看完这篇文章你还不理解 Python 装饰器，只有一种可能...

2017-08-04 刘志军 Python爱好者社区

点击上方图片报名作者微课

作者：刘志军，6年+Python使用经验， 高级开发工程师，目前在互联网医疗行业从事Web系统构架工作
个人公众号：Python之禅（微信ID：vttalk）

看完这篇文章还不理解装饰器，只有一种可能，说明我写的还不够清晰，鼓励鼓励我吧。

讲 Python 装饰器前，我想先举个例子，虽有点污，但跟装饰器这个话题很贴切。

每个人都有的内裤主要功能是用来遮羞，但是到了冬天它没法为我们防风御寒，咋办？我们想到的一个办法就是把内裤改造一下，让它变得更厚更长，这样一来，它不仅有遮羞功能，还能提供保暖，不过有个问题，这个内裤被我们改造成了长裤后，虽然还有遮羞功能，但本质上它不再是一条真正的内裤了。于是聪明的人们发明长裤，在不影响内裤的前提下，直接把长裤套在了内裤外面，这样内裤还是内裤，有了长裤后宝宝再也不冷了。装饰器就像我们这里说的长裤，在不影响内裤作用的前提下，给我们的身子提供了保暖的功效。

谈装饰器前，还要先要明白一件事，Python 中的函数和 Java、C++不太一样，Python 中的函数可以像普通变量一样当做参数传递给另外一个函数，例如：

```
def foo():
    print("foo")

def bar(func):
    func()
```

```
bar(foo)
```

正式回到我们的主题。装饰器本质上是一个 Python 函数或类，它可以让其他函数或类在不需要做任何代码修改的前提下增加额外功能，装饰器的返回值也是一个函数/类对象。它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景，装饰器是解决这类问题的绝佳设计。有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码到装饰器中并继续重用。概括的讲，装饰器的作用就是为已经存在的对象添加额外的功能。

先来看一个简单例子，虽然实际代码可能比这复杂很多：

```
def foo():
    print('i am foo')
```

现在有一个新的需求，希望可以记录下函数的执行日志，于是在代码中添加日志代码：

```
def foo():
    print('i am foo')
    logging.info("foo is running")
```

如果函数 bar()、bar2() 也有类似的需求，怎么做？再写一个 logging 在 bar 函数里？这样就造成大量雷同的代码，为了减少重复写代码，我们可以这样做，重新定义一个新的函数：专门处理日志 ，日志处理完之后再执行真正的业务代码

```
def use_logging(func):
    logging.warn("%s is running" % func.__name__)
    func()

def foo():
    print('i am foo')

use_logging(foo)
```

这样做逻辑上是没问题的，功能是实现了，但是我们调用的时候不再是调用真正的业务逻辑 foo 函数，而是换成了 use_logging 函数，这就破坏了原有的代码结构， 现在我们不得不每次都要把原来的那个 foo 函数作为参数传递给 use_logging 函数，那么有没有更好的方式的呢？当然有，答案就是装饰器。

简单装饰器

```
def use_logging(func):

    def wrapper():
        logging.warn("%s is running" % func.__name__)
        return func()   # 把 foo 当做参数传递进来时，执行func()就相当于执行foo()
    return wrapper
```

```
def foo():
    print('i am foo')

foo = use_logging(foo)  # 因为装饰器 use_logging(foo) 返回的时函数对象 wrapper，这条语句相当于
foo()                   # 执行foo()就相当于执行 wrapper()
```

use_logging 就是一个装饰器，它一个普通的函数，它把执行真正业务逻辑的函数 func 包裹在其中，看起来像 foo 被 use_logging 装饰了一样，use_logging 返回的也是一个函数，这个函数的名字叫 wrapper。在这个例子中，函数进入和退出时 ，被称为一个横切面，这种编程方式被称为面向切面的编程。

@ 语法糖

如果你接触 Python 有一段时间了的话，想必你对 @ 符号一定不陌生了，没错 @ 符号就是装饰器的语法糖，它放在函数开始定义的地方，这样就可以省略最后一步再次赋值的操作。

```
def use_logging(func):

    def wrapper():
        logging.warn("%s is running" % func.__name__)
        return func()
    return wrapper

@use_logging
def foo():
    print("i am foo")

foo()
```

如上所示，有了 @ ，我们就可以省去 `foo = use_logging(foo)` 这一句了，直接调用 foo() 即可得到想要的结果。你们看到了没有，foo() 函数不需要做任何修改，只需在定义的地方加上装饰器，调用的时候还是和以前一样，如果我们有其他的类似函数，我们可以继续调用装饰器来修饰函数，而不用重复修改函数或者增加新的封装。这样，我们就提高了程序的可重复利用性，并增加了程序的可读性。

装饰器在 Python 使用如此方便都要归因于 Python 的函数能像普通的对象一样能作为参数传递给其他函数，可以被赋值给其他变量，可以作为返回值，可以被定义在另外一个函数内。

*args、**kwargs

可能有人问，如果我的业务逻辑函数 foo 需要参数怎么办？比如：

```
def foo(name):
    print("i am %s" % name)
```

我们可以在定义 wrapper 函数的时候指定参数：

```
def wrapper(name):
        logging.warn("%s is running" % func.__name__)
        return func(name)
    return wrapper
```

这样 foo 函数定义的参数就可以定义在 wrapper 函数中。这时，又有人要问了，如果 foo 函数接收两个参数呢？三个参数呢？更有甚者，我可能传很多个。当装饰器不知道 foo 到底有多少个参数时，我们可以用 *args 来代替：

```
def wrapper(*args):
        logging.warn("%s is running" % func.__name__)
        return func(*args)
    return wrapper
```

如此一来，甭管 foo 定义了多少个参数，我都可以完整地传递到 func 中去。这样就不影响 foo 的业务逻辑了。这时还有读者会问，如果 foo 函数还定义了一些关键字参数呢？比如：

```
def foo(name, age=None, height=None):
    print("I am %s, age %s, height %s" % (name, age, height))
```

这时，你就可以把 wrapper 函数指定关键字函数：

```
def wrapper(*args, **kwargs):
        # args是一个数组，kwargs一个字典
        logging.warn("%s is running" % func.__name__)
        return func(*args, **kwargs)
    return wrapper
```

带参数的装饰器

装饰器还有更大的灵活性，例如带参数的装饰器，在上面的装饰器调用中，该装饰器接收唯一的参数就是执行业务的函数 foo 。装饰器的语法允许我们在调用时，提供其它参数，比如 `@decorator(a)` 。这样，就为装饰器的编写和使用提供了更大的灵活性。比如，我们可以在装饰器中指定日志的等级，因为不同业务函数可能需要的日志级别是不一样的。

```
def use_logging(level):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if level == "warn":
                logging.warn("%s is running" % func.__name__)
            elif level == "info":
                logging.info("%s is running" % func.__name__)
            return func(*args)
        return wrapper
```

```
    return decorator

@use_logging(level="warn")
def foo(name='foo'):
    print("i am %s" % name)

foo()
```

上面的 use_logging 是允许带参数的装饰器。它实际上是对原有装饰器的一个函数封装，并返回一个装饰器。我们可以将它理解为一个含有参数的闭包。当我们使用 `@use_logging(level="warn")` 调用的时候，Python 能够发现这一层的封装，并把参数传递到装饰器的环境中。

`@use_logging(level="warn")` 等价于 `@decorator`

类装饰器

没错，装饰器不仅可以是函数，还可以是类，相比函数装饰器，类装饰器具有灵活度大、高内聚、封装性等优点。使用类装饰器主要依靠类的 `__call__` 方法，当使用 @ 形式将装饰器附加到函数上时，就会调用此方法。

```
class Foo(object):
    def __init__(self, func):
        self._func = func

    def __call__(self):
        print ('class decorator runing')
        self._func()
        print ('class decorator ending')

@Foo
def bar():
    print ('bar')

bar()
```

functools.wraps

使用装饰器极大地复用了代码，但是他有一个缺点就是原函数的元信息不见了，比如函数的 `docstring`、`__name__`、参数列表，先看例子：

```
# 装饰器
def logged(func):
    def with_logging(*args, **kwargs):
        print func.__name__      # 输出 'with_logging'
        print func.__doc__       # 输出 None
        return func(*args, **kwargs)
```

```
    return with_logging

# 函数
@logged
def f(x):
   """does some math"""
   return x + x * x

logged(f)
```

不难发现，函数 f 被 `with_logging` 取代了，当然它的 `docstring` ， `__name__` 就是变成了 `with_logging` 函数的信息了。好在我们有 `functools.wraps` ， `wraps` 本身也是一个装饰器，它能把原函数的元信息拷贝到装饰器里面的 func 函数中，这使得装饰器里面的 func 函数也有和原函数 foo 一样的元信息了。

```
from functools import wraps
def logged(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print func.__name__      # 输出 'f'
        print func.__doc__       # 输出 'does some math'
        return func(*args, **kwargs)
    return with_logging

@logged
def f(x):
   """does some math"""
   return x + x * x
```

装饰器顺序

一个函数还可以同时定义多个装饰器，比如：

```
@a
@b
@c
def f ():
    pass
```

它的执行顺序是从里到外，最先调用最里层的装饰器，最后调用最外层的装饰器，它等效于

```
f = a(b(c(f)))
```

Python是一门非常独特的编程语言，它不仅简单易学而且非常强大，有过编程经验的程序员第一次接触Python会大呼”居然可以这样玩”，从此路转粉，“The Zen of Python”被Pythoneer视为编程教条，究竟什么样的代码才称得上地道呢？我们怎样才能写出地道的Python代码？

点击**阅读原文**或扫码咨询

阅读原文