有趣的Python闭包(Closures)

Python

写下这篇博客，起源于Tornado邮件群组的这个问题 how to use outer variable in inner method，这里面老外的回答很有参考价值，关键点基本都说到了。我在这里用一些有趣的例子来做些解析，简要的阐述下Python的闭包规则，首先看一个经典的例子:

```
def foo():
    a = 1
    def bar():
        a = a + 1
        # print a + 1
        # b = a + 1
        # a = 1
        print id(a)

    bar()
    print a, id(a)
```

在Python2.x上运行这个函数会报UnboundLocalError: local variable 'a' referenced before assignment即本地变量在引用前未定义，如何来理解这个错误呢？PEP 227里面介绍到，Python解析器在搜索一个变量的定义时是根据如下三级规则来查找的：

> The Python 2.0 definition specifies exactly three namespaces to check for each name — the local namespace, the global namespace, and the builtin namespace.

这里的local实际上可能还有多级，上面的代码就是一个例子，下面通过对代码做些简单的修改来一步步理解这里面的规律：

- 如果将a = a + 1这句换成print a + 1或者b = a + 1，是不会有问题的，即在内部函数bar内，外部函数foo里的a实际是可见的，可以引用。
- 将a = a + 1换成 a = 1也是没有问题的，但是如果你将两处出现的a的id打印出来你会发现，其实这两个a不是一回事，在内部函数bar里面，本地的a = 1定义了在bar函数范围内的新的一个局部变量，因为名字和外部函数foo里面的变量a名字相同，导致外部函数foo里的a在内部函数bar里实际已不可见。
- 再来说a = a + 1出错是怎么回事，首先a = xxx这种形式，Python解析器认为要在内部函数bar内创建一个新的局部变量a，同时外部函数foo里的a在bar里已不可见，而解析器对接下来对右边的a + 1的解析就是用本地的变量a加1，而这时左边的a即本地的变量a还没有创建(等右边赋值呢)，因此就这就产生了一个是鸡生蛋还是蛋生鸡的问题，导致了上面说的UnboundLocalError的错误。

要解决这个问题，在Python2.x里主要有两个方案：

1. 用别名替代比如b = a + 1，内部函数bar内只引用外部函数foo里的a。
2. 将foo里的a设成一个容器，如list

```
def foo():
    a = [1, ]
    def bar():
        a[0] = a[0] + 1

    bar()
    print a[0]
```

当然这有些时候还是很不方便，因此在Python3.x中引入了一个nonloacal的关键字来解决这个问题，只要在a = a + 1前加一句nonloacal a即可，即显式的指定a不是内部函数bar内的本地变量，这样就可以在bar内正常的使用和再赋值外部函数foo内的变量a了。

在搜索Python闭包相关的材料中，我在StackOverflow上发现一个有趣的有关Python闭包的问题，有兴趣的可以思考思考做做看，结果应该是什么？你预期的结果是什么，若不一致，如果要得到你预期的结果应该怎么改？

```
flist = []

for i in xrange(3):
    def func(x): return x * i
    flist.append(func)

for f in flist:
    print f(2)
```

扩展阅读：

1. PEP 227 — Statically Nested Scopes
2. PEP 3104 — Access to Names in Outer Scopes
3. Lexical closures in Python

转载请注明出处：http://feilong.me/2012/06/interesting-python-closures

由飞龙非龙 发表于 下午 1:23　　添加标签：Closures, nonlocal, 闭包