链滴

《Head First 设计模式》：装饰者模式

作者：jingqueyimu

原文链接：https://ld246.com/article/1594649223284

来源网站：链滴

正文

一、定义

装饰者模式动态地将责任（功能）附加到对象上。若要扩展功能，装饰者提供了比继承更有弹性的替方案。

要点：

- 装饰者和被装饰者有相同的超类型。
- 可以用一个或多个装饰者包装一个对象。
- 既然装饰者和被装饰者有相同的超类型，所以在任何需要原始对象（被装饰者）的场合，都可以用饰过的对象代替它。
- 装饰者可以在被装饰者的行为之前与/或之后，加上自己的行为，甚至将被装饰者的行为整个取代，以到达特定的目的。
- 对象可以在任何时候被装饰，所以可以在运行时动态地、不限量地用装饰者装饰对象。
- 装饰者会导致设计中出现许多小对象，如果过度使用，会让程序变得很复杂。

二、实现步骤

1、创建组件接口

装饰者和被装饰者都必须实现组件接口。

也可以用组件抽象类，然后让装饰者和被装饰者继承组件抽象类，只要装饰者和被装饰者具有相同的类型即可。

```
/**
 * 组件接口（装饰者和被装饰者都必须实现该接口）
 */
public interface Component {

    public void doSomething();
}
```

2、创建具体的组件，并实现组件接口

```
/**
 * 具体组件（被装饰者）
 */
public class ConcreteComponent implements Component {

    @Override
    public void doSomething() {
        System.out.println("ConcreteComponent do something...");
    }
}
```

3、创建装饰者抽象类，并实现组件接口

如果只有一个装饰者，也可以不创建装饰者抽象类，而是由具体的装饰者直接实现组件接口。

```
/**
 * 组件装饰者抽象类
 */
public abstract class ComponentDecorator implements Component {

    protected Component component;

    public ComponentDecorator(Component component) {
        // 通过构造传入组件（被装饰者）
        this.component = component;
    }

    @Override
    public void doSomething() {
        // 委托给组件（被装饰者）
        component.doSomething();
    }
}
```

4、创建具体的装饰者，并继承装饰者抽象类

（1）装饰者 A

```
/**
 * 装饰者A
 */
public class ComponentDecoratorA extends ComponentDecorator {
```

```
    public ComponentDecoratorA(Component component) {
        super(component);
    }

    @Override
    public void doSomething() {
        // 装饰者添加自己的业务代码

        component.doSomething();

        // 装饰者添加自己的业务代码
        System.out.println("ComponentDecoratorA do something...");
    }
}
```

（2）装饰者 B

```
/**
 * 装饰者B
 */
public class ComponentDecoratorB extends ComponentDecorator {

    public ComponentDecoratorB(Component component) {
        super(component);
    }

    @Override
    public void doSomething() {
        // 装饰者添加自己的业务代码

        component.doSomething();

        // 装饰者添加自己的业务代码
        System.out.println("ComponentDecoratorB do something...");
    }
}
```

5、使用装饰者装饰组件

```
public class Test {

    public static void main(String[] args) {
        // 具体组件（被装饰者）
        Component component = new ConcreteComponent();
        // 用装饰者A装饰组件
        ComponentDecorator componentDecoratorA = new ComponentDecoratorA(component)

        // 用装饰者B装饰组件
        ComponentDecorator componentDecoratorB = new ComponentDecoratorB(component)


        component.doSomething();
```

```
        componentDecoratorA.doSomething();
        componentDecoratorB.doSomething();
    }
}
```

三、举个栗子

1、背景

星巴兹是以扩张速度最快而闻名的咖啡连锁店。因为扩张速度实在太快了，他们准备更新订单系统，合乎他们的饮料供应要求——

顾客在购买咖啡时，可以要求在其中加入各种调料，例如：蒸奶、豆浆、摩卡（巧克力风味）或覆盖泡。星巴兹会根据所加入的调料收取不同的费用。所以订单系统必须考虑到这些调料部分。

2、实现

把调料理解为饮料装饰者，然后以饮料为主体，用调料来“装饰”饮料。

（1）创建饮料抽象类

```
/**
 * 饮料抽象类（组件）
 */
public abstract class Beverage {

    protected String description = "Unknown Beverage";

    /**
     * 描述
     */
    public String getDescription() {
        return description;
    }

    /**
     * 价格
     */
    public abstract double cost();
}
```

（2）创建具体的饮料，并继承饮料抽象类

```
/**
 * 浓缩咖啡
 */
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }
```

```
    @Override
    public double cost() {
        return 1.99;
    }
}

/**
 * 综合咖啡
 */
public class HouseBlend extends Beverage {

    public HouseBlend() {
        description = "House Blend Coffee";
    }

    @Override
    public double cost() {
        return 0.89;
    }
}
```

（3）创建调料抽象类，并继承饮料抽象类

```
/**
 * 调料抽象类（装饰者抽象类）
 */
public abstract class CondimentDecorator extends Beverage {

    @Override
    public abstract String getDescription();
}
```

（4）创建具体的调料，并继承调料抽象类

```
/**
 * 摩卡（装饰者）
 */
public class Mocha extends CondimentDecorator {

    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    @Override
    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    @Override
    public double cost() {
```

```
        // 加上摩卡的价格
        return beverage.cost() + 0.20;
    }
}

/**
 * 豆浆（装饰者）
 */
public class Soy extends CondimentDecorator {

    Beverage beverage;

    public Soy(Beverage beverage) {
        this.beverage = beverage;
    }

    @Override
    public String getDescription() {
        return beverage.getDescription() + ", Soy";
    }

    @Override
    public double cost() {
        // 加上豆浆的价格
        return beverage.cost() + 0.15;
    }
}

/**
 * 奶泡（装饰者）
 */
public class Whip extends CondimentDecorator {

    Beverage beverage;

    public Whip(Beverage beverage) {
        this.beverage = beverage;
    }

    @Override
    public String getDescription() {
        return beverage.getDescription() + ", Whip";
    }

    @Override
    public double cost() {
        // 加上奶泡的价格
        return beverage.cost() + 0.10;
    }
}
```

（5）测试

```
public class Test {
```

```
    public static void main(String[] args) {
        // 浓缩咖啡
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription() + " $" + beverage.cost());

        // 综合咖啡
        Beverage beverage2 = new HouseBlend();
        System.out.println(beverage2.getDescription() + " $" + beverage2.cost());
        // 添加摩卡
        beverage2 = new Mocha(beverage2);
        // 添加豆浆
        beverage2 = new Soy(beverage2);
        // 添加奶泡
        beverage = new Whip(beverage2);
        System.out.println(beverage2.getDescription() + " $" + beverage2.cost());
    }
}
```