第8章 python装饰器

装饰器本质上是一个Python函数，它可以让其他函数在不需要做任何代码变动的前提下增加额外功能，装饰器的返回值也是一个函数对象。它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量与函数功能本身无关的雷同代码并继续重用。

本章将从以下方面来介绍Python中的装饰器：

- 函数装饰器
- 类装饰器
- 内置装饰器
- 装饰器链
- 需要注意的地方

1. 函数装饰器

1.1 简单装饰器

[2]
```
def use_logging(func):

    def wrapper():
        print('这是一个简单的装饰器')
        return func()    # 把 foo 当做参数传递进来时，执行func()就相当于执行foo()
    return wrapper

def foo():
    print('i am foo')

foo = use_logging(foo)  # 因为装饰器 use_logging(foo) 返回的是函数对象 wrapper，这条语句相当于  foo = wrapper
foo()                   # 执行foo()就相当于执行 wrapper()
```

```
这是一个简单的装饰器
i am foo
```

1.2 @语法糖

@ 符号就是装饰器的语法糖，它放在函数开始定义的地方，这样就可以省略最后一步再次赋值的操作。

[3]
```
def use_logging(func):

    def wrapper():
        print('这是一个简单的装饰器')
        return func()    # 把 foo 当做参数传递进来时，执行func()就相当于执行foo()
    return wrapper

@use_logging
def foo():
    print('i am foo')

foo()                    # 执行foo()就相当于执行 wrapper()
```

```
这是一个简单的装饰器
i am foo
```

如上所示，有了 @ ，我们就可以省去foo = use_logging(foo)这一句了，直接调用 foo() 即可得到想要的结果。foo() 函数不需要做任何修改，只需在定义的地方加上装饰器，调用的时候还是和以前一样，如果我们有其他的类似函数，我们可以继续调用装饰器来修饰函数，而不用重复修改函数或者增加新的封装。这样，我们就提高了程序的可重复利用性，并增加了程序的可读性。

1.3 *args、**kwargs

如果我的业务逻辑函数 foo 需要参数该怎么办？比如：

[4]
```
def foo(name):
    print("i am %s" % name)
```

这时，我们可以在定义wrapper函数的时候指定参数。并且当装饰器不知道 foo 到底有多少个参数时，我们可以用 *args 来代替

```
[ ]  def wrapper(*args):
         print('test')
         return func(*args)
     return wrapper
```

如此一来，不管foo定义了多少个参数，我们都可以完整地传递到 func 中去。这样就不影响 foo 的业务逻辑了。这时你可能还有疑问，如果 foo 函数还定义了一些关键字参数呢？比如：

```
[ ]  def foo(name, age=None, height=None):
         print("I am %s, age %s, height %s" % (name, age, height))
```

这时，我们就可以把 wrapper 函数指定关键字函数：

```
[ ]  def wrapper(*args, **kwargs):
         # args是一个数组，kwargs一个字典
         print('test')
         return func(*args, **kwargs)
     return wrapper
```

1.4 类方法的函数装饰器

```
[4]  import time

     def decorator(func):
         def wrapper(me_instance):
             start_time = time.time()
             func(me_instance)
             end_time = time.time()
             print(end_time - start_time)
         return wrapper

     class Method(object):

         @decorator
         def func(self):
```

```
            time.sleep(0.8)

p1 = Method()
p1.func() #函数调用
```

```
0.8001229763031006
```

对于类方法来说，都会有一个默认的参数self，它实际表示的是类的一个实例，所以在装饰器的内部函数wrapper也要传入一个参数me_instance就表示将类的实例p1传给wrapper，其他的用法都和函数装饰器相同。

2. 类装饰器

装饰器不仅可以是函数，还可以是类，相比函数装饰器，类装饰器具有灵活度大、高内聚、封装性等优点。使用类装饰器主要依靠类的__call__方法，当使用 @ 形式将装饰器附加到函数上时，就会调用此方法。

[5]
```
class Decorator(object):
    def __init__(self,f):
        self.f = f
    def __call__(self):  # __call__是一个特殊方法，它可将一个类实例变成一个可调用对象
        print("decorator start")
        self.f()
        print("decorator end")

@Decorator
def func():
    print("func")

func()
```

```
decorator start
func
decorator end
```

要使用类装饰器必须实现类中的__call__()方法，就相当于将实例变成了一个方法。

3. 内置装饰器

python常用内置装饰器有属性(property)，类方法(classmethod)，静态方法(staticmethod)

3.1 @property

使调用类中的方法像引用类中的字段属性一样

[17]
```
class TestClass:
    name = 'test'
    def __init__(self,name):
        self.name = name

    @property
    def sayHello(self):
        print("hello",self.name)

cls = TestClass('Tom')
print('通过实例引用属性')
print(cls.name)
print('像引用属性一样调用@property修饰的方法')
cls.sayHello
```

```
通过实例引用属性
Tom
像引用属性一样调用@property修饰的方法
hello Tom
```

下面再引用一个例子来介绍property的更常用的用法：

[23]
```
# 假设现在有个矩形，可以设置其宽、高。
# 如果需要面积，我们可以使用类似于area()的方法来计算。
# 如果需要周长，我们可以使用类似于perimeter()的方法来计算。
# 对于Python这种动态语言来说，可以用perperty()加以包装：
class Rectangle(object):
    def __init__ (self,width,height):
        self.__width = width
        self.__height = height

    def get_width(self):
        return self.__width
    def set_width(self, size):
        self.__width= size
    width=property(get_width,set_width)

    def get_height(self):
        return self.__height
```

```
    def set_height(self,size):
        self.__height=size
    height=property(get_height,set_height)

    def area(self):
        return self.width*self.height
    area=property(area)


    def perimeter(self):
        return (self.width+self.height)*2
    perimeter=property(perimeter)
# 这样，就使用了property()函数包装出了width、height、area、perimeter四个特性

rect=Rectangle(3,4)
rect.width=5
rect.height=6
print(rect.width)
print(rect.height)
print(rect.area)
print(rect.perimeter)
```

```
5
6
30
22
```

有了装饰器语法，以上代码可以简化为：

[25]
```
class Rectangle(object):
    def __init__ (self,width,height):
        self.__width = width
        self.__height = height

    @property
    def width(self):
        return self.__width

    @width.setter
    def width(self, size):
        self.__width= size

    @property
    def height(self):
        return self.__height

    @height.setter
    def height(self,size):
```

```
            self.__height=size

        @property
        def area(self):
            return self.width*self.height

        @property
        def perimeter(self):
            return (self.width+self.height)*2
    rect=Rectangle(3,4)
    rect.width=5
    rect.height=6
    print(rect.width)
    print(rect.height)
    print(rect.area)
    print(rect.perimeter)
```

```
5
6
30
22
```

3.2 @staticmethod

将类中的方法修饰为静态方法，即类不需要创建实例的情况下，可以通过类名直接引用。到达将函数功能与实例解绑的效果。

[20]
```
class TestClass:
    name = 'test'
    def __init__(self,name):
        self.name = name

    @staticmethod
    def fun(self,x,y):
        return x+y

cls = TestClass('Tom')
print('通过实例引用方法')
print(cls.fun(None,2,3)) # 参数个数必须与定义中的个数保持一致，否则报错
print('类名直接引用静态方法')
print(TestClass.fun(None,2,3)) # 参数个数必须与定义中的个数保持一致，否则报错
```

```
通过实例引用方法
5
```

```
类名直接引用静态方法
5
```

3.3 @classmethod

类方法的第一个参数是一个类，是将类本身作为操作的方法。类方法被哪个类调用，就传入哪个类作为第一个参数进行操作。

[21]
```
class Car(object):
    car = 'audi'

    @classmethod
    def value(self,category): # 可定义多个参数，但第一个参数为类本身
        print('%s car of %s'%(category,self.car))

class BMW(Car):
    car = 'BMW'

class Benz(Car):
    car = 'Benz'

print('通过实例调用')
baoma = BMW()
baoma.value('Normal')

print('通过类名直接调用')
Benz.value('SUV')
```

```
通过实例调用
Normal car of BMW
通过类名直接调用
SUV car of Benz
```

4. 装饰器链

一个Python函数可以被多个装饰器修饰，若有多个装饰器时，它的执行顺序是从近到远依次执行。

[21]
```
def makebold(f):
    return lambda: '<b>' + f() + '<b>'
```

```
    def makeitalic(f):
        return lambda: '<i>' + f() + '<i>'

    @makebold
    @makeitalic
    def say():
        return 'Hello'

    print(say())
```

```
<b><i>Hello<i><b>
```

5. 需要注意的地方

5.1 用functools.wraps装饰内层函数

使用装饰器极大地复用了代码，但是他有一个缺点就是原函数的元信息不见了，比如函数的docstring、__name__、参数列表，如下面例子所示：

[27]
```
# 装饰器
def logged(func):
    def with_logging(*args, **kwargs):
        print(func.__name__)      # 输出 'with_logging'
        print(func.__doc__)       # 输出 None
        return func(*args, **kwargs)
    return with_logging

# 函数
@logged
def f(x):
   """does some math"""
   return x + x * x

logged(f)
```

```
<function __main__.logged.<locals>.with_logging>
```

不难发现，函数 f 被with_logging取代了，当然它的docstring，__name__就是变成了with_logging函数的信息了。好在我们有functools.wraps，wraps本身也是一个装饰器，

它能把原函数的元信息拷贝到装饰器里面的 func 函数中，这使得装饰器里面的 func 函数也有和原函数 foo 一样的元信息了。

[28]
```
from functools import wraps
def logged(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print(func.__name__)      # 输出 'f'
        print(func.__doc__)
        return func(*args, **kwargs)
    return with_logging

@logged
def f(x):
   """does some math"""
   return x + x * x
logged(f)
```

```
<function __main__.f>
```

5.2 修改外层变量时记得使用nonlocal

装饰器是对函数对象的一个高级应用。在编写装饰器的过程中，你会经常碰到内层函数需要修改外层函数变量的情况。就像下面这个装饰器一样：

[29]
```
import functools

def counter(func):
    """装饰器：记录并打印调用次数"""
    count = 0
    @functools.wraps(func)
    def decorated(*args,**kwargs):
        # 次数累加
        count += 1
        print(f"Count:{count}")
        return func(*args,**kwargs)
    return decorated

@counter
def foo():
    pass
foo()
```

```
---------------------------------------------------------------------------
---
UnboundLocalError                         Traceback (most recent call
last)
<ipython-input-29-480edbf23e93> in <module>()
     15 def foo():
     16     pass
---> 17 foo()

<ipython-input-29-480edbf23e93> in decorated(*args, **kwargs)
      7     def decorated(*args,**kwargs):
      8         # 次数累加
----> 9         count += 1
     10         print(f"Count:{count}")
     11         return func(*args,**kwargs)

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

为了统计函数调用次数，我们需要在 decorated 函数内部修改外层函数定义的 count 变量的值。但是，上面这段代码是有问题的，在执行它时解释器会报错。

这个错误是由 counter 与 decorated 函数互相嵌套的作用域引起的。当解释器执行到 count+=1 时，并不知道 count 是一个在外层作用域定义的变量，它把 count 当做一个局部变量，并在当前作用域内查找。最终却没有找到有关 count 变量的任何定义，然后抛出错误。

为了解决这个问题，我们需要通过 nonlocal 关键字告诉解释器：“count 变量并不属于当前的 local 作用域，去外面找找吧”，之前的错误就可以得到解决。

[31]
```python
import functools

def counter(func):
    """装饰器：记录并打印调用次数"""
    count = 0
    @functools.wraps(func)
    def decorated(*args,**kwargs):
        # 次数累加
        nonlocal count
        count += 1
        print(f"Count:{count}")
        return func(*args,**kwargs)
    return decorated

@counter
def foo():
```

```
        pass
foo()
```

```
Count:1
```

```
[ ]
```