23 种设计模式汇集

如果你还不了解设计模式是什么的话？

那就先看**设计模式引言**！

⊙**学习 GoF 设计模式的重要性**

⊙**建筑和软件中模式之异同**

A. **创建模式**

- **设计模式之 Singleton(单态/单件)** 阎宏博士讲解：单例（Singleton）模式

  保证一个类只有一个实例,并提供一个访问它的全局访问点

- **设计模式之 Factory(工厂方法和抽象工厂)**

  使用工厂模式就象使用 new 一样频繁.

- **设计模式之 Builder**

  汽车由车轮 方向盘 发动机很多部件组成，同时，将这些部件组装成汽车也是一件复杂的工作，Builder 模式就是将这两种情况分开进行。

- **设计模式之 Prototype(原型)**

  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。

B. **结构模式**

- **设计模式之 Adapter(适配器)**

  使用类再生的两个方式:组合(new)和继承(extends),这个已经在 thinking in java 中提到过.

- **设计模式之 Proxy(代理)**

  以 Jive 为例,剖析代理模式在用户级别授权机制上的应用

- **设计模式之 Facade(门面？)**

  可扩展的使用 JDBC 针对不同的数据库编程,Facade 提供了一种灵活的实现.

- **设计模式之 Composite(组合)**

  就是将类用树形结构组合成一个单位.你向别人介绍你是某单位，你是单位中的一个元素，别人和你做买卖，相当于和单位做买卖。文章中还对 Jive 再进行了剖析。

- **设计模式之 Decorator(装饰器)**

  Decorator 是个油漆工,给你的东东的外表刷上美丽的颜色.

- **设计模式之 Bridge(桥连)**

  将牛郎织女分开(本应在一起,分开他们,形成两个接口),在他们之间搭建一个桥(动态的结合)

- **设计模式之 Flyweight(共享元)**

  提供 Java 运行性能,降低小而大量重复的类的开销.

C. **行为模式**

- **设计模式之 Command(命令)**

  什么是将行为封装,Command 是最好的说明.

- **设计模式之 Observer(观察者)**

  介绍如何使用 Java API 提供的现成 Observer

- 设计模式之 Iterator(迭代器)

  这个模式已经被整合入 Java 的 Collection.在大多数场合下无需自己制造一个 Iterator,只要将对象装入 Collection 中，直接使用 Iterator 进行对象遍历。

- **设计模式之 Template(模板方法)**

实际上向你介绍了为什么要使用 Java 抽象类,该模式原理简单,使用很普遍.

- **设计模式之 Strategy(策略)**

  不同算法各自封装,用户端可随意挑选需要的算法.

- **设计模式之 Chain of Responsibility(责任链)**

  各司其职的类串成一串,好象击鼓传花,当然如果自己能完成,就不要推委给下一个.

- **设计模式之 Mediator(中介)**

  Mediator 很象十字路口的红绿灯,每个车辆只需和红绿灯交互就可以.

- **设计模式之 State(状态)**

  状态是编程中经常碰到的实例,将状态对象化,设立状态变换器,便可在状态中轻松切换.

- **设计模式之 Memento(注释状态？)**

  很简单一个模式,就是在内存中保留原来数据的拷贝.

- **设计模式之 Interpreter(解释器)**

  主要用来对语言的分析,应用机会不多.

- **设计模式之 Visitor(访问者)**

  访问者在进行访问时,完成一系列实质性操作,而且还可以扩展.

设计模式引言

设计面向对象软件比较困难，而设计可复用的面向对象软件就更加困难。你必须找到相关的对象，以适当的粒度将它们归类，再定义类的接口和继承层次，建立对象之间的基本关系。你的设计应该对手头的问题有针对性，同时对将来的问题和需求也要有足够的通用性。

你也希望避免重复设计或尽可能少做重复设计。有经验的面向对象设计者会告诉你，要一下子就得到复用性和灵活性好的设计，即使不是不可能的至少也是非常困难的。一个设计在最终完成之前常要被复用好几次，而且每一次都有所修改。

有经验的面向对象设计者的确能做出良好的设计，而新手则面对众多选择无从下手，总是求助于以前使用过的非面向对象技术。新手需要花费较长时间领会良好的面向对象设计是怎么回事。有经验的设计者显然知道一些新手所不知道的东西，这又是什么呢？

内行的设计者知道：不是解决任何问题都要从头做起。他们更愿意复用以前使用过的解决方案。当找到一个好的解决方案，他们会一遍又一遍地使用。这些经验是他们成为内行的部分原因。因此，你会在许多面向对象系统中看到类和相互通信的对象（ c o m m u n i c a t i n g o b j e c t）的重复模式。这些模式解决特定的设计问题，使面向对象设计更灵活、优雅，最终复用性更好。它们帮助设计者将新的设计建立在以往工作的基础上，复用以往成功的设计方案。

一个熟悉这些模式的设计者不需要再去发现它们，而能够立即将它们应用于设计问题中。以下类比可以帮助说明这一点。小说家和剧本作家很少从头开始设计剧情。他们总是沿袭一些业已存在的模式，像“悲剧性英雄”模式(《麦克白》、《哈姆雷特》等)或“浪漫小说”模式(存在着无数浪漫小说)。同样地，面向对象设计员也沿袭一些模式，像“用对象表示状态”和“修饰对象以便于你能容易地添加/删除属性”等。一旦懂得了模式，许多设计决策自然而然就产生了。

我们都知道设计经验的重要价值。你曾经多少次有过这种感觉—你已经解决过了一个问题但就是不能确切知道是在什么地方或怎么解决的？如果你能记起以前问题的细节和怎么解决它的，你就可以复用以前的经验而不需要重新发现它。然而，我们并没有很好记录下可供他人使用的软件设计经验。

学习 GoF 设计模式的重要性

著名的 EJB 领域顶尖的专家 Richard Monson-Haefel 在其个人网站:www.EJBNow.com 中极力推荐的 GoF 的《设计模式》,原文如下:

Design Patterns

Most developers claim to experience an epiphany reading this book. If you've never read the Design Patterns book then you have suffered a very serious gap in your programming education that should be remedied immediately.

翻译: 很多程序员在读完这本书,宣布自己相当于经历了一次"主显节"(纪念那稣降生和受洗的双重节日),如果你从来没有读过这本书,你会在你的程序教育生涯里存在一个严重裂沟,所以你应该立即挽救弥补!

可以这么说：GoF 设计模式是程序员真正掌握面向对象核心思想的必修课。虽然你可能已经通过了 SUN 的很多令人炫目的技术认证，但是如果你没有学习掌握 GoF 设计模式，只能说明你还是一个技工。

在浏览《Thingking in Java》(第一版)时，你是不是觉得好象这还是一本 Java 基础语言书籍？但又不纯粹是，因为这本书的作者将面向对象的思想巧妙的融合在 Java 的具体技术上，潜移默化的让你感觉到了一种新的语言和新的思想方式的诞生。

但是读完这本书，你对书中这些蕴含的思想也许需要一种更明晰更系统更透彻的了解和掌握，那么你就需要研读 GoF 的《设计模式》了。

《Thingking in Java》(第一版中文)是这样描述设计模式的：他在由 Gamma, Helm 和 Johnson Vlissides 简称 Gang of Four(四人帮),缩写 GoF 编著的《Design Patterns》一书中被定义成一个“里程碑”。事实上，那本书现在已成为几乎所有 OOP(面向对象程序设计)程序员都必备的参考书。(在国外是如此)。

GoF 的《设计模式》是所有面向对象语言(C++ Java C#)的基础，只不过不同的语言将之实现得更方便地使用。

GOF 的设计模式是一座"桥"

就 Java 语言体系来说，GOF 的设计模式是 Java 基础知识和 J2EE 框架知识之间一座隐性的"桥"。

会 Java 的人越来越多，但是一直徘徊在语言层次的程序员不在少数，真正掌握 Java 中接口或抽象类的应用不是很多，大家经常以那些技术只适合大型项目为由，避开或忽略它们，实际中，Java 的接口或抽象类是真正体现 Java 思想的核心所在，这些你都将在 GoF 的设计模式里领略到它们变幻无穷的魔力。

GoF 的设计模式表面上好象也是一种具体的"技术"，而且新的设计模式不断在出现，设计模式自有其自己的发展轨道，而这些好象和 J2EE .Net 等技术也无关！

实际上，GoF 的设计模式并不是一种具体"技术",它讲述的是思想，它不仅仅展示了接口或抽象类在实际案例中的灵活应用和智慧，让你能够真正掌握接口或抽象类的应用，从而在原来的 Java 语言基础上跃进一步，更重要的是，GoF 的设计模式反复向你强调一个宗旨：要让你的程序尽可能的可重用。

这其实在向一个极限挑战：软件需求变幻无穷，计划没有变化快，但是我们还是要寻找出不变的东西，并将它和变化的东西分离开来，这需要非常的智慧和经验。

而 GoF 的设计模式是在这方面开始探索的一块里程碑。

J2EE 等属于一种框架软件，什么是框架软件？它不同于我们以前接触的 Java API 等，那些属于 Toolkist(工具箱)，它不再被动的被使用，被调用，而是深刻的介入到一个领域中去，J2EE 等框架软件设计的目的是将一个领域中不变的东西先定义好，比如整体结构和一些主要职责(如数据库操作 事务跟踪 安全等)，剩余的就是变化的东西，针对这个领域中具体应用产生的具体不同的变化需求，而这些变化东西就是 J2EE 程序员所要做的。

由此可见，设计模式和 J2EE 在思想和动机上是一脉相承，只不过

1.设计模式更抽象，J2EE 是具体的产品代码，我们可以接触到，而设计模式在对每个应用时才会产生具体代码。

2.设计模式是比 J2EE 等框架软件更小的体系结构，J2EE 中许多具体程序都是应用设计模式来完成的，当你深入到 J2EE 的内部代码研究时，这点尤其明显，因此，如果你不具备设计模式的基础知识(GoF 的设计模式)，你很难快速的理解 J2EE。不能理解 J2EE,如何能灵活应用？

3.J2EE 只是适合企业计算应用的框架软件，但是 GoF 的设计模式几乎可以用于任何应用！因此 GoF 的设计模式应该是 J2EE 的重要理论基础之一。

所以说，GoF 的设计模式是 Java 基础知识和 J2EE 框架知识之间一座隐性的"桥"。为什么说隐性的？

GOF 的设计模式是一座隐性的"桥"

因为很多人没有注意到这点，学完 Java 基础语言就直接去学 J2EE,有的甚至鸭子赶架，直接使用起 Weblogic 等具体 J2EE 软件，一段时间下来，发现不过如此，挺简单好用，但是你真正理解 J2EE 了吗？你在具体案例中的应用是否也是在延伸 J2EE 的思想？

如果你不能很好的延伸 J2EE 的思想，那你岂非是大炮轰蚊子，认识到 J2EE 不是适合所有场合的人至少是明智的，但我们更需要将 J2EE 用对地方，那么只有理解 J2EE 此类框架软件的精髓，那么你才能真正灵活应用 Java 解决你的问题，甚至构架出你自己企业的框架来。(我们不能总是使用别人设定好的框架，为什么不能有我们自己的框架？)

因此，首先你必须掌握 GoF 的设计模式。虽然它是隐性，但不是可以越过的。

关于本站“设计模式”

Java 提供了丰富的 API，同时又有强大的数据库系统作底层支持，那么我们的编程似乎变成了类似积木的简单"拼凑"和调用，甚至有人提倡"蓝领程序员",这些都是对现代编程技术的不了解所至.

在真正可复用的面向对象编程中,GoF 的《设计模式》为我们提供了一套可复用的面向对象技术,再配合 Refactoring(重构方法),所以很少存在简单重复的工作,加上 Java 代码的精炼性和面向对象纯洁性(设计模式是 java 的灵魂),编程工作将变成一个让你时刻体验创造快感的激动人心的过程.

为能和大家能共同探讨"设计模式",我将自己在学习中的心得写下来,只是想帮助更多人更容易理解 GoF 的《设计模式》。由于原著都是以 C++为例，以 Java 为例的设计模式基本又都以图形应用为例,而我们更关心 Java 在中间件等服务器方面的应用,因此,本站所有实例都是非图形应用,并且顺带剖析 Jive 论坛系统.同时为降低理解难度，尽量避免使用 UML 图.

如果你有一定的面向对象编程经验,你会发现其中某些设计模式你已经无意识的使用过了;如果你是一个新手,那么从开始就培养自己良好的编程习惯(让你的的程序使用通用的模式,便于他人理解;让你自己减少重复性的编程工作),这无疑是成为一个优秀程序员的必备条件.

整个设计模式贯穿一个原理:面对接口编程，而不是面对实现.目标原则是:降低耦合,增强灵活性.

建筑和软件中模式之异同

CSDN 的透明特别推崇《建筑的永恒之道》,认为从中探寻到软件的永恒之道,并就"设计模式"写了专门文章《探寻软件的永恒之道 》,其中很多观点我看了很受启发,以前我也将"设计模式" 看成一个简单的解决方案,没有从一种高度来看待"设计模式"在软件中地位,下面是我自己的一些想法:

建筑和软件某些地方是可以来比喻的

特别是中国传统建筑,那是很讲模式的,这些都是传统文化使然,比如京剧 一招一式都有套路;中国画,也有套路,树应该怎么画法?有几种画法?艺术大家通常是创造出自己的套路,比如明末清初,水墨画法开始成熟,这时画树就不用勾勒这个模式了,而是一笔下去,浓淡几个叶子,待毛笔的水墨要干枯时,画一下树干,这样,一个活生写意的树就画出来.

我上面这些描述其实都是一种模式,创建模式的人是大师,但是拘泥于模式的人永远是工匠.

再回到传统建筑中,中国的传统建筑是过分注重模式了,所以建筑风格发展不大,基本分南北两派,大家有个感觉,旅游时,到南方,你发现古代名居建筑都差不多;北方由于受满人等少数民族的影响,在建筑色彩上有些与南方迥异,但是很多细节地方都差不多.这些都是模式的体现.

由于建筑受材料和功用以及费用的影响,所用模式种类不多,这点是和软件很大的不同.

正因为这点不同,导致建筑的管理模式和软件的管理模式就有很多不同，有些人认识不到这点,就产生了可以大量使用"软件蓝领"的想法,因为他羡慕建筑中"民工"的低成本.

要知道软件还有一个与建筑截然相反的责任和用途,那就是:现代社会中,计划感不上变化,竞争激烈,所有一切变幻莫测,要应付所有这些变化,首推信息技术中的软件,只有软件能够帮助人类去应付各种变化.而这点正好与建筑想反,建筑是不能帮助人类去应付变化的,(它自己反而要求稳固,老老实实帮助人遮风避雨,总不能叫人类在露天或树叶下打开电脑编软件吧).

软件要帮助人类去应付变化,这是软件的首要责任,所以,软件中模式产生的目的就和建筑不一样了,建筑中的模式产生可以因为很多原因:建筑大师的创意;材料的革新等;建筑中这些模式一旦产生,容易发生另外一个缺点,就是有时会阻碍建筑本身的发展,因为很多人会不思创造,反复使用老的模式进行设计,阻碍建筑的发展.

但是在软件中,这点正好相反,软件模式的产生是因为变化的东西太多,为减轻人类的负担,将一些不变的东西先用模式固化,这样让人类可以更加集中精力对付变化的东西,所以在软件中大量反复使用模式(我个人认为这样的软件就叫框架软件了,比如 J2EE),不但没阻碍软件的发展,反而是推动了软件的发展.因为其他使用这套软件的人就可以将更多精力集中在对付那些无法用模式的应用上来.

可以关于建筑和软件中的模式作用可以总结如下:

在软件中,模式是帮助人类向"变化"战斗,但是在软件中还需要和'变化'直接面对面战斗的武器:人的思维,特别是创造 分析思维等等，这些是软件真正的灵魂，这种思维可以说只要有实践需求(如有新项目)就要求发生，发生频度高，人类的创造或分析思维决定了软件的质量和特点。

而在建筑中,模式可以构成建筑全部知识，当有新的需求(如有新项目)，一般使用旧的模式都可以完成，因此对人类的创造以及分析思维不是每个项目都必须的，也不是非常重要的，对创造性的思维的需求只是属于锦上添花（除非人类以后离开地球居住了）。

设计模式之 Singleton(单态)

模式实战书籍《Java 实用系统开发指南》

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。

进一步深入可参考：

Double-checked locking and the Singleton pattern

When is a singleton not a singleton?

设计模式如何在具体项目中应用见《Java 实用系统开发指南》。

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。
```

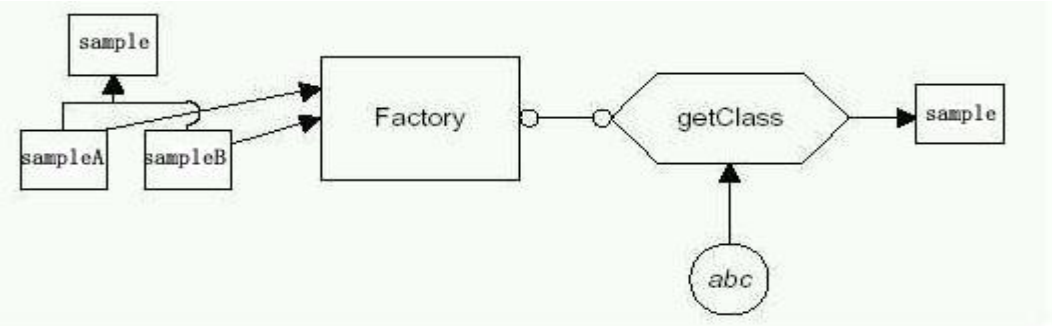
```
    if (which==1)
        return new SampleA();
    else if (which==2)
        return new SampleB();
    }
}
```

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

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 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)

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。

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),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

Sun 公司的 Explore the Dynamic Proxy API Dynamic Proxy Classes

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.

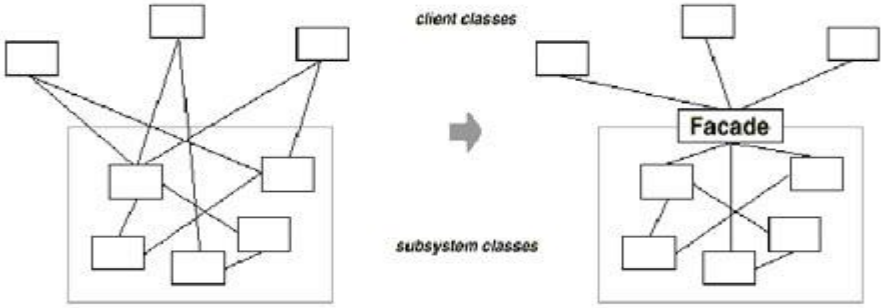
我们做成了一个 **Facade 接口**,使用该接口,上例中的程序就可以更改如下:

```
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.

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。

参考文章：

Composite 模式和树形结构的讨论

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 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 Flyweight(享元) FlyWeight 模式

Flyweight 模式定义:

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 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.

参考:

http://www.patterndepot.com/put/8/command.pdf

http://www.javaworld.com/javaworld/javatips/jw-javatip68.html

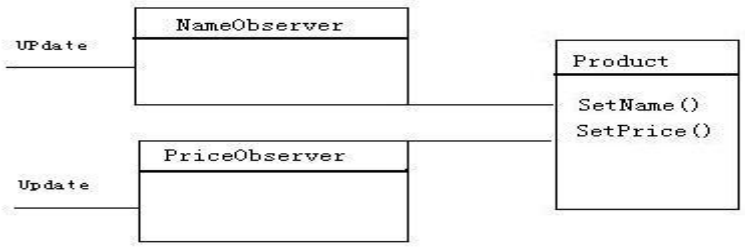
命令模式---我的理解

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 **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");
```

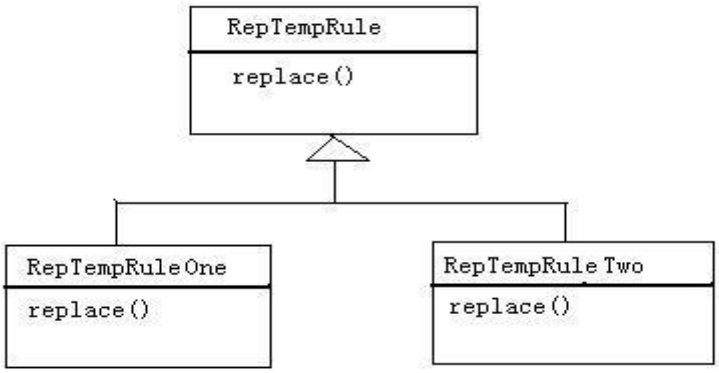
也许你以前还疑惑抽象类有什么用,现在你应该彻底明白了吧? 至于这样做的好处,很显然啊,扩展性强,以后 Benchmark 内容变化,我只要再做一个继承子类就可以,不必修改其他应用代码.

设计模式之 **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 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 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 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 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.

具体示例和代码见 **JavaWorld 的英文原文** , Javabean 表单输入特性参见我的另外一篇文章.

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 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**。.

数据库操作

数据库操作可以中 WEB 开发中最常用到的,很多 Java 开发工具都提供了自动的 Data bean WinZard.只要数据库建立好,相应的操作数据库的 Bean 就基本可以自动完成,本人使用 Jcreator 开发 bean,手工录入觉得也不是很麻烦的事情,下面我常用的数据库操作 bean,完全可以对付访问量不是很大的系统 ：

Mysql 类:

```
import java.sql.*;
import java.io.*;
/**
* 处理数据库的连接和访问
* @author sanware bqlr
```

```
* @version 1.01
*/
public class Mysql {
private Connection conn = null;
private Statement stmt = null;
private PreparedStatement prepstmt = null;
//这是一个全局类,里面放置数据库的参数,如数据库主机 访问用户名 密码等
private static BeansConstants CONST = BeansConstants.getInstance();
/**
* 构造数据库的连接和访问类
*/
public Mysql() throws Exception {
Class.forName(CONST.dbdriver);
conn = DriverManager.getConnection(CONST.dburl);
stmt = conn.createStatement();
}
public Mysql(String sql) throws Exception {
Class.forName(CONST.dbdriver);
conn = DriverManager.getConnection(CONST.dburl);
this.prepareStatement(sql);
}
/**
* 返回连接
* @return Connection 连接
*/
public Connection getConnection() {
return conn;
}
/**
* PreparedStatement
* @return sql 预设 SQL 语句
*/
public void prepareStatement(String sql) throws SQLException {
prepstmt = conn.prepareStatement(sql);
}
/**
* 设置对应值
* @param index 参数索引
* @param value 对应值
*/
public void setString(int index,String value) throws SQLException {
prepstmt.setString(index,value);
}
```

```
public void setInt(int index,int value) throws SQLException {
prepstmt.setInt(index,value);
}
public void setBoolean(int index,boolean value) throws SQLException {
prepstmt.setBoolean(index,value);
}
public void setDate(int index,Date value) throws SQLException {
prepstmt.setDate(index,value);
}
public void setLong(int index,long value) throws SQLException {
prepstmt.setLong(index,value);
}
public void setFloat(int index,float value) throws SQLException {
prepstmt.setFloat(index,value);
}
//File file = new File("test/data.txt");
//int fileLength = file.length();
//InputStream fin = new java.io.FileInputStream(file);
//mysql.setBinaryStream(5,fin,fileLength);
public void setBinaryStream(int index,InputStream in,int length) throws SQLException {
prepstmt.setBinaryStream(index,in,length);
}
public void clearParameters()
throws SQLException
{
prepstmt.clearParameters();
}
/**
* 返回预设状态
*/
public PreparedStatement getPreparedStatement() {
return prepstmt;
}
/**
* 返回状态
* @return Statement 状态
*/
public Statement getStatement() {
return stmt;
}
/**
* 执行 SQL 语句返回字段集
* @param sql SQL 语句
```

```
* @return ResultSet 字段集
*/
public ResultSet executeQuery(String sql) throws SQLException {
if (stmt != null) {
return stmt.executeQuery(sql);
}
else return null;
}
public ResultSet executeQuery() throws SQLException {
if (prepstmt != null) {
return prepstmt.executeQuery();
}
else return null;
}
/**
* 执行 SQL 语句
* @param sql SQL 语句
*/
public void executeUpdate(String sql) throws SQLException {
if (stmt != null)
stmt.executeUpdate(sql);
}
public void executeUpdate() throws SQLException {
if (prepstmt != null)
prepstmt.executeUpdate();
}
/**
* 关闭连接
*/
public void close() throws Exception {
if (stmt != null) {
stmt.close();
stmt = null;
}
if (prepstmt != null) {
prepstmt.close();
prepstmt = null;
}
conn.close();
conn = null;
}
}
```

Mysql 建立好后,以后涉及数据库的操作,只要对象化 Mysql 就可以:

```
private String page_navlink_insert="insert into page_navlink values (?,?,?,?)";
public void insertnavlink() throws Exception
{
    ResultSet rs=null;
    try {
        Mysql mysql = new Mysql(page_navlink_insert);
        mysql.setInt(1,this.siteid);
        mysql.setInt(2,this.pageid);
        mysql.setString(3,this.navlinkname);
        mysql.setString(4,this.pagefile);
        mysql.executeUpdate();
        mysql.close();
        mysql = null;
    } catch (Exception ex) {
        throw new Exception("insertnavlink()"+ex.getMessage());
    }
}
```

在 Jsp 中,就可以直接使用一句语句使用 insertnavlink()了:

```
<jsp:useBean id="NAV" scope="session" class="mysite.Navlink" />
<jsp:setProperty name="NAV" property="*" />
.....
NAV.insertnavlink();
.....
```

频繁访问数据库,需要使用连接池,在 tomcat 4.0 中配置 JNDI,稍微修改一下上面程序就可使用连接池.Tomcat 的连接池配置和 J2EE 类似,因此程序不用修改,也可直接运行在 J2EE 上.

也可以使用第三方连接池, 如很有名的 Poolman.

Composite 模式和树形结构的讨论

最近在使用 Jdon 框架做 JiveJdon3，碰到一个老问题，主题帖和回帖之间是树形结构关系，现在碰到两种方案：

1. 采取 Composite 模式封装复杂的树形结构，这样外界要访问一个主题贴需要树形遍历时，由 Composite 内部来树形算法来遍历，外界只要告诉要怎样的结果即可。

优点：封装了树形结构的遍历算法，外界客户端无需自行进行帖子的树形结构相关判断代码。

缺点：对树形结构的操作有各种各样，而且以后可能有新增新的操作，这可能涉及修改 Composite 模式的 Component 接口。

Composite 模式：http://www.jdon.com/designpatterns/composite.htm

这种采取 Composite 模式封装树形结构遍历的方法，为防止新增新的操作改动接口，可引入访问者模式，下面这篇文章谈到了用 C++实现 Composite 模式+Visitor 模式解决树形结构封装算法:

http://members.home.nl/r.f.pels/articles/misc/C++PatternsAndTreesSeparatingKnowledge.html

它主要讨论了如何分离树形结构的不同类的对象和用户界面的树形结构表现这两个方面。

2.不采取 Composite 模式进行树形结构封装，就象原来 Jive 一样，给每个主题贴提供一个 TreeWalker 对象，这样外界如果要遍历这个主题贴时，自己通过 TreeWalker 去遍历树形结构。

优点：比较符合模型编程，ForumThread 主题贴是一个模型，里面提供一个自身树形结构遍历，非常开放，外界客户端有新的操作，只要得到模型 ForumThread 就可以了。

缺点：TreeWalker 提供的树形结构遍历方法有限，有可能不能满足新的操作或新的功能的需要。

有过这方面考虑的人可一同讨论一下。

Re: Composite 模式和树形结构的讨论 发表: 2005-11-23 14:38 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

这条题目其实应该算非常难的一道题，涉及到 GoF 模式中两个较不常用的模式，而且又解决了日常编程中经常碰到的对象树形结构问题，请真正大牛的人来解决这个实际问题吧。我原意洗耳恭听。是显示您真正水平的时候了。

Re: Composite 模式和树形结构的讨论 发表: 2005-11-25 16:56 回复

whatavery 发表文章: 14/ 注册时间: 2005-11-25 16:17

在这里回复是不是也太不知天高地厚了，因为和 banq 老师的水平差的不是一点半点了。
我认为 TreeWalker 是优于 Composite，我在用 Composite 的时候更倾向于那种隐性的树模式。至使连有些客户并在 UI 上并看不出来。但是这种替代结构我一直不敢特别肯定他的整和/扩展程度，而需要结合其他模式如 Visitor，Adapter 等来扩展。
所谓替代结构，如
Polymorphism 把 if/switch Refactoring 掉
Iterator 把 for Refactoring 掉
Composite 把纯 tree Refactoring 掉
等等
对于这些很纯的 tree，我认为类似于 TreeModel/DefaultTreeModel 这种封装型的才是最好扩展的。而 Composite 在增加系统复杂度的情况下实现，表面上看结构是清楚的，但效率和扩展度，都没直接的 tree 封装来的痛快。
因为和 banq 老师差的水平太多，所以只提供一个投票性的意见，我是第一次发帖子。谢谢 banq 老师写了那么多精彩的文章，其实我回帖不是目的，只是来景仰一下 banq 老师

Re: Composite 模式和树形结构的讨论 发表: 2005-11-26 20:40 回复

鲁中正气 发表文章: 50/ 注册时间: 2005-06-13 17:33

所谓替代结构，如
Iterator 把 for Refactoring 掉
Composite 把纯 tree Refactoring 掉
好！好！口诀阿。
不过，Polymorphism 把 if/switch Refactoring 掉
除了 Polymorphism 还可以用其他方式吧

Re: Composite 模式和树形结构的讨论 发表: 2005-11-29 16:10 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

whatavery 这方面是有一定研究。
TreeWalker/TreeModel 和 Composite 相比，要简单直接多。
Composite 模式是一种内敛式样或者趋向闭合的模式，该模式是将以前在客户端中实现的代码强制收回，这对一个 Open 开放结构有所伤害。
因此，一般使用 Composite 模式都要结合 Visitor 模式，但是，
使用 Composite+Visitor 模式带来缺点是：对访问者进行了一定限制，例如可能只是需要一个帖子集合，必须将将这段代码变成一个访问者类。
但是，如果单纯使用 TreeWalker/TreeModel，TreeModel 中只封装了常见的树操作，万一有新的特殊树操作，需要更改 TreeModel 接口，而且 TreeWalker 客户端会拥有自己对树结构遍历的代码，需要增加新功能或维护时，都需要维护者对

这个树了解，可维护和可拓展差。

现在，我采用的方案是：TreeWalker/TreeModel 和 Composite+Visitor 两个结合，我通过 TreeWalkerService 以服务形式向外提供常见的树操作，而 TreeWalkerService 内部的实现是通过 Composite+Visitor 实现的。当在组件层以后需要有新的树结构操作时，使用 Composite+Visitor 实现。

也就是说：Composite+Visitor 在组件层封装了树形结构，这样在组件层各处就没有零星散落的涉及树形结构操作的代码，如果你需要将这组树形结构数据转化为 XML，那么做一个 Visitor 方法即可。如果在表现层需要操作树形结构，只能通过 TreeWalkerService 了。

这样做到通用和定制，简单和复杂实现一定的统一。

以上只是个人观点，欢迎讨论。

Re: Composite 模式和树形结构的讨论 发表: 2005-11-30 18:26 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

在 IBM 发现早在 2002 年就有老外进行这方面研究，并且提出深度优先访问器，这些估计在 Junit 中得到了很好的实现。

文章连接:深度优先访问器和中断的分派

看来，使用 Composite+Visitor 对树的访问是不错，问题难题还是如何在 Model/Service 这样数据模型的构件系统中实现啊。

对于一个数据模型，我们使用 parentID 和 currentID 两个字段表示这同一个数据模型不同对象之间的父子关系；而使用 Composite 模式，两个重要的子类，Leaf（叶）和 Branch（枝）必须首先确定。

这两者之间就存在一定衔接距离，衔接的功能是：必须根据 parentID 和 currentID 这个简单的信息，找出一颗树中的 Leaf 和 Branch，然后装入 Composite 供 Visitor 访问。

这个衔接功能实际是树的算法，我们需要在内存中（缓存）中首先根据 parentID 和 currentID 建立一颗完整的树，这样就能从这个树中获知 leaf 和 Branch，从而使用 Composite 实现树的操作方法封装。

这是一点思考，望能获得 whatavery 等高手交流指导。

Re: Composite 模式和树形结构的讨论 发表: 2005-12-01 01:13 回复

whatavery 发表文章: 14/ 注册时间: 2005-11-25 16:17

看来 banq 老师对 Composite+Visitor 还是很欣赏的，感觉 banq 老师肯定倾向于用 Composite+Visitor 模式的。

但 banq 老师实在是对 Composite 要求太高了，Composite 是实现了递归树的模型，但我理解老师想要把 Composite 当节点用，这肯定无法满足需求了。

（因为水平和 banq 老师相去太远，理解老师的意思可能不到位，希望老师见谅）老师的意思要是细化到节点，对节点进行封装，从而跳出 Composite 的约束。节点无非三种：Root，Branch，Leaf。但都可以归结为 Branch（Root，Leaf 视为特殊节点），这样就可以细化到节点了，而节点对象需要有一个有 ID 到另一 Branch 对象的映射过程的对象，所以已经产生两个对象：1，Branch 对象 2，Finder 对象。

Branch 对象，持有自身的 parentID 和 currentID，聚合 Finder 对象。

Finder 对象，返回 Branch 对象的 collection（子），或返回 Branch 对象（父）。

且 Finder 对象自身递归。

这样就由递归树的模型细化到了递归节点的模型。

老师对 Composite 要求过于严格了。

对于效率我一般倾向于 primitive，所以我觉得这个模型 ID-对象-ID-映射-对象 也许可行。

banq 老师，我说实话，我发的帖子真向您表示尊敬，至于讨论，我还不够资格。

呵呵，至于看贴的高手就不要笑话我了，我只是初学者。

Re: Composite 模式和树形结构的讨论 发表: 2005-12-01 10:22 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

whatavery 太谦虚了，能有人一起讨论这样一个吃力不讨好的设计非常难得。
你只使用两个对象抽象：Branch 对象 和 Finder 对象 是一种思路，非常不错，有时间我会按照这种思路仔细考虑一下。
我目前思考方向是这样的：根据 Composite 模式定义，需要两个重要子类，第一是复合类 Composite，相当于 Branch，另外一个是非复合类，单个对象，相当于 Leaf，所以，我的代码非常类似"深度优先访问器和中断的分派"一文中"清单 1．带访问器的二叉树"的代码。
在 Jive 中有一个 LongTree，它使用 leftChildren 和 rightSiblings 两个 primitive 数组来保存 Branch 二叉树的特点，然后它根据这两个数组可以得到任何一个节点的子集合，相当于实现你的 Finder 对象功能。
我就使用 LongTree 的这两个数组算法，生成 Composite 模式需要的 Leaf 和 Branch 对象，LongTree 使用这个两个数组是直接实现 TreeWalker（TreeModel），我是用来既实现 treeWalker，又用来实现 Composite+Visitor。

Re: Composite 模式和树形结构的讨论 发表: 2005-12-01 16:07 回复

whatavery 发表文章: 14/ 注册时间: 2005-11-25 16:17

banq 老师对 LongTree 到 Composite 过程中，使不使用 Strategy/Builder 有什么看法?

Re: Composite 模式和树形结构的讨论 发表: 2005-12-01 16:30 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

说的好，Strategy/Builder 都要使用，首先内存中要建立一个二叉树数据结构，这里使用到 Builder 模式，我将 longTree 的 add 方法分解到 Builder 中。
从这个二叉树中延伸出两种算法策略 Strategy：一个遍历提供 getChildern 等方法 treeWalker；另外一个是 Leaf 和 Branch 节点的获得，奠定 Composite 基础。
设计时，我们先穷尽其复杂性，到真正实现时，简化一下，避免让人眼花缭乱。
我现在总结一下通过 Composite+Visitor 访问的具体功能：删除帖子、显示树形结构帖子、将帖子转为 XML 格式。
目前这三个功能方法使用 Composite+Visitor 好像显示不太多好处，如果有更多树形操作代码，就能显示出来。

Re: Composite 模式和树形结构的讨论 发表: 2005-12-02 11:34 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

其实我们讨论到现在，相当于已经把一个普通通用常用的问题用模式语言表达出来，据此可以形成一套解决树形结构访问的应用框架了。最重要的是：树形结构问题基本方向搞清楚，以后再碰到此类问题，可以向这个方向去再深入研究。
模式语言相关的连接地址：
模式采掘、体系结构设计和应用框架开发

Re: Composite 模式和树形结构的讨论 发表: 2005-12-02 11:41 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

另外我想说的是：本主题涉及到数据结构+模式语言的解决方案，可是我们高等教育中只讲数据结构，不讲模式语言，导致出现大量左撇子，重视数据结构 数据库，不重视软件可维护性 可拓展性和重用性。这也是我曾经提出“设计模式”应该是程序基础教育一部分的原因。
另外一个对模式疑问的讨论：

> 懂模式又咋样？不还是一个技工吗？

命令模式（我的理解）

命令模式（我的理解）

前言

第一章：通常的命令模式

第二章：简化的命令模式

第三章：其他要说的内容

前言

以下是我对命令模式的理解。可能和很多其他文章讲述的不太一样。经过我理解加工的。供大家参考！学艺不精，并且写的比较仓促，还请大家指教。

通常的命令模式：

1．1 通常命令模式有一下几个角色

调用者：（命令的执行者）

生成有序的命令队列

按顺序执行命令操作

提供撤销命令操作

记录已经操作的命令

抽象命令：

抽象的命令接口

具体命令：

具体的命令。

由三个要素组成：执行者，执行者要作的操作和被执行的对象组成。当然还可以有其他，比如将对象执行成什么结果。例如：调用 Mypait 类(执行者)将 My rectangle(对象)填充（操作）为红色（结果）。这样就可以完全描述一个命令了。

执行者：

真正执行逻辑操作的对象

1．2 原型：

```
//调用者
public class Invoker{
List commands; //命令集合
public void setCommands(List commands){
this.commands = commands;
}
public void addCommand (Command command,int i){
commands.add(i,command);
}
public void removeCommand (int i){
commands.add(i,command);
}
//得代执行命令
public void action(){
for(Iterator it = list.iterator();it.hasNext();){
Command command = Command) it.next();
Command. execute();
```

```
}
}
……
//还可以有丰富的 redo 和 undo 操作；(当然一些都给基于命令类提供的相应方法)
}
//抽象命令
abstract class Command
{
abstract public void execute();
abstract public void unexecute();
abstract public void reexecute();
//一般有这样这个方法，根据需要可以增删
}
// 具体的命令类 1:写作命令，选择一个作者（Author 类实例对象），让他写作（调用它的 write 方法）写作的对象是书（Book 的实例对象）形成了一个写作的命令，写作的对象是 Book 的实例
public class WriteCommand implement Command
{
Author author; //执行者
Book book; //要执行的对象
public WriteCommand (Author author,Book book) {
this. author = author;
this. book = book;
}
// 在这里执行要执行的操作
public override void Execute()
{
author.write (book);
}
}
// 具体的命令类 2：出版命令，选择一个出版社（publisher 类实例对象），让他出版书（调用它的 publisherBook 方法）出版的对象是书（Book 的实例对象）形成了一个出版的命令
public class PublishCommand implement Command
{
Publisher publisher;
Book book;
public PublishCommand (Publisher publisher) {
this. publisher = publisher;
this. book = book;
}
// Methods
public override void Execute()
{
publisher. publisherBook(book);
}
```

}

// Publisher 和 Author 类为执行者

略

这样我们的客户端代码就可以这样写:

//如果我要出一本书

//一本空白的书

Book book = new Book();

//先找一个作者和出版社

Author author = new Author();

Publisher publisher = new Publisher ();

//产生命令集合

Command writeCommand = new WriteCommand (author,book);

Command publishCommand = new PublishCommand(publisher,book);

List commands = new List ();

Commands.add(writeCommand);

//找个调用者，把命令给它，让他来根据命令协调工作

Invoker invoker = new invoker();

Invoker.setCommands(commands);

public void addCommand (Command command,int i){

commands.add(i,command);

}

invoker.action();

特点：

1》 分布登记统一执行：

在作程序时，经常碰到一些需求，先注册一些操作，并不马上执行，等最终确定后统一执行。如一个具体的例子:用户定制自己的报表，可以订阅饼，柱，折线，曲线图，客户选择相应的报表组合，这样对应一个命令集合，在没确定之前用户可以增删这些报表（命令），等最终确定统一交给调用者根据命令执行，生成组合报表。实现了命令分布提出，确定后统一执行的功能。

2》形如流水线操作：还是出书的例子

//先是一本空白的书：

Book book = new Book();

//找几个作者

Author author1 = new Author();

Author author2 = new Author();

//把写 1，2 章的名类分别给这两个作者

Command writeCommand = new Write1Command (author1，book);

Command writeCommand = new Write2Command (author2，book);

List commands = new List ();

Commands.add(writeCommand);

//调用者

Invoker invoker = new invoker();

Invoker.setCommands(commands);

//流水写书

invoker.action()

实际上在 aciton 这一方法中，invoker 按照命令，让两个作者流水写作这本书。（类似一个书的流水线加工工厂）

这样我们的书就被流水加工成功（当然这本书只有两章）

这样就给了我们一种系统设计的框架，

模型＋工具＋命令

客户端产生命令，命令调用工具操作模型。

Book 相当于模型

Author 相当于和多工具类中的一个

Command 命令

3》系统需要支持命令的撤消(undo)。提供 redo()方法

我们可以和容易的加入 undo 和 redo，这个不难理解

4》在 Invoker 中我们可以实现跟踪，和日志。

5》当系统需要为某项复制增加形的功能的时候，命令模式使新的功能（表现为一种命令）很容易地被加入到服务种里。

命令联系了工具类即执行类和系统逻辑，

简化/变化的命令模式：

命令模式的角色比较多，在实际应用种我们可以根据所需要的功能和不需要的功能加以简化。

1》去掉 调用者

产生命令集合后，我们可以直接在 client 中迭代执行执行操作

2》 变化 调用者 成为 跟踪者

```
//调用者
public class Invoker{
List commands; //已经执行完毕的命令集合
public void addCommand (Command command,int i){
commands.add(i,command);
}
public void action(Command command){
//执行操作
command. execute();
//
commands.add(command);
}
}
……
//还可以有丰富的 redo 和 undo 操作；(当然一些都给基于命令类提供的相应方法)
}
```

这样这个类就记录了所有执行过的操作。

3》去掉 命令 用 map 替代

我们完全可以用 map 代替命令，这样无需定义各种命令类

我们改进例子

```
Author author ＝ new Author();
Publisher publisher ＝ new Publisher ();
Map m = new HashMap;
m.put(author, write);
m.put(author, publisherBook);
```

在 Invoker 的 action 方法：

得代 map

运用 java 反射来调用方法；

4》去掉执行者：

直接在命令中（execute 方法种）加业务逻辑。这样只适合于简单的小的系统.

其他要说的内容

1》 将某些参数传给某个方发的方式很多，除了当作方法的参数外还可以当作类的成员便俩变量传入：

这就为命令的抽象带来了极大的方便

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

当我们已经有了执行者（类 Test）方法 execute（args1，args2 ….argsn）

我们不必向 Command 加入 execute（args1，args2 ….argsn）抽象方法，在说即使加了，在我们迭代的时候也无法判断或十分不容易判断哪个命令调用哪个 execute 方法。

那么我们可以这样

```
class ConcreteCommand：Command
{
Test test;
args1
args2
…..
argsn
public override void Execute()
{
test. execute (args1，args2 ….argsn);
}
}
```

2》 在想跟踪操作的时候，一般为每一个操作对象分配一个调用者，操作对象在调用者中设置。（可以抽象出一个总的调用者，来协调调用每一个具体的调用者）

3》 命令的抽象粒度我觉得是要注意的。

4》 理解思想，不要机械的照搬。消化成自己的，加以灵活的运用和创造在是根本出路。

所谓命令模式的根本思想就是在 先形成命令，在根据命令执行。

Re: 命令模式（我的理解） 发表: 2006-04-13 15:56 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

不错，实际是一个群命令的使用，单个命令模式粒度太小，让人难以把握，但是多个命令群集合在一起就感觉命令模式的作用。

该文群命令一种用法(流水线操作)实际展示了命令模式+职责链的使用，或者说命令组成的 Filter 过滤器模式，这种方式可以重构 if else 逻辑。

使用 Map 的群命令也是常用的一种过滤器实现，这种方式性能好，也可以重构原来的 if else 逻辑。

可结合本人的"**你还在用 if else 吗？**":

Re: 命令模式（我的理解） 发表: 2006-04-14 17:05 回复

xyz 发表文章: 26/ 注册时间: 2005-08-22 20:17

我觉得命令+组合+责任链比较好用，组合模式的加入，可以使得我们的命令可以随意组合，重用！

你还在用 if else 吗？

面向过程设计和面向对象设计的主要区别是：是否在业务逻辑层使用冗长的 if else 判断。如果你还在大量使用 if else，当然，界面表现层除外，即使你使用 Java/C#这样完全面向对象的语言，也只能说明你的思维停留在传统的面向过程语言上。

传统思维习惯分析

为什么会业务逻辑层使用 if else，其实使用者的目的也是为了重用，但是这是面向过程编程的重用，程序员只看到代码重用，因为他看到 if else 几种情况下大部分代码都是重复的，只有个别不同，因此使用 if else 可以避免重复代码，并且认为这是模板 Template 模式。

他范的错误是：程序员只从代码运行顺序这个方向来看待它的代码，这种思维类似水管或串行电路，水沿着水管流动（代码运行次序），当遇到几个分管（子管），就分到这几个分管子在流动，这里就相当于碰到代码的 if else 处了。

而使用 OO，则首先打破这个代码由上向下顺序等同于运行时的先后循序这个规律，代码结构不由执行循序决定，由什么决定呢？由 OO 设计；设计模式会取代这些 if else，但是最后总是由一个 Service 等总类按照运行顺序组装这些 OO 模块，只有一处，这处可包含事务，一般就是 Service，EJB 中是 Session bean。

一旦需求变化，我们更多的可能是 Service 中各个 OO 模块，甚至是只改动 Service 中的 OO 模块执行顺序就能符合需求。

这里我们也看到 OO 分离的思路，将以前过程语言的一个 Main 函数彻底分解，将运行顺序与代码其他逻辑分离开来，而不是象面向过程那样混乱在一起。所以有人感慨，OO 也是要顺序的，这是肯定的，关键是运行顺序要单独分离出来。

是否有 if else 可以看出你有没有将运行顺序分离到家。

设计模式的切入口

经常有人反映，设计模式是不错，但是我很难用到，其实如果你使用 if else 来写代码时（除显示控制以外），就是在写业务逻辑，只不过使用简单的判断语句来作为现实情况的替代者。

还是以大家熟悉的论坛帖子为例子，如 ForumMessage 是一个模型，但是实际中帖子分两种性质：主题贴（第一个根贴）和回帖（回以前帖子的帖子），这里有一个朴素的解决方案：

建立一个 ForumMessage，然后在 ForumMessage 加入 isTopic 这样判断语句，注意，你这里一个简单属性的判断引入，可能导致你的程序其他地方到处存在 if else 的判断。

如果我们改用另外一种分析实现思路，以对象化概念看待，实际中有主题贴和回帖，就是两种对象，但是这两种对象大部分是一致的，因此，我将 ForumMessage 设为表达主题贴；然后创建一个继承 ForumMessage 的子类 ForumMessageReply 作为回帖，这样，我在程序地方，如 Service 中，我已经确定这个 Model 是回帖了，我就直接下溯为 ForumMessageReply 即可，这个有点类似向 Collection 放入对象和取出时的强制类型转换。通过这个手段我消灭了以后程序中 if else 的判断语句出现可能。

从这里体现了，如果分析方向错误，也会导致误用模式。

讨论设计模式举例，不能没有业务上下文场景的案例，否则无法决定是否该用模式，下面举两个对比的例子：

第一. 这个帖子中举例的第一个代码案例是没有上下文的，文中只说明有一段代码：

```
main() {
if（case A）{
//do with strategy A
}else(case B){
//do with strategy B
}else(case C){
//do with strategy C
}
}
```

这段代码只是纯粹的代码，没有业务功能，所以，在这种情况下，我们就很难确定使用什么模式，就是一定用策略模式等，也逃不过还是使用 if else 的命运，设计模式不是魔法，不能将一段毫无意义的代码变得简单了，只能将其体现的业务功能更加容易可拓展了。

第二.在这个帖子中，作者举了一个 PacketParser 业务案例，这段代码是体现业务功能的，是一个数据包的分析，作者也比较了各种模式使用的不同，所以我们还是使用动态代理模式或 Command 模式来消灭那些可能存在的 if else

由以上两个案例表明：业务逻辑是我们使用设计模式的切入点，而在分解业务逻辑时，我们习惯则可能使用 if else 来实现，当你有这种企图或者已经实现代码了，那么就应该考虑是否需要重构 Refactoring 了。

if else 替代者

那么实战中，哪些设计模式可以替代 if else 呢？其实 GoF 设计模式都可以用来替代 if else，我们分别描述如下：

- 状态模式

当数据对象存在各种可能性的状态，而且这种状态将会影响到不同业务结果时，那么我们就应该考虑是否使用状态模式，当然，使用状态模式之前，你必须首先有内存状态这个概念，而不是数据库概念，因为在传统的面向过程的/面向数据库的系统中，你很难发现状态的，从数据库中读取某个值，然后根据这个值进行代码运行分流，这是很多初学者常干的事情。参考文章：状态对象：数据库的替代者

使用传统语言思维的情况还有：使用一个类整数变量标识状态：

```
 public class Order{
private int status;
//说明：
//status=1 表示订货但为查看 ；
//status=2 表示已经查看未处理；
//status=3 表示已经处理未付款
//status=4 表示已经付款未发货
//status=5 表示已经发货
}
```

上述类设计，无疑是将类作为传统语言的函数来使用，这样导致程序代码中存在大量的 if else。

- 策略模式

当你面临几种算法或者公式选择时，可以考虑策略模式，传统过程语言情况是：从数据库中读取算法数值，数值 1 表示策略 1，例如保存到数据库；数值为 2 表示策略 2，例如保存到 XMl 文件中。这里使用 if else 作为策略选择的开关。

- command 模式

传统过程的思维情况是：如果客户端发出代号是 1 或"A"，那么我调用 A.java 这个对象来处理；如果代号是 2 或"B"，我就调用 B.java 来处理，通过 if else 来判断客户端发送过来的代码，然后按事先约定的对应表，调用相应的类来处理。

- MVC 模式

MVC 模式的传统语言误用和 Command 模式类似，在一个 Action 类中，使用 if else 进行前后台调度，如果客户端传送什么命令；我就调用后台什么结果；如果后台处理什么结构，再决定推什么页面，不过，现在我们使用 Struts/JSF 这样 MVC 模式的框架实现者就不必范这种低级错误。

- 职责链模式

职责链模式和 Command 模式是可选的，如果你实在不知道客户端会发出什么代号；也没有一个事先定义好的对照表，那么你只能编写一个个类去碰运气一样打开这个包看一下就可以。与 Command 是不同在 AOP vs Decorator 一文中有分析。

- 代理或动态代理模式

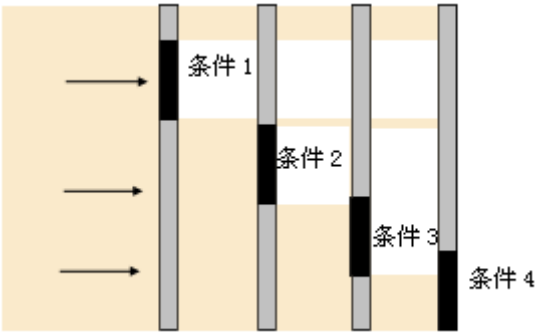
代理对象可以是符合某种条件的代表者，比如，权限检验，传统面向过程思维是：当一个用户登陆后，访问某资源时，使用 if else 进行判断，只有某种条件符合时，才能允许访问，这样权限判断和业务数据逻辑混乱在一起，使用代理模式可以清晰分离，如果嫌不太好，使用动态代理，或者下面 AOP 等方式。

- AOP 或 Decorator 模式

其实使用 filter 过滤器也可以替代我们业务中的 if else，过滤器起到一种过滤和筛选作用，将符合本过滤器条件的对象拦

截下来做某件事情，这就是一个过滤器的功能，多个过滤器组合在一起实际就是 if else 的组合。

所以，如果你实在想不出什么办法，可以使用过滤器，将过滤器看成防火墙就比较好理解，当客户端有一个请求时，经过不同性质的防火墙，这个防火墙是拦截端口的；那个防火墙是安全检查拦截等等。过滤器也如同红蓝白各种光滤镜；红色滤镜只能将通过光线中的红色拦截了；蓝色滤镜将光线中的蓝色拦截下来，这实际上是对光线使用 if else 进行分解。



如图，通过一个个条件过滤器我们立体地实现了对信号的分离，如果你使用 if else，说明你是将图中的条件 1/2/3/4 合并在一起，在同一个地方实现条件判断。

需要深入了解过滤器的实现细节和微小区别，请参考文章：AOP vs Decorator

OO 设计的总结

还有一种伪模式，虽然使用了状态等模式，但是在模式内部实质还是使用 if else 或 switch 进行状态切换或重要条件判断，那么无疑说明还需要进一步努力。更重要的是，不能以模式自居，而且出书示人。

真正掌握面向对象这些思想是一件困难的事情，目前有各种属于揪着自己头发向上拔的解说，都是误人子弟的，所以我觉得初学者读 Thinking in Java（Java 编程思想）是没有用，它试图从语言层次来讲 OO 编程思想，非常失败，作为语言参考书可以，但是作为 Java 体现的 OO 思想的学习资料，就错了。

OO 编程思想是一种方法论，方法论如果没有应用比较，是无法体会这个方法论的特点的，禅是古代一个方法论，悟禅是靠挑水砍柴这些应用才能体会。

那么 OO 思想靠什么应用能够体会到了？是 GoF 设计模式，GoF 设计模式是等于软件人员的挑水砍柴等基本活，所以，如果一个程序员连基本活都不会，他何以自居 OO 程序员？从事 OO 专业设计编程这个工作，如果不掌握设计模式基本功，就象一个做和尚的人不愿意挑水砍柴，他何以立足这个行业？早就被师傅赶下山。

最后总结：将 if else 用在小地方还可以，如简单的数值判断；但是如果按照你的传统习惯思维，在实现业务功能时也使用 if else，那么说明你的思维可能需要重塑，你的编程经验越丰富，传统过程思维模式就容易根深蒂固，想靠自己改变很困难；建议接受专业头脑风暴培训。

用一句话总结：如果你做了不少系统，很久没有使用 if else 了，那么说明你可能真正进入 OO 设计的境地了。（这是本人自己发明的实战性的衡量考核标准）。

从工作流状态机实践中总结状态模式使用心得

状态模式好像是很简单的模式，正因为状态好像是个简单的对象，想复杂化实现设计模式就不是容易，误用情况很多。

我个人曾经设计过一个大型游戏系统的游戏状态机，游戏状态可以说是游戏设计的主要架构，但是由于系统过分复杂和时间仓促，并没有真正实现状态模式。

目前在实现一个电子政务项目中，需要进行流程状态变化，在电子政务设计中，我发现，如果一开始完全按照工作流规范开发，难度很大，它和具体项目实践结合无法把握，而且工作流规范现在有 wfmc，还有 bpml，选择也比较难。因此，我决定走自创的中间道路。

因为，我需要做一个状态机 API，或者说状态机框架，供具体系统调用：类如公文流转应用或信息发报送应用等。

好的状态模式必须做到两点：

1. 状态变化必须从外界其它逻辑划分出来。
2. 状态必须可方便拓展，对其它代码影响非常小。

要做到这两点，必须先明确状态变化机制，状态变化实际是由 Event 事件驱动的，可以认为是 Event-condition-State，在 MVC 模式一般是 Event-condition-Action 实现。状态模式需要封装的是 Event-condition-State 中的 condition-State 部分。

清晰理解状态和流程的关系也非常重要，因为状态不是孤立的，可以说和流程是点和线的关系，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。因此，必须明白使用状态模式实现状态机实际是为了更好地表达和说明流程。

状态和流程以及事件的关系如下：

|Event

___currentState__|_____newState___

图中表示了是事件改变了流程的状态，在业务逻辑中，经常发生的是事件，如果不使用状态模式，需要在很多业务逻辑处实现事件到状态判定和转换，这有很多危险性。

最大的危险是系统没有一个一抓就灵的主体结构，以那个游戏系统为例，在没有状态模式对状态提炼的情况下，状态改变由每个程序员想当然实现，导致每个程序员开发的功能在整合时就无法调试，因为这个程序员可能不知道那个程序员的代码在什么运行条件下改变了游戏状态，结果导致自己的代码无法运行。

这种现象实际上拒绝了项目管理的协作性，大大地拖延项目进度（程序员之间要反复商量讨论对方代码设计）。从这一点也说明，一个好的架构设计是一个项目快速成功完成的基础技术保证，没有这个技术基础，再先进的项目管理手段也是没有效率的，或者是笨拙的。

状态模式对于很多系统来说，确实是架构组成一个重要部分。

下面继续讨论如何实现一个好的状态模式，为了实现好的状态模式，必须在状态模式中封装下面两个部分：

1. 状态转换规则（行为）

2. 状态属性（属性）

状态转换行为有两种划分标准：

1. run 和 next 两个行为，run 是当前状态运行行为，next 是指在 Event 参与下，几种可能转向的下一个状态。

2. stateEnter 和 stateExit， 状态进入和状态退出。

如果用进入一个个房间来表示状态流程的话， 第一种分析是只重视着“在房间里”和“如何转入下一个房间”，这两种行为一旦确定，可以被反复使用，进而一个流程的状态切换可以全部表达出来。

第二中分析方法有所区别，只重视进入房间和离开房间这个两个行为，同样，这种模型也可以被反复利用在其它房间，一个流程的状态切换也可以全部表达出来。

具体选择取决于你的需求，比如，如果你在进入一个状态开始，要做很多初始化工作，那么第二种模型就很适合。

状态变化都离不开一个主体对象，主体对象可以说包含状态变化（行为）和状态属性（属性），假设为 StateOwner，StateOwner 有两个部分组成：Task/事情和状态。任何一个 Task/事情都会对应一个状态。

这样，我们已经抽象出两个主要对象：状态 State 和 StateOwner。

为了封装状态变化细节，我们可以抽象出一个状态机 StateMachine 来专门实现状态根据事情实现转换。

这样，客户端外界通过状态机可访问状态模式这个匣子。在实践中，外界客户端需要和状态机实现数据交换，我们把它也分为两种：属性和行为。

其中属性可能需要外界告诉状态状态变化的主体对象 Task，解决状态的主人问题，是谁的问题；行为可能是需要持久化当前这个主体对象的状态到数据库。

这两种数据交换可以分别通过 StateOwner 和 StateMachine 与整个状态机实现数据交换，这样，具体状态和状态切换也和外界实现了解耦隔离。

因此好的状态模式实现必须有下列步骤：

（1）将每个状态变成 State 的子类，实现每个状态对象化。

（2）在每个状态中，封装着进入下一个状态可能规则，这些规则是状态变化的核心，换句话说，统一了状态转换的规则。具体可采取 run 和 next 这样的转换行为。

下面是一个子状态代码：

```
public class Running extends StateT{
  //
  public void run(StateOwner stateOwner){
     stateOwner.setCurrentState(this);
  }
  //转换到下一个状态的规则
  //当前是 Running 状态，下一个状态可能是暂停、结束或者强制退出等
  //状态，但是绝对不会是 Not_Started 这样的状态
  //转换规则在这里得到了体现。
  public State next(Event e) {
    if(transitions == null){
       addEventState(new EventImp("PAUSE"), new Suspended());
       addEventState(new EventImp("END"), new Completed());
       addEventState(new EventImp("STOP"), new Aborted());
    }
    return super.next(e);
  }
}
```

外界直接调用 StateMachine 的关键方法 transition；实行状态的自动转变。

```
public class StateMachine {
 /**
   * 状态切换
   * 根据 Event 参数，运行相应的状态。
   * 1. 获得当前状态
   * 2. 使用当前状态的 next()转换
   *                    |Event
   * ___currentState__|_____newState___
   *
   * @param inputs
   */
  public final void transition(String taskid, Event e) throws Exception {
    State currentState = readCurrentState(taskid); //从数据库获得当前状态
    StateOwner stateOwner = new StateOwner(taskid, currentState);
    //转换状态
    currentState = currentState.next(e);
    if (currentState != null) {
      currentState.run(stateOwner);
      saveCurrentState(stateOwner); //保存当前状态
    }
  }
}
```

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-08 11:16 回复

iceant 发表文章: 463/ 注册时间: 2002-10-13 22:32

我原来的想法是使用 Event 为中心的处理机制: Event + Listener

拿你说的从一个房间进入另一个房间来说:

临界事件，就是跨过门。所以，在系统中，一般会有三个处理方法:

prePassThroughDoor();

doPassThroughDoor();

postPassThroughDoor();

在每个事件，都可以加入相应的 Listener. 这样可以做到系统的可扩展性。

但是这有个问题，就是所有的事件都必须去手写实现，这会很麻烦。

现在有 AspectJ, 如果我对 AspectJ 的理解没错的话，它很容易就可以实现这一点。你可以在任一方法调用之前，或之后，加入相应的事件点，在事件点上加入 Listener 机制。

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-08 11:34 回复

chenye99 发表文章: 15/ 注册时间: 2003-06-23 10:20

> public class Running extends StateT{
>
> //
> public void run(StateOwner stateOwner){
> stateOwner.setCurrentState(this);
> }
>
> //转换到下一个状态的规则
>
> //当前是 Running 状态，下一个状态可能是暂停、结束或者强
> 仆顺龅?> //状态，但是绝对不会是 Not_Started 这样的状态
> //转换规则在这里得到了体现。
> public State next(Event e) {
> if(transitions == null){
> addEventState(new EventImp("PAUSE"), new
>), new Suspended());
> addEventState(new EventImp("END"), new
>), new Completed());
> addEventState(new EventImp("STOP"), new
>), new Aborted());
> }
> return super.next(e);
> }
>
> }
>
>
>
> 外界直接调用

> StateMachine 的关键方法 transition；实行状态的自动转变。

对于 return super.next(e)的处理方式不是很清楚，可否给出 StateT 的 next 方法的代码？

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-08 18:00 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

StateT 的 next 方法如下：

```
public State next(Event e) throws InvalidStateException{
    logger.debug(" transitions size = " + transitions.size());
    String name = e.getName();
    if (transitions.containsKey(name))
      return (State) transitions.get(name);
    else
      throw new RuntimeException("Input not supported for current state " + " state is " +
this.getClass().getName()                                    + " Event is " + e.getName() );
  }
```

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-09 11:45 回复

chenye99 发表文章: 15/ 注册时间: 2003-06-23 10:20

我理解 Suspended、Completed、Aborted 类也都是 StateT 类的子类，不知是不是这样？

在 StateT 类的 next 方法中，是根据事件来进行跳转，假设可能又两个状态都存在 PAUSE 事件，又怎么区分呢？

在刚看到方案时我的感觉是 StateT 子类中 next 方法应为 return super.next(this,e)，不知是不是我的理解有问题？

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-09 12:19 回复

wild fox 发表文章: 55/ 注册时间: 2003-03-19 17:34

如果你的状态 State 非常多，而且 Event 也非常多，并且每一个 Event 都可能在很多个不同的 State 中起作用并使系统进入到不同的 State,那么，是不是需要在每一个 State 的 next()方法中都要用代码写出来呢？

另外假如对于

>(StateA)---[Event1]--->(StateB)

过了一段时间需要>(StateA)---[Event1]--->(StateC)那么又要到代码里面去修改了？

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-09 14:09 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

to chenye99

你的理解基本没错，从实际角度考虑，没有两个状态存在同样事件的，因为事件相当于外界给予能量，主体有了新能量，本身能量会发生变化，这类似能量守恒定律。

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-09 14:27 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

to wild fox

状态模式只关注某个状态，以及这个状态可能转入的下几个状态。

我们不必在乎一个流程有很多状态，一个个分开对付就可以了。

对于

>(StateA)---[Event1]--->(StateB)

改变到

>(StateA)---[Event1]--->(StateC)

我们所要改变的只是 StateA 这个类的 next 这个方法代码。当然，也可以将 next 方法中可能对应的下一个状态配置在配置文件中，这样就不需要改代码了。

重要的是，我们使用状态模式封装了状态和状态转换规则，使他们没有互相混乱地纠缠在一起。

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-09 18:00 回复

wild fox 发表文章: 55/ 注册时间: 2003-03-19 17:34

我现在在做的项目就是采用的 State + StateMachine 模式的，不过我们采用了配置文件来设置源 State 和目标 State 之间的切换，以次来增加系统的灵活性，虽然有一定的效果，可是，我们不但需要关心当前 State,驱动事件，还需要考虑源 State，以及当前 State 的一些条件来决定目标 State,可是各个 State 中还是有很多 if...else if...else... ，而且，当源 State 和目标 State 之间的切换规则发生了变化时，修改了配置文件以后，往往还是需要对相应的代码进行修改，怎样做才能在 State 间的切换规则发生改变以后尽可能地少对代码进行修改呢？

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-10 08:44 回复

chenye99 发表文章: 15/ 注册时间: 2003-06-23 10:20

"以及当前 State 的一些条件来决定目标 State"可不可以理解为当前 state 也存在不同的状态？如果是这样，是否有将状态进一步细化的可能？

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-10 09:18 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

to wild fox

你们存在的这个情况，说明没有真正用好状态模式，这再一次说明，模式用好真的不容易。

用好状态模式的前提必须有对系统状态和事件最本质的认识和抽象。

但就我举的这个实例，next 方法中实际是给一个 Hashmap 赋值，所以，这个过程完成可以使用 XML 配置。

要做到方便的 XML 配置，前提还是必须将状态和状态转换规则独立分离开，否则 XML 配置将只是一种形式。

XMl 配置是解耦后跟进一步的表现，如果你的逻辑能够做到 XML 配置即可运行，那么说明你之前的解耦设计已经非常成功。

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-10 09:19 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

to chenye99

状态有子状态，类似树形结构。使用状态模式处理子状态会更复杂。

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2003-12-26 17:40 回复

天下为公 发表文章: 31/ 注册时间: 2003-07-15 17:09

EventImp 与
Event 的关系，请说一下，我不是太明白？

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2004-09-24 10:03 回复

wm_creat 发表文章: 10/ 注册时间: 2003-09-13 16:20

那位老兄说得很对啊，你也知道 1 个事件是由一个事物驱动，一个事物同一时间只有一种状态，一个状态却可能对应 0 到 n 个事物，如果你存储状态改变的数据结构没有相关事物信息，将无法区分这个状态是 A 事物驱动的还是 B 事物驱动的，换句话说，这个数据结构的主键是：事物 id，状态 id，而不是一个状态 id 就可以的，不然必定违反数据库原理！

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-26 09:47 回复

chenyongguang 发表文章: 11/ 注册时间: 2005-12-26 09:44

您好,请问 banq,StateOwner 类的源代码是什么?对 StateOwner 的作用不甚理解.谢谢

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-26 09:54 回复

chenyongguang 发表文章: 11/ 注册时间: 2005-12-26 09:44

to banq,banq 老师,能否将您上次讲的公文流转的例子讲的再详细些,目前正在做些这方面的尝试,很多地方理解的不太好.特别是 StateOwner 这个对象的作用不是太理解,能否给出代码并解答,实在是不胜感激啊!

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-26 10:06 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

具体研究可看看工作流方面的东西，基于状态模式，但是更大，有个开源 osworkflow 还是不错。

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-26 10:41 回复

chenyongguang 发表文章: 11/ 注册时间: 2005-12-26 09:44

to banq,
谢谢 banq 老师的这么迅速的回复。有个疑问还想请教您。我觉得各个状态转换应该有用户的参与，在上面的讨论中，用户好象没有提及（还是被 Event 代替？即使是被 event 代替的话，event 中也应该要包含用户信息啊，不知这样的理解对不对？）。如果用户信息要被加入，是放在 State 的各个子类中还是放在 Event 里？是设置为一个 actor 还是一个 role?谢谢！

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-27 10:25 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

>状态态转换应该有用户的参与
注意耦合，状态模式的特点就是用事件作为输入信号，状态作为输出信号，而事件的发生则是由用户触发的：
用户---->事件--->状态。
所以用户离我们讨论的状态模式很远，不用拉进来。

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-28 09:26 回复

chenyongguang 发表文章: 11/ 注册时间: 2005-12-26 09:44

to banq,
谢谢 banq 老师。banq 老师在前面谈到怎样将状态机与应用逻辑解耦。但在实现时，状态机又必须跟应用逻辑结合，特别是状态机进行状态切换的时候，需要访问相关的应用数据。比如一个会议室的申请流程，如何实现状态机与应用数据的交互？

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-28 09:30 回复

chenyongguang 发表文章: 11/ 注册时间: 2005-12-26 09:44

to banq,
是通过事件或参数来向实例化后的状态机传递相关数据吗？如果是，还请 banq 老师详加指点一二。谢谢

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-28 10:12 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

你的问题是一种实际中经常遇到的解耦问题：
本帖开始时的代码 transition 中一段代码
State currentState = readCurrentState(taskid); //从数据库获得当前状态
实际就是和业务逻辑发生关系，通过依赖或关联关系可和业务逻辑层进行交互，当然，之间最好是面向接口的。

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-28 21:40 回复

chenyongguang 发表文章: 11/ 注册时间: 2005-12-26 09:44

通过 banq 老师的多次指点，今天终于在前面介绍的思想和部分代码的基础上实现了一个非常简单的工作流状态机，是一个有关会议室申请的流程，在这里非常感谢 banq 老师和其他的 jdon 道友。但同时也对前面一个道友谈到的观点深有体会，就是状态太多了。（我设计了一个 State,下面有许多具体子类，比如 Committed,Passed,Finished 之类），如下图。
　　有没有更好的方法？另外，比如公文发文流程，它一般有拟稿、核稿、批准、归档等状态，这些状态是不是又要重写？还是可以复用会议室申请的那个工作流状态机？也就是说可不可以抽象出一个一般工作流状态机，然后根据具体应用逻辑，在实例化成会议室申请状态机、公文发文状态机、物品领用状态机之类的？

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-28 21:44 回复

chenyongguang 发表文章: 11/ 注册时间: 2005-12-26 09:44

这里特别想请教一下的是，对公文发文流程中的**会签操作**，该怎么处理？感觉很难啊，敬请指教！

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-29 09:41 回复

banq 发表文章: 7239/ 注册时间: 2002-08-03 17:08

状态机可以细化，分类成很多。
图画得不错，如果使用 UNML 的状态图就不感觉那么“飞舞”了

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2005-12-29 15:06 回复

chenyongguang 发表文章: 11/ 注册时间: 2005-12-26 09:44

to banq,
我用的是 visio 的数据流状态图，本来是准备用 visio 里面的 UML 状态图的。还请 banq 兄不要避重就轻，对我的疑问要

详加解释才是啊，哈哈！

Re: 从工作流状态机实践中总结状态模式使用心得 发表: 2006-04-28 11:43 回复

tuzhihai 发表文章: 1/ 注册时间: 2006-04-28 11:38

但对于下列情况该如何描述：

假设一个对象处于正在执行状态，其要做的事是与另一个系统通信，这个通信可能成功也可能失败，如果成功，该对象应处于完成状态，如果不成功，该对象应该处于失败状态。

这种情况如果用单一的 FROM_STATE--EVENT--TO_STATE 该如何表达？

Advanced form processing using JSP

Use the Memento design pattern with JavaServer Pages and JavaBeans

By Govind Seshadri, JavaWorld.com, 03/01/00

Processing HTML forms sounds like a fairly easy task. For simplistic forms, it usually is. However, this "easy task" (as anyone who has written a halfway sophisticated Web application will confirm) can quickly become tedious. This is because handling HTML forms usually involves a lot more than just parsing the request parameters and outputting a response back to the client. Typically, form processing involves multiple components operating in the background, with each component responsible for a discrete task such as state management, data validation, database access, and so on. While there are numerous examples that demonstrate form processing with Perl scripts and servlets, using JSPs for this purpose has received little attention. There is a reason for this. Apart from the fact that JSP is a fairly new technology, many view it as being suitable mostly for handling the presentation of dynamic content sourced from either JavaBeans or servlets. However, as you shall soon see, the combination of JSP with JavaBeans can be a force to reckon with when processing HTML forms.

In this article, I will examine the handling of a user registration form using JSP. One of the basic programming tenets of JSP is to delegate as much processing as possible to JavaBean components. My JSP form-handling implementation will demonstrate some interesting features. It will not only provide basic data validation for the registration information input by a user, but will also exhibit stateful behavior. This lets you pre-fill the form's input elements with validated data as the user loops through the submission cycle and finally enters the correct data for all of the input elements. So, without further ado, let's dive into the example.

Take a look at Listing 1, which presents the user with a simple registration form, displayed in Figure 1.

Listing 1. register.html

```
<html>
<body>
<form action="/examples/jsp/forms/process.jsp" method=post>
<center>
<table cellpadding=4 cellspacing=2 border=0>
<th bgcolor="#CCCCFF" colspan=2>
<font size=5>USER REGISTRATION</font>
<br>
<font size=1><sup>*</sup> Required Fields</font>
</th>
<tr bgcolor="#c8d8f8">
<td valign=top>
<b>First Name<sup>*</sup></b>
```

```
<br>
<input type="text" name="firstName" value="" size=15 maxlength=20></td>
<td  valign=top>
<b>Last Name<sup>*</sup></b>
<br>
<input type="text" name="lastName" value="" size=15 maxlength=20></td>
</tr>
<tr bgcolor="#c8d8f8">
<td valign=top>
<b>E-Mail<sup>*</sup></b>
<br>
<input type="text" name="email" value="" size=25  maxlength=125>
<br></td>
<td  valign=top>
<b>Zip Code<sup>*</sup></b>
<br>
<input type="text" name="zip" value="" size=5  maxlength=5></td>
</tr>
<tr bgcolor="#c8d8f8">
<td valign=top colspan=2>
<b>User Name<sup>*</sup></b>
<br>
<input type="text" name="userName" size=10 value=""  maxlength=10>
</td>
</tr>
<tr bgcolor="#c8d8f8">
<td valign=top>
<b>Password<sup>*</sup></b>
<br>
<input type="password" name="password1" size=10 value=""  maxlength=10></td>
<td  valign=top>
<b>Confirm Password<sup>*</sup></b>
<br>
<input type="password" name="password2" size=10 value=""  maxlength=10></td>
<br>
</tr>
<tr bgcolor="#c8d8f8">
<td  valign=top colspan=2>
<b>What music are you interested in?</b>
<br>
<input type="checkbox" name="faveMusic"
value="Rock">Rock
<input type="checkbox" name="faveMusic" value="Pop">Pop
<input type="checkbox" name="faveMusic" value="Bluegrass">Bluegrass<br>
```

```
<input type="checkbox" name="faveMusic" value="Blues">Blues
<input type="checkbox" name="faveMusic" value="Jazz">Jazz
<input type="checkbox" name="faveMusic" value="Country">Country<br>
</td>
</tr>
<tr bgcolor="#c8d8f8">
<td  valign=top colspan=2>
<b>Would you like to receive e-mail notifications on our special
sales?</b>
<br>
<input type="radio" name="notify" value="Yes" checked>Yes
<input type="radio" name="notify" value="No" > No
<br><br></td>
</tr>
<tr bgcolor="#c8d8f8">
<td  align=center colspan=2>
<input type="submit" value="Submit"> <input type="reset"
value="Reset">
</td>
</tr>
</table>
</center>
</form>
</body>
</html>
```

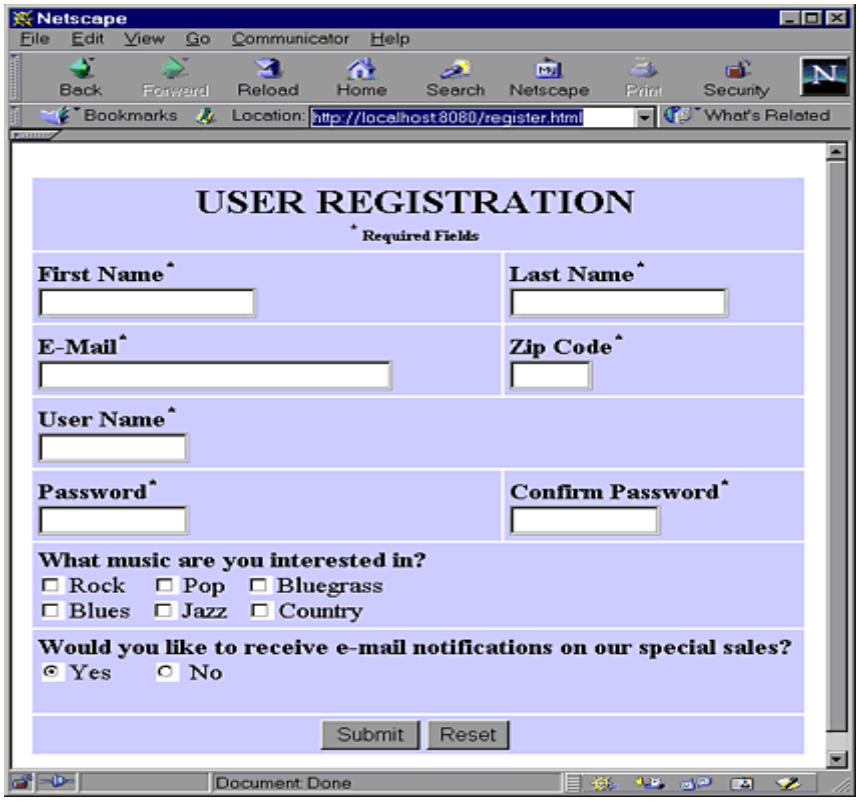


Figure 1. The user registration form

There should be nothing unusual about the form shown in Listing 1. It contains all of the commonly used input elements, including text-entry fields, checkboxes, and radio buttons. However, notice the action clause of the form:

```
<form action="/examples/jsp/forms/process.jsp" method=post>
```

Although typically you may have specified a servlet or Perl script, note that a JSP is perfectly capable of processing data posted from an HTML form. This should not surprise you, since after all, what is JSP? It is nothing but a high-level abstraction of servlets. Thus, in most cases it is entirely feasible to write a JSP equivalent to a servlet.

Still, you should always remember that JSP technology was created for an entirely different purpose than serving as an alternate (some would say easier!) mechanism for creating servlets. JSP is all about facilitating the separation of presentation from dynamic content. Although you can embed any amount of Java code within a JSP page, your best bet is to encapsulate the processing logic within reusable JavaBean components. Nevertheless, in my opinion, it should also be perfectly appropriate to develop *controller* JSP pages. These pages would still delegate the bulk of the processing to component beans, but they would also contain some conditional logic to respond to a user's actions. But these controller pages would never contain presentation logic to display UI elements. This task would always be externalized into separate JSPs, which will be invoked as needed by the controller.

Take a look at Listing 2, which demonstrates a JSP serving as a controller.

Listing 2. process.jsp

```
<%@ page import="java.util.*" %>
<%!
    ResourceBundle bundle =null;
    public void jspInit() {
       bundle = ResourceBundle.getBundle("forms");
      }
%>
<jsp:useBean id="formHandler" class="foo.FormBean" scope="request">
<jsp:setProperty name="formHandler" property="*"/>
</jsp:useBean>
<%
   if (formHandler.validate()) {
%>
    <jsp:forward page="<%=bundle.getString(\"process.success\")%>"/>
<%
   }  else {
%>
    <jsp:forward page="<%=bundle.getString(\"process.retry\")%>"/>
<%
   }
%>
```

Because we are delegating the bulk of the processing to JavaBeans, the first thing the controller has to do is instantiate the bean component. This is done with the `<jsp:useBean>` tag as follows:

```
<jsp:useBean id="formHandler" class="foo.FormBean" scope="request">
```

```
    <jsp:setProperty name="formHandler" property="*"/>
</jsp:useBean>
```

The <jsp:useBean> tag first looks for the bean instance with the specified name, and instantiates a new one only if it cannot find the bean instance within the specified scope. Here, the scope attribute specifies the lifetime of the bean. Newly instantiated beans have page scope by default, if nothing is specified. Observe that in this case, I specify that the bean have request scope before a response is sent back to the client, since more than one JSP is involved in processing the client request.

You may be wondering about the <jsp:setProperty> within the body of the <jsp:useBean> tag. Any scriptlet or <jsp:setProperty> tags present within the body of a <jsp:useBean> tag are executed only when the bean is instantiated, and are used to initialize the bean's properties. Of course, in this case I could have placed the <jsp:setProperty> tag on the outside of the <jsp:useBean>'s body. The difference between the two is that the contents of the body are not executed if the bean is retrieved from the specified scope -- which is moot in this case since the bean is instantiated each time the controller is invoked.

Introspective magic

When developing beans for processing form data, you can follow a common design pattern by matching the names of the bean properties with the names of the form input elements. You would also need to define the corresponding getter/setter methods for each property within the bean. For example, within the FormBean bean (shown in Listing 3), I define the property firstName, as well as the accessor methods getFirstName() and setFirstName(), corresponding to the form input element named firstName. The advantage in this is that you can now direct the JSP engine to parse all the incoming values from the HTML form elements that are part of the request object, then assign them to their corresponding bean properties with a single statement, like this:

```
  <jsp:setProperty name="formHandler" property="*"/>
```

This runtime magic is possible through a process called *introspection*, which lets a class expose its properties on request. The introspection is managed by the JSP engine, and implemented via the Java reflection mechanism. This feature alone can be a lifesaver when processing complex forms containing a significant number of input elements.

Listing 3. FormBean.java

```
package foo;
import java.util.*;
public class FormBean {
  private String firstName;
  private String lastName;
  private String email;
  private String userName;
  private String password1;
  private String password2;
  private String zip;
  private String[] faveMusic;
  private String notify;
  private Hashtable errors;
  public boolean validate() {
    boolean allOk=true;
    if (firstName.equals("")) {
      errors.put("firstName","Please enter your first name");
```

```
    firstName="";
    allOk=false;
  }
  if (lastName.equals("")) {
    errors.put("lastName","Please enter your last name");
    lastName="";
    allOk=false;
  }
  if (email.equals("") || (email.indexOf('@') == -1)) {
    errors.put("email","Please enter a valid email address");
    email="";
    allOk=false;
  }
  if (userName.equals("")) {
    errors.put("userName","Please enter a username");
    userName="";
    allOk=false;
  }
  if (password1.equals("") ) {
    errors.put("password1","Please enter a valid password");
    password1="";
    allOk=false;
  }
  if (!password1.equals("") && (password2.equals("") ||
      !password1.equals(password2))) {
    errors.put("password2","Please confirm your password");
    password2="";
    allOk=false;
  }
  if (zip.equals("") || zip.length() !=5 ) {
    errors.put("zip","Please enter a valid zip code");
    zip="";
    allOk=false;
  } else {
    try {
      int x = Integer.parseInt(zip);
    } catch (NumberFormatException e) {
      errors.put("zip","Please enter a valid zip code");
      zip="";
      allOk=false;
    }
  }
  return allOk;
}
```

```
public String getErrorMsg(String s) {
  String errorMsg =(String)errors.get(s.trim());
  return (errorMsg == null) ? "":errorMsg;
}
public FormBean() {
  firstName="";
  lastName="";
  email="";
  userName="";
  password1="";
  password2="";
  zip="";
  faveMusic = new String[] { "1" };
  notify="";
  errors = new Hashtable();
}
public String getFirstName() {
  return firstName;
}
public String getLastName() {
  return lastName;
}
public String getEmail() {
  return email;
}
public String getUserName() {
  return userName;
}
public String getPassword1() {
  return password1;
}
public String getPassword2() {
  return password2;
}
public String getZip() {
  return zip;
}
public String getNotify() {
  return notify;
}
public String[] getFaveMusic() {
  return faveMusic;
}
public String isCbSelected(String s) {
```

```
  boolean found=false;
  if (!faveMusic[0].equals("1")) {
    for (int i = 0; i < faveMusic.length; i++) {
      if (faveMusic[i].equals(s)) {
        found=true;
        break;
      }
    }
    if (found) return "checked";
  }
  return "";
}
public String isRbSelected(String s) {
  return (notify.equals(s))? "checked" : "";
}
public void setFirstName(String fname) {
  firstName =fname;
}
public void setLastName(String lname) {
  lastName =lname;
}
public void setEmail(String eml) {
  email=eml;
}
public void setUserName(String u) {
  userName=u;
}
public void  setPassword1(String p1) {
  password1=p1;
}
public void  setPassword2(String p2) {
  password2=p2;
}
public void setZip(String z) {
  zip=z;
}
public void setFaveMusic(String[] music) {
  faveMusic=music;
}
public void setErrors(String key, String msg) {
  errors.put(key,msg);
}
public void setNotify(String n) {
  notify=n;
```

```
  }
}
```

It is not mandatory that the bean property names always match those of the form input elements, since you can always perform the mapping yourself, like this:

```
  <jsp:setProperty name="formHandler" param="formElementName"
    property="beanPropName"/>
```

While you can easily map form input types like text-entry fields and radio buttons to a scalar bean property type like `String`, you have to use an indexed property type like `String[]` for handling checkboxes. Nevertheless, as long as you provide a suitable setter method you can still have the JSP engine assign values even for non-scalar types by automatically parsing the corresponding form element values from the request object.

Observe the setter method for setting the values of selected checkboxes:

```
public void setFaveMusic(String[] music) {
    faveMusic=music;
}
```

I've also shown a simple technique to avoid hardcoding the target resources within the controller. Here, the targets are stored within an external properties file, `forms.properties`, as shown in Listing 4. This file may be located anywhere in the `CLASSPATH` that is visible to the JSP container. The `jspInit()` method declared within the page is automatically executed by the JSP container when the JSP page (or to be more accurate, the servlet class representing the page) is loaded into memory, and invoked just once during the JSP's lifetime. By making use of the `ResourceBundle` facility, your page can access the values for the target resources by name, just like you would with a `Hashtable` object.

Listing 4. forms.properties

```
process.success=success.jsp
process.retry=retry.jsp
```

Form handling using the Memento pattern

The processing logic within the controller page is straightforward. After the bean is instantiated, its `validate()` method is invoked. The `validate()` method has a two-pronged effect. If an error is encountered during the validation of any form input element, the `validate()` method not only resets the value of the corresponding bean property, but also sets an appropriate error message, which can later be displayed for that input element.
If any of the required form elements cannot be successfully validated, the controller forwards the request to the JSP page retry.jsp (shown in Listing 5), allowing the user to make changes and resubmit the form. If there are no validation errors, the request is forwarded to success.jsp, shown in Listing 6.

Within retry.jsp, you first obtain a reference to the bean component that was previously instantiated and placed into the request by the controller. Also, since you don't want the user to reenter previously validated data, you refill the form elements by interrogating their previous state from the bean, like this:

```
<input type="text" name="firstName"
   value='<%=formHandler.getFirstName()%>' size=15 maxlength=20>
```

Any error message that may be applicable for the form input element is also retrieved and displayed via:

```
<font size=2 color=red>
  <%=formHandler.getErrorMsg("firstName")%>
</font>
```

Also, observe the way I recreate the prior state of form input elements like radio buttons and checkboxes by using utility methods such as `isRbSelected(String s)` and `isCbSelected(String s)`, which were incorporated into the bean. Figure 2 shows the form generated by retry.jsp, which indicates some validation errors.

Listing 5. retry.jsp

```
<jsp:useBean id="formHandler" class="foo.FormBean" scope="request"/>
<html>
<body>
<form action="process.jsp" method=post>
<center>
<table cellpadding=4 cellspacing=2 border=0>
<th bgcolor="#CCCCFF" colspan=2>
<font size=5>USER REGISTRATION</font>
<br>
<font size=1><sup>*</sup> Required Fields </font>
</th>
<tr bgcolor="#c8d8f8">
<td valign=top>
<B>First Name<sup>*</sup></B>
<br>
<input type="text" name="firstName"
value='<%=formHandler.getFirstName()%>' size=15 maxlength=20>
<br><font size=2
color=red><%=formHandler.getErrorMsg("firstName")%></font>
</td>
<td  valign=top>
<B>Last Name<sup>*</sup></B>
<br>
<input type="text" name="lastName"
value='<%=formHandler.getLastName()%>' size=15 maxlength=20>
<br><font size=2
color=red><%=formHandler.getErrorMsg("lastName")%></font>
</td>
</tr>
<tr bgcolor="#c8d8f8">
<td valign=top>
<B>E-Mail<sup>*</sup></B>
<br>
<input type="text" name="email" value='<%=formHandler.getEmail()%>'
size=25  maxlength=125>
<br><font size=2 color=red><%=formHandler.getErrorMsg("email")%></font>
</td>
<td  valign=top>
<B>Zip Code<sup>*</sup></B>
<br>
```

```
<input type="text" name="zip" value='<%=formHandler.getZip()%>' size=5
maxlength=5>
<br><font size=2 color=red><%=formHandler.getErrorMsg("zip")%></font>
</td>
</tr>
<tr bgcolor="#c8d8f8">
<td valign=top colspan=2>
<B>User Name<sup>*</sup></B>
<br>
<input type="text" name="userName" size=10
value='<%=formHandler.getUserName()%>'  maxlength=10>
<br><font size=2
color=red><%=formHandler.getErrorMsg("userName")%></font>
</td>
</tr>
<tr bgcolor="#c8d8f8">
<td valign=top>
<B>Password<sup>*</sup></B>
<br>
<input type="password" name="password1" size=10
value='<%=formHandler.getPassword1()%>'  maxlength=10>
<br><font size=2
color=red><%=formHandler.getErrorMsg("password1")%></font>
</td>
<td  valign=top>
<B>Confirm Password<sup>*</sup></B>
<br>
<input type="password" name="password2" size=10
value='<%=formHandler.getPassword2()%>'  maxlength=10>
<br><font size=2
color=red><%=formHandler.getErrorMsg("password2")%></font>
</td>
<br>
</tr>
<tr bgcolor="#c8d8f8">
<td colspan=2 valign=top>
<B>What music are you interested in?</B>
<br>
<input type="checkbox" name="faveMusic" value="Rock"
<%=formHandler.isCbSelected("Rock")%>>Rock
<input type="checkbox" name="faveMusic" value="Pop"
<%=formHandler.isCbSelected("Pop")%>>Pop
<input type="checkbox" name="faveMusic" value="Bluegrass"
<%=formHandler.isCbSelected("Bluegrass")%>>Bluegrass<br>
```

```
<input type="checkbox" name="faveMusic" value="Blues"
<%=formHandler.isCbSelected("Blues")%>>Blues
<input type="checkbox" name="faveMusic" value="Jazz"
<%=formHandler.isCbSelected("Jazz")%>>Jazz
<input type="checkbox" name="faveMusic" value="Country"
<%=formHandler.isCbSelected("Country")%>>Country<br>
</td>
</tr>
<tr bgcolor="#c8d8f8">
<td colspan=2 valign=top>
<B>Would you like to receive e-mail notifications on our special
sales?</B>
<br>
<input type="radio" name="notify" value="Yes"
<%=formHandler.isRbSelected("Yes")%>>Yes
<input type="radio" name="notify" value="No"
<%=formHandler.isRbSelected("No")%>> No
<br><br></td>
</tr>
<tr bgcolor="#c8d8f8">
<td colspan=2 align=center>
<input type="submit" value="Submit"> <input type="reset"
value="Reset">
</td>
</tr>
</table>
</center>
</form>
</body>
</html>
```

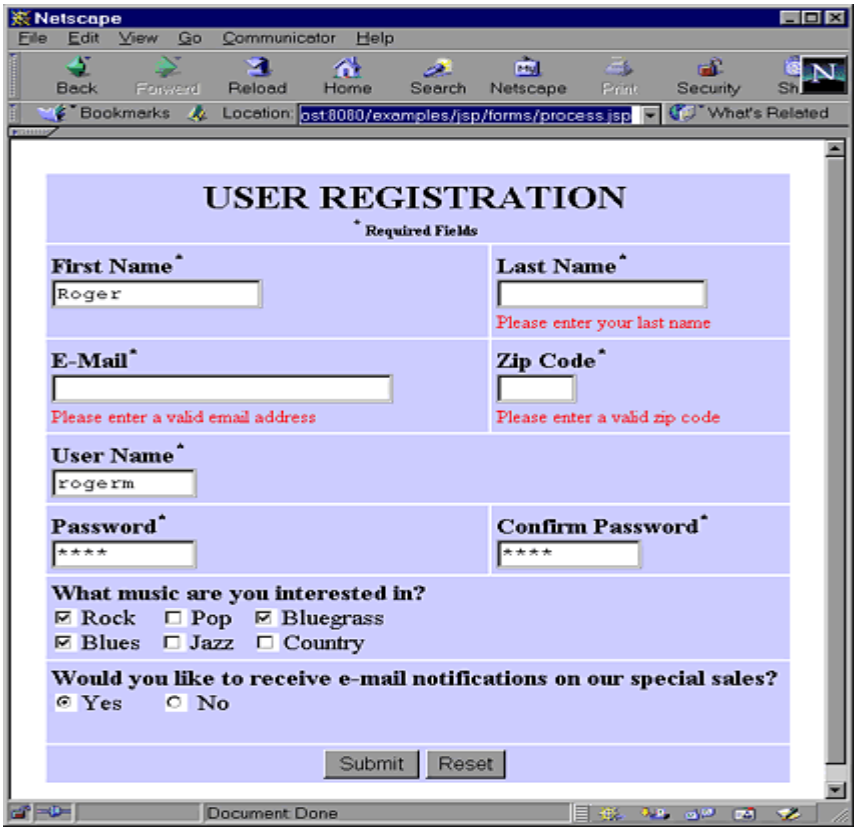


Figure 2. Form generated by retry.jsp

Because retry.jsp also posts data to process.jsp, the controller repeatedly instantiates the bean component and validates the form data until the user has entered valid data for all of the form elements.

Using a bean within a form in this way can be viewed as an implementation of the Memento design pattern. Memento is a behavioral pattern whose intent is to take a snapshot of a portion of an object's state so that the object can be restored to that state later, without violating its encapsulation. Because the bean is created and accessed within the boundary of the same request, the encapsulation of the memento is preserved intact.

As stated earlier, the controller forwards the request to success.jsp only after all of the submitted form data has been successfully validated. Success.jsp in turn extracts the bean component from the request and confirms the registration to the client. Note that while the scalar bean properties can be retrieved using a JSP expression or the `<jsp:getProperty>` tag, you still have to jump through a few hoops in order to display the indexed property type:

```
<%
  String[] faveMusic = formHandler.getFaveMusic();
  if (!faveMusic[0].equals("1")) {
    out.println("<ul>");
    for (int i=0; i<faveMusic.length; i++)
      out.println("<li>"+faveMusic[i]);
    out.println("</ul>");
  } else out.println("Nothing was selected");
%>
```

In the future, you should not have to use scriptlet code to access indexed properties, but would instead use

custom tags that support looping. Figure 3 shows the client on completion of a successful registration.

Listing 6. success.jsp

```
<jsp:useBean id="formHandler" class="foo.FormBean" scope="request"/>
<html>
<body>
<center>
<table cellpadding=4 cellspacing=2 border=0>
<th bgcolor="#CCCCFF" colspan=2>
<font size=5>USER REGISTRATION SUCCESSFUL!</font>
</th>
<font size=4>
<tr bgcolor="#c8d8f8">
<td valign=top>
<b>First Name</b>
<br>
<jsp:getProperty name="formHandler" property="firstName"/>
</td>
<td valign=top>
<b>Last Name</b>
<br>
<jsp:getProperty name="formHandler" property="lastName"/>
</td>
</tr>
<tr bgcolor="#c8d8f8">
<td valign=top>
<b>E-Mail</b>
<br>
<jsp:getProperty name="formHandler" property="email"/>
<br></td>
<td valign=top>
<b>Zip Code</b>
<br>
<jsp:getProperty name="formHandler" property="zip"/>
</td>
</tr>
<tr bgcolor="#c8d8f8">
<td valign=top colspan=2>
<b>User Name</b>
<br>
<jsp:getProperty name="formHandler" property="userName"/>
</td>
</tr>
<tr bgcolor="#c8d8f8">
<td colspan=2 valign=top>
```

```
<b>What music are you interested in?</b>
<br>
<%
  String[] faveMusic = formHandler.getFaveMusic();
  if (!faveMusic[0].equals("1")) {
    out.println("<ul>");
    for (int i=0; i<faveMusic.length; i++)
      out.println("<li>"+faveMusic[i]);
    out.println("</ul>");
  } else out.println("Nothing was selected");
%>
</td>
</tr>
<tr bgcolor="#c8d8f8">
<td colspan=2 valign=top>
<b>Would you like to receive e-mail notifications on our special
sales?</b>
<br>
<jsp:getProperty name="formHandler" property="notify"/>
</td>
</tr>
</font>
</table>
</center>
</body>
</html>
```

Figure 3. Confirmation of a successful registration

Although you saw the bean perform extensive state management and form validation, you may have noticed that the only validation performed on the `userName` element was to simply confirm that it was not blank! Typically, when adding new users to a database, your registration program has to first ensure that the username is unique and was not previously entered into the database. If you already have a database access servlet for this purpose, you may wonder how you can integrate it into the validation scenario discussed thus far. Well, the answer is, "Very easily!" You can simply forward the request to a servlet, just as you would to a JSP page, and continue processing. The servlet in turn can update the bean (or even add new beans to the request) and forward the request to another resource down the chain.

For example, you can make a minor change to the controller, process.jsp, so that it now forwards the request to a servlet on successful validation, instead of sending it to the success.jsp page as it did before:

```
<% if (formHandler.validate()) {
%>
  <jsp:forward page="/servlet/DBHandler"/>
<%
   }  else {
      // continue as before
   }
%>
```

Consider the servlet `DBHandler` shown in Listing 7.

Listing 7. A database access servlet

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import foo.FormBean;
public class DBHandler extends HttpServlet {
  public void doPost (HttpServletRequest request, HttpServletResponse response) {
    try {
      FormBean f = (FormBean) request.getAttribute("formHandler");
      boolean userExists = false;
      //obtain a db connection and perform a db query
      //ensuring that the username does not exist
      //set userExists=true if user is found in db
      //for a simple test, you can disallow the registration
      //of the username "rogerm" as:
      //if (f.getUserName().equals("rogerm")) userExists=true;
      if (userExists) {
        f.setErrors("userName","Duplicate User: Try a different username");
        getServletConfig().getServletContext().
          getRequestDispatcher("/jsp/forms/retry.jsp").
          forward(request, response);
      } else {
        //retrieve the bean properties and store them
```

```
        // into the database.
        getServletConfig().getServletContext().
          getRequestDispatcher("/jsp/forms/success.jsp").
          forward(request, response);
      }
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Since the bean is still present within the request scope, you can easily access it within the servlet, like this:

```
FormBean f = (FormBean) request.getAttribute ("formHandler");
```

Assuming that the servlet performs a database query and finds another user with the same username, the servlet can easily call a setter method within the bean to indicate the error condition, like this:

```
f.setErrors("userName","Duplicate User: Try a different username");
```

Once again, the servlet can then send the request along its merry way by means of a request dispatcher.

```
getServletConfig().getServletContext().
    getRequestDispatcher("/jsp/forms/retry.jsp").
    forward(request, response);
```

Figure 4 shows a situation in which the servlet, `DBHandler`, was able to locate a duplicate username.

Figure 4. Form displaying the duplicate user error generated by the servlet

If the servlet does not find a duplicate user within the database, it can update the database after accessing the registration information from the bean, and then forward the request to the success.jsp page confirming a successful registration.

Deploying the application

I will assume that you are using Sun's latest version of JavaServer Web Development Kit (JSWDK) to run the example. If you aren't, see the Resources section to find out where to get it. Assuming that the server is installed in \jswdk-1.0.1, its default location under Microsoft Windows, deploy the application files as follows:

- Copy register.htm to \jswdk-1.0.1\webpages
- Create the directory "forms" under \jswdk-1.0.1\examples\jsp
- Copy process.jsp to \jswdk-1.0.1\examples\jsp\forms
- Copy retry.jsp to \jswdk-1.0.1\examples\jsp\forms
- Copy success.jsp to \jswdk-1.0.1\examples\jsp\forms
- Copy forms.properties to \jswdk-1.0.1\examples\WEB-INF\jsp\beans
- Copy FormBean.java to \jswdk-1.0.1\examples\WEB-INF\jsp\beans
- Compile FormBean.java by typing javac -d . FormBean.java

  This should create \jswdk-1.0.1\examples\WEB-INF\jsp\beans\foo\FormBean.class. If you are also testing the servlet, you will need to:
- Update process.jsp to forward the request to the servlet as shown earlier
- Update the classpath to include \jswdk-1.0.1\examples\WEB-INF\jsp\beans
- Copy DBHandler.java to \jswdk-1.0.1\examples\WEB-INF\servlets
- Compile DBHandler.java by typing javac DBHandler.java

  Once your server has been started, you should be able to access the application using http://localhost:8080/register.htm as the URL.

Conclusion

While there are numerous established solutions for handling HTML forms, there are many compelling reasons why JSP makes a viable alternative to the more mainstream solutions. JSP, with its component bean-centric approach, may actually ease the processing of complex forms. The JSP container can also reduce the processing burden significantly by instantiating bean components and automatically parsing the request object. Following a bean-centric approach also makes it easier to implement design patterns like Memento, which can play a useful role in the form validation process. Using JSP for form handling does not preclude the use of servlets, as these complementary technologies can be effectively melded using techniques like request chaining.

Author Bio

Govind Seshadri is an Enterprise Java Guru for jGuru.com, and the author of Enterprise Java Computing -- Applications and Architecture from Cambridge University Press (1999). Learn more about Govind at jGuru.com.

Jsp 中的 session 使用

Jsp 的 session 是使用 bean 的一个生存期限,一般为 page,session 意思是在这个用户没有离开网站之前一直有效,如果无法判断用户何时离开,一般依据系统设定,tomcat 中设定为 30 分钟.

我们使用 seesion 功能,可以达到多个 jsp 程序从操作同一个 java bean, 那么这个 java bean 可以作为我们传统意义上的"全局变量池".(在 java 中我们可以使用 static 静态化一个变量和方法,使用 singleton 唯一化对象.)

在项目实践中,我们 Jsp 程序中很多参数需要从数据库中读取,有的参数实际读取一次就可以,如果设计成每个用户每产生一个页面都要读取数据库,很显然,数据库的负载很大,同时也浪费时间,虽然可能有数据库连接池优化,但是尽量少使用数据库是我们编程的原则.

比如,我们的 test.jsp 和 test1.jsp 都需要得到一个参数 userdir,这个 userdir 是从数据库中得知,使用 session 将大大优化性能,程序如下:

设计一个 javabean 存储 userdir.

```
public class UserEnv {
private String userdir = "";
private String userurl = "";
public UserEnv(){
//构建方法初始化 userdir,可以从数据库中读取,这里简单给值 ppp
userdir="pppp";
System.out.println("init userdir, one time");
}
public String getUserdir() throws Exception{
return userdir;
}
}
```

test1.jsp 程序:

```
<%@ page contentType="text/html;charset=ISO8859_1" %>
<jsp:useBean id="myenv" scope="session" class="mysite.UserEnv"/>
<html>
<head>
<title>Untitled</title>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</head>
<body>
this is test1.jsp:<%=myenv.getUserdir()%>
</body>
</html>
```

test2.jsp 程序:

```
<%@ page contentType="text/html;charset=ISO8859_1" %>
<jsp:useBean id="myenv" scope="session" class="mysite.UserEnv"/>
<html>
<head>
<title>Untitled</title>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</head>
<body>
this is test2.jsp:<%=myenv.getUserdir()%>
</body>
</html>
```

无论用户先调用 test1.jsp 还是 test2.jsp, java bean UserEnv 总是先初始化一次, 由于这个 bean 存在周期是 seesion, 因此该用户第二次以后只要在 seesion 有效期内再调用,myenv.getUserdir()将直接从 bean 内存中读取变量,不必再初始化. 这样提高速度,又减少数据库访问量.

这样,我们就有了一个 jsp 程序之间共享变量或方法 的实现办法.

Reflect on the Visitor design pattern

Implement visitors in Java, using reflection

By Jeremy Blosser, JavaWorld.com, 07/14/00

Collections are commonly used in object-oriented programming and often raise code-related questions. For example, "How do you perform an operation across a collection of different objects?"

One approach is to iterate through each element in the collection and then do something specific to each element, based on its class. That can get pretty tricky, especially if you don't know what type of objects are in the collection. If you wanted to print out the elements in the collection, you could write a method like this:

```
public void messyPrintCollection(Collection collection) {
   Iterator iterator = collection.iterator()
   while (iterator.hasNext())
      System.out.println(iterator.next().toString())
}
```

That seems simple enough. You just call the `Object.toString()` method and print out the object, right? What if, for example, you have a vector of hashtables? Then things start to get more complicated. You must check the type of object returned from the collection:

```
public void messyPrintCollection(Collection collection) {
   Iterator iterator = collection.iterator()
   while (iterator.hasNext()) {
      Object o = iterator.next();
      if (o instanceof Collection)
         messyPrintCollection((Collection)o);
      else
         System.out.println(o.toString());
   }
}
```

OK, so now you have handled nested collections, but what about other objects that do not return the `String` that you need from them? What if you want to add quotes around `String` objects and add an f after `Float` objects? The code gets still more complex:

```
public void messyPrintCollection(Collection collection) {
   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());
   }
}
```

You can see that things can start to get intricate really fast. You don't want a piece of code with a huge list of if-else statements! How do you avoid that? The Visitor pattern comes to the rescue.

To implement the Visitor pattern, you create a `Visitor` interface for the visitor, and a `Visitable` interface for the collection to be visited. You then have concrete classes that implement the `Visitor` and `Visitable` interfaces. The two interfaces look like this:

```
public interface Visitor
{
   public void visitCollection(Collection collection);
   public void visitString(String string);
   public void visitFloat(Float float);
}
public interface Visitable
{
   public void accept(Visitor visitor);
}
```

For a concrete `String`, you might have:

```
public class VisitableString implements Visitable
{
   private String value;
   public VisitableString(String string) {
      value = string;
   }
   public void accept(Visitor visitor) {
      visitor.visitString(this);
   }
}
```

In the accept method, you call the correct visitor method for `this` type:

```
visitor.visitString(this)
```

That lets you implement a concrete `Visitor` as the following:

```
public class PrintVisitor implements Visitor
{
   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(String string) {
      System.out.println("'"+string+"'");
   }
   public void visitFloat(Float float) {
      System.out.println(float.toString()+"f");
   }
}
```

By then implementing a `VisitableFloat` class and a `VisitableCollection` class that each call the appropriate visitor

methods, you get the same result as the messy if-else `messyPrintCollection` method but with a much cleaner approach. In `visitCollection()`, you call `Visitable.accept(this)`, which in turn calls the correct visitor method. That is called a double-dispatch; the `Visitor` calls a method in the `Visitable` class, which calls back into the `Visitor` class.

Although you've cleaned up an if-else statement by implementing the visitor, you've still introduced a lot of extra code. You've had to wrap your original objects, `String` and `Float`, in objects implementing the `Visitable` interface. Although annoying, that is normally not a problem since the collections you are usually visiting can be made to contain only objects that implement the `Visitable` interface.

Still, it seems like a lot of extra work. Worse, what happens when you add a new `Visitable` type, say `VisitableInteger`? That is one major drawback of the Visitor pattern. If you want to add a new `Visitable` object, you have to change the `Visitor` interface and then implement that method in each of your `Visitor` implementation classes. You could use an abstract base class `Visitor` with default no-op functions instead of an interface. That would be similar to the `Adapter` classes in Java GUIs. The problem with that approach is that you need to use up your single inheritance, which you often want to save for something else, such as extending `StringWriter`. It would also limit you to only be able to visit `Visitable` objects successfully.

Luckily, Java lets you make the Visitor pattern much more flexible so you can add `Visitable` objects at will. How? The answer is by using reflection. With a `ReflectiveVisitor`, you only need one method in your interface:

```
public interface ReflectiveVisitor {
   public void visit(Object o);
}
```

OK, that was easy enough. `Visitable` can stay the same, and I'll get to that in a minute. For now, I'll implement `PrintVisitor` using reflection:

```
public class PrintVisitor implements ReflectiveVisitor {
   public void visitCollection(Collection collection)
   { ... same as above ... }
   public void visitString(String string)
   { ... same as above ... }
   public void visitFloat(Float float)
   { ... same as above ... }
   public void default(Object o)
   {
      System.out.println(o.toString());
   }
   public void visit(Object o) {
      // Class.getName() returns package information as well.
      // This strips off the package information giving us
      // just the class name
      String methodName = o.getClass().getName();
      methodName = "visit"+
                   methodName.substring(methodName.lastIndexOf('.')+1);
      // Now we try to invoke the method visit<methodName>
      try {
```

```
            // Get the method visitFoo(Foo foo)
            Method m = getClass().getMethod(methodName,
               new Class[] { o.getClass() });
            // Try to invoke visitFoo(Foo foo)
            m.invoke(this, new Object[] { o });
         } catch (NoSuchMethodException e) {
            // No method, so do the default implementation
            default(o);
         }
      }
   }
```

Now you don't need the `Visitable` wrapper class. You can just call `visit()`, and it will dispatch to the correct method. One nice aspect is that `visit()` can dispatch however it sees fit. It doesn't have to use reflection -- it can use a totally different mechanism.

With the new `PrintVisitor`, you have methods for `Collections`, `Strings`, and `Floats`, but then you catch all the unhandled types in the catch statement. You'll expand upon the `visit()` method so that you can try all the superclasses as well. First, you'll add a new method called `getMethod(Class c)` that will return the method to invoke, which looks for a matching method for all the superclasses of `Class c` and then all the interfaces for `Class c`.

```
protected Method getMethod(Class c) {
   Class newc = c;
   Method m = null;
   // Try the superclasses
   while (m == null && newc != Object.class) {
      String method = newc.getName();
      method = "visit" + method.substring(method.lastIndexOf('.') + 1);
      try {
         m = getClass().getMethod(method, new Class[] {newc});
      } catch (NoSuchMethodException e) {
         newc = newc.getSuperclass();
      }
   }
   // Try the interfaces.  If necessary, you
   // can sort them first to define 'visitable' interface wins
   // in case an object implements more than one.
   if (newc == Object.class) {
      Class[] interfaces = c.getInterfaces();
      for (int i = 0; i < interfaces.length; i++) {
         String method = interfaces[i].getName();
         method = "visit" + method.substring(method.lastIndexOf('.') + 1);
         try {
            m = getClass().getMethod(method, new Class[] {interfaces[i]});
         } catch (NoSuchMethodException e) {}
      }
```

```
    }
    if (m == null) {
        try {
            m = thisclass.getMethod("visitObject", new Class[] {Object.class});
        } catch (Exception e) {
            // Can't happen
        }
    }
    return m;
}
```

It looks complicated, but it really isn't. Basically, you just look for methods based on the name of the class you have passed in. If you don't find one, you try its superclasses. Then if you don't find any of those, you try any interfaces. Lastly, you can just try `visitObject()` as a default.

Note that for the sake of those familiar with the traditional Visitor pattern, I've followed the same naming convention for the method names. However, as some of you may have noticed, it would be more efficient to name all the methods "visit" and let the parameter type be the differentiator. If you do that, however, make sure you change the main `visit(Object o)` method name to something like `dispatch(Object o)`. Otherwise, you won't have a default method to fall back on, and you would need to cast to `Object` whenever you call `visit(Object o)` to assure the correct method calling pattern was followed.

Now, you modify the `visit()` method to take advantage of `getMethod()`:

```
public void visit(Object object) {
    try {
      Method method = getMethod(getClass(), object.getClass());
      method.invoke(this, new Object[] {object});
    } catch (Exception e) { }
}
```

Now, your visitor object is much more powerful. You can pass in any arbitrary object and have some method that uses it. Plus, you gain the added benefit of having a default method `visitObject(Object o)` that can catch anything you don't specify. With a little more work, you can even add a method for `visitNull()`.

I've kept the `Visitable` interface in there for a reason. Another side benefit of the traditional Visitor pattern is that it allows the `Visitable` objects to control navigation of the object structure. For example, if you had a `TreeNode` object that implemented `Visitable`, you could have an `accept()` method that traverses to its left and right nodes:

```
public void accept(Visitor visitor) {
    visitor.visitTreeNode(this);
    visitor.visitTreeNode(leftsubtree);
    visitor.visitTreeNode(rightsubtree);
}
```

So, with just one more modification to the `Visitor` class, you can allow for `Visitable`-controlled navigation:

```
public void visit(Object object) throws Exception
{
     Method method = getMethod(getClass(), object.getClass());
      method.invoke(this, new Object[] {object});
```

```
    if (object instanceof Visitable)
    {
        callAccept((Visitable) object);
    }
}
public void callAccept(Visitable visitable) {
   visitable.accept(this);
}
```

If you've implemented a `Visitable` object structure, you can keep the `callAccept()` method as is and use `Visitable`-controlled navigation. If you want to navigate the structure within the visitor, you just override the `callAccept()` method to do nothing.

The power of the Visitor pattern comes into play when using several different visitors across the same collection of objects. For example, I have an interpreter, an infix writer, a postfix writer, an XML writer, and a SQL writer working across the same collection of objects. I could easily write a prefix writer or a SOAP writer for the same collection of objects. In addition, those writers can gracefully work with objects they don't know about or, if I choose, they can throw an exception.

Author Bio

Jeremy Blosser has been programming in Java for five years, during which he has worked for various software companies. He now works for a startup company, Software Instruments. You can visit Jeremy's Website at http://www.blosser.org.

Conclusion

By using Java reflection, you can enhance the Visitor design pattern to provide a powerful way to operate on object structures, giving the flexibility to add new `Visitable` types as needed. I hope you are able to use that pattern somewhere in your coding travels.