策略模式

2020年2月17日 9:49

1. 概念

定义了算法家族，分别封装起来，让他们之间可以相互替换，此模式让算法的变化不会影响到使用算法的用户。

通过策略模式可以消除掉大量if...else...代码。

策略模式仅仅封装算法，但策略模式并不决定在何时使用何种算法，算法的选择由客户端来决定。

例如田忌赛马，可以采用不同的策略，又比如有两个数，可以采用加法策略，乘法策略等等。

2. UML结构图

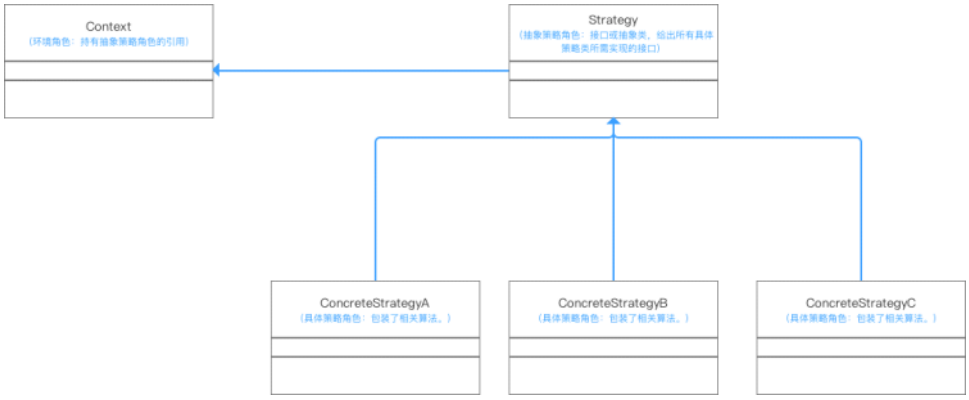

图2.1 策略模式UML结构图

1. 抽象策略角色（Strategy）：接口或抽象类，给出所有具体策略类所需实现的接口。
2. 具体策略角色（ConcreteStrategy）：对抽象策略角色的实现或继承，封装了相关算法。
3. 环境角色（Context）：持有抽象策略角色的引用，调用具体策略角色的相关方法。

3. 策略模式实例

随着双十一的到来，某在线学习网站制定了不同的促销策略，要求能根据不同的情景使用不同的促销策略。现使用策略模式对上述进行实现，结合享元模式进行处理。

1. 抽象策略角色

```
public interface PromotionStrategy {
    void doPromotion();
}
```

2. 具体策略角色

```
public class FanxianPromotionStrategy implements PromotionStrategy {
    @Override
    public void doPromotion() {
        System.out.println("进行返现促销，满200返现200");
    }
}
```

```
public class LijianPromotionStrategy implements PromotionStrategy{
    @Override
```

```
    public void doPromotion() {
        System.out.println("进行立减促销，立减20元");
    }
}
```

```
public class FanxianPromotionStrategy implements PromotionStrategy {
    @Override
    public void doPromotion() {
        System.out.println("进行返现促销，满200返现200");
    }
}
```

```
public class EmptyPromotionStrategy implements PromotionStrategy{
    @Override
    public void doPromotion() {
        System.out.println("暂无促销活动");
    }
}
```

3. 环境角色

```
public class PromotionActivity {
    private PromotionStrategy promotionStrategy;

    public PromotionActivity(PromotionStrategy promotionStrategy) {
        this.promotionStrategy = promotionStrategy;
    }

    public void executePromotionStrategy(){
        promotionStrategy.doPromotion();
    }
}
```

4. 策略工厂

```
public class PromotionStrategyFactory {
    // 采用枚举类型
    private enum PromotionKey{
         LIJIAN("LIJIAN"),MANJIAN("MANJIAN"),FANXIAN("FANXIAN");
         String StringName;
        PromotionKey(String stringName) {
            StringName = stringName;
        }
    }
    private static final PromotionStrategy EMPTY_PROMOTION = new
EmptyPromotionStrategy();

    private static Map<String,PromotionStrategy> PROMOTION_STRATEGY_MAP = new
HashMap<String,PromotionStrategy>();
    // 类记载的时候就把PROMOTION_STRATEGY_MAP进行填充，也可使用享元模式进行动态填充
    static {
        PROMOTION_STRATEGY_MAP.put(PromotionKey.MANJIAN.StringName,new
ManjianPromotionStrategy());
        PROMOTION_STRATEGY_MAP.put(PromotionKey.LIJIAN.StringName,new
LijianPromotionStrategy());
        PROMOTION_STRATEGY_MAP.put(PromotionKey.FANXIAN.StringName,new
FanxianPromotionStrategy());
    }

    private PromotionStrategyFactory() {
    }

    public static PromotionStrategy getPromotionStrategy(String promotionKey){
        PromotionStrategy promotionStrategy =
PROMOTION_STRATEGY_MAP.get(promotionKey);
        return promotionStrategy == null ?  EMPTY_PROMOTION: promotionStrategy;
    }
}
```

5. 客户端

```
@Test
    public void strategyTest2(){
        String promotionKey = "LIJIAN";
        PromotionActivity promotionActivity = new
PromotionActivity(PromotionStrategyFactory.getPromotionStrategy(promotionKey));
        promotionActivity.executePromotionStrategy();

         String promotionKey1 = "MANJIAN";
        PromotionActivity promotionActivity1 = new
PromotionActivity(PromotionStrategyFactory.getPromotionStrategy(promotionKey1));
        promotionActivity1.executePromotionStrategy();

    }
```

6. 结果

```
进行立减促销，立减20元
进行满减促销，满200-20
```

4. 优/缺点

1. 优点

- 开闭原则。
- 避免使用多重条件转移语句。
- 提高了算法的保密性和安全性。
- 策略类之间可以自由切换，由于策略类都实现同一个接口，所以使它们之间可以自由切换。

2. 缺点

1. 客户端必须知道所有策略类，并自行决定使用哪个策略类。
2. 产生很多策略类，可通过享元模式在一定程度上减少类的个数。

5. 使用场景

1. 系统很多类，而他们的区别仅仅在于他们的行为不同。
2. 一个系统需要动态地在几个算法中选择一种。

6. 参考资料

[1] https://www.jianshu.com/p/0c62bf587b9c