**装饰器系列(2)-装饰器进阶函数相关**

上次的文件中我们学习了装饰器的基础知识，具体可以才能够以下链接进行查看：

**装饰器系列(1)-你也能看懂的装饰器基础知识**

本次文章我们一起来学习一下带参数的装饰器的相关知识,**建议大家先学习一下Python中闭包的相关知识。**

**一、带参数的函数装饰器**

在上篇文章中我们写了一个不带参数的装饰器，具体如下：

import time  
  
def use\_time(func):  
 def wrapper(\*args,\*\*kwargs):  
 start\_t = time.time()  
 myf = func(\*args,\*\*kwargs)  
 cost\_time = time.time()-start\_t  
 print(f"函数 {func.\_\_name\_\_} 运行的时间为 {cost\_time}")  
 return myf  
 return wrapper  
  
@use\_time  
def add(a,b):  
 c1 = a + b  
 print('我是add的操作')  
*#调用*  
add(1,3)

**# 输出如下：**

**我是add的操作**

**函数 add 运行的时间为 0.0**

上述案例中我们的装饰器是不带参数的，只是被装饰的函数add()是带有参数的，这个参数(\*args,\*\*kwargs)是不定的。对于add(a,b)这个函数我们可以将use\_time()装饰器更改为以下的样式也是可以的：

def use\_time(func):  
 def wrapper(a,b):  
 start\_t = time.time()  
 myf = func(a,b)  
 cost\_time = time.time()-start\_t  
 print(f"函数 {func.\_\_name\_\_} 运行的时间为 {cost\_time}")  
 return myf  
 return wrapper

那么如果我们需要在函数中加入一些日志，通过参数来控制是否收集日志，那么该怎么实现呢？由于装饰器可以理解为一个闭包（**可参考流畅的Python相关图书进行学习**），将函数当做参数，利用闭包在外层，再定义一个高阶的函数，用于参数的传递。

def mylogger(flag):  
 def use\_time(func):  
 def wrapper(a,b):  
 start\_t = time.time()  
 myf = func(a,b)  
 cost\_time = time.time()-start\_t  
 print(f"函数 {func.\_\_name\_\_} 运行的时间为 {cost\_time}")  
 if flag:  
 print('此次我们会收集函数日志！')  
 return wrapper  
 return use\_time  
  
@mylogger(flag='Yes')  
def add(a,b):  
 c1 = a + b  
 print('我是add的操作')  
*#调用*  
add(1,3)

**# 输出如下：**

**我是add的操作**

**函数 add 运行的时间为 0.0**

**此次我们会收集函数日志！**

可以看出，为了增加对新功能的支持，我们需要在嵌套两层的基础之上又增加了一层的嵌套。

展开来看其相当于下面的调用：

def mylogger(flag):  
 def use\_time(func):  
 def wrapper(\*args,\*\*kwargs):  
 start\_t = time.time()  
 myf = func(\*args,\*\*kwargs)  
 cost\_time = time.time()-start\_t  
 print(f"函数 {func.\_\_name\_\_} 运行的时间为 {cost\_time}")  
 if flag:  
 print('此次我们会收集函数日志！')  
 return myf  
 return wrapper  
 return use\_time  
def add(a,b):  
 c1 = a + b  
 print('我是add的操作')  
  
\_use\_time = mylogger(flag = "Yes")  
add\_1 = \_use\_time(add)  
add\_1(1,2)

**# 输出**

**我是add的操作**

**函数 add 运行的时间为 0.0**

**此次我们会收集函数日志！**

在没有参数的时候是这样的情况：

def mylogger(flag):  
 def use\_time(func):  
 def wrapper():  
 start\_t = time.time()  
 myf = func()  
 cost\_time = time.time()-start\_t  
 print(f"函数 {func.\_\_name\_\_} 运行的时间为 {cost\_time}")  
 if flag:  
 print('此次我们会收集函数日志！')  
 return myf  
 return wrapper  
 return use\_time  
  
def add():  
 print('我是add的操作')  
  
\_use\_time = mylogger(flag = "Yes")  
add\_1 = \_use\_time(add)  
add\_1()

**# 正常输出为：**

**我是add的操作**

**函数 add 运行的时间为 0.0**

**此次我们会收集函数日志！**

**二、可选参数函数装饰器**

上面的案例中可以看出，装饰器带参数和不带参数其实还是存在一定差异的，也就是说接受参数的时候我们需要多一个嵌套，当然了通过设置一些参数，我们就将包含有参和无参的形式全部包含其中了，我们的代码可以更改为这样的形式：

def mylogger(fun=None,flag='1'):  
 def use\_time(inner\_func):  
 def wrapper(\*args,\*\*kwargs):  
 start\_t = time.time()  
 myf = inner\_func(\*args,\*\*kwargs)  
 cost\_time = time.time()-start\_t  
 print(f"函数 {inner\_func.\_\_name\_\_} 运行的时间为 {cost\_time}")  
 return myf  
 return wrapper  
 if fun is None:  
 return use\_time  
 else:  
 return use\_time(fun)

那么如果进行调用的时候，我们使用这三种情况都是可以进行使用的：

*# 1.第一种情况*@mylogger  
def add(a,b):  
 c1 = a + b  
 print('我是add的操作')  
add(1,2)  
  
*# 2.第二种情况*

@mylogger()  
def add(a,b):  
 c1 = a + b  
 print('我是add的操作')  
add(1,2)

*# 3.第三种情况*@mylogger(flag='Yes')  
def add(a,b):  
 c1 = a + b  
 print('我是add的操作')  
add(1,2)

输出是怎样的这里就不在赘述了，小伙伴们自己进行实验。可以看出通过条件fun is None来进行不同的形式(带不带参数)的子函数返回。

**三、functions.wraps()的使用**

我本人在使用装饰器的时候，发现一种情况：

def use\_time(func):  
 def wrapper(\*args,\*\*kwargs):  
 start\_t = time.time()  
 myf = func(\*args,\*\*kwargs)  
 cost\_time = time.time()-start\_t  
 print(f"函数 {func.\_\_name\_\_} 运行的时间为 {cost\_time}")  
 return myf  
 return wrapper  
  
@use\_time  
def add(a,b):  
 *"""  
 用来完成加法操作  
 """* c1 = a + b  
 print('我是add的操作')  
*# 调用*  
add(1,2)  
print(add.\_\_name\_\_)  
print(add.\_\_doc\_\_)

**输出为：**

**我是add的操作**

**函数 add 运行的时间为 0.0**

**wrapper**

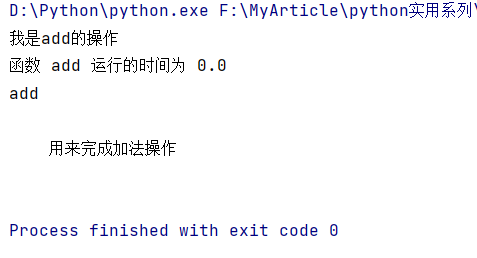
**None**

可以看出，我们想打印出函数的名称和相应的其他属性数据，可是最终输出的时候只有内层的包装函数wrapper。这就很不符合我们的要求，那我们有什么解决办法呢？

也就是说我们需要保存原始函数里面的属性，Python大佬们也发现了这个问题，我们可以使用functools模块下的wraps()函数来帮我们解决问题，具体如下：

import time  
from functools import wraps  
  
def use\_time(func):  
 @wraps(func)  
 def wrapper(\*args,\*\*kwargs):  
 start\_t = time.time()  
 myf = func(\*args,\*\*kwargs)  
 cost\_time = time.time()-start\_t  
 print(f"函数 {func.\_\_name\_\_} 运行的时间为 {cost\_time}")  
 return myf  
 return wrapper  
  
@use\_time  
def add(a,b):  
 *"""  
 用来完成加法操作  
 """* c1 = a + b  
 print('我是add的操作')  
  
add(1,2)  
print(add.\_\_name\_\_)  
print(add.\_\_doc\_\_)

此时我们的输出为：



我们在添加@wraps()来修饰函数wrapper()之后，wraps()则会根据原函数func(即我们的add()函数)来对我们的包装函数wrapper()进行一些操作，这些操作主要是对wrapper()函数的一些名称等属性进行更新，之后将原函数func的属性“移植”到包装函数上，从而保证了原始函数的元数据信息。至于更加详细的过程大家可以去找官方文档进行了解和学习。

**这就告诉我们在写装饰器的时候，就要使用@functools.wraps()来修饰包装函数。**

**四、参考文档：**

（1）《流畅的Python》-Luciano Ramalho

（2）<https://www.liaoxuefeng.com/wiki/1016959663602400/1017451662295584>

（3）https://www.cnblogs.com/tobyqin/p/python-decorator.html