java:23 种设计模式

JAVA 的设计模式经前人总结可以分为 23 种

设计模式根据使用类型可以分为三种：

1、 创建模式：**Factory**（工厂模式）、**Singleton**(单态)、**Builder**(建造者模式)、**Prototype**(原型模式)、工厂方法模式。

2、结构模式：**Flyweight**(共享模式)、**Bridge**(桥模式)、**Decorator**(装饰模式)、**Composite**(组合模式)、**Adapter**(适配器模式)、**Proxy**(代理模式)、**Facade** (外观模式)。

3、 行为模式：**Iterator**（迭代模式）、**Template**(模板模式)、**Chain of Responsibility**(责任链模式)、**Memento**(纪念品模式)、**Mediator**(中介模式)、**Interpreter**(解释器模式)、**Strategy**（策略模式）、**State** 、**Observer**(观察者模式)、**Visitor**(访问模式)、**Command**(命令模式)。

注：以上翻译不甚准确，以英文为准。

创建模式

设计模式之 Factory

工厂模式定义:提供创建对象的接口.

为何使用?

工厂模式是我们最常用的模式了,著名的 Jive 论坛 ,就大量使用了工厂模式，工厂模式在 Java 程序系统可以说是随处可见。

为什么工厂模式是如此常用？因为工厂模式就相当于创建实例对象的 new,我们经常要根据类 Class 生成实例对象，如 A a=new A() 工厂模式也是用来创建实例对象的，所以以后 new 时就要多个心眼，是否可以考虑实用工厂模式，虽然这样做，可能多做一些工作，但会给你系统带来更大的可扩展性和尽量少的修改量。

我们以类 Sample 为例， 如果我们要创建 Sample 的实例对象:

```
Sample sample=new Sample();
```

可是，实际情况是，通常我们都要在创建 sample 实例时做点初始化的工作,比如赋值 查询数据库等。

首先，我们想到的是，可以使用 Sample 的构造函数，这样生成实例就写成:

```
Sample sample=new Sample(参数);
```

但是，如果创建 sample 实例时所做的初始化工作不是象赋值这样简单的事，可能是很长一段代码，如果也写入构造函数中，那你的代码很难看了（就需要 Refactor 重整）。

为什么说代码很难看，初学者可能没有这种感觉，我们分析如下，初始化工作如果是很长一段代码，说明要做的工作很多，将很多工作装入一个方法中，相当于将很多鸡蛋放在一个篮子里,是很危险的,这也是有背于 Java 面向对象的原则,面向对象的封装(Encapsulation)

和分派(Delegation)告诉我们，尽量将长的代码分派“切割”成每段，将每段再“封装”起来(减少段和段之间偶合联系性)，这样，就会将风险分散，以后如果需要修改，只要更改每段，不会再发生牵一动百的事情。

在本例中，首先，我们需要将创建实例的工作与使用实例的工作分开，也就是说，让创建实例所需要的大量初始化工作从 Sample 的构造函数中分离出去。

这时我们就需要 Factory 工厂模式来生成对象了,不能再用上面简单 new Sample(参数)。还有,如果 Sample 有个继承如 MySample, 按照面向接口编程,我们需要将 Sample 抽象成一个接口.现在 Sample 是接口,有两个子类 MySample 和 HisSample .我们要实例化他们时,如下:

```
Sample mysample=new MySample();
Sample hissample=new HisSample();
```

随着项目的深入,Sample 可能还会"生出很多儿子出来", 那么我们要对这些儿子一个个实例化,更糟糕的是,可能还要对以前的代码进行修改:加入后来生出儿子的实例.这在传统程序中是无法避免的.

但如果你一开始就有意识使用了工厂模式,这些麻烦就没有了.

工厂方法

你会建立一个专门生产 Sample 实例的工厂:

```
public class Factory{

   public static Sample creator(int which){

   //getClass 产生 Sample 一般可使用动态类装载装入类。
   if (which==1)
       return new SampleA();
   else if (which==2)
```

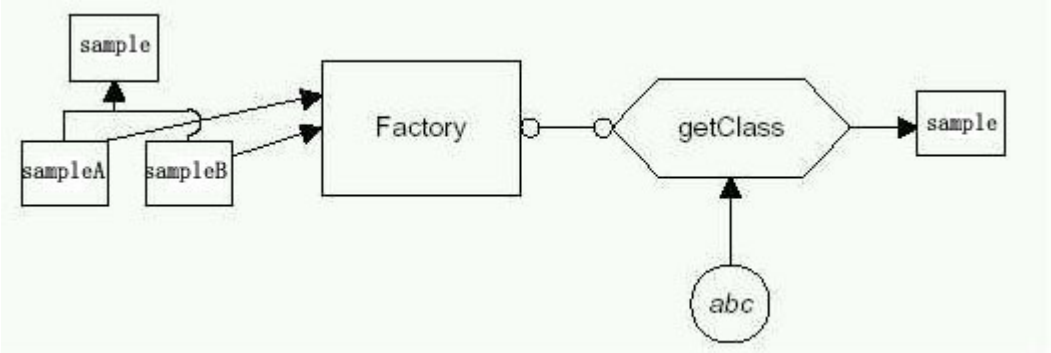
```
        return new SampleB();

    }

}
```

那么在你的程序中,如果要实例化 Sample 时.就使用

```
Sample sampleA=Factory.creator(1);
```

这样,在整个就不涉及到 Sample 的具体子类,达到封装效果,也就减少错误修改的机会,这个原理可以用很通俗的话来比喻:就是具体事情做得越多,越容易范错误.这每个做过具体工作的人都深有体会,相反,官做得越高,说出的话越抽象越笼统,范错误可能性就越少.好象我们从编程序中也能悟出人生道理?呵呵.

使用工厂方法 要注意几个角色，首先你要定义产品接口，如上面的 Sample,产品接口下有 Sample 接口的实现类,如 SampleA,其次要有一个 factory 类，用来生成产品 Sample，如下图，最右边是生产的对象 Sample：



进一步稍微复杂一点，就是在工厂类上进行拓展，工厂类也有继承它的实现类 concreteFactory 了。

抽象工厂

工厂模式中有: 工厂方法(Factory Method) 抽象工厂(Abstract Factory).

这两个模式区别在于需要创建对象的复杂程度上。如果我们创建对象的方法变得复杂了,如上面工厂方法中是创建一个对象 Sample,如果我们还有新的产品接口 Sample2.

这里假设：Sample 有两个 concrete 类 SampleA 和 SamleB，而 Sample2 也有两个 concrete 类 Sample2A 和 SampleB2

那么,我们就将上例中 Factory 变成抽象类,将共同部分封装在抽象类中,不同部分使用子类实现，下面就是将上例中的 Factory 拓展成抽象工厂：

```
public abstract class Factory{

    public abstract Sample creator();

    public abstract Sample2 creator(String
name);

}

public class SimpleFactory extends Factory{

    public Sample creator(){
        .........
        return new SampleA
    }

    public Sample2 creator(String name){
        .........
        return new Sample2A
    }

}

public class BombFactory extends Factory{
```

```
    public Sample creator(){
        ......
        return new SampleB
    }

    public Sample2 creator(String name){
        ......
        return new Sample2B
    }

}
```

从上面看到两个工厂各自生产出一套 Sample 和 Sample2,也许你会疑问，为什么我不可以使用两个工厂方法来分别生产 Sample 和 Sample2?

抽象工厂还有另外一个关键要点，是因为 SimpleFactory 内，生产 Sample 和生产 Sample2 的方法之间有一定联系，所以才要将这两个方法捆绑在一个类中，这个工厂类有其本身特征，也许制造过程是统一的，比如：制造工艺比较简单，所以名称叫 SimpleFactory。

在实际应用中，工厂方法用得比较多一些，而且是和动态类装入器组合在一起应用，

举例<?XML:NAMESPACE PREFIX = O /><O:P></O:P>

我们以 Jive 的 ForumFactory 为例,这个例子在前面的 Singleton 模式中我们讨论过，现在再讨论其工厂模式:

```
public abstract class ForumFactory {

    private static Object initLock = new Object();
    private static String className =
```

```
"com.jivesoftware.forum.database.DbForumFactory";
   private static ForumFactory factory = null;

   public static ForumFactory getInstance(Authorization
authorization) {
      //If no valid authorization passed in, return null.
      if (authorization == null) {
         return null;
      }
      //以下使用了 Singleton 单态模式
      if (factory == null) {
         synchronized(initLock) {
            if (factory == null) {
                  ......

               try {
                     //动态转载类
                     Class c = Class.forName(className);
                     factory = (ForumFactory)c.newInstance();
               }
               catch (Exception e) {
                     return null;
               }
            }
         }
      }

      //Now, 返回 proxy.用来限制授权对 forum 的访问
      return new ForumFactoryProxy(authorization, factory,
```

```
factory.getPermissions(authorization));
    }

    //真正创建 forum 的方法由继承 forumfactory 的子类去完成.
    public abstract Forum createForum(String name, String
description)
    throws UnauthorizedException, ForumAlreadyExistsException;

    ....

}
```

因为现在的 Jive 是通过数据库系统存放论坛帖子等内容数据,如果希望更改为通过文件系统实现,这个工厂方法 ForumFactory 就提供了提供动态接口:

```
private static String className =
"com.jivesoftware.forum.database.DbForumFactory";
```

你可以使用自己开发的创建 forum 的方法代替 com.jivesoftware.forum.database.DbForumFactory 就可以.

在上面的一段代码中一共用了三种模式,除了工厂模式外,还有 Singleton 单态模式,以及 proxy 模式,proxy 模式主要用来授权用户对 forum 的访问,因为访问 forum 有两种人:一个是注册用户 一个是游客 guest,那么那么相应的权限就不一样,而且这个权限是贯穿整个系统的,因此建立一个 proxy,类似网关的概念,可以很好的达到这个效果.

看看 Java 宠物店中的 CatalogDAOFactory:

```
public class CatalogDAOFactory {
```

```
   /**
   * 本方法制定一个特别的子类来实现 DAO 模式。
   * 具体子类定义是在 J2EE 的部署描述器中。
   */
   public static CatalogDAO getDAO() throws CatalogDAOSysException
{
       CatalogDAO catDao = null;
       try {
           InitialContext ic = new InitialContext();
           //动态装入 CATALOG_DAO_CLASS
           //可以定义自己的 CATALOG_DAO_CLASS，从而在无需变更太多代码
           //的前提下，完成系统的巨大变更。
           String className =(String)
ic.lookup(JNDINames.CATALOG_DAO_CLASS);
           catDao = (CatalogDAO)
Class.forName(className).newInstance();
       } catch (NamingException ne) {
           throw new CatalogDAOSysException("
              CatalogDAOFactory.getDAO: NamingException while
                  getting DAO type : \n" + ne.getMessage());
       } catch (Exception se) {
           throw new CatalogDAOSysException("
```

```
            CatalogDAOFactory.getDAO: Exception while getting
                DAO type : \n" + se.getMessage());

        }

        return catDao;

    }

}
```

CatalogDAOFactory 是典型的工厂方法，catDao 是通过动态类装入器 className 获得 CatalogDAOFactory 具体实现子类，这个实现子类在 Java 宠物店是用来操作 catalog 数据库，用户可以根据数据库的类型不同，定制自己的具体实现子类，将自己的子类名给与 CATALOG_DAO_CLASS 变量就可以。

由此可见，工厂方法确实为系统结构提供了非常灵活强大的动态扩展机制，只要我们更换一下具体的工厂方法，系统其他地方无需一点变换，就有可能将系统功能进行改头换面的变化。

设计模式之 Singleton(单态)

单态定义:

Singleton 模式主要作用是保证在 Java 应用程序中，一个类 Class 只有一个实例存在。

在很多操作中，比如建立目录 数据库连接都需要这样的单线程操作。

还有，singleton 能够被状态化；这样，多个单态类在一起就可以作为一个状态仓库一样向外提供服务，比如，你要论坛中的帖子计数器，每次浏览一次需要计数，单态类能否保持住这个计数，并且能 synchronize 的安全自动加 1，如果你要把这个数字永久保存到数据库，你可以在不修改单态接口的情况下方便的做到。

另外方面，Singleton 也能够被无状态化。提供工具性质的功能，

Singleton 模式就为我们提供了这样实现的可能。使用 Singleton 的好处还在于可以节省内存，因为它限制了实例的个数，有利于 Java 垃圾回收（garbage collection）。

我们常常看到工厂模式中类装入器(class loader)中也用 Singleton 模式实现的,因为被装入的类实际也属于资源。

如何使用?

一般 Singleton 模式通常有几种形式:

```
public class Singleton {

   private Singleton(){}

   //在自己内部定义自己一个实例，是不是很奇怪？
   //注意这是 private 只供内部调用

   private static Singleton instance = new Singleton();

   //这里提供了一个供外部访问本 class 的静态方法，可以直接访问
```

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

第二种形式:

```
    public class Singleton {

   private static Singleton instance = null;

   public static synchronized Singleton getInstance() {

   //这个方法比上面有所改进，不用每次都进行生成对象，只是第一次
   //使用时生成实例，提高了效率！
   if (instance==null)
      instance＝new Singleton();
   return instance;    }

}
```

使用 Singleton.getInstance()可以访问单态类。

上面第二中形式是 lazy initialization，也就是说第一次调用时初始 Singleton，以后就不用再生成了。

注意到 lazy initialization 形式中的 synchronized，这个 synchronized 很重要，如果没有 synchronized，那么使用 getInstance()是有可能得到多个 Singleton

实例。关于 lazy initialization 的 Singleton 有很多涉及 double-checked locking (DCL)的讨论，有兴趣者进一步研究。

一般认为第一种形式要更加安全些。

使用 Singleton 注意事项：

有时在某些情况下，使用 Singleton 并不能达到 Singleton 的目的，如有多个 Singleton 对象同时被不同的类装入器装载 ;在 EJB 这样的分布式系统中使用也要注意这种情况，因为 EJB 是跨服务器，跨 JVM 的。

我们以 SUN 公司的宠物店源码(Pet Store 1.3.1)的 ServiceLocator 为例稍微分析一下：

在 Pet Store 中 ServiceLocator 有两种，一个是 EJB 目录下；一个是 WEB 目录下，我们检查这两个 ServiceLocator 会发现内容差不多，都是提供 EJB 的查询定位服务，可是为什么要分开呢？仔细研究对这两种 ServiceLocator 才发现区别：在 WEB 中的 ServiceLocator 的采取 Singleton 模式，ServiceLocator 属于资源定位，理所当然应该使用 Singleton 模式。但是在 EJB 中，Singleton 模式已经失去作用，所以 ServiceLocator 才分成两种，一种面向 WEB 服务的，一种是面向 EJB 服务的。

Singleton 模式看起来简单，使用方法也很方便，但是真正用好，是非常不容易，需要对 Java 的类 线程 内存等概念有相当的了解。

总之：如果你的应用基于容器，那么 Singleton 模式少用或者不用，可以使用相关替代技术。

设计模式之 Builder

Builder 模式定义:

将一个复杂对象的构建与它的表示分离,使得同样的构建过程可以创建不同的表示.

Builder 模式是一步一步创建一个复杂的对象,它允许用户可以只通过指定复杂对象的类型和内容就可以构建它们.用户不知道内部的具体构建细节.Builder 模式是非常类似抽象工厂模式,细微的区别大概只有在反复使用中才能体会到.

为何使用?

是为了将构建复杂对象的***过程***和它的***部件***解耦.注意: 是解耦***过程***和***部件***.

因为一个复杂的对象,不但有很多大量组成部分,如汽车,有很多部件:车轮 方向盘 发动机还有各种小零件等等,部件很多,但远不止这些,如何将这些部件装配成一辆汽车,这个装配过程也很复杂(需要很好的组装技术),Builder 模式就是为了将部件和组装过程分开.

如何使用?

首先假设一个复杂对象是由多个部件组成的,Builder 模式是把复杂对象的创建和部件的创建分别开来,分别用 Builder 类和 Director 类来表示.

首先,需要一个接口,它定义如何创建复杂对象的各个部件:

```
public interface Builder {

   //创建部件 A    比如创建汽车车轮
   void buildPartA();
   //创建部件 B 比如创建汽车方向盘
   void buildPartB();
   //创建部件 C 比如创建汽车发动机
   void buildPartC();


   //返回最后组装成品结果 (返回最后装配好的汽车)
```

```
    //成品的组装过程不在这里进行,而是转移到下面的 Director 类
中进行.
    //从而实现了解耦过程和部件
    Product getResult();

}
```

用 Director 构建最后的复杂对象,而在上面 Builder 接口中封装的是如何创建一个个部件(复杂对象是由这些部件组成的),也就是说 Director 的内容是如何将部件最后组装成成品:

```
public class Director {

    private Builder builder;

    public Director( Builder builder ) {
        this.builder = builder;
    }
    // 将部件 partA partB partC 最后组成复杂对象
    //这里是将车轮 方向盘和发动机组装成汽车的过程
    public void construct() {
        builder.buildPartA();
        builder.buildPartB();
        builder.buildPartC();

    }

}
```

Builder 的具体实现 ConcreteBuilder:

通过具体完成接口 Builder 来构建或装配产品的部件;

定义并明确它所要创建的是什么具体东西;

提供一个可以重新获取产品的接口:

```
public class ConcreteBuilder implements Builder {

   Part partA, partB, partC;
   public void buildPartA() {
      //这里是具体如何构建 partA 的代码


   };
   public void buildPartB() {
      //这里是具体如何构建 partB 的代码
   };
    public void buildPartC() {
      //这里是具体如何构建 partB 的代码
   };
    public Product getResult() {
      //返回最后组装成品结果
   };

}
```

复杂对象:产品 Product:

```
public interface Product { }
```

复杂对象的部件:

```
public interface Part { }
```

我们看看如何调用 Builder 模式:

```
ConcreteBuilder builder = new ConcreteBuilder();
Director director = new Director( builder );

director.construct();
Product product = builder.getResult();
```

Builder 模式的应用

在 Java 实际使用中,我们经常用到"池"(Pool)的概念,当资源提供者无法提供足够的资源,并且这些资源需要被很多用户反复共享时,就需要使用池.

"池"实际是一段内存,当池中有一些复杂的资源的"断肢"(比如数据库的连接池,也许有时一个连接会中断),如果循环再利用这些"断肢",将提高内存使用效率,提高池的性能.修改 Builder 模式中 Director 类使之能诊断"断肢"断在哪个部件上,再修复这个部件.

设计模式之 Prototype

原型模式定义:

用原型实例指定创建对象的种类,并且通过拷贝这些原型创建新的对象.

Prototype 模式允许一个对象再创建另外一个可定制的对象，根本无需知道任何如何创建的细节,工作原理是:通过将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝它们自己来实施创建。

如何使用?

因为 Java 中的提供 clone()方法来实现对象的克隆,所以 Prototype 模式实现一下子变得很简单.

以勺子为例：

```
public abstract class AbstractSpoon implements Cloneable
{
    String spoonName;

    public void setSpoonName(String spoonName) {this.spoonName =
spoonName;}
    public String getSpoonName() {return this.spoonName;}

    public Object clone()
    {
        Object object = null;
        try {
            object = super.clone();
        } catch (CloneNotSupportedException exception) {
            System.err.println("AbstractSpoon is not Cloneable");
        }
```

```
        return object;
    }
}
```

有个具体实现(ConcretePrototype):

```
public class SoupSpoon extends AbstractSpoon
{
    public SoupSpoon()
    {
        setSpoonName("Soup Spoon");
    }
}
```

调用 Prototype 模式很简单:

```
AbstractSpoon spoon = new SoupSpoon();
AbstractSpoon spoon2 = spoon.clone();
```

当然也可以结合工厂模式来创建 AbstractSpoon 实例。

在 Java 中 Prototype 模式变成 clone()方法的使用，由于 Java 的纯洁的面向对象特性，使得在 Java 中使用设计模式变得很自然，两者已经几乎是浑然一体了。这反映在很多模式上，如 Interator 遍历模式。

结构模式

设计模式之 Flyweight

Flyweight 定义:

避免大量拥有相同内容的小类的开销(如耗费内存),使大家共享一个类(元类).

为什么使用?

面向对象语言的原则就是一切都是对象,但是如果真正使用起来,有时对象数可能显得很庞大,比如,字处理软件,如果以每个文字都作为一个对象,几千个字,对象数就是几千,无疑耗费内存,那么我们还是要"求同存异",找出这些对象群的共同点,设计一个元类,封装可以被共享的类,另外,还有一些特性是取决于应用(context),是不可共享的,这也 Flyweight 中两个重要概念内部状态 intrinsic 和外部状态 extrinsic 之分.

说白点,就是先捏一个的原始模型,然后随着不同场合和环境,再产生各具特征的具体模型,很显然,在这里需要产生不同的新对象,所以 Flyweight 模式中常出现 Factory 模式.Flyweight 的内部状态是用来共享的,Flyweight factory 负责维护一个 Flyweight pool(模式池)来存放内部状态的对象.

Flyweight模式是一个提高程序效率和性能的模式,会大大加快程序的运行速度.应用场合很多:比如你要从一个数据库中读取一系列字符串,这些字符串中有许多是重复的,那么我们可以将这些字符串储存在Flyweight池(pool)中.

如何使用?

我们先从 Flyweight 抽象接口开始:

```
public interface Flyweight

{

    public void operation( ExtrinsicState state );
```

```
}

//用于本模式的抽象数据类型(自行设计)

public interface ExtrinsicState { }
```

下面是接口的具体实现(ConcreteFlyweight) ,并为内部状态增加内存空间, ConcreteFlyweight 必须是可共享的,它保存的任何状态都必须是内部(intrinsic), 也就是说,ConcreteFlyweight 必须和它的应用环境场合无关.

```
public class ConcreteFlyweight implements Flyweight {

   private IntrinsicState state;

   public void operation( ExtrinsicState state )

   {

          //具体操作

   }

}
```

当然,并不是所有的 Flyweight 具体实现子类都需要被共享的,所以还有另外一种不共享的 ConcreteFlyweight:

```
public class UnsharedConcreteFlyweight implements Flyweight {

   public void operation( ExtrinsicState state ) { }

}
```

Flyweight factory 负责维护一个 Flyweight 池(存放内部状态),当客户端请求一个共享 Flyweight 时,这个 factory 首先搜索池中是否已经有可适用的,如果

有,factory 只是简单返回送出这个对象,否则,创建一个新的对象,加入到池中,再返回送出这个对象.池

```
public class FlyweightFactory {

    //Flyweight pool

    private Hashtable flyweights = new Hashtable();

    public Flyweight getFlyweight( Object key ) {

   Flyweight flyweight = (Flyweight) flyweights.get(key);

        if( flyweight == null ) {

            //产生新的 ConcreteFlyweight

            flyweight = new ConcreteFlyweight();

            flyweights.put( key, flyweight );

        }

        return flyweight;

    }

}
```

至此,Flyweight 模式的基本框架已经就绪,我们看看如何调用:

```
FlyweightFactory factory = new FlyweightFactory();

Flyweight fly1 = factory.getFlyweight( "Fred" );

Flyweight fly2 = factory.getFlyweight( "Wilma" );
```

.....

从调用上看,好象是个纯粹的 Factory 使用,但奥妙就在于 Factory 的内部设计上.

Flyweight模式在XML等数据源中应用

我们上面已经提到,当大量从数据源中读取字符串,其中肯定有重复的,那么我们使用 Flyweight 模式可以提高效率,以唱片 CD 为例,在一个 XML 文件中,存放了多个 CD 的资料.

每个 CD 有三个字段:

1.出片日期(year)

2.歌唱者姓名等信息(artist)

3.唱片曲目 (title)

其中,歌唱者姓名有可能重复,也就是说,可能有同一个演唱者的多个不同时期 不同曲目的 CD.我们将"歌唱者姓名"作为可共享的 ConcreteFlyweight.其他两个字段作为 UnsharedConcreteFlyweight.

首先看看数据源 XML 文件的内容:

```
<?xml version="1.0"?>

<collection>

<cd>

<title>Another Green World</title>

<year>1978</year>

<artist>Eno, Brian</artist>
```

```
</cd>

<cd>

<title>Greatest Hits</title>

<year>1950</year>

<artist>Holiday, Billie</artist>

</cd>

<cd>

<title>Taking Tiger Mountain (by strategy)</title>

<year>1977</year>

<artist>Eno, Brian</artist>

</cd>

.......

</collection>
```

虽然上面举例 CD 只有 3 张,CD 可看成是大量重复的小类,因为其中成分只有三个字段,而且有重复的(歌唱者姓名).

CD 就是类似上面接口 Flyweight:

```
public class CD {

   private String title;

   private int year;
```

```
    private Artist artist;

    public String getTitle() {    return title;  }

    public int getYear() {       return year;    }

    public Artist getArtist() {       return artist;    }

    public void setTitle(String t){       title = t;}

    public void setYear(int y){year = y;}

    public void setArtist(Artist a){artist = a;}

}
```

将"歌唱者姓名"作为可共享的 ConcreteFlyweight:

```
public class Artist {

    //内部状态

    private String name;

    // note that Artist is immutable.

    String getName(){return name;}

Artist(String n){

        name = n;

    }

}
```

再看看 Flyweight factory,专门用来制造上面的可共享的 ConcreteFlyweight:Artist

```
public class ArtistFactory {

Hashtable pool = new Hashtable();
Artist getArtist(String key){

        Artist result;

        result = (Artist)pool.get(key);

        ////产生新的 Artist

        if(result == null) {

            result = new Artist(key);

            pool.put(key,result);

        }

       return result;

   }

}
```

当你有几千张甚至更多 CD 时,Flyweight 模式将节省更多空间,共享的 flyweight 越多,空间节省也就越大.

设计模式之 Bridge

Bridge 模式定义 ：

将抽象和行为划分开来,各自独立,但能动态的结合。

任何事物对象都有抽象和行为之分，例如人，人是一种抽象，人分男人和女人等；人有行为，行为也有各种具体表现，所以，“人”与“人的行为”两个概念也反映了抽象和行为之分。

在面向对象设计的基本概念中，对象这个概念实际是由属性和行为两个部分组成的，属性我们可以认为是一种静止的，是一种抽象，一般情况下，行为是包含在一个对象中，但是，在有的情况下，我们需要将这些行为也进行归类，形成一个总的行为接口，这就是桥模式的用处。

为什么使用?

不希望抽象部分和行为有一种固定的绑定关系，而是应该可以动态联系的。

如果一个抽象类或接口有多个具体实现(子类、concrete subclass),这些子类之间关系可能有以下两种情况:

1. 这多个子类之间概念是并列的,如前面举例,打桩,有两个 concrete class:方形桩和圆形桩;这两个形状上的桩是并列的,没有概念上的重复。

2.这多个子类之中有内容概念上重叠.那么需要我们把抽象共同部分和行为共同部分各自独立开来,原来是准备放在一个接口里,现在需要设计两个接口：抽象接口和行为接口，分别放置抽象和行为.

例如,一杯咖啡为例,子类实现类为四个：中杯加奶、大杯加奶、 中杯不加奶、大杯不加奶。

但是，我们注意到：上面四个子类中有概念重叠，可从另外一个角度进行考虑，这四个类实际是两个角色的组合：抽象 和行为，其中抽象为：中杯和大杯；行为为：加奶 不加奶（如加橙汁 加苹果汁）.

实现四个子类在抽象和行为之间发生了固定的绑定关系，如果以后动态增加加葡萄汁的行为，就必须再增加两个类：中杯加葡萄汁和大杯加葡萄汁。显然混乱,扩展性极差。

那我们从分离抽象和行为的角度，使用 Bridge 模式来实现。

如何实现?

以上面提到的咖啡 为例. 我们原来打算只设计一个接口(抽象类),使用 Bridge 模式后,我们需要将抽象和行为分开,加奶和不加奶属于行为,我们将它们抽象成一个专门的行为接口.

先看看抽象部分的接口代码:

```
public abstract class Coffee
{
   CoffeeImp coffeeImp;

   public void setCoffeeImp() {
      this.CoffeeImp =
CoffeeImpSingleton.getTheCoffeImp();
   }

   public CoffeeImp getCoffeeImp() {return
this.CoffeeImp;}

   public abstract void pourCoffee();
}
```

其中 CoffeeImp 是加不加奶的行为接口,看其代码如下:

```
public abstract class CoffeeImp
{
```

```
    public abstract void pourCoffeeImp();
}
```

现在我们有了两个抽象类,下面我们分别对其进行继承,实现 concrete class:

```
//中杯
public class MediumCoffee extends Coffee
{
    public MediumCoffee() {setCoffeeImp();}

    public void pourCoffee()
    {
        CoffeeImp coffeeImp = this.getCoffeeImp();
        //我们以重复次数来说明是冲中杯还是大杯 ,重复 2 次是中杯
        for (int i = 0; i < 2; i++)
        {

            coffeeImp.pourCoffeeImp();
        }

    }
}

//大杯
public class SuperSizeCoffee extends Coffee
{
    public SuperSizeCoffee() {setCoffeeImp();}

    public void pourCoffee()
    {
```

```
        CoffeeImp coffeeImp = this.getCoffeeImp();
        //我们以重复次数来说明是冲中杯还是大杯 ,重复 5 次是大杯
        for (int i = 0; i < 5; i++)
        {

            coffeeImp.pourCoffeeImp();
        }

    }
}
```

上面分别是中杯和大杯的具体实现.下面再对行为 CoffeeImp 进行继承:

```
//加奶
public class MilkCoffeeImp extends CoffeeImp
{
    MilkCoffeeImp() {}

    public void pourCoffeeImp()
    {
        System.out.println("加了美味的牛奶");
    }
}

//不加奶
public class FragrantCoffeeImp extends CoffeeImp
{
    FragrantCoffeeImp() {}

    public void pourCoffeeImp()
```

```
    {
        System.out.println("什么也没加,清香");
    }
}
```

Bridge模式的基本框架我们已经搭好了,别忘记定义中还有一句:动态结合,我们现在可以喝到至少四种咖啡:

1.中杯加奶

2.中杯不加奶

3.大杯加奶

4.大杯不加奶

看看是如何动态结合的,在使用之前,我们做个准备工作,设计一个单态类(Singleton)用来 hold 当前的 CoffeeImp:

```
public class CoffeeImpSingleton
{
    private static CoffeeImp coffeeImp;

    public CoffeeImpSingleton(CoffeeImp coffeeImpIn)
     {this.coffeeImp = coffeeImpIn;}

    public static CoffeeImp getTheCoffeeImp()
    {
        return coffeeImp;
    }
}
```

看看中杯加奶 和大杯加奶 是怎么出来的:

```
//拿出牛奶
CoffeeImpSingleton coffeeImpSingleton = new CoffeeImpSingleton(new
MilkCoffeeImp());

//中杯加奶
MediumCoffee mediumCoffee = new MediumCoffee();
mediumCoffee.pourCoffee();

//大杯加奶
SuperSizeCoffee superSizeCoffee = new SuperSizeCoffee();
superSizeCoffee.pourCoffee();
```

注意: Bridge 模式的执行类如 CoffeeImp 和 Coffee 是一对一的关系, 正确创建 CoffeeImp 是该模式的关键,

Bridge 模式在 EJB 中的应用

EJB 中有一个 Data Access Object (DAO)模式,这是将商业逻辑和具体数据资源分开的,因为不同的数据库有不同的数据库操作.将操作不同数据库的行为独立抽象成一个行为接口 DAO.如下:

1.Business Object (类似 Coffee)

实现一些抽象的商业操作:如寻找一个用户下所有的订单

涉及数据库操作都使用 DAOImplementor.

2.Data Access Object (类似 CoffeeImp)

一些抽象的对数据库资源操作

3.DAOImplementor 如 OrderDAOCS, OrderDAOOracle, OrderDAOSybase(类似 MilkCoffeeImp FragrantCoffeeImp)

具体的数据库操作,如"INSERT INTO "等语句,OrderDAOOracle 是 Oracle OrderDAOSybase 是 Sybase 数据库.

4.数据库 (Cloudscape, Oracle, or Sybase database via JDBC API)

设计模式之 Decorator(油漆工)

装饰模式:Decorator 常被翻译成"装饰",我觉得翻译成"油漆工"更形象点,油漆工(decorator)是用来刷油漆的,那么被刷油漆的对象我们称 decoratee.这两种实体在 Decorator 模式中是必须的.

Decorator 定义:

动态给一个对象添加一些额外的职责,就象在墙上刷油漆.使用 Decorator 模式相比用生成子类方式达到功能的扩充显得更为灵活.

为什么使用 Decorator?

我们通常可以使用继承来实现功能的拓展,如果这些需要拓展的功能的种类很繁多,那么势必生成很多子类,增加系统的复杂性,同时,使用继承实现功能拓展,我们必须可预见这些拓展功能,这些功能是编译时就确定了,是静态的.

使用 Decorator 的理由是:这些功能需要由用户动态决定加入的方式和时机.Decorator 提供了"即插即用"的方法,在运行期间决定何时增加何种功能.

如何使用?

举 Adapter 中的打桩示例,在 Adapter 中有两种类:方形桩 圆形桩,Adapter 模式展示如何综合使用这两个类,在 Decorator 模式中,我们是要在打桩时增加一些额外功能,比如,挖坑 在桩上钉木板等,不关心如何使用两个不相关的类.

我们先建立一个接口:

```
public interface Work
{
    public void insert();

}
```

接口 Work 有一个具体实现:插入方形桩或圆形桩,这两个区别对 Decorator 是无所谓.我们以插入方形桩为例:

```
    public class SquarePeg implements Work{
   public void insert(){
       System.out.println("方形桩插入");
   }

}
```

现在有一个应用:需要在桩打入前,挖坑,在打入后,在桩上钉木板,这些额外的功能是动态,可能随意增加调整修改,比如,可能又需要在打桩之后钉架子(只是比喻).

那么我们使用 Decorator 模式,这里方形桩 SquarePeg 是 decoratee(被刷油漆者),我们需要在 decoratee 上刷些"油漆",这些油漆就是那些额外的功能.

```
public class Decorator implements Work{

   private Work work;
   //额外增加的功能被打包在这个 List 中
   private ArrayList others = new ArrayList();

   //在构造器中使用组合 new 方式,引入 Work 对象;
   public Decorator(Work work)
   {
       this.work=work;


       others.add("挖坑");

       others.add("钉木板");
   }
```

```
    public void insert(){

        newMethod();
    }



    //在新方法中,我们在 insert 之前增加其他方法,这里次序先后是
用户灵活指定的
    public void newMethod()
    {
        otherMethod();
        work.insert();



    }

    public void otherMethod()
    {
        ListIterator listIterator =
others.listIterator();
        while (listIterator.hasNext())
        {

System.out.println(((String)(listIterator.next())) +
" 正在进行");
        }

    }

}
```

在上例中,我们把挖坑和钉木板都排在了打桩 insert 前面,这里只是举例说明额外功能次序可以任意安排.

好了,Decorator 模式出来了,我们看如何调用:

```
Work squarePeg = new SquarePeg();
Work decorator = new Decorator(squarePeg);
decorator.insert();
```

Decorator 模式至此完成.

如果你细心,会发现,上面调用类似我们读取文件时的调用:

```
FileReader fr = new FileReader(filename);
BufferedReader br = new BufferedReader(fr);
```

实际上Java 的I/O API就是使用Decorator实现的,I/O变种很多,如果都采取继承方法,将会产生很多子类,显然相当繁琐.

Jive 中的 Decorator 实现

在论坛系统中,有些特别的字是不能出现在论坛中如"打倒 XXX",我们需要过滤这些"反动"的字体.不让他们出现或者高亮度显示.

在IBM Java专栏中专门谈Jive的文章中,有谈及Jive中ForumMessageFilter.java使用了Decorator模式,其实,该程序并没有真正使用Decorator,而是提示说:针对特别论坛可以设计额外增加的过滤功能,那么就可以重组ForumMessageFilter作为Decorator模式了.

所以,我们在分辨是否真正是 Decorator 模式,以及会真正使用 Decorator 模式,一定要把握好 Decorator 模式的定义,以及其中参与的角色(Decoratee 和 Decorator).

设计模式之 Composite(组合)

Composite 模式定义:

将对象以树形结构组织起来,以达成“部分－整体” 的层次结构，使得客户端对单个对象和组合对象的使用具有一致性.

Composite 比较容易理解，想到 Composite 就应该想到树形结构图。组合体内这些对象都有共同接口,当组合体一个对象的方法被调用执行时，Composite 将遍历(Iterator)整个树形结构,寻找同样包含这个方法的对象并实现调用执行。可以用牵一动百来形容。

所以 Composite 模式使用到 Iterator 模式，和 Chain of Responsibility 模式类似。

Composite 好处:

1.使客户端调用简单，客户端可以一致的使用组合结构或其中单个对象，用户就不必关系自己处理的是单个对象还是整个组合结构，这就简化了客户端代码。

2.更容易在组合体内加入对象部件. 客户端不必因为加入了新的对象部件而更改代码。

如何使用 Composite?

首先定义一个接口或抽象类，这是设计模式通用方式了，其他设计模式对接口内部定义限制不多,Composite 却有个规定,那就是要在接口内部定义一个用于访问和管理 Composite 组合体的对象们（或称部件 Component）.

下面的代码是以抽象类定义，一般尽量用接口 interface,

```
public abstract class Equipment
{
    private String name;
    //实价
    public abstract double netPrice();
    //折扣价格
```

```
    public abstract double discountPrice();
    //增加部件方法
    public boolean add(Equipment equipment) { return
false; }
    //删除部件方法
    public boolean remove(Equipment equipment) { return
false; }
    //注意这里，这里就提供一种用于访问组合体类的部件方法。
    public Iterator iter() { return null; }

    public Equipment(final String name)
{ this.name=name; }
}
```

抽象类 Equipment 就是 Component 定义，代表着组合体类的对象们,Equipment 中定义几个共同的方法。

```
public class Disk extends Equipment
{
    public Disk(String name) { super(name); }
    //定义 Disk 实价为 1
    public double netPrice() { return 1.; }
    //定义了 disk 折扣价格是 0.5 对折。
    public double discountPrice() { return .5; }
}
```

Disk 是组合体内的一个对象，或称一个部件，这个部件是个单独元素(Primitive)。还有一种可能是，一个部件也是一个组合体，就是说这个部件下面还有'儿子'，这是树形结构中通常的情况，应该比较容易理解。现在我们先要定义这个组合体：

```
abstract class CompositeEquipment extends Equipment
{
    private int i=0;
    //定义一个 Vector 用来存放'儿子'
    private Lsit equipment=new ArrayList();

    public CompositeEquipment(String name) { super(name); }

    public boolean add(Equipment equipment) {
        this.equipment.add(equipment);
        return true;
     }

    public double netPrice()
    {
       double netPrice=0.;
       Iterator iter=equipment.iterator();
       for(iter.hasNext())
          netPrice+=((Equipment)iter.next()).netPrice();
       return netPrice;
    }

    public double discountPrice()
    {
       double discountPrice=0.;
       Iterator iter=equipment.iterator();
       for(iter.hasNext())

discountPrice+=((Equipment)iter.next()).discountPrice();
```

```
        return discountPrice;
    }


    //注意这里，这里就提供用于访问自己组合体内的部件方法。
    //上面dIsk 之所以没有,是因为Disk是个单独(Primitive)的元素.
    public Iterator iter()
    {
        return equipment.iterator() ;
    {
    //重载Iterator方法
     public boolean hasNext() { return i<equipment.size(); }
    //重载Iterator方法
     public Object next()
     {
        if(hasNext())
            return equipment.elementAt(i++);
        else
            throw new NoSuchElementException();
     }


}
```

上面 CompositeEquipment 继承了 Equipment,同时为自己里面的对象们提供了外部访问的方法,重载了 Iterator,Iterator 是 Java 的 Collection 的一个接口，是 Iterator 模式的实现.

我们再看看 CompositeEquipment 的两个具体类:盘盒 Chassis 和箱子 Cabinet，箱子里面可以放很多东西，如底板，电源盒，硬盘盒等；盘盒里面可以放一些小设备，如硬盘软驱等。无疑这两个都是属于组合体性质的。

```
    public class Chassis extends CompositeEquipment
{
    public Chassis(String name) { super(name); }
    public double netPrice() { return
1.+super.netPrice(); }
    public double discountPrice()
{ return .5+super.discountPrice(); }
}

public class Cabinet extends CompositeEquipment
{
    public Cabinet(String name) { super(name); }
    public double netPrice() { return
1.+super.netPrice(); }
    public double discountPrice()
{ return .5+super.discountPrice(); }
}
```

至此我们完成了整个 Composite 模式的架构。

我们可以看看客户端调用 Composote 代码:

```
Cabinet cabinet=new Cabinet("Tower");

Chassis chassis=new Chassis("PC Chassis");
//将 PC Chassis 装到 Tower 中 (将盘盒装到箱子里)
cabinet.add(chassis);
```

```
//将一个 10GB 的硬盘装到 PC Chassis (将硬盘装到盘盒里)
chassis.add(new Disk("10 GB"));

//调用 netPrice()方法;
System.out.println("netPrice="+cabinet.netPrice());
System.out.println("discountPrice="+cabinet.discountPrice());
```

上面调用的方法 netPrice()或 discountPrice(),实际上 Composite 使用 Iterator 遍历了整个树形结构,寻找同样包含这个方法的对象并实现调用执行.

Composite 是个很巧妙体现智慧的模式，在实际应用中，如果碰到树形结构，我们就可以尝试是否可以使用这个模式。

以论坛为例，一个版(forum)中有很多帖子(message),这些帖子有原始贴，有对原始贴的回应贴,是个典型的树形结构,那么当然可以使用 Composite 模式,那么我们进入 Jive 中看看，是如何实现的.

Jive 解剖

在 Jive 中 ForumThread 是 ForumMessages 的容器 container(组合体).也就是说，ForumThread 类似我们上例中的 CompositeEquipment.它和 messages 的关系如图：

```
[thread]
    |- [message]
    |- [message]
        |- [message]
        |- [message]
            |- [message]
```

我们在 ForumThread 看到如下代码：

```
public interface ForumThread {
....
```

```
    public void addMessage(ForumMessage parentMessage,
ForumMessage newMessage)
            throws UnauthorizedException;

    public void deleteMessage(ForumMessage message)
            throws UnauthorizedException;


    public Iterator messages();
        ....

}
```

类似 CompositeEquipment，提供用于访问自己组合体内的部件方法：增加 删除 遍历.

结合我的其他模式中对 Jive 的分析，我们已经基本大体理解了 Jive 论坛体系的框架，如果你之前不理解设计模式，而直接去看 Jive 源代码，你肯定无法看懂。

:)

设计模式之Adapter(适配器)

适配器模式定义:

将两个不兼容的类纠合在一起使用，属于结构型模式,需要有 Adaptee(被适配者)和 Adaptor(适配器)两个身份.

为何使用?

我们经常碰到要将两个没有关系的类组合在一起使用,第一解决方案是：修改各自类的接口，但是如果我们没有源代码，或者，我们不愿意为了一个应用而修改各自的接口。 怎么办?

使用 Adapter，在这两种接口之间创建一个混合接口(混血儿).

如何使用?

实现 Adapter 方式,其实"think in Java"的"类再生"一节中已经提到,有两种方式：组合(composition)和继承(inheritance).

假设我们要打桩，有两种类：方形桩 圆形桩.

```
public class SquarePeg{
    public void insert(String str){
        System.out.println("SquarePeg insert():"+str);
    }

}

public class RoundPeg{
    public void insertIntohole(String msg){
        System.out.println("RoundPeg insertIntoHole():"+msg);
}
}
```

现在有一个应用,需要既打方形桩,又打圆形桩.那么我们需要将这两个没有关系的类综合应用.假设 RoundPeg 我们没有源代码,或源代码我们不想修改,那么我们使用 Adapter 来实现这个应用:

```
public class PegAdapter extends SquarePeg{

    private RoundPeg roundPeg;
    public PegAdapter(RoundPeg peg)(this.roundPeg=peg;)
    public void insert(String str){ roundPeg.insertIntoHole(str);}

}
```

在上面代码中,RoundPeg 属于 Adaptee,是被适配者.PegAdapter 是 Adapter,将 Adaptee(被适配者 RoundPeg)和 Target(目标 SquarePeg)进行适配.实际上这是将组合方法(composition)和继承(inheritance)方法综合运用.

PegAdapter 首先继承 SquarePeg,然后使用 new 的组合生成对象方式,生成 RoundPeg 的对象 roundPeg,再重载父类 insert()方法。从这里,你也了解使用 new 生成对象和使用 extends 继承生成对象的不同,前者无需对原来的类修改,甚至无需要知道其内部结构和源代码.

如果你有些 Java 使用的经验，已经发现，这种模式经常使用。

进一步使用

上面的 PegAdapter 是继承了 SquarePeg,如果我们需要两边继承，即继承 SquarePeg 又继承 RoundPeg,因为 Java 中不允许多继承，但是我们可以实现(implements)两个接口(interface)

```
public interface IRoundPeg{
    public void insertIntoHole(String msg);

}
```

```
public interface ISquarePeg{
    public void insert(String str);
}
```

下面是新的 RoundPeg 和 SquarePeg，除了实现接口这一区别，和上面的没什么区别。

```
public class SquarePeg implements ISquarePeg{
    public void insert(String str){
        System.out.println("SquarePeg insert():"+str);
    }
}

public class RoundPeg implements IRoundPeg{
    public void insertIntohole(String msg){
        System.out.println("RoundPeg insertIntoHole():"+msg);
    }
}
```

下面是新的 PegAdapter,叫做 two-way adapter:

```
public class PegAdapter implements IRoundPeg,ISquarePeg{

    private RoundPeg roundPeg;
    private SquarePeg squarePeg;

    // 构造方法
    public PegAdapter(RoundPeg peg){this.roundPeg=peg;}
    // 构造方法
    public PegAdapter(SquarePeg peg)(this.squarePeg=peg;)

    public void insert(String str){ roundPeg.insertIntoHole(str);}
}
```

还有一种叫 Pluggable Adapters,可以动态的获取几个 adapters 中一个。使用 Reflection 技术，可以动态的发现类中的 Public 方法。

设计模式之 Proxy(代理)

理解并使用设计模式,能够培养我们良好的面向对象编程习惯,同时在实际应用中,可以如鱼得水,享受游刃有余的乐趣.

代理模式是比较有用途的一种模式,而且变种较多,应用场合覆盖从小结构到整个系统的大结构,Proxy 是代理的意思,我们也许有代理服务器等概念,代理概念可以解释为:在出发点到目的地之间有一道中间层,意为代理.

设计模式中定义: 为其他对象提供一种代理以控制对这个对象的访问.

为什么要使用 Proxy?

1.授权机制 不同级别的用户对同一对象拥有不同的访问权利,如 Jive 论坛系统中,就使用 Proxy 进行授权机制控制,访问论坛有两种人:注册用户和游客(未注册用户),Jive 中就通过类似 ForumProxy 这样的代理来控制这两种用户对论坛的访问权限.

2.某个客户端不能直接操作到某个对象,但又必须和那个对象有所互动.
举例两个具体情况:
(1)如果那个对象是一个是很大的图片,需要花费很长时间才能显示出来,那么当这个图片包含在文档中时,使用编辑器或浏览器打开这个文档,打开文档必须很迅速,不能等待大图片处理完成,这时需要做个图片 Proxy 来代替真正的图片.

(2)如果那个对象在 Internet 的某个远端服务器上,直接操作这个对象因为网络速度原因可能比较慢,那我们可以先用 Proxy 来代替那个对象.

总之原则是,对于开销很大的对象,只有在使用它时才创建,这个原则可以为我们节省很多宝贵的 Java 内存. 所以,有些人认为 Java 耗费资源内存,我以为这和程序编制思路也有一定的关系.

如何使用Proxy?

以Jive论坛系统为例,访问论坛系统的用户有多种类型:注册普通用户 论坛管理者 系统

管理者 游客,注册普通用户才能发言;论坛管理者可以管理他被授权的论坛;系统管理者可以管理所有事务等,这些权限划分和管理是使用Proxy完成的.

Forum 是 Jive 的核心接口,在 Forum 中陈列了有关论坛操作的主要行为,如论坛名称 论坛描述的获取和修改,帖子发表删除编辑等.

在 ForumPermissions 中定义了各种级别权限的用户:

```
public class ForumPermissions implements Cacheable {

/**
* Permission to read object.
*/
public static final int READ = 0;

/**
* Permission to administer the entire sytem.
*/
public static final int SYSTEM_ADMIN = 1;

/**
* Permission to administer a particular forum.
*/
public static final int FORUM_ADMIN = 2;

/**
* Permission to administer a particular user.
*/
public static final int USER_ADMIN = 3;

/**
* Permission to administer a particular group.
```

```
*/
public static final int GROUP_ADMIN = 4;

/**
* Permission to moderate threads.
*/
public static final int MODERATE_THREADS = 5;

/**
* Permission to create a new thread.
*/
public static final int CREATE_THREAD = 6;

/**
* Permission to create a new message.
*/
public static final int CREATE_MESSAGE = 7;

/**
* Permission to moderate messages.
*/
public static final int MODERATE_MESSAGES = 8;

.....

public boolean isSystemOrForumAdmin() {
   return (values[FORUM_ADMIN] ||
values[SYSTEM_ADMIN]);
}

.....
```

```
}
```

因此,Forum 中各种操作权限是和 ForumPermissions 定义的用户级别有关系的,作为接口 Forum 的实现:ForumProxy 正是将这种对应关系联系起来.比如,修改 Forum 的名称,只有论坛管理者或系统管理者可以修改,代码如下:

```
public class ForumProxy implements Forum {

private ForumPermissions permissions;
private Forum forum;
this.authorization = authorization;

public ForumProxy(Forum forum, Authorization
authorization,
ForumPermissions permissions)
{
this.forum = forum;
this.authorization = authorization;
this.permissions = permissions;
}

.....

public void setName(String name) throws
UnauthorizedException,
ForumAlreadyExistsException
{
   //只有是系统或论坛管理者才可以修改名称
   if (permissions.isSystemOrForumAdmin()) {
      forum.setName(name);
```

```
    }
    else {
        throw new UnauthorizedException();
    }
}

...

}
```

而 DbForum 才是接口 Forum 的真正实现,以修改论坛名称为例:

```
public class DbForum implements Forum, Cacheable {
...

public void setName(String name) throws
ForumAlreadyExistsException {

    ....

    this.name = name;
    //这里真正将新名称保存到数据库中
    saveToDb();

    ....
}


...

}
```

凡是涉及到对论坛名称修改这一事件,其他程序都首先得和 ForumProxy 打交道,由 ForumProxy 决定是否有权限做某一样事情,ForumProxy 是个名副其实的"网关","安全代理系统".

在平时应用中,无可避免总要涉及到系统的授权或安全体系,不管你有无意识的使用 Proxy,实际你已经在使用 Proxy 了.

我们继续结合Jive谈入深一点,下面要涉及到工厂模式了,如果你不了解工厂模式,请看我的另外一篇文章:设计模式之Factory

我们已经知道,使用Forum需要通过ForumProxy,Jive中创建一个Forum是使用Factory模式,有一个总的抽象类ForumFactory,在这个抽象类中,调用ForumFactory是通过getInstance()方法实现,这里使用了Singleton(也是设计模式之一,由于介绍文章很多,我就不写了,看这里),getInstance()返回的是ForumFactoryProxy.

为什么不返回 ForumFactory,而返回 ForumFactory 的实现 ForumFactoryProxy?原因是明显的,需要通过代理确定是否有权限创建 forum.

在 ForumFactoryProxy 中我们看到代码如下:

```
public class ForumFactoryProxy extends ForumFactory {

    protected ForumFactory factory;
    protected Authorization authorization;
    protected ForumPermissions permissions;

    public ForumFactoryProxy(Authorization authorization,
ForumFactory factory,
    ForumPermissions permissions)
    {
        this.factory = factory;
        this.authorization = authorization;
```

```
        this.permissions = permissions;
    }

    public Forum createForum(String name, String description)
            throws UnauthorizedException,
ForumAlreadyExistsException
    {
        //只有系统管理者才可以创建 forum
        if (permissions.get(ForumPermissions.SYSTEM_ADMIN)) {
            Forum newForum = factory.createForum(name, description);
            return new ForumProxy(newForum, authorization,
permissions);
        }
        else {
            throw new UnauthorizedException();
    }
}
```

方法 createForum 返回的也是 ForumProxy, Proxy 就象一道墙,其他程序只能和 Proxy 交互操作.

注意到这里有两个 Proxy:ForumProxy 和 ForumFactoryProxy. 代表两个不同的职责:使用 Forum 和创建 Forum;
至于为什么将使用对象和创建对象分开,这也是为什么使用 Factory 模式的原因所在:是为了"封装" "分派";换句话说,尽可能功能单一化,方便维护修改.

Jive 论坛系统中其他如帖子的创建和使用,都是按照 Forum 这个思路而来的.

以上我们讨论了如何使用 Proxy 进行授权机制的访问,Proxy 还可以对用户隐藏另外一种称为 copy-on-write 的优化方式.拷贝一个庞大而复杂的对象是一个开销很大的操作,如

果拷贝过程中,没有对原来的对象有所修改,那么这样的拷贝开销就没有必要.用代理延迟这一拷贝过程.

比如:我们有一个很大的Collection,具体如hashtable,有很多客户端会并发同时访问它.其中一个特别的客户端要进行连续的数据获取,此时要求其他客户端不能再向hashtable 中增加或删除 东东.

最直接的解决方案是:使用 collection 的 lock,让这特别的客户端获得这个 lock,进行连续的数据获取,然后再释放 lock.

```
public void foFetches(Hashtable ht){
    synchronized(ht){
        //具体的连续数据获取动作..
    }

}
```

但是这一办法可能锁住 Collection 会很长时间,这段时间,其他客户端就不能访问该Collection 了.

第二个解决方案是clone这个Collection,然后让连续的数据获取针对clone出来的那个Collection操作.这个方案前提是,这个Collection是可clone的,而且必须有提供深度clone的方法.Hashtable就提供了对自己的clone方法,但不是Key和value对象的clone,关于Clone含义可以参考专门文章.

```
public void foFetches(Hashtable ht){

    Hashttable newht=(Hashtable)ht.clone();

}
```

问题又来了,由于是针对 clone 出来的对象操作,如果原来的母体被其他客户端操作修改了,那么对 clone 出来的对象操作就没有意义了.

最后解决方案:我们可以等其他客户端修改完成后再进行 clone,也就是说,这个特别的客户端先通过调用一个叫 clone 的方法来进行一系列数据获取操作.但实际上没有真正的进行对象拷贝,直至有其他客户端修改了这个对象 Collection.

使用 Proxy 实现这个方案.这就是 copy-on-write 操作.

Proxy 应用范围很广,现在流行的分布计算方式 RMI 和 Corba 等都是 Proxy 模式的应用.

更多Proxy应用,见

http://www.research.umbc.edu/~tarr/cs491/lectures/Proxy.pdf

设计模式之 Facade(外观 总管 Manager)

Facade 模式的定义：为子系统中的一组接口提供一个一致的界面.

Facade 一个典型应用就是数据库 JDBC 的应用,如下例对数据库的操作:

```
public class DBCompare {

   Connection conn = null;
   PreparedStatement prep = null;
   ResultSet rset = null;
   try {
       Class.forName( "<driver>" ).newInstance();
       conn = DriverManager.getConnection( "<database>" );

       String sql = "SELECT * FROM <table> WHERE <column name> = ?";
       prep = conn.prepareStatement( sql );
       prep.setString( 1, "<column value>" );
       rset = prep.executeQuery();
       if( rset.next() ) {
            System.out.println( rset.getString( "<column
name" ) );
        }
   } catch( SException e ) {
       e.printStackTrace();
   } finally {
       rset.close();
       prep.close();
       conn.close();
   }
```

```
}
```

上例是 Jsp 中最通常的对数据库操作办法.

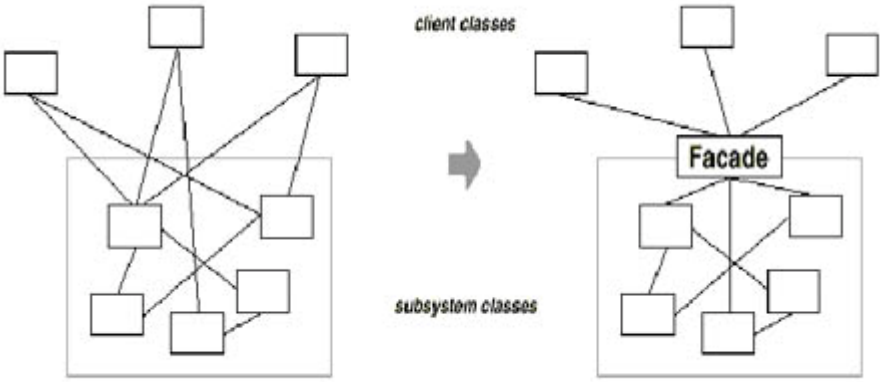
在应用中,经常需要对数据库操作,每次都写上述一段代码肯定比较麻烦,需要将其中不变的部分提炼出来,做成一个接口,这就引入了 facade 外观对象.如果以后我们更换 Class.forName 中的<driver>也非常方便,比如从 Mysql 数据库换到 Oracle 数据库,只要更换 facade 接口中的 driver 就可以.

我们做成了一个Facade接口,使用该接口,上例中的程序就可以更改如下:

```
public class DBCompare {


   String sql = "SELECT * FROM <table> WHERE <column name> = ?";


    try {
        Mysql msql=new mysql(sql);
        prep.setString( 1, "<column value>" );
        rset = prep.executeQuery();
        if( rset.next() ) {
              System.out.println( rset.getString( "<column
name" ) );
         }
    } catch( SException e ) {
        e.printStackTrace();
    } finally {
        mysql.close();
        mysql=null;
    }
}
```

可见非常简单,所有程序对数据库访问都是使用改接口,降低系统的复杂性,增加了灵活性.

如果我们要使用连接池,也只要针对 facade 接口修改就可以.



由上图可以看出, facade 实际上是个理顺系统间关系,降低系统间耦合度的一个常用的办法,也许你已经不知不觉在使用,尽管不知道它就是 facade.

行为模式

设计模式之 Iterator—点名篇

上了这么多年学，我发现一个问题，好象老师都很喜欢点名，甚至点名都成了某些老师的嗜好，一日不点名，就饭吃不香，觉睡不好似的，我就觉得很奇怪，你的课要是讲的好，同学又怎么会不来听课呢，殊不知：“误人子弟，乃是犯罪！”啊。

好了，那么我们现在来看老师这个点名过程是如何实现吧：

1、老规矩，我们先定义老师（Teacher）接口类：

```
public interface Teacher {
  public Iterator createIterator();   //点名
}
```

2、具体的老师（ConcreteTeacher）类是对老师（Teacher）接口的实现：

```
public class ConcreteTeacher implements Teacher{
  private Object[] present = {"张三来了","李四来了","王五没来"};  //同学出勤集合
  public Iterator createIterator(){
    return new ConcreteIterator(this);     //新的点名
  }
  public Object getElement(int index){  //得到当前同学的出勤情况
    if(index<present.length){
      return present[index];
    }
    else{
      return null;
    }
  }
  public int getSize(){
```

```
    return present.length;  //得到同学出勤集合的大小,也就是说要知道班上有多少人
  }
}
```

3、定义点名（Iterator）接口类：

```
public interface Iterator {
  void first();  //第一个
  void next();   //下一个
  boolean isDone();  //是否点名完毕
  Object currentItem();  //当前同学的出勤情况
}
```

4、具体的点名（ConcreteIterator）类是对点名（Iterator）接口的实现：

```
public class ConcreteIterator implements Iterator{
  private ConcreteTeacher teacher;
  private int index = 0;
  private int size = 0;
  public ConcreteIterator(ConcreteTeacher teacher){
    this.teacher = teacher;
    size = teacher.getSize(); //得到同学的数目
    index = 0;
  }
  public void first(){  //第一个
    index = 0;
  }
  public void next(){ //下一个
    if(index<size){
      index++;
    }
  }
  public boolean isDone(){ //是否点名完毕
```

```
    return (index>=size);
  }
  public Object currentItem(){ //当前同学的出勤情况
    return teacher.getElement(index);
  }
}
```

5、编写测试类：

```
public class Test {
  private Iterator it;
  private Teacher teacher = new ConcreteTeacher();
  public void operation(){
    it = teacher.createIterator();  //老师开始点名
    while(!it.isDone()){ //如果没点完
      System.out.println(it.currentItem().toString()); //获得被点到同学的情况
      it.next();   //点下一个
    }
  }
  public static void main(String agrs[]){
    Test test = new Test();
    test.operation();
  }
}
```

6、说明：

A：定义：Iterator 模式可以顺序的访问一个聚集中的元素而不必暴露聚集的内部情况。

B :在本例中，老师(Teacher)给出了创建点名(Iterator)对象的接口,点名(Iterator)定义了遍历同学出勤情况所需的接口。

C：Iterator 模式的优点是当（ConcreteTeacher）对象中有变化是，比如说同学出勤集合中有加入了新的同学，或减少同学时，这种改动对客户端是没有影响的。

设计模式之 Template

Template 模板模式定义:

定义一个操作中算法的骨架,将一些步骤的执行延迟到其子类中.

使用 Java 的抽象类时，就经常会使用到 Template 模式,因此 Template 模式使用很普遍.而且很容易理解和使用。

```
public abstract class Benchmark
{
    /**
    * 下面操作是我们希望在子类中完成
    */
    public abstract void benchmark();

    /**
    * 重复执行 benchmark 次数
    */
    public final long repeat (int count) {
        if (count <= 0)
            return 0;
        else {
            long startTime = System.currentTimeMillis();

        for (int i = 0; i < count; i++)
            benchmark();

        long stopTime = System.currentTimeMillis();
        return stopTime - startTime;
    }
}
```

```
}
```

在上例中,我们希望重复执行 benchmark()操作,但是对 benchmark()的具体内容没有说明,而是延迟到其子类中描述:

```
public class MethodBenchmark extends Benchmark
{
    /**
    * 真正定义 benchmark 内容
    */
    public void benchmark() {

        for (int i = 0; i < Integer.MAX_VALUE; i++){
           System.out.printtln("i="+i);
       }
    }
}
```

至此,Template 模式已经完成,是不是很简单?

我们称 repeat 方法为模板方法， 它其中的 benchmark()实现被延迟到子类 MethodBenchmark 中实现了,

看看如何使用:

```
Benchmark operation = new MethodBenchmark();
long duration = operation.repeat(Integer.parseInt(args[0].trim()));
System.out.println("The operation took " + duration + "
milliseconds");
```

也许你以前还疑惑抽象类有什么用,现在你应该彻底明白了吧? 至于这样做的好处,很显然啊,扩展性强,以后 Benchmark 内容变化,我只要再做一个继承子类就可以,不必修改其他应用代码.

设计模式之 Chain of Responsibility(职责链)

Chain of Responsibility 定义

Chain of Responsibility(CoR) 是用一系列类(classes)试图处理一个请求 request,这些类之间是一个松散的耦合,唯一共同点是在他们之间传递 request. 也就是说，来了一个请求，A 类先处理，如果没有处理，就传递到 B 类处理，如果没有处理，就传递到 C 类处理，就这样象一个链条(chain)一样传递下去。

如何使用?

虽然这一段是如何使用 CoR,但是也是演示什么是 CoR.

有一个 Handler 接口:

```
public interface Handler{
    public void handleRequest();
}
```

这是一个处理 request 的事例， 如果有多种 request,比如 请求帮助 请求打印 或请求格式化：

最先想到的解决方案是：在接口中增加多个请求：

```
public interface Handler{
    public void handleHelp();
    public void handlePrint();
    public void handleFormat();

}
```

具体是一段实现接口 Handler 代码：

```
public class ConcreteHandler implements Handler{
    private Handler successor;
```

```
    public ConcreteHandler(Handler successor){
    this.successor=successor;
}

    public void handleHelp(){
        //具体处理请求 Help 的代码
        ...
    }

    public void handlePrint(){
        //如果是 print 转去处理 Print
        successor.handlePrint();
    }
    public void handleFormat(){
        //如果是 Format 转去处理 format
        successor.handleFormat();
    }

}
```

一共有三个这样的具体实现类，上面是处理 help,还有处理 Print 处理 Format 这大概是我们最常用的编程思路。

虽然思路简单明了，但是有一个扩展问题，如果我们需要再增加一个请求 request 种类，需要修改接口及其每一个实现。

第二方案:将每种 request 都变成一个接口，因此我们有以下代码 ：

```
public interface HelpHandler{
    public void handleHelp();
}
```

```
public interface PrintHandler{
    public void handlePrint();
}

public interface FormatHandler{
    public void handleFormat();
}

public class ConcreteHandler
    implements HelpHandler,PrintHandler,FormatHandlet{
    private HelpHandler helpSuccessor;
    private PrintHandler printSuccessor;
    private FormatHandler formatSuccessor;

    public ConcreteHandler(HelpHandler helpSuccessor,PrintHandler
printSuccessor,FormatHandler                    formatSuccessor)
    {
        this.helpSuccessor=helpSuccessor;
        this.printSuccessor=printSuccessor;
        this.formatSuccessor=formatSuccessor;
    }

    public void handleHelp(){
        .......
    }

    public void handlePrint(){this.printSuccessor=printSuccessor;}

    public void
handleFormat(){this.formatSuccessor=formatSuccessor;}

}
```

这个办法在增加新的请求 request 情况下，只是节省了接口的修改量，接口实现 ConcreteHandler 还需要修改。而且代码显然不简单美丽。

解决方案 3: 在 Handler 接口中只使用一个参数化方法：

```
public interface Handler{
    public void handleRequest(String request);
}
```

那么 Handler 实现代码如下：

```
public class ConcreteHandler implements Handler{
    private Handler successor;

    public ConcreteHandler(Handler successor){
        this.successor=successor;
    }

    public void handleRequest(String request){
        if (request.equals("Help")){
            //这里是处理 Help 的具体代码
        }else
            //传递到下一个
            successor.handle(request);

        }
    }

}
```

这里先假设 request 是 String 类型，如果不是怎么办？当然我们可以创建一个专门类 Request

最后解决方案:接口 Handler 的代码如下：

```
public interface Handler{
```

```
    public void handleRequest(Request request);
}
```

Request 类的定义:

```
public class Request{
    private String type;

    public Request(String type){this.type=type;}

    public String getType(){return type;}

    public void execute(){
        //request 真正具体行为代码
    }
}
```

那么 Handler 实现代码如下：

```
public class ConcreteHandler implements Handler{
    private Handler successor;

    public ConcreteHandler(Handler successor){
        this.successor=successor;
    }

    public void handleRequest(Request request){
        if (request instanceof HelpRequest){
            //这里是处理 Help 的具体代码
        }else if (request instanceof PrintRequst){
            request.execute();
        }else
            //传递到下一个
            successor.handle(request);
```

```
        }
    }

}
```

这个解决方案就是 CoR, 在一个链上,都有相应职责的类,因此叫 **Chain of Responsibility**.

CoR的优点：

因为无法预知来自外界（客户端）的请求是属于哪种类型，每个类如果碰到它不能处理的请求只要放弃就可以。

缺点是效率低，因为一个请求的完成可能要遍历到最后才可能完成，当然也可以用树的概念优化。 在 Java AWT1.0 中，对于鼠标按键事情的处理就是使用 CoR,到 Java.1.1 以后，就使用 Observer 代替 CoR

扩展性差，因为在 CoR 中，一定要有一个统一的接口 Handler.局限性就在这里。

与Command模式区别：

Command 模式需要事先协商客户端和服务器端的调用关系，比如 1 代表 start 2 代表 move 等，这些 都是封装在 request 中，到达服务器端再分解。

CoR 模式就无需这种事先约定，服务器端可以使用 CoR 模式进行客户端请求的猜测，一个个猜测 试验。

设计模式之 Memento(备忘机制)

Memento 备望录模式定义:

memento 是一个保存另外一个对象内部状态拷贝的对象.这样以后就可以将该对象恢复到原先保存的状态.

Memento 模式相对也比较好理解,我们看下列代码:

```
public class Originator {


    private int number;

   private File file = null;

   public Originator(){}

   // 创建一个 Memento
   public Memento getMemento(){
       return new Memento(this);
   }

   // 恢复到原始值
   public void setMemento(Memento m){
        number = m.number;
        file = m.file;
   }

}
```

我们再看看 Memento 类:

```
private class Memento implements java.io.Serializable{
```

```
    private int number;

    private File file = null;

    public Memento( Originator o){

        number = o.number;
        file = o.file;

    }

}
```

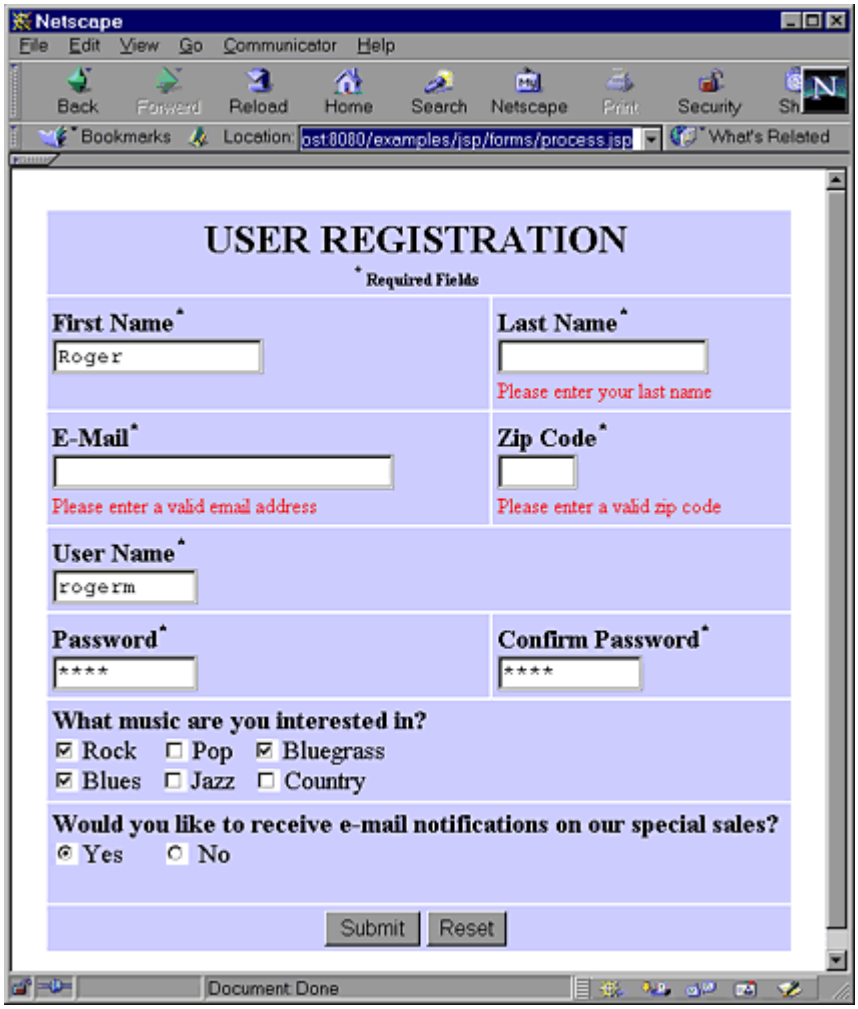
可见 Memento 中保存了 Originator 中的 number 和 file 的值. 通过调用 Originator 中 number 和 file 值改变的话,通过调用 setMemento()方法可以恢复.

Memento 模式的缺点是耗费大,如果内部状态很多,再保存一份,无意要浪费大量内存.

Memento 模式在 Jsp+Javabean 中的应用

在 Jsp 应用中,我们通常有很多表单要求用户输入,比如用户注册,需要输入姓名和 Email 等, 如果一些表项用户没有填写或者填写错误,我们希望在用户按"提交 Submit"后,通过 Jsp 程序检查,发现确实有未填写项目,则在该项目下红字显示警告或错误,同时,还要显示用户刚才已经输入的表项.

如下图中 First Name 是用户已经输入,Last Name 没有输入,我们则提示红字警告.:



这种技术的实现,就是利用了 Javabean 的 scope="request"或 scope="session"特性,也就是 Memento 模式.

设计模式之 Mediator(中介者)

Mediator 中介者模式定义:

用一个中介对象来封装一系列关于对象交互行为.

为何使用 Mediator?

各个对象之间的交互操作非常多;每个对象的行为操作都依赖彼此对方,修改一个对象的行为,同时会涉及到修改很多其他对象的行为,如果使用 Mediator 模式,可以使各个对象间的耦合松散,只需关心和 Mediator 的关系,使多对多的关系变成了一对多的关系,可以降低系统的复杂性,提高可修改扩展性.

如何使用?

首先 有一个接口,用来定义成员对象之间的交互联系方式:

```
public interface Mediator { }
```

Meiator 具体实现,真正实现交互操作的内容:

```
public class ConcreteMediator implements Mediator {

    //假设当前有两个成员.
    private ConcreteColleague1 colleague1 = new
ConcreteColleague1();
    private ConcreteColleague2 colleague2 = new
ConcreteColleague2();

    ...

}
```

再看看另外一个参与者:成员,因为是交互行为,都需要双方提供一些共同接口,这种要求在 Visitor Observer 等模式中都是相同的.

```
public class Colleague {
    private Mediator mediator;
    public Mediator getMediator() {
        return mediator;
    }

    public void setMediator( Mediator mediator ) {
        this.mediator = mediator;
    }
}

public class ConcreteColleague1 { }

public class ConcreteColleague2 { }
```

每个成员都必须知道 Mediator,并且和 Mediator 联系,而不是和其他成员联系.

至此,Mediator 模式框架完成,可以发现 Mediator 模式规定不是很多,大体框架也比较简单,但实际使用起来就非常灵活.

Mediator 模式在事件驱动类应用中比较多,例如界面设计 GUI.;聊天,消息传递等,在聊天应用中,需要有一个 MessageMediator,专门负责 request/reponse 之间任务的调节.

MVC 是 J2EE 的一个基本模式,View Controller 是一种 Mediator,它是 Jsp 和服务器上应用程序间的 Mediator.

设计模式之 Interpreter(解释器)

Interpreter 解释器模式定义:

定义语言的文法 ,并且建立一个解释器来解释该语言中的句子.

Interpreter 似乎使用面不是很广,它描述了一个语言解释器是如何构成的,在实际应用中,我们可能很少去构造一个语言的文法.我们还是来简单的了解一下:

首先要建立一个接口,用来描述共同的操作.

```
public interface AbstractExpression {
    void interpret( Context context );
}
```

再看看包含解释器之外的一些全局信息

```
public interface Context { }
```

AbstractExpression 的具体实现分两种:终结符表达式和非终结符表达式:

```
public class TerminalExpression implements AbstractExpression {
    public void interpret( Context context ) { }
}
```

对于文法中没一条规则,非终结符表达式都必须的:

```
public class NonterminalExpression implements AbstractExpression {
    private AbstractExpression successor;

    public void setSuccessor( AbstractExpression successor ) {
       this.successor = successor;
    }

    public AbstractExpression getSuccessor() {
```

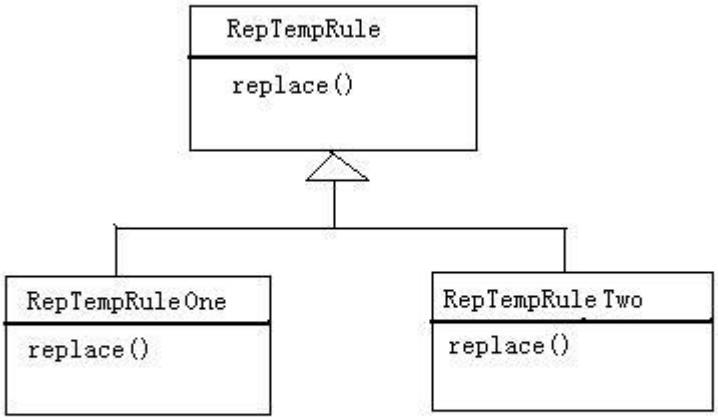
```
        return successor;
    }

    public void interpret( Context context ) { }
}
```

设计模式之 Strategy(策略)

Strategy 策略模式是属于设计模式中 对象行为型模式,主要是定义一系列的算法,把这些算法一个个封装成单独的类.

Stratrgy 应用比较广泛,比如, 公司经营业务变化图, 可能有两种实现方式,一个是线条曲线,一个是框图(bar),这是两种算法,可以使用 Strategy 实现.

这里以字符串替代为例, 有一个文件,我们需要读取后,希望替代其中相应的变量,然后输出.关于替代其中变量的方法可能有多种方法,这取决于用户的要求,所以我们要准备几套变量字符替代方案.



首先,我们建立一个抽象类 RepTempRule 定义一些公用变量和方法:

```
public abstract class RepTempRule{


protected String oldString="";
public void setOldString(String oldString){
```

```
    this.oldString=oldString;
}

protected String newString="";
public String getNewString(){
    return newString;
}




public abstract void replace() throws Exception;



}
```

在 RepTempRule 中 有一个抽象方法 abstract 需要继承明确,这个 replace 里其实是替代的具体方法.

我们现在有两个字符替代方案,

1.将文本中 aaa 替代成 bbb;

2.将文本中 aaa 替代成 ccc;

对应的类分别是 RepTempRuleOne RepTempRuleTwo

```
public class RepTempRuleOne extends RepTempRule{



public void replace() throws Exception{

    //replaceFirst 是 jdk1.4 新特性
```

```
    newString=oldString.replaceFirst("aaa", "bbbb")
    System.out.println("this is replace one");

}



}
```

```
public class RepTempRuleTwo extends RepTempRule{



public void replace() throws Exception{

    newString=oldString.replaceFirst("aaa", "ccc")
    System.out.println("this is replace Two");

}



}
```

第二步：我们要建立一个算法解决类，用来提供客户端可以自由选择算法。

```
    public class RepTempRuleSolve {

    private RepTempRule strategy;

    public RepTempRuleSolve(RepTempRule rule){
        this.strategy=rule;
    }
```

```
   public String getNewContext(Site site,String
oldString) {
       return strategy.replace(site,oldString);
   }

   public void changeAlgorithm(RepTempRule
newAlgorithm) {
       strategy = newAlgorithm;
   }

}
```

调用如下:

```
public class test{

......

   public void testReplace(){

   //使用第一套替代方案
   RepTempRuleSolve solver=new RepTempRuleSolve(new
RepTempRuleSimple());
   solver.getNewContext(site,context);

   //使用第二套

   solver=new RepTempRuleSolve(new RepTempRuleTwo());
   solver.getNewContext(site,context);

   }
```

```
.....

}
```

我们达到了在运行期间，可以自由切换算法的目的。

实际整个 Strategy 的核心部分就是抽象类的使用,使用 Strategy 模式可以在用户需要变化时,修改量很少,而且快速.

Strategy和Factory有一定的类似,Strategy相对简单容易理解,并且可以在运行时刻自由切换。Factory 重点是用来创建对象。

Strategy 适合下列场合:

1.以不同的格式保存文件;

2.以不同的算法压缩文件;

3.以不同的算法截获图象;

4.以不同的格式输出同样数据的图形,比如曲线 或框图 bar 等

设计模式之 State

State 模式的定义：不同的状态,不同的行为;或者说,每个状态有着相应的行为.

何时使用?

State 模式在实际使用中比较多,适合"状态的切换".因为我们经常会使用 If elseif else 进行状态切换, 如果针对状态的这样判断切换反复出现,我们就要联想到是否可以采取 State 模式了.

不只是根据状态,也有根据属性.如果某个对象的属性不同,对象的行为就不一样,这点在数据库系统中出现频率比较高,我们经常会在一个数据表的尾部,加上 property 属性含义的字段,用以标识记录中一些特殊性质的记录,这种属性的改变(切换)又是随时可能发生的,就有可能要使用 State.

是否使用?

在实际使用,类似开关一样的状态切换是很多的,但有时并不是那么明显,取决于你的经验和对系统的理解深度.

这里要阐述的是"开关切换状态" 和" 一般的状态判断"是有一些区别的, " 一般的状态判断"也是有 if..elseif 结构,例如:

```
if (which==1) state="hello";
else if (which==2) state="hi";
else if (which==3) state="bye";
```

这是一个 " 一般的状态判断",state 值的不同是根据 which 变量来决定的,which 和 state 没有关系.如果改成:

```
if (state.euqals("bye")) state="hello";
else if (state.euqals("hello")) state="hi";
else if (state.euqals("hi")) state="bye";
```

这就是 "开关切换状态",是将 state 的状态从"hello"切换到"hi",再切换到""bye";在切换到"hello",好象一个旋转开关,这种状态改变就可以使用 State 模式了.

如果单纯有上面一种将"hello"-->"hi"-->"bye"-->"hello"这一个方向切换,也不一定需要使用 State 模式,因为 State 模式会建立很多子类,复杂化,但是如果又发生另外一个行为:将上面的切换方向反过来切换,或者需要任意切换,就需要 State 了.

请看下例:

```
public class Context{

   private Color state=null;

   public void push(){

       //如果当前 red 状态 就切换到 blue
       if (state==Color.red) state=Color.blue;

       //如果当前 blue 状态 就切换到 green
       else if (state==Color.blue) state=Color.green;

       //如果当前 black 状态 就切换到 red
       else if (state==Color.black) state=Color.red;

       //如果当前 green 状态 就切换到 black
       else if (state==Color.green) state=Color.black;

       Sample sample=new Sample(state);
       sample.operate();
   }

   public void pull(){
```

```
        //与 push 状态切换正好相反

        if (state==Color.green) state=Color.blue;
        else if (state==Color.black) state=Color.green;
        else if (state==Color.blue) state=Color.red;
        else if (state==Color.red) state=Color.black;

        Sample2 sample2=new Sample2(state);
        sample2.operate();
    }
}
```

在上例中,我们有两个动作 push 推和 pull 拉,这两个开关动作,改变了 Context 颜色,至此,我们就需要使用 State 模式优化它.

另外注意:但就上例,state 的变化,只是简单的颜色赋值,这个具体行为是很简单的,State 适合巨大的具体行为,因此在,就本例,实际使用中也不一定非要使用 State 模式,这会增加子类的数目,简单的变复杂.

例如: 银行帐户, 经常会在 Open 状态和 Close 状态间转换.

例如: 经典的 TcpConnection, Tcp 的状态有创建 侦听 关闭三个,并且反复转换,其创建 侦听 关闭的具体行为不是简单一两句就能完成的,适合使用 State

例如:信箱 POP 帐号, 会有四种状态, start HaveUsername Authorized quit,每个状态对应的行为应该是比较大的.适合使用 State

例如:在工具箱挑选不同工具,可以看成在不同工具中切换,适合使用 State.如 具体绘图程序,用户可以选择不同工具绘制方框 直线 曲线,这种状态切换可以使用 State.

如何使用

State 需要两种类型实体参与:

1.state manager 状态管理器 ,就是开关 ,如上面例子的 Context 实际就是一个 state manager, 在 state manager 中有对状态的切换动作.
2.用抽象类或接口实现的父类,,不同状态就是继承这个父类的不同子类.

以上面的 Context 为例.我们要修改它,建立两个类型的实体.

第一步: 首先建立一个父类:

```
public abstract class State{

   public abstract void handlepush(Context c);
   public abstract void handlepull(Context c);
   public abstract void getcolor();

}
```

父类中的方法要对应 state manager 中的开关行为,在 state manager 中 本例就是 Context 中,有两个开关动作 push 推和 pull 拉.那么在状态父类中就要有具体处理这两个动作:handlepush() handlepull(); 同时还需要一个获取 push 或 pull 结果的方法 getcolor()

下面是具体子类的实现:

```
public class BlueState extends State{

   public void handlepush(Context c){
        //根据 push 方法"如果是 blue 状态的切换到 green" ;
        c.setState(new GreenState());

   }
```

```
   public void handlepull(Context c){

        //根据 pull 方法"如果是 blue 状态的切换到 red" ;
       c.setState(new RedState());

   }

   public abstract void getcolor(){ return
(Color.blue)}

}
```

同样 其他状态的子类实现如 blue 一样.

第二步: 要重新改写 State manager 也就是本例的 Context:

```
public class Context{

   private Sate state=null; //我们将原来的 Color state 改成了新建的
State state;

   //setState 是用来改变 state 的状态 使用 setState 实现状态的切换
   pulic void setState(State state){

       this.state=state;

   }

   public void push(){

       //状态的切换的细节部分,在本例中是颜色的变化,已经封装在子类的
handlepush 中实现,这里无需关心
```

```
        state.handlepush(this);

        //因为 sample 要使用 state 中的一个切换结果,使用 getColor()
        Sample sample=new Sample(state.getColor());
        sample.operate();

    }



    public void pull(){

        state.handlepull(this);

        Sample2 sample2=new Sample2(state.getColor());
        sample2.operate();

    }

}
```

至此,我们也就实现了 State 的 refactorying 过程.

以上只是相当简单的一个实例,在实际应用中,handlepush 或 handelpull 的处理是复杂的.

状态模式优点：

（1） 封装转换过程，也就是转换规则

（2） 枚举可能的状态，因此，需要事先确定状态种类。

状态模式可以允许客户端改变状态的转换行为，而状态机则是能够自动改变状态，状态机是一个比较独立的而且复杂的机制，具体可参考一个状态机开源项目：http://sourceforge.net/projects/smframework/

状态模式在工作流或游戏等各种系统中有大量使用，甚至是这些系统的核心功能设计，例如政府 OA 中，一个批文的状态有多种：未办；正在办理；正在批示；正在审核；已经完成等各种状态，使用状态机可以封装这个状态的变化规则，从而达到扩充状态时，不必涉及到状态的使用者。

在网络游戏中，一个游戏活动存在开始；开玩；正在玩；输赢等各种状态，使用状态模式就可以实现游戏状态的总控，而游戏状态决定了游戏的各个方面，使用状态模式可以对整个游戏架构功能实现起到决定的主导作用。

状态模式实质：

使用状态模式前，客户端外界需要介入改变状态，而状态改变的实现是琐碎或复杂的。

使用状态模式后，客户端外界可以直接使用事件 Event 实现，根本不必关心该事件导致如何状态变化，这些是由状态机等内部实现。

这是一种 Event-condition-State，状态模式封装了 condition-State 部分。

每个状态形成一个子类，每个状态只关心它的下一个可能状态，从而无形中形成了状态转换的规则。如果新的状态加入，只涉及它的前一个状态修改和定义。

状态转换有几个方法实现：一个在每个状态实现 next()，指定下一个状态；还有一种方法，设定一个 StateOwner，在 StateOwner 设定 stateEnter 状态进入和 stateExit 状态退出行为。

状态从一个方面说明了流程，流程是随时间而改变，状态是截取流程某个时间片。

相关文章：

从工作流状态机实践中总结状态模式使用心得

参考资源:

the State and Stategy

How to implement state-dependent behavior

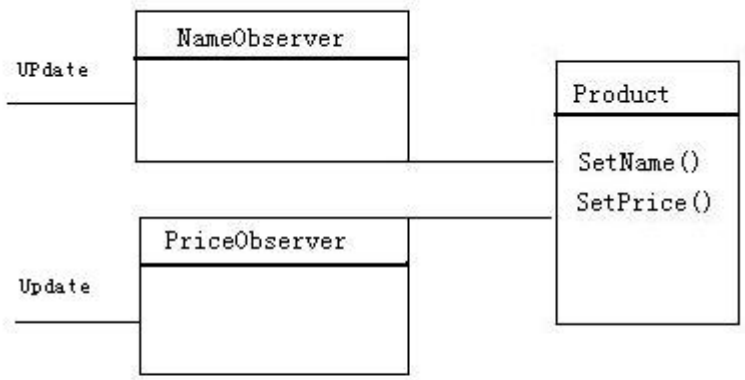
The state patterns

设计模式之 Observer

Java 深入到一定程度,就不可避免的碰到设计模式(design pattern)这一概念,了解设计模式,将使自己对 java 中的接口或抽象类应用有更深的理解.设计模式在 java 的中型系统中应用广泛,遵循一定的编程模式,才能使自己的代码便于理解,易于交流,Observer(观察者)模式是比较常用的一个模式,尤其在界面设计中应用广泛,而本站所关注的是 Java 在电子商务系统中应用,因此想从电子商务实例中分析 Observer 的应用.

虽然网上商店形式多样,每个站点有自己的特色,但也有其一般的共性,单就"商品的变化,以便及时通知订户"这一点,是很多网上商店共有的模式,这一模式类似 Observer patern 观察者模式.

具体的说,如果网上商店中商品在名称 价格等方面有变化,如果系统能自动通知会员,将是网上商店区别传统商店的一大特色.这就需要在商品 product 中加入 Observer 这样角色,以便 product 细节发生变化时,Observer 能自动观察到这种变化,并能进行及时的 update 或 notify 动作.



Java 的 API 还为为我们提供现成的 Observer 接口 Java.util.Observer.我们只要直接使用它就可以.

我们必须 extends Java.util.Observer 才能真正使用它:

1.提供 Add/Delete observer 的方法;

2.提供通知(notisfy) 所有 observer 的方法;

```
    //产品类 可供 Jsp 直接使用 UseBean 调用 该类主要执行产品数据库插入 更新
public class product extends Observable{

    private String name;
    private float price;

    public String getName(){ return name;}
    public void setName(String name){
     this.name=name;
    //设置变化点
     setChanged();
     notifyObservers(name);

    }

    public float getPrice(){ return price;}
    public void setPrice(float price){
     this.price=price;
    //设置变化点
     setChanged();
     notifyObservers(new Float(price));

    }


    //以下可以是数据库更新 插入命令.
    public void saveToDb(){
```

```
    ..................

}
```

我们注意到,在product类中 的setXXX方法中,我们设置了 notify(通知)方法, 当Jsp表单调用setXXX(如何调用见我的另外一篇文章),实际上就触发了notisfyObservers方法,这将通知相应观察者应该采取行动了.

下面看看这些观察者的代码,他们究竟采取了什么行动:

```
//观察者 NameObserver 主要用来对产品名称(name)进行观察的
public class NameObserver implements Observer{

    private String name=null;

    public void update(Observable obj,Object arg){

        if (arg instanceof String){

         name=(String)arg;
         //产品名称改变值在 name 中
         System.out.println("NameObserver :name changet to "+name);

        }

    }

}

//观察者 PriceObserver 主要用来对产品价格(price)进行观察的
public class PriceObserver implements Observer{
```

```
   private float price=0;

   public void update(Observable obj,Object arg){

       if (arg instanceof Float){

        price=((Float)arg).floatValue();

        System.out.println("PriceObserver :price changet to
"+price);

       }

   }

}
```

Jsp 中我们可以来正式执行这段观察者程序:

```
<jsp:useBean id="product" scope="session" class="Product" />
<jsp:setProperty name="product" property="*" />

<jsp:useBean id="nameobs" scope="session" class="NameObserver" />
<jsp:setProperty name="product" property="*" />

<jsp:useBean id="priceobs" scope="session" class="PriceObserver"
/>
<jsp:setProperty name="product" property="*" />

<%

if (request.getParameter("save")!=null)
```

```
{
    product.saveToDb();

    out.println("产品数据变动 保存! 并已经自动通知客户");
}else{

    //加入观察者
    product.addObserver(nameobs);

    product.addObserver(priceobs);
%>

    //request.getRequestURI()是产生本 jsp 的程序名,就是自己调用自己
    <form action="<%=request.getRequestURI()%>" method=post>

    <input type=hidden name="save" value="1">
    产品名称:<input type=text name="name" >
    产品价格:<input type=text name="price">
    <input type=submit>

    </form>
<%

}
%>
```

执行改 Jsp 程序,会出现一个表单录入界面, 需要输入产品名称 产品价格, 点按 Submit 后,还是执行该 jsp 的
if (request.getParameter("save")!=null)之间的代码.

由于这里使用了数据 javabeans 的自动赋值概念,实际程序自动执行了 setName setPrice 语句.你会在服务器控制台中发现下面信息::

NameObserver :name changet to ?????(Jsp 表单中输入的产品名称)

PriceObserver :price changet to ???(Jsp 表单中输入的产品价格);

这说明观察者已经在行动了.!!
同时你会在执行 jsp 的浏览器端得到信息:

产品数据变动 保存! 并已经自动通知客户

上文由于使用 jsp 概念,隐含很多自动动作,现将调用观察者的 Java 代码写如下:

```
public class Test {

   public static void main(String args[]){

Product product=new Product();

NameObserver nameobs=new NameObserver();
PriceObserver priceobs=new PriceObserver();

//加入观察者
product.addObserver(nameobs);
```

```
product.addObserver(priceobs);

product.setName("橘子红了");
product.setPrice(9.22f);

    }

}
```

你会在发现下面信息::

NameObserver :name changet to 橘子红了

PriceObserver :price changet to 9.22

这说明观察者在行动了.!!

设计模式之 Visitor

Visitor 访问者模式定义

作用于某个对象群中各个对象的操作. 它可以使你在不改变这些对象本身的情况下,定义作用于这些对象的新操作.

在 Java 中,Visitor 模式实际上是分离了 collection 结构中的元素和对这些元素进行操作的行为.

为何使用 Visitor?

Java 的 Collection(包括 Vector 和 Hashtable)是我们最经常使用的技术,可是 Collection 好象是个黑色大染缸,本来有各种鲜明类型特征的对象一旦放入后,再取出时,这些类型就消失了.那么我们势必要用 If 来判断,如:

```
Iterator iterator = collection.iterator()
while (iterator.hasNext()) {
    Object o = iterator.next();
    if (o instanceof Collection)
        messyPrintCollection((Collection)o);
    else if (o instanceof String)
        System.out.println("'"+o.toString()+"'");
    else if (o instanceof Float)
        System.out.println(o.toString()+"f");
    else
        System.out.println(o.toString());
}
```

在上例中,我们使用了 instanceof 来判断 o 的类型.

很显然,这样做的缺点代码 If else if 很繁琐.我们就可以使用 Visitor 模式解决它.

如何使用 Visitor?

针对上例,定义接口叫 Visitable,用来定义一个 Accept 操作,也就是说让 Collection 每个元素具备可访问性.

被访问者是我们 Collection 的每个元素 Element,我们要为这些 Element 定义一个可以接受访问的接口(访问和被访问是互动的,只有访问者,被访问者如果表示不欢迎,访问者就不能访问),取名为 Visitable，也可取名为 Element。

```
public interface Visitable
{
    public void accept(Visitor visitor);
}
```

被访问的具体元素继承这个新的接口 Visitable：

```
public class StringElement implements Visitable
{
    private String value;
    public StringElement(String string) {
        value = string;
    }

    public String getValue(){
        return value;
    }


    //定义 accept 的具体内容 这里是很简单的一句调用
    public void accept(Visitor visitor) {
        visitor.visitString(this);
    }
```

```
}
```

上面是被访问者是字符串类型，下面再建立一个 Float 类型的：

```
public class FloatElement implements Visitable
{
    private Float value;
    public FloatElement(Float value) {
        this.value = value;
    }

    public Float getValue(){
        return value;
    }


    //定义 accept 的具体内容 这里是很简单的一句调用
    public void accept(Visitor visitor) {
        visitor.visitFloat(this);
    }
}
```

我们设计一个接口 visitor 访问者，在这个接口中,有一些访问操作，这些访问操作是专门访问对象集合 Collection 中有可能的所有类，目前我们假定有三个行为：访问对象集合中的字符串类型；访问对象集合中的 Float 类型；访问对象集合中的对象集合类型。注意最后一个类型是集合嵌套，通过这个嵌套实现可以看出使用访问模式的一个优点。

接口 visitor 访问者如下：

```
public interface Visitor
{

    public void visitString(StringElement stringE);
    public void visitFloat(FloatElement floatE);
    public void visitCollection(Collection
collection);

}
```

访问者的实现:

```
public class ConcreteVisitor implements Visitor
{
    //在本方法中,我们实现了对 Collection 的元素的成功访问
    public void visitCollection(Collection collection)
{
        Iterator iterator = collection.iterator()
        while (iterator.hasNext()) {
            Object o = iterator.next();
            if (o instanceof Visitable)
                ((Visitable)o).accept(this);
        }
    }

    public void visitString(StringElement stringE) {

System.out.println("'"+stringE.getValue()+"'");
    }
    public void visitFloat(FloatElement floatE){
```

```
System.out.println(floatE.getValue().toString()+"f");
    }

}
```

在上面的 visitCollection 我们实现了对 Collection 每个元素访问,只使用了一个判断语句,只要判断其是否可以访问.

StringElement 只是一个实现，可以拓展为更多的实现，整个核心奥妙在 accept 方法中，在遍历 Collection 时，通过相应的 accept 方法调用具体类型的被访问者。这一步确定了被访问者类型，

如果是 StringElement，而 StringElement 则回调访问者的 visiteString 方法，这一步实现了行为操作方法。

客户端代码：

```
Visitor visitor = new ConcreteVisitor();

StringElement stringE = new StringElement("I am a
String");
visitor.visitString(stringE);

Collection list = new ArrayList();
list.add(new StringElement("I am a String1"));
list.add(new StringElement("I am a String2"));
list.add(new FloatElement(new Float(12)));
list.add(new StringElement("I am a String3"));
visitor.visitCollection(list);
```

客户端代码中的 list 对象集合中放置了多种数据类型，对对象集合中的访问不必象一开始那样，使用 instance of 逐个判断，而是通过访问者模式巧妙实现了。

至此,我们完成了 Visitor 模式基本结构.

使用 Visitor 模式的前提

使用访问者模式是对象群结构中(Collection) 中的对象类型很少改变。

在两个接口 Visitor 和 Visitable 中,确保 Visitable 很少变化,也就是说，确保不能老有新的 Element 元素类型加进来，可以变化的是访问者行为或操作，也就是 Visitor 的不同子类可以有多种,这样使用访问者模式最方便.

如果对象集合中的对象集合经常有变化，那么不但 Visitor 实现要变化，Visistable 也要增加相应行为，GOF 建议是,不如在这些对象类中直接逐个定义操作，无需使用访问者设计模式。

但是在 Java 中，Java 的 Reflect 技术解决了这个问题，因此结合 reflect 反射机制，可以使得访问者模式适用范围更广了。

Reflect技术是在运行期间动态获取对象类型和方法的一种技术,具体实现参考 Javaworld的英文原文.

设计模式之 Command

Command 模式是最让我疑惑的一个模式,我在阅读了很多代码后,才感觉隐约掌握其大概原理,我认为理解设计模式最主要是掌握起原理构造,这样才对自己实际编程有指导作用.Command 模式实际上不是个很具体,规定很多的模式,正是这个灵活性,让人有些 confuse.

Command 定义

n 将来自客户端的请求传入一个对象，无需了解这个请求激活的 动作或有关接受这个请求的处理细节。

这是一种两台机器之间通讯联系性质的模式，类似传统过程语 言的 CallBack 功能。

优点：

解耦了发送者和接受者之间联系。 发送者调用一个操作，接受者接受请求执行相应的动作，因为使用 Command 模式解耦，发送者无需知道接受者任何接口。

不少 Command 模式的代码都是针对图形界面的,它实际就是菜单命令,我们在一个下拉菜单选择一个命令时,然后会执行一些动作.

将这些命令封装成在一个类中,然后用户(调用者)再对这个类进行操作,这就是 Command 模式,换句话说,本来用户(调用者)是直接调用这些命令的,如菜单上打开文档(调用者),就直接指向打开文档的代码,使用 Command 模式,就是在这两者之间增加一个中间者,将这种直接关系拗断,同时两者之间都隔离,基本没有关系了.

显然这样做的好处是符合封装的特性,降低耦合度,Command 是将对行为进行封装的典型模式,Factory 是将创建进行封装的模式,
从 Command 模式,我也发现设计模式一个"通病":好象喜欢将简单的问题复杂化， 喜欢在不同类中增加第三者,当然这样做有利于代码的健壮性 可维护性 还有复用性.

如何使用?

具体的 Command 模式代码各式各样,因为如何封装命令,不同系统,有不同的做法.下面事

例是将命令封装在一个 Collection 的 List 中,任何对象一旦加入 List 中,实际上装入了一个封闭的黑盒中,对象的特性消失了,只有取出时,才有可能模糊的分辨出:

典型的 Command 模式需要有一个接口.接口中有一个统一的方法,这就是"将命令/请求封装为对象":

```
public interface Command {
    public abstract void execute ( );
}
```

具体不同命令/请求代码是实现接口 Command,下面有三个具体命令

```
    public class Engineer implements Command {

    public void execute( ) {
        //do Engineer's command
    }
}

public class Programmer implements Command {

    public void execute( ) {
        //do programmer's command
    }
}

public class Politician implements Command {

    public void execute( ) {
        //do Politician's command
    }
```

```
}
```

按照通常做法,我们就可以直接调用这三个 Command,但是使用 Command 模式,我们要将他们封装起来,扔到黑盒子 List 里去:

```
public class producer{
   public static List produceRequests() {
      List queue = new ArrayList();
      queue.add( new DomesticEngineer() );
      queue.add( new Politician() );
      queue.add( new Programmer() );
      return queue;
   }

}
```

这三个命令进入 List 中后,已经失去了其外表特征,以后再取出,也可能无法分辨出谁是 Engineer 谁是 Programmer 了,看下面客户端如何调用 Command 模式:

```
public class TestCommand {
   public static void main(String[] args) {

      List queue = Producer.produceRequests();
      for (Iterator it = queue.iterator();
it.hasNext(); )

  //客户端直接调用 execute 方法,无需知道被调用者的其它更多类的
方法名。
             ((Command)it.next()).execute();
```

```
    }
}
```

由此可见,调用者基本只和接口打交道,不合具体实现交互,这也体现了一个原则,面向接口编程,这样,以后增加第四个具体命令时,就不必修改调用者 TestCommand 中的代码了.

理解了上面的代码的核心原理,在使用中,就应该各人有自己方法了,特别是在如何分离调用者和具体命令上,有很多实现方法,上面的代码是使用"从 List 过一遍"的做法.这种做法只是为了演示.

使用 Command 模式的一个好理由还因为它能实现 Undo 功能.每个具体命令都可以记住它刚刚执行的动作,并且在需要时恢复.

Command 模式在界面设计中应用广泛.Java 的 Swing 中菜单命令都是使用 Command 模式,由于 Java 在界面设计的性能上还有欠缺,因此界面设计具体代码我们就不讨论,网络上有很多这样的示例.