9 装饰器

> AI领域中的Python开发 — by 丁宁

SIGAI课程录制

- 上节课：闭包，再谈变量作用域，详解`nonlocal`
- 本节课：装饰器的**Why? What? How?**

AI学习与实践平台

www.sigai.cn

概述

- 理解装饰器要从三方面入手：**Why? What? How?**
- 学习装饰器要从模仿开始

Why? 为什么会出现装饰器这个东西?

- 名称管理
- 显示调用
- 就近原则
- 充分复用

不引人多余变量名

- 有时候，写一个闭包，仅仅是为了增强一个函数的功能
- 功能增强完了之后，只对增强了功能的最终函数感兴趣
- 装饰之前的函数引用就变得多余
- 因而出现了`func = decorated_by(func)`这种即席覆盖的写法

看一个例子：修改一个函数的`help`命令的帮助文档

```
def decorate(func):
    func.__doc__ += '\nDecorated by decorate.'
    return func

def add(x, y):
    '''Return the sum of x and y.'''
    return x + y

decorated_add = decorate(add)
```

分析：

- 引入了新的变量名：`decorated_add`
- 原来的`add`函数一般情况下在后面的程序中不会再用到了

改进版一：

```
def decorate(func):
    func.__doc__ += '\nDecorated by decorate.'
    return func

def add(x, y):
    '''Return the sum of x and y.'''
    return x + y

add = decorate(add)
```

分析：

- 没有新的变量名的引入，变量名使用效率提升
- 装饰函数的执行代码需要单独调用，可能不符合就近原则

显示调用，就近原则

- 装饰的次数多了，这种方式依旧显得有些多余

- 而且会带来新的问题：在哪写这句代码比较好?
- 为了编码更加优雅，保持显示调用，遵守就近原则，出现了`@`这个语法糖

改进版二：`@`语法糖的横空出世

```
def decorate(func):
    func.__doc__ += '\nDecorated by decorate.'
    return func

@decorate
def add(x, y):
    '''Return the sum of x and y.'''
    return x + y
```

分析：

- 通过语法糖，保证了装饰过程与原函数彼此之间的独立性
- 同时，还保证了两者代码之间的就近原则，形成一个有机的整体
- 问题：装饰定义函数与被装饰函数在同一个模块中实现，影响了复用效率

分层封装，充分复用

改进版三：装饰器被单独封装在一个模块中：

`DecorateToolBox.py`

```
class DecorateToolBox:

    @classmethod
    def decorate(self, func):
        func.__doc__ += '\nDecorated by decorate.'
        return func
```

`test.py`

```
from DecorateToolBox import DecorateToolBox

@DecorateToolBox.decorate
def add(x, y):
    '''Return the sum of x and y.'''
    return x + y
```

启动Python解释器：

```
>>> from test import add
>>> help(add)
```

解释器进入如下画面：

```
Help on function add in module test:

add(x, y)
    Return the sum of x and y.
    Decorated by decorate.
(END)
```

分析：

- 将不同级别的功能模块封装在不同的文件中，是编写大型程序的基础
- 实现一个装饰器，并不一定需要写出一个闭包
- 类可以不依赖于一个实体而直接调用其方法，得益于`@classmethod`装饰器

总结

- 避免重复，充分复用：装饰器的模块化使程序设计者完美的避免重复的前置和收尾代码
- 显示调用，就近原则：装饰器是显式的，并且在需要装饰器的函数中即席使用

What? 什么是装饰器?

- 装饰器是一个可调用的**对象**，以某种方式**增强**函数的功能
- 装饰器是一个**语法糖**，在**源码**中标记函数（此源码指编译后的源码）
- 解释器解析源码的时候将被装饰的函数作为第一个位置参数传给装饰器
- 装饰器可能会**直接处理**被装饰函数，然后返回它（一般仅修改属性，不修改代码）
- 装饰器也可能用一个新的函数或可调用对象**替换**被装饰函数（但核心功能一般不变）
- 装饰器**仅仅看着像闭包**，其实功能的定位与闭包**有重合也有很大区别**
- 装饰器模式的本质是**元编程**：在**运行时**改变程序行为
- 装饰器的一个不可忽视的特性：**在模块加载时立即执行**
- 装饰器是可以**堆叠**的，**自底向上**逐个装饰
- 装饰器是可以**带参数**的，但此时至少要写**两个**装饰器
- 装饰器的更加**Pythonic**的实现方式其实是在类中实现`__call__()`方法

装饰器的堆叠

```
def deco_1(func):
    print("running deco_1...")
```

```
    return func

def deco_2(func):
    print("running deco_2...")
    return func

@deco_1
@deco_2
def f():
    print("running f...")

if __name__ == '__main__':

    f()
```

运行结果:

```
running deco_2...
running deco_1...
running f...
```

装饰器在导入时立即执行

```
def deco_1(func):
    print("running deco_1...")
    return func

def deco_2(func):
    print("running deco_2...")
    return func

@deco_1
@deco_2
def f():
    print("running f...")

if __name__ == '__main__':

    pass
```

运行结果:

```
running deco_2...
```

```
running deco_1...
```

带参数的装饰器

既然装饰器只能接受一个位置参数，并且是被动的接受解释器传过来的函数引用，那么如何实现带参数的装饰器呢?
问题分析:

1. 限制条件一：装饰器本身只能接受一个位置参数
2. 限制条件二：这个位置参数已经被被装饰函数的引用占据了
3. 问题目标：希望装饰器能够使用外部传入的其他参数
4. 推论：装饰器需要访问或修改外部参数

三种备选方案:

1. 在装饰器内访问全局不可变对象，若需要修改，则使用`global`声明（不安全）
2. 在装饰器内访问外部可变对象（不安全）
3. 让装饰器成为闭包的返回（较安全）

方案：编写一个闭包，接受外部参数，返回一个装饰器

先来看一下参数化之前的装饰器:

```
registry = set()
def register(func):
    registry.add(func)
    return func

@register
def f1():
    print("running f1.")

@register
def f2():
    print("running f2.")

def f3():
    print("running f3.")

def main():
    f1()
    f2()
    f3()
```

```
if __name__ == '__main__':

    print(registry)
    main()
```

输出结果如下：

```
{<function f1 at 0x103906488>, <function f2 at 0x103906598>}
running f1.
running f2.
running f3.
```

参数化之后的装饰器，增加了开关功能：

```
registry = set()

def register(flag=True):
    def decorate(func):
        if flag:
            registry.add(func)
        else:
            registry.discard(func)

        return func
    return decorate

@register()
def f1():
    print("running f1.")

@register(False)
def f2():
    print("running f2.")

@register(True)
def f3():
    print("running f3.")

def main():
    f1()
    f2()
    f3()
```

```
if __name__ == '__main__':

    print(registry)
    main()
```

输出结果如下：

```
{<function f1 at 0x103a06620>, <function f3 at 0x103a06730>}
running f1.
running f2.
running f3.
```

分析：

- 此时，`register`变量被使用了两次
- 第一次是后面的调用：`()`（调用之后才变成一个装饰器）
- 第二次是前面的装饰：`@`（装饰器符合仅能用于装饰器）

> 注意：`register`不是装饰器；`register()`或`register(False)`才是

How? 装饰器怎么用

- 从模仿开始

装饰器的常见使用场景

- 运行前处理：如确认用户授权
- 运行时注册：如注册信号系统
- 运行后清理：如序列化返回值

注册机制或授权机制（往往跟应用开发相关）

- 函数的注册，参考上面的例子
- 将某个功能注册到某个地方，比如**Flask**框架中**URL**的注册
- 比如验证身份信息或加密信息，以确定是否继续后续的运算或操作
- 比如查询一个身份信息是否已经被注册过，如果没有则注册，如果有则直接返回账户信息

参数的数据验证或清洗（往往跟数据清洗或异常处理相关）

我们可以强行对输入参数进行特殊限制：

```
def require_ints(func):
    def temp_func(*args):
        if not all([isinstance(arg, int) for arg in args]):
            raise TypeError("{} only accepts integers as argument
s.".format(func.__name__))
        return func(*args)
    return temp_func

def add(x,y):
    return x + y

@require_ints
def require_ints_add(x,y):
    return x + y

if __name__ == '__main__':
    print(add(1.0, 2.0))
    print(require_ints_add(1.0, 2.0))
```

运行结果如下：

```
3.0
Traceback (most recent call last):
  File "test.py", line 17, in <module>
    print(require_ints_add(1.0, 2.0))
  File "test.py", line 4, in temp_func
    raise TypeError("{} only accepts integers as arguments.".format
(func.__name__))
TypeError: require_ints_add only accepts integers as arguments.
```

复用核心计算模块，仅改变输出方式

让原本返回Python原生数据结构的函数变成输出JSON结构：

```
import json

def json_output(func):
    def temp_func(*args, **kw):
        result = func(*args, **kw)
        return json.dumps(result)
    return temp_func

def generate_a_dict(x):
```

```
    return {str(i): i**2 for i in range(x)}

@json_output
def generate_a_dict_json_output(x):
    return {str(i): i**2 for i in range(x)}

if __name__ == '__main__':
    a, b = generate_a_dict(5), generate_a_dict_json_output(5)
    print(a, type(a))
    print(b, type(b))
```

运行结果如下：

```
{'0': 0, '1': 1, '2': 4, '3': 9, '4': 16} <class 'dict'>
{"0": 0, "1": 1, "2": 4, "3": 9, "4": 16} <class 'str'>
```

总结

- 装饰器作为Python四神器之一，功能非常强大（还有迭代器，生成器，上下文管理器）
- 装饰器用好了能让程序复用性大大提高，代码也会变的非常优雅
- 要理解装饰器，需要先理解一等函数，变量作用域，函数式编程，闭包
- 要用好装饰器，要从模仿开始，多读别人的代码，多模仿，慢慢就有感觉了
- Python内置了很多的装饰器，功能都很强大，编写框架或大型程序的时候你一定会用到