8.1 今日内容介绍

8.2【理解】闭包的概念和基本使用

- 闭包的作用:

  > 闭包可以保存外部函数内的变量，不会随着外部函数调用完而销毁。

- 闭包的概念：在函数嵌套的前提下，内部函数使用了外部函数的变量，并且外部函数返回了内部函数，我们把这个**使用外部函数变量的内部函数称为闭包。**
- 闭包构成的条件:
  - 1. 在函数嵌套的前提
    2. 内部函数使用了外部函数的变量
    3. 外部函数返回了内部函数
- 基本使用:

```
# 1. 在函数嵌套的前提
def func_outer(num1):

    # 定义了一个内部函数(相当于定了一个变量)
    def func_inter(num2):

        # 2. 内部函数使用了外部函数的变量
        print(num1 + num2)

    # 3. 并且外部函数返回了内部函数(不要加括号，加括号是调用函数)
    return func_inter


if __name__ == '__main__':
    # 最终目标是保留外部函数的局部变量
    f = func_outer(100)  # 此时，f就引用了外部函数，引用计数始终未1，不会释放变量
    # 调用内部函数(闭包)
    f(200)  # f就是内部函数对象，此时才是调用了内部函数. 因为f引用了外部函数，所以可以一直获取外部函数的局部变量
    f(300)
```

8.3【理解】闭包的应用

- 实现步骤说明
  - 定义外部函数接收不同的配置信息参数，参数是人名
  - 定义内部函数接收对话信息参数
  - 在内部函数里面把配置信息和对话信息进行拼接输出

```
# 外部函数
def config_name(name):
    # 内部函数
    def say_info(info):
```

```
        print(name + ": " + info)

    return say_info

tom = config_name("Tom")
tom("你好!")
tom("你好，在吗?")

jerry = config_name("jerry")
jerry("不在，不和你玩!")
```

8.4【了解】闭包中变量问题

- 内层定义了和外层同名的变量

> 重新在内层函数中定义了新的变量，此时修改内存的变量，不会修改掉外层的变量值

- 解决办法：`nonlocal 声明变量即可`

```
# 定义一个外部函数
def func_out(num1):

    # 定义一个内部函数
    def func_inner(num2):
        # 这里本意想要修改外部num1的值，实际上是在内部函数定义了一个局部变量num1
        nonlocal num1  # 声明这个变量是外层函数的局部变量
        num1 = 10
        # 内部函数使用了外部函数的变量(num1)
        result = num1 + num2
        print("结果是:", result)

    print(num1)
    func_inner(1)  # 是在调用外层函数时，立刻调用内层函数
    print(num1)

    # 外部函数返回了内部函数，这里返回的内部函数就是闭包
    return func_inner

# 创建闭包实例
f = func_out(1)
```

8.5【理解】装饰器入门

- 装饰器的作用：在不改变源代码和源代码调用方式的基础上，增加新的功能；
- 装饰器使用：

> 1) `闭包函数有且只有一个参数`
>
> 2) `必须是函数类型`

- 写法：

```
# 装饰器：  就是一个闭包函数，但是外层函数的参数有且只能有一个，必须是函数类型。
```

```
def cheke(func):

    def inter():
        # 在闭包中增加功能
        # 扩展知识：未来可以针对不同的版本，增加不同的功能
        # if 高版本用户:

        print("请先登录...")  # 这里先用一句话代替一下实际功能
        # 调用原先的函数
        func()  # func就是被传入的那个函数对象

    return inter



def comment():
    print("发表评论....")


# 使用装饰器装饰原函数
comment = cheke(comment)  # 此处注意生成一个同名的变量，才能产生狸猫换太子的功效


if __name__ == '__main__':
    comment()
```

- 语法糖写法: 以后常用的写法

```
# 使用语法糖方式来装饰函数
@check
def comment():
    print("发表评论")
```

8.6【应用】装饰器的使用

- 装饰器实现已有函数执行时间的统计

```
# 以后的开发中，装饰器会单独一个文件，相当于一个工具包. 谁要用，谁就导入调用
# 使用装饰器统计函数的执行时间
def calculate_time(fn):

    def inter():
        begin = time.time()
        fn()
        end = time.time()
        print("函数执行花费%f" % (end-begin))
        return inter


@calculate_time
def func():
```

```
    for i in range(100000):
        print(i)


func()
```

8.7【应用】通用装饰器

- 普通参数

```
def func_out(fn):
 def inner(name):  # 如果原函数有参数，装饰器的内层函数也需要有参数
     print("请先登录...")
     fn(name)  # 此时fn就是原函数comment，原函数需要传参，name这个fn也需要传参
 return inner

@func_out  # comment = func_out(comment)   comment=inner()
def comment(name):
 print(name + "发表评论")


if __name__ == '__main__':
 comment("王岩")
```

- 可变参数

```
def func_out(fn):
 def inner(*args, **kwargs):
     print("请先登录...")
     fn(*args, **kwargs)
 return inner


@func_out
def comment(*args, **kwargs):
 print(args)
 print(kwargs["name"] + "发表评论")


if __name__ == '__main__':
 comment(1, 2, 3, 4, name="王岩")
```

- 有返回值

```
def func_out(fn):
 def inner(name):
     print("请先登录...")
     return fn(name)
 return inner

@func_out
def comment(name):
 print(name + "发表评论")
```

```
    return "吐槽完毕"


if __name__ == '__main__':
    result = comment("王岩")
    print("返回值:", result)
```

- 通用装饰器
  - 作用：可以装饰不定长参数的函数和有返回值的函数。
  - 格式：

```
def func_out(fn):
    def inner(*args, **kwargs):
        print("请先登录...")
        return fn(*args, **kwargs)
    return inner
```

- 使用：

```
def func_out(fn):
    def inner(*args, **kwargs):  # 如果原函数有参数，装饰器的内层函数也需要有参数
        print("请先登录...")
        return fn(*args, **kwargs)  # 此时fn就是原函数comment，原函数需要传参，这个fn也需要传参
    return inner


@func_out
def comment(*args, **kwargs):
    print(args)
    print(kwargs["name"] + "发表评论")
    return 100


if __name__ == '__main__':
    result = comment(1, 2, 3, 4, name="王岩")
    print("返回值", result)
```

8.8【理解】多重装饰器

- 多重装饰器：一个函数被多个装饰器装饰
- 装饰原则：

> 多个装饰器可以对函数进行多个功能的装饰，装饰顺序是由内到外的进行装饰，调用顺序正好相反。

8.9【应用】带有参数的装饰器

- 好用: 装饰器代码可以进行合并, 复用
- 语法格式:

```
# 为了让装饰器能够传递参数，需要在原先的装饰器的基础上，再嵌套一个，才能保留住参数
def func_calculate(sign):  # sign = "+"
 # 将之前的装饰器函数，整体做缩进，并在外层再嵌套一个外层函数
 def func(fn):
     def inner(a, b):
         if sign == "+":
             print("您正在执行的是加法运算，请稍等")
         else:
             print("您正在执行的是减法运算，请稍等")
         return fn(a, b)

     return inner

 return func


# @func_add  # 系统的固定翻译方式，不会识别多余的参数 add = func(add)

# @func_calculate("+")代码会分2步走
# 1. func_calculate("+")，当做普通函数，给sign赋值，并返回一个装饰器
# 2. 返回的装饰器会和前面的@组合：@func . 此时，会进行展开 add = func(add)，此时add=inner
@func_calculate("+")
def add(a, b):
 result = a + b
 return result
```

8.10【了解】类装饰器

- 作用：使用类来实现装饰器
- 类的书写：

> 必须有两个方法
>
> 1) `__init__` 接收1个参数, 这个参数是函数类型
>
> 2) `__call__` 当对象()时, 会自动调用. 将功能扩展的代码写入这里

```
# 类装饰器
# 类名默认应该大驼峰命名，但是在装饰器中，基本都是小写开头.
# 为了符合这个特性，类装饰器首字母可以小写
class check(object):

    # 在init方法中，接收1个参数，这个参数是函数类型
    def __init__(self, fn):
        self.__fn = fn

    # 在call方法中，扩展功能，并调用原函数
    def __call__(self, *args, **kwargs):
        print(args)
        print(kwargs)
        print("请先登录...")
        self.__fn()
```

```
@check
def comment():
    print("发表评论")


if __name__ == '__main__':
    comment()
```

8.11 总结