# 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。

## 语言版本兼容性

Python的 C API 与 C11 和 C++11 版本的 C 和 C++ 兼容。

这是一个下限：C API 不需要以后 C/C++ 版本的特性 。您 *不* 需要启用编译器的 "c11 模式"。

## 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。

## 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。

## 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`、`PyMODINIT_FUNC`）。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。

**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}
};
```

# 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 标准类型层级结构 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。

## 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);
        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;
}
```

### 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。

## 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：

```c
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)
        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。

## 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。

## 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` 。

## 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。