Python闭包详解

1 快速预览

以下是一段简单的闭包代码示例：

```
def foo():
    m=3
    n=5
    def bar():
        a=4
        return m+n+a
    return bar

>>>bar =  foo()
>>>bar()
12
```

说明：
bar在foo函数的代码块中定义。我们称bar是foo的内部函数。

在bar的局部作用域中可以直接访问foo局部作用域中定义的m、n变量。
简单的说，这种内部函数可以使用外部函数变量的行为，就叫闭包。

那么闭包内部是如何来实现的呢？
我们一步步来，先看两个python内置的object: <code>和<cell>

2 code object

code object是python代码经过编译后的对象。
它用来存储一些与代码有关的信息以及bytecode。

以下代码示例，演示了如何通过编译产生code object

以及使用exec运行该代码，和使用dis方便地查看字节码。

```
import dis
code_obj = compile('sum([1,2,3])',  '', 'single')

>>>exec(code_obj)
6

>>> dis.dis(code_obj)
 1           0 LOAD_NAME                0 (sum)
             3 LOAD_CONST               0 (1)
             6 LOAD_CONST               1 (2)
             9 LOAD_CONST               2 (3)
            12 BUILD_LIST               3
            15 CALL_FUNCTION            1
            18 PRINT_EXPR
            19 LOAD_CONST               3 (None)
            22 RETURN_VALUE
```

那么，这跟我们的例子有什么关系？

```
>>> foo.func_code
<code object foo at 01FE92F0, file "<pyshell#50>",
line 1>
```

我们可以看到，函数定义好之后，就可以通过[函数名.func_code]
访问该函数的code object，之后我们会用到它的一些特性。

3 cell object

cell对象的引入，是为了实现被多个作用域引用的变量。
对每一个这样的变量，都用一个cell对象来保存 **其值** 。

拿之前的示例来说，m和n既在foo函数的作用域中被引用，又在bar
函数的作用域中被引用，所以m, n引用的值，都会在一个cell对象中。

可以通过内部函数的__closure__或者func_closure特性查看cell对象：

```
>>> bar = foo()
>>> bar.__closure__
(<cell at 0x01FE8DF0: int object at 0x0186D888>,
<cell at 0x01F694B0: int object at 0x0186D870>)
```

这两个int型的cell分别存储了m和n的值。
无论是在外部函数中定义，还是在内部函数中调用，引用的指向都是cell对象中的值。

注：内部函数无法修改cell对象中的值，如果尝试修改m的值，编译器会认为m是函数bar的局部变量，同时foo代码块中的m也会被认为是函数foo的局部变量，两个m分别

在各自的作用域下起作用。[1]

4 闭包分析

- 使用dis[2]模块分析foo的bytecode。

```
2           0 LOAD_CONST               1 (3)
            3 STORE_DEREF              0 (m)

3           6 LOAD_CONST               2 (5)
            9 STORE_DEREF              1 (n)

4          12 LOAD_CLOSURE             0 (m)
           15 LOAD_CLOSURE             1 (n)
           18 BUILD_TUPLE              2
           21 LOAD_CONST               3 (<code object bar
at 018D9848, file "<pyshell#1>", line 4>)
           24 MAKE_CLOSURE             0
           27 STORE_FAST               0 (bar)

7          30 LOAD_FAST                0 (bar)
           33 RETURN_VALUE
```

进行逐行分析：

LOAD_CONST 1 (3) :
将foo.func_code.co_consts [1] 的值"3" push进栈。

STORE_DEREF 0 (m) :
从栈顶Pop出"3"包装成cell对象存入cell与自由变量的存储区的第0槽。
将cell对象的地址信息赋给变量m(闭包变量名记录在func_code.cellvars)。
func_code.cellvars的内容为('m', 'n')

LOAD_CLOSURE 0 (m) :
将变量m的值push进栈，类似如下信息：
<cell at 0x01D572B0: int object at 0x0180D6F8>

LOAD_CLOSURE 1 (n) :
类似变量m的处理，不在累述。

当前栈区状态：

1	<cell at 0x01D572B0: int object at 0x0180D6F8>
2	<cell at 0x01D86510: int object at 0x0180D6E0>
3	…

BUILD_TUPLE 2 :
将栈顶的两项取出，创建元组，并将该元组push进栈。

LOAD_CONST 3 :
从foo.func_code.co_consts [3] 取出，该项为内部函数bar的code object的地址，将其push进栈
<code object bar at 018D9848, file "<pyshell#1>", line 4>

栈区状态：

1	<code object bar at 018D9848, file "<pyshell#1>", line 4>
2	(<cell at 0x01D572B0: int object at 0x0180D6F8>, <cell at 0x01D86510: int object at 0x0180D6E0>)
3	…

MAKE_CLOSURE 0 :

创建一个函数对象，pop出栈顶的code object(bar函数的code)地址信息赋给foo的func_code特性；

pop出包含cell对象地址的元组，赋给foo的func_closure特性；
最后将该函数对象地址信息push进栈。

STORE_FAST 0 (bar) :
从栈顶取出之前创建的函数对象的地址信息赋给局部变量bar(局部变量名记录在func_code.co_varnames中)
func_code.co_varnames的内容为('bar',)
将变量bar(记录在func_code.cellvars [0])绑定栈顶的函数对象地址。

LOAD_FAST 0 (bar) :
将变量bar的值压入栈。

RETURN_VALUE
返回栈顶项，print bar可以看到<function bar at 0x01D899F0>

- 再分析bar函数就简单了

```
5        0 LOAD_CONST          1 (4)
         3 STORE_FAST          0 (a)

6        6 LOAD_DEREF          0 (m)
         9 LOAD_DEREF          1 (n)
        12 BINARY_ADD
        13 LOAD_FAST           0 (a)
        16 BINARY_ADD
        17 RETURN_VALUE
```

重点是LOAD_DEREF，该方法主要是将cell对象中的object内容push进栈。大致过程如下：

根据变量m的值找到包装在cell内的int object的地址信息
m的值：<cell at 0x01D572B0: int object at 0x0180D6F8>

根据地址取出int值，push进栈。

5 参考文章

- Closures in Python - ynniv
- Python Closures Explained - Praveen Gollakota
- dis.py -Terry Jan Reedy

Footnotes:

[1] 看完通篇，使用dis分析一下这种情况的bytecode，就能得出这样的结论。

[2] 函数经过编译的bytecode，实际上放在func.func_code.co_code中，dis模块对其做了解析，使其更容易阅读。

标签： python， closure， 闭包

posted @ 2013-07-23 11:14 ChrisChen3121 阅读(4242) 评论(2)