

30 | 装饰器模式：如何优化电商系统中复杂的商品价格策略？

2019-08-01 刘超



你好，我是刘超。

开始今天的学习之前，我想先请你思考一个问题。假设现在有这样一个需求，让你设计一个装修功能，用户可以动态选择不同的装修功能来装饰自己的房子。例如，水电装修、天花板以及粉刷墙等属于基本功能，而设计窗帘装饰窗户、设计吊顶装饰房顶等未必是所有用户都需要的，这些功能则需要实现动态添加。还有就是一旦有新的装修功能，我们也可以实现动态添加。如果要你来负责，你会怎么设计呢？

此时你可能会想了，通常给一个对象添加功能，要么直接修改代码，在对象中添加相应的功能，要么派生对应的子类来扩展。然而，前者每次都需要修改对象的代码，这显然不是理想的面向对象设计，即便后者是通过派生对应的子类来扩展，也很难满足复杂的随意组合功能需求。

面对这种情况，使用装饰器模式应该再合适不过了。它的优势我想你多少知道一点，我在这里总结一下。

装饰器模式能够实现为对象动态添加装修功能，它是从一个对象的外部来给对象添加功能，所以有非常灵活的扩展性，我们可以在对原来的代码毫无修改的前提下，为对象添加新功能。除此之外，装饰器模式还能够实现对象的动态组合，借此我们可以很灵活地给动态组合的对象，匹配所需要的功能。

下面我们就通过实践，具体看看该模式的优势。

什么是装饰器模式？

在这之前，我先简单介绍下什么是装饰器模式。装饰器模式包括了以下几个角色：接口、具体对象、装饰类、具体装饰类。

接口定义了具体对象的一些实现方法；具体对象定义了一些初始化操作，比如开头设计装修功能的案例中，水电装修、天花板以及粉刷墙等都是初始化操作；装饰类则是一个抽象类，主要用来初始化具体对象的一个类；其它的具体装饰类都继承了该抽象类。

下面我们就通过装饰器模式来实现下装修功能，代码如下：

```
/**
 * 定义一个基本装修接口
 * @author admin
 *
 */
public interface IDecorator {

    /**
     * 装修方法
     */
    void decorate();

}
```

```
/**
 * 装修基本类
 * @author admin
 *
 */
public class Decorator implements IDecorator{

    /**
     * 基本实现方法
     */
    public void decorate() {
        System.out.println("水电装修、天花板以及粉刷墙。。。");
    }

}
```

```
/**
 * 基本装饰类
 * @author admin
 *
 */
public abstract class BaseDecorator implements IDecorator{

    private IDecorator decorator;

    public BaseDecorator(IDecorator decorator) {
        this.decorator = decorator;
    }

    /**
     * 调用装饰方法
     */
    public void decorate() {
        if(decorator != null) {
            decorator.decorate();
        }
    }
}
```

```

/**
 * 窗帘装饰类
 * @author admin
 *
 */
public class CurtainDecorator extends BaseDecorator{

    public CurtainDecorator(IDecorator decorator) {
        super(decorator);
    }

    /**
     * 窗帘具体装饰方法
     */
    @Override
    public void decorate() {
        System.out.println("窗帘装饰。。。");
        super.decorate();
    }

}

```

```

public static void main( String[] args )
{
    IDecorator decorator = new Decorator();
    IDecorator curtainDecorator = new CurtainDecorator(decorator);
    curtainDecorator.decorate();

}

```

运行结果：

窗帘装饰。。。
水电装修、天花板以及粉刷墙。。。

通过这个案例，我们可以了解到：如果我们想要在基础类上添加新的装修功能，只需要基于抽象类 **BaseDecorator** 去实现继承类，通过构造函数调用父类，以及重写装修方法实现装修窗帘的功能即可。在 **main** 函数中，我们通过实例化装饰类，调用装修方法，即可在基础装修的前提下，获得窗帘装修功能。

基于装饰器模式实现的装修功能的代码结构简洁易读，业务逻辑也非常清晰，并且如果我们需要扩展新的装修功能，只需要新增一个继承了抽象装饰类的子类即可。

在这个案例中，我们仅实现了业务扩展功能，接下来，我将通过装饰器模式优化电商系统中的商品价格策略，实现不同促销活动的灵活组合。

优化电商系统中的商品价格策略

相信你一定不陌生，购买商品时经常会用到的限时折扣、红包、抵扣券以及特殊抵扣金等，种类很多，如果换到开发视角，实现起来就更复杂了。

例如，每逢双十一，为了加大商城的优惠力度，开发往往要设计红包+限时折扣或红包+抵扣券等组合来实现多重优惠。而在平时，由于某些特殊原因，商家还会赠送特殊抵扣券给购买用户，而特殊抵扣券+各种优惠又是另一种组合方式。

要实现以上这类组合优惠的功能，最快、最普遍的实现方式就是通过大量 **if-else** 的方式来实现。但这种方式包含了大量的逻辑判断，致使其他开发人员很难读懂业务，并且一旦有新的优惠策略或者价格组合策略出现，就需要修改代码逻辑。

这时，刚刚介绍的装饰器模式就很适合用在这里，其相互独立、自由组合以及方便动态扩展功能的特性，可以很好地解决 **if-else** 方式的弊端。下面我们就用装饰器模式动手实现一套商品价格策略的优化方案。

首先，我们先建立订单和商品的属性类，在本次案例中，为了保证简洁性，我只建立了几个关键字段。以下几个重要属性关系为，主订单包含若干详细订单，详细订单中记录了商品信息，商品信息中包含了促销类型信息，一个商品可以包含多个促销类型（本案例只讨论单个促销和组合促销）：

```
/**
 * 主订单
 * @author admin
 *
 */
public class Order {

    private int id; //订单ID
    private String orderNo; //订单号
    private BigDecimal totalPayMoney; //总支付金额
    private List<OrderDetail> list; //详细订单列表
}
```

```
/**
 * 详细订单
 * @author admin
 *
 */
public class OrderDetail {
    private int id; //详细订单ID
    private int orderId; //主订单ID
    private Merchandise merchandise; //商品详情
    private BigDecimal payMoney; //支付单价
}
```

```
/**
 * 商品
 * @author admin
 *
 */
public class Merchandise {

    private String sku;//商品SKU
    private String name; //商品名称
    private BigDecimal price; //商品单价
    private Map<PromotionType, SupportPromotions> supportPromotions; //支持促销类型
}
```



```
/**
 * 促销类型
 * @author admin
 *
 */
public class SupportPromotions implements Cloneable{

    private int id;//该商品促销的ID
    private PromotionType promotionType;//促销类型 1\优惠券 2\红包
    private int priority; //优先级
    private UserCoupon userCoupon; //用户领取该商品的优惠券
    private UserRedPacket userRedPacket; //用户领取该商品的红包

    //重写clone方法
    public SupportPromotions clone(){
        SupportPromotions supportPromotions = null;
        try{
            supportPromotions = (SupportPromotions)super.clone();
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        return supportPromotions;
    }
}
```

```
/**
 * 优惠券
 * @author admin
 *
 */
public class UserCoupon {

    private int id; //优惠券ID
    private int userId; //领取优惠券用户ID
    private String sku; //商品SKU
    private BigDecimal coupon; //优惠金额
}
```

```
/**
 * 红包
 * @author admin
 *
 */
public class UserRedPacket {

    private int id; //红包ID
    private int userId; //领取用户ID
    private String sku; //商品SKU
    private BigDecimal redPacket; //领取红包金额
}
```

接下来，我们再建立一个计算支付金额的接口类以及基本类：

```

/**
 * 计算支付金额接口类
 * @author admin
 *
 */
public interface IBaseCount {

    BigDecimal countPayMoney(OrderDetail orderDetail);

}

```

```

/**
 * 支付基本类
 * @author admin
 *
 */
public class BaseCount implements IBaseCount{

    public BigDecimal countPayMoney(OrderDetail orderDetail) {
        orderDetail.setPayMoney(orderDetail.getMerchandise().getPrice());
        System.out.println("商品原单价金额为: " + orderDetail.getPayMoney());

        return orderDetail.getPayMoney();
    }

}

```

然后，我们再建立一个计算支付金额的抽象类，由抽象类调用基本类：

```
/**
 * 计算支付金额的抽象类
 * @author admin
 *
 */
public abstract class BaseCountDecorator implements IBaseCount{

    private IBaseCount count;

    public BaseCountDecorator(IBaseCount count) {
        this.count = count;
    }

    public BigDecimal countPayMoney(OrderDetail orderDetail) {
        BigDecimal payTotalMoney = new BigDecimal(0);
        if(count!=null) {
            payTotalMoney = count.countPayMoney(orderDetail);
        }
        return payTotalMoney;
    }
}
```

然后，我们再通过继承抽象类来实现我们所需要的修饰类（优惠券计算类、红包计算类）：

```
/**
 * 计算使用优惠券后的金额
 * @author admin
 *
 */
public class CouponDecorator extends BaseCountDecorator{

    public CouponDecorator(IBaseCount count) {
        super(count);
    }

    public BigDecimal countPayMoney(OrderDetail orderDetail) {
        BigDecimal payTotalMoney = new BigDecimal(0);
        payTotalMoney = super.countPayMoney(orderDetail);
        payTotalMoney = countCouponPayMoney(orderDetail);
        return payTotalMoney;
    }

    private BigDecimal countCouponPayMoney(OrderDetail orderDetail) {

        BigDecimal coupon = orderDetail.getMerchandise().getSupportPromotions().get(PromotionType.COUPON).getU
        System.out.println("优惠券金额: " + coupon);

        orderDetail.setPayMoney(orderDetail.getPayMoney().subtract(coupon));
        return orderDetail.getPayMoney();
    }
}
```

```

/**
 * 计算使用红包后的金额
 * @author admin
 *
 */
public class RedPacketDecorator extends BaseCountDecorator{

    public RedPacketDecorator(IBaseCount count) {
        super(count);
    }

    public BigDecimal countPayMoney(OrderDetail orderDetail) {
        BigDecimal payTotalMoney = new BigDecimal(0);
        payTotalMoney = super.countPayMoney(orderDetail);
        payTotalMoney = countCouponPayMoney(orderDetail);
        return payTotalMoney;
    }

    private BigDecimal countCouponPayMoney(OrderDetail orderDetail) {

        BigDecimal redPacket = orderDetail.getMerchandise().getSupportPromotions().get(PromotionType.REDPACKET);
        System.out.println("红包优惠金额: " + redPacket);

        orderDetail.setPayMoney(orderDetail.getPayMoney().subtract(redPacket));
        return orderDetail.getPayMoney();
    }
}

```

最后，我们通过一个工厂类来组合商品的促销类型：

```

/**
 * 计算促销后的支付价格
 * @author admin
 *
 */
public class PromotionFactory {

```

```
public static BigDecimal getPayMoney(OrderDetail orderDetail) {

    //获取给商品设定的促销类型
    Map<PromotionType, SupportPromotions> supportPromotionslist = orderDetail.getMerchandise().getSupportPromotions();

    //初始化计算类
    IBaseCount baseCount = new BaseCount();
    if(supportPromotionslist!=null && supportPromotionslist.size()>0) {
        for(PromotionType promotionType: supportPromotionslist.keySet()) { //遍历设置的促销类型，通过装饰器组合促销
            baseCount = protmotion(supportPromotionslist.get(promotionType), baseCount);
        }
    }
    return baseCount.countPayMoney(orderDetail);
}

/**
 * 组合促销类型
 * @param supportPromotions
 * @param baseCount
 * @return
 */
private static IBaseCount protmotion(SupportPromotions supportPromotions, IBaseCount baseCount) {
    if(supportPromotions.getPromotionType()==PromotionType.COUPON) {
        baseCount = new CouponDecorator(baseCount);
    }else if(supportPromotions.getPromotionType()==PromotionType.REDPACKED) {
        baseCount = new RedPacketDecorator(baseCount);
    }
    return baseCount;
}

}
```

```
public static void main( String[] args ) throws InterruptedException, IOException
{
    Order order = new Order();
    init(order);

    for(OrderDetail orderDetail: order.getList()) {
        BigDecimal payMoney = PromotionFactory.getPayMoney(orderDetail);
        orderDetail.setPayMoney(payMoney);
        System.out.println("最终支付金额: " + orderDetail.getPayMoney());
    }
}
```

运行结果:

```
商品原单价金额为: 20
优惠券金额: 3
红包优惠金额: 10
最终支付金额: 7
```

以上源码可以通过 [Github](#) 下载运行。通过以上案例可知：使用装饰器模式设计的价格优惠策略，实现各个促销类型的计算功能都是相互独立的类，并且可以通过工厂类自由组合各种促销类型。

总结

这讲介绍的装饰器模式主要用来优化业务的复杂度，它不仅简化了我们的业务代码，还优化了业务代码的结构设计，使得整个业务逻辑清晰、易读易懂。

通常，装饰器模式用于扩展一个类的功能，且支持动态添加和删除类的功能。在装饰器模式中，装饰类和被装饰类都只关心自身的业务，不相互干扰，真正实现了解耦。

思考题

责任链模式、策略模式与装饰器模式有很多相似之处。平时，这些设计模式除了在业务中被用到以外，在架构设计中也经常被用到，你是否在源码中见过这几种设计模式的使用场景呢？欢迎你与大家分享。

Java 性能调优实战

覆盖 80% 以上 Java 应用调优场景

刘超

金山软件西山居技术经理



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

精选留言



密码123456

👍 3

责任链。最常见到的就是接收http请求了。帮我们转码，转化成实体类，等等。策略模式。最常简单和用到的就是集合排序，自定义排序规则。装饰器，最常见到的就是各种流，比如字符流，字节流等

2019-08-01



CCC

👍 2

有几个今年秋招的！举个手！老师的课程真的收获很多！

2019-08-01



QQ怪

👍 1

老师这个案例来的太及时了，正想重构公司订单优惠券红包扣除这方面的代码，真的是及时雨啊？厉害厉害👍

2019-08-01



冯传博

👍 0

希望能有个类图，这样就能一目了然的看清楚各个类之间的关系了

2019-08-01



-W.LI-

👍 0

```
public BigDecimal countPayMoney(OrderDetail orderDetail) {
```

```
BigDecimal payTotalMoney = new BigDecimal(0);  
payTotalMoney = super.countPayMoney(orderDetail);  
payTotalMoney = countCouponPayMoney(orderDetail);  
return payTotalMoney;  
}
```

老师好!这里为啥要调用父类的countPayMoney()方法啊?

责任链模式:感觉责任连模式比较固定不怎么会变一层往一层调用, 解耦, 某一层变了不影响别的层。

策略模式:策略模式, 虽然也是封装了很多不同的策略, 但是使用时一般一次只选一个实现类使用, 不会有嵌套。

装饰者模式:责任链有的优点他都有, 装饰者还能动态组合。

谢谢老师, 希望给出详细答案谢谢

2019-08-01



nightmare

👍 0

netty中的pipeline, tomcat中的filter, 属于责任链, springmvc中对参数解析的就是 策略模式, 每一个参数类型一个实现类, 用for循环解析参数 java.io就是经典的装饰器模式

2019-08-01