Closure And Python Scoping Rule

Release 1.0

Jerry Peng

December 16, 2011

Contents

1 Introduction

Closure（闭包）是函数式编程（FP）的核心概念之一，然而在很长一段时间之内我都以为它是指匿名函数，直到最近看《Practical Common Lisp》，才发现之前对它的理解是错误的。随后在熟悉的语言中对其做了一些探索，发现了一些之前不曾了解的细节，在此记录一下。主要介绍以下内容：

- 什么是 Closure
- Python 支持 Closure 吗
- Python 的作用域规则

2 什么是 Closure?

为了避免个人的理解偏差带来影响，先引用 wikipedia 词条上的描述:

> In computer science, a closure (also lexical closure, function closure, function value or functional value) is a function together with a referencing environment for the non-local variables of that function. A closure allows a function to access variables outside its typical scope. Such a function is said to be "closed over" its free variables. The referencing environment binds the nonlocal names to the corresponding variables in

> scope at the time the closure is created, additionally extending their lifetime to at least as long as the lifetime of the closure itself. When the closure is entered at a later time, possibly from a different scope, the function is executed with its non-local variables referring to the ones captured by the closure.

Closure 并不是指匿名函数，而是指嵌套的，绑定了自身引用的作用域的函数（是否匿名没关系）。Closure 的存在需要以下两个条件：

- 函数是一类对象，可以当作数据来传递
- 嵌套的函数可以访问其父作用域中的数据

不借助例子，这个概念实在是太过难以理解，下面以一小段 Groovy 程序为例介绍（为什么不用 Python？因为 Python 的作用域有特殊的规则，无法实现以下程序。为什么不用 Lisp？那货可读性太差，担心影响效果）

```
class ClosureTest {

    static def createCounter(initialVal) {
        def val = initialVal
        return { ->
            val = val + 1
            return val
        }
    }

    static main(args) {
        def counterFunc = createCounter(10)
        println "First invocation: " + counterFunc()
        println "Second invocation: " + counterFunc()
        println "Third invocation: " + counterFunc()
    }
}
```

createCounter 是一个高阶函数，它返回一个函数，函数的作用是从给定的初值开始做累加——调用一次，累加一次，并返回结果。此程序返回以下结果:

```
First invocation: 11
Second invocation: 12
Third invocation: 13
```

每次调用 createCounter，它都会创建一个 Closure，将匿名函数（return 后面大括号里的内容）与 createCounter 的局部作用域绑定起来。本来 createCounter 调用结束的时候，其局部作用域应该是失效了的，但 Closure 将此作用域的生命周期延长到和函数对象一样。

这个绑定关系并不是简单地绑定变量的值，而是整个作用域，因此 Closure 还可以改变其绑定的作用域中某个变量的值，而不仅仅是读取而已。上面的例子中，每一次匿名函数被调用，val 的值都会改变，正因为绑定的是整个作用域，所以其值得以保存。

所以 Closure 并不是指匿名函数，而是指函数与其作用域绑定后的对象。两次调用 createCounter 会创建两个不同的 Closure，绑定了两个作用域，互不影响。

```
static main(args) {
    def counterFunc1 = createCounter(10)
    def counterFunc2 = createCounter(0);
    println "Counter 1 First invocation: " + counterFunc1()
    println "Counter 2 First invocation: " + counterFunc2()
    println "Counter 1 Second invocation: " + counterFunc1()
    println "Counter 2 Second invocation: " + counterFunc2()
```

```
    println "Counter 1 Third invocation: " + counterFunc1()
    println "Counter 2 Third invocation: " + counterFunc2()
}
```

运行结果:

```
Counter 1 First invocation: 11
Counter 2 First invocation: 1
Counter 1 Second invocation: 12
Counter 2 Second invocation: 2
Counter 1 Third invocation: 13
Counter 2 Third invocation: 3
```

3 Python 支持 Closure 吗?

我试图用 Python 写过以上的例子，代码如下:

```
def create_counter(initval):
    val = initval
    def _inner_counter():
        val = val + 1
        return val
    return _inner_counter

if __name__ == '__main__':
    counter = create_counter(0)
    print "First invocation: ", counter()
    print "Second invocation: ", counter()
    print "Third invocation: ", counter()
```

用 Python 2.7 运行发现，它可耻地挂鸟:

```
First invocation:
Traceback (most recent call last):
  File "closure.py", line 10, in <module>
    print "First invocation: ", counter()
  File "closure.py", line 4, in _inner_counter
    val = val + 1
UnboundLocalError: local variable 'val' referenced before assignment
```

这是什么情况？难道我们如此热爱的 Python 不支持这个帅呆了的 feature 吗？

不是的，Python 依然支持闭包，下面的这个曾经让我很困惑的例子可以很好地证明这一点:

```
In [5]: funcs = [lambda : x for x in ['a', 'b', 'c']]

In [6]: funcs[0]()
Out[6]: 'c'

In [7]: funcs[1]()
Out[7]: 'c'

In [8]: funcs[2]()
Out[8]: 'c'
```

funcs 是一个包含三个函数的列表，函数简单地返回一个值。如果 Python 是在创建 lambda 的

时候绑定的是变量的值，那这三个函数必定会依次返回 a, b, c。但事实是，它们返回的都是 c。这说明 lambda 所绑定的是同一个局部作用域，因此其中 x 的值也是迭代完成后的最终值 c。

用下面这个例子来解释或许更加清楚一些：

```
def test():
    func = lambda : 'value of x: %s' % x
    try:
        print func()
    except Exception, e:
        print 'ERROR:', e
    x = 10
    print func()
    x = 'oops'
    print func()
test()
```

结果:

```
ERROR: global name 'x' is not defined
value of x: 10
value of x: oops
```

func 创建的时候，x 还未定义，所以第一次 func 调用会报错。之后 x 初始化成 10，所以第二次调用会返回'value of x: 10'，之后将 x 的值改为'oops' 了以后再次调用，其返回值也反映了 x 的最新值。

由以上两个例子可以看出来，Python 是支持 Closure 的，其行为符合 Closure 的定义：close over 一个局部作用域到一个函数中。那第一个例子为什么无法工作呢？这就需要解释一下 Python 的作用域规则。

4 Python 的作用域规则

可以用 LEGB 来总结 Python 的作用域规则：当一个变量被访问的时候，Python 会按 LEGB 的顺序来搜索变量：

> 要说明的是，这里的访问规则只对普通变量有效，对象属性的规则与这无关（简单地说，访问一个对象的属性与此无关）。

Local 局部作用域，即函数中定义的变量（没有用 global 声明）

Enclosing 嵌套的父级函数的局部作用域，即包含此函数的上级函数的局部作用域，比如上面的示例中的 labmda 所访问的 x 就在其父级函数 test 的局部作用域里。通常也叫 non-local 作用域。

Global(module) 在模块级别定义的全局变量（如果需要在函数内修改它，需要用 global 声明）

Built-in built-in 模块里面的变量，比如 int, Exception 等等

但此规则有一个重要的限制:

> 一个不在局部作用域里的变量默认是只读的，如果试图为其绑定一个新的值，Python 认为是在当前的局部作用域里创建一个新的变量。

下面是个例子：

```
def outer_func():
    x = 3
    def inner_func1():
        print 'inner func 1:', x
    def inner_func2():
        x = 'hello'
        print 'inner func 2:', x
    inner_func1()
    inner_func2()
    print 'outer func:', x
```

输出:

```
inner func 1: 3
inner func 2: hello
outer func: 3
```

inner_func1 中对 x 的访问是只读的，Python 会在父级作用域中搜寻 x，结果在 outer_func 的局部作用域中发现了它，所以 inner_func1 会打印 3。inner_func2 中试图对 x 绑定新的值，Python 解释器认为这是在创建一个新的局部变量 x，其值为'hello'，于是 inner_func2 会打印出'hello'，但这对 outer_func 中的 x 无影响（因为在不同的作用域里），所以最后 outer_func 中打印的还是 3。

这就解释了为什么计数器的例子无法在 Python 上运行了：_inner_counter 里的 var = var + 1 让 Python 认为 var 是一个局部变量，而非外层函数中的 var，而这条赋值语句还试图读取 var 的旧值，所以会报‘赋值之前引用’的错误。

如果确实要在一个函数里修改全局变量，Python 提供了 global 关键字来声明一个变量是全局变量，声明以后就可以修改其值了。然而 global 只能用来修改全局作用域里的变量，对于嵌套函数的情况无能为力，所以计数器的例子在 Python 2.x 中是无法实现的。然而在 Python 3 中，一个新的关键字 nonlocal 的产生解决了这个问题。我们可以用 Python 3 来改写第一个例子：

```
def create_counter(initval):
    val = initval
    def _inner_counter():
        nonlocal val
        val = val + 1
        return val
    return _inner_counter

if __name__ == '__main__':
    counter = create_counter(10)
    print("First invocation: ", counter())
    print("Second invocation: ", counter())
    print("Third invocation: ", counter())
```

运行一下看看，结果果然在预料之中！如下:

```
[~/dev/personal/python/practice]$ python3.2 closure3.py
First invocation:  11
Second invocation:  12
Third invocation:  13
```

吐槽时间: Python 的 lambda 无法支持多条语句，这个不爽啊不爽，导致有时候不得不写一个命名的内部函数。

5 参考资料

- Practical Common Lisp 中对 closure 的解释：http://www.gigamonkeys.com/book/variables.html
- Stack Overflow 上的一个对 Python 作用域的解释：http://stackoverflow.com/questions/291978/short-description-of-python-scoping-rules