Python 装饰器与面向切面编程

作者：AstralWind | 出处：博客园 |

1. 装饰器入门

1.1. 需求是怎么来的？

装饰器的定义很是抽象，我们来看一个小例子。

```
1 def foo():
2     print 'in foo()'
3
4 foo()
```

这是一个很无聊的函数没错。但是突然有一个更无聊的人，我们称呼他为 B 君，说我想看看执行这个函数用了多长时间，好吧，那么我们可以这样做：

```
1 import time
2 def foo():
3     start = time.clock()
4     print 'in foo()'
5     end = time.clock()
6     print 'used:', end - start
7
8 foo()
```

1.2. 以不变应万变，是变也

还记得吗，函数在 Python 中是一等公民，那么我们可以考虑重新定义一个函数 timeit，将 foo 的引用传递给他，然后在 timeit 中调用 foo 并进行计时，这样，我们就达到了不改动 foo 定义的目的，而且，不论 B 君看了多少个函数，我们都不用去修改函数定义了！

```
01 import time
02
03 def foo():
04     print 'in foo()'
05
06 def timeit(func):
```

```
07     start = time.clock()
08     func()
09     end =time.clock()
10     print 'used:', end - start
11
12 timeit(foo)
```

看起来逻辑上并没有问题，一切都很美好并且运作正常！……等等，我们似乎修改了调用部分的代码。原本我们是这样调用的：foo()，修改以后变成 了：timeit(foo)。这样的话，如果 foo 在 N 处都被调用了，你就不得不去修改这 N 处的代码。或者更极端的，考虑其中某处调用的代码无法修改这个 情况，比如：这个函数是你交给别人使用的。

1.3. 最大限度地少改动！

既然如此，我们就来想想办法不修改调用的代码；如果不修改调用代码，也就意味着调用 foo()需要产生调用 timeit(foo)的效果。我们可以 想到将 timeit 赋值给 foo，但是 timeit 似乎带有一个参数……想办法把参数统一吧！如果 timeit(foo)不是直接产生调用效果，而是返回 一个与 foo 参数列表一致的函数的话……就很好办了，将 timeit(foo)的返回值赋值给 foo，然后，调用 foo()的代码完全不用修改！

```
01 #-*- coding: UTF-8 -*-
02 import time
03
04 def foo():
05     print 'in foo()'
06
07 # 定义一个计时器，传入一个，并返回另一个附加了计时功能的方法
08 def timeit(func):
09
10     # 定义一个内嵌的包装函数，给传入的函数加上计时功能的包装
11     def wrapper():
12         start = time.clock()
13         func()
14         end =time.clock()
15         print 'used:', end - start
16
17     # 将包装后的函数返回
18     return wrapper
19
20 foo = timeit(foo)
```

```
21 foo()
```

这样，一个简易的计时器就做好了！我们只需要在定义 foo 以后调用 foo 之前，加上 foo = timeit(foo)，就可以达到计时的目的，这也就是装饰器的概念，看起来像是 foo 被 timeit 装饰了。在在这个例子中，函数进入和退出时需要计 时，这被称为一个横切面(Aspect)，这种编程方式被称为面向切面的编程(Aspect-Oriented Programming)。与传统编程习惯的从上往下执行方式相比较而言，像是在函数执行的流程中横向地插入了一段逻辑。在特定的业务领域里，能减少大量重复代码。面向切面编程还有相当多的术语，这里就不多做介绍，感兴趣的话可以去找找相关的资料。

这个例子仅用于演示，并没有考虑 foo 带有参数和有返回值的情况，完善它的重任就交给你了：）

2. Python 的额外支持

2.1. 语法糖

上面这段代码看起来似乎已经不能再精简了，Python 于是提供了一个语法糖来降低字符输入量。

```
01 import time
02
03 def timeit(func):
04     def wrapper():
05         start = time.clock()
06         func()
07         end =time.clock()
08         print 'used:', end - start
09     return wrapper
10
11 @timeit
12 def foo():
13     print 'in foo()'
14
15 foo()
```

重点关注第 11 行的@timeit，在定义上加上这一行与另外写 foo = timeit(foo)完全等价，千万不要以为@有另外的魔力。除了字符输入少了一些，还有一个额外的好处：这样看上去更有装饰器的感觉。

2.2. 内置的装饰器

内置的装饰器有三个，分别是 staticmethod、classmethod 和 property，作用分别是把类中定义的实例方法变成静态方法、类方法和类属性。由于模块里可以定义函数，所以静态方法和类方法的用处并不是太多，除非你想要完全的面向对象编程。而属性也不是不可或缺的，Java 没有属性也一样活得很滋润。从我个人的 Python 经验来看，我没有使用过 property，使用 staticmethod 和 classmethod 的频率也非常低。

```
01 class Rabbit(object):
02
03     def __init__(self, name):
04         self._name = name
05
06     @staticmethod
07     def newRabbit(name):
08         return Rabbit(name)
09
10     @classmethod
11     def newRabbit2(cls):
12         return Rabbit('')
13
14     @property
15     def name(self):
16         return self._name
```

这里定义的属性是一个只读属性，如果需要可写，则需要再定义一个 setter：

```
1 @name.setter
2 def name(self, name):
3     self._name = name
```

2.3. functools 模块

functools 模块提供了两个装饰器。这个模块是 Python 2.5 后新增的，一般来说大家用的应该都高于这个版本。但我平时的工作环境是 2.4 T-T

2.3.1. wraps(wrapped[, assigned][, updated]):

这是一个很有用的装饰器。看过前一篇反射的朋友应该知道，函数是有几个特殊属性比如函数名，在被装饰后，上例中的函数名 foo 会变成包装函数的名字 wrapper，如果你希望使用反射，可能会导致意外的结果。这个装饰器可以解决这个问题，它能将装饰过的函数的特殊属性保留。

```
01 import time
02 import functools
03
04 def timeit(func):
05     @functools.wraps(func)
06     def wrapper():
07         start = time.clock()
08         func()
09         end =time.clock()
10         print 'used:', end - start
11     return wrapper
12
13 @timeit
14 def foo():
15     print 'in foo()'
16
17 foo()
18 print foo.__name__
```

首先注意第 5 行，如果注释这一行，foo.__name__将是'wrapper'。另外相信你也注意到了，这个装饰器竟然带有一个参数。实际上，他还有另外两个可 选的参数，assigned 中的属性名将使用赋值的方式替换，而 updated 中的属性名将使用 update 的方式合并，你可以通过查看 functools 的源代码获得它们的默认值。对于这个装饰器，相当于 wrapper = functools.wraps(func)(wrapper)。

2.3.2. total_ordering(cls):

这个装饰器在特定的场合有一定用处，但是它是在 Python 2.7 后新增的。它的作用是为实现了至少__lt__、__le__、__gt__、__ge__其中一个的类加上其他的比较方法，这是一个类装饰器。如果觉得不好理解，不妨仔细看看这个装饰器的源代码

```
01 53 def total_ordering(cls):
02 54     """Class decorator that fills in missing ordering methods"""
03 55     convert = {
04 56       '__lt__': [('__gt__', lambda self, other: other < self),
05 57             ('__le__', lambda self, other: not other < self),
06 58             ('__ge__', lambda self, other: not self < other)],
07 59       '__le__': [('__ge__', lambda self, other: other <= self),
08 60             ('__lt__', lambda self, other: not other <= self),
09 61             ('__gt__', lambda self, other: not self <= other)],
10 62       '__gt__': [('__lt__', lambda self, other: other > self),
11 63             ('__ge__', lambda self, other: not other > self),
```

```
            ('__le__', lambda self, other: not self > other)],
      '__ge__': [('__le__', lambda self, other: other >= self),
            ('__gt__', lambda self, other: not other >= self),
            ('__lt__', lambda self, other: not self >= other)]
    }
    roots = set(dir(cls)) & set(convert)
    if not roots:
      raise ValueError('must define at least one ordering operation: < > <= >=')
    root = max(roots)       # prefer __lt__ to __le__ to __gt__ to __ge__
    for opname, opfunc in convert[root]:
      if opname not in roots:
        opfunc.__name__ = opname
        opfunc.__doc__ = getattr(int, opname).__doc__
        setattr(cls, opname, opfunc)
    return cls
```