9.12 使用装饰器扩充类的功能¶

问题¶

你想通过反省或者重写类定义的某部分来修改它的行为，但是你又不希望使用继承或元类的方式。

解决方案¶

这种情况可能是类装饰器最好的使用场景了。例如，下面是一个重写了特殊方法 `__getattribute__` 的类装饰器， 可以打印日志：

```
def log_getattribute(cls):
    # Get the original implementation
    orig_getattribute = cls.__getattribute__

    # Make a newdefinition
    def new_getattribute(self, name):
        print('getting:', name)
        return orig_getattribute(self, name)

    # Attach to the class and return
    cls.__getattribute__ = new_getattribute
    return cls

# Example use
@log_getattribute
class A:
    def __init__(self,x):
        self.x = x
    def spam(self):
        pass
```

下面是使用效果：

```
>>> a = A(42)
>>> a.x
getting: x
42
>>> a.spam()
getting: spam
>>>
```

讨论¶

类装饰器通常可以作为其他高级技术比如混入或元类的一种非常简洁的替代方案。 比如，上面示例中的另外一种实现使用到继承：

```
class LoggedGetattribute:
    def __getattribute__(self, name):
        print('getting:', name)
        return super().__getattribute__(name)

# Example:
class A(LoggedGetattribute):
```

```
class A(LoggedGetattribute):
    def __init__(self,x):
        self.x = x
    def spam(self):
        pass
```

这种方案也行得通，但是为了去理解它，你就必须知道方法调用顺序、`super()`以及其它8.7小节介绍的继承知识。 某种程度上来讲，类装饰器方案就显得更加直观，并且它不会引入新的继承体系。它的运行速度也更快一些， 因为他并不依赖 `super()` 函数。

如果你系想在一个类上面使用多个类装饰器，那么就需要注意下顺序问题。 例如，一个装饰器A会将其装饰的方法完整替换成另一种实现， 而另一个装饰器B只是简单的在其装饰的方法中添加点额外逻辑。 那么这时候装饰器A就需要放在装饰器B的前面。

你还可以回顾一下8.13小节另外一个关于类装饰器的有用的例子。