

编程语言Python编程C / C++

关注者5,135被浏览460,032

如何实现 C/C++ 与 Python 的通信？

想在 C++ 中用 Python 进行数值计算，Python 需要访问 C++ 的变量并计算后返回数值。有什么好办法呢？

关注问题

写回答

邀请回答

1 条评论

分享

41 个回答默认排序



Jerry Jho

Brony

编辑推荐

3,502 人赞同了该回答

以下所有文字均为答主手敲，转载请注明出处和作者 #####
更新：关于ctypes，见拙作 [聊聊Python ctypes 模块 - 蛇之魅惑 - 知乎专栏](#)

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!'\n");
    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:\Python27\include C:\Python27\libs\python27.lib
```



下载知乎客户端

与世界分享知识、经验和见解

相关问题

C、C++、MATLAB、Python、Go 哪个比较适合写算法？

27 个回答

为什么c++开发不了网站 而Python可以 编程语言缺少什么模块吗？

9 个回答

为什么有些人可以不学习c++或者python直接学算法？

10 个回答

Python算水仙花数的程序怎么改进？

12 个回答

一个算法可以用任意的高级编程语言表达吗？

8 个回答

相关推荐



2019 年的魅力声音必修课

张皓翔

共 15 节课

试听



自学计算机编程：进阶专业

★★★★★ 78 人参与



管理视野 17

复旦大学管理学院

164 人读过

阅读

刘看山 · 知乎指南 · 知乎协议 · 知乎隐私保护指引

应用 · 工作 · 申请开通知乎机构号

侵权举报 · 网上有害信息举报专区

违法和不良信息举报：010-82716601

儿童色情信息举报专区

电信与服务业务经营许可证

网络文化经营许可证

联系我们 © 2019 知乎

https://www.zhihu.com/question/23003213

1/14

```
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



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;
```



```
res = great_function(a);
return PyLong_FromLong(res);
}

static PyMethodDef GreateModuleMethods[] = {
    {
        "great_function",
        _great_function,
        METH_VARARGS,
        ""
    },
    {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC initlet_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（PyArg_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initlet_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:\Python27\include C:\Python27\libs\pythor
```

/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
```

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



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\n",PyString_AsString(
        great_function(
            PyString_FromString("hello"),
            PyInt_FromLong(1)
        )
    ));
    tuple = Py_BuildValue("(iis)", 1, 2, "three");
    printf("%d\n",PyInt_AsLong(
        great_function(
            tuple,
            PyInt_FromLong(1)
        )
    ));
}
```

```

        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:\Python27\include C:\Python27\libs\python27.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;
    }
}

```



在Visual Studio命令提示符下编译：

```
cl /LD dllmain.c great_module.c -IC:\Python27\include C:\Python27\libs\python27.lib
```

会得到一个dllmain.dll。我们在Excel里面使用它，没错，传说中的**Excel与Python混合编程**：

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。

首先打开指令集文档：software.intel.com/en-u...

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

%{
```



```
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:\Python27\include C:\Python27\libs\python27
```

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:




```
#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:\Python27\include C:\Python27\libs\
```

Linux，使用C++的编译器

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

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



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

编辑于 2015-08-10

▲ 赞同 3.5K ▼

128 条评论

🔗 分享

★ 收藏

♥ 感谢

收起 ^



季文瀚

Alpha21016

72 人赞同了该回答

-----2016.6原文-----

给最高票的答案做点补充。

Python3的C拓展用法和Python2相比有一些变动。

如果使用Py3环境的话, 最高票答案里第二部分的代码应该改为, 新建great_module.c

```
#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}
};

static struct PyModuleDef great_module = {
    PyModuleDef_HEAD_INIT,
    "great_module",
    NULL,
    -1,
    GreateModuleMethods
};

PyMODINIT_FUNC PyInit_great_module(void)
{
    PyObject *m;
    m = PyModule_Create(&great_module);
    if (m == NULL)
        return NULL;
    printf("init great_module module\n");
    return m;
}
```

这里主要改了后面的init module的部分, Py3里模块初始化以及参数转化的方式都有改变
相关的内容中文的教程和文档都很稀少, 大部分只能靠自己看官方文档摸索或者刷stackoverflow

另外编译的话可以直接用setuptools来编译更方便, 新建setup.py



```
great_module = Extension('great_module', sources=["great_module.c"])
setup(ext_modules=[great_module])
```

命令行执行编译python setup.py build
编译成功后可以测试

```
import great_module

print(great_module.great_function(1))
```

输出: 2

另外推荐用Cython实现给Python写C/C++拓展更加高效,而且会在很多地方自动优化,效率可能会比自己写的纯C拓展更高。

Cython是在pyx后缀的文件里写的,Cython的语法是独立的需要额外学习,而且比较琐碎,可以去看官方文档或者网上其他教程,这里不细说。Ctypes可以直接调用dll/so,用中间类型的数据在c和python间通信。最高票里已经有一篇很好的介绍文章的链接。

Cython, CPython, Ctypes都是很好的工具,适用的范围不一样,关于各自的优劣,我只是初学者,不敢说太多。

上面讲的setuptools是针对distutils做了功能增的包管理工具,下面再讲几个distutils在编译生成c拓展方面的格式

直接编译pyx文件,格式

```
from distutils.core import setup
from Cython.Build import cythonize

setup(name='test', ext_modules=cythonize("test.pyx"))
```

Cython和C源码结合时一般用wrap_test的pyx文件将c源码包起来,然后编译

```
from distutils.core import setup, Extension
from Cython.Build import cythonize

tm = cythonize([Extension("wrap_test", sources=["test.c", "wrap_test.pyx"])]))
setup(ext_modules=tm,)
```

Cython结合Numpy也很方便。

如果使用numpy的话,一般setup.py要这样写

```
from distutils.core import setup, Extension
from Cython.Build import cythonize
import numpy

setup(ext_modules=cythonize("test.pyx"), include_dirs=[numpy.get_include()])
```

之前我写的一个程序里有几个地方用cython重写了numpy的方法,比如diff方法。因为np.diff默认计算差分是按照array(i+1)-array(i)的顺序计算的,而我处理的数据需要计算前一个减后一个。解决办法可以想出来很多,比如把diff得到的数组*(-1)就可以,这样大概损失3-4%的效率,直接改写numpy的源码也可以。然而后来我还是想把这一系列的算法都提速。于是用cython改写了一下numpy的diff方法。结果速度直接提升6-8倍,喜出望外。对numpy提升6-8倍就意味着对原生python快了20-100倍甚至100倍以上(视数据量和算法本身的结构,数据越多提升倍数越高,因为数据少的时候来回的数据格式转换会损失效率,一般来说提升的倍数都会有30-40倍)。后面还遇到过无法向量化的迭代问题,numpy遇到严重的效率瓶颈(甚至不如遍历python list快,因为numpy单个数值运算是弱项),然后我用cython改写了之后效率普遍提升30倍以上。我每一次处理的数据量约百万的级别,用cython改造后的速度已经非常满意了。其他的黑科技比如numba, pypy也研究了一下,但是用了之后感觉兼容性和易用性欠佳。



一个例子抛砖引玉好了。

以计算平均数为例, 如果我要用cython结合c来写的话, 需要如下四个文件。(下面的代码我没编译过, 只是举例给个思路)

1. 头文件mean_cy.h, 定义一个c的方法

```
double c_mean(double* in_array, int size);
```

2. cython本身的pyx文件mean_cy.pyx, 这里的语法是cython自己的, 需要额外学
详见[Welcome to Cython's Documentation](#)

```
import numpy as np
cimport numpy as np

np.import_array()

cdef extern from "mean_cy.h":
    double c_mean(double* in_array, int size)

def mean(np.ndarray[double, ndim=1, mode="c"] in_array not None):
    return c_mean(<double*> np.PyArray_DATA(in_array), in_array.shape[0])
```

3. 实现求平均值方法的c源码source.c

```
#include <math.h>
#include <stdio.h>
#include <malloc.h>
// #include <sys/malloc.h> // mac上的头

double c_mean(double* in_array, int size)
{
    int i;
    double sum=0;
    for(i=0; i<size; i++){
        sum += in_array[i];
    }
    return sum/size;
}
```

4. python执行编译的文件setup.py

```
from distutils.core import setup, Extension
from Cython.Distutils import build_ext
import numpy

setup(
    cmdclass={'build_ext': build_ext},
    ext_modules=[Extension("mean_cy", sources=["mean_cy.pyx", "source.c"], include_dirs=
)
```

这些文件全部放在同一文件夹下, 之后命令行进入该目录, 然后执行

python setup.py build

然后再python setup.py install

可以写个脚本试一下效果比如

```
import mean_cy
import numpy as np

a = [6, 2, 7, 5]
b = np.array(a, dtype=np.float64)

print(mean_cy.mean(b))
```

输出5





夏op

C++、Python程序员, 机器学习, Scala, Spark菜鸟,

10 人赞同了该回答

这个事情做过好多遍, 摸索的过程基本这样的:

1. 通过stdout通信...土到爆, 但上手极快, 简单粗暴;
2. 调用原始的python.h 接口, 编写可以被python import 的so, 支持python调用c++接口, c++接口调用python同样的方式;
3. 使用boost-python 完成2中的功能, 接口简单很多, 本质上没有不同;

这里遇到的主要几个问题在于:

1. 数据的序列化反序列化, 因为有时c++和python之间通信的不是基本类型, 可能是用户自定义类型;
2. 多线程的问题, c++多线程调用python接口时, 需要注意GIL的使用, 貌似因为python解释器不是线程安全的;
3. 对象传递, 大多数情况下, 如果只是静态接口调用, 都比较简单, 考虑一种情况: c++中的对象的一个函数调用python一个接口, 这个python接口中又需要反过来调用这个对象中的另一个接口, 这里就需要考虑怎么把对象相互传递, 我这里是把对象指针地址传递到python中, 在python中调用一个c++的静态接口, 带上地址和其他需要的参数, 在这个c++的静态接口中, 把地址转换成指针在调用..

上面大神提到的swig 和 cpython 没有研究过...感觉走了好多弯路...

编辑于 2015-07-24

▲ 赞同 10 ▼

● 3 条评论

➦ 分享

★ 收藏

♥ 感谢

收起 ^



gashero

编程、程序员 话题的优秀回答者

89 人赞同了该回答

Python与C/CPP的混合编程项目我做过不下20个, 积累了一些经验。2007年时分享过一篇《使用C/C++扩展Python》gashero.yeax.com/?...。

引入Python可以带来更好的可调式性。且如果重负载应用使用C/C++则基本没有性能损失, 并可以让工程师把更多精力放在算法优化获得性能优势上。

简单讲Python与C/C++的直接交互就是两种方向: C/C++写扩展模块给Python调用; 将Python嵌入C/C++。题主说的是后者。而更方便方式是前者。因为内嵌方式决定了你整个交互部分开发完成之前没法做测试。而扩展模块方式则可以先行用Python快速开发出大部分功能, 有需要性能优化的部分逐步优化到C/C++。是更加渐进式的过程。

直接用最基础的方法写扩展模块略有繁杂, 适合对细节的控制。题主时间紧迫则可以考虑Cython, 可以在较短时间里完成些任务。但更多高级功能的玩法则限制很多。

其他交互方式还有多种, 性能就不是那么高了。比如fork()子进程, 用管道通信。开独立进程走mmap()交互, 甚至是本机或其他机器上走socket。

最后, C++做了很多底层抽象, 使得其与其他编程语言的互调用方面麻烦的要死。比较典型的包括类继承, 运算符重载, 引用, 其他还有太多。这些特性使得其他语言调用C++时各种恶心。这不仅是对Python, 而是对所有语言都是如此。不信试试在C程序里调用一个C++运算符重载过的方法。所以, 如非必要, 尽量别用C++。用C简单方便的多, 而任何用以支持大规模项目的架构用Python就是了。

编辑于 2015-07-27

▲ 赞同 89 ▼

● 11 条评论

➦ 分享

★ 收藏

♥ 感谢

收起 ^



匿名用户

8 人赞同了该回答

如果计算时间长, 通信频度低的话, 有比混合编程简单得多的方法, 比如用 HTTP RPC 或者 Protocol Buffer / Thrift 之类。用 IPC 的方法还有一个额外的好处: 允许日后将计算迁移到多机环

知乎

首页发现话题

全程跟进 E3 2019 游戏展

登录加入知乎

赞同 8

6 条评论

分享

收藏

感谢

收起

