12. 前方高能-装饰器进阶

本节主要内容:

1. 通用装饰器回顾
2. 函数的有用信息
3. 带参数的装饰器
4. 多个装饰器同时装饰一个函数

一. 通用装饰器的回顾

开闭原则: 对增加功能开放. 对修改代码封闭

装饰器的作用: 在不改变原有代码的基础上给一个函数增加功能

通用装饰器的写法:

```
def wrapper(fn):
    def inner(*args, **kwargs):
        '''在目标函数之前做的事情'''
        ret = fn(*args, **kwargs)
        '''在目标函数执行之后做的事情'''
        return ret
    return inner

@wrapper
def target_func():
    '''目标函数体'''

target_func()
```

执行过程:

1. 程序从上向下, 当执行到@wrapper的时候. 把函数作为参数传递给wrapper函数. 得到inner函数. 重新赋值给target_func
2. 当执行到target_func的时候. 我们实际上执行的是inner函数. inner函数会先执行目标函数之前的代码. 然后再执行你的目标函数. 执行完目标函数最后执行的是目标函数之后的代码

二. 函数的有用信息

1. 如何给函数添加注释

```
def chi(food, drink):
    """
    这里是函数的注释, 先写一下当前这个函数是干什么的, 比如我这个函数就是一个吃
    :param food: 参数food是什么意思
```

```
    :param drink: 参数drink是什么意思
    :return: 返回的是什么东东
    """
    print(food, drink)
    return "very good"
```

2. 如何获取到函数的相关信息

```
def chi(food, drink):
    """
    这里是函数的注释，先写一下当前这个函数是干什么的，比如我这个函数就是一个吃
    :param food: 参数food是什么意思
    :param drink: 参数drink是什么意思
    :return: 返回的是什么东东
    """
    print(food, drink)
    """
    这个可以获取到么
    """
    print(chi.__doc__)      # 获取函数的文档注释
    print(chi.__name__)     # 获取到函数名称
    return "very good"


chi("吃嘛嘛香", "我不想吃")
print(chi.__doc__)
print(chi.__name__)
```

函数名.__name__可以查看函数的名字
函数名.__doc__ 可以查看函数的文档注释

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__)
```

PS: *args和**kwargs什么时候打散, 什么时候聚合

1. 接收参数的时候是聚合，参数声明
2. 传递参数的时候是打散，给函数传递实参

```
def wrapper(func):
    @wraps(func)
    def inner(*args, **kwargs):     # 这里是聚合
        '''在执行目标函数之前要执行的内容'''
```

```
        ret = func(*args, **kwargs)      # 这里是打散，这里的作用．其实就是为了保证我可以装饰所有函数而准备的
        '''在执行目标函数之后要执行的内容'''
        return ret
    return inner
```

三. 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

四. 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