Python 中的闭包

之前经常听人提到“闭包”这个词，当时对于我这个 Java 程序员来说概念还是很模糊，当时我尝试找了些资料去看，但还是一知半解，最近一边 工作一边学习 Python，看到 Python 中的闭包，结合《Python Cookbook》中的对闭包的介绍真正领悟了闭包的奥秘。

下面先举个例子：

```
def make_adder(addend):
    def adder(augend):
        return augend + addend
    return adder
```

make_adder 函数里嵌套了一个内层函数 adder，这个内层函数就是一个闭包，其实可以也不用管这个“闭包”的概念，先来看下这种模式解决些什么问题，“闭包”只是个名称罢了。

调用 make_adder 函数：p = make_adder(23)，由于内层的函数 adder 里的逻辑用到了 make_adder 函数的入参，而这时这个入参 addend 绑定了值 23，由于 make_adder 函数返回的是函数 adder，所以这时的 p 其实就是内部的 addend 绑定了 23 的函数 adder；同理可知，q = make_adder(44)，这里的 q 就是内部 addend 绑定了 44 的函数 adder，p 和 q 这两个 adder 函数是不相同的，因为内部 addend 绑定的值不同，只是两个函数的模板相同罢了，这时我们执行 p(100)得到结果是 123，q(100)得到结果是 144。这样做有什么用呢？其实可以这样 看：我们可以把 p = make_adder(23)和 q = make_adder(44)看成是配置过程，23 和 44 是配置信息，p(100)和 q(100)根据不同的配置获得不同的结果，这样我们就可以解决开发 中“根据配置信息不同获得不同结果”的问题

就拿《Python Cookbook》中的例子来阐述

我们需要一个字符串过滤的功能，把一个字符串中我们想留下的字符留下，除此之外的字符都过滤掉，这里的“我们想留下的字符”其实就是上面提到的"配 置信息"，我们根据不同的配置信息获得不同的结果，我们根据我们想留下的字符得到不同的过滤后的字符串，所以这里的过滤功能函数这样写：

```
import string
allchars = string.maketrans('', '')
def makefilter(keep):
    delchars = allchars.translate(allchars, keep)
    def thefilter(s):
```

```
        return s.translate(allchars, delchars)
    return thefilter
```

这里用到了 python 的 translate，不清楚的可以看 http://docs.python.org/

makefilter 函数的参数是我们想留下的字符，可以传一个字符串，相当于字符串里所有的字符都要留下，makefilter 函数里有个内层函 数 thefilter，这个函数会得到我们传入 makefilter 函数的想要留下的字符，根据 delchars 它就知道要过滤掉哪些字符，下面我们来执 行下，看下效果：

```
>>> just_vowels = makefilter('aeiouy')      #我们想留下 aeiouy 这些字符
>>> just_vowels('wwwwwwwaeiouy')        #这里的 just_vowels 其实就是知道了我们想留下的字符 aeiouy 后的 thefilter 函数
最后得到的结果：'aeiouy'
```

以上就是闭包的奥秘，其实就是一个“根据不同配置信息得到不同结果”的功能，而且我们在开发中是经常碰到这样的需求的，用闭包来解决真的非常好，代码简单易懂，而且扩展性也非常好！

一，定义

python 中的闭包从表现形式上定义（解释）为：如果在一个内部函数里，对在外部 作用域（但不是在全局作用域）的变量进行引用，那么内部函数就被认为是闭包(closure).这个定义是相对直白的，好理解的，不像其他定义那样学究味 道十足（那些学究味道重的解释，在对一个名词的解释过程中又充满了一堆让人抓狂的其他陌生名词，不适合初学者）。下面举一个简单的例子来说明。

[python] view plaincopy

```
1.  >>>def addx(x):
2.  >>>     def adder(y): return x + y
3.  >>>     return adder
4.  >>> c =  addx(8)
5.  >>> type(c)
6.  <type 'function'>
7.  >>> c.__name__
8.  'adder'
9.  >>> c(10)
10. 18
```

结合这段简单的代码和定义来说明闭包：
如果在一个内部函数里：adder(y)就是这个内部函数，
对在外部作用域（但不是在全局作用域）的变量进行引用：x 就是被引用的变量，
x 在外部作用域 addx 里面，但不在全局作用域里，

则这个内部函数 adder 就是一个闭包。

再稍微讲究一点的解释是，闭包=函数块+定义函数时的环境，adder 就是函数块，x 就是环境，当然这个环境可以有很多，不止一个简单的 x。

二，使用闭包注意事项

1，闭包中是不能修改外部作用域的局部变量的

[python] view plaincopy

```
>>> def foo():
...     m = 0
...     def foo1():
...         m = 1
...         print m
...
...     print m
...     foo1()
...     print m
...
>>> foo()
0
1
0
```

从执行结果可以看出，虽然在闭包里面也定义了一个变量 m，但是其不会改变外部函数中的局部变量 m。

2，以下这段代码是在 python 中使用闭包时一段经典的错误代码

[python] view plaincopy

```
def foo():
    a = 1
    def bar():
        a = a + 1
        return a
    return bar
```

这段程序的本意是要通过在每次调用闭包函数时都对变量 a 进行递增的操作。但在实际使用时

[html] view plaincopy

```
>>> c = foo()
>>> print c()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in bar
```

6. UnboundLocalError: local variable 'a' referenced before assignment

这 是因为在执行代码 c = foo()时，python 会导入全部的闭包函数体 bar()来分析其的局部变量，python 规则指定所有在赋值语句左面的变量都是局部变量，则在闭包 bar()中，变量 a 在赋值符号"="的左面，被 python 认为是 bar()中的局部变量。再接下来执行 print c()时，程序运行至 a = a + 1 时，因为先前已经把 a 归为 bar()中的局部变量，所以 python 会在 bar()中去找在赋值语句右面的 a 的值，结果找不到，就会报错。解决的方法很 简单

[python] view plaincopy

```
1. def foo():
2.     a = [1]
3.     def bar():
4.         a[0] = a[0] + 1
5.         return a[0]
6.     return bar
```

只要将 a 设定为一个容器就可以了。这样使用起来多少有点不爽，所以在 python3 以后，在 a = a + 1 之前，使用语句 nonloacal a 就可以了，该语句显式的指定 a 不是闭包的局部变量。

3，还有一个容易产生错误的事例也经常被人在介绍 python 闭包时提起，我一直都没觉得这个错误和闭包有什么太大的关系，但是它倒是的确是在 python 函数式编程是容易犯的一个错误，我在这里也不妨介绍一下。先看下面这段代码

[python] view plaincopy

```
1. for i in range(3):
2.     print i
```

在程序里面经常会出现这类的循环语句，Python 的问题就在于，当循环结束以后，循环体中的临时变量 i 不会销毁，而是继续存在于执行环境中。还有一个 python 的现象是，python 的函数只有在执行时，才会去找函数体里的变量的值。

[python] view plaincopy

```
1. flist = []
2. for i in range(3):
3.     def foo(x): print x + i
4.     flist.append(foo)
5. for f in flist:
6.     f(2)
```

可 能有些人认为这段代码的执行结果应该是 2,3,4.但是实际的结果是 4,4,4。这是因为当把函数加入 flist 列表里时，python 还没有给 i 赋值， 只有当执行时，

再去找 i 的值是什么，这时在第一个 for 循环结束以后，i 的值是 2，所以以上代码的执行结果是 4,4,4.
解决方法也很简单，改写一下函数的定义就可以了。
[python] view plaincopy

```
1. for i in range(3):
2.     def foo(x,y=i): print x + y
3.     flist.append(foo)
```

三，作用
说了这么多，不免有人要问，那这个闭包在实际的开发中有什么用呢？闭包主要是在函数式开发过程中使用。以下介绍两种闭包主要的用途。

用途 1，当闭包执行完后，仍然能够保持住当前的运行环境。
比如说，如果你希望函数的每次执行结果，都是基于这个函数上次的运行结果。我以一个 类似棋盘游戏的例子来说明。假设棋盘大小为 50*50，左上角为坐标系原点(0,0)，我需要一个函数，接收 2 个参数，分别为方向 (direction)，步长 (step)，该函数控制棋子的运动。棋子运动的新的坐标除了依赖于方向和步长以外，当然还要根据原来所处的坐标点，用闭包 就可以保持住这个棋子原来所处的坐标。
[python] view plaincopy

```
1. origin = [0, 0]  # 坐标系统原点
2. legal_x = [0, 50]  # x 轴方向的合法坐标
3. legal_y = [0, 50]  # y 轴方向的合法坐标
4. def create(pos=origin):
5.     def player(direction,step):
6.         # 这里应该首先判断参数 direction,step 的合法性，比如 direction 不能斜着走，step 不能为负等
7.         # 然后还要对新生成的 x，y 坐标的合法性进行判断处理，这里主要是想介绍闭包，就不详细写了。
8.         new_x = pos[0] + direction[0]*step
9.         new_y = pos[1] + direction[1]*step
10.        pos[0] = new_x
11.        pos[1] = new_y
12.        #注意！此处不能写成 pos = [new_x, new_y]，原因在上文有说过
13.        return pos
14.    return player
15.
16. player = create()  # 创建棋子 player，起点为原点
17. print player([1,0],10)  # 向 x 轴正方向移动 10 步
18. print player([0,1],20)  # 向 y 轴正方向移动 20 步
```

19. print player([-1,0],10) # 向 x 轴负方向移动 10 步

输出为
[python] view plaincopy

```
1. [10, 0]
2. [10, 20]
3. [0, 20]
```

用途 2，闭包可以根据外部作用域的局部变量来得到不同的结果，这有点像一种类似配置功能的作用，我们可以修改外部的变量，闭包根据这个变量展现出不同的功能。比如有时我们需要对某些文件的特殊行进行分析，先要提取出这些特殊行。
[python] view plaincopy

```
1. def make_filter(keep):
2.     def the_filter(file_name):
3.         file = open(file_name)
4.         lines = file.readlines()
5.         file.close()
6.         filter_doc = [i for i in lines if keep in i]
7.         return filter_doc
8.     return the_filter
```

如果我们需要取得文件"result.txt"中含有"pass"关键字的行，则可以这样使用例子程序
[python] view plaincopy

```
1. filter = make_filter("pass")
2. filter_result = filter("result.txt")
```

以上两种使用场景，用面向对象也是可以很简单的实现的，但是在用 Python 进行函数式编程时，闭包对数据的持久化以及按配置产生不同的功能，是很有帮助的。