聊聊Python ctypes 模块

https://zhuanlan.zhihu.com/p/20152309

None Thu Aug, 19 12:21

摘要：模块ctypes是Python内建的用于调用动态链接库函数的功能模块，一定程度上可以用于Python与其他语言的混合编程。由于编写动态链接库，使用C/C++是最常见的方式，故ctypes最常用于Python与C/C++混合编程之中。

==========

1. ctypes 的原理以及优缺点

从ctypes的文档中可以推断，在各个平台上均使用了对应平台动态加载动态链接库的方法，并通过一套类型映射的方式将Python与二进制动态链接库相连接。通过阅读ctypes本身的代码也可以印证这个推断（/Modules/_ctypes/_ctypes.c和/Modules/_ctypes/callproc.c）。在Windows平台下，最终调用的是Windows API中LoadLibrary函数和GetProcAddress函数，在Linux和Mac OS X平台下，最终调用的是Posix标准中的dlopen和dlsym函数。ctypes 实现了一系列的类型转换方法，Python的数据类型会包装或直接推算为C类型，作为函数的调用参数；函数的返回值也经过一系列的包装成为Python类型。也就是说，PyObject* <-> C types的转换是由ctypes内部完成的，这和SWIG是同一个原理。

从ctypes的实现原理不难看出：

ctypes 有以下优点：

- Python内建，不需要单独安装
- 可以直接调用二进制的动态链接库
- 在Python一侧，不需要了解Python内部的工作方式
- 在C/C++一侧，也不需要了解Python内部的工作方式
- 对基本类型的相互映射有良好的支持

ctypes 有以下缺点：

- 平台兼容性差
- 不能够直接调用动态链接库中未经导出的函数或变量
- 对C++的支持差

就个人的经验来看，ctypes 适合于“中轻量级”的Python C/C++混合编程。特别是遇到第三方库提供动态链接库和调用文档，且没有编译器或编译器并不互相兼容的场合下，使用ctypes特别方便。值得注意的是，对于某种需求，在Python本身就可以实现的情况下（例如获取系统时间、读写文件等），应该优先使用Python自身的功能而不要使用操作系统提供的API接口，否则你的程序会丧失跨平台的特性。

2. 一个简单的例子

作为Python文档的一部分，ctypes 提供了完善的文档。但没有Windows API编程经验的初学者读ctypes文档仍然会晕头转向。这里举一个小例子，尽力避开Windows API以及POSIX本身的复杂性，读者只需要了解C语言即可。

在尝试本节例子之前，依然要搭建Python扩展编程环境。见 [搭建Python扩展开发环境 - 蛇之魅惑 - 知乎专栏]

首先我们写一个C语言的小程序，然后把它编译成动态链接库。

```
//great_module.c
#include <nmmintrin.h>

#ifdef _MSC_VER
    #define DLL_EXPORT __declspec( dllexport )
#else
    #define DLL_EXPORT
#endif

DLL_EXPORT int great_function(unsigned int n) {
    return _mm_popcnt_u32(n);
}
```

这个源文件中只有一个函数 great_function，它会调用Intel SSE4.2指令集的POPCNT指令（封装在_mm_popcnt_u32中），即计算一个无符号整数的二进制表示中“1”的个数。如果你的电脑是2010年前购买的，那么很可能不支持SSE4.2指令集，你只需要把return这一行改为 return n+1; 即可，同样能够说明问题。

调用_mm_popcnt_u32需要包含Intel 指令集头文件nmmintrin.h，它虽然不是标准库的一部分，但是所有主流编译器都支持。

中间还有一坨#ifdef...#else...#endif，这个是给MSVC准备的。因为在MSVC下，动态链接库导出的函数必须加 __declspec(dllexport) 进行修饰。而gcc（Linux和Mac OS X的默认编译器）下，所有函数默认均导出。

接下来把它编译为动态链接库。Windows下动态链接库的扩展名是dll，Linux下是so，Mac OS X下是dylib。这里为了方便起见，一律将扩展名设定为dll。

Windows MSVC 下编译命令：（启动Visual Studio命令提示）

```
cl /LD great_module.c /o great_module.dll
```

Windows GCC、Linux、Mac OS X下编译命令相同：

```
gcc -fPIC -shared -msse4.2 great_module.c -o great_module.dll
```

写一个Python程序测试它，这个Python程序是跨平台的：

```
from ctypes import *
great_module = cdll.LoadLibrary('./great_module.dll')
print great_module.great_function(13)
```

整数13是二进制的1101，所以应该输出3

3. 类型映射：基本类型

对于数字和字符串等基本类型。ctypes 采用”中间类型“的方式在Python和C之间搭建桥梁。**对于C类型Tc，均有ctypes类型Tm，将其转换为Python类型Tp。**具体地说，例如某动态链接库中的函数要求参数具有C类型Tc，那么在Python ctypes 调用它的时候，就给予对应的ctypes类型Tm。Tm的值可以通过构造函数的方式传递对应的Python类型Tp。或者，使用它的可修改成员Tm.value。

Tm（ctypes type）、Tc（C type）、Tp （Python type） 之对应关系见下表。

ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character string
c_wchar	wchar_t	1-character unicode string
c_byte	char	int/long
c_ubyte	unsigned char	int/long
c_short	short	int/long
c_ushort	unsigned short	int/long
c_int	int	int/long
c_uint	unsigned int	int/long
c_long	long	int/long
c_ulong	unsigned long	int/long
c_longlong	__int64 Or long long	int/long
c_ulonglong	unsigned __int64 Or unsigned long long	int/long
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	string or None
c_wchar_p	wchar_t * (NUL terminated)	unicode or None
c_void_p	void *	int/long or None

上面一段话比较绕。下面举个例子。

大家熟知的printf函数位于C标准库中。在C代码中调用printf是标准化的，但是，C标准库的**实现**不是标准化的。在Windows中，printf 函数位于%SystemRoot%\System32\msvcrt.dll，在Mac OS X中，它位于 /usr/lib/libc.dylib，在Linux中，一般位于 /usr/lib/libc.so.6。

下面一段代码可以在三大平台上运行：

```
from ctypes import *
from platform import *

cdll_names = {
            'Darwin' : 'libc.dylib',
            'Linux'  : 'libc.so.6',
            'Windows': 'msvcrt.dll'
        }

clib = cdll.LoadLibrary(cdll_names[system()])
clib.printf(c_char_p('Hello %d %f'),c_int(15),c_double(2.3))
```

我们只关注最后一行。printf的原型是

```
int printf (const char * format,...)
```

所以，第一个参数我们用c_char_p创建一个C字符串，并以构造函数的方式用一个Python字符串初始化它。其后，我们给予printf一个int型和一个double型的变量，相应的，我们用c_int和c_double创建对应的C类型变量，并以构造函数的方式初始化它们。

如果不用构造函数，还可以用value成员。以下代码与 clib.printf(c_char_p('Hello %d %f'),c_int(15),c_double(2.3)) 等价：

```
str_format = c_char_p()
int_val = c_int()
double_val = c_double()

str_format.value = 'Hello %d %f'
int_val.value = 15
double_val.value = 2.3
clib.printf(str_format,int_val,double_val)
```

一些C库函数接受指针并修改指针所指向的值。这种情况下相当于数据从C函数流回Python。仍然使用value成员获取值。

```
from ctypes import *
from platform import *

cdll_names = {
            'Darwin' : 'libc.dylib',
            'Linux'  : 'libc.so.6',
            'Windows': 'msvcrt.dll'
        }

clib = cdll.LoadLibrary(cdll_names[system()])
s1 = c_char_p('a')
s2 = c_char_p('b')
s3 = clib.strcat(s1,s2)
print s1.value #ab
```

最后，当 ctypes 可以判断类型对应关系的时候，可以直接将Python类型赋予C函数。ctypes 会进行**隐式类型转换**。例如：

```
s1 = c_char_p('a')
s3 = clib.strcat(s1,'b') # 等价于 s3 = clib.strcat(s1,c_char_p('b'))
print s1.value #ab
```

但是，当 ctypes 无法确定类型对应的时候，会触发异常。

```
clib.printf(c_char_p('Hello %d %f'),15,2.3)
```

异常：

```
Traceback (most recent call last):
  File 'test_printf.py', line 12, in <module>
    clib.printf(c_char_p('Hello %d %f'),15,2.3)
ctypes.ArgumentError: argument 3: <type 'exceptions.TypeError'>: Don't know how to convert parameter 3
```

4. 高级类型映射：数组

在C语言中，char 是一种类型，char [100]是另外一种类型。ctypes 也是一样。使用数组需要预先生成需要的数组类型。

为了方便我们用great_module，增加一个函数 array_get

```
//great_module.c
#ifdef _MSC_VER
    #define DLL_EXPORT __declspec( dllexport )
#else
    #define DLL_EXPORT
#endif
DLL_EXPORT int array_get(int a[], int index) {
    return a[index];
}
```

下面我们在Python里产生数组类型。ctypes 类型重载了操作符 *，因此产生数组类型很容易：

```
from ctypes import *
great_module = cdll.LoadLibrary('./great_module.dll')

type_int_array_10 = c_int * 10

my_array = type_int_array_10()
my_array[2] = c_int(5)
print great_module.array_get(my_array,2)
```

type_int_array_10 即为创建的**数组类型**，如果想得到**数组变量**，则需要**例化**这个类型，即my_array。my_array的每一个成员的类型应该是 c_int，这里将它索引为2的成员赋予值 c_int(5)。当然由于隐式转换的存在，这里写 my_array[2] = 5也完全没有问题。

至于函数返回值的类型，ctypes 规定，总是假设返回值为int。对于array_get而言，碰巧函数返回值也是int，所以具体的数值能被正确的取到。

如果动态链接库中的C函数返回值不是int，需要在调用函数之前显式的告诉ctypes返回值的类型。例如：

```
from ctypes import *
from platform import *

cdll_names = {
            'Darwin' : 'libc.dylib',
            'Linux'  : 'libc.so.6',
            'Windows': 'msvcrt.dll'
        }

clib = cdll.LoadLibrary(cdll_names[system()])
s3 = clib.strcat('a','b')
print s3 # an int value like 5444948
clib.strcat.restype = c_char_p
s4 = clib.strcat('c','d')
print s4 # cd
```

定义一个“高维数组”的方法类似。之所以加了引号，是因为C语言里并没有真正的高维数组，ctype也一样——都是利用数组的数组实现的。

```
from ctypes import *
type_int_array_10 = c_int * 10
type_int_array_10_10 = type_int_array_10 * 10
my_array = type_int_array_10_10()
my_array[1][2] = 3
```

5. 高级类型映射：简单类型指针

ctypes 和C一样区分指针类型和指针变量。复习这两个概念：C语言里，int *是指针类型。用它声明的变量就叫指针变量。指针变量可以被赋予某个变量的地址。

在ctypes中，指针类型用 POINTER(ctypes_type) 创建。例如创建一个类似于C语言的int *：

```
type_p_int = POINTER(c_int)
v = c_int(4)
p_int = type_p_int(v)
print p_int[0]
print p_int.contents
```

其中，type_p_int是一个类型，这个类型是指向int的指针类型。只有将指针类型例化之后才能得到指针变量。在例化为指针变量的同时将其指向变量v。这段代码在C语言里相当于

```
typedef int * type_p_int;
int v = 4;
type_p_int p = &v;
printf('%d',p[0]);
printf('%d',*p);
```

当然，由于Python是依靠绑定传递类型的语言，可以直接使用 ctypes 提供的pointer()得到一个变量的指针变量

```
from ctypes import *

type_p_int = POINTER(c_int)
v = c_int(4)
p_int = type_p_int(v)
print type(p_int)
print p_int[0]
print p_int.contents
#-------
p_int = pointer(v)
print type(p_int)
print p_int[0]
print p_int.contents
```

'#-------' 之前和之后输出的内容是一样的。

6. 高级类型映射：函数指针

函数指针并没有什么特别之处。如果一个动态链接库里的某个C函数需要函数指针，那么可以遵循以下的步骤将一个Python函数包装成函数指针：

1. 查看文档，将C函数指针的原型利用ctypes的CFUNCTYPE包装成ctypes函数指针类型。
2. 利用刚才得到的函数指针类型之构造函数，赋予其Python函数名，即得到函数指针变量。

我们这里举两个例子。

第一个例子来源于ctypes官方文档。它调用的是C标准库中的qsort函数。

我们先观察qsort的文档：

qsort - C++ Reference

它的函数原型是

```
void qsort (void* base, size_t num, size_t size,
            int (*compar)(const void*,const void*));
```

第三个参数即为函数指针作为回调函数，用于给出元素之间大小的判断方法。我们这里使用整数作为判断类型。那么qsort的函数原型可以理解为：

```
void qsort (int* base, size_t num, size_t size,
            int (*compar)(const int*,const int*));
```

其中，回调函数的原型为：

```
int compar(const int*,const int*)
```

使用CFUNCTYPE创建ctypes的函数指针类型：

```
CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
```

CFUNCTYPE的第一个参数是函数的返回值，函数的其他参数紧随其后。

接下来用Python写回调函数的实现：

```
def py_cmp_func(a, b):
    print type(a)
    print 'py_cmp_func', a[0], b[0]
    return a[0] - b[0]
```

最后，用刚才得到的函数指针类型CMPFUNC，以Python回调函数的函数名作为构造函数的参数，就得到了可以用于C函数的函数指针变量：

```
p_c_cmp_func = CMPFUNC(py_cmp_func)
```

完整的代码如下：

```
from ctypes import *
from platform import *

cdll_names = {
            'Darwin' : 'libc.dylib',
            'Linux'  : 'libc.so.6',
            'Windows': 'msvcrt.dll'
        }

clib = cdll.LoadLibrary(cdll_names[system()])

CMPFUNC = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))

def py_cmp_func(a, b):
    print type(a)
    print 'py_cmp_func', a[0], b[0]
    return a[0] - b[0]

type_array_5 = c_int * 5
ia = type_array_5(5, 1, 7, 33, 99)
clib.qsort(ia, len(ia), sizeof(c_int), CMPFUNC(py_cmp_func))
```

注意到，Python函数得到的参数a和b的类型都是 POINTER(c_int) （显示为<class '__main__.LP_c_int'>），对指针变量解引用的方法是之前提到的[0]或者.contents。我们这里应用了ctypes的隐式类型转换，所以a[0]和b[0]可以当成Python的int类型使用。

有趣的是，这段代码在*nix（Linux、Mac OS X）下调用和在Windows下调用，比较的次数是不一样的。Windows似乎更费事。

第二个例子比较实用，但只能在Windows下运行。

我们要利用的是Windows API EnumWindows枚举系统所有窗口的句柄，再根据窗口的句柄列出各个窗口的标题。

EnumWindows的文档见[EnumWindows function (Windows)]

调用Windows API有特殊之处。由于Windows API函数不使用标准C的调用约定（微软一贯的尿性）。故在LoadLibrary时不能够使用cdll.LoadLibrary而使用windll.LoadLibrary。在声明函数指针类型的时候，也不能用CFUNCTYPE而是用WINFUNCTYPE。关于调用约定的问题参见x86 calling conventions

Windows API有很多内建类型，ctypes也对应地提供了支持。代码如下：

```
from ctypes import *
from ctypes import wintypes

WNDENUMPROC = WINFUNCTYPE(wintypes.BOOL,
                          wintypes.HWND,
                          wintypes.LPARAM)
user32 = windll.LoadLibrary('user32.dll')

def EnumWindowsProc(hwnd, lParam):
    length = user32.GetWindowTextLengthW(hwnd) + 1
    buffer = create_unicode_buffer(length)
    user32.GetWindowTextW(hwnd, buffer, length)
    print buffer.value
    return True

user32.EnumWindows(WNDENUMPROC(EnumWindowsProc), 0)
```

7. 其他

ctypes 还对C语言中的结构体、联合体等提供支持。这部分代码比较繁琐，可参见ctypes的文档 https://docs.python.org/2/library/ctypes.html