几种常用的设计模式介绍

1.    设计模式的起源

最早提出“设计模式”概念的是建筑设计大师亚力山大Alexander。在1970年他的《建筑的永恒之道》里描述了投计模式的发现，因为它已经存在了千百年之久，而现代才被通过大量的研究而被发现。

在《建筑的永恒之道》里这样描述：模式是一条由三个部分组成的通用规则：它表示了一个特定环境、一类问题和一个解决方案之间的关系。每一个模式描述了一个不断重复发生的问题，以及该问题解决方案的核心设计。

在他的另一本书《建筑模式语言》中提到了现在已经定义了253种模式。比如：

说明城市主要的结构：亚文化区的镶嵌、分散的工作点、城市的魅力、地方交通区

住宅团组：户型混合、公共性的程度、住宅团组、联排式住宅、丘状住宅、老人天地室内环境和室外环境、阴和阳总是一气呵成

针对住宅：夫妻的领域、儿童的领域、朝东的卧室、农家的厨房、私家的沿街露台、个人居室、起居空间的序列、多床卧室、浴室、大储藏室

针对办公室、车间和公共建筑物：灵活办公空间、共同进餐、共同小组、宾至如归、等候场所、小会议室、半私密办公室

尽管亚力山大的著作是针对建筑领域的，但他的观点实际上适用于所有的工程设计领域，其中也包括软件设计领域。“软件设计模式”，这个术语是在1990年代由Erich Gamma等人从建筑设计领域引入到计算机科学中来的。目前主要有23种。

2.    软件设计模式的分类

2.1.  创建型

创建对象时，不再由我们直接实例化对象；而是根据特定场景，由程序来确定创建对象的方式，从而保证更大的性能、更好的[架构](http://lib.csdn.net/base/architecture)优势。创建型模式主要有简单工厂模式（并不是23种设计模式之一）、工厂方法、抽象工厂模式、单例模式、生成器模式和原型模式。

2.2.  结构型

用于帮助将多个对象组织成更大的结构。结构型模式主要有适配器模式adapter、桥接模式bridge、组合器模式component、装饰器模式decorator、门面模式、亨元模式flyweight和代理模式proxy。

2.3.  行为型

用于帮助系统间各对象的通信，以及如何控制复杂系统中流程。行为型模式主要有命令模式command、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式state、策略模式、模板模式和访问者模式。

3.    常见设计模式介绍

3.1.  单例模式(singleton)

有些时候，允许自由创建某个类的实例没有意义，还可能造成系统性能下降。如果一个类始终只能创建一个实例，则这个类被称为单例类，这种模式就被称为单例模式。

    一般建议单例模式的方法命名为：getInstance()，这个方法的返回类型肯定是单例类的类型了。getInstance方法可以有参数，这些参数可能是创建类实例所需要的参数，当然，大多数情况下是不需要的

|  |
| --- |
| publicclass Singleton {        publicstaticvoid main(String[] args)      {         //创建Singleton对象不能通过构造器，只能通过getInstance方法         Singleton s1 = Singleton.getInstance();         Singleton s2 = Singleton.getInstance();         //将输出true         System.out.println(s1 == s2);      }        //使用一个变量来缓存曾经创建的实例      privatestatic Singleton instance;      //将构造器使用private修饰，隐藏该构造器      private Singleton(){         System.out.println("Singleton被构造！");      }        //提供一个静态方法，用于返回Singleton实例      //该方法可以加入自定义的控制，保证只产生一个Singleton对象      publicstatic Singleton getInstance()      {         //如果instance为null，表明还不曾创建Singleton对象         //如果instance不为null，则表明已经创建了Singleton对象，将不会执行该方法         if (instance == null)         {             //创建一个Singleton对象，并将其缓存起来             instance = new Singleton();         }         returninstance;      }  } |

单例模式主要有如下两个优势：

1)      减少创建[Java](http://lib.csdn.net/base/java)实例所带来的系统开销

2)      便于系统跟踪单个Java实例的生命周期、实例状态等。

3.2.  简单工厂(StaticFactory Method)

简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。简单工厂模式是工厂模式家族中最简单实用的模式，可以理解为是不同工厂模式的一个特殊实现。

A实例调用B实例的方法，称为A依赖于B。如果使用new关键字来创建一个B实例（硬编码耦合），然后调用B实例的方法。一旦系统需要重构：需要使用C类来代替B类时，程序不得不改写A类代码。而用工厂模式则不需要关心B对象的实现、创建过程。

Output，接口

|  |
| --- |
| publicinterface Output  {      //接口里定义的属性只能是常量      intMAX\_CACHE\_LINE = 50;      //接口里定义的只能是public的抽象实例方法      void out();      void getData(String msg);  } |

Printer，Output的一个实现

|  |
| --- |
| //让Printer类实现Output  publicclass Printer implements Output  {      private String[] printData = new String[MAX\_CACHE\_LINE];      //用以记录当前需打印的作业数      privateintdataNum = 0;      publicvoid out()      {         //只要还有作业，继续打印         while(dataNum > 0)         {             System.out.println("打印机打印：" + printData[0]);             //把作业队列整体前移一位，并将剩下的作业数减1             System.arraycopy(printData , 1, printData, 0, --dataNum);         }      }      publicvoid getData(String msg)      {         if (dataNum >= MAX\_CACHE\_LINE)         {             System.out.println("输出队列已满，添加失败");         }         else         {             //把打印数据添加到队列里，已保存数据的数量加1。             printData[dataNum++] = msg;         }      }  } |

BetterPrinter，Output的一个实现

|  |
| --- |
| publicclass BetterPrinter implements Output  {      private String[] printData = new String[MAX\_CACHE\_LINE \* 2];      //用以记录当前需打印的作业数      privateintdataNum = 0;      publicvoid out()      {         //只要还有作业，继续打印         while(dataNum > 0)         {             System.out.println("高速打印机正在打印：" + printData[0]);             //把作业队列整体前移一位，并将剩下的作业数减1             System.arraycopy(printData , 1, printData, 0, --dataNum);         }      }      publicvoid getData(String msg)      {         if (dataNum >= MAX\_CACHE\_LINE \* 2)         {             System.out.println("输出队列已满，添加失败");         }         else         {             //把打印数据添加到队列里，已保存数据的数量加1。             printData[dataNum++] = msg;         }      }  } |

OutputFactory，简单工厂类

|  |
| --- |
| public Output getPrinterOutput(String type) {         if (type.equalsIgnoreCase("better")) {             returnnew BetterPrinter();         } else {             returnnew Printer();         }      } |

Computer

|  |
| --- |
| publicclass Computer  {      private Output out;        public Computer(Output out)      {         this.out = out;      }      //定义一个模拟获取字符串输入的方法      publicvoid keyIn(String msg)      {         out.getData(msg);      }      //定义一个模拟打印的方法      publicvoid print()      {         out.out();      }      publicstaticvoid main(String[] args)      {         //创建OutputFactory         OutputFactory of = new OutputFactory();         //将Output对象传入，创建Computer对象         Computer c = new Computer(of.getPrinterOutput("normal"));         c.keyIn("建筑永恒之道");         c.keyIn("建筑模式语言");         c.print();             c = new Computer(of.getPrinterOutput("better"));         c.keyIn("建筑永恒之道");         c.keyIn("建筑模式语言");         c.print();      } |

使用简单工厂模式的优势：让对象的调用者和对象创建过程分离，当对象调用者需要对象时，直接向工厂请求即可。从而避免了对象的调用者与对象的实现类以硬编码方式耦合，以提高系统的可维护性、可扩展性。工厂模式也有一个小小的缺陷：当产品修改时，工厂类也要做相应的修改。

3.3.  工厂方法(Factory Method)和抽象工厂(Abstract Factory)

如果我们不想在工厂类中进行逻辑判断，程序可以为不同产品类提供不同的工厂，不同的工厂类和产不同的产品。

当使用工厂方法设计模式时，对象调用者需要与具体的工厂类耦合，如：

|  |
| --- |
| //工厂类的定义1  publicclass BetterPrinterFactory      implements OutputFactory  {      public Output getOutput()      {         //该工厂只负责产生BetterPrinter对象         returnnew BetterPrinter();      }  }  //工厂类的定义2  publicclass PrinterFactory      implements OutputFactory  {      public Output getOutput()      {         //该工厂只负责产生Printer对象         returnnew Printer();      }  }  //工厂类的调用  //OutputFactory of = new BetterPrinterFactory();  OutputFactory of = new PrinterFactory();  Computer c = new Computer(of.getOutput()); |

使用简单工厂类，需要在工厂类里做逻辑判断。而工厂类虽然不用在工厂类做判断。但是带来了另一种耦合：客户端代码与不同的工厂类耦合。

为了解决客户端代码与不同工厂类耦合的问题。在工厂类的基础上再增加一个工厂类，该工厂类不制造具体的被调用对象，而是制造不同工厂对象。如：

|  |
| --- |
| //抽象工厂类的定义，在工厂类的基础上再建一个工厂类  publicclass OutputFactoryFactory  {      //仅定义一个方法用于返回输出设备。      publicstatic OutputFactory getOutputFactory(         String type)      {         if (type.equalsIgnoreCase("better"))         {             returnnew BetterPrinterFactory();         }         else         {             returnnew PrinterFactory();         }      }  }    //抽象工厂类的调用  OutputFactory of = OutputFactoryFactory.getOutputFactory("better");  Computer c = new Computer(of.getOutput()); |

3.4.  代理模式(Proxy)

代理模式是一种应用非常广泛的设计模式，当客户端代码需要调用某个对象时，客户端实际上不关心是否准确得到该对象，它只要一个能提供该功能的对象即可，此时我们就可返回该对象的代理（Proxy）。

代理就是一个Java对象代表另一个Java对象来采取行动。如：

|  |
| --- |
| publicclass ImageProxy implements Image  {      //组合一个image实例，作为被代理的对象      private Image image;      //使用抽象实体来初始化代理对象      public ImageProxy(Image image)      {         this.image = image;      }      /\*\*       \* 重写Image接口的show()方法       \* 该方法用于控制对被代理对象的访问，       \* 并根据需要负责创建和删除被代理对象       \*/      publicvoid show()      {         //只有当真正需要调用image的show方法时才创建被代理对象         if (image == null)         {             image = new BigImage();         }         image.show();      }  } |

调用时，先不创建：

|  |
| --- |
| Image image = new ImageProxy(null); |

[hibernate](http://lib.csdn.net/base/javaee)默认启用延迟加载，当系统加载A实体时，A实体关联的B实体并未被加载出来，A实体所关联的B实体全部是代理对象——只有等到A实体真正需要访问B实体时，系统才会去[数据库](http://lib.csdn.net/base/mysql)里抓取B实体所对应的记录。

借助于Java提供的Proxy和InvocationHandler，可以实现在运行时生成动态代理的功能，而动态代理对象就可以作为目标对象使用，而且增强了目标对象的功能。如：

Panther

|  |
| --- |
| publicinterface Panther  {      //info方法声明      publicvoid info();      //run方法声明      publicvoid run();  } |

GunPanther

|  |
| --- |
| publicclass GunPanther implements Panther  {      //info方法实现，仅仅打印一个字符串      publicvoid info()      {         System.out.println("我是一只猎豹！");      }      //run方法实现，仅仅打印一个字符串      publicvoid run()      {         System.out.println("我奔跑迅速");      }  } |

MyProxyFactory，创建代理对象

|  |
| --- |
| publicclass MyProxyFactory  {      //为指定target生成动态代理对象      publicstatic Object getProxy(Object target)         throws Exception      {         //创建一个MyInvokationHandler对象         MyInvokationHandler handler =             new MyInvokationHandler();         //为MyInvokationHandler设置target对象         handler.setTarget(target);         //创建、并返回一个动态代理         return Proxy.newProxyInstance(target.getClass().getClassLoader()             , target.getClass().getInterfaces(), handler);      }  } |

MyInvokationHandler，增强代理的功能

|  |
| --- |
| publicclass MyInvokationHandler implements InvocationHandler  {      //需要被代理的对象      private Object target;      publicvoid setTarget(Object target)      {         this.target = target;      }      //执行动态代理对象的所有方法时，都会被替换成执行如下的invoke方法      public Object invoke(Object proxy, Method method, Object[] args)         throws Exception      {         TxUtil tx = new TxUtil();         //执行TxUtil对象中的beginTx。         tx.beginTx();         //以target作为主调来执行method方法         Object result = method.invoke(target , args);         //执行TxUtil对象中的endTx。         tx.endTx();         return result;      }  } |

TxUtil

|  |
| --- |
| publicclass TxUtil  {      //第一个拦截器方法:模拟事务开始      publicvoid beginTx()      {         System.out.println("=====模拟开始事务=====");      }      //第二个拦截器方法:模拟事务结束      publicvoid endTx()      {         System.out.println("=====模拟结束事务=====");      }  } |

[测试](http://lib.csdn.net/base/softwaretest)

|  |
| --- |
| publicstaticvoid main(String[] args)         throws Exception      {         //创建一个原始的GunDog对象，作为target         Panther target = new GunPanther();         //以指定的target来创建动态代理         Panther panther = (Panther)MyProxyFactory.getProxy(target);         //调用代理对象的info()和run()方法         panther.info();         panther.run();      } |

[spring](http://lib.csdn.net/base/javaee)所创建的AOP代理就是这种动态代理。但是Spring AOP更灵活。

3.5.  命令模式(Command)

某个方法需要完成某一个功能，完成这个功能的大部分步骤已经确定了，但可能有少量具体步骤无法确定，必须等到执行该方法时才可以确定。（在某些编程语言如Ruby、Perl里，允许传入一个代码块作为参数。但Jara暂时还不支持代码块作为参数）。在Java中，传入该方法的是一个对象，该对象通常是某个接口的匿名实现类的实例，该接口通常被称为命令接口，这种设计方式也被称为命令模式。

如：

Command

|  |
| --- |
| publicinterface Command  {      //接口里定义的process方法用于封装“处理行为”      void process(int[] target);  } |

ProcessArray

|  |
| --- |
| publicclass ProcessArray  {      //定义一个each()方法，用于处理数组，      publicvoid each(int[] target , Command cmd)      {         cmd.process(target);      }  } |

TestCommand

|  |
| --- |
| publicclass TestCommand  {      publicstaticvoid main(String[] args)      {         ProcessArray pa = new ProcessArray();         int[] target = {3, -4, 6, 4};         //第一次处理数组，具体处理行为取决于Command对象         pa.each(target , new Command()         {             //重写process()方法，决定具体的处理行为             publicvoid process(int[] target)             {                for (int tmp : target )                {                    System.out.println("迭代输出目标数组的元素:" + tmp);                }             }         });         System.out.println("------------------");         //第二次处理数组，具体处理行为取决于Command对象         pa.each(target , new Command()         {             //重写process方法，决定具体的处理行为             publicvoid process(int[] target)             {                int sum = 0;                for (int tmp : target )                {                    sum += tmp;                }                System.out.println("数组元素的总和是:" + sum);             }         });      }  } |

HibernateTemplate使用了executeXxx()方法弥补了HibernateTemplate的不足，该方法需要接受一个HibernateCallback接口，该接口的代码如下：

|  |
| --- |
| public interface HibernateCallback  {         Object doInHibernate(Session session);  } |

3.6.  策略模式(Strategy)

策略模式用于封装系列的[算法](http://lib.csdn.net/base/datastructure)，这些算法通常被封装在一个被称为Context的类中，客户端程序可以自由选择其中一种算法，或让Context为客户端选择一种最佳算法——使用策略模式的优势是为了支持算法的自由切换。

DiscountStrategy，折扣方法接口

|  |
| --- |
| publicinterface DiscountStrategy  {      //定义一个用于计算打折价的方法      double getDiscount(double originPrice);  } |

OldDiscount，旧书打折算法

|  |
| --- |
| publicclass OldDiscount implements DiscountStrategy {      // 重写getDiscount()方法，提供旧书打折算法      publicdouble getDiscount(double originPrice) {         System.out.println("使用旧书折扣...");         return originPrice \* 0.7;      }  } |

VipDiscount，VIP打折算法

|  |
| --- |
| //实现DiscountStrategy接口，实现对VIP打折的算法  publicclass VipDiscount implements DiscountStrategy {      // 重写getDiscount()方法，提供VIP打折算法      publicdouble getDiscount(double originPrice) {         System.out.println("使用VIP折扣...");         return originPrice \* 0.5;      }  } |

策略定义

|  |
| --- |
| publicclass DiscountContext  {      //组合一个DiscountStrategy对象      private DiscountStrategy strategy;      //构造器，传入一个DiscountStrategy对象      public DiscountContext(DiscountStrategy strategy)      {         this.strategy  = strategy;      }      //根据实际所使用的DiscountStrategy对象得到折扣价      publicdouble getDiscountPrice(double price)      {         //如果strategy为null，系统自动选择OldDiscount类         if (strategy == null)         {             strategy = new OldDiscount();         }         returnthis.strategy.getDiscount(price);      }      //提供切换算法的方法      publicvoid setDiscount(DiscountStrategy strategy)      {         this.strategy = strategy;      }  } |

测试

|  |
| --- |
| publicstaticvoid main(String[] args)      {         //客户端没有选择打折策略类         DiscountContext dc = new DiscountContext(null);         double price1 = 79;         //使用默认的打折策略         System.out.println("79元的书默认打折后的价格是："             + dc.getDiscountPrice(price1));         //客户端选择合适的VIP打折策略         dc.setDiscount(new VipDiscount());         double price2 = 89;         //使用VIP打折得到打折价格         System.out.println("89元的书对VIP用户的价格是："             + dc.getDiscountPrice(price2));      } |

使用策略模式可以让客户端代码在不同的打折策略之间切换，但也有一个小小的遗憾：客户端代码需要和不同的策略耦合。为了弥补这个不足，我们可以考虑使用配置文件来指定DiscountContext使用哪种打折策略——这就彻底分离客户端代码和具体打折策略类。

3.7.  门面模式(Facade)

随着系统的不断改进和开发，它们会变得越来越复杂，系统会生成大量的类，这使得程序流程更难被理解。门面模式可为这些类提供一个简化的接口，从而简化访问这些类的复杂性。

       门面模式（Facade）也被称为正面模式、外观模式，这种模式用于将一组复杂的类包装到一个简单的外部接口中。

原来的方式

|  |
| --- |
| // 依次创建三个部门实例         Payment pay = new PaymentImpl();         Cook cook = new CookImpl();         Waiter waiter = new WaiterImpl();         // 依次调用三个部门实例的方法来实现用餐功能         String food = pay.pay();         food = cook.cook(food);         waiter.serve(food); |

门面模式

|  |
| --- |
| publicclass Facade {      // 定义被Facade封装的三个部门      Payment pay;      Cook cook;      Waiter waiter;        // 构造器      public Facade() {         this.pay = new PaymentImpl();         this.cook = new CookImpl();         this.waiter = new WaiterImpl();      }        publicvoid serveFood() {         // 依次调用三个部门的方法，封装成一个serveFood()方法         String food = pay.pay();         food = cook.cook(food);         waiter.serve(food);      }  } |

门面模式调用

|  |
| --- |
| Facade f = new Facade();         f.serveFood(); |

3.8.  桥接模式(Bridge)

由于实际的需要，某个类具有两个以上的维度变化，如果只是使用继承将无法实现这种需要，或者使得设计变得相当臃肿。而桥接模式的做法是把变化部分抽象出来，使变化部分与主类分离开来，从而将多个的变化彻底分离。最后提供一个管理类来组合不同维度上的变化，通过这种组合来满足业务的需要。

Peppery口味风格接口：

|  |
| --- |
| publicinterface Peppery  {      String style();  } |

口味之一

|  |
| --- |
| publicclass PepperySytle implements Peppery  {      //实现"辣味"风格的方法      public String style()      {         return"辣味很重，很过瘾...";      }  } |

口味之二

|  |
| --- |
| publicclass PlainStyle implements Peppery  {      //实现"不辣"风格的方法      public String style()      {         return"味道清淡，很养胃...";      }  } |

口味的桥梁

|  |
| --- |
| publicabstractclass AbstractNoodle  {      //组合一个Peppery变量，用于将该维度的变化独立出来      protected Peppery style;      //每份Noodle必须组合一个Peppery对象      public AbstractNoodle(Peppery style)      {         this.style = style;      }      publicabstractvoid eat();  } |

材料之一，继承口味

|  |
| --- |
| publicclass PorkyNoodle extends AbstractNoodle  {      public PorkyNoodle(Peppery style)      {         super(style);      }      //实现eat()抽象方法      publicvoid eat()      {         System.out.println("这是一碗稍嫌油腻的猪肉面条。"             + super.style.style());      }  } |

材料之二，继承口味

|  |
| --- |
| publicclass BeefMoodle extends AbstractNoodle  {      public BeefMoodle(Peppery style)      {         super(style);      }      //实现eat()抽象方法      publicvoid eat()      {         System.out.println("这是一碗美味的牛肉面条。"             + super.style.style());      }  } |

主程序

|  |
| --- |
| publicclass Test  {      publicstaticvoid main(String[] args)      {         //下面将得到“辣味”的牛肉面         AbstractNoodle noodle1 = new BeefMoodle(             new PepperySytle());         noodle1.eat();         //下面将得到“不辣”的牛肉面         AbstractNoodle noodle2 = new BeefMoodle(             new PlainStyle());         noodle2.eat();         //下面将得到“辣味”的猪肉面         AbstractNoodle noodle3 = new PorkyNoodle(             new PepperySytle());         noodle3.eat();         //下面将得到“不辣”的猪肉面         AbstractNoodle noodle4 = new PorkyNoodle(             new PlainStyle());         noodle4.eat();      }  } |

[Java](http://lib.csdn.net/base/java) EE应用中常见的DAO模式正是桥接模式的应用。

实际上，一个设计优良的项目，本身就是设计模式最好的教科书，例如Spring框架，当你深入阅读其源代码时，你会发现这个框架处处充满了设计模式的应用场景。

http://www.cnblogs.com/liuling/archive/2013/04/20/observer.html

3.9.  观察者模式(Observer)

　观察者模式结构中包括四种角色：

　　一、主题：主题是一个接口，该接口规定了具体主题需要实现的方法，比如添加、删除观察者以及通知观察者更新数据的方法。

　　二、观察者：观察者也是一个接口，该接口规定了具体观察者用来更新数据的方法。

　　三、具体主题：具体主题是一个实现主题接口的类，该类包含了会经常发生变化的数据。而且还有一个集合，该集合存放的是观察者的引用。

　　四：具体观察者：具体观察者是实现了观察者接口的一个类。具体观察者包含有可以存放具体主题引用的主题接口变量，以便具体观察者让具体主题将自己的引用添加到具体主题的集合中，让自己成为它的观察者，或者让这个具体主题将自己从具体主题的集合中删除，使自己不在时它的观察者。

观察者模式定义了对象间的一对多依赖关系，让一个或多个观察者对象观察一个主题对象。当主题对象的状态发生变化时，系统能通知所有的依赖于此对象的观察者对象，从而使得观察者对象能够自动更新。

在观察者模式中，被观察的对象常常也被称为目标或主题（Subject），依赖的对象被称为观察者（Observer）。

Observer，观察者接口：

观察者：观察者也是一个接口，该接口规定了具体观察者用来更新数据的方法

|  |
| --- |
| publicinterface Observer {      void update(Observable o, Object arg);  } |

Observable，目标或主题：

主题：主题是一个接口，该接口规定了具体主题需要实现的方法，比如添加、删除观察者以及通知观察者更新数据的方法

|  |
| --- |
| import java.util.ArrayList;  import java.util.List;  import java.util.Iterator;    publicabstractclass Observable {      // 用一个List来保存该对象上所有绑定的事件监听器      List<Observer> observers = new ArrayList<Observer>();        // 定义一个方法，用于从该主题上注册观察者      publicvoid registObserver(Observer o) {         observers.add(o);      }        // 定义一个方法，用于从该主题中删除观察者      publicvoid removeObserver(Observer o) {         observers.add(o);      }        // 通知该主题上注册的所有观察者      publicvoid notifyObservers(Object value) {         // 遍历注册到该被观察者上的所有观察者         for (Iterator it = observers.iterator(); it.hasNext();) {             Observer o = (Observer) it.next();             // 显式每个观察者的update方法             o.update(this, value);         }      }  } |

Product被观察类：

具体主题：具体主题是一个实现主题接口的类，该类包含了会经常发生变化的数据。而且还有一个集合，该集合存放的是观察者的引用。

|  |
| --- |
| publicclass Product extends Observable {      // 定义两个属性      private String name;      privatedoubleprice;        // 无参数的构造器      public Product() {      }        public Product(String name, double price) {         this.name = name;         this.price = price;      }        public String getName() {         returnname;      }        // 当程序调用name的setter方法来修改Product的name属性时      // 程序自然触发该对象上注册的所有观察者      publicvoid setName(String name) {         this.name = name;         notifyObservers(name);      }        publicdouble getPrice() {         returnprice;      }        // 当程序调用price的setter方法来修改Product的price属性时      // 程序自然触发该对象上注册的所有观察者      publicvoid setPrice(double price) {         this.price = price;         notifyObservers(price);      }  } |

具体观察者：具体观察者是实现了观察者接口的一个类。具体观察者包含有可以存放具体主题引用的主题接口变量，以便具体观察者让具体主题将自己的引用添加到具体主题的集合中，让自己成为它的观察者，或者让这个具体主题将自己从具体主题的集合中删除，使自己不在时它的观察者。

NameObserver名称观察者：

|  |
| --- |
| import javax.swing.JFrame;  import javax.swing.JLabel;    publicclass NameObserver implements Observer {      // 实现观察者必须实现的update方法      publicvoid update(Observable o, Object arg) {         if (arg instanceof String) {             // 产品名称改变值在name中             String name = (String) arg;             // 启动一个JFrame窗口来显示被观察对象的状态改变             JFrame f = new JFrame("观察者");             JLabel l = new JLabel("名称改变为：" + name);             f.add(l);             f.pack();             f.setVisible(true);             System.out.println("名称观察者:" + o + "物品名称已经改变为: " + name);         }      }  } |

PriceObserver价格观察者：

|  |
| --- |
| publicclass PriceObserver implements Observer {      // 实现观察者必须实现的update方法      publicvoid update(Observable o, Object arg) {         if (arg instanceof Double) {             System.out.println("价格观察者:" + o + "物品价格已经改变为: " + arg);         }      }  } |

测试：

|  |
| --- |
| publicclass Test {      publicstaticvoid main(String[] args) {         // 创建一个被观察者对象         Product p = new Product("电视机", 176);         // 创建两个观察者对象         NameObserver no = new NameObserver();         PriceObserver po = new PriceObserver();         // 向被观察对象上注册两个观察者对象         p.registObserver(no);         p.registObserver(po);         // 程序调用setter方法来改变Product的name和price属性         p.setName("书桌");         p.setPrice(345f);      }  } |

其中Java工具类提供了被观察者抽象基类：java.util.Observable。观察者接口：java.util.Observer。

我们可以把观察者接口理解成事件监听接口，而被观察者对象也可当成事件源处理——换个角度来思考：监听，观察，这两个词语之间有本质的区别吗？Java事件机制的底层实现，本身就是通过观察者模式来实现的。除此之外，主题/订阅模式下的JMS本身就是观察者模式的应用。