

---

# 隔离扩展模块

发行版本 3.11.14

Guido van Rossum and the Python development team

十月 16, 2025

Python Software Foundation  
Email: docs@python.org

## Contents

|     |                        |   |
|-----|------------------------|---|
| 1   | 谁应当阅读本文档               | 2 |
| 2   | 背景                     | 2 |
| 2.1 | 进入模块级状态                | 2 |
| 2.2 | 隔离的模块对象                | 2 |
| 2.3 | 令人惊讶的边界情况              | 3 |
| 3   | 让多解释器下模块保持安全           | 3 |
| 3.1 | 管理全局状态                 | 3 |
| 3.2 | 管理模块级状态                | 3 |
| 3.3 | 回退选项：每个进程限一个模块对象       | 4 |
| 3.4 | 函数对模块状态的访问             | 4 |
| 4   | 堆类型                    | 4 |
| 4.1 | 将静态类型改为堆类型             | 5 |
| 4.2 | 定义堆类型                  | 5 |
| 4.3 | 垃圾回收协议                 | 5 |
| 4.4 | 类对模块状态的访问              | 7 |
| 4.5 | 常规方法对模块状态的访问           | 7 |
| 4.6 | 槽位方法、读取方法和设置方法对模块状态的访问 | 8 |
| 4.7 | 模块状态的生命期               | 9 |
| 5   | 未解决的问题                 | 9 |
| 5.1 | 类级作用域                  | 9 |
| 5.2 | 无损转换为堆类型               | 9 |

---

### 摘要

在传统上，属于 Python 扩展模块的状态都是保存为 C static 变量，它们具有进程级的作用域。本文档描述了此类进程级状态的问题并演示了一种更安全的方式：模块级状态。

本文档还描述了如何在可能的情况下切换到模块级状态。这种转换涉及为状态分配空间、从静态类型到堆类型的潜在切换，以及——也许是最重要的一——从代码访问模块级状态。

# 1 谁应当阅读本文档

本指南是针对想要让扩展更安全地在将 Python 本身用作库的应用程序中使用的 C-API 扩展维护者撰写的。

## 2 背景

解释器是 Python 代码运行所在的上下文。它包含配置（例如导入路径）和运行时状态（例如已导入模块的集合）。

Python 支持在一个进程中运行多个解释器。这里有两种情况需要考虑—用户可能会以下列方式运行解释器：

- 串行，使用多个 `Py_InitializeEx()`/`Py_FinalizeEx()` 循环，以及
- 并行，使用 `Py_NewInterpreter()`/`Py_EndInterpreter()` 管理多个“子解释器”。

这两种情况（以及它们的组合）最适用于将 Python 嵌入到某个库中。库通常不应假定使用它们的应用程序，这包括假定存在一个进程级的“主 Python 解释器”。

在历史上，Python 扩展模块对这种应用场景处理不佳。许多扩展模块（甚至是某些标准库模块）都是使用进程内共享的全局状态，因为 C `static` 变量十分易用。结果，本应专属于某个解释器的数据最终却被多个解释器所共享。除非扩展的开发者小心谨慎，否则当一个模块被相同进程内的多个解释器导入时很容易引入会导致崩溃的边界情况。

不幸的是，解释器级状态很不容易做到。扩展的作者在开发中总是倾向于不考虑多解释器的情况，并且目前要测试此类行为也是很困难的。

### 2.1 进入模块级状态

Python 的 C API 没有关注于解释器级状态，而是演化为更好地支持更细粒度的模块级状态。这意味着 C 层级数据被关联到了模块对象。每个解释器会创建它自己的模块级对象，保持数据的相互分隔。要测试这种分隔，甚至可以将对应于单个扩展的多个模块对象加载到同一个解释器中。

模块级状态提供了一种处理生命周期和资源归属的简单方式：扩展模块将在模块对象被创建时初始化，并在其释放时被清理。在这一点上，模块就像是任何其他 `PyObject*`；没有必要添加—或者去除—处理“解释器关闭”的钩子。

请注意各种不同“全局”状态：进程级、解释器级、线程级状态的应用场景。默认为模块级状态，其他状态也是可选择的，但你应当将它们视为特殊情况：如果你需要它们，你应当给予它们额外的关注和测试。（请注意本指南并没有涉及它们。）

### 2.2 隔离的模块对象

在开发扩展模块时要记住的关键点是多个模块对象可以从单个共享库来创建。例如：

```
>>> import sys
>>> import binascii
>>> old_binascii = binascii
>>> del sys.modules['binascii']
>>> import binascii # create a new module object
>>> old_binascii == binascii
False
```

作为经验法则，这两个模块应该是完全独立的。模块专属的所有对象和状态应该被封装在模块对象内部，不与其他模块对象共享，并在模块对象被释放时进行清理。由于这只是一个经验法则，例外情况也是可能的（参见 *Managing Global State*），但这将需要更多的考虑并注意边界情况。

虽然有些模块不用太多的严格限制，但是隔离的模块使得更容易制定适合各种应用场景的明确期望和指南。

## 2.3 令人惊讶的边界情况

请注意隔离的模块会创造一些令人惊讶的边界情况。最明显的一点，每个模块对象通常都不会与其他类似模块共享它的类和异常。继续上面的例子，请注意 `old_binascii.Error` 和 `binascii.Error` 是单独的对象。在下面的代码中，异常 不会被捕获：

```
>>> old_binascii.Error == binascii.Error
False
>>> try:
...     old_binascii.unhexlify(b'qwertyuiop')
... except binascii.Error:
...     print('boo')
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
binascii.Error: Non-hexadecimal digit found
```

这是预期的结果。请注意纯 Python 模块的行为相同：它是 Python 语言特性的一部分。

最终目标是让扩展模块在 C 层级上保持安全，使破坏不容易实现。“手动”改变 `sys.modules` 被视为是破坏行为。

## 3 让多解释器下模块保持安全

### 3.1 管理全局状态

有时，与一个 Python 模块相关联的状态并不是该模块专属的，而是整个进程（或者比模块“更全局化”的其他东西）共享。例如：

- readline 模块管理一个终端。
- 在电路板上运行的模块想要控制一个板载 LED。

在这些情况下，Python 模块应当提供对全局状态的访问，而不是拥有它。如果可能，编写模块时要让它的多个副本可以独立地访问全局状态（能配合其它的库，不论它们是使用 Python 还是其他语言）。如果这无法做到，可考虑显式加锁。

如果有必要使用进程级全局状态，避免多解释器相关问题的最简单的方式是显式地阻止模块在一个进程中被多次加载——参见[回退选项：每个进程限一个模块对象](#)。

### 3.2 管理模块级状态

要使用模块级状态，请使用多阶段扩展模块初始化。这将标示你的模块能正确地支持多解释器。

将 `PyModuleDef.m_size` 设为一个正数来为模块请求指定字节的本地存储。通常，这将被设为某个模块专属 `struct` 的大小，它可以保存模块的所有 C 层级状态。特别地，它应当是你存放类指针（包括异常，但不包括静态类型）和 C 代码正常运作所需设置（如 `csv` 的 `field_size_limit` 等）的地方。

---

**备注：**另一个选项是将状态保存在模块的 `__dict__` 中，但你必须避免当用户从 Python 代码中修改 `__dict__` 导致的程序崩溃。这通常意味着要在 C 层级上进行错误和类型检查，很容易弄错又很难充分测试。

但是，如果 C 代码不需要模块状态，则仅将其保存在 `__dict__` 中就是一个好主意。

---

如果模块状态包括 PyObject 指针，则模块对象必须持有对这些对象的引用并实现模块层级的钩子 m\_traverse, m\_clear 和 m\_free。它们的作用方式很像类的 tp\_traverse, tp\_clear 和 tp\_free。添加它们将会增加工作量并使代码更冗长；这是为了让模块能干净地卸载所需的代价。

带有模块级状态的模块示例目前可在 [xxlimited](#) 获取；模块初始化的示例见文件的末尾部分。

### 3.3 回退选项：每个进程限一个模块对象

非负的 PyModuleDef.m\_size 值表示一个模块能正确地支持多解释器。如果你的模块还不能做到这样，你可以显式地设置你的模块在每个进程中只能加载一次。例如：

```
static int loaded = 0;

static int
exec_module(PyObject* module)
{
    if (loaded) {
        PyErr_SetString(PyExc_ImportError,
                        "cannot load module more than once per process");
        return -1;
    }
    loaded = 1;
    // ... rest of initialization
}
```

### 3.4 函数对模块状态的访问

从模块层级的函数访问状态是相当直观的。函数通过它们的第一个参数获得模块对象；要提取状态，你可以使用 PyModule\_GetState：

```
static PyObject *
func(PyObject *module, PyObject *args)
{
    my_struct *state = (my_struct*)PyModule_GetState(module);
    if (state == NULL) {
        return NULL;
    }
    // ... rest of logic
}
```

---

**备注：**如果模块状态不存在则 PyModule\_GetState 可能返回 NULL 而不设置异常，即 PyModuleDef.m\_size 为零。在你自己的模块中，你可以任意控制 m\_size，因此这很容易避免。

---

## 4 堆类型

在传统上，在 C 代码中定义的类型都是 静态的；也就是说，static PyTypeObject 结构体在代码中直接定义并使用 PyType\_Ready() 来初始化。

这样的类型必须在进程范围内共享。在模块对象之间共享它们需要注意它们所拥有或访问的任何状态。要限制可能出现的问题，静态类型在 Python 层级上是不可变的：例如，你无法设置 str.myattribute = 123。

**CPython 实现细节：**在解释器之间共享真正不可变的对象是可行的，只要它们不提供对可变对象的访问。但是，在 CPython 中，每个 Python 对象都有一个可变的实现细节：引用计数。对引用计数的更改是由 GIL 来保护的。因此，跨解释器共享任何 Python 对象的代码都隐式地依赖于 CPython 现有的、进程级的 GIL。

因为它们是不可变的进程级全局对象，所以静态类型无法访问“它们的”模块状态。如果任何此种类型的方法需要访问模块状态，则该类型必须被转换为堆分配类型，或者简称为堆类型。此种类型相对更接近由 Python 的 `class` 语句所创建的类。

对于新模块，默认使用堆类型是一个很好的经验法则。

## 4.1 将静态类型改为堆类型

静态类型可以转换为堆类型，但要注意堆类型 API 并非针对静态类型的“无损”转换——也就是说，创建与给定静态类型完全一致的类型来设计的。因此，当在新的 API 中重写类定义时，你很容易在无意中改变一些细节（例如可封存性或所继承的槽位等）。请始终确保测试对你来说重要的细节。

特别要关注以下两点（但请注意这并非一个完整的列表）：

- 不同于静态类型，堆类型对象默认是可变的。请使用 `Py_TPFLAGS_IMMUTABLETYPE` 旗标来防止可变性。
- 堆类型默认继承 `tp_new`，因此有可能通过 Python 代码来初始化它们。你可以使用 `Py_TPFLAGS_DISALLOW_INSTANTIATION` 旗标来防止此特性。

## 4.2 定义堆类型

堆类型可以通过填充 `PyType_Spec` 结构体来创建，它是对于特定类的描述或“蓝图”，并调用 `PyType_FromModuleAndSpec()` 来构造新的类对象。to construct a new class object.

---

**备注：**其他的函数，如 `PyType_FromSpec()`，也可以创建堆类型，但 `PyType_FromModuleAndSpec()` 会将模块关联到类，以允许从方法访问模块状态。

---

类通常应当同时保存在模块的状态（用于从 C 中安全地访问）和模块的 `__dict__` 中（用于从 Python 代码中访问）。

## 4.3 垃圾回收协议

堆类型的实例会持有一个指向其类型的引用。这能确保类型的销毁不会发生在其实例之前，但可能会导致需要由垃圾回收器来打破的引用循环。

要避免内存泄漏，堆类型的实例必须实现垃圾回收协议。也就是说，堆类型应当：

- 具有 `Py_TPFLAGS_HAVE_GC` 旗标。
- 定义一个使用 `Py_tp_traverse` 的遍历函数，它将访问该类型（例如使用 `Py_VISIT(Py_TYPE(self))`）。

请参阅 `Py_TPFLAGS_HAVE_GC` 和 `tp_traverse` 来了解对相关问题的更多考量。

定义堆类型的 API 在有机地增长，使得它目前的使用状况有些尴尬。以下章节将引导您解决常见的问题。

### tp\_traverse 在 Python 3.8 及更低的版本中

从“`tp_traverse`”访问类型的要求是在 Python 3.9 中添加的。如果你要支持 Python 3.8 及更低版本，则遍历函数不可访问类型，因此必须使用更复杂的方式：

```
static int my_traverse(PyObject *self, visitproc visit, void *arg)
{
    if (Py_Version >= 0x03090000) {
        Py_VISIT(Py_TYPE(self));
    }
}
```

(续下页)

```
return 0;
}
```

不幸的是，Py\_Version 直到 Python 3.11 才被加入。作为替代，请使用：

- PY\_VERSION\_HEX，如果不使用稳定 ABI 的话，或者
- sys.version\_info (通过 PySys\_GetObject() 和 PyArg\_ParseTuple())。

## 委托 tp\_traverse

如果你的遍历函数委托给了其基类（或另一个类型）的 tp\_traverse，请确保 Py\_TYPE(self) 只被访问一次。请注意只有堆类型会被预期访问 tp\_traverse 中的类型。

举例来说，如果你的遍历函数包括：

```
base->tp_traverse(self, visit, arg)
```

... 并且 base 可能是一个静态类型，则它也应当包括：

```
if (base->tp_flags & Py_TPFLAGS_HEAPTYPE) {
    // a heap type's tp_traverse already visited Py_TYPE(self)
} else {
    if (Py_Version >= 0x03090000) {
        Py_VISIT(Py_TYPE(self));
    }
}
```

不需要在 tp\_new 和 tp\_clear 中处理该类型的引用计数。

## 定义 tp\_dealloc

如果你的类型有自定义的 tp\_dealloc 函数，则它需要：

- 在任何字段失效之前调用 PyObject\_GC\_UnTrack()，并且
- 递减该类型的引用计数。

要在 tp\_free 被调用时保持类型有效，必须在撤销分配实例 之后递减该类型的引用计数。例如：

```
static void my_dealloc(PyObject *self)
{
    PyObject_GC_UnTrack(self);
    ...
    PyTypeObject *type = Py_TYPE(self);
    type->tp_free(self);
    Py_DECREF(type);
}
```

默认的 tp\_dealloc 函数会执行此操作，因此如果你的类型 没有重载 tp\_dealloc 你就不需要添加它。



## 没有重载 tp\_free

堆类型的 tp\_free 槽位必须设为 PyObject\_GC\_Del()。这是默认的设置；请不要重载它。

## 避免 PyObject\_New

带 GC 追踪的对象需要使用带 GC 感知的函数来分配。

如果你使用 PyObject\_New() 或 PyObject\_NewVar():

- 如有可能，请获取并调用类型的 tp\_alloc 槽位。也就是说，将 TYPE \*o = PyObject\_New(TYPE, typeobj) 替换为:

```
TYPE *o = typeobj->tp_alloc(typeobj, 0);
```

同样地替换 o = PyObject\_NewVar(TYPE, typeobj, size)，但要使用指定大小而不是 0。

- 如果无法执行以上操作（例如在自定义的 tp\_alloc 中），请调用 PyObject\_GC\_New() 或 PyObject\_GC\_NewVar():

```
TYPE *o = PyObject_GC_New(TYPE, typeobj);  
  
TYPE *o = PyObject_GC_NewVar(TYPE, typeobj, size);
```

## 4.4 类对模块状态的访问

如果你有一个使用 PyType\_FromModuleAndSpec() 定义的类型对象，你可以调用 PyType\_GetModule() 来获取关联的模块，然后调用 PyModule\_GetState() 来获取模块的状态。to get the module's state.

要省略一些繁琐的错误处理样板代码，你可以使用 PyType\_GetModuleState() 来合并这两步，得到:

```
my_struct *state = (my_struct*)PyType_GetModuleState(type);  
if (state == NULL) {  
    return NULL;  
}
```

## 4.5 常规方法对模块状态的访问

从一个类的方法访问模块层级的状态在某些方面会更为复杂，但通过 Python 3.9 所引入的 API 这是可能做到的。为了获取状态，你需要首先获取 定义的类，然后从中获取模块状态。

最大的障碍是获取 方法定义所在的类，简称为方法“定义的类”。定义的类可以拥有一个指向作为其组成部分的方法的引用。

不要混淆定义的类和 Py\_TYPE(self)。如果方法是在你的类型的一个 子类上被调用的，则 Py\_TYPE(self) 将指向该子类，它可能是在另一个模块中定义的。

**备注：** 下面的 Python 代码可以演示这一概念。Base.get\_defining\_class 将返回 Base，即使 type(self) == Sub:

```
class Base:  
    def get_type_of_self(self):  
        return type(self)  
  
    def get_defining_class(self):  
        return __class__
```

(续下页)

```
class Sub(Base):
    pass
```

对于要获取其“定义方类”的方法，它必须使用 METH\_METHOD | METH\_FASTCALL | METH\_KEYWORDS 调用惯例以及相应的 PyCMethod 签名：

```
PyObject *PyCMethod(
    PyObject *self,           // object the method was called on
    PyTypeObject *defining_class, // defining class
    PyObject *const *args,    // C array of arguments
    Py_ssize_t nargs,        // length of "args"
    PyObject *kwnames)       // NULL, or dict of keyword arguments
```

一旦你得到了定义的类，即可调用 PyType\_GetModuleState() 来获取它所关联的模块的状态。

例如：

```
static PyObject *
example_method(PyObject *self,
               PyTypeObject *defining_class,
               PyObject *const *args,
               Py_ssize_t nargs,
               PyObject *kwnames)
{
    my_struct *state = (my_struct*)PyType_GetModuleState(defining_class);
    if (state == NULL) {
        return NULL;
    }
    ... // rest of logic
}

PyDoc_STRVAR(example_method_doc, "...");

static PyMethodDef my_methods[] = {
    {"example_method",
     (PyCFunction) (void (*)(void)) example_method,
     METH_METHOD|METH_FASTCALL|METH_KEYWORDS,
     example_method_doc},
    {NULL},
}
```

## 4.6 槽位方法、读取方法和设置方法对模块状态的访问

**备注：** 这是 Python 3.11 的新增特性。

槽位方法—即特殊方法的 C 快速等价物，如 nb\_add 对应 \_\_add\_\_ 而 tp\_new 对应初始化方法—具有不允许传入定义类的非常简单的 API，这不同于 PyCMethod。同样的机制也适用于通过 PyGetSetDef 定义的读取方法和设置方法。

要在这些场景下访问模块状态，请使用 PyType\_GetModuleByDef() 函数，并传入模块定义。一旦你得到该模块，即可调用 PyModule\_GetState() 来获取状态：

```
PyObject *module = PyType_GetModuleByDef(Py_TYPE(self), &module_def);
my_struct *state = (my_struct*)PyModule_GetState(module);
if (state == NULL) {
    return NULL;
}
```



`PyType_GetModuleByDef()` 的作用方式是通过搜索 `method resolution order` (即所有超类) 来找到具有相应模块的第一个超类。

---

**备注：**在非常特别的情况下（继承链跨越由同样定义创建的多个模块），`PyType_GetModuleByDef()` 可能不会返回真正定义方法的类。但是，它总是会返回一个具有同样定义的模块，这将确保具有兼容的 C 内存布局。

---

## 4.7 模块状态的生命期

当一个模块对象被当作垃圾回收时，它的模块状态将被释放。对于每个指向（一部分）模块状态的指针来说，你必须持有一个对模块对象的引用。

通常这不会有问题，因为使用 `PyType_FromModuleAndSpec()` 创建的类型，以及它们的实例，都持有对模块的引用。但是，当你从其他地方，例如对外部库的回调引用模块状态时必须小心谨慎。

## 5 未解决的问题

围绕模块级状态和堆类型仍然存在一些未解决的问题。

改善此状况最好的讨论是在 [capi-sig 邮件列表](#) 进行的。

### 5.1 类级作用域

目前（即 Python 3.11）还无法将状态关联到单个类型而不依赖于 CPython 实现细节（这在未来可能发生改变—或许，会怪异地允许采用适当的类级作用域解决方案）。

### 5.2 无损转换为堆类型

堆类型 API 没有从静态类型进行“无损”转换的设计；所谓无损转换，就是创建与给定静态类型完全一致的类型。