**装饰器系列(3)-基于类的装饰器实现**

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(2)-装饰器进阶函数相关**

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、类装饰器-无参**

在之前的装饰器的实现中，我们都是使用函数的，即我们都是只用函数来装饰函数，那么我们能不能用类装饰器来对一些方法（函数）进行装饰呢？这个也是可以的，我们的做法是在使用的类中定义一个\_\_call\_\_方法：

class Car:  
 def \_\_init\_\_(self, func):  
 self.func = func  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 print('Start Driving')  
 callRes = self.func(\*args, \*\*kwargs)  
 return callRes  
@Car  
def use(carName, user):  
 word = f'{carName} belongs to {user}'  
 return word

res = use('BMW', 'Tom')  
print(res)

**输出为：  
Start Driving**

**BMW belongs to Tom**

可以看出，在运行函数use()的时候，这里先调用了类Car中的\_\_call\_\_函数，这就好比将类作为了装饰器使用，鉴于此，如果我们需要将类设置为装饰器用来装饰函数的时候，只需要在我们的类中定义一个\_\_call\_\_方法即可达到目的。

我们来看一下最后这个被类装饰的函数use对象的元信息：

print(use.\_\_name\_\_)  
 print(use.\_\_name\_\_)  
AttributeError: 'Car' object has no attribute '\_\_name\_\_'

可以看出函数use的元数据已经被改变了，这是因为被装饰函数use已经变成了Car的实例：

print(type(use))  
<class '\_\_main\_\_.Car'>

那么我们如何保留原函数的原信息呢？我们的代码可以更改成这样：

from functools import update\_wrapper  
class Car:  
 def \_\_init\_\_(self, func):  
 update\_wrapper(self,func)  
 self.func = func  
  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 print('Log，start Driving')  
 callRes = self.func(\*args, \*\*kwargs)  
 return callRes  
@Car  
def use(carName, user):  
 word = f'{carName} belongs to {user}'  
 return word  
  
res = use('BMW', 'Tom')  
print(res)  
print(type(use))  
print(use.\_\_name\_\_)

**输出为：**

**start Driving**

**BMW belongs to Tom**

**<class '\_\_main\_\_.Car'>**

**use**

可以看出函数的元数据得到了保留。

**二、类装饰器-有参**

上述装饰器中是没有参数的，即\_\_init\_\_中是不含参数的，我们接下来的装饰器我们来看一下有参数的形式：

import functools  
  
class Car:  
 def \_\_init\_\_(self, func,sleep=1):  
 functools.wraps(func)(self)  
 self.func = func  
 self.sleep = sleep  
  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 print(f'start Driving,wait time is {self.sleep}')  
 callRes = self.func(\*args, \*\*kwargs)  
 return callRes  
  
def sleep\_time(\*\*kwargs):  
 return functools.partial(Car,\*\*kwargs)  
  
@sleep\_time(sleep=2)  
def use(carName, user):  
 word = f'{carName} belongs to {user}'  
 return word  
  
  
res = use('BMW', 'Tom')  
print(res)  
print(type(use))  
print(use.\_\_name\_\_)

**输出为：**

**start Driving,wait time is 2**

**BMW belongs to Tom**

**<class '\_\_main\_\_.Car'>**

**use**

这样我们就实现了含有参数的形式，上述代码中需要进行相关解释：

（1）之前我们使用的是update\_wrapper(self,func)来保证函数的元信息不被改变，这里我们使用了wraps方法。根据官网文档，wraps(func)(self)本质是为类实例增加一个\_\_wrapped\_\_作为类实例属性，所以self.\_\_wrapped\_\_本质就是use函数本身。

1. 本案例中我们新增了一个函数sleep\_time()函数，该函数接受一个待装饰的函数，返回经functools.partial扩展后的functools.partial对象，经过partial扩展之后就能进行函数装饰。functools.partial的一个简单案例为：

import functools  
def add\_number\_one(a):  
 return a+1  
add=functools.partial(add\_number\_one,10)  
print(add())  
*# 11*

1. **wrapt模块的使用**

记得我们之前描述过装饰器使用之后，会隐藏被装饰函数的元数据信息：

def describe(function):  
 def wrapped(\*args, \*\*kwargs):  
 *"""函数功能描述1"""* return function(\*args, \*\*kwargs)  
 return wrapped  
  
@describe  
def print\_name():  
 *"""功能描述2"""* print('Tom')  
  
print(print\_name.\_\_name\_\_)  
print(print\_name.\_\_doc\_\_)

**输出为：**

**wrapped**

**函数功能描述1**

这明显不是我们想要的，我们之前使用方法内置方法来完成即可：

import functools  
  
def describe(function):  
 @functools.wraps(function)  
 def wrapped(\*args, \*\*kwargs):  
 *"""函数功能描述1"""* return function(\*args, \*\*kwargs)  
 return wrapped  
  
@describe  
def print\_name():  
 *"""功能描述2"""* print('Tom')  
  
print(print\_name.\_\_name\_\_)  
print(print\_name.\_\_doc\_\_)

**此时输出：**

**print\_name**

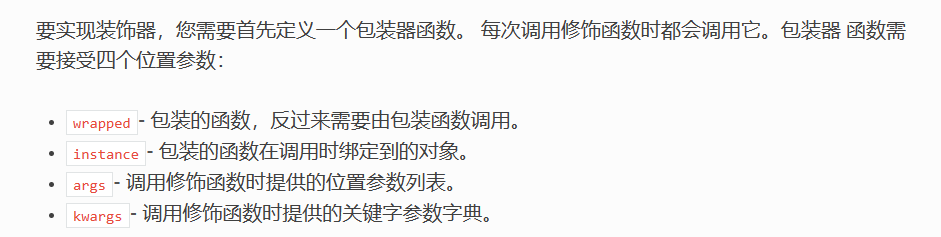
**功能描述2**

满足了我们的要求，那么我们还有什么方法可以是实现这个方法呢？我们可以使用第三方模块wrapt来实现我们的装饰器。

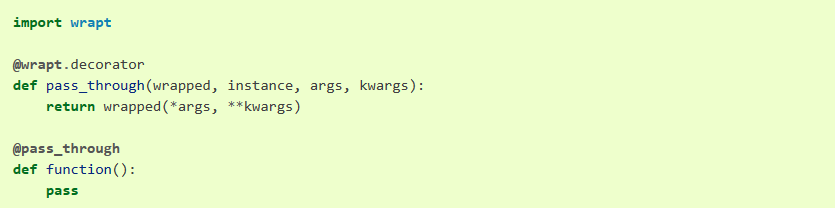
wrapt是一个功能非常完善的包，用于实现各种[装饰器](https://so.csdn.net/so/search?q=%E8%A3%85%E9%A5%B0%E5%99%A8&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/dianyin7770/article/details/101936029/_blank)，方便快捷，官方文档传送门：

<https://wrapt.readthedocs.io/en/latest/index.html>

官网文档介绍到：



不带参数的时候，其基本用法如下：



我们按照文档来实现一个我们的装饰器：

import wrapt  
  
@wrapt.decorator()  
def describe(function):  
 def wrapper(wrapped, instance, args, kwargs):  
 *"""函数功能描述1"""* return function(\*args, \*\*kwargs)  
 return wrapper  
  
@describe  
def print\_name():  
 *"""功能描述2"""* print('Tom')  
  
print(print\_name.\_\_name\_\_)  
print(print\_name.\_\_doc\_\_)

**输出：**

**print\_name**

**功能描述2**

另外我们也可以使用decorator模块，该模块是[python](https://so.csdn.net/so/search?q=python&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_39546520/article/details/_blank)用来专门封装装饰器的模块，使用decorator构造装饰器更加简便，同时被装饰的函数的元数据也保留不变，例如：

from decorator import decorator  
  
@decorator  
def describe(func,\*args, \*\*kwargs):  
 print(f"函数名称：{func.\_\_name\_\_}()")  
 return func(\*args, \*\*kwargs)  
  
@describe  
def print\_name(name):  
 *"""功能描述2"""* print(f'my name is {name}')  
  
print\_name('Tom')  
print(print\_name.\_\_name\_\_)  
print(print\_name.\_\_doc\_\_)

**输出为：**

**函数名称：print\_name()**

**my name is Tom**

**print\_name**

**功能描述2**

1. **总结**

装饰器的相关介绍就到这，装饰器的内容还是非常丰富的，大家还是需要系统的学习，比如从闭包开始，之后我也会进行相关的补充，大家多点实例操作进行理解即可。

**五、参考文档：**

（1）《流畅的Python》-Luciano Ramalho

（2）<https://www.liaoxuefeng.com/wiki/1016959663602400/1017451662295584>  
（3）https://www.zhihu.com/question/26930016/answer/99243411

（4）https://wrapt.readthedocs.io/en/latest/quick-start.html

