

属于混合编程的问题。较全面的介绍一下，不仅限于题主提出的问题。

以下讨论中，Python指它的标准实现，即CPython（虽然不是很严格）

本文分4个部分

1. C/C++ 调用 Python（基础篇）— 仅讨论Python官方提供的实现方式
2. Python 调用 C/C++（基础篇）— 仅讨论Python官方提供的实现方式
3. C/C++ 调用 Python（高级篇）— 使用 Cython
4. Python 调用 C/C++（高级篇）— 使用 SWIG

练习本文中的例子，需要搭建Python扩展开发环境。具体细节见[搭建Python扩展开发环境 - 蛇之魅惑 - 知乎专栏](#)

1 C/C++ 调用 Python（基础篇）

Python 本身就是一个C库。你所看到的可执行体python只不过是个stub。真正的python实体在动态链接库里实现，在Windows平台上，这个文件位于 %SystemRoot%\System32\python27.dll。

你也可以在自己的程序中调用Python，看起来非常容易：

```
//my_python.c

#include <Python.h>

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

{

Py_SetProgramName(argv[0]);

Py_Initialize();

PyRun_SimpleString("print 'Hello Python!' ");

Py_Finalize();
```

```
return 0;
```

```
}
```

在Windows平台下，打开Visual Studio命令提示符，编译命令为

```
cl my_python.c -IC:Python27include C:Python27libspython27.lib
```

在Linux下编译命令为

```
gcc my_python.c -o my_python -I/usr/include/python2.7/ -lpython2.7
```

在Mac OS X 下的编译命令同上

产生可执行文件后，直接运行，结果为输出

```
Hello Python!
```

Python库函数PyRun_SimpleString可以执行字符串形式的Python代码。

虽然非常简单，但这段代码除了能用C语言动态生成一些Python代码之外，并没有什么用处。我们需要的是C语言的数据结构能够和Python交互。

下面举个例子，比如说，有一天我们用Python写了一个功能特别强大的函数：

```
def great_function(a):
```

```
    return a + 1
```

接下来要把它包装成C语言的函数。我们期待的C语言的对应函数应该是这样的：

```
int great_function_from_python(int a) {
```

```
    int res;
```

```
    // some magic
```

```
    return res;
```

```
}
```

首先，复用Python模块得做 ‘import’ ，这里也不例外。所以我们将great_function放到一个module里，比如说，这个module名字叫 great_module.py

接下来就要用C来调用Python了，完整的代码如下：

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

```
int great_function_from_python(int a) {
```

```
int res;
```

```
PyObject *pModule,*pFunc;
```

```
PyObject *pArgs, *pValue;
```

```
/* import */
```

```
pModule = PyImport_Import(PyString_FromString("great_module"));
```

```
/* great_module.great_function */
```

```
pFunc = PyObject_GetAttrString(pModule, "great_function");
```

```
/* build args */
```

```
pArgs = PyTuple_New(1);
```

```
PyTuple_SetItem(pArgs,0, PyInt_FromLong(a));
```

```
/* call */
```

```
pValue = PyObject_CallObject(pFunc, pArgs);
```

```
res = PyInt_AsLong(pValue);
```

```
return res;
```

```
}
```

从上述代码可以窥见Python内部运行的方式：

- 所有Python元素，module、function、tuple、string等等，实际上都是PyObject。C语言里操纵它们，一律使用PyObject*。
- Python的类型与C语言类型可以相互转换。Python类型XXX转换为C语言类型YYY要使用PyXXX_AsYYY函数；C类型YYY转换为Python类型XXX要使用PyXXX_FromYYY函数。
- 也可以创建Python类型的变量，使用PyXXX_New可以创建类型为XXX的变量。
- 若a是Tuple，则a[i] = b对应于 PyTuple_SetItem(a,i,b)，有理由相信还有一个函数PyTuple_GetItem完成取得某一项的值。
- 不仅Python语言很优雅，Python的库函数API也非常优雅。

现在我们得到了一个C语言的函数了，可以写一个main测试它

```
#include <Python.h>

int great_function_from_python(int a);

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

    Py_Initialize();

    printf("%d",great_function_from_python(2));

    Py_Finalize();

}
```

编译的方式就用本节开头使用的方法。

在Linux/Mac OSX运行此示例之前，可能先需要设置环境变量：

```
bash:
```

```
export PYTHONPATH=./$PYTHONPATH
```

csh:

```
setenv PYTHONPATH .:$PYTHONPATH
```

2 Python 调用 C/C++ (基础篇)

这种做法称为Python扩展。

比如说，我们有一个功能强大的C函数：

```
int great_function(int a) {  
  
return a + 1;  
  
}
```

期望在Python里这样使用：

```
>>> from great_module import great_function
```

```
>>> great_function(2)
```

3

考虑最简单的情况。我们把功能强大的函数放入C文件 great_module.c 中。

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

```
int great_function(int a) {  
  
return a + 1;  
  
}
```

```
static PyObject * _great_function(PyObject *self, PyObject *args)
```

```
{
```

```
int _a;
```

```
int res;

if (!PyArg_ParseTuple(args, "i", &_a))

return NULL;

res = great_function(_a);

return PyLong_FromLong(res);

}

static PyMethodDef GreateModuleMethods[] = {

{

"great_function",

_great_function,

METH_VARARGS,

""

},

{NULL, NULL, 0, NULL}

};

PyMODINIT_FUNC initsgreat_module(void) {

(void) Py_InitModule("great_module", GreateModuleMethods);

}
```

除了功能强大的函数great_function外，这个文件中还有以下部分：

- 包裹函数_great_function。它负责将Python的参数转化为C的参数（PyAr

g_ParseTuple) , 调用实际的great_function , 并处理great_function的返回值 , 最终返回给Python环境。

- 导出表GreateModuleMethods。它负责告诉Python这个模块里有哪些函数可以被Python调用。导出表的名字可以随便起, 每一项有4个参数: 第一个参数是提供给Python环境的函数名称, 第二个参数是_great_function , 即包裹函数。第三个参数的含义是参数变长, 第四个参数是一个说明性的字符串。导出表总是以{NULL, NULL, 0, NULL}结束。
- 导出函数initgreat_module。这个名字不是任取的, 是你的module名称添加前缀init。导出函数中将模块名称与导出表进行连接。

在Windows下面, 在Visual Studio命令提示符下编译这个文件的命令是

```
cl /LD great_module.c /o great_module.pyd -IC:Python27include  
C:Python27libspython27.lib
```

/LD 即生成动态链接库。编译成功后在当前目录可以得到 great_module.pyd (实际上是dll)。这个pyd可以在Python环境下直接当作module使用。

在Linux下面, 则用gcc编译:

```
gcc -fPIC -shared great_module.c -o great_module.so  
-I/usr/include/python2.7/ -lpython2.7
```

在当前目录下得到great_module.so, 同理可以在Python中直接使用。

本部分参考资料

- 《Python源码剖析-深度探索动态语言核心技术》是系统介绍CPython实现以及运行原理的优秀教程。
- Python
官方文档的这一章详细介绍了C/C++与Python的双向互动Extending and Embedding the Python Interpreter
- 关于编译环境, 本文所述方法仅为出示原理所用。规范的方式如下: 3. Building C and C++ Extensions with distutils
- 作为字典使用的官方参考文档 Python/C API Reference Manual

用以上的方法实现C/C++与Python的混合编程, 需要对Python的内部实现有相当的了解。接下来介绍当前较为成熟的技术Cython和SWIG。

3 C/C++ 调用 Python (使用Cython)

在前面的小节中谈到，Python的数据类型和C的数据类型貌似是有某种“一一对应”的关系的，此外，由于Python（确切的说是CPython）本身是由C语言实现的，故Python数据类型之间的函数运算也必然与C语言有对应关系。那么，有没有可能“自动”的做替换，把Python代码直接变成C代码呢？答案是肯定的，这就是Cython主要解决的问题。

安装Cython非常简单。Python 2.7.9以上的版本已经自带easy_install：

```
easy_install -U cython
```

在Windows环境下依然需要Visual Studio，由于安装的过程需要编译Cython的源代码，故上述命令需要在Visual Studio命令提示符下完成。一会儿使用Cython的时候，也需要在Visual Studio命令提示符下进行操作，这一点和第一部分的要求是一样的。

继续以例子说明：

```
#great_module.pyx
```

```
cdef public great_function(a,index):
```

```
return a[index]
```

这其中有非Python关键字cdef和public。这些关键字属于Cython。由于我们需要在C语言中使用“编译好的Python代码”，所以得让great_function从外面变得可见，方法就是以“public”修饰。而cdef类似于Python的def，只有使用cdef才可以使用Cython的关键字public。

这个函数中其他的部分与正常的Python代码是一样的。

接下来编译 great_module.pyx

```
cython great_module.pyx
```

得到great_module.h和great_module.c。打开great_module.h可以找到这样一句声明：

```
__PYX_EXTERN_C DL_IMPORT(PyObject) *great_function(PyObject *,  
PyObject *)
```

写一个main使用great_function。注意great_function并不规定a是何种类型，它的功能只是提取a的第index的成员而已，故使用great_function的时候，a可以传入Python String，也可以传入tuple之类的其他可迭代类型。仍然使用之前提到的类型转换函数PyXXX_FromYYY和PyXXX_AsYYY。

```
//main.c  
  
#include <Python.h>  
  
#include "great_module.h"  
  
int main(int argc, char *argv[]) {  
  
PyObject *tuple;  
  
Py_Initialize();  
  
initgreat_module();  
  
printf("%s ",PyString_AsString(  
great_function(  
PyString_FromString("hello"),  
PyInt_FromLong(1)  
)  
));  
  
tuple = Py_BuildValue("(iis)", 1, 2, "three");  
  
printf("%d ",PyInt_AsLong(  
great_function(  

```

```
tuple,  
PyInt_FromLong(1)  
)  
));  
printf("%s ",PyString_AsString(  
great_function(  
tuple,  
PyInt_FromLong(2)  
)  
));  
Py_Finalize();  
}
```

编译命令和第一部分相同：

在Windows下编译命令为

```
cl main.c great_module.c -IC:Python27include C:Python27libspython27.lib
```

在Linux下编译命令为

```
gcc main.c great_module.c -o main -I/usr/include/python2.7/ -lpython2.7
```

这个例子中我们使用了Python的动态类型特性。如果你想指定类型，可以利用Cython的静态类型关键字。例子如下：

```
#great_module.pyx
```

```
cdef public char great_function(const char * a,int index):
```

```
return a[index]
```

cython编译后得到的.h里，great_function的声明是这样的：

```
__PYX_EXTERN_C DL_IMPORT(char) great_function(char const *, int);
```

很开心对不对！

这样的话，我们的main函数已经几乎看不到Python的痕迹了：

```
//main.c
```

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

```
#include "great_module.h"
```

```
int main(int argc, char *argv[]) {
```

```
Py_Initialize();
```

```
initgreat_module();
```

```
printf("%c",great_function("Hello",2));
```

```
Py_Finalize();
```

```
}
```

在这一部分的最后我们给一个看似实用的应用（仅限于Windows）：

还是利用刚才的great_module.pyx，准备一个dllmain.c：

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

```
#include <Windows.h>
```

```
#include "great_module.h"
```

```
extern __declspec(dllexport) int __stdcall _great_function(const char * a, int
b) {

return great_function(a,b);

}
```

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL,DWORD fdwReason,LPVOID
lpReserved) {

switch( fdwReason ) {

case DLL_PROCESS_ATTACH:

Py_Initialize();

initgreat_module();

break;

case DLL_PROCESS_DETACH:

Py_Finalize();

break;

}

return TRUE;

}
```

在Visual Studio命令提示符下编译：

```
cl /LD dllmain.c great_module.c -IC:Python27include
C:Python27libspython27.lib
```

会得到一个dll

main.dll。我们在Excel里面使用它，没错，传说中的Excel与Python混合编程：

```
<noscript>  </noscript
>
```

参考资料：Cython的官方文档，质量非常高：

Welcome to Cython' s Documentation

4 Python调用C/C++（使用SWIG）

用C/C++对脚本语言的功能扩展是非常常见的事情，Python也不例外。除了SWIG，市面上还有若干用于Python扩展的工具包，比较知名的还有Boost.Python、SIP等，此外，Cython由于可以直接集成C/C++代码，并方便的生成Python模块，故也可以完成扩展Python的任务。

答主在这里选用SWIG的一个重要原因是，它不仅可以用于Python，也可以用于其他语言。如今SWIG已经支持C/C++的好基友Java，主流脚本语言Python、Perl、Ruby、PHP、JavaScript、tcl、Lua，还有Go、C#，以及R。SWIG是基于配置的，也就是说，原则上一套配置改变不同的编译方法就能适用各种语言（当然，这是理想情况了.....）

SWIG的安装方便，有Windows的预编译包，解压即用，绿色健康。主流Linux通常集成swig的包，也可以下载源代码自己编译，SWIG非常小巧，通常安装不会出什么问题。

用SWIG扩展Python，你需要有一个待扩展的C/C++库。这个库有可能是你自己写的，也有可能是某个项目提供的。这里举一个不浮夸的例子：希望在Python中用到SSE4指令集的CRC32指令。

首先打开指令集的文档：<https://software.intel.com/en-us/node/514245>

可以看到有6个函数。分析6个函数的原型，其参数和返回值都是简单的整数。于是书写SWIG的配置文件（为了简化起见，未包含2个64位函数）：

```
/* File: mymodule.i */
```

```
%module mymodule
```

```
%{
```

```
#include "nmmintrin.h"
```

```
%}
```

```
int _mm_popcnt_u32(unsigned int v);
```

```
unsigned int _mm_crc32_u8 (unsigned int crc, unsigned char v);
```

```
unsigned int _mm_crc32_u16(unsigned int crc, unsigned short v);
```

```
unsigned int _mm_crc32_u32(unsigned int crc, unsigned int v);
```

接下来使用SWIG将这个配置文件编译为所谓Python Module Wrapper

```
swig -python mymodule.i
```

得到一个 mymodule_wrap.c和一个mymodule.py。把它编译为Python扩展：

Windows :

```
cl /LD mymodule_wrap.c /o _mymodule.pyd -IC:Python27include  
C:Python27libspython27.lib
```

Linux :

```
gcc -fPIC -shared mymodule_wrap.c -o _mymodule.so  
-I/usr/include/python2.7/ -lpython2.7
```

注意输出文件名前面要加一个下划线。

现在可以立即在Python下使用这个module了：

```
>>> import mymodule
```

```
>>> mymodule._mm_popcnt_u32(10)
```

2

回顾这个配置文件分为3个部分：

1. 定义module名称mymodule，通常，module名称要和文件名保持一致。
2. %{ %} 包裹的部分是C语言的代码，这段代码会原封不动的复制到mymodule_wrap.c
3. 欲导出的函数签名列表。直接从头文件里复制过来即可。

还记得本文第2节的那个great_function吗？有了SWIG，事情就会变得如此简单：

```
/* great_module.i */  
  
%module great_module  
  
%{  
  
int great_function(int a) {  
  
return a + 1;  
  
}  
  
%}  
  
int great_function(int a);
```

换句话说，SWIG自动完成了诸如Python类型转换、module初始化、导出代码表生成的诸多工作。

对于C++，SWIG也可以应对。例如以下代码有C++类的定义：

```
//great_class.h  
  
#ifndef GREAT_CLASS  
  
#define GREAT_CLASS  
  
class Great {
```

```
private:
```

```
int s;
```

```
public:
```

```
void setWall (int _s) {s = _s};
```

```
int getWall () {return s};
```

```
};
```

```
#endif // GREAT_CLASS
```

对应的SWIG配置文件

```
/* great_class.i */
```

```
%module great_class
```

```
{
```

```
#include "great_class.h"
```

```
}
```

```
%include "great_class.h"
```

这里不再重新敲一遍class的定义了，直接使用SWIG的%include指令

SWIG编译时要加-c++这个选项，生成的扩展名为cxx

```
swig -c++ -python great_class.i
```

Windows下编译：

```
cl /LD great_class_wrap.cxx /o _great_class.pyd -IC:Python27include  
C:Python27libspython27.lib
```

Linux , 使用C++的编译器

```
g++ -fPIC -shared great_class_wrap.cxx -o _great_class.so  
-I/usr/include/python2.7/ -lpython2.7
```

在Python交互模式下测试：

```
>>> import great_class  
  
>>> c = great_class.Great()  
  
>>> c.setWall(5)  
  
>>> c.getWall()
```

5

也就是说C++的class会直接映射到Python class

SWIG非常强大，对于Python接口而言，简单类型，甚至指针，都无需人工干涉即可自动转换，而复杂类型，尤其是自定义类型，SWIG提供了typemap供转换。而一旦使用了typemap，配置文件将不再在各个语言当中通用。

参考资料：

SWIG的官方文档，质量比较高。SWIG Users Manual

有个对应的中文版官网，很多年没有更新了。

写在最后：

由于CPython自身的结构设计合理，使得Python的C/C++扩展非常容易。如果打算快速完成任务，Cython（C/C++调用Python）和SWIG（Python调用C/C++）是很不错的选择。但是，一旦涉及到比较复杂的转换任务，无论是继续使用Cython还是SWIG，仍然需要学习Python源代码。

本文使用的开发环境：

Python 2.7.10

Cython 0.22

SWIG 3.0.6

Windows 10 x64 RTM

CentOS 7.1 AMD 64

Mac OSX 10.10.4

文中所述原理与具体环境适用性强。

文章所述代码均用于演示，缺乏必备的异常检查

最后，想学习JAVA的小伙伴们！

请关注+私信回复：“学习”就可以拿到一份我为大家准备的学习资料！

java学习资料