Python中的闭包 4820°C

作者：the5fire | 标签： python (/tag/python) 闭包 (/tag/闭包) | 发布：2014-08-12 11:20 p.m.

Python中的闭包

前几天又有人在我的这篇文章 python项目练习一：即时标记 (http://www.the5fire.com/python-practice-1.html#comments) 下留言，关于其中一个闭包和re.sub的使用不太清楚。我在自己的博客上搜索了下，发现没有写过闭包相关的东西，所以决定总结一下，完善博客上Python的内容。

1. 闭包的概念

首先还得从基本概念说起，什么是闭包呢？来看下维基上的解释:

> 在计算机科学中，闭包（Closure）是词法闭包（Lexical Closure）的简称，是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。
> ....

上面提到了两个关键的地方： 自由变量 和 函数, 这两个关键稍后再说。还是得在赘述下“闭包”的意思，望文知意，可以形象的把它理解为一个封闭的包裹，这个包裹就是一个函数，当然还有函数内部对应的逻辑，包裹里面的东西就是自由变量，自由变量可以在随着包裹到处游荡。当然还得有个前提，这个包裹是被创建出来的。

在通过Python的语言介绍一下，一个闭包就是你调用了一个函数A，这个函数A返回了一个函数B给你。这个返回的函数B就叫做闭包。你在调用函数A的时候传递的参数就是自由变量。

举个例子：

```
def func(name):
    def inner_func(age):
        print 'name:', name, 'age:', age
    return inner_func

bb = func('the5fire')
bb(26)  # >>> name: the5fire age: 26
```

这里面调用func的时候就产生了一个闭包——inner_func,并且该闭包持有自由变量——name，因此这也意味着，当函数func的生命周期结束之后，name这个变量依然存在，因为它被闭包引用了，所以不会被回收。

另外再说一点，闭包并不是Python中特有的概念，所有把函数做为一等公民的语言均有闭包的概念。不过像Java这样以class为一等公民的语言中也可以使用闭包，只是它得用类或接口来实现。

更多概念上的东西可以参考最后的参考链接。

2. 为什么使用闭包

基于上面的介绍，不知道读者有没有感觉这个东西和类有点相似，相似点在于他们都提供了对数据的封装。不同的是闭包本身就是个方法。和类一样，我们在编程时经常会把通用的东西抽象成类，（当然，还有对现实世界——业务的建模），以复用通用的功能。闭包也是一样，当我们需要函数粒度的抽象时，闭包就是一个很好的选择。

在这点上闭包可以被理解为一个只读的对象，你可以给他传递一个属性，但它只能提供给你一个执行的接口。因此在程序中我们经常需要这样的一个函数对象——闭包，来帮我们完成一个通用的功能，比如后面会提到的——装饰器。

3. 使用闭包

第一种场景，在python中很重要也很常见的一个使用场景就是装饰器，Python为装饰器提供了一个很友好的“语法糖”——@，让我们可以很方便的使用装饰器，装饰的原理不做过多阐述，简言之你在一个函数func上加上@decorator_func, 就相当于decorator_func(func):

```
def decorator_func(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper

@decorator_func
def func(name):
    print 'my name is', name

# 等价于
decorator_func(func)
```

在装饰器的这个例子中，闭包（wrapper）持有了外部的func这个参数，并且能够接受外部传过来的参数，接受过来的参数在原封不动的传给func，并返回执行结果。

这是个简单的例子，稍微复杂点可以有多个闭包，比如经常使用的那个LRUCache的装饰器，装饰器上可以接受参数@lru_cache(expire=500)这样。实现起来就是两个闭包的嵌套:

```
def lru_cache(expire=5):
    # 默认5s超时
    def func_wrapper(func):
        def inner(*args, **kwargs):
            # cache 处理 bala bala bala
            return func(*args, **kwargs)
        return inner
    return func_wrapper

@lru_cache(expire=10*60)
def get(request, pk)
    # 省略具体代码
    return response()
```

不太懂闭包的同学一定得能够理解上述代码，这是我们之前面试经常会问到的面试题。

第二个场景，就是基于闭包的一个特性——“惰性求值”。这个应用比较常见的是在数据库访问的时候，比如说：

```
# 伪代码示意

class QuerySet(object):
    def __init__(self, sql):
        self.sql = sql
        self.db = Mysql.connect().corsor()  # 伪代码

    def __call__(self):
        return db.execute(self.sql)

def query(sql):
    return QuerySet(sql)

result = query("select name from user_app")
if time > now:
    print result  # 这时才执行数据库访问
```

上面这个不太恰当的例子展示了通过闭包完成惰性求值的功能，但是上面query返回的结果并不是函数，而是具有函数功能的类。有兴趣的可以去看看Django的queryset的实现，原理类似。

第三种场景， 需要对某个函数的参数提前赋值的情况，当然在Python中已经有了很好的解决访问 functools.parial,但是用闭包也能实现。

```
def partial(**outer_kwargs):
    def wrapper(func):
        def inner(*args, **kwargs):
            for k, v in outer_kwargs.items():
                kwargs[k] = v
            return func(*args, **kwargs)
        return inner
    return wrapper

@partial(age=15)
def say(name=None, age=None):
    print name, age

say(name="the5fire")
# 当然用functools比这个简单多了
# 只需要:  functools.partial(say, age=15)(name='the5fire')
```

看起来这又是一个牵强的例子，不过也算是实践了闭包的应用。

最后总结下，闭包这东西理解起来还是很容易的，在Python中的应用也很广泛，这篇文章算是对闭包的一个总结，有任何疑问欢迎留言交流。

4. 参考资料

- 维基百科-闭包 (http://zh.wikipedia.org/wiki/%E9%97%AD%E5%8C%85_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6))
- http://stackoverflow.com/questions/4020419/closures-in-python (http://stackoverflow.com/questions/4020419/closures-in-python)
- http://www.shutupandship.com/2012/01/python-closures-explained.html (http://www.shutupandship.com/2012/01/python-closures-explained.html)
- http://stackoverflow.com/questions/141642/what-limitations-have-closures-in-python-compared-to-language-x-closures (http://stackoverflow.com/questions/141642/what-limitations-have-closures-in-python-compared-to-language-x-closures)
- http://mrevelle.blogspot.com/2006/10/closure-on-closures.html (http://mrevelle.blogspot.com/2006/10/closure-on-closures.html)