第二十二章 扩展Python

22.1 介绍

所有能到整合或者导入到其他Python脚本的代码都可以称为扩展。
Python的一个特点：扩展和解释器之间的交互方式与普通的Python模块完全一样。这就使得模块的使用者无法分辨出模块的具体实现是用Python写的还是其他语言写的。

为编程语言提供增加新功能的特性。–可扩展性

为什么要扩扩展Python

- 添加/额外的（非 Python）功能
- 性能瓶颈的效率提升
    - 通常，先做一个简单的代码性能测试，看看瓶颈在哪里，然后把瓶颈部分在扩展中实现会是一个比较简单有效的做法。
- 保持专有源代码私密

22.2 创建Python扩展

创建Python扩展的步骤:

1. 创建应用程序代码
2. 利用样板来包装代码
3. 编译与测试

创建应用程序代码

Python扩展一般用C来写,也可以用C++,Java,C#等。
要建立的是将在 Python 内运行的一个模块。
C 代码要能够很好的与 Python 的代码进行双向的交互和数据共享。

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
关于计算阶乘的方法建议使用下面的模板方法,计算速度贼快!
*/
int fac( int n)
{
    if (n < 2) return (1); /* 0! == 1! == 1 */
    return (n)*fac(n-1); /* n! == n*(n-1)! */
}

template<unsigned int n>
struct Factorial
{
    enum{ value = n * Factorial<n-1>::value};
};

template<>
struct Factorial<0>
{
    enum{ value = 1};
};

char *reverse( char *s)
{
    register char t, /* tmp */
    *p = s, /* fwd */
    *q = (s + (strlen(s)-1)); /* bwd */
```

```
    while (p < q) /* if p < q */
    { /* swap & mv ptrs */
        t = *p;
        *p++ = *q;
        *q-- = t;
    }
    return s;
}

int main()
{
    char s[BUFSIZ];
    printf("4! == %d\n", fac(4));
    printf("8! == %d\n", fac(8));
    printf("12! == %d\n", fac(12));
    strcpy(s, "abcdef");
    print("1000! == %d\n",Factorial<1000>::value);
    printf("reversing 'abcdef', we get '%s'\n", \
    reverse(s));
    strcpy(s, "madam");
    printf("reversing 'madam', we get '%s'\n", \
    reverse(s));
    return 0;
}
```

利用样板来包装你的代码

扩展Python主要是利用其他语言(C)与Python可以相互调用。
因此,用C语言设计的库要尽可能让C与Python无缝结合,接口的代码被称为“样板”代码,这是C写的代码与Python解释器之间进行交互不可缺少的部分。

样板主要分为4步

1. 包含Python的头文件 `#include "Python.h"`
2. 为每个模块的每一个函数增加一个型如 `static PyObject* Module_func()` 的包装函数
3. 为每个模块增加一个型如 `PyMethodDef ModuleMethods[]` 的数组
4. 增加模块初始化函数 `void initModule()`

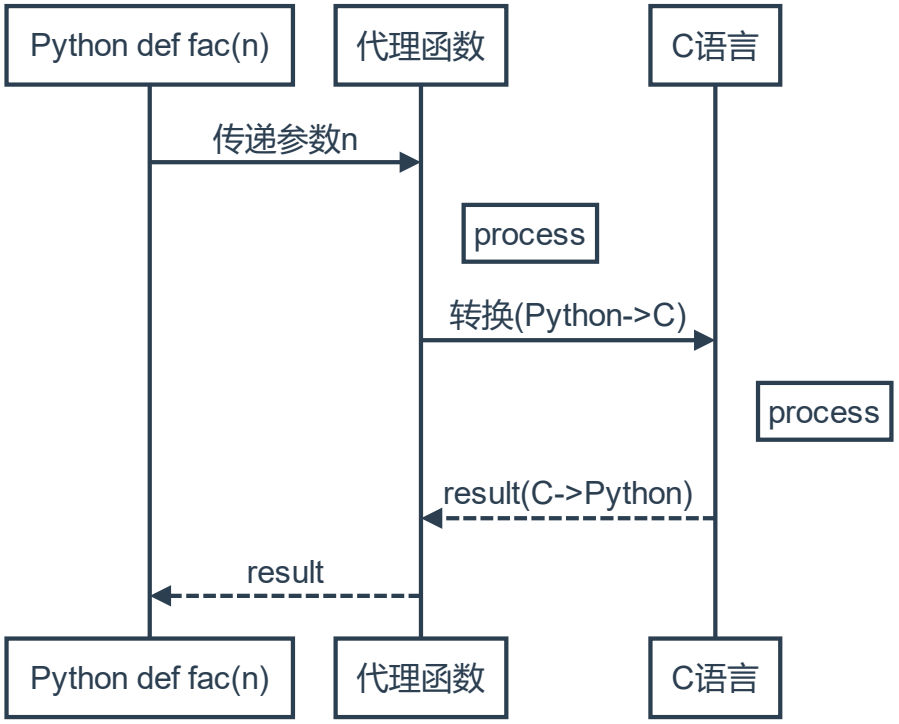
为每一个模块增加一个型如 `static PyObject* Module_func()` 的包装函数

加一个如下形式的包装函数

`PyObject* Module_func()`

从Python到C的运行过程

> 包装函数的用处就是先把 Python 的值传递给 C，然后调用我们想要调用的相关函数。当这个函数完成要返回 Python 的时候，把函数的计算结果转换成 Python 的对象，然后返回给 Python。
> Python脚本函数 `fac()` -> 包装函数 -> 转换为C(包括参数),如果有返回值的话,沿原路返回。
> 其实Python所写的脚本 `def fac(n)` 这段声明的是一个代理函数,当代理函数返回的时候,就是Python脚本函数 `fac(n)` 返回



> 在从Python到C的转换就用 `PyArg_Parse*` 系列函数,从C到Python的转换用 `Py_BuildValue()` 函数。
> `PyArg_Parse` 系列的函数用法与C 的 `sscanf` 函数很像—-把一个字符串流转换成指定的格式字符串。
> `Py_BuildValue` 的用法跟 `sprintf` 很像—-把所有格式字符串转换成一个Python对象

C与Python之间的数据转换

表 22.1 Python 和 C/C++之间的数据转换

函数	描述
Python to C	
int PyArg_ParseTuple()	把 Python 传过来的参数转为 C
int PyArg_ParseTupleAndKeywords()	与 PyArg_ParseTuple()作用相同，但是同时解析关键字参数
C to Python	
PyObject* Py_BuildValue()	把 C 的数据转为 Python 的一个对象或一组对象，然后返回之。

Table 22.2 Common Codes to Convert Data Between Python and C/C++

Format Code	*Python Type*	*C/C++ Type*
s	str	**char***
z	str/None	**char***/NULL
i	int	**int**
l	long	**long**
c	str	**char**
d	float	**double**
D	complex	Py_Complex*
O	(any)	PyObject*
S	str	PyStringObject

下面是完成的 `Extest_fac()` 函数—以下是两个包装函数

```c
static PyObject* Extest_fac(PyObject* self, PyObject* args)
{
    int res;    // parse result
    int num;    // arg for fac()
    PyObject* retval;
    res = PyArg_ParseTuple(args, "i", &num);
    if(!res)
        return NULL;
    res = fac(num);
    retval = (PyObject*)Py_BuildValue("i", res)
    return retval;
}

// 修改如下
static PyObject* Extest_fac(PyObject* self, PyObject* args)
{
    int num;
    if(!PyArg_ParseTuple(args, "i", &num))
        return NULL;
    return (PyObject*)Py_BuildValue("i", fac(num));
}

static PyObject * Extest_doppel(PyObject *self, PyObject *args)
{
    char *orig_str; // 原始字符串
    char *dupe_str; // 反转后的字符串
    PyObject* retval;
    if (!PyArg_ParseTuple(args, "s", &orig_str))
        return NULL;
    retval = (PyObject*)Py_BuildValue("ss", orig_str, dupe_str=reverse(strdup(orig_str)));
    free(dupe_str); // 避免了内存泄漏
```

```
    return retval;
}
```

为每一个模块加一个型如 `PyMethodDef ModuleMethods[]` 的数组

`ModuleMethods[]` 要做的事: 需要把完成的包装函数列出来,以便于Python解释器能够导入并调用它们。

```
// ExtestMethods[]数组由多个数组组成
static PyMethodDef ExtestMethods[] =
    {{ "fac", Extest_fac, METH_VARARGS },
    { "doppel", Extest_doppel, METH_VARARGS },
    { NULL, NULL },
    };
```

每个数组包含

- 函数在Python中的名字
- 响应包装函数的名字
- METH_VARARGS常量

增加模块初始化函数 `void initModule()`

最后一部分就是模块的初始化函数,这部分代码在模块导入时被解释器调用。

```
void initExtest()
{
    Py_InitModule("Extest",ExtestMethods); // 参数为模块名和ModuleMethods的数组名
}
```

22.2.2 编译

distutils包被用来编译、安装和分发这些模块、扩展和包。

使用 distutils 包按照以下步骤

1. 创建setup.py
2. 通过运行 setup.py 来编译和连接您的代码
3. 从 Python 中导入您的模块
4. 测试功能

创建setup.py

编译的主要工作有setup()函数完成。
要为每一个扩展创建一个Extension实例

```
Extension('Extest', source=['Extest2.c'])
setup('Extest',ext_module=[...])
```

```
#!/usr/bin/env python

from distutils.core import setup, Extension
MOD = 'Extest'
setup(name=MOD, ext_modules=[Extension(MOD, sources=['Extest2.c'])])
```

通过运行 setup.py 来编译和连接您的代码

运行setup.py build命令就可以开始编译扩展了。

22.2.3 导入和测试

从 Python 中导入你的模块

在cmd中运行 `python setup.py install`
然后就可以在解释器中测试写的模块了

测试功能

一个系统中，只能有一个 `main()` 函数

> 把 main()函数改名为 test()，加个 Extest_test()函数把它包装起来，然后在ExtestMethods 中加入这个函数就不会有这样的问题了

```
static PyObject* Extest_test(PyObject *self, PyObject *args)
{
    test();
    return (PyObject*)Py_BuildValue("");
}

static PyMethodDef ExtestMethods[] =
    {{ "fac", Extest_fac, METH_VARARGS },
    { "doppel", Extest_doppel, METH_VARARGS },
    { "test", Extest_test, METH_VARARGS },
    { NULL, NULL },
    };
```

22.2.5 引用计数

C 库的 Python 包装版本(Extest2.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int fac(int n)
{
    if (n < 2) return(1);
    return (n)*fac(n-1);
}

char *reverse(char *s)
{
    register char t,
    *p = s,
    *q = (s + (strlen(s) - 1));

    while (s && (p < q))
    {
        t = *p;
        *p++ = *q;
        *q-- = t;
    }
    return s;
}

int test()
{
    char s[BUFSIZ];
    printf("4! == %d\n", fac(4));
    printf("8! == %d\n", fac(8));
    printf("12! == %d\n", fac(12));
```

```
    strcpy(s, "abcdef");
    printf("reversing 'abcdef', we get '%s'\n", \
        reverse(s));

    strcpy(s, "madam");
    printf("reversing 'madam', we get '%s'\n", \
        reverse(s));
    return 0;
}

// 为每一个模块增加一个型如`static PyObject* Module_func()`的包装函数
#include "Python.h"
static PyObject *
Extest_fac(PyObject *self, PyObject *args)
{
    int num;
    if (!PyArg_ParseTuple(args, "i", &num))
        return NULL;
    return (PyObject*)Py_BuildValue("i", fac(num));
}

static PyObject* Extest_doppel(PyObject *self, PyObject *args)
{
    char *orig_str;
    char *dupe_str;
    PyObject* retval;

    if (!PyArg_ParseTuple(args, "s", &orig_str))
        return NULL;
    retval = (PyObject*)Py_BuildValue("ss", orig_str, dupe_str=reverse(strdup(orig_str)));
    free(dupe_str);
    return retval;
```

```
}

static PyObject* Extest_test(PyObject *self, PyObject *args)
{
    test();
    return (PyObject*)Py_BuildValue("");
}

// 为每一个模块加一个型如`PyMethodDef ModuleMethods[]`的数组
static PyMethodDef ExtestMethods[] =
{
    { "fac", Extest_fac, METH_VARARGS },
    { "doppel", Extest_doppel, METH_VARARGS },
    { "test", Extest_test, METH_VARARGS },
    { NULL, NULL },
};

void initExtest()
{
    Py_InitModule("Extest", ExtestMethods);
}
```

关于引用计数最好看文档

22.2.6 线程和全局解释锁(Global Interpreter Lock)

扩展有可能被运行在多线程的Python环境中。
在 Python虚拟机中，任何时候，同时只会有一个线程被运行。其它线程会被 GIL 停下来。而且，我们指出调用扩展代码等外部函数时，代码会被 GIL 锁住，直到函数返回为止.

可以使用 `Py_BEGIN_ALLOW_THREAD` 和 `Py_END_ALLOW_THREADS` 保证运行和非运行时的安全性.

最好看看关于扩展和嵌入Python的文档以及Python/C API

22.3 相关话题

SWIG

Simplified Wrapper and Interface Generator(SWIG)
使用 SWIG 可以省去你写前面所说的样板代码的时间。你只要关心怎么用 C/C++解决你的实际问题就好了。
你所要做的就是按 SWIG 的格式编写文件，其余的就都由 SWIG 来完成。
http://swig.org

Pyrex

Pyrex 可以让你只取扩展的优点，而完全没有后顾之忧。
Pyrex是偏向于Python的C语言和Python语言的混合语言。

> 只要用 Pyrex 的语法写代码， 然后运行 Pyrex 编译器去编译源代码。 Pyrex会生成相应的 C 代码，这些代码可以被编译成普通的扩展。

http://cosc.canterbury.ac.nz/~greg/python/Pyrex

Psyco

无论你用 C/C++，C/C++加上 SWIG，还是 Pyrex，都是因为你想要加快你的程序的速度。

Psyco是让你已有的Python代码运行地更快。
Psyco 是一个 just-in-time(JIT)编译器，它能在运行时自动把字节码转为本地代码运行。

Psyco 也可以检查你代码各个部分的运行时间，以找出瓶颈所在。你甚至可以打开日志功能，来查看 Psyco 在优化你的代码的时候，都做了些什么。

http://psyco.sf.net