

阅读: 177726

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。


```
>>> def now():
...     print('2015-3-25')
...
>>> f = now
>>> f()
2015-3-25
```

函数对象有一个 `__name__` 属性，可以拿到函数的名字：


```
>>> now.__name__
'now'
>>> f.__name__
'now'
```

现在，假设我们要增强 `now()` 函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改 `now()` 函数的定义，这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。

本质上，**decorator** 就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的 **decorator**，可以定义如下：



```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```



观察上面的 `log`，因为它是一个 **decorator**，所以接受一个函数作为参数，并返回一个函数。我们要借助Python的 `@` 语法，把 **decorator** 置于函数的定义处：

```
@log
def now():
    print('2015-3-25')
```

调用 `now()` 函数，不仅会运行 `now()` 函数本身，还会在运行 `now()` 函数前打印一行日志：

```
>>> now()
call now():
2015-3-25
```

把 `@log` 放到 `now()` 函数的定义处，相当于执行了语句：

```
now = log(now)
```

由于 `log()` 是一个 **decorator**，返回一个函数，所以，原来的 `now()` 函数仍然存在，只是现在同名的 `now` 变量指向了新的函数，于是调用 `now()` 将执行新函数，即在 `log()` 函数中返回的 `wrapper()` 函数。

`wrapper()` 函数的参数定义是 `(*args, **kw)`，因此，`wrapper()` 函数可以接受任意参数的调用。在 `wrapper()` 函数内，首先打印日志，再紧接着调用原始函数。

如果 **decorator** 本身需要传入参数，那就需要编写一个返回 **decorator** 的高阶函数，写出来会更复杂。比如，要自定义 `log` 的文本：

```
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print(' %s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

这个3层嵌套的 **decorator** 用法如下：

```
@log('execute')
def now():
    print('2015-3-25')
```

执行结果如下：

```
>>> now()
execute now():
2015-3-25
```

和两层嵌套的 **decorator** 相比，3层嵌套的效果是这样的：

```
>>> now = log('execute')(now)
```

我们来剖析上面的语句，首先执行 `log('execute')`，返回的是 **decorator** 函数，再调用返回的函数，参数是 `now` 函数，返回值最终是 **wrapper** 函数。

以上两种 **decorator** 的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有 `__name__` 等属性，但你看去经过 **decorator** 装饰之后的函数，它们的 `__name__` 已经从原来的 `'now'` 变成了 `'wrapper'`：

```
>>> now.__name__
'wrapper'
```

因为返回的那个 `wrapper()` 函数名字就是 `'wrapper'`，所以，需要把原始函数的 `__name__` 等属性复制到 `wrapper()` 函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写 `wrapper.__name__ = func.__name__` 这样的代码，Python 内置的 `functools.wraps` 就是干这个事的，所以，一个完整的 **decorator** 的写法如下：

```
import functools

def log(func):
    @functools.wraps(func)
```

```
def wrapper(*args, **kw):
    print('call %s():' % func.__name__)
    return func(*args, **kw)
return wrapper
```

或者针对带参数的decorator:

```
import functools

def log(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print(' %s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

`import functools` 是导入 `functools` 模块。模块的概念稍候讲解。现在，只需记住在定义 `wrapper()` 的前面加上 `@functools.wraps(func)` 即可。

## 小结

在面向对象（OOP）的设计模式中，decorator被称为装饰模式。OOP的装饰模式需要通过继承和组合来实现，而Python除了能支持OOP的decorator外，直接从语法层次支持decorator。Python的decorator可以用函数实现，也可以用类实现。

decorator可以增强函数的功能，定义起来虽然有点复杂，但使用起来非常灵活和方便。

请编写一个decorator，能在函数调用的前后打印出 'begin call' 和 'end call' 的日志。

再思考一下能否写出一个 `@log` 的decorator，使它既支持：

```
@log
def f():
    pass
```

又支持：

```
@log('execute')
def f():
    pass
```

## 参考源码

[decorator.py](#)

感觉本站内容不错，读后有收获？

¥ 我要小额赞助，鼓励作者写出更好的教程

还可以分享给朋友



## 珠峰NODE.JS全栈开发

技术陪伴成长社区 珠峰培训

麦子学院  
www.maiziedu.com

百万级python导师亲身指导

## 保你120天 变身python大牛

有时候，你需要的只是一句点拨

立即咨询

## 掘金

一个只有高手分享  
的技术社区

立即加入

> 家 三

## 深度学习在线课程

通向无人驾驶的必经之路

➡

< [匿名函数](#)

[偏函数](#) >

评论

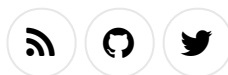
发表评论

Sign In to Make a Comment

---

廖雪峰的官方网站©2015  
Powered by [iTranswarp.js](#)  
由[阿里云](#)托管  
[广告合作](#)

---



友情链接: [中华诗词](#) - [阿里云](#) - [SICP](#) - [4closure](#)

