装饰设计模式

尚硅谷 宋红康

回想一下 Java 当中的各种输入输出流，各种功能一层嵌套一层，就好像不断得给一个产品加功能，加完以后在消费者看来，原来是是什么产品现在还是什么产品，只不过用的时候功能增加了。

1.装饰模式的概念:

- 装饰模式是动态的给一个对象添加一些额外的功能，就增加功能来说，装饰模式比生成子类更为灵活。
- 装饰模式是在不必改变原类文件和使用继承的情况下，动态的扩展一个对象的功能。提供比继承更多的灵活性。
- 装饰模式是创建一个包装对象，也就是使用装饰来包裹真实的对象。

2.装饰模式的实现方式

1. 装饰对象和真实对象有相同的接口/抽象类。这样客户端对象就能以和真实对象相同的方式和装饰对象交互。
2. 装饰对象包含一个真实对象的引用（reference）。
3. 装饰对象接受所有来自客户端的请求。它把这些请求转发给真实的对象。
4. 装饰对象可以在转发这些请求以前或以后增加一些附加功能。这样就确保了在运行时，不用修改给定对象的结构就可以在外部增加附加的功能。在面向对象的设计中，通常是通过继承来实现对给定类的功能扩展。

3.适用性

1. 需要扩展一个类的功能，或给一个类添加附加职责。

2. 需要动态的给一个对象添加功能，这些功能可以再动态的撤销。

3. 需要增加由一些基本功能的排列组合而产生的非常大量的功能，从而使继承关系变的不现实。

4. 当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

4. 代码实现

```
/*
 * 变形金刚在变形前是一辆汽车，它可以在陆地上移动。当它变成机器人之后除了能够在陆地上移动之外，还可以说话；
 * 如果需要，它还可以变成飞机，除了在陆地上移动还可以在天空中飞翔。
 */
/**
 *
 * @Description 声明一个move方法，无论变形金刚如何改变该方法始终都有，是具体构件和抽象装饰类共有的方法。
 * @author shkstart Email:shkstart@126.com
 *
 */
interface Transform {
    public void move();
}

/**
 *
 * @Description ConcreteComponent（具体构件）：Car.java 提供了move方法的实现，
 *              运用构造函数初始化输出当前状态，它是一个可以被装饰的类。在这里Car
 *              被声明为final类型，说明不能通过继承来拓展其功能，需运用类之间的关
 *              联关系来拓展。即装饰器装饰
 * @author shkstart Email:shkstart@126.com
 *
 */
final class Car implements Transform {

    // 初始化
```

```
    public Car() {
        System.out.println("变形金刚->车");
    }

    @Override
    public void move() {
        System.out.println("在陆地上移动");
    }
}

/**
 * @Description Decorator（抽象装饰类）：Changer.java  定义一个抽象构件类型的transform，通过构造函数或者setter方法来给该对象赋值，同时也通过调用transform对象来实现move方法，这样可以保证原方法不被丢失，而且可以在它的子类中增加新的方法，拓展原有功能。
 * @author shkstart Email:shkstart@126.com
 *
 */
class Changer implements Transform {

    private Transform transform;

    public Changer(Transform transform) {
        this.transform = transform;
    }

    @Override
    public void move() {
        transform.move();

    }

}

/**
 *
 * @Description ConcreteDecorator（具体装饰类）：这里采用的是半透明
 * @author shkstart Email:shkstart@126.com
 * @version
 * @date 2019年1月31日下午4:08:18
 *
 */
```

```
class Robot extends Changer {

    public Robot(Transform transform) {
        super(transform);
        System.out.println("->机器人");
    }

    @Override
    public void move() {
        super.move();
        say();
    }
    private void say() {
        System.out.println("说话");
    }

}

class Airplane extends Changer {

    public Airplane(Transform transform) {
        super(transform);
        System.out.println("->飞机");
    }

    @Override
    public void move() {
        super.move();
        fly();
    }

    private void fly() {
        System.out.println("飞翔");
    }

}

public class DecoratorDemo {

    public static void main(String[] args) {
        Transform machine = new Car();
        machine.move();
```

```
        Robot robot = new Robot(machine);
        robot.move();

        Airplane airplane1 = new Airplane(machine);
        airplane1.move();

        Airplane airplane2 = new Airplane(robot);
        airplane2.move();

    }

}
```

5.装饰器模式的应用场景

1、需要扩展一个类的功能。

2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

6.缺点

1. 这种比继承更加灵活机动的特性，也同时意味着更加多的复杂性。
2. 装饰模式会导致设计中出现许多小类，如果过度使用，会使程序变得很复杂。
3. 装饰模式是针对抽象组件（Component）类型编程。但是，如果你要针对具体组件编程时，就应该重新思考你的应用架构，以及装饰者是否合适。当然也可以改变 Component 接口，增加新的公开的行为，实现“半透明”的装饰者模式。在实际项目中要做出最佳选择。

7. 装饰模式与代理模式的对比

装饰模式：

- 在不改变接口的前提下，动态扩展对象的访问。
- 动态继承，让类具有在运行期改变行为的能力。
- 装饰模式，突出的是运行期增加行为，这和继承是不同的，继承是在编译期增加行为。
- 强调：增强

代理模式：

- 在不改变接口的前提下，控制对象的访问。
- 从封装的角度讲，是为了解决类与类之间相互调用而由此导致的耦合关系，可以说是接口的另外一个层引用。比如：在 a 类->b 代理->c 类这个关系中，c 类的一切行为都隐藏在 b 中。即调用者不知道要访问的内容与代理了什么对象。
- 从复用的角度讲，可以解决不同类调用一个复杂类时，仅仅因较小的改变而导致整个复杂类新建一个类。比如：a 类->c 类 1；b 类->c 类 2。
- 可以变为 a 类->ca 代理类->c 类；b 类->cb 代理类-c 类。
- 代理模式，是类之间的封装和（某方面的）复用。
- 强调：限制