

## 17 | 强大的装饰器

2019-06-17 景霄

Python核心技术与实战

[进入课程 >](#)



讲述：冯永吉

时长 09:34 大小 8.77M



你好，我是景霄。这节课，我们一起来学习装饰器。

装饰器一直以来都是 Python 中很有用、很经典的一个 feature，在工程中的应用也十分广泛，比如日志、缓存等等的任务都会用到。然而，在平常工作生活中，我发现不少

人，尤其是初学者，常常因为其相对复杂的表示，对装饰器望而生畏，认为它 “too fancy to learn” ，实际并不如此。

今天这节课，我会以前面所讲的函数、闭包为切入点，引出装饰器的概念、表达和基本用法，最后，再通过实际工程中的例子，让你再次加深理解。


接下来，让我们进入正文一起学习吧！

## 函数 -> 装饰器

### 函数核心回顾

引入装饰器之前，我们首先一起来复习一下，必须掌握的函数的几个核心概念。

第一点，我们要知道，在 Python 中，函数是一等公民（first-class citizen），函数也是对象。我们可以把函数赋予变量，比如下面这段代码：


 复制代码

```
1 def func(message):
2     print('Got a message: {}'.format(message))
3
4 send_message = func
```

```
5 send_message('hello world')
6
7 # 输出
8 Got a message: hello world
```

这个例子中，我们把函数 `func()` 赋予了变量 `send_message`，这样之后你调用 `send_message`，就相当于调用函数 `func()`。


第二点，我们可以把函数当作参数，传入另一个函数中，比如下面这段代码：

 复制代码

```
1 def get_message(message):
2     return 'Got a message: ' + message
3
4
5 def root_call(func, message):
6     print(func(message))
7
8 root_call(get_message, 'hello world')
9
10 # 输出
11 Got a message: hello world
```

这个例子中，我们就把函数 `get_message()` 以参数的形式，传入了函数 `root_call()` 中然后调用它。


第三点，我们可以在函数里定义函数，也就是函数的嵌套。这里我同样举了一个例子：

 复制代码

```
1 def func(message):
2     def get_message(message):
3         print('Got a message: {}'.format(message))
4     return get_message(message)
5
6 func('hello world')
7
8 # 输出
9 Got a message: hello world
```

这段代码中，我们在函数 `func()` 里又定义了新的函数 `get_message()`，调用后作为 `func()` 的返回值返回。

第四点，要知道，函数的返回值也可以是函数对象（闭包），比如下面这个例子：

 复制代码


```
1 def func_closure():
2     def get_message(message):
```

```
3         print('Got a message: {}'.format(message))
4     return get_message
5
6 send_message = func_closure()
7 send_message('hello world')
8
9 # 输出
10 Got a message: hello world
```

这里，函数 `func_closure()` 的返回值是函数对象 `get_message()` 本身，之后，我们将其赋予变量 `send_message`，再调用 `send_message( 'hello world' )`，最后输出了 `'Got a message: hello world'`。

## 简单的装饰器

简单的复习之后，我们接下来学习今天的新知识——装饰器。按照习惯，我们可以先来看一个装饰器的简单例子：

 复制代码


```
1 def my_decorator(func):
2     def wrapper():
3         print('wrapper of decorator')
4         func()
5     return wrapper
6
```

```
7 def greet():
8     print('hello world')
9
10 greet = my_decorator(greet)
11 greet()
12
13 # 输出
14 wrapper of decorator
15 hello world
```

这段代码中，变量 `greet` 指向了内部函数 `wrapper()`，而内部函数 `wrapper()` 中又会调用原函数 `greet()`，因此，最后调用 `greet()` 时，就会先打印 'wrapper of decorator'，然后输出 'hello world'。

这里的函数 `my_decorator()` 就是一个装饰器，它把真正需要执行的函数 `greet()` 包裹在其中，并且改变了它的行为，但是原函数 `greet()` 不变。

事实上，上述代码在 Python 中有更简单、更优雅表示：

 复制代码

```
1 def my_decorator(func):
2     def wrapper():
3         print('wrapper of decorator')
4         func()
5     return wrapper
```


```
6
7 @my_decorator
8 def greet():
9     print('hello world')
10
11 greet()
```

这里的@，我们称之为语法糖，@my\_decorator就相当于前面的greet=my\_decorator(greet)语句，只不过更加简洁。因此，如果你的程序中有其它函数需要做类似的装饰，你只需在它们的上方加上@decorator就可以了，这样就大大提高了函数的重复利用和程序的可读性。

## 带有参数的装饰器

你或许会想到，如果原函数 greet() 中，有参数需要传递给装饰器怎么办？


一个简单的办法，是可以在对应的装饰器函数 wrapper() 上，加上相应的参数，比如：

 复制代码

```
1 def my_decorator(func):
2     def wrapper(message):
3         print('wrapper of decorator')
4         func(message)
```

```
5     return wrapper
6
7
8 @my_decorator
9 def greet(message):
10     print(message)
11
12
13 greet('hello world')
14
15 # 输出
16 wrapper of decorator
17 hello world
```

不过，新的问题来了。如果我另外还有一个函数，也需要使用 `my_decorator()` 装饰器，但是这个新的函数有两个参数，又该怎么办呢？比如：


 复制代码

```
1 @my_decorator
2 def celebrate(name, message):
3     ...
```

事实上，通常情况下，我们会把 `*args` 和 `**kwargs`，作为装饰器内部函数 `wrapper()` 的参数。 `*args` 和 `**kwargs`，



表示接受任意数量和类型的参数，因此装饰器就可以写成下面的形式：


 复制代码

```
1 def my_decorator(func):
2     def wrapper(*args, **kwargs):
3         print('wrapper of decorator')
4         func(*args, **kwargs)
5     return wrapper
```

## 带有自定义参数的装饰器

其实，装饰器还有更大程度的灵活性。刚刚说了，装饰器可以接受原函数任意类型和数量的参数，除此之外，它还可以接受自己定义的参数。

举个例子，比如我想要定义一个参数，来表示装饰器内部函数被执行的次数，那么就可以写成下面这种形式：


 复制代码

```
1 def repeat(num):
2     def my_decorator(func):
3         def wrapper(*args, **kwargs):
4             for i in range(num):
5                 print('wrapper of decorator')
6                 func(*args, **kwargs)
```

```
7         return wrapper
8     return my_decorator
9
10
11 @repeat(4)
12 def greet(message):
13     print(message)
14
15 greet('hello world')
16
17 # 输出:
18 wrapper of decorator
19 hello world
20 wrapper of decorator
21 hello world
22 wrapper of decorator
23 hello world
24 wrapper of decorator
25 hello world
```

## 原函数还是原函数吗？

现在，我们再来看个有趣的现象。还是之前的例子，我们试着打印出 `greet()` 函数的一些元信息：


 复制代码

```
1 greet.__name__
2 ## 输出
3 'wrapper'
4
```

```
5 help(greet)
6 # 输出
7 Help on function wrapper in module __main__:
8
9 wrapper(*args, **kwargs)
```

你会发现，`greet()` 函数被装饰以后，它的元信息变了。元信息告诉我们“它不再是以前的那个 `greet()` 函数，而是被 `wrapper()` 函数取代了”。

为了解决这个问题，我们通常使用内置的装饰器 `@functools.wrap`，它会帮助保留原函数的元信息（也就是将原函数的元信息，拷贝到对应的装饰器函数里）。

 复制代码


```
1 import functools
2
3 def my_decorator(func):
4     @functools.wraps(func)
5     def wrapper(*args, **kwargs):
6         print('wrapper of decorator')
7         func(*args, **kwargs)
8     return wrapper
9
10 @my_decorator
11 def greet(message):
12     print(message)
13
```

```
14 greet.__name__
15
16 # 输出
17 'greet'
```

## 类装饰器

前面我们主要讲了函数作为装饰器的用法，实际上，类也可以作为装饰器。类装饰器主要依赖于函数 `__call__()`，每当你调用一个类的示例时，函数 `__call__()` 就会被执行一次。

我们来看下面这段代码：

 复制代码

```
1 class Count:
2     def __init__(self, func):
3         self.func = func
4         self.num_calls = 0
5
6     def __call__(self, *args, **kwargs):
7         self.num_calls += 1
8         print('num of calls is: {}'.format(self.num_calls))
9         return self.func(*args, **kwargs)
10
11 @Count
12 def example():
13     print("hello world")
```


```
14
15 example()
16
17 # 输出
18 num of calls is: 1
19 hello world
20
21 example()
22
23 # 输出
24 num of calls is: 2
25 hello world
26
27 ...
```



这里，我们定义了类 `Count`，初始化时传入原函数 `func()`，而 `__call__()` 函数表示让变量 `num_calls` 自增 1，然后打印，并且调用原函数。因此，在我们第一次调用函数 `example()` 时，`num_calls` 的值是 1，而在第二次调用时，它的值变成了 2。


## 装饰器的嵌套

回顾刚刚讲的例子，基本都是一个装饰器的情况，但实际上，Python 也支持多个装饰器，比如写成下面这样的形式：

 复制代码


```
1 @decorator1
2 @decorator2
3 @decorator3
4 def func():
5     ...
```

它的执行顺序从里到外，所以上面的语句也等效于下面这行代码：

 复制代码

```
1 decorator1(decorator2(decorator3(func)))
```

这样，'hello world'这个例子，就可以改写成下面这样：

 复制代码

```
1 import functools
2
3 def my_decorator1(func):
4     @functools.wraps(func)
5     def wrapper(*args, **kwargs):
6         print('execute decorator1')
7         func(*args, **kwargs)
8     return wrapper
9
```

```
10
11 def my_decorator2(func):
12     @functools.wraps(func)
13     def wrapper(*args, **kwargs):
14         print('execute decorator2')
15         func(*args, **kwargs)
16     return wrapper
17
18
19 @my_decorator1
20 @my_decorator2
21 def greet(message):
22     print(message)
23
24
25 greet('hello world')
26
27 # 输出
28 execute decorator1
29 execute decorator2
30 hello world
```

## 装饰器用法实例


到此，装饰器的基本概念及用法我就讲完了，接下来，我将结合实际工作中的几个例子，带你加深对它的理解。

## 身份认证

首先是最常见的身份认证的应用。这个很容易理解，举个最常见的例子，你登录微信，需要输入用户名密码，然后点击确认，这样，服务器端便会查询你的用户名是否存在、是否和密码匹配等等。如果认证通过，你就可以顺利登录；如果不通过，就抛出异常并提示你登录失败。

再比如一些网站，你不登录也可以浏览内容，但如果你想要发布文章或留言，在点击发布时，服务器端便会查询你是否登录。如果没有登录，就不允许这项操作等等。

我们来看一个大概的代码示例：

 复制代码

```
1 import functools
2
3 def authenticate(func):
4     @functools.wraps(func)
5     def wrapper(*args, **kwargs):
6         request = args[0]
7         if check_user_logged_in(request): # 如果用户处于
8             return func(*args, **kwargs) # 执行函数 post
9         else:
10             raise Exception('Authentication failed')
11     return wrapper
12
13 @authenticate
14 def post_comment(request, ...)
15     ...
16
```




这段代码中，我们定义了装饰器 `authenticate`；而函数 `post_comment()`，则表示发表用户对某篇文章的评论。每次调用这个函数前，都会先检查用户是否处于登录状态，如果是登录状态，则允许这项操作；如果没有登录，则不允许。

## 日志记录

日志记录同样是很常见的一个案例。在实际工作中，如果你怀疑某些函数的耗时过长，导致整个系统的 `latency`（延迟）增加，所以想在线上测试某些函数的执行时间，那么，装饰器就是一种很常用的手段。

我们通常用下面的方法来表示：

 复制代码

```
1 import time
2 import functools
3
4 def log_execution_time(func):
5     @functools.wraps(func)
6     def wrapper(*args, **kwargs):
7         start = time.perf_counter()
8         res = func(*args, **kwargs)
9         end = time.perf_counter()
```

```
10         print('{} took {} ms'.format(func.__name__, (er
11         return res
12     return wrapper
13
14 @log_execution_time
15 def calculate_similarity(items):
16     ...
```



这里，装饰器 `log_execution_time` 记录某个函数的运行时间，并返回其执行结果。如果你想计算任何函数的执行时间，在这个函数上方加上 `@log_execution_time` 即可。


## 输入合理性检查

再来看今天要讲的第三个应用，输入合理性检查。

在大型公司的机器学习框架中，我们调用机器集群进行模型训练前，往往会用装饰器对其输入（往往是很长的 json 文件）进行合理性检查。这样就可以大大避免，输入不正确对机器造成的巨大开销。

它的写法往往是下面的格式：

```
1 import functools
```

 复制代码

```
2
3 def validation_check(input):
4     @functools.wraps(func)
5     def wrapper(*args, **kwargs):
6         ... # 检查输入是否合法
7
8 @validation_check
9 def neural_network_training(param1, param2, ...):
10     ...
```

其实在工作中，很多情况下都会出现输入不合理的现象。因为我们调用的训练模型往往很复杂，输入的文件有成千上万行，很多时候确实也很难发现。

试想一下，如果没有输入的合理性检查，很容易出现“模型训练了好几个小时后，系统却报错说输入的一个参数不对，成果付之一炬”的现象。这样的“惨案”，大大减缓了开发效率，也对机器资源造成了巨大浪费。

## 缓存

最后，我们来看缓存方面的应用。关于缓存装饰器的用法，其实十分常见，这里我以 Python 内置的 LRU cache 为例来说明（如果你不了解 [LRU cache](#)，可以点击链接自行查阅）。


LRU cache，在 Python 中的表示形式是`@lru_cache`。

`@lru_cache`会缓存进程中的函数参数和结果，当缓存满了以后，会删除 least recently used 的数据。

正确使用缓存装饰器，往往能极大地提高程序运行效率。为什么呢？我举一个常见的例子来说明。

大型公司服务器端的代码中往往存在很多关于设备的检查，比如你使用的设备是安卓还是 iPhone，版本号是多少。这其中的一个原因，就是一些新的 feature，往往只在某些特定的手机系统或版本上才有（比如 Android v200+）。

这样一来，我们通常使用缓存装饰器，来包裹这些检查函数，避免其被反复调用，进而提高程序运行效率，比如写成下面这样：

 复制代码

```
1 @lru_cache
2 def check(param1, param2, ...) # 检查用户设备类型，版本号
3     ...
```

## 总结

这节课，我们一起学习了装饰器的概念及用法。**所谓的装饰器，其实就是通过装饰器函数，来修改原函数的一些功能，使得原函数不需要修改。**

Decorators is to modify the behavior of the function through a wrapper so we don' t have to actually modify the function.

而实际工作中，装饰器通常运用在身份认证、日志记录、输入合理性检查以及缓存等多个领域中。合理使用装饰器，往往能极大地提高程序的可读性以及运行效率。

## 思考题

那么，你平时工作中，通常会在哪些情况下使用装饰器呢？欢迎留言和我讨论，也欢迎你把这篇文章分享给你的同事、朋友，一起交流中进步。

---

# Python 核心技术与实战

系统提升你的 Python 能力

景霄

Facebook 资深工程师



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 16 | 值传递，引用传递或其他，Python里参数是如何...

下一篇 18 | [名师分享] metaclass，是潘多拉魔盒还是阿拉丁...

## 精选留言 (45)

写留言



Wing•三金

2019-06-17

老师能否补充下，用 `@functools.wraps(func)` 来保留原来的元信息，有哪些现实意义呢？

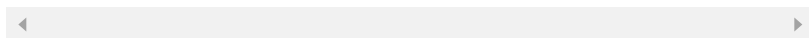


**程序员人生**

2019-06-17

我感觉python的装饰器的应用场景有点像AOP的应用场景，把一些常用的业务逻辑分离，提高程序可重用性，降低耦合度，提高开发效率。

作者回复: 是的，你的理解很正确



**Hector**

2019-06-17

lru cache常用来做一些小规模缓存，比如最近浏览记录，空间浏览记录等等，常用三种策略:1.FIFO(先进先出) 2.最少使用LRU 3.最近最少使用LRU. 看了下源码，原来python原生的functools中的lru是链表写的



**三水**

2019-06-17

请教前辈们或老师一个初入门的问题：在本方前部"函数回顾"中，把函数赋给一个变量时，

第1点: `send_message = func`

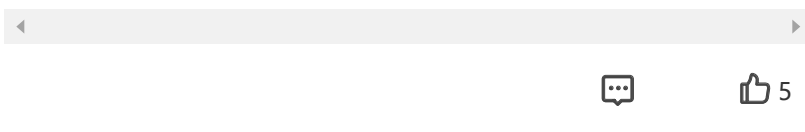
第4点: `send_message = func_closure()`

...

作者回复: 第一点:

直接赋值`send_message = func()`是错误的, 因为`func()`必须接受一个参数, `send_message = func('hello world')`就正确了, 他等同于`send_message = func`然后`send_message('hello world')`

第4点: `func_closure()`是一个闭包, 返回的是函数对象。不能直接用`send_message = func_closure`, 然后`send_message('hello world')`调用, 必须是`send_message = func_closure()`, 然后再`send_message('hello world')`, 这样才能把参数'hello world'正确传给内部函数



CN

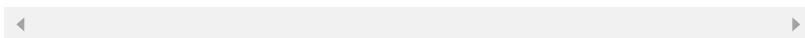
2019-06-17

- 1、总结中, 倒数第二行发现错别字(程序)不是程度。
- 2、类装饰器在实际中有哪些应用场景呢

作者回复: 欢迎指正错别字。类装饰器的用途和函数装饰器差不多, 比如文中所讲的机器学习中需要对输入进行合理性



检查，他也常常可以写成类装饰器的形式，进行调用。写成类的话，优点是程序的分解度更加高，具体用类装饰器和函数装饰器，视情况而定，二者本质是一样的

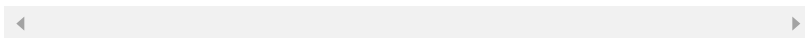


**farFlight**

2019-06-17

请问一下，lru cache不是应该删除最久没有访问的内容吗。

作者回复: LRU cache is to remove the least recently used data when the cache is full。翻译过来可能有点问题，意思就是删除最久没有访问的，我还是直接保留英文解释吧。



**GentleCP**

2019-06-18

老师，装饰器嵌套的时候，执行顺序不是decorator1->decorator2->func吗，应该是从外到内吧，外层的装饰器先执行，打印结果是decorator1

decorator2...

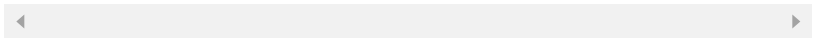


**enjoylearning**

2019-06-17

还有类装饰器，又长见识了，最近正愁参数校验放哪里，参照本文终于开窍了

作者回复: 很高兴看到你有所收获



**木木杰**

2019-06-19

没看懂啊，是我的问题吗？😁



**Miss you, but mis...**

2019-06-18

平时似乎也就property、staticmethod、classmethod用的比较多一点





**ALAN**

2019-06-18

老师，您好，`locked_cached_property`是类，它没有`__call__`方法，但是却能用来装饰另一个方法(name)，这是为什么了？以下是代码，来自flask源码

```
class locked_cached_property(object):...
```



1



1



**Fei**

2019-06-17

老师好，`num_calls`不是实例属性？`example`实例对象一次，`call`两次实例属性`num_calls`得到2。谢谢！



1



**Geek\_59f23e**

2019-06-17

1. 在类装饰器那一节中，‘每当你调用一个类的示例时’，应该是类的实例吧。

另外这里还是有点疑问，类装饰器被调用两次时`self.num_calls`这个变量不是实例变量么，第二次调用时为什么没有生成新的实例，同时把之前的实例变量清空...



1



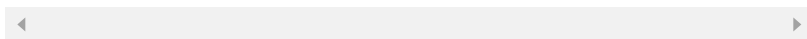
**吴星**



2019-06-17

请教下，为什么count那儿是单例模式吗？为什么二次执行会加1？

作者回复: 因为num\_calls这个变量是类变量，不是具体的实例变量，二次执行相当于调用了函数\_\_call\_\_两次，因此变量num\_calls会变为2



**峥嵘**

2019-06-23

请问老师，在“输入合理性检查”部分，为什么装饰器validation\_check的参数不是func？  
谢谢老师

```
def validation_check(input):...
```



**Topolnside**

2019-06-23

老师，您讲的例子中原来函数没有输出。如果原函数中有返回值，那么wrapper函数中是否也要return原函数的返回值？这样看来一个装饰器只能针对输出相同的函数？还是说函数的输出也能像\*args这样表达成统一格式？



**刘磊**

2019-06-22

输入合理性检查的示例中，@validation\_check(input)使用时需要带上参数吧？因为定义的时候带了参数，示例中没有参数。



**小侠龙旋风**

2019-06-22

Python内置的@property装饰器可以把类的方法伪装成属性调用的方式。

```
class Student(object):  
    def __init__(self, name, score):  
        self.name = name...
```



**刘磊**

2019-06-22

```
def myfunc(message):  
    print('Got a message:{}'.format(message))  
send_message = myfunc
```

```
def main_call(func,message):...
```



**imxintian**

2019-06-21

老师，类装饰器如果有参数，不应该\_\_init\_\_ 接收参数，而\_\_call\_\_接收func嘛？

