The Python/C API

发行版本 *3.13.6*

Guido van Rossum and the Python development team

八月 08, 2025

Python Software Foundation
Email: docs@python.org

Contents

本手册描述了希望编写扩展模块并将 Python 解释器嵌入其应用程序中的 C 和 C++ 程序员可用的 API。同时可以参阅 extending-index，其中描述了扩展编写的一般原则，但没有详细描述 API 函数。

CHAPTER 1

概述

Python 的应用编程接口（API）使得 C 和 C++ 程序员可以在多个层级上访问 Python 解释器。该 API 在 C++ 中同样可用，但为简化描述，通常将其称为 Python/C API。使用 Python/C API 有两个基本的理由。第一个理由是为了特定目的而编写 *扩展模块*；它们是扩展 Python 解释器功能的 C 模块。这可能是最常见的使用场景。第二个理由是将 Python 用作更大规模应用的组件；这种技巧通常被称为在一个应用中 *embedding* Python。

编写扩展模块的过程相对来说更易于理解，可以通过“菜谱”的形式分步骤介绍。使用某些工具可在一定程度上自动化这一过程。虽然人们在其他应用中嵌入 Python 的做法早已有之，但嵌入 Python 的过程没有编写扩展模块那样方便直观。

许多 API 函数在你嵌入或是扩展 Python 这两种场景下都能发挥作用；此外，大多数嵌入 Python 的应用程序也需要提供自定义扩展，因此在尝试在实际应用中嵌入 Python 之前先熟悉编写扩展应该会是个好主意。

1.1 代码标准

如果你想要编写可包含于 CPython 的 C 代码，你 **必须**遵循在 **PEP 7** 中定义的指导原则和标准。这些指导原则适用于任何你所要扩展的 Python 版本。在编写你自己的第三方扩展模块时可以不必遵循这些规范，除非你准备在日后向 Python 贡献这些模块。

1.2 包含文件

使用 Python/C API 所需要的全部函数、类型和宏定义可通过下面这行语句包含到你的代码之中：

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

这意味着包含以下标准头文件：`<stdio.h>`，`<string.h>`，`<errno.h>`，`<limits.h>`，`<assert.h>` 和 `<stdlib.h>`（如果可用）。

备注

由于 Python 可能会定义一些能在某些系统上影响标准头文件的预处理器定义，因此在包含任何标准头文件之前，你 必须先包含 `Python.h`。

推荐总是在 Python.h 前定义 PY_SSIZE_T_CLEAN 。查看解析参数并构建值变量 来了解这个宏的更多内容。

Python.h 所定义的全部用户可见名称（由包含的标准头文件所定义的除外）都带有前缀 Py 或者 _Py。以 _Py 打头的名称是供 Python 实现内部使用的，不应被扩展编写者使用。结构成员名称没有保留前缀。

备注

用户代码永远不应该定义以 Py 或 _Py 开头的名称。这会使读者感到困惑，并危及用户代码对未来 Python 版本的可移植性，这些版本可能会定义以这些前缀之一开头的其他名称。

头文件通常会与 Python 一起安装。在 Unix 上，它们位于 *prefix*/include/pythonversion/ 和 *exec_prefix*/include/pythonversion/ 目录，其中 prefix 和 exec_prefix 是由向 Python 的 **configure** 脚本传入的对应形参定义，而 *version* 则为 '%d.%d' % sys.version_info[:2]。在 Windows 上，头文件安装于 *prefix*/include，其中 prefix 是为安装程序指定的安装目录。

要包括这些头文件，请将两个目录（如果不同）都放到你所用编译器用于包括头文件的搜索目录中。请不要将父目录放入搜索路径然后使用 #include <pythonX.Y/Python.h>；这将使得多平台编译不可用，因为 prefix 下与平台无关的头文件包括了来自 exec_prefix 的平台专属头文件。

C++ 用户应该注意，尽管 API 是完全使用 C 来定义的，但头文件正确地将入口点声明为 extern "C"，因此 API 在 C++ 中使用此 API 不必再做任何特殊处理。

1.3 有用的宏

Python 头文件中定义了一些有用的宏。许多是在靠近它们被使用的地方定义的（例如*Py_RETURN_NONE*）。其他更为通用的则定义在这里。这里所显示的并不是一个完整的列表。

PyMODINIT_FUNC

声明扩展模块 PyInit 初始化函数。函数返回类型为 *PyObject**。该宏声明了平台所要求的任何特殊链接声明，并针对 C++ 将函数为声明为 extern "C"。

初始化函数必须命名为 PyInit_*name*，其中 *name* 是模块名称，并且应为在模块文件中定义的唯一非 static 项。例如:

```
static struct PyModuleDef spam_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "spam",
    ...
};

PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModuleDef_Init(&spam_module);
}
```

Py_ABS(x)

返回 x 的绝对值。

Added in version 3.3.

Py_ALWAYS_INLINE

让编译器始终内联静态的内联函数。编译器可以忽略它并决定不内联该函数。

它可被用来在禁用函数内联的调试模式下构建 Python 时内联严重影响性能的静态内联函数。例如，MSC 在调试模式下构建时就禁用了函数内联。

随意使用 Py_ALWAYS_INLINE 标记内联函数可能导致极差的性能（例如由于增加了代码量）。对于成本/收益分析来说计算机通常都比开发者更聪明。

如果 Python 是 在调试模式下构建的 (即定义了 *Py_DEBUG* 宏)，则 *Py_ALWAYS_INLINE* 宏将不做任何事情。

它必须在函数返回类型之前指明。用法:

```
static inline Py_ALWAYS_INLINE int random(void) { return 4; }
```

Added in version 3.11.

Py_CHARMASK(c)

参数必须为 [-128, 127] 或 [0, 255] 范围内的字符或整数类型。这个宏将 `c` 强制转换为 `unsigned char` 返回。

Py_DEPRECATED(version)

弃用声明。该宏必须放置在符号名称前。

示例:

```
Py_DEPRECATED(3.8) PyAPI_FUNC(int) Py_OldFunction(void);
```

在 3.8 版本发生变更: 添加了 MSVC 支持。

Py_GETENV(s)

与 `getenv(s)` 类似，但是如果从命令行传入了 `-E` 则返回 `NULL` (参见 *PyConfig.use_environment*)。

Py_MAX(x, y)

返回 `x` 和 `y` 当中的最大值。

Added in version 3.3.

Py_MEMBER_SIZE(type, member)

返回结构 (`type`) `member` 的大小，以字节表示。

Added in version 3.6.

Py_MIN(x, y)

返回 `x` 和 `y` 当中的最小值。

Added in version 3.3.

Py_NO_INLINE

启用内联某个函数。例如，它会减少 C 栈消耗：适用于大量内联代码的 LTO+PGO 编译版 (参见 bpo-33720)。

用法：

```
Py_NO_INLINE static int random(void) { return 4; }
```

Added in version 3.11.

Py_STRINGIFY(x)

将 `x` 转换为 C 字符串。例如 `Py_STRINGIFY(123)` 返回 `"123"`。

Added in version 3.4.

Py_UNREACHABLE()

这个可以在你有一个设计上无法到达的代码路径时使用。例如，当一个 `switch` 语句中所有可能的值都已被 `case` 子句覆盖了，就可将其用在 `default:` 子句中。当你非常想在某个位置放一个 `assert(0)` 或 `abort()` 调用时也可以用这个。

在 release 模式下，该宏帮助编译器优化代码，并避免发出不可到达代码的警告。例如，在 GCC 的 release 模式下，该宏使用 `__builtin_unreachable()` 实现。

`Py_UNREACHABLE()` 的一个用法是调用一个不会返回，但却没有声明 `_Py_NO_RETURN` 的函数之后。

如果一个代码路径不太可能是正常代码，但在特殊情况下可以到达，就不能使用该宏。例如，在低内存条件下，或者一个系统调用返回超出预期范围值，诸如此类，最好将错误报告给调用者。如果无法将错误报告给调用者，可以使用*Py_FatalError()*。

Added in version 3.7.

Py_UNUSED(arg)

用于函数定义中未使用的参数，从而消除编译器警告。例如：`int func(int a, int Py_UNUSED(b)) { return a; }`。

Added in version 3.4.

PyDoc_STRVAR(name, str)

创建一个可以在文档字符串中使用的，名字为 `name` 的变量。如果不和文档字符串一起构建 Python，该值将为空。

如 **PEP 7** 所述，使用*PyDoc_STRVAR* 作为文档字符串，以支持不和文档字符串一起构建 Python 的情况。

示例:

```
PyDoc_STRVAR(pop_doc, "Remove and return the rightmost element.");

static PyMethodDef deque_methods[] = {
    // ...
    {"pop", (PyCFunction)deque_pop, METH_NOARGS, pop_doc},
    // ...
}
```

PyDoc_STR(str)

为给定的字符串输入创建一个文档字符串，或者当文档字符串被禁用时，创建一个空字符串。

如 **PEP 7** 所述，使用*PyDoc_STR* 指定文档字符串，以支持不和文档字符串一起构建 Python 的情况。

示例:

```
static PyMethodDef pysqlite_row_methods[] = {
    {"keys", (PyCFunction)pysqlite_row_keys, METH_NOARGS,
        PyDoc_STR("Returns the keys of the row.")},
    {NULL, NULL}
};
```

1.4 对象、类型和引用计数

多数 Python/C API 函数都有一个或多个参数以及一个 *PyObject** 类型的返回值。这种类型是指向任意 Python 对象的不透明数据类型的指针。由于所有 Python 对象类型在大多数情况下都被 Python 语言用相同的方式处理（例如，赋值、作用域规则和参数传递等），因此用单个 C 类型来表示它们是很适宜的。几乎所有 Python 对象都存在于堆中：你不可声明一个类型为*PyObject* 的自动或静态的变量，只能声明类型为 *PyObject** 的指针变量。唯一的例外是 type 对象；因为这种对象永远不能被释放，所以它们通常都是静态的*PyTypeObject* 对象。

所有 Python 对象（甚至 Python 整数）都有一个 *type* 和一个 *reference count*。对象的类型确定它是什么类型的对象（例如整数、列表或用户定义函数；还有更多，如 types 中所述）。对于每个众所周知的类型，都有一个宏来检查对象是否属于该类型；例如，当（且仅当）*a* 所指的对象是 Python 列表时 `PyList_Check(a)` 为真。

1.4.1 引用计数

引用计数之所以重要是因为现有计算机的内存大小是有限的（并且往往限制得很严格）；它会计算有多少不同的地方对一个对象进行了*strong reference*。这些地方可以是另一个对象，也可以是全局（或静态）C 变量，或是某个 C 函数中的局部变量。当某个对象的最后一个*strong reference* 被释放时（即其引用计数变为零），该对象就会被取消分配。如果该对象包含对其他对象的引用，则会释放这些引用。如果不再有对

其他对象的引用，这些对象也会同样地被取消分配，依此类推。(在这里对象之间的相互引用显然是个问题；目前的解决办法，就是“不要这样做”。)

对于引用计数总是会显式地执行操作。通常的做法是使用*Py_INCREF()* 宏来获取对象的新引用（即让引用计数加一），并使用*Py_DECREF()* 宏来释放引用（即让引用计数减一）。*Py_DECREF()* 宏比 incref 宏复杂得多，因为它必须检查引用计数是否为零然后再调用对象的释放器。释放器是一个函数指针，它包含在对象的类型结构体中。如果对象是复合对象类型，如列表，则特定于类型的释放器会负责释放对象中包含的其他对象的引用，并执行所需的其他终结化操作。引用计数不会发生溢出；用于保存引用计数的位数至少会与虚拟内存中不同内存位置的位数相同 (假设 `sizeof(Py_ssize_t) >= sizeof(void*)`)。因此，引用计数的递增是一个简单的操作。

没有必要为每个包含指向对象指针的局部变量持有*strong reference* (即增加引用计数)。理论上说，当变量指向对象时对象的引用计数就会加一，而当变量离开其作用域时引用计数就会减一。不过，这两种情况会相互抵消，所以最后引用计数并没有改变。使用引用计数的唯一真正原因在于只要我们的变量指向对象就可以防止对象被释放。只要我们知道至少还有一个指向某对象的引用与我们的变量同时存在，就没有必要临时获取一个新的*strong reference* (即增加引用计数)。出现引用计数增加的一种重要情况是对象作为参数被传递给扩展模块中的 C 函数而这些函数又在 Python 中被调用；调用机制会保证在调用期间对每个参数持有一个引用。

然而，一个常见的陷阱是从列表中提取对象并在不获取新引用的情况下将其保留一段时间。某个其他操作可能在无意中从列表中移除该对象，释放这个引用，并可能撤销分配其资源。真正的危险在于看似无害的操作可能会唤起任意的 Python 代码来做这件事；有一条代码路径允许控制权从*Py_DECREF()* 流回到用户，因此几乎任何操作都有潜在的危险。

安全的做法是始终使用泛型操作（名称以 `PyObject_`, `PyNumber_`, `PySequence_` 或 `PyMapping_` 开头的函数)。这些操作总是为其返回的对象创建一个新的*strong reference* (即增加引用计数)。这使得调用者有责任在获得结果之后调用*Py_DECREF()*；这种做法很快就能习惯成自然。

引用计数细节

Python/C API 中函数的引用计数最好是使用 *引用所有权* 来解释。所有权是关联到引用，而不是对象（对象不能被拥有：它们总是会被共享)。“拥有一个引用”意味着当不再需要该引用时必须在其上调用 Py_DECREF。所有权也可以被转移，这意味着接受该引用所有权的代码在不再需要它时必须通过调用*Py_DECREF()* 或*Py_XDECREF()* 来最终释放它 --- 或是继续转移这个责任（通常是转给其调用方)。当一个函数将引用所有权转给其调用方时，则称调用方收到一个 *新的* 引用。当未转移所有权时，则称调用方是 *借入* 这个引用。对于*borrowed reference* 来说不需要任何额外操作。

相反地，当调用方函数传入一个对象的引用时，存在两种可能：该函数 *窃取* 了一个对象的引用，或是没有窃取。*窃取引用* 意味着当你向一个函数传入引用时，该函数会假定它拥有该引用，而你将不再对它负有责任。

很少有函数会窃取引用；两个重要的例外是*PyList_SetItem()* 和*PyTuple_SetItem()*，它们会窃取对条目的引用（但不是条目所在的元组或列表!)。这些函数被设计为会窃取引用是因为在使用新创建的对象来填充元组或列表时有一个通常的惯例；例如，创建元组 `(1, 2, "three")` 的代码看起来可以是这样的（暂时不要管错误处理；下面会显示更好的代码编写方式)：

```
PyObject *t;

t = PyTuple_New(3);
PyTuple_SetItem(t, 0, PyLong_FromLong(1L));
PyTuple_SetItem(t, 1, PyLong_FromLong(2L));
PyTuple_SetItem(t, 2, PyUnicode_FromString("three"));
```

在这里，*PyLong_FromLong()* 返回了一个新的引用并且它立即被*PyTuple_SetItem()* 所窃取。当你想要继续使用一个对象而对它的引用将被窃取时，请在调用窃取引用的函数之前使用*Py_INCREF()* 来抓取另一个引用。

顺便提一下，*PyTuple_SetItem()* 是设置元组条目的唯一方式；*PySequence_SetItem()* 和*PyObject_SetItem()* 会拒绝这样做因为元组是不可变数据类型。你应当只对你自己创建的元组使用*PyTuple_SetItem()*。

等价于填充一个列表的代码可以使用*PyList_New()* 和*PyList_SetItem()* 来编写。

然而，在实践中，你很少会使用这些创建和填充元组或列表的方式。有一个通用的函数*Py_BuildValue()*可以根据 C 值来创建大多数常用对象，由一个 格式字符串来指明。例如，上面的两个代码块可以用下面的代码来代替（还会负责错误检测）：

```
PyObject *tuple, *list;

tuple = Py_BuildValue("(iis)", 1, 2, "three");
list = Py_BuildValue("[iis]", 1, 2, "three");
```

在对条目使用*PyObject_SetItem()* 等操作时更常见的做法是只借入引用，比如将参数传递给你正在编写的函数。在这种情况下，它们在引用方面的行为更为清晰，因为你不必为了把引用转走而获取一个新的引用（“让它被偷取”）。例如，这个函数将列表（实际上是任何可变序列）中的所有条目都设为给定的条目：

```
int
set_all(PyObject *target, PyObject *item)
{
    Py_ssize_t i, n;

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    for (i = 0; i < n; i++) {
        PyObject *index = PyLong_FromSsize_t(i);
        if (!index)
            return -1;
        if (PyObject_SetItem(target, index, item) < 0) {
            Py_DECREF(index);
            return -1;
        }
        Py_DECREF(index);
    }
    return 0;
}
```

对于函数返回值的情况略有不同。虽然向大多数函数传递一个引用不会改变你对该引用的所有权责任，但许多返回一个引用的函数会给你该引用的所有权。原因很简单：在许多情况下，返回的对象是临时创建的，而你得到的引用是对该对象的唯一引用。因此，返回对象引用的通用函数，如*PyObject_GetItem()* 和*PySequence_GetItem()*，将总是返回一个新的引用（调用方将成为该引用的所有者）。

一个需要了解的重点在于你是否拥有一个由函数返回的引用只取决于你所调用的函数 --- 附带物 (作为参数传给函数的对象的类型) 不会带来额外影响！因此，如果你使用*PyList_GetItem()* 从一个列表提取条目，你并不会拥有其引用 --- 但是如果你使用*PySequence_GetItem()* (它恰好接受完全相同的参数) 从同一个列表获取同样的条目，你就会拥有一个对所返回对象的引用。

下面是说明你要如何编写一个函数来计算一个整数列表中条目的示例；一个是使用*PyList_GetItem()*，而另一个是使用*PySequence_GetItem()*。

```
long
sum_list(PyObject *list)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;

    n = PyList_Size(list);
    if (n < 0)
        return -1; /* Not a list */
    for (i = 0; i < n; i++) {
        item = PyList_GetItem(list, i); /* 不能失败 */
        if (!PyLong_Check(item)) continue; /* 跳过非整数 */
        value = PyLong_AsLong(item);
```

(续下页)

(接上页)

```
        if (value == -1 && PyErr_Occurred())
            /* 太大的整数无法适应 C long 类型，放弃 */
            return -1;
        total += value;
    }
    return total;
}
```

```
long
sum_sequence(PyObject *sequence)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;
    n = PySequence_Length(sequence);
    if (n < 0)
        return -1; /* 没有长度 */
    for (i = 0; i < n; i++) {
        item = PySequence_GetItem(sequence, i);
        if (item == NULL)
            return -1; /* 不是序列，或其他错误 */
        if (PyLong_Check(item)) {
            value = PyLong_AsLong(item);
            Py_DECREF(item);
            if (value == -1 && PyErr_Occurred())
                /* 太大的整数无法适应 C long 类型，放弃 */
                return -1;
            total += value;
        }
        else {
            Py_DECREF(item); /* 丢弃引用所有权 */
        }
    }
    return total;
}
```

1.4.2 类型

在 Python/C API 中扮演重要角色的其他数据类型很少；大多为简单 C 类型如 `int`, `long`, `double` 和 `char*` 等。有一些结构类型被用来燃烧液体于列出模块所导出的函数或者某个新对象类型的个的一个，还有一个结构类型被用来描述复数的值。这些结构类型将与使用它们的函数放到一起讨论。

type **Py_ssize_t**

属于稳定 ABI. 一个使得 `sizeof(Py_ssize_t) == sizeof(size_t)` 的有符号整数类型。C99 没有直接定义这样的东西（size_t 是一个无符号整数类型）。请参阅 **PEP 353** 了解详情。`PY_SSIZE_T_MAX` 是*Py_ssize_t* 类型的最大正数值。

1.5 异常

Python 程序员只需要处理特定需要处理的错误异常；未处理的异常会自动传递给调用者，然后传递给调用者的调用者，依此类推，直到他们到达顶级解释器，在那里将它们报告给用户并伴随堆栈回溯。

然而，对于 C 程序员来说，错误检查必须总是显式进行的。Python/C API 中的所有函数都可以引发异常，除非在函数的文档中另外显式声明。一般来说，当一个函数遇到错误时，它会设置一个异常，丢弃它所拥有的任何对象引用，并返回一个错误标示。如果没有说明例外的文档，这个标示将为 `NULL` 或 `-1`，具体取决于函数的返回类型。有少量函数会返回一个布尔真/假结果值，其中假值表示错误。有极少的函数没有显式的错误标示或是具有不明确的返回值，并需要用*PyErr_Occurred()* 来进行显式的检测。这些例外总是会被明确地记入文档中。

异常状态是在各个线程的存储中维护的（这相当于在一个无线程的应用中使用全局存储）。一个线程可以处在两种状态之一：异常已经发生，或者没有发生。函数*PyErr_Occurred()* 可以被用来检查此状态：当异常发生时它将返回一个借入的异常类型对象的引用，在其他情况下则返回 NULL。有多个函数可以设置异常状态：*PyErr_SetString()* 是最常见的（尽管不是最通用的）设置异常状态的函数，而*PyErr_Clear()* 可以清除异常状态。

完整的异常状态由三个对象组成 (它为都可以为 NULL): 异常类型、相应的异常值，以及回溯信息。这些对象的含义与 Python 中 sys.exc_info() 的结果相同；然而，它们并不是一样的：Python 对象代表由 Python try ... except 语句所处理的最后一个异常，而 C 层级的异常状态只在异常被传入到 C 函数或在它们之间传递时存在直至其到达 Python 字节码解释器的主事件循环，该事件循环会负责将其转移至 sys.exc_info() 等处。

请注意自 Python 1.5 开始，从 Python 代码访问异常状态的首选的、线程安全的方式是调用函数 sys.exc_info()，它将返回 Python 代码的分线程异常状态。此外，这两种访问异常状态的方式的语义都发生了变化因而捕获到异常的函数将保存并恢复其线程的异常状态以保留其调用方的异常状态。这将防止异常处理代码中由一个看起来很无辜的函数覆盖了正在处理的异常所造成的常见错误；它还减少了在回溯由栈帧所引用的对象的往往不被需要的生命其延长。

作为一般的原则，一个调用另一个函数来执行某些任务的函数应当检查被调用的函数是否引发了异常，并在引发异常时将异常状态传递给其调用方。它应当丢弃它所拥有的任何对象引用，并返回一个错误标示，但它 不应设置另一个异常 --- 那会覆盖刚引发的异常，并丢失有关错误确切原因的重要信息。

上面的 sum_sequence() 示例是一个检测异常并将其传递出去的简单例子。碰巧的是这个示例在检测到错误时不需要清理所拥有的任何引用。下面的示例函数展示了一些错误清理操作。首先，为了提醒你 Python 的受欢迎程度，我们展示了等价的 Python 代码:

```
def incr_item(dict, key):
    try:
        item = dict[key]
    except KeyError:
        item = 0
    dict[key] = item + 1
```

对应的 C 代码如下:

```
int
incr_item(PyObject *dict, PyObject *key)
{
    /* 对象全部初始化为 NULL 用于 Py_XDECREF */
    PyObject *item = NULL, *const_one = NULL, *incremented_item = NULL;
    int rv = -1; /* 返回值初始化为 -1（失败） */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* 只处理 KeyError: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;

        /* 清除错误并使用零: */
        PyErr_Clear();
        item = PyLong_FromLong(0L);
        if (item == NULL)
            goto error;
    }
    const_one = PyLong_FromLong(1L);
    if (const_one == NULL)
        goto error;

    incremented_item = PyNumber_Add(item, const_one);
    if (incremented_item == NULL)
        goto error;

    if (PyObject_SetItem(dict, key, incremented_item) < 0)
```

(续下页)

(接上页)

```
        goto error;
    rv = 0; /* 成功 */
    /* 继续执行清理代码 */

error:
    /* 清理代码，由成功和失败路径所共享 */

    /* 使用 Py_XDECREF() 以忽略 NULL 引用 */
    Py_XDECREF(item);
    Py_XDECREF(const_one);
    Py_XDECREF(incremented_item);

    return rv; /* -1 表示错误，0 表示成功 */
}
```

这个例子代表了 C 语言中 goto 语句一种受到认可的用法！它说明了如何使用*PyErr_ExceptionMatches()* 和*PyErr_Clear()* 来处理特定的异常，以及如何使用*Py_XDECREF()* 来处理可能为 NULL 的自有引用（注意名称中的 'X'；*Py_DECREF()* 在遇到 NULL 引用时将会崩溃）。重要的一点在于用来保存自有引用的变量要被初始化为 NULL 才能发挥作用；类似地，建议的返回值也要被初始化为 -1 (失败) 并且只有在最终执行的调用成功后才会被设置为成功。

1.6 嵌入 Python

只有 Python 解释器的嵌入方（相对于扩展编写者而言）才需要担心的一项重要任务是它的初始化，可能还有它的最终化。解释器的大多数功能只有在解释器被初始化之后才能被使用。

基本的初始化函数是*Py_Initialize()*。此函数将初始化已加载模块表，并创建基本模块 builtins，__main__ 和 sys。它还将初始化模块搜索路径 (sys.path)。

Py_Initialize() 不会设置” 脚本参数列表” (sys.argv)。如果稍后将要执行的 Python 代码需要此变量，则要设置*PyConfig.argv* 并且还要设置*PyConfig.parse_argv*: 参见*Python 初始化配置*。

在大多数系统上（特别是 Unix 和 Windows，虽然在细节上有所不同），*Py_Initialize()* 将根据对标准 Python 解释器可执行文件的位置的最佳猜测来计算模块搜索路径，并设定 Python 库可在相对于 Python 解释器可执行文件的固定位置上找到。特别地，它将相对于在 shell 命令搜索路径 (环境变量 PATH) 上找到的名为 python 的可执行文件所在父目录中查找名为 lib/python*X.Y* 的目录。

举例来说，如果 Python 可执行文件位于 /usr/local/bin/python，它将假定库位于 /usr/local/lib/python*X.Y*。(实际上，这个特定路径还将成为 “回退” 位置，会在当无法在 PATH 中找到名为 python 的可执行文件时被使用。) 用户可以通过设置环境变量 PYTHONHOME，或通过设置 PYTHONPATH 在标准路径之前插入额外的目录来覆盖此行为。

嵌入的应用程序可以通过在调用*Py_InitializeFromConfig()* 之前设置*PyConfig.program_name* 来调整搜索。请注意 PYTHONHOME 仍然会覆盖此设置并且 PYTHONPATH 仍然会被插入到标准路径之前。需要完整控制权的应用程序必须提供它自己的*Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()* 和*Py_GetProgramFullPath()* 实现（这些函数均在 Modules/getpath.c 中定义）。

有时，还需要对 Python 进行 “反初始化”。例如，应用程序可能想要重新启动 (再次调用*Py_Initialize()*) 或者应用程序对 Python 的使用已经完成并想要释放 Python 所分配的内存。这可以通过调用*Py_FinalizeEx()* 来实现。如果当前 Python 处于已初始化状态则*Py_IsInitialized()* 函数将返回真值。有关这些函数的更多信息将在之后的章节中给出。请注意*Py_FinalizeEx()* 不会释放所有由 Python 解释器所分配的内存，例如由扩展模块所分配的内存目前是不会被释放的。

1.7 调试构建

Python 可以附带某些宏来编译以启用对解释器和扩展模块的额外检查。这些检查会给运行时增加大量额外开销因此它们默认未被启用。

各种调试构建版的完整列表见 Python 源代码颁发包中的 Misc/SpecialBuilds.txt。可用的构建版有支持追踪引用计数，调试内存分配器，或是对主解释器事件循环的低层级性能分析等等。本节的剩余部分

将只介绍最常用的几种构建版。

Py_DEBUG

在定义了 Py_DEBUG 宏的情况下编译解释器将产生通常所称的 Python 调试构建版。Py_DEBUG 在 Unix 编译版中是通过添加 --with-pydebug 到 ./configure 命令来启用的。它也可以通过提供非 Python 专属的 _DEBUG 宏来启用。当 Py_DEBUG 在 Unix 编译版中启用时，编译器优化将被禁用。

除了下文描述的引用计数调试，还会执行额外检查，请参阅 Python Debug Build。

定义 Py_TRACE_REFS 将启用引用追踪 (参见 configure --with-trace-refs 选项)。当定义了此宏时，将通过在每个*PyObject* 上添加两个额外字段来维护一个活动对象的循环双链列表。总的分配量也会被追踪。在退出时，所有现存的引用将被打印出来。(在交互模式下这将在解释器运行每条语句之后发生)。

有关更多详细信息，请参阅 Python 源代码中的 Misc/SpecialBuilds.txt 。

1.8 推荐的第三方工具

下列第三方工具提供了为 Python 创建 C, C++ 和 Rust 扩展的更简单或更复杂的方式：

- Cython
- cffi
- HPy
- nanobind (C++)
- Numba
- pybind11 (C++)
- PyO3 (Rust)
- SWIG

使用这些工具可以避免编写与特定版本的 CPython 紧密绑定的代码，避免引用计数错误，并能更多地关注你自己的代码而不是关注如何使用 CPython API。总的来说，Python 的新版本可通过更新此类工具来获得支持，并且你的代码通常都将自动使用更新且更高效的 API。有些工具还支持基于单个源代码集针对其他 Python 实现进行编译。

这些项目并不是由维护 Python 的同一批人提供支持的，程序相关的问题需要直接向项目提出。请记得检查项目是否仍然获得维护与支持，因为上面的列表可能会变得过时。

参见

Python Packaging User Guide: Binary Extensions
"Python Packaging User Guide" 不仅涵盖了几个简化二进制扩展创建的可用工具，还讨论了为什么首先创建扩展模块的各种原因。

CHAPTER 2

C API 的稳定性

除非另有文档说明，Python 的 C API 将遵循 **PEP 387** 所描述的向下兼容策略。对它的大部分改变都是源代码级兼容的（通常只会增加新的 new API）。改变现有 API 或移除 API 只会在弃用期结束之后或需修复严重问题时才会发生。

CPython 的应用程序二进制接口（ABI）可以跨微版本向上和向下兼容（在以相同方式编译的情况下，参见下文*平台的考虑*一节）。因此，针对 Python 3.10.0 编译的代码将适用于 3.10.8，反之亦然，但对于 3.9.x 和 3.11.x 则需要单独编译。

存在具有不同稳定性预期的两个 C API 层次：

- *不稳定 API*，可能在次要版本中发生改变而没有弃用期。它的名称会以 `PyUnstable` 前缀来标记。
- *受限 API*，将会在多个次要版本间保持兼容。当定义了 `Py_LIMITED_API` 时，将只有这个子集会从 `Python.h` 对外公开。

这些将在下文中更详细地讨论。

带有一个下划线前缀的名称，如 `_Py_InternalState`，是可能不经通知就改变甚至是在补丁发布版中改变的私有 API。如果你需要使用这样的 API，请考虑联系 CPython 开发团队 来讨论为你的应用场景添加公有 API。

2.1 不稳定 C API

任何名称带有 `PyUnstable` 前缀的 API 都将对外公开 CPython 的实现细节，并可能不加弃用警告即在次要版本中发生改变（例如从 3.9 到 3.10）。但是，它不会在问题修正发布版中改变（例如从 3.10.0 到 3.10.1）。

它通常是针对专门的，低层级的工具如调试器等。

使用此 API 的项目需要跟随 CPython 开发进程并花费额外的努力来适应改变。

2.2 应用程序二进制接口的稳定版

简单起见，本文档只讨论了 *扩展*，但受限 API 和稳定 ABI 对于 API 的所有用法都能发挥相同的作用–例如嵌入版的 Python 等。

2.2.1 受限 C API

Python 3.2 引入了 *受限 API*，它是 Python 的 C API 的子集。只使用受限 API 的扩展可以一次编译即可在多个 Python 版本上加载。受限 API 内容如下所示。

Py_LIMITED_API

请在包括 `Python.h` 之前定义这个宏以选择只使用受限 API，并选择受限 API 的版本。

将 `Py_LIMITED_API` 定义为对应于你的扩展所支持的最低 Python 版本的 *PY_VERSION_HEX* 值。扩展将与从指定版本开始的所有 Python 3 发布版保持 ABI 兼容，并可使用到该版本为止所引入的受限 API。

不直接使用 `PY_VERSION_HEX` 宏，而是硬编码一个最小的次要版本（例如 `0x030A0000` 表示 Python 3.10）以便在使用未来的 Python 版本进行编译时保持稳定。

你还可以将 `Py_LIMITED_API` 定义为 `3`。其效果与 `0x03020000` 相同（即 Python 3.2，引入受限 API 的版本）。

2.2.2 稳定 ABI

为启用此特性，Python 提供了一个 *稳定 ABI*: 即一组将跨 Python 3.x 各个版本保持 ABI 兼容的符号集合。

备注

稳定 ABI 将防止多种 ABI 问题，如由于缺失符号导致的链接器错误或由于结构体布局或函数签名中的变化导致的数据损坏。不过，Python 中的其他修改可能改变扩展的 *行为*。请参阅 Python 的向下兼容策略 (**PEP 387**) 了解详情。

稳定 ABI 包含在 *受限 API* 中对外公开的符号，但还包含其他符号–例如，为支持旧版本受限 API 所需的函数。

在 Windows 上，使用稳定 ABI 的扩展应当被链接到 `python3.dll` 而不是版本专属的库如 `python39.dll`。

在某些平台上，Python 将查找并载入名称中带有 `abi3` 标签的共享库文件 (例如 `mymodule.abi3.so`)。它不会检查这样的扩展是否兼容稳定 ABI。使用方 (或其打包工具) 需要确保这一些，例如，基于 3.10+ 受限 API 编译的扩展不可被安装于更低版本的 Python 中。

稳定 ABI 中的所有函数都会作为 Python 的共享库中的函数存在，而不仅是作为宏。这使得它们可以在不使用 C 预处理器的语言中使用。

2.2.3 受限 API 的作用域和性能

受限 API 的目标是允许使用在完整 C API 中可用的任何东西，但可能会有性能上的损失。

例如，虽然 *PyList_GetItem()* 是可用的，但其 “不安全的” 宏版本 *PyList_GET_ITEM()* 则是不可用的。这个宏的运行速度更快因为它可以利用版本专属的列表对象实现细节。

在未定义 `Py_LIMITED_API` 的情况下，某些 C API 函数将由宏来执行内联或替换。定义 `Py_LIMITED_API` 会禁用这样的内联，允许提升 Python 的数据结构稳定性，但有可能降低性能。

通过省略 `Py_LIMITED_API` 定义，可以使基于版本专属的 ABI 来编译受限 API 扩展成为可能。这能提升其在相应 Python 版本上的性能，但也将限制其兼容性。基于 `Py_LIMITED_API` 进行编译将产生一个可在版本专属扩展不可用的场合分发的扩展–例如，针对即将发布的 Python 版本的预发布包。

2.2.4 受限 API 警示

请注意使用 `Py_LIMITED_API` 进行编译 *无法完全保证代码能够兼容受限 API 或稳定 ABI*。`Py_LIMITED_API` 仅仅涵盖定义部分，但一个 API 还包括其他因素，如预期的语义等。

`Py_LIMITED_API` 不能处理的一个问题是附带在较低 Python 版本中无效的参数调用某个函数。例如，考虑一个接受 `NULL` 作为参数的函数。在 Python 3.9 中，`NULL` 现在会选择一个默认行为，但在 Python 3.8 中，该参数将被直接使用，导致一个 `NULL` 引用被崩溃。类似的参数也适用于结构体的字段。

另一个问题是当定义了 `Py_LIMITED_API` 时某些结构体字段目前不会被隐藏，即使它们是受限 API 的一部分。

出于这些原因，我们建议用要支持的 *所有* Python 小版本号来测试一个扩展，并最好是用其中 *最低的*版本来编译它。

我们还建议查看所使用 API 的全部文档以检查其是否显式指明为受限 API 的一部分。即使定义了 `Py_LIMITED_API`，少数私有声明还是会出于技术原因（或者甚至是作为程序缺陷在无意中）被暴露出来。

还要注意受限 API 并不必然是稳定的：在 Python 3.8 上用 `Py_LIMITED_API` 编译扩展意味着该扩展能在 Python 3.12 上运行，但它将不一定能用 Python 3.12 编译。特别地，在稳定 ABI 保持稳定的情况下，部分受限 API 可能会被弃用并被移除。

2.3 平台的考虑

ABI 的稳定性不仅取决于 Python，取决于所使用的编译器、低层级库和编译器选项等。对于*稳定 ABI* 的目标来说，这些细节定义了一个 “平台”。它们通常会依赖于 OS 类型和处理器架构等。

确保在特定平台上的所有 Python 版本都以不破坏稳定 ABI 的方式构建是每个特定 Python 分发方的责任。来自 `python.org` 以及许多第三方分发商的 Windows 和 macOS 发布版都必于这种情况。

2.4 受限 API 的内容

目前*受限 API* 包括下面这些项:

- *PY_VECTORCALL_ARGUMENTS_OFFSET*
- *PyAIter_Check()*
- *PyArg_Parse()*
- *PyArg_ParseTuple()*
- *PyArg_ParseTupleAndKeywords()*
- *PyArg_UnpackTuple()*
- *PyArg_VaParse()*
- *PyArg_VaParseTupleAndKeywords()*
- *PyArg_ValidateKeywordArguments()*
- *PyBaseObject_Type*
- *PyBool_FromLong()*
- *PyBool_Type*
- *PyBuffer_FillContiguousStrides()*
- *PyBuffer_FillInfo()*
- *PyBuffer_FromContiguous()*
- *PyBuffer_GetPointer()*
- *PyBuffer_IsContiguous()*
- *PyBuffer_Release()*
- *PyBuffer_SizeFromFormat()*
- *PyBuffer_ToContiguous()*
- PyByteArrayIter_Type

- *PyByteArray_AsString()*
- *PyByteArray_Concat()*
- *PyByteArray_FromObject()*
- *PyByteArray_FromStringAndSize()*
- *PyByteArray_Resize()*
- *PyByteArray_Size()*
- *PyByteArray_Type*
- PyBytesIter_Type
- *PyBytes_AsString()*
- *PyBytes_AsStringAndSize()*
- *PyBytes_Concat()*
- *PyBytes_ConcatAndDel()*
- PyBytes_DecodeEscape()
- *PyBytes_FromFormat()*
- *PyBytes_FromFormatV()*
- *PyBytes_FromObject()*
- *PyBytes_FromString()*
- *PyBytes_FromStringAndSize()*
- PyBytes_Repr()
- *PyBytes_Size()*
- *PyBytes_Type*
- *PyCFunction*
- *PyCFunctionFast*
- *PyCFunctionFastWithKeywords*
- *PyCFunctionWithKeywords*
- PyCFunction_GetFlags()
- PyCFunction_GetFunction()
- PyCFunction_GetSelf()
- *PyCFunction_New()*
- *PyCFunction_NewEx()*
- PyCFunction_Type
- *PyCMethod_New()*
- *PyCallIter_New()*
- *PyCallIter_Type*
- *PyCallable_Check()*
- *PyCapsule_Destructor*
- *PyCapsule_GetContext()*
- *PyCapsule_GetDestructor()*
- *PyCapsule_GetName()*

- *PyCapsule_GetPointer()*
- *PyCapsule_Import()*
- *PyCapsule_IsValid()*
- *PyCapsule_New()*
- *PyCapsule_SetContext()*
- *PyCapsule_SetDestructor()*
- *PyCapsule_SetName()*
- *PyCapsule_SetPointer()*
- PyCapsule_Type
- PyClassMethodDescr_Type
- *PyCodec_BackslashReplaceErrors()*
- *PyCodec_Decode()*
- *PyCodec_Decoder()*
- *PyCodec_Encode()*
- *PyCodec_Encoder()*
- *PyCodec_IgnoreErrors()*
- *PyCodec_IncrementalDecoder()*
- *PyCodec_IncrementalEncoder()*
- *PyCodec_KnownEncoding()*
- *PyCodec_LookupError()*
- *PyCodec_NameReplaceErrors()*
- *PyCodec_Register()*
- *PyCodec_RegisterError()*
- *PyCodec_ReplaceErrors()*
- *PyCodec_StreamReader()*
- *PyCodec_StreamWriter()*
- *PyCodec_StrictErrors()*
- *PyCodec_Unregister()*
- *PyCodec_XMLCharRefReplaceErrors()*
- *PyComplex_FromDoubles()*
- *PyComplex_ImagAsDouble()*
- *PyComplex_RealAsDouble()*
- *PyComplex_Type*
- *PyDescr_NewClassMethod()*
- *PyDescr_NewGetSet()*
- *PyDescr_NewMember()*
- *PyDescr_NewMethod()*
- PyDictItems_Type
- PyDictIterItem_Type

- PyDictIterKey_Type
- PyDictIterValue_Type
- PyDictKeys_Type
- *PyDictProxy_New()*
- PyDictProxy_Type
- PyDictRevIterItem_Type
- PyDictRevIterKey_Type
- PyDictRevIterValue_Type
- PyDictValues_Type
- *PyDict_Clear()*
- *PyDict_Contains()*
- *PyDict_Copy()*
- *PyDict_DelItem()*
- *PyDict_DelItemString()*
- *PyDict_GetItem()*
- *PyDict_GetItemRef()*
- *PyDict_GetItemString()*
- *PyDict_GetItemStringRef()*
- *PyDict_GetItemWithError()*
- *PyDict_Items()*
- *PyDict_Keys()*
- *PyDict_Merge()*
- *PyDict_MergeFromSeq2()*
- *PyDict_New()*
- *PyDict_Next()*
- *PyDict_SetItem()*
- *PyDict_SetItemString()*
- *PyDict_Size()*
- *PyDict_Type*
- *PyDict_Update()*
- *PyDict_Values()*
- *PyEllipsis_Type*
- PyEnum_Type
- *PyErr_BadArgument()*
- *PyErr_BadInternalCall()*
- *PyErr_CheckSignals()*
- *PyErr_Clear()*
- PyErr_Display()
- *PyErr_DisplayException()*

- *PyErr_ExceptionMatches()*
- *PyErr_Fetch()*
- *PyErr_Format()*
- *PyErr_FormatV()*
- *PyErr_GetExcInfo()*
- *PyErr_GetHandledException()*
- *PyErr_GetRaisedException()*
- *PyErr_GivenExceptionMatches()*
- *PyErr_NewException()*
- *PyErr_NewExceptionWithDoc()*
- *PyErr_NoMemory()*
- *PyErr_NormalizeException()*
- *PyErr_Occurred()*
- *PyErr_Print()*
- *PyErr_PrintEx()*
- PyErr_ProgramText()
- *PyErr_ResourceWarning()*
- *PyErr_Restore()*
- *PyErr_SetExcFromWindowsErr()*
- *PyErr_SetExcFromWindowsErrWithFilename()*
- *PyErr_SetExcFromWindowsErrWithFilenameObject()*
- *PyErr_SetExcFromWindowsErrWithFilenameObjects()*
- *PyErr_SetExcInfo()*
- *PyErr_SetFromErrno()*
- *PyErr_SetFromErrnoWithFilename()*
- *PyErr_SetFromErrnoWithFilenameObject()*
- *PyErr_SetFromErrnoWithFilenameObjects()*
- *PyErr_SetFromWindowsErr()*
- *PyErr_SetFromWindowsErrWithFilename()*
- *PyErr_SetHandledException()*
- *PyErr_SetImportError()*
- *PyErr_SetImportErrorSubclass()*
- *PyErr_SetInterrupt()*
- *PyErr_SetInterruptEx()*
- *PyErr_SetNone()*
- *PyErr_SetObject()*
- *PyErr_SetRaisedException()*
- *PyErr_SetString()*
- *PyErr_SyntaxLocation()*

- *PyErr_SyntaxLocationEx()*
- *PyErr_WarnEx()*
- *PyErr_WarnExplicit()*
- *PyErr_WarnFormat()*
- *PyErr_WriteUnraisable()*
- *PyEval_AcquireThread()*
- *PyEval_EvalCode()*
- *PyEval_EvalCodeEx()*
- *PyEval_EvalFrame()*
- *PyEval_EvalFrameEx()*
- *PyEval_GetBuiltins()*
- *PyEval_GetFrame()*
- *PyEval_GetFrameBuiltins()*
- *PyEval_GetFrameGlobals()*
- *PyEval_GetFrameLocals()*
- *PyEval_GetFuncDesc()*
- *PyEval_GetFuncName()*
- *PyEval_GetGlobals()*
- *PyEval_GetLocals()*
- *PyEval_InitThreads()*
- *PyEval_ReleaseThread()*
- *PyEval_RestoreThread()*
- *PyEval_SaveThread()*
- *PyExc_ArithmeticError*
- *PyExc_AssertionError*
- *PyExc_AttributeError*
- *PyExc_BaseException*
- *PyExc_BaseExceptionGroup*
- *PyExc_BlockingIOError*
- *PyExc_BrokenPipeError*
- *PyExc_BufferError*
- *PyExc_BytesWarning*
- *PyExc_ChildProcessError*
- *PyExc_ConnectionAbortedError*
- *PyExc_ConnectionError*
- *PyExc_ConnectionRefusedError*
- *PyExc_ConnectionResetError*
- *PyExc_DeprecationWarning*
- *PyExc_EOFError*

- *PyExc_EncodingWarning*
- *PyExc_EnvironmentError*
- *PyExc_Exception*
- *PyExc_FileExistsError*
- *PyExc_FileNotFoundError*
- *PyExc_FloatingPointError*
- *PyExc_FutureWarning*
- *PyExc_GeneratorExit*
- *PyExc_IOError*
- *PyExc_ImportError*
- *PyExc_ImportWarning*
- *PyExc_IndentationError*
- *PyExc_IndexError*
- *PyExc_InterruptedError*
- *PyExc_IsADirectoryError*
- *PyExc_KeyError*
- *PyExc_KeyboardInterrupt*
- *PyExc_LookupError*
- *PyExc_MemoryError*
- *PyExc_ModuleNotFoundError*
- *PyExc_NameError*
- *PyExc_NotADirectoryError*
- *PyExc_NotImplementedError*
- *PyExc_OSError*
- *PyExc_OverflowError*
- *PyExc_PendingDeprecationWarning*
- *PyExc_PermissionError*
- *PyExc_ProcessLookupError*
- *PyExc_RecursionError*
- *PyExc_ReferenceError*
- *PyExc_ResourceWarning*
- *PyExc_RuntimeError*
- *PyExc_RuntimeWarning*
- *PyExc_StopAsyncIteration*
- *PyExc_StopIteration*
- *PyExc_SyntaxError*
- *PyExc_SyntaxWarning*
- *PyExc_SystemError*
- *PyExc_SystemExit*

- *PyExc_TabError*
- *PyExc_TimeoutError*
- *PyExc_TypeError*
- *PyExc_UnboundLocalError*
- *PyExc_UnicodeDecodeError*
- *PyExc_UnicodeEncodeError*
- *PyExc_UnicodeError*
- *PyExc_UnicodeTranslateError*
- *PyExc_UnicodeWarning*
- *PyExc_UserWarning*
- *PyExc_ValueError*
- *PyExc_Warning*
- *PyExc_WindowsError*
- *PyExc_ZeroDivisionError*
- *PyExceptionClass_Name()*
- *PyException_GetArgs()*
- *PyException_GetCause()*
- *PyException_GetContext()*
- *PyException_GetTraceback()*
- *PyException_SetArgs()*
- *PyException_SetCause()*
- *PyException_SetContext()*
- *PyException_SetTraceback()*
- *PyFile_FromFd()*
- *PyFile_GetLine()*
- *PyFile_WriteObject()*
- *PyFile_WriteString()*
- PyFilter_Type
- *PyFloat_AsDouble()*
- *PyFloat_FromDouble()*
- *PyFloat_FromString()*
- *PyFloat_GetInfo()*
- *PyFloat_GetMax()*
- *PyFloat_GetMin()*
- *PyFloat_Type*
- *PyFrameObject*
- *PyFrame_GetCode()*
- *PyFrame_GetLineNumber()*
- *PyFrozenSet_New()*

- *PyFrozenSet_Type*
- *PyGC_Collect()*
- *PyGC_Disable()*
- *PyGC_Enable()*
- *PyGC_IsEnabled()*
- *PyGILState_Ensure()*
- *PyGILState_GetThisThreadState()*
- *PyGILState_Release()*
- PyGILState_STATE
- *PyGetSetDef*
- PyGetSetDescr_Type
- *PyImport_AddModule()*
- *PyImport_AddModuleObject()*
- *PyImport_AddModuleRef()*
- *PyImport_AppendInittab()*
- *PyImport_ExecCodeModule()*
- *PyImport_ExecCodeModuleEx()*
- *PyImport_ExecCodeModuleObject()*
- *PyImport_ExecCodeModuleWithPathnames()*
- *PyImport_GetImporter()*
- *PyImport_GetMagicNumber()*
- *PyImport_GetMagicTag()*
- *PyImport_GetModule()*
- *PyImport_GetModuleDict()*
- *PyImport_Import()*
- *PyImport_ImportFrozenModule()*
- *PyImport_ImportFrozenModuleObject()*
- *PyImport_ImportModule()*
- *PyImport_ImportModuleLevel()*
- *PyImport_ImportModuleLevelObject()*
- *PyImport_ImportModuleNoBlock()*
- *PyImport_ReloadModule()*
- *PyIndex_Check()*
- *PyInterpreterState*
- *PyInterpreterState_Clear()*
- *PyInterpreterState_Delete()*
- *PyInterpreterState_Get()*
- *PyInterpreterState_GetDict()*
- *PyInterpreterState_GetID()*

- *PyInterpreterState_New()*
- *PyIter_Check()*
- *PyIter_Next()*
- *PyIter_Send()*
- PyListIter_Type
- PyListRevIter_Type
- *PyList_Append()*
- *PyList_AsTuple()*
- *PyList_GetItem()*
- *PyList_GetItemRef()*
- *PyList_GetSlice()*
- *PyList_Insert()*
- *PyList_New()*
- *PyList_Reverse()*
- *PyList_SetItem()*
- *PyList_SetSlice()*
- *PyList_Size()*
- *PyList_Sort()*
- *PyList_Type*
- *PyLongObject*
- PyLongRangeIter_Type
- *PyLong_AsDouble()*
- *PyLong_AsInt()*
- *PyLong_AsLong()*
- *PyLong_AsLongAndOverflow()*
- *PyLong_AsLongLong()*
- *PyLong_AsLongLongAndOverflow()*
- *PyLong_AsSize_t()*
- *PyLong_AsSsize_t()*
- *PyLong_AsUnsignedLong()*
- *PyLong_AsUnsignedLongLong()*
- *PyLong_AsUnsignedLongLongMask()*
- *PyLong_AsUnsignedLongMask()*
- *PyLong_AsVoidPtr()*
- *PyLong_FromDouble()*
- *PyLong_FromLong()*
- *PyLong_FromLongLong()*
- *PyLong_FromSize_t()*
- *PyLong_FromSsize_t()*

- *PyLong_FromString()*
- *PyLong_FromUnsignedLong()*
- *PyLong_FromUnsignedLongLong()*
- *PyLong_FromVoidPtr()*
- *PyLong_GetInfo()*
- *PyLong_Type*
- PyMap_Type
- *PyMapping_Check()*
- *PyMapping_GetItemString()*
- *PyMapping_GetOptionalItem()*
- *PyMapping_GetOptionalItemString()*
- *PyMapping_HasKey()*
- *PyMapping_HasKeyString()*
- *PyMapping_HasKeyStringWithError()*
- *PyMapping_HasKeyWithError()*
- *PyMapping_Items()*
- *PyMapping_Keys()*
- *PyMapping_Length()*
- *PyMapping_SetItemString()*
- *PyMapping_Size()*
- *PyMapping_Values()*
- *PyMem_Calloc()*
- *PyMem_Free()*
- *PyMem_Malloc()*
- *PyMem_RawCalloc()*
- *PyMem_RawFree()*
- *PyMem_RawMalloc()*
- *PyMem_RawRealloc()*
- *PyMem_Realloc()*
- *PyMemberDef*
- PyMemberDescr_Type
- *PyMember_GetOne()*
- *PyMember_SetOne()*
- *PyMemoryView_FromBuffer()*
- *PyMemoryView_FromMemory()*
- *PyMemoryView_FromObject()*
- *PyMemoryView_GetContiguous()*
- PyMemoryView_Type
- *PyMethodDef*

- PyMethodDescr_Type
- *PyModuleDef*
- PyModuleDef_Base
- *PyModuleDef_Init()*
- PyModuleDef_Type
- *PyModule_Add()*
- *PyModule_AddFunctions()*
- *PyModule_AddIntConstant()*
- *PyModule_AddObject()*
- *PyModule_AddObjectRef()*
- *PyModule_AddStringConstant()*
- *PyModule_AddType()*
- *PyModule_Create2()*
- *PyModule_ExecDef()*
- *PyModule_FromDefAndSpec2()*
- *PyModule_GetDef()*
- *PyModule_GetDict()*
- *PyModule_GetFilename()*
- *PyModule_GetFilenameObject()*
- *PyModule_GetName()*
- *PyModule_GetNameObject()*
- *PyModule_GetState()*
- *PyModule_New()*
- *PyModule_NewObject()*
- *PyModule_SetDocString()*
- *PyModule_Type*
- *PyNumber_Absolute()*
- *PyNumber_Add()*
- *PyNumber_And()*
- *PyNumber_AsSsize_t()*
- *PyNumber_Check()*
- *PyNumber_Divmod()*
- *PyNumber_Float()*
- *PyNumber_FloorDivide()*
- *PyNumber_InPlaceAdd()*
- *PyNumber_InPlaceAnd()*
- *PyNumber_InPlaceFloorDivide()*
- *PyNumber_InPlaceLshift()*
- *PyNumber_InPlaceMatrixMultiply()*

- *PyNumber_InPlaceMultiply()*
- *PyNumber_InPlaceOr()*
- *PyNumber_InPlacePower()*
- *PyNumber_InPlaceRemainder()*
- *PyNumber_InPlaceRshift()*
- *PyNumber_InPlaceSubtract()*
- *PyNumber_InPlaceTrueDivide()*
- *PyNumber_InPlaceXor()*
- *PyNumber_Index()*
- *PyNumber_Invert()*
- *PyNumber_Long()*
- *PyNumber_Lshift()*
- *PyNumber_MatrixMultiply()*
- *PyNumber_Multiply()*
- *PyNumber_Negative()*
- *PyNumber_Or()*
- *PyNumber_Positive()*
- *PyNumber_Power()*
- *PyNumber_Remainder()*
- *PyNumber_Rshift()*
- *PyNumber_Subtract()*
- *PyNumber_ToBase()*
- *PyNumber_TrueDivide()*
- *PyNumber_Xor()*
- *PyOS_AfterFork()*
- *PyOS_AfterFork_Child()*
- *PyOS_AfterFork_Parent()*
- *PyOS_BeforeFork()*
- *PyOS_CheckStack()*
- *PyOS_FSPath()*
- *PyOS_InputHook*
- PyOS_InterruptOccurred()
- *PyOS_double_to_string()*
- *PyOS_getsig()*
- PyOS_mystricmp()
- PyOS_mystrnicmp()
- *PyOS_setsig()*
- *PyOS_sighandler_t*
- *PyOS_snprintf()*

- *PyOS_string_to_double()*
- *PyOS_strtol()*
- *PyOS_strtoul()*
- *PyOS_vsnprintf()*
- *PyObject*
- *PyObject.ob_refcnt*
- *PyObject.ob_type*
- *PyObject_ASCII()*
- *PyObject_AsFileDescriptor()*
- *PyObject_Bytes()*
- *PyObject_Call()*
- *PyObject_CallFunction()*
- *PyObject_CallFunctionObjArgs()*
- *PyObject_CallMethod()*
- *PyObject_CallMethodObjArgs()*
- *PyObject_CallNoArgs()*
- *PyObject_CallObject()*
- *PyObject_Calloc()*
- *PyObject_CheckBuffer()*
- *PyObject_ClearWeakRefs()*
- *PyObject_CopyData()*
- *PyObject_DelAttr()*
- *PyObject_DelAttrString()*
- *PyObject_DelItem()*
- *PyObject_DelItemString()*
- *PyObject_Dir()*
- *PyObject_Format()*
- *PyObject_Free()*
- *PyObject_GC_Del()*
- *PyObject_GC_IsFinalized()*
- *PyObject_GC_IsTracked()*
- *PyObject_GC_Track()*
- *PyObject_GC_UnTrack()*
- *PyObject_GenericGetAttr()*
- *PyObject_GenericGetDict()*
- *PyObject_GenericSetAttr()*
- *PyObject_GenericSetDict()*
- *PyObject_GetAIter()*
- *PyObject_GetAttr()*

- *PyObject_GetAttrString()*
- *PyObject_GetBuffer()*
- *PyObject_GetItem()*
- *PyObject_GetIter()*
- *PyObject_GetOptionalAttr()*
- *PyObject_GetOptionalAttrString()*
- *PyObject_GetTypeData()*
- *PyObject_HasAttr()*
- *PyObject_HasAttrString()*
- *PyObject_HasAttrStringWithError()*
- *PyObject_HasAttrWithError()*
- *PyObject_Hash()*
- *PyObject_HashNotImplemented()*
- *PyObject_Init()*
- *PyObject_InitVar()*
- *PyObject_IsInstance()*
- *PyObject_IsSubclass()*
- *PyObject_IsTrue()*
- *PyObject_Length()*
- *PyObject_Malloc()*
- *PyObject_Not()*
- *PyObject_Realloc()*
- *PyObject_Repr()*
- *PyObject_RichCompare()*
- *PyObject_RichCompareBool()*
- *PyObject_SelfIter()*
- *PyObject_SetAttr()*
- *PyObject_SetAttrString()*
- *PyObject_SetItem()*
- *PyObject_Size()*
- *PyObject_Str()*
- *PyObject_Type()*
- *PyObject_Vectorcall()*
- *PyObject_VectorcallMethod()*
- *PyProperty_Type*
- PyRangeIter_Type
- PyRange_Type
- PyReversed_Type
- *PySeqIter_New()*

- *PySeqIter_Type*
- *PySequence_Check()*
- *PySequence_Concat()*
- *PySequence_Contains()*
- *PySequence_Count()*
- *PySequence_DelItem()*
- *PySequence_DelSlice()*
- *PySequence_Fast()*
- *PySequence_GetItem()*
- *PySequence_GetSlice()*
- PySequence_In()
- *PySequence_InPlaceConcat()*
- *PySequence_InPlaceRepeat()*
- *PySequence_Index()*
- *PySequence_Length()*
- *PySequence_List()*
- *PySequence_Repeat()*
- *PySequence_SetItem()*
- *PySequence_SetSlice()*
- *PySequence_Size()*
- *PySequence_Tuple()*
- PySetIter_Type
- *PySet_Add()*
- *PySet_Clear()*
- *PySet_Contains()*
- *PySet_Discard()*
- *PySet_New()*
- *PySet_Pop()*
- *PySet_Size()*
- *PySet_Type*
- *PySlice_AdjustIndices()*
- *PySlice_GetIndices()*
- *PySlice_GetIndicesEx()*
- *PySlice_New()*
- *PySlice_Type*
- *PySlice_Unpack()*
- *PyState_AddModule()*
- *PyState_FindModule()*
- *PyState_RemoveModule()*

- *PyStructSequence_Desc*
- *PyStructSequence_Field*
- *PyStructSequence_GetItem()*
- *PyStructSequence_New()*
- *PyStructSequence_NewType()*
- *PyStructSequence_SetItem()*
- *PyStructSequence_UnnamedField*
- PySuper_Type
- *PySys_Audit()*
- *PySys_AuditTuple()*
- *PySys_FormatStderr()*
- *PySys_FormatStdout()*
- *PySys_GetObject()*
- *PySys_GetXOptions()*
- *PySys_ResetWarnOptions()*
- *PySys_SetArgv()*
- *PySys_SetArgvEx()*
- *PySys_SetObject()*
- *PySys_WriteStderr()*
- *PySys_WriteStdout()*
- *PyThreadState*
- *PyThreadState_Clear()*
- *PyThreadState_Delete()*
- *PyThreadState_Get()*
- *PyThreadState_GetDict()*
- *PyThreadState_GetFrame()*
- *PyThreadState_GetID()*
- *PyThreadState_GetInterpreter()*
- *PyThreadState_New()*
- *PyThreadState_SetAsyncExc()*
- *PyThreadState_Swap()*
- PyThread_GetInfo()
- *PyThread_ReInitTLS()*
- PyThread_acquire_lock()
- PyThread_acquire_lock_timed()
- PyThread_allocate_lock()
- *PyThread_create_key()*
- *PyThread_delete_key()*
- *PyThread_delete_key_value()*

- PyThread_exit_thread()
- PyThread_free_lock()
- *PyThread_get_key_value()*
- PyThread_get_stacksize()
- PyThread_get_thread_ident()
- PyThread_get_thread_native_id()
- PyThread_init_thread()
- PyThread_release_lock()
- *PyThread_set_key_value()*
- PyThread_set_stacksize()
- PyThread_start_new_thread()
- *PyThread_tss_alloc()*
- *PyThread_tss_create()*
- *PyThread_tss_delete()*
- *PyThread_tss_free()*
- *PyThread_tss_get()*
- *PyThread_tss_is_created()*
- *PyThread_tss_set()*
- PyTraceBack_Here()
- PyTraceBack_Print()
- PyTraceBack_Type
- PyTupleIter_Type
- *PyTuple_GetItem()*
- *PyTuple_GetSlice()*
- *PyTuple_New()*
- *PyTuple_Pack()*
- *PyTuple_SetItem()*
- *PyTuple_Size()*
- *PyTuple_Type*
- *PyTypeObject*
- *PyType_ClearCache()*
- *PyType_FromMetaclass()*
- *PyType_FromModuleAndSpec()*
- *PyType_FromSpec()*
- *PyType_FromSpecWithBases()*
- *PyType_GenericAlloc()*
- *PyType_GenericNew()*
- *PyType_GetFlags()*
- *PyType_GetFullyQualifiedName()*

- *PyType_GetModule()*
- *PyType_GetModuleByDef()*
- *PyType_GetModuleName()*
- *PyType_GetModuleState()*
- *PyType_GetName()*
- *PyType_GetQualName()*
- *PyType_GetSlot()*
- *PyType_GetTypeDataSize()*
- *PyType_IsSubtype()*
- *PyType_Modified()*
- *PyType_Ready()*
- *PyType_Slot*
- *PyType_Spec*
- *PyType_Type*
- *PyUnicodeDecodeError_Create()*
- *PyUnicodeDecodeError_GetEncoding()*
- *PyUnicodeDecodeError_GetEnd()*
- *PyUnicodeDecodeError_GetObject()*
- *PyUnicodeDecodeError_GetReason()*
- *PyUnicodeDecodeError_GetStart()*
- *PyUnicodeDecodeError_SetEnd()*
- *PyUnicodeDecodeError_SetReason()*
- *PyUnicodeDecodeError_SetStart()*
- *PyUnicodeEncodeError_GetEncoding()*
- *PyUnicodeEncodeError_GetEnd()*
- *PyUnicodeEncodeError_GetObject()*
- *PyUnicodeEncodeError_GetReason()*
- *PyUnicodeEncodeError_GetStart()*
- *PyUnicodeEncodeError_SetEnd()*
- *PyUnicodeEncodeError_SetReason()*
- *PyUnicodeEncodeError_SetStart()*
- *PyUnicodeIter_Type*
- *PyUnicodeTranslateError_GetEnd()*
- *PyUnicodeTranslateError_GetObject()*
- *PyUnicodeTranslateError_GetReason()*
- *PyUnicodeTranslateError_GetStart()*
- *PyUnicodeTranslateError_SetEnd()*
- *PyUnicodeTranslateError_SetReason()*
- *PyUnicodeTranslateError_SetStart()*

- PyUnicode_Append()
- PyUnicode_AppendAndDel()
- *PyUnicode_AsASCIIString()*
- *PyUnicode_AsCharmapString()*
- PyUnicode_AsDecodedObject()
- PyUnicode_AsDecodedUnicode()
- PyUnicode_AsEncodedObject()
- *PyUnicode_AsEncodedString()*
- PyUnicode_AsEncodedUnicode()
- *PyUnicode_AsLatin1String()*
- *PyUnicode_AsMBCSString()*
- *PyUnicode_AsRawUnicodeEscapeString()*
- *PyUnicode_AsUCS4()*
- *PyUnicode_AsUCS4Copy()*
- *PyUnicode_AsUTF16String()*
- *PyUnicode_AsUTF32String()*
- *PyUnicode_AsUTF8AndSize()*
- *PyUnicode_AsUTF8String()*
- *PyUnicode_AsUnicodeEscapeString()*
- *PyUnicode_AsWideChar()*
- *PyUnicode_AsWideCharString()*
- *PyUnicode_BuildEncodingMap()*
- *PyUnicode_Compare()*
- *PyUnicode_CompareWithASCIIString()*
- *PyUnicode_Concat()*
- *PyUnicode_Contains()*
- *PyUnicode_Count()*
- *PyUnicode_Decode()*
- *PyUnicode_DecodeASCII()*
- *PyUnicode_DecodeCharmap()*
- *PyUnicode_DecodeCodePageStateful()*
- *PyUnicode_DecodeFSDefault()*
- *PyUnicode_DecodeFSDefaultAndSize()*
- *PyUnicode_DecodeLatin1()*
- *PyUnicode_DecodeLocale()*
- *PyUnicode_DecodeLocaleAndSize()*
- *PyUnicode_DecodeMBCS()*
- *PyUnicode_DecodeMBCSStateful()*
- *PyUnicode_DecodeRawUnicodeEscape()*

- *PyUnicode_DecodeUTF16()*
- *PyUnicode_DecodeUTF16Stateful()*
- *PyUnicode_DecodeUTF32()*
- *PyUnicode_DecodeUTF32Stateful()*
- *PyUnicode_DecodeUTF7()*
- *PyUnicode_DecodeUTF7Stateful()*
- *PyUnicode_DecodeUTF8()*
- *PyUnicode_DecodeUTF8Stateful()*
- *PyUnicode_DecodeUnicodeEscape()*
- *PyUnicode_EncodeCodePage()*
- *PyUnicode_EncodeFSDefault()*
- *PyUnicode_EncodeLocale()*
- *PyUnicode_EqualToUTF8()*
- *PyUnicode_EqualToUTF8AndSize()*
- *PyUnicode_FSConverter()*
- *PyUnicode_FSDecoder()*
- *PyUnicode_Find()*
- *PyUnicode_FindChar()*
- *PyUnicode_Format()*
- *PyUnicode_FromEncodedObject()*
- *PyUnicode_FromFormat()*
- *PyUnicode_FromFormatV()*
- *PyUnicode_FromObject()*
- *PyUnicode_FromOrdinal()*
- *PyUnicode_FromString()*
- *PyUnicode_FromStringAndSize()*
- *PyUnicode_FromWideChar()*
- *PyUnicode_GetDefaultEncoding()*
- *PyUnicode_GetLength()*
- *PyUnicode_InternFromString()*
- *PyUnicode_InternInPlace()*
- *PyUnicode_IsIdentifier()*
- *PyUnicode_Join()*
- *PyUnicode_Partition()*
- *PyUnicode_RPartition()*
- *PyUnicode_RSplit()*
- *PyUnicode_ReadChar()*
- *PyUnicode_Replace()*
- PyUnicode_Resize()

- *PyUnicode_RichCompare()*
- *PyUnicode_Split()*
- *PyUnicode_Splitlines()*
- *PyUnicode_Substring()*
- *PyUnicode_Tailmatch()*
- *PyUnicode_Translate()*
- *PyUnicode_Type*
- *PyUnicode_WriteChar()*
- *PyVarObject*
- PyVarObject.ob_base
- *PyVarObject.ob_size*
- *PyVectorcall_Call()*
- *PyVectorcall_NARGS()*
- PyWeakReference
- *PyWeakref_GetObject()*
- *PyWeakref_GetRef()*
- *PyWeakref_NewProxy()*
- *PyWeakref_NewRef()*
- PyWrapperDescr_Type
- *PyWrapper_New()*
- PyZip_Type
- *Py_AddPendingCall()*
- *Py_AtExit()*
- *Py_BEGIN_ALLOW_THREADS*
- *Py_BLOCK_THREADS*
- *Py_BuildValue()*
- *Py_BytesMain()*
- *Py_CompileString()*
- *Py_DecRef()*
- *Py_DecodeLocale()*
- *Py_END_ALLOW_THREADS*
- *Py_EncodeLocale()*
- *Py_EndInterpreter()*
- *Py_EnterRecursiveCall()*
- *Py_Exit()*
- *Py_FatalError()*
- Py_FileSystemDefaultEncodeErrors
- Py_FileSystemDefaultEncoding
- *Py_Finalize()*

- *Py_FinalizeEx()*
- *Py_GenericAlias()*
- *Py_GenericAliasType*
- *Py_GetBuildInfo()*
- *Py_GetCompiler()*
- *Py_GetConstant()*
- *Py_GetConstantBorrowed()*
- *Py_GetCopyright()*
- *Py_GetExecPrefix()*
- *Py_GetPath()*
- *Py_GetPlatform()*
- *Py_GetPrefix()*
- *Py_GetProgramFullPath()*
- *Py_GetProgramName()*
- *Py_GetPythonHome()*
- Py_GetRecursionLimit()
- *Py_GetVersion()*
- Py_HasFileSystemDefaultEncoding
- *Py_IncRef()*
- *Py_Initialize()*
- *Py_InitializeEx()*
- *Py_Is()*
- *Py_IsFalse()*
- *Py_IsFinalizing()*
- *Py_IsInitialized()*
- *Py_IsNone()*
- *Py_IsTrue()*
- *Py_LeaveRecursiveCall()*
- *Py_Main()*
- Py_MakePendingCalls()
- *Py_NewInterpreter()*
- *Py_NewRef()*
- *Py_ReprEnter()*
- *Py_ReprLeave()*
- *Py_SetProgramName()*
- *Py_SetPythonHome()*
- Py_SetRecursionLimit()
- *Py_UCS4*
- *Py_UNBLOCK_THREADS*

- Py_UTF8Mode
- *Py_VaBuildValue()*
- *Py_Version*
- *Py_XNewRef()*
- *Py_buffer*
- Py_intptr_t
- *Py_ssize_t*
- Py_uintptr_t
- *allocfunc*
- *binaryfunc*
- *descrgetfunc*
- *descrsetfunc*
- *destructor*
- *getattrfunc*
- *getattrofunc*
- *getbufferproc*
- *getiterfunc*
- *getter*
- *hashfunc*
- *initproc*
- *inquiry*
- *iternextfunc*
- *lenfunc*
- *newfunc*
- *objobjargproc*
- *objobjproc*
- *releasebufferproc*
- *reprfunc*
- *richcmpfunc*
- *setattrfunc*
- *setattrofunc*
- *setter*
- *ssizeargfunc*
- *ssizeobjargproc*
- ssizessizeargfunc
- ssizessizeobjargproc
- symtable
- *ternaryfunc*
- *traverseproc*

- *unaryfunc*
- *vectorcallfunc*
- *visitproc*

CHAPTER 3

极高层级 API

本章节的函数将允许你执行在文件或缓冲区中提供的 Python 源代码，但它们将不允许你在更细节化的方式与解释器进行交互。

这些函数中有几个可以接受特定的语法前缀符号作为形参。可用的前缀符号有*Py_eval_input*，*Py_file_input* 和*Py_single_input*。这些符号会在接受它们作为形参的函数文档中加以说明。

还要注意这些函数中有几个可以接受 `FILE*` 形参。有一个需要小心处理的特别问题是针对不同 C 库的 `FILE` 结构体可能是不相同且不兼容的。（至少是）在 Windows 中，动态链接的扩展实际上有可能会使用不同的库，所以应当特别注意只有在确定这些函数是由 Python 运行时所使用的相同的库创建的情况下才将 `FILE*` 形参传给它们。

int **PyRun_AnyFile**(FILE *fp, const char *filename)

这是针对下面*PyRun_AnyFileExFlags()* 的简化版接口，将 *closeit* 设为 `0` 而将 *flags* 设为 `NULL`。

int **PyRun_AnyFileFlags**(FILE *fp, const char *filename, *PyCompilerFlags* *flags)

这是针对下面*PyRun_AnyFileExFlags()* 的简化版接口，将 *closeit* 参数设为 `0`。

int **PyRun_AnyFileEx**(FILE *fp, const char *filename, int closeit)

这是针对下面*PyRun_AnyFileExFlags()* 的简化版接口，将 *flags* 参数设为 `NULL`。

int **PyRun_AnyFileExFlags**(FILE *fp, const char *filename, int closeit, *PyCompilerFlags* *flags)

如果 *fp* 指向一个关联到交互设备（控制台或终端输入或 Unix 伪终端）的文件，则返回*PyRun_InteractiveLoop()* 的值，否则返回*PyRun_SimpleFile()* 的结果。*filename* 会使用文件系统的编码格式（`sys.getfilesystemencoding()`）来解码。如果 *filename* 为 `NULL`，此函数会使用 `"???"` 作为文件名。如果 *closeit* 为真值，文件会在 `PyRun_SimpleFileExFlags()` 返回之前被关闭。

int **PyRun_SimpleString**(const char *command)

这是针对下面*PyRun_SimpleStringFlags()* 的简化版接口，将*PyCompilerFlags** 参数设为 `NULL`。

int **PyRun_SimpleStringFlags**(const char *command, *PyCompilerFlags* *flags)

根据 *flags* 参数，在 `__main__` 模块中执行 Python 源代码。如果 `__main__` 尚不存在，它将被创建。成功时返回 `0`，如果引发异常则返回 `-1`。如果发生错误，则将无法获得异常信息。对于 *flags* 的含义，请参阅下文。

请注意如果引发了一个在其他场合下未处理的 `SystemExit`，此函数将不会返回 `-1`，而是退出进程，只要*PyConfig.inspect* 为零就会这样。

int **PyRun_SimpleFile**(FILE *fp, const char *filename)

这是针对下面*PyRun_SimpleFileExFlags()* 的简化版接口，将 *closeit* 设为 `0` 而将 *flags* 设为 `NULL`。

int **PyRun_SimpleFileEx**(FILE *fp, const char *filename, int closeit)

这是针对下面*PyRun_SimpleFileExFlags()* 的简化版接口，将 *flags* 设为 NULL。

int **PyRun_SimpleFileExFlags**(FILE *fp, const char *filename, int closeit, *PyCompilerFlags* *flags)

类似于*PyRun_SimpleStringFlags()*，但 Python 源代码是从 *fp* 读取而不是一个内存中的字符串。*filename* 应为文件名，它将使用*filesystem encoding and error handler* 来解码。如果 *closeit* 为真值，则文件将在 PyRun_SimpleFileExFlags() 返回之前被关闭。

> **备注**
>
> 在 Windows 上，*fp* 应当以二进制模式打开 (即 fopen(filename, "rb"))。否则，Python 可能无法正确地处理使用 LF 行结束符的脚本文件。

int **PyRun_InteractiveOne**(FILE *fp, const char *filename)

这是针对下面*PyRun_InteractiveOneFlags()* 的简化版接口，将 *flags* 设为 NULL。

int **PyRun_InteractiveOneFlags**(FILE *fp, const char *filename, *PyCompilerFlags* *flags)

根据 *flags* 参数读取并执行来自与交互设备相关联的文件的一条语句。用户将得到使用 sys.ps1 和 sys.ps2 的提示。*filename* 将使用*filesystem encoding and error handler* 来解码。

当输入被成功执行时返回 0，如果引发异常则返回 -1，或者如果存在解析错误则返回来自作为 Python 的组成部分发布的 errcode.h 包括文件的错误代码。(请注意 errcode.h 并未被 Python.h 所包括，因此如果需要则必须专门地包括。)

int **PyRun_InteractiveLoop**(FILE *fp, const char *filename)

这是针对下面*PyRun_InteractiveLoopFlags()* 的简化版接口，将 *flags* 设为 NULL。

int **PyRun_InteractiveLoopFlags**(FILE *fp, const char *filename, *PyCompilerFlags* *flags)

读取并执行来自与交互设备相关联的语句直至到达 EOF。用户将得到使用 sys.ps1 和 sys.ps2 的提示。*filename* 将使用*filesystem encoding and error handler* 来解码。当位于 EOF 时将返回 0，或者当失败时将返回一个负数。

int (***PyOS_InputHook**)(void)

属于稳定 ABI. 可以被设为指向一个原型为 int func(void) 的函数。该函数将在 Python 的解释器提示符即将空闲并等待用户从终端输入时被调用。返回值会被忽略。重写这个钩子可被用来将解释器的提示符集成到其他事件循环中，就像 Python 码中 Modules/_tkinter.c 所做的那样。

在 3.12 版本发生变更: 此函数只能被主解释器 调用。

char *(***PyOS_ReadlineFunctionPointer**)(FILE*, FILE*, const char*)

可以被设为指向一个原型为 char *func(FILE *stdin, FILE *stdout, char *prompt) 的函数，重写被用来读取解释器提示符的一行输入的默认函数。该函数被预期为如果字符串 *prompt* 不为 NULL 就输出它，然后从所提供的标准输入文件读取一行输入，并返回结果字符串。例如，readline 模块将这个钩子设置为提供行编辑和 tab 键补全等功能。

结果必须是一个由*PyMem_RawMalloc()* 或*PyMem_RawRealloc()* 分配的字符串，或者如果发生错误则为 NULL。

在 3.4 版本发生变更: 结果必须由*PyMem_RawMalloc()* 或*PyMem_RawRealloc()* 分配，而不是由*PyMem_Malloc()* 或*PyMem_Realloc()* 分配。

在 3.12 版本发生变更: 此函数只能被主解释器 调用。

PyObject ***PyRun_String**(const char *str, int start, *PyObject* *globals, *PyObject* *locals)

返回值：新的引用。 这是针对下面*PyRun_StringFlags()* 的简化版接口，将 *flags* 设为 NULL。

PyObject ***PyRun_StringFlags**(const char *str, int start, *PyObject* *globals, *PyObject* *locals, *PyCompilerFlags* *flags)

返回值：新的引用。 在由对象 *globals* 和 *locals* 指定的上下文中执行来自 *str* 的 Python 源代码，并使用以 *flags* 指定的编译器旗标。*globals* 必须是一个字典；*locals* 可以是任何实现了映射协议的对象。形参 *start* 指定了应当被用来解析源代码的起始形符。

返回将代码作为 Python 对象执行的结果，或者如果引发了异常则返回 NULL。

PyObject ***PyRun_File**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals)

返回值：新的引用。 这是针对下面*PyRun_FileExFlags()* 的简化版接口，将 *closeit* 设为 0 并将 *flags* 设为 NULL。

PyObject ***PyRun_FileEx**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, int closeit)

返回值：新的引用。 这是针对下面*PyRun_FileExFlags()* 的简化版接口，将 *flags* 设为 NULL。

PyObject ***PyRun_FileFlags**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, *PyCompilerFlags* *flags)

返回值：新的引用。 这是针对下面*PyRun_FileExFlags()* 的简化版接口，将 *closeit* 设为 0。

PyObject ***PyRun_FileExFlags**(FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, int closeit, *PyCompilerFlags* *flags)

返回值：新的引用。 类似于*PyRun_StringFlags()*，但 Python 源代码是从 *fp* 读取而不是一个内存中的字符串。*filename* 应为文件名，它将使用*filesystem encoding and error handler* 来解码。如果 *closeit* 为真值，则文件将在*PyRun_FileExFlags()* 返回之前被关闭。

PyObject ***Py_CompileString**(const char *str, const char *filename, int start)

返回值：新的引用。属于稳定 ABI. 这是针对下面*Py_CompileStringFlags()* 的简化版接口，将 *flags* 设为 NULL。

PyObject ***Py_CompileStringFlags**(const char *str, const char *filename, int start, *PyCompilerFlags* *flags)

返回值：新的引用。 这是针对下面*Py_CompileStringExFlags()* 的简化版接口，将 *optimize* 设为 -1。

PyObject ***Py_CompileStringObject**(const char *str, *PyObject* *filename, int start, *PyCompilerFlags* *flags, int optimize)

返回值：新的引用。解析并编译 *str* 中的 Python 源代码，返回结果代码对象。开始形符由 *start* 给出；这可被用来约束可被编译的代码并且应当为*Py_eval_input*, *Py_file_input* 或*Py_single_input*。由 *filename* 指定的文件名会被用来构造代码对象并可能出现在回溯信息或 SyntaxError 异常消息中。如果代码无法被解析或编译则此函数将返回 NULL。

整数 *optimize* 指定编译器的优化级别；值 -1 将选择与 -O 选项相同的解释器优化级别。显式级别为 0 (无优化；__debug__ 为真值)、1 (断言被移除，__debug__ 为假值) 或 2 (文档字符串也被移除)。

Added in version 3.4.

PyObject ***Py_CompileStringExFlags**(const char *str, const char *filename, int start, *PyCompilerFlags* *flags, int optimize)

返回值：新的引用。 与*Py_CompileStringObject()* 类似，但 *filename* 是以*filesystem encoding and error handler* 解码出的字节串。

Added in version 3.2.

PyObject ***PyEval_EvalCode**(*PyObject* *co, *PyObject* *globals, *PyObject* *locals)

返回值：新的引用。属于稳定 ABI. 这是针对*PyEval_EvalCodeEx()* 的简化版接口，只附带代码对象，以及全局和局部变量。其他参数均设为 NULL。

PyObject ***PyEval_EvalCodeEx**(*PyObject* *co, *PyObject* *globals, *PyObject* *locals, *PyObject* *const *args, int argcount, *PyObject* *const *kws, int kwcount, *PyObject* *const *defs, int defcount, *PyObject* *kwdefs, *PyObject* *closure)

返回值：新的引用。属于稳定 ABI. 对一个预编译的代码对象求值，为其求值给出特定的环境。此环境由全局变量的字典，局部变量映射对象，参数、关键字和默认值的数组，仅限关键字 参数的默认值的字典和单元的封闭元组构成。

PyObject ***PyEval_EvalFrame**(*PyFrameObject* *f)

返回值：新的引用。属于稳定 ABI. 对一个执行帧求值。这是针对*PyEval_EvalFrameEx()* 的简化版接口，用于保持向下兼容性。

PyObject ***PyEval_EvalFrameEx**(*PyFrameObject* *f, int throwflag)

返回值：新的引用。属于稳定 ABI. 这是 Python 解释运行不带修饰的主函数。与执行帧 *f* 相关联的代码对象将被执行，解释字节码并根据需要执行调用。额外的 *throwflag* 形参基本可以被忽略——如果为真值，则会导致立即抛出一个异常；这会被用于生成器对象的 `throw()` 方法。

在 3.4 版本发生变更: 该函数现在包含一个调试断言，用以确保不会静默地丢弃活动的异常。

int **PyEval_MergeCompilerFlags**(*PyCompilerFlags* *cf)

此函数会修改当前求值帧的旗标，并在成功时返回真值，失败时返回假值。

int **Py_eval_input**

Python 语法中用于孤立表达式的起始符号；配合*Py_CompileString()* 使用。

int **Py_file_input**

Python 语法中用于从文件或其他源读取语句序列的起始符号；配合*Py_CompileString()* 使用。这是在编译任意长的 Python 源代码时要使用的符号。

int **Py_single_input**

Python 语法中用于单独语句的起始符号；配合*Py_CompileString()* 使用。这是用于交互式解释器循环的符号。

struct **PyCompilerFlags**

这是用来存放编译器旗标的结构体。对于代码仅被编译的情况，它将作为 `int flags` 传入，而对于代码要被执行的情况，它将作为 `PyCompilerFlags *flags` 传入。在这种情况下，`from __future__ import` 可以修改 *flags*。

只要 `PyCompilerFlags *flags` 是 `NULL`，*cf_flags* 就会被视为等同于 `0`，而由于 `from __future__ import` 而产生的任何修改都会被丢弃。

int **cf_flags**

编译器旗标。

int **cf_feature_version**

cf_feature_version 是 Python 的小版本号。它应当被初始化为 `PY_MINOR_VERSION`。

该字段默认会被忽略，当且仅当在*cf_flags* 中设置了 `PyCF_ONLY_AST` 旗标时它才会被使用。

在 3.8 版本发生变更: 增加了 *cf_feature_version* 字段。

现有的编译器旗标可作为宏来使用：

PyCF_ALLOW_TOP_LEVEL_AWAIT

PyCF_ONLY_AST

PyCF_OPTIMIZED_AST

PyCF_TYPE_COMMENTS

请参阅 `ast` Python 模块文档中的 编译器旗标，它们会将这些常量以相同的名称导出。

The "`PyCF`" flags above can be combined with "`CO_FUTURE`" flags such as *CO_FUTURE_ANNOTATIONS* to enable features normally selectable using future statements. See *Code Object Flags* for a complete list.

CHAPTER 4

引用计数

本节介绍的函数和宏被用于管理 Python 对象的引用计数。

Py_ssize_t **Py_REFCNT**(*PyObject* *o)

获取 Python 对象 *o* 的引用计数。

请注意返回的值可能并不真正反映实际持有的对象引用数。例如，有些对象属于*immortal* 对象并具有并不反映实际引用数的非常高的 refcount 值。因此，除了 0 或 1 这两个值，不要依赖返回值的准确性。

使用`Py_SET_REFCNT()` 函数来设置一个对象引用计数。

在 3.10 版本发生变更: `Py_REFCNT()` 被改为内联的静态函数。

在 3.11 版本发生变更: 形参类型不再是 `const PyObject*`。

void **Py_SET_REFCNT**(*PyObject* *o, *Py_ssize_t* refcnt)

将对象 *o* 的引用计数器设为 *refcnt*。

在 启用自由线程的 Python 编译版中，如果 *refcnt* 大于 `UINT32_MAX`，该对象将被设为*immortal* 对象。

此函数对*immortal* 对象没有效果。

Added in version 3.9.

在 3.12 版本发生变更: 永生对象不会被修改。

void **Py_INCREF**(*PyObject* *o)

表示为对象 *o* 获取一个新的*strong reference*，指明该对象正在被使用且不应被销毁。

此函数对*immortal* 对象没有效果。

此函数通常被用来将*borrowed reference* 原地转换为*strong reference*。`Py_NewRef()` 函数可被用来创建新的*strong reference*。

当对象使用完毕后，可调用`Py_DECREF()` 来释放它。

此对象必须不为 `NULL`；如果你不能确定它不为 `NULL`，请使用`Py_XINCREF()`。

不要预期此函数会以任何方式实际地改变 *o*。至少对 **某些对象** 来说，此函数将没有任何效果。

在 3.12 版本发生变更: 永生对象不会被修改。

void **Py_XINCREF**(*PyObject* *o)

与*Py_INCREF()* 类似，但对象 *o* 可以为 NULL，在这种情况下此函数将没有任何效果。

另请参阅*Py_XNewRef()*。

PyObject ***Py_NewRef**(*PyObject* *o)

属于稳定 ABI 自 *3.10* 版起. 为对象创建一个新的*strong reference*: 在 *o* 上调用*Py_INCREF()* 并返回对象 *o*。

当不再需要这个*strong reference* 时，应当在其上调用*Py_DECREF()* 来释放引用。

对象 *o* 必须不为 NULL；如果 *o* 可以为 NULL 则应改用*Py_XNewRef()*。

例如：

```
Py_INCREF(obj);
self->attr = obj;
```

可以写成:

```
self->attr = Py_NewRef(obj);
```

另请参阅*Py_INCREF()*。

Added in version 3.10.

PyObject ***Py_XNewRef**(*PyObject* *o)

属于稳定 ABI 自 *3.10* 版起. 类似于*Py_NewRef()*，但对象 *o* 可以为 NULL。

如果对象 *o* 为 NULL，该函数也 · 将返回 NULL。

Added in version 3.10.

void **Py_DECREF**(*PyObject* *o)

释放一个指向对象 *o* 的*strong reference*，表明该引用不再被使用。

此函数对*immortal* 对象没有效果。

当最后一个*strong reference* 被释放时 (即对象的引用计数变为 0)，将会唤起该对象所属类型的 deallocation 函数 (它必须不为 NULL)。

此函数通常被用于在退出作用域之前删除一个*strong reference*。

此对象必须不为 NULL；如果你不能确定它不为 NULL，请使用*Py_XDECREF()*。

不要预期此函数会以任何方式实际地改变 *o*。至少对 **某些对象** 来说，此函数将没有任何效果。

警告

释放函数会导致任意 Python 代码被唤起（例如当一个带有 `__del__()` 方法的类实例被释放时就是如此）。虽然这些代码中的异常不会被传播，但被执行的代码能够自由访问所有 Python 全局变量。这意味着在调用*Py_DECREF()* 之前任何可通过全局变量获取的对象都应该处于完好的状态。例如，从一个列表中删除对象的代码应该将被删除对象的引用拷贝到一个临时变量中，更新列表数据结构，然后再为临时变量调用*Py_DECREF()*。

在 3.12 版本发生变更: 永生对象不会被修改。

void **Py_XDECREF**(*PyObject* *o)

与*Py_DECREF()* 类似，但对象 *o* 可以为 NULL，在这种情况下此函数将没有任何效果。来自*Py_DECREF()* 的警告同样适用于此处。

void **Py_CLEAR**(*PyObject* *o)

释放一个指向对象 *o* 的*strong reference*。对象可以为 NULL，在此情况下该宏将没有任何效果；在其他情况下其效果与*Py_DECREF()* 相同，区别在于其参数也会被设为 NULL。针对*Py_DECREF()* 的警告不适用于所传递的对象，因为该宏会细心地使用一个临时变量并在释放引用之前将参数设为 NULL。

当需要释放指向一个在垃圾回收期间可能被会遍历的对象的引用时使用该宏是一个好主意。

在 3.12 版本发生变更: 该宏参数现在只会被求值一次。如果该参数具有附带影响，它们将不会再被复制。

void **Py_IncRef**(*PyObject* *o)

属于稳定 ABI. 表示获取一个指向对象 *o* 的新*strong reference*。*Py_XINCREF()* 的函数版本。它可被用于 Python 的运行时动态嵌入。

void **Py_DecRef**(*PyObject* *o)

属于稳定 ABI. 释放一个指向对象 *o* 的*strong reference*。*Py_XDECREF()* 的函数版本。它可被用于 Python 的运行时动态嵌入。

Py_SETREF(dst, src)

该宏可安全地释放一个指向对象 *dst* 的*strong reference*，并将 *dst* 设为 *src*。

在*Py_CLEAR()* 的情况中，这样“直观”的代码可能会是致命的:

```
Py_DECREF(dst);
dst = src;
```

安全的方式是这样:

```
Py_SETREF(dst, src);
```

这样使得在释放对 *dst* 的旧值的引用之前将 *dst* 设为 *src*，从而让任何作为 *dst* 被去除的附带影响被触发的代码不再相信 *dst* 指向一个有效的对象。

Added in version 3.6.

在 3.12 版本发生变更: 该宏参数现在只会被求值一次。如果某个参数具有附带影响，它们将不会再被复制。

Py_XSETREF(dst, src)

使用*Py_XDECREF()* 代替*Py_DECREF()* 的*Py_SETREF* 宏的变种。

Added in version 3.6.

在 3.12 版本发生变更: 该宏参数现在只会被求值一次。如果某个参数具有附带影响，它们将不会再被复制。

CHAPTER 5

异常处理

本章描述的函数将让你处理和触发 Python 异常。了解一些 Python 异常处理的基础知识是很重要的。它的工作原理有点像 POSIX `errno` 变量: (每个线程) 有一个最近发生的错误的全局指示器。大多数 C API 函数在成功执行时将不理会它。大多数 C API 函数也会返回一个错误指示器，如果它们应当返回一个指针则会返回 `NULL`，或者如果它们应当返回一个整数则会返回 `-1` (例外情况: `PyArg_*` 函数返回 `1` 表示成功而 `0` 表示失败)。

具体地说，错误指示器由三个对象指针组成：异常的类型，异常的值，和回溯对象。如果没有错误被设置，这些指针都可以是 `NULL` （尽管一些组合使禁止的，例如，如果异常类型是 `NULL`，你不能有一个非 `NULL` 的回溯）。

当一个函数由于它调用的某个函数失败而必须失败时，通常不会设置错误指示器；它调用的那个函数已经设置了它。而它负责处理错误和清理异常，或在清除其拥有的所有资源后返回（如对象应用或内存分配）。如果不准备处理异常，则 不应该正常地继续。如果是由于一个错误返回，那么一定要向调用者表明已经设置了错误。如果错误没有得到处理或小心传播，对 Python/C API 的其它调用可能不会有预期的行为，并且可能会以某种神秘的方式失败。

备注

错误指示器 **不是** `sys.exc_info()` 的执行结果。前者对应于尚未捕获（因而仍在传播）的异常，而后者会在异常被捕获之后（因而已停止传播）返回它。

5.1 打印和清理

void **PyErr_Clear**()

属于稳定 ABI. 清除错误指示器。如果没有设置错误指示器，则不会有作用。

void **PyErr_PrintEx**(int set_sys_last_vars)

属于稳定 ABI. 将标准回溯打印到 `sys.stderr` 并清除错误指示器。**除非**错误是 `SystemExit`，这种情况下不会打印回溯进程，且会退出 Python 进程，并显示 `SystemExit` 实例指定的错误代码。

只有在错误指示器被设置时才需要调用这个函数，否则这会导致错误!

如果 *set_sys_last_vars* 为非零值，则变量 `sys.last_exc` 将被设为要打印的异常。出于向下兼容性考虑，已弃用的变量 `sys.last_type`, `sys.last_value` 和 `sys.last_traceback` 也会被分别设为该异常的类型, 值和回溯。

在 3.12 版本发生变更: 增加了对 `sys.last_exc` 的设置。

void **PyErr_Print**()

属于稳定 ABI. `PyErr_PrintEx(1)` 的别名。

void **PyErr_WriteUnraisable**(*PyObject* *obj)

属于稳定 ABI. 使用当前异常和 *obj* 参数调用 `sys.unraisablehook()`。

当异常已被设置但解释器不可能实际引发该异常时，这个工具函数会向 `sys.stderr` 打印一条警告消息。例如，当异常发生在 `__del__()` 方法中时就会使用该函数。

该函数调用时将传入单个参数 *obj*，它标识发生不可引发的异常所在的上下文。如果可能，*obj* 的表示形式将打印在警告消息中。如果 *obj* 为 `NULL`，将只打印回溯。

调用此函数时必须设置一个异常。

在 3.4 版本发生变更: 打印回溯信息。如果 *obj* 为 `NULL` 将只打印回溯。

在 3.8 版本发生变更: 使用 `sys.unraisablehook()`。

void **PyErr_FormatUnraisable**(const char *format, ...)

与*PyErr_WriteUnraisable()* 类似，但 *format* 和后续的形参有助于格式化警告消息；它们的含义和值与*PyUnicode_FromFormat()* 中的相同。`PyErr_WriteUnraisable(obj)` 大致等价于 `PyErr_FormatUnraisable("Exception ignored in: %R", obj)`。如果 *format* 为 `NULL`，则只打印回溯信息。

Added in version 3.13.

void **PyErr_DisplayException**(*PyObject* *exc)

属于稳定 ABI 自 *3.12* 版起. 将 `exc` 的标准回溯显示打印到 `sys.stderr`，包括链式异常和注释。

Added in version 3.12.

5.2 抛出异常

这些函数可帮助你设置当前线程的错误指示器。为了方便起见，一些函数将始终返回 `NULL` 指针，以便用于 `return` 语句。

void **PyErr_SetString**(*PyObject* *type, const char *message)

属于稳定 ABI. 这是设置错误指示器最常用的方式。第一个参数指定异常类型；它通常为某个标准异常，例如*PyExc_RuntimeError*。你无需为其创建新的*strong reference* (例如使用*Py_INCREF()*)。第二个参数是一条错误消息；它是用 `'utf-8'` 解码的。

void **PyErr_SetObject**(*PyObject* *type, *PyObject* *value)

属于稳定 ABI. 此函数类似于*PyErr_SetString()*，但是允许你为异常的 “值” 指定任意一个 Python 对象。

PyObject ***PyErr_Format**(*PyObject* *exception, const char *format, ...)

返回值：恒为 *NULL*。属于稳定 ABI. 这个函数设置了一个错误指示器并且返回了 `NULL`，*exception* 应当是一个 Python 中的异常类。*format* 和随后的形参会帮助格式化这个错误的信息；它们与*PyUnicode_FromFormat()* 有着相同的含义和值。*format* 是一个 ASCII 编码的字符串。

PyObject ***PyErr_FormatV**(*PyObject* *exception, const char *format, va_list vargs)

返回值：恒为 *NULL*。属于稳定 ABI 自 *3.5* 版起. 和*PyErr_Format()* 相同，但它接受一个 `va_list` 类型的参数而不是可变数量的参数集。

Added in version 3.5.

void **PyErr_SetNone**(*PyObject* *type)

属于稳定 ABI. 这是 `PyErr_SetObject(type, Py_None)` 的简写。

int **PyErr_BadArgument**()

属于稳定 ABI. 这是 `PyErr_SetString(PyExc_TypeError, message)` 的简写，其中 *message* 指出使用了非法参数调用内置操作。它主要用于内部使用。

PyObject ***PyErr_NoMemory**()

返回值：恒为 *NULL*。属于稳定 ABI. 这是 `PyErr_SetNone(PyExc_MemoryError)` 的简写；它返回 `NULL`，以便当内存耗尽时，对象分配函数可以写 `return PyErr_NoMemory();` 。

PyObject ***PyErr_SetFromErrno**(*PyObject* *type)

返回值：恒为 *NULL*。属于稳定 ABI. 这是一个便捷函数，当在 C 库函数返回错误并设置 C 变量 `errno` 时它会引发一个异常。它构造了一个元组对象，其第一项是整数值 `errno` 而第二项是对应的错误信息（从 `strerror()` 获取），然后调用 `PyErr_SetObject(type, object)`。在 Unix 上，当 `errno` 的值为 `EINTR` 时，表示有一个中断的系统调用，这将会调用*PyErr_CheckSignals()*，如果它设置了错误指示符，则让其保持该设置。该函数总是返回 `NULL`，因此当系统调用返回错误时该系统调用的包装函数可以写入 `return PyErr_SetFromErrno(type);`。

PyObject ***PyErr_SetFromErrnoWithFilenameObject**(*PyObject* *type, *PyObject* *filenameObject)

返回值：恒为 *NULL*。属于稳定 ABI. 与*PyErr_SetFromErrno()* 类似，但如果 *filenameObject* 不为 `NULL`，它将作为第三个参数传递给 *type* 的构造函数。在 `OSError` 异常的情况下，它将被用于定义异常实例的 `filename` 属性。

PyObject ***PyErr_SetFromErrnoWithFilenameObjects**(*PyObject* *type, *PyObject* *filenameObject, *PyObject* *filenameObject2)

返回值：恒为 *NULL*。属于稳定 ABI 自 *3.7* 版起. 类似于*PyErr_SetFromErrnoWithFilenameObject()*，但接受第二个 filename 对象，用于当一个接受两个 filename 的函数失败时触发错误。

Added in version 3.4.

PyObject ***PyErr_SetFromErrnoWithFilename**(*PyObject* *type, const char *filename)

返回值：恒为 *NULL*。属于稳定 ABI. 类似于*PyErr_SetFromErrnoWithFilenameObject()*，但文件名以 C 字符串形式给出。*filename* 是用*filesystem encoding and error handler* 解码的。

PyObject ***PyErr_SetFromWindowsErr**(int ierr)

返回值：恒为 *NULL*。属于稳定 ABI *on Windows* 自 *3.7* 版起. 这是一个用于引发 `OSError` 的便捷函数。如果调用时传入的 *ierr* 值为 `0`，则会改用对 `GetLastError()` 的调用所返回的错误代码。它将调用 Win32 函数 `FormatMessage()` 来获取 *ierr* 或 `GetLastError()` 所给出的错误代码的 Windows 描述，然后构造一个 `OSError` 对象，其中 `winerror` 属性将设为该错误代码，`strerror` 属性将设为相应的错误消息（从 `FormatMessage()` 获得），然后再调用 `PyErr_SetObject(PyExc_OSError, object)`。该函数将总是返回 `NULL`。

Availability: Windows.

PyObject ***PyErr_SetExcFromWindowsErr**(*PyObject* *type, int ierr)

返回值：恒为 *NULL*。属于稳定 ABI *on Windows* 自 *3.7* 版起. 类似于*PyErr_SetFromWindowsErr()*，额外的参数指定要触发的异常类型。

Availability: Windows.

PyObject ***PyErr_SetFromWindowsErrWithFilename**(int ierr, const char *filename)

返回值：恒为 *NULL*。属于稳定 ABI *on Windows* 自 *3.7* 版起. 与*PyErr_SetFromWindowsErr()* 类似，额外的不同点是如果 *filename* 不为 `NULL`，则会使用文件系统编码格式 (`os.fsdecode()`) 进行解码并作为第三个参数传递给 `OSError` 的构造器用于定义异常实例的 `filename` 属性。

Availability: Windows.

PyObject ***PyErr_SetExcFromWindowsErrWithFilenameObject**(*PyObject* *type, int ierr, *PyObject* *filename)

返回值：恒为 *NULL*。属于稳定 ABI *on Windows* 自 *3.7* 版起. 与*PyErr_SetExcFromWindowsErr()* 类似，额外的不同点是如果 *filename* 不为 `NULL`，它将作为第三个参数传递给 `OSError` 的构造器用于定义异常实例的 `filename` 属性。

Availability: Windows.

PyObject ***PyErr_SetExcFromWindowsErrWithFilenameObjects**(*PyObject* *type, int ierr, *PyObject* *filename, *PyObject* *filename2)

返回值：恒为 *NULL*。属于稳定 ABI *on Windows* 自 *3.7* 版起. 类似于*PyErr_SetExcFromWindowsErrWithFilenameObject()*，但是接受第二个 filename 对象。

Availability: Windows.

Added in version 3.4.

PyObject ***PyErr_SetExcFromWindowsErrWithFilename**(*PyObject* *type, int ierr, const char *filename)

返回值：恒为 *NULL*。属于稳定 ABI *on Windows* 自 *3.7* 版起. 类似于*PyErr_SetFromWindowsErrWithFilename()*，额外参数指定要触发的异常类型。

Availability: Windows.

PyObject ***PyErr_SetImportError**(*PyObject* *msg, *PyObject* *name, *PyObject* *path)

返回值：恒为 *NULL*。属于稳定 ABI 自 *3.7* 版起. 这是触发 `ImportError` 的便捷函数。*msg* 将被设为异常的消息字符串。*name* 和 *path*，（都可以为 `NULL`），将用来被设置 `ImportError` 对应的属性 `name` 和 `path`。

Added in version 3.3.

PyObject ***PyErr_SetImportErrorSubclass**(*PyObject* *exception, *PyObject* *msg, *PyObject* *name, *PyObject* *path)

返回值：恒为 *NULL*。属于稳定 ABI 自 *3.6* 版起. 和*PyErr_SetImportError()* 很类似，但这个函数允许指定一个 `ImportError` 的子类来触发。

Added in version 3.6.

void **PyErr_SyntaxLocationObject**(*PyObject* *filename, int lineno, int col_offset)

设置当前异常的文件，行和偏移信息。如果当前异常不是 `SyntaxError`，则它设置额外的属性，使异常打印子系统认为异常是 `SyntaxError`。

Added in version 3.4.

void **PyErr_SyntaxLocationEx**(const char *filename, int lineno, int col_offset)

属于稳定 ABI 自 *3.7* 版起. 类似于*PyErr_SyntaxLocationObject()*，但 *filename* 是用*filesystem encoding and error handler* 解码的字节串。

Added in version 3.2.

void **PyErr_SyntaxLocation**(const char *filename, int lineno)

属于稳定 ABI. 类似于*PyErr_SyntaxLocationEx()*，但省略了 *col_offset* parameter 形参。

void **PyErr_BadInternalCall**()

属于稳定 ABI. 这是 `PyErr_SetString(PyExc_SystemError, message)` 的缩写，其中 *message* 表示使用了非法参数调用内部操作（例如，Python/C API 函数）。它主要用于内部使用。

5.3 发出警告

这些函数可以从 C 代码中发出警告。它们仿照了由 Python 模块 `warnings` 导出的那些函数。它们通常向 *sys.stderr* 打印一条警告信息；当然，用户也有可能已经指定将警告转换为错误，在这种情况下，它们将触发异常。也有可能由于警告机制出现问题，使得函数触发异常。如果没有触发异常，返回值为 `0`；如果触发异常，返回值为 `-1`。（无法确定是否实际打印了警告信息，也无法确定异常触发的原因。这是故意为之）。如果触发了异常，调用者应该进行正常的异常处理（例如，*Py_DECREF()* 持有引用并返回一个错误值）。

int **PyErr_WarnEx**(*PyObject* *category, const char *message, *Py_ssize_t* stack_level)

属于稳定 ABI. 发出一个警告信息。参数 *category* 是一个警告类别（见下面）或 `NULL`；*message* 是一个 UTF-8 编码的字符串。*stack_level* 是一个给出栈帧数量的正数；警告将从该栈帧中当前正在执行的代码行发出。*stack_level* 为 1 的是调用*PyErr_WarnEx()* 的函数，2 是在此之上的函数，以此类推。

警告类别必须是*PyExc_Warning* 的子类，*PyExc_Warning* 是*PyExc_Exception* 的子类；默认警告类别是*PyExc_RuntimeWarning* 。标准 Python 警告类别作为全局变量可用，所有其名称见警告类型 。

有关警告控制的信息，参见模块文档 `warnings` 和命令行文档中的 `-W` 选项。没有用于警告控制的 C API。

int **PyErr_WarnExplicitObject**(*PyObject* *category, *PyObject* *message, *PyObject* *filename, int lineno, *PyObject* *module, *PyObject* *registry)

发出一个对所有警告属性进行显式控制的警告消息。这是位于 Python 函数 `warnings.warn_explicit()` 外层的直接包装；请查看其文档了解详情。*module* 和 *registry* 参数可被设为 `NULL` 以得到相关文档所描述的默认效果。

Added in version 3.4.

int **PyErr_WarnExplicit**(*PyObject* *category, const char *message, const char *filename, int lineno, const char *module, *PyObject* *registry)

属于稳定 ABI. 类似于*PyErr_WarnExplicitObject()* 不过 *message* 和 *module* 是 UTF-8 编码的字符串，而 *filename* 是由*filesystem encoding and error handler* 解码的。

int **PyErr_WarnFormat**(*PyObject* *category, *Py_ssize_t* stack_level, const char *format, ...)

属于稳定 ABI. 类似于*PyErr_WarnEx()* 的函数，但使用*PyUnicode_FromFormat()* 来格式化警告消息。*format* 是使用 ASCII 编码的字符串。

Added in version 3.2.

int **PyErr_ResourceWarning**(*PyObject* *source, *Py_ssize_t* stack_level, const char *format, ...)

属于稳定 ABI 自 *3.6* 版起. 类似于*PyErr_WarnFormat()* 的函数，但 *category* 是 `ResourceWarning` 并且它会将 *source* 传给 `warnings.WarningMessage`。

Added in version 3.6.

5.4 查询错误指示器

PyObject ***PyErr_Occurred**()

返回值：借入的引用。属于稳定 ABI. 测试是否设置了错误指示器。如已设置，则返回异常 *type* (传给对某个 `PyErr_Set*` 函数或*PyErr_Restore()* 的最后一次调用的第一个参数)。如未设置，则返回 `NULL`。你并不会拥有对返回值的引用，因此你不需要对它执行*Py_DECREF()*。

调用时必须持有 GIL。

> **备注**
>
> 不要将返回值与特定的异常进行比较；请改为使用*PyErr_ExceptionMatches()*，如下所示。(比较很容易失败因为对于类异常来说，异常可能是一个实例而不是类，或者它可能是预期的异常的一个子类。)

int **PyErr_ExceptionMatches**(*PyObject* *exc)

属于稳定 ABI. 等价于 `PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)`。此函数应当只在实际设置了异常时才被调用；如果没有任何异常被引发则将发生非法内存访问。

int **PyErr_GivenExceptionMatches**(*PyObject* *given, *PyObject* *exc)

属于稳定 ABI. 如果 *given* 异常与 *exc* 中的异常类型相匹配则返回真值。如果 *exc* 是一个类对象，则当 *given* 是一个子类的实例时也将返回真值。如果 *exc* 是一个元组，则该元组（以及递归的子元组）中的所有异常类型都将被搜索进行匹配。

PyObject ***PyErr_GetRaisedException**(void)

返回值：新的引用。属于稳定 ABI 自 *3.12* 版起. 返回当前被引发的异常，同时清除错误指示器。如果错误指示器尚未设置则返回 `NULL`。

此函数会被需要捕获异常的代码，或需要临时保存和恢复错误指示器的代码所使用。

例如：

```
{
   PyObject *exc = PyErr_GetRaisedException();

   /* ... 可能产生其他错误的代码 ... */

   PyErr_SetRaisedException(exc);
}
```

参见

PyErr_GetHandledException()，保存当前正在处理的异常。

Added in version 3.12.

void **PyErr_SetRaisedException**(*PyObject* *exc)

属于稳定 ABI 自 *3.12* 版起. 将 *exc* 设为当前被引发的异常，如果已设置则清空现有的异常。

警告

此调用将偷取一个对 *exc* 的引用，它必须是一个有效的异常。

Added in version 3.12.

void **PyErr_Fetch**(*PyObject* **ptype, *PyObject* **pvalue, *PyObject* **ptraceback)

属于稳定 ABI. 自 3.12 版本弃用: 使用*PyErr_GetRaisedException()* 代替。

将错误指示符提取到三个变量中并传递其地址。如果未设置错误指示符，则将三个变量都设为 NULL。如果已设置，则将其清除并且你将得到对所提取的每个对象的引用。值和回溯对象可以为 NULL 即使类型对象不为空。

备注

此函数通常只被需要捕获异常或临时保存和恢复错误指示符的旧式代码所使用。

例如：

```
{
   PyObject *type, *value, *traceback;
   PyErr_Fetch(&type, &value, &traceback);

   /* ... 可能产生其他错误的代码 ... */

   PyErr_Restore(type, value, traceback);
}
```

void **PyErr_Restore**(*PyObject* *type, *PyObject* *value, *PyObject* *traceback)

属于稳定 ABI. 自 3.12 版本弃用: 请改用*PyErr_SetRaisedException()*。

根据 *type*, *value* 和 *traceback* 这三个对象设置错误指示符，如果已设置了错误指示符则先清除它。如果三个对象均为 NULL，则清除错误指示符。请不要传入 NULL 类型和非 NULL 的值或回溯。异常类型应当是一个类。请不要传入无效的异常类型或值。(违反这些规则将导致微妙的后继问题。) 此调用会带走对每个对象的引用：你必须在调用之前拥有对每个对象的引用并且在调用之后你将不再拥有这些引用。(如果你不理解这一点，就不要使用此函数。勿谓言之不预。)

> **备注**
>
> 此函数通常只被需要临时保存和恢复错误指示符的旧代码所使用。请使用*PyErr_Fetch()* 来保存当前的错误指示符。

void **PyErr_NormalizeException**(*PyObject* **exc, *PyObject* **val, *PyObject* **tb)

属于稳定 ABI. 自 3.12 版本弃用: 请改用*PyErr_GetRaisedException()*，以避免任何可能的去正规化。

在特定情况下，下面*PyErr_Fetch()* 所返回的值可以是“非正规化的”，即 `*exc` 是一个类对象而 `*val` 不是同一个类的实例。在这种情况下此函数可以被用来实例化类。如果值已经是正规化的，则不做任何操作。实现这种延迟正规化是为了提升性能。

> **备注**
>
> 此函数 不会隐式地在异常值上设置 `__traceback__` 属性。如果想要适当地设置回溯，还需要以下附加代码片段:
>
> ```
> if (tb != NULL) {
>   PyException_SetTraceback(val, tb);
> }
> ```

PyObject ***PyErr_GetHandledException**(void)

属于稳定 ABI 自 *3.11* 版起. 提取激活的异常实例，就如 `sys.exception()` 所返回的一样。这是指一个 已被捕获的异常，而不是刚被引发的异常。返回一个指向该异常的新引用或者 `NULL`。不会修改解释器的异常状态。Does not modify the interpreter's exception state.

> **备注**
>
> 此函数通常不会被需要处理异常的代码所使用。它可被使用的场合是当代码需要临时保存并恢复异常状态的时候。请使用*PyErr_SetHandledException()* 来恢复或清除异常状态。

Added in version 3.11.

void **PyErr_SetHandledException**(*PyObject* *exc)

属于稳定 ABI 自 *3.11* 版起. 设置激活的异常，就是从 `sys.exception()` 所获得的。这是指一个 已被捕获的异常，而不是刚被引发的异常。要清空异常状态，请传入 `NULL`。

> **备注**
>
> 此函数通常不会被需要处理异常的代码所使用。它被使用的场合是在代码需要临时保存并恢复异常状态的时候。请使用*PyErr_GetHandledException()* 来获取异常状态。

Added in version 3.11.

void **PyErr_GetExcInfo**(*PyObject* **ptype, *PyObject* **pvalue, *PyObject* **ptraceback)

属于稳定 ABI 自 *3.7* 版起. 提取旧式的异常信息表示形式，就是从 `sys.exc_info()` 所获得的。这是指一个 已被捕获的异常，而不是刚被引发的异常。返回分别指向三个对象的新引用，其中任何一个都可以为 `NULL`。不会修改异常信息的状态。此函数是为了向下兼容而保留的。更推荐使用*PyErr_GetHandledException()*。

> **备注**

此函数通常不会被需要处理异常的代码所使用。它被使用的场合是在代码需要临时保存并恢复异常状态的时候。请使用*PyErr_SetExcInfo()* 来恢复或清除异常状态。

Added in version 3.3.

void **PyErr_SetExcInfo**(*PyObject* *type, *PyObject* *value, *PyObject* *traceback)

属于稳定 ABI 自 *3.7* 版起. 设置异常信息，就是从 `sys.exc_info()` 所获得的，这是指一个 已被捕获的异常，而不是刚被引发的异常。此函数会偷取对参数的引用。要清空异常状态，请为所有三个参数传入 `NULL`。此函数是为了向下兼容而保留的。更推荐使用*PyErr_SetHandledException()*。

备注

此函数通常不会被需要处理异常的代码所使用。它被使用的场合是在代码需要临时保存并恢复异常状态的情况。请使用*PyErr_GetExcInfo()* 来读取异常状态。

Added in version 3.3.

在 3.11 版本发生变更: `type` 和 `traceback` 参数已不再被使用并且可以为 NULL。解释器现在会根据异常实例（即 `value` 参数）来推断出它们。此函数仍然会偷取对所有三个参数的引用。

5.5 信号处理

int **PyErr_CheckSignals**()

属于稳定 ABI. 这个函数与 Python 的信号处理交互。

如果在主 Python 解释器下从主线程调用该函数，它将检查是否向进程发送了信号，如果是，则唤起相应的信号处理器。如果支持 `signal` 模块，则可以唤起以 Python 编写的信号处理器。

该函数会尝试处理所有待处理信号，然后返回 `0`。但是，如果 Python 信号处理器引发了异常，则设置错误指示符并且函数将立即返回 `-1` (这样其他待处理信号可能还没有被处理：它们将在下次唤起*PyErr_CheckSignals()* 时被处理)。

如果函数从非主线程调用，或在非主 Python 解释器下调用，则它不执行任何操作并返回 0。

这个函数可以由希望被用户请求 (例如按 Ctrl-C) 中断的长时间运行的 C 代码调用。

备注

针对 `SIGINT` 的默认 Python 信号处理器会引发 `KeyboardInterrupt` 异常。

void **PyErr_SetInterrupt**()

属于稳定 ABI. 模拟一个 `SIGINT` 信号到达的效果。这等价于 `PyErr_SetInterruptEx(SIGINT)`。

备注

此函数是异步信号安全的。它可以不带*GIL* 并由 C 信号处理器来调用。

int **PyErr_SetInterruptEx**(int signum)

属于稳定 ABI 自 *3.10* 版起. 模拟一个信号到达的效果。当下次*PyErr_CheckSignals()* 被调用时，将会调用针对指定的信号编号的 Python 信号处理器。

此函数可由自行设置信号处理，并希望 Python 信号处理器会在请求中断时（例如当用户按下 Ctrl-C 来中断操作时）按照预期被唤起的 C 代码来调用。

如果给定的信号不是由 Python 来处理的 (即被设为 `signal.SIG_DFL` 或 `signal.SIG_IGN`)，它将会被忽略。

如果 *signum* 在被允许的信号编号范围之外，将返回 `-1`。在其他情况下，则返回 `0`。错误指示符绝不会被此函数所修改。

> **备注**
>
> 此函数是异步信号安全的。它可以不带*GIL* 并由 C 信号处理器来调用。

Added in version 3.10.

int **PySignal_SetWakeupFd**(int fd)

这个工具函数指定了一个每当收到信号时将被作为以单个字节的形式写入信号编号的目标的文件描述符。*fd* 必须是非阻塞的。它将返回前一个这样的文件描述符。

设置值 `-1` 将禁用该特性；这是初始状态。这等价于 Python 中的 `signal.set_wakeup_fd()`，但是没有任何错误检查。*fd* 应当是一个有效的文件描述符。此函数应当只从主线程来调用。

在 3.5 版本发生变更: 在 Windows 上，此函数现在也支持套接字处理。

5.6 Exception 类

PyObject ***PyErr_NewException**(const char *name, *PyObject* *base, *PyObject* *dict)

返回值：新的引用。属于稳定 ABI. 这个工具函数会创建并返回一个新的异常类。*name* 参数必须为新异常的名称，是 `module.classname` 形式的 C 字符串。*base* 和 *dict* 参数通常为 `NULL`。这将创建一个派生自 `Exception` 的类对象（在 C 中可以通过*PyExc_Exception* 访问）。

新类的 `__module__` 属性将被设为 *name* 参数的前半部分（最后一个点号之前），而类名将被设为后半部分（最后一个点号之后）。*base* 参数可被用来指定替代基类；它可以是一个类或是一个由类组成的元组。*dict* 参数可被用来指定一个由类变量和方法组成的字典。

PyObject ***PyErr_NewExceptionWithDoc**(const char *name, const char *doc, *PyObject* *base, *PyObject* *dict)

返回值：新的引用。属于稳定 ABI. 和*PyErr_NewException()* 一样，除了可以轻松地给新的异常类一个文档字符串：如果 *doc* 属性非空，它将用作异常类的文档字符串。

Added in version 3.2.

int **PyExceptionClass_Check**(*PyObject* *ob)

如果 *ob* 是一个异常类则返回非零值，否则返回零。此函数总是会成功执行。

const char ***PyExceptionClass_Name**(*PyObject* *ob)

属于稳定 ABI 自 *3.8* 版起. 返回异常类 *ob* 的*tp_name*。

5.7 异常对象

PyObject ***PyException_GetTraceback**(*PyObject* *ex)

返回值：新的引用。属于稳定 ABI. 将与异常相关联的回溯作为一个新引用返回，可以通过 `__traceback__` 属性在 Python 中访问。如果没有已关联的回溯，则返回 `NULL`。

int **PyException_SetTraceback**(*PyObject* *ex, *PyObject* *tb)

属于稳定 ABI. 将异常关联的回溯设置为 *tb* 。使用 `Py_None` 清除它。

PyObject ***PyException_GetContext**(*PyObject* *ex)

返回值：新的引用。属于稳定 ABI. 将与异常相关联的上下文（在处理 *ex* 过程中引发的另一个异常实例）作为一个新引用返回，可以通过 `__context__` 属性在 Python 中访问。如果没有已关联的上下文，则返回 `NULL`。

void **PyException_SetContext**(*PyObject* *ex, *PyObject* *ctx)

属于稳定 ABI. 将与异常相关联的上下文设置为 *ctx*。使用 `NULL` 来清空它。没有用来确保 *ctx* 是一个异常实例的类型检查。这将窃取一个指向 *ctx* 的引用。

PyObject ***PyException_GetCause**(*PyObject* *ex)

返回值：新的引用。属于稳定 ABI. 将与异常相关联的原因（一个异常实例，或为 `None`，由 `raise ... from ...` 设置）作为一个新引用返回，可通过 `__cause__` 属性在 Python 中访问。

void **PyException_SetCause**(*PyObject* *ex, *PyObject* *cause)

属于稳定 ABI. 将与异常相关联的原因设为 *cause*。使用 `NULL` 来清空它。不存在类型检查用来确保 *cause* 是一个异常实例或为 `None`。这个偷取一个指向 *cause* 的引用。

`__suppress_context__` 属性会被此函数隐式地设为 `True`。

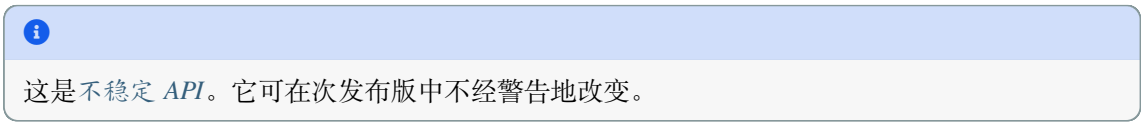
PyObject ***PyException_GetArgs**(*PyObject* *ex)

返回值：新的引用。属于稳定 ABI 自 *3.12* 版起. 返回异常 *ex* 的 `args`。

void **PyException_SetArgs**(*PyObject* *ex, *PyObject* *args)

属于稳定 ABI 自 *3.12* 版起. 将异常 *ex* 的 `args` 设为 *args*。

PyObject ***PyUnstable_Exc_PrepReraiseStar**(*PyObject* *orig, *PyObject* *excs)



这是不稳定 *API*。它可在次发布版中不经警告地改变。

解释器的 `except*` 实现的具体实现部分。*orig* 是被捕获的原始异常，而 *excs* 是需要被引发的异常组成的列表。该列表包含 *orig* 可能存在的未被处理的部分，以及在 `except*` 子句中被引发的异常（因而它们具有与 *orig* 不同的回溯数据）和被重新引发的异常（因而它们具有与 *orig* 相同的回溯）。返回需要被最终引发的 `ExceptionGroup`，或者如果没有要被引发的异常则返回 `None`。

Added in version 3.12.

5.8 Unicode 异常对象

下列函数被用于创建和修改来自 C 的 Unicode 异常。

PyObject ***PyUnicodeDecodeError_Create**(const char *encoding, const char *object, *Py_ssize_t* length, *Py_ssize_t* start, *Py_ssize_t* end, const char *reason)

返回值：新的引用。属于稳定 ABI. 创建一个 `UnicodeDecodeError` 对象并附带 *encoding*, *object*, *length*, *start*, *end* 和 *reason* 等属性。*encoding* 和 *reason* 为 UTF-8 编码的字符串。

PyObject ***PyUnicodeDecodeError_GetEncoding**(*PyObject* *exc)

PyObject ***PyUnicodeEncodeError_GetEncoding**(*PyObject* *exc)

返回值：新的引用。属于稳定 ABI. 返回给定异常对象的 *encoding* 属性

PyObject ***PyUnicodeDecodeError_GetObject**(*PyObject* *exc)

PyObject ***PyUnicodeEncodeError_GetObject**(*PyObject* *exc)

PyObject ***PyUnicodeTranslateError_GetObject**(*PyObject* *exc)

返回值：新的引用。属于稳定 ABI. 返回给定异常对象的 *object* 属性

int **PyUnicodeDecodeError_GetStart**(*PyObject* *exc, *Py_ssize_t* *start)

int **PyUnicodeEncodeError_GetStart**(*PyObject* *exc, *Py_ssize_t* *start)

int **PyUnicodeTranslateError_GetStart**(*PyObject* *exc, *Py_ssize_t* *start)

属于稳定 ABI. 获取给定异常对象的 *start* 属性并将其放入 **start*。*start* 必须不为 `NULL`。成功时返回 `0`，失败时返回 `-1`。

int **PyUnicodeDecodeError_SetStart**(*PyObject* *exc, *Py_ssize_t* start)

int **PyUnicodeEncodeError_SetStart**(*PyObject* *exc, *Py_ssize_t* start)

int **PyUnicodeTranslateError_SetStart**(*PyObject* *exc, *Py_ssize_t* start)

属于稳定 ABI. 将给定异常对象的 *start* 属性设为 *start*。成功时返回 0，失败时返回 -1。

int **PyUnicodeDecodeError_GetEnd**(*PyObject* *exc, *Py_ssize_t* *end)

int **PyUnicodeEncodeError_GetEnd**(*PyObject* *exc, *Py_ssize_t* *end)

int **PyUnicodeTranslateError_GetEnd**(*PyObject* *exc, *Py_ssize_t* *end)

属于稳定 ABI. 获取给定异常对象的 *end* 属性并将其放入 **end*。*end* 必须不为 NULL。成功时返回 0，失败时返回 -1。

int **PyUnicodeDecodeError_SetEnd**(*PyObject* *exc, *Py_ssize_t* end)

int **PyUnicodeEncodeError_SetEnd**(*PyObject* *exc, *Py_ssize_t* end)

int **PyUnicodeTranslateError_SetEnd**(*PyObject* *exc, *Py_ssize_t* end)

属于稳定 ABI. 将给定异常对象的 *end* 属性设为 *end*。成功时返回 0，失败时返回 -1。

PyObject ***PyUnicodeDecodeError_GetReason**(*PyObject* *exc)

PyObject ***PyUnicodeEncodeError_GetReason**(*PyObject* *exc)

PyObject ***PyUnicodeTranslateError_GetReason**(*PyObject* *exc)

返回值：新的引用。属于稳定 ABI. 返回给定异常对象的 *reason* 属性

int **PyUnicodeDecodeError_SetReason**(*PyObject* *exc, const char *reason)

int **PyUnicodeEncodeError_SetReason**(*PyObject* *exc, const char *reason)

int **PyUnicodeTranslateError_SetReason**(*PyObject* *exc, const char *reason)

属于稳定 ABI. 将给定异常对象的 *reason* 属性设为 *reason*。成功时返回 0，失败时返回 -1。

5.9 递归控制

这两个函数提供了一种在 C 层级上进行安全的递归调用的方式，在核心模块与扩展模块中均适用。当递归代码不一定会唤起 Python 代码（后者会自动跟踪其递归深度）时就需要用到它们。它们对于 *tp_call* 实现来说也无必要因为调用协议 会负责递归处理。

int **Py_EnterRecursiveCall**(const char *where)

属于稳定 ABI 自 *3.9* 版起. 标记一个递归的 C 层级调用即将被执行的点位。

如果定义了 USE_STACKCHECK，此函数会使用*PyOS_CheckStack()* 来检查 OS 栈是否溢出。如果发生了这种情况，它将设置一个 MemoryError 并返回非零值。

随后此函数将检查是否达到递归限制。如果是的话，将设置一个 RecursionError 并返回一个非零值。在其他情况下，则返回零。

where 应为一个 UTF-8 编码的字符串如 " in instance check"，它将与由递归深度限制所导致的 RecursionError 消息相拼接。

在 3.9 版本发生变更: 此函数现在也在受限 *API* 中可用。

void **Py_LeaveRecursiveCall**(void)

属于稳定 ABI 自 *3.9* 版起. 结束一个*Py_EnterRecursiveCall()*。必须针对*Py_EnterRecursiveCall()* 的每个成功的唤起操作执行一次调用。

在 3.9 版本发生变更: 此函数现在也在受限 *API* 中可用。

正确地针对容器类型实现*tp_repr* 需要特别的递归处理。在保护栈之外，*tp_repr* 还需要追踪对象以防止出现循环。以下两个函数将帮助完成此功能。从实际效果来说，这两个函数是 C 中对应 reprlib.recursive_repr() 的等价物。

int **Py_ReprEnter**(*PyObject* *object)

属于稳定 ABI. 在*tp_repr* 实现的开头被调用以检测循环。

如果对象已经被处理，此函数将返回一个正整数。在此情况下*tp_repr* 实现应当返回一个指明发生循环的字符串对象。例如，dict 对象将返回 {...} 而 list 对象将返回 [...]。

如果已达到递归限制则此函数将返回一个负正数。在此情况下*tp_repr* 实现通常应当返回 NULL。

在其他情况下，此函数将返回零而*tp_repr* 实现将可正常继续。

void **Py_ReprLeave**(*PyObject* *object)

属于稳定 ABI. 结束一个*Py_ReprEnter()*。必须针对每个返回零的*Py_ReprEnter()* 的唤起操作调用一次。

5.10 异常与警告类型

所有的标准 Python 异常和警告类别都可以用作全局变量，其名称为 PyExc_ 加上 Python 异常名称。这些类型是 *PyObject** 类型；它们都是类对象。

为完整说明，以下是全部的变量：

5.10.1 异常类型

C 名称	Python 名称
PyObject ***PyExc_BaseException** 属于稳定 ABI.	BaseException
PyObject ***PyExc_BaseExceptionGroup** 属于稳定 ABI 自 *3.11* 版起.	BaseExceptionGroup
PyObject ***PyExc_Exception** 属于稳定 ABI.	Exception
PyObject ***PyExc_ArithmeticError** 属于稳定 ABI.	ArithmeticError
PyObject ***PyExc_AssertionError** 属于稳定 ABI.	AssertionError
PyObject ***PyExc_AttributeError** 属于稳定 ABI.	AttributeError
PyObject ***PyExc_BlockingIOError** 属于稳定 ABI 自 *3.7* 版起.	BlockingIOError
PyObject ***PyExc_BrokenPipeError** 属于稳定 ABI 自 *3.7* 版起.	BrokenPipeError
PyObject ***PyExc_BufferError** 属于稳定 ABI.	BufferError
PyObject ***PyExc_ChildProcessError** 属于稳定 ABI 自 *3.7* 版起.	ChildProcessError

续下页

表 1 – 接上页

C 名称	Python 名称
PyObject ***PyExc_ConnectionAbortedError** 属于稳定 ABI 自 *3.7* 版起.	`ConnectionAbortedError`
PyObject ***PyExc_ConnectionError** 属于稳定 ABI 自 *3.7* 版起.	`ConnectionError`
PyObject ***PyExc_ConnectionRefusedError** 属于稳定 ABI 自 *3.7* 版起.	`ConnectionRefusedError`
PyObject ***PyExc_ConnectionResetError** 属于稳定 ABI 自 *3.7* 版起.	`ConnectionResetError`
PyObject ***PyExc_EOFError** 属于稳定 ABI.	`EOFError`
PyObject ***PyExc_FileExistsError** 属于稳定 ABI 自 *3.7* 版起.	`FileExistsError`
PyObject ***PyExc_FileNotFoundError** 属于稳定 ABI 自 *3.7* 版起.	`FileNotFoundError`
PyObject ***PyExc_FloatingPointError** 属于稳定 ABI.	`FloatingPointError`
PyObject ***PyExc_GeneratorExit** 属于稳定 ABI.	`GeneratorExit`
PyObject ***PyExc_ImportError** 属于稳定 ABI.	`ImportError`
PyObject ***PyExc_IndentationError** 属于稳定 ABI.	`IndentationError`
PyObject ***PyExc_IndexError** 属于稳定 ABI.	`IndexError`
PyObject ***PyExc_InterruptedError** 属于稳定 ABI 自 *3.7* 版起.	`InterruptedError`

续下页

表 1 – 接上页

C 名称	Python 名称
PyObject ***PyExc_IsADirectoryError** 属于稳定 ABI 自 *3.7* 版起.	`IsADirectoryError`
PyObject ***PyExc_KeyError** 属于稳定 ABI.	`KeyError`
PyObject ***PyExc_KeyboardInterrupt** 属于稳定 ABI.	`KeyboardInterrupt`
PyObject ***PyExc_LookupError** 属于稳定 ABI.	`LookupError`
PyObject ***PyExc_MemoryError** 属于稳定 ABI.	`MemoryError`
PyObject ***PyExc_ModuleNotFoundError** 属于稳定 ABI 自 *3.6* 版起.	`ModuleNotFoundError`
PyObject ***PyExc_NameError** 属于稳定 ABI.	`NameError`
PyObject ***PyExc_NotADirectoryError** 属于稳定 ABI 自 *3.7* 版起.	`NotADirectoryError`
PyObject ***PyExc_NotImplementedError** 属于稳定 ABI.	`NotImplementedError`
PyObject ***PyExc_OSError** 属于稳定 ABI.	`OSError`
PyObject ***PyExc_OverflowError** 属于稳定 ABI.	`OverflowError`
PyObject ***PyExc_PermissionError** 属于稳定 ABI 自 *3.7* 版起.	`PermissionError`
PyObject ***PyExc_ProcessLookupError** 属于稳定 ABI 自 *3.7* 版起.	`ProcessLookupError`
PyObject ***PyExc_PythonFinalizationError**	`PythonFinalizationError`

续下页

表 1 – 接上页

C 名称	Python 名称
PyObject ***PyExc_RecursionError** 属于稳定 ABI 自 *3.7* 版起.	`RecursionError`
PyObject ***PyExc_ReferenceError** 属于稳定 ABI.	`ReferenceError`
PyObject ***PyExc_RuntimeError** 属于稳定 ABI.	`RuntimeError`
PyObject ***PyExc_StopAsyncIteration** 属于稳定 ABI 自 *3.7* 版起.	`StopAsyncIteration`
PyObject ***PyExc_StopIteration** 属于稳定 ABI.	`StopIteration`
PyObject ***PyExc_SyntaxError** 属于稳定 ABI.	`SyntaxError`
PyObject ***PyExc_SystemError** 属于稳定 ABI.	`SystemError`
PyObject ***PyExc_SystemExit** 属于稳定 ABI.	`SystemExit`
PyObject ***PyExc_TabError** 属于稳定 ABI.	`TabError`
PyObject ***PyExc_TimeoutError** 属于稳定 ABI 自 *3.7* 版起.	`TimeoutError`
PyObject ***PyExc_TypeError** 属于稳定 ABI.	`TypeError`
PyObject ***PyExc_UnboundLocalError** 属于稳定 ABI.	`UnboundLocalError`
PyObject ***PyExc_UnicodeDecodeError** 属于稳定 ABI.	`UnicodeDecodeError`

续下页

表 1 – 接上页

C 名称	Python 名称
PyObject ***PyExc_UnicodeEncodeError** 属于稳定 ABI.	`UnicodeEncodeError`
PyObject ***PyExc_UnicodeError** 属于稳定 ABI.	`UnicodeError`
PyObject ***PyExc_UnicodeTranslateError** 属于稳定 ABI.	`UnicodeTranslateError`
PyObject ***PyExc_ValueError** 属于稳定 ABI.	`ValueError`
PyObject ***PyExc_ZeroDivisionError** 属于稳定 ABI.	`ZeroDivisionError`

Added in version 3.3: *PyExc_BlockingIOError*, *PyExc_BrokenPipeError*, *PyExc_ChildProcessError*, *PyExc_ConnectionError*, *PyExc_ConnectionAbortedError*, *PyExc_ConnectionRefusedError*, *PyExc_ConnectionResetError*, *PyExc_FileExistsError*, *PyExc_FileNotFoundError*, *PyExc_InterruptedError*, *PyExc_IsADirectoryError*, *PyExc_NotADirectoryError*, *PyExc_PermissionError*, *PyExc_ProcessLookupError* and *PyExc_TimeoutError* 介绍如下 **PEP 3151**.

Added in version 3.5: *PyExc_StopAsyncIteration* 和*PyExc_RecursionError*.

Added in version 3.6: *PyExc_ModuleNotFoundError*.

Added in version 3.11: *PyExc_BaseExceptionGroup*.

5.10.2 OSError 别名

以下是*PyExc_OSError* 的兼容性别名。

在 3.3 版本发生变更: 这些别名曾经是单独的异常类型。

C 名称	Python 名称	备注
PyObject ***PyExc_EnvironmentError** 属于稳定 ABI.	`OSError`	
PyObject ***PyExc_IOError** 属于稳定 ABI.	`OSError`	
PyObject ***PyExc_WindowsError** 属于稳定 ABI *on Windows* 自 *3.7* 版起.	`OSError`	[win]

注：

5.10.3 警告类型

C 名称	Python 名称
PyObject ***PyExc_Warning** 属于稳定 ABI.	`Warning`
PyObject ***PyExc_BytesWarning** 属于稳定 ABI.	`BytesWarning`
PyObject ***PyExc_DeprecationWarning** 属于稳定 ABI.	`DeprecationWarning`
PyObject ***PyExc_EncodingWarning** 属于稳定 ABI 自 *3.10* 版起.	`EncodingWarning`
PyObject ***PyExc_FutureWarning** 属于稳定 ABI.	`FutureWarning`
PyObject ***PyExc_ImportWarning** 属于稳定 ABI.	`ImportWarning`
PyObject ***PyExc_PendingDeprecationWarning** 属于稳定 ABI.	`PendingDeprecationWarning`
PyObject ***PyExc_ResourceWarning** 属于稳定 ABI 自 *3.7* 版起.	`ResourceWarning`
PyObject ***PyExc_RuntimeWarning** 属于稳定 ABI.	`RuntimeWarning`
PyObject ***PyExc_SyntaxWarning** 属于稳定 ABI.	`SyntaxWarning`
PyObject ***PyExc_UnicodeWarning** 属于稳定 ABI.	`UnicodeWarning`
PyObject ***PyExc_UserWarning** 属于稳定 ABI.	`UserWarning`

Added in version 3.2: *PyExc_ResourceWarning*.

Added in version 3.10: *PyExc_EncodingWarning*.

CHAPTER 6

工具

本章中的函数执行各种实用工具任务，包括帮助 C 代码提升跨平台可移植性，在 C 中使用 Python 模块，以及解析函数参数并根据 C 中的值构建 Python 中的值等等。

6.1 操作系统实用工具

PyObject ***PyOS_FSPath**(*PyObject* *path)

返回值：新的引用。属于稳定 ABI 自 *3.6* 版起. 返回 *path* 在文件系统中的表示形式。如果该对象是一个 `str` 或 `bytes` 对象，则返回一个新的*strong reference*。如果对象实现了 `os.PathLike` 接口，则只要它是一个 `str` 或 `bytes` 对象就将返回 `__fspath__()`。在其他情况下将引发 `TypeError` 并返回 `NULL`。

Added in version 3.6.

int **Py_FdIsInteractive**(FILE *fp, const char *filename)

如果名称为 *filename* 的标准 Return true (nonzero) if the standard I/O 文件 *fp* 被确认为可交互的则返回真（非零）值。所有 `isatty(fileno(fp))` 为真值的文件都属于这种情况。如果*PyConfig.interactive* 为非零值，此函数在 *filename* 指针为 `NULL` 或者其名称等于字符串 `'<stdin>'` 或 `'???'` 之一时也将返回真值。

此函数不可在 Python 被初始化之前调用。

void **PyOS_BeforeFork**()

属于稳定 ABI *on platforms with fork()* 自 *3.7* 版起. 在进程分叉之前准备某些内部状态的函数。此函数应当在调用 `fork()` 或者任何类似的克隆当前进程的函数之前被调用。只适用于定义了 `fork()` 的系统。

> **警告**
>
> C `fork()` 调用应当只在*"main" 线程* (位于*"main" 解释器*) 中进行。对于 `PyOS_BeforeFork()` 来说也是如此。

Added in version 3.7.

void **PyOS_AfterFork_Parent**()

属于稳定 ABI *on platforms with fork()* 自 *3.7* 版起. 在进程分叉之后更新某些内部状态的函数。此函

数应当在调用 `fork()` 或任何类似的克隆当前进程的函数之后被调用，无论进程克隆是否成功。只适用于定义了 `fork()` 的系统。

警告

C `fork()` 调用应当只在"main" 线程 (位于"main" 解释器) 中进行。对于 `PyOS_AfterFork_Parent()` 来说也是如此。

Added in version 3.7.

void **PyOS_AfterFork_Child**()

属于稳定 ABI *on platforms with fork()* 自 *3.7* 版起. 在进程分叉之后更新内部解释器状态的函数。此函数必须在调用 `fork()` 或任何类似的克隆当前进程的函数之后在子进程中被调用，如果该进程有机会回调到 Python 解释器的话。只适用于定义了 `fork()` 的系统。

警告

C `fork()` 调用应当只在"main" 线程 (位于"main" 解释器) 中进行。对于 `PyOS_AfterFork_Child()` 来说也是如此。

Added in version 3.7.

参见

`os.register_at_fork()` 允许注册可被*PyOS_BeforeFork()*, *PyOS_AfterFork_Parent()* 和*PyOS_AfterFork_Child()* 调用的自定义 Python 函数。

void **PyOS_AfterFork**()

属于稳定 ABI *on platforms with fork()*. 在进程分叉之后更新某些内部状态的函数；如果要继续使用 Python 解释器则此函数应当在新进程中被调用。如果已将一个新的可执行文件载入到新进程中，则不需要调用此函数。

自 3.7 版本弃用: 此函数已被*PyOS_AfterFork_Child()* 取代。

int **PyOS_CheckStack**()

属于稳定 ABI *on platforms with USE_STACKCHECK* 自 *3.7* 版起. 当解释器耗尽栈空间时返回真值。这是一个可靠的检测，但仅在定义了 `USE_STACKCHECK` 时可用（目前是在使用 Microsoft Visual C++ 编译器的特定 Windows 版本上）。`USE_STACKCHECK` 将被自动定义；你绝不应该在你自己的代码中改变此定义。

typedef void (***PyOS_sighandler_t**)(int)

属于稳定 ABI.

PyOS_sighandler_t **PyOS_getsig**(int i)

属于稳定 ABI. 返回信号 *i* 当前的信号处理器。这是一个对 `sigaction()` 或 `signal()` 的简单包装器。请不要直接调用这两个函数！

PyOS_sighandler_t **PyOS_setsig**(int i, *PyOS_sighandler_t* h)

属于稳定 ABI. 将信号 *i* 的信号处理器设为 *h*；返回原来的信号处理器。这是一个对 `sigaction()` 或 `signal()` 的简单包装器。请不要直接调用这两个函数！

wchar_t ***Py_DecodeLocale**(const char *arg, size_t *size)

属于稳定 ABI 自 *3.7* 版起.

警告

此函数不应当被直接调用：请使用*PyConfig* API 以及可确保对 *Python* 进行预初始化的*PyConfig_SetBytesString()* 函数。

此函数不可在 This function must not be called before 对 *Python* 进行预初始化 之前被调用以便正确地配置 LC_CTYPE 语言区域：请参阅*Py_PreInitialize()* 函数。

使用*filesystem encoding and error handler* 来解码一个字节串。如果错误处理器为 surrogateescape 错误处理器，则不可解码的字节将被解码为 U+DC80..U+DCFF 范围内的字符；而如果一个字节序列可被解码为代理字符，则其中的字节会使用 surrogateescape 错误处理器来转义而不是解码它们。

返回一个指向新分配的由宽字符组成的字符串的指针，使用*PyMem_RawFree()* 来释放内存。如果 size 不为 `NULL`，则将排除了 null 字符的宽字符数量写入到 `*size`

在解码错误或内存分配错误时返回 `NULL`。如果 *size* 不为 `NULL`，则 `*size` 将在内存错误时设为 `(size_t)-1` 或在解码错误时设为 `(size_t)-2`。

filesystem encoding and error handler 是由*PyConfig_Read()* 来选择的：参见*PyConfig* 的*filesystem_encoding* 和*filesystem_errors* 等成员。

解码错误绝对不应当发生，除非 C 库有程序缺陷。

请使用*Py_EncodeLocale()* 函数来将字符串编码回字节串。

参见

PyUnicode_DecodeFSDefaultAndSize() 和*PyUnicode_DecodeLocaleAndSize()* 函数。

Added in version 3.5.

在 3.7 版本发生变更: 现在此函数在 Python UTF-8 模式下将使用 UTF-8 编码格式。

在 3.8 版本发生变更: 现在如果在 Windows 上*PyPreConfig.legacy_windows_fs_encoding* 为零则此函数将使用 UTF-8 编码格式；

char *__Py_EncodeLocale__(const wchar_t *text, size_t *error_pos)

属于稳定 ABI 自 *3.7* 版起. 将一个由宽字符组成的字符串编码为*filesystem encoding and error handler*。如果错误处理器为 surrogateescape 错误处理器，则在 U+DC80..U+DCFF 范围内的代理字符会被转换为字节值 0x80..0xFF。

返回一个指向新分配的字节串的指针，使用*PyMem_Free()* 来释放内存。当发生编码错误或内存分配错误时返回 `NULL`。

如果 error_pos 不为 `NULL`，则成功时会将 `*error_pos` 设为 `(size_t)-1`，或是在发生编码错误时设为无效字符的索引号。

filesystem encoding and error handler 是由*PyConfig_Read()* 来选择的：参见*PyConfig* 的*filesystem_encoding* 和*filesystem_errors* 等成员。

请使用*Py_DecodeLocale()* 函数来将字节串解码回由宽字符组成的字符串。

警告

此函数不可在 This function must not be called before 对 *Python* 进行预初始化 之前被调用以便正确地配置 LC_CTYPE 语言区域：请参阅*Py_PreInitialize()* 函数。

> **参见**
>
> *PyUnicode_EncodeFSDefault()* 和*PyUnicode_EncodeLocale()* 函数。

Added in version 3.5.

在 3.7 版本发生变更: 现在此函数在 Python UTF-8 模式下将使用 UTF-8 编码格式。

在 3.8 版本发生变更: 现在如果在 Windows 上*PyPreConfig.legacy_windows_fs_encoding* 为零则此函数将使用 UTF-8 编码格式。

6.2 系统功能

这些是使来自 sys 模块的功能可以让 C 代码访问的工具函数。它们都可用于当前解释器线程的 sys 模块的字典，该字典包含在内部线程状态结构体中。

PyObject ***PySys_GetObject**(const char *name)

返回值：借入的引用。属于稳定 ABI. 返回来自 sys 模块的对象 *name* 或者如果它不存在则返回 NULL，并且不会设置异常。

int **PySys_SetObject**(const char *name, *PyObject* *v)

属于稳定 ABI. 将 sys 模块中的 *name* 设为 *v* 除非 *v* 为 NULL，在此情况下 *name* 将从 sys 模块中被删除。成功时返回 0，发生错误时返回 -1。

void **PySys_ResetWarnOptions**()

属于稳定 ABI. 将 sys.warnoptions 重置为空列表。此函数可在*Py_Initialize()* 之前被调用。

Deprecated since version 3.13, will be removed in version 3.15: 改为清除 sys.warnoptions 和 warnings.filters。

void **PySys_WriteStdout**(const char *format, ...)

属于稳定 ABI. 将以 *format* 描述的输出字符串写入到 sys.stdout。不会引发任何异常，即使发生了截断（见下文）。

format 应当将已格式化的输出字符串的总大小限制在 1000 字节以下 -- 超过 1000 字节后，输出字符串会被截断。特别地，这意味着不应出现不受限制的"%s" 格式；它们应当使用"%.<N>s" 来限制，其中 <N> 是一个经计算使得 <N> 与其他已格式化文本的最大尺寸之和不会超过 1000 字节的十进制数字。还要注意"%f"，它可能为非常大的数字打印出数以百计的数位。

如果发生了错误，sys.stdout 会被清空，已格式化的消息将被写入到真正的 (C 层级) *stdout*。

void **PySys_WriteStderr**(const char *format, ...)

属于稳定 ABI. 类似*PySys_WriteStdout()*，但改为写入到 sys.stderr 或 *stderr*。

void **PySys_FormatStdout**(const char *format, ...)

属于稳定 ABI. 类似 PySys_WriteStdout() 的函数将会使用*PyUnicode_FromFormatV()* 来格式化消息并且不会将消息截短至任意长度。

Added in version 3.2.

void **PySys_FormatStderr**(const char *format, ...)

属于稳定 ABI. 类似*PySys_FormatStdout()*，但改为写入到 sys.stderr 或 *stderr*。

Added in version 3.2.

PyObject ***PySys_GetXOptions**()

返回值：借入的引用。属于稳定 ABI 自 *3.7* 版起. 返回当前 -X 选项的字典，类似于 sys._xoptions。发生错误时，将返回 NULL 并设置一个异常。

Added in version 3.2.

int **PySys_Audit**(const char *event, const char *format, ...)

属于稳定 ABI 自 *3.13* 版起. 引发一个审计事件并附带任何激活的钩子。成功时返回零值或在失败时返回非零值并设置一个异常。

event 字符串参数必须不为 *NULL*。

如果已添加了任何钩子，则将使用 *format* 和其他参数来构造一个要传入的元组。除 `N` 以外，还可使用在*Py_BuildValue()* 中使用的相同格式字符。如果构建的值不是一个元组，它将被添加到一个单元素的元组中。

不可使用 `N` 格式选项。它会消耗一个引用，但是由于无法获知传给此函数的参数是否会被消耗，使用它可能导致引用泄漏。

请注意 `#` 格式字符应当总是被当作*Py_ssize_t* 来处理，无论是否定义了 `PY_SSIZE_T_CLEAN`。

`sys.audit()` 会执行与来自 Python 代码的函数相同的操作。

另请参阅*PySys_AuditTuple()*。

Added in version 3.8.

在 3.8.2 版本发生变更: 要求*Py_ssize_t* 用于 `#` 格式字符。在此之前，会引发一个不可避免的弃用警告。

int **PySys_AuditTuple**(const char *event, *PyObject* *args)

属于稳定 ABI 自 *3.13* 版起. 与*PySys_Audit()* 类似，但会将参数作为 Python 对象传入。*args* 必须是一个 `tuple`。如果不传入参数，则 *args* 可以为 *NULL*。

Added in version 3.13.

int **PySys_AddAuditHook**(*Py_AuditHookFunction* hook, void *userData)

将可调用对象 *hook* 添加到激活的审计钩子列表。在成功时返回零而在失败时返回非零值。如果运行时已经被初始化，还会在失败时设置一个错误。通过此 API 添加的钩子会针对在运行时创建的所有解释器被调用。

userData 指针会被传入钩子函数。因于钩子函数可能由不同的运行时调用，该指针不应直接指向 Python 状态。

此函数可在*Py_Initialize()* 之前被安全地调用。如果在运行时初始化之后被调用，现有的审计钩子将得到通知并可能通过引发一个从 `Exception` 子类化的错误静默地放弃操作（其他错误将不会被静默）。

钩子函数总是会由引发异常的 Python 解释器在持有 GIL 的情况下调用。

请参阅 **PEP 578** 了解有关审计的详细描述。在运行时和标准库中会引发审计事件的函数清单见 审计事件表。更多细节见每个函数的文档。

如果解释器已被初始化，此函数将引发一个审计事件 `sys.addaudithook` 且不附带任何参数。如果有任何现存的钩子引发了一个派生自 `Exception` 的异常，新的钩子将不会被添加且该异常会被清除。因此，调用方不可假定他们的钩子已被添加除非他们能控制所有现存的钩子。

typedef int (***Py_AuditHookFunction**)(const char *event, *PyObject* *args, void *userData)

钩子函数的类型。*event* 是传给*PySys_Audit()* 或*PySys_AuditTuple()* 的 C 字符串形式的事件参数。*args* 会确保为一个*PyTupleObject*。*userData* 是传给 PySys_AddAuditHook() 的参数。

Added in version 3.8.

6.3 过程控制

void **Py_FatalError**(const char *message)

属于稳定 ABI. 打印一个致命错误消息并杀死进程。不会执行任何清理。此函数应当仅在检测到可能令继续使用 Python 解释器会有危险的情况时被唤起；例如对象管理已被破坏的时候。在 Unix 上，会调用标准 C 库函数 `abort()` 并将由它来尝试生成一个 `core` 文件。

The `Py_FatalError()` function is replaced with a macro which logs automatically the name of the current function, unless the `Py_LIMITED_API` macro is defined.

在 3.9 版本发生变更: 自动记录函数名称。

void **Py_Exit**(int status)

属于稳定 ABI. 退出当前进程。这将调用*Py_FinalizeEx()* 然后再调用标准 C 库函数 `exit(status)`。如果*Py_FinalizeEx()* 提示错误，退出状态将被设为 120。

在 3.6 版本发生变更: 来自最终化的错误不会再被忽略。

int **Py_AtExit**(void (*func)())

属于稳定 ABI. 注册一个由*Py_FinalizeEx()* 调用的清理函数。调用清理函数将不传入任何参数且不应返回任何值。最多可以注册 32 个清理函数。当注册成功时，*Py_AtExit()* 将返回 `0`；失败时，它将返回 `-1`。最后注册的清理函数会最先被调用。每个清理函数将至多被调用一次。由于 Python 的内部最终化将在清理函数之前完成，因此 Python API 不应被 *func* 调用。

参见

PyUnstable_AtExit() 用于传递 `void *data` 参数。

6.4 导入模块

PyObject ***PyImport_ImportModule**(const char *name)

返回值：新的引用。属于稳定 ABI. 这是一个对*PyImport_Import()* 的包装器，它接受一个 `const char*` 作为参数而不是 *PyObject**。

PyObject ***PyImport_ImportModuleNoBlock**(const char *name)

返回值：新的引用。属于稳定 ABI. 该函数是*PyImport_ImportModule()* 的一个被遗弃的别名。

在 3.3 版本发生变更: 在导入锁被另一线程掌控时此函数会立即失败。但是从 Python 3.3 起，锁方案在大多数情况下都已切换为针对每个模块加锁，所以此函数的特殊行为已无必要。

Deprecated since version 3.13, will be removed in version 3.15: 使用*PyImport_ImportModule()* 来代替。

PyObject ***PyImport_ImportModuleEx**(const char *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist)

返回值：新的引用。 导入一个模块。请参阅内置 Python 函数 `__import__()` 获取完善的相关描述。

返回值是一个对所导入模块或最高层级包的新引用，或是在导入失败时则为 `NULL` 并设置一个异常。与 `__import__()` 类似，当请求一个包的子模块时返回值通常为该最高层级包，除非给出了一个非空的 *fromlist*。

导入失败将移动不完整的模块对象，就像*PyImport_ImportModule()* 那样。

PyObject ***PyImport_ImportModuleLevelObject**(*PyObject* *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist, int level)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 导入一个模块。关于此函数的最佳说明请参考内置 Python 函数 `__import__()`，因为标准 `__import__()` 函数会直接调用此函数。

返回值是一个对所导入模块或最高层级包的新引用，或是在导入失败时则为 `NULL` 并设置一个异常。与 `__import__()` 类似，当请求一个包的子模块时返回值通常为该最高层级包，除非给出了一个非空的 *fromlist*。

Added in version 3.3.

PyObject ***PyImport_ImportModuleLevel**(const char *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist, int level)

返回值：新的引用。属于稳定 ABI. 类似于*PyImport_ImportModuleLevelObject()*，但其名称为 UTF-8 编码的字符串而不是 Unicode 对象。

在 3.3 版本发生变更: 不再接受 *level* 为负数值。

PyObject ***PyImport_Import**(*PyObject* *name)

返回值：新的引用。属于稳定 ABI. 这是一个调用了当前“导入钩子函数”的更高层级接口（显式指定 *level* 为 0，表示绝对导入）。它将唤起当前全局作用域下 `__builtins__` 中的 `__import__()` 函数。这意味着将使用当前环境下安装的任何导入钩子来完成导入。

该函数总是使用绝对路径导入。

PyObject ***PyImport_ReloadModule**(*PyObject* *m)

返回值：新的引用。属于稳定 ABI. 重载一个模块。返回一个指向被重载模块的新引用，或者在失败时返回 `NULL` 并设置一个异常（在此情况下模块仍然会存在）。

PyObject ***PyImport_AddModuleRef**(const char *name)

返回值：新的引用。属于稳定 ABI 自 *3.13* 版起. 返回对应于模块名称的模块对象。

name 参数的形式可以为 `package.module`。如果存在 modules 字典则首先检查它，如果不存在，则创建一个新模块并在 modules 字典中插入它。

成功时返回一个指向模块的*strong reference*。失败时返回 `NULL` 并设置一个异常。

模块名称 *name* 将使用 UTF-8 解码。

此函数不会加载或导入指定模块；如果模块还未被加载，你将得到一个空的模块对象。请使用*PyImport_ImportModule()* 或它的某个变体形式来导入模块。*name* 使用的带点号名称的包结构如果尚不存在则不会被创建。

Added in version 3.13.

PyObject ***PyImport_AddModuleObject**(*PyObject* *name)

返回值：借入的引用。属于稳定 ABI 自 *3.7* 版起. 类似于*PyImport_AddModuleRef()*，但会返回一个*borrowed reference* 并且 *name* 将是一个 Python `str` 对象。

Added in version 3.3.

PyObject ***PyImport_AddModule**(const char *name)

返回值：借入的引用。属于稳定 ABI. 类似于*PyImport_AddModuleRef()*，但会返回一个*borrowed reference*。

PyObject ***PyImport_ExecCodeModule**(const char *name, *PyObject* *co)

返回值：新的引用。属于稳定 ABI. 给定一个模块名称（可能为 `package.module` 形式）和一个从 Python 字节码文件读取或从内置函数 `compile()` 获取的代码对象，加载该模块。返回对该模块对象的新引用，或者如果发生错误则返回 `NULL` 并设置一个异常。在发生错误的情况下 *name* 会从 `sys.modules` 中被移除，即使 *name* 在进入*PyImport_ExecCodeModule()* 时已存在于 `sys.modules` 中。在 `sys.modules` 中保留未完全初始化的模块是危险的，因为导入这样的模块没有办法知识模块对象是否处于一种未知的（对于模块作者的意图来说可能是已损坏的）状态。

模块的 `__spec__` 和 `__loader__` 如果尚未设置，则会被设为适当的值。相应 spec 的加载器（如已设置）会被设为模块的 `__loader__` 而在其他情况下则会被设为 `SourceFileLoader` 的实例。

模块的 `__file__` 属性将被设为代码对象的 `co_filename`。如果适用，还将设置 `__cached__`。

如果模块已被导入则此函数将重载它。请参阅*PyImport_ReloadModule()* 了解重载模块的预定方式。

如果 *name* 指向一个形式为 `package.module` 的带点号的名称，则任何尚未创建的包结构仍然不会被创建。

另请参阅*PyImport_ExecCodeModuleEx()* 和*PyImport_ExecCodeModuleWithPathnames()*。

在 3.12 版本发生变更: 设置 `__cached__` 和 `__loader__` 的做法已被弃用。请参阅 `ModuleSpec` 了解替代方式。

PyObject ***PyImport_ExecCodeModuleEx**(const char *name, *PyObject* *co, const char *pathname)

返回值：新的引用。属于稳定 ABI. 类似于*PyImport_ExecCodeModule()*，但是如果 *pathname* 不为 `NULL` 则会将其设为模块对象 `__file__` 属性的值。

参见*PyImport_ExecCodeModuleWithPathnames()*。

PyObject ***PyImport_ExecCodeModuleObject**(*PyObject* *name, *PyObject* *co, *PyObject* *pathname, *PyObject* *cpathname)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 类似于*PyImport_ExecCodeModuleEx()*，但如果 *cpathname* 不为 NULL 则会将其设为模块对象 __cached__ 属性的值。在三个函数中，这是推荐使用的一个。

Added in version 3.3.

在 3.12 版本发生变更: 设置 __cached__ 的做法已被弃用。请参阅 ModuleSpec 了解替代方式。

PyObject ***PyImport_ExecCodeModuleWithPathnames**(const char *name, *PyObject* *co, const char *pathname, const char *cpathname)

返回值：新的引用。属于稳定 ABI. 类似于*PyImport_ExecCodeModuleObject()*，但 *name*, *pathname* 和 *cpathname* 为 UTF-8 编码的字符串。如果 *pathname* 也被设为 NULL 则还会尝试根据 *cpathname* 推断出前者的值。

Added in version 3.2.

在 3.3 版本发生变更: 如果只提供了字节码路径则会使用 imp.source_from_cache() 来计算源路径。

在 3.12 版本发生变更: 不再使用已移除的 imp 模块。

long **PyImport_GetMagicNumber**()

属于稳定 ABI. 返回 Python 字节码文件（即 .pyc 文件）的魔数。此魔数应当存在于字节码文件的开头四个字节中，按照小端字节序。出错时返回 -1。

在 3.3 版本发生变更: 失败时返回值 -1。

const char ***PyImport_GetMagicTag**()

属于稳定 ABI. 针对 **PEP 3147** 格式的 Python 字节码文件名返回魔术标签字符串。请记住在 sys.implementation.cache_tag 上的值是应当被用来代替此函数的更权威的值。

Added in version 3.2.

PyObject ***PyImport_GetModuleDict**()

返回值：借入的引用。属于稳定 ABI. 返回用于模块管理的字典 (即 sys.modules)。请注意这是针对每个解释器的变量。

PyObject ***PyImport_GetModule**(*PyObject* *name)

返回值：新的引用。属于稳定 ABI 自 *3.8* 版起. 返回给定名称的已导入模块。如果模块尚未导入则返回 NULL 但不会设置错误。如果查找失败则返回 NULL 并设置错误。

Added in version 3.7.

PyObject ***PyImport_GetImporter**(*PyObject* *path)

返回值：新的引用。属于稳定 ABI. 返回针对一个 sys.path/pkg.__path__ 中条目 *path* 的查找器对象，可能会从 sys.path_importer_cache 字典中获取。如果它尚未被缓存，则会遍历 sys.path_hooks 直至找到一个能处理该路径条目的钩子。如果没有可用的钩子则返回 None；这将告知调用方*path based finder* 无法为该路径条目找到查找器。结果将缓存到 sys.path_importer_cache 中。返回一个指向查找器对象的新引用。

int **PyImport_ImportFrozenModuleObject**(*PyObject* *name)

属于稳定 ABI 自 *3.7* 版起. 加载名称为 *name* 的已冻结模块。成功时返回 1，如果未找到模块则返回 0，如果初始化失败则返回 -1 并设置一个异常。要在加载成功后访问被导入的模块，请使用*PyImport_ImportModule()*。（请注意此名称有误导性 --- 如果模块已被导入此函数将重载它。）

Added in version 3.3.

在 3.4 版本发生变更: __file__ 属性将不再在模块上设置。

int **PyImport_ImportFrozenModule**(const char *name)

属于稳定 ABI. 类似于*PyImport_ImportFrozenModuleObject()*，但其名称为 UTF-8 编码的字符串而不是 Unicode 对象。

struct **_frozen**

这是针对已冻结模块描述器的结构类型定义，与由 **freeze** 工具所生成的一致 (请参看 Python 源代码发行版中的 `Tools/freeze/`)。其定义可在 `Include/import.h` 中找到:

```
struct _frozen {
    const char *name;
    const unsigned char *code;
    int size;
    bool is_package;
};
```

在 3.11 版本发生变更: 新的 `is_package` 字段指明模块是否为一个包。这替代了将 `size` 设为负值的做法。

const struct *_frozen* ***PyImport_FrozenModules**

该指针被初始化为指向一个 *_frozen* 记录的数组，以一个所有成员均为 `NULL` 或零的记录表示结束。当一个冻结模块被导入时，它将在此表中被搜索。第三方代码可以利用此方式来提供动态创建的冻结模块集。

int **PyImport_AppendInittab**(const char *name, *PyObject* *(*initfunc)(void))

属于稳定 ABI. 向现有的内置模块表添加一个模块。这是对 *PyImport_ExtendInittab()* 的便捷包装，如果无法扩展表则返回 `-1`。新的模块可使用名称 *name* 来导入，并使用函数 *initfunc* 作为在第一次尝试导入时调用的初始化函数。此函数应当在 *Py_Initialize()* 之前调用。

struct **_inittab**

描述内置模块列表中一个单独条目的结构体。嵌入 Python 的程序可以将这些结构体的数组与 *PyImport_ExtendInittab()* 结合使用以提供额外的内置模块。该结构体由两个成员组成:

const char ***name**

模块名称，为一个 ASCII 编码的字符串。

PyObject *(***initfunc**)(void)

针对内置于解释器的模块的初始化函数。

int **PyImport_ExtendInittab**(struct *_inittab* *newtab)

向内置模块表添加一组模块。*newtab* 数组必须以一个包含 `NULL` 作为 *name* 字段的哨兵条目结束；未提供哨兵值可能导致内存错误。成功时返回 `0` 或者如果无法分配足够内存来扩展内部表则返回 `-1`。当失败时，将不会向内部表添加任何模块。该函数必须在 *Py_Initialize()* 之前调用。

如果 Python 要被多次初始化，则 *PyImport_AppendInittab()* 或 *PyImport_ExtendInittab()* 必须在每次 Python 初始化之前调用。

6.5 数据 marshal 操作支持

这些例程允许 C 代码处理与 `marshal` 模块所用相同数据格式的序列化对象。其中有些函数可用来将数据写入这种序列化格式，另一些函数则可用来读取并恢复数据。用于存储 marshal 数据的文件必须以二进制模式打开。

数字值在存储时会将最低位字节放在开头。

此模块支持两种数据格式版本：第 0 版为历史版本，第 1 版本会在文件和 marshal 反序列化中共享固化的字符串。第 2 版本会对浮点数使用二进制格式。`Py_MARSHAL_VERSION` 指明了当前文件的格式（当前取值为 2）。

void **PyMarshal_WriteLongToFile**(long value, FILE *file, int version)

将一个 `long` 整数 *value* 以 marshal 格式写入 *file*。这将只写入 *value* 中最低的 32 个比特位；无论本机的 `long` 类型的大小如何。*version* 指明文件格式的版本。

此函数可能失败，在这种情况下它半设置错误提示符。请使用 *PyErr_Occurred()* 进行检测。

void **PyMarshal_WriteObjectToFile**(*PyObject* *value, FILE *file, int version)

将一个 Python 对象 *value* 以 marshal 格式写入 *file*。*version* 指明文件格式的版本。

此函数可能失败，在这种情况下它半设置错误提示符。请使用*PyErr_Occurred()* 进行检测。

PyObject ***PyMarshal_WriteObjectToString**(*PyObject* *value, int version)

返回值：新的引用。 返回一个包含 *value* 的 marshal 表示形式的字节串对象。*version* 指明文件格式的版本。

以下函数允许读取并恢复存储为 marshal 格式的值。

long **PyMarshal_ReadLongFromFile**(FILE *file)

从打开用于读取的 `FILE*` 对应的数据流返回一个 C `long`。使用此函数只能读取 32 位的值，无论本机 `long` 类型的大小如何。

发生错误时，将设置适当的异常 (`EOFError`) 并返回 `-1`。

int **PyMarshal_ReadShortFromFile**(FILE *file)

从打开用于读取的 `FILE*` 对应的数据流返回一个 C `short`。使用此函数只能读取 16 位的值，无论本机 `short` 类型的大小如何。

发生错误时，将设置适当的异常 (`EOFError`) 并返回 `-1`。

PyObject ***PyMarshal_ReadObjectFromFile**(FILE *file)

返回值：新的引用。 从打开用于读取的 `FILE*` 对应的数据流返回一个 Python 对象。

发生错误时，将设置适当的异常 (`EOFError`, `ValueError` 或 `TypeError`) 并返回 `NULL`。

PyObject ***PyMarshal_ReadLastObjectFromFile**(FILE *file)

返回值：新的引用。 从打开用于读取的 `FILE*` 对应的数据流返回一个 Python 对象。不同于*PyMarshal_ReadObjectFromFile()*，此函数假定将不再从该文件读取更多的对象，允许其将文件数据积极地载入内存，以便反序列化过程可以在内存中的数据上操作而不是每次从文件读取一个字节。只有当你确定不会再从文件读取任何内容时方可使用此形式。

发生错误时，将设置适当的异常 (`EOFError`, `ValueError` 或 `TypeError`) 并返回 `NULL`。

PyObject ***PyMarshal_ReadObjectFromString**(const char *data, *Py_ssize_t* len)

返回值：新的引用。 从包含指向 *data* 的 *len* 个字节的字节缓冲区对应的数据流返回一个 Python 对象。

发生错误时，将设置适当的异常 (`EOFError`, `ValueError` 或 `TypeError`) 并返回 `NULL`。

6.6 解析参数并构建值变量

这些函数在创建你自己的扩展函数和方法时很有用处。更多信息和示例可在 extending-index 查看。

这些函数描述的前三个，*PyArg_ParseTuple()*，*PyArg_ParseTupleAndKeywords()*，以及*PyArg_Parse()*，它们都使用格式化字符串来将函数期待的参数告知函数。这些函数都使用相同语法规则的格式化字符串。

6.6.1 解析参数

一个格式化字符串包含 0 或者更多的格式单元。一个格式单元用来描述一个 Python 对象；它通常是一个字符或者由括号括起来的格式单元序列。除了少数例外，一个非括号序列的格式单元通常对应这些函数的具有单一地址的参数。在接下来的描述中，双引号内的表达式是格式单元；圆括号 () 内的是对应这个格式单元的 Python 对象类型；方括号 [] 内的是传递的 C 变量 (变量集) 类型。

字符串和缓存区

> **备注**
>
> 在 Python 3.12 和之前的版本中，宏 PY_SSIZE_T_CLEAN 必须在包括 Python.h 之前定义以使用下文介绍的所有 # 类格式 (s#, y# 等)。这在 Python 3.13 和之后的版本中已不再必要。

这些格式允许将对象按照连续的内存块形式进行访问。你没必要提供返回的 unicode 字符或者字节区的原始数据存储。

除非另有说明，缓冲区是不会以空终止的。

有三种办法可以将字符串和缓冲区转换到 C 类型:

- 像 y* 和 s* 这样的格式会填充一个*Py_buffer* 结构体。这将锁定下层缓冲区以便调用者随后使用这个缓冲区即使是在*Py_BEGIN_ALLOW_THREADS* 块中也不会有可变数据因大小调整或销毁所带来的风险。因此，在你结束处理数据（或任何更早的中止场景）之前 **你必须调用***PyBuffer_Release()*。
- es, es#, et 和 et# 等格式会分配结果缓冲区。在你结束处理数据（或任何更早的中止场景）之后 **你必须调用***PyMem_Free()*。
- 其他格式接受一个 str 或只读的*bytes-like object*，如 bytes，并向其缓冲区提供一个 const char * 指针。在缓冲区是“被借入”的情况下：它将由对应的 Python 对象来管理，并共享此对象的生命期。你不需要自行释放任何内存。

  为确保下层缓冲区可以安全地被借入，对象的*PyBufferProcs.bf_releasebuffer* 字段必须为 NULL。这将不允许普通的可变对象如 bytearray，以及某些只读对象如 bytes 的 memoryview。

  在这个 bf_releasebuffer 要求以外，没有用于验证输入对象是否为不可变对象的检查（例如它是否会接受可写缓冲区的请求，或者另一个线程是否能改变此数据）。

s (str) [const char *]
将一个 Unicode 对象转换成一个指向字符串的 C 指针。一个指针指向一个已经存在的字符串，这个字符串存储的是传如的字符指针变量。C 字符串是已空结束的。Python 字符串不能包含嵌入的无效的代码点；如果由，一个 ValueError 异常会被引发。Unicode 对象被转化成 'utf-8' 编码的 C 字符串。如果转换失败，一个 UnicodeError 异常被引发。

> **备注**
>
> 这个表达式不接受*bytes-like objects*。如果你想接受文件系统路径并将它们转化成 C 字符串，建议使用 O& 表达式配合*PyUnicode_FSConverter()* 作为 转化函数。

在 3.5 版本发生变更: 以前，当 Python 字符串中遇到了嵌入的 null 代码点会引发 TypeError 。

s* (str or *bytes-like object*) [Py_buffer]
这个表达式既接受 Unicode 对象也接受类字节类型对象。它为由调用者提供的*Py_buffer* 结构赋值。这里结果的 C 字符串可能包含嵌入的 NUL 字节。Unicode 对象通过 'utf-8' 编码转化成 C 字符串。

s# (str, read-only *bytes-like object*) [const char *, *Py_ssize_t*]
像是 s*，区别在于它提供了一个借入的缓冲区。结果存储在两个 C 变量中，第一个是指向 C 字符串的指针，第二个是其长度。该字符串可能包含嵌入的空字节。Unicode 对象会使用 'utf-8' 编码格式转换为 C 字符串。

z (str or None) [const char *]
与 s 类似，但 Python 对象也可能为 None，在这种情况下，C 指针设置为 NULL。

z* (str, *bytes-like object* or None) [Py_buffer]
与 s* 类似，但 Python 对象也可能为 None，在这种情况下，*Py_buffer* 结构的 buf 成员设置为 NULL。

z# (str, read-only *bytes-like object* 或者 None) [const char *, *Py_ssize_t*]
与 s# 类似，但 Python 对象也可能为 None，在这种情况下，C 指针设置为 NULL。

y (**read-only** *bytes-like object*) **[const char *]**
这个格式会将一个类字节对象转换为一个指向借入的 字符串的 C 指针；它不接受 Unicode 对象。字节缓冲区不可包含嵌入的空字节；如果包含这样的内容，将会引发 `ValueError` 异常。exception is raised.

在 3.5 版本发生变更: 以前，当字节缓冲区中遇到了嵌入的 null 字节会引发 `TypeError` 。

y* (*bytes-like object*) **[Py_buffer]**
`s*` 的变式，不接受 Unicode 对象，只接受类字节类型变量。**这是接受二进制数据的推荐方法。**

y# (**read-only** *bytes-like object*) **[const char *, *Py_ssize_t*]**
`s#` 的变式，不接受 Unicode 对象，只接受类字节类型变量。

S (**bytes**) **[PyBytesObject *]**
要求 Python 对象为 `bytes` 对象，不尝试进行任何转换。如果该对象不为 bytes 对象则会引发 `TypeError`。C 变量也可被声明为 *PyObject*。

Y (**bytearray**) **[PyByteArrayObject *]**
要求 Python 对象为 `bytearray` 对象，不尝试进行任何转换。如果该对象不为 `bytearray` 对象则会引发 `TypeError`。C 变量也可被声明为 *PyObject*。

U (**str**) **[PyObject *]**
要求 Python 对象为 Unicode 对象，不尝试进行任何转换。如果该对象不为 Unicode 对象则会引发 `TypeError`。C 变量也可被声明为 *PyObject*。

w* (可读写*bytes-like object*) **[Py_buffer]**
这个表达式接受任何实现可读写缓存区接口的对象。它为调用者提供的*Py_buffer* 结构赋值。缓冲区可能存在嵌入的 null 字节。当缓冲区使用完后调用者需要调用*PyBuffer_Release()*。

es (**str**) **[const char *encoding, char **buffer]**
`s` 的变式，它将编码后的 Unicode 字符存入字符缓冲区。它只处理没有嵌 NUL 字节的已编码数据。

此格式需要两个参数。第一个仅用作输入，并且必须为 `const char*`，它指向一个以 NUL 结束的字符串表示的编码格式名称，或者为 `NULL`，这种情况会使用 `'utf-8'` 编码格式。如果 Python 无法识别指定的编码格式则会引发异常。第二个参数必须为 `char**`；它所引用的指针值将被设为带有参数文本内容的缓冲区。文本将以第一个参数所指定的编码格式进行编码。

PyArg_ParseTuple() 会分配一个足够大小的缓冲区，将编码后的数据拷贝进这个缓冲区并且设置 **buffer* 引用这个新分配的内存空间。调用者有责任在使用后调用*PyMem_Free()* 去释放已经分配的缓冲区。

et (**str**, **bytes** or **bytearray**) **[const char *encoding, char **buffer]**
和 `es` 相同，除了不用重编码传入的字符串对象。相反，它假设传入的参数是编码后的字符串类型。

es# (**str**) **[const char *encoding, char **buffer, *Py_ssize_t* *buffer_length]**
`s#` 的变式，它将已编码的 Unicode 字符存入字符缓冲区。不像 `es` 表达式，它允许传入的数据包含 NUL 字符。

它需要三个参数。第一个仅用作输入，并且必须为 `const char*`，它指向一个以 NUL 结束的字符串表示的编码格式名称，或者为 `NULL`，这种情况会使用 `'utf-8'` 编码格式。如果 Python 无法识别指定的编码格式则会引发异常。第二个参数必须为 `char**`；它所引用的指针值将被设为带有参数文本内容的缓冲区。文本将以第一个参数所指定的编码格式进行编码。第三个参数必须为指向一个整数的指针；被引用的整数将被设为输出缓冲区中的字节数。

有两种操作方式：

如果 **buffer* 指向 `NULL` 指针，则函数将分配所需大小的缓冲区，将编码的数据复制到此缓冲区，并设置 **buffer* 以引用新分配的存储。呼叫者负责调用*PyMem_Free()* 以在使用后释放分配的缓冲区。

如果 **buffer* 指向非 `NULL` 指针（已分配的缓冲区），则*PyArg_ParseTuple()* 将使用此位置作为缓冲区，并将 **buffer_length* 的初始值解释为缓冲区大小。然后，它将将编码的数据复制到缓冲区，并终止它。如果缓冲区不够大，将设置一个 `ValueError`。

在这两个例子中，**buffer_length* 被设置为编码后结尾不为 NUL 的数据的长度。

et# (str, bytes 或 bytearray) [const char *encoding, char **buffer, *Py_ssize_t* *buffer_length]
和 es# 相同，除了不用重编码传入的字符串对象。相反，它假设传入的参数是编码后的字符串类型。

在 3.12 版本发生变更: u, u#, Z 和 Z# 已被移除因为它们只用于旧式的 Py_UNICODE* 表示形式。

数字

这些格式允许将 Python 数字或单个字符表示为 C 数字。需要 int, float 或 complex 的格式还可以使用相应的特殊方法 __index__(), __float__() 或 __complex__() 将 Python 对象转换为所需的类型。

对于有符号整数格式，如果值超出了 C 类型的范围则会引发 OverflowError。对于无符号整数格式，则不会进行取值范围检查 --- 当接受字段太小而无法接受值时将静默地截断最高有效位。

b (int) [unsigned char]
将一个非负的 Python 整数转换为无符号微整数，存储于 C unsigned char 中。

B (int) [unsigned char]
将一个 Python 整数转换为微整数并且不进行溢出检查，存储于 C unsigned char 中。

h (int) [short int]
将 Python 整数转换为 C short int。

H (int) [unsigned short int]
将 Python 整数转换为 C unsigned short int，不进行溢出检查。

i (int) [int]
将 Python 整数转换为 C int。

I (int) [unsigned int]
将 Python 整数转换为 C unsigned int，不进行溢出检查。

l (int) [long int]
将 Python 整数转换为 C long int。

k (int) [unsigned long]
将 Python 整数转换为 C unsigned long，不进行溢出检查。

L (int) [long long]
将 Python 整数转换为 C long long。

K (int) [unsigned long long]
将 Python 整数转换为 C:C:expr:'unsigned long-long'，而不进行溢出检查。

n (int) [*Py_ssize_t*]
将一个 Python 整型转化成一个 C *Py_ssize_t* Python 元大小类型。

c (bytes 或者 bytearray 长度为 1) [char]
将一个 Python 字节类型，如一个长度为 1 的 bytes 或 bytearray 对象，转换为 C char 。

在 3.3 版本发生变更: 允许 bytearray 类型的对象。

C (str 长度为 1) [int]
将一个 Python 字符，如一个长度为 1 的 str 对象，转换为 C int。

f (float) [float]
将 Python 浮点数转换成 C 的:c:expr:*float*

d (float) [double]
将 Python 浮点数转换成 C 的:c:expr:*double*

D (complex) [Py_complex]
将一个 Python 复数类型转化成一个 C *Py_complex* Python 复数类型。

其他对象

o (object) [PyObject *]
将 Python 对象（未经任何转换）存储到一个 C 对象指针中。这样 C 程序就能接收到实际传递的对象。对象的新*strong reference* 不会被创建（即其引用计数不会增加）。存储的指针将不为 NULL。

o! (object) [*typeobject*, PyObject *]
将一个 Python 对象存入一个 C 对象指针。这类似于 o，但是接受两个 C 参数：第一个是 Python 类型对象的地址，第二个是存储对象指针的 C 变量 (类型为 *PyObject**)。如果 Python 对象不具有所要求的类型，则会引发 TypeError。

o& (object) [*converter*, *address*]
通过 *converter* 函数将 Python 对象转换为 C 变量。这需要两个参数：第一个是函数，第二个是 C 变量（任意类型）的地址，转换为 void*。转换器函数依次调用如下：

```
status = converter(object, address);
```

其中 *object* 是待转换的 Python 对象而 *address* 为传给 PyArg_Parse* 函数的 void* 参数。返回的 *status* 应当以 1 代表转换成功而以 0 代表转换失败。当转换失败时，*converter* 函数应当引发异常并让 *address* 的内容保持未修改状态。 如果 *converter* 返回 Py_CLEANUP_SUPPORTED，则如果参数解析最终失败它可能会再次被调用，以使转换器有机会释放已分配的任何内存。在第二次调用中，*object* 形参将为 NULL; *address* 将具有与原始调用相同的值。

转换器的例子: *PyUnicode_FSConverter()* 和*PyUnicode_FSDecoder()*。

在 3.1 版本发生变更: 增加了 Py_CLEANUP_SUPPORTED。

p (bool) [int]
测试传入的值是否为真 (一个布尔判断) 并且将结果转化为相对应的 C true/false 整型值。如果表达式为真置 1，假则置 0。它接受任何合法的 Python 值。参见 truth 获取更多关于 Python 如何测试值为真的信息。

Added in version 3.3.

(items) (tuple) [*matching-items*]
对象必须是 Python 序列，它的长度是 *items* 中格式单元的数量。C 参数必须对应 *items* 中每一个独立的格式单元。序列中的格式单元可能有嵌套。

格式化字符串中还有一些其他的字符具有特殊的涵义。这些可能并不嵌套在圆括号中。它们是：

|
表明在 Python 参数列表中剩下的参数都是可选的。C 变量对应的可选参数需要初始化为默认值——当一个可选参数没有指定时，*PyArg_ParseTuple()* 不能访问相应的 C 变量 (变量集) 的内容。

$
PyArg_ParseTupleAndKeywords() only：表明在 Python 参数列表中剩下的参数都是强制关键字参数。当前，所有强制关键字参数都必须也是可选参数，所以格式化字符串中 | 必须一直在 $ 前面。

Added in version 3.3.

:
格式单元的列表结束标志；冒号后的字符串被用来作为错误消息中的函数名 (*PyArg_ParseTuple()* 函数引发的“关联值”异常)。

;
格式单元的列表结束标志；分号后的字符串被用来作为错误消息取代默认的错误消息。: 和 ; 相互排斥。

请注意提供给调用者的任何 Python 对象引用都是 借入引用；不要释放它们（即不要递减它们的引用计数）!

传递给这些函数的附加参数必须是由格式化字符串确定的变量的地址；这些都是用来存储输入元组的值。有一些情况，如上面的格式单元列表中所描述的，这些参数作为输入值使用；在这种情况下，它们应该匹配指定的相应的格式单元。

为了让转换成功，*arg* 对象必须匹配格式并且格式必须被用尽。当成功时，`PyArg_Parse*` 函数将返回真值，否则将返回假值并引发适当的异常。当 `PyArg_Parse*` 函数由于某个格式单元转换出错而失败时，该格式单元及其后续格式单元对应的地址上的变量都将保持原样。

API 函数

int **PyArg_ParseTuple**(*PyObject* *args, const char *format, ...)

属于稳定 ABI. 解析一个函数的参数，表达式中的参数按参数位置顺序存入局部变量中。成功返回 true；失败返回 false 并且引发相应的异常。

int **PyArg_VaParse**(*PyObject* *args, const char *format, va_list vargs)

属于稳定 ABI. 和*PyArg_ParseTuple()* 相同，然而它接受一个 va_list 类型的参数而不是可变数量的参数集。

int **PyArg_ParseTupleAndKeywords**(*PyObject* *args, *PyObject* *kw, const char *format, char *const *keywords, ...)

属于稳定 ABI. 解析一个将位置参数和关键字参数同时转为局部变量的函数的形参。*keywords* 参数是由以空值结束的 ASCII 和 UTF-8 编码 C 字符串表示的关键字形参名称组成的以 `NULL` 结束的数组。空名称代表仅限位置形参。成功时返回真值；失败时，它将返回假值并引发相应的异常。

> **备注**
>
> *keywords* 形参声明在 C 中为 `char *const*` 而在 C++ 中为 `const char *const*`。这可以通过*PY_CXX_CONST* 宏来重写。

在 3.6 版本发生变更: 添加了*positional-only parameters* 的支持。

在 3.13 版本发生变更: 现在 *keywords* 形参类型在 C 中为 `char *const*` 而在 C++ 中为 `const char *const*`，而不是 `char**`。增加了对非 ASCII 关键字形参名称的支持。

int **PyArg_VaParseTupleAndKeywords**(*PyObject* *args, *PyObject* *kw, const char *format, char *const *keywords, va_list vargs)

属于稳定 ABI. 和*PyArg_ParseTupleAndKeywords()* 相同，然而它接受一个 va_list 类型的参数而不是可变数量的参数集。

int **PyArg_ValidateKeywordArguments**(*PyObject**)

属于稳定 ABI. 确保字典中的关键字参数都是字符串。这个函数只被使用于*PyArg_ParseTupleAndKeywords()* 不被使用的情况下，后者已经不再做这样的检查。

Added in version 3.2.

int **PyArg_Parse**(*PyObject* *args, const char *format, ...)

属于稳定 ABI. 解析一个将单独位置形参转为局部变量的函数的形参。成功时返回真值；失败时，它将返回假值并引发相应的异常。

示例:

```
// 使用 METH_O 调用规范的函数
static PyObject*
my_function(PyObject *module, PyObject *arg)
{
    int value;
    if (!PyArg_Parse(arg, "i:my_function", &value)) {
        return NULL;
    }
    // ... 使用 value ...
}
```

int **PyArg_UnpackTuple**(*PyObject* *args, const char *name, *Py_ssize_t* min, *Py_ssize_t* max, ...)

属于稳定 ABI. 一个更简单的形参提取形式，它不使用格式字符串来指定参数类型。使用此方法来提取其形参的函数应当在函数或方法表中声明为*METH_VARARGS*。包含实际形参的元组应当作为

args 传入；它必须确实是一个元组。该元组的长度必须至少为 *min* 且不超过 *max*；*min* 和 *max* 可能相等。额外的参数必须被传给函数，每个参数应当是一个指向 *PyObject** 变量的指针；它们将以来自 *args* 的值来填充；它们将包含*借入引用*。对应于 *args* 未给出的可选形参的变量不会被填充；它们应当由调用方来初始化。此函数在执行成功时返回真值而在 *args* 不为元组或包含错误数量的元素时返回假值；如果执行失败则还将设置一个异常。

这是一个使用该函数的示例，取自 `_weakref` 弱引用辅助模块的源代码:

```
static PyObject *
weakref_ref(PyObject *self, PyObject *args)
{
    PyObject *object;
    PyObject *callback = NULL;
    PyObject *result = NULL;

    if (PyArg_UnpackTuple(args, "ref", 1, 2, &object, &callback)) {
        result = PyWeakref_NewRef(object, callback);
    }
    return result;
}
```

这个例子中调用*PyArg_UnpackTuple()* 完全等价于调用*PyArg_ParseTuple()*:

```
PyArg_ParseTuple(args, "O|O:ref", &object, &callback)
```

PY_CXX_CONST

要插入到*PyArg_ParseTupleAndKeywords()* 和*PyArg_VaParseTupleAndKeywords()* 的 *keywords* 形参声明中位于 `char *const*` 之前的值，如果有的话。默认在 C 中为空而在 C++ 中为 `const` (`const char *const*`)。如需重写，可在包括 `Python.h` 之前将其定义为想要的值。

Added in version 3.13.

6.6.2 创建变量

PyObject ***Py_BuildValue**(const char *format, ...)

返回值：新的引用。属于稳定 ABI. 基于类似 `PyArg_Parse*` 函数族所接受内容的格式字符串和一个值序列来创建一个新值。返回该值或在发生错误的情况下返回 `NULL`；如果返回 `NULL` 则将引发一个异常。

Py_BuildValue() 并不一直创建一个元组。只有当它的格式化字符串包含两个或更多的格式单元才会创建一个元组。如果格式化字符串是空，它返回 `None`；如果它包含一个格式单元，它返回由格式单元描述的的任一对象。用圆括号包裹格式化字符串可以强制它返回一个大小为 0 或者 1 的元组。

当内存缓存区的数据以参数形式传递用来构建对象时，如 `s` 和 `s#` 格式单元，会拷贝需要的数据。调用者提供的缓冲区从来都不会被由*Py_BuildValue()* 创建的对象来引用。换句话说，如果你的代码调用 `malloc()` 并且将分配的内存空间传递给*Py_BuildValue()*，你的代码就有责任在*Py_BuildValue()* 返回时调用 `free()` 。

在下面的描述中，双引号的表达式使格式单元；圆括号 () 内的是格式单元将要返回的 Python 对象类型；方括号 [] 内的是传递的 C 变量 (变量集) 的类型。

字符例如空格，制表符，冒号和逗号在格式化字符串中会被忽略 (但是不包括格式单元，如 `s#`)。这可以使很长的格式化字符串具有更好的可读性。

s (str 或 None) [const char *]
使用 `'utf-8'` 编码将空终止的 C 字符串转换为 Python `str` 对象。如果 C 字符串指针为 `NULL`，则使用 `None`。

s# (str 或 None) [const char *, *Py_ssize_t*]
使用 `'utf-8'` 编码将 C 字符串及其长度转换为 Python `str` 对象。如果 C 字符串指针为 `NULL`，则长度将被忽略，并返回 `None`。

y (bytes) [const char *]
这将 C 字符串转换为 Python `bytes` 对象。如果 C 字符串指针为 `NULL`，则返回 `None`。

y# (bytes) [const char *, *Py_ssize_t*]
这会将 C 字符串及其长度转换为一个 Python 对象。如果该 C 字符串指针为 `NULL`，则返回 `None`。

z (str or None) [const char *]
和 `s` 一样。

z# (str 或 None) [const char *, *Py_ssize_t*]
和 `s#` 一样。

u (str) [const wchar_t *]
将空终止的 `wchar_t` 的 Unicode （UTF-16 或 UCS-4）数据缓冲区转换为 Python Unicode 对象。如果 Unicode 缓冲区指针为 `NULL`，则返回 `None`。

u# (str) [const wchar_t *, *Py_ssize_t*]
将 Unicode（UTF-16 或 UCS-4）数据缓冲区及其长度转换为 Python Unicode 对象。如果 Unicode 缓冲区指针为 `NULL`，则长度将被忽略，并返回 `None`。

U (str 或 None) [const char *]
和 `s` 一样。

U# (str 或 None) [const char *, *Py_ssize_t*]
和 `s#` 一样。

i (int) [int]
将一个基本 C `int` 转换为 Python 整数对象。

b (int) [char]
将一个基本 C `char` 转换为 Python 整数对象。

h (int) [short int]
将一个基本 C `short int` 转换为 Python 整数对象。

l (int) [long int]
将一个 C `long int` 转换为 Python 整数对象。

B (int) [unsigned char]
将一个 C `unsigned char` 转换为 Python 整数对象。

H (int) [unsigned short int]
将一个 C `unsigned short int` 转换为 Python 整数对象。

I (int) [unsigned int]
将一个 C `unsigned int` 转换为 Python 整数对象。

k (int) [unsigned long]
将一个 C `unsigned long` 转换为 Python 整数对象。

L (int) [long long]
将一个 C `long long` 转换为 Python 整数对象。

K (int) [unsigned long long]
将一个 C `unsigned long long` 转换为 Python 整数对象。

n (int) [*Py_ssize_t*]
将一个 C *Py_ssize_t* 类型转化为 Python 整型。

c (bytes 长度为 1) [char]
将一个代表单个字节的 C `int` 转换为长度为 1 的 Python `bytes` 对象。

C (str 长度为 1) [int]
将一个代表单个字符的 C `int` 转换为长度为 1 的 Python `str` 对象。

d (float) [double]
将 C `double` 转换成 Python 浮点数

f (float) [float]
将 C `float` 转换成 Python 浮点数

D (complex) [Py_complex *]
将一个 C *Py_complex* 类型的结构转化为 Python 复数类型。

O (object) [PyObject *]
原封不动地传递一个 Python 对象，但为其创建一个新的*strong reference* (即其引用计数加一)。如果传入的对象是一个 `NULL` 指针，则会假定这是因为产生该参数的调用发现了错误并设置了异常。因此，*Py_BuildValue()* 将返回 `NULL` 但不会引发异常。如果尚未引发异常，则会设置 `SystemError`。

S (object) [PyObject *]
和 `O` 相同。

N (object) [PyObject *]
与 `O` 相同，但它不会创建新的*strong reference*。如果对象是通过调用参数列表中的对象构造器来创建的则该方法将很有用处。

O& (object) [*converter*, *anything*]
通过 *converter* 函数将 *anything* 转换为 Python 对象。该函数在调用时附带 *anything* (它应当兼容 `void*`) 作为其参数并且应返回一个" 新的" Python 对象，或者如果发生错误则返回 `NULL`。

(items) (tuple) [*matching-items*]
将一个 C 变量序列转换成 Python 元组并保持相同的元素数量。

[items] (list) [相关的元素]
将一个 C 变量序列转换成 Python 列表并保持相同的元素数量。

{items} (dict) [相关的元素]
将一个 C 变量序列转换成 Python 字典。每一对连续的 C 变量对作为一个元素插入字典中，分别作为关键字和值。

如果格式字符串中出现错误，则设置 `SystemError` 异常并返回 `NULL`。

PyObject ***Py_VaBuildValue**(const char *format, va_list vargs)

返回值：新的引用。属于稳定 ABI. 和*Py_BuildValue()* 相同，然而它接受一个 va_list 类型的参数而不是可变数量的参数集。

6.7 字符串转换与格式化

用于数字转换和格式化字符串输出的函数

int **PyOS_snprintf**(char *str, size_t size, const char *format, ...)

属于稳定 ABI. 根据格式字符串 *format* 和额外参数，输出不超过 *size* 个字节到 *str*。参见 Unix 手册页面 *snprintf(3)*。

int **PyOS_vsnprintf**(char *str, size_t size, const char *format, va_list va)

属于稳定 ABI. 根据格式字符串 *format* 和变量参数列表 *va*，输出不超过 *size* 个字节到 *str*。参见 Unix 手册页面 *vsnprintf(3)*。

PyOS_snprintf() 和*PyOS_vsnprintf()* 包装 C 标准库函数 `snprintf()` 和 `vsnprintf()` 。它们的目的是保证在极端情况下的一致行为，而标准 C 的函数则不然。

此包装器会确保 `str[size-1]` 在返回时始终为 `'\0'`。它们从不写入超过 *size* 字节 (包括末尾的 `'\0'`) 到 str。两个函数都要求 `str != NULL`, `size > 0`, `format != NULL` 且 `size < INT_MAX`。请注意这意味着不存在可确定所需缓冲区大小的 C99 `n = snprintf(NULL, 0, ...)` 的等价物。

这些函数的返回值（*rv* ）应按照以下规则被解释：

- 当 `0 <= rv < size` 时，输出转换即成功并将 *rv* 个字符写入到 *str* (不包括末尾 `str[rv]` 位置的 `'\0'` 字节)。
- 当 `rv >= size` 时，输出转换会被截断并且需要一个具有 `rv + 1` 字节的缓冲区才能成功执行。在此情况下 `str[size-1]` 为 `'\0'`。

- 当 `rv < 0` 时，”会发生不好的事情。”在此情况下 `str[size-1]` 也为 `'\0'`，但 *str* 的其余部分是未定义的。错误的确切原因取决于底层平台。

以下函数提供与语言环境无关的字符串到数字转换。

unsigned long **PyOS_strtoul**(const char *str, char **ptr, int base)

属于稳定 ABI. 根据给定的 `base` 将 `str` 中字符串的初始部分转换为 `unsigned long` 值，该值必须在 `2` 至 `36` 的开区间内，或者为特殊值 `0`。

空白前缀和字符大小写将被忽略。如果 `base` 为零则会查找 `0b`、`0o` 或 `0x` 前缀以确定基数。如果没有则默认基数为 `10`。基数必须为 0 或在 2 和 36 之间（包括边界值）。如果 `ptr` 不为 `NULL` 则它将包含一个指向扫描结束位置的指针。

如果转换后的值不在对应返回类型的取值范围之内，则会发生取值范围错误(`errno` 被设为 `ERANGE`)并返回 `ULONG_MAX`。如果无法执行转换，则返回 `0`。

另请参阅 Unix 指南页 *strtoul(3)*。

Added in version 3.2.

long **PyOS_strtol**(const char *str, char **ptr, int base)

属于稳定 ABI. 根据给定的 `base` 将 `str` 中字符串的初始部分转换为 `long` 值，该值必须在 `2` 至 `36` 的开区间内，或者为特殊值 `0`。

类似于*PyOS_strtoul()*，但在溢出时将返回一个 `long` 值而不是 `LONG_MAX`。

另请参阅 Unix 指南页 *strtol(3)*。

Added in version 3.2.

double **PyOS_string_to_double**(const char *s, char **endptr, *PyObject* *overflow_exception)

属于稳定 ABI. 将字符串 `s` 转换为 `double` 类型，失败时会引发 Python 异常。接受的字符串集合对应于可被 Python 的 `float()` 构造器所接受的字符集集合，除了 `s` 必须没有前导或尾随空格。转换必须独立于当前的语言区域。

如果 `endptr` 是 `NULL`，转换整个字符串。引发 `ValueError` 并且返回 `-1.0` 如果字符串不是浮点数的有效的表达方式。

如果 `endptr` 不是 `NULL`，尽可能多的转换字符串并将 `*endptr` 设置为指向第一个未转换的字符。如果字符串的初始段不是浮点数的有效的表达方式，将 `*endptr` 设置为指向字符串的开头，引发 ValueError 异常，并且返回 `-1.0` 。

如果 `s` 表示一个太大而不能存储在一个浮点数中的值（比方说，`"1e500"` 在许多平台上是一个字符串）然后如果 `overflow_exception` 是 `NULL` 返回 `Py_HUGE_VAL`（用适当的符号）并且不设置任何异常。在其他方面，`overflow_exception` 必须指向一个 Python 异常对象；引发异常并返回 `-1.0` 。在这两种情况下，设置 `*endptr` 指向转换值之后的第一个字符。

如果在转换期间发生任何其他错误（比如一个内存不足的错误），设置适当的 Python 异常并且返回 `-1.0` 。

Added in version 3.1.

char ***PyOS_double_to_string**(double val, char format_code, int precision, int flags, int *ptype)

属于稳定 ABI. 将 `double` *val* 转换为一个使用给定的 *format_code*, *precision* 和 *flags* 的字符串。

*格式码*必须是以下其中之一，`'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` 或者 `'r'`。对于 `'r'`，提供的 *精度*必须是 0。`'r'` 格式码指定了标准函数 `repr()` 格式。

flags 可以为零或者其他值 `Py_DTSF_SIGN`, `Py_DTSF_ADD_DOT_0` 或 `Py_DTSF_ALT` 或其组合：

- `Py_DTSF_SIGN` 表示总是在返回的字符串前附加一个符号字符，即使 *val* 为非负数。
- `Py_DTSF_ADD_DOT_0` 表示确保返回的字符串看起来不像是一个整数。
- `Py_DTSF_ALT` 表示应用”替代的”格式化规则。相关细节请参阅*PyOS_snprintf()* `'#'` 定义文档。

如果 *ptype* 不为 NULL，则它指向的值将被设为 Py_DTST_FINITE, Py_DTST_INFINITE 或 Py_DTST_NAN 中的一个，分别表示 *val* 是一个有限数字、无限数字或非数字。

返回值是一个指向包含转换后字符串的 *buffer* 的指针，如果转换失败则为 NULL。调用方要负责调用 *PyMem_Free()* 来释放返回的字符串。

Added in version 3.1.

int **PyOS_stricmp**(const char *s1, const char *s2)

不区分大小写的字符串比较。除了忽略大小写之外，该函数的工作方式与 strcmp() 相同。

int **PyOS_strnicmp**(const char *s1, const char *s2, *Py_ssize_t* size)

不区分大小写的字符串比较。除了忽略大小写之外，该函数的工作方式与 strncmp() 相同。

6.8 PyHash API

另请参阅 *PyTypeObject.tp_hash* 成员和 numeric-hash。

type **Py_hash_t**

哈希值类型：有符号整数。

Added in version 3.2.

type **Py_uhash_t**

哈希值类型：无符号整数。

Added in version 3.2.

PyHASH_MODULUS

梅森素数 P = 2**n -1，用于数字哈希方案。

Added in version 3.13.

PyHASH_BITS

P 在 *PyHASH_MODULUS* 中的 n 次幂。

Added in version 3.13.

PyHASH_MULTIPLIER

质因数被用于字符串和多种其他哈希算法中。

Added in version 3.13.

PyHASH_INF

针对正无穷大返回的哈希值。

Added in version 3.13.

PyHASH_IMAG

用于复数虚部的乘数。

Added in version 3.13.

type **PyHash_FuncDef**

PyHash_GetFuncDef() 使用的哈希函数定义。

const char ***name**

哈希函数名称（UTF-8 编码的字符串）。

const int **hash_bits**

以比特位表示的哈希值内部大小。

const int **seed_bits**

以比特位表示的输入种子值大小。

Added in version 3.4.

PyHash_FuncDef ***PyHash_GetFuncDef**(void)

获取哈希函数定义。

> **参见**
>
> **PEP 456** " 安全且可互换的哈希算法"。

Added in version 3.4.

Py_hash_t **Py_HashPointer**(const void *ptr)

对指针值执行哈希运算：将指针值作为整数来处理（在内部将其转换为 `uintptr_t` 类型）。指针不会被撤销引用。

此函数不会失败：它不可能返回 `-1`。

Added in version 3.13.

Py_hash_t **PyObject_GenericHash**(*PyObject* *obj)

将会被放入类型对象的 `tp_hash` 槽位的泛型哈希函数。其结果值仅取决于对象的标识号。

在 CPython 中，它等价于*Py_HashPointer()*。

Added in version 3.13.

6.9 反射

PyObject ***PyEval_GetBuiltins**(void)

返回值：借入的引用。属于稳定 ABI. 自 3.13 版本弃用: 使用*PyEval_GetFrameBuiltins()* 代替。

返回当前执行帧中内置函数的字典，如果当前没有帧正在执行，则返回线程状态的解释器。

PyObject ***PyEval_GetLocals**(void)

返回值：借入的引用。属于稳定 ABI. 自 3.13 版本弃用: 使用*PyEval_GetFrameLocals()* 来获取与在 Python 代码中调用 `locals()` 相同的行为，或者是在*PyEval_GetFrame()* 的结果上调用*PyFrame_GetLocals()* 来访问当前执行帧的 `f_locals` 属性。

返回一个提供对当前执行帧中局部变量访问的映射，或者如果没有当前执行帧则返回 `NULL`。

请参考 `locals()` 了解在不同作用域下返回的映射的详情。

由于此函数会返回一个*borrowed reference*，为已优化作用域 返回的字典将缓存在帧对象上因此会在帧对象存活期间保持存活。不同于*PyEval_GetFrameLocals()* 和 `locals()`，在相同帧中对该函数的后续调用将会更新已缓存字典的内容以反映局部变量状态的变化而不是返回一个新的快照。

在 3.13 版本发生变更: 作为 **PEP 667** 的一部分，*PyFrame_GetLocals()*，`locals()` 和 `FrameType.f_locals` 将不再使用共享的缓存字典。请参阅 有什么新变化条目了解详情。

PyObject ***PyEval_GetGlobals**(void)

返回值：借入的引用。属于稳定 ABI. 自 3.13 版本弃用: 使用*PyEval_GetFrameGlobals()* 代替。

返回当前执行帧中全局变量的字典，如果没有当前执行的帧则返回 `NULL`。

PyFrameObject ***PyEval_GetFrame**(void)

返回值：借入的引用。属于稳定 ABI. 返回当前线程状态的帧，如果没有当前执行的帧则返回 `NULL`。

另请参阅*PyThreadState_GetFrame()*。

PyObject ***PyEval_GetFrameBuiltins**(void)

返回值：新的引用。属于稳定 ABI 自 *3.13* 版起. 返回当前执行帧中内置函数的字典，如果当前没有帧正在执行，则返回线程状态的解释器。

Added in version 3.13.

PyObject ***PyEval_GetFrameLocals**(void)

返回值：新的引用。属于稳定 ABI 自 *3.13* 版起. 返回当前执行帧中局部变量的字典，或者如果没有当前执行帧则返回 NULL。等价于在 Python 代码中调用 `locals()`。

要访问当前帧上的 `f_locals` 而不会在已优化作用域中生成独立的快照，可以在*PyEval_GetFrame()* 的结果上调用*PyFrame_GetLocals()*。

Added in version 3.13.

PyObject ***PyEval_GetFrameGlobals**(void)

返回值：新的引用。属于稳定 ABI 自 *3.13* 版起. 返回当前执行帧中局部变量的字典，或者如果没有当前执行帧则返回 NULL。等价于在 Python 代码中调用 `globals()`。

Added in version 3.13.

const char ***PyEval_GetFuncName**(*PyObject* *func)

属于稳定 ABI. 如果 *func* 是函数、类或实例对象，则返回它的名称，否则返回 *func* 的类型的名称。

const char ***PyEval_GetFuncDesc**(*PyObject* *func)

属于稳定 ABI. 根据 *func* 的类型返回描述字符串。返回值包括函数和方法的"()", " constructor", " instance" 和" object"。与*PyEval_GetFuncName()* 的结果连接，结果将是 *func* 的描述。

6.10 编解码器注册与支持功能

int **PyCodec_Register**(*PyObject* *search_function)

属于稳定 ABI. 注册一个新的编解码器搜索函数。

作为其附带影响，如果 `encodings` 包尚未加载，则会尝试加载它，以确保它在搜索函数列表中始终排在第一位。

int **PyCodec_Unregister**(*PyObject* *search_function)

属于稳定 ABI 自 *3.10* 版起. 注销一个编解码器搜索函数并清空注册表缓存。如果指定搜索函数未被注册，则不做任何操作。成功时返回 0。出错时引发一个异常并返回 -1。

Added in version 3.10.

int **PyCodec_KnownEncoding**(const char *encoding)

属于稳定 ABI. 根据注册的给定 *encoding* 的编解码器是否已存在而返回 `1` 或 `0`。此函数总能成功。

PyObject ***PyCodec_Encode**(*PyObject* *object, const char *encoding, const char *errors)

返回值：新的引用。属于稳定 ABI. 泛型编解码器基本编码 API。

object 使用由 *errors* 所定义的错误处理方法传递给定 *encoding* 的编码器函数。*errors* 可以为 NULL 表示使用为编码器所定义的默认方法。如果找不到编码器则会引发 `LookupError`。

PyObject ***PyCodec_Decode**(*PyObject* *object, const char *encoding, const char *errors)

返回值：新的引用。属于稳定 ABI. 泛型编解码器基本解码 API。

object 使用由 *errors* 所定义的错误处理方法传递给定 *encoding* 的解码器函数。*errors* 可以为 NULL 表示使用为编解码器所定义的默认方法。如果找不到编解码器则会引发 `LookupError`。

6.10.1 Codec 查找 API

在下列函数中，*encoding* 字符串会被查找并转换为小写字母形式，这使得通过此机制查找编码格式实际上对大小写不敏感。如果未找到任何编解码器，则将设置 `KeyError` 并返回 NULL。

PyObject ***PyCodec_Encoder**(const char *encoding)

返回值：新的引用。属于稳定 ABI. 为给定的 *encoding* 获取一个编码器函数。

PyObject ***PyCodec_Decoder**(const char *encoding)

返回值：新的引用。属于稳定 ABI. 为给定的 *encoding* 获取一个解码器函数。

PyObject ***PyCodec_IncrementalEncoder**(const char *encoding, const char *errors)

返回值：新的引用。属于稳定 ABI. 为给定的 *encoding* 获取一个 `IncrementalEncoder` 对象。

PyObject ***PyCodec_IncrementalDecoder**(const char *encoding, const char *errors)

返回值：新的引用。属于稳定 ABI. 为给定的 *encoding* 获取一个 `IncrementalDecoder` 对象。

PyObject ***PyCodec_StreamReader**(const char *encoding, *PyObject* *stream, const char *errors)

返回值：新的引用。属于稳定 ABI. 为给定的 *encoding* 获取一个 `StreamReader` 工厂函数。

PyObject ***PyCodec_StreamWriter**(const char *encoding, *PyObject* *stream, const char *errors)

返回值：新的引用。属于稳定 ABI. 为给定的 *encoding* 获取一个 `StreamWriter` 工厂函数。

6.10.2 用于 Unicode 编码错误处理程序的注册表 API

int **PyCodec_RegisterError**(const char *name, *PyObject* *error)

属于稳定 ABI. 在给定的 *name* 之下注册错误处理回调函数 *error*。该回调函数将在一个编解码器遇到无法编码的字符/无法解码的字节数据并且 *name* 被指定为 encode/decode 函数调用的 error 形参时由该编解码器来调用。

该回调函数会接受一个 `UnicodeEncodeError`, `UnicodeDecodeError` 或 `UnicodeTranslateError` 的实例作为单独参数，其中包含关于有问题字符或字节序列及其在原始序列的偏移量信息（请参阅*Unicode 异常对象* 了解提取此信息的函数详情）。该回调函数必须引发给定的异常，或者返回一个包含有问题序列及相应替换序列的二元组，以及一个表示偏移量的整数，该整数指明应在什么位置上恢复编码/解码操作。

成功则返回 `0`，失败则返回 `-1`。

PyObject ***PyCodec_LookupError**(const char *name)

返回值：新的引用。属于稳定 ABI. 查找在 *name* 之下注册的错误处理回调函数。作为特例还可以传入 `NULL`，在此情况下将返回针对”strict” 的错误处理回调函数。

PyObject ***PyCodec_StrictErrors**(*PyObject* *exc)

返回值：恒为 *NULL*。属于稳定 ABI. 引发 *exc* 作为异常。

PyObject ***PyCodec_IgnoreErrors**(*PyObject* *exc)

返回值：新的引用。属于稳定 ABI. 忽略 unicode 错误，跳过错误的输入。

PyObject ***PyCodec_ReplaceErrors**(*PyObject* *exc)

返回值：新的引用。属于稳定 ABI. 使用 `?` 或 `U+FFFD` 替换 unicode 编码错误。

PyObject ***PyCodec_XMLCharRefReplaceErrors**(*PyObject* *exc)

返回值：新的引用。属于稳定 ABI. 使用 XML 字符引用替换 unicode 编码错误。

PyObject ***PyCodec_BackslashReplaceErrors**(*PyObject* *exc)

返回值：新的引用。属于稳定 ABI. 使用反斜杠转义符 (`\x`, `\u` 和 `\U`) 替换 unicode 编码错误。

PyObject ***PyCodec_NameReplaceErrors**(*PyObject* *exc)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 使用 `\N{...}` 转义符替换 unicode 编码错误。

Added in version 3.5.

6.11 PyTime C API

Added in version 3.13.

时钟 C API 提供对系统时钟的访问。它类似于 Python `time` 模块。

有关与 `datetime` 模块相关的 C API，请参阅*DateTime 对象*。

6.11.1 类型

type **PyTime_t**

以纳秒为单位的时间戳或持续时间，表示为带符号的 64 位整数。

时间戳的参考点取决于所使用的时钟。例如，*PyTime_Time()* 返回相对于 UNIX 纪元的时间戳。

支持的范围约为 [-292.3 年；+292.3 年]。以 Unix 纪元（1970 年 1 月 1 日）为参考，支持的日期范围约为 [1677-09-21；2262-04-11]。确切的限制以常数形式公开：

PyTime_t **PyTime_MIN**

PyTime_t 的最小值。

PyTime_t **PyTime_MAX**

PyTime_t 的最大值。

6.11.2 时钟函数

以下函数采用指向 *PyTime_t* 的指针，并将其设置为特定时钟的值。每个时钟的详细信息在相应的 Python 函数的文档中给出。

成功时函数返回 “0“，失败时返回 “-1“（设置一个异常）。

在整数溢出时，他们设置*PyExc_OverflowError* 异常，并将 `*result` 设置为 `[PyTime_MIN; PyTime_MAX]` 范围内的值。（在当前系统上，整数溢出可能是由于系统时间配置错误引起的。）

与任何其他 C API 一样（除非另有说明），必须使用持有的*GIL* 来调用函数。

int **PyTime_Monotonic**(*PyTime_t* *result)

读取单调时间。有关该时钟的重要详细信息，请参阅 `time.monotonic()`。

int **PyTime_PerfCounter**(*PyTime_t* *result)

读取性能计数器。有关该时钟的重要详细信息，请参阅 `time.perf_counter()`。

int **PyTime_Time**(*PyTime_t* *result)

读取 “Wall Clock” 时间。有关该时钟的重要详细信息，请参阅 `time.time`()`。

6.11.3 原始时钟函数

与时钟函数类似，但不设置错误异常，也不要求调用者持有 GIL。

成功时，函数返回 `0`。

失败时，它们将 `*result` 设置为 `0` 并返回 `-1`，不设置异常。要了解错误原因，请获取 GIL 并调用常规 (非 `Raw`) 函数。请注意，常规函数可能会在 `Raw` 函数失败后成功。

int **PyTime_MonotonicRaw**(*PyTime_t* *result)

与*PyTime_Monotonic()* 类似，但在错误时不设置异常，并且不需要持有 GIL。

int **PyTime_PerfCounterRaw**(*PyTime_t* *result)

与*PyTime_PerfCounter()* 类似，但在错误时不设置异常，并且不需要持有 GIL。

int **PyTime_TimeRaw**(*PyTime_t* *result)

与*PyTime_Time()* 类似，但在错误时不设置异常，并且不需要持有 GIL。

6.11.4 转换函数

double **PyTime_AsSecondsDouble**(*PyTime_t* t)

将时间戳转换为 C `double` 形式的秒数。

该函数不会失败，但请注意 `double` 对于大值的精度有限。

6.12 对 Perf Maps 的支持

在受支持的平台上（在撰写本文档时，只有 Linux），运行时可以利用 *perf map* 文件来使得 Python 函数对于外部性能分析工具可见（例如 perf 等）。正在运行的进行可以在 `/tmp` 目录中创建一个文件，其中包含可将部分可执行代码映射到特定名称的条目。本接口的描述参见 Linux Perf 工具文档。

在 Python 中，这些辅助 API 可供依赖于动态生成机器码的库和特性使用。

请注意这些 API 并不要求持有全局解释器锁（GIL）。

int **PyUnstable_PerfMapState_Init**(void)

> 这是不稳定 *API*。它可在次发布版中不经警告地改变。
>
> 打开 `/tmp/perf-$pid.map` 文件，除非它已经被打开，并创建一个锁来确保线程安全地写入该文件（如果写入是通过*PyUnstable_WritePerfMapEntry()* 执行的）。通常，没有必要显式地调用此函数；只需使用*PyUnstable_WritePerfMapEntry()* 这样它将在第一次调用时初始化状态。
>
> 成功时返回 `0`，创建/打开 perf map 文件失败时返回 `-1`，或者创建锁失败时返回 `-2`。可检查 `errno` 获取有关失败原因的更多信息。

int **PyUnstable_WritePerfMapEntry**(const void *code_addr, unsigned int code_size, const char *entry_name)

> 这是不稳定 *API*。它可在次发布版中不经警告地改变。
>
> 向 `/tmp/perf-$pid.map` 文件写入一个单独条目。此函数是线程安全的。下面显示了一个示例条目：
>
> ```
> # 地址            大小   名称
> 7f3529fcf759 b     py::bar:/run/t.py
> ```
>
> 将在写入条目之前调用*PyUnstable_PerfMapState_Init()*，如果 perf map 文件尚未打开。成功时返回 `0`，或者在失败时返回与*PyUnstable_PerfMapState_Init()* 相同的错误代码。

void **PyUnstable_PerfMapState_Fini**(void)

> 这是不稳定 *API*。它可在次发布版中不经警告地改变。
>
> 关闭*PyUnstable_PerfMapState_Init()* 所打开的 perf map 文件。此函数会在解释器关闭期间由运行时本身调用。通常，应该没有理由显式地调用此函数，除了处理特殊场景例如分叉操作。

CHAPTER 7

抽象对象层

本章中的函数与 Python 对象交互，无论其类型，或具有广泛类的对象类型（例如，所有数值类型，或所有序列类型）。当使用对象类型并不适用时，他们会产生一个 Python 异常。

这些函数是不可能用于未正确初始化的对象的，如一个列表对象被*PyList_New()* 创建，但其中的项目没有被设置为一些非 NULL 的值。

7.1 对象协议

PyObject ***Py_GetConstant**(unsigned int constant_id)

属于稳定 ABI 自 *3.13* 版起. 获取一个指向常量的*strong reference*。

如果 *constant_id* 无效则设置一个异常并返回 NULL。

constant_id 必须是下列常量标识符之一：

常量标识符	值	返回的对象
Py_CONSTANT_NONE	0	None
Py_CONSTANT_FALSE	1	False
Py_CONSTANT_TRUE	2	True
Py_CONSTANT_ELLIPSIS	3	Ellipsis
Py_CONSTANT_NOT_IMPLEMENTI	4	NotImplemented
Py_CONSTANT_ZERO	5	0
Py_CONSTANT_ONE	6	1
Py_CONSTANT_EMPTY_STR	7	''
Py_CONSTANT_EMPTY_BYTES	8	b''
Py_CONSTANT_EMPTY_TUPLE	9	()

仅对无法使用常量标识符的项目才会给出数字值。

Added in version 3.13.

在 CPython 中，所有这些常量都属于*immortal* 对象。

PyObject ***Py_GetConstantBorrowed**(unsigned int constant_id)

属于稳定 ABI 自 *3.13* 版起. 类似于`Py_GetConstant()`，但会返回一个*borrowed reference*。

此函数的主要目的是用于向下兼容：对于新代码推荐使用`Py_GetConstant()`。

该引用是从解释器借入的，并将保持可用直到解释器最终化。

Added in version 3.13.

PyObject ***Py_NotImplemented**

`NotImplemented` 单例，用于标记某个操作没有针对给定类型组合的实现。

Py_RETURN_NOTIMPLEMENTED

正确处理从 C 函数内部返回`Py_NotImplemented` 的问题（即新建一个指向 `NotImplemented` 的*strong reference* 并返回它）。

Py_PRINT_RAW

要与多个打印对象的函数（如`PyObject_Print()` 和`PyFile_WriteObject()`）一起使用的旗标。如果传入，这些函数应当使用对象的 `str()` 而不是 `repr()`。

int **PyObject_Print**(*PyObject* *o, FILE *fp, int flags)

打印对象 *o* 到文件 *fp*。出错时返回 `-1`。flags 参数被用于启用特定的打印选项。目前唯一支持的选项是`Py_PRINT_RAW`；如果给出该选项，则将写入对象的 `str()` 而不是 `repr()`。

int **PyObject_HasAttrWithError**(*PyObject* *o, *PyObject* *attr_name)

属于稳定 ABI 自 *3.13* 版起. 如果 *o* 具有属性 *attr_name* 则返回 `1`，否则返回 `0`。这相当于 Python 表达式 `hasattr(o, attr_name)`。当失败时，将返回 `-1`。

Added in version 3.13.

int **PyObject_HasAttrStringWithError**(*PyObject* *o, const char *attr_name)

属于稳定 ABI 自 *3.13* 版起. 这与*PyObject_HasAttrWithError()* 相同，但 *attr_name* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

Added in version 3.13.

int **PyObject_HasAttr**(*PyObject* *o, *PyObject* *attr_name)

属于稳定 ABI. 如果 *o* 具有属性 *attr_name* 则返回 `1`，否则返回 `0`。此函数总是会成功执行。

> **备注**
>
> 当其调用 `__getattr__()` 和 `__getattribute__()` 方法时发生的异常将不会被传播，而是交给 `sys.unraisablehook()`。想要进行适当的错误处理，请改用*PyObject_HasAttrWithError()*，*PyObject_GetOptionalAttr()* 或*PyObject_GetAttr()*。

int **PyObject_HasAttrString**(*PyObject* *o, const char *attr_name)

属于稳定 ABI. 这与*PyObject_HasAttr()* 相同，但 *attr_name* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

> **备注**
>
> 在其调用 `__getattr__()` 和 `__getattribute__()` 方法或创建临时 `str` 对象期间发生的异常将被静默地忽略。想要进行适当的错误处理，请改用*PyObject_HasAttrStringWithError()*，*PyObject_GetOptionalAttrString()* 或*PyObject_GetAttrString()*。

PyObject ***PyObject_GetAttr**(*PyObject* *o, *PyObject* *attr_name)

返回值：新的引用。属于稳定 ABI. 从对象 *o* 中读取名为 *attr_name* 的属性。成功返回属性值，失败则返回 `NULL`。这相当于 Python 表达式 `o.attr_name`。

如果缺少属性不应被视为执行失败，你可以改用*PyObject_GetOptionalAttr()*。

PyObject ***PyObject_GetAttrString**(*PyObject* *o, const char *attr_name)

返回值：新的引用。属于稳定 ABI. 这与*PyObject_GetAttr()* 相同，但 *attr_name* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

如果缺少属性不应被视为执行失败，你可以改用*PyObject_GetOptionalAttrString()*。

int **PyObject_GetOptionalAttr**(*PyObject* *obj, *PyObject* *attr_name, *PyObject* **result);

属于稳定 ABI 自 *3.13* 版起. *PyObject_GetAttr()* 的变化形式，它在未找到键时不会引发 `AttributeError`。

如果找到该属性，则返回 `1` 并将 **result* 设为指向该属性的新*strong reference*。如果未找到该属性，则返回 `0` 并将 **result* 设为 `NULL`；`AttributeError` 会被静默。如果引发了 `AttributeError` 以外的错误，则返回 `-1` 并将 **result* 设为 `NULL`。

Added in version 3.13.

int **PyObject_GetOptionalAttrString**(*PyObject* *obj, const char *attr_name, *PyObject* **result);

属于稳定 ABI 自 *3.13* 版起. 这与*PyObject_GetOptionalAttr()* 相同，但 *attr_name* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

Added in version 3.13.

PyObject ***PyObject_GenericGetAttr**(*PyObject* *o, *PyObject* *name)

返回值：新的引用。属于稳定 ABI. 通用的属性获取函数，用于放入类型对象的 `tp_getattro` 槽中。它在类的字典中（位于对象的 MRO 中）查找某个描述符，并在对象的 `__dict__` 中查找某个属性。正如 descriptors 所述，数据描述符优先于实例属性，而非数据描述符则不优先。失败则会触发 `AttributeError` 。

int **PyObject_SetAttr**(*PyObject* *o, *PyObject* *attr_name, *PyObject* *v)

属于稳定 ABI. 将对象 *o* 中名为 *attr_name* 的属性值设为 *v* 。失败时引发异常并返回 `-1`；成功时返回 `0` 。这相当于 Python 语句 `o.attr_name = v`。

如果 *v* 为 `NULL`，该属性将被删除。此行为已被弃用而应改用*PyObject_DelAttr()*，但目前还没有移除它的计划。

int **PyObject_SetAttrString**(*PyObject* *o, const char *attr_name, *PyObject* *v)

属于稳定 ABI. 这与*PyObject_SetAttr()* 相同，但 *attr_name* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

如果 *v* 为 `NULL`，该属性将被删除，但是此功能已被弃用而应改用*PyObject_DelAttrString()*。

传给该函数的不同属性名称应当保持在较少的数量，通常是通过使用静态分配的字符串作为 *attr_name* 来做到这一点。对于编译时未知的属性名称，建议直接调用*PyUnicode_FromString()* 和*PyObject_SetAttr()*。更多相关细节，请参阅*PyUnicode_InternFromString()*，它可在内部用于创建键对象。

int **PyObject_GenericSetAttr**(*PyObject* *o, *PyObject* *name, *PyObject* *value)

属于稳定 ABI. 通用的属性设置和删除函数，用于放入类型对象的*tp_setattro* 槽。它在类的字典中（位于对象的 MRO 中）查找数据描述器，如果找到，则将比在实例字典中设置或删除属性优先执行。否则，该属性将在对象的 `__dict__` 中设置或删除。如果成功将返回 `0`，否则将引发 `AttributeError` 并返回 `-1`。

int **PyObject_DelAttr**(*PyObject* *o, *PyObject* *attr_name)

属于稳定 ABI 自 *3.13* 版起. 删除对象 *o* 中名为 *attr_name* 的属性。失败时返回 `-1`。这相当于 Python 语句 `del o.attr_name`。

int **PyObject_DelAttrString**(*PyObject* *o, const char *attr_name)

属于稳定 ABI 自 *3.13* 版起. 这与*PyObject_DelAttr()* 相同，但 *attr_name* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

传给该函数的不同属性名称应当保持在较少的数量，通过是通过使用静态分配的字符串作为 *attr_name* 来做到这一点。对于编译时未知的属性名称，建议直接调用*PyUnicode_FromString()* 和*PyObject_DelAttr()*。更多相关细节，请参阅*PyUnicode_InternFromString()*，它可在内部用于创建供查找的键对象。

PyObject ***PyObject_GenericGetDict**(*PyObject* *o, void *context)

返回值：新的引用。属于稳定 ABI 自 *3.10* 版起. `__dict__` 描述符的获取函数的一种通用实现。必要时会创建该字典。

此函数还可能会被调用以获取对象 *o* 的 `__dict__`。当调用它时可传入 `NULL` 作为 *context*。由于此函数可能需要为字典分配内存，所以在访问对象上的属性时调用*PyObject_GetAttr()* 可能会更为高效。

当失败时，将返回 `NULL` 并设置一个异常。

Added in version 3.3.

int **PyObject_GenericSetDict**(*PyObject* *o, *PyObject* *value, void *context)

属于稳定 ABI 自 *3.7* 版起. `__dict__` 描述符设置函数的一种通用实现。这里不允许删除该字典。

Added in version 3.3.

PyObject ****_PyObject_GetDictPtr**(*PyObject* *obj)

返回一个指向对象 *obj* 的 `__dict__` 的指针。如果不存在 `__dict__`，则返回 `NULL` 并且不设置异常。

此函数可能需要为字典分配内存，所以在访问对象上的属性时调用*PyObject_GetAttr()* 可能会更为高效。

PyObject ***PyObject_RichCompare**(*PyObject* *o1, *PyObject* *o2, int opid)

返回值：新的引用。属于稳定 ABI. 使用由 *opid* 指定的操作来比较 *o1* 和 *o2* 的值，操作必须为*Py_LT*, *Py_LE*, *Py_EQ*, *Py_NE*, *Py_GT* 或*Py_GE* 中的一个，分别对应于 <, <=, ==, !=, > 或 >=。这等价于 Python 表达式 `o1 op o2`，其中 `op` 是与 *opid* 对应的运算符。成功时返回比较结果值，失败时返回 `NULL`。

int **PyObject_RichCompareBool**(*PyObject* *o1, *PyObject* *o2, int opid)

属于稳定 ABI. 使用 *opid* 所指定的操作，例如*PyObject_RichCompare()* 来比较 *o1* 和 *o2* 的值，但在出错时返回 `-1`，在结果为假值时返回 `0`，在其他情况下返回 `1`。

备注

如果 *o1* 和 *o2* 是同一个对象，*PyObject_RichCompareBool()* 将总是为*Py_EQ* 返回 `1` 并为*Py_NE* 返回 `0`。

PyObject ***PyObject_Format**(*PyObject* *obj, *PyObject* *format_spec)

属于稳定 ABI. 格式 *obj* 使用 *format_spec*。这等价于 Python 表达式 `format(obj, format_spec)`。

format_spec 可以为 `NULL`。在此情况下调用将等价于 `format(obj)`。成功时返回已格式化的字符串，失败时返回 `NULL`。

PyObject ***PyObject_Repr**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 计算对象 *o* 的字符串形式。成功时返回字符串，失败时返回 `NULL`。这相当于 Python 表达式 `repr(o)`。由内置函数 `repr()` 调用。

在 3.4 版本发生变更: 该函数现在包含一个调试断言，用以确保不会静默地丢弃活动的异常。

PyObject ***PyObject_ASCII**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 与*PyObject_Repr()* 一样，计算对象 *o* 的字符串形式，但在*PyObject_Repr()* 返回的字符串中用 `\x`、`\u` 或 `\U` 转义非 ASCII 字符。这将生成一个类似于 Python 2 中由*PyObject_Repr()* 返回的字符串。由内置函数 `ascii()` 调用。

PyObject ***PyObject_Str**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 计算对象 *o* 的字符串形式。成功时返回字符串，失败时返回 `NULL`。这相当于 Python 表达式 `str(o)`。由内置函数 `str()` 调用，因此也由 `print()` 函数调用。

在 3.4 版本发生变更: 该函数现在包含一个调试断言，用以确保不会静默地丢弃活动的异常。

PyObject ***PyObject_Bytes**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 计算对象 *o* 的字节形式。失败时返回 `NULL`，成功时返回一个字节串对象。这相当于 *o* 不是整数时的 Python 表达式 `bytes(o)` 。与 `bytes(o)` 不同的是，当 *o* 是整数而不是初始为 0 的字节串对象时，会触发 TypeError。

int **PyObject_IsSubclass**(*PyObject* *derived, *PyObject* *cls)

属于稳定 ABI. 如果 *derived* 类与 *cls* 类相同或为其派生类，则返回 `1`，否则返回 `0`。如果出错则返回 `-1`。

如果 *cls* 是元组，则会对 *cls* 进行逐项检测。如果至少有一次检测返回 `1`，结果将为 `1`，否则将是 `0`。

如果 *cls* 具有 `__subclasscheck__()` 方法，它将被调用以确定 **PEP 3119** 所描述的子类状态。在其他情况下，如果 *derived* 是一个直接或间接子类即包含在 `cls.__mro__` 中则它就是 *cls* 的子类。

通常只有类对象，即 `type` 或其派生类的实例才会被视为类。但是，对象可以通过设置 `__bases__` 属性（它必须是由基类组成的元组）来覆盖此定义。

int **PyObject_IsInstance**(*PyObject* *inst, *PyObject* *cls)

属于稳定 ABI. 如果 *inst* 是 *cls* 类或其子类的实例，则返回 `1`，如果不是则返回 `0`。如果出错则返回 `-1` 并设置一个异常。

如果 *cls* 是元组，则会对 *cls* 进行逐项检测。如果至少有一次检测返回 `1`，结果将为 `1`，否则将是 `0`。

如果 *cls* 具有 `__instancecheck__()` 方法，它将被调用以确定 **PEP 3119** 所描述的子类状态。在其他情况下，如果 *inst* 的类是 *cls* 的子类则它就是 *cls* 的实例。

实例 *inst* 可以通过设置 `__class__` 属性来覆盖它是否会被视为类。

对象 *cls* 可以通过设置 `__bases__` 属性（它必须是由基类组成的元组）来覆盖它是否会被视为类，及其有哪些基类。

Py_hash_t **PyObject_Hash**(*PyObject* *o)

属于稳定 ABI. 计算并返回对象的哈希值 *o*。失败时返回 `-1`。这相当于 Python 表达式 `hash(o)`。

在 3.2 版本发生变更: 现在的返回类型是 Py_hash_t。这是一个大小与*Py_ssize_t* 相同的有符号整数。

Py_hash_t **PyObject_HashNotImplemented**(*PyObject* *o)

属于稳定 ABI. 设置一个 `TypeError` 来指明 `type(o)` 不是*hashable* 并返回 `-1`。此函数在存储于 `tp_hash` 槽位内时会获得特别对待，允许某个类型显式地向解释器指明它是不可哈希对象。

int **PyObject_IsTrue**(*PyObject* *o)

属于稳定 ABI. 如果对象 *o* 被认为是 true，则返回 `1`，否则返回 `0`。这相当于 Python 表达式 `not not o`。失败则返回 `-1`。

int **PyObject_Not**(*PyObject* *o)

属于稳定 ABI. 如果对象 *o* 被认为是 true，则返回 `1`，否则返回 `0`。这相当于 Python 表达式 `not not o`。失败则返回 `-1`。

PyObject ***PyObject_Type**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 当 *o* 不为 `NULL` 时，返回一个与对象 *o* 的类型相对应的类型对象。当失败时，将引发 `SystemError` 并返回 `NULL`。这等同于 Python 表达式 `type(o)`。该函数会新建一个指向返回值的*strong reference*。实际上没有多少理由使用此函数来替代*Py_TYPE()* 函数，后者将返回一个 *PyTypeObject** 类型的指针，除非是需要一个新的*strong reference*。

int **PyObject_TypeCheck**(*PyObject* *o, *PyTypeObject* *type)

如果对象 *o* 是 *type* 类型或其子类型，则返回非零，否则返回 `0`。两个参数都必须非 `NULL`。

Py_ssize_t **PyObject_Size**(*PyObject* *o)

Py_ssize_t **PyObject_Length**(*PyObject* *o)

属于稳定 ABI. 返回对象 *o* 的长度。如果对象 *o* 支持序列和映射协议，则返回序列长度。出错时返回 `-1`。这等同于 Python 表达式 `len(o)`。

Py_ssize_t **PyObject_LengthHint**(*PyObject* *o, *Py_ssize_t* defaultvalue)

返回对象 *o* 的估计长度。首先尝试返回实际长度，然后用 `__length_hint__()` 进行估计，最后返回默认值。出错时返回 `-1`。这等同于 Python 表达式 `operator.length_hint(o, defaultvalue)`。

Added in version 3.4.

PyObject ***PyObject_GetItem**(*PyObject* *o, *PyObject* *key)

返回值：新的引用。属于稳定 ABI. 返回对象 *key* 对应的 *o* 元素，或在失败时返回 `NULL`。这等同于 Python 表达式 `o[key]`。

int **PyObject_SetItem**(*PyObject* *o, *PyObject* *key, *PyObject* *v)

属于稳定 ABI. 将对象 *key* 映射到值 *v*。失败时引发异常并返回 `-1`；成功时返回 `0`。这相当于 Python 语句 `o[key] = v`。该函数 不会偷取 *v* 的引用计数。

int **PyObject_DelItem**(*PyObject* *o, *PyObject* *key)

属于稳定 ABI. 从对象 *o* 中移除对象 *key* 的映射。失败时返回 `-1`。这相当于 Python 语句 `del o[key]`。

int **PyObject_DelItemString**(*PyObject* *o, const char *key)

属于稳定 ABI. 这与*PyObject_DelItem()* 相同，但 *key* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

PyObject ***PyObject_Dir**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 相当于 Python 表达式 `dir(o)`，返回一个（可能为空）适合对象参数的字符串列表，如果出错则返回 `NULL`。如果参数为 `NULL`，类似 Python 的 `dir()`，则返回当前 locals 的名字；这时如果没有活动的执行框架，则返回 `NULL`，但*PyErr_Occurred()* 将返回 false。

PyObject ***PyObject_GetIter**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 等同于 Python 表达式 `iter(o)`。为对象参数返回一个新的迭代器，如果该对象已经是一个迭代器，则返回对象本身。如果对象不能被迭代，会引发 `TypeError`，并返回 `NULL`。

PyObject ***PyObject_SelfIter**(*PyObject* *obj)

返回值：新的引用。属于稳定 ABI. 这等价于 Python `__iter__(self): return self` 方法。它是针对*iterator* 类型设计的，将在*PyTypeObject.tp_iter* 槽位中使用。

PyObject ***PyObject_GetAIter**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI 自 *3.10* 版起. 等同于 Python 表达式 `aiter(o)`。接受一个 `AsyncIterable` 对象，并为其返回一个 `AsyncIterator`。通常返回的是一个新迭代器，但如果参数是一个 `AsyncIterator`，将返回其自身。如果该对象不能被迭代，会引发 `TypeError`，并返回 `NULL`。

Added in version 3.10.

void ***PyObject_GetTypeData**(*PyObject* *o, *PyTypeObject* *cls)

属于稳定 ABI 自 *3.12* 版起. 获取一个指向为 *cls* 保留的子类专属数据的指针。

对象 *o* 必须为 *cls* 的实例，而 *cls* 必须使用负的*PyType_Spec.basicsize* 来创建。Python 不会检查这一点。

发生错误时，将设置异常并返回 `NULL`。

Added in version 3.12.

Py_ssize_t **PyType_GetTypeDataSize**(*PyTypeObject* *cls)

属于稳定 ABI 自 *3.12* 版起. 返回为 *cls* 保留的实例内存空间大小，即*PyObject_GetTypeData()* 所返回的内存大小。

这可能会大于使用*-PyType_Spec.basicsize* 请求到的大小；可以安全地使用这个更大的值 (例如通过 `memset()`)。

类型 *cls* **必须**使用负的*PyType_Spec.basicsize* 来创建。Python 不会检查这一点。

当失败时，将设置异常并返回一个负值。

Added in version 3.12.

void ***PyObject_GetItemData**(*PyObject* *o)

使用*Py_TPFLAGS_ITEMS_AT_END* 获取一个指向类的单独条目数据的指针。

出错时，将设置异常并返回 `NULL`。如果 *o* 没有设置*Py_TPFLAGS_ITEMS_AT_END* 则会引发 `TypeError`。

Added in version 3.12.

int **PyObject_VisitManagedDict**(*PyObject* *obj, *visitproc* visit, void *arg)

访问被管理的 *obj* 的字典。

此函数必须只在设置了*Py_TPFLAGS_MANAGED_DICT* 旗标的类型的遍历函数中被调用。

Added in version 3.13.

void **PyObject_ClearManagedDict**(*PyObject* *obj)

清空被管理的 *obj* 的字典。.

此函数必须只在设置了*Py_TPFLAGS_MANAGED_DICT* 旗标的类型的遍历函数中被调用。

Added in version 3.13.

7.2 调用协议

CPython 支持两种不同的调用协议：*tp_call* 和矢量调用。

7.2.1 *tp_call* 协议

设置*tp_call* 的类的实例都是可调用的。槽位的签名为:

```
PyObject *tp_call(PyObject *callable, PyObject *args, PyObject *kwargs);
```

一个调用是用一个元组表示位置参数，用一个 dict 表示关键字参数，类似于 Python 代码中的 `callable(*args, **kwargs)`。*args** 必须是非空的（如果没有参数，会使用一个空元组），但如果没有关键字参数，**kwargs** 可以是 *NULL*。

这个约定不仅被 *tp_call* 使用：*tp_new* 和*tp_init* 也这样传递参数。

要调用一个对象，请使用*PyObject_Call()* 或者其他的*调用 API*。

7.2.2 Vectorcall 协议

Added in version 3.9.

vectorcall 协议是在 **PEP 590** 被引入的，它是使调用函数更加有效的附加协议。

作为经验法则，如果可调用程序支持 vectorcall，CPython 会更倾向于内联调用。然而，这并不是一个硬性规定。此外，一些第三方扩展直接使用 *tp_call* (而不是使用*PyObject_Call()*)。因此，一个支持 vectorcall 的类也必须实现*tp_call*。此外，无论使用哪种协议，可调对象的行为都必须是相同的。推荐的方法是将*tp_call* 设置为*PyVectorcall_Call()*。值得一提的是:

> **⚠ 警告**
>
> 一个支持 Vectorcall 的类 **必须**也实现具有相同语义的*tp_call*。

在 3.12 版本发生变更: 现在*Py_TPFLAGS_HAVE_VECTORCALL* 旗标在类的 `__call__()` 方法被重新赋值时将会从类中移除。(这将仅在内部设置*tp_call*，因此可能使其行为不同于 vectorcall 函数。) 在更早的 Python 版本中，vectorcall 应当仅被用于*不可变对象* 或静态类型。

如果一个类的 vectorcall 比 *tp_call* 慢，就不应该实现 vectorcall。例如，如果被调用者需要将参数转换为 args 元组和 kwargs dict，那么实现 vectorcall 就没有意义。

类可以通过启用*Py_TPFLAGS_HAVE_VECTORCALL* 旗标并将*tp_vectorcall_offset* 设为对象结构体中 *vectorcallfunc* 出现位置偏移量来实现 vectorcall 协议。这是一个指向具有以下签名的函数的指针:

typedef *PyObject* *(***vectorcallfunc**)(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

属于稳定 ABI 自 *3.12* 版起.

- *callable* 是指被调用的对象。
- ***args* 是一个 C 语言数组，由位置参数和后面的**
  关键字参数的值。如果没有参数，这个值可以是 *NULL* 。
- ***nargsf* 是位置参数的数量加上可能的**
  PY_VECTORCALL_ARGUMENTS_OFFSET 旗标。要从 *nargsf* 获得位置参数的实际数量，请使用*PyVectorcall_NARGS()*。
- ***kwnames* 是一包含所有关键字名称的元组。**
  换句话说，就是 kwargs 字典的键。这些名字必须是字符串 (`str` 或其子类的实例)，并且它们必须是唯一的。如果没有关键字参数，那么 *kwnames* 可以用 *NULL* 代替。

PY_VECTORCALL_ARGUMENTS_OFFSET

属于稳定 ABI 自 *3.12* 版起. 如果在 vectorcall 的 *nargsf* 参数中设置了此标志，则允许被调用者临时更改 `args[-1]` 的值。换句话说，*args* 指向分配向量中的参数 1（不是 0 ）。被调用方必须在返回之前还原 `args[-1]` 的值。

对于*PyObject_VectorcallMethod()* ，这个标志的改变意味着 `args[0]` 可能改变了。

只要调用方能以低代价（不额外分配内存）这样做，就推荐使用*PY_VECTORCALL_ARGUMENTS_OFFSET*。这样做将允许诸如绑定方法之类的可调用对象非常高效地执行前向调用（这种调用将包括一个加在开头的 *self* 参数）。

Added in version 3.8.

要调用一个实现了 vectorcall 的对象，请使用某个*call API* 函数，就像其他可调对象一样。*PyObject_Vectorcall()* 通常是最有效的。

递归控制

在使用 *tp_call* 时，被调用者不必担心*递归*: CPython 对于使用 *tp_call* 进行的调用会使用*Py_EnterRecursiveCall()* 和*Py_LeaveRecursiveCall()*。

为保证效率，这不适用于使用 vectorcall 的调用：被调用方在需要时应当使用 *Py_EnterRecursiveCall* 和 *Py_LeaveRecursiveCall*。

Vectorcall 支持 API

Py_ssize_t **PyVectorcall_NARGS**(size_t nargsf)

属于稳定 ABI 自 *3.12* 版起. 给定一个 vectorcall *nargsf* 实参，返回参数的实际数量。目前等同于:

```
(Py_ssize_t)(nargsf & ~PY_VECTORCALL_ARGUMENTS_OFFSET)
```

然而，应使用 `PyVectorcall_NARGS` 函数以便将来扩展。

Added in version 3.8.

vectorcallfunc **PyVectorcall_Function**(*PyObject* *op)

如果 *op* 不支持 vectorcall 协议（要么是因为类型不支持，要么是因为具体实例不支持），返回 *NULL*。否则，返回存储在 *op* 中的 vectorcall 函数指针。这个函数从不触发异常。

这在检查 *op* 是否支持 vectorcall 时最有用处，可以通过检查 `PyVectorcall_Function(op) != NULL` 来实现。

Added in version 3.9.

PyObject ***PyVectorcall_Call**(*PyObject* *callable, *PyObject* *tuple, *PyObject* *dict)

属于稳定 ABI 自 *3.12* 版起. 调用 * 可调对象 * 的*vectorcallfunc*，其位置参数和关键字参数分别以元组和 dict 形式给出。

这是一个专用函数，用于放入*tp_call* 槽位或是用于 `tp_call` 的实现。它不会检查*Py_TPFLAGS_HAVE_VECTORCALL* 旗标并且它也不会回退到 `tp_call`。

Added in version 3.8.

7.2.3 调用对象的 API

有多个函数可被用来调用 Python 对象。各个函数会将其参数转换为被调用对象所支持的惯例–可以是 *tp_call* 或 vectorcall。为了尽可能少地进行转换，请选择一个适合你所拥有的数据格式的函数。

下表总结了可用的功能；请参阅各个文档以了解详细信息。

函数	callable -- 可调用对象	args	kwargs
PyObject_Call()	PyObject *	元组	dict/NULL
PyObject_CallNoArgs()	PyObject *	---	---
PyObject_CallOneArg()	PyObject *	1 个对象	---
PyObject_CallObject()	PyObject *	元组/NULL	---
PyObject_CallFunction()	PyObject *	format	---
PyObject_CallMethod()	对象 + char*	format	---
PyObject_CallFunctionObjArgs()	PyObject *	可变参数	---
PyObject_CallMethodObjArgs()	对象 + 名称	可变参数	---
PyObject_CallMethodNoArgs()	对象 + 名称	---	---
PyObject_CallMethodOneArg()	对象 + 名称	1 个对象	---
PyObject_Vectorcall()	PyObject *	vectorcall	vectorcall
PyObject_VectorcallDict()	PyObject *	vectorcall	dict/NULL
PyObject_VectorcallMethod()	参数 + 名称	vectorcall	vectorcall

PyObject ***PyObject_Call**(*PyObject* *callable, *PyObject* *args, *PyObject* *kwargs)

返回值：新的引用。属于稳定 ABI. 调用一个可调用的 Python 对象 *callable*，附带由元组 *args* 所给出的参数，以及由字典 *kwargs* 所给出的关键字参数。

args 必须不为 *NULL*；如果不想要参数请使用一个空元组。如果不想要关键字参数，则 *kwargs* 可以为 *NULL*。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

这等价于 Python 表达式 `callable(*args, **kwargs)`。

PyObject ***PyObject_CallNoArgs**(*PyObject* *callable)

返回值：新的引用。属于稳定 ABI 自 *3.10* 版起. 调用一个可调用的 Python 对象 *callable* 并不附带任何参数。这是不带参数调用 Python 可调用对象的最有效方式。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

Added in version 3.9.

PyObject ***PyObject_CallOneArg**(*PyObject* *callable, *PyObject* *arg)

返回值：新的引用。 调用一个可调用的 Python 对象 *callable* 并附带恰好 1 个位置参数 *arg* 而没有关键字参数。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

Added in version 3.9.

PyObject ***PyObject_CallObject**(*PyObject* *callable, *PyObject* *args)

返回值：新的引用。属于稳定 ABI. 调用一个可调用的 Python 对象 *callable*，附带由元组 *args* 所给出的参数。如果不想要传入参数，则 *args* 可以为 *NULL*。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

这等价于 Python 表达式 `callable(*args)`。

PyObject ***PyObject_CallFunction**(*PyObject* *callable, const char *format, ...)

返回值：新的引用。属于稳定 ABI. 调用一个可调用的 Python 对象 *callable*，附带可变数量的 C 参数。这些 C 参数使用 *Py_BuildValue()* 风格的格式字符串来描述。format 可以为 *NULL*，表示没有提供任何参数。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

这等价于 Python 表达式 `callable(*args)`。

请注意如果你只传入 *PyObject** 参数，则 *PyObject_CallFunctionObjArgs()* 是更快速的选择。

在 3.4 版本发生变更: 这个 *format* 类型已从 `char *` 更改。

PyObject ***PyObject_CallMethod**(*PyObject* *obj, const char *name, const char *format, ...)

返回值：新的引用。属于稳定 ABI. 调用 *obj* 对象中名为 *name* 的方法并附带可变数量的 C 参数。这些 C 参数由*Py_BuildValue()* 格式字符串来描述并应当生成一个元组。

格式可以为 *NULL*，表示未提供任何参数。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

这和 Python 表达式 `obj.name(arg1, arg2, ...)` 是一样的。

请注意如果你只传入 *PyObject** 参数，则*PyObject_CallMethodObjArgs()* 是更快速的选择。

在 3.4 版本发生变更: The types of *name* and *format* were changed from `char *`.

PyObject ***PyObject_CallFunctionObjArgs**(*PyObject* *callable, ...)

返回值：新的引用。属于稳定 ABI. 调用一个 Python 可调用对象 *callable*，附带可变数量的 *PyObject** 参数。这些参数以可变数量的形参的形式提供并以 *NULL* 结尾。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

这和 Python 表达式 `callable(arg1, arg2, ...)` 是一样的。

PyObject ***PyObject_CallMethodObjArgs**(*PyObject* *obj, *PyObject* *name, ...)

返回值：新的引用。属于稳定 ABI. 调用 Python 对象 *obj* 中的一个方，其中方法名称由 *name* 中的 Python 字符串对象给出。它将附带可变数量的 *PyObject** 参数被调用。这些参数以可变数量的形参的形式提供并以 *NULL* 结尾。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

PyObject ***PyObject_CallMethodNoArgs**(*PyObject* *obj, *PyObject* *name)

调用 Python 对象 *obj* 中的一个方法并不附带任何参数，其中方法名称由 *name* 中的 Python 字符串对象给出。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

Added in version 3.9.

PyObject ***PyObject_CallMethodOneArg**(*PyObject* *obj, *PyObject* *name, *PyObject* *arg)

调用 Python 对象 *obj* 中的一个方法并附带单个位置参数 *arg*，其中方法名称由 *name* 中的 Python 字符串对象给出。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

Added in version 3.9.

PyObject ***PyObject_Vectorcall**(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

属于稳定 ABI 自 *3.12* 版起. 调用一个可调用的 Python 对象 *callable*。附带的参数与*vectorcallfunc* 相同。如果 *callable* 支持*vectorcall*，则它会直接调用存放在 *callable* 中的 vectorcall 函数。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

Added in version 3.9.

PyObject ***PyObject_VectorcallDict**(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwdict)

调用 *callable* 并附带与在*vectorcall* 协议中传入的完全相同的位置参数，但会加上以字典 *kwdict* 形式传入的关键字参数。*args* 数组将只包含位置参数。

无论在内部使用哪种协议，都需要进行参数的转换。因此，此函数应当仅在调用方已经拥有作为关键字参数的字典，但没有作为位置参数的元组时才被使用。

Added in version 3.9.

PyObject ***PyObject_VectorcallMethod**(*PyObject* *name, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

属于稳定 ABI 自 *3.12* 版起. 使用 vectorcall 调用惯例来调用一个方法。方法的名称以 Python 字符串 *name* 的形式给出。调用方法的对象为 *args[0]*，而 *args* 数组从 *args[1]* 开始的部分则代表调用的参数。必须传入至少一个位置参数。*nargsf* 为包括 *args[0]* 在内的位置参数的数量，如果 `args[0]` 的值可能被临时改变则还要加上*PY_VECTORCALL_ARGUMENTS_OFFSET*。关键字参数可以像在*PyObject_Vectorcall()* 中那样传入。

如果对象具有*Py_TPFLAGS_METHOD_DESCRIPTOR* 特性，此函数将调用未绑定的方法对象并传入完整的 *args* vector 作为参数。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

Added in version 3.9.

7.2.4 调用支持 API

int **PyCallable_Check**(*PyObject* *o)

属于稳定 ABI. 确定对象 *o* 是可调对象。如果对象是可调对象则返回 `1`，其他情况返回 `0`。这个函数不会调用失败。

7.3 数字协议

int **PyNumber_Check**(*PyObject* *o)

属于稳定 ABI. 如果对象 *o* 提供数字的协议，返回真 `1`，否则返回假。这个函数不会调用失败。

在 3.8 版本发生变更: 如果 *o* 是一个索引整数则返回 `1`。

PyObject ***PyNumber_Add**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 、*o2* 相加的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 + o2`。

PyObject ***PyNumber_Subtract**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 减去 *o2* 的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 - o2`。

PyObject ***PyNumber_Multiply**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 、*o2* 相乘的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 * o2`。

PyObject ***PyNumber_MatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 返回 *o1* 、*o2* 做矩阵乘法的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 @ o2`。

Added in version 3.5.

PyObject ***PyNumber_FloorDivide**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 除以 *o2* 向下取整的值，失败时返回 `NULL`。这等价于 Python 表达式 `o1 // o2`。

PyObject ***PyNumber_TrueDivide**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 除以 *o2* 的数学值的合理近似值，或失败时返回 `NULL`。返回的是" 近似值" 因为二进制浮点数本身就是近似值；不可能以二进制精确表示所有实数。此函数可以在传入两个整数时返回一个浮点值。此函数等价于 Python 表达式 `o1 / o2`。

PyObject ***PyNumber_Remainder**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 除以 *o2* 得到的余数，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 % o2`。

PyObject ***PyNumber_Divmod**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 参考内置函数 `divmod()`。如果失败，返回 `NULL`。等价于 Python 表达式 `divmod(o1, o2)`。

PyObject ***PyNumber_Power**(*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

返回值：新的引用。属于稳定 ABI. 请参阅内置函数 `pow()`。如果失败，返回 `NULL`。等价于 Python 中的表达式 `pow(o1, o2, o3)`，其中 *o3* 是可选的。如果要忽略 *o3*，则需传入 *Py_None* 作为代替（如果传入 `NULL` 会导致非法内存访问）。

PyObject ***PyNumber_Negative**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 返回 *o* 的负值，如果失败，返回 `NULL` 。等价于 Python 表达式 `-o`。

PyObject ***PyNumber_Positive**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 返回 *o*，如果失败，返回 `NULL` 。等价于 Python 表达式 `+o`。

PyObject ***PyNumber_Absolute**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 返回 *o* 的绝对值，如果失败，返回 `NULL`。等价于 Python 表达式 `abs(o)`。

PyObject ***PyNumber_Invert**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 返回 *o* 的按位取反后的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `~o`。

PyObject ***PyNumber_Lshift**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 左移 *o2* 个比特后的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 << o2`。

PyObject ***PyNumber_Rshift**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 右移 *o2* 个比特后的结果，如果失败，返回 `NULL` 。等价于 Python 表达式 `o1 >> o2`。

PyObject ***PyNumber_And**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 和 *o2* “按位与”的结果，如果失败，返回 `NULL` 。等价于 Python 表达式 `o1 & o2`。

PyObject ***PyNumber_Xor**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 和 *o2* “按位异或”的结果，如果失败，返回 `NULL` 。等价于 Python 表达式 `o1 ^ o2`。

PyObject ***PyNumber_Or**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 和 *o2* “按位或”的结果，如果失败，返回 `NULL` 。等价于 Python 表达式 `o1 | o2`。

PyObject ***PyNumber_InPlaceAdd**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 、*o2* 相加的结果，如果失败，返回 `NULL`。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 += o2`。

PyObject ***PyNumber_InPlaceSubtract**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 、*o2* 相减的结果，如果失败，返回 `NULL` 。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 -= o2`。

PyObject ***PyNumber_InPlaceMultiply**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 、*o2** 相乘的结果，如果失败，返回 *“NULL“*。当 **o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 *= o2`。

PyObject ***PyNumber_InPlaceMatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 返回 *o1* 、*o2* 做矩阵乘法后的结果，如果失败，返回 `NULL` 。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 @= o2`。

Added in version 3.5.

PyObject ***PyNumber_InPlaceFloorDivide**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 除以 *o2* 后向下取整的结果，如果失败，返回 `NULL`。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 //= o2`。

PyObject ***PyNumber_InPlaceTrueDivide**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 除以 *o2* 的数学值的合理近似值，或失败时返回 NULL。返回的是" 近似值" 因为二进制浮点数本身就是近似值；不可能以二进制精确表示所有实数。此函数可以在传入两个整数时返回一个浮点数。此运算在 *o1* 支持的时候会 原地执行。此函数等价于 Python 语句 `o1 /= o2`。

PyObject ***PyNumber_InPlaceRemainder**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 除以 *o2* 得到的余数，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 %= o2`。

PyObject ***PyNumber_InPlacePower**(*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

返回值：新的引用。属于稳定 ABI. 请参阅内置函数 `pow()`。如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。当 *o3* 是*Py_None* 时，等价于 Python 语句 `o1 **= o2`；否则等价于在原来位置储存结果的 `pow(o1, o2, o3)` 。如果要忽略 *o3*，则需传入*Py_None* （传入 NULL 会导致非法内存访问）。

PyObject ***PyNumber_InPlaceLshift**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 左移 *o2* 个比特后的结果，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 <<= o2`。

PyObject ***PyNumber_InPlaceRshift**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 返回 *o1* 右移 *o2* 个比特后的结果，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 >>= o2`。

PyObject ***PyNumber_InPlaceAnd**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 成功时返回 *o1* 和 *o2* " 按位与" 的结果，失败时返回 NULL。在 *o1* 支持的前提下该操作将 原地执行。等价与 Python 语句 `o1 &= o2`。

PyObject ***PyNumber_InPlaceXor**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 成功时返回 *o1* 和 *o2* " 按位异或" 的结果，失败时返回 NULL。在 *o1* 支持的前提下该操作将 原地执行。等价与 Python 语句 `o1 ^= o2`。

PyObject ***PyNumber_InPlaceOr**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 成功时返回 *o1* 和 *o2* " 按位或" 的结果，失败时返回 NULL。在 *o1* 支持的前提下该操作将 原地执行。等价于 Python 语句 `o1 |= o2`。

PyObject ***PyNumber_Long**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 成功时返回 *o* 转换为整数对象后的结果，失败时返回 NULL。等价于 Python 表达式 `int(o)`。

PyObject ***PyNumber_Float**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 成功时返回 *o* 转换为浮点对象后的结果，失败时返回 NULL。等价于 Python 表达式 `float(o)`。

PyObject ***PyNumber_Index**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 成功时返回 *o* 转换为 Python int 类型后的结果，失败时返回 NULL 并引发 `TypeError` 异常。

在 3.10 版本发生变更: 结果总是为 `int` 类型。在之前版本中，结果可能为 `int` 的子类的实例。

PyObject ***PyNumber_ToBase**(*PyObject* *n, int base)

返回值：新的引用。属于稳定 ABI. 返回整数 *n* 转换成以 *base* 为基数的字符串后的结果。这个 *base* 参数必须是 2，8，10 或者 16 。对于基数 2，8，或 16 ，返回的字符串将分别加上基数标识 `'0b'`, `'0o'`, or `'0x'`。如果 *n* 不是 Python 中的整数 *int* 类型，就先通过*PyNumber_Index()* 将它转换成整数类型。

Py_ssize_t **PyNumber_AsSsize_t**(*PyObject* *o, *PyObject* *exc)

属于稳定 ABI. 如果 *o* 可以被解读为一个整数则返回 *o* 转换成的*Py_ssize_t* 值。如果调用失败，则会引发一个异常并返回 `-1`。

如果 *o* 可以被转换为 Python 的 int 值但尝试转换为*Py_ssize_t* 值则会引发 `OverflowError`，则 *exc* 参数将为所引发的异常类型 (通常为 `IndexError` 或 `OverflowError`)。如果 *exc* 为 NULL，

则异常会被清除并且值会在为负整数时被裁剪为 `PY_SSIZE_T_MIN` 而在为正整数时被裁剪为 `PY_SSIZE_T_MAX`。

int **PyIndex_Check**(*PyObject* *o)

属于稳定 ABI 自 *3.8* 版起. 返回 `1` 如果 *o* 是一个索引整数（将 `nb_index` 槽位填充到 `tp_as_number` 结构体），或者在其他情况下返回 `0`。此函数总是会成功执行。

7.4 序列协议

int **PySequence_Check**(*PyObject* *o)

属于稳定 ABI. 如果对象提供了序列协议则返回 `1`，否则返回 `0`。请注意对于具有 `__getitem__()` 方法的 Python 类返回 `1`，除非它们是 `dict` 的子类，因为在通常情况下无法确定这种类支持哪种键类型。该函数总是会成功执行。

Py_ssize_t **PySequence_Size**(*PyObject* *o)

Py_ssize_t **PySequence_Length**(*PyObject* *o)

属于稳定 ABI. 成功时返回序列中 *o* 的对象数, 失败时返回 `-1` . 相当于 Python 的 `len(o)` 表达式.

PyObject ***PySequence_Concat**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 成功时返回 *o1* 和 *o2* 的拼接，失败时返回 `NULL`。这等价于 Python 表达式 `o1 + o2`。

PyObject ***PySequence_Repeat**(*PyObject* *o, *Py_ssize_t* count)

返回值：新的引用。属于稳定 ABI. 返回序列对象 *o* 重复 *count* 次的结果，失败时返回 `NULL`。这等价于 Python 表达式 `o * count`。

PyObject ***PySequence_InPlaceConcat**(*PyObject* *o1, *PyObject* *o2)

返回值：新的引用。属于稳定 ABI. 成功时返回 *o1* 和 *o2* 的拼接，失败时返回 `NULL`。在 *o1* 支持的情况下操作将 原地完成。这等价于 Python 表达式 `o1 += o2`。

PyObject ***PySequence_InPlaceRepeat**(*PyObject* *o, *Py_ssize_t* count)

返回值：新的引用。属于稳定 ABI. Return the result of repeating sequence object 返回序列对象 *o* 重复 *count* 次的结果，失败时返回 `NULL`。在 *o* 支持的情况下该操作会 原地完成。这等价于 Python 表达式 `o *= count`。

PyObject ***PySequence_GetItem**(*PyObject* *o, *Py_ssize_t* i)

返回值：新的引用。属于稳定 ABI. 返回 *o* 中的第 *i* 号元素，失败时返回 `NULL`。这等价于 Python 表达式 `o[i]`。

PyObject ***PySequence_GetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

返回值：新的引用。属于稳定 ABI. 返回序列对象 *o* 的 *i1* 到 *i2* 的切片，失败时返回 `NULL`。这等价于 Python 表达式 `o[i1:i2]`。

int **PySequence_SetItem**(*PyObject* *o, *Py_ssize_t* i, *PyObject* *v)

属于稳定 ABI. 将对象 *v* 赋值给 *o* 的第 *i* 号元素。失败时会引发异常并返回 `-1`；成功时返回 `0`。这相当于 Python 语句 `o[i] = v`。此函数 不会改变对 *v* 的引用。

如果 *v* 为 `NULL`，元素将被删除，但是此特性已被弃用而应改用*`PySequence_DelItem()`*。

int **PySequence_DelItem**(*PyObject* *o, *Py_ssize_t* i)

属于稳定 ABI. 删除对象 *o* 的第 *i* 号元素。失败时返回 `-1`。这相当于 Python 语句 `del o[i]`。

int **PySequence_SetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2, *PyObject* *v)

属于稳定 ABI. 将序列对象 *v* 赋值给序列对象 *o* 的从 *i1* 到 *i2* 切片。这相当于 Python 语句 `o[i1:i2] = v`。

int **PySequence_DelSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

属于稳定 ABI. 删除序列对象 *o* 的从 *i1* 到 *i2* 的切片。失败时返回 `-1`。这相当于 Python 语句 `del o[i1:i2]`。

Py_ssize_t **PySequence_Count**(*PyObject* *o, *PyObject* *value)

属于稳定 ABI. 返回 *value* 在 *o* 中出现的次数，即返回使得 `o[key] == value` 的键的数量。失败时返回 `-1`。这相当于 Python 表达式 `o.count(value)`。

int **PySequence_Contains**(*PyObject* *o, *PyObject* *value)

属于稳定 ABI. 确定 *o* 是否包含 *value*。如果 *o* 中的某一项等于 *value*，则返回 `1`，否则返回 `0`。出错时，返回 `-1`。这相当于 Python 表达式 `value in o`。

Py_ssize_t **PySequence_Index**(*PyObject* *o, *PyObject* *value)

属于稳定 ABI. 返回第一个索引 *i*，其中 `o[i] == value`。出错时，返回 `-1`。相当于 Python 的 `o.index(value)` 表达式.

PyObject ***PySequence_List**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 返回一个列表对象，其内容与序列或可迭代对象 *o* 相同，失败时返回 `NULL`。返回的列表保证是一个新对象。这等价于 Python 表达式 `list(o)`。

PyObject ***PySequence_Tuple**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 返回一个元组对象，其内容与序列或可迭代对象 *o* 相同，失败时返回 `NULL`。如果 *o* 为元组，则将返回一个新的引用，在其他情况下将使用适当的内容构造一个元组。这等价于 Python 表达式 `tuple(o)`。

PyObject ***PySequence_Fast**(*PyObject* *o, const char *m)

返回值：新的引用。属于稳定 ABI. 将序列或可迭代对象 *o* 作为其他 `PySequence_Fast*` 函数族可用的对象返回。如果该对象不是序列或可迭代对象，则会引发 `TypeError` 并将 *m* 作为消息文本。失败时返回 `NULL`。

`PySequence_Fast*` 函数之所以这样命名，是因为它们会假定 *o* 是一个*PyTupleObject* 或*PyListObject* 并直接访问 *o* 的数据字段。

作为 CPython 的实现细节，如果 *o* 已经是一个序列或列表，它将被直接返回。

Py_ssize_t **PySequence_Fast_GET_SIZE**(*PyObject* *o)

在 *o* 由*PySequence_Fast()* 返回且 *o* 不为 `NULL` 的情况下返回 *o* 长度。也可以通过在 *o* 上调用*PySequence_Size()* 来获取大小，但是*PySequence_Fast_GET_SIZE()* 的速度更快因为它可以假定 *o* 为列表或元组。

PyObject ***PySequence_Fast_GET_ITEM**(*PyObject* *o, *Py_ssize_t* i)

返回值：借入的引用。在 *o* 由*PySequence_Fast()* 返回且 *o* 不 `NULL`，并且 *i* d 在索引范围内的情况下返回 *o* 的第 *i* 号元素。

PyObject ****PySequence_Fast_ITEMS**(*PyObject* *o)

返回 PyObject 指针的底层数组。假设 *o* 由*PySequence_Fast()* 返回且 *o* 不为 `NULL`。

请注意，如果列表调整大小，重新分配可能会重新定位 items 数组. 因此，仅在序列无法更改的上下文中使用基础数组指针.

PyObject ***PySequence_ITEM**(*PyObject* *o, *Py_ssize_t* i)

返回值：新的引用。返回 *o* 的第 *i* 个元素或在失败时返回 `NULL`。此形式比*PySequence_GetItem()* 理馔，但不会检查 *o* 上的*PySequence_Check()* 是否为真值，也不会对负序号进行调整。

7.5 映射协议

参见*PyObject_GetItem()*、*PyObject_SetItem()* 与*PyObject_DelItem()*。

int **PyMapping_Check**(*PyObject* *o)

属于稳定 ABI. 如果对象提供了映射协议或是支持切片则返回 `1`，否则返回 `0`。请注意它将为具有 `__getitem__()` 方法的 Python 类返回 `1`，因为在通常情况下无法确定该类支持哪种键类型。此函数总是会成功执行。

Py_ssize_t **PyMapping_Size**(*PyObject* *o)

Py_ssize_t **PyMapping_Length**(*PyObject* *o)

属于稳定 ABI. 成功时返回对象 *o* 中键的数量，失败时返回 `-1`。这相当于 Python 表达式 `len(o)`。

PyObject ***PyMapping_GetItemString**(*PyObject* *o, const char *key)

返回值：新的引用。属于稳定 ABI. 这与 *PyObject_GetItem()* 相同，但 *key* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

int **PyMapping_GetOptionalItem**(*PyObject* *obj, *PyObject* *key, *PyObject* **result)

属于稳定 ABI 自 *3.13* 版起. *PyObject_GetItem()* 的变化形式，它在未找到键时不会引发 `KeyError`。

如果找到了键，则返回 `1` 并将 ***result* 设为指向相应值的新*strong reference*。如果未找到键，则返回 `0` 并将 ***result* 设为 `NULL`；`KeyError` 会被静默。如果引发了 `KeyError` 以外的错误，则返回 `-1` 并将 ***result* 设为 `NULL`。

Added in version 3.13.

int **PyMapping_GetOptionalItemString**(*PyObject* *obj, const char *key, *PyObject* **result)

属于稳定 ABI 自 *3.13* 版起. 这与 *PyMapping_GetOptionalItem()* 相同，但 *key* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

Added in version 3.13.

int **PyMapping_SetItemString**(*PyObject* *o, const char *key, *PyObject* *v)

属于稳定 ABI. 这与 *PyObject_SetItem()* 相同，但 *key* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

int **PyMapping_DelItem**(*PyObject* *o, *PyObject* *key)

这是 *PyObject_DelItem()* 的一个别名。

int **PyMapping_DelItemString**(*PyObject* *o, const char *key)

这与 *PyObject_DelItem()* 相同，但 *key* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

int **PyMapping_HasKeyWithError**(*PyObject* *o, *PyObject* *key)

属于稳定 ABI 自 *3.13* 版起. 如果映射对象具有键 *key* 则返回 `1`，否则返回 `0`。这相当于 Python 表达式 `key in o`。当失败时，将返回 `-1`。

Added in version 3.13.

int **PyMapping_HasKeyStringWithError**(*PyObject* *o, const char *key)

属于稳定 ABI 自 *3.13* 版起. 这与 *PyMapping_HasKeyWithError()* 相同，但 *key* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

Added in version 3.13.

int **PyMapping_HasKey**(*PyObject* *o, *PyObject* *key)

属于稳定 ABI. 如果映射对象具有键 *key* 则返回 `1`，否则返回 `0`。这相当于 Python 表达式 `key in o`。此函数总是会成功执行。

> **备注**
>
> 在其调用 `__getitem__()` 方法时发生的异常将被静默地忽略。想要进行适当的错误处理，请改用 *PyMapping_HasKeyWithError()*，*PyMapping_GetOptionalItem()* 或 *PyObject_GetItem()*。

int **PyMapping_HasKeyString**(*PyObject* *o, const char *key)

属于稳定 ABI. 这与 *PyMapping_HasKey()* 相同，但 *key* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

> **备注**
>
> 在其调用 `__getitem__()` 方法或创建临时 `str` 对象时发生的异常将被静默地忽略。想要进行适当的错误处理，请改用*PyMapping_HasKeyStringWithError()*，*PyMapping_GetOptionalItemString()* 或*PyMapping_GetItemString()*。

PyObject ***PyMapping_Keys**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 成功时，返回对象 *o* 中的键的列表。失败时，返回 `NULL`。

在 3.7 版本发生变更: 在之前版本中，此函数返回一个列表或元组。

PyObject ***PyMapping_Values**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 成功时，返回对象 *o* 中的值的列表。失败时，返回 `NULL`。

在 3.7 版本发生变更: 在之前版本中，此函数返回一个列表或元组。

PyObject ***PyMapping_Items**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 成功时，返回对象 *o* 中条目的列表，其中每个条目是一个包含键值对的元组。失败时，返回 `NULL`。

在 3.7 版本发生变更: 在之前版本中，此函数返回一个列表或元组。

7.6 迭代器协议

迭代器有两个函数。

int **PyIter_Check**(*PyObject* *o)

属于稳定 ABI 自 *3.8* 版起. 如果对象 Return non-zero if the object *o* 可以被安全把传给*PyIter_Next()* 则返回非零值，否则返回 `0`。此函数总是会成功执行。

int **PyAIter_Check**(*PyObject* *o)

属于稳定 ABI 自 *3.10* 版起. 如果对象 *o* 提供了 `AsyncIterator` 协议则返回非零值，否则返回 `0`。此函数总是会成功执行。

Added in version 3.10.

PyObject ***PyIter_Next**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 从迭代器 *o* 返回下一个值。对象必须可被*PyIter_Check()* 确认为迭代器（需要调用方来负责检查）。如果没有剩余的值，则返回 `NULL` 并且不设置异常。如果在获取条目时发生了错误，则返回 `NULL` 并且传递异常。

要为迭代器编写一个一个循环，C 代码应该看起来像这样

```
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* 传播错误 */
}

while ((item = PyIter_Next(iterator))) {
    /* 对 item 进行一些操作 */
    ...
    /* 完成后释放引用 */
    Py_DECREF(item);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
    /* 传播错误 */
```

(续下页)

(接上页)

```
}
else {
    /* 继续进行有用的工作 */
}
```

type **PySendResult**

用于代表*PyIter_Send()* 的不同结果的枚举值。

Added in version 3.10.

PySendResult **PyIter_Send**(*PyObject* *iter, *PyObject* *arg, *PyObject* **presult)

属于稳定 ABI 自 *3.10* 版起. 将 *arg* 值发送到迭代器 *iter*。返回:

- PYGEN_RETURN，如果迭代器返回的话。返回值会通过 *presult* 来返回。
- PYGEN_NEXT，如果迭代器生成值的话。生成的值会通过 *presult* 来返回。
- PYGEN_ERROR，如果迭代器引发异常的话。*presult* 会被设为 NULL。

Added in version 3.10.

7.7 缓冲协议

在 Python 中可使用一些对象来包装对底层内存数组或称 *缓冲* 的访问。此类对象包括内置的 bytes 和 bytearray 以及一些如 array.array 这样的扩展类型。第三方库也可能会为了特殊的目的而定义它们自己的类型，例如用于图像处理和数值分析等。

虽然这些类型中的每一种都有自己的语义，但它们具有由可能较大的内存缓冲区支持的共同特征。在某些情况下，希望直接访问该缓冲区而无需中间复制。

Python 以*缓冲区协议* 的形式在 C 和 Python 层级上提供这样的功能。此协议包括两个方面:

- 在生产者方面，一个类型可以导出一个 “缓冲区接口”，它允许将该类型的所有对象对外暴露有关其下层缓冲区的信息。此接口的描述参见*缓冲区对象结构体* 一节；Python 层级参见 python-buffer-protocol。
- 在消费者方面，有几种方式可被用于获得指向一个对象的原始底层数据的指针（例如一个方法的形参）。Python 层级参见 memoryview。

一些简单的对象例如 bytes 和 bytearray 会以面向字节的形式公开它们的底层缓冲区。也可能会用其他形式；例如 array.array 所公开的元素可以是多字节值。

缓冲区接口的消费者的一个例子是文件对象的 write() 方法：任何可以输出为一系列字节流的对象都可以被写入文件。然而 write() 只需要对传入对象内容的只读权限，其他的方法如 readinto() 需要对参数内容的写入权限。缓冲区接口使用对象可以选择性地允许或拒绝读写或只读缓冲区的导出。

对于缓冲区接口的使用者而言，有两种方式来获取一个目的对象的缓冲:

- 使用正确的参数来调用*PyObject_GetBuffer()* 函数;
- 调用*PyArg_ParseTuple()* (或其同级对象之一) 并传入 y*, w* or s* *格式代码* 中的一个。

在这两种情况下，当不再需要缓冲区时必须调用*PyBuffer_Release()* 。如果此操作失败，可能会导致各种问题，例如资源泄漏。

Added in version 3.12: 现在可在 Python 中使用缓冲区协议，参见 python-buffer-protocol 和 memoryview。

7.7.1 缓冲区结构

缓冲区结构 (或者简单地称为 “buffers”) 对于将二进制数据从另一个对象公开给 Python 程序员非常有用。它们还可以用作零拷贝切片机制。使用它们引用内存块的能力，可以很容易地将任何数据公开给 Python 程序员。内存可以是 C 扩展中的一个大的常量数组，也可以是在传递到操作系统库之前用于操作的原始内存块，或者可以用来传递本机内存格式的结构化数据。

与 Python 解释器公开的大多部数据类型不同，缓冲区不是*PyObject* 指针而是简单的 C 结构。这使得它们可以非常简单地创建和复制。当需要为缓冲区加上泛型包装器时，可以创建一个内存视图 对象。

有关如何编写并导出对象的简短说明，请参阅缓冲区对象结构。要获取缓冲区对象，请参阅*PyObject_GetBuffer()*。

type **Py_buffer**

属于稳定 ABI （包括所有成员）自 *3.11* 版起.

void ***buf**

指向由缓冲区字段描述的逻辑结构开始的指针。这可以是导出程序底层物理内存块中的任何位置。例如，使用负的*strides* 值可能指向内存块的末尾。

对于*contiguous* ，‘邻接’数组，值指向内存块的开头。

PyObject ***obj**

对导出对象的新引用。该引用由消费方拥有，并由*PyBuffer_Release()* 自动释放（即引用计数递减）并设置为 `NULL`。该字段相当于任何标准 C-API 函数的返回值。

作为一种特殊情况，对于由*PyMemoryView_FromBuffer()* 或*PyBuffer_FillInfo()* 包装的 *temporary* 缓冲区，此字段为 `NULL`。通常，导出对象不得使用此方案。

Py_ssize_t **len**

`product(shape) * itemsize`。对于连续数组，这是基础内存块的长度。对于非连续数组，如果逻辑结构复制到连续表示形式，则该长度将具有该长度。

仅当缓冲区是通过保证连续性的请求获取时，才访问 `((char *)buf)[0]` up to `((char *)buf)[len-1]` 时才有效。在大多数情况下，此类请求将为*PyBUF_SIMPLE* 或*PyBUF_WRITABLE*。

int **readonly**

缓冲区是否为只读的指示器。此字段由*PyBUF_WRITABLE* 标志控制。

Py_ssize_t **itemsize**

单个元素的项大小（以字节为单位）。与 `struct.calcsize()` 调用非 `NULL` *format* 的值相同。

重要例外：如果使用者请求的缓冲区没有*PyBUF_FORMAT* 标志，*format* 将设置为 `NULL`，但*itemsize* 仍具有原始格式的值。

如果*shape* 存在，则相等的 `product(shape) * itemsize == len` 仍然存在，使用者可以使用*itemsize* 来导航缓冲区。

如果*shape* 是 `NULL`，因为结果为*PyBUF_SIMPLE* 或*PyBUF_WRITABLE* 请求，则使用者必须忽略*itemsize*，并假设 `itemsize == 1`。

char ***format**

在 `struct` 模块样式语法中以 *NULL* 结束的字符串描述单个条目的内容。如果这是 `NULL`，将假定为 `"B"` (无符号字节型)。

此字段由*PyBUF_FORMAT* 标志控制。

int **ndim**

内存表示为 n 维数组形式对应的维度数。如果为 `0`，则*buf* 指向表示标量的单个条目。在这种情况下，*shape*, *strides* 和*suboffsets* 必须为 `NULL`。最大维度数由*PyBUF_MAX_NDIM* 给出。

Py_ssize_t ***shape**

一个长度为*Py_ssize_t* 的数组*ndim* 表示作为 n 维数组的内存形状。请注意，`shape[0] * ... * shape[ndim-1] * itemsize` 必须等于*len*。

Shape 形状数组中的值被限定在 `shape[n] >= 0` 。`shape[n] == 0` 这一情形需要特别注意。更多信息请参阅*complex arrays* 。

shape 数组对于使用者来说是只读的。

Py_ssize_t ***strides**

一个长度为*Py_ssize_t* 的数组*ndim* 给出要跳过的字节数以获取每个尺寸中的新元素。

Stride 步幅数组中的值可以为任何整数。对于常规数组，步幅通常为正数，但是使用者必须能够处理 `strides[n] <= 0` 的情况。更多信息请参阅*complex arrays* 。

strides 数组对用户来说是只读的。

Py_ssize_t ***suboffsets**

一个长度为*ndim* 类型为*Py_ssize_t* 的数组。如果 `suboffsets[n] >= 0`，则第 n 维存储的是指针，suboffset 值决定了解除引用时要给指针增加多少字节的偏移。suboffset 为负值，则表示不应解除引用（在连续内存块中移动）。

如果所有子偏移均为负（即无需取消引用），则此字段必须为 `NULL`（默认值）。

Python Imaging Library (PIL) 中使用了这种类型的数组表达方式。请参阅*complex arrays* 来了解如何从这样一个数组中访问元素。

suboffsets 数组对于使用者来说是只读的。

void ***internal**

供输出对象内部使用。比如可能被输出程序重组为一个整数，用于存储一个标志，标明在缓冲区释放时是否必须释放 shape、strides 和 suboffsets 数组。消费者程序 不得修改该值。

常量:

PyBUF_MAX_NDIM

内存表示的最大维度数。导出程序必须遵守这个限制，多维缓冲区的使用者应该能够处理最多 `PyBUF_MAX_NDIM` 个维度。目前设置为 64。

7.7.2 缓冲区请求的类型

通常，通过*PyObject_GetBuffer()* 向输出对象发送缓冲区请求，即可获得缓冲区。由于内存的逻辑结构复杂，可能会有很大差异，缓冲区使用者可用 *flags* 参数指定其能够处理的缓冲区具体类型。

所有*Py_buffer* 字段均由请求类型无歧义地定义。

与请求无关的字段

以下字段不会被 *flags* 影响，并且必须总是用正确的值填充：*obj*, *buf*，*len*，*itemsize*，*ndim*。

只读，格式

PyBUF_WRITABLE

控制*readonly* 字段。如果设置了，输出程序必须提供一个可写的缓冲区，否则将报告失败。在其他情况下，输出程序可以提供一个只读或可写的缓冲区，但该选择必须对所有消费者程序保持一致。例如，可以使用 *PyBUF_SIMPLE* | *PyBUF_WRITABLE* 来请求一个简单的可写缓冲区。be used to request a simple writable buffer.

PyBUF_FORMAT

控制*format* 字段。如果设置，则必须正确填写此字段。其他情况下，此字段必须为 `NULL`。

PyBUF_WRITABLE 可以和下一节的所有标志联用。由于*PyBUF_SIMPLE* 定义为 0，所以*PyBUF_WRITABLE* 可以作为一个独立的标志，用于请求一个简单的可写缓冲区。

PyBUF_FORMAT 必须与任何旗标执行 | 运算但*PyBUF_SIMPLE* 除外，因为后者已应用了格式 `B` (无符号字节串)。`PyBUF_FORMAT` 不可单独使用。

形状，步幅，子偏移量

控制内存逻辑结构的标志按照复杂度的递减顺序列出。注意，每个标志包含它下面的所有标志。

请求	形状	步幅	子偏移量
PyBUF_INDIRECT	是	是	如果需要的话
PyBUF_STRIDES	是	是	NULL
PyBUF_ND	是	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

连续性的请求

可以显式地请求 C 或 Fortran 连续，不管有没有步幅信息。若没有步幅信息，则缓冲区必须是 C-连续的。

请求	形状	步幅	子偏移量	邻接
PyBUF_C_CONTIGUOUS	是	是	NULL	C
PyBUF_F_CONTIGUOUS	是	是	NULL	F
PyBUF_ANY_CONTIGUOUS	是	是	NULL	C 或 F
PyBUF_ND	是	NULL	NULL	C

复合请求

所有可能的请求都由上一节中某些标志的组合完全定义。为方便起见，缓冲区协议提供常用的组合作为单个标志。

在下表中，*U* 代表连续性未定义。消费者程序必须调用*PyBuffer_IsContiguous()* 以确定连续性。

请求	形状	步幅	子偏移量	邻接	只读	format
PyBUF_FULL	是	是	如果需要的话	U	0	是
PyBUF_FULL_RO	是	是	如果需要的话	U	1 或 0	是
PyBUF_RECORDS	是	是	NULL	U	0	是
PyBUF_RECORDS_RO	是	是	NULL	U	1 或 0	是
PyBUF_STRIDED	是	是	NULL	U	0	NULL
PyBUF_STRIDED_RO	是	是	NULL	U	1 或 0	NULL
PyBUF_CONTIG	是	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	是	NULL	NULL	C	1 或 0	NULL

7.7.3 复杂数组

NumPy-风格：形状和步幅

NumPy 风格数组的逻辑结构由*itemsize* 、*ndim* 、*shape* 和*strides* 定义。

如果 `ndim == 0` ，*buf* 指向的内存位置被解释为大小为*itemsize* 的标量。这时，*shape* 和*strides* 都为 `NULL`。

如果*strides* 为 `NULL`，则数组将被解释为一个标准的 n 维 C 语言数组。否则，消费者程序必须按如下方式访问 n 维数组：

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

如上所述，*buf* 可以指向实际内存块中的任意位置。输出者程序可以用该函数检查缓冲区的有效性。

```
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """验证形参代表已分配内存范围内一个可用的数组：
           char *mem: 物理内存块的起始
           memlen: 物理内存块的长度
           offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False

    if ndim <= 0:
        return ndim == 0 and not shape and not strides
    if 0 in shape:
```

(续下页)

(接上页)

```
        return True

    imin = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] <= 0)
    imax = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] > 0)

    return 0 <= offset+imin and offset+imax+itemsize <= memlen
```

PIL-风格：形状，步幅和子偏移量

除了常规项之外，PIL 风格的数组还可以包含指针，必须跟随这些指针才能到达维度的下一个元素。例如，常规的三维 C 语言数组 `char v[2][2][3]` 可以看作是一个指向 2 个二维数组的 2 个指针：`char (*v[2])[2][3]`。在子偏移表示中，这两个指针可以嵌入在*buf* 的开头，指向两个可以位于内存任何位置的 `char x[2][3]` 数组。

这是一个函数，当 n 维索引所指向的 N-D 数组中有 `NULL` 步长和子偏移量时，它返回一个指针

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                       Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

7.7.4 缓冲区相关函数

int **PyObject_CheckBuffer**(*PyObject* *obj)

属于稳定 ABI 自 *3.11* 版起. 如果 *obj* 支持缓冲区接口，则返回 `1`，否则返回 `0`。返回 `1` 时不保证*PyObject_GetBuffer()* 一定成功。本函数一定调用成功。

int **PyObject_GetBuffer**(*PyObject* *exporter, *Py_buffer* *view, int flags)

属于稳定 ABI 自 *3.11* 版起. 向 *exporter* 发送请求以按照 *flags* 指定的内容填充 *view*。如果 exporter 无法提供要求类型的缓冲区，则它必须引发 `BufferError`，将 `view->obj` 设为 `NULL` 并返回 `-1`。

成功时，填充 *view*，将 `view->obj` 设为对 *exporter* 的新引用，并返回 0。当链式缓冲区提供程序将请求重定向到一个对象时，`view->obj` 可以引用该对象而不是 *exporter* (参见缓冲区对象结构)。

PyObject_GetBuffer() 必须与*PyBuffer_Release()* 同时调用成功，类似于 `malloc()` 和 `free()`。因此，消费者程序用完缓冲区后，*PyBuffer_Release()* 必须保证被调用一次。

void **PyBuffer_Release**(*Py_buffer* *view)

属于稳定 ABI 自 *3.11* 版起. 释放缓冲区 *view* 并释放对视图的支持对象 `view->obj` 的*strong reference* (即递减引用计数)。该函数必须在缓冲区不再使用时调用，否则可能会发生引用泄漏。

若该函数针对的缓冲区不是通过*PyObject_GetBuffer()* 获得的，将会出错。

Py_ssize_t **PyBuffer_SizeFromFormat**(const char *format)

属于稳定 ABI 自 *3.11* 版起. 从*format* 返回隐含的*itemsize*。如果出错，则引发异常并返回 -1。

Added in version 3.9.

int **PyBuffer_IsContiguous**(const *Py_buffer* *view, char order)

属于稳定 ABI 自 *3.11* 版起. 如果 *view* 定义的内存是 C 风格（*order* 为 `'C'`）或 Fortran 风格（*order* 为 `'F'`）*contiguous* 或其中之一（*order* 是 `'A'`），则返回 `1`。否则返回 `0`。该函数总会成功。

void *__PyBuffer_GetPointer__(const *Py_buffer* *view, const *Py_ssize_t* *indices)

属于稳定 ABI 自 *3.11* 版起. 获取给定 *view* 内的 *indices* 所指向的内存区域。*indices* 必须指向一个 `view->ndim` 索引的数组。

int __PyBuffer_FromContiguous__(const *Py_buffer* *view, const void *buf, *Py_ssize_t* len, char fort)

属于稳定 ABI 自 *3.11* 版起. 从 *buf* 复制连续的 *len* 字节到 *view* 。*fort* 可以是 `'C'` 或 `'F'` （对应于 C 风格或 Fortran 风格的顺序）。成功时返回 `0` ，错误时返回 `-1` 。

int __PyBuffer_ToContiguous__(void *buf, const *Py_buffer* *src, *Py_ssize_t* len, char order)

属于稳定 ABI 自 *3.11* 版起. 从 *src* 复制 *len* 字节到 *buf* ，成为连续字节串的形式。*order* 可以是 `'C'` 或 `'F'` 或 `'A'` （对应于 C 风格、Fortran 风格的顺序或其中任意一种）。成功时返回 `0` ，出错时返回 `-1` 。

如果 *len* != *src->len* 则此函数将报错。

int __PyObject_CopyData__(*PyObject* *dest, *PyObject* *src)

属于稳定 ABI 自 *3.11* 版起. 将数据从 *src* 拷贝到 *dest* 缓冲区。可以在 C 风格或 Fortran 风格的缓冲区之间进行转换。

成功时返回 `0`，出错时返回 `-1`。

void __PyBuffer_FillContiguousStrides__(int ndims, *Py_ssize_t* *shape, *Py_ssize_t* *strides, int itemsize, char order)

属于稳定 ABI 自 *3.11* 版起. 用给定形状的*contiguous* 字节串数组 (如果 *order* 为 `'C'` 则为 C 风格，如果 *order* 为 `'F'` 则为 Fortran 风格) 来填充 *strides* 数组，每个元素具有给定的字节数。

int __PyBuffer_FillInfo__(*Py_buffer* *view, *PyObject* *exporter, void *buf, *Py_ssize_t* len, int readonly, int flags)

属于稳定 ABI 自 *3.11* 版起. 处理导出程序的缓冲区请求，该导出程序要公开大小为 *len* 的 *buf* ，并根据 *readonly* 设置可写性。*bug* 被解释为一个无符号字节序列。

参数 *flags* 表示请求的类型。该函数总是按照 flag 指定的内容填入 *view*，除非 *buf* 设为只读，并且 *flag* 中设置了 *PyBUF_WRITABLE* 标志。

成功时，将 `view->obj` 设为对 *exporter* 的新引用并返回 0。否则，引发 `BufferError`，将 `view->obj` 设为 `NULL` 并返回 `-1`;

如果此函数用作*getbufferproc* 的一部分，则 *exporter* 必须设置为导出对象，并且必须在未修改的情况下传递 *flags*。否则，*exporter* 必须是 `NULL`。

CHAPTER 8

具体的对象层

本章中的函数特定于某些 Python 对象类型。将错误类型的对象传递给它们并不是一个好主意；如果您从 Python 程序接收到一个对象，但不确定它是否具有正确的类型，则必须首先执行类型检查；例如，要检查对象是否为字典，请使用*PyDict_Check()*。本章的结构类似于 Python 对象类型的“家族树”。

警告

虽然本章所描述的函数会仔细检查传入对象的类型，但是其中许多函数不会检查传入的对象是否为 NULL。允许传入 NULL 可能导致内存访问冲突和解释器的立即终止。

8.1 基本对象

本节描述 Python 类型对象和单一实例对象 象 None。

8.1.1 类型对象

type **PyTypeObject**

属于受限 API （作为不透明的结构体）. 对象的 C 结构用于描述 built-in 类型。

PyTypeObject **PyType_Type**

属于稳定 ABI. 这是属于 type 对象的 type object，它在 Python 层面和 `type` 是相同的对象。

int **PyType_Check**(*PyObject* *o)

如果对象 *o* 是一个类型对象，包括派生自标准类型对象的类型实例则返回非零值。在所有其它情况下都返回 0。此函数将总是成功执行。

int **PyType_CheckExact**(*PyObject* *o)

如果对象 *o* 是一个类型对象，但不是标准类型对象的子类型则返回非零值。在所有其它情况下都返回 0。此函数将总是成功执行。

unsigned int **PyType_ClearCache**()

属于稳定 ABI. 清空内部查找缓存。返回当前版本标签。

unsigned long **PyType_GetFlags**(*PyTypeObject* *type)

属于稳定 ABI. 返回 *type* 的*tp_flags* 成员。此函数主要是配合 `Py_LIMITED_API` 使用；单独的旗

标位会确保在各个 Python 发布版之间保持稳定，但对*tp_flags* 本身的访问并不是受限 *API* 的一部分。

Added in version 3.2.

在 3.4 版本发生变更: 返回类型现在是 `unsigned long` 而不是 `long`。

PyObject ***PyType_GetDict**(*PyTypeObject* *type)

返回类型对象的内部命名空间，它在其他情况下只能通过只读代理 (`cls.__dict__`) 对外公开。这可以代替直接访问*tp_dict*。返回的字典必须视为是只读的。

该函数用于特定的嵌入和语言绑定场景，在这些场景下需要直接访问该字典而间接访问（例如通过代理或*PyObject_GetAttr()* 访问）并不足够。

扩展模块在设置它们自己的类型时应当继续直接或间接地使用 `tp_dict`。

Added in version 3.12.

void **PyType_Modified**(*PyTypeObject* *type)

属于稳定 ABI. 使该类型及其所有子类型的内部查找缓存失效。此函数必须在对该类型的属性或基类进行任何手动修改之后调用。

int **PyType_AddWatcher**(*PyType_WatchCallback* callback)

注册 *callback* 作为类型监视器。返回一个非负的整数 ID，它必须传给将来对*PyType_Watch()* 的调用。如果出错（例如没有足够的可用监视器 ID），则返回 `-1` 并设置一个异常。

在自由线程构建版中，*PyType_AddWatcher()* 不是线程安全的，因此它必须一开始就被调用（在产生第一个线程之前）。

Added in version 3.12.

int **PyType_ClearWatcher**(int watcher_id)

清除由 *watcher_id* (之前从*PyType_AddWatcher()* 返回) 所标识的 watcher。成功时返回 `0`，出错时（例如 *watcher_id* 未被注册）返回 `-1`。

扩展在调用 `PyType_ClearWatcher` 时绝不能使用不是之前调用*PyType_AddWatcher()* 所返回的 *watcher_id*。

Added in version 3.12.

int **PyType_Watch**(int watcher_id, *PyObject* *type)

将 *type* 标记为已监视。每当*PyType_Modified()* 报告 *type* 发生变化时*PyType_AddWatcher()* 赋予 *watcher_id* 的回调将被调用。(如果在 *type* 的一系列连续修改之间没有调用 `_PyType_Lookup()`，则回调只能被调用一次；这是一个实现细节并可能发生变化)。

扩展在调用 `PyType_Watch` 时绝不能使用不是之前调用*PyType_AddWatcher()* 所返回的 *watcher_id*。

Added in version 3.12.

typedef int (***PyType_WatchCallback**)(*PyObject* *type)

类型监视器回调函数的类型。

回调不可以修改 *type* 或是导致*PyType_Modified()* 在 *type* 或其 MRO 中的任何类型上被调用；违反此规则可能导致无限递归。

Added in version 3.12.

int **PyType_HasFeature**(*PyTypeObject* *o, int feature)

如果类型对象 *o* 设置了特性 *feature* 则返回非零值。类型特性是用单个比特位旗标来表示的。

int **PyType_IS_GC**(*PyTypeObject* *o)

如果类型对象包括了对循环检测器的支持则返回真值；这将测试类型旗标*Py_TPFLAGS_HAVE_GC*。

int **PyType_IsSubtype**(*PyTypeObject* *a, *PyTypeObject* *b)

属于稳定 ABI. 如果 *a* 是 *b* 的子类型则返回真值。

此函数只检查实际的子类型，这意味着 `__subclasscheck__()` 不会在 *b* 上被调用。请调用*PyObject_IsSubclass()* 来执行与 `issubclass()` 所做的相同检查。

PyObject ***PyType_GenericAlloc**(*PyTypeObject* *type, *Py_ssize_t* nitems)

返回值：新的引用。属于稳定 ABI. 类型对象的*tp_alloc* 槽位的通用处理器。请使用 Python 的默认内存分配机制来分配一个新的实例并将其所有内容初始化为 `NULL`。

PyObject ***PyType_GenericNew**(*PyTypeObject* *type, *PyObject* *args, *PyObject* *kwds)

返回值：新的引用。属于稳定 ABI. 类型对象的*tp_new* 槽位的通用处理器。请使用类型的*tp_alloc* 槽位来创建一个新的实例。

int **PyType_Ready**(*PyTypeObject* *type)

属于稳定 ABI. 最终化一个类型对象。这应当在所有类型对象上调用以完成它们的初始化。此函数会负责从一个类型的基类添加被继承的槽位。成功时返回 `0`，或是在出错时返回 `-1` 并设置一个异常。

> **备注**
>
> 如果某些基类实现了 GC 协议并且所提供的类型的旗标中未包括*Py_TPFLAGS_HAVE_GC*，则将自动从其父类实现 GC 协议。相反地，如果被创建的类型的旗标中确实包含*Py_TPFLAGS_HAVE_GC* 则它 **必须**自己实现 GC 协议，至少要实现*tp_traverse* 句柄。

PyObject ***PyType_GetName**(*PyTypeObject* *type)

返回值：新的引用。属于稳定 ABI 自 *3.11* 版起. 返回类型名称。等同于获取类型的 `__name__` 属性。

Added in version 3.11.

PyObject ***PyType_GetQualName**(*PyTypeObject* *type)

返回值：新的引用。属于稳定 ABI 自 *3.11* 版起. 返回类型的限定名称。等同于获取类型的 `__qualname__` 属性。

Added in version 3.11.

PyObject ***PyType_GetFullyQualifiedName**(*PyTypeObject* *type)

属于稳定 ABI 自 *3.13* 版起. 返回类型的完整限定名称。等同于 `f"{type.__module__}.{type.__qualname__}"`，或者如果 `type.__module__` 不是字符串或是等于 `"builtins"` 则等同于 `type.__qualname__`。

Added in version 3.13.

PyObject ***PyType_GetModuleName**(*PyTypeObject* *type)

属于稳定 ABI 自 *3.13* 版起. 返回类型的模块名称。等价于获取 `type.__module__` 属性。

Added in version 3.13.

void ***PyType_GetSlot**(*PyTypeObject* *type, int slot)

属于稳定 ABI 自 *3.4* 版起. 返回存储在给定槽位中的函数指针。如果结果为 `NULL`，则表示或者该槽位为 `NULL`，或者该函数调用传入了无效的形参。调用方通常要将结果指针转换到适当的函数类型。

请参阅*PyType_Slot.slot* 查看可用的 *slot* 参数值。

Added in version 3.4.

在 3.10 版本发生变更: *PyType_GetSlot()* 现在可以接受所有类型。在此之前，它被限制为堆类型。

PyObject ***PyType_GetModule**(*PyTypeObject* *type)

属于稳定 ABI 自 *3.10* 版起. 返回当使用*PyType_FromModuleAndSpec()* 创建类型时关联到给定类型的模块对象。

如果没有关联到给定类型的模块，则设置 `TypeError` 并返回 `NULL`。

此函数通常被用于获取方法定义所在的模块。请注意在这样的方法中，`PyType_GetModule(Py_TYPE(self))` 可能不会返回预期的结果。`Py_TYPE(self)` 可以是目标类的一个子类，而子类并不一定是在与其超类相同的模块中定义的。请参阅*PyCMethod* 了解如何获取方法定义所在的类。请参阅*PyType_GetModuleByDef()* 了解有关无法使用 `PyCMethod` 的情况。

Added in version 3.9.

void ***PyType_GetModuleState***(*PyTypeObject* *type)

属于稳定 ABI 自 *3.10* 版起. 返回关联到给定类型的模块对象的状态。这是一个在*PyType_GetModule()* 的结果上调用*PyModule_GetState()* 的快捷方式。

如果没有关联到给定类型的模块，则设置 `TypeError` 并返回 `NULL`。

如果 *type* 有关联的模块但其状态为 `NULL`，则返回 `NULL` 且不设置异常。

Added in version 3.9.

PyObject ***PyType_GetModuleByDef***(*PyTypeObject* *type, struct *PyModuleDef* *def)

返回值：借入的引用。属于稳定 ABI 自 *3.13* 版起. 找到所属模块基于给定的*PyModuleDef* *def* 创建的第一个上级类，并返回该模块。

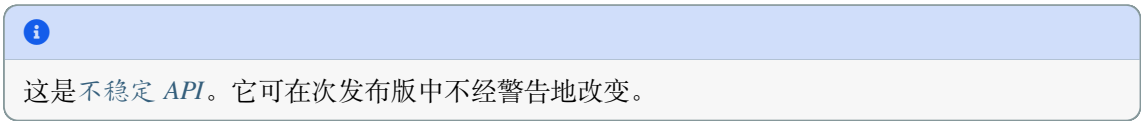
如果未找到模块，则会引发 `TypeError` 并返回 `NULL`。

此函数预期会与*PyModule_GetState()* 一起使用以便从槽位方法 (如*tp_init* 或*nb_add*) 及其他定义方法的类无法使用*PyCMethod* 调用惯例来传递的场合获取模块状态。

返回的引用是从 *type* 借入的，并且只要你还持有对 *type* 的引用就会保持有效。请不要通过*Py_DECREF()* 或类似函数来释放它。

Added in version 3.11.

int ***PyUnstable_Type_AssignVersionTag***(*PyTypeObject* *type)



这是不稳定 *API*。它可在次发布版中不经警告地改变。

尝试为给定的类型设置一个版本标签。

如果类型已有合法的版本标签或已设置了新的版本标签则返回 1，或者如果无法设置新的标签则返回 0。

Added in version 3.12.

创建堆分配类型

下列函数和结构体可被用来创建堆类型。

PyObject ***PyType_FromMetaclass***(*PyTypeObject* *metaclass, *PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

属于稳定 ABI 自 *3.12* 版起. 根据 *spec* (参见*Py_TPFLAGS_HEAPTYPE*) 创建并返回一个堆类型。

元类 *metaclass* 用于构建结果类型对象。当 *metaclass* 为 `NULL` 时，元类将派生自 *bases* (或者如果 *bases* 为 `NULL` 则派生自 *Py_tp_base[s]* 槽位，见下文)。

不支持重写*tp_new* 的元类，除非 `tp_new` 为 `NULL`。(为了向下兼容，其他 `PyType_From*` 函数允许这样的元类。它们将忽略 `tp_new`，可能导致不完整的初始化。这样的元类已被弃用并在 Python 3.14+ 中停止支持。)

bases 参数可被用来指定基类；它可以是单个类或由多个类组成的元组。如果 *bases* 为 `NULL`，则会改用 *Py_tp_bases* 槽位。如果该槽位也为 `NULL`，则会改用 *Py_tp_base* 槽位。如果该槽位同样为 `NULL`，则新类型将派生自 `object`。

module 参数可被用来记录新类定义所在的模块。它必须是一个模块对象或为 NULL。如果不为 NULL，则该模块会被关联到新类型并且可在之后通过*PyType_GetModule()* 来获取。这个关联模块不可被子类继承；它必须为每个类单独指定。

此函数会在新类型上调用*PyType_Ready()*。

请注意此函数 不能完全匹配调用 type() 或使用 class 语句的行为。对于用户提供的类型或元类，推荐调用 type (或元类) 而不是 PyType_From* 函数。特别地:

- __new__() 不会在新类上被调用 (它必须被设为 type.__new__)。
- __init__() 不会在新类上被调用。
- __init_subclass__() 不会在任何基类上调用。
- __set_name__() 不会在新的描述器上调用。

Added in version 3.12.

PyObject ***PyType_FromModuleAndSpec**(*PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

返回值：新的引用。属于稳定 ABI 自 *3.10* 版起. 等价于 PyType_FromMetaclass(NULL, module, spec, bases)。

Added in version 3.9.

在 3.10 版本发生变更: 此函数现在接受一个单独类作为 *bases* 参数并接受 NULL 作为 tp_doc 槽位。

在 3.12 版本发生变更: 该函数现在可以找到并使用与所提供的基类相对应的元类。在此之前，只会返回 type 实例。

元类的*tp_new* 将被 忽略。这可能导致不完整的初始化。创建元类重写*tp_new* 的类的做法已被弃用并且在 Python 3.14+ 中将不再被允许。

PyObject ***PyType_FromSpecWithBases**(*PyType_Spec* *spec, *PyObject* *bases)

返回值：新的引用。属于稳定 ABI 自 *3.3* 版起. 等价于 PyType_FromMetaclass(NULL, NULL, spec, bases)。

Added in version 3.3.

在 3.12 版本发生变更: 该函数现在可以找到并使用与所提供的基类相对应的元类。在此之前，只会返回 type 实例。

元类的*tp_new* 将被 忽略。这可能导致不完整的初始化。创建元类重写*tp_new* 的类的做法已被弃用并且在 Python 3.14+ 中将不再被允许。

PyObject ***PyType_FromSpec**(*PyType_Spec* *spec)

返回值：新的引用。属于稳定 ABI. 等价于 PyType_FromMetaclass(NULL, NULL, spec, NULL)。

在 3.12 版本发生变更: 该函数现在可以找到并使用与 *Py_tp_base[s]* 槽位中提供的基类相对应的元类。在此之前，只会返回 type 实例。

元类的*tp_new* 将被 忽略。这可能导致不完整的初始化。创建元类重写*tp_new* 的类的做法已被弃用并且在 Python 3.14+ 中将不再被允许。

type **PyType_Spec**

属于稳定 ABI （包括所有成员）. 定义一个类型的行为的结构体。

const char ***name**

类型的名称，用来设置*PyTypeObject.tp_name*。

int **basicsize**

如果为正数，则以字节为单位指定实例的大小。它用于设置*PyTypeObject.tp_basicsize*。

如果为零，则指定应当继承*tp_basicsize*。

如为负数，则以其绝对值指定该类的实例在超类的基础之上还需要多少空间。使用*PyObject_GetTypeData()* 来获取指向通过此方式保留的子类专属内存的指针。对于负的 basicsize，Python 将在需要满足*tp_basicsize* 的对齐要求时插入填充字节。

在 3.12 版本发生变更: 在之前版本中，此字段不能为负数。

int **itemsize**

可变大小类型中一个元素的大小，以字节为单位。用于设置*PyTypeObject.tp_itemsize*。注意事项请参阅 tp_itemsize 文档。

如果为零，则会继承*tp_itemsize*。扩展任意可变大小的类是很危险的，因为某些类型使用固定偏移量来标识可变大小的内存，这样就会与子类使用的固定大小的内存相重叠。为了防止出错，只有在以下情况下才可以继承 itemsize:

- 基类不是可变大小的 (即其*tp_itemsize*)。
- 所请求的*PyType_Spec.basicsize* 为正值，表明基类的内存布局是已知的。
- 所请求的*PyType_Spec.basicsize* 为零，表明子类不会直接访问实例的内存。
- 具有*Py_TPFLAGS_ITEMS_AT_END* 旗标。

unsigned int **flags**

类型旗标，用来设置*PyTypeObject.tp_flags*。

如果未设置 Py_TPFLAGS_HEAPTYPE 旗标，则*PyType_FromSpecWithBases()* 会自动设置它。

PyType_Slot ***slots**

PyType_Slot 结构体的数组。以特殊槽位值 {0, NULL} 来结束。

每个槽位 ID 应当只被指定一次。

type **PyType_Slot**

属于稳定 ABI （包括所有成员）. 定义一个类型的可选功能的结构体，包含一个槽位 ID 和一个值指针。

int **slot**

槽位 ID。

槽位 ID 的类名像是结构体*PyTypeObject*, *PyNumberMethods*, *PySequenceMethods*, *PyMappingMethods* 和*PyAsyncMethods* 的字段名附加一个 Py_ 前缀。举例来说，使用:

- Py_tp_dealloc 设置*PyTypeObject.tp_dealloc*
- Py_nb_add 设置*PyNumberMethods.nb_add*
- Py_sq_length 设置*PySequenceMethods.sq_length*

下列 “offset” 字段不可使用*PyType_Slot* 来设置:

- *tp_weaklistoffset* (如果可能请改用*Py_TPFLAGS_MANAGED_WEAKREF*)
- *tp_dictoffset* (如果可能请改用*Py_TPFLAGS_MANAGED_DICT*)
- *tp_vectorcall_offset* (请使用*PyMemberDef* 中的 "__vectorcalloffset__")

如果无法转为 MANAGED 旗标 (例如，对于 vectorcall 或是为了支持早于 Python 3.12 的版本)，请在*Py_tp_members* 中指定 offset。详情参见*PyMemberDef documentation*。

以下字段在创建堆类型时完全不可设置:

- *tp_vectorcall* (请使用*tp_new* 和/或*tp_init*)
- 内部字段: *tp_dict*, *tp_mro*, *tp_cache*, *tp_subclasses* 和*tp_weaklist*。

在某些平台上设置 Py_tp_bases 或 Py_tp_base 可能会有问题。为了避免问题，请改用*PyType_FromSpecWithBases()* 的 *bases* 参数。

在 3.9 版本发生变更: *PyBufferProcs* 中的槽位可能会在不受限 API 中被设置。

在 3.11 版本发生变更: 现在*bf_getbuffer* 和*bf_releasebuffer* 将在受限 *API* 中可用。

void ***pfunc**

该槽位的预期值。在大多数情况下，这将是一个指向函数的指针。

`Py_tp_doc` 以外的槽位均不可为 `NULL`。

8.1.2 `None` 对象

请注意，Python/C API 中并没有直接公开 `None` 的*`PyTypeObject`*。由于 `None` 是一个单例，测试对象标识号（在 C 语言中使用 `==` 运算符）就足够了。出于同样的原因也没有 `PyNone_Check()` 函数。

PyObject ***Py_None**

Python `None` 对象，表示缺少实际值。该对象没有任何方法并且属于*immortal* 对象。

在 3.12 版本发生变更: *`Py_None`* 属于*immortal* 对象。

Py_RETURN_NONE

从一个函数返回*`Py_None`*。

8.2 数值对象

8.2.1 整数型对象

所有整数都实现为长度任意的长整数对象。

在出错时，大多数 `PyLong_As*` API 都会返回 `(return type)-1`，这与数字无法区分开。请采用*`PyErr_Occurred()`* 来加以区分。

type **PyLongObject**

属于受限 API （作为不透明的结构体）. 表示 Python 整数对象的*`PyObject`* 子类型。

PyTypeObject **PyLong_Type**

属于稳定 ABI. 这个*`PyTypeObject`* 的实例表示 Python 的整数类型。与 Python 语言中的 `int` 相同。

int **PyLong_Check**(*PyObject* *p)

如果参数是*`PyLongObject`* 或*`PyLongObject`* 的子类型，则返回 True。该函数一定能够执行成功。

int **PyLong_CheckExact**(*PyObject* *p)

如果其参数属于*`PyLongObject`*，但不是*`PyLongObject`* 的子类型则返回真值。此函数总是会成功执行。

PyObject ***PyLong_FromLong**(long v)

返回值：新的引用。属于稳定 ABI. 由 *v* 返回一个新的*`PyLongObject`* 对象，失败时返回 `NULL` 。

当前的实现维护着一个整数对象数组，包含 `-5` 和 `256` 之间的所有整数对象。若创建一个位于该区间的 int 时，实际得到的将是对已有对象的引用。

PyObject ***PyLong_FromUnsignedLong**(unsigned long v)

返回值：新的引用。属于稳定 ABI. 基于 C `unsigned long` 返回一个新的*`PyLongObject`* 对象，失败时返回 `NULL`。

PyObject ***PyLong_FromSsize_t**(*Py_ssize_t* v)

返回值：新的引用。属于稳定 ABI. 由 C *`Py_ssize_t`* 返回一个新的*`PyLongObject`* 对象，失败时返回 `NULL` 。

PyObject ***PyLong_FromSize_t**(size_t v)

返回值：新的引用。属于稳定 ABI. 由 C `size_t` 返回一个新的*`PyLongObject`* 对象，失败则返回 `NULL` 。

PyObject ***PyLong_FromLongLong**(long long v)

返回值：新的引用。属于稳定 ABI. 基于 C `long long` 返回一个新的*`PyLongObject`*，失败时返回 `NULL`。

PyObject ***PyLong_FromUnsignedLongLong**(unsigned long long v)

返回值：新的引用。属于稳定 ABI. 基于 C `unsigned long long` 返回一个新的*PyLongObject* 对象，失败时返回 `NULL`。

PyObject ***PyLong_FromDouble**(double v)

返回值：新的引用。属于稳定 ABI. 由 *v* 的整数部分返回一个新的*PyLongObject* 对象，失败则返回 `NULL` 。

PyObject ***PyLong_FromString**(const char *str, char **pend, int base)

返回值：新的引用。属于稳定 ABI. 根据 *str* 字符串值返回一个新的*PyLongObject*，它将根据 *base* 指定的基数来解读，或是在失败时返回 `NULL`。如果 *pend* 不为 `NULL`，则在成功时 **pend* 将指向 *str* 中末尾而在出错时将指向第一个无法处理的字符。如果 *base* 为 `0`，则 *str* 将使用 integers 定义来解读；在此情况下，非零十进制数以零开头将会引发 `ValueError`。如果 *base* 不为 `0`，则必须在 `2` 和 `36`，包括这两个值。开头和末尾的空格以及基数标示符之后和数码之间的单下划线将被忽略。如果没有数码或 *str* 中数码和末尾空格之后不以 NULL 结束，则将引发 `ValueError`。

参见

Python 方法 `int.to_bytes()` 和 `int.from_bytes()` 用于*PyLongObject* 到/从字节数组之间以 `256` 为基数进行转换。你可以使用*PyObject_CallMethod()* 从 C 调用它们。

PyObject ***PyLong_FromUnicodeObject**(*PyObject* *u, int base)

返回值：新的引用。 将字符串 *u* 中的 Unicode 数字序列转换为 Python 整数值。

Added in version 3.3.

PyObject ***PyLong_FromVoidPtr**(void *p)

返回值：新的引用。属于稳定 ABI. 从指针 *p* 创建一个 Python 整数。可以使用*PyLong_AsVoidPtr()* 返回的指针值。

PyObject ***PyLong_FromNativeBytes**(const void *buffer, size_t n_bytes, int flags)

将包含在 *buffer* 开头 *n_bytes* 中的值解读为一个二的补码有符号数，基于它创建一个 Python 整数。

flags 与针对*PyLong_AsNativeBytes()* 的相同。传入 `-1` 将选择 CPython 编译时所用的原生端序并将假定主比特位是符号位。传入 `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` 将产生与调用*PyLong_FromUnsignedNativeBytes()* 相同的结果。其他旗标将被忽略。

Added in version 3.13.

PyObject ***PyLong_FromUnsignedNativeBytes**(const void *buffer, size_t n_bytes, int flags)

将包含在 *buffer* 开头 *n_bytes* 中的值解读为一个无符号数，基于它创建一个 Python 整数。

flags 与针对*PyLong_AsNativeBytes()* 的相同。传入 `-1` 将选择 CPython 编译时所用的原生端序并将假定主比特位不是符号位。其他旗标将被忽略。

Added in version 3.13.

long **PyLong_AsLong**(*PyObject* *obj)

属于稳定 ABI. 返回 *obj* 的 C `long` 表示形式。如果 *obj* 不是*PyLongObject* 的实例，则会先调用其 `__index__()` 方法（如果存在）将其转换为*PyLongObject*。

如果 *obj* 的值超出了 `long` 的取值范围则会引发 `OverflowError`。

出错则返回 `-1` 。请用*PyErr_Occurred()* 找出具体问题。

在 3.8 版本发生变更: 如果可能将使用 `__index__()`。

在 3.10 版本发生变更: 此函数将不再使用 `__int__()`。

long **PyLong_AS_LONG**(*PyObject* *obj)

一个已经*soft deprecated* 的别名。完全等价于推荐使用的 `PyLong_AsLong`。特别地，它可能因 `OverflowError` 或其他异常而失败。

自 3.14 版本弃用: 此函数已被软弃用。

int **PyLong_AsInt**(*PyObject* *obj)

属于稳定 ABI 自 *3.13* 版起. 类似于*PyLong_AsLong()*，将会将结果存放到一个 C `int` 而不是 C `long`。

Added in version 3.13.

long **PyLong_AsLongAndOverflow**(*PyObject* *obj, int *overflow)

属于稳定 ABI. 返回 *obj* 的 C `long` 表示形式。如果 *obj* 不是*PyLongObject* 的实例，则会先调用其 `__index__()` 方法（如果存在）将其转换为*PyLongObject*。

如果 *obj* 的值大于 `LONG_MAX` 或小于 `LONG_MIN`，则会把 **overflow* 分别置为 `1` 或 `-1`，并返回 `-1`；否则，将 **overflow* 置为 `0`。如果发生其他异常则按常规把 **overflow* 置为 `0` 并返回 `-1`。

出错则返回 `-1` 。请用*PyErr_Occurred()* 找出具体问题。

在 3.8 版本发生变更: 如果可能将使用 `__index__()`。

在 3.10 版本发生变更: 此函数将不再使用 `__int__()`。

long long **PyLong_AsLongLong**(*PyObject* *obj)

属于稳定 ABI. 返回 *obj* 的 C `long long` 表示形式。如果 *obj* 不是*PyLongObject* 的实例，则会先调用其 `__index__()` 方法（如果存在）将其转换为*PyLongObject*。

如果 *obj* 值超出 `long long` 的取值范围则会引发 `OverflowError`。

出错则返回 `-1` 。请用*PyErr_Occurred()* 找出具体问题。

在 3.8 版本发生变更: 如果可能将使用 `__index__()`。

在 3.10 版本发生变更: 此函数将不再使用 `__int__()`。

long long **PyLong_AsLongLongAndOverflow**(*PyObject* *obj, int *overflow)

属于稳定 ABI. 返回 *obj* 的 C `long long` 表示形式。如果 *obj* 不是*PyLongObject* 的实例，则会先调用其 `__index__()` 方法（如果存在）将其转换为*PyLongObject*。

如果 *obj* 的值大于 `LLONG_MAX` 或小于 `LLONG_MIN`，则会把 **overflow* 分别置为 `1` 或 `-1`，并返回 `-1`；否则，将 **overflow* 置为 `0`。如果发生其他异常则按常规把 **overflow* 置为 `0` 并返回 `-1`。

出错则返回 `-1` 。请用*PyErr_Occurred()* 找出具体问题。

Added in version 3.2.

在 3.8 版本发生变更: 如果可能将使用 `__index__()`。

在 3.10 版本发生变更: 此函数将不再使用 `__int__()`。

Py_ssize_t **PyLong_AsSsize_t**(*PyObject* *pylong)

属于稳定 ABI. 返回 *pylong* 的 C 语言*Py_ssize_t* 形式。*pylong* 必须是*PyLongObject* 的实例。

如果 *pylong* 的值超出了*Py_ssize_t* 的取值范围则会引发 `OverflowError`。

出错则返回 `-1` 。请用*PyErr_Occurred()* 找出具体问题。

unsigned long **PyLong_AsUnsignedLong**(*PyObject* *pylong)

属于稳定 ABI. 返回 *pylong* 的 C `unsigned long` 表示形式。*pylong* 必须是*PyLongObject* 的实例。

如果 *pylong* 的值超出了 `unsigned long` 的取值范围则会引发 `OverflowError`。

出错时返回 `(unsigned long)-1` ，请利用*PyErr_Occurred()* 辨别具体问题。

size_t **PyLong_AsSize_t**(*PyObject* *pylong)

属于稳定 ABI. 返回 *pylong* 的 C 语言 `size_t` 形式。*pylong* 必须是*PyLongObject* 的实例。

如果 *pylong* 的值超出了 `size_t` 的取值范围则会引发 `OverflowError`。

出错时返回 `(size_t)-1` ，请利用*PyErr_Occurred()* 辨别具体问题。

unsigned long long **PyLong_AsUnsignedLongLong**(*PyObject* *pylong)

属于稳定 ABI. 返回 *pylong* 的 C `unsigned long long` 表示形式。*pylong* 必须是*PyLongObject* 的实例。

如果 *pylong* 的值超出 `unsigned long long` 的取值范围则会引发 `OverflowError`。

出错时返回 `(unsigned long long)-1`，请利用*PyErr_Occurred()* 辨别具体问题。

在 3.1 版本发生变更: 现在 *pylong* 为负值会触发 `OverflowError`，而不是 `TypeError`。

unsigned long **PyLong_AsUnsignedLongMask**(*PyObject* *obj)

属于稳定 ABI. 返回 *obj* 的 C `unsigned long` 表示形式。如果 *obj* 不是*PyLongObject* 的实例，则会先调用其 `__index__()` 方法（如果存在）将其转换为*PyLongObject*。

如果 *obj* 的值超出了 `unsigned long` 的取值范围，则返回该值对 `ULONG_MAX + 1` 求模的余数。

出错时返回 `(unsigned long)-1`，请利用*PyErr_Occurred()* 辨别具体问题。

在 3.8 版本发生变更: 如果可能将使用 `__index__()`。

在 3.10 版本发生变更: 此函数将不再使用 `__int__()`。

unsigned long long **PyLong_AsUnsignedLongLongMask**(*PyObject* *obj)

属于稳定 ABI. 返回 *obj* 的 C `unsigned long long` 表示形式。如果 *obj* 不是*PyLongObject* 的实例，则会先调用其 `__index__()` 方法（如果存在）将其转换为*PyLongObject*。

如果 *obj* 的值超出了 `unsigned long long` 的取值范围，则返回该值对 `ULLONG_MAX + 1` 求模的余数。

出错时返回 `(unsigned long long)-1`，请利用*PyErr_Occurred()* 辨别具体问题。

在 3.8 版本发生变更: 如果可能将使用 `__index__()`。

在 3.10 版本发生变更: 此函数将不再使用 `__int__()`。

double **PyLong_AsDouble**(*PyObject* *pylong)

属于稳定 ABI. 返回 *pylong* 的 C `double` 表示形式。*pylong* 必须是*PyLongObject* 的实例。

如果 *pylong* 的值超出了 `double` 的取值范围则会引发 `OverflowError`。

出错时返回 `-1.0`，请利用*PyErr_Occurred()* 辨别具体问题。

void ***PyLong_AsVoidPtr**(*PyObject* *pylong)

属于稳定 ABI. 将一个 Python 整数 *pylong* 转换为 C `void` 指针。如果 *pylong* 无法被转换，则将引发 `OverflowError`。这只是为了保证将通过*PyLong_FromVoidPtr()* 创建的值产生一个可用的 `void` 指针。

出错时返回 `NULL`，请利用*PyErr_Occurred()* 辨别具体问题。

Py_ssize_t **PyLong_AsNativeBytes**(*PyObject* *pylong, void *buffer, *Py_ssize_t* n_bytes, int flags)

将 Python 整数值 *pylong* 拷贝到一个大小为 *n_bytes* 的原生 *buffer* 中。*flags* 可设为 `-1` 以使其行为类似于 C 强制转换类型，或是用下文中的值来控制其行为。

当发生错误时将返回 `-1` 并设置一个异常。如果 *pylong* 无法被解读为一个整数，或者如果 *pylong* 为负值并且设置了 `Py_ASNATIVEBYTES_REJECT_NEGATIVE` 旗标就可能出现这种情况。

在其他情况下，将返回存储该值所需要的字节数量。如果该数量小于等于 *n_bytes*，则将拷贝整个值。缓冲区的 *n_bytes* 将全部被写入：更大的缓冲区将以零值填充。

如果返回的值大于 *n_bytes*，该值将被截断：从该值的低位开始写入尽可能多的数位，而高位部分将被忽略。这与典型的 C 网络向下强制转换行为相匹配。

> **备注**
>
> 溢出不会被视为错误。如果返回值大于 *n_bytes*，高位部分将被丢弃。

绝对不会返回 0。

值将总是作为二的补码被拷贝。

用法示例:

```
int32_t value;
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, &value, sizeof(value), -1);
if (bytes < 0) {
    // 失败。 设置一个提示失败原因的 Python 异常。
    return NULL;
}
else if (bytes <= (Py_ssize_t)sizeof(value)) {
    // 成功!
}
else {
    // 发生溢出，但 'value' 包含了截断后的
    // pylong 的低比特位部分。
}
```

将零值传给 *n_bytes* 将返回一个足够容纳该值的缓冲区大小。这可能会大于基于技术考虑所需要的值，但也不会过于离谱。如果 *n_bytes=0*，则 *buffer* 可能为 NULL。

备注

将 *n_bytes=0* 传给此函数不是确定值所需比特位长度的准确方式。

要获取一个大小未知的完整 Python 值，可以调用此函数两次：首先确定缓冲区大小，然后填充它:

```
// 询问我们需要多少空间。
Py_ssize_t expected = PyLong_AsNativeBytes(pylong, NULL, 0, -1);
if (expected < 0) {
    // 失败。Python 已设置异常，并提供了原因
    return NULL;
}
assert(expected != 0);  // 根据 API 定义，不可能为 0
uint8_t *bignum = malloc(expected);
if (!bignum) {
    PyErr_SetString(PyExc_MemoryError, "bignum malloc failed.");
    return NULL;
}
// 安全地获取整个值
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, bignum, expected, -1);
if (bytes < 0) {  // 已设置异常
    free(bignum);
    return NULL;
}
else if (bytes > expected) {  // 这种情况不应该发生
    PyErr_SetString(PyExc_RuntimeError,
        "Unexpected bignum truncation after a size check.");
    free(bignum);
    return NULL;
}
// 上述预检查成功的预期情况
// ... 使用 bignum ...
free(bignum);
```

flags 可以是 -1 (Py_ASNATIVEBYTES_DEFAULTS) 表示选择最接近 C 强制转换类型的默认行为，或是下表中其他旗标的组合。请注意 -1 不能与其他旗标组合使用。

目前，-1 对应于 Py_ASNATIVEBYTES_NATIVE_ENDIAN | Py_ASNATIVEBYTES_UNSIGNED_BUFFER。

标志位	值
Py_ASNATIVEBYTES_DEFAULTS	-1
Py_ASNATIVEBYTES_BIG_ENDIAN	0
Py_ASNATIVEBYTES_LITTLE_ENDIAN	1
Py_ASNATIVEBYTES_NATIVE_ENDIAN	3
Py_ASNATIVEBYTES_UNSIGNED_BUFFER	4
Py_ASNATIVEBYTES_REJECT_NEGATIVE	8
Py_ASNATIVEBYTES_ALLOW_INDEX	16

指定 `Py_ASNATIVEBYTES_NATIVE_ENDIAN` 将覆盖任何其他端序旗标。传入 `2` 被保留用于后续版本。

在默认情况下，将会请求足够的缓冲区以包括符号位。例如在设置 *n_bytes=1* 转换 128 时，该函数将返回 2 (或更大的值) 以存储一个零值符号位。

如果指定了 `Py_ASNATIVEBYTES_UNSIGNED_BUFFER`，将在计算大小时忽略零值符号位。例如，这将允许将 128 放入一个单字节缓冲区。如果目标缓冲区随后又被当作是带符号位的，则一个正数输入值可能会变成负值。请注意此旗标不会影响对负值的处理：对于这种情况，总是会请求用作符号位的空间。

指定 `Py_ASNATIVEBYTES_REJECT_NEGATIVE` 将导致当 *pylong* 为负值时设置一个异常。如果没有此旗标，只要至少有足够容纳一个符号位的空间就将拷贝负值，无论是否指定了 `Py_ASNATIVEBYTES_UNSIGNED_BUFFER`。

如果指定了 `Py_ASNATIVEBYTES_ALLOW_INDEX` 并且传入一个非整数值，则会先调用其 `__index__()` 方法。这可能导致 Python 代码执行并允许运行其他线程，在此情况下将会改变其他正在使用的对象或值。当 *flags* 为 `-1` 时，则不设置此选项，而非整数值将会引发 `TypeError`。

> **备注**
>
> 如果使用默认 *flags* (`-1` 或不带 *REJECT_NEGATIVE* 的 *UNSIGNED_BUFFER*)，则多个 Python 整数可映射为单个值而不会溢出。例如，`255` 和 `-1` 都可放入一个单字节缓冲区并设置其全部比特位。这与典型的 C 强制转换行为相匹配。

Added in version 3.13.

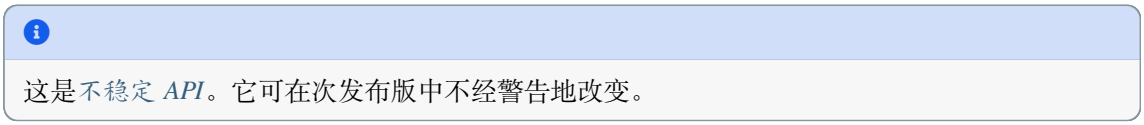
PyObject ***PyLong_GetInfo**(void)

属于稳定 ABI. 成功时，返回一个只读的*named tuple*，它保存着有关 Python 内部整数表示形式的信息。请参阅 `sys.int_info` 了解关于单独字段的描述。

当失败时，将返回 `NULL` 并设置一个异常。

Added in version 3.1.

int **PyUnstable_Long_IsCompact**(const *PyLongObject* *op)



这是不稳定 API。它可在次发布版中不经警告地改变。

如果 *op* 为紧凑形式则返回 1，否则返回 0。

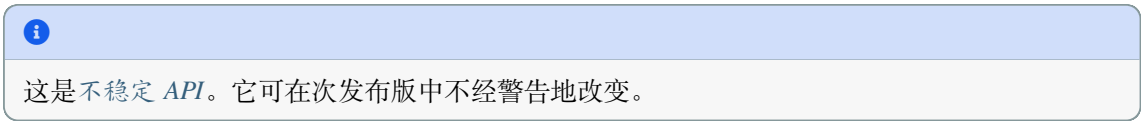
此函数使得注重性能的代码可以实现小整数的“快速路径”。对于紧凑值将使用 `PyUnstable_Long_CompactValue()`；对于其他值则回退为 `PyLong_As*` 函数或者 `PyLong_AsNativeBytes()`。

此项加速对于大多数用户来说是可以忽略的。

具体有哪些值会被视为紧凑形式属于实现细节并可能发生改变。

Added in version 3.12.

Py_ssize_t **PyUnstable_Long_CompactValue**(const *PyLongObject* *op)



这是不稳定 API。它可在次发布版中不经警告地改变。

如果 *op* 为紧凑形式，如 `PyUnstable_Long_IsCompact()` 所确定的，则返回它的值。

在其他情况下，返回值是未定义的。

Added in version 3.12.

8.2.2 布尔对象

在 Python 中布尔值是作为整数的子类实现的。只有两个布尔值，`Py_False` 和 `Py_True`。因此，正常的创建和删除功能不适用于布尔值。不过，下列的宏则是可用的。

PyTypeObject **PyBool_Type**

属于稳定 ABI. 这个 `PyTypeObject` 的实例代表一个 Python 布尔类型；它与 Python 层面的 `bool` 是相同的对象。

int **PyBool_Check**(*PyObject* *o)

如果 *o* 的类型为 `PyBool_Type` 则返回真值。此函数总是会成功执行。

PyObject ***Py_False**

Python `False` 对象。该对象没有任何方法并且属于*immortal* 对象。

在 3.12 版本发生变更: `Py_False` 属于*immortal* 对象。

PyObject ***Py_True**

Python `True` 对象。该对象没有任何方法并且属于*immortal* 对象。

在 3.12 版本发生变更: `Py_True` 属于*immortal* 对象。

Py_RETURN_FALSE

从一个函数返回 `Py_False`。

Py_RETURN_TRUE

从一个函数返回 `Py_True`。

PyObject ***PyBool_FromLong**(long v)

返回值：新的引用。属于稳定 ABI. 返回 `Py_True` 或 `Py_False`，具体取决于 *v* 的逻辑值。

8.2.3 浮点数对象

type **PyFloatObject**

这个*PyObject* 的子类型代表一个 Python 浮点数对象。

PyTypeObject **PyFloat_Type**

属于稳定 ABI. 这个*PyTypeObject* 实例代表 Python 浮点数类型。这与 Python 层面的 `float` 是同一个对象。

int **PyFloat_Check**(*PyObject* *p)

如果它的参数是一个*PyFloatObject* 或者*PyFloatObject* 的子类型则返回真值。此函数总是会成功执行。

int **PyFloat_CheckExact**(*PyObject* *p)

如果它的参数是一个*PyFloatObject* 但不是*PyFloatObject* 的子类型则返回真值。此函数总是会成功执行。

PyObject ***PyFloat_FromString**(*PyObject* *str)

返回值：新的引用。属于稳定 ABI. 根据字符串 *str* 的值创建一个*PyFloatObject*，失败时返回 `NULL`。

PyObject ***PyFloat_FromDouble**(double v)

返回值：新的引用。属于稳定 ABI. 根据 *v* 创建一个*PyFloatObject* 对象，失败时返回 `NULL`。

double **PyFloat_AsDouble**(*PyObject* *pyfloat)

属于稳定 ABI. 返回 *pyfloat* 的内容的 C `double` 表示形式。如果 *pyfloat* 不是一个 Python 浮点数对象但是具有 `__float__()` 方法，则会先调用此方法来将 *pyfloat* 转换为浮点数。如果 `__float__()` 未定义则将回退至 `__index__()`。此方法在失败时将返回 `-1.0`，因此开发者应当调用*PyErr_Occurred()* 来检测错误。

在 3.8 版本发生变更: 如果可能将使用 `__index__()`。

double **PyFloat_AS_DOUBLE**(*PyObject* *pyfloat)

返回 *pyfloat* 的 C `double` 表示形式，但不带错误检测。

PyObject ***PyFloat_GetInfo**(void)

返回值：新的引用。属于稳定 ABI. 返回一个 structseq 实例，其中包含有关 float 的精度、最小值和最大值的信息。它是头文件 `float.h` 的一个简单包装。

double **PyFloat_GetMax**()

属于稳定 ABI. 返回 C `double` 形式的最大可表示有限浮点数 *DBL_MAX*。

double **PyFloat_GetMin**()

属于稳定 ABI. 返回 C `double` 形式的最小正规化正浮点数 *DBL_MIN*。

打包与解包函数

打包与解包函数提供了独立于平台的高效方式来将浮点数值存储为字节串。Pack 例程根据 C `double` 产生字节串，而 Unpack 例程根据这样的字节串产生 C `double`。后缀 (2, 4 or 8) 指明字节串中的字节数。

在明显使用 IEEE 754 格式的平台上这些函数是通过拷贝比特位来实现的。在其他平台上，2 字节格式与 IEEE 754 binary16 半精度格式相同，4 字节格式 (32 位) 与 IEEE 754 binary32 单精度格式相同，而 8 字节格式则与 IEEE 754 双精度格式相同，不过 INF 和 NaN (如果平台存在这两种值) 未得到正确处理，而试图对包含 IEEE INF 或 NaN 的字节串执行解包将会引发一个异常。

在具有比 IEEE 754 所支持的更高精度，或更大动态范围的非 IEEE 平台上，不是所有的值都能被打包；在具有更低精度，或更小动态范围的非 IEEE 平台上，则不是所有的值都能被解包。在这种情况下发生的事情有一部分将是偶然的（无奈）。

Added in version 3.11.

打包函数

打包例程会写入 2, 4 或 8 个字节，从 *p* 开始。*le* 是一个 `int` 参数，如果你想要字节串为小端序格式 (指数部分放在后面，位于 `p+1`, `p+3` 或 `p+6` `p+7`) 则其应为非零值，如果你想要大端序格式 (指数部分放在前面，位于 *p*) 则其应为零。`PY_BIG_ENDIAN` 常量可被用于使用本机端序：在大端序处理器上等于 `1`，在小端序处理器上则等于 `0`。

返回值: 如果一切正常则为 `0`，如果出错则为 `-1` (并会设置一个异常，最大可能为 `OverflowError`)。

在非 IEEE 平台上存在两个问题:

- 如果 *x* 为 NaN 或无穷大则此函数的行为是未定义的。
- `-0.0` 和 `+0.0` 将产生相同的字节串。

int **PyFloat_Pack2**(double x, char *p, int le)

将 C double 打包为 IEEE 754 binary16 半精度格式。

int **PyFloat_Pack4**(double x, char *p, int le)

将 C double 打包为 IEEE 754 binary32 单精度格式。

int **PyFloat_Pack8**(double x, char *p, int le)

将 C double 打包为 IEEE 754 binary64 双精度格式。

解包函数

解包例程会读取 2, 4 或 8 个字节，从 *p* 开始。*le* 是一个 `int` 参数，如果字节串为小端序格式 (指数部门放在后面，位于 `p+1`, `p+3` 或 `p+6` 和 `p+7`) 则其应为非零值，如果为大端序格式 (指数部分放在前面，位于 *p*) 则其应为零。`PY_BIG_ENDIAN` 常量可被用于使用本机端序：在大端序处理器上等于 `1`，在小端序处理器上则等于 `0`。

返回值: 解包后的 double。出错时，返回值为 `-1.0` 且*PyErr_Occurred()* 为真值 (并且会设置一个异常，最大可能为 `OverflowError`)。

请注意在非 IEEE 平台上此函数将拒绝解包表示 NaN 或无穷大的字节串。

double **PyFloat_Unpack2**(const char *p, int le)

将 IEEE 754 binary16 半精度格式解包为 C double。

double **PyFloat_Unpack4**(const char *p, int le)

将 IEEE 754 binary32 单精度格式解包为 C double。

double **PyFloat_Unpack8**(const char *p, int le)

将 IEEE 754 binary64 双精度格式解包为 C double。

8.2.4 复数对象

从 C API 看，Python 的复数对象由两个不同的部分实现：一个是在 Python 程序使用的 Python 对象，另外的是一个代表真正复数值的 C 结构体。API 提供了函数共同操作两者。

表示复数的 C 结构体

需要注意的是接受这些结构体的作为参数并当做结果返回的函数，都是传递“值”而不是引用指针。此规则适用于整个 API。

type **Py_complex**

对应于 Python 复数对象的值部分的 C 结构体。大部分用于处理数据对象的函数都使用该类型的结构体作为相应的输入或输出值。

double **real**

double **imag**

其结构定义如下:

```
typedef struct {
    double real;
    double imag;
} Py_complex;
```

Py_complex **_Py_c_sum**(*Py_complex* left, *Py_complex* right)

返回两个复数的和，用 C 类型*Py_complex* 表示。

Py_complex **_Py_c_diff**(*Py_complex* left, *Py_complex* right)

返回两个复数的差，用 C 类型*Py_complex* 表示。

Py_complex **_Py_c_neg**(*Py_complex* num)

返回复数 *num* 的负值，用 C *Py_complex* 表示。

Py_complex **_Py_c_prod**(*Py_complex* left, *Py_complex* right)

返回两个复数的乘积，用 C 类型*Py_complex* 表示。

Py_complex **_Py_c_quot**(*Py_complex* dividend, *Py_complex* divisor)

返回两个复数的商，用 C 类型*Py_complex* 表示。

如果 *divisor* 为空，则此方法将返回零并将 `errno` 设为 `EDOM`。

Py_complex **_Py_c_pow**(*Py_complex* num, *Py_complex* exp)

返回 *num* 的 *exp* 次幂，用 C 类型*Py_complex* 表示。

如果 *num* 为空且 *exp* 不是正实数，则此方法将返回零并将 `errno` 设为 `EDOM`。

表示复数的 Python 对象

type **PyComplexObject**

这个 C 类型*PyObject* 的子类型代表一个 Python 复数对象。

PyTypeObject **PyComplex_Type**

属于稳定 ABI. 这是个属于*PyTypeObject* 的代表 Python 复数类型的实例。在 Python 层面的类型 `complex` 是同一个对象。

int **PyComplex_Check**(*PyObject* *p)

如果它的参数是一个*PyComplexObject* 或者*PyComplexObject* 的子类型则返回真值。此函数总是会成功执行。

int **PyComplex_CheckExact**(*PyObject* *p)

如果它的参数是一个*PyComplexObject* 但不是*PyComplexObject* 的子类型则返回真值。此函数总是会成功执行。

PyObject ***PyComplex_FromCComplex**(*Py_complex* v)

返回值：新的引用。 根据一个 C *Py_complex* 值新建 Python 复数对象。当发生错误时将返回 `NULL` 并设置一个异常。

PyObject ***PyComplex_FromDoubles**(double real, double imag)

返回值：新的引用。属于稳定 ABI. 根据 *real* 和 *imag* 返回一个新的*PyComplexObject* 对象。当发生错误时将返回 `NULL` 并设置一个异常。

double **PyComplex_RealAsDouble**(*PyObject* *op)

属于稳定 ABI. 以 C 类型 `double` 返回 *op* 的实部。

如果 *op* 不是一个 Python 复数对象但是具有 `__complex__()` 方法，则会先调用该方法将 *op* 转换为 Python 复数对象。如果未定义 `__complex__()` 则将回退为调用*PyFloat_AsDouble()* 并返回其结果。

当失败时，此方法将返回 `-1.0` 并设置一个异常，因此开发者应当调用*PyErr_Occurred()* 来检查错误。

在 3.13 版本发生变更: 如果可能将使用 `__complex__()`。

double **PyComplex_ImagAsDouble**(*PyObject* *op)

属于稳定 ABI. 以 C 类型 double 返回 *op* 的虚部。

如果 *op* 不是一个 Python 复数对象但是具有 __complex__() 方法，则会先调用该方法将 *op* 转换为 Python 复数对象。如果未定义 __complex__() 则将回退为调用*PyFloat_AsDouble()* 并在成功时返回 0.0。

当失败时，此方法将返回 -1.0 并设置一个异常，因此开发者应当调用*PyErr_Occurred()* 来检查错误。

在 3.13 版本发生变更: 如果可能将使用 __complex__()。

Py_complex **PyComplex_AsCComplex**(*PyObject* *op)

返回复数 *op* 的 C 类型*Py_complex* 值。

如果 *op* 不是一个 Python 复数对象但是具有 __complex__() 方法，则会先调用该方法将 *op* 转换为 Python 复数对象。如果未定义 __complex__() 则将回退至 __float__()。如果未定义 __float__() 则将回退至 __index__()。

当失败时，此方法将返回*Py_complex* 其中*real* 为 -1.0 并设置一个异常，因此开发者应当调用*PyErr_Occurred()* 来检查错误。

在 3.8 版本发生变更: 如果可能将使用 __index__()。

8.3 序列对象

序列对象的一般操作在前一章中讨论过; 本节介绍 Python 语言固有的特定类型的序列对象。

8.3.1 bytes 对象

这些函数在期望附带一个字节串形参但却附带了一个非字节串形参被调用时会引发 TypeError。

type **PyBytesObject**

这种*PyObject* 的子类型表示一个 Python 字节对象。

PyTypeObject **PyBytes_Type**

属于稳定 ABI. *PyTypeObject* 的实例代表一个 Python 字节类型，在 Python 层面它与 bytes 是相同的对象。

int **PyBytes_Check**(*PyObject* *o)

如果对象 *o* 是一个 bytes 对象或者 bytes 类型的子类型的实例则返回真值。此函数总是会成功执行。

int **PyBytes_CheckExact**(*PyObject* *o)

如果对象 *o* 是一个 bytes 对象但不是 bytes 类型的子类型的实例则返回真值。此函数总是会成功执行。

PyObject ***PyBytes_FromString**(const char *v)

返回值：新的引用。属于稳定 ABI. 成功时返回一个以字符串 *v* 的副本为值的新字节串对象，失败时返回 NULL。形参 *v* 不可为 NULL；它不会被检查。

PyObject ***PyBytes_FromStringAndSize**(const char *v, *Py_ssize_t* len)

返回值：新的引用。属于稳定 ABI. 成功时返回一个以字符串 *v* 的副本为值且长度为 *len* 的新字节串对象，失败时返回 NULL。如果 *v* 为 NULL，则不初始化字节串对象的内容。

PyObject ***PyBytes_FromFormat**(const char *format, ...)

返回值：新的引用。属于稳定 ABI. 接受一个 C printf() 风格的 *format* 字符串和可变数量的参数，计算结果 Python 字节串对象的大小并返回参数值经格式化后的字节串对象。可变数量的参数必须均为 C 类型并且必须恰好与 *format* 字符串中的格式字符相对应。允许使用下列格式字符串:

格式字符	类型	注释
`%%`	不适用	文字% 字符。
`%c`	int	一个字节，被表示为一个 C 语言的整型
`%d`	int	相当于 `printf("%d")`.[1]
`%u`	unsigned int	相当于 `printf("%u")`.[1]
`%ld`	长整型	相当于 `printf("%ld")`.[1]
`%lu`	unsigned long	相当于 `printf("%lu")`.[1]
`%zd`	*Py_ssize_t*	相当于 `printf("%zd")`.[1]
`%zu`	size_t	相当于 `printf("%zu")`.[1]
`%i`	int	相当于 `printf("%i")`.[1]
`%x`	int	相当于 `printf("%x")`.[1]
`%s`	const char*	以 null 为终止符的 C 字符数组。
`%p`	const void*	一个 C 指针的十六进制表示形式。基本等价于 `printf("%p")` 但它会确保以字面值 `0x` 开头，不论系统平台上 `printf` 的输出是什么。

无法识别的格式字符会导致将格式字符串的其余所有内容原样复制到结果对象，并丢弃所有多余的参数。

PyObject ***PyBytes_FromFormatV**(const char *format, va_list vargs)

返回值：新的引用。属于稳定 ABI. 与*PyBytes_FromFormat()* 完全相同，除了它需要两个参数。

PyObject ***PyBytes_FromObject**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 返回字节表示实现缓冲区协议的对象 *o*。

Py_ssize_t **PyBytes_Size**(*PyObject* *o)

属于稳定 ABI. 返回字节对象 *o* 中字节的长度。

Py_ssize_t **PyBytes_GET_SIZE**(*PyObject* *o)

类似于*PyBytes_Size()*，但是不带错误检测。

char ***PyBytes_AsString**(*PyObject* *o)

属于稳定 ABI. 返回对应 *o* 的内容的指针。该指针指向 *o* 的内部缓冲区，其中包含 `len(o) + 1` 个字节。缓冲区的最后一个字节总是为空，不论是否存在其他空字节。该数据不可通过任何形式来修改，除非是刚使用 `PyBytes_FromStringAndSize(NULL, size)` 创建该对象。它不可被撤销分配。如果 *o* 根本不是一个字节串对象，则*PyBytes_AsString()* 将返回 `NULL` 并引发 `TypeError`。

char ***PyBytes_AS_STRING**(*PyObject* *string)

类似于*PyBytes_AsString()*，但是不带错误检测。

int **PyBytes_AsStringAndSize**(*PyObject* *obj, char **buffer, *Py_ssize_t* *length)

属于稳定 ABI. 通过输出变量 *buffer* 和 *length* 返回对象 *obj* 以空值作为结束的内容。成功时返回 `0`。

如果 *length* 为 `NULL`，字节串对象就不包含嵌入的空字节；如果包含，则该函数将返回 `-1` 并引发 `ValueError`。

该缓冲区指向 *obj* 的内部缓冲，它的末尾包含一个额外的空字节（不算在 *length* 当中）。该数据不可通过任何方式来修改，除非是刚使用 `PyBytes_FromStringAndSize(NULL, size)` 创建该对象。它不可被撤销分配。如果 *obj* 根本不是一个字节串对象，则*PyBytes_AsStringAndSize()* 将返回 `-1` 并引发 `TypeError`。

在 3.5 版本发生变更: 以前，当字节串对象中出现嵌入的空字节时将引发 `TypeError`。

void **PyBytes_Concat**(*PyObject* **bytes, *PyObject* *newpart)

属于稳定 ABI. 在 **bytes* 中创建新的字节串对象，其中包含添加到 *bytes* 的 *newpart* 的内容；调用者将获得新的引用。对 *bytes* 原值的引用将被收回。如果无法创建新对象，对 *bytes* 的旧引用仍将被丢弃且 **bytes* 的值将被设为 `NULL`；并将设置适当的异常。

[1] 对于整数说明符（d，u，ld，lu，zd，zu，i，x）：当给出精度时，0 转换标志是有效的。

void **PyBytes_ConcatAndDel**(*PyObject* **bytes, *PyObject* *newpart)

属于稳定 ABI. 在 **bytes* 中创建一个新的字节串对象，其中包含添加到 *bytes* 的 *newpart* 的内容。此版本将释放对 *newpart* 的*strong reference* (即递减其引用计数)。

int **_PyBytes_Resize**(*PyObject* **bytes, *Py_ssize_t* newsize)

改变字节串对象的大小。*newsize* 将为字节串对象的新长度。你可以将它当作是创建一个新的字节串对象并销毁旧的对象，但是更为高效。传入一个现有字节串对象的地址作为 lvalue（它可以被写入），以及想要的新大小。当成功时，**bytes* 将存放改变大小后的字节串对象并返回 `0`；**bytes* 中的地址可能与其输入值不同。如果重新分配失败，则位于 **bytes* 的原始字节串对象将被释放，**bytes* 将被设为 `NULL`，同时设置 `MemoryError`，然后返回 `-1`。

8.3.2 字节数组对象

type **PyByteArrayObject**

这个`PyObject` 的子类型表示一个 Python 字节数组对象。

PyTypeObject **PyByteArray_Type**

属于稳定 ABI. Python bytearray 类型表示为`PyTypeObject` 的实例；这与 Python 层面的 `bytearray` 是相同的对象。

类型检查宏

int **PyByteArray_Check**(*PyObject* *o)

如果对象 *o* 是一个 bytearray 对象或者 bytearray 类型的子类型的实例则返回真值。此函数总是会成功执行。

int **PyByteArray_CheckExact**(*PyObject* *o)

如果对象 *o* 是一个 bytearray 对象但不是 bytearray 类型的子类型的实例则返回真值。此函数总是会成功执行。

直接 API 函数

PyObject ***PyByteArray_FromObject**(*PyObject* *o)

返回值：新的引用。属于稳定 ABI. 根据任何实现了缓冲区协议 的对象 *o*，返回一个新的字节数组对象。

当失败时，将返回 `NULL` 并设置一个异常。

PyObject ***PyByteArray_FromStringAndSize**(const char *string, *Py_ssize_t* len)

返回值：新的引用。属于稳定 ABI. 根据 *string* 和它的长度 *len* 新建一个字节数组对象。

当失败时，将返回 `NULL` 并设置一个异常。

PyObject ***PyByteArray_Concat**(*PyObject* *a, *PyObject* *b)

返回值：新的引用。属于稳定 ABI. 连接字节数组 *a* 和 *b* 并返回一个带有结果的新的字节数组。

当失败时，将返回 `NULL` 并设置一个异常。

Py_ssize_t **PyByteArray_Size**(*PyObject* *bytearray)

属于稳定 ABI. 在检查 `NULL` 指针后返回 *bytearray* 的大小。

char ***PyByteArray_AsString**(*PyObject* *bytearray)

属于稳定 ABI. 在检查 `NULL` 指针后返回将 *bytearray* 返回为一个字符数组。返回的数组总是会附加一个额外的空字节。

int **PyByteArray_Resize**(*PyObject* *bytearray, *Py_ssize_t* len)

属于稳定 ABI. 将 *bytearray* 的内部缓冲区的大小调整为 *len*。

宏

这些宏减低安全性以换取性能，它们不检查指针。

char ***PyByteArray_AS_STRING**(*PyObject* *bytearray)
: 类似于*PyByteArray_AsString()*，但是不带错误检测。

Py_ssize_t **PyByteArray_GET_SIZE**(*PyObject* *bytearray)
: 类似于*PyByteArray_Size()*，但是不带错误检测。

8.3.3 Unicode 对象和编解码器

Unicode 对象

自从 python3.3 中实现了 **PEP 393** 以来，Unicode 对象在内部使用各种表示形式，以便在保持内存效率的同时处理完整范围的 Unicode 字符。对于所有代码点都低于 128、256 或 65536 的字符串，有一些特殊情况；否则，代码点必须低于 1114112（这是完整的 Unicode 范围）。

UTF-8 表示将按需创建并缓存在 Unicode 对象中。

备注

Py_UNICODE 表示形式在 Python 3.12 中同被弃用的 API 一起被移除了，查阅 **PEP 623** 以获得更多信息。

Unicode 类型

以下是用于 Python 中 Unicode 实现的基本 Unicode 对象类型：

type **Py_UCS4**

type **Py_UCS2**

type **Py_UCS1**
: 属于稳定 ABI. 这些类型是无符号整数类型的类型定义，其宽度足以分别包含 32 位、16 位和 8 位字符。当需要处理单个 Unicode 字符时，请使用*Py_UCS4*。

  Added in version 3.3.

type **Py_UNICODE**
: 这是 `wchar_t` 的类型定义，根据平台的不同它可能为 16 位类型或 32 位类型。

  在 3.3 版本发生变更: 在以前的版本中，这是 16 位类型还是 32 位类型，这取决于您在构建时选择的是“窄”还是“宽”Unicode 版本的 Python。

  Deprecated since version 3.13, will be removed in version 3.15.

type **PyASCIIObject**

type **PyCompactUnicodeObject**

type **PyUnicodeObject**
: 这些关于*PyObject* 的子类型表示了一个 Python Unicode 对象。在几乎所有情形下，它们不应该被直接使用，因为所有处理 Unicode 对象的 API 函数都接受并返回*PyObject* 类型的指针。

  Added in version 3.3.

PyTypeObject **PyUnicode_Type**
: 属于稳定 ABI. 这个*PyTypeObject* 实例代表 Python Unicode 类型。它以 `str` 的形式暴露给 Python 代码。

PyTypeObject **PyUnicodeIter_Type**
: 属于稳定 ABI. 这是*PyTypeObject* 实例代表 Python Unicode 迭代器类型。它被用来迭代 Unicode 字符串对象。

以下 API 是 C 宏和静态内联函数，用于快速检查和访问 Unicode 对象的内部只读数据：

int **PyUnicode_Check**(*PyObject* *obj)

如果对象 *obj* 是 Unicode 对象或 Unicode 子类型的实例则返回真值。此函数总是会成功执行。

int **PyUnicode_CheckExact**(*PyObject* *obj)

如果对象 *obj* 是一个 Unicode 对象，但不是某个子类型的实例则返回真值。此函数总是会成功执行。

int **PyUnicode_READY**(*PyObject* *unicode)

返回 `0`。此 API 仅为向下兼容而保留。

Added in version 3.3.

自 3.10 版本弃用: 此 API 从 Python 3.12 起将不做任何事。

Py_ssize_t **PyUnicode_GET_LENGTH**(*PyObject* *unicode)

返回以码位点数量表示的 Unicode 字符串长度。*unicode* 必须为“规范”表示的 Unicode 对象（不会检查这一点）。

Added in version 3.3.

Py_UCS1 ***PyUnicode_1BYTE_DATA**(*PyObject* *unicode)

Py_UCS2 ***PyUnicode_2BYTE_DATA**(*PyObject* *unicode)

Py_UCS4 ***PyUnicode_4BYTE_DATA**(*PyObject* *unicode)

返回一个用于直接字符访问的指向转换为 UCS1、UCS2 或 UCS4 整数类型的规范表示的指针。如果规范表示具有正确的字符大小，则不执行检查；使用*PyUnicode_KIND()* 选择正确的函数。

Added in version 3.3.

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

返回*PyUnicode_KIND()* 宏的值。

Added in version 3.3.

在 3.12 版本发生变更: `PyUnicode_WCHAR_KIND` 已被移除。

int **PyUnicode_KIND**(*PyObject* *unicode)

返回一个 PyUnicode 类型的常量（见上文），指明此 see above) that indicate how many bytes per character this Unicode 对象用来存储每个字符所使用的字节数。*unicode* 必须为“规范”表示的 Unicode 对象（不会检查这一点）。

Added in version 3.3.

void ***PyUnicode_DATA**(*PyObject* *unicode)

返回一个指向原始 Unicode 缓冲区的空指针。*unicode* 必须为“规范”表示的 Unicode 对象（不会检查这一点）。

Added in version 3.3.

void **PyUnicode_WRITE**(int kind, void *data, *Py_ssize_t* index, *Py_UCS4* value)

写入一个规范表示的 *data* (如同用*PyUnicode_DATA()* 获取)。此函数不会执行正确性检查，被设计为在循环中使用。调用者应当如同从其他调用中获取一样缓存 *kind* 值和 *data* 指针。*index* 是字符串中的索引号 (从 0 开始) 而 *value* 是应写入该位置的新码位值。

Added in version 3.3.

Py_UCS4 **PyUnicode_READ**(int kind, void *data, *Py_ssize_t* index)

从规范表示的 *data* (如同用*PyUnicode_DATA()* 获取) 中读取一个码位。不会执行检查或就绪调用。

Added in version 3.3.

Py_UCS4 **PyUnicode_READ_CHAR**(*PyObject* *unicode, *Py_ssize_t* index)

从 Unicode 对象 *unicode* 读取一个字符，必须为“规范”表示形式。如果你执行多次连续读取则此函数的效率将低于*PyUnicode_READ()*。

Added in version 3.3.

Py_UCS4 **PyUnicode_MAX_CHAR_VALUE**(*PyObject* *unicode)

返回适合基于 *unicode* 创建另一个字符串的最大码位点，该参数必须为“规范”表示形式。这始终是一种近似但比在字符串上执行迭代更高效。

Added in version 3.3.

int **PyUnicode_IsIdentifier**(*PyObject* *unicode)

属于稳定 ABI. 如果字符串按照语言定义是合法的标识符则返回 `1`，参见 identifiers 小节。否则返回 `0`。

在 3.9 版本发生变更: 如果字符串尚未就绪则此函数不会再调用*Py_FatalError()*。

Unicode 字符属性

Unicode 提供了许多不同的字符特性。最常需要的宏可以通过这些宏获得，这些宏根据 Python 配置映射到 C 函数。

int **Py_UNICODE_ISSPACE**(*Py_UCS4* ch)

根据 *ch* 是否为空白字符返回 `1` 或 `0`。

int **Py_UNICODE_ISLOWER**(*Py_UCS4* ch)

根据 *ch* 是否为小写字符返回 `1` 或 `0`。

int **Py_UNICODE_ISUPPER**(*Py_UCS4* ch)

根据 *ch* 是否为大写字符返回 `1` 或 `0`

int **Py_UNICODE_ISTITLE**(*Py_UCS4* ch)

根据 *ch* 是否为标题化的大小写返回 `1` 或 `0`。

int **Py_UNICODE_ISLINEBREAK**(*Py_UCS4* ch)

根据 *ch* 是否为换行类字符返回 `1` 或 `0`。

int **Py_UNICODE_ISDECIMAL**(*Py_UCS4* ch)

根据 *ch* 是否为十进制数字符返回 `1` 或 `0`。

int **Py_UNICODE_ISDIGIT**(*Py_UCS4* ch)

根据 *ch* 是否为数码类字符返回 `1` 或 `0`。

int **Py_UNICODE_ISNUMERIC**(*Py_UCS4* ch)

根据 *ch* 是否为数值类字符返回 `1` 或 `0`。

int **Py_UNICODE_ISALPHA**(*Py_UCS4* ch)

根据 *ch* 是否为字母类字符返回 `1` 或 `0`。

int **Py_UNICODE_ISALNUM**(*Py_UCS4* ch)

根据 *ch* 是否为字母数字类字符返回 `1` 或 `0`。

int **Py_UNICODE_ISPRINTABLE**(*Py_UCS4* ch)

根据 *ch* 是否为可打印字符返回 `1` 或 `0`，基于 `str.isprintable()` 来判断。

这些 API 可用于快速直接的字符转换:

Py_UCS4 **Py_UNICODE_TOLOWER**(*Py_UCS4* ch)

返回转换为小写形式的字符 *ch*。

Py_UCS4 **Py_UNICODE_TOUPPER**(*Py_UCS4* ch)

返回转换为大写形式的字符 *ch*。

Py_UCS4 **Py_UNICODE_TOTITLE**(*Py_UCS4* ch)

返回转换为标题大小写形式的字符 *ch*。

int **Py_UNICODE_TODECIMAL**(*Py_UCS4* ch)

将字符 *ch* 转换为十进制正整数返回。如果无法转换则返回 `-1`。此函数不会引发异常。

int **Py_UNICODE_TODIGIT**(*Py_UCS4* ch)

将字符 *ch* 转换为单个数码位的整数返回。如果无法转换则返回 `-1`。此函数不会引发异常。

double **Py_UNICODE_TONUMERIC**(*Py_UCS4* ch)

将字符 *ch* 转换为双精度浮点数返回。如果无法转换则返回 `-1.0`。此函数不会引发异常。

这些 API 可被用来操作代理项：

int **Py_UNICODE_IS_SURROGATE**(*Py_UCS4* ch)

检测 *ch* 是否为代理项 (`0xD800 <= ch <= 0xDFFF`)。

int **Py_UNICODE_IS_HIGH_SURROGATE**(*Py_UCS4* ch)

检测 *ch* 是否为高代理项 (`0xD800 <= ch <= 0xDBFF`)。

int **Py_UNICODE_IS_LOW_SURROGATE**(*Py_UCS4* ch)

检测 *ch* 是否为低代理项 (`0xDC00 <= ch <= 0xDFFF`)。

Py_UCS4 **Py_UNICODE_JOIN_SURROGATES**(*Py_UCS4* high, *Py_UCS4* low)

合并两个代理码位并返回单个*Py_UCS4* 值。*high* 和 *low* 分别为一个代理对的开头和末尾代理项。*high* 必须在 [0xD800; 0xDBFF] 范围内而 *low* 必须在 [0xDC00; 0xDFFF] 范围内。

创建和访问 Unicode 字符串

要创建 Unicode 对象和访问其基本序列属性，请使用这些 API：

PyObject ***PyUnicode_New**(*Py_ssize_t* size, *Py_UCS4* maxchar)

返回值：新的引用。 创建一个新的 Unicode 对象。*maxchar* 应为可放入字符串的实际最大码位。作为一个近似值，它可被向上舍入到序列 127, 255, 65535, 1114111 中最接近的值。

这是分配新的 Unicode 对象的推荐方式。使用此函数创建的对象不可改变大小。

发生错误时，将设置异常并返回 `NULL`。

Added in version 3.3.

PyObject ***PyUnicode_FromKindAndData**(int kind, const void *buffer, *Py_ssize_t* size)

返回值：新的引用。以给定的 *kind* 创建一个新的 Unicode 对象（可能的值为*PyUnicode_1BYTE_KIND* 等，即*PyUnicode_KIND()* 所返回的值）。*buffer* 必须指向由此分类所给出的，以每字符 1, 2 或 4 字节单位的 *size* 大小的数组。

如有必要，输入 *buffer* 将被拷贝并转换为规范表示形式。例如，如果 *buffer* 是一个 UCS4 字符串 (*PyUnicode_4BYTE_KIND*) 且仅由 UCS1 范围内的码位组成，它将被转换为 UCS1 (*PyUnicode_1BYTE_KIND*)。

Added in version 3.3.

PyObject ***PyUnicode_FromStringAndSize**(const char *str, *Py_ssize_t* size)

返回值：新的引用。属于稳定 ABI. 根据字符缓冲区 *str* 创建一个 Unicode 对象。字节数据将按 UTF-8 编码格式来解读。缓冲区会被拷贝到新的对象中。返回值可以是一个共享对象，即其数据不允许修改。

此函数会因以下情况而引发 `SystemError`：

- *size* < 0,
- *str* 为 `NULL` 且 *size* > 0

在 3.12 版本发生变更: *str* == `NULL` 且 *size* > 0 不再被允许。

PyObject ***PyUnicode_FromString**(const char *str)

返回值：新的引用。属于稳定 ABI. 根据 UTF-8 编码的以空值结束的字符缓冲区 *str* 创建一个 Unicode 对象。

PyObject ***PyUnicode_FromFormat**(const char *format, ...)

返回值：新的引用。属于稳定 ABI. 接受一个 C `printf()` 风格的 *format* 字符串和可变数量的参数，计算结果 Python Unicode 字符串的大小并返回包含已格式化值的字符串。可变数量的参数必须均为 C 类型并且必须恰好与 *format* ASCII 编码字符串中的格式字符相对应。

转换标记符包含两个或更多字符并具有以下组成，且必须遵循此处规定的顺序：

1. `'%'` 字符，用于标记转换符的起始。
2. 转换旗标（可选），用于影响某些转换类型的结果。
3. 最小字段宽度（可选）。如果指定为 `'*'` (星号)，则实际宽度会在下一参数中给出，该参数必须为 `int` 类型，要转换的对象则放在最小字段宽度和可选精度之后。
4. 精度（可选），以在 `'.'` (点号) 之后加精度值的形式给出。如果指定为 `'*'` (星号)，则实际精度会在下一参数中给出，该参数必须为 `int` 类型，要转换的对象则放在精度之后。
5. 长度修饰符（可选）。
6. 转换类型。

转换旗标为：

标志	含意
`0`	转换将为数字值填充零字符。
`-`	转换值将靠左对齐（如果同时给出则会覆盖 `0` 旗标）。

以下整数转换的长度修饰符 (`d`, `i`, `o`, `u`, `x`, or `X`) 指明参数的类型 (默认为 `int`):

修饰符	类型
`l`	`long` 或 `unsigned long`
`ll`	`long long` 或 `unsigned long long`
`j`	`intmax_t` 或 `uintmax_t`
`z`	`size_t` 或 `ssize_t`
`t`	`ptrdiff_t`

针对以下转换 `s` 或 `V` 的长度修饰符 `l` 指明参数的类型为 `const wchar_t*`。

转换指示符如下：

转换指示符	类型	注释
`%`	不适用	字面的 `%` 字符。
`d`, `i`	由长度修饰符指明	有符号 C 整数的十进制表示。
`u`	由长度修饰符指明	无符号 C 整数的十进制表示。
`o`	由长度修饰符指明	无符号 C 整数的八进制表示。
`x`	由长度修饰符指明	无符号 C 整数的十六进制表示（小写）。
`X`	由长度修饰符指明	无符号 C 整数的十六进制表示（大写）。
`c`	`int`	单个字符。
`s`	`const char*` 或 `const wchar_t*`	以 null 为终止符的 C 字符数组。
`p`	`const void*`	一个 C 指针的十六进制表示形式。基本等价于 `printf("%p")` 但它会确保以字面值 `0x` 开头而不管系统平台上的 `printf` 输出是什么。
`A`	*PyObject**	`ascii()` 调用的结果。
`U`	*PyObject**	一个 Unicode 对象。
`V`	*PyObject**, `const char*` 或 `const wchar_t*`	一个 Unicode 对象 (可以为 `NULL`) 和一个以空值结束的 C 字符数组作为第二个形参（如果第一个形参为 `NULL`，第二个形参将被使用）。
`S`	*PyObject**	调用*PyObject_Str()* 的结果。
`R`	*PyObject**	调用*PyObject_Repr()* 的结果。
`T`	*PyObject**	要获取对象类型的完整限定名称；请调用*PyType_GetFullyQualifiedName()*。
`#T`	*PyObject**	类似于 `T` 格式，但会使用冒号 (`:`) 作为模块名称和限定名称之间的分隔符。
`N`	*PyTypeObject**	获取类型的完整限定名称；调用*PyType_GetFullyQualifiedName()*。
`#N`	*PyTypeObject**	类似于 `N` 格式，但会使用冒号 (`:`) 作为模块名称和限定名称之间的分隔符。

备注

格式符的宽度单位是字符数而不是字节数。格式符的精度单位对于 `"%s"` 和 `"%V"` (如果 `PyObject*` 参数为 `NULL`) 是字节数或 `wchar_t` 项数 (如果使用了长度修饰符 `l`)，而对于 `"%A"`, `"%U"`, `"%S"`, `"%R"` 和 `"%V"` (如果 `PyObject*` 参数不为 `NULL`) 则为字符数。

备注

与 C `printf()` 不同的是 `0` 旗标即使在为整数转换 (`d`, `i`, `u`, `o`, `x`, or `X`) 指定精度时也是有效的。

在 3.2 版本发生变更: 增加了对 `"%lld"` 和 `"%llu"` 的支持。

在 3.3 版本发生变更: 增加了对 `"%li"`, `"%lli"` 和 `"%zi"` 的支持。

在 3.4 版本发生变更: 增加了对 `"%s"`, `"%A"`, `"%U"`, `"%V"`, `"%S"`, `"%R"` 的宽度和精度格式符支持。

在 3.12 版本发生变更: 支持转换说明符 `o` 和 `X`。支持长度修饰符 `j` 和 `t`。长度修饰符现在将应用于所有整数转换。长度修饰符 `l` 现在将应用于转换说明符 `s` 和 `V`。支持可变宽度和精度 `*`。支持旗标 `-`。

不可识别的格式字符现在会设置一个 `SystemError`。在之前版本中它会导致所有剩余格式字符串被原样拷贝到结果字符串，并丢弃任何额外的参数。

在 3.13 版本发生变更: 增加了对 `%T`, `%#T`, `%N` 和 `%#N` 等格式的支持。

PyObject ***PyUnicode_FromFormatV**(const char *format, va_list vargs)

返回值：新的引用。属于稳定 ABI. 等同于*PyUnicode_FromFormat()* 但它将接受恰好两个参数。

PyObject ***PyUnicode_FromObject**(*PyObject* *obj)

返回值：新的引用。属于稳定 ABI. 如有必要将把一个 Unicode 子类型的实例拷贝为新的真实 Unicode 对象。如果 *obj* 已经是一个真实 Unicode 对象（而非子类型），则返回一个新的指向该对象的*strong reference*。

非 Unicode 或其子类型的对象将导致 `TypeError`。

PyObject ***PyUnicode_FromOrdinal**(int ordinal)

返回值：新的引用。属于稳定 ABI. 根据给定的 Unicode 码点 *ordinal* 创建一个 Unicode 对象。

码点必须在 `range(0x110000)` 范围内。如果超出范围则会引发 `ValueError`。

PyObject ***PyUnicode_FromEncodedObject**(*PyObject* *obj, const char *encoding, const char *errors)

返回值：新的引用。属于稳定 ABI. 将一个已编码的对象 *obj* 解码为 Unicode 对象。

`bytes`, `bytearray` 和其他字节类对象 将按照给定的 *encoding* 来解码并使用由 *errors* 定义的错误处理方式。两者均可为 `NULL` 即让接口使用默认值（请参阅内置编解码器 了解详情）。

所有其他对象，包括 Unicode 对象，都将导致设置 `TypeError`。

如有错误该 API 将返回 `NULL`。调用方要负责递减指向所返回对象的引用。

PyObject ***PyUnicode_BuildEncodingMap**(*PyObject* *string)

返回值：新的引用。属于稳定 ABI. 返回一个适用于对自定义单字节编码格式进行解码的映射。给定一个表示编码表的至多 256 个字符的 Unicode 字符串 *string*，返回一个将字符编号映射到字节值的紧凑内部映射对象或字典。对于不合法的输入将引发 `TypeError` 并返回 `NULL`。.. versionadded:: 3.2

const char ***PyUnicode_GetDefaultEncoding**(void)

属于稳定 ABI. 返回默认字符编码格式名称，即 `"utf-8"`。参见 `sys.getdefaultencoding()`。

返回的字符串不需要被释放，并将保持可用直到解释器关闭。

Py_ssize_t **PyUnicode_GetLength**(*PyObject* *unicode)

属于稳定 ABI 自 *3.7* 版起. 返回 Unicode 对象码位的长度。

发生错误时，将设置异常并返回 `-1`。

Added in version 3.3.

Py_ssize_t **PyUnicode_CopyCharacters**(*PyObject* *to, *Py_ssize_t* to_start, *PyObject* *from, *Py_ssize_t* from_start, *Py_ssize_t* how_many)

将一个 Unicode 对象中的字符拷贝到另一个对象中。此函数会在必要时执行字符转换并会在可能的情况下回退到 `memcpy()`。在出错时将返回 `-1` 并设置一个异常，在其他情况下将返回拷贝的字符数量。

Added in version 3.3.

Py_ssize_t **PyUnicode_Fill**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* length, *Py_UCS4* fill_char)

使用一个字符填充字符串：将 *fill_char* 写入 `unicode[start:start+length]`。

如果 *fill_char* 值大于字符串最大字符值，或者如果字符串有 1 以上的引用将执行失败。

返回写入的字符数量，或者在出错时返回 `-1` 并引发一个异常。

Added in version 3.3.

int **PyUnicode_WriteChar**(*PyObject* *unicode, *Py_ssize_t* index, *Py_UCS4* character)

属于稳定 ABI 自 *3.7* 版起. 将一个字符写入到字符串。字符串必须通过*PyUnicode_New()* 创建。由于 Unicode 字符串应当是不可变的，因此该字符串不能被共享，或是被哈希。

该函数将检查 *unicode* 是否为 Unicode 对象，索引是否未越界，并且对象是否可被安全地修改（即其引用计数为一）。

成功时返回 0，出错时返回 -1 并设置一个异常。

Added in version 3.3.

Py_UCS4 **PyUnicode_ReadChar**(*PyObject* *unicode, *Py_ssize_t* index)

属于稳定 ABI 自 *3.7* 版起. 从字符串读取一个字符。该函数将检查 *unicode* 是否为 Unicode 对象且索引是否未越界，这不同于*PyUnicode_READ_CHAR()*，后者不会执行任何错误检查。

成功时返回字符，出错时返回 -1 并设置一个异常。

Added in version 3.3.

PyObject ***PyUnicode_Substring**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* end)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 返回 *unicode* 的一个子串，从字符索引 *start* (含) 到字符索引 *end* (不含)。不支持负索引号。出错时，将设置一个异常并返回 NULL。

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4**(*PyObject* *unicode, *Py_UCS4* *buffer, *Py_ssize_t* buflen, int copy_null)

属于稳定 ABI 自 *3.7* 版起. 将字符串 *unicode* 拷贝到一个 UCS4 缓冲区，包括一个空字符，如果设置了 *copy_null* 的话。出错时返回 NULL 并设置一个异常（特别是当 *buflen* 小于 *unicode* 的长度时，将设置 SystemError 异常）。成功时返回 *buffer*。

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4Copy**(*PyObject* *unicode)

属于稳定 ABI 自 *3.7* 版起. 将字符串 *unicode* 拷贝到使用*PyMem_Malloc()* 分配的新 UCS4 缓冲区。如果执行失败，将返回 NULL 并设置 MemoryError。返回的缓冲区将总是会添加一个额外的空码位。

Added in version 3.3.

语言区域编码格式

当前语言区域编码格式可被用来解码来自操作系统的文本。

PyObject ***PyUnicode_DecodeLocaleAndSize**(const char *str, *Py_ssize_t* length, const char *errors)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 解码字符串在 Android 和 VxWorks 上使用 UTF-8，在其他平台上则使用当前语言区域编码格式。支持的错误处理器有 "strict" 和 "surrogateescape" (**PEP 383**)。如果 *errors* 为 NULL 则解码器将使用 "strict" 错误处理器。*str* 必须以一个空字符结束但不可包含嵌入的空字符。

使用*PyUnicode_DecodeFSDefaultAndSize()* 以*filesystem encoding and error handler* 来解码字符串。

此函数将忽略 Python UTF-8 模式。

> **参见**
>
> The *Py_DecodeLocale()* 函数。

Added in version 3.3.

在 3.7 版本发生变更: 此函数现在也会为 surrogateescape 错误处理器使用当前语言区域编码格式，但在 Android 上例外。在之前版本中，*Py_DecodeLocale()* 将被用于 surrogateescape，而当前语言区域编码格式将被用于 strict。

PyObject ***PyUnicode_DecodeLocale**(const char *str, const char *errors)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 类似于*PyUnicode_DecodeLocaleAndSize()*，但会使用 strlen() 来计算字符串长度。

Added in version 3.3.

PyObject ***PyUnicode_EncodeLocale**(*PyObject* *unicode, const char *errors)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 编码 Unicode 对象在 Android 和 VxWorks 上使用 UTF-8，在其他平台上使用当前语言区域编码格式。支持的错误处理器有 "strict" 和 "surrogateescape" (**PEP 383**)。如果 *errors* 为 NULL 则编码器将使用 "strict" 错误处理器。返回一个 bytes 对象。*unicode* 不可包含嵌入的空字符。

使用 *PyUnicode_EncodeFSDefault()* 将字符串编码为 *filesystem encoding and error handler*。

此函数将忽略 Python UTF-8 模式。

> **参见**
>
> *Py_EncodeLocale()* 函数。

Added in version 3.3.

在 3.7 版本发生变更: 此函数现在也会为 surrogateescape 错误处理器使用当前语言区域编码格式，但在 Android 上例外。在之前版本中，*Py_EncodeLocale()* 将被用于 surrogateescape，而当前语言区域编码格式将被用于 strict。

文件系统编码格式

使用 *filesystem encoding and error handler* 的编码和解码函数 (**PEP 383** 和 **PEP 529**)。

要在参数解析期间将文件名编码为 bytes，应当使用 "O&" 转换器，传入 PyUnicode_FSConverter() 作为转换函数：

int **PyUnicode_FSConverter**(*PyObject* *obj, void *result)

属于稳定 ABI. *PyArg_Parse** 转换器: 使用 *PyUnicode_EncodeFSDefault()* 编码 str 对象 -- 直接获取或通过 os.PathLike 接口 -- 为 bytes; bytes 对象将被原样输出。*result* 必须是一个类型为 *PyObject** (或 *PyBytesObject**) 的 C 变量的地址。执行成功时，将把该变量设为指向一个字节串对象 的新的 *strong reference*，它必须在其不再使用时被释放并返回一个非零值 (*Py_CLEANUP_SUPPORTED*)。结果中不允许嵌入空字节。执行失败时，将返回 0 并设置一个异常。

如果 *obj* 为 NULL，该函数将释放存放在 *result* 所引用的变量中的强引用并返回 1。

Added in version 3.1.

在 3.6 版本发生变更: 接受一个 *path-like object*。

要在参数解析期间将文件名解码为 str，应当使用 "O&" 转换器，传入 PyUnicode_FSDecoder() 作为转换函数：

int **PyUnicode_FSDecoder**(*PyObject* *obj, void *result)

属于稳定 ABI. *PyArg_Parse** 转换器: 使用 *PyUnicode_DecodeFSDefaultAndSize()* 解码 bytes 对象 -- 直接获取或通过 os.PathLike 接口 -- 为 str; str 对象将被原样输出。*result* 必须是一个类型为 *PyObject** (或 *PyUnicodeObject**) 的 C 变量的地址。执行成功时，将把该变量设为指向一个 *Unicode 对象* 的新的 *strong reference*，它必须在其不再使用时被释放并返回一个非零值 (*Py_CLEANUP_SUPPORTED*)。结果中不允许嵌入空字节。执行失败时，将返回 0 并设置一个异常。

如果 *obj* 为 NULL，则释放指向 *result* 所引用的对象的强引用并返回 1。

Added in version 3.2.

在 3.6 版本发生变更: 接受一个 *path-like object*。

PyObject ***PyUnicode_DecodeFSDefaultAndSize**(const char *str, *Py_ssize_t* size)

返回值：新的引用。属于稳定 ABI. 使用 *filesystem encoding and error handler* 解码字符串。

如果你需要以当前语言区域编码格式解码字符串，请使用 *PyUnicode_DecodeLocaleAndSize()*。

> **参见**
>
> The *Py_DecodeLocale()* 函数。

在 3.6 版本发生变更: 现在将使用文件系统编码格式和错误处理器。

PyObject ***PyUnicode_DecodeFSDefault**(const char *str)

返回值：新的引用。属于稳定 ABI. 使用*filesystem encoding and error handler* 解码以空值结尾的字符串。

如果字符串长度已知，则使用*PyUnicode_DecodeFSDefaultAndSize()*。

在 3.6 版本发生变更: 现在将使用文件系统编码格式和错误处理器。

PyObject ***PyUnicode_EncodeFSDefault**(*PyObject* *unicode)

返回值：新的引用。属于稳定 ABI. 使用*filesystem encoding and error handler* 编码一个 Unicode 对象，并返回 `bytes`。请注意结果 `bytes` 对象可以包含空字节。

如果你需要以当前语言区域编码格式编码字符串，请使用*PyUnicode_EncodeLocale()*。

> **参见**
>
> *Py_EncodeLocale()* 函数。

Added in version 3.2.

在 3.6 版本发生变更: 现在将使用文件系统编码格式和错误处理器。

wchar_t 支持

在受支持的平台上支持 `wchar_t`:

PyObject ***PyUnicode_FromWideChar**(const wchar_t *wstr, *Py_ssize_t* size)

返回值：新的引用。属于稳定 ABI. 根据给定的 *size* 的 `wchar_t` 缓冲区 *wstr* 创建一个 Unicode 对象。传入 `-1` 作为 *size* 表示该函数必须使用 `wcslen()` 自动计算缓冲区长度。失败时将返回 `NULL`。

Py_ssize_t **PyUnicode_AsWideChar**(*PyObject* *unicode, wchar_t *wstr, *Py_ssize_t* size)

属于稳定 ABI. 将 Unicode 对象的内容拷贝到 `wchar_t` 缓冲区 *wstr* 中。至多拷贝 *size* 个 `wchar_t` 字符（不包括可能存在的末尾空结束字符）。返回拷贝的 `wchar_t` 字符数或在出错时返回 `-1`。

当 *wstr* 为 `NULL` 时，则改为返回存储包括结束空值在内的所有 *unicode* 内容所需的 *size*。

请注意结果 `wchar_t*` 字符串可能是以空值结束也可能不是。调用方要负责确保 `wchar_t*` 字符串以空值结束以防应用有此要求。此外，请注意 `wchar_t*` 字符串有可能包含空字符，这将导致字符串在与大多数 C 函数一起使用时被截断。

wchar_t ***PyUnicode_AsWideCharString**(*PyObject* *unicode, *Py_ssize_t* *size)

属于稳定 ABI 自 *3.7* 版起. 将 Unicode 对象转换为宽字符串。输出字符串将总是以空字符结尾。如果 *size* 不为 `NULL`，则会将宽字符的数量（不包括结尾空字符）写入到 **size* 中。请注意结果 `wchar_t` 字符串可能包含空字符，这将导致在大多数 C 函数中使用时字符串被截断。如果 *size* 为 `NULL` 并且 `wchar_t*` 字符串包含空字符则将引发 `ValueError`。

成功时返回由*PyMem_New* 分配的缓冲区（使用*PyMem_Free()* 来释放它）。发生错误时，则返回 `NULL` 并且 **size* 将是未定义的。如果内存分配失败则会引发 `MemoryError`。

Added in version 3.2.

在 3.7 版本发生变更: 如果 *size* 为 `NULL` 且 `wchar_t*` 字符串包含空字符则会引发 `ValueError`。

内置编解码器

Python 提供了一组以 C 编写以保证运行速度的内置编解码器。所有这些编解码器均可通过下列函数直接使用。

下列 API 大都接受 encoding 和 errors 两个参数，它们具有与在内置 `str()` 字符串对象构造器中同名参数相同的语义。

将 encoding 设为 `NULL` 将使用默认编码格式即 UTF-8。文件系统调用应当使用 *PyUnicode_FSConverter()* 来编码文件名。这将在内部使用 *filesystem encoding and error handler*。

错误处理方式由 errors 设置并且也可以设为 `NULL` 表示使用为编解码器定义的默认处理方式。所有内置编解码器的默认错误处理方式是"strict" (会引发 `ValueError`)。

编解码器都使用类似的接口。为了保持简单只有与下列泛型编解码器的差异才会记录在文档中。

泛型编解码器

以下是泛型编解码器的 API:

PyObject ***PyUnicode_Decode**(const char *str, *Py_ssize_t* size, const char *encoding, const char *errors)

返回值：新的引用。属于稳定 ABI. 通过解码已编码字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。*encoding* 和 *errors* 具有与 `str()` 内置函数中同名形参相同的含义。要使用的编解码将使用 Python 编解码器注册表来查找。如果编解码器引发了异常则返回 `NULL`。

PyObject ***PyUnicode_AsEncodedString**(*PyObject* *unicode, const char *encoding, const char *errors)

返回值：新的引用。属于稳定 ABI. 编码一个 Unicode 对象并将结果作为 Python 字节串对象返回。*encoding* 和 *errors* 具有与 Unicode `encode()` 方法中同名形参相同的含义。要使用的编解码器将使用 Python 编解码器注册表来查找。如果编解码器引发了异常则返回 `NULL`。

UTF-8 编解码器

以下是 UTF-8 编解码器 API:

PyObject ***PyUnicode_DecodeUTF8**(const char *str, *Py_ssize_t* size, const char *errors)

返回值：新的引用。属于稳定 ABI. 通过解码 UTF-8 编码的字节串 *str* 的的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 `NULL`。

PyObject ***PyUnicode_DecodeUTF8Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

返回值：新的引用。属于稳定 ABI. 如果 *consumed* 为 `NULL`，则行为类似于 *PyUnicode_DecodeUTF8()*。如果 *consumed* 不为 `NULL`，则末尾的不完整 UTF-8 字节序列将不被视为错误。这些字节将不会被解码并且已被解码的字节数将存储在 *consumed* 中。

PyObject ***PyUnicode_AsUTF8String**(*PyObject* *unicode)

返回值：新的引用。属于稳定 ABI. 使用 UTF-8 编码 Unicode 对象并将结果作为 Python 字节串对象返回。错误处理方式为"strict"。如果编解码器引发了异常则将返回 `NULL`。

如果字符串包含代理码位 (`U+D800` - `U+DFFF`) 则该函数的执行将失败。

const char ***PyUnicode_AsUTF8AndSize**(*PyObject* *unicode, *Py_ssize_t* *size)

属于稳定 ABI 自 *3.10* 版起. 返回一个指向 Unicode 对象的 UTF-8 编码格式数据的指针，并将已编码数据的大小（以字节为单位）存储在 *size* 中。*size* 参数可以为 `NULL`；在此情况下数据的大小将不会被存储。返回的缓冲区总是会添加一个额外的空字节（不包括在 *size* 中），无论是否存在任何其他的空码位。

发生错误时，设置一个异常，将 *size* 设为 `-1` (如果不为 NULL) 并返回 `NULL`。

如果字符串包含代理码位 (`U+D800` - `U+DFFF`) 则该函数的执行将失败。

这将缓存 Unicode 对象中字符串的 UTF-8 表示形式，并且后续调用将返回指向同一缓存区的指针。调用方不必负责释放该缓冲区。缓冲区会在 Unicode 对象被作为垃圾回收时被释放并使指向它的指针失效。

Added in version 3.3.

在 3.7 版本发生变更: 返回类型现在是 `const char *` 而不是 `char *`。

在 3.10 版本发生变更: 此函数是受限 *API* 的组成部分。

const char ***PyUnicode_AsUTF8**(*PyObject* *unicode)

类似于*PyUnicode_AsUTF8AndSize()*，但不会存储大小值。

> **警告**
>
> 此函数没有任何针对嵌入 *unicode* 内部的 空字符 的特殊行为。因此，字符串中包含的空字符将保留在返回的字符串中，这在某些 C 函数中可能会被解读为字符串的结束，导致其被截断。如果截断会带来问题，建议改用*PyUnicode_AsUTF8AndSize()*。

Added in version 3.3.

在 3.7 版本发生变更: 返回类型现在是 `const char *` 而不是 `char *`。

UTF-32 编解码器

以下是 UTF-32 编解码器 API:

PyObject ***PyUnicode_DecodeUTF32**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

返回值：新的引用。属于稳定 ABI. 从 UTF-32 编码的缓冲区数据解码 *size* 个字节并返回相应的 Unicode 对象。*errors* (如果不为 `NULL`) 定义了错误处理方式。默认为"strict"。

如果 *byteorder* 不为 `NULL`，解码器将使用给定的字节序进行解码:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

如果 `*byteorder` 为零，且输入数据的前四个字节为字节序标记 (BOM)，则解码器将切换为该字节序并且 BOM 将不会被拷贝到结果 Unicode 字符串中。如果 `*byteorder` 为 `-1` 或 `1`，则字节序标记会被拷贝到输出中。

在完成后，**byteorder* 将在输入数据的末尾被设为当前字节序。

如果 *byteorder* 为 `NULL`，编解码器将使用本机字节序。

如果编解码器引发了异常则返回 `NULL`。

PyObject ***PyUnicode_DecodeUTF32Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

返回值：新的引用。属于稳定 ABI. 如果 *consumed* 为 `NULL`，则行为类似于*PyUnicode_DecodeUTF32()*。如果 *consumed* 不为 `NULL`，则*PyUnicode_DecodeUTF32Stateful()* 将不把末尾的不完整 UTF-32 字节序列（如字节数不可被四整除）视为错误。这些字节将不会被解码并且已被解码的字节数将存储在 *consumed* 中。

PyObject ***PyUnicode_AsUTF32String**(*PyObject* *unicode)

返回值：新的引用。属于稳定 ABI. 返回使用 UTF-32 编码格式本机字节序的 Python 字节串。字节串将总是以 BOM 标记打头。错误处理方式为"strict"。如果编解码器引发了异常则返回 `NULL`。

UTF-16 编解码器

以下是 UTF-16 编解码器的 API:

PyObject ***PyUnicode_DecodeUTF16**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

返回值：新的引用。属于稳定 ABI. 从 UTF-16 编码的缓冲区数据解码 *size* 个字节并返回相应的 Unicode 对象。*errors* (如果不为 `NULL`) 定义了错误处理方式。默认为"strict"。

如果 *byteorder* 不为 `NULL`，解码器将使用给定的字节序进行解码:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

如果 *byteorder 为零，且输入数据的前两个字节为字节序标记 (BOM)，则解码器将切换为该字节序并且 BOM 将不会被拷贝到结果 Unicode 字符串中。如果 *byteorder 为 -1 或 1，则字节序标记会被拷贝到输出中 (它将是一个 \ufeff 或 \ufffe 字符)。

在完成后，*byteorder 将在输入数据的末尾被设为当前字节序。

如果 *byteorder* 为 NULL，编解码器将使用本机字节序。

如果编解码器引发了异常则返回 NULL。

PyObject ***PyUnicode_DecodeUTF16Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

返回值：新的引用。属于稳定 ABI. 如果 *consumed* 为 NULL，则行为类似于 *PyUnicode_DecodeUTF16()*。如果 *consumed* 不为 NULL，则 *PyUnicode_DecodeUTF16Stateful()* 将不把末尾的不完整 UTF-16 字节序列（如为奇数个字节或为分开的替代对）视为错误。这些字节将不会被解码并且已被解码的字节数将存储在 *consumed* 中。

PyObject ***PyUnicode_AsUTF16String**(*PyObject* *unicode)

返回值：新的引用。属于稳定 ABI. 返回使用 UTF-16 编码格式本机字节序的 Python 字节串。字节串将总是以 BOM 标记打头。错误处理方式为”strict”。如果编解码器引发了异常则返回 NULL。

UTF-7 编解码器

以下是 UTF-7 编解码器 API:

PyObject ***PyUnicode_DecodeUTF7**(const char *str, *Py_ssize_t* size, const char *errors)

返回值：新的引用。属于稳定 ABI. 通过解码 UTF-7 编码的字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 NULL。

PyObject ***PyUnicode_DecodeUTF7Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

返回值：新的引用。属于稳定 ABI. 如果 *consumed* 为 NULL，则行为类似于 *PyUnicode_DecodeUTF7()*。如果 *consumed* 不为 NULL，则末尾的不完整 UTF-7 base-64 部分将不被视为错误。这些字节将不会被解码并且已被解码的字节数将存储在 *consumed* 中。

Unicode-Escape 编解码器

以下是”Unicode Escape” 编解码器的 API:

PyObject ***PyUnicode_DecodeUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

返回值：新的引用。属于稳定 ABI. 通过解码 Unicode-Escape 编码的字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 NULL。

PyObject ***PyUnicode_AsUnicodeEscapeString**(*PyObject* *unicode)

返回值：新的引用。属于稳定 ABI. 使用 Unicode-Escape 编码 Unicode 对象并将结果作为字节串对象返回。错误处理方式为”strict”。如果编解码器引发了异常则将返回 NULL。

Raw-Unicode-Escape 编解码器

以下是”Raw Unicode Escape” 编解码器的 API:

PyObject ***PyUnicode_DecodeRawUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

返回值：新的引用。属于稳定 ABI. 通过解码 Raw-Unicode-Escape 编码的字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 NULL。

PyObject ***PyUnicode_AsRawUnicodeEscapeString**(*PyObject* *unicode)

返回值：新的引用。属于稳定 ABI. 使用 Raw-Unicode-Escape 编码 Unicode 对象并将结果作为字节串对象返回。错误处理方式为”strict”。如果编解码器引发了异常则将返回 NULL。

Latin-1 编解码器

以下是 Latin-1 编解码器的 API: Latin-1 对应于前 256 个 Unicode 码位且编码器在编码期间只接受这些码位。

PyObject ***PyUnicode_DecodeLatin1**(const char *str, *Py_ssize_t* size, const char *errors)

返回值：新的引用。属于稳定 ABI. 通过解码 Latin-1 编码的字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 NULL。

PyObject ***PyUnicode_AsLatin1String**(*PyObject* *unicode)

返回值：新的引用。属于稳定 ABI. 使用 Latin-1 编码 Unicode 对象并将结果作为 Python 字节串对象返回。错误处理方式为"strict"。如果编解码器引发了异常则将返回 NULL。

ASCII 编解码器

以下是 ASCII 编解码器的 API。只接受 7 位 ASCII 数据。任何其他编码的数据都将导致错误。

PyObject ***PyUnicode_DecodeASCII**(const char *str, *Py_ssize_t* size, const char *errors)

返回值：新的引用。属于稳定 ABI. 通过解码 ASCII 编码的字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 NULL。

PyObject ***PyUnicode_AsASCIIString**(*PyObject* *unicode)

返回值：新的引用。属于稳定 ABI. 使用 ASCII 编码 Unicode 对象并将结果作为 Python 字节串对象返回。错误处理方式为"strict"。如果编解码器引发了异常则将返回 NULL。

字符映射编解码器

此编解码器的特殊之处在于它可被用来实现许多不同的编解码器（而且这实际上就是包括在 encodings 包中的大部分标准编解码器的实现方式）。此编解码器使用映射来编码和解码字符。所提供的映射对象必须支持 __getitem__() 映射接口；字典和序列均可胜任。

以下是映射编解码器的 API:

PyObject ***PyUnicode_DecodeCharmap**(const char *str, *Py_ssize_t* length, *PyObject* *mapping, const char *errors)

返回值：新的引用。属于稳定 ABI. 通过使用给定的 *mapping* 对象解码已编码字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 NULL。

如果 *mapping* 为 NULL，则将应用 Latin-1 编码格式。否则 *mapping* 必须为字节码位值（0 至 255 范围内的整数）到 Unicode 字符串的映射、整数（将被解读为 Unicode 码位）或 None。未映射的数据字节 -- 这样的数据将导致 LookupError，以及被映射到 None 的数据，0xFFFE 或 '\ufffe'，将被视为未定义的映射并导致报错。

PyObject ***PyUnicode_AsCharmapString**(*PyObject* *unicode, *PyObject* *mapping)

返回值：新的引用。属于稳定 ABI. 使用给定的 *mapping* 对象编码 Unicode 对象并将结果作为字节串对象返回。错误处理方式为"strict"。如果编解码器引发了异常则将返回 NULL。

mapping 对象必须将整数 Unicode 码位映射到字节串对象、0 至 255 范围内的整数或 None。未映射的字符码位（将导致 LookupError 的数据）以及映射到 None 的数据将被视为"未定义的映射"并导致报错。

以下特殊的编解码器 API 会将 Unicode 映射至 Unicode。

PyObject ***PyUnicode_Translate**(*PyObject* *unicode, *PyObject* *table, const char *errors)

返回值：新的引用。属于稳定 ABI. 通过应用字符映射表来转写字符串并返回结果 Unicode 对象。如果编解码器引发了异常则返回 NULL。

字符映射表必须将整数 Unicode 码位映射到整数 Unicode 码位或 None (这将删除相应的字符)。

映射表只需提供 __getitem__() 接口；字典和序列均可胜任。未映射的字符码位（将导致 LookupError 的数据）将保持不变并被原样拷贝。

errors 具有用于编解码器的通常含义。它可以为 NULL 表示使用默认的错误处理方式。

Windows 中的 MBCS 编解码器

以下是 MBCS 编解码器的 API。目前它们仅在 Windows 中可用并使用 Win32 MBCS 转换器来实现转换。请注意 MBCS（或 DBCS）是一类编码格式，而非只有一个。目标编码格式是由运行编解码器的机器上的用户设置定义的。

PyObject ***PyUnicode_DecodeMBCS**(const char *str, *Py_ssize_t* size, const char *errors)

返回值：新的引用。属于稳定 ABI *on Windows* 自 *3.7* 版起. 通过解码 MBCS 编码的字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 NULL。

PyObject ***PyUnicode_DecodeMBCSStateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

返回值：新的引用。属于稳定 ABI *on Windows* 自 *3.7* 版起. 如果 *consumed* 为 NULL，则行为类似于 *PyUnicode_DecodeMBCS()*。如果 *consumed* 不为 NULL，则 *PyUnicode_DecodeMBCSStateful()* 将不会解码末尾的不完整字节并且已被解码的字节数将存储在 *consumed* 中。

PyObject ***PyUnicode_DecodeCodePageStateful**(int code_page, const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

返回值：新的引用。属于稳定 ABI *on Windows* 自 *3.7* 版起. 类似于 *PyUnicode_DecodeMBCSStateful()*，区别在于使用由 *code_page* 所指定的代码页。

PyObject ***PyUnicode_AsMBCSString**(*PyObject* *unicode)

返回值：新的引用。属于稳定 ABI *on Windows* 自 *3.7* 版起. 使用 MBCS 编码 Unicode 对象并将结果作为 Python 字节串对象返回。错误处理方式为"strict"。如果编解码器引发了异常则将返回 NULL。

PyObject ***PyUnicode_EncodeCodePage**(int code_page, *PyObject* *unicode, const char *errors)

返回值：新的引用。属于稳定 ABI *on Windows* 自 *3.7* 版起. 使用指定的代码页编码 Unicode 对象并返回一个 Python 字节串对象。如果编解码器引发了异常则返回 NULL。使用 CP_ACP 代码页来获取 MBCS 解码器。

Added in version 3.3.

方法与槽位函数

以下 API 可以处理输入的 Unicode 对象和字符串（在描述中我们称其为字符串）并返回适当的 Unicode 对象或整数值。

如果发生异常它们都将返回 NULL 或 -1。

PyObject ***PyUnicode_Concat**(*PyObject* *left, *PyObject* *right)

返回值：新的引用。属于稳定 ABI. 拼接两个字符串得到一个新的 Unicode 字符串。

PyObject ***PyUnicode_Split**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

返回值：新的引用。属于稳定 ABI. 拆分一个字符串得到一个 Unicode 字符串的列表。如果 *sep* 为 NULL，则将根据空格来拆分所有子字符串。否则，将根据指定的分隔符来拆分。最多拆分数为 *maxsplit*。如为负值，则没有限制。分隔符不包括在结果列表中。

当失败时，将返回 NULL 并设置一个异常。

等价于 str.split()。

PyObject ***PyUnicode_RSplit**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

返回值：新的引用。属于稳定 ABI. 类似于 *PyUnicode_Split()*，但拆分将从字符串末尾开始进行。

当失败时，将返回 NULL 并设置一个异常。

等价于 str.rsplit()。

PyObject ***PyUnicode_Splitlines**(*PyObject* *unicode, int keepends)

返回值：新的引用。属于稳定 ABI. 根据分行符来拆分 Unicode 字符串，返回一个 Unicode 字符串的列表。CRLF 将被视为一个分行符。如果 *keepends* 为 0，则行分隔符将不包括在结果字符串中。

PyObject ***PyUnicode_Partition**(*PyObject* *unicode, *PyObject* *sep)

返回值：新的引用。属于稳定 ABI. 在 *sep* 首次出现的位置拆分 Unicode 字符串，并返回一个包含分隔符之前部分、分隔符本身，以及分隔符之后部分的 3 元组。如果分隔符未找到，则返回一个包含字符串本身，后面附带两个空字符串的 3 元组。

sep 必须不为空。

当失败时，将返回 NULL 并设置一个异常。

等价于 `str.partition()`。

PyObject ***PyUnicode_RPartition**(*PyObject* *unicode, *PyObject* *sep)

返回值：新的引用。属于稳定 ABI. 类似于*PyUnicode_Partition()*，但会在 *sep* 最后一次出现的位置拆分 Unicode 字符串。如果分隔符未找到，则返回一个包含两个空字符串，后面附带字符串本身的 3 元组。

sep 必须不为空。

当失败时，将返回 NULL 并设置一个异常。

等价于 `str.rpartition()`。

PyObject ***PyUnicode_Join**(*PyObject* *separator, *PyObject* *seq)

返回值：新的引用。属于稳定 ABI. 使用给定的 *separator* 合并一个字符串列表并返回结果 Unicode 字符串。

Py_ssize_t **PyUnicode_Tailmatch**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

属于稳定 ABI. 如果 *substr* 在给定的端点 (*direction* == `-1` 表示前缀匹配, *direction* == `1` 表示后缀匹配) 与 `unicode[start:end]` 相匹配则返回 `1`，否则返回 `0`。如果发生错误则返回 `-1`。

Py_ssize_t **PyUnicode_Find**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

属于稳定 ABI. 返回使用给定的 *direction* (*direction* == `1` 表示前向搜索，*direction* == `-1` 表示后向搜索) 时 *substr* 在 `unicode[start:end]` 中首次出现的位置。返回值为首个匹配的索引号；值为 `-1` 表示未找到匹配，`-2` 则表示发生了错误并设置了异常。

Py_ssize_t **PyUnicode_FindChar**(*PyObject* *unicode, *Py_UCS4* ch, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

属于稳定 ABI 自 *3.7* 版起. 返回使用给定的 *direction* (*direction* == `1` 表示前向搜索，*direction* == `-1` 表示后向搜索) 时字符 *ch* 在 `unicode[start:end]` 中首次出现的位置。返回值为首个匹配的索引号；值为 `-1` 表示未找到匹配，`-2` 则表示发生错误并设置了异常。

Added in version 3.3.

在 3.7 版本发生变更: 现在 *start* 和 *end* 被调整为与 `unicode[start:end]` 类似的行为。

Py_ssize_t **PyUnicode_Count**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end)

属于稳定 ABI. 返回 *substr* 在 `unicode[start:end]` 中不重叠出现的次数。如果发生错误则返回 `-1`。

PyObject ***PyUnicode_Replace**(*PyObject* *unicode, *PyObject* *substr, *PyObject* *replstr, *Py_ssize_t* maxcount)

返回值：新的引用。属于稳定 ABI. 将 *unicode* 中 *substr* 替换为 *replstr* 至多 *maxcount* 次并返回结果 Unicode 对象。*maxcount* == `-1` 表示全部替换。

int **PyUnicode_Compare**(*PyObject* *left, *PyObject* *right)

属于稳定 ABI. 比较两个字符串并返回 `-1`, `0`, `1` 分别表示小于、等于和大于。

此函数执行失败时返回 `-1`，因此应当调用*PyErr_Occurred()* 来检查错误。

int **PyUnicode_EqualToUTF8AndSize**(*PyObject* *unicode, const char *string, *Py_ssize_t* size)

属于稳定 ABI 自 *3.13* 版起. 将一个 Unicode 对象与一个按 UTF-8 或 ASCII 编码来解读的字符缓冲区进行比较并在两者相等时返回真值 (`1`)，否则返回假值 (`0`)。如果 Unicode 对象包含代理码位 (`U+D800 - U+DFFF`) 或者如果 C 字符串不是有效的 UTF-8 编码，则返回假值 (`0`)。

此函数不会引发异常。

Added in version 3.13.

int **PyUnicode_EqualToUTF8**(*PyObject* *unicode, const char *string)

属于稳定 ABI 自 *3.13* 版起. 类似于*PyUnicode_EqualToUTF8AndSize()*，但会使用 `strlen()` 来计算 *string* 的长度。如果 Unicode 对象包含空字符，则返回假值 (`0`)。

Added in version 3.13.

int **PyUnicode_CompareWithASCIIString**(*PyObject* *unicode, const char *string)

属于稳定 ABI. 将 Unicode 对象 *unicode* 与 *string* 进行比较并返回 `-1`, `0`, `1` 分别表示小于、等于和大于。最好只传入 ASCII 编码的字符串，但如果输入字符串包含非 ASCII 字符则此函数会将其按 ISO-8859-1 编码格式来解读。

此函数不会引发异常。

PyObject ***PyUnicode_RichCompare**(*PyObject* *left, *PyObject* *right, int op)

返回值：新的引用。属于稳定 ABI. 对两个 Unicode 字符串执行富比较并返回以下值之一:

- `NULL` 用于引发了异常的情况
- *Py_True* 或*Py_False* 用于成功完成比较的情况
- *Py_NotImplemented* 用于类型组合未知的情况

可能的 *op* 值有*Py_GT*, *Py_GE*, *Py_EQ*, *Py_NE*, *Py_LT*, 和*Py_LE*。

PyObject ***PyUnicode_Format**(*PyObject* *format, *PyObject* *args)

返回值：新的引用。属于稳定 ABI. 根据 *format* 和 *args* 返回一个新的字符串对象；这等同于 `format % args`。

int **PyUnicode_Contains**(*PyObject* *unicode, *PyObject* *substr)

属于稳定 ABI. 检查 *substr* 是否包含在 *unicode* 中并相应返回真值或假值。

substr 必须强制转为一个单元素 Unicode 字符串。如果发生错误则返回 `-1`。

void **PyUnicode_InternInPlace**(*PyObject* **p_unicode)

属于稳定 ABI. 原地内部化参数 **p_unicode*。该参数必须是一个指向 Python Unicode 字符串对象的指针变量的地址。如果已存在与 **p_unicode* 相同的内部化字符串，则将其设为 **p_unicode* (释放对旧字符串的引用并新建一个指向该内部化字符串对象的*strong reference*)，否则将保持 **p_unicode* 不变并将其内部化。

（澄清说明：虽然这里大量提及了引用，但请将此函数视为引用无关的。你必须拥有你传入的对象；在调用之后你将不再拥有传入的引用，但你将新拥有结果对象。）

此函数绝不会引发异常。在发生错误时，它将保持其参数不变而不会将其内部化。

`str` 的子类的实例不可被内部化，也就是说，*PyUnicode_CheckExact*(**p_unicode*) 必须为真值。如果其不为真值，那么 -- 就像发生其他错误时一样 -- 该参数将保持不变。

请注意被内部化的字符串不是“永生的”。你必须保留对结果的引用才能从内部化获益。

PyObject ***PyUnicode_InternFromString**(const char *str)

返回值：新的引用。属于稳定 ABI. *PyUnicode_FromString()* 和*PyUnicode_InternInPlace()* 的结合，用于静态分配的字符串。

返回一个新的（“拥有的”）引用，它指向一个已被内部化的新 Unicode 字符串，或一个具有相同值的先前已被内部化的字符串对象。

Python 可以保留一个指向结果的引用，或是使其成为*immortal* 对象，以防止其被立即被作为垃圾回收。对于内部化未限定数量的不同字符串，例如来自用户输入的字符串，建议直接调用*PyUnicode_FromString()* 和*PyUnicode_InternInPlace()*。

8.3.4 元组对象

type **PyTupleObject**

这个*PyObject* 的子类型代表一个 Python 的元组对象。

PyTypeObject **PyTuple_Type**

属于稳定 ABI. *PyTypeObject* 的实例代表一个 Python 元组类型，这与 Python 层面的 `tuple` 是相同的对象。

int **PyTuple_Check**(*PyObject* *p)

如果 *p* 是一个 tuple 对象或者 tuple 类型的子类型的实例则返回真值。此函数总是会成功执行。

int **PyTuple_CheckExact**(*PyObject* *p)

如果 *p* 是一个 tuple 对象但不是 tuple 类型的子类型的实例则返回真值。此函数总是会成功执行。

PyObject ***PyTuple_New**(*Py_ssize_t* len)

返回值：新的引用。属于稳定 ABI. 返回一个长度为 *len* 的新元组对象，或者失败时返回 `NULL` 并设置一个异常。

PyObject ***PyTuple_Pack**(*Py_ssize_t* n, ...)

返回值：新的引用。属于稳定 ABI. 返回一个长度为 *n* 的新元组对象，或者失败时返回 `NULL` 并设置一个异常。元组值初始化为指向 Python 对象的后续 *n* 个 C 参数。`PyTuple_Pack(2, a, b)` 等价于 `Py_BuildValue("(OO)", a, b)`。

Py_ssize_t **PyTuple_Size**(*PyObject* *p)

属于稳定 ABI. 接受一个指向元组对象的指针，并返回该元组的大小。失败时，返回 `-1` 并设置一个异常。

Py_ssize_t **PyTuple_GET_SIZE**(*PyObject* *p)

类似于*PyTuple_Size()*，但是不带错误检测。

PyObject ***PyTuple_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

返回值：借入的引用。属于稳定 ABI. 返回 *p* 所指向的元组中位于 *pos* 处的对象。如果 *pos* 为负值或超出范围，则返回 `NULL` 并设置一个 `IndexError` 异常。

返回的引用是从元组 *p* 借入的（也就是说：它只在你持有对 *p* 的引用时才是可用的）。要获取*strong reference*，请使用*Py_NewRef(PyTuple_GetItem(...))* 或*PySequence_GetItem()*。

PyObject ***PyTuple_GET_ITEM**(*PyObject* *p, *Py_ssize_t* pos)

返回值：借入的引用。 类似于*PyTuple_GetItem()*，但不检查其参数。

PyObject ***PyTuple_GetSlice**(*PyObject* *p, *Py_ssize_t* low, *Py_ssize_t* high)

返回值：新的引用。属于稳定 ABI. 返回一个 *p* 指向的元组的 *low* 和 *high* 之间的切片，或者失败时返回 `NULL` 并设置一个异常。

这等价于 Python 表达式 `p[low:high]`。不支持从元组末尾进行索引。

int **PyTuple_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

属于稳定 ABI. 在 *p* 指向的元组的 *pos* 位置插入对对象 *o* 的引用。成功时返回 `0`；如果 *pos* 越界，则返回 `-1`，并抛出一个 `IndexError` 异常。

> **备注**
>
> 此函数会“窃取”对 *o* 的引用，并丢弃对元组中已在受影响位置的条目的引用。

void **PyTuple_SET_ITEM**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

类似于*PyTuple_SetItem()*，但不进行错误检查，并且应该只是被用来填充全新的元组。

当 Python 以 调试模式或 启用断言构建时将把绑定检测作为断言来执行。

> **备注**
>
> 这个函数会“窃取”一个对 *o* 的引用，但是，不与*PyTuple_SetItem()* 不同，它 不会丢弃对任何被替换项的引用；元组中位于 *pos* 位置的任何引用都将被泄漏。

int **_PyTuple_Resize**(*PyObject* **p, *Py_ssize_t* newsize)

可以用于调整元组的大小。*newsize* 将是元组的新长度。因为元组 被认为是不可变的，所以只有在对象仅有一个引用时，才应该使用它。如果元组已经被代码的其他部分所引用，请不要使用此项。元组在最后总是会增长或缩小。把它看作是销毁旧元组并创建一个新元组，只会更有效。成功时返回 `0` 。客户端代码不应假定 `*p` 的结果值将与调用此函数之前的值相同。如果替换了 `*p` 引用的对象，则原始的 `*p` 将被销毁。失败时，返回 `-1`，将 `*p` 设置为 `NULL`，并引发 `MemoryError` 或者 `SystemError`。

8.3.5 结构序列对象

结构序列对象是等价于 `namedtuple()` 的 C 对象，即一个序列，其中的条目也可以通过属性访问。要创建结构序列，你首先必须创建特定的结构序列类型。

PyTypeObject ***PyStructSequence_NewType**(*PyStructSequence_Desc* *desc)

返回值：新的引用。属于稳定 ABI. 根据 *desc* 中的数据创建一个新的结构序列类型，如下所述。可以使用*PyStructSequence_New()* 创建结果类型的实例。

失败时返回 `NULL` 并设置一个异常。

void **PyStructSequence_InitType**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

从 *desc* 就地初始化结构序列类型 *type*。

int **PyStructSequence_InitType2**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

类似于*PyStructSequence_InitType()*，但成功时返回 `0` 而失败时返回 `-1` 并设置一个异常。

Added in version 3.4.

type **PyStructSequence_Desc**

属于稳定 ABI （包括所有成员）. 包含要创建的结构序列类型的元信息。

const char ***name**

类型的完整限定名称；使用以空值结束的 UTF-8 编码。该名称必须包含模块名。

const char ***doc**

指向类型的文档字符串的指针或以 `NULL` 表示忽略。

PyStructSequence_Field ***fields**

指向以 `NULL` 结尾的数组的指针，该数组包含新类型的字段名。

int **n_in_sequence**

Python 端可见的字段数（如果用作元组）。

type **PyStructSequence_Field**

属于稳定 ABI （包括所有成员）. 描述结构序列的一个字段。由于结构序列是以元组为模型的，因此所有字段的类型都是 *PyObject** 。*PyStructSequence_Desc* 的*fields* 数组中的索引决定了描述结构序列的是哪个字段。

const char ***name**

字段的名称或 `NULL` 表示结束已命名字段列表，设为*PyStructSequence_UnnamedField* 则保持未命名状态。

const char ***doc**

字段文档字符串或 `NULL` 表示省略。

const char *const **PyStructSequence_UnnamedField**

属于稳定 ABI 自 *3.11* 版起. 字段名的特殊值将保持未命名状态。

在 3.9 版本发生变更: 这个类型已从 `char *` 更改。

PyObject ***PyStructSequence_New**(*PyTypeObject* *type)

返回值：新的引用。属于稳定 ABI. 创建 *type* 的实例，该实例必须使用*PyStructSequence_NewType()* 创建。

失败时返回 `NULL` 并设置一个异常。

PyObject ***PyStructSequence_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

返回值：借入的引用。属于稳定 ABI. 返回 *p* 所指向的结构序列中位于 *pos* 处的对象。

当 Python 以 调试模式或 启用断言构建时将把绑定检测作为断言来执行。

PyObject ***PyStructSequence_GET_ITEM**(*PyObject* *p, *Py_ssize_t* pos)

返回值：借入的引用。 *PyStructSequence_GetItem()* 的别名。

在 3.13 版本发生变更: 现在被实现为*PyStructSequence_GetItem()* 的别名。

void **PyStructSequence_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

属于稳定 ABI. 将结构序列 *p* 的索引 *pos* 处的字段设置为值 *o*。与*PyTuple_SET_ITEM()* 一样，它应该只用于填充全新的实例。

当 Python 以 调试模式或 启用断言构建时将把绑定检测作为断言来执行。

> **备注**
>
> 这个函数“窃取”了指向 *o* 的一个引用。

void **PyStructSequence_SET_ITEM**(*PyObject* *p, *Py_ssize_t* *pos, *PyObject* *o)

PyStructSequence_SetItem() 的别名。

在 3.13 版本发生变更: 现在被实现为*PyStructSequence_SetItem()* 的别名。

8.3.6 列表对象

type **PyListObject**

这个 C 类型*PyObject* 的子类型代表一个 Python 列表对象。

PyTypeObject **PyList_Type**

属于稳定 ABI. 这是个属于*PyTypeObject* 的代表 Python 列表类型的实例。在 Python 层面和类型 `list` 是同一个对象。

int **PyList_Check**(*PyObject* *p)

如果 *p* 是一个 list 对象或者 list 类型的子类型的实例则返回真值。此函数总是会成功执行。

int **PyList_CheckExact**(*PyObject* *p)

如果 *p* 是一个 list 对象但不是 list 类型的子类型的实例则返回真值。此函数总是会成功执行。

PyObject ***PyList_New**(*Py_ssize_t* len)

返回值：新的引用。属于稳定 ABI. 成功时返回一个长度为 *len* 的新列表，失败时返回 `NULL`。

> **备注**
>
> 当 *len* 大于零时，被返回的列表对象的条目将被设为 `NULL`。因此你不能使用抽象 API 函数如*PySequence_SetItem()* 或者在使用*PyList_SetItem()* 或*PyList_SET_ITEM()* 将所有条目设为真实对象之前将对象暴露给 Python 代码。以下 API 在该列表完全初始化之前将是安全的 API: *PyList_SetItem()* 和*PyList_SET_ITEM()*。

Py_ssize_t **PyList_Size**(*PyObject* *list)

属于稳定 ABI. 返回 *list* 中列表对象的长度；这等于在列表对象调用 `len(list)`。

Py_ssize_t **PyList_GET_SIZE**(*PyObject* *list)

类似于*PyList_Size()*，但是不带错误检测。

PyObject ***PyList_GetItemRef**(*PyObject* *list, *Py_ssize_t* index)

返回值：新的引用。属于稳定 ABI 自 *3.13* 版起. 返回 *list* 所指向的列表中 *index* 位置上的对象。位置值必须为非负数；不支持从列表末尾反向索引。如果 *index* 超出范围 (`<0 or >=len(list)`)，则返回 `NULL` 并设置 `IndexError` 异常。

Added in version 3.13.

PyObject ***PyList_GetItem**(*PyObject* *list, *Py_ssize_t* index)

返回值：借入的引用。属于稳定 ABI. 类似于`PyList_GetItemRef()`，但返回一个*borrowed reference*而不是*strong reference*。

PyObject ***PyList_GET_ITEM**(*PyObject* *list, *Py_ssize_t* i)

返回值：借入的引用。 类似于`PyList_GetItem()`，但是不带错误检测。

int **PyList_SetItem**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

属于稳定 ABI. 将列表中索引为 *index* 的项设为 *item*。成功时返回 `0`。如果 *index* 超出范围则返回 `-1` 并设定 `IndexError` 异常。

> **备注**
>
> 此函数会“偷走”一个对 *item* 的引用并丢弃一个对列表中受影响位置上的已有条目的引用。

void **PyList_SET_ITEM**(*PyObject* *list, *Py_ssize_t* i, *PyObject* *o)

不带错误检测的宏版本`PyList_SetItem()`。这通常只被用于新列表中之前没有内容的位置进行填充。

当 Python 以 调试模式或 启用断言构建时将把绑定检测作为断言来执行。

> **备注**
>
> 该宏会“偷走”一个对 *item* 的引用，但与`PyList_SetItem()` 不同的是它 不会丢弃对任何被替换条目的引用；在 *list* 的 *i* 位置上的任何引用都将被泄露。

int **PyList_Insert**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

属于稳定 ABI. 将条目 *item* 插入到列表 *list* 索引号 *index* 之前的位置。如果成功将返回 `0`；如果不成功则返回 `-1` 并设置一个异常。相当于 `list.insert(index, item)`。

int **PyList_Append**(*PyObject* *list, *PyObject* *item)

属于稳定 ABI. 将对象 *item* 添加到列表 *list* 的末尾。如果成功将返回 `0`；如果不成功则返回 `-1` 并设置一个异常。相当于 `list.append(item)`。

PyObject ***PyList_GetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high)

返回值：新的引用。属于稳定 ABI. 返回一个对象列表，包含 *list* 当中位于 *low* 和 *high* 之间的对象。如果不成功则返回 `NULL` 并设置异常。相当于 `list[low:high]`。不支持从列表末尾进行索引。

int **PyList_SetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high, *PyObject* *itemlist)

属于稳定 ABI. 将 *list* 当中 *low* 与 *high* 之间的切片设为 *itemlist* 的内容。相当于 `list[low:high] = itemlist`。*itemlist* 可以为 `NULL`，表示赋值为一个空列表（删除切片）。成功时返回 `0`，失败时返回 `-1`。这里不支持从列表末尾进行索引。

int **PyList_Extend**(*PyObject* *list, *PyObject* *iterable)

使用 *iterable* 的内容扩展 *list*。这与 `PyList_SetSlice(list, PY_SSIZE_T_MAX, PY_SSIZE_T_MAX, iterable)` 相同并与 `list.extend(iterable)` 或 `list += iterable` 类似。

如果 *list* 不是 `list` 对象则会引发异常并返回 `-1`。成功时返回 0。

Added in version 3.13.

int **PyList_Clear**(*PyObject* *list)

从 *list* 移除所有条目。这与 `PyList_SetSlice(list, 0, PY_SSIZE_T_MAX, NULL)` 相同并与 `list.clear()` 或 `del list[:]` 类似。

如果 *list* 不是 `list` 对象则会引发异常并返回 `-1`。成功时返回 0。

Added in version 3.13.

int **PyList_Sort**(*PyObject* *list)

属于稳定 ABI. 对 *list* 中的条目进行原地排序。成功时返回 `0`，失败时返回 `-1`。这等价于 `list.sort()`。

int **PyList_Reverse**(*PyObject* *list)

属于稳定 ABI. 对 *list* 中的条目进行原地反转。成功时返回 `0`，失败时返回 `-1`。这等价于 `list.reverse()`。

PyObject ***PyList_AsTuple**(*PyObject* *list)

返回值：新的引用。属于稳定 ABI. 返回一个新的元组对象，其中包含 *list* 的内容；等价于 `tuple(list)`。

8.4 容器对象

8.4.1 字典对象

type **PyDictObject**

这个*PyObject* 的子类型代表一个 Python 字典对象。

PyTypeObject **PyDict_Type**

属于稳定 ABI. Python 字典类型表示为*PyTypeObject* 的实例。这与 Python 层面的 `dict` 是相同的对象。

int **PyDict_Check**(*PyObject* *p)

如果 *p* 是一个 dict 对象或者 dict 类型的子类型的实例则返回真值。此函数总是会成功执行。

int **PyDict_CheckExact**(*PyObject* *p)

如果 *p* 是一个 dict 对象但不是 dict 类型的子类型的实例则返回真值。此函数总是会成功执行。

PyObject ***PyDict_New**()

返回值：新的引用。属于稳定 ABI. 返回一个新的空字典，失败时返回 `NULL`。

PyObject ***PyDictProxy_New**(*PyObject* *mapping)

返回值：新的引用。属于稳定 ABI. 返回 `types.MappingProxyType` 对象，用于强制执行只读行为的映射。这通常用于创建视图以防止修改非动态类类型的字典。

void **PyDict_Clear**(*PyObject* *p)

属于稳定 ABI. 清空现有字典的所有键值对。

int **PyDict_Contains**(*PyObject* *p, *PyObject* *key)

属于稳定 ABI. 确定 *key* 是否包含在字典 *p* 中。如果 *key* 匹配上 *p* 的某一项，则返回 `1`，否则返回 `0`。返回 `-1` 表示出错。这等同于 Python 表达式 `key in p`。

int **PyDict_ContainsString**(*PyObject* *p, const char *key)

这与*PyDict_Contains()* 相同，但 *key* 被指定为一个 `const char*` UTF-8 编码的字节串，而不是*PyObject**。

Added in version 3.13.

PyObject ***PyDict_Copy**(*PyObject* *p)

返回值：新的引用。属于稳定 ABI. 返回与 *p* 包含相同键值对的新字典。

int **PyDict_SetItem**(*PyObject* *p, *PyObject* *key, *PyObject* *val)

属于稳定 ABI. 使用 *key* 作为键将 *val* 插入字典 *p*。*key* 必须为*hashable*；如果不是，则将引发 `TypeError`。成功时返回 `0`，失败时返回 `-1`。此函数不会窃取对 *val* 的引用。

int **PyDict_SetItemString**(*PyObject* *p, const char *key, *PyObject* *val)

属于稳定 ABI. 这与*PyDict_SetItem()* 相同，但 *key* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

int **PyDict_DelItem**(*PyObject* *p, *PyObject* *key)

属于稳定 ABI. 移除字典 *p* 中键为 *key* 的条目。*key* 必须是*hashable*；如果不是，则会引发 `TypeError`。如果字典中没有 *key*，则会引发 `KeyError`。成功时返回 `0` 或者失败时返回 `-1`。

int **PyDict_DelItemString**(*PyObject* *p, const char *key)

属于稳定 ABI. 这与*PyDict_DelItem()* 相同，但 *key* 被指定为 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

int **PyDict_GetItemRef**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

属于稳定 ABI 自 *3.13* 版起. 返回一个新的指向字典 *p* 中对应键 *key* 的对象的*strong reference*:

- 如果存在该键，则将 **result* 设为一个新的指向该值的*strong reference* 并返回 `1`。
- 如果不存在该键，则将 **result* 设为 `NULL` 并返回 `0`。
- 发生错误时，将引发异常并返回 `-1`。

Added in version 3.13.

另请参阅*PyObject_GetItem()* 函数。

PyObject ***PyDict_GetItem**(*PyObject* *p, *PyObject* *key)

返回值：借入的引用。属于稳定 ABI. 返回一个指向字典 *p* 中对应键 *key* 的对象的*borrowed reference*。如果不存在键 *key* 则返回 `NULL` 且 不会设置异常。

> **备注**
>
> 在调用 `__hash__()` 和 `__eq__()` 方法时发生的异常将被静默地忽略。建议改用*PyDict_GetItemWithError()* 函数。

在 3.10 版本发生变更: 在不保持*GIL* 的情况下调用此 API 曾因历史原因而被允许。现在已不再被允许。

PyObject ***PyDict_GetItemWithError**(*PyObject* *p, *PyObject* *key)

返回值：借入的引用。属于稳定 ABI. *PyDict_GetItem()* 的变种，它不会屏蔽异常。当异常发生时将返回 `NULL` **并且**设置一个异常。如果键不存在则返回 `NULL` **并且不会**设置一个异常。

PyObject ***PyDict_GetItemString**(*PyObject* *p, const char *key)

返回值：借入的引用。属于稳定 ABI. 这与*PyDict_GetItem()* 一样，但 *key* 是由一个 `const char*` UTF-8 编码的字节串来指定的，而不是 *PyObject**。

> **备注**
>
> 在调用 `__hash__()` 和 `__eq__()` 方法时或者在创建临时 `str` 对象期间发生的异常将被静默地忽略。建议改用*PyDict_GetItemWithError()* 函数并附带你自己的*PyUnicode_FromString()* *key*。

int **PyDict_GetItemStringRef**(*PyObject* *p, const char *key, *PyObject* **result)

属于稳定 ABI 自 *3.13* 版起. 类似于*PyDict_GetItemRef()*，但 *key* 被指定为一个 `const char*` UTF-8 编码的字节串，而不是 *PyObject**。

Added in version 3.13.

PyObject ***PyDict_SetDefault**(*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

返回值：借入的引用。 这跟 Python 层面的 `dict.setdefault()` 一样。如果键 *key* 存在，它返回在字典 *p* 里面对应的值。如果键不存在，它会和值 *defaultobj* 一起插入并返回 *defaultobj* 。这个函数只计算 *key* 的哈希函数一次，而不是在查找和插入时分别计算它。

Added in version 3.4.

int **PyDict_SetDefaultRef**(*PyObject* *p, *PyObject* *key, *PyObject* *default_value, *PyObject* **result)

如果键 *key* 在字典 *p* 中尚不存在则将该键和值 *default_value* 插入到该字典中。如果 *result* 不为 `NULL`，那么当该键不存在时将 **result* 设为指向 *default_value* 的*strong reference*，或者当 *key* 已存在于该字典中时将其设为原有的值。如果该键已存在并且未插入 *default_value* 则返回 `1`，或者如果如果该键不存在并且已插入 *default_value* 则返回 `0`。当执行失败时，将返回 `-1`，设置一个异常，并将 `*result` 设为 `NULL`。

澄清一点：如果你在调用此函数前持有指向 *default_value* 的强引用，那么在它返回之后，你将同时持有指向 *default_value* 和 **result* (如果它不为 `NULL`) 的强引用。两者可能指向同一个对象：在此情况下你将持有两个指向它的单独引用。

Added in version 3.13.

int **PyDict_Pop**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

从字典 *p* 中移除 *key* 并可选择返回被移除的值。当键不存在时不会引发 `KeyError`。

- 如果键存在，则在 *result* 不为 `NULL` 时将 **result* 设为一个新的指向被移除值的引用，并返回 `1`。
- 如果不存在该键，则在 *result* 不为 `NULL` 时将 **result* 设为 `NULL`，并返回 `0`。
- 发生错误时，将引发异常并返回 `-1`。

类似于 `dict.pop()`，但没有默认值并且当键不存在时不会引发 `KeyError`。

Added in version 3.13.

int **PyDict_PopString**(*PyObject* *p, const char *key, *PyObject* **result)

类似于*PyDict_Pop()*，但 *key* 是以一个 `const char*` UTF-8 编码的字节串形式指定的，而不是 *PyObject**。

Added in version 3.13.

PyObject ***PyDict_Items**(*PyObject* *p)

返回值：新的引用。属于稳定 ABI. 返回一个包含字典中所有键值项的*PyListObject*。

PyObject ***PyDict_Keys**(*PyObject* *p)

返回值：新的引用。属于稳定 ABI. 返回一个包含字典中所有键 (keys) 的*PyListObject*。

PyObject ***PyDict_Values**(*PyObject* *p)

返回值：新的引用。属于稳定 ABI. 返回一个包含字典中所有值 (values) 的*PyListObject*。

Py_ssize_t **PyDict_Size**(*PyObject* *p)

属于稳定 ABI. 返回字典中项目数，等价于对字典 *p* 使用 `len(p)`。

int **PyDict_Next**(*PyObject* *p, *Py_ssize_t* *ppos, *PyObject* **pkey, *PyObject* **pvalue)

属于稳定 ABI. 迭代字典 *p* 中的所有键值对。在第一次调用此函数开始迭代之前，由 *ppos* 所引用的*Py_ssize_t* 必须被初始化为 `0`；该函数将为字典中的每个键值对返回真值，一旦所有键值对都报告完毕则返回假值。形参 *pkey* 和 *pvalue* 应当指向 *PyObject** 变量，它们将分别使用每个键和值来填充，或者也可以为 `NULL`。通过它们返回的任何引用都是暂借的。*ppos* 在迭代期间不应被更改。它的值表示内部字典结构中的偏移量，并且由于结构是稀疏的，因此偏移量并不连续。

例如：

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* 用这些值做些有趣的事... */
    ...
}
```

字典 *p* 不应该在遍历期间发生改变。在遍历字典时，改变键中的值是安全的，但仅限于键的集合不发生改变。例如:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    long i = PyLong_AsLong(value);
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}
```

此函数在没有外部同步的自由线程编译版中不是线程安全的。你可以使用*Py_BEGIN_CRITICAL_SECTION*在迭代字典时锁定它:

```
Py_BEGIN_CRITICAL_SECTION(self->dict);
while (PyDict_Next(self->dict, &pos, &key, &value)) {
    ...
}
Py_END_CRITICAL_SECTION();
```

int **PyDict_Merge**(*PyObject* *a, *PyObject* *b, int override)

属于稳定 ABI. 对映射对象 *b* 进行迭代，将键值对添加到字典 *a*。*b* 可以是一个字典，或任何支持*PyMapping_Keys()* 和*PyObject_GetItem()* 的对象。如果 *override* 为真值，则如果在 *b* 中找到相同的键则 *a* 中已存在的相应键值对将被替换，否则如果在 *a* 中没有相同的键则只是添加键值对。当成功时返回 `0` 或者当引发异常时返回 `-1`。

int **PyDict_Update**(*PyObject* *a, *PyObject* *b)

属于稳定 ABI. 这与 C 中的 `PyDict_Merge(a, b, 1)` 一样，也类似于 Python 中的 `a.update(b)`，差别在于*PyDict_Update()* 在第二个参数没有"keys" 属性时不会回退到迭代键值对的序列。当成功时返回 `0` 或者当引发异常时返回 `-1`。

int **PyDict_MergeFromSeq2**(*PyObject* *a, *PyObject* *seq2, int override)

属于稳定 ABI. 将 *seq2* 中的键值对更新或合并到字典 *a*。*seq2* 必须为产生长度为 2 的用作键值对的元素的可迭代对象。当存在重复的键时，如果 *override* 真值则最后出现的键胜出。当成功时返回 `0` 或者当引发异常时返回 `-1`。等价的 Python 代码（返回值除外）：

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

int **PyDict_AddWatcher**(*PyDict_WatchCallback* callback)

在字典上注册 *callback* 来作为 watcher。返回值为非负数的整数 id，作为将来调用*PyDict_Watch()* 的时候使用。如果出现错误（比如没有足够的可用 watcher ID），返回 `-1` 并且设置异常。

Added in version 3.12.

int **PyDict_ClearWatcher**(int watcher_id)

清空由之前从*PyDict_AddWatcher()* 返回的 *watcher_id* 所标识的 watcher。成功时返回 `0`，出错时（例如当给定的 *watcher_id* 未被注册）返回 `-1`。

Added in version 3.12.

int **PyDict_Watch**(int watcher_id, *PyObject* *dict)

将字典 *dict* 标记为已被监视。由*PyDict_AddWatcher()* 授权 *watcher_id* 对应的回调将在 *dict* 被修改或释放时被调用。成功时返回 0，出错时返回 -1。

Added in version 3.12.

int **PyDict_Unwatch**(int watcher_id, *PyObject* *dict)

将字典 *dict* 标记为不再被监视。由*PyDict_AddWatcher()* 授权 *watcher_id* 对应的回调在 *dict* 被修改或释放时将不再被调用。该字典在此之前必须已被此监视器所监视。成功时返回 0，出错时返回 -1。

Added in version 3.12.

type **PyDict_WatchEvent**

由以下可能的字典监视器事件组成的枚举：PyDict_EVENT_ADDED, PyDict_EVENT_MODIFIED, PyDict_EVENT_DELETED, PyDict_EVENT_CLONED, PyDict_EVENT_CLEARED 或 PyDict_EVENT_DEALLOCATED。

Added in version 3.12.

typedef int (***PyDict_WatchCallback**)(*PyDict_WatchEvent* event, *PyObject* *dict, *PyObject* *key, *PyObject* *new_value)

字典监视器回调函数的类型。

如果 *event* 是 PyDict_EVENT_CLEARED 或 PyDict_EVENT_DEALLOCATED，则 *key* 和 *new_value* 都将为 NULL。如果 *event* 是 PyDict_EVENT_ADDED 或 PyDict_EVENT_MODIFIED，则 *new_value* 将为 *key* 的新值。如果 *event* 是 PyDict_EVENT_DELETED，则将从字典中删除 *key* 而 *new_value* 将为 NULL。

PyDict_EVENT_CLONED 会在另一个字典合并到之前为空的 *dict* 时发生。为保证此操作的效率，该场景不会发出针对单个键的 PyDict_EVENT_ADDED 事件；而是发出单个 PyDict_EVENT_CLONED，而 *key* 将为源字典。

该回调可以检查但不能修改 *dict*；否则会产生不可预料的影响，包括无限递归。请不要在该回调中触发 Python 代码的执行，因为它可能产生修改 dict 的附带影响。

如果 *event* 是 PyDict_EVENT_DEALLOCATED，则在回调中接受一个对即将销毁的字典的新引用将使其重生并阻止其在此时被释放。当重生的对象以后再被销毁时，任何在当时已激活的监视器回调将再次被调用。

回调会在已通知的对 *dict* 的修改完成之前执行，这样在此之前的 *dict* 状态可以被检查。

如果该回调设置了一个异常，则它必须返回 -1；此异常将作为不可引发的异常使用*PyErr_WriteUnraisable()* 打印出来。在其他情况下它应当返回 0。

在进入回调时可能已经设置了尚未处理的异常。在此情况下，回调应当返回 0 并仍然设置同样的异常。这意味着该回调可能不会调用任何其他可设置异常的 API 除非它先保存并清空异常状态，并在返回之前恢复它。

Added in version 3.12.

8.4.2 集合对象

这一节详细介绍了针对 set 和 frozenset 对象的公共 API。任何未在下面列出的功能最好是使用抽象对象协议（包括*PyObject_CallMethod()*, *PyObject_RichCompareBool()*, *PyObject_Hash()*, *PyObject_Repr()*, *PyObject_IsTrue()*, *PyObject_Print()* 以及*PyObject_GetIter()*）或者抽象数字协议（包括*PyNumber_And()*, *PyNumber_Subtract()*, *PyNumber_Or()*, *PyNumber_Xor()*, *PyNumber_InPlaceAnd()*, *PyNumber_InPlaceSubtract()*, *PyNumber_InPlaceOr()* 以及*PyNumber_InPlaceXor()*）。

type **PySetObject**

这个*PyObject* 的子类型被用来保存 set 和 frozenset 对象的内部数据。它类似于*PyDictObject* 的地方在于对小尺寸集合来说它是固定大小的（很像元组的存储方式），而对于中等和大尺寸集合来说它将指向单独的可变大小的内存块（很像列表的存储方式）。此结构体的字段不应被视为公有并且可能发生改变。所有访问都应当通过已写入文档的 API 来进行而不可通过直接操纵结构体中的值。

PyTypeObject **PySet_Type**

属于稳定 ABI. 这是一个*PyTypeObject* 实例，表示 Python `set` 类型。

PyTypeObject **PyFrozenSet_Type**

属于稳定 ABI. 这是一个*PyTypeObject* 实例，表示 Python `frozenset` 类型。

下列类型检查宏适用于指向任意 Python 对象的指针。类似地，这些构造函数也适用于任意可迭代的 Python 对象。

int **PySet_Check**(*PyObject* *p)

如果 *p* 是一个 `set` 对象或者是其子类型的实例则返回真值。此函数总是会成功执行。

int **PyFrozenSet_Check**(*PyObject* *p)

如果 *p* 是一个 `frozenset` 对象或者是其子类型的实例则返回真值。此函数总是会成功执行。

int **PyAnySet_Check**(*PyObject* *p)

如果 *p* 是一个 `set` 对象、`frozenset` 对象或者是其子类型的实例则返回真值。此函数总是会成功执行。

int **PySet_CheckExact**(*PyObject* *p)

如果 *p* 是一个 `set` 对象但不是其子类型的实例则返回真值。此函数总是会成功执行。

Added in version 3.10.

int **PyAnySet_CheckExact**(*PyObject* *p)

如果 *p* 是一个 `set` 或 `frozenset` 对象但不是其子类型的实例则返回真值。此函数总是会成功执行。

int **PyFrozenSet_CheckExact**(*PyObject* *p)

如果 *p* 是一个 `frozenset` 对象但不是其子类型的实例则返回真值。此函数总是会成功执行。

PyObject ***PySet_New**(*PyObject* *iterable)

返回值：新的引用。属于稳定 ABI. 返回一个新的 `set`，其中包含 *iterable* 所返回的对象。*iterable* 可以为 `NULL` 表示创建一个新的空集合。成功时返回新的集合，失败时返回 `NULL`。如果 *iterable* 实际上不是可迭代对象则引发 `TypeError`。该构造器也适用于拷贝集合 (`c=set(s)`)。

PyObject ***PyFrozenSet_New**(*PyObject* *iterable)

返回值：新的引用。属于稳定 ABI. 返回一个新的 `frozenset`，其中包含 *iterable* 所返回的对象。*iterable* 可以为 `NULL` 表示创建一个新的空冻结集合。成功时返回新的冻结集合，失败时返回 `NULL`。如果 *iterable* 实际上不是可迭代对象则引发 `TypeError`。

下列函数和宏适用于 `set` 或 `frozenset` 的实例或是其子类型的实例。

Py_ssize_t **PySet_Size**(*PyObject* *anyset)

属于稳定 ABI. 返回 `set` 或 `frozenset` 对象的长度。等同于 `len(anyset)`。如果 *anyset* 不是 `set`, `frozenset` 或其子类型的实例，则会引发 `SystemError`。

Py_ssize_t **PySet_GET_SIZE**(*PyObject* *anyset)

宏版本的*PySet_Size()*，不带错误检测。

int **PySet_Contains**(*PyObject* *anyset, *PyObject* *key)

属于稳定 ABI. 如果找到则返回 `1`，如果未找到则返回 `0`，如果遇到错误则返回 `-1`。与 Python `__contains__()` 方法不同，该函数不会自动将不可哈希的集合转换为临时冻结集合。如果 *key* 是不可哈希对象则会引发 `TypeError`。如果 *anyset* 不是 `set`, `frozenset` 或其子类型的实例则会引发 `SystemError`。

int **PySet_Add**(*PyObject* *set, *PyObject* *key)

属于稳定 ABI. 添加 *key* 到一个 `set` 实例。也可用于 `frozenset` 实例（与*PyTuple_SetItem()* 的类似之处是它也可被用来为全新的冻结集合在公开给其他代码之前填充全新的值）。成功时返回 `0` 而失败时返回 `-1`。如果 *key* 为不可哈希对象则会引发 `TypeError`。如果没有增长空间则会引发 `MemoryError`。如果 *set* 不是 `set` 或其子类型的实例则会引发 `SystemError`。

下列函数适用于 `set` 或其子类型的实例，但不可用于 `frozenset` 或其子类型的实例。

int **PySet_Discard**(*PyObject* *set, *PyObject* *key)

属于稳定 ABI. 如果找到并已删除则返回 1，如未找到（无操作）则返回 0，如果遇到错误则返回 -1。对于不存在的键不会引发 KeyError。如果 *key* 为不可哈希对象则会引发 TypeError。与 Python discard() 方法不同，该函数不会自动将不可哈希的集合转换为临时的冻结集合。如果 *set* 不是 set 或其子类的实例则会引发 SystemError。

PyObject ***PySet_Pop**(*PyObject* *set)

返回值：新的引用。属于稳定 ABI. 返回 *set* 中任意对象的新引用，并从 *set* 中移除该对象。失败时返回 NULL。如果集合为空则会引发 KeyError。如果 *set* 不是 set 或其子类型的实例则会引发 SystemError。

int **PySet_Clear**(*PyObject* *set)

属于稳定 ABI. 清空现有的所有元素的集合。成功时返回 0。如果 *set* 不是 set 或其子类型的实际则返回 -1 并引发 SystemError。

8.5 Function 对象

8.5.1 Function 对象

有一些特定于 Python 函数的函数。

type **PyFunctionObject**

用于函数的 C 结构体。

PyTypeObject **PyFunction_Type**

这是一个*PyTypeObject* 实例并表示 Python 函数类型。它作为 types.FunctionType 向 Python 程序员公开。

int **PyFunction_Check**(*PyObject* *o)

如果 *o* 是一个函数对象 (类型为*PyFunction_Type*) 则返回真值。形参必须不为 NULL。此函数总是会成功执行。

PyObject ***PyFunction_New**(*PyObject* *code, *PyObject* *globals)

返回值：新的引用。 返回与代码对象 *code* 关联的新函数对象。*globals* 必须是一个字典，该函数可以访问全局变量。

函数的文档字符串和名称是从代码对象中提取的。__module__ 是从 *globals* 中提取的。参数 defaults, annotations 和 closure 被设为 NULL。__qualname__ 被设为与代码对象的 co_qualname 字段相同的值。

PyObject ***PyFunction_NewWithQualName**(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

返回值：新的引用。 类似*PyFunction_New()*，但还允许设置函数对象的 __qualname__ 属性。*qualname* 应当是一个 unicode 对象或为 NULL；如为 NULL，则 __qualname__ 属性会被设为与代码对象的 co_qualname 字段相同的值。

Added in version 3.3.

PyObject ***PyFunction_GetCode**(*PyObject* *op)

返回值：借入的引用。 返回与函数对象 *op* 关联的代码对象。

PyObject ***PyFunction_GetGlobals**(*PyObject* *op)

返回值：借入的引用。 返回与函数对象 *op* 相关联的全局字典。

PyObject ***PyFunction_GetModule**(*PyObject* *op)

返回值：借入的引用。 向 函数对象 *op* 的 __module__ 属性返回一个*borrowed reference*。该值可以为 *NULL*。

这通常为一个包含模块名称的 字符串，但可以通过 Python 代码设为任何其他对象。

PyObject ***PyFunction_GetDefaults**(*PyObject* *op)

返回值：借入的引用。 返回函数对象 *op* 的参数默认值。这可以是一个参数元组或 NULL。

int **PyFunction_SetDefaults**(*PyObject* *op, *PyObject* *defaults)

为函数对象 *op* 设置参数默认值。*defaults* 必须为 `Py_None` 或一个元组。

失败时引发 `SystemError` 异常并返回 `-1` 。

void **PyFunction_SetVectorcall**(*PyFunctionObject* *func, *vectorcallfunc* vectorcall)

设置给定函数对象 *func* 的 vectorcall 字段。

警告：使用此 API 的扩展必须保留未修改的（默认）vectorcall 函数的行为！

Added in version 3.12.

PyObject ***PyFunction_GetKwDefaults**(*PyObject* *op)

返回值：借入的引用。返回函数对象 *op* 的仅限关键字参数默认值。这可以是一个参数元组或 `NULL`。

PyObject ***PyFunction_GetClosure**(*PyObject* *op)

返回值：借入的引用。 返回关联到函数对象 *op* 的闭包。这可以是 `NULL` 或 cell 对象的元组。

int **PyFunction_SetClosure**(*PyObject* *op, *PyObject* *closure)

设置关联到函数对象 *op* 的闭包。*closure* 必须为 `Py_None` 或 cell 对象的元组。

失败时引发 `SystemError` 异常并返回 `-1` 。

PyObject ***PyFunction_GetAnnotations**(*PyObject* *op)

返回值：借入的引用。 返回函数对象 *op* 的标注。这可以是一个可变字典或 `NULL`。

int **PyFunction_SetAnnotations**(*PyObject* *op, *PyObject* *annotations)

设置函数对象 *op* 的标注。*annotations* 必须为一个字典或 `Py_None`。

失败时引发 `SystemError` 异常并返回 `-1` 。

PyObject ***PyFunction_GET_CODE**(*PyObject* *op)

PyObject ***PyFunction_GET_GLOBALS**(*PyObject* *op)

PyObject ***PyFunction_GET_MODULE**(*PyObject* *op)

PyObject ***PyFunction_GET_DEFAULTS**(*PyObject* *op)

PyObject ***PyFunction_GET_KW_DEFAULTS**(*PyObject* *op)

PyObject ***PyFunction_GET_CLOSURE**(*PyObject* *op)

PyObject ***PyFunction_GET_ANNOTATIONS**(*PyObject* *op)

返回值：借入的引用。 这些函数与它们的 `PyFunction_Get*` 对应物类似，但不会进行类型检查。传入*PyFunction_Type* 的实例以外的任何东西都是未定义的行为。

int **PyFunction_AddWatcher**(*PyFunction_WatchCallback* callback)

注册 *callback* 作为当前解释器的函数监视器。返回一个可被传给*PyFunction_ClearWatcher()* 的 ID。如果出现错误（比如没有足够的可用监视器 ID），则返回 `-1` 并设置一个异常。

Added in version 3.12.

int **PyFunction_ClearWatcher**(int watcher_id)

清空当前解释器在之前从 Clear watcher identified by previously returned from *PyFunction_AddWatcher()* 返回的由 *watcher_id* 所标识的监视器。成功时返回 `0`，或者出错时（比如当给定的 *watcher_id* 未被注册）返回 `-1` 并设置一个异常。

Added in version 3.12.

type **PyFunction_WatchEvent**

由可能的函数监视事件组成的枚举：

- `PyFunction_EVENT_CREATE`
- `PyFunction_EVENT_DESTROY`
- `PyFunction_EVENT_MODIFY_CODE`
- `PyFunction_EVENT_MODIFY_DEFAULTS`

- `PyFunction_EVENT_MODIFY_KWDEFAULTS`

Added in version 3.12.

typedef int (***PyFunction_WatchCallback**)(*PyFunction_WatchEvent* event, *PyFunctionObject* *func, *PyObject* *new_value)

函数监视器回调函数的类型。

如果 *event* 为 `PyFunction_EVENT_CREATE` 或 `PyFunction_EVENT_DESTROY` 则 *new_value* 将为 `NULL`。在其他情况下，*new_value* 将为被修改的属性持有一个指向要保存在 *func* 中的新值的*borrowed reference*。

该回调可以检查但不能修改 *func*; 这样做可能具有不可预知的影响，包括无限递归。

如果 *event* 是 `PyFunction_EVENT_CREATE`，则回调会在 *func* 完成初始化之后被唤起。在其他情况下，回调会在对 *func* 的修改执行之前被唤起，这样就可以对 *func* 之前的状态进行检查。如有可能函数对象的创建允许被运行时执行优化。在此情况下将不发出任何事件。虽然根据不同的优化决定这会产生可被观察到的运行时行为变化，但是它不会改变所执行的 Python 代码的语义。

如果 *event* 是 `PyFunction_EVENT_DESTROY`，则在回调中接受一个即将销毁的函数的引用将使其重生，并阻止其在此时被释放。当重生的对象以后再被销毁时，任何在当时已激活的监视器回调将再次被调用。

如果该回调设置了一个异常，则它必须返回 `-1`；此异常将作为不可引发的异常使用*PyErr_WriteUnraisable()* 打印出来。在其他情况下它应当返回 `0`。

在进入回调时可能已经设置了尚未处理的异常。在此情况下，回调应当返回 `0` 并仍然设置同样的异常。这意味着该回调可能不会调用任何其他可设置异常的 API 除非它先保存并清空异常状态，并在返回之前恢复它。

Added in version 3.12.

8.5.2 实例方法对象

实例方法是*PyCFunction* 的包装器，也是将*PyCFunction* 与类对象绑定的新方法。它取代了以前的调用 `PyMethod_New(func, NULL, class)`。

PyTypeObject **PyInstanceMethod_Type**

这个*PyTypeObject* 实例代表 Python 实例方法类型。它并不对 Python 程序公开。

int **PyInstanceMethod_Check**(*PyObject* *o)

如果 *o* 是一个实例方法对象 (类型为*PyInstanceMethod_Type*) 则返回真值。形参必须不为 `NULL`。此函数总是会成功执行。

PyObject ***PyInstanceMethod_New**(*PyObject* *func)

返回值：新的引用。 返回一个新的实例方法对象，*func* 应为任意可调用对象。*func* 将在实例方法被调用时作为函数被调用。

PyObject ***PyInstanceMethod_Function**(*PyObject* *im)

返回值：借入的引用。 返回关联到实例方法 *im* 的函数对象。

PyObject ***PyInstanceMethod_GET_FUNCTION**(*PyObject* *im)

返回值：借入的引用。 宏版本的*PyInstanceMethod_Function()*，略去了错误检测。

8.5.3 方法对象

方法是绑定的函数对象。方法总是会被绑定到一个用户自定义类的实例。未绑定方法（绑定到一个类的方法）已不再可用。

PyTypeObject **PyMethod_Type**

这个*PyTypeObject* 实例代表 Python 方法类型。它作为 `types.MethodType` 向 Python 程序公开。

int **PyMethod_Check**(*PyObject* *o)

如果 *o* 是一个方法对象 (类型为*PyMethod_Type*) 则返回真值。形参必须不为 NULL。此函数总是会成功执行。

PyObject ***PyMethod_New**(*PyObject* *func, *PyObject* *self)

返回值：新的引用。 返回一个新的方法对象，*func* 应为任意可调用对象，*self* 为该方法应绑定的实例。在方法被调用时 *func* 将作为函数被调用。*self* 必须不为 NULL。

PyObject ***PyMethod_Function**(*PyObject* *meth)

返回值：借入的引用。 返回关联到方法 *meth* 的函数对象。

PyObject ***PyMethod_GET_FUNCTION**(*PyObject* *meth)

返回值：借入的引用。 宏版本的*PyMethod_Function()*，略去了错误检测。

PyObject ***PyMethod_Self**(*PyObject* *meth)

返回值：借入的引用。 返回关联到方法 *meth* 的实例。

PyObject ***PyMethod_GET_SELF**(*PyObject* *meth)

返回值：借入的引用。 宏版本的*PyMethod_Self()*，略去了错误检测。

8.5.4 Cell 对象

“Cell” 对象用于实现由多个作用域引用的变量。对于每个这样的变量，一个 “Cell” 对象为了存储该值而被创建；引用该值的每个堆栈框架的局部变量包含同样使用该变量的对外部作用域的 “Cell” 引用。访问该值时，将使用 “Cell” 中包含的值而不是单元格对象本身。这种对 “Cell” 对象的非关联化的引用需要支持生成的字节码；访问时不会自动非关联化这些内容。“Cell” 对象在其他地方可能不太有用。

type **PyCellObject**

用于 Cell 对象的 C 结构体。

PyTypeObject **PyCell_Type**

与 Cell 对象对应的类型对 象。

int **PyCell_Check**(*PyObject* *ob)

如果 *ob* 是一个 cell 对象则返回真值；*ob* 必须不为 NULL。此函数总是会成功执行。

PyObject ***PyCell_New**(*PyObject* *ob)

返回值：新的引用。 创建并返回一个包含值 *ob* 的新 cell 对象。形参可以为 NULL。

PyObject ***PyCell_Get**(*PyObject* *cell)

返回值：新的引用。 返回 cell 对象 *cell* 的内容，可以为 NULL。如果 *cell* 不是一个 cell 对象，则返回 NULL 并设置一个异常。

PyObject ***PyCell_GET**(*PyObject* *cell)

返回值：借入的引用。 返回 cell 对象 *cell* 的内容，但是不检测 *cell* 是否非 NULL 并且为一个 cell 对象。

int **PyCell_Set**(*PyObject* *cell, *PyObject* *value)

将 cell 对象 *cell* 的内容设为 *value*。这将释放任何对该 cell 对象当前内容的引用。*value* 可以为 NULL。*cell* 必须不为 NULL。

当成功时，返回 0。如果 *cell* 不是一个 cell 对象，则设置一个异常并返回 -1。

void **PyCell_SET**(*PyObject* *cell, *PyObject* *value)

将 cell 对象 *cell* 的值设为 *value*。不会调整引用计数，并且不会进行检测以保证安全；*cell* 必须为非 NULL 并且为一个 cell 对象。

8.5.5 代码对象

代码对象是 CPython 实现的低层级细节。每个代表一块尚未绑定到函数中的可执行代码。

type **PyCodeObject**

用于描述代码对象的对象的 C 结构。此类型字段可随时更改。

PyTypeObject **PyCode_Type**

这一个代表 Python 代码对象的*PyTypeObject* 实例。

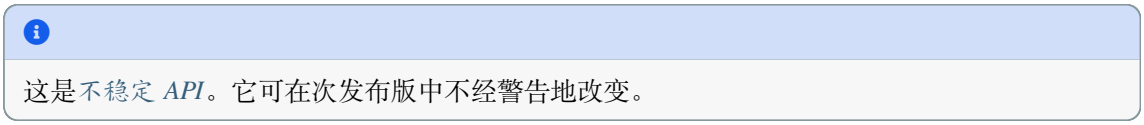
int **PyCode_Check**(*PyObject* *co)

如果 *co* 是一个 代码对象则返回真值。此函数总是会成功执行。

Py_ssize_t **PyCode_GetNumFree**(*PyCodeObject* *co)

返回代码对象中*自由（闭包）变量* 的数量。

int **PyUnstable_Code_GetFirstFree**(*PyCodeObject* *co)

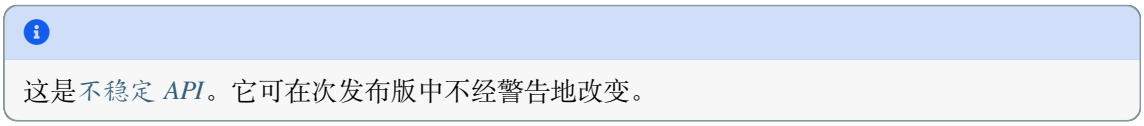


这是*不稳定 API*。它可在次发布版中不经警告地改变。

返回代码对象中第一个*自由（闭包）变量* 的位置。

在 3.13 版本发生变更: 作为*不稳定 C API* 的一部分由 PyCode_GetFirstFree 更名而来。旧名称已被弃用，但在签名再次更改之前将保持可用。

PyCodeObject ***PyUnstable_Code_New**(int argcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)



这是*不稳定 API*。它可在次发布版中不经警告地改变。

返回一个新的代码对象。如果你需要一个用空代码对象来创建帧，请改用*PyCode_NewEmpty()*。

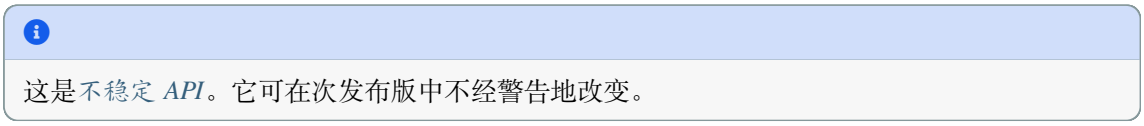
由于字节码的定义经常变化，可以直接调用*PyUnstable_Code_New()* 来绑定某个确定的 Python 版本。

此函数的许多参数以复杂的方式相互依赖，这意味着参数值的细微改变可能导致不正确的执行或 VM 崩溃。使用此函数需要极度小心。

在 3.11 版本发生变更: 添加了 qualname 和 exceptiontable 形参。

在 3.12 版本发生变更: 由 PyCode_New 更名而来，是*不稳定 C API* 的一部分。旧名称已被弃用，但在签名再次更改之前仍然可用。

PyCodeObject ***PyUnstable_Code_NewWithPosOnlyArgs**(int argcount, int posonlyargcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)



这是*不稳定 API*。它可在次发布版中不经警告地改变。

与*PyUnstable_Code_New()* 类似，但额外增加了一个针对仅限位置参数的”posonlyargcount”。适用于 PyUnstable_Code_New 的适用事项同样适用于这个函数。

Added in version 3.8: 作为 PyCode_NewWithPosOnlyArgs

在 3.11 版本发生变更: 增加了 qualname 和 exceptiontable 形参。

在 3.12 版本发生变更: 重命名为 PyUnstable_Code_NewWithPosOnlyArgs。旧名称已被弃用，但在签名再次更改之前将保持可用。

PyCodeObject ***PyCode_NewEmpty**(const char *filename, const char *funcname, int firstlineno)

返回值：新的引用。 返回一个具有指定用户名、函数名和首行行号的空代码对象。结果代码对象如果被执行则将引发一个 Exception。

int **PyCode_Addr2Line**(*PyCodeObject* *co, int byte_offset)

返回在 byte_offset 位置或之前以及之后发生的指令的行号。如果你只需要一个帧的行号，请改用*PyFrame_GetLineNumber()*。

要高效地对代码对象中的行号进行迭代，请使用 **在 PEP 626 中描述的 API**。

int **PyCode_Addr2Location**(*PyObject* *co, int byte_offset, int *start_line, int *start_column, int *end_line, int *end_column)

将传入的 int 指针设为 byte_offset 处的指令的源代码行编号和列编号。当没有任何特定元素的信息时则将值设为 0。

如果函数执行成功则返回 1 否则返回 0。

Added in version 3.11.

PyObject ***PyCode_GetCode**(*PyCodeObject* *co)

等价于 Python 代码 getattr(co, 'co_code')。返回一个指向表示代码对象中的字节码的*PyBytesObject* 的强引用。当出错时，将返回 NULL 并引发一个异常。

这个 PyBytesObject 可以由解释器按需创建并且不必代表 CPython 所实际执行的字节码。此函数的主要用途是调试器和性能分析工具。

Added in version 3.11.

PyObject ***PyCode_GetVarnames**(*PyCodeObject* *co)

等价于 Python 代码 getattr(co, 'co_varnames')。返回一个指向包含局部变量名称的*PyTupleObject* 的新引用。当出错时，将返回 NULL 并引发一个异常。

Added in version 3.11.

PyObject ***PyCode_GetCellvars**(*PyCodeObject* *co)

等价于 Python 代码 getattr(co, 'co_cellvars')。返回一个包含被嵌套的函数所引用的局部变量名称的*PyTupleObject* 的新引用。当出错时，将返回 NULL 并引发一个异常。

Added in version 3.11.

PyObject ***PyCode_GetFreevars**(*PyCodeObject* *co)

等价于 Python 代码 getattr(co, 'co_freevars')。返回一个指向包含*PyTupleObject 自由（闭包）变量* 名称的新引用。当出错时，将返回 NULL 并引发一个异常。

Added in version 3.11.

int **PyCode_AddWatcher**(*PyCode_WatchCallback* callback)

注册 *callback* 作为当前解释器的代码对象监视器。返回一个可被传给*PyCode_ClearWatcher()* 的 ID。如果出现错误（例如没有足够的可用监视器 ID），则返回 -1 并设置一个异常。

Added in version 3.12.

int **PyCode_ClearWatcher**(int watcher_id)

清除之前从*PyCode_AddWatcher()* 返回的当前解释器中由 *watcher_id* 所标识的监视器。成功时返回 0，或者出错时（例如当给定的 *watcher_id* 未被注册）返回 -1 并设置异常。

Added in version 3.12.

type **PyCodeEvent**

由可能的代码对象监视器事件组成的枚举: - PY_CODE_EVENT_CREATE - PY_CODE_EVENT_DESTROY

Added in version 3.12.

typedef int (***PyCode_WatchCallback**)(*PyCodeEvent* event, *PyCodeObject* *co)

代码对象监视器回调函数的类型。

如果 *event* 为 PY_CODE_EVENT_CREATE，则回调会在 *co* 完全初始化之后被唤起。在其他情况下，回调会在 *co* 的销毁执行之前被唤起，因此可以检查 *co* 在之前的状态。

如果 *event* 为 PY_CODE_EVENT_DESTROY，则在回调中接受一个即将被销毁的代码对象的引用将使其重生，并阻止其在此时被释放。当重生的对象以后再被销毁时，任何在当时已激活的监视器回调将再次被调用。

本 API 的用户不应依赖内部运行时的实现细节。这类细节可能包括但不限于创建和销毁代码对象的确切顺序和时间。虽然这些细节的变化可能会导致监视器可观察到的差异（包括回调是否被唤起），但不会改变正在执行的 Python 代码的语义。

如果该回调设置了一个异常，则它必须返回 -1；此异常将作为不可引发的异常使用*PyErr_WriteUnraisable()* 打印出来。在其他情况下它应当返回 0。

在进入回调时可能已经设置了尚未处理的异常。在此情况下，回调应当返回 0 并仍然设置同样的异常。这意味着该回调可能不会调用任何其他可设置异常的 API 除非它先保存并清空异常状态，并在返回之前恢复它。

Added in version 3.12.

8.5.6 Code Object Flags

Code objects contain a bit-field of flags, which can be retrieved as the co_flags Python attribute (for example using *PyObject_GetAttrString()*), and set using a *flags* argument to *PyUnstable_Code_New()* and similar functions.

Flags whose names start with CO_FUTURE_ correspond to features normally selectable by future statements. These flags can be used in *PyCompilerFlags.cf_flags*. Note that many CO_FUTURE_ flags are mandatory in current versions of Python, and setting them has no effect.

The following flags are available. For their meaning, see the linked documentation of their Python equivalents.

旗标	含意
CO_OPTIMIZED	inspect.CO_OPTIMIZED
CO_NEWLOCALS	inspect.CO_NEWLOCALS
CO_VARARGS	inspect.CO_VARARGS
CO_VARKEYWORDS	inspect.CO_VARKEYWORDS
CO_NESTED	inspect.CO_NESTED
CO_GENERATOR	inspect.CO_GENERATOR
CO_COROUTINE	inspect.CO_COROUTINE
CO_ITERABLE_COROUTINE	inspect.CO_ITERABLE_COROUTINE
CO_ASYNC_GENERATOR	inspect.CO_ASYNC_GENERATOR
CO_FUTURE_DIVISION	no effect (__future__.division)
CO_FUTURE_ABSOLUTE_IMPORT	no effect (__future__.absolute_import)
CO_FUTURE_WITH_STATEMENT	no effect (__future__.with_statement)
CO_FUTURE_PRINT_FUNCTION	no effect (__future__.print_function)
CO_FUTURE_UNICODE_LITERALS	no effect (__future__.unicode_literals)
CO_FUTURE_GENERATOR_STOP	no effect (__future__.generator_stop)
CO_FUTURE_ANNOTATIONS	__future__.annotations

8.5.7 附加信息

为了支持对帧求值的低层级扩展，如外部即时编译器等，可以在代码对象上附加任意的额外数据。

这些函数是不稳定 C API 层的一部分：该功能是 CPython 的实现细节，此 API 可能随时改变而不发出弃用警告。

Py_ssize_t **PyUnstable_Eval_RequestCodeExtraIndex**(*freefunc* free)

这是不稳定 API。它可在次发布版中不经警告地改变。

返回一个新的不透明索引值用于向代码对象添加数据。

通常情况下（对于每个解释器）你只需调用该函数一次然后将调用结果与 PyCode_GetExtra 和 PyCode_SetExtra 一起使用以操作单个代码对象上的数据。

如果 *free* 没有不为 NULL: 当代码对象被释放时，*free* 将在存储于新索引下的非 NULL 数据上被调用。当存储 PyObject 时使用 Py_DecRef()。

Added in version 3.6: 作为 _PyEval_RequestCodeExtraIndex

在 3.12 版本发生变更: 重命名为 PyUnstable_Eval_RequestCodeExtraIndex。旧的私有名称已被弃用，但在 API 更改之前仍将可用。

int **PyUnstable_Code_GetExtra**(*PyObject* *code, *Py_ssize_t* index, void **extra)

这是不稳定 API。它可在次发布版中不经警告地改变。

将 *extra* 设为存储在给定索引下的额外数据。成功时将返回 0。失败时将设置一个异常并返回 -1。

如果未在索引下设置数据，则将 *extra* 设为 NULL 并返回 0 而不设置异常。

Added in version 3.6: 作为 _PyCode_GetExtra

在 3.12 版本发生变更: 重命名为 PyUnstable_Code_GetExtra。旧的私有名称已被弃用，但在 API 更改之前仍将可用。

int **PyUnstable_Code_SetExtra**(*PyObject* *code, *Py_ssize_t* index, void *extra)

这是不稳定 API。它可在次发布版中不经警告地改变。

将存储在给定索引下的额外数据设为 *extra*。成功时将返回 0。失败时将设置一个异常并返回 -1。

Added in version 3.6: 作为 _PyCode_SetExtra

在 3.12 版本发生变更: 重命名为 PyUnstable_Code_SetExtra。旧的私有名称已被弃用，但在 API 更改之前仍将可用。

8.6 其他对象

8.6.1 文件对象

这些 API 是对内置文件对象的 Python 2 C API 的最小化模拟，它过去依赖于 C 标准库的带缓冲 I/O (FILE*) 支持。在 Python 3 中，文件和流使用新的 io 模块，该萨凡纳的操作系统的低层级无缓冲 I/O 之上定义了几个层。下面介绍的函数是针对这些新 API 的便捷 C 包装器，主要用于解释器的内部错误报告；建议第三方代码改为访问 io API。

PyObject ***PyFile_FromFd**(int fd, const char *name, const char *mode, int buffering, const char *encoding, const char *errors, const char *newline, int closefd)

返回值：新的引用。属于稳定 ABI. 根据已打开文件 *fd* 的文件描述符创建一个 Python 文件对象。参数 *name*, *encoding*, *errors* 和 *newline* 可以为 NULL 表示使用默认值；*buffering* 可以为 *-1* 表示使用默认值。*name* 会被忽略仅保留用于向下兼容。失败时返回 NULL。有关参数的更全面描述，请参阅 io.open() 函数的文档。

> **警告**
>
> 由于 Python 流具有自己的缓冲层，因此将它们与 OS 级文件描述符混合会产生各种问题（例如数据的意外排序）。

在 3.2 版本发生变更: 忽略 *name* 属性。

int **PyObject_AsFileDescriptor**(*PyObject* *p)

属于稳定 ABI. 将与 *p* 关联的文件描述符作为 `int` 返回。如果对象是整数，则返回其值。如果不是，则如果对象存在 `fileno()` 方法则调用该方法；该方法必须返回一个整数，它将作为文件描述符的值返回。失败时将设置异常并返回 `-1`。

PyObject ***PyFile_GetLine**(*PyObject* *p, int n)

返回值：新的引用。属于稳定 ABI. 等价于 `p.readline([n])`，这个函数从对象 *p* 中读取一行。*p* 可以是文件对象或具有 `readline()` 方法的任何对象。如果 *n* 是 `0`，则无论该行的长度如何，都会读取一行。如果 *n* 大于 `0`，则从文件中读取不超过 *n* 个字节；可以返回行的一部分。在这两种情况下，如果立即到达文件末尾，则返回空字符串。但是，如果 *n* 小于 `0`，则无论长度如何都会读取一行，但是如果立即到达文件末尾，则引发 `EOFError`。

int **PyFile_SetOpenCodeHook**(*Py_OpenCodeHookFunction* handler)

重写 `io.open_code()` 的正常行为，将其形参通过所提供的处理程序来传递。

handler 函数的类型为：

typedef *PyObject* *(***Py_OpenCodeHookFunction**)(*PyObject**, void*)

等价于 `PyObject *(*)(PyObject *path, void *userData)`，其中 *path* 会确保为*PyUnicodeObject*。

userData 指针会被传入钩子函数。因于钩子函数可能由不同的运行时调用，该指针不应直接指向 Python 状态。

鉴于这个钩子专门在导入期间使用的，请避免在新模块执行期间进行导入操作，除非已知它们为冻结状态或者是在 `sys.modules` 中可用。

一旦钩子被设定，它就不能被移除或替换，之后对*PyFile_SetOpenCodeHook()* 的调用也将失败，如果解释器已经被初始化，函数将返回 -1 并设置一个异常。

此函数可以安全地在*Py_Initialize()* 之前调用。

引发一个不带参数的 审计事件 `setopencodehook`。

Added in version 3.8.

int **PyFile_WriteObject**(*PyObject* *obj, *PyObject* *p, int flags)

属于稳定 ABI. 将对象 *obj* 写入文件对象 *p*。*flags* 唯一支持的旗标是*Py_PRINT_RAW*；如果给定，则写入对象的 `str()` 而不是 `repr()`。成功时返回 `0`，失败时返回 `-1`；将设置适当的异常。

int **PyFile_WriteString**(const char *s, *PyObject* *p)

属于稳定 ABI. 将字符串 *s* 写入文件对象 *p*。成功返回 `0` 失败返回 `-1`；将设定相应的异常。

8.6.2 模块对象

PyTypeObject **PyModule_Type**

属于稳定 ABI. 这个 C 类型实例*PyTypeObject* 用来表示 Python 中的模块类型。在 Python 程序中该实例被暴露为 `types.ModuleType`。

int **PyModule_Check**(*PyObject* *p)

当 *p* 为模块类型的对象，或是模块子类型的对象时返回真值。该函数永远有返回值。

int **PyModule_CheckExact**(*PyObject* *p)

当 *p* 为模块类型的对象且不是*PyModule_Type* 的子类型的对象时返回真值。该函数永远有返回值。

PyObject ***PyModule_NewObject**(*PyObject* *name)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 返回一个新的模块对象，该对象的 `module.__name__` 将设为 *name*。模块的 `__name__`, `__doc__`, `__package__` 和 `__loader__` 属性将被填充(除 `__name__` 外全都设为 `None`)。调用方要负责设置 `__file__` 属性。

当发生错误时将返回 `NULL` 并设置一个异常。

Added in version 3.3.

在 3.4 版本发生变更: 现在 `__package__` 和 `__loader__` 将被设为 `None`。

PyObject ***PyModule_New**(const char *name)

返回值：新的引用。属于稳定 ABI. 这类似于*PyModule_NewObject()*，但其名称为 UTF-8 编码的字符串而不是 Unicode 对象。

PyObject ***PyModule_GetDict**(*PyObject* *module)

返回值：借入的引用。属于稳定 ABI. 返回实现 *module* 的命名空间的字典对象；此对象与模块对象的 `__dict__` 属性相同。如果 *module* 不是一个模块对象（或模块对象的子类型），则会引发 `SystemError` 并返回 `NULL`。

建议扩展使用其他 `PyModule_*` 和 `PyObject_*` 函数而不是直接操纵模块的 `__dict__`。

PyObject ***PyModule_GetNameObject**(*PyObject* *module)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 返回 *module* 的 `__name__` 值。如果模块未提供该值，或者如果它不是一个字符串，则会引发 `SystemError` 并返回 `NULL`。

Added in version 3.3.

const char ***PyModule_GetName**(*PyObject* *module)

属于稳定 ABI. 类似于*PyModule_GetNameObject()* 但返回 `'utf-8'` 编码的名称。

void ***PyModule_GetState**(*PyObject* *module)

属于稳定 ABI. 返回模块的“状态”，也就是说，返回指向在模块创建时分配的内存块的指针，或者 `NULL`。参见*PyModuleDef.m_size*。

PyModuleDef ***PyModule_GetDef**(*PyObject* *module)

属于稳定 ABI. 返回指向模块创建所使用的*PyModuleDef* 结构体的指针，或者如果模块不是使用结构体定义创建的则返回 `NULL`。

PyObject ***PyModule_GetFilenameObject**(*PyObject* *module)

返回值：新的引用。属于稳定 ABI. 返回使用 *module* 的 `__file__` 属性所加载的 *module* 所对应的文件名。如果未定义该属性，或者如果它不是一个字符串，则会引发 `SystemError` 并返回 `NULL`；在其他情况下将返回一个指向 Unicode 对象的引用。

Added in version 3.2.

const char ***PyModule_GetFilename**(*PyObject* *module)

属于稳定 ABI. 类似于*PyModule_GetFilenameObject()* 但会返回编码为'utf-8' 的文件名。

自 3.2 版本弃用: *PyModule_GetFilename()* 对于不可编码的文件名会引发 `UnicodeEncodeError`，请改用*PyModule_GetFilenameObject()*。

初始化 C 模块

模块对象通常是基于扩展模块（导出初始化函数的共享库），或内部编译模块（其中使用*PyImport_AppendInittab()* 添加初始化函数）。请参阅 building 或 extending-with-embedding 了解详情。

初始化函数可以向*PyModule_Create()* 传入一个模块定义实例，并返回结果模块对象，或者通过返回定义结构体本身来请求“多阶段初始化”。

type **PyModuleDef**

属于稳定 ABI （包括所有成员）. 模块定义结构，它保存创建模块对象所需的所有信息。每个模块通常只有一个这种类型的静态初始化变量

PyModuleDef_Base **m_base**

始终将此成员初始化为 `PyModuleDef_HEAD_INIT`。

const char ***m_name**

新模块的名称。

const char ***m_doc**

模块的文档字符串；一般会使用通过*PyDoc_STRVAR* 创建的文档字符串变量。

Py_ssize_t **m_size**

可以把模块的状态保存在为单个模块分配的内存区域中，使用*PyModule_GetState()* 检索，而不是保存在静态全局区。这使得模块可以在多个子解释器中安全地使用。

这个内存区域将在创建模块时根据 *m_size* 分配，并在调用*m_free* 函数（如果存在）在取消分配模块对象时释放。

将 `m_size` 设置为 `-1`，意味着这个模块具有全局状态，因此不支持子解释器。

将其设置为非负值，意味着模块可以重新初始化，并指定其状态所需要的额外内存大小。多阶段初始化需要非负的 `m_size`。

请参阅 **PEP 3121** 了解详情。

PyMethodDef ***m_methods**

一个指向模块函数表的指针，由*PyMethodDef* 描述。如果模块没有函数，可以为 `NULL`。

PyModuleDef_Slot ***m_slots**

由针对多阶段初始化的槽位定义组成的数组，以一个 `{0, NULL}` 条目结束。当使用单阶段初始化时，*m_slots* 必须为 `NULL`。

在 3.5 版本发生变更: 在 3.5 版之前，此成员总是被设为 `NULL`，并被定义为:

inquiry **m_reload**

traverseproc **m_traverse**

在模块对象的垃圾回收遍历期间所调用的遍历函数，如果不需要则为 `NULL`。

如果模块状态已被请求但尚未分配则不会调用此函数。在模块创建之后至模块执行之前（调用*Py_mod_exec* 函数）就属于这种情况。更确切地说，如果*m_size* 大于 0 且模块状态（由*PyModule_GetState()* 返回）为 `NULL` 则不会调用此函数。

在 3.9 版本发生变更: 在模块状态被分配之前不再调用。

inquiry **m_clear**

在模块对象的垃圾回收清理期间所调用的清理函数，如果不需要则为 `NULL`。

如果模块状态已被请求但尚未分配则不会调用此函数。在模块创建之后至模块执行之前（调用*Py_mod_exec* 函数）就属于这种情况。更确切地说，如果*m_size* 大于 0 且模块状态（由*PyModule_GetState()* 返回）为 `NULL` 则不会调用此函数。

就像*PyTypeObject.tp_clear* 那样，这个函数并不总是在模块被释放前被调用。例如，当引用计数足以确定一个对象不再被使用时，就会直接调用*m_free*，而不使用循环垃圾回收器。

在 3.9 版本发生变更: 在模块状态被分配之前不再调用。

freefunc **m_free**

在模块对象的释放期间所调用的函数，如果不需要则为 `NULL`。

如果模块状态已被请求但尚未分配则不会调用此函数。在模块创建之后至模块执行之前（调用*Py_mod_exec* 函数）就属于这种情况。更确切地说，如果*m_size* 大于 0 且模块状态（由*PyModule_GetState()* 返回）为 `NULL` 则不会调用此函数。

在 3.9 版本发生变更: 在模块状态被分配之前不再调用。

单阶段初始化

模块初始化函数可以直接创建并返回模块对象，称为“单阶段初始化”，使用以下两个模块创建函数中的一个：

PyObject ***PyModule_Create**(*PyModuleDef* *def)

返回值：新的引用。根据在 *def* 中给出的定义创建一个新的模块对象。它的行为类似于 *PyModule_Create2()* 将 *module_api_version* 设为 PYTHON_API_VERSION。

PyObject ***PyModule_Create2**(*PyModuleDef* *def, int module_api_version)

返回值：新的引用。属于稳定 ABI. 创建一个新的模块对象，在参数 *def* 中给出定义，设定 API 版本为参数 *module_api_version* 。如果该版本与正在运行的解释器版本不匹配，则会触发 RuntimeWarning。

当发生错误时将返回 NULL 并设置一个异常。

> **备注**
>
> 大多数时候应该使用 *PyModule_Create()* 代替使用此函数，除非你确定需要使用它。

在初始化函数返回之前，生成的模块对象通常使用 *PyModule_AddObjectRef()* 等函数进行填充。

多阶段初始化

指定扩展的另一种方式是请求“多阶段初始化”。以这种方式创建的扩展模块行为更类似于 Python 模块：初始化分为 创建阶段，即创建模块对象时，以及 执行阶段，即填充模块时。这种区分与类的 __new__() 和 __init__() 方法类似。

不同于使用单阶段初始化创建的模块，这些模块不是单例对象。举例来说，如果 sys.modules 条目被移除且模块被重新导入，一个新模块对象将被创建，并且通常会以新的方法和类型对象充实其内容。旧模块将成为正常垃圾回收的目标。这类似于纯 Python 模块的行为。

额外的模块实例可能会在子解释器 中或者 Python 运行时重新初始化之后 (*Py_Finalize()* 和 *Py_Initialize()*) 被创建。在这些情况下，在模块实例之间共享 Python 对象将可能导致程序崩溃或未定义的行为。

为避免这种问题，每个扩展模块的实例都应当是 隔离的：对一个实例的修改不应隐式地影响其他的实例，以及所有的状态，包括对 Python 对象的引用，都应当是特定模块实例专属的。请参阅 isolating-extensions-howto 了解更多的细节和实用的指南。

一个避免这些问题的简单方式是 针对重复的初始化引发一个错误。

所有使用多阶段初始化创建的模块都应该支持子解释器，否则要显式地发出缺乏支持的信号。这往往是通过隔离或阻止重复的初始化，如上文所述。一个模块也可以使用 *Py_mod_multiple_interpreters* 槽位将其限制于主解释器中。

要请求多阶段初始化，初始化函数 (PyInit_modulename) 返回一个包含非空的 *m_slots* 属性的 *PyModuleDef* 实例。在它被返回之前，这个 PyModuleDef 实例必须先使用以下函数初始化：

PyObject ***PyModuleDef_Init**(*PyModuleDef* *def)

返回值：借入的引用。属于稳定 ABI 自 *3.5* 版起. 确保模块定义是一个正确初始化的 Python 对象，拥有正确的类型和引用计数。

返回转换为 PyObject* 的 *def* ，如果发生错误，则返回 NULL。

Added in version 3.5.

模块定义的 *m_slots* 成员必须指向一个 PyModuleDef_Slot 结构体数组：

type **PyModuleDef_Slot**

int **slot**

槽位 ID，从下面介绍的可用值中选择。

void ***value**

槽位值，其含义取决于槽位 ID。

Added in version 3.5.

m_slots 数组必须以一个 id 为 0 的槽位结束。

可用的槽位类型是:

Py_mod_create

指定一个函数供调用以创建模块对象本身。该槽位的 *value* 指针必须指向一个具有如下签名的函数:

PyObject ***create_module**(*PyObject* *spec, *PyModuleDef* *def)

该函数接受一个 `ModuleSpec` 实例，如 **PEP 451** 所定义的，以及模块定义。它应当返回一个新的模块对象，或者设置一个错误并返回 `NULL`。

此函数应当保持最小化。特别地，它不应当调用任意 Python 代码，因为尝试再次导入同一个模块可能会导致无限循环。

多个 `Py_mod_create` 槽位不能在一个模块定义中指定。

如果未指定 `Py_mod_create`，导入机制将使用*PyModule_New()* 创建一个普通的模块对象。名称是获取自 *spec* 而非定义，以允许扩展模块动态地调整它们在模块层级结构中的位置并通过符号链接以不同的名称被导入，同时共享同一个模块定义。

不要求返回的对象必须为*PyModule_Type* 的实例。任何类型均可使用，只要它支持设置和获取导入相关的属性。但是，如果 `PyModuleDef` 具有非 `NULL` 的 `m_traverse`, `m_clear`, `m_free`；非零的 `m_size`；或者 `Py_mod_create` 以外的槽位则只能返回 `PyModule_Type` 的实例。

Py_mod_exec

指定一个供调用以 执行模块的函数。这造价于执行一个 Python 模块的代码：通常，此函数会向模块添加类和常量。此函数的签名为:

int **exec_module**(*PyObject* *module)

如果指定了多个 `Py_mod_exec` 槽位，将按照它们在 *m_slots* 数组中出现的顺序进行处理。

Py_mod_multiple_interpreters

指定以下的值之一:

Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED

该模块不支持在子解释器中导入。

Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED

该模块支持在子解释器中导入，但是它们必须要共享主解释器的 GIL。(参见 isolating-extensions-howto。)

Py_MOD_PER_INTERPRETER_GIL_SUPPORTED

该模块支持在子解释器中导入，即使它们有自己的 GIL。(参见 isolating-extensions-howto。)

此槽位决定在子解释器中导入此模块是否会失败。

在一个模块定义中不能指定多个 `Py_mod_multiple_interpreters` 槽位。

如果未指定 `Py_mod_multiple_interpreters`，则导入机制默认为 `Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED`。

Added in version 3.12.

Py_mod_gil

指定以下的值之一:

Py_MOD_GIL_USED

这个模块依赖于全局解释器锁 (GIL) 的存在，并可访问全局状态而不带同步。

Py_MOD_GIL_NOT_USED

这个模块可以在不激活 GIL 的情况下安全运行。

这个槽位会被未配置 `--disable-gil` 的 Python 构建版所忽略。在其他情况下，它将决定导入此模块是否会导致 GIL 被自动启用。请参阅 whatsnew313-free-threaded-cpython 了解详情。

多个 `Py_mod_gil` 槽位不能在一个模块定义中指定。

如果未指定 `Py_mod_gil`，则导入机制默认为 `Py_MOD_GIL_USED`。

Added in version 3.13.

有关多阶段初始化的更多细节，请参阅 PEP:*489*

底层模块创建函数

当使用多阶段初始化时，将会调用以下函数。例如，在动态创建模块对象的时候，可以直接使用它们。注意，必须调用 `PyModule_FromDefAndSpec` 和 `PyModule_ExecDef` 来完整地初始化一个模块。

PyObject ***PyModule_FromDefAndSpec**(*PyModuleDef* *def, *PyObject* *spec)

返回值：新的引用。 根据在 *def* 中给出的定义和 ModuleSpec *spec* 创建一个新的模块对象。它的行为类似于*PyModule_FromDefAndSpec2()* 将 *module_api_version* 设为 `PYTHON_API_VERSION`。

Added in version 3.5.

PyObject ***PyModule_FromDefAndSpec2**(*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 创建一个新的模块对象，在参数 *def* 和 *spec* 中给出定义，设置 API 版本为参数 *module_api_version*。如果该版本与正在运行的解释器版本不匹配，则会触发 `RuntimeWarning`。

当发生错误时将返回 `NULL` 并设置一个异常。

> **备注**
>
> 大多数时候应该使用*PyModule_FromDefAndSpec()* 代替使用此函数，除非你确定需要使用它。

Added in version 3.5.

int **PyModule_ExecDef**(*PyObject* *module, *PyModuleDef* *def)

属于稳定 ABI 自 *3.7* 版起. 执行参数 *def* 中给出的任意执行槽（*Py_mod_exec*）。

Added in version 3.5.

int **PyModule_SetDocString**(*PyObject* *module, const char *docstring)

属于稳定 ABI 自 *3.7* 版起. 将 *module* 的文档字符串设置为 *docstring*。当使用 `PyModule_Create` 或 `PyModule_FromDefAndSpec` 从 `PyModuleDef` 创建模块时，会自动调用此函数。

Added in version 3.5.

int **PyModule_AddFunctions**(*PyObject* *module, *PyMethodDef* *functions)

属于稳定 ABI 自 *3.7* 版起. 将以 `NULL` 结尾的 *functions* 数组中的函数添加到 *module* 模块中。有关单个条目的更多细节，请参与*PyMethodDef* 文档（由于缺少共享的模块命名空间，在 C 中实现的模块级“函数”通常将模块作为它的第一个参数，与 Python 类的实例方法类似）。当使用 `PyModule_Create` 或 `PyModule_FromDefAndSpec` 从 `PyModuleDef` 创建模块时，会自动调用此函数。

Added in version 3.5.

支持函数

模块初始化函数（单阶段初始化）或通过模块的执行槽位调用的函数（多阶段初始化），可以使用以下函数，来帮助初始化模块的状态：

int **PyModule_AddObjectRef**(*PyObject* *module, const char *name, *PyObject* *value)

属于稳定 ABI 自 *3.10* 版起. 将一个名称为 *name* 的对象添加到 *module* 模块中。这是一个方便的函数，可以在模块的初始化函数中使用。

如果成功，返回 `0`。如果发生错误，引发异常并返回 `-1`。

用法示例：

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (obj == NULL) {
        return -1;
    }
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_DECREF(obj);
    return res;
}
```

为了方便，该函数接受 `NULL` *value* 并设置一个异常。在此情况下，将返回 `-1` 并让所引发的异常保持不变。

这个例子也可以写成不显式地检查 *obj* 是否为 `NULL`:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_XDECREF(obj);
    return res;
}
```

注意在此情况下应当使用 `Py_XDECREF()` 而不是 `Py_DECREF()`，因为 *obj* 可能为 `NULL`。

传给该函数的不同 *name* 字符串应当保持在较少的数量，通常是通过仅使用静态分配的字符串作为 *name* 来做到这一点。对于编译时未知的名称，建议直接调用*PyUnicode_FromString()* 和*PyObject_SetAttr()*。更多相关细节，请参阅*PyUnicode_InternFromString()*，它可在内部用于创建键对象。

Added in version 3.10.

int **PyModule_Add**(*PyObject* *module, const char *name, *PyObject* *value)

属于稳定 ABI 自 *3.13* 版起. 类似于*PyModule_AddObjectRef()*，但会“偷取”一个指向 *value* 的引用。它在被调用时可附带一个返回新引用的函数的结果而无需检查其结果或是将其保存到一个变量。

用法示例：

```
if (PyModule_Add(module, "spam", PyBytes_FromString(value)) < 0) {
    goto error;
}
```

Added in version 3.13.

int **PyModule_AddObject**(*PyObject* *module, const char *name, *PyObject* *value)

属于稳定 ABI. 类似于*PyModule_AddObjectRef()*，但会在成功时偷取一个对 *value* 的引用（如果它返回 `0` 值）。

推荐使用新的*PyModule_Add()* 或*PyModule_AddObjectRef()* 函数，因为误用*PyModule_AddObject()* 函数很容易导致引用泄漏。

> **备注**
>
> 与其他窃取引用的函数不同，`PyModule_AddObject()` 只在 **成功**时释放对 *value* 的引用。
>
> 这意味着必须检查它的返回值，调用方代码必须在发生错误时手动为 *value* 执行*Py_XDECREF()*。

用法示例：

```
PyObject *obj = PyBytes_FromString(value);
if (PyModule_AddObject(module, "spam", obj) < 0) {
    // 如果 'obj' 不为 NULL 且 PyModule_AddObject() 执行失败，
    // 则 'obj' 强引用必须使 Py_XDECREF() 来删除。
    // 如果 'obj' 为 NULL，则 Py_XDECREF() 不做任何操作。
    Py_XDECREF(obj);
    goto error;
}
// PyModule_AddObject() 会偷取一个对 obj 的引用：
// 这里不需要 Py_XDECREF(obj)。
```

自 3.13 版本弃用: *PyModule_AddObject()* 处于*soft deprecated* 状态。

int **PyModule_AddIntConstant**(*PyObject* *module, const char *name, long value)

属于稳定 ABI. 将一个整数常量作为 *name* 添加到 *module* 中。这个便捷函数可在模块的初始化函数中使用。当发生错误时将返回 `-1` 并设置一个异常，成功时则返回 `0`。

这是一个调用*PyLong_FromLong()* 和*PyModule_AddObjectRef()* 的便捷函数；请参阅其文档了解详情。

int **PyModule_AddStringConstant**(*PyObject* *module, const char *name, const char *value)

属于稳定 ABI. 将一个字符串常量作为 *name* 添加到 *module* 中。这个便捷函数可在模块初始化函数中使用。字符串 *value* 必须以 `NULL` 结尾。当发生错误时将返回 `-1`，成功时则返回 `0`。

这是一个调用*PyUnicode_InternFromString()* 和*PyModule_AddObjectRef()* 的便捷函数；请参阅其文档了解详情。

PyModule_AddIntMacro(module, macro)

将一个整数常量添加到 *module* 中。名称和值取自 *macro*。例如 `PyModule_AddIntMacro(module, AF_INET)` 将值为 *AF_INET* 的整数常量 *AF_INET* 添加到 *module* 中。当发生错误时将抬 `-1` 并设置一个异常，成功时将返回 `0`。

PyModule_AddStringMacro(module, macro)

将一个字符串常量添加到 *module* 模块中。

int **PyModule_AddType**(*PyObject* *module, *PyTypeObject* *type)

属于稳定 ABI 自 *3.10* 版起. 将一个类型对象添加到 *module* 中。类型对象是通过在内部调用*PyType_Ready()* 来最终化的。类型对象的名称取自*tp_name* 在点号之后的部分。当发生错误时将返回 `-1` 并设置一个异常，成功时将返回 `0`。

Added in version 3.9.

int **PyUnstable_Module_SetGIL**(*PyObject* *module, void *gil)

> 这是不稳定 *API*。它可在次发布版中不经警告地改变。

指明 *module* 是否支持不带全局解释器锁 (GIL) 运行，使用来自*Py_mod_gil* 的值。它必须在 *module* 的初始化函数执行期间被调用。如果此函数在模块初始化期间未被调用，导入机制将假定该模块不支持不带 GIL 运行。此函数仅在配置了 `--disable-gil` 的 Python 编译版中可用。当发生错误时将返回 `-1` 并设置一个异常，成功时将返回 `0`。

Added in version 3.13.

查找模块

单阶段初始化创建可以在当前解释器上下文中被查找的单例模块。这使得仅通过模块定义的引用，就可以检索模块对象。

这些函数不适用于通过多阶段初始化创建的模块，因为可以从一个模块定义创建多个模块对象。

PyObject ***PyState_FindModule**(*PyModuleDef* *def)

返回值：借入的引用。属于稳定 ABI. 返回当前解释器中由 *def* 创建的模块对象。此方法要求模块对象此前已通过*PyState_AddModule()* 函数附加到解释器状态中。如果找不到相应的模块对象，或模块对象还未附加到解释器状态，返回 NULL。

int **PyState_AddModule**(*PyObject* *module, *PyModuleDef* *def)

属于稳定 ABI 自 *3.3* 版起. 将传给函数的模块对象附加到解释器状态。这将允许通过*PyState_FindModule()* 来访问该模块对象。

仅在使用单阶段初始化创建的模块上有效。

Python 会在导入一个模块后自动调用 PyState_AddModule，因此从模块初始化代码中调用它是没有必要的（但也没有害处）。显式的调用仅在模块自己的初始化代码后继调用了 PyState_FindModule 的情况下才是必要的。此函数主要是为了实现替代导入机制（或是通过直接调用它，或是通过引用它的实现来获取所需的状态更新详情）。

调用时必须携带 GIL。

出错时返回 -1 并设置一个异常，成功时返回 0。

Added in version 3.3.

int **PyState_RemoveModule**(*PyModuleDef* *def)

属于稳定 ABI 自 *3.3* 版起. 从解释器状态中移除由 *def* 创建的模块对象。当发生错误时将返回 -1 并设置一个异常，成功时将返回 0。

调用时必须携带 GIL。

Added in version 3.3.

8.6.3 迭代器对象

Python 提供了两个通用迭代器对象。第一个是序列迭代器，它可与支持 __getitem__() 方法的任意序列一起使用。第二个迭代器使用一个可调用对象和一个哨兵值，为序列中的每个项目调用可调用对象，并在返回哨兵值时结束迭代。

PyTypeObject **PySeqIter_Type**

属于稳定 ABI. *PySeqIter_New()* 返回迭代器对象的类型对象和内置序列类型内置函数 iter() 的单参数形式。

int **PySeqIter_Check**(*PyObject* *op)

如果 *op* 的类型为*PySeqIter_Type* 则返回真值。此函数总是会成功执行。

PyObject ***PySeqIter_New**(*PyObject* *seq)

返回值：新的引用。属于稳定 ABI. 返回一个与常规序列对象一起使用的迭代器 *seq*。当序列订阅操作引发 IndexError 时，迭代结束。

PyTypeObject **PyCallIter_Type**

属于稳定 ABI. 由函数*PyCallIter_New()* 和 iter() 内置函数的双参数形式返回的迭代器对象类型对象。

int **PyCallIter_Check**(*PyObject* *op)

如果 *op* 的类型为*PyCallIter_Type* 则返回真值。此函数总是会成功执行。

PyObject ***PyCallIter_New**(*PyObject* *callable, *PyObject* *sentinel)

返回值：新的引用。属于稳定 ABI. 返回一个新的迭代器。第一个参数 *callable* 可以是任何可以在没有参数的情况下调用的 Python 可调用对象；每次调用都应该返回迭代中的下一个项目。当 *callable* 返回等于 *sentinel* 的值时，迭代将终止。

8.6.4 描述符对象

“描述符”是描述对象的某些属性的对象。它们存在于类型对象的字典中。

PyTypeObject **PyProperty_Type**
属于稳定 ABI. 内建描述符类型的类型对象。

PyObject ***PyDescr_NewGetSet**(*PyTypeObject* *type, struct *PyGetSetDef* *getset)
返回值：新的引用。属于稳定 ABI.

PyObject ***PyDescr_NewMember**(*PyTypeObject* *type, struct *PyMemberDef* *meth)
返回值：新的引用。属于稳定 ABI.

PyObject ***PyDescr_NewMethod**(*PyTypeObject* *type, struct *PyMethodDef* *meth)
返回值：新的引用。属于稳定 ABI.

PyObject ***PyDescr_NewWrapper**(*PyTypeObject* *type, struct wrapperbase *wrapper, void *wrapped)
返回值：新的引用。

PyObject ***PyDescr_NewClassMethod**(*PyTypeObject* *type, *PyMethodDef* *method)
返回值：新的引用。属于稳定 ABI.

int **PyDescr_IsData**(*PyObject* *descr)
如果描述符对象 *descr* 描述的是一个数据属性则返回非零值，或者如果它描述的是一个方法则返回 0。*descr* 必须为一个描述符对象；不会进行错误检测。

PyObject ***PyWrapper_New**(*PyObject**, *PyObject**)
返回值：新的引用。属于稳定 ABI.

8.6.5 切片对象

PyTypeObject **PySlice_Type**
属于稳定 ABI. 切片对象的类型对象。它与 Python 层面的 `slice` 是相同的对象。

int **PySlice_Check**(*PyObject* *ob)
如果 *ob* 是一个 slice 对象则返回真值；*ob* 必须不为 `NULL`。此函数总是会成功执行。

PyObject ***PySlice_New**(*PyObject* *start, *PyObject* *stop, *PyObject* *step)
返回值：新的引用。属于稳定 ABI. 返回一个具有给定值的新切片对象。*start*, *stop* 和 *step* 形参会被用作该切片对象相同名称的属性值。这些值中的任何一个都可以为 `NULL`，在此情况下将使用 `None` 作为相应的属性值。

当无法分配新对象时将返回 `NULL` 并设置一个异常。

int **PySlice_GetIndices**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)
属于稳定 ABI. 从切片对象 *slice* 提取 start, stop 和 step 索引号，将序列长度视为 *length*。大于 *length* 的序列号将被当作错误。

成功时返回 `0`，出错时返回 `-1` 并且不设置异常（除非某个索引号不为 `None` 且无法被转换为整数，在这种情况下将返回 `-1` 并且设置一个异常）。

你可能不会打算使用此函数。

在 3.2 版本发生变更: 之前 *slice* 形参的形参类型是 `PySliceObject*`。

int **PySlice_GetIndicesEx**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step, *Py_ssize_t* *slicelength)
属于稳定 ABI. *PySlice_GetIndices()* 的可用替代。从切片对象 *slice* 提取 start, stop 和 step 索引号，将序列长度视为 *length*，并将切片的长度保存在 *slicelength* 中，超出范围的索引号会以与普通切片一致的方式进行剪切。

成功时返回 `0` 而在出错时返回 `-1` 并设置一个异常。

备注

此函数对于可变大小序列来说是不安全的。对它的调用应被替换为*PySlice_Unpack()* 和*PySlice_AdjustIndices()* 的组合，其中

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength) < 0) {
    // 返回错误
}
```

会被替换为

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // 返回错误
}
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

在 3.2 版本发生变更: 之前 *slice* 形参的形参类型是 `PySliceObject*`。

在 3.6.1 版本发生变更: 如果 `Py_LIMITED_API` 未设置或设置为 `0x03050400` 与 `0x03060000` 之间的值（不包括边界）或 `0x03060100` 或更大则 `PySlice_GetIndicesEx()` 会被实现为一个使用 `PySlice_Unpack()` 和 `PySlice_AdjustIndices()` 的宏。参数 *start*, *stop* 和 *step* 会被多被求值。

自 3.6.1 版本弃用: 如果 `Py_LIMITED_API` 设置为小于 `0x03050400` 或 `0x03060000` 与 `0x03060100` 之间的值（不包括边界）则 `PySlice_GetIndicesEx()` 为已弃用的函数。

int **PySlice_Unpack**(*PyObject* *slice, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

属于稳定 ABI 自 *3.7* 版起. 从切片对象中将 start, stop 和 step 数据成员提取为 C 整数。会静默地将大于 `PY_SSIZE_T_MAX` 的值减小为 `PY_SSIZE_T_MAX`，静默地将小于 `PY_SSIZE_T_MIN` 的 start 和 stop 值增大为 `PY_SSIZE_T_MIN`，并静默地将小于 `-PY_SSIZE_T_MAX` 的 step 值增大为 `-PY_SSIZE_T_MAX`。

出错时返回 `-1` 并设置一个异常，成功时返回 `0`。

Added in version 3.6.1.

Py_ssize_t **PySlice_AdjustIndices**(*Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* step)

属于稳定 ABI 自 *3.7* 版起. 将 start/end 切片索引号根据指定的序列长度进行调整。超出范围的索引号会以与普通切片一致的方式进行剪切。

返回切片的长度。此操作总是会成功。不会调用 Python 代码。

Added in version 3.6.1.

Ellipsis 对象

PyTypeObject **PyEllipsis_Type**

属于稳定 ABI. Python `Ellipsis` 对象的类型。与 Python 层的 `types.EllipsisType` 为同一对象。

PyObject ***Py_Ellipsis**

Python `Ellipsis` 对象。此对象没有任何方法。像*Py_None* 一样，它是一个*immortal* 单例对象。

在 3.12 版本发生变更: *Py_Ellipsis* 是永久性对象。

8.6.6 MemoryView 对象

一个 `memoryview` 对象 C 级别的缓冲区接口 暴露为一个可以像任何其他对象一样传递的 Python 对象。

PyObject ***PyMemoryView_FromObject**(*PyObject* *obj)

返回值：新的引用。属于稳定 ABI. 从提供缓冲区接口的对象创建 memoryview 对象。如果 *obj* 支持可写缓冲区导出，则 memoryview 对象将可以被读/写，否则它可能是只读的，也可以是导出器自行决定的读/写。

PyBUF_READ

用于请求只读缓冲区的旗标。

PyBUF_WRITE

用于请求可写缓冲区的旗标。

PyObject ***PyMemoryView_FromMemory**(char *mem, *Py_ssize_t* size, int flags)

返回值：新的引用。属于稳定 ABI 自 *3.7* 版起. 使用 *mem* 作为底层缓冲区创建一个 memoryview 对象。*flags* 可以是*PyBUF_READ* 或者*PyBUF_WRITE* 之一.

Added in version 3.3.

PyObject ***PyMemoryView_FromBuffer**(const *Py_buffer* *view)

返回值：新的引用。属于稳定 ABI 自 *3.11* 版起. 创建一个包含给定缓冲区结构 *view* 的 memoryview 对象。对于简单的字节缓冲区，*PyMemoryView_FromMemory()* 是首选函数。

PyObject ***PyMemoryView_GetContiguous**(*PyObject* *obj, int buffertype, char order)

返回值：新的引用。属于稳定 ABI. 从定义缓冲区接口的对象创建一个 memoryview 对象*contiguous* 内存块（在'C' 或'F'ortran *order* 中）。如果内存是连续的，则 memoryview 对象指向原始内存。否则，复制并且 memoryview 指向新的 bytes 对象。

buffertype 可以为*PyBUF_READ* 或*PyBUF_WRITE* 中的一个。

int **PyMemoryView_Check**(*PyObject* *obj)

如果 *obj* 是一个 memoryview 对象则返回真值。目前不允许创建 `memoryview` 的子类。此函数总是会成功执行。

Py_buffer ***PyMemoryView_GET_BUFFER**(*PyObject* *mview)

返回指向 memoryview 的导出缓冲区私有副本的指针。*mview* **必须**是一个 memoryview 实例；这个宏不检查它的类型，你必须自己检查，否则你将面临崩溃风险。

PyObject ***PyMemoryView_GET_BASE**(*PyObject* *mview)

返回 memoryview 所基于的导出对象的指针，或者如果 memoryview 已由函数*PyMemoryView_FromMemory()* 或*PyMemoryView_FromBuffer()* 创建则返回 `NULL`。*mview* **必须**是一个 memoryview 实例。

8.6.7 弱引用对象

Python 支持“弱引用”作为一类对象。具体来说，有两种直接实现弱引用的对象。第一种就是简单的引用对象，第二种尽可能地作用为一个原对象的代理。

int **PyWeakref_Check**(*PyObject* *ob)

如果 *ob* 是一个引用或代理对象则返回非零值。此函数总是会成功执行。

int **PyWeakref_CheckRef**(*PyObject* *ob)

如果 *ob* 是一个引用对象则返回非零值。此函数总是会成功执行。

int **PyWeakref_CheckProxy**(*PyObject* *ob)

如果 *ob* 是一个代理对象则返回非零值。此函数总是会成功执行。

PyObject ***PyWeakref_NewRef**(*PyObject* *ob, *PyObject* *callback)

返回值：新的引用。属于稳定 ABI. 返回对象 *ob* 的弱引用对象。该函数总是会返回一个新的引用，但不保证创建一个新的对象；它有可能返回一个现有的引用对象。第二个形参 *callback* 可以是一个可调用对象，它会在 *ob* 被作为垃圾回收时接收通知；它应当接受一个单独形参，即弱引用对象本身。*callback* 也可以是 `None` 或 `NULL`。如果 *ob* 不是一个弱引用对象，或者如果 *callback* 不是可调用对象，`None` 或 `NULL`，该函数将返回 `NULL` 并引发 `TypeError`。

PyObject ***PyWeakref_NewProxy**(*PyObject* *ob, *PyObject* *callback)

返回值：新的引用。属于稳定 ABI. 返回对象 *ob* 的弱引用代理对象。该函数总是会返回一个新的引用，但不保证创建一个新的对象；它有可能返回一个现有的代理对象。第二个形参 *callback* 可以是一个可调用对象，它会在 *ob* 被作为垃圾回收时接收通知；它应当接受一个单独形参，即弱引用对象本身。*callback* 也可以是 `None` 或 `NULL`。如果 *ob* 不是一个弱引用对象，或者如果 *callback* 不是可调用对象，`None` 或 `NULL`，该函数将返回 `NULL` 并引发 `TypeError`。

int **PyWeakref_GetRef**(*PyObject* *ref, *PyObject* **pobj)

属于稳定 ABI 自 *3.13* 版起. 基于一个弱引用 *ref* 获取一个指向被引用对象的*strong reference* 存入到 **pobj*。

- 成功时，将 **pobj* 设为一个新的指向被引用对象的*strong reference* 并返回 1。
- 如果引用不可用，则将 **pobj* 设为 `NULL` 并返回 0。
- 发生错误时，将引发异常并返回 -1。

Added in version 3.13.

PyObject ***PyWeakref_GetObject**(*PyObject* *ref)

返回值：借入的引用。属于稳定 ABI. 基于一个弱引用 *ref* 返回一个指向被引用对象的*borrowed reference*。如果引用已不可用，则返回 `Py_None`。

> **备注**
>
> 该函数返回被引用对象的一个*borrowed reference*。这意味着应该总是在该对象上调用`Py_INCREF()`，除非是当它在借入引用的最后一次被使用之前无法被销毁的时候。

Deprecated since version 3.13, will be removed in version 3.15: 请改用`PyWeakref_GetRef()`。

PyObject ***PyWeakref_GET_OBJECT**(*PyObject* *ref)

返回值：借入的引用。类似于`PyWeakref_GetObject()`，但是不带错误检测。

Deprecated since version 3.13, will be removed in version 3.15: 请改用`PyWeakref_GetRef()`。

void **PyObject_ClearWeakRefs**(*PyObject* *object)

属于稳定 ABI. 此函数将被`tp_dealloc` 处理器调用以清空弱引用。

此函数将迭代 *object* 的弱引用并调用这些引用中可能存在的回调。它将在尝试了所有回调之后返回。

void **PyUnstable_Object_ClearWeakRefsNoCallbacks**(*PyObject* *object)

> 这是不稳定 *API*。它可在次发布版中不经警告地改变。

清空 *object* 的弱引用而不调用回调。

此函数将由`tp_dealloc` 处理器针对带有终结器 (即 `__del__()`) 的类型进行调用。针对这些对象的处理器会先调用`PyObject_ClearWeakRefs()` 来清空弱引用并调用其回调，然后调用终结器，最后调用此函数来清空终结器可能创建的任何弱引用。

在大多数情况下，更适当的做法是使用`PyObject_ClearWeakRefs()` 而不是此函数来清空弱引用。

Added in version 3.13.

8.6.8 Capsule 对象

有关使用这些对象的更多信息请参阅 using-capsules。

Added in version 3.1.

type **PyCapsule**

这个`PyObject` 的子类型代表一个隐藏的值，适用于需要将隐藏值（作为 `void*` 指针）通过 Python 代码传递到其他 C 代码的 C 扩展模块。它常常被用来让在一个模块中定义的 C 函数指针在其他模块中可用，这样就可以使用常规导入机制来访问在动态加载的模块中定义的 C API。

type **PyCapsule_Destructor**

属于稳定 ABI. Capsule 的析构器回调的类型。定义如下：

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

参阅*PyCapsule_New()* 来获取 PyCapsule_Destructor 返回值的语义。

int **PyCapsule_CheckExact**(*PyObject* *p)

如果参数是一个*PyCapsule* 则返回真值。此函数总是会成功执行。

PyObject ***PyCapsule_New**(void *pointer, const char *name, *PyCapsule_Destructor* destructor)

返回值：新的引用。属于稳定 ABI. 创建一个封装了 *pointer* 的*PyCapsule*。*pointer* 参考可以不为 NULL。

在失败时设置一个异常并返回 NULL。

字符串 *name* 可以是 NULL 或是一个指向有效的 C 字符串的指针。如果不为 NULL，则此字符串必须比 capsule 长（虽然也允许在 *destructor* 中释放它。）

如果 *destructor* 参数不为 NULL，则当它被销毁时将附带 capsule 作为参数来调用。

如果此 capsule 将被保存为一个模块的属性，则 *name* 应当被指定为 `modulename.attributename`。这将允许其他模块使用*PyCapsule_Import()* 来导入此 capsule。

void ***PyCapsule_GetPointer**(*PyObject* *capsule, const char *name)

属于稳定 ABI. 提取保存在 capsule 中的 *pointer*。在失败时设置一个异常并返回 NULL。

name 参数必须与 capsule 中存储的名称完全一致。如果存储在 capsule 中的名称是 NULL，传入的 *name* 也必须是 NULL。Python 使用 C 函数 `strcmp()` 来比较 capsule 名称。

PyCapsule_Destructor **PyCapsule_GetDestructor**(*PyObject* *capsule)

属于稳定 ABI. 返回保存在 capsule 中的当前析构器。在失败时设置一个异常并返回 NULL。

capsule 具有 NULL 析构器是合法的。这会使得 NULL 返回码有些歧义；请使用*PyCapsule_IsValid()* 或*PyErr_Occurred()* 来消除歧义。

void ***PyCapsule_GetContext**(*PyObject* *capsule)

属于稳定 ABI. 返回保存在 capsule 中的当前上下文。在失败时设置一个异常并返回 NULL。

capsule 具有 NULL 上下文是全法的。这会使得 NULL 返回码有些歧义；请使用*PyCapsule_IsValid()* 或*PyErr_Occurred()* 来消除歧义。

const char ***PyCapsule_GetName**(*PyObject* *capsule)

属于稳定 ABI. 返回保存在 capsule 中的当前名称。在失败时设置一个异常并返回 NULL。

capsule 具有 NULL 名称是合法的。这会使得 NULL 返回码有些歧义；请使用*PyCapsule_IsValid()* 或*PyErr_Occurred()* 来消除歧义。

void ***PyCapsule_Import**(const char *name, int no_block)

属于稳定 ABI. 从一个模块内的包装属性导入一个指向 C 对象的指针。*name* 形参应当指定该属性的完整名称，就像 `module.attribute` 这样。储存在包装中的 *name* 必须与此字符串完全匹配。

此函数会按 `.` 字符拆分 *name*，并导入第一个元素。随后它会使用属性查找来进行其他元素。

成功时返回 capsule 的内部 指针。在失败时设置一个异常并返回 NULL。

> **备注**
>
> 如果 *name* 指向某个子模块或子包的属性，要使属性查找能够成功这个子模块或子包必须在此之前已使用其他方式导入 (例如，通过使用*PyImport_ImportModule()*)。

在 3.3 版本发生变更: *no_block* 不再有任何影响。

int **PyCapsule_IsValid**(*PyObject* *capsule, const char *name)

属于稳定 ABI. 确定 *capsule* 是否是一个有效的。有效的 capsule 必须不为 NULL，传递*PyCapsule_CheckExact()*，在其中存储一个不为 NULL 的指针，并且其内部名称与 *name* 形参相匹配。（请参阅*PyCapsule_GetPointer()* 了解如何对 capsule 名称进行比较的有关信息。）

换句话说，如果*PyCapsule_IsValid()* 返回真值，则对任何访问器（以 PyCapsule_Get 开头的任何函数）的调用都保证会成功。

如果对象有效并且匹配传入的名称则返回非零值。否则返回 0。此函数一定不会失败。

int **PyCapsule_SetContext**(*PyObject* *capsule, void *context)

属于稳定 ABI. 将 *capsule* 内部的上下文指针设为 *context*。

成功时返回 0。失败时返回非零值并设置一个异常。

int **PyCapsule_SetDestructor**(*PyObject* *capsule, *PyCapsule_Destructor* destructor)

属于稳定 ABI. 将 *capsule* 内部的析构器设为 *destructor*。

成功时返回 0。失败时返回非零值并设置一个异常。

int **PyCapsule_SetName**(*PyObject* *capsule, const char *name)

属于稳定 ABI. 将 *capsule* 内部的名称设为 *name*。如果不为 NULL，则名称的存在期必须比 capsule 更长。如果之前保存在 capsule 中的 *name* 不为 NULL，则不会尝试释放它。

成功时返回 0。失败时返回非零值并设置一个异常。

int **PyCapsule_SetPointer**(*PyObject* *capsule, void *pointer)

属于稳定 ABI. 将 *capsule* 内部的空指针设为 *pointer*。指针不可为 NULL。

成功时返回 0。失败时返回非零值并设置一个异常。

8.6.9 帧对象

type **PyFrameObject**

属于受限 API （作为不透明的结构体）. 用于描述帧对象的对象 C 结构体。

此结构体中无公有成员。

在 3.11 版本发生变更: 此结构体的成员已从公有 C API 中移除。请参阅 What's New entry 了解详情。

可以使用函数*PyEval_GetFrame()* 与*PyThreadState_GetFrame()* 去获取一个帧对象。

可参考：*Reflection 1*

PyTypeObject **PyFrame_Type**

帧对象的类型。它与 Python 层中的 types.FrameType 是同一对象。

在 3.11 版本发生变更: 在之前版本中，此类型仅在包括 <frameobject.h> 之后可用。

int **PyFrame_Check**(*PyObject* *obj)

如果 *obj* 是一个帧对象则返回非零值。

在 3.11 版本发生变更: 在之前版本中，只函数仅在包括 <frameobject.h> 之后可用。

PyFrameObject ***PyFrame_GetBack**(*PyFrameObject* *frame)

返回值：新的引用。 获取 *frame* 为下一个外部帧。

返回一个*strong reference*，或者如果 *frame* 没有外部帧则返回 NULL。

Added in version 3.9.

PyObject ***PyFrame_GetBuiltins**(*PyFrameObject* *frame)

返回值：新的引用。 获取 *frame* 的 f_builtins 属性。

返回一个*strong reference*。此结果不可为 NULL。

Added in version 3.11.

PyCodeObject ***PyFrame_GetCode**(*PyFrameObject* *frame)

返回值：新的引用。属于稳定 ABI 自 *3.10* 版起. 获取 *frame* 的代码。

返回一个*strong reference*。

结果（帧代码）不可为 `NULL`。

Added in version 3.9.

PyObject ***PyFrame_GetGenerator**(*PyFrameObject* *frame)

返回值：新的引用。 获取拥有该帧的生成器、协程或异步生成器，或者如果该帧不被某个生成器所拥有则为 `NULL`。不会引发异常，即使其返回值为 `NULL`。

返回一个*strong reference*，或者 `NULL`。

Added in version 3.11.

PyObject ***PyFrame_GetGlobals**(*PyFrameObject* *frame)

返回值：新的引用。 获取 *frame* 的 `f_globals` 属性。

返回一个*strong reference*。此结果不可为 `NULL`。

Added in version 3.11.

int **PyFrame_GetLasti**(*PyFrameObject* *frame)

获取 *frame* 的 `f_lasti` 属性。

如果 `frame.f_lasti` 为 `None` 则返回 -1。

Added in version 3.11.

PyObject ***PyFrame_GetVar**(*PyFrameObject* *frame, *PyObject* *name)

返回值：新的引用。 获取 *frame* 的变量 *name*。

- 成功时返回一个指向变量值的*strong reference*。
- 引发 `NameError` 并返回 `NULL` 如果该变量不存在。
- 引发异常并返回 “NULL“错误。

name 必须是 `str` 类型的。

Added in version 3.12.

PyObject ***PyFrame_GetVarString**(*PyFrameObject* *frame, const char *name)

返回值：新的引用。 和*`PyFrame_GetVar()`* 相似，但该变量名是一个使用 UTF-8 编码的 C 字符串。

Added in version 3.12.

PyObject ***PyFrame_GetLocals**(*PyFrameObject* *frame)

返回值：新的引用。 获取 *frame* 的 `f_locals` 属性。如果该帧指向一个*optimized scope*，这将返回一个允许修改 locals 的直通写入代理对象。在所有其他情况下 (类、模块、`exec()`、`eval()`) 它将直接返回代表该帧的 locals 的映射 (如为 `locals()` 所描述的)。

返回一个*strong reference*。

Added in version 3.11.

在 3.13 版本发生变更: 作为 **PEP 667** 的组成部分，返回一个*`PyFrameLocalsProxy_Type`* 的实例。

int **PyFrame_GetLineNumber**(*PyFrameObject* *frame)

属于稳定 ABI 自 *3.10* 版起. 返回 *frame* 当前正在执行的行号。

帧 locals 代理

Added in version 3.13.

帧对象的 `f_locals` 属性是“帧 locals 代理”的一个实例。该代理对象将对外公开一个下层帧 locals 字典的直写视图。这确保了由 `f_locals` 暴露的变量总是与帧本身的现有局部变量内容一致。

请参阅 **PEP 667** 了解详情。

PyTypeObject **PyFrameLocalsProxy_Type**

帧 `locals()` 代理对象的类型。

int **PyFrameLocalsProxy_Check**(*PyObject* *obj)

如果 *obj* 是一个帧 `locals()` 代理则返回非零值。

内部帧

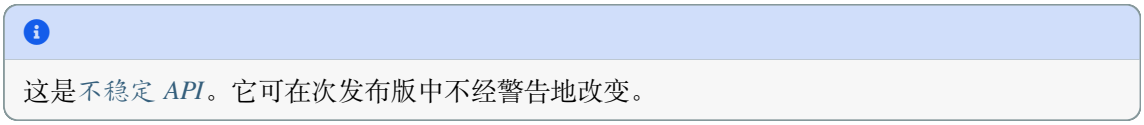
除非使用:pep:*523*，否则你不会需要它。

struct **_PyInterpreterFrame**

解释器的内部帧表示。

Added in version 3.11.

PyObject ***PyUnstable_InterpreterFrame_GetCode**(struct *_PyInterpreterFrame* *frame);

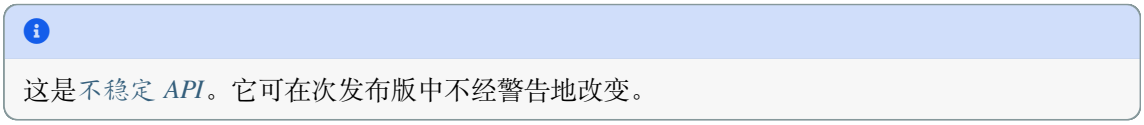


这是不稳定 *API*。它可在次发布版中不经警告地改变。

返回一个指向帧的代码对象的*strong reference*。

Added in version 3.12.

int **PyUnstable_InterpreterFrame_GetLasti**(struct *_PyInterpreterFrame* *frame);

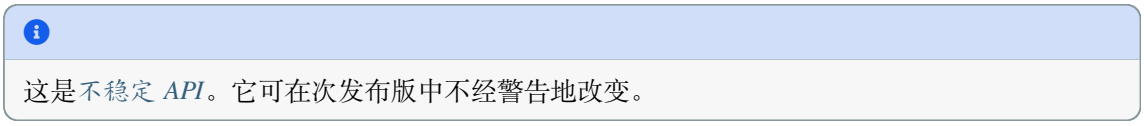


这是不稳定 *API*。它可在次发布版中不经警告地改变。

将字节偏移量返回到最后执行的指令中。

Added in version 3.12.

int **PyUnstable_InterpreterFrame_GetLine**(struct *_PyInterpreterFrame* *frame);



这是不稳定 *API*。它可在次发布版中不经警告地改变。

返回正在执行的指令的行数，如果没有行数，则返回-1。

Added in version 3.12.

8.6.10 生成器对象

生成器对象是 Python 用来实现生成器迭代器的对象。它们通常通过迭代产生值的函数来创建，而不是显式调用*PyGen_New()* 或*PyGen_NewWithQualName()*。

type **PyGenObject**

用于生成器对象的 C 结构体。

PyTypeObject **PyGen_Type**

与生成器对象对应的类型对 象。

int **PyGen_Check**(*PyObject* *ob)

如果 *ob* 是一个 generator 对象则返回真值；*ob* 必须不为 NULL。此函数总是会成功执行。

int **PyGen_CheckExact**(*PyObject* *ob)

如果 *ob* 的类型是*PyGen_Type* 则返回真值；*ob* 必须不为 NULL。此函数总是会成功执行。

PyObject ***PyGen_New**(*PyFrameObject* *frame)

返回值：新的引用。 基于 *frame* 对象创建并返回一个新的生成器对象。此函数会取走一个对 *frame* 的引用。参数必须不为 NULL。

PyObject ***PyGen_NewWithQualName**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

返回值：新的引用。 基于 *frame* 对象创建并返回一个新的生成器对象，其中 __name__ 和 __qualname__ 设为 *name* 和 *qualname*。此函数会取走一个对 *frame* 的引用。*frame* 参数必须不为 NULL。

8.6.11 协程对象

Added in version 3.5.

协程对象是使用 async 关键字声明的函数返回的。

type **PyCoroObject**

用于协程对象的 C 结构体。

PyTypeObject **PyCoro_Type**

与协程对象对应的类型对 象。

int **PyCoro_CheckExact**(*PyObject* *ob)

如果 *ob* 的类型是*PyCoro_Type* 则返回真值；*ob* 必须不为 NULL。此函数总是会成功执行。

PyObject ***PyCoro_New**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

返回值：新的引用。基于 *frame* 对象创建并返回一个新的协程对象，其中 __name__ 和 __qualname__ 设为 *name* 和 *qualname*。此函数会取得一个对 *frame* 的引用。*frame* 参数必须不为 NULL。

8.6.12 上下文变量对象

Added in version 3.7.

在 3.7.1 版本发生变更:

备注

在 Python 3.7.1 中，所有上下文变量 C API 的签名被 **更改**为使用*PyObject* 指针而不是*PyContext*，*PyContextVar* 以及*PyContextToken*，例如:

```
// 在3.7.0:
PyContext *PyContext_New(void);

// 在3.7.1+:
PyObject *PyContext_New(void);
```

请参阅 bpo-34762 了解详情。

本节深入介绍了 contextvars 模块的公用 C API。

type **PyContext**
用于表示 `contextvars.Context` 对象的 C 结构体。

type **PyContextVar**
用于表示 `contextvars.ContextVar` 对象的 C 结构体。

type **PyContextToken**
用于表示 `contextvars.Token` 对象的 C 结构体。

PyTypeObject **PyContext_Type**
表示 *context* 类型的类型对象。

PyTypeObject **PyContextVar_Type**
表示 *context variable* 类型的类型对象。

PyTypeObject **PyContextToken_Type**
表示 *context variable token* 类型的类型对象。

类型检查宏:

int **PyContext_CheckExact**(*PyObject* *o)
如果 *o* 的类型为 *PyContext_Type* 则返回真值。*o* 必须不为 `NULL`。此函数总是会成功执行。

int **PyContextVar_CheckExact**(*PyObject* *o)
如果 *o* 的类型为 *PyContextVar_Type* 则返回真值。*o* 必须不为 `NULL`。此函数总是会成功执行。

int **PyContextToken_CheckExact**(*PyObject* *o)
如果 *o* 的类型为 *PyContextToken_Type* 则返回真值。*o* 必须不为 `NULL`。此函数总是会成功执行。

上下文对象管理函数:

PyObject ***PyContext_New**(void)
返回值：新的引用。 创建一个新的空上下文对象。如果发生错误则返回 `NULL`。

PyObject ***PyContext_Copy**(*PyObject* *ctx)
返回值：新的引用。 创建所传入的 *ctx* 上下文对象的浅拷贝。如果发生错误则返回 `NULL`。

PyObject ***PyContext_CopyCurrent**(void)
返回值：新的引用。 创建当前线程上下文的浅拷贝。如果发生错误则返回 `NULL`。

int **PyContext_Enter**(*PyObject* *ctx)
将 *ctx* 设为当前线程的当前上下文。成功时返回 `0`，出错时返回 `-1`。

int **PyContext_Exit**(*PyObject* *ctx)
取消激活 *ctx* 上下文并将之前的上下文恢复为当前线程的当前上下文。成功时返回 `0`，出错时返回 `-1`。

上下文变量函数:

PyObject ***PyContextVar_New**(const char *name, *PyObject* *def)
返回值：新的引用。 创建一个新的 `ContextVar` 对象。形参 *name* 用于自我检查和调试目的。形参 *def* 为上下文变量指定默认值，或为 `NULL` 表示无默认值。如果发生错误，这个函数会返回 `NULL`。

int **PyContextVar_Get**(*PyObject* *var, *PyObject* *default_value, *PyObject* **value)
获取上下文变量的值。如果在查找过程中发生错误，返回’’-1’’，如果没有发生错误，无论是否找到值，都返回’’0’’，

如果找到上下文变量，*value* 将是指向它的指针。如果上下文变量 *没有找到*，*value* 将指向：

- *default_value*，如果非 `NULL`;
- *var* 的默认值，如果不是 `NULL`；
- `NULL`

除了返回 `NULL`，这个函数会返回一个新的引用。

PyObject ***PyContextVar_Set**(*PyObject* *var, *PyObject* *value)

返回值：新的引用。 在当前上下文中将 *var* 设为 *value*。返回针对此修改的新凭据对象，或者如果发生错误则返回 NULL。

int **PyContextVar_Reset**(*PyObject* *var, *PyObject* *token)

将上下文变量 *var* 的状态重置为它在返回 *token* 的 *PyContextVar_Set()* 被调用之前的状态。此函数成功时返回 0，出错时返回 -1。

8.6.13 DateTime 对象

datetime 模块提供了各种日期和时间对象。在使用这些函数之前，必须在你的源代码中包含头文件 datetime.h (请注意此文件并未包括在 Python.h 中)，并且 PyDateTime_IMPORT 必须被唤起，通常是作为模块初始化函数的一部分。这个宏会将指向特定 C 结构体的指针放入一个静态变量 PyDateTimeAPI 中，它将被下列的宏所使用。

type **PyDateTime_Date**

PyObject 的这个子类型表示 Python 日期对象。

type **PyDateTime_DateTime**

PyObject 的这个子类型表示 Python 日期时间对象。

type **PyDateTime_Time**

PyObject 的这个子类型表示 Python 时间对象。

type **PyDateTime_Delta**

PyObject 的这个子类型表示两个日期时间值之间的差值。

PyTypeObject **PyDateTime_DateType**

这个 *PyTypeObject* 的实例代表 Python 日期类型；它与 Python 层面的 datetime.date 对象相同。

PyTypeObject **PyDateTime_DateTimeType**

这个 *PyTypeObject* 的实例代表 Python 日期时间类型；它与 Python 层面的 datetime.datetime 对象相同。

PyTypeObject **PyDateTime_TimeType**

这个 *PyTypeObject* 的实例代表 Python 时间类型；它与 Python 层面的 datetime.time 对象相同。

PyTypeObject **PyDateTime_DeltaType**

这个 *PyTypeObject* 的实例是代表两个日期时间值之间差值的 Python 类型；它与 Python 层面的 datetime.timedelta 对象相同。

PyTypeObject **PyDateTime_TZInfoType**

这个 *PyTypeObject* 的实例代表 Python 时区信息类型；它与 Python 层面的 datetime.tzinfo 对象相同。

宏访问 UTC 单例:

PyObject ***PyDateTime_TimeZone_UTC**

返回表示 UTC 的时区单例，与 datetime.timezone.utc 为同一对象。

Added in version 3.7.

类型检查宏:

int **PyDate_Check**(*PyObject* *ob)

如果 *ob* 为 *PyDateTime_DateType* 类型或 PyDateTime_DateType 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDate_CheckExact**(*PyObject* *ob)

如果 *ob* 为 *PyDateTime_DateType* 类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDateTime_Check**(*PyObject* *ob)

如果 *ob* 为*PyDateTime_DateTimeType* 类型或 PyDateTime_DateTimeType 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDateTime_CheckExact**(*PyObject* *ob)

如果 *ob* 为*PyDateTime_DateTimeType* 类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyTime_Check**(*PyObject* *ob)

如果 *ob* 为*PyDateTime_TimeType* 类型或 PyDateTime_TimeType 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyTime_CheckExact**(*PyObject* *ob)

如果 *ob* 为*PyDateTime_TimeType* 类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDelta_Check**(*PyObject* *ob)

如果 *ob* 为*PyDateTime_DeltaType* 类型或 PyDateTime_DeltaType 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDelta_CheckExact**(*PyObject* *ob)

如果 *ob* 为*PyDateTime_DeltaType* 类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyTZInfo_Check**(*PyObject* *ob)

如果 *ob* 为*PyDateTime_TZInfoType* 类型或 PyDateTime_TZInfoType 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyTZInfo_CheckExact**(*PyObject* *ob)

如果 *ob* 为*PyDateTime_TZInfoType* 类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

用于创建对象的宏：

PyObject ***PyDate_FromDate**(int year, int month, int day)

返回值：新的引用。 返回指定年、月、日的 datetime.date 对象。

PyObject ***PyDateTime_FromDateAndTime**(int year, int month, int day, int hour, int minute, int second, int usecond)

返回值：新的引用。 返回具有指定 year, month, day, hour, minute, second 和 microsecond 属性的 datetime.datetime 对象。

PyObject ***PyDateTime_FromDateAndTimeAndFold**(int year, int month, int day, int hour, int minute, int second, int usecond, int fold)

返回值：新的引用。 返回具有指定 year, month, day, hour, minute, second, microsecond 和 fold 属性的 datetime.datetime 对象。

Added in version 3.6.

PyObject ***PyTime_FromTime**(int hour, int minute, int second, int usecond)

返回值：新的引用。 返回具有指定 hour, minute, second and microsecond 属性的 datetime.time 对象。

PyObject ***PyTime_FromTimeAndFold**(int hour, int minute, int second, int usecond, int fold)

返回值：新的引用。 返回具有指定 hour, minute, second, microsecond 和 fold 属性的 datetime.time 对象。

Added in version 3.6.

PyObject ***PyDelta_FromDSU**(int days, int seconds, int useconds)

返回值：新的引用。 返回代表给定天、秒和微秒数的 datetime.timedelta 对象。将执行正规化操作以使最终的微秒和秒数处在 datetime.timedelta 对象的文档指明的区间之内。

PyObject ***PyTimeZone_FromOffset**(*PyObject* *offset)

返回值：新的引用。 返回一个 datetime.timezone 对象，该对象具有以 *offset* 参数表示的未命名固定时差。

Added in version 3.7.

PyObject ***PyTimeZone_FromOffsetAndName**(*PyObject* *offset, *PyObject* *name)

返回值：新的引用。 返回一个 `datetime.timezone` 对象，该对象具有以 *offset* 参数表示的固定时差和时区名称 *name*。

Added in version 3.7.

一些用来从日期对象中提取字段的宏。参数必须是*PyDateTime_Date* 包括其子类(如*PyDateTime_DateTime*) 的实例。参数不能为 `NULL`，且不会检查类型:

int **PyDateTime_GET_YEAR**(*PyDateTime_Date* *o)

以正整数的形式返回年份值。

int **PyDateTime_GET_MONTH**(*PyDateTime_Date* *o)

返回月，从 0 到 12 的整数。

int **PyDateTime_GET_DAY**(*PyDateTime_Date* *o)

返回日期，从 0 到 31 的整数。

一些用来从日期时间对象中提取字段的宏。参数必须是*PyDateTime_DateTime* 包括其子类的实例。参数不能为 `NULL`，并且不会检查类型:

int **PyDateTime_DATE_GET_HOUR**(*PyDateTime_DateTime* *o)

返回小时，从 0 到 23 的整数。

int **PyDateTime_DATE_GET_MINUTE**(*PyDateTime_DateTime* *o)

返回分钟，从 0 到 59 的整数。

int **PyDateTime_DATE_GET_SECOND**(*PyDateTime_DateTime* *o)

返回秒，从 0 到 59 的整数。

int **PyDateTime_DATE_GET_MICROSECOND**(*PyDateTime_DateTime* *o)

返回微秒，从 0 到 999999 的整数。

int **PyDateTime_DATE_GET_FOLD**(*PyDateTime_DateTime* *o)

返回折叠值，为整数 0 或 1。

Added in version 3.6.

PyObject ***PyDateTime_DATE_GET_TZINFO**(*PyDateTime_DateTime* *o)

返回 tzinfo (可以为 `None`)。

Added in version 3.10.

一些用来从时间对象中提取字段的宏。参数必须是*PyDateTime_Time* 包括其子类的实例。参数不能为 `NULL`，且不会检查类型:

int **PyDateTime_TIME_GET_HOUR**(*PyDateTime_Time* *o)

返回小时，从 0 到 23 的整数。

int **PyDateTime_TIME_GET_MINUTE**(*PyDateTime_Time* *o)

返回分钟，从 0 到 59 的整数。

int **PyDateTime_TIME_GET_SECOND**(*PyDateTime_Time* *o)

返回秒，从 0 到 59 的整数。

int **PyDateTime_TIME_GET_MICROSECOND**(*PyDateTime_Time* *o)

返回微秒，从 0 到 999999 的整数。

int **PyDateTime_TIME_GET_FOLD**(*PyDateTime_Time* *o)

返回折叠值，为整数 0 或 1。

Added in version 3.6.

PyObject ***PyDateTime_TIME_GET_TZINFO**(*PyDateTime_Time* *o)

返回 tzinfo (可以为 None)。

Added in version 3.10.

一些用来从时间差对象中提取字段的宏。参数必须是*PyDateTime_Delta* 包括其子类的实例。参数不能为 NULL，并且不会检查类型:

int **PyDateTime_DELTA_GET_DAYS**(*PyDateTime_Delta* *o)

返回天数，从-999999999 到 999999999 的整数。

Added in version 3.3.

int **PyDateTime_DELTA_GET_SECONDS**(*PyDateTime_Delta* *o)

返回秒数，从 0 到 86399 的整数。

Added in version 3.3.

int **PyDateTime_DELTA_GET_MICROSECONDS**(*PyDateTime_Delta* *o)

返回微秒数，从 0 到 999999 的整数。

Added in version 3.3.

一些便于模块实现 DB API 的宏:

PyObject ***PyDateTime_FromTimestamp**(*PyObject* *args)

返回值：新的引用。 创建并返回一个给定参数元组的新 datetime.datetime 对象，适合传给 datetime.datetime.fromtimestamp()。

PyObject ***PyDate_FromTimestamp**(*PyObject* *args)

返回值：新的引用。 创建并返回一个新的 datetime.date 对象，给定一个适合传递给 datetime.date.fromtimestamp() 的参数元组

8.6.14 类型注解对象

提供几种用于类型提示的内置类型。目前存在两种类型 -- GenericAlias 和 Union。只有 GenericAlias 会向 C 开放。

PyObject ***Py_GenericAlias**(*PyObject* *origin, *PyObject* *args)

属于稳定 ABI 自 *3.9* 版起. 创建一个 GenericAlias 对象。相当于调用 Python 类 types.GenericAlias。参数 *origin* 和 *args* 分别设置 GenericAlias 的 __origin__ 和 __args__ 属性。*origin* 应该是一个 *PyTypeObject**，而 *args* 可以是一个 *PyTupleObject** 或者任意 PyObject*。如果传递的 *args* 不是一个元组，则会自动构造一个单元组并将 __args__ 设置为 (args,)。对参数进行了最小限度的检查，因此即使 *origin* 不是类型函数也会成功。GenericAlias 的 __parameters__ 属性是从 __args__ 懒加载的。如果失败，则会引发一个异常并返回 NULL。

下面是一个如何创建一个扩展类型泛型的例子:

```
...
static PyMethodDef my_obj_methods[] = {
    // 其他方法。
    ...
    {"__class_getitem__", Py_GenericAlias, METH_O|METH_CLASS, "See PEP 585"}
    ...
}
```

参见

数据模型方法 __class_getitem__()。

Added in version 3.9.

PyTypeObject **Py_GenericAliasType**

属于稳定 ABI 自 *3.9* 版起. 由*Py_GenericAlias()* 所返回的对象的 C 类型。等价于 Python 中的 `types.GenericAlias` 。

Added in version 3.9.

CHAPTER 9

初始化，最终化和线程

请参阅*Python 初始化配置* 了解如何在初始化之前配置解释器的详情。

9.1 在 Python 初始化之前

在一个植入了 Python 的应用程序中，`Py_Initialize()` 函数必须在任何其他 Python/C API 函数之前被调用；例外的只有个别函数和全局配置变量。

在初始化 Python 之前，可以安全地调用以下函数：

- 初始化解释器的函数：
  - `Py_Initialize()`
  - `Py_InitializeEx()`
  - `Py_InitializeFromConfig()`
  - `Py_BytesMain()`
  - `Py_Main()`
  - 运行时预初始化相关函数在*Python 初始化配置* 中介绍
- 配置函数：
  - `PyImport_AppendInittab()`
  - `PyImport_ExtendInittab()`
  - `PyInitFrozenExtensions()`
  - `PyMem_SetAllocator()`
  - `PyMem_SetupDebugHooks()`
  - `PyObject_SetArenaAllocator()`
  - `Py_SetProgramName()`
  - `Py_SetPythonHome()`
  - `PySys_ResetWarnOptions()`
  - 配置相关函数在*Python 初始化配置* 中介绍

- 信息函数:
  - *Py_IsInitialized()*
  - *PyMem_GetAllocator()*
  - *PyObject_GetArenaAllocator()*
  - *Py_GetBuildInfo()*
  - *Py_GetCompiler()*
  - *Py_GetCopyright()*
  - *Py_GetPlatform()*
  - *Py_GetVersion()*
  - *Py_IsInitialized()*
- 工具
  - *Py_DecodeLocale()*
  - 状态报告和工具相关函数在*Python 初始化配置* 中介绍
- 内存分配器:
  - *PyMem_RawMalloc()*
  - *PyMem_RawRealloc()*
  - *PyMem_RawCalloc()*
  - *PyMem_RawFree()*
- 同步:
  - *PyMutex_Lock()*
  - *PyMutex_Unlock()*

备注

虽然它们看起来与上面列出的某些函数类似，但以下函数 **不应** 在解释器被初始化之前调用: *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()*, *PyEval_InitThreads()* 和*Py_RunMain()*。

9.2 全局配置变量

Python 有负责控制全局配置中不同特性和选项的变量。这些标志默认被 命令行选项。

当一个选项设置一个旗标时，该旗标的值将是设置选项的次数。例如，`-b` 会将*Py_BytesWarningFlag* 设为 1 而 `-bb` 会将*Py_BytesWarningFlag* 设为 2.

int **Py_BytesWarningFlag**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.bytes_warning*，参见*Python 初始化配置*。

当将 `bytes` 或 `bytearray` 与 `str` 比较或者将 `bytes` 与 `int` 比较时发出警告。如果大于等于 `2` 则报错。

由 `-b` 选项设置。

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_DebugFlag**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.parser_debug*，参见*Python 初始化配置*。

开启解析器调试输出（限专家使用，依赖于编译选项）。

由 `-d` 选项和 `PYTHONDEBUG` 环境变量设置。

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_DontWriteBytecodeFlag**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.write_bytecode*，参见*Python 初始化配置*。

如果设置为非零, Python 不会在导入源代码时尝试写入 `.pyc` 文件

由 `-B` 选项和 `PYTHONDONTWRITEBYTECODE` 环境变量设置。

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_FrozenFlag**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.pathconfig_warnings*，参见*Python 初始化配置*。

当在*Py_GetPath()* 中计算模块搜索路径时屏蔽错误消息。

由 `_freeze_importlib` 和 `frozenmain` 程序使用的私有旗标。

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_HashRandomizationFlag**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.hash_seed* 和*PyConfig.use_hash_seed*，参见*Python 初始化配置*。

如果 `PYTHONHASHSEED` 环境变量被设为非空字符串则设为 `1`。

如果该旗标为非零值，则读取 `PYTHONHASHSEED` 环境变量来初始化加密哈希种子。

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_IgnoreEnvironmentFlag**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.use_environment*，参见*Python 初始化配置*。

忽略所有 `PYTHON*` 环境变量，例如可能设置的 `PYTHONPATH` 和 `PYTHONHOME`。

由 `-E` 和 `-I` 选项设置。

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_InspectFlag**

此 API 被保留用于向下兼容：应当改为采用设置*PyConfig.inspect*，参见*Python 初始化配置*。

当将脚本作为第一个参数传入或是使用了 `-c` 选项时，则会在执行该脚本或命令后进入交互模式，即使在 `sys.stdin` 并非一个终端时也是如此。

由 `-i` 选项和 `PYTHONINSPECT` 环境变量设置。

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_InteractiveFlag**

此 API 被保留用于向下兼容：应当改为采用设置*PyConfig.interactive*，参见*Python 初始化配置*。

由 `-i` 选项设置。

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_IsolatedFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.isolated*，参见*Python 初始化配置*。

以隔离模式运行 Python. 在隔离模式下 `sys.path` 将不包含脚本的目录或用户的 site-packages 目录。

由 `-I` 选项设置。

Added in version 3.4.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_LegacyWindowsFSEncodingFlag**

此 API 被保留用于向下兼容：应当改为设置*PyPreConfig.legacy_windows_fs_encoding*，参见*Python 初始化配置*。

如果该旗标为非零值，则使用 mbcs 编码和 "replace" 错误处理器，而不是 UTF-8 编码和 surrogatepass 错误处理器作用*filesystem encoding and error handler*。

如果 PYTHONLEGACYWINDOWSFSENCODING 环境变量被设为非空字符串则设为 1。

更多详情请参阅 **PEP 529**。

Availability: Windows.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_LegacyWindowsStdioFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.legacy_windows_stdio*，参见*Python 初始化配置*。

如果该旗标为非零值，则会使用 io.FileIO 而不是 io._WindowsConsoleIO 作为 sys 标准流。

如果 PYTHONLEGACYWINDOWSSTDIO 环境变量被设为非空字符串则设为 1。

有关更多详细信息，请参阅 **PEP 528**。

Availability: Windows.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_NoSiteFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.site_import*，参见*Python 初始化配置*。

禁用 site 的导入及其所附带的基于站点对 sys.path 的操作。如果 site 会在稍后被显式地导入也会禁用这些操作 (如果你希望触发它们则应调用 site.main())。

由 -S 选项设置。

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_NoUserSiteDirectory**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.user_site_directory*，参见*Python 初始化配置*。

不要将 用户 site-packages 目录添加到 sys.path。

由 -s 和 -I 选项以及 PYTHONNOUSERSITE 环境变量设置。

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_OptimizeFlag**

此 API 被保留用于向下兼容：应当改为*PyConfig.optimization_level*，参见*Python 初始化配置*。

由 -O 选项和 PYTHONOPTIMIZE 环境变量设置。

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_QuietFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.quiet*，参见*Python 初始化配置*。

即使在交互模式下也不显示版权和版本信息。

由 -q 选项设置。

Added in version 3.2.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_UnbufferedStdioFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.buffered_stdio*，参见*Python 初始化配置*。

强制 stdout 和 stderr 流不带缓冲。

由 `-u` 选项和 `PYTHONUNBUFFERED` 环境变量设置。

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_VerboseFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.verbose*，参见*Python 初始化配置*。

每次初始化模块时打印一条消息，显示加载模块的位置（文件名或内置模块）。如果大于或等于 `2`，则为搜索模块时检查的每个文件打印一条消息。此外还会在退出时提供模块清理信息。

由 `-v` 选项和 `PYTHONVERBOSE` 环境变量设置。

Deprecated since version 3.12, will be removed in version 3.14.

9.3 初始化和最终化解释器

void **Py_Initialize**()

属于稳定 ABI. 初始化 Python 解释器。在嵌入 Python 的应用程序中，它应当在使用任何其他 Python/C API 函数之前被调用；请参阅*在 Python 初始化之前* 了解少数的例外情况。

这将初始化已加载的模块表 (`sys.modules`)，并创建基础模块 `builtins`，`__main__` 和 `sys`。它还会初始化模块搜索路径 (`sys.path`)。它不会设置 `sys.argv`；对于此设置请使用*Python 初始化配置* API。当第二次被调用时（在未先调用*Py_FinalizeEx()* 的情况下）将不会执行任何操作。它没有返回值；如果初始化失败则会发生致命错误。

使用*Py_InitializeFromConfig()* 来自定义*Python 初始化配置*。

> **备注**
>
> 在 Windows 上，将控制台模式从 `O_TEXT` 改为 `O_BINARY`，这还将影响使用 C 运行时的非 Python 的控制台使用。

void **Py_InitializeEx**(int initsigs)

属于稳定 ABI. 如果 *initsigs* 为 `1` 则此函数的工作方式与*Py_Initialize()* 类似。如果 *initsigs* 为 `0`，它将跳过信号处理器的初始化注册，这在将 CPython 作为更大应用程序的一部分嵌入时会很有用处。

使用*Py_InitializeFromConfig()* 来自定义*Python 初始化配置*。

PyStatus **Py_InitializeFromConfig**(const *PyConfig* *config)

根据 *config* 配置来初始化 Python，如*使用 PyConfig 初始化* 中所描述的。.

请参阅*Python 初始化配置* 一节了解有关预初始化解释器，填充运行时配置结构体，以及查询所返回的状态结构体的详情。

int **Py_IsInitialized**()

属于稳定 ABI. 如果 Python 解释器已初始化，则返回真值（非零）；否则返回假值（零）。在调用*Py_FinalizeEx()* 之后，此函数将返回假值直到*Py_Initialize()* 再次被调用。

int **Py_IsFinalizing**()

属于稳定 ABI 自 *3.13* 版起. 如果主 Python 解释器*正在关闭* 则返回真（非零）值。在其他情况下返回假（零）值。

Added in version 3.13.

int **Py_FinalizeEx**()

属于稳定 ABI 自 *3.6* 版起. 撤销*Py_Initialize()* 所做的所有初始化和随后对 Python/C API 函数的使用，并销毁自上次调用*Py_Initialize()* 以来创建但尚未销毁的所有子解释器 (见下文*Py_NewInterpreter()*)。在理想情况下，这会释放 Python 解释器分配的所有内存。当第二次调用时（在没有再次调用*Py_Initialize()* 的情况下），该函数不执行任何操作。

由于这是*Py_Initialize()* 的逆向操作，因而它应当在激活同一解释器的同一线程中被调用。这意味着主线程和主解释器。当*Py_RunMain()* 仍然运行时则绝不应调用此函数。

通常返回值为 `0`。如果在最终化（刷新缓冲的数据）期间发生错误，则返回 `-1`。

提供此函数的原因有很多。嵌入应用程序可能希望重新启动 Python，而不必重新启动应用程序本身。从动态可加载库（或 DLL）加载 Python 解释器的应用程序可能希望在卸载 DLL 之前释放 Python 分配的所有内存。在搜索应用程序内存泄漏的过程中，开发人员可能希望在退出应用程序之前释放 Python 分配的所有内存。

程序问题和注意事项: 模块和模块中对象的销毁是按随机顺序进行的；这可能导致依赖于其他对象（甚至函数）或模块的析构器（即 `__del__()` 方法）出错。Python 所加载的动态加载扩展模块不会被卸载。Python 解释器所分配的少量内存可能不会被释放（如果发现内存泄漏，请报告问题）。对象间循环引用所占用的内存不会被释放。扩展模块所分配的某些内存可能不会被释放。如果某些扩展的初始化例程被调用多次它们可能无法正常工作；如果应用程序多次调用了*Py_Initialize()* 和*Py_FinalizeEx()* 就可能发生这种情况。

引发一个不带参数的 审计事件 `cpython._PySys_ClearAuditHooks`。

Added in version 3.6.

void **Py_Finalize**()

属于稳定 ABI. 这是一个不考虑返回值的*Py_FinalizeEx()* 的向下兼容版本。

int **Py_BytesMain**(int argc, char **argv)

属于稳定 ABI 自 *3.8* 版起. 类似于*Py_Main()* 但 *argv* 是一个字节串数组，允许调用方应用程序将文本编码步骤委托给 CPython 运行时。

Added in version 3.8.

int **Py_Main**(int argc, wchar_t **argv)

属于稳定 ABI. 标准解释器的主程序，封装了完整的初始化/最终化循环，以及一些附加行为以实现从环境和命令行读取配置设置，然后按照 using-on-cmdline 的规则执行 `__main__`。

这适用于希望支持完整 CPython 命令行界面的程序，而不仅是在更大应用程序中嵌入 Python 运行时。

argc 和 *argv* 形参与传给 C 程序的 `main()` 函数的形参类似，不同之处在于 *argv* 的条目会先使用*Py_DecodeLocale()* 转换为 `wchar_t`。还有一个重要的注意事项是参数列表条目可能会被修改为指向并非被传入的字符串（不过，参数列表所指向的字符串内容不会被修改）。

如果参数列表不是表示一个有效的 Python 命令行则返回值为 `2`，否则将与*Py_RunMain()* 相同。

在记录于运行时配置 一节的 CPython 运行时配置 API 文档中（不考虑错误处理），`Py_Main` 大致相当于:

```
PyConfig config;
PyConfig_InitPythonConfig(&config);
PyConfig_SetArgv(&config, argc, argv);
Py_InitializeFromConfig(&config);
PyConfig_Clear(&config);

Py_RunMain();
```

在正常使用中，嵌入式应用程序将调用此函数 *而不是* 直接调用*Py_Initialize()*，*Py_InitializeEx()* 或*Py_InitializeFromConfig()*，并且所有设置都将如本文档的其他部分所描述的那样被应用。如果此函数改在某个先前的运行时初始化 API 调用 *之后* 被调用，那么到底那个环境和命令行配置会被更新将取决于具体的版本（因为它要依赖当运行时被初始化时究竟有哪些设置在它们已被设置一次之后是正确地支持被修改的）。

int **Py_RunMain**(void)

在完整配置的 CPython 运行时中执行主模块。

执行在命令行或配置中指定的命令 (*PyConfig.run_command*)、脚本 (*PyConfig.run_filename*) 或模块 (*PyConfig.run_module*)。如果这些值均未设置，则使用 __main__ 模块的全局命令空间来运行交互式 Python 提示符 (REPL)。

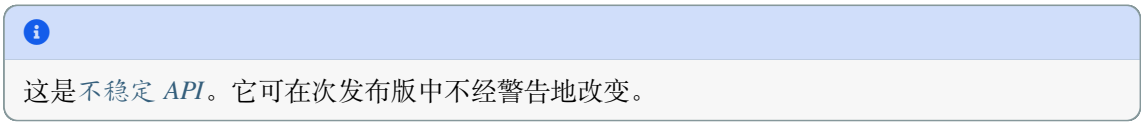
如果*PyConfig.inspect* 未设置（默认），则当解释器正常退出（也就是说未引发异常）时返回值将为 0，未处理的 SystemExit 的退出状态，或者对于任何其他未处理异常则为 1。

如果*PyConfig.inspect* 已设置（例如当使用了 -i 选项时），则当解释器退出时执行将不会返回，而是会使用 __main__ 模块的全局命名空间在交互式 Python 提示符 (REPL) 中恢复。如果解释器附带异常退出，该异常将在 REPL 会话中被立即引发。随后函数的返回值将由 *REPL 会话*的终结方式来决定: 0, 1 或者 SystemExit 的状态，如上文所指明的。

此函数总是会在它返回之前最终化 Python 解释器。

请参阅*Python 配置* 查看一个使用*Py_RunMain()* 在隔离模式下始终运行定制的 Python 的示例。

int **PyUnstable_AtExit**(*PyInterpreterState* *interp, void (*func)(void*), void *data)



这是不稳定 *API*。它可在次发布版中不经警告地改变。

为目标解释器 *interp* 注册一个 atexit 回调。这与*Py_AtExit()* 类似，但它接受一个显式的解释器和用于回调的数据指针。

必须为 *interp* 持有*GIL*。

Added in version 3.13.

9.4 进程级参数

void **Py_SetProgramName**(const wchar_t *name)

属于稳定 ABI. 此 API 被保留用于向下兼容：应当改为设置*PyConfig.program_name*，参见*Python 初始化配置*。

如果要调用该函数，应当在首次调用*Py_Initialize()* 之前调用它。它将告诉解释器程序的 main() 函数的 argv[0] 参数的值（转换为宽字符）。*Py_GetPath()* 和下面的某些其他函数会使用它在相对于解释器的位置上查找可执行文件的 Python 运行时库。默认值是 'python'。参数应当指向静态存储中的一个以零值结束的宽字符串，其内容在程序执行期间不会发生改变。Python 解释器中的任何代码都不会改变该存储的内容。

使用*Py_DecodeLocale()* 解码字节串以得到一个 wchar_t* 字符串。

自 3.11 版本弃用.

wchar_t ***Py_GetProgramName**()

属于稳定 ABI. 返回用*Py_SetProgramName()* 设置的程序名称，或默认的名称。返回的字符串指向静态存储；调用者不应修改其值。

此函数不应在*Py_Initialize()* 之前被调用，否则将返回 NULL。

在 3.10 版本发生变更: 现在如果它在*Py_Initialize()* 之前被调用将返回 NULL。

Deprecated since version 3.13, will be removed in version 3.15: 改为获取 sys.executable。

wchar_t ***Py_GetPrefix**()

属于稳定 ABI. 返回针对已安装的独立于平台文件的 *prefix*。这是通过基于使用*PyConfig.program_name* 设置的程序名称和某些环境变量所派生的一系列复杂规则来获取的；举例来说，如果程序名称为 '/usr/local/bin/python'，则 prefix 为 '/usr/local'。返回的字符串将指向静态存储；调用方不应修改其值。这对应于最高层级 Makefile 中的 **prefix** 变量以及在编译时传给

configure 脚本的 --prefix 参数。该值将作为 sys.base_prefix 供 Python 代码使用。它仅适用于 Unix。另请参见下一个函数。

此函数不应在*Py_Initialize()* 之前被调用，否则将返回 NULL。

在 3.10 版本发生变更: 现在如果它在*Py_Initialize()* 之前被调用将返回 NULL。

Deprecated since version 3.13, will be removed in version 3.15: 改为获取 sys.base_prefix，或者如果需要处理 虚拟环境则为获取 sys.prefix。

wchar_t ***Py_GetExecPrefix**()

属于稳定 ABI. 返回针对已安装的 依赖于平台文件的 *exec-prefix*。这是通过基于使用*PyConfig.program_name* 设置的程序名称和某些环境变量所派生的一系列复杂规则来获取的；举例来说，如果程序名称为 '/usr/local/bin/python'，则 exec-prefix 为 '/usr/local'。返回的字符串将指向静态存储；调用方不应修改其值。这对应于最高层级 Makefile 中的 **exec_prefix** 变量以及在编译时传给 **configure** 脚本的 --exec-prefix 参数。该值将作为 sys.base_exec_prefix 供 Python 代码使用。它仅适用于 Unix。

背景：当依赖于平台的文件（如可执行文件和共享库）是安装于不同的目录树中的时候 exec-prefix 将会不同于 prefix。在典型的安装中，依赖于平台的文件可能安装于 the /usr/local/plat 子目录树而独立于平台的文件可能安装于 /usr/local。

总而言之，平台是一组硬件和软件资源的组合，例如所有运行 Solaris 2.x 操作系统的 Sparc 机器会被视为相同平台，但运行 Solaris 2.x 的 Intel 机器是另一种平台，而运行 Linux 的 Intel 机器又是另一种平台。相同操作系统的不同主要发布版通常也会构成不同的平台。非 Unix 操作系统的情况又有所不同；这类系统上的安装策略差别巨大因此 prefix 和 exec-prefix 是没有意义的，并将被设为空字符串。请注意已编译的 Python 字节码是独立于平台的（但并不独立于它们编译时所使用的 Python 版本!）

系统管理员知道如何配置 **mount** 或 **automount** 程序以在平台间共享 /usr/local 而让 /usr/local/plat 成为针对不同平台的不同文件系统。

此函数不应在*Py_Initialize()* 之前被调用，否则将返回 NULL。

在 3.10 版本发生变更: 现在如果它在*Py_Initialize()* 之前被调用将返回 NULL。

Deprecated since version 3.13, will be removed in version 3.15: 改为获取 sys.base_exec_prefix，或者如果需要处理 虚拟环境则为获取 sys.exec_prefix。

wchar_t ***Py_GetProgramFullPath**()

属于稳定 ABI. 返回 Python 可执行文件的完整程序名称；这是作为基于程序名称（由*PyConfig.program_name* 设置）派生默认模块搜索路径的附带影响计算得出的。返回的字符串将指向静态存储；调用方不应修改其值。该值将以 sys.executable 的名称供 Python 代码访问。

此函数不应在*Py_Initialize()* 之前被调用，否则将返回 NULL。

在 3.10 版本发生变更: 现在如果它在*Py_Initialize()* 之前被调用将返回 NULL。

Deprecated since version 3.13, will be removed in version 3.15: 改为获取 sys.executable。

wchar_t ***Py_GetPath**()

属于稳定 ABI. 返回默认模块搜索路径；这是基于程序名称（由*PyConfig.program_name* 设置）和某些环境变量计算得出的。返回的字符串由一系列以依赖于平台的分隔符分开的目录名称组成。此分隔符在 Unix 和 macOS 上为 ':'，在 Windows 上为 ';'。返回的字符串将指向静态存储；调用方不应修改其值。列表 sys.path 将在解释器启动时使用该值来初始化；它可以在随后被修改（并且通常都会被修改）以变更用于加载模块的搜索路径。

此函数不应在*Py_Initialize()* 之前被调用，否则将返回 NULL。

在 3.10 版本发生变更: 现在如果它在*Py_Initialize()* 之前被调用将返回 NULL。

Deprecated since version 3.13, will be removed in version 3.15: 改为获取 sys.path。

const char ***Py_GetVersion**()

属于稳定 ABI. 返回 Python 解释器的版本。这将为如下形式的字符串

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

第一个单词（到第一个空格符为止）是当前的 Python 版本；前面的字符是以点号分隔的主要和次要版本号。返回的字符串将指向静态存储；调用方不应修改其值。该值将以 `sys.version` 的名称供 Python 代码使用。

另请参阅*Py_Version* 常量。

const char ***Py_GetPlatform**()

属于稳定 ABI. 返回当前平台的平台标识符。在 Unix 上，这将以操作系统的“官方”名称为基础，转换为小写形式，再加上主版本号；例如，对于 Solaris 2.x，或称 SunOS 5.x，该值将为 `'sunos5'`。在 macOS 上，它将为 `'darwin'`。在 Windows 上它将为 `'win'`。返回的字符串指向静态存储；调用方不应修改其值。Python 代码可通过 `sys.platform` 获取该值。

const char ***Py_GetCopyright**()

属于稳定 ABI. 返回当前 Python 版本的官方版权字符串，例如

`'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'`

返回的字符串指向静态存储；调用者不应修改其值。Python 代码可通过 `sys.copyright` 获取该值。

const char ***Py_GetCompiler**()

属于稳定 ABI. 返回用于编译当前 Python 版本的编译器指令，为带方括号的形式，例如:

```
"[GCC 2.7.2.2]"
```

返回的字符串指向静态存储；调用者不应修改其值。Python 代码可以从变量 `sys.version` 中获取该值。

const char ***Py_GetBuildInfo**()

属于稳定 ABI. 返回有关当前 Python 解释器实例的序列号和构建日期和时间的信息，例如:

```
"#67, Aug  1 1997, 22:34:28"
```

返回的字符串指向静态存储；调用者不应修改其值。Python 代码可以从变量 `sys.version` 中获取该值。

void **PySys_SetArgvEx**(int argc, wchar_t **argv, int updatepath)

属于稳定 ABI. 此 API 被保留用于向下兼容：应当改为设置*PyConfig.argv*, *PyConfig.parse_argv* 和*PyConfig.safe_path*，参见*Python 初始化配置*。

根据 *argc* 和 *argv* 设置 `sys.argv`。这些形参与传给程序的 `main()` 函数的类似，区别在于第一项应当指向要执行的脚本文件而不是 Python 解释器对应的可执行文件。如果没有要运行的脚本，则 *argv* 中的第一项可以为空字符串。如果此函数无法初始化 `sys.argv`，则将使用*Py_FatalError()* 发出严重情况信号。

如果 *updatepath* 为零，此函数将完成操作。如果 *updatepath* 为非零值，则此函数还将根据以下算法修改 `sys.path`:

- 如果在 `argv[0]` 中传入一个现有脚本，则脚本所在目录的绝对路径将被添加到 `sys.path` 的开头。
- 在其他情况下 (也就是说，如果 *argc* 为 `0` 或 `argv[0]` 未指向现有文件名)，则将在 `sys.path` 的开头添加一个空字符串，这等价于添加当前工作目录 (`"."`)。

使用*Py_DecodeLocale()* 解码字节串以得到一个 `wchar_t*` 字符串。

另请参阅*Python 初始化配置* 的*PyConfig.orig_argv* 和*PyConfig.argv* 成员。

备注

建议在出于执行单个脚本以外的目的嵌入 Python 解释器的应用传入 `0` 作为 *updatepath*，并在需要时更新 `sys.path` 本身。参见 **CVE 2008-5983**。

在 3.1.3 之前的版本中，你可以通过在调用*PySys_SetArgv()* 之后手动弹出第一个 sys.path 元素，例如使用:

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

Added in version 3.1.3.

自 3.11 版本弃用.

void **PySys_SetArgv**(int argc, wchar_t **argv)

属于稳定 ABI. 此 API 仅为向下兼容而保留：应当改为设置*PyConfig.argv* 并改用*PyConfig.parse_argv*，参见*Python 初始化配置*。

此函数相当于*PySys_SetArgvEx()* 设置了 *updatepath* 为 1 除非 **python** 解释器启动时附带了 -I。

使用*Py_DecodeLocale()* 解码字节串以得到一个 wchar_t* 字符串。

另请参阅*Python 初始化配置* 的*PyConfig.orig_argv* 和*PyConfig.argv* 成员。

在 3.4 版本发生变更: *updatepath* 值依赖于 -I。

自 3.11 版本弃用.

void **Py_SetPythonHome**(const wchar_t *home)

属于稳定 ABI. 此 API 被保留用于向下兼容：应当改为设置*PyConfig.home*，参见*Python 初始化配置*。

设置默认的"home" 目录，也就是标准 Python 库所在的位置。请参阅 PYTHONHOME 了解该参数字符串的含义。

此参数应当指向静态存储中一个以零值结束的字符串，其内容在程序执行期间将保持不变。Python 解释器中的代码绝不会修改此存储中的内容。

使用*Py_DecodeLocale()* 解码字节串以得到一个 wchar_t* 字符串。

自 3.11 版本弃用.

wchar_t ***Py_GetPythonHome**()

属于稳定 ABI. 返回默认的"home"，就是由*PyConfig.home* 所设置的值，或者在设置了 PYTHONHOME 环境变量的情况下则为该变量的值。

此函数不应在*Py_Initialize()* 之前被调用，否则将返回 NULL。

在 3.10 版本发生变更: 现在如果它在*Py_Initialize()* 之前被调用将返回 NULL。

Deprecated since version 3.13, will be removed in version 3.15: 改为获取*PyConfig.home* 或 PYTHONHOME 环境变量。

9.5 线程状态和全局解释器锁

Python 解释器不是完全线程安全的。为了支持多线程的 Python 程序，设置了一个全局锁，称为*global interpreter lock* 或*GIL*，当前线程必须在持有它之后才能安全地访问 Python 对象。如果没有这个锁，即使最简单的操作也可能在多线程的程序中导致问题：例如，当两个线程同时增加相同对象的引用计数时，引用计数可能最终只增加了一次而不是两次。

因此，规则要求只有获得*GIL* 的线程才能在 Python 对象上执行操作或调用 Python/C API 函数。为了模拟并发执行，解释器会定期尝试切换线程 (参见 sys.setswitchinterval())。锁也会在读写文件等可能造成阻塞的 I/O 操作时释放，以便其他 Python 线程可以同时运行。

Python 解释器会在一个名为*PyThreadState* 的数据结构体中保存一些线程专属的记录信息。还有一个全局变量指向当前的*PyThreadState*: 它可以使用*PyThreadState_Get()* 来获取。

9.5.1 从扩展扩展代码中释放 GIL

大多数操作*GIL* 的扩展代码具有以下简单结构:

```
将线程状态保存到一个局部变量中。
释放全局解释器锁。
... 执行某些阻塞式的 I/O 操作 ...
重新获取全局解释器锁。
从局部变量中恢复线程状态。
```

这是如此常用因此增加了一对宏来简化它:

```
Py_BEGIN_ALLOW_THREADS
... 执行某些阻塞式的 I/O 操作 ...
Py_END_ALLOW_THREADS
```

Py_BEGIN_ALLOW_THREADS 宏将打开一个新块并声明一个隐藏的局部变量；*Py_END_ALLOW_THREADS* 宏将关闭这个块。

上面的代码块可扩展为下面的代码:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... 执行某些阻塞式的 I/O 操作 ...
PyEval_RestoreThread(_save);
```

这些函数的工作原理如下：全局解释器锁被用来保护指向当前线程状态的指针。当释放锁并保存线程状态时，必须在锁被释放之前获取当前线程状态指针（因为另一个线程可以立即获取锁并将自己的线程状态存储到全局变量中）。相应地，当获取锁并恢复线程状态时，必须在存储线程状态指针之前先获取锁。

备注

调用系统 I/O 函数是释放 GIL 的最常见用例，但它在调用不需要访问 Python 对象的长期运行计算，比如针对内存缓冲区进行操作的压缩或加密函数之前也很有用。举例来说，在对数据执行压缩或哈希操作时标准 `zlib` 和 `hashlib` 模块就会释放 GIL。

9.5.2 非 Python 创建的线程

当使用专门的 Python API（如 `threading` 模块）创建线程时，会自动关联一个线程状态因而上面显示的代码是正确的。但是，如果线程是用 C 创建的（例如由具有自己的线程管理的第三方库创建），它们就不持有 GIL 也没有对应的线程状态结构体。

如果你需要从这些线程调用 Python 代码（这通常会是上述第三方库所提供的回调 API 的一部分），你必须首先通过创建线程状态数据结构体向解释器注册这些线程，然后获取 GIL，最后存储它们的线程状态指针，这样你才能开始使用 Python/C API。完成以上步骤后，你应当重置线程状态指针，释放 GIL，最后释放线程状态数据结构体。

PyGILState_Ensure() 和*PyGILState_Release()* 函数会自动完成上述的所有操作。从 C 线程调用到 Python 的典型方式如下:

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* 在此执行 Python 动作。 */
result = CallSomeFunction();
/* 评估结果或处理异常 */

/* 释放线程。 在此之后不再允许 Python API。 */
PyGILState_Release(gstate);
```

请注意 PyGILState_* 函数会假定只有一个全局解释器（由*Py_Initialize()* 自动创建）。Python 支持创建额外的解释器（使用*Py_NewInterpreter()* 创建），但不支持混合使用多个解释器和 PyGILState_* API。

9.5.3 有关 fork() 的注意事项

有关线程的另一个需要注意的重要问题是它们在面对 C fork() 调用时的行为。在大多数支持 fork() 的系统中，当一个进程执行 fork 之后将只有发出 fork 的线程存在。这对需要如何处理锁以及 CPython 的运行时内所有的存储状态都会有实质性的影响。

只保留“当前”线程这一事实意味着任何由其他线程所持有的锁永远不会被释放。Python 通过在 fork 之前获取内部使用的锁，并随后释放它们的方式为 os.fork() 解决了这个问题。此外，它还会重置子进程中的任何 lock-objects。在扩展或嵌入 Python 时，没有办法通知 Python 在 fork 之前或之后需要获取或重置的附加（非 Python）锁。需要使用 OS 工具例如 pthread_atfork() 来完成同样的事情。此外，在扩展或嵌入 Python 时，直接调用 fork() 而不是通过 os.fork() (并返回到或调用至 Python 中) 调用可能会导致某个被 fork 之后失效的线程所持有的 Python 内部锁发生死锁。*PyOS_AfterFork_Child()* 会尝试重置必要的锁，但并不总是能够做到。

所有其他线程都将结束这一事实也意味着 CPython 的运行时状态必须妥善清理，os.fork() 就是这样做的。这意味着最终化归属于当前解释器的所有其他*PyThreadState* 对象以及所有其他*PyInterpreterState* 对象。由于这一点以及*”main” 解释器* 的特殊性质，fork() 应当只在该解释器的”main” 线程中被调用，而 CPython 全局运行时最初就是在该线程中初始化的。只有当 exec() 将随后立即被调用的情况是唯一的例外。

9.5.4 高阶 API

这些是在编写 C 扩展代码或在嵌入 Python 解释器时最常用的类型和函数：

type **PyInterpreterState**

属于受限 API（作为不透明的结构体）. 该数据结构代表多个合作线程所共享的状态。属于同一解释器的线程将共享其模块管理以及其他一些内部条目。该结构体中不包含公有成员。

最初归属于不同解释器的线程不会共享任何东西，但进程状态如可用内存、打开的文件描述符等等除外。全局解释器锁也会被所有线程共享，无论它们归属于哪个解释器。

type **PyThreadState**

属于受限 API（作为不透明的结构体）. 该数据结构代表单个线程的状态。唯一的公有数据成员为：

PyInterpreterState ***interp**

该线程的解释器状态。

void **PyEval_InitThreads**()

属于稳定 ABI. 不执行任何操作的已弃用函数。

在 Python 3.6 及更老的版本中，此函数会在 GIL 不存在时创建它。

在 3.9 版本发生变更: 此函数现在不执行任何操作。

在 3.7 版本发生变更: 该函数现在由*Py_Initialize()* 调用，因此你无需再自行调用它。

在 3.2 版本发生变更: 此函数已不再被允许在*Py_Initialize()* 之前调用。

自 3.9 版本弃用.

PyThreadState ***PyEval_SaveThread**()

属于稳定 ABI. 释放全局解释器锁 (如果已创建) 并将线程状态重置为 NULL，返回之前的线程状态 (不为 NULL)。如果锁已被创建，则当前线程必须已获取到它。

void **PyEval_RestoreThread**(*PyThreadState* *tstate)

属于稳定 ABI. 获取全局解释器锁 (如果已创建) 并将线程状态设为 *tstate*，它必须不为 NULL。如果锁已被创建，则当前线程必须尚未获取它，否则将发生死锁。

> **备注**
>
> 当运行时正在最终化时从某个线程调用此函数将终结该线程，即使线程不是由 Python 创建的。你可以在调用此函数之前使用*Py_IsFinalizing()* 或 `sys.is_finalizing()` 来检查解释器是否还处于最终化过程中以避免不必要的终结操作。

PyThreadState ***PyThreadState_Get**()

属于稳定 ABI. 返回当前线程状态。全局解释器锁必须被持有。在当前状态为 `NULL` 时，这将发出一个致命错误 (这样调用方将无须检查是否为 `NULL`)。

另请参阅*PyThreadState_GetUnchecked()*。

PyThreadState ***PyThreadState_GetUnchecked**()

与*PyThreadState_Get()* 类似，但如果其为 NULL 则不会杀死进程并设置致命错误。调用方要负责检查结果是否为 NULL。

Added in version 3.13: 在 Python 3.5 到 3.12 中，此函数是私有的并且命名为 `_PyThreadState_UncheckedGet()`。

PyThreadState ***PyThreadState_Swap**(*PyThreadState* *tstate)

属于稳定 ABI. 交换当前线程状态与由可能为 `NULL` 的参数 *tstate* 所给出的线程状态。

GIL 不需要被持有，但当 *tstate* 不为 `NULL` 时都会被持有直到返回。

下列函数使用线程级本地存储，并且不能兼容子解释器：

PyGILState_STATE **PyGILState_Ensure**()

属于稳定 ABI. 确保当前线程已准备好调用 Python C API 而不管 Python 或全局解释器锁的当前状态如何。只要每次调用都与*PyGILState_Release()* 的调用相匹配就可以通过线程调用此函数任意多次。一般来说，只要线程状态恢复到 Release() 之前的状态就可以在*PyGILState_Ensure()* 和*PyGILState_Release()* 调用之间使用其他与线程相关的 API。例如，可以正常使用*Py_BEGIN_ALLOW_THREADS* 和*Py_END_ALLOW_THREADS* 宏。

返回值是一个当*PyGILState_Ensure()* 被调用时的线程状态的不透明“句柄”，并且必须被传递给*PyGILState_Release()* 以确保 Python 处于相同状态。虽然允许递归调用，但这些句柄不能被共享——每次对*PyGILState_Ensure()* 的单独调用都必须保存其对*PyGILState_Release()* 的调用的句柄。

当该函数返回时，当前线程将持有 GIL 并能够调用任意 Python 代码。执行失败将导致致命级错误。

> **备注**
>
> 当运行时正在最终化时从某个线程调用此函数将终结该线程，即使线程不是由 Python 创建的。你可以在调用此函数之前使用*Py_IsFinalizing()* 或 `sys.is_finalizing()` 来检查解释器是否还处于最终化过程中以避免不必要的终结操作。

void **PyGILState_Release**(PyGILState_STATE)

属于稳定 ABI. 释放之前获取的任何资源。在此调用之后，Python 的状态将与其在对相应*PyGILState_Ensure()* 调用之前的一样 (但是通常此状态对调用方来说将是未知的，对 GILState API 的使用也是如此)。

对*PyGILState_Ensure()* 的每次调用都必须与在同一线程上对*PyGILState_Release()* 的调用相匹配。

PyThreadState ***PyGILState_GetThisThreadState**()

属于稳定 ABI. 获取此线程的当前线程状态。如果当前线程上没有使用过 GILState API 则可以返回 `NULL`。请注意主线程总是会有这样一个线程状态，即使没有在主线程上执行过自动线程状态调用。这主要是一个辅助/诊断函数。

int **PyGILState_Check**()

如果当前线程持有 GIL 则返回 1 否则返回 0。此函数可以随时从任何线程调用。只有当它的 Python 线程状态已经初始化并且当前持有 GIL 时它才会返回 1。这主要是一个辅助/诊断函数。例如在回调上下文或内存分配函数中会很有用处，当知道 GIL 被锁定时可以允许调用方执行敏感的操作或是在其他情况下做出不同的行为。

Added in version 3.4.

以下的宏被使用时通常不带末尾分号；请在 Python 源代码发布包中查看示例用法。

Py_BEGIN_ALLOW_THREADS

属于稳定 ABI. 此宏会扩展为 `{ PyThreadState *_save; _save = PyEval_SaveThread();`。请注意它包含一个开头花括号；它必须与后面的*Py_END_ALLOW_THREADS* 宏匹配。有关此宏的进一步讨论请参阅上文。

Py_END_ALLOW_THREADS

属于稳定 ABI. 此宏扩展为 `PyEval_RestoreThread(_save); }`。注意它包含一个右花括号；它必须与之前的*Py_BEGIN_ALLOW_THREADS* 宏匹配。请参阅上文以进一步讨论此宏。

Py_BLOCK_THREADS

属于稳定 ABI. 这个宏扩展为 `PyEval_RestoreThread(_save);`: 它等价于没有关闭花括号的*Py_END_ALLOW_THREADS*。

Py_UNBLOCK_THREADS

属于稳定 ABI. 这个宏扩展为 `_save = PyEval_SaveThread();`: 它等价于没有开始花括号和变量声明的*Py_BEGIN_ALLOW_THREADS*。

9.5.5 底层级 API

下列所有函数都必须在*Py_Initialize()* 之后被调用。

在 3.7 版本发生变更: *Py_Initialize()* 现在会初始化*GIL*。

PyInterpreterState ***PyInterpreterState_New**()

属于稳定 ABI. 创建一个新的解释器状态对象。不需要持有全局解释器锁，但如果有必要序列化对此函数的调用则可能会持有。

引发一个不带参数的 审计事件 `cpython.PyInterpreterState_New`。

void **PyInterpreterState_Clear**(*PyInterpreterState* *interp)

属于稳定 ABI. 重置解释器状态对象中的所有信息。必须持有全局解释器锁。

引发一个不带参数的 审计事件 `cpython.PyInterpreterState_Clear`。

void **PyInterpreterState_Delete**(*PyInterpreterState* *interp)

属于稳定 ABI. 销毁解释器状态对象。不需要持有全局解释器锁。解释器状态必须使用之前对*PyInterpreterState_Clear()* 的调用来重置。

PyThreadState ***PyThreadState_New**(*PyInterpreterState* *interp)

属于稳定 ABI. 创建属于给定解释器对象的新线程状态对象。全局解释器锁不需要保持，但如果需要序列化对此函数的调用，则可以保持。

void **PyThreadState_Clear**(*PyThreadState* *tstate)

属于稳定 ABI. 重置线程状态对象中的所有信息。必须持有全局解释器锁。

在 3.9 版本发生变更: 此函数现在会调用 `PyThreadState.on_delete` 回调。在之前版本中，此操作是发生在*PyThreadState_Delete()* 中的。

在 3.13 版本发生变更: `PyThreadState.on_delete` 回调已被移除。

void **PyThreadState_Delete**(*PyThreadState* *tstate)

属于稳定 ABI. 销毁线程状态对象。不需要持有全局解释器锁。线程状态必须使用之前对*PyThreadState_Clear()* 的调用来重置。

void **PyThreadState_DeleteCurrent**(void)

销毁当前线程状态并释放全局解释器锁。与*PyThreadState_Delete()* 类似，必须持有全局解释器锁。线程状态必须已通过之前对*PyThreadState_Clear()* 的调用来重置。

PyFrameObject ***PyThreadState_GetFrame**(*PyThreadState* *tstate)

属于稳定 ABI 自 *3.10* 版起. 获取 Python 线程状态 *tstate* 的当前帧。

返回一个*strong reference*。如果没有当前执行的帧则返回 NULL。

另请参阅*PyEval_GetFrame()*。

tstate 必须不为 NULL。

Added in version 3.9.

uint64_t **PyThreadState_GetID**(*PyThreadState* *tstate)

属于稳定 ABI 自 *3.10* 版起. 获取 Python 线程状态 *tstate* 的唯一线程状态标识符。

tstate 必须不为 NULL。

Added in version 3.9.

PyInterpreterState ***PyThreadState_GetInterpreter**(*PyThreadState* *tstate)

属于稳定 ABI 自 *3.10* 版起. 获取 Python 线程状态 *tstate* 对应的解释器。

tstate 必须不为 NULL。

Added in version 3.9.

void **PyThreadState_EnterTracing**(*PyThreadState* *tstate)

暂停 Python 线程状态 *tstate* 中的追踪和性能分析。

使用*PyThreadState_LeaveTracing()* 函数来恢复它们。

Added in version 3.11.

void **PyThreadState_LeaveTracing**(*PyThreadState* *tstate)

恢复 Python 线程状态 *tstate* 中被*PyThreadState_EnterTracing()* 函数暂停的追踪和性能分析。

另请参阅*PyEval_SetTrace()* 和*PyEval_SetProfile()* 函数。

Added in version 3.11.

PyInterpreterState ***PyInterpreterState_Get**(void)

属于稳定 ABI 自 *3.9* 版起. 获取当前解释器。

如果不存在当前 Python 线程状态或不存在当前解释器则将发出致命级错误信号。它无法返回 NULL。

呼叫者必须持有 GIL。

Added in version 3.9.

int64_t **PyInterpreterState_GetID**(*PyInterpreterState* *interp)

属于稳定 ABI 自 *3.7* 版起. 返回解释器的唯一 ID。如果执行过程中发生任何错误则将返回 -1 并设置错误。

呼叫者必须持有 GIL。

Added in version 3.7.

PyObject ***PyInterpreterState_GetDict**(*PyInterpreterState* *interp)

属于稳定 ABI 自 *3.8* 版起. 返回一个存储解释器专属数据的字典。如果此函数返回 NULL 则没有任何异常被引发并且调用方应当将解释器专属字典视为不可用。

这不是*PyModule_GetState()* 的替代，扩展仍应使用它来存储解释器专属的状态信息。

Added in version 3.8.

PyObject ***PyUnstable_InterpreterState_GetMainModule**(*PyInterpreterState* *interp)

这是不稳定 *API*。它可在次发布版中不经警告地改变。

为给定的解释器返回一个指向 `__main__` 模块对象 的*strong reference*。

呼叫者必须持有 GIL。

Added in version 3.13.

typedef *PyObject* *(***_PyFrameEvalFunction**)(*PyThreadState* *tstate, *_PyInterpreterFrame* *frame, int throwflag)

帧评估函数的类型

throwflag 形参将由生成器的 `throw()` 方法来使用：如为非零值，则处理当前异常。

在 3.9 版本发生变更: 此函数现在可接受一个 *tstate* 形参。

在 3.11 版本发生变更: *frame* 形参由 `PyFrameObject*` 改为 `_PyInterpreterFrame*`。

_PyFrameEvalFunction **_PyInterpreterState_GetEvalFrameFunc**(*PyInterpreterState* *interp)

获取帧评估函数。

请参阅 **PEP 523** "Adding a frame evaluation API to CPython"。

Added in version 3.9.

void **_PyInterpreterState_SetEvalFrameFunc**(*PyInterpreterState* *interp, *_PyFrameEvalFunction* eval_frame)

设置帧评估函数。

请参阅 **PEP 523** "Adding a frame evaluation API to CPython"。

Added in version 3.9.

PyObject ***PyThreadState_GetDict**()

返回值：借入的引用。属于稳定 ABI. 返回一个扩展可以在其中存储线程专属状态信息的字典。每个扩展都应当使用一个独有的键用来在该字典中存储状态。在没有可用的当前线程状态时也可以调用此函数。如果此函数返回 `NULL`，则还没有任何异常被引发并且调用方应当假定没有可用的当前线程状态。

int **PyThreadState_SetAsyncExc**(unsigned long id, *PyObject* *exc)

属于稳定 ABI. 在一个线程中异步地引发异常。*id* 参数是目标线程的线程 id；*exc* 是要引发的异常对象。该函数不会窃取任何对 *exc* 的引用。为防止随意滥用，你必须编写你自己的 C 扩展来调用它。调用时必须持有 GIL。返回已修改的线程状态数量；该值通常为一，但如果未找到线程 id 则会返回 0。如果 *exc* 为 "NULL"，则会清除线程的待处理异常（如果存在）。这将不会引发异常。

在 3.7 版本发生变更: *id* 形参的类型已从 `long` 变为 `unsigned long`。

void **PyEval_AcquireThread**(*PyThreadState* *tstate)

属于稳定 ABI. 获取全局解释器锁并将当前线程状态设为 *tstate*，它必须不为 `NULL`。锁必须在此之前已被创建。如果该线程已获取锁，则会发生死锁。

备注

当运行时正在最终化时从某个线程调用此函数将终结该线程，即使线程不是由 Python 创建的。你可以在调用此函数之前使用 *Py_IsFinalizing()* 或 `sys.is_finalizing()` 来检查解释器是否还处于最终化过程中以避免不必要的终结操作。

在 3.8 版本发生变更: 已被更新为与*PyEval_RestoreThread()*, *Py_END_ALLOW_THREADS()* 和*PyGILState_Ensure()* 保持一致，如果在解释器正在最终化时被调用则会终结当前线程。

PyEval_RestoreThread() 是一个始终可用的（即使线程尚未初始化）更高层级函数。

void **PyEval_ReleaseThread**(*PyThreadState* *tstate)

属于稳定 ABI. 将当前线程状态重置为 NULL 并释放全局解释器锁。在此之前锁必须已被创建并且必须由当前的线程所持有。*tstate* 参数必须不为 NULL，该参数仅被用于检查它是否代表当前线程状态 --- 如果不是，则会报告一个致命级错误。

PyEval_SaveThread() 是一个始终可用的（即使线程尚未初始化）更高层级函数。

9.6 子解释器支持

虽然在大多数用例中，你都只会嵌入一个单独的 Python 解释器，但某些场景需要你在同一个进程甚至同一个线程中创建多个独立的解释器。子解释器让你能够做到这一点。

“主”解释器是在运行时初始化时创建的第一个解释器。它通常是一个进程中唯一的 Python 解释器。与子解释器不同，主解释器具有唯一的进程全局责任比如信号处理等。它还负责在运行时初始化期间的执行并且通常还是运行时最终化期间的活动解释器。*PyInterpreterState_Main()* 函数将返回一个指向其状态的指针。

你可以使用*PyThreadState_Swap()* 函数在子解释器之间进行切换。你可以使用下列函数来创建和销毁它们:

type **PyInterpreterConfig**

包含用于配置子解释器的大部分形参的结构体。其值仅在*Py_NewInterpreterFromConfig()* 中被使用而绝不会被运行时所修改。

Added in version 3.12.

结构体字段:

int **use_main_obmalloc**

如果该值为 0 则子解释器将使用自己的“对象”分配器状态。否则它将使用（共享）主解释器的状态。

如果该值为 0 则*check_multi_interp_extensions* 必须为 1 (非零值)。如果该值为 1 则*gil* 不可为*PyInterpreterConfig_OWN_GIL*。

int **allow_fork**

如果该值为 0 则运行时将不支持在当前激活了子解释器的任何线程中 fork 进程。否则 fork 将不受限制。

请注意当 fork 被禁止时 subprocess 模块将仍然可用。

int **allow_exec**

如果该值为 0 则运行时将不支持在当前激活了子解释器的任何线程中通过 exec (例如 os.execv()) 替换当前进程。否则 exec 将不受限制。

请注意当 exec 被禁止时 subprocess 模块将仍然可用。

int **allow_threads**

如果该值为 0 则子解释器的 threading 模块将不会创建线程。否则线程将被允许。

int **allow_daemon_threads**

如果该值为 0 则子解释器的 threading 模块将不会创建守护线程。否则将允许守护线程（只要*allow_threads* 是非零值）。

int **check_multi_interp_extensions**

如果该值为 0 则所有扩展模块均可在当前子解释器被激活的任何线程中被导入，包括旧式的 (单阶段初始化) 模块。否则将只有多阶段初始化扩展模块 (参见 **PEP 489**) 可以被导入。(另请参阅*Py_mod_multiple_interpreters*。)

如果*use_main_obmalloc* 为 0 则该值必须为 1 (非零值)。

int **gil**

这将确定针对子解释器的 GIL 操作方式。它可以是以下的几种之一：

PyInterpreterConfig_DEFAULT_GIL

使用默认选择 (*PyInterpreterConfig_SHARED_GIL*)。

PyInterpreterConfig_SHARED_GIL

使用（共享）主解释器的 GIL。

PyInterpreterConfig_OWN_GIL

使用子解释器自己的 GIL。

如果该值为*PyInterpreterConfig_OWN_GIL* 则*PyInterpreterConfig.use_main_obmalloc* 必须为 0。

PyStatus **Py_NewInterpreterFromConfig**(*PyThreadState* **tstate_p, const *PyInterpreterConfig* *config)

新建一个子解释器。这是一个 (几乎) 完全隔离的 Python 代码执行环境。特别需要注意，新的子解释器具有全部已导入模块的隔离的、独立的版本，包括基本模块 `builtins`, `__main__` 和 `sys` 等。已加载模块表 (`sys.modules`) 和模块搜索路径 (`sys.path`) 也是隔离的。新环境没有 `sys.argv` 变量。它具有新的标准 I/O 流文件对象 `sys.stdin`, `sys.stdout` 和 `sys.stderr` (不过这些对象都指向相同的底层文件描述符)。

给定的 *config* 控制着初始化解释器所使用的选项。

成功后，*tstate_p* 将被设为新的子解释器中创建的第一个线程状态。该线程状态是在当前线程状态中创建的。请注意并没有真实的线程被创建；请参阅下文有关线程状态的讨论。如果创建新的解释器没有成功，则 *tstate_p* 将被设为 `NULL`；不会设置任何异常因为异常状态是存储在当前的线程状态中而当前线程状态并不一定存在。

与所有其他 Python/C API 函数一样，在调用此函数之前必须先持有全局解释器锁并且在其返回时仍继续持有。同样地在进入函数时也必须设置当前线程状态。执行成功后，返回的线程状态将被设为当前线程状态。如果创建的子解释器具有自己的 GIL 那么调用方解释器的 GIL 将被释放。当此函数返回时，新的解释器的 GIL 将由当前线程持有而之前的解释器的 GIL 在此将保持释放状态。

Added in version 3.12.

子解释器在彼此相互隔离，并让特定功能受限的情况下是最有效率的:

```
PyInterpreterConfig config = {
    .use_main_obmalloc = 0,
    .allow_fork = 0,
    .allow_exec = 0,
    .allow_threads = 1,
    .allow_daemon_threads = 0,
    .check_multi_interp_extensions = 1,
    .gil = PyInterpreterConfig_OWN_GIL,
};
PyThreadState *tstate = NULL;
PyStatus status = Py_NewInterpreterFromConfig(&tstate, &config);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}
```

请注意该配置只会被短暂使用而不会被修改。在初始化期间配置的值会被转换成各种*PyInterpreterState* 值。配置的只读副本可以被内部存储于*PyInterpreterState* 中。

扩展模块将以如下方式在（子）解释器之间共享:

- 对于使用多阶段初始化的模块，例如*PyModule_FromDefAndSpec()*，将为每个解释器创建并初始化一个单独的模块对象。只有 C 层级的静态和全局变量能在这些模块对象之间共享。
- 对于使用单阶段初始化的模块，例如*PyModule_Create()*，当特定扩展被首次导入时，它将被正常初始化，并会保存其模块字典的一个 (浅) 拷贝。当同一扩展被另一个 (子) 解释器导入时，将初始化一个新模块并填充该拷贝的内容；扩展的 `init` 函数不会被调用。因此模块字

典中的对象最终会被 (子) 解释器所共享，这可能会导致预期之外的行为 (参见下文的*Bugs and caveats*)。

请注意这不同于在调用*Py_FinalizeEx()* 和*Py_Initialize()* 完全重新初始化解释器之后导入扩展时所发生的情况；对于那种情况，扩展的 initmodule 函数 会被再次调用。与多阶段初始化一样，这意味着只有 C 层级的静态和全局变量能在这些模块之间共享。

PyThreadState ***Py_NewInterpreter**(void)

属于稳定 ABI. 新建一个子解释器。这在本质上只是针对*Py_NewInterpreterFromConfig()* 的包装器，其配置保留了现有的行为。结果是一个未隔离的子解释器，它会共享主解释器的 GIL，允许 fork/exec，允许守护线程，也允许单阶段初始化模块。

void **Py_EndInterpreter**(*PyThreadState* *tstate)

属于稳定 ABI. 销毁由给定的线程状态所代表的（子）解释器。给定的线程状态必须为当前的线程状态。请参阅下文中关于线程状态的讨论。当调用返回时，当前的线程状态将为 NULL。与此解释器相关联的所有线程状态都会被销毁。在调用此函数之前必须持有目标解释器所使用的全局解释器锁。当其返回时将不再持有 GIL。

Py_FinalizeEx() 将销毁所有在当前时间点上尚未被明确销毁的子解释器。

9.6.1 解释器级 GIL

使用*Py_NewInterpreterFromConfig()* 你将可以创建一个与其他解释器完全隔离的子解释器，包括具有自己的 GIL。这种隔离带来的最大好处在于这样的解释器执行 Python 代码时不会被其他解释器所阻塞或者阻塞任何其他解释器。因此在运行 Python 代码时单个 Python 进程可以真正地利用多个 CPU 核心。这种隔离还能鼓励开发者采取不同于仅使用线程的并发方式。(参见 **PEP 554**)。

使用隔离的解释器要求谨慎地保持隔离状态。尤其是意味着不要在未确保线程安全的情况下共享任何对象或可变的状态。由于引用计数的存在即使是在其他情况下不可变的对象 (例如 None, (1, 5)) 通常也不可被共享。针对此问题的一种简单但效率较低的解决方式是在使用某些状态 (或对象) 时总是使用一个全局锁。或者，对象实际上不可变的对象 (如整数或字符串) 可以通过将其设为*immortal* 对象而无视其引用计数来确保其安全性。事实上，对于内置单例、小整数和其他一些内置对象都是这样做的。

如果你能保持隔离状态那么你将能获得真正的多核计算能力而不会遇到自由线程所带来的复杂性。如果未能保持隔离状态那么你将面对自由线程所带来的全部后果，包括线程竞争和难以调试的崩溃。

除此之外，使用多个相互隔离的解释器的一个主要挑战是如何在它们之间安全 (不破坏隔离状态)、高效地进行通信。运行时和标准库还没有为此提供任何标准方式。未来的标准库模块将会帮助减少保持隔离状态所需的工作量并为解释器之间的数据通信（和共享）公开有效的工具。

Added in version 3.12.

9.6.2 错误和警告

由于子解释器 (以及主解释器) 都是同一个进程的组成部分，它们之间的隔离状态并非完美 --- 举例来说，使用低层级的文件操作如 os.close() 时它们可能 (无意或恶意地) 影响它们各自打开的文件。由于 (子) 解释器之间共享扩展的方式，某些扩展可能无法正常工作；在使用单阶段初始化或者 (静态) 全局变量时尤其如此。在一个子解释器中创建的对象有可能被插入到另一个 (子) 解释器的命名空间中；这种情况应当尽可能地避免。

应当特别注意避免在子解释器之间共享用户自定义的函数、方法、实例或类，因为由这些对象执行的导入操作可能会影响错误的已加载模块的 (子) 解释器的字典。同样重要的一点是应当避免共享可被上述对象访问的对象。

还要注意的一点是将此功能与 PyGILState_* API 结合使用是很微妙的，因为这些 API 会假定 Python 线程状态与操作系统级线程之间存在双向投影关系，而子解释器的存在打破了这一假定。强烈建议你不要在一对互相匹配的*PyGILState_Ensure()* 和*PyGILState_Release()* 调用之间切换子解释器。此外，使用这些 API 以允许从非 Python 创建的线程调用 Python 代码的扩展 (如 ctypes) 在使用子解释器时很可能会出现问题。

9.7 异步通知

提供了一种向主解释器线程发送异步通知的机制。这些通知将采用函数指针和空指针参数的形式。

int **Py_AddPendingCall**(int (*func)(void*), void *arg)

属于稳定 ABI. 将一个函数加入从主解释器线程调用的计划任务。成功时，将返回 `0` 并将 *func* 加入要被主线程调用的等待队列。失败时，将返回 `-1` 但不会设置任何异常。

当成功加入队列后，*func* 将 最终附带参数 *arg* 被主解释器线程调用。对于正常运行的 Python 代码来说它将被异步地调用，但要同时满足以下两个条件：

- 位于*bytecode* 的边界上；
- 主线程持有*global interpreter lock* (因此 *func* 可以使用完整的 C API)。

func 必须在成功时返回 `0`，或在失败时返回 `-1` 并设置一个异常集合。*func* 不会被中断来递归地执行另一个异步通知，但如果全局解释器锁被释放则它仍可被中断以切换线程。

此函数的运行不需要当前线程状态，也不需要全局解释器锁。

要在子解释器中调用函数，调用方必须持有 GIL。否则，函数 *func* 可能会被安排给错误的解释器来调用。

> **警告**
>
> 这是一个低层级函数，只在非常特殊的情况下有用。不能保证 *func* 会尽快被调用。如果主线程忙于执行某个系统调用，*func* 将不会在系统调用返回之前被调用。此函数通常 **不适合**从任意 C 线程调用 Python 代码。作为替代，请使用*PyGILStateAPI*。

Added in version 3.1.

在 3.9 版本发生变更: 如果此函数在子解释器中被调用，则函数 *func* 将被安排在子解释器中调用，而不是在主解释器中调用。现在每个子解释器都有自己的计划调用列表。

9.8 分析和跟踪

Python 解释器为附加的性能分析和执行跟踪工具提供了一些低层级的支持。它们可被用于性能分析、调试和覆盖分析工具。

这个 C 接口允许性能分析或跟踪代码避免调用 Python 层级的可调用对象带来的开销，它能直接执行 C 函数调用。此工具的基本属性没有变化；这个接口允许针对每个线程安装跟踪函数，并且向跟踪函数报告的基本事件与之前版本中向 Python 层级跟踪函数报告的事件相同。

typedef int (***Py_tracefunc**)(*PyObject* *obj, *PyFrameObject* *frame, int what, *PyObject* *arg)

使用*PyEval_SetProfile()* 和*PyEval_SetTrace()* 注册的跟踪函数的类型。第一个形参是作为 *obj* 传递给注册函数的对象，*frame* 是与事件相关的帧对象，*what* 是常量*PyTrace_CALL*, *PyTrace_EXCEPTION*, *PyTrace_LINE*, *PyTrace_RETURN*, *PyTrace_C_CALL*, *PyTrace_C_EXCEPTION*, *PyTrace_C_RETURN* 或*PyTrace_OPCODE* 中的一个，而 *arg* 将依赖于 *what* 的值：

what 的值	*arg* 的含义
PyTrace_CALL	总是*Py_None*.
PyTrace_EXCEPTION	`sys.exc_info()` 返回的异常信息。
PyTrace_LINE	总是*Py_None*.
PyTrace_RETURN	返回给调用方的值，或者如果是由异常导致的则返回 `NULL`。
PyTrace_C_CALL	正在调用函数对象。
PyTrace_C_EXCEPTION	正在调用函数对象。
PyTrace_C_RETURN	正在调用函数对象。
PyTrace_OPCODE	总是*Py_None*.

int **PyTrace_CALL**

当对一个函数或方法的新调用被报告，或是向一个生成器增加新条目时传给*Py_tracefunc* 函数的 *what* 形参的值。请注意针对生成器函数的迭代器的创建情况不会被报告因为在相应的帧中没有向 Python 字节码转移控制权。

int **PyTrace_EXCEPTION**

当一个异常被引发时传给*Py_tracefunc* 函数的 *what* 形参的值。在处理完任何字节码之后将附带 *what* 的值调用回调函数，在此之后该异常将会被设置在正在执行的帧中。这样做的效果是当异常传播导致 Python 栈展开时，被调用的回调函数将随异常传播返回到每个帧。只有跟踪函数才会接收到这些事件；性能分析器并不需要它们。

int **PyTrace_LINE**

当一个行编号事件被报告时传给*Py_tracefunc* 函数 (但不会传给性能分析函数) 的 *what* 形参的值。它可以通过将 `f_trace_lines` 设为 *0* 在某个帧中被禁用。

int **PyTrace_RETURN**

当一个调用即将返回时传给*Py_tracefunc* 函数的 *what* 形参的值。

int **PyTrace_C_CALL**

当一个 C 函数即将被调用时传给*Py_tracefunc* 函数的 *what* 形参的值。

int **PyTrace_C_EXCEPTION**

当一个 C 函数引发异常时传给*Py_tracefunc* 函数的 *what* 形参的值。

int **PyTrace_C_RETURN**

当一个 C 函数返回时传给*Py_tracefunc* 函数的 *what* 形参的值。

int **PyTrace_OPCODE**

当一个新操作码即将被执行时传给*Py_tracefunc* 函数 (但不会传给性能分析函数) 的 *what* 形参的值。在默认情况下此事件不会被发送：它必须通过在某个帧上将 `f_trace_opcodes` 设为 *1* 来显式地请求。

void **PyEval_SetProfile**(*Py_tracefunc* func, *PyObject* *obj)

将性能分析器函数设为 *func*。*obj* 形参将作为第一个形参传给该函数，它可以是任意 Python 对象或为 `NULL`。如果性能分析函数需要维护状态，则为每个线程的 *obj* 使用不同的值将提供一个方便而线程安全的存储位置。这个性能分析函数将针对除*PyTrace_LINE* *PyTrace_OPCODE* 和*PyTrace_EXCEPTION* 以外的所有被监控事件进行调用。

另请参阅 `sys.setprofile()` 函数。

调用方必须持有*GIL*。

void **PyEval_SetProfileAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

类似于*PyEval_SetProfile()* 但会在属于当前解释器的所有在运行线程中设置性能分析函数而不是仅在当前线程上设置。

调用方必须持有*GIL*。

与*PyEval_SetProfile()* 一样，该函数会忽略任何被引发的异常同时在所有线程中设置性能分析函数。

Added in version 3.12.

void **PyEval_SetTrace**(*Py_tracefunc* func, *PyObject* *obj)

将跟踪函数设为 *func*。这类似于*PyEval_SetProfile()*，区别在于跟踪函数会接收行编号事件和操作码级事件，但不会接收与被调用的 C 函数对象相关的任何事件。使用*PyEval_SetTrace()* 注册的任何跟踪函数将不会接收*PyTrace_C_CALL*、*PyTrace_C_EXCEPTION* 或*PyTrace_C_RETURN* 作为 *what* 形参的值。

另请参阅 `sys.settrace()` 函数。

调用方必须持有*GIL*。

void **PyEval_SetTraceAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

类似于*PyEval_SetTrace()* 但会在属于当前解释器的所有在运行线程中设置跟踪函数而不是仅在当前线程上设置。

调用方必须持有*GIL*。

与*PyEval_SetTrace()* 一样，该函数会忽略任何被引发的异常同时在所有线程中设置跟踪函数。

Added in version 3.12.

9.9 引用追踪

Added in version 3.13.

typedef int (***PyRefTracer**)(*PyObject**, int event, void *data)

使用*PyRefTracer_SetTracer()* 注册的追踪函数的类型。第一个形参是刚创建（当 **event** 被设为*PyRefTracer_CREATE* 时）或将销毁（当 **event** 被设为*PyRefTracer_DESTROY* 时）的 Python 对象。**data** 参数是当*PyRefTracer_SetTracer()* 被调用时所提供的不透明指针。

Added in version 3.13.

int **PyRefTracer_CREATE**

当一个 Python 对象被创建时传给*PyRefTracer* 函数的 *event* 形参。

int **PyRefTracer_DESTROY**

当一个 Python 对象被销毁时传给*PyRefTracer* 函数的 *event* 形参。

int **PyRefTracer_SetTracer**(*PyRefTracer* tracer, void *data)

注册一个引用追踪函数。该函数将在新的 Python 对象被创建或对象被销毁时被调用。如果提供了 **data** 则它必须是一个当追踪函数被调用时所提供的不透明指针。成功时返回 `0`。发生错误时将设置一个异常并返回 `-1`。

请注意该追踪函数 **不可**在其内部创建 Python 对象否则调用将被重入。该追踪器也 **不可**清除任何现有异常或者设置异常。每次当追踪器被调用时都将持有 GIL。

当调用此函数时必须持有 GIL。

Added in version 3.13.

PyRefTracer **PyRefTracer_GetTracer**(void **data)

获取已注册的引用追踪函数以及当*PyRefTracer_SetTracer()* 被调用时所注册的不透明数据指针的值。如果未注册任何追踪器则此函数将返回 NULL 并将 **data** 指针设为 NULL。

当调用此函数时必须持有 GIL。

Added in version 3.13.

9.10 高级调试器支持

这些函数仅供高级调试工具使用。

PyInterpreterState ***PyInterpreterState_Head**()

将解释器状态对象返回到由所有此类对象组成的列表的开头。

PyInterpreterState ***PyInterpreterState_Main**()

返回主解释器状态对象。

PyInterpreterState ***PyInterpreterState_Next**(*PyInterpreterState* *interp)

从由解释器状态对象组成的列表中返回 *interp* 之后的下一项。

PyThreadState ***PyInterpreterState_ThreadHead**(*PyInterpreterState* *interp)

在由与解释器 *interp* 相关联的线程组成的列表中返回指向第一个*PyThreadState* 对象的指针。

PyThreadState ***PyThreadState_Next**(*PyThreadState* *tstate)

从由属于同一个*PyInterpreterState* 对象的线程状态对象组成的列表中返回 *tstate* 之后的下一项。

9.11 线程本地存储支持

Python 解释器提供也对线程本地存储 (TLS) 的低层级支持，它对下层的原生 TLS 实现进行了包装以支持 Python 层级的线程本地存储 API (`threading.local`)。CPython 的 C 层级 API 与 pthreads 和 Windows 所提供的类似：使用一个线程键和函数来为每个线程关联一个 `void*` 值。

当调用这些函数时 *无须* 持有 GIL；它们会提供自己的锁机制。

请注意 `Python.h` 并不包括 TLS API 的声明，你需要包括 `pythread.h` 来使用线程本地存储。

备注

这些 API 函数都不会为 `void*` 的值处理内存管理问题。你需要自己分配和释放它们。如果 `void*` 值碰巧为 *PyObject**，这些函数也不会对它们执行引用计数操作。

9.11.1 线程专属存储 (TSS) API

引入 TSSAPI 是为了取代 CPython 解释器中现有 TLS API 的使用。该 API 使用一个新类型*Py_tss_t* 而不是 `int` 来表示线程键。

Added in version 3.7.

参见

"A New C-API for Thread-Local Storage in CPython" (**PEP 539**)

type **Py_tss_t**

该数据结构表示线程键的状态，其定义可能依赖于下层的 TLS 实现，并且它有一个表示键初始化状态的内部字段。该结构体中不存在公有成员。

当未定义*Py_LIMITED_API* 时，允许由*Py_tss_NEEDS_INIT* 执行此类型的静态分配。

Py_tss_NEEDS_INIT

这个宏将扩展为*Py_tss_t* 变量的初始化器。请注意这个宏不会用*Py_LIMITED_API* 来定义。

动态分配

Py_tss_t 的动态分配，在使用*Py_LIMITED_API* 编译的扩展模块中是必须的，在这些模块由于此类型的实现在编译时是不透明的因此它不可能静态分配。

Py_tss_t ***PyThread_tss_alloc**()

属于稳定 ABI 自 *3.7* 版起. 返回一个与使用*Py_tss_NEEDS_INIT* 初始化的值的状态相同的值，或者当动态分配失败时则返回 `NULL`。

void **PyThread_tss_free**(*Py_tss_t* *key)

属于稳定 ABI 自 *3.7* 版起. 在首次调用*PyThread_tss_delete()* 以确保任何相关联的线程局部变量已被撤销赋值之后释放由*PyThread_tss_alloc()* 所分配的给定的 *key*。如果 *key* 参数为 `NULL` 则这将无任何操作。

备注

被释放的 key 将变成一个悬空指针。你应当将 key 重置为 `NULL`。

方法

这些函数的形参 *key* 不可为 NULL。并且，如果给定的*Py_tss_t* 还未被*PyThread_tss_create()* 初始化则*PyThread_tss_set()* 和*PyThread_tss_get()* 的行为将是未定义的。

int **PyThread_tss_is_created**(*Py_tss_t* *key)

属于稳定 ABI 自 *3.7* 版起. 如果给定的*Py_tss_t* 已通过 has been initialized by *PyThread_tss_create()* 被初始化则返回一个非零值。

int **PyThread_tss_create**(*Py_tss_t* *key)

属于稳定 ABI 自 *3.7* 版起. 当成功初始化一个 TSS 键时将返回零值。如果 *key* 参数所指向的值未被*Py_tss_NEEDS_INIT* 初始化则其行为是未定义的。此函数可在相同的键上重复调用 -- 在已初始化的键上调用它将不执行任何操作并立即成功返回。

void **PyThread_tss_delete**(*Py_tss_t* *key)

属于稳定 ABI 自 *3.7* 版起. 销毁一个 TSS 键以便在所有线程中遗忘与该键相关联的值，并将该键的初始化状态改为未初始化的。已销毁的键可以通过*PyThread_tss_create()* 再次被初始化。此函数可以在同一个键上重复调用 -- 但在一个已被销毁的键上调用将是无效的。

int **PyThread_tss_set**(*Py_tss_t* *key, void *value)

属于稳定 ABI 自 *3.7* 版起. 返回零值来表示成功将一个 void* 值与当前线程中的 TSS 键相关联。每个线程都有一个从键到 void* 值的独立映射。

void ***PyThread_tss_get**(*Py_tss_t* *key)

属于稳定 ABI 自 *3.7* 版起. 返回当前线程中与一个 TSS 键相关联的 void* 值。如果当前线程中没有与该键相关联的值则返回 NULL。

9.11.2 线程本地存储 (TLS) API

自 3.7 版本弃用: 此 API 已被*线程专属存储 (TSS) API* 所取代。

备注

这个 API 版本不支持原生 TLS 键采用无法被安全转换为 int 的的定义方式的平台。在这样的平台上，*PyThread_create_key()* 将立即返回一个失败状态，并且其他 TLS 函数在这样的平台上也都无效。

由于上面提到的兼容性问题，不应在新代码中使用此版本的 API。

int **PyThread_create_key**()

属于稳定 ABI.

void **PyThread_delete_key**(int key)

属于稳定 ABI.

int **PyThread_set_key_value**(int key, void *value)

属于稳定 ABI.

void ***PyThread_get_key_value**(int key)

属于稳定 ABI.

void **PyThread_delete_key_value**(int key)

属于稳定 ABI.

void **PyThread_ReInitTLS**()

属于稳定 ABI.

9.12 同步原语

C-API 提供了一个基本的互斥锁。

type **PyMutex**

一个互斥锁。PyMutex 应当被初始化为零以代表未加锁状态。例如:

```
PyMutex mutex = {0};
```

PyMutex 的实例不应被拷贝或移动。PyMutex 的内容和地址都是有意义的，它必须在内存中保持一个固定的、可写的位置。

> **备注**
>
> PyMutex 目前占用一个字节，但这个大小应当被视为是不稳定的。这个大小可能在未来的 Python 发布版中发生改变而不会设置弃用期。

Added in version 3.13.

void **PyMutex_Lock**(*PyMutex* *m)

锁定互斥锁 *m*。如果另一个线程已经锁定了它，调用方线程将阻塞直至互斥锁被解锁。在阻塞期间，如果线程持有*GIL* 则会临时释放它。

Added in version 3.13.

void **PyMutex_Unlock**(*PyMutex* *m)

解锁互斥锁 *m*。该互斥锁必须已被锁定 --- 否则，此函数将发生致命错误。

Added in version 3.13.

9.12.1 Python 关键节 API

此关键节 API 为*自由线程* CPython 的每对象锁之上提供了一个死锁避免层。它们旨在替代对*global interpreter lock* 的依赖，而在具有全局解释器锁的 Python 版本上将不做任何操作。

关键节机制通过在调用*PyEval_SaveThread()* 期间隐式地挂起活动的关键节并释放锁来避免死锁。当*PyEval_RestoreThread()* 被调用时，最近的关键节将被恢复，并重新获取它的锁。这意味着关键节 API 提供了与传统锁相比更弱的保证 -- 它们有用是因为它们的行为与*GIL* 类似。

宏所使用的函数和结构体是针对 C 宏不可用的场景而公开的。它们应当仅被用于给定的宏扩展中。请注意这些结构体的大小和内容在未来的 Python 版本中可能发生改变。

> **备注**
>
> 需要同时锁定两个对象的操作必须使用*Py_BEGIN_CRITICAL_SECTION2*。你 不可使用嵌套的关键节来同时锁定一个以上的对象，因为内层的关键节可能会挂起外层的关键节。这个 API 没有提供同时锁定两个以上对象的办法。

用法示例:

```
static PyObject *
set_field(MyObject *self, PyObject *value)
{
   Py_BEGIN_CRITICAL_SECTION(self);
   Py_SETREF(self->field, Py_XNewRef(value));
   Py_END_CRITICAL_SECTION();
   Py_RETURN_NONE;
}
```

在上面的例子中，*Py_SETREF* 调用了*Py_DECREF*，它可以通过一个对象的取消分配函数来调用任意代码。当由最终化器触发的代码发生阻塞并调用*PyEval_SaveThread()* 时关键节 API 将通过允许运行临时挂起关键节来避免由于重入和锁顺序导致的潜在死锁。

Py_BEGIN_CRITICAL_SECTION(op)

为对象 *op* 获取每对象锁并开始一个关键节。

在自由线程构建版中，该宏将扩展为:

```
{
    PyCriticalSection _py_cs;
    PyCriticalSection_Begin(&_py_cs, (PyObject*)(op))
```

在默认构建版中，该宏将扩展为 `{`。

Added in version 3.13.

Py_END_CRITICAL_SECTION()

结束关键节并释放每对象锁。

在自由线程构建版中，该宏将扩展为:

```
    PyCriticalSection_End(&_py_cs);
}
```

在默认构建版中，该宏将扩展为 `}`。

Added in version 3.13.

Py_BEGIN_CRITICAL_SECTION2(a, b)

为对象 *a* 和 *b* 获取每对象锁并开始一个关键节。这些锁是按连续顺序获取的（最低的地址在最前）以避免锁顺序列死锁。

在自由线程构建版中，该宏将扩展为:

```
{
    PyCriticalSection2 _py_cs2;
    PyCriticalSection2_Begin(&_py_cs2, (PyObject*)(a), (PyObject*)(b))
```

在默认构建版中，该宏将扩展为 `{`。

Added in version 3.13.

Py_END_CRITICAL_SECTION2()

结束关键节并释放每对象锁。

在自由线程构建版中，该宏将扩展为:

```
    PyCriticalSection2_End(&_py_cs2);
}
```

在默认构建版中，该宏将扩展为 `}`。

Added in version 3.13.

CHAPTER 10

Python 初始化配置

Added in version 3.8.

Python 可以使用*Py_InitializeFromConfig()* 和*PyConfig* 结构体来初始化。它可以使用*Py_PreInitialize()* 和*PyPreConfig* 结构体来预初始化。

有两种配置方式：

- *Python 配置* 可被用于构建一个定制的 Python，其行为与常规 Python 类似。例如，环境变量和命令行参数可被用于配置 Python。
- *隔离配置* 可被用于将 Python 嵌入到应用程序。它将 Python 与系统隔离开来。例如，环境变量将被忽略，LC_CTYPE 语言区域设置保持不变并且不会注册任何信号处理器。

Py_RunMain() 函数可被用来编写定制的 Python 程序。

参见*Initialization, Finalization, and Threads*.

参见

PEP 587 ”Python 初始化配置”.

10.1 示例

定制的 Python 的示例总是会以隔离模式运行:

```
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

    /* 解码命令行参数。
       隐式地预初始化 Python (隔离模式)。 */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto exception;
```

(续下页)

(接上页)

```
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();

exception:
    PyConfig_Clear(&config);
    if (PyStatus_IsExit(status)) {
        return status.exitcode;
    }
    /* 显示错误消息然后退出进程
       并设置非零值退出码 */
    Py_ExitStatusException(status);
}
```

10.2 PyWideStringList

type **PyWideStringList**

由 `wchar_t*` 字符串组成的列表。

如果 *length* 为非零值，则 *items* 必须不为 `NULL` 并且所有字符串均必须不为 `NULL`。

方法

PyStatus **PyWideStringList_Append**(*PyWideStringList* *list, const wchar_t *item)

将 *item* 添加到 *list*。

Python 必须被预初始化以便调用此函数。

PyStatus **PyWideStringList_Insert**(*PyWideStringList* *list, *Py_ssize_t* index, const wchar_t *item)

将 *item* 插入到 *list* 的 *index* 位置上。

如果 *index* 大于等于 *list* 的长度，则将 *item* 添加到 *list*。

index 必须大于等于 `0`。

Python 必须被预初始化以便调用此函数。

结构体字段:

Py_ssize_t **length**

List 长度。

wchar_t ***items**

列表项目。

10.3 PyStatus

type **PyStatus**

存储初始函数状态：成功、错误或退出的结构体。

对于错误，它可以存储造成错误的 C 函数的名称。

结构体字段:

int **exitcode**
: 退出码。传给 `exit()` 的参数。

const char ***err_msg**
: 错误信息

const char ***func**
: 造成错误的函数的名称，可以为 `NULL`。

创建状态的函数:

PyStatus **PyStatus_Ok**(void)
: 完成。

PyStatus **PyStatus_Error**(const char *err_msg)
: 带消息的初始化错误。

err_msg 不可为 `NULL`。

PyStatus **PyStatus_NoMemory**(void)
: 内存分配失败（内存不足）。

PyStatus **PyStatus_Exit**(int exitcode)
: 以指定的退出代码退出 Python。

处理状态的函数:

int **PyStatus_Exception**(*PyStatus* status)
: 状态为错误还是退出？如为真值，则异常必须被处理；例如通过调用*Py_ExitStatusException()*。

int **PyStatus_IsError**(*PyStatus* status)
: 结果错误吗?

int **PyStatus_IsExit**(*PyStatus* status)
: 结果是否退出?

void **Py_ExitStatusException**(*PyStatus* status)
: 如果 *status* 是一个退出码则调用 `exit(exitcode)`。如果 *status* 是一个错误码则打印错误消息并设置一个非零退出码再退出。必须在 `PyStatus_Exception(status)` 为非零值时才能被调用。

备注

在内部，Python 将使用设置 `PyStatus.func` 的宏，而创建状态的函数则会将 `func` 设为 `NULL`。

示例:

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
    return PyStatus_Ok();
}

int main(int argc, char **argv)
{
    void *ptr;
    PyStatus status = alloc(&ptr, 16);
```

(续下页)

(接上页)

```
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
    PyMem_Free(ptr);
    return 0;
}
```

10.4 PyPreConfig

type **PyPreConfig**

用于预初始化 Python 的结构体。

用于初始化预先配置的函数:

void **PyPreConfig_InitPythonConfig**(*PyPreConfig* *preconfig)

通过*Python 配置* 来初始化预先配置。

void **PyPreConfig_InitIsolatedConfig**(*PyPreConfig* *preconfig)

通过隔离配置 来初始化预先配置。

结构体字段:

int **allocator**

Python 内存分配器名称:

- PYMEM_ALLOCATOR_NOT_SET (0): 不改变内存分配器 (使用默认)。
- PYMEM_ALLOCATOR_DEFAULT (1): 默认内存分配器。
- PYMEM_ALLOCATOR_DEBUG (2): 默认内存分配器 附带调试钩子。
- PYMEM_ALLOCATOR_MALLOC (3): 使用 C 库的 malloc()。
- PYMEM_ALLOCATOR_MALLOC_DEBUG (4): 强制使用 malloc() 附带调试钩子。
- PYMEM_ALLOCATOR_PYMALLOC (5): *Python pymalloc* 内存分配器。
- PYMEM_ALLOCATOR_PYMALLOC_DEBUG (6): *Python pymalloc* 内存分配器 附带调试钩子。
- PYMEM_ALLOCATOR_MIMALLOC (6): 使用 mimalloc，一个快速的 malloc 替代。
- PYMEM_ALLOCATOR_MIMALLOC_DEBUG (7): 使用 mimalloc，一个快速的 malloc 替代，它带有调试钩子。

如果 Python 是使用 --without-pymalloc 进行配置则 PYMEM_ALLOCATOR_PYMALLOC 和 PYMEM_ALLOCATOR_PYMALLOC_DEBUG 将不被支持。

如果 Python 是使用 --without-mimalloc 进行配置或者如果下层的原子化支持不可用则 PYMEM_ALLOCATOR_MIMALLOC 和 PYMEM_ALLOCATOR_MIMALLOC_DEBUG 将不被支持。

参见*Memory Management*.

默认值: PYMEM_ALLOCATOR_NOT_SET。

int **configure_locale**

将 LC_CTYPE 语言区域设为用户选择的语言区域。

如果等于 0，则将*coerce_c_locale* 和*coerce_c_locale_warn* 的成员设为 0。

参见*locale encoding*。

默认值: 在 Python 配置中为 1，在隔离配置中为 0。

int **coerce_c_locale**

如果等于 2，强制转换 C 语言区域。

如果等于 1，则读取 LC_CTYPE 语言区域来确定其是否应当被强制转换。

参见*locale encoding*。

默认值: 在 Python 配置中为 -1，在隔离配置中为 0。

int **coerce_c_locale_warn**

如为非零值，则会在 C 语言区域被强制转换时发出警告。

默认值: 在 Python 配置中为 -1，在隔离配置中为 0。

int **dev_mode**

Python 开发模式: 参见*PyConfig.dev_mode*。

默认值: 在 Python 模式中为 -1，在隔离模式中为 0。

int **isolated**

隔离模式：参见*PyConfig.isolated*。

默认值: 在 Python 模式中为 0，在隔离模式中为 1。

int **legacy_windows_fs_encoding**

如为非零值:

- 设置*PyPreConfig.utf8_mode* 为 0,
- 设置*PyConfig.filesystem_encoding* 为 "mbcs",
- 设置*PyConfig.filesystem_errors* 为 "replace".

基于 PYTHONLEGACYWINDOWSFSENCODING 环境变量值完成初始化。

仅在 Windows 上可用。#ifdef MS_WINDOWS 宏可被用于 Windows 专属的代码。

默认值: 0.

int **parse_argv**

如为非零值，*Py_PreInitializeFromArgs()* 和*Py_PreInitializeFromBytesArgs()* 将以与常规 Python 解析命令行参数的相同方式解析其 argv 参数：参见 命令行参数。

默认值: 在 Python 配置中为 1，在隔离配置中为 0。

int **use_environment**

使用 环境变量? 参见*PyConfig.use_environment*。

默认值: 在 Python 配置中为 1 而在隔离配置中为 0。

int **utf8_mode**

如为非零值，则启用 Python UTF-8 模式。

通过 -X utf8 命令行选项和 PYTHONUTF8 环境变量设为 0 或 1。

如果 LC_CTYPE 语言区域为 C 或 POSIX 也会被设为 1。

默认值: 在 Python 配置中为 -1 而在隔离配置中为 0。

10.5 使用 PyPreConfig 预初始化 Python

Python 的预初始化:

- 设置 Python 内存分配器 (*PyPreConfig.allocator*)
- 配置 LC_CTYPE 语言区域 (*locale encoding*)
- 设置 Python UTF-8 模式 (*PyPreConfig.utf8_mode*)

当前的预配置 (PyPreConfig 类型) 保存在 _PyRuntime.preconfig 中。

用于预初始化 Python 的函数:

PyStatus **Py_PreInitialize**(const *PyPreConfig* *preconfig)

根据 *preconfig* 预配置来预初始化 Python。

preconfig 不可为 NULL。

PyStatus **Py_PreInitializeFromBytesArgs**(const *PyPreConfig* *preconfig, int argc, char *const *argv)

根据 *preconfig* 预配置来预初始化 Python。

如果 *preconfig* 的*parse_argv* 为非零值则解析 *argv* 命令行参数（字节串）。

preconfig 不可为 NULL。

PyStatus **Py_PreInitializeFromArgs**(const *PyPreConfig* *preconfig, int argc, wchar_t *const *argv)

根据 *preconfig* 预配置来预初始化 Python。

如果 *preconfig* 的*parse_argv* 为非零值则解析 *argv* 命令行参数（宽字符串）。

preconfig 不可为 NULL。

调用方要负责使用*PyStatus_Exception()* 和*Py_ExitStatusException()* 来处理异常（错误或退出）。

对于*Python 配置* (*PyPreConfig_InitPythonConfig()*)，如果 Python 是用命令行参数初始化的，那么在预初始化 Python 时也必须传递命令行参数，因为它们会对编码格式等预配置产生影响。例如，-X utf8 命令行选项将启用 Python UTF-8 模式。

PyMem_SetAllocator() 可在*Py_PreInitialize()* 之后、*Py_InitializeFromConfig()* 之前被调用以安装自定义的内存分配器。如果*PyPreConfig.allocator* 被设为 PYMEM_ALLOCATOR_NOT_SET 则可在*Py_PreInitialize()* 之前被调用。

像*PyMem_RawMalloc()* 这样的 Python 内存分配函数不能在 Python 预初始化之前使用，而直接调用 malloc() 和 free() 则始终会是安全的。*Py_DecodeLocale()* 不能在 Python 预初始化之前被调用。

使用预初始化来启用 Python UTF-8 模式的例子:

```
PyStatus status;
PyPreConfig preconfig;
PyPreConfig_InitPythonConfig(&preconfig);

preconfig.utf8_mode = 1;

status = Py_PreInitialize(&preconfig);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}

/* 此时，Python 将使用 UTF-8 */

Py_Initialize();
/* ... 在此使用 Python API ... */
Py_Finalize();
```

10.6 PyConfig

type **PyConfig**

包含了大部分用于配置 Python 的形参的结构体。

在完成后，必须使用*PyConfig_Clear()* 函数来释放配置内存。

结构体方法:

void **PyConfig_InitPythonConfig**(*PyConfig* *config)

通过*Python 配置* 来初始化配置。

void **PyConfig_InitIsolatedConfig**(*PyConfig* *config)

通过隔离配置 来初始化配置。

PyStatus **PyConfig_SetString**(*PyConfig* *config, wchar_t *const *config_str, const wchar_t *str)

将宽字符串 *str* 拷贝至 `*config_str`。

在必要时预初始化 *Python*。

PyStatus **PyConfig_SetBytesString**(*PyConfig* *config, wchar_t *const *config_str, const char *str)

使用*Py_DecodeLocale()* 对 *str* 进行解码并将结果设置到 `*config_str`。

在必要时预初始化 *Python*。

PyStatus **PyConfig_SetArgv**(*PyConfig* *config, int argc, wchar_t *const *argv)

根据宽字符串列表 *argv* 设置命令行参数 (*config* 的*argv* 成员)。

在必要时预初始化 *Python*。

PyStatus **PyConfig_SetBytesArgv**(*PyConfig* *config, int argc, char *const *argv)

根据字节串列表 *argv* 设置命令行参数 (*config* 的*argv* 成员)。使用*Py_DecodeLocale()* 对字节串进行解码。

在必要时预初始化 *Python*。

PyStatus **PyConfig_SetWideStringList**(*PyConfig* *config, *PyWideStringList* *list, *Py_ssize_t* length, wchar_t **items)

将宽字符串列表 *list* 设置为 *length* 和 *items*。

在必要时预初始化 *Python*。

PyStatus **PyConfig_Read**(*PyConfig* *config)

读取所有 Python 配置。

已经初始化的字段会保持不变。

调用此函数时不再计算或修改用于路径配置 的字段，如 Python 3.11 那样。

PyConfig_Read() 函数对*PyConfig.argv* 参数只会解析一次：在参数解析完成后*PyConfig.parse_argv* 将被设为 `2`。由于 Python 参数是从*PyConfig.argv* 提取的，因此解析参数两次会将应用程序选项解析为 Python 选项。

在必要时预初始化 *Python*。

在 3.10 版本发生变更: *PyConfig.argv* 参数现在只会被解析一次，在参数解析完成后，*PyConfig.parse_argv* 将被设为 `2`，只有当*PyConfig.parse_argv* 等于 `1` 时才会解析参数。

在 3.11 版本发生变更: *PyConfig_Read()* 不会再计算所有路径，因此在*Python 路径配置* 下列出的字段可能不会再更新直到*Py_InitializeFromConfig()* 被调用。

void **PyConfig_Clear**(*PyConfig* *config)

释放配置内存

如有必要大多数 `PyConfig` 方法将会预初始化 *Python*。在这种情况下，Python 预初始化配置 (*PyPreConfig*) 将以*PyConfig* 为基础。如果要调整与*PyPreConfig* 相同的配置字段，它们必须在调用*PyConfig* 方法之前被设置:

- *PyConfig.dev_mode*
- *PyConfig.isolated*
- *PyConfig.parse_argv*
- *PyConfig.use_environment*

此外，如果使用了*PyConfig_SetArgv()* 或*PyConfig_SetBytesArgv()*，则必须在调用其他方法之前调用该方法，因为预初始化配置取决于命令行参数（如果*parse_argv* 为非零值）。

这些方法的调用者要负责使用 PyStatus_Exception() 和 Py_ExitStatusException() 来处理异常（错误或退出）。

结构体字段:

PyWideStringList **argv**

根据*argv* 设置 sys.argv 命令行参数。这些形参与传给程序的 main() 函数的类似，区别在于其中第一项应当指向要执行的脚本文件而不是 Python 解释器对应的可执行文件。如果没有要运行的脚本，而*argv* 中的第一项可以为空字符串。

将*parse_argv* 设为 1 将以与普通 Python 解析 Python 命令行参数相同的方式解析*argv* 再从*argv* 中剥离 Python 参数。

如果*argv* 为空，则会添加一个空字符串以确保 sys.argv 始终存在并且永远不为空。

默认值: NULL.

另请参阅*orig_argv* 成员。

int **safe_path**

如果等于零，Py_RunMain() 会在启动时向 sys.path 开头添加一个可能不安全的路径:

- 如果*argv[0]* 等于 L"-m" (python -m module)，则添加当前工作目录。
- 如果是运行脚本 (python script.py)，则添加脚本的目录。如果是符号链接，则会解析符号链接。
- 在其他情况下 (python -c code 和 python)，将添加一个空字符串，这表示当前工作目录。

通过 -P 命令行选项和 PYTHONSAFEPATH 环境变量设置为 1。

默认值：Python 配置中为 0，隔离配置中为 1。

Added in version 3.11.

wchar_t ***base_exec_prefix**

sys.base_exec_prefix.

默认值: NULL.

Python 路径配置 的一部分。

另请参阅*PyConfig.exec_prefix*。

wchar_t ***base_executable**

Python 基础可执行文件: sys._base_executable。

由 __PYVENV_LAUNCHER__ 环境变量设置。

如为 NULL 则从*PyConfig.executable* 设置。

默认值: NULL.

Python 路径配置 的一部分。

另请参阅*PyConfig.executable*。

wchar_t ***base_prefix**

sys.base_prefix.

默认值: NULL.

Python 路径配置 的一部分。

另请参阅*PyConfig.prefix*。

int **buffered_stdio**

如果等于 0 且*configure_c_stdio* 为非零值，则禁用 C 数据流 stdout 和 stderr 的缓冲。

通过 -u 命令行选项和 PYTHONUNBUFFERED 环境变量设置为 0。

stdin 始终以缓冲模式打开。

默认值: 1.

int **bytes_warning**

如果等于 1，则在将 bytes 或 bytearray 与 str 进行比较，或将 bytes 与 int 进行比较时发出警告。

如果大于等于 2，则在这些情况下引发 BytesWarning 异常。

由 -b 命令行选项执行递增。

默认值: 0.

int **warn_default_encoding**

如为非零值，则在 io.TextIOWrapper 使用默认编码格式时发出 EncodingWarning 警告。详情请参阅 io-encoding-warning。

默认值: 0.

Added in version 3.10.

int **code_debug_ranges**

如果等于 0，则禁用在代码对象中包括末尾行和列映射。并且禁用在特定错误位置打印回溯标记。

通过 PYTHONNODEBUGRANGES 环境变量和 -X no_debug_ranges 命令行选项设置为 0。

默认值: 1.

Added in version 3.11.

wchar_t ***check_hash_pycs_mode**

控制基于哈希值的 .pyc 文件的验证行为: --check-hash-based-pycs 命令行选项的值。

有效的值:

- L"always": 无论'check_source' 旗标的值是什么都会对源文件进行哈希验证。
- L"never": 假定基于哈希值的 pyc 始终是有效的。
- L"default": 基于哈希值的 pyc 中的'check_source' 旗标确定是否验证无效。

默认值: L"default"。

参见 **PEP 552** "Deterministic pycs"。

int **configure_c_stdio**

如为非零值，则配置 C 标准流:

- 在 Windows 中，在 stdin, stdout 和 stderr 上设置二进制模式 (O_BINARY)。
- 如果*buffered_stdio* 等于零，则禁用 stdin, stdout 和 stderr 流的缓冲。
- 如果*interactive* 为非零值，则启用 stdin 和 stdout 上的流缓冲（Windows 中仅限 stdout）。

默认值: 在 Python 配置中为 1，在隔离配置中为 0。

int **dev_mode**

如果为非零值，则启用 Python 开发模式。

通过 -X dev 选项和 PYTHONDEVMODE 环境变量设置为 1。

默认值: 在 Python 模式中为 -1，在隔离模式中为 0。

int **dump_refs**

转储 Python 引用?

如果为非零值，则转储所有在退出时仍存活的对象。

由 PYTHONDUMPREFS 环境变量设置为 1。

需要定义了 Py_TRACE_REFS 宏的特殊 Python 编译版：参见 configure --with-trace-refs 选项。

默认值: 0.

wchar_t ***exec_prefix**

安装依赖于平台的 Python 文件的站点专属目录前缀: sys.exec_prefix。

默认值: NULL.

Python 路径配置 的一部分。

另请参阅*PyConfig.base_exec_prefix*。

wchar_t ***executable**

Python 解释器可执行二进制文件的绝对路径: sys.executable。

默认值: NULL.

Python 路径配置 的一部分。

另请参阅*PyConfig.base_executable*。

int **faulthandler**

启用 faulthandler?

如果为非零值，则在启动时调用 faulthandler.enable()。

通过 -X faulthandler 和 PYTHONFAULTHANDLER 环境变量设为 1。

默认值: 在 Python 模式中为 -1，在隔离模式中为 0。

wchar_t ***filesystem_encoding**

文件系统编码格式: sys.getfilesystemencoding()。

在 macOS, Android 和 VxWorks 上：默认使用 "utf-8"。

在 Windows 上: 默认使用 "utf-8", 或者如果*PyPreConfig* 的*legacy_windows_fs_encoding* 为非零值则使用 "mbcs"。

在其他平台上的默认编码格式：

- 如果*PyPreConfig.utf8_mode* 为非零值则使用 "utf-8"。
- 如果 Python 检测到 nl_langinfo(CODESET) 声明为 ASCII 编码格式，而 mbstowcs() 是从其他的编码格式解码（通常为 Latin1）则使用 "ascii"。
- 如果 nl_langinfo(CODESET) 返回空字符串则使用 "utf-8"。
- 在其他情况下，使用*locale encoding*: nl_langinfo(CODESET) 的结果。

在 Python 启动时，编码格式名称会规范化为 Python 编解码器名称。例如，"ANSI_X3.4-1968" 将被替换为 "ascii"。

参见*filesystem_errors* 的成员。

wchar_t ***filesystem_errors**

文件系统错误处理器: sys.getfilesystemencodeerrors()。

在 Windows 上：默认使用 "surrogatepass"，或者如果*PyPreConfig* 的*legacy_windows_fs_encoding* 为非零值则使用 "replace"。

在其他平台上：默认使用 "surrogateescape"。

支持的错误处理器：

- "strict"
- "surrogateescape"
- "surrogatepass" (仅支持 UTF-8 编码格式)

参见*filesystem_encoding* 的成员。

unsigned long **hash_seed**

int **use_hash_seed**

随机化的哈希函数种子。

如果*use_hash_seed* 为零，则在 Python 启动时随机选择一个种子，并忽略*hash_seed*。

由 PYTHONHASHSEED 环境变量设置。

默认的 *use_hash_seed* 值：在 Python 模式下为 -1，在隔离模式下为 0。

wchar_t ***home**

设置默认的 Python "home" 目录，即标准 Python 库所在的位置 (参见 PYTHONHOME)。

由 PYTHONHOME 环境变量设置。

默认值: NULL.

Python 路径配置 输入的一部分。

int **import_time**

如为非零值，则对导入时间执行性能分析。

通过 -X importtime 选项和 PYTHONPROFILEIMPORTTIME 环境变量设置为 1。

默认值: 0.

int **inspect**

在执行脚本或命令之后进入交互模式。

如果大于 0 ，则启用检查：当脚本作为第一个参数传入或使用了 -c 选项时，在执行脚本或命令后进入交互模式，即使在 sys.stdin 看来并非一个终端时也是如此。

通过 -i 命令行选项执行递增。如果 PYTHONINSPECT 环境变量为非空值则设为 1。

默认值: 0.

int **install_signal_handlers**

安装 Python 信号处理器?

默认值：在 Python 模式下为 1，在隔离模式下为 0。

int **interactive**

如果大于 0，则启用交互模式（REPL）。

由 -i 命令行选项执行递增。

默认值: 0.

int **int_max_str_digits**

配置 整数字符串转换长度限制。初始值为 -1 表示该值将从命令行或环境获取否则默认为 4300 (sys.int_info.default_max_str_digits)。值为 0 表示禁用限制。大于 0 但小于 640 (sys.int_info.str_digits_check_threshold) 的值将不被支持并会产生错误。

通过 -X int_max_str_digits 命令行旗标或 PYTHONINTMAXSTRDIGITS 环境变量配置。

默认值：在 Python 模式下为 -1 。在孤立模式下为 4300 (sys.int_info.default_max_str_digits)。

Added in version 3.12.

int **cpu_count**

如果*cpu_count* 的值不为 -1 则它将覆盖 os.cpu_count(), os.process_cpu_count() 和 multiprocessing.cpu_count() 的返回值。

通过 -X cpu_count=*n*|*default* 命令行旗标或 PYTHON_CPU_COUNT 环境变量来配置。

默认值: -1。

Added in version 3.13.

int **isolated**

如果大于 0 ，则启用隔离模式:

- 将*safe_path* 设为 1: 在 Python 启动时将不在 sys.path 前添加有潜在不安全性的路径，如当前目录、脚本所在目录或空字符串。
- 将*use_environment* 设为 0: 忽略 PYTHON 环境变量。
- 将*user_site_directory* 设为 0: 不要将用户级站点目录添加到 sys.path。
- Python REPL 将不导入 readline 也不在交互提示符中启用默认的 readline 配置。

通过 -I 命令行选项设置为 1。

默认值: 在 Python 模式中为 0，在隔离模式中为 1。

另请参阅隔离配置 和*PyPreConfig.isolated*。

int **legacy_windows_stdio**

如为非零值，则使用 io.FileIO 代替 io._WindowsConsoleIO 作为 sys.stdin、sys.stdout 和 sys.stderr。

如果 PYTHONLEGACYWINDOWSSTDIO 环境变量被设为非空字符串则设为 1。

仅在 Windows 上可用。#ifdef MS_WINDOWS 宏可被用于 Windows 专属的代码。

默认值: 0.

另请参阅 **PEP 528** (将 Windows 控制台编码格式更改为 UTF-8)。

int **malloc_stats**

如为非零值，则在退出时转储*Python pymalloc 内存分配器* 的统计数据。

由 PYTHONMALLOCSTATS 环境变量设置为 1。

如果 Python 是 使用 --without-pymalloc 选项进行配置则该选项将被忽略。

默认值: 0.

wchar_t ***platlibdir**

平台库目录名称: sys.platlibdir。

由 PYTHONPLATLIBDIR 环境变量设置。

默认值: 由 configure --with-platlibdir 选项设置的 PLATLIBDIR 宏的值 (默认值: "lib"，在 Windows 上则为 "DLLs")。

Python 路径配置 输入的一部分。

Added in version 3.9.

在 3.11 版本发生变更: 目前在 Windows 系统中该宏被用于定位标准库扩展模块，通常位于 DLLs 下。不过，出于兼容性考虑，请注意在任何非标准布局包括树内构建和虚拟环境中，该值都将被忽略。

wchar_t ***pythonpath_env**

模块搜索路径 (sys.path) 为一个用 DELIM (os.pathsep) 分隔的字符串。

由 PYTHONPATH 环境变量设置。

默认值: NULL.

Python 路径配置 输入的一部分。

PyWideStringList **module_search_paths**

int **module_search_paths_set**

模块搜索路径: `sys.path`。

如果*module_search_paths_set* 等于 0，*Py_InitializeFromConfig()* 将替代*module_search_paths* 并将*module_search_paths_set* 设为 1。

默认值：空列表 (`module_search_paths`) 和 0 (`module_search_paths_set`)。

Python 路径配置 的一部分。

int **optimization_level**

编译优化级别：

- 0: Peephole 优化器，将 `__debug__` 设为 `True`。
- 1: 0 级，移除断言，将 `__debug__` 设为 `False`。
- 2: 1 级，去除文档字符串。

通过 `-O` 命令行选项递增。设置为 `PYTHONOPTIMIZE` 环境变量值。

默认值: 0.

PyWideStringList **orig_argv**

传给 Python 可执行程序的原始命令行参数列表: `sys.orig_argv`。

如果*orig_argv* 列表为空并且*argv* 不是一个只包含空字符串的列表，*PyConfig_Read()* 将在修改*argv* 之前把*argv* 拷贝至*orig_argv* (如果*parse_argv* 不为空)。

另请参阅*argv* 成员和*Py_GetArgcArgv()* 函数。

默认值：空列表。

Added in version 3.10.

int **parse_argv**

解析命令行参数?

如果等于 1，则以与常规 Python 解析 命令行参数相同的方式解析*argv*，并从*argv* 中剥离 Python 参数。

PyConfig_Read() 函数对*PyConfig.argv* 参数只会解析一次：在参数解析完成后*PyConfig.parse_argv* 将被设为 2。由于 Python 参数是从*PyConfig.argv* 提取的，因此解析参数两次会将应用程序选项解析为 Python 选项。

默认值：在 Python 模式下为 1，在隔离模式下为 0。

在 3.10 版本发生变更: 现在只有当*PyConfig.parse_argv* 等于 1 时才会解析*PyConfig.argv* 参数。

int **parser_debug**

解析器调试模式。如果大于 0，则打开解析器调试输出（仅针对专家，取决于编译选项）。

通过 `-d` 命令行选项递增。设置为 `PYTHONDEBUG` 环境变量值。

需要 Python 调试编译版 (必须已定义 `Py_DEBUG` 宏)。

默认值: 0.

int **pathconfig_warnings**

如为非零值，则允许计算路径配置以将警告记录到 `stderr` 中。如果等于 0，则抑制这些警告。

默认值：在 Python 模式下为 1，在隔离模式下为 0。

Python 路径配置 输入的一部分。

在 3.11 版本发生变更: 现在也适用于 Windows。

wchar_t ***prefix**

安装依赖于平台的 Python 文件的站点专属目录前缀: `sys.prefix`。

默认值: `NULL`.

Python 路径配置 的一部分。

另请参阅*PyConfig.base_prefix*。

wchar_t ***program_name**

用于初始化*executable* 和在 Python 初始化期间早期错误消息中使用的程序名称。

- 在 macOS 上，如果设置了 `PYTHONEXECUTABLE` 环境变量则会使用它。
- 如果定义了 `WITH_NEXT_FRAMEWORK` 宏，当设置了 `__PYVENV_LAUNCHER__` 环境变量时将会使用它。
- 如果*argv* 的 `argv[0]` 可用并且不为空值则会使用它。
- 否则，在 Windows 上将使用 `L"python"`，在其他平台上将使用 `L"python3"`。

默认值: `NULL`.

Python 路径配置 输入的一部分。

wchar_t ***pycache_prefix**

缓存 `.pyc` 文件被写入到的目录: `sys.pycache_prefix`。

通过 `-X pycache_prefix=PATH` 命令行选项和 `PYTHONPYCACHEPREFIX` 环境变量设置。命令行选项优先级更高。

如果为 `NULL`，则 `sys.pycache_prefix` 将被设为 `None`。

默认值: `NULL`.

int **quiet**

安静模式。如果大于 `0`，则在交互模式下启动 Python 时不显示版权和版本。

由 `-q` 命令行选项执行递增。

默认值: `0`.

wchar_t ***run_command**

`-c` 命令行选项的值。

由*Py_RunMain()* 使用。

默认值: `NULL`.

wchar_t ***run_filename**

通过命令行传入的文件名：不包含 `-c` 或 `-m` 的附加命令行参数。它会被*Py_RunMain()* 函数使用。

例如，对于命令行 `python3 script.py arg` 它将被设为 `script.py`。

另请参阅*PyConfig.skip_source_first_line* 选项。

默认值: `NULL`.

wchar_t ***run_module**

`-m` 命令行选项的值。

由*Py_RunMain()* 使用。

默认值: `NULL`.

wchar_t ***run_presite**

package.module 模块路径，它应当在运行 site.py 之前被导入。

通过 -X presite=package.module 命令行选项和 PYTHON_PRESITE 环境变量设置。命令行选项优先级更高。

需要 Python 调试编译版 (必须已定义 Py_DEBUG 宏)。

默认值: NULL.

int **show_ref_count**

是否要在退出时显示总引用计数 (不包括*immortal* 对象)?

通过 -X showrefcount 命令行选项设置为 1。

需要 Python 调试编译版 (必须已定义 Py_REF_DEBUG 宏)。

默认值: 0.

int **site_import**

在启动时导入 site 模块?

如果等于零，则禁用模块站点的导入以及由此产生的与站点相关的 sys.path 操作。

如果以后显式地导入 site 模块也要禁用这些操作（如果你希望触发这些操作，请调用 site.main() 函数）。

通过 -S 命令行选项设置为 0。

sys.flags.no_site 会被设为*site_import* 取反后的值。

默认值: 1.

int **skip_source_first_line**

如为非零值，则跳过*PyConfig.run_filename* 源的第一行。

它将允许使用非 Unix 形式的 #!cmd。这是针对 DOS 专属的破解操作。

通过 -x 命令行选项设置为 1。

默认值: 0.

wchar_t ***stdio_encoding**

wchar_t ***stdio_errors**

sys.stdin、sys.stdout 和 sys.stderr 的编码格式和编码格式错误（但 sys.stderr 将始终使用 "backslashreplace" 错误处理器）。

如果 PYTHONIOENCODING 环境变量非空则会使用它。

默认编码格式：

- 如果*PyPreConfig.utf8_mode* 为非零值则使用 "UTF-8"。
- 在其他情况下，使用*locale encoding*。

默认错误处理器：

- 在 Windows 上：使用 "surrogateescape"。
- 如果*PyPreConfig.utf8_mode* 为非零值, 或者如果 LC_CTYPE 语言区域为”C” 或”POSIX” 则使用 "surrogateescape"。
- 在其他情况下则使用 "strict"。

另请参阅*PyConfig.legacy_windows_stdio*。

int **tracemalloc**

启用 tracemalloc?

如果为非零值，则在启动时调用 `tracemalloc.start()`。

通过 `-X tracemalloc=N` 命令行选项和 `PYTHONTRACEMALLOC` 环境变量设置。

默认值: 在 Python 模式中为 `-1`，在隔离模式中为 `0`。

int **perf_profiling**

启用与 perf 性能分析器兼容的模式?

如果为非零值，则初始化 perf trampoline。更多信息参见 perf_profiling。

对于带栈指针的 perf 支持通过 `-X perf` 命令行选项和 `PYTHON_PERF_JIT_SUPPORT` 环境变量设置，对于带 DWARF JIT 信息的 perf 支持通过 `-X perf_jit` 命令行选项和 `PYTHON_PERF_JIT_SUPPORT` 环境变量设置。

默认值: `-1`。

Added in version 3.12.

int **use_environment**

使用 环境变量?

如果等于零，则忽略 环境变量。

由 `-E` 环境变量设置为 `0`。

默认值: 在 Python 配置中为 `1` 而在隔离配置中为 `0`。

int **user_site_directory**

如果为非零值，则将用户站点目录添加到 `sys.path`。

通过 `-s` 和 `-I` 命令行选项设置为 `0`。

由 `PYTHONNOUSERSITE` 环境变量设置为 `0`。

默认值：在 Python 模式下为 `1`，在隔离模式下为 `0`。

int **verbose**

详细模式。如果大于 `0`，则每次导入模块时都会打印一条消息，显示加载模块的位置（文件名或内置模块）。

如果大于等于 `2`，则为搜索模块时每个被检查的文件打印一条消息。还在退出时提供关于模块清理的信息。

由 `-v` 命令行选项执行递增。

通过 `PYTHONVERBOSE` 环境变量值设置。

默认值: `0`.

PyWideStringList **warnoptions**

`warnings` 模块用于构建警告过滤器的选项，优先级从低到高: `sys.warnoptions`。

`warnings` 模块以相反的顺序添加 `sys.warnoptions`: 最后一个*PyConfig.warnoptions* 条目将成为 `warnings.filters` 的第一个条目并将最先被检查（最高优先级）。

`-W` 命令行选项会将其值添加到*warnoptions* 中，它可以被多次使用。

`PYTHONWARNINGS` 环境变量也可被用于添加警告选项。可以指定多个选项，并以逗号 (`,`) 分隔。

默认值：空列表。

int **write_bytecode**

如果等于 `0`，Python 将不会尝试在导入源模块时写入 `.pyc` 文件。

通过 `-B` 命令行选项和 `PYTHONDONTWRITEBYTECODE` 环境变量设置为 `0`。

`sys.dont_write_bytecode` 会被初始化为*write_bytecode* 取反后的值。

默认值: 1.

PyWideStringList **xoptions**

-X 命令行选项的值: sys._xoptions。

默认值：空列表。

如果*parse_argv* 为非零值，则*argv* 参数将以与常规 Python 解析 命令行参数相同的方式被解析，并从*argv* 中剥离 Python 参数。

xoptions 选项将会被解析以设置其他选项：参见 -X 命令行选项。

在 3.9 版本发生变更: show_alloc_count 字段已被移除。

10.7 使用 PyConfig 初始化

基于一个已填充内容的配置结构体初始化解释器是通过调用*Py_InitializeFromConfig()* 来处理的。

调用方要负责使用*PyStatus_Exception()* 和*Py_ExitStatusException()* 来处理异常（错误或退出）。

如果使用了*PyImport_FrozenModules()*、*PyImport_AppendInittab()* 或*PyImport_ExtendInittab()*，则必须在 Python 预初始化之后、Python 初始化之前设置或调用它们。如果 Python 被多次初始化，则必须在每次初始化 Python 之前调用*PyImport_AppendInittab()* 或*PyImport_ExtendInittab()*。

当前的配置 (PyConfig 类型) 保存在 PyInterpreterState.config 中。

设置程序名称的示例:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* 设置程序名称。 隐式地预初始化 Python。 */
    status = PyConfig_SetString(&config, &config.program_name,
                                L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);
    return;

exception:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

更完整的示例会修改默认配置，读取配置，然后覆盖某些参数。请注意自 3.11 版开始，许多参数在初始化之前不会被计算，因此无法从配置结构体中读取值。在调用初始化之前设置的任何值都将不会被初始化操作改变:

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;

    PyConfig config;
```

(续下页)

(接上页)

```
    PyConfig_InitPythonConfig(&config);

    /* 在读取配置之前设置程序名称
       (基于语言区域的编码格式来解码字节串)。

       Implicitly preinitialize Python. */
    status = PyConfig_SetBytesString(&config, &config.program_name,
                                     program_name);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* 一次性读取所有配置 */
    status = PyConfig_Read(&config);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* 显式地指定 sys.path */
    /* 如果你希望修改默认的路径集合,
       可先完成初始化再使用 PySys_GetObject("path") */
    config.module_search_paths_set = 1;
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/stdlib");
    if (PyStatus_Exception(status)) {
        goto done;
    }
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/more/modules");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Override executable computed by PyConfig_Read() */
    status = PyConfig_SetString(&config, &config.executable,
                                L"/path/to/my_executable");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    status = Py_InitializeFromConfig(&config);

done:
    PyConfig_Clear(&config);
    return status;
}
```

10.8 隔离配置

PyPreConfig_InitIsolatedConfig() 和*PyConfig_InitIsolatedConfig()* 函数会创建一个配置来将 Python 与系统隔离开来。例如，将 Python 嵌入到某个应用程序。

该配置将忽略全局配置变量、环境变量、命令行参数 (*PyConfig.argv* 将不会被解析) 和用户站点目录。C 标准流 (例如 `stdout`) 和 LC_CTYPE 语言区域将保持不变。信号处理器将不会被安装。

该配置仍然会使用配置文件来确定未被指明的路径。请确保指定了*PyConfig.home* 以避免计算默认的路径配置。

10.9 Python 配置

PyPreConfig_InitPythonConfig() 和*PyConfig_InitPythonConfig()* 函数会创建一个配置来构建一个行为与常规 Python 相同的自定义 Python。

环境变量和命令行参数将被用于配置 Python，而全局配置变量将被忽略。

此函数将根据 LC_CTYPE 语言区域、PYTHONUTF8 和 PYTHONCOERCECLOCALE 环境变量启用 C 语言区域强制转换 (**PEP 538**) 和 Python UTF-8 模式 (**PEP 540**)。

10.10 Python 路径配置

PyConfig 包含多个用于路径配置的字段：

- 路径配置输入：
  - *PyConfig.home*
  - *PyConfig.platlibdir*
  - *PyConfig.pathconfig_warnings*
  - *PyConfig.program_name*
  - *PyConfig.pythonpath_env*
  - 当前工作目录：用于获取绝对路径
  - PATH 环境变量用于获取程序的完整路径 (来自*PyConfig.program_name*)
  - __PYVENV_LAUNCHER__ 环境变量
  - （仅限 Windows only）注册表 HKEY_CURRENT_USER 和 HKEY_LOCAL_MACHINE 的”SoftwarePythonPythonCoreX.YPythonPath” 项下的应用程序目录（其中 X.Y 为 Python 版本）。
- 路径配置输出字段：
  - *PyConfig.base_exec_prefix*
  - *PyConfig.base_executable*
  - *PyConfig.base_prefix*
  - *PyConfig.exec_prefix*
  - *PyConfig.executable*
  - *PyConfig.module_search_paths_set*, *PyConfig.module_search_paths*
  - *PyConfig.prefix*

如果至少有一个“输出字段”未被设置，Python 就会计算路径配置来填充未设置的字段。如果*module_search_paths_set* 等于 0，则*module_search_paths* 将被覆盖并且*module_search_paths_set* 将被设置为 1。

通过显式地设置上述所有路径配置输出字段可以完全忽略计算默认路径配置的函数。即使字符串不为空也会被视为已设置。如果 module_search_paths_set 被设为 1 则 module_search_paths 会被视为已设置。在这种情况下，module_search_paths 将不加修改地被使用。

将*pathconfig_warnings* 设为 0 以便在计算路径配置时抑制警告（仅限 Unix，Windows 不会记录任何警告）。

如果*base_prefix* 或*base_exec_prefix* 字段未设置，它们将分别从*prefix* 和*exec_prefix* 继承其值。

Py_RunMain() 和*Py_Main()* 将修改 sys.path：

- 如果*run_filename* 已设置并且是一个包含 __main__.py 脚本的目录，则会将*run_filename* 添加到 sys.path 的开头。
- 如果*isolated* 为零：

  - 如果设置了*run_module*，则将当前目录添加到 sys.path 的开头。如果无法读取当前目录则不执行任何操作。
  - 如果设置了*run_filename*，则将文件名的目录添加到 sys.path 的开头。
  - 在其他情况下，则将一个空字符串添加到 sys.path 的开头。

如果*site_import* 为非零值，则 sys.path 可通过 site 模块修改。如果*user_site_directory* 为非零值且用户的 site-package 目录存在，则 site 模块会将用户的 site-package 目录附加到 sys.path。

路径配置会使用以下配置文件：

- pyvenv.cfg
- ._pth 文件 (例如: python._pth)
- pybuilddir.txt (仅 Unix)

如果存在 ._pth 文件:

- 将*isolated* 设为 1。
- 将*use_environment* 设为 0。
- 将*site_import* 设为 0。
- 将*safe_path* 设为 1。

使用 __PYVENV_LAUNCHER__ 环境变量来设置*PyConfig.base_executable*。

10.11 Py_GetArgcArgv()

void **Py_GetArgcArgv**(int *argc, wchar_t ***argv)

在 Python 修改原始命令行参数之前，获取这些参数。

另请参阅*PyConfig.orig_argv* 成员。

10.12 多阶段初始化私有暂定 API

本节介绍的私有暂定 API 引入了多阶段初始化，它是 **PEP 432** 的核心特性：

- “核心”初始化阶段，“最小化的基本 Python”：
  - 内置类型；
  - 内置异常；
  - 内置和已冻结模块；
  - sys 模块仅部分初始化（例如: sys.path 尚不存在）。
- ” 主要” 初始化阶段，Python 被完全初始化：
  - 安装并配置 importlib；
  - 应用路径配置；
  - 安装信号处理器；
  - 完成 sys 模块初始化 (例如：创建 sys.stdout 和 sys.path)；
  - 启用 faulthandler 和 tracemalloc 等可选功能；
  - 导入 site 模块；
  - 等等.

私有临时 API：

- PyConfig._init_main: 如果设为 0，*Py_InitializeFromConfig()* 将在“核心”初始化阶段停止。

PyStatus **_Py_InitializeMain**(void)

进入“主要”初始化阶段，完成 Python 初始化。

在“核心”阶段不会导入任何模块，也不会配置 `importlib` 模块: 路径配置 只会在“主要”阶段期间应用。这可能允许在 Python 中定制 Python 以覆盖或微调路径配置，也可能会安装自定义的 `sys.meta_path` 导入器或导入钩子等等。

在核心阶段之后主要阶段之前，将有可能在 Python 中计算路径配置，这是 **PEP 432** 的动机之一。

“核心”阶段并没有完整的定义：在这一阶段什么应该可用什么不应该可用都尚未被指明。该 API 被标记为私有和暂定的：也就是说该 API 可以随时被修改甚至被移除直到设计出适用的公共 API。

在“核心”和“主要”初始化阶段之间运行 Python 代码的示例:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config._init_main = 0;

    /* ... 自定义 'config' 配置 ... */

    status = Py_InitializeFromConfig(&config);
    PyConfig_Clear(&config);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }

    /* 使用 sys.stderr 因为 sys.stdout 只能由 _Py_InitializeMain() 创建 */
    int res = PyRun_SimpleString(
        "import sys; "
        "print('Run Python code before _Py_InitializeMain', "
               "file=sys.stderr)");
    if (res < 0) {
        exit(1);
    }

    /* ... 这里添加更多配置代码 ... */

    status = _Py_InitializeMain();
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
}
```

CHAPTER 11

内存管理

11.1 概述

在 Python 中，内存管理涉及到一个包含所有 Python 对象和数据结构的私有堆（heap）。这个私有堆的管理由内部的 *Python 内存管理器*（*Python memory manager*）保证。Python 内存管理器有不同的组件来处理各种动态存储管理方面的问题，如共享、分割、预分配或缓存。

在最底层，一个原始内存分配器通过与操作系统的内存管理器交互，确保私有堆中有足够的空间来存储所有与 Python 相关的数据。在原始内存分配器的基础上，几个对象特定的分配器在同一堆上运行，并根据每种对象类型的特点实现不同的内存管理策略。例如，整数对象在堆内的管理方式不同于字符串、元组或字典，因为整数需要不同的存储需求和速度与空间的权衡。因此，Python 内存管理器将一些工作分配给对象特定分配器，但确保后者在私有堆的范围内运行。

Python 堆内存的管理是由解释器来执行，用户对它没有控制权，即使他们经常操作指向堆内内存块的对象指针，理解这一点十分重要。Python 对象和其他内部缓冲区的堆空间分配是由 Python 内存管理器按需通过本文档中列出的 Python/C API 函数进行的。

为了避免内存破坏，扩展的作者永远不应该试图用 C 库函数导出的函数来对 Python 对象进行操作，这些函数包括：`malloc()`,`calloc()`,`realloc()` 和 `free()`。这将导致 C 分配器和 Python 内存管理器之间的混用，引发严重后果，这是由于它们实现了不同的算法，并在不同的堆上操作。但是，我们可以安全地使用 C 库分配器为单独的目的分配和释放内存块，如下例所示：

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...执行一些涉及 buf 的 I/O 操作...
res = PyBytes_FromString(buf);
free(buf); /* 已分配的 */
return res;
```

在这个例子中，I/O 缓冲区的内存请求是由 C 库分配器处理的。Python 内存管理器只参与了分配作为结果返回的字节对象。

然而，在大多数情况下，都建议专门基于 Python 堆来分配内存，因为后者是由 Python 内存管理器控制的。例如，当解释器使用 C 编写的新对象类型进行扩展时就必须这样做。使用 Python 堆的另一个理由是需要能 *通知* Python 内存管理器有关扩展模块的内存需求。即使所请求的内存全部只用于内部的、高度特定的目的，将所有的内存请求交给 Python 内存管理器能让解释器对其内存占用的整体情况有更准确的了解。因此，在特定情况下，Python 内存管理器可能会触发或不触发适当的操作，如垃圾回收、内存压缩

或其他的预防性操作。请注意通过使用前面例子所演示的 C 库分配器，为 I/O 缓冲区分配的内存将完全不受 Python 内存管理器的控制。

参见

环境变量 `PYTHONMALLOC` 可被用来配置 Python 所使用的内存分配器。

环境变量 `PYTHONMALLOCSTATS` 可以用来在每次创建和关闭新的 pymalloc 对象区域时打印*pymalloc 内存分配器* 的统计数据。

11.2 分配器域

所有分配函数都归属于三个不同的“域”之一 (另请参阅*PyMemAllocatorDomain*)。这些域代表不同的分配策略并针对不同的目的进行了优化。每个域如何分配内存及每个域会调用哪些内部函数的详情被认为是实现细节，但是出于调试目的可以在这里 找到一张简化的表格。用于分配和释放内存块的 API 必须来自同一个域。例如，*PyMem_Free()* 必须被用来释放使用*PyMem_Malloc()* 分配的内存。

三个分配域分别是：

- 原始域：用于为通用内存缓冲区分配内存，其分配 必须转到系统分配器或可以在没有*GIL* 的情况下使用的分配器。内存将直接请求自系统。参见原始内存接口。
- “内存”域：用于为 Python 缓冲区和通用内存缓冲区分配内存，其分配必须在持有*GIL* 的情况下进行。内存将从 Python 私有堆获取。参见内存接口。
- 对象域：用于为 Python 对象分配内存。内存将从 Python 私有堆获取。参见对象分配器。

备注

自由线程 构建版要求仅 Python 对象使用“对象”域来分配并且所有 Python 对象都使用该域来分配。这不同于之前的 Python 版本，在之前版本中这只是最佳实践而非硬件要求。

例如，缓冲区（非 Python 对象）的分配应当使用*PyMem_Malloc()*，*PyMem_RawMalloc()* 或 `malloc()`，而不能用*PyObject_Malloc()*。

参见 内存分配 API。

11.3 原始内存接口

以下函数集封装了系统分配器。这些函数是线程安全的，不需要持有全局解释器锁。

默认原始内存分配器 使用以下函数: `malloc()`，`calloc()`，`realloc()` 和 `free()`；当请求零个字节时则调用 `malloc(1)` (或 `calloc(1, 1)`)。

Added in version 3.4.

void ***PyMem_RawMalloc**(size_t n)

属于稳定 ABI 自 *3.13* 版起. 分配 *n* 个字节并返回一个指向所分配内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_RawMalloc(1)` 一样。但是内存不会以任何方式被初始化。

void ***PyMem_RawCalloc**(size_t nelem, size_t elsize)

属于稳定 ABI 自 *3.13* 版起. 分配 *nelem* 个元素，每个元素的大小为 *elsize* 个字节，并返回指向所分配的内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。内存会被初始化为零。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_RawCalloc(1, 1)` 一样。

Added in version 3.5.

void *PyMem_RawRealloc(void *p, size_t n)

属于稳定 ABI 自 *3.13* 版起. 将 *p* 指向的内存块大小调整为 *n* 字节。以新旧内存块大小中的最小值为准，其中内容保持不变，

如果 *p* 是 NULL，则相当于调用 PyMem_RawMalloc(n)；如果 *n* 等于 0，则内存块大小会被调整，但不会被释放，返回非 NULL 指针。

除非 *p* 是 NULL，否则它必须是之前调用*PyMem_RawMalloc()*、*PyMem_RawRealloc()*或*PyMem_RawCalloc()*所返回的。

如果请求失败，*PyMem_RawRealloc()* 返回 NULL，*p* 仍然是指向先前内存区域的有效指针。

void PyMem_RawFree(void *p)

属于稳定 ABI 自 *3.13* 版起. 释放 *p* 指向的内存块。*p* 必须是之前调用*PyMem_RawMalloc()*、*PyMem_RawRealloc()*或*PyMem_RawCalloc()*所返回的指针。否则，或在 PyMem_RawFree(p) 之前已经调用过的情况下，未定义的行为会发生。

如果 *p* 是 NULL, 那么什么操作也不会进行。

11.4 内存接口

以下函数集，仿照 ANSI C 标准，并指定了请求零字节时的行为，可用于从 Python 堆分配和释放内存。

默认内存分配器 使用了*pymalloc 内存分配器*.

警告

在使用这些函数时，必须持有全局解释器锁（*GIL*）。

在 3.6 版本发生变更: 现在默认的分配器是 pymalloc 而非系统的 malloc()。

void *PyMem_Malloc(size_t n)

属于稳定 ABI. 分配 *n* 个字节并返回一个指向所分配内存的 void* 类型指针，如果请求失败则返回 NULL。

请求零字节可能返回一个独特的非 NULL 指针，就像调用了 PyMem_Malloc(1) 一样。但是内存不会以任何方式被初始化。

void *PyMem_Calloc(size_t nelem, size_t elsize)

属于稳定 ABI 自 *3.7* 版起. 分配 *nelem* 个元素，每个元素的大小为 *elsize* 个字节，并返回指向所分配的内存的 void* 类型指针，如果请求失败则返回 NULL。内存会被初始化为零。

请求零字节可能返回一个独特的非 NULL 指针，就像调用了 PyMem_Calloc(1, 1) 一样。

Added in version 3.5.

void *PyMem_Realloc(void *p, size_t n)

属于稳定 ABI. 将 *p* 指向的内存块大小调整为 *n* 字节。以新旧内存块大小中的最小值为准，其中内容保持不变，

如果 *p* 是 NULL，则相当于调用 PyMem_Malloc(n)；如果 *n* 等于 0，则内存块大小会被调整，但不会被释放，返回非 NULL 指针。

除非 *p* 是 NULL，否则它必须是之前调用*PyMem_Malloc()*、*PyMem_Realloc()*或*PyMem_Calloc()*所返回的。

如果请求失败，*PyMem_Realloc()* 返回 NULL，*p* 仍然是指向先前内存区域的有效指针。

void PyMem_Free(void *p)

属于稳定 ABI. 释放 *p* 指向的内存块。*p* 必须是之前调用*PyMem_Malloc()*、*PyMem_Realloc()*或*PyMem_Calloc()*所返回的指针。否则，或在 PyMem_Free(p) 之前已经调用过的情况下，未定义的行为会发生。

如果 *p* 是 NULL，那么什么操作也不会进行。

以下面向类型的宏为方便而提供。注意 *TYPE* 可以指任何 C 类型。

PyMem_New(TYPE, n)

与 *PyMem_Malloc()* 相同，但会分配 `(n * sizeof(TYPE))` 字节的内存。返回一个转换为 `TYPE*` 的指针。内存不会以任何方式被初始化。

PyMem_Resize(p, TYPE, n)

与 *PyMem_Realloc()* 类似，但内存块的大小被调整为 `(n * sizeof(TYPE))` 个字节。返回一个转换为 `TYPE*` 的指针。在返回时，*p* 将是一个指向新内存区域的指针，或者如果执行失败则为 NULL。

这是一个 C 预处理宏，*p* 总是被重新赋值。请保存 *p* 的原始值，以避免在处理错误时丢失内存。

void **PyMem_Del**(void *p)

与 *PyMem_Free()* 相同

此外，我们还提供了以下宏集用于直接调用 Python 内存分配器，而不涉及上面列出的 C API 函数。但是请注意，使用它们并不能保证跨 Python 版本的二进制兼容性，因此在扩展模块被弃用。

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

11.5 对象分配器

以下函数集，仿照 ANSI C 标准，并指定了请求零字节时的行为，可用于从 Python 堆分配和释放内存。

备注

当通过*自定义内存分配器* 部分描述的方法拦截该域中的分配函数时，无法保证这些分配器返回的内存可以被成功地转换成 Python 对象。

默认对象分配器 使用*pymalloc 内存分配器*.

警告

在使用这些函数时，必须持有*全局解释器锁（GIL）* 。

void ***PyObject_Malloc**(size_t n)

属于稳定 ABI. 分配 *n* 个字节并返回一个指向所分配内存的 `void*` 类型指针，如果请求失败则返回 NULL。

请求零字节可能返回一个独特的非 NULL 指针，就像调用了 `PyObject_Malloc(1)` 一样。但是内存不会以任何方式被初始化。

void ***PyObject_Calloc**(size_t nelem, size_t elsize)

属于稳定 ABI *自 3.7 版起*. 分配 *nelem* 个元素，每个元素的大小为 *elsize* 个字节，并返回指向所分配的内存的 `void*` 类型指针，如果请求失败则返回 NULL。内存会被初始化为零。

请求零字节可能返回一个独特的非 NULL 指针，就像调用了 `PyObject_Calloc(1, 1)` 一样。

Added in version 3.5.

void *PyObject_Realloc(void *p, size_t n)

属于稳定 ABI. 将 *p* 指向的内存块大小调整为 *n* 字节。以新旧内存块大小中的最小值为准，其中内容保持不变，

如果 *p* 是 NULL，则相当于调用 PyObject_Malloc(n)；如果 *n* 等于 0，则内存块大小会被调整，但不会被释放，返回非 NULL 指针。

除非 *p* 是 NULL，否则它必须是之前调用*PyObject_Malloc()*、*PyObject_Realloc()* 或*PyObject_Calloc()* 所返回的。

如果请求失败，*PyObject_Realloc()* 返回 NULL，*p* 仍然是指向先前内存区域的有效指针。

void PyObject_Free(void *p)

属于稳定 ABI. 释放 *p* 指向的内存块。*p* 必须是之前调用*PyObject_Malloc()*、*PyObject_Realloc()* 或*PyObject_Calloc()* 所返回的指针。否则，或在 PyObject_Free(p) 之前已经调用过的情况下，未定义行为会发生。

如果 *p* 是 NULL，那么什么操作也不会进行。

11.6 默认内存分配器

默认内存分配器：

配置	名称	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
发布版本	"pymalloc"	malloc	pymalloc	pymalloc
调试构建	"pymalloc_debug'	malloc + debug	pymalloc + debug	pymalloc + debug
没有 pymalloc 的发布版本	"malloc"	malloc	malloc	malloc
没有 pymalloc 的调试构建	"malloc_debug"	malloc + debug	malloc + debug	malloc + debug

说明：

- 名称：PYTHONMALLOC 环境变量的值。
- malloc：来自 C 标准库的系统分配器，C 函数：malloc()、calloc()、realloc() 和 free()。
- pymalloc：*pymalloc 内存分配器*.
- mimalloc: *mimalloc 内存分配器*。如果 mimalloc 不受支持则将使用 pymalloc 分配器。
- ”+ debug”: 附带*Python 内存分配器的调试钩子*.
- “调试构建”：调试模式下的 Python 构建。

11.7 自定义内存分配器

Added in version 3.4.

type PyMemAllocatorEx

用于描述内存块分配器的结构体。该结构体下列字段:

域	含意
`void *ctx`	作为第一个参数传入的用户上下文
`void* malloc(void *ctx, size_t size)`	分配一个内存块
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	分配一个初始化为 0 的内存块
`void* realloc(void *ctx, void *ptr, size_t new_size)`	分配一个内存块或调整其大小
`void free(void *ctx, void *ptr)`	释放一个内存块

在 3.5 版本发生变更: `PyMemAllocator` 结构被重命名为*`PyMemAllocatorEx`* 并新增了一个 `calloc` 字段。

type **PyMemAllocatorDomain**

用来识别分配器域的枚举类。域有:

PYMEM_DOMAIN_RAW

函数

- *PyMem_RawMalloc()*
- *PyMem_RawRealloc()*
- *PyMem_RawCalloc()*
- *PyMem_RawFree()*

PYMEM_DOMAIN_MEM

函数

- *PyMem_Malloc()*,
- *PyMem_Realloc()*
- *PyMem_Calloc()*
- *PyMem_Free()*

PYMEM_DOMAIN_OBJ

函数

- *PyObject_Malloc()*
- *PyObject_Realloc()*
- *PyObject_Calloc()*
- *PyObject_Free()*

void **PyMem_GetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

获取指定域的内存块分配器。

void **PyMem_SetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

设置指定域的内存块分配器。

当请求零字节时，新的分配器必须返回一个独特的非 `NULL` 指针。

对于*`PYMEM_DOMAIN_RAW`* 域，分配器必须是线程安全的：当分配器被调用时将不持有*GIL*。

对于其余的域，分配器也必须是线程安全的：分配器可以在不共享 `GIL` 的不同解释器中被调用。

如果新的分配器不是钩子（不调用之前的分配器），必须调用*`PyMem_SetupDebugHooks()`* 函数在新分配器上重新安装调试钩子。

另请参阅*`PyPreConfig.allocator`* 和*Preinitialize Python with PyPreConfig*。

> **警告**
>
> *PyMem_SetAllocator()* 没有以下合约:
>
> - 可以在*Py_PreInitialize()* 之后*Py_InitializeFromConfig()* 之前调用它来安装自定义的内存分配器。对于所安装的分配器除了域的规定以外没有任何其他限制（例如 Raw Domain 允许分配器在不持有 GIL 的情况下被调用）。请参阅有关分配器域的章节 来了解详情。
> - 如果在 Python 已完成初始化之后（即*Py_InitializeFromConfig()* 被调用之后）被调用则自定义分配器 **must** 必须包装现有的分配器。将现有分配器替换为任意的其他分配器是**不受支持的**。

在 3.12 版本发生变更: 所有分配器都必须是线程安全的。

void **PyMem_SetupDebugHooks**(void)

设置*Python 内存分配器的调试钩子* 以检测内存错误。

11.8 Python 内存分配器的调试钩子

当 Python 在调试模式下构建, *PyMem_SetupDebugHooks()* 函数在*Python 预初始化* 时被调用, 以在 Python 内存分配器上设置调试钩子以检测内存错误。

`PYTHONMALLOC` 环境变量可被用于在以发行模式下编译的 Python 上安装调试钩子（例如: `PYTHONMALLOC=debug`）。

PyMem_SetupDebugHooks() 函数可被用于在调用了*PyMem_SetAllocator()* 之后设置调试钩子。

这些调试钩子用特殊的、可辨认的位模式填充动态分配的内存块。新分配的内存用字节 `0xCD`（`PYMEM_CLEANBYTE`）填充，释放的内存用字节 `0xDD`（`PYMEM_DEADBYTE`）填充。内存块被填充了字节 `0xFD`（`PYMEM_FORBIDDENBYTE`）的“禁止字节”包围。这些字节串不太可能是合法的地址、浮点数或 ASCII 字符串

运行时检查:

- 检测对 API 的违反。例如: 检测对*PyMem_Malloc()* 分配的内存块调用*PyObject_Free()*。
- 检测缓冲区起始位置前的写入（缓冲区下溢）。
- 检测缓冲区终止位置后的写入（缓冲区溢出）。
- 检测当调用*PYMEM_DOMAIN_OBJ*（如: *PyObject_Malloc()*）和*PYMEM_DOMAIN_MEM*（如: *PyMem_Malloc()*）域的分配器函数时是否持有*GIL*。

在出错时，调试钩子使用 `tracemalloc` 模块来回溯内存块被分配的位置。只有当 `tracemalloc` 正在追踪 Python 内存分配，并且内存块被追踪时，才会显示回溯。

让 *S* = `sizeof(size_t)`。将 `2*S` 个字节添加到每个被请求的 *N* 字节数据块的两端。内存的布局像是这样，其中 p 代表由类似 malloc 或类似 realloc 的函数所返回的地址 (`p[i:j]` 表示从 `*(p+i)` 左侧开始到 `*(p+j)` 左侧止的字节数据切片；请注意对负索引号的处理与 Python 切片是不同的）:

`p[-2*S:-S]`

最初所要求的字节数。这是一个 size_t，为大端序（易于在内存转储中读取）。

`p[-S]`

API 标识符（ASCII 字符）:

- `'r'` 表示*PYMEM_DOMAIN_RAW*。
- `'m'` 表示*PYMEM_DOMAIN_MEM*。
- `'o'` 表示*PYMEM_DOMAIN_OBJ*。

`p[-S+1:0]`

PYMEM_FORBIDDENBYTE 的副本。用于捕获下层的写入和读取。

p[0:N]
所请求的内存，用 PYMEM_CLEANBYTE 的副本填充，用于捕获对未初始化内存的引用。当调用 realloc 之类的函数来请求更大的内存块时，额外新增的字节也会用 PYMEM_CLEANBYTE 来填充。当调用 free 之类的函数时，这些字节会用 PYMEM_DEADBYTE 来重写，以捕获对已释放内存的引用。当调用 realloc 之类的函数来请求更小的内存块时，多余的旧字节也会用 PYMEM_DEADBYTE 来填充。

p[N:N+S]
PYMEM_FORBIDDENBYTE 的副本。用于捕获超限的写入和读取。

p[N+S:N+2*S]
仅当定义了 `PYMEM_DEBUG_SERIALNO` 宏时会被使用（默认情况下将不定义）。

一个序列号，每次调用 malloc 或 realloc 之类的函数时都会递增 1。大端序的 `size_t`。如果之后检测到了“被破坏的内存”，此序列号提供了一个很好的手段用来在下次运行时设置中断点，以捕获该内存块被破坏的瞬间。obmalloc.c 中的静态函数 bumpserialno() 是唯一会递增序列号的函数，它的存在让你可以轻松地设置这样的中断点。

一个 realloc 之类或 free 之类的函数会先检查两端的 PYMEM_FORBIDDENBYTE 字节是否完好。如果它们被改变了，则会将诊断输出写入到 stderr，并且程序将通过 Py_FatalError() 中止。另一种主要的失败模式是当程序读到某种特殊的比特模式并试图将其用作地址时触发内存错误。如果你随即进入调试器并查看该对象，你很可能会看到它已完全被填充为 PYMEM_DEADBYTE (意味着已释放的内存被使用) 或 PYMEM_CLEANBYTE (意味着未初始货摊内存被使用)。

在 3.6 版本发生变更: *PyMem_SetupDebugHooks()* 函数现在也能在使用发布模式编译的 Python 上工作。当发生错误时，调试钩子现在会使用 `tracemalloc` 来获取已分配内存块的回溯信息。调试钩子现在还会在*PYMEM_DOMAIN_OBJ* 和*PYMEM_DOMAIN_MEM* 作用域的函数被调用时检查是否持有 GIL。

在 3.8 版本发生变更: 字节模式 `0xCB` (`PYMEM_CLEANBYTE`)、`0xDB` (`PYMEM_DEADBYTE`) 和 `0xFB` (`PYMEM_FORBIDDENBYTE`) 已被 `0xCD`、`0xDD` 和 `0xFD` 替代以使用与 Windows CRT 调试 `malloc()` 和 `free()` 相同的值。

11.9 pymalloc 分配器

Python 有一个针对短生命周期的小对象（小于或等于 512 字节）进行了优化的 *pymalloc* 分配器。它使用名为“arena”的内存映射，在 32 位平台上的固定大小为 256 KiB，在 64 位平台上的固定大小为 1 MiB。对于大于 512 字节的分配，它会回退为*PyMem_RawMalloc()* 和*PyMem_RawRealloc()*。

pymalloc 是*PYMEM_DOMAIN_MEM* (例如: *PyMem_Malloc()*) 和*PYMEM_DOMAIN_OBJ* (例如: *PyObject_Malloc()*) 域的默认分配器。

arena 分配器使用以下函数:

- Windows 上的 `VirtualAlloc()` 和 `VirtualFree()`,
- `mmap()` 和 `munmap()`，如果可用的话,
- 否则，`malloc()` 和 `free()` 。

如果 Python 配置了 `--without-pymalloc` 选项，那么此分配器将被禁用。也可以在运行时使用 `PYTHONMALLOC`` (例如: ``PYTHONMALLOC=malloc`) 环境变量来禁用它。

Typically, it makes sense to disable the pymalloc allocator when building Python with AddressSanitizer (`--with-address-sanitizer`) which helps uncover low level bugs within the C code.

11.9.1 自定义 pymalloc Arena 分配器

Added in version 3.4.

type **PyObjectArenaAllocator**
用来描述一个 arena 分配器的结构体。这个结构体有三个字段:

域	含意
`void *ctx`	作为第一个参数传入的用户上下文
`void* alloc(void *ctx, size_t size)`	分配一块 size 字节的区域
`void free(void *ctx, void *ptr, size_t size)`	释放一块区域

void **PyObject_GetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

获取 arena 分配器

void **PyObject_SetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

设置 arena 分配器

11.10 mimalloc 分配器

Added in version 3.13.

Python 会在下层平台提供支持的情况下支持 mimalloc 分配器。mimalloc ” 是一个具有优良运行效率特性的通用分配器。它最初由 Daan Leijen 针对 Koka 和 Lean 语言运行时系统开发。”

11.11 tracemalloc C API

Added in version 3.7.

int **PyTraceMalloc_Track**(unsigned int domain, uintptr_t ptr, size_t size)

在 `tracemalloc` 模块中跟踪一个已分配的内存块。

成功时返回 `0`，出错时返回 `-1`(无法分配内存来保存跟踪信息)。如果禁用了 tracemalloc 则返回 `-2`。

如果内存块已被跟踪，则更新现有跟踪信息。

int **PyTraceMalloc_Untrack**(unsigned int domain, uintptr_t ptr)

在 `tracemalloc` 模块中取消跟踪一个已分配的内存块。如果内存块未被跟踪则不执行任何操作。

如果 tracemalloc 被禁用则返回 `-2`，否则返回 `0`。

11.12 例子

以下是来自概述 小节的示例，经过重写以使 I/O 缓冲区是通过使用第一个函数集从 Python 堆中分配的:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...执行一些涉及 buf 的 I/O 操作... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* 使用 PyMem_Malloc 分配的 */
return res;
```

使用面向类型函数集的相同代码:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...执行一些涉及 buf 的 I/O 操作... */
res = PyBytes_FromString(buf);
```

(续下页)

(接上页)

```
PyMem_Del(buf); /* 使用 PyMem_New 分配的 */
return res;
```

请注意在以上两个示例中，缓冲区总是通过归属于相同集的函数来操纵的。事实上，对于一个给定的内存块必须使用相同的内存 API 族，以便使得混合不同分配器的风险减至最低。以下代码序列包含两处错误，其中一个被标记为 *fatal* 因为它混合了两种在不同堆上操作的不同分配器。

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* 错误 -- 应为 PyMem_Free() */
free(buf2);       /* 正确 -- 通过 malloc() 分配的 */
free(buf1);       /* 致命 -- 应为 PyMem_Del()  */
```

除了用于处理来自 Python 堆的原始内存块的函数，Python 中的对象还通过*PyObject_New*、*PyObject_NewVar* 和*PyObject_Del()* 进行分配和释放。

这些将在有关如何在 C 中定义和实现新对象类型的下一章中讲解。

CHAPTER 12

对象实现支持

本章描述了定义新对象类型时所使用的函数、类型和宏。

12.1 在堆上分配对象

PyObject ***_PyObject_New**(*PyTypeObject* *type)

返回值：新的引用。

PyVarObject ***_PyObject_NewVar**(*PyTypeObject* *type, *Py_ssize_t* size)

返回值：新的引用。

PyObject ***PyObject_Init**(*PyObject* *op, *PyTypeObject* *type)

返回值：借入的引用。属于稳定 ABI. 为新分配的对象 *op* 初始化它的类型和初始引用。返回初始化后的对象。对象的其他字段不会被影响。

PyVarObject ***PyObject_InitVar**(*PyVarObject* *op, *PyTypeObject* *type, *Py_ssize_t* size)

返回值：借入的引用。属于稳定 ABI. 它的功能和*PyObject_Init()* 一样，并且会初始化变量大小对象的长度信息。

PyObject_New(TYPE, typeobj)

使用 C 结构类型 *TYPE* 和 Python 类型对象 *typeobj* (`PyTypeObject*`) 分配一个新的 Python 对象。f 未在该 Python 对象标头中定义的字段不会被初始化。调用方将拥有对该对象的唯一引用（即引用计数将为 1）。内存分配的大小由类型对象的*tp_basicsize* 字段决定。

请注意如果 *typeobj* 设置了*Py_TPFLAGS_HAVE_GC* 则此函数将不适用。对于这样的对象，请改用*PyObject_GC_New()*。

PyObject_NewVar(TYPE, typeobj, size)

使用 C 结构类型 *TYPE* 和 Python 类型对象 *typeobj* (`PyTypeObject*`) 分配一个新的 Python 对象。未在该 Python 对象标头中定义的字段不会被初始化。被分配的内存允许 *TYPE* 结构加 *typeobj* 的*tp_itemsize* 字段所给出的 *size* (`Py_ssize_t`) 个字段。这对于实现像元组这样能够在构造时确定其大小的对象来说很有用。将字段数组嵌入到相同的内在分配中可减少内存分配的次数，这提高了内存管理效率。

请注意如果 *typeobj* 设置了*Py_TPFLAGS_HAVE_GC* 则此函数将不适用。对于这样的对象，请改用*PyObject_GC_NewVar()*。

void **PyObject_Del**(void *op)

释放使用*PyObject_New* 或*PyObject_NewVar* 分配给一个对象的内存。这通常由在对象的类型中指定的*tp_dealloc* 处理器来调用。在此调用之后该对象中的字段不应再被访问因为原来的内存已不再是一个有效的 Python 对象。

PyObject **_Py_NoneStruct**

这个对象是像 `None` 一样的 Python 对象。它可以使用*Py_None* 宏访问，该宏的拿到指向该对象的指针。

参见

模块对象
分配内存和创建扩展模块

12.2 公用对象结构体

大量的结构体被用于定义 Python 的对象类型。这一节描述了这些的结构体和它们的使用方法。

12.2.1 基本的对象类型和宏

所有的 Python 对象最终都会在对象的内存表示的开始部分共享少量的字段。这些字段由*PyObject* 和*PyVarObject* 类型来表示，相应地，这些类型又是由一些宏扩展来定义的，它们也直接或间接地被用于所有其他 Python 对象的定义。附加的宏可以在引用计数 下找到。

type **PyObject**

属于受限 API. （仅特定成员属于稳定 *ABI*。） 所有对象类型都是此类型的扩展。这是一个包含了 Python 将对象的指针当作对象来处理所需的信息的类型。在一个普通的“发行”编译版中，它只包含对象的引用计数和指向对应类型对象的指针。没有什么对象被实际声明为*PyObject*，但每个指向 Python 对象的指针都可以被转换为 *PyObject**。对成员的访问必须通过使用*Py_REFCNT* 和*Py_TYPE* 宏来完成。

type **PyVarObject**

属于受限 API. （仅特定成员属于稳定 *ABI*。） 这是一个添加了*ob_size* 字段的*PyObject* 扩展。它仅用于具有某些 长度标记的对象。此类型并不经常在 Python/C API 中出现。对成员的访问必须通过使用*Py_REFCNT*, *Py_TYPE* 和*Py_SIZE* 宏来完成。

PyObject_HEAD

这是一个在声明代表无可变长度对象的新类型时所使用的宏。PyObject_HEAD 宏被扩展为:

```
PyObject ob_base;
```

参见上面*PyObject* 的文档。

PyObject_VAR_HEAD

这是一个在声明代表每个实例具有可变长度的对象时所使用的宏。PyObject_VAR_HEAD 宏被扩展为:

```
PyVarObject ob_base;
```

参见上面*PyVarObject* 的文档。

PyTypeObject **PyBaseObject_Type**

属于稳定 ABI. 所有其他对象的基类，与 Python 中的 `object` 相同。

int **Py_Is**(*PyObject* *x, *PyObject* *y)

属于稳定 ABI 自 *3.10* 版起. 测试 *x* 是否为 *y* 对象，与 Python 中的 `x is y` 相同。

Added in version 3.10.

int **Py_IsNone**(*PyObject* *x)

属于稳定 ABI 自 *3.10* 版起. 测试一个对象是否为 `None` 单例，与 Python 中的 `x is None` 相同。

Added in version 3.10.

int **Py_IsTrue**(*PyObject* *x)

属于稳定 ABI 自 *3.10* 版起. 测试一个对象是否为 `True` 单例，与 Python 中的 `x is True` 相同。

Added in version 3.10.

int **Py_IsFalse**(*PyObject* *x)

属于稳定 ABI 自 *3.10* 版起. 测试一个对象是否为 `False` 单例，与 Python 中的 `x is False` 相同。

Added in version 3.10.

PyTypeObject ***Py_TYPE**(*PyObject* *o)

返回值：借入的引用。 获取 Python 对象 *o* 的类型。

返回一个*borrowed reference*。

使用*Py_SET_TYPE()* 函数来设置一个对象类型。

在 3.11 版本发生变更: *Py_TYPE()* 被改为一个内联的静态函数。形参类型不再是 `const` *PyObject*`*`。

int **Py_IS_TYPE**(*PyObject* *o, *PyTypeObject* *type)

如果对象 *o* 的类型为 *type* 则返回非零值。否则返回零。等价于: `Py_TYPE(o) == type`。

Added in version 3.9.

void **Py_SET_TYPE**(*PyObject* *o, *PyTypeObject* *type)

将对象 *o* 的类型设为 *type*。

Added in version 3.9.

Py_ssize_t **Py_SIZE**(*PyVarObject* *o)

获取 Python 对象 *o* 的大小。

使用*Py_SET_SIZE()* 函数来设置一个对象大小。

在 3.11 版本发生变更: *Py_SIZE()* 被改为一个内联静态函数。形参类型不再是 `const` *PyVarObject*`*`。

void **Py_SET_SIZE**(*PyVarObject* *o, *Py_ssize_t* size)

将对象 *o* 的大小设为 *size*。

Added in version 3.9.

PyObject_HEAD_INIT(type)

这是一个为新的*PyObject* 类型扩展初始化值的宏。该宏扩展为:

```
_PyObject_EXTRA_INIT
1, type,
```

PyVarObject_HEAD_INIT(type, size)

这是一个为新的*PyVarObject* 类型扩展初始化值的宏，包括*ob_size* 字段。该宏会扩展为:

```
_PyObject_EXTRA_INIT
1, type, size,
```

12.2.2 实现函数和方法

type **PyCFunction**

属于稳定 ABI. 用于在 C 中实现大多数 Python 可调用对象的函数类型。该类型的函数接受两个 *PyObject** 形参并返回一个这样的值。如果返回值为 NULL，则将设置一个异常。如果不为 NULL，则返回值将被解读为 Python 中暴露的函数的返回值。此函数必须返回一个新的引用。

函数的签名为:

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

type **PyCFunctionWithKeywords**

属于稳定 ABI. 用于在 C 中实现具有*METH_VARARGS | METH_KEYWORDS* 签名的 Python 可调用对象的函数类型。函数的签名为:

```
PyObject *PyCFunctionWithKeywords(PyObject *self,
                                  PyObject *args,
                                  PyObject *kwargs);
```

type **PyCFunctionFast**

属于稳定 ABI 自 *3.13* 版起. 用于在 C 中实现具有*METH_FASTCALL* 签名的 Python 可调用对象的函数类型。函数的签名为:

```
PyObject *PyCFunctionFast(PyObject *self,
                          PyObject *const *args,
                          Py_ssize_t nargs);
```

type **PyCFunctionFastWithKeywords**

属于稳定 ABI 自 *3.13* 版起. 用于在 C 中实现具有*METH_FASTCALL | METH_KEYWORDS* 签名的 Python 可调用对象的函数类型。函数的签名为:

```
PyObject *PyCFunctionFastWithKeywords(PyObject *self,
                                      PyObject *const *args,
                                      Py_ssize_t nargs,
                                      PyObject *kwnames);
```

type **PyCMethod**

用于在 C 中实现具有*METH_METHOD | METH_FASTCALL | METH_KEYWORDS* 签名的 Python 可调用对象的函数类型。函数的签名为:

```
PyObject *PyCMethod(PyObject *self,
                    PyTypeObject *defining_class,
                    PyObject *const *args,
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

Added in version 3.9.

type **PyMethodDef**

属于稳定 ABI（包括所有成员）. 用于描述一个扩展类型的方法的结构体。该结构体有四个字段:

const char ***ml_name**

方法的名称。

PyCFunction **ml_meth**

指向 C 语言实现的指针。

int **ml_flags**

指明调用应当如何构建的旗标位。

const char ***ml_doc**

指向文档字符串的内容。

ml_meth 是一个 C 函数指针。该函数可以为不同类型，但它们将总是返回 *PyObject**。如果该函数不属于*PyCFunction*，则编译器将要求在方法表中进行转换。尽管*PyCFunction* 将第一个参数定义为 *PyObject**，但该方法的实现使用 *self* 对象的特定 C 类型也很常见。

ml_flags 字段是可以包含以下旗标的位字段。每个旗标表示一个调用惯例或绑定惯例。

调用惯例有如下这些:

METH_VARARGS

这是典型的调用惯例，其中方法的类型为*PyCFunction*。该函数接受两个 *PyObject** 值。第一个是用于方法的 *self* 对象；对于模块函数，它将为模块对象。第二个形参 (常被命名为 *args*) 是一个代表所有参数的元组对象。该形参通常是使用*PyArg_ParseTuple()* 或*PyArg_UnpackTuple()* 来处理的。

METH_KEYWORDS

只能用于同其他旗标形成特定的组合: *METH_VARARGS* | *METH_KEYWORDS*, *METH_FASTCALL* | *METH_KEYWORDS* 和*METH_METHOD* | *METH_FASTCALL* | *METH_KEYWORDS*。

METH_VARARGS* | *METH_KEYWORDS

带有这些旗标的方法必须为*PyCFunctionWithKeywords* 类型。该函数接受三个形参: *self*, *args*, *kwargs* 其中 *kwargs* 是一个包含所有关键字参数的字典或者如果没有关键字参数则可以为 NULL。这些形参通常是使用*PyArg_ParseTupleAndKeywords()* 来处理的。

METH_FASTCALL

快速调用惯例仅支持位置参数。这些方法的类型为*PyCFunctionFast*。第一个形参为 *self*，第二个形参是由表示位置参数的由 *PyObject** 值组成的 C 数组而第三个形参是位置参数的数量（数组的长度）。

Added in version 3.7.

在 3.10 版本发生变更: METH_FASTCALL 现在是稳定 *ABI* 的一部分。

METH_FASTCALL* | *METH_KEYWORDS

METH_FASTCALL 的扩展也支持关键字参数，它使用类型为*PyCFunctionFastWithKeywords* 的方法。关键字参数的传递方式与*vectorcall 协议* 中的相同：还存在额外的第四个 *PyObject** 形参，它是一个代表关键字参数名称（它会保证是字符串）的元组，或者如果没有关键字则可以是 NULL。关键字参数的值存放在 *args* 数组中，在位置参数之后。

Added in version 3.7.

METH_METHOD

只能与其他旗标组合使用: *METH_METHOD* | *METH_FASTCALL* | *METH_KEYWORDS*。

METH_METHOD* | *METH_FASTCALL* | *METH_KEYWORDS

METH_FASTCALL | *METH_KEYWORDS* 的扩展支持 定义式类，也就是包含相应方法的类。定义式类可以是 Py_TYPE(self) 的超类。

该方法必须为*PyCMethod* 类型，与在 self 之后添加了 defining_class 参数的 METH_FASTCALL | METH_KEYWORDS 一样。

Added in version 3.9.

METH_NOARGS

如果通过*METH_NOARGS* 旗标列出了参数则没有形参的方法无需检查是否给出了参数。它们必须为*PyCFunction* 类型。第一个形参通常被命名为 *self* 并将持有对模块或对象实例的引用。在所有情况下第二个形参都将为 NULL。

该函数必须有 2 个形参。由于第二个形参不会被使用，*Py_UNUSED* 可以被用来防止编译器警告。

METH_O

具有一个单独对象参数的方法可使用*METH_O* 旗标列出，而不必唤起*PyArg_ParseTuple()* 并附带 "O" 参数。它们的类型为*PyCFunction*，带有 *self* 形参，以及代表该单独参数的 *PyObject** 形参。

这两个常量不是被用来指明调用惯例而是在配合类方法使用时指明绑定。它们不会被用于在模块上定义的函数。对于任何给定方法这些旗标最多只会设置其中一个。

METH_CLASS

该方法将接受类型对象而不是类型的实例作为第一个形参。它会被用于创建 *类方法*，类似于使用 `classmethod()` 内置函数所创建的结果。

METH_STATIC

该方法将接受 `NULL` 而不是类型的实例作为第一个形参。它会被用于创建 *静态方法*，类似于使用 `staticmethod()` 内置函数所创建的结果。

另一个常量控制方法是否将被载入来替代具有相同方法名的另一个定义。

METH_COEXIST

该方法将被加载以替代现有的定义。如果没有 *METH_COEXIST*，默认将跳过重复的定义。由于槽位包装器会在方法表之前被加载，例如当存在 *sq_contains* 槽位时，将会生成一个名为 `__contains__()` 的已包装方法并阻止加载同名的相应 PyCFunction。如果定义了此旗标，PyCFunction 将被加载以替代此包装器对象并与槽位共存。因为对 PyCFunction 的调用相比对包装器对象调用更为优化所以这是很有帮助的。

PyObject ***PyCMethod_New**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module, *PyTypeObject* *cls)

返回值：*新的引用*。属于稳定 ABI 自 *3.9* 版起. 将 *ml* 转为一个 Python *callable* 对象。调用方必须确保 *ml* 的生命期长于*callable*。通常，*ml* 会被定义为一个静态变量。

self 形参将在唤起时作为 `ml->ml_meth` 中 C 函数的 *self* 参数传入。*self* 可以为 `NULL`。

callable 对象的 `__module__` 属性可以根据给定的 *module* 参数来设置。*module* 应为一个 Python 字符串，它将被用作函数定义所在的模块名称。如果不可用，它将被设为 `None` 或 `NULL`。

参见

`function.__module__`

cls 形参将被作为 C 函数的 *defining_class* 参数传入。如果在 `ml->ml_flags` 上设置了*METH_METHOD* 则必须设置该形参。

Added in version 3.9.

PyObject ***PyCFunction_NewEx**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module)

返回值：*新的引用*。属于稳定 ABI. 等价于 `PyCMethod_New(ml, self, module, NULL)`。

PyObject ***PyCFunction_New**(*PyMethodDef* *ml, *PyObject* *self)

返回值：*新的引用*。属于稳定 ABI 自 *3.4* 版起. 等价于 `PyCMethod_New(ml, self, NULL, NULL)`。

12.2.3 访问扩展类型的属性

type **PyMemberDef**

属于稳定 ABI （包括所有成员）. 描述某个 C 结构成员对应类型的属性的结构体。在定义类时，要把由这些结构组成的以 NULL 结尾的数组放在*tp_members* 槽位中。

其中的字段及顺序如下:

const char ***name**

成员名称。NULL 值表示 `PyMemberDef[]` 数组的结束。

字符串应当是静态的，它不会被复制。

int **type**

C 结构体中成员的类型。请参阅成员类型 了解可能的取值。

Py_ssize_t **offset**

成员在类型的对象结构体中所在位置的以字节为单位的偏移量。

int **flags**

零个或多个成员旗标，使用按位或运算进行组合。

const char ***doc**

文档字符串，或者为空。该字符串应当是静态的，它不会被拷贝。通常，它是使用*PyDoc_STR* 来定义的。

默认情况下 (当*flags* 为 `0` 时)，成员同时允许读取和写入访问。使用*Py_READONLY* 旗标表示只读访问。某些类型，如*Py_T_STRING*，隐含要求*Py_READONLY*。只有*Py_T_OBJECT_EX* (以及旧式的*T_OBJECT*) 成员可以删除。

对于堆分配类型（使用*PyType_FromSpec()* 或类似函数创建），`PyMemberDef` 可能包含特殊成员 `"__vectorcalloffset__"` 的定义，与类型对象中的*tp_vectorcall_offset* 相对应。它们必须用 `Py_T_PYSSIZET` 和 `Py_READONLY` 来定义，例如:

```
static PyMemberDef spam_type_members[] = {
    {"__vectorcalloffset__", Py_T_PYSSIZET,
     offsetof(Spam_object, vectorcall), Py_READONLY},
    {NULL}  /* 哨兵 */
};
```

（您可能需要为 `offsetof()` 添加 `#include <stddef.h>`。）

旧式的偏移量*tp_dictoffset* 和*tp_weaklistoffset* 可使用 `"__dictoffset__"` 和 `"__weaklistoffset__"` 成员进行类似的定义，但强烈建议扩展程序改用*Py_TPFLAGS_MANAGED_DICT* 和*Py_TPFLAGS_MANAGED_WEAKREF*。

在 3.12 版本发生变更: `PyMemberDef` 将始终可用。在之前版本中，它需要包括 `"structmember.h"`。

PyObject ***PyMember_GetOne**(const char *obj_addr, struct *PyMemberDef* *m)

属于稳定 ABI. 获取属于地址 Get an attribute belonging to the object at address *obj_addr* 上的对象的某个属性。该属性是以 `PyMemberDef` *m* 来描述的。出错时返回 `NULL`。

在 3.12 版本发生变更: `PyMember_GetOne` 将总是可用。在之前版本中，它需要包括 `"structmember.h"`。

int **PyMember_SetOne**(char *obj_addr, struct *PyMemberDef* *m, *PyObject* *o)

属于稳定 ABI. 将属于位于地址 *obj_addr* 的对象的属性设置到对象 *o*。要设置的属性由 `PyMemberDef` *m* 描述。成功时返回 `0` 而失败时返回负值。

在 3.12 版本发生变更: `PyMember_SetOne` 将总是可用。在之前版本中，它需要包括 `"structmember.h"`。

成员旗标

以下旗标可被用于*PyMemberDef.flags*:

Py_READONLY

不可写入。

Py_AUDIT_READ

在读取之前发出一个 `object.__getattr__` 审计事件。

Py_RELATIVE_OFFSET

表示该 `PyMemberDef` 条目的*offset* 是指明来自子类专属数据的偏移量，而不是来自 `PyObject` 的偏移量。

只能在使用负的*basicsize* 创建类时被用作*Py_tp_members* 槽位 的组成部分。它在此种情况下是强制要求。

这个旗标只能在*PyType_Slot* 中使用。在类创建期间设置*tp_members* 时，Python 会清除它并将*PyMemberDef.offset* 设为相对于 `PyObject` 结构体的偏移量。

在 3.10 版本发生变更: 通过 `#include "structmember.h"` 提供的 `RESTRICTED`、`READ_RESTRICTED` 和 `WRITE_RESTRICTED` 宏已被弃用。`READ_RESTRICTED` 和 `RESTRICTED` 等同于*Py_AUDIT_READ*; `WRITE_RESTRICTED` 则没有任何作用。

在 3.12 版本发生变更: READONLY 宏被更名为*Py_READONLY*。PY_AUDIT_READ 宏被更名为 Py_ 前缀。新名称现在将始终可用。在之前的版本中，这些名称需要 #include "structmember.h"。该头文件仍然可用并提供了原有的名称。

成员类型

PyMemberDef.type 可以是下列与各种 C 类型相对应的宏之一。在 Python 中访问该成员时，它将被转换为对应的 Python 类型。当从 Python 设置成员时，它将被转换回 C 类型。如果无法转换，则会引发一个异常如 TypeError 或 ValueError。

除非标记为 (D)，否则不能使用 del 或 delattr() 删除以这种方式定义的属性。

宏名称	C 类型	Python 类型
Py_T_BYTE	`char`	`int`
Py_T_SHORT	`short`	`int`
Py_T_INT	`int`	`int`
Py_T_LONG	`long`	`int`
Py_T_LONGLONG	`long long`	`int`
Py_T_UBYTE	`unsigned char`	`int`
Py_T_UINT	`unsigned int`	`int`
Py_T_USHORT	`unsigned short`	`int`
Py_T_ULONG	`unsigned long`	`int`
Py_T_ULONGLONG	`unsigned long long`	`int`
Py_T_PYSSIZET	*Py_ssize_t*	`int`
Py_T_FLOAT	`float`	`float`
Py_T_DOUBLE	`double`	`float`
Py_T_BOOL	`char` (写为 0 或 1)	`bool`
Py_T_STRING	`const char*` (*)	`str` (RO)
Py_T_STRING_INPLACE	`const char[]` (*)	`str` (RO)
Py_T_CHAR	`char` (0-127)	`str` (**)
Py_T_OBJECT_EX	*PyObject*`*`	`object` (D)

(*): 以零结束的 UTF8 编码的 C 字符串。使用 `Py_T_STRING` 时的 C 表示形式是一个指针；使用 `Py_T_STRING_INPLACE` 时字符串将直接存储在结构体中。

(**): 长度为 1 的字符串。只接受 ASCII 字符。

(RO): 表示*Py_READONLY*。

(D): 可以删除，在这种情况下指针会被设为 NULL。读取 NULL 指针会引发 AttributeError。

Added in version 3.12: 在之前的版本中，这些宏仅通过 #include "structmember.h" 提供并且其名称不带 Py_ 前缀 (例如 T_INT)。头文件仍然可用并包含这些旧名称，以及下列已被弃用的类型:

T_OBJECT

与 Py_T_OBJECT_EX 类似，但 NULL 会被转换为 None。这将在 Python 中产生令人吃惊的行为: 删除该属性实际上会将其设置为 None。

T_NONE

总是为 None。必须与*Py_READONLY* 一起使用。

定义读取器和设置器

type **PyGetSetDef**

属于稳定 ABI （包括所有成员）. 用于定义针对某个类型的特征属性式的访问的结构体。另请参阅*PyTypeObject.tp_getset* 槽位的描述。

const char ***name**

属性名称

getter **get**

用于获取属性的 C 函数。

setter **set**

可选的用于设置或删除属性的 C 函数。如为 NULL，则属性将是只读的。

const char ***doc**

可选的文档字符串

void ***closure**

可选的用户数据指针，为 getter 和 setter 提供附加数据。

typedef *PyObject* *(***getter**)(*PyObject**, void*)

属于稳定 ABI. get 函数接受一个 *PyObject** 形参 (相应的实例) 和一个用户数据指针 (关联的 closure):

它应当在成功时返回一个新的引用或在失败时返回 NULL 并设置异常。

typedef int (***setter**)(*PyObject**, *PyObject**, void*)

属于稳定 ABI. set 函数接受两个 *PyObject** 形参 (相应的实例和要设置的值) 和一个用户数据指针 (关联的 closure):

对于属性要被删除的情况第二个形参应为 NULL。成功时应返回 0 或在失败时返回 -1 并设置异常。

12.3 类型对象结构体

Python 对象系统中最重要的一个结构体也许是定义新类型的结构体: *PyTypeObject* 结构体。类型对象可以使用任何 PyObject_* 或 PyType_* 函数来处理，但并未提供大多数 Python 应用程序会感兴趣的东西。这些对象是对象行为的基础，所以它们对解释器本身及任何实现新类型的扩展模块都非常重要。

与大多数标准类型相比，类型对象相当大。这么大的原因是每个类型对象存储了大量的值，大部分是 C 函数指针，每个指针实现了类型功能的一小部分。本节将详细描述类型对象的字段。这些字段将按照它们在结构中出现的顺序进行描述。

除了下面的快速参考，例子 小节提供了快速了解*PyTypeObject* 的含义和用法的例子。

12.3.1 快速参考

"tp 槽位"

PyTypeObject 槽位[Page 268, 1]	类型	特殊方法/属性	信息[Page 268, 2]			
			C	T	D	I
<R> *tp_name*	const char *	__name__	X	X		
tp_basicsize	*Py_ssize_t*		X	X		X
tp_itemsize	*Py_ssize_t*			X		X
tp_dealloc	*destructor*		X	X		X
tp_vectorcall_offset	*Py_ssize_t*			X		X
(*tp_getattr*)	*getattrfunc*	__getattribute__, __getattr__				G
(*tp_setattr*)	*setattrfunc*	__setattr__, __delattr__				G
tp_as_async	*PyAsyncMethods* *	子槽位				%
tp_repr	*reprfunc*	__repr__	X	X		X
tp_as_number	*PyNumberMethods* *	子槽位				%
tp_as_sequence	*PySequenceMethods* *	子槽位				%
tp_as_mapping	*PyMappingMethods* *	子槽位				%
tp_hash	*hashfunc*	__hash__	X			G
tp_call	*ternaryfunc*	__call__		X		X
tp_str	*reprfunc*	__str__	X			X
tp_getattro	*getattrofunc*	__getattribute__, __getattr__	X	X		G
tp_setattro	*setattrofunc*	__setattr__, __delattr__	X	X		G
tp_as_buffer	*PyBufferProcs* *	子槽位				%
tp_flags	unsigned long		X	X		?
tp_doc	const char *	__doc__	X	X		
tp_traverse	*traverseproc*			X		G
tp_clear	*inquiry*			X		G
tp_richcompare	*richcmpfunc*	__lt__, __le__, __eq__, __ne__, __gt__, __ge__	X			G
(*tp_weaklistoffset*)	*Py_ssize_t*			X		?
tp_iter	*getiterfunc*	__iter__				X
tp_iternext	*iternextfunc*	__next__				X
tp_methods	*PyMethodDef* []		X	X		
tp_members	*PyMemberDef* []			X		
tp_getset	*PyGetSetDef* []		X	X		
tp_base	*PyTypeObject* *	__base__			X	
tp_dict	*PyObject* *	__dict__			?	
tp_descr_get	*descrgetfunc*	__get__				X
tp_descr_set	*descrsetfunc*	__set__, __delete__				X
(*tp_dictoffset*)	*Py_ssize_t*			X		?
tp_init	*initproc*	__init__	X	X		X
tp_alloc	*allocfunc*		X		?	?
tp_new	*newfunc*	__new__	X	X	?	?
tp_free	*freefunc*		X	X	?	?
tp_is_gc	*inquiry*			X		X
<*tp_bases*>	*PyObject* *	__bases__			~	
<*tp_mro*>	*PyObject* *	__mro__			~	
[*tp_cache*]	*PyObject* *					
[*tp_subclasses*]	void *	__subclasses__				
[*tp_weaklist*]	*PyObject* *					
(*tp_del*)	*destructor*					
[*tp_version_tag*]	unsigned int					
tp_finalize	*destructor*	__del__				X
tp_vectorcall	*vectorcallfunc*					

续下页

表 1 – 接上页

PyTypeObject 槽位[1]	类型	特殊方法/属性	信息[2] C T D I
[*tp_watched*]	unsigned char		

子槽位

槽位	类型	特殊方法
am_await	*unaryfunc*	__await__
am_aiter	*unaryfunc*	__aiter__
am_anext	*unaryfunc*	__anext__
am_send	*sendfunc*	
nb_add	*binaryfunc*	__add__ __radd__
nb_inplace_add	*binaryfunc*	__iadd__
nb_subtract	*binaryfunc*	__sub__ __rsub__
nb_inplace_subtract	*binaryfunc*	__isub__
nb_multiply	*binaryfunc*	__mul__ __rmul__
nb_inplace_multiply	*binaryfunc*	__imul__
nb_remainder	*binaryfunc*	__mod__ __rmod__
nb_inplace_remainder	*binaryfunc*	__imod__
nb_divmod	*binaryfunc*	__divmod__ __rdivmod__
nb_power	*ternaryfunc*	__pow__ __rpow__
nb_inplace_power	*ternaryfunc*	__ipow__
nb_negative	*unaryfunc*	__neg__
nb_positive	*unaryfunc*	__pos__
nb_absolute	*unaryfunc*	__abs__
nb_bool	*inquiry*	__bool__
nb_invert	*unaryfunc*	__invert__
nb_lshift	*binaryfunc*	__lshift__ __rlshift__
nb_inplace_lshift	*binaryfunc*	__ilshift__
nb_rshift	*binaryfunc*	__rshift__ __rrshift__
nb_inplace_rshift	*binaryfunc*	__irshift__

续下页

[1] (): 括号中的插槽名称表示（实际上）已弃用。
<>: 尖括号内的名称在初始时应设为 NULL 并被视为是只读的。
[]: 方括号内的名称仅供内部使用。
<R> (作为前缀) 表示字段是必需的 (不能是 NULL)。

[2] 列:
"O": 在*PyBaseObject_Type* 上设置
"T": 在*PyType_Type* 上设置
"D": 默认设置 (如果方法槽被设置为 NULL)

```
X - PyType_Ready 如其为 NULL 则设置该值
~ - PyType_Ready 始终设置该值 (它应当为 NULL)
? - PyType_Ready 根据其他槽位可能设置该值

另请参阅继承列 ("I")。
```

"I": 继承

```
X - 如果使用 *NULL* 值定义则类型槽位将通过 *PyType_Ready* 继承
% - 子结构体的槽位是单独继承的
G - 已继承, 但仅会与其他槽位相结合; 参见槽位的说明
? - 较复杂; 参见槽位的说明
```

注意，有些方法槽是通过普通属性查找链有效继承的。

表 2 – 接上页

槽位	类型	特殊方法
nb_and	*binaryfunc*	__and__ __rand__
nb_inplace_and	*binaryfunc*	__iand__
nb_xor	*binaryfunc*	__xor__ __rxor__
nb_inplace_xor	*binaryfunc*	__ixor__
nb_or	*binaryfunc*	__or__ __ror__
nb_inplace_or	*binaryfunc*	__ior__
nb_int	*unaryfunc*	__int__
nb_reserved	void *	
nb_float	*unaryfunc*	__float__
nb_floor_divide	*binaryfunc*	__floordiv__
nb_inplace_floor_divide	*binaryfunc*	__ifloordiv__
nb_true_divide	*binaryfunc*	__truediv__
nb_inplace_true_divide	*binaryfunc*	__itruediv__
nb_index	*unaryfunc*	__index__
nb_matrix_multiply	*binaryfunc*	__matmul__ __rmatmul__
nb_inplace_matrix_multiply	*binaryfunc*	__imatmul__
mp_length	*lenfunc*	__len__
mp_subscript	*binaryfunc*	__getitem__
mp_ass_subscript	*objobjargproc*	__setitem__, __delitem__
sq_length	*lenfunc*	__len__
sq_concat	*binaryfunc*	__add__
sq_repeat	*ssizeargfunc*	__mul__
sq_item	*ssizeargfunc*	__getitem__
sq_ass_item	*ssizeobjargproc*	__setitem__ __delitem__
sq_contains	*objobjproc*	__contains__
sq_inplace_concat	*binaryfunc*	__iadd__
sq_inplace_repeat	*ssizeargfunc*	__imul__
bf_getbuffer	*getbufferproc()*	__buffer__
bf_releasebuffer	*releasebufferproc()*	__release_buffer__

槽位 typedef

typedef	参数类型	返回类型
allocfunc	*PyTypeObject* * *Py_ssize_t*	*PyObject* *
destructor	*PyObject* *	void
freefunc	void *	void
traverseproc	*PyObject* * *visitproc* void *	int
newfunc	*PyTypeObject* * *PyObject* * *PyObject* *	*PyObject* *
initproc	*PyObject* * *PyObject* * *PyObject* *	int
reprfunc	*PyObject* *	*PyObject* *
getattrfunc	*PyObject* * const char *	*PyObject* *
setattrfunc	*PyObject* * const char * *PyObject* *	int
getattrofunc	*PyObject* * *PyObject* *	*PyObject* *
setattrofunc	*PyObject* * *PyObject* * *PyObject* *	int
descrgetfunc	*PyObject* * *PyObject* * *PyObject* *	*PyObject* *
descrsetfunc	*PyObject* * *PyObject* * *PyObject* *	int
hashfunc	*PyObject* *	Py_hash_t
richcmpfunc		*PyObject* *

请参阅槽位类型 *typedef* 里有更多详细信息。

12.3.2 PyTypeObject 定义

针对*PyTypeObject* 的结构定义可以在 Include/cpython/object.h 中找到。为了方便参考，这里复述了其中的定义：

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* 用于打印，格式为 "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* 用于分配 */

    /* 用于实现标准操作的方法 */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* 原名为 tp_compare (Python 2)
                                    或 tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* 用于标准类的方法集 */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* 更多标准操作（这些用于二进制兼容） */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* 用于以输入/输出缓冲区方式访问对象的函数 */
    PyBufferProcs *tp_as_buffer;

    /* 用于定义可选/扩展特性是否存在的旗标 */
    unsigned long tp_flags;

    const char *tp_doc; /* 文档字符串 */

    /* 在 2.0 发布版中分配的含义 */
    /* 为所有可访问的对象调用函数 */
    traverseproc tp_traverse;

    /* 删除对所包含对象的引用 */
    inquiry tp_clear;

    /* 在 2.1 发布版中分配的含义 */
    /* 富比较操作 */
    richcmpfunc tp_richcompare;

    /* 弱引用的启用 */
    Py_ssize_t tp_weaklistoffset;

    /* 迭代器 */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;
```

(续下页)

(接上页)

```
    /* 属性描述器和子类化内容 */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    // 堆类型的强引用，静态类型的借入引用
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* 低层级的释放内存例程 */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* 方法解析顺序 */
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    /* 类型属性缓存版本标签。 在 2.6 版中添加 */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;

    /* 类型监视器针对此类型的位设置 */
    unsigned char tp_watched;
} PyTypeObject;
```

12.3.3 PyObject 槽位

类型对象结构体扩展了*PyVarObject* 结构体。*ob_size* 字段用于动态类型（由 `type_new()` 创建，通常由 class 语句调用）。请注意*PyType_Type*（元类型）会初始化*tp_itemsize*，这意味着它的实例（即类型对象）必须具有*ob_size* 字段。

Py_ssize_t *PyObject*.**ob_refcnt**

属于稳定 ABI. 这是类型对象的引用计数，由 `PyObject_HEAD_INIT` 宏初始化为 `1`。请注意对于静态分配的类型对象，类型的实例（其*ob_type* 指向该类型的对象）不会被计入引用。但对于动态分配的类型对象，实例会被计入引用。

继承：

子类型不继承此字段。

PyTypeObject **PyObject*.**ob_type**

属于稳定 ABI. 这是类型的类型，换句话说就是元类型，它由宏 `PyObject_HEAD_INIT` 的参数来做初始化，它的值一般情况下是 `&PyType_Type`。可是为了使动态可载入扩展模块至少在 Windows 上可用，编译器会报错这是一个不可用的初始化。因此按照惯例传递 `NULL` 给宏 `PyObject_HEAD_INIT` 并且在模块的初始化函数开始时候其他任何操作之前初始化这个字段。典型做法是这样的：

```
Foo_Type.ob_type = &PyType_Type;
```

这应当在创建类型的任何实例之前完成。*PyType_Ready()* 会检查*ob_type* 是否为 `NULL`，如果是，则将其初始化为基类的*ob_type* 字段。如果该字段为非零值则*PyType_Ready()* 将不会更改它。

继承：

此字段会被子类型继承。

12.3.4 PyVarObject 槽位

Py_ssize_t *PyVarObject*.**ob_size**

属于稳定 ABI. 对于静态分配的内存对象，它应该初始化为 0。对于动态分配的类型对象，该字段具有特殊的内部含义。

该字段应当使用*Py_SIZE()* 和*Py_SET_SIZE()* 宏来访问。

继承：

子类型不继承此字段。

12.3.5 PyTypeObject 槽

每个槽位都有一个小节来描述继承关系。如果*PyType_Ready()* 可以在字段被设为 NULL 时设置一个值那么还会有一个“默认”小节。（请注意在*PyBaseObject_Type* 和*PyType_Type* 上设置的许多字段实际上就是默认值。）

const char **PyTypeObject*.**tp_name**

指向包含类型名称的以 NUL 结尾的字符串的指针。对于可作为模块全局访问的类型，该字符串应为模块全名，后面跟一个点号，然后再加类型名称；对于内置类型，它应当只是类型名称。如果模块是包的子模块，则包的全名将是模块的全名的一部分。例如，在包 P 的子包 Q 中的模块 M 中定义的名为 T 的类型应当具有*tp_name* 初始化器 "P.Q.M.T"。

对于动态分配的类型对象，这应为类型名称，而模块名称将作为 '__module__' 键的值显式地保存在类型字典中。

对于静态分配的类型对象，*tp_name* 字段应当包含一个点号。最后一个点号之前的所有内容都可作为 __module__ 属性访问，而最后一个点号之后的所有内容都可作为 __name__ 属性访问。

如果不存在点号，则整个*tp_name* 字段将作为 __name__ 属性访问，而 __module__ 属性则将是未定义的（除非在字典中显式地设置，如上文所述）。这意味着无法对你的类型执行 pickle。此外，它也不会在用 pydoc 创建的模块文档中列出。

该字段不可为 NULL。它是*PyTypeObject()* 中唯一的必填字段（除了潜在的*tp_itemsize* 以外）。

继承：

子类型不继承此字段。

Py_ssize_t *PyTypeObject*.**tp_basicsize**

Py_ssize_t *PyTypeObject*.**tp_itemsize**

通过这些字段可以计算出该类型实例以字节为单位的大小。

类型可分为两种：实例为固定长度且 tp_itemsize 字段值为零的类型；实例为可变长度且 tp_itemsize 字段值不为零的类型。对于实例为固定长度的类型，所有实例都具有相同的由 tp_basicsize 给出的大小。（这条规则的例外情况可通过使用*PyUnstable_Object_GC_NewWithExtraData()* 来实现。）

对于实例为可变长度的类型，其实例必须具有*ob_size* 字段，且实例大小为 tp_basicsize 加 N 乘以 tp_itemsize，其中 N 为对象的“长度”。

像*PyObject_NewVar()* 这样的函数接受 N 值作为参数，并会将其保存在实例的*ob_size* 字段中。请注意*ob_size* 字段在此之后可能还有其他用处。例如，int 实例会以具体实现所定义的方式来使用*ob_size* 的比特位；下层的存储及其大小应当使用 PyLong_Export() 来访问。

> **备注**
>
> *ob_size* 字段应当使用*Py_SIZE()* 和*Py_SET_SIZE()* 宏来访问。

此外，实例布局中存在*ob_size* 字段并不意味着该实例结构体是可变长度的。例如，list 类型实例即为固定长度，虽然其实例具有*ob_size* 字段。（和 int 一样，请避免直接读取 list 的 ob_size。要改为调用*PyList_Size()* 函数。）

tp_basicsize 包括类型的*tp_base* 所需数据大小，加上每个实例所需额外数据的大小。

设置 tp_basicsize 的正确方式是在被用于声明实例布局的结构体上使用 sizeof 运算符。这个结构体必须包括被用于声明基类型的结构体。换句话说，tp_basicsize 必须大于等于基类型的 tp_basicsize。

由于任何类型都是 object 的子类型，这个结构体必须包括*PyObject* 或*PyVarObject* (具体取决于*ob_size* 是否应当被包括)。它们通常是分别由*PyObject_HEAD* 或*PyObject_VAR_HEAD* 宏来定义的。

基础大小不包括 GC 标头大小，因为该标头不是*PyObject_HEAD* 的一部分。

对于用于声明基类型的结构体位置未知的情况，请参见*PyType_Spec.basicsize* 和*PyType_FromMetaclass()*。

有关对齐的说明：

- tp_basicsize 必须是 _Alignof(PyObject) 的位数。当如建议的那样在包括了*PyObject_HEAD* 的 struct 上使用 sizeof 时，编译器会确保这一点。当没有使用 C struct，或者当使用像 __attribute__((packed)) 这样的编译器扩展时，这将是你的责任。
- 如果可变条目需要特定的对齐，则 tp_basicsize 和 tp_itemsize 必须均为该对齐值的倍数。举例来说，如果一个类型的可变部分存储了一个 double，你就要负责让这两个字段都是 _Alignof(double) 的倍数。

继承：

这些字段是由子类型分别来继承的。(也就是说，如果字段被设为零，则*PyType_Ready()* 将拷贝来自基类型的值，这表示实例不需要额外的存储。)

如果基类型有一个非零的*tp_itemsize*，那么在子类型中将*tp_itemsize* 设置为不同的非零值通常是不安全的（不过这取决于该基类型的具体实现)。

destructor *PyTypeObject*.**tp_dealloc**

指向实例析构函数的指针。除非保证类型的实例永远不会被释放（就像单例对象 None 和 Ellipsis 那样)，否则必须定义这个函数。函数声明如下：

```
void tp_dealloc(PyObject *self);
```

当新引用计数为零时，*Py_DECREF()* 和*Py_XDECREF()* 宏会调用析构函数。此时，实例仍然存在，但已没有了对它的引用。析构函数应当释放该实例拥有的所有引用，释放实例拥有的所有内存缓冲区（使用与分配缓冲区时所用分配函数相对应的释放函数)，并调用类型的*tp_free* 函数。如果该类型不可子类型化（未设置*Py_TPFLAGS_BASETYPE* 旗标位)，则允许直接调用对象的释放器而不必通过*tp_free*。对象的释放器应为分配实例时所使用的释放器；如果实例是使用*PyObject_New* 或*PyObject_NewVar* 分配的，则释放器通常为*PyObject_Del()*；如果实例是使用*PyObject_GC_New* 或*PyObject_GC_NewVar* 分配的，则释放器通常为*PyObject_GC_Del()*。

如果该类型支持垃圾回收（设置了*Py_TPFLAGS_HAVE_GC* 旗标位)，则析构器应在清除任何成员字段之前调用*PyObject_GC_UnTrack()*。

```
static void foo_dealloc(foo_object *self) {
    PyObject_GC_UnTrack(self);
    Py_CLEAR(self->ref);
    Py_TYPE(self)->tp_free((PyObject *)self);
}
```

最后，如果该类型是堆分配的 (*Py_TPFLAGS_HEAPTYPE*)，则在调用类型释放器后，释放器应释放对其类型对象的所有引用 (通过*Py_DECREF()*)。为了避免悬空指针，建议的实现方式如下：

```
static void foo_dealloc(foo_object *self) {
    PyTypeObject *tp = Py_TYPE(self);
    // free references and buffers here
    tp->tp_free(self);
    Py_DECREF(tp);
}
```

> **警告**
>
> 在应用垃圾回收机制的 Python 中，`tp_dealloc` 可以从任意 Python 线程被调用，而不仅是创建该对象的线程（如果该对象成为引用计数循环的一部分，则该循环可能会被任何线程上的垃圾回收操作所回收）。这对 Python API 调用来说不是问题，因为 `tp_dealloc` 调用所在的线程将持有全局解释器锁（GIL）。但是，如果被销毁的对象又销毁了来自其他 C 或 C++ 库的对象，则应当小心地确保在调用 `tp_dealloc` 的线程上销毁这些对象不会破坏这个库的任何资源。

继承：

此字段会被子类型继承。

Py_ssize_t *PyTypeObject*.**tp_vectorcall_offset**

一个相对使用*vectorcall 协议* 实现调用对象的实例级函数的可选的偏移量，这是一种比简单的*tp_call* 更有效的替代选择。

该字段仅在设置了*Py_TPFLAGS_HAVE_VECTORCALL* 旗标时使用。在此情况下，它必须为一个包含*vectorcallfunc* 指针实例中的偏移量的正整数。

vectorcallfunc 指针可能为 `NULL`，在这种情况下实例的行为就像*Py_TPFLAGS_HAVE_VECTORCALL* 没有被设置一样：调用实例操作会回退至*tp_call*。

任何设置了 `Py_TPFLAGS_HAVE_VECTORCALL` 的类也必须设置*tp_call* 并确保其行为与 *vectorcallfunc* 函数一致。这可以通过将 *tp_call* 设为*PyVectorcall_Call()* 来实现。

在 3.8 版本发生变更: 在 3.8 版之前，这个槽位被命名为 `tp_print`。在 Python 2.x 中，它被用于打印到文件。在 Python 3.0 至 3.7 中，它没有被使用。

在 3.12 版本发生变更: 在 3.12 版之前，不推荐可变堆类型 实现 vectorcall 协议。当用户在 Python 代码中设置 `__call__` 时，只有 *tp_call* 会被更新，很可能使它与 vectorcall 函数不一致。自 3.12 起，设置 `__call__` 将通过清除*Py_TPFLAGS_HAVE_VECTORCALL* 旗标来禁用 vectorcall 优化。

继承：

该字段总是会被继承。但是，*Py_TPFLAGS_HAVE_VECTORCALL* 旗标并不总是会被继承。如果它未被设置，则子类不会使用*vectorcall*，除非显式地调用了*PyVectorcall_Call()*。

getattrfunc *PyTypeObject*.**tp_getattr**

一个指向获取属性字符串函数的可选指针。

该字段已弃用。当它被定义时，应该和*tp_getattro* 指向同一个函数，但接受一个 C 字符串参数表示属性名，而不是 Python 字符串对象。

继承：

分组: *tp_getattr*, *tp_getattro*

该字段会被子类和*tp_getattro* 所继承：当子类型的*tp_getattr* 和*tp_getattro* 均为 `NULL` 时该子类型将从它的基类型同时继承*tp_getattr* 和*tp_getattro*。

setattrfunc *PyTypeObject*.**tp_setattr**

一个指向函数以便设置和删除属性的可选指针。

该字段已弃用。当它被定义时，应该和*tp_setattro* 指向同一个函数，但接受一个 C 字符串参数表示属性名，而不是 Python 字符串对象。

继承：

分组: *tp_setattr*, *tp_setattro*

该字段会被子类型和*tp_setattro* 所继承：当子类型的*tp_setattr* 和*tp_setattro* 均为 `NULL` 时该子类型将同时从它的基类型继承*tp_setattr* 和*tp_setattro*。

PyAsyncMethods **PyTypeObject.tp_as_async*

指向一个包含仅与在 C 层级上实现*awaitable* 和*asynchronous iterator* 协议的对象相关联的字段的附加结构体。请参阅异步对象结构体 了解详情。

Added in version 3.5: 在之前被称为 `tp_compare` 和 `tp_reserved`。

继承：

tp_as_async 字段不会被继承，但所包含的字段会被单独继承。

reprfunc *PyTypeObject*.**tp_repr**

一个实现了内置函数 `repr()` 的函数的可选指针。

该签名与*PyObject_Repr()* 的相同:

```
PyObject *tp_repr(PyObject *self);
```

该函数必须返回一个字符串或 Unicode 对象。在理想情况下，该函数应当返回一个字符串，当将其传给 `eval()` 时，只要有合适的环境，就会返回一个具有相同值的对象。如果这不可行，则它应当返回一个以 `'<'` 开头并以 `'>'` 结尾的可被用来推断出对象的类型和值的字符串。

继承：

此字段会被子类型继承。

默认：

如果未设置该字段，则返回 `<%s object at %p>` 形式的字符串，其中 `%s` 将替换为类型名称，`%p` 将替换为对象的内存地址。

PyNumberMethods **PyTypeObject.tp_as_number*

指向一个附加结构体的指针，其中包含只与执行数字协议的对象相关的字段。这些字段的文档参见数字对象结构体。

继承：

tp_as_number 字段不会被继承，但所包含的字段会被单独继承。

PySequenceMethods **PyTypeObject.tp_as_sequence*

指向一个附加结构体的指针，其中包含只与执行序列协议的对象相关的字段。这些字段的文档见序列对象结构体。

继承：

tp_as_sequence 字段不会被继承，但所包含的字段会被单独继承。

PyMappingMethods **PyTypeObject.tp_as_mapping*

指向一个附加结构体的指针，其中包含只与执行映射协议的对象相关的字段。这些字段的文档见映射对象结构体。

继承：

tp_as_mapping 字段不会继承，但所包含的字段会被单独继承。

hashfunc *PyTypeObject*.**tp_hash**

一个指向实现了内置函数 `hash()` 的函数的可选指针。

其签名与*PyObject_Hash()* 的相同:

```
Py_hash_t tp_hash(PyObject *);
```

`-1` 不应作为正常返回值被返回；当计算哈希值过程中发生错误时，函数应设置一个异常并返回 `-1`。

当该字段（和*tp_richcompare*）都未设置，尝试对该对象取哈希会引发 `TypeError`。这与将其设为*PyObject_HashNotImplemented()* 相同。

此字段可被显式设为*PyObject_HashNotImplemented()* 以阻止从父类型继承哈希方法。在 Python 层面这被解释为 `__hash__ = None` 的等价物，使得 `isinstance(o, collections.Hashable)` 正

确返回 False.。请注意反过来也是如此：在 Python 层面设置一个类的 __hash__ = None 会使得 tp_hash 槽位被设置为*PyObject_HashNotImplemented()*。

继承：

分组: *tp_hash*, *tp_richcompare*

该字段会被子类型同*tp_richcompare* 一起继承：当子类型的*tp_richcompare* 和*tp_hash* 均为 NULL 时子类型将同时继承*tp_richcompare* 和*tp_hash*。

默认：

PyBaseObject_Type 使用*PyObject_GenericHash()*。

ternaryfunc *PyTypeObject*.**tp_call**

一个可选的实现对象调用的指向函数的指针。如果对象不是可调用对象则该值应为 NULL。其签名与*PyObject_Call()* 的相同:

```
PyObject *tp_call(PyObject *self, PyObject *args, PyObject *kwargs);
```

继承：

此字段会被子类型继承。

reprfunc *PyTypeObject*.**tp_str**

一个可选的实现内置 str() 操作的函数的指针。(请注意 str 现在是一个类型，str() 是调用该类型的构造器。该构造器将调用*PyObject_Str()* 执行实际操作，而*PyObject_Str()* 将调用该处理器。)

其签名与*PyObject_Str()* 的相同:

```
PyObject *tp_str(PyObject *self);
```

该函数必须返回一个字符串或 Unicode 对象。它应当是一个“友好”的对象字符串表示形式，因为这就是要在 print() 函数中与其他内容一起使用的表示形式。

继承：

此字段会被子类型继承。

默认：

当未设置该字段时，将调用*PyObject_Repr()* 来返回一个字符串表示形式。

getattrofunc *PyTypeObject*.**tp_getattro**

一个指向获取属性字符串函数的可选指针。

其签名与*PyObject_GetAttr()* 的相同:

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

可以方便地将该字段设为*PyObject_GenericGetAttr()*，它实现了查找对象属性的通常方式。

继承：

分组: *tp_getattr*, *tp_getattro*

该字段会被子类同*tp_getattr* 一起继承：当子类型的*tp_getattr* 和*tp_getattro* 均为 NULL 时子类型将同时继承*tp_getattr* 和*tp_getattro*。

默认：

PyBaseObject_Type 使用*PyObject_GenericGetAttr()*。

setattrofunc *PyTypeObject*.**tp_setattro**

一个指向函数以便设置和删除属性的可选指针。

其签名与*PyObject_SetAttr()* 的相同:

```
int tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

此外，还必须支持将 *value* 设为 NULL 来删除属性。通常可以方便地将该字段设为*PyObject_GenericSetAttr()*，它实现了设备对象属性的通常方式。

继承：

分组: *tp_setattr*, *tp_setattro*

该字段会被子类型同*tp_setattr* 一起继承：当子类型的*tp_setattr* 和*tp_setattro* 均为 NULL 时子类型将同时继承*tp_setattr* 和*tp_setattro*。

默认：

PyBaseObject_Type 使用*PyObject_GenericSetAttr()*。

PyBufferProcs **PyTypeObject*.**tp_as_buffer**

指向一个包含只与实现缓冲区接口的对象相关的字段的附加结构体的指针。这些字段的文档参见缓冲区对象结构体。

继承：

tp_as_buffer 字段不会被继承，但所包含的字段会被单独继承。

unsigned long *PyTypeObject*.**tp_flags**

该字段是针对多个旗标的位掩码。某些旗标指明用于特定场景的变化语义；另一些旗标则用于指明类型对象（或通过*tp_as_number*, *tp_as_sequence*, *tp_as_mapping* 和*tp_as_buffer* 引用的扩展结构体）中的特定字段，它们在历史上并不总是有效；如果这样的旗标位是清晰的，则它所保护的类型字段必须不可被访问并且必须被视为具有零或 NULL 值。

继承：

这个字段的继承很复杂。大多数旗标位都是单独继承的，也就是说，如果基类型设置了一个旗标位，则子类型将继承该旗标位。从属于扩展结构体的旗标位仅在扩展结构体被继承时才会被继承，也就是说，基类型的旗标位值会与指向扩展结构体的指针一起被拷贝到子类型中。*Py_TPFLAGS_HAVE_GC* 旗标位会与*tp_traverse* 和*tp_clear* 字段一起被继承，也就是说，如果*Py_TPFLAGS_HAVE_GC* 旗标位在子类型中被清空并且子类型中的*tp_traverse* 和*tp_clear* 字段存在并具有 NULL 值。.. XXX 那么大多数旗标位 真的都是单独继承的吗?

默认：

PyBaseObject_Type 使用 Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE。

位掩码:

目前定义了以下位掩码；可以使用 | 运算符对它们进行 OR 运算以形成*tp_flags* 字段的值。宏*PyType_HasFeature()* 接受一个类型和一个旗标值 *tp* 和 *f*，并检查 tp->tp_flags & f 是否为非零值。

Py_TPFLAGS_HEAPTYPE

当类型对象本身在堆上被分配时会设置这个比特位，例如，使用*PyType_FromSpec()* 动态创建的类型。在此情况下，其实例的*ob_type* 字段会被视为指向该类型的引用，而类型对象将在一个新实例被创建时执行 INCREF，并在实例被销毁时执行 DECREF（这不会应用于子类型的实例；只有实例的 ob_type 所引用的类型会执行 INCREF 和 DECREF）。堆类型应当也支持垃圾回收 因为它们会形成对它们自己的模块对象的循环引用。

继承：

???

Py_TPFLAGS_BASETYPE

当此类型可被用作另一个类型的基类型时该比特位将被设置。如果该比特位被清除，则此类型将无法被子类型化（类似于 Java 中的"final" 类）。

继承：

???

Py_TPFLAGS_READY

当此类型对象通过*PyType_Ready()* 被完全实例化时该比特位将被设置。

继承：

???

Py_TPFLAGS_READYING

当*PyType_Ready()* 处在初始化此类型对象过程中时该比特位将被设置。

继承：

???

Py_TPFLAGS_HAVE_GC

当对象支持垃圾回收时会设置这个旗标位。如果设置了这个位，则实例必须使用*PyObject_GC_New* 来创建并使用*PyObject_GC_Del()* 来销毁。更多信息参见*使对象类型支持循环垃圾回收*。这个位还会假定类型对象中存在 GC 相关字段*tp_traverse* 和*tp_clear*。

继承：

分组: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

Py_TPFLAGS_HAVE_GC 旗标位会与*tp_traverse* 和*tp_clear* 字段一起被继承，也就是说，如果*Py_TPFLAGS_HAVE_GC* 旗标位在子类型中被清空并且子类型中的*tp_traverse* 和*tp_clear* 字段存在并具有 `NULL` 值的话。

Py_TPFLAGS_DEFAULT

这是一个从属于类型对象及其扩展结构体的存在的所有位的位掩码。目前，它包括以下的位: `Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`。

继承：

???

Py_TPFLAGS_METHOD_DESCRIPTOR

这个位指明对象的行为类似于未绑定方法。

如果为 `type(meth)` 设置了该旗标，那么:

- `meth.__get__(obj, cls)(*args, **kwds)` (其中 `obj` 不为 None) 必须等价于 `meth(obj, *args, **kwds)`。
- `meth.__get__(None, cls)(*args, **kwds)` 必须等价于 `meth(*args, **kwds)`。

此旗标为 `obj.meth()` 这样的典型方法调用启用优化: 它将避免为 `obj.meth` 创建临时的“绑定方法”对象。

Added in version 3.8.

继承：

此旗标绝不会被没有设置*Py_TPFLAGS_IMMUTABLETYPE* 旗标的类型所继承。对于扩展类型，当*tp_descr_get* 被继承时它也会被继承。

Py_TPFLAGS_MANAGED_DICT

该比特位指明类的实例具有 `__dict__` 属性，并且该字典的空间是由 VM 管理的。

如果设置了该旗标，则*Py_TPFLAGS_HAVE_GC* 也应当被设置。

类型遍历函数必须调用*PyObject_VisitManagedDict()* 而它的清空函数必须调用*PyObject_ClearManagedDict()*。

Added in version 3.12.

继承：

此旗标将被继承，除非某个超类设置了*tp_dictoffset* 字段。

Py_TPFLAGS_MANAGED_WEAKREF

该比特位表示类的实例应当是可被弱引用的。

Added in version 3.12.

继承：

此旗标将被继承，除非某个超类设置了*tp_weaklistoffset* 字段。

Py_TPFLAGS_ITEMS_AT_END

仅适用于可变大小的类型，也就是说，具有非零*tp_itemsize* 值的类型。

表示此类型的实例的可变大小部分位于该实例内存区的末尾，其偏移量为 `Py_TYPE(obj)->tp_basicsize`（每个子类可能不一样）。

当设置此旗标时，请确保所有子类要么使用此内存布局，要么不是可变大小。Python 不会检查这一点。

Added in version 3.12.

继承：

这个旗标会被继承。

Py_TPFLAGS_LONG_SUBCLASS

Py_TPFLAGS_LIST_SUBCLASS

Py_TPFLAGS_TUPLE_SUBCLASS

Py_TPFLAGS_BYTES_SUBCLASS

Py_TPFLAGS_UNICODE_SUBCLASS

Py_TPFLAGS_DICT_SUBCLASS

Py_TPFLAGS_BASE_EXC_SUBCLASS

Py_TPFLAGS_TYPE_SUBCLASS

这些旗标被*PyLong_Check()* 等函数用来快速确定一个类型是否为内置类型的子类；这样的专用检测比泛用检测如*PyObject_IsInstance()* 要更快速。继承自内置类型的自定义类型应当正确地设置其*tp_flags*，否则与这样的类型进行交互的代码将因所使用的检测种类而出现不同的行为。

Py_TPFLAGS_HAVE_FINALIZE

当类型结构体中存在*tp_finalize* 槽位时会设置这个比特位。

Added in version 3.4.

自 3.8 版本弃用：此旗标已不再是必要的，因为解释器会假定类型结构体中总是存在*tp_finalize* 槽位。

Py_TPFLAGS_HAVE_VECTORCALL

当类实现了*vectorcall 协议* 时会设置这个比特位。请参阅*tp_vectorcall_offset* 了解详情。

继承：

如果继承了*tp_call* 则也会继承这个比特位。

Added in version 3.9.

在 3.12 版本发生变更：现在当类的 `__call__()` 方法被重新赋值时该旗标将从类中移除。

现在该旗标能被可变类所继承。

Py_TPFLAGS_IMMUTABLETYPE

不可变的类型对象会设置这个比特位：类型属性无法被设置或删除。

PyType_Ready() 会自动对静态类型 应用这个旗标。

继承：

这个旗标不会被继承。

Added in version 3.10.

Py_TPFLAGS_DISALLOW_INSTANTIATION

不允许创建此类型的实例：将*tp_new* 设为 NULL 并且不会在类型字符中创建 `__new__` 键。

这个旗标必须在创建该类型之前设置，而不是在之后。例如，它必须在该类型调用*PyType_Ready()* 之前被设置。

如果*tp_base* 为 NULL 或者 `&PyBaseObject_Type` 和*tp_new* 为 NULL 则该旗标会在静态类型 上自动设置。

继承：

这个旗标不会被继承。但是，子类将不能被实例化，除非它们提供了不为 NULL 的*tp_new* (这只能通过 C API 实现)。

> **备注**
>
> 要禁止直接实例化一个类但允许实例化其子类 (例如对于*abstract base class*)，请勿使用此旗标。替代的做法是，让*tp_new* 只对子类可用。

Added in version 3.10.

Py_TPFLAGS_MAPPING

这个比特位指明该类的实例可以在被用作 `match` 代码块的目标时匹配映射模式。它会在注册或子类化 `collections.abc.Mapping` 时自动设置，并在注册 `collections.abc.Sequence` 时取消设置。

> **备注**
>
> *Py_TPFLAGS_MAPPING* 和*Py_TPFLAGS_SEQUENCE* 是互斥的；同时启用两个旗标将导致报错。

继承：

这个旗标将被尚未设置*Py_TPFLAGS_SEQUENCE* 的类型所继承。

> **参见**
>
> **PEP 634** ——结构化模式匹配：规范

Added in version 3.10.

Py_TPFLAGS_SEQUENCE

这个比特位指明该类的实例可以在被用作 `match` 代码块的目标时匹配序列模式。它会在注册或子类化 `collections.abc.Sequence` 时自动设置，并在注册 `collections.abc.Mapping` 时取消设置。

> **备注**
>
> *Py_TPFLAGS_MAPPING* 和*Py_TPFLAGS_SEQUENCE* 是互斥的；同时启用两个旗标将导致报错。

继承：

这个旗标将被尚未设置*Py_TPFLAGS_MAPPING* 的类型所继承。

> **参见**
>
> **PEP 634** ——结构化模式匹配：规范

Added in version 3.10.

Py_TPFLAGS_VALID_VERSION_TAG

内部使用。请不要设置或取消设置此旗标。用于指明一个类具有被修改的调用*PyType_Modified()*

> **警告**
>
> 此旗标存在于头文件中，但未被使用。它将在未来某个 CPython 版本中被移除。

const char **PyTypeObject*.**tp_doc**

一个可选的指向给出该类型对象的文档字符串的以 NUL 结束的 C 字符串的指针。该指针被暴露为类型和类型实例上的 `__doc__` 属性。

继承：

这个字段 不会被子类型继承。

traverseproc *PyTypeObject*.**tp_traverse**

一个可选的指向针对垃圾回收器的遍历函数的指针。该指针仅会在设置了*Py_TPFLAGS_HAVE_GC* 旗标位时被使用。函数签名为:

```
int tp_traverse(PyObject *self, visitproc visit, void *arg);
```

有关 Python 垃圾回收方案的更多信息可在*使对象类型支持循环垃圾回收* 一节中查看。

tp_traverse 指针被垃圾回收器用来检测循环引用。*tp_traverse* 函数的典型实现会在实例的每个属于该实例所拥有的 Python 对象的成员上简单地调用*Py_VISIT()*。例如，以下是来自 `_thread` 扩展模块的函数 `local_traverse()`:

```
static int
local_traverse(localobject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

请注意*Py_VISIT()* 仅能在可以参加循环引用的成员上被调用。虽然还存在一个 `self->key` 成员，但它只能为 `NULL` 或 Python 字符串因而不能成为循环引用的一部分。

在另一方面，即使你知道某个成员永远不会成为循环引用的一部分，作为调试的辅助你仍然可能想要访问它因此 `gc` 模块的 `get_referents()` 函数将会包括它。

堆类型 (*Py_TPFLAGS_HEAPTYPE*) 必须这样访问其类型:

```
Py_VISIT(Py_TYPE(self));
```

它只是从 Python 3.9 开始才需要。为支持 Python 3.8 和更旧的版本，这一行必须是有条件的:

```
#if PY_VERSION_HEX >= 0x03090000
    Py_VISIT(Py_TYPE(self));
#endif
```

如果在*tp_flags* 字段中设置了*Py_TPFLAGS_MANAGED_DICT* 比特位，则遍历函数必须这样调用*PyObject_VisitManagedDict()*:

```
PyObject_VisitManagedDict((PyObject*)self, visit, arg);
```

警告

当实现*tp_traverse* 时，只有实例所 拥有的成员 (就是有指向它们的强引用) 才必须被访问。举例来说，如果一个对象通过*tp_weaklist* 槽位支持弱引用，那么支持链表 (*tp_weaklist* 所指向的对象) 的指针就 **不能**被访问因为实例并不直接拥有指向自身的弱引用 (弱引用列表被用来支持弱引用机制，但实例没有指向其中的元素的强引用，因为即使实例还存在它们也允许被删除)。

请注意*Py_VISIT()* 要求传给 `local_traverse()` 的 *visit* 和 *arg* 形参具有指定的名称；不要随意命名它们。

堆分配类型 的实例会持有一个指向其类型的引用。因此它们的遍历函数必须要么访问*Py_TYPE(self)*，要么通过调用其他堆分配类型（例如一个堆分配超类）的 `tp_traverse` 将此任务委托出去。如果没有这样做，类型对象可能不会被垃圾回收。

在 3.9 版本发生变更: 堆分配类型应当访问 `tp_traverse` 中的 `Py_TYPE(self)`。在较早的 Python 版本中，由于 bug 40217，这样做可能会导致在超类中发生崩溃。

继承:

分组: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

该字段会与*tp_clear* 和*Py_TPFLAGS_HAVE_GC* 旗标位一起被子类型所继承: 如果旗标位, *tp_traverse* 和*tp_clear* 在子类型中均为零则它们都将从基类型继承。

inquiry *PyTypeObject*.**tp_clear**

一个可选的指向针对垃圾回收器的清理函数的指针。该指针仅会在设置了*Py_TPFLAGS_HAVE_GC* 旗标位时被使用。函数签名为:

```
int tp_clear(PyObject *);
```

tp_clear 成员函数被用来打破垃圾回收器在循环垃圾中检测到的循环引用。总的来说，系统中的所有*tp_clear* 函数必须合到一起以打破所有引用循环。这是个微妙的问题，并且如有任何疑问都需要提供*tp_clear* 函数。例如，元组类型不会实现*tp_clear* 函数，因为有可能证明完全用元组是不会构成循环引用的。因此其他类型的*tp_clear* 函数必须足以打破任何包含元组的循环。这不是立即能明确的，并且很少会有避免实现*tp_clear* 的适当理由。

tp_clear 的实现应当丢弃实例指向其成员的可能为 Python 对象的引用，并将指向这些成员的指针设为 `NULL`，如下面的例子所示:

```
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

应当使用*Py_CLEAR()* 宏，因为清除引用是很微妙的：指向被包含对象的引用必须在指向被包含对象的指针被设为 NULL 之后才能被释放 (通过*Py_DECREF()*)。这是因为释放引用可能会导致被包含的对象变成垃圾，触发一连串的回收活动，其中可能包括唤起任意 Python 代码 (由于关联到被包含对象的终结器或弱引用回调)。如果这样的代码有可能再次引用 *self*，那么这时指向被包含对象的指针为 NULL 就是非常重要的，这样 *self* 就知道被包含对象不可再被使用。*Py_CLEAR()* 宏将以安全的顺序执行此操作。

如果在*tp_flags* 字段中设置了*Py_TPFLAGS_MANAGED_DICT* 比特位，则遍历函数必须这样调用*PyObject_ClearManagedDict()*：

```
PyObject_ClearManagedDict((PyObject*)self);
```

请注意*tp_clear* 并非总是在实例被取消分配之前被调用。例如，当引用计数足以确定对象不再被使用时，就不会涉及循环垃圾回收器而是直接调用*tp_dealloc*。

因为*tp_clear* 函数的目的是打破循环引用，所以不需要清除所包含的对象如 Python 字符串或 Python 整数，它们无法参与循环引用。另一方面，清除所包含的全部 Python 对象，并编写类型的*tp_dealloc* 函数来唤起*tp_clear* 也很方便。

有关 Python 垃圾回收方案的更多信息可在使对象类型支持循环垃圾回收 一节中查看。

继承：

分组: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

该字段会与*tp_traverse* 和*Py_TPFLAGS_HAVE_GC* 旗标位一起被子类型所继承：如果旗标位，*tp_traverse* 和*tp_clear* 在子类型中均为零则它们都将从基类型继承。

richcmpfunc *PyTypeObject*.**tp_richcompare**

一个可选的指向富比较函数的指针，函数的签名为:

```
PyObject *tp_richcompare(PyObject *self, PyObject *other, int op);
```

第一个形参将保证为*PyTypeObject* 所定义的类型的实例。

该函数应当返回比较的结果 (通常为 Py_True 或 Py_False)。如果未定义比较运算，它必须返回 Py_NotImplemented，如果发生了其他错误则它必须返回 NULL 并设置一个异常条件。

以下常量被定义用作*tp_richcompare* 和*PyObject_RichCompare()* 的第三个参数：

常量	对照
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

定义以下宏是为了简化编写丰富的比较函数：

Py_RETURN_RICHCOMPARE(VAL_A, VAL_B, op)

从该函数返回 Py_True 或 Py_False，这取决于比较的结果。VAL_A 和 VAL_B 必须是可通过 C 比较运算符进行排序的（例如，它们可以为 C 整数或浮点数）。第三个参数指明所请求的运算，与*PyObject_RichCompare()* 的参数一样。

返回值是一个新的*strong reference*。

发生错误时，将设置异常并从该函数返回 NULL。

Added in version 3.7.

继承：

分组: *tp_hash*, *tp_richcompare*

该字段会被子类型同*tp_hash* 一起继承：当子类型的*tp_richcompare* 和*tp_hash* 均为 NULL 时子类型将同时继承*tp_richcompare* 和*tp_hash*。

默认：

PyBaseObject_Type 提供了一个*tp_richcompare* 的实现，它可以被继承。但是，如果只定义了*tp_hash*，则不会使用被继承的函数并且该类型的实例将无法参加任何比较。

Py_ssize_t *PyTypeObject*.**tp_weaklistoffset**

虽然此字段仍然受到支持，但是如果可能就应当改用*Py_TPFLAGS_MANAGED_WEAKREF*。

如果此类型的实例是可被弱引用的，则该字段将大于零并包含在弱引用列表头的实例结构体中的偏移量（忽略 GC 头，如果存在的话）；该偏移量将被*PyObject_ClearWeakRefs()* 和 PyWeakref_* 函数使用。实例结构体需要包括一个 *PyObject** 类型的字段并初始化为 NULL。

不要将该字段与*tp_weaklist* 混淆；后者是指向类型对象本身的弱引用的列表头。

同时设置*Py_TPFLAGS_MANAGED_WEAKREF* 位和*tp_weaklistoffset* 将导致错误。

继承：

该字段会被子类型继承，但注意参阅下面列出的规则。子类型可以覆盖此偏移量；这意味着子类型将使用不同于基类型的弱引用列表。由于列表头总是通过*tp_weaklistoffset* 找到的，所以这应该不成问题。

默认：

如果在*tp_flags* 字段中设置了*Py_TPFLAGS_MANAGED_WEAKREF* 位，则*tp_weaklistoffset* 将被设为负值，用以表明使用此字段是不安全的。

getiterfunc *PyTypeObject*.**tp_iter**

一个可选的指向函数的指针，该函数返回对象的*iterator*。它的存在通常表明该类型的实例为*iterable* (尽管序列在没有此函数的情况下也可能为可迭代对象)。

此函数的签名与*PyObject_GetIter()* 的相同:

```
PyObject *tp_iter(PyObject *self);
```

继承：

此字段会被子类型继承。

iternextfunc *PyTypeObject*.**tp_iternext**

一个可选的指向函数的指针，该函数返回*iterator* 中的下一项。其签名为:

```
PyObject *tp_iternext(PyObject *self);
```

当该迭代器被耗尽时，它必须返回 NULL；StopIteration 异常可能会设置也可能不设置。当发生另一个错误时，它也必须返回 NULL。它的存在表明该类型的实际是迭代器。

迭代器类型也应当定义*tp_iter* 函数，并且该函数应当返回迭代器实例本身（而不是新的迭代器实例）。

此函数的签名与*PyIter_Next()* 的相同。

继承：

此字段会被子类型继承。

struct *PyMethodDef* **PyTypeObject.tp_methods*

一个可选的指向*PyMethodDef* 结构体的以 NULL 结束的静态数组的指针，它声明了此类型的常规方法。

对于该数组中的每一项，都会向类型的字典 (参见下面的*tp_dict*) 添加一个包含方法描述器的条目。

继承：

该字段不会被子类型所继承（方法是通过不同的机制来继承的）。

struct *PyMemberDef* **PyTypeObject.tp_members*

一个可选的指向*PyMemberDef* 结构体的以 NULL 结束的静态数组的指针，它声明了此类型的常规数据成员（字段或槽位）。

对于该数组中的每一项，都会向类型的字典 (参见下面的*tp_dict*) 添加一个包含方法描述器的条目。

继承：

该字段不会被子类型所继承（成员是通过不同的机制来继承的）。

struct *PyGetSetDef* **PyTypeObject.tp_getset*

一个可选的指向*PyGetSetDef* 结构体的以 NULL 结束的静态数组的指针，它声明了此类型的实例中的被计算属性。

对于该数组中的每一项，都会向类型的字典 (参见下面的*tp_dict*) 添加一个包含读写描述器的条目。

继承：

该字段不会被子类型所继承（被计算属性是通过不同的机制来继承的）。

PyTypeObject **PyTypeObject.tp_base*

一个可选的指向类型特征属性所继承的基类型的指针。在这个层级上，只支持单继承；多重继承需要通过调用元类型动态地创建类型对象。

> **备注**
>
> 槽位初始化需要遵循初始化全局变量的规则。C99 要求初始化器为“地址常量”。隐式转换为指针的函数指示器如*PyType_GenericNew()* 都是有效的 C99 地址常量。
>
> 但是，生成地址常量并不需要应用于非静态变量如*PyBaseObject_Type* 的单目运算符'&'。编译器可能支持该运算符（如 gcc），但 MSVC 则不支持。这两种编译器在这一特定行为上都是严格符合标准的。
>
> 因此，应当在扩展模块的初始化函数中设置*tp_base*。

继承：

该字段不会被子类型继承（显然）。

默认：

该字段默认为 `&PyBaseObject_Type` (对 Python 程序员来说即 `object` 类型)。

PyObject **PyTypeObject.tp_dict*

类型的字典将由*PyType_Ready()* 存储到这里。

该字段通常应当在 PyType_Ready 被调用之前初始化为 NULL；它也可以初始化为一个包含类型初始属性的字典。一旦*PyType_Ready()* 完成类型的初始化，该类型的额外属性只有在它们不与被重载

的操作 (如 `__add__()`) 相对应的情况下才会被添加到该字典中。一旦类型的初始化结束，该字段就应被视为是只读的。

某些类型不会将它们的字典存储在该槽位中。请使用*PyType_GetDict()* 来获取任意类型对应的字典。

在 3.12 版本发生变更: 内部细节：对于静态内置类型，该值总是为 `NULL`。这种类型的字典是存储在 `PyInterpreterState` 中。请使用*PyType_GetDict()* 来获取任意类型的字典。

继承：

该字段不会被子类型所继承（但在这里定义的属性是通过不同的机制来继承的）。

默认：

如果该字段为 `NULL`，*PyType_Ready()* 将为它分配一个新字典。

> **警告**
>
> 通过字典 C-API 使用*PyDict_SetItem()* 或修改*tp_dict* 是不安全的。

descrgetfunc *PyTypeObject*.**tp_descr_get**

一个可选的指向“描述器获取”函数的指针。

函数的签名为:

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

继承：

此字段会被子类型继承。

descrsetfunc *PyTypeObject*.**tp_descr_set**

一个指向用于设置和删除描述器值的函数的选项指针。

函数的签名为:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

将 *value* 参数设为 `NULL` 以删除该值。

继承：

此字段会被子类型继承。

Py_ssize_t *PyTypeObject*.**tp_dictoffset**

虽然此字段仍然受到支持，但是如果可能就应当改用*Py_TPFLAGS_MANAGED_DICT*。

如果该类型的实例具有一个包含实例变量的字典，则此字段将为非零值并包含该实例变量字典的类型的实例的偏移量；该偏移量将由*PyObject_GenericGetAttr()* 使用。

不要将该字段与*tp_dict* 混淆；后者是由类型对象本身的属性组成的字典。

该值指定字典相对实例结构体开始位置的偏移量。

tp_dictoffset 应当被视为是只读的。用于获取指向字典调用*PyObject_GenericGetDict()* 的指针。调用*PyObject_GenericGetDict()* 可能需要为字典分配内存，因此在访问对象上的属性时调用*PyObject_GetAttr()* 可能会更有效率。

同时设置*Py_TPFLAGS_MANAGED_DICT* 位和*tp_dictoffset* 将导致报错。

继承：

该字段会被子类型所继承。子类型不应重写这个偏移量；这样做是不安全的，如果 C 代码试图在之前的偏移量上访问字典的话。要正确地支持继承，请使用*Py_TPFLAGS_MANAGED_DICT*。

默认：

这个槽位没有默认值。对于静态类型，如果该字段为 NULL 则不会为实例创建 __dict__。

如果在*tp_flags* 字段中设置了*Py_TPFLAGS_MANAGED_DICT* 比特位，则*tp_dictoffset* 将被设为 -1，以表明使用该字段是不安全的。

initproc *PyTypeObject*.**tp_init**

一个可选的指向实例初始化函数的指针。

此函数对应于类的 __init__() 方法。和 __init__() 一样，创建实例时不调用 __init__() 是有可能的，并且通过再次调用实例的 __init__() 方法将其重新初始化也是有可能的。

函数的签名为:

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds);
```

self 参数是将要初始化的实例；*args* 和 *kwds* 参数代表调用 __init__() 时传入的位置和关键字参数。

tp_init 函数如果不为 NULL，将在通过调用类型正常创建其实例时被调用，即在类型的*tp_new* 函数返回一个该类型的实例时。如果*tp_new* 函数返回了一个不是原始类型的子类型的其他类型的实例，则*tp_init* 函数不会被调用；如果*tp_new* 返回了一个原始类型的子类型的实例，则该子类型的*tp_init* 将被调用。

成功时返回 0，发生错误时则返回 -1 并在错误上设置一个异常。and sets an exception on error.

继承：

此字段会被子类型继承。

默认：

对于静态类型 来说该字段没有默认值。

allocfunc *PyTypeObject*.**tp_alloc**

指向一个实例分配函数的可选指针。

函数的签名为:

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

继承：

该字段会被静态子类型继承，但不会被动态子类型（通过 class 语句创建的子类型）继承。

默认：

对于动态子类型，该字段总是会被设为*PyType_GenericAlloc()*，以强制应用标准的堆分配策略。

对于静态子类型，*PyBaseObject_Type* 将使用*PyType_GenericAlloc()*。这是适用于所有静态定义类型的推荐值。

newfunc *PyTypeObject*.**tp_new**

一个可选的指向实例创建函数的指针。

函数的签名为:

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds);
```

subtype 参数是被创建的对象的类型；*args* 和 *kwds* 参数表示调用类型时传入的位置和关键字参数。请注意 *subtype* 不是必须与被调用的*tp_new* 函数所属的类型相同；它可以是该类型的子类型（但不能是完全无关的类型）。

tp_new 函数应当调用 subtype->tp_alloc(subtype, nitems) 来为对象分配空间，然后只执行绝对有必要的进一步初始化操作。可以安全地忽略或重复的初始化操作应当放在*tp_init* 处理器中。一个关键的规则是对于不可变类型来说，所有初始化操作都应当在*tp_new* 中发生，而对于可变类型，大部分初始化操作都应当推迟到*tp_init* 再执行。

设置*Py_TPFLAGS_DISALLOW_INSTANTIATION* 旗标以禁止在 Python 中创建该类型的实例。

继承：

该字段会被子类型所继承，例外情况是它不会被*tp_base* 为 NULL 或 &PyBaseObject_Type 的静态类型 所继承。

默认：

对于静态类型 该字段没有默认值。这意味着如果槽位被定义为 NULL，则无法调用此类型来创建新的实例；应当存在其他办法来创建实例，例如工厂函数等。

freefunc *PyTypeObject*.**tp_free**

一个可选的指向实例释放函数的指针。函数的签名为:

```
void tp_free(void *self);
```

一个兼容该签名的初始化器是*PyObject_Free()*。

继承：

该字段会被静态子类型继承，但不会被动态子类型（通过 class 语句创建的子类型）继承

默认：

在动态子类型中，该字段会被设为一个适合与*PyType_GenericAlloc()* 以及*Py_TPFLAGS_HAVE_GC* 旗标位的值相匹配的释放器。

对于静态子类型，*PyBaseObject_Type* 将使用*PyObject_Del()*。

inquiry *PyTypeObject*.**tp_is_gc**

可选的指向垃圾回收器所调用的函数的指针。

垃圾回收器需要知道某个特定的对象是否可以被回收。在一般情况下，垃圾回收器只需要检查这个对象类型的*tp_flags* 字段、以及*Py_TPFLAGS_HAVE_GC* 标识位即可做出判断；但是有一些类型同时混合包含了静态和动态分配的实例，其中静态分配的实例不应该也无法被回收。本函数为后者情况而设计：对于可被垃圾回收的实例，本函数应当返回 1；对于不可被垃圾回收的实例，本函数应当返回 0。函数的签名为:

```
int tp_is_gc(PyObject *self);
```

（此对象的唯一样例是类型本身。元类型*PyType_Type* 定义了该函数来区分静态和动态分配的类型。）

继承：

此字段会被子类型继承。

默认：

此槽位没有默认值。如果该字段为 NULL，则将使用*Py_TPFLAGS_HAVE_GC* 作为相同功能的替代。

PyObject ****PyTypeObject*.**tp_bases**

基类型的元组。

此字段应当被设为 NULL 并被视为只读。Python 将在类型初始化时 填充它。

对于动态创建的类，可以使用 Py_tp_bases 槽位 来代替*PyType_FromSpecWithBases()* 的 *bases* 参数。推荐使用参数形式。

警告

多重继承不适合静态定义的类型。如果你将 tp_bases 设为一个元组，Python 将不会引发错误，但某些槽位将只从第一个基类型继承。

继承：

这个字段不会被继承。

PyObject **PyTypeObject.tp_mro*

包含基类型的扩展集的元组，以类型本身开始并以 `object` 作为结束，使用方法解析顺序。

此字段应当被设为 `NULL` 并被视为只读。Python 将在类型初始化时 填充它。

继承：

这个字段不会被继承；它是通过*PyType_Ready()* 计算得到的。

PyObject **PyTypeObject.tp_cache**

尚未使用。仅供内部使用。

继承：

这个字段不会被继承。

void **PyTypeObject.tp_subclasses*

一组子类。仅限内部使用的。可能为无效的指针。

要获取子类的列表，则调用 Python 方法 `__subclasses__()`。

在 3.12 版本发生变更: 对于某些类型，该字段将不带有效的 *PyObject**。类型已被改为 `void*` 以指明这一点。

继承：

这个字段不会被继承。

PyObject **PyTypeObject.tp_weaklist*

弱引用列表头，用于指向该类型对象的弱引用。不会被继承。仅限内部使用。

在 3.12 版本发生变更: 内部细节：对于静态内置类型这将总是为 `NULL`，即使添加了弱引用也是如此。每个弱引用都转而保存在 `PyInterpreterState` 上。请使用公共 C-API 或内部 `_PyObject_GET_WEAKREFS_LISTPTR()` 宏来避免此差异。

继承：

这个字段不会被继承。

destructor *PyTypeObject*.tp_del*

该字段已被弃用。请改用*tp_finalize*。

unsigned int *PyTypeObject*.tp_version_tag*

用于索引至方法缓存。仅限内部使用。

继承：

这个字段不会被继承。

destructor *PyTypeObject*.tp_finalize*

一个可选的指向实例最终化函数的指针。函数的签名为:

```
void tp_finalize(PyObject *self);
```

如果设置了*tp_finalize*，解释器将在最终化特定实例时调用它一次。它将由垃圾回收器调用（如果实例是单独循环引用的一部分）或是在对象被释放之前被调用。不论是哪种方式，它都肯定会在尝试打破循环引用之前被调用，以确保它所操作的对象处于正常状态。

tp_finalize 不应改变当前异常状态；因此，编写非关键终结器的推荐做法如下:

```
static void
local_finalize(PyObject *self)
{
    /* 保存当前异常，如果有的话。 */
    PyObject *exc = PyErr_GetRaisedException();

    /* ... */
```

(续下页)

(接上页)

```
    /* 恢复保存的异常。 */
    PyErr_SetRaisedException(exc);
}
```

继承：

此字段会被子类型继承。

Added in version 3.4.

在 3.8 版本发生变更: 在 3.8 版之前必须设置*Py_TPFLAGS_HAVE_FINALIZE* 旗标才能让该字段被使用。现在已不再需要这样做。

参见

” 安全的对象最终化” (**PEP 442**)

vectorcallfunc *PyTypeObject*.**tp_vectorcall**

用于此类型对象的调用的 vectorcall 函数。换句话说，它是被用来实现 `type.__call__` 的*vectorcall*。如果 `tp_vectorcall` 为 `NULL`，默认调用实现将使用 `__new__()` 并且 `__init__()` 将被使用。

继承：

这个字段不会被继承。

Added in version 3.9: （这个字段从 3.8 起即存在，但是从 3.9 开始投入使用）

unsigned char *PyTypeObject*.**tp_watched**

内部对象。请勿使用。

Added in version 3.12.

12.3.6 静态类型

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

这就导致了与在 Python 中定义的类型相关联的类型限制：

- 静态类型只能拥有一个基类；换句话说，他们不能使用多重继承。
- 静态类型对象（但并非它们的实例）是不可变对象。不可能在 Python 中添加或修改类型对象的属性。
- 静态类型对象是跨*子解释器* 共享的，因此它们不应包括任何子解释器专属的状态。

此外，由于*PyTypeObject* 只是作为不透明结构的*受限 API* 的一部分，因此任何使用静态类型的扩展模块都必须针对特定的 Python 次版本进行编译。

12.3.7 堆类型

一种*静态类型的* 替代物是 *堆分配类型*，或者简称 *堆类型*，它与使用 Python 的 `class` 语句创建的类紧密对应。堆类型设置了*Py_TPFLAGS_HEAPTYPE* 旗标。

这是通过填充*PyType_Spec* 结构体并调用*PyType_FromSpec()*，*PyType_FromSpecWithBases()*，*PyType_FromModuleAndSpec()* 或*PyType_FromMetaclass()* 来实现的。

12.3.8 数字对象结构体

type **PyNumberMethods**

该结构体持有指向被对象用来实现数字协议的函数的指针。每个函数都被数字协议 一节中记录的对应名称的函数所使用。

结构体定义如下:

```
typedef struct {
     binaryfunc nb_add;
     binaryfunc nb_subtract;
     binaryfunc nb_multiply;
     binaryfunc nb_remainder;
     binaryfunc nb_divmod;
     ternaryfunc nb_power;
     unaryfunc nb_negative;
     unaryfunc nb_positive;
     unaryfunc nb_absolute;
     inquiry nb_bool;
     unaryfunc nb_invert;
     binaryfunc nb_lshift;
     binaryfunc nb_rshift;
     binaryfunc nb_and;
     binaryfunc nb_xor;
     binaryfunc nb_or;
     unaryfunc nb_int;
     void *nb_reserved;
     unaryfunc nb_float;

     binaryfunc nb_inplace_add;
     binaryfunc nb_inplace_subtract;
     binaryfunc nb_inplace_multiply;
     binaryfunc nb_inplace_remainder;
     ternaryfunc nb_inplace_power;
     binaryfunc nb_inplace_lshift;
     binaryfunc nb_inplace_rshift;
     binaryfunc nb_inplace_and;
     binaryfunc nb_inplace_xor;
     binaryfunc nb_inplace_or;

     binaryfunc nb_floor_divide;
     binaryfunc nb_true_divide;
     binaryfunc nb_inplace_floor_divide;
     binaryfunc nb_inplace_true_divide;

     unaryfunc nb_index;

     binaryfunc nb_matrix_multiply;
     binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;
```

备注

双目和三目函数必须检查其所有操作数的类型，并实现必要的转换（至少有一个操作数是所定义类型的实例）。如果没有为所给出的操作数定义操作，则双目和三目函数必须返回 `Py_NotImplemented`，如果发生了其他错误则它们必须返回 `NULL` 并设置一个异常。

备注

> *nb_reserved* 字段应当始终为 NULL。在之前版本中其名称为 nb_long，并在 Python 3.0.1 中改名。

binaryfunc *PyNumberMethods*.**nb_add**

binaryfunc *PyNumberMethods*.**nb_subtract**

binaryfunc *PyNumberMethods*.**nb_multiply**

binaryfunc *PyNumberMethods*.**nb_remainder**

binaryfunc *PyNumberMethods*.**nb_divmod**

ternaryfunc *PyNumberMethods*.**nb_power**

unaryfunc *PyNumberMethods*.**nb_negative**

unaryfunc *PyNumberMethods*.**nb_positive**

unaryfunc *PyNumberMethods*.**nb_absolute**

inquiry *PyNumberMethods*.**nb_bool**

unaryfunc *PyNumberMethods*.**nb_invert**

binaryfunc *PyNumberMethods*.**nb_lshift**

binaryfunc *PyNumberMethods*.**nb_rshift**

binaryfunc *PyNumberMethods*.**nb_and**

binaryfunc *PyNumberMethods*.**nb_xor**

binaryfunc *PyNumberMethods*.**nb_or**

unaryfunc *PyNumberMethods*.**nb_int**

void **PyNumberMethods*.**nb_reserved**

unaryfunc *PyNumberMethods*.**nb_float**

binaryfunc *PyNumberMethods*.**nb_inplace_add**

binaryfunc *PyNumberMethods*.**nb_inplace_subtract**

binaryfunc *PyNumberMethods*.**nb_inplace_multiply**

binaryfunc *PyNumberMethods*.**nb_inplace_remainder**

ternaryfunc *PyNumberMethods*.**nb_inplace_power**

binaryfunc *PyNumberMethods*.**nb_inplace_lshift**

binaryfunc *PyNumberMethods*.**nb_inplace_rshift**

binaryfunc *PyNumberMethods*.**nb_inplace_and**

binaryfunc *PyNumberMethods*.**nb_inplace_xor**

binaryfunc *PyNumberMethods*.**nb_inplace_or**

binaryfunc *PyNumberMethods*.**nb_floor_divide**

binaryfunc *PyNumberMethods*.**nb_true_divide**

binaryfunc *PyNumberMethods*.**nb_inplace_floor_divide**

binaryfunc *PyNumberMethods*.**nb_inplace_true_divide**

unaryfunc *PyNumberMethods*.**nb_index**

binaryfunc *PyNumberMethods*.**nb_matrix_multiply**

binaryfunc *PyNumberMethods*.**nb_inplace_matrix_multiply**

12.3.9 映射对象结构体

type **PyMappingMethods**

该结构体持有指向对象用于实现映射协议的函数的指针。它有三个成员：

lenfunc *PyMappingMethods*.**mp_length**

该函数将被*PyMapping_Size()* 和*PyObject_Size()* 使用，并具有相同的签名。如果对象没有定义长度则此槽位可被设为 NULL。

binaryfunc *PyMappingMethods*.**mp_subscript**

该函数将被*PyObject_GetItem()* 和*PySequence_GetSlice()* 使用，并具有与 PyObject_GetItem() 相同的签名。此槽位必须被填充以便*PyMapping_Check()* 函数返回 1，否则它可以为 NULL。

objobjargproc *PyMappingMethods*.**mp_ass_subscript**

该函数将被*PyObject_SetItem()*，*PyObject_DelItem()*，*PySequence_SetSlice()* 和*PySequence_DelSlice()* 使用。它具有与 PyObject_SetItem() 相同的签名，但 *v* 也可以被设为 NULL 以删除一个条目。如果此槽位为 NULL，则对象将不支持条目赋值和删除。

12.3.10 序列对象结构体

type **PySequenceMethods**

该结构体持有指向对象用于实现序列协议的函数的指针。

lenfunc *PySequenceMethods*.**sq_length**

此函数被*PySequence_Size()* 和*PyObject_Size()* 所使用，并具有与它们相同的签名。它还被用于通过*sq_item* 和*sq_ass_item* 槽位来处理负索引号。

binaryfunc *PySequenceMethods*.**sq_concat**

此函数被*PySequence_Concat()* 所使用并具有相同的签名。在尝试通过*nb_add* 槽位执行数值相加之后它还会被用于 + 运算符。

ssizeargfunc *PySequenceMethods*.**sq_repeat**

此函数被*PySequence_Repeat()* 所使用并具有相同的签名。在尝试通过*nb_multiply* 槽位执行数值相乘之后它还会被用于 * 运算符。

ssizeargfunc *PySequenceMethods*.**sq_item**

此函数被*PySequence_GetItem()* 所使用并具有相同的签名。在尝试通过*mp_subscript* 槽位执行下标操作之后它还会被用于*PyObject_GetItem()*。该槽位必须被填充以便*PySequence_Check()* 函数返回 1，否则它可以为 NULL。

负索引号是按如下方式处理的：如果*sq_length* 槽位已被填充，它将被调用并使用序列长度来计算出正索引号并传给*sq_item*。如果 sq_length 为 NULL，索引号将原样传给此函数。

ssizeobjargproc *PySequenceMethods*.**sq_ass_item**

此函数被*PySequence_SetItem()* 所使用并具有相同的签名。在尝试通过*mp_ass_subscript* 槽位执行条目赋值和删除操作之后它还会被用于*PyObject_SetItem()* 和*PyObject_DelItem()*。如果对象不支持条目和删除则该槽位可以保持为 NULL。

objobjproc *PySequenceMethods*.**sq_contains**

该函数可供*PySequence_Contains()* 使用并具有相同的签名。此槽位可以保持为 NULL，在此情况下 PySequence_Contains() 只需遍历该序列直到找到一个匹配。

binaryfunc *PySequenceMethods*.**sq_inplace_concat**

此函数被*PySequence_InPlaceConcat()* 所使用并具有相同的签名。它应当修改它的第一个操作数，并将其返回。该槽位可以保持为 NULL，在此情况下 PySequence_InPlaceConcat() 将回退到*PySequence_Concat()*。在尝试通过*nb_inplace_add* 槽位执行数字原地相加之后它还会被用于增强赋值运算符 +=。

ssizeargfunc *PySequenceMethods*.**sq_inplace_repeat**

此函数被*PySequence_InPlaceRepeat()* 所使用并具有相同的签名。它应当修改它的第一个操作数，并将其返回。该槽位可以保持为 NULL，在此情况下 PySequence_InPlaceRepeat() 将回退到*PySequence_Repeat()*。在尝试通过*nb_inplace_multiply* 槽位执行数字原地相乘之后它还会被用于增强赋值运算符 *=。

12.3.11 缓冲区对象结构体

type **PyBufferProcs**

此结构体持有指向*缓冲区协议* 所需要的函数的指针。该协议定义了导出方对象要如何向消费方对象暴露其内部数据。

getbufferproc *PyBufferProcs*.**bf_getbuffer**

此函数的签名为:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

处理发给 *exporter* 的请求来填充 *flags* 所指定的 *view*。除第 (3) 点外，此函数的实现必须执行以下步骤:

(1) 检查请求是否能被满足。如果不能，则会引发 BufferError，将 view->obj 设为 NULL 并返回 -1。

(2) 填充请求的字段。

(3) 递增用于保存导出次数的内部计数器。

(4) 将 view->obj 设为 *exporter* 并递增 view->obj。

(5) 返回 0。

如果 *exporter* 是缓冲区提供方的链式或树型结构的一部分，则可以使用两种主要方案:

- 重导出：树型结构的每个成员作为导出对象并将 view->obj 设为对其自身的新引用。
- 重定向：缓冲区请求将被重定向到树型结构的根对象。在此，view->obj 将为对根对象的新引用。

view 中每个字段的描述参见*缓冲区结构体* 一节，导出方对于特定请求应当如何反应参见*缓冲区请求类型* 一节。

所有在*Py_buffer* 结构体中被指向的内存都属于导出方并必须保持有效直到不再有任何消费方。*format*, *shape*, *strides*, *suboffsets* 和*internal* 对于消费方来说是只读的。

PyBuffer_FillInfo() 提供了一种暴露简单字节缓冲区同时正确处理地所有请求类型的简便方式。

PyObject_GetBuffer() 是针对包装此函数的消费方的接口。

releasebufferproc *PyBufferProcs*.**bf_releasebuffer**

此函数的签名为:

```
void (PyObject *exporter, Py_buffer *view);
```

处理释放缓冲区资源的请求。如果不需要释放任何资源，则*PyBufferProcs.bf_releasebuffer* 可以为 NULL。在其他情况下，此函数的标准实现将执行以下的可选步骤：

(1) 递减用于保存导出次数的内部计数器。

(2) 如果计数器为 0，则释放所有关联到 *view* 的内存。

导出方必须使用*internal* 字段来记录缓冲区专属的资源。该字段将确保恒定，而消费方则可能将原始缓冲区作为 *view* 参数传入。

此函数不可递减 view->obj，因为这是在*PyBuffer_Release()* 中自动完成的（此方案适用于打破循环引用）。

PyBuffer_Release() 是针对包装此函数的消费方的接口。

12.3.12 异步对象结构体

Added in version 3.5.

type **PyAsyncMethods**

此结构体将持有指向需要用来实现*awaitable* 和*asynchronous iterator* 对象的函数的指针。

结构体定义如下：

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
    sendfunc am_send;
} PyAsyncMethods;
```

unaryfunc *PyAsyncMethods*.**am_await**

此函数的签名为：

```
PyObject *am_await(PyObject *self);
```

返回的对象必须为*iterator*，即对其执行*PyIter_Check()* 必须返回 1。

如果一个对象不是*awaitable* 则此槽位可被设为 NULL。

unaryfunc *PyAsyncMethods*.**am_aiter**

此函数的签名为：

```
PyObject *am_aiter(PyObject *self);
```

必须返回一个*asynchronous iterator* 对象。请参阅 __anext__() 了解详情。

如果一个对象没有实现异步迭代协议则此槽位可被设为 NULL。

unaryfunc *PyAsyncMethods*.**am_anext**

此函数的签名为：

```
PyObject *am_anext(PyObject *self);
```

必须返回一个*awaitable* 对象。请参阅 __anext__() 了解详情。此槽位可被设为 NULL。

sendfunc *PyAsyncMethods*.**am_send**

此函数的签名为：

```
PySendResult am_send(PyObject *self, PyObject *arg, PyObject **result);
```

请参阅*PyIter_Send()* 了解详情。此槽位可被设为 NULL。

Added in version 3.10.

12.3.13 槽位类型 typedef

typedef *PyObject* *(***allocfunc**)(*PyTypeObject* *cls, *Py_ssize_t* nitems)

属于稳定 ABI. 此函数的设计目标是将内存分配与内存初始化进行分离。它应当返回一个指向足够容纳实例长度，适当对齐，并初始化为零的内存块的指针，但将*ob_refcnt* 设为 `1` 并将*ob_type* 设为 type 参数。如果类型的*tp_itemsize* 为非零值，则对象的*ob_size* 字段应当被初始化为 *nitems* 而分配内存块的长度应为 `tp_basicsize + nitems*tp_itemsize`，并舍入到 `sizeof(void*)` 的倍数；在其他情况下，*nitems* 将不会被使用而内存块的长度应为*tp_basicsize*。

此函数不应执行任何其他实例初始化操作，即使是分配额外内存也不应执行；那应当由*tp_new* 来完成。

typedef void (***destructor**)(*PyObject**)

属于稳定 ABI.

typedef void (***freefunc**)(void*)

参见*tp_free*。

typedef *PyObject* *(***newfunc**)(*PyTypeObject**, *PyObject**, *PyObject**)

属于稳定 ABI. 参见*tp_new*。

typedef int (***initproc**)(*PyObject**, *PyObject**, *PyObject**)

属于稳定 ABI. 参见*tp_init*。

typedef *PyObject* *(***reprfunc**)(*PyObject**)

属于稳定 ABI. 参见*tp_repr*。

typedef *PyObject* *(***getattrfunc**)(*PyObject* *self, char *attr)

属于稳定 ABI. 返回对象的指定属性的值。

typedef int (***setattrfunc**)(*PyObject* *self, char *attr, *PyObject* *value)

属于稳定 ABI. 为对象设置指定属性的值。将 value 参数设为 `NULL` 将删除该属性。

typedef *PyObject* *(***getattrofunc**)(*PyObject* *self, *PyObject* *attr)

属于稳定 ABI. 返回对象的指定属性的值。

参见*tp_getattro*。

typedef int (***setattrofunc**)(*PyObject* *self, *PyObject* *attr, *PyObject* *value)

属于稳定 ABI. 为对象设置指定属性的值。将 value 参数设为 `NULL` 将删除该属性。

参见*tp_setattro*。

typedef *PyObject* *(***descrgetfunc**)(*PyObject**, *PyObject**, *PyObject**)

属于稳定 ABI. 参见*tp_descr_get*。

typedef int (***descrsetfunc**)(*PyObject**, *PyObject**, *PyObject**)

属于稳定 ABI. 参见*tp_descr_set*。

typedef *Py_hash_t* (***hashfunc**)(*PyObject**)

属于稳定 ABI. 参见*tp_hash*。

typedef *PyObject* *(***richcmpfunc**)(*PyObject**, *PyObject**, int)

属于稳定 ABI. 参见*tp_richcompare*。

typedef *PyObject* *(***getiterfunc**)(*PyObject**)

属于稳定 ABI. 参见*tp_iter*。

typedef *PyObject* *(***iternextfunc**)(*PyObject**)

属于稳定 ABI. 参见*tp_iternext*。

typedef *Py_ssize_t* (***lenfunc**)(*PyObject**)

属于稳定 ABI.

typedef int (***getbufferproc**)(*PyObject**, *Py_buffer**, int)

属于稳定 ABI 自 *3.12* 版起.

typedef void (***releasebufferproc**)(*PyObject**, *Py_buffer**)

属于稳定 ABI 自 *3.12* 版起.

typedef *PyObject* *(***unaryfunc**)(*PyObject**)

属于稳定 ABI.

typedef *PyObject* *(***binaryfunc**)(*PyObject**, *PyObject**)

属于稳定 ABI.

typedef *PySendResult* (***sendfunc**)(*PyObject**, *PyObject**, *PyObject***)

参见*am_send*。

typedef *PyObject* *(***ternaryfunc**)(*PyObject**, *PyObject**, *PyObject**)

属于稳定 ABI.

typedef *PyObject* *(***ssizeargfunc**)(*PyObject**, *Py_ssize_t*)

属于稳定 ABI.

typedef int (***ssizeobjargproc**)(*PyObject**, *Py_ssize_t*, *PyObject**)

属于稳定 ABI.

typedef int (***objobjproc**)(*PyObject**, *PyObject**)

属于稳定 ABI.

typedef int (***objobjargproc**)(*PyObject**, *PyObject**, *PyObject**)

属于稳定 ABI.

12.3.14 例子

下面是一些 Python 类型定义的简单示例。其中包括你可能会遇到的通常用法。有些演示了令人困惑的边际情况。要获取更多示例、实践信息以及教程，请参阅 defining-new-types 和 new-types-topics。

一个基本的静态类型:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_new = myobj_new,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
};
```

你可能还会看到带有更繁琐的初始化器的较旧代码（特别是在 CPython 代码库中）：

```
static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "mymod.MyObject",               /* tp_name */
    sizeof(MyObject),               /* tp_basicsize */
    0,                              /* tp_itemsize */
    (destructor)myobj_dealloc,      /* tp_dealloc */
    0,                              /* tp_vectorcall_offset */
    0,                              /* tp_getattr */
```

(续下页)

(接上页)

```
    0,                                  /* tp_setattr */
    0,                                  /* tp_as_async */
    (reprfunc)myobj_repr,               /* tp_repr */
    0,                                  /* tp_as_number */
    0,                                  /* tp_as_sequence */
    0,                                  /* tp_as_mapping */
    0,                                  /* tp_hash */
    0,                                  /* tp_call */
    0,                                  /* tp_str */
    0,                                  /* tp_getattro */
    0,                                  /* tp_setattro */
    0,                                  /* tp_as_buffer */
    0,                                  /* tp_flags */
    PyDoc_STR("My objects"),            /* tp_doc */
    0,                                  /* tp_traverse */
    0,                                  /* tp_clear */
    0,                                  /* tp_richcompare */
    0,                                  /* tp_weaklistoffset */
    0,                                  /* tp_iter */
    0,                                  /* tp_iternext */
    0,                                  /* tp_methods */
    0,                                  /* tp_members */
    0,                                  /* tp_getset */
    0,                                  /* tp_base */
    0,                                  /* tp_dict */
    0,                                  /* tp_descr_get */
    0,                                  /* tp_descr_set */
    0,                                  /* tp_dictoffset */
    0,                                  /* tp_init */
    0,                                  /* tp_alloc */
    myobj_new,                          /* tp_new */
};
```

一个支持弱引用、实例字典和哈希运算的类型:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
        Py_TPFLAGS_HAVE_GC | Py_TPFLAGS_MANAGED_DICT |
        Py_TPFLAGS_MANAGED_WEAKREF,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};
```

一个不能被子类化且不能被调用以使用*Py_TPFLAGS_DISALLOW_INSTANTIATION*旗标创建实例（例如使用单独的工厂函数）的 str 子类:

```
typedef struct {
    PyUnicodeObject raw;
    char *extra;
} MyStr;

static PyTypeObject MyStr_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyStr",
    .tp_basicsize = sizeof(MyStr),
    .tp_base = NULL,  // set to &PyUnicode_Type in module init
    .tp_doc = PyDoc_STR("my custom str"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_DISALLOW_INSTANTIATION,
    .tp_repr = (reprfunc)myobj_repr,
};
```

最简单的固定长度实例静态类型:

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

最简单的具有可变长度实例的静态类型:

```
typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};
```

12.4 使对象类型支持循环垃圾回收

Python 对循环引用的垃圾检测与回收需要“容器”对象类型的支持，此类型的容器对象中可能包含其它容器对象。不保存其它对象的引用的类型，或者只保存原子类型（如数字或字符串）的引用的类型，不需要显式提供垃圾回收的支持。

要创建一个容器类，类型对象的*tp_flags*字段必须包括*Py_TPFLAGS_HAVE_GC*并提供一个*tp_traverse*处理器的实现。如果该类型的实例是可变的，则还必须提供*tp_clear*的实现。

Py_TPFLAGS_HAVE_GC
设置了此标志位的类型的对象必须符合此处记录的规则。为方便起见，下文把这些对象称为容器对象。

容器类型的构造函数必须符合两个规则:

1. 该对象的内在必须使用*PyObject_GC_New*或*PyObject_GC_NewVar*来分配。
2. 初始化了所有可能包含其他容器的引用的字段后，它必须调用*PyObject_GC_Track()*。

同样的，对象的释放器必须符合两个类似的规则:

1. 在引用其它容器的字段失效前，必须调用*PyObject_GC_UnTrack()*。

2. 必须使用*PyObject_GC_Del()* 释放对象的内存。

> **警告**
>
> 如果一个类型添加了 Py_TPFLAGS_HAVE_GC，则它 必须实现至少一个*tp_traverse* 句柄或显式地使用来自其一个或多个子类的句柄。
>
> 当调用*PyType_Ready()* 或者某些间接调用该函数的 API 如*PyType_FromSpecWithBases()* 或*PyType_FromSpec()* 时解释器将自动填充*tp_flags*, *tp_traverse* 和*tp_clear* 字段，如果该类型是继承自实现了垃圾回收器协议的类并且该子类 没有包括*Py_TPFLAGS_HAVE_GC* 旗标的话。

PyObject_GC_New(TYPE, typeobj)

类似于*PyObject_New* 但专用于设置了*Py_TPFLAGS_HAVE_GC* 旗标的容器对象。

PyObject_GC_NewVar(TYPE, typeobj, size)

与*PyObject_NewVar* 类似但专用于设置了*Py_TPFLAGS_HAVE_GC* 旗标的容器对象。

PyObject ***PyUnstable_Object_GC_NewWithExtraData**(*PyTypeObject* *type, size_t extra_size)

> 这是不稳定 *API*。它可在次发布版中不经警告地改变。

与*PyObject_GC_New* 类似但会在对象的末尾分配 *extra_size* 个字节（在*tp_basicsize* 偏移量处）。除*Python 对象标头* 外，分配的内存将初始化为零。

附加数据将与对象一起被释放，但在其他情况下则不会由 Python 来管理。

> **警告**
>
> 此函数被标记为非稳定的因为在实例之后保留附加数据的机制尚未确定。要分配可变数量的字段，推荐改用*PyVarObject* 和*tp_itemsize*。

Added in version 3.12.

PyObject_GC_Resize(TYPE, op, newsize)

重新调整*PyObject_NewVar* 所分配对象的大小。返回调整大小后的类型为 `TYPE*` 的对象（指向任意 C 类型）或在失败时返回 `NULL`。

op 必须为 *PyVarObject** 类型并且不能已被回收器所追踪。*newsize* 必须为*Py_ssize_t* 类型。

void **PyObject_GC_Track**(*PyObject* *op)

属于稳定 ABI. 把对象 *op* 加入到垃圾回收器跟踪的容器对象中。对象在被回收器跟踪时必须保持有效的，因为回收器可能在任何时候开始运行。在*tp_traverse* 处理前的所有字段变为有效后，必须调用此函数，通常在靠近构造函数末尾的位置。

int **PyObject_IS_GC**(*PyObject* *obj)

如果对象实现了垃圾回收器协议则返回非零值，否则返回 0。

如果此函数返回 0 则对象无法被垃圾回收器追踪。

int **PyObject_GC_IsTracked**(*PyObject* *op)

属于稳定 ABI 自 *3.9* 版起. 如果 *op* 对象的类型实现了 GC 协议且 *op* 目前正被垃圾回收器追踪则返回 1，否则返回 0。

这类似于 Python 函数 `gc.is_tracked()`。

Added in version 3.9.

int **PyObject_GC_IsFinalized**(*PyObject* *op)

属于稳定 ABI 自 *3.9* 版起. 如果 *op* 对象的类型实现了 GC 协议且 *op* 已经被垃圾回收器终结则返回 1，否则返回 0。

这类似于 Python 函数 `gc.is_finalized()`。

Added in version 3.9.

void **PyObject_GC_Del**(void *op)

属于稳定 ABI. 使用*PyObject_GC_New* 或*PyObject_GC_NewVar* 释放分配给对象的内存。

void **PyObject_GC_UnTrack**(void *op)

属于稳定 ABI. 从回收器跟踪的容器对象集合中移除 *op* 对象。请注意可以在此对象上再次调用*PyObject_GC_Track()* 以将其加回到被跟踪对象集合。释放器 (*tp_dealloc* 句柄) 应当在*tp_traverse* 句柄所使用的任何字段失效之前为对象调用此函数。

在 3.8 版本发生变更: `_PyObject_GC_TRACK()` 和 `_PyObject_GC_UNTRACK()` 宏已从公有 C API 中删除。

tp_traverse 处理接收以下类型的函数形参。

typedef int (***visitproc**)(*PyObject* *object, void *arg)

属于稳定 ABI. 传给*tp_traverse* 处理的访问函数的类型。*object* 是容器中需要被遍历的一个对象，第三个形参对应于*tp_traverse* 处理的 *arg* 。Python 核心使用多个访问者函数实现循环引用的垃圾检测，不需要用户自行实现访问者函数。

tp_traverse 处理必须是以下类型:

typedef int (***traverseproc**)(*PyObject* *self, *visitproc* visit, void *arg)

属于稳定 ABI. 用于容器对象的遍历函数。它的实现必须对 *self* 所直接包含的每个对象调用 *visit* 函数，*visit* 的形参为所包含对象和传给处理程序的 *arg* 值。*visit* 函数调用不可附带 `NULL` 对象作为参数。如果 *visit* 返回非零值，则该值应当被立即返回。

为了简化*tp_traverse* 处理的实现，Python 提供了一个*Py_VISIT()* 宏。若要使用这个宏，必须把*tp_traverse* 的参数命名为 *visit* 和 *arg* 。

Py_VISIT(o)

如果 *PyObject** *o* 不为 `NULL`，则调用 *visit* 回调，附带参数 *o* 和 *arg*。如果 *visit* 返回一个非零值，则将返回该值。使用此宏之后，*tp_traverse* 处理器看起来是这样的:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

tp_clear 处理程序必须为*inquiry* 类型，如果对象不可变则为 `NULL`。

typedef int (***inquiry**)(*PyObject* *self)

属于稳定 ABI. 丢弃产生循环引用的引用。不可变对象不需要声明此方法，因为他们不可能直接产生循环引用。需要注意的是，对象在调用此方法后必须仍是有效的（不能对引用只调用*Py_DECREF()* 方法）。当垃圾回收器检测到该对象在循环引用中时，此方法会被调用。

12.4.1 控制垃圾回收器状态

这个 C-API 提供了以下函数用于控制垃圾回收的运行。

Py_ssize_t **PyGC_Collect**(void)

属于稳定 ABI. 执行完全的垃圾回收，如果垃圾回收器已启用的话。（请注意 `gc.collect()` 会无条件地执行它。）

返回已回收的 + 无法回收的不可获取对象的数量。如果垃圾回收器被禁用或已在执行回收，则立即返回 `0`。在垃圾回收期间发生的错误会被传给 `sys.unraisablehook`。此函数不会引发异常。

int **PyGC_Enable**(void)

属于稳定 ABI 自 *3.10* 版起. 启用垃圾回收器：类似于 `gc.enable()`。返回之前的状态，0 为禁用而 1 为启用。

Added in version 3.10.

int **PyGC_Disable**(void)

属于稳定 ABI 自 *3.10* 版起. 禁用垃圾回收器：类似于 `gc.disable()`。返回之前的状态，0 为禁用而 1 为启用。

Added in version 3.10.

int **PyGC_IsEnabled**(void)

属于稳定 ABI 自 *3.10* 版起. 查询垃圾回收器的状态：类似于 `gc.isenabled()`。返回当前的状态，0 为禁用而 1 为启用。

Added in version 3.10.

12.4.2 查询垃圾回收器状态

该 C-API 提供了以下接口用于查询有关垃圾回收器的信息。

void **PyUnstable_GC_VisitObjects**(*gcvisitobjects_t* callback, void *arg)

这是不稳定 *API*。它可在次发布版中不经警告地改变。

在全部活动的支持 GC 的对象上运行所提供的 *callback*。*arg* 会被传递给所有 *callback* 的唤起操作。

警告

如果新对象被回调（取消）分配后再被访问其行为是未定义的。

垃圾回收在运行期间被禁用。在回调中显式地运行回收可能导致未定义的行为，例如多次访问同一对象或完全不访问。

Added in version 3.12.

typedef int (***gcvisitobjects_t**)(*PyObject* *object, void *arg)

要传给 *PyUnstable_GC_VisitObjects()* 的访问者函数的类型。*arg* 与传给 `PyUnstable_GC_VisitObjects` 的 *arg* 相同。返回 `1` 以继续迭代，返回 `0` 以停止迭代。其他返回值目前被保留因此返回任何其他值的行为都是未定义的。

Added in version 3.12.

CHAPTER 13

API 和 ABI 版本管理

CPython 在下列宏中暴露其版本号。请注意这对应于 **编译**用版本代码，而不是 **运行时**使用的版本。

请参阅*C API 的稳定性* 查看跨版本的 API 和 ABI 稳定情。

PY_MAJOR_VERSION

`3` (`3.4.1a2` 中的第一段)。

PY_MINOR_VERSION

`4` (`3.4.1a2` 中的第二段)。

PY_MICRO_VERSION

`1` (`3.4.1a2` 中第三段的数字)。

PY_RELEASE_LEVEL

`a` (`3.4.1a2` 中第 3 段的字母)。可能为 `0xA` 即 alpha, `0xB` 即 beta, `0xC` 即 release candidate 或 `0xF` 即 final。

PY_RELEASE_SERIAL

`2` (`3.4.1a2` 中的末尾数字)。零代表最终发布版。

PY_VERSION_HEX

编码为单个整数形式的 Python 版本号。

底层的版本信息可通过按以下方式将其当作 32 比特的数字处理来获取:

字节串	位数（大端字节序）	含意	`3.4.1a2` 的值
1	1-8	`PY_MAJOR_VERSION`	`0x03`
2	9-16	`PY_MINOR_VERSION`	`0x04`
3	17-24	`PY_MICRO_VERSION`	`0x01`
4	25-28	`PY_RELEASE_LEVEL`	`0xA`
	29-32	`PY_RELEASE_SERIAL`	`0x2`

这样 `3.4.1a2` 即十六进制版本号的 `0x030401a2` 而 `3.10.0` 即十六进制版本号的 `0x030a00f0`。

用于进行数值比较，例如 `#if PY_VERSION_HEX >= ...`。

该版本还可通过符号*`Py_Version`* 获取。

const unsigned long **Py_Version**

属于稳定 ABI 自 *3.11* 版起. Python 运行时版本号编码在一个整数常量中，所用格式与*PY_VERSION_HEX* 宏的相同。这包含了在运行时使用的 Python 版本。

Added in version 3.11.

所有提到的宏都定义在 Include/patchlevel.h。

CHAPTER 14

监控 C API

在 3.13 版中添加。

一个扩展可能需要与事件监视系统进行交互。订阅事件和注册回调都可通过在 `sys.monitoring` 中暴露的 Python API 来完成。

CHAPTER 15

生成执行事件

下面的函数使得扩展可以在模拟执行 Python 代码时触发监控事件。这样的函数都接受一个 `PyMonitoringState` 结构体，其中包含有关事件激活状态的简明信息以及事件参数，此类参数包括代表代码对象的 `PyObject*`、指令偏移量，有时还包括额外的、事件专属的参数（请参阅 `sys.monitoring` 了解有关不同事件回调签名的详情）。`codelike` 参数应为 `types.CodeType` 的实例或是某个模拟它的类型。

VM 会在触发事件时禁用跟踪，因此用户不需要额外的操作。

监控函数被调用时不应设置异常，除非是下面列出的用于处理当前异常的函数。

type **PyMonitoringState**

事件类型状态的表示形式。它由用户分配，但其内容则由下文描述的监控 API 函数来维护。

下面的所有函数均在成功时返回 0 并在失败时返回 -1 (同时设置一个异常)。

请参阅 `sys.monitoring` 获取事件的描述。

int **PyMonitoring_FirePyStartEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

发出 `PY_START` 事件。

int **PyMonitoring_FirePyResumeEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

发出 `PY_RESUME` 事件。

int **PyMonitoring_FirePyReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

发出 `PY_RETURN` 事件。

int **PyMonitoring_FirePyYieldEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

发出 `PY_YIELD` 事件。

int **PyMonitoring_FireCallEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *callable, *PyObject* *arg0)

发出 `CALL` 事件。

int **PyMonitoring_FireLineEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, int lineno)

发出 `LINE` 事件。

int **PyMonitoring_FireJumpEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

发出 JUMP 事件。

int **PyMonitoring_FireBranchEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

发出 BRANCH 事件。

int **PyMonitoring_FireCReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

发出 C_RETURN 事件。

int **PyMonitoring_FirePyThrowEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

使用当前（即*PyErr_GetRaisedException()* 返回的）异常发出 PY_THROW 事件。

int **PyMonitoring_FireRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

使用当前（即*PyErr_GetRaisedException()* 所返回的）异常发出 RAISE 事件。event with the current).

int **PyMonitoring_FireCRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

使用当前（即*PyErr_GetRaisedException()* 所返回的）异常发出 C_RAISE 事件。

int **PyMonitoring_FireReraiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

使用当前（即*PyErr_GetRaisedException()* 所返回的）异常发出 RERAISE 事件。

int **PyMonitoring_FireExceptionHandledEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

使用当前（即*PyErr_GetRaisedException()* 所返回的）异常发出 EXCEPTION_HANDLED 事件。

int **PyMonitoring_FirePyUnwindEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

使用当前（即*PyErr_GetRaisedException()* 所返回的）异常发出 PY_UNWIND 事件。

int **PyMonitoring_FireStopIterationEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *value)

发出 STOP_ITERATION 事件。如果 value 是一个 StopIteration 实例，它将被使用。在其他情况下，将新建一个 StopIteration 实例并以 value 作为其参数。

15.1 管理监控状态

监控状态可在监控作用域的协助下进行管理。一个作用域通常对应一个 python 函数。

int **PyMonitoring_EnterScope**(*PyMonitoringState* *state_array, uint64_t *version, const uint8_t *event_types, *Py_ssize_t* length)

进入一个监控作用域。event_types 是由可从该作用域发生事件的事件 ID 组成的数组。例如，PY_START 事件的 ID 值为 PY_MONITORING_EVENT_PY_START，其对应数字等于 sys.monitoring.events.PY_START 的以 2 为底的对数。state_array 是由对应 event_types 中每个事件的监控状态组成的数组，它由用户进行分配但是由 PyMonitoring_EnterScope() 使用事件激活状态相关信息填充。event_types 的大小 (因而也是 state_array 的大小) 由 length 给出。

version 是一个指向应与 state_array 一起由用户分配的值并初始化为 0，然后仅由 PyMonitoring_EnterScope() 本身来设置。它允许此函数确定事件状态自上次调用以来是否发生改变，并在它们未改变时立即返回。

这里所称的作用域是词法意义下的作用域：一个函数、类或方法。PyMonitoring_EnterScope() 应当在进入词法作用域时被调用。在模拟一个递归 Python 函数之类的情况下，作用域可被重进入，并重用相同的 *state_array* 和 *version*。当某个代码执行暂停时，例如在模拟一个生成器时，此作用域需要被退出并重进入。

对应 *event_types* 的宏如下：

宏	事件
PY_MONITORING_EVENT_BRANCH	BRANCH
PY_MONITORING_EVENT_CALL	CALL
PY_MONITORING_EVENT_C_RAISE	C_RAISE
PY_MONITORING_EVENT_C_RETURN	C_RETURN
PY_MONITORING_EVENT_EXCEPTION_HANDLED	EXCEPTION_HANDLED
PY_MONITORING_EVENT_INSTRUCTION	INSTRUCTION
PY_MONITORING_EVENT_JUMP	JUMP
PY_MONITORING_EVENT_LINE	LINE
PY_MONITORING_EVENT_PY_RESUME	PY_RESUME
PY_MONITORING_EVENT_PY_RETURN	PY_RETURN
PY_MONITORING_EVENT_PY_START	PY_START
PY_MONITORING_EVENT_PY_THROW	PY_THROW
PY_MONITORING_EVENT_PY_UNWIND	PY_UNWIND
PY_MONITORING_EVENT_PY_YIELD	PY_YIELD
PY_MONITORING_EVENT_RAISE	RAISE
PY_MONITORING_EVENT_RERAISE	RERAISE
PY_MONITORING_EVENT_STOP_ITERATION	STOP_ITERATION

int **PyMonitoring_ExitScope**(void)

退出使用 `PyMonitoring_EnterScope()` 进入的上一个作用域。

int **PY_MONITORING_IS_INSTRUMENTED_EVENT**(uint8_t ev)

如果对应事件 ID *ev* 的事件属于 局部事件则返回真值。

Added in version 3.13.

自 3.13.3 版本弃用: 此函数状态为*soft deprecated*。

APPENDIX A

术语对照表

>>>

interactive shell 中默认的 Python 提示符。往往会显示于能以交互方式在解释器里执行的样例代码之前。

...

具有以下含义：

- *interactive* shell 中输入特殊代码时默认的 Python 提示符，特殊代码包括缩进的代码块，左右成对分隔符（圆括号、方括号、花括号或三重引号等）之内，或是在指定一个装饰器之后。
- `Ellipsis` 内置常量。

abstract base class -- 抽象基类

抽象基类简称 ABC，是对*duck-typing* 的补充，它提供了一种定义接口的新方式，相比之下其他技巧例如 `hasattr()` 显得过于笨拙或有微妙错误（例如使用 魔术方法）。ABC 引入了虚拟子类，这种类并非继承自其他类，但却仍能被 `isinstance()` 和 `issubclass()` 所认可；详见 `abc` 模块文档。Python 自带许多内置的 ABC 用于实现数据结构（在 `collections.abc` 模块中）、数字（在 `numbers` 模块中）、流（在 `io` 模块中）、导入查找器和加载器（在 `importlib.abc` 模块中）。你可以使用 `abc` 模块来创建自己的 ABC。

annotation -- 标注

关联到某个变量、类属性、函数形参或返回值的标签，被约定作为类型注解 来使用。

局部变量的标注在运行时不可访问，但全局变量、类属性和函数的标注会分别存放模块、类和函数的 `__annotations__` 特殊属性中。

参见*variable annotation*, *function annotation*, **PEP 484** 和 **PEP 526**，对此功能均有介绍。另请参见 annotations-howto 了解使用标注的最佳实践。

argument -- 参数

在调用函数时传给*function* （或*method* ）的值。参数分为两种：

- *关键字参数*: 在函数调用中前面带有标识符（例如 `name=`）或者作为包含在前面带有 `**` 的字典里的值传入。举例来说，`3` 和 `5` 在以下对 `complex()` 的调用中均属于关键字参数:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *位置参数*: 不属于关键字参数的参数。位置参数可出现于参数列表的开头以及/或者作为前面带有 `*` 的*iterable* 里的元素被传入。举例来说，`3` 和 `5` 在以下调用中均属于位置参数:

```
complex(3, 5)
complex(*(3, 5))
```

参数会被赋值给函数体中对应的局部变量。有关赋值规则参见 calls 一节。根据语法，任何表达式都可用来表示一个参数；最终算出的值会被赋给对应的局部变量。

另参见*parameter* 术语表条目，常见问题中 参数与形参的区别以及 **PEP 362**。

asynchronous context manager -- 异步上下文管理器
此种对象通过定义 `__aenter__()` 和 `__aexit__()` 方法来对 `async with` 语句中的环境进行控制。由 **PEP 492** 引入。

asynchronous generator -- 异步生成器
返回值为*asynchronous generator iterator* 的函数。它与使用 `async def` 定义的协程函数很相似，不同之处在于它包含 `yield` 表达式以产生一系列可在 `async for` 循环中使用的值。

此术语通常是指异步生成器函数，但在某些情况下则可能是指 异步生成器迭代器。如果需要清楚表达具体含义，请使用全称以避免歧义。

一个异步生成器函数可能包含 `await` 表达式或者 `async for` 以及 `async with` 语句。

asynchronous generator iterator -- 异步生成器迭代器
asynchronous generator 函数所创建的对象。

此对象属于*asynchronous iterator*，当使用 `__anext__()` 方法调用时会返回一个可等待对象来执行异步生成器函数的函数体直到下一个 `yield` 表达式。

每个 `yield` 会临时暂停处理过程，记住执行状态（包括局部变量和挂起的 try 语句）。当 异步生成器迭代器通过 `__anext__()` 所返回的另一个可等待对象有效地恢复时，它会从离开位置恢复处理过程。参见 **PEP 492** 和 **PEP 525**。

asynchronous iterable -- 异步可迭代对象
一个可以在 `async for` 语句中使用的对象。必须通过它的 `__aiter__()` 方法返回一个*asynchronous iterator*。由 **PEP 492** 引入。

asynchronous iterator -- 异步迭代器
一个实现了 `__aiter__()` 和 `__anext__()` 方法的对象。`__anext__()` 必须返回一个*awaitable* 对象。`async for` 会处理异步迭代器的 `__anext__()` 方法所返回的可等待对象直到其引发一个 `StopAsyncIteration` 异常。由 **PEP 492** 引入。

attribute -- 属性
关联到一个对象的值，通常使用点号表达式按名称来引用。举例来说，如果对象 *o* 具有属性 *a* 则可以用 *o.a* 来引用它。

如果对象允许，将未被定义为 identifiers 的非标识名称用作一个对象的属性也是可以的，例如使用 `setattr()`。这样的属性将无法使用点号表达式来访问，而是必须通过 `getattr()` 来获取。

awaitable -- 可等待对象
一个可在 `await` 表达式中使用的对象。可以是*coroutine* 或是具有 `__await__()` 方法的对象。参见 **PEP 492**。

BDFL
“终身仁慈独裁者”的英文缩写，即 Guido van Rossum，Python 的创造者。

binary file -- 二进制文件
file object 能够读写字节型对象。二进制文件的例子包括以二进制模式 (`'rb'`, `'wb'` 或 `'rb+'`) 打开的文件、`sys.stdin.buffer`、`sys.stdout.buffer` 以及 `io.BytesIO` 和 `gzip.GzipFile` 的实例。

另请参见*text file* 了解能够读写 `str` 对象的文件对象。

borrowed reference -- 借入引用
在 Python 的 C API 中，借用引用是指一种对象引用，使用该对象的代码并不持有该引用。如果对象被销毁则它就会变成一个悬空指针。例如，垃圾回收器可以移除对象的最后一个*strong reference* 来销毁它。

推荐在*borrowed reference* 上调用 `Py_INCREF()` 以将其原地转换为*strong reference*，除非是当该对象无法在借入引用的最后一次使用之前被销毁。`Py_NewRef()` 函数可以被用来创建一个新的*strong reference*。

bytes-like object -- 字节型对象
支持缓冲协议 并且能导出 C-*contiguous* 缓冲的对象。这包括所有 `bytes`、`bytearray` 和 `array.array` 对象，以及许多普通 `memoryview` 对象。字节型对象可在多种二进制数据操作中使用；这些操作包括压缩、保存为二进制文件以及通过套接字发送等。

某些操作需要可变的二进制数据。这种对象在文档中常被称为“可读写字节类对象”。可变缓冲对象的例子包括 `bytearray` 以及 `bytearray` 的 `memoryview`。其他操作要求二进制数据存放于不可变对象 (” 只读字节类对象”)；这种对象的例子包括 `bytes` 以及 `bytes` 对象的 `memoryview`。

bytecode -- 字节码
Python 源代码会被编译为字节码，即 CPython 解释器中表示 Python 程序的内部代码。字节码还会缓存在 `.pyc` 文件中，这样第二次执行同一文件时速度更快（可以免去将源码重新编译为字节码）。这种” 中间语言” 运行在根据字节码执行相应机器码的*virtual machine* 之上。请注意不同 Python 虚拟机上的字节码不一定通用，也不一定能在不同 Python 版本上兼容。

字节码指令列表可以在 dis 模块的文档中查看。

callable -- 可调用对象
可调用对象就是可以执行调用运算的对象，并可能附带一组参数 (参见*argument*)，使用以下语法:

```
callable(argument1, argument2, argumentN)
```

function，还可扩展到*method* 等，就属于可调用对象。实现了 `__call__()` 方法的类的实例也属于可调用对象。

callback -- 回调
一个作为参数被传入以用以在未来的某个时刻被调用的子例程函数。

class -- 类
用来创建用户定义对象的模板。类定义通常包含对该类的实例进行操作的方法定义。

class variable -- 类变量
在类中定义的变量，并且仅限在类的层级上修改 (而不是在类的实例中修改)。

closure variable -- 闭包变量
引用自*nested scope* 的*free variable*，它是在外层作用域中定义而不是在运行时自全局或内置命名空间中取得。可能使用 `nonlocal` 关键字显式地定义以允许写入访问，或者如果变量仅供读取则只需隐式地定义。

例如，在以下代码的 `inner` 函数中，`x` 和 `print` 均为自由变量，但只有 `x` 属于 闭包变量:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

由于 `codeobject.co_freevars` 属性的存在（虽然名称如此，但其仅包括闭包变量名称而非列出所有被引用的自由变量），更一般化的术语*free variable* 有时在即便本意是专指闭包变量时也会被使用。

complex number -- 复数
对普通实数系统的扩展，其中所有数字都被表示为一个实部和一个虚部的和。虚数是虚数单位（`-1` 的平方根）的实倍数，通常在数学中写为 `i`，在工程学中写为 `j`。Python 内置了对复数的支持，采用工程学标记方式；虚部带有一个 `j` 后缀，例如 `3+1j`。如果需要 `math` 模块内对象的对应复数版本，请使用 `cmath`，复数的使用是一个比较高级的数学特性。如果你感觉没有必要，忽略它们也几乎不会有任何问题。

context -- 上下文
此术语根据其所在场合和使用方式的不同而具有不同的含义。一些常见的含义为:

- 通过 `with` 语句由一个*context manager* 所创建的临时环境或状态。
- 一组关联到特定 `contextvars.Context` 对象并通过 `ContextVar` 对象来访问的键值绑定。另请参见*context variable*。
- 一个 `contextvars.Context` 对象。另请参见*current context*。

上下文管理协议
`__enter__()` 和 `__exit__()` 方法将由 `with` 语句来调用。参见 **PEP 343**。

context manager -- 上下文管理器
一个实现了*context management protocol* 并负责控制某个 `with` 语句内的环境的对象。参见 **PEP 343**。

context variable -- 上下文变量
一个具体值取决于哪个上下文是*current context* 的变量。这些值是通过 `contextvars.ContextVar` 对象来访问的。上下文变量主要被用来隔离并发的异步任务之间的状态。

contiguous -- 连续
一个缓冲如果是 *C 连续*或 *Fortran 连续*就会被认为是连续的。零维缓冲是 C 和 Fortran 连续的。在一维数组中，所有条目必须在内存中彼此相邻地排列，采用从零开始的递增索引顺序。在多维 C-连续数组中，当按内存地址排列时用最后一个索引访问条目时速度最快。但是在 Fortran 连续数组中则是用第一个索引最快。

coroutine -- 协程
协程是子例程的更一般形式。子例程可以在某一点进入并在另一点退出。协程则可以在许多不同的点上进入、退出和恢复。它们可通过 `async def` 语句来实现。参见 **PEP 492**。

coroutine function -- 协程函数
返回一个*coroutine* 对象的函数。协程函数可通过 `async def` 语句来定义，并可能包含 `await`、`async for` 和 `async with` 关键字。这些特性是由 **PEP 492** 引入的。

CPython
Python 编程语言的规范实现，在 python.org 上发布。"CPython" 一词用于在必要时将此实现与其他实现例如 Jython 或 IronPython 相区别。

current context -- 当前上下文
在当前被 `ContextVar` 对象用来访问 (获取或设置) 上下文变量 的值的*context* (`contextvars.Context` 对象)。每个线程都具有它自己的当前上下文。用于执行异步任务 (参见 `asyncio`) 的框架会在每个任务开始或恢复执行时将任务关联到一个成为当前上下文的上下文。

decorator -- 装饰器
返回值为另一个函数的函数，通常使用 `@wrapper` 语法形式来进行函数变换。装饰器的常见例子包括 `classmethod()` 和 `staticmethod()`。

装饰器语法只是一种语法糖，以下两个函数定义在语义上完全等价:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

同样的概念也适用于类，但通常较少这样使用。有关装饰器的详情可参见 函数定义和 类定义的文档。

descriptor -- 描述器
任何定义了 `__get__()`, `__set__()` 或 `__delete__()` 方法的对象。当一个类属性为描述器时，它的特殊绑定行为就会在属性查找时被触发。通常情况下，使用 *a.b* 来获取、设置或删除一个属性时会在 *a* 类的字典中查找名称为 *b* 的对象，但如果 *b* 是一个描述器，则会调用对应的描述器方法。理解描述器的概念是更深层次理解 Python 的关键，因为这是许多重要特性的基础，包括函数、方法、特征属性、类方法、静态方法以及对超类的引用等等。

有关描述器的方法的更多信息，请参阅 descriptors 或 描述器使用指南。

dictionary -- 字典
一个关联数组，其中的任意键都映射到相应的值。键可以是任何具有 `__hash__()` 和 `__eq__()` 方法的对象。在 Perl 中称为 hash。

dictionary comprehension -- 字典推导式
处理一个可迭代对象中的所有或部分元素并返回结果字典的一种紧凑写法。`results = {n: n ** 2 for n in range(10)}` 将生成一个由键 `n` 到值 `n ** 2` 的映射构成的字典。参见 comprehensions。

dictionary view -- 字典视图
从 `dict.keys()`, `dict.values()` 和 `dict.items()` 返回的对象被称为字典视图。它们提供了字典条目的一个动态视图，这意味着当字典改变时，视图也会相应改变。要将字典视图强制转换为真正的列表，可使用 `list(dictview)`。参见 dict-views。

docstring -- 文档字符串
作为类、函数或模块之内的第一个表达式出现的字符串字面值。它在代码块被执行时将被忽略，但会被编译器识别并放入所在类、函数或模块的 `__doc__` 属性中。由于它可用于代码内省，因此是存放对象的文档的规范位置。

duck-typing -- 鸭子类型
指一种编程风格，它并不依靠查找对象类型来确定其是否具有正确的接口，而是直接调用或使用其方法或属性（“看起来像鸭子，叫起来也像鸭子，那么肯定就是鸭子。”）由于强调接口而非特定类型，设计良好的代码可通过允许多态替代来提升灵活性。鸭子类型避免使用 `type()` 或 `isinstance()` 检测。(但要注意鸭子类型可以使用抽象基类 作为补充。) 而往往会采用 `hasattr()` 检测或是*EAFP* 编程。

dunder
An informal short-hand for "double underscore", used when talking about a *special method*. For example, `__init__` is often pronounced "dunder init".

EAFP
“求原谅比求许可更容易”的英文缩写。这种 Python 常用代码编写风格会假定所需的键或属性存在，并在假定错误时捕获异常。这种简洁快速风格的特点就是大量运用 `try` 和 `except` 语句。于其相对的则是所谓*LBYL* 风格，常见于 C 等许多其他语言。

expression -- 表达式
可以求出某个值的语法单元。换句话说，一个表达式就是表达元素例如字面值、名称、属性访问、运算符或函数调用的汇总，它们最终都会返回一个值。与许多其他语言不同，并非所有语言构件都是表达式。还存在不能被用作表达式的*statement*，例如 `while`。赋值也是属于语句而非表达式。

extension module -- 扩展模块
以 C 或 C++ 编写的模块，使用 Python 的 C API 来与语言核心以及用户代码进行交互。

f-string -- f-字符串
带有 `'f'` 或 `'F'` 前缀的字符串字面值通常被称为“f-字符串”即 格式化字符串字面值的简写。参见 **PEP 498**。

file object -- 文件对象
对外公开面向文件的 API（带有 `read()` 或 `write()` 等方法）以使用下层资源的对象。根据其创建方式的不同，文件对象可以处理对真实磁盘文件、其他类型的存储或通信设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等）。文件对象也被称为 文件型对象或 流。

实际上共有三种类别的文件对象: 原始二进制文件, 缓冲二进制文件 以及文本文件。它们的接口定义均在 `io` 模块中。创建文件对象的规范方式是使用 `open()` 函数。

file-like object -- 文件型对象
file object 的同义词。

filesystem encoding and error handler -- 文件系统编码格式与错误处理器
Python 用来从操作系统解码字节串和向操作系统编码 Unicode 的编码格式与错误处理器。

文件系统编码格式必须保证能成功解码长度在 128 以下的所有字节串。如果文件系统编码格式无法提供此保证，则 API 函数可能会引发 `UnicodeError`。

sys.getfilesystemencoding() 和 sys.getfilesystemencodeerrors() 函数可被用来获取文件系统编码格式与错误处理器。

filesystem encoding and error handler 是在 Python 启动时通过*PyConfig_Read()* 函数来配置的：请参阅*PyConfig* 的*filesystem_encoding* 和*filesystem_errors* 等成员。

另请参见*locale encoding*。

finder -- 查找器
一种会尝试查找被导入模块的*loader* 的对象。

存在两种类型的查找器：元路径查找器 配合 sys.meta_path 使用，以及路径条目查找器 配合 sys.path_hooks 使用。

请参阅 finders-and-loaders 和 importlib 以了解更多细节。

floor division -- 向下取整除法
向下舍入到最接近的整数的数学除法。向下取整除法的运算符是 // 。例如，表达式 11 // 4 的计算结果是 2 ，而与之相反的是浮点数的真正除法返回 2.75 。注意 (-11) // 4 会返回 -3 因为这是 -2.75 向下舍入得到的结果。见 **PEP 238** 。

free threading -- 自由线程
一种线程模型，在同一个解释器内部的多个线程可以同时运行 Python 字节码。与此相对的是*global interpreter lock*，在同一时刻只允许一个线程执行 Python 字节码。参见 **PEP 703**。

free variable -- 自由变量
在正式场合下，如 语言执行模型所定义的，自由变量是指在某个命名空间中被使用的不属于该命名空间中的局部变量的任何变量。参见*closure variable* 中的样例。在实际应用中，由于 codeobject.co_freevars 属性被如此命名，该术语有时也被用作*closure variable* 的同义词。

function -- 函数
可以向调用者返回某个值的一组语句。还可以向其传入零个或多个参数 并在函数体执行中被使用。另见*parameter*, *method* 和 function 等节。

function annotation -- 函数标注
即针对函数形参或返回值的*annotation* 。

函数标注通常用于类型提示：例如以下函数预期接受两个 int 参数并预期返回一个 int 值:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

函数标注语法的详解见 function 一节。

参见*variable annotation* 和 **PEP 484**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标的最佳实践。

__future__
future 语句, from __future__ import <feature> 指示编译器使用将在未来的 Python 发布版中成为标准的语法和语义来编译当前模块。__future__ 模块文档记录了可能的 *feature* 取值。通过导入此模块并对其变量求值，你可以看到每项新特性在何时被首次加入到该语言中以及它将（或已）在何时成为默认:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection -- 垃圾回收
释放不再被使用的内存空间的过程。Python 是通过引用计数和一个能够检测和打破循环引用的循环垃圾回收器来执行垃圾回收的。可以使用 gc 模块来控制垃圾回收器。

generator -- 生成器
返回一个*generator iterator* 的函数。它看起来很像普通函数，不同点在于其包含 yield 表达式以便产生一系列值供给 for-循环使用或是通过 next() 函数逐一获取。

通常是指生成器函数，但在某些情况下也可能是指 生成器迭代器。如果需要清楚表达具体含义，请使用全称以避免歧义。

generator iterator -- 生成器迭代器
generator 函数所创建的对象。

每个 `yield` 会临时暂停处理过程，记住执行状态（包括局部变量和挂起的 try 语句）。当 生成器迭代器恢复时，它会从离开位置继续执行（不同于每次被唤起时都从新开始的普通函数）。

generator expression -- 生成器表达式
返回一个*iterator* 的*expression*。它看起来很像普通表达式后带有定义了一个循环变量、范围的 `for` 子句，以及一个可选的 `if` 子句。以下复合表达式会为外层函数生成一系列值:

```
>>> sum(i*i for i in range(10))         # 平方值 0, 1, 4, ... 81 之和
285
```

generic function -- 泛型函数
为不同的类型实现相同操作的多个函数所组成的函数。在调用时会由调度算法来确定应该使用哪个实现。

另请参见*single dispatch* 术语表条目、`functools.singledispatch()` 装饰器以及 **PEP 443**。

generic type -- 泛型
可参数化的*type*；通常为 `list` 或 `dict` 这样的 容器类。用于类型提示 和注解。

更多细节参见 泛型别名类型、**PEP 483**、**PEP 484**、**PEP 585** 和 `typing` 模块。

GIL
参见*global interpreter lock*。

global interpreter lock -- 全局解释器锁
CPython 解释器所采用的一种机制，它确保同一时刻只有一个线程在执行 Python *bytecode*。此机制通过设置对象模型（包括 `dict` 等重要内置类型）针对并发访问的隐式安全简化了 CPython 实现。给整个解释器加锁使得解释器多线程运行更方便，其代价则是牺牲了在多处理器上的并行性。

不过，某些标准库或第三方库的扩展模块被设计为在执行计算密集型任务如压缩或哈希时释放 GIL。此外，在执行 I/O 操作时也总是会释放 GIL。

在 Python 3.13 中，GIL 可以使用 `--disable-gil` 构建配置选项来禁用。在使用此选项构建 Python 之后，代码必须附带 `-X gil=0` 或在设置 `PYTHON_GIL=0` 环境变量后运行。此特性将为多线程应用程序启用性能提升并让高效率地使用多核 CPU 更加容易。要了解详情，请参阅 **PEP 703**。

hash-based pyc -- 基于哈希的 pyc
使用对应源文件的哈希值而非最后修改时间来确定其有效性的字节码缓存文件。参见 pyc-invalidation。

hashable -- 可哈希
一个对象如果具有在其生命期内绝不改变的哈希值（它需要有 `__hash__()` 方法），并可以同其他对象进行比较（它需要有 `__eq__()` 方法）就被称为 可哈希对象。可哈希对象必须具有相同的哈希值比较结果才会相等。

可哈希性使得对象能够作为字典键或集合成员使用，因为这些数据结构要在内部使用哈希值。

大多数 Python 中的不可变内置对象都是可哈希的；可变容器（例如列表或字典）都不可哈希；不可变容器（例如元组和 frozenset）仅当它们的元素均为可哈希时才是可哈希的。用户定义类的实例对象默认是可哈希的。它们在比较时一定不相同（除非是与自己比较），它们的哈希值的生成是基于它们的 `id()`。

IDLE
Python 的集成开发与学习环境。idle 是 Python 标准发行版附带的基本编辑器和解释器环境。

immortal -- 永生对象
永生对象是在 **PEP 683** 中引入的 CPython 实现细节。

如果对象属于永生对象，则它的*reference count* 永远不会被修改，因而它在解释器运行期间永远不会被取消分配。例如，`True` 和 `None` 在 CPython 中都属于永生对象。

immutable -- 不可变对象
具有固定值的对象。不可变对象包括数字、字符串和元组。这样的对象不能被改变。如果必须存储一个不同的值，则必须创建新的对象。它们在需要常量哈希值的地方起着重要作用，例如作为字典中的键。

import path -- 导入路径
由多个位置（或路径条目）组成的列表，会被模块的*path based finder* 用来查找导入目标。在导入时，此位置列表通常来自 `sys.path`，但对次级包来说也可能来自上级包的 `__path__` 属性。

importing -- 导入
令一个模块中的 Python 代码能为另一个模块中的 Python 代码所使用的过程。

importer -- 导入器
查找并加载模块的对象；此对象既属于*finder* 又属于*loader*。

interactive -- 交互
Python 带有一个交互式解释器，这意味着你可以在解释器提示符后输入语句和表达式，立即执行并查看其结果。只需不带参数地启动 `python` 命令（也可以在你的计算机主菜单中选择相应菜单项）。在测试新想法或检验模块和包的时候这会非常方便（记住 `help(x)` 函数）。有关交互模式的详情，参见 tut-interac。

interpreted -- 解释型
Python 一是种解释型语言，与之相对的是编译型语言，虽然两者的区别由于字节码编译器的存在而会有所模糊。这意味着源文件可以直接运行而不必显式地创建可执行文件再运行。解释型语言通常具有比编译型语言更短的开发/调试周期，但是其程序往往运行得更慢。参见*interactive*。

interpreter shutdown -- 解释器关闭
当被要求关闭时，Python 解释器将进入一个特殊运行阶段并逐步释放所有已分配资源，例如模块和各种关键内部结构等。它还会多次调用垃圾回收器。这会触发用户定义析构器或弱引用回调中的代码执行。在关闭阶段执行的代码可能会遇到各种异常，因为其所依赖的资源已不再有效（常见的例子有库模块或警告机制等）。

解释器需要关闭的主要原因有 `__main__` 模块或所运行的脚本已完成执行。

iterable -- 可迭代对象
一种能够逐个返回其成员项的对象。可迭代对象的例子包括所有序列类型（如 `list`, `str` 和 `tuple` 等）以及某些非序列类型如 `dict`, 文件对象 以及任何定义了 `__iter__()` 方法或实现了*sequence* 语义的 `__getitem__()` 方法的自定义类的对象。

可迭代对象可被用于 `for` 循环以及许多其他需要一个序列的地方 (`zip()`, `map()`, ...)。当一个可迭代对象作为参数被传给内置函数 `iter()` 时，它会返回该对象的迭代器。这种迭代器适用于对值集合的一次性遍历。在使用可迭代对象时，你通常不需要调用 `iter()` 或者自己处理迭代器对象。`for` 语句会自动为你处理那些操作，创建一个临时的未命名变量用来在循环期间保存迭代器。参见*iterator*, *sequence* 和*generator*。

iterator -- 迭代器
用来表示一连串数据流的对象。重复调用迭代器的 `__next__()` 方法 (或将其传给内置函数 `next()`) 将逐个返回流中的项。当没有数据可用时则将引发 `StopIteration` 异常。到这时迭代器对象中的数据项已耗尽，继续调用其 `__next__()` 方法只会再次引发 `StopIteration`。迭代器必须具有 `__iter__()` 方法用来返回该迭代器对象自身，因此迭代器必定也是可迭代对象，可被用于其他可迭代对象适用的大部分场合。一个显著的例外是那些会多次重复访问迭代项的代码。容器对象 (例如 `list`) 在你每次将其传入 `iter()` 函数或是在 `for` 循环中使用时都会产生一个新的迭代器。如果在此情况下你尝试用迭代器则会返回在之前迭代过程中被耗尽的同一迭代器对象，使其看起来就像是一个空容器。

更多信息可查看 typeiter。

CPython 没有强制推行迭代器定义 `__iter__()` 的要求。还要注意的是自由线程 CPython 并不保证迭代器操作的线程安全性。

key function -- 键函数
键函数或称整理函数，是能够返回用于排序或排位的值的可调用对象。例如，`locale.strxfrm()` 可用于生成一个符合特定区域排序约定的排序键。

Python 中有许多工具都允许用键函数来控制元素的排位或分组方式。其中包括 `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` 以及 `itertools.groupby()`。

有多种方式可以创建一个键函数。例如，`str.lower()` 方法可以用作忽略大小写排序的键函数。或者，键函数也可通过 `lambda` 表达式来创建例如 `lambda r: (r[0], r[2])`。此外，`operator.attrgetter()`, `operator.itemgetter()` 和 `operator.methodcaller()` 是键函数的三个构造器。请查看 排序指引来获取创建和使用键函数的示例。

keyword argument -- 关键字参数
参见*argument*。

lambda
由一个单独*expression* 构成的匿名内联函数，表达式会在调用时被求值。创建 lambda 函数的句法为 `lambda [parameters]: expression`

LBYL
“先查看后跳跃”的英文缩写。这种代码编写风格会在进行调用或查找之前显式地检查前提条件。此风格与*EAFP* 方式恰成对比，其特点是大量使用 `if` 语句。

在多线程环境中，LBYL 方式会导致“查看”和“跳跃”之间发生条件竞争风险。例如，以下代码 `if key in mapping: return mapping[key]` 可能由于在检查操作之后其他线程从 *mapping* 中移除了 *key* 而出错。这种问题可通过加锁或使用 EAFP 方式来解决。

lexical analyzer -- 词法分析器
分词器的正式名称；参见*token*。

list -- 列表
一种 Python 内置*sequence*。虽然名为列表，但它更类似于其他语言中的数组而非链表，因为访问元素的时间复杂度为 *O*(1)。

list comprehension -- 列表推导式
处理一个序列中的所有或部分元素并返回结果列表的一种紧凑写法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 将生成一个 0 到 255 范围内的十六进制偶数对应字符串（0x..）的列表。其中 `if` 子句是可选的，如果省略则 `range(256)` 中的所有元素都会被处理。

loader -- 加载器
负责加载模块的对象。它必须定义 `exec_module()` 和 `create_module()` 方法以实现 `Loader` 接口。加载器通常由一个*finder* 返回。另请参阅：

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

locale encoding -- 语言区域编码格式
在 Unix 上，它是 LC_CTYPE 语言区域的编码格式。它可以通过 `locale.setlocale(locale.LC_CTYPE, new_locale)` 来设置。

在 Windows 上，它是 ANSI 代码页 (如: `"cp1252"`)。

在 Android 和 VxWorks 上，Python 使用 `"utf-8"` 作为语言区域编码格式。

`locale.getencoding()` 可被用来获取语言区域编码格式。

另请参阅*filesystem encoding and error handler*。

magic method -- 魔术方法
special method 的非正式同义词。

mapping -- 映射
一种支持任意键查找并实现了 `collections.abc.Mapping` 或 `collections.abc.MutableMapping` 抽象基类所规定方法的容器对象。此类对象的例子包括 `dict`, `collections.defaultdict`, `collections.OrderedDict` 以及 `collections.Counter`。

meta path finder -- 元路径查找器
sys.meta_path 的搜索所返回的*finder*。元路径查找器与*path entry finders* 存在关联但并不相同。

请查看 importlib.abc.MetaPathFinder 了解元路径查找器所实现的方法。

metaclass -- 元类
一种用于创建类的类。类定义包含类名、类字典和基类列表。元类负责接受上述三个参数并创建相应的类。大部分面向对象的编程语言都会提供一个默认实现。Python 的特别之处在于可以创建自定义元类。大部分用户永远不需要这个工具，但当需要出现时，元类可提供强大而优雅的解决方案。它们已被用于记录属性访问日志、添加线程安全性、跟踪对象创建、实现单例，以及其他许多任务。

更多详情参见 metaclasses。

method -- 方法
在类内部定义的函数。如果作为该类的实例的一个属性来调用，方法将会获取实例对象作为其第一个*argument* (通常命名为 self)。参见*function* 和*nested scope*。

method resolution order -- 方法解析顺序
方法解析顺序就是在查找成员时搜索基类的顺序。请参阅 python_2.3_mro 了解自 2.3 发布版起 Python 解释器所使用算法的详情。

module -- 模块
此对象是 Python 代码的一种组织单位。各模块具有独立的命名空间，可包含任意 Python 对象。模块可通过*importing* 操作被加载到 Python 中。

另见*package*。

module spec -- 模块规格
一个命名空间，其中包含用于加载模块的相关导入信息。是 importlib.machinery.ModuleSpec 的实例。

另请参阅 module-specs。

MRO
参见*method resolution order*。

mutable -- 可变对象
可变对象可以在其 id() 保持固定的情况下改变其取值。另请参见*immutable*。

named tuple -- 具名元组
术语“具名元组”可用于任何继承自元组，并且其中的可索引元素还能使用名称属性来访问的类型或类。这样的类型或类还可能拥有其他特性。

有些内置类型属于具名元组，包括 time.localtime() 和 os.stat() 的返回值。另一个例子是 sys.float_info:

```
>>> sys.float_info[1]                   # 索引访问
1024
>>> sys.float_info.max_exp              # 命名字段访问
1024
>>> isinstance(sys.float_info, tuple)   # 属于元组
True
```

有些具名元组是内置类型（比如上面的例子）。此外，具名元组还可通过常规类定义从 tuple 继承并定义指定名称的字段的方式来创建。这样的类可以手工编号，或者也可以通过继承 typing.NamedTuple，或者使用工厂函数 collections.namedtuple() 来创建。后一种方式还会添加一些手工编写或内置的具名元组所没有的额外方法。

namespace -- 命名空间
命名空间是存放变量的场所。命名空间有局部、全局和内置的，还有对象中的嵌套命名空间（在方法之内）。命名空间通过防止命名冲突来支持模块化。例如，函数 builtins.open 与 os.open() 可通过各自的命名空间来区分。命名空间还通过明确哪个模块实现那个函数来帮助提高可读性和可维护性。例如，random.seed() 或 itertools.islice() 这种写法明确了这些函数是由 random 与 itertools 模块分别实现的。

namespace package -- 命名空间包
一种仅被用作子包的容器的*package*。命名空间包可以没有实体表示物，具体而言就是不同于*regular package* 因为它们没有 `__init__.py` 文件。

命名空间包允许几个可单独安装的包具有共同的父包。在其他情况下，则推荐使用*regular package*。

要了解更多信息，请参阅 **PEP 420** 和 reference-namespace-package。

另可参见*module*。

nested scope -- 嵌套作用域
在一个定义范围内引用变量的能力。例如，在另一函数之内定义的函数可以引用前者的变量。请注意嵌套作用域默认只对引用有效而对赋值无效。局部变量的读写都受限于最内层作用域。类似的，全局变量的读写则作用于全局命名空间。通过 `nonlocal` 关键字可允许写入外层作用域。

new-style class -- 新式类
对目前已被应用于所有类对象的类形式的旧称谓。在较早的 Python 版本中，只有新式类能够使用 Python 新增的更灵活我，如 `__slots__`、描述器、特征属性、`__getattribute__()`、类方法和静态方法等。

object -- 对象
任何具有状态（属性或值）以及预定义行为（方法）的数据。object 也是任何*new-style class* 的最顶层基类名。

optimized scope -- 已优化的作用域
当代码被编译时编译器已充分知晓目标局部变量名称的作用域，这允许对这些名称的读写进行优化。针对函数、生成器、协程、推导式和生成器表达式的局部命名空间都是这样的已优化作用域。注意：大部分解释器优化将应用于所有作用域，只有那些依赖于已知的局部和非局部变量名称集合的优化会仅限于已优化的作用域。

package -- 包
一种可包含子模块或递归地包含子包的 Python *module*。从技术上说，包是具有 `__path__` 属性的 Python 模块。

另参见*regular package* 和*namespace package*。

parameter -- 形参
function（或方法）定义中的命名实体，它指定函数可以接受的一个*argument*（或在某些情况下，多个实参）。有五种形参：

- *positional-or-keyword*：位置或关键字，指定一个可以作为*位置参数* 传入也可以作为*关键字参数* 传入的实参。这是默认的形参类型，例如下面的 *foo* 和 *bar*：

```
def func(foo, bar=None): ...
```

- *positional-only*：仅限位置，指定一个只能通过位置传入的参数。仅限位置形参可通过在函数定义的形参列表中它们之后包含一个 `/` 字符来定义，例如下面的 *posonly1* 和 *posonly2*：

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*：仅限关键字，指定一个只能通过关键字传入的参数。仅限关键字形参可通过在函数定义的形参列表中包含单个可变位置形参或者在多个可变位置形参之前放一个 `*` 来定义，例如下面的 *kw_only1* 和 *kw_only2*：

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*：可变位置，指定可以提供由一个任意数量的位置参数构成的序列（附加在其他形参已接受的位置参数之后）。这种形参可通过在形参名称前加缀 `*` 来定义，例如下面的 *args*：

```
def func(*args, **kwargs): ...
```

- *var-keyword*：可变关键字，指定可以提供任意数量的关键字参数（附加在其他形参已接受的关键字参数之后）。这种形参可通过在形参名称前加缀 `**` 来定义，例如上面的 *kwargs*。

形参可以同时指定可选和必选参数，也可以为某些可选参数指定默认值。

另参见*argument* 术语表条目、参数与形参的区别中的常见问题、`inspect.Parameter` 类、function 一节以及 **PEP 362**。

path entry -- 路径入口
import path 中的一个单独位置，会被*path based finder* 用来查找要导入的模块。

path entry finder -- 路径入口查找器
任一可调用对象使用 `sys.path_hooks` (即*path entry hook*) 返回的*finder*，此种对象能通过*path entry* 来定位模块。

请参看 `importlib.abc.PathEntryFinder` 以了解路径入口查找器所实现的各个方法。

path entry hook -- 路径入口钩子
一种可调用对象，它在知道如何查找特定*path entry* 中的模块的情况下能够使用 `sys.path_hooks` 列表返回一个*path entry finder*。

path based finder -- 基于路径的查找器
默认的一种*元路径查找器*，可在一个*import path* 中查找模块。

path-like object -- 路径类对象
代表一个文件系统路径的对象。路径类对象可以是一个表示路径的 `str` 或者 `bytes` 对象，还可以是一个实现了 `os.PathLike` 协议的对象。一个支持 `os.PathLike` 协议的对象可通过调用 `os.fspath()` 函数转换为 `str` 或者 `bytes` 类型的文件系统路径；`os.fsdecode()` 和 `os.fsencode()` 可被分别用来确保获得 `str` 或 `bytes` 类型的结果。此对象是由 **PEP 519** 引入的。

PEP
“Python 增强提议”的英文缩写。一个 PEP 就是一份设计文档，用来向 Python 社区提供信息，或描述一个 Python 的新增特性及其进度或环境。PEP 应当提供精确的技术规格和所提议特性的原理说明。

PEP 应被作为提出主要新特性建议、收集社区对特定问题反馈以及为必须加入 Python 的设计决策编写文档的首选机制。PEP 的作者有责任在社区内部建立共识，并应将不同意见也记入文档。

参见 **PEP 1**。

portion -- 部分
构成一个命名空间包的单个目录内文件集合（也可能存放于一个 zip 文件内），具体定义见 **PEP 420**。

positional argument -- 位置参数
参见*argument*。

provisional API -- 暂定 API
暂定 API 是指被有意排除在标准库的向后兼容性保证之外的应用编程接口。虽然此类接口通常不会再有重大改变，但只要其被标记为暂定，就可能在核心开发者确定有必要的情况下进行向后不兼容的更改（甚至包括移除该接口）。此种更改并不会随意进行 -- 仅在 API 被加入之前未考虑到的严重基础性缺陷被发现时才可能会这样做。

即便是对暂定 API 来说，向后不兼容的更改也会被视为“最后的解决方案”——任何问题被确认时都会尽可能先尝试找到一种向后兼容的解决方案。

这种处理过程允许标准库持续不断地演进，不至于被有问题的长期性设计缺陷所困。详情见 **PEP 411**。

provisional package -- 暂定包
参见*provisional API*。

Python 3000
Python 3.x 发布路线的昵称（这个名字在版本 3 的发布还遥遥无期的时候就已出现了）。有时也被缩写为“Py3k”。

Pythonic
指一个思路或一段代码紧密遵循了 Python 语言最常用的风格和理念，而不是使用其他语言中通用的概念来实现代码。例如，Python 的常用风格是使用 `for` 语句循环来遍历一个可迭代对象中的所有元素。许多其他语言没有这样的结构，因此不熟悉 Python 的人有时会选择使用一个数字计数器:

```
for i in range(len(food)):
    print(food[i])
```

而相应的更简洁更 Pythonic 的方法是这样的:

```
for piece in food:
    print(piece)
```

qualified name -- 限定名称
一个以点号分隔的名称，显示从模块的全局作用域到该模块中定义的某个类、函数或方法的“路径”，相关定义见 **PEP 3155**。对于最高层级的函数和类，限定名称与对象名称一致:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

当被用于引用模块时，完整限定名称意为标示该模块的以点号分隔的整个路径，其中包含其所有的父包，例如 `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count -- 引用计数
指向某个对象的引用的数量。当一个对象的引用计数降为零时，它就会被释放。特殊的*immortal* 对象具有永远不会被修改的引用计数，因此这种对象永远不会被释放。引用计数对 Python 代码来说通常是不可见的，但它是*CPython* 实现的一个关键元素。程序员可以调用 `sys.getrefcount()` 函数来返回特定对象的引用计数。

In *CPython*, reference counts are not considered to be stable or well-defined values; the number of references to an object, and how that number is affected by Python code, may be different between versions.

regular package -- 常规包
传统型的*package*，例如包含有一个 `__init__.py` 文件的目录。

另参见*namespace package*。

REPL
“读取-求值-打印循环” read-eval-print loop 的缩写，*interactive* 解释器 shell 的另一个名字。

__slots__
一种写在类内部的声明，通过预先声明实例属性等对象并移除实例字典来节省内存。虽然这种技巧很流行，但想要用好却并不容易，最好是只保留在少数情况下采用，例如极耗内存的应用程序，并且其中包含大量实例。

sequence -- 序列
一种*iterable*，它支持通过 `__getitem__()` 特殊方法来使用整数索引进行高效的元素访问，并定义了一个返回序列长度的 `__len__()` 方法。内置序列类型有 `list`, `str`, `tuple` 和 `bytes` 等。请注意虽然 `dict` 也支持 `__getitem__()` 和 `__len__()`，但它被归类为映射而非序列，因为它使用任意的*hashable* 键而不是整数来查找元素。

`collections.abc.Sequence` 抽象基类定义了一个更丰富的接口，它在 `__getitem__()` 和 `__len__()` 之外，还添加了 `count()`, `index()`, `__contains__()` 和 `__reversed__()`。实现此扩展接口的类型可以使用 `register()` 来显式地注册。要获取有关通用序列方法的更多文档，请参阅 通用序列操作。

set comprehension -- 集合推导式
处理一个可迭代对象中的所有或部分元素并返回结果集合的一种紧凑写法。`results = {c for c in 'abracadabra' if c not in 'abc'}` 将生成字符串集合 `{'r', 'd'}`。参见 comprehensions。

single dispatch -- 单分派
一种*generic function* 分派形式，其实现是基于单个参数的类型来选择的。

slice -- 切片
通常只包含了特定*sequence* 的一部分的对象。切片是通过使用下标标记来创建的，在 `[]` 中给出几个以冒号分隔的数字，例如 `variable_name[1:3:5]`。方括号（下标）标记在内部使用 `slice` 对象。

soft deprecated -- 软弃用
被软弃用的 API 不应在新编写的代码中使用，但在已有代码中使用仍是安全的。这样的 API 将保留在文档中并被测试，但不会再获得进一步的功能增强。

与普通的弃用不同，软弃用没有 API 移除计划也不会发出弃用警告。

参见 PEP 387: Soft Deprecation。

special method -- 特殊方法
一种由 Python 隐式调用的方法，用来对某个类型执行特定操作例如相加等等。这种方法的名称的首尾都为双下划线。特殊方法的文档参见 specialnames。

standard library -- 标准库
作为官方 Python 解释器软件包的组成部分一同发布的包, 模块 和扩展模块 集合。该集合成员的实际内容可能因系统平台、可用系统库或其他条件的不同而发生变化。相关文档可在 library-index 查看。

另请参阅 `sys.stdlib_module_names` 获取所有可用标准库模块名称的列表。

statement -- 语句
语句是程序段（一个代码“块”）的组成单位。一条语句可以是一个*expression* 或某个带有关键字的结构，例如 `if`、`while` 或 `for`。

static type checker -- 静态类型检查器
读取 Python 代码并进行分析，以查找问题例如拼写错误的外部工具。另请参阅类型提示 以及 `typing` 模块。

stdlib -- 标准库
standard library 的缩写。

strong reference -- 强引用
在 Python 的 C API 中，强引用是指为持有引用的代码所拥有的对象的引用。在创建引用时可通过调用*`Py_INCREF()`* 来获取强引用而在删除引用时可通过*`Py_DECREF()`* 来释放它。

`Py_NewRef()` 函数可被用于创建一个对象的强引用。通常，必须在退出某个强引用的作用域时在该强引用上调用*`Py_DECREF()`* 函数，以避免引用的泄漏。

另请参阅*borrowed reference*。

text encoding -- 文本编码格式
在 Python 中，一个字符串是一串 Unicode 代码点（范围为 `U+0000--U+10FFFF`）。为了存储或传输一个字符串，它需要被序列化为一串字节。

将一个字符串序列化为一个字节序列被称为“编码”，而从字节序列中重新创建字符串被称为“解码”。

有各种不同的文本序列化 编码器，它们被统称为” 文本编码格式”。

text file -- 文本文件
一种能够读写 `str` 对象的*file object*。通常一个文本文件实际是访问一个面向字节的数据流并自动处理*text encoding*。文本文件的例子包括以文本模式（`'r'` 或 `'w'`）打开的文件、`sys.stdin`、`sys.stdout` 以及 `io.StringIO` 的实例。

另请参看*binary file* 了解能够读写字节型对象 的文件对象。

形符
一个源代码小单元，由 词法分析器 (或称 分词器) 生成。名称、数字、字符串、运算符、换行符等均由词元来表示。

`tokenize` 模块对外暴露了 Python 的词法分析器。`token` 模块包含了有关各种词元类型的信息。

triple-quoted string -- 三引号字符串
首尾各带三个连续双引号（"）或者单引号（'）的字符串。它们在功能上与首尾各用一个引号标注的字符串没有什么不同，但是有多种用处。它们允许你在字符串内包含未经转义的单引号和双引号，并且可以跨越多行而无需使用连接符，在编写文档字符串时特别好用。

type -- 类型
Python 对象的类型决定它属于什么种类；每个对象都具有特定的类型。对象的类型可通过其 `__class__` 属性来访问或是用 `type(obj)` 来获取。

type alias -- 类型别名
一个类型的同义词，创建方式是把类型赋值给特定的标识符。

类型别名的作用是简化*类型注解*。例如:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

可以这样提高可读性:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

参见 `typing` 和 **PEP 484**，其中有对此功能的详细描述。

type hint -- 类型注解
annotation 为变量、类属性、函数的形参或返回值指定预期的类型。

类型提示是可选的而不是 Python 的强制要求，但它们对*静态类型检查器* 很有用处。它们还能协助 IDE 实现代码补全与重构。

全局变量、类属性和函数的类型注解可以使用 `typing.get_type_hints()` 来访问，但局部变量则不可以。

参见 `typing` 和 **PEP 484**，其中有对此功能的详细描述。

universal newlines -- 通用换行
一种解读文本流的方式，将以下所有符号都识别为行结束标志：Unix 的行结束约定 `'\n'`、Windows 的约定 `'\r\n'` 以及旧版 Macintosh 的约定 `'\r'`。参见 **PEP 278** 和 **PEP 3116** 和 `bytes.splitlines()` 了解更多用法说明。

variable annotation -- 变量标注
对变量或类属性的*annotation*。

在标注变量或类属性时，还可选择为其赋值:

```
class C:
    field: 'annotation'
```

变量标注通常被用作*类型提示*：例如以下变量预期接受 `int` 类型的值:

```
count: int = 0
```

变量标注语法的详细解释见 annassign 一节。

参见*function annotation*, **PEP 484** 和 **PEP 526**，其中描述了此功能。另请参阅 annotations-howto 以了解使用标注的最佳实践。

virtual environment -- 虚拟环境
一种采用协作式隔离的运行时环境，允许 Python 用户和应用程序在安装和升级 Python 分发包时不会干扰到同一系统上运行的其他 Python 应用程序的行为。

另参见 `venv`。

virtual machine -- 虚拟机
一台完全通过软件定义的计算机。Python 虚拟机可执行字节码编译器所生成的*bytecode*。

海象运算符
A light-hearted way to refer to the assignment expression operator `:=` because it looks a bit like a walrus if you turn your head.

Zen of Python -- Python 之禅
列出 Python 设计的原则与哲学，有助于理解与使用这种语言。查看其具体内容可在交互模式提示符中输入"`import this`"。

APPENDIX B

关于本文档

Python 的文档是用 Sphinx 从 reStructuredText 源生成的，*Sphinx* 是一个专为处理 Python 文档而编写的文档生成器。

本文档及其工具链之开发，皆在于志愿者之努力，亦恰如 Python 本身。如果您想为此作出贡献，请阅读 reporting-bugs 了解如何参与。我们随时欢迎新的志愿者！

特别鸣谢：

- Fred L. Drake, Jr.，原始 Python 文档工具集之创造者，众多文档之作者；
- 用于创建 reStructuredText 和 Docutils 套件的 Docutils 项目；
- Fredrik Lundh 的 Alternative Python Reference 项目，为 Sphinx 提供许多好的点子。

B.1 Python 文档的贡献者

有很多对 Python 语言，Python 标准库和 Python 文档有贡献的人，随 Python 源代码分发的 Misc/ACKS 文件列出了部分贡献者。

有了 Python 社区的输入和贡献，Python 才有了如此出色的文档——谢谢你们！

APPENDIX C

历史和许可证

C.1 该软件的历史

Python 是在 1990 年代初由 Guido van Rossum 在荷兰的 Stichting Mathematisch Centrum (CWI，见 https://www.cwi.nl) 作为一门名为 ABC 的语言的后继者创造的。Guido 仍然是 Python 的主要作者，尽管它也包括了许多来自其他人的贡献。

在 1995 年，Guido 在弗吉尼亚州 Reston 市的 Corporation for National Research Initiatives (CNRI，见 https://www.cnri.reston.va.us) 继续他对 Python 的工作，他在那里发布了该软件的多个版本。

在 2000 年 5 月，Guido 和 Python 核心开发团队移至 BeOpen.com 组建了 BeOpen PythonLabs 团队。同年 10 月，PythonLabs 团队移至 Digital Creations，后改名为 Zope Corporation。在 2001 年，Python Software Foundation (PSF，见 https://www.python.org/psf/) 成立，这是一个专门为持有与 Python 相关的知识产权而创立的非营利组织。Zope Corporation 是 PSF 的一个赞助成员。

所有 Python 发布版都是开源的 (有关开源的定义见 https://opensource.org)。在历史上，大多数但并非全部 Python 发布版还是 GPL 兼容的；下表总结了各个版本的情况。

发布版本	源自	年份	所有者	GPL 兼容? (1)
0.9.0 至 1.2	n/a	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	是 (2)
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 及更高	2.1.1	2001 至今	PSF	是

备注

(1) GPL 兼容并不意味着我们是基于 GPL 发布 Python。与 GPL 不同，所有 Python 许可证都允许你分发经修改的版本而无需开源你所做的修改。GPL 兼容的许可证使得 Python 可以与其他基于

GPL 发布的软件结合使用；其他许可证则不可以。

(2) 按照 Richard Stallman 的说法，1.6.1 不是 GPL 兼容的，因为它的许可证包含特定的法律条款。但是按照 CNRI 的说法，Stallman 的律师已告诉 CNRI 的律师 1.6.1 与 GPL " 并非不兼容"。

感谢众多在 Guido 指导下工作的外部志愿者，使得这些发布成为可能。

C.2 获取或以其他方式使用 Python 的条款和条件

Python 软件和文档的使用许可均为 Python Software Foundation License Version 2。

从 Python 3.8.6 开始，文档中的示例、操作指导和其他代码均采用 PSF License Version 2 和*Zero-Clause BSD license* 双重使用许可。

某些包含在 Python 中的软件基于不同的许可。这些许可会与相应许可之下的代码一同列出。有关这些许可的不完整列表请参阅收录软件的许可与鸣谢。

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to Python.

4. PSF is making Python available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This License
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

8. By copying, installing or otherwise using Python, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 用于 PYTHON 2.0 的 BEOPEN.COM 许可协议

BEOPEN PYTHON 开源许可协议第 1 版

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 用于 PYTHON 1.6.1 的 CNRI 许可协议

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
```

(续下页)

(接上页)

```
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 用于 PYTHON 0.9.0 至 1.2 的 CWI 许可协议

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(续下页)

(接上页)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 收录软件的许可与鸣谢

本节是 Python 发行版中收录的第三方软件的许可和致谢清单，该清单是不完整且不断增长的。

C.3.1 Mersenne Twister

作为 `random` 模块下层的 `_random` C 扩展包括基于从 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 下载的代码。以下是原始代码的完整注释:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
```

(续下页)

(接上页)

```
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 套接字

`socket` 使用了 `getaddrinfo()` 和 `getnameinfo()` WIDE 项目的不同源文件中: https://www.wide.ad.jp/

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 异步套接字服务

`test.support.asynchat` 和 `test.support.asyncore` 模块包含以下说明。:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
```

(续下页)

(接上页)

```
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie 管理

`http.cookies` 模块包含以下声明:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 执行追踪

`trace` 模块包含以下声明:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
```

(续下页)

(接上页)

```
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 与 UUdecode 函数

uu 编解码器包含以下声明:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 远程过程调用

xmlrpc.client 模块包含以下声明:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(续下页)

(接上页)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test.test_epoll` 模块包含以下说明:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` 模块关于 kqueue 的接口包含以下声明:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

Python/pyhash.c 文件包含 Marek Majkowski' 对 Dan Bernstein 的 SipHash24 算法的实现。它包含以下声明:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 和 dtoa

Python/dtoa.c 文件提供了 C 函数 dtoa 和 strtod，用于 C 双精度数值和字符串之间的转换，它派生自由 David M. Gay 编写的同名文件。目前该文件可在 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 访问。在 2009 年 3 月 16 日检索到的原始文件包含以下版权和许可声明:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

如果操作系统有支持则 hashlib, posix 和 ssl 会使用 OpenSSL 库来提升性能。此外，Python 的 Windows 和 macOS 安装程序可能会包括 OpenSSL 库的副本，所以我们也在此包括一份 OpenSSL 许可证的副本。对于 OpenSSL 3.0 发布版，及其后续衍生版本，均使用 Apache License v2:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/
```

(续下页)

(接上页)

```
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.
```

(续下页)

(接上页)

```
2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
```

(续下页)

(接上页)

```
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

pyexpat 扩展是使用所包括的 expat 源副本来构建的，除非配置了 --with-system-expat:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
```

(续下页)

(接上页)

```
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

作为 `ctypes` 模块下层的 `_ctypes` C 扩展是使用包括了 libffi 源的副本构建的，除非构建时配置了 `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

如果系统上找到的 zlib 版本太旧而无法用于构建，则使用包含 zlib 源代码的拷贝来构建 `zlib` 扩展:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
```

(续下页)

(接上页)

```
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

tracemalloc 使用的哈希表的实现基于 cfuhash 项目:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

作为 decimal 模块下层的 _decimal C 扩展是使用包括了 libmpdec 库的副本构建的，除非构建时配置了 --with-system-libmpdec:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
```

(续下页)

(接上页)

```
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 测试套件

`test` 包中的 C14N 2.0 测试集 (`Lib/test/xmltestdata/c14n-20/`) 提取自 W3C 网站 https://www.w3.org/TR/xml-c14n2-testcases/ 并根据 3 条款版 BSD 许可证发行:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT 许可证:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
```

(续下页)

(接上页)

```
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 asyncio

`asyncio` 模块的某些部分来自 uvloop 0.16，它是基于 MIT 许可证发行的:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

文件 `Python/qsbr.c` 改编自 subr_smr.c 中 FreeBSD 的"Global Unbounded Sequences" 安全内存回收方案。该文件是基于 2 条款 BSD 许可证分发的:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR
```

(续下页)

(接上页)

```
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

APPENDIX D

版权所有

Python 与这份文档：

有关完整的许可证和许可信息，请参见历史和许可证。

Bibliography

[win] `PyExc_WindowsError` 仅在 Windows 上定义；通过测试是否定义了预处理器宏 `MS_WINDOWS` 来保护用到它的代码。

索引

非字母

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z