1. 使用 C 或 C++ 扩展 Python

如果你会用 C，添加新的 Python 内置模块会很简单。以下两件不能用 Python 直接做的事，可以通过 *extension modules* 来实现：实现新的内置对象类型；调用 C 的库函数和系统调用。

为了支持扩展，Python API（应用程序编程接口）定义了一系列函数、宏和变量，可以访问 Python 运行时系统的大部分内容。Python 的 API 可以通过在一个 C 源文件中引用 `"Python.h"` 头文件来使用。

扩展模块的编写方式取决与你的目的以及系统设置；下面章节会详细介绍。

> **备注：** C扩展接口特指CPython，扩展模块无法在其他Python实现上工作。在大多数情况下，应该避免写C扩展，来保持可移植性。举个例子，如果你的用例调用了C库或系统调用，你应该考虑使用 `ctypes` 模块或 cffi 库，而不是自己写C代码。这些模块允许你写Python代码来接口C代码，并且相较于编写和编译C扩展模块，该方法在不同Python实现之间具有更高的可移植性。

1.1. 一个简单的例子

让我们创建一个扩展模块 `spam` (Monty Python 粉丝最喜欢的食物...) 并且想要创建对应 C 库函数 `system()` [1] 的 Python 接口。 这个函数接受一个以 null 结尾的字符串参数并返回一个整数。 我们希望可以在 Python 中以如下方式调用此函数:

```
>>> import spam
>>> status = spam.system("ls -l")
```

首先创建一个 `spammodule.c` 文件。 （传统上，如果一个模块叫 `spam`，则对应实现它的 C 文件叫 `spammodule.c`；如果这个模块名字非常长，比如 `spammify`，则这个模块的文件可以直接叫 `spammify.c`。）

文件中开始的两行是：

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

这会导入 Python API（如果你喜欢，你可以在这里添加描述模块目标和版权信息的注释）。

> **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`。
>
> `#define PY_SSIZE_T_CLEAN` 被用来指明 `Py_ssize_t` 应当在某些 API 中代替 `int` 使用。 它从 Python 3.13 起已不再需要，但我们保留它用于向下兼容。 请参阅 字符串和缓存区 获取该宏的描述。

由 `Python.h` 定义的用户可见的符号都带有 `Py` 或 `PY` 前缀，只有在标准头文件中定义的符号除外。

小技巧: 为保持向下兼容，`Python.h` 包括了一些标准头文件。 C 扩展应当包括它们要使用的标准头文件，而不应依赖这些隐式的包括。 如果使用受限 C API 3.13 版或更新的版本，隐式的包括如下：

- `<assert.h>`
- `<intrin.h>` (在 Windows 上)
- `<inttypes.h>`
- `<limits.h>`
- `<math.h>`
- `<stdarg.h>`
- `<wchar.h>`
- `<sys/types.h>` (如果提供)

如果未定义 `Py_LIMITED_API`，或是设为 3.12 版或更旧的版本，则还将包括如下头文件：

- `<ctype.h>`
- `<unistd.h>` (在 POSIX 上)

如果未定义 `Py_LIMITED_API`，或是设为 3.10 版或更旧的版本，则还将包括如下头文件：

- `<errno.h>`
- `<stdio.h>`
- `<stdlib.h>`
- `<string.h>`

下面添加C函数到扩展模块，当调用 `spam.system(string)` 时会做出响应，(我们稍后会看到调用)：

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return PyLong_FromLong(sts);
}
```

有个直接翻译参数列表的方法(举个例子，单独的 `"ls -l"`)到要传递给C函数的参数。C函数总是有两个参数，通常名字是 *self* 和 *args* 。

对模块级函数， *self* 参数指向模块对象；对于方法则指向对象实例。

args 参数是指向一个 Python 的 tuple 对象的指针，其中包含参数。 每个 tuple 项对应一个调用参数。 这些参数也全都是 Python 对象 --- 要在我们的 C 函数中使用它们就需要先将其转换为 C 值。 Python API 中的函数 `PyArg_ParseTuple()` 会检查参数类型并将其转换为 C 值。 它使用模板字符串确定需要的参数类型以及存储被转换的值的 C 变量类型。 细节将稍后说明。

`PyArg_ParseTuple()` 在所有参数都有正确类型且组成部分按顺序放在传递进来的地址里时，返回真(非零)。其在传入无效参数时返回假(零)。在后续例子里，还会抛出特定异常，使得调用的函数可以理解返回 `NULL` (也就是例子里所见)。

1.2. 关于错误和异常

整个 Python 解释器系统有一个如下所述的重要惯例：当一个函数运行失败时，它应当设置一个异常条件并返回一个错误值（通常为 `-1` 或 `NULL` 指针）。 异常信息保存在解释器线程状态的三个成员中。 如果没有异常则它们的值为 `NULL`。 在其他情况下它们是 `sys.exc_info()` 所返回的 Python 元组的成员的 C 对应物。 它们分别是异常类型、异常实例和回溯对象。 理解它们对于理解错误是如何被传递的非常重要。

Python API中定义了一些函数来设置这些变量。

最常用的就是 `PyErr_SetString()`。 其参数是异常对象和 C 字符串。 异常对象一般是像 `PyExc_ZeroDivisionError` 这样的预定义对象。 C 字符串指明异常原因，并被转换为一个 Python 字符串对象存储为异常的"关联值"。

另一个有用的函数是 `PyErr_SetFromErrno()` ，仅接受一个异常对象，异常描述包含在全局变量 `errno` 中。最通用的函数还是 `PyErr_SetObject()` ，包含两个参数，分别为异常对象和异常描述。你不需要使用 `Py_INCREF()` 来增加传递到其他函数的参数对象的引用计数。

你可以通过 `PyErr_Occurred()` 在不造成破坏的情况下检测是否设置了异常。 这将返回当前异常对象，或者如果未发生异常则返回 `NULL`。 你通常不需要调用 `PyErr_Occurred()` 来查看函数调用中是否发生了错误，因为你应该能从返回值中看出来。

当一个函数 *f* 调用另一个函数 *g* 时检测到后者出错了，*f* 应当自己返回一个错误值 (通常为 `NULL` 或 `-1`)。 它 *不应* 调用某个 `PyErr_*` 函数 --- 这类函数已经被 *g* 调用过了。 *f* 的调用者随后也应当返回一个错误来提示 *它的* 调用者，同样 *不应* 调用 `PyErr_*`，依此类推 --- 错误的最详细原因已经由首先检测到它的函数报告了。 一旦这个错误到达 Python 解释器的主循环，它会中止当前执行的 Python 代码并尝试找出由 Python 程序员所指定的异常处理器。

（在某些情况下模块确实能够通过调用其它 `PyErr_*` 函数来给出更为详细的错误消息，并且在这些情况下是可以这样做的。 但是按照一般规则，这是不必要的，并可能导致有关错误的信息丢失：大多数操作会由于种种原因而失败。）

想要忽略由一个失败的函数调用所设置的异常，异常条件必须通过调用 `PyErr_Clear()` 显式地被清除。 C 代码应当调用 `PyErr_Clear()` 的唯一情况是如果它不想将错误传给解释器而是想完全由自己来处理它（可能是尝试其他方法，或是假装没有出错）。

每次失败的 `malloc()` 调用必须转换为一个异常。 `malloc()` (或 `realloc()`)的直接调用者必须调用 `PyErr_NoMemory()` 来返回错误来提示。所有对象创建函数(例如 `PyLong_FromLong()`)已经这么

做了，所以这个提示仅用于直接调用 `malloc()` 的情况。

还要注意的是，除了 `PyArg_ParseTuple()` 等重要的例外，返回整数状态码的函数通常都是返回正值或零来表示成功，而以 `-1` 表示失败，如同 Unix 系统调用一样。

最后，当你返回一个错误指示器时要注意清理垃圾（通过为你已经创建的对象执行 `Py_XDECREF()` 或 `Py_DECREF()` 调用）！

选择引发哪个异常完全取决于你的喜好。 所有 Python 内置异常都有对应的预声明 C 对象，例如 `PyExc_ZeroDivisionError`，你可以直接使用它们。 当然，你应当明智地选择异常 --- 不要使用 `PyExc_TypeError` 来表示文件无法打开（可能应该用 `PyExc_OSError` 比较好）。 如果参数列表有问题，`PyArg_ParseTuple()` 函数通常会引发 `PyExc_TypeError`。 如果你希望一个参数的值必须在特定范围内或必须满足其他条件，则适宜使用 `PyExc_ValueError`。

你也可以定义你的模块独有的新异常。 做到这点的最简单方式是在文件的开头声明一个静态全局对象变量:

```
static PyObject *SpamError = NULL;
```

并通过在模块的 `Py_mod_exec` 函数 (`spam_module_exec()`) 中调用 `PyErr_NewException()` 来初始化它:

```
SpamError = PyErr_NewException("spam.error", NULL, NULL);
```

由于 `SpamError` 是一个全局变量，它将在模块每次重新初始化时被覆盖，即当 `Py_mod_exec` 函数被调用时。

在目前，让我们避免这个问题：我们将通过引发 `ImportError` 来阻止重复的初始化:

```
static PyObject *SpamError = NULL;

static int
spam_module_exec(PyObject *m)
{
    if (SpamError != NULL) {
        PyErr_SetString(PyExc_ImportError,
                        "cannot initialize spam module more than once");
        return -1;
    }
    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    if (PyModule_AddObjectRef(m, "SpamError", SpamError) < 0) {
        return -1;
    }

    return 0;
}

static PyModuleDef_Slot spam_module_slots[] = {
    {Py_mod_exec, spam_module_exec},
    {0, NULL}
};
```

```
static struct PyModuleDef spam_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "spam",
    .m_size = 0,  // non-negative
    .m_slots = spam_module_slots,
};

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

请注意该异常对象的 Python 名称为 spam.error。 PyErr_NewException() 函数可以创建基类为 Exception 的类 (除非传入了另一个类而不是 NULL)，如 内置异常 中所描述的。

请注意 SpamError 变量保留了一个对新创建的异常类的引用；这是有意为之的！ 由于异常可能会被外部代码从模块中删除，因此需要拥有一个对该类的引用以确保它不会被丢弃，从而导致 SpamError 成为一个悬空指针。 如果异常类成为悬空指针，则引发该异常的 C 代码可能会导致核心转储或其他预期之外的附带影响。

在目前，用于移除该引用的 Py_DECREF() 调用是缺失的。 即使是在 Python 解释器关闭时，全局变量 SpamError 也不会被当作垃圾回收。 它将会"泄漏"。 不过，我们确实能保证这在每个进程中最多发生一次。

本样例稍后将讨论 PyMODINIT_FUNC 作为函数返回类型的用法。

可在扩展模块中调用 PyErr_SetString() 来引发 spam.error 异常，如下所示:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    if (sts < 0) {
        PyErr_SetString(SpamError, "System command failed");
        return NULL;
    }
    return PyLong_FromLong(sts);
}
```

1.3. 回到例子

回到前面的例子，你应该明白下面的代码:

```
if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
```

如果在参数列表中检测到错误，它将返回 `NULL` (该值是返回对象指针的函数的错误提示)，这取决于 `PyArg_ParseTuple()` 设置的异常。 在其他情况下参数的字符串值会被拷贝到局部变量 `command`。 这是一个指针赋值并且你不应该修改它所指向的字符串 (因此在标准 C 中，变量 `command` 应当被正确地声明为 `const char *command`)。

下一个语句使用UNIX系统函数 `system()` ，传递给他的参数是刚才从 `PyArg_ParseTuple()` 取出的:

```
sts = system(command);
```

我们的 `spam.system()` 函数必须以 Python 对象的形式返回 `sts` 的值。 这是通过使用函数 `PyLong_FromLong()` 完成的。

```
return PyLong_FromLong(sts);
```

在这种情况下，会返回一个整数对象，(这个对象会在Python堆里面管理)。

如果你有一个不返回有用参数的 C 函数（即返回 `void` 的函数），则对应的 Python 函数必须返回 `None`。 你必须使用这种写法（它是通过 `Py_RETURN_NONE` 宏来实现的）

```
Py_INCREF(Py_None);
return Py_None;
```

`Py_None` 是特殊 Python 对象 `None` 所对应的 C 名称。 它是一个真正的 Python 对象而不是 `NULL` 指针，如我们所见，后者在大多数上下文中都意味着“错误”。

1.4. 模块方法表和初始化函数

我承诺过要向大家展示如何从 Python 程序中调用 `spam_system()`。 首先，我们需要在“方法表”中列出它的名称和地址:

```
static PyMethodDef spam_methods[] = {
    ...
    {"system",  spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

注意第三个参数（`METH_VARARGS`），这个标志指定会使用C的调用惯例。可选值有 `METH_VARARGS` 、 `METH_VARARGS | METH_KEYWORDS` 。值 `0` 代表使用 `PyArg_ParseTuple()` 的陈旧变量。

如果单独使用 `METH_VARARGS` ，函数会等待Python传来tuple格式的参数，并最终使用 `PyArg_ParseTuple()` 进行解析。

如果应当将关键字参数传给该函数则可以在第三个字段中设置 `METH_KEYWORDS` 比特位。 在此情况下，C 函数应当接受第三个 `PyObject *` 形参，它将为一个由关键字组成的字典。 使用 `PyArg_ParseTupleAndKeywords()` 来将参数解析为函数。

这个方法表必须被模块定义结构所引用。

```
static struct PyModuleDef spam_module = {
    ...
    .m_methods = spam_methods,
    ...
};
```

这个结构体必须在模块的初始化函数中传递给解释器。 初始化函数必须命名为 `PyInit_name()`，其中 *name* 是模块的名称，并且应该是模块文件中定义的唯一非 `static` 条目:

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

请注意 `PyMODINIT_FUNC` 将函数声明为 `PyObject *` 返回类型，声明了平台所要求的任何特殊链接声明，并针对于= C++ 将函数声明为 `extern "C"`。

`PyInit_spam()` 会在每个解释器首次导入其 `spam` 模块时被调用。 （请参看下文中有关嵌入式 Python 的说明。） 必须通过 `PyModuleDef_Init()` 返回一个指向模块定义的指针，以便导入机制能够创建该模块并将其保存到 `sys.modules` 中。

当嵌入 Python 时，`PyInit_spam()` 函数将不会被自动调用除非在 `PyImport_Inittab` 表中有条目。 要将模块添加到初始化表中，请使用 `PyImport_AppendInittab()`，并可选择随后导入该模块:

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

int
main(int argc, char *argv[])
{
    PyStatus status;
    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* 添加一个内置模块，在 Py_Initialize 之前 */
    if (PyImport_AppendInittab("spam", PyInit_spam) == -1) {
        fprintf(stderr, "Error: could not extend in-built modules table\n");
        exit(1);
    }

    /* 将 argv[0] 传给 Python 解释器 */
    status = PyConfig_SetBytesString(&config, &config.program_name, argv[0]);
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    /* 初始化 Python 解释器。 必需的操作。
       如果此步骤失败，将导致致命错误。 */
    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);
```

```
    /* 可以选择导入模块，或是作为替代，
       导入可以被延迟直到由嵌入的脚本
       来导入它。 */
    PyObject *pmodule = PyImport_ImportModule("spam");
    if (!pmodule) {
        PyErr_Print();
        fprintf(stderr, "Error: could not import module 'spam'\n");
    }

    // ... 在此使用 Python C API ...

    return 0;

  exception:
     PyConfig_Clear(&config);
     Py_ExitStatusException(status);
}
```

备注： 如果你声明一个全局变量或局部静态变量，模块可能在重新初始化时出现预料之外的附带影响，例如在从 `sys.modules` 中移除条目或将已编译的模块导入到一个进程中的多个解释器（或者在未干预 `exec()` 的情况下执行 `fork()` 之后）的时候。 如果模块状态没有被完全 隔离，开发者应当考虑将模块标记为不支持子解释器 (通过 `Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED`)。

在 Python 源代码发布包的 `Modules/xxlimited.c` 中包括了一个更详细的示例。 此文件可被用作代码模板或是学习样例。

1.5. 编译和链接

在你能使用你的新写的扩展之前，你还需要做两件事情：使用 Python 系统来编译和链接。如果你使用动态加载，这取决于你使用的操作系统的动态加载机制；更多信息请参考编译扩展模块的章节（构建 C/C++ 扩展 章节），以及在 Windows 上编译需要的额外信息（在 Windows 上构建 C 和 C++ 扩展 章节）。

如果你不使用动态加载，或者想要让模块永久性的作为Python解释器的一部分，就必须修改配置设置，并重新构建解释器。幸运的是在Unix上很简单，只需要把你的文件（`spammodule.c` 为例）放在解压缩源码发行包的 `Modules/` 目录下，添加一行到 `Modules/Setup.local` 来描述你的文件：

```
spam spammodule.o
```

然后在顶层目录运行 **make** 来重新构建解释器。你也可以在 `Modules/` 子目录使用 **make**，但是你必须先重建 `Makefile` 文件，然后运行 '**make** Makefile' 命令。（你每次修改 `Setup` 文件都需要这样操作。）

如果你的模块需要额外的链接，这些内容可以列出在配置文件里，举个实例：

```
spam spammodule.o -lX11
```

1.6. 在C中调用Python函数

迄今为止，我们一直把注意力集中于让Python调用C函数，其实反过来也很有用，就是用C调用Python函数。这在回调函数中尤其有用。如果一个C接口使用回调，那么就要实现这个回调机制。

幸运的是，Python解释器是比较方便回调的，并给标准Python函数提供了标准接口。(这里就不再详述解析Python字符串作为输入的方式，如果有兴趣可以参考 `Python/pythonmain.c` 中的 `-c` 命令行代码。)

调用Python函数很简单，首先Python程序要传递Python函数对象。应该提供个函数(或其他接口)来实现。当调用这个函数时，用全局变量保存Python函数对象的指针，还要调用 (`Py_INCREF()`) 来增加引用计数，当然不用全局变量也没什么关系。举个例子，如下函数可能是模块定义的一部分：

```
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable");
            return NULL;
        }
        Py_XINCREF(temp);         /* 添加一个指向新回调的引用 */
        Py_XDECREF(my_callback);  /* 丢弃之前的回调 */
        my_callback = temp;       /* 记住新的回调 */
        /* 返回 "None" 的样例 */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
```

此函数必须使用 `METH_VARARGS` 旗标注册到解释器；这将在 模块方法表和初始化函数 一节中详细描述。 `PyArg_ParseTuple()` 函数及其参数的文档见 提取扩展函数的参数 一节。

`Py_XINCREF()` 和 `Py_XDECREF()` 这两个宏可增加/减少一个对象的引用计数，并且当存在 `NULL` 指针时仍可保证安全 (但请注意在这个上下文中 *temp* 将不为 `NULL`)。 更多相关信息请参考 引用计数 章节。

随后，当要调用此函数时，你将调用 C 函数 `PyObject_CallObject()`。 该函数有两个参数，它们都属于指针，指向任意 Python 对象：即 Python 函数，及其参数列表。 参数列表必须总是一个元组对象，其长度即参数的个数量。 要不带参数地调用 Python 函数，则传入 `NULL` 或一个空元组；要带一个参数调用它，则传入一个单元组。 `Py_BuildValue()` 会在其格式字符串包含一对圆括号内的零个或多个格式代码时返回一个元组。 例如:

```
int arg;
PyObject *arglist;
PyObject *result;
...
arg = 123;
```

```
...
/* 此时将调用回调 */
arglist = Py_BuildValue("(i)", arg);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
```

PyObject_CallObject() 返回Python对象指针，这也是Python函数的返回值。PyObject_CallObject() 是一个对其参数 "引用计数无关" 的函数。例子中新的元组创建用于参数列表，并且在 PyObject_CallObject() 之后立即使用了 Py_DECREF() 。

PyObject_CallObject() 的返回值总是"新"的：要么是一个新建的对象；要么是已有对象，但增加了引用计数。所以除非你想把结果保存在全局变量中，你需要对这个值使用 Py_DECREF()，即使你对里面的内容（特别！）不感兴趣。

但是在你这么做之前，很重要的一点是检查返回值不是 NULL。 如果是的话，Python 函数会终止并引发异常。 如果调用 PyObject_CallObject() 的 C 代码是在 Python 中唤起的，它应当立即返回一个错误来告知其 Python 调用者，以便解释器能打印栈回溯信息，或者让调用方 Python 代码能处理该异常。 如果这无法做到或不合本意，则应当通过调用 PyErr_Clear() 来清除异常。 例如:

```
if (result == NULL)
    return NULL; /* 回传错误 */
...使用 result...
Py_DECREF(result);
```

依赖于具体的回调函数，你还要提供一个参数列表到 PyObject_CallObject() 。在某些情况下参数列表是由Python程序提供的，通过接口再传到回调函数对象。这样就可以不改变形式直接传递。另外一些时候你要构造一个新的元组来传递参数。最简单的方法就是 Py_BuildValue() 函数构造tuple。举个例子，你要传递一个事件代码时可以用如下代码:

```
PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */
/* 可以在此使用 result */
Py_DECREF(result);
```

注意 Py_DECREF(arglist) 所在处会立即调用，在错误检查之前。当然还要注意一些常规的错误，比如 Py_BuildValue() 可能会遭遇内存不足等等。

当你调用函数时还需要注意，用关键字参数调用 PyObject_Call() ，需要支持普通参数和关键字参数。有如如上例子中，我们使用 Py_BuildValue() 来构造字典。

```
PyObject *dict;
...
dict = Py_BuildValue("{s:i}", "name", val);
result = PyObject_Call(my_callback, NULL, dict);
Py_DECREF(dict);
if (result == NULL)
    return NULL; /* 回传错误 */
```

```
/* 可以在此使用 result */
Py_DECREF(result);
```

1.7. 提取扩展函数的参数

函数 PyArg_ParseTuple() 的声明如下：

```
int PyArg_ParseTuple(PyObject *arg, const char *format, ...);
```

参数 *arg* 必须是一个元组对象，包含从 Python 传递给 C 函数的参数列表。*format* 参数必须是一个格式字符串，语法请参考 Python C/API 手册中的 解析参数并构建值变量。剩余参数是各个变量的地址，类型要与格式字符串对应。

注意 PyArg_ParseTuple() 会检测他需要的Python参数类型，却无法检测传递给他的C变量地址，如果这里出错了，可能会在内存中随机写入东西，小心。

注意任何由调用者提供的 Python 对象引用是 *借来的* 引用；不要递减它们的引用计数！

一些调用的例子：

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

```
int ok;
int i, j;
long k, l;
const char *s;
Py_ssize_t size;

ok = PyArg_ParseTuple(args, ""); /* 无参数 */
    /* Python 调用: f() */
```

```
ok = PyArg_ParseTuple(args, "s", &s); /* 一个字符串 */
    /* 可能的 Python 调用: f('whoops!') */
```

```
ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* 两个长整型和一个字符串 */
    /* 可能的 Python 调用: f(1, 2, 'three') */
```

```
ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
    /* 一对整数和一个字符串，其大小也将被返回 */
    /* 可能的 Python 调用: f((1, 2), 'three') */
```

```
{
    const char *file;
    const char *mode = "r";
    int bufsize = 0;
    ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize);
    /* 一个字符串，并可选择传入另一个字符串和一个整数 */
    /* 可能的 Python 调用:
       f('spam')
       f('spam', 'w')
```

```
       f('spam', 'wb', 100000) */
}
```

```
{
    int left, top, right, bottom, h, v;
    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
             &left, &top, &right, &bottom, &h, &v);
    /* 一个矩型和一个点 */
    /* 可能的 Python 调用:
       f(((0, 0), (400, 300)), (10, 10)) */
}
```

```
{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* 一个复数，并提供一个函数名用于错误处理 */
    /* Possible Python call: myfunction(1+2j) */
}
```

1.8. 给扩展函数的关键字参数

函数 PyArg_ParseTupleAndKeywords() 声明如下：

```
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict,
                                const char *format, char * const *kwlist, ...);
```

arg 与 *format* 形参与 PyArg_ParseTuple() 函数所定义的一致。 *kwdict* 形参是作为第三个参数从 Python 运行时接收的关键字字典。 *kwlist* 形参是以 NULL 结尾的字符串列表，它被用来标识形参；名称从左至右与来自 *format* 的类型信息相匹配。 如果执行成功，PyArg_ParseTupleAndKeywords() 会返回真值，否则返回假值并引发一个适当的异常。

备注： 嵌套的元组在使用关键字参数时无法生效，不在 *kwlist* 中的关键字参数会导致 TypeError 异常。

如下例子是使用关键字参数的例子模块，作者是 Geoff Philbrick (philbrick@hks.com):

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

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    const char *state = "a stiff";
    const char *action = "voom";
    const char *type = "Norwegian Blue";

    static char *kwlist[] = {"voltage", "state", "action", "type", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist,
                                     &voltage, &state, &action, &type))
        return NULL;
```

```
    printf("-- This parrot wouldn't %s if you put %i Volts through it.\n",
           action, voltage);
    printf("-- Lovely plumage, the %s -- It's %s!\n", type, state);

    Py_RETURN_NONE;
}

static PyMethodDef keywdarg_methods[] = {
    /* 函数的转换是必要的因为 PyCFunction 值
     * 仅接受两个 PyObject* 形参，而 keywdarg_parrot()
     * 接受三个。
     */
    {"parrot", (PyCFunction)(void(*)(void))keywdarg_parrot, METH_VARARGS | METH_KE
     "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL}   /* sentinel */
};

static struct PyModuleDef keywdarg_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "keywdarg",
    .m_size = 0,
    .m_methods = keywdarg_methods,
};

PyMODINIT_FUNC
PyInit_keywdarg(void)
{
    return PyModuleDef_Init(&keywdarg_module);
}
```

1.9. 构造任意值

这个函数与 PyArg_ParseTuple() 很相似，声明如下：

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

接受一个格式字符串，与 PyArg_ParseTuple() 相同，但是参数必须是原变量的地址指针(输入给函数，而非输出)。最终返回一个Python对象适合于返回C函数调用给Python代码。

一个与 PyArg_ParseTuple() 的不同是，后面可能需要的要求返回一个元组(Python参数里诶包总是在内部描述为元组)，比如用于传递给其他Python函数以参数。 Py_BuildValue() 并不总是生成元组，在多于1个格式字符串时会生成元组，而如果格式字符串为空则返回 None ，一个参数则直接返回该参数的对象。如果要求强制生成一个长度为0的元组，或包含一个元素的元组，需要在格式字符串中加上括号。

例子(左侧是调用，右侧是Python值结果)：

```
Py_BuildValue("")                       None
Py_BuildValue("i", 123)                 123
Py_BuildValue("iii", 123, 456, 789)     (123, 456, 789)
Py_BuildValue("s", "hello")             'hello'
Py_BuildValue("y", "hello")             b'hello'
Py_BuildValue("ss", "hello", "world")   ('hello', 'world')
Py_BuildValue("s#", "hello", 4)         'hell'
Py_BuildValue("y#", "hello", 4)         b'hell'
```

```
Py_BuildValue("()")                      ()
Py_BuildValue("(i)", 123)                (123,)
Py_BuildValue("(ii)", 123, 456)          (123, 456)
Py_BuildValue("(i,i)", 123, 456)         (123, 456)
Py_BuildValue("[i,i]", 123, 456)         [123, 456]
Py_BuildValue("{s:i,s:i}",
              "abc", 123, "def", 456)    {'abc': 123, 'def': 456}
Py_BuildValue("((ii)(ii)) (ii)",
              1, 2, 3, 4, 5, 6)          (((1, 2), (3, 4)), (5, 6))
```

1.10. 引用计数

在C/C++语言中，程序员负责动态分配和回收堆heap当中的内存。在C里，通过函数 `malloc()` 和 `free()` 来完成。在C++里是操作 `new` 和 `delete` 来实现相同的功能。

每个使用 `malloc()` 分配的内存块最终都应当通过恰好一次对 `free()` 的调用返回到可用内存池中。调用 `free()` 的时机非常重要。 如果一个块地址被遗忘而没有为它执行 `free()` 调用，它所占用的内存在程序终结之前将无法被重新使用。 这就称为 *内存泄漏*。 另一方面，如果程序为一个块地址调用了 `free()` 然后却继续使用该内存块，它将与通过另一个 `malloc()` 调用对该内存块的重新使用产生冲突。 这就称为 *使用已释放的内存*。 它所造成的后果与引用未初始化数据一样糟糕 --- 核心转储、错误结果、意外崩溃等等。

内存泄露往往发生在一些并不常见的代码流程上面。比如一个函数申请了内存以后，做了些计算，然后释放内存块。现在一些对函数的修改可能增加对计算的测试并检测错误条件，然后过早的从函数返回了。这很容易忘记在退出前释放内存，特别是后期修改的代码。这种内存泄漏，一旦引入，通常很长时间都难以检测到，错误退出被调用的频度较低，而现代电脑又有非常巨大的虚拟内存，所以泄漏仅在长期运行或频繁调用泄漏函数时才会变得明显。因此，有必要避免内存泄漏，通过代码规范会策略来最小化此类错误。

Python通过 `malloc()` 和 `free()` 包含大量的内存分配和释放，同样需要避免内存泄漏和野指针。他选择的方法就是 *引用计数* 。其原理比较简单：每个对象都包含一个计数器，计数器的增减与对象引用的增减直接相关，当引用计数为0时，表示对象已经没有存在的意义了，对象就可以删除了。

另一个叫法是 *自动垃圾回收* 。(有时引用计数也被看作是垃圾回收策略，于是这里的"自动"用以区分两者)。自动垃圾回收的优点是用户不需要明确的调用 `free()` 。(另一个优点是改善速度或内存使用，然而这并不难)。缺点是对C，没有可移植的自动垃圾回收器，而引用计数则可以可移植的实现(只要 `malloc()` 和 `free()` 函数是可用的，这也是C标准担保的)。也许以后有一天会出现可移植的自动垃圾回收器，但在此前我们必须与引用计数一起工作。

Python使用传统的引用计数实现，也提供了循环监测器，用以检测引用循环。这使得应用无需担心直接或间接的创建了循环引用，这是引用计数垃圾收集的一个弱点。引用循环是对象(可能直接)的引用了本身，所以循环中的每个对象的引用计数都不是0。典型的引用计数实现无法回收处于引用循环中的对象，或者被循环所引用的对象，哪怕没有循环以外的引用了。

循环检测器能够检测垃圾回收循环并能回收它们。 `gc` 模块提供了一种运行该检测器的方式(`collect()` 函数)，以及多个配置接口和在运行时禁用该检测器的功能。

1.10.1. Python中的引用计数

有两个宏 `Py_INCREF(x)` 和 `Py_DECREF(x)`，会处理引用计数的增减。 `Py_DECREF()` 也会在引用计数到达0时释放对象。为了灵活，并不会直接调用 `free()`，而是通过对象的 *类型对象* 的函数指针来调用。为了这个目的(或其他的)，每个对象同时包含一个指向自身类型对象的指针。

最大的问题依旧：何时使用 `Py_INCREF(x)` 和 `Py_DECREF(x)`？我们首先引入一些概念。没有人"拥有"一个对象，你可以 *拥有一个引用* 到一个对象。一个对象的引用计数定义为拥有引用的数量。引用的拥有者有责任调用 `Py_DECREF()`，在引用不再需要时。引用的拥有关系可以被传递。有三种办法来处置拥有的引用：传递、存储、调用 `Py_DECREF()`。忘记处置一个拥有的引用会导致内存泄漏。

还可以 *借用* [2] 一个对象的引用。借用的引用不应该调用 `Py_DECREF()`。借用者必须确保不能持有对象超过拥有者借出的时间。在拥有者处置对象后使用借用的引用是有风险的，应该完全避免 [3]。

借用相对于引用的优点是你无需担心整条路径上代码的引用，或者说，通过借用你无需担心内存泄漏的风险。借用的缺点是一些看起来正确代码上的借用可能会在拥有者处置后使用对象。

借用可以变为拥有引用，通过调用 `Py_INCREF()`。 这不会影响已经借出的拥有者的状态。 这会创建一个新的拥有引用，并给予完全的拥有者责任（新的拥有者必须恰当的处置引用，就像之前的拥有者那样）。

1.10.2. 拥有规则

当一个对象引用传递进出一个函数时，函数的接口应该指定拥有关系的传递是否包含引用。

大多数函数返回一个对象的引用，并传递引用拥有关系。通常，所有创建对象的函数，例如 `PyLong_FromLong()` 和 `Py_BuildValue()`，会传递拥有关系给接收者。即便是对象不是真正新的，你仍然可以获得对象的新引用。一个实例是 `PyLong_FromLong()` 维护了一个流行值的缓存，并可以返回已缓存项目的新引用。

很多另一个对象提取对象的函数，也会传递引用关系，例如 `PyObject_GetAttrString()`。这里的情况不够清晰，一些不太常用的例程是例外的 `PyTuple_GetItem()`， `PyList_GetItem()`， `PyDict_GetItem()`， `PyDict_GetItemString()` 都是返回从元组、列表、字典里借用的引用。

函数 `PyImport_AddModule()` 也会返回借用的引用，哪怕可能会返回创建的对象：这个可能因为一个拥有的引用对象是存储在 `sys.modules` 里。

当你传递一个对象引用到另一个函数时，通常函数是借用出去的。如果需要存储，就使用 `Py_INCREF()` 来变成独立的拥有者。这个规则有两个重要的例外： `PyTuple_SetItem()` 和 `PyList_SetItem()`。这些函数接受传递来的引用关系，哪怕会失败！(注意 `PyDict_SetItem()` 及其同类不会接受引用关系，他们是"正常的")。

当一个C函数被Python调用时，会从调用方传来的参数借用引用。调用者拥有对象的引用，所以借用的引用生命周期可以保证到函数返回。只要当借用的引用需要存储或传递时，就必须转换为拥有的引用，通过调用 `Py_INCREF()`。

Python调用从C函数返回的对象引用时必须是拥有的引用---拥有关系被从函数传递给调用者。

1.10.3. 危险的薄冰

有少数情况下，借用的引用看起来无害，但却可能导致问题。这通常是因为解释器的隐式调用，并可能导致引用拥有者处置这个引用。

首先需要特别注意的情况是使用 Py_DECREF() 到一个无关对象，而这个对象的引用是借用自一个列表的元素。举个实例：

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

这个函数首先借用一个引用 `list[0]`，然后替换 `list[1]` 为值 `0`，最后打印借用的引用。看起来无害是吧，但却不是。

让我们跟随控制流进入 PyList_SetItem()。 该列表拥有对其全部条目的引用，因此当条目 1 被替换时，它必须丢弃原来的条目 1。 现在让我们假定原来的条目 1 是某个用户自定义类的实例，并让我们假定该类定义了 __del__() 方法。 如果该类实例的引用计数为 1，丢弃它将会调用其 __del__() 方法。 在内部，PyList_SetItem() 会调用被替换条目的 Py_DECREF()，这将唤起被替换条目的对应的 tp_dealloc 函数。 在撤销分配期间，tp_dealloc 会调用 tp_finalize，它被映射到用于类实例的 __del__() 方法 (参见 **PEP 442**)。 以上整个过程是在 PyList_SetItem() 调用内部同步发生的。

由于它是用 Python 编写的，因此 __del__() 方法可以执行任意 Python 代码。 它是否可以使 bug() 中对 `item` 的引用失效呢？ 当然可以！ 假定传入 bug() 的列表可以被 __del__() 方法访问，它就可以执行一条语句实现 `del list[0]` 的效果，假定这是对该对象的最后一次引用，它就会释放与之相关联的内存，从而使 `item` 失效。

解决方法是，当你知道了问题的根源，就容易了：临时增加引用计数。正确版本的函数代码如下：

```
void
no_bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

这是一个真实的故事。 一个较旧版本的 Python 曾经包含此问题的变化形式，有人在 C 语言调试器中花费了大量时间，才弄明白为什么他的 __del__() 方法会失败……

有关借入引用的问题的第二种情况是涉及线程的变种。 通常， Python 解释器中的多个线程不会相互影响，因为有一个 全局锁 在保护 Python 的整个对象空间。 不过，有可能使用宏 Py_BEGIN_ALLOW_THREADS 来临时释放这个锁，并使用 Py_END_ALLOW_THREADS 来重新获取它。 这

在阻塞型 I/O 调用操作中很常见，可以让其他线程在等待 I/O 结束期间使用处理器。 显然，下面的函数与之前那个存在相同的问题:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

1.10.4. NULL指针

通常，接受对象引用作为参数的函数不希望你传给它们 `NULL` 指针，并且当你这样做时将会转储核心（或在以后导致核心转储）。 返回对象引用的函数通常只在要指明发生了异常时才返回 `NULL`。 不检测 `NULL` 参数的原因在于这些函数经常要将它们所接收的对象传给其他函数 --- 如果每个函数都检测 `NULL`，将会导致大量的冗余检测而使代码运行得更缓慢。

更好的做法是仅在“源头”上检测 `NULL`，即在接收到一个可能为 `NULL` 的指针，例如来自 `malloc()` 或是一个可能引发异常的函数的时候。

`Py_INCREF()` 和 `Py_DECREF()` 等宏不会检测 `NULL` 指针 --- 但是，它们的变种 `Py_XINCREF()` 和 `Py_XDECREF()` 则会检测。

用于检测特定对象类型的宏 (`Pytype_Check()`) 不会检测 `NULL` 指针 --- 同样地，有大量代码会连续调用这些宏来测试一个对象是否为几种不同预期类型之一，这将会生成冗余的测试。 不存在带有 `NULL` 检测的变体。

C 函数调用机制会保证传给 C 函数的参数列表 (本示例中为 `args`) 绝不会为 `NULL` --- 实际上它会保证其总是为一个元组 [4]。

任何时候将 `NULL` 指针“泄露”给 Python 用户都会是个严重的错误。

1.11. 在C++中编写扩展

还可以在C++中编写扩展模块，只是有些限制。如果主程序(Python解释器)是使用C编译器来编译和链接的，全局或静态对象的构造器就不能使用。而如果是C++编译器来链接的就没有这个问题。函数会被Python解释器调用(通常就是模块初始化函数)必须声明为 `extern "C"`。而是否在 `extern "C" {...}` 里包含Python头文件则不是那么重要，因为如果定义了符号 `__cplusplus` 则已经是这么声明的了(所有现代C++编译器都会定义这个符号)。

1.12. 给扩展模块提供C API

很多扩展模块提供了新的函数和类型供Python使用，但有时扩展模块里的代码也可以被其他扩展模块使用。例如，一个扩展模块可以实现一个类型 "collection" 看起来是没有顺序的。就像是Python列表类型，拥有C API允许扩展模块来创建和维护列表，这个新的集合类型可以有一堆C函数用于给其他扩展模块直接使用。

开始看起来很简单：只需要编写函数(无需声明为 `static`)，提供恰当的头文件，以及C API的文档。实际上在所有扩展模块都是静态链接到Python解释器时也是可以正常工作的。当模块以共享库链接时，一个模块中的符号定义对另一个模块不可见。可见的细节依赖于操作系统，一些系统的Python解释器使用全局命名空间(例如Windows)，有些则在链接时需要一个严格的已导入符号列表(一个例子是AIX)，或者提供可选的不同策略(如Unix系列)。即便是符号是全局可见的，你要调用的模块也可能尚未加载。

可移植性需要不能对符号可见性做任何假设。这意味着扩展模块里的所有符号都应该声明为 `static` ，除了模块的初始化函数，来避免与其他扩展模块的命名冲突(在段落 模块方法表和初始化函数 中讨论) 。这意味着符号应该 *必须* 通过其他导出方式来供其他扩展模块访问。

Python 提供了一个特别的机制用来从一个扩展模块向另一个扩展模块传递 C 层级的信息 (指针): Capsule。 一个 Capsule 就是一个存储了指针 (`void*`) 的 Python 数据类型。 Capsule 只能通过其 C API 来创建和访问，但它们可以像任何其他 Python 对象一样被传递。 特别地，它们可以被赋值给扩展模块命名空间中的一个名称。 其他扩展模块将可以导入这个模块，获取该名称对应的值，然后从 Capsule 中获取指针。

Capsule可以用多种方式导出C API给扩展模块。每个函数可以用自己的Capsule，或者所有C API指针可以存储在一个数组里，数组地址再发布给Capsule。存储和获取指针也可以用多种方式，供客户端模块使用。

无论你选择哪个方法，为你的 Capsule 指定适当的名称都很重要。 函数 `PyCapsule_New()` 接受一个 name 形参 (`const char*`)；允许你传入一个 `NULL` 作为名称，但我们强烈推荐你指定名称。 正确地命名的 Capsule 提供了一定的运行时类型安全性；没有可行的方式能区别两个未命名的 Capsule。

通常来说，Capsule用于暴露C API，其名字应该遵循如下规范：

```
modulename.attributename
```

便利函数 `PyCapsule_Import()` 可以方便的载入通过Capsule提供的C API，仅在Capsule的名字匹配时。这个行为为C API用户提供了高度的确定性来载入正确的C API。

下面的例子演示了一种将大部分负担交给导出模块编写者的处理方式，这对于常用的库模块来说是合适的。 它会将所有 C API 指针（在这个例子里只有一个！）储存到一个 `void` 指针数组，它将成为一个 Capsule 的值。 与模块对应的头文件提供了一个宏用来管理导入模块和获取其 C API 指针；客户端模块只需要在访问 C API 之前调用这个宏即可。

导出模块是对 一个简单的例子 部分的 `spam` 模块的修改。 函数 `spam.system()` 并不直接调用 C 库函数 `system()`，而是调用一个函数 `PySpam_System()`，这个函数在现实中当然会做一些更复杂的事情（比如在每条命令中添加“sapm”） 。 该函数 `PySpam_System()` 也会导出给其他扩展模块。

函数 `PySpam_System()` 是一个纯 C 函数，像其他函数一样声明为 `static`:

```
static int
PySpam_System(const char *command)
{
```

```
    return system(command);
}
```

函数 spam_system() 已按如下方式修改:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = PySpam_System(command);
    return PyLong_FromLong(sts);
}
```

在模块开头，在此行后:

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

添加另外两行:

```
#define SPAM_MODULE
#include "spammodule.h"
```

`#define` 被用来告知头文件它被包括在导出的模块中，而不是客户端模块。 最终，模块的 mod_exec 函数必须负责初始化 C API 指针数组:

```
static int
spam_module_exec(PyObject *m)
{
static void *PySpam_API[PySpam_API_pointers];
PyObject *c_api_object;

/* 初始化 C API 指针数组 */
PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

/* 创建包含 API 指针数组地址的 Capsule */
c_api_object = PyCapsule_New((void *)PySpam_API, "spam._C_API", NULL);

if (PyModule_Add(m, "_C_API", c_api_object) < 0) {
return -1;
}

return 0;
}
```

请注意 `PySpam_API` 被声明为 `static`；否则指针数组会在 `PyInit_spam()` 终结时消失!

头文件 `spammodule.h` 里的一堆工作，看起来如下所示:

```
#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
```

```
extern "C" {
#endif

/* 用于 spammodule 的头文件 */

/* C API 函数 */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO (const char *command)

/* C API 指针的总数 */
#define PySpam_API_pointers 1


#ifdef SPAM_MODULE
/* 该节将在编译 spammodule.c 时使用 */

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

#else
/* 该节将在使用 spammodule 的 API 的模块中使用 */

static void **PySpam_API;

#define PySpam_System \
 (*(PySpam_System_RETURN (*)PySpam_System_PROTO) PySpam_API[PySpam_System_NUM])

/* 出错时返回 -1，成功时返回 0。
 * 如果有异常 PyCapsule_Import 将设置一个异常。
 */
static int
import_spam(void)
{
    PySpam_API = (void **)PyCapsule_Import("spam._C_API", 0);
    return (PySpam_API != NULL) ? 0 : -1;
}

#endif

#ifdef __cplusplus
}
#endif

#endif /* !defined(Py_SPAMMODULE_H) */
```

客户端模块要访问函数 `PySpam_System()` 所必须做的全部事情就是在其 `mod_exec` 函数中调用函数 `import_spam()`（更准确地说是宏）：

```
static int
client_module_exec(PyObject *m)
{
if (import_spam() < 0) {
return -1;
}
/* 额外的初始化可在此进行 */
return 0;
}
```

这种方法的主要缺点是，文件 `spammodule.h` 过于复杂。当然，对每个要导出的函数，基本结构是相似的，所以只需要学习一次。

最后需要提醒的是Capsule提供了额外的功能，用于存储在Capsule里的指针的内存分配和释放。细节参考 Python/C API参考手册的章节 Capsule 对象 和Capsule的实现(在Python源码发行包的 `Include/pycapsule.h` 和 `Objects/pycapsule.c`)。

备注

[1] 这个函数的接口已经在标准模块 os 里了，这里作为一个简单而直接的例子。

[2] 术语"借用"一个引用是不完全正确的：拥有者仍然有引用的拷贝。

[3] 检查引用计数至少为1 **没有用** ，引用计数本身可以在已经释放的内存里，并有可能被其他对象所用。

[4] 当你使用 "旧式" 风格调用约定时，这些保证不成立，尽管这依旧存在于很多旧代码中。