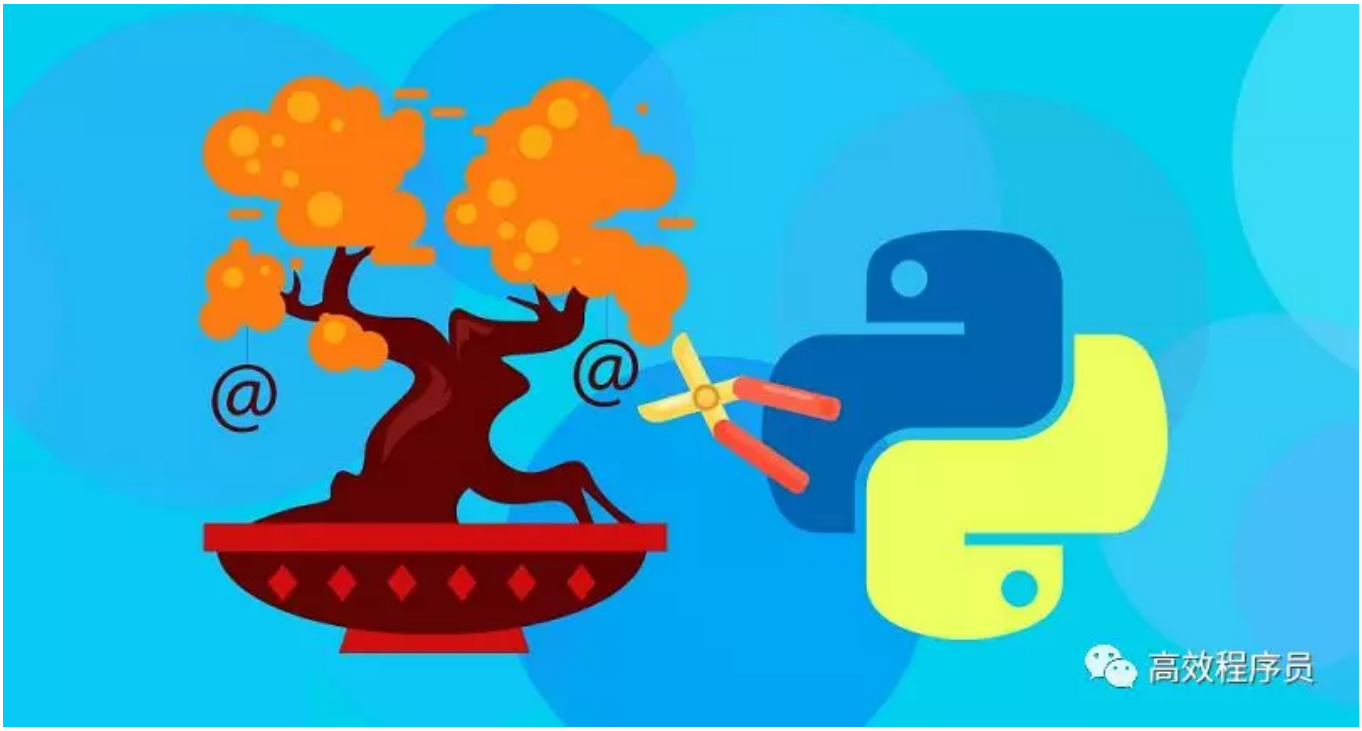
# Python 装饰器

原创：Waleon 高效程序员 2019-06-06


Python 中有一个非常有趣的特性 - 装饰器，它允许我们动态地更改行为或扩展函数的功能。




装饰器比较难懂，但是一旦理解，便能用它做很多功能强大的事情，比如：日志打印、性能检测、事务处理、缓存、权限校验等。

在深入理解装饰器之前，最好先了解一下函数的一些高级用法（参考：Python 函数是第一类对象、Python 闭包）。

# 空装饰器

所谓空装饰器，是指一个什么都不做的装饰器，这可以说是 Python 中最简单的装饰器。之所以介绍它，仅仅是为了解释语法而已！

一起来看看，如何定义一个空装饰器：

```
>>> def null_decorator(func):
...     return func
...
>>>
```

这里，null_decorator 是一个高阶函数，它接收一个函数作为输入，并将其直接返回（不做任何修改）。

现在，我们再定义一个函数：

```
>>> def greet():
...     print('Hello')
...
>>>
```

然后用 null_decorator 来装饰它：

```
>>> greet = null_decorator(greet)
>>> greet()
Hello
```

虽然语在法上没任何问题，但是这种写法不太优雅，所以 Python 支持了 @ 语法糖：

```
>>> @null_decorator
... def greet():
...     print('Hello')
...
>>>
>>> greet()
Hello
```

这等同于上面的写法，只不过更加简便罢了。

2

## 装饰无参函数

在对语法有所了解之后，是时候编写一个有实际操作的装饰器了。

假设，要定义一个打印日志的装饰器，可以这样写：

```
>>> def log(func):
...     def wrapper():
...         print('call {}()'.format(func.__name__))
...         return func()
...     return wrapper
...
>>>
```

不同于上面的空装饰器，这个装饰器并非简单地返回输入函数，而是动态地定义了一个新函数（闭包）- wrapper()，并使用它来包装输入函数，以便在调用时修改它的行为。

如果用它装饰原始的 greet() 函数，会发生什么？

```
>>> @log
... def greet():
...     print('Hello')
...
>>>
>>> greet()
call greet()
Hello
```

可以看到，在调用 greet() 函数时，不仅会运行函数本身，还会在运行之前打印一行日志。

3

## 装饰器链

此外，Python 也支持装饰器链（即：将多个装饰器应用于同一函数），这会累积它们的效果。

例如，再定义一个测试性能的装饰器：

```
>>> import time
>>>
>>> def performance(func):
...     def wrapper():
...         start_time = time.time()  # 开始时间
...         print('start time: {}'.format(start_time))
...         r = func()
...         end_time = time.time()    # 结束时间
...         print('end time: {}'.format(end_time))
...         print('take {} seconds'.format(end_time - start_time))  # 消耗的时间
...         return r
...     return wrapper
...
>>>
```

然后，将 log 和 performance 装饰器同时应用于 greet() 函数：

```
>>> @log
... @performance
... def greet():
...     print('Hello')
...
>>>
```

如果运行函数，你期望看到什么结果？是先执行 @log，还是 @performance 呢？

```
>>> greet()
call wrapper()
start time: 1559795538.9981225
Hello
end time: 1559795539.0026038
take 0.0044813156127929 69 seconds
>>>
```

很明显，这清楚地说明了装饰器的应用顺序：从下到上（即：先执行 @performance，然后执行 @log）。

如果分解上面的例子，那么函数的调用链如下所示：

```
greet = log(performance(greet))
```

先将 greet 应用于 performance，然后将结果应用于 log，从而得到包装后的函数。

4

# 装饰带参数的函数

上面的装饰器很简单，但仅适用于没有任何参数的函数。倘若我们的函数包含参数，该怎么办？

例如，为 greet() 函数添加一个参数，用于自定义问候语句：

```
>>> @log
... def greet(name):
...     print('Hello, {}.'.format(name))
...
>>>
>>> greet('Waleon')
...
TypeError: wrapper() takes 0 positional arguments but 1 was given
```

咦，分明和上面的写法一样，这里为何出错了呢？这是因为 wrapper() 不接受参数，而我们在传递时却给它指定了一个！

要解决这个问题，则需要对 log 略作修改：

```
>>> def log(func):
...     def wrapper(name):      # 加上参数
...         print('call {}()'.format(func.__name__))
...         return func(name)  # 调用时，也应该匹配
...     return wrapper
...
>>>
```

现在，再来尝试一下：

```
>>> @log
... def greet(name):
...     print('Hello, {}.'.format(name))
...
>>>
>>> greet('Waleon')
call greet()
Hello, Waleon.
```

虽然程序正常运行，但这并不是完美方案，因为这个装饰器不适用于任意数量参数的函数。

其实，实现这样一个通用装饰器非常简单，这个魔法交由 * args 和 ** kwargs 来完成就好了：

```
>>> def log(func):
...     def wrapper(*args, **kwargs):
...         print('call {}()'.format(func.__name__))
...         return func(*args, **kwargs)
...     return wrapper
...
>>>
```

不妨尝试一下，使用不同数量参数的函数：

```
>>> @log
... def greet():      # 无参
...     print('Hello')
...
>>> greet()
call greet()
Hello
>>>
>>> @log
... def greet(name):  # 一个参数
...     print('Hello, {}.'.format(name))
...
>>> greet('Waleon')
call greet()
Hello, Waleon.
```

通常，还可以用这种方式来追踪函数的参数以及返回值。

5

## 复制元数据

在使用装饰器时，我们是将一个函数替换为另一个函数。但这个过程有一个缺点，就是它隐藏了原始函数附带的一些元数据。

例如，原始函数的名称、docstring 和参数列表：

```
>>> @log
... def greet():
...     '''greet to someone'''
```

```
...     print('Hello')
...
>>>
```

如果尝试访问该函数的元数据，将得到的是装饰器中闭包的元数据：

```
>>> greet.__name__
'wrapper'
>>> print(greet.__doc__)
None
```

这会使调试变得困难，值得庆幸的是，Python 提供了一个快速解决方案 - functools.wraps。

可以在装饰器中使用它，以将原始函数的元数据复制到 wrapper() 中：

```
>>> import functools
>>>
>>> def log(func):
...     @functools.wraps(func)  # 重点
...     def wrapper(*args, **kwargs):
...         print('call {}()'.format(func.__name__))
...         return func(*args, **kwargs)
...     return wrapper
...
>>>
```

验证一下，和期望结果一样：

```
>>> @log
... def greet():
...     '''greet to someone'''
...     print('Hello')
...
>>>
>>> greet.__name__
'greet'
>>> greet.__doc__
'greet to someone'
```

**建议：**应尽量在装饰器中使用 functools.wraps，这是一种很好的编程习惯。不需要花费太多时间，还能避免调试带来的麻烦，何乐而不为！