python装饰器入门与提高

#1. 介绍

Python 2.2 开始提供了装饰器（decorator），装饰器作为修改函数的一种便捷方式，为程序员编写程序提供了便利性和灵活性，适当使用装饰器，能够有效的提高代码的可读性和可维护性，然而，装饰器并没有被广泛的使用，主要还是因为大多数人并不理解装饰器的工作机制。

本文首先介绍了装饰器的概念和用法（第2节），然后介绍了装饰器使用过程中的注意事项（第3节），之后讨论了装饰器的使用场景和注意是想（第4节），最后提供了一些装饰器的学习素材（第5节）。

2. 装饰器

装饰器本质上就是一个函数，这个函数接收其他函数作为参数，并将其以一个新的修改后的函数进行替换。概念比较抽象，一起来看两个装饰器的例子。

2.1 装饰器的概念

考虑这样一组函数，它们在被调用时需要对某些参数进行检查，在本例中，需要对用户名进行检查，以判断用户是否有相应的权限进行某些操作。

程序清单1

```
class Store(object):
    def get_food(self, username, food):
        if username != 'admin':
            raise Exception("This user is not allowed to get food")
        return self.storage.get(food)

    def put_food(self, username, food):
        if username != 'admin':
            raise Exception("This user is not allowed to put food")
        self.storage.put(food)
```

显然，代码有重复，作为一个有追求的工程师，我们严格遵守DRY(Don't repeat yourself)原则，于是，代码被改写成了程序清单2这样：

程序清单2

```
def check_is_admin(username):
    if username != 'admin':
        raise Exception("This user is not allowed to get food")

class Store(object):
    def get_food(self, username, food):
        check_is_admin(username)
        return self.storage.get(food)

    def put_food(self, username, food):
        check_is_admin(username)
        return self.storage.put(food)
```

现在代码整洁一点了，但是，有装饰器能够做的更好：

程序清单3

```
def check_is_admin(f):
    def wrapper(*args, **kwargs):
        if kwargs.get('username') != 'admin':
```

```
            raise Exception("This user is not allowed to get food")
        return f(*arg, **kargs)
    return wrapper

class Storage(object):
    @check_is_admin
    def get_food(self, username, food):
        return self.storage.get(food)

    @check_is_admin
    def put_food(self, username, food):
        return storage.put(food)
```

上面这段代码就是使用装饰器的典型例子:函数里面定义了一个函数，并将定义的这个函数作为返回值。这个例子足够简单，所以它的好处也不够明显，但是，却可以很好的演示装饰器的语法。

即使这样，我们也可以说，程序清单3比程序清单2更好，因为程序清单3能够将条件检查与具体逻辑分隔开来。在本例中，`check_is_admin`只是预检查，它的重要性不如具体的业务逻辑。我们将业务逻辑看做是这段程序的"重点"的话，那么，程序清单3一眼看过去就能看到"重点"，而程序清单2则不能，需要简单的思考（转个弯）才能区分条件检查和业务逻辑。当然，你可能觉得这没什么，但是，作为一名有追求的工程师，我们希望我们写出的代码能和散文一样优美。

2.2 装饰器的本质

前面说过，**装饰器本质上就是一个函数，这个函数接收其他函数作为参数，并将其以一个新的修改后的函数进行替换**。下面这个例子能够更好地理解这句话。

程序清单4

```
def bread(func):
    def wrapper():
        print "</''''''\>"
        func()
        print "</______\>"
    return wrapper

sandwich_copy = bread(sandwich)
sandwich_copy()
```

输出结果如下：

```
</''''''\>
--ham--
</______\>
```

分析如下： bread是一个函数，它接受一个函数作为参数，然后返回一个新的函数，新的函数对原来的函数进行了一些修改和扩展，且这个新函数可以当做普通函数进行调用。

下面这段代码和程序清单4输出结果一摸一样，只是用了python提供的装饰器语法，看起来更加简单直接。

**程序清单5 **

```
def bread(func):
    def wrapper():
        print "</''''''\>"
        func()
        print "</______\>"
    return wrapper

@bread
def sandwich(food="--ham--"):
    print food
```

到这里，我们应该已经能够理解装饰器的作用和用法了，再强调一遍：装饰器本质上就是一个函数，这个函数接收其他函数作为参数，并将其以一个新的修改后的函数进行替换

3. 使用装饰器需要注意的地方

我们在上一节中演示了装饰器的用法，可以看到，装饰器其实很好理解，也非常简单。但是，要用好装饰器，还有一些我们需要注意的地方，这一节就对这些需要注意的地方进行了讨论，首先讨论了在使用装饰器的情况下，如何保留原有函数的属性(见3.1节)；然后讨论了如何实现一个更加智能的装饰器；之后讨论了使用多个装饰器时，各个装饰器的调用顺序(见3.3节)；最后说明了如何给装饰器传递参数(见3.4节)。

3.1 函数的属性变化

装饰器动态创建的新函数替换原来的函数，但是，新函数缺少很多原函数的属性，如docstring和名字。

程序清单6

```
def is_admin(f):
    def wrapper(*args, **kwargs):
        if kwargs.get("username") != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*args, **kwargs)
    return wrapper

def foobar(username='someone'):
    """Do crazy stuff"""
    pass

@is_admin
def barfoo(username='someone'):
    """Do crazy stuff"""
    pass

def main():
    print foobar.func_doc
    print foobar.__name__

    print barfoo.func_doc
    print barfoo.__name__

if __name__ == '__main__':
    main()
```

程序清单6的输出结果：

```
Do crazy stuff
foobar

None
wrapper
```

程序清单6中，我们定义了两个函数`foobar`与`barfoo`，其中，`barfoo`使用装饰器进行了封装，我们获取`foobar`与`barfoo`的docstring和函数名字，可以看到，使用了装饰器的函数，不能够正确获取函数原有的docstring与名字，为了解决这个问题，可以使用python内置的 functools模块。

程序清单7

```
import functools

def is_admin(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        if kwargs.get("username") != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*arg, **kwargs)
    return wrapper
```

我们只需要增加一行代码，就能够正确地获取函数的属性。

此外，我们也可以向下面这样：

```
def is_admin(f):
    def wrapper(*args, **kwargs):
        if kwargs.get("username") != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*args, **kwargs)
    return functools.wraps(f)(wrapper) # important
```

当然，个人推荐第一种方法，因为第一种方法可读性更强。

3.2 使用inspect获取函数参数

下面看一下程序清单8,它是否会正确输出结果呢？

程序清单8

```
import functools

def check_is_admin(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        print kwargs
        if kwargs.get('username') != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*args, **kwargs)
    return wrapper


@check_is_admin
def get_food(username, food='chocolate'):
    return "{0} get food: {1}".format(username, food)


def main():
    print get_food('admin')

if __name__ == '__main__':
    main()
```

程序清单8会抛出一个异常，因为我们传入的'admin'是一个位置参数，而我们却去关键字参数(kwargs)获取用户名，因此，`kwargs.get('username')返回None，那么，权限检查发现，用户没有相应的权限，抛出异常。

为了提供一个更加智能的装饰器，我们需要使用python的inspect模块。inspect能够取出函数的签名，并对其进行操作，如下所示：

程序清单9

```
import functools
import inspect

def check_is_admin(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        func_args = inspect.getcallargs(f, *args, **kwargs)
        print func_args
        if func_args.get('username') != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*args, **kwargs)
    return wrapper


@check_is_admin
def get_food(username, food='chocolate'):
    return "{0} get food: {1}".format(username, food)


def main():
    print get_food('admin')
```

```
if __name__ == '__main__':
    main()
```

承担主要工作的函数是inspect.getcallargs，它返回一个将参数名字和值作为键值对的字典，这程序清单7中，这个函数返回`{'username':'admin', 'food':'chocolate'}`。这意味着我们的装饰器不必检查参数username是基于位置的参数还是基于关键字的参数，而只需在字典中查找即可。

3.3 多个装饰器的调用顺序

多个装饰器的调用顺序也很好理解，我们一stackoverflow上的这个问题为例进行说明。

问题

How can I make two decorators in Python that would do the following?

```
@makebold
@makeitalic
def say():
   return "Hello"
```

which should return

```
<b><i>Hello</i></b>
```

答案

```
def makebold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped

def makeitalic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped

@makebold
@makeitalic
def hello():
    return "hello world"

print hello() ## returns <b><i>hello world</i></b>
```

分析

我们在2.2节说过，装饰器就是在外层进行了封装：

```
@bread
sandwich()

sandwich_copy = bread(sandwich)
```

那么，封装两层可以像这样理解：

```
@makebold
@makeitalic
hello()

hello-copy = makebold(makeitalic(helo))
```

因此，这样理解以后，对于多个装饰器的调用顺序，就不再有疑问了。

3.4 给装饰器传递参数

下面通过一个官方的例子来看如何给装饰器传递参数。官方介绍了一个非常有用的装饰器，即设置超时器。如果函数调用超时，则抛出异常。

程序清单10

```
def timeout(seconds, error_message = 'Function call timed out'):
   def decorated(func):
       def _handle_timeout(signum, frame):
           raise TimeoutError(error_message)

       def wrapper(*args, **kwargs):
           signal.signal(signal.SIGALRM, _handle_timeout)
           signal.alarm(seconds)
           try:
               result = func(*args, **kwargs)
           finally:
               signal.alarm(0)
           return result

       return functools.wraps(func)(wrapper)

   return decorated
```

使用方法如下：

```
import time

@timeout(1, 'Function slow; aborted')
def slow_function():
    time.sleep(5)
```

对应于我们这篇博客一直讨论的例子，传递参数的代码如下所示：

程序清单11

```
def is_admin(admin='admin'):
    def decorated(f):
        @functools.wraps(f)
        def wrapper(*args, **kwargs):
            if kwargs.get("username") != admin:
                raise Exception("This user is not allowed to get food")
            return f(*args, **kwargs)
        return wrapper
    return decorated


@is_admin(admin='root')
def barfoo(username='someone'):
    """Do crazy stuff"""
    print '{0} get food'.format(username)


if __name__ == '__main__':
    barfoo(username='root')
```

4. 装饰器的使用场景与缺点

4.1 装饰器的使用场景

装饰器虽然语法比较复杂，但是，在一些场景下，也确实比较有用。包括：

- 注入参数（提供默认参数，生成参数）
- 记录函数行为（日志、缓存、计时什么的）
- 预处理／后处理（配置上下文什么的）
- 修改调用时的上下文（线程异步或者并行，类方法）

下面这个例子演示了上面提到的3中情况，如下所示：

程序清单12

```
def benchmark(func):
    """
    A decorator that prints the time a function takes
    to execute.
    """
    import time
    def wrapper(*args, **kwargs):
        t = time.clock()
        res = func(*args, **kwargs)
        print func.__name__, time.clock()-t
        return res
    return wrapper


def logging(func):
    """
    A decorator that logs the activity of the script.
    (it actually just prints it, but it could be logging!)
    """
    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs)
        print func.__name__, args, kwargs
        return res
    return wrapper


def counter(func):
    """
    A decorator that counts and prints the number of times a function has been execute
d
    """
    def wrapper(*args, **kwargs):
        wrapper.count = wrapper.count + 1
        res = func(*args, **kwargs)
        print "{0} has been used: {1}x".format(func.__name__, wrapper.count)
        return res
    wrapper.count = 0
    return wrapper

@counter
@benchmark
@logging
def reverse_string(string):
    return str(reversed(string))
```

关于装饰器的例子，官方列出了一个长长的列表，这里很多代码可以直接拿来使用，如果需要详细地了解装饰器的使用场景，可以学习一下这份列表。

4.2 装饰器有哪些缺点

在我们目前的实际项目中，装饰器使用还不够多，所以没有积累很多的经验，下面是国外大神给出的装饰器的缺点，以供参考：

- Decorators were introduced in Python 2.4, so be sure your code will be run on >= 2.4.
- Decorators slow down the function call. Keep that in mind.
- You cannot un-decorate a function. (There are hacks to create decorators that can be removed, but nobody uses them.) So once a function is decorated, it's decorated for all the code.
- Decorators wrap functions, which can make them hard to debug.

5. 其他学习资料

本文较为全面地介绍了装饰器的用法，也给出了装饰器的使用场景和缺点。如果还需要进一步的学习装饰器，可以了解一下下面这几份资料：

- python decorator library
- source code of flask
- Magic decorator syntax for asynchronous code in Python
- A Python decorator that helps ensure that a Python Process is running only once

赖明星 / 2015-08-09