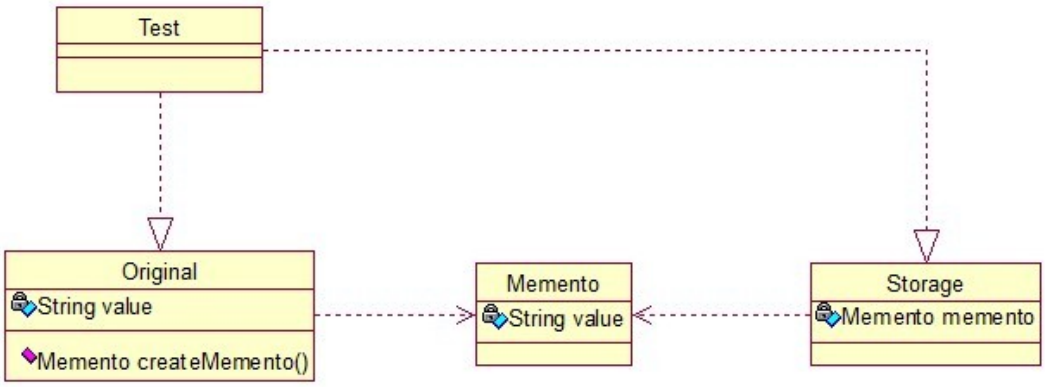
设计模式

一、备忘录模式（Memento）

备忘录模式（Memento）主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象，个人觉得叫备份模式更形象些，通俗的讲下：假设有原始类 A，A 中有各种属性，A 可以决定需要备份的属性，备忘录类 B 是用来存储 A 的一些内部状态，类 C 呢，就是一个用来存储备忘录的，且只能存储，不能修改等操作。做个图来分析一下：



Original 类是原始类，里面有需要保存的属性 value 及创建一个备忘录类，用来保存 value 值。Memento 类是备忘录类，Storage 类是存储备忘录的类，持有 Memento 类的实例，该模式很好理解。直接看源码：

```
public class Original {
    private String value;

    public Original(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public Memento createMemento() {
        return new Memento(value);
```

```
    }

    public void restoreMemento(Memento memento) {
        this.value = memento.getValue();
    }
}
```

```
public class Memento {
    private String value;

    public Memento(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

```
public class Storage {
    private Memento memento;

    public Storage(Memento memento) {
        this.memento = memento;
    }

    public Memento getMemento() {
        return memento;
    }

    public void setMemento(Memento memento) {
        this.memento = memento;
    }
}
```

测试类:

```
public class Test {
    public static void main(String[] args) {
        // 创建原始类
        Original origi = new Original("egg");
```

```
        // 创建备忘录
        Storage storage = new Storage(origi.createMemento());

        // 修改原始类的状态
        System.out.println("初始化状态为：" + origi.getValue());
        origi.setValue("niu");
        System.out.println("修改后的状态为：" + origi.getValue());

        // 回复原始类的状态
        origi.restoreMemento(storage.getMemento());
        System.out.println("恢复后的状态为：" + origi.getValue());
    }
}
```

输出：

初始化状态为：egg

修改后的状态为：niu

恢复后的状态为：egg

简单描述下：新建原始类时，value 被初始化为 egg，后经过修改，将 value 的值置为 niu，最后倒数第二行进行恢复状态，结果成功恢复了。其实我觉得这个模式叫“备份-恢复”模式最形象。

二、策略（Strategy）模式

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。

(一)、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。

(2)、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。

(3)、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**。

(4)、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。

(5)、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。

(6)、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。

(7)、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。

(8)、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：

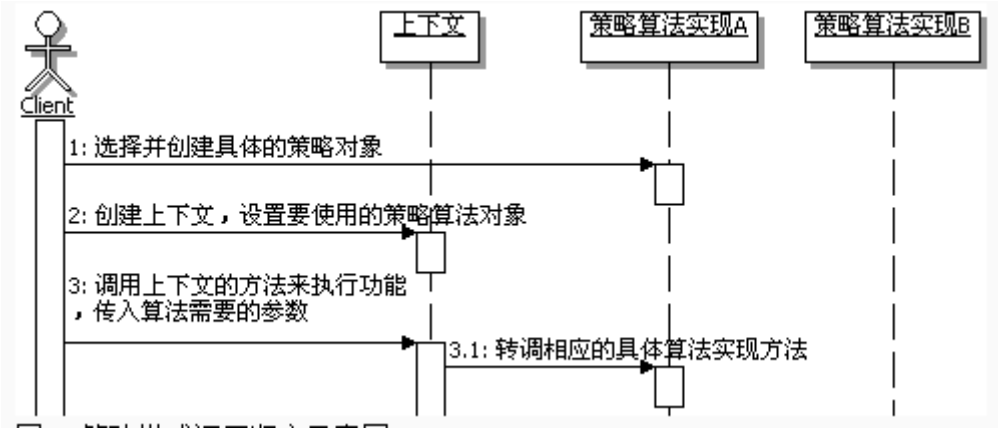


图 1 策略模式调用顺序示意图

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);
}
```

(2)、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");
    }
}
```

(3)、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);
        }
    }
}
```

(4)、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。

(二)、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;
    }
}
```

(2)、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？

(3)、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。

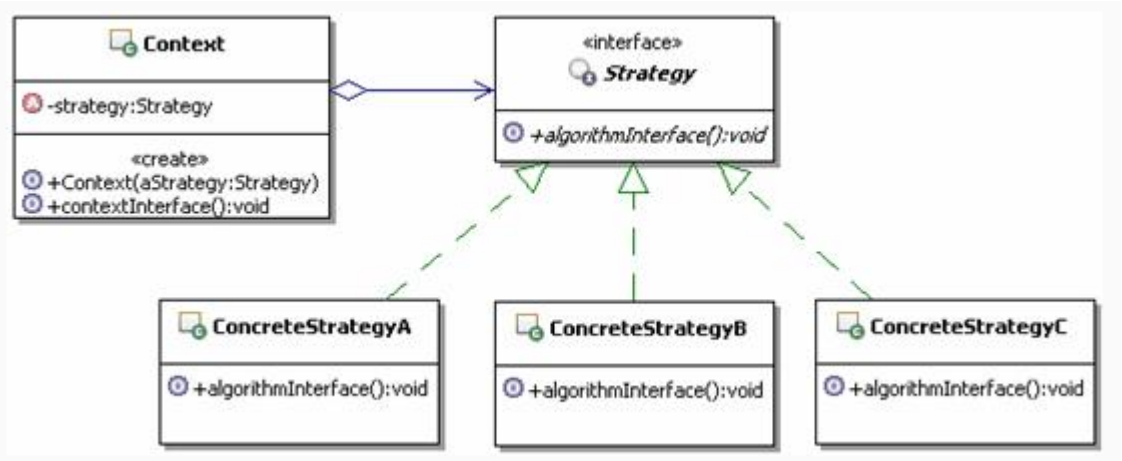


图 2 策略模式结构示意图

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。

(4)、策略模式示例代码

a)、 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();
}
```

b)、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
    }
}
```

c)、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。

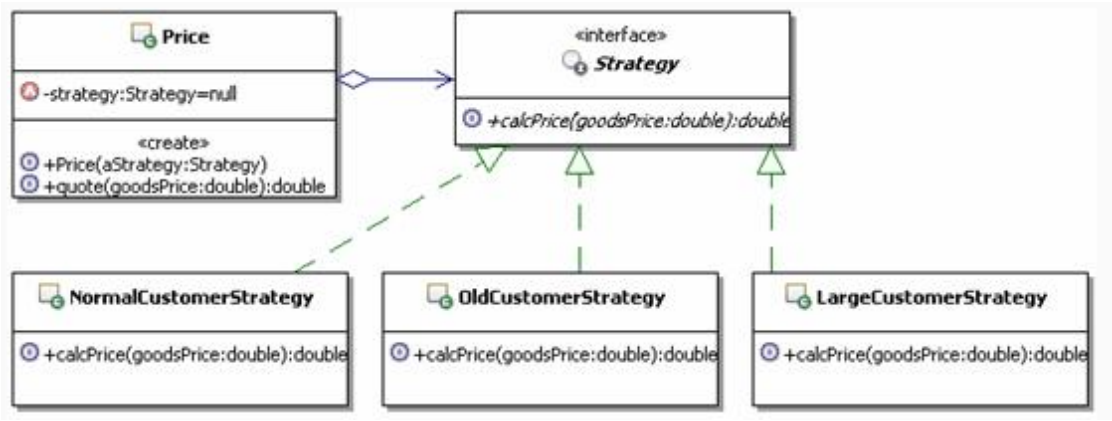


图 3 使用策略模式实现示例的结构示意图

(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);
}
```

(2)、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);
    }
}
```

(3)、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);
    }
}
```

(4)、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。

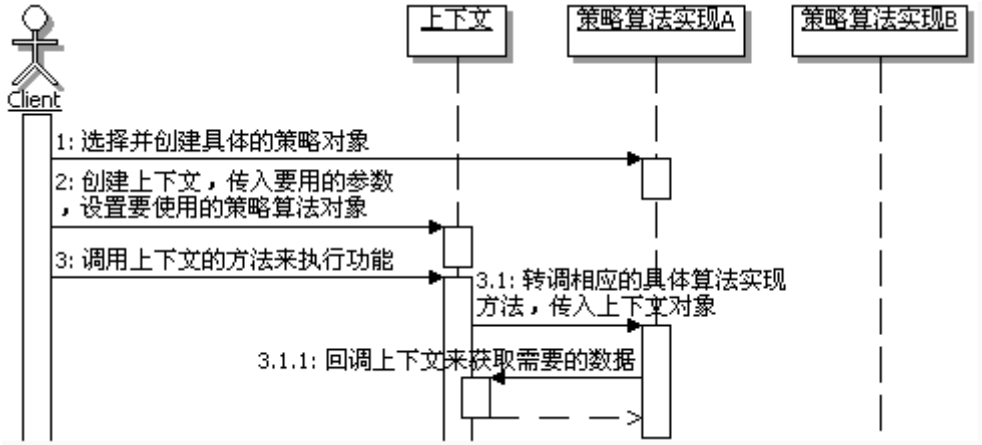


图 4 策略模式调用顺序示意图

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:

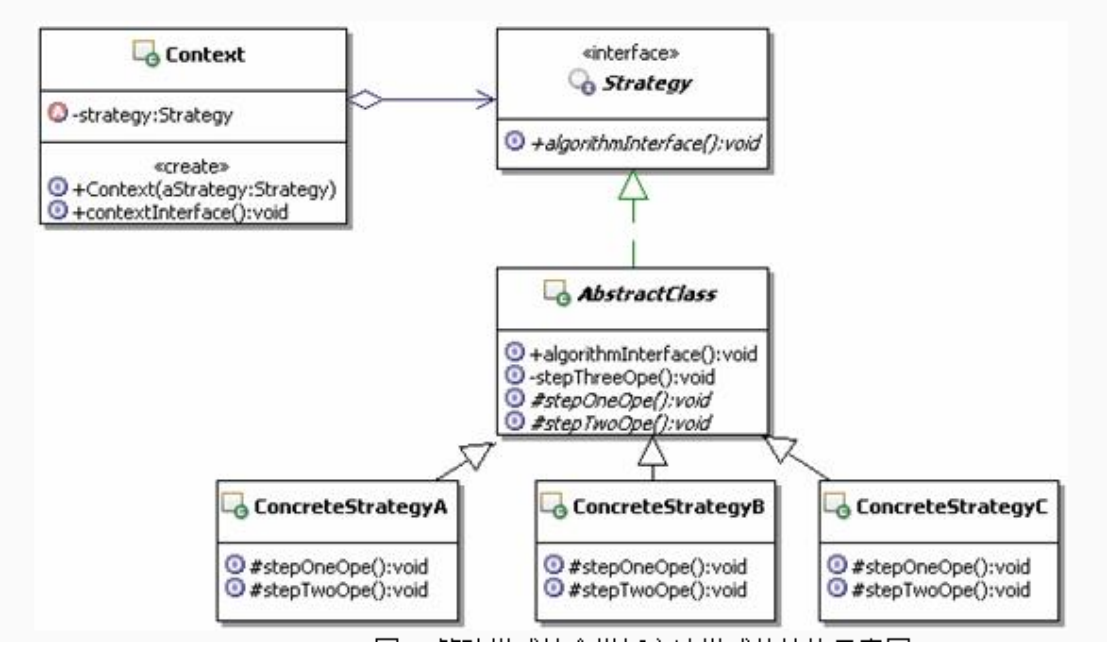


图 5 策略模式结合模板方法模式的结构示意图

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);
}
```

(2)、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);
}
```

(3)、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);
        }
    }
}
```

(4)、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");
    }
}
```

(三)、总结

(1)、优缺点

优点：

i. 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。

ii. 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。

iii. 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：

i. 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。

ii. 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。

iii. 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。

(2)、思考

i. 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。

ii. 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。

iii. 何时选用策略模式

a)、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。

b)、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

c)、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

d)、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

(四)、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。

(2)、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。

(3)、策略模式和享元模式

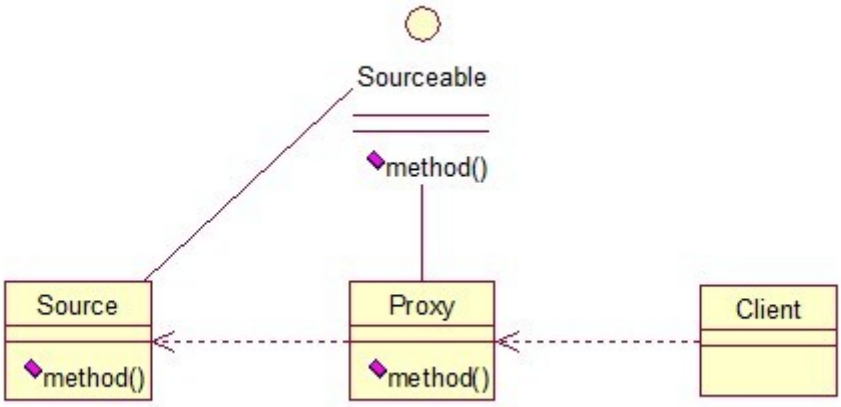
这两个模式可组合使用。

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。

三、代理模式（Proxy）

(一)、综述

代理模式就是多一个代理类出来，替原对象进行一些操作，比如我们在租房子的时候回去找中介，为什么呢？因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。先来看看关系图：
工厂模式可以分为三类：



代码:

```
public interface Sourceable {
    public void method();
}
```

```
public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}
```

```
public class Proxy implements Sourceable {
    private Source source;

    public Proxy() {
        super();
        this.source = new Source();
    }

    @Override
    public void method() {
        before();
        source.method();
        atfer();
    }

    private void atfer() {
        System.out.println("after proxy!");
    }

    private void before() {
```

```
        System.out.println("before proxy!");
    }
}
```

测试类：

```
public class ProxyTest {
    public static void main(String[] args) {
        Sourceable source = new Proxy();
        source.method();
    }
}
```

输出：

before proxy!

the original method!

after proxy!

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。

2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

四、单例设计模式

(一)、懒汉（线程不安全）

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

这种写法 lazy loading 很明显，但是致命的是在多线程不能正常工作。

(二)、懒汉（线程安全）

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
    }

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

这种写法能够在多线程中很好的工作，而且看起来它也具备很好的 lazy loading，但是，遗憾的是，效率很低，99%情况下不需要同步。

(三)、饿汉

```
public class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton() {
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

这种方式基于 classloder 机制避免了多线程的同步问题，不过，instance 在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用 getInstance 方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化 instance 显然没有达到 lazy loading 的效果。而且不能延迟加载，某种意义上会增加系统的负载。

(四)、饿汉（变种）

```
public class Singleton {

    private static final Singleton instance;

    static {
        instance = new Singleton();
```

```
    }

    private Singleton() {
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```

表面上看起来差别挺大，其实更第三种方式差不多，都是在类初始化即实例化 instance。

(五)、静态内部类

```
public class Singleton {
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    private Singleton() {
    }

    public static final Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}
```

这种方式同样利用了 classloder 的机制来保证初始化 instance 时只有一个线程，它跟第三种和第四种方式不同的是（很细微的差别）：第三种和第四种方式是只要 Singleton 类被装载了，那么 instance 就会被实例化（没有达到 lazy loading 效果），而这种方式是 Singleton 类被装载了，instance 不一定被初始化。因为 SingletonHolder 类没有被主动使用，只有显示通过调用 getInstance 方法时，才会显示装载 SingletonHolder 类，从而实例化 instance。想象一下，如果实例化 instance 很消耗资源，我想让他延迟加载，另外一方面，我不希望在 Singleton 类加载时就实例化，因为我不能确保 Singleton 类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化 instance 显然是不合适的。这个时候，这种方式相比第三和第四种方式就显得很合理。

(六)、枚举

```
public enum Singleton {
    INSTANCE;

    public void whateverMethod() {
    }
}
```

这种方式是 Effective Java 作者 Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，可谓是很坚强的壁垒啊，不过，个人认为由于 1.5 中才加入 enum 特性，用这种方式写不免让人感觉生疏，在实际工作中，我也很少看见有人这么写过。

(七)、双重校验锁

```
public class Singleton {
    private volatile static Singleton singleton;

    private Singleton() {
    }

    public static Singleton getSingleton() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

这个是第二种方式的升级版，俗称双重检查锁定，在 JDK1.5 之后，双重检查锁定才能够正常达到单例效果。

(八)、总结

有两个问题需要注意:

1、如果单例由不同的类装载器装入，那便有可能存在多个单例类的实例。假定不是远端存取，例如一些 servlet 容器对每个 servlet 使用完全不同的类 装载器，这样的话如果有两个 servlet 访问一个单例类，它们就都会有各自的实例。

2、如果 Singleton 实现了 java.io.Serializable 接口，那么这个类的实例就可能被序列化和复原。不管怎样，如果你序列化一个单例类的对象，接下来复原多个那个对象，那你就会有多个单例类的实例。

对第一个问题修复的办法是:

```
private static Class getClass(String classname)
            throws ClassNotFoundException {
        ClassLoader classLoader =
Thread.currentThread().getContextClassLoader();

        if (classLoader == null)
```

```
            classLoader = Singleton.class.getClassLoader();

        return (classLoader.loadClass(classname));
    }
```

对第二个问题修复的办法是:

```
public class Singleton implements java.io.Serializable {
    public static Singleton INSTANCE = new Singleton();

    protected Singleton() {

    }

    private Object readResolve() {
        return INSTANCE;
    }
}
```

第三种和第五种方式，简单易懂，而且在 JVM 层实现了线程安全（如果不是多个类加载器环境），一般的情况下，我会使用第三种方式，只有在要明确实现 lazy loading 效果时才会使用第五种方式，另外，如果涉及到反序列化创建对象时我会试着使用枚举的方式来实现单例，不过，我一直会保证我的程序是线程安全的，而且我永远不会使用第一种和第二种方式，如果有其他特殊的需求，我可能会使用第七种方式，毕竟，JDK1.5 已经没有双重检查锁定的问题了。

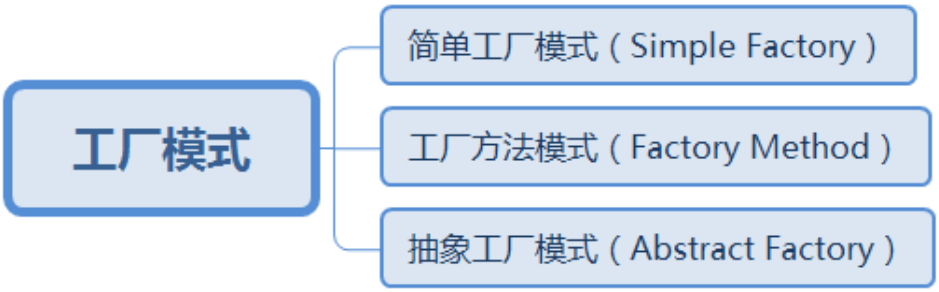
不过一般来说，第一种不算单例，第四种和第三种就是一种，如果算的话，第五种也可以分开写了。所以说，一般单例都是五种写法。懒汉，恶汉，双重校验锁，枚举和静态内部类。

五、工厂模式

(一)、综述

工厂模式主要是为创建对象提供过渡接口，以便将创建对象的具体过程屏蔽隔离起来，达到提高灵活性的目的。

工厂模式可以分为三类:



这三种模式从上到下逐步抽象，并且更具一般性。

GOF在《设计模式》一书中将工厂模式分为两类：工厂方法模式（Factory Method）与抽象工厂模式（Abstract Factory）。将简单工厂模式（Simple Factory）看为工厂方法模式的一种特例，两者归为一类。

区别

工厂方法模式：	抽象工厂模式：
一个抽象产品类，可以派生出多个具体产品类。	多个抽象产品类，每个抽象产品类可以派生出多个具体产品类。
一个抽象工厂类，可以派生出多个具体工厂类。	一个抽象工厂类，可以派生出多个具体工厂类。
每个具体工厂类只能创建一个具体产品类的实例。	每个具体工厂类可以创建多个具体产品类的实例。

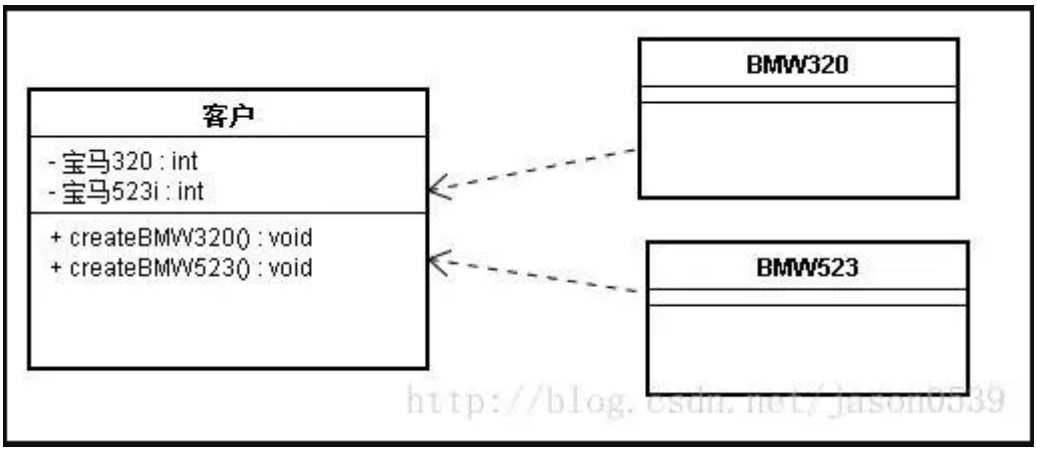
区别总结：

工厂方法模式只有一个抽象产品类，而抽象工厂模式有多个。

工厂方法模式的具体工厂类只能创建一个具体产品类的实例，而抽象工厂模式可以创建多个。

(二)、简单工厂模式（Simple Factory）

建立一个工厂（一个函数或一个类方法）来制造新的对象。

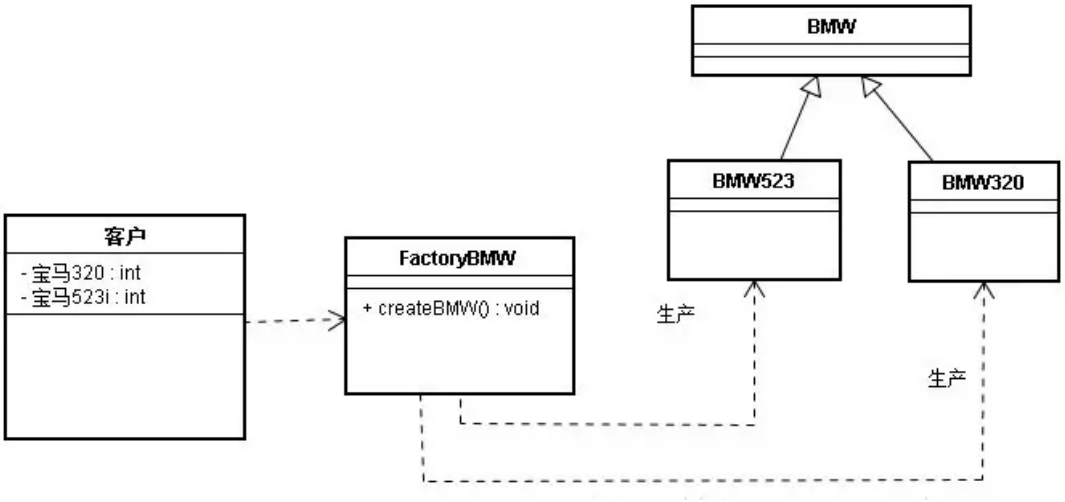



```
public class BMW320 {
    public BMW320(){
        System.out.println("制造-->BMW320");
    }
}
```

```
public class BMW523 {
    public BMW523(){
        System.out.println("制造-->BMW523");
    }
}
```

```
public class Customer {
    public static void main(String[] args) {
        BMW320 bmw320 = new BMW320();
        BMW523 bmw523 = new BMW523();
    }
}
```

客户需要知道怎么去创建一款车,客户和车就紧密耦合在一起了.为了降低耦合,就出现了工厂类,把创建宝马的操作细节都放到了工厂里面去,客户直接使用工厂的创建工厂方法,传入想要的宝马车型号就行了,而不必去知道创建的细节.这就是工业革命了: **简单工厂模式** 即我们建立一个工厂类方法来制造新的对象。如图:



产品类：

```
//抽象产品
abstract class BMW {
    public BMW(){

    }
}
```

```
//具体产品
public class BMW320 extends BMW {
    public BMW320() {
        System.out.println("制造-->BMW320");
    }
}
```

```
//具体产品
public class BMW523 extends BMW{
    public BMW523(){
        System.out.println("制造-->BMW523");
    }
}
```

工厂类：

```
public class Factory {
    public BMW createBMW(int type) {
        switch (type) {
```

```
        case 320:
            return new BMW320();

        case 523:
            return new BMW523();

        default:
            break;
        }
        return null;
    }
}
```

客户类：

```
public class Customer {
    public static void main(String[] args) {
        Factory factory = new Factory();
        BMW bmw320 = factory.createBMW(320);
        BMW bmw523 = factory.createBMW(523);
    }
}
```

简单工厂模式又称静态工厂方法模式。从命名上就可以看出这个模式一定很简单。它存在的目的很简单：定义一个用于创建对象的接口。

先来看看它的组成：

1) 工厂类角色：这是本模式的核心，含有一定的商业逻辑和判断逻辑，用来创建产品

2) 抽象产品角色：它一般是具体产品继承的父类或者实现的接口。

3) 具体产品角色：工厂类所创建的对象就是此角色的实例。在 java 中由一个具体类实现。

下面我们从开闭原则（对扩展开放；对修改封闭）上来分析下简单工厂模式。当客户不再满足现有的车型号的时候，想要一种速度快的新型车，只要这种车符合抽象产品制定的合同，那么只要通知工厂类知道就可以被客户使用了。所以对产品部分来说，它是符合开闭原则的；但是工厂部分好像不太理想，**因为每增加一种新型车，都要在工厂类中增加相应的创建业务逻辑（createBMW(int type)方法需要新增 case），这显然是违背开闭原则的**。可想而知对于新产品的加入，工厂类是很被动的。对于这样的工厂类，我们称它为全能类或者上帝类。

我们举的例子是最简单的情况，而在实际应用中，很可能产品是一个多层次的树状结构。由于简单工厂模式中只有一个工厂类来对应这些产品，所以这可能会把我们的上帝累坏了，也累坏了我们这些程序员。

于是工厂方法模式作为救世主出现了。 工厂类定义成了接口,而每新增的车种类型,就增加该车种类型对应工厂类的实现,这样工厂的设计就可以扩展了,而不必去修改原来的代码。

(三)、工厂方法模式（Factory Method）

工厂方法模式去掉了简单工厂模式中工厂方法的静态属性，使得它可以被子类继承。这样在简单工厂模式里集中在工厂方法上的压力可以由工厂方法模式里不同的工厂子类来分担。

工厂方法模式组成：

1)抽象工厂角色： 这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在 java 中它由抽象类或者接口来实现。

2)具体工厂角色：它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。

3)抽象产品角色：它是具体产品继承的父类或者是实现的接口。在 java 中一般有抽象类或者接口来实现。

4)具体产品角色：具体工厂角色所创建的对象就是此角色的实例。在 java 中由具体的类来实现。

工厂方法模式使用继承自抽象工厂角色的多个子类来代替简单工厂模式中的“上帝类”。正如上面所说，这样便分担了对象承受的压力；而且这样使得结构变得灵活 起来——当有新的产品产生时，只要按照抽象产品角色、抽象工厂角色提供的合同来生成，那么就可以被客户使用，而不必去修改任何已有 的代码。可以看出工厂角色的结构也是符合开闭原则的！ 代码如下：

产品类：

```
abstract class BMW {
    public BMW(){

    }
}
```

```
public class BMW320 extends BMW {
    public BMW320() {
        System.out.println("制造-->BMW320");
    }
}
```

```
public class BMW523 extends BMW{
    public BMW523(){
        System.out.println("制造-->BMW523");
    }
}
```

创建工厂类：

```
interface FactoryBMW {
    BMW createBMW();
```

```
}
```

```
public class FactoryBMW320 implements FactoryBMW{
     @Override
    public BMW320 createBMW() {
         return new BMW320();
    }
 }
```

```
public class FactoryBMW523 implements FactoryBMW {
    @Override
    public BMW523 createBMW() {
        return new BMW523();
    }
}
```

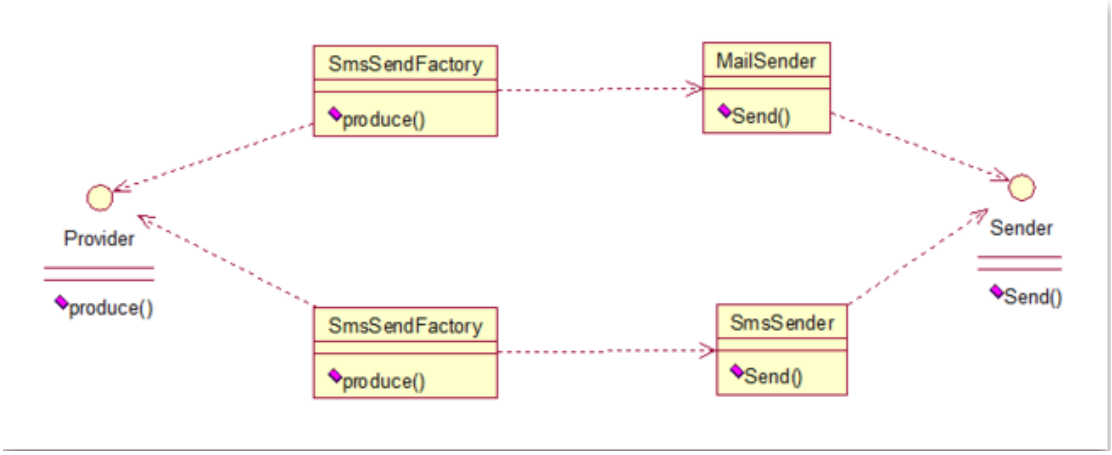
客户类:

```
public class Customer {
    public static void main(String[] args) {
        FactoryBMW320 factoryBMW320 = new FactoryBMW320();
        BMW320 bmw320 = factoryBMW320.createBMW();

        FactoryBMW523 factoryBMW523 = new FactoryBMW523();
        BMW523 bmw523 = factoryBMW523.createBMW();
    }
}
```

工厂方法模式仿佛已经很完美的对对象的创建进行了包装,使得客户程序中仅仅处理抽象产品角色提供的接口，但使得对象的数量成倍增长。当产品种类非常多时，会出现大量的与之对应的工厂对象，这不是我们所希望的。

(四)、抽象工厂模式（Abstract Factory）

工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。因为抽象工厂不太好理解，我们先看看图，然后就和代码，就比较容易理解。



请看例子:

```
public interface Sender {
    public void Send();
}
```

两个实现类:

```
public class MailSender implements Sender {
    @Override
    public void Send() {
        System.out.println("this is mailsender!");
    }
}
```

```
public class SmsSender implements Sender {
    @Override
    public void Send() {
        System.out.println("this is sms sender!");
    }
}
```

两个工厂类:

```
public class SendMailFactory implements Provider {
    @Override
    public Sender produce(){
        return new MailSender();
    }
}
public class SendSmsFactory implements Provider{
    @Override
```

```
    public Sender produce() {
        return new SmsSender();
    }
}
```

再提供一个接口：

```
public interface Provider {
    public Sender produce();
}
```

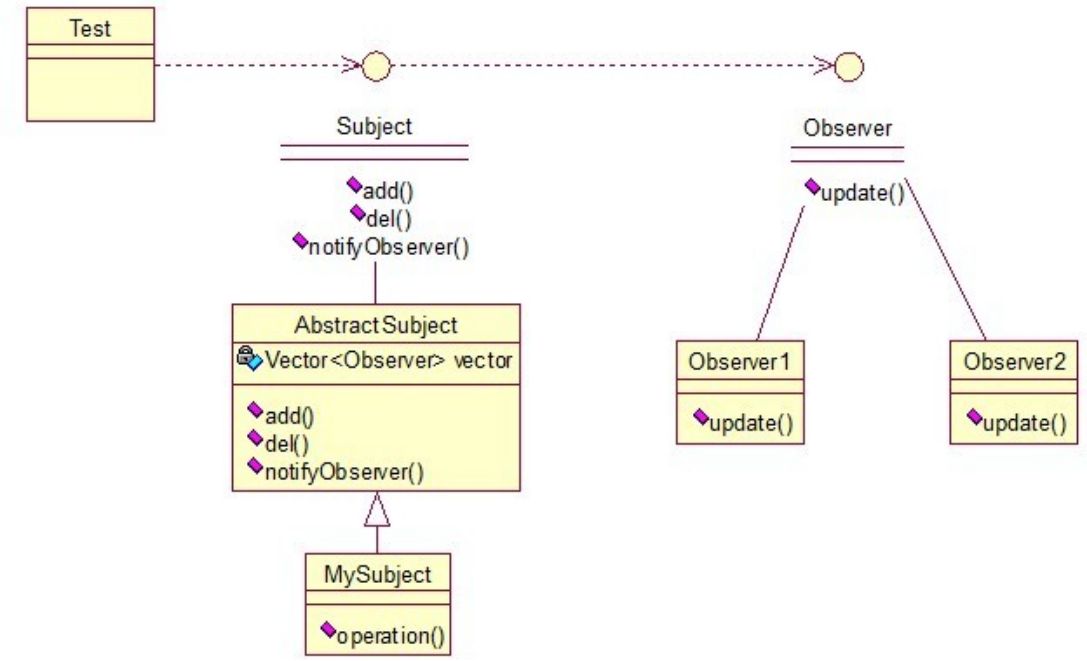
测试类：

```
public class Test {
    public static void main(String[] args) {
        Provider provider = new SendMailFactory();
        Sender sender = provider.produce();
        sender.Send();
    }
}
```

其实这个模式的好处就是，如果你现在想增加一个功能：发及时信息，则只需做一个实现类，实现 Sender 接口，同时做一个工厂类，实现 Provider 接口，就 OK 了，无需去改动现成的代码。这样做，拓展性较好！

六、观察者（Observer）模式

观察者（Observer）模式是类和类之间的关系，不涉及到继承。观察者模式很好理解，类似于邮件订阅和 RSS 订阅，当我们浏览一些博客或 wiki 时，经常会看到 RSS 图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。先来看看关系图：



其中，MySubject 类就是我们的主对象，Observer1 和 Observer2 是依赖于 MySubject 的对象，当 MySubject 变化时，Observer1 和 Observer2 必然变化。AbstractSubject 类中定义着需要监控的对象列表，可以对其进行修改：增加或删除被监控对象，且当 MySubject 变化时，负责通知在列表内存在的对象。我们看实现代码：

一个 Observer 接口：

```
public interface Observer {
    public void update();
}
```

两个实现类：

```
public class Observer1 implements Observer {
    @Override
    public void update() {
        System.out.println("observer1 has received!");
    }
}
```

```
public class Observer2 implements Observer {
    @Override
    public void update() {
        System.out.println("observer2 has received!");
    }
}
```

Subject 接口及实现类：

```java
public interface Subject {
    /*增加观察者*/
    public void add(Observer observer);

    /*删除观察者*/
    public void del(Observer observer);

    /*通知所有的观察者*/
    public void notifyObservers();

    /*自身的操作*/
    public void operation();
}
```

```java
public abstract class AbstractSubject implements Subject {
    private Vector<Observer> vector = new Vector<Observer>();

    @Override
    public void add(Observer observer) {
        vector.add(observer);
    }

    @Override
    public void del(Observer observer) {
        vector.remove(observer);
    }

    @Override
    public void notifyObservers() {
        Enumeration<Observer> enumo = vector.elements();
        while (enumo.hasMoreElements()) {
            enumo.nextElement().update();
        }
    }
}
```

```java
public class MySubject extends AbstractSubject {
    @Override
    public void operation() {
        System.out.println("update self!");
        notifyObservers();
    }
}
```

测试类：

```
public class ObserverTest {
    public static void main(String[] args) {
        Subject sub = new MySubject();
        sub.add(new Observer1());
        sub.add(new Observer2());

        sub.operation();
    }
}
```

输出：

update self!

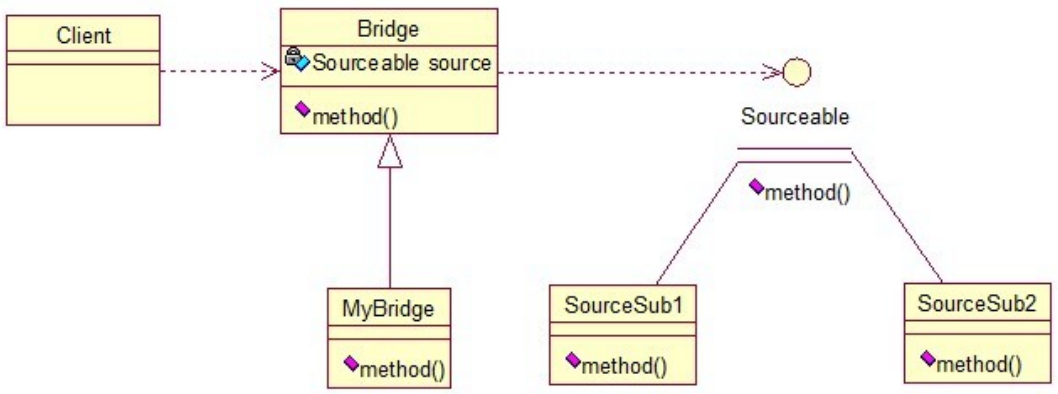
observer1 has received!

observer2 has received!

这些东西，其实不难，只是有些抽象，不太容易整体理解，建议读者：根据关系图，新建项目，自己写代码（或者参考我的代码）,按照总体思路走一遍，这样才能体会它的思想，理解起来容易！

七、桥接（Bridge）模式

桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：将抽象化与实现化解耦，使得二者可以独立变化，像我们常用的 JDBC 桥 DriverManager 一样，JDBC 进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，原因就是 JDBC 提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了。我们来看看关系图：



实现代码：

先定义接口：

```
public interface Sourceable {
```

```
    public void method();
}
```

分别定义两个实现类:

```
public class SourceSub1 implements Sourceable {
    @Override
    public void method() {
        System.out.println("this is the first sub!");
    }
}
```

```
public class SourceSub2 implements Sourceable {
    @Override
    public void method() {
        System.out.println("this is the second sub!");
    }
}
```

定义一个桥，持有 Sourceable 的一个实例:

```
public abstract class Bridge {
    private Sourceable source;

    public void method() {
        source.method();
    }

    public Sourceable getSource() {
        return source;
    }

    public void setSource(Sourceable source) {
        this.source = source;
    }
}
```

```
public class MyBridge extends Bridge {
    @Override
    public void method() {
        getSource().method();
    }
}
```

测试类:

```
public class BridgeTest {
```

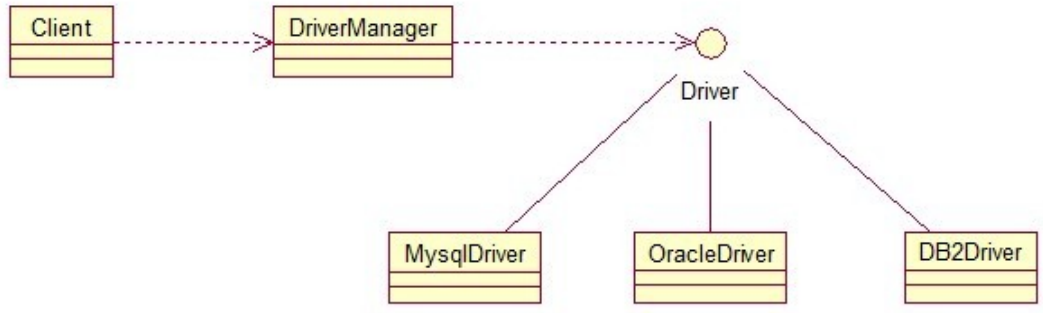
```
    public static void main(String[] args) {
        Bridge bridge = new MyBridge();

        /*调用第一个对象*/
        Sourceable source1 = new SourceSub1();
        bridge.setSource(source1);
        bridge.method();

        /*调用第二个对象*/
        Sourceable source2 = new SourceSub2();
        bridge.setSource(source2);
        bridge.method();
    }
}
```

输出：

this is the first sub!

this is the second sub!

这样，就通过对 Bridge 类的调用，实现了对接口 Sourceable 的实现类 SourceSub1 和 SourceSub2 的调用。接下来我再画个图，大家就应该明白了，因为这个图是我们 JDBC 连接的原理，有数据库学习基础的，一结合就都懂了。

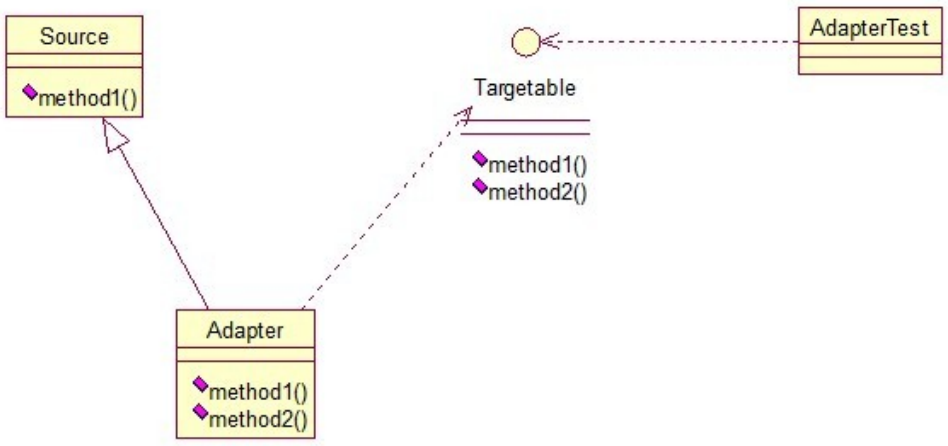


八、适配器（Adapter）设计模式

(一)、综述

适配器模式（Adapter）将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

(二)、类的适配器模式



核心思想就是：有一个 Source 类，拥有一个方法，待适配，目标接口时 Targetable，通过 Adapter 类，将 Source 的功能扩展到 Targetable 里，看代码：

```
public class Source {
    public void method1() {
        System.out.println("this is original method!");
    }
}
```

```
public interface Targetable {
    /* 与原类中的方法相同 */
    public void method1();

    /* 新类的方法 */
    public void method2();
}
```

```
public class Adapter extends Source implements Targetable {

        @Override
        public void method2() {
            System.out.println("this is the targetable method!");
        }
    }
```

Adapter 类继承 Source 类，实现 Targetable 接口，下面是测试类：

```
public class AdapterTest {
    public static void main(String[] args) {
        Targetable target = new Adapter();
        target.method1();
        target.method2();
```

```
    }
}
```

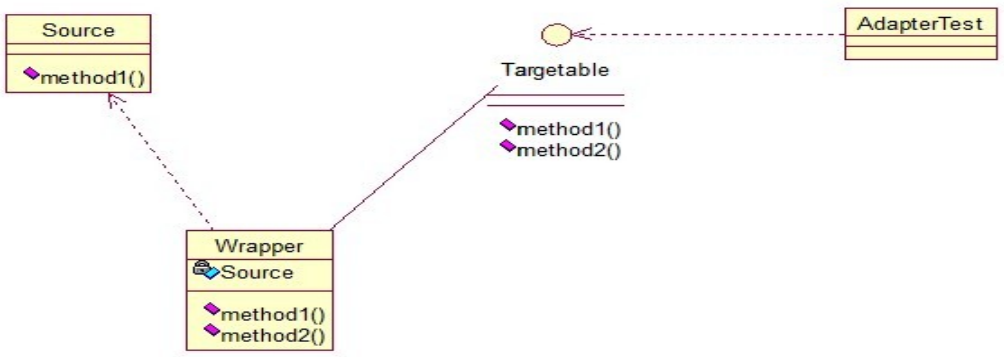
输出：

this is original method!
this is the targetable method!

这样 Targetable 接口的实现类就具有了 Source 类的功能。

(三)、对象的适配器模式

基本思路和类的适配器模式相同，只是将 Adapter 类作修改，这次不继承 Source 类，而是持有 Source 类的实例，以达到解决兼容性的问题。看图：




```
public class Wrapper implements Targetable {
    private Source source;

    public Wrapper(Source source) {
        super();
        this.source = source;
    }

    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }

    @Override
    public void method1() {
        source.method1();
    }
}
```

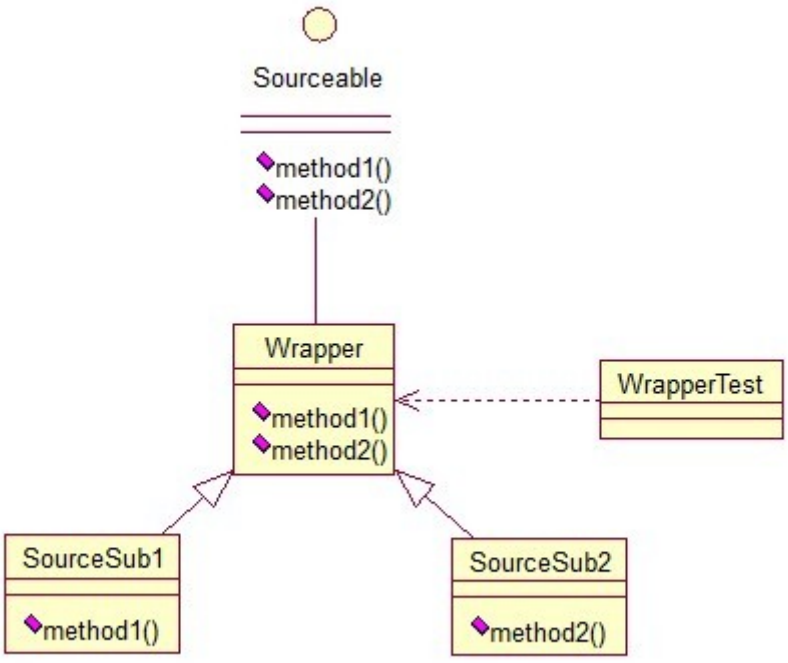
测试类：

```
public class AdapterTest {
    public static void main(String[] args) {
        Source source = new Source();
        Targetable target = new Wrapper(source);
        target.method1();
        target.method2();
    }
}
```

输出与第一种一样，只是适配的方法不同而已。

(四)、接口的适配器模式

接口的适配器是这样的，有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。看一下类图:



这个很好理解，在实际开发中，我们也常会遇到这种接口中定义了太多的方法，以致于有时我们在一些实现类中并不是都需要。看代码:

```
public interface Sourceable {
    public void method1();

    public void method2();
}
```

抽象类 Wrapper2:

```
public abstract class Wrapper2 implements Sourceable {

    public void method1() {
    }

    public void method2() {
    }
}
```

```
public class SourceSub1 extends Wrapper2 {
    public void method1() {
        System.out.println("the sourceable interface's first Sub1!");
    }
}
```

```
public class SourceSub2 extends Wrapper2 {
    public void method2() {
        System.out.println("the sourceable interface's second Sub2!");
    }
}
```

```
public class WrapperTest {
    public static void main(String[] args) {
        Sourceable source1 = new SourceSub1();
        Sourceable source2 = new SourceSub2();

        source1.method1();
        source1.method2();
        source2.method1();
        source2.method2();
    }
}
```

测试输出:

the sourceable interface's first Sub1!
the sourceable interface's second Sub2!

达到了我们的效果!

(五)、总结

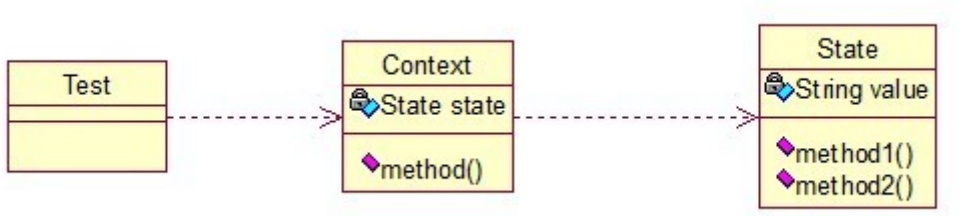
类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个 Wrapper 类，持有原类的一个实例，在 Wrapper 类的方法中，调用实例的方法就行。

接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类 Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

九、状态（State）设计模式

状态（State）模式核心思想就是：当对象的状态改变时，同时改变其行为，很好理解！就拿 QQ 来说，有几种状态，在线、隐身、忙碌等，每个状态对应不同的操作，而且你的好友也能看到你的状态，所以，状态模式就两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。看图：



其中，State 类是个状态类，Context 类可以实现切换，我们来看看代码：

```
/**
 * @Title: 状态类的核心类
 * @author: 夜阑珊
 */
public class State {
    private String value;

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public void method1() {
        System.out.println("execute the first opt!");
    }
```

```
    public void method2() {
        System.out.println("execute the second opt!");
    }
}
```

```
/**
 * @Title: 状态模式的切换类
 * @author: 夜阑珊
 */
public class Context {
    private State state;

    public Context(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }

    public void setState(State state) {
        this.state = state;
    }

    public void method() {
        if (state.getValue().equals("state1")) {
            state.method1();
        } else if (state.getValue().equals("state2")) {
            state.method2();
        }
    }
}
```

测试类:

```
public class Test {
    public static void main(String[] args) {
        State state = new State();
        Context context = new Context(state);

        //设置第一种状态
        state.setValue("state1");
        context.method();

        //设置第二种状态
        state.setValue("state2");
```

```
        context.method();
    }
}
```

输出：

execute the first opt!

execute the second opt!

根据这个特性，状态模式在日常开发中用的挺多的，尤其是做网站的时候，我们有时希望根据对象的某一属性，区别开他们的一些功能，比如说简单的权限控制等。