工厂模式和策略模式的综合使用

原创 张彦峰ZYF 已于 2023-03-14 14:41:45 修改 10642 收藏 31 版权
分类专栏： 设计模式应用 JAVA基础讲解与总结 文章标签： 策略模式 java 开发语言

目录

目录
一、简单的工厂模式了解与使用
（一）基本概念理解
（二）简单工厂模式的认识和对应角...
基本认识
角色理解
（三）使用场景和典型应用
二、简单的策略模式了解与使用
（一）基本概念理解
（二）策略模式认识和对应角色的分析
基本认识
角色理解
（三）使用场景和典型应用
三、工厂模式和策略模式的综合使用
(一)应用背景介绍
（二）设计策略类
1.定义一个计算应付价格接口
2.用户是专属会员对应策略类
3.用户是超级会员对应策略类
4.用户是普通会员对应策略类
（三）设计对应的简单工厂模式
（四）知识补充：Spring Bean的注册
（五）具体实现应用测试如下
1.具体测试代码

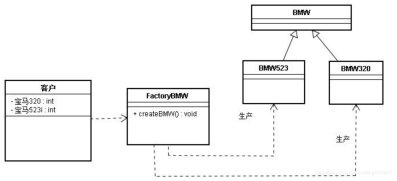
其实在很多的开发设计中，将工厂模式和策略模式的综合使用的案例是很多的，而且解决的实际问题也一样很多，本次对基本的简单工厂模式和策略模式做简单介绍，重点放在两者结合后的具体应用上做分析和讲解。

一、简单的工厂模式了解与使用

（一）基本概念理解

建立一个工厂，能轻松方便地构造对象实例，而不必关心构造对象实例的细节和复杂过程，这就是简单工厂的主要功能。

比方如下图：客户需要一辆宝马，具体的简单工厂会根据用户的实际需求去生产对应型号的宝马，最后返回客户需要的宝马产品。



（二）简单工厂模式的认识和对应角色的分析

基本认识

简单工厂模式(Simple Factory Pattern)需要定义一个工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类。因为在简单工厂模式中用于创建实例的方法是静态(static)方法，因此简单工厂模式又被称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式，但不属于GOF23种设计模式。

角色理解

- Factory（工厂角色）：工厂角色即工厂类，它是简单工厂模式的核心，负责实现创建所有产品实例的内部逻辑；工厂类可以被外界直接调用，创建所需的产品对象；在工厂类中提供了静态的工厂方法factoryMethod()，它的返回类型为抽象产品类型Product
- Product（抽象产品角色）：它是工厂类所创建的所有对象的父类，封装了各种产品对象的公有方法，它的引入将提高系统的灵活性，使得在工厂类中只需定义一个通用的工厂方法，因为所有创建的具体产品对象都是其子类对象。
- ConcreteProduct（具体产品角色）：它是简单工厂模式的创建目标，所有被创建的对象都充当这个角色的某个具体类的实例。每一个具体产品角色都继承了抽象产品角色，需要实现在抽象产品中声明的抽象方法

在简单工厂模式中，客户端通过工厂类来创建一个产品类的实例，而无须直接使用new关键字来创建对象，它是工厂模式家族中最简单的一员。

（三）使用场景和典型应用

- 工厂类负责创建的对象比较少，由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂。
- 客户端只知道传入工厂类的参数，对于如何创建对象并不关心。
- 典型应用：Calendar 类获取日历类对象、JDBC 获取数据库连接、Logback 中的 LoggerFactory 获取 Logger 对象

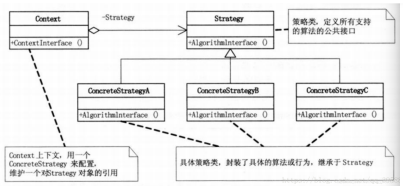
二、简单的策略模式了解与使用

（一）基本概念理解

主要指对象有某个行为，但是在不同的场景中，该行为有不同的实现算法。

比如每个人都要“交个人所得税”，但是“在美国交个人所得税”和“在中国交个人所得税”就有不同的算税方法。

在实际的代码中，外卖平台上的某家店铺为了促销，设置了多种会员优惠，其中包含超级会员折扣8折、普通会员折扣9折和普通用户没有折扣三种，也就是针对不同的会员有不同的优惠力度。



（二）策略模式认识和对应角色的分析

基本认识

策略模式是行为模式之一，它对一系列的算法加以封装，为所有算法定义一个抽象的算法接口，并通过继承该抽象算法接口对所有的算法加以封装和实现，具体的算法选择交由客户端决定（策略）。Strategy 模式主要用来平滑地处理算法的切换。

角色理解

- Strategy : 策略（算法）抽象。
- ConcreteStrategy :各种策略（算法） 的具体实现
- Contenxt :策略的外部封装类，或者说策略的容器类。根据不同策略执行不同的行为，策略由外部环境决定。

（三）使用场景和典型应用

- 策略模式的等级结构定义了一个算法或行为族，恰当使用继承可以把公共的代码移到父类里面，从而避免重复的代码。
- 策略模式提供了可以替换继承关系的办法。
- 使用策略模式可以避免使用多重条件转移语句。多重转移语句不易维护。它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起。统统列在一个多重转移语句里面，比使用继承的办法还要原始和落后。

三、工厂模式和策略模式的综合使用

(一)应用背景介绍

假设我们要做一个外卖平台，有这样的需求：

- 外卖平台上的某家店铺为了促销，设置了多种会员优惠，其中包含超级会员折扣8折、普通会员折扣9折和普通用户没有折扣三种。
- 希望用户在付款的时候，根据用户的会员等级，就可以知道用户符合哪种折扣策略，进而进行打折，计算出应付金额。
- 随着业务发展，新的需求要求专属会员要在店铺下单金额大于30元的时候才可以享受优惠。
- 接着，又有一个变态的需求，如果用户的超级会员已经到期了，并且到期时间在一周内，那么就对用户的单笔订单按照超级会员进行折扣，并在收银台进行强提醒，引导用户再次开通会员，而且折扣只进行一次。

（二）设计策略类

根据要求，用户具有会员等级，对应的会员等级在付款的时候享受对应的折扣策略，而后面是具体业务在不同变动时新增的内容，可以在具体的策略中去单独操作或在外围确定用户身份后进行基本的处理。

总之，从基本操作上来看，付款的行为是必然存在的，不同的用户在对应不同的变动下支付的具体策略有所不同，所以，可以确定一个基本的接口来表达具体要付款的行为，不同用户对具体等级的实现策略放在对应的实现类中做处理：

1.定义一个计算应付价格接口

```java
/**
 * @author yanfengzhang
 */
public interface UserPayService {

    /**
     * 功能描述：计算应付价格
     * @author yanfengzhang
     * @date 2019-12-31 18:09
     * @param orderPrice BigDecimal
     * @return BigDecimal
     */
    BigDecimal quote(BigDecimal orderPrice);
}
```

2.用户是专属会员对应策略类

```java
/**
 * 描述：用户是专属会员---订单金额大于30元,7折价格/否则9折价格
 *
 * @author yanfengzhang
 * @date 2019-12-31 18:11
 */
@Service
public class ParticularlyVipPayServiceImpl implements UserPayService, InitializingBean {
    @Override
    public BigDecimal quote(BigDecimal orderPrice) {
        int payPrice = orderPrice.intValue();
        if (payPrice > 30) {
            return new BigDecimal(payPrice * 0.7);
        }
        return new BigDecimal(payPrice * 0.9);
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        UserPayServiceStrategyFactory.register("ParticularlyVip", this);
    }
}
```

3.用户是超级会员对应策略类

```java
/**
 * 描述：用户是超级会员---8折价格
 *
 * @author yanfengzhang
 * @date 2019-12-31 18:13
 */
@Service
public class SuperVipPayServiceImpl implements UserPayService , InitializingBean {
    @Override
    public BigDecimal quote(BigDecimal orderPrice) {
        int payPrice = orderPrice.intValue();
        return new BigDecimal(payPrice * 0.8);
```

```
    }
     14
    @Override
    public void afterPropertiesSet() throws Exception {
        UserPayServiceStrategyFactory.register("SuperVip",this);
    }
}
```

4.用户是普通会员对应策略类

```
/**
 * 描述：用户是普通会员
 * 情况1：该用户超级会员刚过期并且尚未使用过临时折扣-->临时折扣使用次数更新-->8折价格
 * 情况2：非以上情况-->9折价格
 *
 * @author yanfengzhang
 * @date 2019-12-31 18:15
 */
@Service
public class VipPayServiceImpl implements UserPayService, InitializingBean {
    @Override
    public BigDecimal quote(BigDecimal orderPrice) {
        int payPrice = orderPrice.intValue();
        /*该用户超级会员刚过期并且尚未使用过临时折扣*/
        if (conditions()) {
            /*临时折扣使用次数更新*/
            updateSomething();
            return new BigDecimal(payPrice * 0.8);
        }
        return new BigDecimal(payPrice * 0.9);
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        UserPayServiceStrategyFactory.register("Vip", this);
    }

    /**
     * 功能描述：满足一定的条件
     *
     * @author yanfengzhang
     * @date 2020-01-02 09:11
     */
    private boolean conditions() {
        return true;
    }

    private void updateSomething() {

    }
}
```

（三）设计对应的简单工厂模式

为了方便我们从Spring中获取UserPayService的各个策略类，我们创建一个工厂类来实现针对不同的用户实现对应的支付策略，具体如下：

```
/**
 * 描述：获取UserPayService的各个策略类
 * UserPayServiceStrategyFactory中定义了一个Map，用来保存所有的策略类的实例，并提供一个getByUserType方法
 *
 * @author yanfengzhang
 * @date 2019-12-31 18:28
 */
public class UserPayServiceStrategyFactory {
    private static Map<String, UserPayService> services = new ConcurrentHashMap<String,UserPayS

    public  static UserPayService getByUserType(String type){
        return services.get(type);
    }

    public static void register(String userType,UserPayService userPayService){
        Assert.notNull(userType,"userType can't be null");
        services.put(userType,userPayService);
    }
}
```

（四）知识补充：Spring Bean的注册

UserPayServiceStrategyFactory提供了register方法，用来注册策略服务的。各个策略类调用register方法，把Spring通过IOC创建出来的Bean注册进去就行了。这种需求，可以借用Spring种提供的InitializingBean接口，这个接口为Bean提供了属性初始化后的处理方法，它只包括afterPropertiesSet方法，凡是继承该接口的类，在bean的属性初始化后都会执行该方法。

只需要每一个策略服务的实现类都实现InitializingBean接口，并实现其afterPropertiesSet方法，在这个方法中调用UserPayServiceStrategyFactory.register即可。

这样，在Spring初始化的时候，当创建VipPayService、SuperVipPayService和ParticularlyVipPayService的时候，会在Bean的属性初始化之后，把这个Bean注册到UserPayServiceStrategyFactory中。

（五）具体实现应用测试如下

1.具体测试代码

```
/**
 * 描述：通过策略模式、工厂模式以及Spring的InitializingBean，提升了代码的可读性以及可维护性，彻底消灭了一坨i
 * 注：1.这里使用的并不是严格意义上面的策略模式和工厂模式。
 *
 * @author yanfengzhang
 * @date 2019-12-31 18:21
 */
@RunWith(SpringRunner.class)
@SpringBootTest(classes = ZYFApplication.class, webEnvironment = SpringBootTest.WebEnvironment
public class DeleteIfElseSkill {
    @Test
    public void testDeleteIfElseSkill() {
        User user1 = new User();
        user1.setVipType("ParticularlyVip");
        user1.setOrderPrice(new BigDecimal("100"));

        User user2 = new User();
        user2.setVipType("SuperVip");
        user2.setOrderPrice(new BigDecimal("100"));

```

```
        User user3 = new User();22          user3.setVipType("Vip");
        user3.setOrderPrice(new BigDecimal("100"));


        BigDecimal payPrice1 = UserPayServiceStrategyFactory.getByUserType(user1.getVipType()).
        System.out.println("用户为专属会员，并且订单金额为50，按会员优惠最后应支付：" + payPrice1);

        BigDecimal payPrice2 = UserPayServiceStrategyFactory.getByUserType(user2.getVipType()).
        System.out.println("用户为超级会员，并且订单金额为50，按会员优惠最后应支付：" + payPrice2);

        BigDecimal payPrice3 = UserPayServiceStrategyFactory.getByUserType(user3.getVipType()).
        System.out.println("用户为普通会员，并且订单金额为50，按会员优惠最后应支付：" + payPrice3);
    }

    @Data
    private static class User {
        private String uid;
        private String vipType;
        private BigDecimal orderPrice;
    }
}
```

2.对应结果展示

```
用户为专属会员，并且订单金额为50，按会员优惠最后应支付：70
用户为超级会员，并且订单金额为50，按会员优惠最后应支付：80
用户为普通会员，并且订单金额为50，按会员优惠最后应支付：80
```

参考书籍、文献和资料

1.https://blog.csdn.net/jason0539/article/details/23020989

2.设计模式 | 简单工厂模式及典型应用_简单工厂模式的应用_小旋锋的博客-CSDN博客

3.https://blog.csdn.net/lzb348110175/article/details/91633620

4.https://blog.csdn.net/Crazy_Cw/article/details/106818217

5.业务复杂=if else？刚来的大神竟然用策略+工厂彻底干掉了他们！