装饰器

装饰器也是一个函数，它是让其他函数在不改变变动的前提下增加额外的功能。

装饰器是一个闭包，把一个函数当作参数返回一个替代版的函数，本质是一个返回函数的函数（即返回值为函数对象）。python3 支持用@符号直接将装饰器应用到函数。

装饰器工作场景：插入日志、性能测试、事务处理等等。

一、简单的装饰器

很多时候我们需要计算函数的运行时间，如果一个个写很麻烦也毫无意义，通过装饰器可以只写一次，复用多次。

```
from datetime import datetime
# 计算函数运行时间的装饰器
def runtime(func):
    def wrapper(*args,**kwargs):
        print("Function Name:" + func.__name__)
        start_time = datetime.now()
        print("[{}]:The function starts.".format(start_time))
        result = func(*args,**kwargs)
        end_time = datetime.now()
        continue_time = (end_time-start_time).microseconds / 1000
        print("[{}]:The function ends.".format(end_time))
        print("[{}]:The function's runtime is {} ms".format(datetime.now(),continue_time))
        return result
    return wrapper

# 以求和函数为例
@runtime
def total(*args):
    s = 0
    for x in args:
        s = s + x
    return s

t = total(*[n for n in range(1000)])
print(t)

"""
Output:
    Function Name:sm
    [2019-05-21 15:10:30.604334]:The function starts.
    [2019-05-21 15:10:30.604834]:The function ends.
    [2019-05-21 15:10:30.604834]:The function's runtime is 0.5 ms
    499500
"""
```

可以看出装饰器的作用是：**total = runtime(total)**，因为求和函数有返回值，故装饰器函数内也需要返回值，由于在函数执行后装饰器还要有一些操作，所以可以用一个变量存储函数返回的结果，等等装饰器工作完成再返回。

二、@functools.wraps

接着上面的代码，如果执行 **print(total.__name__)**，会发现输出的不是 total，而是 wrapper，就是装饰器函数内定义的那个函数，要改变这个下次可以利用 functools 里的 wraps 装饰器，我重新写一个简单的装饰器来看看：

```
from functools import wraps
def decorator(func):
    @wraps(func)
    def wrapper(*args,**kwargs):
        func(*args,**kwargs)

    return wrapper

@decorator
def sayHi():
    pass

print(sayHi.__name__)                    # sayHi
```

三、带参数的装饰器

比如你想要输出日志，但是要指定输出的哪个文件，这时候装饰器就需要参数了

```
from functools import wraps

def log(filename):
    def log2file(func):
        @wraps(func)
        def wrappers(*args,**kwargs):
            result = func(*args,**kwargs)
            log_string = "Method Name :{}\nargs:{}\n**kwargs:{}" \
                         "\nreturn value:{}".format(func.__name__,args,kwargs,result)
            with open(filename,"a") as f:
                f.write(log_string)
            return result
        return wrappers
    return log2file

@log("log.txt")
def total(*l):
    sum = 0
    for x in l:
        sum += x
    return x

total(50)
```

log.txt 内容：

```
Method Name :total
args:(50,)
**kwargs:{}
return value:50
```

这里 total = log("log.txt")(total)，简单分析下， log("log.txt")返回的是函数 log2file，实际上变成了 total = log2file(total)，而 log2file 返回的是 wrapper 函数，所以 total = wrapper，最后调用 total 时，是有返回值的，所以 wrapper 函数也必须有返回值。

四、多个装饰器的执行顺序

```
@runtime
@log("log.txt")
def total(*l):
    sum = 0
    for x in l:
        sum += x
    return x
```

像这种同一个函数调用多个装饰器的要注意业务逻辑是否对顺序有要求，没有要求则随意谁先谁后，若有要求则必须按照调用顺序排列。

装饰器遵循的是就近原则，先执行离函数近的，再执行远的，像上面这种写法，相当于 total = runtime(log("log.txt")(total))

五、类装饰器

```
from functools import wraps

# 只打印日志
class log(object):
    def __init__(self,logfile="out.log"):
        self.__logfile = logfile

    def __call__(self, func):
        @wraps(func)
        def wrappers(*args,**kwargs):
            log_string = func.__name__ + " was called"
            with open(self.__logfile,"a") as f:
                f.write(log_string)
            self.email()
            return func(*args,**kwargs)

        return wrappers

    def email(self):
        pass

class email_log(log):
    def __init__(self,email="admin@qq.com",*args,**kwargs):
        self.__email = email
        super().__init__(*args,**kwargs)

    def email(self):
        print("邮件发送了")
```

类装饰器本质上和函数装饰器一样，函数装饰器是**@函数名**或者**@函数名(参数)**，而类装饰器是**@类名()**或者**@类名(参数)**，但是，必须实现__call__方法。以@log()为例，假设对 f() 装饰，首先是 log()创建对象，然后由于实现了__call__方法，变成了 callable 对象，所以可以执行 f = log()(f)。类装饰器还有个好处是可以通过继承，进一步简化代码和优化理解。