

闭包

闭包的构成条件

- 在函数嵌套（函数里面再定义函数）的前提下
- 内部函数使用了外部函数的变量（参数）
- 外部函数的返回值是内部函数的引用

闭包如何理解

- 一般来说，如果一个函数结束，函数内部的变量、参数会被释放掉；而闭包则不同，它在外函数结束时，会把内部函数中用到的外部函数的变量、参数保存到内部函数的__closure__属性中，以提供给内部函数使用

```
def outer(a):  
    b = 5  
  
    def inner():  
        c = 7  
        print(a + b + c)  
  
    return inner
```

```
""" 这里的inner_func就是一个闭包，本质上就是inner函数，  
只不过还打包了它所依赖的参数a和变量b而已 """  
inner_func = outer(3)
```

""" 自定义函数都会有一个__closure__属性，如果不是闭包函数，则返回None

如果是则返回一个由cell对象组成的元组，每个cell对象的cell_contents属性就是外部函数保存的变量 """

```
print(inner_func.__closure__)
print(inner_func.__closure__[0].cell_contents)
# 3 (外部函数保存的参数a的值)
print(inner_func.__closure__[1].cell_contents)
# 5 (外部函数保存的变量b的值)
inner_func()
```

```
def outer():
    funcs = []

    for k in range(3):
        def inner():
            return k * k
        funcs.append(inner)

    return funcs
```

""" 这里的f1, f2, f3都是闭包，本质上都是inner函数，且使用了outer函数的变量k

outer函数在结束时，将变量k保存到内部函数的__closure__属性中，而k最后为2 """

```
f1, f2, f3 = outer()
```

```
print(f1.__closure__[0].cell_contents)
print(f2.__closure__[0].cell_contents)
print(f3.__closure__[0].cell_contents)
print(f1())
print(f2())
```

```
print(f3())
```

装饰器

装饰器，顾名思义就是起装饰作用的，不改变原来函数作用的，只是在原来函数上增加些额外的功能。

装饰器并不是编码必须性，不使用装饰器也完全可以，很多时候使用它是为了：

- 使代码更加优雅，结构更加清晰
- 将实现特定的功能代码封装成装饰器，提高代码复用率，增强代码可读性

不带参数的函数装饰器

```
import time

def timer(func):

    def wrapper(sleep_time):
        t1 = time.time()
        func(sleep_time)
        t2 = time.time()
        cost_time = t2 - t1
        print(f"花费时间：{cost_time}秒")
```

```
return wrapper
```

```
@timer # 这个装饰器就是函数  
def function(sleep_time):  
    time.sleep(sleep_time)
```

```
"""
```

执行顺序说明：

代码从上往下执行，先导入`time`模块，定义`timer`函数，再执行到`@timer`装饰器，

该函数装饰器没有调用，再定义`function`函数，当被装饰的函数定义好了，则将被

装饰的函数作为参数传入装饰器函数并执行，即

`timer(function)`，返回`wrapper`函数的引用，

所以当最后执行`function(3)`时，其实等价于`wrapper(3)`，而`wrapper`函数中又调用了`func`函数，

即`function`函数 """

```
function(3)
```

带参数的函数装饰器

```
import time
```

```
def interaction(name):
```

```
    def wrapper(func):
```

```
        def deco(work_time):
```

```
            # print("deco函数被调用")
```

```

        print(f"{name}, 你好, 请把要洗的衣物放
进来!")

        func(work_time)
        print(f"{name}, 我帮你把衣物洗好了, 快
来拿!")

    return deco

    return wrapper

@interaction("张三")
def washer(work_time):
    time.sleep(work_time)
    print("滴滴滴...")

```

"""

执行顺序说明:

代码从上往下执行, 先导入`time`模块, 定义`interaction`函数, 再执行到`@interaction`装饰器, 该函数装饰器被调用, 则执行该函数, 定义`wrapper`函数, 并返回其引用, 再定义`washer`函数, 当被装饰的函数定义好了, 则将被装饰的函数作为参数传入刚刚执行装饰器返回的`wrapper`函数并执行, 即`wrapper(washer)`, 再定义`deco`函数, 并返回其引用, 所以当最后执行`washer(3)`时, 其实等价于

`deco(3)`, 而`deco`函数中又调用了`func`函数, 即`washer`函数

"""

`washer(3)`

不带参数的类装饰器

```

import time

class Timer:

    def __init__(self, func):
        self.func = func

    def __call__(self, sleep_time):
        t1 = time.time()
        self.func(sleep_time)
        t2 = time.time()
        cost_time = t2 - t1
        print(f"花费时间: {cost_time}秒")

@Timer # 这个装饰器就是类
def function(sleep_time):
    time.sleep(sleep_time)

"""

```

执行顺序说明：

代码从上往下执行，先导入`time`模块，定义`Timer`类和方法，执行到`@Timer`装饰器时，

该类装饰器没有实例化，再定义`function`函数，当被装饰的函数定义好了，则将被

装饰的函数作为参数传入类装饰器并实例化，即

`Timer(function)`，实例化调用初始化方法，

创建实例变量，`__new__`返回实例对象，当最后执行

`function(3)`时，其实等价于`Timer(function)(3)`，

即调用`__call__`方法，而该方法中又调用了`func`函数，即

`function`函数 """

`function(3)`

带参数的类装饰器

```
import time

class Interaction:

    def __init__(self, name):
        self.name = name

    def __call__(self, func):
        def deco(work_time):
            print(f"{self.name}, 你好, 请把要洗的
衣物放进来!")
            func(work_time)
            print(f"{self.name}, 我帮你把衣物洗好
了, 快来拿!")
        return deco

@Interaction("张三")
def washer(work_time):
    time.sleep(work_time)
    print("滴滴滴...")
```

"""

执行顺序说明:

代码从上往下执行, 先导入`time`模块, 定义`Interaction`类和方法, 执行到`@Interaction`装饰器时, 该类装饰器进行实例化, 调用初始化方法, 创建实例变量, `__new__`返回实例对象, 再定义`washer`函数,

当被装饰的函数定义好了，则将被装饰的函数作为参数调用实例对象，即`Interaction("张三")(washer)`，即调用`__call__`方法，定义`deco`函数并返回其引用，所以当最后执行`washer(3)`时，其实等价于`deco(3)`，而其中又调用了`func`函数，即`washer`函数 """
`washer(3)`

多个装饰器

```
import time

def deco(func):

    def wrapper1(*args):
        res = func(*args)
        return res

    return wrapper1

def timer(func):

    def wrapper2(*args):
        start_time = time.time()
        res = func(*args)
        end_time = time.time()
        print("函数耗时:{}".format(end_time-
start_time))
        return res
```



```
return wrapper2
```

```
@deco
```

```
@timer
```

```
def add(*args):  
    time.sleep(2)  
    return sum(args)
```

```
"""
```

执行顺序说明：（执行顺序参考堆栈）

代码从上往下执行，先导入time模块，定义deco函数和timer函数，执行到@deco和@timer，

这两个装饰器都没有调用，再定义add函数，当被装饰的函数定义好了，则将被装饰的函数作

为参数传入装饰器函数并执行，即timer(add)，定义wrapper2并返回其引用，然后把返回的

wrapper2作为参数传入装饰器函数并执行，即

deco(wrapper2)，定义wrapper1并返回其引用，

所以最后add(3, 4, 5)时，其实等价于wrapper1(3, 4, 5)，而执行wrapper1(3, 4, 5)时，

其中的func即deco的参数func，即wrapper2，所以又会调用wrapper2函数，而wrapper2函数

中的func即timer的参数func，即add，所以再调用add得到结果，wrapper2函数中返回的res，

再给wrapper1中的res再最后返回 """

```
print(add(3, 4, 5))
```

内置装饰器

@classmethod

- 将类中的方法装饰为类方法

@staticmethod

- 将类中的方法装饰为静态方法

@property

- 将类中的方法装饰为只读属性

```
class Student:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    @property
    def adult_age_p(self):
        return 18

    def adult_age(self):
        return 18

    @property
    def adult_flag(self):
        return self.age >= self.adult_age_p

stu = Student("张三", 18)
```

```
print(stu.adult_age()) # 没有加@property, 必须使
用正常的调用方法的形式, 在后面加()
print(stu.adult_age_p) # 加了@property, 用调用属
性的形式来调用方法, 后面不需要加()
print(stu.adult_flag) # True
stu.age = 17 # 可以修改stu对象的age属性
# stu.adult_age_p = 19 # 报错: @property将方法装
饰为只读属性, 不能修改
print(stu.adult_flag) # False
```