19.1 闭包

1.为什么要使用闭包？

问题：初中里学过函数，例如 y=kx+b, $y=ax^2 + bx + c$

以 y=kx+b 为例，请计算一条线上的过个点 即 给 x 值 计算出 y 值

 # 第 1 种

k = 1

b = 2

y = k*x+b

缺点：如果需要多次计算，那么就的写多次 y = k*x+b 这样的式子

 # 第 2 种

```
def line_2(k, b, x):
        print(k*x+b)


line_2(1, 2, 0)
line_2(1, 2, 1)
line_2(1, 2, 2)
```

缺点：如果想要计算多次这条线上的 y 值，那么每次都需要传递 k，b 的值，麻烦

```
print("-"*50)
```

 # 第 3 种: 全局变量

```
k = 1
```

```
b = 2
def line_3(x):
    print(k*x+b)

line_3(0)
line_3(1)
line_3(2)
k = 11
b = 22
line_3(0)
line_3(1)
line_3(2)
# 缺点：如果要计算多条线上的 y 值，那么需要每次对全局变量进行修改，代码会增多，麻烦

print("-"*50)

# 第 4 种：缺省参数
def line_4(x, k=1, b=2):
    print(k*x+b)

line_4(0)
line_4(1)
line_4(2)

line_4(0, k=11, b=22)
line_4(1, k=11, b=22)
line_4(2, k=11, b=22)
```

优点：比全局变量的方式好在：k, b 是函数 line_4 的一部分 而不是全局变量，因为全局变量可以任意的被其他函数所修改

缺点：如果要计算多条线上的 y 值，那么需要在调用的时候进行传递参数，麻烦

```
print("-"*50)
```

 # 第 5 种：实例对象

```
class Line5(object):
    def __init__(self, k, b):
        self.k = k
        self.b = b

    def __call__(self, x):
        print(self.k * x + self.b)


line_5_1 = Line5(1, 2)
# 对象.方法()
# 对象()
line_5_1(0)
line_5_1(1)
line_5_1(2)
line_5_2 = Line5(11, 22)
line_5_2(0)
line_5_2(1)
line_5_2(2)
```

缺点：为了计算多条线上的 y 值，所以需要保存多个 k, b 的值，因此用了很多个实例对象， 浪费资源

```
print("-"*50)

# 第 6 种：闭包

def line_6(k, b):
    def create_y(x):
        print(k*x+b)
    return create_y


line_6_1 = line_6(1, 2)
line_6_1(0)
line_6_1(1)
line_6_1(2)
line_6_2 = line_6(11, 22)
line_6_2(0)
line_6_2(1)
line_6_2(2)
```

思考：函数、匿名函数、闭包、对象 当做实参时 有什么区别？

1. 匿名函数能够完成基本的**简单功能**，，，传递是这个函数的引用 **只有功能**(lambda)

2. 普通函数能够完成较为**复杂的功能**，，，传递是这个函数的引用 **只有功能**

3. 闭包能够将较为复杂的功能，，，传递是这个闭包中的**函数以及数据**，因此传递是功能+数据（**相对于对象，占用空间少**）

4. 对象能够完成最为复杂的功能，，，传递是**很多数据+很多功能**，因此传递是功能+数据

2. 函数引用

```
def test1():
    print("--- in test1 func----")

# 调用函数
test1()

# 引用函数
ret = test1

print(id(ret))
print(id(test1))

#通过引用调用函数
ret()
```

运行结果:

```
--- in test1 func----
140212571149040
140212571149040
--- in test1 func----
```

函数与变量一样，也有一个 id 值

3. 什么是闭包

```
# 定义一个函数
def test(number):

    # 在函数内部再定义一个函数，并且这个函数用到了外边函数的变量，那么将这个函数以及用到的一些变量称之为闭包
    def test_in(number_in):
        print("in test_in 函数, number_in is %d" % number_in)
        return number+number_in
    # 其实这里返回的就是闭包的结果
    return test_in


# 给 test 函数赋值，这个 20 就是给参数 number
ret = test(20)

# 注意这里的 100 其实给参数 number_in
print(ret(100))
```

```
#注 意这里的 200 其实给参数 number_in
print(ret(200))
```

运行结果：

```
in test_in 函数, number_in is 100
120

in test_in 函数, number_in is 200
220
```

4. 看一个闭包的实际例子：

```
def line_conf(a, b):
    def line(x):
        return a*x + b
    return line

line1 = line_conf(1, 1)
line2 = line_conf(4, 5)
print(line1(5))
print(line2(5))
```

这个例子中，函数 line 与变量 a,b 构成闭包。在创建闭包的时候，我们通过 line_conf 的参数 a,b 说明了这两个变量的取值，这样，我们就确定了函数的最终形式(y = x + 1 和 y = 4x + 5)。我们只需要变换参数 a,b，就可以获得不同的直线表达函数。**由此，我们可以看到，闭包也具有提高代码可复用性的作用。**

如果没有闭包，我们需要每次创建直线函数的时候同时说明 a,b,x。这样，我们就需要更多的参数传递，也减少了代码的可移植性。

注意点:

由于闭包引用了外部函数的局部变量，则外部函数的局部变量没有及时释放，消耗内存

5. 修改外部函数中的变量

如果需要修改外部函数的变量，可以这样写不？

```
x = 300

def test1():

      x = 200

      def test2():

            print("----1----x=%d" % x)
```

```
            x = 100
            print("----2----x=%d" % x)


      return test2


t1 = test1()
t1()
```

执行会发现报错

```
UnboundLocalError: local variable 'x' referenced before assignment
```

因为我们打印 x 时，没有提前定义，需要在打印之前增加 nonlocal x，使用外部函数的变量。

```
def counter(start=0):
    def incr():
        nonlocal start
        start += 1
        return start
    return incr

c1 = counter(5)
print(c1())
print(c1())

c2 = counter(50)
print(c2())
print(c2())

print(c1())
print(c1())

print(c2())
print(c2())
```

19.2 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：

来看老大说：

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：对扩展开放**

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，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

3.装饰器实际的原理

```
def set_func(func):
    def call_func():
        print("---这是权限验证 1----")
        print("---这是权限验证 2----")
        func()
    return call_func


@set_func  # 等价于 test1 = set_func(test1)
def test1():
    print("-----test1----")



# ret = set_func(test1)
# ret()
```

test1 = set_func(test1)

test1()

同时注意，如果代码在闭包内两个函数之间，下面例子中 print("---开始进行装饰")代码在装饰函数时已经进行了执行，而不是调用 test1 的时候

```
def set_func(func):
    print("---开始进行装饰")
    def call_func(a):
        print("---这是权限验证 1----")
        print("---这是权限验证 2----")
        func(a)
    return call_func


@set_func  # 相当于 test1 = set_func(test1)
def test1(num):
    print("-----test1----%d" % num)


@set_func  # 相当于 test2 = set_func(test2)
def test2(num):
    print("-----test2----%d" % num)

# 装饰器在调用函数之前，已经被 python 解释器执行了，所以要牢记 当调用函数之前 其实已经装饰好了，尽管调用就可以了
test1(100)
```

4. 多个装饰器装饰同一个函数

```
def add_first(func):
    print("---开始进行装饰权限 1 的功能---")
    def call_func(*args, **kwargs):
        print("---这是权限验证 1----")
        return func(*args, **kwargs)
    return call_func


def add_second(func):
    print("---开始进行装饰权限 2 的功能---")
```

```
    def call_func(*args, **kwargs):
        print("---这是权限验证 2----")
        return func(*args, **kwargs)
    return call_func


@add_first
@add_second
def test1():
    print("------test1------")

test1()
```

执行结果如下：

```
---开始进行装饰权限 2 的功能---
---开始进行装饰权限 1 的功能---
---这是权限验证 1----
---这是权限验证 2----
------test1------
```

从上面执行结果可以看出，**装饰是由内而外进行装饰，离函数近的先装饰，执行是由外而内，类似于栈的操作**，但是为了避免大家出错，我们再次演练一个实例

```
# 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>
```

5. 装饰器(decorator)功能

1. 引入日志---在执行某个函数前或者函数后记录日志
2. 函数执行时间统计

```
import time

def set_func(func):
   def call_func():
      start_time = time.time()
      func()
      stop_time = time.time()
      print("alltimeis %f" % (stop_time - start_time))
   return call_func

@set_func
def test1():
   print("-----test1----")
   for i in range(100000):
      pass
```

```
test1()
```

1. 执行函数前预备处理
2. 执行函数后清理功能
3. 权限校验等场景
4. 缓存

6. 装饰器示例

例 1:无参数的函数

```
from time import ctime, sleep

def timefun(func):
    def wrapped_func():
        print("%s called at %s" % (func.__name__, ctime()))
        func()
    return wrapped_func

@timefun
def foo():
    print("I am foo")

foo()
sleep(2)
foo()
```

上面代码理解装饰器执行行为可理解成

```
foo = timefun(foo)
# foo 先作为参数赋值给 func 后,foo 接收指向 timefun 返回的 wrapped_func
foo()
# 调用 foo(),即等价调用 wrapped_func()
# 内部函数 wrapped_func 被引用，所以外部函数的 func 变量(自由变量)并没有释放
# func 里保存的是原 foo 函数对象
```

例 2:被装饰的函数有参数

```
from time import ctime, sleep

def timefun(func):
    def wrapped_func(a, b):
        print("%s called at %s" % (func.__name__, ctime()))
        print(a, b)
        func(a, b)
    return wrapped_func
```

```
@timefun
def foo(a, b):
    print(a+b)

foo(3,5)
sleep(2)
foo(2,4)
```

例 3:被装饰的函数有不定长参数

```
def set_func(func):
   print("---开始进行装饰")
   def call_func(*args, **kwargs):
      print("---这是权限验证 1----")
      print("---这是权限验证 2----")
      # func(args, kwargs)  # 不行，相当于传递了 2 个参数 ：1 个元组，1 个字典
      func(*args, **kwargs)  # 拆包的过程
   return call_func


@set_func  # 相当于 test1 = set_func(test1)
def test1(num, *args, **kwargs):
   print("-----test1----%d" % num)
   print("-----test1----" , args)
   print("-----test1----" , kwargs)


test1(100)
test1(100, 200)
test1(100, 200, 300, mm=100)
```

采用 *args, **kwargs 参数设计，无论是有参数，无参数，或者多个参数，均能实现传递

例 4:装饰器中的 return

```
from time import ctime, sleep

def timefun(func):
    def wrapped_func():
        print("%s called at %s" % (func.__name__, ctime()))
```

```
        func()
    return wrapped_func

@timefun
def foo():
    print("I am foo")

@timefun
def getInfo():
    return '----hahah---'

foo()
sleep(1)
foo()
ret=getInfo()
print(ret)
```

执行结果:

```
foo called at Fri Nov  4 21:55:35 2016
I am foo
foo called at Fri Nov  4 21:55:37 2016
I am foo
getInfo called at Fri Nov  4 21:55:37 2016
None
```

如果修改装饰器为 return func()，则运行结果：

```
foo called at Fri Nov  4 21:55:57 2016
I am foo
foo called at Fri Nov  4 21:55:59 2016
I am foo
getInfo called at Fri Nov  4 21:55:59 2016
----hahah---
```

总结：

- **一般情况下为了让装饰器更通用，加上 return**

例 5:装饰器带参数,在原有装饰器的基础上，设置外部变量

```
#decorator2.py

from time import ctime, sleep
```

```
def timefun_arg(pre="hello"):
    def timefun(func):
        def wrapped_func():
            print("%s called at %s %s" % (func.__name__, ctime(), pre))
            return func()
        return wrapped_func
    return timefun

# 下面的装饰过程
# 1. 调用 timefun_arg("wangdao")
# 2. 将步骤 1 得到的返回值，即 time_fun 返回， 然后 time_fun(foo)
# 3. 将 time_fun(foo)的结果返回，即 wrapped_func
# 4. 让 foo = wrapped_fun，即 foo 现在指向 wrapped_func
@timefun_arg("wangdao")
def foo():
    print("I am foo")

@timefun_arg("python")
def too():
    print("I am too")

foo()
sleep(2)
foo()

too()
sleep(2)
too()
```

可以理解为

```
foo()==timefun_arg("wangdao")(foo)()
```

不是很理解？我们来看一个场景，假如 test1 和 test2 函数需要不同的权限验证

```
def set_func(func):

    def call_func(*args, **kwargs):

            level = args[0]

            if level == 1:

                print("----权限级别 1，验证----")

            elif level == 2:

                print("----权限级别 2，验证----")

            return func()
```

```
        return call_func



@set_func
def test1():
        print("-----test1---")
        return "ok"

@set_func
def test2():
        print("-----test2---")
        return "ok"

# 这种方式不好：
# 1. 如果 test1 之前被调用了 N 次，那么就需要修改 N 个
# 2. 调用函数时，验证的级别应该是函数定义的开发者设定
#    而不是调用者设定
test1(1)
test2(2)
```

上面的手法是不合适的，那如何做才是合理的呢？

```
def set_level(level_num):
        def set_func(func):
                def call_func(*args, **kwargs):
                        if level_num == 1:
                                print("----权限级别 1，验证----")
                        elif level_num == 2:
                                print("----权限级别 2，验证----")
```

```
                return func()
            return call_func
        return set_func


# 带有参数的装饰器装饰过程分为 2 步：
# 1. 调用 set_level 函数，把 1 当做实参
# 2. set_level 返回一个装饰器的引用，即 set_func
# 3. 用返回的 set_func 对 test1 函数进行装饰（装饰过程与之前一样）
@set_level(1)
def test1():
    print("-----test1---")
    return "ok"


@set_level(2)
def test2():
    print("-----test2---")
    return "ok"


test1()
test2()
```

例 6：类装饰器（了解）

装饰器函数其实是这样一个接口约束，它必须接受一个 callable 对象作为参数，然后返回一个 callable 对象。在 Python 中一般 callable 对象都是函数，但也有例外。只要某个对象重写了 `__call__()` 方法，那么这个对象就是 callable 的。

```
class Test():
    def __call__(self):
        print('call me!')

t = Test()
t()  # call me
```

类装饰器 demo

```
class Test(object):
    def __init__(self, func):
        print("---初始化---")
        print("func name is %s"%func.__name__)
        self.__func = func
    def __call__(self):
        print("---装饰器中的功能---")
        self.__func()
#说明：
#1. 当用 Test 来装作装饰器对 test 函数进行装饰的时候，首先会创建 Test 的实例对象
#    并且会把 test 这个函数名当做参数传递到__init__方法中
#    即在__init__方法中的属性__func 指向了 test 指向的函数
#
#2. test 指向了用 Test 创建出来的实例对象
#
#3. 当在使用 test()进行调用时，就相当于让这个对象()，因此会调用这个对象的__call__方法
#
#4. 为了能够在__call__方法中调用原来 test 指向的函数体，所以在__init__方法中就需要一个实例属性来保存这个函数体的引用
#    所以才有了 self.__func = func 这句代码，从而在调用__call__方法中能够调用到 test 之前的函数体
@Test   # 相当于 test = Test(test)
def test():
    print("----test---")
test()#如果把这句话注释，重新运行程序，依然会看到"--初始化--"
```

运行结果如下：

```
---初始化---
func name is test
---装饰器中的功能---
----test---
```

例 7 增加 warps 的作用

```
# ! /usr/bin/python3

#warps 的作用是让函数显示自己的名字和备注


from functools import wraps
```

```
def my_decorator(func):
    def wper(*args, **kwargs):
        '''decorator'''
        print('Calling decorated function...')
        return func(*args, **kwargs)
    return wper

@my_decorator
def example():
    """Docstring"""
    print('Called example function')


print(example.__name__, example.__doc__)#wper decorator

from functools import wraps


def my_decorator(func):
    @wraps(func)
    def wper(*args, **kwargs):
        '''decorator'''
        print('Calling decorated function...')
        return func(*args, **kwargs)

    return wper


@my_decorator
def example():
    """Docstring"""
    print('Called example function')


print(example.__name__, example.__doc__)  # example Docstring
```