# 策略（Strategy）设计模式

目录

[策略（Strategy）设计模式 1](#_Toc26283990)

[一、 模式讲解 2](#_Toc26283991)

[认识策略模式 2](#_Toc26283992)

[**(1)、 策略模式的功能** 2](#_Toc26283993)

[**(2)、 策略模式和if-else语句** 2](#_Toc26283994)

[**(3)、 算法的平等性** 2](#_Toc26283995)

[**(4)、 谁来选择具体的策略算法** 2](#_Toc26283996)

[**(5)、 Strategy的实现方式** 2](#_Toc26283997)

[**(6)、 运行时策略的唯一性** 2](#_Toc26283998)

[**(7)、 增加新的策略** 2](#_Toc26283999)

[**(8)、 策略模式调用顺序示意图** 3](#_Toc26284000)

[容错恢复机制 4](#_Toc26284001)

[二、 场景一（报价管理） 6](#_Toc26284002)

[(一)、 不用模式的解决方案 7](#_Toc26284003)

[(二)、 有何问题 7](#_Toc26284004)

[(三)、 策略模式来解决 9](#_Toc26284005)

[(四)、 策略模式示例代码 10](#_Toc26284006)

[使用策略模式重写示例 12](#_Toc26284007)

[扩展示例，实现方式一 14](#_Toc26284008)

[扩展示例，实现方式二 16](#_Toc26284009)

[策略模式结合模板方法模式 17](#_Toc26284010)

[三、 总结 20](#_Toc26284011)

[四、 相关模式 22](#_Toc26284012)

策略（Strategy）模式就是定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。

## 模式讲解

### 认识策略模式

1. **策略模式的功能**

策略模式的功能是把具体的算法实现，从具体的业务处理里面独立出来，把它们实现成为单独的算法类，从而形成一系列的算法，并让这些算法可以相互替换。

策略模式的重心不是如何来实现算法，而是如何组织、调用这些算法，从而让程序结构更灵活、具有更好的维护性和扩展性。

1. **策略模式和if-else语句**

看了前面的示例，很多朋友会发现，每个策略算法具体实现的功能，就是原来在if-else结构中的具体实现。

没错，其实多个if-elseif语句表达的就是一个平等的功能结构，你要么执行if，要不你就执行else，或者是else if，这个时候，if块里面的实现和else块里面的实现从运行地位上来讲就是平等的。

而策略模式就是把各个平等的具体实现封装到单独的策略实现类了，然后通过上下文来与具体的策略类进行交互。

因此多个if-else语句可以考虑使用策略模式。

1. **算法的平等性**

策略模式一个很大的特点就是各个策略算法的平等性。对于一系列具体的策略算法，大家的地位是完全一样的，正是因为这个平等性，才能实现算法之间可以相互替换。

所有的策略算法在实现上也是相互独立的，相互之间是没有依赖的。

所以可以这样描述这一系列策略算法：**策略算法是相同行为的不同实现**。

1. **谁来选择具体的策略算法**

在策略模式中，可以在两个地方来进行具体策略的选择。

一个是在客户端，在使用上下文的时候，由客户端来选择具体的策略算法，然后把这个策略算法设置给上下文。前面的示例就是这种情况。

还有一个是客户端不管，由上下文来选择具体的策略算法，这个在后面讲容错恢复的时候给大家演示一下。

1. **Strategy的实现方式**

在前面的示例中，Strategy都是使用的接口来定义的，这也是常见的实现方式。但是如果多个算法具有公共功能的话，可以把Strategy实现成为抽象类，然后把多个算法的公共功能实现到Strategy里面。

1. **运行时策略的唯一性**

运行期间，策略模式在每一个时刻只能使用一个具体的策略实现对象，虽然可以动态的在不同的策略实现中切换，但是同时只能使用一个。

1. **增加新的策略**

在前面的示例里面，体会到了策略模式中切换算法的方便，但是增加一个新的算法会怎样呢？比如现在要实现如下的功能：对于公司的“战略合作客户”，统一8折。

其实很简单，策略模式可以让你很灵活的扩展新的算法。具体的做法是：先写一个策略算法类来实现新的要求，然后在客户端使用的时候指定使用新的策略算法类就可以了。

 还是通过示例来说明。先添加一个实现要求的策略类，示例代码如下：

*/\*\**

*\* 具体算法实现，为战略合作客户客户计算应报的价格*

*\*/*

public class CooperateCustomerStrategy implements Strategy{

public double calcPrice(double *goodsPrice*) {

System.out.println("对于战略合作客户，统一8折");

return goodsPrice\*0.8;

}

}

然后在客户端指定使用策略的时候指定新的策略算法实现，示例如下：

public class Client2 {

public static void main(String[] *args*) {

*//1：选择并创建需要使用的策略对象*

***Strategy strategy = new CooperateCustomerStrategy ();***

*//2：创建上下文*

Price ctx = new Price(strategy);

*//3：计算报价*

double quote = ctx.quote(1000);

System.out.println("向客户报价："+quote);

}

}

除了加粗部分变动外，客户端没有其他的变化。

除了客户端发生变化外，已有的上下文、策略接口定义和策略的已有实现，都不需要做任何的修改，可见能很方便的扩展新的策略算法。

1. **策略模式调用顺序示意图**

策略模式的调用顺序，有两种常见的情况，一种如同前面的示例，具体如下：

先是客户端来选择并创建具体的策略对象

然后客户端创建上下文

接下来客户端就可以调用上下文的方法来执行功能了，在调用的时候，从客户端传入算法需要的参数

上下文接到客户的调用请求，会把这个请求转发给它持有的Strategy

这种情况的调用顺序示意图如图3所示：

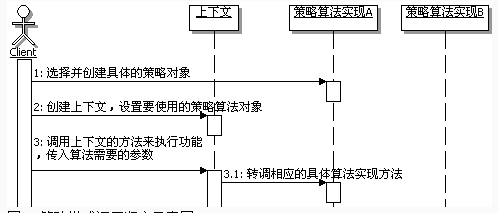


图 1策略模式调用顺序示意图

策略模式调用还有一种情况，就是把Context当做参数来传递给Strategy，这种方式的调用顺序图，在讲具体的Context和Strategy的关系时再给出。

### 容错恢复机制

容错恢复机制是应用程序开发中非常常见的功能。那么什么是容错恢复呢？简单点说就是：程序运行的时候，正常情况下应该按照某种方式来做，如果按照某种方式来做发生错误的话，系统并不会崩溃，也不会就此不能继续向下运行了，而是有容忍出错的能力，不但能容忍程序运行出现错误，还提供出现错误后的备用方案，也就是恢复机制，来代替正常执行的功能，使程序继续向下运行。

举个实际点的例子吧，比如在一个系统中，所有对系统的操作都要有日志记录，而且这个日志还需要有管理界面，这种情况下通常会把日志记录在数据库里面，方便后续的管理，但是在记录日志到数据库的时候，可能会发生错误，比如暂时连不上数据库了，那就先记录在文件里面，然后在合适的时候把文件中的记录再转录到数据库中。

对于这样的功能的设计，就可以采用策略模式，把日志记录到数据库和日志记录到文件当作两种记录日志的策略，然后在运行期间根据需要进行动态的切换。

在这个例子的实现中，要示范由上下文来选择具体的策略算法，前面的例子都是由客户端选择好具体的算法，然后设置到上下文中。

下面还是通过代码来示例一下。

1. 日志策略接口

*/\*\**

*\* 日志记录策略的接口*

*\*/*

public interface LogStrategy {

*/\*\**

*\* 记录日志*

*\* @param msg 需记录的日志信息*

*\*/*

public void log(String *msg*);

}

1. 日志策略接口实现

*/\*\**

*\* 把日志记录到数据库*

*\*/*

public class DbLog implements LogStrategy{

public void log(String *msg*) {

*//制造错误*

if(msg!=null && msg.trim().length()>5){

int a = 5/0;

}

System.out.println("现在把 '"+msg+"' 记录到数据库中");

}

}

*/\*\**

*\* 把日志记录到文件*

*\*/*

public class FileLog implements LogStrategy{

public void log(String *msg*) {

System.out.println("现在把 '"+msg+"' 记录到文件中");

}

}

1. 定义使用这些策略的上下文

*/\*\**

*\* 记录日志的上下文*

*\*/*

public class LogContext {

*/\*\**

*\* 记录日志的方法，提供给客户端使用*

*\**

*\* @param msg 需要记录的日志信息*

*\*/*

public void log(String *msg*) {

*//在上下文里，自行实现对具体策略的选择*

*//优先选用策略：记录到数据库*

LogStrategy strategy = new DbLog();

try {

strategy.log(msg);

} catch (Exception *e*) {

*//出错了，就记录到文件中*

strategy = new FileLog();

strategy.pay(msg);

}

}

}

1. 客户端

public class Client {

public static void main(String[] *args*) {

LogContext log = new LogContext();

log.log("记录日志");

log.log("再次记录日志");

}

}

运行结果：

现在把'记录日志'记录到数据库中

现在把'再次记录日志'记录到文件中

小结一下，通过上面的示例，会看到策略模式的一种简单应用，也顺便了解一下基本的容错恢复机制的设计和实现。在实际的应用中，需要设计容错恢复的系统一般要求都比较高，应用也会比较复杂，但是基本的思路是差不多的。

## 场景一（报价管理）

向客户报价，对于销售部门的人来讲，这是一个非常重大、非常复杂的问题，对不同的客户要报不同的价格，比如：

对普通客户或者是新客户报的是全价

对老客户报的价格，根据客户年限，给予一定的折扣

对大客户报的价格，根据大客户的累计消费金额，给予一定的折扣

还要考虑客户购买的数量和金额，比如：虽然是新用户，但是一次购买的数量非常大，或者是总金额非常高，也会有一定的折扣，还有，报价人员的职务高低，也决定了他是否有权限对价格进行一定的浮动折扣，甚至在不同的阶段，对客户的报价也不同，一般情况是刚开始比较高，越接近成交阶段，报价越趋于合理。

总之，向客户报价是非常复杂的，因此在一些CRM（客户关系管理）的系统中，会有一个单独的报价管理模块，来处理复杂的报价功能。

为了演示的简洁性，假定现在需要实现一个简化的报价管理，实现如下的功能：

（1）对普通客户或者是新客户报全价

（2）对老客户报的价格，统一折扣5%

（3）对大客户报的价格，统一折扣10%

该怎么实现呢？

1. 不用模式的解决方案

要实现对不同的人员报不同的价格的功能，无外乎就是判断起来麻烦点，也不多难，很快就有朋友能写出如下的实现代码，示例代码如下：

*/\*\**

*\* 价格管理，主要完成计算向客户所报价格的功能*

*\*/*

public class Price {

*/\*\**

*\* 报价，对不同类型的，计算不同的价格*

*\* @param goodsPrice 商品销售原价*

*\* @param customerType 客户类型*

*\* @return 计算出来的，应该给客户报的价格*

*\*/*

public double quote(double *goodsPrice*,String *customerType*){

if(customerType.equals("普通客户 ")){

System.out.println("对于新客户或者是普通客户，没有折扣 ");

return goodsPrice;

}else if(customerType.equals("老客户 ")){

System.out.println("对于老客户，统一折扣 5%");

return goodsPrice\*(1-0.05);

}else if(customerType.equals("大客户 ")){

System.out.println("对于大客户，统一折扣 10%");

return goodsPrice\*(1-0.1);

}

*//其余人员都是报原价*

return goodsPrice;

}

}

1. 有何问题

上面的写法是很简单的，也很容易想，但是仔细想想，这样实现，问题可不小，比如：

第一个问题：价格类包含了所有计算报价的算法，使得价格类，尤其是报价这个方法比较庞杂，难以维护。

        有朋友可能会想，这很简单嘛，把这些算法从报价方法里面拿出去，形成独 立的方法不就可以解决这个问题了吗？据此写出如下的实现代码，示例代码如下：

*/\*\**

*\* 价格管理，主要完成计算向客户所报价格的功能*

*\*/*

public class Price {

*/\*\**

*\* 报价，对不同类型的，计算不同的价格*

*\* @param goodsPrice 商品销售原价*

*\* @param customerType 客户类型*

*\* @return 计算出来的，应该给客户报的价格*

*\*/*

public double quote(double *goodsPrice*,String *customerType*){

if(customerType.equals("普通客户 ")){

return this.calcPriceForNormal(goodsPrice);

}else if(customerType.equals("老客户 ")){

return this.calcPriceForOld(goodsPrice);

}else if(customerType.equals("大客户 ")){

return this.calcPriceForLarge(goodsPrice);

}

*//其余人员都是报原价*

return goodsPrice;

}

*/\*\**

*\* 为新客户或者是普通客户计算应报的价格*

*\* @param goodsPrice 商品销售原价*

*\* @return 计算出来的，应该给客户报的价格*

*\*/*

private double calcPriceForNormal(double *goodsPrice*){

System.out.println("对于新客户或者是普通客户，没有折扣 ");

return goodsPrice;

}

*/\*\**

*\* 为老客户计算应报的价格*

*\* @param goodsPrice 商品销售原价*

*\* @return 计算出来的，应该给客户报的价格*

*\*/*

private double calcPriceForOld(double *goodsPrice*){

System.out.println("对于老客户，统一折扣 5%");

return goodsPrice\*(1-0.05);

}

*/\*\**

*\* 为大客户计算应报的价格*

*\* @param goodsPrice 商品销售原价*

*\* @return 计算出来的，应该给客户报的价格*

*\*/*

private double calcPriceForLarge(double *goodsPrice*){

System.out.println("对于大客户，统一折扣 10%");

return goodsPrice\*(1-0.1);

}

}

这样看起来，比刚开始稍稍好点，计算报价的方法会稍稍简单一点，这样维护起来也稍好一些，某个算法发生了变化，直接修改相应的私有方法就可以了。扩展起来也容易一点，比如要增加一个“战略合作客户”的类型，报价为直接8折，就只需要在价格类里面新增加一个私有的方法来计算新的价格，然后在计算报价的方法里面新添一个else-if即可。看起来似乎很不错了。

真的很不错了吗？

再想想，问题还是存在，只不过从计算报价的方法挪动到价格类里面了，假如有100个或者更多这样的计算方式，这会让这个价格类非常庞大，难以维护。而且，维护和扩展都需要去修改已有的代码，这是很不好的，违反了开-闭原则。

第二个问题：经常会有这样的需要，在不同的时候，要使用不同的计算方式。

        比如：在公司周年庆的时候，所有的客户额外增加3%的折扣；在换季促销的时候，普通客户是额外增加折扣2%，老客户是额外增加折扣3%，大客户是额外增加折扣5%。这意味着计算报价的方式会经常被修改，或者被切换。   
        通常情况下应该是被切换，因为过了促销时间，又还回到正常的价格体系上来了。而现在的价格类中计算报价的方法，是固定调用各种计算方式，这使得切换调用不同的计算方式很麻烦，每次都需要修改if-else里面的调用代码。   
        看到这里，可能有朋友会想， 那么到底应该如何实现，才能够让价格类中的计算报价的算法，能很容易的实现可维护、可扩展，又能动态的切换变化呢？

1. 策略模式来解决

仔细分析上面的问题，先来把它抽象一下，各种计算报价的计算方式就好比是具体的算法，而使用这些计算方式来计算报价的程序，就相当于是使用算法的客户。

再分析上面的实现方式，为什么会造成那些问题，根本原因，就在于算法和使用算法的客户是耦合的，甚至是密不可分的，在上面实现中，具体的算法和使用算法的客户是同一个类里面的不同方法。

现在要解决那些问题，按照策略模式的方式，应该先把所有的计算方式独 立出来，每个计算方式做成一个单独的算法类，从而形成一系列的算法，并且为这一系列算法定义一个公共的接口，这些算法实现是同一接口的不同实现，地位是平等的，可以相互替换。这样一来，要扩展新的算法就变成了增加一个新的算法实现类，要维护某个算法，也只是修改某个具体的算法实现即可，不会对其它代码造成影响。也就是说这样就解决了可维护、可扩展的问题。

为了实现让算法能独 立于使用它的客户，策略模式引入了一个上下文的对象，这个对象负责持有算法，但是不负责决定具体选用哪个算法，把选择算法的功能交给了客户，由客户选择好具体的算法后，设置到上下文对象里面，让上下文对象持有客户选择的算法，当客户通知上下文对象执行功能的时候，上下文对象会去转调具体的算法。这样一来，具体的算法和直接使用算法的客户是分离的。

具体的算法和使用它的客户分离过后，使得算法可独 立于使用它的客户而变化，并且能够动态的切换需要使用的算法，只要客户端动态的选择使用不同的算法，然后设置到上下文对象中去，实际调用的时候，就可以调用到不同的算法。

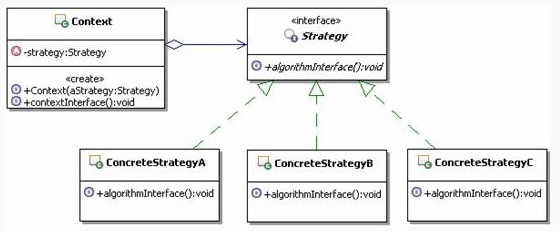


图 2 策略模式结构示意图

**Strategy：**

策略接口，用来约束一系列具体的策略算法。Context使用这个接口来调用具体的策略实现定义的算法。

**ConcreteStrategy：**

具体的策略实现，也就是具体的算法实现。

**Context：**

上下文，负责和具体的策略类交互，通常上下文会持有一个真正的策略实现，上下文还可以让具体的策略类来获取上下文的数据，甚至让具体的策略类来回调上下文的方法。

1. 策略模式示例代码
2. 定义算法的接口

*/\*\**

*\* 策略，定义算法的接口*

*\*/*

public interface Strategy {

*/\*\**

*\* 某个算法的接口，可以有传入参数，也可以有返回值*

*\*/*

public void algorithmInterface();

}

1. 算法实现

ConcreteStrategyA.java

*/\*\**

*\* 实现具体的算法*

*\*/*

public class ConcreteStrategyA implements Strategy {

public void algorithmInterface() {

*//具体的算法实现*

}

}

ConcreteStrategyB.java

*/\*\**

*\* 实现具体的算法*

*\*/*

public class ConcreteStrategyB implements Strategy {

public void algorithmInterface() {

*//具体的算法实现*

}

}

ConcreteStrategyC.java

*/\*\**

*\* 实现具体的算法*

*\*/*

public class ConcreteStrategyC implements Strategy {

public void algorithmInterface() {

*//具体的算法实现*

}

}

1. 上下文实现

*/\*\**

*\* 上下文对象，通常会持有一个具体的策略对象*

*\*/*

public class Context {

*/\*\**

*\* 持有一个具体的策略对象*

*\*/*

private Strategy strategy;

*/\*\**

*\* 构造方法，传入一个具体的策略对象*

*\* @param aStrategy 具体的策略对象*

*\*/*

public Context(Strategy *aStrategy*) {

this.strategy = aStrategy;

}

*/\*\**

*\* 上下文对客户端提供的操作接口，可以有参数和返回值*

*\*/*

public void contextInterface() {

*//通常会转调具体的策略对象进行算法运算*

strategy.algorithmInterface();

}

}

### 使用策略模式重写示例

要使用策略模式来重写前面报价的示例，大致有如下改变:

首先需要定义出算法的接口。

然后把各种报价的计算方式单独出来，形成算法类。

对于Price这个类，把它当做上下文，在计算报价的时候，不再需要判断，直接使用持有的具体算法进行运算即可。选择使用哪一个算法的功能挪出去，放到外部使用的客户端去。

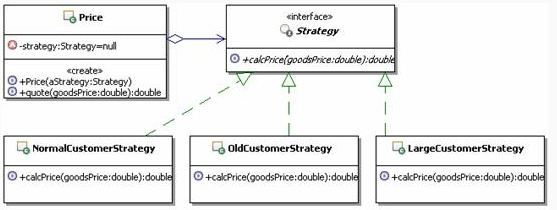


图 3使用策略模式实现示例的结构示意图

1. 策略接口

*/\*\**

*\* 策略，定义计算报价算法的接口*

*\*/*

public interface Strategy {

*/\*\**

*\* 计算应报的价格*

*\* @param goodsPrice 商品销售原价*

*\* @return 计算出来的，应该给客户报的价格*

*\*/*

public double calcPrice(double *goodsPrice*);

}

1. 具体的算法实现

新客户或者是普通客户计算应报的价格的实现

*/\*\**

*\* 具体算法实现，为新客户或者是普通客户计算应报的价格*

*\*/*

public class NormalCustomerStrategy implements Strategy{

public double calcPrice(double *goodsPrice*) {

System.out.println("对于新客户或者是普通客户，没有折扣");

return goodsPrice;

}

}

老客户计算应报的价格的实现

*/\*\**

*\* 具体算法实现，为老客户计算应报的价格*

*\*/*

public class OldCustomerStrategy implements Strategy{

public double calcPrice(double *goodsPrice*) {

System.out.println("对于老客户，统一折扣5%");

return goodsPrice\*(1-0.05);

}

}

大客户计算应报的价格的实现

*/\*\**

*\* 具体算法实现，为大客户计算应报的价格*

*\*/*

public class LargeCustomerStrategy implements Strategy{

public double calcPrice(double *goodsPrice*) {

System.out.println("对于大客户，统一折扣10%");

return goodsPrice\*(1-0.1);

}

}

1. 上下文的实现

**上下文的实现，也就是原来的价格类，它的变化比较大，主要有：**

原来那些私有的，用来做不同计算的方法，已经去掉了，独 立出去做成了算法类

原来报价方法里面，对具体计算方式的判断，去掉了，让客户端来完成选择具体算法的功能

新添加持有一个具体的算法实现，通过构造方法传入

原来报价方法的实现，变化成了转调具体算法来实现

*/\*\**

*\* 价格管理，主要完成计算向客户所报价格的功能*

*\*/*

public class Price {

*/\*\**

*\* 持有一个具体的策略对象*

*\*/*

private Strategy strategy = null;

*/\*\**

*\* 构造方法，传入一个具体的策略对象*

*\* @param aStrategy 具体的策略对象*

*\*/*

public Price(Strategy *aStrategy*){

this.strategy = aStrategy;

}

*/\*\**

*\* 报价，计算对客户的报价*

*\* @param goodsPrice 商品销售原价*

*\* @return 计算出来的，应该给客户报的价格*

*\*/*

public double quote(double *goodsPrice*){

return this.strategy.calcPrice(goodsPrice);

}

}

1. 测试

public class Client {

public static void main(String[] *args*) {

*//1：选择并创建需要使用的策略对象*

Strategy strategy = new LargeCustomerStrategy ();

*//2：创建上下文*

Price ctx = new Price(strategy);

*//3：计算报价*

double quote = ctx.quote(1000);

System.out.println("向客户报价："+quote);

}

}

### 扩展示例，实现方式一

经过上面的测试可以看出，通过使用策略模式，已经实现好了两种支付方式了。如果现在要增加一种支付方式，要求能支付到银行卡，该怎么扩展最简单呢？

应该新增加一种支付到银行卡的策略实现，然后通过继承来扩展支付上下文，在里面添加新的支付方式需要的新的数据，比如银行卡账户，然后在客户端使用新的上下文和新的策略实现就可以了，这样已有的实现都不需要改变，完全遵循开-闭原则。

先看看扩展的支付上下文对象的实现，示例代码如下：

*/\*\**

*\* 扩展的支付上下文对象*

*\*/*

public class PaymentContext2 extends PaymentContext {

*/\*\**

*\* 银行帐号*

*\*/*

private String account = null;

*/\*\**

*\* 构造方法，传入被支付工资的人员，应支付的金额和具体的支付策略*

*\* @param userName 被支付工资的人员*

*\* @param money 应支付的金额*

*\* @param account 支付到的银行帐号*

*\* @param strategy 具体的支付策略*

*\*/*

public PaymentContext2(String *userName*,double *money*,

String *account*,PaymentStrategy *strategy*){

super(userName,money,strategy);

this.account = account;

}

public String getAccount() {

return account;

}

}

然后看看新的策略算法的实现，示例代码如下：

*/\*\**

*\* 支付到银行卡*

*\*/*

public class Card implements PaymentStrategy{

public void pay(PaymentContext *ctx*) {

*// 这个新的算法自己知道要使用扩展的支付上下文，所以强制造型一下*

PaymentContext2 ctx2 = (PaymentContext2)ctx;

System.out.println(" 现在给 "+ctx2.getUserName()+" 的 "

+ctx2.getAccount()+" 帐号支付了 "+ctx2.getMoney()+" 元 ");

*// 连接银行，进行转帐，就不去管了*

}

}

最后看看客户端怎么使用这个新的策略呢？原有的代码不变，直接添加新的测试就可以了，示例代码如下：

public class Client {

public static void main(String[] *args*) {

*//创建相应的支付策略*

PaymentStrategy strategyRMB = new RMBCash();

PaymentStrategy strategyDollar = new DollarCash();

*//准备小李的支付工资上下文*

PaymentContext ctx1 =

new PaymentContext("小李 ",5000,strategyRMB);

*//向小李支付工资*

ctx1.payNow();

*//切换一个人，给 petter支付工资*

PaymentContext ctx2 =

new PaymentContext("Petter",8000,strategyDollar);

ctx2.payNow();

**// 测试新添加的支付方式**

**PaymentStrategy strategyCard = new Card();**

**PaymentContext ctx3 = new PaymentContext2("小王",9000,"010998877656",strategyCard);**

**ctx3.payNow();**

}

}

再次测试，体会一下，运行结果如下：

|  |
| --- |
| 现在给小李人民币现金支付 5000.0元  现在给 Petter美元现金支付 8000.0元  现在给小王的 010998877656帐号支付了 9000.0元 |

### 扩展示例，实现方式二

同样还是实现上面这个功能：现在要增加一种支付方式，要求能支付到银行卡。

上面这种实现方式，是通过扩展上下文对象来准备新的算法需要的数据。还有另外一种方式，那就是通过策略的构造方法来传入新算法需要的数据。这样实现的话，就不需要扩展上下文了，直接添加新的策略算法实现就好了。示例代码如下：

*/\*\**

*\* 支付到银行卡*

*\*/*

public class Card2 implements PaymentStrategy{

*/\*\**

*\* 帐号信息*

*\*/*

private String account = "";

*/\*\**

*\* 构造方法，传入帐号信息*

*\* @param account 帐号信息*

*\*/*

public Card2(String *account*){

this.account = account;

}

public void pay(PaymentContext *ctx*) {

System.out.println(" 现在给 "+ctx.getUserName()+" 的 "

+this.account+" 帐号支付了 "+ctx.getMoney()+" 元 ");

*// 连接银行，进行转帐，就不去管了*

}

}

直接在客户端测试就可以了，测试示例代码如下：

public class Client {

public static void main(String[] *args*) {

*//测试新添加的支付方式*

PaymentStrategy strategyCard2 = new Card2("010998877656");

PaymentContext ctx4 =

new PaymentContext("小张",9000,strategyCard2);

ctx4.payNow();

}

}

比较：

**对于扩展上下文的方式**：这样实现，所有策略的实现风格更统一，策略需要的数据都统一从上下文来获取，这样在使用方法上也很统一；另外，在上下文中添加新的数据，别的相应算法也可以用得上，可以视为公共的数据。但缺点也很明显，如果这些数据只有一个特定的算法来使用，那么这些数据有些浪费；另外每次添加新的算法都去扩展上下文，容易形成复杂的上下文对象层次，也未见得有必要。

**对于在策略算法的实现上添加自己需要的数据的方式**：这样实现，比较好想，实现简单。但是缺点也很明显，跟其它策略实现的风格不一致，其它策略都是从上下文中来获取数据，而这个策略的实现一部分数据来自上下文，一部分数据来自自己，有些不统一；另外，这样一来，外部使用这些策略算法的时候也不一样了，不太好以一个统一的方式来动态切换策略算法。   
        两种实现各有优劣，至于如何选择，那就具体问题，具体的分析了。

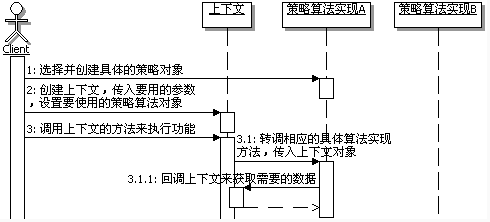


图 4策略模式调用顺序示意图

### 策略模式结合模板方法模式

在实际应用策略模式的过程中，经常会出现这样一种情况，就是发现这一系列算法的实现上存在公共功能，甚至这一系列算法的实现步骤都是一样的，只是在某些局部步骤上有所不同，这个时候，就需要对策略模式进行些许的变化使用了。

对于一系列算法的实现上存在公共功能的情况，策略模式可以有如下三种实现方式：

一个是在上下文当中实现公共功能，让所有具体的策略算法回调这些方法。

另外一种情况就是把策略的接口改成抽象类，然后在里面实现具体算法的公共功能。

还有一种情况是给所有的策略算法定义一个抽象的父类，让这个父类去实现策略的接口，然后在这个父类里面去实现公共的功能。

更进一步，如果这个时候发现“一系列算法的实现步骤都是一样的，只是在某些局部步骤上有所不同”的情况，那就可以在这个抽象类里面定义算法实现的骨架，然后让具体的策略算法去实现变化的部分。这样的一个结构自然就变成了策略模式来结合模板方法模式了，那个抽象类就成了模板方法模式的模板类。

在上一章我们讨论过模板方法模式来结合策略模式的方式，也就是主要的结构是模板方法模式，局部采用策略模式。而这里讨论的是策略模式来结合模板方法模式，也就是主要的结构是策略模式，局部实现上采用模板方法模式。通过这个示例也可以看出来，模式之间的结合是没有定势的，要具体问题具体分析。

此时策略模式结合模板方法模式的系统结构如下图所示：

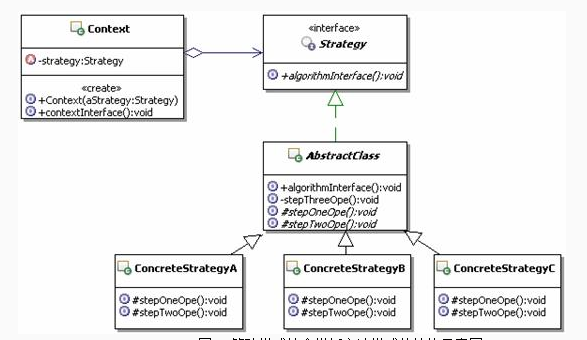


图 5 策略模式结合模板方法模式的结构示意图

还是用实际的例子来说吧，比如上面那个记录日志的例子，如果现在需要在所有的消息前面都添加上日志时间，也就是说现在记录日志的步骤变成了：第一步为日志消息添加日志时间；第二步具体记录日志。

1. 记录日志的策略接口没有变化，为了看起来方便，还是示例一下，示例代码如下：

*/\*\**

*\* 日志记录策略的接口*

*\*/*

public interface LogStrategy {

*/\*\**

*\* 记录日志*

*\* @param msg 需记录的日志信息*

*\*/*

public void log(String *msg*);

}

1. 增加一个实现这个策略接口的抽象类，在里面定义记录日志的算法骨架，相当于模板方法模式的模板，示例代码如下：

*/\*\**

*\* 实现日志策略的抽象模板，实现给消息添加时间*

*\*/*

public abstract class LogStrategyTemplate implements LogStrategy{

public final void log(String *msg*) {

*//第一步：给消息添加记录日志的时间*

DateFormat df = new SimpleDateFormat(

"yyyy-MM-dd HH:mm:ss SSS");

msg = df.format(new java.util.Date())+" 内容是："+ msg;

*//第二步：真正执行日志记录*

doLog(msg);

}

*/\*\**

*\* 真正执行日志记录，让子类去具体实现*

*\* @param msg 需记录的日志信息*

*\*/*

protected abstract void doLog(String *msg*);

}

1. 个时候那两个具体的日志算法实现也需要做些改变，不再直接实现策略接口了，而是继承模板，实现模板方法了。这个时候记录日志到数据库的类，示例代码如下：

*/\*\**

*\* 把日志记录到数据库*

*\*/*

public class DbLog extends LogStrategyTemplate{

*//除了定义上发生了改变外，具体的实现没变*

public void doLog(String *msg*) {

*//制造错误*

if(msg!=null && msg.trim().length()>5){

int a = 5/0;

}

System.out.println("现在把 '"+msg+"' 记录到数据库中");

}

}

同理实现记录日志到文件的类如下：

*/\*\**

*\* 把日志记录到数据库*

*\*/*

public class FileLog extends LogStrategyTemplate{

public void doLog(String *msg*) {

System.out.println("现在把 '"+msg+"' 记录到文件中");

}

}

算法实现的改变不影响使用算法的上下文，上下文跟前面一样，示例代码如下：

*/\*\**

*\* 日志记录的上下文*

*\*/*

public class LogContext {

*/\*\**

*\* 记录日志的方法，提供给客户端使用*

*\* @param msg 需记录的日志信息*

*\*/*

public void log(String *msg*){

*//在上下文里面，自行实现对具体策略的选择*

*//优先选用策略：记录到数据库*

LogStrategy strategy = new DbLog();

try{

strategy.log(msg);

}catch(Exception *err*){

*//出错了，那就记录到文件中*

strategy = new FileLog();

strategy.log(msg);

}

}

}

1. 客户端跟以前也一样，示例代码如下：

public class Client {

public static void main(String[] *args*) {

LogContext log = new LogContext();

log.log("记录日志");

log.log("再次记录日志");

}

}

## 总结

1. 优缺点

*优点：*

1. 定义一系列算法

策略模式的功能就是定义一系列算法，实现让这些算法可以相互替换。所以会为这一系列算法定义公共的接口，以约束一系列算法要实现的功能。如果这一系列算法具有公共功能，可以把策略接口实现成为抽象类，把这些公共功能实现到父类里面。

1. 避免多重条件语句

根据前面的示例会发现，策略模式的一系列策略算法是平等的，可以互换的，写在一起就是通过if-else结构来组织，如果此时具体的算法实现里面又有条件语句，就构成了多重条件语句，使用策略模式能避免这样的多重条件语句。

1. 更好的扩展性

在策略模式中扩展新的策略实现非常容易，只要增加新的策略实现类，然后在选择使用策略的地方选择使用这个新的策略实现就好了。

*确定：*

1. 客户必须了解每种策略的不同

策略模式也有缺点，比如让客户端来选择具体使用哪一个策略，这就可能会让客户需要了解所有的策略，还要了解各种策略的功能和不同，这样才能做出正确的选择，而且这样也暴露了策略的具体实现。

1. 增加了对象数目

由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很可观。

1. 只适合扁平的算法结构

策略模式的一系列算法地位是平等的，是可以相互替换的，事实上构成了一个扁平的算法结构，也就是在一个策略接口下，有多个平等的策略算法，就相当于兄弟算法。而且在运行时刻只有一个算法被使用，这就限制了算法使用的层级，使用的时候不能嵌套使用。

对于出现需要嵌套使用多个算法的情况，比如折上折、折后返卷等业务的实现，需要组合或者是嵌套使用多个算法的情况，可以考虑使用装饰模式、或是变形的职责链、或是AOP等方式来实现。

1. 思考
2. 策略模式的本质

**分离算法，选择实现**。

仔细思考策略模式的结构和实现的功能，会发现，如果没有上下文，策略模式就回到了最基本的接口和实现了，只要是面向接口编程的，那么就能够享受到接口的封装隔离带来的好处。也就是通过一个统一的策略接口来封装和隔离具体的策略算法，面向接口编程的话，自然不需要关心具体的策略实现，也可以通过使用不同的实现类来实例化接口，从而实现切换具体的策略。

看起来好像没有上下文什么事情，但是如果没有上下文，那么就需要客户端来直接与具体的策略交互，尤其是当需要提供一些公共功能，或者是相关状态存储的时候，会大大增加客户端使用的难度。因此，引入上下文还是很必要的，有了上下文，这些工作就由上下文来完成了，客户端只需要与上下文交互就可以了，这样会让整个设计模式更独立、更有整体性，也让客户端更简单。

但纵观整个策略模式实现的功能和设计，它的本质还是“分离算法，选择实现”，因为分离并封装了算法，才能够很容易的修改和添加算法；也能很容易的动态切换使用不同的算法，也就是动态选择一个算法来实现需要的功能了。

1. 对设计原则的体现

从设计原则上来看，策略模式很好的体现了开-闭原则。策略模式通过把一系列可变的算法进行封装，并定义出合理的使用结构，使得在系统出现新算法的时候，能很容易的把新的算法加入到已有的系统中，而已有的实现不需要做任何修改。这在前面的示例中已经体现出来了，好好体会一下。

从设计原则上来看，策略模式还很好的体现了里氏替换原则。策略模式是一个扁平结构，一系列的实现算法其实是兄弟关系，都是实现同一个接口或者继承的同一个父类。这样只要使用策略的客户保持面向抽象类型编程，就能够使用不同的策略的具体实现对象来配置它，从而实现一系列算法可以相互替换。

1. 何时选用策略模式
2. 出现有许多相关的类，仅仅是行为有差别的情况，可以使用策略模式来使用多个行为中的一个来配置一个类的方法，实现算法动态切换。
3. 出现同一个算法，有很多不同的实现的情况，可以使用策略模式来把这些“不同的实现”实现成为一个算法的类层次
4. 需要封装算法中，与算法相关的数据的情况，可以使用策略模式来避免暴露这些跟算法相关的数据结构
5. 出现抽象一个定义了很多行为的类，并且是通过多个if-else语句来选择这些行为的情况，可以使用策略模式来代替这些条件语句

## 相关模式

1. 策略模式和状态模式

这两个模式从模式结构上看是一样的，但是实现的功能是不一样的。

状态模式是根据状态的变化来选择相应的行为，不同的状态对应不同的类，每个状态对应的类实现了该状态对应的功能，在实现功能的同时，还会维护状态数据的变化。这些实现状态对应的功能的类之间是不能相互替换的。

策略模式是根据需要或者是客户端的要求来选择相应的实现类，各个实现类是平等的，是可以相互替换的。

另外策略模式可以让客户端来选择需要使用的策略算法，而状态模式一般是由上下文，或者是在状态实现类里面来维护具体的状态数据，通常不由客户端来指定状态。

1. 策略模式和模板方法模式

这两个模式可组合使用，如同前面示例的那样。

模板方法重在封装算法骨架，而策略模式重在分离并封装算法实现。

1. 策略模式和享元模式

这两个模式可组合使用。

策略模式分离并封装出一系列的策略算法对象，这些对象的功能通常都比较单一，很多时候就是为了实现某个算法的功能而存在，因此，针对这一系列的、多个细粒度的对象，可以应用享元模式来节省资源，但前提是这些算法对象要被频繁的使用，如果偶尔用一次，就没有必要做成享元了。