便未设置 show=”tree” 也一样。

数据项的属性

可在插入和数据项操作时设置以下属性。

选项	描述
文本	用于显示的文本标签。
图片	Tk 图片对象，显示在文本标签左侧。
值	关联的数据值列表。 每个数据项关联的数据数量应与 columns 属性相同。如果比 columns 属性的少，剩下的值将视为空。如果多于 columns 属性的，多余数据将被忽略。
open	`True` 或 `False`，表明是否显示数据项的子树。
tags	与该数据项关联的 tag 列表。

tag 属性

tag 可定义以下属性：

选项	描述
foreground	定义文本前景色。
背景	定义单元格或数据项的背景色。
font	定义文本的字体。
图片	定义数据项的图片，当 image 属性为空时使用。

列标识

列标识可用以下格式给出：

- 由 columns 属性给出的符号名。
- 整数值 n，指定第 n 列。
- #n 的字符串格式，n 是整数，指定第 n 个显示列。

🄵解：

- 数据项属性的显示顺序可能与存储顺序不一样。
- #0 列一定是指 tree 列，即便未指定 show="tree" 也是一样。

数据列号是指属性值列表中的索引值，显示列号是指显示在树控件中的列号。树的文本标签将显示在 #0 列。如果未设置 displaycolumns 属性，则数据列 n 将显示在第 #n+1 列。再次强调一下，**#0 列一定是指 tree 列**。

虚拟事件

Treeview 控件会生成以下虚拟事件。

Event	描述
«TreeviewSelect»	当选中项发生变化时生成。
«TreeviewOpen»	当焦点所在项的 open= True 时立即生成。
«TreeviewClose»	当焦点所在项的 open= True 之后立即生成。

Treeview.focus() 和*Treeview.selection()* 方法可用于确认涉及的数据项。

ttk.Treeview

class tkinter.ttk.**Treeview**

bbox(*item*, *column=None*)
返回某 *数据项*的边界（相对于控件窗口的坐标），形式为 (x, y, width, height) 。

若给出了 *column*，则返回该单元格的边界。若该 *数据项*不可见（即从属于已关闭项或滚动至屏幕外），则返回空字符串。

get_children(*item=None*)
返回从属于 *item* 的下级数据项列表。

若未给出 *item* ，则返回根的下级。

set_children(*item*, **newchildren*)
用 *newchildren* 替换 *item* 的下级数据。

对于 *item* 中存在而 *newchildren* 中不存在的数据项，会从树中移除。*newchildren* 中的数据不能是 *item* 的上级。注意，未给出 *newchildren* 会导致 *item* 的子项被移除。

column(*column*, *option=None*, ***kw*)
查询或修改 *column* 的属性。

如果未给出 *kw*，则返回属性值的字典。若指定了 *option*，则会返回该属性值。否则将设置属性值。

合法的属性/值可为：

- **id** 返回列名。这是只读属性。

- **anchor：标准的 Tk 锚点值**。 指定该列的文本在单元格内的对齐方式。
- **minwidth：宽度**。 列的最小宽度，单位是像素。在缩放控件或用户拖动某一列时，Treeview 会保证列宽不小于此值。
- **stretch: `True/False`** 指明列宽度是否应该在部件大小被改变时进行相应的调整。
- **width: width** 以像素表示的列宽度。

要配置树的列，则调用此方法并附带参数 column = "#0"

`delete`(**items*)
删除所有 *items* 及其下属。

根不能删除。

`detach`(**items*)
将所有 *items* 与树解除关联。

数据项及其下属依然存在，后续可以重新插入，目前只是不显示出来。

根不能解除关联。

`exists`(*item*)
如果给出的 *item* 位于树中，则返回 `True`。

`focus`(*item=None*)
如果给出 *item* 则设为当前焦点。否则返回当前焦点所在数据项，若无则返回"。

`heading`(*column*, *option=None*, ***kw*)
查询或修改某 *column* 的标题。

若未给出 *kw*，则返回列标题组成的列表。若给出了 *option* 则返回对应属性值。否则，设置属性值。

合法的属性/值可为：

- **text：文本**。 显示为列标题的文本。
- **image：图片名称** 指定显示在列标题右侧的图片。
- **anchor：锚点** 指定列标题文本的对齐方式。应为标准的 Tk 锚点值。
- **command：回调函数** 点击列标题时执行的回调函数。

若要对 tree 列进行设置，请带上 column = "#0" 进行调用。

`identify`(*component*, *x*, *y*)
返回 *x*、*y* 位置上 *component* 控件的描述信息，如果此处没有这种控件，则返回空字符串。

`identify_row`(*y*)
返回 *y* 位置上的数据项 ID。

`identify_column`(*x*)
返回 *x* 位置上的单元格所在的数据列 ID。

tree 列的 ID 为 #0 。

`identify_region`(*x*, *y*)
返回以下值之一：

region	区域
标题栏	树的标题栏区域。
heading	两个列标题之间的间隔区域。
tree	树区域。
cell	数据单元格。

可用性：Tk 8.6。

identify_element(*x*, *y*)
返回位于 *x* 、*y* 的数据项。

可用性：Tk 8.6。

index(*item*)
返回 *item* 在其数据项列表中的整数索引。

insert(*parent*, *index*, *iid=None*, ***kw*)
新建一个数据项并返回其 ID。

parent 是父项的 ID，若要新建顶级项则为空字符串。*index* 是整数或“end”，指明在父项的子项列表中的插入位置。如果 *index* 小于等于 0，则在开头插入新节点；如果 *index* 大于或等于当前子节点数，则将其插入末尾。如果给出了 *iid*，则将其用作数据项 ID；*iid* 不得存在于树中。否则会新生成一个唯一 ID。

此处可设置的属性请参阅*Item Options* 。

item(*item*, *option=None*, ***kw*)
查询或修改某 *item* 的属性。

如果未给出 option，则返回属性/值构成的字典。如果给出了 *option*，则返回该属性的值。否则，将属性设为 *kw* 给出的值。

move(*item*, *parent*, *index*)
将 *item* 移至指定位置，父项为 *parent* ，子项列表索引为 *index* 。

将数据项移入其子项之下是非法的。如果 *index* 小于等于 0，*item* 将被移到开头；如果大于等于子项的总数，则被移至最后。如果 *item* 已解除关联，则会被重新连接。

next(*item*)
返回 *item* 的下一个相邻项，如果 *item* 是父项的最后一个子项，则返回”。

parent(*item*)
返回 *item* 的父项 ID，如果 *item* 为顶级节点，则返回”。

prev(*item*)
返回 *item* 的前一个相邻项，若 *item* 为父项的第一个子项，则返回”。

reattach(*item*, *parent*, *index*)
Treeview.move() 的别名。

see(*item*)
确保 *item* 可见。

将 *item* 所有上级的 open 属性设为 `True`，必要时会滚动控件，让 *item* 处于树的可见部分。

selection()
返回由选中项构成的元组。

3.8 版更變: `selection()` 不再接受参数了。若要改变选中的状态，请使用以下方法。

selection_set(**items*)
让 *items* 成为新的选中项。

3.6 版更變: *items* 可作为多个单独的参数传递，而不是作为一个元组。

selection_add(**items*)
将 *items* 加入选中项。

3.6 版更變: *items* 可作为多个单独的参数传递，而不是作为一个元组。

selection_remove(**items*)
从选中项中移除 *items* 。

3.6 版更變: *items* 可作为多个单独的参数传递，而不是作为一个元组。

selection_toggle(**items*)
切换 *items* 中各项的选中状态。

3.6 版更變: *items* 可作为多个单独的参数传递，而不是作为一个元组。

set(*item*, *column=None*, *value=None*)
若带一个参数，则返回指定 *item* 的列/值字典。若带两个参数，则返回 *column* 的当前值。若带三个参数，则将 *item* 的 *column* 设为 *value*。

tag_bind(*tagname*, *sequence=None*, *callback=None*)
为 tag 为 *tagname* 的数据项绑定事件 *sequence* 的回调函数。当事件分发给该数据项时，tag 参数为 *tagname* 的全部数据项的回调都会被调用到。

tag_configure(*tagname*, *option=None*, ***kw*)
查询或修改 *tagname* 指定项的参数。

如果给出了 *kw*，则返回 *tagname* 项的属性字典。如果给出了 *option*，则返回 *tagname* 项的 *option* 属性值。否则，设置 *tagname* 项的属性值。

tag_has(*tagname*, *item=None*)
如果给出了 *item* ，则依据 *item* 是否具备 *tagname* 而返回 1 或 0。否则，返回 tag 为 *tagname* 的所有数据项构成的列表。

可用性：Tk 8.6。

xview(**args*)
查询或修改 Treeview 的横向位置。

yview(**args*)
查询或修改 Treeview 的纵向位置。

25.8.11 Ttk 风格

ttk 的每种控件都赋有一个样式，指定了控件内的元素及其排列方式，以及元素属性的动态和默认设置。默认情况下，样式名与控件的类名相同，但可能会被控件的 style 属性覆盖。如果不知道控件的类名，可用 `Misc.winfo_class()` 方法获取（somewidget.winfo_class()）。

也参考:

Tcl'2004 会议报告 文章解释了主题引擎的工作原理。

class tkinter.ttk.**Style**
用于操控风格数据库的类。

configure(*style*, *query_opt=None*, ***kw*)
查询或设置 *style* 的默认参数。

Each key in *kw* is an option and each value is a string identifying the value for that option.

例如，要将默认按钮改为扁平样式，并带有留白和各种背景色：

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()
```

(下页继续)

(繼續上一頁)

```
ttk.Style().configure("TButton", padding=6, relief="flat",
   background="#ccc")

btn = ttk.Button(text="Sample")
btn.pack()

root.mainloop()
```

map(*style*, *query_opt=None*, ***kw*)
查询或设置 *style* 的指定属性的动态值。

kw 的每个键都是一个属性，每个值通常应为列表或元组，其中包含以元组、列表或其他形式组合而成的状态标识（statespec）。状态标识是由一个或多个状态组合，加上一个值组成。

举个例子能更清晰些：

```
import tkinter
from tkinter import ttk

root = tkinter.Tk()

style = ttk.Style()
style.map("C.TButton",
    foreground=[('pressed', 'red'), ('active', 'blue')],
    background=[('pressed', '!disabled', 'black'), ('active', 'white')]
    )

colored_btn = ttk.Button(text="Test", style="C.TButton").pack()

root.mainloop()
```

请注意，要点是属性的（状态，值）序列的顺序，如果前景色属性的顺序改为 `[('active', 'blue'), ('pressed', 'red')]`，则控件处于激活或按下状态时的前景色将为蓝色。

lookup(*style*, *option*, *state=None*, *default=None*)
返回 *style* 中的 *option* 属性值。

如果给出了 *state*，则应是一个或多个状态组成的序列。如果设置了 *default* 参数，则在属性值缺失时会用作后备值。

若要检测按钮的默认字体，可以：

```
from tkinter import ttk

print(ttk.Style().lookup("TButton", "font"))
```

layout(*style*, *layoutspec=None*)
按照 *style* 定义控件布局。如果省略了 *layoutspec*，则返回该样式的布局属性。

若给出了 *layoutspec*，则应为一个列表或其他的序列类型（不包括字符串），其中的数据项应为元组类型，第一项是布局名称，第二项的格式应符合*Layouts* 的描述。

以下示例有助于理解这种格式（这里并没有实际意义）：

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()
```

(下页继续)

(繼續上一頁)

```
style = ttk.Style()
style.layout("TMenubutton", [
   ("Menubutton.background", None),
   ("Menubutton.button", {"children":
       [("Menubutton.focus", {"children":
           [("Menubutton.padding", {"children":
               [("Menubutton.label", {"side": "left", "expand": 1})]
           })]
       })]
   }),
])

mbtn = ttk.Menubutton(text='Text')
mbtn.pack()
root.mainloop()
```

element_create(*elementname*, *etype*, **args*, ***kw*)
在当前主题中创建一个新元素 *etype*，应为 image、from 或 vsapi。后者仅在 Windows XP 和 Vista 版的 Tk 8.6a 中可用，此处不再赘述。

如果用了 image，则 *args* 应包含默认的图片名，后面跟着状态标识/值（这里是 imagespec），*kw* 可带有以下属性：

- **border=padding** padding 是由不超过四个整数构成的列表，分别定义了左、顶、右、底的边界。
- **height=height** 定义了元素的最小高度。如果小于零，则默认采用图片本身的高度。
- **padding=padding** 定义了元素的内部留白。若未指定则默认采用 border 值。
- **sticky=spec** 定义了图片的对齐方式。spec 包含零个或多个 n、s、w、e 字符。
- **width=width** 定义了元素的最小宽度。如果小于零，则默认采用图片本身的宽度。

如果 *etype* 的值用了 from，则*element_create()* 将复制一个现有的元素。*args* 应包含主题名和可选的要复制的元素。若未给出要克隆的元素，则采用空元素。*kw* 参数将被丢弃。

element_names()
返回当前主题已定义的元素列表。

element_options(*elementname*)
返回 *elementname* 的属性列表。

theme_create(*themename*, *parent=None*, *settings=None*)
新建一个主题。

如果 *themename* 已经存在，则会报错。如果给出了 *parent*，则新主题将从父主题继承样式、元素和布局。若给出了 *settings*，则语法应与*theme_settings()* 的相同。

theme_settings(*themename*, *settings*)
将当前主题临时设为 *themename*，并应用 *settings*，然后恢复之前的主题。

settings 中的每个键都是一种样式而每个值可能包含'configure', 'map', 'layout' 和'element create' 等键并且它们被预期具有与分别由*Style.configure()*, *Style.map()*, *Style.layout()* 和*Style.element_create()* 方法所指定的相符的格式。

以下例子会对 Combobox 的默认主题稍作修改：

```
from tkinter import ttk
import tkinter

root = tkinter.Tk()

style = ttk.Style()
style.theme_settings("default", {
   "TCombobox": {
       "configure": {"padding": 5},
       "map": {
           "background": [("active", "green2"),
                          ("!disabled", "green4")],
           "fieldbackground": [("!disabled", "green3")],
           "foreground": [("focus", "OliveDrab1"),
                          ("!disabled", "OliveDrab2")]
       }
   }
})

combo = ttk.Combobox().pack()

root.mainloop()
```

theme_names()
返回所有已知主题的列表。

theme_use(*themename=None*)
若未给出 *themename*，则返回正在使用的主题。否则，将当前主题设为 *themename*，刷新所有控件并引发 «ThemeChanged» 事件。

布局

布局在没有属性时可以为 None，或是定义了元素排列方式的属性字典。布局机制采用了位置管理器的简化版本：给定一个初始容器（cavity），为每个元素都分配一个包装（parcel）。合法的选项/值包括：

- **side: whichside** 指定元素置于容器的哪一侧；顶、右、底或左。如果省略，则该元素将占据整个容器。
- **sticky: nswe** 指定元素在已分配包装盒内的放置位置。
- **unit: 0 或 1** 如果设为 1，则将元素及其所有后代均视作单个元素以供*Widget.identify()* 等使用。它被用于滚动条之类带有控制柄的东西。
- **children: [sublayout...]** 指定要放置于元素内的元素列表。每个元素都是一个元组（或其他序列类型），其中第一项是布局名称，另一项是个*Layout* 。

25.9 tkinter.tix --- TK 扩展包

源代码: Lib/tkinter/tix.py

3.6 版後已🄴用: 这个 TK 扩展已无人维护所以请不要在新代码中使用。请改用*tkinter.ttk*。

---

tkinter.tix (Tk Interface Extension) 模块提供了更丰富的额外可视化部件集。虽然标准 Tk 库包含许多有用的部件，但还远不够完备。*tkinter.tix* 库提供了标准 Tk 所缺少的大量常用部件: *HList*, *ComboBox*, *Control* (即 SpinBox) 以及一系列可滚动的部件。*tkinter.tix* 还包括了大量在多种不同领域的应用中很常用的部件: *NoteBook*, *FileEntry*, *PanedWindow* 等等；总共有超过 40 种。

使用这些新增部件，你可以为应用程序引入新的交互技术，创建更好用且更直观的用户界面。你在设计应用程序时可以通过选择最适合的部件来匹配你的应用程序和用户的特殊需求。

也参考:

Tix 主页 Tix 的主页。其中包括附加文档和下载资源的链接。

Tix Man Pages 在线版本的指南页面和参考材料。

Tix Programming Guide 在线版本的程序员参考材料。

Tix Development Applications 开发 Tix 和 Tkinter 程序的 Tix 应用。Tide 应用在 Tk 在 Tkinter 下工作，并包括了 **TixInspect**，这是一个可远程修改和调试 Tix/Tk/Tkinter 应用的检查工具。

25.9.1 使用 Tix

class tkinter.tix.**Tk**(*screenName=None*, *baseName=None*, *className='Tix'*)
最常用于代表应用主窗口的最高层级部件。它具有一个相关联的 Tcl 解释器。interpreter.

tkinter.tix 模块中的类子类化了*tkinter* 中的类。前者会导入后者，因此*tkinter.tix* 要使用 Tkinter，你所要做的就是导入一个模块。通常，你可以只导入*tkinter.tix*，并将最高层级调用由*tkinter.Tk* 替换为 tix.Tk:

```
from tkinter import tix
from tkinter.constants import *
root = tix.Tk()
```

要使用*tkinter.tix*，你必须安装有 Tix 部件，通常会与你的 Tk 部分一起安装。要测试你的安装，请尝试以下代码:

```
from tkinter import tix
root = tix.Tk()
root.tk.eval('package require Tix')
```

25.9.2 Tix 部件

Tix 将 40 多个部件类引入到*tkinter* 工具集中。

基本部件

class tkinter.tix.**Balloon**
Balloon 会在部件上弹出以提供帮助信息。当用户将光标移到一个与 Balloon 部件绑定的部件内时，将在屏幕上弹出一个显示描述性消息的小窗口。

class tkinter.tix.**ButtonBox**
ButtonBox 部件会创建一组按钮框，例如常用的 Ok Cancel 按钮框。

class tkinter.tix.**ComboBox**
ComboBox 部件类似于 MS Windows 中的组合框控件。用户可以通过在输入框子部件中输入或是在列表框子部件中选择来选定一个选项。

class tkinter.tix.**Control**
The Control 控件又名 SpinBox 控件。用户可通过点按两个方向键或直接输入内容来调整数值。更新的数值将被检查是否在用户定义的上下限之内。

class tkinter.tix.**LabelEntry**
LabelEntry 部件将输入框部件和标签打包为一个部件。它可被用来简化” 输入表单” 类界面的创建。

class tkinter.tix.**LabelFrame**
LabelFrame 部件将框架部件和标签打包为一个部件。要在一个 LabelFrame 部件中创建部件，应当创建与 `frame` 子部件相关联的新部件并在 `frame` 子部件中管理它们。

class tkinter.tix.**Meter**
Meter 部件可用来显示可能会耗费很长时间运行的后台任务的进度。

class tkinter.tix.**OptionMenu**
OptionMenu 可创建一个选项按钮菜单。

class tkinter.tix.**PopupMenu**
PopupMenu 部件可被用来替代 `tk_popup` 命令。`Tix` *PopupMenu* 部件的优势在于它所需要的应用操纵代码较少。

class tkinter.tix.**Select**
Select 控件是一组按钮子控件的容器。它可被用来为用户提供单选钮或复选钮形式的选项。

class tkinter.tix.**StdButtonBox**
StdButtonBox 部件是一个用于 Motif 风格对话框的标准按钮组。

文件选择器

class tkinter.tix.**DirList**
DirList 部件显示一个目录、它的上级目录和子目录的列表视图。用户可以选择列表中显示的某个目录或切换到另一个目录。

class tkinter.tix.**DirTree**
DirTree 部件显示一个目录、它的上级目录和子目录的树状视图。用户可以选择其中显示的某个目录或切换到另一个目录。

class tkinter.tix.**DirSelectDialog**
DirSelectDialog 部件以对话框窗口形式表示文件系统中的目录。用户可以使用该对话框窗口在文件系统中漫游以选择所需的目录。

class tkinter.tix.**DirSelectBox**
DirSelectBox 类似于标准的 Motif(TM) 目录选择框。它通常用于让用户选择一个目录。DirSelectBox 会将最近选择的目录存放在一个 ComboBox 部件中以便可以再次快速地选择它们。

class tkinter.tix.**ExFileSelectBox**
ExFileSelectBox 部件通常是嵌入在 tixExFileSelectDialog 部件中。它为用户提供了一种方便的选择文件方法。*ExFileSelectBox* 部件的风格非常类似于 MS Windows 3.1 中的标准文件对话框。

class tkinter.tix.**FileSelectBox**
FileSelectBox 类似于标准的 Motif(TM) 文件选择框。它通常用于让用户选择一个文件。FileSelectBox 会将最近选择的文件存放在一个*ComboBox* 部件中以便可以再次快速地选择它们。

class tkinter.tix.**FileEntry**
FileEntry 部件可被用于输入一个文件名。用户可以手动输入文件名。或者用户也可以按输入框旁边的按钮控件，这将打开一个文件选择对话框。

层级式列表框

class tkinter.tix.**HList**
HList 部件可被用于显示任何具有层级结构的数据，例如文件系统目录树。其中的列表条目带有缩进并按照它们在层级中的位置以分支线段相连。

class tkinter.tix.**CheckList**
CheckList 部件可显示一个供用户选择的条目列表。CheckList 的功能类似于 Tk 复选钮或单选钮部件，不同之处在于它能够处理比复选钮或单选钮多得多的条目。

class tkinter.tix.**Tree**
Tree 部件可被用于以树形显示具有层级结构的数据。用户可以通过打开或关闭部分树枝来调整树形视图。

表格式列表框

class tkinter.tix.**TList**
TList 部件可被用于以表格形式显示数据。*TList* 部件中的列表条目类似于 Tk 列表框部件中的条目。主要差异在于 (1) *TList* 部件能以二维格式显示列表条目 (2) 你可以在列表条目中使用图片以及多种颜色和字体。

管理器部件

class tkinter.tix.**PanedWindow**
PanedWindow 部件允许用户交互式地控制多个面板的大小。这些面板可以垂直或水平地排列。用户通过拖动两个面板间的控制柄来改变面板的大小。

class tkinter.tix.**ListNoteBook**
ListNoteBook 部件非常类似于 TixNoteBook 部件：它可被用于在有限空间内显示多个窗口，就像是一个“笔记本”。笔记本可分为许多页面（窗口）。同一时刻只能显示其中一个页面。用户可以通过在 hlist 子部件中选择所需页面的名称来切换这些页面。

class tkinter.tix.**NoteBook**
NoteBook 部件可被用于在有限空间内显示多个窗口，就像是一个“笔记本”。笔记本可分为许多页面。同一时刻只能显示其中一个页面。用户可以通过选择 NoteBook 部件顶端的可视化“选项卡”来切换这些页面。

图像类型

tkinter.tix 模块增加了:

- 将 pixmap 功能提供给所有*tkinter.tix* 和*tkinter* 部件以使用 XPM 文件创建彩色图像。
- Compound 图像类型可被用于创建由许多横行构成的图像；每一行都包含从左至右排列的一组条目（文本、位图、图像或空白）。例如，某个组合图像可被用于在一个 Tk Button 部件内同时显示一张位图和一个文本字符串。

其他部件

class tkinter.tix.**InputOnly**
InputOnly 部件可接收来自用户的输入，此功能可通过 bind 命令实现（仅限 Unix）。

表单布局管理器

tkinter.tix 还额外提供了以下部件来增强*tkinter* 的功能:

class tkinter.tix.**Form**
Form 布局管理器是以针对所有 Tk 部件的附加规则为基础的。

25.9.3 Tix 命令

class tkinter.tix.**tixCommand**
Tix 命令 提供了对 Tix 内部状态和 Tix 应用程序上下文等杂项元素的访问。大部分由这些方法控制的信息作为一个整体被发给应用程序，或是发给一个屏幕或显示区域，而不是某个特定窗口。

要查看当前的设置，通常的用法是:

```
from tkinter import tix
root = tix.Tk()
print(root.tix_configure())
```

tixCommand.**tix_configure**(*cnf=None*, ***kw*)
查询或修改 Tix 应用程序上下文的配置选项。如果未指定任何选项，则返回包含所有选项的字典。如果指定了不带值的选项，则该方法返回描述指定选项的列表（如果未指定选项则此列表与所返回值对应的子列表相同）。如果指定了一个或多个选项-值对，则该方法会将指定的选项修改为指定的值；在此情况下该方法将返回一个空字符串。选项可以是配置选项中的任何一个。

tixCommand.**tix_cget**(*option*)
返回由 *option* 给出的配置选项的当前值。选项可以是配置选项中的任何一个。

tixCommand.**tix_getbitmap**(*name*)
在某个位图目录中定位名称为 name.xpm 或 name 的位图文件（位图目录参见*tix_addbitmapdir()* 方法）。通过使用*tix_getbitmap()*，你可以避免在你的应用程序中硬编码位图文件的路径名。执行成功时，它返回位图文件的完整路径名，并带有前缀字符 @。返回值可被用于配置 Tk 和 Tix 部件的 bitmap 选项。

tixCommand.**tix_addbitmapdir**(*directory*)
Tix 维护了一个列表以供*tix_getimage()* 和*tix_getbitmap()* 方法在其中搜索图像文件。标准位图目录是 $TIX_LIBRARY/bitmaps。*tix_addbitmapdir()* 方法向该列表添加了 *directory*。通过使用此方法，应用程序的图像文件也可使用*tix_getimage()* 或*tix_getbitmap()* 方法来定位。

tixCommand.**tix_filedialog**([*dlgclass*])
返回可在来自该应用程序的同不调用之间共享的选择对话框。此方法将在首次被调用时创建一个选择对话框部件。此后对*tix_filedialog()* 的所有调用都将返回该对话框。可以传入一个字符串形式的可选形参 dlgclass 来指明所需的选择对话框类型。可用的选项有 tix, FileSelectDialog 或 tixExFileSelectDialog。

tixCommand.**tix_getimage**(*self*, *name*)
在某个位图目录（参见上文的*tix_addbitmapdir()* 方法）中定位名为 name.xpm, name.xbm 或 name.ppm 的图像文件。如果存在多个同名文件（但扩展名不同），则会按照 X 显示的深度选择图像类型：单色显示选择 xbm 图像而彩色显示则选择彩色图像。通过使用*tix_getimage()*，你可以避免在你的应用程序中硬编码图像文件的路径名。当执行成功时，此方法将返回新创建图像的名称，它可被用于配置 Tk 和 Tix 部件的 image 选项。

tixCommand.**tix_option_get**(*name*)
获取由 Tix 规格机制维护的选项。

tixCommand.**tix_resetoptions**(*newScheme*, *newFontSet*[, *newScmPrio*])
将 Tix 应用程序的规格设置与字体设置分别重置为 *newScheme* 和 *newFontSet*。这只会影响调用此方法之后创建的部件。因此，最好是在 Tix 应用程序的任何部件被创建之前调用 resetoptions 方法。

可以给出可选的形参 *newScmPrio* 来重置由 Tix 规格所设置的 Tk 选项的优先级。

由于 Tk 处理 X 选项数据库的特别方式，在 Tix 被导入并初始化之后，将无法再使用 `tix_config()` 方法来重置颜色方案和字体集。而必须要使用*`tix_resetoptions()`* 方法。

25.10 IDLE

源代码： Lib/idlelib/

IDLE 是 Python 所内置的开发与学习环境。

IDLE 具有以下特性：

- 编码于 100% 纯正的 Python，使用名为*`tkinter`* 的图形用户界面工具
- 跨平台：在 Windows、Unix 和 macOS 上工作近似。
- 提供输入输出高亮和错误信息的 Python 命令行窗口（交互解释器）
- 提供多次撤销操作、Python 语法高亮、智能缩进、函数调用提示、自动补全等功能的多窗口文本编辑器
- 在多个窗口中检索，在编辑器中替换文本，以及在多个文件中检索（通过 grep）
- 提供持久保存的断点调试、单步调试、查看本地和全局命名空间功能的调试器
- 配置、浏览以及其它对话框

25.10.1 目录

IDEL 有两种主要的窗口类型：Shell 窗口和编辑器窗口。其中编辑器窗口可以同时打开多个。并且对于 Windows 和 Linux 平台，窗口顶部主菜单各不相同。以下每个菜单说明项，都标识了与之关联的平台类型。

导出窗口，例如使用编辑 => 在文件中查找是编辑器窗口的的一个子类型。它们目前有着相同的主菜单，但是默认标题和上下文菜单不同。

在 macOS 上，只有一个应用程序菜单。它会根据当前选择的窗口动态变化。它具有一个 IDLE 菜单，并且下面描述的某些条目已移动到符合 Apple 准则的位置。

文件菜单（命令行和编辑器）

新建文件 创建一个文件编辑器窗口。

打开... 使用打开窗口以打开一个已存在的文件。

近期文件 打开一个近期文件列表，选取一个以打开它。

打开模块... 打开一个已存在的模块（搜索 sys.path）

类浏览器 于当前所编辑的文件中使用树形结构展示函数、类以及方法。在命令行中，首先打开一个模块。

路径浏览 在树状结构中展示 sys.path 目录、模块、函数、类和方法。

保存 如果文件已经存在，则将当前窗口保存至对应的文件。自打开或上次保存之后经过修改的文件的窗口标题栏首尾将出现星号 * 。如果没有对应的文件，则使用“另存为”代替。

保存为... 使用“保存为”对话框保存当前窗口。被保存的文件将作为当前窗口新的对应文件。

另存为副本... 保存当前窗口至另一个文件，而不修改当前对应文件。

打印窗口 通过默认打印机打印当前窗口。

Close Window Close the current window (if an unsaved editor, ask to save; if an unsaved Shell, ask to quit execution). Calling `exit()` or `close()` in the Shell window also closes Shell. If this is the only window, also exit IDLE.

Exit IDLE Close all windows and quit IDLE (ask to save unsaved edit windows).

编辑菜单（命令行和编辑器）

撤销操作 撤销当前窗口的最近一次操作。最高可以撤回 1000 条操作记录。

重做 重做当前窗口最近一次所撤销的操作。

剪切 复制选区至系统剪贴板，然后删除选区。

复制 复制选区至系统剪贴板。

粘贴 插入系统剪贴板的内容至当前窗口。

剪贴板功能也可用于上下文目录。

全选 选择当前窗口的全部内容。

查找... 打开一个提供多选项的查找窗口。

再次查找 重复上一次搜索（如果有的话）。

查找选区 查找当前选中的字符串，如果存在

在文件中查找... 打开文件查找对话框。将结果输出至新的输出窗口。

替换... 打开查找并替换对话框。

前往行 将光标移到所请求行的开头并使该行可见。对于超过文件尾的请求将会移到文件尾。清除所有选区并更新行列状态。

提示完成 打开一个可滚动列表以允许选择现有的名称。请参阅下面编辑与导航一节中的自动补全。

展开文本 展开键入的前缀以匹配同一窗口中的完整单词；重复以获得不同的扩展。

显示调用贴士 在函数的右括号后，打开一个带有函数参数提示的小窗口。请参阅下面的“编辑和导航”部分中的*Calltips* 。

显示周围括号 突出显示周围的括号。

格式菜单（仅 window 编辑器）

增加缩进 将选定的行右缩进（默认为 4 个空格）。

减少缩进 将选定的行左缩进（默认为 4 个空格）。

注释 在所选行的前面插入 ##。

取消注释 从所选行中删除开头的 # 或 ##。

制表符化 将 前导空格变成制表符。（注意：我们建议使用 4 个空格来缩进 Python 代码。）

取消制表符化 将 所有制表符转换为正确的空格数。

缩进方式切换 打开一个对话框，以在制表符和空格之间切换。

缩进宽度调整 打开一个对话框以更改缩进宽度。Python 社区接受的默认值为 4 个空格。

格式段落 在注释块或多行字符串或字符串中的选定行中，重新格式化当前以空行分隔的段落。段落中的所有行的格式都将少于 N 列，其中 N 默认为 72。

删除尾随空格 通过将 str.rstrip 应用于每行（包括多行字符串中的行），删除行尾非空白字符之后的尾随空格和其他空白字符。除 Shell 窗口外，在文件末尾删除多余的换行符。

运行菜单（仅 window 编辑器）

运行模块 执行检查模块 。如果没有错误，重新启动 shell 以清理环境，然后执行模块。输出显示在 shell 窗口中。请注意，输出需要使用“打印”或“写入”。执行完成后，Shell 将保留焦点并显示提示。此时，可以交互地探索执行的结果。这类似于在命令行执行带有 `python -i file` 的文件。

运行... 定制 与运行模块 相同，但使用自定义设置运行该模块。命令行参数扩展*sys.argv* ，就像在命令行上传递一样。该模块可以在命令行管理程序中运行，而无需重新启动。

检查模块 检查“编辑器”窗口中当前打开的模块的语法。如果尚未保存该模块，则 IDLE 会提示用户保存或自动保存，如在“空闲设置”对话框的“常规”选项卡中所选择的那样。如果存在语法错误，则会在“编辑器”窗口中指示大概位置。

Python Shell 打开或唤醒 Python Shell 窗口。

Shell 菜单（仅 window 编辑器）

查看最近重启 将 Shell 窗口滚动到上一次 Shell 重启。

重启 Shell 重启 shell 以清理环境，重置显示和异常处理。

上一条历史记录 循环浏览历史记录中与当前条目匹配的早期命令。

下一条历史记录 循环浏览历史记录中与当前条目匹配的后续命令。

中断执行 停止正在运行的程序。

调试菜单（仅 window 编辑器）

跳转到文件/行 查看当前行。以光标提示，且上一行为文件名和行号。如果找到目标，如果文件尚未打开则打开该文件，并显示目标行。使用此菜单项来查看异常回溯中引用的源代码行以及用文件中查找功能找到的行。也可在 Shell 窗口和 Output 窗口的上下文菜单中使用。

调试器（切换） 激活后，在 Shell 中输入的代码或从编辑器中运行的代码将在调试器下运行。在编辑器中，可以使用上下文菜单设置断点。此功能不完整，具有实验性。

堆栈查看器 在树状目录中显示最后一个异常的堆栈回溯，可以访问本地和全局。

自动打开堆栈查看器 在未处理的异常上切换自动打开堆栈查看器。

选项菜单（命令行和编辑器）

配置 IDLE 打开配置对话框并更改以下各项的首选项：字体、缩进、键绑定、文本颜色主题、启动窗口和大小、其他帮助源和扩展名。在 MacOS 上，通过在应用程序菜单中选择首选项来打开配置对话框。有关详细信息，请参阅：帮助和首选项下的首选项设置。

大多数配置选项适用于所有窗口或将来的所有窗口。以下选项仅适用于活动窗口。

显示/隐藏代码上下文（仅 window 编辑器） 在编辑窗口顶部打开一个面板来显示在窗口顶部滚动的代码块上下文。请参阅下文“编辑与导航”章节中的代码上下文。

显示/隐藏行号（仅 window 编辑器） 在编辑窗口左侧打开一个显示代码文本行编号的列。默认为关闭显示，这可以在首选项中修改 (参见设置首选项)。

缩放/还原高度 在窗口的正常尺寸和最大高度之间进行切换。初始尺寸默认为 40 行每行 80 字符，除非在配置 IDLE 对话框的通用选项卡中做了修改。屏幕的最大高度由首次在屏幕上将缩小的窗口最大化的操作来确定。改变屏幕设置可能使保存的高度失效。此切换操作在窗口最大化状态下无效。

Window 菜单（命令行和编辑器）

列出所有打开的窗口的名称；选择一个将其带到前台（必要时对其进行去符号化）。

帮助菜单（命令行和编辑器）

关于 IDLE 显示版本，版权，许可证，荣誉等。

IDLE 帮助 显示此 IDLE 文档，详细介绍菜单选项，基本编辑和导航以及其他技巧。

Python 文档 访问本地 Python 文档（如果已安装），或启动 Web 浏览器并打开 docs.python.org 显示最新的 Python 文档。

海龟演示 使用示例 Python 代码运行 turtledemo 模块和海龟绘图

可以在“常规”选项卡下的“配置 IDLE”对话框中添加其他帮助源。有关“帮助”菜单选项的更多信息，请参见下面的帮助源 小节。

上下文菜单

通过在窗口中右击（在 macOS 上则为按住 Control 键点击）来打开一个上下文菜单。上下文菜单也具有编辑菜单中的标准剪贴板功能。

剪切 复制选区至系统剪贴板，然后删除选区。

复制 复制选区至系统剪贴板。

粘贴 插入系统剪贴板的内容至当前窗口。

编辑器窗口也具有断点功能。设置了断点的行会被特别标记。断点仅在启用调试器运行时有效。文件的断点会被保存在用户的 `.idlerc` 目录中。

设置断点 在当前行设置断点

清除断点 清除当前行断点

shell 和输出窗口还具有以下内容。

跳转到文件/行 与调试菜单相同。

Shell 窗口也有一个输出折叠功能，参见下文的 *Python Shell* 窗口 小节。

压缩 如果将光标位于输出行上，则会折叠在上方代码和下方提示直到'Squeezed text' 标签之间的所有输出。

25.10.2 编辑和导航

编辑窗口

IDLE 可以在启动时打开编辑器窗口，这取决于选项设置和你启动 IDLE 的方式。在此之后，请使用 File 菜单。对于给定的文件只能打开一个编辑器窗口。

标题栏包含文件名称、完整路径，以及运行该窗口的 Python 和 IDLE 版本。状态栏包含行号 ('Ln') 和列号 ('Col')。行号从 1 开始；列号则从 0 开始。

IDE 会定扩展名为.py* 的文件包含 Python 代码而其他文件不包含。可使用 Run 菜单来运行 Python 代码。

按键绑定

在本节中，'C' 是指 Windows 和 Unix 上的 `Control` 键，以及 macOS 上的 `Command` 键。

- `Backspace` 向左删除; `Del` 向右删除
- `C-Backspace` 向左删除单词; `C-Del` 向右删除单词
- 方向键和 `Page Up`/`Page Down` 移动
- `C-LeftArrow` 和 `C-RightArrow` 按单词移动
- `Home`/`End` 跳转到行首/尾
- `C-Home`/`C-End` 跳转到文档首/尾
- 一些有用的 Emacs 绑定是从 Tcl / Tk 继承的:
  - `C-a` 行首
  - `C-e` 行尾
  - `C-k` 删除行（但未将其放入剪贴板）
  - `C-l` 将插入点设为窗口中心
  - `C-b` 后退一个字符而不删除该字符（通常你也可以用方向键进行此操作）
  - `C-f` 前进一个字符而不删除该字符（通常你也可以用方向键进行此操作）
  - `C-p` 向上一行（通常你也可以用方向键进行此操作）
  - `C-d` 删除下一个字符

标准的键绑定（例如 `C-c` 复制和 `C-v` 粘贴）仍会有效。键绑定可在配置 IDLE 对话框中选择。

自动缩进

在一个代码块开头的语句之后，下一行会缩进 4 个空格符（在 Python Shell 窗口中是一个制表符）。在特定关键字之后（break, return 等），下一行将不再缩进。在开头的缩进中，按 `Backspace` 将会删除 4 个空格符。`Tab` 则会插入空格符（在 Python Shell 窗口中是一个制表符），具体数量取决于缩进宽度。目前，Tab 键按照 Tcl/Tk 的规定设置为四个空格符。

另请参阅*Format 菜单* 的缩进/取消缩进区的命令。

完成

当被请求并且可用时，将为模块名、类属性、函数或文件名提供补全。每次请求方法将显示包含现有名称的补全提示框。（例外情况参见下文的 Tab 补全。）对于任意提示框，要改变被补全的名称和提示框中被高亮的条目，可以通过输入和删除字符、按 `Up`, `Down`, `PageUp`, `PageDown`, `Home` 和 `End` 键；或是在提示框中单击。要关闭补全提示框可以通过 `Escape`, `Enter` 或按两次 `Tab` 键或是在提示框外单击。在提示框内双击则将执行选择并关闭。

有一种打开提示框的方式是输入一个关键字符并等待预设的一段间隔。此间隔默认为 2 秒；这可以在设置对话框中定制。（要防止自动弹出，可将时延设为一个很大的豪秒数值，例如 100000000。）对于导入的模块名或者类和函数属性，请输入'.'。对于根目录下的文件名，请在开头引号之后立即输入*os.sep* 或*os.altsep*。（在 Windows 下，可以先指定一个驱动器。）可通过输入目录名和分隔符来进入子目录。

除了等待，或是在提示框关闭之后，可以使用 Edit 菜单的 Show Completions 来立即打开一个补全提示框。默认的热键是 `C-space`。如果在打开提示框之前输入一某个名称的前缀，则将显示第一个匹配项或最接近的项。结果将与在提示框已显示之后输入前缀时相同。在一个引号之后执行 Show Completions 将会实例当前目录下的文件名而不是根目录下的。

在输入前缀后按 `Tab` 键的效果通常与 Show Completions 相同。（如果未输入前缀则为缩进。）但是，如果输入的前缀只有一个匹配项，则该匹配项会立即被添加到编辑器文本中而不打开补全提示框。

在字符串以外且开头不带'.' 地输入前缀并执行'Show Completions' 或按 `Tab` 键将打开一个包含关键字、内置名称和现有模块级名称的补全提示框。

当在编辑器（而非 Shell）中编辑代码时，可以通过运行你的代码并在此后不重启 Shell 来增加可用的模块级名称。这在文件顶部添加了导入语句之后会特别有用。这还会增加可用的属性补全。

Completion boxes initially exclude names beginning with '_' or, for modules, not included in '__all__'. The hidden names can be accessed by typing '_' after '.', either before or after the box is opened.

提示

当在一个 可用的函数名称之后输入 (时将自动显示一个调用提示。函数名称表达式可以包括点号和方括号索引操作。调用提示将保持打开直到它被点击、光标移出参数区、或是输入了) 。当光标位于某个定义的参数区时，可以在菜单中选择 Edit 的"Show Call Tip" 或是输入其快捷键来显示调用提示。

调用提示是由函数的签名和文档字符串到第一个空行或第五个非空行为止的内容组成的。（某些内置函数没有可访问的签名。）签名中的'/' 或'*' 指明其前面或后面的参数仅限以位置或名称（关键字）方式传入。具体细节可能会改变。

在 Shell 中，可访问的函数取决于有哪些模块已被导入用户进程，包括由 IDLE 本身导入的模块，以及哪些定义已被运行，以上均从最近的重启动开始算起。

例如，重启动 Shell 并输入 `itertools.count(`。将显示调用提示，因为 IDLE 出于自身需要已将 itertools 导入了用户进程。（此行为可能会改变。）输入 `turtle.write(` 则不显示任何提示。因为 IDLE 本身不会导入 turtle。菜单项和快捷键同样不会有任何反应。输入 `import turtle`。则在此之后，`turtle.write(` 将显示调用提示。

在编辑器中，import 语句在文件运行之前是没有效果的。在输入 import 语句之后、添加函数定义之后，或是打开一个现有文件之后可以先运行一下文件。

代码上下文

在一个包含 Python 代码的编辑器窗口内部，可以切换代码上下文以便显示或隐藏窗口顶部的面板。当显示时，此面板可以冻结代码块的开头行，例如以 `class`, `def` 或 `if` 关键字开头的行，这样的行在不显示时面板时可能离开视野。此面板的大小将根据需要扩展和收缩以显示当前层级的全部上下文，直至达到配置 IDLE 对话框中所定义的最大行数（默认为 15）。如果如果没有当前上下文行而此功能被启用，则将显示一个空行。点击上下文面板中的某一行将把该行移至编辑器顶部。

上下文面板的文本和背景颜色可在配置 IDLE 对话框的 Highlights 选项卡中进行配置。

Python Shell 窗口

通过 IDLE 的 Shell 可以输入、编辑和召回整条语句。大部分控制台和终端在同一时刻只能处理单个物理行。

当向 Shell 粘贴代码时，它并不会被编译和执行，直到按下 `Return` 键。在此之前可以先编辑所粘贴的代码。如果向 Shell 粘贴了多行代码，多条语句会被当作一条语句来编译，结果将引发*`SyntaxError`*。

之前小节中描述的编辑功能在交互式地输入代码时也可使用。IDLE 的 Shell 窗口还会响应以下按键。

- `C-c` 中断执行命令
- `C-d` 发送文件结束命令；如果在 >>> 提示符后按下会关闭窗口。
- `Alt-/` (扩展单词) 也有助于减少输入量

  历史命令

  - `Alt-p` 提取与你所输入键匹配的前一条命令。在 macOS 上请使用 `C-p`。
  - `Alt-n` 提取下一条命令。在 macOS 上请使用 `C-n`。
  - 在任意的前一条命令上按 `Return` 将提取该命令

文本颜色

IDLE 文本默认为白底黑字，但有特殊含义的文本将以彩色显示。对于 Shell 来说包括 Shell 输出，Shell 错误，用户输出和用户错误。对于 Shell 提示符下或编辑器中的 Python 代码来说则包括关键字，内置类和函数名称，`class` 和 `def` 之后的名称，字符串和注释等。对于任意文本窗口来说则包括光标（如果存在）、找到的文本（如果可能）和选定的文本。

广西着色是在背景上完成的，因此有时会看到非着色的文本。要改变颜色方案，请使用配置 IDLE 对话框的高亮选项卡。编辑器中的调试器断点行标记和弹出面板和对话框中的文本则是用户不可配置的。

25.10.3 启动和代码执行

在附带 `-s` 选项启用的情况下，IDLE 将会执行环境变量 `IDLESTARTUP` 或 `PYTHONSTARTUP` 所引用的文件。IDLE 会先检查 `IDLESTARTUP`；如果 `IDLESTARTUP` 存在则会运行被引用的文件。如果 `IDLESTARTUP` 不存在，则 IDLE 会检查 `PYTHONSTARTUP`。这些环境变量所引用的文件是存放经常被 IDLE Shell 所使用的函数，或者执行导入常用模块的 import 语句的便捷场所。

此外，`Tk` 也会在存在启动文件时加载它。请注意 Tk 文件会被无条件地加载。此附加文件名为 `.Idle.py` 并且会在用户的家目录下查找。此文件中的语句将在 Tk 的命名空间中执行，所以此文件不适用于导入要在 IDLE 的 Python Shell 中使用的函数。

命令行语法

```
idle.py [-c command] [-d] [-e] [-h] [-i] [-r file] [-s] [-t title] [-] [arg] ...

-c command  run command in the shell window
-d          enable debugger and open shell window
-e          open editor window
-h          print help message with legal combinations and exit
-i          open shell window
-r file     run file in shell window
-s          run $IDLESTARTUP or $PYTHONSTARTUP first, in shell window
-t title    set title of shell window
-           run stdin in shell (- must be last option before args)
```

如果有参数：

- 如果使用了 `-`, `-c` 或 `r`，则放在 `sys.argv[1:...]` 和 `sys.argv[0]` 中的所有参数都会被设为 `''`, `'-c'` 或 `'-r'`。不会打开任何编辑器窗口，即使是在选项对话框中的默认设置。
- 在其他情况下，参数为要打开编辑的文件而 `sys.argv` 反映的是传给 IDLE 本身的参数。

启动失败

IDLE 使用一个套接字在 IDLE GUI 进程和用户代码执行进程之间通信。当 Shell 启动或重启动时必须建立一个连接。(重启动会以一个内容为'RESTART' 的分隔行来标示)。如果用户进程无法连接到 GUI 进程，它通常会显示一个包含'cannot connect' 消息的 `Tk` 错误提示框来引导用户。随后将会退出程序。

有一个 Unix 系统专属的连接失败是由系统网络设置中错误配置的掩码规则导致的。当从一个终端启动 IDLE 时，用户将看到一条以 `** Invalid host:` 开头的消息。有效的值为 `127.0.0.1 (idlelib.rpc.LOCALHOST)`。用户可以在一个终端窗口输入 `tcpconnect -irv 127.0.0.1 6543` 并在另一个终端窗口中输入 `tcplisten <same args>` 来进行诊断。

导致连接失败的一个常见原因是用户创建的文件与标准库模块同名，例如 *random.py* 和 *tkinter.py*。当这样的文件与要运行的文件位于同一目录中时，IDLE 将无法导入标准库模块。可用的解决办法是重命名用户文件。

虽然现在已不太常见，但杀毒软件或防火墙程序也有可能会阻止连接。如果无法将此类程序设为允许连接，那么为了运行 IDLE 就必须将其关闭。允许这样的内部连接是安全的，因为数据在外部端口上不可见。一个类似的问题是错误的网络配置阻止了连接。

Python 的安装问题有时会使 IDLE 退出：存在多个版本时可能导致程序崩溃，或者单独安装时可能需要管理员权限。如果想要避免程序崩溃，或是不想以管理员身份运行，最简单的做法是完全卸载 Python 并重新安装。

有时会出现 pythonw.exe 僵尸进程问题。在 Windows 上，可以使用任务管理员来检查并停止该进程。有时由程序崩溃或键盘中断（control-C）所发起的重启动可能会出现连接失败。关闭错误提示框或使用 Shell 菜单中的 Restart Shell 可能会修复此类临时性错误。

当 IDLE 首次启动时，它会尝试读取 `~/.idlerc/` 中的用户配置文件（~ 是用户的家目录）。如果配置有问题，则应当显示一条错误消息。除随机磁盘错误之外，此类错误均可通过不手动编辑这些文件来避免。请使用 Options 菜单来打开配置对话框。一旦用户配置文件出现错误，最好的解决办法就是删除它并使用配置对话框重新设置。

如果 IDLE 退出时没有发出任何错误消息，并且它不是通过控制台启动的，请尝试通过控制台或终端 (`python -m idlelib`) 来启动它以查看是否会出现错误消息。

在基于 Unix 的系统上使用 tcl/tk 低于 `8.6.11` 的版本 (查看 `About IDLE`) 时特定字体的特定字符可能导致终端提示 tk 错误消息。这可能发生在启动 IDLE 编辑包此种字符的文件或是在之后输入此种字符的时候。如果无法升级 tcl/tk，可以重新配置 IDLE 来使用其他的字体。

运行用户代码

除了少量例外，使用 IDLE 执行 Python 代码的结果应当与使用默认方法，即在文本模式的系统控制台或终端窗口中直接通过 Python 解释器执行同样的代码相同。但是，不同的界面和操作有时会影响显示的结果。例如，`sys.modules` 初始时具有更多条目，而 `threading.active_count()` 将返回 2 而不是 1。

在默认情况下，IDLE 会在单独的 OS 进程中运行用户代码而不是在运行 Shell 和编辑器的用户界面进程中运行。在执行进程中，它会将 `sys.stdin`, `sys.stdout` 和 `sys.stderr` 替换为从 Shell 窗口获取输入并向其发送输出的对象。保存在 `sys.__stdin__`, `sys.__stdout__` 和 `sys.__stderr__` 中的原始值不会被改变，但可能会为 `None`。

将打印输出从一个进程发送到另一个进程中的文本部件要比打印到同一个进程中的系统终端慢。这在打印多个参数时将有更明显的影响，因为每个参数、每个分隔符和换行符对应的字符串都要单独发送。在开发中，这通常不算是问题，但如果希望能在 IDLE 中更快地打印，可以将想要显示的所有内容先格式化并拼接到一起然后打印单个字符串。格式字符串和*str.join()* 都可以被用于合并字段和文本行。

IDLE 的标准流替换不会被执行进程中创建的子进程所继承，不论它是由用户代码直接创建还是由 multiprocessing 之类的模块创建的。如果这样的子进程使用了 `input` 获取 sys.stdin 或者使用了 `print` 或 `write` 输出到 sys.stdout 或 sys.stderr，则应当在命令行窗口中启动 IDLE。这样二级子进程将会被附加到该窗口进行输出和输出。

如果 `sys` 被用户代码重置，例如使用了 `importlib.reload(sys)`，则 IDLE 的修改将丢失，来自键盘的输入和向屏幕的输出将无法正确运作。

当 Shell 获得焦点时，它将控制键盘与屏幕。这通常会保持透明，但一些直接访问键盘和屏幕的函数将会不起作用。这也包括那些确定是否有键被按下以及是哪个键被按下的系统专属函数。

在执行进程中运行的 IDLE 代码会向调用栈添加在其他情况下不存在的帧。IDLE 包装了 `sys.getrecursionlimit` 和 `sys.setrecursionlimit` 以减少额外栈帧的影响。

当用户代码直接或者通过调用 sys.exit 引发 SystemExit 时，IDLE 将返回 Shell 提示符而非完全退出。

Shell 中的用户输出

当一个程序输出文本时，结果将由相应的输出设备来确定。当 IDLE 执行用户代码时，`sys.stdout` 和 `sys.stderr` 会被连接到 IDLE Shell 的显示区。它的某些特性是从底层的 Tk Text 部件继承而来。其他特性则是程序所添加的。总之，Shell 被设计用于开发环境而非生产环境运行。

例如，Shell 绝不会丢弃输出。一个向 Shell 发送无限输出的程序将最终占满内存，导致内存错误。作为对比，某些系统文本模式窗口只会保留输出的最后 n 行。例如，Windows 控制台可由用户设置保留 1 至 9999 行，默认为 300 行。

A Tk Text widget, and hence IDLE's Shell, displays characters (codepoints) in the BMP (Basic Multilingual Plane) subset of Unicode. Which characters are displayed with a proper glyph and which with a replacement box depends on the operating system and installed fonts. Tab characters cause the following text to begin after the next tab stop. (They occur every 8 'characters'). Newline characters cause following text to appear on a new line. Other control characters are ignored or displayed as a space, box, or something else, depending on the operating system and font. (Moving the text cursor through such output with arrow keys may exhibit some surprising spacing behavior.)

```
>>> s = 'a\tb\a<\x02><\r>\bc\nd'   # Enter 22 chars.
>>> len(s)
14
>>> s  # Display repr(s)
'a\tb\x07<\x02><\r>\x08c\nd'
>>> print(s, end='')  # Display s as is.
# Result varies by OS and font.  Try it.
```

repr 函数会被用于表达式值的交互式回显。它将返回输入字符串的一个修改版本，其中的控制代码、部分 BMP 码位以及所有非 BMP 码位都将被替换为转义代码。如上面所演示的，它使用户可以辨识字符串中的字符，无论它们会如何显示。

普通的与错误的输出通常会在与代码输入和彼此之间保持区分 (显示于不同的行)。它们也会分别使用不同的高亮颜色。

对于 SyntaxError 回溯信息，表示检测到错误位置的正常'^' 标记被替换为带有代表错误的文本颜色高亮。当从文件运行的代码导致了其他异常时，用户可以右击回溯信息行在 IDLE 编辑器中跳转到相应的行。如有必要将打开相应的文件。

Shell 具有将输出行折叠为一个'Squeezed text' 标签的特殊功能。此功能将自动应用于超过 N 行的输出 (默认 N = 50)。N 可以在设置对话框中 General 页的 PyShell 区域中修改。行数更少的输出也可通过在输出上右击来折叠。此功能适用于过多的输出行数导致滚动操作变慢的情况。

已折叠的输出可通过双击该标签来原地展开。也可通过右击该标签将其发送到剪贴板或单独的查看窗口。

开发 tkinter 应用程序

IDLE 有意与标准 Python 保持区别以方便 tkinter 程序的开发。在标准 Python 中输入 `import tkinter as tk; root = tk.Tk()` 不会显示任何东西。在 IDLE 中输入同样的代码则会显示一个 tk 窗口。在标准 Python 中，还必须输入 `root.update()` 才会将窗口显示出来。IDLE 会在幕后执行同样的方法，每秒大约 20 次，即每隔大约 50 毫秒。下面输入 `b = tk.Button(root, text='button'); b.pack()`。在标准 Python 中仍然不会有任何可见的变化，直到输入 `root.update()`。

大多数 tkinter 程序都会运行 `root.mainloop()`，它通常直到 tk 应用被销毁时才会返回。如果程序是通过 `python -i` 或 IDLE 编辑器运行的，则 >>> Shell 提示符将直到 `mainloop()` 返回时才会出现，这时将结束程序的交互。

当通过 IDLE 编辑器运行 tkinter 程序时，可以注释掉 mainloop 调用。这样将立即回到 Shell 提示符并可与正在运行的应用程序交互。请记得当在标准 Python 中运行时重新启用 mainloop 调用。

在没有子进程的情况下运行

By default, IDLE executes user code in a separate subprocess via a socket, which uses the internal loopback interface. This connection is not externally visible and no data is sent to or received from the Internet. If firewall software complains anyway, you can ignore it.

如果创建套接字连接的尝试失败，IDLE 将会通知你。这样的失败可能是暂时性的，但是如果持续存在，问题可能是防火墙阻止了连接或某个系统配置错误。在问题得到解决之前，可以使用 -n 命令行开关来运行 IDLE。

如果 IDLE 启动时使用了 -n 命令行开关则它将在单个进程中运行并且将不再创建运行 RPC Python 执行服务器的子进程。这适用于 Python 无法在你的系统平台上创建子进程或 RPC 套接字接口的情况。但是，在这种模式下用户代码没有与 IDLE 本身相隔离。而且，当选择 Run/Run Module (F5) 时运行环境也不会重启。如果你的代码已被修改，你必须为受影响的模块执行 reload() 并重新导入特定的条目 (例如 from foo import baz) 才能让修改生效。出于这些原因，在可能的情况下最好还是使用默认的子进程来运行 IDLE。

3.4 版後已🄵用.

25.10.4 帮助和偏好

帮助源

Help 菜单项"IDLE Help" 会显示标准库参考中 IDLE 一章的带格式 HTML 版本。这些内容放在只读的 tkinter 文本窗口中，与在浏览器中看到的内容类似。可使用鼠标滚轮、滚动条或上下方向键来浏览文本。或是点击 TOC (Table of Contents) 按钮并在打开的选项框中选择一个节标题。

Help 菜单项"Python Docs" 会打开更丰富的帮助源，包括教程，通过 `docs.python.org/x.y` 来访问，其中'x.y' 是当前运行的 Python 版本。如果你的系统有此文档的离线副本 (这可能是一个安装选项)，则将打开这个副本。

选定的 URL 可以使用配置 IDLE 对话框的 General 选项卡随时在帮助菜单中添加或移除。

首选项设置

字体首选项、高亮、按键和通用首选项可通过 Option 菜单的配置 IDLE 项来修改。非默认的用户设置将保存在用户家目录下的 `.idlerc` 目录中。用户配置文件错误导致的问题可通过编辑或删除 `.idlerc` 中的一个或多个文件来解决。

在 Font 选项卡中，可以查看使用多种语言的多个字符的示例文本来了解字体或字号效果。可以编辑示例文本来添加想要的其他字符。请使用示例文本选择等宽字体。如果某些字符在 Shell 或编辑器中的显示有问题，可以将它们添加到示例文本的开头并尝试改变字号和字体。

在 Highlights 和 Keys 选项卡中，可以选择内置或自定义的颜色主题和按键集合。要将更新的内置颜色主题或按键集合与旧版 IDLE 一起使用，可以将其保存为新的自定义主题或按键集合就将可在旧版 IDLE 中使用。

macOS 上的 IDLE

在 System Preferences: Dock 中，可以将"Prefer tabs when opening documents" 设为"Always"。但是该设置不能兼容 IDLE 所使用的 tk/tkinter GUI 框架，并会使得部分 IDLE 特性失效。

扩展

IDLE 可以包含扩展插件。扩展插件的首选项可通过首选项对话框的 Extensions 选项卡来修改。请查看 idlelib 目录下 config-extensions.def 的开头来了解详情。目前唯一的扩展插件是 zzdummy，它也是一个测试用的示例。

CHAPTER 26

開發工具

本章中描述的各模块可帮你编写 Python 程序。例如，*pydoc* 模块接受一个模块并根据该模块的内容来生成文档。*doctest* 和*unittest* 这两个模块包含了用于编写单元测试的框架，并可用于自动测试所编写的代码，验证预期的输出是否产生。**2to3** 程序能够将 Python 2.x 源代码翻译成有效的 Python 3.x 源代码。

本章中描述的模块列表是：

26.1 typing --- 类型提示支持

3.5 版新加入.

源码： Lib/typing.py

備🄴： Python 运行时不强制执行函数和变量类型注解，但这些注解可用于类型检查器、IDE、静态检查器等第三方工具。

This module provides runtime support for type hints. The most fundamental support consists of the types *Any*, *Union*, *Callable*, *TypeVar*, and *Generic*. For a full specification, please see **PEP 484**. For a simplified introduction to type hints, see **PEP 483**.

下面的函数接收与返回的都是字符串，注解方式如下：

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

`greeting` 函数中，参数 `name` 的类型是*str*，返回类型也是*str*。子类型也可以当作参数。

New features are frequently added to the `typing` module. The typing_extensions package provides backports of these new features to older versions of Python.

26.1.1 Relevant PEPs

Since the initial introduction of type hints in **PEP 484** and **PEP 483**, a number of PEPs have modified and enhanced Python's framework for type annotations. These include:

- **PEP 526: Syntax for Variable Annotations** *Introducing* syntax for annotating variables outside of function definitions, and `ClassVar`
- **PEP 544: Protocols: Structural subtyping (static duck typing)** *Introducing* `Protocol` and the `@runtime_checkable` decorator
- **PEP 585: Type Hinting Generics In Standard Collections** *Introducing* `types.GenericAlias` and the ability to use standard library classes as *generic types*
- **PEP 586: Literal Types** *Introducing* `Literal`
- **PEP 589: TypedDict: Type Hints for Dictionaries with a Fixed Set of Keys** *Introducing* `TypedDict`
- **PEP 591: Adding a final qualifier to typing** *Introducing* `Final` and the `@final` decorator
- **PEP 593: Flexible function and variable annotations** *Introducing* `Annotated`

26.1.2 类型别名

把类型赋给别名，就可以定义类型别名。本例中，`Vector` 和 `list[float]` 相同，可互换：

```
Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# typechecks; a list of floats qualifies as a Vector.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

类型别名适用于简化复杂的类型签名。例如：

```
from collections.abc import Sequence

ConnectionOptions = dict[str, str]
Address = tuple[str, int]
Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# The static type checker will treat the previous type signature as
# being exactly equivalent to this one.
def broadcast_message(
        message: str,
        servers: Sequence[tuple[tuple[str, int], dict[str, str]]]) -> None:
    ...
```

注意，`None` 是一种类型提示特例，已被 `type(None)` 取代。

26.1.3 NewType

Use the *NewType()* helper to create distinct types:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

静态类型检查器把新类型当作原始类型的子类，这种方式适用于捕捉逻辑错误：

```
def get_user_name(user_id: UserId) -> str:
    ...

# typechecks
user_a = get_user_name(UserId(42351))

# does not typecheck; an int is not a UserId
user_b = get_user_name(-1)
```

UserId 类型的变量可执行所有 int 操作，但返回结果都是 int 类型。这种方式允许在预期 int 时传入 UserId，还能防止意外创建无效的 UserId：

```
# 'output' is of type 'int', not 'UserId'
output = UserId(23413) + UserId(54341)
```

Note that these checks are enforced only by the static type checker. At runtime, the statement `Derived = NewType('Derived', Base)` will make `Derived` a callable that immediately returns whatever parameter you pass it. That means the expression `Derived(some_value)` does not create a new class or introduce any overhead beyond that of a regular function call.

更确切地说，在运行时，`some_value is Derived(some_value)` 表达式总为 True。

也就是说，不能创建 Derived 的子类型，因为，在运行时，它是标识函数，不是真正的类型：

```
from typing import NewType

UserId = NewType('UserId', int)

# Fails at runtime and does not typecheck
class AdminUserId(UserId): pass
```

然而，我们可以在" 派生的" NewType 的基础上创建一个*NewType*。

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

同时，ProUserId 的类型检查也可以按预期执行。

详见 **PEP 484**。

備🄵: 回顾上文，类型别名声明了两种彼此 等价的类型。`Alias = Original` 时，静态类型检查器认为 Alias 与 Original 完全等价。这种方式适用于简化复杂类型签名。

反之，NewType 声明把一种类型当作另一种类型的子类型。Derived = NewType('Derived', Original) 时，静态类型检查器把 Derived 当作 Original 的子类，即，Original 类型的值不能用在预期 Derived 类型的位置。这种方式适用于以最小运行时成本防止逻辑错误。

3.5.2 版新加入.

26.1.4 可调对象（Callable）

预期特定签名回调函数的框架可以用 Callable[[Arg1Type, Arg2Type], ReturnType] 实现类型提示。

例如：

```
from collections.abc import Callable

def feeder(get_next_item: Callable[[], str]) -> None:
    # Body

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    # Body

async def on_update(value: str) -> None:
    # Body
callback: Callable[[str], Awaitable[None]] = on_update
```

无需指定调用签名，用省略号字面量替换类型提示里的参数列表：Callable[..., ReturnType]，就可以声明可调对象的返回类型。

26.1.5 泛型（Generic）

容器中，对象的类型信息不能以泛型方式静态推断，因此，抽象基类扩展支持下标，用于表示容器元素的预期类型。

```
from collections.abc import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                    overrides: Mapping[str, str]) -> None: ...
```

Generics can be parameterized by using a factory available in typing called *TypeVar*.

```
from collections.abc import Sequence
from typing import TypeVar

T = TypeVar('T')      # Declare type variable

def first(l: Sequence[T]) -> T:   # Generic function
    return l[0]
```

26.1.6 用户定义的泛型类型

用户定义的类可以定义为泛型类。

```
from typing import TypeVar, Generic
from logging import Logger

T = TypeVar('T')

class LoggedVar(Generic[T]):
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

`Generic[T]` 是定义类 `LoggedVar` 的基类，该类使用单类型参数 `T`。在该类体内，`T` 是有效的类型。

The *Generic* base class defines `__class_getitem__()` so that `LoggedVar[t]` is valid as a type:

```
from collections.abc import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

A generic type can have any number of type variables. All varieties of *TypeVar* are permissible as parameters for a generic type:

```
from typing import TypeVar, Generic, Sequence

T = TypeVar('T', contravariant=True)
B = TypeVar('B', bound=Sequence[bytes], covariant=True)
S = TypeVar('S', int, str)

class WeirdTrio(Generic[T, B, S]):
    ...
```

Generic 类型变量的参数应各不相同。下列代码就是无效的：

```
from typing import TypeVar, Generic
...

T = TypeVar('T')

class Pair(Generic[T, T]):   # INVALID
    ...
```

Generic 支持多重继承：

```
from collections.abc import Sized
from typing import TypeVar, Generic

T = TypeVar('T')

class LinkedList(Sized, Generic[T]):
    ...
```

继承自泛型类时，可以修正某些类型变量：

```
from collections.abc import Mapping
from typing import TypeVar

T = TypeVar('T')

class MyDict(Mapping[str, T]):
    ...
```

比如，本例中 MyDict 调用的单参数，T。

未指定泛型类的类型参数时，每个位置的类型都预设为*Any*。下例中，MyIterable 不是泛型，但却隐式继承了 Iterable[Any]：

```
from collections.abc import Iterable

class MyIterable(Iterable): # Same as Iterable[Any]
```

还支持用户定义的泛型类型别名。例如：

```
from collections.abc import Iterable
from typing import TypeVar, Union
S = TypeVar('S')
Response = Union[Iterable[S], int]

# Return type here is same as Union[Iterable[str], int]
def response(query: str) -> Response[str]:
    ...

T = TypeVar('T', int, float, complex)
Vec = Iterable[tuple[T, T]]

def inproduct(v: Vec[T]) -> T: # Same as Iterable[tuple[T, T]]
    return sum(x*y for x, y in v)
```

3.7 版更變: *Generic* 不再支持自定义元类。

抽象基类可作为用户定义的泛型类的基类，且不会与元类冲突。现已不再支持泛型元类。参数化泛型的输出结果会被缓存，typing 模块的大多数类型都可哈希、可进行等价对比。

26.1.7 Any 类型

Any 是一种特殊的类型。静态类型检查器认为所有类型均与*Any* 兼容，同样，*Any* 也与所有类型兼容。

也就是说，可对*Any* 类型的值执行任何操作或方法调用，并赋值给任意变量：

```
from typing import Any

a: Any = None
a = []          # OK
a = 2           # OK

s: str = ''
s = a           # OK

def foo(item: Any) -> int:
    # Typechecks; 'item' could be any type,
    # and that type might have a 'bar' method
    item.bar()
    ...
```

注意，*Any* 类型的值赋给更精确的类型时，不执行类型检查。例如，把 a 赋给 s，在运行时，即便 s 已声明为*str* 类型，但接收*int* 值时，静态类型检查器也不会报错。

此外，未指定返回值与参数类型的函数，都隐式地默认使用*Any*：

```
def legacy_parser(text):
    ...
    return data

# A static type checker will treat the above
# as having the same signature as:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

需要混用动态与静态类型代码时，此操作把*Any* 当作 应急出口。

Any 和*object* 的区别。与*Any* 相似，所有类型都是*object* 的子类型。然而，与*Any* 不同，object 不可逆：*object* 不是其它类型的子类型。

就是说，值的类型是*object* 时，类型检查器几乎会拒绝所有对它的操作，并且，把它赋给更精确的类型变量（或返回值）属于类型错误。例如：

```
def hash_a(item: object) -> int:
    # Fails; an object does not have a 'magic' method.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Typechecks
    item.magic()
    ...

# Typechecks, since ints and strs are subclasses of object
hash_a(42)
hash_a("foo")

# Typechecks, since Any is compatible with all types
```

(下页继续)

(繼續上一頁)

```
hash_b(42)
hash_b("foo")
```

使用*object*，说明值能以类型安全的方式转为任何类型。使用*Any*，说明值是动态类型。

26.1.8 名义子类型 vs 结构子类型

Initially **PEP 484** defined the Python static type system as using *nominal subtyping*. This means that a class `A` is allowed where a class `B` is expected if and only if `A` is a subclass of `B`.

此项要求以前也适用于抽象基类，例如，*Iterable*。这种方式的问题在于，定义类时必须显式说明，既不 Pythonic，也不是动态类型式 Python 代码的惯用写法。例如，下列代码就遵从了 **PEP 484** 的规范：

```
from collections.abc import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

PEP 544 允许用户在类定义时不显式说明基类，从而解决了这一问题，静态类型检查器隐式认为 `Bucket` 既是 `Sized` 的子类型，又是 `Iterable[int]` 的子类型。这就是 *结构子类型*（又称为静态鸭子类型）：

```
from collections.abc import Iterator, Iterable

class Bucket:  # Note: no base classes
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket())  # Passes type check
```

此外，结构子类型的优势在于，通过继承特殊类*Protocol*，用户可以定义新的自定义协议（见下文中的例子）。

26.1.9 模块内容

本模块定义了下列类、函数和修饰器。

備🄵： 本模块定义了一些类型，作为标准库中已有的类的子类，从而可以让*Generic*支持 `[]` 中的类型变量。Python 3.9 中，这些标准库的类已支持 `[]`，因此，这些类型就变得冗余了。

Python 3.9 弃用了这些冗余类型，但解释器并未提供相应的弃用警告。标记弃用类型的工作留待支持 Python 3.9 及以上版本的类型检查器实现。

Python 3.9.0 发布五年后的首个 Python 发行版将从*typing*模块中移除这些弃用类型。详见 **PEP 585** 《标准集合的类型提示泛型》。

特殊类型原语

特殊类型

这些类型可用于类型注解，但不支持 `[]`。

typing.**Any**
不受限的特殊类型。

- 所有类型都与*Any* 兼容。
- *Any* 与所有类型都兼容。

typing.**NoReturn**
标记没有返回值的函数的特殊类型。例如：

```
from typing import NoReturn

def stop() -> NoReturn:
    raise RuntimeError('no way')
```

3.5.4 版新加入.

3.6.2 版新加入.

特殊形式

可用于类型注解，且支持 `[]` ，每种形式都有其独特的句法。

typing.**Tuple**
元组类型；`Tuple[X, Y]` 是二项元组类型，第一个元素的类型是 X，第二个元素的类型是 Y。空元组的类型可写为 `Tuple[()]`。

例：`Tuple[T1, T2]` 是二项元组，类型变量分别为 T1 和 T2。`Tuple[int, float, str]` 是由整数、浮点数、字符串组成的三项元组。

可用省略号字面量指定同质变长元组，例如，`Tuple[int, ...]` 。*Tuple* 与 `Tuple[Any, ...]` 等价，也与*tuple* 等价。

3.9 版後已🄵用: *builtins.tuple* 现已支持 `[]`。详见 **PEP 585** 与*GenericAlias 类型*。

typing.**Union**
联合类型；`Union[X, Y]` 的意思是，非 X 即 Y。

可用 `Union[int, str]` 等形式定义联合类型。具体如下：

- 参数必须是某种类型，且至少有一个。
- 联合类型之联合类型会被展平，例如：

```
Union[Union[int, str], float] == Union[int, str, float]
```

- 单参数之联合类型就是该参数自身，例如：

```
Union[int] == int  # The constructor actually returns int
```

- 冗余的参数会被跳过，例如：

```
Union[int, str, int] == Union[int, str]
```

- 比较联合类型，不涉及参数顺序，例如：

```
Union[int, str] == Union[str, int]
```

- 联合类型不能作为子类，也不能实例化。
- 不支持 `Union[X][Y]` 这种写法。
- `Optional[X]` 是 `Union[X, None]` 的缩写。

3.7 版更變: 在运行时，不要移除联合类型中的显式子类。

typing.**Optional**
可选类型。

`Optional[X]` 等价于 `Union[X, None]` 。

注意，可选类型与含默认值的可选参数不同。含默认值的可选参数不需要在类型注解上添加 `Optional` 限定符，因为它仅是可选的。例如：

```
def foo(arg: int = 0) -> None:
    ...
```

另一方面，显式应用 `None` 值时，不管该参数是否可选，`Optional` 都适用。例如：

```
def foo(arg: Optional[int] = None) -> None:
    ...
```

typing.**Callable**
可调类型；`Callable[[int], str]` 是把（int）转为 str 的函数。

下标句法必须与参数列表和返回类型这两个值一起使用。参数列表只能是类型列表或省略号；返回类型只能是单一类型。

没有说明可选参数或关键字参数的句法；这类函数类型很少用作回调类型。`Callable[..., ReturnType]`（省略号字面量）可用于为接受任意数量参数，并返回 `ReturnType` 的可调对象提供类型提示。纯*Callable* 等价于 `Callable[..., Any]`，进而等价于*collections.abc.Callable* 。

3.9 版後已🄵用: *collections.abc.Callable* 现已支持 `[]`。详见 **PEP 585** 与*GenericAlias 类型*。

class typing.**Type**(*Generic[CT_co]*)
用 `C` 注解的变量可以接受类型 `C` 的值。反之，用 `Type[C]` 注解的变量可以接受类自身的值—准确地说，是接受 `C` 的 类对象，例如：

```
a = 3         # Has type 'int'
b = int       # Has type 'Type[int]'
c = type(a)   # Also has type 'Type[int]'
```

注意，`Type[C]` 为协变量：

```
class User: ...
class BasicUser(User): ...
class ProUser(User): ...
class TeamUser(User): ...

# Accepts User, BasicUser, ProUser, TeamUser, ...
def make_new_user(user_class: Type[User]) -> User:
    # ...
    return user_class()
```

`Type[C]` 为协变量的意思是指，`C` 的所有子类都应使用与 `C` 相同的构造器签名及类方法签名。类型检查器应标记违反此项规定的内容，但也应允许符合指定基类构造器调用的子类进行构造器调用。**PEP 484** 修订版将来可能会调整类型检查器对这种特例的处理方式。

Type 合法的参数仅有类、*Any*、*类型变量* 以及上述类型的联合类型。例如：

```
def new_non_team_user(user_class: Type[Union[BasicUser, ProUser]]): ...
```

`Type[Any]` 等价于 `Type`，进而等价于 Python 元类架构的根基，`type`。

3.5.2 版新加入.

3.9 版後已𝔽用: *builtins.type* 现已支持 `[]`。详见 **PEP 585** 与*GenericAlias 类型*。

typing.**Literal**

表示类型检查器对应变量或函数参数的值等价于给定字面量（或多个字面量之一）的类型。例如：

```
def validate_simple(data: Any) -> Literal[True]:  # always returns True
    ...

MODE = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: MODE) -> str:
    ...

open_helper('/some/path', 'r')  # Passes type check
open_helper('/other/path', 'typo')  # Error in type checker
```

`Literal[...]` 不能创建子类。在运行时，任意值均可作为 `Literal[...]` 的类型参数，但类型检查器可以对此加以限制。字面量类型详见 **PEP 586** 。

3.8 版新加入.

3.9.1 版更變: `Literal` 现在能去除形参的重复。`Literal` 对象的相等性比较不再依赖顺序。现在如果有某个参数不为*hashable*，`Literal` 对象在相等性比较期间将引发*TypeError*。

typing.**ClassVar**

标记类变量的特殊类型构造器。

如 **PEP 526** 所述，打包在 ClassVar 内的变量注解是指，给定属性应当用作类变量，而不应设置在类实例上。用法如下：

```
class Starship:
    stats: ClassVar[dict[str, int]] = {} # class variable
    damage: int = 10                     # instance variable
```

ClassVar 仅接受类型，也不能使用下标。

ClassVar 本身不是类，不应用于*isinstance()* 或*issubclass()*。*ClassVar* 不改变 Python 运行时行为，但可以用于第三方类型检查器。例如，类型检查器会认为以下代码有错：

```
enterprise_d = Starship(3000)
enterprise_d.stats = {} # Error, setting class variable on instance
Starship.stats = {}     # This is OK
```

3.5.3 版新加入.

typing.**Final**

告知类型检查器某名称不能再次赋值或在子类中重写的特殊类型构造器。例如：

```
MAX_SIZE: Final = 9000
MAX_SIZE += 1  # Error reported by type checker

class Connection:
    TIMEOUT: Final[int] = 10

class FastConnector(Connection):
    TIMEOUT = 1  # Error reported by type checker
```

这些属性没有运行时检查。详见 **PEP 591**。

3.8 版新加入.

typing.**Annotated**

A type, introduced in **PEP 593** (`Flexible function and variable annotations`), to decorate existing types with context-specific metadata (possibly multiple pieces of it, as `Annotated` is variadic). Specifically, a type `T` can be annotated with metadata `x` via the typehint `Annotated[T, x]`. This metadata can be used for either static analysis or at runtime. If a library (or tool) encounters a typehint `Annotated[T, x]` and has no special logic for metadata `x`, it should ignore it and simply treat the type as `T`. Unlike the `no_type_check` functionality that currently exists in the `typing` module which completely disables typechecking annotations on a function or a class, the `Annotated` type allows for both static typechecking of `T` (which can safely ignore `x`) together with runtime access to `x` within a specific application.

毕竟，如何解释注解（如有）由处理 `Annotated` 类型的工具/库负责。工具/库处理 `Annotated` 类型时，扫描所有注解以确定是否需要进行处理（例如，使用 `isinstance()`）。

工具/库不支持注解，或遇到未知注解时，应忽略注解，并把注解类型当作底层类型。

是否允许客户端在一个类型上使用多个注解，以及如何合并这些注解，由处理注解的工具决定。

`Annotated` 类型支持把多个相同（或不同）的单个（或多个）类型注解置于任意节点。因此，使用这些注解的工具/库要负责处理潜在的重复项。例如，执行值范围分析时，应允许以下操作：

```
T1 = Annotated[int, ValueRange(-10, 5)]
T2 = Annotated[T1, ValueRange(-20, 3)]
```

传递 `include_extras=True` 至*`get_type_hints()`*，即可在运行时访问额外的注解。

语义详情：

- `Annotated` 的第一个参数必须是有效类型。
- 支持多个类型标注（`Annotated` 支持可变参数）：

  ```
  Annotated[int, ValueRange(3, 10), ctype("char")]
  ```

- 调用 `Annotated` 至少要有两个参数（`Annotated[int]` 是无效的）
- 注解的顺序会被保留，且影响等价检查：

  ```
  Annotated[int, ValueRange(3, 10), ctype("char")] != Annotated[
      int, ctype("char"), ValueRange(3, 10)
  ]
  ```

- 嵌套 `Annotated` 类型会被展平，元数据从最内层注解依序展开：

  ```
  Annotated[Annotated[int, ValueRange(3, 10)], ctype("char")] == Annotated[
      int, ValueRange(3, 10), ctype("char")
  ]
  ```

- 不移除注解重复项：

```
Annotated[int, ValueRange(3, 10)] != Annotated[
    int, ValueRange(3, 10), ValueRange(3, 10)
]
```

- `Annotated` 可用于嵌套或泛型别名：

```
T = TypeVar('T')
Vec = Annotated[list[tuple[T, T]], MaxLen(10)]
V = Vec[int]

V == Annotated[list[tuple[int, int]], MaxLen(10)]
```

3.9 版新加入.

构建泛型类型

以下内容是创建泛型类型的基石，但不在注解内使用。

class typing.**Generic**

用于泛型类型的抽象基类。

泛型类型一般通过继承含一个或多个类型变量的类实例进行声明。例如，泛型映射类型定义如下：

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
        # Etc.
```

该类的用法如下：

```
X = TypeVar('X')
Y = TypeVar('Y')

def lookup_name(mapping: Mapping[X, Y], key: X, default: Y) -> Y:
    try:
        return mapping[key]
    except KeyError:
        return default
```

class typing.**TypeVar**

类型变量。

用法：

```
T = TypeVar('T')  # Can be anything
S = TypeVar('S', bound=str)  # Can be any subtype of str
A = TypeVar('A', str, bytes)  # Must be exactly str or bytes
```

类型变量主要是为静态类型检查器提供支持，用于泛型类型与泛型函数定义的参数。有关泛型类型，详见*Generic*。泛型函数的写法如下：

```
def repeat(x: T, n: int) -> Sequence[T]:
    """Return a list containing n references to x."""
    return [x]*n
```

(下页继续)

(繼續上一頁)

```
def print_capitalized(x: S) -> S:
    """Print x capitalized, and return x."""
    print(x.capitalize())
    return x


def concatenate(x: A, y: A) -> A:
    """Add two strings or bytes objects together."""
    return x + y
```

Note that type variables can be *bound*, *constrained*, or neither, but cannot be both bound *and* constrained.

Constrained type variables and bound type variables have different semantics in several important ways. Using a *constrained* type variable means that the `TypeVar` can only ever be solved as being exactly one of the constraints given:

```
a = concatenate('one', 'two')  # Ok, variable 'a' has type 'str'
b = concatenate(StringSubclass('one'), StringSubclass('two'))  # Inferred type of␣
↪variable 'b' is 'str',
                                                               # despite
↪'StringSubclass' being passed in
c = concatenate('one', b'two')  # error: type variable 'A' can be either 'str' or
↪'bytes' in a function call, but not both
```

Using a *bound* type variable, however, means that the `TypeVar` will be solved using the most specific type possible:

```
print_capitalized('a string')  # Ok, output has type 'str'

class StringSubclass(str):
    pass

print_capitalized(StringSubclass('another string'))  # Ok, output has type
↪'StringSubclass'
print_capitalized(45)  # error: int is not a subtype of str
```

Type variables can be bound to concrete types, abstract types (ABCs or protocols), and even unions of types:

```
U = TypeVar('U', bound=str|bytes)  # Can be any subtype of the union str|bytes
V = TypeVar('V', bound=SupportsAbs)  # Can be anything with an __abs__ method
```

Bound type variables are particularly useful for annotating *classmethods* that serve as alternative constructors. In the following example (© Raymond Hettinger), the type variable `C` is bound to the `Circle` class through the use of a forward reference. Using this type variable to annotate the `with_circumference` classmethod, rather than hardcoding the return type as `Circle`, means that a type checker can correctly infer the return type even if the method is called on a subclass:

```
import math

C = TypeVar('C', bound='Circle')

class Circle:
    """An abstract circle"""

    def __init__(self, radius: float) -> None:
        self.radius = radius
```

(下页继续)

(繼續上一頁)

```
    # Use a type variable to show that the return type
    # will always be an instance of whatever ``cls`` is
    @classmethod
    def with_circumference(cls: type[C], circumference: float) -> C:
        """Create a circle with the specified circumference"""
        radius = circumference / (math.pi * 2)
        return cls(radius)


class Tire(Circle):
    """A specialised circle (made out of rubber)"""

    MATERIAL = 'rubber'


c = Circle.with_circumference(3)  # Ok, variable 'c' has type 'Circle'
t = Tire.with_circumference(4)  # Ok, variable 't' has type 'Tire' (not 'Circle')
```

在运行时，`isinstance(x, T)` 会触发*TypeError* 异常。一般而言，*isinstance()* 和*issubclass()* 不应与类型搭配使用。

Type variables may be marked covariant or contravariant by passing `covariant=True` or `contravariant=True`. See **PEP 484** for more details. By default, type variables are invariant.

`typing.`**AnyStr**

`AnyStr` is a *constrained type variable* defined as `AnyStr = TypeVar('AnyStr', str, bytes)`.

这里指的是，它可以接受任意同类字符串，但不支持混用不同类别的字符串。例如：

```
def concat(a: AnyStr, b: AnyStr) -> AnyStr:
    return a + b

concat(u"foo", u"bar")  # Ok, output has type 'unicode'
concat(b"foo", b"bar")  # Ok, output has type 'bytes'
concat(u"foo", b"bar")  # Error, cannot mix unicode and bytes
```

class `typing.`**Protocol**(*Generic*)

Protocol 类的基类。Protocol 类的定义如下：

```
class Proto(Protocol):
    def meth(self) -> int:
        ...
```

这些类主要与静态类型检查器搭配使用，用来识别结构子类型（静态鸭子类型），例如：

```
class C:
    def meth(self) -> int:
        return 0

def func(x: Proto) -> int:
    return x.meth()

func(C())  # Passes static type check
```

详见 **PEP 544**。Protocol 类用*runtime_checkable()* （见下文）装饰，忽略类型签名，仅检查给定属性是否存在，充当简要的运行时协议。

Protocol 类可以是泛型，例如：

```
class GenProto(Protocol[T]):
    def meth(self) -> T:
        ...
```

3.8 版新加入.

@typing.**runtime_checkable**

用于把 Protocol 类标记为运行时协议。

该协议可以与*isinstance()* 和*issubclass()* 一起使用。应用于非协议的类时，会触发*TypeError*。该指令支持简易结构检查，与*collections.abc* 的*Iterable* 非常类似，只擅长做一件事。例如：

```
@runtime_checkable
class Closable(Protocol):
    def close(self): ...

assert isinstance(open('/some/file'), Closable)
```

備🄵: *runtime_checkable()* 只检查所需方法是否存在，但却不检查类型签名！例如，*builtins.complex* 支持 `__float__()`，因此，它能通过*SupportsFloat* 的*issubclass()* 检查。然而，`complex.__float__` 方法其实只是为了触发含更多信息的*TypeError*。

3.8 版新加入.

其他特殊指令

这些特殊指令是声明类型的基石，但不在注解内使用。

class typing.**NamedTuple**

collections.namedtuple() 的类型版本。

用法：

```
class Employee(NamedTuple):
    name: str
    id: int
```

相当于：

```
Employee = collections.namedtuple('Employee', ['name', 'id'])
```

为字段提供默认值，要在类体内赋值：

```
class Employee(NamedTuple):
    name: str
    id: int = 3

employee = Employee('Guido')
assert employee.id == 3
```

带默认值的字段必须在不带默认值的字段后面。

The resulting class has an extra attribute `__annotations__` giving a dict that maps the field names to the field types. (The field names are in the `_fields` attribute and the default values are in the `_field_defaults` attribute, both of which are part of the *namedtuple()* API.)

`NamedTuple` 子类也支持文档字符串与方法：

```
class Employee(NamedTuple):
    """Represents an employee."""
    name: str
    id: int = 3

    def __repr__(self) -> str:
        return f'<Employee {self.name}, id={self.id}>'
```

反向兼容用法：

```
Employee = NamedTuple('Employee', [('name', str), ('id', int)])
```

3.6 版更變: 添加了对 **PEP 526** 中变量注解句法的支持。

3.6.1 版更變: 添加了对默认值、方法、文档字符串的支持。

3.8 版更變: `_field_types` 和 `__annotations__` 属性现已使用常规字典，不再使用 `OrderedDict` 实例。

3.9 版更變: 移除了 `_field_types` 属性，改用具有相同信息，但更标准的 `__annotations__` 属性。

`typing.`**`NewType`**(*name*, *tp*)

用于为类型检查器标明不同类型的辅助函数，详见*NewType*。在运行时，它返回一个返回其参数的函数。用法如下：

```
UserId = NewType('UserId', int)
first_user = UserId(1)
```

3.5.2 版新加入.

`class` `typing.`**`TypedDict`**(*dict*)

把类型提示添加至字典的特殊构造器。在运行时，它是纯*dict*。

`TypedDict` 声明一个字典类型，该类型预期所有实例都具有一组键集，其中，每个键都与对应类型的值关联。运行时不检查此预期，而是由类型检查器强制执行。用法如下：

```
class Point2D(TypedDict):
    x: int
    y: int
    label: str

a: Point2D = {'x': 1, 'y': 2, 'label': 'good'}  # OK
b: Point2D = {'z': 3, 'label': 'bad'}           # Fails type check

assert Point2D(x=1, y=2, label='first') == dict(x=1, y=2, label='first')
```

To allow using this feature with older versions of Python that do not support **PEP 526**, `TypedDict` supports two additional equivalent syntactic forms:

- Using a literal *dict* as the second argument:

```
Point2D = TypedDict('Point2D', {'x': int, 'y': int, 'label': str})
```

- Using keyword arguments:

```
Point2D = TypedDict('Point2D', x=int, y=int, label=str)
```

The functional syntax should also be used when any of the keys are not valid identifiers, for example because they are keywords or contain hyphens. Example:

```
# raises SyntaxError
class Point2D(TypedDict):
    in: int  # 'in' is a keyword
    x-y: int  # name with hyphens

# OK, functional syntax
Point2D = TypedDict('Point2D', {'in': int, 'x-y': int})
```

By default, all keys must be present in a `TypedDict`. It is possible to override this by specifying totality. Usage:

```
class Point2D(TypedDict, total=False):
    x: int
    y: int

# Alternative syntax
Point2D = TypedDict('Point2D', {'x': int, 'y': int}, total=False)
```

This means that a `Point2D` `TypedDict` can have any of the keys omitted. A type checker is only expected to support a literal `False` or `True` as the value of the `total` argument. `True` is the default, and makes all items defined in the class body required.

It is possible for a `TypedDict` type to inherit from one or more other `TypedDict` types using the class-based syntax. Usage:

```
class Point3D(Point2D):
    z: int
```

`Point3D` has three items: `x`, `y` and `z`. It is equivalent to this definition:

```
class Point3D(TypedDict):
    x: int
    y: int
    z: int
```

A `TypedDict` cannot inherit from a non-`TypedDict` class, notably including *Generic*. For example:

```
class X(TypedDict):
    x: int

class Y(TypedDict):
    y: int

class Z(object): pass  # A non-TypedDict class

class XY(X, Y): pass  # OK

class XZ(X, Z): pass  # raises TypeError

T = TypeVar('T')
class XT(X, Generic[T]): pass  # raises TypeError
```

A `TypedDict` can be introspected via `__annotations__`, *__total__*, *__required_keys__*, and *__optional_keys__*.

__total__

`Point2D.__total__` gives the value of the `total` argument. Example:

```
>>> from typing import TypedDict
>>> class Point2D(TypedDict): pass
>>> Point2D.__total__
True
>>> class Point2D(TypedDict, total=False): pass
>>> Point2D.__total__
False
>>> class Point3D(Point2D): pass
>>> Point3D.__total__
True
```

__required_keys__

__optional_keys__

`Point2D.__required_keys__` and `Point2D.__optional_keys__` return *frozenset* objects containing required and non-required keys, respectively. Currently the only way to declare both required and non-required keys in the same `TypedDict` is mixed inheritance, declaring a `TypedDict` with one value for the `total` argument and then inheriting it from another `TypedDict` with a different value for `total`. Usage:

```
>>> class Point2D(TypedDict, total=False):
...     x: int
...     y: int
...
>>> class Point3D(Point2D):
...     z: int
...
>>> Point3D.__required_keys__ == frozenset({'z'})
True
>>> Point3D.__optional_keys__ == frozenset({'x', 'y'})
True
```

更多示例与 `TypedDict` 的详细规则，详见 **PEP 589**。

3.8 版新加入.

泛型具象容器

对应的内置类型

class typing.**Dict**(*dict, MutableMapping[KT, VT]*)

dict 的泛型版本。适用于注解返回类型。注解参数时，最好使用*Mapping* 等抽象容器类型。

该类型用法如下：

```
def count_words(text: str) -> Dict[str, int]:
    ...
```

3.9 版後已🄴用: *builtins.dict* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**List**(*list, MutableSequence[T]*)

list 的泛型版本。适用于注解返回类型。注解参数时，最好使用*Sequence* 或*Iterable* 等抽象容器类型。

该类型用法如下：

```
T = TypeVar('T', int, float)

def vec2(x: T, y: T) -> List[T]:
    return [x, y]

def keep_positives(vector: Sequence[T]) -> List[T]:
    return [item for item in vector if item > 0]
```

3.9 版後已🄵用: *builtins.list* 现已支持 `[]`。详见 **PEP 585** 与*GenericAlias 类型*。

class typing.**Set**(*set, MutableSet[T]*)
builtins.set 的泛型版本。适用于注解返回类型。注解参数时，最好使用*AbstractSet* 等抽象容器类型。

3.9 版後已🄵用: *builtins.set* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**FrozenSet**(*frozenset, AbstractSet[T_co]*)
builtins.frozenset 的泛型版本。

3.9 版後已🄵用: *builtins.frozenset* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

備🄵: *Tuple* 是一种特殊形式。

---

collections 对应类型

class typing.**DefaultDict**(*collections.defaultdict, MutableMapping[KT, VT]*)
collections.defaultdict 的泛型版本。

3.5.2 版新加入.

3.9 版後已🄵用: *collections.defaultdict* 现已支持 `[]`。详见 **PEP 585** 与*GenericAlias 类型*。

class typing.**OrderedDict**(*collections.OrderedDict, MutableMapping[KT, VT]*)
collections.OrderedDict 的泛型版本。

3.7.2 版新加入.

3.9 版後已🄵用: *collections.OrderedDict* 现已支持 `[]`。详见 **PEP 585** 与*GenericAlias 类型*。

class typing.**ChainMap**(*collections.ChainMap, MutableMapping[KT, VT]*)
collections.ChainMap 的泛型版本。

3.5.4 版新加入.

3.6.1 版新加入.

3.9 版後已🄵用: *collections.ChainMap* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Counter**(*collections.Counter, Dict[T, int]*)
collections.Counter 的泛型版本。

3.5.4 版新加入.

3.6.1 版新加入.

3.9 版後已🄵用: *collections.Counter* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Deque**(*deque, MutableSequence[T]*)
collections.deque 的泛型版本。

3.5.4 版新加入.

3.6.1 版新加入.

3.9 版後已🄵用: *collections.deque* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

其他具象类型

class typing.**IO**
class typing.**TextIO**
class typing.**BinaryIO**
泛型类型 `IO[AnyStr]` 及其子类 `TextIO(IO[str])` 与 `BinaryIO(IO[bytes])` 表示 I/O 流的类型，例如*open()* 所返回的对象。

Deprecated since version 3.8, will be removed in version 3.12: 这些类型也在 `typing.io` 命名空间中，它从未得到类型检查器的支持并将被移除。

class typing.**Pattern**
class typing.**Match**
这些类型对应的是从*re.compile()* 和*re.match()* 返回的类型。这些类型（及相应的函数）是 `AnyStr` 中的泛型并可通过编写 `Pattern[str]`, `Pattern[bytes]`, `Match[str]` 或 `Match[bytes]` 来具体指定。

Deprecated since version 3.8, will be removed in version 3.12: 这些类型也在 `typing.re` 命名空间中，它从未得到类型检查器的支持并将被移除。

3.9 版後已🄵用: *re* 模块中的 `Pattern` 与 `Match` 类现已支持 `[]`。详见 **PEP 585** 与*GenericAlias 类型*。

class typing.**Text**
`Text` 是 `str` 的别名。提供了对 Python 2 代码的向下兼容：Python 2 中，`Text` 是 `unicode` 的别名。

使用 `Text` 时，值中必须包含 unicode 字符串，以兼容 Python 2 和 Python 3：

```
def add_unicode_checkmark(text: Text) -> Text:
    return text + u' \u2713'
```

3.5.2 版新加入.

抽象基类

collections.abc 对应的容器

class typing.**AbstractSet**(*Sized, Collection[T_co]*)
collections.abc.Set 的泛型版本。

3.9 版後已🄵用: *collections.abc.Set* 现已支持 `[]`。详见 **PEP 585** 与*GenericAlias 类型*。

class typing.**ByteString**(*Sequence[int]*)
collections.abc.ByteString 的泛型版本。

该类型代表了*bytes*、*bytearray*、*memoryview* 等字节序列类型。

作为该类型的简称，*bytes* 可用于标注上述任意类型的参数。

3.9 版後已🄵用: *collections.abc.ByteString* 现已支持 `[]`。详见 **PEP 585** 与*GenericAlias 类型*。

class typing.**Collection**(*Sized, Iterable[T_co], Container[T_co]*)
collections.abc.Collection 的泛型版本。

3.6.0 版新加入.

3.9 版後已🄵用: *collections.abc.Collection* 现已支持 `[]`。详见 **PEP 585** 与*GenericAlias 类型*。

class typing.**Container**(*Generic[T_co]*)
collections.abc.Container 的泛型版本。

3.9 版後已🄵用: *collections.abc.Container* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**ItemsView**(*MappingView, Generic[KT_co, VT_co]*)
collections.abc.ItemsView 的泛型版本。

3.9 版後已🄵用: *collections.abc.ItemsView* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**KeysView**(*MappingView[KT_co], AbstractSet[KT_co]*)
collections.abc.KeysView 的泛型版本。

3.9 版後已🄵用: *collections.abc.KeysView* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Mapping**(*Sized, Collection[KT], Generic[VT_co]*)
collections.abc.Mapping 的泛型版本。用法如下:

```
def get_position_in_index(word_list: Mapping[str, int], word: str) -> int:
    return word_list[word]
```

3.9 版後已🄵用: *collections.abc.Mapping* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**MappingView**(*Sized, Iterable[T_co]*)
collections.abc.MappingView 的泛型版本。

3.9 版後已🄵用: *collections.abc.MappingView* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**MutableMapping**(*Mapping[KT, VT]*)
collections.abc.MutableMapping 的泛型版本。

3.9 版後已🄵用: *collections.abc.MutableMapping* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**MutableSequence**(*Sequence[T]*)
collections.abc.MutableSequence 的泛型版本。

3.9 版後已🄵用: *collections.abc.MutableSequence* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**MutableSet**(*AbstractSet[T]*)
collections.abc.MutableSet 的泛型版本。

3.9 版後已🄵用: *collections.abc.MutableSet* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Sequence**(*Reversible[T_co], Collection[T_co]*)
collections.abc.Sequence 的泛型版本。

3.9 版後已🄵用: *collections.abc.Sequence* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**ValuesView**(*MappingView[VT_co]*)
collections.abc.ValuesView 的泛型版本。

3.9 版後已🄵用: *collections.abc.ValuesView* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

collections.abc 对应的其他类型

class typing.**Iterable**(*Generic[T_co]*)
collections.abc.Iterable 的泛型版本。

3.9 版後已𝔽用: *collections.abc.Iterable* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Iterator**(*Iterable[T_co]*)
collections.abc.Iterator 的泛型版本。

3.9 版後已𝔽用: *collections.abc.Iterator* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Generator**(*Iterator[T_co], Generic[T_co, T_contra, V_co]*)
生成器可以由泛型类型 `Generator[YieldType, SendType, ReturnType]` 注解。例如:

```
def echo_round() -> Generator[int, float, str]:
    sent = yield 0
    while sent >= 0:
        sent = yield round(sent)
    return 'Done'
```

注意，与 typing 模块里的其他泛型不同，*Generator* 的 `SendType` 属于逆变行为，不是协变行为，也是不变行为。

如果生成器只产生值，可将 `SendType` 与 `ReturnType` 设为 `None`:

```
def infinite_stream(start: int) -> Generator[int, None, None]:
    while True:
        yield start
        start += 1
```

此外，还可以把生成器的返回类型注解为 `Iterable[YieldType]` 或 `Iterator[YieldType]`:

```
def infinite_stream(start: int) -> Iterator[int]:
    while True:
        yield start
        start += 1
```

3.9 版後已𝔽用: *collections.abc.Generator* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Hashable**
An alias to *collections.abc.Hashable*.

class typing.**Reversible**(*Iterable[T_co]*)
collections.abc.Reversible 的泛型版本。

3.9 版後已𝔽用: *collections.abc.Reversible* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Sized**
An alias to *collections.abc.Sized*.

异步编程

class typing.**Coroutine**(*Awaitable[V_co], Generic[T_co, T_contra, V_co]*)

collections.abc.Coroutine 的泛型版本。类型变量的差异和顺序与*Generator* 的内容相对应，例如：

```
from collections.abc import Coroutine
c: Coroutine[list[str], str, int]  # Some coroutine defined elsewhere
x = c.send('hi')                   # Inferred type of 'x' is list[str]
async def bar() -> None:
    y = await c                    # Inferred type of 'y' is int
```

3.5.3 版新加入.

3.9 版後已🄵用: *collections.abc.Coroutine* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**AsyncGenerator**(*AsyncIterator[T_co], Generic[T_co, T_contra]*)

异步生成器可由泛型类型 `AsyncGenerator[YieldType, SendType]` 注解。例如：

```
async def echo_round() -> AsyncGenerator[int, float]:
    sent = yield 0
    while sent >= 0.0:
        rounded = await round(sent)
        sent = yield rounded
```

与常规生成器不同，异步生成器不能返回值，因此没有 `ReturnType` 类型参数。与*Generator* 类似，`SendType` 也属于逆变行为。

如果生成器只产生值，可将 `SendType` 设置为 `None`：

```
async def infinite_stream(start: int) -> AsyncGenerator[int, None]:
    while True:
        yield start
        start = await increment(start)
```

此外，可用 `AsyncIterable[YieldType]` 或 `AsyncIterator[YieldType]` 注解生成器的返回类型：

```
async def infinite_stream(start: int) -> AsyncIterator[int]:
    while True:
        yield start
        start = await increment(start)
```

3.6.1 版新加入.

3.9 版後已🄵用: *collections.abc.AsyncGenerator* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**AsyncIterable**(*Generic[T_co]*)

collections.abc.AsyncIterable 的泛型版本。

3.5.2 版新加入.

3.9 版後已🄵用: *collections.abc.AsyncIterable* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**AsyncIterator**(*AsyncIterable[T_co]*)

collections.abc.AsyncIterator 的泛型版本。

3.5.2 版新加入.

3.9 版後已𝔽用: *collections.abc.AsyncIterator* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**Awaitable**(*Generic[T_co]*)
collections.abc.Awaitable 的泛型版本。

3.5.2 版新加入.

3.9 版後已𝔽用: *collections.abc.Awaitable* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

上下文管理器类型

class typing.**ContextManager**(*Generic[T_co]*)
contextlib.AbstractContextManager 的泛型版本。

3.5.4 版新加入.

3.6.0 版新加入.

3.9 版後已𝔽用: *contextlib.AbstractContextManager* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

class typing.**AsyncContextManager**(*Generic[T_co]*)
contextlib.AbstractAsyncContextManager 的泛型版本。

3.5.4 版新加入.

3.6.2 版新加入.

3.9 版後已𝔽用: *contextlib.AbstractAsyncContextManager* 现已支持 `[]`。详见 **PEP 585** 和*GenericAlias 类型*。

协议

这些协议由*runtime_checkable()* 装饰。

class typing.**SupportsAbs**
含抽象方法 `__abs__` 的抽象基类，是其返回类型里的协变量。

class typing.**SupportsBytes**
含抽象方法 `__bytes__` 的抽象基类。

class typing.**SupportsComplex**
含抽象方法 `__complex__` 的抽象基类。

class typing.**SupportsFloat**
含抽象方法 `__float__` 的抽象基类。

class typing.**SupportsIndex**
含抽象方法 `__index__` 的抽象基类。

3.8 版新加入.

class typing.**SupportsInt**
含抽象方法 `__int__` 的抽象基类。

class typing.**SupportsRound**
含抽象方法 `__round__` 的抽象基类，是其返回类型的协变量。

函数与装饰器

typing.**cast**(*typ*, *val*)
把值强制转换为类型。

不变更返回值。对类型检查器而言，代表了返回值具有指定的类型，但运行时故意不做任何检查（以便让检查速度尽量快）。

@typing.**overload**
@overload 装饰器可以修饰支持多个不同参数类型组合的函数或方法。@overload - 装饰定义的系列必须紧跟一个非 @overload-装饰定义（用于同一个函数/方法）。@overload-装饰定义仅是为了协助类型检查器，因为该装饰器会被非 @overload-装饰定义覆盖，后者用于运行时，而且会被类型检查器忽略。在运行时直接调用 @overload 装饰的函数会触发*NotImplementedError*。下面的重载示例给出了比联合类型或类型变量更精准的类型：

```
@overload
def process(response: None) -> None:
    ...
@overload
def process(response: int) -> tuple[int, str]:
    ...
@overload
def process(response: bytes) -> str:
    ...
def process(response):
    <actual implementation>
```

详见 **PEP 484**，与其他类型语义进行对比。

@typing.**final**
告知类型检查器被装饰的方法不能被覆盖，且被装饰的类不能作为子类的装饰器，例如：

```
class Base:
    @final
    def done(self) -> None:
        ...
class Sub(Base):
    def done(self) -> None:  # Error reported by type checker
        ...

@final
class Leaf:
    ...
class Other(Leaf):  # Error reported by type checker
    ...
```

这些属性没有运行时检查。详见 **PEP 591**。

3.8 版新加入.

@typing.**no_type_check**
标明注解不是类型提示的装饰器。

用作类或函数的*decorator*。用于类时，递归地应用于该类中定义的所有方法，（但不影响超类或子类中定义的方法）。

本方法可直接修改函数。

@typing.**no_type_check_decorator**
让其他装饰器具有*no_type_check()* 效果的装饰器。

本装饰器用*no_type_check()* 里的装饰函数打包其他装饰器。

@typing.**type_check_only**
标记类或函数内不可用于运行时的装饰器。

在运行时，该装饰器本身不可用。实现返回的是私有类实例时，它主要是用于标记在类型存根文件中定义的类。

```
@type_check_only
class Response:  # private or not available at runtime
    code: int
    def get_header(self, name: str) -> str: ...

def fetch_response() -> Response: ...
```

注意，建议不要返回私有类实例，最好将之设为公共类。

内省辅助器

typing.**get_type_hints**(*obj*, *globalns=None*, *localns=None*, *include_extras=False*)
返回函数、方法、模块、类对象的类型提示的字典。

一般情况下，与 `obj.__annotations__` 相同。此外，可通过在 `globals` 与 `locals` 命名空间里进行评估，以此来处理编码为字符串字面量的前向引用。如有需要，在默认值设置为 `None` 时，可为函数或方法注解添加 `Optional[t]`。对于类 `C`，则返回由所有 `__annotations__` 与 `C.__mro__` 逆序合并而成的字典。

本函数以递归地方式用 `T` 替换所有 `Annotated[T, ...]`，除非将 `include_extras` 的值设置为 `True` （详见*Annotated*）。例如：

```
class Student(NamedTuple):
    name: Annotated[str, 'some marker']

get_type_hints(Student) == {'name': str}
get_type_hints(Student, include_extras=False) == {'name': str}
get_type_hints(Student, include_extras=True) == {
    'name': Annotated[str, 'some marker']
}
```

3.9 版更變: **PEP 593** 的组成部分，添加了 `include_extras` 参数。

typing.**get_args**(*tp*)

typing.**get_origin**(*tp*)
为泛型类型与特殊类型形式提供了基本的内省功能。

对于 `X[Y, Z, ...]` 形式的类型对象，这些函数返回 `X` 与 `(Y, Z, ...)`。如果 `X` 是内置对象或*collections* class 的泛型别名，会将其标准化为原始类。如果 `X` 是包含在其他泛型类型中的*Union* 或*Literal*，`(Y, Z, ...)` 的顺序会因类型缓存，而与原始参数 `[Y, Z, ...]` 的顺序不同。对于不支持的对象会相应地返回 `None` 或 `()`。例如：

```
assert get_origin(Dict[str, int]) is dict
assert get_args(Dict[int, str]) == (int, str)

assert get_origin(Union[int, str]) is Union
assert get_args(Union[int, str]) == (int, str)
```

3.8 版新加入.

class typing.**ForwardRef**
用于字符串前向引用的内部类型表示的类。例如，`List["SomeClass"]` 会被隐式转换为 `List[ForwardRef("SomeClass")]`。这个类不应由用户来实例化，但可以由内省工具使用。

備 🄵: PEP 585 泛型类型例如 `list["SomeClass"]` 将不会被隐式地转换为 `list[ForwardRef("SomeClass")]` 因而将不会自动解析为 `list[SomeClass]`。

3.7.4 版新加入.

常量

typing.**TYPE_CHECKING**
被第三方静态类型检查器假定为 `True` 的特殊常量。在运行时为 `False`。用法如下：

```
if TYPE_CHECKING:
    import expensive_mod

def fun(arg: 'expensive_mod.SomeType') -> None:
    local_var: expensive_mod.AnotherType = other_fun()
```

第一个类型注解必须用引号标注，才能把它当作“前向引用”，从而在解释器运行时中隐藏 `expensive_mod` 引用。局部变量的类型注释不会被评估，因此，第二个注解不需要用引号引起来。

備🄵: If `from __future__ import annotations` is used, annotations are not evaluated at function definition time. Instead, they are stored as strings in `__annotations__`. This makes it unnecessary to use quotes around the annotation (see PEP 563).

3.5.2 版新加入.

26.2 pydoc --- 文档生成器和在线帮助系统

源代码: Lib/pydoc.py

---

pydoc 模块自动利用 Python 模块生成文档。生成的文档可在控制台中显示为文本页面，还可在 Web 浏览器中查阅，或保存为 HTML 文件。

对于模块、类、函数和方法，显示的文档内容取自文档字符串（即 `__doc__` 属性），并会递归地从其带文档的成员中获取。如果没有文档字符串，*pydoc* 会尝试从类、函数或方法定义上方，或是模块顶部的注释行段落获取 (参见*inspect.getcomments()*).

内置函数*help()* 会发起调用交互式解释器的在线帮助系统，该系统使用*pydoc* 在终端上生成文本形式的文档内容。同样的文本文档也可以在 Python 解释器以外通过在操作系统的命令提示符下以脚本方式运行 **pydoc** 来查看。例如，运行

```
pydoc sys
```

在终端提示符下将通过*sys* 模块显示文档内容，其样式类似于 Unix **man** 命令所显示的指南页面。**pydoc** 的参数可以为函数、模块、包，或带点号的对模块中的类、方法或函数以及包中的模块的引用。如果传给

pydoc 的参数像是一个路径（即包含所在操作系统的路径分隔符，例如 Unix 的正斜杠），并且其指向一个现有的 Python 源文件，则会为该文件生成文档内容。

備𝔼: 为了找到对象及其文档内容，*pydoc* 会导入文档所在的模块。因此，任何模块层级的代码都将被执行。请使用 `if __name__ == '__main__':` 语句来确保一个文件的特定代码仅在作为脚本被发起调用时执行而不是在被导入时执行。

当打印输出到控制台时，**pydoc** 会尝试对输出进行分页以方便阅读。如果设置了 `PAGER` 环境变量，**pydoc** 将使用该变量值作为分页程序。

在参数前指定 `-w` 旗标将把 HTML 文档写入到当前目录下的一个文件中，而不是在控制台中显示文本。

在参数前指定 `-k` 旗标将在全部可用模块的提要行中搜索参数所给定的关键字，具体方式同样类似于 Unix **man** 命令。模块的提要行就是其文档字符串的第一行。

你还可以使用 **pydoc** 在本机上启动一个 HTTP 服务，这将向来访的 Web 浏览器提供文档服务。**pydoc -p 1234** 将在 1234 端口上启动 HTTP 服务，允许你在你喜欢的 Web 服务器中通过 `http://localhost:1234/` 浏览文档内容。指定 `0` 作为端口号将会任意选择一个未使用的端口。

pydoc -n <hostname> 将启动在给定主机名上执行监听的服务。默认主机名为'localhost' 但如果你希望能从其他机器搜索该服务器，你可能会想要改变服务器所响应的主机名。在开发阶段此特性会特别有用，因为这样你将能在一个容器中运行 pydoc。

pydoc -b 将启动服务并额外打开一个 Web 浏览器访问模块索引页。所发布的每个页面顶端都带有导航栏，你可以点击 *Get* 获取特定条目的帮助，点击 *Search* 在所有模块的提要行中搜索特定关键词，或是点击 *Module index*, *Topics* 和 *Keywords* 前往相应的页面。

当 **pydoc** 生成文档内容时，它会使用当前环境和路径来定位模块。因此，发起调用 **pydoc spam** 得到的文档版本会与你启动 Python 解释器并输入 `import spam` 时得到的模块版本完全相同。

核心模块的模块文档位置对应于 `https://docs.python.org/X.Y/library/` 其中 `X` 和 `Y` 是 Python 解释器的主要版本号和小版本号。这可通过设置 `PYTHONDOCS` 环境变量来重载为指向不同的 URL 或包含 Library Reference Manual 页面的本地目录。

3.2 版更變: 添加 `-b` 选项。

3.3 版更變: 命令行选项 `-g` 已经移除。

3.4 版更變: *pydoc* 现在会使用*inspect.signature()* 而非*inspect.getfullargspec()* 来从可调用对象中提取签名信息。

3.7 版更變: 添加 `-n` 选项。

26.3 Python 开发模式

3.7 版新加入.

Python 开发模式引入了额外的运行时检查，由于成本太高，所以默认情况下是不启用的。如果代码是正确的，它不应该比默认情况的更加详细；只有在检测到问题的时候才会发出新的警告。

它可以通过 `-X dev` 命令行选项或通过设置 `PYTHONDEVMODE` 环境变量为 `1` 来启用。

26.4 Python 开发模式的效果

启用 Python 开发模式后的效果，与以下命令类似，不过还有下面的额外效果：

```
PYTHONMALLOC=debug PYTHONASYNCIODEBUG=1 python3 -W default -X faulthandler
```

Python 开发模式的效果：

- 加入 `default` *warning filter* 。下述警告信息将会显示出来：
  - *DeprecationWarning*
  - *ImportWarning*
  - *PendingDeprecationWarning*
  - *ResourceWarning*

  通常上述警告是由默认的*warning filters* 负责处理的。

  其行为相当于使用了 `-W default` 命令行选项。

  使用命令行参数 `-W error` 或将环境变量 `PYTHONWARNINGS` 设为 `error`，可将警告视为错误。
- 在内存分配程序中安装调试钩子，用以查看：
  - 缓冲区下溢
  - 缓冲区上溢
  - 内存分配 API 冲突
  - 不安全的 GIL 调用

  参见 C 函数 `PyMem_SetupDebugHooks()` 。

  效果类似于将环境变量 `PYTHONMALLOC` 设为 `debug`。

  若要启用 Python 开发模式，却又不要在内存分配程序中安装调试钩子，请将环境变量 `PYTHONMALLOC` 设为 `default`。
- 在启动 Python 时调用*faulthandler.enable()* ，会安装 `SIGSEGV` 、`SIGFPE` 、`SIGABRT` 、`SIGBUS` 和 `SIGILL` 信号的处理程序，以便在程序崩溃时将 Python 跟踪信息转储下来。

  其行为如同使用了 `-X faulthandler` 命令行选项或将 `PYTHONFAULTHANDLER` 环境变量设为 `1`。
- 启用*asyncio debug mode*。比如*asyncio* 会检查没有等待的协程并记录下来。

  效果如同将环境变量 `PYTHONASYNCIODEBUG` 设为 `1`。
- 检查字符串编码和解码函数的 *encoding* 和 *errors* 参数。例如：*open()* 、*str.encode()* 和*bytes.decode()*。

  为了获得最佳性能，默认只会在第一次编码/解码错误时才会检查 *errors* 参数，有时 *encoding* 参数为空字符串时还会被忽略。
- *io.IOBase* 的析构函数会记录 `close()` 触发的异常。
- 将*sys.flags* 的 `dev_mode` 属性设为 `True`。

Python 开发模式下，默认不会启用*tracemalloc* 模块，因为其性能和内存开销太大。启用*tracemalloc* 模块后，能够提供有关错误来源的一些额外信息。例如，*ResourceWarning* 记录了资源分配的跟踪信息，而缓冲区溢出错误记录了内存块分配的跟踪信息。

Python 开发模式不会阻止命令行参数 `-O` 删除 `assert` 语句，也不会阻止将*__debug__* 设为 `False`。

3.8 版更變: 现在，*io.IOBase* 的析构函数会记录 `close()` 触发的异常。

3.9 版更變: 现在，字符串编码和解码操作时会检查 *encoding* 和 *errors* 参数。

26.5 ResourceWarning 示例

以下示例将统计由命令行指定的文本文件的行数:

```
import sys

def main():
    fp = open(sys.argv[1])
    nlines = len(fp.readlines())
    print(nlines)
    # The file is closed implicitly

if __name__ == "__main__":
    main()
```

上述代码没有显式关闭文件。默认情况下，Python 不会触发任何警告。下面用 README.txt 文件测试下，有 269 行。

```
$ python3 script.py README.txt
269
```

启用 Python 开发模式后，则会显示一条*ResourceWarning* 警告。

```
$ python3 -X dev script.py README.txt
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst'␣
↪mode='r' encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
```

启用*tracemalloc* 后，则还会显示打开文件的那行代码:

```
$ python3 -X dev -X tracemalloc=5 script.py README.rst
269
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='README.rst'␣
↪mode='r' encoding='UTF-8'>
  main()
Object allocated at (most recent call last):
  File "script.py", lineno 10
    main()
  File "script.py", lineno 4
    fp = open(sys.argv[1])
```

修正方案就是显式关闭文件。下面用上下文管理器作为示例:

```
def main():
    # Close the file explicitly when exiting the with block
    with open(sys.argv[1]) as fp:
        nlines = len(fp.readlines())
    print(nlines)
```

未能显式关闭资源，会让资源打开时长远超预期；在退出 Python 时可能会导致严重问题。这在 CPython 中比较糟糕，但在 PyPy 中会更糟。显式关闭资源能让应用程序更加稳定可靠。

26.6 文件描述符错误示例

显示自身的第一行代码:

```
import os

def main():
    fp = open(__file__)
    firstline = fp.readline()
    print(firstline.rstrip())
    os.close(fp.fileno())
    # The file is closed implicitly

main()
```

默认情况下，Python 不会触发任何警告:

```
$ python3 script.py
import os
```

在 Python 开发模式下，会在析构文件对象时显示*ResourceWarning* 并记录 “Bad file descriptor” 错误。

```
$ python3 script.py
import os
script.py:10: ResourceWarning: unclosed file <_io.TextIOWrapper name='script.py' mode=
↪'r' encoding='UTF-8'>
  main()
ResourceWarning: Enable tracemalloc to get the object allocation traceback
Exception ignored in: <_io.TextIOWrapper name='script.py' mode='r' encoding='UTF-8'>
Traceback (most recent call last):
  File "script.py", line 10, in <module>
    main()
OSError: [Errno 9] Bad file descriptor
```

`os.close(fp.fileno())` 会关闭文件描述符。当文件对象析构函数试图再次关闭文件描述符时会失败，并触发 `Bad file descriptor` 错误。每个文件描述符只允许关闭一次。在最坏的情况下，关闭两次会导致程序崩溃（示例可参见 bpo-18748 ）。

修正方案是删除 `os.close(fp.fileno())` 这一行，或者打开文件时带上 `closefd=False` 参数。

26.7 `doctest` --- 测试交互性的 Python 示例

** 源代码 ** Lib/doctest.py

---

doctest 模块寻找像 Python 交互式代码的文本，然后执行这些代码来确保它们的确就像展示的那样正确运行，有许多方法来使用 doctest:

- 通过验证所有交互式示例仍然按照记录的方式工作，以此来检查模块的文档字符串是否是最新的。
- 通过验证来自一个测试文件或一个测试对象的交互式示例按预期工作，来进行回归测试。
- 为一个包写指导性的文档，用输入输出的例子来说明。取决于是强调例子还是说明性的文字，这有一种” 文本测试” 或” 可执行文档” 的风格。

下面是一个小却完整的示例模块:

```
"""
This is the "example" module.

The example module supplies one function, factorial().  For example,

>>> factorial(5)
120
"""

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """

    import math
    if not n >= 0:
        raise ValueError("n must be >= 0")
    if math.floor(n) != n:
        raise ValueError("n must be exact integer")
    if n+1 == n:  # catch a value like 1e300
        raise OverflowError("n too large")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

如果你直接在命令行里运行 `example.py`，*doctest* 将发挥他的作用。

```
$ python example.py
$
```

没有输出！这很正常，这意味着所有的例子都成功了。把 -v 传给脚本，*doctest* 会打印出它所尝试的详细日志，并在最后打印出一个总结。

```
$ python example.py -v
Trying:
    factorial(5)
Expecting:
    120
ok
Trying:
    [factorial(n) for n in range(6)]
Expecting:
    [1, 1, 2, 6, 24, 120]
ok
```

以此类推，最终以：

```
Trying:
    factorial(1e100)
Expecting:
    Traceback (most recent call last):
        ...
    OverflowError: n too large
ok
2 items passed all tests:
   1 tests in __main__
   8 tests in __main__.factorial
9 tests in 2 items.
9 passed and 0 failed.
Test passed.
$
```

这就是对于高效地使用*doctest* 你所需要知道的一切！开始上手吧。下面的章节提供了完整的细节。请注意，在标准的 Python 测试套件和库中有许多 doctest 的例子。特别有用的例子可以在标准测试文件 `Lib/test/test_doctest.py` 中找到。

26.7.1 简单用法：检查 Docstrings 中的示例

开始使用 doctest 的最简单方法（但不一定是你将继续这样做的方式）是结束每个模块 `M` 使用：

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

doctest 会随后检查模块 `M` 中的文档字符串。

以脚本形式运行该模块会使文档中的例子得到执行和验证：

```
python M.py
```

这不会显示任何东西，除非一个例子失败了，在这种情况下，失败的例子和失败的原因会被打印到 stdout，最后一行的输出是 “***Test Failed*** N failures.“，其中 *N* 是失败的例子的数量。

用 “-v“来运行它来切换，而不是：

```
python M.py -v
```

并将所有尝试过的例子的详细报告打印到标准输出，最后还有各种总结。

你可以通过向*testmod()* 传递 verbose=True 来强制执行 verbose 模式，或者通过传递 verbose=False 来禁止它。在这两种情况下，sys.argv 都不会被*testmod()* 检查（所以传递 -v 或不传递都没有影响）。

还有一个命令行快捷方式用于运行*testmod()*。你可以指示 Python 解释器直接从标准库中运行 doctest 模块，并在命令行中传递模块名称:

```
python -m doctest -v example.py
```

这将导入 example.py 作为一个独立的模块，并对其运行*testmod()*。注意，如果该文件是一个包的一部分，并且从该包中导入了其他子模块，这可能无法正确工作。

关于*testmod()* 的更多信息，请参见基本 *API* 部分。

26.7.2 简单的用法：检查文本文件中的例子

doctest 的另一个简单应用是测试文本文件中的交互式例子。这可以用*testfile()* 函数来完成::

```
import doctest
doctest.testfile("example.txt")
```

这个简短的脚本执行并验证文件 example.txt 中包含的任何交互式 Python 示例。该文件的内容被当作一个巨大的文档串来处理；该文件不需要包含一个 Python 程序！例如，也许 example.txt 包含以下内容:

```
The ``example`` module
======================

Using ``factorial``
-------------------

This is an example text file in reStructuredText format.  First import
``factorial`` from the ``example`` module:

    >>> from example import factorial

Now use it:

    >>> factorial(6)
    120
```

运行 “doctest.testfile(”example.txt”)“，然后发现这个文档中的错误:

```
File "./example.txt", line 14, in example.txt
Failed example:
    factorial(6)
Expected:
    120
Got:
    720
```

与*testmod()* 一样，*testfile()* 不会显示任何东西，除非一个例子失败。如果一个例子失败了，那么失败的例子和失败的原因将被打印到 stdout，使用的格式与*testmod()* 相同。

默认情况下，*testfile()* 在调用模块的目录中寻找文件。参见章节基本 *API*，了解可用于告诉它在其他位置寻找文件的可选参数的描述。

像*testmod()* 一样，*testfile()* 的详细程度可以通过命令行 -v 切换或可选的关键字参数 *verbose* 来设置。

还有一个命令行快捷方式用于运行*testfile()*。你可以指示 Python 解释器直接从标准库中运行 doctest 模块，并在命令行中传递文件名:

```
python -m doctest -v example.txt
```

因为文件名没有以 .py 结尾，*doctest* 推断它必须用*testfile()* 运行，而不是*testmod()*。

关于*testfile()* 的更多信息，请参见基本 *API* 一节。

26.7.3 它是如何工作的

这一节详细研究了 doctest 的工作原理：它查看哪些文档串，它如何找到交互式的用例，它使用什么执行环境，它如何处理异常，以及如何用选项标志来控制其行为。这是你写 doctest 例子所需要知道的信息；关于在这些例子上实际运行 doctest 的信息，请看下面的章节。

哪些文件串被检查了？

模块的文档串以及所有函数、类和方法的文档串都将被搜索。导入模块的对象不被搜索。

此外，如果 M.__test__ 存在并且"为真值"，则它必须是一个字典，其中每个条目都将一个（字符串）名称映射到一个函数对象、类对象或字符串。从 M.__test__ 找到的函数和类对象的文档字符串会被搜索，而字符串会被当作文档字符串来处理。在输出时，每个键 K 在 M.__test__ 中都显示为其名称

```
<name of M>.__test__.K
```

任何发现的类都会以类似的方式进行递归搜索，以测试其包含的方法和嵌套类中的文档串。

CPython implementation detail: Prior to version 3.4, extension modules written in C were not fully searched by doctest.

文档串的例子是如何被识别的？

在大多数情况下，对交互式控制台会话的复制和粘贴功能工作得很好，但是 doctest 并不试图对任何特定的 Python shell 进行精确的模拟。

```
>>> # comments are ignored
>>> x = 12
>>> x
12
>>> if x == 13:
...     print("yes")
... else:
...     print("no")
...     print("NO")
...     print("NO!!!")
...
no
NO
NO!!!
>>>
```

任何预期的输出必须紧随包含代码的最后 '>>> ' 或 '... ' 行，预期的输出（如果有的话）延伸到下一 '>>> ' 行或全空白行。

fine 输出:

- 预期输出不能包含一个全白的行，因为这样的行被认为是预期输出的结束信号。如果预期的输出包含一个空行，在你的测试例子中，在每一个预期有空行的地方加上 “<BLANKLINE>“。
- 所有硬制表符都被扩展为空格，使用 8 列的制表符。由测试代码生成的输出中的制表符不会被修改。因为样本输出中的任何硬制表符都会被扩展，这意味着如果代码输出包括硬制表符，文档测试通过的唯一方法是*NORMALIZE_WHITESPACE* 选项或者指令 是有效的。另外，测试可以被重写，以捕获输出并将其与预期值进行比较，作为测试的一部分。这种对源码中标签的处理是通过试错得出的，并被证明是最不容易出错的处理方式。通过编写一个自定义的*DocTestParser* 类，可以使用一个不同的算法来处理标签。
- 向 stdout 的输出被捕获，但不向 stderr 输出（异常回溯通过不同的方式被捕获）。
- 如果你在交互式会话中通过反斜线续行，或出于任何其他原因使用反斜线，你应该使用原始文件串，它将完全保留你输入的反斜线:

```
>>> def f(x):
...     r'''Backslashes in a raw docstring: m\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

否则，反斜杠将被解释为字符串的一部分。例如，上面的 “n“会被解释为一个换行符。另外，你可以在 doctest 版本中把每个反斜杠加倍（而不使用原始字符串）：

```
>>> def f(x):
...     '''Backslashes in a raw docstring: m\\n'''
>>> print(f.__doc__)
Backslashes in a raw docstring: m\n
```

- 起始列并不重要:

```
>>> assert "Easy!"
      >>> import math
          >>> math.floor(1.9)
          1
```

并从预期的输出中剥离出与开始该例子的初始 '>>> ' 行中出现的同样多的前导空白字符。

什么是执行上下文

默认情况下，每次*doctest* 找到要测试的文档串时，它都会使用 M 的 * 浅层副本 *，这样运行测试就不会改变模块的真正全局变量，而且 M 的一个测试也不会留下临时变量，从而意外地让另一个测试通过。这意味着例子可以自由地使用 M 中的任何顶级定义的名字，以及正在运行的文档串中早期定义的名字。用例不能看到其他文档串中定义的名字。

你可以通过将 “globs=your_dict“传递给*testmod()* 或*testfile()* 来强制使用你自己的 dict 作为执行环境。

异常如何处理？

没问题，只要回溯是这个例子产生的唯一输出：只要粘贴回溯即可。[1] 由于回溯所包含的细节可能会迅速变化（例如，确切的文件路径和行号），这是 doctest 努力使其接受的内容具有灵活性的一种情况。

简单实例:

[1] 不支持同时包含预期输出和异常的用例。试图猜测一个在哪里结束，另一个在哪里开始，太容易出错了，而且这也会使测试变得混乱。

```
>>> [1, 2, 3].remove(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

如果*ValueError*被触发，该测试就会成功，`list.remove(x): x not in list`的细节如图所示。

异常的预期输出必须以回溯头开始，可以是以下两行中的任何一行，缩进程度与例子中的第一行相同:

```
Traceback (most recent call last):
Traceback (innermost last):
```

回溯头的后面是一个可选的回溯堆栈，其内容被 doctest 忽略。回溯堆栈通常是省略的，或者从交互式会话中逐字复制的。

回溯堆栈的后面是最有趣的部分：包含异常类型和细节的一行（几行）。这通常是回溯的最后一行，但如果异常有多行细节，则可以延伸到多行:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: multi
    line
detail
```

最后三行（以*ValueError*开头）将与异常的类型和细节进行比较，其余的被忽略。

最佳实践是省略回溯栈，除非它为这个例子增加了重要的文档价值。因此，最后一个例子可能更好，因为:

```
>>> raise ValueError('multi\n    line\ndetail')
Traceback (most recent call last):
    ...
ValueError: multi
    line
detail
```

请注意，回溯的处理方式非常特别。特别是，在重写的例子中，`...`的使用与 doctest 的*ELLIPSIS*选项无关。该例子中的省略号可以不写，也可以是三个（或三百个）逗号或数字，或者是一个缩进的 Monty Python 短剧的剧本。

有些细节你应该读一遍，但不需要记住：

- Doctest 不能猜测你的预期输出是来自异常回溯还是来自普通打印。因此，例如，一个期望`ValueError: 42 is prime`的用例将通过测试，无论*ValueError*是真的被触发，或者该用例只是打印了该回溯文本。在实践中，普通输出很少以回溯标题行开始，所以这不会产生真正的问题。
- 回溯堆栈的每一行（如果有的话）必须比例子的第一行缩进，或者以一个非字母数字的字符开始。回溯头之后的第一行缩进程度相同，并且以字母数字开始，被认为是异常细节的开始。当然，这对真正的回溯来说是正确的事情。
- 当*IGNORE_EXCEPTION_DETAIL* doctest 选项被指定时，最左边的冒号后面的所有内容以及异常名称中的任何模块信息都被忽略。
- 交互式 shell 省略了一些*SyntaxError*的回溯头行。但 doctest 使用回溯头行来区分异常和非异常。所以在罕见的情况下，如果你需要测试一个省略了回溯头的*SyntaxError*，你将需要手动添加回溯头行到你的测试用例中。
- 对于一些*SyntaxError*，Python 使用 ^ 标记显示语法错误的字符位置:

```
>>> 1 1
  File "<stdin>", line 1
    1 1
      ^
SyntaxError: invalid syntax
```

由于显示错误位置的行在异常类型和细节之前，它们不被 doctest 检查。例如，下面的测试会通过，尽管它把 "^"标记放在了错误的位置:

```
>>> 1 1
  File "<stdin>", line 1
    1 1
    ^
SyntaxError: invalid syntax
```

选项标记

一系列选项旗标控制着 doctest 的各方面行为。旗标的符号名称以模块常量的形式提供，可以一起 按位或并传递给各种函数。这些名称也可以在*doctest 指令* 中使用，并且可以通过 `-o` 选项传递给 doctest 命令行接口。

3.4 版新加入: 命令行选项 `-o` 。

第一组选项定义了测试语义，控制 doctest 如何决定实际输出是否与用例的预期输出相匹配方面的问题。

`doctest.`**`DONT_ACCEPT_TRUE_FOR_1`**
默认情况下，如果一个预期的输出块只包含 `1`，那么实际的输出块只包含 `1` 或只包含 `True` 就被认为是匹配的，同样，`0` 与 `False` 也是如此。当*DONT_ACCEPT_TRUE_FOR_1* 被指定时，两种替换都不允许。默认行为是为了适应 Python 将许多函数的返回类型从整数改为布尔值；期望" 小整数" 输出的测试在这些情况下仍然有效。这个选项可能会消失，但不会在几年内消失。

`doctest.`**`DONT_ACCEPT_BLANKLINE`**
默认情况下，如果一个预期输出块包含一个只包含字符串 `<BLANKLINE>` 的行，那么该行将与实际输出中的一个空行相匹配。因为一个真正的空行是对预期输出的限定，这是传达预期空行的唯一方法。当*DONT_ACCEPT_BLANKLINE* 被指定时，这种替换是不允许的。

`doctest.`**`NORMALIZE_WHITESPACE`**
当指定时，所有的空白序列（空白和换行）都被视为相等。预期输出中的任何空白序列将与实际输出中的任何空白序列匹配。默认情况下，空白必须完全匹配。*NORMALIZE_WHITESPACE* 在预期输出非常长的一行，而你想把它包在源代码的多行中时特别有用。

`doctest.`**`ELLIPSIS`**
当指定时，预期输出中的省略号（`...`）可以匹配实际输出中的任何子串。这包括跨行的子串和空子串，所以最好保持简单的用法。复杂的用法会导致与 ".*"在正则表达式中容易出现的"oops, it matched too much!" 相同的意外情况。

`doctest.`**`IGNORE_EXCEPTION_DETAIL`**
When specified, doctests expecting exceptions pass so long as an exception of the expected type is raised, even if the details (message and fully-qualified exception name) don't match.

For example, an example expecting `ValueError: 42` will pass if the actual exception raised is `ValueError: 3*14`, but will fail if, say, a *TypeError* is raised instead. It will also ignore any fully-qualified name included before the exception class, which can vary between implementations and versions of Python and the code/libraries in use. Hence, all three of these variations will work with the flag specified:

```
>>> raise Exception('message')
Traceback (most recent call last):
Exception: message
```

(下页继续)

(繼續上一頁)

```
>>> raise Exception('message')
Traceback (most recent call last):
builtins.Exception: message

>>> raise Exception('message')
Traceback (most recent call last):
__main__.Exception: message
```

Note that *ELLIPSIS* can also be used to ignore the details of the exception message, but such a test may still fail based on whether the module name is present or matches exactly.

3.2 版更變: *IGNORE_EXCEPTION_DETAIL* 现在也忽略了与包含被测异常的模块有关的任何信息。

doctest.**SKIP**
当指定时，完全不运行这个用例。这在 doctest 用例既是文档又是测试案例的情况下很有用，一个例子应该包括在文档中，但不应该被检查。例如，这个例子的输出可能是随机的；或者这个例子可能依赖于测试驱动程序所不能使用的资源。

SKIP 标志也可用于临时" 注释" 用例。

doctest.**COMPARISON_FLAGS**
一个比特或运算将上述所有的比较标志放在一起。

第二组选项控制测试失败的报告方式：

doctest.**REPORT_UDIFF**
当指定时，涉及多行预期和实际输出的故障将使用统一的差异来显示。

doctest.**REPORT_CDIFF**
当指定时，涉及多行预期和实际输出的故障将使用上下文差异来显示。

doctest.**REPORT_NDIFF**
当指定时，差异由 "difflib.Differ"来计算，使用与流行的:file:*ndiff.py*'工具相同的算法。这是唯一一种标记行内和行间差异的方法。例如，如果一行预期输出包含数字 *"1"*，而实际输出包含字母 "l"，那么就会插入一行，用圆点标记不匹配的列位置。

doctest.**REPORT_ONLY_FIRST_FAILURE**
当指定时，在每个 doctest 中显示第一个失败的用例，但隐藏所有其余用例的输出。这将防止 doctest 报告由于先前的失败而中断的正确用例；但也可能隐藏独立于第一个失败的不正确用例。当*REPORT_ONLY_FIRST_FAILURE* 被指定时，其余的用例仍然被运行，并且仍然计入报告的失败总数；只是输出被隐藏了。

doctest.**FAIL_FAST**
当指定时，在第一个失败的用例后退出，不尝试运行其余的用例。因此，报告的失败次数最多为 1。这个标志在调试时可能很有用，因为第一个失败后的用例甚至不会产生调试输出。

doctest 命令行接受选项 "-f"作为 "-o FAIL_FAST"的简洁形式。

3.4 版新加入.

doctest.**REPORTING_FLAGS**
一个比特或操作将上述所有的报告标志组合在一起。

还有一种方法可以注册新的选项标志名称，不过这并不有用，除非你打算通过子类来扩展*doctest* 内部。

doctest.**register_optionflag**(*name*)
用给定的名称创建一个新的选项标志，并返回新标志的整数值。*register_optionflag()* 可以在继承*OutputChecker* 或*DocTestRunner* 时使用，以创建子类支持的新选项。*register_optionflag()* 应始终使用以下方式调用:

```
MY_FLAG = register_optionflag('MY_FLAG')
```

指令

Doctest 指令可以用来修改单个例子的*option flags* 。Doctest 指令是在一个用例的源代码后面的特殊 Python 注释。

```
directive             ::=   "#" "doctest:" directive_options
directive_options     ::=   directive_option ("," directive_option)\*
directive_option      ::=   on_or_off directive_option_name
on_or_off             ::=   "+" \| "-"
directive_option_name ::=   "DONT_ACCEPT_BLANKLINE" \| "NORMALIZE_WHITESPACE" \| ...
```

+ 或 - 与指令选项名称之间不允许有空格。指令选项名称可以是上面解释的任何一个选项标志名称。

一个用例的 doctest 指令可以修改 doctest 对该用例的行为。使用 + 来启用指定的行为，或者使用 - 来禁用它。

例如，这个测试将会通过:

```
>>> print(list(range(20)))
[0,   1,  2,  3,  4,  5,  6,  7,  8,  9,
10,  11, 12, 13, 14, 15, 16, 17, 18, 19]
```

如果没有这个指令，它就会失败，这是因为实际的输出在个位数的列表元素前没有两个空格，也因为实际的输出是在单行上。这个测试也通过了，而且也需要一个指令来完成:

```
>>> print(list(range(20)))
[0, 1, ..., 18, 19]
```

在一个物理行上可以使用多个指令，用逗号分隔:

```
>>> print(list(range(20)))
[0,    1, ...,   18,    19]
```

如果在一个用例中使用了多个指令注释，那么它们将被合并:

```
>>> print(list(range(20)))
...
[0,    1, ...,   18,    19]
```

正如前面的例子所示，你可以在你的用例中添加只包含指令的行。当一个用例太长，指令不能舒适地放在同一行时，这可能很有用:

```
>>> print(list(range(5)) + list(range(10, 20)) + list(range(30, 40)))
...
[0, ..., 4, 10, ..., 19, 30, ..., 39]
```

请注意，由于所有的选项都是默认禁用的，而指令只适用于它们出现的用例，所以启用选项 (通过指令中的 +) 通常是唯一有意义的选择。然而，选项标志也可以被传递给运行测试的函数，建立不同的默认值。在这种情况下，通过指令中的 - 来禁用一个选项可能是有用的。

警告

doctest 是严格地要求在预期输出中完全匹配。如果哪怕只有一个字符不匹配，测试就会失败。这可能会让你吃惊几次，在你确切地了解到 Python 对输出的保证和不保证之前。例如，当打印一个集合时，Python 不保证元素以任何特定的顺序被打印出来，所以像:

```
>>> foo()
{"Hermione", "Harry"}
```

是不可靠的！一个变通方法是做:

```
>>> foo() == {"Hermione", "Harry"}
True
```

来取代。另一个是使用

```
>>> d = sorted(foo())
>>> d
['Harry', 'Hermione']
```

備🄴: 在 Python 3.6 之前，当打印一个 dict 时，Python 并不保证键值对以任何特定的顺序被打印。

还有其他的问题，但你会明白的。

另一个不好的做法是打印嵌入了对象地址的变量，如:

```
>>> id(1.0) # certain to fail some of the time
7948648
>>> class C: pass
>>> C()   # the default repr() for instances embeds an address
<__main__.C instance at 0x00AC18F0>
```

ELLIPSIS 指令为最后一个例子提供了一个不错的方法:

```
>>> C()
<__main__.C instance at 0x...>
```

浮点数在不同的平台上也会有小的输出变化，因为 Python 在浮点数的格式化上依赖于平台的 C 库，而 C 库在这个问题上的质量差异很大。:

```
>>> 1./7  # risky
0.14285714285714285
>>> print(1./7) # safer
0.142857142857
>>> print(round(1./7, 6)) # much safer
0.142857
```

形式 “I/2.**J“的数字在所有的平台上都是安全的，我经常设计一些测试的用例来产生该形式的数:

```
>>> 3./4  # utterly safe
0.75
```

简单的分数也更容易让人理解，这也使得文件更加完善。

26.7.4 基本 API

函数*testmod()* 和*testfile()* 为 doctest 提供了一个简单的接口，应该足以满足大多数基本用途。关于这两个函数的不太正式的介绍，请参见简单用法：检查 *Docstrings* 中的示例 和简单的用法：检查文本文件中的例子 部分。

doctest.**testfile**(*filename*, *module_relative=True*, *name=None*, *package=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *parser=DocTestParser()*, *encoding=None*)

除了 *filename*，所有的参数都是可选的，而且应该以关键字的形式指定。

测试名为 *filename* 的文件中的用例。返回 “(failure_count, test_count)“。

可选参数 *module_relative* 指定了文件名的解释方式。

- 如果 *module_relative* 是 `True` (默认)，那么 *filename* 指定一个独立于操作系统的模块相对路径。默认情况下，这个路径是相对于调用模块的目录的；但是如果指定了 *package* 参数，那么它就是相对于该包的。为了保证操作系统的独立性，*filename* 应该使用字符来分隔路径段，并且不能是一个绝对路径 (即不能以 / 开始)。
- 如果 *module_relative* 是 `False`，那么 *filename* 指定了一个操作系统特定的路径。路径可以是绝对的，也可以是相对的；相对路径是相对于当前工作目录而言的。

可选参数 *name* 给出了测试的名称；默认情况下，或者如果是 “None“，那么使用 “os.path.basename(filename)“。

可选参数 *package* 是一个 Python 包或一个 Python 包的名字，其目录应被用作模块相关文件名的基础目录。如果没有指定包，那么调用模块的目录将作为模块相关文件名的基础目录。如果 *module_relative* 是 “False“，那么指定 *package* 是错误的。

可选参数 *globs* 给出了一个在执行示例时用作全局变量的 dict。这个 dict 的一个新的浅层副本将为 doctest 创建，因此它的用例将从一个干净的地方开始。默认情况下，或者如果为 “None“，使用一个新的空 dict。

可选参数 *extraglobs* 给出了一个合并到用于执行用例全局变量中的 dict。这就像*dict.update()* 一样：如果 *globs* 和 *extraglobs* 有一个共同的键，那么 *extraglobs* 中的相关值会出现在合并的 dict 中。默认情况下，或者为 “None“ ，则不使用额外的全局变量。这是一个高级功能，允许对 doctest 进行参数化。例如，可以为一个基类写一个测试，使用该类的通用名称，然后通过传递一个 *extraglobs* dict，将通用名称映射到要测试的子类，从而重复用于测试任何数量的子类。

可选的参数 *verbose* 如果为真值会打印很多东西，如果为假值则只打印失败信息；默认情况下，或者为 `None`，只有当 `'-v'` 在 `sys.argv` 中时才为真值。

可选参数 *report* 为 True 时，在结尾处打印一个总结，否则在结尾处什么都不打印。在 verbose 模式下，总结是详细的，否则总结是非常简短的（事实上，如果所有的测试都通过了，总结就是空的）。

可选参数 *optionflags* （默认值为 0）是选项标志的 bitwise OR 。参见章节选项标记 。

可选参数 *raise_on_error* 默认为 False。如果是 True，在一个用例中第一次出现失败或意外的异常时，会触发一个异常。这允许对失败进行事后调试。默认行为是继续运行例子。

可选参数 *parser* 指定一个*DocTestParser* （或子类），它应该被用来从文件中提取测试。它默认为一个普通的解析器（即 “DocTestParser()“）。

可选参数 *encoding* 指定了一个编码，应该用来将文件转换为 unicode。

doctest.**testmod**(*m=None*, *name=None*, *globs=None*, *verbose=None*, *report=True*, *optionflags=0*, *extraglobs=None*, *raise_on_error=False*, *exclude_empty=False*)

所有的参数都是可选的，除了 *m* 之外，都应该以关键字的形式指定。

测试从模块 *m* （或模块*__main__* ，如果 *m* 没有被提供或为 “None“）可达到的函数和类的文档串中的用例，从 “m.__doc__“开始。

也测试从 dict `m.__test__` 可达到的用例，如果它存在并且不是 `None` 。`m.__test__` 将名字（字符串）映射到函数、类和字符串；函数和类的文档串被搜索到的用例；字符串被直接搜索到，就像它们是文档串一样。

只搜索附属于模块 *m* 中的对象的文档串。

返回 `(failure_count, test_count)` 。

可选参数 *name* 给出了模块的名称；默认情况下，或者如果为 `None` ，则为 `m.__name__` 。

可选参数 *exclude_empty* 默认为假值。如果为真值，没有找到任何 doctest 的对象将被排除在考虑范围之外。默认情况下是向后兼容，所以仍然使用 `doctest.master.summarize()` 和*testmod()* 的代码会继续得到没有测试的对象的输出。较新的*DocTestFinder* 构造器的 *exclude_empty* 参数默认为真值。

可选参数 *extraglobs* 、*verbose* 、*report* 、*optionflags* 、*raise_on_error* 和 *globs* 与上述函数*testfile()* 的参数相同，只是 *globs* 默认为 `m.__dict__` 。

doctest.**run_docstring_examples**(*f*, *globs*, *verbose=False*, *name="NoName"*, *compileflags=None*, *optionflags=0*)

与对象 *f* 相关的测试用例；例如，*f* 可以是一个字符串、一个模块、一个函数或一个类对象。

dict 参数 *globs* 的浅层拷贝被用于执行环境。

可选参数 *name* 在失败信息中使用，默认为 `"NoName"` 。

如果可选参数 *verbose* 为真，即使没有失败也会产生输出。默认情况下，只有在用例失败的情况下才会产生输出。

可选参数 *compileflags* 给出了 Python 编译器在运行例子时应该使用的标志集。默认情况下，或者如果为 `None` ，标志是根据 *globs* 中发现的未来特征集推导出来的。

可选参数 *optionflags* 的作用与上述*testfile()* 函数中的相同。

26.7.5 Unittest API

doctest 提供了两个函数，可以用来从模块和包含测试的文本文件中创建*unittest* 测试套件。要与*unittest* 测试发现集成，请在你的测试模块中包含一个 `load_tests()` 函数：：

```
import unittest
import doctest
import my_module_with_doctests

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(my_module_with_doctests))
    return tests
```

有两个主要函数用于从文本文件和带 doctest 的模块中创建*unittest.TestSuite* 实例。

doctest.**DocFileSuite**(**paths*, *module_relative=True*, *package=None*, *setUp=None*, *tearDown=None*, *globs=None*, *optionflags=0*, *parser=DocTestParser()*, *encoding=None*)

将一个或多个文本文件中的 doctest 测试转换为一个*unittest.TestSuite* 。

返回的*unittest.TestSuite* 将由 unittest 框架运行，并运行每个文件中的交互式示例。如果任何文件中的用例失败了，那么合成的单元测试就会失败，并触发一个 `failureException` 异常，显示包含该测试的文件名和一个（有时是近似的）行号。

传递一个或多个要检查的文本文件的路径（作为字符串）。

选项可以作为关键字参数提供：

可选参数 *module_relative* 指定了 *paths* 中的文件名应该如何解释。

- 如果 *module_relative* 是 `True`（默认值），那么 *paths* 中的每个文件名都指定了一个独立于操作系统的模块相对路径。默认情况下，这个路径是相对于调用模块的目录的；但是如果指定了 *package* 参数，那么它就是相对于该包的。为了保证操作系统的独立性，每个文件名都应该使用字符来分隔路径段，并且不能是绝对路径（即不能以 / 开始）。
- 如果 *module_relative* 是 `False`，那么 *paths* 中的每个文件名都指定了一个操作系统特定的路径。路径可以是绝对的，也可以是相对的；相对路径是关于当前工作目录的解析。

可选参数 *package* 是一个 Python 包或一个 Python 包的名字，其目录应该被用作 *paths* 中模块相关文件名的基本目录。如果没有指定包，那么调用模块的目录将作为模块相关文件名的基础目录。如果 *module_relative* 是 `False`，那么指定 *package* 是错误的。

可选的参数 *setUp* 为测试套件指定了一个设置函数。在运行每个文件中的测试之前，它被调用。*setUp* 函数将被传递给一个*DocTest* 对象。setUp 函数可以通过测试的 *globs* 属性访问测试的全局变量。

可选的参数 *tearDown* 指定了测试套件的卸载函数。在运行每个文件中的测试后，它会被调用。*tearDown* 函数将被传递一个*DocTest* 对象。setUp 函数可以通过测试的 *globs* 属性访问测试的全局变量。

可选参数 *globs* 是一个包含测试的初始全局变量的字典。这个字典的一个新副本为每个测试创建。默认情况下，*globs* 是一个新的空字典。

可选参数 *optionflags* 为测试指定默认的 doctest 选项，通过将各个选项的标志连接在一起创建。参见章节选项标记 。参见下面的函数*set_unittest_reportflags()* ，以了解设置报告选项的更好方法。

可选参数 *parser* 指定一个*DocTestParser* （或子类），它应该被用来从文件中提取测试。它默认为一个普通的解析器（即 “DocTestParser()“）。

可选参数 *encoding* 指定了一个编码，应该用来将文件转换为 unicode。

该全局 `__file__` 被添加到提供给用*DocFileSuite()* 从文本文件加载的 doctest 的全局变量中。

doctest.**DocTestSuite**(*module=None*, *globs=None*, *extraglobs=None*, *test_finder=None*, *setUp=None*, *tearDown=None*, *checker=None*)

将一个模块的 doctest 测试转换为*unittest.TestSuite* 。

返回的*unittest.TestSuite* 将由 unittest 框架运行，并运行模块中的每个 doctest。如果任何一个测试失败，那么合成的单元测试就会失败，并触发一个 `failureException` 异常，显示包含该测试的文件名和一个（有时是近似的）行号。

可选参数 *module* 提供了要测试的模块。它可以是一个模块对象或一个（可能是带点的）模块名称。如果没有指定，则使用调用此函数的模块。

可选参数 *globs* 是一个包含测试的初始全局变量的字典。这个字典的一个新副本为每个测试创建。默认情况下，*globs* 是一个新的空字典。

可选参数 *extraglobs* 指定了一组额外的全局变量，这些变量被合并到 *globs* 中。默认情况下，不使用额外的全局变量。

可选参数 *test_finder* 是*DocTestFinder* 对象（或一个可替换的对象），用于从模块中提取测试。

可选参数 *setUp* 、*tearDown* 和 *optionflags* 与上述函数*DocFileSuite()* 相同。

这个函数使用与*testmod()* 相同的搜索技术。

3.5 版更變: 如果 *module* 不包含任何文件串，则*DocTestSuite()* 返回一个空的*unittest.TestSuite*，而不是触发*ValueError*。

DocTestSuite() 从 `doctest.DocTestCase` 实例中创建一个*unittest.TestSuite* ，而 `DocTestCase` 是*unittest.TestCase* 的子类。`DocTestCase` 在这里没有记录（这是一个内部细节），但研究其代码可以回答关于*unittest* 集成的实际细节问题。

同样，*DocFileSuite()* 从 `doctest.DocFileCase` 实例中创建一个*unittest.TestSuite* ，并且 `DocFileCase` 是 `DocTestCase` 的一个子类。

所以这两种创建*unittest.TestSuite* 的方式都是运行 DocTestCase 的实例。这一点很重要，因为有一个微妙的原因：当你自己运行*doctest* 函数时，你可以直接控制使用中的*doctest* 选项，通过传递选项标志给*doctest* 函数。然而，如果你正在编写一个*unittest* 框架，*unittest* 最终会控制测试的运行时间和方式。框架作者通常希望控制*doctest* 的报告选项（也许，例如，由命令行选项指定），但没有办法通过*unittest* 向*doctest* 测试运行者传递选项。

出于这个原因，*doctest* 也支持一个概念，即*doctest* 报告特定于*unittest* 支持的标志，通过这个函数:

doctest.**set_unittest_reportflags**(*flags*)
设置要使用的*doctest* 报告标志。

参数 *flags* 是选项标志的 bitwise OR 。参见章节选项标记 。只有” 报告标志” 可以被使用。

这是一个模块的全局设置，并影响所有未来由模块*unittest* 运行的测试：DocTestCase 的 runTest() 方法会查看 DocTestCase 实例构建时为测试用例指定的选项标志。如果没有指定报告标志（这是典型的和预期的情况），*doctest* 的*unittest* 报告标志被 bitwise ORed 到选项标志中，这样增加的选项标志被传递给*DocTestRunner* 实例来运行 doctest。如果在构建 DocTestCase 实例时指定了任何报告标志，那么*doctest* 的*unittest* 报告标志会被忽略。

unittest 报告标志的值在调用该函数之前是有效的，由该函数返回。

26.7.6 高级 API

基本 API 是一个简单的封装，旨在使 doctest 易于使用。它相当灵活，应该能满足大多数用户的需求；但是，如果你需要对测试进行更精细的控制，或者希望扩展 doctest 的功能，那么你应该使用高级 API 。

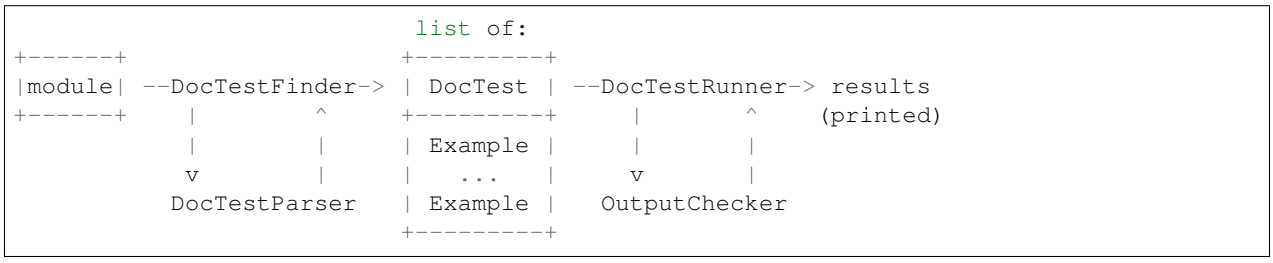
高级 API 围绕着两个容器类，用于存储从 doctest 案例中提取的交互式用例:

- *Example*: 一个单一的 Python *statement* ，与它的预期输出配对。
- *DocTest*: 一组*Example*s 的集合，通常从一个文档字符串或文本文件中提取。

定义了额外的处理类来寻找、解析和运行，并检查 doctest 的用例。

- *DocTestFinder*: 查找给定模块中的所有文档串，并使用*DocTestParser* 从每个包含交互式用例的文档串中创建一个*DocTest* 。
- *DocTestParser*: 从一个字符串（如一个对象的文档串）创建一个*DocTest* 对象。
- *DocTestRunner*: 执行*DocTest* 中的用例，并使用*OutputChecker* 来验证其输出。
- *OutputChecker*: 将一个测试用例的实际输出与预期输出进行比较，并决定它们是否匹配。

这些处理类之间的关系总结在下图中:



DocTest 对象

class doctest.**DocTest**(*examples*, *globs*, *name*, *filename*, *lineno*, *docstring*)
应该在单一命名空间中运行的 doctest 用例的集合。构造函数参数被用来初始化相同名称的属性。

DocTest 定义了以下属性。它们由构造函数初始化，不应该被直接修改。

examples
一个*Example* 对象的列表，它编码了应该由该测试运行的单个交互式 Python 用例。

globs
例子应该运行的命名空间（又称 globals ）。这是一个将名字映射到数值的字典。例子对名字空间的任何改变（比如绑定新的变量）将在测试运行后反映在*globs* 中。

name
识别*DocTest* 的字符串名称。通常情况下，这是从测试中提取的对象或文件的名称。

filename
这个*DocTest* 被提取的文件名；或者为 “None“，如果文件名未知，或者*DocTest* 没有从文件中提取。

lineno
filename 中的行号，这个*DocTest* 开始的地方，或者行号不可用时为 “None“。这个行号相对于文件的开头来说是零的。

docstring
从测试中提取的字符串，或者如果字符串不可用，或者为 None ，如果测试没有从字符串中提取。

Example 对象

class doctest.**Example**(*source*, *want*, *exc_msg=None*, *lineno=0*, *indent=0*, *options=None*)
单个交互式用例，由一个 Python 语句及其预期输出组成。构造函数参数被用来初始化相同名称的属性。

Example 定义了以下属性。它们由构造函数初始化，不应该被直接修改。

source
一个包含该用例源码的字符串。源码由一个 Python 语句组成，并且总是以换行结束；构造函数在必要时添加一个换行。

want
运行这个用例的源码的预期输出（可以是 stdout ，也可以是异常情况下的回溯）。*want* 以一个换行符结束，除非没有预期的输出，在这种情况下它是一个空字符串。构造函数在必要时添加一个换行。

exc_msg
用例产生的异常信息，如果这个例子被期望产生一个异常；或者为 None ，如果它不被期望产生一个异常。这个异常信息与*traceback.format_exception_only()* 的返回值进行比较。*exc_msg* 以换行结束，除非是 None 。

lineno
包含本例的字符串中的行号，即本例的开始。这个行号相对于包含字符串的开头来说是以零开始的。

indent
用例在包含字符串中的缩进，即在用例的第一个提示前有多少个空格字符。

options
一个从选项标志到 True 或 False 的字典映射，用于覆盖这个例子的默认选项。任何不包含在这个字典中的选项标志都被保留为默认值（由*DocTestRunner* 的 optionflags 指定）。默认情况下，没有选项被设置。

DocTestFinder 对象

class doctest.**DocTestFinder**(*verbose=False*, *parser=DocTestParser()*, *recurse=True*, *exclude_empty=True*)

一个处理类，用于从一个给定的对象的 docstring 和其包含的对象的 docstring 中提取与之相关的*DocTest* 。*DocTest* 可以从模块、类、函数、方法、静态方法、类方法和属性中提取。

可选的参数 *verbose* 可以用来显示查找器搜索到的对象。它的默认值是 `False` （无输出）。

可选的参数 *parser* 指定了*DocTestParser* 对象（或一个可替换的对象），用于从文档串中提取 doctest 。

如果可选的参数 *recurse* 是 False ，那么*DocTestFinder.find()* 将只检查给定的对象，而不是任何包含的对象。

如果可选参数 *exclude_empty* 为 False ，那么*DocTestFinder.find()* 将包括对文档字符串为空的对象的测试。

DocTestFinder 定义了以下方法：

find(*obj[, name][, module][, globs][, extraglobs]*)

返回*DocTest* 的列表，该列表由 *obj* 的文档串或其包含的任何对象的文档串定义。

可选参数 *name* 指定了对象的名称；这个名称将被用来为返回的*DocTest* 构建名称。如果没有指定 *name*，则使用 `obj.__name__` 。

可选参数 *module* 是包含给定对象的模块。如果没有指定模块或者是 `None` ，那么测试查找器将试图自动确定正确的模块。该对象被使用的模块：

- 作为一个默认的命名空间，如果没有指定 *globs* 。
- 为了防止 DocTestFinder 从其他模块导入的对象中提取 DocTest 。(包含有除 *module* 以外的模块的对象会被忽略)。
- 找到包含该对象的文件名。
- 找到该对象在其文件中的行号。

如果 *module* 是 `False` ，将不会试图找到这个模块。这是不明确的，主要用于测试 doctest 本身：如果 *module* 是 `False` ，或者是 `None` 但不能自动找到，那么所有对象都被认为属于（不存在的）模块，所以所有包含的对象将（递归地）被搜索到 doctest 。

每个*DocTest* 的 globals 是由 *globs* 和 *extraglobs* 组合而成的（*extraglobs* 的绑定覆盖 *globs* 的绑定）。为每个*DocTest* 创建一个新的 globals 字典的浅层拷贝。如果没有指定 *globs* ，那么它默认为模块的 *__dict__* ，如果指定了或者为 `{}` ，则除外。如果没有指定 *extraglobs* ，那么它默认为 `{}` 。

DocTestParser 对象

class doctest.**DocTestParser**

一个处理类，用于从一个字符串中提取交互式的用例，并使用它们来创建一个*DocTest* 对象。

DocTestParser 定义了以下方法：

get_doctest(*string*, *globs*, *name*, *filename*, *lineno*)

从给定的字符串中提取所有的测试用例，并将它们收集到一个*DocTest* 对象中。

globs 、*name* 、*filename* 和 *lineno* 是新的*DocTest* 对象的属性。更多信息请参见*DocTest* 的文档。

get_examples(*string*, *name='<string>'*)

ä»□ç»□å®□ç□□å□ç¬¦ä¸²ä¸æ□□å□□æ□□æ□□ç□□æµ□è¯□ç□¨ä¾□ï¼□å¹¶ä»¥ *Example* å¯¹è±¡å□□è¡¨ç□□å½¢å¼□è¿□å□□ã□□ è¡□æ□°ä»¥ 0 ä¸ºå□ºç¡□ã□□ å□¯é□□å□□æ□° *name* æ□¯è¯□å□«è¿□ä¸ªå□ç¬¦ä¸²ç□□å□□ç§°ï¼□å□ªç□¨äº□é□□è¯¯ä¿¡æ□¯ã□□

parse(*string*, *name='<string>'*)
å°?ç»?å®?ç???å?ç¬¦ä¸²å???æ???ç?¨ä¾?å???ä¸é?´æ???æ?¬ï¼?å¹¶ä»¥ *Example* å???å?ç¬¦ä¸²ä¸ªæ¤?ç?????å???è¡¨å½¢å¼?è¿?å???å??? *Example* ç???è¡?å? · ä»¥ 0 ä¸ºå?ºç¡?ã??å?¯é???å???æ?° *name* æ?¯è¯?å«è¿?ä¸ªå?ç¬¦ä¸²ç???å???ç§°ï¼?å?ªç?¨ä?º é???è¯¯ä¿¡æ?¯ã??

DocTestRunner 对象

class doctest.**DocTestRunner**(*checker=None*, *verbose=None*, *optionflags=0*)
ä¸?ä¸ªå¤?ç???ç±»ï¼?ç?¨äº?æ§è¡?å???é?è¯? *DocTest* ä¸ç???ä¤ºä¾?å¼?ç?¨ã??

é¢?æ???è¾¾å?ºå???å®?é? è¾¾å?ºä¹?é?´ç???æ¯è¾¾å?¦ç?±ä¸ª *OutputChecker* å®?æ???ç???ã??è¿?ç§?æ¯è¾¾å?¦ä»¥ç?¨ä¸ä¸ªé???é¡¹æ ?å¿?æ¥å®?å¶?ï¼?æ?¯å¤?ä¸ª 选项标记 é?¨å???ã??å¦?æ???é???é¡¹æ ?å¿?ä¸ä¤ºï¼?é?£ä¹?ä¹?å?¯ä»¥é???è¿?å???æ???é? å?½æ°ä¼ é?? *OutputChecker* ç???å?ç±»æ?¥å®?å?¶æ¯è¾¾å?¦ã??

æµ?è¯?è¿?è¡?ç???æ¾?ç¤ºè¾¾å?¦å?¯ä»¥é???è¿?ä¸ç§?æ?¹ä¼?æ§å?¶ã???é¡¹å ?ï¼?ä¸ªè¾¾å?¦å½?æ°å?¯ä»¥è¢«ä¼ é??ç»? TestRunner.run() ï¼?è¿?ä¸ªå½?æ°å°?è¢«ç«?è?ç?¨ï¼?å¹¶ä¸?æ¾?ç¤ºå?¯å?¯ä»¥é???è®¤å¼?æ?¯ sys.stdout.write ã?? å¦?æ???å?ªæ???è?· è¾¾å?ºæ?¯ä¸å¤?ç???ï¼?é?£ä¹?ä¹?å?¯ä»¥é???è¿?å?ç±»å???DocTestRunner ï¼?å¹¶é???å??? *report_start()* ， *report_success()* ， *report_unexpected_exception()* ， å?? *report_failure()* æ?¹æ³?æ?¥å®?å?¶æ¾?ç¤ºè¾¾å?¦ã??

å?¯é???ç???å ³é®?å?????æ?° *checker* æ???å®?äº? *OutputChecker* å¯¹è±¡ï¼?æ???æ?¿æ?¢ï¼?ï¼?å®?å?¶è¯¥è?¥å¦?ä½? doctest ç?¨ä¾?ç???å®?é? è¾¾å?¦ã??

å?¯é???ç???å ³é®?å?????æ?° *verbose* æ?§å?¶ *DocTestRunner* ç???è¯¦ç»?å°?ã?? å¦?æ??? *verbose* æ?¯ True ï¼?é?£ä¹?æ¯ä¸ªç?¨ä¾?æ?¯é½ä¼?è¢«æ??å?°æ?¥ï¼?å½?å®?æ£å?¨è¿?æ?¶ã??å¦?æ??? *verbose*æ?¯ *"False"* ï¼?å?????å?ªæ??å?°å¤±è´¥ç?¨ä¾?å¦?æ?¯ *verbose* æ²¡æ???æ???å®?ï¼?æ?¯é??è? ä¸º None ï¼?å¦?æ???å½?ä½¿ç?¨ä»»è¡?å¼? -v ï¼?å???ä½¿ç?¨verboseè¾¾å?¦ã??

å?¯é???ç???å ³é®?å?????æ?° *optionflags* å?¯ä»¥ç?¨æ¥?æ§å?¶æµ?è¯?è¿?è¡?å?¯ä¦?ä½?æ¯è¾¾å?¦é¢?æ???è¾¾å?ºï¼?ä»¥å???å¦?ä½?æ¾?ç¤ºå¤±è´¥ã???æ?´å¤?å¤±è´¥ä¿¡æ?¯ï¼?è¯¯ · å???è§?ç« è?? 选项标记 ã??

DocTestParser 定义了以下方法：

report_start(*out*, *test*, *example*)
报告测试运行器即将处理给定的用例。提供这个方法是为了让*DocTestRunner* 的子类能够定制他们的输出；它不应该被直接调用。

example 是即将被处理的用。*test* 是 包含用例的测试。*out* 是传递给*DocTestRunner.run()* 的输出函数。

report_success(*out*, *test*, *example*, *got*)
报告给定的例子运行成功。提供这个方法是为了让*DocTestRunner* 的子类能够定制他们的输出；它不应该被直接调用。

example 是即将被处理的用例。*got* 是这个例子的实际输出。*test* 是包含 *example* 的测试。*out* 是传递给*DocTestRunner.run()* 的输出函数。

report_failure(*out*, *test*, *example*, *got*)
报告给定的用例运行失败。提供这个方法是为了让*DocTestRunner* 的子类能够定制他们的输出；它不应该被直接调用。

example 是即将被处理的用例。*got* 是这个例子的实际输出。*test* 是包含 *example* 的测试。*out* 是传递给*DocTestRunner.run()* 的输出函数。

report_unexpected_exception(*out*, *test*, *example*, *exc_info*)
报告给定的用例触发了一个异常。提供这个方法是为了让*DocTestRunner* 的子类能够定制他们的输出；它不应该被直接调用。

example 是将要被处理的用例。*exc_info* 是一个元组，包含关于异常的信息（如由*sys.exc_info()*返回）。*test* 是包含 *example* 的测试。*out* 是传递给*DocTestRunner.run()* 的输出函数。

run(*test*, *compileflags=None*, *out=None*, *clear_globs=True*)
在 *test* （一个*DocTest* 对象）中运行这些用例，并使用写入函数 *out* 显示结果。

这些用例都是在命名空间 `test.globs` 中运行的。如果 *clear_globs* 为 True （默认），那么这个命名空间将在测试运行后被清除，以帮助进行垃圾回收。如果你想在测试完成后检查命名空间，那么使用 *clear_globs=False* 。

compileflags 给出了 Python 编译器在运行例子时应该使用的标志集。如果没有指定，那么它将默认为适用于 *globs* 的 future-import 标志集。

每个用例的输出都使用*DocTestRunner* 的输出检查器进行检查，结果由 `DocTestRunner.report_*()` 方法进行格式化。

summarize(*verbose=None*)
打印这个 DocTestRunner 运行过的所有测试用例的摘要，并返回一个*named tuple* `TestResults(failed, attempted)` 。

可选的 *verbose* 参数控制摘要的详细程度。如果没有指定 verbose ，那么将使用*DocTestRunner* 的 verbose 。

OutputChecker 对象

class doctest.**OutputChecker**
一个用于检查测试用例的实际输出是否与预期输出相匹配的类。*OutputChecker* 定义了两个方法：*check_output()* ，比较给定的一对输出，如果它们匹配则返回 `True` ；*output_difference()* ，返回一个描述两个输出之间差异的字符串。

OutputChecker 定义了以下方法：

check_output(*want*, *got*, *optionflags*)
如果一个用例的实际输出（*got* ）与预期输出（*want* ）匹配，则返回 `True` 。如果这些字符串是相同的，总是被认为是匹配的；但是根据测试运行器使用的选项标志，也可能有几种非精确的匹配类型。参见章节选项标记 了解更多关于选项标志的信息。

output_difference(*example*, *got*, *optionflags*)
返回一个字符串，描述给定用例（*example* ）的预期输出和实际输出（*got* ）之间的差异。*optionflags* 是用于比较 *want* 和 *got* 的选项标志集。

26.7.7 调试

Doctest 提供了几种调试 doctest 用例的机制：

- 有几个函数将测试转换为可执行的 Python 程序，这些程序可以在 Python 调试器，*pdb* 下运行。
- *DebugRunner* 类是*DocTestRunner* 的一个子类，它为第一个失败的用例触发一个异常，包含关于这个用例的信息。这些信息可以用来对这个用例进行事后调试。
- 由*DocTestSuite()* 生成的*unittest* 用例支持由*unittest.TestCase* 定义的*debug()* 方法，。
- 你可以在 doctest 的用例中加入对*pdb.set_trace()* 的调用，当这一行被执行时，你会进入 Python 调试器。然后你可以检查变量的当前值，等等。例如，假设 `a.py` 只包含这个模块 docstring

```
"""
>>> def f(x):
...     g(x*2)
```

(下页继续)

(繼續上一頁)

```
>>> def g(x):
...     print(x+3)
...     import pdb; pdb.set_trace()
>>> f(3)
9
"""
```

那么一个交互式 Python 会话可能是这样的:

```
>>> import a, doctest
>>> doctest.testmod(a)
--Return--
> <doctest a[1]>(3)g()->None
-> import pdb; pdb.set_trace()
(Pdb) list
  1     def g(x):
  2         print(x+3)
  3  ->     import pdb; pdb.set_trace()
[EOF]
(Pdb) p x
6
(Pdb) step
--Return--
> <doctest a[0]>(2)f()->None
-> g(x*2)
(Pdb) list
  1     def f(x):
  2  ->     g(x*2)
[EOF]
(Pdb) p x
3
(Pdb) step
--Return--
> <doctest a[2]>(1)?()->None
-> f(3)
(Pdb) cont
(0, 3)
>>>
```

将测试转换为 Python 代码的函数，并可能在调试器下运行合成的代码:

doctest.**script_from_examples**(*s*)

将带有用例的文本转换为脚本。

参数 *s* 是一个包含测试用例的字符串。该字符串被转换为 Python 脚本，其中 *s* 中的 doctest 用例被转换为常规代码，其他的都被转换为 Python 注释。生成的脚本将以字符串的形式返回。例如，

```
import doctest
print(doctest.script_from_examples(r"""
    Set x and y to 1 and 2.
    >>> x, y = 1, 2

    Print their sum:
    >>> print(x+y)
    3
"""))
```

显示:

```
# Set x and y to 1 and 2.
x, y = 1, 2
#
# Print their sum:
print(x+y)
# Expected:
## 3
```

这个函数在内部被其他函数使用（见下文），但当你想把一个交互式 Python 会话转化为 Python 脚本时，也会很有用。

doctest.**testsource**(*module*, *name*)

将一个对象的 doctest 转换为一个脚本。

参数 *module* 是一个模块对象，或者一个模块带点的名称，包含对其 doctest 感兴趣的对象。参数 *name* 是具有感兴趣的测试的对象的名称（在模块中）。结果是一个字符串，包含该对象的文本串转换为 Python 脚本，如上面*script_from_examples()* 所述。例如，如果模块 `a.py` 包含一个顶级函数 `f()`，那么

```
import a, doctest
print(doctest.testsource(a, "a.f"))
```

打印函数 `f()` 的文档串的脚本版本，将测试转换为代码，其余部分放在注释中。

doctest.**debug**(*module*, *name*, *pm=False*)

对一个对象的 doctest 进行调试。

module 和 *name* 参数与上面函数*testsource()* 的参数相同。被命名对象的文本串的合成 Python 脚本被写入一个临时文件，然后该文件在 Python 调试器*pdb* 的控制下运行。

`module.__dict__` 的一个浅层拷贝被用于本地和全局的执行环境。

可选参数 *pm* 控制是否使用事后调试。如果 *pm* 为 True，则直接运行脚本文件，只有当脚本通过引发一个未处理的异常而终止时，调试器才会介入。如果是这样，就会通过*pdb.post_mortem()* 调用事后调试，并传递未处理异常的跟踪对象。如果没有指定 *pm*，或者是 False，脚本将从一开始就在调试器下运行，通过传递一个适当的*exec()* 调用给*pdb.run()*。

doctest.**debug_src**(*src*, *pm=False*, *globs=None*)

在一个字符串中调试 doctest。

这就像上面的函数*debug()*，只是通过 *src* 参数，直接指定一个包含测试用例的字符串。

可选参数 *pm* 的含义与上述函数*debug()* 的含义相同。

可选的参数 *globs* 给出了一个字典，作为本地和全局的执行环境。如果没有指定，或者为 `None`，则使用一个空的字典。如果指定，则使用字典的浅层拷贝。

DebugRunner 类，以及它可能触发的特殊异常，是测试框架作者最感兴趣的，在此仅作简要介绍。请看源代码，特别是*DebugRunner* 的文档串（这是一个测试！）以了解更多细节。

class doctest.**DebugRunner**(*checker=None*, *verbose=None*, *optionflags=0*)

DocTestRunner 的一个子类，一旦遇到失败，就会触发一个异常。如果一个意外的异常发生，就会引发一个*UnexpectedException* 异常，包含测试、用例和原始异常。如果输出不匹配，那么就会引发一个*DocTestFailure* 异常，包含测试、用例和实际输出。

关于构造函数参数和方法的信息，请参见*DocTestRunner* 部分的文档高级 *API*。

DebugRunner 实例可能会触发两种异常。

exception doctest.**DocTestFailure**(*test*, *example*, *got*)

DocTestRunner 触发的异常，表示一个 doctest 用例的实际输出与预期输出不一致。构造函数参数被用来初始化相同名称的属性。

DocTestFailure 定义了以下属性:

DocTestFailure.**test**
当该用例失败时正在运行的*DocTest* 对象。

DocTestFailure.**example**
失败的*Example* 。

DocTestFailure.**got**
用例的实际输出。

exception doctest.**UnexpectedException**(*test*, *example*, *exc_info*)
一个由*DocTestRunner* 触发的异常，以提示一个 doctest 用例引发了一个意外的异常。构造函数参数被用来初始化相同名称的属性。

UnexpectedException 定义了以下属性:

UnexpectedException.**test**
当该用例失败时正在运行的*DocTest* 对象。

UnexpectedException.**example**
失败的*Example* 。

UnexpectedException.**exc_info**
一个包含意外异常信息的元组，由 sys.ex_info() 返回。

26.7.8 肥皂盒

正如介绍中提到的，*doctest* 已经发展到有三个主要用途:

1. 检查 docstring 中的用例。
2. 回归测试
3. 可执行的文档/文字测试。

这些用途有不同的要求，区分它们是很重要的。特别是，用晦涩难懂的测试用例来填充你的文档字符串会使文档变得很糟糕。

在编写文档串时，要谨慎地选择文档串的用例。这是一门需要学习的艺术——一开始可能并不自然。用例应该为文档增加真正的价值。一个好的用例往往可以抵得上许多文字。如果用心去做，这些用例对你的用户来说是非常有价值的，而且随着时间的推移和事情的变化，收集这些用例所花费的时间也会得到很多倍的回报。我仍然惊讶于我的*doctest* 用例在一个“无害”的变化后停止工作。

Doctest 也是回归测试的一个很好的工具，特别是如果你不吝啬解释的文字。通过交错的文本和用例，可以更容易地跟踪真正的测试，以及为什么。当测试失败时，好的文本可以使你更容易弄清问题所在，以及如何解决。的确，你可以在基于代码的测试中写大量的注释，但很少有程序员这样做。许多人发现，使用 doctest 方法反而能使测试更加清晰。也许这只是因为 doctest 使写散文比写代码容易一些，而在代码中写注释则有点困难。我认为它比这更深入：当写一个基于 doctest 的测试时，自然的态度是你想解释你的软件的细微之处，并用例子来说明它们。这反过来又自然地导致了测试文件从最简单的功能开始，然后逻辑地发展到复杂和边缘案例。一个连贯的叙述是结果，而不是一个孤立的函数集合，似乎是随机的测试孤立的功能位。这是一种不同的态度，产生不同的结果，模糊了测试和解释之间的区别。

回归测试最好限制在专用对象或文件中。有几种组织测试的选择:

- 编写包含测试案例的文本文件作为交互式例子，并使用*testfile()* 或*DocFileSuite()* 来测试这些文件。建议这样做，尽管对于新的项目来说是最容易做到的，从一开始就设计成使用 doctest 。
- 定义命名为 _regrtest_topic 的函数，由单个文档串组成，包含命名主题的测试用例。这些函数可以包含在与模块相同的文件中，或分离出来成为一个单独的测试文件。
- 定义一个从回归测试主题到包含测试用例的文档串的 __test__ 字典映射。

当你把你的测试放在一个模块中时，这个模块本身就可以成为测试运行器。当一个测试失败时，你可以安排你的测试运行器只重新运行失败的测试，同时调试问题。下面是这样一个测试运行器的最小例子:

```
if __name__ == '__main__':
    import doctest
    flags = doctest.REPORT_NDIFF|doctest.FAIL_FAST
    if len(sys.argv) > 1:
        name = sys.argv[1]
        if name in globals():
            obj = globals()[name]
        else:
            obj = __test__[name]
        doctest.run_docstring_examples(obj, globals(), name=name,
                                       optionflags=flags)
    else:
        fail, total = doctest.testmod(optionflags=flags)
        print("{} failures out of {} tests".format(fail, total))
```

🄵解

26.8 unittest --- 單元測試框架

源代码： Lib/unittest/__init__.py

(假如你已經熟悉相關基礎的測試概念，你可能會希望跳過以下段落，直接参考*assert 方法清單*。)

unittest 原生的單元測試框架最初由 JUnit 開發，和其他程式語言相似有主要的單元測試框架。支援自動化測試，對測試分享安裝與關閉程式碼，集合所有匯總的測試，🄵且獨立各個測試報告框架。

unittest 用來作🄵實現支援一些重要的物件導向方法的概念。

test fixture *test fixture* 表示为了开展一项或多项测试所需要进行的准备工作，以及所有相关的清理操作。举个例子，这可能包含创建临时或代理的数据库、目录，再或者启动一个服务器进程。

test case（測試用例） 一個 *test case* 是一個獨立的單元測試。這是用來確認一個特定設定的輸入的特殊回饋。*unittest* 提供一個基礎類🄵，類🄵*TestCase*，可以用來建立一個新的測試條例。

test suite（測試套件） *test suite* 是一個搜集測試條例，測試套件，或是兩者皆有。它需要一起被執行🄵用來匯總測試。

test runner（測試執行器） *test runner* 是一個編排測試執行與提供結果給使用者的一個元件。執行器可以使用圖形化介面，文字介面或是回傳一個特🄴值用來標示出執行測試的結果。

也參考:

***doctest* 模組** 另一個執行測試的模組，但使用不一樣的測試方法與規範。

Simple Smalltalk Testing: With Patterns Kent Beck 的原始論文討論使用*unittest* 這樣模式的測試框架。

pytest 第三方单元测试框架，提供轻量化的语法来编写测试，例如：`assert func(10) == 42`。

The Python Testing Tools Taxonomy 一份詳細的 Python 測試工具列表，包含 functional testing 框架和 mock object 函式庫。

Testing in Python Mailing List 一個專門興趣的群組用來討論 Python 中的測試方式與測試工具。

随 Python 源代码分发的脚本 `Tools/unittestgui/unittestgui.py` 是一个一个用于发现和执行测试的用户图形界面工具。这主要是为了方便新手使用单元测试制作的。在生产环境中，测试应由一个持续集成（CI）系统运行，如 Buildbot, Jenkins ， Travis-CI, 或 AppVeyor。

26.8.1 簡單範例

unittest 模組提供一系列豐富的工具用來建構與執行測試。本節將展示這一系列工具中一部份，它們已能滿足大部份使用者需求。

這是一段簡短的🄵本用來測試 3 個字串方法：

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

測試用例 (testcase) 可以透過繼承*unittest.TestCase* 類🄵來建立。這🄵定義了三個獨立的物件方法，名稱皆以 `test` 開頭。這樣的命名方式能告知 test runner 哪些物件方法🄵定義的測試。

每個測試的關鍵🄵呼叫*assertEqual()* 來確認是否🄵期望的結果；*assertTrue()* 或是*assertFalse()* 用來驗證一個條件式；*assertRaises()* 用來驗證是否觸發一個特定的 exception。使用這些物件方法來取代 `assert` 陳述句，將能使 test runner 收集所有的測試結果🄵🄵生一個報表。

通过*setUp()* 和*tearDown()* 方法，可以设置测试开始前与完成后需要执行的指令。在组织你的测试代码中，对此有更为详细的描述。

最後將顯示一個簡單的方法去執行測試*unittest.main()* 提供一個命令執行列介面測試🄵本。當透過命令執行列執行，輸出結果將會像是：

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

在測試時加入 `-v` 選項將指示*unittest.main()* 提高 verbosity 層級，🄵生以下的輸出：

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

以上的例子顯示大多數使用*unittest* 特徵足以滿足大多數日常測試的需求。接下來第一部分文件的剩余部分將繼續探索完整特徵設定。

26.8.2 命令執行列介面 (Command-Line Interface)

單元測試模組可以透過命令執行列執行測試模組，物件甚至個🄵的測試方法：

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

你可以通過一個串列與任何模組名稱的組合，完全符合類🄵與方法的名稱。

測試模組可以根據檔案路徑指定：

```
python -m unittest tests/test_something.py
```

這允許你使用 shell 檔案名稱補完功能 (filename completion) 來指定測試模組。給定的檔案路徑必須亦能被當作模組 import。此路徑轉🄵🄵模組名稱的方式🄵移除'.py' 🄵將路徑分隔符 (path separator) 轉🄵成'.'。假如你的測試檔案無法被 import 成模組，你應該直接執行該測試檔案。

通過增加 -v 的旗標數，可以在你執行測試時得到更多細節（更高的 verbosity）：

```
python -m unittest -v test_module
```

若執行時不代任何引數，將執行*Test Discovery*（測試探索）：

```
python -m unittest
```

列出所有命令列選項：

```
python -m unittest -h
```

3.2 版更變: 在早期的版本可以個🄵執行測試方法和不需要模組或是類🄴。

命令列模式選項

unittest supports these command-line options:

-b, --buffer
Standard output 與 standard error stream 將在測試執行被緩衝 (buffer)。這些輸出在測試通過時被🄵🄵。若是測試錯誤或失則，這些輸出將會正常地被印出，🄵且被加入至錯誤訊息中。

-c, --catch
Control-C 測試執行過程中等待正確的測試結果🄵回報目前🄵止所有的測試結果。第二個 Control-C 🄵出一般例外*KeyboardInterrupt*。

參照*Signal Handling* 針對函式提供的功能。

-f, --failfast
在第一次錯誤或是失敗停止執行測試。

-k
Only run test methods and classes that match the pattern or substring. This option may be used multiple times, in which case all test cases that match any of the given patterns are included.

包含通配符（*）的模式使用*fnmatch.fnmatchcase()* 对测试名称进行匹配。另外，该匹配是大小写敏感的。

模式对测试加载器导入的测试方法全名进行匹配。

例如，`-k foo` 可以匹配到 `foo_tests.SomeTest.test_something` 和 `bar_tests.SomeTest.test_foo`，但是不能匹配到 `bar_tests.FooTest.test_something`。

--locals
透過 traceback 顯示本地變數。

3.2 版新加入: 增加命令列模式選項 `-b`、`-c` 與 `-f`。

3.5 版新加入: 命令列選項 `--locals`。

3.7 版新加入: 命令行选项 `-k`。

對執行所有的專案或是一個子集合測試，命令列模式可以可以被用來做測試探索。

26.8.3 Test Discovery（測試探索）

3.2 版新加入.

單元測試支援簡單的 test discovery（測試探索）。🄵了相容於測試探索，所有的測試檔案都要是模組或是套件（包含*namespace packages*），🄵能從專案的最上層目🄵中 import（代表它們的檔案名稱必須是有效的 identifiers）。

Test discovery（測試探索）實作在*`TestLoader.discover()`*，但也可以被用於命令列模式。基本的命令列模式用法如下：

```
cd project_directory
python -m unittest discover
```

備🄵： `python -m unittest` 作🄵捷徑，其功能相當於 `python -m unittest discover`。假如你想傳遞引數至探索測試的話，一定要明確地加入 `discover` 子指令。

`discover` 子指令有以下幾個選項：

-v, --verbose
詳細（verbose）輸出

-s, --start-directory directory
開始尋找的資料夾（預設🄵 `.`）

-p, --pattern pattern
匹配測試檔案的模式（預設🄵 `test*.py`）

-t, --top-level-directory directory
專案的最高階層目🄵 (defaults to start directory)

`-s`, *`-p`*, 和*`-t`* 選項依照傳遞位置作🄵引數排序順序。以下兩個命令列被視🄵等價：

```
python -m unittest discover -s project_directory -p "*_test.py"
python -m unittest discover project_directory "*_test.py"
```

正如可以传入路径那样，传入一个包名作为起始目录也是可行的，如 `myproject.subpackage.test`。你提供的包名会被导入，它在文件系统中的位置会被作为起始目录。

警示： 探索性测试通过导入测试对测试进行加载。在找到所有你指定的开始目录下的所有测试文件后，它把路径转换为包名并进行导入。如 `foo/bar/baz.py` 会被导入为 `foo.bar.baz`。

如果你有一个全局安装的包，并尝试对这个包的副本进行探索性测试，可能会从错误的地方开始导入。如果出现这种情况，测试会输出警告并退出。

如果你使用包名而不是路径作为开始目录，搜索时会假定它导入的是你想要的目录，所以你不会收到警告。

测试模块和包可以通过*load_tests protocol* 自定义测试的加载和搜索。

3.4 版更變: 测试发现支持初始目录下的*命名空间包*。注意你也需要指定顶层目录（例如: `python -m unittest discover -s root/namespace -t root`）。

26.8.4 组织你的测试代码

单元测试的构建单位是 *test cases*： 独立的、包含执行条件与正确性检查的方案。在*unittest* 中，测试用例表示为*unittest.TestCase* 的实例。通过编写*TestCase* 的子类或使用*FunctionTestCase* 编写你自己的测试用例。

一个*TestCase* 实例的测试代码必须是完全自含的，因此它可以独立运行，或与其它任意组合任意数量的测试用例一起运行。

TestCase 的最简单的子类需要实现一个测试方法（例如一个命名以 `test` 开头的方法）以执行特定的测试代码:

```
import unittest

class DefaultWidgetSizeTestCase(unittest.TestCase):
    def test_default_widget_size(self):
        widget = Widget('The widget')
        self.assertEqual(widget.size(), (50, 50))
```

可以看到，为了进行测试，我们使用了基类*TestCase* 提供的其中一个 `assert*()` 方法。若测试不通过，将会引发一个带有说明信息的异常，并且*unittest* 会将这个测试用例标记为测试不通过。任何其它类型的异常将会被当做错误处理。

可能同时存在多个前置操作相同的测试，我们可以把测试的前置操作从测试代码中拆解出来，并实现测试前置方法*setUp()* 。在运行测试时，测试框架会自动地为每个单独测试调用前置方法。

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

備🄴: 多个测试运行的顺序由内置字符串排序方法对测试名进行排序的结果决定。

在测试运行时，若*setUp()* 方法引发异常，测试框架会认为测试发生了错误，因此测试方法不会被运行。

相似的，我们提供了一个*tearDown()* 方法在测试方法运行后进行清理工作。

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

若*setUp()* 成功运行，无论测试方法是否成功，都会运行*tearDown()* 。

这样的一个测试代码运行的环境被称为 *test fixture* 。一个新的 TestCase 实例作为一个测试脚手架，用于运行各个独立的测试方法。在运行每个测试时，*setUp()* 、*tearDown()* 和 __init__() 会被调用一次。

推荐你根据用例所测试的功能将测试用 TestCase 分组。*unittest* 为此提供了 *test suite*：*unittest* 的*TestSuite* 类是一个代表。通常情况下，调用*unittest.main()* 就能正确地找到并执行这个模块下所有用 TestCase 分组的测试。

然而，如果你需要自定义你的测试套件的话，你可以参考以下方法组织你的测试：

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_widget_size'))
    suite.addTest(WidgetTestCase('test_widget_resize'))
    return suite

if __name__ == '__main__':
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

你可以把测试用例和测试套件放在与被测试代码相同的模块中（比如 widget.py），但将测试代码放在单独的模块中（比如 test_widget.py）有几个优势。

- 测试模块可以从命令行被独立调用。
- 更容易在分发的代码中剥离测试代码。
- 降低没有好理由的情况下修改测试代码以通过测试的诱惑。
- 测试代码应比被测试代码更少地被修改。
- 被测试代码可以更容易地被重构。
- 对用 C 语言写成的模块无论如何都得单独写成一个模块，为什么不保持一致呢?
- 如果测试策略发生了改变，没有必要修改源代码。

26.8.5 复用已有的测试代码

一些用户希望直接使用*unittest* 运行已有的测试代码，而不需要把已有的每个测试函数转化为一个*TestCase* 的子类。

因此，*unittest* 提供*FunctionTestCase* 类。这个*TestCase* 的子类可用于打包已有的测试函数，并支持设置前置与后置函数。

假定有一个测试函数：

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
    # ...
```

可以创建等价的测试用例如下，其中前置和后置方法是可选的。

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB,
                                     tearDown=deleteSomethingDB)
```

備🄴: 用*FunctionTestCase* 可以快速将现有的测试转换成基于*unittest* 的测试，但不推荐你这样做。花点时间继承*TestCase* 会让以后重构测试无比轻松。

在某些情况下，现有的测试可能是用*doctest* 模块编写的。如果是这样，*doctest* 提供了一个 DocTestSuite 类，可以从现有基于*doctest* 的测试中自动构建*unittest.TestSuite* 用例。

26.8.6 跳过测试与预计的失败

3.1 版新加入.

Unittest 支持跳过单个或整组的测试用例。它还支持把测试标注成“预期失败”的测试，这些坏测试会失败，但不会算进*TestResult* 的失败里。

要跳过测试只需使用*skip() decorator* 或其附带条件的版本，在*setUp()* 内部使用*TestCase.skipTest()*，或是直接引发*SkipTest*。

跳过测试的基本用法如下:

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available")
        # test code that depends on the external resource
        pass
```

在🄵嗦模式下运行以上测试例子时，程序输出如下:

```
test_format (__main__.MyTestCase) ... skipped 'not supported in this library version'
test_nothing (__main__.MyTestCase) ... skipped 'demonstrating skipping'
test_maybe_skipped (__main__.MyTestCase) ... skipped 'external resource not available'
test_windows_support (__main__.MyTestCase) ... skipped 'requires Windows'

----------------------------------------------------------------------
Ran 4 tests in 0.005s

OK (skipped=4)
```

跳过测试类的写法跟跳过测试方法的写法相似:

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

TestCase.setUp() 也可以跳过测试。可以用于所需资源不可用的情况下跳过接下来的测试。

使用*expectedFailure()* 装饰器表明这个测试预计失败。:

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

你可以很容易地编写在测试时调用*skip()* 的装饰器作为自定义的跳过测试装饰器。下面这个装饰器会跳过测试，除非所传入的对象具有特定的属性:

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(obj, attr))
```

以下的装饰器和异常实现了跳过测试和预期失败两种功能:

`@unittest.`**`skip`**`(`*reason*`)`
: 跳过被此装饰器装饰的测试。*reason* 为测试被跳过的原因。

`@unittest.`**`skipIf`**`(`*condition*, *reason*`)`
: 当 *condition* 为真时，跳过被装饰的测试。

`@unittest.`**`skipUnless`**`(`*condition*, *reason*`)`
: 跳过被装饰的测试，除非 *condition* 为真。

`@unittest.`**`expectedFailure`**
: 将测试标记为预期的失败或错误。如果测试失败或在测试函数自身（而非在某个 *test fixture* 方法）中出现错误则将认为是测试成功。如果测试通过，则将认为是测试失败。

`exception` `unittest.`**`SkipTest`**`(`*reason*`)`
: 引发此异常以跳过一个测试。

  通常来说，你可以使用*TestCase.skipTest()* 或其中一个跳过测试的装饰器实现跳过测试的功能，而不是直接引发此异常。

被跳过的测试的*setUp()* 和*tearDown()* 不会被运行。被跳过的类的*setUpClass()* 和*tearDownClass()* 不会被运行。被跳过的模组的 `setUpModule()` 和 `tearDownModule()` 不会被运行。

26.8.7 使用子测试区分测试迭代

3.4 版新加入.

当你的几个测试之间的差异非常小，例如只有某些形参不同时，unittest 允许你使用*subTest()* 上下文管理器在一个测试方法体的内部区分它们。

例如，以下测试:

```
class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

可以得到以下输出:

```
======================================================================
FAIL: test_even (__main__.NumbersTest) (i=1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=3)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=5)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

如果不使用子测试，程序遇到第一次错误之后就会停止。而且因为 “i“的值不显示，错误也更难找。

```
======================================================================
FAIL: test_even (__main__.NumbersTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

26.8.8 类与函数

本节深入介绍了*unittest* 的 API。

测试用例

class unittest.**TestCase**(*methodName='runTest'*)

TestCase 类的实例代表了*unittest* 宇宙中的逻辑测试单元。该类旨在被当作基类使用，特定的测试将由其实体子类来实现。该类实现了测试运行器所需的接口以允许它驱动测试，并实现了可被测试代码用来检测和报告各种类型的失败的方法。

每个*TestCase* 实例将运行一个单位的基础方法：即名为 *methodName* 的方法。在使用*TestCase* 的大多数场景中，你都不需要修改 *methodName* 或重新实现默认的 `runTest()` 方法。

3.2 版更變: *TestCase* 不需要提供 *methodName* 即可成功初始化。这使得从交互式解释器试验*TestCase* 更为容易。

TestCase 的实例提供了三组方法：一组用来运行测试，另一组被测试实现用来检查条件和报告失败，还有一些查询方法用来收集有关测试本身的信息。

第一组（用于运行测试的）方法是：

setUp()

为测试预备而调用的方法。此方法会在调用测试方法之前被调用；除了*AssertionError* 或*SkipTest*，此方法所引发的任何异常都将被视为错误而非测试失败。默认的实现将不做任何事情。

tearDown()

在测试方法被调用并记录结果之后立即被调用的方法。此方法即使在测试方法引发异常时仍会被调用，因此子类中的实现将需要特别注意检查内部状态。除*AssertionError* 或*SkipTest* 外，此方法所引发的任何异常都将被视为额外的错误而非测试失败（因而会增加总计错误报告数）。此方法将只在*setUp()* 成功执行时被调用，无论测试方法的结果如何。默认的实现将不做任何事情。

setUpClass()

在一个单独类中的测试运行之前被调用的类方法。`setUpClass` 会被作为唯一的参数在类上调用且必须使用*classmethod()* 装饰器:

```
@classmethod
def setUpClass(cls):
    ...
```

查看*Class and Module Fixtures* 获取更详细的说明。

3.2 版新加入.

tearDownClass()

在一个单独类的测试完成运行之后被调用的类方法。`tearDownClass` 会被作为唯一的参数在类上调用且必须使用*classmethod()* 装饰器:

```
@classmethod
def tearDownClass(cls):
    ...
```

查看*Class and Module Fixtures* 获取更详细的说明。

3.2 版新加入.

run(*result=None*)
运行测试，将结果收集至作为 *result* 传入的*TestResult*。如果 *result* 被省略或为 None，则会创建一个临时的结果对象（通过调用*defaultTestResult()* 方法）并使用它。结果对象会被返回给*run()* 的调用方。

同样的效果也可通过简单地调用*TestCase* 实例来达成。

3.3 版更變: 之前版本的 run 不会返回结果。也不会调用实例。

skipTest(*reason*)
在测试方法或*setUp()* 执行期间调用此方法将跳过当前测试。详情参见跳过测试与预计的失败。

3.1 版新加入.

subTest(*msg=None*, ***params*)
返回一个上下文管理器以将其中的代码块作为子测试来执行。可选的 *msg* 和 *params* 是将在子测试失败时显示的任意值，以便让你能清楚地标识它们。

一个测试用例可以包含任意数量的子测试声明，并且它们可以任意地嵌套。

查看使用子测试区分测试迭代 获取更详细的信息。

3.4 版新加入.

debug()
运行测试而不收集结果。这允许测试所引发的异常被传递给调用方，并可被用于支持在调试器中运行测试。

TestCase 类提供了一些断言方法用于检查并报告失败。下表列出了最常用的方法（请查看下文的其他表来了解更多的断言方法）：

方法	检查对象	引入版本
assertEqual(a, b)	`a == b`	
assertNotEqual(a, b)	`a != b`	
assertTrue(x)	`bool(x) is True`	
assertFalse(x)	`bool(x) is False`	
assertIs(a, b)	`a is b`	3.1
assertIsNot(a, b)	`a is not b`	3.1
assertIsNone(x)	`x is None`	3.1
assertIsNotNone(x)	`x is not None`	3.1
assertIn(a, b)	`a in b`	3.1
assertNotIn(a, b)	`a not in b`	3.1
assertIsInstance(a, b)	`isinstance(a, b)`	3.2
assertNotIsInstance(a, b)	`not isinstance(a, b)`	3.2

这些断言方法都支持 *msg* 参数，如果指定了该参数，它将被用作测试失败时的错误消息 (另请参阅*longMessage*)。请注意将 *msg* 关键字参数传给*assertRaises()*, *assertRaisesRegex()*, *assertWarns()*, *assertWarnsRegex()* 的前提是它们必须被用作上下文管理器。

assertEqual(*first*, *second*, *msg=None*)
测试 *first* 和 *second* 是否相等。如果两个值的比较结果是不相等，则测试将失败。

此外，如果 *first* 和 *second* 的类型完全相同且属于 list, tuple, dict, set, frozenset 或 str 或者属于通过*addTypeEqualityFunc()* 注册子类的类型则将会调用类型专属的相等判断函数以便生成更有用的默认错误消息 (另请参阅类型专属方法列表)。

3.1 版更變: 增加了对类型专属的相等判断函数的自动调用。

3.2 版更變: 增加了*assertMultiLineEqual()* 作为用于比较字符串的默认类型相等判断函数。

assertNotEqual(*first*, *second*, *msg=None*)
测试 *first* 和 *second* 是否不等。如果两个值的比较结果是相等，则测试将失败。

assertTrue(*expr*, *msg=None*)
assertFalse(*expr*, *msg=None*)
测试 *expr* 是否为真值（或假值）。

请注意这等价于 `bool(expr) is True` 而不等价于 `expr is True` (后者要使用 `assertIs(expr, True)`)。当存在更专门的方法时也应避免使用此方法 (例如应使用 `assertEqual(a, b)` 而不是 `assertTrue(a == b)`)，因为它们在测试失败时会提供更有用的错误消息。

assertIs(*first*, *second*, *msg=None*)
assertIsNot(*first*, *second*, *msg=None*)
测试 *first* 和 *second* 是（或不是）同一个对象。

3.1 版新加入.

assertIsNone(*expr*, *msg=None*)
assertIsNotNone(*expr*, *msg=None*)
测试 *expr* 是 (或不是) `None`。

3.1 版新加入.

assertIn(*member*, *container*, *msg=None*)
assertNotIn(*member*, *container*, *msg=None*)
测试 *member* 是 (或不是) *container* 的成员。

3.1 版新加入.

assertIsInstance(*obj*, *cls*, *msg=None*)
assertNotIsInstance(*obj*, *cls*, *msg=None*)
测试 *obj* 是 (或不是) *cls* (此参数可以为一个类或包含类的元组，即*isinstance()* 所接受的参数) 的实例。要检测是否为指定类型，请使用*assertIs(type(obj), cls)*。

3.2 版新加入.

还可以使用下列方法来检查异常、警告和日志消息的产生:

方法	检查对象	引入版本
*assertRaises(exc, fun, *args, **kwds)*	`fun(*args, **kwds)` 引发 *exc*	
*assertRaisesRegex(exc, r, fun, *args, **kwds)*	`fun(*args, **kwds)` 引发了 *exc* 并且消息可与正则表达式 *r* 相匹配	3.1
*assertWarns(warn, fun, *args, **kwds)*	`fun(*args, **kwds)` 引发了 *warn*	3.2
*assertWarnsRegex(warn, r, fun, *args, **kwds)*	`fun(*args, **kwds)` 引发了 *warn* 并且消息可与正则表达式 *r* 相匹配	3.2
assertLogs(logger, level)	`with` 代码块在 *logger* 上使用最小的 *level* 级别写入了日志	3.4

assertRaises(*exception*, *callable*, **args*, ***kwds*)
assertRaises(*exception*, *, *msg=None*)
测试当 *callable* 附带任何同时被传给*assertRaises()* 的位置或关键字参数被调用时是否引发了异常。如果引发了 *exception* 则测试通过，如果引发了另一个异常则报错，或者如果未引发任何异常则测试失败。要捕获一组异常中的任何一个，可以将包含多个异常类的元组作为 *exception* 传入。

如果只给出了 *exception* 和可能的 *msg* 参数，则返回一个上下文管理器以便被测试的代码可以被写成内联形式而不是被写成函数:

```
with self.assertRaises(SomeException):
    do_something()
```

当被作为上下文管理器使用时，*assertRaises()* 接受额外的关键字参数 *msg*。

上下文管理器将把捕获的异常对象存入在其 `exception` 属性中。这适用于需要对所引发异常执行额外检查的场合:

```
with self.assertRaises(SomeException) as cm:
    do_something()

the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

3.1 版更變: 添加了将*assertRaises()* 用作上下文管理器的功能。

3.2 版更變: 增加了 `exception` 属性。

3.3 版更變: 增加了 *msg* 关键字参数在作为上下文管理器时使用。

assertRaisesRegex(*exception*, *regex*, *callable*, **args*, ***kwds*)
assertRaisesRegex(*exception*, *regex*, *, *msg=None*)

与*assertRaises()* 类似但还会测试 *regex* 是否匹配被引发异常的字符串表示形式。*regex* 可以是一个正则表达式对象或包含正则表达式的字符串以提供给*re.search()* 使用。例如:

```
self.assertRaisesRegex(ValueError, "invalid literal for.*XYZ'$",
                       int, 'XYZ')
```

或是:

```
with self.assertRaisesRegex(ValueError, 'literal'):
   int('XYZ')
```

3.1 版新加入: 以方法名 “assertRaisesRegexp“添加。

3.2 版更變: 重命名为*assertRaisesRegex()*。

3.3 版更變: 增加了 *msg* 关键字参数在作为上下文管理器时使用。

assertWarns(*warning*, *callable*, **args*, ***kwds*)
assertWarns(*warning*, *, *msg=None*)

测试当 *callable* 附带任何同时被传给*assertWarns()* 的位置或关键字参数被调用时是否触发了警告。如果触发了 *warning* 则测试通过，否则测试失败。引发任何异常则报错。要捕获一组警告中的任何一个，可将包含多个警告类的元组作为 *warnings* 传入。

如果只给出了 *warning* 和可能的 *msg* 参数，则返回一个上下文管理器以便被测试的代码可以被写成内联形式而不是被写成函数:

```
with self.assertWarns(SomeWarning):
    do_something()
```

当被作为上下文管理器使用时，*assertWarns()* 接受额外的关键字参数 *msg*。

上下文管理器将把捕获的警告对象保存在其 `warning` 属性中，并把触发警告的源代码行保存在 `filename` 和 `lineno` 属性中。这适用于需要对捕获的警告执行额外检查的场合:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()

self.assertIn('myfile.py', cm.filename)
self.assertEqual(320, cm.lineno)
```

无论被调用时警告过滤器是否就位此方法均可工作。

3.2 版新加入.

3.3 版更變: 增加了 *msg* 关键字参数在作为上下文管理器时使用。

assertWarnsRegex(*warning*, *regex*, *callable*, **args*, ***kwds*)

assertWarnsRegex(*warning*, *regex*, ***, *msg=None*)

与*assertWarns()* 类似但还会测试 *regex* 是否匹配被触发警告的消息文本。*regex* 可以是一个正则表达式对象或包含正则表达式的字符串以提供给*re.search()* 使用。例如:

```
self.assertWarnsRegex(DeprecationWarning,
                      r'legacy_function\(\) is deprecated',
                      legacy_function, 'XYZ')
```

或是:

```
with self.assertWarnsRegex(RuntimeWarning, 'unsafe frobnicating'):
    frobnicate('/etc/passwd')
```

3.2 版新加入.

3.3 版更變: 增加了 *msg* 关键字参数在作为上下文管理器时使用。

assertLogs(*logger=None*, *level=None*)

一个上下文管理器，它测试在 *logger* 或其子对象上是否至少记录了一条至少为指定 *level* 以上级别的消息。

如果给出了 *logger* 则它应为一个*logging.Logger* 对象或为一个指定日志记录器名称的*str*。默认为根日志记录器，它将捕获未被非传播型后继日志记录器所拦阻的所有消息。

如果给出了 *level* 则它应为一个用数字表示的日志记录级别或其字符串形式 (例如 `"ERROR"` 或 `logging.ERROR`)。默认为 `logging.INFO`。

如果在 `with` 代码块内部发出了至少一条与 *logger* 和 *level* 条件相匹配的消息则测试通过，否则测试失败。

上下文管理器返回的对象是一个记录辅助器，它会记录所匹配的日志消息。它有两个属性:

records
: 所匹配的日志消息*logging.LogRecord* 对象组成的列表。

output
: 由*str* 对象组成的列表，内容为所匹配消息经格式化后的输出。

示例:

```
with self.assertLogs('foo', level='INFO') as cm:
   logging.getLogger('foo').info('first message')
   logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

3.4 版新加入.

还有其他一些方法可用于执行更专门的检查，例如:

方法	检查对象	引入版本
assertAlmostEqual(a, b)	`round(a-b, 7) == 0`	
assertNotAlmostEqual(a, b)	`round(a-b, 7) != 0`	
assertGreater(a, b)	`a > b`	3.1
assertGreaterEqual(a, b)	`a >= b`	3.1
assertLess(a, b)	`a < b`	3.1
assertLessEqual(a, b)	`a <= b`	3.1
assertRegex(s, r)	`r.search(s)`	3.1
assertNotRegex(s, r)	`not r.search(s)`	3.2
assertCountEqual(a, b)	*a* 和 *b* 具有同样数量的相同元素，无论其顺序如何。	3.2

assertAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)
assertNotAlmostEqual(*first*, *second*, *places=7*, *msg=None*, *delta=None*)
测试 *first* 是否约等于 *second*，比较的标准是计算差值并舍入到 *places* 所指定的十进制位数 (默认为 7 位)，再与零相比较。请注意此方法是将结果值舍入到指定的 十进制位数 (即相当于*round()* 函数) 而非 有效位数。

如果提供了 *delta* 而非 *places* 则 *first* 和 *second* 之间的差值必须小于等于 (或大于) *delta*。

同时提供 *delta* 和 *places* 将引发*TypeError*。

3.2 版更變: *assertAlmostEqual()* 会自动将几乎相等的对象视为相等。而如果对象相等则*assertNotAlmostEqual()* 会自动测试失败。增加了 *delta* 关键字参数。

assertGreater(*first*, *second*, *msg=None*)
assertGreaterEqual(*first*, *second*, *msg=None*)
assertLess(*first*, *second*, *msg=None*)
assertLessEqual(*first*, *second*, *msg=None*)
根据方法名分别测试 *first* 是否 >, >=, < 或 <= *second*。如果不是，则测试失败:

```
>>> self.assertGreaterEqual(3, 4)
AssertionError: "3" unexpectedly not greater than or equal to "4"
```

3.1 版新加入.

assertRegex(*text*, *regex*, *msg=None*)
assertNotRegex(*text*, *regex*, *msg=None*)
测试一个 *regex* 是否匹配 文本。如果不匹配，错误信息中将包含匹配模式和 文本 * (或部分匹配失败的 * 文本)。*regex* 可以是正则表达式对象或能够用于*re.search()* 的包含正则表达式的字符串。

3.1 版新加入: 以方法名 “assertRegexpMatches“添加。

3.2 版更變: 方法 `assertRegexpMatches()` 已被改名为*assertRegex()*。

3.2 版新加入: *assertNotRegex()*

3.5 版新加入: `assertNotRegexpMatches` 这个名字是*assertNotRegex()* 的已被弃用的别名。

assertCountEqual(*first*, *second*, *msg=None*)
测试序列 *first* 与 *second* 是否包含同样的元素，无论其顺序如何。当存在差异时，将生成一条错误消息来列出两个序列之间的差异。

重复的元素 不会在 *first* 和 *second* 的比较中被忽略。它会检查每个元素在两个序列中的出现次数是否相同。等价于: `assertEqual(Counter(list(first)), Counter(list(second)))` 但还适用于包含不可哈希对象的序列。

3.2 版新加入.

assertEqual() 方法会将相同类型对象的相等性检查分派给不同的类型专属方法。这些方法已被大多数内置类型所实现，但也可以使用*addTypeEqualityFunc()* 来注册新的方法:

addTypeEqualityFunc(*typeobj*, *function*)

注册一个由*assertEqual()* 调用的特定类型专属方法来检查恰好为相同 *typeobj* (而非子类) 的两个对象是否相等。*function* 必须接受两个位置参数和第三个 msg=None 关键字参数，就像*assertEqual()* 那样。当检测到前两个形参之间不相等时它必须引发*self.failureException(msg)* -- 可能还会提供有用的信息并在错误消息中详细解释不相等的原因。

3.1 版新加入.

以下是*assertEqual()* 自动选用的不同类型的比较方法。一般情况下不需要直接在测试中调用这些方法。

方法	用作比较	引入版本
assertMultiLineEqual(a, b)	字符串	3.1
assertSequenceEqual(a, b)	序列	3.1
assertListEqual(a, b)	列表	3.1
assertTupleEqual(a, b)	元组	3.1
assertSetEqual(a, b)	集合	3.1
assertDictEqual(a, b)	字典	3.1

assertMultiLineEqual(*first*, *second*, *msg=None*)

测试多行字符串 *first* 是否与字符串 *second* 相等。当不相等时将在错误消息中包括两个字符串之间差异的高亮显示。此方法会在通过*assertEqual()* 进行字符串比较时默认被使用。

3.1 版新加入.

assertSequenceEqual(*first*, *second*, *msg=None*, *seq_type=None*)

测试两个序列是否相等。如果提供了 *seq_type*，则 *first* 和 *second* 都必须为 *seq_type* 的实例否则将引发失败。如果两个序列不相等则会构造一个错误消息来显示两者之间的差异。

此方法不会被*assertEqual()* 直接调用，但它会被用于实现*assertListEqual()* 和*assertTupleEqual()*。

3.1 版新加入.

assertListEqual(*first*, *second*, *msg=None*)

assertTupleEqual(*first*, *second*, *msg=None*)

测试两个列表或元组是否相等。如果不相等，则会构造一个错误消息来显示两者之间的差异。如果某个形参的类型不正确也会引发错误。这些方法会在通过*assertEqual()* 进行列表或元组比较时默认被使用。

3.1 版新加入.

assertSetEqual(*first*, *second*, *msg=None*)

测试两个集合是否相等。如果不相等，则会构造一个错误消息来列出两者之间的差异。此方法会在通过*assertEqual()* 进行集合或冻结集合比较时默认被使用。

如果 *first* 或 *second* 没有 `set.difference()` 方法则测试失败。

3.1 版新加入.

assertDictEqual(*first*, *second*, *msg=None*)
测试两个字典是否相等。如果不相等，则会构造一个错误消息来显示两个字典的差异。此方法会在对*assertEqual()* 的调用中默认被用来进行字典的比较。

3.1 版新加入.

最后*TestCase* 还提供了以下的方法和属性:

fail(*msg=None*)
无条件地发出测试失败消息，附带错误消息 *msg* 或 None。

failureException
这个类属性给出测试方法所引发的异常。如果某个测试框架需要使用专门的异常，并可能附带额外的信息，则必须子类化该类以便与框架“正常互动”。这个属性的初始值为*AssertionError*。

longMessage
这个类属性决定当将一个自定义失败消息作为 msg 参数传给一个失败的 assertXYY 调用时会发生什么。默认值为 True。在此情况下，自定义消息会被添加到标准失败消息的末尾。当设为 False 时，自定义消息会替换标准消息。

类设置可以通过在调用断言方法之前将一个实例属性 self.longMessage 赋值为 True 或 False 在单个测试方法中进行重载。

类设置会在每个测试调用之前被重置。

3.1 版新加入.

maxDiff
这个属性控制来自在测试失败时报告 diffs 的断言方法的 diffs 输出的最大长度。它默认为 80*8 个字符。这个属性所影响的断言方法有*assertSequenceEqual()* (包括所有委托给它的序列比较方法), *assertDictEqual()* 以及*assertMultiLineEqual()*。

将 maxDiff 设为 None 表示不限制 diffs 的最大长度。

3.2 版新加入.

测试框架可使用下列方法在测试时收集信息:

countTestCases()
返回此测试对象所提供的测试数量。对于*TestCase* 实例，该数量将总是为 1。

defaultTestResult()
返回此测试类所要使用的测试结果类的实例（如果未向*run()* 方法提供其他结果实例）。

对于*TestCase* 实例，该返回值将总是为*TestResult* 的实例；*TestCase* 的子类应当在有必要时重载此方法。

id()
返回一个标识指定测试用例的字符串。该返回值通常为测试方法的完整名称，包括模块名和类名。

shortDescription()
返回测试的描述，如果未提供描述则返回 None。此方法的默认实现将在可用的情况下返回测试方法的文档字符串的第一行，或者返回 None。

3.1 版更變: 在 3.1 中已修改此方法将测试名称添加到简短描述中，即使存在文档字符串。这导致了与单元测试扩展的兼容性问题因而在 Python 3.2 中将添加测试名称操作改到*TextTestResult* 中。

addCleanup(*function*, */*, **args*, ***kwargs*)
在*tearDown()* 之后添加了一个要调用的函数来清理测试期间所使用的资源。函数将按它们被添加的相反顺序被调用 (LIFO)。它们在调用时将附带它们被添加时传给*addCleanup()* 的任何参数和关键字参数。

如果*setUp()* 失败，即意味着*tearDown()* 未被调用，则已添加的任何清理函数仍将被调用。

3.1 版新加入.

doCleanups()
此方法会在*tearDown()* 之后无条件地被调用，或者如果*setUp()* 引发了异常则会在*setUp()* 之后被调用。

它将负责调用由*addCleanup()* 添加的所有清理函数。如果你需要在*tearDown()* 之前调用清理函数则可以自行调用*doCleanups()*。

doCleanups() 每次会弹出清理函数栈中的一个方法，因此它可以在任何时候被调用。

3.1 版新加入.

classmethod addClassCleanup(*function*, */*, **args*, ***kwargs*)
在 Add a function to be called after *tearDownClass()* 之后添加了一个要调用的函数来清理测试类运行期间所使用的资源。函数将按它们被添加的相反顺序被调用 (LIFO)。它们在调用时将附带它们被添加时传给*addClassCleanup()* 的任何参数和关键字参数。

如果*setUpClass()* 失败，即意味着*tearDownClass()* 未被调用，则已添加的任何清理函数仍将被调用。

3.8 版新加入.

classmethod doClassCleanups()
此方法会在*tearDownClass()* 之后无条件地被调用，或者如果*setUpClass()* 引发了异常则会在*setUpClass()* 之后被调用。

它将负责访问由*addClassCleanup()* 添加的所有清理函数。如果你需要在*tearDownClass()* 之前调用清理函数则可以自行调用*doClassCleanups()*。

doClassCleanups() 每次会弹出清理函数栈中的一个方法，因此它在任何时候被调用。

3.8 版新加入.

class unittest.**IsolatedAsyncioTestCase**(*methodName='runTest'*)
这个类提供了与*TestCase* 类似的 API 并也接受协程作为测试函数。

3.8 版新加入.

coroutine asyncSetUp()
为测试预备而调用的方法。此方法会在 `setUp()` 之后被调用。此方法将在调用测试方法之前立即被调用；除了*AssertionError* 或*SkipTest*，此方法所引发的任何异常都将被视为错误而非测试失败。默认的实现将不做任何事情。

coroutine asyncTearDown()
在测试方法被调用并记录结果之后立即被调用的方法。此方法会在 `tearDown()` 之前被调用。此方法即使在测试方法引发异常时仍会被调用，因此子类中的实现将需要特别注意检查内部状态。除*AssertionError* 或*SkipTest* 外，此方法所引发的任何异常都将被视为额外的错误而非测试失败（因而会增加总计错误报告数）。此方法将只在*asyncSetUp()* 成功执行时被调用，无论测试方法的结果如何。默认的实现将不做任何事情。

addAsyncCleanup(*function*, */*, **args*, ***kwargs*)
此方法接受一个可被用作清理函数的协程。

run(*result=None*)
设置一个新的事件循环来运行测试，将结果收集至作为 *result* 传入的*TestResult*。如果 *result* 被省略或为 `None`，则会创建一个临时的结果对象（通过调用 `defaultTestResult()` 方法）并使用它。结果对象会被返回给*run()* 的调用方。在测试结束时事件循环中的所有任务都将被取消。

一个显示先后顺序的例子:

```
from unittest import IsolatedAsyncioTestCase

events = []


class Test(IsolatedAsyncioTestCase):


    def setUp(self):
        events.append("setUp")

    async def asyncSetUp(self):
        self._async_connection = await AsyncConnection()
        events.append("asyncSetUp")

    async def test_response(self):
        events.append("test_response")
        response = await self._async_connection.get("https://example.com")
        self.assertEqual(response.status_code, 200)
        self.addAsyncCleanup(self.on_cleanup)

    def tearDown(self):
        events.append("tearDown")

    async def asyncTearDown(self):
        await self._async_connection.close()
        events.append("asyncTearDown")

    async def on_cleanup(self):
        events.append("cleanup")

if __name__ == "__main__":
    unittest.main()
```

在运行测试之后，events 将会包含 ["setUp", "asyncSetUp", "test_response", "asyncTearDown", "tearDown", "cleanup"]。

class unittest.**FunctionTestCase**(*testFunc*, *setUp=None*, *tearDown=None*, *description=None*)

这个类实现了*TestCase*的部分接口，允许测试运行方驱动测试，但不提供可被测试代码用来检查和报告错误的方法。这个类被用于创建使用传统测试代码的测试用例，允许它被集成到基于*unittest*的测试框架中。

已弃用的别名

由于历史原因，某些*TestCase*方法具有一个或几个已目前已弃用的别名。下表列出了它们的正确名称和已弃用的别名:

方法名	已弃用的别名	已弃用的别名
assertEqual()	failUnlessEqual	assertEquals
assertNotEqual()	failIfEqual	assertNotEquals
assertTrue()	failUnless	assert_
assertFalse()	failIf	
assertRaises()	failUnlessRaises	
assertAlmostEqual()	failUnlessAlmostEqual	assertAlmostEquals
assertNotAlmostEqual()	failIfAlmostEqual	assertNotAlmostEquals
assertRegex()		assertRegexpMatches
assertNotRegex()		assertNotRegexpMatches
assertRaisesRegex()		assertRaisesRegexp

3.1 版後已🄵用: 在第二列中列出的 fail* 别名已经被弃用。

3.2 版後已🄵用: 在第三列中列出的 assert* 别名已经被弃用。

3.2 版後已🄵用: `assertRegexpMatches` 和 `assertRaisesRegexp` 已经被重命名为*assertRegex()* 和*assertRaisesRegex()*。

3.5 版後已🄵用: `assertNotRegexpMatches` 这个名称已被弃用并应改用*assertNotRegex()*。

分组测试

class unittest.**TestSuite**(*tests=()*)

这个类代表对单独测试用例和测试套件的聚合。这个类提供给测试运行方所需的接口以允许其像任何其他测试用例一样运行。运行一个*TestSuite* 实例与对套件执行迭代来逐一运行每个测试的效果相同。

如果给出了 *tests*，则它必须是一个包含单独测试用例的可迭代对象或是将被用于初始构建测试套件的其他测试套件。额外的方法被提供用来在随后向测试集添加测试用例和测试套件。

TestSuite 对象的行为与*TestCase* 对象很相似，区别在于它们并不会真正实现一个测试。它们会被用来将测试聚合为多个要同时运行的测试分组。还有一些额外的方法会被用来向*TestSuite* 实例添加测试:

addTest(*test*)

向测试套件添加*TestCase* 或*TestSuite*。

addTests(*tests*)

将来自包含*TestCase* 和*TestSuite* 实例的可迭代对象的所有测试添加到这个测试套件。

这等价于对 *tests* 进行迭代，并为其中的每个元素调用*addTest()*。

TestSuite 与*TestCase* 共享下列方法:

run(*result*)

运行与这个套件相关联的测试，将结果收集到作为 *result* 传入的测试结果对象中。请注意与*TestCase.run()* 的区别，*TestSuite.run()* 必须传入结果对象。

debug()

运行与这个套件相关联的测试而不收集结果。这允许测试所引发的异常被传递给调用方并可被用于支持在调试器中运行测试。

countTestCases()

返回此测试对象所提供的测试数量，包括单独的测试和子套件。

__iter__()
由*TestSuite* 分组的测试总是可以通过迭代来访问。其子类可以通过重载*__iter__()* 来惰性地提供测试。请注意此方法可在单个套件上多次被调用（例如在计数测试或相等性比较时），为此在*TestSuite.run()* 之前重复迭代所返回的测试对于每次调用迭代都必须相同。在*TestSuite.run()* 之后，调用方不应继续访问此方法所返回的测试，除非调用方使用重载了 TestSuite._removeTestAtIndex() 的子类来保留对测试的引用。

3.2 版更變: 在较早的版本中*TestSuite* 会直接访问测试而不是通过迭代，因此只重载*__iter__()* 并不足以提供所有测试。

3.4 版更變: 在较早的版本中*TestSuite* 会在*TestSuite.run()* 之后保留对每个*TestCase* 的引用。其子类可以通过重载 TestSuite._removeTestAtIndex() 来恢复此行为。

在*TestSuite* 对象的典型应用中，*run()* 方法是由 TestRunner 发起调用而不是由最终用户测试来控制。

加载和运行测试

class unittest.**TestLoader**
TestLoader 类可被用来基于类和模块创建测试套件。通常，没有必要创建该类的实例；*unittest* 模块提供了一个可作为*unittest.defaultTestLoader* 共享的实例。但是，使用子类或实例允许对某些配置属性进行定制。

TestLoader 对象具有下列属性:

errors
包含在加载测试期间遇到的非致命错误的列表。在任何时候都不会被加载方重置。致命错误是通过相关方法引发一个异常来向调用方发出信号的。非致命错误也是由一个将在运行时引发原始错误的合成测试来提示的。

3.5 版新加入.

TestLoader 对象具有下列方法:

loadTestsFromTestCase(*testCaseClass*)
返回一个包含在*TestCase* 所派生的 testCaseClass 中的所有测试用例的测试套件。

会为每个由*getTestCaseNames()* 指明的方法创建一个测试用例实例。在默认情况下这些都是以 test 开头的方法名称。如果*getTestCaseNames()* 不返回任何方法，但 runTest() 方法已被实现，则会为该方法创建一个单独的测试用例。

loadTestsFromModule(*module*, *pattern=None*)
返回包含在给定模块中的所有测试用例的测试套件。此方法会在 *module* 中搜索从派生自*TestCase* 的类并为该类定义的每个测试方法创建一个类实例。

備𝔽: 虽然使用*TestCase* 所派生的类的层级结构可以方便地共享配置和辅助函数，但在不打算直接实例化的基类上定义测试方法并不能很好地配合此方法使用。不过，当配置有差异并且定义在子类当中时这样做还是有用处的。

如果一个模块提供了 load_tests 函数则它将被调用以加载测试。这允许模块自行定制测试加载过程。这就称为*load_tests protocol*。*pattern* 参数会被作为传给 load_tests 的第三个参数。

3.2 版更變: 添加了对 load_tests 的支持。

3.5 版更變: 未写入文档的非官方 *use_load_tests* 默认参数已被弃用并忽略，但是它仍然被接受以便向下兼容。此方法现在还接受一个仅限关键字参数 *pattern*，它会被作为传给 load_tests 的第三个参数。

loadTestsFromName(*name*, *module=None*)
返回由给出了字符串形式规格描述的所有测试用例组成的测试套件。

描述名称 *name* 是一个“带点号的名称”，它可以被解析为一个模块、一个测试用例类、一个测试用例类内部的测试方法、一个*TestSuite* 实例，或者一个返回*TestCase* 或*TestSuite* 实例的可调用对象。这些检查将按在此列出的顺序执行；也就是说，一个可能的测试用例类上的方法将作为“一个测试用例内部的测试方法”而非作为“一个可调用对象”被选定。

举例来说，如果你有一个模块 SampleTests，其中包含一个派生自*TestCase* 的类 SampleTestCase，其中包含三个测试方法 (test_one(), test_two() 和 test_three())。则描述名称 'SampleTests.SampleTestCase' 将使此方法返回一个测试套件，它将运行全部三个测试方法。使用描述名称 'SampleTests.SampleTestCase.test_two' 将使它返回一个测试套件，它将仅运行 test_two() 测试方法。描述名称可以指向尚未被导入的模块和包；它们将作为附带影响被导入。

本模块可以选择相对于给定的 *module* 来解析 *name*。

3.5 版更變: 如果在遍历 *name* 时发生了*ImportError* 或*AttributeError* 则在运行时引发该错误的合成测试将被返回。这些错误被包括在由 self.errors 所积累的错误中。

loadTestsFromNames(*names*, *module=None*)
类似于*loadTestsFromName()*，但是接受一个名称序列而不是单个名称。返回值是一个测试套件，它支持为每个名称所定义的所有测试。

getTestCaseNames(*testCaseClass*)
返回由 *testCaseClass* 中找到的方法名称组成的已排序的序列；这应当是*TestCase* 的一个子类。

discover(*start_dir*, *pattern='test*.py'*, *top_level_dir=None*)
通过从指定的开始目录向其子目录递归来找出所有测试模块，并返回一个包含该结果的 TestSuite 对象。只有与 *pattern* 匹配的测试文件才会被加载。（使用 shell 风格的模式匹配。）只有可导入的模块名称（即有效的 Python 标识符）将会被加载。

所有测试模块都必须可以从项目的最高层级上导入。如果起始目录不是最高层级目录则必须单独指明最高层级目录。

如果导入某个模块失败，比如因为存在语法错误，则会将其记录为单独的错误并将继续查找模块。如果导入失败是因为引发了*SkipTest*，则会将其记录为跳过而不是错误。

如果找到了一个包（即包含名为 __init__.py 的文件的目录），则将在包中查找 load_tests 函数。如果存在此函数则将对其执行调用 package.load_tests(loader, tests, pattern)。测试发现操作会确保在执行期间仅检查测试一次，即使 load_tests 函数本身调用了 loader.discover 也是如此。.

如果 load_tests 存在则发现操作 不会对包执行递归处理，load_tests 将负责加载包中的所有测试。is responsible for loading all tests in the package.

模式特意地不被当作 loader 属性来保存以使包能够自己继续执行发现操作。*top_level_dir* 则会被保存以使 load_tests 不需要将此参数传入到 loader.discover()。

start_dir 可以是一个带点号的名称或是一个目录。

3.2 版新加入.

3.4 版更變: 在导入时引发*SkipTest* 的模块会被记录为跳过，而不是错误。

3.4 版更變: *start_dir* 可以是一个命名空间包。

3.4 版更變: 路径在被导入之前会先被排序以使得执行顺序保持一致，即使下层文件系统的顺序不是取决于文件名的。

3.5 版更變: 现在 load_tests 会检查已找到的包，无论它们的路径是否与 *pattern* 匹配，因为包名称是无法与默认的模式匹配的。

TestLoader 的下列属性可通过子类化或在实例上赋值来配置:

testMethodPrefix
给出将被解读为测试方法的方法名称的前缀的字符串。默认值为 `'test'`。

这会影响*getTestCaseNames()* 以及所有 `loadTestsFrom*()` 方法。

sortTestMethodsUsing
将被用来在*getTestCaseNames()* 以及所有 `loadTestsFrom*()` 方法中比较方法名称以便对它们进行排序。

suiteClass
根据一个测试列表来构造测试套件的可调用对象。不需要结果对象上的任何方法。默认值为*TestSuite* 类。

这会影响所有 `loadTestsFrom*()` 方法。

testNamePatterns
由 Unix shell 风格通配符的测试名称模式组成的列表，供测试方法进行匹配以包括在测试套件中(参见 `-v` 选项)。

如果该属性不为 `None` (默认值)，则将要包括在测试套件中的所有测试方法都必须匹配该列表中的某个模式。请注意匹配总是使用*fnmatch.fnmatchcase()* 来执行，因此不同于传给 `-v` 选项的模式，简单的子字符串模式将必须使用 `*` 通配符来进行转换。

这会影响所有 `loadTestsFrom*()` 方法。

3.7 版新加入.

class unittest.**TestResult**
这个类被用于编译有关哪些测试执行成功而哪些失败的信息。

存放一组测试的结果的*TestResult* 对象。*TestCase* 和*TestSuite* 类将确保结果被正确地记录；测试创建者无须担心如何记录测试的结果。

建立在*unittest* 之上的测试框架可能会想要访问通过运行一组测试所产生的*TestResult* 对象用来报告信息；`TestRunner.run()` 方法是出于这个目的而返回*TestResult* 实例的。

TestResult 实例具有下列属性，在检查运行一组测试的结果的时候很有用处。

errors
一个包含*TestCase* 实例和保存了格式化回溯信息的字符串 2 元组的列表。每个元组代表一个引发了非预期的异常的测试。

failures
一个包含*TestCase* 实例和保存了格式化回溯信息的字符串 2 元组的列表。每个元组代表一个使用 `TestCase.assert*()` 方法显式地发出失败信号的测试。

skipped
一个包含 2-tuples of *TestCase* 实例和保存了跳过测试原因的字符串 2 元组的列表。

3.1 版新加入.

expectedFailures
一个包含*TestCase* 实例和保存了格式化回溯信息的 2 元组的列表。每个元组代表测试用例的一个已预期的失败或错误。

unexpectedSuccesses
一个包含被标记为已预期失败，但却测试成功的*TestCase* 实例的列表。

shouldStop
当测试的执行应当被*stop()* 停止时则设为 `True`。

testsRun
目前已运行的测试的总数量。

buffer
如果设为真值，sys.stdout 和 sys.stderr 将在*startTest()* 和*stopTest()* 被调用之间被缓冲。被收集的输出将仅在测试失败或发生错误时才会被回显到真正的 sys.stdout 和 sys.stderr。任何输出还会被附加到失败/错误消息中。

3.2 版新加入.

failfast
如果设为真值则*stop()* 将在首次失败或错误时被调用，停止测试运行。

3.2 版新加入.

tb_locals
如果设为真值则局部变量将被显示在回溯信息中。

3.5 版新加入.

wasSuccessful()
如果当前所有测试都已通过则返回 True，否则返回 False。

3.4 版更變: 如果有任何来自测试的*unexpectedSuccesses* 被*expectedFailure()* 装饰器所标记则返回 False。

stop()
此方法可被调用以提示正在运行的测试集要将*shouldStop* 属性设为 True 来表示其应当被中止。TestRunner 对象应当认同此旗标并返回而不再运行任何额外的测试。

例如，该特性会被*TextTestRunner* 类用来在当用户从键盘发出一个中断信号时停止测试框架。提供了 TestRunner 实现的交互式工具也可通过类似方式来使用该特性。

TestResult 类的下列方法被用于维护内部数据结构，并可在子类中被扩展以支持额外的报告需求。这特别适用于构建支持在运行测试时提供交互式报告的工具。

startTest(*test*)
当测试用例 *test* 即将被运行时被调用。

stopTest(*test*)
在测试用例 *test* 已经执行后被调用，无论其结果如何。

startTestRun()
在任何测试被执行之前被调用一次。

3.1 版新加入.

stopTestRun()
在所有测试被执行之后被调用一次。

3.1 版新加入.

addError(*test*, *err*)
当测试用例 *test* 引发了非预期的异常时被调用。*err* 是一个元组，其形式与*sys.exc_info()* 的返回值相同: (type, value, traceback)。

默认实现会将一个元组 (test, formatted_err) 添加到实例的*errors* 属性，其中 *formatted_err* 是派生自 *err* 的已格式化回溯信息。

addFailure(*test*, *err*)
当测试用例 *test* 发出了失败信号时将被调用。*err* 是一个元组，其形式与*sys.exc_info()* 的返回值相同: (type, value, traceback)。

默认实现会将一个元组 (test, formatted_err) 添加到实例的*failures* 属性，其中 *formatted_err* 是派生自 *err* 的已格式化回溯信息。

addSuccess(*test*)
当测试用例 *test* 成功时被调用。

默认实现将不做任何操作。

addSkip(*test*, *reason*)
当测试用例 *test* 被跳过时将被调用。*reason* 是给出的跳过测试的理由。

默认实现会将一个元组 (test, reason) 添加到实例的*skipped* 属性。

addExpectedFailure(*test*, *err*)
当测试用例 *test* 失败或发生错误，但是使用了*expectedFailure()* 装饰器来标记时将被调用。

默认实现会将一个元组 (test, formatted_err) 添加到实例的*expectedFailures* 属性，其中 *formatted_err* 是派生自 *err* 的已格式化回溯信息。

addUnexpectedSuccess(*test*)
当测试用例 *test* 使用了 was marked with the *expectedFailure()* 装饰器来标记，但是却执行成功时将被调用。

默认实现会将该测试添加到实例的*unexpectedSuccesses* 属性。

addSubTest(*test*, *subtest*, *outcome*)
当一个子测试结束时将被调用。*test* 是对应于该测试方法的测试用例。*subtest* 是一个描述该子测试的*TestCase* 实例。

如果 *outcome* 为*None*，则该子测试执行成功。否则，它将失败并引发一个异常，*outcome* 是一个元组，其形式与*sys.exc_info()* 的返回值相同: (type, value, traceback)。

默认实现在测试结果为成功时将不做任何事，并会将子测试的失败记录为普通的失败。

3.4 版新加入.

class unittest.**TextTestResult**(*stream*, *descriptions*, *verbosity*)
TestResult 的一个具体实现，由*TextTestRunner* 使用。

3.2 版新加入: 这个类在之前被命名为 _TextTestResult。这个旧名字仍然作为别名存在，但已被弃用。

unittest.**defaultTestLoader**
用于分享的*TestLoader* 类实例。如果不需要自制*TestLoader*，则可以使用该实例而不必重复创建新的实例。

class unittest.**TextTestRunner**(*stream=None*, *descriptions=True*, *verbosity=1*, *failfast=False*, *buffer=False*, *resultclass=None*, *warnings=None*, *, *tb_locals=False*)
一个将结果输出到流的基本测试运行器。如果 *stream* 为默认的 None，则会使用*sys.stderr* 作为输出流。这个类具有一些配置形参，但实际上都非常简单。运行测试套件的图形化应用程序应当提供替代实现。这样的实现应当在添加新特性到 unittest 时接受 **kwargs 作为修改构造运行器的接口。

在默认情况下这个运行器会显示*DeprecationWarning*, *PendingDeprecationWarning*, *ResourceWarning* 和*ImportWarning*，即使它们被默认忽略。由*deprecated unittest methods* 所导致的弃用警告也会被作为特例，并且当警告过滤器为 'default' 或 'always' 时，对于每个模块它们将仅显示一次，以避免过多的警告消息。此种行为可使用 Python 的 -Wd 或 -Wa 选项 (参见 警告控制) 并让 *warnings* 保持为 None 来覆盖。

3.2 版更變: 增加了 warnings 参数。

3.2 版更變: 默认流会在实例化而不是在导入时被设为*sys.stderr*。

3.5 版更變: 增加了 tb_locals 形参。

_makeResult()
此方法将返回由*run()* 使用的 TestResult 实例。它不应当被直接调用，但可在子类中被重载以提供自定义的 TestResult。

_makeResult() 会实例化传给 TextTestRunner 构造器的 resultclass 参数所指定的类或可迭代对象。如果没有提供 resultclass 则默认为*TextTestResult*。结果类会使用以下参数来实例化:

```
stream, descriptions, verbosity
```

run(*test*)
此方法是 TextTestRunner 的主要公共接口。此方法接受一个*TestSuite* 或*TestCase* 实例。通过调用*_makeResult()* 创建*TestResult* 来运行测试并将结果打印到标准输出。

unittest.**main**(*module='__main__'*, *defaultTest=None*, *argv=None*, *testRunner=None*, *testLoader=unittest.defaultTestLoader*, *exit=True*, *verbosity=1*, *failfast=None*, *catchbreak=None*, *buffer=None*, *warnings=None*)
从 *module* 加载一组测试并运行它们的命令行程序；这主要是为了让测试模块能方便地执行。此函数的最简单用法是在测试脚本末尾包括下列行:

```
if __name__ == '__main__':
    unittest.main()
```

你可以通过传入冗余参数运行测试以获得更详细的信息:

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

defaultTest 参数是要运行的单个测试名称，或者如果未通过 *argv* 指定任何测试名称则是包含多个测试名称的可迭代对象。如果未指定或为 None 且未通过 *argv* 指定任何测试名称，则会运行在 *module* 中找到的所有测试。

argv 参数可以是传给程序的选项列表，其中第一个元素是程序名。如未指定或为 None，则会使用*sys.argv* 的值。

testRunner 参数可以是测试运行器类或是其已创建的实例。在默认情况下 main 会调用*sys.exit()* 并附带一个退出码来指明测试运行是成功还是失败。

testLoader 参数必须是一个*TestLoader* 实例，其默认值为*defaultTestLoader*。

main 支持通过传入 exit=False 参数以便在交互式解释器中使用。这将在标准输出中显示结果而不调用*sys.exit()*:

```
>>> from unittest import main
>>> main(module='test_module', exit=False)
```

failfast, *catchbreak* 和 *buffer* 形参的效果与同名的*command-line options* 一致。

warnings 参数指定在运行测试时所应使用的警告过滤器。如果未指定，则默认的 None 会在将 -W 选项传给 **python** 命令时被保留 (参见 警告控制)，而在其他情况下将被设为 'default'。

调用 main 实际上将返回一个 TestProgram 类的实例。这会把测试运行结果保存为 result 属性。

3.1 版更變: 增加了 *exit* 形参。

3.2 版更變: 增加了 *verbosity*, *failfast*, *catchbreak*, *buffer* 和 *warnings* 形参。

3.4 版更變: *defaultTest* 形参被修改为也接受一个由测试名称组成的迭代器。

load_tests 协议

3.2 版新加入.

模块或包可以通过实现一个名为 `load_tests` 的函数来定制在正常测试运行或测试发现期间要如何从中加载测试。

如果一个测试模块定义了 `load_tests` 则它将被*TestLoader.loadTestsFromModule()* 调用并传入下列参数:

```
load_tests(loader, standard_tests, pattern)
```

其中 *pattern* 会通过 `loadTestsFromModule` 传入。它的默认值为 `None`。

它应当返回一个*TestSuite*。

loader 是执行载入操作的*TestLoader* 实例。*standard_tests* 是默认要从该模块载入的测试。测试模块通常只需从标准测试集中添加或移除测试。第三个参数是在作为测试发现的一部分载入包时使用的。

一个从指定*TestCase* 类集合中载入测试的 `load_tests` 函数看起来可能是这样的:

```
test_cases = (TestCase1, TestCase2, TestCase3)

def load_tests(loader, tests, pattern):
    suite = TestSuite()
    for test_class in test_cases:
        tests = loader.loadTestsFromTestCase(test_class)
        suite.addTests(tests)
    return suite
```

如果发现操作是在一个包含包的目录中开始的，不论是通过命令行还是通过调用*TestLoader.discover()*，则将在包 `__init__.py` 中检查 `load_tests`。如果不存在此函数，则发现将在包内部执行递归，就像它是另一个目录一样。在其他情况下，包中测试的发现操作将留给 `load_tests` 执行，它将附带下列参数被调用:

```
load_tests(loader, standard_tests, pattern)
```

这应当返回代表包中所有测试的*TestSuite*。(`standard_tests` 将只包含从 `__init__.py` 获取的测试。)

因为模式已被传入 `load_tests` 所以包可以自由地继续（还可能修改）测试发现操作。针对一个测试包的'无操作' `load_tests` 函数看起来是这样的:

```
def load_tests(loader, standard_tests, pattern):
    # top level directory cached on loader instance
    this_dir = os.path.dirname(__file__)
    package_tests = loader.discover(start_dir=this_dir, pattern=pattern)
    standard_tests.addTests(package_tests)
    return standard_tests
```

3.5 版更變: 发现操作不会再检查包名称是否匹配 *pattern*，因为包名称不可能匹配默认的模式。

26.8.9 类与模块设定

类与模块设定是在*TestSuite* 中实现的。当测试套件遇到来自新类的测试时则来自之前的类（如果存在）的 `tearDownClass()` 会被调用，然后再调用来自新类的 `setUpClass()`。

类似地如果测试是来自之前的测试的另一个模块则来自之前模块的 `tearDownModule` 将被运行，然后再运行来自新模块的 `setUpModule`。

在所有测试运行完毕后最终的 `tearDownClass` 和 `tearDownModule` 将被运行。

请注意共享设定不适用于一些 [潜在的] 特性例如测试并行化并且它们会破坏测试隔离。它们应当被谨慎地使用。

由 unittest 测试加载器创建的测试的默认顺序是将所有来自相同模块和类的测试归入相同分组。这将导致 `setUpClass` / `setUpModule` (等) 对于每个类和模块都恰好被调用一次。如果你将顺序随机化，以便使得来自不同模块和类的测试彼此相邻，那么这些共享的设定函数就可能会在一次测试运行中被多次调用。

共享的设定不适用与非标准顺序的套件。对于不想支持共享设定的框架来说 `BaseTestSuite` 仍然可用。

如果在共享的设定函数中引发了任何异常则测试将被报告错误。因为没有对应的测试实例，所以会创建一个 `_ErrorHolder` 对象（它具有与*TestCase* 相同的接口）来代表该错误。如果你只是使用标准 unittest 测试运行器那么这个细节并不重要，但是如果你是一个框架开发者那么这可能会有关系。

setUpClass 和 tearDownClass

这些必须被实现为类方法:

```
import unittest

class Test(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls._connection = createExpensiveConnectionObject()

    @classmethod
    def tearDownClass(cls):
        cls._connection.destroy()
```

如果你希望在基类上的 `setUpClass` 和 `tearDownClass` 被调用则你必须自己云调用它们。在*TestCase* 中的实现是空的。

如果在 `setUpClass` 中引发了异常则类中的测试将不会被运行并且 `tearDownClass` 也不会被运行。跳过的类中的 `setUpClass` 或 `tearDownClass` 将不会被运行。如果引发的异常是*SkipTest* 异常则类将被报告为已跳过而非发生错误。

setUpModule 和 tearDownModule

这些应当被实现为函数:

```
def setUpModule():
    createConnection()

def tearDownModule():
    closeConnection()
```

如果在 `setUpModule` 中引发了异常则模块中的任何测试都将不会被运行并且 `tearDownModule` 也不会被运行。如果引发的异常是*SkipTest* 异常则模块将被报告为已跳过而非发生错误。

要添加即使在发生异常时也必须运行的清理代码，请使用 addModuleCleanup:

unittest.**addModuleCleanup**(*function*, */*, **args*, ***kwargs*)
在 tearDownModule() 之后添加一个要调用的函数来清理测试类运行期间所使用的资源。函数将按它们被添加的相反顺序被调用 (LIFO)。它们在调用时将附带它们被添加时传给*addModuleCleanup()*的任何参数和关键字参数。

如果 setUpModule() 失败，即意味着 tearDownModule() 未被调用，则已添加的任何清理函数仍将被调用。

3.8 版新加入.

unittest.**doModuleCleanups**()
此函数会在 tearDownModule() 之后无条件地被调用，或者如果 setUpModule() 引发了异常则会在 setUpModule() 之后被调用。

It is responsible for calling all the cleanup functions added by *addModuleCleanup()*. If you need cleanup functions to be called *prior* to tearDownModule() then you can call *doModuleCleanups()* yourself.

doModuleCleanups() 每次会弹出清理函数栈中的一个方法，因此它可以在任何时候被调用。

3.8 版新加入.

26.8.10 信号处理

3.2 版新加入.

unittest 的*-c/--catch* 命令行选项，加上*unittest.main()* 的 catchbreak 形参，提供了在测试运行期间处理 control-C 的更友好方式。在捕获中断行为被启用时 control-C 将允许当前运行的测试能够完成，而测试运行将随后结束并报告已有的全部结果。第二个 control-C 将会正常地引发*KeyboardInterrupt*。

处理 control-C 信号的句柄会尝试与安装了自定义*signal.SIGINT* 处理句柄的测试代码保持兼容。如果是 unittest 处理句柄而 *不是*已安装的*signal.SIGINT* 处理句柄被调用，即它被系统在测试的下层替换并委托处理，则它会调用默认的处理句柄。这通常会是替换了已安装处理句柄并委托处理的代码所预期的行为。对于需要禁用 unittest control-C 处理的单个测试则可以使用*removeHandler()* 装饰器。

还有一些工具函数让框架开发者可以在测试框架内部启用 control-C 处理功能。

unittest.**installHandler**()
安装 control-C 处理句柄。当接收到*signal.SIGINT* 时（通常是响应用户按下 control-C）所有已注册的结果都会执行*stop()* 调用。

unittest.**registerResult**(*result*)
注册一个*TestResult* 对象用于 control-C 的处理。注册一个结果将保存指向它的弱引用，因此这并不能防止结果被作为垃圾回收。

如果 control-C 未被启用则注册*TestResult* 对象将没有任何附带影响，因此不论是否启用了该项处理测试框架都可以无条件地注册他们独立创建的所有结果。

unittest.**removeResult**(*result*)
移除一个已注册的结果。一旦结果被移除则*stop()* 将不再会作为针对 control-C 的响应在结果对象上被调用。

unittest.**removeHandler**(*function=None*)
当不附带任何参数被调用时此函数将移除已被安装的 control-C 处理句柄。此函数还可被用作测试装饰器以在测试被执行时临时性地移除处理句柄:

```
@unittest.removeHandler
def test_signal_handling(self):
    ...
```

26.9 unittest.mock --- mock 对象库

3.3 版新加入.

源代码： Lib/unittest/mock.py

---

unittest.mock 是一个用于测试的 Python 库。它允许使用 mock 对象替换受测试系统的部分，并对它们如何已经被使用进行断言。

unittest.mock 提供的*Mock* 类，能在整个测试套件中模拟大量的方法。创建后，就可以断言调用了哪些方法/属性及其参数。还可以以常规方式指定返回值并设置所需的属性。

此外，mock 提供了用于修补测试范围内模块和类级别属性的*patch()* 装饰器，和用于创建独特对象的*sentinel* 。阅读*quick guide* 中的案例了解如何使用*Mock* ， *MagicMock* 和*patch()* 。

mock 被设计为配合*unittest* 使用且它是基于'action -> assertion' 模式而非许多模拟框架所使用的'record -> replay' 模式。

在 Python 的早期版本要单独使用*unittest.mock* ，在 PyPI 获取 mock 。

26.9.1 快速上手

当您访问对象时，*Mock* 和*MagicMock* 将创建所有属性和方法，并保存他们在使用时的细节。你可以通过配置，指定返回值或者限制可访问属性，然后断言他们如何被调用：

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

通过 side_effect 设置副作用 (side effects)，可以是一个 mock 被调用是抛出的异常：

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
```

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]
...
>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

Mock 还可以通过其他方法配置和控制其行为。例如 mock 可以通过设置 *spec* 参数来从一个对象中获取其规格 (specification)。如果访问 mock 的属性或方法不在 spec 中，会报*AttributeError* 错误。

使用*patch()* 装饰去/上下文管理器，可以更方便地测试一个模块下的类或对象。你指定的对象会在测试过程中替换成 mock （或者其他对象），测试结束后恢复。

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

備𝔽: 当你嵌套 patch 装饰器时，mock 将以执行顺序传递给装饰器函数（*Python* 装饰器正常顺序）。由于从下至上，因此在上面的示例中，首先 mock 传入的 `module.ClassName1` 。

在查找对象的名称空间中修补对象使用*patch()* 。使用起来很简单，阅读*在哪里打补丁* 来快速上手。

patch() 也可以 with 语句中使用上下文管理。

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)
...
>>> mock_method.assert_called_once_with(1, 2, 3)
```

还有一个*patch.dict()* 用于在一定范围内设置字典中的值，并在测试结束时将字典恢复为其原始状态：

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

Mock 支持 Python *魔术方法* 。使用模式方法最简单的方式是使用*MagicMock* class. 。它可以做如下事情：

```
>>> mock = MagicMock()
>>> mock.__str__.return_value = 'foobarbaz'
>>> str(mock)
'foobarbaz'
>>> mock.__str__.assert_called_with()
```

Mock 能指定函数（或其他 Mock 实例）为魔术方法，它们将被适当地调用。*MagicMock* 是预先创建了所有魔术方法（所有有用的方法）的 Mock 。

下面是一个使用了普通 Mock 类的魔术方法的例子

```
>>> mock = Mock()
>>> mock.__str__ = Mock(return_value='wheeeeee')
>>> str(mock)
'wheeeeee'
```

使用*auto-speccing* 可以保证测试中的模拟对象与要替换的对象具有相同的 api 。在 patch 中可以通过 *autospec* 参数实现自动推断，或者使用*create_autospec()* 函数。自动推断会创建一个与要替换对象相同的属相和方法的模拟对象，并且任何函数和方法（包括构造函数）都具有与真实对象相同的调用签名。

这么做是为了因确保不当地使用 mock 导致与生产代码相同的失败：

```
>>> from unittest.mock import create_autospec
>>> def function(a, b, c):
...     pass
...
>>> mock_function = create_autospec(function, return_value='fishy')
>>> mock_function(1, 2, 3)
'fishy'
>>> mock_function.assert_called_once_with(1, 2, 3)
>>> mock_function('wrong arguments')
Traceback (most recent call last):
 ...
TypeError: <lambda>() takes exactly 3 arguments (1 given)
```

在类中使用*create_autospec()* 时，会复制 __init__ 方法的签名，另外在可调用对象上使用时，会复制 __call__ 方法的签名。

26.9.2 Mock 类

Mock 是一个可以灵活的替换存根 (stubs) 的对象，可以测试所有代码。Mock 是可调用的，在访问其属性时创建一个新的 mock[1] 。访问相同的属性只会返回相同的 mock 。Mock 会保存调用记录，可以通过断言获悉代码的调用。

MagicMock 是*Mock* 的子类，它有所有预创建且可使用的魔术方法。在需要模拟不可调用对象时，可以使用*NonCallableMock* 和*NonCallableMagicMock*

patch() 装饰器便于*Mock* 对象临时替换特定模块中的类。*patch()* 默认创建*MagicMock* 。可以使用*patch()* 方法的 *new_callable* 参数指定*Mock* 的替代类。

class unittest.mock.**Mock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

创建一个新的*Mock* 对象。通过可选参数指定*Mock* 对象的行为：

- *spec*: 可以是要给字符串列表，也可以是充当模拟对象规范的现有对象（类或实例）。如果传入一个对象，则通过在该对象上调用 dir 来生成字符串列表（不支持的魔法属性和方法除外）。访问不在此列表中的任何属性都将触发*AttributeError* 。

  如果 *spec* 是一个对象（而不是字符串列表），则*__class__* 返回 *spec* 对象的类。这允许模拟程序通过*isinstance()* 测试。

- *spec_set*： *spec* 的更严格的变体。如果使用了该属性，尝试模拟 *set* 或 *get* 的属性不在 *spec_set* 所包含的对象中时，会抛出*AttributeError* 。

- *side_effect*： 每当调用 Mock 时都会调用的函数。参见*side_effect* 属性。对于引发异常或动态更改返回值很有用。该函数使用与 mock 函数相同的参数调用，并且除非返回*DEFAULT* ，否则该函数的返回值将用作返回值。

  另外，*side_effect* 可以是异常类或实例。此时，调用模拟程序时将引发异常。

  如果 *side_effect* 是可迭代对象，则每次调用 mock 都将返回可迭代对象的下一个值。

  设置 *side_effect* 为 None 即可清空。

- *return_value*： 调用 mock 的返回值。默认情况下，是一个新的 Mock（在首次访问时创建）。参见*return_value* 属性。

[1] 仅有的例外是魔术方法和属性（其名称前后都带有双下划线）。Mock 不会创建它们而是将引发*AttributeError*。这是因为解释器将会经常隐式地请求这些方法，并且在它准备接受一个魔术方法却得到一个新的 Mock 对象时会 相当困惑。如果你需要魔术方法支持请参阅魔术方法。

- *unsafe*： 默认情况下，如果任何以 *assert* 或 *assret* 开头的属性都将引发*AttributeError* 。当 `unsafe=True` 时可以访问。

  3.5 版新加入.

- *wraps*： 要包装的 mock 对象。如果 *wraps* 不是 `None` ，那么调用 Mock 会将调用传递给 *wraps* 的对象（返回实际结果）。对模拟的属性访问将返回一个 Mock 对象，该对象包装了 *wraps* 对象的相应属性（因此，尝试访问不存在的属性将引发*AttributeError* ）。

  如果明确指定 *return_value* ，调用是，不会返回包装对象，而是返回 *return_value* 。

- *name*： mock 的名称。在调试时很有用。名称会传递到子 mock 。

还可以使用任意关键字参数来调用 mock 。创建模拟后，将使用这些属性来设置 mock 的属性。有关详细信息，请参见*configure_mock()* 方法。

assert_called()
断言至少被调用过一次。

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called()
```

3.6 版新加入.

assert_called_once()
断言仅被调用一次。

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_once()
Traceback (most recent call last):
...
AssertionError: Expected 'method' to have been called once. Called 2 times.
```

3.6 版新加入.

assert_called_with(**args*, ***kwargs*)
此方法是断言上次调用已以特定方式进行的一种便捷方法：

```
>>> mock = Mock()
>>> mock.method(1, 2, 3, test='wow')
<Mock name='mock.method()' id='...'>
>>> mock.method.assert_called_with(1, 2, 3, test='wow')
```

assert_called_once_with(**args*, ***kwargs*)
断言 mock 仅被调用一次，并且向该调用传入了指定的参数。

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar='baz')
>>> mock.assert_called_once_with('foo', bar='baz')
>>> mock('other', bar='values')
>>> mock.assert_called_once_with('other', bar='values')
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

assert_any_call(**args*, ***kwargs*)
断言使用指定的参数调用。

如果 mock 曾经被调用过则断言通过，不同于*assert_called_with()*和*assert_called_once_with()* 那样只有在调用是最近的一次时才会通过，而对于*assert_called_once_with()* 则它还必须是唯一的一次调用。

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, arg='thing')
>>> mock('some', 'thing', 'else')
>>> mock.assert_any_call(1, 2, arg='thing')
```

assert_has_calls(*calls*, *any_order=False*)
断言 mock 已附带指定的参数被调用。将针对这些调用检查*mock_calls* 列表。

如果 *any_order* 为假值则调用必须是顺序进行的。在指定的调用之前或之后还可以有额外的调用。

如果 *any_order* 为真值则调用可以是任意顺序的，但它们都必须在*mock_calls* 中出现。

```
>>> mock = Mock(return_value=None)
>>> mock(1)
>>> mock(2)
>>> mock(3)
>>> mock(4)
>>> calls = [call(2), call(3)]
>>> mock.assert_has_calls(calls)
>>> calls = [call(4), call(2), call(3)]
>>> mock.assert_has_calls(calls, any_order=True)
```

assert_not_called()
断言 mock 从未被调用过。

```
>>> m = Mock()
>>> m.hello.assert_not_called()
>>> obj = m.hello()
>>> m.hello.assert_not_called()
Traceback (most recent call last):
  ...
AssertionError: Expected 'hello' to not have been called. Called 1 times.
```

3.5 版新加入.

reset_mock(***, *return_value=False*, *side_effect=False*)
reset_mock 方法将在 mock 对象上重围所有的调用属性:

```
>>> mock = Mock(return_value=None)
>>> mock('hello')
>>> mock.called
True
>>> mock.reset_mock()
>>> mock.called
False
```

3.6 版更變: 向 reset_mock 函数添加了两个仅限关键字参数。

这在你想要创建一系列重用相同对象的断言时会很有用处。请注意*reset_mock()* 不会清除返回值、*side_effect* 或任何你使用普通赋值所默认设置的子属性。如果你想要重置 *return_value* 或*side_effect*，则要为相应的形参传入 `True`。子 mock 和返回值 mock (如果有的话) 也会被重置。

備🄵: *return_value* 和 *side_effect* 均为仅限关键字参数。

mock_add_spec(*spec*, *spec_set=False*)
为 mock 添加规格说明。*spec* 可以是一个对象或字符串列表。只有 *spec* 上的属性可以作为来自 mock 的属性被获取。

如果 *spec_set* 为真值则只有 spec 上的属性可以被设置。

attach_mock(*mock*, *attribute*)
附加一个 mock 作为这一个的属性，替换它的名称和上级。对附加 mock 的调用将记录在这一个的 *method_calls* 和 *mock_calls* 属性中。

configure_mock(***kwargs*)
通过关键字参数在 mock 上设置属性。

属性加返回值和附带影响可以使用标准点号标记在子 mock 上设置并在方法调用中解包一个字典:

```
>>> mock = Mock()
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock.configure_mock(**attrs)
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

同样的操作可在对 mock 的构造器调用中达成:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

configure_mock() 的存在是使得 mock 被创建之后的配置更为容易。

__dir__()
Mock 对象会将 `dir(some_mock)` 的结果限制为有用结果。对于带有 *spec* 的 mock 这还包括 mock 的所有被允许的属性。

请查看 *FILTER_DIR* 了解此过滤做了什么，以及如何停用它。

_get_child_mock(***kw*)
创建针对属性和返回值的子 mock。默认情况下子 mock 将为与其上级相同的类型。Mock 的子类可能需要重载它来定制子 mock 的创建方式。

对于非可调用的 mock 将会使用可调用的变化形式（而非不是任意的自定义子类）。

called
一个表示 mock 对象是否已被调用的布尔值:

```
>>> mock = Mock(return_value=None)
>>> mock.called
False
>>> mock()
>>> mock.called
True
```

call_count

一个告诉你 mock object 对象已被调用多少次的整数值:

```
>>> mock = Mock(return_value=None)
>>> mock.call_count
0
>>> mock()
>>> mock()
>>> mock.call_count
2
```

return_value

设置这个来配置通过调用该 mock 所返回的值:

```
>>> mock = Mock()
>>> mock.return_value = 'fish'
>>> mock()
'fish'
```

默认的返回值是一个 mock 对象并且你可以通过正常方式来配置它:

```
>>> mock = Mock()
>>> mock.return_value.attribute = sentinel.Attribute
>>> mock.return_value()
<Mock name='mock()()' id='...'>
>>> mock.return_value.assert_called_with()
```

return_value 也可以在构造器中设置:

```
>>> mock = Mock(return_value=3)
>>> mock.return_value
3
>>> mock()
3
```

side_effect

这可以是当该 This can either be a function to be called when the mock 被调用时将被调用的一个函数，可调用对象或者要被引发的异常（类或实例）。

如果你传入一个函数则它将附带与该 mock 相同的参数被调用并且除了该函数返回*DEFAULT* 单例的情况以外对该 mock 的调用都将随后返回该函数所返回的任何东西。如果该函数返回*DEFAULT* 则该 mock 将返回其正常值 (来自*return_value*)。

如果你传入一个可迭代对象，它会被用来获取一个在每次调用时必须产生一个值的迭代器。这个值可以是一个要被引发的异常实例，或是一个要从该调用返回给 mock 的值 (*DEFAULT* 处理与函数的情况一致)。

一个引发异常（来测试 API 的异常处理）的 mock 的示例:

```
>>> mock = Mock()
>>> mock.side_effect = Exception('Boom!')
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

使用*side_effect* 来返回包含多个值的序列:

```
>>> mock = Mock()
>>> mock.side_effect = [3, 2, 1]
>>> mock(), mock(), mock()
(3, 2, 1)
```

使用一个可调用对象:

```
>>> mock = Mock(return_value=3)
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> mock.side_effect = side_effect
>>> mock()
3
```

side_effect 可以在构造器中设置。下面是在 mock 被调用时增加一个该属性值并返回它的例子:

```
>>> side_effect = lambda value: value + 1
>>> mock = Mock(side_effect=side_effect)
>>> mock(3)
4
>>> mock(-8)
-7
```

将*side_effect* 设为 None 可以清除它:

```
>>> m = Mock(side_effect=KeyError, return_value=3)
>>> m()
Traceback (most recent call last):
 ...
KeyError
>>> m.side_effect = None
>>> m()
3
```

call_args

这可以是 None (如果 mock 没有被调用)，或是 mock 最近一次被调用时附带的参数。这将采用元组的形式：第一个成员也可以通过 args 特征属性来访问，它是 mock 被调用时所附带的任何位置参数 (或为空元组)，而第二个成员也可以通过 kwargs 特征属性来访问，它则是任何关键字参数 (或为空字典)。

```
>>> mock = Mock(return_value=None)
>>> print(mock.call_args)
None
>>> mock()
>>> mock.call_args
call()
```

(下页继续)

(繼續上一頁)

```
>>> mock.call_args == ()
True
>>> mock(3, 4)
>>> mock.call_args
call(3, 4)
>>> mock.call_args == ((3, 4),)
True
>>> mock.call_args.args
(3, 4)
>>> mock.call_args.kwargs
{}
>>> mock(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args
call(3, 4, 5, key='fish', next='w00t!')
>>> mock.call_args.args
(3, 4, 5)
>>> mock.call_args.kwargs
{'key': 'fish', 'next': 'w00t!'}
```

call_args, 以及列表*call_args_list*, *method_calls* 和*mock_calls* 的成员都是*call* 对象。这些对象属性元组，因此它们可被解包以获得单独的参数并创建更复杂的断言。参见*作为元组的 call*。

3.8 版更變: 增加了 `args` 和 `kwargs` 特征属性。properties.

call_args_list

这是一个已排序的对 mock 对象的所有调用的列表（因此该列表的长度就是它已被调用的次数）。在执行任何调用之前它将是一个空列表。*call* 对象可以被用来方便地构造调用列表以与*call_args_list* 相比较。

```
>>> mock = Mock(return_value=None)
>>> mock()
>>> mock(3, 4)
>>> mock(key='fish', next='w00t!')
>>> mock.call_args_list
[call(), call(3, 4), call(key='fish', next='w00t!')]
>>> expected = [(), ((3, 4),), ({'key': 'fish', 'next': 'w00t!'},)]
>>> mock.call_args_list == expected
True
```

call_args_list 的成员均为*call* 对象。它们可作为元组被解包以获得单个参数。参见*作为元组的 call*。

method_calls

除了会追踪对其自身的调用，mock 还会追踪对方法和属性，以及 它们的方法和属性的访问:

```
>>> mock = Mock()
>>> mock.method()
<Mock name='mock.method()' id='...'>
>>> mock.property.method.attribute()
<Mock name='mock.property.method.attribute()' id='...'>
>>> mock.method_calls
[call.method(), call.property.method.attribute()]
```

method_calls 的成员均为*call* 对象。它们可以作为元组被解包以获得单个参数。参见*作为元组的 call*.

mock_calls

mock_calls 会记录 所有对 mock 对象、它的方法、魔术方法的调用 以及返回值的 mock。

```
>>> mock = MagicMock()
>>> result = mock(1, 2, 3)
>>> mock.first(a=3)
<MagicMock name='mock.first()' id='...'>
>>> mock.second()
<MagicMock name='mock.second()' id='...'>
>>> int(mock)
1
>>> result(1)
<MagicMock name='mock()()' id='...'>
>>> expected = [call(1, 2, 3), call.first(a=3), call.second(),
... call.__int__(), call()(1)]
>>> mock.mock_calls == expected
True
```

mock_calls 的成员均为*call* 对象。它们可以作为元组被解包以获得单个参数。参见*作为元组的 call*。

備🄵: *mock_calls* 的记录方式意味着在进行嵌套调用时，之前调用的形参不会被记录因而这样的比较将总是相等:

```
>>> mock = MagicMock()
>>> mock.top(a=3).bottom()
<MagicMock name='mock.top().bottom()' id='...'>
>>> mock.mock_calls
[call.top(a=3), call.top().bottom()]
>>> mock.mock_calls[-1] == call.top(a=-1).bottom()
True
```

__class__

通常一个对象的*__class__* 属性将返回其类型。对于具有 spec 的 mock 对象来说，__class__ 将改为返回 spec 类。这将允许 mock 对象为它们所替换 / 屏蔽的对象跳过*isinstance()* 测试:

```
>>> mock = Mock(spec=3)
>>> isinstance(mock, int)
True
```

__class__ 是可以被赋值的，这将允许 mock 跳过*isinstance()* 检测而不强制要求你使用 spec:

```
>>> mock = Mock()
>>> mock.__class__ = dict
>>> isinstance(mock, dict)
True
```

class unittest.mock.**NonCallableMock**(*spec=None*, *wraps=None*, *name=None*, *spec_set=None*, ***kwargs*)

不可调用的*Mock* 版本。其构造器的形参具有与*Mock* 相同的含义，区别在于 *return_value* 和 *side_effect* 在不可调用的 mock 上没有意义。

使用类或实例作为 spec 或 spec_set 的 mock 对象能够跳过*isinstance()* 测试:

```
>>> mock = Mock(spec=SomeClass)
>>> isinstance(mock, SomeClass)
True
>>> mock = Mock(spec_set=SomeClass())
>>> isinstance(mock, SomeClass)
True
```

Mock 类具有对模拟魔术方法的支持。请参阅*魔术方法* 了解完整细节。

mock 操作类和*patch()* 装饰器都接受任意关键字参数用于配置。对于*patch()* 装饰器来说关键字参数会被传给所创建 mock 的构造器。这些关键字被用于配置 mock 的属性:

```
>>> m = MagicMock(attribute=3, other='fish')
>>> m.attribute
3
>>> m.other
'fish'
```

子 mock 的返回值和附带效果也可使用带点号的标记通过相同的方式来设置。由于你无法直接在调用中使用带点号的名称因此你需要创建一个字典并使用 ** 来解包它:

```
>>> attrs = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> mock = Mock(some_attribute='eggs', **attrs)
>>> mock.some_attribute
'eggs'
>>> mock.method()
3
>>> mock.other()
Traceback (most recent call last):
  ...
KeyError
```

使用 *spec* (或 *spec_set*) 创建的可调用 mock 将在匹配调用与 mock 时内省规格说明对象的签名。因此，它可以匹配实际调用的参数而不必关心它们是按位置还是按名称传入的:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, c=3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(1, 2, 3)
>>> mock.assert_called_with(a=1, b=2, c=3)
```

这适用于*assert_called_with()*, *assert_called_once_with()*, *assert_has_calls()* 和*assert_any_call()*。当执行*自动 spec* 时，它还将应用于 mock 对象的方法调用。

3.4 版更變: 添加了在附带规格说明和自动规格说明的 mock 对象上的签名内省

class unittest.mock.**PropertyMock**(**args*, ***kwargs*)

旨在作为类的特征属性或其他描述器使用的 mock。*PropertyMock* 提供了 __get__() 和 __set__() 方法以便你可以在它被提取时指定一个返回值。

当从一个对象提取*PropertyMock* 实例时将不附带任何参数地调用该 mock。如果设置它则调用该 mock 时将附带被设置的值。

```
>>> class Foo:
...     @property
...     def foo(self):
```

(下页继续)

(繼續上一頁)

```
...         return 'something'
...     @foo.setter
...     def foo(self, value):
...         pass
...
>>> with patch('__main__.Foo.foo', new_callable=PropertyMock) as mock_foo:
...     mock_foo.return_value = 'mockity-mock'
...     this_foo = Foo()
...     print(this_foo.foo)
...     this_foo.foo = 6
...
mockity-mock
>>> mock_foo.mock_calls
[call(), call(6)]
```

由于 mock 属性的存储方式你无法直接将*PropertyMock* 附加到一个 mock 对象。但是你可以将它附加到 mock 类型对象:

```
>>> m = MagicMock()
>>> p = PropertyMock(return_value=3)
>>> type(m).foo = p
>>> m.foo
3
>>> p.assert_called_once_with()
```

class unittest.mock.**AsyncMock**(*spec=None*, *side_effect=None*, *return_value=DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *unsafe=False*, ***kwargs*)

MagicMock 的异步版本。*AsyncMock* 对象的行为方式将使该对象被识别为异步函数，其调用的结果将为可等待对象。

```
>>> mock = AsyncMock()
>>> asyncio.iscoroutinefunction(mock)
True
>>> inspect.isawaitable(mock())
True
```

调用 `mock()` 的结果是一个异步函数，它在被等待之后将具有 `side_effect` 或 `return_value` 的结果:

- 如果 `side_effect` 是一个函数，则异步函数将返回该函数的结果,
- 如果 `side_effect` 是一个异常，则异步函数将引发该异常,
- 如果 `side_effect` 是一个可迭代对象，则异步函数将返回该可迭代对象的下一个值，但是，如果结果序列被耗尽，则会立即引发 `StopAsyncIteration`,
- 如果 `side_effect` 未被定义，则异步函数将返回 is not defined, the async function will return the value defined by `return_value` 所定义的值，因而，在默认情况下，异步函数会返回一个新的*AsyncMock* 对象。

将*Mock* 或*MagicMock* 的 *spec* 设为异步函数将导致在调用后返回一个协程对象。

```
>>> async def async_func(): pass
...
>>> mock = MagicMock(async_func)
>>> mock
```

(下页继续)

(繼續上一頁)

```
<MagicMock spec='function' id='...'>
>>> mock()
<coroutine object AsyncMockMixin._mock_call at ...>
```

将*Mock*, *MagicMock* 或*AsyncMock* 的 *spec* 设为带有异步和同步函数的类将自动删除其中的同步函数并将它们设为*MagicMock* (如果上级 mock 是*AsyncMock* 或*MagicMock*) 或者*Mock* (如果上级 mock 是*Mock*)。所有异步函数都将为*AsyncMock*。

```
>>> class ExampleClass:
...     def sync_foo():
...         pass
...     async def async_foo():
...         pass
...
>>> a_mock = AsyncMock(ExampleClass)
>>> a_mock.sync_foo
<MagicMock name='mock.sync_foo' id='...'>
>>> a_mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
>>> mock = Mock(ExampleClass)
>>> mock.sync_foo
<Mock name='mock.sync_foo' id='...'>
>>> mock.async_foo
<AsyncMock name='mock.async_foo' id='...'>
```

3.8 版新加入.

assert_awaited()

断言 mock 已被等待至少一次。请注意这是从被调用的对象中分离出来的，必须要使用 `await` 关键字:

```
>>> mock = AsyncMock()
>>> async def main(coroutine_mock):
...     await coroutine_mock
...
>>> coroutine_mock = mock()
>>> mock.called
True
>>> mock.assert_awaited()
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited.
>>> asyncio.run(main(coroutine_mock))
>>> mock.assert_awaited()
```

assert_awaited_once()

断言 mock 已被等待恰好一次。

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.assert_awaited_once()
>>> asyncio.run(main())
>>> mock.method.assert_awaited_once()
```

(下页继续)

(繼續上一頁)

```
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_awaited_with(**args*, ***kwargs*)

斷言上一次等待传入了指定的参数。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_with('foo', bar='bar')
>>> mock.assert_awaited_with('other')
Traceback (most recent call last):
...
AssertionError: expected call not found.
Expected: mock('other')
Actual: mock('foo', bar='bar')
```

assert_awaited_once_with(**args*, ***kwargs*)

断言 mock 被等待恰好一次并且附带了指定的参数。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
>>> asyncio.run(main('foo', bar='bar'))
>>> mock.assert_awaited_once_with('foo', bar='bar')
Traceback (most recent call last):
...
AssertionError: Expected mock to have been awaited once. Awaited 2 times.
```

assert_any_await(**args*, ***kwargs*)

断言 mock 已被等待并附带了指定的参数。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> asyncio.run(main('foo', bar='bar'))
>>> asyncio.run(main('hello'))
>>> mock.assert_any_await('foo', bar='bar')
>>> mock.assert_any_await('other')
Traceback (most recent call last):
...
AssertionError: mock('other') await not found
```

assert_has_awaits(*calls*, *any_order=False*)

断言 mock 已附带了指定的调用被等待。将针对这些等待检查*await_args_list* 列表。

如果 *any_order* 为假值则等待必须是顺序进行的。在指定的等待之前或之后还可以有额外的调用。

如果 *any_order* 为真值则等待可以是任意顺序的，但它们都必须在*await_args_list* 中出现。

```
>>> mock = AsyncMock()
>>> async def main(*args, **kwargs):
...     await mock(*args, **kwargs)
...
>>> calls = [call("foo"), call("bar")]
>>> mock.assert_has_awaits(calls)
Traceback (most recent call last):
...
AssertionError: Awaits not found.
Expected: [call('foo'), call('bar')]
Actual: []
>>> asyncio.run(main('foo'))
>>> asyncio.run(main('bar'))
>>> mock.assert_has_awaits(calls)
```

assert_not_awaited()
断言 mock 从未被等待过。

```
>>> mock = AsyncMock()
>>> mock.assert_not_awaited()
```

reset_mock(**args*, ***kwargs*)
参见*Mock.reset_mock()*。还会将*await_count* 设为 0，将*await_args* 设为 None，并清空*await_args_list*。

await_count
一个追踪 mock 对象已被等待多少次的整数值。

```
>>> mock = AsyncMock()
>>> async def main():
...     await mock()
...
>>> asyncio.run(main())
>>> mock.await_count
1
>>> asyncio.run(main())
>>> mock.await_count
2
```

await_args
这可能为 None (如果 mock 从未被等待)，或为该 mock 上一次被等待所附带的参数。其功能与*Mock.call_args* 相同。

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args
>>> asyncio.run(main('foo'))
>>> mock.await_args
call('foo')
>>> asyncio.run(main('bar'))
>>> mock.await_args
call('bar')
```

await_args_list
这是由对 mock 对象按顺序执行的所有等待组成的列表（因此该列表的长度即它被等待的次数）。在有任何等待被执行之前它将为一个空列表。Before any awaits have been made it is an empty list.

```
>>> mock = AsyncMock()
>>> async def main(*args):
...     await mock(*args)
...
>>> mock.await_args_list
[]
>>> asyncio.run(main('foo'))
>>> mock.await_args_list
[call('foo')]
>>> asyncio.run(main('bar'))
>>> mock.await_args_list
[call('foo'), call('bar')]
```

调用

Mock 对象是可调用对象。调用将把值集合作为*return_value* 属性返回。默认的返回值是一个新的 Mock 对象；它会在对返回值的首次访问（不论是显式访问还是通过调用 Mock）时被创建——但它会被保存并且每次都返回相同的对象。

对该对象的调用将被记录在*call_args* 和*call_args_list* 等属性中。

如果设置了*side_effect* 则它将在调用被记录之后被调用，因此如果 side_effect 引发了异常该调用仍然会被记录。

让一个 mock 在被调用时引发异常的最简单方式是将*side_effect* 设为一个异常类或实例:

```
>>> m = MagicMock(side_effect=IndexError)
>>> m(1, 2, 3)
Traceback (most recent call last):
  ...
IndexError
>>> m.mock_calls
[call(1, 2, 3)]
>>> m.side_effect = KeyError('Bang!')
>>> m('two', 'three', 'four')
Traceback (most recent call last):
  ...
KeyError: 'Bang!'
>>> m.mock_calls
[call(1, 2, 3), call('two', 'three', 'four')]
```

如果 side_effect 为函数则该函数所返回的对象就是调用该 mock 所返回的对象。side_effect 函数在被调用时将附带与该 mock 相同的参数。这允许你根据输入动态地改变返回值:

```
>>> def side_effect(value):
...     return value + 1
...
>>> m = MagicMock(side_effect=side_effect)
>>> m(1)
2
>>> m(2)
3
>>> m.mock_calls
[call(1), call(2)]
```

如果你想让该 mock 仍然返回默认的返回值（一个新的 mock 对象），或是任何设定的返回值，那么有两种方式可以做到这一点。从 side_effect 内部返回 mock.return_value，或者返回*DEFAULT*:

```
>>> m = MagicMock()
>>> def side_effect(*args, **kwargs):
...     return m.return_value
...
>>> m.side_effect = side_effect
>>> m.return_value = 3
>>> m()
3
>>> def side_effect(*args, **kwargs):
...     return DEFAULT
...
>>> m.side_effect = side_effect
>>> m()
3
```

要移除一个 side_effect，并返回到默认的行为，请将 side_effect 设为 None:

```
>>> m = MagicMock(return_value=6)
>>> def side_effect(*args, **kwargs):
...     return 3
...
>>> m.side_effect = side_effect
>>> m()
3
>>> m.side_effect = None
>>> m()
6
```

side_effect 也可以是任意可迭代对象。对该 mock 的重复调用将返回来自该可迭代对象的值（直到该可迭代对象被耗尽并导致*StopIteration* 被引发）：

```
>>> m = MagicMock(side_effect=[1, 2, 3])
>>> m()
1
>>> m()
2
>>> m()
3
>>> m()
Traceback (most recent call last):
  ...
StopIteration
```

如果该可迭代对象有任何成员属于异常则它们将被引发而不是被返回:

```
>>> iterable = (33, ValueError, 66)
>>> m = MagicMock(side_effect=iterable)
>>> m()
33
>>> m()
Traceback (most recent call last):
 ...
ValueError
>>> m()
66
```

删除属性

Mock 对象会根据需要创建属性。这允许它们可以假装成任意类型的对象。

你可能想要一个 mock 对象在调用*hasattr()* 时返回 False，或者在获取某个属性时引发*AttributeError*。你可以通过提供一个对象作为 mock 的 spec 属性来做到这点，但这并不总是很方便。

你可以通过删除属性来“屏蔽”它们。属性一旦被删除，访问它将引发*AttributeError*。

```
>>> mock = MagicMock()
>>> hasattr(mock, 'm')
True
>>> del mock.m
>>> hasattr(mock, 'm')
False
>>> del mock.f
>>> mock.f
Traceback (most recent call last):
    ...
AttributeError: f
```

Mock 的名称与 name 属性

由于”name” 是*Mock* 构造器的参数之一，如果你想让你的 mock 对象具有”name” 属性你不可以在创建时传入该参数。有两个替代方式。一个选项是使用*configure_mock()*:

```
>>> mock = MagicMock()
>>> mock.configure_mock(name='my_name')
>>> mock.name
'my_name'
```

一个更简单的选项是在 mock 创建之后简单地设置”name” 属性:

```
>>> mock = MagicMock()
>>> mock.name = "foo"
```

附加 Mock 作为属性

当你附加一个 mock 作为另一个 mock 的属性（或作为返回值）时它会成为该 mock 的” 子对象”。对子对象的调用会被记录在父对象的*method_calls* 和*mock_calls* 属性中。这适用于配置子 mock 然后将它们附加到父对象，或是将 mock 附加到将记录所有对子对象的调用的父对象上并允许你创建有关 mock 之间的调用顺序的断言:

```
>>> parent = MagicMock()
>>> child1 = MagicMock(return_value=None)
>>> child2 = MagicMock(return_value=None)
>>> parent.child1 = child1
>>> parent.child2 = child2
>>> child1(1)
>>> child2(2)
>>> parent.mock_calls
[call.child1(1), call.child2(2)]
```

这里有一个例外情况是如果 mock 设置了名称。这允许你在出于某些理由不希望其发生时避免” 父对象” 的影响。

```
>>> mock = MagicMock()
>>> not_a_child = MagicMock(name='not-a-child')
>>> mock.attribute = not_a_child
>>> mock.attribute()
<MagicMock name='not-a-child()' id='...'>
>>> mock.mock_calls
[]
```

通过*patch()* 创建的 mock 会被自动赋予名称。要将具有名称的 mock 附加到父对象上你应当使用*attach_mock()* 方法:

```
>>> thing1 = object()
>>> thing2 = object()
>>> parent = MagicMock()
>>> with patch('__main__.thing1', return_value=None) as child1:
...     with patch('__main__.thing2', return_value=None) as child2:
...         parent.attach_mock(child1, 'child1')
...         parent.attach_mock(child2, 'child2')
...         child1('one')
...         child2('two')
...
>>> parent.mock_calls
[call.child1('one'), call.child2('two')]
```

26.9.3 patch 装饰器

patch 装饰器仅被用于在它们所装饰的函数作用域内部为对象添加补丁。它们会自动为你执行去除补丁的处理，即使是在引发了异常的情况下。所有这些函数都还可在 with 语句中使用或是作为类装饰器。

patch

備F: 问题的关键是要在正确的命名空间中打补丁。参见*where to patch* 一节。

---

unittest.mock.**patch**(*target*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch() 可以作为函数装饰器、类装饰器或上下文管理器。在函数或 with 语句的内部，*target* 会打上一个 *new* 对象补丁。当函数/with 语句退出时补丁将被撤销。

如果 *new* 被省略，那么如果被打补丁的对象是一个异步函数则 target 将被替换为*AsyncMock* 否则替换为*MagicMock*。如果*patch()* 被用作装饰器并且 *new* 被省略，那么已创建的 mock 将作为一个附加参数传入被装饰的函数。如果*patch()* 被用作上下文管理器那么已创建的 mock 将被该上下文管理器所返回。is returned by the context manager.

target 应当为 `'package.module.ClassName'` 形式的字符串。*target* 将被导入并且该指定对象会被替换为 *new* 对象，因此 *target* 必须是可以从你调用*patch()* 的环境中导入的。target 会在被装饰的函数被执行的时候被导入，而非在装饰的时候。

spec 和 *spec_set* 关键字参数会被传递给*MagicMock*，如果 patch 为你创建了此对象的话。

此外你还可以传入 `spec=True` 或 `spec_set=True`，这将使 patch 将被模拟的对象作为 spec/spec_set 对象传入。

new_callable 允许你指定一个不同的类，或者可调用对象，它将被调用以创建 *新的*对象。在默认情况下将指定*AsyncMock* 用于异步函数，*MagicMock* 用于其他函数。

另一种更强形式的 *spec* 是 *autospec*。如果你设置了 `autospec=True` 则将以来自被替换对象的 spec 来创建 mock。mock 的所有属性也将具有被替换对象相应属性的 spec。被模拟的方法和函数将检查它们的参数并且如果使用了错误的签名调用它们则将引发 *TypeError*。对于替换了一个类的 mock，它们的返回值（即‘实例’）将具有与该类相同的 spec。请参阅 *create_autospec()* 函数以及 *自动 spec*。

除了 `autospec=True` 你还可以传入 `autospec=some_object` 以使用任意对象而不是被替换的对象作为 spec。

在默认情况下 *patch()* 将无法替换不存在的属性。如果你传入 `create=True`，且该属性并不存在，则 patch 将在调用被打补丁的函数时为你创建该属性，并在退出被打补丁的函数时再次删除它。这适用于编写针对生产代码在运行时创建的属性的测试。它默认是被关闭的因为这具有危险性。当它被开启时你将能够针对实际上并不存在的 API 编写通过测试!

備𝔽: 3.5 版更變: 如果你要给某个模块的内置函数打补丁则不必传入 `create=True`，它默认就会被添加。

Patch 可以被用作 `TestCase` 类装饰器。其作用是装饰类中的每个测试方法。当你的测试方法共享同一个补丁集时这将减少模板代码。*patch()* 会通过查找以 `patch.TEST_PREFIX` 打头的名称来找到测试。其默认值为 `'test'`，这与 *unittest* 打到测试的方式一致。你可以通过设置 `patch.TEST_PREFIX` 来指定其他的前缀。

Patch 可以通过 with 语句作为上下文管理器使用。这时补丁将应用于 with 语句的缩进代码块。如果你使用了”as” 则打补丁的对象将被绑定到”as” 之后的名称；这非常适用于当 *patch()* 为你创建 mock 对象的情况。

patch() 可接受任意关键字参数。如果打补丁的对象是异步的则这些参数将被传给 *AsyncMock*，否则传给 *MagicMock*，或者是指定的 *new_callable*。

`patch.dict(...)`,`patch.multiple(...)` 和 `patch.object(...)` 可用于其他使用场景。

patch() 作为函数装饰器，为你创建 mock 并将其传入被装饰的函数:

```
>>> @patch('__main__.SomeClass')
... def function(normal_argument, mock_class):
...     print(mock_class is SomeClass)
...
>>> function(None)
True
```

为类打补丁将把该类替换为 *MagicMock* 的 *实例*。如果该类是在受测试的代码中被实例化的则它将为所要使用的 mock 的 *return_value*。

如果该类被多次实例化则你可以使用 *side_effect* 来每次返回一个新 mock。或者你也可以将 *return_value* 设为你希望的任何对象。

要在被打补丁的类的 *实例* 的方法上配置返回值你必须在 `return_value` 操作。例如:

```
>>> class Class:
...     def method(self):
...         pass
...
>>> with patch('__main__.Class') as MockClass:
...     instance = MockClass.return_value
...     instance.method.return_value = 'foo'
...     assert Class() is instance
...     assert Class().method() == 'foo'
...
```

如果你使用 *spec* 或 *spec_set* 并且 *patch()* 替换的是 *class*，那么所创建的 mock 的返回值将具有同样的 spec。

```
>>> Original = Class
>>> patcher = patch('__main__.Class', spec=True)
>>> MockClass = patcher.start()
>>> instance = MockClass()
>>> assert isinstance(instance, Original)
>>> patcher.stop()
```

new_callable 参数适用于当你想要使用其他类来替代所创建的 mock 默认的*MagicMock* 的场合。例如，如果你想要使用*NonCallableMock*:

```
>>> thing = object()
>>> with patch('__main__.thing', new_callable=NonCallableMock) as mock_thing:
...     assert thing is mock_thing
...     thing()
...
Traceback (most recent call last):
  ...
TypeError: 'NonCallableMock' object is not callable
```

另一个使用场景是用*io.StringIO* 实例来替换某个对象:

```
>>> from io import StringIO
>>> def foo():
...     print('Something')
...
>>> @patch('sys.stdout', new_callable=StringIO)
... def test(mock_stdout):
...     foo()
...     assert mock_stdout.getvalue() == 'Something\n'
...
>>> test()
```

当*patch()* 为你创建 mock 时，通常你需要做的第一件事就是配置该 mock。某些配置可以在对 patch 的调用中完成。你在调用时传入的任何关键字参数都将被用来在所创建的 mock 上设置属性:

```
>>> patcher = patch('__main__.thing', first='one', second='two')
>>> mock_thing = patcher.start()
>>> mock_thing.first
'one'
>>> mock_thing.second
'two'
```

除了所创建的 mock 的属性上的属性，例如*return_value* 和*side_effect*，还可以配置子 mock 的属性。将这些属性直接作为关键字参数传入在语义上是无效的，但是仍然能够使用 ** 将以这些属性为键的字典扩展至一个*patch()* 调用中

```
>>> config = {'method.return_value': 3, 'other.side_effect': KeyError}
>>> patcher = patch('__main__.thing', **config)
>>> mock_thing = patcher.start()
>>> mock_thing.method()
3
>>> mock_thing.other()
Traceback (most recent call last):
  ...
KeyError
```

在默认情况下，尝试给某个模块中并不存在的函数（或者某个类中的方法或属性）打补丁将会失败并引发*AttributeError*:

```
>>> @patch('sys.non_existing_attribute', 42)
... def test():
...     assert sys.non_existing_attribute == 42
...
>>> test()
Traceback (most recent call last):
  ...
AttributeError: <module 'sys' (built-in)> does not have the attribute 'non_existing_
↪attribute'
```

但在对*patch()* 的调用中添加 `create=True` 将使之前示例的效果符合预期:

```
>>> @patch('sys.non_existing_attribute', 42, create=True)
... def test(mock_stdout):
...     assert sys.non_existing_attribute == 42
...
>>> test()
```

3.8 版更變: 如果目标为异步函数那么*patch()* 现在将返回一个*AsyncMock*。

patch.object

patch.**object**(*target*, *attribute*, *new=DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

用一个 mock 对象为对象 (*target*) 中指定名称的成员 (*attribute*) 打补丁。

patch.object() 可以被用作装饰器、类装饰器或上下文管理器。*new*, *spec*, *create*, *spec_set*, *autospec* 和 *new_callable* 等参数的含义与*patch()* 的相同。与*patch()* 类似，*patch.object()* 接受任意关键字参数用于配置它所创建的 mock 对象。

当用作类装饰器时*patch.object()* 将认可 `patch.TEST_PREFIX` 作为选择所要包装方法的标准。

你可以附带三个参数或两个参数来调用*patch.object()*。三个参数的形式将接受要打补丁的对象、属性的名称以及将要替换该属性的对象。

将附带两个参数的形式调用时你将省略替换对象，还会为你创建一个 mock 并作为附加参数传入被装饰的函数:

```
>>> @patch.object(SomeClass, 'class_method')
... def test(mock_method):
...     SomeClass.class_method(3)
...     mock_method.assert_called_with(3)
...
>>> test()
```

传给*patch.object()* 的 *spec*, *create* 和其他参数的含义与*patch()* 的同名参数相同。

patch.dict

patch.**dict**(*in_dict*, *values=()*, *clear=False*, ***kwargs*)

为一个目录或目录类对象打补丁，并在测试之后将该目录恢复到其初始状态。

in_dict 可以是一个字典或映射类容器。如果它是一个映射则它必须至少支持获取、设置和删除条目以及对键执行迭代。

in_dict 也可以是一个指定字典名称的字符串，然后将通过导入操作来获取该字典。

values 可以是一个要在字典中设置的值的字典。*values* 也可以是一个包含 `(key, value)` 对的可迭代对象。

如果 *clear* 为真值则该字典将在设置新值之前先被清空。

patch.dict() 也可以附带任意关键字参数调用以设置字典中的值。

3.8 版更變: 现在当*patch.dict()* 被用作上下文管理器时将返回被打补丁的字典。now returns the patched dictionary when used as a context manager.

patch.dict() 可被用作上下文管理器、装饰器或类装饰器:

```
>>> foo = {}
>>> @patch.dict(foo, {'newkey': 'newvalue'})
... def test():
...     assert foo == {'newkey': 'newvalue'}
>>> test()
>>> assert foo == {}
```

当被用作类装饰器时*patch.dict()* 将认可 `patch.TEST_PREFIX`(默认值为 `'test'`)作为选择所在包装方法的标准:

```
>>> import os
>>> import unittest
>>> from unittest.mock import patch
>>> @patch.dict('os.environ', {'newkey': 'newvalue'})
... class TestSample(unittest.TestCase):
...     def test_sample(self):
...         self.assertEqual(os.environ['newkey'], 'newvalue')
```

如果你在为你的测试使用不同的前缀，你可以通过设置 `patch.TEST_PREFIX` 来将不同的前缀告知打补丁方。有关如何修改该值的详情请参阅*TEST_PREFIX*。

patch.dict() 可被用来向一个字典添加成员，或者简单地让测试修改一个字典，并确保当测试结束时恢复该字典。

```
>>> foo = {}
>>> with patch.dict(foo, {'newkey': 'newvalue'}) as patched_foo:
...     assert foo == {'newkey': 'newvalue'}
...     assert patched_foo == {'newkey': 'newvalue'}
...     # You can add, update or delete keys of foo (or patched_foo, it's the same␣
↪dict)
...     patched_foo['spam'] = 'eggs'
...
>>> assert foo == {}
>>> assert patched_foo == {}
```

```
>>> import os
>>> with patch.dict('os.environ', {'newkey': 'newvalue'}):
```

(下页继续)

(繼續上一頁)

```
...     print(os.environ['newkey'])
...
newvalue
>>> assert 'newkey' not in os.environ
```

可以在*patch.dict()* 调用中使用关键字来设置字典的值:

```
>>> mymodule = MagicMock()
>>> mymodule.function.return_value = 'fish'
>>> with patch.dict('sys.modules', mymodule=mymodule):
...     import mymodule
...     mymodule.function('some', 'args')
...
'fish'
```

patch.dict() 可以用于实际上不是字典的字典类对象。它们最少必须支持条目获取、设置、删除以及迭代或成员检测两者中的一个。这对应于魔术方法 `__getitem__()`, `__setitem__()`, `__delitem__()` 以及 `__iter__()` 或 `__contains__()`。

```
>>> class Container:
...     def __init__(self):
...         self.values = {}
...     def __getitem__(self, name):
...         return self.values[name]
...     def __setitem__(self, name, value):
...         self.values[name] = value
...     def __delitem__(self, name):
...         del self.values[name]
...     def __iter__(self):
...         return iter(self.values)
...
>>> thing = Container()
>>> thing['one'] = 1
>>> with patch.dict(thing, one=2, two=3):
...     assert thing['one'] == 2
...     assert thing['two'] == 3
...
>>> assert thing['one'] == 1
>>> assert list(thing) == ['one']
```

patch.multiple

patch.**multiple**(*target*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

在单个调用中执行多重补丁。它接受要打补丁的对象（一个对象或一个通过导入来获取对象的字符串）以及用于补丁的关键字参数:

```
with patch.multiple(settings, FIRST_PATCH='one', SECOND_PATCH='two'):
    ...
```

如果你希望*patch.multiple()* 为你创建 mock 则要使用*DEFAULT* 作为值。在此情况下所创建的 mock 会通过关键字参数传入被装饰的函数，而当*patch.multiple()* 被用作上下文管理器时则将返回一个字典。

patch.multiple() 可以被用作装饰器、类装饰器或上下文管理器。*spec*, *spec_set*, *create*, *autospec* 和

new_callable 等参数的含义与*patch()* 的相同。这些参数将被应用到*patch.multiple()* 所打的 所有补丁。

当被用作类装饰器时*patch.multiple()* 将认可 patch.TEST_PREFIX 作为选择所要包装方法的标准。

如果你希望*patch.multiple()* 为你创建 mock，那么你可以使用*DEFAULT* 作为值。如果你使用*patch.multiple()* 作为装饰器则所创建的 mock 会作为关键字参数传入被装饰的函数。

```
>>> thing = object()
>>> other = object()

>>> @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(thing, other):
...     assert isinstance(thing, MagicMock)
...     assert isinstance(other, MagicMock)
...
>>> test_function()
```

patch.multiple() 可以与其他 patch 装饰器嵌套使用，但要将作为关键字传入的参数要放在*patch()* 所创建的标准参数 之后:

```
>>> @patch('sys.exit')
... @patch.multiple('__main__', thing=DEFAULT, other=DEFAULT)
... def test_function(mock_exit, other, thing):
...     assert 'other' in repr(other)
...     assert 'thing' in repr(thing)
...     assert 'exit' in repr(mock_exit)
...
>>> test_function()
```

如果*patch.multiple()* 被用作上下文管理器，则上下文管理器的返回值将是一个以所创建的 mock 的名称为键的字典:

```
>>> with patch.multiple('__main__', thing=DEFAULT, other=DEFAULT) as values:
...     assert 'other' in repr(values['other'])
...     assert 'thing' in repr(values['thing'])
...     assert values['thing'] is thing
...     assert values['other'] is other
...
```

补丁方法: start 和 stop

所有补丁对象都具有 start() 和 stop() 方法。使用这些方法可以更简单地在 setUp 方法上打补丁，还可以让你不必嵌套使用装饰器或 with 语句就能打多重补丁。

要使用这些方法请按正常方式调用*patch()*, *patch.object()* 或*patch.dict()* 并保留一个指向所返回 patcher 对象的引用。然后你可以调用 start() 来原地打补丁并调用 stop() 来恢复它。

如果你使用*patch()* 来创建自己的 mock 那么将可通过调用 patcher.start 来返回它。

```
>>> patcher = patch('package.module.ClassName')
>>> from package import module
>>> original = module.ClassName
>>> new_mock = patcher.start()
>>> assert module.ClassName is not original
>>> assert module.ClassName is new_mock
```

(下页继续)

(繼續上一頁)

```
>>> patcher.stop()
>>> assert module.ClassName is original
>>> assert module.ClassName is not new_mock
```

此操作的一个典型应用场景是在一个 TestCase 的 setUp 方法中执行多重补丁:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher1 = patch('package.module.Class1')
...         self.patcher2 = patch('package.module.Class2')
...         self.MockClass1 = self.patcher1.start()
...         self.MockClass2 = self.patcher2.start()
...
...     def tearDown(self):
...         self.patcher1.stop()
...         self.patcher2.stop()
...
...     def test_something(self):
...         assert package.module.Class1 is self.MockClass1
...         assert package.module.Class2 is self.MockClass2
...
>>> MyTest('test_something').run()
```

警示: 如果你要使用这个技巧则你必须通过调用 stop 来确保补丁被“恢复”。这可能要比你想像的更麻烦，因为如果在 setUp 中引发了异常那么 tearDown 将不会被调用。*unittest.TestCase.addCleanup()* 可以简化此操作:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('package.module.Class')
...         self.MockClass = patcher.start()
...         self.addCleanup(patcher.stop)
...
...     def test_something(self):
...         assert package.module.Class is self.MockClass
...
```

一项额外的好处是你不再需要保留指向 patcher 对象的引用。

还可以通过使用*patch.stopall()* 来停止已启动的所有补丁。

patch.**stopall**()
停止所有激活的补丁。仅会停止通过 start 启动的补丁。

为内置对象打补丁

你可以为一个模块中的任何内置函数打补丁。以以示例是为内置函数*ord()* 打补丁:

```
>>> @patch('__main__.ord')
... def test(mock_ord):
...     mock_ord.return_value = 101
...     print(ord('c'))
...
>>> test()
101
```

TEST_PREFIX

所有补丁都可被用作类装饰器。当以这种方式使用时它们将会包装类中的每个测试方法。补丁会将以名字以 'test' 开头的方法识别为测试方法。这与*unittest.TestLoader* 查找测试方法的默认方式相同。

你可能会想要为你的测试使用不同的前缀。你可以通过设置 patch.TEST_PREFIX 来告知打补丁方不同的前缀:

```
>>> patch.TEST_PREFIX = 'foo'
>>> value = 3
>>>
>>> @patch('__main__.value', 'not three')
... class Thing:
...     def foo_one(self):
...         print(value)
...     def foo_two(self):
...         print(value)
...
>>>
>>> Thing().foo_one()
not three
>>> Thing().foo_two()
not three
>>> value
3
```

嵌套补丁装饰器

如果你想要应用多重补丁那么你可以简单地堆叠多个装饰器。

你可以使用以下模式来堆叠多个补丁装饰器:

```
>>> @patch.object(SomeClass, 'class_method')
... @patch.object(SomeClass, 'static_method')
... def test(mock1, mock2):
...     assert SomeClass.static_method is mock1
...     assert SomeClass.class_method is mock2
...     SomeClass.static_method('foo')
...     SomeClass.class_method('bar')
...     return mock1, mock2
...
>>> mock1, mock2 = test()
>>> mock1.assert_called_once_with('foo')
>>> mock2.assert_called_once_with('bar')
```

请注意装饰器是从下往上被应用的。这是 Python 应用装饰器的标准方式。被创建并传入你的测试函数的 mock 的顺序也将匹配这个顺序。

补丁的位置

patch() 通过（临时性地）修改某一个对象的 名称指向另一个对象来发挥作用。可以有多个名称指向任意单独对象，因此要让补丁起作用你必须确保已为被测试的系统所使用的名称打上补丁。

基本原则是你要在对象 被查找的地方打补丁，这不一定就是它被定义的地方。一组示例将有助于厘清这一点。

想像我们有一个想要测试的具有如下结构的项目:

```
a.py
    -> Defines SomeClass

b.py
    -> from a import SomeClass
    -> some_function instantiates SomeClass
```

现在我们要测试 some_function 但我们想使用*patch()* 来模拟 SomeClass。问题在于当我们导入模块 b 时，我们将必须让它从模块 a 导入 SomeClass。如果我们使用*patch()* 来模拟 a.SomeClass 那么它将不会对我们的测试造成影响；模块 b 已经拥有对 真正的 SomeClass 的引用因此看上去我们的补丁不会有任何影响。

关键在于对 SomeClass 打补丁操作是在它被使用（或它被查找）的地方。在此情况下实际上 some_function 将在模块 b 中查找 SomeClass，而我们已经在那里导入了它。补丁看上去应该是这样:

```
@patch('b.SomeClass')
```

但是，再考虑另一个场景，其中不是 from a import SomeClass 而是模块 b 执行了 import a 并且 some_function 使用了 a.SomeClass。这两个导入形式都很常见。在这种情况下我们要打补丁的类将在该模块中被查找因而我们必须改为对 a.SomeClass 打补丁:

```
@patch('a.SomeClass')
```

对描述器和代理对象打补丁

patch 和*patch.object* 都能正确地对描述器打补丁并恢复：包括类方法、静态方法和特征属性。你应当在 类而不是在实例上为它们打补丁。它们还适用于代理属性访问的 某些对象，例如 django 设置对象。

26.9.4 MagicMock 与魔术方法支持

模拟魔术方法

Mock 支持模拟 Python 协议方法，或称 “魔术方法”。这允许 mock 对象替代容器或其他实现了 Python 协议的对象。

因为查找魔术方法的方式不同于普通方法[2]，这种支持采用了特别的实现。这意味着只有特定的魔术方法受到支持。受支持的是 几乎所有魔术方法。如果有你需要但被遗漏的请告知我们。

[2] 魔术方法 应当是在类中而不是在实例中查找。不同的 Python 版本对这个规则的应用并不一致。受支持的协议方法应当适用于所有受支持的 Python 版本。

你可以通过在某个函数或 mock 实例中设置魔术方法来模拟它们。如果你是使用函数则它 必须接受 `self` 作为第一个参数[3]。

```
>>> def __str__(self):
...     return 'fooble'
...
>>> mock = Mock()
>>> mock.__str__ = __str__
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__str__ = Mock()
>>> mock.__str__.return_value = 'fooble'
>>> str(mock)
'fooble'
```

```
>>> mock = Mock()
>>> mock.__iter__ = Mock(return_value=iter([]))
>>> list(mock)
[]
```

一个这样的应用场景是在 `with` 语句中模拟作为上下文管理器的对象:

```
>>> mock = Mock()
>>> mock.__enter__ = Mock(return_value='foo')
>>> mock.__exit__ = Mock(return_value=False)
>>> with mock as m:
...     assert m == 'foo'
...
>>> mock.__enter__.assert_called_with()
>>> mock.__exit__.assert_called_with(None, None, None)
```

对魔术方法的调用不会在*method_calls* 中出现，但它们会被记录在*mock_calls* 中。

備F: 如果你使用 *spec* 关键字参数来创建 mock 那么尝试设置不包含在 spec 中的魔术方法将引发*AttributeError*。

受支持魔术方法的完整列表如下:

- `__hash__`, `__sizeof__`, `__repr__` 和 `__str__`
- `__dir__`, `__format__` 和 `__subclasses__`
- `__round__`, `__floor__`, `__trunc__` 和 `__ceil__`
- 比较运算: `__lt__`, `__gt__`, `__le__`, `__ge__`, `__eq__` 和 `__ne__`
- 容器方法: `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`, `__len__`, `__iter__`, `__reversed__` 和 `__missing__`
- 上下文管理器: `__enter__`, `__exit__`, `__aenter__` 和 `__aexit__`
- 单目数值运算方法: `__neg__`, `__pos__` 和 `__invert__`
- 数值运算方法 (包括右手和原地形式): `__add__`, `__sub__`, `__mul__`, `__matmul__`, `__div__`, `__truediv__`, `__floordiv__`, `__mod__`, `__divmod__`, `__lshift__`, `__rshift__`, `__and__`, `__xor__`, `__or__` 和 `__pow__`

[3] 该函数基本上是与类挂钩的，但每个 `Mock` 实例都会与其他实例保持隔离。

- 数值转换方法: __complex__, __int__, __float__ 和 __index__
- 描述器方法: __get__, __set__ and __delete__
- 封存方法: __reduce__, __reduce_ex__, __getinitargs__, __getnewargs__, __getstate__ 和 __setstate__
- 文件系统路径表示: __fspath__
- 异步迭代方法: __aiter__ and __anext__

3.8 版更變: 增加了对*os.PathLike.__fspath__()* 的支持。

3.8 版更變: 增加了对 __aenter__, __aexit__, __aiter__ 和 __anext__ 的支持。

下列方法均存在但是 不受支持，因为它们或者被 mock 所使用，或者无法动态设置，或者可能导致问题:

- __getattr__, __setattr__, __init__ 和 __new__
- __prepare__, __instancecheck__, __subclasscheck__, __del__

MagicMock

存在两个版本的 MagicMock: *MagicMock* 和*NonCallableMagicMock*.

class unittest.mock.**MagicMock**(**args*, ***kw*)

MagicMock 是包含了大部分魔术方法的默认实现的*Mock* 的子类。你可以使用 MagicMock 而无须自行配置魔术方法。without having to configure the magic methods yourself.

构造器形参的含义与*Mock* 的相同。

如果你使用了 *spec* 或 *spec_set* 参数则将 只有存在于 spec 中的魔术方法会被创建。

class unittest.mock.**NonCallableMagicMock**(**args*, ***kw*)

MagicMock 的不可调用对象版本。

其构造器的形参具有与*MagicMock* 相同的含义，区别在于 *return_value* 和 *side_effect* 在不可调用的 mock 上没有意义。

魔术方法是通过*MagicMock* 对象来设置的，因此你可以用通常的方式来配置它们并使用它们:

```
>>> mock = MagicMock()
>>> mock[3] = 'fish'
>>> mock.__setitem__.assert_called_with(3, 'fish')
>>> mock.__getitem__.return_value = 'result'
>>> mock[2]
'result'
```

在默认情况下许多协议方法都需要返回特定类型的对象。这些方法都预先配置了默认的返回值，以便它们在你对返回值不感兴趣时可以不做任何事就能被使用。如果你想要修改默认值则你仍然可以手动 设置返回值。

方法及其默认返回值:

- __lt__: NotImplemented
- __gt__: NotImplemented
- __le__: NotImplemented
- __ge__: NotImplemented
- __int__: 1
- __contains__: False

- __len__: 0
- __iter__: iter([])
- __exit__: False
- __aexit__: False
- __complex__: 1j
- __float__: 1.0
- __bool__: True
- __index__: 1
- __hash__: mock 的默认 hash
- __str__: mock 的默认 str
- __sizeof__: mock 的默认 sizeof

例如:

```
>>> mock = MagicMock()
>>> int(mock)
1
>>> len(mock)
0
>>> list(mock)
[]
>>> object() in mock
False
```

两个相等性方法 __eq__() 和 __ne__() 是特殊的。它们基于标识号进行默认的相等性比较，使用*side_effect* 属性，除非你修改它们的返回值以返回其他内容:

```
>>> MagicMock() == 3
False
>>> MagicMock() != 3
True
>>> mock = MagicMock()
>>> mock.__eq__.return_value = True
>>> mock == 3
True
```

MagicMock.__iter__() 的返回值可以是任意可迭代对象而不要求必须是迭代器:

```
>>> mock = MagicMock()
>>> mock.__iter__.return_value = ['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
['a', 'b', 'c']
```

如果返回值 是迭代器，则对其执行一次迭代就会将它耗尽因而后续执行的迭代将会输出空列表:

```
>>> mock.__iter__.return_value = iter(['a', 'b', 'c'])
>>> list(mock)
['a', 'b', 'c']
>>> list(mock)
[]
```

MagicMock 已配置了所有受支持的魔术方法，只有某些晦涩和过时的魔术方法是例外。如果你需要仍然可以设置它们。

在 MagicMock 中受到支持但默认未被设置的魔术方法有:

- __subclasses__
- __dir__
- __format__
- __get__, __set__ 和 __delete__
- __reversed__ 和 __missing__
- __reduce__, __reduce_ex__, __getinitargs__, __getnewargs__, __getstate__ 和 __setstate__
- __getformat__ 和 __setformat__

26.9.5 辅助对象

sentinel

unittest.mock.**sentinel**
: sentinel 对象提供了一种为你的测试提供独特对象的便捷方式。

  属性是在你通过名称访问它们时按需创建的。访问相同的属性将始终返回相同的对象。返回的对象会有一个合理的 repr 以使测试失败消息易于理解。

  3.7 版更變: 现在 sentinel 属性会在它们被*copy* 或*pickle* 时保存其标识。

在测试时你可能需要测试是否有一个特定的对象作为参数被传给了另一个方法，或是被其返回。通常的做法是创建一个指定名称的 sentinel 对象来执行这种测试。*sentinel* 提供了一种创建和测试此类对象的标识的便捷方式。

在这个示例中我们为 method 打上便捷补丁以返回 sentinel.some_object:

```
>>> real = ProductionClass()
>>> real.method = Mock(name="method")
>>> real.method.return_value = sentinel.some_object
>>> result = real.method()
>>> assert result is sentinel.some_object
>>> result
sentinel.some_object
```

DEFAULT

unittest.mock.**DEFAULT**
: *DEFAULT* 对象是一个预先创建的 sentinel (实际为 sentinel.DEFAULT)。它可被*side_effect* 函数用来指明其应当使用正常的返回值。

call

unittest.mock.**call**(**args*, ***kwargs*)

call() 是一个可创建更简单断言的辅助对象，用于同*call_args*, *call_args_list*, *mock_calls* 和*method_calls* 进行比较。*call()* 也可配合*assert_has_calls()* 使用。

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, a='foo', b='bar')
>>> m()
>>> m.call_args_list == [call(1, 2, a='foo', b='bar'), call()]
True
```

call.**call_list**()

对于代表多个调用的 call 对象，*call_list()* 将返回一个包含所有中间调用以及最终调用的列表。

call_list 特别适用于创建针对“链式调用”的断言。链式调用是指在一行代码中执行的多个调用。这将使得一个 mock 的中存在多个条目*mock_calls*。手动构造调用的序列将会很烦琐。

call_list() 可以根据同一个链式调用构造包含多个调用的序列:

```
>>> m = MagicMock()
>>> m(1).method(arg='foo').other('bar')(2.0)
<MagicMock name='mock().method().other()()' id='...'>
>>> kall = call(1).method(arg='foo').other('bar')(2.0)
>>> kall.call_list()
[call(1),
 call().method(arg='foo'),
 call().method().other('bar'),
 call().method().other()(2.0)]
>>> m.mock_calls == kall.call_list()
True
```

根据构造方式的不同，call 对象可以是一个 (位置参数, 关键字参数) 或 (名称, 位置参数, 关键字参数) 元组。当你自行构造它们时这没有什么关系，但是*Mock.call_args*, *Mock.call_args_list* 和*Mock.mock_calls* 属性当中的 call 对象可以被反查以获取它们所包含的单个参数。

Mock.call_args 和*Mock.call_args_list* 中的 call 对象是 (位置参数, 关键字参数) 二元组而*Mock.mock_calls* 中以及你自行构造的 call 对象则是 (名称, 位置参数, 关键字参数) 三元组。

你可以使用它们作为“元组”的特性来为更复杂的内省和断言功能获取单个参数。位置参数是一个元组（如无位置参数则为空元组）而关键字参数是一个字典:

```
>>> m = MagicMock(return_value=None)
>>> m(1, 2, 3, arg='one', arg2='two')
>>> kall = m.call_args
>>> kall.args
(1, 2, 3)
>>> kall.kwargs
{'arg': 'one', 'arg2': 'two'}
>>> kall.args is kall[0]
True
>>> kall.kwargs is kall[1]
True
```

```
>>> m = MagicMock()
>>> m.foo(4, 5, 6, arg='two', arg2='three')
<MagicMock name='mock.foo()' id='...'>
>>> kall = m.mock_calls[0]
```

(下页继续)

(繼續上一頁)

```
>>> name, args, kwargs = kall
>>> name
'foo'
>>> args
(4, 5, 6)
>>> kwargs
{'arg': 'two', 'arg2': 'three'}
>>> name is m.mock_calls[0][0]
True
```

create_autospec

unittest.mock.**create_autospec**(*spec*, *spec_set=False*, *instance=False*, ***kwargs*)

使用另一对象作为 spec 来创建 mock 对象。mock 的属性将使用 *spec* 对象上的对应属性作为其 spec。

被模拟的函数或方法的参数将会被检查以确保它们是附带了正确的签名被调用的。

如果 *spec_set* 为 `True` 则尝试设置不存在于 spec 对象中的属性将引发*AttributeError*。

如果将类用作 spec 则 mock（该类的实例）的返回值将为这个 spec。你可以通过传入 `instance=True` 来将某个类用作一个实例对象的 spec。被返回的 mock 将仅在该 mock 的实例为可调用对象时才会是可调用对象。

create_autospec() 也接受被传入所创建 mock 的构造器的任意关键字参数。

请参阅*自动 spec* 来了解如何通过*create_autospec()* 以及*patch()* 的 *autospec* 参数来自动设置 spec。

3.8 版更變: 如果目标为异步函数那么*create_autospec()* 现在将返回一个*AsyncMock*。

ANY

unittest.mock.**ANY**

有时你可能需要设置关于要模拟的调用中的某些参数的断言，但是又不想理会其他参数或者想要从*call_args* 中单独拿出它们并针对它们设置更复杂的断言。

为了忽略某些参数你可以传入与任意对象相等的对象。这样再调用*assert_called_with()* 和*assert_called_once_with()* 时无论传入什么参数都将执行成功。

```
>>> mock = Mock(return_value=None)
>>> mock('foo', bar=object())
>>> mock.assert_called_once_with('foo', bar=ANY)
```

ANY 也可被用在与*mock_calls* 这样的调用列表相比较的场合:

```
>>> m = MagicMock(return_value=None)
>>> m(1)
>>> m(1, 2)
>>> m(object())
>>> m.mock_calls == [call(1), call(1, 2), ANY]
True
```

FILTER_DIR

unittest.mock.**FILTER_DIR**

FILTER_DIR 是一个控制 mock 对象响应*dir()* 的方式的模块级变量（只适用于 Python 2.6 之后的版本）。默认值为 True，这将使用下文描述的过滤操作，以便只显示有用的成员。如果你不想要这样的过滤，或是出于诊断目的需要将其关闭，则应设置 mock.FILTER_DIR = False。

当启用过滤时，dir(some_mock) 将只显示有用的属性并将包括正常情况下不会被显示的任何动态创建的属性。如果 mock 是使用 *spec* (当然也可以是 *autospec*) 创建的则原有的所有属性都将被显示，即使它们还未被访问过:

```
>>> dir(Mock())
['assert_any_call',
 'assert_called',
 'assert_called_once',
 'assert_called_once_with',
 'assert_called_with',
 'assert_has_calls',
 'assert_not_called',
 'attach_mock',
 ...
>>> from urllib import request
>>> dir(Mock(spec=request))
['AbstractBasicAuthHandler',
 'AbstractDigestAuthHandler',
 'AbstractHTTPHandler',
 'BaseHandler',
 ...
```

许多不太有用的（是*Mock* 的而不是被模拟对象的私有成员）开头带有下划线和双下划线的属性已从在*Mock* 上对*dir()* 的调用的结果中被过滤。如果你不喜欢此行为你可以通过设置模块级开关*FILTER_DIR* 来将其关闭:

```
>>> from unittest import mock
>>> mock.FILTER_DIR = False
>>> dir(mock.Mock())
['_NonCallableMock__get_return_value',
 '_NonCallableMock__get_side_effect',
 '_NonCallableMock__return_value_doc',
 '_NonCallableMock__set_return_value',
 '_NonCallableMock__set_side_effect',
 '__call__',
 '__class__',
 ...
```

或者你也可以直接使用 vars(my_mock) (实例成员) 和 dir(type(my_mock)) (类型成员) 来忽略 mock.FILTER_DIR 设置绕过过滤。

mock_open

unittest.mock.**mock_open**(*mock=None*, *read_data=None*)

创建 mock 来代替使用*open()* 的辅助函数。它对于*open()* 被直接调用或被用作上下文管理器的情况都适用。

mock 参数是要配置的 mock 对象。如为 None (默认值) 则将为你创建一个*MagicMock*，其 API 会被限制为可用于标准文件处理的方法或属性。

read_data 是供文件处理的 read()，*readline()* 和*readlines()* 方法返回的字符串。调用这些方法将会从 *read_data* 获取数据直到它被耗尽。对这些方法的模拟是相当简化的：每次 *mock* 被调用时，*read_data* 都将从头开始。如果你需要对你提供给测试代码的数据有更多控制那么你将需要自行定制这个 mock。如果这还不够用，那么通过一些 PyPI 上的内存文件系统包可以提供更真实的测试文件系统。

3.4 版更變: 增加了对*readline()* 和*readlines()* 的支持。对 read() 的模拟被改为消耗 *read_data* 而不是每次调用时返回它。

3.5 版更變: *read_data* 现在会在每次 *mock* 的调用时重置。

3.8 版更變: 为实现增加了 __iter__() 以使迭代操作（例如在 for 循环中）能正确地消耗 *read_data*。

将*open()* 用作上下文管理器一确保你的文件处理被正确关闭的最佳方式因而十分常用:

```
with open('/some/path', 'w') as f:
    f.write('something')
```

这里的问题在于即使你模拟了对*open()* 的调用但被用作上下文管理器（并调用其 __enter__() 和 __exit__() 方法）的也是 被返回的对象。

通过*MagicMock* 来模拟上下文管理器是十分常见且十分麻烦的因此需要使用一个辅助函数。

```
>>> m = mock_open()
>>> with patch('__main__.open', m):
...     with open('foo', 'w') as h:
...         h.write('some stuff')
...
>>> m.mock_calls
[call('foo', 'w'),
 call().__enter__(),
 call().write('some stuff'),
 call().__exit__(None, None, None)]
>>> m.assert_called_once_with('foo', 'w')
>>> handle = m()
>>> handle.write.assert_called_once_with('some stuff')
```

以及针对读取文件:

```
>>> with patch('__main__.open', mock_open(read_data='bibble')) as m:
...     with open('foo') as h:
...         result = h.read()
...
>>> m.assert_called_once_with('foo')
>>> assert result == 'bibble'
```

自动 spec

自动 spec 是基于现有 mock 的 `spec` 特性。它将 mock 的 api 限制为原始对象 (spec) 的 api，但它是递归（惰性实现）的因而 mock 的属性只有与 spec 的属性相同的 api。除此之外被模块的函数 / 方法具有与原对应物相同的调用签名因此如果它们被不正确地调用时会引发*TypeError*。

在我开始解释自动 spec 如何运作之前，先说明一下它的必要性。

Mock 是一个非常强大而灵活的对象，但当它被用来模拟来自被测试系统的对象时有两个缺陷。其中一个缺陷是*Mock* 的 api 特有的而另一个则是使用 mock 对象时普遍存在的问题。

第一个问题为*Mock* 所专属。*Mock* 具有两个特别好用的断言方法: *assert_called_with()* 和*assert_called_once_with()*。

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
>>> mock(1, 2, 3)
>>> mock.assert_called_once_with(1, 2, 3)
Traceback (most recent call last):
  ...
AssertionError: Expected 'mock' to be called once. Called 2 times.
```

因为 mock 会根据需要自动创建属性，并允许你附带任意参数调用它们，所以如果你写错了这两个断言方法则你的断言将会失效:

```
>>> mock = Mock(name='Thing', return_value=None)
>>> mock(1, 2, 3)
>>> mock.assret_called_once_with(4, 5, 6)  # Intentional typo!
```

你的测试将因这个拼写错误而静默并且错误地通过。

第二个问题是 mock 操作中普遍存在的。如果你重构了你的部分代码，例如修改了一些成员的名称等等，则代码中任何仍在使用 *旧 api* 但其使用的是 mock 而非真实对象的测试仍将通过。这意味着即使你的代码已被破坏你的测试却仍可能全部通过。

请注意这是为什么你在单元测试之外还需要集成测试的原因之一。孤立地测试每个部分时全都正常而顺滑，但是如果你没有测试你的各个单元“联成一体”的情况如何那么就仍然存在测试可以发现大量错误的空间。

`mock` 已经提供了一个对此有帮助的特性，称为 spec 控制。如果你使用类或实例作为一个 mock 的 `spec` 那么你将仅能访问 mock 中只存在于实际的类中的属性。

```
>>> from urllib import request
>>> mock = Mock(spec=request.Request)
>>> mock.assret_called_with  # Intentional typo!
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
```

这个 spec 仅会应用于 mock 本身，因此对于 mock 中的任意方法我们仍然面临相同的问题:

```
>>> mock.has_data()
<mock.Mock object at 0x...>
>>> mock.has_data.assret_called_with()  # Intentional typo!
```

自动 spec 解决了这个问题。你可以将 `autospec=True` 传给*patch()* / *patch.object()* 或是使用*create_autospec()* 函数来创建带有 spec 的 mock。如果你是将 `autospec=True` 参数传给*patch()* 那么被替代的那个对象将被用作 spec 对象。因为 spec 控制是“惰性地”执行的（spec 在 mock 中的属性被访

问时才会被创建）所以即使是非常复杂或深度嵌套的对象（例如需要导入本身已导入了多个模块的模块）你也可以使用它而不会有太大的性能损失。

以下是一个实际应用的示例:

```
>>> from urllib import request
>>> patcher = patch('__main__.request', autospec=True)
>>> mock_request = patcher.start()
>>> request is mock_request
True
>>> mock_request.Request
<MagicMock name='request.Request' spec='Request' id='...'>
```

你可以看到 `request.Request` 有一个 spec。`request.Request` 的构造器接受两个参数 (其中一个是 *self*)。如果我们尝试不正确地调用它则将导致:

```
>>> req = request.Request()
Traceback (most recent call last):
  ...
TypeError: <lambda>() takes at least 2 arguments (1 given)
```

该 spec 也将应用于被实例化的类（即附带 i.e. the return value of spec 的 mock 的返回值）:

```
>>> req = request.Request('foo')
>>> req
<NonCallableMagicMock name='request.Request()' spec='Request' id='...'>
```

`Request` 对象是不可调用对象，因此实例化我们的被模拟 `request.Request` 的返回值是一个不可调用的 mock。有了这个 spec 我们的断言中出现任何拼写问题都将引发正确的错误:

```
>>> req.add_header('spam', 'eggs')
<MagicMock name='request.Request().add_header()' id='...'>
>>> req.add_header.assret_called_with  # Intentional typo!
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'assret_called_with'
>>> req.add_header.assert_called_with('spam', 'eggs')
```

在许多情况下你将只需将 In many cases you will just be able to add `autospec=True` 添加到你现有的*patch()* 调用中即可防止拼写错误和 api 变化所导致的问题。

除了通过*patch()* 来使用 *autospec* 还有一个*create_autospec()* 可以直接创建带有自动 spec 的 mock:

```
>>> from urllib import request
>>> mock_request = create_autospec(request)
>>> mock_request.Request('foo', 'bar')
<NonCallableMagicMock name='mock.Request()' spec='Request' id='...'>
```

不过这并非没有缺点和限制，这也就是为什么它不是默认行为。为了知道在 spec 对象上有哪些属性是可用的，autospec 必须对 spec 进行自省（访问其属性）。当你遍历 mock 上的属性时在原始对象上的对应遍历也将在底层进行。如果你的任何带 spec 的对象具有可触发代码执行的特征属性或描述器则你可能会无法使用 autospec。在另一方面更好的的做法是将你的对象设计为可以安全地执行自省[4]。

一个更严重的问题在于实例属性往往是在 `__init__()` 方法中被创建而在类中完全不存在。*autospec* 无法获取动态创建的属性而使得 api 被限制于可见的属性。

[4] 这仅适用于类或已实例化的对象。调用一个被模拟的类来创建一个 mock 实例 不会创建真的实例。只有属性查找——以及对*dir()* 的调用——会被执行。

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

要解决这个问题有几种不同的方式。最容易但多少有些烦扰的方式是简单地在 mock 创建完成后再设置所需的属性。Just because *autospec* 只是不允许你获取不存在于 spec 上的属性但并不会阻止你设置它们:

```
>>> with patch('__main__.Something', autospec=True):
...   thing = Something()
...   thing.a = 33
...
```

spec 和 *autospec* 都有更严格的版本 确实能阻止你设置不存在的属性。这在你希望确保你的代码只能 设置有效的属性时也很有用，但显然它会阻止下面这个特定的应用场景:

```
>>> with patch('__main__.Something', autospec=True, spec_set=True):
...   thing = Something()
...   thing.a = 33
...
Traceback (most recent call last):
  ...
AttributeError: Mock object has no attribute 'a'
```

解决此问题的最好方式可能是添加类属性作为在 `__init__()` 中初始化的实例属性的默认值。请注意如果你只在 `__init__()` 中设置默认属性那么通过类属性来提供它们（当然会在实例之间共享）也将有更快的速度。例如

```
class Something:
    a = 33
```

这带来了另一个问题。为今后将变为不同类型对象的那些成员提供默认值 `None` 是比较常见的做法。`None` 作为 spec 是没有用处的，因为它会使你无法访问 *any* 任何属性或方法。由于 `None` 作为 spec 将 永远不会有任何用处，并且有可能要指定某个通常为其他类型的成员，因此 autospec 不会为被设为 `None` 的成员使用 spec。它们将为普通的 mock (嗯——应为 MagicMocks):

```
>>> class Something:
...     member = None
...
>>> mock = create_autospec(Something)
>>> mock.member.foo.bar.baz()
<MagicMock name='mock.member.foo.bar.baz()' id='...'>
```

如果你不喜欢修改你的生产类来添加默认值那么还有其他的选项。其中之一是简单地使用一个实例而非类作为 spec。另一选项则是创建一个生产类的子类并向该子类添加默认值而不影响到生产类。这两个选项都需要你使用一个替代对象作为 spec。值得庆幸的是*patch()* 支持这样做——你可以简单地传入替代对象作为 *autospec* 参数:

```
>>> class Something:
...   def __init__(self):
...     self.a = 33
...
>>> class SomethingForTest(Something):
...   a = 33
...
>>> p = patch('__main__.Something', autospec=SomethingForTest)
>>> mock = p.start()
>>> mock.a
<NonCallableMagicMock name='Something.a' spec='int' id='...'>
```

将 mock 封包

unittest.mock.**seal**(*mock*)

封包将在访问被封包的 mock 的属性或其任何已经被递归地模拟的属性时禁止自动创建 mock。

如果一个带有名称或 spec 的 mock 实例被分配给一个属性则将不会在封包链中处理它。这可以防止人们对 mock 对象的固定部分执行封包。

```
>>> mock = Mock()
>>> mock.submock.attribute1 = 2
>>> mock.not_submock = mock.Mock(name="sample_name")
>>> seal(mock)
>>> mock.new_attribute  # This will raise AttributeError.
>>> mock.submock.attribute2  # This will raise AttributeError.
>>> mock.not_submock.attribute2  # This won't raise.
```

3.7 版新加入.

26.10 unittest.mock 上手指南

3.3 版新加入.

26.10.1 使用 mock

模拟方法调用

使用*Mock* 的常见场景：

- 模拟函数调用
- 记录“对象上的方法调用”

你可能需要替换一个对象上的方法，用于确认此方法被系统中的其他部分调用过，并且调用时使用了正确的参数。

```
>>> real = SomeClass()
>>> real.method = MagicMock(name='method')
>>> real.method(3, 4, 5, key='value')
<MagicMock name='method()' id='...'>
```

使用了 mock（本例中的 real.method）之后，它有方法和属性可以让你针对它是被如何使用的下断言。

備🄴: 在多数示例中，*Mock* 与*MagicMock* 两个类可以相互替换，而 MagicMock 是一个更适用的类，通常情况下，使用它就可以了。

如果 mock 被调用，它的*called* 属性就会变成 True，更重要的是，我们可以使用*assert_called_with()* 或者*assert_called_once_with()* 方法来确认它在被调用时使用了正确的参数。

在如下的测试示例中，验证对于 ProductionClass().method 的调用会导致 something 的调用。

```
>>> class ProductionClass:
...     def method(self):
...         self.something(1, 2, 3)
...     def something(self, a, b, c):
...         pass
...
>>> real = ProductionClass()
>>> real.something = MagicMock()
>>> real.method()
>>> real.something.assert_called_once_with(1, 2, 3)
```

对象上的方法调用的 mock

上一个例子中我们直接在对象上给方法打补丁以检查它是否被正确地调用。另一个常见的用例是将一个对象传给一个方法（或被测试系统的某个部分）然后检查它是否以正确的方式被使用。

下面这个简单的 ProductionClass 具有一个 closer 方法。如果它附带一个对象被调用那么它就会调用其中的 close。

```
>>> class ProductionClass:
...     def closer(self, something):
...         something.close()
...
```

所以为了测试它我们需要传入一个带有 close 方法的对象并检查它是否被正确地调用。

```
>>> real = ProductionClass()
>>> mock = Mock()
>>> real.closer(mock)
>>> mock.close.assert_called_with()
```

我们不需要做任何事来在我们的 mock 上提供'close' 方法。访问 close 的操作就会创建它。因此，如果'close' 还未被调用那么在测试时访问它就将创建它，但是*assert_called_with()* 则会引发一个失败的异常。

模拟类

一个常见的用例是模拟被测试的代码所实例化的类。当你给一个类打上补丁，该类就会被替换为一个 mock。实例是通过 该用该类来创建的。这意味着你要通过查看被模拟类的返回值来访问"mock 实例"。

在下面的例子中我们有一个函数 some_function 实例化了 Foo 并调用该实例中的一个方法。对*patch()* 的调用会将类 Foo 替换为一个 mock。Foo 实例是调用该 mock 的结果，所以它是通过修改*return_value* 来配置的。

```
>>> def some_function():
...     instance = module.Foo()
...     return instance.method()
...
>>> with patch('module.Foo') as mock:
...     instance = mock.return_value
...     instance.method.return_value = 'the result'
...     result = some_function()
...     assert result == 'the result'
```

命名你的 mock

给你的 mock 起个名字可能会很有用。名字会显示在 mock 的 repr 中并在 mock 出现于测试失败消息中时可以帮助理解。这个名字也会被传播给 mock 的属性或方法:

```
>>> mock = MagicMock(name='foo')
>>> mock
<MagicMock name='foo' id='...'>
>>> mock.method
<MagicMock name='foo.method' id='...'>
```

追踪所有的调用

通常你会想要追踪对某个方法的多次调用。*mock_calls* 属性记录了所有对 mock 的子属性的调用——并且还包括对它们的子属性的调用。

```
>>> mock = MagicMock()
>>> mock.method()
<MagicMock name='mock.method()' id='...'>
>>> mock.attribute.method(10, x=53)
<MagicMock name='mock.attribute.method()' id='...'>
>>> mock.mock_calls
[call.method(), call.attribute.method(10, x=53)]
```

如果你做了一个有关 mock_calls 的断言并且有任何非预期的方法被调用，则断言将失败。这很有用处，因为除了断言你所预期的调用已被执行，你还会检查它们是否以正确的顺序被执行并且没有额外的调用:

你使用*call* 对象来构造列表以便与 mock_calls 进行比较:

```
>>> expected = [call.method(), call.attribute.method(10, x=53)]
>>> mock.mock_calls == expected
True
```

然而，返回 mock 的调用的形参不会被记录，这意味着不可能追踪附带了重要形参的创建上级对象的嵌套调用:

```
>>> m = Mock()
>>> m.factory(important=True).deliver()
<Mock name='mock.factory().deliver()' id='...'>
>>> m.mock_calls[-1] == call.factory(important=False).deliver()
True
```

设置返回值和属性

在 mock 对象上设置返回值是非常容易的:

```
>>> mock = Mock()
>>> mock.return_value = 3
>>> mock()
3
```

当然你也可以对 mock 上的方法做同样的操作:

```
>>> mock = Mock()
>>> mock.method.return_value = 3
>>> mock.method()
3
```

返回值也可以在构造器中设置:

```
>>> mock = Mock(return_value=3)
>>> mock()
3
```

如果你需要在你的 mock 上设置一个属性，只需这样做:

```
>>> mock = Mock()
>>> mock.x = 3
>>> mock.x
3
```

有时你会想要模拟更复杂的情况，例如这个例子 `mock.connection.cursor().execute("SELECT 1")`。如果我们希望这个调用返回一个列表，那么我们还必须配置嵌套调用的结果。

我们可以像这样使用*call* 在一个“链式调用”中构造调用集合以便随后方便地设置断言:

```
>>> mock = Mock()
>>> cursor = mock.connection.cursor.return_value
>>> cursor.execute.return_value = ['foo']
>>> mock.connection.cursor().execute("SELECT 1")
['foo']
>>> expected = call.connection.cursor().execute("SELECT 1").call_list()
>>> mock.mock_calls
[call.connection.cursor(), call.connection.cursor().execute('SELECT 1')]
>>> mock.mock_calls == expected
True
```

对 `.call_list()` 的调用会将我们的调用对象转成一个代表链式调用的调用列表。

通过 mock 引发异常

一个很有用的属性是*side_effect*。如果你将该属性设为一个异常类或者实例那么当 mock 被调用时该异常将会被引发。

```
>>> mock = Mock(side_effect=Exception('Boom!'))
>>> mock()
Traceback (most recent call last):
  ...
Exception: Boom!
```

附带影响函数和可迭代对象

side_effect 也可以被设为一个函数或可迭代对象。side_effect 作为可迭代对象的应用场景适用于你的 mock 将要被多次调用，并且你希望每次调用都返回不同的值的情况。当你将 side_effect 设为一个可迭代对象时每次对 mock 的调用将返回可迭代对象的下一个值。

```
>>> mock = MagicMock(side_effect=[4, 5, 6])
>>> mock()
4
>>> mock()
5
>>> mock()
6
```

对于更高级的用例，例如根据 mock 调用时附带的参数动态改变返回值，side_effect 可以指定一个函数。该函数将附带与 mock 相同的参数被调用。该函数所返回的就是调用所返回的对象:

```
>>> vals = {(1, 2): 1, (2, 3): 2}
>>> def side_effect(*args):
...     return vals[args]
...
>>> mock = MagicMock(side_effect=side_effect)
>>> mock(1, 2)
1
>>> mock(2, 3)
2
```

模拟异步迭代器

从 Python 3.8 起，AsyncMock 和 MagicMock 支持通过 __aiter__ 来模拟 async-iterators。__aiter__ 的*return_value* 属性可以被用来设置要用于迭代的返回值。

```
>>> mock = MagicMock()  # AsyncMock also works here
>>> mock.__aiter__.return_value = [1, 2, 3]
>>> async def main():
...     return [i async for i in mock]
...
>>> asyncio.run(main())
[1, 2, 3]
```

模拟异步上下文管理器

从 Python 3.8 起，AsyncMock 和 MagicMock 支持通过 __aenter__ 和 __aexit__ 来模拟 async-context-managers。在默认情况下，__aenter__ 和 __aexit__ 将为返回异步函数的 AsyncMock 实例。

```
>>> class AsyncContextManager:
...     async def __aenter__(self):
...         return self
...     async def __aexit__(self, exc_type, exc, tb):
...         pass
...
>>> mock_instance = MagicMock(AsyncContextManager())  # AsyncMock also works here
>>> async def main():
...     async with mock_instance as result:
```

(下页继续)

(繼續上一頁)

```
...         pass
...
>>> asyncio.run(main())
>>> mock_instance.__aenter__.assert_awaited_once()
>>> mock_instance.__aexit__.assert_awaited_once()
```

基于现有对象创建模拟对象

使用模拟操作的一个问题是它会将你的测试与你的 mock 实现相关联而不是与你的真实代码相关联。假设你有一个实现了 `some_method` 的类。在对另一个类的测试中，你提供了一个 同样提供了 `some_method` 的模拟该对象的 mock 对象。如果后来你重构了第一个类，使得它不再具有 `some_method` ——那么你的测试将继续保持通过，尽管现在你的代码已经被破坏了！

Mock 允许你使用 allows you to provide an object as a specification for the mock, using the *spec* 关键字参数来提供一个对象作为 mock 的规格说明。在 mock 上访问不存在于你的规格说明对象中的方法 / 属性将立即引发一个属性错误。如果你修改你的规格说明的实现，，那么使用了该类的测试将立即开始失败而不需要你在这些测试中实例化该类。

```
>>> mock = Mock(spec=SomeClass)
>>> mock.old_method()
Traceback (most recent call last):
   ...
AttributeError: object has no attribute 'old_method'
```

使用规格说明还可以启用对 mock 的调用的更聪明的匹配操作，无论是否有将某些形参作为位置或关键字参数传入:

```
>>> def f(a, b, c): pass
...
>>> mock = Mock(spec=f)
>>> mock(1, 2, 3)
<Mock name='mock()' id='140161580456576'>
>>> mock.assert_called_with(a=1, b=2, c=3)
```

如果你想要让这些更聪明的匹配操作也适用于 mock 上的方法调用，你可以使用*auto-speccing*。

如果你想要更强形式的规格说明以防止设置任意属性并获取它们那么你可以使用 *spec_set* 来代替 *spec*。

26.10.2 补丁装饰器

備F: 在查找对象的名称空间中修补对象使用*patch()* 。使用起来很简单，阅读*在哪里打补丁* 来快速上手。

测试中的一个常见需求是为类属性或模块属性打补丁，例如修补内置对象或修补某个模块中的类来测试其是否被实例化。模块和类都可算是全局对象，因此对它们打补丁的操作必须在测试完成之后被还原否则补丁将持续影响其他测试并导致难以诊断的问题。

为此 mock 提供了三个便捷的装饰器: *patch()*, *patch.object()* 和*patch.dict()*。`patch` 接受单个字符串，其形式 `package.module.Class.attribute` 指明你要修补的属性。它还可选择接受一个值用来替换指定的属性（或者类对象等等）。'patch.object' 接受一个对象和你想要修补的属性名称，并可选择接受要用作补丁的值。

`patch.object`:

```
>>> original = SomeClass.attribute
>>> @patch.object(SomeClass, 'attribute', sentinel.attribute)
... def test():
...     assert SomeClass.attribute == sentinel.attribute
...
>>> test()
>>> assert SomeClass.attribute == original

>>> @patch('package.module.attribute', sentinel.attribute)
... def test():
...     from package.module import attribute
...     assert attribute is sentinel.attribute
...
>>> test()
```

如果你要给一个模块 (包括*builtins*) 打补丁则可使用*patch()* 来代替*patch.object()*:

```
>>> mock = MagicMock(return_value=sentinel.file_handle)
>>> with patch('builtins.open', mock):
...     handle = open('filename', 'r')
...
>>> mock.assert_called_with('filename', 'r')
>>> assert handle == sentinel.file_handle, "incorrect file handle returned"
```

如有必要模块名可以是“带点号”的，其形式如 package.module:

```
>>> @patch('package.module.ClassName.attribute', sentinel.attribute)
... def test():
...     from package.module import ClassName
...     assert ClassName.attribute == sentinel.attribute
...
>>> test()
```

一个良好的模式是实际地装饰测试方法本身:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'attribute', sentinel.attribute)
...     def test_something(self):
...         self.assertEqual(SomeClass.attribute, sentinel.attribute)
...
>>> original = SomeClass.attribute
>>> MyTest('test_something').test_something()
>>> assert SomeClass.attribute == original
```

如果你想要通过 Mock 来打补丁，你可以只附带一个参数使用*patch()* (或附带两个参数使用*patch.object()*)。这将为你创建 mock 并传递给测试函数 / 方法:

```
>>> class MyTest(unittest.TestCase):
...     @patch.object(SomeClass, 'static_method')
...     def test_something(self, mock_method):
...         SomeClass.static_method()
...         mock_method.assert_called_with()
...
>>> MyTest('test_something').test_something()
```

你可以使用以下模式来堆叠多个补丁装饰器:

```
>>> class MyTest(unittest.TestCase):
...     @patch('package.module.ClassName1')
...     @patch('package.module.ClassName2')
...     def test_something(self, MockClass2, MockClass1):
...         self.assertIs(package.module.ClassName1, MockClass1)
...         self.assertIs(package.module.ClassName2, MockClass2)
...
>>> MyTest('test_something').test_something()
```

当你嵌套 patch 装饰器时将以它们被应用的相同顺序（即 *Python* 应用装饰器的正常顺序）将 mock 传入被装饰的函数。也就是说从下往上，因此在上面的示例中 `test_module.ClassName2` 的 mock 会被最先传入。

还有一个*patch.dict()* 用于在一定范围内设置字典中的值，并在测试结束时将字典恢复为其原始状态：

```
>>> foo = {'key': 'value'}
>>> original = foo.copy()
>>> with patch.dict(foo, {'newkey': 'newvalue'}, clear=True):
...     assert foo == {'newkey': 'newvalue'}
...
>>> assert foo == original
```

`patch`, `patch.object` 和 `patch.dict` 都可被用作上下文管理器。

在你使用*patch()* 为你创建 mock 时，你可以使用 with 语句的"as" 形式来获得对 mock 的引用：

```
>>> class ProductionClass:
...     def method(self):
...         pass
...
>>> with patch.object(ProductionClass, 'method') as mock_method:
...     mock_method.return_value = None
...     real = ProductionClass()
...     real.method(1, 2, 3)
...
>>> mock_method.assert_called_with(1, 2, 3)
```

作为替代 `patch`, `patch.object` 和 `patch.dict` 可以被用作类装饰器。当以此方式使用时其效果与将装饰器单独应用到每个以"test" 打头的方法上相同。

26.10.3 更多示例

下面是一些针对更为高级应用场景的补充示例。

模拟链式调用

实际上一旦你理解了*return_value* 属性那么使用 mock 模拟链式调用就会相当直观。当一个 mock 首次被调用，或者当你在它被调用前获取其 `return_value` 时，将会创建一个新的*Mock*。

这意味着你可以通过检视 `return_value` mock 来了解从调用被模拟对象返回的对象是如何被使用的：

```
>>> mock = Mock()
>>> mock().foo(a=2, b=3)
<Mock name='mock().foo()' id='...'>
>>> mock.return_value.foo.assert_called_with(a=2, b=3)
```

从这里开始只需一个步骤即可配置并创建有关链式调用的断言。当然还有另一种选择是首先以更易于测试的方式来编写你的代码...

因此，如果我们有这样一些代码:

```
>>> class Something:
...     def __init__(self):
...         self.backend = BackendProvider()
...     def method(self):
...         response = self.backend.get_endpoint('foobar').create_call('spam', 'eggs
↪').start_call()
...         # more code
```

假定 BackendProvider 已经过良好测试，我们要如何测试 method()？特别地，我们希望测试代码段 # more code 是否以正确的方式使用了响应对象。

由于这个链式调用来自一个实例属性我们可以对 backend 属性在 Something 实例上进行猴子式修补。在这个特定情况下我们只对最后调用 start_call 的返回值感兴趣所以我们不需要进行太多的配置。让我们假定它返回的是“文件类”对象，因此我们将确保我们的响应对象使用内置的*open()* 作为其 spec。

为了做到这一点我们创建一个 mock 实例作为我们的 mock 后端并为它创建一个 mock 响应对象。要将该响应对象设为最后的 start_call 的返回值我们可以这样做:

```
mock_backend.get_endpoint.return_value.create_call.return_value.start_call.return_
↪value = mock_response
```

我们可以通过更好一些的方式做到这一点，即使用*configure_mock()* 方法直接为我们设置返回值:

```
>>> something = Something()
>>> mock_response = Mock(spec=open)
>>> mock_backend = Mock()
>>> config = {'get_endpoint.return_value.create_call.return_value.start_call.return_
↪value': mock_response}
>>> mock_backend.configure_mock(**config)
```

有了这些我们就能准备好给“mock 后端”打上猴子补丁并可以执行真正的调用:

```
>>> something.backend = mock_backend
>>> something.method()
```

使用*mock_calls* 我们可以通过一个断言来检查链式调用。一个链式调用就是在一行代码中连续执行多个调用，所以在 mock_calls 中将会有多个条目。我们可以使用*call.call_list()* 来为我们创建这个调用列表:

```
>>> chained = call.get_endpoint('foobar').create_call('spam', 'eggs').start_call()
>>> call_list = chained.call_list()
>>> assert mock_backend.mock_calls == call_list
```

部分模拟

在某些测试中我希望模拟对*datetime.date.today()* 的调用以返回一个已知的日期，但我又不想阻止被测试的代码创建新的日期对象。很不幸*datetime.date* 是用 C 语言编写的，因此我不能简单地给静态的 `date.today()` 方法打上猴子补丁。

我找到了实现这一点的简单方式即通过一个 mock 来实际包装日期类，但通过对构造器的调用传递给真实的类（并返回真实的实例）。

这里使用*patch 装饰器* 来模拟被测试模块中的 `date` 类。模拟 date 类中的 `side_effect` 属性随后被设为一个返回真实日期的 lambda 函数。当模拟 date 类被调用时将由 `side_effect` 构造并返回一个真实日期。

```
>>> from datetime import date
>>> with patch('mymodule.date') as mock_date:
...     mock_date.today.return_value = date(2010, 10, 8)
...     mock_date.side_effect = lambda *args, **kw: date(*args, **kw)
...
...     assert mymodule.date.today() == date(2010, 10, 8)
...     assert mymodule.date(2009, 6, 8) == date(2009, 6, 8)
```

请注意我们没有在全局范围上修补*datetime.date*，我们只是在 使用它的模块中给 `date` 打补丁。参见*补丁的位置*。

当 `date.today()` 被调用时将返回一个已知的日期，但对 `date(...)` 构造器的调用仍会返回普通的日期。如果不是这样你会发现你必须使用与被测试的代码完全相同的算法来计算出预期的结果，这是测试工作中的一个经典的反模式。

对 date 构造器的调用会被记录在 `mock_date` 属性中 (`call_count` 等)，它们也可能对你的测试有用处。

有关处理模块日期或其他内置类的一种替代方式的讨论请参见 这篇博客文章。

模拟生成器方法

Python 生成器是指在被迭代时使用 `yield` 语句来返回一系列值的函数或方法[1]。

调用生成器方法 / 函数将返回生成器对象。生成器对象随后会被迭代。迭代操作对应的协议方法是*__iter__()*，因此我们可以使用*MagicMock* 来模拟它。

以下是一个使用”iter” 方法模拟为生成器的示例类:

```
>>> class Foo:
...     def iter(self):
...         for i in [1, 2, 3]:
...             yield i
...
>>> foo = Foo()
>>> list(foo.iter())
[1, 2, 3]
```

我们要如何模拟这个类，特别是它的”iter” 方法呢?

为了配置从迭代操作（隐含在对*list* 的调用中）返回的值，我们需要配置调用 `foo.iter()` 所返回的对象。

```
>>> mock_foo = MagicMock()
>>> mock_foo.iter.return_value = iter([1, 2, 3])
>>> list(mock_foo.iter())
[1, 2, 3]
```

[1] 此外还有生成器表达式和更多的生成器 进阶用法，但在这里我们不去关心它们。有关生成器及其强大功能的一个很好的介绍请参阅: 针对系统程序员的生成器妙招。

对每个测试方法应用相同的补丁

如果你想要为多个测试方法准备好一些补丁那么最简单的方式就是将 patch 装饰器应用到每个方法。这在感觉上会造成不必要的重复。对于 Python 2.6 或更新的版本你可以使用*patch()* (其所有不同的形式) 作为类装饰器。这将把补丁应用于类中的所有测试方法。测试方法是通过名称前缀为 `test` 来标识的:

```
>>> @patch('mymodule.SomeClass')
... class MyTest(unittest.TestCase):
...
...     def test_one(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def test_two(self, MockSomeClass):
...         self.assertIs(mymodule.SomeClass, MockSomeClass)
...
...     def not_a_test(self):
...         return 'something'
...
>>> MyTest('test_one').test_one()
>>> MyTest('test_two').test_two()
>>> MyTest('test_two').not_a_test()
'something'
```

另一种管理补丁的方式是使用*补丁方法: start 和 stop*。它允许你将打补丁操作移至你的 `setUp` 和 `tearDown` 方法中。

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         self.patcher = patch('mymodule.foo')
...         self.mock_foo = self.patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
...     def tearDown(self):
...         self.patcher.stop()
...
>>> MyTest('test_foo').run()
```

如果你要使用这个技巧则你必须通过调用 `stop` 来确保补丁被“恢复”。这可能要比你想像的更麻烦，因为如果在 setUp 中引发了异常那么 tearDown 将不会被调用。*unittest.TestCase.addCleanup()* 可以做到更方便:

```
>>> class MyTest(unittest.TestCase):
...     def setUp(self):
...         patcher = patch('mymodule.foo')
...         self.addCleanup(patcher.stop)
...         self.mock_foo = patcher.start()
...
...     def test_foo(self):
...         self.assertIs(mymodule.foo, self.mock_foo)
...
>>> MyTest('test_foo').run()
```

模拟未绑定方法

当前在编写测试时我需要修补一个 *未绑定方法* (在类上而不是在实例上为方法打补丁)。我需要将 self 作为第一个参数传入因为我想对哪些对象在调用这个特定方法进行断言。问题是这里你不能用 mock 来打补丁，因为如果你用 mock 来替换一个未绑定方法那么当从实例中获取时它就不会成为一个已绑定方法，因而它不会获得传入的 self。绕过此问题的办法是改用一个真正的函数来修补未绑定方法。*patch()* 装饰器让使用 mock 来给方法打补丁变得如此简单以至于创建一个真正的函数成为一件麻烦事。

如果将 `autospec=True` 传给 patch 那么它就会用一个 *真正的函数对象来打补丁。这个函数对象具有与它所替换的函数相同的签名，但会在内部将操作委托给一个 mock。你仍然可以通过与以前完全相同的方式来自动创建你的 mock。但是这将意味着一件事，就是如果你用它来修补一个类上的非绑定方法那么如果它是从一个实例中获取则被模拟的函数将被转为已绑定方法。传给它的第一个参数将为 `self`，而这真是我想要的:

```
>>> class Foo:
...   def foo(self):
...     pass
...
>>> with patch.object(Foo, 'foo', autospec=True) as mock_foo:
...   mock_foo.return_value = 'foo'
...   foo = Foo()
...   foo.foo()
...
'foo'
>>> mock_foo.assert_called_once_with(foo)
```

如果我们不使用 `autospec=True` 那么这个未绑定方法会改为通过一个 Mock 补丁来修补，而不是附带 `self` 来调用。

通过 mock 检查多次调用

mock 有一个很好的 API 用于针对你的 mock 对象如何被使用来下断言。

```
>>> mock = Mock()
>>> mock.foo_bar.return_value = None
>>> mock.foo_bar('baz', spam='eggs')
>>> mock.foo_bar.assert_called_with('baz', spam='eggs')
```

如果你的 mock 只会被调用一次你可以使用 `assert_called_once_with()` 方法，这也将断言 `call_count` 为一。

```
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
>>> mock.foo_bar()
>>> mock.foo_bar.assert_called_once_with('baz', spam='eggs')
Traceback (most recent call last):
    ...
AssertionError: Expected to be called once. Called 2 times.
```

`assert_called_with` 和 `assert_called_once_with` 都是有关 *最近调用的断言。如果你的 mock 将被多次调用，并且你想要针对 所有这些调用下断言你可以使用 *call_args_list*:

```
>>> mock = Mock(return_value=None)
>>> mock(1, 2, 3)
>>> mock(4, 5, 6)
>>> mock()
>>> mock.call_args_list
[call(1, 2, 3), call(4, 5, 6), call()]
```

使用*call* 辅助对象可以方便地针对这些调用下断言。你可以创建一个预期调用的列表并将其与 call_args_list 比较。这看起来与 call_args_list 的 repr 非常相似:

```
>>> expected = [call(1, 2, 3), call(4, 5, 6), call()]
>>> mock.call_args_list == expected
True
```

处理可变参数

另一种很少见，但可能给你带来麻烦的情况会在你的 mock 附带可变参数被调用的时候发生。call_args 和 call_args_list 将保存对这些参数的 *引用*。如果这些参数被受测试的代码所改变那么你将无法再针对当该 mock 被调用时附带的参数值下断言。

下面是一些演示此问题的示例代码。设想在'mymodule' 中定义了下列函数:

```
def frob(val):
    pass

def grob(val):
    "First frob and then clear val"
    frob(val)
    val.clear()
```

当我们想要测试 grob 调用 frob 并附带了正确的参数时将可看到发生了什么:

```
>>> with patch('mymodule.frob') as mock_frob:
...     val = {6}
...     mymodule.grob(val)
...
>>> val
set()
>>> mock_frob.assert_called_with({6})
Traceback (most recent call last):
    ...
AssertionError: Expected: (({6},), {})
Called with: ((set(),), {})
```

对于 mock 的一个可能性是复制你传入的参数。如果你创建依赖于对象标识号相等性的断言那么这可能会在后面导致问题。

下面是一个使用 side_effect 功能的解决方案。如果你为 mock 提供了 side_effect 函数那么 side_effect 将附带与该 mock 相同的参数被调用。这使我们有机会拷贝这些参数并保存它们用于之后进行断言。在这个例子中我使用了 另一个 mock 来保存参数以便我可以使用该 mock 的方法来进行断言。在这里辅助函数再次为我设置好了所需的功能。

```
>>> from copy import deepcopy
>>> from unittest.mock import Mock, patch, DEFAULT
>>> def copy_call_args(mock):
...     new_mock = Mock()
...     def side_effect(*args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         new_mock(*args, **kwargs)
...         return DEFAULT
...     mock.side_effect = side_effect
...     return new_mock
```

(下页继续)

(繼續上一頁)

```
...
>>> with patch('mymodule.frob') as mock_frob:
...     new_mock = copy_call_args(mock_frob)
...     val = {6}
...     mymodule.grob(val)
...
>>> new_mock.assert_called_with({6})
>>> new_mock.call_args
call({6})
```

调用 copy_call_args 时会传入将被调用的 mock。它将返回一个新的 mock 供我们进行断言。side_effect 函数会拷贝这些参数并附带该副本来调用我们的 new_mock。

備註: 如果你的 mock 只会被使用一次那么有更容易的方式可以在它们被调用时检查参数。你可以简单地在 side_effect 函数中执行检查。

```
>>> def side_effect(arg):
...     assert arg == {6}
...
>>> mock = Mock(side_effect=side_effect)
>>> mock({6})
>>> mock(set())
Traceback (most recent call last):
    ...
AssertionError
```

一个替代方式是创建一个*Mock* 或*MagicMock* 的子类来拷贝 (使用*copy.deepcopy()*) 参数。下面是一个示例实现:

```
>>> from copy import deepcopy
>>> class CopyingMock(MagicMock):
...     def __call__(self, /, *args, **kwargs):
...         args = deepcopy(args)
...         kwargs = deepcopy(kwargs)
...         return super().__call__(*args, **kwargs)
...
>>> c = CopyingMock(return_value=None)
>>> arg = set()
>>> c(arg)
>>> arg.add(1)
>>> c.assert_called_with(set())
>>> c.assert_called_with(arg)
Traceback (most recent call last):
    ...
AssertionError: Expected call: mock({1})
Actual call: mock(set())
>>> c.foo
<CopyingMock name='mock.foo' id='...'>
```

当你子类化 Mock 或 MagicMock 时所有动态创建的属性以及 return_value 都将自动使用你的子类。这意味着 CopyingMock 的所有子类也都将为 CopyingMock 类型。

嵌套补丁

使用 patch 作为上下文管理器很不错，但是如果你要执行多个补丁你将不断嵌套 with 语句使得代码越来越深地向右缩进:

```
>>> class MyTest(unittest.TestCase):
...
...     def test_foo(self):
...         with patch('mymodule.Foo') as mock_foo:
...             with patch('mymodule.Bar') as mock_bar:
...                 with patch('mymodule.Spam') as mock_spam:
...                     assert mymodule.Foo is mock_foo
...                     assert mymodule.Bar is mock_bar
...                     assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').test_foo()
>>> assert mymodule.Foo is original
```

使用 unittest `cleanup` 函数和*补丁方法: start 和 stop* 我们可以达成同样的效果而无须嵌套缩进。一个简单的辅助方法 `create_patch` 会为我们执行打补丁操作并返回所创建的 mock:

```
>>> class MyTest(unittest.TestCase):
...
...     def create_patch(self, name):
...         patcher = patch(name)
...         thing = patcher.start()
...         self.addCleanup(patcher.stop)
...         return thing
...
...     def test_foo(self):
...         mock_foo = self.create_patch('mymodule.Foo')
...         mock_bar = self.create_patch('mymodule.Bar')
...         mock_spam = self.create_patch('mymodule.Spam')
...
...         assert mymodule.Foo is mock_foo
...         assert mymodule.Bar is mock_bar
...         assert mymodule.Spam is mock_spam
...
>>> original = mymodule.Foo
>>> MyTest('test_foo').run()
>>> assert mymodule.Foo is original
```

使用 MagicMock 模拟字典

你可能会想要模拟一个字典或其他容器对象，记录所有对它的访问并让它的行为仍然像是一个字典。

要做到这点我们可以用*MagicMock*，它的行为类似于字典，并会使用*side_effect* 将字典访问委托给下层的在我们控制之下的一个真正的字典。

当我们的 MagicMock 的 `__getitem__()` 和 `__setitem__()` 方法被调用（即普通的字典访问操作）时 `side_effect` 将附带键（对于 `__setitem__` 则还将附带值）被调用。我们也可以控制返回的对象。

在 MagicMock 被使用之后我们可以使用*call_args_list* 等属性来针对该字典是如何被使用的下断言。

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> def getitem(name):
```

(下页继续)

(繼續上一頁)

```
...       return my_dict[name]
...
>>> def setitem(name, val):
...     my_dict[name] = val
...
>>> mock = MagicMock()
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

備F: MagicMock 的一个可用替代是使用 Mock 并 仅仅提供你明确需要的魔术方法:

```
>>> mock = Mock()
>>> mock.__getitem__ = Mock(side_effect=getitem)
>>> mock.__setitem__ = Mock(side_effect=setitem)
```

第三个选项是使用 MagicMock 但传入 dict 作为 *spec* (或 *spec_set*) 参数以使得所创建的 MagicMock 只有字典魔术方法是可用的:

```
>>> mock = MagicMock(spec_set=dict)
>>> mock.__getitem__.side_effect = getitem
>>> mock.__setitem__.side_effect = setitem
```

通过提供这些附带影响函数，mock 的行为将类似于普通字典但又会记录所有访问。如果你尝试访问一个不存在的键它甚至会引发*KeyError*。

```
>>> mock['a']
1
>>> mock['c']
3
>>> mock['d']
Traceback (most recent call last):
    ...
KeyError: 'd'
>>> mock['b'] = 'fish'
>>> mock['d'] = 'eggs'
>>> mock['b']
'fish'
>>> mock['d']
'eggs'
```

在它被使用之后你可以使用普通的 mock 方法和属性进行有关访问操作的断言:

```
>>> mock.__getitem__.call_args_list
[call('a'), call('c'), call('d'), call('b'), call('d')]
>>> mock.__setitem__.call_args_list
[call('b', 'fish'), call('d', 'eggs')]
>>> my_dict
{'a': 1, 'b': 'fish', 'c': 3, 'd': 'eggs'}
```

模拟子类及其属性

你可能出于各种原因想要子类化*Mock*。其中一个可能的原因是为了添加辅助方法。下面是一个笨兮兮的示例:

```
>>> class MyMock(MagicMock):
...     def has_been_called(self):
...         return self.called
...
>>> mymock = MyMock(return_value=None)
>>> mymock
<MyMock id='...'>
>>> mymock.has_been_called()
False
>>> mymock()
>>> mymock.has_been_called()
True
```

The standard behaviour for `Mock` 实例的标准行为是属性和返回值 mock 具有与它们所访问的 mock 相同的类型。这将确保 `Mock` 的属性均为 `Mocks` 而 `MagicMock` 的属性均为 `MagicMocks`[2]。因此如果你通过子类化来添加辅助方法那么它们也将在你的子类的实例的属性和返回值 mock 上可用。

```
>>> mymock.foo
<MyMock name='mock.foo' id='...'>
>>> mymock.foo.has_been_called()
False
>>> mymock.foo()
<MyMock name='mock.foo()' id='...'>
>>> mymock.foo.has_been_called()
True
```

有时这很不方便。例如，有一位用户 子类化 mock 来创建一个 Twisted 适配器。将它也应用到属性实际上会导致出错。

`Mock` (它的所有形式) 使用一个名为 `_get_child_mock` 的方法来创建这些用于属性和返回值的 “子 mock”。你可以通过重载此方法来防止你的子类被用于属性。其签名被设为接受任意关键字参数 (`**kwargs`) 并且它们会被传递给 mock 构造器:

```
>>> class Subclass(MagicMock):
...     def _get_child_mock(self, /, **kwargs):
...         return MagicMock(**kwargs)
...
>>> mymock = Subclass()
>>> mymock.foo
<MagicMock name='mock.foo' id='...'>
>>> assert isinstance(mymock, Subclass)
>>> assert not isinstance(mymock.foo, Subclass)
>>> assert not isinstance(mymock(), Subclass)
```

[2] 此规则的一个例外涉及不可调用 mock。属性会使用可调用对象版本是因为如非如此则不可调用 mock 将无法拥有可调用的方法。

通过 patch.dict 模拟导入

有一种会令模拟变困难的情况是当你在函数内部有局部导入。这更难模拟的原因是它们不是使用来自我们能打补丁的模拟命名空间中的对象。

一般来说局部导入是应当避免的。局部导入有时是为了防止循环依赖，而这个问题 通常都有更好的解决办法（重构代码）或者通过延迟导入来防止“前期成本”。这也可以通过比无条件地局部导入更好的方式来解决（将模块保存为一个类或模块属性并且只在首次使用时执行导入）。

除此之外还有一个办法可以使用 mock 来影响导入的结果。导入操作会从*sys.modules* 字典提取一个 对象。请注意是提取一个 对象，不是必须为模块。首次导入一个模块将使一个模块对象被放入 *sys.modules*，因此通常当你执行导入时你将得到一个模块。然而这并不是必然的。

这意味着你可以使用*patch.dict()* 来 临时性地将一个 mock 放入*sys.modules*。在补丁激活期间的任何导入操作都将得到该 mock。当补丁完成时（被装饰的函数退出，with 语句代码块结束或者 patcher.stop() 被调用）则之前存在的任何东西都将被安全地恢复。

下面是一个模拟'fooble' 模拟的示例。

```
>>> import sys
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     import fooble
...     fooble.blob()
...
<Mock name='mock.blob()' id='...'>
>>> assert 'fooble' not in sys.modules
>>> mock.blob.assert_called_once_with()
```

你可以看到 import fooble 成功执行，而当退出时*sys.modules* 中将不再有'fooble'。

这同样适用于 from module import name 形式:

```
>>> mock = Mock()
>>> with patch.dict('sys.modules', {'fooble': mock}):
...     from fooble import blob
...     blob.blip()
...
<Mock name='mock.blob.blip()' id='...'>
>>> mock.blob.blip.assert_called_once_with()
```

稍微多做一点工作你还可以模拟包的导入:

```
>>> mock = Mock()
>>> modules = {'package': mock, 'package.module': mock.module}
>>> with patch.dict('sys.modules', modules):
...     from package.module import fooble
...     fooble()
...
<Mock name='mock.module.fooble()' id='...'>
>>> mock.module.fooble.assert_called_once_with()
```

追踪调用顺序和不太冗长的调用断言

Mock 类允许你通过*method_calls* 属性来追踪在你的 mock 对象上的方法调用的 顺序。这并不允许你追踪单独 mock 对象之间的调用顺序，但是我们可以使用*mock_calls* 来达到同样的效果。

因为 mock 会追踪 mock_calls 中对子 mock 的调用，并且访问 mock 的任意属性都会创建一个子 mock，所以我们可以基于父 mock 创建单独的子 mock。随后对这些子 mock 的调用将按顺序被记录在父 mock 的 mock_calls 中:

```
>>> manager = Mock()
>>> mock_foo = manager.foo
>>> mock_bar = manager.bar
```

```
>>> mock_foo.something()
<Mock name='mock.foo.something()' id='...'>
>>> mock_bar.other.thing()
<Mock name='mock.bar.other.thing()' id='...'>
```

```
>>> manager.mock_calls
[call.foo.something(), call.bar.other.thing()]
```

我们可以随后通过与管理 mock 上的 mock_calls 属性进行比较来进行有关这些调用，包括调用顺序的断言:

```
>>> expected_calls = [call.foo.something(), call.bar.other.thing()]
>>> manager.mock_calls == expected_calls
True
```

如果 patch 创建并准备好了你的 mock 那么你可以使用*attach_mock()* 方法将它们附加到管理 mock 上。在附加之后所有调用都将被记录在管理器的 mock_calls 中。

```
>>> manager = MagicMock()
>>> with patch('mymodule.Class1') as MockClass1:
...     with patch('mymodule.Class2') as MockClass2:
...         manager.attach_mock(MockClass1, 'MockClass1')
...         manager.attach_mock(MockClass2, 'MockClass2')
...         MockClass1().foo()
...         MockClass2().bar()
<MagicMock name='mock.MockClass1().foo()' id='...'>
<MagicMock name='mock.MockClass2().bar()' id='...'>
>>> manager.mock_calls
[call.MockClass1(),
call.MockClass1().foo(),
call.MockClass2(),
call.MockClass2().bar()]
```

如果已经进行了许多调用，但是你只对它们的一个特定序列感兴趣则有一种替代方式是使用*assert_has_calls()* 方法。这需要一个调用的列表（使用*call* 对象来构建）。如果该调用序列在*mock_calls* 中则断言将成功。

```
>>> m = MagicMock()
>>> m().foo().bar().baz()
<MagicMock name='mock().foo().bar().baz()' id='...'>
>>> m.one().two().three()
<MagicMock name='mock.one().two().three()' id='...'>
>>> calls = call.one().two().three().call_list()
>>> m.assert_has_calls(calls)
```

即使链式调用 `m.one().two().three()` 不是对 mock 的唯一调用，该断言仍将成功。

有时可能会对一个 mock 进行多次调用，而你只对断言其中的 某些调用感兴趣。你甚至可能对顺序也不关心。在这种情况下你可以将 `any_order=True` 传给 `assert_has_calls`:

```
>>> m = MagicMock()
>>> m(1), m.two(2, 3), m.seven(7), m.fifty('50')
(...)
>>> calls = [call.fifty('50'), call(1), call.seven(7)]
>>> m.assert_has_calls(calls, any_order=True)
```

更复杂的参数匹配

使用与*ANY* 一样的基本概念我们可以实现匹配器以便在用作 mock 的参数的对象上执行更复杂的断言。

假设我们准备将某个对象传给一个在默认情况下基于对象标识相等（这是 Python 中用户自定义类的默认行为）的 mock。要使用*assert_called_with()* 我们就将必须传入完全相同的对象。如果我们只对该对象的某些属性感兴趣那么我们可以创建一个能为我们检查这些属性的匹配器。

在这个示例中你可以看到为何执行对 `assert_called_with` 的‘标准’调用并不足够:

```
>>> class Foo:
...     def __init__(self, a, b):
...         self.a, self.b = a, b
...
>>> mock = Mock(return_value=None)
>>> mock(Foo(1, 2))
>>> mock.assert_called_with(Foo(1, 2))
Traceback (most recent call last):
    ...
AssertionError: Expected: call(<__main__.Foo object at 0x...>)
Actual call: call(<__main__.Foo object at 0x...>)
```

一个针对我们的 `Foo` 类的比较函数看上去会是这样的:

```
>>> def compare(self, other):
...     if not type(self) == type(other):
...         return False
...     if self.a != other.a:
...         return False
...     if self.b != other.b:
...         return False
...     return True
...
```

而一个可以使用这样的比较函数进行相等性比较运算的匹配器对象看上去会是这样的:

```
>>> class Matcher:
...     def __init__(self, compare, some_obj):
...         self.compare = compare
...         self.some_obj = some_obj
...     def __eq__(self, other):
...         return self.compare(self.some_obj, other)
...
```

将所有这些放在一起:

```
>>> match_foo = Matcher(compare, Foo(1, 2))
>>> mock.assert_called_with(match_foo)
```

Matcher 是用我们的比较函数和我们想要比较的 Foo 对象来实例化的。在 assert_called_with 中将会调用 Matcher 的相等性方法，它会将调用 mock 时附带的对象与我们创建我们的匹配器时附带的对象进行比较。如果它们匹配则 assert_called_with 通过，而如果不匹配则会引发*AssertionError*:

```
>>> match_wrong = Matcher(compare, Foo(3, 4))
>>> mock.assert_called_with(match_wrong)
Traceback (most recent call last):
    ...
AssertionError: Expected: ((<Matcher object at 0x...>,), {})
Called with: ((<Foo object at 0x...>,), {})
```

通过一些调整你可以让比较函数直接引发*AssertionError* 并提供更有用的失败消息。

从 1.5 版开始，Python 测试库 PyHamcrest 提供了类似的功能，在这里可能会很有用，它采用的形式是相等性匹配器 (hamcrest.library.integration.match_equality)。

26.11 2to3 - 自動將 Python 2 的程式碼轉成 Python 3

2to3 是一个 Python 程序，它可以读取 Python 2.x 的源代码并使用一系列的 修复器来将其转换为合法的 Python 3.x 代码。标准库已包含了丰富的修复器，这足以处理几乎所有代码。不过 2to3 的支持库*lib2to3* 是一个很灵活通用的库，所以还可以编写你自己的 2to3 修复器。

26.11.1 使用 2to3

2to3 通常会作为脚本和 Python 解释器一起安装，你可以在 Python 根目录的 Tools/scripts 文件夹下找到它。

2to3 的基本调用参数是一个需要转换的文件或目录列表。对于目录，会递归地寻找其中的 Python 源码。

這邊有簡單的 Python 2 的原始檔案 example.py:

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

它可以在命令行中使用 2to3 转换成 Python 3.x 版本的代码：

```
$ 2to3 example.py
```

这个命令会打印出和源文件的区别。通过传入 -w 参数，2to3 也可以把需要的修改写回到原文件中（除非传入了 -n 参数，否则会为原始文件创建一个副本）：

```
$ 2to3 -w example.py
```

在转换完成后，example.py 看起来像是这样：

```
def greet(name):
    print("Hello, {0}!".format(name))
print("What's your name?")
```

(下页继续)

(繼續上一頁)

```
name = input()
greet(name)
```

注释和缩进都会在转换过程中保持不变。

默认情况下，2to3 会执行预定义修复器 的集合。使用 -l 参数可以列出所有可用的修复器。使用 -f 参数可以明确指定需要使用的修复器集合。而使用 -x 参数则可以明确指定不使用的修复器。下面的例子会只使用 imports 和 has_key 修复器运行：

```
$ 2to3 -f imports -f has_key example.py
```

这个命令会执行除了 apply 之外的所有修复器：

```
$ 2to3 -x apply example.py
```

有一些修复器是需要 显式指定的，它们默认不会执行，必须在命令行中列出才会执行。比如下面的例子，除了默认的修复器以外，还会执行 idioms 修复器：

```
$ 2to3 -f all -f idioms example.py
```

注意这里使用 all 来启用所有默认的修复器。

有些情况下 2to3 会找到源码中有一些需要修改，但是无法自动处理的代码。在这种情况下，2to3 会在差异处下面打印一个警告信息。你应该定位到相应的代码并对其进行修改，以使其兼容 Python 3.x。

2to3 也可以重构 doctests。使用 -d 开启这个模式。需要注意 * 只有 * doctests 会被重构。这种模式下不需要文件是合法的 Python 代码。举例来说，reST 文档中类似 doctests 的示例也可以使用这个选项进行重构。

-v 选项可以输出更多转换程序的详细信息。

由于某些 print 语句可被解读为函数调用或是语句，2to3 并不总能读取包含 print 函数的文件。当 2to3 检测到存在 from __future__ import print_function 编译器指令时，会修改其内部语法将*print()* 解读为函数。这一变动也可以使用 -p 旗标手动开启。使用 -p 来为已转换过 print 语句的代码运行修复器。也可以使用 -e 将*exec()* 解读为函数。

-o 或 --output-dir 选项可以指定将转换后的文件写入其他目录中。由于这种情况下不会覆写原始文件，所以创建副本文件毫无意义，因此也需要使用 -n 选项来禁用创建副本。

3.2.3 版新加入: 增加了 -o 选项。

-W 或 --write-unchanged-files 选项用来告诉 2to3 始终需要输出文件，即使没有任何改动。这在使用 -o 参数时十分有用，这样就可以将整个 Python 源码包完整地转换到另一个目录。这个选项隐含了 -w 选项，否则等于没有作用。

3.2.3 版新加入: 增加了 -W 选项。

--add-suffix 选项接受一个字符串，用来作为后缀附加在输出文件名后面的后面。由于写入的文件名与原始文件不同，所以没有必要创建副本，因此 -n 选项也是必要的。举个例子：

```
$ 2to3 -n -W --add-suffix=3 example.py
```

这样会把转换后的文件写入 example.py3 文件。

3.2.3 版新加入: 增加了 --add-suffix 选项。

将整个项目从一个目录转换到另一个目录可以用这样的命令：

```
$ 2to3 --output-dir=python3-version/mycode -W -n python2-version/mycode
```

26.11.2 修复器

转换代码的每一个步骤都封装在修复器中。可以使用 2to3 -l 来列出可用的修复器。之前已经提到，每个修复器都可以独立地打开或是关闭。下面会对各个修复器做更详细的描述。

apply
移除对 apply() 的使用，举例来说，apply(function, *args, **kwargs) 会被转换成 function(*args, **kwargs)。

asserts
将已弃用的*unittest* 方法替换为正确的。

從	到
failUnlessEqual(a, b)	*assertEqual(a, b)*
assertEquals(a, b)	*assertEqual(a, b)*
failIfEqual(a, b)	*assertNotEqual(a, b)*
assertNotEquals(a, b)	*assertNotEqual(a, b)*
failUnless(a)	*assertTrue(a)*
assert_(a)	*assertTrue(a)*
failIf(a)	*assertFalse(a)*
failUnlessRaises(exc, cal)	*assertRaises(exc, cal)*
failUnlessAlmostEqual(a, b)	*assertAlmostEqual(a, b)*
assertAlmostEquals(a, b)	*assertAlmostEqual(a, b)*
failIfAlmostEqual(a, b)	*assertNotAlmostEqual(a, b)*
assertNotAlmostEquals(a, b)	*assertNotAlmostEqual(a, b)*

basestring
将 basestring 转换为*str*。

buffer
将 buffer 转换为*memoryview*。这个修复器是可选的，因为*memoryview* API 和 buffer 很相似，但不完全一样。

dict
修复字典迭代方法。dict.iteritems() 会转换成*dict.items()*，dict.iterkeys() 会转换成*dict.keys()*，dict.itervalues() 会转换成*dict.values()*。类似的，dict.viewitems()，dict.viewkeys() 和 dict.viewvalues() 会分别转换成*dict.items()*，*dict.keys()* 和*dict.values()*。另外也会将原有的*dict.items()*，*dict.keys()* 和*dict.values()* 方法调用用*list* 包装一层。

except
将 except X, T 转换为 except X as T。

exec
将 exec 语句转换为*exec()* 函数调用。

execfile
移除 execfile() 的使用。execfile() 的实参会使用*open()*，*compile()* 和*exec()* 包装。

exitfunc
将对 sys.exitfunc 的赋值改为使用*atexit* 模块代替。

filter
将*filter()* 函数用*list* 包装一层。

funcattrs
修复已经重命名的函数属性。比如 my_function.func_closure 会被转换为 my_function.__closure__。

future
移除 `from __future__ import new_feature` 语句。

getcwdu
将 `os.getcwdu()` 重命名为*os.getcwd()*。

has_key
将 `dict.has_key(key)` 转换为 `key in dict`。

idioms
这是一个可选的修复器，会进行多种转换，将 Python 代码变成更加常见的写法。类似 `type(x) is SomeClass` 和 `type(x) == SomeClass` 的类型对比会被转换成 `isinstance(x, SomeClass)`。`while 1` 转换成 `while True`。这个修复器还会在合适的地方使用*sorted()* 函数。举个例子，这样的代码块：

```
L = list(some_iterable)
L.sort()
```

会被转换为：

```
L = sorted(some_iterable)
```

import
检测 sibling imports，并将其转换成相对 import。

imports
处理标准库模块的重命名。

imports2
处理标准库中其他模块的重命名。这个修复器由于一些技术上的限制，因此和*imports* 拆分开了。

input
将 `input(prompt)` 转换为 `eval(input(prompt))`。

intern
将 `intern()` 转换为*sys.intern()*。

isinstance
修复*isinstance()* 函数第二个实参中重复的类型。举例来说，`isinstance(x, (int, int))` 会转换为 `isinstance(x, int)`，`isinstance(x, (int, float, int))` 会转换为 `isinstance(x, (int, float))`。

itertools_imports
移除 `itertools.ifilter()`，`itertools.izip()` 以及 `itertools.imap()` 的 import。对 `itertools.ifilterfalse()` 的 import 也会替换成*itertools.filterfalse()*。

itertools
修改 `itertools.ifilter()`，`itertools.izip()` 和 `itertools.imap()` 的调用为对应的内建实现。`itertools.ifilterfalse()` 会替换成*itertools.filterfalse()*。

long
将 `long` 重命名为*int*。

map
用*list* 包装*map()*。同时也会将 `map(None, x)` 替换为 `list(x)`。使用 `from future_builtins import map` 禁用这个修复器。

metaclass
将老的元类语法（类体中的 `__metaclass__ = Meta`）替换为新的（`class X(metaclass=Meta)`）。

methodattrs
修复老的方法属性名。例如 `meth.im_func` 会被转换为 `meth.__func__`。

ne
转换老的不等语法，将 <> 转为 !=。

next
将迭代器的 next() 方法调用转为*next()* 函数。也会将*next()* 方法重命名为*__next__()*。

nonzero
Renames definitions of methods called __nonzero__() to __bool__().

numliterals
将八进制字面量转为新的语法。

operator
将*operator* 模块中的许多方法调用转为其他的等效函数调用。如果有需要，会添加适当的 import 语句，比如 import collections.abc。有以下转换映射：

從	到
operator.isCallable(obj)	callable(obj)
operator.sequenceIncludes(obj)	operator.contains(obj)
operator.isSequenceType(obj)	isinstance(obj, collections.abc.Sequence)
operator.isMappingType(obj)	isinstance(obj, collections.abc.Mapping)
operator.isNumberType(obj)	isinstance(obj, numbers.Number)
operator.repeat(obj, n)	operator.mul(obj, n)
operator.irepeat(obj, n)	operator.imul(obj, n)

paren
在列表生成式中增加必须的括号。例如将 [x for x in 1, 2] 转换为 [x for x in (1, 2)]。

print
将 print 语句转换为*print()* 函数。

raise
将 raise E, V 转换为 raise E(V)，将 raise E, V, T 转换为 raise E(V).with_traceback(T)。如果 E 是元组，这样的转换是不正确的，因为用元组代替异常的做法在 3.0 中已经移除了。

raw_input
将 raw_input() 转换为*input()*。

reduce
将 reduce() 转换为*functools.reduce()*。

reload
将 reload() 转换为*importlib.reload()*。

renames
将 sys.maxint 转换为*sys.maxsize*。

repr
将反引号 repr 表达式替换为*repr()* 函数。

set_literal
将*set* 构造函数替换为 set literals 写法。这个修复器是可选的。

standarderror
将 StandardError 重命名为*Exception*。

sys_exc
将弃用的 sys.exc_value, sys.exc_type, sys.exc_traceback 替换为*sys.exc_info()* 的用法。

throw
修复生成器的 throw() 方法的 API 变更。

tuple_params
移除隐式的元组参数解包。这个修复器会插入临时变量。

types
修复*type* 模块中一些成员的移除引起的代码问题。

unicode
将 unicode 重命名为*str*。

urllib
将*urllib* 和 urllib2 重命名为*urllib* 包。

ws_comma
移除逗号分隔的元素之间多余的空白。这个修复器是可选的。

xrange
将 xrange() 重命名为*range()*，并用*list* 包装原有的*range()*。

xreadlines
将 for x in file.xreadlines() 转换为 for x in file。

zip
用*list* 包装*zip()*。如果使用了 from future_builtins import zip 的话会禁用。

26.11.3 lib2to3 ——2to3 支持库

源代码： Lib/lib2to3/

3.10 版後已🄵用: Python 3.9 将切换到 PEG 解析器 (参见 **PEP 617**)，Python 3.10 可能会包含 lib2to3 的 LL(1) 解析器所不能解析的新语法。lib2to3 模块可能会在未来的 Python 版本中被移出标准库。请考虑使用第三方替代例如 LibCST 或 parso。

備🄵： *lib2to3* API 并不稳定，并可能在未来大幅修改。

26.12 test --- Python 回归测试包

備🄵： *test* 包只供 Python 内部使用。它的记录是为了让 Python 的核心开发者受益。我们不鼓励在 Python 标准库之外使用这个包，因为这里提到的代码在 Python 的不同版本之间可能会改变或被删除而不另行通知。

---

test 包包含了 Python 的所有回归测试，以及*test.support* 和 test.regrtest 模块。*test.support* 用于增强你的测试，而 test.regrtest 驱动测试套件。

test`包中每个名字以 ``test_` 开头的模块都是一个特定模块或功能的测试套件。所有新的测试应该使用*unittest* 或*doctest* 模块编写。一些旧的测试是使用“传统”的测试风格编写的，即比较打印出来的输出到 “sys.stdout“；这种测试风格被认为是过时的。

也参考:

模块*unittest* 编写 PyUnit 回归测试.

模組*doctest* 嵌入到文档字符串的测试。

26.12.1 为 test 包编写单元测试

使用*unittest* 模块的测试最好是遵循一些准则。其中一条是测试模块的名称要以 test_ 打头并以被测试模块的名称结尾。测试模块中的测试方法应当以 test_ 打头并以该方法所测试的内容的说明结尾。这很有必要因为这样测试驱动程序就会将这些方法识别为测试方法。此外，该方法不应当包括任何文档字符串。应当使用注释 (例如 # Tests function returns only True or False) 来为测试方法提供文档说明。这样做是因为文档字符串如果存在则会被打印出来因此无法指明正在运行哪个测试。

有一个基本模板经常会被使用:

```
import unittest
from test import support

class MyTestCase1(unittest.TestCase):

    # Only use setUp() and tearDown() if necessary

    def setUp(self):
        ... code to execute in preparation for tests ...

    def tearDown(self):
        ... code to execute to clean up after tests ...

    def test_feature_one(self):
        # Test feature one.
        ... testing code ...

    def test_feature_two(self):
        # Test feature two.
        ... testing code ...

    ... more test methods ...

class MyTestCase2(unittest.TestCase):
    ... same structure as MyTestCase1 ...

... more test classes ...

if __name__ == '__main__':
    unittest.main()
```

这种代码模式允许测试套件由 test.regrtest 运行，作为支持*unittest* CLI 的脚本单独运行，或者通过 python -m unittest CLI 来运行。

回归测试的目标是尝试破坏代码。这引出了一些需要遵循的准则:

- 测试套件应当测试所有的类、函数和常量。这不仅包括要向外界展示的外部 API 也包括“私有”的代码。

- 白盒测试（在编写测试时检查被测试的代码）是最推荐的。黑盒测试（只测试已发布的用户接口）因不够完整而不能确保所有边界和边缘情况都被测试到。
- 确保所有可能的值包括无效的值都被测试到。这能确保不仅全部的有效值都可被接受而且不适当的值也能被正确地处理。
- 消耗尽可能多的代码路径。测试发生分支的地方从而调整输入以确保通过代码采取尽可能多的不同路径。
- 为受测试的代码所发现的任何代码缺陷添加明确的测试。这将确保如果代码在将来被改变错误也不会再次出现。
- 确保在你的测试完成后执行清理（例如关闭并删除所有临时文件）。
- 如果某个测试依赖于操作系统上的特定条件那么要在尝试测试之前先验证该条件是否已存在。
- 尽可能少地导入模块并尽可能快地完成操作。这可以最大限度地减少测试的外部依赖性并且还可以最大限度地减少导入模块带来的附带影响所导致的异常行为。
- 尝试最大限度地重用代码。在某些情况下，测试结果会因使用不同类型的输入这样的小细节而变化。可通过一个指定输入的类来子类化一个基本测试类来最大限度地减少重复代码:

```
class TestFuncAcceptsSequencesMixin:

    func = mySuperWhammyFunction

    def test_func(self):
        self.func(self.arg)

class AcceptLists(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = [1, 2, 3]

class AcceptStrings(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = 'abc'

class AcceptTuples(TestFuncAcceptsSequencesMixin, unittest.TestCase):
    arg = (1, 2, 3)
```

  当使用这种模式时，请记住所有继承自*unittest.TestCase*的类都会作为测试来运行。上面例子中的 Mixin 类没有任何数据所以其本身是无法运行的，因此它不是继承自*unittest.TestCase*。

也参考:

测试驱动的开发 Kent Beck 所著的阐述在实现代码之前编写驱动的书。

26.12.2 使用命令行界面运行测试

通过使用 -m 选项*test* 包可以作为脚本运行以驱动 Python 的回归测试套件: **python -m test**。在内部，它使用 test.regrtest；之前 Python 版本所使用的 **python -m test.regrtest** 调用仍然有效。运行该脚本自身会自动开始运行*test* 包中的所有回归测试。它通过在包中查找所有名称以 test_ 打头的模块，导入它们，并在有 test_main() 函数时执行它或是在没有 test_main 时通过 unittest.TestLoader.loadTestsFromModule 载入测试。要执行的测试的名称也可以被传递给脚本。指定一个单独的回归测试 (**python -m test test_spam**) 将使输出最小化并且只打印测试通过或失败的消息。

直接运行*test* 将允许设置哪些资源可供测试使用。你可以通过使用 -u 命令行选项来做到这一点。指定 all 作为 -u 选项的值将启用所有可能的资源: **python -m test -uall**。如果只需要一项资源（这是更为常见的情况），可以在 all 之后加一个以逗号分隔的列表来指明不需要的资源。命令 **python -m test -uall,-audio,-largefile** 将运行*test* 并使用除 audio 和 largefile 资源之外的所有资源。要查看所有资源的列表和更多的命令行选项，请运行 **python -m test -h**。

另外一些执行回归测试的方式依赖于执行测试所在的系统平台。在 Unix 上，你可以在构建 Python 的最高层级目录中运行 **make test**。在 Windows 上，在你的 PCbuild 目录中执行 **rt.bat** 将运行所有的回归测试。

26.13 test.support --- 针对 Python 测试套件的工具

test.support 模块提供了对 Python 的回归测试套件的支持。

備🄵: *test.support* 不是一个公用模块。这篇文档是为了帮助 Python 开发者编写测试。此模块的 API 可能被改变而不顾及发行版本之间的向下兼容性问题。

此模块定义了以下异常:

exception test.support.**TestFailed**
当一个测试失败时将引发的异常。此异常已被弃用而应改用基于*unittest* 的测试以及*unittest.TestCase* 的断言方法。

exception test.support.**ResourceDenied**
unittest.SkipTest 的子类。当一个资源（例如网络连接）不可用时将被引发。由*requires()* 函数所引发。

test.support 模块定义了以下常量:

test.support.**verbose**
当启用详细输出时为 True。当需要有关运行中的测试的更详细信息时应当被选择。*verbose* 是由 test.regrtest 来设置的。

test.support.**is_jython**
如果所运行的解释器是 Jython 时为 True。

test.support.**is_android**
如果系统是 Android 时为 True。

test.support.**unix_shell**
如果系统不是 Windows 时则为 shell 的路径；否则为 None。

test.support.**FS_NONASCII**
一个可通过*os.fsencode()* 编码的非 ASCII 字符。

test.support.**TESTFN**
设置为一个可以安全地用作临时文件名的名称。任何被创建的临时文件都应当被关闭和撤销链接（移除）。

test.support.**TESTFN_UNICODE**
设置为用于临时文件的非 ASCII 名称。

test.support.**TESTFN_ENCODING**
Set to *sys.getfilesystemencoding()*.

test.support.**TESTFN_UNENCODABLE**
设置为一个应当在严格模式下不可使用文件系统编码格式来编码的文件名（str 类型）。如果无法生成这样的文件名则可以为 None。

test.support.**TESTFN_UNDECODABLE**
设置为一个应当在严格模式下不可使用文件系统编码格式来编码的文件名（bytes 类型）。如果无法生成这样的文件名则可以为 None。

test.support.**TESTFN_NONASCII**
Set to a filename containing the *FS_NONASCII* character.

test.support.**LOOPBACK_TIMEOUT**
使用网络服务器监听网络本地环回接口如 127.0.0.1 的测试的以秒为单位的超时值。

该超时长到足以防止测试失败：它要考虑客户端和服务器可能会在不同线程甚至不同进程中运行。

该超时应当对于*socket.socket* 的*connect()*, *recv()* 和*send()* 方法都足够长。

其默认值为 5 秒。

参见*INTERNET_TIMEOUT*。

test.support.**INTERNET_TIMEOUT**
Timeout in seconds for network requests going to the Internet.

The timeout is short enough to prevent a test to wait for too long if the Internet request is blocked for whatever reason.

通常使用*INTERNET_TIMEOUT* 的超时不应该将测试标记为失败，而是跳过测试：参见*transient_internet()*。

其默认值是 1 分钟。

参见*LOOPBACK_TIMEOUT*。

test.support.**SHORT_TIMEOUT**
如果测试耗时“太长”而要将测试标记为失败的以秒为单位的超时值。

该超时值取决于 regrtest --timeout 命令行选项。

如果一个使用*SHORT_TIMEOUT* 的测试在慢速 buildbots 上开始随机失败，请使用*LONG_TIMEOUT* 来代替。

其默认值为 30 秒。

test.support.**LONG_TIMEOUT**
用于检测测试何时挂起的以秒为单位的超时值。

它的长度足够在最慢的 Python buildbot 上降低测试失败的风险。如果测试耗时“过长”也不应当用它将该测试标记为失败。此超时值依赖于 regrtest --timeout 命令行选项。

其默认值为 5 分钟。

另请参见*LOOPBACK_TIMEOUT*, *INTERNET_TIMEOUT* 和*SHORT_TIMEOUT*。

test.support.**SAVEDCWD**
设置为*os.getcwd()*。

test.support.**PGO**
当测试对 PGO 没有用处时设置是否要跳过测试。

test.support.**PIPE_MAX_SIZE**
一个通常大于下层 OS 管道缓冲区大小的常量，以产生写入阻塞。

test.support.**SOCK_MAX_SIZE**
一个通常大于下层 OS 套接字缓冲区大小的常量，以产生写入阻塞。

test.support.**TEST_SUPPORT_DIR**
设为包含*test.support* 的最高层级目录。

test.support.**TEST_HOME_DIR**
设为 test 包的最高层级目录。

test.support.**TEST_DATA_DIR**
设为 test 包中的 data 目录。

test.support.**MAX_Py_ssize_t**
设为大内存测试的*sys.maxsize*。

test.support.**max_memuse**
通过*set_memlimit()* 设为针对大内存测试的内存限制。由*MAX_Py_ssize_t* 来限制。

test.support.**real_max_memuse**
通过*set_memlimit()* 设为针对大内存测试的内存限制。不受*MAX_Py_ssize_t* 的限制。

test.support.**MISSING_C_DOCSTRINGS**
Return `True` if running on CPython, not on Windows, and configuration not set with `WITH_DOC_STRINGS`.

test.support.**HAVE_DOCSTRINGS**
Check for presence of docstrings.

test.support.**TEST_HTTP_URL**
定义用于网络测试的韧性 HTTP 服务器的 URL。

test.support.**ALWAYS_EQ**
等于任何对象的对象。用于测试混合类型比较。

test.support.**NEVER_EQ**
不等于任何对象的对象 (即使是*ALWAYS_EQ*)。用于测试混合类型比较。

test.support.**LARGEST**
大于任何对象的对象（除了其自身）。用于测试混合类型比较。

test.support.**SMALLEST**
小于任何对象的对象（除了其自身）。用于测试混合类型比较。Used to test mixed type comparison.

test.support 模块定义了以下函数:

test.support.**forget**(*module_name*)
从 `sys.modules` 移除名为 *module_name* 的模块并删除该模块的已编译字节码文件。

test.support.**unload**(*name*)
从 `sys.modules` 中删除 *name*。

test.support.**unlink**(*filename*)
Call *os.unlink()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.

test.support.**rmdir**(*filename*)
Call *os.rmdir()* on *filename*. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the file.

test.support.**rmtree**(*path*)
Call *shutil.rmtree()* on *path* or call *os.lstat()* and *os.rmdir()* to remove a path and its contents. On Windows platforms, this is wrapped with a wait loop that checks for the existence of the files.

test.support.**make_legacy_pyc**(*source*)
将 **PEP 3147/PEP 488** pyc 文件移至旧版 pyc 位置并返回该旧版 pyc 文件的文件系统路径。*source* 的值是源文件的文件系统路径。它不必真实存在，但是 PEP 3147/488 pyc 文件必须存在。

test.support.**is_resource_enabled**(*resource*)
如果 *resource* 已启用并可用则返回 `True`。可用资源列表只有当 `test.regrtest` 正在执行测试时才会被设置。

test.support.**python_is_optimized**()
如果 Python 编译未使用 `-O0` 或 `-Og` 则返回 `True`。

test.support.**with_pymalloc**()
返回 `_testcapi.WITH_PYMALLOC`。

test.support.**requires**(*resource*, *msg=None*)
如果 *resource* 不可用则引发*ResourceDenied*。如果该异常被引发则 *msg* 为传给*ResourceDenied*的参数。如果被 __name__ 为 '__main__' 的函数调用则总是返回 True。在测试由 test.regrtest 执行时使用。

test.support.**system_must_validate_cert**(*f*)
Raise *unittest.SkipTest* on TLS certification validation failures.

test.support.**sortdict**(*dict*)
返回 *dict* 按键排序的 repr。

test.support.**findfile**(*filename*, *subdir=None*)
返回名为 *filename* 的文件的路径。如果未找到匹配结果则返回 *filename*。这并不等于失败因为它也算是该文件的路径。

设置 *subdir* 指明要用来查找文件的相对路径而不是直接在路径目录中查找。

test.support.**create_empty_file**(*filename*)
创建一个名为 *filename* 的空文件。如果文件已存在，则清空其内容。

test.support.**fd_count**()
统计打开的文件描述符数量。

test.support.**match_test**(*test*)
Match *test* to patterns set in *set_match_tests()*.

test.support.**set_match_tests**(*patterns*)
Define match test with regular expression *patterns*.

test.support.**run_unittest**(**classes*)
执行传给函数的*unittest.TestCase* 子类。此函数会扫描类中带有 test_ 前缀的方法并单独执行这些测试。

将字符串作为形参传递也是合法的；这些形参应为 sys.modules 中的键。每个被关联的模块将由 unittest.TestLoader.loadTestsFromModule() 执行扫描。这通常出现在以下 test_main() 函数中:

```
def test_main():
    support.run_unittest(__name__)
```

这将运行在指定模块中定义的所有测试。

test.support.**run_doctest**(*module*, *verbosity=None*, *optionflags=0*)
在给定的 *module* 上运行*doctest.testmod()*。返回 (failure_count, test_count)。

如果 *verbosity* 为 None，*doctest.testmod()* 运行时的消息详细程度将设为*verbose*。否则，运行时的消息详细程度将设为 None。*optionflags* 将作为 optionflags 传给*doctest.testmod()*。

test.support.**setswitchinterval**(*interval*)
将*sys.setswitchinterval()* 设为给定的 *interval*。请为 Android 系统定义一个最小间隔以防止系统挂起。

test.support.**check_impl_detail**(***guards*)
Use this check to guard CPython's implementation-specific tests or to run them only on the implementations guarded by the arguments:

```
check_impl_detail()               # Only on CPython (default).
check_impl_detail(jython=True)    # Only on Jython.
check_impl_detail(cpython=False)  # Everywhere except CPython.
```

test.support.**check_warnings**(**filters*, *quiet=True*)
一个用于*warnings.catch_warnings()* 以更容易地测试特定警告是否被正确引发的便

捷包装器。它大致等价于调用 warnings.catch_warnings(record=True) 并将*warnings.simplefilter()* 设为 always 并附带自动验证已记录结果的选项。

check_warnings 接受 ("message regexp", WarningCategory) 形式的 2 元组作为位置参数。如果提供了一个或多个 *filters*，或者如果可选的关键字参数 *quiet* 为 False，则它会检查确认警告是符合预期的：每个已指定的过滤器必须匹配至少一个被包围的代码或测试失败时引发的警告，并且如果有任何未能匹配已指定过滤器的警告被引发则测试将失败。要禁用这些检查中的第一项，请将 *quiet* 设为 True。

如果未指定任何参数，则默认为:

```
check_warnings(("", Warning), quiet=True)
```

在此情况下所有警告都会被捕获而不会引发任何错误。

在进入该上下文管理器时，将返回一个 WarningRecorder 实例。来自*catch_warnings()* 的下层警告列表可通过该记录器对象的*warnings* 属性来访问。作为一个便捷方式，该对象中代表最近的警告的属性也可通过该记录器对象来直接访问（参见以下示例）。如果未引发任何警告，则在其他情况下预期代表一个警告的任何对象属性都将返回 None。

该记录器对象还有一个 reset() 方法，该方法会清空警告列表。

该上下文管理器被设计为像这样来使用:

```
with check_warnings(("assertion is always true", SyntaxWarning),
                    ("", UserWarning)):
    exec('assert(False, "Hey!")')
    warnings.warn(UserWarning("Hide me!"))
```

在此情况下如果两个警告都未被引发，或是引发了其他的警告，则*check_warnings()* 将会引发一个错误。

当一个测试需要更深入地查看这些警告，而不是仅仅检查它们是否发生时，可以使用这样的代码:

```
with check_warnings(quiet=True) as w:
    warnings.warn("foo")
    assert str(w.args[0]) == "foo"
    warnings.warn("bar")
    assert str(w.args[0]) == "bar"
    assert str(w.warnings[0].args[0]) == "foo"
    assert str(w.warnings[1].args[0]) == "bar"
    w.reset()
    assert len(w.warnings) == 0
```

在这里所有的警告都将被捕获，而测试代码会直接测试被捕获的警告。

3.2 版更變: 新增可选参数 *filters* 和 *quiet*。

test.support.**check_no_resource_warning**(*testcase*)
检测是否没有任何*ResourceWarning* 被引发的上下文管理器。你必须在该上下文管理器结束之前移除可能发出*ResourceWarning* 的对象。

test.support.**set_memlimit**(*limit*)
针对大内存测试设置*max_memuse* 和*real_max_memuse* 的值。

test.support.**record_original_stdout**(*stdout*)
存放来自 *stdout* 的值。它会在回归测试开始时处理 stdout。

test.support.**get_original_stdout**()
返回*record_original_stdout()* 所设置的原始 stdout 或者如果未设置则为 sys.stdout。

test.support.**args_from_interpreter_flags**()
返回在 sys.flags 和 sys.warnoptions 中重新产生当前设置的命令行参数列表。

test.support.**optim_args_from_interpreter_flags**()
返回在 sys.flags 中重新产生当前优化设置的命令行参数列表。

test.support.**captured_stdin**()
test.support.**captured_stdout**()
test.support.**captured_stderr**()
使用*io.StringIO* 对象临时替换指定流的上下文管理器。

使用输出流的示例:

```
with captured_stdout() as stdout, captured_stderr() as stderr:
    print("hello")
    print("error", file=sys.stderr)
assert stdout.getvalue() == "hello\n"
assert stderr.getvalue() == "error\n"
```

使用输入流的示例:

```
with captured_stdin() as stdin:
    stdin.write('hello\n')
    stdin.seek(0)
    # call test code that consumes from sys.stdin
    captured = input()
self.assertEqual(captured, "hello")
```

test.support.**temp_dir**(*path=None*, *quiet=False*)
一个在 *path* 上创建临时目录并输出该目录的上下文管理器。

如果 *path* 为 None，则会使用*tempfile.mkdtemp()* 来创建临时目录。如果 *quiet* 为 False，则该上下文管理器在发生错误时会引发一个异常。在其他情况下，如果 *path* 已被指定并且无法创建，则只会发出一个警告。

test.support.**change_cwd**(*path*, *quiet=False*)
一个临时性地将当前工作目录改为 *path* 并输出该目录的上下文管理器。

如果 *quiet* 为 False，此上下文管理器将在发生错误时引发一个异常。在其他情况下，它将只发出一个警告并将当前工作目录保持原状。

test.support.**temp_cwd**(*name='tempcwd'*, *quiet=False*)
一个临时性地创建新目录并改变当前工作目录（CWD）的上下文管理器。

临时性地改变当前工作目录之前此上下文管理器会在当前目录下创建一个名为 *name* 的临时目录。如果 *name* 为 None，则会使用*tempfile.mkdtemp()* 创建临时目录。

如果 *quiet* 为 False 并且无法创建或修改 CWD，则会引发一个错误。在其他情况下，只会引发一个警告并使用原始 CWD。

test.support.**temp_umask**(*umask*)
一个临时性地设置进程掩码的上下文管理器。

test.support.**disable_faulthandler**()
A context manager that replaces sys.stderr with sys.__stderr__.

test.support.**gc_collect**()
强制收集尽可能多的对象。这是有必要的因为垃圾回收器并不能保证及时回收资源。这意味着 __del__ 方法的调用可能会晚于预期而弱引用的存活长于预期。

test.support.**disable_gc**()
A context manager that disables the garbage collector upon entry and reenables it upon exit.

test.support.**swap_attr**(*obj*, *attr*, *new_val*)
上下文管理器用一个新对象来交换一个属性。

用法:

```
with swap_attr(obj, "attr", 5):
    ...
```

这把 obj.attr 设为 5 并在 with 语句块内保持，在语句块结束时恢复原值。如果 attr 不存在于 obj 中，它将被创建并在语句块结束时被删除。

旧值 (或者如果不存在旧值则为 None) 将被赋给"as" 子句的目标，如果存在子句的话。

test.support.**swap_item**(*obj*, *attr*, *new_val*)
上下文件管理器用一个新对象来交换一个条目。

用法:

```
with swap_item(obj, "item", 5):
    ...
```

这将把 obj["item"] 设为 5 并在 with 语句块内保持，在语句块结束时恢复旧值。如果 item 不存在于 obj 中，它将被创建并在语句块结束时被删除。

旧值 (或者如果不存在旧值则为 None) 将被赋给"as" 子句的目标，如果存在子句的话。

test.support.**print_warning**(*msg*)
打印一个警告到*sys.__stderr__*。将消息格式化为: f"Warning -- {msg}"。如果 *msg* 包含多行，则为每行添加 "Warning -- " 前缀。

3.9 版新加入.

test.support.**wait_process**(*pid*, *, *exitcode*, *timeout=None*)
等待直到进程 *pid* 结束并检查进程退出代码是否为 *exitcode*。

如果进程退出代码不等于 *exitcode* 则引发*AssertionError*。

如果进程运行时长超过 *timeout* 秒 (默认为*SHORT_TIMEOUT*)，则杀死进程并引发*AssertionError*。超时特性在 Windows 上不可用。

3.9 版新加入.

test.support.**wait_threads_exit**(*timeout=60.0*)
等待直到 with 语句中所有已创建线程退出的上下文管理器。

test.support.**start_threads**(*threads*, *unlock=None*)
Context manager to start *threads*. It attempts to join the threads upon exit.

test.support.**calcobjsize**(*fmt*)
Return *struct.calcsize()* for nP{fmt}0n or, if gettotalrefcount exists, 2PnP{fmt}0P.

test.support.**calcvobjsize**(*fmt*)
Return *struct.calcsize()* for nPn{fmt}0n or, if gettotalrefcount exists, 2PnPn{fmt}0P.

test.support.**checksizeof**(*test*, *o*, *size*)
对于测试用例 *test*，断言 *o* 的 sys.getsizeof 加 GC 头的大小等于 *size*。

test.support.**can_symlink**()
如果操作系统支持符号链接则返回 True，否则返回 False。

test.support.**can_xattr**()
如果操作系统支持 xattr 支返回 True，否则返回 False。

@test.support.**skip_unless_symlink**
一个用于运行需要符号链接支持的测试的装饰器。

@test.support.**skip_unless_xattr**
一个用于运行需要 xattr 支持的测试的装饰器。

@test.support.**anticipate_failure**(*condition*)
一个有条件地用*unittest.expectedFailure()* 来标记测试的装饰器。任何对此装饰器的使用都应当具有标识相应追踪事项的有关联注释。

@test.support.**run_with_locale**(*catstr*, **locales*)
一个在不同语言区域下运行函数的装饰器，并在其结束后正确地重置语言区域。*catstr* 是字符串形式的语言区域类别 (例如 "LC_ALL")。传入的 *locales* 将依次被尝试，并将使用第一个有效的语言区域。

@test.support.**run_with_tz**(*tz*)
一个在指定时区下运行函数的装饰器，并在其结束后正确地重置时区。

@test.support.**requires_freebsd_version**(**min_version*)
Decorator for the minimum version when running test on FreeBSD. If the FreeBSD version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_linux_version**(**min_version*)
Decorator for the minimum version when running test on Linux. If the Linux version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_mac_version**(**min_version*)
Decorator for the minimum version when running test on macOS. If the macOS version is less than the minimum, raise *unittest.SkipTest*.

@test.support.**requires_IEEE_754**
用于在非 non-IEEE 754 平台上跳过测试的装饰器。

@test.support.**requires_zlib**
用于当*zlib* 不存在时跳过测试的装饰器。

@test.support.**requires_gzip**
用于当*gzip* 不存在时跳过测试的装饰器。

@test.support.**requires_bz2**
用于当*bz2* 不存在时跳过测试的装饰器。

@test.support.**requires_lzma**
用于当*lzma* 不存在时跳过测试的装饰器。

@test.support.**requires_resource**(*resource*)
用于当 *resource* 不可用时跳过测试的装饰器。

@test.support.**requires_docstrings**
用于仅当*HAVE_DOCSTRINGS* 时才运行测试的装饰器。

@test.support.**cpython_only**(*test*)
表示仅适用于 CPython 的测试的装饰器。

@test.support.**impl_detail**(*msg=None*, ***guards*)
用于在 *guards* 上发起调用*check_impl_detail()* 的装饰器。如果调用返回 False，则使用 *msg* 作为跳过测试的原因。

@test.support.**no_tracing**(*func*)
用于在测试期间临时关闭追踪的装饰器。

@test.support.**refcount_test**(*test*)
用于涉及引用计数的测试的装饰器。如果测试不是由 CPython 运行则该装饰器不会运行测试。在测试期间会取消设置任何追踪函数以由追踪函数导致的意外引用计数。

@test.support.**reap_threads**(*func*)
用于确保即使测试失败线程仍然会被清理的装饰器。

@test.support.**bigmemtest**(*size*, *memuse*, *dry_run=True*)
用于大内存测试的装饰器。

size 是测试所请求的大小（以任意的，由测试解读的单位。）*memuse* 是测试的每单元字节数，或是对它的良好估计。例如，一个需要两个字节缓冲区，每个缓冲区 4 GiB，则可以用 `@bigmemtest(size=_4G, memuse=2)` 来装饰。

size 参数通常作为额外参数传递给被测试的方法。如果 *dry_run* 为 `True`，则传给测试方法的值可能少于所请求的值。如果 *dry_run* 为 `False`，则意味着当当未指定 `-M` 时测试将不支持虚拟运行。

@test.support.**bigaddrspacetest**(*f*)
Decorator for tests that fill the address space. *f* is the function to wrap.

test.support.**make_bad_fd**()
通过打开并关闭临时文件来创建一个无效的文件描述符，并返回其描述器。

test.support.**check_syntax_error**(*testcase*, *statement*, *errtext=''*, *, *lineno=None*, *offset=None*)
用于通过尝试编译 *statement* 来测试 *statement* 中的语法错误。*testcase* 是测试的*unittest* 实例。*errtext* 是应当匹配所引发的*SyntaxError* 的字符串表示形式的正则表达式。如果 *lineno* 不为 `None`，则与异常所在的行进行比较。如果 *offset* 不为 `None`，则与异常的偏移量进行比较。

test.support.**check_syntax_warning**(*testcase*, *statement*, *errtext=''*, *, *lineno=1*, *offset=None*)
用于通过尝试编译 *statement* 来测试 *statement* 中的语法警告。还会测试*SyntaxWarning* 是否只发出了一次，以及它在转成错误时是否将被转换为*SyntaxError*。*testcase* 是用于测试的*unittest* 实例。*errtext* 是应当匹配所发出的*SyntaxWarning* 以及所引发的*SyntaxError* 的字符串表示形式的正则表达式。如果 *lineno* 不为 `None`，则与警告和异常所在的行进行比较。如果 *offset* 不为 `None`，则与异常的偏移量进行比较。

3.8 版新加入.

test.support.**open_urlresource**(*url*, **args*, ***kw*)
打开 *url*。如果打开失败，则引发*TestFailed*。

test.support.**import_module**(*name*, *deprecated=False*, *, *required_on()*)
此函数会导入并返回指定名称的模块。不同于正常的导入，如果模块无法被导入则此函数将引发*unittest.SkipTest*。

如果 *deprecated* 为 `True` 则在此导入操作期间模块和包的弃用消息会被屏蔽。如果某个模块在特定平台上是必需的而在其他平台上是可选的，请为包含平台前缀的可迭代对象设置 *required_on*，此对象将与*sys.platform* 进行比对。

3.1 版新加入.

test.support.**import_fresh_module**(*name*, *fresh=()*, *blocked=()*, *deprecated=False*)
此函数会在执行导入之前通过从 `sys.modules` 移除指定模块来导入并返回指定 Python 模块的新副本。请注意这不同于 `reload()`，原来的模块不会受到此操作的影响。

fresh 是包含在执行导入之前还要从 `sys.modules` 缓存中移除的附加模块名称的可迭代对象。

blocked 是包含模块名称的可迭代对象，导入期间在模块缓存中它会被替换为 `None` 以确保尝试导入将引发*ImportError*。

指定名称的模块以及任何在 *fresh* 和 *blocked* 形参中指明的模块会在开始导入之前被保存并在全新导入完成时被重新插入到 `sys.modules` 中。

如果 *deprecated* 为 `True` 则在此导入操作期间模块和包的弃用消息会被屏蔽。

如果指定名称的模块无法被导入则此函数将引发*ImportError*.

用法示例:

```
# Get copies of the warnings module for testing without affecting the
# version being used by the rest of the test suite. One copy uses the
# C implementation, the other is forced to use the pure Python fallback
# implementation
py_warnings = import_fresh_module('warnings', blocked=['_warnings'])
c_warnings = import_fresh_module('warnings', fresh=['_warnings'])
```

3.1 版新加入.

test.support.**modules_setup**()
返回*sys.modules* 的副本。

test.support.**modules_cleanup**(*oldmodules*)
移除 *oldmodules* 和 encodings 以外的模块以保留内部缓冲区。

test.support.**threading_setup**()
返回当前线程计数和悬空线程的副本。

test.support.**threading_cleanup**(**original_values*)
清理未在 *original_values* 中指定的线程。被设计为如果有一个测试在后台离开正在运行的线程时会发出警告。

test.support.**join_thread**(*thread*, *timeout=30.0*)
在 *timeout* 秒之内合并一个 *thread*。如果线程在 *timeout* 秒后仍然存活则引发*AssertionError*。

test.support.**reap_children**()
只要有子进程启动就在 test_main 的末尾使用此函数。这将有助于确保没有多余的子进程（僵尸）存在占用资源并在查找引用泄漏时造成问题。

test.support.**get_attribute**(*obj*, *name*)
获取一个属性，如果引发了*AttributeError* 则会引发*unittest.SkipTest*。

test.support.**catch_threading_exception**()
使用*threading.excepthook()* 来捕获*threading.Thread* 异常的上下文管理器。

Attributes set when an exception is caught:

- exc_type
- exc_value
- exc_traceback
- thread

参见*threading.excepthook()* 文档。

这些属性在上下文管理器退出时将被删除。

用法:

```
with support.catch_threading_exception() as cm:
    # code spawning a thread which raises an exception
    ...

    # check the thread exception, use cm attributes:
    # exc_type, exc_value, exc_traceback, thread
    ...

# exc_type, exc_value, exc_traceback, thread attributes of cm no longer
# exists at this point
# (to avoid reference cycles)
```

3.8 版新加入.

test.support.**catch_unraisable_exception**()
使用*sys.unraisablehook()* 来捕获不可引发的异常的上下文管理器。

存储异常值 (cm.unraisable.exc_value) 会创建一个引用循环。引用循环将在上下文管理器退出时被显式地打破。

存储对象 (cm.unraisable.object) 如果被设置为一个正在最终化的对象则可以恢复它。退出上下文管理器将清除已存在对象。

用法:

```
with support.catch_unraisable_exception() as cm:
    # code creating an "unraisable exception"
    ...

    # check the unraisable exception: use cm.unraisable
    ...

# cm.unraisable attribute no longer exists at this point
# (to break a reference cycle)
```

3.8 版新加入.

test.support.**load_package_tests**(*pkg_dir*, *loader*, *standard_tests*, *pattern*)
在测试包中使用的*unittest* load_tests 协议的通用实现。*pkg_dir* 是包的根目录；*loader*, *standard_tests* 和 *pattern* 是 load_tests 所期望的参数。在简单的情况下，测试包的 __init__.py 可以是下面这样的:

```
import os
from test.support import load_package_tests

def load_tests(*args):
    return load_package_tests(os.path.dirname(__file__), *args)
```

test.support.**fs_is_case_insensitive**(*directory*)
如果 *directory* 的文件系统对大小写敏感则返回 True。

test.support.**detect_api_mismatch**(*ref_api*, *other_api*, *, *ignore=()*)
返回未在 *other_api* 中找到的 *ref_api* 的属性、函数或方法的集合，除去在 *ignore* 中指明的要在这个检查中忽略的已定义条目列表。

在默认情况下这将跳过以'_' 打头的私有属性但包括所有魔术方法，即以'__' 打头和结尾的方法。

3.5 版新加入.

test.support.**patch**(*test_instance*, *object_to_patch*, *attr_name*, *new_value*)
用 *new_value* 重载 *object_to_patch.attr_name*。并向 *test_instance* 添加清理过程以便为 *attr_name* 恢复 *object_to_patch*。*attr_name* 应当是 *object_to_patch* 的一个有效属性。

test.support.**run_in_subinterp**(*code*)
在子解释器中运行 *code*。如果启用了*tracemalloc* 则会引发*unittest.SkipTest*。

test.support.**check_free_after_iterating**(*test*, *iter*, *cls*, *args=()*)
Assert that *iter* is deallocated after iterating.

test.support.**missing_compiler_executable**(*cmd_names=[]*)
检查在 *cmd_names* 中列出名称的或者当 *cmd_names* 为空时所有的编译器可执行文件是否存在并返回第一个丢失的可执行文件或者如果未发现任何丢失则返回 None。

test.support.**check__all__**(*test_case*, *module*, *name_of_module=None*, *extra=()*, *blacklist=()*)
断言 *module* 的 __all__ 变量包含全部公有名称。

模块的公有名称（它的 API）是根据它们是否符合公有名称惯例并在 *module* 中被定义来自动检测的。

name_of_module 参数可以（用字符串或元组的形式）指定一个 API 可以被定义为什么模块以便被检测为一个公共 API。一种这样的情况会在 *module* 从其他模块，可能是一个 C 后端 (如 csv 和它的 _csv) 导入其公共 API 的某一组成部分时发生。

extra 参数可以是一个在其他情况下不会被自动检测为"public" 的名称集合，例如没有适当 __module__ 属性的对象。如果提供该参数，它将被添加到自动检测到的对象中。

The *blacklist* argument can be a set of names that must not be treated as part of the public API even though their names indicate otherwise.

用法示例:

```
import bar
import foo
import unittest
from test import support

class MiscTestCase(unittest.TestCase):
    def test__all__(self):
        support.check__all__(self, foo)

class OtherTestCase(unittest.TestCase):
    def test__all__(self):
        extra = {'BAR_CONST', 'FOO_CONST'}
        blacklist = {'baz'}  # Undocumented name.
        # bar imports part of its API from _bar.
        support.check__all__(self, bar, ('bar', '_bar'),
                             extra=extra, blacklist=blacklist)
```

3.6 版新加入.

test.support.**adjust_int_max_str_digits**(*max_digits*)
This function returns a context manager that will change the global *sys.set_int_max_str_digits()* setting for the duration of the context to allow execution of test code that needs a different limit on the number of digits when converting between an integer and string.

3.9.14 版新加入.

test.support 模块定义了以下的类:

class test.support.**TransientResource**(*exc*, ***kwargs*)
Instances are a context manager that raises *ResourceDenied* if the specified exception type is raised. Any keyword arguments are treated as attribute/value pairs to be compared against any exception raised within the with statement. Only if all pairs match properly against attributes on the exception is *ResourceDenied* raised.

class test.support.**EnvironmentVarGuard**
用于临时性地设置或取消设置环境变量的类。其实例可被用作上下文管理器并具有完整的字典接口用来查询/修改下层的 os.environ。在从上下文管理器退出之后所有通过此实例对环境变量进行的修改都将被回滚。

3.1 版更變: 增加了字典接口。

EnvironmentVarGuard.**set**(*envvar*, *value*)
临时性地将环境变量 envvar 的值设为 value。

EnvironmentVarGuard.**unset**(*envvar*)
临时性地取消设置环境变量 envvar。

class test.support.**SuppressCrashReport**
一个用于在预期会使子进程崩溃的测试时尽量防止弹出崩溃对话框的上下文管理器。

在 Windows 上，它会使用 SetErrorMode 来禁用 Windows 错误报告对话框。

在 UNIX 上，会使用*resource.setrlimit()* 来将*resource.RLIMIT_CORE* 的软限制设为 0 以防止创建核心转储文件。

在这两个平台上，旧值都将被 __exit__() 恢复。

class test.support.**CleanImport**(**module_names*)
A context manager to force import to return a new module reference. This is useful for testing module-level behaviors, such as the emission of a DeprecationWarning on import. Example usage:

```
with CleanImport('foo'):
    importlib.import_module('foo')  # New reference.
```

class test.support.**DirsOnSysPath**(**paths*)
A context manager to temporarily add directories to sys.path.

这将创建*sys.path* 的一个副本，添加作为位置参数传入的任何目录，然后在上下文结束时将*sys.path* 还原到副本的设置。

请注意该上下文管理器代码块中 所有对*sys.path* 的修改，包括对象的替换，都将在代码块结束时被还原。

class test.support.**SaveSignals**
用于保存和恢复由 Python 句柄的所注册的信号处理句柄。

class test.support.**Matcher**

matches(*self*, *d*, ***kwargs*)
尝试对单个字典与所提供的参数进行匹配。

match_value(*self*, *k*, *dv*, *v*)
尝试对单个已存储值 (*dv*) 与所提供的值 (*v*) 进行匹配。

class test.support.**WarningsRecorder**
用于为单元测试记录警告的类。请参阅以上*check_warnings()* 的文档来了解详情。

class test.support.**BasicTestRunner**

run(*test*)
运行 *test* 并返回结果。

class test.support.**FakePath**(*path*)
简单的*path-like object*。它实现了 __fspath__() 方法，该方法将返回 *path* 参数。如果 *path* 为一个异常，它将在 __fspath__() 中被引发。

26.14 test.support.socket_helper --- 用于套接字测试的工具

test.support.socket_helper 模块提供了对套接字测试的支持。

3.9 版新加入.

test.support.socket_helper.**IPV6_ENABLED**
设置为 True 如果主机打开 IPv6, 否则 False .

test.support.socket_helper.**find_unused_port**(*family=socket.AF_INET*, *socktype=socket.SOCK_STREAM*)
返回一个应当适合绑定的未使用端口。这是通过创建一个与 sock 形参相同协议族和类型的临时套接字来达成的 (默认为*AF_INET*, *SOCK_STREAM*)，并将其绑定到指定的主机地址 (默认为 0.0.0.0) 并将端口设为 0，以从 OS 引出一个未使用的瞬时端口。这个临时套接字随后将被关闭并删除，然后返回该瞬时端口。

这个方法或*bind_port()* 应当被用于任何在测试期间需要绑定到特定端口的测试。具体使用哪个取决于调用方代码是否会创建 Python 套接字，或者是否需要在构造器中提供或向外部程序提供未使用的端口（例如传给 openssl 的 s_server 模式的 -accept 参数）。在可能的情况下将总是优先使用*bind_port()* 而非*find_unused_port()*。不建议使用硬编码的端口因为将使测试的多个实例无法同时运行，这对 buildbot 来说是个问题。

test.support.socket_helper.**bind_port**(*sock*, *host=HOST*)
将套接字绑定到一个空闲端口并返回端口号。这依赖于瞬时端口以确保我们能使用一个未绑定端口。这很重要因为可能会有许多测试同时运行，特别是在 buildbot 环境中。如果 sock.family 为*AF_INET* 而 sock.type 为*SOCK_STREAM*，并且套接字上设置了 SO_REUSEADDR 或 SO_REUSEPORT 则此方法将引发异常。测试绝不应该为 TCP/IP 套接字设置这些套接字选项。唯一需要设置这些选项的情况是通过多个 UDP 套接字来测试组播。

此外，如果 SO_EXCLUSIVEADDRUSE 套接字选项是可用的（例如在 Windows 上），它将在套接字上被设置。这将阻止其他任何人在测试期间绑定到我们的主机/端口。

test.support.socket_helper.**bind_unix_socket**(*sock*, *addr*)
绑定一个 unix 套接字，如果*PermissionError* 被引发则会引发*unittest.SkipTest*。

@test.support.socket_helper.**skip_unless_bind_unix_socket**
一个用于运行需要 Unix 套接字 bind() 功能的测试的装饰器。

test.support.socket_helper.**transient_internet**(*resource_name*, ***, *timeout=30.0*, *errnos=()*)
一个在互联网连接的各种问题以异常的形式表现出来时会引发*ResourceDenied* 的上下文管理器。

26.15 test.support.script_helper --- 用于 Python 执行测试工具

test.support.script_helper 模块提供了对 Python 的脚本执行测试的支持。

test.support.script_helper.**interpreter_requires_environment**()
如果 sys.executable interpreter 需要环境变量才能运行则返回 True。

这被设计用来配合 @unittest.skipIf() 以便标注需要使用 to annotate tests that need to use an assert_python*() 函数来启动隔离模式 (-I) 或无环境模式 (-E) 子解释器的测试。

正常的编译和测试运行不会进入这种状况但它在尝试从一个使用 Python 的当前家目录查找逻辑找不到明确的家目录的解释器运行标准库测试套件时有可能发生。

设置 PYTHONHOME 是一种能让大多数测试套件在这种情况下运行的办法。PYTHONPATH 或 PYTHONUSERSITE 是另外两个可影响解释器是否能启动的常见环境变量。

test.support.script_helper.**run_python_until_end**(**args*, ***env_vars*)
基于 *env_vars* 设置环境以便在子进程中运行解释器。它的值可以包括 `__isolated`, `__cleanenv`, `__cwd`, and `TERM`。

3.9 版更變: 此函数不会再从 *stderr* 去除空格符。

test.support.script_helper.**assert_python_ok**(**args*, ***env_vars*)
断言附带 *args* 和可选的环境变量 *env_vars* 运行解释器会成功 (`rc == 0`) 并返回一个 `(return code, stdout, stderr)` 元组。

If the `__cleanenv` keyword is set, *env_vars* is used as a fresh environment.

Python is started in isolated mode (command line option `-I`), except if the `__isolated` keyword is set to `False`.

3.9 版更變: 此函数不会再从 *stderr* 去除空格符。

test.support.script_helper.**assert_python_failure**(**args*, ***env_vars*)
断言附带 *args* 和可选的环境变量 *env_vars* 运行解释器会失败 (`rc != 0`) 并返回一个 `(return code, stdout, stderr)` 元组。

更多选项请参阅*assert_python_ok()*。

3.9 版更變: 此函数不会再从 *stderr* 去除空格符。

test.support.script_helper.**spawn_python**(**args*, *stdout=subprocess.PIPE*, *stderr=subprocess.STDOUT*, ***kw*)
使用给定的参数运行一个 Python 子进程。

kw 是要传给*subprocess.Popen()* 的额外关键字参数。返回一个*subprocess.Popen* 对象。

test.support.script_helper.**kill_python**(*p*)
运行给定的*subprocess.Popen* 进程直至完成并返回 stdout.

test.support.script_helper.**make_script**(*script_dir*, *script_basename*, *source*, *omit_suffix=False*)
在路径 *script_dir* 和 *script_basename* 中创建包含 *source* 的脚本。如果 *omit_suffix* 为 `False`，则为名称添加 `.py`。返回完整的脚本路径。

test.support.script_helper.**make_zip_script**(*zip_dir*, *zip_basename*, *script_name*, *name_in_zip=None*)
使用 *zip_dir* 和 *zip_basename* 创建扩展名为 `zip` 的 zip 文件，其中包含 *script_name* 中的文件。*name_in_zip* 为归档名。返回一个包含 `(full path, full path of archive name)` 的元组。

test.support.script_helper.**make_pkg**(*pkg_dir*, *init_source=''*)
创建一个名为 *pkg_dir* 的目录，其中包含一个 `__init__` 文件并以 *init_source* 作为其内容。

test.support.script_helper.**make_zip_pkg**(*zip_dir*, *zip_basename*, *pkg_name*, *script_basename*, *source*, *depth=1*, *compiled=False*)
使用 *zip_dir* 和 *zip_basename* 创建一个 zip 包目录，其中包含一个空的 `__init__` 文件和一个包含 *source* 的文件 *script_basename*。如果 *compiled* 为 `True`，则两个源文件将被编译并添加到 zip 包中。返回一个以完整 zip 路径和 zip 文件归档名为元素的元组。

26.16 test.support.bytecode_helper --- 用于测试正确字节码生成的支持工具

test.support.bytecode_helper 模块提供了对测试和检查字节码生成的支持。

3.9 版新加入.

The module defines the following class:

class test.support.bytecode_helper.**BytecodeTestCase**(*unittest.TestCase*)
这个类具有用于检查字节码的自定义断言。

BytecodeTestCase.**get_disassembly_as_string**(*co*)
以字符串形式返回 *co* 的汇编码。

BytecodeTestCase.**assertInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)
如果找到 *opname* 则返回 instr，否则抛出*AssertionError*。

BytecodeTestCase.**assertNotInBytecode**(*x*, *opname*, *argval=_UNSPECIFIED*)
如果找到 *opname* 则抛出*AssertionError*。

CHAPTER 27

调试和分析

这些库可以帮助你进行 Python 开发：调试器使你能够逐步执行代码，分析堆栈帧并设置中断点等等，性能分析器可以运行代码并为你提供执行时间的详细数据，使你能够找出你的程序中的瓶颈。审计事件提供运行时行为的可见性，如果没有此工具则需要进行侵入式调试或修补。

27.1 审计事件表

下表包含了在整个 CPython 运行时和标准库中由 *sys.audit()* 或 PySys_Audit() 调用所引发的全部事件。这些调用是在 3.8.0 或更高版本中添加了 (参见 **PEP 578**)。

请参阅 *sys.addaudithook()* 和 PySys_AddAuditHook() 了解有关处理这些事件的详细信息。

CPython implementation detail: 此表是根据 CPython 文档生成的，可能无法表示其他实现所引发的事件。请参阅你的运行时专属的文档了解实际引发的事件。

Audit event	Arguments
array.__new__	typecode, initializer
builtins.breakpoint	breakpointhook
builtins.id	id
builtins.input	prompt
builtins.input/result	result
code.__new__	code, filename, name, argcount, posonlyargcount, kwonlyargcount, nlocals
compile	source, filename
cpython.PyInterpreterState_Clear	
cpython.PyInterpreterState_New	
cpython._PySys_ClearAuditHooks	
cpython.run_command	command
cpython.run_file	filename
cpython.run_interactivehook	hook
cpython.run_module	module-name

表 1 – 繼續上一頁

Audit event	Arguments
cpython.run_startup	filename
cpython.run_stdin	
ctypes.addressof	obj
ctypes.call_function	func_pointer, arguments
ctypes.cdata	address
ctypes.cdata/buffer	pointer, size, offset
ctypes.create_string_buffer	init, size
ctypes.create_unicode_buffer	init, size
ctypes.dlopen	name
ctypes.dlsym	library, name
ctypes.dlsym/handle	handle, name
ctypes.get_errno	
ctypes.get_last_error	
ctypes.seh_exception	code
ctypes.set_errno	errno
ctypes.set_last_error	error
ctypes.string_at	address, size
ctypes.wstring_at	address, size
ensurepip.bootstrap	root
exec	code_object
fcntl.fcntl	fd, cmd, arg
fcntl.flock	fd, operation
fcntl.ioctl	fd, request, arg
fcntl.lockf	fd, cmd, len, start, whence
ftplib.connect	self, host, port
ftplib.sendcmd	self, cmd
function.__new__	code
gc.get_objects	generation
gc.get_referents	objs
gc.get_referrers	objs
glob.glob	pathname, recursive
imaplib.open	self, host, port
imaplib.send	self, data
import	module, filename, sys.path, sys.meta_path, sys.path_hooks
marshal.dumps	value, version
marshal.load	
marshal.loads	bytes
mmap.__new__	fileno, length, access, offset
msvcrt.get_osfhandle	fd
msvcrt.locking	fd, mode, nbytes
msvcrt.open_osfhandle	handle, flags
nntplib.connect	self, host, port
nntplib.putline	self, line
object.__delattr__	obj, name
object.__getattr__	obj, name
object.__setattr__	obj, name, value
open	file, mode, flags
os.add_dll_directory	path
os.chdir	path

表 1 – 繼續上一頁

Audit event	Arguments
os.chflags	path, flags
os.chmod	path, mode, dir_fd
os.chown	path, uid, gid, dir_fd
os.exec	path, args, env
os.fork	
os.forkpty	
os.fwalk	top, topdown, onerror, follow_symlinks, dir_fd
os.getxattr	path, attribute
os.kill	pid, sig
os.killpg	pgid, sig
os.link	src, dst, src_dir_fd, dst_dir_fd
os.listdir	path
os.listxattr	path
os.lockf	fd, cmd, len
os.mkdir	path, mode, dir_fd
os.posix_spawn	path, argv, env
os.putenv	key, value
os.remove	path, dir_fd
os.removexattr	path, attribute
os.rename	src, dst, src_dir_fd, dst_dir_fd
os.rmdir	path, dir_fd
os.scandir	path
os.setxattr	path, attribute, value, flags
os.spawn	mode, path, args, env
os.startfile	path, operation
os.symlink	src, dst, dir_fd
os.system	command
os.truncate	fd, length
os.unsetenv	key
os.utime	path, times, ns, dir_fd
os.walk	top, topdown, onerror, followlinks
pathlib.Path.glob	self, pattern
pathlib.Path.rglob	self, pattern
pdb.Pdb	
pickle.find_class	module, name
poplib.connect	self, host, port
poplib.putline	self, line
pty.spawn	argv
resource.prlimit	pid, resource, limits
resource.setrlimit	resource, limits
setopencodehook	
shutil.chown	path, user, group
shutil.copyfile	src, dst
shutil.copymode	src, dst
shutil.copystat	src, dst
shutil.copytree	src, dst
shutil.make_archive	base_name, format, root_dir, base_dir
shutil.move	src, dst
shutil.rmtree	path

表 1 – 繼續上一頁

Audit event	Arguments
shutil.unpack_archive	filename, extract_dir, format
signal.pthread_kill	thread_id, signalnum
smtplib.connect	self, host, port
smtplib.send	self, data
socket.__new__	self, family, type, protocol
socket.bind	self, address
socket.connect	self, address
socket.getaddrinfo	host, port, family, type, protocol
socket.gethostbyaddr	ip_address
socket.gethostbyname	hostname
socket.gethostname	
socket.getnameinfo	sockaddr
socket.getservbyname	servicename, protocolname
socket.getservbyport	port, protocolname
socket.sendmsg	self, address
socket.sendto	self, address
socket.sethostname	name
sqlite3.connect	database
subprocess.Popen	executable, args, cwd, env
sys._current_frames	
sys._getframe	
sys.addaudithook	
sys.excepthook	hook, type, value, traceback
sys.set_asyncgen_hooks_finalizer	
sys.set_asyncgen_hooks_firstiter	
sys.setprofile	
sys.settrace	
sys.unraisablehook	hook, unraisable
syslog.closelog	
syslog.openlog	ident, logoption, facility
syslog.setlogmask	maskpri
syslog.syslog	priority, message
telnetlib.Telnet.open	self, host, port
telnetlib.Telnet.write	self, buffer
tempfile.mkdtemp	fullpath
tempfile.mkstemp	fullpath
urllib.Request	fullurl, data, headers, method
webbrowser.open	url
winreg.ConnectRegistry	computer_name, key
winreg.CreateKey	key, sub_key, access
winreg.DeleteKey	key, sub_key, access
winreg.DeleteValue	key, value
winreg.DisableReflectionKey	key
winreg.EnableReflectionKey	key
winreg.EnumKey	key, index
winreg.EnumValue	key, index
winreg.ExpandEnvironmentStrings	str
winreg.LoadKey	key, sub_key, file_name
winreg.OpenKey	key, sub_key, access

表 1 – 繼續上一頁

Audit event	Arguments
winreg.OpenKey/result	`key`
winreg.PyHKEY.Detach	`key`
winreg.QueryInfoKey	`key`
winreg.QueryReflectionKey	`key`
winreg.QueryValue	`key`, `sub_key`, `value_name`
winreg.SaveKey	`key`, `file_name`
winreg.SetValue	`key`, `sub_key`, `type`, `value`

下列事件只在内部被引发，而不会回应任何 CPython 公共 API:

审计事件	实参
_winapi.CreateFile	`file_name`, `desired_access`, `share_mode`, `creation_disposition`, `flags_and_attributes`
_winapi.CreateJunction	`src_path`, `dst_path`
_winapi.CreateNamedPipe	`name`, `open_mode`, `pipe_mode`
_winapi.CreatePipe	
_winapi.CreateProcess	`application_name`, `command_line`, `current_directory`
_winapi.OpenProcess	`process_id`, `desired_access`
_winapi.TerminateProcess	`handle`, `exit_code`
ctypes.PyObj_FromPtr	`obj`

27.2 bdb --- 调试器框架

源代码: Lib/bdb.py

---

bdb 模块处理基本的调试器函数，例如设置中断点或通过调试器来管理执行。

定义了以下异常:

exception bdb.**BdbQuit**
由*Bdb* 类引发用于退出调试器的异常。

bdb 模块还定义了两个类:

class bdb.**Breakpoint**(*self*, *file*, *line*, *temporary=0*, *cond=None*, *funcname=None*)
这个类实现了临时性中断点、忽略计数、禁用与（重新）启用，以及条件设置等。

中断点通过一个名为 `bpbynumber` 的列表基于数字并通过 `bplist` 基于 `(file, line)` 对进行索引。前者指向一个*Breakpoint* 类的单独实例。后者指向一个由这种实例组成的列表，因为在每一行中可能存在多个中断点。

当创建一个中断点时，它所关联的文件名应当为规范的形式。如果定义了 *funcname*，则当该函数的第一行被执行时将增加一次中断点命中计数。带条件的中断点将总被增加一次计数。

Breakpoint 的实例具有下列方法:

deleteMe()
从关联到文件/行的列表中删除此中断点。如果它是该位置上的最后一个中断点，还将删除相应的文件/行条目。

enable()
将此中断点标记为启用。

disable()
将此中断点标记为禁用。

bpformat()
返回一个带有关于此中断点的所有信息的，格式良好的字符串:

- 此中断点的编号。
- 它是否为临时性的。
- 它的 file,line 位置。
- 导致中断的条件。
- 它是否必须在此后被忽略 N 次。
- 此中断点的命中计数。

3.2 版新加入.

bpprint(*out=None*)
将*bpformat()* 的输出打印到文件 *out*，或者如果为 None 则打印到标准输出。, to standard output.

class bdb.**Bdb**(*skip=None*)
Bdb 类是作为通用的 Python 调试器基类。

这个类负责追踪工具的细节；所派生的类应当实现用户交互。标准调试器类 (*pdb.Pdb*) 就是一个例子。

如果给出了 *skip* 参数，它必须是一个包含 glob 风格的模块名称模式的可迭代对象。调试器将不会步进到来自与这些模式相匹配的模块的帧。一个帧是否会被视为来自特定的模块是由帧的 __name__ 全局变量来确定的。

3.1 版新加入: *skip* 参数。

Bdb 的以下方法通常不需要被重载。

canonic(*filename*)
用于获取规范形式文件名的辅助方法，也就是大小写规范的（在大小写不敏感文件系统上）绝对路径，并去除两边的尖括号。

reset()
将 botframe, stopframe, returnframe 和 quitting 属性设置为准备开始调试的值。

trace_dispatch(*frame*, *event*, *arg*)
此函数被安装为被调试帧的追踪函数。它的返回值是新的追踪函数（在大多数情况下就是它自身）。

默认实现会决定如何分派帧，这取决于即将被执行的事件的类型（作为字符串传入）。*event* 可以是下列值之一:

- "line": 一个新的代码行即将被执行。
- "call": 一个函数即将被调用，或者进入了另一个代码块。
- "return": 一个即将返回的函数或其他代码块。
- "exception": 一个异常已发生。
- "c_call": 一个 C 函数即将被调用。
- "c_return": 一个 C 函数已返回。
- "c_exception": 一个 C 函数引发了异常。

对于 Python 事件，调用了专门的函数（见下文）。对于 C 事件，不执行任何操作。

arg 形参取决于之前的事件。

请参阅*sys.settrace()* 的文档了解追踪函数的更多信息。对于代码和帧对象的详情，请参考 types。

dispatch_line(*frame*)
如果调试器应当在当前行上停止，则发起调用*user_line()* 方法（该方法应当在子类中被重载）。如果设置了 Bdb.quitting 旗标（可以通过*user_line()* 设置）则将引发*BdbQuit* 异常。返回一个对*trace_dispatch()* 方法的引用以便在该作用域内进一步地追踪。

dispatch_call(*frame*, *arg*)
如果调试器应当在此函数调用上停止，则发起调用*user_call()* 方法（该方法应当在子类中被重载）。如果设置了 Bdb.quitting 旗标（可以通过*user_call()* 设置）则将引发*BdbQuit* 异常。返回一个对*trace_dispatch()* 方法的引用以便在该作用域内进一步地追踪。

dispatch_return(*frame*, *arg*)
如果调试器应当在此函数返回时停止，则发起调用*user_return()* 方法（该方法应当在子类中被重载）。如果设置了 Bdb.quitting 旗标（可以通过*user_return()* 设置）则将引发*BdbQuit* 异常。返回一个对*trace_dispatch()* 方法的引用以便在该作用域内进一步地追踪。

dispatch_exception(*frame*, *arg*)
如果调试器应当在此异常上停止，则发起调用*user_exception()* 方法（该方法应当在子类中被重载）。如果设置了 Bdb.quitting 旗标（可以通过*user_exception()* 设置）则将引发*BdbQuit* 异常。返回一个对*trace_dispatch()* 方法的引用以便在该作用域内进一步地追踪。

通常情况下派生的类不会重载下列方法，但是如果想要重新定义停止和中断点的定义则可能会重载它们。

stop_here(*frame*)
此方法将检查调用栈中 *frame* 是否位于 botframe 的下方。botframe 是调试开始所在的帧。

break_here(*frame*)
此方法将检查 *frame* 所属的文件名和行，或者至少当前函数中是否存在中断点。如果中断点是临时性的，此方法将删除它。

break_anywhere(*frame*)
此方法将检查当前帧的文件名中是否存在中断点。

派生的类应当重载这些方法以获取调试器操作的控制权。

user_call(*frame*, *argument_list*)
当有可能在被调用函数内部的任何地方需要一个中断时此方法将从*dispatch_call()* 被调用。

user_line(*frame*)
当*stop_here()* 或*break_here()* 产生 True 时此方法将从*dispatch_line()* 被调用。

user_return(*frame*, *return_value*)
当*stop_here()* 产生 True 时此方法将从*dispatch_return()* 被调用。

user_exception(*frame*, *exc_info*)
当*stop_here()* 产生 True 时此方法将从*dispatch_exception()* 被调用。

do_clear(*arg*)
处理当一个中断点属于临时性中断点时是否必须要移除它。

此方法必须由派生类来实现。

派生类与客户端可以调用以下方法来影响步进状态。

set_step()
在一行代码之后停止。Stop after one line of code.

set_next(*frame*)
在给定帧以内或以下的下一行停止。

set_return(*frame*)
当从给定的帧返回时停止。

set_until(*frame*)
当到达不大于当前行的行或当从当前帧返回时停止。

set_trace([*frame*])
从 *frame* 开始调试。如果未指定 *frame*，则从调用者的帧开始调试。

set_continue()
仅在中断点上或是当结束时停止。如果不存在中断点，则将系统追踪函数设为 None。

set_quit()
将 quitting 属性设为 True。这将在对某个 dispatch_*() 方法的下一次调用中引发*BdbQuit*。

派生的类和客户端可以调用下列方法来操纵中断点。如果出现错误则这些方法将返回一个包含错误消息的字符串，或者如果一切正常则返回 None。

set_break(*filename*, *lineno*, *temporary=0*, *cond*, *funcname*)
设置一个新的中断点。如果对于作为参数传入的 *filename* 不存在 *lineno*，则返回一条错误消息。*filename* 应为规范的形式，如在*canonic()* 方法中描述的。

clear_break(*filename*, *lineno*)
删除位于 *filename* 中 *lineno* 上的中断点。如果未设置过中断点。则将返回一条错误消息。

clear_bpbynumber(*arg*)
删除 Breakpoint.bpbynumber 中索引号为 *arg* 的中断点。如果 *arg* 不是数字或超出范围，则返回一条错误消息。

clear_all_file_breaks(*filename*)
删除 *filename* 中的所有中断点。如果未设置任何中断点，则返回一条错误消息。

clear_all_breaks()
删除全部现有的中断点。

get_bpbynumber(*arg*)
返回由指定数字所指明的中断点。如果 *arg* 是一个字符串，它将被转换为一个数字。如果 *arg* 不是一个表示数字的字符串，如果给定的中断点不存在或者已被删除，则会引发*ValueError*。

3.2 版新加入.

get_break(*filename*, *lineno*)
检查 *filename* 中在 *lineno* 上是否有中断点。

get_breaks(*filename*, *lineno*)
返回 *filename* 中在 *lineno* 上的所有中断点，或者如果未设置任何中断点则返回一个空列表。

get_file_breaks(*filename*)
返回 *filename* 中在 *lineno* 上的所有中断点，或者如果未设置任何中断点则返回一个空列表。

get_all_breaks()
返回已设置的所有中断点。

派生类与客户端可以调用以下方法来获取一个代表栈回溯信息的数组结构。

get_stack(*f*, *t*)
获取一个帧及其所有上级（调用方）和下级帧的记录列表，以及上级部分的大小。

format_stack_entry(*frame_lineno*, *lprefix=': '*)
返回一个包含有关栈条目信息的字符串，由一个 (frame, lineno) 元组来标识:

- 包含帧的规范形式的文件名。

- 函数名称，或者 "<lambda>"。
- 输入参数。
- 返回值。
- 代码的行（如果存在）。

以下两个方法可由客户端调用以使用一个调试器来调试一条以字符串形式给出的*statement*。

run(*cmd*, *globals=None*, *locals=None*)
调试一条通过*exec()* 函数执行的语句。*globals* 默认为 __main__.__dict__，*locals* 默认为 *globals*。

runeval(*expr*, *globals=None*, *locals=None*)
调试一条通过*eval()* 函数执行的表达式。*globals* 和 *locals* 的含义与在*run()* 中的相同。

runctx(*cmd*, *globals*, *locals*)
为了向下兼容。调用*run()* 方法。

runcall(*func*, */*, **args*, ***kwds*)
调试一个单独的函数调用，并返回其结果。

最后，这个模块定义了以下函数:

bdb.**checkfuncname**(*b*, *frame*)
检查是否应当在此中断，根据中断点 *b* 的设置方式。

如果是通过行号设置的，它将检查 b.line 是否与同样作为参数传入的帧中的行号一致。如果中断点是通过函数名称设置的，则必须检查是否位于正确的帧（正确的函数）以及是否位于其中第一个可执行的行。

bdb.**effective**(*file*, *line*, *frame*)
确定在这一行代码中是否存在有效的（激活的）中断点。返回一个包含中断点和表示是否可以删除临时中断点的布尔值的元组。如果没有匹配的中断点则返回 (None, None)。

bdb.**set_trace**()
使用一个来自调用方的帧的*Bdb* 实例开始调试。

27.3 faulthandler ——转储 Python 的跟踪信息

3.3 版新加入.

当故障、超时或收到用户信号时，利用本模块内的函数可转储 Python 跟踪信息。调用*faulthandler.enable()* 可安装 SIGSEGV 、SIGFPE 、SIGABRT 、SIGBUS 和 SIGILL 信号的故障处理程序。通过设置 PYTHONFAULTHANDLER 环境变量或 -X faulthandler 命令行参数，还可以在启动时开启这些设置。

故障处理程序与操作系统的故障处理程序兼容，比如 Apport 或 Windows 故障处理程序等。如果 sigaltstack() 函数可用，本模块将为信号处理程序使用备用堆栈。这样即便堆栈溢出也能转储跟踪信息。

故障处理程序将在灾难性场合调用，因此只能使用信号安全的函数（比如不能在堆上分配内存）。由于这一限制，与正常的 Python 跟踪相比，转储量是最小的。

- 只支持 ASCII 码。编码时会用到 backslashreplace 错误处理程序。
- 每个字符串限制在 500 个字符以内。
- 只会显式文件名、函数名和行号。（不显示源代码）

- 上限是 100 页内存帧和 100 个线程。
- 反序排列：最近的调用最先显示。

默认情况下，Python 的跟踪信息会写入*sys.stderr*。为了能看到跟踪信息，应用程序必须运行于终端中。日志文件也可以传给*faulthandler.enable()*。

本模块是用 C 语言实现的，所以才能在崩溃或 Python 死锁时转储跟踪信息。

在 Python 启动时，*Python 开发模式* 会调用*faulthandler.enable()*。

27.3.1 转储跟踪信息

faulthandler.**dump_traceback**(*file=sys.stderr*, *all_threads=True*)
将所有线程的跟踪数据转储到 *file* 中。如果 *all_threads* 为 `False`，则只转储当前线程。

3.5 版更變: 增加了向本函数传入文件描述符的支持。

27.3.2 错误处理程序的状态

faulthandler.**enable**(*file=sys.stderr*, *all_threads=True*)
启用故障处理程序：为 `SIGSEGV` 、`SIGFPE` 、`SIGABRT` 、`SIGBUS` 和 `SIGILL` 信号安装处理程序，以转储 Python 跟踪信息。如果 *all_threads* 为 `True`，则会为每个运行中的线程生成跟踪信息。否则只转储当前线程。

该 文件必须保持打开状态，直至禁用故障处理程序为止：参见文件描述符相关话题。

3.5 版更變: 增加了向本函数传入文件描述符的支持。

3.6 版更變: 在 Windows 系统中，同时会安装一个 Windows 异常处理程序。

faulthandler.**disable**()
停用故障处理程序：卸载由*enable()* 安装的信号处理程序。

faulthandler.**is_enabled**()
检查故障处理程序是否被启用。

27.3.3 一定时间后转储跟踪数据。

faulthandler.**dump_traceback_later**(*timeout*, *repeat=False*, *file=sys.stderr*, *exit=False*)
在 *timeout* 秒超时后，转储所有线程的跟踪信息，如果 *repeat* 为 `True`，则每隔 *timeout* 秒转储一次。如果 *exit* 为 `True`，则在转储跟踪信息后调用 `_exit()`，参数 status=1。请注意，`_exit()` 会立即关闭进程，这意味着不做任何清理工作，如刷新文件缓冲区等。如果调用两次函数，则新的调用将取代之前的参数，超时时间也会重置。计时器的精度为亚秒级。

file 必须保持打开状态，直至跟踪信息转储完毕，或调用了*cancel_dump_traceback_later()*：参见文件描述符相关话题。

本函数用一个看门狗线程实现。

3.7 版更變: 该函数现在总是可用。

3.5 版更變: 增加了向本函数传入文件描述符的支持。

faulthandler.**cancel_dump_traceback_later**()
取消*dump_traceback_later()* 的最后一次调用。

27.3.4 转储用户信号的跟踪信息。

faulthandler.**register**(*signum*, *file=sys.stderr*, *all_threads=True*, *chain=False*)
注册一个用户信号：为 *signum* 信号安装一个处理程序，将所有线程或当前线程（*all_threads* 为 `False` 时）的跟踪信息转储到 *file* 中。如果 chain 为 `True`，则调用前一个处理程序。

file 必须保持打开状态，直至信号被*unregister()* 注销：参见文件描述符相关话题。

Windows 中不可用。

3.5 版更變: 增加了向本函数传入文件描述符的支持。

faulthandler.**unregister**(*signum*)
注销一个用户信号：卸载由*register()* 安装的 *signum* 信号处理程序。如果信号已注册，返回 `True`，否则返回 `False`。

Windows 中不可用。

27.3.5 文件描述符相关话题

enable() 、*dump_traceback_later()* 和*register()* 保留其 *file* 参数给出的文件描述符。如果文件关闭，文件描述符将被一个新文件重新使用；或者用*os.dup2()* 替换了文件描述符，则跟踪信息将被写入另一个文件。每次文件被替换时，都会再次调用这些函数。

27.3.6 示例

在 Linux 中启用和停用内存段故障的默认处理程序：

```
$ python3 -c "import ctypes; ctypes.string_at(0)"
Segmentation fault

$ python3 -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700 (most recent call first):
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_at
  File "<stdin>", line 1 in <module>
Segmentation fault
```

27.4 pdb --- Python 的调试器

源代码： Lib/pdb.py

---

pdb 模块定义了一个交互式源代码调试器，用于 Python 程序。它支持在源码行间设置（有条件的）断点和单步执行，检视堆栈帧，列出源码列表，以及在任何堆栈帧的上下文中运行任意 Python 代码。它还支持事后调试，可以在程序控制下调用。

调试器是可扩展的——调试器实际被定义为*Pdb* 类。该类目前没有文档，但通过阅读源码很容易理解它。扩展接口使用了*bdb* 和*cmd* 模块。

调试器的提示符是 `(Pdb)`。在调试器的控制下运行程序的典型用法是:

```
>>> import pdb
>>> import mymodule
>>> pdb.run('mymodule.test()')
> <string>(0)?()
(Pdb) continue
> <string>(1)?()
(Pdb) continue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

3.3 版更變: 由*readline* 模块实现的 Tab 补全可用于补全本模块的命令和命令的参数，例如，Tab 补全会提供当前的全局变量和局部变量，用作 `p` 命令的参数。

也可以将 `pdb.py` 作为脚本调用，来调试其他脚本。例如:

```
python3 -m pdb myscript.py
```

当作为脚本调用时，如果要调试的程序异常退出，pdb 调试将自动进入事后调试。事后调试之后（或程序正常退出之后），pdb 将重新启动程序。自动重启会保留 pdb 的状态（如断点），在大多数情况下，这比在退出程序的同时退出调试器更加实用。

3.2 版新加入: `pdb.py` 现在接受 `-c` 选项，可以执行命令，这与将该命令写入 `.pdbrc` 文件相同，请参阅调试器命令。

3.7 版新加入: `pdb.py` 现在接受 `-m` 选项，该选项用于执行一个模块，类似于 `python3 -m`。与脚本相同，调试器将暂停在待执行模块第一行前。

The typical usage to break into the debugger is to insert:

```
import pdb; pdb.set_trace()
```

at the location you want to break into the debugger, and then run the program. You can then step through the code following this statement, and continue running without the debugger using the *continue* command.

3.7 版新加入: 内置函数*breakpoint()*，当以默认参数调用它时，可以用来代替 `import pdb; pdb.set_trace()`。

检查已崩溃程序的典型用法是:

```
>>> import pdb
>>> import mymodule
>>> mymodule.test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./mymodule.py", line 4, in test
    test2()
  File "./mymodule.py", line 3, in test2
    print(spam)
NameError: spam
>>> pdb.pm()
> ./mymodule.py(3)test2()
-> print(spam)
(Pdb)
```

本模块定义了下列函数，每个函数进入调试器的方式略有不同:

pdb.**run**(*statement*, *globals=None*, *locals=None*)
在调试器控制范围内执行 *statement*（以字符串或代码对象的形式提供）。调试器提示符会在执行代码前出现，你可以设置断点并键入*continue*，也可以使用*step* 或*next* 逐步执行语句（上述所有命令在

后文有说明)。可选参数 *globals* 和 *locals* 指定代码执行环境，默认时使用*__main__* 模块的字典。(请参阅内置函数*exec()* 或*eval()* 的说明。)

pdb.**runeval**(*expression*, *globals=None*, *locals=None*)
在调试器控制范围内执行 *expression* (以字符串或代码对象的形式提供)。*runeval()* 返回时将返回表达式的值。本函数在其他方面与*run()* 类似。

pdb.**runcall**(*function*, **args*, ***kwds*)
使用给定的参数调用 *function* (以函数或方法对象的形式提供，不能是字符串)。*runcall()* 返回的是所调用函数的返回值。调试器提示符将在进入函数后立即出现。

pdb.**set_trace**(*, *header=None*)
在调用本函数的堆栈帧处进入调试器。用于硬编码一个断点到程序中的固定点处，即使该代码不在调试状态(如断言失败时)。如果传入 *header*，它将在调试开始前被打印到控制台。

3.7 版更變: 仅关键字参数 *header*。

pdb.**post_mortem**(*traceback=None*)
进入 *traceback* 对象的事后调试。如果没有给定 *traceback*，默认使用当前正在处理的异常之一(默认时，必须存在正在处理的异常)。

pdb.**pm**()
在*sys.last_traceback* 中查找 traceback，并进入其事后调试。

run* 函数和*set_trace()* 都是别名，用于实例化*Pdb* 类和调用同名方法。如果要使用其他功能，则必须自己执行以下操作:

class pdb.**Pdb**(*completekey='tab'*, *stdin=None*, *stdout=None*, *skip=None*, *nosigint=False*, *readrc=True*)
Pdb 是调试器类。

completekey、*stdin* 和 *stdout* 参数都会传递给底层的*cmd.Cmd* 类，请参考相应的描述。

如果给出 *skip* 参数，则它必须是一个迭代器，可以迭代出 glob-style 样式的模块名称。如果遇到匹配上述样式的模块，调试器将不会进入来自该模块的堆栈帧。[1]

默认情况下，当发出 continue 命令时，Pdb 将为 SIGINT 信号设置一个处理程序(SIGINT 信号是用户在控制台按 Ctrl-C 时发出的)。这使用户可以按 Ctrl-C 再次进入调试器。如果希望 Pdb 不要修改 SIGINT 处理程序，请将 *nosigint* 设置为 true。

readrc 参数默认为 true，它控制 Pdb 是否从文件系统加载.pdbrc 文件。

启用跟踪且带有 *skip* 参数的调用示范:

```
import pdb; pdb.Pdb(skip=['django.*']).set_trace()
```

引发一个审计事件 pdb.Pdb，没有附带参数。

3.1 版新加入: *skip* 参数。

3.2 版新加入: *nosigint* 参数。在这之前，Pdb 不为 SIGINT 设置处理程序。

3.6 版更變: *readrc* 参数。

run(*statement*, *globals=None*, *locals=None*)
runeval(*expression*, *globals=None*, *locals=None*)
runcall(*function*, **args*, ***kwds*)
set_trace()
请参阅上文解释同名函数的文档。

[1] 一个帧是否会被认为源自特定模块是由帧全局变量 __name__ 来决定的。

27.4.1 调试器命令

下方列出的是调试器可接受的命令。如下所示，大多数命令可以缩写为一个或两个字母。如 h(elp) 表示可以输入 h 或 help 来输入帮助命令（但不能输入 he 或 hel，也不能是 H 或 Help 或 HELP）。命令的参数必须用空格（空格符或制表符）分隔。在命令语法中，可选参数括在方括号 ([]) 中，使用时请勿输入方括号。命令语法中的选择项由竖线 (|) 分隔。

输入一个空白行将重复最后输入的命令。例外：如果最后一个命令是*list* 命令，则会列出接下来的 11 行。

调试器无法识别的命令将被认为是 Python 语句，并在正在调试的程序的上下文中执行。Python 语句也可以用感叹号 (!) 作为前缀。这是检查正在调试的程序的强大方法，甚至可以修改变量或调用函数。当此类语句发生异常，将打印异常名称，但调试器的状态不会改变。

调试器支持*别名*。别名可以有参数，使得调试器对被检查的上下文有一定程度的适应性。

Multiple commands may be entered on a single line, separated by `;;`. (A single `;` is not used as it is the separator for multiple commands in a line that is passed to the Python parser.) No intelligence is applied to separating the commands; the input is split at the first `;;` pair, even if it is in the middle of a quoted string. A workaround for strings with double semicolons is to use implicit string concatenation `';'';'` or `";"";"`.

如果文件 .pdbrc 存在于用户主目录或当前目录中，则将其读入并执行，等同于在调试器提示符下键入该文件。这对于别名很有用。若两个文件都存在则首先读取主目录中的文件，且本地文件可以覆盖其中定义的别名。

3.2 版更變: .pdbrc 现在可以包含继续调试的命令，如*continue* 或*next*。文件中的这些命令以前是无效的。

h(elp) [command]
: 不带参数时，显示可用的命令列表。参数为 *command* 时，打印有关该命令的帮助。help pdb 显示完整文档（即*pdb* 模块的文档字符串）。由于 *command* 参数必须是标识符，因此要获取 ! 的帮助必须输入 help exec。

w(here)
: 打印堆栈回溯，最新一帧在底部。有一个箭头指向当前帧，该帧决定了大多数命令的上下文。

d(own) [count]
: 在堆栈回溯中，将当前帧向下移动 *count* 级（默认为 1 级，移向更新的帧）。

u(p) [count]
: 在堆栈回溯中，将当前帧向上移动 *count* 级（默认为 1 级，移向更老的帧）。

b(reak) [([filename:]lineno | function) [, condition]]
: 如果带有 *lineno* 参数，则在当前文件相应行处设置一个断点。如果带有 *function* 参数，则在该函数的第一条可执行语句处设置一个断点。行号可以加上文件名和冒号作为前缀，以在另一个文件（可能是尚未加载的文件）中设置一个断点。另一个文件将在*sys.path* 范围内搜索。请注意，每个断点都分配有一个编号，其他所有断点命令都引用该编号。

  如果第二个参数存在，它应该是一个表达式，且它的计算值为 true 时断点才起作用。

  如果不带参数执行，将列出所有中断，包括每个断点、命中该断点的次数、当前的忽略次数以及关联的条件（如果有）。

tbreak [([filename:]lineno | function) [, condition]]
: 临时断点，在第一次命中时会自动删除。它的参数与*break* 相同。

cl(ear) [filename:lineno | bpnumber ...]
: 如果参数是 *filename:lineno*，则清除此行上的所有断点。如果参数是空格分隔的断点编号列表，则清除这些断点。如果不带参数，则清除所有断点（但会先提示确认）。

disable [bpnumber ...]
: 禁用断点，断点以空格分隔的断点编号列表给出。禁用断点表示它不会导致程序停止执行，但是与清除断点不同，禁用的断点将保留在断点列表中并且可以（重新）启用。

enable [bpnumber ...]
启用指定的断点。

ignore bpnumber [count]
为指定的断点编号设置忽略次数。如果省略 count，则忽略次数将设置为 0。忽略次数为 0 时断点将变为活动状态。如果为非零值，在每次达到断点，且断点未禁用，且关联条件计算值为 true 的情况下，该忽略次数会递减。

condition bpnumber [condition]
为断点设置一个新 *condition*，它是一个表达式，且它的计算值为 true 时断点才起作用。如果没有给出 *condition*，则删除现有条件，也就是将断点设为无条件。

commands [bpnumber]
为编号是 *bpnumber* 的断点指定一系列命令。命令内容将显示在后续的几行中。输入仅包含 end 的行来结束命令列表。举个例子:

```
(Pdb) commands 1
(com) p some_variable
(com) end
(Pdb)
```

要删除断点上的所有命令，请输入 commands 并立即以 end 结尾，也就是不指定任何命令。

如果不带 *bpnumber* 参数，commands 作用于最后一个被设置的断点。

可以为断点指定命令来重新启动程序。只需使用*continue* 或*step* 命令或其他可以继续运行程序的命令。

如果指定了某个继续运行程序的命令（目前包括*continue*, *step*, *next*, *return*, *jump*, *quit* 及它们的缩写）将终止命令列表（就像该命令后紧跟着 end）。因为在任何时候继续运行下去（即使是简单的 next 或 step），都可能会遇到另一个断点，该断点可能具有自己的命令列表，这导致要执行的列表含糊不清。

如果在命令列表中加入'silent' 命令，那么在该断点处停下时就不会打印常规信息。如果希望断点打印特定信息后继续运行，这可能是理想的。如果没有其他命令来打印一些信息，则看不到已达到断点的迹象。

s(tep)
运行当前行，在第一个可以停止的位置（在被调用的函数内部或在当前函数的下一行）停下。

n(ext)
继续运行，直到运行到当前函数的下一行，或当前函数返回为止。（*next* 和*step* 之间的区别在于，*step* 进入被调用函数内部并停止，而*next* （几乎）全速运行被调用函数，仅在当前函数的下一行停止。）

unt(il) [lineno]
如果不带参数，则继续运行，直到行号比当前行大时停止。

如果带有行号，则继续运行，直到行号大于或等于该行号时停止。在这两种情况下，当前帧返回时也将停止。

3.2 版更變: 允许明确给定行号。

r(eturn)
继续运行，直到当前函数返回。

c(ont(inue))
继续运行，仅在遇到断点时停止。

j(ump) lineno
设置即将运行的下一行。仅可用于堆栈最底部的帧。它可以往回跳来再次运行代码，也可以往前跳来跳过不想运行的代码。

需要注意的是，不是所有的跳转都是允许的 -- 例如，不能跳转到 `for` 循环的中间或跳出 `finally` 子句。

l(ist) [first[, last]]
列出当前文件的源代码。如果不带参数，则列出当前行周围的 11 行，或继续前一个列表。如果用 . 作为参数，则列出当前行周围的 11 行。如果带有一个参数，则列出那一行周围的 11 行。如果带有两个参数，则列出所给的范围中的代码；如果第二个参数小于第一个参数，则将其解释为列出行数的计数。

当前帧中的当前行用 -> 标记。如果正在调试异常，且最早抛出或传递该异常的行不是当前行，则那一行用 >> 标记。

3.2 版新加入: >> 标记。

ll | longlist
列出当前函数或帧的所有源代码。相关行的标记与*list* 相同。

3.2 版新加入.

a(rgs)
打印当前函数的参数列表。

p expression
在当前上下文中运行 *expression* 并打印它的值。

備𝔽: `print()` 也可以使用，但它不是一个调试器命令 --- 它执行 Python *print()* 函数。

---

pp expression
与*p* 命令类似，但表达式的值使用*pprint* 模块美观地打印。

whatis expression
打印 *expression* 的类型。

source expression
尝试获取给定对象的源代码并显示出来。

3.2 版新加入.

display [expression]
如果表达式的值发生改变则显示它的值，每次将停止执行当前帧。

不带表达式则列出当前帧的所有显示表达式。

3.2 版新加入.

undisplay [expression]
不再显示当前帧中的表达式。不带表达式则清除当前帧的所有显示表达式。

3.2 版新加入.

interact
启动一个交互式解释器（使用*code* 模块），它的全局命名空间将包含当前作用域中的所有（全局和局部）名称。

3.2 版新加入.

alias [name [command]]
创建一个标识为 *name* 的别名来执行 *command*。执行的命令 不可加上引号。可替换形参可通过 `%1`, `%2` 等来标示，而 `%*` 会被所有形参所替换。如果没有给出命令，则会显示 *name* 的当前别名。如果没有给出参数，则会列出所有别名。

别名允许嵌套并可包含能在 pdb 提示符下合法输入的任何内容。请注意内部 pdb 命令 可以被别名所覆盖。这样的命令将被隐藏直到别名被移除。别名会递归地应用到命令行的第一个单词；行内的其他单词不会受影响。

作为示例，这里列出了两个有用的别名（特别适合放在 .pdbrc 文件中）：

```
# Print instance variables (usage "pi classInst")
alias pi for k in %1.__dict__.keys(): print("%1.",k,"=",%1.__dict__[k])
# Print instance variables in self
alias ps pi self
```

unalias name
删除指定的别名。

! statement
在当前堆栈帧的上下文中执行 (单行) *statement*。感叹号可以被省略，除非语句的第一个单词与调试器命令重名。要设置全局变量，你可以在同一行上为赋值命令添加前缀的 global 语句，例如:

```
(Pdb) global list_options; list_options = ['-l']
(Pdb)
```

run [args ...]
restart [args ...]
重启被调试的 Python 程序。如果提供了参数，它会用*shlex* 来拆分且拆分结果将被用作新的*sys.argv*。历史、中断点、动作和调试器选项将被保留。*restart* 是*run* 的一个别名。

q(uit)
退出调试器。被执行的程序将被中止。

debug code
进入一个对代码参数执行步进的递归调试器（该参数是在当前环境中执行的任意表达式或语句）。

retval
打印函数最后一次返回的返回值。

F解

27.5 Python Profilers 分析器

源代码： Lib/profile.py 和 Lib/pstats.py

27.5.1 profile 分析器简介

cProfile 和*profile* 提供了 Python 程序的 确定性性能分析。*profile* 是一组统计数据，描述程序的各个部分执行的频率和时间。这些统计数据可以通过*pstats* 模块格式化为报表。

Python 标准库提供了同一分析接口的两种不同实现：

1. 对于大多数用户，建议使用*cProfile*；这是一个 C 扩展插件，因为其合理的运行开销，所以适合于分析长时间运行的程序。该插件基于 lsprof，由 Brett Rosen 和 Ted Chaotter 贡献。
2. *profile* 是一个纯 Python 模块（*cProfile* 就是模拟其接口的 C 语言实现），但它会显著增加配置程序的开销。如果你正在尝试以某种方式扩展分析器，则使用此模块可能会更容易完成任务。该模块最初由 Jim Roskind 设计和编写。

備𝔽: profiler 分析器模块被设计为给指定的程序提供执行概要文件，而不是用于基准测试目的（*timeit* 才是用于此目标的，它能获得合理准确的结果）。这特别适用于将 Python 代码与 C 代码进行基准测试：分析器为 Python 代码引入开销，但不会为 C 级别的函数引入开销，因此 C 代码似乎比任何 Python 代码都更快。

27.5.2 实时用户手册

本节是为“不想阅读手册”的用户提供的。它提供了非常简短的概述，并允许用户快速对现有应用程序执行评测。

要分析采用单个参数的函数，可以执行以下操作：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")')
```

（如果*cProfile*在您的系统上不可用，请使用*profile*。）

上述操作将运行*re.compile()*并打印分析结果，如下所示：

```
      197 function calls (192 primitive calls) in 0.002 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.001    0.001 <string>:1(<module>)
     1    0.000    0.000    0.001    0.001 re.py:212(compile)
     1    0.000    0.000    0.001    0.001 re.py:268(_compile)
     1    0.000    0.000    0.000    0.000 sre_compile.py:172(_compile_charset)
     1    0.000    0.000    0.000    0.000 sre_compile.py:201(_optimize_charset)
     4    0.000    0.000    0.000    0.000 sre_compile.py:25(_identityfunction)
   3/1    0.000    0.000    0.000    0.000 sre_compile.py:33(_compile)
```

第一行显示监听了 197 个调用。在这些调用中，有 192 个是 *原始的*，这意味着调用不是通过递归引发的。下一行: `Ordered by: standard name`，表示最右边列中的文本字符串用于对输出进行排序。列标题包括：

ncalls 调用次数

tottime 在指定函数中消耗的总时间（不包括调用子函数的时间）

percall 是 `tottime` 除以 `ncalls` 的商

cumtime 指定的函数及其所有子函数（从调用到退出）消耗的累积时间。这个数字对于递归函数来说是准确的。

percall 是 `cumtime` 除以原始调用（次数）的商（即：函数运行一次的平均时间）

filename:lineno(function) 提供相应数据的每个函数

如果第一列中有两个数字（例如 3/1），则表示函数递归。第二个值是原始调用次数，第一个是调用的总次数。请注意，当函数不递归时，这两个值是相同的，并且只打印单个数字。

profile 运行结束时，打印输出不是必须的。也可以通过为 `run()` 函数指定文件名，将结果保存到文件中：

```
import cProfile
import re
cProfile.run('re.compile("foo|bar")', 'restats')
```

pstats.Stats 类从文件中读取 profile 结果，并以各种方式对其进行格式化。

cProfile 和*profile* 文件也可以作为脚本调用，以分析另一个脚本。例如：

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

-o 将 profile 结果写入文件而不是标准输出

-s 指定*sort_stats()* 排序值之一以对输出进行排序。这仅适用于未提供 -o 的情况

-m 指定要分析的是模块而不是脚本。

3.7 版新加入: *cProfile* 添加 -m 选项

3.8 版新加入: *profile* 添加 -m 选项

The *pstats* module's *Stats* class has a variety of methods for manipulating and printing the data saved into a profile results file:

```
import pstats
from pstats import SortKey
p = pstats.Stats('restats')
p.strip_dirs().sort_stats(-1).print_stats()
```

The *strip_dirs()* method removed the extraneous path from all the module names. The *sort_stats()* method sorted all the entries according to the standard module/line/name string that is printed. The *print_stats()* method printed out all the statistics. You might try the following sort calls:

```
p.sort_stats(SortKey.NAME)
p.print_stats()
```

The first call will actually sort the list by function name, and the second call will print out the statistics. The following are some interesting calls to experiment with:

```
p.sort_stats(SortKey.CUMULATIVE).print_stats(10)
```

This sorts the profile by cumulative time in a function, and then only prints the ten most significant lines. If you want to understand what algorithms are taking time, the above line is what you would use.

If you were looking to see what functions were looping a lot, and taking a lot of time, you would do:

```
p.sort_stats(SortKey.TIME).print_stats(10)
```

to sort according to time spent within each function, and then print the statistics for the top ten functions.

你也可以尝试：

```
p.sort_stats(SortKey.FILENAME).print_stats('__init__')
```

This will sort all the statistics by file name, and then print out statistics for only the class init methods (since they are spelled with `__init__` in them). As one final example, you could try:

```
p.sort_stats(SortKey.TIME, SortKey.CUMULATIVE).print_stats(.5, 'init')
```

This line sorts statistics with a primary key of time, and a secondary key of cumulative time, and then prints out some of the statistics. To be specific, the list is first culled down to 50% (re: `.5`) of its original size, then only lines containing `init` are maintained, and that sub-sub-list is printed.

If you wondered what functions called the above functions, you could now (`p` is still sorted according to the last criteria) do:

```
p.print_callers(.5, 'init')
```

and you would get a list of callers for each of the listed functions.

If you want more functionality, you're going to have to read the manual, or guess what the following functions do:

```
p.print_callees()
p.add('restats')
```

Invoked as a script, the *pstats* module is a statistics browser for reading and examining profile dumps. It has a simple line-oriented interface (implemented using *cmd*) and interactive help.

27.5.3 profile 和 cProfile 模块参考

profile 和*cProfile* 模块都提供下列函数：

profile.**run**(*command*, *filename=None*, *sort=-1*)

This function takes a single argument that can be passed to the *exec()* function, and an optional file name. In all cases this routine executes:

```
exec(command, __main__.__dict__, __main__.__dict__)
```

and gathers profiling statistics from the execution. If no file name is present, then this function automatically creates a *Stats* instance and prints a simple profiling report. If the sort value is specified, it is passed to this *Stats* instance to control how the results are sorted.

profile.**runctx**(*command*, *globals*, *locals*, *filename=None*, *sort=-1*)

This function is similar to *run()*, with added arguments to supply the globals and locals dictionaries for the *command* string. This routine executes:

```
exec(command, globals, locals)
```

and gathers profiling statistics as in the *run()* function above.

class profile.**Profile**(*timer=None*, *timeunit=0.0*, *subcalls=True*, *builtins=True*)

This class is normally only used if more precise control over profiling is needed than what the `cProfile.run()` function provides.

A custom timer can be supplied for measuring how long code takes to run via the *timer* argument. This must be a function that returns a single number representing the current time. If the number is an integer, the *timeunit* specifies a multiplier that specifies the duration of each unit of time. For example, if the timer returns times measured in thousands of seconds, the time unit would be `.001`.

Directly using the *Profile* class allows formatting profile results without writing the profile data to a file:

```
import cProfile, pstats, io
from pstats import SortKey
pr = cProfile.Profile()
pr.enable()
# ... do something ...
pr.disable()
s = io.StringIO()
sortby = SortKey.CUMULATIVE
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print(s.getvalue())
```

The *Profile* class can also be used as a context manager (supported only in *cProfile* module. see 上下文管理器类型):

```
import cProfile

with cProfile.Profile() as pr:
    # ... do something ...

pr.print_stats()
```

3.8 版更變: 添加了上下文管理器支持。

enable()
开始收集分析数据。仅在*cProfile* 可用。

disable()
停止收集分析数据。仅在*cProfile* 可用。

create_stats()
停止收集分析数据，并在内部将结果记录为当前 profile。

print_stats(*sort=-1*)
Create a *Stats* object based on the current profile and print the results to stdout.

dump_stats(*filename*)
将当前 profile 的结果写入 *filename* 。

run(*cmd*)
Profile the cmd via *exec()*.

runctx(*cmd*, *globals*, *locals*)
Profile the cmd via *exec()* with the specified global and local environment.

runcall(*func*, */*, **args*, ***kwargs*)
Profile `func(*args, **kwargs)`

Note that profiling will only work if the called command/function actually returns. If the interpreter is terminated (e.g. via a *sys.exit()* call during the called command/function execution) no profiling results will be printed.

27.5.4 `Stats` 类

Analysis of the profiler data is done using the *Stats* class.

class pstats.**Stats**(**filenames or profile*, *stream=sys.stdout*)
This class constructor creates an instance of a "statistics object" from a *filename* (or list of filenames) or from a `Profile` instance. Output will be printed to the stream specified by *stream*.

The file selected by the above constructor must have been created by the corresponding version of *profile* or *cProfile*. To be specific, there is *no* file compatibility guaranteed with future versions of this profiler, and there is no compatibility with files produced by other profilers, or the same profiler run on a different operating system. If several files are provided, all the statistics for identical functions will be coalesced, so that an overall view of several processes can be considered in a single report. If additional files need to be combined with data in an existing *Stats* object, the *add()* method can be used.

Instead of reading the profile data from a file, a `cProfile.Profile` or *profile.Profile* object can be used as the profile data source.

Stats 对象有以下方法:

strip_dirs()
This method for the *Stats* class removes all leading path information from file names. It is very useful in

reducing the size of the printout to fit within (close to) 80 columns. This method modifies the object, and the stripped information is lost. After performing a strip operation, the object is considered to have its entries in a "random" order, as it was just after object initialization and loading. If *strip_dirs()* causes two function names to be indistinguishable (they are on the same line of the same filename, and have the same function name), then the statistics for these two entries are accumulated into a single entry.

add(**filenames*)

This method of the *Stats* class accumulates additional profiling information into the current profiling object. Its arguments should refer to filenames created by the corresponding version of *profile.run()* or `cProfile.run()`. Statistics for identically named (re: file, line, name) functions are automatically accumulated into single function statistics.

dump_stats(*filename*)

Save the data loaded into the *Stats* object to a file named *filename*. The file is created if it does not exist, and is overwritten if it already exists. This is equivalent to the method of the same name on the *profile.Profile* and `cProfile.Profile` classes.

sort_stats(**keys*)

This method modifies the *Stats* object by sorting it according to the supplied criteria. The argument can be either a string or a SortKey enum identifying the basis of a sort (example: `'time'`, `'name'`, `SortKey.TIME` or `SortKey.NAME`). The SortKey enums argument have advantage over the string argument in that it is more robust and less error prone.

When more than one key is provided, then additional keys are used as secondary criteria when there is equality in all keys selected before them. For example, `sort_stats(SortKey.NAME, SortKey.FILE)` will sort all the entries according to their function name, and resolve all ties (identical function names) by sorting by file name.

For the string argument, abbreviations can be used for any key names, as long as the abbreviation is unambiguous.

The following are the valid string and SortKey:

有效字符串参数	有效枚举参数	含义
`'calls'`	SortKey.CALLS	调用次数
`'cumulative'`	SortKey.CUMULATIVE	累积时间
`'cumtime'`	N/A	累积时间
`'file'`	N/A	文件名
`'filename'`	SortKey.FILENAME	文件名
`'module'`	N/A	文件名
`'ncalls'`	N/A	调用次数
`'pcalls'`	SortKey.PCALLS	原始调用计数
`'line'`	SortKey.LINE	行号
`'name'`	SortKey.NAME	函数名称
`'nfl'`	SortKey.NFL	名称/文件/行
`'stdname'`	SortKey.STDNAME	标准名称
`'time'`	SortKey.TIME	内部时间
`'tottime'`	N/A	内部时间

Note that all sorts on statistics are in descending order (placing most time consuming items first), where as name, file, and line number searches are in ascending order (alphabetical). The subtle distinction between `SortKey.NFL` and `SortKey.STDNAME` is that the standard name is a sort of the name as printed, which means that the embedded line numbers get compared in an odd way. For example, lines 3, 20, and 40 would (if the file names were the same) appear in the string order 20, 3 and 40. In contrast, `SortKey.NFL` does a numeric compare of the line numbers. In fact, `sort_stats(SortKey.NFL)` is the same as `sort_stats(SortKey.NAME, SortKey.FILENAME, SortKey.LINE)`.

For backward-compatibility reasons, the numeric arguments `-1`, `0`, `1`, and `2` are permitted. They are interpreted as `'stdname'`, `'calls'`, `'time'`, and `'cumulative'` respectively. If this old style format (numeric) is used, only one sort key (the numeric key) will be used, and additional arguments will be silently ignored.

3.7 版新加入: Added the SortKey enum.

reverse_order()
This method for the *Stats* class reverses the ordering of the basic list within the object. Note that by default ascending vs descending order is properly selected based on the sort key of choice.

print_stats(**restrictions*)
This method for the *Stats* class prints out a report as described in the *profile.run()* definition.

The order of the printing is based on the last *sort_stats()* operation done on the object (subject to caveats in *add()* and *strip_dirs()*).

The arguments provided (if any) can be used to limit the list down to the significant entries. Initially, the list is taken to be the complete set of profiled functions. Each restriction is either an integer (to select a count of lines), or a decimal fraction between 0.0 and 1.0 inclusive (to select a percentage of lines), or a string that will interpreted as a regular expression (to pattern match the standard name that is printed). If several restrictions are provided, then they are applied sequentially. For example:

```
print_stats(.1, 'foo:')
```

would first limit the printing to first 10% of list, and then only print functions that were part of filename `.*foo:`. In contrast, the command:

```
print_stats('foo:', .1)
```

would limit the list to all functions having file names `.*foo:`, and then proceed to only print the first 10% of them.

print_callers(**restrictions*)
This method for the *Stats* class prints a list of all functions that called each function in the profiled database. The ordering is identical to that provided by *print_stats()*, and the definition of the restricting argument is also identical. Each caller is reported on its own line. The format differs slightly depending on the profiler that produced the stats:

- With *profile*, a number is shown in parentheses after each caller to show how many times this specific call was made. For convenience, a second non-parenthesized number repeats the cumulative time spent in the function at the right.
- With *cProfile*, each caller is preceded by three numbers: the number of times this specific call was made, and the total and cumulative times spent in the current function while it was invoked by this specific caller.

print_callees(**restrictions*)
This method for the *Stats* class prints a list of all function that were called by the indicated function. Aside from this reversal of direction of calls (re: called vs was called by), the arguments and ordering are identical to the *print_callers()* method.

get_stats_profile()
This method returns an instance of StatsProfile, which contains a mapping of function names to instances of FunctionProfile. Each FunctionProfile instance holds information related to the function's profile such as how long the function took to run, how many times it was called, etc...

3.9 版新加入: Added the following dataclasses: StatsProfile, FunctionProfile. Added the following function: get_stats_profile.

27.5.5 什么是确定性性能分析？

确定性性能分析旨在反映这样一个事实：即所有 函数调用、函数返回和 异常事件都被监控，并且对这些事件之间的间隔（在此期间用户的代码正在执行）进行精确计时。相反，统计分析（不是由该模块完成）随机采样有效指令指针，并推断时间花费在哪里。后一种技术传统上涉及较少的开销（因为代码不需要检测），但只提供了时间花在哪里的相对指示。

在 Python 中，由于在执行过程中总有一个活动的解释器，因此执行确定性评测不需要插入指令的代码。Python 自动为每个事件提供一个:dfn: 钩子（可选回调）。此外，Python 的解释特性往往会给执行增加太多开销，以至于在典型的应用程序中，确定性分析往往只会增加很小的处理开销。结果是，确定性分析并没有那么代价高昂，但是它提供了有关 Python 程序执行的大量运行时统计信息。

调用计数统计信息可用于识别代码中的错误（意外计数），并识别可能的内联扩展点（高频调用）。内部时间统计可用于识别应仔细优化的" 热循环" 。累积时间统计可用于识别算法选择上的高级别错误。请注意，该分析器中对累积时间的异常处理，允许直接比较算法的递归实现与迭代实现的统计信息。

27.5.6 局限性

一个限制是关于时间信息的准确性。确定性性能分析存在一个涉及精度的基本问题。最明显的限制是，底层的" 时钟" 周期大约为 0.001 秒（通常）。因此，没有什么测量会比底层时钟更精确。如果进行了足够的测量，那么" 误差" 将趋于平均。不幸的是，删除第一个错误会导致第二个错误来源。

第二个问题是，从调度事件到分析器调用获取时间函数实际 获取时钟状态，这需要" 一段时间" 。类似地，从获取时钟值（然后保存）开始，直到再次执行用户代码为止，退出分析器事件句柄时也存在一定的延迟。因此，多次调用单个函数或调用多个函数通常会累积此错误。尽管这种方式的误差通常小于时钟的精度（小于一个时钟周期），但它 可以累积并变得非常可观。

与开销较低的*cProfile* 相比，*profile* 的问题更为严重。出于这个原因，*profile* 提供了一种针对指定平台的自我校准方法，以便可以在很大程度上（平均地）消除此误差。校准后，结果将更准确（在最小二乘意义上），但它有时会产生负数（当调用计数异常低，且概率之神对您不利时：-）。因此 不要对产生的负数感到惊慌。它们应该只在你手工校准分析器的情况下才会出现，实际上结果比没有校准的情况要好。

27.5.7 准确估量

profile 模块的 profiler 会从每个事件处理时间中减去一个常量，以补偿调用 time 函数和存储结果的开销。默认情况下，常数为 0。对于特定的平台，可用以下程序获得更好修正常数（*局限性* ）。

```
import profile
pr = profile.Profile()
for i in range(5):
    print(pr.calibrate(10000))
```

The method executes the number of Python calls given by the argument, directly and again under the profiler, measuring the time for both. It then computes the hidden overhead per profiler event, and returns that as a float. For example, on a 1.8Ghz Intel Core i5 running macOS, and using Python's time.process_time() as the timer, the magical number is about 4.04e-6.

The object of this exercise is to get a fairly consistent result. If your computer is *very* fast, or your timer function has poor resolution, you might have to pass 100000, or even 1000000, to get consistent results.

当你有一个一致的答案时，有三种方法可以使用：

```
import profile

# 1. Apply computed bias to all Profile instances created hereafter.
```

(下页继续)

(繼續上一頁)

```
profile.Profile.bias = your_computed_bias

# 2. Apply computed bias to a specific Profile instance.
pr = profile.Profile()
pr.bias = your_computed_bias

# 3. Specify computed bias in instance constructor.
pr = profile.Profile(bias=your_computed_bias)
```

If you have a choice, you are better off choosing a smaller constant, and then your results will "less often" show up as negative in profile statistics.

27.5.8 使用自定义计时器

If you want to change how current time is determined (for example, to force use of wall-clock time or elapsed process time), pass the timing function you want to the `Profile` class constructor:

```
pr = profile.Profile(your_time_func)
```

The resulting profiler will then call `your_time_func`. Depending on whether you are using *profile.Profile* or `cProfile.Profile`, `your_time_func`'s return value will be interpreted differently:

profile.Profile `your_time_func` should return a single number, or a list of numbers whose sum is the current time (like what *os.times()* returns). If the function returns a single time number, or the list of returned numbers has length 2, then you will get an especially fast version of the dispatch routine.

Be warned that you should calibrate the profiler class for the timer function that you choose (see 准确估量). For most machines, a timer that returns a lone integer value will provide the best results in terms of low overhead during profiling. (*os.times()* is *pretty* bad, as it returns a tuple of floating point values). If you want to substitute a better timer in the cleanest fashion, derive a class and hardwire a replacement dispatch method that best handles your timer call, along with the appropriate calibration constant.

cProfile.Profile `your_time_func` should return a single number. If it returns integers, you can also invoke the class constructor with a second argument specifying the real duration of one unit of time. For example, if `your_integer_time_func` returns times measured in thousands of seconds, you would construct the `Profile` instance as follows:

```
pr = cProfile.Profile(your_integer_time_func, 0.001)
```

As the `cProfile.Profile` class cannot be calibrated, custom timer functions should be used with care and should be as fast as possible. For the best results with a custom timer, it might be necessary to hard-code it in the C source of the internal `_lsprof` module.

Python 3.3 adds several new functions in *time* that can be used to make precise measurements of process or wall-clock time. For example, see *time.perf_counter()*.

27.6 timeit --- 测量小代码片段的执行时间

源码： Lib/timeit.py

此模块提供了一种简单的方法来计算一小段 Python 代码的耗时。它有命令執行列介面 以及一个可调用 方法。它避免了许多测量时间的常见陷阱。另见 Tim Peter 在 O'Reilly 出版的 *Python Cookbook* 第二版中“算法”章节的概述。

27.6.1 基礎範例

以下示例显示了如何使用命令執行列介面 来比较三个不同的表达式：

```
$ python3 -m timeit '"-".join(str(n) for n in range(100))'
10000 loops, best of 5: 30.2 usec per loop
$ python3 -m timeit '"-".join([str(n) for n in range(100)])'
10000 loops, best of 5: 27.5 usec per loop
$ python3 -m timeit '"-".join(map(str, range(100)))'
10000 loops, best of 5: 23.2 usec per loop
```

这可以通过*Python 接口* 实现

```
>>> import timeit
>>> timeit.timeit('"-".join(str(n) for n in range(100))', number=10000)
0.3018611848820001
>>> timeit.timeit('"-".join([str(n) for n in range(100)])', number=10000)
0.2727368790656328
>>> timeit.timeit('"-".join(map(str, range(100)))', number=10000)
0.23702679807320237
```

从*Python 接口* 还可以传出一个可调用对象：

```
>>> timeit.timeit(lambda: "-".join(map(str, range(100))), number=10000)
0.19665591977536678
```

但请注意*timeit()* 仅在使用命令行界面时会自动确定重复次数。在示例 一节你可以找到更多的进阶示例。

27.6.2 Python 接口

该模块定义了三个便利函数和一个公共类：

timeit.**timeit**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *number=1000000*, *globals=None*)
使用给定语句、*setup* 代码和 *timer* 函数创建一个*Timer* 实例，并执行 *number* 次其*timeit()* 方法。可选的 *globals* 参数指定用于执行代码的命名空间。

3.5 版更變: 添加可选参数 *globals* 。

timeit.**repeat**(*stmt='pass'*, *setup='pass'*, *timer=<default timer>*, *repeat=5*, *number=1000000*, *globals=None*)
使用给定语句、*setup* 代码和 *timer* 函数创建一个*Timer* 实例，并使用给定的 *repeat* 计数和 *number* 执行运行其*repeat()* 方法。可选的 *globals* 参数指定用于执行代码的命名空间。

3.5 版更變: 添加可选参数 *globals* 。

3.7 版更變: *repeat* 的默认值由 3 更改为 5 。

timeit.**default_timer**()
默认的计时器，总是*time.perf_counter()* 。

3.3 版更變: *time.perf_counter()* 现在是默认计时器。

class timeit.**Timer**(*stmt='pass'*, *setup='pass'*, *timer=<timer function>*, *globals=None*)
用于小代码片段的计数执行速度的类。

构造函数接受一个将计时的语句、一个用于设置的附加语句和一个定时器函数。两个语句都默认为 `'pass'`；计时器函数与平台有关（请参阅模块文档字符串）。*stmt* 和 *setup* 也可能包含多个以 `;` 或换行符分隔的语句，只要它们不包含多行字符串文字即可。该语句默认在 timeit 的命名空间内执行；可以通过将命名空间传递给 *globals* 来控制此行为。

要测量第一个语句的执行时间，请使用*timeit()* 方法。*repeat()* 和*autorange()* 方法是方便的方法来调用*timeit()* 多次。

setup 的执行时间从总体计时执行中排除。

stmt 和 *setup* 参数也可以使用不带参数的可调用对象。这将在一个计时器函数中嵌入对它们的调用，然后由*timeit()* 执行。请注意，由于额外的函数调用，在这种情况下，计时开销会略大一些。

3.5 版更變: 添加可选参数 *globals* 。

timeit(*number=1000000*)
执行 *number* 次主要语句。这将执行一次 setup 语句，然后返回执行主语句多次所需的时间，以秒为单位测量为浮点数。参数是通过循环的次数，默认为一百万。要使用的主语句、setup 语句和 timer 函数将传递给构造函数。

備𝔽: 默认情况下，*timeit()* 暂时关闭*garbage collection* 。这种方法的优点在于它使独立时序更具可比性。缺点是 GC 可能是所测量功能性能的重要组成部分。如果是这样，可以在 *setup* 字符串中的第一个语句重新启用 GC。例如:

```
timeit.Timer('for i in range(10): oct(i)', 'gc.enable()').timeit()
```

---

autorange(*callback=None*)
自动决定调用多少次*timeit()* 。

这是一个便利函数，它反复调用*timeit()* ，以便总时间 >= 0.2 秒，返回最终（循环次数，循环所用的时间）。它调用*timeit()* 的次数以序列 1, 2, 5, 10, 20, 50, ... 递增，直到所用的时间至少为 0.2 秒。

如果给出 *callback* 并且不是 `None` ，则在每次试验后将使用两个参数调用它：`callback(number, time_taken)` 。

3.6 版新加入.

repeat(*repeat=5*, *number=1000000*)
调用*timeit()* 几次。

这是一个方便的函数，它反复调用*timeit()* ，返回结果列表。第一个参数指定调用*timeit()* 的次数。第二个参数指定*timeit()* 的 *number* 参数。

備𝔽: 从结果向量计算并报告平均值和标准差这些是很诱人的。但是，这不是很有用。在典型情况下，最低值给出了机器运行给定代码段的速度的下限；结果向量中较高的值通常不是由 Python 的速度变化引起的，而是由于其他过程干扰你的计时准确性。所以结果的*min()* 可能是你应该感兴趣的唯一数字。之后，你应该看看整个向量并应用常识而不是统计。

3.7 版更變: *repeat* 的默认值由 3 更改为 5 。

print_exc(*file=None*)
帮助程序从计时代码中打印回溯。

典型使用:

```
t = Timer(...)       # outside the try/except
try:
    t.timeit(...)    # or t.repeat(...)
except Exception:
    t.print_exc()
```

与标准回溯相比，优势在于将显示已编译模板中的源行。可选的 *file* 参数指向发送回溯的位置；它默认为*sys.stderr* 。

27.6.3 命令執行列介面

从命令行调用程序时，使用以下表单:

```
python -m timeit [-n N] [-r N] [-u U] [-s S] [-h] [statement ...]
```

如果了解以下选项:

-n N, **--number**=N
执行' 语句' 多少次

-r N, **--repeat**=N
重复计时器的次数（默认为 5）

-s S, **--setup**=S
最初要执行一次的语句（默认为 pass ）

-p, **--process**
测量进程时间，而不是 wallclock 时间，使用*time.process_time()* 而不是*time.perf_counter()* ，这是默认值

3.3 版新加入.

-u, **--unit**=U
指定定时器输出的时间单位；可以选择 nsec，usec，msec 或 sec

3.5 版新加入.

-v, **--verbose**
打印原始计时结果；重复更多位数精度

-h, **--help**
打印一条简短的使用信息并退出

可以通过将每一行指定为单独的语句参数来给出多行语句；通过在引号中包含参数并使用前导空格可以缩进行。多个*-s* 选项的处理方式相似。

如果未给出*-n*，则会通过尝试按序列 1, 2, 5, 10, 20, 50, ... 递增的数值来计算合适的循环次数，直到总计时间至少为 0.2 秒。

default_timer() 测量可能受到在同一台机器上运行的其他程序的影响，因此在需要精确计时时最好的做法是重复几次计时并使用最佳时间。*-r* 选项对此有利；在大多数情况下，默认的 5 次重复可能就足够了。你可以使用*time.process_time()* 来测量 CPU 时间。

備𝔽: 执行 pass 语句会产生一定的基线开销。这里的代码不会试图隐藏它，但你应该知道它。可以通过不带参数调用程序来测量基线开销，并且 Python 版本之间可能会有所不同。

27.6.4 示例

可以提供一个在开头只执行一次的 setup 语句:

```
$ python -m timeit -s 'text = "sample string"; char = "g"'  'char in text'
5000000 loops, best of 5: 0.0877 usec per loop
$ python -m timeit -s 'text = "sample string"; char = "g"'  'text.find(char)'
1000000 loops, best of 5: 0.342 usec per loop
```

In the output, there are three fields. The loop count, which tells you how many times the statement body was run per timing loop repetition. The repetition count ('best of 5') which tells you how many times the timing loop was repeated, and finally the time the statement body took on average within the best repetition of the timing loop. That is, the time the fastest repetition took divided by the loop count.

```
>>> import timeit
>>> timeit.timeit('char in text', setup='text = "sample string"; char = "g"')
0.41440500499993504
>>> timeit.timeit('text.find(char)', setup='text = "sample string"; char = "g"')
1.7246671520006203
```

使用*Timer*类及其方法可以完成同样的操作:

```
>>> import timeit
>>> t = timeit.Timer('char in text', setup='text = "sample string"; char = "g"')
>>> t.timeit()
0.3955516149999312
>>> t.repeat()
[0.40183617287970225, 0.37027556854118704, 0.38344867356679524, 0.3712595970846668, 0.
↪37866875250654886]
```

以下示例显示如何计算包含多行的表达式。在这里我们对比使用*hasattr()*与 try/except 的开销来测试缺失与提供对象属性:

```
$ python -m timeit 'try:' '  str.__bool__' 'except AttributeError:' '  pass'
20000 loops, best of 5: 15.7 usec per loop
$ python -m timeit 'if hasattr(str, "__bool__"): pass'
50000 loops, best of 5: 4.26 usec per loop

$ python -m timeit 'try:' '  int.__bool__' 'except AttributeError:' '  pass'
200000 loops, best of 5: 1.43 usec per loop
$ python -m timeit 'if hasattr(int, "__bool__"): pass'
100000 loops, best of 5: 2.23 usec per loop
```

```
>>> import timeit
>>> # attribute is missing
>>> s = """\
... try:
...     str.__bool__
... except AttributeError:
...     pass
```

(下页继续)

(繼續上一頁)

```
... """
>>> timeit.timeit(stmt=s, number=100000)
0.9138244460009446
>>> s = "if hasattr(str, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.5829014980008651
>>>
>>> # attribute is present
>>> s = """\
... try:
...     int.__bool__
... except AttributeError:
...     pass
... """
>>> timeit.timeit(stmt=s, number=100000)
0.04215312199994514
>>> s = "if hasattr(int, '__bool__'): pass"
>>> timeit.timeit(stmt=s, number=100000)
0.08588060699912603
```

要让*timeit* 模块访问你定义的函数，你可以传递一个包含 import 语句的 *setup* 参数:

```
def test():
    """Stupid test function"""
    L = [i for i in range(100)]

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

另一种选择是将*globals()* 传递给 *globals* 参数，这将导致代码在当前的全局命名空间中执行。这比单独指定 import 更方便

```
def f(x):
    return x**2
def g(x):
    return x**4
def h(x):
    return x**8

import timeit
print(timeit.timeit('[func(42) for func in (f,g,h)]', globals=globals()))
```

27.7 `trace` --- 跟踪 Python 语句执行

源代码: Lib/trace.py

模块*trace* module 允许你跟踪程序的执行过程，生成带注释的语句覆盖率列表，打印调用/被调用关系以及列出在程序运行期间执行过的函数。可以在其他程序或者命令行中使用它

也参考:

Coverage.py 流行的第三方代码覆盖工具，可输出 HTML ，并提供分支覆盖等高级功能。

27.7.1 命令行用法

trace 模块可由命令行调用。用法如此简单：

```
python -m trace --count -C . somefile.py ...
```

上述命令将执行 somefile.py，并在当前目录生成执行期间所有已导入 Python 模块的带注释列表。

--help
显示用法并退出。

--version
显示模块版本并退出。

3.8 版新加入: 加入了 --module 选项，允许运行可执行模块。

主要的可选参数

在调用*trace* 时，至少须指定以下可选参数之一。*-listfuncs* 与*-trace* 、*-count* 相互排斥。如果给出*--listfuncs*，就再不会接受*--count* 和*--trace* ，反之亦然。

-c, --count
在程序完成时生成一组带有注解的列表文件，显示每个语句被执行的次数。参见下面的*-coverdir* 、*-file* 和*-no-report*。

-t, --trace
执行时显示行。

-l, --listfuncs
显示程序运行时执行到的函数。

-r, --report
由之前用了*--count* 和*--file* 运行的程序产生一个带有注解的报表。不会执行代码。

-T, --trackcalls
显示程序运行时暴露出来的调用关系。

修饰器

-f, --file=<file>
用于累计多次跟踪运行计数的文件名。应与*--count* 一起使用。

-C, --coverdir=<dir>
报表文件的所在目录。package.module 的覆盖率报表将被写入文件 *dir/package/module*.cover。

-m, --missing
生成带注解的报表时，用 >>>>>> 标记未执行的行。

-s, --summary
在用到*--count* 或*--report* 时，将每个文件的简短摘要写到 stdout。

-R, --no-report
不生成带注解的报表。如果打算用*--count* 执行多次运行，然后在最后产生一组带注解的报表，该选项就很有用。

-g, --timing
在每一行前面加上时间，自程序运行算起。只在跟踪时有用。

过滤器

以下参数可重复多次。

--ignore-module=<mod>
忽略给出的模块名及其子模块（若为包）。参数可为逗号分隔的名称列表。

--ignore-dir=<dir>
忽略指定目录及其子目录下的所有模块和包。参数可为*os.pathsep* 分隔的目录列表。

27.7.2 编程接口

class trace.**Trace**(*count=1*, *trace=1*, *countfuncs=0*, *countcallers=0*, *ignoremods=()*, *ignoredirs=()*, *infile=None*, *outfile=None*, *timing=False*)
创建一个对象来跟踪单个语句或表达式的执行。所有参数均为选填。*count* 可对行号计数。*trace* 启用单行执行跟踪。*countfuncs* 可列出运行过程中调用的函数。*countcallers* 可跟踪调用关系。*ignoremods* 是要忽略的模块或包的列表。*ignoredirs* 是要忽略的模块或包的目录列表。*infile* 是个文件名，从该文件中读取存储的计数信息。*outfile* 是用来写入最新计数信息的文件名。*timing* 可以显示相对于跟踪开始时间的时间戳。

run(*cmd*)
执行命令，并根据当前跟踪参数从执行过程中收集统计数据。*cmd* 必须为字符串或 code 对象，可供传入*exec()*。

runctx(*cmd*, *globals=None*, *locals=None*)
在定义的全局和局部环境中，执行命令并收集当前跟踪参数下的执行统计数据。若没有定义 *globals* 和 *locals* ，则默认为空字典。

runfunc(*func*, */*, **args*, ***kwds*)
在*Trace* 对象的控制下，用给定的参数调用 *func*，并采用当前的跟踪参数。

results()
返回一个*CoverageResults* 对象，包含之前对指定*Trace* 实例调用 `run`、`runctx` 和 `runfunc` 的累积结果。累积的跟踪结果不会重置。

class trace.**CoverageResults**
用于覆盖跟踪结果的容器，由*Trace.results()* 创建。用户不应直接去创建。

update(*other*)
从另一个*CoverageResults* 对象中合并跟踪数据。

write_results(*show_missing=True*, *summary=False*, *coverdir=None*)
写入代码覆盖结果。设置 *show_missing* 可显示未命中的行。设置 *summary* 可在输出中包含每个模块的覆盖率摘要信息。*coverdir* 可指定覆盖率结果文件的输出目录，为 `None` 则结果将置于源文件所在目录中。

以下例子简单演示了编程接口的用法：

```
import sys
import trace

# create a Trace object, telling it what to ignore, and whether to
# do tracing or line-counting or both.
tracer = trace.Trace(
    ignoredirs=[sys.prefix, sys.exec_prefix],
    trace=0,
    count=1)
```

(下页继续)

(繼續上一頁)

```
# run the new command using the given tracer
tracer.run('main()')

# make a report, placing output in the current directory
r = tracer.results()
r.write_results(show_missing=True, coverdir=".")
```

27.8 tracemalloc --- 跟踪内存分配

3.4 版新加入.

源代码： Lib/tracemalloc.py

---

tracemalloc 模块是一个用于对 python 已申请的内存块进行 debug 的工具。它能提供以下信息:

- 定位对象分配内存的位置
- 按文件、按行统计 python 的内存块分配情况: 总大小、块的数量以及块平均大小。
- 对比两个内存快照的差异，以便排查内存泄漏

To trace most memory blocks allocated by Python, the module should be started as early as possible by setting the `PYTHONTRACEMALLOC` environment variable to `1`, or by using `-X tracemalloc` command line option. The *`tracemalloc.start()`* function can be called at runtime to start tracing Python memory allocations.

By default, a trace of an allocated memory block only stores the most recent frame (1 frame). To store 25 frames at startup: set the `PYTHONTRACEMALLOC` environment variable to `25`, or use the `-X tracemalloc=25` command line option.

27.8.1 示例

显示前 10 项

显示内存分配最多的 10 个文件:

```
import tracemalloc

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)
```

Python 测试套件的输出示例:

```
[ Top 10 ]
<frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
<frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
/usr/lib/python3.4/collections/__init__.py:368: size=244 KiB, count=2315, average=108␣
↪B
/usr/lib/python3.4/unittest/case.py:381: size=185 KiB, count=779, average=243 B
/usr/lib/python3.4/unittest/case.py:402: size=154 KiB, count=378, average=416 B
/usr/lib/python3.4/abc.py:133: size=88.7 KiB, count=347, average=262 B
<frozen importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
<frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
<string>:5: size=49.7 KiB, count=148, average=344 B
/usr/lib/python3.4/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

We can see that Python loaded `4855 KiB` data (bytecode and constants) from modules and that the *collections* module allocated `244 KiB` to build *namedtuple* types.

更多选项，请参见*Snapshot.statistics()*

计算差异

获取两个快照并显示差异：

```
import tracemalloc
tracemalloc.start()
# ... start your application ...

snapshot1 = tracemalloc.take_snapshot()
# ... call the function leaking memory ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)
```

Example of output before/after running some tests of the Python test suite:

```
[ Top 10 differences ]
<frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369),␣
↪average=117 B
/usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106),␣
↪average=119 B
/usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589),␣
↪average=519 B
<frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526),␣
↪average=139 B
/usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334),␣
↪average=86 B
/usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1),␣
↪average=96.0 KiB
/usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109),␣
↪average=784 B
/usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143),␣
↪average=557 B
/usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969),␣
↪average=76 B
```

(下页继续)

(繼續上一頁)

```
/usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126),␣
↪average=546 B
```

We can see that Python has loaded `8173 KiB` of module data (bytecode and constants), and that this is `4428 KiB` more than had been loaded before the tests, when the previous snapshot was taken. Similarly, the *linecache* module has cached `940 KiB` of Python source code to format tracebacks, all of it since the previous snapshot.

If the system has little free memory, snapshots can be written on disk using the *Snapshot.dump()* method to analyze the snapshot offline. Then use the *Snapshot.load()* method reload the snapshot.

获取一个内存块的溯源

一段找出程序中最大内存块溯源的代码:

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# pick the biggest memory block
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)
```

一段 Python 单元测试的输出案例 (限制最大 25 层堆栈)

```
903 memory blocks: 870.1 KiB
  File "<frozen importlib._bootstrap>", line 716
  File "<frozen importlib._bootstrap>", line 1036
  File "<frozen importlib._bootstrap>", line 934
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/doctest.py", line 101
    import pdb
  File "<frozen importlib._bootstrap>", line 284
  File "<frozen importlib._bootstrap>", line 938
  File "<frozen importlib._bootstrap>", line 1068
  File "<frozen importlib._bootstrap>", line 619
  File "<frozen importlib._bootstrap>", line 1581
  File "<frozen importlib._bootstrap>", line 1614
  File "/usr/lib/python3.4/test/support/__init__.py", line 1728
    import doctest
  File "/usr/lib/python3.4/test/test_pickletools.py", line 21
    support.run_doctest(pickletools)
  File "/usr/lib/python3.4/test/regrtest.py", line 1276
    test_runner()
  File "/usr/lib/python3.4/test/regrtest.py", line 976
```

(下页继续)

(繼續上一頁)

```
    display_failure=not verbose)
  File "/usr/lib/python3.4/test/regrtest.py", line 761
    match_tests=ns.match_tests)
  File "/usr/lib/python3.4/test/regrtest.py", line 1563
    main()
  File "/usr/lib/python3.4/test/__main__.py", line 3
    regrtest.main_in_temp_cwd()
  File "/usr/lib/python3.4/runpy.py", line 73
    exec(code, run_globals)
  File "/usr/lib/python3.4/runpy.py", line 160
    "__main__", fname, loader, pkg_name)
```

We can see that the most memory was allocated in the *importlib* module to load data (bytecode and constants) from modules: `870.1 KiB`. The traceback is where the *importlib* loaded data most recently: on the `import pdb` line of the *doctest* module. The traceback may change if a new module is loaded.

Pretty top

Code to display the 10 lines allocating the most memory with a pretty output, ignoring `<frozen importlib._bootstrap>` and `<unknown>` files:

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        print("#%s: %s:%s: %.1f KiB"
              % (index, frame.filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# ... run your application ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)
```

Python 测试套件的输出示例：

```
Top 10 lines
#1: Lib/base64.py:414: 419.8 KiB
    _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
#2: Lib/base64.py:306: 419.8 KiB
    _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
#3: collections/__init__.py:368: 293.6 KiB
    exec(class_definition, namespace)
#4: Lib/abc.py:133: 115.2 KiB
    cls = super().__new__(mcls, name, bases, namespace)
#5: unittest/case.py:574: 103.1 KiB
    testMethod()
#6: Lib/linecache.py:127: 95.4 KiB
    lines = fp.readlines()
#7: urllib/parse.py:476: 71.8 KiB
    for a in _hexdig for b in _hexdig}
#8: <string>:5: 62.0 KiB
#9: Lib/_weakrefset.py:37: 60.0 KiB
    self.data = set()
#10: Lib/base64.py:142: 59.8 KiB
    _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
6220 other: 3602.8 KiB
Total allocated size: 5303.1 KiB
```

更多选项，请参见*Snapshot.statistics()*

Record the current and peak size of all traced memory blocks

The following code computes two sums like `0 + 1 + 2 + ...` inefficiently, by creating a list of those numbers. This list consumes a lot of memory temporarily. We can use *get_traced_memory()* and *reset_peak()* to observe the small memory usage after the sum is computed as well as the peak memory usage during the computations:

```
import tracemalloc

tracemalloc.start()

# Example code: compute a sum with a large temporary list
large_sum = sum(list(range(100000)))

first_size, first_peak = tracemalloc.get_traced_memory()

tracemalloc.reset_peak()

# Example code: compute a sum with a small temporary list
small_sum = sum(list(range(1000)))

second_size, second_peak = tracemalloc.get_traced_memory()

print(f"{first_size=}, {first_peak=}")
print(f"{second_size=}, {second_peak=}")
```

输出:

```
first_size=664, first_peak=3592984
second_size=804, second_peak=29704
```

Using *reset_peak()* ensured we could accurately record the peak during the computation of `small_sum`, even

though it is much smaller than the overall peak size of memory blocks since the *start()* call. Without the call to *reset_peak()*, `second_peak` would still be the peak from the computation `large_sum` (that is, equal to `first_peak`). In this case, both peaks are much higher than the final memory usage, and which suggests we could optimise (by removing the unnecessary call to *list*, and writing `sum(range(...))`).

27.8.2 API

函数

`tracemalloc.`**`clear_traces`**`()`
Clear traces of memory blocks allocated by Python.

另见*stop()*.

`tracemalloc.`**`get_object_traceback`**`(`*obj*`)`
Get the traceback where the Python object *obj* was allocated. Return a *Traceback* instance, or `None` if the *tracemalloc* module is not tracing memory allocations or did not trace the allocation of the object.

See also *gc.get_referrers()* and *sys.getsizeof()* functions.

`tracemalloc.`**`get_traceback_limit`**`()`
Get the maximum number of frames stored in the traceback of a trace.

The *tracemalloc* module must be tracing memory allocations to get the limit, otherwise an exception is raised.

The limit is set by the *start()* function.

`tracemalloc.`**`get_traced_memory`**`()`
Get the current size and peak size of memory blocks traced by the *tracemalloc* module as a tuple: `(current: int, peak: int)`.

`tracemalloc.`**`reset_peak`**`()`
Set the peak size of memory blocks traced by the *tracemalloc* module to the current size.

Do nothing if the *tracemalloc* module is not tracing memory allocations.

This function only modifies the recorded peak size, and does not modify or clear any traces, unlike *clear_traces()*. Snapshots taken with *take_snapshot()* before a call to *reset_peak()* can be meaningfully compared to snapshots taken after the call.

See also *get_traced_memory()*.

3.9 版新加入.

`tracemalloc.`**`get_tracemalloc_memory`**`()`
Get the memory usage in bytes of the *tracemalloc* module used to store traces of memory blocks. Return an *int*.

`tracemalloc.`**`is_tracing`**`()`
`True` if the *tracemalloc* module is tracing Python memory allocations, `False` otherwise.

See also *start()* and *stop()* functions.

`tracemalloc.`**`start`**`(`*nframe: int=1*`)`
Start tracing Python memory allocations: install hooks on Python memory allocators. Collected tracebacks of traces will be limited to *nframe* frames. By default, a trace of a memory block only stores the most recent frame: the limit is `1`. *nframe* must be greater or equal to `1`.

You can still read the original number of total frames that composed the traceback by looking at the *Traceback.total_nframe* attribute.

Storing more than 1 frame is only useful to compute statistics grouped by 'traceback' or to compute cumulative statistics: see the *Snapshot.compare_to()* and *Snapshot.statistics()* methods.

Storing more frames increases the memory and CPU overhead of the *tracemalloc* module. Use the *get_tracemalloc_memory()* function to measure how much memory is used by the *tracemalloc* module.

The PYTHONTRACEMALLOC environment variable (PYTHONTRACEMALLOC=NFRAME) and the -X tracemalloc=NFRAME command line option can be used to start tracing at startup.

See also *stop()*, *is_tracing()* and *get_traceback_limit()* functions.

tracemalloc.**stop**()
Stop tracing Python memory allocations: uninstall hooks on Python memory allocators. Also clears all previously collected traces of memory blocks allocated by Python.

Call *take_snapshot()* function to take a snapshot of traces before clearing them.

See also *start()*, *is_tracing()* and *clear_traces()* functions.

tracemalloc.**take_snapshot**()
Take a snapshot of traces of memory blocks allocated by Python. Return a new *Snapshot* instance.

The snapshot does not include memory blocks allocated before the *tracemalloc* module started to trace memory allocations.

Tracebacks of traces are limited to *get_traceback_limit()* frames. Use the *nframe* parameter of the *start()* function to store more frames.

The *tracemalloc* module must be tracing memory allocations to take a snapshot, see the *start()* function.

See also the *get_object_traceback()* function.

域过滤器

class tracemalloc.**DomainFilter**(*inclusive: bool*, *domain: int*)
Filter traces of memory blocks by their address space (domain).

3.6 版新加入.

inclusive
If *inclusive* is True (include), match memory blocks allocated in the address space *domain*.

If *inclusive* is False (exclude), match memory blocks not allocated in the address space *domain*.

domain
Address space of a memory block (int). Read-only property.

过滤器

class tracemalloc.**Filter**(*inclusive: bool*, *filename_pattern: str*, *lineno: int=None*, *all_frames: bool=False*, *domain: int=None*)
对内存块的跟踪进行筛选。

See the *fnmatch.fnmatch()* function for the syntax of *filename_pattern*. The '.pyc' file extension is replaced with '.py'.

示例：

- Filter(True, subprocess.__file__) only includes traces of the *subprocess* module
- Filter(False, tracemalloc.__file__) excludes traces of the *tracemalloc* module

- `Filter(False, "<unknown>")` excludes empty tracebacks

3.5 版更變: The `'.pyo'` file extension is no longer replaced with `'.py'`.

3.6 版更變: Added the *domain* attribute.

domain
Address space of a memory block (`int` or `None`).

tracemalloc uses the domain `0` to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

inclusive
If *inclusive* is `True` (include), only match memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

If *inclusive* is `False` (exclude), ignore memory blocks allocated in a file with a name matching *filename_pattern* at line number *lineno*.

lineno
Line number (`int`) of the filter. If *lineno* is `None`, the filter matches any line number.

filename_pattern
Filename pattern of the filter (`str`). Read-only property.

all_frames
If *all_frames* is `True`, all frames of the traceback are checked. If *all_frames* is `False`, only the most recent frame is checked.

This attribute has no effect if the traceback limit is `1`. See the *get_traceback_limit()* function and *Snapshot.traceback_limit* attribute.

Frame

class tracemalloc.**Frame**
Frame of a traceback.

The *Traceback* class is a sequence of *Frame* instances.

filename
文件名（字符串）

lineno
行号（整形）

快照

class tracemalloc.**Snapshot**
Snapshot of traces of memory blocks allocated by Python.

The *take_snapshot()* function creates a snapshot instance.

compare_to(*old_snapshot: Snapshot*, *key_type: str*, *cumulative: bool=False*)
Compute the differences with an old snapshot. Get statistics as a sorted list of *StatisticDiff* instances grouped by *key_type*.

See the *Snapshot.statistics()* method for *key_type* and *cumulative* parameters.

The result is sorted from the biggest to the smallest by: absolute value of *StatisticDiff.size_diff*, *StatisticDiff.size*, absolute value of *StatisticDiff.count_diff*, *Statistic.count* and then by *StatisticDiff.traceback*.

dump(*filename*)
将快照写入文件

使用*load()* 重载快照。

filter_traces(*filters*)
Create a new *Snapshot* instance with a filtered *traces* sequence, *filters* is a list of *DomainFilter* and *Filter* instances. If *filters* is an empty list, return a new *Snapshot* instance with a copy of the traces.

All inclusive filters are applied at once, a trace is ignored if no inclusive filters match it. A trace is ignored if at least one exclusive filter matches it.

3.6 版更變: *DomainFilter* instances are now also accepted in *filters*.

classmethod load(*filename*)
从文件载入快照。

另见*dump()*.

statistics(*key_type: str*, *cumulative: bool=False*)
获取*Statistic* 信息列表，按 *key_type* 分组排序:

key_type	描述
`'filename'`	文件名
`'lineno'`	文件名和行号
`'traceback'`	回溯

If *cumulative* is `True`, cumulate size and count of memory blocks of all frames of the traceback of a trace, not only the most recent frame. The cumulative mode can only be used with *key_type* equals to `'filename'` and `'lineno'`.

The result is sorted from the biggest to the smallest by: *Statistic.size*, *Statistic.count* and then by *Statistic.traceback*.

traceback_limit
Maximum number of frames stored in the traceback of *traces*: result of the *get_traceback_limit()* when the snapshot was taken.

traces
Traces of all memory blocks allocated by Python: sequence of *Trace* instances.

The sequence has an undefined order. Use the *Snapshot.statistics()* method to get a sorted list of statistics.

统计

class tracemalloc.**Statistic**
统计内存分配

Snapshot.statistics() 返回*Statistic* 实例的列表。.

参见*StatisticDiff* 类。

count
内存块数（整形）。

size
Total size of memory blocks in bytes (`int`).

traceback
Traceback where the memory block was allocated, *Traceback* instance.

StatisticDiff

class tracemalloc.**StatisticDiff**
Statistic difference on memory allocations between an old and a new *Snapshot* instance.

Snapshot.compare_to() returns a list of *StatisticDiff* instances. See also the *Statistic* class.

count
Number of memory blocks in the new snapshot (int): 0 if the memory blocks have been released in the new snapshot.

count_diff
Difference of number of memory blocks between the old and the new snapshots (int): 0 if the memory blocks have been allocated in the new snapshot.

size
Total size of memory blocks in bytes in the new snapshot (int): 0 if the memory blocks have been released in the new snapshot.

size_diff
Difference of total size of memory blocks in bytes between the old and the new snapshots (int): 0 if the memory blocks have been allocated in the new snapshot.

traceback
Traceback where the memory blocks were allocated, *Traceback* instance.

跟踪

class tracemalloc.**Trace**
Trace of a memory block.

The *Snapshot.traces* attribute is a sequence of *Trace* instances.

3.6 版更變: Added the *domain* attribute.

domain
Address space of a memory block (int). Read-only property.

tracemalloc uses the domain 0 to trace memory allocations made by Python. C extensions can use other domains to trace other resources.

size
Size of the memory block in bytes (int).

traceback
Traceback where the memory block was allocated, *Traceback* instance.

回溯

class tracemalloc.**Traceback**
Sequence of *Frame* instances sorted from the oldest frame to the most recent frame.

A traceback contains at least 1 frame. If the tracemalloc module failed to get a frame, the filename "<unknown>" at line number 0 is used.

When a snapshot is taken, tracebacks of traces are limited to *get_traceback_limit()* frames. See the *take_snapshot()* function. The original number of frames of the traceback is stored in the *Traceback.total_nframe* attribute. That allows to know if a traceback has been truncated by the traceback limit.

The *Trace.traceback* attribute is an instance of *Traceback* instance.

3.7 版更變: Frames are now sorted from the oldest to the most recent, instead of most recent to oldest.

total_nframe
Total number of frames that composed the traceback before truncation. This attribute can be set to `None` if the information is not available.

3.9 版更變: The *Traceback.total_nframe* attribute was added.

format(*limit=None*, *most_recent_first=False*)
Format the traceback as a list of lines. Use the *linecache* module to retrieve lines from the source code. If *limit* is set, format the *limit* most recent frames if *limit* is positive. Otherwise, format the `abs(limit)` oldest frames. If *most_recent_first* is `True`, the order of the formatted frames is reversed, returning the most recent frame first instead of last.

Similar to the *traceback.format_tb()* function, except that *format()* does not include newlines.

示例:

```
print("Traceback (most recent call first):")
for line in traceback:
    print(line)
```

输出:

```
Traceback (most recent call first):
  File "test.py", line 9
    obj = Object()
  File "test.py", line 12
    tb = tracemalloc.get_object_traceback(f())
```

CHAPTER 28

軟體封裝與發布

这些库可帮助你发布和安装 Python 软件。虽然这些模块设计为与 'Python 包索引 <https://pypi.org>'__ 结合使用，但它们也可以与本地索引服务器一起使用，或者根本不使用任何索引服务器。

28.1 distutils --- 构建和安装 Python 模块

distutils 包为将待构建和安装的额外的模块，打包成 Python 安装包提供支持。新模块既可以是百分百的纯 Python，也可以是用 C 写的扩展模块，或者可以是一组包含了同时用 Python 和 C 编码的 Python 包。

大多数 Python 用户 不会想要直接使用这个包，而是使用 Python 包官方维护的跨版本工具。特别地，setuptools 是一个对于*distutils* 的增强选项，它能提供：

- 对声明项目依赖的支持
- 额外的用于配置哪些文件包含在源代码发布中的机制（包括与版本控制系统集成需要的插件）
- 生成项目“进入点”的能力，进入点可用作应用插件系统的基础
- 自动在安装时间生成 Windows 命令行可执行文件的能力，而不是需要预编译它们
- 跨所有受支持的 Python 版本上的一致的表现

推荐的 pip 安装器用 `setuptools` 运行所有的 `setup.py` 脚本，即使脚本本身只引了 `distutils` 包。参考 Python Packaging User Guide 获得更多信息。

为了打包工具的作者和用户能更好理解当前的打包和分发系统，遗留的基于*distutils* 的用户文档和 API 参考保持可用：

- install-index
- distutils-index

28.2 ensurepip --- 引导 pip 安装器

3.4 版新加入.

---

ensurepip 包为在已有的 Python 安装实例或虚拟环境中引导 pip 安装器提供了支持。需要使用引导才能使用 pip 的这一事实也正好反映了 pip 是一个独立的项目，有其自己的发布周期，其最新版本随 CPython 解释器的维护版本和新特性版本一同捆绑。

在大多数情况下，Python 的终端使用者不需要直接调用这个模块（pip 默认应该已被引导），不过，如果在安装 Python（或创建虚拟环境）之时跳过了安装 pip 步骤，或者日后特意卸载了 pip ，则需要使用这个模块。

備🄴: 这个模块 *无需* 访问互联网。引导启动 pip 所需的全部组件均包含在包的内部。

也参考:

installing-index 安装 Python 包的终端使用者教程

PEP 453: 在 Python 安装实例中显式引导启动 pip 这个模块的原始缘由以及规范文档

28.2.1 命令行界面

使用解释器的 -m 参数调用命令行接口。

最简单的调用方式为:

```
python -m ensurepip
```

This invocation will install pip if it is not already installed, but otherwise does nothing. To ensure the installed version of pip is at least as recent as the one bundled with ensurepip, pass the --upgrade option:

```
python -m ensurepip --upgrade
```

在默认情况下，pip 会被安装到当前虚拟环境（如果激活了虚拟环境）或系统的包目录（如果未激活虚拟环境）。安装位置可通过两个额外的命令行选项来控制。

- --root 1: 相对于给定的根目录而不是当前已激活虚拟环境（如果存在）的根目录或当前 Python 安装版的默认根目录来安装 pip。
- --user: 将 pip 安装到用户包目录而不是全局安装到当前 Python 安装版（此选项不允许在已激活虚拟环境中使用）。

在默认情况下，脚本 pipX 和 pipX.Y 将被安装（其中 X.Y 表示被用来发起调用 ensurepip 的 Python 的版本）。所安装的脚本可通过两个额外的命令行选项来控制:

- --altinstall: 如果请求了一个替代安装版，则 pipX 脚本将 不会被安装。
- --default-pip: 如果请求了一个" 默认的 pip" 安装版，则除了两个常规脚本之外还将安装 pip 脚本。

同时提供这两个脚本选择选项将会触发异常。

28.2.2 模块 API

ensurepip 暴露了两个函数用于编程:

ensurepip.**version**()
Returns a string specifying the bundled version of pip that will be installed when bootstrapping an environment.

ensurepip.**bootstrap**(*root=None*, *upgrade=False*, *user=False*, *altinstall=False*, *default_pip=False*, *verbosity=0*)
初始创建 pip 到当前的或指定的环境中。

root 指明要作为相对安装路径的替代根目录。如果 *root* 为 None，则安装会使用当前环境的默认安装位置。

upgrade indicates whether or not to upgrade an existing installation of an earlier version of pip to the bundled version.

user indicates whether to use the user scheme rather than installing globally.

By default, the scripts pipX and pipX.Y will be installed (where X.Y stands for the current version of Python).

If *altinstall* is set, then pipX will *not* be installed.

If *default_pip* is set, then pip will be installed in addition to the two regular scripts.

Setting both *altinstall* and *default_pip* will trigger *ValueError*.

verbosity controls the level of output to *sys.stdout* from the bootstrapping operation.

Raises an *auditing event* ensurepip.bootstrap with argument root.

備🄵: The bootstrapping process has side effects on both sys.path and os.environ. Invoking the command line interface in a subprocess instead allows these side effects to be avoided.

備🄵: The bootstrapping process may install additional modules required by pip, but other software should not assume those dependencies will always be present by default (as the dependencies may be removed in a future version of pip).

28.3 venv --- 创建虚拟环境

3.3 版新加入.

源码: Lib/venv/

---

venv 模块支持使用自己的站点目录创建轻量级“虚拟环境”，可选择与系统站点目录隔离。每个虚拟环境都有自己的 Python 二进制文件（与用于创建此环境的二进制文件的版本相匹配），并且可以在其站点目录中拥有自己独立的已安装 Python 软件包集。

有关 Python 虚拟环境的更多信息，请参阅 **PEP 405** 。

也参考:

Python 打包用户指南：创建和使用虚拟环境

28.3.1 创建虚拟环境

通过执行 venv 指令来创建一个虚拟环境:

```
python3 -m venv /path/to/new/virtual/environment
```

运行此命令将创建目标目录（父目录若不存在也将创建），并放置一个 pyvenv.cfg 文件在其中，文件中有一个 home 键，它的值指向运行此命令的 Python 安装（目标目录的常用名称是 .venv）。它还会创建一个 bin 子目录（在 Windows 上是 Scripts），其中包含 Python 二进制文件的副本或符号链接（视创建环境时使用的平台或参数而定）。它还会创建一个（初始为空的）lib/pythonX.Y/site-packages 子目录（在 Windows 上是 Lib\site-packages）。如果指定了一个现有的目录，这个目录就将被重新使用。

3.6 版後已🄵用: pyvenv 是 Python 3.3 和 3.4 中创建虚拟环境的推荐工具，不过 在 Python 3.6 中已弃用。

3.5 版更變: 现在推荐使用 venv 来创建虚拟环境。

在 Windows 上，调用 venv 命令如下:

```
c:\>c:\Python35\python -m venv c:\path\to\myenv
```

或者，如果已经为 Python 安装配置好 PATH 和 PATHEXT 变量:

```
c:\>python -m venv c:\path\to\myenv
```

本命令如果以 -h 参数运行，将显示可用的选项:

```
usage: venv [-h] [--system-site-packages] [--symlinks | --copies] [--clear]
            [--upgrade] [--without-pip] [--prompt PROMPT] [--upgrade-deps]
            ENV_DIR [ENV_DIR ...]

Creates virtual Python environments in one or more target directories.

positional arguments:
  ENV_DIR               A directory to create the environment in.

optional arguments:
  -h, --help            show this help message and exit
  --system-site-packages
                        Give the virtual environment access to the system
                        site-packages dir.
  --symlinks            Try to use symlinks rather than copies, when symlinks
                        are not the default for the platform.
  --copies              Try to use copies rather than symlinks, even when
                        symlinks are the default for the platform.
  --clear               Delete the contents of the environment directory if it
                        already exists, before environment creation.
  --upgrade             Upgrade the environment directory to use this version
                        of Python, assuming Python has been upgraded in-place.
  --without-pip         Skips installing or upgrading pip in the virtual
                        environment (pip is bootstrapped by default)
  --prompt PROMPT       Provides an alternative prompt prefix for this
                        environment.
  --upgrade-deps        Upgrade core dependencies: pip setuptools to the
                        latest version in PyPI

Once an environment has been created, you may wish to activate it, e.g. by
sourcing an activate script in its bin directory.
```

3.9 版更變: 添加 --upgrade-deps 选项，用于将 pip + setuptools 升级到 PyPI 上的最新版本

3.4 版更變: 默认安装 pip，并添加 `--without-pip` 和 `--copies` 选项

3.4 版更變: 在早期版本中，如果目标目录已存在，将引发错误，除非使用了 `--clear` 或 `--upgrade` 选项。

備F: 虽然 Windows 支持符号链接，但不推荐使用它们。特别注意，在文件资源管理器中双击 `python.exe` 将立即解析符号链接，并忽略虚拟环境。

備F: 在 Microsoft Windows 上，为了启用 `Activate.ps1` 脚本，可能需要修改用户的执行策略。可以运行以下 PowerShell 命令来执行此操作:

PS C:> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

参阅 About Execution Policies 以获取更多信息。

生成的 `pyvenv.cfg` 文件还包括 `include-system-site-packages` 键，如果运行 `venv` 时带有 `--system-site-packages` 选项，则键值为 `true`，否则为 `false`。

除非采用 `--without-pip` 选项，否则将会调用*ensurepip* 将 `pip` 引导到虚拟环境中。

可以向 `venv` 传入多个路径，此时将根据给定的选项，在所给的每个路径上创建相同的虚拟环境。

创建虚拟环境后，可以使用虚拟环境的二进制目录中的脚本来“激活”该环境。不同平台调用的脚本是不同的（须将 <*venv*> 替换为包含虚拟环境的目录路径）:

平台	Shell	用于激活虚拟环境的命令
POSIX	bash/zsh	$ source <venv>/bin/activate
	fish	$ source <venv>/bin/activate.fish
	csh/tcsh	$ source <venv>/bin/activate.csh
	PowerShell Core	$ <venv>/bin/Activate.ps1
Windows	cmd.exe	C:\> <venv>\Scripts\activate.bat
	PowerShell	PS C:\> <venv>\Scripts\Activate.ps1

当一个虚拟环境被激活时，`VIRTUAL_ENV` 环境变量会被设为该虚拟环境的路径。这可被用来检测程序是否运行在虚拟环境中。

激活环境不是 *必须的*，激活只是将虚拟环境的二进制目录添加到搜索路径中，这样”python” 命令将调用虚拟环境的 Python 解释器，可以运行其中已安装的脚本，而不必输入其完整路径。但是，安装在虚拟环境中的所有脚本都应在不激活的情况下可运行，并自动与虚拟环境的 Python 一起运行。

在 shell 中输入”deactivate” 可以退出虚拟环境。具体机制取决于不同平台，并且是内部实现（通常使用脚本或 shell 函数）。

3.4 版新加入: `fish` 和 `csh` 激活脚本。

3.8 版新加入: 在 POSIX 上安装 PowerShell 激活脚本，以支持 PowerShell Core。

備F: 虚拟环境是一个 Python 环境，安装到其中的 Python 解释器、库和脚本与其他虚拟环境中的内容是隔离的，且（默认）与“系统级” Python（操作系统的一部分）中安装的库是隔离的。

虚拟环境是一个目录树，其中包含 Python 可执行文件和其他文件，其他文件指示了这是一个是虚拟环境。

常用安装工具如 setuptools 和 pip 可以在虚拟环境中按预期工作。换句话说，当虚拟环境被激活，它们就会将 Python 软件包安装到虚拟环境中，无需明确指示。

当虚拟环境被激活（即虚拟环境的 Python 解释器正在运行），属性*sys.prefix* 和*sys.exec_prefix* 指向的是虚拟环境的基础目录，而*sys.base_prefix* 和*sys.base_exec_prefix* 指向非虚拟环境的 Python

安装，即曾用于创建虚拟环境的那个 Python 安装。如果虚拟环境没有被激活，则*sys.prefix* 与*sys.base_prefix* 相同，且*sys.exec_prefix* 与*sys.base_exec_prefix* 相同（它们均指向非虚拟环境的 Python 安装）。

当虚拟环境被激活，所有*distutils* 配置文件中更改安装路径的选项都会被忽略，以防止无意中将项目安装在虚拟环境之外。

在命令行 shell 中工作时，用户可以运行虚拟环境可执行文件目录中的 `activate` 脚本来激活虚拟环境（调用该文件的确切文件名和命令取决于 shell），这会将虚拟环境的可执行文件目录添加到当前 shell 的 `PATH` 环境变量。在其他情况下，无需激活虚拟环境。安装到虚拟环境中的脚本有"shebang" 行，指向虚拟环境的 Python 解释器。这意味着无论 `PATH` 的值如何，脚本都将与该解释器一起运行。在 Windows 上，如果已安装 Python Launcher for Windows，则支持处理"shebang" 行（此功能在 Python 3.3 中添加，详情请参阅 **PEP 397**）。这样，在 Windows 资源管理器中双击已安装的脚本，应该会使用正确的解释器运行该脚本，而在 `PATH` 中无需指向其虚拟环境。

28.3.2 API

上述的高级方法使用了一个简单的 API，该 API 提供了一种机制，第三方虚拟环境创建者可以根据其需求自定义环境创建过程，该 API 为*EnvBuilder* 类。

class venv.**EnvBuilder**(*system_site_packages=False*, *clear=False*, *symlinks=False*, *upgrade=False*, *with_pip=False*, *prompt=None*, *upgrade_deps=False*)

EnvBuilder 类在实例化时接受以下关键字参数：

- `system_site_packages` -- 一个布尔值，要求系统 Python 的 site-packages 对环境可用（默认为 `False`）。
- `clear` -- 一个布尔值，如果为 true，则在创建环境前将删除目标目录的现有内容。
- `symlinks` -- 一个布尔值，指示是否尝试符号链接 Python 二进制文件，而不是进行复制。
- `upgrade` -- 一个布尔值，如果为 true，则将使用当前运行的 Python 去升级一个现有的环境，这主要在原位置的 Python 更新后使用（默认为 `False`）。
- `with_pip` -- 一个布尔值，如果为 true，则确保在虚拟环境中已安装 pip。这使用的是带有 `--default-pip` 选项的*ensurepip*。
- `prompt` -- 激活虚拟环境后显示的提示符（默认为 `None`，表示使用环境所在的目录名称）。如果使用了 `"."` 这一特殊字符串，则使用当前目录的基本名称作为提示符。
- `upgrade_deps` -- 将基本 venv 模块更新为 PyPI 上的最新版本。

3.4 版更變: 添加 `with_pip` 参数

3.6 版新加入: 添加 `prompt` 参数

3.9 版新加入: 添加 `upgrade_deps` 参数

第三方虚拟环境工具的创建者可以自由地将此处提供的*EnvBuilder* 类作为基类。

返回的 env-builder 是一个对象，包含一个 `create` 方法：

create(*env_dir*)

指定要建立虚拟环境的目标目录（绝对路径或相对于当前路径）来创建虚拟环境。`create` 方法将在指定目录中创建环境，或者引发对应的异常。

EnvBuilder 类的 `create` 方法定义了可用于定制子类的钩子：

```
def create(self, env_dir):
    """
    Create a virtualized Python environment in a directory.
    env_dir is the target directory to create an environment in.
    """
    env_dir = os.path.abspath(env_dir)
    context = self.ensure_directories(env_dir)
    self.create_configuration(context)
    self.setup_python(context)
    self.setup_scripts(context)
    self.post_setup(context)
```

每个方法*ensure_directories()*, *create_configuration()*, *setup_python()*, *setup_scripts()* 和*post_setup()* 都可以被重写。

ensure_directories(*env_dir*)
创建环境目录和所有必需的目录，并返回一个上下文对象。该对象只是一个容器，保存属性（如路径），供其他方法使用。允许目录已经存在，如果指定了 `clear` 或 `upgrade` 就允许在现有环境目录上进行操作。

create_configuration(*context*)
在环境中创建 `pyvenv.cfg` 配置文件。

setup_python(*context*)
在环境中创建 Python 可执行文件的拷贝或符号链接。在 POSIX 系统上，如果给定了可执行文件 `python3.x`，将创建指向该可执行文件的 `python` 和 `python3` 符号链接，除非相同名称的文件已经存在。

setup_scripts(*context*)
将适用于平台的激活脚本安装到虚拟环境中。

upgrade_dependencies(*context*)
升级环境中 venv 依赖的核心软件包（当前为 `pip` 和 `setuptools`）。通过在环境中使用 `pip` 可执行文件来完成。

3.9 版新加入.

post_setup(*context*)
占位方法，可以在第三方实现中重写，用于在虚拟环境中预安装软件包，或是其他创建后要执行的步骤。

3.7.2 版更變: Windows 现在为 `python[w].exe` 使用重定向脚本，而不是复制实际的二进制文件。仅在 3.7.2 中，除非运行的是源码树中的构建，否则*setup_python()* 不会执行任何操作。

3.7.3 版更變: Windows 将重定向脚本复制为*setup_python()* 的一部分而非*setup_scripts()*。在 3.7.2 中不是这种情况。使用符号链接时，将链接至原始可执行文件。

此外，*EnvBuilder* 提供了如下实用方法，可以从子类的*setup_scripts()* 或*post_setup()* 调用，用来将自定义脚本安装到虚拟环境中。

install_scripts(*context*, *path*)
path 是一个目录的路径，该目录应包含子目录”common”, ”posix”, ”nt”，每个子目录存有发往对应环境中 bin 目录的脚本。在下列占位符替换完毕后，将复制”common” 的内容和与*os.name* 对应的子目录：

- `__VENV_DIR__` 会被替换为环境目录的绝对路径。
- `__VENV_NAME__` 会被替换为环境名称（环境目录的最后一个字段）。
- `__VENV_PROMPT__` 会被替换为提示符（用括号括起来的环境名称紧跟着一个空格）。
- `__VENV_BIN_NAME__` 会被替换为 bin 目录的名称（`bin` 或 `Scripts` ）。

- `__VENV_PYTHON__` 会被替换为环境可执行文件的绝对路径。

允许目录已存在（用于升级现有环境时）。

有一个方便实用的模块级别的函数:

venv.**create**(*env_dir*, *system_site_packages=False*, *clear=False*, *symlinks=False*, *with_pip=False*, *prompt=None*)

通过关键词参数来创建一个*EnvBuilder*，并且使用 *env_dir* 参数来调用它的*create()* 方法。

3.3 版新加入.

3.4 版更變: 添加 `with_pip` 参数

3.6 版更變: 添加 `prompt` 参数

28.3.3 一个扩展 EnvBuilder 的例子

下面的脚本展示了如何通过实现一个子类来扩展*EnvBuilder*。这个子类会安装 setuptools 和 pip 到被创建的虚拟环境中。

```
import os
import os.path
from subprocess import Popen, PIPE
import sys
from threading import Thread
from urllib.parse import urlparse
from urllib.request import urlretrieve
import venv

class ExtendedEnvBuilder(venv.EnvBuilder):
    """
    This builder installs setuptools and pip so that you can pip or
    easy_install other packages into the created virtual environment.

    :param nodist: If true, setuptools and pip are not installed into the
                   created virtual environment.
    :param nopip: If true, pip is not installed into the created
                  virtual environment.
    :param progress: If setuptools or pip are installed, the progress of the
                     installation can be monitored by passing a progress
                     callable. If specified, it is called with two
                     arguments: a string indicating some progress, and a
                     context indicating where the string is coming from.
                     The context argument can have one of three values:
                     'main', indicating that it is called from virtualize()
                     itself, and 'stdout' and 'stderr', which are obtained
                     by reading lines from the output streams of a subprocess
                     which is used to install the app.

                     If a callable is not specified, default progress
                     information is output to sys.stderr.
    """

    def __init__(self, *args, **kwargs):
        self.nodist = kwargs.pop('nodist', False)
        self.nopip = kwargs.pop('nopip', False)
        self.progress = kwargs.pop('progress', None)
```

(下页继续)

(繼續上一頁)

```
        self.verbose = kwargs.pop('verbose', False)
        super().__init__(*args, **kwargs)

    def post_setup(self, context):
        """
        Set up any packages which need to be pre-installed into the
        virtual environment being created.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        os.environ['VIRTUAL_ENV'] = context.env_dir
        if not self.nodist:
            self.install_setuptools(context)
        # Can't install pip without setuptools
        if not self.nopip and not self.nodist:
            self.install_pip(context)

    def reader(self, stream, context):
        """
        Read lines from a subprocess' output stream and either pass to a progress
        callable (if specified) or write progress information to sys.stderr.
        """
        progress = self.progress
        while True:
            s = stream.readline()
            if not s:
                break
            if progress is not None:
                progress(s, context)
            else:
                if not self.verbose:
                    sys.stderr.write('.')
                else:
                    sys.stderr.write(s.decode('utf-8'))
                sys.stderr.flush()
        stream.close()

    def install_script(self, context, name, url):
        _, _, path, _, _, _ = urlparse(url)
        fn = os.path.split(path)[-1]
        binpath = context.bin_path
        distpath = os.path.join(binpath, fn)
        # Download script into the virtual environment's binaries folder
        urlretrieve(url, distpath)
        progress = self.progress
        if self.verbose:
            term = '\n'
        else:
            term = ''
        if progress is not None:
            progress('Installing %s ...%s' % (name, term), 'main')
        else:
            sys.stderr.write('Installing %s ...%s' % (name, term))
            sys.stderr.flush()
        # Install in the virtual environment
```

(下页继续)

(繼續上一頁)

```
        args = [context.env_exe, fn]
        p = Popen(args, stdout=PIPE, stderr=PIPE, cwd=binpath)
        t1 = Thread(target=self.reader, args=(p.stdout, 'stdout'))
        t1.start()
        t2 = Thread(target=self.reader, args=(p.stderr, 'stderr'))
        t2.start()
        p.wait()
        t1.join()
        t2.join()
        if progress is not None:
            progress('done.', 'main')
        else:
            sys.stderr.write('done.\n')
        # Clean up - no longer needed
        os.unlink(distpath)

    def install_setuptools(self, context):
        """
        Install setuptools in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://bitbucket.org/pypa/setuptools/downloads/ez_setup.py'
        self.install_script(context, 'setuptools', url)
        # clear up the setuptools archive which gets downloaded
        pred = lambda o: o.startswith('setuptools-') and o.endswith('.tar.gz')
        files = filter(pred, os.listdir(context.bin_path))
        for f in files:
            f = os.path.join(context.bin_path, f)
            os.unlink(f)

    def install_pip(self, context):
        """
        Install pip in the virtual environment.

        :param context: The information for the virtual environment
                        creation request being processed.
        """
        url = 'https://bootstrap.pypa.io/get-pip.py'
        self.install_script(context, 'pip', url)

def main(args=None):
    compatible = True
    if sys.version_info < (3, 3):
        compatible = False
    elif not hasattr(sys, 'base_prefix'):
        compatible = False
    if not compatible:
        raise ValueError('This script is only for use with '
                         'Python 3.3 or later')
    else:
        import argparse

        parser = argparse.ArgumentParser(prog=__name__,
                                         description='Creates virtual Python '
```

(下页继续)

(繼續上一頁)

```
                                            'environments in one or '
                                            'more target '
                                            'directories.')
        parser.add_argument('dirs', metavar='ENV_DIR', nargs='+',
                            help='A directory in which to create the '
                                 'virtual environment.')
        parser.add_argument('--no-setuptools', default=False,
                            action='store_true', dest='nodist',
                            help="Don't install setuptools or pip in the "
                                 "virtual environment.")
        parser.add_argument('--no-pip', default=False,
                            action='store_true', dest='nopip',
                            help="Don't install pip in the virtual "
                                 "environment.")
        parser.add_argument('--system-site-packages', default=False,
                            action='store_true', dest='system_site',
                            help='Give the virtual environment access to the '
                                 'system site-packages dir.')
        if os.name == 'nt':
            use_symlinks = False
        else:
            use_symlinks = True
        parser.add_argument('--symlinks', default=use_symlinks,
                            action='store_true', dest='symlinks',
                            help='Try to use symlinks rather than copies, '
                                 'when symlinks are not the default for '
                                 'the platform.')
        parser.add_argument('--clear', default=False, action='store_true',
                            dest='clear', help='Delete the contents of the '
                                               'virtual environment '
                                               'directory if it already '
                                               'exists, before virtual '
                                               'environment creation.')
        parser.add_argument('--upgrade', default=False, action='store_true',
                            dest='upgrade', help='Upgrade the virtual '
                                                 'environment directory to '
                                                 'use this version of '
                                                 'Python, assuming Python '
                                                 'has been upgraded '
                                                 'in-place.')
        parser.add_argument('--verbose', default=False, action='store_true',
                            dest='verbose', help='Display the output '
                                                 'from the scripts which '
                                                 'install setuptools and pip.')
        options = parser.parse_args(args)
        if options.upgrade and options.clear:
            raise ValueError('you cannot supply --upgrade and --clear together.')
        builder = ExtendedEnvBuilder(system_site_packages=options.system_site,
                                     clear=options.clear,
                                     symlinks=options.symlinks,
                                     upgrade=options.upgrade,
                                     nodist=options.nodist,
                                     nopip=options.nopip,
                                     verbose=options.verbose)
        for d in options.dirs:
            builder.create(d)
```

(下页继续)

(繼續上一頁)

```
if __name__ == '__main__':
    rc = 1
    try:
        main()
        rc = 0
    except Exception as e:
        print('Error: %s' % e, file=sys.stderr)
    sys.exit(rc)
```

这个脚本同样可以 在线下载。

28.4 zipapp ——管理可执行的 Python zip 打包文件

3.5 版新加入.

源代码： Lib/zipapp.py

本模块提供了一套管理工具，用于创建包含 Python 代码的压缩文件，这些文件可以 直接由 Python 解释器执行。本模块提供*命令執行列介面* 和*Python API*。

28.4.1 简单示例

下述例子展示了用*命令執行列介面* 根据含有 Python 代码的目录创建一个可执行的打包文件。运行后该打包文件时，将会执行 myapp 模块中的 main 函数。

```
$ python -m zipapp myapp -m "myapp:main"
$ python myapp.pyz
<output from myapp>
```

28.4.2 命令執行列介面

若要从命令行调用，则采用以下形式：

```
$ python -m zipapp source [options]
```

如果 *source* 是个目录，将根据 *source* 的内容创建一个打包文件。如果 *source* 是个文件，则应为一个打包文件，将会复制到目标打包文件中（如果指定了 -info 选项，将会显示 shebang 行的内容）。

可以接受以下参数：

-o <output>, **--output**=<output>
: 将程序的输出写入名为 *output* 的文件中。若未指定此参数，输出的文件名将与输入的 *source* 相同，并添加扩展名 .pyz。如果显式给出了文件名，将会原样使用（因此必要时应包含扩展名 .pyz）。

  如果 *source* 是个打包文件，必须指定一个输出文件名（这时 *output* 必须与 *source* 不同）。

-p <interpreter>, **--python**=<interpreter>
: 给打包文件加入 #! 行，以便指定 解释器作为运行的命令行。另外，还让打包文件在 POSIX 平台上可执行。默认不会写入 #! 行，也不让文件可执行。

-m <mainfn>, **--main**=<mainfn>
在打包文件中写入一个 `__main__.py` 文件，用于执行 *mainfn*。*mainfn* 参数的形式应为“pkg.mod:fn”，其中“pkg.mod”是打包文件中的某个包/模块，“fn”是该模块中的一个可调用对象。`__main__.py` 文件将会执行该可调用对象。

在复制打包文件时，不能设置*--main* 参数。

-c, --compress
利用 deflate 方法压缩文件，减少输出文件的大小。默认情况下，打包文件中的文件是不压缩的。

在复制打包文件时，*--compress* 无效。

3.7 版新加入.

--info
显示嵌入在打包文件中的解释器程序，以便诊断问题。这时会忽略其他所有参数，SOURCE 必须是个打包文件，而不是目录。

-h, --help
打印简短的用法信息并退出。

28.4.3 Python API

该模块定义了两个快捷函数：

zipapp.**create_archive**(*source*, *target=None*, *interpreter=None*, *main=None*, *filter=None*, *compressed=False*)
由 *source* 创建一个应用程序打包文件。source 可以是以下形式之一：

- 一个目录名，或指向目录的*path-like object* ，这时将根据目录内容新建一个应用程序打包文件。
- 一个已存在的应用程序打包文件名，或指向这类文件的*path-like object*，这时会将该文件复制为目标文件（会稍作修改以反映出 *interpreter* 参数的值）。必要时文件名中应包括 `.pyz` 扩展名。
- 一个以字节串模式打开的文件对象。该文件的内容应为应用程序打包文件，且假定文件对象定位于打包文件的初始位置。

target 参数定义了打包文件的写入位置：

- 若是个文件名，或是*path-like object*，打包文件将写入该文件中。
- 若是个打开的文件对象，打包文件将写入该对象，该文件对象必须在字节串写入模式下打开。
- 如果省略了 target （或为 `None`），则 source 必须为一个目录，target 将是与 source 同名的文件，并加上 `.pyz` 扩展名。

参数 *interpreter* 指定了 Python 解释器程序名，用于执行打包文件。这将以“释伴（shebang）”行的形式写入打包文件的头部。在 POSIX 平台上，操作系统会进行解释，而在 Windows 平台则会由 Python 启动器进行处理。省略 *interpreter* 参数则不会写入释伴行。如果指定了解释器，且目标为文件名，则会设置目标文件的可执行属性位。

参数 *main* 指定某个可调用程序的名称，用作打包文件的主程序。仅当 source 为目录且不含 `__main__.py` 文件时，才能指定该参数。*main* 参数应采用“pkg.module:callable”的形式，通过导入“pkg.module”并不带参数地执行给出的可调用对象，即可执行打包文件。如果 source 是目录且不含“__main__.py“ 文件，省略 *main* 将会出错，生成的打包文件将无法执行。

可选参数 *filter* 指定了回调函数，将传给代表被添加文件路径的 Path 对象（相对于源目录）。在文件添加完成后，应返回 `True`。

可选参数 *compressed* 指定是否要压缩打包文件。若设为 `True`，则打包中的文件将用 deflate 方法进行压缩；否则就不会压缩。本参数在复制现有打包文件时无效。

若 *source* 或 *target* 指定的是文件对象，则调用者有责任在调用 create_archive 之后关闭文件。

当复制已有的打包文件时，提供的文件对象只需 `read` 和 `readline` 方法，或 `write` 方法。当由目录创建打包文件时，若目标为文件对象，将会将其传给类，且必须提供 `zipfile.ZipFile` 类所需的方法。

3.7 版新加入: 加入了 *filter* 和 *compressed* 参数。

`zipapp.`**`get_interpreter`**(*archive*)

返回打包文件开头的行指定的解释器程序。如果没有 `#!` 行，则返回*None*。参数 *archive* 可为文件名或在字节串模式下打开以供读取的文件类对象。`#!` 行假定是在打包文件的开头。

28.4.4 示例

将目录打包成一个文件并运行它。

```
$ python -m zipapp myapp
$ python myapp.pyz
<output from myapp>
```

同样还可用*create_archive()* 函数完成:

```
>>> import zipapp
>>> zipapp.create_archive('myapp', 'myapp.pyz')
```

要让应用程序能在 POSIX 平台上直接执行，需要指定所用的解释器。

```
$ python -m zipapp myapp -p "/usr/bin/env python"
$ ./myapp.pyz
<output from myapp>
```

若要替换已有打包文件中的释伴行，请用*create_archive()* 函数另建一个修改好的打包文件:

```
>>> import zipapp
>>> zipapp.create_archive('old_archive.pyz', 'new_archive.pyz', '/usr/bin/python3')
```

若要原地更新打包文件，可用 `BytesIO` 对象在内存中进行替换，然后再覆盖源文件。注意，原地覆盖文件会有风险，出错时会丢失原文件。这里没有考虑出错情况，但生产代码则应进行处理。另外，这种方案仅当内存足以容纳打包文件时才有意义:

```
>>> import zipapp
>>> import io
>>> temp = io.BytesIO()
>>> zipapp.create_archive('myapp.pyz', temp, '/usr/bin/python2')
>>> with open('myapp.pyz', 'wb') as f:
>>>     f.write(temp.getvalue())
```

28.4.5 指定解释器程序

注意，如果指定了解释器程序再发布应用程序打包文件，需要确保所用到的解释器是可移植的。Windows 的 Python 启动器支持大多数常见的 POSIX `#!` 行，但还需要考虑一些其他问题。

- 如果采用“/usr/bin/env python”（或其他格式的 python 调用命令，比如“/usr/bin/python”），需要考虑默认版本既可能是 Python 2 又可能是 Python 3，应让代码在两个版本下均能正常运行。
- 如果用到的 Python 版本明确，如“/usr/bin/env python3”，则没有该版本的用户将无法运行应用程序。（如果代码不兼容 Python 2，可能正该如此）。
- 因为无法指定“python X.Y 以上版本”，所以应小心“/usr/bin/env python3.4”这种精确版本的指定方式，因为对于 Python 3.5 的用户就得修改释伴行，比如：

通常应该用“/usr/bin/env python2”或“/usr/bin/env python3”的格式，具体根据代码适用于 Python 2 还是 3 而定。

28.4.6 用 zipapp 创建独立运行的应用程序

利用 *zipapp* 模块可以创建独立运行的 Python 程序，以便向最终用户发布，仅需在系统中装有合适版本的 Python 即可运行。操作的关键就是把应用程序代码和所有依赖项一起放入打包文件中。

创建独立运行打包文件的步骤如下：

1. 照常在某个目录中创建应用程序，于是会有一个 `myapp` 目录，里面有个“__main__.py“ 文件，以及所有支持性代码。
2. 用 pip 将应用程序的所有依赖项装入 `myapp` 目录。

   ```
   $ python -m pip install -r requirements.txt --target myapp
   ```

   （这里假定在 `requirements.txt` 文件中列出了项目所需的依赖项，也可以在 pip 命令行中列出依赖项）。
3. pip 在 `myapp` 中创建的 `.dist-info` 目录，是可以删除的。这些目录保存了 pip 用于管理包的元数据，由于接下来不会再用到 pip，所以不是必须存在，当然留下来也不会有什么坏处。
4. 用以下命令打包：

   ```
   $ python -m zipapp -p "interpreter" myapp
   ```

这会生成一个独立的可执行文件，可在任何装有合适解释器的机器上运行。详情参见*指定解释器程序*。可以单个文件的形式分发给用户。

在 Unix 系统中，`myapp.pyz` 文件将以原有文件名执行。如果喜欢“普通”的命令名，可以重命名该文件，去掉扩展名 `.pyz` 。在 Windows 系统中，`myapp.pyz[w]` 是可执行文件，因为 Python 解释器在安装时注册了扩展名 “.pyz“ 和 `.pyzw` 。

制作 Windows 可执行文件

在 Windows 系统中，可能没有注册扩展名 `.pyz`，另外有些场合无法“透明”地识别已注册的扩展（最简单的例子是，`subprocess.run(['myapp'])` 就找不到——需要明确指定扩展名）。

因此，在 Windows 系统中，通常最好由 zipapp 创建一个可执行文件。虽然需要用到 C 编译器，但还是相对容易做到的。基本做法有赖于以下事实，即 zip 文件内可预置任意数据，Windows 的 exe 文件也可以附带任意数据。因此，创建一个合适的启动程序并将 `.pyz` 文件附在后面，最后就能得到一个单文件的可执行文件，可运行 Python 应用程序。

合适的启动程序可以简单如下：

```
#define Py_LIMITED_API 1
#include "Python.h"

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#ifdef WINDOWS
int WINAPI wWinMain(
    HINSTANCE hInstance,      /* handle to current instance */
    HINSTANCE hPrevInstance,  /* handle to previous instance */
    LPWSTR lpCmdLine,         /* pointer to command line */
    int nCmdShow              /* show state of window */
)
#else
int wmain()
#endif
{
    wchar_t **myargv = _alloca((__argc + 1) * sizeof(wchar_t*));
    myargv[0] = __wargv[0];
    memcpy(myargv + 1, __wargv, __argc * sizeof(wchar_t *));
    return Py_Main(__argc+1, myargv);
}
```

若已定义了预处理器符号 `WINDOWS`，上述代码将会生成一个 GUI 可执行文件。若未定义则生成一个可执行的控制台文件。

直接使用标准的 MSVC 命令行工具，或利用 distutils 知道如何编译 Python 源代码，即可编译可执行文件：

```
>>> from distutils.ccompiler import new_compiler
>>> import distutils.sysconfig
>>> import sys
>>> import os
>>> from pathlib import Path

>>> def compile(src):
>>>     src = Path(src)
>>>     cc = new_compiler()
>>>     exe = src.stem
>>>     cc.add_include_dir(distutils.sysconfig.get_python_inc())
>>>     cc.add_library_dir(os.path.join(sys.base_exec_prefix, 'libs'))
>>>     # First the CLI executable
>>>     objs = cc.compile([str(src)])
>>>     cc.link_executable(objs, exe)
>>>     # Now the GUI executable
>>>     cc.define_macro('WINDOWS')
>>>     objs = cc.compile([str(src)])
```

(下页继续)

(繼續上一頁)

```
>>>     cc.link_executable(objs, exe + 'w')

>>> if __name__ == "__main__":
>>>     compile("zastub.c")
```

生成的启动程序用到了“受限 ABI”，所以可在任意版本的 Python 3.x 中运行。只要用户的 PATH 中包含了 Python（python3.dll）路径即可。

若要得到完全独立运行的发行版程序，可将附有应用程序的启动程序，与“内嵌版” Python 打包在一起即可。这样在架构匹配（32 位或 64 位）的任一 PC 上都能运行。

注意事项

要将应用程序打包为单个文件，存在一些限制。大多数情况下，无需对应用程序进行重大修改即可解决。

1. 如果应用程序依赖某个带有 C 扩展的包，则此程序包无法由打包文件运行（这是操作系统的限制，因为可执行代码必须存在于文件系统中，操作系统才能加载）。这时可去除打包文件中的依赖关系，然后要求用户事先安装好该程序包，或者与打包文件一起发布并在 __main__.py 中增加代码，将未打包模块的目录加入 sys.path 中。采用增加代码方式时，一定要为目标架构提供合适的二进制文件（可能还需在运行时根据用户的机器选择正确的版本加入 sys.path）。
2. 若要如上所述发布一个 Windows 可执行文件，就得确保用户在 PATH 中包含 “python3.dll“ 的路径（安装程序默认不会如此），或者应把应用程序与内嵌版 Python 一起打包。
3. 上述给出的启动程序采用了 Python 嵌入 API。这意味着应用程序将会是 sys.executable，而 * 不是 * 传统的 Python 解释器。代码及依赖项需做好准备。例如，如果应用程序用到了*multiprocessing* 模块，就需要调用*multiprocessing.set_executable()* 来让模块知道标准 Python 解释器的位置。

28.4.7 Python 打包应用程序的格式

自 2.6 版开始，Python 即能够执行包含文件的打包文件了。为了能被 Python 执行，应用程序的打包文件必须为包含 __main__.py 文件的标准 zip 文件，__main__.py 文件将作为应用程序的入口运行。类似于常规的 Python 脚本，父级（这里指打包文件）将放入*sys.path*，因此可从打包文件中导入更多的模块。

zip 文件格式允许在文件中预置任意数据。利用这种能力，zip 应用程序格式在文件中预置了一个标准的 POSIX “释伴”行（#!/path/to/interpreter）。

因此，Python zip 应用程序的格式会如下所示：

1. 可选的释伴行，包含字符 b'#!'，后面是解释器名，然后是换行符（b'\n'）。解释器名可为操作系统“释伴”处理所能接受的任意值，或为 Windows 系统中的 Python 启动程序。解释器名在 Windows 中应用 UTF-8 编码，在 POSIX 中则用*sys.getfilesystemencoding()*。
2. 标准的打包文件由*zipfile* 模块生成。其中 必须包含一个名为 “__main__.py“ 的文件（必须位于打包文件的“根”目录——不能位于某个子目录中）。打包文件中的数据可以是压缩或未压缩的。

如果应用程序的打包文件带有释伴行，则在 POSIX 系统中可能需要启用可执行属性，以允许直接执行。

不一定非要用本模块中的工具创建应用程序打包文件，本模块只是提供了便捷方案，上述格式的打包文件可用任何方式创建，均可被 Python 接受。

CHAPTER 29

Python 运行时服务

本章里描述的模块提供了和 Python 解释器及其环境交互相关的广泛服务。以下是综述：

29.1 sys --- 系统相关的参数和函数

该模块提供了一些变量和函数。这些变量可能被解释器使用，也可能由解释器提供。这些函数会影响解释器。本模块总是可用的。

sys.**abiflags**
在 POSIX 系统上，以标准的 configure 脚本构建的 Python 中，这个变量会包含 **PEP 3149** 中定义的 ABI 标签。

3.8 版更變: 默认的 flags 变为了空字符串（用于 pymalloc 的 m 旗标已经移除）

3.2 版新加入.

sys.**addaudithook**(*hook*)
将可调用对象 *hook* 附加到当前（子）解释器的活动的审计钩子列表中。

当通过*sys.audit()* 函数引发审计事件时，每个钩子将按照其被加入的先后顺序被调用，调用时会传入事件名称和参数元组。由 PySys_AddAuditHook() 添加的原生钩子会先被调用，然后是当前（子）解释器中添加的钩子。接下来这些钩子会记录事件，引发异常来中止操作，或是完全终止进程。

调用*sys.addaudithook()* 时它自身将引发一个名为 sys.addaudithook 的审计事件且不附带参数。如果任何现有的钩子引发了派生自*RuntimeError* 的异常，则新的钩子不会被添加并且该异常会被抑制。其结果就是，调用者无法确保他们的钩子已经被添加，除非他们控制了全部现有的钩子。

请参阅审计事件表 以获取由 CPython 引发的所有事件，并参阅 **PEP 578** 了解最初的设计讨论。

3.8 版新加入.

3.8.1 版更變: 派生自*Exception* （而非*RuntimeError* ）的异常不会被抑制。

CPython implementation detail: 启用跟踪时（参阅*settrace()* ），仅当可调用对象（钩子）的 __cantrace__ 成员设置为 true 时，才会跟踪该钩子。否则，跟踪功能将跳过该钩子。

sys.**argv**
一个列表，其中包含了被传递给 Python 脚本的命令行参数。argv[0] 为脚本的名称（是否是完整的路径名取决于操作系统）。如果是通过 Python 解释器的命令行参数 -c 来执行的，argv[0] 会被设置成字符串 '-c' 。如果没有脚本名被传递给 Python 解释器，argv[0] 为空字符串。

为了遍历标准输入，或者通过命令行传递的文件列表，参照*fileinput* 模块

備𝔼: 在 Unix 上，系统传递的命令行参数是字节类型的。Python 使用文件系统编码和"surrogateescape" 错误处理方案对它们进行解码。当需要原始字节时，可以通过 [os.fsencode(arg) for arg in sys.argv] 来获取。

---

sys.**audit**(*event*, **args*)
引发一个审计事件并触发任何激活的审计钩子。*event* 是一个用于标识事件的字符串，*args* 会包含有关事件的更多信息的可选参数。特定事件的参数的数量和类型会被视为是公有的稳定 API 且不应当在版本之间进行修改。

例如，有一个审计事件的名称为 os.chdir。此事件具有一个名为 *path* 的参数，该参数将包含所请求的新工作目录。

sys.audit() 将调用现有的审计钩子，传入事件名称和参数，并将重新引发来自任何钩子的第一个异常。通常来说，如果有一个异常被引发，则它不应当被处理且其进程应当被尽可能快地终止。这将允许钩子实现来决定对特定事件要如何反应：它们可以只是将事件写入日志或是通过引发异常来中止操作。

钩子程序由*sys.addaudithook()* 或 PySys_AddAuditHook() 函数添加。

与本函数相等效的原生函数是 PySys_Audit()，应尽量使用原生函数。

参阅审计事件表 以获取 CPython 定义的所有审计事件。

3.8 版新加入.

sys.**base_exec_prefix**
在 site.py 运行之前，Python 启动的时候被设置为跟*exec_prefix* 同样的值。如果不是运行在虚拟环境 中，两个值会保持相同；如果 site.py 发现处于一个虚拟环境中，*prefix* 和*exec_prefix* 将会指向虚拟环境。然而*base_prefix* 和*base_exec_prefix* 将仍然会指向基础的 Python 环境（用来创建虚拟环境的 Python 环境）

3.3 版新加入.

sys.**base_prefix**
在 site.py 运行之前，Python 启动的时候被设置为跟*prefix* 同样的值。如果不是运行在虚拟环境 中, 两个值会保持相同；如果 site.py 发现处于一个虚拟环境中，*prefix* 和*exec_prefix* 将会指向虚拟环境。然而*base_prefix* 和*base_exec_prefix* 将仍然会指向基础的 Python 环境（用来创建虚拟环境的 Python 环境）

3.3 版新加入.

sys.**byteorder**
本地字节顺序的指示符。在大端序（最高有效位优先）操作系统上值为 'big' ，在小端序（最低有效位优先）操作系统上为 'little' 。

sys.**builtin_module_names**
一个元素为字符串的元组。包含了所有的被编译进 Python 解释器的模块。（这个信息无法通过其他的办法获取，modules.keys() 只包括被导入过的模块。）

sys.**call_tracing**(*func*, *args*)
在启用跟踪时调用 func(*args) 来保存跟踪状态，然后恢复跟踪状态。这将从检查点的调试器调用，以便递归地调试其他的一些代码。

sys.**copyright**
一个字符串，包含了 Python 解释器有关的版权信息

sys.**_clear_type_cache**()
清除内部的类型缓存。类型缓存是为了加速查找方法和属性的。在调试引用泄漏的时候调用这个函数只会清除不必要的引用。

这个函数应该只在内部为了一些特定的目的使用。

sys.**_current_frames**()
返回一个字典，存放着每个线程的标识符与（调用本函数时）该线程栈顶的帧（当前活动的帧）之间的映射。注意*traceback* 模块中的函数可以在给定某一帧的情况下构建调用堆栈。

这对于调试死锁最有用：本函数不需要死锁线程的配合，并且只要这些线程的调用栈保持死锁，它们就是冻结的。在调用本代码来检查栈顶的帧的那一刻，非死锁线程返回的帧可能与该线程当前活动的帧没有任何关系。

这个函数应该只在内部为了一些特定的目的使用。

引发一个审计事件 sys._current_frames，没有附带参数。

sys.**breakpointhook**()
本钩子函数由内建函数*breakpoint()* 调用。默认情况下，它将进入*pdb* 调试器，但可以将其改为任何其他函数，以选择使用哪个调试器。

该函数的特征取决于其调用的函数。例如，默认绑定（即 pdb.set_trace() ）不要求提供参数，但可以将绑定换成要求提供附加参数（位置参数/关键字参数）的函数。内建函数 breakpoint() 直接将其 *args 和 **kws 传入。breakpointhooks() 返回的所有内容都会从 breakpoint() 返回。

默认的实现首先会查询环境变量 PYTHONBREAKPOINT。如果将该变量设置为 "0"，则本函数立即返回，表示在断点处无操作。如果未设置该环境变量或将其设置为空字符串，则调用 pdb.set_trace()。否则，此变量应指定要运行的函数，指定函数时应使用 Python 的点导入命名法，如 package.subpackage.module.function。这种情况下将导入 package.subpackage.module，且导入的模块必须有一个名为 function() 的可调用对象。该可调用对象会运行，*args 和 **kws 会传入，且无论 function() 返回什么，sys.breakpointhook() 都将返回到F建函数*breakpoint()*。

请注意，如果在导入 PYTHONBREAKPOINT 指定的可调用对象时出错，则将报告一个*RuntimeWarning* 并忽略断点。

另请注意，如果以编程方式覆盖 sys.breakpointhook()，则 不会查询 PYTHONBREAKPOINT。

3.7 版新加入.

sys.**_debugmallocstats**()
将有关 CPython 内存分配器状态的底层的信息打印至 stderr。

如果 Python 被配置为 --with-pydebug，本方法还将执行一些开销较大的内部一致性检查。

3.3 版新加入.

CPython implementation detail: 本函数仅限 CPython。此处没有定义确切的输出格式，且可能会更改。

sys.**dllhandle**
指向 Python DLL 句柄的整数。

可用性: Windows。

sys.**displayhook**(*value*)
如果 *value* 不是 None，则本函数会将 repr(value) 打印至 sys.stdout，并将 *value* 保存在 builtins._ 中。如果 repr(value) 无法用 sys.stdout.errors 错误处理方案（可能为 'strict' ）编码为 sys.stdout.encoding，则用 'backslashreplace' 错误处理方案将其编码为 sys.stdout.encoding。

在交互式 Python 会话中运行*expression* 产生结果后，将在结果上调用 sys.displayhook。若要自定义这些 value 的显示，可以将 sys.displayhook 指定为另一个单参数函数。

伪代码:

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

3.2 版更變: 在发生*UnicodeEncodeError* 时使用 'backslashreplace' 错误处理方案。

sys.**dont_write_bytecode**
如果该值为 true，则 Python 在导入源码模块时将不会尝试写入 .pyc 文件。该值会被初始化为 True 或 False，依据是 -B 命令行选项和 PYTHONDONTWRITEBYTECODE 环境变量，可以自行设置该值，来控制是否生成字节码文件。

sys.**pycache_prefix**
如果将该值设为某个目录（不是 None），Python 会将字节码缓存文件 .pyc 写入到以该目录为根的并行目录树中（并从中读取），而不是在源码树中的 __pycache__ 目录下读写。源码树中所有的 __pycache__ 目录都将被忽略，并将在 pycache prefix 内写入新的 *.pyc* 文件。因此，如果使用*compileall* 作为预构建步骤，则必须确保预构建时使用的 pycache prefix （如果有）与将来运行的时候相同。

相对路径将解释为相对于当前工作目录。

该值的初值设置，依据 -X pycache_prefix=PATH 命令行选项或 PYTHONPYCACHEPREFIX 环境变量的值（命令行优先）。如果两者均未设置，则为 None。

3.8 版新加入.

sys.**excepthook**(*type*, *value*, *traceback*)
本函数会将所给的回溯和异常输出到 sys.stderr 中。

当抛出一个异常，且未被捕获时，解释器将调用 sys.excepthook 并带有三个参数：异常类、异常实例和一个回溯对象。在交互式会话中，这会在控制权返回到提示符之前发生。在 Python 程序中，这会在程序退出之前发生。如果要自定义此类顶级异常的处理过程，可以将另一个 3 个参数的函数赋给 sys.excepthook。

引发一个审计事件 sys.excepthook，附带参数 hook, type, value, traceback。

也参考:

sys.unraisablehook() 函数处理无法抛出的异常，*threading.excepthook()* 函数处理*threading.Thread.run()* 抛出的异常。

sys.**__breakpointhook__**
sys.**__displayhook__**
sys.**__excepthook__**

sys.**__unraisablehook__**
程序开始时，这些对象存有 breakpointhook、displayhook、excepthook 和 unraisablehook 的初始值。保存它们是为了可以在 breakpointhook、displayhook 和 excepthook、unraisablehook 被破坏或被替换时恢复它们。

3.7 版新加入: __breakpointhook__

3.8 版新加入: __unraisablehook__

sys.**exc_info**()
本函数返回的元组包含三个值，它们给出当前正在处理的异常的信息。返回的信息仅限于当前线程和当前堆栈帧。如果当前堆栈帧没有正在处理的异常，则信息将从下级被调用的堆栈帧或上级调用者等位置获取，依此类推，直到找到正在处理异常的堆栈帧为止。此处的“处理异常”指的是“执行 except 子句”。任何堆栈帧都只能访问当前正在处理的异常的信息。

如果整个堆栈都没有正在处理的异常，则返回包含三个 None 值的元组。否则返回值为 (type, value, traceback)。它们的含义是：*type* 是正在处理的异常类型（它是*BaseException* 的子类）；*value* 是异常实例（异常类型的实例）；*traceback* 是一个 回溯对象，该对象封装了最初发生异常时的调用堆栈。

sys.**exec_prefix**
一个字符串，提供特定域的目录前缀，该目录中安装了与平台相关的 Python 文件，默认也是 '/usr/local'。该目录前缀可以在构建时使用 **configure** 脚本的 --exec-prefix 参数进行设置。具体而言，所有配置文件（如 pyconfig.h 头文件）都安装在目录 *exec_prefix*/lib/python*X.Y*/config 中，共享库模块安装在 *exec_prefix*/lib/python*X.Y*/lib-dynload 中，其中 *X.Y* 是 Python 的版本号，如 3.2。

備F: 如果在一个虚拟环境 中，那么该值将在 site.py 中被修改，指向虚拟环境。Python 安装位置仍然可以用*base_exec_prefix* 来获取。

---

sys.**executable**
一个字符串，提供 Python 解释器的可执行二进制文件的绝对路径，仅在部分系统中此值有意义。如果 Python 无法获取其可执行文件的真实路径，则*sys.executable* 将为空字符串或 None。

sys.**exit**([*arg*])
Raise a *SystemExit* exception, signaling an intention to exit the interpreter.

可选参数 *arg* 可以是表示退出状态的整数（默认为 0），也可以是其他类型的对象。如果它是整数，则 shell 等将 0 视为“成功终止”，非零值视为“异常终止”。大多数系统要求该值的范围是 0--127，否则会产生不确定的结果。某些系统为退出代码约定了特定的含义，但通常尚不完善；Unix 程序通常用 2 表示命令行语法错误，用 1 表示所有其他类型的错误。传入其他类型的对象，如果传入 None 等同于传入 0，如果传入其他对象则将其打印至*stderr*，且退出代码为 1。特别地，sys.exit("some error message") 可以在发生错误时快速退出程序。

Since *exit()* ultimately ”only” raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted. Cleanup actions specified by finally clauses of try statements are honored, and it is possible to intercept the exit attempt at an outer level.

3.6 版更變: 在 Python 解释器捕获*SystemExit* 后，如果在清理中发生错误（如清除标准流中的缓冲数据时出错），则退出状态码将变为 120。

sys.**flags**
具名元组 *flags* 含有命令行标志的状态。这些属性是只读的。

属性	标志
debug	-d
inspect	-i
interactive	-i
isolated	-I
optimize	-O 或 -OO
dont_write_bytecode	-B
no_user_site	-s
no_site	-S
ignore_environment	-E
verbose	-v
bytes_warning	-b
quiet	-q
hash_randomization	-R
dev_mode	-X dev (*Python 开发模式*)
utf8_mode	-X utf8
int_max_str_digits	-X int_max_str_digits (*integer string conversion length limitation*)

3.2 版更變: 为新的 -q 标志添加了 quiet 属性

3.2.3 版新加入: hash_randomization 属性

3.3 版更變: 删除了过时的 division_warning 属性

3.4 版更變: 为 -I isolated 标志添加了 isolated 属性。

3.7 版更變: 为新的*Python 开发模式* 添加了 dev_mode 属性，为新的 -X utf8 标志添加了 utf8_mode 属性。

3.9.14 版更變: Added the int_max_str_digits attribute.

sys.**float_info**

一个*具名元组*，存有浮点型的相关信息。它包含的是关于精度和内部表示的底层信息。这些值与标准头文件 float.h 中为 C 语言定义的各种浮点常量对应，详情请参阅 1999 ISO/IEC C 标准 [C99] 的 5.2.4.2.2 节，'Characteristics of floating types（浮点型的特性）'。

属性	float.h 宏	说明
epsilon	DBL_EPSILON	大于 1.0 的最小值和 1.0 之间的差，表示为浮点数 另请参阅*math.ulp()*。
dig	DBL_DIG	浮点数可以真实表示的最大十进制数字；见下文
mant_dig	DBL_MANT_DIG	浮点数精度：radix 基数下的浮点数有效位数
max	DBL_MAX	可表示的最大正浮点数（非无穷）
max_exp	DBL_MAX_EXP	使得 radix**(e-1) 是可表示的浮点数（非无穷）的最大整数 *e*
max_10_exp	DBL_MAX_10_EXP	使得 10**e 在可表示的浮点数（非无穷）范围内的最大整数 *e*
min	DBL_MIN	可表示的最小正 规格化浮点数 使用*math.ulp(0.0)* 获取可表示的最小正 非规格化浮点数
min_exp	DBL_MIN_EXP	使得 radix**(e-1) 是规格化浮点数的最小整数 *e*
min_10_exp	DBL_MIN_10_EXP	使得 10**e 是规格化浮点数的最小整数 *e*
radix	FLT_RADIX	指数表示法中采用的基数
rounds	FLT_ROUNDS	整数常数，表示算术运算中的舍入方式。它反映了解释器启动时系统的 FLT_ROUNDS 宏的值。关于可能的值及其含义的说明，请参阅 C99 标准 5.2.4.2.2 节。

关于 sys.float_info.dig 属性的进一步说明。如果 s 是表示十进制数的字符串，而该数最多有 sys.float_info.dig 位有效数字，则将 s 转换为 float 再转回去将恢复原先相同十进制值的字符串:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'    # decimal string with 15 significant digits
>>> format(float(s), '.15g')  # convert to float and back -> same value
'3.14159265358979'
```

但是对于超过 sys.float_info.dig 位有效数字的字符串，转换前后并非总是相同:

```
>>> s = '9876543211234567'    # 16 significant digits is too many!
>>> format(float(s), '.16g')  # conversion changes value
'9876543211234568'
```

sys.**float_repr_style**
一个字符串，反映*repr()* 函数在浮点数上的行为。如果该字符串是 'short'，那么对于（非无穷的）浮点数 x，repr(x) 将会生成一个短字符串，满足 float(repr(x)) == x 的特性。这是 Python 3.1 及更高版本中的常见行为。否则 float_repr_style 的值将是 'legacy'，此时 repr(x) 的行为方式将与 Python 3.1 之前的版本相同。

3.1 版新加入.

sys.**getallocatedblocks**()
返回解释器当前已分配的内存块数，无论它们大小如何。本函数主要用于跟踪和调试内存泄漏。因为解释器有内部缓存，所以不同调用之间结果会变化。可能需要调用*_clear_type_cache()* 和*gc.collect()* 使结果更容易预测。

如果当前 Python 构建或实现无法合理地计算此信息，允许*getallocatedblocks()* 返回 0。

3.4 版新加入.

sys.**getandroidapilevel**()
返回一个整数，表示 Android 构建时 API 版本。

可用性：Android。

3.7 版新加入.

sys.**getdefaultencoding**()
返回当前 Unicode 实现所使用的默认字符串编码名称。

sys.**getdlopenflags**()
返回当前 dlopen() 调用所使用的标志位的值。标志值对应的符号名称可以在*os* 模块中找到（形如 RTLD_xxx 的常量，如*os.RTLD_LAZY* ）。

可用性: Unix。

sys.**getfilesystemencoding**()
返回编码名称，该编码用于在 Unicode 文件名和 bytes 文件名之间转换。为获得最佳兼容性，任何时候都应使用 str 表示文件名，尽管用字节来表示文件名也是支持的。函数如果需要接受或返回文件名，它应支持 str 或 bytes，并在内部将其转换为系统首选的表示形式。

该编码始终是 ASCII 兼容的。

应使用*os.fsencode()* 和*os.fsdecode()* 来保证所采用的编码和错误处理方案都是正确的。

- 在 UTF-8 模式下，任何平台上的编码均为 utf-8。
- 在 macOS 上，编码为 'utf-8'。
- 在 Unix 上，编码是语言环境编码。

- 在 Windows 上取决于用户配置，编码可能是 'utf-8' 或 'mbcs'。
- 在 Android 上，编码为 'utf-8'。
- 在 VxWorks 上，编码为 'utf-8'。

3.2 版更變: *getfilesystemencoding()* 的结果将不再有可能是 None。

3.6 版更變: Windows 不再保证会返回 'mbcs'。详情请参阅 **PEP 529** 和*_enablelegacywindowsfsencoding()*。

3.7 版更變: 在 UTF-8 模式下返回'utf-8' 。

sys.**getfilesystemencodeerrors**()
返回错误处理方案的名称，该错误处理方案将在 Unicode 文件名和 bytes 文件名转换时生效。编码的名称是由*getfilesystemencoding()* 返回的。

应使用*os.fsencode()* 和*os.fsdecode()* 来保证所采用的编码和错误处理方案都是正确的。

3.6 版新加入.

sys.**get_int_max_str_digits**()
Returns the current value for the *integer string conversion length limitation*. See also *set_int_max_str_digits()*.

3.9.14 版新加入.

sys.**getrefcount**(*object*)
返回 *object* 的引用计数。返回的计数通常比预期的多一，因为它包括了作为*getrefcount()* 参数的这一次（临时）引用。

sys.**getrecursionlimit**()
返回当前的递归限制值，即 Python 解释器堆栈的最大深度。此限制可防止无限递归导致的 C 堆栈溢出和 Python 崩溃。该值可以通过*setrecursionlimit()* 设置。

sys.**getsizeof**(*object*[, *default*])
返回对象的大小（以字节为单位）。该对象可以是任何类型。所有内建对象返回的结果都是正确的，但对于第三方扩展不一定正确，因为这与具体实现有关。

只计算直接分配给对象的内存消耗，不计算它所引用的对象的内存消耗。

对象不提供计算大小的方法时，如果传入过 *default* 则返回它，否则抛出*TypeError* 异常。

如果对象由垃圾回收器管理，则*getsizeof()* 将调用对象的 __sizeof__ 方法，并在上层添加额外的垃圾回收器。

可以参考 recursive sizeof recipe 中的示例，关于递归调用*getsizeof()* 来得到各个容器及其所有内容物的大小。

sys.**getswitchinterval**()
返回解释器的“线程切换间隔时间”，请参阅*setswitchinterval()*。

3.2 版新加入.

sys.**_getframe**([*depth*])
返回来自调用栈的一个帧对象。如果传入可选整数 *depth*，则返回从栈顶往下相应调用层数的帧对象。如果该数比调用栈更深，则抛出*ValueError*。*depth* 的默认值是 0，返回调用栈顶部的帧。

引发一个审计事件 sys._getframe，没有附带参数。

CPython implementation detail: 这个函数应该只在内部为了一些特定的目的使用。不保证它在所有 Python 实现中都存在。

sys.**getprofile**()
返回由*setprofile()* 设置的性能分析函数。

sys.**gettrace**()
返回由*settrace()* 设置的跟踪函数。

CPython implementation detail: *gettrace()* 函数仅用于实现调试器，性能分析器，打包工具等。它的行为是实现平台的一部分，而不是语言定义的一部分，因此并非在所有 Python 实现中都可用。

sys.**getwindowsversion**()
返回一个具名元组，描述当前正在运行的 Windows 版本。元素名称包括 *major*, *minor*, *build*, *platform*, *service_pack*, *service_pack_minor*, *service_pack_major*, *suite_mask*, *product_type* 和 *platform_version*。*service_pack* 包含一个字符串，*platform_version* 包含一个三元组，其他所有值都是整数。元素也可以通过名称来访问，所以 `sys.getwindowsversion()[0]` 与 `sys.getwindowsversion().major` 是等效的。为保持与旧版本的兼容性，只有前 5 个元素可以用索引检索。

platform 将会是 `2 (VER_PLATFORM_WIN32_NT)`。

product_type 可能是以下值之一：

常数	含义
`1 (VER_NT_WORKSTATION)`	系统是工作站。
`2 (VER_NT_DOMAIN_CONTROLLER)`	系统是域控制器。
`3 (VER_NT_SERVER)`	系统是服务器，但不是域控制器。

本函数包装了 Win32 `GetVersionEx()` 函数，参阅 Microsoft 文档有关 `OSVERSIONINFOEX()` 的内容可获取这些字段的更多信息。

platform_version 返回当前操作系统的主要版本、次要版本和编译版本号，而不是为该进程所模拟的版本。它旨在用于日志记录而非特性检测。

備🄵: *platform_version* 会从 kernel32.dll 获取版本号，这个版本可能与 OS 版本不同。请使用*platform* 模块来获取准确的 OS 版本号。

可用性: Windows。

3.2 版更變: 更改为具名元组，添加 *service_pack_minor*, *service_pack_major*, *suite_mask* 和 *product_type*。

3.6 版更變: 添加了 *platform_version*

sys.**get_asyncgen_hooks**()
返回一个 *asyncgen_hooks* 对象，该对象类似于*namedtuple*，形式为 *(firstiter, finalizer)*，其中 *firstiter* 和 *finalizer* 为 `None` 或函数，函数以异步生成器迭代器 作为参数，并用于在事件循环中干预异步生成器的终结。

3.6 版新加入: 详情请参阅 **PEP 525**。

備🄵: 本函数已添加至暂定软件包（详情请参阅 **PEP 411** ）。

---

sys.**get_coroutine_origin_tracking_depth**()
获取由*set_coroutine_origin_tracking_depth()* 设置的协程来源的追踪深度。

3.7 版新加入.

備🄵: 本函数已添加至暂定软件包（详情请参阅 **PEP 411** ）。仅将其用于调试目的。

---

sys.**hash_info**
一个具名元组，给出数字类型的哈希的实现参数。关于数字类型的哈希的详情请参阅数字类型的哈希运算。

属性	说明
width	用于哈希值的位宽度
modulus	用于数字散列方案的素数模数 P。
inf	为正无穷大返回的哈希值
nan	为 nan 返回的哈希值
imag	用于复数虚部的乘数
algorithm	字符串、字节和内存视图的哈希算法的名称
hash_bits	哈希算法的内部输出大小。
seed_bits	散列算法的种子密钥的大小

3.2 版新加入.

3.4 版更變: 添加了 *algorithm*, *hash_bits* 和 *seed_bits*

sys.**hexversion**
编码为单个整数的版本号。该整数会确保每个版本都自增，其中适当包括了未发布版本。举例来说，要测试 Python 解释器的版本不低于 1.5.2，请使用:

```
if sys.hexversion >= 0x010502F0:
    # use some advanced feature
    ...
else:
    # use an alternative implementation or warn the user
    ...
```

之所以称它为 hexversion，是因为只有将它传入内置函数 *hex()* 后，其结果才看起来有意义。也可以使用具名元组 *sys.version_info*，它对相同信息有着更人性化的编码。

关于 hexversion 的更多信息可以在 apiabiversion 中找到。

sys.**implementation**
一个对象，该对象包含当前运行的 Python 解释器的实现信息。所有 Python 实现中都必须存在下列属性。

name 是当前实现的标识符，如 'cpython'。实际的字符串由 Python 实现定义，但保证是小写字母。

version 是一个具名元组，格式与 *sys.version_info* 相同。它表示 Python 实现的版本。另一个（由 sys.version_info 表示）是当前解释器遵循的相应 Python 语言的版本，两者具有不同的含义。例如，对于 PyPy 1.8，sys.implementation.version 可能是 sys.version_info(1, 8, 0, 'final', 0)，而 sys.version_info 则是 sys.version_info(2, 7, 2, 'final', 0)。对于 CPython 而言两个值是相同的，因为它是参考实现。

hexversion 是十六进制的实现版本，类似于 *sys.hexversion*。

cache_tag 是导入机制使用的标记，用于已缓存模块的文件名。按照惯例，它将由实现的名称和版本组成，如 'cpython-33'。但如果合适，Python 实现可以使用其他值。如果 cache_tag 被置为 None，表示模块缓存已禁用。

sys.implementation 可能包含相应 Python 实现的其他属性。这些非标准属性必须以下划线开头，此处不详细阐述。无论其内容如何，*sys.implementation* 在解释器运行期间或不同实现版本之间都不会更改。（但是不同 Python 语言版本间可能会不同。）详情请参阅 **PEP 421**。

3.3 版新加入.

備🄵: 新的必要属性的添加必须经过常规的 PEP 过程。详情请参阅 **PEP 421**。

---

sys.**int_info**
一个具名元组，包含 Python 内部整数表示形式的信息。这些属性是只读的。

属性	说明
bits_per_digit	每个数字占有的位数。Python 内部将整数存储在基底 2**int_info.bits_per_digit
sizeof_digit	用于表示数字的 C 类型的字节大小
default_max_str_digits	default value for *sys.get_int_max_str_digits()* when it is not otherwise explicitly configured.
str_digits_check_threshold	minimum non-zero value for *sys.set_int_max_str_digits()*, PYTHONINTMAXSTRDIGITS, or -X int_max_str_digits.

3.1 版新加入.

3.9.14 版更變: Added default_max_str_digits and str_digits_check_threshold.

sys.**__interactivehook__**
当本属性存在，则以 交互模式启动解释器时，将自动（不带参数地）调用本属性的值。该过程是在读取 PYTHONSTARTUP 文件之后完成的，所以可以在该文件中设置这一钩子。*site* 模块设置了这一属性。

引发一个审计事件 cpython.run_interactivehook，附带参数 hook。

3.4 版新加入.

sys.**intern**(*string*)
将 *string* 插入"interned"（驻留）字符串表，返回被插入的字符串 -- 它是 *string* 本身或副本。驻留字符串对提高字典查找的性能很有用 -- 如果字典中的键已驻留，且所查找的键也已驻留，则键（取散列后）的比较可以用指针代替字符串来比较。通常，Python 程序使用到的名称会被自动驻留，且用于保存模块、类或实例属性的字典的键也已驻留。

驻留字符串不是永久存在的，对*intern()* 返回值的引用必须保留下来，才能发挥驻留字符串的优势。

sys.**is_finalizing**()
如果 Python 解释器正在关闭 则返回*True*，否则返回*False*。

3.5 版新加入.

sys.**last_type**
sys.**last_value**
sys.**last_traceback**
这三个变量并非总是有定义，仅当有异常未处理，且解释器打印了错误消息和堆栈回溯时，才会给它们赋值。它们的预期用途，是允许交互中的用户导入调试器模块，进行事后调试，而不必重新运行导致错误的命令。（通常使用 import pdb; pdb.pm() 进入事后调试器，详情请参阅*pdb* 模块。）

这些变量的含义与上述*exc_info()* 返回值的含义相同。

sys.**maxsize**
一个整数，表示 Py_ssize_t 类型的变量可以取到的最大值。在 32 位平台上通常为 2**31 - 1，在 64 位平台上通常为 2**63 - 1。

sys.**maxunicode**
一个整数，表示最大的 Unicode 码点值，如 1114111 （十六进制为 0x10FFFF ）。

3.3 版更變: 在 **PEP 393** 之前，sys.maxunicode 曾是 0xFFFF 或 0x10FFFF，具体取决于配置选项，该选项指定将 Unicode 字符存储为 UCS-2 还是 UCS-4。

sys.**meta_path**
一个由元路径查找器 对象组成的列表，当查找需要导入的模块时，会调用这些对象的*find_spec()*

方法，观察这些对象是否能找到所需模块。调用*find_spec()* 方法最少需要传入待导入模块的绝对名称。如果待导入模块包含在一个包中，则父包的 __path__ 属性将作为第二个参数被传入。该方法返回模块规格，找不到模块则返回 None。

也参考:

importlib.abc.MetaPathFinder 抽象基类，定义了*meta_path* 内的查找器对象的接口。

importlib.machinery.ModuleSpec *find_spec()* 返回的实例所对应的具体类。

3.4 版更變: 在 Python 3.4 中通过 **PEP 451** 引入了模块规格。早期版本的 Python 会寻找一个称为*find_module()* 的方法。如果某个*meta_path* 条目没有*find_spec()* 方法，就会回退去调用前一种方法。

sys.**modules**
一个字典，将模块名称映射到已加载的模块。可以操作该字典来强制重新加载模块，或是实现其他技巧。但是，替换的字典不一定会按预期工作，并且从字典中删除必要的项目可能会导致 Python 崩溃。

sys.**path**
一个由字符串组成的列表，用于指定模块的搜索路径。初始化自环境变量 PYTHONPATH，再加上一条与安装有关的默认路径。

程序启动时将初始化本列表，列表的第一项 path[0] 目录含有调用 Python 解释器的脚本。如果脚本目录不可用（比如以交互方式调用了解释器，或脚本是从标准输入中读取的），则 path[0] 为空字符串，这将导致 Python 优先搜索当前目录中的模块。注意，脚本目录将插入在 PYTHONPATH 的条目 * 之前 *。

程序可以随意修改本列表用于自己的目的。只能向*sys.path* 中添加 string 和 bytes 类型，其他数据类型将在导入期间被忽略。

也参考:

site 模块，该模块描述了如何使用.pth 文件来扩展*sys.path*。

sys.**path_hooks**
一个由可调用对象组成的列表，这些对象接受一个路径作为参数，并尝试为该路径创建一个查找器。如果成功创建查找器，则可调用对象将返回它，否则将引发*ImportError* 异常。

本特性最早在 **PEP 302** 中被提及。

sys.**path_importer_cache**
一个字典，作为查找器 对象的缓存。key 是传入*sys.path_hooks* 的路径，value 是相应已找到的查找器。如果路径是有效的文件系统路径，但在*sys.path_hooks* 中未找到查找器，则存入 None。

本特性最早在 **PEP 302** 中被提及。

3.3 版更變: 未找到查找器时，改为存储 None，而不是*imp.NullImporter*。

sys.**platform**
本字符串是一个平台标识符，举例而言，该标识符可用于将特定平台的组件追加到*sys.path* 中。

对于 Unix 系统（除 Linux 和 AIX 外），该字符串是 *Python* 构建时的 uname -s 返回的小写操作系统名称，并附加了 uname -r 返回的系统版本的第一部分，如 'sunos5' 或 'freebsd8'。除非需要检测特定版本的系统，否则建议使用以下习惯用法:

```
if sys.platform.startswith('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
elif sys.platform.startswith('aix'):
    # AIX-specific code here...
```

对于其他系统，值是：

系统	平台值
AIX	`'aix'`
Linux	`'linux'`
Windows	`'win32'`
Windows/Cygwin	`'cygwin'`
macOS	`'darwin'`

3.3 版更變: 在 Linux 上，*sys.platform* 将不再包含副版本号。它将总是 `'linux'` 而不是 `'linux2'` 或 `'linux3'`。由于旧版本的 Python 会包含该版本号，因此推荐总是使用上述 `startswith` 习惯用法。

3.8 版更變: 在 AIX 上，*sys.platform* 将不再包含副版本号。它将总是 `'aix'` 而不是 `'aix5'` 或 `'aix7'`。由于旧版本的 Python 会包含该版本号，因此推荐总是使用上述 `startswith` 习惯用法。

也参考:

os.name 更加简略。*os.uname()* 提供系统的版本信息。

platform 模块提供了对系统标识更详细的检查。

sys.**platlibdir**
平台专用库目录。用于构建标准库的路径和已安装扩展模块的路径。

在大多数平台上，它等同于 `"lib"` 。在 Fedora 和 SuSE 上，它等同于给出了以下 `sys.path` 路径的 64 位平台上的 `"lib64"` （其中 `X.Y` 是 Python 的 `major.minor` 版本）。

- `/usr/lib64/pythonX.Y/`: 标准库（如*os* 模块的 `os.py` ）
- `/usr/lib64/pythonX.Y/lib-dynload/`: 标准库的 C 扩展模块（如*errno* 模块，确切的文件名取决于平台）
- `/usr/lib/pythonX.Y/site-packages/` (请使用 `lib`, 而非*sys.platlibdir*): 第三方模块
- `/usr/lib64/pythonX.Y/site-packages/`: 第三方包的 C 扩展模块

3.9 版新加入.

sys.**prefix**
A string giving the site-specific directory prefix where the platform independent Python files are installed; on Unix, the default is `'/usr/local'`. This can be set at build time with the `--prefix` argument to the **configure** script. See *安装路径* for derived paths.

備𝔽: 如果在一个*虚拟环境* 中，那么该值将在 `site.py` 中被修改，指向虚拟环境。Python 安装位置仍然可以用*base_prefix* 来获取。

---

sys.**ps1**
sys.**ps2**
字符串，指定解释器的首要和次要提示符。仅当解释器处于交互模式时，它们才有定义。这种情况下，它们的初值为 `'>>> '` 和 `'... '`。如果赋给其中某个变量的是非字符串对象，则每次解释器准备读取新的交互式命令时，都会重新运行该对象的*str()*，这可以用来实现动态的提示符。

sys.**setdlopenflags**(*n*)
设置解释器在调用 `dlopen()` 时用到的标志，例如解释器在加载扩展模块时。首先，调用 `sys.setdlopenflags(0)` 将在导入模块时对符号启用惰性解析。要在扩展模块之间共享符号，请调用 `sys.setdlopenflags(os.RTLD_GLOBAL)`。标志值的符号名称可以在*os* 模块中找到（即 `RTLD_xxx` 常量，如*os.RTLD_LAZY* ）。

可用性: Unix。

sys.**set_int_max_str_digits**(*n*)
Set the *integer string conversion length limitation* used by this interpreter. See also *get_int_max_str_digits()*.

3.9.14 版新加入.

sys.**setprofile**(*profilefunc*)
设置系统的性能分析函数，该函数使得在 Python 中能够实现一个 Python 源代码性能分析器。关于 Python Profiler 的更多信息请参阅*Python Profilers 分析器* 章节。性能分析函数的调用方式类似于系统的跟踪函数（参阅*settrace()*），但它是通过不同的事件调用的，例如，不是每执行一行代码就调用它一次（仅在调用某函数和从某函数返回时才会调用性能分析函数，但即使某函数发生异常也会算作返回事件）。该函数是特定于单个线程的，但是性能分析器无法得知线程之间的上下文切换，因此在存在多个线程的情况下使用它是没有意义的。另外，因为它的返回值不会被用到，所以可以简单地返回 None。性能分析函数中的错误将导致其自身被解除设置。

性能分析函数应接收三个参数：*frame*、*event* 和 *arg*。*frame* 是当前的堆栈帧。*event* 是一个字符串：'call'、'return'、'c_call'、'c_return' 或 'c_exception'。*arg* 取决于事件类型。

引发一个审计事件 sys.setprofile，不附带任何参数。

这些事件具有以下含义：

'call' 表示调用了某个函数（或进入了其他的代码块）。性能分析函数将被调用，*arg* 为 None。

'return' 表示某个函数（或别的代码块）即将返回。性能分析函数将被调用，*arg* 是即将返回的值，如果此次返回事件是由于抛出异常，*arg* 为 None。

'c_call' 表示即将调用某个 C 函数。它可能是扩展函数或是内建函数。*arg* 是 C 函数对象。

'c_return' 表示返回了某个 C 函数。*arg* 是 C 函数对象。

'c_exception' 表示某个 C 函数抛出了异常。*arg* 是 C 函数对象。

sys.**setrecursionlimit**(*limit*)
将 Python 解释器堆栈的最大深度设置为 *limit*。此限制可防止无限递归导致的 C 堆栈溢出和 Python 崩溃。

不同平台所允许的最高限值不同。当用户有需要深度递归的程序且平台支持更高的限值，可能就需要调高限值。进行该操作需要谨慎，因为过高的限值可能会导致崩溃。

如果新的限值低于当前的递归深度，将抛出*RecursionError* 异常。

3.5.1 版更變: 如果新的限值低于当前的递归深度，现在将抛出*RecursionError* 异常。

sys.**setswitchinterval**(*interval*)
设置解释器的线程切换间隔时间（单位为秒）。该浮点数决定了“时间片”的理想持续时间，时间片将分配给同时运行的 Python 线程。请注意，实际值可能更高，尤其是使用了运行时间长的内部函数或方法时。同时，在时间间隔末尾调度哪个线程是操作系统的决定。解释器没有自己的调度程序。

3.2 版新加入.

sys.**settrace**(*tracefunc*)
设置系统的跟踪函数，使得用户在 Python 中就可以实现 Python 源代码调试器。该函数是特定于单个线程的，所以要让调试器支持多线程，必须为正在调试的每个线程都用*settrace()* 注册一个跟踪函数，或使用*threading.settrace()*。

跟踪函数应接收三个参数：*frame*、*event* 和 *arg*。*frame* 是当前的堆栈帧。*event* 是一个字符串：'call'、'line'、'return'、'exception' 或 'opcode'。*arg* 取决于事件类型。

每次进入 trace 函数的新的局部作用范围，都会调用 trace 函数（*event* 会被设置为 'call'），它应该返回一个引用，指向即将用在新作用范围上的局部跟踪函数；如果不需要跟踪当前的作用范围，则返回 None。

局部跟踪函数应返回对自身的引用（或对另一个函数的引用，用来在其作用范围内进行进一步的跟踪），或者返回 None 来停止跟踪其作用范围。

如果跟踪函数出错，则该跟踪函数将被取消设置，类似于调用 settrace(None)。

这些事件具有以下含义：

'call' 表示调用了某个函数（或进入了其他的代码块）。全局跟踪函数将被调用，*arg* 为 None。返回值将指定局部跟踪函数。

'line' 表示解释器即将执行新一行代码或重新执行循环条件。局部跟踪函数将被调用，*arg* 为 None，其返回值将指定新的局部跟踪函数。关于其工作原理的详细说明，请参见 Objects/lnotab_notes.txt。要在该堆栈帧禁用每行触发事件，可以在堆栈帧上将 f_trace_lines 设置为*False*。

'return' 表示某个函数（或别的代码块）即将返回。局部跟踪函数将被调用，*arg* 是即将返回的值，如果此次返回事件是由于抛出异常，*arg* 为 None。跟踪函数的返回值将被忽略。

'exception' 表示发生了某个异常。局部跟踪函数将被调用，*arg* 是一个 (exception, value, traceback) 元组，返回值将指定新的局部跟踪函数。

'opcode' 表示解释器即将执行一个新的操作码（操作码的详情请参阅*dis*）。局部跟踪函数将被调用，*arg* 为 None，其返回值将指定新的局部跟踪函数。每操作码触发事件默认情况下都不发出：必须在堆栈帧上将 f_trace_opcodes 显式地设置为*True* 来请求这些事件。

注意，由于异常是在链式调用中传播的，所以每一级都会产生一个 'exception' 事件。

更细微的用法是，可以显式地通过赋值 frame.f_trace = tracefunc 来设置跟踪函数，而不是用现有跟踪函数的返回值去间接设置它。当前帧上的跟踪函数必须激活，而*settrace()* 还没有做这件事。注意，为了使上述设置起效，必须使用*settrace()* 来安装全局跟踪函数才能启用运行时跟踪机制，但是它不必与上述是同一个跟踪函数（它可以是一个开销很低的跟踪函数，只返回 None，即在各个帧上立即将其自身禁用）。

关于代码对象和帧对象的更多信息请参考 types。

引发一个审计事件 sys.settrace，不附带任何参数。

CPython implementation detail: *settrace()* 函数仅用于实现调试器，性能分析器，打包工具等。它的行为是实现平台的一部分，而不是语言定义的一部分，因此并非在所有 Python 实现中都可用。

3.7 版更變: 添加了 'opcode' 事件类型；为帧对象添加了 f_trace_lines 和 f_trace_opcodes 属性

sys.**set_asyncgen_hooks**(*firstiter*, *finalizer*)
接受两个可选的关键字参数，要求它们是可调用对象，且接受一个异步生成器迭代器 作为参数。*firstiter* 对象将在异步生成器第一次迭代时调用。*finalizer* 将在异步生成器即将被销毁时调用。

引发一个审计事件 sys.set_asyncgen_hooks_firstiter，不附带任何参数。

引发一个审计事件 sys.set_asyncgen_hooks_finalizer，不附带任何参数。

之所以会引发两个审计事件，是因为底层的 API 由两个调用组成，每个调用都须要引发自己的事件。

3.6 版新加入: 更多详情请参阅 **PEP 525**，*finalizer* 方法的参考示例可参阅 Lib/asyncio/base_events.py 中 asyncio.Loop.shutdown_asyncgens 的实现。

備🄵: 本函数已添加至暂定软件包（详情请参阅 **PEP 411**）。

---

sys.**set_coroutine_origin_tracking_depth**(*depth*)
用于启用或禁用协程溯源。启用后，协程对象上的 cr_origin 属性将包含一个元组，它由多个（文件名 filename，行号 line number，函数名 function name）元组组成，整个元组描述出了协程对象创建过程的回溯，元组首端是最近一次的调用。禁用后，cr_origin 将为 None。

要启用，请向 *depth* 传递一个大于零的值，它指定了有多少帧将被捕获信息。要禁用，请将 *depth* 置为零。

该设置是特定于单个线程的。

3.7 版新加入.

備𝔽: 本函数已添加至暂定软件包（详情请参阅 **PEP 411** ）。仅将其用于调试目的。

---

sys.**_enablelegacywindowsfsencoding**()
将默认文件系统编码和错误处理方案分别更改为'mbcs' 和'replace'，这是为了与 Python 3.6 前的版本保持一致。

这等同于在启动 Python 前先定义好 PYTHONLEGACYWINDOWSFSENCODING 环境变量。

可用性: Windows。

3.6 版新加入: 更多详情请参阅 **PEP 529**。

sys.**stdin**
sys.**stdout**
sys.**stderr**
解释器用于标准输入、标准输出和标准错误的文件对象:

- stdin 用于所有交互式输入（包括对*input()* 的调用）;
- stdout 用于*print()* 和*expression* 语句的输出，以及用于*input()* 的提示符;
- 解释器自身的提示符和它的错误消息都发往 stderr。

这些流都是常规文本文件，与*open()* 函数返回的对象一致。它们的参数选择如下:

- 字符编码取决于各个平台。在非 Windows 平台上使用的是语言环境 (locale) 编码（可参阅*locale.getpreferredencoding()* ）。

  在 Windows 上，控制台设备使用 UTF-8 编码。非字符设备（如磁盘文件和管道）使用系统语言环境编码（即 ANSI 代码页）。非控制台字符设备（即 isatty() 返回的是 True，如 NUL）在启动时，会把控制台输入代码页和输出代码页的值分别用于 stdin 和 stdout/stderr。如果进程原本没有附加到控制台，则默认为系统语言环境编码。

  要重写控制台的特殊行为，可以在启动 Python 前设置 PYTHONLEGACYWINDOWSSTDIO 环境变量。此时，控制台代码页将用于其他字符设备。

  在所有平台上，都可以通过在 Python 启动前设置 PYTHONIOENCODING 环境变量来重写字符编码，或通过新的 -X utf8 命令行选项和 PYTHONUTF8 环境变量来设置。但是，对 Windows 控制台来说，上述方法仅在设置了 PYTHONLEGACYWINDOWSSTDIO 后才起效。

- 交互模式下，stdout 流是行缓冲的。其他情况下，它像常规文本文件一样是块缓冲的。两种情况下的 stderr 流都是行缓冲的。要使得两个流都变成无缓冲，可以传入 -u 命令行选项或设置 PYTHONUNBUFFERED 环境变量。

3.9 版更變: 非交互模式下，stderr 现在是行缓冲的，而不是全缓冲的。

備𝔽: 要从标准流写入或读取二进制数据，请使用底层二进制*buffer* 对象。例如，要将字节写入*stdout*，请使用 sys.stdout.buffer.write(b'abc')。

但是，如果你在写一个库（并且不限制执行库代码时的上下文），那么请注意，标准流可能会被替换为文件类对象，如*io.StringIO*，它们是不支持 buffer 属性的。

---

sys.**__stdin__**

sys.**__stdout__**
sys.**__stderr__**
程序开始时，这些对象存有 stdin、stderr 和 stdout 的初始值。它们在程序结束前都可以使用，且在需要向实际的标准流打印内容时很有用，无论 sys.std* 对象是否已重定向。

如果实际文件已经被覆盖成一个损坏的对象了，那它也可用于将实际文件还原成能正常工作的文件对象。但是，本过程的最佳方法应该是，在原来的流被替换之前就显式地保存它，并使用这一保存的对象来还原。

備F: 某些情况下的 stdin、stdout 和 stderr 以及初始值 __stdin__、__stdout__ 和 __stderr__ 可以是 None。通常发生在未连接到控制台的 Windows GUI app 中，以及在用 **pythonw** 启动的 Python app 中。

---

sys.**thread_info**
一个具名元组，包含线程实现的信息。

属性	说明
name	线程实现的名称： • 'nt': Windows 线程 • 'pthread': POSIX 线程 • 'solaris': Solaris 线程
lock	锁实现的名称： • 'semaphore': 锁使用信号量 • 'mutex+cond': 锁使用互斥和条件变量 • None 如果此信息未知
version	线程库的名称和版本。它是一个字符串，如果此信息未知，则为 None 。

3.3 版新加入.

sys.**tracebacklimit**
当该变量值设置为整数，在发生未处理的异常时，它将决定打印的回溯信息的最大层级数。默认为 1000。当将其设置为 0 或小于 0，将关闭所有回溯信息，并且只打印异常类型和异常值。

sys.**unraisablehook**(*unraisable*, /)
处理一个无法抛出的异常。

它会在发生了一个异常但 Python 没有办法处理时被调用。例如，当一个析构器引发了异常，或在垃圾回收 (*gc.collect()*) 期间引发了异常。

unraisable 参数具有以下属性:

- *exc_type*: 异常类型
- *exc_value*: 异常值，可以是 None.
- *exc_traceback*: 异常回溯，可以是 None.
- *err_msg*: 错误信息，可以是 None.
- *object*: 导致异常的对象，可以为 None.

默认的钩子程序会将 *err_msg* 和 *object* 格式化为：f'{err_msg}: {object!r}'；如果 *err_msg* 为 None 则采用”Exception ignored in” 错误信息。

要改变无法抛出的异常的处理过程，可以重写*sys.unraisablehook()*。

使用定制钩子存放 *exc_value* 可能会创建引用循环。它应当在不再需要异常时被显式地清空以打破引用循环。

如果一个 *object* 正在被销毁，那么使用自定义的钩子储存该对象可能会将其复活。请在自定义钩子生效后避免储存 *object*，以避免对象的复活。

另请参阅*excepthook()*，它处理未捕获的异常。

引发一个审计事件 `sys.unraisablehook` 并附带参数 `hook`, `unraisable`。

3.8 版新加入.

`sys.`**`version`**
一个包含 Python 解释器版本号加编译版本号以及所用编译器等额外信息的字符串。此字符串会在交互式解释器启动时显示。请不要从中提取版本信息，而应当使用*version_info* 以及*platform* 模块所提供的函数。

`sys.`**`api_version`**
这个解释器的 C API 版本。当你在调试 Python 及期扩展模板的版本冲突这个功能非常有用。

`sys.`**`version_info`**
一个包含版本号五部分的元组: *major*, *minor*, *micro*, *releaselevel* 和 *serial*。除 *releaselevel* 外的所有值均为整数；发布级别值则为 `'alpha'`, `'beta'`, `'candidate'` 或 `'final'`。对应于 Python 版本 2.0 的 `version_info` 值为 `(2, 0, 0, 'final', 0)`。这些部分也可按名称访问，因此 `sys.version_info[0]` 就等价于 `sys.version_info.major`，依此类推。

3.1 版更變: 增加了以名称表示的各部分属性。

`sys.`**`warnoptions`**
这是警告框架的一个实现细节；请不要修改此值。有关警告框架的更多信息请参阅*warnings* 模块。

`sys.`**`winver`**
用于在 Windows 平台上组成注册表键的版本号。这在 Python DLL 中存储为 1000 号字符串资源。其值通常是*version* 的头三个字符。它在*sys* 模块中提供是为了信息展示目的；修改此值不会影响 Python 所使用的注册表键。

可用性: Windows。

`sys.`**`_xoptions`**
一个字典，包含通过 `-X` 命令行选项传递的旗标，这些旗标专属于各种具体实现。选项名称将会映射到对应的值（如果显式指定）或者*True*。例如:

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython implementation detail: 这是 CPython 专属的访问通过 `-X` 传递的选项的方式。其他实现可能会通过其他方式导出它们，或者完全不导出。

3.2 版新加入.

引用

29.2 sysconfig ——提供对 Python 配置信息的访问支持

3.2 版新加入.

源代码： Lib/sysconfig.py

---

sysconfig 模块提供了对 Python 配置信息的访问支持，比如安装路径列表和有关当前平台的配置变量。

29.2.1 配置变量

Python 的发行版中包含一个 `Makefile` 和一个 `pyconfig.h` 头文件，这是构建 Python 二进制文件本身和用 *distutils* 编译的第三方 C 语言扩展所必需的。

sysconfig 将这些文件中的所有变量放在一个字典对象中，可用 *get_config_vars()* 或 *get_config_var()* 访问。

Notice that on Windows, it's a much smaller set.

`sysconfig.`**`get_config_vars`**`(`*args*`)`
With no arguments, return a dictionary of all configuration variables relevant for the current platform.

With arguments, return a list of values that result from looking up each argument in the configuration variable dictionary.

For each argument, if the value is not found, return `None`.

`sysconfig.`**`get_config_var`**`(`*name*`)`
Return the value of a single variable *name*. Equivalent to `get_config_vars().get(name)`.

If *name* is not found, return `None`.

用法示例:

```
>>> import sysconfig
>>> sysconfig.get_config_var('Py_ENABLE_SHARED')
0
>>> sysconfig.get_config_var('LIBDIR')
'/usr/local/lib'
>>> sysconfig.get_config_vars('AR', 'CXX')
['ar', 'g++']
```

29.2.2 安装路径

Python uses an installation scheme that differs depending on the platform and on the installation options. These schemes are stored in *sysconfig* under unique identifiers based on the value returned by *os.name*.

Every new component that is installed using *distutils* or a Distutils-based system will follow the same scheme to copy its file in the right places.

Python currently supports six schemes:

- *posix_prefix*: scheme for POSIX platforms like Linux or macOS. This is the default scheme used when Python or a component is installed.

- *posix_home*: scheme for POSIX platforms used when a *home* option is used upon installation. This scheme is used when a component is installed through Distutils with a specific home prefix.
- *posix_user*: scheme for POSIX platforms used when a component is installed through Distutils and the *user* option is used. This scheme defines paths located under the user home directory.
- *nt*: scheme for NT platforms like Windows.
- *nt_user*: scheme for NT platforms, when the *user* option is used.
- *osx_framework_user*: scheme for macOS, when the *user* option is used.

Each scheme is itself composed of a series of paths and each path has a unique identifier. Python currently uses eight paths:

- *stdlib*: directory containing the standard Python library files that are not platform-specific.
- *platstdlib*: directory containing the standard Python library files that are platform-specific.
- *platlib*: directory for site-specific, platform-specific files.
- *purelib*: directory for site-specific, non-platform-specific files.
- *include*: directory for non-platform-specific header files.
- *platinclude*: directory for platform-specific header files.
- *scripts*: directory for script files.
- *data*: directory for data files.

sysconfig provides some functions to determine these paths.

sysconfig.**get_scheme_names**()
Return a tuple containing all schemes currently supported in *sysconfig*.

sysconfig.**get_path_names**()
Return a tuple containing all path names currently supported in *sysconfig*.

sysconfig.**get_path**(*name*[, *scheme*[, *vars*[, *expand*]]])
Return an installation path corresponding to the path *name*, from the install scheme named *scheme*.

name has to be a value from the list returned by *get_path_names()*.

sysconfig stores installation paths corresponding to each path name, for each platform, with variables to be expanded. For instance the *stdlib* path for the *nt* scheme is: `{base}/Lib`.

get_path() will use the variables returned by *get_config_vars()* to expand the path. All variables have default values for each platform so one may call this function and get the default value.

If *scheme* is provided, it must be a value from the list returned by *get_scheme_names()*. Otherwise, the default scheme for the current platform is used.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary return by *get_config_vars()*.

If *expand* is set to `False`, the path will not be expanded using the variables.

If *name* is not found, raise a *KeyError*.

sysconfig.**get_paths**([*scheme*[, *vars*[, *expand*]]])
Return a dictionary containing all installation paths corresponding to an installation scheme. See *get_path()* for more information.

If *scheme* is not provided, will use the default scheme for the current platform.

If *vars* is provided, it must be a dictionary of variables that will update the dictionary used to expand the paths.

If *expand* is set to false, the paths will not be expanded.

If *scheme* is not an existing scheme, `get_paths()` will raise a `KeyError`.

29.2.3 其他功能

`sysconfig.`**`get_python_version`**`()`
Return the `MAJOR.MINOR` Python version number as a string. Similar to `'%d.%d' % sys.version_info[:2]`.

`sysconfig.`**`get_platform`**`()`
Return a string that identifies the current platform.

This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by 'os.uname()'), although the exact information included depends on the OS; e.g., on Linux, the kernel version isn't particularly important.

返回值的示例：

- linux-i586
- linux-alpha (?)
- solaris-2.6-sun4u

Windows 将返回以下之一：

- win-amd64 (在 AMD64, aka x86_64, Intel64, 和 EM64T 上的 64 位 Windows)
- win32 (all others - specifically, sys.platform is returned)

macOS can return:

- macosx-10.6-ppc
- macosx-10.4-ppc64
- macosx-10.3-i386
- macosx-10.4-fat

对于其他非-POSIX 平台，目前只是返回`sys.platform`。

`sysconfig.`**`is_python_build`**`()`
Return `True` if the running Python interpreter was built from source and is being run from its built location, and not from a location resulting from e.g. running `make install` or installing via a binary installer.

`sysconfig.`**`parse_config_h`**`(`*fp*[, *vars*]`)`
Parse a `config.h`-style file.

fp is a file-like object pointing to the `config.h`-like file.

A dictionary containing name/value pairs is returned. If an optional dictionary is passed in as the second argument, it is used instead of a new dictionary, and updated with the values read in the file.

`sysconfig.`**`get_config_h_filename`**`()`
返回 `pyconfig.h` 的目录

`sysconfig.`**`get_makefile_filename`**`()`
返回 `Makefile` 的目录

29.2.4 Using `sysconfig` as a script

You can use *sysconfig* as a script with Python's *-m* option:

```
$ python -m sysconfig
Platform: "macosx-10.4-i386"
Python version: "3.2"
Current installation scheme: "posix_prefix"

Paths:
        data = "/usr/local"
        include = "/Users/tarek/Dev/svn.python.org/py3k/Include"
        platinclude = "."
        platlib = "/usr/local/lib/python3.2/site-packages"
        platstdlib = "/usr/local/lib/python3.2"
        purelib = "/usr/local/lib/python3.2/site-packages"
        scripts = "/usr/local/bin"
        stdlib = "/usr/local/lib/python3.2"

Variables:
        AC_APPLE_UNIVERSAL_BUILD = "0"
        AIX_GENUINE_CPLUSPLUS = "0"
        AR = "ar"
        ARFLAGS = "rc"
        ...
```

This call will print in the standard output the information returned by *get_platform()*, *get_python_version()*, *get_path()* and *get_config_vars()*.

29.3 `builtins` --- 内建对象

该模块提供对 Python 的所有 "内置" 标识符的直接访问；例如，`builtins.open` 是内置函数的全名*open()*。请参阅𝔽建函式 和𝔽建常數 的文档。

大多数应用程序通常不会显式访问此模块，但在提供与内置值同名的对象的模块中可能很有用，但其中还需要内置该名称。例如，在一个想要实现*open()* 函数的模块中，它包装了内置的*open()*，这个模块可以直接使用:

```
import builtins

def open(path):
    f = builtins.open(path, 'r')
    return UpperCaser(f)

class UpperCaser:
    '''Wrapper around a file that converts output to upper-case.'''

    def __init__(self, f):
        self._f = f

    def read(self, count=-1):
        return self._f.read(count).upper()

    # ...
```

作为一个实现细节，大多数模块都将名称 __builtins__ 作为其全局变量的一部分提供。__builtins__ 的值通常是这个模块或者这个模块的值__dict__ 属性。由于这是一个实现细节，因此 Python 的替代实现可能不会使用它。

29.4 __main__ --- 顶层脚本环境

'__main__' 是顶层代码执行的作用域的名称。模块的 __name__ 在通过标准输入、脚本文件或是交互式命令读入的时候会等于 '__main__'。

模块可以通过检查自己的 __name__ 来得知是否运行在 main 作用域中，这使得模块可以在作为脚本或是通过 python -m 运行时条件性地执行一些代码，而在被 import 时不会执行。

```
if __name__ == "__main__":
    # execute only if run as a script
    main()
```

对软件包来说，通过加入 __main__.py 模块可以达到同样的效果，当使用 -m 运行模块时，其中的代码会被执行。

29.5 warnings ——警告信息的控制

源代码： Lib/warnings.py

通常以下情况会引发警告：提醒用户注意程序中的某些情况，而这些情况（通常）还不值得触发异常并终止程序。例如，当程序用到了某个过时的模块时，就可能需要发出一条警告。

Python 程序员可调用本模块中定义的*warn()* 函数来发布警告。（C 语言程序员则用 PyErr_WarnEx()；详见 exceptionhandling ）。

警告信息通常会写入*sys.stderr*，但可以灵活改变，从忽略所有警告到变成异常都可以。警告的处理方式可以依据警告类型 、警告信息的文本和发出警告的源位置而进行变化。同一源位置重复出现的警告通常会被抑制。

控制警告信息有两个阶段：首先，每次引发警告时，决定信息是否要发出；然后，如果要发出信息，就用可由用户设置的钩子进行格式化并打印输出。

警告过滤器 控制着是否发出警告信息，也即一系列的匹配规则和动作。调用*filterwarnings()* 可将规则加入过滤器，调用*resetwarnings()* 则可重置为默认状态。

警告信息的打印输出是通过调用*showwarning()* 完成的，该函数可被重写；默认的实现代码是调用*formatwarning()* 进行格式化，自己编写的代码也可以调用此格式化函数。

也参考：

利用*logging.captureWarnings()* 可以采用标准的日志部件处理所有警告。

29.5.1 警告类别

警告的类别由一些内置的异常表示。这种分类有助于对警告信息进行分组过滤。

虽然在技术上警告类别属于内置异常，但也只是在此记录一下而已，因为在概念上他们属于警告机制的一部分。

通过对某个标准的警告类别进行派生，用户代码可以定义其他的警告类别。警告类别必须是*Warning* 类的子类。

目前已定义了以下警告类别的类：

类	描述
Warning	这是所有警告类别的基类。它是*Exception* 的子类。
UserWarning	The default category for *warn()*.
DeprecationWarning	已废弃特性警告的基类，这些警告是为其他 Python 开发者准备的（默认会忽略，除非在 __main__ 中用代码触发）。
SyntaxWarning	用于警告可疑语法的基类。
RuntimeWarning	用于警告可疑运行时特性的基类
FutureWarning	用于警告已废弃特性的基类，这些警告是为 Python 应用程序的最终用户准备的。
PendingDeprecationWarning	用于警告即将废弃功能的基类（默认忽略）。
ImportWarning	导入模块时触发的警告的基类（默认忽略）。
UnicodeWarning	用于 Unicode 相关警告的基类。
BytesWarning	*bytes* 和*bytearray* 相关警告的基类
ResourceWarning	Base category for warnings related to resource usage (ignored by default).

3.7 版更變: 以前*DeprecationWarning* 和*FutureWarning* 是根据某个功能是否完全删除或改变其行为来区分的。现在是根据受众和默认警告过滤器的处理方式来区分的。

29.5.2 警告过滤器

警告过滤器控制着警告是否被忽略、显示或转为错误（触发异常）。

从概念上讲，警告过滤器维护着一个经过排序的过滤器类别列表；任何具体的警告都会依次与列表中的每种过滤器进行匹配，直到找到一个匹配项；过滤器决定了匹配项的处理方式。每个列表项均为 (*action* , *message* , *category* , *module* , *lineno*) 格式的元组，其中：

- *action* 是以下字符串之一：

值	处置
"default"	为发出警告的每个位置（模块 + 行号）打印第一个匹配警告
"error"	将匹配警告转换为异常
"ignore"	从不打印匹配的警告
"always"	总是打印匹配的警告
"module"	为发出警告的每个模块打印第一次匹配警告（无论行号如何）
"once"	无论位置如何，仅打印第一次出现的匹配警告

- *message* 是包含正则表达式的字符串，警告信息的开头必须与之匹配。该表达式编译时不区分大小写。
- *category* 是警告类别的类（*Warning* 的子类），警告类别必须是其子类，才能匹配。
- *module* 是个字符串，包含了模块名称必须匹配的正则表达式。该表达式编译时大小写敏感。
- *lineno* 是个整数，发生警告的行号必须与之匹配，或与所有行号匹配。

由于*Warning*类是由内置类*Exception*派生出来的，要把某个警告变成错误，只要触发 “category(message)“ 即可。

如果警告不匹配所有已注册的过滤器，那就会应用 “default” 动作（正如其名）。

警告过滤器的介绍

警告过滤器由传给 Python 解释器的命令行 -W 选项和 PYTHONWARNINGS 环境变量初始化。解释器在 sys.warningoptions 中保存了所有给出的参数，但不作解释；*warnings*模块在第一次导入时会解析这些参数（无效的选项被忽略，并会先向*sys.stderr*打印一条信息）。

每个警告过滤器的设定格式为冒号分隔的字段序列：

```
action:message:category:module:line
```

这些字段的含义在警告过滤器 中描述。当一行中列出多个过滤器时（如 PYTHONWARNINGS），过滤器间用逗号隔开，后面的优先于前面的（因为是从左到右应用的，最近应用的过滤器优先于前面的）。

常用的警告过滤器适用于所有的警告、特定类别的警告、由特定模块和包引发的警告。下面是一些例子：

```
default                       # Show all warnings (even those ignored by default)
ignore                        # Ignore all warnings
error                         # Convert all warnings to errors
error::ResourceWarning        # Treat ResourceWarning messages as errors
default::DeprecationWarning   # Show DeprecationWarning messages
ignore,default:::mymodule     # Only report warnings triggered by "mymodule"
error:::mymodule[.*]          # Convert warnings to errors in "mymodule"
                              # and any subpackages of "mymodule"
```

默认警告过滤器

Python 默认安装了几个警告过滤器，可以通过 -W 命令行参数、PYTHONWARNINGS 环境变量及调用*filterwarnings()*进行覆盖。

在常规发布的版本中，默认的警告过滤器包括（按优先顺序排列）：

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

在调试版本中，默认警告过滤器的列表是空的。

3.2 版更變: 除了*PendingDeprecationWarning*之外，*DeprecationWarning*现在默认会被忽略。

3.7 版更變: *DeprecationWarning*在被 __main__ 中的代码直接触发时，默认会再次显示。

3.7 版更變: 如果指定两次 -b，则*BytesWarning*不再出现在默认的过滤器列表中，而是通过 sys.warningoptions 进行配置。

重写默认的过滤器

Python 应用程序的开发人员可能希望在默认情况下向用户隐藏 所有 Python 级别的警告，而只在运行测试或其他调试时显示这些警告。用于向解释器传递过滤器配置的 `sys.warningoptions` 属性可以作为一个标记，表示是否应该禁用警告：

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

建议 Python 代码测试的开发者使用如下代码，以确保被测代码默认显示 所有警告：

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # Change the filter in this process
    os.environ["PYTHONWARNINGS"] = "default" # Also affect subprocesses
```

最后，建议在 `__main__` 以外的命名空间运行用户代码的交互式开发者，请确保*DeprecationWarning* 在默认情况下是可见的，可采用如下代码（这里 `user_ns` 是用于执行交互式输入代码的模块）：

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,
                                   module=user_ns.get("__name__"))
```

29.5.3 暂时禁止警告

如果明知正在使用会引起警告的代码，比如某个废弃函数，但不想看到警告（即便警告已经通过命令行作了显式配置），那么可以使用*catch_warnings* 上下文管理器来抑制警告。

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()
```

在上下文管理器中，所有的警告将被简单地忽略。这样就能使用已知的过时代码而又不必看到警告，同时也不会限制警告其他可能不知过时的代码。注意：只能保证在单线程应用程序中生效。如果两个以上的线程同时使用*catch_warnings* 上下文管理器，行为不可预知。

29.5.4 测试警告

要测试由代码引发的警告，请采用*catch_warnings* 上下文管理器。有了它，就可以临时改变警告过滤器以方便测试。例如，以下代码可捕获所有的警告以便查看：

```
import warnings

def fxn():
    warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
    # Cause all warnings to always be triggered.
    warnings.simplefilter("always")
    # Trigger a warning.
    fxn()
    # Verify some things
    assert len(w) == 1
    assert issubclass(w[-1].category, DeprecationWarning)
    assert "deprecated" in str(w[-1].message)
```

也可以用 `error` 取代 `always`，让所有的警告都成为异常。需要注意的是，如果某条警告已经因为 `once` / `default` 规则而被引发，那么无论设置什么过滤器，该条警告都不会再出现，除非该警告有关的注册数据被清除。

一旦上下文管理器退出，警告过滤器将恢复到刚进此上下文时的状态。这样在多次测试时可防止意外改变警告过滤器，从而导致不确定的测试结果。模块中的*showwarning()* 函数也被恢复到初始值。注意：这只能在单线程应用程序中得到保证。如果两个以上的线程同时使用*catch_warnings* 上下文管理器，行为未定义。

当测试多项操作会引发同类警告时，重点是要确保每次操作都会触发新的警告（比如，将警告设置为异常并检查操作是否触发异常，检查每次操作后警告列表的长度是否有增加，否则就在每次新操作前将以前的警告列表项删除）。

29.5.5 为新版本的依赖关系更新代码

在默认情况下，主要针对 Python 开发者（而不是 Python 应用程序的最终用户）的警告类别，会被忽略。

值得注意的是，这个“默认忽略”的列表包含*DeprecationWarning*（适用于每个模块，除了 `__main__`），这意味着开发人员应该确保在测试代码时应将通常忽略的警告显示出来，以便未来破坏性 API 变化时及时收到通知（无论是在标准库还是第三方包）。

理想情况下，代码会有一个合适的测试套件，在运行测试时会隐含地启用所有警告（由*unittest* 模块提供的测试运行程序就是如此）。

在不太理想的情况下，可以通过向 Python 解释器传入 `-Wd` (这是 `-W default` 的简写) 或设置环境变量 `PYTHONWARNINGS=default` 来检查应用程序是否用到了已弃用的接口。这样可以启用对所有警告的默认处理操作，包括那些默认忽略的警告。要改变遇到警告后执行的动作，可以改变传给 `-W` 的参数 (例如 `-W error`)。请参阅 `-W` 旗标来了解更多的细节。

29.5.6 可用的函数

warnings.**warn**(*message*, *category=None*, *stacklevel=1*, *source=None*)
引发警告、忽略或者触发异常。如果给出 *category* 参数，则必须是警告类别类；默认为*UserWarning*。或者 *message* 可为*Warning* 的实例，这时 *category* 将被忽略，转而采用 `message.__class__`。在这种情况下，错误信息文本将是 `str(message)`。如果某条警告被警告过滤器 改成了错误，本函数将触发一条异常。参数 *stacklevel* 可供 Python 包装函数使用，比如:

```
def deprecation(message):
    warnings.warn(message, DeprecationWarning, stacklevel=2)
```

这会让警告指向 `deprecation()` 的调用者，而不是 `deprecation()` 本身的来源（因为后者会破坏引发警告的目的）。

source 是发出*ResourceWarning* 的被销毁对象。

3.6 版更變: 加入 *source* 参数。

warnings.**warn_explicit**(*message*, *category*, *filename*, *lineno*, *module=None*, *registry=None*, *module_globals=None*, *source=None*)
这是*warn()* 函数的底层接口，显式传入消息、类别、文件名和行号，以及可选的模块名和注册表（应为模块的 `__warningregistry__` 字典）。模块名称默认为去除了 `.py` 的文件名；如果未传递注册表，警告就不会被抑制。*message* 必须是个字符串，*category* 是*Warning* 的子类；或者 *message* 可为*Warning* 的实例，且 *category* 将被忽略。

module_globals 应为发出警告的代码所用的全局命名空间。（该参数用于从 zip 文件或其他非文件系统导入模块时显式源码）。

source 是发出*ResourceWarning* 的被销毁对象。

3.6 版更變: 加入 *source* 参数。

warnings.**showwarning**(*message*, *category*, *filename*, *lineno*, *file=None*, *line=None*)
将警告信息写入文件。默认的实现代码是调用 "formatwarning(message, category, filename, lineno, line)" 并将结果字符串写入 *file*，默认文件为*sys.stderr*。通过将任何可调用对象赋给 `warnings.showwarning` 可替换掉该函数。*line* 是要包含在警告信息中的一行源代码；如果未提供 *line*，*showwarning()* 将尝试读取由 *filename* 和 *lineno* 指定的行。

warnings.**formatwarning**(*message*, *category*, *filename*, *lineno*, *line=None*)
以标准方式格式化一条警告信息。将返回一个字符串，可能包含内嵌的换行符，并以换行符结束。如果未提供 *line*，*formatwarning()* 将尝试读取由 *filename* 和 *lineno* 指定的行。

warnings.**filterwarnings**(*action*, *message=''*, *category=Warning*, *module=''*, *lineno=0*, *append=False*)
在警告过滤器种类 列表中插入一条数据项。默认情况下，该数据项将被插到前面；如果 *append* 为 True，则会插到后面。这里会检查参数的类型，编译 *message* 和 *module* 正则表达式，并将他们作为一个元组插入警告过滤器的列表中。如果两者都与某种警告匹配，那么靠近列表前面的数据项就会覆盖后面的项。省略的参数默认匹配任意值。

warnings.**simplefilter**(*action*, *category=Warning*, *lineno=0*, *append=False*)
在警告过滤器种类 列表中插入一条简单数据项。函数参数的含义与*filterwarnings()* 相同，但不需要正则表达式，因为插入的过滤器总是匹配任何模块中的任何信息，只要类别和行号匹配即可。

warnings.**resetwarnings**()
重置警告过滤器。这将丢弃之前对*filterwarnings()* 的所有调用，包括 `-W` 命令行选项和对*simplefilter()* 的调用效果。

29.5.7 可用的上下文管理器

class warnings.**catch_warnings**(*, *record=False*, *module=None*)

该上下文管理器会复制警告过滤器和*showwarning()* 函数，并在退出时恢复。如果 *record* 参数是*False* (默认)，则在进入时会返回*None*。如果 *record* 为*True*，则返回一个列表，列表由自定义*showwarning()* 函数所用对象逐步填充（该函数还会抑制 sys.stdout 的输出）。列表中每个对象的属性与*showwarning()* 的参数名称相同。

module 参数代表一个模块，当导入*warnings* 时，将被用于代替返回的模块，其过滤器将被保护。该参数主要是为了测试*warnings* 模块自身。

備🄵: *catch_warnings* 管理器的工作方式，是替换并随后恢复模块的*showwarning()* 函数和内部的过滤器种类列表。这意味着上下文管理器将会修改全局状态，因此不是线程安全的。

29.6 dataclasses --- 数据类

源码: Lib/dataclasses.py

这个模块提供了一个装饰器和一些函数，用于自动添加生成的*special method*，例如 __init__() 和 __repr__() 到用户定义的类。它最初描述于 **PEP 557** 。

在这些生成的方法中使用的成员变量是使用 **PEP 526** 类型标注来定义的。例如以下代码:

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

将在添加的内容中包括如下所示的 __init__():

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0):
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

请注意，此方法会自动添加到类中：它不会在上面显示的 InventoryItem 定义中直接指定。

3.7 版新加入.

29.6.1 模块级装饰器、类和函数

@dataclasses.**dataclass**(*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*)

这个函数是*decorator*，用于将生成的*special method* 添加到类中，如下所述。

dataclass() 装饰器会检查类以查找 `field`。`field` 被定义为具有类型标注 的类变量。除了下面描述的两个例外，在*dataclass()* 中没有什么东西会去检查在变量标注中所指定的类型。

所有生成的方法中的字段顺序是它们在类定义中出现的顺序。

dataclass() 装饰器将向类中添加各种 “dunder” 方法，如下所述。如果所添加的方法已存在于类中，则行为将取决于下面所列出的参数。装饰器会返回调用它的类本身；不会创建新的类。

如果*dataclass()* 仅用作没有参数的简单装饰器，它就像它具有此签名中记录的默认值一样。也就是说，这三种*dataclass()* 用法是等价的:

```
@dataclass
class C:
    ...

@dataclass()
class C:
    ...

@dataclass(init=True, repr=True, eq=True, order=False, unsafe_hash=False,␣
↪frozen=False)
class C:
   ...
```

dataclass() 的参数有:

- `init`: 如果为真值（默认），将生成一个 `__init__()` 方法。

  如果类已定义 `__init__()`，则忽略此参数。

- `repr`： 如果为真值（默认），将生成一个 `__repr__()` 方法。生成的 repr 字符串将具有类名以及每个字段的名称和 repr，按照它们在类中定义的顺序。不包括标记为从 repr 中排除的字段。例如: `InventoryItem(name='widget', unit_price=3.0, quantity_on_hand=10)`。

  如果类已定义 `__repr__()`，则忽略此参数。

- `eq`： 如果为 true（默认值），将生成 `__eq__()` 方法。此方法将类作为其字段的元组按顺序比较。比较中的两个实例必须是相同的类型。

  如果类已定义 `__eq__()`，则忽略此参数。

- `order`： 如果为真值（默认为 `False`），则 `__lt__()`、`__le__()`、`__gt__()` 和 `__ge__()` 方法将生成。这将类作为其字段的元组按顺序比较。比较中的两个实例必须是相同的类型。如果 `order` 为真值并且 `eq` 为假值，则引发*ValueError*。

  如果类已经定义了 `__lt__()`、`__le__()`、`__gt__()` 或者 `__ge__()` 中的任意一个，将引发*TypeError*。

- `unsafe_hash`： 如果为 `False`（默认值），则根据 `eq` 和 `frozen` 的设置方式生成 `__hash__()` 方法。

  `__hash__()` 由内置的*hash()* 使用，当对象被添加到散列集合（如字典和集合）时。有一个 `__hash__()` 意味着类的实例是不可变的。可变性是一个复杂的属性，取决于程序员的意图，`__eq__()` 的存在性和行为，以及*dataclass()* 装饰器中 `eq` 和 `frozen` 标志的值。

默认情况下，*dataclass()* 不会隐式添加 __hash__() 方法，除非这样做是安全的。它也不会添加或更改现有的明确定义的 __hash__() 方法。设置类属性 __hash__ = None 对 Python 具有特定含义，如 __hash__() 文档中所述。

如果 __hash__() 没有显式定义，或者它被设为 None，则*dataclass()* 可能会添加一个隐式 __hash__() 方法。虽然并不推荐，但你可以用 unsafe_hash=True 来强制*dataclass()* 创建一个 __hash__() 方法。如果你的类在逻辑上不可变但却仍然可被修改那么可能就是这种情况。这是一个特殊用例并且应当被仔细地考虑。

以下是隐式创建 __hash__() 方法的规则。请注意，你不能在数据类中都使用显式的 __hash__() 方法并设置 unsafe_hash=True；这将导致*TypeError*。

如果 eq 和 frozen 都是 true，默认情况下*dataclass()* 将为你生成一个 __hash__() 方法。如果 eq 为 true 且 frozen 为 false，则 __hash__() 将被设置为 None，标记它不可用（因为它是可变的）。如果 eq 为 false，则 __hash__() 将保持不变，这意味着将使用超类的 __hash__() 方法（如果超类是*object*，这意味着它将回到基于 id 的 hash）。

- frozen: 如为真值 (默认值为 False)，则对字段赋值将会产生异常。这模拟了只读的冻结实例。如果在类中定义了 __setattr__() 或 __delattr__() 则将会引发*TypeError*。参见下文的讨论。

fields 可以选择使用普通的 Python 语法指定默认值:

```
@dataclass
class C:
    a: int       # 'a' has no default value
    b: int = 0   # assign a default value for 'b'
```

在这个例子中，a 和 b 都将包含在添加的 __init__() 方法中，它们将被定义为:

```
def __init__(self, a: int, b: int = 0):
```

如果具有默认值的字段之后存在没有默认值的字段，将会引发*TypeError*。无论此情况是发生在单个类中还是作为类继承的结果，都是如此。

dataclasses.**field**(*, *default=MISSING*, *default_factory=MISSING*, *repr=True*, *hash=None*, *init=True*, *compare=True*, *metadata=None*)

对于常见和简单的用例，不需要其他功能。但是，有些数据类功能需要额外的每字段信息。为了满足这种对附加信息的需求，你可以通过调用提供的*field()* 函数来替换默认字段值。例如:

```
@dataclass
class C:
    mylist: list[int] = field(default_factory=list)

c = C()
c.mylist += [1, 2, 3]
```

如上所示，MISSING 值是一个 sentinel 对象，用于检测是否提供了 default 和 default_factory 参数。使用此 sentinel 是因为 None 是 default 的有效值。没有代码应该直接使用 MISSING 值。

field() 参数有:

- default：如果提供，这将是该字段的默认值。这是必需的，因为*field()* 调用本身会替换一般的默认值。
- default_factory：如果提供，它必须是一个零参数可调用对象，当该字段需要一个默认值时，它将被调用。除了其他目的之外，这可以用于指定具有可变默认值的字段，如下所述。同时指定 default 和 default_factory 将产生错误。
- init：如果为 true（默认值），则该字段作为参数包含在生成的 __init__() 方法中。

- repr： 如果为 true（默认值），则该字段包含在生成的 __repr__() 方法返回的字符串中。
- compare： 如果为 true（默认值），则该字段包含在生成的相等性和比较方法中（__eq__()，__gt__() 等等）。
- hash：这可以是布尔值或 None 。如果为 true，则此字段包含在生成的 __hash__() 方法中。如果为 None （默认值），请使用 compare 的值，这通常是预期的行为。如果字段用于比较，则应在 hash 中考虑该字段。不鼓励将此值设置为 None 以外的任何值。

  设置 hash=False 但 compare=True 的一个可能原因是，如果一个计算 hash 的代价很高的字段是检验等价性需要的，但还有其他字段可以计算类型的 hash 。即使从 hash 中排除某个字段，它仍将用于比较。
- metadata ： 这可以是映射或 None 。None 被视为一个空的字典。这个值包含在 MappingProxyType() 中，使其成为只读，并暴露在*Field* 对象上。数据类根本不使用它，它是作为第三方扩展机制提供的。多个第三方可以各自拥有自己的键值，以用作元数据中的命名空间。

如果通过调用*field()* 指定字段的默认值，则该字段的类属性将替换为指定的 default 值。如果没有提供 default ，那么将删除类属性。目的是在*dataclass()* 装饰器运行之后，类属性将包含字段的默认值，就像指定了默认值一样。例如，之后:

```
@dataclass
class C:
    x: int
    y: int = field(repr=False)
    z: int = field(repr=False, default=10)
    t: int = 20
```

类属性 C.z 将是 10 ，类属性 C.t 将是 20，类属性 C.x 和 C.y 将不设置。

class dataclasses.**Field**

Field 对象描述每个定义的字段。这些对象在内部创建，并由*fields()* 模块级方法返回（见下文）。用户永远不应该直接实例化*Field* 对象。其有文档的属性是:

- name： 字段的名字。
- type： 字段的类型。
- default 、default_factory 、init 、repr 、hash 、compare 以及 metadata 与具有和*field()* 声明中相同的意义和值。

可能存在其他属性，但它们是私有的，不能被审查或依赖。

dataclasses.**fields**(*class_or_instance*)

返回*Field* 对象的元组，用于定义此数据类的字段。接受数据类或数据类的实例。如果没有传递一个数据类或实例将引发*TypeError* 。不返回 ClassVar 或 InitVar 的伪字段。

dataclasses.**asdict**(*obj*, *, *dict_factory=dict*)

Converts the dataclass obj to a dict (by using the factory function dict_factory). Each dataclass is converted to a dict of its fields, as name: value pairs. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with *copy.deepcopy()*.

Example of using *asdict()* on nested dataclasses:

```
@dataclass
class Point:
     x: int
     y: int

@dataclass
```

(下页继续)

(繼續上一頁)

```
class C:
     mylist: list[Point]

p = Point(10, 20)
assert asdict(p) == {'x': 10, 'y': 20}

c = C([Point(0, 0), Point(10, 4)])
assert asdict(c) == {'mylist': [{'x': 0, 'y': 0}, {'x': 10, 'y': 4}]}
```

To create a shallow copy, the following workaround may be used:

```
dict((field.name, getattr(obj, field.name)) for field in fields(obj))
```

asdict() raises *TypeError* if `obj` is not a dataclass instance.

dataclasses.**astuple**(*obj*, *, *tuple_factory=tuple*)

Converts the dataclass `obj` to a tuple (by using the factory function `tuple_factory`). Each dataclass is converted to a tuple of its field values. dataclasses, dicts, lists, and tuples are recursed into. Other objects are copied with *copy.deepcopy()*.

继续前一个例子:

```
assert astuple(p) == (10, 20)
assert astuple(c) == ([(0, 0), (10, 4)],)
```

To create a shallow copy, the following workaround may be used:

```
tuple(getattr(obj, field.name) for field in dataclasses.fields(obj))
```

astuple() raises *TypeError* if `obj` is not a dataclass instance.

dataclasses.**make_dataclass**(*cls_name*, *fields*, *, *bases=()*, *namespace=None*, *init=True*, *repr=True*, *eq=True*, *order=False*, *unsafe_hash=False*, *frozen=False*)

创建一个名为 `cls_name` 的新数据类，字段为 `fields` 中定义的字段，基类为 `bases` 中给出的基类，并使用 `namespace` 中给出的命名空间进行初始化。`fields` 是一个可迭代的元素，每个元素都是 `name`、`(name, type)` 或 `(name, type, Field)`。如果只提供 "name"，`type` 为 `typing.Any`。`init`、`repr`、`eq`、`order`、`unsafe_hash` 和 `frozen` 的值与它们在*dataclass()* 中的含义相同。

此函数不是严格要求的，因为用于任何创建带有 `__annotations__` 的新类的 Python 机制都可以应用*dataclass()* 函数将该类转换为数据类。提供此功能是为了方便。例如:

```
C = make_dataclass('C',
                   [('x', int),
                     'y',
                    ('z', int, field(default=5))],
                   namespace={'add_one': lambda self: self.x + 1})
```

等价于

```
@dataclass
class C:
    x: int
    y: 'typing.Any'
    z: int = 5

    def add_one(self):
        return self.x + 1
```

dataclasses.**replace**(*obj*, */*, ***changes*)

Creates a new object of the same type as obj, replacing fields with values from changes. If obj is not a Data Class, raises *TypeError*. If values in changes do not specify fields, raises *TypeError*.

新返回的对象通过调用数据类的 __init__() 方法创建。这确保了如果存在 __post_init__()，其也被调用。

如果存在没有默认值的仅初始化变量，必须在调用*replace()* 时指定，以便它们可以传递给 __init__() 和 __post_init__()。

changes 包含任何定义为 init=False 的字段是错误的。在这种情况下会引发*ValueError*。

提前提醒 init=False 字段在调用*replace()* 时的工作方式。如果它们完全被初始化的话，它们不是从源对象复制的，而是在 __post_init__() 中初始化。估计 init=False 字段很少能被正确地使用。如果使用它们，那么使用备用类构造函数或者可能是处理实例复制的自定义 replace()（或类似命名的）方法可能是明智的。

dataclasses.**is_dataclass**(*obj*)

如果其形参为 dataclass 或其实例则返回 True，否则返回 False。

如果你需要知道一个类是否是一个数据类的实例（而不是一个数据类本身），那么再添加一个 not isinstance(obj, type) 检查:

```
def is_dataclass_instance(obj):
    return is_dataclass(obj) and not isinstance(obj, type)
```

29.6.2 初始化后处理

生成的 __init__() 代码将调用一个名为 __post_init__() 的方法，如果在类上已经定义了 __post_init__()。它通常被称为 self.__post_init__()。但是，如果定义了任何 InitVar 字段，它们也将按照它们在类中定义的顺序传递给 __post_init__()。如果没有 __ init__() 方法生成，那么 __post_init__() 将不会被自动调用。

在其他用途中，这允许初始化依赖于一个或多个其他字段的字段值。例如:

```
@dataclass
class C:
    a: float
    b: float
    c: float = field(init=False)

    def __post_init__(self):
        self.c = self.a + self.b
```

由*dataclass()* 所生成的 __init__() 方法不会调用基类的 __init__() 方法。如果基类有需要被调用的 __init__() 方法，通常是在 __post_init__() 方法中调用此方法:

```
@dataclass
class Rectangle:
    height: float
    width: float

@dataclass
class Square(Rectangle):
    side: float

    def __post_init__(self):
        super().__init__(self.side, self.side)
```

但是请注意，一般来说 dataclass 生成的 `__init__()` 方法不需要被调用，因为派生的 dataclass 将负责初始化任何自身为 dataclass 的基类的所有字段。

有关将参数传递给 `__post_init__()` 的方法，请参阅下面有关仅初始化变量的段落。另请参阅关于*replace()* 处理 `init=False` 字段的警告。

29.6.3 类变量

两个地方*dataclass()* 实际检查字段类型的之一是确定字段是否是如 **PEP 526** 所定义的类变量。它通过检查字段的类型是否为 `typing.ClassVar` 来完成此操作。如果一个字段是一个 `ClassVar`，它将被排除在考虑范围之外，并被数据类机制忽略。这样的 `ClassVar` 伪字段不会由模块级的*fields()* 函数返回。

29.6.4 仅初始化变量

另一个*dataclass()* 检查类型注解地方是为了确定一个字段是否是一个仅初始化变量。它通过查看字段的类型是否为 `dataclasses.InitVar` 类型来实现。如果一个字段是一个 `InitVar`，它被认为是一个称为仅初始化字段的伪字段。因为它不是一个真正的字段，所以它不会被模块级的*fields()* 函数返回。仅初始化字段作为参数添加到生成的 `__init__()` 方法中，并传递给可选的 `__post_init__()` 方法。数据类不会使用它们。

例如，假设一个字段将从数据库初始化，如果在创建类时未提供其值:

```
@dataclass
class C:
    i: int
    j: int = None
    database: InitVar[DatabaseType] = None

    def __post_init__(self, database):
        if self.j is None and database is not None:
            self.j = database.lookup('j')

c = C(10, database=my_database)
```

在这种情况下，*fields()* 将返回 `i` 和 `j` 的*Field* 对象，但不包括 `database`。

29.6.5 冻结的实例

无法创建真正不可变的 Python 对象。但是，通过将 `frozen=True` 传递给*dataclass()* 装饰器，你可以模拟不变性。在这种情况下，数据类将向类添加 `__setattr__()` 和 `__delattr__()` 方法。些方法在调用时会引发*FrozenInstanceError*。

使用 `frozen=True` 时会有很小的性能损失：`__ init__()` 不能使用简单的赋值来初始化字段，并必须使用 `object.__setattr__()`。

29.6.6 继承

当数组由*dataclass()* 装饰器创建时，它会查看反向 MRO 中的所有类的基类（即从*object* 开始），并且对于它找到的每个数据类，将该基类中的字段添加到字段的有序映射中。添加完所有基类字段后，它会将自己的字段添加到有序映射中。所有生成的方法都将使用这种组合的，计算的有序字段映射。由于字段是按插入顺序排列的，因此派生类会重载基类。一个例子:

```
@dataclass
class Base:
    x: Any = 15.0
    y: int = 0

@dataclass
class C(Base):
    z: int = 10
    x: int = 15
```

最后的字段列表依次是 x 、 y 、 z 。 x 的最终类型是 int ， 如类 C 中所指定的那样。

为 C 生成的 __init__() 方法看起来像:

```
def __init__(self, x: int = 15, y: int = 0, z: int = 10):
```

29.6.7 默认工厂函数

如果一个*field()* 指定了一个 default_factory ， 当需要该字段的默认值时，将使用零参数调用它。例如，要创建列表的新实例，请使用:

```
mylist: list = field(default_factory=list)
```

如果一个字段被排除在 __init__() 之外（使用 init=False ）并且字段也指定 default_factory ， 则默认的工厂函数将始终从生成的 __init__() 函数调用。发生这种情况是因为没有其他方法可以为字段提供初始值。

29.6.8 可变的默认值

Python 在类属性中存储默认成员变量值。思考这个例子，不使用数据类:

```
class C:
    x = []
    def add(self, element):
        self.x.append(element)

o1 = C()
o2 = C()
o1.add(1)
o2.add(2)
assert o1.x == [1, 2]
assert o1.x is o2.x
```

请注意，类 C 的两个实例共享相同的类变量 x ， 如预期的那样。

使用数据类，如果此代码有效:

```
@dataclass
class D:
    x: List = []
    def add(self, element):
        self.x += element
```

它生成的代码类似于:

```
class D:
    x = []
    def __init__(self, x=x):
        self.x = x
    def add(self, element):
        self.x += element

assert D().x is D().x
```

这与使用类 C 的原始示例具有相同的问题。也就是说，在创建类实例时没有为 x 指定值的类 D 的两个实例将共享相同的 x 副本。由于数据类只使用普通的 Python 类创建，因此它们也会共享此行为。数据类没有通用的方法来检测这种情况。相反，如果数据类检测到类型为 list 、dict 或 set 的默认参数，则会引发*TypeError* 。这是一个部分解决方案，但它可以防止许多常见错误。

使用默认工厂函数是一种创建可变类型新实例的方法，并将其作为字段的默认值:

```
@dataclass
class D:
    x: list = field(default_factory=list)

assert D().x is not D().x
```

29.6.9 异常

exception dataclasses.**FrozenInstanceError**
在使用 frozen=True 定义的数据类上调用隐式定义的 __setattr__() 或 __delattr__() 时引发。这是*AttributeError* 的一个子类。

29.7 contextlib --- 为 with 语句上下文提供的工具

源代码 Lib/contextlib.py

此模块为涉及 with 语句的常见任务提供了实用的工具。更多信息请参见上下文管理器类型 和 context-managers。

29.7.1 工具

提供的函数和类:

class contextlib.**AbstractContextManager**
一个为实现了 object.__enter__() 与 object.__exit__() 的类提供的*abstract base class*。为 object.__enter__() 提供的一个默认实现是返回 self 而 object.__exit__() 是一个默认返回 None 的抽象方法。参见上下文管理器类型 的定义。

3.6 版新加入.

class contextlib.**AbstractAsyncContextManager**
一个为实现了 object.__aenter__() 与 object.__aexit__() 的类提供的*abstract base class*。为 object.__aenter__() 提供的一个默认实现是返回 self 而 object.__aexit__() 是一个默认返回 None 的抽象方法。参见 async-context-managers 的定义。

3.7 版新加入.

@contextlib.**contextmanager**
这个函数是一个*decorator*，它可以定义一个支持 with 语句上下文管理器的工厂函数，而不需要创建一个类或区 __enter__() 与 __exit__() 方法。

尽管许多对象原生支持使用 with 语句，但有些需要被管理的资源并不是上下文管理器，并且没有实现 close() 方法而不能使用 contextlib.closing 。

下面是一个抽象的示例，展示如何确保正确的资源管理:

```
from contextlib import contextmanager

@contextmanager
def managed_resource(*args, **kwds):
    # Code to acquire resource, e.g.:
    resource = acquire_resource(*args, **kwds)
    try:
        yield resource
    finally:
        # Code to release resource, e.g.:
        release_resource(resource)

>>> with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

被装饰的函数在被调用时，必须返回一个*generator* 迭代器。这个迭代器必须只 yield 一个值出来，这个值会被用在 with 语句中，绑定到 as 后面的变量，如果给定了的话。

当生成器发生 yield 时，嵌套在 with 语句中的语句体会被执行。语句体执行完毕离开之后，该生成器将被恢复执行。如果在该语句体中发生了未处理的异常，则该异常会在生成器发生 yield 时重新被引发。因此，你可以使用 try...except...finally 语句来捕获该异常（如果有的话），或确保进行了一些清理。如果仅出于记录日志或执行某些操作（而非完全抑制异常）的目的捕获了异常，生成器必须重新引发该异常。否则生成器的上下文管理器将向 with 语句指示该异常已经被处理，程序将立即在 with 语句之后恢复并继续执行。

contextmanager() 使用*ContextDecorator* 因此它创建的上下文管理器不仅可以用在 with 语句中，还可以用作一个装饰器。当它用作一个装饰器时，每一次函数调用时都会隐式创建一个新的生成器实例（这使得*contextmanager()* 创建的上下文管理器满足了支持多次调用以用作装饰器的需求，而非“一次性”的上下文管理器）。

3.2 版更變: *ContextDecorator* 的使用。

@contextlib.**asynccontextmanager**

与*contextmanager()* 类似，但创建的是 asynchronous context manager 。

这个函数是一个*decorator* ，它可以定义一个支持 async with 语句的异步上下文管理器的工厂函数，而不需要创建一个类或区分 __aenter__() 与 __aexit__() 方法。它必须被作用在一个*asynchronous generator* 函数上

一个简单的示例：

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def get_connection():
    conn = await acquire_db_connection()
    try:
        yield conn
    finally:
        await release_db_connection(conn)

async def get_all_users():
    async with get_connection() as conn:
        return conn.query('SELECT ...')
```

3.7 版新加入.

contextlib.**closing**(*thing*)

返回一个在语句块执行完成时关闭 *things* 的上下文管理器。这基本上等价于：

```
from contextlib import contextmanager

@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

并允许你编写这样的代码：

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

而无需显式地关闭 page 。即使发生错误，在退出 with 语句块时，page.close() 也同样会被调用。

contextlib.**nullcontext**(*enter_result=None*)

返回一个从 __enter__ 返回 *enter_result* 的上下文管理器，除此之外不执行任何操作。它旨在用于可选上下文管理器的一种替代，例如：

```
def myfunction(arg, ignore_exceptions=False):
    if ignore_exceptions:
        # Use suppress to ignore all exceptions.
        cm = contextlib.suppress(Exception)
    else:
        # Do not ignore any exceptions, cm has no effect.
        cm = contextlib.nullcontext()
```

(下页继续)

(繼續上一頁)

```
    with cm:
        # Do something
```

一个使用 *enter_result* 的例子：

```
def process_file(file_or_path):
    if isinstance(file_or_path, str):
        # If string, open file
        cm = open(file_or_path)
    else:
        # Caller is responsible for closing file
        cm = nullcontext(file_or_path)

    with cm as file:
        # Perform processing on the file
```

3.7 版新加入.

contextlib.**suppress**(**exceptions*)

Return a context manager that suppresses any of the specified exceptions if they occur in the body of a `with` statement and then resumes execution with the first statement following the end of the `with` statement.

与完全抑制异常的任何其他机制一样，该上下文管理器应当只用来抑制非常具体的错误，并确保该场景下静默地继续执行程序是通用的正确做法。

例如:

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove('somefile.tmp')

with suppress(FileNotFoundError):
    os.remove('someotherfile.tmp')
```

这段代码等价于：

```
try:
    os.remove('somefile.tmp')
except FileNotFoundError:
    pass

try:
    os.remove('someotherfile.tmp')
except FileNotFoundError:
    pass
```

该上下文管理器是*reentrant* 。

3.4 版新加入.

contextlib.**redirect_stdout**(*new_target*)

用于将*sys.stdout* 临时重定向到一个文件或类文件对象的上下文管理器。

该工具给已有的将输出硬编码写到 stdout 的函数或类提供了额外的灵活性。

For example, the output of *help()* normally is sent to *sys.stdout*. You can capture that output in a string by redirecting the output to an *io.StringIO* object. The replacement stream is returned from the `__enter__` method and so is available as the target of the `with` statement:

```
with redirect_stdout(io.StringIO()) as f:
    help(pow)
s = f.getvalue()
```

如果要把*help()* 的输出写到磁盘上的一个文件，重定向该输出到一个常规文件:

```
with open('help.txt', 'w') as f:
    with redirect_stdout(f):
        help(pow)
```

如果要把*help()* 的输出写到 *sys.stderr*：

```
with redirect_stdout(sys.stderr):
    help(pow)
```

需要注意的点在于，*sys.stdout* 的全局副作用意味着此上下文管理器不适合在库代码和大多数多线程应用程序中使用。它对子进程的输出没有影响。不过对于许多工具脚本而言，它仍然是一个有用的方法。

该上下文管理器是*reentrant* 。

3.4 版新加入.

contextlib.**redirect_stderr**(*new_target*)

与*redirect_stdout()* 类似，不过是将*sys.stderr* 重定向到一个文件或类文件对象。

该上下文管理器是*reentrant* 。

3.5 版新加入.

class contextlib.**ContextDecorator**

一个使上下文管理器能用作装饰器的基类。

与往常一样，继承自 ContextDecorator 的上下文管理器必须实现 __enter__ 与 __exit__ 。即使用作装饰器，__exit__ 依旧会保持可能的异常处理。

ContextDecorator 被用在*contextmanager()* 中，因此你自然获得了这项功能。

ContextDecorator 的示例:

```
from contextlib import ContextDecorator

class mycontext(ContextDecorator):
    def __enter__(self):
        print('Starting')
        return self

    def __exit__(self, *exc):
        print('Finishing')
        return False

>>> @mycontext()
... def function():
...     print('The bit in the middle')
...
>>> function()
Starting
The bit in the middle
Finishing
```

(下页继续)

(繼續上一頁)

```
>>> with mycontext():
...     print('The bit in the middle')
...
Starting
The bit in the middle
Finishing
```

这个改动只是针对如下形式的一个语法糖：

```
def f():
    with cm():
        # Do stuff
```

ContextDecorator 使得你可以这样改写：

```
@cm()
def f():
    # Do stuff
```

这能清楚地表明，cm 作用于整个函数，而不仅仅是函数的一部分（同时也能保持不错的缩进层级）。

现有的上下文管理器即使已经有基类，也可以使用 ContextDecorator 作为混合类进行扩展：

```
from contextlib import ContextDecorator

class mycontext(ContextBaseClass, ContextDecorator):
    def __enter__(self):
        return self

    def __exit__(self, *exc):
        return False
```

備𝔽： 由于被装饰的函数必须能够被多次调用，因此对应的上下文管理器必须支持在多个 with 语句中使用。如果不是这样，则应当使用原来的具有显式 with 语句的形式使用该上下文管理器。

3.2 版新加入.

class contextlib.**ExitStack**

该上下文管理器的设计目标是使得在编码中组合其他上下文管理器和清理函数更加容易，尤其是那些可选的或由输入数据驱动的上下文管理器。

例如，通过一个如下的 with 语句可以很容易处理一组文件：

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # All opened files will automatically be closed at the end of
    # the with statement, even if attempts to open files later
    # in the list raise an exception
```

The __enter__() method returns the *ExitStack* instance, and performs no additional operations.

每个实例维护一个注册了一组回调的栈，这些回调在实例关闭时以相反的顺序被调用（显式或隐式地在 with 语句的末尾）。请注意，当一个栈实例被垃圾回收时，这些回调将 不会被隐式调用。

通过使用这个基于栈的模型，那些通过 __init__ 方法获取资源的上下文管理器（如文件对象）能够被正确处理。

由于注册的回调函数是按照与注册相反的顺序调用的，因此最终的行为就像多个嵌套的 `with` 语句用在这些注册的回调函数上。这个行为甚至扩展到了异常处理：如果内部的回调函数抑制或替换了异常，则外部回调收到的参数是基于该更新后的状态得到的。

这是一个相对底层的 API，它负责正确处理栈里回调退出时依次展开的细节。它为相对高层的上下文管理器提供了一个合适的基础，使得它能根据应用程序的需求使用特定方式操作栈。

3.3 版新加入.

enter_context(*cm*)
Enters a new context manager and adds its `__exit__()` method to the callback stack. The return value is the result of the context manager's own `__enter__()` method.

These context managers may suppress exceptions just as they normally would if used directly as part of a `with` statement.

push(*exit*)
Adds a context manager's `__exit__()` method to the callback stack.

As `__enter__` is *not* invoked, this method can be used to cover part of an `__enter__()` implementation with a context manager's own `__exit__()` method.

If passed an object that is not a context manager, this method assumes it is a callback with the same signature as a context manager's `__exit__()` method and adds it directly to the callback stack.

By returning true values, these callbacks can suppress exceptions the same way context manager `__exit__()` methods can.

The passed in object is returned from the function, allowing this method to be used as a function decorator.

callback(*callback*, */*, **args*, ***kwds*)
Accepts an arbitrary callback function and arguments and adds it to the callback stack.

Unlike the other methods, callbacks added this way cannot suppress exceptions (as they are never passed the exception details).

The passed in callback is returned from the function, allowing this method to be used as a function decorator.

pop_all()
Transfers the callback stack to a fresh *ExitStack* instance and returns it. No callbacks are invoked by this operation - instead, they will now be invoked when the new stack is closed (either explicitly or implicitly at the end of a `with` statement).

For example, a group of files can be opened as an "all or nothing" operation as follows:

```
with ExitStack() as stack:
    files = [stack.enter_context(open(fname)) for fname in filenames]
    # Hold onto the close method, but don't call it yet.
    close_files = stack.pop_all().close
    # If opening any file fails, all previously opened files will be
    # closed automatically. If all files are opened successfully,
    # they will remain open even after the with statement ends.
    # close_files() can then be invoked explicitly to close them all.
```

close()
Immediately unwinds the callback stack, invoking callbacks in the reverse order of registration. For any context managers and exit callbacks registered, the arguments passed in will indicate that no exception occurred.

class contextlib.**AsyncExitStack**
An asynchronous context manager, similar to *ExitStack*, that supports combining both synchronous and asynchronous context managers, as well as having coroutines for cleanup logic.

The `close()` method is not implemented, *aclose()* must be used instead.

coroutine enter_async_context(*cm*)
Similar to `enter_context()` but expects an asynchronous context manager.

push_async_exit(*exit*)
Similar to `push()` but expects either an asynchronous context manager or a coroutine function.

push_async_callback(*callback*, */*, **args*, ***kwds*)
Similar to `callback()` but expects a coroutine function.

coroutine aclose()
Similar to `close()` but properly handles awaitables.

Continuing the example for *asynccontextmanager()*:

```
async with AsyncExitStack() as stack:
    connections = [await stack.enter_async_context(get_connection())
        for i in range(5)]
    # All opened connections will automatically be released at the end of
    # the async with statement, even if attempts to open a connection
    # later in the list raise an exception.
```

3.7 版新加入.

29.7.2 例子和配方

This section describes some examples and recipes for making effective use of the tools provided by *contextlib*.

Supporting a variable number of context managers

The primary use case for *ExitStack* is the one given in the class documentation: supporting a variable number of context managers and other cleanup operations in a single `with` statement. The variability may come from the number of context managers needed being driven by user input (such as opening a user specified collection of files), or from some of the context managers being optional:

```
with ExitStack() as stack:
    for resource in resources:
        stack.enter_context(resource)
    if need_special_resource():
        special = acquire_special_resource()
        stack.callback(release_special_resource, special)
    # Perform operations that use the acquired resources
```

As shown, *ExitStack* also makes it quite easy to use `with` statements to manage arbitrary resources that don't natively support the context management protocol.

Catching exceptions from __enter__ methods

It is occasionally desirable to catch exceptions from an `__enter__` method implementation, *without* inadvertently catching exceptions from the `with` statement body or the context manager's `__exit__` method. By using *ExitStack* the steps in the context management protocol can be separated slightly in order to allow this:

```
stack = ExitStack()
try:
    x = stack.enter_context(cm)
except Exception:
```

(下页继续)

(繼續上一頁)

```
    # handle __enter__ exception
else:
    with stack:
        # Handle normal case
```

Actually needing to do this is likely to indicate that the underlying API should be providing a direct resource management interface for use with `try/except/finally` statements, but not all APIs are well designed in that regard. When a context manager is the only resource management API provided, then *ExitStack* can make it easier to handle various situations that can't be handled directly in a `with` statement.

Cleaning up in an `__enter__` implementation

As noted in the documentation of *ExitStack.push()*, this method can be useful in cleaning up an already allocated resource if later steps in the `__enter__()` implementation fail.

Here's an example of doing this for a context manager that accepts resource acquisition and release functions, along with an optional validation function, and maps them to the context management protocol:

```
from contextlib import contextmanager, AbstractContextManager, ExitStack

class ResourceManager(AbstractContextManager):

    def __init__(self, acquire_resource, release_resource, check_resource_ok=None):
        self.acquire_resource = acquire_resource
        self.release_resource = release_resource
        if check_resource_ok is None:
            def check_resource_ok(resource):
                return True
        self.check_resource_ok = check_resource_ok

    @contextmanager
    def _cleanup_on_error(self):
        with ExitStack() as stack:
            stack.push(self)
            yield
            # The validation check passed and didn't raise an exception
            # Accordingly, we want to keep the resource, and pass it
            # back to our caller
            stack.pop_all()

    def __enter__(self):
        resource = self.acquire_resource()
        with self._cleanup_on_error():
            if not self.check_resource_ok(resource):
                msg = "Failed validation for {!r}"
                raise RuntimeError(msg.format(resource))
        return resource

    def __exit__(self, *exc_details):
        # We don't need to duplicate any of our resource release logic
        self.release_resource()
```

Replacing any use of `try-finally` and flag variables

A pattern you will sometimes see is a `try-finally` statement with a flag variable to indicate whether or not the body of the `finally` clause should be executed. In its simplest form (that can't already be handled just by using an `except` clause instead), it looks something like this:

```
cleanup_needed = True
try:
    result = perform_operation()
    if result:
        cleanup_needed = False
finally:
    if cleanup_needed:
        cleanup_resources()
```

As with any `try` statement based code, this can cause problems for development and review, because the setup code and the cleanup code can end up being separated by arbitrarily long sections of code.

ExitStack makes it possible to instead register a callback for execution at the end of a `with` statement, and then later decide to skip executing that callback:

```
from contextlib import ExitStack

with ExitStack() as stack:
    stack.callback(cleanup_resources)
    result = perform_operation()
    if result:
        stack.pop_all()
```

This allows the intended cleanup up behaviour to be made explicit up front, rather than requiring a separate flag variable.

If a particular application uses this pattern a lot, it can be simplified even further by means of a small helper class:

```
from contextlib import ExitStack

class Callback(ExitStack):
    def __init__(self, callback, /, *args, **kwds):
        super().__init__()
        self.callback(callback, *args, **kwds)

    def cancel(self):
        self.pop_all()

with Callback(cleanup_resources) as cb:
    result = perform_operation()
    if result:
        cb.cancel()
```

If the resource cleanup isn't already neatly bundled into a standalone function, then it is still possible to use the decorator form of *ExitStack.callback()* to declare the resource cleanup in advance:

```
from contextlib import ExitStack

with ExitStack() as stack:
    @stack.callback
    def cleanup_resources():
        ...
    result = perform_operation()
```

(下页继续)

(繼續上一頁)

```
    if result:
        stack.pop_all()
```

Due to the way the decorator protocol works, a callback function declared this way cannot take any parameters. Instead, any resources to be released must be accessed as closure variables.

Using a context manager as a function decorator

ContextDecorator makes it possible to use a context manager in both an ordinary `with` statement and also as a function decorator.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, inheriting from *ContextDecorator* provides both capabilities in a single definition:

```
from contextlib import ContextDecorator
import logging

logging.basicConfig(level=logging.INFO)

class track_entry_and_exit(ContextDecorator):
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        logging.info('Entering: %s', self.name)

    def __exit__(self, exc_type, exc, exc_tb):
        logging.info('Exiting: %s', self.name)
```

Instances of this class can be used as both a context manager:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

And also as a function decorator:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Note that there is one additional limitation when using context managers as function decorators: there's no way to access the return value of `__enter__()`. If that value is needed, then it is still necessary to use an explicit `with` statement.

也參考:

PEP 343 - "with" 语句 Python `with` 语句的规范描述、背景和示例。

29.7.3 Single use, reusable and reentrant context managers

Most context managers are written in a way that means they can only be used effectively in a `with` statement once. These single use context managers must be created afresh each time they're used - attempting to use them a second time will trigger an exception or otherwise not work correctly.

This common limitation means that it is generally advisable to create context managers directly in the header of the `with` statement where they are used (as shown in all of the usage examples above).

Files are an example of effectively single use context managers, since the first `with` statement will close the file, preventing any further IO operations using that file object.

Context managers created using *contextmanager()* are also single use context managers, and will complain about the underlying generator failing to yield if an attempt is made to use them a second time:

```
>>> from contextlib import contextmanager
>>> @contextmanager
... def singleuse():
...     print("Before")
...     yield
...     print("After")
...
>>> cm = singleuse()
>>> with cm:
...     pass
...
Before
After
>>> with cm:
...     pass
...
Traceback (most recent call last):
    ...
RuntimeError: generator didn't yield
```

Reentrant context managers

More sophisticated context managers may be "reentrant". These context managers can not only be used in multiple `with` statements, but may also be used *inside* a `with` statement that is already using the same context manager.

threading.RLock is an example of a reentrant context manager, as are *suppress()* and *redirect_stdout()*. Here's a very simple example of reentrant use:

```
>>> from contextlib import redirect_stdout
>>> from io import StringIO
>>> stream = StringIO()
>>> write_to_stream = redirect_stdout(stream)
>>> with write_to_stream:
...     print("This is written to the stream rather than stdout")
...     with write_to_stream:
...         print("This is also written to the stream")
...
>>> print("This is written directly to stdout")
This is written directly to stdout
>>> print(stream.getvalue())
This is written to the stream rather than stdout
This is also written to the stream
```

Real world examples of reentrancy are more likely to involve multiple functions calling each other and hence be far more complicated than this example.

Note also that being reentrant is *not* the same thing as being thread safe. *redirect_stdout()*, for example, is definitely not thread safe, as it makes a global modification to the system state by binding *sys.stdout* to a different stream.

Reusable context managers

Distinct from both single use and reentrant context managers are "reusable" context managers (or, to be completely explicit, "reusable, but not reentrant" context managers, since reentrant context managers are also reusable). These context managers support being used multiple times, but will fail (or otherwise not work correctly) if the specific context manager instance has already been used in a containing with statement.

threading.Lock is an example of a reusable, but not reentrant, context manager (for a reentrant lock, it is necessary to use *threading.RLock* instead).

Another example of a reusable, but not reentrant, context manager is *ExitStack*, as it invokes *all* currently registered callbacks when leaving any with statement, regardless of where those callbacks were added:

```
>>> from contextlib import ExitStack
>>> stack = ExitStack()
>>> with stack:
...     stack.callback(print, "Callback: from first context")
...     print("Leaving first context")
...
Leaving first context
Callback: from first context
>>> with stack:
...     stack.callback(print, "Callback: from second context")
...     print("Leaving second context")
...
Leaving second context
Callback: from second context
>>> with stack:
...     stack.callback(print, "Callback: from outer context")
...     with stack:
...         stack.callback(print, "Callback: from inner context")
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Callback: from outer context
Leaving outer context
```

As the output from the example shows, reusing a single stack object across multiple with statements works correctly, but attempting to nest them will cause the stack to be cleared at the end of the innermost with statement, which is unlikely to be desirable behaviour.

Using separate *ExitStack* instances instead of reusing a single instance avoids that problem:

```
>>> from contextlib import ExitStack
>>> with ExitStack() as outer_stack:
...     outer_stack.callback(print, "Callback: from outer context")
...     with ExitStack() as inner_stack:
...         inner_stack.callback(print, "Callback: from inner context")
```

(下页继续)

(繼續上一頁)

```
...         print("Leaving inner context")
...     print("Leaving outer context")
...
Leaving inner context
Callback: from inner context
Leaving outer context
Callback: from outer context
```

29.8 abc --- 抽象基类

源代码： Lib/abc.py

该模块提供了在 Python 中定义*抽象基类* (ABC) 的组件，在 **PEP 3119** 中已有概述。查看 PEP 文档了解为什么需要在 Python 中增加这个模块。（也可查看 **PEP 3141** 以及*numbers* 模块了解基于 ABC 的数字类型继承关系。）

collections 模块中有一些派生自 ABC 的具体类；当然这些类还可以进一步被派生。此外，*collections.abc* 子模块中有一些 ABC 可被用于测试一个类或实例是否提供特定的接口，例如它是否可哈希或它是否为映射等。

该模块提供了一个元类*ABCMeta*，可以用来定义抽象类，另外还提供一个工具类*ABC*，可以用它以继承的方式定义抽象基类。

class abc.**ABC**

一个使用*ABCMeta* 作为元类的工具类。抽象基类可以通过从*ABC* 派生来简单地创建，这就避免了在某些情况下会令人混淆的元类用法，例如：

```
from abc import ABC

class MyABC(ABC):
    pass
```

注意*ABC* 的类型仍然是*ABCMeta*，因此继承*ABC* 仍然需要关注元类使用中的注意事项，比如可能会导致元类冲突的多重继承。当然你也可以直接使用*ABCMeta* 作为元类来定义抽象基类，例如：

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass
```

3.4 版新加入.

class abc.**ABCMeta**

用于定义抽象基类（ABC）的元类。

使用该元类以创建抽象基类。抽象基类可以像 mix-in 类一样直接被子类继承。你也可以将不相关的具体类（包括内建类）和抽象基类注册为“抽象子类”——这些类以及它们的子类会被内建函数*issubclass()* 识别为对应的抽象基类的子类，但是该抽象基类不会出现在其 MRO（Method Resolution Order，方法解析顺序）中，抽象基类中实现的方法也不可调用（即使通过*super()* 调用也不行）。[1]

使用*ABCMeta* 作为元类创建的类含有如下方法：

[1] C++ 程序员需要注意：Python 中虚基类的概念和 C++ 中的并不相同。

register(*subclass*)
将“子类”注册为该抽象基类的“抽象子类”，例如：

```
from abc import ABC

class MyABC(ABC):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

3.3 版更變: 返回注册的子类，使其能够作为类装饰器。

3.4 版更變: 你可以使用*get_cache_token()* 函数来检测对*register()* 的调用。

你也可以在虚基类中重载这个方法。

__subclasshook__(*subclass*)
(必须定义为类方法。)

检查 *subclass* 是否是该抽象基类的子类。也就是说对于那些你希望定义为该抽象基类的子类的类，你不用对每个类都调用*register()* 方法了，而是可以直接自定义 issubclass 的行为。(这个类方法是在抽象基类的 __subclasscheck__() 方法中调用的。)

该方法必须返回 True, False 或是 NotImplemented。如果返回 True，*subclass* 就会被认为是这个抽象基类的子类。如果返回 False，无论正常情况是否应该认为是其子类，统一视为不是。如果返回 NotImplemented，子类检查会按照正常机制继续执行。

为了对这些概念做一演示，请看以下定义 ABC 的示例：

```
class Foo:
    def __getitem__(self, index):
        ...
    def __len__(self):
        ...
    def get_iterator(self):
        return iter(self)

class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None

    def get_iterator(self):
        return self.__iter__()

    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented

MyIterable.register(Foo)
```

ABC MyIterable 定义了标准的迭代方法*__iter__()* 作为一个抽象方法。这里给出的实现仍可在

子类中被调用。get_iterator() 方法也是 MyIterable 抽象基类的一部分，但它并非必须被非抽象派生类所重载。

这里定义的*__subclasshook__()* 类方法指明了任何在其*__dict__* (或在其通过*__mro__* 列表访问的基类) 中具有*__iter__()* 方法的类也都会被视为 MyIterable。

最后，末尾行使得 Foo 成为 MyIterable 的一个虚子类，即使它没有定义*__iter__()* 方法（它使用了以 __len__() 和 __getitem__() 术语定义的旧式可迭代对象协议）。请注意这将不会使 get_iterator 成为 Foo 的一个可用方法，它是被另外提供的。

此外，*abc* 模块还提供了这些装饰器：

@abc.**abstractmethod**

用于声明抽象方法的装饰器。

使用此装饰器要求类的元类是*ABCMeta* 或是从该类派生。一个具有派生自*ABCMeta* 的元类的类不可以被实例化，除非它全部的抽象方法和特征属性均已被重载。抽象方法可通过任何普通的“super”调用机制来调用。*abstractmethod()* 可被用于声明特性属性和描述器的抽象方法。

不支持动态添加抽象方法到一个类，或试图在方法或类被创建后修改其抽象状态等操作。*abstractmethod()* 只会影响使用常规继承所派生的子类；通过 ABC 的 register() 方法注册的“虚子类”不会受到影响。

当*abstractmethod()* 与其他方法描述符配合应用时，它应当被应用为最内层的装饰器，如以下用法示例所示：

```
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, arg1):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg2):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg3):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

为了能正确地与抽象基类机制实现互操作，描述符必须使用 __isabstractmethod__ 将自身标识为抽象的。通常，如果被用于组成描述符的任何方法都是抽象的则此属性应当为 True。例如，Python 的内置*property* 所做的就等价于：

```
class Descriptor:
    ...
    @property
    def __isabstractmethod__(self):
        return any(getattr(f, '__isabstractmethod__', False) for
                   f in (self._fget, self._fset, self._fdel))
```

備🄵: 不同于 Java 抽象方法，这些抽象方法可能具有一个实现。这个实现可在重载它的类上通过*super()* 机制来调用。这在使用协作多重继承的框架中可以被用作超调用的一个端点。

---

abc 模块还支持下列旧式装饰器:

@abc.**abstractclassmethod**

3.2 版新加入.

3.3 版後已🄵用: 现在可以让*classmethod* 配合*abstractmethod()* 使用，使得此装饰器变得冗余。

内置*classmethod()* 的子类，指明一个抽象类方法。在其他方面它都类似于*abstractmethod()*。

这个特例已被弃用，因为现在当*classmethod()* 装饰器应用于抽象方法时它会被正确地标识为抽象的:

```
class C(ABC):
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg):
        ...
```

@abc.**abstractstaticmethod**

3.2 版新加入.

3.3 版後已🄵用: 现在可以让*staticmethod* 配合*abstractmethod()* 使用，使得此装饰器变得冗余。

内置*staticmethod()* 的子类，指明一个抽象静态方法。在其他方面它都类似于*abstractmethod()*。

这个特例已被弃用，因为现在当*staticmethod()* 装饰器应用于抽象方法时它会被正确地标识为抽象的:

```
class C(ABC):
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg):
        ...
```

@abc.**abstractproperty**

3.3 版後已🄵用: 现在可以让*property*，property.getter()，property.setter() 和 property.deleter() 配合*abstractmethod()* 使用，使得此装饰器变得冗余。

内置*property()* 的子类，指明一个抽象特性属性。

这个特例已被弃用，因为现在当*property()* 装饰器应用于抽象方法时它会被正确地标识为抽象的:

```
class C(ABC):
    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
```

上面的例子定义了一个只读特征属性；你也可以通过适当地将一个或多个下层方法标记为抽象的来定义可读写的抽象特征属性:

```
class C(ABC):
    @property
    def x(self):
        ...

    @x.setter
    @abstractmethod
    def x(self, val):
        ...
```

如果只有某些组件是抽象的，则只需更新那些组件即可在子类中创建具体的特征属性:

```
class D(C):
    @C.x.setter
    def x(self, val):
        ...
```

abc 模块还提供了这些函数:

abc.**get_cache_token**()
返回当前抽象基类的缓存令牌

此令牌是一个不透明对象（支持相等性测试），用于为虚子类标识抽象基类缓存的当前版本。此令牌会在任何 ABC 上每次调用*ABCMeta.register()* 时发生更改。

3.4 版新加入.

🄵解

29.9 atexit --- 退出处理器

---

atexit 模块定义了清理函数的注册和反注册函数. 被注册的函数会在解释器正常终止时执行. *atexit* 会按照注册顺序的 * 逆序 * 执行; 如果你注册了 A, B 和 C, 那么在解释器终止时会依序执行 C, B, A.

注意: 通过该模块注册的函数, 在程序被未被 Python 捕获的信号杀死时并不会执行, 在检测到 Python 内部致命错误以及调用了*os._exit()* 时也不会执行.

3.7 版更變: 当配合 C-API 子解释器使用时，已注册函数是它们所注册解释器中的局部对象。

atexit.**register**(*func*, **args*, ***kwargs*)
将 *func* 注册为终止时执行的函数. 任何传给 *func* 的可选的参数都应当作为参数传给*register()*. 可以多次注册同样的函数及参数.

在正常的程序终止时 (举例来说, 当调用了*sys.exit()* 或是主模块的执行完成时), 所有注册过的函数都会以后进先出的顺序执行. 这样做是假定更底层的模块通常会比高层模块更早引入, 因此需要更晚清理.

如果在 exit 处理句柄执行期间引发了异常，将会打印回溯信息 (除非引发的是*SystemExit*) 并且异常信息会被保存。在所有 exit 处理句柄都获得运行机会之后，所引发的最后一个异常会被重新引发。

这个函数返回 *func* 对象，可以把它当作装饰器使用。

atexit.**unregister**(*func*)
将 *func* 移出当解释器关闭时要运行的函数列表。如果 *func* 之前未被注册则*unregister()* 将静默地不做任何事。如果 *func* 已被注册一次以上，则该函数每次在*atexit* 调用栈中的出现都将被移除。当取消注册时会在内部使用相等性比较 (==)，因而函数引用不需要具有匹配的标识号。

也参考:

模块*readline* 使用*atexit* 读写*readline* 历史文件的有用的例子。

29.9.1 atexit 示例

以下简单例子演示了一个模块在被导入时如何从文件初始化一个计数器，并在程序终结时自动保存计数器的更新值，此操作不依赖于应用在终结时对此模块进行显式调用。:

```
try:
    with open('counterfile') as infile:
        _count = int(infile.read())
except FileNotFoundError:
    _count = 0

def incrcounter(n):
    global _count
    _count = _count + n

def savecounter():
    with open('counterfile', 'w') as outfile:
        outfile.write('%d' % _count)

import atexit

atexit.register(savecounter)
```

位置和关键字参数也可传入*register()* 以便传递给被调用的已注册函数:

```
def goodbye(name, adjective):
    print('Goodbye %s, it was %s to meet you.' % (name, adjective))

import atexit

atexit.register(goodbye, 'Donny', 'nice')
# or:
atexit.register(goodbye, adjective='nice', name='Donny')
```

作为*decorator*: 使用:

```
import atexit

@atexit.register
def goodbye():
    print('You are now leaving the Python sector.')
```

只有在函数不需要任何参数调用时才能工作.

29.10 traceback --- 打印或检索堆栈回溯

源代码： Lib/traceback.py

该模块提供了一个标准接口来提取、格式化和打印 Python 程序的堆栈跟踪结果。它完全模仿 Python 解释器在打印堆栈跟踪结果时的行为。当您想要在程序控制下打印堆栈跟踪结果时，例如在“封装”解释器时，这是非常有用的。

这个模块使用 traceback 对象——这是存储在*sys.last_traceback* 中的对象类型变量，并作为*sys.exc_info()* 的第三项被返回。

这个模块定义了以下函数：

traceback.**print_tb**(*tb*, *limit=None*, *file=None*)
如果 *limit* 是正整数，那么从 traceback 对象”tb” 输出最高 *limit* 个（从调用函数开始的）栈的堆栈回溯条目；如果 *limit* 是负数就输出 `abs(limit)` 个回溯条目；又如果 *limit* 被省略或者为 `None`，那么就会输出所有回溯条目。如果 *file* 被省略或为 `None` 那么就会输出至标准输出 “sys.stderr“否则它应该是一个打开的文件或者文件类对象来接收输出

3.5 版更變: 添加了对负数值 *limit* 的支持

traceback.**print_exception**(*etype*, *value*, *tb*, *limit=None*, *file=None*, *chain=True*)
打印回溯对象 *tb* 到 *file* 的异常信息和整个堆栈回溯。这和*print_tb()* 比有以下方面不同：

- 如果 *tb* 不为 `None`，它将打印头部 `Traceback (most recent call last):`
- 它将在输出完堆栈回溯后，输出异常中的 *etype* 和 *value* 信息
- if *type(value)* is *SyntaxError* and *value* has the appropriate format, it prints the line where the syntax error occurred with a caret indicating the approximate position of the error.

The optional *limit* argument has the same meaning as for *print_tb()*. If *chain* is true (the default), then chained exceptions (the `__cause__` or `__context__` attributes of the exception) will be printed as well, like the interpreter itself does when printing an unhandled exception.

3.5 版更變: The *etype* argument is ignored and inferred from the type of *value*.

traceback.**print_exc**(*limit=None*, *file=None*, *chain=True*)
This is a shorthand for `print_exception(*sys.exc_info(), limit, file, chain)`.

traceback.**print_last**(*limit=None*, *file=None*, *chain=True*)
This is a shorthand for `print_exception(sys.last_type, sys.last_value, sys.last_traceback, limit, file, chain)`. In general it will work only after an exception has reached an interactive prompt (see *sys.last_type*).

traceback.**print_stack**(*f=None*, *limit=None*, *file=None*)
Print up to *limit* stack trace entries (starting from the invocation point) if *limit* is positive. Otherwise, print the last `abs(limit)` entries. If *limit* is omitted or `None`, all entries are printed. The optional *f* argument can be used to specify an alternate stack frame to start. The optional *file* argument has the same meaning as for *print_tb()*.

3.5 版更變: 添加了对负数值 *limit* 的支持

traceback.**extract_tb**(*tb*, *limit=None*)
Return a *StackSummary* object representing a list of ”pre-processed” stack trace entries extracted from the traceback object *tb*. It is useful for alternate formatting of stack traces. The optional *limit* argument has the same meaning as for *print_tb()*. A ”pre-processed” stack trace entry is a *FrameSummary* object containing attributes `filename`, `lineno`, `name`, and `line` representing the information that is usually printed for a stack trace. The `line` is a string with leading and trailing whitespace stripped; if the source is not available it is `None`.

traceback.**extract_stack**(*f=None*, *limit=None*)
Extract the raw traceback from the current stack frame. The return value has the same format as for *extract_tb()*. The optional *f* and *limit* arguments have the same meaning as for *print_stack()*.

traceback.**format_list**(*extracted_list*)
Given a list of tuples or *FrameSummary* objects as returned by *extract_tb()* or *extract_stack()*, return a list of strings ready for printing. Each string in the resulting list corresponds to the item with the same index in the argument list. Each string ends in a newline; the strings may contain internal newlines as well, for those items whose source text line is not `None`.

traceback.**format_exception_only**(*etype*, *value*)
Format the exception part of a traceback. The arguments are the exception type and value such as given by `sys.last_type` and `sys.last_value`. The return value is a list of strings, each ending in a newline. Normally, the list contains a single string; however, for *SyntaxError* exceptions, it contains several lines that (when printed) display detailed information about where the syntax error occurred. The message indicating which exception occurred is the always last string in the list.

traceback.**format_exception**(*etype*, *value*, *tb*, *limit=None*, *chain=True*)
Format a stack trace and the exception information. The arguments have the same meaning as the corresponding arguments to *print_exception()*. The return value is a list of strings, each ending in a newline and some containing internal newlines. When these lines are concatenated and printed, exactly the same text is printed as does *print_exception()*.

3.5 版更變: The *etype* argument is ignored and inferred from the type of *value*.

traceback.**format_exc**(*limit=None*, *chain=True*)
This is like `print_exc(limit)` but returns a string instead of printing to a file.

traceback.**format_tb**(*tb*, *limit=None*)
A shorthand for `format_list(extract_tb(tb, limit))`.

traceback.**format_stack**(*f=None*, *limit=None*)
A shorthand for `format_list(extract_stack(f, limit))`.

traceback.**clear_frames**(*tb*)
Clears the local variables of all the stack frames in a traceback *tb* by calling the `clear()` method of each frame object.

3.4 版新加入.

traceback.**walk_stack**(*f*)
Walk a stack following `f.f_back` from the given frame, yielding the frame and line number for each frame. If *f* is `None`, the current stack is used. This helper is used with *StackSummary.extract()*.

3.5 版新加入.

traceback.**walk_tb**(*tb*)
Walk a traceback following `tb_next` yielding the frame and line number for each frame. This helper is used with *StackSummary.extract()*.

3.5 版新加入.

The module also defines the following classes:

29.10.1 `TracebackException` Objects

3.5 版新加入.

`TracebackException` objects are created from actual exceptions to capture data for later printing in a lightweight fashion.

class traceback.**TracebackException**(*exc_type*, *exc_value*, *exc_traceback*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*)

Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the *`StackSummary`* class.

Note that when locals are captured, they are also shown in the traceback.

__cause__
A *`TracebackException`* of the original `__cause__`.

__context__
A *`TracebackException`* of the original `__context__`.

__suppress_context__
The `__suppress_context__` value from the original exception.

stack
A *`StackSummary`* representing the traceback.

exc_type
The class of the original traceback.

filename
For syntax errors - the file name where the error occurred.

lineno
For syntax errors - the line number where the error occurred.

text
For syntax errors - the text where the error occurred.

offset
For syntax errors - the offset into the text where the error occurred.

msg
For syntax errors - the compiler error message.

classmethod from_exception(*exc*, ***, *limit=None*, *lookup_lines=True*, *capture_locals=False*)
Capture an exception for later rendering. *limit*, *lookup_lines* and *capture_locals* are as for the *`StackSummary`* class.

Note that when locals are captured, they are also shown in the traceback.

format(***, *chain=True*)
Format the exception.

If *chain* is not `True`, `__cause__` and `__context__` will not be formatted.

The return value is a generator of strings, each ending in a newline and some containing internal newlines. *`print_exception()`* is a wrapper around this method which just prints the lines to a file.

The message indicating which exception occurred is always the last string in the output.

format_exception_only()
Format the exception part of the traceback.

The return value is a generator of strings, each ending in a newline.

Normally, the generator emits a single string; however, for *SyntaxError* exceptions, it emits several lines that (when printed) display detailed information about where the syntax error occurred.

The message indicating which exception occurred is always the last string in the output.

29.10.2 StackSummary Objects

3.5 版新加入.

StackSummary objects represent a call stack ready for formatting.

class traceback.**StackSummary**

classmethod extract(*frame_gen*, *, *limit=None*, *lookup_lines=True*, *capture_locals=False*)
Construct a *StackSummary* object from a frame generator (such as is returned by *walk_stack()* or *walk_tb()*).

If *limit* is supplied, only this many frames are taken from *frame_gen*. If *lookup_lines* is `False`, the returned *FrameSummary* objects will not have read their lines in yet, making the cost of creating the *StackSummary* cheaper (which may be valuable if it may not actually get formatted). If *capture_locals* is `True` the local variables in each *FrameSummary* are captured as object representations.

classmethod from_list(*a_list*)
Construct a *StackSummary* object from a supplied list of *FrameSummary* objects or old-style list of tuples. Each tuple should be a 4-tuple with filename, lineno, name, line as the elements.

format()
Returns a list of strings ready for printing. Each string in the resulting list corresponds to a single frame from the stack. Each string ends in a newline; the strings may contain internal newlines as well, for those items with source text lines.

For long sequences of the same frame and line, the first few repetitions are shown, followed by a summary line stating the exact number of further repetitions.

3.6 版更變: Long sequences of repeated frames are now abbreviated.

29.10.3 FrameSummary Objects

3.5 版新加入.

FrameSummary objects represent a single frame in a traceback.

class traceback.**FrameSummary**(*filename*, *lineno*, *name*, *lookup_line=True*, *locals=None*, *line=None*)
Represent a single frame in the traceback or stack that is being formatted or printed. It may optionally have a stringified version of the frames locals included in it. If *lookup_line* is `False`, the source code is not looked up until the *FrameSummary* has the `line` attribute accessed (which also happens when casting it to a tuple). `line` may be directly provided, and will prevent line lookups happening at all. *locals* is an optional local variable dictionary, and if supplied the variable representations are stored in the summary for later display.

29.10.4 Traceback Examples

This simple example implements a basic read-eval-print loop, similar to (but less useful than) the standard Python interactive interpreter loop. For a more complete implementation of the interpreter loop, refer to the *code* module.

```
import sys, traceback

def run_user_code(envdir):
    source = input(">>> ")
    try:
        exec(source, envdir)
    except Exception:
        print("Exception in user code:")
        print("-"*60)
        traceback.print_exc(file=sys.stdout)
        print("-"*60)

envdir = {}
while True:
    run_user_code(envdir)
```

The following example demonstrates the different ways to print and format the exception and traceback:

```
import sys, traceback

def lumberjack():
    bright_side_of_death()

def bright_side_of_death():
    return tuple()[0]

try:
    lumberjack()
except IndexError:
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print("*** print_tb:")
    traceback.print_tb(exc_traceback, limit=1, file=sys.stdout)
    print("*** print_exception:")
    # exc_type below is ignored on 3.5 and later
    traceback.print_exception(exc_type, exc_value, exc_traceback,
                              limit=2, file=sys.stdout)
    print("*** print_exc:")
    traceback.print_exc(limit=2, file=sys.stdout)
    print("*** format_exc, first and last line:")
    formatted_lines = traceback.format_exc().splitlines()
    print(formatted_lines[0])
    print(formatted_lines[-1])
    print("*** format_exception:")
    # exc_type below is ignored on 3.5 and later
    print(repr(traceback.format_exception(exc_type, exc_value,
                                          exc_traceback)))
    print("*** extract_tb:")
    print(repr(traceback.extract_tb(exc_traceback)))
    print("*** format_tb:")
    print(repr(traceback.format_tb(exc_traceback)))
    print("*** tb_lineno:", exc_traceback.tb_lineno)
```

The output for the example would look similar to this:

```
*** print_tb:
  File "<doctest...>", line 10, in <module>
    lumberjack()
*** print_exception:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** print_exc:
Traceback (most recent call last):
  File "<doctest...>", line 10, in <module>
    lumberjack()
  File "<doctest...>", line 4, in lumberjack
    bright_side_of_death()
IndexError: tuple index out of range
*** format_exc, first and last line:
Traceback (most recent call last):
IndexError: tuple index out of range
*** format_exception:
['Traceback (most recent call last):\n',
 '  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n',
 'IndexError: tuple index out of range\n']
*** extract_tb:
[<FrameSummary file <doctest...>, line 10 in <module>>,
 <FrameSummary file <doctest...>, line 4 in lumberjack>,
 <FrameSummary file <doctest...>, line 7 in bright_side_of_death>]
*** format_tb:
['  File "<doctest...>", line 10, in <module>\n    lumberjack()\n',
 '  File "<doctest...>", line 4, in lumberjack\n    bright_side_of_death()\n',
 '  File "<doctest...>", line 7, in bright_side_of_death\n    return tuple()[0]\n']
*** tb_lineno: 10
```

The following example shows the different ways to print and format the stack:

```
>>> import traceback
>>> def another_function():
...     lumberstack()
...
>>> def lumberstack():
...     traceback.print_stack()
...     print(repr(traceback.extract_stack()))
...     print(repr(traceback.format_stack()))
...
>>> another_function()
  File "<doctest>", line 10, in <module>
    another_function()
  File "<doctest>", line 3, in another_function
    lumberstack()
  File "<doctest>", line 6, in lumberstack
    traceback.print_stack()
[('<doctest>', 10, '<module>', 'another_function()'),
 ('<doctest>', 3, 'another_function', 'lumberstack()'),
 ('<doctest>', 7, 'lumberstack', 'print(repr(traceback.extract_stack()))')]
```

(下页继续)

(繼續上一頁)

```
['  File "<doctest>", line 10, in <module>\n    another_function()\n',
 '  File "<doctest>", line 3, in another_function\n    lumberstack()\n',
 '  File "<doctest>", line 8, in lumberstack\n    print(repr(traceback.format_
↪stack()))\n']
```

This last example demonstrates the final few formatting functions:

```
>>> import traceback
>>> traceback.format_list([('spam.py', 3, '<module>', 'spam.eggs()'),
...                        ('eggs.py', 42, 'eggs', 'return "bacon"')])
['  File "spam.py", line 3, in <module>\n    spam.eggs()\n',
 '  File "eggs.py", line 42, in eggs\n    return "bacon"\n']
>>> an_error = IndexError('tuple index out of range')
>>> traceback.format_exception_only(type(an_error), an_error)
['IndexError: tuple index out of range\n']
```

29.11 __future__ --- Future 语句定义

源代码： Lib/__future__.py

__future__ 是一个真正的模块，这主要有 3 个原因：

- 避免混淆已有的分析 import 语句并查找 import 的模块的工具。
- 确保 future 语句在 2.1 之前的版本运行时至少能抛出 runtime 异常（import *__future__* 会失败，因为 2.1 版本之前没有这个模块）。
- 当引入不兼容的修改时，可以记录其引入的时间以及强制使用的时间。这是一种可执行的文档，并且可以通过 import *__future__* 来做程序性的检查。

`__future__.py` 中的每一条语句都是以下格式的：

```
FeatureName = _Feature(OptionalRelease, MandatoryRelease,
                       CompilerFlag)
```

通常 *OptionalRelease* 要比 *MandatoryRelease* 小，并且都是和*sys.version_info* 格式一致的 5 元素元组。

```
(PY_MAJOR_VERSION, # the 2 in 2.1.0a3; an int
 PY_MINOR_VERSION, # the 1; an int
 PY_MICRO_VERSION, # the 0; an int
 PY_RELEASE_LEVEL, # "alpha", "beta", "candidate" or "final"; string
 PY_RELEASE_SERIAL # the 3; an int
)
```

OptionalRelease 记录了一个特性首次发布时的 Python 版本。

在 *MandatoryRelases* 还没有发布时，*MandatoryRelease* 表示该特性会变成语言的一部分的预测时间。

其他情况下，*MandatoryRelease* 用来记录这个特性是何时成为语言的一部分的。从该版本往后，使用该特性将不需要 future 语句，不过很多人还是会加上对应的 import。

MandatoryRelease 也可能是 `None`, 表示这个特性已经被撤销。

`_Feature` 类的实例有两个对应的方法，`getOptionalRelease()` 和 `getMandatoryRelease()`。

CompilerFlag 是一个（位）标记，对于动态编译的代码，需要将这个标记作为第四个参数传入内建函数*compile()* 中以开启对应的特性。这个标记存储在 _Feature 类实例的 compiler_flag 属性中。

__future__ 中不会删除特性的描述。从 Python 2.1 中首次加入以来，通过这种方式引入了以下特性：

特性	可选版本	强制加入版本	效果
nested_scopes	2.1.0b1	2.2	**PEP 227**: *Statically Nested Scopes*
generators	2.2.0a1	2.3	**PEP 255**: *Simple Generators*
division	2.2.0a2	3.0	**PEP 238**: *Changing the Division Operator*
absolute_import	2.5.0a1	3.0	**PEP 328**: *Imports: Multi-Line and Absolute/Relative*
with_statement	2.5.0a1	2.6	**PEP 343**: *The "with" Statement*
print_function	2.6.0a2	3.0	**PEP 3105**: *Make print a function*
unicode_literals	2.6.0a2	3.0	**PEP 3112**: *Bytes literals in Python 3000*
generator_stop	3.5.0b1	3.7	**PEP 479**: *StopIteration handling inside generators*
annotations	3.7.0b1	TBD[1]	**PEP 563**: *Postponed evaluation of annotations*

也参考:

future 编译器怎样处理 future import。

29.12 gc --- 垃圾回收器接口

此模块提供可选的垃圾回收器的接口，提供的功能包括：关闭收集器、调整收集频率、设置调试选项。它同时提供对回收器找到但是无法释放的不可达对象的访问。由于 Python 使用了带有引用计数的回收器，如果你确定你的程序不会产生循环引用，你可以关闭回收器。可以通过调用 gc.disable() 关闭自动垃圾回收。若要调试一个存在内存泄漏的程序，调用 gc.set_debug(gc.DEBUG_LEAK)；需要注意的是，它包含 gc.DEBUG_SAVEALL，使得被垃圾回收的对象会被存放在 gc.garbage 中以待检查。

gc 模块提供下列函数：

gc.**enable**()
: 启用自动垃圾回收

gc.**disable**()
: 停用自动垃圾回收

gc.**isenabled**()
: 如果启用了自动回收则返回 True。

gc.**collect**(*generation=2*)
: 若被调用时不包含参数，则启动完全的垃圾回收。可选的参数 *generation* 可以是一个整数，指明需要回收哪一代（从 0 到 2）的垃圾。当参数 *generation* 无效时，会引发*ValueError* 异常。返回发现的不可达对象的数目。

  每当运行完整收集或最高代 (2) 收集时，为多个内置类型所维护的空闲列表会被清空。由于特定类型特别是*float* 的实现，在某些空闲列表中并非所有项都会被释放。

[1] from __future__ import annotations was previously scheduled to become mandatory in Python 3.10, but the Python Steering Council twice decided to delay the change (announcement for Python 3.10; announcement for Python 3.11). No final decision has been made yet. See also **PEP 563** and **PEP 649**.

gc.**set_debug**(*flags*)
设置垃圾回收器的调试标识位。调试信息会被写入 sys.stderr 。此文档末尾列出了各个标志位及其含义；可以使用位操作对多个标志位进行设置以控制调试器。

gc.**get_debug**()
返回当前调试标识位。

gc.**get_objects**(*generation=None*)
返回一个收集器所跟踪的所有对象的列表，所返回的列表除外。如果 *generation* 不为 None，则只返回收集器所跟踪的属于该生成的对象。

3.8 版更變: 新的 *generation* 形参。

引发一个审计事件 gc.get_objects，附带参数 generation。

gc.**get_stats**()
返回一个包含三个字典对象的列表，每个字典分别包含对应代的从解释器开始运行的垃圾回收统计数据。字典的键的数目在将来可能发生改变，目前每个字典包含以下内容：

- collections 是该代被回收的次数；
- collected 是该代中被回收的对象总数；
- uncollectable 是在这一代中被发现无法收集的对象总数（因此被移动到*garbage* 列表中）。

3.4 版新加入.

gc.**set_threshold**(*threshold0*[, *threshold1*[, *threshold2*]])
设置垃圾回收阈值（收集频率）。将 *threshold0* 设为零会禁用回收。

垃圾回收器把所有对象分类为三代，其依据是对象在多少次垃圾回收后幸存。新建对象会被放在最年轻代（第 0 代）。如果一个对象在一次垃圾回收后幸存，它会被移入下一个较老代。由于第 2 代是最老代，这一代的对象在一次垃圾回收后仍会保留原样。为了确定何时要运行，垃圾回收器会跟踪自上一次回收后对象分配和释放的数量。当分配数量减去释放数量的结果值大于 *threshold0* 时，垃圾回收就会开始。初始时只有第 0 代会被检查。如果自第 1 代被检查后第 0 代已被检查超过 *threshold1* 次，则第 1 也会被检查。对于第三代来说情况还会更复杂，请参阅 Collecting the oldest generation 来了解详情。

gc.**get_count**()
将当前回收计数以形为 (count0, count1, count2) 的元组返回。

gc.**get_threshold**()
将当前回收阈值以形为 (threshold0, threshold1, threshold2) 的元组返回。

gc.**get_referrers**(**objs*)
返回直接引用任意一个 *objs* 的对象列表。这个函数只定位支持垃圾回收的容器；引用了其它对象但不支持垃圾回收的扩展类型不会被找到。

需要注意的是，已经解除对 *objs* 引用的对象，但仍存在于循环引用中未被回收时，仍然会被作为引用者出现在返回的列表当中。若要获取当前正在引用 *objs* 的对象，需要调用*collect()* 然后再调用*get_referrers()* 。

> **警告:** 在使用*get_referrers()* 返回的对象时必须要小心，因为其中一些对象可能仍在构造中因此处于暂时的无效状态。不要把*get_referrers()* 用于调试以外的其它目的。

引发一个审计事件 gc.get_referrers，附带参数 objs。

gc.**get_referents**(**objs*)
返回被任意一个参数中的对象直接引用的对象的列表。返回的被引用对象是被参数中的对象的 C 语言级别方法（若存在）tp_traverse 访问到的对象，可能不是所有的实际直接可达对象。只有支持垃圾回收的对象支持 tp_traverse 方法，并且此方法只会在需要访问涉及循环引用的对象时使用。因此，

可以有以下例子：一个整数对其中一个参数是直接可达的，这个整数有可能出现或不出现在返回的结果列表当中。

引发一个审计事件 gc.get_referents，附带参数 objs。

gc.**is_tracked**(*obj*)
当对象正在被垃圾回收器监控时返回 True，否则返回 False。一般来说，原子类的实例不会被监控，而非原子类（如容器、用户自定义的对象）会被监控。然而，会有一些特定类型的优化以便减少垃圾回收器在简单实例（如只含有原子性的键和值的字典）上的消耗。

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

3.1 版新加入.

gc.**is_finalized**(*obj*)
如果给定对象已被垃圾回收器终结则返回 True，否则返回 False。

```
>>> x = None
>>> class Lazarus:
...     def __del__(self):
...         global x
...         x = self
...
>>> lazarus = Lazarus()
>>> gc.is_finalized(lazarus)
False
>>> del lazarus
>>> gc.is_finalized(x)
True
```

3.9 版新加入.

gc.**freeze**()
冻结 gc 所跟踪的所有对象——将它们移至永久代并忽略所有未来的集合。这可以在 POSIX fork() 调用之前使用以便令对写入复制保持友好或加速收集。并且在 POSIX fork() 调用之前的收集也可以释放页面以供未来分配，这也可能导致写入时复制，因此建议在主进程中禁用 gc 并在 fork 之前冻结，而在子进程中启用 gc。

3.7 版新加入.

gc.**unfreeze**()
解冻永久代中的对象，并将它们放回到年老代中。

3.7 版新加入.

gc.**get_freeze_count**()
返回永久代中的对象数量。

3.7 版新加入.

提供以下变量仅供只读访问（你可以修改但不应该重绑定它们）：

gc.**garbage**
一个回收器发现不可达而又无法被释放的对象（不可回收对象）列表。从 Python 3.4 开始，该列表在大多数时候都应该是空的，除非使用了含有非 NULL tp_del 空位的 C 扩展类型的实例。

如果设置了*DEBUG_SAVEALL*，则所有不可访问对象将被添加至该列表而不会被释放。

3.2 版更變: 当*interpreter shutdown* 即解释器关闭时，若此列表非空，会产生*ResourceWarning*，即资源警告，在默认情况下此警告不会被提醒。如果设置了*DEBUG_UNCOLLECTABLE*，所有无法被回收的对象会被打印。

3.4 版更變: 根据 **PEP 442**，带有 __del__() 方法的对象最终不再会进入*gc.garbage*。

gc.**callbacks**
在垃圾回收器开始前和完成后会被调用的一系列回调函数。这些回调函数在被调用时使用两个参数：*phase* 和 *info*。

phase 可为以下两值之一：

”start”: 垃圾回收即将开始。

”stop”: 垃圾回收已结束。

info is a dict providing more information for the callback. The following keys are currently defined:

”generation”（代）：正在被回收的最久远的一代。

”collected”（已回收的）：当 *phase* 为”stop” 时，被成功回收的对象的数目。

”uncollectable”（不可回收的）：当 *phase* 为”stop” 时，不能被回收并被放入*garbage*的对象的数目。

应用程序可以把他们自己的回调函数加入此列表。主要的使用场景有：

统计垃圾回收的数据，如：不同代的回收频率、回收所花费的时间。

使应用程序可以识别和清理他们自己的在*garbage*中的不可回收类型的对象。

3.3 版新加入.

以下常量被用于*set_debug()*：

gc.**DEBUG_STATS**
在回收完成后打印统计信息。当回收频率设置较高时，这些信息会比较有用。

gc.**DEBUG_COLLECTABLE**
当发现可回收对象时打印信息。

gc.**DEBUG_UNCOLLECTABLE**
打印找到的不可回收对象的信息（指不能被回收器回收的不可达对象）。这些对象会被添加到 garbage 列表中。

3.2 版更變: 当*interpreter shutdown* 时，即解释器关闭时，若*garbage*列表中存在对象，这些对象也会被打印输出。

gc.**DEBUG_SAVEALL**
设置后，所有回收器找到的不可达对象会被添加进 *garbage* 而不是直接被释放。这在调试一个内存泄漏的程序时会很有用。

gc.**DEBUG_LEAK**
调试内存泄漏的程序时，使回收器打印信息的调试标识位。（等价于 DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL）。

29.13 inspect --- 检查对象

源代码: Lib/inspect.py

---

inspect 模块提供了一些有用的函数帮助获取对象的信息，例如模块、类、方法、函数、回溯、帧对象以及代码对象。例如它可以帮助你检查类的内容，获取某个方法的源代码，取得并格式化某个函数的参数列表，或者获取你需要显示的回溯的详细信息。

该模块提供了 4 种主要的功能：类型检查、获取源代码、检查类与函数、检查解释器的调用堆栈。

29.13.1 类型和成员

getmembers() 函数获取对象的成员，例如类或模块。函数名以"is" 开始的函数主要作为*getmembers()* 的第 2 个参数使用。它们也可用于判定某对象是否有如下的特殊属性:

类型	属性	描述
module 模块	__doc__	文档字符串
	__file__	文件名 (内置模块没有文件名)
class 类	__doc__	文档字符串
	__name__	类定义时所使用的名称
	__qualname__	qualified name -- 限定名称
	__module__	该类型被定义时所在的模块的名称
method 方法	__doc__	文档字符串
	__name__	该方法定义时所使用的名称
	__qualname__	qualified name -- 限定名称
	__func__	实现该方法的函数对象
	__self__	该方法被绑定的实例，若没有绑定则为 None
	__module__	定义此方法的模块的名称
函数	__doc__	文档字符串
	__name__	用于定义此函数的名称
	__qualname__	qualified name -- 限定名称
	__code__	包含已编译函数的代码对象*bytecode*
	__defaults__	所有位置或关键字参数的默认值的元组
	__kwdefaults__	保存仅关键字参数的所有默认值的映射
	__globals__	包含此函数定义的全局命名空间
	__annotations__	参数名称到注解的映射；保留键 "return" 用于返回值注解。
	__module__	此函数定义所在的模块名称
回溯	tb_frame	此级别的框架对象
	tb_lasti	在字节码中最后尝试的指令的索引
	tb_lineno	当前行在 Python 源代码中的行号
	tb_next	下一个内部回溯对象（由本层调用）
框架	f_back	下一个外部帧对象（此帧的调用者）
	f_builtins	此帧所看到的 buildins 命名空间
	f_code	在此帧中执行的代码对象
	f_globals	此帧所看到的全局命名空间
	f_lasti	在字节码中最后尝试的指令的索引
	f_lineno	当前行在 Python 源代码中的行号
	f_locals	此帧所看到的局部命名空间
	f_trace	此帧的追踪函数，或 "None"

繼續

表 1 – 繼續上一頁

类型	属性	描述
code	co_argcount	参数数量（不包括仅关键字参数、* 或 ** 参数）
	co_code	原始编译字节码的字符串
	co_cellvars	单元变量名称的元组 (通过包含作用域引用)
	co_consts	字节码中使用的常量元组
	co_filename	创建此代码对象的文件的名称
	co_firstlineno	第一行在 Python 源码的行号
	co_flags	`CO_*` 标志的位图，详见此处
	co_lnotab	编码的行号到字节码索引的映射
	co_freevars	自由变量的名字组成的元组（通过函数闭包引用）
	co_posonlyargcount	仅限位置参数的数量
	co_kwonlyargcount	仅限关键字参数的数量（不包括 ** 参数）
	co_name	定义此代码对象的名称
	co_names	局部变量名称的元组
	co_nlocals	局部变量的数量
	co_stacksize	需要虚拟机堆栈空间
	co_varnames	参数名和局部变量的元组
生成器	__name__	名称
	__qualname__	qualified name -- 限定名称
	gi_frame	框架
	gi_running	生成器在运行吗?
	gi_code	code
	gi_yieldfrom	通过 “yield from“迭代的对象，或 “None“
协程	__name__	名称
	__qualname__	qualified name -- 限定名称
	cr_await	正在等待的对象，或 “None“
	cr_frame	框架
	cr_running	这个协程正在运行吗?
	cr_code	code
	cr_origin	协程被创建的位置，或 “None“。参见*sys.set_coroutine_origin_tracking_dep*
builtin	__doc__	文档字符串
	__name__	此函数或方法的原始名称
	__qualname__	qualified name -- 限定名称
	__self__	方法绑定到的实例，或 “None“

3.5 版更變: 为生成器添加 “__qualname__“和 “gi_yieldfrom“属性。

生成器的 “__name__“属性现在由函数名称设置，而不是代码对象名称，并且现在可以被修改。

3.7 版更變: 为协程添加 “cr_origin“属性。

`inspect.`**`getmembers`**(*object*[, *predicate*])

返回一个对象上的所有成员，组成以（名称，值）对为元素的列表，按名称排序。如果提供了可选的 *predicate* 参数（会对每个成员的 值对象进行一次调用），则仅包含该断言为真的成员。

備𝔽: 如果要让*getmembers()* 返回元类中定义的类属性，那么实参须为一个类且这些属性在元类的自定义方法 `__dir__()` 中列出。

`inspect.`**`getmodulename`**(*path*)

返回由文件名 *path* 表示的模块名字，但不包括外层的包名。文件扩展名会检查是否在*importlib.machinery.all_suffixes()* 列出的条目中。如果符合，则文件路径的最后一个组成部分会去掉后缀名后被返回；否则返回 “None“。

值得注意的是，这个函数 仅返回可以作为 Python 模块的名字，而有可能指向一个 Python 包的路径仍然会返回 “None“。

3.3 版更變: 该函数直接基于*importlib*。

inspect.**ismodule**(*object*)
当该对象是一个模块时返回 “True“。

inspect.**isclass**(*object*)
当该对象是一个类时返回 “True“，无论是内置类或者 Python 代码中定义的类。

inspect.**ismethod**(*object*)
当该对象是一个 Python 写成的绑定方法时返回 “True“。

inspect.**isfunction**(*object*)
当该对象是一个 Python 函数时（包括使用*lambda* 表达式创造的函数），返回 “True“。

inspect.**isgeneratorfunction**(*object*)
当该对象是一个 Python 生成器函数时返回 “True“。

3.8 版更變: 对于使用*functools.partial()* 封装的函数，如果被封装的函数是一个 Python 生成器函数，现在也会返回 “True“。

inspect.**isgenerator**(*object*)
当该对象是一个生成器时返回 “True“。

inspect.**iscoroutinefunction**(*object*)
当该对象是一个协程函数 （通过 `async def` 语法定义的函数）。

3.5 版新加入.

3.8 版更變: 对于使用*functools.partial()* 封装的函数，如果被封装的函数是一个协程函数 ，现在也会返回 “True“。

inspect.**iscoroutine**(*object*)
当该对象是一个由 `async def` 函数创建的协程 时返回 “True“。

3.5 版新加入.

inspect.**isawaitable**(*object*)
如果该对象可以在 `await` 表达式中使用时返回 “True“。

也可以用于区分基于生成器的协程和常规的生成器：

```
def gen():
    yield
@types.coroutine
def gen_coro():
    yield

assert not isawaitable(gen())
assert isawaitable(gen_coro())
```

3.5 版新加入.

inspect.**isasyncgenfunction**(*object*)
如果该对象是一个异步生成器 函数则返回 “True“，例如：

```
>>> async def agen():
...     yield 1
...
>>> inspect.isasyncgenfunction(agen)
True
```

3.6 版新加入.

3.8 版更變: 对于使用*functools.partial()* 封装的函数，如果被封装的函数是一个异步生成器，现在也会返回 “True“。

inspect.**isasyncgen**(*object*)
如果该对象是一个由异步生成器 函数创建的异步生成器迭代器，则返回 True。

3.6 版新加入.

inspect.**istraceback**(*object*)
如果该对象是一个回溯则返回 True。

inspect.**isframe**(*object*)
如果该对象是一个帧则返回 True。

inspect.**iscode**(*object*)
如果该对象是一个代码对象则返回 True。

inspect.**isbuiltin**(*object*)
如果该对象是一个内置函数或一个绑定的内置方法，则返回 True。

inspect.**isroutine**(*object*)
如果该对象是一个用户定义的或内置的函数或者方法，则返回 True。

inspect.**isabstract**(*object*)
如果该对象是一个抽象基类则返回 True。

inspect.**ismethoddescriptor**(*object*)
如果该对象是一个方法描述器，但*ismethod()* 、*isclass()* 、*isfunction()* 及*isbuiltin()* 均不为真。

例如，该函数对于 int.__add__ 为真。一个通过此测试的对象可以有 __get__() 方法，但不能有 __set__() 方法，除此以外的属性集合是可变的。一个*__name__* 属性通常是合理的，而 __doc__ 属性常常也是。

即使是通过描述器实现的函数，如果通过其他某一个测试，则*ismethoddescriptor()* 测试则会返回 False。这单纯是因为其他测试提供了更多保证。比如，当一个对象通过*ismethod()* 测试时，你可以使用 __func__ 等属性。

inspect.**isdatadescriptor**(*object*)
如果该对象是一个数据描述器则返回 True。

数据描述器拥有 __set__ 或 __delete__ 方法。比如属性（通过 Python 定义）、getset 和成员。后二者是通过 C 定义的，并且也有对应的更具体的测试，并且在不同的 Python 实现中也是健壮的。典型地，数据描述器会拥有*__name__* 和 __doc__ 属性（属性、getset 和成员都包含这两个属性），但这并无保证。

inspect.**isgetsetdescriptor**(*object*)
如果该对象是一个 getset 描述器则返回 True。

CPython implementation detail: getset 是在扩展模块中通过 PyGetSetDef 结构体定义的属性。对于不包含这种类型的 Python 实现，这个方法将永远返回 False。

inspect.**ismemberdescriptor**(*object*)
如果该对象是一个成员描述器则返回 True。

CPython implementation detail: 成员描述器是在扩展模块中通过 PyMemberDef 结构体定义的属性。对于不包含这种类型的 Python 实现，这个方法将永远返回 False。

29.13.2 获取源代码

inspect.**getdoc**(*object*)
获取对象的文档字符串并通过*cleandoc()* 进行清理。如果对象本身并未提供文档字符串并且这个对象是一个类、一个方法、一个属性或者一个描述器，将通过继承层级结构获取文档字符串。

3.5 版更變: 文档字符串没有被重写的话将会被继承。

inspect.**getcomments**(*object*)
任意多行注释作为单一一个字符串返回。对于类、函数和方法，选取紧贴在该对象的源代码之前的注释；对于模块，选取 Python 源文件顶部的注释。如果对象的源代码不可获得，返回 None。这可能是因为对象是 C 语言中或者是在交互式命令行中定义的。

inspect.**getfile**(*object*)
返回定义了这个对象的文件名（包括文本文件或二进制文件）。如何该对象是一个内置模块、类或函数则会失败并引发一个*TypeError*。

inspect.**getmodule**(*object*)
尝试猜测该对象是在哪个模块中定义的。

inspect.**getsourcefile**(*object*)
返回定义了这个对象的 Python 源文件名。如果该对象是一个内置模块、类或函数则会失败并引发一个*TypeError*。

inspect.**getsourcelines**(*object*)
返回一个源代码行的列表和对象的起始行。实参可以是一个模块、类、方法、函数、回溯、帧或者代码对象。与该对象相关的源代码会按行构成一个列表，并且会有一个行号表示其中第一行代码出现在源文件的第几行。如果源代码不能被获取，则会引发一个*OSError*。

3.3 版更變: 现在会引发*OSError* 而不是*IOError*，后者现在是前者的一个别名。

inspect.**getsource**(*object*)
返回对象的源代码文本。实参可以是一个模块、类、方法、函数、回溯、帧或者代码对象。源代码会作为单一一个字符串被返回。如果源代码不能被获取，则会引发一个*OSError*。

3.3 版更變: 现在会引发*OSError* 而不是*IOError*，后者现在是前者的一个别名。

inspect.**cleandoc**(*doc*)
清理文档字符串中为对齐当前代码块进行的缩进

第一行的所有前缀空白符会被移除。从第二行开始所有可以被统一去除的空白符也会被去除。之后，首尾的空白行也会被移除。同时，所有制表符会被展开到空格。

29.13.3 使用 Signature 对象对可调用对象进行内省

3.3 版新加入.

Signature 对象代表了一个可调用对象的调用签名和返回值标注。要获取一个 Signature 对象，使用*signature()* 函数。

inspect.**signature**(*callable*, *, *follow_wrapped=True*)
返回给定的 callable 的一个*Signature* 对象:

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)
```

(下页继续)

(繼續上一頁)

```
>>> str(sig)
'(a, *, b:int, **kwargs)'

>>> str(sig.parameters['b'])
'b:int'

>>> sig.parameters['b'].annotation
<class 'int'>
```

接受各类的 Python 可调用对象，包括单纯的函数、类，到*functools.partial()* 对象。

Raises *ValueError* if no signature can be provided, and *TypeError* if that type of object is not supported.

函数签名中的斜杠（/）表示在它之前的参数是仅限位置的。详见 编程常见问题中关于仅限位置参数的条目

3.5 版新加入: `follow_wrapped` 形参。传递 `False` 来获得特定关于 `callable` 的签名（`callable.__wrapped__` 将不会用来解包被修饰的可调用对象）。

備🄵: 一些可调用对象可能在特定 Python 实现中无法被内省。例如，在 CPython 中，部分通过 C 语言定义的内置函数不提供关于其参数的元数据。

---

class inspect.**Signature**(*parameters=None*, *, *return_annotation=Signature.empty*)

一个 Signature 对象代表了一个函数的调用签名和返回值标注。对于函数接受的每个参数，它对应地在自身的*parameters* 容器中存储一个*Parameter* 对象。

可选参数 *parameters* 是一个*Parameter* 对象组成的序列，它会在之后被验证不存在名字重复的参数，并且参数处于正确的顺序，即仅限位置参数最前，之后紧接着可位置可关键字参数，并且有默认值参数在无默认值参数之前。

可选参数 *return_annotation* 可以是任意 Python 对象，是该可调用对象中"return" 的标注。

Signature 对象是 不可变的。使用*Signature.replace()* 来构造一个修改后的副本。

3.5 版更變: Signature 对象既可封存又可哈希。

empty

该类的一个特殊标记来明确指出返回值标注缺失。

parameters

一个参数名字到对应*Parameter* 对象的有序映射。参数以严格的定义顺序出现，包括仅关键字参数。

3.7 版更變: Python 从 3.7 版起才显式地保证了它保持仅关键字参数的定义顺序，尽管实践上在 Python 3 中一直保持了这个顺序。

return_annotation

可调用对象的“返回值”标注。如果可调用对象没有“返回值”标注，这个属性会被设置为*Signature.empty*。

bind(**args*, ***kwargs*)

构造一个位置和关键字实参到形参的映射。如果 `*args` 和 `**kwargs` 符合签名，则返回一个*BoundArguments*；否则引发一个*TypeError* 。

bind_partial(**args*, ***kwargs*)

与*Signature.bind()* 作用方式相同，但允许省略部分必要的参数（模仿*functools.partial()* 的行为）。返回*BoundArguments*，或者在传入参数不符合签名的情况下，引发一个*TypeError*。

replace(**[, parameters][, return_annotation]*)
基于 replace 函数被调用的目标，创建一个新的 Signature 实例。可以传递不同的 parameters 和/或 return_annotation 来覆盖原本签名的对应的属性。要从复制的 Signature 中移除 return_annotation，可以传入*Signature.empty* 。

```
>>> def test(a, b):
...     pass
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```

classmethod from_callable(*obj*, *, *follow_wrapped=True*)
Return a *Signature* (or its subclass) object for a given callable obj. Pass follow_wrapped=False to get a signature of obj without unwrapping its __wrapped__ chain.

该函数简化了创建*Signature* 的子类的过程：

```
class MySignature(Signature):
    pass
sig = MySignature.from_callable(min)
assert isinstance(sig, MySignature)
```

3.5 版新加入.

class inspect.**Parameter**(*name*, *kind*, *, *default=Parameter.empty*, *annotation=Parameter.empty*)
Parameter 对象是 不可变的。不要直接修改 Parameter 对象，而是通过*Parameter.replace()* 来创建一个修改后的副本。

3.5 版更變：Parameter 对象既可封存（pickle）又可哈希。

empty
该类的一个特殊标记来明确指出默认值和标注的缺失。

name
参数的名字字符串。这个名字必须是一个合法的 Python 标识符。

CPython implementation detail: CPython 会为代码对象构造形如 .0 的隐式形参名，用以实现推导式和生成器表达式。

3.6 版更變: 这些形参名会被此模块暴露为形如 implicit0 一样的名字。

default
该参数的默认值。如果该参数没有默认值，这个参数会被设置为*Parameter.empty* 。

annotation
该参数的标注。如果该参数没有标注，该属性会被设置为*Parameter.empty* 。

kind
描述实参值会如何绑定到形参。可能的取值（可以通过*Parameter* 获得，比如 Parameter.KEYWORD_ONLY ）：

名称	意义
POSITIONAL_ONLY	值必须以位置参数的方式传递。仅限位置参数是在函数定义中出现在 / 之前（如果有）的条目。
POSITIONAL_OR_KEYWORD	值既可以以关键字参数的形式提供，也可以以位置参数的形式提供（这是 Python 写成的函数的标准绑定行为的）。
VAR_POSITIONAL	没有绑定到其他形参的位置实参组成的元组。这对应于 Python 函数定义中的 `*args` 形参。
KEYWORD_ONLY	值必须以关键字实参的形式提供。仅限关键字参数是在 Python 函数定义中出现在 `*` 或 `*args` 之后的条目。
VAR_KEYWORD	一个未绑定到其他形参的关键字参数的字典。这对应于 Python 函数定义中的 `**kwargs` 形参。

例如，打印所有没有默认值的仅关键字参数：

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     if (param.kind == param.KEYWORD_ONLY and
...                       param.default is param.empty):
...         print('Parameter:', param)
Parameter: c
```

`kind.`**`description`**

描述 Parameter.kind 的枚举值。

3.8 版新加入.

例子：打印所有参数的描述：

```
>>> def foo(a, b, *, c, d=10):
...     pass

>>> sig = signature(foo)
>>> for param in sig.parameters.values():
...     print(param.kind.description)
positional or keyword
positional or keyword
keyword-only
keyword-only
```

`replace`(**[, name][, kind][, default][, annotation]*)

基于 replace 函数被调用的目标，创建一个新的 Parameter 实例。要覆写*`Parameter`* 的属性，传递对应的参数。要移除 Parameter 的默认值和/或标注，传递*`Parameter.empty`*。

```
>>> from inspect import Parameter
>>> param = Parameter('foo', Parameter.KEYWORD_ONLY, default=42)
>>> str(param)
'foo=42'

>>> str(param.replace()) # Will create a shallow copy of 'param'
'foo=42'

>>> str(param.replace(default=Parameter.empty, annotation='spam'))
"foo:'spam'"
```

3.4 版更變: 在 Python 3.3 中，Parameter 对象当 `kind` 为 `POSITIONAL_ONLY` 时允许 `name` 被设置为 `None`。这现在已不再被允许。

class inspect.**BoundArguments**

调用*Signature.bind()* 或*Signature.bind_partial()* 的结果。容纳实参到函数的形参的映射。

arguments

一个形参名到实参值的可变映射。仅包含显式绑定的参数。对*arguments* 的修改会反映到*args* 和*kwargs* 上。

应当在任何参数处理目的中与*Signature.parameters* 结合使用。

備𝔽: *Signature.bind()* 和*Signature.bind_partial()* 中采用默认值的参数被跳过。然而，如果有需要的话，可以使用*BoundArguments.apply_defaults()* 来添加它们。

3.9 版更變: *arguments* 现在的类型是*dict*。之前，它的类型是*collections.OrderedDict*。

args

位置参数的值的元组。由*arguments* 属性动态计算。

kwargs

关键字参数值的字典。由*arguments* 属性动态计算。

signature

向父*Signature* 对象的一个引用。

apply_defaults()

设置缺失的参数的默认值。

对于变长位置参数（*args），默认值是一个空元组。

对于变长关键字参数（**kwargs）默认值是一个空字典。

```
>>> def foo(a, b='ham', *args): pass
>>> ba = inspect.signature(foo).bind('spam')
>>> ba.apply_defaults()
>>> ba.arguments
{'a': 'spam', 'b': 'ham', 'args': ()}
```

3.5 版新加入.

args 和*kwargs* 属性可以被用于调用函数:

```
def test(a, *, b):
    ...

sig = signature(test)
ba = sig.bind(10, b=20)
test(*ba.args, **ba.kwargs)
```

也参考:

PEP 362 - 函数签名对象。 包含具体的规范，实现细节和样例。

29.13.4 类与函数

inspect.**getclasstree**(*classes*, *unique=False*)
将给定的类的列表组织成嵌套列表的层级结构。每当一个内层列表出现时，它包含的类均派生自紧接着该列表之前的条目的类。每个条目均是一个二元组，包含一个类和它的基类组成的元组。如果 *unique* 参数为真值，则给定列表中的每个类将恰有一个对应条目。否则，运用了多重继承的类和它们的后代将出现多次。

inspect.**getargspec**(*func*)
获取一个 Python 函数的形参的名字和默认值。将返回一个具名元组 `ArgSpec(args, varargs, keywords, defaults)`。*args* 是一个形参名字的列表。*varargs* 和 *keywords* 是 `*` 和 `**` 形参的名字或 `None`。*defaults* 是一个默认参数值的元组或没有默认值时取 `None`。如果该元组有 *n* 个元素，则它们对应 *args* 中的最后 *n* 个元素。

3.0 版後已🄴用: 通常可以替换为使用更新后的 API *getfullargspec()*，后者还能正确处理函数标注和仅限关键字形参。

又或者，使用*signature()* 和*Signature* 对象，这提供一个更加结构化的处理可调用对象的内省 API。

inspect.**getfullargspec**(*func*)
获取一个 Python 函数的形参的名字和默认值。将返回一个具名元组：

```
FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults,
annotations)
```

args 是一个位置参数的名字的列表。*varargs* 是 `*` 形参的名字或 `None` 表示不接受任意长位置参数时。*varkw* 是 `**` 参数的名字，或 `None` 表示不接受任意关键字参数。*defaults* 是一个包含了默认参数值的 *n* 元组分别对应最后 *n* 个位置参数，或 `None` 则表示没有默认值。*kwonlyargs* 是一个仅关键词参数列表，保持定义时的顺序。*kwonlydefaults* 是一个字典映射自 *kwonlyargs* 中包含的形参名。*annotations* 是一个字典，包含形参值到标注的映射。其中包含一个特殊的键 `"return"` 代表函数返回值的标注（如果有的话）。

注意：*signature()* 和*Signature* 对象 提供可调用对象内省更推荐的 API，并且支持扩展模块 API 中可能出现的额外的行为（比如仅限位置参数）。该函数被保留的主要原因是保持兼容 Python 2 的 `inspect` 模块 API。

3.4 版更變: 该函数现在基于*signature()* 但仍然忽略 `__wrapped__` 属性，并且在签名中包含绑定方法中的第一个绑定参数。

3.6 版更變: 该方法在 Python 3.5 中曾因*signature()* 被文档归为弃用。但该决定已被推翻以恢复一个明确受支持的标准接口，以便运用一份源码通用 Python 2/3 间遗留的*getargspec()* API 的迁移。

3.7 版更變: Python 从 3.7 版起才显式地保证了它保持仅关键字参数的定义顺序，尽管实践上在 Python 3 中一直保持了这个顺序。

inspect.**getargvalues**(*frame*)
获取传入特定的帧的实参的信息。将返回一个具名元组 `ArgInfo(args, varargs, keywords, locals)`。*args* 是一个参数名字的列表。*varargs* 和 *keyword* 是 `*` 和 `**` 参数的名字或 `None`。*locals* 是给定的帧的局部环境字典。

備🄴: 该函数因疏忽在 Python 3.5 中被错误地标记为弃用。

---

inspect.**formatargspec**(*args*[, *varargs*, *varkw*, *defaults*, *kwonlyargs*, *kwonlydefaults*, *annotations*[, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*, *formatreturns*, *formatannotations*]])
将*getfullargspec()* 的返回值格式化为美观的参数规格。

最初的七个参数是 `(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations)`。

其他 6 个参数分别是会被调用以转化参数名、* 参数名、** 参数名、默认值、返回标注和独立标注为字符串的函数。

例如:

```
>>> from inspect import formatargspec, getfullargspec
>>> def f(a: int, b: float):
...     pass
...
>>> formatargspec(*getfullargspec(f))
'(a: int, b: float)'
```

3.5 版後已𝔽用: 使用*signature()* 和*Signature 对象*，它为可调用对象提供一个更好的内省 API。

inspect.**formatargvalues**(*args*[, *varargs*, *varkw*, *locals*, *formatarg*, *formatvarargs*, *formatvarkw*, *formatvalue*])

将*getargvalues()* 返回的四个值格式化为美观的参数规格。format* 的参数是对应的可选格式化函数以转化名字和值为字符串。

備𝔽: 该函数因疏忽在 Python 3.5 中被错误地标记为弃用。

---

inspect.**getmro**(*cls*)

返回由类 cls 的全部基类按方法解析顺序组成的元组，包括 cls 本身。所有类不会在此元组中出现多于一次。注意方法解析顺序取决于 cls 的类型。除非使用一个非常奇怪的用户定义元类型，否则 cls 会是元组的第一个元素。

inspect.**getcallargs**(*func*, */*, **args*, ***kwds*)

仿照调用方式绑定 *args* 和 *kwds* 到 Python 函数或方法 *func* 参数名。对于绑定方法，也绑定第一个参数（通常命名为 `self`）到关联的实例。返回一个字典，映射自参数名（包括可能存在的 * 和 ** 参数）到他们对应于 *args* 和 *kwds* 中的值。假使 *func* 被错误地调用，即是说 `func(*args, **kwds)` 会因函数签名不一致引发一个异常，那么也会引发一个相同种类的异常，并附上相同或类似的消息。例如:

```
>>> from inspect import getcallargs
>>> def f(a, b=1, *pos, **named):
...     pass
>>> getcallargs(f, 1, 2, 3) == {'a': 1, 'named': {}, 'b': 2, 'pos': (3,)}
True
>>> getcallargs(f, a=2, x=4) == {'a': 2, 'named': {'x': 4}, 'b': 1, 'pos': ()}
True
>>> getcallargs(f)
Traceback (most recent call last):
...
TypeError: f() missing 1 required positional argument: 'a'
```

3.2 版新加入.

3.5 版後已𝔽用: 改使用*Signature.bind()* 和*Signature.bind_partial()*。

inspect.**getclosurevars**(*func*)

获取自 Python 函数或方法 *func* 引用的外部名字到它们的值的映射。返回一个*具名元组* `ClosureVars(nonlocals, globals, builtins, unbound)`。*nonlocals* 映射引用的名字到词法闭包变量，*globals* 映射到函数的模块级全局，*builtins* 映射到函数体内可见的内置变量。*unbound* 是在函数中引用但不能解析到给定的模块全局和内置变量的名字的集合。

如果 *func* 不是 Python 函数或方法，将引发*TypeError*。

3.3 版新加入.

inspect.**unwrap**(*func*, *, *stop=None*)
获取 *func* 所包装的对象。它追踪 __wrapped__ 属性链并返回最后一个对象。

stop 是一个可选的回调，接受包装链的一个对象作为唯一参数，以允许通过让回调返回真值使解包装更早中止。如果回调不曾返回一个真值，将如常返回链中的最后一个对象。例如，*signature()* 使用该参数来在遇到具有 __signature__ 参数的对象时停止解包装。

如果遇到循环，则引发*ValueError*。

3.4 版新加入.

29.13.5 解释器栈

当下列函数返回“帧记录”时，每个记录是一个具名元组 FrameInfo(frame, filename, lineno, function, code_context, index)。该元组包含帧对象、文件名、当前行的行号、函数名、来源于源代码的若干行上下文组成的列表和当前行在该列表中的索引。

3.5 版更變: 不再返回一个元组而是返回一个具名元组。

備🄴: 保留帧对象的引用（可见于这些函数返回的帧记录的第一个元素）会导致你的程序产生循环引用。每当一个循环引用被创建，所有可从产生循环的对象访问的对象的生命周期将会被大幅度延长，即便 Python 的可选的循环检测器被启用。如果这类循环必须被创建，确保它们会被显式地打破以避免对象销毁被延迟从而导致占用内存增加。

尽管循环检测器能够处理这种情况，这些帧（包括其局部变量）的销毁可以通过在 finally 子句中移除循环来产生确定的行为。对于循环检测器在编译 Python 时被禁用或者使用*gc.disable()* 时，这样处理更加尤为重要。比如：

```
def handle_stackframe_without_leak():
    frame = inspect.currentframe()
    try:
        # do something with the frame
    finally:
        del frame
```

如果你希望保持帧更长的时间（比如在之后打印回溯），你也可以通过 frame.clear() 方法打破循环引用。

大部分这些函数支持的可选的 *context* 参数指定返回时包含的上下文的行数，以当前行为中心。

inspect.**getframeinfo**(*frame*, *context=1*)
获取一个帧或者一个回溯对象的信息。返回一个具名元组 Traceback(filename, lineno, function, code_context, index)。

inspect.**getouterframes**(*frame*, *context=1*)
获取某帧及其所有外部帧的帧记录的列表。这些帧表示了导致 *frame* 被创建的一系列调用。返回的列表中的第一个记录代表了 *frame* 本身；最后一个记录代表了 *frame* 的栈上的最外层的调用。

3.5 版更變: 返回一个具名元组 FrameInfo(frame, filename, lineno, function, code_context, index) 的列表。

inspect.**getinnerframes**(*traceback*, *context=1*)
获取一个回溯所在的帧和它所有的内部的帧的列表。这些帧代表了作为 *frame* 的后续的帧。第一条记录代表了 *traceback* 本身；最后一条记录代表了引发异常的位置。

3.5 版更變: 返回一个具名元组 FrameInfo(frame, filename, lineno, function, code_context, index) 的列表。

inspect.**currentframe**()
返回调用者的栈帧对应的帧对象。

CPython implementation detail: 该函数依赖于 Python 解释器对于栈帧的支持，这并非在 Python 的所有实现中被保证。该函数在不支持 Python 栈帧的实现中运行会返回 None。

inspect.**stack**(*context=1*)
返回调用者的栈的帧记录列表。第一个记录代表调用者，最后一个记录代表了栈上最外层的调用。

3.5 版更變: 返回一个具名元组 FrameInfo(frame, filename, lineno, function, code_context, index) 的列表。

inspect.**trace**(*context=1*)
返回介于当前帧和引发了当前正在处理的异常的帧之间的所有帧记录的列表。列表中的第一条记录代表调用者；最后一条记录代表了引发异常的地方。

3.5 版更變: 返回一个具名元组 FrameInfo(frame, filename, lineno, function, code_context, index) 的列表。

29.13.6 静态地获取属性

getattr() 和*hasattr()* 都可能会在获取或者判断属性是否存在时触发代码执行。描述符，就和特性一样，会被调用，__getattr__() 和 __getattribute__() 可能会被调用。

对于你想要静态地内省的情况，比如文档工具，这会显得不方便。*getattr_static()* 拥有与*getattr()* 相同的签名，但避免了获取属性时执行代码。

inspect.**getattr_static**(*obj*, *attr*, *default=None*)
获取属性而不触发描述器协议的动态查找能力 __getattr__() 或 __getattribute__()。

注意：该函数可能无法获取 getattr 能获取的全部的属性（比如动态地创建的属性），并且可能发现一些 getattr 无法找到的属性（比如描述器会引发 AttributeError）。它也能够返回描述器对象本身而非实例成员。

如果实例的*__dict__* 被其他成员遮盖（比如一个特性）则该函数无法找到实例成员。

3.2 版新加入.

getattr_static() 不解析描述器。比如槽描述器或 C 语言中实现的 getset 描述器。该描述器对象会被直接返回，而不处理底层属性。

你可以用类似下方的代码的方法处理此事。注意，对于任意 getset 描述符，使用这段代码仍可能触发代码执行。

```
# example code for resolving the builtin descriptor types
class _foo:
    __slots__ = ['foo']

slot_descriptor = type(_foo.foo)
getset_descriptor = type(type(open(__file__)).name)
wrapper_descriptor = type(str.__dict__['__add__'])
descriptor_types = (slot_descriptor, getset_descriptor, wrapper_descriptor)

result = getattr_static(some_object, 'foo')
if type(result) in descriptor_types:
    try:
        result = result.__get__()
    except AttributeError:
        # descriptors can raise AttributeError to
        # indicate there is no underlying value
```

(下页继续)

(繼續上一頁)

```
        # in which case the descriptor itself will
        # have to do
        pass
```

29.13.7 生成器和协程的当前状态

当实现协程调度器或其他更高级的生成器用途时，判断一个生成器是正在执行、等待启动或继续或执行，又或者已经被终止是非常有用的。*getgeneratorstate()* 允许方便地判断一个生成器的当前状态。

inspect.**getgeneratorstate**(*generator*)

获取生成器迭代器的当前状态。

可能的状态是：

- GEN_CREATED：等待开始执行。
- GEN_RUNNING：正在被解释器执行。
- GEN_SUSPENDED：当前挂起于一个 yield 表达式。
- GEN_CLOSED：执行已经完成。

3.2 版新加入.

inspect.**getcoroutinestate**(*coroutine*)

获取协程对象的当前状态。该函数设计为用于使用 `async def` 函数创建的协程函数，但也能接受任何包括 `cr_running` 和 `cr_frame` 的类似协程的对象。

可能的状态是：

- CORO_CREATED：等待开始执行。
- CORO_RUNNING：当前正在被解释器执行。
- CORO_SUSPENDED：当前挂起于一个 await 表达式。
- CORO_CLOSED：执行已经完成。

3.5 版新加入.

生成器当前的内部状态也可以被查询。这通常在测试目的中最为有用，来保证内部状态如预期一样被更新：

inspect.**getgeneratorlocals**(*generator*)

获取 *generator* 里的实时局部变量到当前值的映射。返回一个由名字映射到值的字典。这与在生成器的主体内调用*locals()* 是等效的，并且相同的警告也适用。

如果 *generator* 是一个没有关联帧的生成器，则返回一个空字典。如果 *generator* 不是一个 Python 生成器对象，则引发*TypeError*。

CPython implementation detail: 该函数依赖于生成器为内省暴露一个 Python 栈帧，这并非在 Python 的所有实现中被保证。在这种情况下，该函数将永远返回一个空字典。

3.3 版新加入.

inspect.**getcoroutinelocals**(*coroutine*)

该函数可类比于*getgeneratorlocals()*，只是作用于由 `async def` 函数创建的协程。

3.5 版新加入.

29.13.8 代码对象位标志

Python 代码对象有一个 `co_flags` 属性，它是下列标志的位图。

`inspect.`**`CO_OPTIMIZED`**
代码对象已经经过优化，会采用快速局部变量。

`inspect.`**`CO_NEWLOCALS`**
如果被置位，当代码对象被执行时会创建一个新的字典作为帧的 `f_locals`。

`inspect.`**`CO_VARARGS`**
代码对象拥有一个变长位置形参（类似 `*args`）。

`inspect.`**`CO_VARKEYWORDS`**
代码对象拥有一个可变关键字形惨死（类似 `**kwrags`）。

`inspect.`**`CO_NESTED`**
该标志当代码对象是一个嵌套函数时被置位。

`inspect.`**`CO_GENERATOR`**
当代码对象是一个生成器函数，即调用时会返回一个生成器对象，则该标志被置位。

`inspect.`**`CO_NOFREE`**
当没有自由或单元变量时，标志被置位。

`inspect.`**`CO_COROUTINE`**
当代码对象是一个协程函数时被置位。当代码对象被执行时它返回一个协程。详见 **PEP 492**。

3.5 版新加入.

`inspect.`**`CO_ITERABLE_COROUTINE`**
该标志被用于将生成器转变为基于生成器的协程。包含此标志的生成器对象可以被用于 `await` 表达式，并可以 `yield from` 协程对象。详见 **PEP 492**。

3.5 版新加入.

`inspect.`**`CO_ASYNC_GENERATOR`**
当代码对象是一个异步生成器函数时该标志被置位。当代码对象被运行时它将返回一个异步生成器对象。详见 **PEP 525**。

3.6 版新加入.

備🄵： 这些标志特指于 CPython，并且在其他 Python 实现中可能从未被定义。更进一步地说，这些标志是一种实现细节，并且可能在将来的 Python 发行中被移除或弃用。推荐使用 *inspect* 模块的公共 API 来进行任何内省需求。

29.13.9 命令行界面

inspect 模块也提供一个从命令行使用基本的内省能力。

默认地，命令行接受一个模块的名字并打印模块的源代码。也可通过后缀一个冒号和目标对象的限定名称来打印一个类或者一个函数。

`--details`
打印特定对象的信息而非源码。

29.14 site ——指定域的配置钩子

源代码: Lib/site.py

这个模块将在初始化时被自动导入。此自动导入可以通过使用解释器的 -S 选项来屏蔽。

导入此模块将会附加域特定的路径到模块搜索路径并且添加一些内建对象，除非使用了 -S 选项。那样的话，模块可以被安全地导入，而不会对模块搜索路径和内建对象有自动的修改或添加。要明确地触发通常域特定的添加，调用函数*site.main()*。

3.3 版更變: 在之前即便使用了 -S，导入此模块仍然会触发路径操纵。

It starts by constructing up to four directories from a head and a tail part. For the head part, it uses sys.prefix and sys.exec_prefix; empty heads are skipped. For the tail part, it uses the empty string and then lib/site-packages (on Windows) or lib/python*X.Y*/site-packages (on Unix and macOS). For each of the distinct head-tail combinations, it sees if it refers to an existing directory, and if so, adds it to sys.path and also inspects the newly added path for configuration files.

3.5 版更變: 对"site-python" 目录的支持已被移除。

如果名为"pyvenv.cfg" 的文件存在于 sys.executable 之上的一个目录中，则 sys.prefix 和 sys.exec_prefix 将被设置为该目录，并且还会检查 site-packages （sys.base_prefix 和 sys.base_exec_prefix 始终是 Python 安装的" 真实" 前缀）。如果"pyvenv.cfg"（引导程序配置文件）包含设置为非"true"（不区分大小写）的"include-system-site-packages" 键，则不会在系统级前缀中搜索 site-packages；反之则会。

一个路径配置文件是具有 *name*.pth 命名格式的文件，并且存在上面提到的四个目录之一中；它的内容是要添加到 sys.path 中的额外项目（每行一个）。不存在的项目不会添加到 sys.path，并且不会检查项目指向的是目录还是文件。项目不会被添加到 sys.path 超过一次。空行和由 # 起始的行会被跳过。以 import 开始的行（跟着空格或 TAB）会被执行。

備🄵: 每次启动 Python，在 .pth 文件中的可执行行都将会被运行，而不管特定的模块实际上是否需要被使用。因此，其影响应降至最低。可执行行的主要预期目的是使相关模块可导入（加载第三方导入钩子，调整 PATH 等）。如果它发生了，任何其他的初始化都应当在模块实际导入之前完成。将代码块限制为一行是一种有意采取的措施，不鼓励在此处放置更复杂的内容。

例如，假设 sys.prefix 和 sys.exec_prefix 已经被设置为 /usr/local。Python X.Y 的库之后被安装为 /usr/local/lib/python*X.Y*。假设有一个拥有三个孙目录 foo, bar 和 spam 的子目录 /usr/local/lib/python*X.Y*/site-packages，并且有两个路径配置文件 foot.pth 和 bar.pth。假定 foo.pth 内容如下:

```
# foo package configuration

foo
bar
bletch
```

并且 bar.pth 包含:

```
# bar package configuration

bar
```

则下面特定版目录将以如下顺序被添加到 sys.path。

```
/usr/local/lib/pythonX.Y/site-packages/bar
/usr/local/lib/pythonX.Y/site-packages/foo
```

请注意 bletch 已被省略因为它并不存在；bar 目前在 foo 目录之前因为 bar.pth 按字母顺序排在 foo.pth 之前；而 spam 已被省略因为它在两个路径配置文件中都未被提及。

在这些路径操作之后，会尝试导入一个名为 sitecustomize 的模块，它可以执行任意站点专属的定制。它通常是由系统管理员在 site-packages 目录下创建的。如果此导入失败并引发*ImportError* 或其子类异常，并且异常的 name 属性等于 'sitecustomize'，则它会被静默地忽略。如果 Python 是在没有可用输出流的情况下启动的，例如在 Windows 上使用 pythonw.exe (它默认被用于启动 start IDLE)，则来自 sitecustomize 的输出尝试会被忽略。任何其他异常都会导致静默且可能令人迷惑不解的进程失败。

在此之后，会尝试导入一个名为 usercustomize 的模块，它可以执行任意用户专属的定制，如果*ENABLE_USER_SITE* 为真值的话。这个文件应该在用户的 site-packages 目录中创建（见下文），该目录是 sys.path 的组成部分，除非被 -s 所禁用。如果此导入失败并引发*ImportError* 或者其子类异常，并且异常的 name 属性等于 'usercustomize'，它会被静默地忽略。

请注意对于某些非 Unix 系统来说，sys.prefix 和 sys.exec_prefix 均为空值，并且路径操作会被跳过；但是仍然会尝试导入 sitecustomize 和 usercustomize。

29.14.1 Readline 配置

在支持*readline* 的系统上，这个模块也将导入并配置*rlcompleter* 模块，如果 Python 是以 交互模式启动并且不带 -S 选项的话。默认的行为是启用 tab 键补全并使用 ~/.python_history 作为历史存档文件。要禁用它，请删除（或重载）你的 sitecustomize 或 usercustomize 模块或 PYTHONSTARTUP 文件中的*sys.__interactivehook__* 属性。

3.4 版更變: rlcompleter 和 history 会被自动激活。

29.14.2 模块内容

site.**PREFIXES**
site-packages 目录的前缀列表。

site.**ENABLE_USER_SITE**
显示用户 site-packages 目录状态的旗标。True 意味着它被启用并被添加到 sys.path。False 意味着它按照用户请求被禁用 (通过 -s 或 PYTHONNOUSERSITE)。None 意味着它因安全理由（user 或 group id 和 effective id 之间不匹配）或是被管理员所禁用。

site.**USER_SITE**
Path to the user site-packages for the running Python. Can be None if *getusersitepackages()* hasn't been called yet. Default value is ~/.local/lib/pythonX.Y/site-packages for UNIX and non-framework macOS builds, ~/Library/Python/X.Y/lib/python/site-packages for macOS framework builds, and %APPDATA%\Python\PythonXY\site-packages on Windows. This directory is a site directory, which means that .pth files in it will be processed.

site.**USER_BASE**
Path to the base directory for the user site-packages. Can be None if *getuserbase()* hasn't been called yet. Default value is ~/.local for UNIX and macOS non-framework builds, ~/Library/Python/X.Y for macOS framework builds, and %APPDATA%\Python for Windows. This value is used by Distutils to compute the installation directories for scripts, data files, Python modules, etc. for the user installation scheme. See also PYTHONUSERBASE.

site.**main**()
将所有的标准站点专属目录添加到模块搜索路径。这个函数会在导入此模块时被自动调用，除非 Python 解释器启动时附带了 -S 旗标。

3.3 版更變: 这个函数使用无条件调用。

site.**addsitedir**(*sitedir*, *known_paths=None*)
将一个目录添加到 sys.path 并处理其 .pth 文件。通常被用于 sitecustomize 或 usercustomize (见下文)。

site.**getsitepackages**()
返回包含所有全局 site-packages 目录的列表。

3.2 版新加入.

site.**getuserbase**()
返回用户基准目录的路径*USER_BASE*。如果它尚未被初始化，则此函数还将参照 PYTHONUSERBASE 来设置它。

3.2 版新加入.

site.**getusersitepackages**()
返回用户专属 site-packages 目录的路径*USER_SITE*。如果它尚未被初始化，则此函数还将参照*USER_BASE* 来设置它。要确定用户专属 site-packages 是否已被添加到 sys.path 则应当使用*ENABLE_USER_SITE*。

3.2 版新加入.

29.14.3 命令行界面

site 模块还提供了一个从命令行获取用户目录的方式:

```
$ python3 -m site --user-site
/home/user/.local/lib/python3.3/site-packages
```

如果它被不带参数地调用，它将在标准输出打印*sys.path* 的内容，再打印*USER_BASE* 的值以及该目录是否存在，然后打印*USER_SITE* 的相应信息，最后打印*ENABLE_USER_SITE* 的值。

--user-base
输出用户基本的路径。

--user-site
输出用户 site-packages 目录的路径。

如果同时给出了两个选项，则将打印用户基准目录和用户站点信息（总是按此顺序），并以*os.pathsep* 分隔。

如果给出了其中一个选项，脚本将退出并返回以下值中的一个：如果用户级 site-packages 目录被启用则为 0，如果它被用户禁用则为 1，如果它因安全理由或被管理员禁用则为 2，如果发生错误则为大于 2 的值。

也参考:

PEP 370 -- 分用户的 site-packages 目录

CHAPTER 30

自定义 Python 解释器

本章中描述的模块允许编写类似于 Python 的交互式解释器的接口。如果你想要一个支持附加一些特殊功能到 Python 语言的 Python 解释器，你应该看看 *code* 模块。（*codeop* 模块是低层级的，用于支持编译可能不完整的 Python 代码块。）

本章描述的完整模块列表如下：

30.1 code --- 解释器基类

源代码： Lib/code.py

code 模块提供了在 Python 中实现 read-eval-print 循环的功能。它包含两个类和一些快捷功能，可用于构建提供交互式解释器的应用程序。

class code.**InteractiveInterpreter**(*locals=None*)
这个类处理解析器和解释器状态（用户命名空间的）；它不处理缓冲器、终端提示区或着输入文件名（文件名总是显示地传递）。可选的 *locals* 参数指定一个字典，字典里面包含将在此类执行的代码；它默认创建新的字典，其键 '__name__' 设置为 '__console__'，键 '__doc__' 设置为 None 。

class code.**InteractiveConsole**(*locals=None*, *filename="<console>"*)
尽可能模拟交互式 Python 解释器的行为。此类建立在 *InteractiveInterpreter* 的基础上，使用熟悉的 sys.ps1 和 sys.ps2 作为输入提示符，并有输入缓冲。

code.**interact**(*banner=None*, *readfunc=None*, *local=None*, *exitmsg=None*)
运行一个 read-eval-print 循环的便捷函数。这会创建一个新的 *InteractiveConsole* 实例。如果提供了 *readfunc*，会设置为 *InteractiveConsole.raw_input()* 方法。如果提供了 *local*，则将其传递给 *InteractiveConsole* 的构造函数，以用作解释器循环的默认命名空间。然后，如果提供了 *banner* 和 *exitmsg*，实例的 *interact()* 方法会以此为标题和退出消息。控制台对象在使用后将被丢弃。

3.6 版更變: 加入 *exitmsg* 参数。

code.**compile_command**(*source*, *filename="<input>"*, *symbol="single"*)
这个函数主要用来模拟 Python 解释器的主循环（即 read-eval-print 循环）。难点的部分是当用户输入不

完整命令时，判断能否通过之后的输入来完成（要么成为完整的命令，要么语法错误）。该函数 几乎和实际的解释器主循环的判断是相同的。

source 是源字符串；*filename* 是可选的用作读取源的文件名，默认为 `'<input>'`；*symbol* 是可选的语法开启符号，应为 `'single'` (默认), `'eval'` 或 `'exec'`。

如果命令完整且有效则返回一个代码对象(等价于 `compile(source, filename, symbol)`)；如果命令不完整则返回 `None`；如果命令完整但包含语法错误则会引发*SyntaxError* 或*OverflowError* 而如果命令包含无效字面值则将引发*ValueError*。

30.1.1 交互解释器对象

`InteractiveInterpreter.`**`runsource`**(*source*, *filename="<input>"*, *symbol="single"*)

在解释器中编译并运行一段源码。所用参数与*compile_command()* 一样；*filename* 的默认值为 `'<input>'`，*symbol* 则为 `'single'`。可能发生以下情况之一：

- 输入不正确；*compile_command()* 引发了一个异常 (*SyntaxError* 或*OverflowError*)。将通过调用*showsyntaxerror()* 方法打印语法回溯信息。*runsource()* 返回 `False`。
- 输入不完整，需要更多输入；函数*compile_command()* 返回 `None` 。方法*runsource()* 返回 `True` 。
- 输入完整；*compile_command()* 返回了一个代码对象。将通过调用*runcode()* 执行代码（该方法也会处理运行时异常，*SystemExit* 除外)。*runsource()* 返回 `False`。

该返回值用于决定使用 `sys.ps1` 还是 `sys.ps2` 来作为下一行的输入提示符。

`InteractiveInterpreter.`**`runcode`**(*code*)

执行一个代码对象。当发生异常时，调用*showtraceback()* 来显示回溯。除*SystemExit* （允许传播）以外的所有异常都会被捕获。

有关*KeyboardInterrupt* 的说明，该异常可能发生于此代码的其他位置，并且并不总能被捕获。调用者应当准备好处理它。

`InteractiveInterpreter.`**`showsyntaxerror`**(*filename=None*)

显示刚发生的语法错误。这不会显示堆栈回溯因为语法错误并无此种信息。如果给出了 *filename*，它会被放入异常来替代 Python 解析器所提供的默认文件名，因为它在从一个字符串读取时总是会使用 `'<string>'`。输出将由*write()* 方法来写入。

`InteractiveInterpreter.`**`showtraceback`**()

显示刚发生的异常。我们移除了第一个堆栈条目因为它从属于解释器对象的实现。输出将由*write()* 方法来写入。

3.5 版更變: 将显示完整的链式回溯，而不只是主回溯。

`InteractiveInterpreter.`**`write`**(*data*)

将一个字符串写入到标准错误流 (`sys.stderr`)。所有派生类都应重载此方法以提供必要的正确输出处理。

30.1.2 交互式控制台对象

InteractiveConsole 类是*InteractiveInterpreter* 的子类，因此它提供了解释器对象的所有方法，还有以下的额外方法。

InteractiveConsole.**interact**(*banner=None*, *exitmsg=None*)
近似地模拟交互式 Python 终端。可选的 *banner* 参数指定要在第一次交互前打印的条幅；默认情况下会类似于标准 Python 解释器所打印的内容，并附上外加圆括号的终端对象类名（这样就不会与真正的解释器混淆——因为确实太像了！）

可选的 *exitmsg* 参数指定要在退出时打印的退出消息。传入空字符串可以屏蔽退出消息。如果 *exitmsg* 未给出或为 None，则将打印默认消息。

3.4 版更變: 要禁止打印任何条幅消息，请传递一个空字符串。

3.6 版更變: 退出时打印退出消息。

InteractiveConsole.**push**(*line*)
将一行源文本推入解释器。行内容不应带有末尾换行符；它可以有内部换行符。行内容会被添加到一个缓冲区并且会调用解释器的 runsource() 方法，附带缓冲区内容的拼接结果作为源文本。如果显示命令已执行或不合法，缓冲区将被重置；否则，则命令尚未结束，缓冲区将在添加行后保持原样。如果要求更多输入则返回值为 True，如果行已按某种方式被处理则返回值为 False (这与 runsource() 相同)。

InteractiveConsole.**resetbuffer**()
从输入缓冲区中删除所有未处理的内容。

InteractiveConsole.**raw_input**(*prompt=""*)
输出提示并读取一行。返回的行不包含末尾的换行符。当用户输入 EOF 键序列时，会引发*EOFError* 异常。默认实现是从 sys.stdin 读取；子类可以用其他实现代替。

30.2 codeop --- 编译 Python 代码

源代码： Lib/codeop.py

---

codeop 模块提供了可以模拟 Python 读取-执行-打印循环的实用程序，就像在*code* 模块中一样。因此，您可能不希望直接使用该模块；如果你想在程序中包含这样一个循环，你可能需要使用*code* 模块。

这个任务有两个部分:

1. 能够判断一行输入是否完成了一个 Python 语句：简而言之，告诉我们是否要打印"»>" 或"..."。
2. 记住用户已输入了哪些 future 语句，这样后续的输入可以在这些语句被启用的状态下被编译。

codeop 模块提供了分别以及同时执行这两个部分的方式。

只执行前一部分:

codeop.**compile_command**(*source*, *filename="<input>"*, *symbol="single"*)
尝试编译 *source*，这应当是一个 Python 代码字符串，并且在 *source* 是有效的 Python 代码时返回一个代码对象。在此情况下，代码对象的 filename 属性将为 *filename*，其默认值为 '<input>'。如果 *source* 不是有效的 Python 代码而是有效的 Python 代码的一个前缀时将返回 None。

如果 *source* 存在问题，将引发异常。如果存在无效的 Python 语法将引发*SyntaxError*，而如果存在无效的字面值则将引发*OverflowError* 或*ValueError*。

symbol 参数确定 *source* 是作为一条语句 (对应默认值 'single')，作为一系列语句 ('exec') 还是作为一个*expression* ('eval') 进行编译。任何其他值都将导致引发*ValueError*。

備🄵: 解析器有可能（但很不常见）会在到达源码结尾之前停止解析并成功输出结果；在这种情况下，末尾的符号可能会被忽略而不是引发错误。例如，一个反斜杠加两个换行符之后可以跟随任何无意义的符号。一旦解析器 API 得到改进将修正这个问题。

class codeop.**Compile**
这个类的实例具有 __call__() 方法，其签名与内置函数*compile()* 相似，区别在于如果该实例编译了包含*__future__* 语句的程序文本，则实例会‘记住’并使用已生效的语句编译所有后续程序文本。

class codeop.**CommandCompiler**
这个类的实例具有 __call__() 方法，其签名与*compile_command()* 相似；区别在于如果该实例编译了包含 __future__ 语句的程序文本，则实例会‘记住’并使用已生效的语句编译编译所有后续程序文本。

CHAPTER 31

匯入模組

本章中介绍的模块提供了导入其他 Python 模块和挂钩以自定义导入过程的新方法。

本章描述的完整模块列表如下：

31.1 zipimport --- 从 Zip 存档中导入模块

源代码： Lib/zipimport.py

此模块添加了从 ZIP 格式档案中导入 Python 模块（`*.py` ， `*.pyc` ）和包的能力。通常不需要明确地使用 *zipimport* 模块，内置的 `import` 机制会自动将此模块用于 ZIP 档案路径的 *sys.path* 项目上。

通常，*sys.path* 是字符串的目录名称列表。此模块同样允许 *sys.path* 的一项成为命名 ZIP 文件档案的字符串。ZIP 档案可以容纳子目录结构去支持包的导入，并且可以将归档文件中的路径指定为仅从子目录导入。比如说，路径 `example.zip/lib/` 将只会从档案中的 `lib/` 子目录导入。

Any files may be present in the ZIP archive, but importers are only invoked for `.py` and `.pyc` files. ZIP import of dynamic modules (`.pyd`, `.so`) is disallowed. Note that if an archive only contains `.py` files, Python will not attempt to modify the archive by adding the corresponding `.pyc` file, meaning that if a ZIP archive doesn't contain `.pyc` files, importing may be rather slow.

3.8 版更變: 以前，不支持带有档案注释的 ZIP 档案。

也参考:

PKZIP Application Note Phil Katz 编写的 ZIP 文件格式文档，此格式和使用的算法的创建者。

PEP 273 - 从 ZIP 压缩包导入模块 由 James C. Ahlstrom 编写，他也提供了实现。Python 2.3 遵循 **PEP 273** 的规范，但是使用 Just van Rossum 编写的使用了 **PEP 302** 中描述的导入钩的实现。

PEP 302 - 新导入钩 PEP 添加导入钩来有助于模块运作。

此模块定义了一个异常：

exception zipimport.**ZipImportError**
异常由 zipimporter 对象引发。这是*ImportError* 的子类，因此，也可以捕获为*ImportError* 。

31.1.1 zipimporter 对象

zipimporter 是用于导入 ZIP 文件的类。

class zipimport.**zipimporter**(*archivepath*)
创建新的 zipimporter 实例。*archivepath* 必须是指向 ZIP 文件的路径，或者 ZIP 文件中的特定路径。例如，`foo/bar.zip/lib` 的 *archivepath* 将在 ZIP 文件 `foo/bar.zip` 中的 `lib` 目录中查找模块（只要它存在）。

如果 *archivepath* 没有指向一个有效的 ZIP 档案，引发*ZipImportError* 。

find_module(*fullname*[, *path*])
搜索由 *fullname* 指定的模块。*fullname* 必须是完全合格的（点分的）模块名。它返回 zipimporter 实例本身如果模块被找到，或者返回*None* 如果没找到指定模块。可选的 *path* 被忽略，这是为了与导入器协议兼容。

get_code(*fullname*)
返回指定模块的代码对象。如果不能找到模块会引发*ZipImportError* 错误。

get_data(*pathname*)
返回与 *pathname* 相关联的数据。如果不能找到文件则引发*OSError* 错误。

3.3 版更變: 曾经是*IOError* 被引发而不是*OSError* 。

get_filename(*fullname*)
如果导入了指定的模块 `__file__` ，则返回为该模块设置的值。如果未找到模块则引发*ZipImportError* 错误。

3.1 版新加入.

get_source(*fullname*)
返回指定模块的源代码。如果没有找到模块则引发*ZipImportError* ，如果档案包含模块但是没有源代码，返回*None* 。

is_package(*fullname*)
如果由 *fullname* 指定的模块是一个包则返回 `True` 。如果不能找到模块则引发*ZipImportError* 错误。

load_module(*fullname*)
加载由 *fullname* 指定的模块。*fullname* 必须是完全限定的（点分的）模块名。它返回已加载模块，或者当找不到模块时引发*ZipImportError* 错误。

archive
导入器关联的 ZIP 文件的文件名，没有可能的子路径。

prefix
ZIP 文件中搜索的模块的子路径。这是一个指向 ZIP 文件根目录的 zipimporter 对象的空字符串。

当与斜杠结合使用时，*archive* 和*prefix* 属性等价于赋予*zipimporter* 构造器的原始 *archivepath* 参数。

31.1.2 示例

这是一个从 ZIP 档案中导入模块的例子 - 请注意*zipimport* 模块不需要明确地使用。

```
$ unzip -l example.zip
Archive:  example.zip
  Length      Date    Time    Name
 --------    ----    ----    ----
     8467  11-26-02 22:30   jwzthreading.py
 --------                   -------
     8467                   1 file
$ ./python
Python 2.3 (#1, Aug 1 2003, 19:54:32)
>>> import sys
>>> sys.path.insert(0, 'example.zip')  # Add .zip file to front of path
>>> import jwzthreading
>>> jwzthreading.__file__
'example.zip/jwzthreading.py'
```

31.2 pkgutil --- 包扩展工具

源代码: Lib/pkgutil.py

该模块为导入系统提供了工具，尤其是在包支持方面。

class pkgutil.**ModuleInfo**(*module_finder*, *name*, *ispkg*)
一个包含模块信息的简短摘要的命名元组。

3.6 版新加入.

pkgutil.**extend_path**(*path*, *name*)
Extend the search path for the modules which comprise a package. Intended use is to place the following code in a package's __init__.py:

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

This will add to the package's __path__ all subdirectories of directories on *sys.path* named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

It also looks for *.pkg files beginning where * matches the *name* argument. This feature is similar to *.pth files (see the *site* module for more information), except that it doesn't special-case lines starting with import. A *.pkg file is trusted at face value: apart from checking for duplicates, all entries found in a *.pkg file are added to the path, regardless of whether they exist on the filesystem. (This is a feature.)

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that *sys.path* is a sequence. Items of *sys.path* that are not strings referring to existing directories are ignored. Unicode items on *sys.path* that cause errors when used as filenames may cause this function to raise an exception (in line with *os.path.isdir()* behavior).

class pkgutil.**ImpImporter**(*dirname=None*)
PEP 302 Finder that wraps Python's "classic" import algorithm.

If *dirname* is a string, a **PEP 302** finder is created that searches that directory. If *dirname* is `None`, a **PEP 302** finder is created that searches the current *sys.path*, plus any modules that are frozen or built-in.

Note that *ImpImporter* does not currently support being used by placement on *sys.meta_path*.

3.3 版後已🄵用: This emulation is no longer needed, as the standard import mechanism is now fully **PEP 302** compliant and available in *importlib*.

class pkgutil.**ImpLoader**(*fullname*, *file*, *filename*, *etc*)
Loader that wraps Python's "classic" import algorithm.

3.3 版後已🄵用: This emulation is no longer needed, as the standard import mechanism is now fully **PEP 302** compliant and available in *importlib*.

pkgutil.**find_loader**(*fullname*)
Retrieve a module *loader* for the given *fullname*.

This is a backwards compatibility wrapper around *importlib.util.find_spec()* that converts most failures to *ImportError* and only returns the loader rather than the full `ModuleSpec`.

3.3 版更變: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

3.4 版更變: Updated to be based on **PEP 451**

pkgutil.**get_importer**(*path_item*)
Retrieve a *finder* for the given *path_item*.

The returned finder is cached in *sys.path_importer_cache* if it was newly created by a path hook.

The cache (or part of it) can be cleared manually if a rescan of *sys.path_hooks* is necessary.

3.3 版更變: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

pkgutil.**get_loader**(*module_or_name*)
Get a *loader* object for *module_or_name*.

If the module or package is accessible via the normal import mechanism, a wrapper around the relevant part of that machinery is returned. Returns `None` if the module cannot be found or imported. If the named module is not already imported, its containing package (if any) is imported, in order to establish the package `__path__`.

3.3 版更變: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

3.4 版更變: Updated to be based on **PEP 451**

pkgutil.**iter_importers**(*fullname=''*)
Yield *finder* objects for the given module name.

If fullname contains a `'.'`, the finders will be for the package containing fullname, otherwise they will be all registered top level finders (i.e. those on both *sys.meta_path* and *sys.path_hooks*).

If the named module is in a package, that package is imported as a side effect of invoking this function.

If no module name is specified, all top level finders are produced.

3.3 版更變: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

pkgutil.**iter_modules**(*path=None*, *prefix=''*)
Yields *ModuleInfo* for all submodules on *path*, or, if *path* is `None`, all top-level modules on *sys.path*.

path should be either `None` or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

備F: Only works for a *finder* which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for *importlib.machinery.FileFinder* and *zipimport.zipimporter*.

3.3 版更變: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

pkgutil.**walk_packages**(*path=None*, *prefix=''*, *onerror=None*)
Yields *ModuleInfo* for all modules recursively on *path*, or, if *path* is `None`, all accessible modules.

path should be either `None` or a list of paths to look for modules in.

prefix is a string to output on the front of every module name on output.

Note that this function must import all *packages* (*not* all modules!) on the given *path*, in order to access the `__path__` attribute to find submodules.

onerror is a function which gets called with one argument (the name of the package which was being imported) if any exception occurs while trying to import a package. If no *onerror* function is supplied, *ImportError*s are caught and ignored, while all other exceptions are propagated, terminating the search.

例如:

```
# list all modules python can access
walk_packages()

# list all submodules of ctypes
walk_packages(ctypes.__path__, ctypes.__name__ + '.')
```

備F: Only works for a *finder* which defines an `iter_modules()` method. This interface is non-standard, so the module also provides implementations for *importlib.machinery.FileFinder* and *zipimport.zipimporter*.

3.3 版更變: Updated to be based directly on *importlib* rather than relying on the package internal **PEP 302** import emulation.

pkgutil.**get_data**(*package*, *resource*)
从包中获取一个资源。

This is a wrapper for the *loader* *get_data* API. The *package* argument should be the name of a package, in standard module format (`foo.bar`). The *resource* argument should be in the form of a relative filename, using `/` as the path separator. The parent directory name `..` is not allowed, and nor is a rooted name (starting with a `/`).

The function returns a binary string that is the contents of the specified resource.

For packages located in the filesystem, which have already been imported, this is the rough equivalent of:

```
d = os.path.dirname(sys.modules[package].__file__)
data = open(os.path.join(d, resource), 'rb').read()
```

If the package cannot be located or loaded, or it uses a *loader* which does not support *get_data*, then `None` is returned. In particular, the *loader* for *namespace packages* does not support *get_data*.

pkgutil.**resolve_name**(*name*)
Resolve a name to an object.

This functionality is used in numerous places in the standard library (see bpo-12915) - and equivalent functionality is also in widely used third-party packages such as setuptools, Django and Pyramid.

It is expected that *name* will be a string in one of the following formats, where W is shorthand for a valid Python identifier and dot stands for a literal period in these pseudo-regexes:

- `W(.W)*`
- `W(.W)*:(W(.W)*)?`

The first form is intended for backward compatibility only. It assumes that some part of the dotted name is a package, and the rest is an object somewhere within that package, possibly nested inside other objects. Because the place where the package stops and the object hierarchy starts can't be inferred by inspection, repeated attempts to import must be done with this form.

In the second form, the caller makes the division point clear through the provision of a single colon: the dotted name to the left of the colon is a package to be imported, and the dotted name to the right is the object hierarchy within that package. Only one import is needed in this form. If it ends with the colon, then a module object is returned.

The function will return an object (which might be a module), or raise one of the following exceptions:

ValueError -- if *name* isn't in a recognised format.

ImportError -- if an import failed when it shouldn't have.

AttributeError -- If a failure occurred when traversing the object hierarchy within the imported package to get to the desired object.

3.9 版新加入.

31.3 modulefinder --- 查找脚本使用的模块

源码： Lib/modulefinder.py

该模块提供了一个*ModuleFinder* 类，可用于确定脚本导入的模块集。modulefinder.py 也可以作为脚本运行，给出 Python 脚本的文件名作为参数，之后将打印导入模块的报告。

modulefinder.**AddPackagePath**(*pkg_name*, *path*)
记录名为 *pkg_name* 的包可以在指定的 *path* 中找到。

modulefinder.**ReplacePackage**(*oldname*, *newname*)
允许指定名为 *oldname* 的模块实际上是名为 *newname* 的包。

class modulefinder.**ModuleFinder**(*path=None*, *debug=0*, *excludes=[]*, *replace_paths=[]*)
该类提供*run_script()* 和*report()* 方法，用于确定脚本导入的模块集。*path* 可以是搜索模块的目录列表；如果没有指定，则使用 sys.path 。*debug* 设置调试级别；更高的值使类打印调试消息，关于它正在做什么。*excludes* 是要从分析中排除的模块名称列表。*replace_paths* 是将在模块路径中替换的 (oldpath, newpath) 元组的列表。

report()
将报告打印到标准输出，列出脚本导入的模块及其路径，以及缺少或似乎缺失的模块。

run_script(*pathname*)
分析 *pathname* 文件的内容，该文件必须包含 Python 代码。

modules
一个将模块名称映射到模块的字典。请参阅*ModuleFinder 的示例用法*。

31.3.1 ModuleFinder 的示例用法

稍后将分析的脚本（bacon.py）：

```
import re, itertools

try:
    import baconhameggs
except ImportError:
    pass

try:
    import guido.python.ham
except ImportError:
    pass
```

将输出 bacon.py 报告的脚本:

```
from modulefinder import ModuleFinder

finder = ModuleFinder()
finder.run_script('bacon.py')

print('Loaded modules:')
for name, mod in finder.modules.items():
    print('%s: ' % name, end='')
    print(','.join(list(mod.globalnames.keys())[:3]))

print('-'*50)
print('Modules not imported:')
print('\n'.join(finder.badmodules.keys()))
```

输出样例（可能因架构而异）：

```
Loaded modules:
_types:
copyreg:  _inverted_registry,_slotnames,__all__
sre_compile:  isstring,_sre,_optimize_unicode
_sre:
sre_constants:  REPEAT_ONE,makedict,AT_END_LINE
sys:
re:  __module__,finditer,_expand
itertools:
__main__:  re,itertools,baconhameggs
sre_parse:  _PATTERNENDERS,SRE_FLAG_UNICODE
array:
types:  __module__,IntType,TypeType
--------------------------------------------------
Modules not imported:
guido.python.ham
baconhameggs
```

31.4 runpy ——查找并执行 Python 模块

源代码： Lib/runpy.py

---

runpy 模块用于找到并运行 Python 的模块，而无需首先导入。主要用于实现 -m 命令行开关，以允许用 Python 模块命名空间而不是文件系统来定位脚本。

请注意，这 并非一个沙盒模块——所有代码都在当前进程中运行，所有副作用（如其他模块对导入操作进行了缓存）在函数返回后都会留存。

此外，在*runpy* 函数返回后，任何由已执行代码定义的函数和类都不能保证正确工作。如果某使用场景不能接收此限制，那么选用*importlib* 可能更合适些。

runpy 模块提供两个函数：

runpy.**run_module**(*mod_name*, *init_globals=None*, *run_name=None*, *alter_sys=False*)

执行给定模块的代码，并返回结果模块的 globals 字典。该模块的代码首先会用标准的导入机制去查找定位 (详情请参阅 **PEP 302**)，然后在全新的模块命名空间中运行。

参数 *mod_name* 应该是一个绝对模块名。如果模块名指向一个包，而不是普通的模块，那么该包会被导入，然后执行包中的 __main__ 子模块，并返回结果模块的 globals 字典。

可选的字典参数 *init_globals* 可用来在代码执行前预填充模块的 globals 字典。给出的字典参数不会被修改。如果字典中定义了以下任意一个特殊全局变量，这些定义都会被*run_module()* 覆盖。

The special global variables __name__, __spec__, __file__, __cached__, __loader__ and __package__ are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

若可选参数 __name__ 不为*None* 则设为 *run_name*，若此名称的模块是一个包则设为 mod_name + '.__main__'，否则设为 *mod_name* 参数。

__spec__ 将设为合适的 实际导入模块 (也就是说，__spec__.name 一定是 *mod_name* 或 mod_name + '.__main__'，而不是 *run_name*)。

__file__ 、__cached__、__loader__ 和 __package__ 根据模块规范进行 正常设置

如果给出了参数 *alter_sys* 并且值为*True*，那么 sys.argv[0] 将被更新为 __file__ 的值，sys.modules[__name__] 将被更新为临时模块对象。在函数返回前，sys.argv[0] 和 sys.modules[__name__] 将会复原。

注意，这种对*sys* 的操作不是线程安全的。其他线程可能会看到初始化不完整的模块，以及变动后的参数列表。如果从线程中的代码调用此函数，建议单实例运行*sys* 模块。

也参考：

-m 选项由命令行提供相同功能。

3.1 版更變: 加入了查找 __main__ 子模块并执行软件包的能力。

3.2 版更變: 加入了 __cached__ 全局变量（参见 **PEP 3147** ）。

3.4 版更變: 充分利用 **PEP 451** 加入的模块规格功能。使得以这种方式运行的模块能够正确设置 __cached__，并确保真正的模块名称总是可以通过 __spec__.name 的形式访问。

runpy.**run_path**(*path_name*, *init_globals=None*, *run_name=None*)

执行指定位置的代码，并返回结果模块的 globals 字典。与提供给 CPython 命令行的脚本名称一样，给出的路径可以指向 Python 源文件、编译过的字节码文件或包含 “__main__“ 模块的有效 sys.path 项（例如一个包含顶级 “__main__.py“ 文件的 zip 文件）。

对于直接的脚本而言，指定代码将直接在一个新的模块命名空间中运行。对于一个有效的 sys.path 项（通常是一个 zip 文件或目录），其首先会被添加到 sys.path 的开头。然后，本函数用更新后的路径

查找并执行*__main__* 模块。请注意，即便在指定位置不存在主模块，也没有特别的保护措施来防止调用存在于 sys.path 其他地方的*__main__*。

利用可选的字典参数 *init_globals*，可在代码执行前预填模块的 globals 字典。给出的字典参数不会被修改。如果给出的字典中定义了下列特殊全局变量，这些定义均会被*run_module()* 覆盖。

The special global variables __name__, __spec__, __file__, __cached__, __loader__ and __package__ are set in the globals dictionary before the module code is executed (Note that this is a minimal set of variables - other variables may be set implicitly as an interpreter implementation detail).

如果该可选参数不为*None*，则 __name__ 被设为 *run_name*，否则为 '<run_path>'。

如果提供的路径直接引用了一个脚本文件（无论是源码文件还是预编译的字节码），那么 __file__ 将设为给出的路径，而 __spec__、__cached__、__loader__ 和 __package__ 都将设为*None*。

如果给出的路径是对有效 sys.path 项的引用，那么 __spec__ 将为导入的 __main__ 模块进行正确设置 (也就是说，__spec__.name 将一定是 __main__)。__file__, __cached__, __loader__ 和 __package__ 将依据模块规格 进行常规设置。

A number of alterations are also made to the *sys* module. Firstly, sys.path may be altered as described above. sys.argv[0] is updated with the value of path_name and sys.modules[__name__] is updated with a temporary module object for the module being executed. All modifications to items in *sys* are reverted before the function returns.

注意，与*run_module()* 不同的是，本函数对*sys* 的修改不是可有可无的，因为这些调整对于 sys.path 项能够顺利执行至关重要。由于依然存在线程安全的限制，在线程代码中使用本函数时应采用导入锁进行序列运行，或者委托给一个单独的进程。

也参考:

using-on-interface-options 用于在命令行上实现同等功能（python path/to/script）。

3.2 版新加入.

3.4 版更變: 已作更新，以便充分利用 **PEP 451** 加入的模块规格功能。使得从有效 sys.path 项导入 “__main__“ 而不是直接执行的情况下，能够正确设置 __cached__。

也参考:

PEP 338 -- 将模块作为脚本执行 PEP 由 Nick Coghlan 撰写并实现。

PEP 366 ——主模块的显式相对导入 PEP 由 Nick Coghlan 撰写并实现。

PEP 451 ——导入系统采用的 ModuleSpec 类型 PEP 由 Eric Snow 撰写并实现。

using-on-general ——CPython 命令行详解

importlib.import_module() 函数

31.5 importlib --- import 的实现

3.1 版新加入.

源代码 Lib/importlib/__init__.py

31.5.1 簡介

importlib 包的目的有两个。第一个目的是在 Python 源代码中提供 import 语句的实现（并且因此而扩展*__import__()* 函数）。这提供了一个可移植到任何 Python 解释器的 import 实现。相比使用 Python 以外的编程语言实现方式，这一实现更加易于理解。

第二个目的是实现 import 的部分被公开在这个包中，使得用户更容易创建他们自己的自定义对象 (通常被称为*importer*) 来参与到导入过程中。

也参考:

import import 语句的语言参考

包规格说明 包的初始规范。自从编写这个文档开始，一些语义已经发生改变了（比如基于*sys.modules* 中 None 的重定向）。

***__import__()* 函数** import 语句是这个函数的语法糖。

PEP 235 在忽略大小写的平台上进行导入

PEP 263 定义 Python 源代码编码

PEP 302 新导入钩子

PEP 328 导入：多行和绝对/相对

PEP 366 主模块显式相对导入

PEP 420 隐式命名空间包

PEP 451 导入系统的一个模块规范类型

PEP 488 消除 PYO 文件

PEP 489 多阶段扩展模块初始化

PEP 552 确定性的 pyc 文件

PEP 3120 使用 UTF-8 作为默认的源编码

PEP 3147 PYC 仓库目录

31.5.2 函数

importlib.**__import__**(*name*, *globals=None*, *locals=None*, *fromlist=()*, *level=0*)
内置*__import__()* 函数的实现。

備🄵: 程序式地导入模块应该使用*import_module()* 而不是这个函数。

---

importlib.**import_module**(*name*, *package=None*)
导入一个模块。参数 *name* 指定了以绝对或相对导入方式导入什么模块 (比如要么像这样 pkg.mod 或者这样 ..mod)。如果参数 name 使用相对导入的方式来指定，那么那个参数 *package* 必须设置为那个包名，这个包名作为解析这个包名的锚点 (比如 import_module('..mod', 'pkg.subpkg') 将会导入 pkg.mod)。

import_module() 函数是一个对*importlib.__import__()* 进行简化的包装器。这意味着该函数的所有主义都来自于*importlib.__import__()*。这两个函数之间最重要的不同点在于*import_module()* 返回指定的包或模块 (例如 pkg.mod)，而*__import__()* 返回最高层级的包或模块 (例如 pkg)。

如果动态导入一个自从解释器开始执行以来被创建的模块（即创建了一个 Python 源代码文件），为了让导入系统知道这个新模块，可能需要调用*invalidate_caches()*。

3.3 版更變: 父包会被自动导入。

importlib.**find_loader**(*name*, *path=None*)

查找一个模块的加载器，可选择地在指定的 *path* 里面。如果这个模块是在*sys.modules*，那么返回 sys.modules[name].__loader__（除非这个加载器是 None 或者是没有被设置，在这样的情况下，会引起*ValueError* 异常）。否则使用*sys.meta_path* 的一次搜索就结束。如果未发现加载器，则返回 None。

点状的名称没有使得它父包或模块隐式地导入，因为它需要加载它们并且可能不需要。为了适当地导入一个子模块，需要导入子模块的所有父包并且使用正确的参数提供给 *path*。

3.3 版新加入.

3.4 版更變: 如果没有设置 __loader__，会引起*ValueError* 异常，就像属性设置为 None 的时候一样。

3.4 版後已🄵用: 使用*importlib.util.find_spec()* 来代替。

importlib.**invalidate_caches**()

使查找器存储在*sys.meta_path* 中的内部缓存无效。如果一个查找器实现了 invalidate_caches()，那么它会被调用来执行那个无效过程。如果创建/安装任何模块，同时正在运行的程序是为了保证所有的查找器知道新模块的存在，那么应该调用这个函数。

3.3 版新加入.

importlib.**reload**(*module*)

重新加载之前导入的 *module*。那个参数必须是一个模块对象，所以它之前必须已经成功导入了。这在你已经使用外部编辑器编辑过了那个模块的源代码文件并且想在退出 Python 解释器之前试验这个新版本的模块的时候将很适用。函数的返回值是那个模块对象（如果重新导入导致一个不同的对象放置在*sys.modules* 中，那么那个模块对象是有可能会不同）。

当执行*reload()* 的时候:

- Python 模块的代码会被重新编译并且那个模块级的代码被重新执行，通过重新使用一开始加载那个模块的*loader*，定义一个新的绑定在那个模块字典中的名称的对象集合。扩展模块的 “init“函数不会被调用第二次。
- 与 Python 中的所有的其它对象一样，旧的对象只有在它们的引用计数为 0 之后才会被回收。
- 模块命名空间中的名称重新指向任何新的或更改后的对象。
- 其他旧对象的引用（例如那个模块的外部名称）不会被重新绑定到引用的新对象的，并且如果有需要，必须在出现的每个命名空间中进行更新。

有一些其他注意事项:

当一个模块被重新加载的时候，它的字典（包含了那个模块的全区变量）会被保留。名称的重新定义会覆盖旧的定义，所以通常来说这不是问题。如果一个新模块没有定义在旧版本模块中定义的名称，则将保留旧版本中的定义。这一特性可用于作为那个模块的优点，如果它维护一个全局表或者对象的缓存——使用 try 语句，就可以测试表的存在并且跳过它的初始化，如果有需要的话:

```
try:
    cache
except NameError:
    cache = {}
```

重新加载内置的或者动态加载模块，通常来说不是很有用处。不推荐重新加载”*sys*，*__main__*，*builtins* 和其它关键模块。在很多例子中，扩展模块并不是设计为不止一次的初始化，并且当重新加载时，可能会以任意方式失败。

如果一个模块使用 `from ... import ...` 导入的对象来自另外一个模块，给其它模块调用*reload()* 不会重新定义来自这个模块的对象——解决这个问题的一种方式是重新执行 `from` 语句，另一种方式是使用 `import` 和限定名称 (*module.name*) 来代替。

如果一个模块创建一个类的实例，重新加载定义那个类的模块不影响那些实例的方法定义———它们继续使用旧类中的定义。对于子类来说同样是正确的。

3.4 版新加入.

3.7 版更變: 当重新加载的那个模块缺少 `ModuleSpec` 的时候，会引起*ModuleNotFoundError* 异常。

31.5.3 importlib.abc ——关于导入的抽象基类

源代码： Lib/importlib/abc.py

The *importlib.abc* module contains all of the core abstract base classes used by `import`. Some subclasses of the core abstract base classes are also provided to help in implementing the core ABCs.

ABC 类的层次结构：

```
object
 +-- Finder (deprecated)
 |    +-- MetaPathFinder
 |    +-- PathEntryFinder
 +-- Loader
      +-- ResourceLoader --------+
      +-- InspectLoader          |
           +-- ExecutionLoader --+
                                 +-- FileLoader
                                 +-- SourceLoader
```

class importlib.abc.**Finder**
代表*finder* 的一个抽象基类

3.3 版後已🄵用: 使用*MetaPathFinder* 或*PathEntryFinder* 来代替。

abstractmethod find_module(*fullname*, *path=None*)
为指定的模块查找*loader* 定义的抽象方法。本来是在 **PEP 302** 指定的，这个方法是在*sys.meta_path* 和基于路径的导入子系统中使用。

3.4 版更變: 当被调用的时候，返回 `None` 而不是引发*NotImplementedError*。

class importlib.abc.**MetaPathFinder**
代表*meta path finder* 的一个抽象基类。为了保持兼容性，这是*Finder* 的一个子类。

3.3 版新加入.

find_spec(*fullname*, *path*, *target=None*)
一个抽象方法，用于查找指定模块的*spec* 。若是顶层导入，*path* 将为 `None`。否则就是查找子包或模块，*path* 将是父级包的 `__path__` 值。找不到则会返回 `None`。传入的 `target` 是一个模块对象，查找器可以用来对返回的规格进行更有依据的猜测。在实现具体的 `MetaPathFinders` 代码时，可能会用到*importlib.util.spec_from_loader()* 。

3.4 版新加入.

find_module(*fullname*, *path*)
一个用于查找指定的模块中*loader* 的遗留方法。如果这是最高层级的导入，*path* 的值将会是 `None`。否则，这是一个查找子包或者模块的方法，并且 *path* 的值将会是来自父包的 `__path__` 的值。如果未发现加载器，返回 `None`。

如果定义了*find_spec()* 方法，则提供了向后兼容的功能。

3.4 版更變: 当调用这个方法的时候返回 None 而不是引发*NotImplementedError*。可以使用*find_spec()* 来提供功能。

3.4 版後已F用: 使用*find_spec()* 来代替。

invalidate_caches()
当被调用的时候，一个可选的方法应该将查找器使用的任何内部缓存进行无效。将在*sys.meta_path* 上的所有查找器的缓存进行无效的时候，这个函数被*importlib.invalidate_caches()* 所使用。

3.4 版更變: 当方法被调用的时候，方法返回是 None 而不是 NotImplemented 。

class importlib.abc.**PathEntryFinder**
path entry finder 的一个抽象基类。尽管这个基类和*MetaPathFinder* 有一些相似之处，但是 PathEntryFinder 只在由 PathFinder 提供的基于路径导入子系统中使用。这个抽象类是*Finder* 的一个子类，仅仅是因为兼容性的原因。

3.3 版新加入.

find_spec(*fullname*, *target=None*)
一个抽象方法，用于查找指定模块的*spec*。搜索器将只在指定的*path entry* 内搜索该模块。找不到则会返回 None。在实现具体的 PathEntryFinders 代码时，可能会用到*importlib.util.spec_from_loader()* 。

3.4 版新加入.

find_loader(*fullname*)
一个用于在模块中查找一个*loader* 的遗留方法。返回一个 (loader, portion) 的 2 元组，portion 是一个贡献给命名空间包部分的文件系统位置的序列。加载器可能是 None，同时正在指定的 portion 表示的是贡献给命名空间包的文件系统位置。portion 可以使用一个空列表来表示加载器不是命名空间包的一部分。如果 loader 是 None 并且 portion 是一个空列表，那么命名空间包中无加载器或者文件系统位置可查找到（即在那个模块中未能找到任何东西）。

如果定义了*find_spec()* ，则提供了向后兼容的功能。

3.4 版更變: 返回 (None, []) 而不是引发*NotImplementedError*。当可于提供相应的功能的时候，使用*find_spec()*。

3.4 版後已F用: 使用*find_spec()* 来代替。

find_module(*fullname*)
Finder.find_module`的具体实现，该方法等价于``self.find_loader(fullname)[0]`()。

3.4 版後已F用: 使用*find_spec()* 来代替。

invalidate_caches()
当被调用的时候，一个可选的方法应该将查找器使用的任何内部缓存进行无效。当将所有缓存的查找器的缓存进行无效的时候，该函数被 PathFinder.invalidate_caches() 使用。

class importlib.abc.**Loader**
loader 的抽象基类。关于一个加载器的实际定义请查看 **PEP 302**。

加载器想要支持资源读取应该实现一个由*importlib.abc.ResourceReader* 指定的“get_resource_reader(fullname)“ 方法。

3.7 版更變: 引入了可选的 get_resource_reader() 方法。

create_module(*spec*)
当导入一个模块的时候，一个返回将要使用的那个模块对象的方法。这个方法可能返回 None ，这暗示着应该发生默认的模块创建语义。”

3.4 版新加入.

3.5 版更變: 从 Python 3.6 开始，当定义了*exec_module()* 的时候，这个方法将不会是可选的。

exec_module(*module*)
当一个模块被导入或重新加载时，一个抽象方法在它自己的命名空间中执行那个模块。当调用 exec_module() 的时候，那个模块应该已经被初始化了。当这个方法存在时，必须定义*create_module()* 。

3.4 版新加入.

3.6 版更變: *create_module()* 也必须被定义。

load_module(*fullname*)
用于加载一个模块的传统方法。如果这个模块不能被导入，将引起*ImportError* 异常，否则返回那个被加载的模块。

如果请求的模块已经存在*sys.modules*，应该使用并且重新加载那个模块。否则加载器应该是创建一个新的模块并且在任何家过程开始之前将这个新模块插入到*sys.modules* 中，来阻止递归导入。如果加载器插入了一个模块并且加载失败了，加载器必须从*sys.modules* 中将这个模块移除。在加载器开始执行之前，已经在*sys.modules* 中的模块应该被忽略 (查看*importlib.util.module_for_loader()*)。

加载器应该在模块上面设置几个属性。(要知道当重新加载一个模块的时候，那些属性某部分可以改变):

- **__name__** 模块的名字
- **__file__** 模块数据存储的路径 (不是为了内置的模块而设置)
- **__cached__** 被存储或应该被存储的模块的编译版本的路径（当这个属性不恰当的时候不设置)。
- **__path__** 指定在一个包中搜索路径的一个字符串列表。这个属性不在模块上面进行设置。
- **__package__** 当模块作为子模块加载时，其所在包的完全限定名称（顶层模块则为空字符串)。对于包而言，与 __name__ 相同。*importlib.util.module_for_loader()* 装饰器能够处理 __package__ 的细节。
- **__loader__** 用来加载那个模块的加载器。*importlib.util.module_for_loader()* 装饰器可以处理 __package__ 的细节。

当*exec_module()* 可用的时候，那么则提供了向后兼容的功能。

3.4 版更變: 当这个方法被调用的时候，触发*ImportError* 异常而不是*NotImplementedError*。当*exec_module()* 可用的时候，使用它的功能。

3.4 版後已🄵用: 加载模块推荐的使用的 API 是*exec_module()* (和*create_module()*)。加载器应该实现它而不是 load_module()。当 exec_module() 被实现的时候，导入机制关心的是 load_module() 所有其他的责任。

module_repr(*module*)
一个遗留方法，在实现时计算并返回给定模块的 repr，作为字符串。模块类型的默认 repr() 将根据需要使用此方法的结果。

3.3 版新加入.

3.4 版更變: 是可选的方法而不是一个抽象方法。

3.4 版後已🄵用: 现在导入机制会自动地关注这个方法。

class importlib.abc.**ResourceReader**
被 *TraversableResources* 取代

提供读取 *resources* 能力的一个*abstract base class* 。

从这个 ABC 的视角出发，*resource* 指一个包附带的二进制文件。常见的如在包的 `__init__.py` 文件旁的数据文件。这个类存在的目的是为了将对数据文件的访问进行抽象，这样包就和其数据文件的存储方式无关了。不论这些文件是存放在一个 zip 文件里还是直接在文件系统内。

对于该类中的任一方法，*resource* 参数的值都需要是一个在概念上表示文件名称的*path-like object*。这意味着任何子目录的路径都不该出现在 *resouce* 参数值内。因为对于阅读器而言，包的位置就代表着「目录」。因此目录和文件名就分别对应于包和资源。这也是该类的实例都需要和一个包直接关联（而不是潜在指代很多包或者一整个模块）的原因。

想支持资源读取的加载器需要提供一个返回实现了此 ABC 的接口的 `get_resource_reader(fullname)` 方法。如果通过全名指定的模块不是一个包，这个方法应该返回`None`。当指定的模块是一个包时，应该只返回一个与这个抽象类 ABC 兼容的对象。

3.7 版新加入.

abstractmethod open_resource(*resource*)
返回一个打开的*file-like object* 用于 *resource* 的二进制读取。

如果无法找到资源，将会引发`FileNotFoundError`。

abstractmethod resource_path(*resource*)
返回 *resource* 的文件系统路径。

如果资源并不实际存在于文件系统中，将会引发`FileNotFoundError`。

abstractmethod is_resource(*name*)
如果 *name* 被视作资源，则返回 True。如果 *name* 不存在，则引发`FileNotFoundError` 异常。

abstractmethod contents()
反回由字符串组成的*iterable*，表示这个包的所有内容。请注意并不要求迭代器返回的所有名称都是实际的资源，例如返回`is_resource()` 为假值的名称也是可接受的。

允许非资源名字被返回是为了允许存储的一个包和它的资源的方式是已知先验的并且非资源名字会有用的情况。比如，允许返回子目录名字，目的是当得知包和资源存储在文件系统上面的时候，能够直接使用子目录的名字。

这个抽象方法返回了一个不包含任何内容的可迭代对象。

class importlib.abc.**ResourceLoader**
一个*loader* 的抽象基类，它实现了可选的 **PEP 302** 协议用于从存储后端加载任意资源。

3.7 版後已🄴用: 由于要支持使用`importlib.abc.ResourceReader` 类来加载资源，这个 ABC 已经被弃用了。

abstractmethod get_data(*path*)
一个用于返回位于 *path* 的字节数据的抽象方法。有一个允许存储任意数据的类文件存储后端的加载器能够实现这个抽象方法来直接访问这些被存储的数据。如果不能够找到 *path*，则会引发`OSError` 异常。*path* 被希望使用一个模块的 `__file` 属性或来自一个包的 `__path__` 来构建。

3.4 版更變: 引发`OSError` 异常而不是`NotImplementedError` 异常。

class importlib.abc.**InspectLoader**
一个实现加载器检查模块可选的 **PEP 302** 协议的*loader* 的抽象基类。

get_code(*fullname*)
返回一个模块的代码对象，或如果模块没有一个代码对象（例如，对于内置的模块来说，这会是这种情况），则为 None。如果加载器不能找到请求的模块，则引发`ImportError` 异常。

備🄴: 当这个方法有一个默认的实现的时候，出于性能方面的考虑，如果有可能的话，建议覆盖它。

3.4 版更變: 不再抽象并且提供一个具体的实现。

abstractmethod get_source(*fullname*)
一个返回模块源的抽象方法。使用*universal newlines* 作为文本字符串被返回，将所有可识别行分割符翻译成 `'\n'` 字符。如果没有可用的源（例如，一个内置模块），则返回 `None`。如果加载器不能找到指定的模块，则引发*ImportError* 异常。

3.4 版更變: 引发*ImportError* 而不是*NotImplementedError*。

is_package(*fullname*)
可选方法，如果模块为包，则返回 True，否则返回 False。如果*loader* 找不到模块，则会触发*ImportError*。

3.4 版更變: 引发*ImportError* 而不是*NotImplementedError*。

static source_to_code(*data*, *path='<string>'*)
创建一个来自 Python 源码的代码对象。

参数 *data* 可以是任意*compile()* 函数支持的类型（例如字符串或字节串）。参数 *path* 应该是源代码来源的路径，这可能是一个抽象概念（例如位于一个 zip 文件中）。

在有后续代码对象的情况下，可以在一个模块中通过运行 “exec(code, module.__dict__)“来执行它。

3.4 版新加入.

3.5 版更變: 使得这个方法变成静态的。

exec_module(*module*)
Loader.exec_module() 的实现。

3.4 版新加入.

load_module(*fullname*)
Loader.load_module() 的实现。

3.4 版後已🄵用: 使用*exec_module()* 来代替。

class importlib.abc.**ExecutionLoader**
一个继承自*InspectLoader* 的抽象基类，当被实现时，帮助一个模块作为脚本来执行。这个抽象基类表示可选的 **PEP 302** 协议。

abstractmethod get_filename(*fullname*)
一个用来为指定模块返回 `__file__` 的值的抽象方法。如果无路径可用，则引发*ImportError*。

如果源代码可用，那么这个方法返回源文件的路径，不管是否是用来加载模块的字节码。

3.4 版更變: 引发*ImportError* 而不是*NotImplementedError*。

class importlib.abc.**FileLoader**(*fullname*, *path*)
一个继承自*ResourceLoader* 和*ExecutionLoader*，提供*ResourceLoader.get_data()* 和*ExecutionLoader.get_filename()* 具体实现的抽象基类。

参数 *fullname* 是加载器要处理的模块的完全解析的名字。参数 *path* 是模块文件的路径。

3.3 版新加入.

name
加载器可以处理的模块的名字。

path
模块的文件路径

load_module(*fullname*)
调用 super 的 “load_module()“。

3.4 版後已🄵用: 使用*Loader.exec_module()* 来代替。

abstractmethod get_filename(*fullname*)
返回*path*。

abstractmethod get_data(*path*)
读取 *path* 作为二进制文件并且返回来自它的字节数据。

class importlib.abc.**SourceLoader**
一个用于实现源文件（和可选地字节码）加载的抽象基类。这个类继承自*ResourceLoader*和*ExecutionLoader*，需要实现：

- *ResourceLoader.get_data()*
- ***ExecutionLoader.get_filename()*** 应该是只返回源文件的路径；不支持无源加载。

由这个类定义的抽象方法用来添加可选的字节码文件支持。不实现这些可选的方法（或导致它们引发*NotImplementedError*异常）导致这个加载器只能与源代码一起工作。实现这些方法允许加载器能与源 *和*字节码文件一起工作。不允许只提供字节码的 *无源式*加载。字节码文件是通过移除 Python 编译器的解析步骤来加速加载的优化，并且因此没有开放出字节码专用的 API。

path_stats(*path*)
返回一个包含关于指定路径的元数据的*dict*的可选的抽象方法。支持的字典键有：

- `'mtime'` (必选项): 一个表示源码修改时间的整数或浮点数；
- `'size'` (可选项)：源码的字节大小。

字典中任何其他键会被忽略，以允许将来的扩展。如果不能处理该路径，则会引发*OSError*。

3.3 版新加入.

3.4 版更變: 引发*OSError*而不是*NotImplemented*。

path_mtime(*path*)
返回指定文件路径修改时间的可选的抽象方法。

3.3 版後已🄵用: 在有了*path_stats()*的情况下，这个方法被弃用了。没必要去实现它了，但是为了兼容性，它依然处于可用状态。如果文件路径不能被处理，则引发*OSError*异常。

3.4 版更變: 引发*OSError*而不是*NotImplemented*。

set_data(*path*, *data*)
往一个文件路径写入指定字节的的可选的抽象方法。任何中间不存在的目录不会被自动创建。

由于路径是只读的，当写入的路径产生错误时（*errno.EACCES*/*PermissionError*），不会传播异常。

3.4 版更變: 当被调用时，不再引起*NotImplementedError*异常。

get_code(*fullname*)
*InspectLoader.get_code()*的具体实现。

exec_module(*module*)
*Loader.exec_module()*的具体实现。

3.4 版新加入.

load_module(*fullname*)
Concrete implementation of *Loader.load_module()*.

3.4 版後已🄵用: 使用*exec_module()*来代替。

get_source(*fullname*)
*InspectLoader.get_source()*的具体实现。

is_package(*fullname*)
InspectLoader.is_package() 的具体实现。一个模块被确定为一个包的条件是：它的文件路径（由*ExecutionLoader.get_filename()* 提供）当文件扩展名被移除时是一个命名为 `__init__` 的文件，并且这个模块名字本身不是以 "__init__"结束。

class importlib.abc.**Traversable**
拥有部分 pathlib.Path 方法的对象，适用于遍历目录和打开文件。

3.9 版新加入.

abstractmethod name()
The base name of this object without any parent references.

abstractmethod iterdir()
Yield Traversable objects in self.

abstractmethod is_dir()
Return True if self is a directory.

abstractmethod is_file()
Return True if self is a file.

abstractmethod joinpath(*child*)
Return Traversable child in self.

abstractmethod __truediv__(*child*)
Return Traversable child in self.

abstractmethod open(*mode='r'*, **args*, ***kwargs*)
mode may be 'r' or 'rb' to open as text or binary. Return a handle suitable for reading (same as *pathlib.Path.open*).

When opening as text, accepts encoding parameters such as those accepted by *io.TextIOWrapper*.

read_bytes()
Read contents of self as bytes.

read_text(*encoding=None*)
Read contents of self as text.

class importlib.abc.**TraversableResources**
用作资源读取器的抽象基类，能够为 `files` 接口提供服务。其为 ResourceReader 的子类，并具体实现了 ResourceReader 的抽象方法。因此，任何提供 TraversableReader 的加载器也提供 ResourceReader。

3.9 版新加入.

31.5.4 importlib.resources -- 资源

源码： Lib/importlib/resources.py

3.7 版新加入.

这个模块使得 Python 的导入系统提供了访问 * 包 * 内的 * 资源 * 的功能。如果能够导入一个包，那么就能够访问那个包里面的资源。资源可以以二进制或文本模式方式被打开或读取。

资源非常类似于目录内部的文件，要牢记的是这仅仅是一个比喻。资源和包不是与文件系统上的物理文件和目录一样存在着。

備🄴： 本模块提供了类似于 pkg_resources Basic Resource Access 的功能，但没有该包的性能开销。这样读取包中的资源就更为容易，语义也更稳定一致。

该模块有独立的向下移植版本，using importlib.resources 和 migrating from pkg_resources to importlib.resources 提供了更多信息。

加载器想要支持资源读取应该实现一个由*importlib.abc.ResourceReader* 指定的“get_resource_reader(fullname)“ 方法。

定义了以下类型。

importlib.resources.**Package**
Package 类型定义为 Union[str, ModuleType] 。这意味着只要函数说明接受 Package 的地方，就可以传入字符串或模块。模块对象必须拥有一个可解析的 __spec__.submodule_search_locations，不能是 None。

importlib.resources.**Resource**
此类型描述了传入本包各函数的资源名称。定义为 Union[str, os.PathLike]。

有以下函数可用：

importlib.resources.**files**(*package*)
返回一个 importlib.resources.abc.Traversable 对象，代表包的资源容器（可视为目录）及其资源（可视为文件）。Traversable 可以包含其他容器（可视为子目录）。

package 是包名或符合 Package 要求的模块对象。

3.9 版新加入.

importlib.resources.**as_file**(*traversable*)
给出代表某个文件的 importlib.resources.abc.Traversable 对象，通常是来自*importlib.resources.files()*，返回上下文管理器以供 with 语句使用。上下文管理器提供一个*pathlib.Path* 对象。

退出上下文管理程序时，可以清理所有临时文件，比如从压缩文件中提取资源时创建的那些文件。

如果 Traversable 方法（read_text 等）不够用，需要文件系统中的实际文件时，请使用 as_file。

3.9 版新加入.

importlib.resources.**open_binary**(*package*, *resource*)
以二进制读方式打开 *package* 内的 *resource*。

package 是包名或符合 Package 要求的模块对象。*resource* 是要在 *package* 内打开的资源名；不能包含路径分隔符，也不能有子资源（即不能为目录）。本函数将返回一个 “typing.BinaryIO“ 实例，即一个供读取的已打开的二进制 I/O 流。

importlib.resources.**open_text**(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
以文本读方式打开 *package* 内的 *resource*。默认情况下，资源将以 UTF-8 格式打开以供读取。

package 是包名或符合 Package 要求的模块对象。*resource* 是要在 *package* 内打开的资源名；不能包含路径分隔符，也不能有子资源（即不能是目录）。*encoding* 和 *errors* 的含义与内置*open()* 的一样。

本函数返回一个 typing.TextIO 实例，即一个打开的文本 I/O 流对象以供读取。

importlib.resources.**read_binary**(*package*, *resource*)
读取并返回 *package* 中的 *resource* 内容，格式为 bytes。

package 是包名或符合 Package 要求的模块对象。*resource* 是要在 *package* 内打开的资源名；不能包含路径分隔符，也不能有子资源（即不能是目录）。资源内容以*bytes* 的形式返回。

importlib.resources.**read_text**(*package*, *resource*, *encoding='utf-8'*, *errors='strict'*)
读取并返回 *package* 中 *resource* 的内容，格式为 str。默认情况下，资源内容将以严格的 UTF-8 格式读取。

package 是包名或符合 `Package` 要求的模块对象。*resource* 是要在 *package* 内打开的资源名；不能包含路径分隔符，也不能有子资源（即不能是目录）。*encoding* 和 *errors* 的含义与内置*open()* 的一样。资源内容将以*str* 的形式返回。

`importlib.resources.`**`path`**(*package*, *resource*)
返回 *resource* 实际的文件系统路径。本函数返回一个上下文管理器，以供 `with` 语句中使用。上下文管理器提供一个*pathlib.Path* 对象。

退出上下文管理程序时，可以清理所有临时文件，比如从压缩文件中提取资源时创建的那些文件。

package 是包名或符合 `Package` 要求的模块对象。*resource* 是要在 *package* 内打开的资源名；不能包含路径分隔符，也不能有子资源（即不能是目录）。

`importlib.resources.`**`is_resource`**(*package*, *name*)
如果包中存在名为 *name* 的资源，则返回 `True`，否则返回 `False`。请记住，目录 不是资源！*package* 为包名或一个符合 `Package` 要求的模块对象。

`importlib.resources.`**`contents`**(*package*)
返回一个用于遍历包内各命名项的可迭代对象。该可迭代对象将返回*str* 资源（如文件）及非资源（如目录）。该迭代器不会递归进入子目录。

package 是包名或符合 `Package` 要求的模块对象。

31.5.5 `importlib.machinery` ——导入器和路径钩子函数。

源代码： Lib/importlib/machinery.py

本模块包含多个对象，以帮助 `import` 查找并加载模块。

`importlib.machinery.`**`SOURCE_SUFFIXES`**
一个字符串列表，表示源模块的可识别的文件后缀。

3.3 版新加入.

`importlib.machinery.`**`DEBUG_BYTECODE_SUFFIXES`**
一个字符串列表，表示未经优化字节码模块的文件后缀。

3.3 版新加入.

3.5 版後已🄵用: 改用*BYTECODE_SUFFIXES* 。

`importlib.machinery.`**`OPTIMIZED_BYTECODE_SUFFIXES`**
一个字符串列表，表示已优化字节码模块的文件后缀。

3.3 版新加入.

3.5 版後已🄵用: 改用*BYTECODE_SUFFIXES* 。

`importlib.machinery.`**`BYTECODE_SUFFIXES`**
一个字符串列表，表示字节码模块的可识别的文件后缀（包含前导的句点符号）。

3.3 版新加入.

3.5 版更變: 该值不再依赖于 `__debug__` 。

`importlib.machinery.`**`EXTENSION_SUFFIXES`**
一个字符串列表，表示扩展模块的可识别的文件后缀。

3.3 版新加入.

importlib.machinery.**all_suffixes**()
返回字符串的组合列表，代表标准导入机制可识别模块的所有文件后缀。这是个助手函数，只需知道某个文件系统路径是否会指向模块，而不需要任何关于模块种类的细节（例如*inspect.getmodulename()*）。

3.3 版新加入.

class importlib.machinery.**BuiltinImporter**
用于导入内置模块的*importer*。所有已知的内置模块都已列入*sys.builtin_module_names*。此类实现了*importlib.abc.MetaPathFinder* 和*importlib.abc.InspectLoader* 抽象基类。

此类只定义类的方法，以减轻实例化的开销。

3.5 版更變: 作为 **PEP 489** 的一部分，现在内置模块导入器实现了 Loader.create_module() 和 Loader.exec_module()。

class importlib.machinery.**FrozenImporter**
用于已冻结模块的*importer*。此类实现了*importlib.abc.MetaPathFinder* 和*importlib.abc.InspectLoader* 抽象基类。

此类只定义类的方法，以减轻实例化的开销。

3.4 版更變: 有了 create_module() 和 exec_module() 方法。

class importlib.machinery.**WindowsRegistryFinder**
Finder 用于查找在 Windows 注册表中声明的模块。该类实现了基础的*importlib.abc.MetaPathFinder*。

此类只定义类的方法，以减轻实例化的开销。

3.3 版新加入.

3.6 版後已F用: 改用*site* 配置。未来版本的 Python 可能不会默认启用该查找器。

class importlib.machinery.**PathFinder**
用于*sys.path* 和包的 __path__ 属性的*Finder*。该类实现了基础的*importlib.abc.MetaPathFinder*。

此类只定义类的方法，以减轻实例化的开销。

classmethod find_spec(*fullname*, *path=None*, *target=None*)
类方法试图在*sys.path* 或 *path* 上为 *fullname* 指定的模块查找*spec*。对于每个路径条目，都会查看*sys.path_importer_cache*。如果找到非 False 的对象，则将其用作*path entry finder* 来查找要搜索的模块。如果在*sys.path_importer_cache* 中没有找到条目，那会在*sys.path_hooks* 检索该路径条目的查找器，找到了则和查到的模块信息一起存入*sys.path_importer_cache*。如果查找器没有找到，则缓存中的查找器和模块信息都存为 None，并返回。

3.4 版新加入.

3.5 版更變: 如果当前工作目录不再有效（用空字符串表示），则返回 None，但在*sys.path_importer_cache* 中不会有缓存值。

classmethod find_module(*fullname*, *path=None*)
一个过时的*find_spec()* 封装对象。

3.4 版後已F用: 使用*find_spec()* 来代替。

classmethod invalidate_caches()
为所有存于*sys.path_importer_cache* 中的查找器，调用其*importlib.abc.PathEntryFinder.invalidate_caches()* 方法。*sys.path_importer_cache* 中为 None 的条目将被删除。

3.7 版更變: *sys.path_importer_cache* 中的 None 条目将被删除。

3.4 版更變: 调用*sys.path_hooks* 中的对象，当前工作目录为 `''` (即空字符串)。

class importlib.machinery.**FileFinder**(*path*, **loader_details*)
importlib.abc.PathEntryFinder 的一个具体实现，它会缓存来自文件系统的结果。

参数 *path* 是查找器负责搜索的目录。

loader_details 参数是数量不定的二元组，每个元组包含加载器及其可识别的文件后缀列表。加载器应为可调用对象，可接受两个参数，即模块的名称和已找到文件的路径。

查找器将按需对目录内容进行缓存，通过对每个模块的检索进行状态统计，验证缓存是否过期。因为缓存的滞后性依赖于操作系统文件系统状态信息的粒度，所以搜索模块、新建文件、然后搜索新文件代表的模块，这会存在竞争状态。如果这些操作的频率太快，甚至小于状态统计的粒度，那么模块搜索将会失败。为了防止这种情况发生，在动态创建模块时，请确保调用*importlib.invalidate_caches()*。

3.3 版新加入.

path
查找器将要搜索的路径。

find_spec(*fullname*, *target=None*)
尝试在*path* 中找到处理 *fullname* 的规格。

3.4 版新加入.

find_loader(*fullname*)
试图在*path* 内找到处理 *fullname* 的加载器。

invalidate_caches()
清理内部缓存。

classmethod path_hook(**loader_details*)
一个类方法，返回供*sys.path_hooks* 使用的闭包。根据直接给出的路径参数和间接给出的 *loader_details*，闭包会返回一个*FileFinder* 的实例。

如果给闭包的参数不是已存在的目录，将会触发*ImportError*。

class importlib.machinery.**SourceFileLoader**(*fullname*, *path*)
importlib.abc.SourceLoader 的一个具体实现，该实现子类化了*importlib.abc.FileLoader* 并提供了其他一些方法的具体实现。

3.3 版新加入.

name
该加载器将要处理的模块名称。

path
源文件的路径

is_package(*fullname*)
如果*path* 看似包的路径，则返回 `True`。

path_stats(*path*)
importlib.abc.SourceLoader.path_stats() 的具体代码实现。

set_data(*path*, *data*)
importlib.abc.SourceLoader.set_data() 的具体代码实现。

load_module(*name=None*)
importlib.abc.Loader.load_module() 的具体代码实现，这里要加载的模块名是可选的。

3.6 版後已𝔽用: 改用*importlib.abc.Loader.exec_module()* 。

class importlib.machinery.**SourcelessFileLoader**(*fullname*, *path*)
importlib.abc.FileLoader 的具体代码实现，可导入字节码文件（也即源代码文件不存在）。

请注意，直接用字节码文件（而不是源代码文件），会让模块无法应用于所有的 Python 版本或字节码格式有所改动的新版本 Python。

3.3 版新加入.

name
加载器将要处理的模块名。

path
二进制码文件的路径。

is_package(*fullname*)
根据*path* 确定该模块是否为包。

get_code(*fullname*)
返回由*path* 创建的*name* 的代码对象。

get_source(*fullname*)
因为用此加载器时字节码文件没有源码文件，所以返回 None。

load_module(*name=None*)

importlib.abc.Loader.load_module() 的具体代码实现，这里要加载的模块名是可选的。

3.6 版後已🄵用: 改用*importlib.abc.Loader.exec_module()* 。

class importlib.machinery.**ExtensionFileLoader**(*fullname*, *path*)
importlib.abc.ExecutionLoader 的具体代码实现，用于扩展模块。

参数 *fullname* 指定了加载器要支持的模块名。参数 *path* 是指向扩展模块文件的路径。

3.3 版新加入.

name
装载器支持的模块名。

path
扩展模块的路径。

create_module(*spec*)
根据 **PEP 489** ，由给定规范创建模块对象。

3.5 版新加入.

exec_module(*module*)
根据 **PEP 489**，初始化给定的模块对象。

3.5 版新加入.

is_package(*fullname*)
根据*EXTENSION_SUFFIXES* ，如果文件路径指向某个包的 __init__ 模块，则返回 True。

get_code(*fullname*)
返回 None，因为扩展模块缺少代码对象。

get_source(*fullname*)
返回 None，因为扩展模块没有源代码。

get_filename(*fullname*)
返回*path*。

3.4 版新加入.

class importlib.machinery.**ModuleSpec**(*name*, *loader*, *, *origin=None*, *loader_state=None*, *is_package=None*)

关于模块导入系统相关状态的规范。通常这是作为模块的 `__spec__` 属性暴露出来。在以下描述中，括号里的名字给出了模块对象中直接可用的属性。比如 `module.__spec__.origin == module.__file__`。但是请注意，虽然 值通常是相等的，但它们可以不同，因为两个对象之间没有进行同步。因此 `__path__` 有可能在运行时做过更新，而这不会自动反映在 `__spec__.submodule_search_locations` 中。

3.4 版新加入.

name

(`__name__`)

一个字符串，表示模块的完全限定名称。

loader

(`__loader__`)

模块加载时应采用的*Loader* 。*Finders* 应确保设置本属性。

origin

(`__file__`)

装载模块所在位置的名称，如内置模块为 “buildin”，从源代码加载的模块为文件名。通常应设置 “origin”，但它可能为 `None` (默认值)，表示未指定 (如命名空间包)。

submodule_search_locations

(`__path__`)

如果是包（否则为），子模块所在位置的字符串列表（否则为 `None`）。

loader_state

依模块不同的额外数据的容器，以供加载过程中使用（或 `None`）。

cached

(`__cached__`)

字符串，表示编译后的模块所在位置（或 `None`）。

parent

(`__package__`)

包的完全限定名称（只读），模块应作为其子模块进行加载（对于顶层模块则为空字符串）。对于包而言，其值与 `__name__` 相同。

has_location

布尔值，表示模块的 “origin” 属性是否指向可加载的位置。

31.5.6 importlib.util ——导入器的工具程序代码

源代码： Lib/importlib/util.py

本模块包含了帮助构建*importer* 的多个对象。

importlib.util.**MAGIC_NUMBER**
代表字节码版本号的字节串。若要有助于加载/写入字节码，可考虑采用*importlib.abc.SourceLoader*。

3.4 版新加入.

importlib.util.**cache_from_source**(*path*, *debug_override=None*, *, *optimization=None*)
返回 **PEP 3147**/**PEP 488** 定义的，与源 *path* 相关联的已编译字节码文件的路径。例如，如果 *path* 为 `/foo/bar/baz.py` 则 Python 3.2 中的返回值将是 `/foo/bar/__pycache__/baz.cpython-32.pyc`。字符串 `cpython-32` 来自于当前的魔法标签 (参见 `get_tag()`; 如果 `sys.implementation.cache_tag` 未定义则将会引发*NotImplementedError*)。

参数 *optimization* 用于指定字节码文件的优化级别。空字符串代表没有优化，所以 *optimization* 为的 `/foo/bar/baz.py`，将会得到字节码路径为 `/foo/bar/__pycache__/baz.cpython-32.pyc`。`None` 会导致采用解释器的优化。任何其他字符串都会被采用，所以 *optimization* 为 `''` 的 `/foo/bar/baz.py` 会导致字节码路径为 `/foo/bar/__pycache__/baz.cpython-32.opt-2.pyc`。*optimization* 字符串只能是字母数字，否则会触发*ValueError*。

debug_override 参数已废弃，可用于覆盖系统的 `__debug__` 值。`True` 值相当于将 *optimization* 设为空字符串。`False` 则相当于 *optimization* 设为 `1`。如果 *debug_override* 和 *optimization* 都不为 `None`，则会触发*TypeError*。

3.4 版新加入.

3.5 版更變: 增加了 *optimization* 参数，废弃了 *debug_override* 参数。

3.6 版更變: 接受一个类路径对象。

importlib.util.**source_from_cache**(*path*)
根据指向一个 **PEP 3147** 文件名的 *path*，返回相关联的源代码文件路径。举例来说，如果 *path* 为 `/foo/bar/__pycache__/baz.cpython-32.pyc` 则返回的路径将是 `/foo/bar/baz.py`。*path* 不需要已存在，但如果它未遵循 **PEP 3147** 或 **PEP 488** 的格式，则会引发*ValueError*。如果未定义 `sys.implementation.cache_tag`，则会引发*NotImplementedError*。

3.4 版新加入.

3.6 版更變: 接受一个类路径对象。

importlib.util.**decode_source**(*source_bytes*)
对代表源代码的字节串进行解码，并将其作为带有通用换行符的字符串返回（符合*importlib.abc.InspectLoader.get_source()* 要求）。

3.4 版新加入.

importlib.util.**resolve_name**(*name*, *package*)
将模块的相对名称解析为绝对名称。

如果 **name** 前面没有句点，那就简单地返回 **name**。这样就能采用 “importlib.util.resolve_name('sys', __spec__.parent)“ 之类的写法，而无需检查是否需要 **package** 参数。

如果 **name** 是相对模块名称，但 **package** 为 False 值（如 `None` 或空字符串），则会触发*ImportError*。如果相对名称会离开所在的包（如从 `spam` 包中请求 `..bacon`），则还会触发*ImportError*。

3.3 版新加入.

3.9 版更變: 为了改善与 import 语句的一致性，对于无效的相对导入尝试会引发*ImportError* 而不是*ValueError*。

importlib.util.**find_spec**(*name*, *package=None*)
查找模块的*spec*，可选相对指定的 **包** 名。如果该模块位于*sys.modules* 中，则会返回 sys.modules[name].__spec__ （除非 spec 为 None 或未作设置，这时会触发*ValueError*）。否则将用*sys.meta_path* 进行搜索。若找不到则返回 None。

如果 **name** 为一个子模块（带有一个句点），则会自动导入父级模块。

name 和 **package** 的用法与 import_module() 相同。

3.4 版新加入.

3.7 版更變: 如果 **package** 实际上不是一个包（即缺少 __path__ 属性）则会引发*ModuleNotFoundError* 而不是*AttributeError*。

importlib.util.**module_from_spec**(*spec*)
基于 **spec** 和*spec.loader.create_module* 创建一个新模块。

如果*spec.loader.create_module* 未返回 None，那么先前已存在的属性不会被重置。另外，如果*AttributeError* 是在访问 **spec** 或设置模块属性时触发的，则不会触发。

本函数比*types.ModuleType* 创建新模块要好，因为用到 **spec** 模块设置了尽可能多的导入控制属性。

3.5 版新加入.

@importlib.util.**module_for_loader**
importlib.abc.Loader.load_module() 的一个*decorator*，用来选取合适的模块对象以供加载。被装饰方法的签名应带有两个位置参数（如: load_module(self, module)），其中第二个参数将是加载器用到的模块 **对象**。请注意，由于假定有两个参数，所以装饰器对静态方法不起作用。

装饰的方法将接受要加载的模块的 **name**，正如*loader* 一样。如果在*sys.modules* 中没有找到该模块，那么将构造一个新模块。不管模块来自哪里，__loader__ 设置为 **self** ，并且 __package__ 是根据*importlib.abc.InspectLoader.is_package()* 的返回值设置的。这些属性会无条件进行设置以便支持重载。

如果被装饰的方法触发异常，并且已有模块加入*sys.modules* 中，那么该模块将被移除，以防*sys.modules* 中残留一个部分初始化的模块。如果该模块原先已在*sys.modules* 中，则会保留。

3.3 版更變: 有可能时自动设置 __loader__ 和 __package__ 。

3.4 版更變: 无条件设置 __name__ 、__loader__ 、__package__ 以支持再次加载。

3.4 版後已🄵用: 现在，导入机制直接执行本函数提供的所有功能。

@importlib.util.**set_loader**
一个*decorator*，用于*importlib.abc.Loader.load_module()* 在返回的模块上设置 __loader__ 属性。如果该属性已被设置，装饰器就什么都不做。这里假定被封装方法的第一个位置参数（即 self）就是 __loader__ 要设置的。

3.4 版更變: 如果设为 None ，则会去设置 __loader__ ，就像该属性不存在一样。

3.4 版後已🄵用: 现在导入机制会自动用到本方法。

@importlib.util.**set_package**
一个用于*importlib.abc.Loader.load_module()* 的*decorator* ，以便设置返回模块的 __package__ 属性。如果 __package__ 已设置且不为 None，则不会做改动。

3.4 版後已🄵用: 现在导入机制会自动用到本方法。

importlib.util.**spec_from_loader**(*name*, *loader*, ***, *origin=None*, *is_package=None*)
一个工厂函数，用于创建基于加载器的 ModuleSpec 实例。形参的含义与 ModuleSpec 的相同。该函数会利用当前可用的*loader* API，比如 InspectLoader.is_package()，以填充所有缺失的规格信息。

3.4 版新加入.

importlib.util.**spec_from_file_location**(*name*, *location*, *, *loader=None*, *submodule_search_locations=None*)

一个工厂函数，用于创建基于文件路径的 ModuleSpec 实例。缺失的信息将通过利用加载器 API 以及模块将基于文件的隐含条件在 spec 上进行填充。

3.4 版新加入.

3.6 版更變: 接受一个类路径对象。

importlib.util.**source_hash**(*source_bytes*)

以字节串的形式返回 *source_bytes* 的哈希值。基于哈希值的 .pyc 文件在头部嵌入了对应源文件内容的*source_hash()*。

3.7 版新加入.

class importlib.util.**LazyLoader**(*loader*)

此类会延迟执行模块加载器，直至该模块有一个属性被访问到。

此类**只**适用于定义了*exec_module()* 的加载器，因为需要控制模块的类型。同理，加载器的*create_module()* 方法必须返回 None 或 __class__ 属性可被改变且不用*slots* 的类型。最后，用于替换*sys.modules* 内容的模块将无法工作，因为无法在整个解释器中安全地替换模块的引用；如果检测到这种替换，将触发*ValueError*。

備🄵: 如果项目对启动时间要求很高，只要模块未被用过，此类能够最小化加载模块的开销。对于启动时间并不重要的项目来说，由于加载过程中产生的错误信息会被暂时搁置，因此强烈不建议使用此类。

3.5 版新加入.

3.6 版更變: 开始调用*create_module()*，移除*importlib.machinery.BuiltinImporter* 和*importlib.machinery.ExtensionFileLoader* 的兼容性警告。

classmethod **factory**(*loader*)

静态方法，返回创建延迟加载器的可调用对象。就是说用在加载器用类而不是实例传递的场合。

```
suffixes = importlib.machinery.SOURCE_SUFFIXES
loader = importlib.machinery.SourceFileLoader
lazy_loader = importlib.util.LazyLoader.factory(loader)
finder = importlib.machinery.FileFinder(path, (lazy_loader, suffixes))
```

31.5.7 示例

用编程方式导入

要以编程方式导入一个模块，请使用*importlib.import_module()*：

```
import importlib

itertools = importlib.import_module('itertools')
```

检查某模块可否导入。

若要查看某个模块是否可导入，但不需要实际执行导入，则应使用*importlib.util.find_spec()*：

```
import importlib.util
import sys

# For illustrative purposes.
name = 'itertools'

if name in sys.modules:
    print(f"{name!r} already in sys.modules")
elif (spec := importlib.util.find_spec(name)) is not None:
    # If you chose to perform the actual import ...
    module = importlib.util.module_from_spec(spec)
    sys.modules[name] = module
    spec.loader.exec_module(module)
    print(f"{name!r} has been imported")
else:
    print(f"can't find the {name!r} module")
```

直接导入源文件。

若要直接导入 Python 源码文件，请使用一下方案（仅 Python 3.5 以上版本有效）：

```
import importlib.util
import sys

# For illustrative purposes.
import tokenize
file_path = tokenize.__file__
module_name = tokenize.__name__

spec = importlib.util.spec_from_file_location(module_name, file_path)
module = importlib.util.module_from_spec(spec)
sys.modules[module_name] = module
spec.loader.exec_module(module)
```

设置导入器

对于深度定制的导入，通常需要实现一个*importer*。这意味着得同时管理*finder* 和*loader*。根据不同的需求，有两种类型的查找器可供选择：*meta path finder* 或*path entry finder*。前者应位于*sys.meta_path* 之上，而后者是用*path entry hook* 在*sys.path_hooks* 上创建但与*sys.path* 一起工作，可能会创建一个查找器。以下例子将演示如何注册自己的导入器，以供导入使用（关于自建导入器请阅读本包内定义的类文档）：

```
import importlib.machinery
import sys

# For illustrative purposes only.
SpamMetaPathFinder = importlib.machinery.PathFinder
SpamPathEntryFinder = importlib.machinery.FileFinder
loader_details = (importlib.machinery.SourceFileLoader,
                  importlib.machinery.SOURCE_SUFFIXES)
```

(下页继续)

(繼續上一頁)

```
# Setting up a meta path finder.
# Make sure to put the finder in the proper location in the list in terms of
# priority.
sys.meta_path.append(SpamMetaPathFinder)

# Setting up a path entry finder.
# Make sure to put the path hook in the proper location in the list in terms
# of priority.
sys.path_hooks.append(SpamPathEntryFinder.path_hook(loader_details))
```

importlib.import_module() 的近似实现

导入过程本身是用 Python 代码实现的，这样就能通过 importlib 将大多数导入机制暴露出来。以下代码近似实现了*importlib.import_module()*，以帮助说明 importlib 暴露出来的各种 API （importlib 的用法适用于 Python 3.4 以上版本，其他代码适用于 Python 3.6 以上版本）。

```
import importlib.util
import sys

def import_module(name, package=None):
    """An approximate implementation of import."""
    absolute_name = importlib.util.resolve_name(name, package)
    try:
        return sys.modules[absolute_name]
    except KeyError:
        pass

    path = None
    if '.' in absolute_name:
        parent_name, _, child_name = absolute_name.rpartition('.')
        parent_module = import_module(parent_name)
        path = parent_module.__spec__.submodule_search_locations
    for finder in sys.meta_path:
        spec = finder.find_spec(absolute_name, path)
        if spec is not None:
            break
    else:
        msg = f'No module named {absolute_name!r}'
        raise ModuleNotFoundError(msg, name=absolute_name)
    module = importlib.util.module_from_spec(spec)
    sys.modules[absolute_name] = module
    spec.loader.exec_module(module)
    if path is not None:
        setattr(parent_module, child_name, module)
    return module
```

31.6 使用 importlib.metadata

源代码： Lib/importlib/metadata.py

3.8 版新加入.

備🄵： 这个功能是暂定的，可能会偏离标准库通常的版本语义。

---

importlib.metadata 是一个提供对已安装包的元数据访问的库。这个库部分建立在 Python 的导入系统上，旨在取代 pkg_resources 的 entry point API 和 metadata API 中的类似功能。通过和 Python 3.7 或更高版本中的*importlib.resources* 一同使用（对于旧版本的 Python 则作为 importlib_resources 向后移植），这可以消除对使用较旧且较为低效的 pkg_resources 包的需要。

此处所说的“已安装的包”通常指通过 pip 等工具安装在 Python site-packages 目录下的第三方包。具体来说，它指的是一个具有可发现的 dist-info 或 egg-info 目录以及 **PEP 566** 或其更早的规范所定义的元数据的包。默认情况下，包的元数据可以存在于文件系统中或*sys.path* 上的压缩文件中。通过扩展机制，元数据几乎可以存在于任何地方。

31.6.1 概述

假设你想得到你用 pip 安装的一个包的版本字符串。我们首先创建一个虚拟环境，并在其中安装一些东西：

```
$ python3 -m venv example
$ source example/bin/activate
(example) $ pip install wheel
```

你可以通过运行以下代码得到 “wheel“的版本字符串：

```
(example) $ python
>>> from importlib.metadata import version
>>> version('wheel')
'0.32.3'
```

你也可以获得以组名为关键字的入口点集合，比如 console_scripts 和 distutils.commands 。每个组包含一个入口点 对象的序列。

你可以得到分发的元数据：

```
>>> list(metadata('wheel'))
['Metadata-Version', 'Name', 'Version', 'Summary', 'Home-page', 'Author', 'Author-
↪email', 'Maintainer', 'Maintainer-email', 'License', 'Project-URL', 'Project-URL',
↪'Project-URL', 'Keywords', 'Platform', 'Classifier', 'Classifier', 'Classifier',
↪'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier',
↪'Classifier', 'Classifier', 'Classifier', 'Requires-Python', 'Provides-Extra',
↪'Requires-Dist', 'Requires-Dist']
```

你也可以获得分发的版本号，列出它的构成文件，并且得到分发的分发的依赖 列表。

31.6.2 可用 API

这个包通过其公共 API 提供了以下功能。

入口点

The `entry_points()` function returns a dictionary of all entry points, keyed by group. Entry points are represented by `EntryPoint` instances; each `EntryPoint` has a `.name`, `.group`, and `.value` attributes and a `.load()` method to resolve the value. There are also `.module`, `.attr`, and `.extras` attributes for getting the components of the `.value` attribute:

```
>>> eps = entry_points()
>>> list(eps)
['console_scripts', 'distutils.commands', 'distutils.setup_keywords', 'egg_info.
↪writers', 'setuptools.installation']
>>> scripts = eps['console_scripts']
>>> wheel = [ep for ep in scripts if ep.name == 'wheel'][0]
>>> wheel
EntryPoint(name='wheel', value='wheel.cli:main', group='console_scripts')
>>> wheel.module
'wheel.cli'
>>> wheel.attr
'main'
>>> wheel.extras
[]
>>> main = wheel.load()
>>> main
<function main at 0x103528488>
```

`group` 和 `name` 是由软件包作者定义的任意值，通常客户端会希望解析某个特定组的所有入口点。阅读 the setuptools docs 以了解更多关于入口点、其定义和用法的信息。

分发的元数据

每个分发都包含某些元数据，你可以通过 `metadata()` 函数提取它们：

```
>>> wheel_metadata = metadata('wheel')
```

The keys of the returned data structure[1] name the metadata keywords, and their values are returned unparsed from the distribution metadata:

```
>>> wheel_metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
```

[1] Technically, the returned distribution metadata object is an *email.message.EmailMessage* instance, but this is an implementation detail, and not part of the stable API. You should only use dictionary-like methods and syntax to access the metadata contents.

分发的版本

version() 函数是以字符串形式获取分发的版本号的最快方式：

```
>>> version('wheel')
'0.32.3'
```

分发的文件

你可以获得分发内包含的所有文件的集合。files() 函数传入一个分发包的名字并返回所有这个分发安装的文件。每个返回的文件对象都是一个*pathlib.PurePath* 的具有额外由元数据指定的 dist，size 和 hash 属性的子类 PackagePath 的实例。例如：

```
>>> util = [p for p in files('wheel') if 'util.py' in str(p)][0]
>>> util
PackagePath('wheel/util.py')
>>> util.size
859
>>> util.dist
<importlib.metadata._hooks.PathDistribution object at 0x101e0cef0>
>>> util.hash
<FileHash mode: sha256 value: bYkw5oMccfazVCoYQwKkkemoVyMAFoR34mmKBx8R1NI>
```

当你获得了文件对象，你可以读取其内容：

```
>>> print(util.read_text())
import base64
import sys
...
def as_bytes(s):
    if isinstance(s, text_type):
        return s.encode('utf-8')
    return s
```

你也可以使用 locate 方法来获得文件的绝对路径：

```
>>> util.locate()
PosixPath('/home/gustav/example/lib/site-packages/wheel/util.py')
```

当列出包含文件的元数据文件（RECORD 或 SOURCES.txt）不存在时，files() 函数将返回 None 。调用者可能会想要将对 files() 的调用封装在 always_iterable 中，或者用其他方法来应对目标分发元数据存在性未知的情况。

分发的依赖

使用 requires() 函数来获得分发的依赖集合：

```
>>> requires('wheel')
["pytest (>=3.0.0) ; extra == 'test'", "pytest-cov ; extra == 'test'"]
```

31.6.3 分发

以上的 API 是最常见而方便的用法，但是你也可以通过 Distribution 类获得以上所有信息。Distribution 是一个代表 Python 包的元数据的抽象对象。你可以这样获取 Distribution 实例：

```
>>> from importlib.metadata import distribution
>>> dist = distribution('wheel')
```

因此，可以通过 Distribution 实例获得版本号：

```
>>> dist.version
'0.32.3'
```

Distribution 实例具有所有可用的附加元数据：

```
>>> dist.metadata['Requires-Python']
'>=2.7, !=3.0.*, !=3.1.*, !=3.2.*, !=3.3.*'
>>> dist.metadata['License']
'MIT'
```

此处并未描述全部可用的元数据集合。请参见 **PEP 566** 以了解更多细节。

31.6.4 扩展搜索算法

因为包的元数据无法通过搜索*sys.path* 或通过包加载器获得，包的元数据是通过导入系统的 查找器找到的。importlib.metadata 查询在*sys.meta_path* 上的*元数据查找器* 列表以获得分发包的元数据。

Python 默认的 PathFinder 包含一个调用 importlib.metadata.MetadataPathFinder 来查找从典型的文件系统路径加载发布的钩子。

抽象基类*importlib.abc.MetaPathFinder* 定义了 Python 导入系统期望的查找器接口。importlib.metadata 通过寻找*sys.meta_path* 上查找器可选的 find_distributions 可调用的属性扩展这个协议，并将这个扩展接口作为 DistributionFinder 抽象基类提供，它定义了这个抽象方法：

```
@abc.abstractmethod
def find_distributions(context=DistributionFinder.Context()):
    """Return an iterable of all Distribution instances capable of
    loading the metadata for packages for the indicated ``context``.
    """
```

DistributionFinder.Context 对象提供了指示搜索路径和匹配名称的属性 .path 和 .name，也可能提供其他相关的上下文。

这在实践中意味着要支持在文件系统外的其他位置查找分发包的元数据，你需要子类化 Distribution 并实现抽象方法，之后从一个自定义查找器的 find_distributions() 方法返回这个派生的 Distribution 实例。

备注

CHAPTER 32

Python 语言服务

Python 提供了许多模块来帮助使用 Python 语言。这些模块支持标记化、解析、语法分析、字节码反汇编以及各种其他工具。

这些模块包括：

32.1 `parser` --- 访问 Python 解析树

The *parser* module provides an interface to Python's internal parser and byte-code compiler. The primary purpose for this interface is to allow Python code to edit the parse tree of a Python expression and create executable code from this. This is better than trying to parse and modify an arbitrary Python code fragment as a string because parsing is performed in a manner identical to the code forming the application. It is also faster.

警告： parser 模块已被弃用并将在未来的 Python 版本中移除。对于大多数用例你都可以使用*ast* 模块来控制抽象语法树（AST）的生成和编译阶段。

There are a few things to note about this module which are important to making use of the data structures created. This is not a tutorial on editing the parse trees for Python code, but some examples of using the *parser* module are presented.

Most importantly, a good understanding of the Python grammar processed by the internal parser is required. For full information on the language syntax, refer to reference-index. The parser itself is created from a grammar specification defined in the file `Grammar/Grammar` in the standard Python distribution. The parse trees stored in the ST objects created by this module are the actual output from the internal parser when created by the *expr()* or *suite()* functions, described below. The ST objects created by *sequence2st()* faithfully simulate those structures. Be aware that the values of the sequences which are considered "correct" will vary from one version of Python to another as the formal grammar for the language is revised. However, transporting code from one Python version to another as source text will always allow correct parse trees to be created in the target version, with the only restriction being that migrating to an older version of the interpreter will not support more recent language constructs. The parse trees are not typically compatible from one version to another, though source code has usually been forward-compatible within a major release series.

Each element of the sequences returned by *st2list()* or *st2tuple()* has a simple form. Sequences representing non-terminal elements in the grammar always have a length greater than one. The first element is an integer which identifies a production in the grammar. These integers are given symbolic names in the C header file `Include/graminit.h` and the Python module *symbol*. Each additional element of the sequence represents a component of the production as recognized in the input string: these are always sequences which have the same form as the parent. An important aspect of this structure which should be noted is that keywords used to identify the parent node type, such as the keyword `if` in an `if_stmt`, are included in the node tree without any special treatment. For example, the `if` keyword is represented by the tuple `(1, 'if')`, where `1` is the numeric value associated with all `NAME` tokens, including variable and function names defined by the user. In an alternate form returned when line number information is requested, the same token might be represented as `(1, 'if', 12)`, where the `12` represents the line number at which the terminal symbol was found.

Terminal elements are represented in much the same way, but without any child elements and the addition of the source text which was identified. The example of the `if` keyword above is representative. The various types of terminal symbols are defined in the C header file `Include/token.h` and the Python module *token*.

The ST objects are not required to support the functionality of this module, but are provided for three purposes: to allow an application to amortize the cost of processing complex parse trees, to provide a parse tree representation which conserves memory space when compared to the Python list or tuple representation, and to ease the creation of additional modules in C which manipulate parse trees. A simple ”wrapper” class may be created in Python to hide the use of ST objects.

The *parser* module defines functions for a few distinct purposes. The most important purposes are to create ST objects and to convert ST objects to other representations such as parse trees and compiled code objects, but there are also functions which serve to query the type of parse tree represented by an ST object.

也参考:

模块*symbol* 代表解析树内部节点的有用常量。

模块*token* 代表解析树叶子节点和测试节点值的函数的有用常量。

32.1.1 创建 ST 对象

ST objects may be created from source code or from a parse tree. When creating an ST object from source, different functions are used to create the `'eval'` and `'exec'` forms.

`parser.`**`expr`**(*source*)
The *expr()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'eval')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.`**`suite`**(*source*)
The *suite()* function parses the parameter *source* as if it were an input to `compile(source, 'file.py', 'exec')`. If the parse succeeds, an ST object is created to hold the internal parse tree representation, otherwise an appropriate exception is raised.

`parser.`**`sequence2st`**(*sequence*)
This function accepts a parse tree represented as a sequence and builds an internal representation if possible. If it can validate that the tree conforms to the Python grammar and all nodes are valid node types in the host version of Python, an ST object is created from the internal representation and returned to the called. If there is a problem creating the internal representation, or if the tree cannot be validated, a *ParserError* exception is raised. An ST object created this way should not be assumed to compile correctly; normal exceptions raised by compilation may still be initiated when the ST object is passed to *compilest()*. This may indicate problems not related to syntax (such as a *MemoryError* exception), but may also be due to constructs such as the result of parsing `del f(0)`, which escapes the Python parser but is checked by the bytecode compiler.

Sequences representing terminal tokens may be represented as either two-element lists of the form `(1, 'name')` or as three-element lists of the form `(1, 'name', 56)`. If the third element is present, it is assumed to be a

valid line number. The line number may be specified for any subset of the terminal symbols in the input tree.

parser.**tuple2st**(*sequence*)
This is the same function as *sequence2st()*. This entry point is maintained for backward compatibility.

32.1.2 转换 ST 对象

ST objects, regardless of the input used to create them, may be converted to parse trees represented as list- or tuple-trees, or may be compiled into executable code objects. Parse trees may be extracted with or without line numbering information.

parser.**st2list**(*st*, *line_info=False*, *col_info=False*)
This function accepts an ST object from the caller in *st* and returns a Python list representing the equivalent parse tree. The resulting list representation can be used for inspection or the creation of a new parse tree in list form. This function does not fail so long as memory is available to build the list representation. If the parse tree will only be used for inspection, *st2tuple()* should be used instead to reduce memory consumption and fragmentation. When the list representation is required, this function is significantly faster than retrieving a tuple representation and converting that to nested lists.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. Note that the line number provided specifies the line on which the token *ends*. This information is omitted if the flag is false or omitted.

parser.**st2tuple**(*st*, *line_info=False*, *col_info=False*)
This function accepts an ST object from the caller in *st* and returns a Python tuple representing the equivalent parse tree. Other than returning a tuple instead of a list, this function is identical to *st2list()*.

If *line_info* is true, line number information will be included for all terminal tokens as a third element of the list representing the token. This information is omitted if the flag is false or omitted.

parser.**compilest**(*st*, *filename='<syntax-tree>'*)
The Python byte compiler can be invoked on an ST object to produce code objects which can be used as part of a call to the built-in *exec()* or *eval()* functions. This function provides the interface to the compiler, passing the internal parse tree from *st* to the parser, using the source file name specified by the *filename* parameter. The default value supplied for *filename* indicates that the source was an ST object.

Compiling an ST object may result in exceptions related to compilation; an example would be a *SyntaxError* caused by the parse tree for `del f(0)`: this statement is considered legal within the formal grammar for Python but is not a legal language construct. The *SyntaxError* raised for this condition is actually generated by the Python byte-compiler normally, which is why it can be raised at this point by the *parser* module. Most causes of compilation failure can be diagnosed programmatically by inspection of the parse tree.

32.1.3 Queries on ST Objects

Two functions are provided which allow an application to determine if an ST was created as an expression or a suite. Neither of these functions can be used to determine if an ST was created from source code via *expr()* or *suite()* or from a parse tree via *sequence2st()*.

parser.**isexpr**(*st*)
When *st* represents an `'eval'` form, this function returns `True`, otherwise it returns `False`. This is useful, since code objects normally cannot be queried for this information using existing built-in functions. Note that the code objects created by *compilest()* cannot be queried like this either, and are identical to those created by the built-in *compile()* function.

parser.**issuite**(*st*)
This function mirrors *isexpr()* in that it reports whether an ST object represents an `'exec'` form, commonly

known as a "suite." It is not safe to assume that this function is equivalent to `not isexpr(st)`, as additional syntactic fragments may be supported in the future.

32.1.4 异常和错误处理

The parser module defines a single exception, but may also pass other built-in exceptions from other portions of the Python runtime environment. See each function for information about the exceptions it can raise.

exception parser.**ParserError**
Exception raised when a failure occurs within the parser module. This is generally produced for validation failures rather than the built-in *SyntaxError* raised during normal parsing. The exception argument is either a string describing the reason of the failure or a tuple containing a sequence causing the failure from a parse tree passed to *sequence2st()* and an explanatory string. Calls to *sequence2st()* need to be able to handle either type of exception, while calls to other functions in the module will only need to be aware of the simple string values.

Note that the functions *compilest()*, *expr()*, and *suite()* may raise exceptions which are normally raised by the parsing and compilation process. These include the built in exceptions *MemoryError*, *OverflowError*, *SyntaxError*, and *SystemError*. In these cases, these exceptions carry all the meaning normally associated with them. Refer to the descriptions of each function for detailed information.

32.1.5 ST 对象

Ordered and equality comparisons are supported between ST objects. Pickling of ST objects (using the *pickle* module) is also supported.

parser.**STType**
The type of the objects returned by *expr()*, *suite()* and *sequence2st()*.

ST 对象具有以下方法:

ST.**compile**(*filename='<syntax-tree>'*)
和 `compilest(st, filename)` 相同.

ST.**isexpr**()
和 `isexpr(st)` 相同。

ST.**issuite**()
和 `issuite(st)` 相同。

ST.**tolist**(*line_info=False*, *col_info=False*)
和 `st2list(st, line_info, col_info)` 相同。

ST.**totuple**(*line_info=False*, *col_info=False*)
和 `st2tuple(st, line_info, col_info)` 相同。

32.1.6 示例: `compile()` 的模拟

While many useful operations may take place between parsing and bytecode generation, the simplest operation is to do nothing. For this purpose, using the *parser* module to produce an intermediate data structure is equivalent to the code

```
>>> code = compile('a + 5', 'file.py', 'eval')
>>> a = 5
>>> eval(code)
10
```

The equivalent operation using the *parser* module is somewhat longer, and allows the intermediate internal parse tree to be retained as an ST object:

```
>>> import parser
>>> st = parser.expr('a + 5')
>>> code = st.compile('file.py')
>>> a = 5
>>> eval(code)
10
```

An application which needs both ST and code objects can package this code into readily available functions:

```
import parser

def load_suite(source_string):
    st = parser.suite(source_string)
    return st, st.compile()

def load_expression(source_string):
    st = parser.expr(source_string)
    return st, st.compile()
```

32.2 ast --- 抽象语法树

源代码： Lib/ast.py

---

ast 模块帮助 Python 程序处理 Python 语法的抽象语法树。抽象语法或许会随着 Python 的更新发布而改变；该模块能够帮助理解当前语法在编程层面的样貌。

抽象语法树可通过将*ast.PyCF_ONLY_AST* 作为旗标传递给*compile()* 内置函数来生成，或是使用此模块中提供的*parse()* 辅助函数。返回结果将是一个对象树，，其中的类都继承自*ast.AST*。抽象语法树可被内置的*compile()* 函数编译为一个 Python 代码对象。

32.2.1 抽象文法

抽象文法目前定义如下

```
-- ASDL's 4 builtin types are:
-- identifier, int, string, constant

module Python
{
    mod = Module(stmt* body, type_ignore* type_ignores)
        | Interactive(stmt* body)
        | Expression(expr body)
        | FunctionType(expr* argtypes, expr returns)

    stmt = FunctionDef(identifier name, arguments args,
                       stmt* body, expr* decorator_list, expr? returns,
                       string? type_comment)
          | AsyncFunctionDef(identifier name, arguments args,
                             stmt* body, expr* decorator_list, expr? returns,
```

(下页继续)

(繼續上一頁)

```
                              string? type_comment)

          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             stmt* body,
             expr* decorator_list)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value, string? type_comment)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without parens
          | AnnAssign(expr target, expr annotation, expr? value, int simple)

          -- use 'orelse' because else is a keyword in target languages
          | For(expr target, expr iter, stmt* body, stmt* orelse, string? type_
↪comment)
          | AsyncFor(expr target, expr iter, stmt* body, stmt* orelse, string? type_
↪comment)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body, string? type_comment)
          | AsyncWith(withitem* items, stmt* body, string? type_comment)

          | Raise(expr? exc, expr? cause)
          | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
          | Assert(expr test, expr? msg)

          | Import(alias* names)
          | ImportFrom(identifier? module, alias* names, int? level)

          | Global(identifier* names)
          | Nonlocal(identifier* names)
          | Expr(expr value)
          | Pass | Break | Continue

          -- col_offset is the byte offset in the utf8 string the parser uses
          attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

          -- BoolOp() can use left & right?
    expr = BoolOp(boolop op, expr* values)
          | NamedExpr(expr target, expr value)
          | BinOp(expr left, operator op, expr right)
          | UnaryOp(unaryop op, expr operand)
          | Lambda(arguments args, expr body)
          | IfExp(expr test, expr body, expr orelse)
          | Dict(expr* keys, expr* values)
          | Set(expr* elts)
          | ListComp(expr elt, comprehension* generators)
          | SetComp(expr elt, comprehension* generators)
          | DictComp(expr key, expr value, comprehension* generators)
          | GeneratorExp(expr elt, comprehension* generators)
          -- the grammar constrains where yield expressions can occur
          | Await(expr value)
```

(下页继续)

(繼續上一頁)

```
          | Yield(expr? value)
          | YieldFrom(expr value)
          -- need sequences for compare to distinguish between
          -- x < 4 < 3 and (x < 4) < 3
          | Compare(expr left, cmpop* ops, expr* comparators)
          | Call(expr func, expr* args, keyword* keywords)
          | FormattedValue(expr value, int? conversion, expr? format_spec)
          | JoinedStr(expr* values)
          | Constant(constant value, string? kind)

          -- the following expression can appear in assignment context
          | Attribute(expr value, identifier attr, expr_context ctx)
          | Subscript(expr value, expr slice, expr_context ctx)
          | Starred(expr value, expr_context ctx)
          | Name(identifier id, expr_context ctx)
          | List(expr* elts, expr_context ctx)
          | Tuple(expr* elts, expr_context ctx)

          -- can appear only in Subscript
          | Slice(expr? lower, expr? upper, expr? step)

           -- col_offset is the byte offset in the utf8 string the parser uses
          attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    expr_context = Load | Store | Del

    boolop = And | Or

    operator = Add | Sub | Mult | MatMult | Div | Mod | Pow | LShift
                 | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs, int is_async)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset, int? end_lineno, int? end_
↪col_offset)

    arguments = (arg* posonlyargs, arg* args, arg? vararg, arg* kwonlyargs,
                 expr* kw_defaults, arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation, string? type_comment)
           attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    -- keyword arguments supplied to call (NULL identifier for **kwargs)
    keyword = (identifier? arg, expr value)
               attributes (int lineno, int col_offset, int? end_lineno, int? end_col_
↪offset)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)
```

(下页继续)

(繼續上一頁)

```
    withitem = (expr context_expr, expr? optional_vars)

    type_ignore = TypeIgnore(int lineno, string tag)
}
```

32.2.2 节点类

class ast.**AST**

这是所有 AST 节点类的基类。实际上，这些节点类派生自 `Parser/Python.asdl` 文件，其中定义的语法树示例如下。它们在 C 语言模块 `_ast` 中定义，并被导出至*ast* 模块。

抽象语法定义的每个左侧符号 (比方说，`ast.stmt` 或者 `ast.expr`) 定义了一个类。另外，在抽象语法定义的右侧，对每一个构造器也定义了一个类；这些类继承自树左侧的类。比如，*ast.BinOp* 继承自 `ast.expr`。对于多分支产生式 (也就是" 和规则")，树右侧的类是抽象的；只有特定构造器结点的实例能被构造。

_fields

每个具体类都有个属性*_fields*, 用来给出所有子节点的名字。

每个具体类的实例对它每个子节点都有一个属性，对应类型如文法中所定义。比如，*ast.BinOp* 的实例有个属性 `left`，类型是 `ast.expr`.

如果这些属性在文法中标记为可选（使用问号），对应值可能会是 `None`。如果这些属性有零或多个（用星号标记），对应值会用 Python 的列表来表示。所有可能的属性必须在用*compile()* 编译得到 AST 时给出，且是有效的值。

lineno
col_offset
end_lineno
end_col_offset

Instances of `ast.expr` and `ast.stmt` subclasses have *lineno*, *col_offset*, *end_lineno*, and *end_col_offset* attributes. The *lineno* and *end_lineno* are the first and last line numbers of the source text span (1-indexed so the first line is line 1), and the *col_offset* and *end_col_offset* are the corresponding UTF-8 byte offsets of the first and last tokens that generated the node. The UTF-8 offset is recorded because the parser uses UTF-8 internally.

注意编译器不需要结束位置，所以结束位置是可选的。结束偏移在最后一个符号 * 之后 *，例如你可以通过 `source_line[node.col_offset : node.end_col_offset]` 获得一个单行表达式节点的源码片段。

一个类的构造器 `ast.T` 像下面这样 parse 它的参数。

- 如果有位置参数，它们必须和 `T._fields` 中的元素一样多；他们会像这些名字的属性一样被赋值。
- 如果有关键字参数，它们必须被设为和给定值同名的属性。

比方说，要创建和填充节点*ast.UnaryOp*，你得用

```
node = ast.UnaryOp()
node.op = ast.USub()
node.operand = ast.Constant()
node.operand.value = 5
node.operand.lineno = 0
node.operand.col_offset = 0
```

(下页继续)

(繼續上一頁)

```
node.lineno = 0
node.col_offset = 0
```

或者更紧凑点

```
node = ast.UnaryOp(ast.USub(), ast.Constant(5, lineno=0, col_offset=0),
                   lineno=0, col_offset=0)
```

3.8 版更變: 类*ast.Constant* 现在用于所有常量。

3.9 版更變: 简单索引由它们的值表示，扩展切片表示为元组。

3.8 版後已🄵用: 旧的类 `ast.Num`、`ast.Str`、`ast.Bytes`、`ast.NameConstant` 和 `ast.Ellipsis` 仍然有效，但是它们会在未来的 Python 版本中被移除。同时，实例化它们会返回一个不同类的实例。

3.9 版後已🄵用: 旧的类 `ast.Index` 和 `ast.ExtSlice` 仍然有效，但是它们会在未来的 Python 版本中被移除。同时，实例化它们会返回一个不同类的实例。

備🄵: 在此显示的特定节点类的描述最初是改编自杰出的 Green Tree Snakes 项目及其所有贡献者。

字面值

class `ast.`**Constant**(*value*)

一个常量。`Constant` 字面值的 `value` 属性即为其代表的 Python 对象。它可以代表简单的数字，字符串或者 `None` 对象，但是也可以代表所有元素都是常量的不可变容器，例如元组（tuple）或冻结集合（frozenset）。

```
>>> print(ast.dump(ast.parse('123', mode='eval'), indent=4))
Expression(
    body=Constant(value=123))
```

class `ast.`**FormattedValue**(*value*, *conversion*, *format_spec*)

节点是以一个 f-字符串形式的格式化字段来代表的。如果该字符串只包含单个格式化字段而没有任何其他内容则节点可以被隔离，否则它将在*JoinedStr* 中出现。

- `value` 为任意的表达式节点（如一个字面值、变量或函数调用）。
- `conversion` 是一个整数:
  - -1: 无格式化
  - 115: `!s` 字符串格式化
  - 114: `!r` repr 格式化
  - 97: `!a` ascii 格式化
- `format_spec` 是一个代表值的格式化的*JoinedStr* 节点，或者如果未指定格式则为 `None`。`conversion` 和 `format_spec` 可以被同时设置。

class `ast.`**JoinedStr**(*values*)

一个 f-字符串，由一系列*FormattedValue* 和*Constant* 节点组成。

```
>>> print(ast.dump(ast.parse('f"sin({a}) is {sin(a):.3}"', mode='eval'),␣
↪indent=4))
Expression(
```

(下页继续)

(繼續上一頁)

```
    body=JoinedStr(
        values=[
            Constant(value='sin('),
            FormattedValue(
                value=Name(id='a', ctx=Load()),
                conversion=-1),
            Constant(value=') is '),
            FormattedValue(
                value=Call(
                    func=Name(id='sin', ctx=Load()),
                    args=[
                        Name(id='a', ctx=Load())],
                    keywords=[]),
                conversion=-1,
                format_spec=JoinedStr(
                    values=[
                        Constant(value='.3')]))]))
```

class ast.**List**(*elts*, *ctx*)

class ast.**Tuple**(*elts*, *ctx*)

一个列表或元组。elts 保存一个代表元素的节点的列表。ctx 在容器为赋值的目标时 (如 (x, y)=something) 是*Store*，否则是*Load*。

```
>>> print(ast.dump(ast.parse('[1, 2, 3]', mode='eval'), indent=4))
Expression(
    body=List(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
>>> print(ast.dump(ast.parse('(1, 2, 3)', mode='eval'), indent=4))
Expression(
    body=Tuple(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)],
        ctx=Load()))
```

class ast.**Set**(*elts*)

一个集合。elts 保存一个代表集合的元组的节点的列表。

```
>>> print(ast.dump(ast.parse('{1, 2, 3}', mode='eval'), indent=4))
Expression(
    body=Set(
        elts=[
            Constant(value=1),
            Constant(value=2),
            Constant(value=3)]))
```

class ast.**Dict**(*keys*, *values*)

一个字典。keys 和 values 保存分别代表键和值的节点的列表，按照匹配的顺序（即当调用 dictionary.keys() 和 dictionary.values() 时将返回的结果）。

当使用字典字面值进行字典解包操作时要扩展的表达式放入 values 列表，并将 None 放入 keys 的对应位置。

```
>>> print(ast.dump(ast.parse('{"a":1, **d}', mode='eval'), indent=4))
Expression(
    body=Dict(
        keys=[
            Constant(value='a'),
            None],
        values=[
            Constant(value=1),
            Name(id='d', ctx=Load())]))
```

变量

class ast.**Name**(*id*, *ctx*)

一个变量名。id 将名称保存为字符串，而 ctx 为下列类型之一。

class ast.**Load**
class ast.**Store**
class ast.**Del**

变量引用可被用来载入一个变量的值，为其赋一个新值，或是将其删除。变量引用会给出一个上下文来区分这几种情况。

```
>>> print(ast.dump(ast.parse('a'), indent=4))
Module(
    body=[
        Expr(
            value=Name(id='a', ctx=Load()))],
    type_ignores=[])

>>> print(ast.dump(ast.parse('a = 1'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store())],
            value=Constant(value=1))],
    type_ignores=[])

>>> print(ast.dump(ast.parse('del a'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='a', ctx=Del())])],
    type_ignores=[])
```

class ast.**Starred**(*value*, *ctx*)

一个 *var 变量引用。value 保存变量，通常为一个*Name* 节点。此类型必须在构建*Call* 节点并传入 *args 时被使用。

```
>>> print(ast.dump(ast.parse('a, *b = it'), indent=4))
Module(
    body=[
        Assign(
            targets=[
                Tuple(
```

(下页继续)

(繼續上一頁)

```
                    elts=[
                        Name(id='a', ctx=Store()),
                        Starred(
                            value=Name(id='b', ctx=Store()),
                            ctx=Store())],
                    ctx=Store())],
            value=Name(id='it', ctx=Load()))],
    type_ignores=[])
```

表达式

class ast.**Expr**(*value*)

当一个表达式，例如函数调用，本身作为一个语句出现并且其返回值未被使用或存储时，它会被包装在此容器中。value 保存本节中的其他节点之一，一个*Constant*, *Name*, *Lambda*, *Yield* 或者*YieldFrom* 节点。

```
>>> print(ast.dump(ast.parse('-a'), indent=4))
Module(
    body=[
        Expr(
            value=UnaryOp(
                op=USub(),
                operand=Name(id='a', ctx=Load()))))],
    type_ignores=[])
```

class ast.**UnaryOp**(*op*, *operand*)

一个单目运算。op 是运算符，而 operand 是任意表达式节点。

class ast.**UAdd**
class ast.**USub**
class ast.**Not**
class ast.**Invert**

单目运算符对应的形符。*Not* 是 not 关键字，*Invert* 是 ~ 运算符。

```
>>> print(ast.dump(ast.parse('not x', mode='eval'), indent=4))
Expression(
    body=UnaryOp(
        op=Not(),
        operand=Name(id='x', ctx=Load())))
```

class ast.**BinOp**(*left*, *op*, *right*)

一个双目运算（如相加或相减）。op 是运算符，而 left 和 right 是任意表达式节点。

```
>>> print(ast.dump(ast.parse('x + y', mode='eval'), indent=4))
Expression(
    body=BinOp(
        left=Name(id='x', ctx=Load()),
        op=Add(),
        right=Name(id='y', ctx=Load())))
```

class ast.**Add**
class ast.**Sub**
class ast.**Mult**
class ast.**Div**
class ast.**FloorDiv**

class ast.**Mod**
class ast.**Pow**
class ast.**LShift**
class ast.**RShift**
class ast.**BitOr**
class ast.**BitXor**
class ast.**BitAnd**
class ast.**MatMult**
双目运算符对应的形符。

class ast.**BoolOp**(*op*, *values*)
一个布尔运算，'or' 或者'and'。op 是*Or* 或者*And*。values 是参与运算的值。具有相同运算符的连续运算，如 a or b or c，会被折叠为具有多个值的单个节点。

这不包括 not，它属于*UnaryOp*。

```
>>> print(ast.dump(ast.parse('x or y', mode='eval'), indent=4))
Expression(
    body=BoolOp(
        op=Or(),
        values=[
            Name(id='x', ctx=Load()),
            Name(id='y', ctx=Load())]))
```

class ast.**And**
class ast.**Or**
布尔运算符对应的形符。

class ast.**Compare**(*left*, *ops*, *comparators*)
两个或更多值之间的比较运算。left 是参加比较的第一个值，ops 是由运算符组成的列表，而 comparators 是由参加比较的第一个元素之后的值组成的列表。

```
>>> print(ast.dump(ast.parse('1 <= a < 10', mode='eval'), indent=4))
Expression(
    body=Compare(
        left=Constant(value=1),
        ops=[
            LtE(),
            Lt()],
        comparators=[
            Name(id='a', ctx=Load()),
            Constant(value=10)]))
```

class ast.**Eq**
class ast.**NotEq**
class ast.**Lt**
class ast.**LtE**
class ast.**Gt**
class ast.**GtE**
class ast.**Is**
class ast.**IsNot**
class ast.**In**
class ast.**NotIn**
比较运算符对应的形符。

class ast.**Call**(*func*, *args*, *keywords*, *starargs*, *kwargs*)
一个函数调用。func 是函数，它通常是一个*Name* 或*Attribute* 对象。对于其参数:

- `args` 保存由按位置传入的参数组成的列表。
- `keywords` 保存了一个代表以关键字传入的参数的*keyword* 对象的列表。

当创建一个 `Call` 节点时，需要有 `args` 和 `keywords`，但它们可以为空列表。`starargs` 和 `kwargs` 是可选的。

```
>>> print(ast.dump(ast.parse('func(a, b=c, *d, **e)', mode='eval'), indent=4))
Expression(
    body=Call(
        func=Name(id='func', ctx=Load()),
        args=[
            Name(id='a', ctx=Load()),
            Starred(
                value=Name(id='d', ctx=Load()),
                ctx=Load())],
        keywords=[
            keyword(
                arg='b',
                value=Name(id='c', ctx=Load())),
            keyword(
                value=Name(id='e', ctx=Load()))]))
```

class `ast.`**keyword**(*arg*, *value*)
传给函数调用或类定义的关键字参数。`arg` 是形参名称对应的原始字符串，`value` 是要传入的节点。

class `ast.`**IfExp**(*test*, *body*, *orelse*)
一个表达式例如 `a if b else c`。每个字段保存一个单独节点，因而在下面的示例中，三个节点均为*Name* 节点。

```
>>> print(ast.dump(ast.parse('a if b else c', mode='eval'), indent=4))
Expression(
    body=IfExp(
        test=Name(id='b', ctx=Load()),
        body=Name(id='a', ctx=Load()),
        orelse=Name(id='c', ctx=Load())))
```

class `ast.`**Attribute**(*value*, *attr*, *ctx*)
属性访问，例如 `d.keys`。`value` 是一个节点，通常为*Name*。`attr` 是一个给出属性名称的纯字符串，而 `ctx` 根据属性操作的方式可以为*Load*, *Store* 或*Del*。

```
>>> print(ast.dump(ast.parse('snake.colour', mode='eval'), indent=4))
Expression(
    body=Attribute(
        value=Name(id='snake', ctx=Load()),
        attr='colour',
        ctx=Load()))
```

class `ast.`**NamedExpr**(*target*, *value*)

一个带名称的表达式。此 AST 节点是由赋值表达式运算符（或称海象运算符）产生的。与第一个参数可以有多个节点的*Assign* 节点不同，在此情况下 `target` 和 `value` 都必须为单独节点。

```
>>> print(ast.dump(ast.parse('(x := 4)', mode='eval'), indent=4))
Expression(
    body=NamedExpr(
        target=Name(id='x', ctx=Store()),
        value=Constant(value=4)))
```

抽取

class ast.**Subscript**(*value*, *slice*, *ctx*)

抽取操作，如 `l[1]`。`value` 是被抽取的对象（通常为序列或映射）。`slice` 是索引号、切片或键。它可以是一个包含*Slice* 的*Tuple*。`ctx` 根据抽取所执行的操作可以为*Load*, *Store* 或*Del*。

```
>>> print(ast.dump(ast.parse('l[1:2, 3]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Tuple(
            elts=[
                Slice(
                    lower=Constant(value=1),
                    upper=Constant(value=2)),
                Constant(value=3)],
            ctx=Load()),
        ctx=Load()))
```

class ast.**Slice**(*lower*, *upper*, *step*)

常规切片 (形式如 `lower:upper` 或 `lower:upper:step`)。只能在*Subscript* 的 *slice* 字段内部出现，可以是直接切片对象或是作为*Tuple* 的元素。

```
>>> print(ast.dump(ast.parse('l[1:2]', mode='eval'), indent=4))
Expression(
    body=Subscript(
        value=Name(id='l', ctx=Load()),
        slice=Slice(
            lower=Constant(value=1),
            upper=Constant(value=2)),
        ctx=Load()))
```

推导式

class ast.**ListComp**(*elt*, *generators*)
class ast.**SetComp**(*elt*, *generators*)
class ast.**GeneratorExp**(*elt*, *generators*)
class ast.**DictComp**(*key*, *value*, *generators*)

列表和集合推导式、生成器表达式以及字典推导式。`elt` (或 `key` 和 `value`) 是一个代表将针对每个条目被求值的部分的单独节点。

`generators` 是一个由*comprehension* 节点组成的列表。

```
>>> print(ast.dump(ast.parse('[x for x in numbers]', mode='eval'), indent=4))
Expression(
    body=ListComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                ifs=[],
                is_async=0)]))
>>> print(ast.dump(ast.parse('{x: x**2 for x in numbers}', mode='eval'),␣
↪indent=4))
```

(下页继续)

(繼續上一頁)

```
Expression(
    body=DictComp(
        key=Name(id='x', ctx=Load()),
        value=BinOp(
            left=Name(id='x', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                ifs=[],
                is_async=0)]))
>>> print(ast.dump(ast.parse('{x for x in numbers}', mode='eval'), indent=4))
Expression(
    body=SetComp(
        elt=Name(id='x', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='x', ctx=Store()),
                iter=Name(id='numbers', ctx=Load()),
                ifs=[],
                is_async=0)]))
```

class ast.**comprehension**(*target*, *iter*, *ifs*, *is_async*)

推导式中的一个 `for` 子句。`target` 是针对每个元素使用的引用——通常为一个*Name* 或*Tuple* 节点。`iter` 是要执行迭代的对象。`ifs` 是一个由测试表达式组成的列表：每个 `for` 子句都可以拥有多个 `ifs`。

`is_async` 表明推导式是异步的 (使用 `async for` 而不是 `for`)。它的值是一个整数 (0 或 1)。

```
>>> print(ast.dump(ast.parse('[ord(c) for line in file for c in line]', mode='eval
↪'),
...                indent=4)) # Multiple comprehensions in one.
Expression(
    body=ListComp(
        elt=Call(
            func=Name(id='ord', ctx=Load()),
            args=[
                Name(id='c', ctx=Load())],
            keywords=[]),
        generators=[
            comprehension(
                target=Name(id='line', ctx=Store()),
                iter=Name(id='file', ctx=Load()),
                ifs=[],
                is_async=0),
            comprehension(
                target=Name(id='c', ctx=Store()),
                iter=Name(id='line', ctx=Load()),
                ifs=[],
                is_async=0)]))

>>> print(ast.dump(ast.parse('(n**2 for n in it if n>5 if n<10)', mode='eval'),
...                indent=4)) # generator comprehension
Expression(
    body=GeneratorExp(
```

(下页继续)

(繼續上一頁)

```
        elt=BinOp(
            left=Name(id='n', ctx=Load()),
            op=Pow(),
            right=Constant(value=2)),
        generators=[
            comprehension(
                target=Name(id='n', ctx=Store()),
                iter=Name(id='it', ctx=Load()),
                ifs=[
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    Compare(
                        left=Name(id='n', ctx=Load()),
                        ops=[
                            Lt()],
                        comparators=[
                            Constant(value=10)])],
                is_async=0)]))

>>> print(ast.dump(ast.parse('[i async for i in soc]', mode='eval'),
...                indent=4)) # Async comprehension
Expression(
    body=ListComp(
        elt=Name(id='i', ctx=Load()),
        generators=[
            comprehension(
                target=Name(id='i', ctx=Store()),
                iter=Name(id='soc', ctx=Load()),
                ifs=[],
                is_async=1)]))
```

语句

class ast.**Assign**(*targets*, *value*, *type_comment*)

一次赋值。`targets` 是一个由节点组成的列表，而 `value` 是一个单独节点。

`targets` 中有多个节点表示将同一个值赋给多个目标。解包操作是通过在 `targets` 中放入一个*Tuple* 或*List* 来表示的。

type_comment

`type_comment` 是带有以注释表示的类型标注的可选的字符串。

```
>>> print(ast.dump(ast.parse('a = b = 1'), indent=4)) # Multiple assignment
Module(
    body=[
        Assign(
            targets=[
                Name(id='a', ctx=Store()),
                Name(id='b', ctx=Store())],
            value=Constant(value=1))],
    type_ignores=[])
```

(下页继续)

(繼續上一頁)

```
>>> print(ast.dump(ast.parse('a,b = c'), indent=4)) # Unpacking
Module(
    body=[
        Assign(
            targets=[
                Tuple(
                    elts=[
                        Name(id='a', ctx=Store()),
                        Name(id='b', ctx=Store())],
                    ctx=Store())],
            value=Name(id='c', ctx=Load()))],
    type_ignores=[])
```

class ast.**AnnAssign**(*target*, *annotation*, *value*, *simple*)

一次带有类型标注的赋值。`target` 为单独节点并可以是*Name*, *Attribute* 或*Subscript* 之一。`annotation` 为标注，例如一个*Constant* 或*Name* 节点。`value` 为可选的单独节点。`simple` 为一个布尔整数，其值对于 `target` 中不出现在括号之内因而是纯名称而非表达式的*Name* 节点将为 True。

```
>>> print(ast.dump(ast.parse('c: int'), indent=4))
Module(
    body=[
        AnnAssign(
            target=Name(id='c', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=1)],
    type_ignores=[])

>>> print(ast.dump(ast.parse('(a): int = 1'), indent=4)) # Annotation with␣
↪parenthesis
Module(
    body=[
        AnnAssign(
            target=Name(id='a', ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            value=Constant(value=1),
            simple=0)],
    type_ignores=[])

>>> print(ast.dump(ast.parse('a.b: int'), indent=4)) # Attribute annotation
Module(
    body=[
        AnnAssign(
            target=Attribute(
                value=Name(id='a', ctx=Load()),
                attr='b',
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)],
    type_ignores=[])

>>> print(ast.dump(ast.parse('a[1]: int'), indent=4)) # Subscript annotation
Module(
    body=[
        AnnAssign(
            target=Subscript(
```

(下页继续)

(繼續上一頁)

```
                value=Name(id='a', ctx=Load()),
                slice=Constant(value=1),
                ctx=Store()),
            annotation=Name(id='int', ctx=Load()),
            simple=0)],
    type_ignores=[])
```

class ast.**AugAssign**(*target*, *op*, *value*)

增强赋值，如 `a += 1`。在下面的例子中，`target` 是一个针对 `x` (带有*Store* 上下文) 的*Name* 节点，`op` 为*Add*，而 `value` 是一个值为 1 的*Constant*。

The `target` attribute connot be of class *Tuple* or *List*, unlike the targets of *Assign*.

```
>>> print(ast.dump(ast.parse('x += 2'), indent=4))
Module(
    body=[
        AugAssign(
            target=Name(id='x', ctx=Store()),
            op=Add(),
            value=Constant(value=2))],
    type_ignores=[])
```

class ast.**Raise**(*exc*, *cause*)

一条 `raise` 语句。`exc` 是要引发的异常，对于一个单独的 `raise` 通常为*Call* 或*Name*，或者为 `None`。`cause` 是针对 `raise x from y` 中 `y` 的可选部分。

```
>>> print(ast.dump(ast.parse('raise x from y'), indent=4))
Module(
    body=[
        Raise(
            exc=Name(id='x', ctx=Load()),
            cause=Name(id='y', ctx=Load()))],
    type_ignores=[])
```

class ast.**Assert**(*test*, *msg*)

一条断言。`test` 保存条件，例如为一个*Compare* 节点。`msg` 保存失败消息。

```
>>> print(ast.dump(ast.parse('assert x,y'), indent=4))
Module(
    body=[
        Assert(
            test=Name(id='x', ctx=Load()),
            msg=Name(id='y', ctx=Load()))],
    type_ignores=[])
```

class ast.**Delete**(*targets*)

代表一条 `del` 语句。`targets` 是一个由节点组成的列表，例如*Name*, *Attribute* 或*Subscript* 节点。

```
>>> print(ast.dump(ast.parse('del x,y,z'), indent=4))
Module(
    body=[
        Delete(
            targets=[
                Name(id='x', ctx=Del()),
                Name(id='y', ctx=Del()),
```

(下页继续)

(繼續上一頁)

```
                Name(id='z', ctx=Del())])],
    type_ignores=[])
```

class ast.**Pass**

一条 pass 语句。

```
>>> print(ast.dump(ast.parse('pass'), indent=4))
Module(
    body=[
        Pass()],
    type_ignores=[])
```

其他仅在函数或循环内部可用的语句将在其他小节中描述。

导入

class ast.**Import**(*names*)

一条导入语句。names 是一个由*alias* 节点组成的列表。

```
>>> print(ast.dump(ast.parse('import x,y,z'), indent=4))
Module(
    body=[
        Import(
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')])],
    type_ignores=[])
```

class ast.**ImportFrom**(*module*, *names*, *level*)

代表 from x import y。module 是一个'from' 名称的原始字符串，不带任何前导点号，或者为 None 表示 from . import foo 这样的语句。level 是一个保存相对导入层级的整数（0 表示绝对导入）。

```
>>> print(ast.dump(ast.parse('from y import x,y,z'), indent=4))
Module(
    body=[
        ImportFrom(
            module='y',
            names=[
                alias(name='x'),
                alias(name='y'),
                alias(name='z')],
            level=0)],
    type_ignores=[])
```

class ast.**alias**(*name*, *asname*)

两个形参均为名称的原始字符串。如果要使用常规名称则 asname 可以为 None。

```
>>> print(ast.dump(ast.parse('from ..foo.bar import a as b, c'), indent=4))
Module(
    body=[
        ImportFrom(
            module='foo.bar',
            names=[
```

(下页继续)

(繼續上一頁)

```
                alias(name='a', asname='b'),
                alias(name='c')],
            level=2)],
    type_ignores=[])
```

控制流

備F: 可选的子句如 `else` 如果不存在则会被存储为一个空列表。

class ast.**If**(*test*, *body*, *orelse*)

一条 `if` 语句。`test` 保存一个单独节点，如一个*Compare* 节点。`body` 和 `orelse` 各自保存一个节点列表。

`elif` 子句在 AST 中没有特别的表示形式，而是作为上文介绍的 `orelse` 部分之内的一个额外*If* 节点出现。

```
>>> print(ast.dump(ast.parse("""
... if x:
...     ...
... elif y:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        If(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                If(
                    test=Name(id='y', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))],
                    orelse=[
                        Expr(
                            value=Constant(value=Ellipsis))])])],
    type_ignores=[])
```

class ast.**For**(*target*, *iter*, *body*, *orelse*, *type_comment*)

一个 `for` 循环。`target` 保存循环赋值的变量，是一个单独*Name*, *Tuple* 或*List* 节点。`iter` 保存要被循环的条目，同样也是一个单独节点。`body` 和 `orelse` 包含要执行的节点列表。`orelse` 中的会在循环正常结束而不是通过 `break` 语句结束时被执行。

type_comment

`type_comment` 是带有以注释表示的类型标注的可选的字符串。

```
>>> print(ast.dump(ast.parse("""
... for x in y:
...     ...
```

(下页继续)

(繼續上一頁)

```
... else:
...     ...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='x', ctx=Store()),
            iter=Name(id='y', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])],
    type_ignores=[])
```

class ast.**While**(*test*, *body*, *orelse*)

一个 `while` 循环。`test` 保存条件，如一个*Compare* 节点。

```
>> print(ast.dump(ast.parse("""
... while x:
...     ...
... else:
...     ...
... """), indent=4))
Module(
    body=[
        While(
            test=Name(id='x', ctx=Load()),
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))])],
    type_ignores=[])
```

class ast.**Break**

class ast.**Continue**

`break` 和 `continue` 语句。

```
>>> print(ast.dump(ast.parse("""\
... for a in b:
...     if a > 5:
...         break
...     else:
...         continue
...
... """), indent=4))
Module(
    body=[
        For(
            target=Name(id='a', ctx=Store()),
            iter=Name(id='b', ctx=Load()),
            body=[
                If(
```

(下页继续)

(繼續上一頁)

```
                    test=Compare(
                        left=Name(id='a', ctx=Load()),
                        ops=[
                            Gt()],
                        comparators=[
                            Constant(value=5)]),
                    body=[
                        Break()],
                    orelse=[
                        Continue()])],
            orelse=[])],
    type_ignores=[])
```

class ast.**Try**(*body*, *handlers*, *orelse*, *finalbody*)

try 代码块。所有属性都是要执行的节点列表，除了 handlers，它是一个*ExceptHandler* 节点列表。

```
>>> print(ast.dump(ast.parse("""
... try:
...     ...
... except Exception:
...     ...
... except OtherException as e:
...     ...
... else:
...     ...
... finally:
...     ...
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=Constant(value=Ellipsis))],
            handlers=[
                ExceptHandler(
                    type=Name(id='Exception', ctx=Load()),
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))]),
                ExceptHandler(
                    type=Name(id='OtherException', ctx=Load()),
                    name='e',
                    body=[
                        Expr(
                            value=Constant(value=Ellipsis))])],
            orelse=[
                Expr(
                    value=Constant(value=Ellipsis))],
            finalbody=[
                Expr(
                    value=Constant(value=Ellipsis))])],
    type_ignores=[])
```

class ast.**ExceptHandler**(*type*, *name*, *body*)

一个单独的 except 子句。type 是它将匹配的异常，通常为一个*Name* 节点（或 None 表示捕获全部

的 except: 子句)。name 是一个用于存放异常的别名的原始字符串，或者如果子句没有 as foo 则为 None。body 为一个节点列表。

```
>>> print(ast.dump(ast.parse("""\
... try:
...     a + 1
... except TypeError:
...     pass
... """), indent=4))
Module(
    body=[
        Try(
            body=[
                Expr(
                    value=BinOp(
                        left=Name(id='a', ctx=Load()),
                        op=Add(),
                        right=Constant(value=1)))],
            handlers=[
                ExceptHandler(
                    type=Name(id='TypeError', ctx=Load()),
                    body=[
                        Pass()])],
            orelse=[],
            finalbody=[])],
    type_ignores=[])
```

class ast.**With**(*items*, *body*, *type_comment*)

一个 with 代码块。items 是一个代表上下文管理器的*withitem* 节点列表，而 body 是该上下文中的缩进代码块。

type_comment

type_comment 是带有以注释表示的类型标注的可选的字符串。

class ast.**withitem**(*context_expr*, *optional_vars*)

一个 with 代码块中单独的上下文管理器。context_expr 为上下文管理器，通常为一个*Call* 节点。optional_vars 为一个针对 as foo 部分的*Name*, *Tuple* 或*List*，或者如果未使用别名则为 None。

```
>>> print(ast.dump(ast.parse("""\
... with a as b, c as d:
...    something(b, d)
... """), indent=4))
Module(
    body=[
        With(
            items=[
                withitem(
                    context_expr=Name(id='a', ctx=Load()),
                    optional_vars=Name(id='b', ctx=Store())),
                withitem(
                    context_expr=Name(id='c', ctx=Load()),
                    optional_vars=Name(id='d', ctx=Store()))],
            body=[
                Expr(
                    value=Call(
                        func=Name(id='something', ctx=Load()),
                        args=[
                            Name(id='b', ctx=Load()),
```

(下页继续)

(繼續上一頁)

```
                            Name(id='d', ctx=Load())],
                        keywords=[]))])],
    type_ignores=[])
```

函数与类定义

class ast.**FunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*)

一个函数定义。

- name 是函数名称的原始字符串。
- args 是一个*arguments* 节点。
- body 是函数内部的节点列表。
- decorator_list 是要应用的装饰器列表，最外层的最先保存（即列表中的第一项将最后被应用）。
- returns 是返回标注。

type_comment

type_comment 是带有以注释表示的类型标注的可选的字符串。

class ast.**Lambda**(*args*, *body*)

lambda 是可在表达式内部使用的最小化函数定义。不同于*FunctionDef*，body 是保存一个单独节点。

```
>>> print(ast.dump(ast.parse('lambda x,y: ...'), indent=4))
Module(
    body=[
        Expr(
            value=Lambda(
                args=arguments(
                    posonlyargs=[],
                    args=[
                        arg(arg='x'),
                        arg(arg='y')],
                    kwonlyargs=[],
                    kw_defaults=[],
                    defaults=[]),
                body=Constant(value=Ellipsis)))],
    type_ignores=[])
```

class ast.**arguments**(*posonlyargs*, *args*, *vararg*, *kwonlyargs*, *kw_defaults*, *kwarg*, *defaults*)

函数的参数。

- posonlyargs, args 和 kwonlyargs 均为*arg* 节点的列表。
- vararg 和 kwarg 均为单独的*arg* 节点，指向 *args, **kwargs 形参。
- kw_defaults 是一个由仅限关键字参数默认值组成的列表。如果有一个为 None，则对应的参数为必须的参数。
- defaults 是一个由可按位置传入的参数的默认值组成的列表。如果默认值个数少于参数个数，则它们将对应最后 n 个参数。

class ast.**arg**(*arg*, *annotation*, *type_comment*)

列表中的一个单独参数。arg 为参数名称原始字符串，annotation 为其标注，如一个 Str 或*Name* 节点。

type_comment
type_comment 是一个可选的将注释用作类型标注的字符串。

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... def f(a: 'annotation', b=1, c=2, *d, e, f=3, **g) -> 'return annotation':
...     pass
... """), indent=4))
Module(
    body=[
        FunctionDef(
            name='f',
            args=arguments(
                posonlyargs=[],
                args=[
                    arg(
                        arg='a',
                        annotation=Constant(value='annotation')),
                    arg(arg='b'),
                    arg(arg='c')],
                vararg=arg(arg='d'),
                kwonlyargs=[
                    arg(arg='e'),
                    arg(arg='f')],
                kw_defaults=[
                    None,
                    Constant(value=3)],
                kwarg=arg(arg='g'),
                defaults=[
                    Constant(value=1),
                    Constant(value=2)]),
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())],
            returns=Constant(value='return annotation'))],
    type_ignores=[])
```

class ast.**Return**(*value*)
A return 语句。

```
>>> print(ast.dump(ast.parse('return 4'), indent=4))
Module(
    body=[
        Return(
            value=Constant(value=4))],
    type_ignores=[])
```

class ast.**Yield**(*value*)
class ast.**YieldFrom**(*value*)
一个 yield 或 yield from 表达式。因为这些属于表达式，所以如果发回的值未被使用则必须将它们包装在*Expr* 节点中。

```
>>> print(ast.dump(ast.parse('yield x'), indent=4))
Module(
    body=[
```

(下页继续)

(繼續上一頁)

```
        Expr(
            value=Yield(
                value=Name(id='x', ctx=Load()))))],
    type_ignores=[])

>>> print(ast.dump(ast.parse('yield from x'), indent=4))
Module(
    body=[
        Expr(
            value=YieldFrom(
                value=Name(id='x', ctx=Load()))))],
    type_ignores=[])
```

class ast.**Global**(*names*)
class ast.**Nonlocal**(*names*)

global 和 nonlocal 语句。names 为一个由原始字符串组成的列表。

```
>>> print(ast.dump(ast.parse('global x,y,z'), indent=4))
Module(
    body=[
        Global(
            names=[
                'x',
                'y',
                'z'])],
    type_ignores=[])

>>> print(ast.dump(ast.parse('nonlocal x,y,z'), indent=4))
Module(
    body=[
        Nonlocal(
            names=[
                'x',
                'y',
                'z'])],
    type_ignores=[])
```

class ast.**ClassDef**(*name*, *bases*, *keywords*, *starargs*, *kwargs*, *body*, *decorator_list*)

一个类定义。

- name 为类名称的原始字符串。
- bases 为一个由显式指明的基类节点组成的列表。
- keywords 为一个由*keyword* 节点组成的列表，主要用于‘元类’。其他关键字将被传给这个元类，参见 PEP-3115。
- starargs 和 kwargs 各为一个单独的节点，与在函数调用中的一致。starargs 将被展开加入到基类的列表中，而 kwargs 将被传给元类。
- body 是一个由代表类定义内部代码的节点组成的列表。
- decorator_list 是一个节点的列表，与*FunctionDef* 中的一致。

```
>>> print(ast.dump(ast.parse("""\
... @decorator1
... @decorator2
... class Foo(base1, base2, metaclass=meta):
```

(下页继续)

(繼續上一頁)

```
...     pass
... """), indent=4))
Module(
    body=[
        ClassDef(
            name='Foo',
            bases=[
                Name(id='base1', ctx=Load()),
                Name(id='base2', ctx=Load())],
            keywords=[
                keyword(
                    arg='metaclass',
                    value=Name(id='meta', ctx=Load()))],
            body=[
                Pass()],
            decorator_list=[
                Name(id='decorator1', ctx=Load()),
                Name(id='decorator2', ctx=Load())])],
    type_ignores=[])
```

async 与 await

class ast.**AsyncFunctionDef**(*name*, *args*, *body*, *decorator_list*, *returns*, *type_comment*)

一个 `async def` 函数定义。具有与*FunctionDef* 相同的字段。

class ast.**Await**(*value*)

一个 `await` 表达式。`value` 是它所等待的值。仅在*AsyncFunctionDef* 的函数体内可用。

```
>>> print(ast.dump(ast.parse("""\
... async def f():
...     await other_func()
... """), indent=4))
Module(
    body=[
        AsyncFunctionDef(
            name='f',
            args=arguments(
                posonlyargs=[],
                args=[],
                kwonlyargs=[],
                kw_defaults=[],
                defaults=[]),
            body=[
                Expr(
                    value=Await(
                        value=Call(
                            func=Name(id='other_func', ctx=Load()),
                            args=[],
                            keywords=[])))],
            decorator_list=[])],
    type_ignores=[])
```

class ast.**AsyncFor**(*target*, *iter*, *body*, *orelse*, *type_comment*)

class ast.**AsyncWith**(*items*, *body*, *type_comment*)

`async for` 循环和 `async with` 上下文管理器。它们分别具有与*For* 和*With* 相同的字段。仅在*AsyncFunctionDef* 的函数体内可用。

備𝔽: 当一个字符串由*ast.parse()* 来解析时，所返回的树中的运算符节点 (为 ast.operator, ast.unaryop, ast.cmpop, ast.boolop 和 ast.expr_context 的子类) 将均为单例对象。对其中某一个 (例如*ast.Add*) 的修改将反映到同一个值所出现的其他位置上。

32.2.3 ast 中的辅助函数

除了节点类，*ast* 模块里为遍历抽象语法树定义了这些工具函数和类:

ast.**parse**(*source*, *filename='<unknown>'*, *mode='exec'*, *, *type_comments=False*, *feature_version=None*)
把源码解析为 AST 节点。和 compile(source, filename, mode,ast.PyCF_ONLY_AST) 等价。

如果给出 type_comments=True，解析器会被修改以检查并返回 **PEP 484** 和 **PEP 526** 所描述的类型注释。这相当于将*ast.PyCF_TYPE_COMMENTS* 添加到传给*compile()* 的旗标中。这将报告针对未在正确放置类型注释的语法错误。没有这个旗标，类型注释将被忽略，而指定 AST 节点上的 type_comment 字段将总是为 None。此外，# type: ignore 注释的位置将作为 Module 的 type_ignores 属性被返回（在其他情况下则总是为空列表）。

并且，如果 mode 为 'func_type'，则输入语法会进行与 **PEP 484** ”签名类型注释” 对应的修改，例如 (str, int) -> List[str]。

此外，将 feature_version 设为元组 (major, minor) 将会尝试使用该 will attempt to parse using that Python 版本的语法来进行解析。目前 major 必须等于 3。例如，设置 feature_version=(3, 4) 将允许使用 async 和 await 作为变量名。最低受支持版本为 (3, 4)；最高则为 sys.version_info[0:2]。

If source contains a null character ('0'), *ValueError* is raised.

> **警告:** Note that successfully parsing source code into an AST object doesn't guarantee that the source code provided is valid Python code that can be executed as the compilation step can raise further *SyntaxError* exceptions. For instance, the source return 42 generates a valid AST node for a return statement, but it cannot be compiled alone (it needs to be inside a function node).
>
> In particular, *ast.parse()* won't do any scoping checks, which the compilation step does.

> **警告:** 足够复杂或是巨大的字符串可能导致 Python 解释器的崩溃，因为 Python 的 AST 编译器是有栈深限制的。

3.8 版更變: 增加了 type_comments, mode='func_type' 和 feature_version。

ast.**unparse**(*ast_obj*)
反向解析一个*ast.AST* 对象并生成一个包含当再次使用*ast.parse()* 解析时将产生同样的*ast.AST* 对象的代码的字符串。

> **警告:** 所产生的代码字符串将不一定与生成*ast.AST* 对象的原始代码完全一致（不带任何编译器优化，例如常量元组/冻结集合等）。

> **警告:** 尝试反向解析一个高度复杂的表达式可能会导致*RecursionError*。

3.9 版新加入.

ast.**literal_eval**(*node_or_string*)
对表达式节点以及包含 Python 字面量或容器的字符串进行安全的求值。传入的字符串或者节点里可能只包含下列的 Python 字面量结构: 字符串，字节对象 (bytes)，数值，元组，列表，字典，集合，布尔值和 None。

这可被用于安全地对包含不受信任来源的 Python 值的字符串进行求值而不必解析这些值本身。它并不能对任意的复杂表达式进行求值，例如涉及运算符或索引操作的表达式。

警告: 足够复杂或是巨大的字符串可能导致 Python 解释器的崩溃，因为 Python 的 AST 编译器是有栈深限制的。

3.2 版更變: 目前支持字节和集合。

3.9 版更變: 现在支持通过 'set()' 创建空集合。

ast.**get_docstring**(*node*, *clean=True*)
返回给定 *node* (必须为*FunctionDef*, *AsyncFunctionDef*, *ClassDef* 或 Module 节点) 的文档字符串，或者如果没有文档字符串则返回 None。如果 *clean* 为真值，则通过*inspect.cleandoc()* 清除文档字符串的缩进。

3.5 版更變: 目前支持*AsyncFunctionDef*

ast.**get_source_segment**(*source*, *node*, *, *padded=False*)
获取生成 *node* 的 *source* 的源代码段。如果丢失了某些位置信息 (lineno, end_lineno, col_offset 或 end_col_offset)，则返回 None。

如果 *padded* 为 True，则多行语句的第一行将以与其初始位置相匹配的空格填充。

3.8 版新加入.

ast.**fix_missing_locations**(*node*)
当你通过*compile()* 来编译节点树时，编译器会准备接受每个支持 lineno 和 col_offset 属性的节点的相应信息。对已生成节点来说这是相当繁琐的，因此这个辅助工具会递归地为尚未设置这些属性的节点添加它们，具体做法是将其设为父节点的对应值。它将从 *node* 开始递归地执行。

ast.**increment_lineno**(*node*, *n=1*)
从 *node* 开始按 *n* 递增节点树中每个节点的行号和结束行号。这在“移动代码”到文件中的不同位置时很有用处。

ast.**copy_location**(*new_node*, *old_node*)
在可能的情况下将源位置 (lineno, col_offset, end_lineno 和 end_col_offset) 从 *old_node* 拷贝到 *new_node*，并返回 *new_node*。

ast.**iter_fields**(*node*)
针对于 *node* 上在 node._fields 中出现的每个字段产生一个 (fieldname, value) 元组。

ast.**iter_child_nodes**(*node*)
产生 *node* 所有的直接子节点，也就是说，所有为节点的字段所有为节点列表的字段条目。

ast.**walk**(*node*)
递归地产生节点树中从 *node* 开始（包括 *node* 本身）的所有下级节点，没有确定的排序方式。这在你仅想要原地修改节点而不关心具体上下文时很有用处。

class ast.**NodeVisitor**
一个遍历抽象语法树并针对所找到的每个节点调用访问器函数的节点访问器基类。该函数可能会返回一个由*visit()* 方法所提供的值。

这个类应当被子类化，并由子类来添加访问器方法。

visit(*node*)
访问一个节点。默认实现会调用名为 `self.visit_classname` 的方法其中 *classname* 为节点类的名称，或者如果该方法不存在则为*generic_visit()*。

generic_visit(*node*)
该访问器会在节点的所有子节点上调用*visit()*。

请注意所有包含自定义访问器方法的节点的子节点将不会被访问除非访问器调用了*generic_visit()* 或是自行访问它们。

如果你想在遍历期间应用对节点的修改则请不要使用*NodeVisitor*。对此目的可使用一个允许修改的特殊访问器 (*NodeTransformer*)。

3.8 版後已F用: `visit_Num()`, `visit_Str()`, `visit_Bytes()`, `visit_NameConstant()` 和 `visit_Ellipsis()` 等方法现在已被弃用且在未来的 Python 版本中将不会再被调用。请添加 `visit_Constant()` 方法来处理所有常量节点。

class ast.**NodeTransformer**
子类*NodeVisitor* 用于遍历抽象语法树，并允许修改节点。

NodeTransformer 将遍历抽象语法树并使用 visitor 方法的返回值去替换或移除旧节点。如果 visitor 方法的返回值为 `None`，则该节点将从其位置移除，否则将替换为返回值。当返回值是原始节点时，无需替换。

如下是一个转换器示例，它将所有出现的名称 (`foo`) 重写为 `data['foo']`:

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        return Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Constant(value=node.id),
            ctx=node.ctx
        )
```

请记住，如果您正在操作的节点具有子节点，则必须先转换其子节点或为该节点调用 `generic_visit()` 方法。

对于属于语句集合（适用于所有语句节点）的节点，访问者还可以返回节点列表而不仅仅是单个节点。

如果*NodeTransformer* 引入了新的（不属于原节点树一部分的）节点而没有给出它们的位置信息（如 `lineno` 等），则应当调用*fix_missing_locations()* 并传入新的子节点树来重新计算位置信息:

```
tree = ast.parse('foo', mode='eval')
new_tree = fix_missing_locations(RewriteName().visit(tree))
```

通常你可以像这样使用转换器:

```
node = YourTransformer().visit(node)
```

ast.**dump**(*node*, *annotate_fields=True*, *include_attributes=False*, *, *indent=None*)
返回 *node* 中树结构的格式化转储。这主要适用于调试目的。如果 *annotate_fields* 为真值（默认），返回的字符串将显示字段的名称和值。如果 *annotate_fields* 为假值，结果字符串将通过省略无歧义的字段名称变得更为紧凑。默认情况下不会转储行号和列偏移等属性。如果需要，可将 *include_attributes* 设为真值。

如果 *indent* 是一个非负整数或者字符串，那么节点树将被美化输出为指定的缩进级别。如果缩进级别为 0、负数或者 `""` 则将只插入换行符。`None` (默认) 将选择最紧凑的表示形式。使用一个正整数将让每个级别缩进相应数量的空格。如果 *indent* 是一个字符串 (如 `"\t"`)，该字符串会被用于缩进每个级别。

3.9 版更變: 添加了 *indent* 选项。

32.2.4 编译器旗标

下列旗标可被传给*compile()* 用来改变程序编译的效果:

ast.**PyCF_ALLOW_TOP_LEVEL_AWAIT**
启用对最高层级 `await`, `async for`, `async with` 以及异步推导式的支持。

3.8 版新加入.

ast.**PyCF_ONLY_AST**
生成并返回一个抽象语法树而不是返回一个已编译的代码对象。

ast.**PyCF_TYPE_COMMENTS**
启用对 **PEP 484** 和 **PEP 526** 风格的类型注释的支持 (`# type: <type>`, `# type: ignore <stuff>`)。

3.8 版新加入.

32.2.5 命令行用法

3.9 版新加入.

ast 模块可以在命令行下作为脚本来执行。具体做法非常简单:

```
python -m ast [-m <mode>] [-a] [infile]
```

可以接受以下选项:

-h, --help
显示帮助信息并退出。

-m <mode>
--mode <mode>
指明哪种代码必须被编译，相当于*parse()* 中的 *mode* 参数。

--no-type-comments
不要解析类型注释。

-a, --include-attributes
包括属性如行号和列偏移。

-i <indent>
--indent <indent>
AST 中节点的缩进（空格数）。

如果指定了 `infile` 则其内容将被解析为 AST 并转储至 stdout。在其他情况下，将从 stdin 读取内容。

也参考:

Green Tree Snakes，一个外部文档资源，包含处理 Python AST 的完整细节。

ASTTokens 会为 Python AST 标注生成它们的源代码中的形符和文本的位置。这对执行源代码转换的工具很有帮助。

leoAst.py 通过在形符和 ast 节点之间插入双向链接统一了 Python 程序基于形符的和基于解析树的视图。

LibCST 将代码解析为一个实体语法树（Concrete Syntax Tree），它看起来像是 ast 树而又保留了所有格式化细节。它对构建自动化重构（codemod）应用和代码质量检查工具很有用处。

Parso 是一个支持错误恢复和不同 Python 版本的（在多个 Python 版本中）往返解析的 Python 解析器。Parso 还能列出你的 Python 文件中的许多错误。

32.3 symtable ——访问编译器的符号表

Source code： Lib/symtable.py

符号表由编译器在生成字节码之前根据 AST 生成。符号表负责计算代码中每个标识符的作用域。*symtable* 提供了一个查看这些表的接口。

32.3.1 符号表的生成

symtable.**symtable**(*code*, *filename*, *compile_type*)
返回 Python 源 代码顶层的*SymbolTable*。*filename* 是代码文件名。*compile_type* 的含义类似*compile()* 的 *mode* 参数。

32.3.2 符号表的查看

class symtable.**SymbolTable**
某个代码块的命名空间表。构造函数不公开。

get_type()
返回符号表的类型。可能是 'class' 、'module' 或 'function'。

get_id()
返回符号表的标识符

get_name()
返回符号表的名称。若为类的符号表则返回类名；若为函数的符号表则为函数名；若是全局符号表则为 'top'（*get_type()* 返回 'module'）。

get_lineno()
返回符号表所代表的代码块中第一行的编号。

is_optimized()
如果符号表中的局部变量可能被优化过，则返回 True。

is_nested()
如果代码块是嵌套类或函数，则返回 True。

has_children()
如果代码块中有嵌套的命名空间，则返回 True。可通过*get_children()* 读取。

get_identifiers()
返回符号表中的符号名列表。

lookup(*name*)
在符号表中查找 *name* 并返回一个*Symbol* 实例。

get_symbols()
返回符号表中所有符号的*Symbol* 实例的列表。

get_children()
返回嵌套符号表的列表。

class symtable.**Function**
函数或方法的命名空间。该类继承自*SymbolTable*。

get_parameters()
返回由函数的参数名组成的元组。

get_locals()
返回函数中局部变量名组成的元组。

get_globals()
返回函数中全局变量名组成的元组。

get_nonlocals()
返回函数中非局部变量名组成的元组。

get_frees()
返回函数中自由变量名组成的元组。

class symtable.**Class**
类的命名空间。继承自*SymbolTable*。

get_methods()
返回类中声明的方法名组成的元组。

class symtable.**Symbol**
SymbolTable 中的数据项，对应于源码中的某个标识符。构造函数不公开。

get_name()
返回符号名

is_referenced()
如果符号在代码块中被引用了，则返回 True。

is_imported()
如果符号是由导入语句创建的，则返回 True。

is_parameter()
如果符号是参数，返回 True。

is_global()
如果符号是全局变量，则返回 True。

is_nonlocal()
如果符号为非局部变量，则返回 True。

is_declared_global()
如果符号用 global 声明为全局变量，则返回 True。

is_local()
如果符号在代码块内是局部变量，则返回 True。

is_annotated()
如果符号带有注解，则返回 True。

3.6 版新加入.

is_free()
如果符号在代码块中被引用，但未赋值，则返回 True。

is_assigned()
如果符号在代码块中赋值，则返回 True。

is_namespace()
如果符号名绑定引入了新的命名空间，则返回 True。

如果符号名用于 function 或 class 语句，则为 True。

例如:

```
>>> table = symtable.symtable("def some_func(): pass", "string", "exec")
>>> table.lookup("some_func").is_namespace()
True
```

注意，一个符号名可以与多个对象绑定。如果结果为 True，则该符号名还可以绑定到其他对象上，比如 int 或 list，且不会引入新的命名空间。

get_namespaces()
返回与符号名绑定的命名空间的列表。

get_namespace()
返回与符号名绑定的命名空间。如果绑定的命名空间超过一个，则会触发*ValueError*。

32.4 symbol --- 与 Python 解析树一起使用的常量

源代码: Lib/symbol.py

此模块提供用于表示解析树内部节点数值的常量。与大多数 Python 不同，这些常量使用小写字符名称。请参阅 Python 发行版中的 Grammar/Grammar 文件来获取该语言语法上下文中对这些名称的定义。这些名称所映射的特定数字值可能会在 Python 版本之间更改。

警告: symbol 模块已弃用并将在未来的 Python 版本中被移除。

此模块还提供了一个额外的数据对象:

symbol.**sym_name**
将此模块中定义的常量的数值映射回名称字符串的字典，允许生成更加人类可读的解析树表示。

32.5 token --- 与 Python 解析树一起使用的常量

源码: Lib/token.py

此模块提供表示解析树（终端令牌）的叶节点的数值的常量。请参阅 Python 发行版中的文件 Grammar/Grammar，以获取语言语法上下文中名称的定义。名称映射到的特定数值可能会在 Python 版本之间更改。

该模块还提供从数字代码到名称和一些函数的映射。这些函数镜像了 Python C 头文件中的定义。

token.**tok_name**
将此模块中定义的常量的数值映射回名称字符串的字典，允许生成更加人类可读的解析树表示。

token.**ISTERMINAL**(*x*)
对终端标记值返回 True。

token.**ISNONTERMINAL**(*x*)
对非终端标记值返回 True。

token.**ISEOF**(*x*)
如果 *x* 是表示输入结束的标记则返回 True。

标记常量是:

token.**ENDMARKER**

token.**NAME**

token.**NUMBER**

token.**STRING**

token.**NEWLINE**

token.**INDENT**

token.**DEDENT**

token.**LPAR**
"(" 的权标值。

token.**RPAR**
")" 的权标值。

token.**LSQB**
"[" 的权标值。

token.**RSQB**
"]" 的权标值。

token.**COLON**
":" 的权标值。

token.**COMMA**
"," 的权标值。

token.**SEMI**
";" 的权标值。

token.**PLUS**
"+" 的权标值。

token.**MINUS**
"-" 的权标值。

token.**STAR**
"*" 的权标值。

token.**SLASH**
"/" 的权标值。

token.**VBAR**
"|" 的权标值。

token.**AMPER**
"&" 的权标值。

token.**LESS**
"<" 的权标值。

token.**GREATER**
">" 的权标值。

token.**EQUAL**
"=" 的形符值。

token.**DOT**
"." 的形符值。

token.**PERCENT**
"%"

token.**LBRACE**
Token value for "{".

token.**RBRACE**
"}" 的形符值。

token.**EQEQUAL**
"==" 的形符值。

token.**NOTEQUAL**
"!=" 的形符值。

token.**LESSEQUAL**
"<=" 的形符值。

token.**GREATEREQUAL**
">=" 的形符值。

token.**TILDE**
"~" 的形符值。

token.**CIRCUMFLEX**
"^" 的形符值。

token.**LEFTSHIFT**
"<<" 的形符值。

token.**RIGHTSHIFT**
">>" 的形符值。

token.**DOUBLESTAR**
"**" 的形符值。

token.**PLUSEQUAL**
"+=" 的形符值。

token.**MINEQUAL**
"-=" 的形符值。

token.**STAREQUAL**
"*=" 的形符值。

token.**SLASHEQUAL**
"/=" 的形符值。

token.**PERCENTEQUAL**
"%=" 的形符值。

token.**AMPEREQUAL**
"&=" 的形符值。

token.**VBAREQUAL**
"|=" 的形符值。

token.**CIRCUMFLEXEQUAL**
"^=" 的形符值。

token.**LEFTSHIFTEQUAL**
"<<=" 的形符值。

token.**RIGHTSHIFTEQUAL**
"`>>=`" 的形符值。

token.**DOUBLESTAREQUAL**
"`**=`" 的形符值。

token.**DOUBLESLASH**
"`//`" 的形符值。

token.**DOUBLESLASHEQUAL**
"`//=`" 的形符值。

token.**AT**
"`@`" 的形符值。

token.**ATEQUAL**
"`@=`" 的形符值。

token.**RARROW**
"`->`" 的形符值。

token.**ELLIPSIS**
"`...`" 的形符值。

token.**COLONEQUAL**
"`:=`" 的形符值。

token.**OP**

token.**AWAIT**

token.**ASYNC**

token.**TYPE_IGNORE**

token.**TYPE_COMMENT**

token.**ERRORTOKEN**

token.**N_TOKENS**

token.**NT_OFFSET**

C 标记生成器不使用以下标记类型值，但*tokenize* 模块需要这些标记类型值。

token.**COMMENT**
标记值用于表示注释。

token.**NL**
标记值用于表示非终止换行符。*NEWLINE* 标记表示 Python 代码逻辑行的结束；当在多条物理线路上继续执行逻辑代码行时，会生成 `NL` 标记。

token.**ENCODING**
指示用于将源字节解码为文本的编码的标记值。*tokenize.tokenize()* 返回的第一个标记将始终是一个 `ENCODING` 标记。

token.**TYPE_COMMENT**
表示类型注释被识别的形符值。此种形符仅在*ast.parse()* 附带 `type_comments=True` 被发起调用时才会产生。

3.5 版更變: 补充*AWAIT* 和*ASYNC* 标记。

3.7 版更變: 补充*COMMENT* 、*NL* 和*ENCODING* 标记。

3.7 版更變: 移除*AWAIT* 和*ASYNC* 标记。”async” 和”await” 现在被标记为*NAME* 标记。

3.8 版更變: 增加了*TYPE_COMMENT*, *TYPE_IGNORE*, *COLONEQUAL*. Added *AWAIT* 和*ASYNC* 形符（它们对于支持解析对于*ast.parse()* 的 `feature_version` 设为 6 或更低的较老的 Python 版本是必须的）。

32.6 keyword --- 检验 Python 关键字

源码： Lib/keyword.py

此模块允许 Python 程序确定某个字符串是否为 关键字。

`keyword.`**`iskeyword`**(*s*)
如果 *s* 是一个 Python 关键字则返回 `True`。

`keyword.`**`kwlist`**
包含解释器定义的所有 关键字的序列。如果所定义的任何关键字仅在特定*__future__* 语句生效时被激活，它们也将被包含在内。

`keyword.`**`issoftkeyword`**(*s*)
如果 *s* 是一个 Python 的软 关键字则返回 `True`。

3.9 版新加入.

`keyword.`**`softkwlist`**
包含解释器定义的所有软 关键字的序列。如果所定义的任何软关键字仅在特定*__future__* 语句生效时被激活，它们也将被包含在内。

3.9 版新加入.

32.7 tokenize --- 对 Python 代码使用的标记解析器

源码： Lib/tokenize.py

---

tokenize 模块为 Python 源代码提供了一个词法扫描器，用 Python 实现。该模块中的扫描器也将注释作为标记返回，这使得它对于实现“漂亮的输出器”非常有用，包括用于屏幕显示的着色器。

为了简化标记流的处理，所有的 运算符和 定界符以及*Ellipsis* 返回时都会打上通用的*OP* 标记。可以通过*tokenize.tokenize()* 返回的*named tuple* 对象的 `exact_type` 属性来获得确切的标记类型。

32.7.1 对输入进行解析标记

主要的入口是一个生成器*generator*:

`tokenize.`**`tokenize`**(*readline*)
生成器*tokenize()* 需要一个 *readline* 参数，这个参数必须是一个可调用对象，且能提供与文件对象的*io.IOBase.readline()* 方法相同的接口。每次调用这个函数都要返回字节类型输入的一行数据。

生成器产生 5 个具有这些成员的元组：令牌类型；令牌字符串；指定令牌在源中开始的行和列的 2 元组 `(srow, scol)`；指定令牌在源中结束的行和列的 2 元组 `(erow, ecol)`；以及发现令牌的行。所传递的行（最后一个元组项）是 实际的行。5 个元组以*named tuple* 的形式返回，字段名是：`type string start end line`。

返回的*named tuple* 有一个额外的属性，名为 `exact_type`，包含了*OP* 标记的确切操作符类型。对于所有其他标记类型，`exact_type` 等于命名元组的 `type` 字段。

3.1 版更變: 增加了对 named tuple 的支持。

3.3 版更變: 添加了对 `exact_type` 的支持。

根据:pep:*263* ，*tokenize()* 通过寻找 UTF-8 BOM 或编码 cookie 来确定文件的源编码。

tokenize.**generate_tokens**(*readline*)
对读取 unicode 字符串而不是字节的源进行标记。

和*tokenize()* 一样，*readline* 参数是一个返回单行输入的可调用参数。然而，*generate_tokens()* 希望 *readline* 返回一个 str 对象而不是字节。

其结果是一个产生具名元组的的迭代器，与*tokenize()* 完全一样。它不会产生*ENCODING* 标记。

所有来自*token* 模块的常量也可从*tokenize* 导出。

提供了另一个函数来逆转标记化过程。这对于创建对脚本进行标记、修改标记流并写回修改后脚本的工具很有用。

tokenize.**untokenize**(*iterable*)
将令牌转换为 Python 源代码。*iterable* 必须返回至少有两个元素的序列，即令牌类型和令牌字符串。任何额外的序列元素都会被忽略。

重构的脚本以单个字符串的形式返回。结果被保证为标记回与输入相匹配，因此转换是无损的，并保证来回操作。该保证只适用于标记类型和标记字符串，因为标记之间的间距（列位置）可能会改变。

它返回字节，使用*ENCODING* 标记进行编码，这是由*tokenize()* 输出的第一个标记序列。如果输入中没有编码令牌，它将返回一个字符串。

tokenize() 需要检测它所标记源文件的编码。它用来做这件事的函数是可用的:

tokenize.**detect_encoding**(*readline*)
detect_encoding() 函数用于检测解码 Python 源文件时应使用的编码。它需要一个参数，readline ，与*tokenize()* 生成器的使用方式相同。

它最多调用 readline 两次，并返回所使用的编码（作为一个字符串）和它所读入的任何行（不是从字节解码的）的 list 。

它从 UTF-8 BOM 或编码 cookie 的存在中检测编码格式，如 **PEP 263** 所指明的。如果 BOM 和 cookie 都存在，但不一致，将会引发*SyntaxError*。请注意，如果找到 BOM ，将返回 `'utf-8-sig'` 作为编码格式。

如果没有指定编码，那么将返回默认的 `'utf-8'` 编码.

使用*open()* 来打开 Python 源文件：它使用*detect_encoding()* 来检测文件编码。

tokenize.**open**(*filename*)
使用由*detect_encoding()* 检测到的编码，以只读模式打开一个文件。

3.2 版新加入.

exception tokenize.**TokenError**
当文件中任何地方没有完成 docstring 或可能被分割成几行的表达式时触发，例如:

```
"""Beginning of
docstring
```

或是:

```
[1,
 2,
 3
```

注意，未封闭的单引号字符串不会导致错误发生。它们被标记为*ERRORTOKEN* ，然后是其内容的标记化。

32.7.2 命令行用法

3.3 版新加入.

tokenize 模块可以作为一个脚本从命令行执行。这很简单。

```
python -m tokenize [-e] [filename.py]
```

可以接受以下选项：

-h, --help
显示此帮助信息并退出

-e, --exact
使用确切的类型显示令牌名称

如果 `filename.py` 被指定，其内容会被标记到 stdout 。否则，标记化将在 stdin 上执行。

32.7.3 示例

脚本改写器的例子，它将 float 文本转换为 Decimal 对象：。

```
from tokenize import tokenize, untokenize, NUMBER, STRING, NAME, OP
from io import BytesIO

def decistmt(s):
    """Substitute Decimals for floats in a string of statements.

    >>> from decimal import Decimal
    >>> s = 'print(+21.3e-5*-.1234/81.7)'
    >>> decistmt(s)
    "print (+Decimal ('21.3e-5')*-Decimal ('.1234')/Decimal ('81.7'))"

    The format of the exponent is inherited from the platform C library.
    Known cases are "e-007" (Windows) and "e-07" (not Windows).  Since
    we're only showing 12 digits, and the 13th isn't close to 5, the
    rest of the output should be platform-independent.

    >>> exec(s)  #doctest: +ELLIPSIS
    -3.21716034272e-0...7

    Output from calculations with Decimal should be identical across all
    platforms.

    >>> exec(decistmt(s))
    -3.217160342717258261933904529E-7
    """
    result = []
    g = tokenize(BytesIO(s.encode('utf-8')).readline)  # tokenize the string
    for toknum, tokval, _, _, _ in g:
        if toknum == NUMBER and '.' in tokval:  # replace NUMBER tokens
            result.extend([
                (NAME, 'Decimal'),
                (OP, '('),
                (STRING, repr(tokval)),
                (OP, ')')
            ])
```

(下页继续)

(繼續上一頁)

```
        else:
            result.append((toknum, tokval))
    return untokenize(result).decode('utf-8')
```

从命令行进行标记化的例子。脚本:

```
def say_hello():
    print("Hello, World!")

say_hello()
```

将被标记为以下输出，其中第一列是发现标记的行 / 列坐标范围，第二列是标记的名称，最后一列是标记的值（如果有）。

```
$ python -m tokenize hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          OP             '('
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           OP             '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          OP             ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           OP             '('
4,10-4,11:          OP             ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

可以使用`-e`选项来显示确切的标记类型名称。

```
$ python -m tokenize -e hello.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,13:           NAME           'say_hello'
1,13-1,14:          LPAR           '('
1,14-1,15:          RPAR           ')'
1,15-1,16:          COLON          ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,9:            NAME           'print'
2,9-2,10:           LPAR           '('
2,10-2,25:          STRING         '"Hello, World!"'
2,25-2,26:          RPAR           ')'
2,26-2,27:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,9:            NAME           'say_hello'
4,9-4,10:           LPAR           '('
```

(下页继续)

(繼續上一頁)

```
4,10-4,11:          RPAR           ')'
4,11-4,12:          NEWLINE        '\n'
5,0-5,0:            ENDMARKER      ''
```

以编程方式对文件进行标记的例子，用*generate_tokens()* 读取 unicode 字符串而不是字节:

```
import tokenize

with tokenize.open('hello.py') as f:
    tokens = tokenize.generate_tokens(f.readline)
    for token in tokens:
        print(token)
```

或者通过*tokenize()* 直接读取字节数据:

```
import tokenize

with open('hello.py', 'rb') as f:
    tokens = tokenize.tokenize(f.readline)
    for token in tokens:
        print(token)
```

32.8 tabnanny --- 模糊缩进检测

源代码: Lib/tabnanny.py

目前，该模块旨在作为脚本调用。但是可以使用下面描述的*check()* 函数将其导入 IDE。

備🄵: 此模块提供的 API 可能会在将来的版本中更改；此类更改可能无法向后兼容。

---

tabnanny.**check**(*file_or_dir*)
如果 *file_or_dir* 是目录而非符号链接，则递归地在名为 *file_or_dir* 的目录树中下行，沿途检查所有 .py 文件。如果 *file_or_dir* 是一个普通 Python 源文件，将检查其中的空格相关问题。诊断消息将使用*print()* 函数写入到标准输出。

tabnanny.**verbose**
此旗标指明是否打印详细消息。如果作为脚本调用则是通过 -v 选项来增加。

tabnanny.**filename_only**
此旗标指明是否只打印包含空格相关问题文件的文件名。如果作为脚本调用则是通过 -q 选项来设为真值。

exception tabnanny.**NannyNag**
如果检测到模糊缩进则由*process_tokens()* 引发。在*check()* 中捕获并处理。

tabnanny.**process_tokens**(*tokens*)
此函数由*check()* 用来处理由*tokenize* 模块所生成的标记。

也参考:

模块*tokenize* 用于 Python 源代码的词法扫描程序。

32.9 pyclbr --- Python 模块浏览器支持

源代码: Lib/pyclbr.py

---

pyclbr 模块提供了对于以 Python 编写的模块中定义的函数、类和方法的受限信息。这种信息足够用来实现一个模块浏览器。这种信息是从 Python 源代码中直接提取而非通过导入模块，因此该模块可以安全地用于不受信任的代码。此限制使得非 Python 实现的模块无法使用此模块，包括所有标准和可选的扩展模块。

pyclbr.**readmodule**(*module*, *path=None*)
返回一个将模块层级的类名映射到类描述器的字典。如果可能，将会包括已导入基类的描述器。形参 *module* 为要读取模块名称的字符串；它可能是某个包内部的模块名称。*path* 如果给出则为添加到 sys.path 开头的目录路径序列，它会被用于定位模块的源代码。

此函数为原始接口，仅保留用于向下兼容。它会返回以下内容的过滤版本。

pyclbr.**readmodule_ex**(*module*, *path=None*)
返回一个基于字典的树，其中包含与模块中每个用 def 或 class 语句定义的函数和类相对应的函数和类描述器。被返回的字典会将模块层级的函数和类名映射到它们的描述器。嵌套的对象会被输入到它们的上级子目录中。与 readmodule 一样，*module* 指明要读取的模块而 *path* 会被添加到 sys.path。如果被读取的模块是一个包，则返回的字典将具有 '__path__' 键，其值是一个包含包搜索路径的列表。

3.7 版新加入: 嵌套定义的描述器。它们通过新的子属性来访问。每个定义都会有一个新的上级属性。

这些函数所返回的描述器是 Function 和 Class 类的实例。用户不应自行创建这些类的实例。

32.9.1 函式物件

Function 类的实例描述了由 def 语句所定义的函数。它们具有下列属性:

Function.**file**
函数定义所在的文件名称。

Function.**module**
定义了所描述函数的模块名称。

Function.**name**
函数名称。

Function.**lineno**
定义在文件中起始位置的行号。

Function.**parent**
对于最高层级函数为 None。对于嵌套函数则为上级函数。

3.7 版新加入.

Function.**children**
将名称映射到嵌套函数和类描述器的字典。

3.7 版新加入.

32.9.2 Class 对象

Class 类的实例描述了由 class 语句所定义的类。它们具有与 Function 对象相同的属性以及两个额外属性。

Class.**file**
类定义所在的文件名称。

Class.**module**
定义了所描述类的模块名称。

Class.**name**
类名称。

Class.**lineno**
定义在文件中起始位置的行号。

Class.**parent**
对于最高层级类为 None。对于嵌套类则为上级类。

3.7 版新加入.

Class.**children**
将名称映射到嵌套函数和类描述器的字典。

3.7 版新加入.

Class.**super**
一个 Class 对象的列表，它们描述了所描述类的直接基类。被命名为超类但无法被*readmodule_ex()*发现的类会作为类名字符串而非 Class 对象列出。

Class.**methods**
一个将方法名映射到行号的字典。此属性可从更新的子目录中获取，仅保留用于向下兼容。

32.10 py_compile --- 编译 Python 源文件

源代码: Lib/py_compile.py

---

py_compile 模块提供了用来从源文件生成字节码的函数和另一个用于当模块源文件作为脚本被调用时的函数。

虽然不太常用，但这个函数在安装共享模块时还是很有用的，特别是当一些用户可能没有权限在包含源代码的目录中写字节码缓存文件时。

exception py_compile.**PyCompileError**
当编译文件过程中发生错误时，抛出的异常。

py_compile.**compile**(*file*, *cfile=None*, *dfile=None*, *doraise=False*, *optimize=-1*, *invalidation_mode=PycInvalidationMode.TIMESTAMP*, *quiet=0*)
将源文件编译成字节码并写出字节码缓存文件。源代码从名为 *file* 的文件中加载。字节码会写入到 *cfile*，默认为 **PEP 3147/PEP 488** 路径，以 .pyc 结尾。例如，如果 *file* 是 /foo/bar/baz.py 则对于 Python 3.2 *cfile* 将默认为 /foo/bar/__pycache__/baz.cpython-32.pyc。如果指定了 *dfile*，则在错误信息中它将代替 *file* 作为源文件的名称。如果 *doraise* 为真值，则当编译 *file* 遇到错误时将会引发*PyCompileError*。如果 *doraise* 为假值（默认），则将错误信息写入到 sys.stderr，但不会引发异常。此函数返回编译后字节码文件的路径，即 *cfile* 所使用的值。

doraise 和 *quiet* 参数确定在编译文件时如何处理错误。如果 *quiet* 为 0 或 1，并且 *doraise* 为假值，则会启用默认行为：写入错误信息到 sys.stderr，并且函数将返回 None 而非一个路径。如果 *doraise* 为

真值，则将改为引发*PyCompileError*。但是如果 *quiet* 为 2，则不会写入消息，并且 *doraise* 也不会有效果。

如果 *cfile* 所表示（显式指定或计算得出）的路径为符号链接或非常规文件，则将引发*FileExistsError*。此行为是用来警告如果允许写入编译后字节码文件到这些路径则导入操作将会把它们转为常规文件。这是使用文件重命名来将最终编译后字节码文件放置到位以防止并发文件写入问题的导入操作的附带效果。

optimize 控制优化级别并会被传给内置的*compile()* 函数。默认值 `-1` 表示选择当前解释器的优化级别。

invalidation_mode 应当是*PycInvalidationMode* 枚举的成员，它控制在运行时如何让已生成的字节码缓存失效。如果设置了 `SOURCE_DATE_EPOCH` 环境变量则默认值为*PycInvalidationMode.CHECKED_HASH*，否则默认值为*PycInvalidationMode.TIMESTAMP*。

3.2 版更變: 将 *cfile* 的默认值改成与 **PEP 3147** 兼容。之前的默认值是 *file* + `'c'` (如果启用优化则为 `'o'`)。同时也添加了 *optimize* 形参。

3.4 版更變: 将代码更改为使用*importlib* 执行字节码缓存文件写入。这意味着文件创建/写入的语义现在与*importlib* 所做的相匹配，例如权限、写入和移动语义等等。同时也添加了当 *cfile* 为符号链接或非常规文件时引发*FileExistsError* 的预警设置。

3.7 版更變: *invalidation_mode* 形参是根据 **PEP 552** 的描述添加的。如果设置了 `SOURCE_DATE_EPOCH` 环境变量，*invalidation_mode* 将被强制设为*PycInvalidationMode.CHECKED_HASH*。

3.7.2 版更變: `SOURCE_DATE_EPOCH` 环境变量不会再覆盖 *invalidation_mode* 参数的值，而改为确定其默认值。

3.8 版更變: 增加了 *quiet* 形参。

class py_compile.**PycInvalidationMode**

一个由可用方法组成的枚举，解释器可以用来确定字节码文件是否与源文件保持一致。`.pyc` 文件在其标头中指明了所需的失效模式。请参阅 pyc-invalidation 了解有关 Python 在运行时如何让 `.pyc` 文件失效的更多信息。

3.7 版新加入.

TIMESTAMP

`.pyc` 文件包括时间戳和源文件的大小，Python 将在运行时将其与源文件的元数据进行比较以确定 `.pyc` 文件是否需要重新生成。

CHECKED_HASH

`.pyc` 文件包括源文件内容的哈希值，Python 将在运行时将其与源文件内容进行比较以确定 `.pyc` 文件是否需要重新生成。

UNCHECKED_HASH

类似于*CHECKED_HASH*，`.pyc` 文件包括源文件内容的哈希值。但是，Python 将在运行时假定 `.pyc` 文件是最新的而完全不会将 `.pyc` 与源文件进行验证。

此选项适用于 `.pycs` 由 Python 以外的某个系统例如构建系统来确保最新的情况。

py_compile.**main**(*args=None*)

编译多个源文件。在 *args* 中（或者当 *args* 为 `None` 时则是在命令行中）指定的文件会被编译并将结果字节码以正常方式来缓存。此函数不会搜索目录结构来定位源文件；它只编译显式指定的文件。如果 `'-'` 是 args 中唯一的值，则会从标准输入获取文件列表。

3.2 版更變: 添加了对 `'-'` 的支持。

当此模块作为脚本运行时，会使用*main()* 来编译命令行中指定的所有文件。如果某个文件无法被编译则退出状态将为非零值。

也参考:

模块*compileall* 编译一个目录树中所有 Python 源文件的工具。

32.11 compileall --- Byte-compile Python libraries

Source code: Lib/compileall.py

This module provides some utility functions to support installing Python libraries. These functions compile Python source files in a directory tree. This module can be used to create the cached byte-code files at library installation time, which makes them available for use even by users who don't have write permission to the library directories.

32.11.1 Command-line use

This module can work as a script (using **python -m compileall**) to compile Python sources.

directory ...
file ...
Positional arguments are files to compile or directories that contain source files, traversed recursively. If no argument is given, behave as if the command line was `-l <directories from sys.path>`.

-l
Do not recurse into subdirectories, only compile source code files directly contained in the named or implied directories.

-f
Force rebuild even if timestamps are up-to-date.

-q
Do not print the list of files compiled. If passed once, error messages will still be printed. If passed twice (`-qq`), all output is suppressed.

-d `destdir`
Directory prepended to the path to each file being compiled. This will appear in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

-s `strip_prefix`

-p `prepend_prefix`
Remove (`-s`) or append (`-p`) the given prefix of paths recorded in the `.pyc` files. Cannot be combined with `-d`.

-x `regex`
regex is used to search the full path to each file considered for compilation, and if the regex produces a match, the file is skipped.

-i `list`
Read the file `list` and add each line that it contains to the list of files and directories to compile. If `list` is `-`, read lines from `stdin`.

-b
Write the byte-code files to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

-r
Control the maximum recursion level for subdirectories. If this is given, then `-l` option will not be taken into account. **python -m compileall <directory> -r 0** is equivalent to **python -m compileall <directory> -l**.

-j `N`
Use *N* workers to compile the files within the given directory. If `0` is used, then the result of *os.cpu_count()* will be used.

--invalidation-mode `[timestamp|checked-hash|unchecked-hash]`
Control how the generated byte-code files are invalidated at runtime. The `timestamp` value, means that `.pyc` files with the source timestamp and size embedded will be generated. The `checked-hash` and `unchecked-hash` values cause hash-based pycs to be generated. Hash-based pycs embed a hash of the source file contents rather than a timestamp. See pyc-invalidation for more information on how Python validates bytecode cache files at runtime. The default is `timestamp` if the `SOURCE_DATE_EPOCH` environment variable is not set, and `checked-hash` if the `SOURCE_DATE_EPOCH` environment variable is set.

-o `level`
Compile with the given optimization level. May be used multiple times to compile for multiple levels at a time (for example, `compileall -o 1 -o 2`).

-e `dir`
Ignore symlinks pointing outside the given directory.

--hardlink-dupes
If two `.pyc` files with different optimization level have the same content, use hard links to consolidate duplicate files.

3.2 版更變: Added the `-i`, `-b` and `-h` options.

3.5 版更變: Added the `-j`, `-r`, and `-qq` options. `-q` option was changed to a multilevel value. `-b` will always produce a byte-code file ending in `.pyc`, never `.pyo`.

3.7 版更變: Added the `--invalidation-mode` option.

3.9 版更變: Added the `-s`, `-p`, `-e` and `--hardlink-dupes` options. Raised the default recursion limit from 10 to *sys.getrecursionlimit()*. Added the possibility to specify the `-o` option multiple times.

There is no command-line option to control the optimization level used by the *compile()* function, because the Python interpreter itself already provides the option: **python -O -m compileall**.

Similarly, the *compile()* function respects the *sys.pycache_prefix* setting. The generated bytecode cache will only be useful if *compile()* is run with the same *sys.pycache_prefix* (if any) that will be used at runtime.

32.11.2 Public functions

`compileall.`**compile_dir**(*dir*, *maxlevels=sys.getrecursionlimit()*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *workers=1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)
Recursively descend the directory tree named by *dir*, compiling all `.py` files along the way. Return a true value if all the files compiled successfully, and a false value otherwise.

The *maxlevels* parameter is used to limit the depth of the recursion; it defaults to `sys.getrecursionlimit()`.

If *ddir* is given, it is prepended to the path to each file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *force* is true, modules are re-compiled even if the timestamps are up to date.

If *rx* is given, its `search` method is called on the complete path to each file considered for compilation, and if it returns a true value, the file is skipped. This can be used to exclude files matching a regular expression, given as a *re.Pattern* object.

If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in *compile()* function. Accepts also a sequence of optimization levels which lead to multiple compilations of one `.py` file in one call.

The argument *workers* specifies how many workers are used to compile files in parallel. The default is to not use multiple workers. If the platform can't use multiple workers and *workers* argument is given, then sequential compilation will be used as a fallback. If *workers* is 0, the number of cores in the system is used. If *workers* is lower than `0`, a *ValueError* will be raised.

invalidation_mode should be a member of the *py_compile.PycInvalidationMode* enum and controls how the generated pycs are invalidated at runtime.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str`, `bytes` or *os.PathLike*.

If *hardlink_dupes* is true and two `.pyc` files with different optimization level have the same content, use hard links to consolidate duplicate files.

3.2 版更變: Added the *legacy* and *optimize* parameter.

3.5 版更變: Added the *workers* parameter.

3.5 版更變: *quiet* parameter was changed to a multilevel value.

3.5 版更變: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

3.6 版更變: 接受一个类路径对象。

3.7 版更變: The *invalidation_mode* parameter was added.

3.7.2 版更變: The *invalidation_mode* parameter's default value is updated to None.

3.8 版更變: Setting *workers* to 0 now chooses the optimal number of cores.

3.9 版更變: Added *stripdir*, *prependdir*, *limit_sl_dest* and *hardlink_dupes* arguments. Default value of *maxlevels* was changed from `10` to `sys.getrecursionlimit()`

compileall.**compile_file**(*fullname*, *ddir=None*, *force=False*, *rx=None*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*, ***, *stripdir=None*, *prependdir=None*, *limit_sl_dest=None*, *hardlink_dupes=False*)

Compile the file with path *fullname*. Return a true value if the file compiled successfully, and a false value otherwise.

If *ddir* is given, it is prepended to the path to the file being compiled for use in compilation time tracebacks, and is also compiled in to the byte-code file, where it will be used in tracebacks and other messages in cases where the source file does not exist at the time the byte-code file is executed.

If *rx* is given, its `search` method is passed the full path name to the file being compiled, and if it returns a true value, the file is not compiled and `True` is returned. This can be used to exclude files matching a regular expression, given as a *re.Pattern* object.

If *quiet* is `False` or `0` (the default), the filenames and other information are printed to standard out. Set to `1`, only errors are printed. Set to `2`, all output is suppressed.

If *legacy* is true, byte-code files are written to their legacy locations and names, which may overwrite byte-code files created by another version of Python. The default is to write files to their **PEP 3147** locations and names, which allows byte-code files from multiple versions of Python to coexist.

optimize specifies the optimization level for the compiler. It is passed to the built-in *compile()* function. Accepts also a sequence of optimization levels which lead to multiple compilations of one `.py` file in one call.

invalidation_mode should be a member of the *py_compile.PycInvalidationMode* enum and controls how the generated pycs are invalidated at runtime.

The *stripdir*, *prependdir* and *limit_sl_dest* arguments correspond to the `-s`, `-p` and `-e` options described above. They may be specified as `str`, `bytes` or *os.PathLike*.

If *hardlink_dupes* is true and two `.pyc` files with different optimization level have the same content, use hard links to consolidate duplicate files.

3.2 版新加入.

3.5 版更變: *quiet* parameter was changed to a multilevel value.

3.5 版更變: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

3.7 版更變: The *invalidation_mode* parameter was added.

3.7.2 版更變: The *invalidation_mode* parameter's default value is updated to None.

3.9 版更變: Added *stripdir*, *prependdir*, *limit_sl_dest* and *hardlink_dupes* arguments.

`compileall.`**`compile_path`**(*skip_curdir=True*, *maxlevels=0*, *force=False*, *quiet=0*, *legacy=False*, *optimize=-1*, *invalidation_mode=None*)

Byte-compile all the `.py` files found along `sys.path`. Return a true value if all the files compiled successfully, and a false value otherwise.

If *skip_curdir* is true (the default), the current directory is not included in the search. All other parameters are passed to the *compile_dir()* function. Note that unlike the other compile functions, `maxlevels` defaults to `0`.

3.2 版更變: Added the *legacy* and *optimize* parameter.

3.5 版更變: *quiet* parameter was changed to a multilevel value.

3.5 版更變: The *legacy* parameter only writes out `.pyc` files, not `.pyo` files no matter what the value of *optimize* is.

3.7 版更變: The *invalidation_mode* parameter was added.

3.7.2 版更變: The *invalidation_mode* parameter's default value is updated to None.

To force a recompile of all the `.py` files in the `Lib/` subdirectory and all its subdirectories:

```
import compileall

compileall.compile_dir('Lib/', force=True)

# Perform same compilation, excluding files in .svn directories.
import re
compileall.compile_dir('Lib/', rx=re.compile(r'[/\\][.]svn'), force=True)

# pathlib.Path objects can also be used.
import pathlib
compileall.compile_dir(pathlib.Path('Lib/'), force=True)
```

也參考:

模块*py_compile* Byte-compile a single source file.

32.12 dis --- Python bytecode 的反組譯器

原始碼： Lib/dis.py

---

dis 模組支援反組譯分析 CPython *bytecode*。CPython bytecode 作F輸入的模組被定義於 Include/opcode.h F且被编譯器和直譯器所使用。

CPython implementation detail: 字节码是 CPython 解释器的实现细节。不保证不会在 Python 版本之间添加、删除或更改字节码。不应考虑将此模块的跨 Python VM 或 Python 版本的使用。

3.6 版更變: 每条指令使用 2 个字节。以前字节数因指令而异。

示例：给出函数 myfunc():

```
def myfunc(alist):
    return len(alist)
```

可以使用以下命令显示 myfunc() 的反汇编

```
>>> dis.dis(myfunc)
  2           0 LOAD_GLOBAL              0 (len)
              2 LOAD_FAST                0 (alist)
              4 CALL_FUNCTION            1
              6 RETURN_VALUE
```

("2" 是行号)。

32.12.1 字节码分析

3.4 版新加入.

字节码分析 API 允许将 Python 代码片段包装在*Bytecode* 对象中，以便轻松访问已编译代码的详细信息。

class dis.**Bytecode**(*x*, *, *first_line=None*, *current_offset=None*)

分析的字节码对应于函数、生成器、异步生成器、协程、方法、源代码字符串或代码对象（由*compile()* 返回）。

这是下面列出的许多函数的便利包装，最值得注意的是*get_instructions()*，迭代于*Bytecode* 的实例产生字节码操作*Instruction* 的实例。

如果 *first_line* 不是 None，则表示应该为反汇编代码中的第一个源代码行报告的行号。否则，源行信息（如果有的话）直接来自反汇编的代码对象。

如果 *current_offset* 不是 None，则它指的是反汇编代码中的指令偏移量。设置它意味着*dis()* 将针对指定的操作码显示“当前指令”标记。

classmethod from_traceback(*tb*)

从给定回溯构造一个*Bytecode* 实例，将设置 *current_offset* 为异常负责的指令。

codeobj

已编译的代码对象。

first_line

代码对象的第一个源代码行（如果可用）

dis()
返回字节码操作的格式化视图（与*dis.dis()* 打印相同，但作为多行字符串返回）。

info()
返回带有关于代码对象的详细信息的格式化多行字符串，如*code_info()* 。

3.7 版更變: 现在可以处理协程和异步生成器对象。

示例:

```
>>> bytecode = dis.Bytecode(myfunc)
>>> for instr in bytecode:
...     print(instr.opname)
...
LOAD_GLOBAL
LOAD_FAST
CALL_FUNCTION
RETURN_VALUE
```

32.12.2 分析函数

dis 模块还定义了以下分析函数，它们将输入直接转换为所需的输出。如果只执行单个操作，它们可能很有用，因此中间分析对象没用：

dis.**code_info**(*x*)
返回格式化的多行字符串，其包含详细代码对象信息的用于被提供的函数、生成器、异步生成器、协程、方法、源代码字符串或代码对象。

请注意，代码信息字符串的确切内容是高度依赖于实现的，它们可能会在 Python VM 或 Python 版本中任意更改。

3.2 版新加入.

3.7 版更變: 现在可以处理协程和异步生成器对象。

dis.**show_code**(*x*, *, *file=None*)
将提供的函数、方法。源代码字符串或代码对象的详细代码对象信息打印到 *file* （如果未指定 *file* ，则为 `sys.stdout` ）。

这是 `print(code_info(x), file=file)` 的便捷简写，用于在解释器提示符下进行交互式探索。

3.2 版新加入.

3.4 版更變: 添加 *file* 形参。

dis.**dis**(*x=None*, *, *file=None*, *depth=None*)
反汇编 *x* 对象。*x* 可以表示模块、类、方法、函数、生成器、异步生成器、协程、代码对象、源代码字符串或原始字节码的字节序列。对于模块，它会反汇编所有功能。对于一个类，它反汇编所有方法（包括类和静态方法）。对于代码对象或原始字节码序列，它每字节码指令打印一行。它还递归地反汇编嵌套代码对象（推导式代码，生成器表达式和嵌套函数，以及用于构建嵌套类的代码）。在被反汇编之前，首先使用*compile()* 内置函数将字符串编译为代码对象。如果未提供任何对象，则此函数会反汇编最后一次回溯。

如果提供的话，反汇编将作为文本写入提供的 *file* 参数，否则写入 `sys.stdout` 。

递归的最大深度受 *depth* 限制，除非它是 `None` 。`depth=0` 表示没有递归。

3.4 版更變: 添加 *file* 形参。

3.7 版更變: 实现了递归反汇编并添加了 *depth* 参数。

3.7 版更變: 现在可以处理协程和异步生成器对象。

dis.**distb**(*tb=None*, *, *file=None*)
如果没有传递，则使用最后一个回溯来反汇编回溯的堆栈顶部函数。指示了导致异常的指令。

如果提供的话，反汇编将作为文本写入提供的 *file* 参数，否则写入 sys.stdout 。

3.4 版更變: 添加 *file* 形参。

dis.**disassemble**(*code*, *lasti=-1*, *, *file=None*)
dis.**disco**(*code*, *lasti=-1*, *, *file=None*)
反汇编代码对象，如果提供了 *lasti* ，则指示最后一条指令。输出分为以下几列:

1. 行号，用于每行的第一条指令
2. 当前指令，表示为 `-->` ，
3. 一个标记的指令，用 >> 表示，
4. 指令的地址，
5. 操作码名称，
6. 操作参数，和
7. 括号中参数的解释。

参数解释识别本地和全局变量名称、常量值、分支目标和比较运算符。

如果提供的话，反汇编将作为文本写入提供的 *file* 参数，否则写入 sys.stdout 。

3.4 版更變: 添加 *file* 形参。

dis.**get_instructions**(*x*, *, *first_line=None*)
在所提供的函数、方法、源代码字符串或代码对象中的指令上返回一个迭代器。

迭代器生成一系列 *Instruction* ，命名为元组，提供所提供代码中每个操作的详细信息。

如果 *first_line* 不是 None ，则表示应该为反汇编代码中的第一个源代码行报告的行号。否则，源行信息（如果有的话）直接来自反汇编的代码对象。

3.4 版新加入.

dis.**findlinestarts**(*code*)
此生成器函数使用代码对象 *code* 的 co_firstlineno 和 co_lnotab 属性来查找源代码中行开头的偏移量。它们生成为 (offset, lineno) 对。请参阅 objects/lnotab_notes.txt ，了解 co_lnotab 格式以及如何解码它。

3.6 版更變: 行号可能会减少。以前，他们总是在增加。

dis.**findlabels**(*code*)
检测作为跳转目标的原始编译后字节码字符串 *code* 中的所有偏移量，并返回这些偏移量的列表。

dis.**stack_effect**(*opcode*, *oparg=None*, *, *jump=None*)
使用参数 *oparg* 计算 *opcode* 的堆栈效果。

如果代码有一个跳转目标并且 *jump* 是 True ，则 drag_effect() 将返回跳转的堆栈效果。如果 *jump* 是 False ，它将返回不跳跃的堆栈效果。如果 *jump* 是 None （默认值），它将返回两种情况的最大堆栈效果。

3.4 版新加入.

3.8 版更變: 添加 *jump* 参数。

32.12.3 Python 字节码说明

get_instructions() 函数和*Bytecode* 类提供字节码指令的详细信息的*Instruction* 实例：

class dis.**Instruction**
字节码操作的详细信息

opcode
操作的数字代码，对应于下面列出的操作码值和操作码集合 中的字节码值。

opname
人类可读的操作名称

arg
操作的数字参数（如果有的话），否则为 None

argval
已解析的 arg 值（如果已知），否则与 arg 相同

argrepr
人类可读的操作参数描述

offset
在字节码序列中启动操作索引

starts_line
行由此操作码（如果有）启动，否则为 None

is_jump_target
如果其他代码跳到这里，则为 True ，否则为 False

3.4 版新加入.

Python 编译器当前生成以下字节码指令。

一般指令

NOP
什么都不做。用作字节码优化器的占位符。

POP_TOP
删除堆栈顶部（TOS）项。

ROT_TWO
交换两个最顶层的堆栈项。

ROT_THREE
将第二个和第三个堆栈项向上提升一个位置，顶项移动到位置三。

ROT_FOUR
将第二个、第三个和第四个堆栈项向上提升一个位置，将顶项移动到第四个位置。

3.8 版新加入.

DUP_TOP
复制堆栈顶部的引用。

3.2 版新加入.

DUP_TOP_TWO
复制堆栈顶部的两个引用，使它们保持相同的顺序。

3.2 版新加入.

一元操作

一元操作获取堆栈顶部元素，应用操作，并将结果推回堆栈。

UNARY_POSITIVE
实现 `TOS = +TOS` 。

UNARY_NEGATIVE
实现 `TOS = -TOS` 。

UNARY_NOT
实现 `TOS = not TOS` 。

UNARY_INVERT
实现 `TOS = ~TOS` 。

GET_ITER
实现 `TOS = iter(TOS)` 。

GET_YIELD_FROM_ITER
如果 `TOS` 是一个*generator iterator* 或*coroutine* 对象则保持原样。否则实现 `TOS = iter(TOS)` 。

3.5 版新加入.

二元操作

二元操作从堆栈中删除堆栈顶部（TOS）和第二个最顶层堆栈项（TOS1）。它们执行操作，并将结果放回堆栈。

BINARY_POWER
实现 `TOS = TOS1 ** TOS` 。

BINARY_MULTIPLY
实现 `TOS = TOS1 * TOS` 。

BINARY_MATRIX_MULTIPLY
实现 `TOS = TOS1 @ TOS` 。

3.5 版新加入.

BINARY_FLOOR_DIVIDE
实现 `TOS = TOS1 // TOS`。

BINARY_TRUE_DIVIDE
实现 `TOS = TOS1 / TOS` 。

BINARY_MODULO
实现 `TOS = TOS1 % TOS` 。

BINARY_ADD
实现 `TOS = TOS1 + TOS` 。

BINARY_SUBTRACT
实现 `TOS = TOS1 - TOS` 。

BINARY_SUBSCR
实现 `TOS = TOS1[TOS]` 。

BINARY_LSHIFT
实现 `TOS = TOS1 << TOS` 。

BINARY_RSHIFT
实现 `TOS = TOS1 >> TOS` 。

BINARY_AND
实现 `TOS = TOS1 & TOS`。

BINARY_XOR
实现 `TOS = TOS1 ^ TOS`。

BINARY_OR
实现 `TOS = TOS1 | TOS`。

就地操作

就地操作就像二元操作，因为它们删除了 TOS 和 TOS1，并将结果推回到堆栈上，但是当 TOS1 支持它时，操作就地完成，并且产生的 TOS 可能是（但不一定）原来的 TOS1。

INPLACE_POWER
就地实现 `TOS = TOS1 ** TOS`。

INPLACE_MULTIPLY
就地实现 `TOS = TOS1 * TOS`。

INPLACE_MATRIX_MULTIPLY
就地实现 `TOS = TOS1 @ TOS`。

3.5 版新加入.

INPLACE_FLOOR_DIVIDE
就地实现 `TOS = TOS1 // TOS`。

INPLACE_TRUE_DIVIDE
就地实现 `TOS = TOS1 / TOS`。

INPLACE_MODULO
就地实现 `TOS = TOS1 % TOS`。

INPLACE_ADD
就地实现 `TOS = TOS1 + TOS`。

INPLACE_SUBTRACT
就地实现 `TOS = TOS1 - TOS`。

INPLACE_LSHIFT
就地实现 `TOS = TOS1 << TOS`。

INPLACE_RSHIFT
就地实现 `TOS = TOS1 >> TOS`。

INPLACE_AND
就地实现 `TOS = TOS1 & TOS`。

INPLACE_XOR
就地实现 `TOS = TOS1 ^ TOS`。

INPLACE_OR
就地实现 `TOS = TOS1 | TOS`。

STORE_SUBSCR
实现 `TOS1[TOS] = TOS2`。

DELETE_SUBSCR
实现 `del TOS1[TOS]`。

协程操作码

GET_AWAITABLE
实现 `TOS = get_awaitable(TOS)`，其中 `get_awaitable(o)` 返回 `o` 如果 `o` 是一个有 CO_ITERABLE_COROUTINE 标志的协程对象或生成器对象，否则解析 `o.__await__`。

3.5 版新加入.

GET_AITER
实现 `TOS = TOS.__aiter__()`。

3.5 版新加入.

3.7 版更變: 已经不再支持从 `__aiter__` 返回可等待对象。

GET_ANEXT
实现 `PUSH(get_awaitable(TOS.__anext__()))`。参见 GET_AWAITABLE 获取更多 `get_awaitable` 的细节

3.5 版新加入.

END_ASYNC_FOR
终止一个 `async for` 循环。处理等待下一个项目时引发的异常。如果 TOS 是*StopAsyncIteration*，从堆栈弹出 7 个值，并使用后三个恢复异常状态。否则，使用堆栈中的三个值重新引发异常。从块堆栈中删除异常处理程序块。

3.8 版新加入.

BEFORE_ASYNC_WITH
从栈顶对象解析 `__aenter__` 和 `__aexit__`。将 `__aexit__` 和 `__aenter__()` 的结果推入堆栈。

3.5 版新加入.

SETUP_ASYNC_WITH
创建一个新的帧对象。

3.5 版新加入.

其他操作码

PRINT_EXPR
实现交互模式的表达式语句。TOS 从堆栈中被移除并打印。在非交互模式下，表达式语句以*POP_TOP*终止。

SET_ADD(*i*)
调用 `set.add(TOS1[-i], TOS)`。用于实现集合推导。

LIST_APPEND(*i*)
调用 `list.append(TOS1[-i], TOS)`。用于实现列表推导式。

MAP_ADD(*i*)
调用 `dict.__setitem__(TOS1[-i], TOS1, TOS)`。用于实现字典推导。

3.1 版新加入.

3.8 版更變: 映射值为 TOS，映射键为 TOS1。之前，它们被颠倒了。

对于所有*SET_ADD*、*LIST_APPEND*和*MAP_ADD*指令，当弹出添加的值或键值对时，容器对象保留在堆栈上，以便它可用于循环的进一步迭代。

RETURN_VALUE
返回 TOS 到函数的调用者。

YIELD_VALUE
弹出 TOS 并从一个*generator*生成它。

YIELD_FROM
弹出 TOS 并将其委托给它作为*generator* 的子迭代器。

3.3 版新加入.

SETUP_ANNOTATIONS
检查 `__annotations__` 是否在 `locals()` 中定义，如果没有，它被设置为空 `dict` 。只有在类或模块体静态地包含*variable annotations* 时才会发出此操作码。

3.6 版新加入.

IMPORT_STAR
将所有不以 `'_'` 开头的符号直接从模块 TOS 加载到局部命名空间。加载所有名称后弹出该模块。这个操作码实现了 `from module import *` 。

POP_BLOCK
从块堆栈中删除一个块。有一块堆栈，每帧用于表示 `try` 语句等。

POP_EXCEPT
从块堆栈中删除一个块。弹出的块必须是异常处理程序块，在进入 except 处理程序时隐式创建。除了从帧堆栈弹出无关值之外，最后三个弹出值还用于恢复异常状态。

RERAISE
重新引发当前位于栈顶的异常。

3.9 版新加入.

WITH_EXCEPT_START
调用堆栈中 7 号位置上的函数并附带栈顶位置的三项作为参数。用来在 `with` 语句内发生异常时实现调用 `context_manager.__exit__(*exc_info())`。

3.9 版新加入.

LOAD_ASSERTION_ERROR
将*AssertionError* 推入栈顶。由 `assert` 语句使用。

3.9 版新加入.

LOAD_BUILD_CLASS
将 `builtins .__ build_class__()` 推到堆栈上。它之后被*CALL_FUNCTION* 调用来构造一个类。

SETUP_WITH(*delta*)
此操作码会在 with 代码块开始之前执行多个操作。首先，它从上下文管理器加载 `__exit__()` 并将其推入栈顶以供*WITH_EXCEPT_START* 后续使用。然后，调用 `__enter__()`，并推入一个指向 *delta* 的 finally 代码块。最后，将调用 `__enter__()` 方法的结果推入栈顶。下一个操作码将忽略它 (*POP_TOP*)，或将其存储在一个或多个变量 (*STORE_FAST*, *STORE_NAME* 或*UNPACK_SEQUENCE*) 中。

3.2 版新加入.

以下所有操作码均使用其参数。

STORE_NAME(*namei*)
实现 `name = TOS`。*namei* 是 *name* 在代码对象的 `co_names` 属性中的索引。在可能的情况下，编译器会尝试使用*STORE_FAST* 或*STORE_GLOBAL*。

DELETE_NAME(*namei*)
实现 `del name`，其中 *namei* 是代码对象的 `co_names` 属性的索引。

UNPACK_SEQUENCE(*count*)
将 TOS 解包为 *count* 个单独的值，它们将按从右至左的顺序被放入堆栈。

UNPACK_EX(*counts*)
实现使用带星号的目标进行赋值：将 TOS 中的可迭代对象解包为单独的值，其中值的总数可以小于可迭代对象中的项数：新值之一将是由所有剩余项构成的列表。

counts 的低字节是列表值之前的值的数量，*counts* 中的高字节则是之后的值的数量。结果值会按从右至左的顺序入栈。

STORE_ATTR(*namei*)
实现 `TOS.name = TOS1`，其中 *namei* 是 name 在 `co_names` 中的索引号。

DELETE_ATTR(*namei*)
实现 `del TOS.name`，使用 *namei* 作为 `co_names` 中的索引号。

STORE_GLOBAL(*namei*)
类似于*STORE_NAME* 但会将 name 存储为全局变量。

DELETE_GLOBAL(*namei*)
类似于*DELETE_NAME* 但会删除一个全局变量。

LOAD_CONST(*consti*)
将 `co_consts[consti]` 推入栈顶。

LOAD_NAME(*namei*)
将与 `co_names[namei]` 相关联的值推入栈顶。

BUILD_TUPLE(*count*)
创建一个使用了来自栈的 *count* 个项的元组，并将结果元组推入栈顶。

BUILD_LIST(*count*)
类似于*BUILD_TUPLE* 但会创建一个列表。

BUILD_SET(*count*)
类似于*BUILD_TUPLE* 但会创建一个集合。

BUILD_MAP(*count*)
将一个新字典对象推入栈顶。弹出 `2 * count` 项使得字典包含 *count* 个条目: `{..., TOS3: TOS2, TOS1: TOS}`。

3.5 版更變: 字典是根据栈中的项创建而不是创建一个预设大小包含 *count* 项的空字典。

BUILD_CONST_KEY_MAP(*count*)
BUILD_MAP 版本专用于常量键。弹出的栈顶元素包含一个由键构成的元组，然后从 `TOS1` 开始从构建字典的值中弹出 *count* 个值。

3.6 版新加入.

BUILD_STRING(*count*)
拼接 *count* 个来自栈的字符串并将结果字符串推入栈顶。

3.6 版新加入.

LIST_TO_TUPLE
从堆栈中弹出一个列表并推入一个包含相同值的元组。

3.9 版新加入.

LIST_EXTEND(*i*)
调用 `list.extend(TOS1[-i], TOS)`。用于构建列表。

3.9 版新加入.

SET_UPDATE(*i*)
调用 `set.update(TOS1[-i], TOS)`。用于构建集合。

3.9 版新加入.

DICT_UPDATE(*i*)
调用 `dict.update(TOS1[-i], TOS)`。用于构建字典。

3.9 版新加入.

DICT_MERGE
类似于*DICT_UPDATE* 但对于重复的键会引发异常。

3.9 版新加入.

LOAD_ATTR(*namei*)
将 TOS 替换为 `getattr(TOS, co_names[namei])`。

COMPARE_OP(*opname*)
执行布尔运算操作。操作名称可在 `cmp_op[opname]` 中找到。

IS_OP(*invert*)
执行 `is` 比较，或者如果 `invert` 为 1 则执行 `is not`。

3.9 版新加入.

CONTAINS_OP(*invert*)
执行 `in` 比较，或者如果 `invert` 为 1 则执行 `not in`。

3.9 版新加入.

IMPORT_NAME(*namei*)
导入模块 `co_names[namei]`。会弹出 TOS 和 TOS1 以提供 *fromlist* 和 *level* 参数给*__import__()*。模块对象会被推入栈顶。当前命名空间不受影响：对于一条标准 import 语句，会执行后续的*STORE_FAST* 指令来修改命名空间。

IMPORT_FROM(*namei*)
从在 TOS 内找到的模块中加载属性 `co_names[namei]`。结果对象会被推入栈顶，以便由后续的*STORE_FAST* 指令来保存。

JUMP_FORWARD(*delta*)
将字节码计数器的值增加 *delta*。

POP_JUMP_IF_TRUE(*target*)
如果 TOS 为真值，则将字节码计数器的值设为 *target*。TOS 会被弹出。

3.1 版新加入.

POP_JUMP_IF_FALSE(*target*)
如果 TOS 为假值，则将字节码计数器的值设为 *target*。TOS 会被弹出。

3.1 版新加入.

JUMP_IF_NOT_EXC_MATCH(*target*)
检测堆栈中的第二个值是否为匹配 TOS 的异常，如果不是则会跳转。从堆栈中弹出两个值。

3.9 版新加入.

JUMP_IF_TRUE_OR_POP(*target*)
如果 TOS 为真值，则将字节码计数器的值设为 *target* 并将 TOS 留在栈顶。否则（如 TOS 为假值），TOS 会被弹出。

3.1 版新加入.

JUMP_IF_FALSE_OR_POP(*target*)
如果 TOS 为假值，则将字节码计数器的值设为 *target* 并将 TOS 留在栈顶。否则（如 TOS 为真值），TOS 会被弹出。

3.1 版新加入.

JUMP_ABSOLUTE(*target*)
将字节码计数器的值设为 *target*。

FOR_ITER (*delta*)
TOS 是一个*iterator*。请调用其`__next__()`方法。如果此操作产生了一个新值，则将其推入栈顶（将迭代器留在其下方）。如果迭代器提示已耗尽，TOS 会被弹出，并且字节码计数器将增加 *delta*。

LOAD_GLOBAL (*namei*)
加载名称为 `co_names[namei]` 的全局对象推入栈顶。

SETUP_FINALLY (*delta*)
将一个来自 try-finally 或 try-except 子句的 try 代码块推入代码块栈顶。相对 finally 代码块或第一个 except 代码块 *delta* 个点数。

LOAD_FAST (*var_num*)
将指向局部对象 `co_varnames[var_num]` 的引用推入栈顶。

STORE_FAST (*var_num*)
将 TOS 存放到局部对象 `co_varnames[var_num]`。

DELETE_FAST (*var_num*)
移除局部对象 `co_varnames[var_num]`。

LOAD_CLOSURE (*i*)
将一个包含在单元的第 *i* 个空位中的对单元的引用推入栈顶并释放可用的存储空间。如果 *i* 小于 *co_cellvars* 的长度则变量的名称为 `co_cellvars[i]`。否则为 `co_freevars[i - len(co_cellvars)]`。

LOAD_DEREF (*i*)
加载包含在单元的第 *i* 个空位中的单元并释放可用的存储空间。将一个对单元所包含对象的引用推入栈顶。

LOAD_CLASSDEREF (*i*)
类似于*LOAD_DEREF* 但在查询单元之前会首先检查局部对象字典。这被用于加载类语句体中的自由变量。

3.4 版新加入.

STORE_DEREF (*i*)
将 TOS 存放到包含在单元的第 *i* 个空位中的单元内并释放可用存储空间。

DELETE_DEREF (*i*)
清空包含在单元的第 *i* 个空位中的单元并释放可用存储空间。被用于 `del` 语句。

3.2 版新加入.

RAISE_VARARGS (*argc*)
使用 `raise` 语句的 3 种形式之一引发异常，具体形式取决于 *argc* 的值:

- 0: `raise` (重新引发之前的异常)
- 1: `raise TOS` (在 `TOS` 上引发异常实例或类型)
- 2: `raise TOS1 from TOS` (在 `TOS1` 上引发异常实例或类型并将 `__cause__` 设为 `TOS`)

CALL_FUNCTION (*argc*)
调用一个可调用对象并传入位置参数。*argc* 指明位置参数的数量。栈顶包含位置参数，其中最右边的参数在最顶端。在参数之下是一个待调用的可调用对象。`CALL_FUNCTION` 会从栈中弹出所有参数以及可调用对象，附带这些参数调用该可调用对象，并将可调用对象所返回的返回值推入栈顶。

3.6 版更變: 此操作码仅用于附带位置参数的调用。

CALL_FUNCTION_KW (*argc*)
调用一个可调用对象并传入位置参数（如果有的话）和关键字参数。*argc* 指明位置参数和关键字参数的总数量。栈顶元素包含一个关键字参数名称的元组，名称必须为字符串。在元组之下是与元组顺序相对应的关键字参数值。在它之下则是位置参数，其中最右边的参数在最顶端。在参数之下是要调用

的可调用对象。CALL_FUNCTION_KW 会从栈中弹出所有参数及可调用对象，附带这些参数调用该可调用对象，并将可调用对象所返回的返回值推入栈顶。

3.6 版更變: 关键字参数会被打包为一个元组而非字典，*argc* 指明参数的总数量。

CALL_FUNCTION_EX (*flags*)
调用一个可调用对象并附带位置参数和关键字参数变量集合。如果设置了 *flags* 的最低位，则栈顶包含一个由额外关键字参数组成的映射对象。在调用该可调用对象之前，映射对象和可迭代对象会被分别“解包”并将它们的内容分别作为关键字参数和位置参数传入。CALL_FUNCTION_EX 会中栈中弹出所有参数及可调用对象，附带这些参数调用该可调用对象，并将可调用对象所返回的返回值推入栈顶。

3.6 版新加入.

LOAD_METHOD (*namei*)
从 TOS 对象加载一个名为 co_names[namei] 的方法。TOS 将被弹出。此字节码可区分两种情况：如果 TOS 有一个名称正确的方法，字节码会将未绑定方法和 TOS 推入栈顶。TOS 将在调用未绑定方法时被用作*CALL_METHOD* 的第一个参数 (self)。否则会将 NULL 和属性查找所返回的对象推入栈顶。

3.7 版新加入.

CALL_METHOD (*argc*)
调用一个方法。*argc* 是位置参数的数量。关键字参数不受支持。此操作码被设计用于配合*LOAD_METHOD* 使用。位置参数放在栈顶。在它们之下放在栈中的是由*LOAD_METHOD* 所描述的两个条目（或者是 self 和一个未绑定方法对象，或者是 NULL 和一个任意可调用对象）。它们会被全部弹出并将返回值推入栈顶。

3.7 版新加入.

MAKE_FUNCTION (*flags*)
将一个新函数对象推入栈顶。从底端到顶端，如果参数带有指定的旗标值则所使用的栈必须由这些值组成。

- 0x01 一个默认值的元组，用于按位置排序的仅限位置形参以及位置或关键字形参
- 0x02 一个仅限关键字形参的默认值的字典
- 0x04 是一个标注字典
- 0x08 一个包含用于自由变量的单元的元组，生成一个闭包
- 与函数相关联的代码 (在 TOS1)
- 函数的*qualified name* (在 TOS)

BUILD_SLICE (*argc*)
将一个切片对象推入栈顶。*argc* 必须为 2 或 3。如果为 2，则推入 slice(TOS1, TOS)；如果为 3，则推入 slice(TOS2, TOS1, TOS)。请参阅*slice()* 内置函数了解详细信息。

EXTENDED_ARG (*ext*)
为任意带有大到无法放入默认的单字节的参数的操作码添加前缀。*ext* 存放一个附加字节作为参数中的高比特位。对于每个操作码，最多允许三个 EXTENDED_ARG 前缀，构成两字节到三字节的参数。

FORMAT_VALUE (*flags*)
用于实现格式化字面值字符串（f-字符串）。从栈中弹出一个可选的 *fmt_spec*，然后是一个必须的 *value*。*flags* 的解读方式如下：

- (flags & 0x03) == 0x00: *value* 按原样格式化。
- (flags & 0x03) == 0x01: 在格式化 *value* 之前调用其*str()*。
- (flags & 0x03) == 0x02: 在格式化 *value* 之前调用其*repr()*。
- (flags & 0x03) == 0x03: 在格式化 *value* 之前调用其*ascii()*。
- (flags & 0x04) == 0x04: 从栈中弹出 *fmt_spec* 并使用它，否则使用空的 *fmt_spec*。

使用 PyObject_Format() 执行格式化。结果会被推入栈顶。

3.6 版新加入.

HAVE_ARGUMENT
这不是一个真正的操作码。它用于标明使用参数和不使用参数的操作码 (分别为 < HAVE_ARGUMENT 和 >= HAVE_ARGUMENT) 之间的分隔线。

3.6 版更變: 现在每条指令都带有参数，但操作码 < HAVE_ARGUMENT 会忽略它。之前仅限操作码 >= HAVE_ARGUMENT 带有参数。

32.12.4 操作码集合

提供这些集合用于字节码指令的自动内省:

dis.**opname**
操作名称的序列，可使用字节码来索引。

dis.**opmap**
映射操作名称到字节码的字典

dis.**cmp_op**
所有比较操作名称的序列。

dis.**hasconst**
访问常量的字节码序列。

dis.**hasfree**
访问自由变量的字节码序列（请注意这里所说的‘自由’是指在当前作用域中被内部作用域所引用的名称，或在外部作用域中被此作用域所引用的名称。它 并不包括对全局或内置作用域的引用）。

dis.**hasname**
按名称访问属性的字节码序列。

dis.**hasjrel**
具有相对跳转目标的字节码序列。

dis.**hasjabs**
具有绝对跳转目标的字节码序列。

dis.**haslocal**
访问局部变量的字节码序列。

dis.**hascompare**
布尔运算的字节码序列。

32.13 pickletools --- pickle 开发者工具集

源代码: Lib/pickletools.py

此模块包含与*pickle* 模块内部细节有关的多个常量，一些关于具体实现的详细注释，以及一些能够分析封存数据的有用函数。此模块的内容对需要操作*pickle* 的 Python 核心开发者来说很有用处；*pickle* 的一般用户则可能会感觉*pickletools* 模块与他们无关。

32.13.1 命令行语法

3.2 版新加入.

当从命令行发起调用时，`python -m pickletools` 将对一个或更多 pickle 文件的内容进行拆解。请注意如果你查看 pickle 中保存的 Python 对象而非 pickle 格式的细节，你可能需要改用 `-m pickle`。但是，当你想检查的 pickle 文件来自某个不受信任的源时，`-m pickletools` 是更安全的选择，因为它不会执行 pickle 字节码。

例如，对于一个封存在文件 `x.pickle` 中的元组 `(1, 2)`:

```
$ python -m pickle x.pickle
(1, 2)

$ python -m pickletools x.pickle
    0: \x80 PROTO      3
    2: K    BININT1    1
    4: K    BININT1    2
    6: \x86 TUPLE2
    7: q    BINPUT     0
    9: .    STOP
highest protocol among opcodes = 2
```

命令行选项

-a, --annotate
使用简短的操作码描述来标注每一行。

-o, --output=<file>
输出应当写入到的文件名称。

-l, --indentlevel=<num>
一个新的 MARK 层级所需缩进的空格数。

-m, --memo
当反汇编多个对象时，保留各个反汇编的备忘记录。

-p, --preamble=<preamble>
当指定一个以上的 pickle 文件时，在每次反汇编之前打印给定的前言。

32.13.2 编程接口

pickletools.**dis**(*pickle*, *out=None*, *memo=None*, *indentlevel=4*, *annotate=0*)
将 pickle 的符号化反汇编数据输出到文件类对象 *out*，默认为 `sys.stdout`。*pickle* 可以是一个字符串或一个文件类对象。*memo* 可以是一个将被用作 pickle 的备忘记录的 Python 字典；它可被用来对由同一封存器创建的多个封存对象执行反汇编。由 `MARK` 操作码指明的每个连续级别将会缩进 *indentlevel* 个空格。如果为 *annotate* 指定了一个非零值，则输出中的每个操作码将以一个简短描述来标注。*annotate* 的值会被用作标注所应开始的列的提示。

3.2 版新加入: *annotate* 参数。

pickletools.**genops**(*pickle*)
提供包含 pickle 中所有操作码的*iterator*，返回一个 `(opcode, arg, pos)` 三元组的序列。*opcode* 是 `OpcodeInfo` 类的一个实例；*arg* 是 Python 对象形式的 opcode 参数的已解码值；*pos* 是 opcode 所在的位置。*pickle* 可以是一个字符串或一个文件类对象。

pickletools.**optimize**(*picklestring*)

在消除未使用的 PUT 操作码之后返回一个新的等效 pickle 字符串。优化后的 pickle 将更为简短，耗费更为的传输时间，要求更少的存储空间并能更高效地解封。

CHAPTER 33

杂项服务

本章中介绍的模块提供了所有 Python 版本中提供的各种杂项服务。这是一个概述：

33.1 `formatter` --- 通用格式化输出

3.4 版後已🄴用: 因为被使用的次数很少，此格式化模块已经被弃用了。

This module supports two interface definitions, each with multiple implementations: The *formatter* interface, and the *writer* interface which is required by the formatter interface.

Formatter objects transform an abstract flow of formatting events into specific output events on writer objects. Formatters manage several stack structures to allow various properties of a writer object to be changed and restored; writers need not be able to handle relative changes nor any sort of "change back" operation. Specific writer properties which may be controlled via formatter objects are horizontal alignment, font, and left margin indentations. A mechanism is provided which supports providing arbitrary, non-exclusive style settings to a writer as well. Additional interfaces facilitate formatting events which are not reversible, such as paragraph separation.

Writer objects encapsulate device interfaces. Abstract devices, such as file formats, are supported as well as physical devices. The provided implementations all work with abstract devices. The interface makes available mechanisms for setting the properties which formatter objects manage and inserting data into the output.

33.1.1 The Formatter Interface

Interfaces to create formatters are dependent on the specific formatter class being instantiated. The interfaces described below are the required interfaces which all formatters must support once initialized.

One data element is defined at the module level:

`formatter.`**`AS_IS`**
: Value which can be used in the font specification passed to the `push_font()` method described below, or as the new value to any other `push_property()` method. Pushing the `AS_IS` value allows the corresponding `pop_property()` method to be called without having to track whether the property was changed.

The following attributes are defined for formatter instance objects:

formatter.**writer**
: The writer instance with which the formatter interacts.

formatter.**end_paragraph**(*blanklines*)
: Close any open paragraphs and insert at least *blanklines* before the next paragraph.

formatter.**add_line_break**()
: Add a hard line break if one does not already exist. This does not break the logical paragraph.

formatter.**add_hor_rule**(**args*, ***kw*)
: Insert a horizontal rule in the output. A hard break is inserted if there is data in the current paragraph, but the logical paragraph is not broken. The arguments and keywords are passed on to the writer's `send_line_break()` method.

formatter.**add_flowing_data**(*data*)
: Provide data which should be formatted with collapsed whitespace. Whitespace from preceding and successive calls to *add_flowing_data()* is considered as well when the whitespace collapse is performed. The data which is passed to this method is expected to be word-wrapped by the output device. Note that any word-wrapping still must be performed by the writer object due to the need to rely on device and font information.

formatter.**add_literal_data**(*data*)
: Provide data which should be passed to the writer unchanged. Whitespace, including newline and tab characters, are considered legal in the value of *data*.

formatter.**add_label_data**(*format*, *counter*)
: Insert a label which should be placed to the left of the current left margin. This should be used for constructing bulleted or numbered lists. If the *format* value is a string, it is interpreted as a format specification for *counter*, which should be an integer. The result of this formatting becomes the value of the label; if *format* is not a string it is used as the label value directly. The label value is passed as the only argument to the writer's `send_label_data()` method. Interpretation of non-string label values is dependent on the associated writer.

  Format specifications are strings which, in combination with a counter value, are used to compute label values. Each character in the format string is copied to the label value, with some characters recognized to indicate a transform on the counter value. Specifically, the character `'1'` represents the counter value formatter as an Arabic number, the characters `'A'` and `'a'` represent alphabetic representations of the counter value in upper and lower case, respectively, and `'I'` and `'i'` represent the counter value in Roman numerals, in upper and lower case. Note that the alphabetic and roman transforms require that the counter value be greater than zero.

formatter.**flush_softspace**()
: Send any pending whitespace buffered from a previous call to *add_flowing_data()* to the associated writer object. This should be called before any direct manipulation of the writer object.

formatter.**push_alignment**(*align*)
: Push a new alignment setting onto the alignment stack. This may be *AS_IS* if no change is desired. If the alignment value is changed from the previous setting, the writer's `new_alignment()` method is called with the *align* value.

formatter.**pop_alignment**()
: Restore the previous alignment.

formatter.**push_font**(*(size*, *italic*, *bold*, *teletype)*)
: Change some or all font properties of the writer object. Properties which are not set to *AS_IS* are set to the values passed in while others are maintained at their current settings. The writer's `new_font()` method is called with the fully resolved font specification.

formatter.**pop_font**()
: Restore the previous font.

formatter.**push_margin**(*margin*)
: Increase the number of left margin indentations by one, associating the logical tag *margin* with the new indentation.

The initial margin level is `0`. Changed values of the logical tag must be true values; false values other than *AS_IS* are not sufficient to change the margin.

`formatter.`**`pop_margin`**`()`
Restore the previous margin.

`formatter.`**`push_style`**`(`**styles*`)`
Push any number of arbitrary style specifications. All styles are pushed onto the styles stack in order. A tuple representing the entire stack, including *AS_IS* values, is passed to the writer's `new_styles()` method.

`formatter.`**`pop_style`**`(`*n=1*`)`
Pop the last *n* style specifications passed to *push_style()*. A tuple representing the revised stack, including *AS_IS* values, is passed to the writer's `new_styles()` method.

`formatter.`**`set_spacing`**`(`*spacing*`)`
Set the spacing style for the writer.

`formatter.`**`assert_line_data`**`(`*flag=1*`)`
Inform the formatter that data has been added to the current paragraph out-of-band. This should be used when the writer has been manipulated directly. The optional *flag* argument can be set to false if the writer manipulations produced a hard line break at the end of the output.

33.1.2 Formatter Implementations

Two implementations of formatter objects are provided by this module. Most applications may use one of these classes without modification or subclassing.

`class` `formatter.`**`NullFormatter`**`(`*writer=None*`)`
A formatter which does nothing. If *writer* is omitted, a *NullWriter* instance is created. No methods of the writer are called by *NullFormatter* instances. Implementations should inherit from this class if implementing a writer interface but don't need to inherit any implementation.

`class` `formatter.`**`AbstractFormatter`**`(`*writer*`)`
The standard formatter. This implementation has demonstrated wide applicability to many writers, and may be used directly in most circumstances. It has been used to implement a full-featured World Wide Web browser.

33.1.3 The Writer Interface

Interfaces to create writers are dependent on the specific writer class being instantiated. The interfaces described below are the required interfaces which all writers must support once initialized. Note that while most applications can use the *AbstractFormatter* class as a formatter, the writer must typically be provided by the application.

`writer.`**`flush`**`()`
Flush any buffered output or device control events.

`writer.`**`new_alignment`**`(`*align*`)`
Set the alignment style. The *align* value can be any object, but by convention is a string or `None`, where `None` indicates that the writer's "preferred" alignment should be used. Conventional *align* values are `'left'`, `'center'`, `'right'`, and `'justify'`.

`writer.`**`new_font`**`(`*font*`)`
Set the font style. The value of *font* will be `None`, indicating that the device's default font should be used, or a tuple of the form `(size, italic, bold, teletype)`. Size will be a string indicating the size of font that should be used; specific strings and their interpretation must be defined by the application. The *italic*, *bold*, and *teletype* values are Boolean values specifying which of those font attributes should be used.

writer.**new_margin**(*margin*, *level*)
Set the margin level to the integer *level* and the logical tag to *margin*. Interpretation of the logical tag is at the writer's discretion; the only restriction on the value of the logical tag is that it not be a false value for non-zero values of *level*.

writer.**new_spacing**(*spacing*)
Set the spacing style to *spacing*.

writer.**new_styles**(*styles*)
Set additional styles. The *styles* value is a tuple of arbitrary values; the value `AS_IS` should be ignored. The *styles* tuple may be interpreted either as a set or as a stack depending on the requirements of the application and writer implementation.

writer.**send_line_break**()
Break the current line.

writer.**send_paragraph**(*blankline*)
Produce a paragraph separation of at least *blankline* blank lines, or the equivalent. The *blankline* value will be an integer. Note that the implementation will receive a call to `send_line_break()` before this call if a line break is needed; this method should not include ending the last line of the paragraph. It is only responsible for vertical spacing between paragraphs.

writer.**send_hor_rule**(**args*, ***kw*)
Display a horizontal rule on the output device. The arguments to this method are entirely application- and writer-specific, and should be interpreted with care. The method implementation may assume that a line break has already been issued via `send_line_break()`.

writer.**send_flowing_data**(*data*)
Output character data which may be word-wrapped and re-flowed as needed. Within any sequence of calls to this method, the writer may assume that spans of multiple whitespace characters have been collapsed to single space characters.

writer.**send_literal_data**(*data*)
Output character data which has already been formatted for display. Generally, this should be interpreted to mean that line breaks indicated by newline characters should be preserved and no new line breaks should be introduced. The data may contain embedded newline and tab characters, unlike data provided to the `send_formatted_data()` interface.

writer.**send_label_data**(*data*)
Set *data* to the left of the current left margin, if possible. The value of *data* is not restricted; treatment of non-string values is entirely application- and writer-dependent. This method will only be called at the beginning of a line.

33.1.4 Writer Implementations

Three implementations of the writer object interface are provided as examples by this module. Most applications will need to derive new writer classes from the `NullWriter` class.

class formatter.**NullWriter**
A writer which only provides the interface definition; no actions are taken on any methods. This should be the base class for all writers which do not need to inherit any implementation methods.

class formatter.**AbstractWriter**
A writer which can be used in debugging formatters, but not much else. Each method simply announces itself by printing its name and arguments on standard output.

class formatter.**DumbWriter**(*file=None*, *maxcol=72*)
Simple writer class which writes output on the *file object* passed in as *file* or, if *file* is omitted, on standard output.

The output is simply word-wrapped to the number of columns specified by *maxcol*. This class is suitable for reflowing a sequence of paragraphs.

CHAPTER 34

Windows 系统相关模块

本章节叙述的模块只在 Windows 平台上可用。

34.1 msvcrt --- 来自 MS VC++ 运行时的有用例程

这些函数提供了对 Windows 平台上一些有用功能的访问。一些更高级别的模块使用这些函数来构建其服务的 Windows 实现。例如，*getpass* 模块在实现*getpass()* 函数时使用了这些函数。

关于这些函数的更多信息可以在平台 API 文档中找到。

该模块实现了控制台 I/O API 的普通和宽字符变体。普通的 API 只处理 ASCII 字符，国际化应用受限。应该尽可能地使用宽字符 API 。

3.3 版更變: 此模块中过去会引发*IOError* 的操作现在将引发*OSError*。

34.1.1 文件操作

msvcrt.**locking**(*fd*, *mode*, *nbytes*)
基于文件描述符 *fd* 从 C 运行时锁定文件的某一部分。失败时引发*OSError*。锁定的文件区域从当前文件位置开始扩展 *nbytes* 个字节，并可能持续到超出文件末尾。*mode* 必须为下面列出的 LK_* 之一。一个文件中的多个区域可以被同时锁定，但是不能重叠。相邻区域不会被合并；它们必须单独被解锁。

引发一个审计事件 msvcrt.locking，附带参数 fd, mode, nbytes。

msvcrt.**LK_LOCK**
msvcrt.**LK_RLCK**
锁定指定的字节数据。如果字节数据无法被锁定，程序会在 1 秒之后立即重试。如果在 10 次尝试后字节数据仍无法被锁定，则会引发*OSError*。

msvcrt.**LK_NBLCK**
msvcrt.**LK_NBRLCK**
锁定指定的字节数据。如果字节数据无法被锁定，则会引发*OSError*。

msvcrt.**LK_UNLCK**
解锁指定的字节数据，该对象必须在之前被锁定。

msvcrt.**setmode**(*fd*, *flags*)
设置文件描述符 *fd* 的行结束符转写模式。要将其设为文本模式，则 *flags* 应当为*os.O_TEXT*；设为二进制模式，则应当为*os.O_BINARY*。

msvcrt.**open_osfhandle**(*handle*, *flags*)
基于文件句柄 *handle* 创建一个 C 运行时文件描述符。*flags* 形参应当*os.O_APPEND*, *os.O_RDONLY* 和*os.O_TEXT* 按位 OR 的结果。返回的文件描述符可以被用作*os.fdopen()* 的形参以创建一个文件对象。

引发一个审计事件 msvcrt.open_osfhandle，附带参数 handle, flags。

msvcrt.**get_osfhandle**(*fd*)
返回文件描述符 *fd* 的文件句柄。如果 *fd* 不能被识别则会引发*OSError*。

引发一个审计事件 msvcrt.get_osfhandle，附带参数 fd。

34.1.2 控制台 I/O

msvcrt.**kbhit**()
如果有某个按键正在等待被读取则返回 True。

msvcrt.**getch**()
读取一个按键并将结果字符返回为一个字节串。不会有内容回显到缝制台。如果还没有任何键被按下此调用将会阻塞，但它将不会等待 Enter 被按下。如果按下的键是一个特殊功能键，此函数将返回 '\000' 或 '\xe0'；下一次调用将返回键代码。Control-C 按钮无法使用此函数来读取。

msvcrt.**getwch**()
getch() 的宽字符版本，返回一个 Unicode 值。

msvcrt.**getche**()
类似于*getch()*，但按键如果表示一个可打印字符则它将被回显。

msvcrt.**getwche**()
getche() 的宽字符版本，返回一个 Unicode 值。

msvcrt.**putch**(*char*)
将字符串 *char* 打印到终端，不使用缓冲区。

msvcrt.**putwch**(*unicode_char*)
putch() 的宽字符版本，接受一个 Unicode 值。

msvcrt.**ungetch**(*char*)
使得字节串 *char* 被“推回”终端缓冲区；它将是被*getch()* 或*getche()* 读取的下一个字符。

msvcrt.**ungetwch**(*unicode_char*)
ungetch() 的宽字符版本，接受一个 Unicode 值。

34.1.3 其他函数

msvcrt.**heapmin**()
强制 malloc() 堆清空自身并将未使用的块返回给操作系统。失败时，这将引发*OSError*。

34.2 winreg --- Windows 注册表访问

这些函数将 Windows 注册表 API 暴露给 Python。为了确保即便程序员忘记显式关闭时也能够正确关闭，这里没有用整数作为注册表句柄，而是采用了句柄对象。

3.3 版更變: 该模块中的几个函数被用于引发*WindowsError*，该异常现在是*OSError* 的别名。

34.2.1 函数

该模块提供了下列函数：

winreg.**CloseKey**(*hkey*)
关闭之前打开的注册表键。参数 *hkey* 指之前打开的键。

> **備🄵:** 如果没有使用该方法关闭 *hkey* (或者通过*hkey.Close()*)，在对象 *hkey* 被 Python 销毁时会将其关闭。

winreg.**ConnectRegistry**(*computer_name*, *key*)
建立到另一台计算上上的预定义注册表句柄的连接，并返回一个*handle 对象*.

computer_name 是远程计算机的名称，以 r"\\computername" 的形式。如果是 None，将会使用本地计算机。

key 是所连接到的预定义句柄。

返回值是所开打键的句柄。如果函数失败，则引发一个*OSError* 异常。

引发一个审计事件 winreg.ConnectRegistry，附带参数 computer_name, key。

3.3 版更變: 参考上文。

winreg.**CreateKey**(*key*, *sub_key*)
创建或打开特定的键，返回一个*handle 对象*。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 是用于命名该方法所打开或创建的键的字符串。

如果 *key* 是预定义键之一，*sub_key* 可能会是 None。该情况下，返回的句柄就是传入函数的句柄。

如果键已经存在，则该函数打开已经存在的该键。

返回值是所开打键的句柄。如果函数失败，则引发一个*OSError* 异常。

引发一个审计事件 winreg.CreateKey，附带参数 key, sub_key, access。

引发一个审计事件 winreg.OpenKey/result，附带参数 key。

3.3 版更變: 参考上文。

winreg.**CreateKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_WRITE*)
创建或打开特定的键，返回一个*handle 对象*。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 是用于命名该方法所打开或创建的键的字符串。

reserved 是一个保留的证书，必须是零。默认值为零。

access 为一个整数，用于给键的预期安全访问指定访问掩码。默认值为*KEY_WRITE*。参阅*Access Rights* 了解其它允许值。

如果 *key* 是预定义键之一，*sub_key* 可能会是 None。该情况下，返回的句柄就是传入函数的句柄。

如果键已经存在，则该函数打开已经存在的该键。

返回值是所开打键的句柄。如果函数失败，则引发一个*OSError* 异常。

引发一个审计事件 winreg.CreateKey，附带参数 key, sub_key, access。

引发一个审计事件 winreg.OpenKey/result，附带参数 key。

3.2 版新加入.

3.3 版更變: 参考上文。

winreg.**DeleteKey**(*key*, *sub_key*)
删除指定的键。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 这个字符串必须是由 *key* 参数所指定键的一个子项。该值项不可以是 None，同时键也不可以有子项。

该方法不能删除带有子项的键。

如果方法成功，则整个键，包括其所有值项都会被移除。如果方法失败，则引发一个*OSError* 异常。

引发一个审计事件 winreg.DeleteKey，附带参数 key, sub_key, access。

3.3 版更變: 参考上文。

winreg.**DeleteKeyEx**(*key*, *sub_key*, *access=KEY_WOW64_64KEY*, *reserved=0*)
删除指定的键。

備🄴: 函数*DeleteKeyEx()* 通过 RegDeleteKeyEx 这个 Windows API 函数实现，该函数为 Windows 的 64 位版本专属。参阅 RegDeleteKeyEx 文档。

---

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 这个字符串必须是由 *key* 参数所指定键的一个子项。该值项不可以是 None，同时键也不可以有子项。

reserved 是一个保留的证书，必须是零。默认值为零。

access 为一个整数，用于给键的预期安全访问指定访问掩码。默认值为常量 _WOW64_64KEY 。参阅*Access Rights* 了解其它允许值。

该方法不能删除带有子项的键。

如果方法成功，则整个键，包括其所有值项都会被移除。如果方法失败，则引发一个*OSError* 异常。

在不支持的 Windows 版本之上，将会引发*NotImplementedError* 异常。

引发一个审计事件 winreg.DeleteKey，附带参数 key, sub_key, access。

3.2 版新加入.

3.3 版更變: 参考上文。

winreg.**DeleteValue**(*key*, *value*)
从某个注册键中删除一个命名值项。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

value 为标识所要删除值项的字符串。

引发一个审计事件 `winreg.DeleteValue`，附带参数 `key`, `value`。

winreg.**EnumKey**(*key*, *index*)
列举某个已经打开注册表键的子项，并返回一个字符串。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

index 为一个整数，用于标识所获取键的索引。

每次调用该函数都会获取一个子项的名字。通常它会被反复调用，直到引发*OSError* 异常，这说明已经没有更多的可用值了。

引发一个审计事件 `winreg.EnumKey`，附带参数 `key`, `index`。

3.3 版更變: 参考上文。

winreg.**EnumValue**(*key*, *index*)
列举某个已经打开注册表键的值项，并返回一个元组。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

index 为一个整数，用于标识要获取值项的索引。

每次调用该函数都会获取一个子项的名字。通常它会被反复调用，直到引发*OSError* 异常，这说明已经没有更多的可用值了。

结果为 3 元素的元组。

索引	意义
0	用于标识值项名称的字符串。
1	保存值项数据的对象，其类型取决于背后的注册表类型。
2	标识值项数据类型的整数。(请查阅*SetValueEx()* 文档中的表格)

引发一个审计事件 `winreg.EnumValue`，附带参数 `key`, `index`。

3.3 版更變: 参考上文。

winreg.**ExpandEnvironmentStrings**(*str*)
Expands environment variable placeholders `%NAME%` in strings like *REG_EXPAND_SZ*:

```
>>> ExpandEnvironmentStrings('%windir%')
'C:\\Windows'
```

引发一个审计事件 `winreg.ExpandEnvironmentStrings`，附带参数 `str`。

winreg.**FlushKey**(*key*)
将某个键的所有属性写入注册表。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

没有必要调用*FlushKey()* 去改动注册表键。注册表的变动是由其延迟刷新机制更新到磁盘的。在系统关机时，也会将注册表的变动写入磁盘。与*CloseKey()* 不同，*FlushKey()* 方法只有等到所

有数据都写入注册表后才会返回。只有需要绝对确认注册表变动已写入磁盘时，应用程序才应去调用*FlushKey()*。

備𝔽: 如果不知道是否要调用*FlushKey()*，可能就是不需要。

winreg.**LoadKey**(*key*, *sub_key*, *file_name*)
在指定键之下创建一个子键，并将指定文件中的注册表信息存入该子键中。

key 是由*ConnectRegistry()* 返回的句柄，或者是常量*HKEY_USERS* 或*HKEY_LOCAL_MACHINE* 之一。

sub_key 是个字符串，用于标识需要载入的子键。

file_name 是要加载注册表数据的文件名。该文件必须是用*SaveKey()* 函数创建的。在文件分配表（FAT）文件系统中，文件名可能不带扩展名。

如果调用*LoadKey()* 的进程没有 SE_RESTORE_PRIVILEGE 权限，则调用会失败。请注意，特权与权限不同——更多细节请参阅 RegLoadKey 文档。

如果 *key* 是由*ConnectRegistry()* 返回的句柄，那么 *file_name* 指定的路径是相对于远程计算机而言的。

引发一个审计事件 winreg.LoadKey，附带参数 key, sub_key, file_name。

winreg.**OpenKey**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)
winreg.**OpenKeyEx**(*key*, *sub_key*, *reserved=0*, *access=KEY_READ*)
打开指定的注册表键，返回*handle 对象*。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

sub_key 是个字符串，标识了需要打开的子键。

reserved 是个保留整数，必须为零。默认值为零。

access 是个指定访问掩码的整数，掩码描述了注册表键所需的安全权限。默认值为*KEY_READ*。其他合法值参见访问权限。

返回结果为一个新句柄，指向指定的注册表键。

如果调用失败，则会触发*OSError* 。

引发一个审计事件 winreg.OpenKey，附带参数 key, sub_key, access。

引发一个审计事件 winreg.OpenKey/result，附带参数 key。

3.2 版更變: 允许使用命名参数。

3.3 版更變: 参考上文。

winreg.**QueryInfoKey**(*key*)
以元组形式返回某注册表键的信息。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

结果为 3 元素的元组。

索引	意义
0	表示此注册表键有多少个子键的整数值。
1	整数值，给出了此注册表键的值的数量。
2	整数值，给出了此注册表键的最后修改时间，单位为自 1601 年 1 月 1 日以来的 100 纳秒。

引发一个审计事件 winreg.QueryInfoKey，附带参数 key。

winreg.**QueryValue**(*key*, *sub_key*)
读取某键的未命名值，形式为字符串。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

sub_key 是个字符串，用于保存与某个值相关的子键名称。如果本参数为 None 或空，函数将读取由*SetValue()* 方法为 *key* 键设置的值。

注册表中的值包含名称、类型和数据。本方法将读取注册表键值的第一个名称为 NULL 的数据。可是底层的 API 调用不会返回类型，所以如果可能的话，请一定使用*QueryValueEx()*。

引发一个审计事件 winreg.QueryValue，附带参数 key, sub_key, value_name。

winreg.**QueryValueEx**(*key*, *value_name*)
读取已打开注册表键指定值名称的类型和数据。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

value_name 是字符串，表示要查询的值。

结果为二元组：

索引	意义
0	注册表项的值。
1	整数值，给出该值的注册表类型（请查看文档中的表格了解*SetValueEx()*）。

引发一个审计事件 winreg.QueryValue，附带参数 key, sub_key, value_name。

winreg.**SaveKey**(*key*, *file_name*)
将指定注册表键及其所有子键存入指定的文件。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

file_name 是要保存注册表数据的文件名。该文件不能已存在。如果文件名包括扩展名，也不能在文件分配表（FAT）文件系统中用于*LoadKey()* 方法。

如果 *key* 代表远程计算机的注册表键，那么 *file_name* 所描述的路径是相对于远程计算机的。本方法的调用者必须拥有 SeBackupPrivilege 特权。请注意，特权与权限是不同的——更多细节请参见 用户权利和权限之间的冲突文档。

本函数将 NULL 传给 API 的 *security_attributes*。

引发一个审计事件 winreg.SaveKey，附带参数 key, file_name。

winreg.**SetValue**(*key*, *sub_key*, *type*, *value*)
将值与指定的注册表键关联。

key 为某个已经打开的键，或者预定义的*HKEY_** 常量 之一。

sub_key 是个字符串，用于命名与该值相关的子键。

type 是个整数，用于指定数据的类型。目前这必须是*REG_SZ*，意味着只支持字符串。请用*SetValueEx()* 函数支持其他的数据类型。

value 是设置新值的字符串。

如果 *sub_key* 参数指定的注册表键不存在，SetValue 函数会创建一个。

值的长度受到可用内存的限制。较长的值（超过 2048 字节）应存为文件，并将文件名存入配置注册表。这有助于提高注册表的使用效率。

由 *key* 参数标识的注册表键，必须已用*KEY_SET_VALUE* 方式打开。

引发一个审计事件 winreg.SetValue，附带参数 key, sub_key, type, value。

winreg.**SetValueEx**(*key*, *value_name*, *reserved*, *type*, *value*)
将数据存入已打开的注册表键的值中。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

value_name 是个字符串，用于命名与值相关的子键。

reserved 可以是任意数据——传给 API 的总是 0。

type 是个整数，用于指定数据的类型。请参阅*Value Types* 了解可用的类型。

value 是设置新值的字符串。

本方法也可为指定的注册表键设置额外的值和类型信息。注册表键必须已用*KEY_SET_VALUE* 方式打开。

请用*CreateKey()* 或*OpenKey()* 方法打开注册表键。

值的长度受到可用内存的限制。较长的值（超过 2048 字节）应存为文件，并将文件名存入配置注册表。这有助于提高注册表的使用效率。

引发一个审计事件 `winreg.SetValue`，附带参数 `key`, `sub_key`, `type`, `value`。

winreg.**DisableReflectionKey**(*key*)
禁用运行于 64 位操作系统的 32 位进程的注册表重定向。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

如果在 32 位操作系统上执行，一般会触发*NotImplementedError*。

如果注册表键不在重定向列表中，函数会调用成功，但没有实际效果。禁用注册表键的重定向不会影响任何子键的重定向。

引发一个审计事件 `winreg.DisableReflectionKey`，附带参数 `key`。

winreg.**EnableReflectionKey**(*key*)
恢复已禁用注册表键的重定向。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

如果在 32 位操作系统上执行，一般会触发*NotImplementedError*。

恢复注册表键的重定向不会影响任何子键的重定向。

引发一个审计事件 `winreg.EnableReflectionKey`，附带参数 `key`。

winreg.**QueryReflectionKey**(*key*)
确定给定注册表键的重定向状况。

key 为某个已经打开的键，或者预定义的*HKEY_* 常量* 之一。

如果重定向已禁用则返回 `True`。

如果在 32 位操作系统上执行，一般会触发*NotImplementedError*。

引发一个审计事件 `winreg.QueryReflectionKey`，附带参数 `key`。

34.2.2 常数

以下常量定义成可供很多 _winreg 函数使用。

HKEY_* 常量

winreg.**HKEY_CLASSES_ROOT**
本注册表键下的注册表项定义了文件的类型（或类别）及相关属性。Shell 和 COM 应用程序将使用该注册表键下保存的信息。

winreg.**HKEY_CURRENT_USER**
属于该注册表键的表项定义了当前用户的偏好。这些偏好值包括环境变量设置、程序组数据、颜色、打印机、网络连接和应用程序参数。

winreg.**HKEY_LOCAL_MACHINE**
属于该注册表键的表项定义了计算机的物理状态，包括总线类型、系统内存和已安装软硬件等数据。

winreg.**HKEY_USERS**
属于该注册表键的表项定义了当前计算机中新用户的默认配置和当前用户配置。

winreg.**HKEY_PERFORMANCE_DATA**
属于该注册表键的表项可用于读取性能数据。这些数据其实并不存放于注册表中；注册表提供功能让系统收集数据。

winreg.**HKEY_CURRENT_CONFIG**
包含有关本地计算机系统当前硬件配置的信息。

winreg.**HKEY_DYN_DATA**
Windows 98 以上版本不使用该注册表键。

访问权限

更多信息，请参阅 注册表密钥安全和访问。

winreg.**KEY_ALL_ACCESS**
组合了 STANDARD_RIGHTS_REQUIRED 、*KEY_QUERY_VALUE* 、*KEY_SET_VALUE* 、*KEY_CREATE_SUB_KEY* 、*KEY_ENUMERATE_SUB_KEYS* 、*KEY_NOTIFY* 和*KEY_CREATE_LINK* 访问权限。

winreg.**KEY_WRITE**
组合了 STANDARD_RIGHTS_WRITE 、*KEY_SET_VALUE* 和*KEY_CREATE_SUB_KEY* 访问权限。

winreg.**KEY_READ**
组合了 STANDARD_RIGHTS_READ 、*KEY_QUERY_VALUE* 、*KEY_ENUMERATE_SUB_KEYS* 和*KEY_NOTIFY* 。

winreg.**KEY_EXECUTE**
等价于*KEY_READ*。

winreg.**KEY_QUERY_VALUE**
查询注册表键值时需要用到。

winreg.**KEY_SET_VALUE**
创建、删除或设置注册表值时需要用到。

winreg.**KEY_CREATE_SUB_KEY**
创建注册表键的子键时需要用到。

winreg.**KEY_ENUMERATE_SUB_KEYS**
枚举注册表键的子键时需要用到。

winreg.**KEY_NOTIFY**
为注册表键或子键请求修改通知时需要用到。

winreg.**KEY_CREATE_LINK**
保留给系统使用。

64 位系统特有

详情请参阅 Accessing an Alternate Registry View。

winreg.**KEY_WOW64_64KEY**
表示 64 位 Windows 中的应用程序应在 64 位注册表视图上操作。

winreg.**KEY_WOW64_32KEY**
表示 64 位 Windows 中的应用程序应在 32 位注册表视图上操作。

注册表值的类型

详情请参阅 Registry Value Types。

winreg.**REG_BINARY**
任意格式的二进制数据。

winreg.**REG_DWORD**
32 位数值。

winreg.**REG_DWORD_LITTLE_ENDIAN**
32 位低字节序格式的数字。相当于*REG_DWORD*。

winreg.**REG_DWORD_BIG_ENDIAN**
32 位高字节序格式的数字。

winreg.**REG_EXPAND_SZ**
包含环境变量（`%PATH%`）的字符串，以空字符结尾。

winreg.**REG_LINK**
Unicode 符号链接。

winreg.**REG_MULTI_SZ**
一串以空字符结尾的字符串，最后以两个空字符结尾。Python 会自动处理这种结尾形式。

winreg.**REG_NONE**
未定义的类型。

winreg.**REG_QWORD**
64 位数字。

3.6 版新加入.

winreg.**REG_QWORD_LITTLE_ENDIAN**
64 位低字节序格式的数字。相当于*REG_QWORD*。

3.6 版新加入.

winreg.**REG_RESOURCE_LIST**
设备驱动程序资源列表。

winreg.**REG_FULL_RESOURCE_DESCRIPTOR**
硬件设置。

winreg.**REG_RESOURCE_REQUIREMENTS_LIST**
硬件资源列表。

winreg.**REG_SZ**
空字符结尾的字符串。

34.2.3 注册表句柄对象

该对象封装了 Windows HKEY 对象，对象销毁时会自动关闭。为确保资源得以清理，可调用*Close()* 方法或*CloseKey()* 函数。

本模块中的所有注册表函数都会返回注册表句柄对象。

本模块中所有接受注册表句柄对象的注册表函数，也能接受一个整数，但鼓励大家使用句柄对象。

注册表句柄对象支持 __bool__() 语义——因此：

```
if handle:
    print("Yes")
```

将会打印出 Yes 。

句柄对象还支持比较语义，因此若多个句柄对象都引用了同一底层 Windows 句柄值，那么比较操作结果将为 True。

句柄对象可转换为整数（如利用内置函数*int()*），这时会返回底层的 Windows 句柄值。用*Detach()* 方法也可返回整数句柄，同时会断开与 Windows 句柄的连接。

PyHKEY.**Close**()
关闭底层的 Windows 句柄。

如果句柄已关闭，不会引发错误。

PyHKEY.**Detach**()
断开与 Windows 句柄的连接。

结果为一个整数，存有被断开连接之前的句柄值。如果该句柄已断开连接或关闭，则返回 0。

调用本函数后，注册表句柄将被迅速禁用，但并没有关闭。当需要底层的 Win32 句柄在句柄对象的生命周期之后仍然存在时，可以调用这个函数。

引发一个审计事件 winreg.PyHKEY.Detach，附带参数 key。

PyHKEY.**__enter__**()
PyHKEY.**__exit__**(**exc_info*)
HKEY 对象实现了 __enter__() 和 __exit__() 方法，因此支持 with 语句的上下文协议：

```
with OpenKey(HKEY_LOCAL_MACHINE, "foo") as key:
    ...  # work with key
```

在离开 with 语句块时，*key* 会自动关闭。

34.3 winsound ——Windows 系统的声音播放接口

通过*winsound* 模块可访问 Windows 平台的基础音频播放机制。包括一些函数和几个常量。

winsound.**Beep**(*frequency*, *duration*)
让 PC 的扬声器发出提示音。*frequency* 参数可指定声音的频率，单位是赫兹，必须位于 37 到 32,767 之间。*duration* 参数则指定了声音应持续的毫秒数。若系统无法让扬声器发声，则会触发*RuntimeError*。

winsound.**PlaySound**(*sound*, *flags*)
由平台 API 调用底层的 PlaySound() 函数。参数 *sound* 可以是文件名、系统音频的别名、*bytes-like object* 的音频数据或 “None“。如何解释取决于 *flags* 的值，可为以下常数的二进制 OR 组合。如果 *sound* 参数为 None，则当前播放的波形音频会全部停止。如果系统报错，则会触发*RuntimeError*。

winsound.**MessageBeep**(*type=MB_OK*)
由平台 API 调用底层的 MessageBeep() 函数。用于播放注册表中指定的音频。*type* 参数指定播放的音频；可能的值是 -1、MB_ICONASTERISK、MB_ICONEXCLAMATION、MB_ICONHAND、MB_ICONQUESTION 和 MB_OK，下面会介绍。值 -1 会生成一个” 简单的嘀声”；若其他的音频无法播放，这是最后的退路。如果系统报错，则会触发*RuntimeError* 。

winsound.**SND_FILENAME**
参数 *sound* 指明 WAV 文件名。不要与*SND_ALIAS* 一起使用。

winsound.**SND_ALIAS**
参数 *sound* 是注册表内关联的音频名称。如果注册表中无此名称，则播放系统默认的声音，除非同时设定了*SND_NODEFAULT* 。如果没有注册默认声音，则会触发*RuntimeError*。请勿与*SND_FILENAME* 一起使用。

所有的 Win32 系统至少支持以下音频名称；大多数系统支持的音频都多于这些：

PlaySound() name	对应的控制面板音频名
'SystemAsterisk'	星号
'SystemExclamation'	叹息声
'SystemExit'	退出 Windows
'SystemHand'	关键性停止音
'SystemQuestion'	问题

例如:

```
import winsound
# Play Windows exit sound.
winsound.PlaySound("SystemExit", winsound.SND_ALIAS)

# Probably play Windows default sound, if any is registered (because
# "*" probably isn't the registered name of any sound).
winsound.PlaySound("*", winsound.SND_ALIAS)
```

winsound.**SND_LOOP**
循环播放音频。为避免阻塞，必须同时使用*SND_ASYNC* 标志。不能与*SND_MEMORY* 一起使用。

winsound.**SND_MEMORY**
PlaySound() 的 *sound* 形参是一个 WAV 文件的内存镜像，作为一个*bytes-like object*。

備🄴: 本模块不支持异步播放音频的内存镜像，所以该标志和*SND_ASYNC* 的组合将触发*RuntimeError*。

winsound.**SND_PURGE**
停止播放指定声音的所有实例。

> **備𝔽:** 新版 Windows 平台不支持本标志。

winsound.**SND_ASYNC**
立即返回，允许异步播放音频。

winsound.**SND_NODEFAULT**
即便找不到指定的音频，也不播放系统默认音频。

winsound.**SND_NOSTOP**
不打断正在播放的音频。

winsound.**SND_NOWAIT**
如果音频驱动程序忙，则立即返回。

> **備𝔽:** 新版 Windows 平台不支持本标志。

winsound.**MB_ICONASTERISK**
播放 SystemDefault 音频。

winsound.**MB_ICONEXCLAMATION**
播放 SystemExclamation 音频。

winsound.**MB_ICONHAND**
播放 SystemHand 音频。

winsound.**MB_ICONQUESTION**
播放 SystemQuestion 音频。

winsound.**MB_OK**
播放 SystemDefault 音频。

CHAPTER 35

Unix 专有服务

本章描述的模块提供了 Unix 操作系统独有特性的接口，在某些情况下也适用于它的某些或许多衍生版。以下为模块概览：

35.1 posix --- 最常见的 POSIX 系统调用

此模块提供了对基于 C 标准和 POSIX 标准（一种稍加修改的 Unix 接口）进行标准化的系统功能的访问。

请勿直接导入此模块。而应导入*os*模块，它提供了此接口的 可移植版本。在 Unix 上，*os*模块提供了*posix*接口的一个超集。在非 Unix 操作系统上*posix*模块将不可用，但会通过*os*接口提供它的一个可用子集。一旦导入了*os*，用它替代*posix*时就 没有性能惩罚。此外，*os*还提供了一些附加功能，例如在 os.environ 中的某个条目被修改时会自动调用*putenv()*。

错误将作为异常被报告；对于类型错误会给出普通异常，而系统调用所报告的异常则会引发*OSError*。

35.1.1 大文件支持

某些操作系统（包括 AIX, HP-UX, Irix 和 Solaris）可对 int 和 long 为 32 位值的 C 编程模型提供大于 2 GiB 的文件的支持。这在通常情况下是以将相关数据长度和偏移类型定义为 64 位值的方式来实现的。这样的文件有时被称为 大文件。

Python 中的大文件支持会在 off_t 的大小超过 long 且 long long 至少与 off_t 一样大时被启用。要启用此模式可能必须在启用特定编译旗标的情况下执行 Python 配置和编译。例如，在最近几个版本的 Irix 中默认启用了大文件支持，但在 Solaris 2.6 和 2.7 中你还需要执行这样的操作：

```
CFLAGS="`getconf LFS_CFLAGS` OPT="-g -O2 $CFLAGS" \
        ./configure
```

在支持大文件的 Linux 系统中，可以这样做：

```
CFLAGS='-D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64' OPT="-g -O2 $CFLAGS" \
        ./configure
```

35.1.2 重要的模块内容

除了*os*模块文档已说明的许多函数，*posix*还定义了下列数据项:

posix.**environ**
一个表示解释器启动时间点的字符串环境的字典。键和值的类型在 Unix 上为 bytes 而在 Windows 上为 str。例如，environ[b'HOME'] (Windows 上的 environ['HOME']) 是你的家目录的路径名，等价于 C 中的 getenv("HOME")。

修改此字典不会影响由*execv()*，*popen()* 或*system()* 所传入的字符串环境；如果你需要修改环境，请将 environ 传给*execve()* 或者为*system()* 或*popen()* 的命令字符串添加变量赋值和 export 语句。

3.2 版更變: 在 Unix 上，键和值为 bytes 类型。

備𝔼: *os* 模块提供了对 environ 的替代实现，它会在被修改时更新环境。还要注意更新*os.environ* 将导致此字典失效。推荐使用这个*os* 模块版本而不是直接访问*posix* 模块。

35.2 pwd --- 用户密码数据库

此模块可以访问 Unix 用户账户名及密码数据库，在所有 Unix 版本上均可使用。

密码数据库中的条目以元组对象返回，属性对应 passwd 中的结构（属性如下所示，可参考 <pwd.h>):

索引	属性	意义
0	pw_name	登录名
1	pw_passwd	密码，可能已经加密
2	pw_uid	用户 ID 数值
3	pw_gid	组 ID 数值
4	pw_gecos	用户名或备注
5	pw_dir	用户主目录
6	pw_shell	用户的命令解释器

其中 uid 和 gid 是整数，其他是字符串，如果找不到对应的项目，抛出*KeyError* 异常。

備𝔼: 传统的 Unix 系统中，pw_passwd 的值通常使用 DES 导出的算法加密（参阅*crypt* 模块)。不过现在的 unix 系统使用 *影子密码系统*。在这些 unix 上，*pw_passwd* 只包含星号（'*'）或字母（'x'），而加密的密码存储在文件 /etc/shadow 中，此文件不是全局可读的。在 *pw_passwd* 中是否包含有用信息是系统相关的。如果可以访问到加密的密码，就需要使用*spwd* 模块了。

本模块定义如下内容:

pwd.**getpwuid**(*uid*)
给定用户的数值 ID，返回密码数据库的对应项目。

pwd.**getpwnam**(*name*)
给定用户名，返回密码数据库的对应项目。

pwd.**getpwall**()
返回密码数据库中所有项目的列表，顺序不是固定的。

也参考:

模块*grp* 针对用户组数据库的接口，与本模块类似。

模块*spwd* 针对影子密码数据库的接口，与本模块类似。

35.3 grp --- 组数据库

该模块提供对 Unix 组数据库的访问。它在所有 Unix 版本上都可用。

Group database entries are reported as a tuple-like object, whose attributes correspond to the members of the `group` structure (Attribute field below, see `<grp.h>`):

索引	属性	意义
0	gr_name	组名
1	gr_passwd	（加密的）组密码；通常为空
2	gr_gid	数字组 ID
3	gr_mem	组内所有成员的用户名

gid 是整数，名称和密码是字符串，成员列表是字符串列表。（注意，大多数用户未根据密码数据库显式列为所属组的成员。请检查两个数据库以获取完整的成员资格信息。还要注意，以 + 或 - 开头的 `gr_name` 可能是 YP/NIS 引用，可能无法通过*`getgrnam()`* 或*`getgrgid()`* 访问。）

本模块定义如下内容：

grp.**getgrgid**(*gid*)
返回给定数字组 ID 的组数据库条目。如果找不到要求的条目，则会引发*`KeyError`* 错误。

3.6 版後已🄵用: 从 Python 3.6 开始，弃用对*`getgrgid()`* 中的 float 或 string 等非 integer 参数的支持。

grp.**getgrnam**(*name*)
返回给定组名的组数据库条目。如果找不到要求的条目，则会引发*`KeyError`* 错误。

grp.**getgrall**()
以任意顺序返回所有可用组条目的列表。

也参考:

模块*pwd* 用户数据库的接口，与此类似。

模块*spwd* 针对影子密码数据库的接口，与本模块类似。

35.4 termios --- POSIX 风格的 tty 控制

此模块提供了针对 tty I/O 控制的 POSIX 调用的接口。有关此类调用的完整描述，请参阅 *termios(3)* Unix 指南页。它仅在当安装时配置了支持 POSIX *termios* 风格的 tty I/O 控制的 Unix 版本上可用。

此模块中的所有函数均接受一个文件描述符 *fd* 作为第一个参数。这可以是一个整数形式的文件描述符，例如 sys.stdin.fileno() 所返回的对象，或是一个*file object*，例如 sys.stdin 本身。

这个模块还定义了与此处所提供的函数一起使用的所有必要的常量；这些常量与它们在 C 中的对应常量同名。请参考你的系统文档了解有关如何使用这些终端控制接口的更多信息。

这个模块定义了以下函数:

termios.**tcgetattr**(*fd*)
: 对于文件描述符 *fd* 返回一个包含 tty 属性的列表，形式如下: [iflag, oflag, cflag, lflag, ispeed, ospeed, cc]，其中 *cc* 为一个包含 tty 特殊字符的列表（每一项都是长度为 1 的字符串，索引号为 VMIN 和 VTIME 的项除外，这些字段如有定义则应为整数）。对旗标和速度以及 *cc* 数组中索引的解读必须使用在*termios* 模块中定义的符号常量来完成。

termios.**tcsetattr**(*fd*, *when*, *attributes*)
: 根据 *attributes* 列表设置文件描述符 *fd* 的 tty 属性，该列表即*tcgetattr()* 所返回的对象。*when* 参数确定何时改变属性: TCSANOW 表示立即改变，TCSADRAIN 表示在传输所有队列输出后再改变，或 TCSAFLUSH 表示在传输所有队列输出并丢失所有队列输入后再改变。

termios.**tcsendbreak**(*fd*, *duration*)
: 在文件描述符 *fd* 上发送一个中断。*duration* 为零表示发送时长为 0.25--0.5 秒的中断；*duration* 非零值的含义取决于具体系统。

termios.**tcdrain**(*fd*)
: 进入等待状态直到写入文件描述符 *fd* 的所有输出都传送完毕。

termios.**tcflush**(*fd*, *queue*)
: 在文件描述符 *fd* 上丢弃队列数据。*queue* 选择器指定哪个队列: TCIFLUSH 表示输入队列，TCOFLUSH 表示输出队列，或 TCIOFLUSH 表示两个队列同时。

termios.**tcflow**(*fd*, *action*)
: 在文件描述符 *fd* 上挂起一战恢复输入或输出。*action* 参数可以为 TCOOFF 表示挂起输出，TCOON 表示重启输出，TCIOFF 表示挂起输入，或 TCION 表示重启输入。

也参考:

模块tty 针对常用终端控制操作的便捷函数。

35.4.1 示例

这个函数可提示输入密码并且关闭回显。请注意其采取的技巧是使用一个单独的*tcgetattr()* 调用和一个 try ... finally 语句来确保旧的 tty 属性无论在何种情况下都会被原样保存:

```
def getpass(prompt="Password: "):
    import termios, sys
    fd = sys.stdin.fileno()
    old = termios.tcgetattr(fd)
    new = termios.tcgetattr(fd)
    new[3] = new[3] & ~termios.ECHO          # lflags
    try:
        termios.tcsetattr(fd, termios.TCSADRAIN, new)
```

(下页继续)

(繼續上一頁)

```
        passwd = input(prompt)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old)
    return passwd
```

35.5 tty --- 终端控制功能

Source code: Lib/tty.py

---

tty 模块定义了将 tty 放入 cbreak 和 raw 模式的函数。

因为它需要*termios* 模块，所以只能在 Unix 上运行。

tty 模块定义了以下函数：

tty.**setraw**(*fd*, *when=termios.TCSAFLUSH*)
将文件描述符 *fd* 的模式更改为 raw 。如果 *when* 被省略，则默认为 `termios.TCSAFLUSH` ，并传递给*termios.tcsetattr()* 。

tty.**setcbreak**(*fd*, *when=termios.TCSAFLUSH*)
将文件描述符 *fd* 的模式更改为 cbreak 。如果 *when* 被省略，则默认为 `termios.TCSAFLUSH` ，并传递给*termios.tcsetattr()* 。

也参考：

模块*termios* 低级终端控制接口。

35.6 pty --- 伪终端工具

源代码： Lib/pty.py

---

pty 模块定义了一些处理“伪终端”概念的操作：启动另一个进程并能以程序方式在其控制终端中进行读写。

由于伪终端处理高度依赖于具体平台，因此此功能只有针对 Linux 的代码。（Linux 代码也可在其他平台上工作，但是未经测试。）

pty 模块定义了下列函数：

pty.**fork**()
分叉。将子进程的控制终端连接到一个伪终端。返回值为 `(pid, fd)`。请注意子进程获得 *pid* 0 而 *fd* 为 *invalid*。父进程返回值为子进程的 *pid* 而 *fd* 为一个连接到子进程的控制终端（并同时连接到子进程的标准输入和输出）的文件描述符。

pty.**openpty**()
打开一个新的伪终端对，如果可能将使用*os.openpty()*，或是针对通用 Unix 系统的模拟代码。返回一个文件描述符对 `(master, slave)`，分别表示主从两端。

pty.**spawn**(*argv*[, *master_read*[, *stdin_read*]])
生成一个进程，并将其控制终端连接到当前进程的标准 io。这常被用来应对坚持要从控制终端读取数据的程序。在 pty 背后生成的进程预期最后将被终止，而且当它被终止时 *spawn* 将会返回。

会向 *master_read* 和 *stdin_read* 函数传入一个文件描述符供它们读取，并且它们总是应当返回一个字节串。为了强制 spawn 在子进程退出之前返回所以应当抛出*OSError*。

两个函数的默认实现在每次函数被调用时将读取并返回至多 1024 个字节。会向 *master_read* 回调传入伪终端的主文件描述符以从子进程读取输出，而向 *stdin_read* 传入文件描述符 0 以从父进程的标准输入读取数据。

从两个回调返回空字节串会被解读为文件结束 (EOF) 条件，在此之后回调将不再被调用。如果 *stdin_read* 发出 EOF 信号则控制终端就不能再与父进程或子进程进行通信。除非子进程将不带任何输入就退出，否则随后 *spawn* 将一直循环下去。如果 *master_read* 发出 EOF 信号则会有相同的行为结果（至少是在 Linux 上）。

如果两个回调都发出 EOF 信号则 *spawn* 可能将永不返回，除非在你的平台上当传入三个空列表时 *select* 会抛出一个错误。这是一个程序缺陷，相关文档见 问题 26228。

从子进程中的*os.waitpid()* 返回退出状态值。

可以使用 `waitstatus_to_exitcode()` 来将退出状态转换为退出码。

引发一个审计事件 `pty.spawn`，附带参数 `argv`。

3.4 版更變: *spawn()* 现在会从子进程的*os.waitpid()* 返回状态值。

35.6.1 示例

以下程序的作用类似于 Unix 命令 *script(1)*，它使用一个伪终端来记录一个"typescript" 里终端进程的所有输入和输出:

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```

35.7 fcntl ——系统调用 fcntl 和 ioctl

本模块基于文件描述符来进行文件控制和 I/O 控制。它是 Unix 系统调用 fcntl() 和 ioctl() 的接口。关于这些调用的完整描述，请参阅 Unix 手册的 *fcntl(2)* 和 *ioctl(2)* 页面。

本模块的所有函数都接受文件描述符 *fd* 作为第一个参数。可以是一个整数形式的文件描述符，比如 sys.stdin.fileno() 的返回结果，或为*io.IOBase* 对象，比如 sys.stdin 提供一个*fileno()*，可返回一个真正的文件描述符。

3.3 版更變: 本模块的操作以前触发的是*IOError*，现在则会触发*OSError*。

3.8 版更變: fcntl 模块现在有了 F_ADD_SEALS 、F_GET_SEALS 和 F_SEAL_* 常量，用于文件描述符*os.memfd_create()* 的封装。

3.9 版更變: On macOS, the fcntl module exposes the F_GETPATH constant, which obtains the path of a file from a file descriptor. On Linux(>=3.15), the fcntl module exposes the F_OFD_GETLK, F_OFD_SETLK and F_OFD_SETLKW constants, which are used when working with open file description locks.

这个模块定义了以下函数:

fcntl.**fcntl**(*fd*, *cmd*, *arg=0*)

对文件描述符 *fd* 执行 *cmd* 操作（能够提供*fileno()* 方法的文件对象也可以接受）。*cmd* 可用的值与操作系统有关，在*fcntl* 模块中可作为常量使用，名称与相关 C 语言头文件中的一样。参数 *arg* 可以是整数或*bytes* 对象。若为整数值，则本函数的返回值是 C 语言 fcntl() 调用的整数返回值。若为字节串，则其代表一个二进制结构，比如由*struct.pack()* 创建的数据。该二进制数据将被复制到一个缓冲区，缓冲区地址传给 C 调用 fcntl()。调用成功后的返回值位于缓冲区内，转换为一个*bytes* 对象。返回的对象长度将与 *arg* 参数的长度相同。上限为 1024 字节。如果操作系统在缓冲区中返回的信息大于 1024 字节，很可能导致内存段冲突，或更为不易察觉的数据错误。

如果 fcntl() 调用失败，会触发*OSError*。

引发一条*auditing* 事件 fcntl.fcntl，参数为 fd 、cmd、arg。

fcntl.**ioctl**(*fd*, *request*, *arg=0*, *mutate_flag=True*)

本函数与*fcntl()* 函数相同，只是参数的处理更加复杂。

request 参数的上限是 32 位。*termios* 模块中包含了可用作 *request* 参数其他常量，名称与相关 C 头文件中定义的相同。

参数 *arg* 可为整数、支持只读缓冲区接口的对象（如*bytes* ）或支持读写缓冲区接口的对象（如*bytearray* ）。

除了最后一种情况，其他情况下的行为都与*fcntl()* 函数一样。

如果传入的是个可变缓冲区，那么行为就由 *mutate_flag* 参数决定。

如果为 False，缓冲区的可变性将被忽略，行为与只读缓冲区一样，只是没有了上述 1024 字节的上限——只要传入的缓冲区能容纳操作系统放入的数据即可。

如果 *mutate_flag* 为 True（默认值），那么缓冲区（实际上）会传给底层的系统调用*ioctl()* ，其返回代码则会回传给调用它的 Python，而缓冲区的新数据则反映了*ioctl()* 的运行结果。这里做了一点简化，因为若是给出的缓冲区少于 1024 字节，首先会被复制到一个 1024 字节长的静态缓冲区再传给*ioctl()* ，然后把结果复制回给出的缓冲区去。

如果 ioctl() 调用失败，则会触发*OSError* 异常。

例子:

```
>>> import array, fcntl, struct, termios, os
>>> os.getpgrp()
13341
>>> struct.unpack('h', fcntl.ioctl(0, termios.TIOCGPGRP, "  "))[0]
13341
>>> buf = array.array('h', [0])
>>> fcntl.ioctl(0, termios.TIOCGPGRP, buf, 1)
0
>>> buf
array('h', [13341])
```

触发一条*auditing 事件* `fcntl.ioctl`，参数为 `fd` 、`request` 、`arg`。

`fcntl.`**`flock`**(*fd*, *operation*)

在文件描述符 *fd* 上执行加锁操作 *operation* (也接受能提供*`fileno()`* 方法的文件对象)。详见 Unix 手册 *`flock(2)`*。(在某些系统中，此函数是用 `fcntl()` 模拟出来的。)

如果 `flock()` 调用失败，就会触发*`OSError`* 异常。

触发一条*审计事件* `fcntl.flock`，参数为 `fd` 、`operation`。

`fcntl.`**`lockf`**(*fd*, *cmd*, *len=0*, *start=0*, *whence=0*)

本质上是对*`fcntl()`* 加锁调用的封装。*fd* 是要加解锁的文件描述符（也接受能提供*`fileno()`* 方法的文件对象），*cmd* 是以下值之一：

- `LOCK_UN` ——解锁
- `LOCK_SH` ——获取一个共享锁
- `LOCK_EX` ——获取一个独占锁

如果 *cmd* 为 `LOCK_SH` 或 `LOCK_EX`，则还可以与 `LOCK_NB` 进行按位或运算，以避免在获取锁时出现阻塞。如果用了 `LOCK_NB`，无法获取锁时将触发*`OSError`*，此异常的 *errno* 属性将被设为 `EACCES` 或 `EAGAIN` (视操作系统而定；为了保证可移植性，请检查这两个值)。至少在某些系统上，只有当文件描述符指向需要写入而打开的文件时，才可以使用 `LOCK_EX`。

len 是要锁定的字节数，*start* 是自 *whence* 开始锁定的字节偏移量，*whence* 与*`io.IOBase.seek()`* 的定义一样。

- `0` ——自文件起始位置（*`os.SEEK_SET`*）。
- `1` ——自缓冲区当前位置（*`os.SEEK_CUR`* ）
- `2`——自文件末尾（*`os.SEEK_END`*)

start 的默认值为 0，表示从文件起始位置开始。*len* 的默认值是 0，表示加锁至文件末尾。*whence* 的默认值也是 0。

触发一条*审计事件* `fcntl.lockf`，参数为 `fd` 、`cmd`、`len`、`start`、`whence`。

示例（都是运行于符合 SVR4 的系统）：

```
import struct, fcntl, os

f = open(...)
rv = fcntl.fcntl(f, fcntl.F_SETFL, os.O_NDELAY)

lockdata = struct.pack('hhllhh', fcntl.F_WRLCK, 0, 0, 0, 0, 0)
rv = fcntl.fcntl(f, fcntl.F_SETLKW, lockdata)
```

注意，在第一个例子中，返回值变量 *rv* 将存有整数；在第二个例子中，该变量中将存有一个*`bytes`* 对象。*lockdata* 变量的结构布局视系统而定——因此可能采用*`flock()`* 调用会更好。

也参考:

模块*os* 如果*os* 模块中存在加锁标志*O_SHLOCK* 和*O_EXLOCK* （仅在 BSD 上），那么*os.open()* 函数提供了*lockf()* 和*flock()* 函数的替代方案。

35.8 resource --- 资源使用信息

该模块提供了测量和控制程序所利用的系统资源的基本机制。

符号常量被用来指定特定的系统资源，并要求获得关于当前进程或其子进程的使用信息。

当系统调用失败时，会触发一个*OSError* 。

exception resource.**error**
一个被弃用的*OSError* 的别名。

3.3 版更變: 根据 **PEP 3151**，这个类是*OSError* 的别名。

35.8.1 资源限制

资源的使用可以通过下面描述的*setrlimit()* 函数来限制。每个资源都被一对限制所控制：一个软限制和一个硬限制。软限制是当前的限制，并且可以由一个进程随着时间的推移而降低或提高。软限制永远不能超过硬限制。硬限制可以降低到大于软限制的任何数值，但不能提高。（只有拥有超级用户有效 UID 的进程才能提高硬限制。）

可以被限制的具体资源取决于系统。它们在 man *getrlimit(2)* 中描述。下面列出的资源在底层操作系统支持的情况下被支持；那些不能被操作系统检查或控制的资源在本模块中没有为这些平台定义。

resource.**RLIM_INFINITY**
用来表示无限资源的极限的常数。

resource.**getrlimit**(*resource*)
返回一个包含 *resource* 当前软限制和硬限制的元组。如果指定了一个无效的资源，则触发*ValueError* ，如果底层系统调用意外失败，则引发*error* 。

resource.**setrlimit**(*resource*, *limits*)
设置 *resource* 的新的消耗极限。参数 *limits* 必须是一个由两个整数组成的元组 (soft, hard) ，描述了新的限制。*RLIM_INFINITY* 的值可以用来请求一个无限的限制。

如果指定了一个无效的资源，如果新的软限制超过了硬限制，或者如果一个进程试图提高它的硬限制，将触发*ValueError* 。当资源的硬限制或系统限制不是无限时，指定一个*RLIM_INFINITY* 的限制将导致*ValueError* 。一个有效 UID 为超级用户的进程可以请求任何有效的限制值，包括无限，但如果请求的限制超过了系统规定的限制，则仍然会产生*ValueError* 。

如果底层系统调用失败，setrlimit 也可能触发*error* 。

VxWorks 只支持设置*RLIMIT_NOFILE* 。

触发一个*auditing event* resource.setrlimit``使用参数 ``resource , limits 。

resource.**prlimit**(*pid*, *resource*[, *limits*])
将*setrlimit()* 和*getrlimit()* 合并为一个函数，支持获取和设置任意进程的资源限制。如果 *pid* 为 0，那么该调用适用于当前进程。*resource* 和 *limits* 的含义与*setrlimit()* 相同，只是 *limits* 是可选的。

当 *limits* 没有给出时，该函数返回进程 *pid* 的 *resource* 限制。当 *limits* 被给定时，进程的 *resource* 限制被设置，并返回以前的资源限制。

当 *pid* 找不到时，触发*ProcessLookupError*；当用户没有进程的权限时，触发*PermissionError*。

触发一个*auditing event* `resource.prlimit` 带有参数 `pid`，`resource`，`limits`。

Availability: Linux 2.6.36 或更新版本带有 glibc 2.13 或更新版本

3.4 版新加入.

这些符号定义了资源的消耗可以通过下面描述的*setrlimit()* 和*getrlimit()* 函数来控制。这些符号的值正是 C 程序所使用的常数。

Unix man 页面 *getrlimit(2)* 列出了可用的资源。注意，并非所有系统都使用相同的符号或相同的值来表示相同的资源。本模块并不试图掩盖平台的差异——没有为某一平台定义的符号在该平台上将无法从本模块中获得。

`resource.`**RLIMIT_CORE**
当前进程可以创建的核心文件的最大大小（以字节为单位）。如果需要更大的核心文件来包含整个进程的镜像，这可能会导致创建一个部分核心文件。

`resource.`**RLIMIT_CPU**
一个进程可以使用的最大处理器时间（以秒为单位）。如果超过了这个限制，一个 `SIGXCPU` 信号将被发送给进程。() 参见*signal* 模块文档，了解如何捕捉这个信号并做一些有用的事情，例如，将打开的文件刷新到磁盘上）。

`resource.`**RLIMIT_FSIZE**
进程可能创建的文件的最大大小。

`resource.`**RLIMIT_DATA**
进程的堆的最大大小（以字节为单位）。

`resource.`**RLIMIT_STACK**
当前进程的调用堆栈的最大大小（字节）。这只影响到多线程进程中主线程的堆栈。

`resource.`**RLIMIT_RSS**
应该提供给进程的最大常驻内存大小。

`resource.`**RLIMIT_NPROC**
当前进程可能创建的最大进程数。

`resource.`**RLIMIT_NOFILE**
当前进程打开的文件描述符的最大数量。

`resource.`**RLIMIT_OFILE**
BSD 对*RLIMIT_NOFILE* 的命名。

`resource.`**RLIMIT_MEMLOCK**
可能被锁定在内存中的最大地址空间。

`resource.`**RLIMIT_VMEM**
进程可能占用的最大映射内存区域。

`resource.`**RLIMIT_AS**
进程可能占用的地址空间的最大区域（以字节为单位）。

`resource.`**RLIMIT_MSGQUEUE**
可分配给 POSIX 消息队列的字节数。

Availability: Linux 2.6.8 或更新版本。

3.4 版新加入.

`resource.`**RLIMIT_NICE**
进程的 Nice 级别的上限（计算为 20 - rlim_cur）。

Availability: Linux 2.6.12 或更新版本

3.4 版新加入.

resource.**RLIMIT_RTPRIO**
实时优先级的上限。

Availability: Linux 2.6.12 或更新版本

3.4 版新加入.

resource.**RLIMIT_RTTIME**
在实时调度下，一个进程在不进行阻塞性系统调用的情况下，可以花费的 CPU 时间限制（以微秒计）。

Availability: Linux 2.6.25 或更新版本

3.4 版新加入.

resource.**RLIMIT_SIGPENDING**
进程可能排队的信号数量。

Availability: Linux 2.6.8 或更新版本。

3.4 版新加入.

resource.**RLIMIT_SBSIZE**
这个用户使用的套接字缓冲区的最大大小（字节数）。这限制了这个用户在任何时候都可以持有的网络内存数量，因此也限制了 mbufs 的数量。

Availability: FreeBSD 9 或更新版本

3.4 版新加入.

resource.**RLIMIT_SWAP**
这个用户 ID 的所有进程可能保留或使用的交换空间的最大大小（字节数）。这个限制只有在 vm.overcommit sysctl 的第 1 位被置 1 时才会被强制执行。请参阅 tuning(7) 以获得关于这个系统检测器的完整介绍。

Availability: FreeBSD 9 或更新版本

3.4 版新加入.

resource.**RLIMIT_NPTS**
该用户 ID 创建的伪终端的最大数量。

Availability: FreeBSD 9 或更新版本

3.4 版新加入.

35.8.2 资源用量

这些函数被用来检索资源使用信息。

resource.**getrusage**(*who*)
这个函数返回一个对象，描述当前进程或其子进程所消耗的资源，由 *who* 参数指定。*who* 参数应该使用下面描述的 RUSAGE_* 常数之一来指定。

一个简单的示例：

```
from resource import *
import time

# a non CPU-bound task
```

(下页继续)

(繼續上一頁)

```
time.sleep(3)
print(getrusage(RUSAGE_SELF))

# a CPU-bound task
for i in range(10 ** 8):
    _ = 1 + 1
print(getrusage(RUSAGE_SELF))
```

返回值的字段分别描述了某一特定系统资源的使用情况，例如，在用户模式下运行的时间或进程从主内存中换出的次数。有些值取决于内部的时钟周期，例如进程使用的内存量。

为了向后兼容，返回值也可以作为一个 16 个元素的元组来访问。

返回值中的 `ru_utime` 和 `ru_stime` 字段是浮点值，分别代表在用户模式下执行的时间和在系统模式下执行的时间。其余的值是整数。关于这些值的详细信息，请查阅 *getrusage(2)* man page 。这里介绍一个简短的摘要。

索引	域	资源
0	ru_utime	用户模式下的时间（浮点数秒）
1	ru_stime	系统模式下的时间（浮点数秒）
2	ru_maxrss	最大的常驻内存大小
3	ru_ixrss	共享内存大小
4	ru_idrss	未共享的内存大小
5	ru_isrss	未共享的堆栈大小
6	ru_minflt	不需要 I/O 的页面故障
7	ru_majflt	需要 I/O 的页面故障数
8	ru_nswap	swap out 的数量
9	ru_inblock	块写入操作数
10	ru_oublock	块输出操作数
11	ru_msgsnd	发送消息数
12	ru_msgrcv	收到消息数
13	ru_nsignals	收到信号数
14	ru_nvcsw	主动上下文切换
15	ru_nivcsw	被动上下文切换

如果指定了一个无效的 *who* 参数，这个函数将触发一个*ValueError* 。在特殊情况下，它也可能触发*error* 异常。

resource.**getpagesize**()
返回一个系统页面的字节数。(这不需要和硬件页的大小相同)。

下面的 `RUSAGE_*` 符号被传递给*getrusage()* 函数，以指定应该为哪些进程提供信息。

resource.**RUSAGE_SELF**
传递给*getrusage()* 以请求调用进程消耗的资源，这是进程中所有线程使用的资源总和。

resource.**RUSAGE_CHILDREN**
传递给*getrusage()* 以请求被终止和等待的调用进程的子进程所消耗的资源。

resource.**RUSAGE_BOTH**
传递给*getrusage()* 以请求当前进程和子进程所消耗的资源。并非所有系统都能使用。

resource.**RUSAGE_THREAD**
传递给*getrusage()* 以请求当前线程所消耗的资源。并非所有系统都能使用。

3.2 版新加入.

35.9 Unix syslog 库例程

此模块提供一个接口到 Unix `syslog` 日常库. 参考 Unix 手册页关于 `syslog` 设施的详细描述.

此模块包装了系统的 `syslog` 例程族。一个能与 syslog 服务器对话的纯 Python 库则以*logging.handlers* 模块中 `SysLogHandler` 类的形式提供。

这个模块定义了以下函数:

`syslog.`**`syslog`**(*message*)
`syslog.`**`syslog`**(*priority*, *message*)
将字符串 *message* 发送到系统日志记录器。如有必要会添加末尾换行符。每条消息都带有一个由 *facility* 和 *level* 组成的优先级标价签。可选的 *priority* 参数默认值为 `LOG_INFO`，它确定消息的优先级。如果未在 *priority* 中使用逻辑或 (`LOG_INFO | LOG_USER`) 对 facility 进行编码，则会使用在*openlog()* 调用中所给定的值。

如果*openlog()* 未在对*syslog()* 的调用之前被调用，则将不带参数地调用 `openlog()`。

引发审计事件 `syslog.syslog` 使用参数 `priority`, `message`.

`syslog.`**`openlog`**([*ident*[, *logoption*[, *facility*]]])
后续*syslog()* 调用的日志选项可以通过调用*openlog()* 来设置。如果日志当前未打开则*syslog()* 将不带参数地调用*openlog()*。

可选的 *ident* 关键字参数是在每条消息前添加的字符串，默认为 `sys.argv[0]` 去除打头的路径部分。可选的 *logoption* 关键字参数（默认为 0）是一个位字段 -- 请参见下文了解可能的组合值。可选的 *facility* 关键字参数 (默认为 `LOG_USER`) 为没有显式编码 facility 的消息设置默认的 facility。

引发审计事件 `syslog.openlog` 使用参数 `ident`, `logoption`, `facility`.

3.2 版更變: 在之前的版本中，不允许关键字参数，并且要求必须有 *ident*。*ident* 的默认值依赖于系统库，它往往为 `python` 而不是 Python 程序文件的实际名称。

`syslog.`**`closelog`**()
重置日志模块值并且调用系统库 `closelog()`.

这使得此模块在初始导入时行为固定。例如，*openlog()* 将在首次调用*syslog()* 时被调用（如果*openlog()* 还未被调用过），并且 *ident* 和其他*openlog()* 形参会被重置为默认值。

引发一个审计事件 `syslog.closelog` 不附带任何参数。

`syslog.`**`setlogmask`**(*maskpri*)
将优先级掩码设为 *maskpri* 并返回之前的掩码值。调用*syslog()* 并附带未在 *maskpri* 中设置的优先级将会被忽略。默认设置为记录所有优先级。函数 `LOG_MASK(pri)` 可计算单个优先级 *pri* 的掩码。函数 `LOG_UPTO(pri)` 可计算包括 *pri* 在内的所有优先级的掩码。

引发一个审计事件 `syslog.setlogmask` 附带参数 `maskpri`。

此模块定义了一下常量:

优先级级别 (高到低): `LOG_EMERG`, `LOG_ALERT`, `LOG_CRIT`, `LOG_ERR`, `LOG_WARNING`, `LOG_NOTICE`, `LOG_INFO`, `LOG_DEBUG`.

设施: `LOG_KERN`, `LOG_USER`, `LOG_MAIL`, `LOG_DAEMON`, `LOG_AUTH`, `LOG_LPR`, `LOG_NEWS`, `LOG_UUCP`, `LOG_CRON`, `LOG_SYSLOG`, `LOG_LOCAL0` to `LOG_LOCAL7`，如果 `<syslog.h>` 中有定义则还有 `LOG_AUTHPRIV`。

日志选项: `LOG_PID`, `LOG_CONS`, `LOG_NDELAY`，如果 `<syslog.h>` 中有定义则还有 `LOG_ODELAY`, `LOG_NOWAIT` 以及 `LOG_PERROR`。

35.9.1 示例

简单示例

一个简单的示例集:

```
import syslog

syslog.syslog('Processing started')
if error:
    syslog.syslog(syslog.LOG_ERR, 'Processing started')
```

一个设置多种日志选项的示例，其中有在日志消息中包含进程 ID，以及将消息写入用于邮件日志记录的目标设施等:

```
syslog.openlog(logoption=syslog.LOG_PID, facility=syslog.LOG_MAIL)
syslog.syslog('E-mail processing initiated...')
```

CHAPTER 36

被取代的模块

本章中描述的模块均已弃用，仅保留用于向后兼容。它们已经被其他模块所取代。

36.1 aifc --- 读写 AIFF 和 AIFC 文件

源代码： Lib/aifc.py

3.11 版後已🄳用: The *aifc* module is deprecated (see **PEP 594** for details).

本模块提供读写 AIFF 和 AIFF-C 文件的支持。AIFF 是音频交换文件格式 (Audio Interchange File Format)，一种用于在文件中存储数字音频采样的格式。AIFF-C 是该格式的更新版本，其中包括压缩音频数据的功能。

音频文件内有许多参数，用于描述音频数据。采样率或帧率是每秒对声音采样的次数。通道数表示音频是单声道，双声道还是四声道。每个通道的每个帧包含一次采样。采样大小是以字节表示的每次采样的大小。因此，一帧由 `nchannels * samplesize`（通道数 * 采样大小）字节组成，而一秒钟的音频包含 `nchannels * samplesize * framerate`（通道数 * 采样大小 * 帧率）字节。

例如，CD 质量的音频采样大小为 2 字节（16 位），使用 2 个声道（立体声），且帧速率为 44,100 帧/秒。这表示帧大小为 4 字节 (2*2)，一秒钟占用 2*2*44100 字节（176,400 字节）。

aifc 模块定义了以下函数：

aifc.**open**(*file*, *mode=None*)

打开一个 AIFF 或 AIFF-C 文件并返回一个对象实例，该实例具有下方描述的方法。参数 *file* 是文件名称字符串或文件对象。当打开文件用于读取时，*mode* 必须为 `'r'` 或 `'rb'`，当打开文件用于写入时，*mode* 必须为 `'w'` 或 `'wb'`。如果该参数省略，则使用 `file.mode` 的值（如果有），否则使用 `'rb'`。当文件用于写入时，文件对象应该支持 seek 操作，除非提前获知写入的采样总数，并使用 `writeframesraw()` 和 `setnframes()`。*open()* 函数可以在 `with` 语句中使用。当 `with` 块执行完毕，将调用*close()* 方法。

3.4 版更變: 支持了 `with` 语句。

当打开文件用于读取时，由*open()* 返回的对象具有以下几种方法：

aifc.**getnchannels**()
返回音频的通道数（单声道为 1，立体声为 2）。

aifc.**getsampwidth**()
返回以字节表示的单个采样的大小。

aifc.**getframerate**()
返回采样率（每秒的音频帧数）。

aifc.**getnframes**()
返回文件中的音频帧总数。

aifc.**getcomptype**()
返回一个长度为 4 的字节数组，描述了音频文件中使用的压缩类型。对于 AIFF 文件，返回值为 `b'NONE'`。

aifc.**getcompname**()
返回一个字节数组，可转换为人类可读的描述，描述的是音频文件中使用的压缩类型。对于 AIFF 文件，返回值为 `b'not compressed'`。

aifc.**getparams**()
返回一个*namedtuple()* (`nchannels, sampwidth, framerate, nframes, comptype, compname`)，与 `get*()` 方法的输出相同。

aifc.**getmarkers**()
返回一个列表，包含音频文件中的所有标记。标记由一个 3 元素的元组组成。第一个元素是标记 ID（整数），第二个是标记位置，从数据开头算起的帧数（整数），第三个是标记的名称（字符串）。

aifc.**getmark**(*id*)
根据传入的标记 *id* 返回元组，元组与*getmarkers()* 中描述的一致。

aifc.**readframes**(*nframes*)
从音频文件读取并返回后续 *nframes* 个帧。返回的数据是一个字符串，包含每个帧所有通道的未压缩采样值。

aifc.**rewind**()
倒回读取指针。下一次*readframes()* 将从头开始。

aifc.**setpos**(*pos*)
移动读取指针到指定的帧上。

aifc.**tell**()
返回当前的帧号。

aifc.**close**()
关闭 AIFF 文件。调用此方法后，对象将无法再使用。

打开文件用于写入时，*open()* 返回的对象具有上述所有方法，但 `readframes()` 和 `setpos()` 除外，并额外具备了以下方法。只有调用了 `set*()` 方法之后，才能调用相应的 `get*()` 方法。在首次调用 `writeframes()` 或 `writeframesraw()` 之前，必须填写除帧数以外的所有参数。

aifc.**aiff**()
创建一个 AIFF 文件，默认创建 AIFF-C 文件，除非文件名以 `'.aiff'` 为后缀，在此情况下默认创建 AIFF 文件。

aifc.**aifc**()
创建一个 AIFF-C 文件。默认创建 AIFF-C 文件，除非文件名以 `'.aiff'` 为后缀，在此情况下默认创建 AIFF 文件。

aifc.**setnchannels**(*nchannels*)
指明音频文件中的通道数。

aifc.**setsampwidth**(*width*)
指明以字节为单位的音频采样大小。

aifc.**setframerate**(*rate*)
指明以每秒帧数表示的采样频率。

aifc.**setnframes**(*nframes*)
指明要写入到音频文件的帧数。如果未设定此形参或者未正确设定，则文件需要支持位置查找。

aifc.**setcomptype**(*type*, *name*)
指明压缩类型。如果未指明，则音频数据将不会被压缩。在 AIFF 文件中，压缩是无法实现的。name 形参应当为以字节数组表示的人类可读的压缩类型描述，type 形参应当为长度为 4 的字节数组。目前支持的压缩类型如下: `b'NONE'`,`b'ULAW'`,`b'ALAW'`,`b'G722'`。

aifc.**setparams**(*nchannels*, *sampwidth*, *framerate*, *comptype*, *compname*)
一次性设置上述所有参数。该参数是由多个形参组成的元组。这意味着可以使用*getparams()* 调用的结果作为*setparams()* 的参数。

aifc.**setmark**(*id*, *pos*, *name*)
添加具有给定 id (大于 0)，以及在给定位置上给定名称的标记。此方法可在*close()* 之前的任何时候被调用。

aifc.**tell**()
返回输出文件中的当前写入位置。适用于与*setmark()* 进行协同配合。

aifc.**writeframes**(*data*)
将数据写入到输出文件。此方法只能在设置了音频文件形参之后被调用。

3.4 版更變: 现在可接受任意*bytes-like object*。

aifc.**writeframesraw**(*data*)
类似于*writeframes()*，不同之处在于音频文件的标头不会被更新。

3.4 版更變: 现在可接受任意*bytes-like object*。

aifc.**close**()
关闭 AIFF 文件。文件的标头会被更新以反映音频数据的实际大小。在调用此方法之后，对象将无法再被使用。

36.2 asynchat --- 异步套接字指令/响应处理程序

源代码: Lib/asynchat.py

3.6 版後已F用: *asynchat* will be removed in Python 3.12 (see **PEP 594** for details). Please use *asyncio* instead.

備F: 该模块仅为提供向后兼容。我们推荐在新代码中使用*asyncio* 。

此模块在*asyncore* 框架之上构建，简化了异步客户端和服务器并使得处理元素为以任意字符串结束或者为可变长度的协议更加容易。*asynchat* 定义了一个可以由你来子类化的抽象类*async_chat*，提供了 `collect_incoming_data()` 和 `found_terminator()` 等方法的实现。它使用与*asyncore* 相同的异步循环，并且可以在通道映射中自由地混合*asyncore.dispatcher* 和*asynchat.async_chat* 这两种类型的通道。一般来说*asyncore.dispatcher* 服务器通道在接收到传入的连接请求时会生成新的*asynchat.async_chat* 通道对象。

class asynchat.**async_chat**
这个类是*asyncore.dispatcher* 的抽象子类。对于实际使用的代码你必须子类

化*async_chat*，提供有意义的*collect_incoming_data()* 和*found_terminator()* 方法。*asyncore.dispatcher* 的方法也可以被使用，但它们在消息/响应上下文中并不是全都有意义。

与*asyncore.dispatcher* 类似，*async_chat* 也定义了一组通过对 `select()` 调用之后的套接字条件进行分析所生成的事件。一旦启动轮询循环*async_chat* 对象的方法就会被事件处理框架调用而无须程序员方面做任何操作。

两个可被修改的类属性，用以提升性能，甚至也可能会节省内存。

ac_in_buffer_size
异步输入缓冲区大小(默认为 `4096`)。

ac_out_buffer_size
异步输出缓冲区大小(默认为 `4096`)。

与*asyncore.dispatcher* 不同，*async_chat* 允许你定义一个 FIFO 队列 *producers*。其中的生产者只需要一个方法 `more()`，该方法应当返回要在通道上传输的数据。生产者通过让其 `more()` 方法返回空字节串对象来表明其处于耗尽状态 (意即它已不再包含数据)。此时*async_chat* 对象会将该生产者从队列中移除并开始使用下一个生产者，如果有下一个的话。当生产者队列为空时 `handle_write()` 方法将不执行任何操作。你要使用通道对象的*set_terminator()* 方法来描述如何识别来自远程端点的入站传输的结束或是重要的中断点。

要构建一个可用的*async_chat* 子类，你的输入方法*collect_incoming_data()* 和*found_terminator()* 必须要处理通道异步接收的数据。这些参数的描述见下文。

`async_chat.`**`close_when_done`**`()`
将 `None` 推入生产者队列。当此生产者被弹出队列时它将导致通道被关闭。

`async_chat.`**`collect_incoming_data`**`(`*data*`)`
调用时附带 *data*，其中包含任意数量的已接收数据。必须被重载的默认方法将引发一个*NotImplementedError* 异常。

`async_chat.`**`discard_buffers`**`()`
在紧急情况下此方法将丢弃输入和/或输出缓冲区以及生产者队列中的任何数据。

`async_chat.`**`found_terminator`**`()`
当输入数据流能匹配*set_terminator()* 所设定的终结条件时会被调用。必须被重载的默认方法将引发一个*NotImplementedError* 异常。被缓冲的输入数据应当可以通过实例属性来获取。

`async_chat.`**`get_terminator`**`()`
返回通道的当前终结器。

`async_chat.`**`push`**`(`*data*`)`
将数据推入通道的队列以确保其被传输。要让通道将数据写到网络中你只需要这样做就足够了，虽然以更复杂的方式使用你自己的生产者也是有可能的，例如为了实现加密和分块。

`async_chat.`**`push_with_producer`**`(`*producer*`)`
获取一个生产者对象并将其加入到与通道相关联的生产者队列中。当所有当前已推入的生产者都已被耗尽时通道将通过调用其 `more()` 方法来耗用此生产者的数据并将数据发送至远程端点。

`async_chat.`**`set_terminator`**`(`*term*`)`
设置可在通道上被识别的终结条件。`term` 可以是三种类型值中的任意一种，对应于处理入站协议数据的三种不同方式。

term	描述
string	当在输入流中发现该字符串时将会调用*found_terminator()*
integer	当接收到指定数量的字符时将会调用*found_terminator()*
`None`	通道会不断地持续收集数据

请注意终结器之后的任何数据将可在*found_terminator()* 被调用后由通道来读取。

36.2.1 asynchat 示例

下面的例子片段显示了如何通过*async_chat* 来读取 HTTP 请求。Web 服务器可以为每个入站的客户端连接创建 `http_request_handler` 对象。请注意在初始时通道终结器会被设置为匹配 HTTP 标头末尾的空行，并且会用一个旗标来指明标头正在被读取。

一旦完成了标头的读取，如果请求类型为 POST (表明输入流中存在更多的数据) 则会使用 `Content-Length:` 标头来设置一个数值终结器以从通道读取适当数量的数据。

一旦完成了对所有相关输入的处理，将会在设置通道终结器为 `None` 以确保忽略掉 Web 客户端所发送的任何无关数据之后调用 `handle_request()` 方法。

```
import asynchat

class http_request_handler(asynchat.async_chat):

    def __init__(self, sock, addr, sessions, log):
        asynchat.async_chat.__init__(self, sock=sock)
        self.addr = addr
        self.sessions = sessions
        self.ibuffer = []
        self.obuffer = b""
        self.set_terminator(b"\r\n\r\n")
        self.reading_headers = True
        self.handling = False
        self.cgi_data = None
        self.log = log

    def collect_incoming_data(self, data):
        """Buffer the data"""
        self.ibuffer.append(data)

    def found_terminator(self):
        if self.reading_headers:
            self.reading_headers = False
            self.parse_headers(b"".join(self.ibuffer))
            self.ibuffer = []
            if self.op.upper() == b"POST":
                clen = self.headers.getheader("content-length")
                self.set_terminator(int(clen))
            else:
                self.handling = True
                self.set_terminator(None)
                self.handle_request()
        elif not self.handling:
            self.set_terminator(None)  # browsers sometimes over-send
            self.cgi_data = parse(self.headers, b"".join(self.ibuffer))
            self.handling = True
            self.ibuffer = []
            self.handle_request()
```

36.3 asyncore --- 异步套接字处理器

源码： Lib/asyncore.py

3.6 版後已🄵用: *asyncore* will be removed in Python 3.12 (see **PEP 594** for details). Please use *asyncio* instead.

備🄵： 该模块仅为提供向后兼容。我们推荐在新代码中使用*asyncio* 。

该模块提供用于编写异步套接字服务客户端与服务端的基础构件。

只有两种方法让单个处理器上的程序“同一时间完成不止一件事”。多线程编程是最简单和最流行的方法，但是还有另一种非常不同的技术，它可以让你拥有多线程的几乎所有优点，而无需实际使用多线程。它仅仅在你的程序主要受 I/O 限制时有用，那么。如果你的程序受处理器限制，那么先发制人的预定线程可能就是你真正需要的。但是，网络服务器很少受处理器限制。

如果你的操作系统在其 I/O 库中支持 `select()` 系统调用（几乎所有操作系统），那么你可以使用它来同时处理多个通信通道；在 I/O 正在“后台”时进行其他工作。虽然这种策略看起来很奇怪和复杂，特别是起初，它在很多方面比多线程编程更容易理解和控制。*asyncore* 模块为您解决了许多难题，使得构建复杂的高性能网络服务器和客户端的任务变得轻而易举。对于“会话”应用程序和协议，伴侣*asynchat* 模块是非常宝贵的。

这两个模块背后的基本思想是创建一个或多个网络 *通道*，类的实例*asyncore.dispatcher* 和*asynchat.async_chat* 。创建通道会将它们添加到全局映射中，如果你不为它提供自己的 *映射*，则由*loop()* 函数使用。

一旦创建了初始通道，调用*loop()* 函数将激活通道服务，该服务将一直持续到最后一个通道（包括在异步服务期间已添加到映射中的任何通道）关闭。

asyncore.**loop**([*timeout*[, *use_poll*[, *map*[, *count*]]]])

进入一个轮询循环，其在循环计数超出或所有打开的通道关闭后终止。所有参数都是可选的。*count* 形参默认为 `None` ，导致循环仅在所有通道关闭时终止。*timeout* 形参为适当的*select()* 或*poll()* 调用设置超时参数，以秒为单位; 默认值为 30 秒。*use_poll* 形参，如果为 True ，则表示*poll()* 应优先使用*select()* (默认为 “False“)。

map 形参是一个条目为所监视通道的字典。当通道关闭时它们会被从映射中删除。如果省略 *map*，则会使用一个全局映射。通道 (*asyncore.dispatcher*, *asynchat.async_chat* 及其子类的实例) 可以在映射中任意混合。

class asyncore.**dispatcher**

dispatcher 类是对低层级套接字对象的轻量包装器。要让它更有用处，可以从异步循环调用一些事件处理方法。在其他方面，它可以被当作是普通的非阻塞型套接字对象。

在特定时间或特定连接状态下触发的低层级事件可通知异步循环发生了特定的高层级事件。例如，如果我们请求了一个套接字以连接到另一台主机，我们会在套接字首次变得可写时得知连接已建立（在此刻你将知道可以向其写入并预期能够成功）。包含的高层级事件有:

事件	描述
`handle_connect()`	由首个读取或写入事件引起
`handle_close()`	由不带可用数据的读取事件引起
`handle_accepted()`	由在监听套接字上的读取事件引起

在异步处理过程中，每个已映射通道的*readable()* 和*writable()* 方法会被用来确定是否要将通道的套接字添加到已执行 `select()` 或 `poll()` 用于读取和写入事件的通道列表中。

因此，通道事件的集合要大于基本套接字事件。可以在你的子类中被重载的全部方法集合如下:

handle_read()
当异步循环检测到通道的套接字上的 `read()` 调用将要成功时会被调用。

handle_write()
当异步循环检测到一个可写套接字可以被写入时会被调用。通常此方法将实现必要的缓冲机制以保证运行效率。例如:

```
def handle_write(self):
    sent = self.send(self.buffer)
    self.buffer = self.buffer[sent:]
```

handle_expt()
当一个套接字连接存在带外（OOB）数据时会被调用。这几乎从来不会发生，因为 OOB 虽然受支持但很少被使用。

handle_connect()
当活动打开方的套接字实际建立连接时会被调用。可能会发送一条“欢迎”消息，或者向远程端点发起协议协商等。

handle_close()
当套接字关闭时会被调用。

handle_error()
当一个异常被引发并且未获得其他处理时会被调用。默认版本将打印精简的回溯信息。

handle_accept()
当可以与发起对本地端点的*connect()* 调用的新远程端点建立连接时会在侦听通道（被动打开方）上被调用。在 3.2 版中已被弃用；请改用*handle_accepted()*。

3.2 版後已🄵用.

handle_accepted(*sock*, *addr*)
当与发起对本地端点的*connect()* 调用的新远程端点已建立连接时会在侦听通道（被动打开方）上被调用。*sock* 是可被用于在连接上发送和接收数据的 *新建*套接字对象，而 *addr* 是绑定到连接另一端的套接字的地址。

3.2 版新加入.

readable()
每次在异步循环之外被调用以确定是否应当将一个通道的套接字添加到可能在其上发生读取事件的列表中。默认方法会简单地返回 `True`，表示在默认情况下，所有通道都希望能读取事件。

writable()
每次在异步循环之外被调用以确定是否应当将一个通道的套接字添加到可能在其上发生写入事件的列表中。默认方法会简单地返回 `True`，表示在默认情况下，所有通道都希望能写入事件。

此外，每个通道都委托或扩展了许多套接字方法。它们大部分都与其套接字的对应方法几乎一样。

create_socket(*family=socket.AF_INET*, *type=socket.SOCK_STREAM*)
这与普通套接字的创建相同，并会使用同样的创建选项。请参阅*socket* 文档了解有关创建套接字的信息。

3.3 版更變: *family* 和 *type* 参数可以被省略。

connect(*address*)
与普通套接字对象一样，*address* 是一个元组，它的第一个元素是要连接的主机，第二个元素是端口号。

send(*data*)
将 *data* 发送到套接字的远程端点。

recv(*buffer_size*)
从套接字的远程端点读取至多 *buffer_size* 个字节。读到空字节串表明通道已从另一端被关闭。

请注意*recv()* 可能会引发*BlockingIOError*，即使*select.select()* 或*select.poll()* 报告套接字已准备好被读取。

listen(*backlog*)
侦听与套接字的连接。*backlog* 参数指明排入连接队列的最大数量且至少应为 1；最大值取决于具体系统（通常为 5）。

bind(*address*)
将套接字绑定到 *address*。套接字必须尚未被绑定。(*address* 的格式取决于具体的地址族 --- 请参阅*socket* 文档了解更多信息。) 要将套接字标记为可重用的 (设置 SO_REUSEADDR 选项)，请调用*dispatcher* 对象的 set_reuse_addr() 方法。

accept()
接受一个连接。此套接字必须绑定到一个地址上并且侦听连接。返回值可以是 None 或一个 (conn, address) 对，其中 *conn* 是一个可用来在此连接上发送和接收数据的 *新的*套接字对象，而 *address* 是绑定到连接另一端套接字的地址。当返回 None 时意味着连接没有建立，在此情况下服务器应当忽略此事件并继续侦听后续的入站连接。

close()
关闭套接字。在此套接字对象上的后续操作都将失败。远程端点将不再接收任何数据（在排入队列的数据被清空之后）。当套接字被垃圾回收时会自动关闭。

class asyncore.**dispatcher_with_send**
dispatcher 的一个添加了简单缓冲输出功能的子类，适用于简单客户端。对于更复杂的用法请使用*asynchat.async_chat*。

class asyncore.**file_dispatcher**
file_dispatcher 接受一个文件描述符或*file object* 以及一个可选的 map 参数，并对其进行包装以配合 poll() 或 loop() 函数使用。如果提供一个文件对象或任何具有 fileno() 方法的对象，其方法将被调用并传递给*file_wrapper* 构造器。

Availability: Unix.

class asyncore.**file_wrapper**
file_wrapper 接受一个整数形式的文件描述符并调用*os.dup()* 来复制其句柄，以便原始句柄可以独立于 file_wrapper 被关闭。这个类实现了足够的方法来模拟套接字以供*file_dispatcher* 类使用。

Availability: Unix.

36.3.1 asyncore 示例基本 HTTP 客户端

下面是一个非常基本的 HTTP 客户端，它使用了*dispatcher* 类来实现套接字处理:

```
import asyncore

class HTTPClient(asyncore.dispatcher):

    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.connect( (host, 80) )
        self.buffer = bytes('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' %
                            (path, host), 'ascii')

    def handle_connect(self):
```

(下页继续)

(繼續上一頁)

```
        pass

    def handle_close(self):
        self.close()

    def handle_read(self):
        print(self.recv(8192))

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]


client = HTTPClient('www.python.org', '/')
asyncore.loop()
```

36.3.2 asyncore 示例基本回显服务器

下面是一个基本的回显服务器，它使用了*dispatcher*类来接受连接并将入站连接发送给处理程序:

```
import asyncore

class EchoHandler(asyncore.dispatcher_with_send):

    def handle_read(self):
        data = self.recv(8192)
        if data:
            self.send(data)

class EchoServer(asyncore.dispatcher):

    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
        self.create_socket()
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(5)

    def handle_accepted(self, sock, addr):
        print('Incoming connection from %s' % repr(addr))
        handler = EchoHandler(sock)

server = EchoServer('localhost', 8080)
asyncore.loop()
```

36.4 audioop --- 处理原始音频数据

3.11 版後已𝔽用: The *audioop* module is deprecated (see **PEP 594** for details).

---

audioop 模块包含针对声音片段的一些有用操作。它操作的声音片段由 8、16、24 或 32 位宽的有符号整型采样值组成，存储在类字节串对象 中。除非特别说明，否则所有标量项目均为整数。

3.4 版更變: 增加了对 24 位采样的支持。现在，所有函数都接受任何类字节串对象。而传入字符串会立即导致错误。

本模块提供对 a-LAW、u-LAW 和 Intel/DVI ADPCM 编码的支持。

部分更复杂的操作仅接受 16 位采样，而其他操作始终需要采样大小（以字节为单位）作为该操作的参数。

此模块定义了下列变量和函数：

exception audioop.**error**
所有错误都会抛出此异常，比如采样值的字节数未知等等。

audioop.**add**(*fragment1*, *fragment2*, *width*)
两个采样作为参数传入，返回一个片段，该片段是两个采样的和。*width* 是采样位宽（以字节为单位），可以取 `1`, `2`, `3` 或 `4`。两个片段的长度应相同。如果发生溢出，较长的采样将被截断。

audioop.**adpcm2lin**(*adpcmfragment*, *width*, *state*)
将 Intel/DVI ADPCM 编码的片段解码为线性片段。关于 ADPCM 编码的详情请参阅*lin2adpcm()* 的描述。返回一个元组 `(sample, newstate)`，其中 sample 的位宽由 *width* 指定。

audioop.**alaw2lin**(*fragment*, *width*)
将 a-LAW 编码的声音片段转换为线性编码声音片段。由于 a-LAW 编码采样值始终为 8 位，因此这里的 *width* 仅指输出片段的采样位宽。

audioop.**avg**(*fragment*, *width*)
返回片段中所有采样值的平均值。

audioop.**avgpp**(*fragment*, *width*)
返回片段中所有采样值的平均峰峰值。由于没有进行过滤，因此该例程的实用性尚存疑。

audioop.**bias**(*fragment*, *width*, *bias*)
返回一个片段，该片段由原始片段中的每个采样值加上偏差组成。在溢出时采样值会回卷 (wrap around)。

audioop.**byteswap**(*fragment*, *width*)
“按字节交换”片段中的所有采样值，返回修改后的片段。将大端序采样转换为小端序采样，反之亦然。

3.4 版新加入.

audioop.**cross**(*fragment*, *width*)
将片段作为参数传入，返回其中过零点的数量。

audioop.**findfactor**(*fragment*, *reference*)
返回一个系数 *F* 使得 `rms(add(fragment, mul(reference, -F)))` 最小，即返回的系数乘以 *reference* 后与 *fragment* 最匹配。两个片段都应包含 2 字节宽的采样。

本例程所需的时间与 `len(fragment)` 成正比。

audioop.**findfit**(*fragment*, *reference*)
尽可能尝试让 *reference* 匹配 *fragment* 的一部分（*fragment* 应较长）。从概念上讲，完成这些靠从 *fragment* 中取出切片，使用*findfactor()* 计算最佳匹配，并最小化结果。两个片段都应包含 2 字节宽的采样。返回一个元组 `(offset, factor)`，其中 *offset* 是在 *fragment* 中的偏移量（整数），表示从此处开始最佳匹配，而 *factor* 是由*findfactor()* 定义的因数（浮点数）。

audioop.**findmax**(*fragment*, *length*)
在 *fragment* 中搜索所有长度为 *length* 的采样切片（不是字节!）中，能量最大的那一个切片，即返回 *i* 使得 `rms(fragment[i*2:(i+length)*2])` 最大。两个片段都应包含 2 字节宽的采样。

本例程所需的时间与 `len(fragment)` 成正比。

audioop.**getsample**(*fragment*, *width*, *index*)
返回片段中采样值索引 *index* 的值。

audioop.**lin2adpcm**(*fragment*, *width*, *state*)
将采样转换为 4 位 Intel/DVI ADPCM 编码。ADPCM 编码是一种自适应编码方案，其中每个 4 比特数字是一个采样值与下一个采样值之间的差除以（不定的）步长。IMA 已选择使用 Intel/DVI ADPCM 算法，因此它很可能成为标准。

state 是一个表示编码器状态的元组。编码器返回一个元组 `(adpcmfrag, newstate)`，而 *newstate* 要在下一次调用 *lin2adpcm()* 时传入。在初始调用中，可以将 `None` 作为 state 传递。*adpcmfrag* 是 ADPCM 编码的片段，每个字节打包了 2 个 4 比特值。

audioop.**lin2alaw**(*fragment*, *width*)
将音频片段中的采样值转换为 a-LAW 编码，并将其作为字节对象返回。a-LAW 是一种音频编码格式，仅使用 8 位采样即可获得大约 13 位的动态范围。Sun 音频硬件等使用该编码。

audioop.**lin2lin**(*fragment*, *width*, *newwidth*)
将采样在 1、2、3 和 4 字节格式之间转换。

備F: 在某些音频格式（如.WAV 文件）中，16、24 和 32 位采样是有符号的，但 8 位采样是无符号的。因此，当将这些格式转换为 8 位宽采样时，还需使结果加上 128:

```
new_frames = audioop.lin2lin(frames, old_width, 1)
new_frames = audioop.bias(new_frames, 1, 128)
```

反之，将 8 位宽的采样转换为 16、24 或 32 位时，必须采用相同的处理。

---

audioop.**lin2ulaw**(*fragment*, *width*)
将音频片段中的采样值转换为 u-LAW 编码，并将其作为字节对象返回。u-LAW 是一种音频编码格式，仅使用 8 位采样即可获得大约 14 位的动态范围。Sun 音频硬件等使用该编码。

audioop.**max**(*fragment*, *width*)
返回片段中所有采样值的最大 绝对值。

audioop.**maxpp**(*fragment*, *width*)
返回声音片段中的最大峰峰值。

audioop.**minmax**(*fragment*, *width*)
返回声音片段中所有采样值的最小值和最大值组成的元组。

audioop.**mul**(*fragment*, *width*, *factor*)
返回一个片段，该片段由原始片段中的每个采样值乘以浮点值 *factor* 组成。如果发生溢出，采样将被截断。

audioop.**ratecv**(*fragment*, *width*, *nchannels*, *inrate*, *outrate*, *state*[, *weightA*[, *weightB*]])
转换输入片段的帧速率。

state 是一个表示转换器状态的元组。转换器返回一个元组 `(newfragment, newstate)`，而 *newstate* 要在下一次调用 *ratecv()* 时传入。初始调用应传入 `None` 作为 state。

参数 *weightA* 和 *weightB* 是简单数字滤波器的参数，默认分别为 `1` 和 `0`。

audioop.**reverse**(*fragment*, *width*)
将片段中的采样值反转，返回修改后的片段。

audioop.**rms**(*fragment*, *width*)
返回片段的均方根值，即 `sqrt(sum(S_i^2)/n)`。

测量音频信号的能量。

audioop.**tomono**(*fragment*, *width*, *lfactor*, *rfactor*)
将立体声片段转换为单声道片段。左通道乘以 *lfactor*，右通道乘以 *rfactor*，然后两个通道相加得到单声道信号。

audioop.**tostereo**(*fragment*, *width*, *lfactor*, *rfactor*)
由单声道片段生成立体声片段。立体声片段中的两对采样都是从单声道计算而来的，即左声道是乘以 *lfactor*，右声道是乘以 *rfactor*。

audioop.**ulaw2lin**(*fragment*, *width*)
将 u-LAW 编码的声音片段转换为线性编码声音片段。由于 u-LAW 编码采样值始终为 8 位，因此这里的 *width* 仅指输出片段的采样位宽。

请注意，诸如*mul()* 或*max()* 之类的操作在单声道和立体声间没有区别，即所有采样都作相同处理。如果出现问题，应先将立体声片段拆分为两个单声道片段，之后再重组。以下是如何进行该操作的示例:

```
def mul_stereo(sample, width, lfactor, rfactor):
    lsample = audioop.tomono(sample, width, 1, 0)
    rsample = audioop.tomono(sample, width, 0, 1)
    lsample = audioop.mul(lsample, width, lfactor)
    rsample = audioop.mul(rsample, width, rfactor)
    lsample = audioop.tostereo(lsample, width, 1, 0)
    rsample = audioop.tostereo(rsample, width, 0, 1)
    return audioop.add(lsample, rsample, width)
```

如果使用 ADPCM 编码器构造网络数据包，并且希望协议是无状态的（即能够容忍数据包丢失），则不仅需要传输数据，还应该传输状态。请注意，必须将 * 初始 * 状态（传入*lin2adpcm()* 的状态）发送给解码器，不能发送最终状态（编码器返回的状态）。如果要使用*struct.Struct* 以二进制保存状态，可以将第一个元素（预测值）用 16 位编码，将第二个元素（增量索引）用 8 位编码。

本 ADPCM 编码器从不与其他 ADPCM 编码器对立，仅针对自身。本开发者可能会误读标准，这种情况下它们将无法与相应标准互操作。

乍看之下 `find*()` 例程可能有些可笑。它们主要是用于回声消除，一种快速有效的方法是选取输出样本中能量最高的片段，在输入样本中定位该片段，然后从输入样本中减去整个输出样本:

```
def echocancel(outputdata, inputdata):
    pos = audioop.findmax(outputdata, 800)    # one tenth second
    out_test = outputdata[pos*2:]
    in_test = inputdata[pos*2:]
    ipos, factor = audioop.findfit(in_test, out_test)
    # Optional (for better cancellation):
    # factor = audioop.findfactor(in_test[ipos*2:ipos*2+len(out_test)],
    #              out_test)
    prefill = '\0'*(pos+ipos)*2
    postfill = '\0'*(len(inputdata)-len(prefill)-len(outputdata))
    outputdata = prefill + audioop.mul(outputdata, 2, -factor) + postfill
    return audioop.add(inputdata, outputdata, 2)
```

36.5 cgi --- 通用网关接口支持

源代码： Lib/cgi.py

3.11 版後已🄵用: The *cgi* module is deprecated (see **PEP 594** for details and alternatives).

通用网关接口 (CGI) 脚本的支持模块

本模块定义了一些工具供以 Python 编写的 CGI 脚本使用。

36.5.1 简介

CGI 脚本是由 HTTP 服务器发起调用，通常用来处理通过 HTML `<FORM>` 或 `<ISINDEX>` 元素提交的用户输入。

在大多数情况下，CGI 脚本存放在服务器的 `cgi-bin` 特殊目录下。HTTP 服务器将有关请求的各种信息（例如客户端的主机名、所请求的 URL、查询字符串以及许多其他内容）放在脚本的 shell 环境中，然后执行脚本，并将脚本的输出发回到客户端。

脚本的输入也会被连接到客户端，并且有时表单数据也会以此方式来读取；在其他时候表单数据会通过 URL 的“查询字符串”部分来传递。本模块的目标是处理不同的应用场景并向 Python 脚本提供一个更为简单的接口。它还提供了一些工具为脚本调试提供帮助，而最近增加的还有对通过表单上传文件的支持（如果你的浏览器支持该功能的话）。

CGI 脚本的输出应当由两部分组成，并由一个空行分隔。前一部分包含一些标头，它们告诉客户端后面会提供何种数据。生成一个最小化标头部分的 Python 代码如下所示:

```
print("Content-Type: text/html")    # HTML is following
print()                             # blank line, end of headers
```

后一部分通常为 HTML，提供给客户端软件来显示格式良好包含标题的文本、内联图片等内容。下面是打印一段简单 HTML 的 Python 代码:

```
print("<TITLE>CGI script output</TITLE>")
print("<H1>This is my first CGI script</H1>")
print("Hello, world!")
```

36.5.2 使用 cgi 模块。

通过敲下 `import cgi` 来开始。

当你在写一个新脚本时，考虑加上这些语句:

```
import cgitb
cgitb.enable()
```

这会激活一个特殊的异常处理句柄，它将在发生任何错误时将详细错误报告显示到 Web 浏览器中。如果你不希望向你的脚本的用户显示你的程序的内部细节，你可以改为将报告保存到文件中，使用这样的代码即可:

```
import cgitb
cgitb.enable(display=0, logdir="/path/to/logdir")
```

在脚本开发期间使用此特性会很有帮助。*cgitb* 所产生的报告提供了在追踪程序问题时能为你节省大量时间的信息。你可以在完成测试你的脚本并确信它能正确工作之后再移除 cgitb 行。

To get at submitted form data, use the FieldStorage class. If the form contains non-ASCII characters, use the *encoding* keyword parameter set to the value of the encoding defined for the document. It is usually contained in the META tag in the HEAD section of the HTML document or by the *Content-Type* header. This reads the form contents from the standard input or the environment (depending on the value of various environment variables set according to the CGI standard). Since it may consume standard input, it should be instantiated only once.

FieldStorage 实例可以像 Python 字典一样来检索。它允许通过 in 运算符进行成员检测，也支持标准字典方法*keys()* 和内置函数*len()*。包含空字符串的表单字段会被忽略而不会出现在字典中；要保留这样的值，请在创建 FieldStorage 实例时为可选的 *keep_blank_values* 关键字形参提供一个真值。

举例来说，下面的代码（假定 *Content-Type* 标头和空行已经被打印）会检查字段 name 和 addr 是否均被设为非空字符串:

```
form = cgi.FieldStorage()
if "name" not in form or "addr" not in form:
    print("<H1>Error</H1>")
    print("Please fill in the name and addr fields.")
    return
print("<p>name:", form["name"].value)
print("<p>addr:", form["addr"].value)
...further form processing here...
```

在这里的字段通过 form[key] 来访问，它们本身就是 FieldStorage (或 MiniFieldStorage，取决于表单的编码格式) 的实例。实例的 value 属性会产生字段的字符串值。getvalue() 方法直接返回这个字符串；它还接受可选的第二个参数作为当请求的键不存在时要返回的默认值。

如果提交的表单数据包含一个以上的同名字段，由 form[key] 所提取的对象将不是一个 FieldStorage 或 MiniFieldStorage 实例而是由这种实例组成的列表。类似地，在这种情况下，form.getvalue(key) 将会返回一个字符串列表。如果你预计到这种可能性（当你的 HTML 表单包含多个同名字段时），请使用*getlist()* 方法，它总是返回一个值的列表（这样你就不需要对只有单个项的情况进行特别处理）。例如，这段代码拼接了任意数量的 username 字段，以逗号进行分隔:

```
value = form.getlist("username")
usernames = ",".join(value)
```

如果一个字段是代表上传的文件，请通过 value 属性访问该值或是通过 getvalue() 方法以字节形式将整个文件读入内存。这可能不是你想要的结果。你可以通过测试 filename 属性或 file 属性来检测上传的文件。然后你可以从 file 属性读取数据，直到它作为 FieldStorage 实例的垃圾回收的一部分被自动关闭 (*read()* 和*readline()* 方法将返回字节数据):

```
fileitem = form["userfile"]
if fileitem.file:
    # It's an uploaded file; count lines
    linecount = 0
    while True:
        line = fileitem.file.readline()
        if not line: break
        linecount = linecount + 1
```

FieldStorage 对象还支持在 with 语句中使用，该语句结束时将自动关闭它们。

如果在获取上传文件的内容时遇到错误（例如，当用户点击回退或取消按钮中断表单提交时）该字段中对象的 done 属性值将被设为 -1。

文件上传标准草案考虑到了从一个字段上传多个文件的可能性（使用递归的 *multipart/** 编码格式）。当这种情况发生时，该条目将是一个类似字典的 FieldStorage 条目。这可以通过检测它的 type 属性来确

定，该属性应当是 *multipart/form-data* (或者可能是匹配 *multipart/** 的其他 MIME 类型)。在这种情况下，它可以像最高层级的表单对象一样被递归地迭代处理。

当一个表单按“旧”格式提交时（即以查询字符串或是单个 *application/x-www-form-urlencoded* 类型的数据部分的形式），这些条目实际上将是 MiniFieldStorage 类的实例。在这种情况下，list, file 和 filename 属性将总是为 None。

通过 POST 方式提交并且也带有查询字符串的表单将同时包含 FieldStorage 和 MiniFieldStorage 条目。

3.4 版更變: file 属性会在创建 FieldStorage 实例的垃圾回收操作中被自动关闭。

3.5 版更變: 为 FieldStorage 类增加了上下文管理协议支持。

36.5.3 更高层级的接口

前面的部分解释了如何使用 FieldStorage 类来读取 CGI 表单数据。本部分则会描述一个更高层级的接口，它被添加到此类中以允许人们以更为可读和自然的方式行事。这个接口并不会完全取代前面的部分所描述的技巧 --- 例如它们在高效处理文件上传时仍然很有用处。

此接口由两个简单的方法组成。你可以使用这两个方法以通用的方式处理表单数据，而无需担心在一个名称下提交的值是只有一个还是有多个。

在前面的部分中，你已学会当你预期用户在一个名称下提交超过一个值的时候编写以下代码:

```
item = form.getvalue("item")
if isinstance(item, list):
    # The user is requesting more than one item.
else:
    # The user is requesting only one item.
```

这种情况很常见，例如当一个表单包含具有相同名称的一组复选框的时候:

```
<input type="checkbox" name="item" value="1" />
<input type="checkbox" name="item" value="2" />
```

但是在多数情况下，一个表单中的一个特定名称只对应一个表单控件。因此你可能会编写包含以下代码的脚本:

```
user = form.getvalue("user").upper()
```

这段代码的问题在于你绝不能预期客户端会向你的脚本提供合法的输入。举例来说，如果一个好奇的用户向查询字符串添加了另一个 user=foo 对，则该脚本将会崩溃，因为在这种情况下 getvalue("user") 方法调用将返回一个列表而不是字符串。在一个列表上调用 *upper()* 方法是不合法的（因为列表并没有这个方法）因而会引发 *AttributeError* 异常。

因此，读取表单数据值的正确方式应当总是使用检查所获取的值是单一值还是值列表的代码。这很麻烦并且会使脚本缺乏可读性。

一种更便捷的方式是使用这个更高层级接口所提供的 *getfirst()* 和 *getlist()* 方法。

FieldStorage.**getfirst**(*name*, *default=None*)

此方法总是只返回与表单字段 *name* 相关联的单一值。此方法在同一名称下提交了多个值的情况下将仅返回第一个值。请注意所接收的值顺序在不同浏览器上可能发生变化因而是不确定的。[1] 如果指定的表单字段或值不存在则此方法将返回可选形参 *default* 所指定的值。如果未指定此形参则默认值为 None。

[1] 请注意，新版的 HTML 规范确实注明了请求字段的顺序，但判断请求是否合法非常繁琐和容易出错，可能来自不符合要求的浏览器，甚至不是来自浏览器。

FieldStorage.**getlist**(*name*)
此方法总是返回与表单字段 *name* 相关联的值列表。如果 *name* 指定的表单字段或值不存在则此方法将返回一个空列表。如果指定的表单字段只包含一个值则它将返回只有一项的列表。

使用这两个方法你将能写出优雅简洁的代码:

```
import cgi
form = cgi.FieldStorage()
user = form.getfirst("user", "").upper()    # This way it's safe.
for item in form.getlist("item"):
    do_something(item)
```

36.5.4 函数

这些函数在你想要更多控制，或者如果你想要应用一些此模块中在其他场景下实现的算法时很有用处。

cgi.**parse**(*fp=None*, *environ=os.environ*, *keep_blank_values=False*, *strict_parsing=False*, *separator="&"*)
在环境中或从某个文件中解析一个查询 (文件默认为 sys.stdin)。*keep_blank_values*, *strict_parsing* 和 *separator* 形参会被原样传给*urllib.parse.parse_qs()*。

cgi.**parse_multipart**(*fp*, *pdict*, *encoding="utf-8"*, *errors="replace"*, *separator="&"*)
解析 *multipart/form-data* 类型（用于文件上传）的输入。参数中 *fp* 为输入文件，*pdict* 为包含 *Content-Type* 标头中的其他形参的字典，*encoding* 为请求的编码格式。

像*urllib.parse.parse_qs()* 那样返回一个字典：其中的键为字段名称，值为对应字段的值列表。对于非文件字段，其值均为字符串列表。

这很容易使用，但如果你预期要上传巨量字节数据时就不太适合了 --- 在这种情况下，请改用更为灵活的 FieldStorage 类。

3.7 版更變: 增加了 *encoding* 和 *errors* 形参。对于非文件字段，其值现在为字符串列表而非字节串列表。

3.9.2 版更變: 增加了 *separator* 形参。

cgi.**parse_header**(*string*)
将一个 MIME 标头 (例如 *Content-Type*) 解析为一个主值和一个参数字典。

cgi.**test**()
对 CGI 执行健壮性检测，适于作为主程序。写入最小化的 HTTP 标头并以 HTML 格式来格式化提供给脚本的所有信息。

cgi.**print_environ**()
以 HTML 格式来格式化 shell 环境。

cgi.**print_form**(*form*)
以 HTML 格式来格式化表单。

cgi.**print_directory**()
以 HTML 格式来格式化当前目录。

cgi.**print_environ_usage**()
以 HTML 格式打印有用的环境变量列表（供 CGI 使用）。

36.5.5 对于安全性的关注

有一条重要的规则：如果你发起调用一个外部程序（通过*os.system()*, *os.popen()* 或其他具有类似功能的函数），需要非常确定你不会把从客户端接收的任意字符串直接传给 shell。这是一个著名的安全漏洞，网络中聪明的黑客可以通过它来利用容易上当的 CGI 脚本发起调用任何 shell 命令。即使 URL 的一部分或字段名称也是不可信任的，因为请求并不一定是来自你的表单！

为了安全起见，如果你必须将从表单获取的字符串传给 shell 命令，你应当确保该字符串仅包含字母数字类字符、连字符、下划线和句点。

36.5.6 在 Unix 系统上安装你的 CGI 脚本

请阅读你的 HTTP 服务器的文档并咨询你所用系统的管理员来找到 CGI 脚本应当安装到哪个目录；通常是服务器目录树中的 `cgi-bin` 目录。

请确保你的脚本可被“其他人”读取和执行；Unix 文件模式应为八进制数 `0o755` (使用 `chmod 0755 filename`)。请确保脚本的第一行包含 `#!` 且位置是从第 1 列开始，后面带有 Python 解释器的路径名，例如:

```
#!/usr/local/bin/python
```

请确保该 Python 解释器存在并且可被“其他人”执行。

请确保你的脚本需要读取或写入的任何文件都分别是“其他人”可读取或可写入的 --- 它们的模式应为可读取 `0o644` 或可写入 `0o666`。这是因为出于安全理由，HTTP 服务器是作为没有任何特殊权限的”nobody” 用户来运行脚本的。它只能读取（写入、执行）任何人都能读取（写入、执行）的文件。执行时的当前目录（通常为服务器的 cgi-bin 目录）和环境变量集合也与你在登录时所得到的不同。特别地，不可依赖于 shell 的可执行文件搜索路径 (`PATH`) 或 Python 模块搜索路径 (`PYTHONPATH`) 的任何相关设置。

如果你需要从 Python 的默认模块搜索路径之外的目录载入模块，你可以在导入其他模块之前在你的脚本中改变路径。例如:

```
import sys
sys.path.insert(0, "/usr/home/joe/lib/python")
sys.path.insert(0, "/usr/local/lib/python")
```

（在此方式下，最后插入的目录将最先被搜索！）

针对非 Unix 系统的指导会有所变化；请查看你的 HTTP 服务器的文档（通常会有关于 CGI 脚本的部分）。

36.5.7 测试你的 CGI 脚本

很不幸，当你在命令行中尝试 CGI 脚本时它通常会无法运行，而能在命令行中完美运行的脚本则可能会在运行于服务器时神秘地失败。但有一个理由使你仍然应当在命令行中测试你的脚本：如果它包含语法错误，Python 解释器将根本不会执行它，而 HTTP 服务器将很可能向客户端发送令人费解的错误信息。

假定你的脚本没有语法错误，但它仍然无法起作用，你将别无选择，只能继续阅读下一节。

36.5.8 调试 CGI 脚本

首先，请检查是否有安装上的小错误 --- 仔细阅读上面关于安装 CGI 脚本的部分可以使你节省大量时间。如果你不确定你是否正确理解了安装过程，请尝试将此模块 (`cgi.py`) 的副本作为 CGI 脚本安装。当作为脚本被发起调用时，该文件将以 HTML 格式转储其环境和表单内容。请给它赋予正确的模式等，并向它发送一个请求。如果它是安装在标准的 `cgi-bin` 目录下，应该可以通过在你的浏览器中输入表单的 URL 来向它发送请求。

```
http://yourhostname/cgi-bin/cgi.py?name=Joe+Blow&addr=At+Home
```

如果此操作给出类型为 404 的错误，说明服务器找不到此脚本 -- 也许你需要将它安装到不同的目录。如果它给出另一种错误，说明存在安装问题，你应当解决此问题才能继续操作。如果你得到一个格式良好的环境和表单内容清单（在这个例子中，应当会列出的有字段"addr" 值为"At Home" 以及"name" 值为"Joe Blow"），则说明 `cgi.py` 脚本已正确安装。如果你为自己的脚本执行了同样的过程，现在你应该能够调试它了。

下一步骤可以是在你的脚本中调用*cgi* 模块的*test()* 函数：用这一条语句替换它的主代码

```
cgi.test()
```

这将产生从安装 `cgi.py` 文件本身所得到的相同结果。

当某个常规 Python 脚本触发了未处理的异常，(无论出于什么原因：模块名称出错、文件无法打开等)，Python 解释器就会打印出一条完整的跟踪信息并退出。在 CGI 脚本触发异常时，Python 解释器依然会如此，但最有可能的是，跟踪信息只会停留在某个 HTTP 服务日志文件中，或者被完全丢弃。

幸运的是，只要执行 某些代码，就可以利用*cgitb* 模块将跟踪信息发送给浏览器。如果你还没有这样做过，只需添加以下几行代码:

```
import cgitb
cgitb.enable()
```

然后再运行一下看；发生问题时应能看到详细的报告，或许能让崩溃的原因更清晰一些。

如果怀疑是*cgitb* 模块导入的问题，可以采用一个功能更强的方法（只用到内置模块）：

```
import sys
sys.stderr = sys.stdout
print("Content-Type: text/plain")
print()
...your code here...
```

这得靠 Python 解释器来打印跟踪信息。输出的类型为纯文本，不经过任何 HTML 处理。如果代码正常，则客户端会显示原有的 HTML。如果触发了异常，很可能在输出前两行后会显示一条跟踪信息。因为不会继续进行 HTML 解析，所以跟踪信息肯定能被读到。

36.5.9 常见问题和解决方案

- 大部分 HTTP 服务器会对 CGI 脚本的输出进行缓存，等脚本执行完毕再行输出。这意味着在脚本运行时，不可能在客户端屏幕上显示出进度情况。
- 请查看上述安装说明。
- 请查看 HTTP 服务器的日志文件。(在另一个单独窗口中执行 `tail -f logfile` 可能会很有用!)
- 一定要先检查脚本是否有语法错误，做法类似：`python script.py` 。
- 如果脚本没有语法错误，试着在脚本的顶部添加 `import cgitb; cgitb.enable()`。

- 当调用外部程序时，要确保其可被读取。通常这意味着采用绝对路径名------ 在 CGI 脚本中，PATH 通常不会设为很有用的值。
- 在读写外部文件时，要确保其能被 CGI 脚本归属的用户读写：通常是运行网络服务的用户，或由网络服务的 suexec 功能明确指定的一些用户。
- 不要试图给 CGI 脚本赋予 set-uid 模式。这在大多数系统上这都行不通，出于安全考虑也不应如此。

F解

36.6 cgitb --- 用于 CGI 脚本的回溯管理器

源代码: Lib/cgitb.py

3.11 版後已F用: The *cgitb* module is deprecated (see **PEP 594** for details).

---

cgitb 模块提供了用于 Python 脚本的特殊异常处理程序。(这个名称有一点误导性。它最初是设计用来显示 HTML 格式的 CGI 脚本详细回溯信息。但后来被一般化为也可显示纯文本格式的回溯信息。) 激活这个模块之后，如果发生了未被捕获的异常，将会显示详细的已格式化的报告。报告显示内容包括每个层级的源代码摘录，还有当前正在运行的函数的参数和局部变量值，以帮助你调试问题。你也可以选择将此信息保存至文件而不是将其发送至浏览器。

要启用此特性，只需简单地将此代码添加到你的 CGI 脚本的最顶端:

```
import cgitb
cgitb.enable()
```

enable() 函数的选项可以控制是将报告显示在浏览器中，还是将报告记录到文件供以后进行分析。

cgitb.**enable**(*display=1*, *logdir=None*, *context=5*, *format="html"*)
此函数可通过设置*sys.excepthook* 的值以使*cgitb* 模块接管解释器默认的异常处理机制。

可选参数 *display* 默认为 1 并可被设为 0 来停止将回溯发送至浏览器。如果给出了参数 *logdir*，则回溯会被写入文件。*logdir* 的值应当是一个用于存放所写入文件的目录。可选参数 *context* 是要在回溯中的当前源代码行前后显示的上下文行数；默认为 5。如果可选参数 *format* 为 "html"，输出将为 HTML 格式。任何其它值都会强制启用纯文本输出。默认取值为 "html"。

cgitb.**text**(*info*, *context=5*)
此函数用于处理 *info* (一个包含*sys.exc_info()* 返回结果的 3 元组) 所描述的异常，将其回溯格式化为文本并将结果作为字符串返回。可选参数 *context* 是要在回溯中的当前源码行前后显示的上下文行数；默认为 5。

cgitb.**html**(*info*, *context=5*)
此函数用于处理 *info* (一个包含*sys.exc_info()* 返回结果的 3 元组) 所描述的异常，将其回溯格式化为 HTML 并将结果作为字符串返回。可选参数 *context* 是要在回溯中的当前源码行前后显示的上下文行数；默认为 5。

cgitb.**handler**(*info=None*)
此函数使用默认设置处理异常（即在浏览器中显示报告，但不记录到文件）。当你捕获了一个异常并希望使用*cgitb* 来报告它时可以使用此函数。可选的 *info* 参数应为一个包含异常类型，异常值和回溯对象的 3 元组，与*sys.exc_info()* 所返回的元组完全一致。如果未提供 *info* 参数，则会从*sys.exc_info()* 获取当前异常。

36.7 chunk --- 读取 IFF 分块数据

源代码: Lib/chunk.py

3.11 版後已🄵用: The *chunk* module is deprecated (see **PEP 594** for details).

本模块提供了一个读取使用 EA IFF 85 分块的数据的接口 chunks.[1] 这种格式使用的场合有 Audio Interchange File Format (AIFF/AIFF-C) 和 Real Media File Format (RMFF) 等。与它们密切相关的 WAVE 音频文件也可使用此模块来读取。

一个分块具有以下结构:

偏移	长度	内容
0	4	区块 ID
4	4	大端字节顺序的块大小，不包括头
8	*n*	数据字节，其中 *n* 是前一字段中给出的大小
8 + *n*	0 或 1	如果 *n* 为奇数且使用块对齐，则需要填充字节

ID 是一个 4 字节的字符串，用于标识块的类型。

大小字段（32 位的值，使用大端字节序编码）给出分块数据的大小，不包括 8 字节的标头。

使用由一个或更多分块组成的 IFF 类型文件。此处定义的*Chunk* 类的建议使用方式是在每个分块开始时实例化一个实例并从实例读取直到其末尾，在那之后可以再实例化新的实例。到达文件末尾时，创建新实例将会失败并引发*EOFError* 异常。

class chunk.**Chunk**(*file*, *align=True*, *bigendian=True*, *inclheader=False*)

代表一个分块的类。*file* 参数预期为一个文件类对象。特别地也允许该类的实例。唯一必需的方法是 `read()`。如果存在*seek()* 和*tell()* 方法并且没有引发异常，它们也会被使用。如果存在这些方法并且引发了异常，则它们不应改变目标对象。如果可选参数 *align* 为真值，则分块应当以 2 字节边界对齐。如果 *align* 为假值，则不使用对齐。此参数默认为真值。如果可选参数 *bigendian* 为假值，分块大小应当为小端序。这对于 WAVE 音频文件是必须的。此参数默认为真值。如果可选参数 *inclheader* 为真值，则分块标头中给出的大小将包括标头的大小。此参数默认为假值。

Chunk 对象支持下列方法:

getname()

返回分块的名称（ID）。这是分块的头 4 个字节。

getsize()

返回分块的大小。

close()

关闭并跳转到分块的末尾。这不会关闭下层的文件。

在*close()* 方法已被调用后其余方法将会引发*OSError*。在 Python 3.3 之前，它们曾会引发*IOError*，现在这是*OSError* 的一个别名。

isatty()

返回 `False`。

seek(*pos*, *whence=0*)

设置分块的当前位置。*whence* 参数为可选项并且默认为 `0` (绝对文件定位)；其他值还有 `1` (相对当前位置查找) 和 `2` (相对文件末尾查找)。没有返回值。如果下层文件不支持查找，则只允许向前查找。

[1] "EA IFF 85" 交换格式文件标准, Jerry Morrison, Electronic Arts, 1985 年 1 月。

tell()
将当前位置返回到分块。

read(*size=-1*)
从分块读取至多 *size* 个字节（如果在获得 *size* 个字节之前已到达分块末尾则读取的字节会少于此数量）。如果 *size* 参数为负值或被省略，则读取所有字节直到分块末尾。当立即遇到分块末尾则返回空字节串对象。

skip()
跳到分块末尾。此后对分块再次调用 *read()* 将返回 `b''`。如果你对分块的内容不感兴趣，则应当调用此方法以使文件指向下一分块的开头。

𝔽解

36.8 crypt ——检查 Unix 口令的函数

源代码： Lib/struct.py

3.11 版後已𝔽用: The *crypt* module is deprecated (see **PEP 594** for details and alternatives). The *hashlib* module is a potential replacement for certain use cases.

本模块实现了连接 *crypt(3)* 的接口，是一个基于改进 DES 算法的单向散列函数；更多细节请参阅 Unix man 手册。可能的用途包括保存经过哈希的口令，这样就可以在不存储实际口令的情况下对其进行验证，或者尝试用字典来破解 Unix 口令。

请注意，本模块的执行取决于当前系统中 *crypt(3)* 的实际实现。因此，当前实现版本可用的扩展均可在本模块使用。

可用性: Unix。在 VxWorks 上不可用。

36.8.1 哈希方法

3.3 版新加入.

crypt 模块定义了哈希方法的列表（不是所有的方法在所有平台上都可用）。

crypt.**METHOD_SHA512**
基于 SHA-512 哈希函数的模块化加密格式方法，具备 16 个字符的 salt 和 86 个字符的哈希算法。这是最强的哈希算法。

crypt.**METHOD_SHA256**
另一种基于 SHA-256 哈希函数的模块化加密格式方法，具备 16 个字符的 salt 和 43 个字符的哈希算法。

crypt.**METHOD_BLOWFISH**
另一种基于 Blowfish 的模块化加密格式方法，有 22 个字符的 salt 和 31 个字符的哈希算法。

3.7 版新加入.

crypt.**METHOD_MD5**
另一种基于 MD5 哈希函数的模块化加密格式方法，具备 8 个字符的 salt 和 22 个字符的哈希算法。

crypt.**METHOD_CRYPT**
传统的方法，具备 2 个字符的 salt 和 13 个字符的哈希算法。这是最弱的方法。

36.8.2 模块属性

3.3 版新加入.

crypt.**methods**
可用口令哈希算法的列表，形式为 crypt.METHOD_* 对象。该列表从最强到最弱进行排序。

36.8.3 模块函数

crypt 模块定义了以下函数：

crypt.**crypt**(*word*, *salt=None*)
word will usually be a user's password as typed at a prompt or in a graphical interface. The optional *salt* is either a string as returned from *mksalt()*, one of the crypt.METHOD_* values (though not all may be available on all platforms), or a full encrypted password including salt, as returned by this function. If *salt* is not provided, the strongest method available in *methods* will be used.

查验口令通常是传入纯文本密码 *word* ，和之前*crypt()* 调用的结果进行比较，应该与本次调用的结果相同。

salt (随机的 2 或 16 个字符的字符串，可能带有 $digit{TX-PL-LABEL}#x60; 前缀以提示相关方法) 将被用来扰乱加密算法。*salt* 中的字符必须在 [./a-zA-Z0-9] 集合中，但 Modular Crypt Format 除外，它会带有 $digit{TX-PL-LABEL}#x60; 前缀。

返回哈希后的口令字符串，将由 salt 所在字母表中的字符组成。

由于有些 *crypt(3)* 扩展允许不同的值，*salt* 大小也可不同，建议在查验口令时采用完整的加密后口令作为 salt。

3.3 版更變: 除了字符串之外，*salt* 还可接受 crypt.METHOD_* 值。

crypt.**mksalt**(*method=None*, *, *rounds=None*)
Return a randomly generated salt of the specified method. If no *method* is given, the strongest method available in *methods* is used.

返回一个字符串，可用作传入*crypt()* 的 *salt* 参数。

rounds 指定了 METHOD_SHA256, METHOD_SHA512 和 METHOD_BLOWFISH 的循环次数。对于 METHOD_SHA256 和 METHOD_SHA512 而言，必须为介于 1000 和 999_999_999 之间的整数，默认值为 5000。而对于 METHOD_BLOWFISH，则必须为 16 (2^4) 和 2_147_483_648 (2^{31}) 之间的二的幂，默认值为 4096 (2^{12})。

3.3 版新加入.

3.7 版更變: 加入 *rounds* 参数。

36.8.4 示例

以下简单示例演示了典型用法（需要一个时间固定的比较操作来限制留给计时攻击的暴露面。*hmac.compare_digest()* 即很适用）：

```
import pwd
import crypt
import getpass
from hmac import compare_digest as compare_hash

def login():
    username = input('Python login: ')
```

(下页继续)

(繼續上一頁)

```
    cryptedpasswd = pwd.getpwnam(username)[1]
    if cryptedpasswd:
        if cryptedpasswd == 'x' or cryptedpasswd == '*':
            raise ValueError('no support for shadow passwords')
        cleartext = getpass.getpass()
        return compare_hash(crypt.crypt(cleartext, cryptedpasswd), cryptedpasswd)
    else:
        return True
```

采用当前强度最高的方法生成哈希值，并与原口令进行核对：

```
import crypt
from hmac import compare_digest as compare_hash

hashed = crypt.crypt(plaintext)
if not compare_hash(hashed, crypt.crypt(plaintext, hashed)):
    raise ValueError("hashed version doesn't validate against original")
```

36.9 imghdr --- 推测图像类型

源代码 Lib/imghdr.py

3.11 版後已🄴用: The *imghdr* module is deprecated (see **PEP 594** for details and alternatives).

---

imghdr 模块推测文件或字节流中的图像的类型。

imghdr 模块定义了以下类型：

imghdr.**what**(*filename*, *h=None*)

测试包含在命名为 *filename* 的文件中的图像数据，并且返回描述此类图片的字符串。如果可选的 *h* 被提供，*filename* 将被忽略并且 *h* 包含将被测试的二进制流。

3.6 版更變: 接受一个类路径对象。

接下来的图像类型是可识别的，返回值来自 *what()*：

值	图像格式
'rgb'	SGI 图像库文件
'gif'	GIF 87a 和 89a 文件
'pbm'	便携式位图文件
'pgm'	便携式灰度图文件
'ppm'	便携式像素表文件
'tiff'	TIFF 文件
'rast'	Sun 光栅文件
'xbm'	X 位图文件
'jpeg'	JFIF 或 Exif 格式的 JPEG 数据
'bmp'	BMP 文件
'png'	便携式网络图像
'webp'	WebP 文件
'exr'	OpenEXR 文件

3.5 版新加入: *exr* 和 *webp* 格式被添加。

你可以扩展此*imghdr* 可以被追加的这个变量识别的文件格式的列表：

imghdr.**tests**
执行单个测试的函数列表。每个函数都有两个参数：字节流和类似开放文件的对象。当*what()* 用字节流调用时，类文件对象将是 None。

如果测试成功，这个测试函数应当返回一个描述图像类型的字符串，否则返回 None。

示例：

```
>>> import imghdr
>>> imghdr.what('bass.gif')
'gif'
```

36.10 imp ——由代码内部访问 import 。

源代码： Lib/imp.py

3.4 版後已🄵用：*imp* 模块已被废弃，请改用*importlib*。

本模块提供了一个接口，用于实现 import 语句的机制。这里定义了以下常量和函数：

imp.**get_magic**()
Return the magic string value used to recognize byte-compiled code files (.pyc files). (This value may be different for each Python version.)

3.4 版後已🄵用: Use *importlib.util.MAGIC_NUMBER* instead.

imp.**get_suffixes**()
Return a list of 3-element tuples, each describing a particular type of module. Each triple has the form (suffix, mode, type), where *suffix* is a string to be appended to the module name to form the filename to search for, *mode* is the mode string to pass to the built-in *open()* function to open the file (this can be 'r' for text files or 'rb' for binary files), and *type* is the file type, which has one of the values *PY_SOURCE*, *PY_COMPILED*, or *C_EXTENSION*, described below.

3.3 版後已🄵用: Use the constants defined on *importlib.machinery* instead.

imp.**find_module**(*name*[, *path*])
Try to find the module *name*. If *path* is omitted or None, the list of directory names given by sys.path is searched, but first a few special places are searched: the function tries to find a built-in module with the given name (*C_BUILTIN*), then a frozen module (*PY_FROZEN*), and on some systems some other places are looked in as well (on Windows, it looks in the registry which may point to a specific file).

Otherwise, *path* must be a list of directory names; each directory is searched for files with any of the suffixes returned by *get_suffixes()* above. Invalid names in the list are silently ignored (but all list items must be strings).

If search is successful, the return value is a 3-element tuple (file, pathname, description):

file is an open *file object* positioned at the beginning, *pathname* is the pathname of the file found, and *description* is a 3-element tuple as contained in the list returned by *get_suffixes()* describing the kind of module found.

If the module is built-in or frozen then *file* and *pathname* are both None and the *description* tuple contains empty strings for its suffix and mode; the module type is indicated as given in parentheses above. If the search is unsuccessful, *ImportError* is raised. Other exceptions indicate problems with the arguments or environment.

If the module is a package, *file* is None, *pathname* is the package path and the last item in the *description* tuple is *PKG_DIRECTORY*.

This function does not handle hierarchical module names (names containing dots). In order to find *P.M*, that is, submodule *M* of package *P*, use *find_module()* and *load_module()* to find and load package *P*, and then use *find_module()* with the *path* argument set to `P.__path__`. When *P* itself has a dotted name, apply this recipe recursively.

3.3 版後已F用: Use *importlib.util.find_spec()* instead unless Python 3.3 compatibility is required, in which case use *importlib.find_loader()*. For example usage of the former case, see the 示例 section of the *importlib* documentation.

`imp.`**`load_module`**(*name*, *file*, *pathname*, *description*)

Load a module that was previously found by *find_module()* (or by an otherwise conducted search yielding compatible results). This function does more than importing the module: if the module was already imported, it will reload the module! The *name* argument indicates the full module name (including the package name, if this is a submodule of a package). The *file* argument is an open file, and *pathname* is the corresponding file name; these can be `None` and `''`, respectively, when the module is a package or not being loaded from a file. The *description* argument is a tuple, as would be returned by *get_suffixes()*, describing what kind of module must be loaded.

If the load is successful, the return value is the module object; otherwise, an exception (usually *ImportError*) is raised.

Important: the caller is responsible for closing the *file* argument, if it was not `None`, even when an exception is raised. This is best done using a `try ... finally` statement.

3.3 版後已F用: If previously used in conjunction with *imp.find_module()* then consider using *importlib.import_module()*, otherwise use the loader returned by the replacement you chose for *imp.find_module()*. If you called *imp.load_module()* and related functions directly with file path arguments then use a combination of *importlib.util.spec_from_file_location()* and *importlib.util.module_from_spec()*. See the 示例 section of the *importlib* documentation for details of the various approaches.

`imp.`**`new_module`**(*name*)

Return a new empty module object called *name*. This object is *not* inserted in `sys.modules`.

3.4 版後已F用: Use *importlib.util.module_from_spec()* instead.

`imp.`**`reload`**(*module*)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the *module* argument).

When `reload(module)` is executed:

- Python modules' code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module's dictionary. The `init` function of extension modules is not called a second time.
- 与 Python 中的所有的其它对象一样，旧的对象只有在引用计数为 0 之后才会被回收。
- 模块命名空间中的名称重新指向任何新的或更改后的对象。
- 其他旧对象的引用（例如那个模块的外部名称）不会被重新绑定到引用的新对象的，并且如果有需要，必须在出现的每个命名空间中进行更新。

有一些其他注意事项：

当一个模块被重新加载的时候，它的字典（包含了那个模块的全区变量）会被保留。名称的重新定义会覆盖旧的定义，所以通常来说这不是问题。如果一个新模块没有定义在旧版本模块中定义的名称，则将保留旧版本中的定义。这一特性可用于作为那个模块的优点，如果它维护一个全局表或者对象的缓存——使用 `try` 语句，就可以测试表的存在并且跳过它的初始化，如果有需要的话：

```
try:
    cache
except NameError:
    cache = {}
```

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for *sys*, *__main__* and *builtins*. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using `from` ... `import` ..., calling *reload()* for the other module does not redefine the objects imported from it --- one way around this is to re-execute the `from` statement, another is to use `import` and qualified names (*module*.*name*) instead.

如果一个模块创建一个类的实例，重新加载定义那个类的模块不影响这个实例的方法定义———它们继续使用旧类的定义。对于子类，也是一样的。

3.3 版更變: Relies on both `__name__` and `__loader__` being defined on the module being reloaded instead of just `__name__`.

3.4 版後已🄵用: Use *importlib.reload()* instead.

The following functions are conveniences for handling **PEP 3147** byte-compiled file paths.

3.2 版新加入.

`imp.`**`cache_from_source`**(*path*, *debug_override=None*)

Return the **PEP 3147** path to the byte-compiled file associated with the source *path*. For example, if *path* is `/foo/bar/baz.py` the return value would be `/foo/bar/__pycache__/baz.cpython-32.pyc` for Python 3.2. The `cpython-32` string comes from the current magic tag (see *get_tag()*; if `sys.implementation.cache_tag` is not defined then *NotImplementedError* will be raised). By passing in `True` or `False` for *debug_override* you can override the system's value for `__debug__`, leading to optimized bytecode.

path need not exist.

3.3 版更變: If `sys.implementation.cache_tag` is `None`, then *NotImplementedError* is raised.

3.4 版後已🄵用: Use *importlib.util.cache_from_source()* instead.

3.5 版更變: The *debug_override* parameter no longer creates a `.pyo` file.

`imp.`**`source_from_cache`**(*path*)

Given the *path* to a **PEP 3147** file name, return the associated source code file path. For example, if *path* is `/foo/bar/__pycache__/baz.cpython-32.pyc` the returned path would be `/foo/bar/baz.py`. *path* need not exist, however if it does not conform to **PEP 3147** format, a *ValueError* is raised. If `sys.implementation.cache_tag` is not defined, *NotImplementedError* is raised.

3.3 版更變: Raise *NotImplementedError* when `sys.implementation.cache_tag` is not defined.

3.4 版後已🄵用: Use *importlib.util.source_from_cache()* instead.

`imp.`**`get_tag`**()

Return the **PEP 3147** magic tag string matching this version of Python's magic number, as returned by *get_magic()*.

3.4 版後已🄵用: Use `sys.implementation.cache_tag` directly starting in Python 3.3.

The following functions help interact with the import system's internal locking mechanism. Locking semantics of imports are an implementation detail which may vary from release to release. However, Python ensures that circular imports work without any deadlocks.

imp.**lock_held**()
Return `True` if the global import lock is currently held, else `False`. On platforms without threads, always return `False`.

On platforms with threads, a thread executing an import first holds a global import lock, then sets up a per-module lock for the rest of the import. This blocks other threads from importing the same module until the original import completes, preventing other threads from seeing incomplete module objects constructed by the original thread. An exception is made for circular imports, which by construction have to expose an incomplete module object at some point.

3.3 版更變: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

3.4 版後已F用.

imp.**acquire_lock**()
Acquire the interpreter's global import lock for the current thread. This lock should be used by import hooks to ensure thread-safety when importing modules.

Once a thread has acquired the import lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

On platforms without threads, this function does nothing.

3.3 版更變: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

3.4 版後已F用.

imp.**release_lock**()
Release the interpreter's global import lock. On platforms without threads, this function does nothing.

3.3 版更變: The locking scheme has changed to per-module locks for the most part. A global import lock is kept for some critical tasks, such as initializing the per-module locks.

3.4 版後已F用.

The following constants with integer values, defined in this module, are used to indicate the search result of *find_module()*.

imp.**PY_SOURCE**
The module was found as a source file.

3.3 版後已F用.

imp.**PY_COMPILED**
The module was found as a compiled code object file.

3.3 版後已F用.

imp.**C_EXTENSION**
The module was found as dynamically loadable shared library.

3.3 版後已F用.

imp.**PKG_DIRECTORY**
The module was found as a package directory.

3.3 版後已F用.

imp.**C_BUILTIN**
The module was found as a built-in module.

3.3 版後已F用.

imp.**PY_FROZEN**
The module was found as a frozen module.

3.3 版後已𝔽用.

class imp.**NullImporter**(*path_string*)
The *NullImporter* type is a **PEP 302** import hook that handles non-directory path strings by failing to find any modules. Calling this type with an existing directory or empty string raises *ImportError*. Otherwise, a *NullImporter* instance is returned.

Instances have only one method:

find_module(*fullname*[, *path*])
This method always returns None, indicating that the requested module could not be found.

3.3 版更變: None is inserted into sys.path_importer_cache instead of an instance of *NullImporter*.

3.4 版後已𝔽用: Insert None into sys.path_importer_cache instead.

36.10.1 示例

The following function emulates what was the standard import statement up to Python 1.4 (no hierarchical module names). (This *implementation* wouldn't work in that version, since *find_module()* has been extended and *load_module()* has been added in 1.4.)

```
import imp
import sys

def __import__(name, globals=None, locals=None, fromlist=None):
    # Fast path: see if the module has already been imported.
    try:
        return sys.modules[name]
    except KeyError:
        pass

    # If any of the following calls raises an exception,
    # there's a problem we can't handle -- let the caller handle it.

    fp, pathname, description = imp.find_module(name)

    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        # Since we may exit via an exception, close fp explicitly.
        if fp:
            fp.close()
```

36.11 mailcap --- Mailcap 文件处理

源代码: Lib/mailcap.py

3.11 版後已🄵用: The *mailcap* module is deprecated (see **PEP 594** for details). The *mimetypes* module provides an alternative.

Mailcap 文件可用来配置支持 MIME 的应用例如邮件阅读器和 Web 浏览器如何响应具有不同 MIME 类型的文件。("mailcap" 这个名称源自短语"mail capability"。) 例如，一个 mailcap 文件可能包含 `video/mpeg; xmpeg %s` 这样的行。然后，如果用户遇到 MIME 类型为 *video/mpeg* 的邮件消息或 Web 文档时，`%s` 将被替换为一个文件名（通常是一个临时文件）并且将自动启动 **xmpeg** 程序来查看该文件。

mailcap 格式的文档见 **RFC 1524**, "A User Agent Configuration Mechanism For Multimedia Mail Format Information"，但它并不是一个因特网标准。不过，mailcap 文件在大多数 Unix 系统上都受到支持。

`mailcap.`**`findmatch`**(*caps*, *MIMEtype*, *key='view'*, *filename='/dev/null'*, *plist=[]*)

返回一个 2 元组；其中第一个元素是包含所要执行命令的字符串 (它可被传递给*os.system()*)，第二个元素是对应于给定 MIME 类型的 mailcap 条目。如果找不到匹配的 MIME 类型，则将返回 `(None, None)`。

key 是所需字段的名称，它代表要执行的活动类型；默认值是'view'，因为在最通常的情况下你只是想要查看 MIME 类型数据的正文。其他可能的值还有'compose' 和'edit'，分别用于想要创建给定 MIME 类型正文或修改现有正文数据的情况。请参阅 **RFC 1524** 获取这些字段的完整列表。

filename 是在命令行中用来替换 `%s` 的文件名；默认值 `'/dev/null'` 几乎肯定不是你想要的，因此通常你要通过指定一个文件名来重载它。

plist 可以是一个包含命名形参的列表；默认值只是一个空列表。列表中的每个条目必须为包含形参名称的字符串、等于号 (`'='`) 以及形参的值。Mailcap 条目可以包含形如 `%{foo}` 的命名形参，它将由名为'foo' 的形参的值所替换。例如，如果命令行 `showpartial %{id} %{number} %{total}` 是在一个 mailcap 文件中，并且 *plist* 被设为 `['id=1', 'number=2', 'total=3']`，则结果命令行将为 `'showpartial 1 2 3'`。

在 mailcap 文件中，可以指定可选的"test" 字段来检测某些外部条件（例如所使用的机器架构或窗口系统）来确定是否要应用 mailcap 行。*findmatch()* 将自动检查此类条件并在检查未通过时跳过条目。

`mailcap.`**`getcaps`**()

返回一个将 MIME 类型映射到 mailcap 文件条目列表的字典。此字典必须被传给*findmatch()* 函数。条目会被存储为字典列表，但并不需要了解此表示形式的细节。

此信息来自在系统中找到的所有 mailcap 文件。用户的 mailcap 文件 `$HOME/.mailcap` 中的设置将覆盖系统 mailcap 文件 `/etc/mailcap`, `/usr/etc/mailcap` 和 `/usr/local/etc/mailcap` 中的设置。

一个用法示例:

```
>>> import mailcap
>>> d = mailcap.getcaps()
>>> mailcap.findmatch(d, 'video/mpeg', filename='tmp1223')
('xmpeg tmp1223', {'view': 'xmpeg %s'})
```

36.12 msilib --- 读写 Microsoft Installer 文件

源代码: Lib/msilib/__init__.py

3.11 版後已🄵用: The *msilib* module is deprecated (see **PEP 594** for details).

The *msilib* supports the creation of Microsoft Installer (`.msi`) files. Because these files often contain an embedded "cabinet" file (`.cab`), it also exposes an API to create CAB files. Support for reading `.cab` files is currently not implemented; read support for the `.msi` database is possible.

This package aims to provide complete access to all tables in an `.msi` file, therefore, it is a fairly low-level API. Two primary applications of this package are the *distutils* command `bdist_msi`, and the creation of Python installer package itself (although that currently uses a different version of `msilib`).

The package contents can be roughly split into four parts: low-level CAB routines, low-level MSI routines, higher-level MSI routines, and standard table structures.

`msilib.`**`FCICreate`**(*cabname*, *files*)
Create a new CAB file named *cabname*. *files* must be a list of tuples, each containing the name of the file on disk, and the name of the file inside the CAB file.

The files are added to the CAB file in the order they appear in the list. All files are added into a single CAB file, using the MSZIP compression algorithm.

Callbacks to Python for the various steps of MSI creation are currently not exposed.

`msilib.`**`UuidCreate`**()
Return the string representation of a new unique identifier. This wraps the Windows API functions `UuidCreate()` and `UuidToString()`.

`msilib.`**`OpenDatabase`**(*path*, *persist*)
Return a new database object by calling MsiOpenDatabase. *path* is the file name of the MSI file; *persist* can be one of the constants `MSIDBOPEN_CREATEDIRECT`, `MSIDBOPEN_CREATE`, `MSIDBOPEN_DIRECT`, `MSIDBOPEN_READONLY`, or `MSIDBOPEN_TRANSACT`, and may include the flag `MSIDBOPEN_PATCHFILE`. See the Microsoft documentation for the meaning of these flags; depending on the flags, an existing database is opened, or a new one created.

`msilib.`**`CreateRecord`**(*count*)
Return a new record object by calling `MSICreateRecord()`. *count* is the number of fields of the record.

`msilib.`**`init_database`**(*name*, *schema*, *ProductName*, *ProductCode*, *ProductVersion*, *Manufacturer*)
Create and return a new database *name*, initialize it with *schema*, and set the properties *ProductName*, *ProductCode*, *ProductVersion*, and *Manufacturer*.

schema must be a module object containing `tables` and `_Validation_records` attributes; typically, *msilib.schema* should be used.

The database will contain just the schema and the validation records when this function returns.

`msilib.`**`add_data`**(*database*, *table*, *records*)
Add all *records* to the table named *table* in *database*.

The *table* argument must be one of the predefined tables in the MSI schema, e.g. `'Feature'`, `'File'`, `'Component'`, `'Dialog'`, `'Control'`, etc.

records should be a list of tuples, each one containing all fields of a record according to the schema of the table. For optional fields, `None` can be passed.

Field values can be ints, strings, or instances of the Binary class.

class msilib.**Binary**(*filename*)
Represents entries in the Binary table; inserting such an object using *add_data()* reads the file named *filename* into the table.

msilib.**add_tables**(*database*, *module*)
Add all table content from *module* to *database*. *module* must contain an attribute *tables* listing all tables for which content should be added, and one attribute per table that has the actual content.

This is typically used to install the sequence tables.

msilib.**add_stream**(*database*, *name*, *path*)
Add the file *path* into the `_Stream` table of *database*, with the stream name *name*.

msilib.**gen_uuid**()
Return a new UUID, in the format that MSI typically requires (i.e. in curly braces, and with all hexdigits in uppercase).

也参考:

FCICreate UuidCreate UuidToString

36.12.1 Database Objects

Database.**OpenView**(*sql*)
Return a view object, by calling `MSIDatabaseOpenView()`. *sql* is the SQL statement to execute.

Database.**Commit**()
Commit the changes pending in the current transaction, by calling `MSIDatabaseCommit()`.

Database.**GetSummaryInformation**(*count*)
Return a new summary information object, by calling `MsiGetSummaryInformation()`. *count* is the maximum number of updated values.

Database.**Close**()
Close the database object, through `MsiCloseHandle()`.

3.7 版新加入.

也参考:

MSIDatabaseOpenView MSIDatabaseCommit MSIGetSummaryInformation MsiCloseHandle

36.12.2 View Objects

View.**Execute**(*params*)
Execute the SQL query of the view, through `MSIViewExecute()`. If *params* is not `None`, it is a record describing actual values of the parameter tokens in the query.

View.**GetColumnInfo**(*kind*)
Return a record describing the columns of the view, through calling `MsiViewGetColumnInfo()`. *kind* can be either `MSICOLINFO_NAMES` or `MSICOLINFO_TYPES`.

View.**Fetch**()
Return a result record of the query, through calling `MsiViewFetch()`.

View.**Modify**(*kind*, *data*)
Modify the view, by calling `MsiViewModify()`. *kind* can be one of `MSIMODIFY_SEEK`, `MSIMODIFY_REFRESH`, `MSIMODIFY_INSERT`, `MSIMODIFY_UPDATE`, `MSIMODIFY_ASSIGN`, `MSIMODIFY_REPLACE`, `MSIMODIFY_MERGE`, `MSIMODIFY_DELETE`,

MSIMODIFY_INSERT_TEMPORARY, MSIMODIFY_VALIDATE, MSIMODIFY_VALIDATE_NEW, MSIMODIFY_VALIDATE_FIELD, or MSIMODIFY_VALIDATE_DELETE.

data must be a record describing the new data.

View.**Close**()
Close the view, through MsiViewClose().

也參考:

MsiViewExecute MSIViewGetColumnInfo MsiViewFetch MsiViewModify MsiViewClose

36.12.3 Summary Information Objects

SummaryInformation.**GetProperty**(*field*)
Return a property of the summary, through MsiSummaryInfoGetProperty(). *field* is the name of the property, and can be one of the constants PID_CODEPAGE, PID_TITLE, PID_SUBJECT, PID_AUTHOR, PID_KEYWORDS, PID_COMMENTS, PID_TEMPLATE, PID_LASTAUTHOR, PID_REVNUMBER, PID_LASTPRINTED, PID_CREATE_DTM, PID_LASTSAVE_DTM, PID_PAGECOUNT, PID_WORDCOUNT, PID_CHARCOUNT, PID_APPNAME, or PID_SECURITY.

SummaryInformation.**GetPropertyCount**()
Return the number of summary properties, through MsiSummaryInfoGetPropertyCount().

SummaryInformation.**SetProperty**(*field*, *value*)
Set a property through MsiSummaryInfoSetProperty(). *field* can have the same values as in *GetProperty()*, *value* is the new value of the property. Possible value types are integer and string.

SummaryInformation.**Persist**()
Write the modified properties to the summary information stream, using MsiSummaryInfoPersist().

也參考:

MsiSummaryInfoGetProperty MsiSummaryInfoGetPropertyCount MsiSummaryInfoSetProperty MsiSummaryInfoPersist

36.12.4 Record Objects

Record.**GetFieldCount**()
Return the number of fields of the record, through MsiRecordGetFieldCount().

Record.**GetInteger**(*field*)
Return the value of *field* as an integer where possible. *field* must be an integer.

Record.**GetString**(*field*)
Return the value of *field* as a string where possible. *field* must be an integer.

Record.**SetString**(*field*, *value*)
Set *field* to *value* through MsiRecordSetString(). *field* must be an integer; *value* a string.

Record.**SetStream**(*field*, *value*)
Set *field* to the contents of the file named *value*, through MsiRecordSetStream(). *field* must be an integer; *value* a string.

Record.**SetInteger**(*field*, *value*)
Set *field* to *value* through MsiRecordSetInteger(). Both *field* and *value* must be an integer.

Record.**ClearData**()
Set all fields of the record to 0, through MsiRecordClearData().

也參考:

MsiRecordGetFieldCount MsiRecordSetString MsiRecordSetStream MsiRecordSetInteger MsiRecordClearData

36.12.5 Errors

All wrappers around MSI functions raise `MSIError`; the string inside the exception will contain more detail.

36.12.6 CAB Objects

class msilib.**CAB**(*name*)

The class *CAB* represents a CAB file. During MSI construction, files will be added simultaneously to the `Files` table, and to a CAB file. Then, when all files have been added, the CAB file can be written, then added to the MSI file.

name is the name of the CAB file in the MSI file.

append(*full*, *file*, *logical*)

Add the file with the pathname *full* to the CAB file, under the name *logical*. If there is already a file named *logical*, a new file name is created.

Return the index of the file in the CAB file, and the new name of the file inside the CAB file.

commit(*database*)

Generate a CAB file, add it as a stream to the MSI file, put it into the `Media` table, and remove the generated file from the disk.

36.12.7 Directory Objects

class msilib.**Directory**(*database*, *cab*, *basedir*, *physical*, *logical*, *default*[, *componentflags*])

Create a new directory in the Directory table. There is a current component at each point in time for the directory, which is either explicitly created through *start_component()*, or implicitly when files are added for the first time. Files are added into the current component, and into the cab file. To create a directory, a base directory object needs to be specified (can be `None`), the path to the physical directory, and a logical directory name. *default* specifies the DefaultDir slot in the directory table. *componentflags* specifies the default flags that new components get.

start_component(*component=None*, *feature=None*, *flags=None*, *keyfile=None*, *uuid=None*)

Add an entry to the Component table, and make this component the current component for this directory. If no component name is given, the directory name is used. If no *feature* is given, the current feature is used. If no *flags* are given, the directory's default flags are used. If no *keyfile* is given, the KeyPath is left null in the Component table.

add_file(*file*, *src=None*, *version=None*, *language=None*)

Add a file to the current component of the directory, starting a new one if there is no current component. By default, the file name in the source and the file table will be identical. If the *src* file is specified, it is interpreted relative to the current directory. Optionally, a *version* and a *language* can be specified for the entry in the File table.

glob(*pattern*, *exclude=None*)

Add a list of files to the current component as specified in the glob pattern. Individual files can be excluded in the *exclude* list.

remove_pyc()

Remove `.pyc` files on uninstall.

也參考:

Directory Table File Table Component Table FeatureComponents Table

36.12.8 相关特性

class msilib.**Feature**(*db*, *id*, *title*, *desc*, *display*, *level=1*, *parent=None*, *directory=None*, *attributes=0*)
Add a new record to the `Feature` table, using the values *id*, *parent.id*, *title*, *desc*, *display*, *level*, *directory*, and *attributes*. The resulting feature object can be passed to the `start_component()` method of *Directory*.

set_current()
Make this feature the current feature of *msilib*. New components are automatically added to the default feature, unless a feature is explicitly specified.

也參考:

Feature Table

36.12.9 GUI classes

msilib provides several classes that wrap the GUI tables in an MSI database. However, no standard user interface is provided; use `bdist_msi` to create MSI files with a user-interface for installing Python packages.

class msilib.**Control**(*dlg*, *name*)
Base class of the dialog controls. *dlg* is the dialog object the control belongs to, and *name* is the control's name.

event(*event*, *argument*, *condition=1*, *ordering=None*)
Make an entry into the `ControlEvent` table for this control.

mapping(*event*, *attribute*)
Make an entry into the `EventMapping` table for this control.

condition(*action*, *condition*)
Make an entry into the `ControlCondition` table for this control.

class msilib.**RadioButtonGroup**(*dlg*, *name*, *property*)
Create a radio button control named *name*. *property* is the installer property that gets set when a radio button is selected.

add(*name*, *x*, *y*, *width*, *height*, *text*, *value=None*)
Add a radio button named *name* to the group, at the coordinates *x*, *y*, *width*, *height*, and with the label *text*. If *value* is `None`, it defaults to *name*.

class msilib.**Dialog**(*db*, *name*, *x*, *y*, *w*, *h*, *attr*, *title*, *first*, *default*, *cancel*)
Return a new *Dialog* object. An entry in the `Dialog` table is made, with the specified coordinates, dialog attributes, title, name of the first, default, and cancel controls.

control(*name*, *type*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *control_next*, *help*)
Return a new *Control* object. An entry in the `Control` table is made with the specified parameters.

This is a generic method; for specific types, specialized methods are provided.

text(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*)
Add and return a `Text` control.

bitmap(*name*, *x*, *y*, *width*, *height*, *text*)
Add and return a `Bitmap` control.

line(*name*, *x*, *y*, *width*, *height*)
Add and return a `Line` control.

pushbutton(*name*, *x*, *y*, *width*, *height*, *attributes*, *text*, *next_control*)
Add and return a `PushButton` control.

radiogroup(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `RadioButtonGroup` control.

checkbox(*name*, *x*, *y*, *width*, *height*, *attributes*, *property*, *text*, *next_control*)
Add and return a `CheckBox` control.

也参考:

Dialog Table Control Table Control Types ControlCondition Table ControlEvent Table EventMapping Table RadioButton Table

36.12.10 Precomputed tables

msilib provides a few subpackages that contain only schema and table definitions. Currently, these definitions are based on MSI version 2.0.

msilib.**schema**
This is the standard MSI schema for MSI 2.0, with the *tables* variable providing a list of table definitions, and *_Validation_records* providing the data for MSI validation.

msilib.**sequence**
This module contains table contents for the standard sequence tables: *AdminExecuteSequence*, *AdminUISequence*, *AdvtExecuteSequence*, *InstallExecuteSequence*, and *InstallUISequence*.

msilib.**text**
This module contains definitions for the UIText and ActionText tables, for the standard installer actions.

36.13 `nis` --- Sun 的 NIS (黄页) 接口

3.11 版後已F用: The *nis* module is deprecated (see **PEP 594** for details).

---

nis 模块提供了对 NIS 库的轻量级包装，适用于多个主机的集中管理。

因为 NIS 仅存在于 Unix 系统，此模块仅在 Unix 上可用。

nis 模块定义了以下函数:

nis.**match**(*key*, *mapname*, *domain=default_domain*)
返回 *key* 在映射 *mapname* 中的匹配结果，如无结果则会引发错误 (*nis.error*)。两个参数都应为字符串，*key* 定长 8 个比特。返回值为任意字节数组（可包含 `NULL` 和其他特殊值）。

请注意如果 *mapname* 是另一名称的别名则会先检查别名。

domain 参数可允许重载用于查找的 NIS 域。如果未指定，则会在默认 NIS 域中查找。

nis.**cat**(*mapname*, *domain=default_domain*)
返回一个字典，其元素为 *key* 到 *value* 的映射使得 `match(key, mapname)==value`。请注意字典的键和值均为任意字节数组。

请注意如果 *mapname* 是另一名称的别名则会先检查别名。

domain 参数可允许重载用于查找的 NIS 域。如果未指定，则会在默认 NIS 域中查找。

nis.**maps**(*domain=default_domain*)
返回全部可用映射的列表。

domain 参数可允许重载用于查找的 NIS 域。如果未指定，则会在默认 NIS 域中查找。

nis.**get_default_domain**()
返回系统默认的 NIS 域。

nis 模块定义了以下异常：

exception nis.**error**
当 NIS 函数返回一个错误码时引发的异常。

36.14 nntplib --- NNTP 协议客户端

源代码: Lib/nntplib.py

3.11 版後已🄴用: The *nntplib* module is deprecated (see **PEP 594** for details).

此模块定义了*NNTP* 类来实现网络新闻传输协议的客户端。它可被用于实现一个新闻阅读或发布器，或是新闻自动处理程序。它兼容了 **RFC 3977** 以及较旧的 **RFC 977** 和 **RFC 2980**。

下面是此模块的两个简单用法示例。列出某个新闻组的一些统计数据并打印最近 10 篇文章的主题：

```
>>> s = nntplib.NNTP('news.gmane.io')
>>> resp, count, first, last, name = s.group('gmane.comp.python.committers')
>>> print('Group', name, 'has', count, 'articles, range', first, 'to', last)
Group gmane.comp.python.committers has 1096 articles, range 1 to 1096
>>> resp, overviews = s.over((last - 9, last))
>>> for id, over in overviews:
...     print(id, nntplib.decode_header(over['subject']))
...
1087 Re: Commit privileges for Łukasz Langa
1088 Re: 3.2 alpha 2 freeze
1089 Re: 3.2 alpha 2 freeze
1090 Re: Commit privileges for Łukasz Langa
1091 Re: Commit privileges for Łukasz Langa
1092 Updated ssh key
1093 Re: Updated ssh key
1094 Re: Updated ssh key
1095 Hello fellow committers!
1096 Re: Hello fellow committers!
>>> s.quit()
'205 Bye!'
```

要基于一个二进制文件发布文章 (假定文章包含有效的标头，并且你有在特定新闻组上发布内容的权限)：

```
>>> s = nntplib.NNTP('news.gmane.io')
>>> f = open('article.txt', 'rb')
>>> s.post(f)
'240 Article posted successfully.'
>>> s.quit()
'205 Bye!'
```

此模块本身定义了以下的类：

class nntplib.**NNTP**(*host*, *port=119*, *user=None*, *password=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

返回一个新的*NNTP*对象，代表一个对运行于主机 *host*，在端口 *port* 上监听的 NNTP 服务器的连接。可以为套接字连接指定可选的 *timeout*。如果提供了可选的 *user* 和 *password*，或者如果在 `/.netrc` 中存在适合的凭证并且可选的旗标 *usenetrc* 为真值，则会使用 `AUTHINFO USER` 和 `AUTHINFO PASS` 命令在服务器上标识和认证用户。如果可选的旗标 *readermode* 为真值，则会在执行认证之前发送 `mode reader` 命令。在某些时候如果你是连接本地机器上的 NNTP 服务器并且想要调用读取者专属命令如 `group` 那么还必须使用读取者模式。如果你收到预料之外的*NNTPPermanentError*，你可能需要设置 *readermode*。*NNTP* 类支持使用 `with` 语句来无条件地消费*OSError* 异常并在结束时关闭 NNTP 连接，例如:

```
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.io') as n:
...     n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.comp.
↪python.committers')
>>>
```

引发一个审计事件 `nntplib.connect` 附带参数 `self`, `host`, `port`。

引发一个审计事件 `nntplib.putline`，附带参数 `self`, `line`。

3.2 版更變: *usenetrc* 现在默认为 `False`。

3.3 版更變: 支持了 `with` 语句。

3.9 版更變: 如果 *timeout* 参数设置为 0，创建非阻塞套接字时，它将引发*ValueError* 来阻止该操作。

class nntplib.**NNTP_SSL**(*host*, *port=563*, *user=None*, *password=None*, *ssl_context=None*, *readermode=None*, *usenetrc=False*[, *timeout*])

返回一个新的*NNTP_SSL* 对象，代表一个对运行于主机 *host*，在端口 *port* 上监听的 NNTP 服务器的连接。*NNTP_SSL* 对象具有与*NNTP* 对象相同的方法。如果 *port* 被省略，则会使用端口 563 (NNTPS)。*ssl_context* 也是可选的，且为一个*SSLContext* 对象。请阅读安全考量 来了解最佳实践。所有其他形参的行为都与*NNTP* 的相同。

请注意 **RFC 4642** 不再推荐使用 563 端口的 SSL，建议改用下文描述的 STARTTLS。但是，某些服务器只支持前者。

引发一个审计事件 `nntplib.connect` 附带参数 `self`, `host`, `port`。

引发一个审计事件 `nntplib.putline`，附带参数 `self`, `line`。

3.2 版新加入.

3.4 版更變: 本类现在支持通过*ssl.SSLContext.check_hostname* 和 服务器名称提示（参阅*ssl.HAS_SNI*）进行主机名检查。

3.9 版更變: 如果 *timeout* 参数设置为 0，创建非阻塞套接字时，它将引发*ValueError* 来阻止该操作。

exception nntplib.**NNTPError**

派生自标准异常*Exception*，这是*nntplib* 模块中引发的所有异常的基类。该类的实例具有以下属性:

response

可用的服务器响应，为一*str* 对象。

exception nntplib.**NNTPReplyError**

从服务器收到意外答复时，将引发本异常。

exception nntplib.**NNTPTemporaryError**

收到 400--499 范围内的响应代码时所引发的异常。

exception nntplib.**NNTPPermanentError**
收到 500--599 范围内的响应代码时所引发的异常。

exception nntplib.**NNTPProtocolError**
当从服务器收到不是以数字 1--5 开头的答复时所引发的异常。

exception nntplib.**NNTPDataError**
当响应数据中存在错误时所引发的异常。

36.14.1 NNTP 对象

当连接时，*NNTP* 和*NNTP_SSL* 对象支持以下方法和属性。

属性

NNTP.**nntp_version**
代表服务器所支持的 NNTP 协议版本的整数。在实践中，这对声明遵循 **RFC 3977** 的服务器应为 2 而对其他服务器则为 1。

3.2 版新加入.

NNTP.**nntp_implementation**
描述 NNTP 服务器软件名称和版本的字符串，如果服务器未声明此信息则为*None*。

3.2 版新加入.

方法

作为几乎全部方法所返回元组的第一项返回的 *response* 是服务器的响应：以三位数字代码打头的字符串。如果服务器的响应是提示错误，则方法将引发上述异常之一。

以下方法中许多都接受一个可选的仅限关键字参数 *file*。当提供了 *file* 参数时，它必须为打开用于二进制写入的*file object*，或要写入的磁盘文件名称。此类方法随后将把服务器返回的任意数据（除了响应行和表示结束的点号）写入到文件中；此类方法通常返回的任何行列表、元组或对象都将为空值。

3.2 版更變: 以下方法中许多都已被重写和修正，这使得它们不再与 3.1 中的同名方法相兼容。

NNTP.**quit**()
发送 QUIT 命令并关闭连接。一旦此方法被调用，NNTP 对象的其他方法都不应再被调用。

NNTP.**getwelcome**()
返回服务器发送的欢迎消息，作为连接开始的回复。(该消息有时包含与用户有关的免责声明或帮助信息。)

NNTP.**getcapabilities**()
返回服务器所声明的 **RFC 3977** 功能，其形式为将功能名称映射到（可能为空的）值列表的*dict* 实例。在不能识别 CAPABILITIES 命令的旧式服务器上，会返回一个空字典。

```
>>> s = NNTP('news.gmane.io')
>>> 'POST' in s.getcapabilities()
True
```

3.2 版新加入.

NNTP.**login**(*user=None*, *password=None*, *usenetrc=True*)
发送 AUTHINFO 命令并附带用户名和密码。如果 *user* 和 *password* 为 None 且 *usenetrc* 为真值，则会在可能的情况下使用来自 ~/.netrc 的凭证。

除非被有意延迟，登录操作通常会在*NNTP* 对象初始化期间被执行因而没有必要单独调用此函数。要强制延迟验证，你在创建该对象时不能设置 *user* 或 *password*，并必须将 *usenetrc* 设为 False。

3.2 版新加入.

NNTP.**starttls**(*context=None*)
发送 STARTTLS 命令。这将在 NNTP 连接上启用加密。*context* 参数是可选的且应为*ssl.SSLContext* 对象。请阅读*安全考量* 了解最佳实践。

请注意此操作可能不会在传输验证信息之后立即完成，只要有可能验证默认会在*NNTP* 对象初始化期间发生。请参阅*NNTP.login()* 了解有关如何屏蔽此行为的信息。

3.2 版新加入.

3.4 版更變: 此方法现在支持使用*ssl.SSLContext.check_hostname* 和服务器名称指示 (参见*ssl.HAS_SNI*) 进行主机名检查。

NNTP.**newgroups**(*date*, *, *file=None*)
发送 NEWGROUPS 命令。*date* 参数应为*datetime.date* 或*datetime.datetime* 对象。返回一个 (response, groups) 对，其中 *groups* 是代表给定 i *date* 以来所新建的分组。但是如果提供了 *file*，则 *groups* 将为空值。

```
>>> from datetime import date, timedelta
>>> resp, groups = s.newgroups(date.today() - timedelta(days=3))
>>> len(groups)
85
>>> groups[0]
GroupInfo(group='gmane.network.tor.devel', last='4', first='1', flag='m')
```

NNTP.**newnews**(*group*, *date*, *, *file=None*)
发送 NEWNEWS 命令。这里，*group* 是新闻组名称或为 '*'，而 *date* 与*newgroups()* 中的含义相同。返回一个 (response, articles) 对，其中 *articles* 为消息 ID 列表。

此命令经常会被 NNTP 服务器管理员禁用。

NNTP.**list**(*group_pattern=None*, *, *file=None*)
发送 LIST 或 LIST ACTIVE 命令。返回一个 (response, list) 对，其中 *list* 是代表此 NNTP 服务器上所有可用新闻组的元组列表，并可选择匹配模式字符串 *group_pattern*。每个元组的形式为 (group, last, first, flag)，其中 *group* 为新闻组名称，*last* 和 *first* 是最后一个和第一个文章的编号，而 *flag* 通常为下列值之一:

- y: 允许来自对等方的本地发帖和文章。
- m: 新闻组受到管制因而所有发帖必须经过审核。
- n: 不允许本地发帖，只允许来自组员的文章。
- j: 来自组员的文章会被转入垃圾分组。
- x: 不允许本地发帖，而来自组员的文章会被忽略。
- =foo.bar: 文章会被转入 foo.bar 分组。

如果 *flag* 具有其他值，则新闻组的状态应当被视为未知。

此命令可能返回非常庞大的结果，特别是当未指明 *group_pattern* 的时候。最好是离线缓存其结果，除非你确实需要刷新它们。

3.2 版更變: 增加了 *group_pattern*。

NNTP.**descriptions**(*grouppattern*)
发送 LIST NEWSGROUPS 命令，其中 *grouppattern* 为 **RFC 3977** 中规定的 wildmat 字符串（它实际上与 DOS 或 UNIX shell 通配字符串相同）。返回一个 (response, descriptions) 对，其中 *descriptions* 是将新闻组名称映射到文本描述的字典。

```
>>> resp, descs = s.descriptions('gmane.comp.python.*')
>>> len(descs)
295
>>> descs.popitem()
('gmane.comp.python.bio.general', 'BioPython discussion list (Moderated)')
```

NNTP.**description**(*group*)
获取单个新闻组 *group* 的描述。如果匹配到一个以上的新闻组（如果'group' 是一个真实的 wildmat 字符串），则返回第一个匹配结果。如果未匹配到任何新闻组，则返回空字符串。

此方法略去了来自服务器的响应代码。如果需要响应代码，请使用*descriptions()*。

NNTP.**group**(*name*)
发送 GROUP 命令，其中 *name* 为新闻组名称。该新闻组如果存在，则会被选定为当前新闻组。返回一个元组 (response, count, first, last, name)，其中 *count* 是该新闻组中（估计的）文章数量，*first* 是新闻组中第一篇文章的编号，*last* 是新闻组中最后一篇文章的编号，而 *name* 是新闻组名称。

NNTP.**over**(*message_spec*, *, *file=None*)
发送 OVER 命令，或是旧式服务器上的 XOVER 命令。*message_spec* 可以是表示消息 ID 的字符串，或是指明当前新闻组内文章范围的数字元组 (first, last)，或是指明当前新闻组内从 (first, None) *first* 到最后一篇文章的元组，或者为*None* 表示选定当前新闻组内的当前文章。

返回一个 (response, overviews) 对。其中 *overviews* 是一个包含 (article_number, overview) 元组的列表，每个元组对应 *message_spec* 所选定的一篇文章。每个 *overview* 则是包含同样数量条目的字典，但具体数量取决于服务器。这些条目或是为消息标头（对应键为小写的标头名称）或是为 metadata 项（对应键为以 ":" 打头的 metadata 名称）。以下条目会由 NNTP 规范描述来确保提供:

- subject, from, date, message-id 和 references 标头
- :bytes metadata: 整个原始文章数据的字节数（包括标头和消息体）
- :lines metadata: 文章消息体的行数

每个条目的值或者为字符串，或者在没有值时为*None*。

建议在标头值可能包含非 ASCII 字符的时候对其使用*decode_header()* 函数:

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, overviews = s.over((last, last))
>>> art_num, over = overviews[0]
>>> art_num
117216
>>> list(over.keys())
['xref', 'from', ':lines', ':bytes', 'references', 'date', 'message-id', 'subject
↪']
>>> over['from']
'=?UTF-8?B?Ik1hcnRpbiB2LiBMw7Z3aXMi?= <martin@v.loewis.de>'
>>> nntplib.decode_header(over['from'])
'"Martin v. Löwis" <martin@v.loewis.de>'
```

3.2 版新加入.

NNTP.**help**(*, *file=None*)
发送 HELP 命令。返回一个 (response, list) 对，其中 *list* 为帮助字符串列表。

NNTP.**stat**(*message_spec=None*)
发送 STAT 命令，其中 *message_spec* 为消息 ID (包裹在 '<' 和 '>' 中) 或者当前新闻组中的文章编号。如果 *message_spec* 被省略或为*None*，则会选择当前新闻组中的当前文章。反回一个三元组 (response, number, id)，其中 *number* 为文章编号而 *id* 为消息 ID。

```
>>> _, _, first, last, _ = s.group('gmane.comp.python.devel')
>>> resp, number, message_id = s.stat(first)
>>> number, message_id
(9099, '<20030112190404.GE29873@epoch.metaslash.com>')
```

NNTP.**next**()
发送 NEXT 命令。返回与*stat()* 类似的结果。

NNTP.**last**()
发送 LAST 命令。返回与*stat()* 类似的结果。

NNTP.**article**(*message_spec=None*, *, *file=None*)
发送 ARTICLE 命令，其中 *message_spec* 的含义与*stat()* 中的相同。返回一个元组 (response, info)，其中 *info* 是一个*namedtuple*，包含三个属性 *number*, *message_id* 和 *lines* (按此顺序)。*number* 是新闻组中的文章数量 (或者如果该信息不可用则为 0)，*message_id* 为字符串形式的消息 ID，而 *lines* 为由包括标头和消息体的原始消息的行组成的列表 (不带末尾换行符)。

```
>>> resp, info = s.article('<20030112190404.GE29873@epoch.metaslash.com>')
>>> info.number
0
>>> info.message_id
'<20030112190404.GE29873@epoch.metaslash.com>'
>>> len(info.lines)
65
>>> info.lines[0]
b'Path: main.gmane.org!not-for-mail'
>>> info.lines[1]
b'From: Neal Norwitz <neal@metaslash.com>'
>>> info.lines[-3:]
[b'There is a patch for 2.3 as well as 2.2.', b'', b'Neal']
```

NNTP.**head**(*message_spec=None*, *, *file=None*)
与*article()* 类似，但会发送 HEAD 命令。返回的 *lines* (或写入到 *file*) 将只包含消息标头，不包含消息体。

NNTP.**body**(*message_spec=None*, *, *file=None*)
与*article()* 类似，但会发送 BODY 命令。返回的 *lines* (或写入到 *file*) 将只包含消息体，不包含标头。

NNTP.**post**(*data*)
使用 POST 命令发布文章。*data* 参数是以二进制读取模式打开的*file object*，或是任意包含字节串对象的可迭代对象 (表示要发布的文章的原始行数据)。它应当代表一篇适当格式的新闻组文章，包含所需的标头。*post()* 方法会自动对以 . 打头的行数据进行转义并添加结束行。

如果此方法执行成功，将返回服务器的响应。如果服务器拒绝响应，则会引发*NNTPReplyError*。

NNTP.**ihave**(*message_id*, *data*)
发送 IHAVE 命令。*message_id* 为要发给服务器的消息 ID (包裹在 '<' 和 '>' 中)。*data* 形参和返回值与*post()* 的一致。

NNTP.**date**()
返回一个 (response, date) 对。*date* 是包含服务器当前日期与时间的*datetime* 对象。

NNTP.**slave**()
发送 SLAVE 命令。返回服务器的 响应。

NNTP.**set_debuglevel**(*level*)
设置实例的调试级别。它控制着打印调试输出信息的数量。默认值 0 不产生调试输出。值 1 产生中等数量的调试输出，通常每个请求或响应各产生一行。大于等于 2 的值产生最多的调试输出，在连接上发送和接收的每一行信息都会被记录下来（包括消息文本)。

以下是在 **RFC 2980** 中定义的可选 NNTP 扩展。其中一些已被 **RFC 3977** 中的新命令所取代。

NNTP.**xhdr**(*hdr*, *str*, *, *file=None*)
发送 XHDR 命令。*hdr* 参数是标头关键字，例如 `'subject'`。*str* 参数的形式应为 `'first-last'`，其中 *first* 和 *last* 是要搜索的首篇和末篇文章编号。返回一个 `(response, list)` 对，其中 *list* 是 `(id, text)` 对的列表，其中 *id* 是文章编号（字符串类型）而 *text* 是该文章的请求标头。如果提供了 *file* 形参，则 XHDR 命令的输出会保存到文件中。如果 *file* 为字符串，则此方法将打开指定名称的文件，向其写入内容并将其关闭。如果 *file* 为*file object*，则将在该文件对象上调用 `write()` 方法来保存命令所输出的行信息。如果提供了 *file*，则返回的 *list* 将为空列表。

NNTP.**xover**(*start*, *end*, *, *file=None*)
发送 XOVER 命令。*start* 和 *end* 是限制所选取文章范围的文章编号。返回值与*over()* 的相同。推荐改用*over()*，因为它将在可能的情况下自动使用更新的 OVER 命令。

36.14.2 工具函数

这个模块还定义了下列工具函数:

nntplib.**decode_header**(*header_str*)
解码标头值，恢复任何被转义的非 ASCII 字符。*header_str* 必须为*str* 对象。将返回被恢复的值。推荐使用此函数来以人类可读的形式显示某些标头:

```
>>> decode_header("Some subject")
'Some subject'
>>> decode_header("=?ISO-8859-15?Q?D=E9buter_en_Python?=")
'Débuter en Python'
>>> decode_header("Re: =?UTF-8?B?cHJvYmzDqG1lIGRlIG1hdHJpY2U=?=")
'Re: problème de matrice'
```

36.15 optparse --- 解析器的命令行选项

源代码： Lib/optparse.py

3.2 版後已🄵用: *optparse* 模块已被弃用并且将不再继续开发；开发将转至*argparse* 模块进行。

---

optparse 是一个相比原有*getopt* 模块更为方便、灵活和强大的命令行选项解析库。*optparse* 使用更为显明的命令行解析风格：创建一个*OptionParser* 的实例，向其中填充选项，然后解析命令行。*optparse* 允许用户以传统的 GNU/POSIX 语法来指定选项，并为你生成额外的用法和帮助消息。

下面是在一个简单脚本中使用*optparse* 的示例:

```
from optparse import OptionParser
...
parser = OptionParser()
parser.add_option("-f", "--file", dest="filename",
                  help="write report to FILE", metavar="FILE")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

(options, args) = parser.parse_args()
```

With these few lines of code, users of your script can now do the ”usual thing” on the command-line, for example:

```
<yourscript> --file=outfile -q
```

As it parses the command line, *optparse* sets attributes of the `options` object returned by `parse_args()` based on user-supplied command-line values. When `parse_args()` returns from parsing this command line, `options.filename` will be `"outfile"` and `options.verbose` will be `False`. *optparse* supports both long and short options, allows short options to be merged together, and allows options to be associated with their arguments in a variety of ways. Thus, the following command lines are all equivalent to the above example:

```
<yourscript> -f outfile --quiet
<yourscript> --quiet --file outfile
<yourscript> -q -foutfile
<yourscript> -qfoutfile
```

Additionally, users can run one of the following

```
<yourscript> -h
<yourscript> --help
```

and *optparse* will print out a brief summary of your script's options:

```
Usage: <yourscript> [options]

Options:
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  write report to FILE
  -q, --quiet           don't print status messages to stdout
```

where the value of *yourscript* is determined at runtime (normally from `sys.argv[0]`).

36.15.1 背景

optparse was explicitly designed to encourage the creation of programs with straightforward, conventional command-line interfaces. To that end, it supports only the most common command-line syntax and semantics conventionally used under Unix. If you are unfamiliar with these conventions, read this section to acquaint yourself with them.

术语

参数 a string entered on the command-line, and passed by the shell to `execl()` or `execv()`. In Python, arguments are elements of `sys.argv[1:]` (`sys.argv[0]` is the name of the program being executed). Unix shells also use the term "word".

It is occasionally desirable to substitute an argument list other than `sys.argv[1:]`, so you should read "argument" as "an element of `sys.argv[1:]`, or of some other list provided as a substitute for `sys.argv[1:]`".

选项 an argument used to supply extra information to guide or customize the execution of a program. There are many different syntaxes for options; the traditional Unix syntax is a hyphen ("-") followed by a single letter, e.g. `-x` or `-F`. Also, traditional Unix syntax allows multiple options to be merged into a single argument, e.g. `-x -F` is equivalent to `-xF`. The GNU project introduced `--` followed by a series of hyphen-separated words, e.g. `--file` or `--dry-run`. These are the only two option syntaxes provided by *optparse*.

Some other option syntaxes that the world has seen include:

- a hyphen followed by a few letters, e.g. `-pf` (this is *not* the same as multiple options merged into a single argument)

- a hyphen followed by a whole word, e.g. `-file` (this is technically equivalent to the previous syntax, but they aren't usually seen in the same program)
- a plus sign followed by a single letter, or a few letters, or a word, e.g. `+f`, `+rgb`
- a slash followed by a letter, or a few letters, or a word, e.g. `/f`, `/file`

These option syntaxes are not supported by *optparse*, and they never will be. This is deliberate: the first three are non-standard on any environment, and the last only makes sense if you're exclusively targeting Windows or certain legacy platforms (e.g. VMS, MS-DOS).

可选参数: an argument that follows an option, is closely associated with that option, and is consumed from the argument list when that option is. With *optparse*, option arguments may either be in a separate argument from their option:

```
-f foo
--file foo
```

or included in the same argument:

```
-ffoo
--file=foo
```

Typically, a given option either takes an argument or it doesn't. Lots of people want an "optional option arguments" feature, meaning that some options will take an argument if they see it, and won't if they don't. This is somewhat controversial, because it makes parsing ambiguous: if `-a` takes an optional argument and `-b` is another option entirely, how do we interpret `-ab`? Because of this ambiguity, *optparse* does not support this feature.

positional 参数 something leftover in the argument list after options have been parsed, i.e. after options and their arguments have been parsed and removed from the argument list.

必选选项 an option that must be supplied on the command-line; note that the phrase "required option" is self-contradictory in English. *optparse* doesn't prevent you from implementing required options, but doesn't give you much help at it either.

For example, consider this hypothetical command-line:

```
prog -v --report report.txt foo bar
```

`-v` and `--report` are both options. Assuming that `--report` takes one argument, `report.txt` is an option argument. `foo` and `bar` are positional arguments.

What are options for?

Options are used to provide extra information to tune or customize the execution of a program. In case it wasn't clear, options are usually *optional*. A program should be able to run just fine with no options whatsoever. (Pick a random program from the Unix or GNU toolsets. Can it run without any options at all and still make sense? The main exceptions are `find`, `tar`, and `dd`---all of which are mutant oddballs that have been rightly criticized for their non-standard syntax and confusing interfaces.)

Lots of people want their programs to have "required options". Think about it. If it's required, then it's *not optional*! If there is a piece of information that your program absolutely requires in order to run successfully, that's what positional arguments are for.

As an example of good command-line interface design, consider the humble `cp` utility, for copying files. It doesn't make much sense to try to copy files without supplying a destination and at least one source. Hence, `cp` fails if you run it with no arguments. However, it has a flexible, useful syntax that does not require any options at all:

```
cp SOURCE DEST
cp SOURCE ... DEST-DIR
```

You can get pretty far with just that. Most `cp` implementations provide a bunch of options to tweak exactly how the files are copied: you can preserve mode and modification time, avoid following symlinks, ask before clobbering existing files, etc. But none of this distracts from the core mission of `cp`, which is to copy either one file to another, or several files to another directory.

位置位置

Positional arguments are for those pieces of information that your program absolutely, positively requires to run.

A good user interface should have as few absolute requirements as possible. If your program requires 17 distinct pieces of information in order to run successfully, it doesn't much matter *how* you get that information from the user---most people will give up and walk away before they successfully run the program. This applies whether the user interface is a command-line, a configuration file, or a GUI: if you make that many demands on your users, most of them will simply give up.

In short, try to minimize the amount of information that users are absolutely required to supply---use sensible defaults whenever possible. Of course, you also want to make your programs reasonably flexible. That's what options are for. Again, it doesn't matter if they are entries in a config file, widgets in the "Preferences" dialog of a GUI, or command-line options---the more options you implement, the more flexible your program is, and the more complicated its implementation becomes. Too much flexibility has drawbacks as well, of course; too many options can overwhelm users and make your code much harder to maintain.

36.15.2 教程

While *optparse* is quite flexible and powerful, it's also straightforward to use in most cases. This section covers the code patterns that are common to any *optparse*-based program.

First, you need to import the OptionParser class; then, early in the main program, create an OptionParser instance:

```
from optparse import OptionParser
...
parser = OptionParser()
```

Then you can start defining options. The basic syntax is:

```
parser.add_option(opt_str, ...,
                  attr=value, ...)
```

Each option has one or more option strings, such as `-f` or `--file`, and several option attributes that tell *optparse* what to expect and what to do when it encounters that option on the command line.

Typically, each option will have one short option string and one long option string, e.g.:

```
parser.add_option("-f", "--file", ...)
```

You're free to define as many short option strings and as many long option strings as you like (including zero), as long as there is at least one option string overall.

The option strings passed to *OptionParser.add_option()* are effectively labels for the option defined by that call. For brevity, we will frequently refer to *encountering an option* on the command line; in reality, *optparse* encounters *option strings* and looks up options from them.

Once all of your options are defined, instruct *optparse* to parse your program's command line:

```
(options, args) = parser.parse_args()
```

(If you like, you can pass a custom argument list to `parse_args()`, but that's rarely necessary: by default it uses `sys.argv[1:]`.)

`parse_args()` 返回两个值:

- `options`, an object containing values for all of your options---e.g. if `--file` takes a single string argument, then `options.file` will be the filename supplied by the user, or `None` if the user did not supply that option
- `args`, the list of positional arguments leftover after parsing options

This tutorial section only covers the four most important option attributes: *`action`*, *`type`*, *`dest`* (destination), and *`help`*. Of these, *`action`* is the most fundamental.

Understanding option actions

Actions tell *`optparse`* what to do when it encounters an option on the command line. There is a fixed set of actions hard-coded into *`optparse`*; adding new actions is an advanced topic covered in section *Extending optparse*. Most actions tell *`optparse`* to store a value in some variable---for example, take a string from the command line and store it in an attribute of `options`.

If you don't specify an option action, *`optparse`* defaults to `store`.

The store action

The most common option action is `store`, which tells *`optparse`* to take the next argument (or the remainder of the current argument), ensure that it is of the correct type, and store it to your chosen destination.

例如:

```
parser.add_option("-f", "--file",
                  action="store", type="string", dest="filename")
```

Now let's make up a fake command line and ask *`optparse`* to parse it:

```
args = ["-f", "foo.txt"]
(options, args) = parser.parse_args(args)
```

When *`optparse`* sees the option string `-f`, it consumes the next argument, `foo.txt`, and stores it in `options.filename`. So, after this call to `parse_args()`, `options.filename` is `"foo.txt"`.

Some other option types supported by *`optparse`* are `int` and `float`. Here's an option that expects an integer argument:

```
parser.add_option("-n", type="int", dest="num")
```

Note that this option has no long option string, which is perfectly acceptable. Also, there's no explicit action, since the default is `store`.

Let's parse another fake command-line. This time, we'll jam the option argument right up against the option: since `-n42` (one argument) is equivalent to `-n 42` (two arguments), the code

```
(options, args) = parser.parse_args(["-n42"])
print(options.num)
```

will print `42`.

If you don't specify a type, *`optparse`* assumes `string`. Combined with the fact that the default action is `store`, that means our first example can be a lot shorter:

```
parser.add_option("-f", "--file", dest="filename")
```

If you don't supply a destination, *optparse* figures out a sensible default from the option strings: if the first long option string is `--foo-bar`, then the default destination is `foo_bar`. If there are no long option strings, *optparse* looks at the first short option string: the default destination for `-f` is `f`.

optparse also includes the built-in `complex` type. Adding types is covered in section *Extending optparse*.

Handling boolean (flag) options

Flag options---set a variable to true or false when a particular option is seen---are quite common. *optparse* supports them with two separate actions, `store_true` and `store_false`. For example, you might have a `verbose` flag that is turned on with `-v` and off with `-q`:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose")
```

Here we have two different options with the same destination, which is perfectly OK. (It just means you have to be a bit careful when setting default values---see below.)

When *optparse* encounters `-v` on the command line, it sets `options.verbose` to `True`; when it encounters `-q`, `options.verbose` is set to `False`.

Other actions

Some other actions supported by *optparse* are:

`"store_const"` store a constant value

`"append"` append this option's argument to a list

`"count"` increment a counter by one

`"callback"` 调用指定函数

These are covered in section *参考指南*, and section *Option Callbacks*.

默认值

All of the above examples involve setting some variable (the "destination") when certain command-line options are seen. What happens if those options are never seen? Since we didn't supply any defaults, they are all set to `None`. This is usually fine, but sometimes you want more control. *optparse* lets you supply a default value for each destination, which is assigned before the command line is parsed.

First, consider the verbose/quiet example. If we want *optparse* to set `verbose` to `True` unless `-q` is seen, then we can do this:

```
parser.add_option("-v", action="store_true", dest="verbose", default=True)
parser.add_option("-q", action="store_false", dest="verbose")
```

Since default values apply to the *destination* rather than to any particular option, and these two options happen to have the same destination, this is exactly equivalent:

```
parser.add_option("-v", action="store_true", dest="verbose")
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

考慮一下：

```
parser.add_option("-v", action="store_true", dest="verbose", default=False)
parser.add_option("-q", action="store_false", dest="verbose", default=True)
```

Again, the default value for `verbose` will be `True`: the last default value supplied for any particular destination is the one that counts.

A clearer way to specify default values is the `set_defaults()` method of OptionParser, which you can call at any time before calling `parse_args()`:

```
parser.set_defaults(verbose=True)
parser.add_option(...)
(options, args) = parser.parse_args()
```

As before, the last value specified for a given option destination is the one that counts. For clarity, try to use one method or the other of setting default values, not both.

Generating help

optparse's ability to generate help and usage text automatically is useful for creating user-friendly command-line interfaces. All you have to do is supply a *help* value for each option, and optionally a short usage message for your whole program. Here's an OptionParser populated with user-friendly (documented) options:

```
usage = "usage: %prog [options] arg1 arg2"
parser = OptionParser(usage=usage)
parser.add_option("-v", "--verbose",
                  action="store_true", dest="verbose", default=True,
                  help="make lots of noise [default]")
parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose",
                  help="be vewwy quiet (I'm hunting wabbits)")
parser.add_option("-f", "--filename",
                  metavar="FILE", help="write output to FILE")
parser.add_option("-m", "--mode",
                  default="intermediate",
                  help="interaction mode: novice, intermediate, "
                       "or expert [default: %default]")
```

If *optparse* encounters either `-h` or `--help` on the command-line, or if you just call `parser.print_help()`, it prints the following to standard output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]
```

(If the help output is triggered by a help option, *optparse* exits after printing the help text.)

There's a lot going on here to help *optparse* generate the best possible help message:

- the script defines its own usage message:

```
usage = "usage: %prog [options] arg1 arg2"
```

optparse expands `%prog` in the usage string to the name of the current program, i.e. `os.path.basename(sys.argv[0])`. The expanded string is then printed before the detailed option help.

If you don't supply a usage string, *optparse* uses a bland but sensible default: `"Usage: %prog [options]"`, which is fine if your script doesn't take any positional arguments.

- every option defines a help string, and doesn't worry about line-wrapping---*optparse* takes care of wrapping lines and making the help output look good.
- options that take a value indicate this fact in their automatically-generated help message, e.g. for the "mode" option:

```
-m MODE, --mode=MODE
```

Here, "MODE" is called the meta-variable: it stands for the argument that the user is expected to supply to `-m`/`--mode`. By default, *optparse* converts the destination variable name to uppercase and uses that for the meta-variable. Sometimes, that's not what you want---for example, the `--filename` option explicitly sets `metavar="FILE"`, resulting in this automatically-generated option description:

```
-f FILE, --filename=FILE
```

This is important for more than just saving space, though: the manually written help text uses the meta-variable `FILE` to clue the user in that there's a connection between the semi-formal syntax `-f FILE` and the informal semantic description "write output to FILE". This is a simple but effective way to make your help text a lot clearer and more useful for end users.

- options that have a default value can include `%default` in the help string---*optparse* will replace it with *str()* of the option's default value. If an option has no default value (or the default value is `None`), `%default` expands to `none`.

Grouping Options

When dealing with many options, it is convenient to group these options for better help output. An *OptionParser* can contain several option groups, each of which can contain several options.

An option group is obtained using the class *OptionGroup*:

class optparse.**OptionGroup**(*parser*, *title*, *description=None*)

where

- parser is the *OptionParser* instance the group will be inserted in to
- title is the group title
- description, optional, is a long description of the group

OptionGroup inherits from `OptionContainer` (like *OptionParser*) and so the `add_option()` method can be used to add an option to the group.

Once all the options are declared, using the *OptionParser* method `add_option_group()` the group is added to the previously defined parser.

Continuing with the parser defined in the previous section, adding an *OptionGroup* to a parser is easy:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
```

(下页继续)

(繼續上一頁)

```
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)
```

This would result in the following help output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or
                        expert [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

A bit more complete example might involve using more than one group: still extending the previous example:

```
group = OptionGroup(parser, "Dangerous Options",
                    "Caution: use these options at your own risk.  "
                    "It is believed that some of them bite.")
group.add_option("-g", action="store_true", help="Group option.")
parser.add_option_group(group)

group = OptionGroup(parser, "Debug Options")
group.add_option("-d", "--debug", action="store_true",
                 help="Print debug information")
group.add_option("-s", "--sql", action="store_true",
                 help="Print all SQL statements executed")
group.add_option("-e", action="store_true", help="Print every action done")
parser.add_option_group(group)
```

that results in the following output:

```
Usage: <yourscript> [options] arg1 arg2

Options:
  -h, --help            show this help message and exit
  -v, --verbose         make lots of noise [default]
  -q, --quiet           be vewwy quiet (I'm hunting wabbits)
  -f FILE, --filename=FILE
                        write output to FILE
  -m MODE, --mode=MODE  interaction mode: novice, intermediate, or expert
                        [default: intermediate]

  Dangerous Options:
    Caution: use these options at your own risk.  It is believed that some
    of them bite.

    -g                  Group option.
```

(下页继续)

(繼續上一頁)

```
  Debug Options:
    -d, --debug          Print debug information
    -s, --sql            Print all SQL statements executed
    -e                   Print every action done
```

Another interesting method, in particular when working programmatically with option groups is:

`OptionParser.`**`get_option_group`**`(`*opt_str*`)`

Return the *OptionGroup* to which the short or long option string *opt_str* (e.g. `'-o'` or `'--option'`) belongs. If there's no such *OptionGroup*, return `None`.

Printing a version string

Similar to the brief usage string, *optparse* can also print a version string for your program. You have to supply the string as the `version` argument to OptionParser:

```
parser = OptionParser(usage="%prog [-f] [-q]", version="%prog 1.0")
```

`%prog` is expanded just like it is in `usage`. Apart from that, `version` can contain anything you like. When you supply it, *optparse* automatically adds a `--version` option to your parser. If it encounters this option on the command line, it expands your `version` string (by replacing `%prog`), prints it to stdout, and exits.

For example, if your script is called `/usr/bin/foo`:

```
$ /usr/bin/foo --version
foo 1.0
```

The following two methods can be used to print and get the `version` string:

`OptionParser.`**`print_version`**`(`*file=None*`)`

Print the version message for the current program (`self.version`) to *file* (default stdout). As with *print_usage()*, any occurrence of `%prog` in `self.version` is replaced with the name of the current program. Does nothing if `self.version` is empty or undefined.

`OptionParser.`**`get_version`**`()`

Same as *print_version()* but returns the version string instead of printing it.

How `optparse` handles errors

There are two broad classes of errors that *optparse* has to worry about: programmer errors and user errors. Programmer errors are usually erroneous calls to *OptionParser.add_option()*, e.g. invalid option strings, unknown option attributes, missing option attributes, etc. These are dealt with in the usual way: raise an exception (either `optparse.OptionError` or *TypeError*) and let the program crash.

Handling user errors is much more important, since they are guaranteed to happen no matter how stable your code is. *optparse* can automatically detect some user errors, such as bad option arguments (passing `-n 4x` where `-n` takes an integer argument), missing arguments (`-n` at the end of the command line, where `-n` takes an argument of any type). Also, you can call `OptionParser.error()` to signal an application-defined error condition:

```
(options, args) = parser.parse_args()
...
if options.a and options.b:
    parser.error("options -a and -b are mutually exclusive")
```

In either case, *optparse* handles the error the same way: it prints the program's usage message and an error message to standard error and exits with error status 2.

Consider the first example above, where the user passes `4x` to an option that takes an integer:

```
$ /usr/bin/foo -n 4x
Usage: foo [options]

foo: error: option -n: invalid integer value: '4x'
```

Or, where the user fails to pass a value at all:

```
$ /usr/bin/foo -n
Usage: foo [options]

foo: error: -n option requires an argument
```

optparse-generated error messages take care always to mention the option involved in the error; be sure to do the same when calling `OptionParser.error()` from your application code.

If *optparse*'s default error-handling behaviour does not suit your needs, you'll need to subclass OptionParser and override its `exit()` and/or `error()` methods.

Putting it all together

Here's what *optparse*-based scripts usually look like:

```
from optparse import OptionParser
...
def main():
    usage = "usage: %prog [options] arg"
    parser = OptionParser(usage)
    parser.add_option("-f", "--file", dest="filename",
                      help="read data from FILENAME")
    parser.add_option("-v", "--verbose",
                      action="store_true", dest="verbose")
    parser.add_option("-q", "--quiet",
                      action="store_false", dest="verbose")
    ...
    (options, args) = parser.parse_args()
    if len(args) != 1:
        parser.error("incorrect number of arguments")
    if options.verbose:
        print("reading %s..." % options.filename)
    ...

if __name__ == "__main__":
    main()
```

36.15.3 参考指南

创建解析器

The first step in using *optparse* is to create an OptionParser instance.

class optparse.**OptionParser**(...)

The OptionParser constructor has no required arguments, but a number of optional keyword arguments. You should always pass them as keyword arguments, i.e. do not rely on the order in which the arguments are declared.

usage (默认: **"%prog [options]"**) The usage summary to print when your program is run incorrectly or with a help option. When *optparse* prints the usage string, it expands `%prog` to `os.path.basename(sys.argv[0])` (or to `prog` if you passed that keyword argument). To suppress a usage message, pass the special value `optparse.SUPPRESS_USAGE`.

option_list (默认: **[]**) A list of Option objects to populate the parser with. The options in `option_list` are added after any options in `standard_option_list` (a class attribute that may be set by OptionParser subclasses), but before any version or help options. Deprecated; use *add_option()* after creating the parser instead.

option_class (默认: **optparse.Option**) Class to use when adding options to the parser in *add_option()*.

version (默认: **None**) A version string to print when the user supplies a version option. If you supply a true value for `version`, *optparse* automatically adds a version option with the single option string `--version`. The substring `%prog` is expanded the same as for `usage`.

conflict_handler (默认: **"error"**) Specifies what to do when options with conflicting option strings are added to the parser; see section *Conflicts between options*.

description (默认: **None**) A paragraph of text giving a brief overview of your program. *optparse* reformats this paragraph to fit the current terminal width and prints it when the user requests help (after `usage`, but before the list of options).

formatter (default: a new **IndentedHelpFormatter**) An instance of optparse.HelpFormatter that will be used for printing help text. *optparse* provides two concrete classes for this purpose: IndentedHelpFormatter and TitledHelpFormatter.

add_help_option (默认: **True**) If true, *optparse* will add a help option (with option strings `-h` and `--help`) to the parser.

prog The string to use when expanding `%prog` in `usage` and `version` instead of `os.path.basename(sys.argv[0])`.

epilog (默认: **None**) A paragraph of help text to print after the option help.

填充解析器

There are several ways to populate the parser with options. The preferred way is by using *OptionParser.add_option()*, as shown in section 教程. `add_option()` can be called in one of two ways:

- pass it an Option instance (as returned by `make_option()`)
- pass it any combination of positional and keyword arguments that are acceptable to `make_option()` (i.e., to the Option constructor), and it will create the Option instance for you

The other alternative is to pass a list of pre-constructed Option instances to the OptionParser constructor, as in:

```
option_list = [
    make_option("-f", "--filename",
                action="store", type="string", dest="filename"),
    make_option("-q", "--quiet",
                action="store_false", dest="verbose"),
    ]
parser = OptionParser(option_list=option_list)
```

(`make_option()` is a factory function for creating Option instances; currently it is an alias for the Option constructor. A future version of *optparse* may split Option into several classes, and `make_option()` will pick the right class to instantiate. Do not instantiate Option directly.)

定义选项

Each Option instance represents a set of synonymous command-line option strings, e.g. `-f` and `--file`. You can specify any number of short or long option strings, but you must specify at least one overall option string.

The canonical way to create an `Option` instance is with the `add_option()` method of *OptionParser*.

`OptionParser.`**`add_option`**`(`*option*`)`
`OptionParser.`**`add_option`**`(`**opt_str*, *attr*=*value*, ...`)`

To define an option with only a short option string:

```
parser.add_option("-f", attr=value, ...)
```

And to define an option with only a long option string:

```
parser.add_option("--foo", attr=value, ...)
```

The keyword arguments define attributes of the new Option object. The most important option attribute is *action*, and it largely determines which other attributes are relevant or required. If you pass irrelevant option attributes, or fail to pass required ones, *optparse* raises an `OptionError` exception explaining your mistake.

An option's *action* determines what *optparse* does when it encounters this option on the command-line. The standard option actions hard-coded into *optparse* are:

`"store"` 存储此选项的参数（默认）

`"store_const"` store a constant value

`"store_true"` store `True`

`"store_false"` store `False`

`"append"` append this option's argument to a list

`"append_const"` 将常量值附加到列表

`"count"` increment a counter by one

`"callback"` 调用指定函数

`"help"` 打印用法消息，包括所有选项和文档

(If you don't supply an action, the default is `"store"`. For this action, you may also supply *type* and *dest* option attributes; see *Standard option actions*.)

As you can see, most actions involve storing or updating a value somewhere. *optparse* always creates a special object for this, conventionally called `options` (it happens to be an instance of `optparse.Values`). Option arguments (and various other values) are stored as attributes of this object, according to the *dest* (destination) option attribute.

For example, when you call

```
parser.parse_args()
```

one of the first things *optparse* does is create the `options` object:

```
options = Values()
```

If one of the options in this parser is defined with

```
parser.add_option("-f", "--file", action="store", type="string", dest="filename")
```

and the command-line being parsed includes any of the following:

```
-ffoo
-f foo
--file=foo
--file foo
```

then *optparse*, on seeing this option, will do the equivalent of

```
options.filename = "foo"
```

The *type* and *dest* option attributes are almost as important as *action*, but *action* is the only one that makes sense for *all* options.

Option attributes

The following option attributes may be passed as keyword arguments to *OptionParser.add_option()*. If you pass an option attribute that is not relevant to a particular option, or fail to pass a required option attribute, *optparse* raises `OptionError`.

`Option.`**`action`**
(默认: `"store"`)

Determines *optparse*'s behaviour when this option is seen on the command line; the available options are documented *here*.

`Option.`**`type`**
(默认: `"string"`)

The argument type expected by this option (e.g., `"string"` or `"int"`); the available option types are documented *here*.

`Option.`**`dest`**
(default: derived from option strings)

If the option's action implies writing or modifying a value somewhere, this tells *optparse* where to write it: *dest* names an attribute of the `options` object that *optparse* builds as it parses the command line.

`Option.`**`default`**
The value to use for this option's destination if the option is not seen on the command line. See also *OptionParser.set_defaults()*.

`Option.`**`nargs`**
(默认: 1)

How many arguments of type *type* should be consumed when this option is seen. If > 1, *optparse* will store a tuple of values to *dest*.

Option.**const**
For actions that store a constant value, the constant value to store.

Option.**choices**
For options of type `"choice"`, the list of strings the user may choose from.

Option.**callback**
For options with action `"callback"`, the callable to call when this option is seen. See section *Option Callbacks* for detail on the arguments passed to the callable.

Option.**callback_args**
Option.**callback_kwargs**
Additional positional and keyword arguments to pass to `callback` after the four standard callback arguments.

Option.**help**
Help text to print for this option when listing all available options after the user supplies a *help* option (such as `--help`). If no help text is supplied, the option will be listed without help text. To hide this option, use the special value `optparse.SUPPRESS_HELP`.

Option.**metavar**
(default: derived from option strings)

Stand-in for the option argument(s) to use when printing help text. See section 教程 for an example.

Standard option actions

The various option actions all have slightly different requirements and effects. Most actions have several relevant option attributes which you may specify to guide *optparse*'s behaviour; a few have required attributes, which you must specify for any option using that action.

- `"store"` [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is converted to a value according to *type* and stored in *dest*. If *nargs* > 1, multiple arguments will be consumed from the command line; all will be converted according to *type* and stored to *dest* as a tuple. See the *Standard option types* section.

  If *choices* is supplied (a list or tuple of strings), the type defaults to `"choice"`.

  If *type* is not supplied, it defaults to `"string"`.

  If *dest* is not supplied, *optparse* derives a destination from the first long option string (e.g., `--foo-bar` implies `foo_bar`). If there are no long option strings, *optparse* derives a destination from the first short option string (e.g., `-f` implies `f`).

  示例:

```
parser.add_option("-f")
parser.add_option("-p", type="float", nargs=3, dest="point")
```

  As it parses the command line

```
-f foo.txt -p 1 -3.5 4 -fbar.txt
```

  optparse will set

```
options.f = "foo.txt"
options.point = (1.0, -3.5, 4.0)
options.f = "bar.txt"
```

- "store_const" [required: *const*; relevant: *dest*]

  The value *const* is stored in *dest*.

  示例:

```
parser.add_option("-q", "--quiet",
                  action="store_const", const=0, dest="verbose")
parser.add_option("-v", "--verbose",
                  action="store_const", const=1, dest="verbose")
parser.add_option("--noisy",
                  action="store_const", const=2, dest="verbose")
```

  If --noisy is seen, *optparse* will set

```
options.verbose = 2
```

- "store_true" [relevant: *dest*]

  A special case of "store_const" that stores True to *dest*.

- "store_false" [relevant: *dest*]

  Like "store_true", but stores False.

  示例:

```
parser.add_option("--clobber", action="store_true", dest="clobber")
parser.add_option("--no-clobber", action="store_false", dest="clobber")
```

- "append" [relevant: *type*, *dest*, *nargs*, *choices*]

  The option must be followed by an argument, which is appended to the list in *dest*. If no default value for *dest* is supplied, an empty list is automatically created when *optparse* first encounters this option on the command-line. If *nargs* > 1, multiple arguments are consumed, and a tuple of length *nargs* is appended to *dest*.

  The defaults for *type* and *dest* are the same as for the "store" action.

  示例:

```
parser.add_option("-t", "--tracks", action="append", type="int")
```

  If -t3 is seen on the command-line, *optparse* does the equivalent of:

```
options.tracks = []
options.tracks.append(int("3"))
```

  If, a little later on, --tracks=4 is seen, it does:

```
options.tracks.append(int("4"))
```

  The append action calls the append method on the current value of the option. This means that any default value specified must have an append method. It also means that if the default value is non-empty, the default elements will be present in the parsed value for the option, with any values from the command line appended after those default values:

```
>>> parser.add_option("--files", action="append", default=['~/.mypkg/defaults'])
>>> opts, args = parser.parse_args(['--files', 'overrides.mypkg'])
>>> opts.files
['~/.mypkg/defaults', 'overrides.mypkg']
```

- "append_const" [required: *const*; relevant: *dest*]

  Like "store_const", but the value *const* is appended to *dest*; as with "append", *dest* defaults to None, and an empty list is automatically created the first time the option is encountered.

- "count" [relevant: *dest*]

  Increment the integer stored at *dest*. If no default value is supplied, *dest* is set to zero before being incremented the first time.

  示例:

```
parser.add_option("-v", action="count", dest="verbosity")
```

  The first time -v is seen on the command line, *optparse* does the equivalent of:

```
options.verbosity = 0
options.verbosity += 1
```

  Every subsequent occurrence of -v results in

```
options.verbosity += 1
```

- "callback" [required: *callback*; relevant: *type*, *nargs*, *callback_args*, *callback_kwargs*]

  Call the function specified by *callback*, which is called as

```
func(option, opt_str, value, parser, *args, **kwargs)
```

  See section *Option Callbacks* for more detail.

- "help"

  Prints a complete help message for all the options in the current option parser. The help message is constructed from the usage string passed to OptionParser's constructor and the *help* string passed to every option.

  If no *help* string is supplied for an option, it will still be listed in the help message. To omit an option entirely, use the special value optparse.SUPPRESS_HELP.

  optparse automatically adds a *help* option to all OptionParsers, so you do not normally need to create one.

  示例:

```
from optparse import OptionParser, SUPPRESS_HELP

# usually, a help option is added automatically, but that can
# be suppressed using the add_help_option argument
parser = OptionParser(add_help_option=False)

parser.add_option("-h", "--help", action="help")
parser.add_option("-v", action="store_true", dest="verbose",
                  help="Be moderately verbose")
parser.add_option("--file", dest="filename",
                  help="Input file to read data from")
parser.add_option("--secret", help=SUPPRESS_HELP)
```

  If *optparse* sees either -h or --help on the command line, it will print something like the following help message to stdout (assuming sys.argv[0] is "foo.py"):

```
Usage: foo.py [options]

Options:
  -h, --help        Show this help message and exit
  -v                Be moderately verbose
  --file=FILENAME   Input file to read data from
```

After printing the help message, *optparse* terminates your process with `sys.exit(0)`.

- `"version"`

  Prints the version number supplied to the OptionParser to stdout and exits. The version number is actually formatted and printed by the `print_version()` method of OptionParser. Generally only relevant if the `version` argument is supplied to the OptionParser constructor. As with *help* options, you will rarely create `version` options, since *optparse* automatically adds them when needed.

Standard option types

optparse has five built-in option types: `"string"`, `"int"`, `"choice"`, `"float"` and `"complex"`. If you need to add new option types, see section *Extending optparse*.

Arguments to string options are not checked or converted in any way: the text on the command line is stored in the destination (or passed to the callback) as-is.

Integer arguments (type `"int"`) are parsed as follows:

- if the number starts with `0x`, it is parsed as a hexadecimal number
- if the number starts with `0`, it is parsed as an octal number
- if the number starts with `0b`, it is parsed as a binary number
- otherwise, the number is parsed as a decimal number

The conversion is done by calling *int()* with the appropriate base (2, 8, 10, or 16). If this fails, so will *optparse*, although with a more useful error message.

`"float"` and `"complex"` option arguments are converted directly with *float()* and *complex()*, with similar error-handling.

`"choice"` options are a subtype of `"string"` options. The *choices* option attribute (a sequence of strings) defines the set of allowed option arguments. `optparse.check_choice()` compares user-supplied option arguments against this master list and raises `OptionValueError` if an invalid string is given.

解析参数

The whole point of creating and populating an OptionParser is to call its `parse_args()` method:

```
(options, args) = parser.parse_args(args=None, values=None)
```

输入参数的位置

args the list of arguments to process (default: `sys.argv[1:]`)

values an `optparse.Values` object to store option arguments in (default: a new instance of `Values`) -- if you give an existing object, the option defaults will not be initialized on it

and the return values are

options the same object that was passed in as `values`, or the optparse.Values instance created by *optparse*

args the leftover positional arguments after all options have been processed

The most common usage is to supply neither keyword argument. If you supply `values`, it will be modified with repeated *`setattr()`* calls (roughly one for every option argument stored to an option destination) and returned by `parse_args()`.

If `parse_args()` encounters any errors in the argument list, it calls the OptionParser's `error()` method with an appropriate end-user error message. This ultimately terminates your process with an exit status of 2 (the traditional Unix exit status for command-line errors).

Querying and manipulating your option parser

The default behavior of the option parser can be customized slightly, and you can also poke around your option parser and see what's there. OptionParser provides several methods to help you out:

`OptionParser.`**`disable_interspersed_args`**`()`
Set parsing to stop on the first non-option. For example, if `-a` and `-b` are both simple options that take no arguments, *`optparse`* normally accepts this syntax:

```
prog -a arg1 -b arg2
```

and treats it as equivalent to

```
prog -a -b arg1 arg2
```

To disable this feature, call *`disable_interspersed_args()`*. This restores traditional Unix syntax, where option parsing stops with the first non-option argument.

Use this if you have a command processor which runs another command which has options of its own and you want to make sure these options don't get confused. For example, each command might have a different set of options.

`OptionParser.`**`enable_interspersed_args`**`()`
Set parsing to not stop on the first non-option, allowing interspersing switches with command arguments. This is the default behavior.

`OptionParser.`**`get_option`**`(`*opt_str*`)`
Returns the Option instance with the option string *opt_str*, or `None` if no options have that option string.

`OptionParser.`**`has_option`**`(`*opt_str*`)`
Return `True` if the OptionParser has an option with option string *opt_str* (e.g., `-q` or `--verbose`).

`OptionParser.`**`remove_option`**`(`*opt_str*`)`
If the *`OptionParser`* has an option corresponding to *opt_str*, that option is removed. If that option provided any other option strings, all of those option strings become invalid. If *opt_str* does not occur in any option belonging to this *`OptionParser`*, raises *`ValueError`*.

Conflicts between options

If you're not careful, it's easy to define options with conflicting option strings:

```
parser.add_option("-n", "--dry-run", ...)
...
parser.add_option("-n", "--noisy", ...)
```

(This is particularly true if you've defined your own OptionParser subclass with some standard options.)

Every time you add an option, *`optparse`* checks for conflicts with existing options. If it finds any, it invokes the current conflict-handling mechanism. You can set the conflict-handling mechanism either in the constructor:

```
parser = OptionParser(..., conflict_handler=handler)
```

or with a separate call:

```
parser.set_conflict_handler(handler)
```

The available conflict handlers are:

"error" (默认) assume option conflicts are a programming error and raise `OptionConflictError`

"resolve" resolve option conflicts intelligently (see below)

As an example, let's define an *OptionParser* that resolves conflicts intelligently and add conflicting options to it:

```
parser = OptionParser(conflict_handler="resolve")
parser.add_option("-n", "--dry-run", ..., help="do no harm")
parser.add_option("-n", "--noisy", ..., help="be noisy")
```

At this point, *optparse* detects that a previously-added option is already using the `-n` option string. Since `conflict_handler` is `"resolve"`, it resolves the situation by removing `-n` from the earlier option's list of option strings. Now `--dry-run` is the only way for the user to activate that option. If the user asks for help, the help message will reflect that:

```
Options:
  --dry-run     do no harm
  ...
  -n, --noisy   be noisy
```

It's possible to whittle away the option strings for a previously-added option until there are none left, and the user has no way of invoking that option from the command-line. In that case, *optparse* removes that option completely, so it doesn't show up in help text or anywhere else. Carrying on with our existing OptionParser:

```
parser.add_option("--dry-run", ..., help="new dry-run option")
```

At this point, the original `-n`/`--dry-run` option is no longer accessible, so *optparse* removes it, leaving this help text:

```
Options:
  ...
  -n, --noisy   be noisy
  --dry-run     new dry-run option
```

清理

OptionParser instances have several cyclic references. This should not be a problem for Python's garbage collector, but you may wish to break the cyclic references explicitly by calling `destroy()` on your OptionParser once you are done with it. This is particularly useful in long-running applications where large object graphs are reachable from your OptionParser.

Other methods

OptionParser supports several other public methods:

`OptionParser.`**`set_usage`**(*usage*)
Set the usage string according to the rules described above for the `usage` constructor keyword argument. Passing `None` sets the default usage string; use `optparse.SUPPRESS_USAGE` to suppress a usage message.

`OptionParser.`**`print_usage`**(*file=None*)
Print the usage message for the current program (`self.usage`) to *file* (default stdout). Any occurrence of the string `%prog` in `self.usage` is replaced with the name of the current program. Does nothing if `self.usage` is empty or not defined.

`OptionParser.`**`get_usage`**()
Same as *`print_usage()`* but returns the usage string instead of printing it.

`OptionParser.`**`set_defaults`**(*dest=value, ...*)
Set default values for several option destinations at once. Using *`set_defaults()`* is the preferred way to set default values for options, since multiple options can share the same destination. For example, if several "mode" options all set the same destination, any one of them can set the default, and the last one wins:

```
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced",
                  default="novice")     # overridden below
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice",
                  default="advanced")   # overrides above setting
```

To avoid this confusion, use *`set_defaults()`*:

```
parser.set_defaults(mode="advanced")
parser.add_option("--advanced", action="store_const",
                  dest="mode", const="advanced")
parser.add_option("--novice", action="store_const",
                  dest="mode", const="novice")
```

36.15.4 Option Callbacks

When *`optparse`*'s built-in actions and types aren't quite enough for your needs, you have two choices: extend *`optparse`* or define a callback option. Extending *`optparse`* is more general, but overkill for a lot of simple cases. Quite often a simple callback is all you need.

There are two steps to defining a callback option:

- define the option itself using the `"callback"` action
- write the callback; this is a function (or method) that takes at least four arguments, as described below

Defining a callback option

As always, the easiest way to define a callback option is by using the *OptionParser.add_option()* method. Apart from *action*, the only option attribute you must specify is `callback`, the function to call:

```
parser.add_option("-c", action="callback", callback=my_callback)
```

`callback` is a function (or other callable object), so you must have already defined `my_callback()` when you create this callback option. In this simple case, *optparse* doesn't even know if `-c` takes any arguments, which usually means that the option takes no arguments---the mere presence of `-c` on the command-line is all it needs to know. In some circumstances, though, you might want your callback to consume an arbitrary number of command-line arguments. This is where writing callbacks gets tricky; it's covered later in this section.

optparse always passes four particular arguments to your callback, and it will only pass additional arguments if you specify them via *callback_args* and *callback_kwargs*. Thus, the minimal callback function signature is:

```
def my_callback(option, opt, value, parser):
```

The four arguments to a callback are described below.

There are several other option attributes that you can supply when you define a callback option:

type has its usual meaning: as with the `"store"` or `"append"` actions, it instructs *optparse* to consume one argument and convert it to *type*. Rather than storing the converted value(s) anywhere, though, *optparse* passes it to your callback function.

nargs also has its usual meaning: if it is supplied and > 1, *optparse* will consume *nargs* arguments, each of which must be convertible to *type*. It then passes a tuple of converted values to your callback.

callback_args a tuple of extra positional arguments to pass to the callback

callback_kwargs a dictionary of extra keyword arguments to pass to the callback

How callbacks are called

All callbacks are called as follows:

```
func(option, opt_str, value, parser, *args, **kwargs)
```

where

option is the Option instance that's calling the callback

opt_str is the option string seen on the command-line that's triggering the callback. (If an abbreviated long option was used, `opt_str` will be the full, canonical option string---e.g. if the user puts `--foo` on the command-line as an abbreviation for `--foobar`, then `opt_str` will be `"--foobar"`.)

value is the argument to this option seen on the command-line. *optparse* will only expect an argument if *type* is set; the type of `value` will be the type implied by the option's type. If *type* for this option is `None` (no argument expected), then `value` will be `None`. If *nargs* > 1, `value` will be a tuple of values of the appropriate type.

parser is the OptionParser instance driving the whole thing, mainly useful because you can access some other interesting data through its instance attributes:

parser.largs the current list of leftover arguments, ie. arguments that have been consumed but are neither options nor option arguments. Feel free to modify `parser.largs`, e.g. by adding more arguments to it. (This list will become `args`, the second return value of `parse_args()`.)

parser.rargs the current list of remaining arguments, ie. with `opt_str` and `value` (if applicable) removed, and only the arguments following them still there. Feel free to modify `parser.rargs`, e.g. by consuming more arguments.

parser.values the object where option values are by default stored (an instance of optparse.OptionValues). This lets callbacks use the same mechanism as the rest of *optparse* for storing option values; you don't need to mess around with globals or closures. You can also access or modify the value(s) of any options already encountered on the command-line.

args is a tuple of arbitrary positional arguments supplied via the *callback_args* option attribute.

kwargs is a dictionary of arbitrary keyword arguments supplied via *callback_kwargs*.

Raising errors in a callback

The callback function should raise `OptionValueError` if there are any problems with the option or its argument(s). *optparse* catches this and terminates the program, printing the error message you supply to stderr. Your message should be clear, concise, accurate, and mention the option at fault. Otherwise, the user will have a hard time figuring out what they did wrong.

Callback example 1: trivial callback

Here's an example of a callback option that takes no arguments, and simply records that the option was seen:

```
def record_foo_seen(option, opt_str, value, parser):
    parser.values.saw_foo = True

parser.add_option("--foo", action="callback", callback=record_foo_seen)
```

Of course, you could do that with the `"store_true"` action.

Callback example 2: check option order

Here's a slightly more interesting example: record the fact that `-a` is seen, but blow up if it comes after `-b` in the command-line.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use -a after -b")
    parser.values.a = 1
...
parser.add_option("-a", action="callback", callback=check_order)
parser.add_option("-b", action="store_true", dest="b")
```

Callback example 3: check option order (generalized)

If you want to re-use this callback for several similar options (set a flag, but blow up if `-b` has already been seen), it needs a bit of work: the error message and the flag that it sets must be generalized.

```
def check_order(option, opt_str, value, parser):
    if parser.values.b:
        raise OptionValueError("can't use %s after -b" % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("-a", action="callback", callback=check_order, dest='a')
parser.add_option("-b", action="store_true", dest="b")
parser.add_option("-c", action="callback", callback=check_order, dest='c')
```

Callback example 4: check arbitrary condition

Of course, you could put any condition in there---you're not limited to checking the values of already-defined options. For example, if you have options that should not be called when the moon is full, all you have to do is this:

```
def check_moon(option, opt_str, value, parser):
    if is_moon_full():
        raise OptionValueError("%s option invalid when moon is full"
                               % opt_str)
    setattr(parser.values, option.dest, 1)
...
parser.add_option("--foo",
                  action="callback", callback=check_moon, dest="foo")
```

(The definition of `is_moon_full()` is left as an exercise for the reader.)

Callback example 5: fixed arguments

Things get slightly more interesting when you define callback options that take a fixed number of arguments. Specifying that a callback option takes arguments is similar to defining a `"store"` or `"append"` option: if you define *type*, then the option takes one argument that must be convertible to that type; if you further define *nargs*, then the option takes *nargs* arguments.

Here's an example that just emulates the standard `"store"` action:

```
def store_value(option, opt_str, value, parser):
    setattr(parser.values, option.dest, value)
...
parser.add_option("--foo",
                  action="callback", callback=store_value,
                  type="int", nargs=3, dest="foo")
```

Note that *optparse* takes care of consuming 3 arguments and converting them to integers for you; all you have to do is store them. (Or whatever; obviously you don't need a callback for this example.)

Callback example 6: variable arguments

Things get hairy when you want an option to take a variable number of arguments. For this case, you must write a callback, as *optparse* doesn't provide any built-in capabilities for it. And you have to deal with certain intricacies of conventional Unix command-line parsing that *optparse* normally handles for you. In particular, callbacks should implement the conventional rules for bare `--` and `-` arguments:

- either `--` or `-` can be option arguments
- bare `--` (if not the argument to some option): halt command-line processing and discard the `--`
- bare `-` (if not the argument to some option): halt command-line processing but keep the `-` (append it to `parser.largs`)

If you want an option that takes a variable number of arguments, there are several subtle, tricky issues to worry about. The exact implementation you choose will be based on which trade-offs you're willing to make for your application (which is why *optparse* doesn't support this sort of thing directly).

Nevertheless, here's a stab at a callback for an option with variable arguments:

```
def vararg_callback(option, opt_str, value, parser):
    assert value is None
    value = []

    def floatable(str):
        try:
            float(str)
            return True
        except ValueError:
            return False

    for arg in parser.rargs:
        # stop on --foo like options
        if arg[:2] == "--" and len(arg) > 2:
            break
        # stop on -a, but not on -3 or -3.0
        if arg[:1] == "-" and len(arg) > 1 and not floatable(arg):
            break
        value.append(arg)

    del parser.rargs[:len(value)]
    setattr(parser.values, option.dest, value)

...
parser.add_option("-c", "--callback", dest="vararg_attr",
                  action="callback", callback=vararg_callback)
```

36.15.5 Extending `optparse`

Since the two major controlling factors in how *optparse* interprets command-line options are the action and type of each option, the most likely direction of extension is to add new actions and new types.

Adding new types

To add new types, you need to define your own subclass of *optparse*'s `Option` class. This class has a couple of attributes that define *optparse*'s types: *TYPES* and *TYPE_CHECKER*.

`Option.`**`TYPES`**
A tuple of type names; in your subclass, simply define a new tuple *TYPES* that builds on the standard one.

`Option.`**`TYPE_CHECKER`**
A dictionary mapping type names to type-checking functions. A type-checking function has the following signature:

```
def check_mytype(option, opt, value)
```

where `option` is an `Option` instance, `opt` is an option string (e.g., `-f`), and `value` is the string from the command line that must be checked and converted to your desired type. `check_mytype()` should return an object of the hypothetical type `mytype`. The value returned by a type-checking function will wind up in the OptionValues instance returned by `OptionParser.parse_args()`, or be passed to a callback as the `value` parameter.

Your type-checking function should raise `OptionValueError` if it encounters any problems. `OptionValueError` takes a single string argument, which is passed as-is to *OptionParser*'s `error()` method, which in turn prepends the program name and the string `"error:"` and prints everything to stderr before terminating the process.

Here's a silly example that demonstrates adding a `"complex"` option type to parse Python-style complex numbers on the command line. (This is even sillier than it used to be, because *optparse* 1.3 added built-in support for complex numbers, but never mind.)

First, the necessary imports:

```
from copy import copy
from optparse import Option, OptionValueError
```

You need to define your type-checker first, since it's referred to later (in the *TYPE_CHECKER* class attribute of your Option subclass):

```
def check_complex(option, opt, value):
    try:
        return complex(value)
    except ValueError:
        raise OptionValueError(
            "option %s: invalid complex value: %r" % (opt, value))
```

Finally, the Option subclass:

```
class MyOption (Option):
    TYPES = Option.TYPES + ("complex",)
    TYPE_CHECKER = copy(Option.TYPE_CHECKER)
    TYPE_CHECKER["complex"] = check_complex
```

(If we didn't make a *copy()* of *Option.TYPE_CHECKER*, we would end up modifying the *TYPE_CHECKER* attribute of *optparse*'s Option class. This being Python, nothing stops you from doing that except good manners and common sense.)

That's it! Now you can write a script that uses the new option type just like any other *optparse*-based script, except you have to instruct your OptionParser to use MyOption instead of Option:

```
parser = OptionParser(option_class=MyOption)
parser.add_option("-c", type="complex")
```

Alternately, you can build your own option list and pass it to OptionParser; if you don't use `add_option()` in the above way, you don't need to tell OptionParser which option class to use:

```
option_list = [MyOption("-c", action="store", type="complex", dest="c")]
parser = OptionParser(option_list=option_list)
```

Adding new actions

Adding new actions is a bit trickier, because you have to understand that *optparse* has a couple of classifications for actions:

"store" actions actions that result in *optparse* storing a value to an attribute of the current OptionValues instance; these options require a *dest* attribute to be supplied to the Option constructor.

"typed" actions actions that take a value from the command line and expect it to be of a certain type; or rather, a string that can be converted to a certain type. These options require a *type* attribute to the Option constructor.

These are overlapping sets: some default "store" actions are `"store"`, `"store_const"`, `"append"`, and `"count"`, while the default "typed" actions are `"store"`, `"append"`, and `"callback"`.

When you add an action, you need to categorize it by listing it in at least one of the following class attributes of Option (all are lists of strings):

`Option.`**`ACTIONS`**
All actions must be listed in ACTIONS.

`Option.`**`STORE_ACTIONS`**
"store" actions are additionally listed here.

`Option.`**`TYPED_ACTIONS`**
"typed" actions are additionally listed here.

`Option.`**`ALWAYS_TYPED_ACTIONS`**
Actions that always take a type (i.e. whose options always take a value) are additionally listed here. The only effect of this is that *optparse* assigns the default type, `"string"`, to options with no explicit type whose action is listed in *ALWAYS_TYPED_ACTIONS*.

In order to actually implement your new action, you must override Option's `take_action()` method and add a case that recognizes your action.

For example, let's add an `"extend"` action. This is similar to the standard `"append"` action, but instead of taking a single value from the command-line and appending it to an existing list, `"extend"` will take multiple values in a single comma-delimited string, and extend an existing list with them. That is, if `--names` is an `"extend"` option of type `"string"`, the command line

```
--names=foo,bar --names blah --names ding,dong
```

would result in a list

```
["foo", "bar", "blah", "ding", "dong"]
```

Again we define a subclass of Option:

```
class MyOption(Option):

    ACTIONS = Option.ACTIONS + ("extend",)
    STORE_ACTIONS = Option.STORE_ACTIONS + ("extend",)
    TYPED_ACTIONS = Option.TYPED_ACTIONS + ("extend",)
```

(下页继续)

(繼續上一頁)

```
    ALWAYS_TYPED_ACTIONS = Option.ALWAYS_TYPED_ACTIONS + ("extend",)

    def take_action(self, action, dest, opt, value, values, parser):
        if action == "extend":
            lvalue = value.split(",")
            values.ensure_value(dest, []).extend(lvalue)
        else:
            Option.take_action(
                self, action, dest, opt, value, values, parser)
```

Features of note:

- `"extend"` both expects a value on the command-line and stores that value somewhere, so it goes in both *STORE_ACTIONS* and *TYPED_ACTIONS*.
- to ensure that *optparse* assigns the default type of `"string"` to `"extend"` actions, we put the `"extend"` action in *ALWAYS_TYPED_ACTIONS* as well.
- `MyOption.take_action()` implements just this one new action, and passes control back to `Option.take_action()` for the standard *optparse* actions.
- `values` is an instance of the optparse_parser.Values class, which provides the very useful `ensure_value()` method. `ensure_value()` is essentially *getattr()* with a safety valve; it is called as

  ```
  values.ensure_value(attr, value)
  ```

  If the `attr` attribute of `values` doesn't exist or is `None`, then ensure_value() first sets it to `value`, and then returns 'value. This is very handy for actions like `"extend"`, `"append"`, and `"count"`, all of which accumulate data in a variable and expect that variable to be of a certain type (a list for the first two, an integer for the latter). Using `ensure_value()` means that scripts using your action don't have to worry about setting a default value for the option destinations in question; they can just leave the default as `None` and `ensure_value()` will take care of getting it right when it's needed.

36.16 ossaudiodev --- 访问兼容 OSS 的音频设备

3.11 版後已𝔽用: The *ossaudiodev* module is deprecated (see **PEP 594** for details).

该模块允许您访问 OSS（开放式音响系统）音频接口。OSS 可用于广泛的开源和商业 Unices，并且是 Linux 和最新版本的 FreeBSD 的标准音频接口。

3.3 版更變: 此模块中过去会引发 *IOError* 的操作现在将引发 *OSError*。

也参考:

开放之声系统程序员手册 OSS C API 的官方文档

该模块定义了大量由 OSS 设备驱动提供的常量；请参阅 "<sys/soundcard.h>" Linux 或 FreeBSD 上的列表。

ossaudiodev 定义了下列变量和函数:

exception ossaudiodev.**OSSAudioError**

此异常会针对特定错误被引发。其参数为一个描述错误信息的字符串。

(如果 *ossaudiodev* 从系统调用例如 `open()`, `write()` 或 `ioctl()` 接收到错误，它将引发 *OSError*。由 *ossaudiodev* 直接检测到的错误将引发 *OSSAudioError*。)

(为了向下兼容，此异常类也可通过 `ossaudiodev.error` 访问。)

ossaudiodev.**open**(*mode*)
ossaudiodev.**open**(*device*, *mode*)
打开一个音频设备并返回 OSS 音频设备对象。此对象支持许多文件类方法，例如 `read()`, `write()` 和 `fileno()` (不过传统的 Unix 读/写语义与 OSS 音频设备的存在一些细微的差异)。它还支持一些音频专属的方法；完整的方法列表见下文。

device 是要使用的音频设备文件名。如果未指定，则此模块会先在环境变量 `AUDIODEV` 中查找要使用的设备。如果未找到，它将回退为 `/dev/dsp`。

mode 可以为 `'r'` 表示只读（录音）访问，`'w'` 表示只写（回放）访问以及 `'rw'` 表示同时读写。由于许多声卡在同一时间只允许单个进程打开录音机或播放器，因此好的做法是只根据活动的需要打开设备。并且，有些声卡是半双工的：它们可以被打开用于读取或写入，但不能同时读写。

请注意这里特殊的调用语法: *first* 参数是可选的，而第二个参数则是必需的。这是出于历史原因要与*ossaudiodev* 所替代的 `linuxaudiodev` 模块保持兼容。

ossaudiodev.**openmixer**([*device*])
打开一个混音设备并返回 OSS 混音设备对象。*device* 是要使用的混音设备文件名。如果未指定，则此模块会先在环境变量 `MIXERDEV` 中查找要使用的设备。如果未找到，它将回退为 `/dev/mixer`。

36.16.1 音频设备对象

在你写入或读取音频设备之前，你必须按照正确的顺序调用三个方法:

1. `setfmt()` 设置输出格式
2. `channels()` 设置声道数量
3. `speed()` 设置采样率

或者，你也可以使用 `setparameters()` 方法一次性地设置全部三个音频参数。这更为便捷，但可能不会在所有场景下都一样灵活。

open() 所返回的音频设备对象定义了下列方法和（只读）属性:

oss_audio_device.**close**()
显式地关闭音频设备。当你完成写入或读取音频设备后，你应当显式地关闭它。已关闭的设备不可被再次使用。

oss_audio_device.**fileno**()
返回与设备相关联的文件描述符。

oss_audio_device.**read**(*size*)
从音频输入设备读取 *size* 个字节并返回为 Python 字符串。与大多数 Unix 设备驱动不同，处于阻塞模式（默认）的 OSS 音频设备将阻塞*read()* 直到所请求大小的数据全部可用。

oss_audio_device.**write**(*data*)
将一个*bytes-like object* *data* 写入音频设备并返回写入的字节数。如果音频设备处于阻塞模式（默认），则总是会写入完整数据（这还是不同于通常的 Unix 设备语义）。如果设备处于非阻塞模式，则可能会有部分数据未被写入 --- 参见*writeall()*。

3.5 版更變: 现在支持可写的字节类对象。

oss_audio_device.**writeall**(*data*)
将一个*bytes-like object* *data* 写入音频设备：等待直到音频设备能够接收数据，将根据其所能接收的数据量尽可能多地写入，并重复操作直至 *data* 被完全写入。如果设备处于阻塞模式（默认），则其效果与*write()* 相同；*writeall()* 仅适用于非阻塞模式。它没有返回值，因为写入的数据量总是等于所提供的数据量。

3.5 版更變: 现在支持可写的字节类对象。

3.2 版更變: 音频设备对象还支持上下文管理协议，就是说它们可以在 with 语句中使用。

下列方法各自映射一个 ioctl() 系统调用。对应关系很明显: 例如，setfmt() 对应 SNDCTL_DSP_SETFMT ioctl，而 sync() 对应 SNDCTL_DSP_SYNC (这在查阅 OSS 文档时很有用)。如果下层的 ioctl() 失败，它们将引发*OSError*。

oss_audio_device.**nonblock**()
将设备转为非阻塞模式。一旦处于非阻塞模式，将无法将其转回阻塞模式。

oss_audio_device.**getfmts**()
返回声卡所支持的音频输出格式的位掩码。OSS 支持的一部分格式如下:

文件格式	描述
AFMT_MU_LAW	一种对数编码格式（被 Sun .au 文件和 /dev/audio 所使用）
AFMT_A_LAW	一种对数编码格式
AFMT_IMA_ADPCM	一种 4:1 压缩格式，由 Interactive Multimedia Association 定义
AFMT_U8	无符号的 8 位音频
AFMT_S16_LE	有符号的 16 位音频，采用小端字节序（如 Intel 处理器所用的）
AFMT_S16_BE	有符号的 16 位音频，采用大端字节序（如 68k, PowerPC, Sparc 所用的）
AFMT_S8	有符号的 8 位音频
AFMT_U16_LE	无符号的 16 位小端字节序音频
AFMT_U16_BE	无符号的 16 位大端字节序音频

请参阅 OSS 文档获取音频格式的完整列表，还要注意大多数设备都只支持这些列表的一个子集。某些较旧的设备仅支持 AFMT_U8；目前最为常用的格式是 AFMT_S16_LE。

oss_audio_device.**setfmt**(*format*)
尝试将当前音频格式设为 *format* --- 请参阅*getfmts()* 获取格式列表。返回为设备设置的音频格式，这可能并非所请求的格式。也可被用来返回当前音频格式 --- 这可以通过传入特殊的" 音频格式" AFMT_QUERY 来实现。

oss_audio_device.**channels**(*nchannels*)
将输出声道数设为 *nchannels*。值为 1 表示单声道，2 表示立体声。某些设备可能拥有 2 个以上的声道，并且某些高端设备还可能不支持单声道。返回为设备设置的声道数。

oss_audio_device.**speed**(*samplerate*)
尝试将音频采样率设为每秒 *samplerate* 次采样。返回实际设置的采样率。大多数设备都不支持任意的采样率。常见的采样率为:

采样率	描述
8000	/dev/audio 的默认采样率
11025	语音录音
22050	
44100	CD 品质的音频（16 位采样和 2 通道）
96000	DVD 品质的音频（24 位采样）

oss_audio_device.**sync**()
等待直到音频设备播放完其缓冲区中的所有字节。（这会在设备被关闭时隐式地发生。）OSS 建议关闭再重新打开设备而不是使用*sync()*。

oss_audio_device.**reset**()
立即停止播放或录制并使设备返回可接受命令的状态。OSS 文档建议在调用*reset()* 之后关闭并重新打开设备。

oss_audio_device.**post**()
告知设备在输出中可能有暂停，使得设备可以更智能地处理暂停。你可以在播放一个定点音效之后、等待用户输入之前或执行磁盘 I/O 之前使用此方法。

下列便捷方法合并了多个 ioctl，或是合并了一个 ioctl 与某些简单的运算。

oss_audio_device.**setparameters**(*format*, *nchannels*, *samplerate*[, *strict=False*])
在一次方法调用中设置关键的音频采样参数 --- 采样格式、声道数和采样率。*format*, *nchannels* 和 *samplerate* 应当与在*setfmt()*, *channels()* 和*speed()* 方法中所指定的一致。如果 *strict* 为真值，则*setparameters()* 会检查每个参数是否确实被设置为所请求的值，如果不是则会引发*OSSAudioError*。返回一个元组 (*format*, *nchannels*, *samplerate*) 指明由设备驱动实际设置的参数值 (即与*setfmt()*, *channels()* 和*speed()* 的返回值相同)。

例如，:

```
(fmt, channels, rate) = dsp.setparameters(fmt, channels, rate)
```

等价于

```
fmt = dsp.setfmt(fmt)
channels = dsp.channels(channels)
rate = dsp.rate(rate)
```

oss_audio_device.**bufsize**()
返回硬件缓冲区的大小，以采样数表示。

oss_audio_device.**obufcount**()
返回硬件缓冲区中待播放的采样数。

oss_audio_device.**obuffree**()
返回可以被加入硬件缓冲区队列以非阻塞模式播放的采样数。

音频设备对象还支持几个只读属性:

oss_audio_device.**closed**
指明设备是否已被关闭的布尔值。

oss_audio_device.**name**
包含设备文件名称的字符串。

oss_audio_device.**mode**
文件的 I/O 模式，可以为 "r", "rw" 或 "w"。

36.16.2 混音器设备对象

混音器对象提供了两个文件类方法:

oss_mixer_device.**close**()
此方法会关闭打开的混音器设备文件。在文件被关闭后任何继续使用混音器的尝试都将引发*OSError*。

oss_mixer_device.**fileno**()
返回打开的混音器设备文件的文件处理句柄号。

3.2 版更變: 混音器设备还支持上下文管理协议。

其余方法都是混音专属的:

oss_mixer_device.**controls**()
此方法返回一个表示可用的混音控件的位掩码 (" 控件" 是专用的可混合" 声道"，例如 SOUND_MIXER_PCM 或 SOUND_MIXER_SYNTH)。该掩码会指定所有可用混音控件的一个子集 --- 它们是在模块层级上定义的 SOUND_MIXER_* 常量。举例来说，要确定当前混音器对象是否支持 PCM 混音器，就使用以下 Python 代码:

```
mixer=ossaudiodev.openmixer()
if mixer.controls() & (1 << ossaudiodev.SOUND_MIXER_PCM):
    # PCM is supported
    ... code ...
```

对于大多数目的来说，SOUND_MIXER_VOLUME (主音量) 和 SOUND_MIXER_PCM 控件应该足够了 --- 但使用混音器的代码应当在选择混音器控件时保持灵活。例如在 Gravis Ultrasound 上，SOUND_MIXER_VOLUME 是不存在的。

oss_mixer_device.**stereocontrols**()
返回一个表示立体声混音控件的位掩码。如果设置了比特位，则对应的控件就是立体声的；如果未设置，则控件为单声道或者不被混音器所支持（请配合*controls()* 使用以确定是哪种情况）。

请查看*controls()* 函数的代码示例了解如何从位掩码获取数据。

oss_mixer_device.**reccontrols**()
返回一个指明可被用于录音的混音器控件的位掩码。请查看*controls()* 的代码示例了解如何读取位掩码。

oss_mixer_device.**get**(*control*)
返回给定混音控件的音量。返回的音量是一个 2 元组 (left_volume,right_volume)。音量被表示为从 0 (静音) 到 100 (最大音量) 的数字。如果控件是单声道的，仍然会返回一个 2 元组，但两个音量必定相同。

如果指定了无效的控件则会引发*OSSAudioError*，或者如果指定了不受支持的控件则会引发*OSError*。

oss_mixer_device.**set**(*control*, *(left*, *right)*)
将给定混音控件的音量设为 (left,right)。left 和 right 必须为整数并在 0 (静音) 至 100 (最大音量) 之间。当执行成功的，新的音量将以 2 元组形式返回。请注意这可能不完全等于所指定的音量，因为某些声卡的混音器有精度限制。

如果指定了无效的混音控件，或者指定的音量超出限制则会引发*OSSAudioError*。

oss_mixer_device.**get_recsrc**()
此方法返回一个表示当前被用作录音源的的控件的位掩码。

oss_mixer_device.**set_recsrc**(*bitmask*)
调用此函数来指定一个录音源。如果成功则返回一个指明新录音源的位掩码；如果指定了无效的源则会引发*OSError*。如果要将当前录音源设为麦克风输入:

```
mixer.setrecsrc (1 << ossaudiodev.SOUND_MIXER_MIC)
```

36.17 pipes --- 终端管道接口

源代码： Lib/pipes.py

3.11 版後已🄵用: The *pipes* module is deprecated (see **PEP 594** for details). Please use the *subprocess* module instead.

---

pipes 定义了一个类用来抽象 *pipeline* 的概念 --- 将数据从一个文件转到另一文件的转换器序列。

由于模块使用了 **/bin/sh** 命令行，因此要求有 POSIX 或兼容*os.system()* 和*os.popen()* 的终端程序。

pipes 模块定义了以下的类:

class pipes.**Template**
对管道的抽象。

示例:

```
>>> import pipes
>>> t = pipes.Template()
>>> t.append('tr a-z A-Z', '--')
>>> f = t.open('pipefile', 'w')
>>> f.write('hello world')
>>> f.close()
>>> open('pipefile').read()
'HELLO WORLD'
```

36.17.1 模板对象

模板对象有以下方法:

Template.**reset**()
将一个管道模板恢复为初始状态。

Template.**clone**()
返回一个新的等价的管道模板。

Template.**debug**(*flag*)
如果 *flag* 为真值，则启用调试。否则禁用调试。当启用调试时，要执行的命令会被打印出来，并且会给予终端 `set -x` 命令以输出更详细的信息。

Template.**append**(*cmd*, *kind*)
在末尾添加一个新的动作。*cmd* 变量必须为一个有效的 bourne 终端命令。*kind* 变量由两个字母组成。

第一个字母可以为 `'-'` (这表示命令将读取其标准输入), `'f'` (这表示命令将读取在命令行中给定的文件) 或 `'.'` (这表示命令将不读取输入，因而必须放在前面。)

类似地，第二个字母可以为 `'-'` (这表示命令将写入到标准输出), `'f'` (这表示命令将写入在命令行中给定的文件) 或 `'.'` (这表示命令将不执行写入，因而必须放在末尾。)

Template.**prepend**(*cmd*, *kind*)
在开头添加一个新的动作。请参阅*append()* 获取相应参数的说明。

Template.**open**(*file*, *mode*)
返回一个文件类对象，打开到 *file*，但是将从管道读取或写入。请注意只能给出 `'r'`, `'w'` 中的一个。

Template.**copy**(*infile*, *outfile*)
通过管道将 *infile* 拷贝到 *outfile*。

36.18 smtpd --- SMTP 服务器

源代码: Lib/smtpd.py

该模块提供了几个类来实现 SMTP （电子邮件）服务器。

3.6 版後已🄵用: *smtpd* will be removed in Python 3.12 (see **PEP 594** for details). The aiosmtpd package is a recommended replacement for this module. It is based on *asyncio* and provides a more straightforward API.

有几个服务器的实现；一个是通用的无为实现，可以被重写，而另外两个则提供特定的邮件发送策略。

此外，SMTPChannel 可以被扩展以实现与 SMTP 客户端非常具体的交互行为。

该代码支持 **RFC 5321**，加上 **RFC 1870** SIZE 和 **RFC 6531** SMTPUTF8 扩展。

36.18.1 SMTPServer 对象

class smtpd.**SMTPServer**(*localaddr*, *remoteaddr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*)

新建一个*SMTPServer* 对象，它会绑定到本机地址 *localaddr*。它将把 *remoteaddr* 当作上游 SMTP 中继器。*localaddr* 和 *remoteaddr* 都应当是*(host, port)* 元组。该对象继承自*asyncore.dispatcher*，因而会在实例化时将自己插入到*asyncore* 的事件循环。

data_size_limit 指定将在 DATA 命令中被接受的最大字节数。值为 None 或 0 表示无限制。

map 是用于连接的套接字映射（初始为空的字典是适当的值）。如果未指定则会使用*asyncore* 全局套接字映射。

enable_SMTPUTF8 决定是否应当启用 SMTPUTF8 扩展（如 **RFC 6531** 所定义的。默认值为 False。当设为 True 时，会接受 SMTPUTF8 作为 MAIL 命令的形参并在被提供时将其传给 kwargs['mail_options'] 列表中的*process_message()*。*decode_data* 和 *enable_SMTPUTF8* 不可同时被设为 True。

decode_data 指明 SMTP 事务的数据部分是否应当使用 UTF-8 来解码。当 *decode_data* 为 False 时（默认值），服务器会声明 8BITMIME 扩展 (**RFC 6152**)，接受来自 MAIL 命令的 BODY=8BITMIME 形参，并在该形参存在时将其传给 kwargs['mail_options'] 列表中的*process_message()* 方法。*decode_data* 和 *enable_SMTPUTF8* 不可同时被设为 True。

process_message(*peer*, *mailfrom*, *rcpttos*, *data*, ***kwargs*)

引发*NotImplementedError* 异常。请在子类中重载此方法以实际运用此消息。在构造器中作为 *remoteaddr* 传入的任何东西都可以 _remoteaddr 属性的形式来访问。*peer* 是远程主机的地址，*mailfrom* 是封包的发送方，*rcpttos* 是封包的接收方而 *data* 是包含电子邮件内容的字符串（应该为 **RFC 5321** 格式）。

如果构造器关键字参数 *decode_data* 被设为 True，则 *data* 参数将为 Unicode 字符串。如果被设为 False，则将为字节串对象。

kwargs 是包含附加信息的字典。如果给出 decode_data=True 作为初始参数则该字典为空，否则它会包含以下的键:

***mail_options*:** 由 MAIL 命令所接收的所有参数组成的列表 (其元素为大写形式的字符串；例如: ['BODY=8BITMIME', 'SMTPUTF8'])。

***rcpt_options*:** 与 *mail_options* 类似但是针对 RCPT 命令。目前不支持任何 RCPT TO 选项，因此其值将总是为空列表。

process_message 的实现应当使用 **kwargs 签名来接收任意关键字参数，因为未来的增强特性可能会向 kwargs 字典添加新键。

返回 None 以请求一个正常的 250 Ok 响应；在其他情况下则以 **RFC 5321** 格式返回所需的响应字符串。

channel_class

重载这个子类以使用自定义的*SMTPChannel* 来管理 SMTP 客户端。

3.4 版新加入: *map* 构造器参数。

3.5 版更變: *localaddr* 和 *remoteaddr* 现在可以包含 IPv6 地址。

3.5 版新加入: *decode_data* 和 *enable_SMTPUTF8* 构造器形参，以及当 *decode_data* 为 False 时传给*process_message()* 的 *kwargs* 形参。

3.6 版更變: *decode_data* 现在默认为 False。

36.18.2 DebuggingServer 对象

class smtpd.**DebuggingServer**(*localaddr*, *remoteaddr*)
创建一个新的调试服务器。参数是针对每个*SMTPServer*。消息将被丢弃，并在 stdout 上打印出来。

36.18.3 PureProxy 对象

class smtpd.**PureProxy**(*localaddr*, *remoteaddr*)
创建一个新的纯代理服务器。参数是针对每个*SMTPServer*。一切都将被转发到 *remoteaddr*。请注意运行此对象有很大的机会令你成为一个开放的中继站，所以需要小心。

36.18.4 MailmanProxy 对象

class smtpd.**MailmanProxy**(*localaddr*, *remoteaddr*)
Deprecated since version 3.9, will be removed in version 3.11: *MailmanProxy* 已被弃用，它依赖于一个已不存在的 Mailman 模块因而本来就不再可用了。

创建一个新的纯代理服务器。参数是针对每个*SMTPServer*。一切都将被转发到 *remoteaddr*，除非本地 mailman 配置知道另一个地址，在那种情况下它将由 mailman 来处理。请注意运行此对象有很大的机会令你成为一个开放的中继站，所以需要小心。

36.18.5 SMTPChannel 对象

class smtpd.**SMTPChannel**(*server*, *conn*, *addr*, *data_size_limit=33554432*, *map=None*, *enable_SMTPUTF8=False*, *decode_data=False*)
创建一个新的*SMTPChannel* 对象，该对象会管理服务器和单个 SMTP 客户端之间的通信。

conn 和 *addr* 是针对下述的每个实例变量。

data_size_limit 指定将在 DATA 命令中被接受的最大字节数。值为 None 或 0 表示无限制。

enable_SMTPUTF8 确定 SMTPUTF8 扩展 (如 **RFC 6531** 所定义的) 是否应当被启用。默认值为 False。*decode_data* 和 *enable_SMTPUTF8* 不能被同时设为 True。

可以在 *map* 中指定一个字典以避免使用全局套接字映射。

decode_data 指明 SMTP 事务的数据部分是否应当使用 UTF-8 来解码。默认值为 False。*decode_data* 和 *enable_SMTPUTF8* 不能被同时设为 True。

要使用自定义的 SMTPChannel 实现你必须重载你的*SMTPServer* 的*SMTPServer.channel_class*。

3.5 版更變: 添加了 *decode_data* 和 *enable_SMTPUTF8* 形参。

3.6 版更變: *decode_data* 现在默认为 False。

SMTPChannel 具有下列实例变量:

smtp_server
存放生成此通道的*SMTPServer*。

conn
存放连接到客户端的套接字对象。

addr
存放客户端的地址，*socket.accept* 所返回的第二个值。

received_lines
存放从客户端接收的行字符串列表 (使用 UTF-8 解码)。所有行的 "\r\n" 行结束符都会被转写为 "\n"。

smtp_state
存放通道的当前状态。其初始值将为 COMMAND 而在客户端发送”DATA” 行后将为 DATA。

seen_greeting
存放包含客户端在其”HELO” 中发送的问候信息的字符串。

mailfrom
存放包含客户端在”MAIL FROM:” 中标识的地址的字符串。

rcpttos
存放包含客户端在”RCPT TO:” 行中标识的地址的字符串。

received_data
存放客户端在 DATA 状态期间发送的所有数据的字符串，直至但不包括末尾的 "\r\n.\r\n"。

fqdn
存放由*socket.getfqdn()* 所返回的服务器完整限定域名。

peer
存放由 conn.getpeername() 所返回的客户端对等方名称，其中 conn 为*conn*。

SMTPChannel 在接收到来自客户端的命令行时会通过发起调用名为 smtp_<command> 的方法来进行操作。在基类*SMTPChannel* 中具有用于处理下列命令（并对他们作出适当反应）的方法:

命令	所采取的行动
HELO	接受来自客户端的问候语，并将其存储在*seen_greeting* 中。将服务器设置为基本命令模式。
EHLO	接受来自客户的问候并将其存储在*seen_greeting* 中。将服务器设置为扩展命令模式。
NOOP	不采取任何措施。
QUIT	干净地关闭连接。
MAIL	接受”MAIL FROM:” 句法并将所提供的地址保存为*mailfrom*。在扩展命令模式下，还接受 **RFC 1870** SIZE 属性并根据 *data_size_limit* 的值作出适当返回。
RCPT	接受”RCPT TO:” 句法并将所提供的地址保存在*rcpttos* 列表中。
RSET	重置*mailfrom*, *rcpttos*, 和*received_data* ，但不重置问候语。
DATA	将内部状态设为 DATA 并将来自客户端的剩余行保存在*received_data* 中直至接收到终止符 "\r\n.\r\n"。
HELP	返回有关命令语法的最少信息
VRFY	返回代码 252（服务器不知道该地址是否有效）
EXPN	报告该命令未实现。

36.19 sndhdr --- 推测声音文件的类型

源代码 Lib/sndhdr.py

3.11 版後已🄵用: The *sndhdr* module is deprecated (see **PEP 594** for details and alternatives).

---

sndhdr 提供了企图猜测文件中的声音数据类型的功能函数。当这些函数可以推测出存储在文件中的声音数据的类型是，它们返回一个*collections.namedtuple()*，包含了五种属性:（filetype, framerate, nchannels, nframes, sampwidth）。这些 *type* 的值表示数据的类型，会是以下字符串之一: 'aifc',

'aiff', 'au', 'hcom', 'sndr', 'sndt', 'voc', 'wav', '8svx', 'sb', 'ub', or 'ul' 。*sampling_rate* 可能是实际值或者当未知或者难以解码时的 0。类似的，*channels* 也会返回实际值或者在无法推测或者难以解码时返回 0。*frames* 则是实际值或 -1。元组的最后一项，*bits_per_sample* 将会为比特表示的 sample 大小或者 A-LAW 时为 'A'，u-LAW 时为 'U'。

sndhdr.**what**(*filename*)
使用*whathdr()* 推测存储在 *filename* 文件中的声音数据的类型。如果成功，返回上述的命名元组，否则返回 None。

3.5 版更變: 将结果从元组改为命名元组。

sndhdr.**whathdr**(*filename*)
基于文件头推测存储在文件中的声音数据类型。文件名由 *filename* 给出。这个函数在成功时返回上述命名元组，或者在失败时返回 None。

3.5 版更變: 将结果从元组改为命名元组。

36.20 spwd ——shadow 密码库

3.11 版後已𝔼用: The *spwd* module is deprecated (see **PEP 594** for details and alternatives).

该模块提供对 Unix shadow 密码库的访问能力。可用于各种 Unix 版本。

访问 shadow 密码数据库须拥有足够的权限（通常意味着必须采用 root 账户）。

shadow 密码库的每条数据均表示为一个类似元组的对象，其属性对应着 “spwd“ 结构的成员（下面列出了各属性字段，参见‘<shadow.h>）。

索引	属性	意义
0	sp_namp	登录名
1	sp_pwdp	加密后的密码
2	sp_lstchg	最后修改日期
3	sp_min	两次修改间隔的最小天数
4	sp_max	两次修改间隔的最大天数
5	sp_warn	提前警告用户密码过期的天数
6	sp_inact	密码过期至账户禁用之间的天数
7	sp_expire	账户过期的天数，自 1970-01-01 算起
8	sp_flag	保留字段

sp_namp 和 sp_pwdp 条目是字符串，其他的均为整数。如果未找到所需条目则会触发*KeyError*。

定义了以下函数:

spwd.**getspnam**(*name*)
返回指定用户名的 shadow 密码库记录。

3.6 版更變: 如果当前用户权限不足，会触发*PermissionError*，而非*KeyError*。

spwd.**getspall**()
返回所有可用的 shadow 密码库记录列表，顺序随机。

也参考:

模块*grp* 针对用户组数据库的接口，与本模块类似。

模块*pwd* 访问普通密码库的接口，与本模块类似。

36.21 sunau --- 读写 Sun AU 文件

源代码: Lib/sunau.py

3.11 版後已𝔽用: The *sunau* module is deprecated (see **PEP 594** for details).

---

sunau 模拟提供了一个处理 Sun AU 声音格式的便利接口。请注意此模块与*aifc* 和*wave* 是兼容接口的。

音频文件由标头和数据组成。标头的字段为:

域	内容
magic word	四个字节 `.snd`
header size	标头的大小，包括信息，以字节为单位。
data size	数据的物理大小，以字节为单位。
编码	指示音频样本的编码方式。
sample rate	采样率
# of channels	采样中的通道数。
info	提供音频文件描述的 ASCII 字符串（用空字节填充）。

除了 info 字段，所有标头字段的大小都是 4 字节。它们都是采用大端字节序编码的 32 位无符号整数。

sunau 模块定义了以下函数:

sunau.**open**(*file*, *mode*)
如果 *file* 是一个字符串，打开相应名称的文件，否则就把它作为可定位的文件类对象来处理。*mode* 可以是

'r' 只读模式。

'w' 只写模式。

注意它不支持同时读/写文件。

mode 为 `'r'` 时返回一个 `AU_read` 对象，而 *mode* 为 `'w'` 或 `'wb'` 时返回一个 `AU_write` 对象。

sunau 模块定义了以下异常:

exception sunau.**Error**
当 Sun AU 规范或实现的低效导致无法操作时引发的错误。

sunau 模块定义了以下数据条目:

sunau.**AUDIO_FILE_MAGIC**
位于每个有效的 Sun AU 文件开头的整数，以大端序形式存储。这是一个被当作整数来解读的字符串 `.snd`。

sunau.**AUDIO_FILE_ENCODING_MULAW_8**
sunau.**AUDIO_FILE_ENCODING_LINEAR_8**
sunau.**AUDIO_FILE_ENCODING_LINEAR_16**
sunau.**AUDIO_FILE_ENCODING_LINEAR_24**
sunau.**AUDIO_FILE_ENCODING_LINEAR_32**
sunau.**AUDIO_FILE_ENCODING_ALAW_8**
AU 标头中被此模块所支持的 encoding 字段值。

sunau.**AUDIO_FILE_ENCODING_FLOAT**
sunau.**AUDIO_FILE_ENCODING_DOUBLE**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G721**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G722**

sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_3**
sunau.**AUDIO_FILE_ENCODING_ADPCM_G723_5**
AU 标头中附加的已知但不被此模块所支持的 encoding 字段值。

36.21.1 AU_read 对象

由上面的*open()* 所返回的 AU_read 对象具有以下几种方法:

AU_read.**close**()
关闭流，并使实例不可用。(此方法会在删除对象时自动调用。)

AU_read.**getnchannels**()
返回音频的通道数（单声道为 1，立体声为 2)。

AU_read.**getsampwidth**()
返回采样字节长度。

AU_read.**getframerate**()
返回采样频率。

AU_read.**getnframes**()
返回音频总帧数。

AU_read.**getcomptype**()
返回压缩类型。受支持的压缩类型有 'ULAW', 'ALAW' 和 'NONE'。

AU_read.**getcompname**()
getcomptype() 的人类可读的版本。受支持的类型将为相应的名称 'CCITT G.711 u-law', 'CCITT G.711 A-law' 和 'not compressed'。

AU_read.**getparams**()
返回一个*namedtuple()* (nchannels, sampwidth, framerate, nframes, comptype, compname)，与 get*() 方法的输出相同。

AU_read.**readframes**(*n*)
读取至多 *n* 帧音频并作为*bytes* 对象返回。数据将以线性格式返回。如果原始数据为 u-LAW 格式，则它将被转换。

AU_read.**rewind**()
设置当前文件指针位置。

以下两个方法都使用指针，具体实现由其底层决定。

AU_read.**setpos**(*pos*)
设置文件指针到特定位置。只有从*tell()* 返回的值才可被用作 *pos*。

AU_read.**tell**()
返回当前文件指针的位置。请注意该返回值与文件中的实例位置无关。

以下两个函数是为了与*aifc* 保持兼容而定义的，实际不做任何事件。

AU_read.**getmarkers**()
返回 None。

AU_read.**getmark**(*id*)
引发错误异常。

36.21.2 AU_write 对象

由上面的*open()* 所返回的 AU_write 对象具有以下几种方法:

AU_write.**setnchannels**(*n*)
设置声道数。

AU_write.**setsampwidth**(*n*)
设置采样宽度（字节长度。）

3.4 版更變: 增加了对 24 位采样的支持。

AU_write.**setframerate**(*n*)
设置帧速率。

AU_write.**setnframes**(*n*)
设置总帧数。如果写入了更多的帧，此值将会被更改。

AU_write.**setcomptype**(*type*, *name*)
设置压缩类型和描述。对于输出只支持 `'NONE'` 和 `'ULAW'`。

AU_write.**setparams**(*tuple*)
tuple 应该是 `(nchannels, sampwidth, framerate, nframes, comptype, compname)`，每项的值可用于 `set*()` 方法。设置所有形参。

AU_write.**tell**()
返回文件中的当前位置，其含义与*AU_read.tell()* 和*AU_read.setpos()* 方法的一致。

AU_write.**writeframesraw**(*data*)
写入音频数据但不更新 *nframes*。

3.4 版更變: 现在可接受任意*bytes-like object*。

AU_write.**writeframes**(*data*)
写入音频数据并更新 *nframes*。

3.4 版更變: 现在可接受任意*bytes-like object*。

AU_write.**close**()
确保 *nframes* 正确，并关闭文件。

此方法会在删除时被调用。

请注意在调用 `writeframes()` 和 `writeframesraw()` 之后再设置任何形参都是无效的。

36.22 `telnetlib` -- Telnet 客户端

源代码： Lib/telnetlib.py

3.11 版後已🄴用: The *telnetlib* module is deprecated (see **PEP 594** for details and alternatives).

---

telnetlib 模块提供一个实现 Telnet 协议的类*Telnet*。关于此协议的细节请参见 **RFC 854** 。此外，它还为协议字符（见下文）和 telnet 选项提供了对应的符号常量。telnet 选项对应的符号名遵循 `arpa/telnet.h` 中的定义，但删除了前缀 `TELOPT_`。对于不在 `arpa/telnet.h` 的选项的符号常量名，请参考本模块源码。

telnet 命令的符号常量名有：IAC, DONT, DO, WONT, WILL, SE (Subnegotiation End), NOP (No Operation), DM (Data Mark), BRK (Break), IP (Interrupt process), AO (Abort output), AYT (Are You There), EC (Erase Character), EL (Erase Line), GA (Go Ahead), SB (Subnegotiation Begin).

class telnetlib.**Telnet**(*host=None*, *port=0*[, *timeout*])

Telnet 表示到 Telnet 服务器的连接. 实例初始化后默认不连接；必须使用*open()* 方法来建立连接。或者，可选参数 host 和 port 也可以传递给构造函数，在这种情况下，到服务器的连接将在构造函数返回前建立。可选参数 *timeout* 为阻塞操作（如连接尝试）指定一个以秒为单位的超时时间（如果没有指定，将使用全局默认设置）。

不要重新打开一个已经连接的实例。

这个类有很多 read_*() 方法。请注意，其中一些方法在读取结束时会触发*EOFError* 异常，这是由于连接对象可能出于其它原因返回一个空字符串。请参阅下面的个别描述。

Telnet 对象一个上下文管理器，可以在 with 语句中使用。当 with 块结束，*close()* 方法会被调用:

```
>>> from telnetlib import Telnet
>>> with Telnet('localhost', 23) as tn:
...     tn.interact()
...
```

3.6 版更變: 添加了上下文管理器的支持

也参考:

RFC 854 - Telnet 协议规范 Telnet 协议的定义。

36.22.1 Telnet 对象

Telnet 实例有以下几种方法:

Telnet.**read_until**(*expected*, *timeout=None*)

读取直到遇到给定字节串 *expected* 或 *timeout* 秒已经过去。

当没有找到匹配时，返回可用的内容，也可能返回空字节。如果连接已关闭且没有可用的熟数据，将触发*EOFError*。

Telnet.**read_all**()

读取数据，直到遇到 EOF；连接关闭前都会保持阻塞。

Telnet.**read_some**()

在达到 EOF 前，读取至少一个字节的熟数据。如果命中 EOF，返回 b''。如果没有立即可用的数据，则阻塞。

Telnet.**read_very_eager**()

在不阻塞 I/O 的情况下读取所有的内容（eager）。

如果连接已关闭并且没有可用的熟数据，将会触发*EOFError* 。如果没有熟数据可用返回 b'' 。除非在一个 IAC 序列的中间，否则不要进行阻塞。

Telnet.**read_eager**()

读取现成的数据。

如果连接已关闭并且没有可用的熟数据，将会触发*EOFError* 。如果没有熟数据可用返回 b'' 。除非在一个 IAC 序列的中间，否则不要进行阻塞。

Telnet.**read_lazy**()

处理并返回已经在队列中的数据（lazy）。

如果连接已关闭并且没有可用的数据，将会触发*EOFError* 。如果没有熟数据可用则返回 b'' 。除非在一个 IAC 序列的中间，否则不要进行阻塞。

Telnet.**read_very_lazy**()
返回熟数据队列任何可用的数据（very lazy）。

如果连接已关闭并且没有可用的数据，将会触发*EOFError*。如果没有熟数据可用则返回 b'' 。该方法永远不会阻塞。

Telnet.**read_sb_data**()
返回在 SB/SE 对之间收集的数据（子选项 begin/end）。当使用 SE 命令调用回调函数时，该回调函数应该访问这些数据。该方法永远不会阻塞。

Telnet.**open**(*host*, *port=0*[, *timeout*])
连接主机。第二个可选参数是端口号，默认为标准 Telnet 端口（23）。可选参数 *timeout* 指定一个以秒为单位的超时时间用于像连接尝试这样的阻塞操作（如果没有指定，将使用全局默认超时设置）。

不要尝试重新打开一个已经连接的实例。

触发*auditing event* telnetlib.Telnet.open，参数为 self，host，port。

Telnet.**msg**(*msg*, **args*)
当调试级别 > 0 时打印一条调试信息。如果存在额外参数，则它们会被替换在使用标准字符串格式化操作符的信息中。

Telnet.**set_debuglevel**(*debuglevel*)
设置调试级别。*debuglevel* 的值越高，得到的调试输出越多（在 sys.stdout ）。

Telnet.**close**()
关闭连接对象。

Telnet.**get_socket**()
返回内部使用的套接字对象。

Telnet.**fileno**()
返回内部使用的套接字对象的文件描述符。

Telnet.**write**(*buffer*)
向套接字写入一个字节字符串，将所有 IAC 字符加倍。如果连接被阻塞，这可能也会阻塞。如果连接关闭可能触发*OSError*。

触发*auditing event* telnetlib.Telnet.write，参数为 self，buffer。

3.3 版更變: 曾经该函数抛出*socket.error*，现在这是*OSError* 的别名。

Telnet.**interact**()
交互函数，模拟一个非常笨拙的 Telnet 客户端。

Telnet.**mt_interact**()
多线程版的*interact()*.

Telnet.**expect**(*list*, *timeout=None*)
一直读取，直到匹配列表中的某个正则表达式。

第一个参数是一个正则表达式列表，可以是已编译的 (*正则表达式对象*)，也可以是未编译的 (字节串)。第二个可选参数是超时，单位是秒；默认一直阻塞。

返回一个包含三个元素的元组：列表中的第一个匹配的正则表达式的索引；返回的匹配对象；包括匹配在内的读取过的字节。

如果找到了文件的结尾且没有字节被读取，触发*EOFError*。否则，当没有匹配时，返回 (-1, None, data)，其中 *data* 是到目前为止接受到的字节（如果发生超时，则可能是空字节）。

如果一个正则表达式以贪婪匹配结束（例如 .*），或者多个表达式可以匹配同一个输出，则结果是不确定的，可能取决于 I/O 计时。

Telnet.**set_option_negotiation_callback**(*callback*)
每次在输入流上读取 telnet 选项时，这个带有如下参数的 *callback*（如果设置了）会被调用：callback(telnet socket, command (DO/DONT/WILL/WONT), option)。telnetlib 之后不会再执行其它操作。

36.22.2 Telnet 示例

一个简单的说明性典型用法例子:

```
import getpass
import telnetlib

HOST = "localhost"
user = input("Enter your remote account: ")
password = getpass.getpass()

tn = telnetlib.Telnet(HOST)

tn.read_until(b"login: ")
tn.write(user.encode('ascii') + b"\n")
if password:
    tn.read_until(b"Password: ")
    tn.write(password.encode('ascii') + b"\n")

tn.write(b"ls\n")
tn.write(b"exit\n")

print(tn.read_all().decode('ascii'))
```

36.23 uu --- 对 uuencode 文件进行编码与解码

源代码: Lib/uu.py

3.11 版後已🄴用: The *uu* module is deprecated (see **PEP 594** for details). *base64* is a modern alternative.

此模块使用 uuencode 格式来编码和解码文件，以便任意二进制数据可通过仅限 ASCII 码的连接进行传输。在任何要求文件参数的地方，这些方法都接受文件类对象。为了保持向下兼容，也接受包含路径名称的字符串，并且将打开相应的文件进行读写；路径名称 `'-'` 被解读为标准输入或输出。但是，此接口已被弃用；在 Windows 中调用者最好是自行打开文件，并在需要时确保模式为 `'rb'` or `'wb'`。

此代码由 Lance Ellinghouse 贡献，并由 Jack Jansen 修改。

uu 模块定义了以下函数:

uu.**encode**(*in_file*, *out_file*, *name=None*, *mode=None*, *, *backtick=False*)
使用 uuencode 将 *in_file* 文件编码为 *out_file* 文件。经过 uuencoded 编码的文件将具有指定 *name* 和 *mode* 作为解码该文件默认结果的标头。默认值会相应地从 *in_file* 或 `'-'` 以及 `0o666` 中提取。如果 *backtick* 为真值，零会用 `'`` '` 而不是空格来表示。

3.7 版更變: 增加 *backtick* 参数

uu.**decode**(*in_file*, *out_file=None*, *mode=None*, *quiet=False*)
调用此函数会解码 uuencod 编码的 *in_file* 文件并将结果放入 *out_file* 文件。如果 *out_file* 是一个路径名称，*mode* 会在必须创建文件时用于设置权限位。*out_file* 和 *mode* 的默认值会从 uuencode 标头中提取。但是，如果标头中指定的文件已存在，则会引发*uu.Error*。

如果输入由不正确的 uuencode 编码器生成，*decode()* 可能会打印一条警告到标准错误，这样 Python 可以从该错误中恢复。将 *quiet* 设为真值可以屏蔽此警告。

exception uu.**Error**
Exception 的子类，此异常可由*uu.decode()* 在多种情况下引发，如上文所述，此外还包括格式错误的标头或被截断的输入文件等。

也参考:

模块*binascii* 支持模块，包含 ASCII 到二进制和二进制到 ASCII 转换。

36.24 xdrlib --- 编码与解码 XDR 数据

源代码: Lib/xdrlib.py

3.11 版後已F用: The *xdrlib* module is deprecated (see **PEP 594** for details).

---

xdrlib 模块为外部数据表示标准提供支持，该标准的描述见 **RFC 1014**，由 Sun Microsystems, Inc. 在 1987 年 6 月撰写。它支持该 RFC 中描述的大部分数据类型。

xdrlib 模块定义了两个类，一个用于将变量打包为 XDR 表示形式，另一个用于从 XDR 表示形式解包。此外还有两个异常类。

class xdrlib.**Packer**
Packer 是用于将数据打包为 XDR 表示形式的类。*Packer* 类的实例化不附带参数。

class xdrlib.**Unpacker**(*data*)
Unpacker 是用于相应地从字符串缓冲区解包 XDR 数据值的类。输入缓冲区将作为 *data* 给出。

也参考:

RFC 1014 - XDR: 外部数据表示标准 这个 RFC 定义了最初编写此模块时 XDR 所用的数据编码格式。显然它已被 **RFC 1832** 所淘汰。

RFC 1832 - XDR: 外部数据表示标准 更新的 RFC，它提供了经修订的 XDR 定义。

36.24.1 Packer 对象

Packer 实例具有下列方法:

Packer.**get_buffer**()
将当前打包缓冲区以字符串的形式返回。

Packer.**reset**()
将打包缓冲区重置为空字符串。

总体来说，你可以通过调用适当的 pack_type() 方法来打包任何最常见的 XDR 数据类型。每个方法都是接受单个参数，即要打包的值。受支持的简单数据类型打包方法如下: pack_uint(), pack_int(), pack_enum(), pack_bool(), pack_uhyper() 以及 pack_hyper()。

Packer.**pack_float**(*value*)
打包单精度浮点数 *value*。

Packer.**pack_double**(*value*)
打包双精度浮点数 *value*。

以下方法支持打包字符串、字节串以及不透明数据。

Packer.**pack_fstring**(*n*, *s*)
打包固定长度字符串 *s*。*n* 为字符串的长度，但它 不会被打包进数据缓冲区。如有必要字符串会以空字节串填充以保证 4 字节对齐。

Packer.**pack_fopaque**(*n*, *data*)
打包固定长度不透明数据流，类似于*pack_fstring()*。

Packer.**pack_string**(*s*)
打包可变长度字符串 *s*。先将字符串的长度打包为无符号整数，再用*pack_fstring()* 来打包字符串数据。

Packer.**pack_opaque**(*data*)
打包可变长度不透明数据流，类似于*pack_string()*。

Packer.**pack_bytes**(*bytes*)
打包可变长度字节流，类似于*pack_string()*。

下列方法支持打包数组和列表:

Packer.**pack_list**(*list*, *pack_item*)
打包由同质条目构成的 *list*。此方法适用于不确定长度的列表；即其长度无法在遍历整个列表之前获知。对于列表中的每个条目，先打包一个无符号整数 `1`，再添加列表中数据的值。*pack_item* 是在打包单个条目时要调用的函数。在列表的末尾，会再打包一个无符号整数 `0`。

例如，要打包一个整数列表，代码看起来会是这样:

```
import xdrlib
p = xdrlib.Packer()
p.pack_list([1, 2, 3], p.pack_int)
```

Packer.**pack_farray**(*n*, *array*, *pack_item*)
打包由同质条目构成的固定长度列表 (*array*)。*n* 为列表长度；它 不会被打包到缓冲区，但是如果 `len(array)` 不等于 *n* 则会引发*ValueError*。如上所述，*pack_item* 是在打包每个元素时要使用的函数。

Packer.**pack_array**(*list*, *pack_item*)
打包由同质条目构成的可变长度 *list*。先将列表的长度打包为无符号整数，再像上面的*pack_farray()* 一样打包每个元素。

36.24.2 Unpacker 对象

Unpacker 类提供以下方法:

Unpacker.**reset**(*data*)
使用给定的 *data* 重置字符串缓冲区。

Unpacker.**get_position**()
返回数据缓冲区中的当前解包位置。

Unpacker.**set_position**(*position*)
将数据缓冲区的解包位置设为 *position*。你应当小心使用*get_position()* 和*set_position()*。

Unpacker.**get_buffer**()
将当前解包数据缓冲区以字符串的形式返回。

Unpacker.**done**()
表明解包完成。如果数据没有全部完成解包则会引发*Error* 异常。

此外，每种可通过*Packer* 打包的数据类型都可通过*Unpacker* 来解包。解包方法的形式为 `unpack_type()`，并且不接受任何参数。该方法将返回解包后的对象。

Unpacker.**unpack_float**()
解包单精度浮点数。

Unpacker.**unpack_double**()
解包双精度浮点数，类似于*unpack_float()*。

此外，以下方法可用来解包字符串、字节串以及不透明数据:

Unpacker.**unpack_fstring**(*n*)
解包并返回固定长度字符串。*n* 为期望的字符数量。会预设以空字节串填充以保证 4 字节对齐。

Unpacker.**unpack_fopaque**(*n*)
解包并返回固定长度数据流，类似于*unpack_fstring()*。

Unpacker.**unpack_string**()
解包并返回可变长度字符串。先将字符串的长度解包为无符号整数，再用*unpack_fstring()* 来解包字符串数据。

Unpacker.**unpack_opaque**()
解包并返回可变长度不透明数据流，类似于*unpack_string()*。

Unpacker.**unpack_bytes**()
解包并返回可变长度字节流，类似于*unpack_string()*。

下列方法支持解包数组和列表:

Unpacker.**unpack_list**(*unpack_item*)
解包并返回同质条目的列表。该列表每次解包一个元素，先解包一个无符号整数旗标。如果旗标为 1，则解包条目并将其添加到列表。旗标为 0 表明列表结束。*unpack_item* 为在解包条目时调用的函数。

Unpacker.**unpack_farray**(*n*, *unpack_item*)
解包并（以列表形式）返回由同质条目构成的固定长度数组。*n* 为期望的缓冲区内列表元素数量。如上所述，*unpack_item* 是解包每个元素时要使用的函数。

Unpacker.**unpack_array**(*unpack_item*)
解包并返回由同质条目构成的可变长度 *list*。先将列表的长度解包为无符号整数，再像上面的*unpack_farray()* 一样解包每个元素。

36.24.3 异常

此模块中的异常会表示为类实例代码:

exception xdrlib.**Error**
基本异常类。*Error* 具有一个公共属性 msg，其中包含对错误的描述。

exception xdrlib.**ConversionError**
从*Error* 所派生的类。不包含额外的实例变量。

以下是一个应该如何捕获这些异常的示例:

```
import xdrlib
p = xdrlib.Packer()
try:
    p.pack_double(8.01)
except xdrlib.ConversionError as instance:
    print('packing the double failed:', instance.msg)
```

CHAPTER 37

Security Considerations

The following modules have specific security considerations:

- `cgi`: *CGI security considerations*
- `hashlib`: *all constructors take a "usedforsecurity" keyword-only argument disabling known insecure and blocked algorithms*
- `http.server` is not suitable for production use, only implementing basic security checks. See the *security considerations*.
- `logging`: *Logging configuration uses eval()*
- `multiprocessing`: *Connection.recv() uses pickle*
- `pickle`: *Restricting globals in pickle*
- `random` shouldn't be used for security purposes, use `secrets` instead
- `shelve`: *shelve is based on pickle and thus unsuitable for dealing with untrusted sources*
- `ssl`: *SSL/TLS security considerations*
- `subprocess`: *Subprocess security considerations*
- `tempfile`: *mktemp is deprecated due to vulnerability to race conditions*
- `xml`: *XML vulnerabilities*
- `zipfile`: *maliciously prepared .zip files can cause disk volume exhaustion*

APPENDIX A

術語表

>>> 互動式 shell 的預設 Python 提示字元。常見於能在直譯器中以互動方式被執行的程式碼範例。

... 可以表示：

- 在一個被縮排的程式碼區塊、在一對匹配的左右定界符（delimiter，例如括號、方括號、花括號或三引號）🄵部，或是在指定一個裝飾器 (decorator) 之後，要輸入程式碼時，互動式 shell 顯示的預設 Python 提示字元。
- 🄵建常數*`Ellipsis`*。

2to3 一個試著將 Python 2.x 程式碼轉🄵🄵 Python 3.x 程式碼的工具，它是透過處理大部分的不相容性來達成此目的，而這些不相容性能🄵透過剖析原始碼和遍歷剖析樹而被檢測出來。

2to3 在可以標准函式庫中以*`lib2to3`* 被使用；它提供了一個獨立的入口點，在 `Tools/scripts/2to3`。請參🄵*2to3 - 自動將 Python 2 的程式碼轉成 Python 3*。

abstract base class（抽象基底類🄵） 抽象基底類🄵（又稱🄵 ABC）提供了一種定義介面的方法，作🄵*duck-typing*（鴨子型🄵）的補充。其他類似的技術，像是*`hasattr()`*，則顯得笨拙或是帶有細微的錯誤（例如使用魔術方法 (magic method)）。ABC 🄵用🄵擬的 subclass（子類🄵），它們🄵不繼承自另一個 class（類🄵），但仍可被*`isinstance()`* 及*`issubclass()`* 辨識；請參🄵*`abc`* 模組的🄵明文件。Python 有許多🄵建的 ABC，用於資料結構（在*`collections.abc`* 模組）、數字（在*`numbers`* 模組）、串流（在*`io`* 模組）及 import 尋檢器和載入器（在*`importlib.abc`* 模組）。你可以使用*`abc`* 模組建立自己的 ABC。

annotation（🄵釋） 一個與變數、class 屬性、函式的參數或回傳值相關聯的標🄵。照慣例，它被用來作🄵*type hint*（型🄵提示）。

在運行時 (runtime)，區域變數的🄵釋無法被存取，但全域變數、class 屬性和函式的🄵解，會分🄵被儲存在模組、class 和函式的 `__annotations__` 特殊屬性中。

請參🄵*variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，這些章節皆有此功能的🄵明。

argument（引數） 呼叫函式時被傳遞給*function*（或*method*）的值。引數有兩種：

- 關鍵字引數 *(keyword argument)*：在函式呼叫中，以識🄵字（identifier，例如 `name=`）開頭的引數，或是以 `**` 後面 dictionary（字典）🄵的值被傳遞的引數。例如，`3` 和 `5` 都是以下*`complex()`* 呼叫中的關鍵字引數：

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *位置引數 (positional argument)*：不是關鍵字引數的引數。位置引數可在一個引數列表的起始處出現，和（或）作🄵 `*` 之後的*iterable*（可🄴代物件）中的元素被傳遞。例如，`3` 和 `5` 都是以下呼叫中的位置引數：

```
complex(3, 5)
complex(*(3, 5))
```

引數會被指定給函式主體中的附名區域變數。關於支配這個指定過程的規則，請參🄵 calls 章節。在語法上，任何運算式都可以被用來表示一個引數；其評估值會被指定給區域變數。

另請參🄵術語表的*parameter*（參數）條目、常見問題中的引數和參數之間的差🄴，以及 **PEP 362**。

asynchronous context manager（非同步情境管理器） 一個可以控制 `async with` 陳述式中所見環境的物件，而它是透過定義 `__aenter__()` 和 `__aexit__()` method（方法）來控制的。由 **PEP 492** 引入。

asynchronous generator（非同步🄵生器） 一個會回傳*asynchronous generator iterator*（非同步🄵生器🄵代器）的函式。它看起來像一個以 `async def` 定義的協程函式 (coroutine function)，但不同的是它包含了 `yield` 運算式，能生成一系列可用於 `async for` 🄵圈的值。

這個術語通常用來表示一個非同步🄵生器函式，但在某些情境中，也可能是表示非同步🄵生器🄵代器 *(asynchronous generator iterator)*。萬一想表達的意思不🄴清楚，那就使用完整的術語，以避免歧義。

一個非同步🄵生器函式可能包含 `await` 運算式，以及 `async for` 和 `async with` 陳述式。

asynchronous generator iterator（非同步🄵生器🄵代器） 一個由*asynchronous generator*（非同步🄵生器）函式所建立的物件。

這是一個*asynchronous iterator*（非同步🄵代器），當它以 `__anext__()` method 被呼叫時，會回傳一個可等待物件 (awaitable object)，該物件將執行非同步🄵生器的函式主體，直到遇到下一個 `yield` 運算式。

每個 `yield` 會暫停處理程序，🄵記住位置執行狀態（包括區域變數及擱置中的 try 陳述式）。當非同步🄵生器🄵代器以另一個被 `__anext__()` 回傳的可等待物件有效地回復時，它會從停止的地方繼續執行。請參🄵 **PEP 492** 和 **PEP 525**。

asynchronous iterable（非同步可🄴代物件） 一個物件，它可以在 `async for` 陳述式中被使用。必須從它的 `__aiter__()` method 回傳一個*asynchronous iterator*（非同步🄵代器）。由 **PEP 492** 引入。

asynchronous iterator（非同步🄵代器） 一個實作 `__aiter__()` 和 `__anext__()` method 的物件。`__anext__` 必須回傳一個*awaitable*（可等待物件）。`async for` 會解析非同步🄵代器的 `__anext__()` method 所回傳的可等待物件，直到它引發 `StopAsyncIteration` 例外。由 **PEP 492** 引入。

attribute（屬性） 一個與某物件相關聯的值，該值能透過使用點分隔運算式 (dotted expression) 的名稱被參照。例如，如果物件 *o* 有一個屬性 *a*，則該屬性能以 *o.a* 被參照。

awaitable（可等待物件） 一個可以在 `await` 運算式中被使用的物件。它可以是一個*coroutine*（協程），或是一個有 `__await__()` method 的物件。另請參🄵 **PEP 492**。

BDFL Benevolent Dictator For Life（終身仁慈獨裁者），又名 Guido van Rossum，Python 的創造者。

binary file（二進制檔案） 一個能🄴讀取和寫入*bytes-like objects*（類位元組串物件）的*file object*（檔案物件）。二進制檔案的例子有：以二進制模式（`'rb'`、`'wb'` 或 `'rb+'`）開🄴的檔案、`sys.stdin.buffer`、`sys.stdout.buffer`，以及*io.BytesIO* 和*gzip.GzipFile* 實例。

另請參🄵*text file*（文字檔案），它是一個能🄴讀取和寫入*str* 物件的檔案物件。

bytes-like object（類位元組串物件） 一個支援 bufferobjects 且能🄴匯出 C-*contiguous* 緩衝區的物件。這包括所有的*bytes*、*bytearray* 和*array.array* 物件，以及許多常見的*memoryview* 物件。類位元組串

物件可用於處理二進制資料的各種運算；這些運算包括壓縮、儲存至二進制檔案和透過 socket（插座）發送。

有些運算需要二進制資料是可變的。F明文件通常會將這些物件稱F「可讀寫的類位元組串物件」。可變緩衝區的物件包括*bytearray*，以及*bytearray* 的*memoryview*。其他的運算需要讓二進制資料被儲存在不可變物件（「唯讀的類位元組串物件」）中；這些物件包括*bytes*，以及*bytes* 物件的*memoryview*。

bytecode（位元組碼） Python 的原始碼會被編譯成位元組碼，它是 Python 程式在 CPython 直譯器中的F部表示法。該位元組碼也會被暫存在 `.pyc` 檔案中，以便第二次執行同一個檔案時能F更快速（可以不用從原始碼重新編譯F位元組碼）。這種「中間語言 (intermediate language)」據F是運行在一個*virtual machine*（F擬機器）上，該F擬機器會執行與每個位元組碼對應的機器碼 (machine code)。要注意的是，位元組碼理論上是無法在不同的 Python F擬機器之間運作的，也不能在不同版本的 Python 之間保持穩定。

位元組碼的指令列表可以在*dis* 模組的F明文件中找到。

callback（回呼） 作F引數被傳遞的一個副程式 (subroutine) 函式，會在未來的某個時間點被執行。

class（類F） 一個用於建立使用者定義物件的模板。Class 的定義通常會包含 method 的定義，這些 method 可以在 class 的實例上進行操作。

class variable（類F變數） 一個在 class 中被定義，且應該只能在 class 層次（意即不是在 class 的實例中）被修改的變數。

coercion（F制轉型） 在涉及兩個不同型F引數的操作過程中，將某一種型F的實例F F另一種型F的隱式轉F (implicit conversion) 過程。例如，`int(3.15)` 會將浮點數轉F F整數 `3`，但在 `3+4.5` 中，每個引數是不同的型F（一個 int，一個 float），而這兩個引數必須在被轉F F相同的型F之後才能相加，否則就會引發*TypeError*。如果F有F制轉型，即使所有的引數型F皆相容，它們都必須要由程式設計師正規化 (normalize) F相同的值，例如，要用 `float(3)+4.5` 而不能只是 `3+4.5`。

complex number（F數） 一個我們熟悉的實數系統的擴充，在此所有數字都會被表示F一個實部和一個F部之和。F數就是F數單位（`-1` 的平方根）的實數倍，此單位通常在數學中被寫F `i`，在工程學中被寫F `j`。Python F建了對F數的支援，它是用後者的記法來表示F數；F部會帶著一個後綴的 `j` 被編寫，例如 `3+1j`。若要將*math* 模組F的工具等效地用於F數，請使用*cmath* 模組。F數的使用是一個相當進階的數學功能。如果你F有察覺到對它們的需求，那F幾乎能確定你可以安全地忽略它們。

context manager（情境管理器） 一個可以控制 `with` 陳述式中所見環境的物件，而它是透過定義 `__enter__()` 和 `__exit__()` method 來控制的。請參F **PEP 343**。

context variable（情境變數） 一個變數，其值可以根據上下文的情境而有所不同。這類似執行緒區域儲存區 (Thread-Local Storage)，在其中，一個變數在每個執行緒可能具有不同的值。然而，關於情境變數，在一個執行緒中可能會有多個情境，而情境變數的主要用途，是在F行的非同步任務 (concurrent asynchronous task) 中，對於變數狀態的追F。請參F*contextvars*。

contiguous（連續的） 如果一個緩衝區是 *C-contiguous* 或是 *Fortran contiguous*，則它會確切地被視F是連續的。零維 (zero-dimensional) 的緩衝區都是 C 及 Fortran contiguous。在一維 (one-dimensional) 陣列中，各項目必須在記憶體中彼此相鄰地排列，而其索引順序是從零開始遞增。在多維的 (multidimensional) C-contiguous 陣列中，按記憶體位址的順序訪問各個項目時，最後一個索引的變化最快。然而，在 Fortran contiguous 陣列中，第一個索引的變化最快。

coroutine（協程） 協程是副程式 (subroutine) 的一種更F廣義的形式。副程式是在某個時間點被進入F在另一個時間點被退出。協程可以在許多不同的時間點被進入、退出和回復。它們能F以 `async def` 陳述式被實作。另請參F **PEP 492**。

coroutine function（協程函式） 一個回傳*coroutine*（協程）物件的函式。一個協程函式能以 `async def` 陳述式被定義，F可能會包含 `await`、`async for` 和 `async with` 關鍵字。這些關鍵字由 **PEP 492** 引入。

CPython Python 程式語言的標準實作 (canonical implementation)，被發布在 python.org 上。「CPython」這個術語在必要時被使用，以區分此實作與其它語言的實作，例如 Jython 或 IronPython。

decorator（裝飾器） 一個函式，它會回傳另一個函式，通常它會使用 `@wrapper` 語法，被應用🄵一種函式的變🄵 (function transformation)。裝飾器的常見範例是*`classmethod()`* 和*`staticmethod()`*。

裝飾器語法只是語法糖。以下兩個函式定義在語義上是等效的：

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

Class 也存在相同的概念，但在那🄵比較不常用。關於裝飾器的更多🄵容，請參🄵函式定義和 class 定義的🄵明文件。

descriptor（描述器） 任何定義了 `__get__()`、`__set__()` 或 `__delete__()` method 的物件。當一個 class 屬性是一個描述器時，它的特殊連結行🄴會在屬性查找時被觸發。通常，使用 *a.b* 來取得、設定或🄵除某個屬性時，會在 *a* 的 class 字典中查找名稱🄵 *b* 的物件，但如果 *b* 是一個描述器，則相對應的描述器 method 會被呼叫。對描述器的理解是深入理解 Python 的關鍵，因🄵它們是許多功能的基礎，這些功能包括函式、method、屬性 (property)、class method、🄵態 method，以及對 super class（父類🄵）的參照。

關於描述器 method 的更多資訊，請參🄵 descriptors 或描述器使用指南。

dictionary（字典） 一個關聯陣列 (associative array)，其中任意的鍵會被映射到值。鍵可以是任何帶有 `__hash__()` 和 `__eq__()` method 的物件。在 Perl 中被稱🄵雜🄵 (hash)。

dictionary comprehension（字典綜合運算） 一種緊密的方法，用來處理一個可🄵代物件中的全部或部分元素，🄵將處理結果以一個字典回傳。`results = {n: n ** 2 for n in range(10)}` 會🄵生一個字典，它包含了鍵 `n` 映射到值 `n ** 2`。請參🄵 comprehensions。

dictionary view（字典檢視） 從*`dict.keys()`*、*`dict.values()`* 及*`dict.items()`* 回傳的物件被稱🄵字典檢視。它們提供了字典中項目的動態檢視，這表示當字典有變動時，該檢視會反映這些變動。若要🄵制將字典檢視轉🄴完整的 list（串列），須使用 `list(dictview)`。請參🄵字典视图对象。

docstring（🄵明字串） 一個在 class、函式或模組中，作🄵第一個運算式出現的字串文本。雖然它在套件執行時會被忽略，但它會被編譯器辨識，🄵被放入所屬 class、函式或模組的 `__doc__` 屬性中。由於🄵明字串可以透過🄵省 (introspection) 來🄵覽，因此它是物件的🄵明文件存放的標准位置。

duck-typing（鴨子型🄵） 一種程式設計風格，它不是藉由檢查一個物件的型🄵來確定它是否具有正確的介面；取而代之的是，method 或屬性會單純地被呼叫或使用。（「如果它看起來像一🄵鴨子而且叫起來像一🄵鴨子，那🄵它一定是一🄵鴨子。」）因🄵🄵調介面而非特定型🄴，精心設計的程式碼能讓多形替代 (polymorphic substitution) 來增進它的靈活性。鴨子型🄵要避免使用*`type()`* 或*`isinstance()`* 進行測試。（但是請注意，鴨子型🄵可以用抽象基底類🄵 *(abstract base class)* 來補充。）然而，它通常會🄵用*`hasattr()`* 測試，或是*EAFP* 程式設計風格。

EAFP Easier to ask for forgiveness than permission.（請求寬恕比請求許可更容易。）這種常見的 Python 編碼風格會先假設有效的鍵或屬性的存在，🄵在該假設被推翻時再捕獲例外。這種乾🄵且快速的風格，其特色是存在許多的 `try` 和 `except` 陳述式。該技術與許多其他語言（例如 C）常見的*LBYL* 風格形成了對比。

expression（運算式） 一段可以被評估🄵求值的語法。🄵句話🄵，一個運算式就是文字、名稱、屬性存取、運算子或函式呼叫等運算式元件的累積，而這些元件都能回傳一個值。與許多其他語言不同的是，🄵非所有的 Python 語言構造都是運算式。另外有一些*statement*（陳述式）不能被用作運算式，例如 `while`。賦值 (assignment) 也是陳述式，而不是運算式。

extension module（擴充模組） 一個以 C 或 C++ 編寫的模組，它使用 Python 的 C API 來與核心及使用者程式碼進行互動。

f-string（f 字串） 以 `'f'` 或 `'F'` 開頭的字串文本通常被稱為「f 字串」，它是格式化的字串文本的縮寫。另請參閱 **PEP 498**。

file object（檔案物件） 一個讓使用者透過檔案導向 (file-oriented) API（如 `read()` 或 `write()` 等 method）來操作底層資源的物件。根據檔案物件被建立的方式，它能協調對真實磁碟檔案或是其他類型的儲存器或通訊裝置（例如標準輸入 / 輸出、記憶體緩衝區、socket（插座）、管 (pipe) 等）的存取。檔案物件也被稱為*類檔案物件 (file-like object)* 或*串流 (stream)*。

實際上，有三種檔案物件：原始的*二進制檔案*、緩衝的*二進制檔案*和*文字檔案*。它們的介面在 `io` 模組中被定義。建立檔案物件的標準方法是使用 `open()` 函式。

file-like object（類檔案物件） *file object*（檔案物件）的同義字。

finder（尋檢器） 一個物件，它會嘗試正在被 import 的模組尋找 *loader*（載入器）。

從 Python 3.3 開始，有兩種類型的尋檢器：*元路徑尋檢器 (meta path finder)* 會使用 `sys.meta_path`，而*路徑項目尋檢器 (path entry finder)* 會使用 `sys.path_hooks`。

請參閱 **PEP 302**、**PEP 420** 和 **PEP 451** 以了解更多細節。

floor division（向下取整除法） 向下無條件舍去到最接近整數的數學除法。向下取整除法的運算子是 `//`。例如，運算式 `11 // 4` 的計算結果是 `2`，與 float（浮點數）真除法所回傳的 `2.75` 不同。請注意，`(-11) // 4` 的結果是 `-3`，因是 `-2.75` 被向下無條件舍去。請參閱 **PEP 238**。

function（函式） 一連串的陳述式，它能向呼叫者回傳一些值。它也可以被傳遞零個或多個*引數*，這些引數可被使用於函式本體的執行。另請參閱 *parameter*（參數）、*method*（方法），以及 function 章節。

function annotation（函式註釋） 函式參數或回傳值的一個 *annotation*（註釋）。

函式註釋通常被使用於*型提示*：例如，這個函式預期會得到兩個 `int` 引數，會有一個 `int` 回傳值：

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

函式註釋的語法在 function 章節有詳細解釋。

請參閱 *variable annotation* 和 **PEP 484**，皆有此功能的描述。

__future__ future 陳述式：`from __future__ import <feature>`，會指示編譯器使用那些在 Python 未來的發布版本中將成標準的語法或語義，來編譯當前的模組。而 `__future__` 模組則記了 *feature*（功能）可能的值。透過 import 此模組對其變數求值，你可以看見一個新的功能是何時首次被新增到此語言中，以及它何時將會（或已經）成預設的功能：

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection（垃圾回收） 當記憶體不再被使用時，將其釋放的過程。Python 執行垃圾回收，是透過參照計數 (reference counting)，以及一個能檢測和中斷參照循環 (reference cycle) 的循環垃圾回收器 (cyclic garbage collector) 來完成。垃圾回收器可以使用 `gc` 模組對其進行控制。

generator（生器） 一個會回傳 *generator iterator*（生器代器）的函式。它看起來像一個正常的函式，但不同的是它包含了 `yield` 運算式，能生一系列的值，這些值可用於 for 圈，或是以 `next()` 函式，每次檢索其中的一個值。

這個術語通常用來表示一個生器函式，但在某些情境中，也可能是表示生器代器。萬一想表達的意思不清楚，那就使用完整的術語，以避免歧義。

generator iterator（生器代器） 一個由 *generator*（生器）函式所建立的物件。

每個 `yield` 會暫停處理程序，記住位置執行狀態（包括區域變數及擱置中的 try 陳述式）。當生器代器回復時，它會從停止的地方繼續執行（與那些每次調用時都要重新開始的函式有所不同）。

generator expression（🄵生器運算式） 一個會回傳🄵代器的運算式。它看起來像一個正常的運算式，後面接著一個 `for` 子句，該子句定義了🄵圈變數、範圍以及一個選擇性的 `if` 子句。該組合運算式會🄵外層函數🄵生多個值：

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function（泛型函式） 一個由多個函式組成的函式，該函式會對不同的型🄵實作相同的運算。呼叫期間應該使用哪種實作，是由調度演算法 (dispatch algorithm) 來🄵定。

另請參🄵*single dispatch*（單一調度）術語表條目、`functools.singledispatch()` 裝飾器和 **PEP 443**。

generic type（泛型型🄵） A *type* that can be parameterized; typically a container class such as `list` or `dict`. Used for *type hints* and *annotations*.

For more details, see *generic alias types*, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

GIL 請參🄵*global interpreter lock*（全域直譯器鎖）。

global interpreter lock（全域直譯器鎖） *CPython* 直譯器所使用的機制，用以確保每次都只有一個執行緒能執行 Python 的*bytecode*（位元組碼）。透過使物件模型（包括關鍵的🄵建型🄵，如`dict`）自動地避免🄵行存取 (concurrent access) 的危險，此機制可以簡化 CPython 的實作。鎖定整個直譯器，會使直譯器更容易成🄵多執行緒 (multi-threaded)，但代價是會犧牲掉多處理器的機器能🄵提供的一大部分平行性 (parallelism)。

然而，有些擴充模組，無論是標準的或是第三方的，它們被設計成在執行壓縮或雜🄵等計算密集 (computationally-intensive) 的任務時，可以解除 GIL。另外，在執行 I/O 時，GIL 總是會被解除。

過去對於建立「無限制執行緒」直譯器（以更高的精細度鎖定共享資料的直譯器）的努力🄵未成功，因🄵在一般的單一處理器情🄵下，效能會有所損失。一般認🄵，若要克服這個效能問題，會使實作變得🄵雜許多，進而付出更高的維護成本。

hash-based pyc（雜🄵架構的 pyc） 一個位元組碼 (bytecode) 暫存檔，它使用雜🄵值而不是對應原始檔案的最後修改時間，來確定其有效性。請參🄵 pyc-invalidation。

hashable（可雜🄵的） 如果一個物件有一個雜🄵值，該值在其生命🄵期中永不改變（它需要一個 `__hash__()` method），且可與其他物件互相比較（它需要一個 `__eq__()` method），那🄵它就是一個可雜🄵物件。比較結果🄵相等的多個可雜🄵物件，它們必須擁有相同的雜🄵值。

可雜🄵性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因🄵這些資料結構都在其🄵部使用了雜🄵值。

大多數的 Python 不可變🄵建物件都是可雜🄵的；可變的容器（例如 list 或 dictionary）🄵不是；而不可變的容器（例如 tuple（元組）和 frozenset），只有當它們的元素是可雜🄵的，它們本身才是可雜🄵的。若物件是使用者自定 class 的實例，則這些物件會被預設🄵可雜🄵的。它們在互相比較時都是不相等的（除非它們與自己比較），而它們的雜🄵值則是衍生自它們的`id()`。

IDLE Python 的 Integrated Development Environment（整合開發環境）。IDLE 是一個基本的編輯器和直譯器環境，它和 Python 的標準發行版本一起被提供。

immutable（不可變物件） 一個具有固定值的物件。不可變物件包括數字、字串和 tuple（元組）。這類物件是不能被改變的。如果一個不同的值必須被儲存，則必須建立一個新的物件。它們在需要🄵定雜🄵值的地方，扮演重要的角色，例如 dictionary（字典）中的一個鍵。

import path（匯入路徑） 一個位置（或路徑項目）的列表，而那些位置就是在 import 模組時，會被*path based finder*（基於路徑的尋檢器）搜尋模組的位置。在 import 期間，此位置列表通常是來自`sys.path`，但對於子套件 (subpackage) 而言，它也可能是來自父套件的 `__path__` 屬性。

importing（匯入） 一個過程。一個模組中的 Python 程式碼可以透過此過程，被另一個模組中的 Python 程式碼使用。

importer（匯入器） 一個能F尋找及載入模組的物件；它既是*finder*（尋檢器）也是*loader*（載入器）物件。

interactive（互動的） Python 有一個互動式直譯器，這表示你可以在直譯器的提示字元輸入陳述式和運算式，立即執行它們F且看到它們的結果。只要F動 `python`，不需要任何引數（可能藉由從你的電腦的主選單選擇它）。這是測試新想法或檢查模塊和包的非常F大的方法（請記住 help(x)）。

interpreted（直譯的） Python 是一種直譯語言，而不是編譯語言，不過這個區分可能有些模糊，因F有位元組碼 (bytecode) 編譯器的存在。這表示原始檔案可以直接被運行，而不需明確地建立另一個執行檔，然後再執行它。直譯語言通常比編譯語言有更短的開發 / 除錯F期，不過它們的程式通常也運行得較慢。另請參F*interactive*（互動的）。

interpreter shutdown（直譯器關閉） 當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵F部結構。它也會多次呼叫*垃圾回收器 (garbage collector)*。這能F觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，F執行其中的程式碼。在關閉階段被執行的程式碼會遇到各種例外，因F它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

直譯器關閉的主要原因，是 `__main__` 模組或正被運行的F本已經執行完成。

iterable（可F代物件） 一種能F一次回傳一個其中成員的物件。可F代物件的例子包括所有的序列型F（像是*list*、*str* 和*tuple*）和某些非序列型F，像是*dict*、*檔案物件*，以及你所定義的任何 class 物件，只要那些 class 有 `__iter__()` method 或是實作*Sequence*（序列）語意的 `__getitem__()` method，該物件就是可F代物件。

可F代物件可用於 `for` F圈和許多其他需要一個序列的地方 (*zip()*、*map()*...)。當一個可F代物件作F引數被傳遞給F建函式*iter()* 時，它會F該物件回傳一個F代器。此F代器適用於針對一組值進行一遍 (one pass) 運算。使用F代器時，通常不一定要呼叫*iter()* 或自行處理F代器物件。`for` 陳述式會自動地F你處理這些事，它會建立一個暫時性的未命名變數，用於在F圈期間保有該F代器。另請參F*iterator*（F代器）、*sequence*（序列）和*generator*（F生器）。

iterator（F代器） 一個表示資料流的物件。重F地呼叫F代器的*__next__()* method（或是將它傳遞給F建函式*next()*）會依序回傳資料流中的各項目。當不再有資料時，則會引發*StopIteration* 例外。此時，該F代器物件已被用盡，而任何對其 `__next__()` method 的進一步呼叫，都只會再次引發*StopIteration*。F代器必須有一個 `__iter__()` method，它會回傳F代器物件本身，所以每個F代器也都是可F代物件，且可以用於大多數適用其他可F代物件的場合。一個明顯的例外，是嘗試多遍F代 (multiple iteration passes) 的程式碼。一個容器物件（像是*list*）在每次你將它傳遞給*iter()* 函式或在 `for` F圈中使用它時，都會F生一個全新的F代器。使用F代器嘗試此事（多遍F代）時，只會回傳在前一遍F代中被用過的、同一個已被用盡的F代器物件，使其看起來就像一個空的容器。

在迭代器类型文中可以找到更多資訊。

key function（鍵函式） 鍵函式或理序函式 (collation function) 是一個可呼叫 (callable) 函式，它會回傳一個用於排序 (sorting) 或定序 (ordering) 的值。例如，*locale.strxfrm()* 被用來F生一個了解區域特定排序慣例的排序鍵。

Python 中的許多工具，都接受以鍵函式來控制元素被定序或分組的方式。它們包括*min()*、*max()*、*sorted()*、*list.sort()*、*heapq.merge()*、*heapq.nsmallest()*、*heapq.nlargest()* 和*itertools.groupby()*。

有幾種方法可以建立一個鍵函式。例如，*str.lower()* method 可以作F不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 `lambda` 運算式被建造，例如 `lambda r: (r[0], r[2])`。另外，*operator* 模組提供了三個鍵函式的建構函式 (constructor)：*attrgetter()*、*itemgetter()* 和*methodcaller()*。關於如何建立和使用鍵函式的範例，請參F如何排序。

keyword argument（關鍵字引數） 請參F*argument*（引數）。

lambda 由單一*expression*（運算式）所組成的一個匿名行F函式 (inline function)，於該函式被呼叫時求值。建立 lambda 函式的語法是 `lambda [parameters]: expression`。

LBYL Look before you leap.（三思而後行。）這種編碼風格會在進行呼叫或查找之前，明確地測試先F條件。這種風格與*EAFP* 方式形成對比，且它的特色是會有許多 `if` 陳述式的存在。

在一個多執行緒環境中，LBYL 方式有在「三思」和「後行」之間引入了競🄵條件 (race condition) 的風險。例如以下程式碼 `if key in mapping: return mapping[key]`，如果另一個執行緒在測試之後但在查找之前，從 *mapping* 中移除了 *key*，則該程式碼就會失效。這個問題可以用鎖 (lock) 或使用 EAFP 編碼方式來解🄵。

list（串列） 一個 Python 🄵建的*sequence*（序列）。🄵管它的名字是 list，它其實更類似其他語言中的一個陣列 (array) 而較不像一個鏈結串列 (linked list)，因🄵存取元素的時間🄵雜度是 O(1)。

list comprehension（串列綜合運算） 一種用來處理一個序列中的全部或部分元素，🄵將處理結果以一個 list 回傳的簡要方法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 會🄵生一個字串 list，其中包含 0 到 255 範圍🄵，所有偶數的十六進位數 (0x..)。`if` 子句是選擇性的。如果省略它，則 `range(256)` 中的所有元素都會被處理。

loader（載入器） 一個能🄵載入模組的物件。它必須定義一個名🄵 `load_module()` 的 method（方法）。載入器通常是被*finder*（尋檢器）回傳。更多細節請參🄵 **PEP 302**，關於*abstract base class*（抽象基底類🄵），請參🄵*importlib.abc.Loader*。

magic method（魔術方法） *special method*（特殊方法）的一個非正式同義詞。

mapping（對映） 一個容器物件，它支援任意鍵的查找，且能實作*abstract base classes（抽象基底類🄵）*中，*Mapping* 或*MutableMapping* 所指定的 method。範例包括*dict*、*collections.defaultdict*、*collections.OrderedDict* 和*collections.Counter*。

meta path finder（元路徑尋檢器） 一種經由搜尋*sys.meta_path* 而回傳的*finder*（尋檢器）。元路徑尋檢器與*路徑項目尋檢器 (path entry finder)* 相關但是不同。

關於元路徑尋檢器實作的 method，請參🄵*importlib.abc.MetaPathFinder*。

metaclass（元類🄵） 一種 class 的 class。Class 定義過程會建立一個 class 名稱、一個 class dictionary（字典），以及一個 base class（基底類🄵）的列表。Metaclass 負責接受這三個引數，🄵建立該 class。大多數的物件導向程式語言會提供一個預設的實作。Python 的特🄵之處在於它能🄵建立自訂的 metaclass。大部分的使用者從未需要此工具，但是當需要時，metaclass 可以提供🄵大且優雅的解🄵方案。它們已被用於記🄵屬性存取、增加執行緒安全性、追🄵物件建立、實作單例模式 (singleton)，以及許多其他的任務。

更多資訊可以在 metaclasses 章節中找到。

method（方法） 一個在 class 本體🄵被定義的函式。如果 method 作🄵其 class 實例的一個屬性被呼叫，則它將會得到該實例物件成🄵它的第一個*argument*（引數）（此引數通常被稱🄵 `self`）。請參🄵*function*（函式）和*nested scope*（巢狀作用域）。

method resolution order（方法解析順序） 方法解析順序是在查找某個成員的過程中，base class（基底類🄵）被搜尋的順序。關於第 2.3 版至今，Python 直譯器所使用的演算法細節，請參🄵 Python 2.3 版方法解析順序。

module（模組） 一個擔任 Python 程式碼的組織單位 (organizational unit) 的物件。模組有一個命名空間，它包含任意的 Python 物件。模組是藉由*importing* 的過程，被載入至 Python。

另請參🄵*package*（套件）。

module spec（模組規格） 一個命名空間，它包含用於載入模組的 import 相關資訊。它是*importlib.machinery.ModuleSpec* 的一個實例。

MRO 請參🄵*method resolution order*（方法解析順序）。

mutable（可變物件） 可變物件可以改變它們的值，但維持它們的*id()*。另請參🄵*immutable*（不可變物件）。

named tuple（附名元組） 術語「named tuple（附名元組）」是指從 tuple 繼承的任何型🄵或 class，且它的可索引 (indexable) 元素也可以用附名屬性來存取。這些型🄵或 class 也可以具有其他的特性。

有些🄵建型🄵是 named tuple，包括由*time.localtime()* 和*os.stat()* 回傳的值。另一個例子是*sys.float_info*：

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

有些 named tuple 是🄵建型🄵（如上例）。或者，一個 named tuple 也可以從一個正規的 class 定義來建立，只要該 class 是繼承自 *tuple*，且定義了附名欄位 (named field) 即可。這類的 class 可以手工編寫，也可以使用工廠函式 (factory function) *collections.namedtuple()* 來建立。後者技術也增加了一些額外的 method，這些 method 可能是在手寫或🄵建的 named tuple 中，無法找到的。

namespace（命名空間） 變數被儲存的地方。命名空間是以 dictionary（字典）被實作。有區域的、全域的及🄵建的命名空間，而在物件中（在 method 中）也有巢狀的命名空間。命名空間藉由防止命名衝突，來支援模組化。例如，函式 *builtins.open* 和 *os.open()* 是透過它們的命名空間來區分彼此。命名空間也藉由明確地區分是哪個模組在實作一個函式，來增進可讀性及可維護性。例如，寫出 *random.seed()* 或 *itertools.islice()* 明確地表示，這些函式分🄵是由 *random* 和 *itertools* 模組在實作。

namespace package（命名空間套件） 一個 **PEP 420** *package*（套件），它只能作🄵子套件 (subpackage) 的一個容器。命名空間套件可能🄵有實體的表示法，而且具體來🄵它們不像是一個 *regular package*（正規套件），因🄵它們🄵🄵有 `__init__.py` 這個檔案。

另請參🄵 *module*（模組）。

nested scope（巢狀作用域） 能🄵參照外層定義 (enclosing definition) 中的變數的能力。舉例來🄵，一個函式如果是在另一個函式中被定義，則它便能🄵參照外層函式中的變數。請注意，在預設情🄵下，巢狀作用域僅適用於參照，而無法用於賦值。區域變數能在最🄵層作用域中讀取及寫入。同樣地，全域變數是在全域命名空間中讀取及寫入。`nonlocal` 容許對外層作用域進行寫入。

new-style class（新式類🄵） 一個舊名，它是指現在所有的 class 物件所使用的 class 風格。在早期的 Python 版本中，只有新式 class 才能使用 Python 較新的、多樣的功能，像是 `__slots__`、描述器 (descriptor)、屬性 (property)、`__getattribute__()`、class method（類🄵方法）和 static method（🄵態方法）。

object（物件） 具有狀態（屬性或值）及被定義的行🄵（method）的任何資料。它也是任何 *new-style class*（新式類🄵）的最終 base class（基底類🄵）。

package（套件） 一個 Python 的 *module*（模組），它可以包含子模組 (submodule) 或是遞🄵的子套件 (subpackage)。技術上而言，套件就是具有 `__path__` 屬性的一個 Python 模組。

另請參🄵 *regular package*（正規套件）和 *namespace package*（命名空間套件）。

parameter（參數） 在 *function*（函式）或 method 定義中的一個命名實體 (named entity)，它指明該函式能🄵接受的一個 *argument*（引數），或在某些情🄵下指示多個引數。共有有五種不同的參數類型：

- *positional-or-keyword*（位置或關鍵字）：指明一個可以按照位置或是作🄵關鍵字引數被傳遞的引數。這是參數的預設類型，例如以下的 *foo* 和 *bar*：

```
def func(foo, bar=None): ...
```

- *positional-only*（僅限位置）：指明一個只能按照位置被提供的引數。在函式定義的參數列表中包含一個 `/` 字元，就可以在該字元前面定義僅限位置參數，例如以下的 *posonly1* 和 *posonly2*：

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*（僅限關鍵字）：指明一個只能以關鍵字被提供的引數。在函式定義的參數列表中，包含一個任意數量位置參數 (var-positional parameter) 或是單純的 `*` 字元，就可以在其後方定義僅限關鍵字參數，例如以下的 *kw_only1* 和 *kw_only2*：

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*（任意數量位置）：指明一串能以任意序列被提供的位置引數（在已被其他參數接受的任何位置引數之外）。這類參數是透過在其參數名稱字首加上 * 來定義的，例如以下的 *args*：

```
def func(*args, **kwargs): ...
```

- *var-keyword*（任意數量關鍵字）：指明可被提供的任意數量關鍵字引數（在已被其他參數接受的任何關鍵字引數之外）。這類參數是透過在其參數名稱字首加上 ** 來定義的，例如上面範例中的 *kwargs*。

參數可以指明引數是選擇性的或必需的，也可以🄵一些選擇性的引數指定預設值。

另請參🄵術語表的*argument*（引數）條目、常見問題中的引數和參數之間的差🄵、`inspect.Parameter` class、function 章節，以及 **PEP 362**。

path entry（路徑項目） 在*import path*（匯入路徑）中的一個位置，而*path based finder*（基於路徑的尋檢器）會參考該位置來尋找要 import 的模組。

path entry finder（路徑項目尋檢器） 被`sys.path_hooks` 中的一個可呼叫物件 (callable)（意即一個*path entry hook*）所回傳的一種*finder*，它知道如何以一個*path entry*定位模組。

關於路徑項目尋檢器實作的 method，請參🄵`importlib.abc.PathEntryFinder`。

path entry hook（路徑項目🄵） 在 `sys.path_hook` 列表中的一個可呼叫物件 (callable)，若它知道如何在一個特定的*path entry* 中尋找模組，則會回傳一個*path entry finder*（路徑項目尋檢器）。

path based finder（基於路徑的尋檢器） 預設的*元路徑尋檢器 (meta path finder)* 之一，它會在一個*import path* 中搜尋模組。

path-like object（類路徑物件） 一個表示檔案系統路徑的物件。類路徑物件可以是一個表示路徑的`str` 或`bytes` 物件，或是一個實作`os.PathLike` 協定的物件。透過呼叫`os.fspath()` 函式，一個支援`os.PathLike` 協定的物件可以被轉🄵🄵`str` 或`bytes` 檔案系統路徑；而`os.fsdecode()` 及`os.fsencode()` 則分🄵可用於確保`str` 及`bytes` 的結果。由 **PEP 519** 引入。

PEP Python Enhancement Proposal（Python 增🄵提案）。PEP 是一份設計🄵明文件，它能🄵 Python 社群提供資訊，或是描述 Python 的一個新功能或該功能的程序和環境。PEP 應該要提供簡潔的技術規範以及被提案功能的運作原理。

PEP 的存在目的，是要成🄵重大新功能的提案、社群中關於某個問題的意見收集，以及已納入 Python 的設計🄵策的記🄵，這些過程的主要機制。PEP 的作者要負責在社群🄵建立共識🄵記🄵反對意見。

請參🄵 **PEP 1**。

portion（部分） 在單一目🄵中的一組檔案（也可能儲存在一個 zip 檔中），這些檔案能對一個命名空間套件 (namespace package) 有所貢獻，如同 **PEP 420** 中的定義。

positional argument（位置引數） 請參🄵*argument*（引數）。

provisional API（暫行 API） 暫行 API 是指，從標准函式庫的向後相容性 (backwards compatibility) 保證中，故意被排除的 API。雖然此類介面，只要它們被標示🄵暫行的，理論上🄵不會有重大的變更，但如果核心開發人員認🄵有必要，也可能會出現向後不相容的變更（甚至包括移除該介面）。這種變更🄵不會無端地🄵生——只有 API 被納入之前未察覺的嚴重基本缺陷被揭露時，它們才會發生。

即使對於暫行 API，向後不相容的變更也會被視🄵「最後的解🄵方案」——對於任何被發現的問題，仍然會盡可能找出一個向後相容的解🄵方案。

這個過程使得標准函式庫能隨著時間不斷進化，而避免耗費過長的時間去鎖定有問題的設計錯誤。請參🄵 **PEP 411** 了解更多細節。

provisional package（暫行套件） 請參🄵*provisional API*（暫行 API）。

Python 3000 Python 3.x 系列版本的🄵稱（很久以前創造的，當時第 3 版的發布是在🄵遠的未來。）也可以縮寫🄵「Py3k」。

Pythonic（Python 風格的） 一個想法或一段程式碼，它應用了 Python 語言最常見的慣用語，而不是使用其他語言常見的概念來實作程式碼。例如，Python 中常見的一種習慣用法，是使用一個 `for` 陳述式，對一個可🄴代物件的所有元素進行🄵圈。許多其他語言🄵🄵有這種類型的架構，所以不熟悉 Python 的人有時會使用一個數值計數器來代替：

```
for i in range(len(food)):
    print(food[i])
```

相較之下，以下方法更簡潔、更具有 Python 風格：

```
for piece in food:
    print(piece)
```

qualified name（限定名稱） 一個「點分隔名稱」，它顯示從一個模組的全域作用域到該模組中定義的 class、函式或 method 的「路徑」，如 **PEP 3155** 中的定義。對於頂層的函式和 class 而言，限定名稱與其物件名稱相同：

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

當用於引用模組時，*完全限定名稱 (fully qualified name)* 是表示該模組的完整點分隔路徑，包括任何的父套件，例如 `email.mime.text`：

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count（參照計數） 對於一個物件的參照次數。當一個物件的參照計數下降到零時，它會被解除配置 (deallocated)。參照計數通常在 Python 程式碼中看不到，但它🄵是*CPython* 實作的一個關鍵元素。*sys* 模組定義了一個*getrefcount()* 函式，程序設計師可以呼叫該函式來回傳一個特定物件的參照計數。

regular package（正規套件） 一個傳統的*package*（套件），例如一個包含 `__init__.py` 檔案的目🄵。

另請參🄵*namespace package*（命名空間套件）。

__slots__ 在 class 🄵部的一個宣告，它藉由預先宣告實例屬性的空間，以及消除實例 dictionary（字典），來節省記憶體。雖然該技術很普遍，但它有點難以正確地使用，最好保留給那種在一個記憶體關鍵 (memory-critical) 的應用程式中存在大量實例的罕見情🄵。

sequence（序列） 一個*iterable*（可🄴代物件），它透過 `__getitem__()` special method（特殊方法），使用整數索引來支援高效率的元素存取，🄵定義了一個 `__len__()` method 來回傳該序列的長度。一些🄵建序列型🄵包括*list*、*str*、*tuple* 和*bytes*。請注意，雖然*dict* 也支援 `__getitem__()` 和 `__len__()`，但它被視🄵對映 (mapping) 而不是序列，因🄵其查找方式是使用任意的*immutable* 鍵，而不是整數。

抽象基底類🄵 (abstract base class) *collections.abc.Sequence* 定義了一個更加豐富的介面，🄵不僅止於 `__getitem__()` 和 `__len__()`，還增加了 `count()`、`index()`、`__contains__()` 和 `__reversed__()`。實作此擴充介面的型🄵，可以使用*register()* 被明確地🄵🄵。

set comprehension（集合綜合運算） 一種緊密的方法，用來處理一個可F代物件中的全部或部分元素，F將處理結果以一個 set 回傳。`results = {c for c in 'abracadabra' if c not in 'abc'}` 會F生一個字串 set：`{'r', 'd'}`。請參F comprehensions。

single dispatch（單一調度） *generic function*（泛型函式）調度的一種形式，在此，實作的選擇是基於單一引數的型F。

slice（切片） 一個物件，它通常包含一段*sequence*（序列）的某一部分。建立一段切片的方法是使用下標符號（subscript notation）"[]"，若要給出多個數字，則在數字之間使用冒號，例如 `in variable_name[1:3:5]`。在括號（下標）符號的F部，會使用*slice*物件。

special method（特殊方法） 一種會被 Python 自動呼叫的 method，用於對某種型F執行某種運算，例如加法。這種 method 的名稱會在開頭和結尾有兩個下底F。Special method 在 specialnames 中有詳細F明。

statement（陳述式） 陳述式是一個套組（suite，一個程式碼「區塊」）中的一部分。陳述式可以是一個*expression*（運算式），或是含有關鍵字（例如 `if`、`while` 或 `for`）的多種結構之一。

text encoding（文字編碼） A string in Python is a sequence of Unicode code points (in range `U+0000--U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as "encoding", and recreating the string from the sequence of bytes is known as "decoding".

There are a variety of different text serialization *codecs*, which are collectively referred to as "text encodings".

text file（文字檔案） 一個能F讀取和寫入*str*物件的一個*file object*（檔案物件）。通常，文字檔案實際上是存取位元組導向的資料流 (byte-oriented datastream) F會自動處理*text encoding*（文字編碼）。文字檔案的例子有：以文字模式（`'r'` 或 `'w'`）開F的檔案、*sys.stdin*、*sys.stdout* 以及*io.StringIO* 的實例。

另請參F*binary file*（二進制檔案），它是一個能F讀取和寫入*類位元組串物件 (bytes-like object)* 的檔案物件。

triple-quoted string（三引號F字串） 由三個雙引號（ "") 或單引號 (') 的作F邊界的一個字串。雖然它們FF有提供F於單引號字串的任何額外功能，但基於許多原因，它們仍是很有用的。它們讓你可以在字串中包含未跳F (unescaped) 的單引號和雙引號，而且它們不需使用連續字元 (continuation character) 就可以跨越多行，這使得它們在編寫F明字串時特F有用。

type（型F） 一個 Python 物件的型FF定了它是什F類型的物件；每個物件都有一個型F。一個物件的型F可以用它的*__class__* 屬性來存取，或以 `type(obj)` 來檢索。

type alias（型FF名） 一個型F的同義詞，透過將型F指定給一個識F符 (identifier) 來建立。

型FF名對於簡化*型F提示 (type hint)* 很有用。例如：

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

可以寫成這樣，更具有可讀性：

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

請參F*typing* 和 **PEP 484**，有此功能的描述。

type hint（型F提示） 一種*annotation*（F釋），它指定一個變數、一個 class 屬性或一個函式的參數或回傳值的預期型F。

型F提示是選擇性的，而不是被 Python F制的，但它們對F態型F分析工具很有用，F能協助 IDE 完成程式碼的補全 (completion) 和重構 (refactoring)。

全域變數、class 屬性和函式（不含區域變數）的型F提示，都可以使用*typing.get_type_hints()*來存取。

請參F*typing* 和 **PEP 484**，有此功能的描述。

universal newlines（通用F行字元） 一種解譯文字流 (text stream) 的方式，會將以下所有的情F識F F一行的結束：Unix 行尾慣例 `'\n'`、Windows 慣例 `'\r\n'` 和舊的 Macintosh 慣例 `'\r'`。請參F **PEP 278** 和 **PEP 3116**，以及用於*bytes.splitlines()* 的附加用途。

variable annotation（變數F釋） 一個變數或 class 屬性的*annotation*（F釋）。

F釋變數或 class 屬性時，賦值是選擇性的：

```
class C:
    field: 'annotation'
```

變數F釋通常用於型F提示 *(type hint)*：例如，這個變數預期會取得*int*（整數）值：

```
count: int = 0
```

變數F釋的語法在 annassign 章節有詳細的解釋。

請參F*function annotation*（函式F釋）、**PEP 484** 和 **PEP 526**，皆有此功能的描述。

virtual environment（F擬環境） 一個協作隔離 (cooperatively isolated) 的執行環境，能讓 Python 的使用者和應用程式得以安裝和升級 Python 發F套件，而不會對同一個系統上運行的其他 Python 應用程式的行F F生干擾。

另請參F*venv*。

virtual machine（F擬機器） 一部完全由軟體所定義的電腦 (computer)。Python 的F擬機器會執行由*bytecode*（位元組碼）編譯器所發出的位元組碼。

Zen of Python（Python 之F） Python 設計原則與哲學的列表，其F容有助於理解和使用此語言。此列表可以透過在互動式提式字元後輸入「`import this`」來找到它。

APPENDIX B

關於這些🄵明文件

這些🄵明文件是透過 Sphinx （一個專🄵 Python 🄵明文件所撰寫的文件處理器）將使用 reStructuredText 撰寫的原始檔轉🄵而成。

如同 Python 自身，透過自願者的努力下🄵出文件與封裝後自動化執行工具。若想要回報臭蟲，請見 reporting-bugs 頁面，🄵含相關資訊。我們永遠歡迎新的自願者加入！

致謝：

- Fred L. Drake, Jr.，原始 Python 文件工具集的創造者以及一大部份🄵容的作者。
- 創造 reStructuredText 和 Docutils 工具組的 Docutils 專案；
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

B.1 Python 文件的貢獻者們

許多人都曾🄵 Python 這門語言、Python 標准函式庫和 Python 🄵明文件貢獻過。Python 所發🄵的原始碼中含有部份貢獻者的清單，請見 Misc/ACKS 。

正因🄵 Python 社群的撰寫與貢獻才有這份這🄵棒的🄵明文件 -- 感謝所有貢獻過的人們！

APPENDIX C

沿革與授權

C.1 軟體沿革

Python 是由荷蘭數學和計算機科學研究學會（CWI，見 https://www.cwi.nl/）的 Guido van Rossum 於 1990 年代早期所創造，目的是作🄵一種稱🄵 ABC 語言的後繼者。🄵管 Python 包含了許多來自其他人的貢獻，Guido 仍是其主要作者。

1995 年，Guido 在維吉尼亞州雷斯頓的國家創新研究公司（CNRI，見 https://www.cnri.reston.va.us/）繼續他在 Python 的工作，🄵在那🄵發🄵了該軟體的多個版本。

2000 年五月，Guido 和 Python 核心開發團隊轉移到 BeOpen.com 🄵成立了 BeOpen PythonLabs 團隊。同年十月，PythonLabs 團隊轉移到 Digital Creations（現🄵 Zope Corporation；見 https://www.zope.org/）。2001 年，Python 軟體基金會（PSF，見 https://www.python.org/psf/）成立，這是一個專🄵擁有 Python 相關的智慧財🄵權而創立的非營利組織。Zope Corporation 是 PSF 的一個贊助會員。

所有的 Python 版本都是開源的（有關開源的定義，參🄵 https://opensource.org/）。歷史上，大多數但非全部的 Python 版本，也是 GPL 相容的；以下表格總結各個版本的差🄵。

發🄵版本	源自	年份	擁有者	GPL 相容性
0.9.0 至 1.2	不適用	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	否
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 以上	2.1.1	2001 至今	PSF	是

備🄵： GPL 相容🄵不表示我們是在 GPL 下發🄵 Python。不像 GPL，所有的 Python 授權都可以讓您發🄵修改

後的版本，但不一定要使您的變更成🄵開源。GPL 相容的授權使得 Python 可以結合其他在 GPL 下發🄵的軟體一起使用；但其它的授權則不行。

感謝許多的外部志工，在 Guido 指導下的付出，使得這些版本的發🄵成🄵可能。

C.2 關於存取或以其他方式使用 Python 的合約條款

Python 軟體和🄵明文件的授權是基於*PSF 授權合約*。

從 Python 3.8.6 開始，🄵明文件中的範例、程式庫和其他程式碼，是被雙重授權 (dual licensed) 於 PSF 授權合約以及*Zero-Clause BSD 授權*。

有些被納入 Python 中的軟體是基於不同的授權。這些授權將會與其授權之程式碼一起被列出。關於這些授權的不完整清單，請參🄵被收🄵軟體的授權與致謝。

C.2.1 用於 PYTHON 3.9.13 的 PSF 授權合約

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.9.13 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.9.13 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2022 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.9.13 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.9.13 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   3.9.13.

4. PSF is making Python 3.9.13 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 3.9.13 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.9.13
```

```
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.9.13, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach␣
↪of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 3.9.13, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 用於 PYTHON 2.0 的 BEOPEN.COM 授權合約

BEOPEN PYTHON 開源授權合約第 1 版

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
```

(下页继续)

(繼續上一頁)

```
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 用於 PYTHON 1.6.1 的 CNRI 授權合約

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
```

(下页继续)

(繼續上一頁)

```
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 用於 PYTHON 0.9.0 至 1.2 的 CWI 授權合約

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 用於 PYTHON 3.9.13 🄵明文件🄵程式碼的 ZERO-CLAUSE BSD 授權

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 被收🄵軟體的授權與致謝

本節是一個不完整但持續增加的授權與致謝清單，對象是在 Python 發🄵版本中所收🄵的第三方軟體。

C.3.1 Mersenne Twister

_random 模組包含了以 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 的下載🄵容🄵基礎的程式碼。以下是原始程式碼的完整聲明：

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

socket 模組使用 `getaddrinfo()` 和 `getnameinfo()` 函式，它們在 WIDE 專案（http://www.wide.ad.jp/）🄵，於不同的原始檔案中被編碼：

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 非同步 socket 服務

asynchat 和*asyncore* 模組包含以下聲明：

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie 管理

http.cookies 模組包含以下聲明：

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 執行追🄵

trace 模組包含以下聲明：

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 與 UUdecode 函式

uu 模組包含以下聲明:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                        All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 遠端程序呼叫

xmlrpc.client 模組包含以下聲明:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(下页继续)

(繼續上一頁)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test_epoll` 模組包含以下聲明:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` 模組對於 kqueue 介面包含以下聲明:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

(下页继续)

(繼續上一頁)

```
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

Python/pyhash.c 檔案包含 Marek Majkowski' 基於 Dan Bernstein 的 SipHash24 演算法的實作。它包含以下聲明:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 與 dtoa

Python/dtoa.c 檔案提供了 C 的 dtoa 和 strtod 函式，用於將 C 的雙精度浮點數和字串互相轉F。該檔案是衍生自 David M. Gay 建立的同名檔案，後者現在可以從 http://www.netlib.org/fp/ 下載。於 2009 年 3 月 16 日所檢索的原始檔案包含以下版權與授權聲明:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
```

(下页继续)

(繼續上一頁)

```
*
*****************************************************************/
```

C.3.12 OpenSSL

如果 OpenSSL 函式庫可被作業系統使用，則*hashlib*、*posix*、*ssl*、*crypt* 模組會使用它來提升效能。此外，因🄵 Windows 和 macOS 的 Python 安裝程式可能包含 OpenSSL 函式庫的副本，所以我們也在此收🄵 OpenSSL 授權的副本：

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
   *
   * Redistribution and use in source and binary forms, with or without
   * modification, are permitted provided that the following conditions
   * are met:
   *
   * 1. Redistributions of source code must retain the above copyright
   *    notice, this list of conditions and the following disclaimer.
   *
   * 2. Redistributions in binary form must reproduce the above copyright
   *    notice, this list of conditions and the following disclaimer in
   *    the documentation and/or other materials provided with the
   *    distribution.
   *
   * 3. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
```

(下页继续)

(繼續上一頁)

```
    *
    * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
    * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
    * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
    * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
    * 1. Redistributions of source code must retain the copyright
    *    notice, this list of conditions and the following disclaimer.
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in the
    *    documentation and/or other materials provided with the distribution.
    * 3. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
```

(下页继续)

(繼續上一頁)

```
    *     Eric Young (eay@cryptsoft.com)"
    *    The word 'cryptographic' can be left out if the rouines from the library
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
    *
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
    * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
    * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
    * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
    * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
    * SUCH DAMAGE.
    *
    * The licence and distribution terms for any publically available version or
    * derivative of this code cannot be changed.  i.e. this code cannot simply be
    * copied and put under another distribution licence
    * [including the GNU Public Licence.]
    */
```

C.3.13 expat

除非在建置 pyexpat 擴充時設定🄵 `--with-system-expat`，否則該擴充會用一個🄵含 expat 原始碼的副本來建置：

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

除非在建置 _ctypes 擴充時設定🄵 --with-system-libffi，否則該擴充會用一個🄴含 libffi 原始碼的副本來建置：

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

如果在系統上找到的 zlib 版本太舊以致於無法用於建置*zlib* 擴充，則該擴充會用一個🄴含 zlib 原始碼的副本來建置：

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

tracemalloc 使用的雜🄴表 (hash table) 實作，是以 cfuhash 專案🄴基礎：

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

除非在建置 _decimal 模組時設定🄵 --with-system-libmpdec，否則該模組會用一個🄴含 libmpdec 函式庫的副本來建置：

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
```

(下页继续)

(繼續上一頁)

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 測試套件

test 程式包中的 C14N 2.0 測試套件 (Lib/test/xmltestdata/c14n-20/) 是從 W3C 網站 https://www.w3.org/TR/xml-c14n2-testcases/ 被檢索，且是基於 3-clause BSD 授權被發🄵：

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

APPENDIX D

版權宣告

Python 和這份🄴明文件的版權：

完整的授權條款資訊請參見沿革與授權。

Bibliography

[Frie09] Friedl, Jeffrey. Mastering Regular Expressions. 3rd ed., O'Reilly Media, 2009. 该书的第三版不再包含 Python，但第一版极详细地覆盖了正则表达式模式串的编写。

[C99] ISO/IEC 9899:1999. "Programming languages -- C." 该标准的公开草案可从 http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf获得。

Python 模組索引

f

g

h

i

j

k

l

m

n

o

p

q

r

s

t

u

V

W

X

Z

非字母

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z