装饰器

装饰器是程序开发中经常会用到的一个功能，用好了装饰器，开发效率如虎添翼，所以这也是Python面试中必问的问题。但对于好多初次接触这个知识的人来讲，这个功能有点绕，自学时直接绕过去了，然后面试问到了就挂了，因为装饰器是程序开发的基础知识，这个都不会，别跟人家说你会Python, 看了下面的文章，保证你学会装饰器。

1、先明白这段代码

```
#### 第一波 ####
def foo():
    print('foo')

foo  # 表示是函数
foo()  # 表示执行foo函数

#### 第二波 ####
def foo():
    print('foo')

foo = lambda x: x + 1

foo()  # 执行lambda表达式，而不再是原来的foo函数，因为foo这个名字被重新指向了另外一个匿名函数
```

函数名仅仅是个变量，只不过指向了定义的函数而已，所以才能通过 函数名()调用，如果 函数名=xxx被修改了，那么当在执行 函数名()时，调用的就不知之前的那个函数了

2、需求来了

初创公司有N个业务部门，基础平台部门负责提供底层的功能，如：数据库操作、redis调用、监控API等功能。业务部门使用基础功能时，只需调用基础平台提供的功能即可。如下：

```
############### 基础平台提供的功能如下 ###############

def f1():
    print('f1')

def f2():
    print('f2')

def f3():
    print('f3')

def f4():
    print('f4')

############### 业务部门A 调用基础平台提供的功能 ###############

f1()
f2()
f3()
f4()

############### 业务部门B 调用基础平台提供的功能 ###############

f1()
f2()
f3()
f4()
```

目前公司有条不紊的进行着，但是，以前基础平台的开发人员在写代码时候没有关注验证相关的问题，即：基础平台的提供的功能可以被任何人使用。现在需要对基础平台的所有功能进行重构，为平台提供的所有功能添加验证机制，即：执行功能前，先进行验证。

老大把工作交给 Low B，他是这么做的：

> 跟每个业务部门交涉，每个业务部门自己写代码，调用基础平台的功能之前先验证。诶，这样一来基础平台就不需要做任何修改了。太棒了，有充足的时间泡妹子...

当天Low B 被开除了…

老大把工作交给 Low BB，他是这么做的：

```
############### 基础平台提供的功能如下 ###############

def f1():
    # 验证1
    # 验证2
    # 验证3
    print('f1')

def f2():
    # 验证1
    # 验证2
    # 验证3
    print('f2')

def f3():
    # 验证1
    # 验证2
    # 验证3
    print('f3')

def f4():
    # 验证1
    # 验证2
    # 验证3
    print('f4')

############### 业务部门不变 ###############
### 业务部门A 调用基础平台提供的功能###

f1()
f2()
f3()
f4()

### 业务部门B 调用基础平台提供的功能 ###

f1()
f2()
f3()
f4()
```

过了一周 Low BB 被开除了…

老大把工作交给 Low BBB，他是这么做的：

> 只对基础平台的代码进行重构，其他业务部门无需做任何修改

```
############### 基础平台提供的功能如下 ###############

def check_login():
    # 验证1
    # 验证2
    # 验证3
    pass


def f1():

    check_login()

    print('f1')

def f2():

    check_login()

    print('f2')

def f3():

    check_login()

    print('f3')

def f4():

    check_login()

    print('f4')
```

老大看了下Low BBB 的实现，嘴角漏出了一丝的欣慰的笑，语重心长的跟Low BBB聊了个天：

老大说：

写代码要遵循 `开放封闭` 原则，虽然在这个原则是用的面向对象开发，但是也适用于函数式编程，简单来说，它规定已经实现的功能代码不允许被修改，但可以被扩展，即：

- 封闭：已实现的功能代码块
- 开放：对扩展开发

如果将开放封闭原则应用在上述需求中，那么就不允许在函数 f1 、f2、f3、f4的内部进行修改代码，老板就给了Low BBB一个实现方案：

```
def w1(func):
    def inner():
        # 验证1
        # 验证2
        # 验证3
        func()
    return inner

@w1
def f1():
    print('f1')
@w1
def f2():
    print('f2')
@w1
def f3():
    print('f3')
@w1
def f4():
    print('f4')
```

对于上述代码，也是仅仅对基础平台的代码进行修改，就可以实现在其他人调用函数 f1 f2 f3 f4 之前都进行【验证】操作，并且其他业务部门无需做任何操作。

Low BBB心惊胆战的问了下，这段代码的内部执行原理是什么呢?

老大正要生气，突然Low BBB的手机掉到地上，恰巧屏保就是Low BBB的女友照片，老大一看一紧一抖，喜笑颜开，决定和Low BBB交个好朋友。

详细的开始讲解了：

单独以f1为例：

```
def w1(func):
    def inner():
        # 验证1
        # 验证2
        # 验证3
        func()
    return inner

@w1
def f1():
    print('f1')
```

python解释器就会从上到下解释代码，步骤如下：

1. def w1(func): ==>将w1函数加载到内存
2. @w1

没错， 从表面上看解释器仅仅会解释这两句代码，因为函数在 没有被调用之前其内部代码不会被执行。

从表面上看解释器着实会执行这两句，但是 @w1 这一句代码里却有大文章， @函数名 是python的一种语法糖。

上例@w1内部会执行一下操作：

执行w1函数

> 执行w1函数 ，并将 @w1 下面的函数作为w1函数的参数，即：**@w1 等价于 w1(f1)** 所以，内部就会去执行：

```
def inner():
    #验证 1
    #验证 2
    #验证 3
    f1()    # func是参数，此时 func 等于 f1
return inner# 返回的 inner，inner代表的是函数，非执行函数 ,其实就是将原来的 f1 函数塞进另外一个函数中
```

w1的返回值

> 将执行完的w1函数返回值 赋值 给@w1下面的函数的函数名f1 即将w1的返回值再重新赋值给 f1，即：

```
新f1 = def inner():
            #验证 1
            #验证 2
            #验证 3
            原来f1()
       return inner
```

> 所以，以后业务部门想要执行 f1 函数时，就会执行 新f1 函数，在新f1 函数内部先执行验证，再执行原来的f1函数，然后将原来f1 函数的返回值返回给了业务调用者。

如此一来， 即执行了验证的功能，又执行了原来f1函数的内容，并将原f1函数返回值 返回给业务调用者。Low BBB 你明白了吗？要是没明白的话，我晚上去你家帮你解决吧！！！

3. 再议装饰器

```
# 定义函数：完成包裹数据
def makeBold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped

# 定义函数：完成包裹数据
def makeItalic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped

@makeBold
def test1():
    return "hello world-1"

@makeItalic
def test2():
    return "hello world-2"

@makeBold
@makeItalic
def test3():
    return "hello world-3"

print(test1())
print(test2())
print(test3())
```

运行结果:

```
<b>hello world-1</b>
<i>hello world-2</i>
<b><i>hello world-3</i></b>
```

4. 装饰器(decorator)功能

1. 引入日志
2. 函数执行时间统计
3. 执行函数前预备处理
4. 执行函数后清理功能
5. 权限校验等场景
6. 缓存

5. 装饰器示例

例1:无参数的函数

```
def check_time(action):
    def do_action():
        action()
    return do_action

@check_time
def go_to_bed():
     print('去睡觉')

go_to_bed()
```

上面代码理解装饰器执行行为可理解成

```
result = check_time(go_to_bed)  # 把go_to_bed 当做参数传入给 check_time函数，再定义一个变量用来保存check_time的运行结果
result()  # check_time 函数的返回值result是一个函数，result()再调用这个函数，让它再调用go_to_bed函数
```

例2:被装饰的函数有参数

```
def check_time(action):
    def do_action(a,b):
        action(a,b)
    return do_action

@check_time
def go_to_bed(a,b):
     print('{}去{}睡觉'.format(a,b))

go_to_bed("zhangsan","床上")
```

例3:被装饰的函数有不定长参数

```
def test(cal):
    def do_cal(*args,**kwargs):
        cal(*args,**kwargs)
    return do_cal

@test
def demo(*args):
    sum = 0
    for x in args:
        sum +=x
    print(sum)

demo(1, 2, 3, 4)
```

例4:装饰器中的return

```
def test(cal):
    def do_cal(*args,**kwargs):
        return cal(*args,**kwargs)  # 需要再这里写return语句，表示调用函数，获取函数的返回值并返回
    return do_cal

@test
def demo(a,b):
    return a + b


print(demo(1, 2))  #3
```

总结:

- 一般情况下为了让装饰器更通用，可以有return

例5:装饰器带参数

```
def outer_check(time):
    def check_time(action):
        def do_action():
            if time < 22:
                return action()
            else:
                return '对不起，您不具有该权限'
        return do_action
    return check_time

@outer_check(23)
def play_game():
    return '玩儿游戏'

print(play_game())
```

提高：使用装饰器实现权限验证

以下代码不要求掌握，如果能看懂最好，如果能自己手动写出来，那就太棒了！

```
def outer_check(base_permission):
    def check_permission(action):
        def do_action(my_permission):
            if my_permission & base_permission:
                return action(my_permission)
            else:
                return '对不起，您不具有该权限'
        return do_action
    return check_permission

READ_PERMISSION = 1
WRITE_PERMISSION = 2
EXECUTE_PERMISSION = 4

@outer_check(base_permission=READ_PERMISSION)
def read(my_permission):
    return '读取数据'

@outer_check(base_permission=WRITE_PERMISSION)
def write(my_permission):
    return '写入数据'

@outer_check(base_permission=EXECUTE_PERMISSION)
def execute(my_permission):
    return '执行程序'

print(read(5))
```