Python 的 C 扩展开发惯例

目录

1. 简介

本文记录Python的扩展模块开发中的常用惯例，以便读者可以低成本的使用Python扩展模块来提高应用性能。

文本不是扩展模块的入门教程，而是面向对扩展模块有一定概念，但一直未能有效应用起来的读者。

1.1 Python扩展模块的用途和优点

系统组件隔离和模块化

通过把每个C/C++函数提供给Python接口，使得函数之间不共享任何状态，实现了良好的组建隔离，有助于开发和测试。同时由于参数全部通过Python传递，易于打印和中断，可调试性有了很大的提高。

使用第三方库

并非每种库都有Python支持，此时就需要自己编写扩展模块实现系统对接了。但现代流行的大型库，很多都有官方的Python扩展模块，使得应用质量有了较大提高，典型如OpenCV和PyCUDA。

1.2 设计扩展模块的流程

- 编写setup.py：配置扩展模块元信息的脚本
- 引入必要头文件：以Python.h为主的几个常用头文件
- 设计导出函数表：导出给Python使用的函数表
- 模块初始化函数：几个必要的初始化步骤

2. setup.py脚本

setup.py脚本是用于构建扩展模块的配置。简单的网上有很多，而如下给出一个项目中比较实用的例子:

```
import platform
from distutils.core import setup,Extension
from distutils import sysconfig

cfg_vars=sysconfig.get_config_vars()
if 'OPT' in cfg_vars:
    cfg_vars['OPT']=cfg_vars['OPT'].replace('-Wstrict-
```

```
        os.environ['ARCHFLAGS']='-arch i386 -arch x86_64'
        extmodlist=[mod_expy,]
    else:
        raise RuntimeError('Unknown
    system()=%s'%repr(platform.system()))

    setup(name='libxx',
        version='1.0',
        description='libxx-python',
        author='gashero',
        author_email='xxx@xxx.xxx',
        packages=['libxx',],
        ext_modules=extmodlist,
        url='http://example.com/',
        long_description='xxxxxx xxxxxx',
    )
```

相对于各种Hello World级别的setup.py。如上脚本增添的实用性功能：

- 提供了更完善的编译参数，比如编译时特定的宏定义，编译器参数等
- 引用了其他第三方库
- 区分不同平台的编译，对Linux和Mac有所区分
- 提供欧冠你了更加完善的元信息

通常大家自己编写的扩展模块规模都不大，如上setup.py脚本也足够使用了。如果涉及到更复杂的多个源文件的扩展模块，只需要继续增加Extension对象即可。

3. 函数接口、参数传递、简单返回值

3.1 函数接口

用于定义函数，并传递给Python调用。就是定义PyMethodDef类型的实例。一个例子如下·

最后一个字段则是函数的文档，用于 help() 函数时显示。

3.2 参数传递

使用匿名参数和包含关键字参数的函数定义是不同的，两者如下:

```
static PyObject* func(PyObject* self, PyObject* args);
static PyObject* func(PyObject* self, PyObject* args, PyObject
**kwargs);
```

如果没有对应的内容则会传递来NULL。对于模块函数，而不是对象方法，可以不关注self。

参数解析过程使用 `PyArg_ParseTuple()` 和 `PyArg_ParseTupleAndKeywords()` 。用一个字符串来描述参数列表，随后传入各个对象的引用地址。

鉴于参数和返回值的类型可能不是对双方都友好，所以一般建议不要传递的类型：

- 体系结构相关的数字类型，如int、long，而是传递长度已知的数字类型
- 结构体，因为结构体实际存储的位置可能是有生命周期的，同样适用于C++的对象
- 地址指针，因为也是体系结构相关的，包括函数指针

实际上需要传递地址、结构体、对象时，一般建议在扩展模块里写个对象，虽然更困难一些，但是兼容性会好的多。

不要把任何内容留在C/C++程序里，如全局变量。

3.3 简单返回值

函数的返回值必须是Python对象。

简单的没有返回值的函数，可以直接在函数末尾使用一个宏来实现:

元组在Python中的作用是不可修改的，这对于返回内容给Python是很友好的，而且性能也比列表要好。

列表的价值在于可以存储长度不固定的内容。比如C/C++很多的第三方库会用迭代的方式返回内容。在函数开始时并不知道能返回多少个对象，只能每次调用next来获取下一个对象。此时就适合创建个list，然后每次用append()方法来加入元素。例如:

```
PyObject *retlist=PyList_New(0);     //创建长度为0的队列
while(pos) {
    PyList_Append(retlist, Py_BuildValue("{}"));
    pos=pos->next;
}
```

字典也可以用于返回一个单独的结构体内容。即将C/C++结构体中每个字段作为字典中的一个元素来设置，并返回。这种方式的可读性会很好。

使用dict存储信息的例子，key使用字符串:

```
PyObject *fmtinfo=PyDict_New();
PyDict_SetItemString(fmtinfo, "index", PyInt_FromLong(1));
```


但是不建议用于返回list中的dict，毕竟性能消耗较多。对于返回多个结构体，更适合的方式还是list中的tuple。

返回内容长度可预估的情况下使用 Py_BuildValue() 更合适。

对于不需要理解内容的字符串，可以使用buffer类型，会比str类型更节省内存和性能。buffer类型相对于str类型的区别是，buffer不会检查系统中内容相同的两个字符串。而str会确保系统里不会有重复的字符串。这个计算过程也是要耗时间的。

新建buffer，获取其指针后进行操作的例子:

用来通知Python，在函数里出现了错误。几个常用的内置异常：

```
PyExc_ZeroDivisionError ：被0除
PyExc_IOError ：IO错误
PyExc_TypeError ：类型错误，如参数类型不对
PyExc_ValueError ：值的范围错误
PyExc_RuntimeError ：运行时错误
PyExc_OSError ：各种与OS交互时的错误
```

可以根据自己的喜好来抛出异常。实际抛出异常的方法如：

```
if (ret<0) {
    PyErr_SetString(PyExc_RuntimeError,"Runtime failed!");
    return NULL;
}
```

有时希望抛出的异常包含一些参数，比如错误码，来方便更好的调试。可以用如下方法:

```
if (ret<0) {
    PyErr_Format(PyExc_OSError, "ERROR[%d]=%s",
            errno, strerror(errno));
}
```


各种C/C++库里大量使用了错误码，所以如上方法的使用非常广泛。在Python里看到函数出错抛出的异常，并包含了错误码是比平时写纯C/C++程序动则crash要美好的多。而编写扩展模块的过程使得这个工作规范化了。

5.2 引用计数

扩展模块的引用计数是很魔幻的，也是广为诟病的一个问题。

GIL是限制Python使用多核心的直接原因，根本原因是Python解释器内部有一些全局变量，典型如异常处理。而又有很多Python API和第三方模块在使用这些全局变量，使得GIL的改进一直得不到进展。

在扩展模块层面是可以释放GIL的，使得CPU控制权交还给Python，而当前C/C++代码也可以继续执行。但务必注意的是任何Python API的调用都必须在GIL控制下进行。所以在执行密集计算的任务前释放GIL，完成计算后，重新申请GIL，再进行返回值和异常处理。

在扩展模块的函数里释放和申请GIL的第一种写法：

```
static PyObject *fun(PyObject *self, PyObject *args) {
    //....
    PyThreadState *_save;
    _save=PyEval_SaveThread();
    block();
    PyEval_RestoreThread(_save);
    //...
}
```

该方法还需要在模块初始化时调用 PyEval_InitThreads() 。

另一种方法则简便的多:


```
Py_BEGIN_ALLOW_THREADS;
//可能阻塞的操作
Py_END_ALLOW_THREADS;
```

所以，一种简单的使用多核计算的方法是，把任务拆分成多个小份，每个小份都放在一个线程中运行。线程里调用扩展模块中的计算函数，计算函数在实际计算时释放GIL。

这种方式可以有效的利用Python来管理线程，而无需在C/C++里使用pthread之类的麻烦事，只需要最简单的做计算就好了。