15.9 用SWIG包装C代码¶

问题¶

你想让你写的C代码作为一个C扩展模块来访问，想通过使用 Swig包装生成器 来完成。

解决方案¶

Swig通过解析C头文件并自动创建扩展代码来操作。 要使用它，你先要有一个C头文件。例如，我们示例的头文件如下：

```
/* 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);
```

一旦你有了这个头文件，下一步就是编写一个Swig”接口”文件。 按照约定，这些文件以”.i”后缀并且类似下面这样：

```
// sample.i - Swig interface
%module sample
%{
#include "sample.h"
%}

/* Customizations */
%extend Point {
    /* Constructor for Point objects */
    Point(double x, double y) {
        Point *p = (Point *) malloc(sizeof(Point));
        p->x = x;
        p->y = y;
        return p;
   };
};

/* Map int *remainder as an output argument */
%include typemaps.i
%apply int *OUTPUT { int * remainder };

/* Map the argument pattern (double *a, int n) to arrays */
%typemap(in) (double *a, int n)(Py_buffer view) {
  view.obj = NULL;
  if (PyObject_GetBuffer($input, &view, PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT) == -1) {
    SWIG_fail;
  }
  if (strcmp(view.format,"d") != 0) {
    PyErr_SetString(PyExc_TypeError, "Expected an array of doubles");
    SWIG_fail;
  }
  $1 = (double *) view.buf;
  $2 = view.len / sizeof(double);
}

%typemap(freearg) (double *a, int n) {
  if (view$argnum.obj) {
    PyBuffer_Release(&view$argnum);
  }
```

```
}

/* C declarations to be included in the extension module */

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);
```

一旦你写好了接口文件，就可以在命令行工具中调用Swig了：

```
bash % swig -python -py3 sample.i
bash %
```

swig的输出就是两个文件，sample_wrap.c和sample.py。 后面的文件就是用户需要导入的。 而sample_wrap.c文件是需要被编译到名叫 `_sample` 的支持模块的C代码。 这个可以通过跟普通扩展模块一样的技术来完成。 例如，你创建了一个如下所示的 `setup.py` 文件：

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

setup(name='sample',
      py_modules=['sample.py'],
      ext_modules=[
        Extension('_sample',
                  ['sample_wrap.c'],
                  include_dirs = [],
                  define_macros = [],

                  undef_macros = [],
                  library_dirs = [],
                  libraries = ['sample']
                  )
        ]
)
```

要编译和测试，在setup.py上执行python3，如下：

```
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 sample_wrap.c
 -o build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o
sample_wrap.c: In function ‘SWIG_InitializeModule’:
sample_wrap.c:3589: warning: statement with no effect
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_64-3.3/sample.o
 build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o -o _sample.so -lsample
bash %
```

如果一切正常的话，你会发现你就可以很方便的使用生成的C扩展模块了。例如：

```
>>> import sample
>>> sample.gcd(42,8)
2
>>> sample.divide(42,8)
[5, 2]
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
2.8284271247461903
>>> p1.x
2.0
```

```
>>> p1.y
3.0
>>> import array
>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>>
```

讨论¶

Swig是Python历史中构建扩展模块的最古老的工具之一。 Swig能自动化很多包装生成器的处理。

所有Swig接口都以类似下面这样的为开头：

```
%module sample
%{
#include "sample.h"
%}
```

这个仅仅只是声明了扩展模块的名称并指定了C头文件， 为了能让编译通过必须要包含这些头文件（位于 %{ 和 %} 的代码）， 将它们之间复制粘贴到输出代码中，这也是你要放置所有包含文件和其他编译需要的定义的地方。

Swig接口的底下部分是一个C声明列表，你需要在扩展中包含它。 这通常从头文件中被复制。在我们的例子中，我们仅仅像下面这样直接粘贴在头文件中：

```
%module sample
%{
#include "sample.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);
```

有一点需要强调的是这些声明会告诉Swig你想要在Python模块中包含哪些东西。 通常你需要编辑这个声明列表或相应的修改下它。 例如，如果你不想某些声明被包含进来，你要将它从声明列表中移除掉。

使用Swig最复杂的地方是它能给C代码提供大量的自定义操作。 这个主题太大，这里无法展开，但是我们在本节还剩展示了一些自定义的东西。

第一个自定义是 `%extend` 指令允许方法被附加到已存在的结构体和类定义上。 我例子中，这个被用来添加一个Point结构体的构造器方法。 它可以让你像下面这样使用这个结构体：

```
>>> p1 = sample.Point(2,3)
>>>
```

如果略过的话，Point对象就必须以更加复杂的方式来被创建：

```
>>> # Usage if %extend Point is omitted
>>> p1 = sample.Point()
>>> p1.x = 2.0
>>> p1.y = 3
```

第二个自定义涉及到对 `typemaps.i` 库的引入和 `%apply` 指令， 它会指示Swig参数签名 `int *remainder` 要被当做是输出值。 这个实际上是一个模式匹配规则。 在接下来的所有声明中，任何时候只要碰上 `int *remainder`，他就会被作为输出。 这个自定义方法可以让 `divide()` 函数返回两个值。

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

```
>>>
```

最后一个涉及到 `%typemap` 指令的自定义可能是这里展示的最高级的特性了。 一个typemap就是一个在输入中特定参数模式的规则。 在本节中，一个typemap被定义为匹配参数模式 `(double *a, int n)`. 在typemap内部是一个C代码片段，它告诉Swig怎样将一个Python对象转换为相应的C参数。 本节代码使用了Python的缓存协议去匹配任何看上去类似双精度数组的输入参数 （比如NumPy数组、array模块创建的数组等），更多请参考15.3小节。

在typemap代码内部，$1和$2这样的变量替换会获取typemap模式的C参数值 （比如$1映射为 `double *a` ）。$input指向一个作为输入的 `PyObject *` 参数， 而 `$argnum` 就代表参数的个数。

编写和理解typemaps是使用Swig最基本的前提。 不仅是说代码更神秘，而且你需要理解Python C API和Swig和它交互的方式。 Swig文档有更多这方面的细节，可以参考下。

不过，如果你有大量的C代码需要被暴露为扩展模块。 Swig是一个非常强大的工具。关键点在于Swig是一个处理C声明的编译器， 通过强大的模式匹配和自定义组件，可以让你更改声明指定和类型处理方式。 更多信息请去查阅 Swig网站， 还有 特定于Python的相关文档