第 35 章　工厂方法模式+策略模式

本章内容

- 迷你版的交易系统
- 混编小结

35.1 迷你版的交易系统

大家可能对银行的交易系统充满敬畏之情，一听说是银行的 IT 人员，立马想当然地认为这是个很厉害的人物，那我们今天就来对银行的交易系统做一个初步探讨。国内一家大型集团（全球 500 强之一）计划建立全国“一卡通”计划，每个员工配备一张 IC 卡，该卡基本上就是万能的，门禁系统用它，办公系统使用它作为认证，你想打开自己的邮箱，没有它就甭想了,它还可以用来进行消费，比如到食堂吃饭，到园区内的商店消费等等，甚至洗澡、理发、借书、买书等等都可以用它，只要这张卡内有余额，在集团内部就是一张借记卡（当然还有一些内部的补助通过该卡发放，合理避税嘛）。我们要讲解的就是“一卡通”项目联机交易子系统，类似于银行的交易系统，可以说它是交易系统的 mini 版吧。

该项目还是具有一定的挑战性，集团公司的架构分为三层：总部、省级分部、市级机构，业务要求是“一卡通”推广到全国，一名员工从北京出差到了上海，凭一卡通他能在北京做的事情在上海同样能完成。对于联机交易子项目，异地分支机构与总部之间的通信采用了 MQ(Message Queue，消息队列)传递消息，也就是我们观察者模式的 BOSS 版，与目前的通过 POS 机刷信用卡基本上是一个道理。

联机交易子系统有一个非常重要的子模块（Module）——扣款子模块。这个模块太重要了！从业务上来说，扣款失败就代表着所有的商业交易关闭，这是不允许发生的；从技术上来说，扣款的异常处理、事务处理、鲁棒性都是不容忽视的，特别是饭点时间，并发量是很恐怖的，这对我们架构师提出了很高的要求。

我们详细分析一下扣款子模块，每个员工都有一张 IC 卡，他的 IC 卡上有以下两种金额。

- 固定金额

固定金额是指员工不能提现的金额，即使你的固定金额有 10 万元，你也只能干瞪眼看着，不能提取现金，那这部分金额有来做什么呢？只能用来特定消费，什么特定消费？员工日常必需的消费，例如食堂内吃饭、理发、健身等活动。

- 自由金额，

自由金额是可以提现的，当然也可以用于消费。每个月初，总部都会为每个员工的 IC 卡中打入固定数量的金额，然后提倡大家在集团内的商店消费。

在实际的系统开发中，架构设计的是一张 IC 卡绑定两个账户：固定账户和自由账号，本书为

了简化描述还是使用固定金额和自由金额。既然有消费，系统肯定要扣款处理，系统内有两套扣款规则。

- 扣款策略一

该类型的扣款会对 IC 卡上的两个金额都会产生影响，计算公式如下：

IC 卡固定余额 = IC 卡现有固定余额 - 交易金额 / 2

IC 卡自由余额 = IC 卡现有自由金额 - 交易金额 / 2

也就是说，该类型的的消费分别在固定金额和自由金额上各扣除一半。它适用于什么消费场景呢？固定消费场景，例如吃饭、理发等情况下的扣款，为什么要这么做呢？为了防止乱请客的情况，你请别人吃饭，好，你自己也要出一半，你乐意呀。

- 扣款策略二

全部从自由金额的上扣除，由于集团内的各种消费、服务非常齐全，而且比市面价格稍低，员工还是很乐意到这里消费的，而且很多员工本身就住在集团附近，基本上就是“公司即家，家即公司”。

今天要讲的重点就是这两种消费的扣款策略，该怎么设计？先想清楚了再回答，你要知道这种联机交易，日后允许你大规模变更的可能性基本上是零，所以系统设计的时候要做到可拆卸(Pluggable)的架构，避免日后维护的大量开支。

很明显，这是一个策略模式的实际应用，但是你还记得策略模式是有缺陷的吗？它的具体策略必须暴露出去，而且还要由上层模块初始化，这不合适，与迪米特原则有冲突，高层次模块对低层次的模块应该仅仅处在“接触”的层次上，而不应该是“耦合”的关系，否则，维护的工作量就会非常大。问题提出了，那我们就应该想办法来修改这个缺陷，正好工厂方法模式可以帮我们产生指定的对象，但是问题又来了，工厂方法模式要指定一个类，它才能产生对象，怎么办？引入一个配置文件进行映射，避免系统僵化情况的发生，我们以枚举类完成该任务。

还有一个问题，一个交易的扣款模式是固定的，根据其交易编号而定，那我们怎么把交易编号与扣款策略对应起来呢？采用状态模式或责任链模式都可以，如果采用状态则认为交易编号就是一个交易对象的状态，对于一笔确定的交易（一个已经生成了的对象），它的状态不会发生从一个状态过渡到另一个状态的情况，也就是说它的状态只有一个，执行完毕后即结束，不存在多状态的问题；如果采用责任链模式，则可以以交易编码作为链中的判断依据，由每个执行节点进行判断，返回相应的扣款模式。但是在实际中，采用了关系型数据库存储扣款规则与交易编码的对应关系，为了简化该部分的讲义，我们在下面的设计中使用了条件判断语句来代替。

还有，我们分析了这么多，这么复杂的扣款模块总要进行一个封装吧，你不能让上层的业务模

块直接深入到我们模块的内部吧，于是门面模式又摆在了眼前。

分析完毕，我们要先画出类图，挑柿子就选软的捏，那我们做设计也遵循一下这个原则，先选最简单的业务，然后画出类图，那我们先定义交易中使用到的两个类:IC 卡类和交易类，如图 35-1 所示。

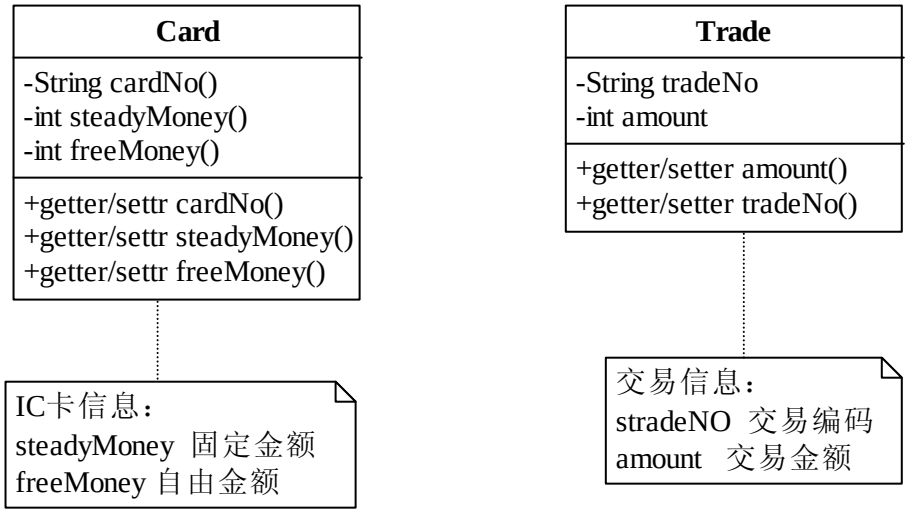

图35-1 IC卡类和交易类

每个 IC 卡有三个属性，分别是 IC 卡号码、固定金额、自由金额，然后通过 getter/setter 方法来访问，如代码清单 35-1 所示。

代码清单35-1 IC卡类

```
public class Card {
    //IC 卡号码
    private String cardNo="";
    //卡内的固定交易金额
    private int steadyMoney =0;
    //卡内自由交易金额
    private int freeMoney =0;
    //getter/setter 方法
    public String getCardNo() {
        return cardNo;
    }

    public void setCardNo(String cardNo) {
        this.cardNo = cardNo;
    }

    public int getSteadyMoney() {
```

```
                return steadyMoney;
        }

        public void setSteadyMoney(int steadyMoney) {
                this.steadyMoney = steadyMoney;
        }

        public int getFreeMoney() {
                return freeMoney;
        }

        public void setFreeMoney(int freeMoney) {
                this.freeMoney = freeMoney;
        }
    }
```

细心的读者可能注意到，你的金额怎么都是整数类型呀，这不对呀，应该是 double 类型或者 BigDecimal 类型呀。是，一般非银行的交易系统，比如超市的收银系统，系统内都是存放的 int 整数类型，在显示的时候才转换为货币类型。

交易信息 Trade 类，负责记录每一笔交易，它是由监听程序监听 MQ 队列而产生的，有两个属性：交易编号和交易金额，其中的交易编号对整个交易非常重要，18 位字符（在银行的交易系统中，这里可不是字符串，一般是十进制数组或二进制数字，要考虑系统的性能，数字运算可比字符运算快得多），包括 POS 机编号、商户编号、校验码等等，我们这里暂时用不到，就不多做介绍，我们只要知道它是一个非常有用的编码就成。交易金额为整数类型，实际金额放大 100 倍即可。如代码清单 35-2 所示。

代码清单35-2 交易类

```
    public class Trade {
        //交易编号
        private String tradeNo = "";
        //交易金额
        private int amount = 0;
        //getter/setter 方法
        public String getTradeNo() {
                return tradeNo;
        }

        public void setTradeNo(String postNo) {
                this.tradeNo = postNo;
        }
```

```
        public int getAmount() {
                return amount;
        }

        public void setAmount(int amount) {
                this.amount = amount;
        }
}
```

两个最简单也是在应用中最常使用的对象定义完毕，我们就需要来定义我们的策略了，非常明显的策略模式，类图如图 35-2 所示。

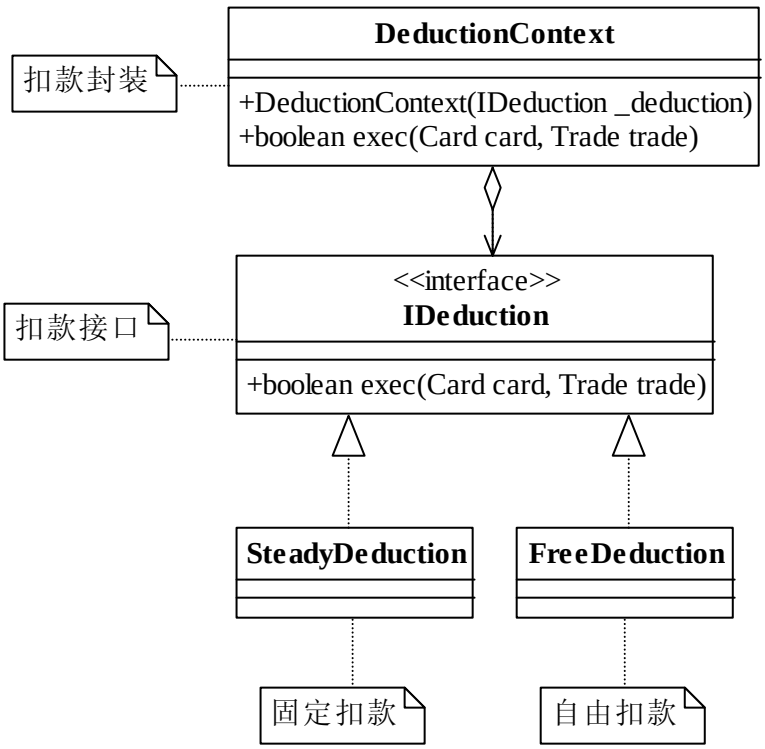

图35-2　扣款策略类图

典型的策略模式，扣款有两种策略：固定扣款和自由扣款，比较简单，不多说，我们来看代码，先看抽象策略，也就是扣款接口，如代码清单 35-3 所示。

代码清单35-3　扣款策略接口

```
public interface IDeduction {
        //扣款,提供交易和卡信息，进行扣款，并返回扣款是否成功
        public boolean exec(Card card,Trade trade);
}
```

固定扣款的规则是固定金额和自由金额各扣除交易金额的一半，如代码清单 35-4 所示。

代码清单35-4 扣款策略一

```
public class SteadyDeduction implements IDeduction {
    //固定性交易扣款
    public boolean exec(Card card, Trade trade) {
        //固定金额和自由金额各扣除 50%
        int halfMoney = (int)Math.rint(trade.getAmount() / 2.0);
        card.setFreeMoney(card.getFreeMoney() - halfMoney);
        card.setSteadyMoney(card.getSteadyMoney() - halfMoney);
        return true;
    }
}
```

这个具体策略也非常简单，就是两个金额各自减去交易额的一半（注意除数是 2.0，可不是 2），然后再四舍五入，算法确实简单。该逻辑没有考虑账户余额不足的情况，也没有考虑异常情况，比如并发情况，读者可以想想看，一张卡有两笔消费同时发生时，是不是就发生错误了？一张卡同时有两笔消费会出现这种情况吗？会的，网络阻塞的情况，MQ 多通道发送，在网络繁忙的情况下是有可能出现该问题，这里就不多介绍，有兴趣的读者可以看看 MQ 的资料。我们在这里的讲解实现的是一个快乐路径，认为所有的交易都是在安全可靠的环境中发生的，并且所有的系统环境都满足我们的要求。我们再来看另一个策略，更简单，如代码清单 35-5 所示。

代码清单35-5 扣款策略二

```
public class FreeDeduction implements IDeduction {
    //自由扣款
    public boolean exec(Card card, Trade trade) {
        //直接从自由余额中扣除
        card.setFreeMoney(card.getFreeMoney() -
trade.getAmount());
        return true;
    }
}
```

卡内的自由金额减去交易金额再修改卡内自由金额就完事了，异常情况不考虑。这两个具体的策略与我们的交易类型没有任何关系，也不应该有关系，策略模式就是提供两个可以相互替换的策略，至于在什么时候使用什么策略，则不是由策略模式来决定的。策略模式还有一个角色没出场，即封装角色，如代码清单 35-6 所示。

代码清单35-6 扣款策略的封装

```
public class DeductionContext {
    //扣款策略
    private IDeduction deduction = null;
```

```
        //构造函数传递策略
        public DeductionContext(IDeduction _deduction){
                this.deduction = _deduction;
        }
        //执行扣款
        public boolean exec(Card card,Trade trade){
                return this.deduction.exec(card, trade);
        }
    }
```

典型，太典型的策略上下文角色。扣款模块的策略已经定义完毕了，然后我们需要想办法解决策略模式的缺陷：它把所有的策略类都暴露出去，这不行，暴露得越多以后的修改风险也就越大（这是不是类似于女人的衣服：暴露得越多，被男同胞意淫的可能性就越大呢），怎么修改呢？增加一个映射配置文件，实现策略类的隐藏，我们使用枚举担当此任，对策略类进行映射处理，避免高层模块直接访问策略类，同时由工厂方法模式根据映射产生策略对象，解决得很优秀，类图如图 35-3 所示。

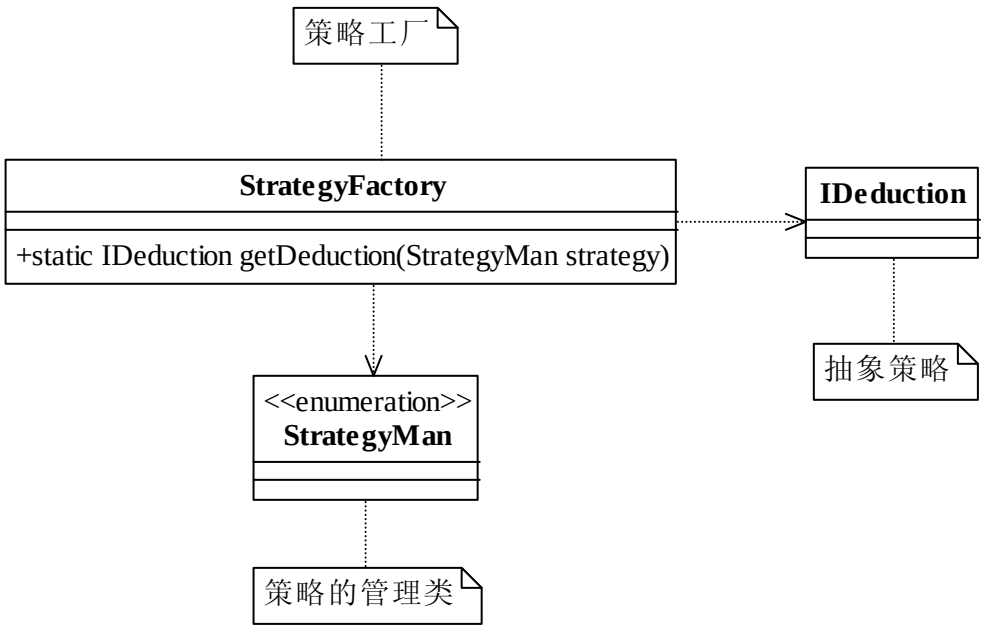

图35-3　策略工厂类图

又是一个简单得不能再简单的模式——工厂方法模式，通过 StrategyMan 负责对具体策略的映射，我们看它的代码，如代码清单 35-7 所示。

代码清单35-7 策略枚举

```
public enum StrategyMan {
```

```
        SteadyDeduction("com.cbf4life.common.SteadyDeduction"),
        FreeDeduction("com.cbf4life.common.FreeDeduction");

        String value = "";
        private StrategyMan(String _value){
                this.value = _value;
        }

        public String getValue(){
                return this.value;
        }
    }
```

类似的代码解释过很多遍了，不多说，它就是一个登记容器，所有的具体策略都在这里登记，然后提供给工厂方法模式。策略工厂如代码清单 35-8 所示。

代码清单35-8 策略工厂

```
    public class StrategyFactory {
        //策略工厂
        public static IDeduction getDeduction(StrategyMan strategy){
                IDeduction deduction = null;
                try {
                        deduction =
(IDeduction)Class.forName(strategy.getValue()).newInstance();
                }  catch (Exception e) {
                        // 异常处理
                }
                return deduction;
        }
    }
```

一个简单的工厂，根据策略管理类的枚举项创建一个策略对象，简单而实用，策略模式的缺陷也弥补成功，那~~这么复杂的系统怎么让高层模块访问？你看不出复杂？那是因为我们写的都是快乐路径，太多情况都没有考虑，在实际项目中有仅仅就并发处理和事务管理这两部分就够让你头疼了。既然系统很复杂，是不是需要封装一下，我们请出门面模式由它进行封装，如代码清单 35-9 所示。

代码清单35-9 扣款模块封装

```
    public class DeductionFacade {
        //对外公布的扣款信息
        public static Card deduct(Card card,Trade trade){
                //获得消费策略
```

```
                StrategyMan reg = getDeductionType(trade);
                //初始化一个消费策略对象
                IDeduction deduction =
StrategyFactory.getDeduction(reg);
                //产生一个策略上下问
                DeductionContext context = new
DeductionContext(deduction);
                //进行扣款处理
                context.exec(card, trade);
                //返回扣款处理完毕后的数据
                return card;
        }
        //获得对应的商户消费策略
        private static StrategyMan getDeductionType(Trade trade){
                //模拟操作
                if(trade.getTradeNo().contains("abc")){
                        return StrategyMan.FreeDeduction;
                }else{
                        return StrategyMan.SteadyDeduction;
                }
        }
   }
```

这次为什么要先展示代码而后写类图呢？那是因为这段代码比写类图更能让你理解。读者注意一下 getDeductionType 方法，这个方法在实际的项目中是存在的，但是与上面的写法有天壤之别，为啥？在实际项目中，数据库中保存了策略代码与交易编码的对应关系，直接通过数据库的 SQL 语句就可以返回对应的扣款策略。这里我们采用大家最熟悉的条件转移来实现，也是比较清晰和容易理解的。

可能读者要问了，在门面模式中已经明确地说明，门面类中不允许有业务逻辑存在，但是你这里还是有了一个 getDeductionType 方法，它可代表的是一个判断逻辑呀，这是为什么呢？是的，该方法完全可以移到其他 Hepler 类中，由于我们是示例代码，暂没有明确的业务含义，故编写在此处，读者在实际应用中，请把该方法放置到其他类中。

好，所有用到的模式都介绍完毕了，我们把完整的类图整理一下，如图 35-4 所示。

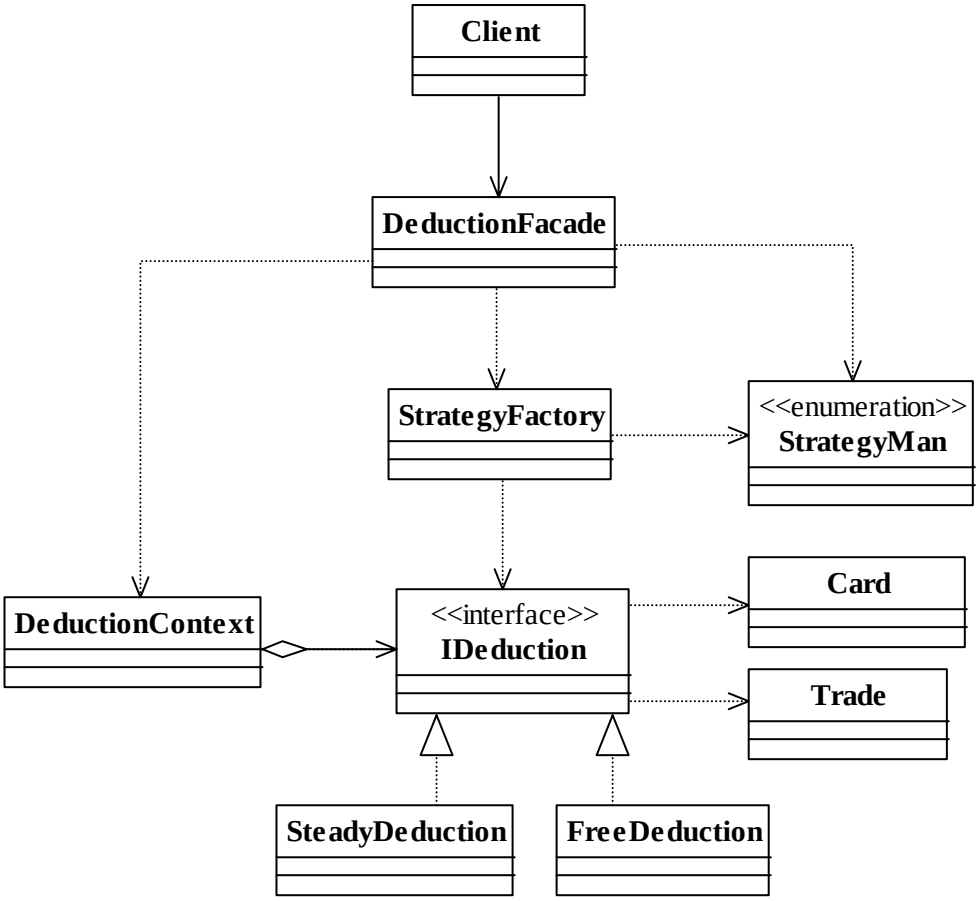

图35-4　扣款子模块完整类图

蜘蛛网了？这还不是复杂的，真实系统比这复杂得多，幸好我们在之前已经分析过，还是比较容易看懂的。我们所有的开发都完成了，是不是应该写一个测试类来展示一下我们的成果，如代码清单 35-10 所示。

代码清单35-10　场景类

```
public class Client {
    //模拟交易
    public static void main(String[] args) {
            //初始化一张 IC 卡
            Card card = initIC();
            //显示一下卡内信息
            System.out.println("========初始卡信息: =========");
            showCard(card);
            //是否停止运行标志
            boolean flag = true;
            while(flag){
                    Trade trade = createTrade();
```

```
                        DeductionFacade.deduct(card, trade);
                        //交易成功，打印出成功处理消息
                        System.out.println("\n======交易凭证========");
                        System.out.println(trade.getTradeNo()+" 交易成
功！");
                        System.out.println("本次发生的交易金额为："+
trade.getAmount()/100.0 + " 元");
                        //展示一下卡内信息
                        showCard(card);

                        System.out.print("\n 是否需要退出？(Y/N)");
                        if(getInput().equalsIgnoreCase("y")){
                                flag = false;  //退出；
                        }
                }
        }
        //初始化一个 IC 卡
        private static Card initIC(){
                Card card = new Card();
                card.setCardNo("1100010001000");
                card.setFreeMoney(100000);  //一千元
                card.setSteadyMoney(80000); //八百元
                return card;
        }
        //产生一条交易
        private static Trade createTrade(){
                Trade trade = new Trade();
                System.out.print("请输入交易编号：");
                trade.setTradeNo(getInput());
                System.out.print("请输入交易金额：");
                trade.setAmount(Integer.parseInt(getInput()));
                //返回交易
                return trade;
        }
        //打印出当前卡内交易余额
        public static void showCard(Card card){

                System.out.println("IC 卡编号:" + card.getCardNo());
                System.out.println("固定类型余额："+
card.getSteadyMoney()/100.0 + " 元");
                System.out.println("自由类型余额："+
card.getFreeMoney()/100.0 + " 元");
        }
        //获得键盘输入
```

```
        public static String getInput(){
                String str ="";
                try {
                        str = (new BufferedReader(new
InputStreamReader(System.in))).readLine();
                } catch (IOException e) {
                        //异常处理
                }
                return str;
        }
   }
```

类比较长，耐心一点，还是非常简单的，对其中 Client 类的方法说明如下：

- initIC 方法

初始化一张 IC 卡，方便我们进行测试。

- createTrade 方法

创建一笔交易，完成我们测试任务。

- showCard 方法

显示 IC 卡内的信息，你到商店买东西，刷完卡了总要给你个纸条吧，上面记录你消费了多少，现在卡内剩余多少等等，该方法的作用既是如此。

- getInput 方法

获得从键盘输入的字符，以回车符作为终结标志。

方法介绍完毕了，我们运行一下看看，结果如下所示：

```
========初始卡信息：=========
IC 卡编号:1100010001000
固定类型余额：800.0 元
自由类型余额：1000.0 元
请输入交易编号：abcdef
请输入交易金额：10000

======交易凭证========
abcdef 交易成功!
本次发生的交易金额为：100.0 元
IC 卡编号:1100010001000
固定类型余额：800.0 元
自由类型余额：900.0 元

是否需要退出？(Y/N)
```

我们模拟了一笔自由消费，直接从自由类型金额中扣除了，我们再模拟一笔固定类型的消费，运行结果如下所示：

```
========初始卡信息：=========
IC 卡编号:1100010001000
固定类型余额：800.0 元
自由类型余额：1000.0 元
请输入交易编号：abcdef
请输入交易金额：10000

======交易凭证========
abcdef 交易成功!
本次发生的交易金额为：100.0 元
IC 卡编号:1100010001000
固定类型余额：800.0 元
自由类型余额：900.0 元

是否需要退出？(Y/N)n
请输入交易编号：1001
请输入交易金额：1234

======交易凭证========
1001 交易成功!
本次发生的交易金额为：12.34 元
IC 卡编号:1100010001000
固定类型余额：793.83 元
自由类型余额：893.83 元

是否需要退出？(Y/N)
```

交易成功！到这里为止，我们的联机交易中扣款子模块开发完毕了！是不是很简单，银行业的交易系统也就是这么回事，想做交易系统？去吧！

35.2 混编小结

我们回顾一下，我们在该案例中使用了几个模式：

- 策略模式

负责对扣款策略进行封装，保证两个策略是可以自由切换的，而且日后增加扣款策略也是非常简单容易的。

- 工厂方法模式

修正策略模式必须对外暴露具体策略的问题，由工厂方法模式直接产生一个具体策略对象，而其他模块则不需要依赖具体的策略。

- 门面模式

负责对复杂的扣款系统进行封装，封装的结果就是避免高层模块深入子系统内部，同时提供系统的高内聚、低耦合的特性。

我们主要使用了这三个模式，好处是什么呢？灵活，稳定，我们可以设想一下，可能有哪些业务变化？

- 扣款策略变更

这个没问题，增加一个新扣款策略，三步就可以完成：实现 IDeduction 接口，增加 StrategyMan 配置项，扩展扣款策略的利用（也就是门面模式的的 getDeductionType 方法，在实际的项目中这里只需要增加数据库的配置项），那减少一个策略呢？简单，修改扣款策略的利用，那变更一个扣款策略呢？也简单，扩展一个实现类口可以了。

- 变更扣款策略的利用规则

也简单，我们的系统不想大修改，还记得我们提出的状态模式吗？这个就是为策略的利用服务的，变更它就能满足你的要求，想把 IC 卡也纳入策略利用的规则，也简单，不复杂。其实这个变更还真发生了，系统投产后，业务提出考虑退休人员的情况，退休人员的 IC 卡与普通在职员工一样，但是它的扣款不仅仅是根据交易编码，还要根据 IC 卡对象，系统的变更做法是增加一个扣款策略，同时扩展扣款利用策略，也就是数据库的配置项，在 getDeductionType 中新扩展了一个功能：根据 IC 卡号，确认是否是退休人员，是退休人员，则使用新的扣款策略，非常简单的扩展。

这就是一个 mini 版的金融交易系统，没啥复杂的，剩下的问题就开始考虑系统的鲁棒性吧，这才是难点。在项目中快乐路径是最容易实现的，一旦融入悲伤路径，系统的复杂性就大大提高，所以呀，才有需求分析时要先要描述一个成功场景，然后再一步一步地增加扩展场景的迭代分析模式，否则一上来就考虑一堆的扩展场景，这会让你思维混乱、手足无措，直接进入郁闷假死状态。