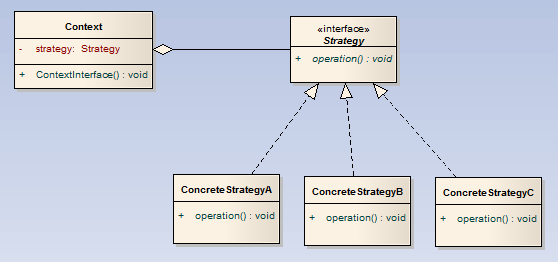
# @JAVA设计模式学习17——策略模式 - alaric's blog - ITeye技术网站

http://alaric.iteye.com/blog/1920714

**策略（Strategy）模式**：又名Policy，它的用意是定义一组算法，把它们一个个封装起来，并且使他们可以相互替换。策略模式可以独立于使用他们的客户端而变化。GOF策略模式静态结构类图如下：  


通过上图可以看出策略模式有以下角色构成：

1、抽象策略（Strategy）角色：抽象策略角色由抽象类或接口来承担，它给出具体策略角色需要实现的接口；

2、具体策略（ConcreteStrategy）角色：实现封装了具体的算法或行为；

3、场景（Context）角色：持有抽象策略类的引用。

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。

策略模式在程序设计中也很常用，在板桥（banq）的博客里有篇文章叫 “你还在用if else吗？”  
“<http://www.jdon.com/artichect/ifelse.htm>”讲的很好，策略模式不但是继承的代替方案而且能很好地解决if else问题，下面举个实例来说明，怎么使用策略模式。

需求如下：

某支付系统接入以下几种商户进行充值：易宝网易，快线网银，19pay手机支付，支付宝支付，骏网一卡通，由于每家充值系统的结算比例不一样，而且同一家商户的不同充值方式也有所不同，具体系统情况比较复杂，像支付宝既有支付宝账号支付和支付宝网银支付等这些暂时不考虑，为了讲述策略模式这里简单描述，假如分为四种，手机支付，网银支付，商户账号支付和点卡支付。因为没个支付结算比例不同，所以对手续费低的做一些优惠活动，尽可能让用户使用手续费低的支付方式来充值，这样降低渠道费用，增加收入，具体优惠政策如下：

①网银充值，8.5折；

②商户充值，9折；

③手机充值，没有优惠；

④点卡充值，收取1%的渠道费；

对于一个新手的代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **package** strategy;
3. **public** **class** Example {
5. /\*\*
6. \*
7. \*作者：alaric
8. \*时间：2013-8-5上午11:00:06
9. \*描述：计算用户所付金额
10. \*/
11. **public** Double calRecharge(Double charge ,RechargeTypeEnum type ){
13. **if**(type.equals(RechargeTypeEnum.E\_BANK)){
14. **return** charge\*0.85;
15. }**else** **if**(type.equals(RechargeTypeEnum.BUSI\_ACCOUNTS)){
16. **return** charge\*0.90;
17. }**else** **if**(type.equals(RechargeTypeEnum.MOBILE)){
18. **return** charge;
19. }**else** **if**(type.equals(RechargeTypeEnum.CARD\_RECHARGE)){
20. **return** charge+charge\*0.01;
21. }**else**{
22. **return** **null**;
23. }
25. }
27. }

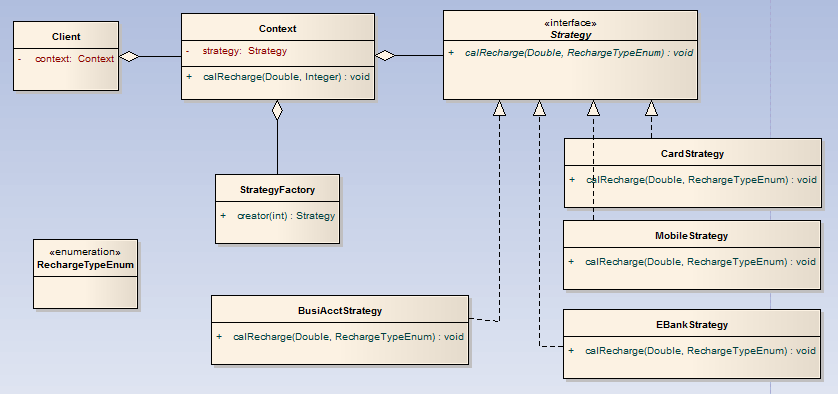
**Java代码  [收藏代码](javascript:void())**

1. **package** strategy;
3. **public** **enum** RechargeTypeEnum {
5. E\_BANK(1, "网银"),
7. BUSI\_ACCOUNTS(2, "商户账号"),
9. MOBILE(3,"手机卡充值"),
11. CARD\_RECHARGE(4,"充值卡")
12. ;
14. /\*\*
15. \* 状态值
16. \*/
17. **private** **int** value;
19. /\*\*
20. \* 类型描述
21. \*/
22. **private** String description;


26. **private** RechargeTypeEnum(**int** value, String description) {
27. **this**.value = value;
28. **this**.description = description;
29. }
31. **public** **int** value() {
32. **return** value;
33. }
34. **public** String description() {
35. **return** description;
36. }

39. **public** **static** RechargeTypeEnum valueOf(**int** value) {
40. **for**(RechargeTypeEnum type : RechargeTypeEnum.values()) {
41. **if**(type.value() == value) {
42. **return** type;
43. }
44. }
45. **return** **null**;
46. }
47. }

 可以看出上面四种不同的计算方式在一个方法内部，不利于扩展和维护，当然也不符合面向对象设计原则。对以上的代码利用策略模式进行修改，类图如下：

  
 实例代码如下：

**Java代码  [收藏代码](javascript:void())**

1. **package** strategy.strategy;
3. **import** strategy.RechargeTypeEnum;
5. /\*\*
6. \*
7. \*作者：alaric
8. \*时间：2013-8-5上午11:03:17
9. \*描述：策略抽象类
10. \*/
11. **public** **interface** Strategy {
13. /\*\*
14. \*
15. \*作者：alaric
16. \*时间：2013-8-5上午11:05:11
17. \*描述：策略行为方法
18. \*/
19. **public** Double calRecharge(Double charge ,RechargeTypeEnum type );
20. }

**Java代码  [收藏代码](javascript:void())**

1. **package** strategy.strategy;
3. **import** strategy.RechargeTypeEnum;
4. /\*\*
5. \*
6. \*作者：alaric
7. \*时间：2013-8-5上午11:14:23
8. \*描述：网银充值
9. \*/
10. **public** **class** EBankStrategy **implements** Strategy{
12. @Override
13. **public** Double calRecharge(Double charge, RechargeTypeEnum type) {
14. **return** charge\*0.85;
15. }


19. }

**Java代码  [收藏代码](javascript:void())**

1. **package** strategy.strategy;
3. **import** strategy.RechargeTypeEnum;
4. /\*\*
5. \*
6. \*作者：alaric
7. \*时间：2013-8-5上午11:14:08
8. \*描述：商户账号充值
9. \*/
10. **public** **class** BusiAcctStrategy **implements** Strategy{
12. @Override
13. **public** Double calRecharge(Double charge, RechargeTypeEnum type) {
14. // TODO Auto-generated method stub
15. **return** charge\*0.90;
16. }
18. }

**Java代码  [收藏代码](javascript:void())**

1. **package** strategy.strategy;
3. **import** strategy.RechargeTypeEnum;
4. /\*\*
5. \*
6. \*作者：alaric
7. \*时间：2013-8-5上午11:14:43
8. \*描述：手机充值
9. \*/
10. **public** **class** MobileStrategy **implements** Strategy {
12. @Override
13. **public** Double calRecharge(Double charge, RechargeTypeEnum type) {
14. // TODO Auto-generated method stub
15. **return** charge;
16. }
18. }

**Java代码  [收藏代码](javascript:void())**

1. **package** strategy.strategy;
3. **import** strategy.RechargeTypeEnum;
4. /\*\*
5. \*
6. \*作者：alaric
7. \*时间：2013-8-5上午11:13:46
8. \*描述：充值卡充值
9. \*/
10. **public** **class** CardStrategy **implements** Strategy{
12. @Override
13. **public** Double calRecharge(Double charge, RechargeTypeEnum type) {
14. **return** charge+charge\*0.01;
15. }
17. }

**Java代码  [收藏代码](javascript:void())**

1. **package** strategy.strategy;
3. **import** strategy.RechargeTypeEnum;
5. /\*\*
6. \*
7. \*作者：alaric
8. \*时间：2013-8-5上午11:03:38
9. \*描述：场景类
10. \*/
11. **public** **class** Context {
13. **private** Strategy strategy;
15. **public** Double calRecharge(Double charge, Integer type) {
16. strategy = StrategyFactory.getInstance().creator(type);
17. **return** strategy.calRecharge(charge, RechargeTypeEnum.valueOf(type));
18. }
20. **public** Strategy getStrategy() {
21. **return** strategy;
22. }
24. **public** **void** setStrategy(Strategy strategy) {
25. **this**.strategy = strategy;
26. }
28. }

**Java代码  [收藏代码](javascript:void())**

1. **package** strategy.strategy;
3. **import** java.util.HashMap;
4. **import** java.util.Map;
6. **import** strategy.RechargeTypeEnum;
7. /\*\*
8. \*
9. \*作者：alaric
10. \*时间：2013-8-5上午11:31:12
11. \*描述：策略工厂 使用单例模式
12. \*/
13. **public** **class** StrategyFactory {
15. **private** **static** StrategyFactory factory = **new** StrategyFactory();
16. **private** StrategyFactory(){
17. }
18. **private** **static** Map<Integer ,Strategy> strategyMap = **new** HashMap<>();
19. **static**{
20. strategyMap.put(RechargeTypeEnum.E\_BANK.value(), **new** EBankStrategy());
21. strategyMap.put(RechargeTypeEnum.BUSI\_ACCOUNTS.value(), **new** BusiAcctStrategy());
22. strategyMap.put(RechargeTypeEnum.MOBILE.value(), **new** MobileStrategy());
23. strategyMap.put(RechargeTypeEnum.CARD\_RECHARGE.value(), **new** CardStrategy());
24. }
25. **public** Strategy creator(Integer type){
26. **return** strategyMap.get(type);
27. }
28. **public** **static** StrategyFactory getInstance(){
29. **return** factory;
30. }
31. }

**Java代码  [收藏代码](javascript:void())**

1. **package** strategy.strategy;
3. **import** strategy.RechargeTypeEnum;
5. **public** **class** Client {
7. /\*\*
8. \* 作者：alaric 时间：2013-8-5上午11:33:52 描述：
9. \*/
10. **public** **static** **void** main(String[] args) {
12. Context context = **new** Context();
13. // 网银充值100 需要付多少
14. Double money = context.calRecharge(100D,
15. RechargeTypeEnum.E\_BANK.value());
16. System.out.println(money);
18. // 商户账户充值100 需要付多少
19. Double money2 = context.calRecharge(100D,
20. RechargeTypeEnum.BUSI\_ACCOUNTS.value());
21. System.out.println(money2);
23. // 手机充值100 需要付多少
24. Double money3 = context.calRecharge(100D,
25. RechargeTypeEnum.MOBILE.value());
26. System.out.println(money3);
28. // 充值卡充值100 需要付多少
29. Double money4 = context.calRecharge(100D,
30. RechargeTypeEnum.CARD\_RECHARGE.value());
31. System.out.println(money4);
32. }
34. }

运行结果：

85.0

90.0

100.0

101.0

从上面类图和代码可以看出，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。

**设计模式系列目录：**

[JAVA设计模式学习1——统一建模语言UML](http://alaric.iteye.com/blog/1722197)

[JAVA设计模式学习2——面向对象设计原则](http://alaric.iteye.com/blog/1722198)

[JAVA设计模式学习3——设计模式概述](http://alaric.iteye.com/blog/1722199)

[JAVA设计模式学习4——简单工厂模式](http://alaric.iteye.com/blog/1908619)

[JAVA设计模式学习5——工厂方法模式](http://alaric.iteye.com/blog/1908876)

[JAVA设计模式学习6——抽象工厂模式](http://alaric.iteye.com/blog/1908902)

[JAVA设计模式学习7——单例模式](http://alaric.iteye.com/blog/1909007)

[JAVA设计模式学习8——建造模式](http://alaric.iteye.com/blog/1909867)

[JAVA设计模式学习9——原型模式](http://alaric.iteye.com/blog/1909960)

[JAVA设计模式学习10——组合模式](http://alaric.iteye.com/blog/1910919)

[JAVA设计模式学习11——适配器模式](http://alaric.iteye.com/blog/1911161)

[JAVA设计模式学习12——装饰器模式](http://alaric.iteye.com/blog/1911980)

[JAVA设计模式学习13——代理模式](http://alaric.iteye.com/blog/1913048)

[JAVA设计模式学习14——享元模式](http://alaric.iteye.com/blog/1914498)

[JAVA设计模式学习15——门面模式](http://alaric.iteye.com/blog/1915854)

[JAVA设计模式学习16——桥梁模式](http://alaric.iteye.com/blog/1918381)

[JAVA设计模式学习17——策略模式](http://alaric.iteye.com/blog/1920714)

[JAVA设计模式学习18——模版方法模式](http://alaric.iteye.com/blog/1922820)

[JAVA设计模式学习19——观察者模式](http://alaric.iteye.com/blog/1924169)

[JAVA设计模式学习20——责任链模式](http://alaric.iteye.com/blog/1926447)

[JAVA设计模式学习21——命令模式](http://alaric.iteye.com/blog/1927894)

[JAVA设计模式学习22——备忘录模式](http://alaric.iteye.com/blog/1931253)

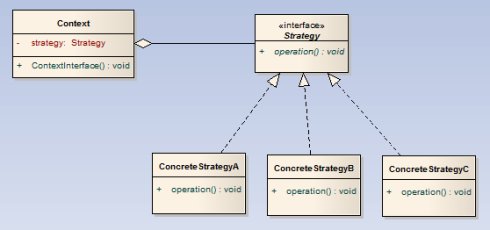
[JAVA设计模式学习23——状态模式](http://alaric.iteye.com/blog/1938400)

[JAVA设计模式学习24——访问者模式](http://alaric.iteye.com/blog/1942517)

[JAVA设计模式学习25——迭代器模式](http://alaric.iteye.com/blog/1978409)

# JAVA设计模式:策略模式(你还在使用IF ELSE IF吗？)\_Scoffied\_新浪博客

http://blog.sina.com.cn/s/blog\_4b81125f0101m7va.html

**策略（Strategy）模式**：又名Policy，它的用意是定义一组算法，把它们一个个封装起来，并且使他们可以相互替换。策略模式可以独立于使用他们的客户端而变化。GOF策略模式静态结构类图如下：  


通过上图可以看出策略模式有以下角色构成：

1、抽象策略（Strategy）角色：抽象策略角色由抽象类或接口来承担，它给出具体策略角色需要实现的接口；

2、具体策略（ConcreteStrategy）角色：实现封装了具体的算法或行为；

3、场景（Context）角色：持有抽象策略类的引用。

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。

策略模式在程序设计中也很常用，在板桥（banq）的博客里有篇文章叫 “你还在用if else吗？”  
“<http://www.jdon.com/artichect/ifelse.htm>”讲的很好，策略模式不但是继承的代替方案而且能很好地解决if else问题，下面举个实例来说明，怎么使用策略模式。

需求如下：

某支付系统接入以下几种商户进行充值：易宝网易，快线网银，19pay手机支付，支付宝支付，骏网一卡通，由于每家充值系统的结算比例不一样，而且 同一家商户的不同充值方式也有所不同，具体系统情况比较复杂，像支付宝既有支付宝账号支付和支付宝网银支付等这些暂时不考虑，为了讲述策略模式这里简单描 述，假如分为四种，手机支付，网银支付，商户账号支付和点卡支付。因为没个支付结算比例不同，所以对手续费低的做一些优惠活动，尽可能让用户使用手续费低 的支付方式来充值，这样降低渠道费用，增加收入，具体优惠政策如下：

①网银充值，8.5折；

②商户充值，9折；

③手机充值，没有优惠；

④点卡充值，收取1%的渠道费；

对于一个新手的代码如下：

Java代码

package strategy;

public class Example {

    public Double calRecharge(Double charge ,RechargeTypeEnum type ){

       if(type.equals(RechargeTypeEnum.E\_BANK)){

           return charge\*0.85;

       }else if(type.equals(RechargeTypeEnum.BUSI\_ACCOUNTS)){

           return charge\*0.90;

       }else if(type.equals(RechargeTypeEnum.MOBILE)){

           return charge;

       }else if(type.equals(RechargeTypeEnum.CARD\_RECHARGE)){

           return charge+charge\*0.01;

       }else{

           return null;

       }

    }

}

Java代码

package strategy;

public enum RechargeTypeEnum {

    E\_BANK(1, "网银"),

    BUSI\_ACCOUNTS(2, "商户账号"),

    MOBILE(3,"手机卡充值"),

    CARD\_RECHARGE(4,"充值卡")

    ;

    private int value;

    private String description;

    private RechargeTypeEnum(int value, String description) {

       this.value = value;

       this.description = description;

    }

    public int value() {

       return value;

    }

    public String description() {

       return description;

    }

    public static RechargeTypeEnum valueOf(int value) {

        for(RechargeTypeEnum type : RechargeTypeEnum.values()) {

            if(type.value() == value) {

                return type;

            }

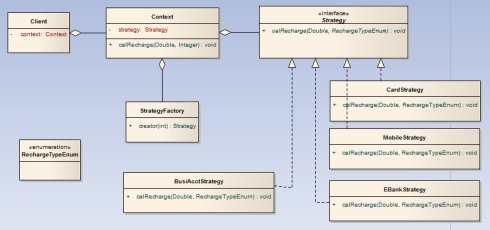
        }

        return null;

    }

}

 可以看出上面四种不同的计算方式在一个方法内部，不利于扩展和维护，当然也不符合面向对象设计原则。对以上的代码利用策略模式进行修改，类图如下：

  
 实例代码如下：

Java代码

package strategy.strategy;

import strategy.RechargeTypeEnum;

public interface Strategy {

    public Double calRecharge(Double charge ,RechargeTypeEnum type );

}

Java代码

package strategy.strategy;

import strategy.RechargeTypeEnum;

public class EBankStrategy implements Strategy{

    @Override

    public Double calRecharge(Double charge, RechargeTypeEnum type) {

       return charge\*0.85;

    }

}

Java代码

package strategy.strategy;

import strategy.RechargeTypeEnum;

public class BusiAcctStrategy implements Strategy{

    @Override

    public Double calRecharge(Double charge, RechargeTypeEnum type) {

       // TODO Auto-generated method stub

       return charge\*0.90;

    }

}

Java代码

package strategy.strategy;

import strategy.RechargeTypeEnum;

public class MobileStrategy implements Strategy {

    @Override

    public Double calRecharge(Double charge, RechargeTypeEnum type) {

       // TODO Auto-generated method stub

       return charge;

    }

}

Java代码

package strategy.strategy;

import strategy.RechargeTypeEnum;

public class CardStrategy implements Strategy{

    @Override

    public Double calRecharge(Double charge, RechargeTypeEnum type) {

       return charge+charge\*0.01;

    }

}

Java代码

package strategy.strategy;

import strategy.RechargeTypeEnum;

public class Context {

    private Strategy strategy;

    public Double calRecharge(Double charge, Integer type) {

       strategy = StrategyFactory.getInstance().creator(type);

       return strategy.calRecharge(charge, RechargeTypeEnum.valueOf(type));

    }

    public Strategy getStrategy() {

       return strategy;

    }

    public void setStrategy(Strategy strategy) {

       this.strategy = strategy;

    }

}

Java代码

package strategy.strategy;

import java.util.HashMap;

import java.util.Map;

import strategy.RechargeTypeEnum;

public class StrategyFactory {

    private static StrategyFactory factory = new StrategyFactory();

    private StrategyFactory(){

    }

    private static Map strategyMap = new HashMap<>();

    static{

       strategyMap.put(RechargeTypeEnum.E\_BANK.value(), new EBankStrategy());

       strategyMap.put(RechargeTypeEnum.BUSI\_ACCOUNTS.value(), new BusiAcctStrategy());

       strategyMap.put(RechargeTypeEnum.MOBILE.value(), new MobileStrategy());

       strategyMap.put(RechargeTypeEnum.CARD\_RECHARGE.value(), new CardStrategy());

    }

    public Strategy creator(Integer type){

       return strategyMap.get(type);

    }

    public static StrategyFactory getInstance(){

       return factory;

    }

}

Java代码

package strategy.strategy;

import strategy.RechargeTypeEnum;

public class Client {

    public static void main(String[] args) {

       Context context = new Context();

       // 网银充值100 需要付多少

       Double money = context.calRecharge(100D,

              RechargeTypeEnum.E\_BANK.value());

       System.out.println(money);

       // 商户账户充值100 需要付多少

       Double money2 = context.calRecharge(100D,

              RechargeTypeEnum.BUSI\_ACCOUNTS.value());

       System.out.println(money2);

       // 手机充值100 需要付多少

       Double money3 = context.calRecharge(100D,

              RechargeTypeEnum.MOBILE.value());

       System.out.println(money3);

       // 充值卡充值100 需要付多少

       Double money4 = context.calRecharge(100D,

              RechargeTypeEnum.CARD\_RECHARGE.value());

       System.out.println(money4);

    }

}

运行结果：

85.0

90.0

100.0

101.0

从上面类图和代码可以看出，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。

 转自：http://alaric.iteye.com/blog/1920714