# 策略模式

## 什么是策略模式

策略模式让算法独立于使用它的客户而独立变化。策略模式重点是封装不同的算法和行为，不同的场景下可以相互替换。策略模式是开闭原则的体现，开闭原则讲的是一个软件实体应该对扩展开放对修改关闭。策略模式在新的策略增加时，不会影响其他类的修改，增加了扩展性，也就是对扩展是开放的；对于场景来说，只依赖于抽象，而不依赖于具体实现，所以对修改是关闭的。策略模式的认识可以借助《java与模式》一书中写到诸葛亮的锦囊妙计来学习，在不同的场景下赵云打开不同的锦囊，便化险为夷，锦囊便是抽象策略，具体的锦囊里面的计策便是具体的策略角色，场景就是赵云，变化的处境选择具体策略的条件。策略模式可以有效的避免过多的if-else的出现。

## 策略模式的特性

下面通过一个策略模式的UML图表示：

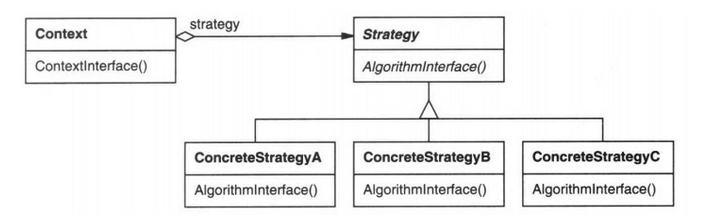


图2-1 策略模式UML图

* 抽象策略角色： 策略类，通常由一个接口或者抽象类实现。
* 具体策略角色：包装了相关的算法和行为。
* 环境角色：持有一个策略类的引用，最终给客户端调用。

## 举例说明

根据现在广大人民都喜欢vip来举例子吧。

* 一级vip 9折
* 二级vip 8折
* 三级vip 7折

常规的方法来展示：

public class VIP {  
 enum VipType {  
 *one*,*two*,*three* }  
 double getMoney(double money,VipType type){  
 switch (type){  
 case *one*:  
 return money\*0.9;  
 case *two*:  
 return money\*0.8;  
 case *three*:  
 return money\*0.7;  
 }  
 return money;  
 }  
}

创建一个计算钱的类，根据输入的参数，使用switch来完成。

例子:

VIP vip=new VIP();  
System.*out*.println(vip.getMoney(1000, VIP.VipType.*one*));  
System.*out*.println(vip.getMoney(1000, VIP.VipType.*two*));  
System.*out*.println(vip.getMoney(1000, VIP.VipType.*three*));

总结：

看起来好像很简单的样子，但是如果有很多很多种vip，VIP类会变得非常庞大；如果计算方式改变呢，每次都需要修改这个类，维护起来会非常费劲。

接下来展现我们的方法了：

1. 定义算法接口（抽象策略角色）

public interface Strategy {  
 double getMoney(double monty, VIP.VipType type);  
}

1. 具体实现多个实现（具体策略角色）

public class OneStrategy implements Strategy {  
 @Override  
 public double getMoney(double monty, VIP.VipType type) {  
 return monty\*0.9;  
 }  
}

public class TwoStrategy implements Strategy {  
 @Override  
 public double getMoney(double monty, VIP.VipType type) {  
 return monty\*0.8;  
 }  
}

public class ThreeStrategy implements Strategy {  
 @Override  
 public double getMoney(double monty, VIP.VipType type) {  
 return monty\*0.7;  
 }  
}

1. 定义上下文（环境角色）

首先定义一个工厂：

import java.util.HashMap;  
import java.util.Map;  
  
public class StrategyFactory {  
 private static StrategyFactory *factory*;  
  
 private StrategyFactory() {  
 }  
  
 private static Map<VIP.VipType, Strategy> *map* = new HashMap<>();  
  
 static {  
 *map*.put(VIP.VipType.*one*, new OneStrategy());  
 *map*.put(VIP.VipType.*two*, new TwoStrategy());  
 *map*.put(VIP.VipType.*three*, new ThreeStrategy());  
 }  
  
 public static StrategyFactory getInstance() {  
 if (*factory* == null) {  
 synchronized (StrategyFactory.class) {  
 if (*factory* == null) {  
 *factory* = new StrategyFactory();  
 }  
 }  
 }  
 return *factory*;  
 }  
  
 public Strategy getStrategy(VIP.VipType vipType) {  
 return *map*.get(vipType);  
 }  
}

用来生产不同的具体决策对象。

接下来定义上下文：

public class Context {  
 private Strategy strategy;//策略父类  
  
 double getMoney(double money, VIP.VipType vipType){  
 strategy= StrategyFactory.*getInstance*().getStrategy(vipType);//获得具体策略  
 return strategy.getMoney(money,vipType);  
 }  
  
 public Strategy getStrategy() {  
 return strategy;  
 }  
  
 public void setStrategy(Strategy strategy) {  
 this.strategy = strategy;  
 }  
}

实例：

Context context = new Context();  
System.*out*.println(context.getMoney(1000, VIP.VipType.*one*));  
System.*out*.println(context.getMoney(1000, VIP.VipType.*two*));  
System.*out*.println(context.getMoney(1000, VIP.VipType.*three*));

总结：

策略模式把具体的算法封装到了具体策略角色内部，增强了可扩展性，隐蔽了实现细节；它替代继承来实现，避免了if- else这种不易维护的条件语句。当然我们也可以看到，策略模式由于独立策略实现，使得系统内增加了很多策略类；对客户端来说必须知道兜友哪些具体策略， 而且需要知道选择具体策略的条件。

## 优缺点

**优点：**

1. 抽象性更好，重用一下算法和行为。
2. 提供了可以替换继承关系的办法，算法封装在独立的Strategy中，可以更好的理解和扩展。
3. 消除了一些if else条件语句。
4. 可以提供相同行为的不同实现。

**缺点:**

1. 客户端必须知道所有的策略类，并自行决定使用哪一个策略类
2. Strategy和Context之间的通信开销 ：无论各个Strategy实现的算法是简单还是复杂, 它们都共享Strategy定义的接口。
3. 策略模式将造成产生很多策略类：可以通过使用享元模式在一定程度上减少对象的数量。