15.2 简单的C扩展模块¶

问题¶

你想不依靠其他工具，直接使用Python的扩展API来编写一些简单的C扩展模块。

解决方案¶

对于简单的C代码，构建一个自定义扩展模块是很容易的。 作为第一步，你需要确保你的C代码有一个正确的头文件。例如：

```
/* sample.h */

#include <math.h>

extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

通常来讲，这个头文件要对应一个已经被单独编译过的库。 有了这些，下面我们演示下编写扩展函数的一个简单例子：

```
#include "Python.h"
#include "sample.h"

/* int gcd(int, int) */
static PyObject *py_gcd(PyObject *self, PyObject *args) {
  int x, y, result;

  if (!PyArg_ParseTuple(args,"ii", &x, &y)) {
    return NULL;
  }
  result = gcd(x,y);
  return Py_BuildValue("i", result);
}

/* int in_mandel(double, double, int) */
static PyObject *py_in_mandel(PyObject *self, PyObject *args) {
  double x0, y0;
  int n;
  int result;

  if (!PyArg_ParseTuple(args, "ddi", &x0, &y0, &n)) {
    return NULL;
  }
  result = in_mandel(x0,y0,n);
  return Py_BuildValue("i", result);
}

/* int divide(int, int, int *) */
static PyObject *py_divide(PyObject *self, PyObject *args) {
  int a, b, quotient, remainder;
  if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
    return NULL;
  }
  quotient = divide(a,b, &remainder);
  return Py_BuildValue("(ii)", quotient, remainder);
}
```

```
/* Module method table */
static PyMethodDef SampleMethods[] = {
  {"gcd",  py_gcd, METH_VARARGS, "Greatest common divisor"},
  {"in_mandel", py_in_mandel, METH_VARARGS, "Mandelbrot test"},
  {"divide", py_divide, METH_VARARGS, "Integer division"},
  { NULL, NULL, 0, NULL}
};

/* Module structure */
static struct PyModuleDef samplemodule = {
  PyModuleDef_HEAD_INIT,

  "sample",           /* name of module */
  "A sample module",  /* Doc string (may be NULL) */
  -1,                 /* Size of per-interpreter state or -1 */
  SampleMethods       /* Method table */
};

/* Module initialization function */
PyMODINIT_FUNC
PyInit_sample(void) {
  return PyModule_Create(&samplemodule);
}
```

要绑定这个扩展模块，像下面这样创建一个 `setup.py` 文件：

```
# setup.py
from distutils.core import setup, Extension

setup(name='sample',
      ext_modules=[
        Extension('sample',
                  ['pysample.c'],
                  include_dirs = ['/some/dir'],
                  define_macros = [('FOO','1')],
                  undef_macros = ['BAR'],
                  library_dirs = ['/usr/local/lib'],
                  libraries = ['sample']
                  )
        ]
)
```

为了构建最终的函数库，只需简单的使用 `python3 buildlib.py build_ext --inplace` 命令即可：

```
bash % python3 setup.py build_ext --inplace
running build_ext
building 'sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
 -I/usr/local/include/python3.3m -c pysample.c
 -o build/temp.macosx-10.6-x86_64-3.3/pysample.o
gcc -bundle -undefined dynamic_lookup
build/temp.macosx-10.6-x86_64-3.3/pysample.o \
 -L/usr/local/lib -lsample -o sample.so
bash %
```

如上所示，它会创建一个名字叫 `sample.so` 的共享库。当被编译后，你就能将它作为一个模块导入进来了：

```
>>> import sample
>>> sample.gcd(35, 42)
7
>>> sample.in_mandel(0, 0, 500)
1
>>> sample.in_mandel(2.0, 1.0, 500)

0
>>> sample.divide(42, 8)
(5, 2)
>>>
```

如果你是在Windows机器上面尝试这些步骤，可能会遇到各种环境和编译问题，你需要花更多点时间去配置。 Python的

二进制分发通常使用了Microsoft Visual Studio来构建。 为了让这些扩展能正常工作，你需要使用同样或兼容的工具来编译它。 参考相应的 Python文档

讨论¶

在尝试任何手写扩展之前，最好能先参考下Python文档中的 扩展和嵌入Python解释器 . Python的C扩展API很大，在这里整个去讲述它没什么实际意义。 不过对于最核心的部分还是可以讨论下的。

首先，在扩展模块中，你写的函数都是像下面这样的一个普通原型：

```
static PyObject *py_func(PyObject *self, PyObject *args) {
  ...
}
```

`PyObject` 是一个能表示任何Python对象的C数据类型。 在一个高级层面，一个扩展函数就是一个接受一个Python对象（在 PyObject *args中）元组并返回一个新Python对象的C函数。 函数的 `self` 参数对于简单的扩展函数没有被使用到，不过如果你想定义新的类或者是C中的对象类型的话就能派上用场了。比如如果扩展函数是一个类的一个方法， 那么 `self` 就能引用那个实例了。

`PyArg_ParseTuple()` 函数被用来将Python中的值转换成C中对应表示。 它接受一个指定输入格式的格式化字符串作为输入，比如“i”代表整数，“d”代表双精度浮点数， 同样还有存放转换后结果的C变量的地址。 如果输入的值不匹配这个格式化字符串，就会抛出一个异常并返回一个NULL值。 通过检查并返回NULL，一个合适的异常会在调用代码中被抛出。

`Py_BuildValue()` 函数被用来根据C数据类型创建Python对象。 它同样接受一个格式化字符串来指定期望类型。 在扩展函数中，它被用来返回结果给Python。 `Py_BuildValue()` 的一个特性是它能构建更加复杂的对象类型，比如元组和字典。 在 `py_divide()` 代码中，一个例子演示了怎样返回一个元组。不过，下面还有一些实例：

```
return Py_BuildValue("i", 34);      // Return an integer
return Py_BuildValue("d", 3.4);     // Return a double
return Py_BuildValue("s", "Hello"); // Null-terminated UTF-8 string
return Py_BuildValue("(ii)", 3, 4); // Tuple (3, 4)
```

在扩展模块底部，你会发现一个函数表，比如本节中的 `SampleMethods` 表。 这个表可以列出C函数、Python中使用的名字、文档字符串。 所有模块都需要指定这个表，因为它在模块初始化时要被使用到。

最后的函数 `PyInit_sample()` 是模块初始化函数，但该模块第一次被导入时执行。 这个函数的主要工作是在解释器中注册模块对象。

最后一个要点需要提出来，使用C函数来扩展Python要考虑的事情还有很多，本节只是一小部分。 （实际上，C API包含了超过500个函数）。你应该将本节当做是一个入门篇。 更多高级内容，可以看看 `PyArg_ParseTuple()` 和 `Py_BuildValue()` 函数的文档， 然后进一步扩展开。