15.1 使用ctypes访问C代码¶

问题¶

你有一些C函数已经被编译到共享库或DLL中。你希望可以使用纯Python代码调用这些函数， 而不用编写额外的C代码或使用第三方扩展工具。

解决方案¶

对于需要调用C代码的一些小的问题，通常使用Python标准库中的 `ctypes` 模块就足够了。 要使用 `ctypes` ，你首先要确保你要访问的C代码已经被编译到和Python解释器兼容 （同样的架构、字大小、编译器等）的某个共享库中了。 为了进行本节的演示，假设你有一个共享库名字叫 `libsample.so` ，里面的内容就是15章介绍部分那样。 另外还假设这个 `libsample.so` 文件被放置到位于 `sample.py` 文件相同的目录中了。

要访问这个函数库，你要先构建一个包装它的Python模块，如下这样：

```
# sample.py
import ctypes
import os

# Try to locate the .so file in the same directory as this file
_file = 'libsample.so'
_path = os.path.join(*(os.path.split(__file__)[:-1] + (_file,)))
_mod = ctypes.cdll.LoadLibrary(_path)

# int gcd(int, int)
gcd = _mod.gcd
gcd.argtypes = (ctypes.c_int, ctypes.c_int)
gcd.restype = ctypes.c_int

# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int

# int divide(int, int, int *)
_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int

def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x, y, rem)

    return quot,rem.value

# void avg(double *, int n)
# Define a special type for the 'double *' argument
class DoubleArrayType:
    def from_param(self, param):
        typename = type(param).__name__
        if hasattr(self, 'from_' + typename):
            return getattr(self, 'from_' + typename)(param)
        elif isinstance(param, ctypes.Array):
            return param
        else:
            raise TypeError("Can't convert %s" % typename)

    # Cast from array.array objects
    def from_array(self, param):
        if param.typecode != 'd':
            raise TypeError('must be an array of doubles')
        ptr, _ = param.buffer_info()
        return ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))
```

```
    # Cast from lists/tuples
    def from_list(self, param):
        val = ((ctypes.c_double)*len(param))(*param)
        return val

    from_tuple = from_list

    # Cast from a numpy array
    def from_ndarray(self, param):
        return param.ctypes.data_as(ctypes.POINTER(ctypes.c_double))

DoubleArray = DoubleArrayType()
_avg = _mod.avg
_avg.argtypes = (DoubleArray, ctypes.c_int)
_avg.restype = ctypes.c_double

def avg(values):
    return _avg(values, len(values))

# struct Point { }
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]

# double distance(Point *, Point *)
distance = _mod.distance
distance.argtypes = (ctypes.POINTER(Point), ctypes.POINTER(Point))
distance.restype = ctypes.c_double
```

如果一切正常，你就可以加载并使用里面定义的C函数了。例如：

```
>>> 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)
>>> sample.avg([1,2,3])
2.0
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
4.242640687119285
>>>
```

讨论¶

本小节有很多值得我们详细讨论的地方。 首先是对于C和Python代码一起打包的问题，如果你在使用 `ctypes` 来访问编译后的C代码， 那么需要确保这个共享库放在 `sample.py` 模块同一个地方。 一种可能是将生成的 `.so` 文件放置在要使用它的Python代码同一个目录下。 我们在 `recipe-sample.py` 中使用 `__file__` 变量来查看它被安装的位置， 然后构造一个指向同一个目录中的 `libsample.so` 文件的路径。

如果C函数库被安装到其他地方，那么你就要修改相应的路径。 如果C函数库在你机器上被安装为一个标准库了， 那么可以使用 `ctypes.util.find_library()` 函数来查找：

```
>>> from ctypes.util import find_library
>>> find_library('m')
'/usr/lib/libm.dylib'
>>> find_library('pthread')
'/usr/lib/libpthread.dylib'
>>> find_library('sample')
'/usr/local/lib/libsample.so'
>>>
```

一旦你知道了C函数库的位置，那么就可以像下面这样使用 `ctypes.cdll.LoadLibrary()` 来加载它， 其中 `_path` 是标

准库的全路径：

```
_mod = ctypes.cdll.LoadLibrary(_path)
```

函数库被加载后，你需要编写几个语句来提取特定的符号并指定它们的类型。 就像下面这个代码片段一样：

```
# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int
```

在这段代码中，`.argtypes` 属性是一个元组，包含了某个函数的输入按时， 而 `.restype` 就是相应的返回类型。 `ctypes` 定义了大量的类型对象（比如c_double, c_int, c_short, c_float等）， 代表了对应的C数据类型。如果你想让Python能够传递正确的参数类型并且正确的转换数据的话， 那么这些类型签名的绑定是很重要的一步。如果你没有这么做，不但代码不能正常运行， 还可能会导致整个解释器进程挂掉。 使用ctypes有一个麻烦点的地方是原生的C代码使用的术语可能跟Python不能明确的对应上来。 `divide()` 函数是一个很好的例子，它通过一个参数除以另一个参数返回一个结果值。 尽管这是一个很常见的C技术，但是在Python中却不知道怎样清晰的表达出来。 例如，你不能像下面这样简单的做：

```
>>> divide = _mod.divide
>>> divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
>>> x = 0
>>> divide(10, 3, x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 3: <class 'TypeError'>: expected LP_c_int
instance instead of int
>>>
```

就算这个能正确的工作，它会违反Python对于整数的不可更改原则，并且可能会导致整个解释器陷入一个黑洞中。 对于涉及到指针的参数，你通常需要先构建一个相应的ctypes对象并像下面这样传进去：

```
>>> x = ctypes.c_int()
>>> divide(10, 3, x)
3
>>> x.value
1
>>>
```

在这里，一个 `ctypes.c_int` 实例被创建并作为一个指针被传进去。 跟普通Python整形不同的是，一个 `c_int` 对象是可以被修改的。 `.value` 属性可被用来获取或更改这个值。

对于那些不像Python的C调用，通常可以写一个小的包装函数。 这里，我们让 `divide()` 函数通过元组来返回两个结果：

```
# int divide(int, int, int *)
_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int

def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x,y,rem)
    return quot, rem.value
```

`avg()` 函数又是一个新的挑战。C代码期望接受到一个指针和一个数组的长度值。 但是，在Python中，我们必须考虑这个问题：数组是啥？它是一个列表？一个元组？ 还是 `array` 模块中的一个数组？还是一个 `numpy` 数组？还是说所有都是？ 实际上，一个Python“数组”有多种形式，你可能想要支持多种可能性。

`DoubleArrayType` 演示了怎样处理这种情况。 在这个类中定义了一个单个方法 `from_param()` 。 这个方法的角色是接受一个单个参数然后将其向下转换为一个合适的ctypes对象 （本例中是一个 `ctypes.c_double` 的指针）。 在 `from_param()` 中，你可以做任何你想做的事。 参数的类型名被提取出来并被用于分发到一个更具体的方法中去。 例如，如果一个列表被传递过来，那么 `typename` 就是 `list` ， 然后 `from_list` 方法被调用。

对于列表和元组，`from_list` 方法将其转换为一个 `ctypes` 的数组对象。 这个看上去有点奇怪，下面我们使用一个交互

式例子来将一个列表转换为一个 ctypes 数组：

```
>>> nums = [1, 2, 3]
>>> a = (ctypes.c_double * len(nums))(*nums)
>>> a
<__main__.c_double_Array_3 object at 0x10069cd40>
>>> a[0]
1.0
>>> a[1]
2.0
>>> a[2]
3.0
>>>
```

对于数组对象，from_array() 提取底层的内存指针并将其转换为一个 ctypes 指针对象。例如：

```
>>> import array
>>> a = array.array('d',[1,2,3])
>>> a
array('d', [1.0, 2.0, 3.0])
>>> ptr_ = a.buffer_info()
>>> ptr
4298687200
>>> ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))
<__main__.LP_c_double object at 0x10069cd40>
>>>
```

from_ndarray() 演示了对于 numpy 数组的转换操作。 通过定义 DoubleArrayType 类并在 avg() 类型签名中使用它，那么这个函数就能接受多个不同的类数组输入了：

```
>>> import sample
>>> sample.avg([1,2,3])
2.0
>>> sample.avg((1,2,3))
2.0
>>> import array
>>> sample.avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> sample.avg(numpy.array([1.0,2.0,3.0]))
2.0
>>>
```

本节最后一部分向你演示了怎样处理一个简单的C结构。 对于结构体，你只需要像下面这样简单的定义一个类，包含相应的字段和类型即可：

```
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]
```

一旦类被定义后，你就可以在类型签名中或者是需要实例化结构体的代码中使用它。例如：

```
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> p1.x
1.0
>>> p1.y
2.0
>>> sample.distance(p1,p2)
4.242640687119285
>>>
```

最后一些小的提示：如果你想在Python中访问一些小的C函数，那么 ctypes 是一个很有用的函数库。 尽管如此，如果你想要去访问一个很大的库，那么可能就需要其他的方法了，比如 Swig (15.9节会讲到) 或 Cython（15.10节）。

对于大型库的访问有个主要问题，由于ctypes并不是完全自动化， 那么你就必须花费大量时间来编写所有的类型签名，就像例子中那样。 如果函数库够复杂，你还得去编写很多小的包装函数和支持类。 另外，除非你已经完全精通了所有底层的C接口细节，包括内存分配和错误处理机制， 通常一个很小的代码缺陷、访问越界或其他类似错误就能让

Python程序奔溃。

作为 `ctypes` 的一个替代，你还可以考虑下CFFI。CFFI提供了很多类似的功能， 但是使用C语法并支持更多高级的C代码类型。 到写这本书为止，CFFI还是一个相对较新的工程， 但是它的流行度正在快速上升。 甚至还有在讨论在Python将来的版本中将它包含进去。因此，这个真的值得一看。