15. 前方高能-装饰器初识

本节主要内容:

1. 装饰器
2. 带参数的装饰器
3. 多个装饰器同时装饰一个函数

一. 装饰器

在说装饰器之前啊. 我们先说一个软件设计的原则: 开闭原则, 又被成为开放封闭原则, 你的代码对功能的扩展是开放的, 你的程序对修改源代码是封闭的. 这样的软件设计思路可以更好的维护和开发.

开放: 对功能扩展开放

封闭: 对修改代码封闭

接下来我们来看装饰器. 首先我们先模拟一下女娲造人.

```
def create_people():
    print("女娲很厉害. 捏个泥人吹口气就成了人了")

create_people()
```

ok! 很简单. 但是现在问题来了. 上古时期啊. 天气很不稳定. 这时候呢大旱三年. 女娲再去造人啊就很困难了. 因为啥呢? 没水. 也就是说. 女娲想造人必须得先和泥. 浇点儿水才能造人.

```
def create_people():
    print("浇水")      # 添加了个浇水功能
    print("女娲很厉害. 捏个泥人吹口气就成了人了")

create_people()
```

搞定. 但是, 我们来想想. 是不是违背了我们最开始的那个约定"开闭原则", 我们是添加了新的功能. 对添加功能开放. 但是修改了源代码啊. 这个就不好了. 因为开闭原则对修改是封闭的. 那怎么办. 我们可以这样做.

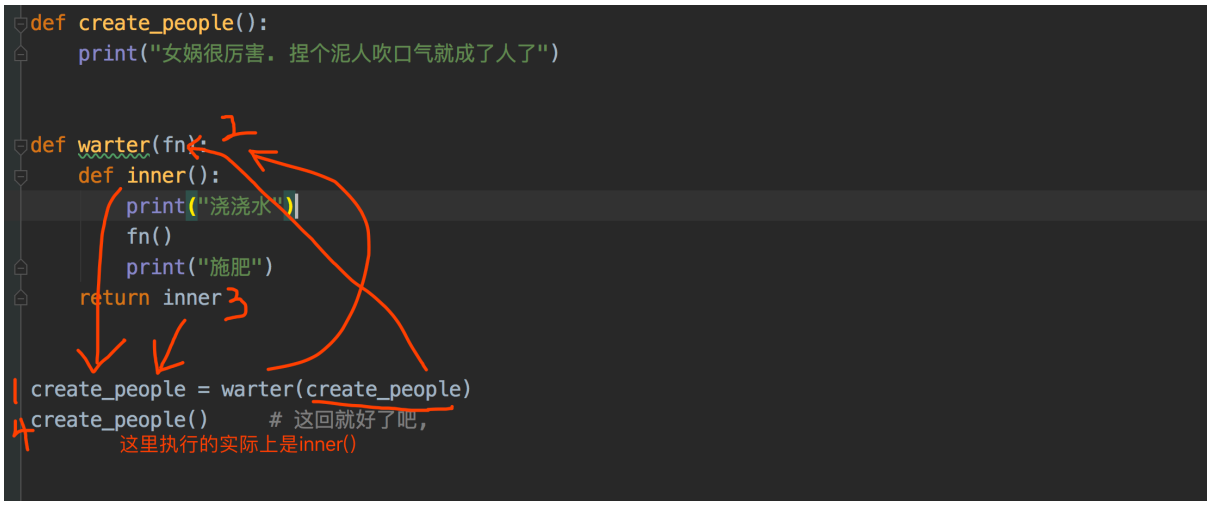
```
def create_people():
   # print("浇水")      # 添加了个浇水功能, 不符合开闭原则了
    print("女娲很厉害. 捏个泥人吹口气就成了人了")

def warter():
    print("先浇水")
    create_people()  # 造人

# create_people()    # 这个就不行了.
warter()        # 访问浇水就好了
```

现在问题又来了. 你这个函数写好了. 但是由于你添加了功能. 重新创建了个函数. 在这之前访问过这个函数的人就必须要修改代码来访问新的函数water() 这也要修改代码. 这个也不好. 依然违背开闭原则. 而且. 如果你这个函数被大量的人访问过. 你让他们所有人都去改. 那你就要倒霉了. 不干死你就见鬼了.

那怎么办才能既不修改原代码, 又能添加新功能呢? 这个时候我们就需要一个装饰器了. 装饰器的作用就是在不修改原有代码的基础上, 给函数扩展功能.

```
def create_people():
   # print("浇水")      # 添加了个浇水功能, 不符合开闭原则了
    print("女娲很厉害. 捏个泥人吹口气就成了人了")

def warter(fn):
    def inner():
        print("浇浇水")
        fn()
        print("施肥")
    return inner

# # create_people()   # 这个就不行了.
# warter()        # 访问浇水就好了

func = warter(create_people)
func()  # 有人问了. 下游访问的不依然是func么, 不还是要改么?

create_people = warter(create_people)
create_people()     # 这回就好了吧,
```

说一下执行流程:



1. 首先访问warter(create_people).

2. 把你的目标函数传递给warter的形参fn. 那么后面如果执行了fn意味着执行了你的目

标函数create_people

3. warter()执行就一句话. 返回inner函数. 这个时候. 程序认为warter()函数执行完. 那么前面的create_people函数名被重新覆盖成inner函数
4. 执行create_people函数. 实际上执行的是inner函数. 而inner中访问的恰恰使我们最开始传递进去的原始的create_people函数

结论: 我们使用warter函数把create_people给包装了一下. 在不修改create_people的前提下. 完成了对create_people函数的功能添加

这是一个装饰器的雏形. 接下来我们观察一下代码. 很不好理解. 所以呢. 我们可以使用语法糖来简化我们的代码

```
def warter(fn):
    def inner():
        print("浇浇水")
        fn()
        print("施肥")
    return inner


@warter     # 相当于 create_people = warter(create_people)
def create_people():
    print("女娲很厉害. 捏个泥人吹口气就成了人了")


create_people()

结果:
浇浇水
女娲很厉害. 捏个泥人吹口气就成了人了
施肥
```

我们发现, 代码运行的结果是一样的. 所谓的语法糖语法: @装饰器
类似的操作在我们生活中还有很多. 比方说. 约一约.

```
# 2--> 现在啊, 这个行情比较不好, 什么牛鬼蛇神都出来了.
# 那怎么办呢? 问问金老板. 金老板在前面给大家带路
# 这时候我们就需要在约之前啊. 先问问金老板了. 所以也要给函数添加一个功能, 这里依然可以使用装饰器
def wen_jin(fn):
    def inner():
        print("问问金老板, 行情怎么样, 质量好不好")
        fn()
        print("艹, 金老板骗我")
    return inner
```

```
@wen_jin
def yue():  # 1--> 约一约函数
    print("约一约")

yue()
```

ok, 接下来. 我们来看一下, 我约的话, 我想约个人. 比如约wusir, 这时, 我们要给函数添加一个参数

```
# 2--> 现在啊, 这个行情比较不好, 什么牛鬼蛇神都出来了.
# 那怎么办呢? 问问金老板. 金老板在前面给大家带路
# 这时候我们就需要在约之前啊. 先问问金老板了. 所以也要给函数添加一个功能, 这里依然可以使用装饰器
def wen_jin(fn):
    def inner():
        print("问问金老板, 行情怎么样, 质量好不好")
        fn()
        print("艹, 金老板骗我")
    return inner

@wen_jin
def yue(name):  # 1--> 约一约函数
    print("约一约", name)

yue("wusir")

结果:
Traceback (most recent call last):
  File "/Users/sylar/PycharmProjects/oldboy/fun_2.py", line 131, in <module>
    yue("wusir")
TypeError: inner() takes 0 positional arguments but 1 was given
```

程序报错. 分析原因: 我们在外面访问yue()的时候. 实际上访问的是inner函数. 而inner函数没有参数. 我们给了参数. 这肯定要报错的. 那么该怎么改呢? 给inner加上参数就好了

```
def wen_jin(fn):
    def inner(name):
        print("问问金老板, 行情怎么样, 质量好不好")
        fn(name)
        print("艹, 金老板骗我")
    return inner

@wen_jin
def yue(name):
    print("约一约", name)

yue("wusir")
```

这样就够了么? 如果我的yue()改成两个参数呢? 你是不是还要改inner. 对了. 用*args和

**kwargs来搞定多个参数的问题

```
def wen_jin(fn):
    def inner(*args, **kwargs):  # 接收任意参数
        print("问问金老板，行情怎么样，质量好不好")
        fn(*args, **kwargs)    # 把接收到的内容打散再传递给目标函数
        print("艹，金老板骗我")
    return inner

@wen_jin
def yue(name):
    print("约一约", name)

yue("wusir")
```

搞定．这时 wen_jin()函数就是一个可以处理带参数的函数的装饰器

光有参数还不够. 返回值呢?

```
def wen_jin(fn):
    def inner(*args, **kwargs):
        print("问问金老板，行情怎么样，质量好不好")
        ret = fn(*args, **kwargs)    # 执行目标函数．获取目标函数的返回值
        print("艹，金老板骗我")
        return ret    # 把返回值返回给调用者
    return inner

@wen_jin
def yue(name):
    print("约一约", name)
    return "小萝莉"    # 函数正常返回

r = yue("wusir")    # 这里接收到的返回值是inner返回的. inner的返回值是目标函数的返回值
print(r)
```

返回值和参数我们都搞定了. 接下来给出装饰器的完整模型代码(必须记住)

```
# 装饰器：对传递进来的函数进行包装．可以在目标函数之前和之后添加任意的功能．
def wrapper(func):
    def inner(*args, **kwargs):
        '''在执行目标函数之前要执行的内容'''
        ret = func(*args, **kwargs)
        '''在执行目标函数之后要执行的内容'''
        return ret
    return inner

# @wrapper 相当于  target_func = wrapper(target_func) 语法糖
@wrapper
def target_func():
```

```
    print("我是目标函数")

# 调用目标函数
target_func()
```

请把以上面的代码写10遍, 并理解. 分析每一步的作用.

接下来. 我们来看一看被装饰器装饰之后的函数名:

```
# 装饰器: 对传递进来的函数进行包装. 可以在目标函数之前和之后添加任意的功能.
def wrapper(func):
    def inner(*args, **kwargs):
        '''在执行目标函数之前要执行的内容'''
        ret = func(*args, **kwargs)
        '''在执行目标函数之后要执行的内容'''
        return ret
    return inner

# @wrapper 相当于  target_func = wrapper(target_func) 语法糖
@wrapper
def target_func():
    print("我是目标函数")

# 调用目标函数
target_func()
print(target_func.__name__)     # inner

结果:
inner
```

我们虽然访问的是target_func函数. 但是实际上执行的是inner函数. 这样就会给下游的程序员带来困惑. 之前不是一直执行的是target_func么. 为什么突然换成了inner. inner是个什么鬼?? 为了不让下游程序员有这样的困惑. 我们需要把函数名修改一下. 具体修改方案:

```
from functools import wraps # 引入函数模块

# 装饰器: 对传递进来的函数进行包装. 可以在目标函数之前和之后添加任意的功能.
def wrapper(func):
    @wraps(func)    # 使用函数原来的名字
    def inner(*args, **kwargs):
        '''在执行目标函数之前要执行的内容'''
        ret = func(*args, **kwargs)
        '''在执行目标函数之后要执行的内容'''
        return ret
    return inner

# @wrapper 相当于  target_func = wrapper(target_func) 语法糖
@wrapper
def target_func():
```

```
    print("我是目标函数")

# 调用目标函数
target_func()
print(target_func.__name__)     # 不再是inner. 而是target_func了

@wrapper
def new_target_func():
    print("我是另一个目标函数")
new_target_func()
print(new_target_func.__name__)
```

二. 装饰器传参

现在来这样一个场景. 还是昨天的约.

```
from functools import wraps

def wrapper(fn):
    @wraps(fn)
    def inner(*args, **kwargs):
        print("问问金老板啊, 行情怎么样.")
        ret = fn(*args, **kwargs)
        print("艹, 金老板骗我")
        return ret
    return inner

@wrapper
def yue():
    print("约一次又不会死")

yue()
```

那么现在如果查的很严. 怎么办呢? 打电话问金老板严不严. 那如果整体风声都不是那么紧呢. 是不是就不需要问金老板了. 所以. 我们需要一个开关来控制是否要询问金老板. 这时我们就需要给装饰器传递一个参数. 来通知装饰器要用怎么样的方式来装饰你的目标函数

```
from functools import wraps


def wrapper_out(flag):
    def wrapper(fn):
        @wraps(fn)
        def inner(*args, **kwargs):
            if flag == True:    # 查的严啊. 先问问吧
                print("问问金老板啊, 行情怎么样.")
                ret = fn(*args, **kwargs)
                print("艹, 金老板骗我")
                return ret
            else:   # 查的不严. 你慌什么
```

```
                ret = fn(*args, **kwargs)
                return ret
        return inner
    return wrapper


@wrapper_out(False)      # 传递True和False来控制装饰器内部的运行效果
def yue():
    print("约一次又不会死")


yue()
```

注意: 咱们之前的写法是@wrapper 其中wrapper是一个函数. 那么也就是说. 如果我能让wrapper这里换成个函数就行了. wrapper(True)返回的结果是wrapper也是一个函数啊. 刚刚好和前面的@组合成一个@wrapper. 依然还是原来那个装饰器. 只不过这里套了3层. 但你要能看懂. 其实还是原来那个装饰器@wrapper

执行步骤: 先执行wrapper(True) 然后再@返回值. 返回值恰好是wrapper. 结果就是@wrapper

三. 多个装饰器装饰同一个函数

先读一下这样一个代码.

```
def wrapper1(fn):
    def inner(*args, **kwargs):
        print("111")
        ret = fn(*args, **kwargs)
        print("222")
        return ret
    return inner


def wrapper2(fn):
    def inner(*args, **kwargs):
        print("333")
        ret = fn(*args, **kwargs)
        print("444")
        return ret
    return inner


@wrapper2
@wrapper1
def eat():
    print("我想吃水果")

eat()

结果:
```

```
333
111
我想吃水果
222
444
```

执行顺序: 首先@wrapper1装饰起来. 然后获取到一个新函数是wrapper1中的inner. 然后执行@wrapper2.这个时候. wrapper2装饰的就是wrapper1中的inner了. 所以. 执行顺序就像: 第二层装饰器前(第一层装饰器前(目标)第一层装饰器后)第二层装饰器后. 程序从左到右执行起来. 就是我们看到的结果