### 扩展和嵌入 Python 解释器（基于Python3.10.0）

本文档描述了如何使用 C 或 C++ 编写模块以使用新模块来扩展 Python 解释器的功能。 这些模块不仅可以定义新的函数，还可以定义新的对象类型及其方法。 该文档还描述了如何将 Python 解释器嵌入到另一个应用程序中，以用作扩展语言。 最后，它展示了如何编译和链接扩展模块，以便它们可以动态地（在运行时）加载到解释器中，如果底层操作系统支持此特性的话。

本文档假设你具备有关 Python 的基本知识。有关该语言的非正式介绍，请参阅 Python 教程 。 Python 语言参考手册 给出了更正式的语言定义。 Python 标准库 包含现有的对象类型、函数和模块（内置和用 Python 编写）的文档，使语言具有广泛的应用范围。

关于整个 Python/C API 的详细介绍，请参阅独立的 Python/C API 参考手册 。

#### 推荐的第三方工具

本指南仅介绍了作为此 CPython 版本的一部分提供的创建扩展的基本工具。 第三方工具，如 Cython 、 cffi 、 SWIG 和 Numba 提供了更简单和更复杂的方法来为 Python 创建 C 和 C ++ 扩展。

#### 不使用第三方工具创建扩展

本指南的这一部分包括在没有第三方工具帮助的情况下创建 C 和 C ++ 扩展。它主要用于这些工具的创建者，而不是建议你创建自己的 C 扩展的方法。

##### 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。

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，而且可移植性更好。不知为何编译失败了。

#### 一个简单的例子（Python中调用C函数）

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)。

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 提取扩展函数的参数 来了解这个宏的更多内容。

所有在 Python.h 中定义的用户可见的符号都具有 Py 或 PY 前缀，已在标准头文件中定义的那些除外。 考虑到便利性，也由于其在 Python 解释器中被广泛使用，"Python.h" 还包含了一些标准头文件: <stdio.h>，<string.h>，<errno.h> 和 <stdlib.h>。 如果后面的头文件在你的系统上不存在，它还会直接声明函数 malloc()，free() 和 realloc()。

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)。

##### 关于错误和异常

在 Python 解释器中有一个重要的惯例：当一个函数出错时，它应当设置异常条件并返回错误值（通常为 NULL 指针）。 异常存储于解释器内部的静态全局变量中；如此变量为 NULL 表示未发生异常。 还有第二个全局变量用于保存异常的“关联值”（即 raise 的第二个参数）。 第三个变量包含 Python 代码产生错误情况下的栈回溯信息。 这三个变量是 Python 中 sys.exc\_info() 的结果在 C 中的对应物（请参阅 Python 库参考的 sys 模块部分）。 了解它们对于理解错误的传递方式是非常重要的。

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 不应该 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 不应该 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程序。

（在某些情况下，当模块确实能够通过调用其它 PyErr\_\* 函数给出更加详细的错误消息，并且在这些情况是可以这样做的。 但是按照一般规则，这是不必要的，并可能导致有关错误原因的信息丢失：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。 所有内置的 Python 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 来表示一个文件无法被打开 (那大概应该用 PyExc\_IOError)。 如果参数列表有问题，PyArg\_ParseTuple() 函数通常会引发 PyExc\_TypeError。 如果你想要一个参数的值必须处于特定范围之内或必须满足其他条件，则适宜使用 PyExc\_ValueError。

你也可以为你的模块定义一个唯一的新异常。需要在文件前部声明一个静态对象变量，如:

static PyObject \*SpamError;

并且在你的模块的初始化函数 (PyInit\_spam()) 中使用一个异常对象来初始化:

PyMODINIT\_FUNC

PyInit\_spam(void)

{

PyObject \*m;

m = PyModule\_Create(&spammodule);

if (m == NULL)

return NULL;

SpamError = PyErr\_NewException("spam.error", NULL, NULL);

Py\_XINCREF(SpamError);

if (PyModule\_AddObject(m, "error", SpamError) < 0) {

Py\_XDECREF(SpamError);

Py\_CLEAR(SpamError);

Py\_DECREF(m);

return NULL;

}

return m;

}

注意异常对象的Python名字是 spam.error 。而 PyErr\_NewException() 函数可以创建一个类，其基类为 Exception (除非是另一个类传入以替换 NULL )， 细节参见 [内置异常](https://www.bookstack.cn/read/python-3.10.0-zh/177bd954d1cc6696.md#bltin-exceptions) 。

同样注意的是创建类保存了 SpamError 的一个引用，这是有意的。为了防止被垃圾回收掉，否则 SpamError 随时会成为野指针。

一会讨论 PyMODINIT\_FUNC 作为函数返回类型的用法。

spam.error 异常可以在扩展模块中抛出，通过 PyErr\_SetString() 函数调用，如下：

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);

}

##### 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所使用的错误提示)，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() 函数必须返回 sts 的值作为Python对象。这通过使用函数 PyLong\_FromLong() 来实现。

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)。

如果你的 C 函数没有有用的返回值 (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（可以通过 Py\_RETURN\_NONE 宏来实现）:

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”。

##### 模块方法表和初始化函数

为了展示 spam\_system() 如何被Python程序调用。把函数声明为可以被Python调用，需要先定义一个方法表 “method table” 。

static PyMethodDef SpamMethods[] = {

...

{"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。

METH\_KEYWORDS 值表示接受关键字参数。这种情况下C函数需要接受第三个 PyObject \* 对象，表示字典参数，使用 PyArg\_ParseTupleAndKeywords() 来解析出参数。

这个方法表必须被模块定义结构所引用。

static struct PyModuleDef spammodule = {

PyModuleDef\_HEAD\_INIT,

"spam", /\* name of module \*/

spam\_doc, /\* module documentation, may be NULL \*/

-1, /\* size of per-interpreter state of the module,

or -1 if the module keeps state in global variables. \*/

SpamMethods

};

这个结构体必须传递给解释器的模块初始化函数。初始化函数必须命名为 PyInit\_name() ，其中 name 是模块的名字，并应该定义为非 static ，且在模块文件里：

PyMODINIT\_FUNC

PyInit\_spam(void)

{

return PyModule\_Create(&spammodule);

}

注意 PyMODINIT\_FUNC 将函数声明为 PyObject \* 返回类型，声明了任何平台所要求的特殊链接声明，并针对 C++ 将函数声明为 extern "C"。

当 Python 程序首次导入 spam 模块时， PyInit\_spam() 会被调用。 （有关嵌入 Python 的注释参见下文。） 它将调用 PyModule\_Create()，该函数会返回一个模块对象，并基于在模块定义中找到的表将内置函数对象插入到新创建的模块中（该表是一个 PyMethodDef 结构体的数组）。 PyModule\_Create() 返回一个指向它所创建的模块对象的指针。 它可能会因程度严重的特定错误而中止，或者在模块无法成功初始化时返回 NULL。 初始化函数必须返回模块对象给其调用者，这样它就可以被插入到 sys.modules 中。

当嵌入Python时， PyInit\_spam() 函数不会被自动调用，除非放在 PyImport\_Inittab 表里。要添加模块到初始化表，使用 PyImport\_AppendInittab() ，可选的跟着一个模块的导入。

int

main(int argc, char \*argv[])

{

wchar\_t \*program = Py\_DecodeLocale(argv[0], NULL);

if (program == NULL) {

fprintf(stderr, "Fatal error: cannot decode argv[0]\n");

exit(1);

}

/\* Add a built-in module, before Py\_Initialize \*/

if (PyImport\_AppendInittab("spam", PyInit\_spam) == -1) {

fprintf(stderr, "Error: could not extend in-built modules table\n");

exit(1);

}

/\* Pass argv[0] to the Python interpreter \*/

Py\_SetProgramName(program);

/\* Initialize the Python interpreter. Required.

If this step fails, it will be a fatal error. \*/

Py\_Initialize();

/\* Optionally import the module; alternatively,

import can be deferred until the embedded script

imports it. \*/

PyObject \*pmodule = PyImport\_ImportModule("spam");

if (!pmodule) {

PyErr\_Print();

fprintf(stderr, "Error: could not import module 'spam'\n");

}

...

PyMem\_RawFree(program);

return 0;

}

要从 sys.modules 删除实体或导入已编译模块到一个进程里的多个解释器(或使用 fork() 而没用 exec() )会在一些扩展模块上产生错误。扩展模块作者可以在初始化内部数据结构时给出警告。

更多关于模块的现实的例子包含在Python源码包的 Modules/xxmodule.c 中。这些文件可以用作你的代码模板，或者学习。脚本 modulator.py 包含在源码发行版或Windows安装中，提供了一个简单的GUI，用来声明需要实现的函数和对象，并且可以生成供填入的模板。脚本在 Tools/modulator/ 目录。查看README以了解用法。

不像我们的 spam 例子， xxmodule 使用了 多阶段初始化 (Python3.5开始引入)， PyInit\_spam 会返回一个 PyModuleDef 结构体，然后创建的模块放到导入机制。细节参考 PEP 489 的多阶段初始化。

##### 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](https://www.bookstack.cn/read/python-3.10.0-zh/40b98f6c5499dcd4.md#building) 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

##### 完整代码（Python3.7@cygwin）

spammodule.c

#define PY\_SSIZE\_T\_CLEAN

#include <python3.7m/Python.h>

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);

}

static PyMethodDef SpamMethods[] = {

{"system", spam\_system, METH\_VARARGS,

"Execute a shell command."},

{NULL, NULL, 0, NULL} /\* Sentinel \*/

};

static struct PyModuleDef spammodule = {

PyModuleDef\_HEAD\_INIT,

"spam", /\* name of module \*/

//spam\_doc, /\* module documentation, may be NULL \*/

NULL,

-1, /\* size of per-interpreter state of the module,

or -1 if the module keeps state in global variables. \*/

SpamMethods

};

PyMODINIT\_FUNC

PyInit\_spam(void)

{

return PyModule\_Create(&spammodule);

}

int

main(int argc, char \*argv[])

{

wchar\_t \*program = Py\_DecodeLocale(argv[0], NULL);

if (program == NULL) {

fprintf(stderr, "Fatal error: cannot decode argv[0]\n");

exit(1);

}

/\* Add a built-in module, before Py\_Initialize \*/

if (PyImport\_AppendInittab("spam", PyInit\_spam) == -1) {

fprintf(stderr, "Error: could not extend in-built modules table\n");

exit(1);

}

/\* Pass argv[0] to the Python interpreter \*/

Py\_SetProgramName(program);

/\* Initialize the Python interpreter. Required.

If this step fails, it will be a fatal error. \*/

Py\_Initialize();

/\* Optionally import the module; alternatively,

import can be deferred until the embedded script

imports it. \*/

PyObject \*pmodule = PyImport\_ImportModule("spam");

if (!pmodule) {

PyErr\_Print();

fprintf(stderr, "Error: could not import module 'spam'\n");

}

PyMem\_RawFree(program);

return 0;

}

setup.py

from distutils.core import setup, Extension

module1 = Extension('spam',

sources = ['spammodule.c'])

setup (name = 'PackageName',

version = '1.0',

description = 'This is a demo package',

ext\_modules = [module1])

编译命令

python3 setup.py build