# Anonymous的博客 - - I'm lovin' IT 追求心随意动

◎转载请注明我的博客链接 违者必究◎ 身体 心态 事业 .....



个人资料

rulinma

访问： 1742364次

积分： 25881

等级： BLOG 7

排名： 第181名

原创： 899篇 转载： 69篇

译文： 0篇 评论： 582条







## 设计模式大全

标签： 设计模式 decorator prototype 算法 command 产品

2006-12-22 21:07 71282人阅读 评论(12) 收藏 举报

分类：

软件工程、原理（49）



Longronglin之设计模式:

**Christopher Alexander** 说过：“每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”。

模式描述为：在一定环境中解决某一问题的方案，包括三个基本元素--问题，解决方案和环境。

阅读类图和对象图请先学习**UML**

创建模式 结构模式 行为模式

创建模式：对类的实例化过程的抽象。一些系统在创建对象时，需要动态地决定怎样创建对象，创建哪些对象，以及如何组合和表示这些对象。创建模式描述了怎样构造和封装这些动态的决定。包含类的创建模式和对象的创建模式。

结构模式：描述如何将类或对象结合在一起形成更大的结构。分为类的结构模式和对象的结构模式。类的结构模式使用继承把类，接口等组合在一起，以形成更大的结构。类的结构模式是静态的。对象的结构模式描述怎样把各种不同类型的对象组合在一起，以实现新的功能的方法。对象的结构模式是动态的。

行为模式：对在不同的对象之间划分责任和算法的抽象化。不仅仅是关于类和对象的，并是关于他们之间的相互作用。类的行为模式使用继承关系在几个类之间分配行为。对象的行为模式则使用对象的聚合来分配行为。

设计模式使用排行：

| 频率 | 所属类型 | 模式名称 | 模式 | 简单定义 |
|---|---|---|---|---|
| 5 | 创建型 | Singleton | 单件 | 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。 |
| 5 | 结构型 | Composite | 组合模式 | 将对象组合成树形结构以表示部分整体的关系，Composite使得用户对单个对象和组合对象的使用具有一致性。 |
| 5 | 结构型 | FAÇADE | 外观 | 为子系统中的一组接口提供一致的界面，facade提供了一高层接口，这个接口使得子系统更容易使用。 |
| 5 | 结构型 | Proxy | 代理 | 为其他对象提供一种代理以控制对这个对象的访问 |
| 5 | 行为型 | Iterator | 迭代器 | 提供一个方法顺序访问一个聚合对象的各个元素，而又不需要暴露该对象的内部表示。 |
| 5 | 行为型 | Observer | 观察者 | 定义对象间一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知自动更新。 |
| 5 | 行为型 | Template Method | 模板方法 | 定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，Template Method使得子类可以不改变一个算法的结构即可以重定义该算法得某些特定步骤。 |
| 4 | 创建型 | Abstract Factory | 抽象工厂 | 提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们的具体类。 |
| 4 | 创建型 | Factory Method | 工厂方法 | 定义一个用于创建对象的接口，让子类决定实例化哪一个类，Factory Method使一个类的实例化延迟到了子类。 |
| 4 | 结构型 | Adapter | 适配器 | 将一类的接口转换成客户希望的另外一个接口，Adapter模式使得原本由于接口不兼容而不能一起工作那些类可以一起工作。 |
| | | | | 动态地给一个对象增加一些额外的职责，就增加的功能来 |

编译原理 (5)
计算机历史 (4)
论文相关 (21)
软件工程、原理 (50)
面试 (4)
PHP (1)
Python (0)
添物 计算机 (17)
添物 c语言 (1)
添物 c语言 编程 (0)

推荐文章

* Android 反编译初探 应用是如何被注入广告的
* 凭兴趣求职80%会失败，为什么
* 安卓微信自动抢红包插件优化和实现
* 【游戏设计模式】之四 《游戏编程模式》全书内容提炼总结
* 带你开发一款给Apk中自动注入代码工具icodetools(完善篇)

最新评论

Spring的IOC简单实例
一根葱的无奈: 博客中的"发射机制"应该改为"反射机制"
weblogic开发EJB
Dwyane-Kwok: 看不明白
个人创业了，做了个网站和App，游离于移动互联网边缘: 第二个csdn?
数据字典的设计
马金兴: 学习了，举的例子很详细
数据字典的设计
飞s羽u逐n魂: 很详细的总结，看了后一目了然
数据字典的设计
生命奇迹泉: 感觉有点明白了
数据字典的设计
李亚松-: +1
编译原理解析
sinat_27307123: 请输入源程序文件名（包括路径）：K:\1.txt请输入词法分析输出文件名（包括路径）：1.txt词法...
编译原理解析
sinat_27307123: 你好 我代码完全复制你的 为啥词法分析错误 我用的VS2013 谢谢
设计模式大全
熊猫小牛牛: 总结的太好了，我转载下，留着看

文章存档

2016年08月 (6)
2016年07月 (16)
2016年06月 (13)
2015年12月 (1)
2015年07月 (1)
展开

阅读排行

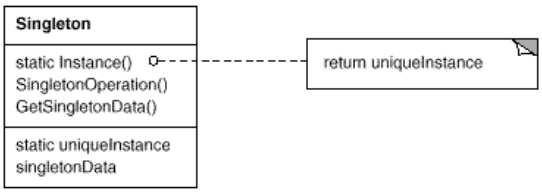
设计模式大全 (71273)
ER图实例解析 (60079)
数据字典的设计 (33991)
Java读取文件内容并转换 (29704)
JavaEE程序员必读图书J

| | | | | |
|---|---|---|---|---|
| 4 | 结构型 | Decorator | 装饰 | 说，Decorator模式相比生成子类更加灵活。 |
| 4 | 行为型 | Command | 命令 | 将一个请求封装为一个对象，从而使你可以用不同的请求对客户进行参数化，对请求排队和记录请求日志，以及支持可撤销的操作。 |
| 4 | 行为型 | State | 状态 | 允许对象在其内部状态改变时改变他的行为。对象看起来似乎改变了他的类。 |
| 4 | 行为型 | Strategy | 策略模式 | 定义一系列的算法，把他们一个个封装起来，并使他们可以互相替换，本模式使得算法可以独立于使用它们的客户。 |
| 3 | 创建型 | Builder | 生成器 | 将一个复杂对象的构建与他的表示相分离，使得同样的构建过程可以创建不同的表示。 |
| 3 | 结构型 | Bridge | 桥接 | 将抽象部分与它的实现部分相分离，使他们可以独立的变化。 |
| 3 | 行为型 | China of Responsibility | 职责链 | 使多个对象都有机会处理请求，从而避免请求的送发者和接收者之间的耦合关系 |
| 2 | 创建型 | 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来创建新的对象。 |
| 2 | 结构型 | Flyweight | 享元 | 享元模式以共享的方式高效的支持大量的细粒度对象。享元模式能做到共享的关键是区分内蕴状态和外蕴状态。内蕴状态存储在享元内部，不会随环境的改变而有所不同。外蕴状态是随环境的改变而改变的。 |
| 2 | 行为型 | Mediator | 中介者 | 用一个中介对象封装一些列的对象交互。 |
| 2 | 行为型 | Visitor | 访问者模式 | 表示一个作用于某对象结构中的各元素的操作，它使你可以在不改变各元素类的前提下定义作用于这个元素的新操作。 |
| 1 | 行为型 | Interpreter | 解释器 | 给定一个语言，定义他的文法的一个表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。 |
| 1 | 行为型 | Memento | 备忘录 | 在不破坏对象的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。 |

一： 单例模式**(Singleton)**

单例模式：Singleton的作用是保证在应用程序中，一个类Class只有一个实例存在。并提供全局访问。

结构：




账本类：**1** 单一实例 **2** 给多个对象共享 **3** 自己创建

网页计数器

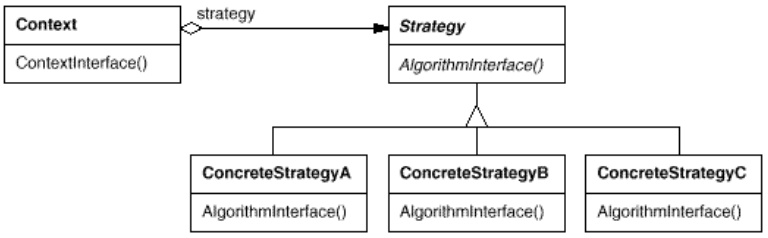
```
public class LazySingleton
{
    private static LazySingleton newInstance = null;
        private LazySingleton ()
        {
        }
         public static synchronized  LazySingleton getInstance ()
        {
           if (newInstance == null)
                    {
                newInstance = new LazySingleton ();
            }
            return newInstance;
       }
}
```

singleton限制了实例个数，有利于gc的回收。

二：策略模式**(Strategy)**

策略模式：策略模式针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。策略模式把行为和环境分开。环境类负责维持和查询行为类，各种算法在具体的策略类中提供。由于算法和环境独立开来，算法的增减，修改都不会影响到环境和客户端。

结构：



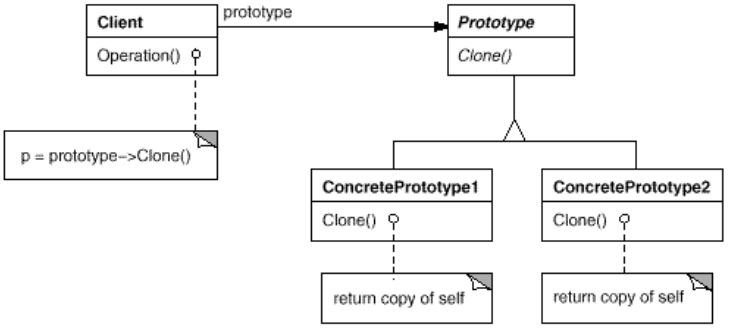

使用QQ泡MM时使用外挂 客户端：ME抽象类： 外挂 具体：策略（图片，笑话，名人名言）

图书销售算法（不同书本折扣的算法）

三：原型模式(Prototype)

原型模式：通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的方法创建出更多同类型的对象。原始模型模式允许动态的增加或减少产品类，产品类不需要非得有任何事先确定的等级结构，原始模型模式适用于任何的等级结构。缺点是每一个类都必须配备一个克隆方法

结构：




复印技术： 1 不是同一个对象 2 属同类

短消息（转发） 1-n个MM

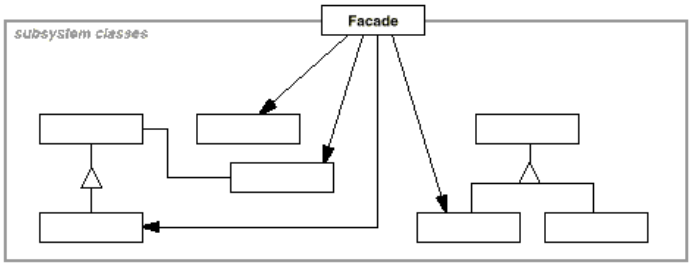
因为Java中的提供clone()方法来实现对象的克隆,所以Prototype模式实现一下子变得很简单.

四：门面模式(Façade)

门面模式：外部与一个子系统的通信必须通过一个统一的门面对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用，减少复杂性。每一个子系统只有一个门面类，而且此门面类只有一个实例，也就是说它是一个单例模式。但整个系统可以有多个门面类。

1 门面角色 2 子系统角色

结构：




Facade典型应用就是数据库JDBC的应用和Session的应用

ME---àMM---à(father,mum,sister,brother)

五：备忘录模式(Memento)

Memento模式：Memento对象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。模式的用意是在不破坏封装的条件下，将一个对象的状态捕捉住，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。模式所涉及的角色有三个，备忘录角色、发起人角色和负责人角色。

备忘录角色的作用：

（1） 将发起人对象的内部状态存储起来，备忘录可以根据发起人对象的判断来决定存储多少发起人对象的内部状态。

（2） 备忘录可以保护其内容不被发起人对象之外的任何对象所读取。

发起人角色的作用：

（1） 创建一个含有当前内部状态的备忘录对象。

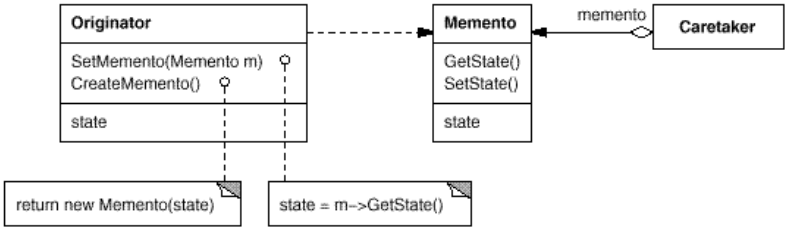
（2） 使用备忘录对象存储其内部状态。

负责人角色的作用：

（1） 负责保存备忘录对象。

（2） 不检查备忘录对象的内容

结构：



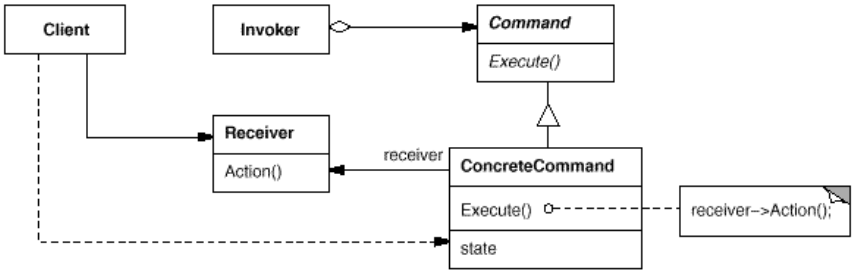

备份系统时使用

**GHOST**

六： 命令模式（**Command**）

命令模式：命令模式把一个请求或者操作封装到一个对象中。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。命令模式允许请求的一方和发送的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否执行，何时被执行以及是怎么被执行的。系统支持命令的撤消。

结构：



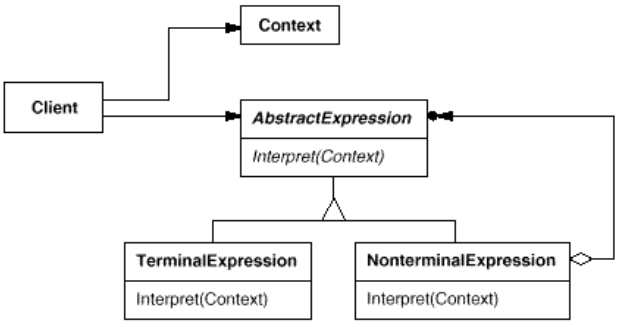

**MM**（客户端）**--àME**（请求者）**--à**命令角色**--à**（具体命令）**-à**代理处（接收者）**--àMM**

上网 **IE**输入 **http**地址 发送命令

七： 解释器**(Interpreter)**

解释器模式：给定一个语言后，解释器模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器来解释这个语言中的句子。解释器模式将描述怎样在有了一个简单的文法后，使用模式设计解释这些语句。在解释器模式里面提到的语言是指任何解释器对象能够解释的任何组合。在解释器模式中需要定义一个代表文法的命令类的等级结构，也就是一系列的组合规则。每一个命令对象都有一个解释方法，代表对命令对象的解释。命令对象的等级结构中的对象的任何排列组合都是一个语言。

结构：



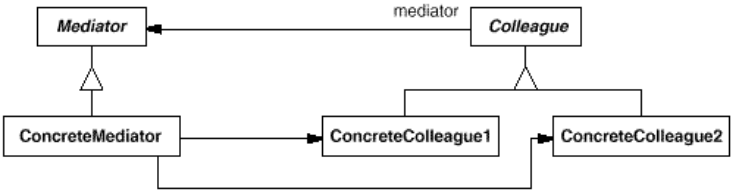

编译原理之编译器

文言文注释：一段文言文，将它翻译成白话文

八：调停者模式**(Mediator)**

调停者模式：包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用。从而使他们可以松散偶合。当某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用。保证这些作用可以彼此独立的变化。调停者模式将多对多的相互作用转化为一对多的相互作用。调停者模式将对象的行为和协作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

结构：

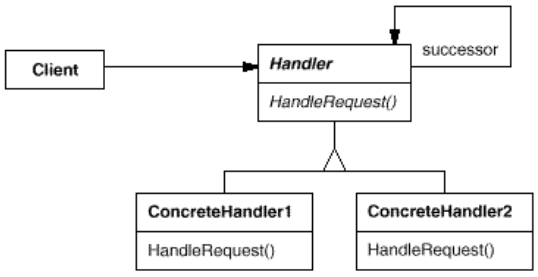



法院和原告，被告的关系

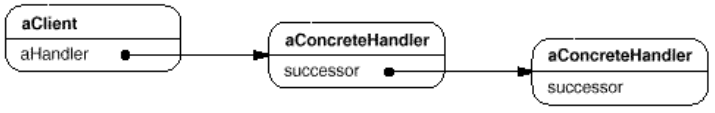
九：责任链模式**(CHAIN OF RESPONSIBLEITY)**

责任链模式：执行者的不确定性 在责任链模式中，很多对象由每一个对象对其下家的引用而接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。客户并不知道链上的哪一个对象最终处理这个请求，系统可以在不影响客户端的情况下动态的重新组织链和分配责任。处理者有两个选择：承担责任或者把责任推给下家。一个请求可以最终不被任何接收端对象所接受。

结构：




典型的对象结构：

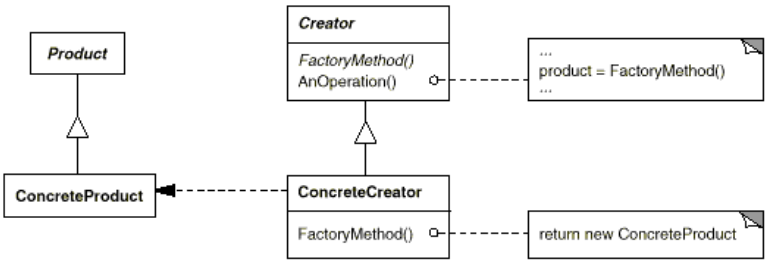



喝酒时通过成语接龙决定谁喝酒**(**马到成功－功不可没－没完没了**)**

十：工厂模式（**Factory**）

工厂模式： 定义一个用于创建对象的接口，让接口子类通过工厂方法决定实例化哪一个类。

结构：

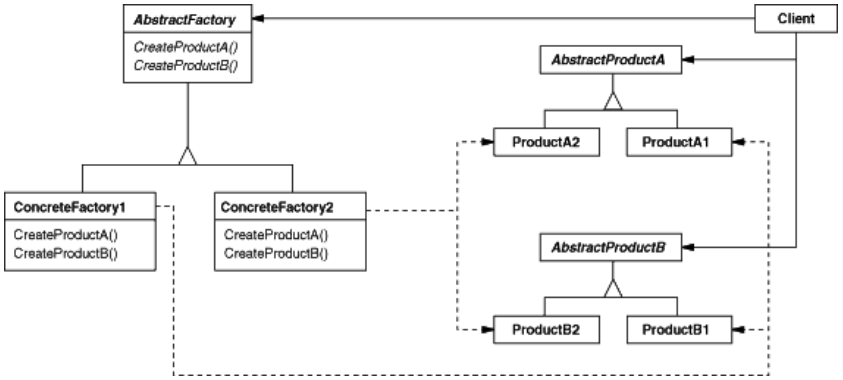



水果园—〉（葡萄园，苹果园）--〉（葡萄，苹果）（各自生产）

十一：抽象工厂模式（**Abstract Factory**）

抽象工厂模式：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

结构：




女娲造人---〉（阴，阳）--〉（人，兽）----〉（男人，女人，公兽，母兽）**(**人和兽属于不同的产品类**)**

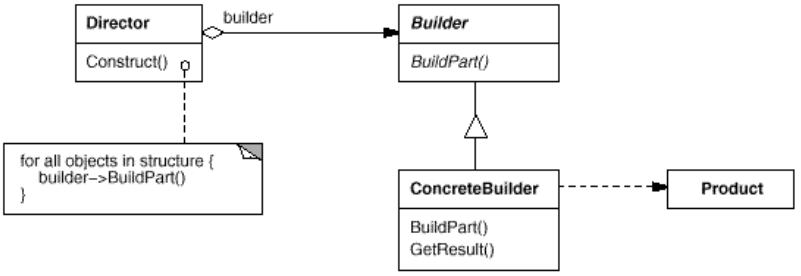
十二：建造模式（**Builder**）

建造模式：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。

将产品的内部表象和产品的生成过程分割开来，从而使一个建造过程生成具有不同的内部表象的产品对象。建造模式使得产品内部表象可以独立的变化，客户不必知道产品内部组成的细节。建造模式可以强制实行一种分步骤进行的建造过程。

结构：



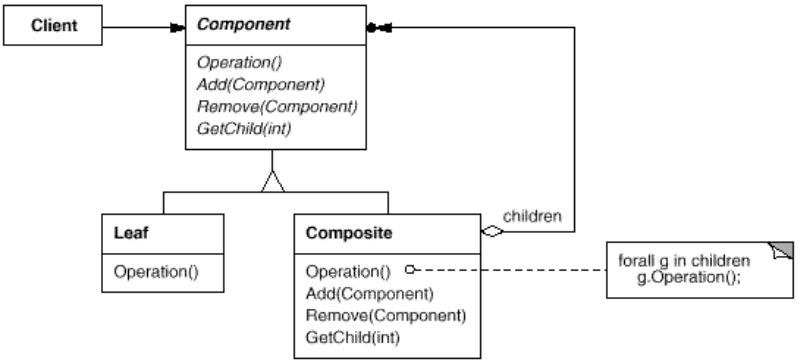

交互图：

汽车制造

十三：合成模式（**Composite**）

合成模式： 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.合成模式就是一个处理对象的树结构的模式。合成模式把部分与整体的关系用树结构表示出来。合成模式使得客户端把一个个单独的成分对象和由他们复合而成的合成对象同等看待。

结构：



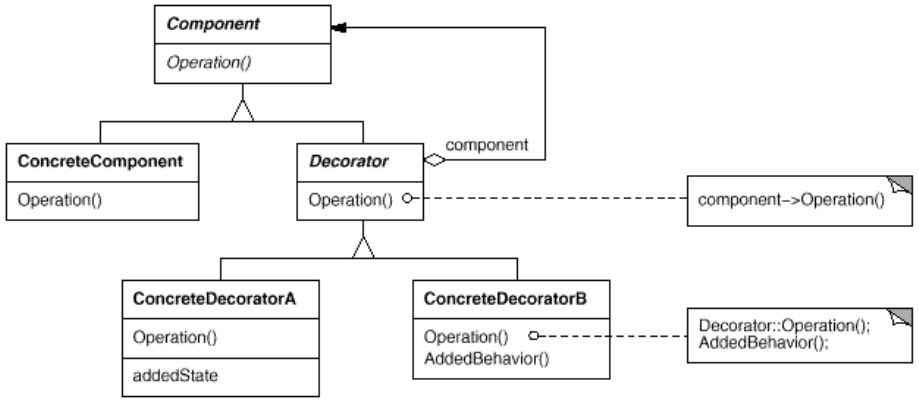

**windows**的目录树（文件系统）

十四：装饰模式（**DECORATOR**）

装饰模式：装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案，提供比继承更多的灵活性。动态给一个对象增加功能，这些功能可以再动态的撤消。增加由一些基本功能的排列组合而产生的非常大量的功能。

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.

结构：




在**visio**中文件可以使用背景进行装饰

变废为宝

十五：设计模式之**Adapter(**适配器**)**

适配器模式**:**把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口原因不匹配而无法一起工作的两个类能够一起

工作。适配类可以根据参数返还一个合适的实例给客户端

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

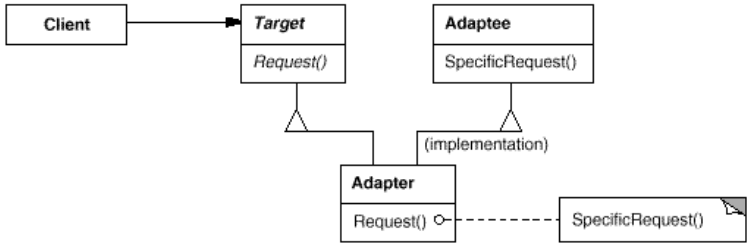
为何使用?

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).

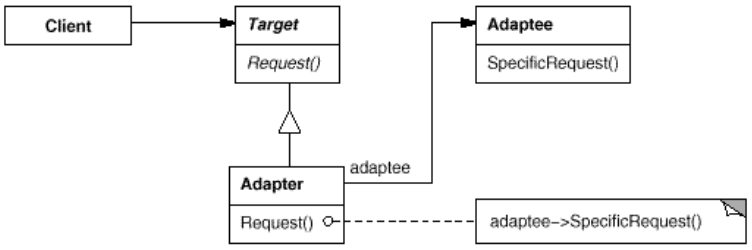
如何使用?

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).

结构：




对象结构：



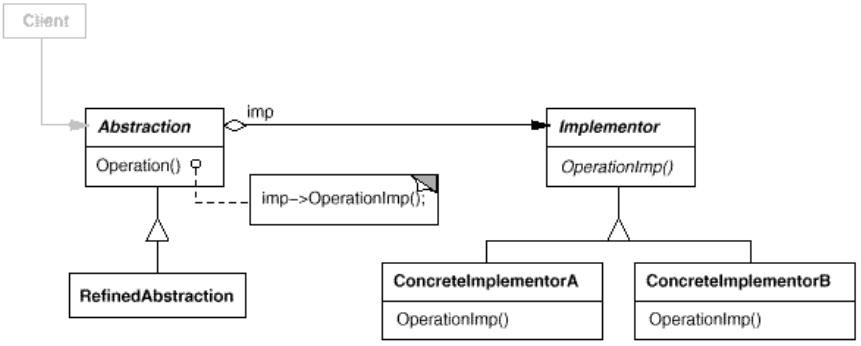

充电器（手机和220V电压）

jdbc-odbc桥

十六：桥梁模式（Bridge）

桥梁模式：将抽象化与实现化脱耦，使得二者可以独立的变化。也就是说将他们之间的强关联变成弱关联，也就是指在一个软件系统的抽象化和实现化之间使用组合/聚合关系而不是继承关系，从而使两者可以独立的变化。

结构：

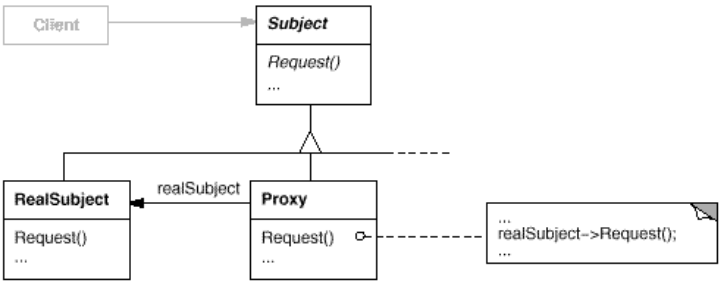



jdbc驱动程序

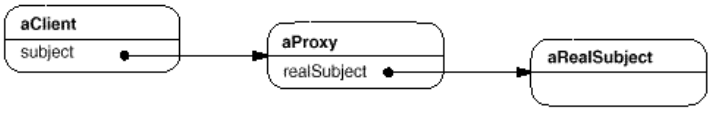
十七：代理模式（Proxy）

代理模式：代理模式给某一个对象提供一个代理对象，并由代理对象控制对源对象的引用。代理就是一个人或一个机构代表另一个人或者一个机构采取行动。某些情况下，客户不想或者不能够直接引用一个对象，代理对象可以在客户和目标对象直接起到中介的作用。客户端分辨不出代理主题对象与真实主题对象。代理模式可以并不知道真正的被代理对象，而仅仅持有一个被代理对象的接口，这时候代理对象不能够创建被代理对象，被代理对象必须有系统的其他角色代为创建并传入。

结构：




运行时的代理结构：




用代理服务器连接出网

销售代理（厂商）律师代理（客户）

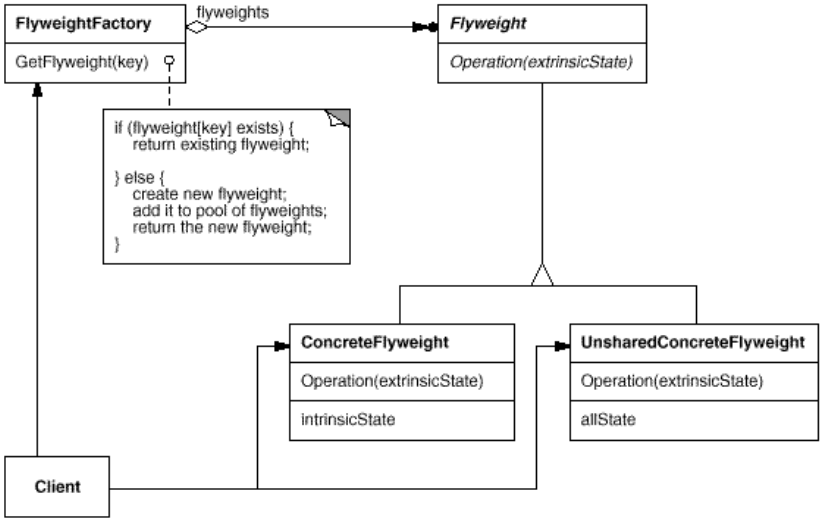
**foxmail**

枪手

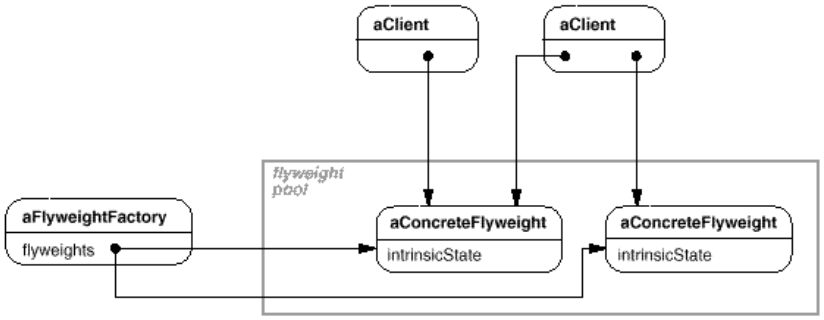
十八：享元模式（**Flyweight**）

享元模式以共享的方式高效的支持大量的细粒度对象。享元模式能做到共享的关键是区分内蕴状态和外蕴状态。内蕴状态存储在享元内部，不会随环境的改变而有所不同。外蕴状态是随环境的改变而改变的。外蕴状态不能影响内蕴状态，它们是相互独立的。将可以共享的状态和不可以共享的状态从常规类中区分开来，将不可以共享的状态从类里剔除出去。客户端不可以直接创建被共享的对象，而应当使用一个工厂对象负责创建被共享的对象。享元模式大幅度的降低内存中对象的数量。

结构：




共享方法：

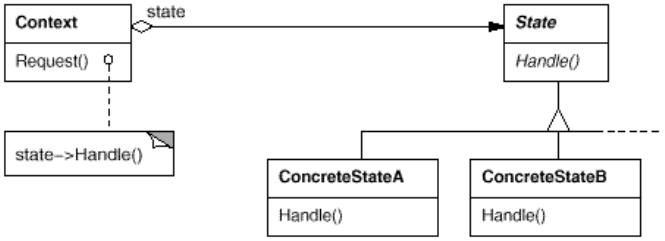



字体的**26**个字母和各自的斜体等

十九：状态模式（**State**）

状态模式：状态模式允许一个对象在其内部状态改变的时候改变行为。这个对象看上去象是改变了它的类一样。状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。状态模式需要对每一个系统可能取得的状态创立一个状态类的子类。当系统的状态变化时，系统便改变所选的子类。

结构：




人心情不同时表现不同有不同的行为

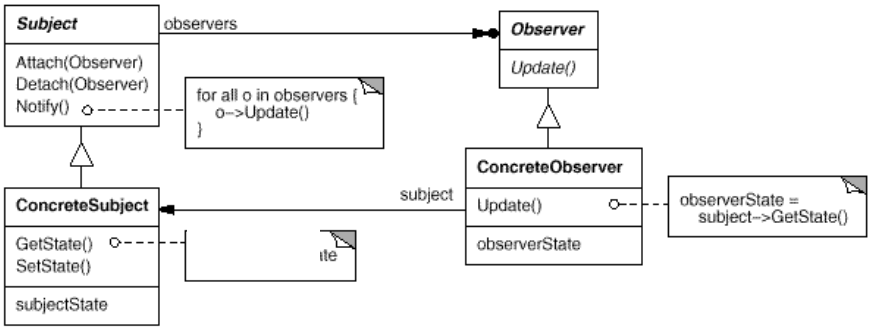
编钟

登录**login logout**

二十：观察者模式（**Observer**）

观察者模式：观察者模式定义了一种一队多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使他们能够自动更新自己。发布订阅。

结构：



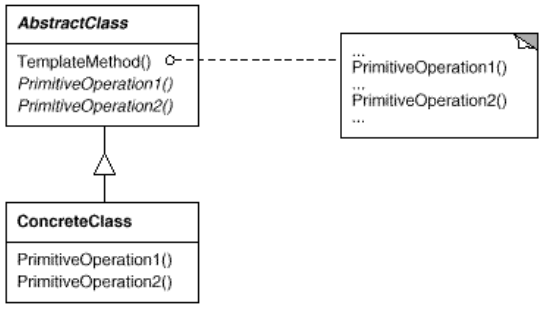

公司邮件系统**everyone@sina.com**的应用。当公司员工向这个邮箱发邮件时会发给公司的每一个员工。如果设置了**Outlook**则会及时收到通知。

接收到短消息

二十一：模板方法模式（**Template**）

模板方法模式：模板方法模式准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。先制定一个顶级逻辑框架，而将逻辑的细节留给具体的子类去实现。

结构：

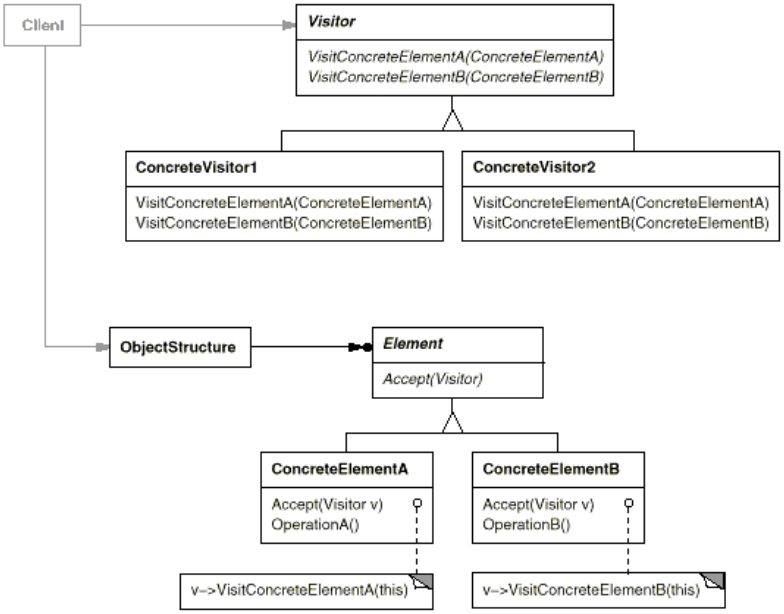



使用网页设计时使用的模板架构网页（骨架） 算法的各个逻辑系统

二十二：访问者模式（**Visitor**）

访问者模式：访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构可以保持不变。访问者模式适用于数据结构相对未定的系统，它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对自由的演化。访问者模式使得增加新的操作变的很容易，就是增加一个新的访问者类。访问者模式将有关的行为集中到一个访问者对象中，而不是分散到一个个的节点类中。当使用访问者模式时，要将尽可能多的对象浏览逻辑放在访问者类中，而不是放到它的子类中。访问者模式可以跨过几个类的等级结构访问属于不同的等级结构的成员类。

结构：

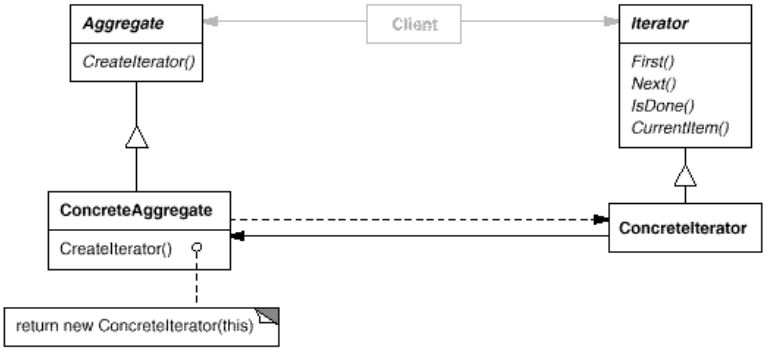



电脑销售系统： 访问者（自己）---〉电脑配置系统（主板，**CPU**，内存。。。。。。）

二十三：迭代子模式（**Iterator**）

迭代子模式：迭代子模式可以顺序访问一个聚集中的元素而不必暴露聚集的内部表象。多个对象聚在一起形成的总体称之为聚集，聚集对象是能够包容一组对象的容器对象。迭代子模式将迭代逻辑封装到一个独立的子对象中，从而与聚集本身隔开。迭代子模式简化了聚集的界面。每一个聚集对象都可以有一个或一个以上的迭代子对象，每一个迭代子的迭代状态可以是彼此独立的。迭代算法可以独立于聚集角色变化。

结构：

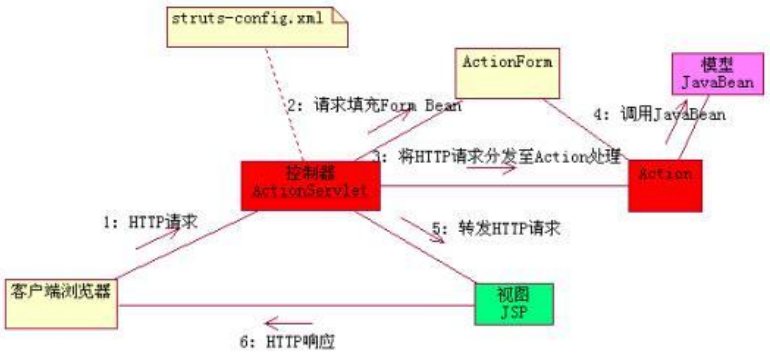



查询数据库，返回结果集（**map**， **list**， **set**）

二十四：**MVC**模式

MVC模式：它强制性的使应用程序的输入、处理和输出分开。使用MVC应用程序被分成三个核心部件：模型、视图、控制器。它们各自处理自己的任务。相互通信。

MVC还使用了的设计模式，如：用来指定视图缺省控制器的Factory Method和用来增加视图滚动的Decorator。但是MVC的主要关系还是由Observer、Composite和Strategy三个设计模式给出的。

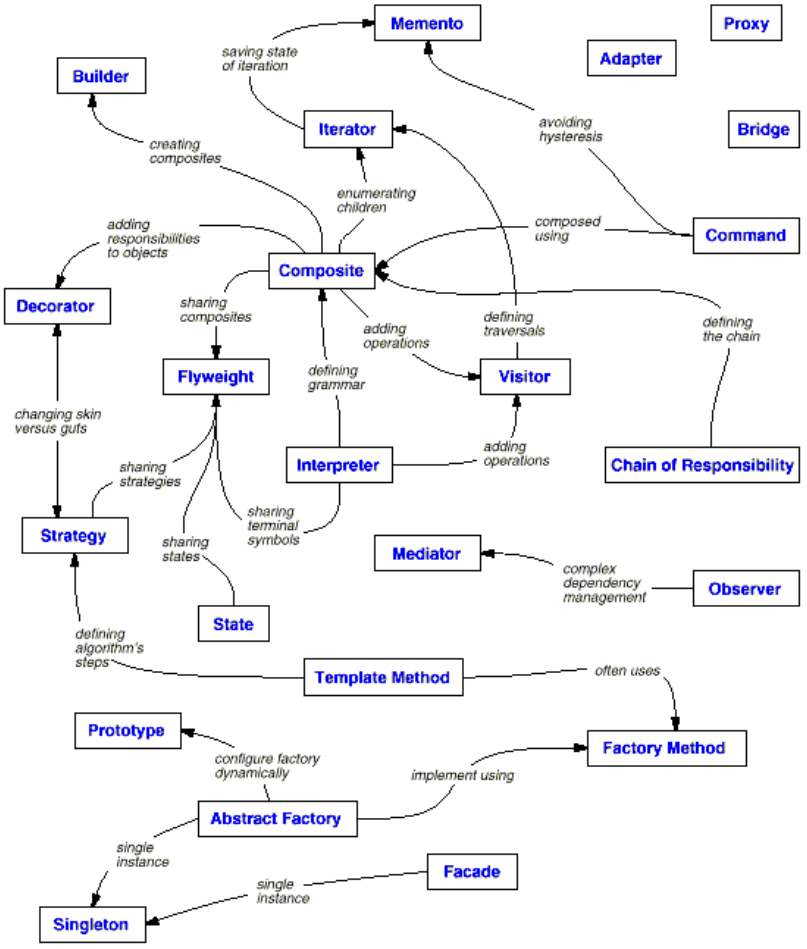



struts图解：其中不同颜色代表MVC的不同部分：红色（控制器）、紫色（模型）和绿色（视图）

**struts**应用 **spring** 应用

设计模式的使用：

模式关系图:




个人图解：（^_^）没有看到下面的图解时想的

门面模式可以使用一个单体实例对象实现

抽象工厂可以创建单体实例 也可以使用工厂方法也可以使用原型创建对象实例

模板方法可以使用工厂方法实现创建实例使用策略模式定义算法使用

策略模式可以使用享元实例 与装饰模式可以相互使用

享元模式被状态，解释器，合成等模式。共享

解释器模式通过访问模式实现其动作 通过享元实现基本元素的共享

装饰模式使用策略可以实现不同的装饰效果

迭代器模式通过访问者访问对象元素 通过备忘录模式实现纪录的记忆功能 访问合成的对象

命令模式通过使用备忘录模式（参考） 执行命令

建造模式可以使用合成模式创建合成产品

责任链模式使用合成模式定义链

调停者模式可以使观察者的观察受其影响

实际图解：

关模式（相互关系）：

Abstract Factory类通常用工厂方法（Factory Method）实现，但它们也可以用Prototype实现。一个具体的工厂通常是一个单件Singleton。Abstract Factory与Builder相似，因为它也可以创建复杂对象。主要的区别是Builder模式着重于一步步构造一个复杂对象。而Abstract Factory着重于多个系列的产品对象（简单的或是复杂的）。Builder在最后一步返回产品，而对于Abstract Factory来说，产品是立即返回的。Composite通常是用Builder生成的。

Factory方法通常在Template Methods中被调用。Prototypes不需要创建Creator的子类。但是，它们通常要求一个针对Product类的Initialize操作。Creator使用Initialize来初始化对象。Factory Method不需要这样的操作。多态迭代器靠Factory Method来例化适当的迭代器子类。Factory Method模式常被模板方法调用。

Prototype和Abstract Factory模式在某种方面是相互竞争的。但是它们也可以一起使用。Abstract Factory可以存储一个被克隆的原型的集合，并且返回产品对象。大量使用Composite和Decorator模式的设计通常也可从Prototype模式处获益。

很多模式可以使用Singleton模式实现。参见Abstract Factory、Builder，和Prototype。

模式Bridge的结构与对象适配器类似，但是Bridge模式的出发点不同：Bridge目的是将接口部分和实现部分分离，从而对它们可以较为容易也相对独立的加以改变。而Adapter则意味着改变一个已有对象的接口。

Decorator模式增强了其他对象的功能而同时又不改变它的接口。因此Decorator对应用程序的透明性比适配器要好。结果是Decorator支持递归组合，而纯粹使用适配器是不可能实现这一点的。模式Proxy在不改变它的接口的条件下，为另一个对象定义了一个代理。Abstract Factory模式可以用来创建和配置一个特定的Bridge模式。

Adapter模式用来帮助无关的类协同工作，它通常在系统设计完成后才会被使用。然而，Bridge模式则是在系统开始时就被使用，它使得抽象接口和实现部分可以独立进行改变。适配器Adapter为它所适配的对象提供了一个不同的接口。相反，代理提供了与它的实体相同的接口。然而，用于访问保护的代理可能会拒绝执行实体会执行的操作，因此，它的接口实际上可能只是实体接口的一个子集。

Decorator模式经常与Composite模式一起使用。当装饰和组合一起使用时，它们通常有一个公共的父类。因此装饰必须支持具有Add、Remove和GetChild 操作。Decorator模式不同于Adapter模式，因为装饰仅改变对象的职责而不改变它的接口；而适配器将给对象一个全新的接口。Composite模式可以将装饰视为一个退化的、仅有一个组件的组合。然而，装饰仅给对象添加一些额外的职责—它的目的不在于对象聚集。用一个装饰你可以改变对象的外表；而Strategy模式使得你可以改变对象的内核。这是改变对象的两种途径。

尽管Decorator的实现部分与Proxy相似，但Decorator的目的不一样。Decorator为对象添加一个或多个功能，而代理则控制对对象的访问。代理的实现Decorator的实现类似，但是在相似的程度上有所差别。Protection Proxy的实现可能与Decorator的实现差不多。另一方面， Remote Proxy不包含对实体的直接引用，而只是一个间接引用，如"主机I D，主机上的局部地址。"Virtual Proxy开始的时候使用一个间接引用，例如一个文件名，但最终将获取并使用一个直接引用。

Abstract Factory模式可以与Facade模式一起使用以提供一个接口，这一接口可用来以一种子系统独立的方式创建子系统对象。Abstract Factory也可以代替Facade模式隐藏那些与平台相关的类。Mediator模式与Facade模式的相似之处是，它抽象了一些已有的类的功能。然而，Mediator的目的是对同事之间的任意通讯进行抽象，通常集中不属于任何单个对象的功能。Mediator的同事对象知道中介者并与它通信，而不是直接与其他同类对象通信。相对而言，Facade模式仅对子系统对象的接口进行抽象，从而使它们更容易使用；它并不定义新功能，子系统也不知道Facade的存在。通常来讲，仅需要一个Facade对象，因此Facade对象通常属于Singleton模式

Chain of Responsibility常与Composite一起使用。这种情况下，一个构件的父构件可作为它的后继。

Composite抽象语法树是一个复合模式的实例。Composite模式可被用来实现宏命令。

Memento可用来保持某个状态，命令用这一状态来取消它的效果。在被放入历史表列前必须被拷贝的命令起到一种原型的作用。Memento常与迭代器模式一起使用。迭代器可使用一个Memento来捕获一个迭代的状态。迭代器在其内部存储Memento。

Flyweight说明了如何在抽象语法树中共享终结符。

Iterator解释器可用一个迭代器遍历该结构。

Visitor可用来在一个类中维护抽象语法树中的各节点的行为。访问者可以用于对一个由Composite模式定义的对象结构进行操作。迭代器常被应用到象复合这样的递归结构上。

Facade与中介者的不同之处在于它是对一个对象子系统进行抽象，从而提供了一个更为方便的接口。它的协议是单向的，即Facade对象对这个子系统类提出请求，但反之则不行。相反， Mediator提供了各Colleague对象不支持或不能支持的协作行为，而且协议是多向的。Colleague可使用Observer模式与Mediator通信。

Command命令可使用备忘录来为可撤消的操作维护状态。如前所述备忘录可用于迭代。

Mediator通过封装复杂的更新语义。

Singleton使用Singleton模式来保证它是唯一的并且是可全局访问的。

Flyweight解释了何时以及怎样共享状态对象。状态对象通常是Singleton。Strategy对象经常是很好的轻量级对象。

Strategy模板方法使用继承来改变算法的一部分。Strategy使用委托来改变整个算法。

Interpreter访问者可以用于解释。

创建型模式的讨论

用一个系统创建的那些对象的类对系统进行参数化有两种常用方法。一种是生成创建对象的类的子类；这对应于使用Factory Method模式。这种方法的主要缺点是，仅为了改变产品类，就可能需要创建一个新的子类。这样的改变可能是级联的（Cascade）。例如，如果产品的创建者本身是由一个工厂方法创建的，那么你也必须重定义它的创建者。另一种对系统进行参数化的方法更多的依赖于对象复合：定义一个对象负责明确产品对象的类，并将它作为该系统的参数。这是Abstract Factory、Builder和Prototype模式的关键特征。所有这三个模式都涉及到创建一个新的负责创建产品对象的"工厂对象"。Abstract Factory由

这个工厂对象产生多个类的对象。Builder由这个工厂对象使用一个相对复杂的协议，逐步创建一个复杂产品。Prototype由该工厂对象通过拷贝原型对象来创建产品对象。在这种情况下，因为原型负责返回产品对象，所以工厂对象和原型是同一个对象。

结构型模式的讨论

你可能已经注意到了结构型模式之间的相似性，尤其是它们的参与者和协作之间的相似性。这可能是因为结构型模式依赖于同一个很小的语言机制集合构造代码和对象：单继承和多重继承机制用于基于类的模式，而对象组合机制用于对象式模式。但是这些相似性掩盖了这些模式的不同意图。在本节中，我们将对比这些结构型模式，使你对它们各自的优点有所了解。

Adapter与Bridge

Adapter模式和Bridge模式具有一些共同的特征。它们都给另一对象提供了一定程度上的间接性，因而有利于系统的灵活性。它们都涉及到从自身以外的一个接口向这个对象转发请求。这些模式的不同之处主要在于它们各自的用途。Bridge模式主要是为了解决两个已有接口之间不匹配的问题。它不考虑这些接口是怎样实现的，也不考虑它们各自可能会如何演化。这种方式不需要对两个独立设计的类中的任一个进行重新设计，就能够使它们协同工作。另一方面， Bridge模式则对抽象接口与它的（可能是多个）实现部分进行桥接。虽然这一模式允许你修改实现它的类，它仍然为用户提供了一个稳定的接口。Bridge模式也会在系统演化时适应新的实现。由于这些不同点， Adapter和Bridge模式通常被用于软件生命周期的不同阶段。当你发现两个不兼容的类必须同时工作时，就有必要使用Adapter模式，其目的一般是为了避免代码重复。此处耦合不可预见。相反， Bridge的使用者必须事先知道：一个抽象将有多个实现部分，并且抽象和实现两者是独立演化的。Adapter模式在类已经设计好后实施；而Bridge模式在设计类之前实施。这并不意味着Adapter模式不如Bridge模式，只是因为它们针对了不同的问题。你可能认为facade是另外一组对象的适配器。但这种解释忽视了一个事实：即facade定义一个新的接口，而Adapter则复用一个原有的接口。记住，适配器使两个已有的接口协同工作，而不是定义一个全新的接口。

Composite、Decorator与Proxy

Composite模式和Decorator模式具有类似的结构图，这说明它们都基于递归组合来组织可变数目的对象。这一共同点可能会使你认为，Decorator对象是一个退化的Composite，但这一观点没有领会Decorator模式要点。相似点仅止于递归组合，同样，这是因为这两个模式的目的不同。Decorator 旨在使你能够不需要生成子类即可给对象添加职责。这就避免了静态实现所有功能组合，从而导致子类急剧增加。Composite则有不同的目的，它旨在构造类，使多个相关的对象能够以统一的方式处理，而多重对象可以被当作一个对象来处理。它重点不在于修饰，而在于表示。尽管它们的目的截然不同，但却具有互补性。因此Composite 和Decorator模式通常协同使用。在使用这两种模式进行设计时，我们无需定义新的类，仅需将一些对象插接在一起即可构建应用。这时系统中将会有一个抽象类，它有一些Composite子类和Decorator子类，还有

一些实现系统的基本构建模块。此时， composites 和decorator将拥有共同的接口。从Decorator模式的角度看，Composite是一个ConcreteComponet。而从Composite模式的角度看，Decorator则是一个leaf。当然，他们不一定要同时使用，正如我们所见，它们的目的有很大的差别。

另一种与Decorator模式结构相似的模式是Proxy这两种模式都描述了怎样为对象提供一定程度上的间接引用，proxy 和Decorator对象的实现部分都保留了指向另一个对象的指针，它们向这个对象发送请求。然而同样，它们具有不同的设计目的。像Decorator模式一样， Proxy 模式构成一个对象并为用户提供一致的接口。但与Decorator模式不同的是， Proxy 模式不能动态地添加或分离性质，它也不是为递归组合而设

计的。它的目的是，当直接访问一个实体不方便或不符合需要时，为这个实体提供一个替代者，例如，实体在远程设备上，访问受到限制或者实体是持久存储的。在Proxy模式中，实体定义了关键功能，而Proxy 提供（或拒绝）对它的访问。在Decorator模式中，组件仅提供了部分功能，而一个或多个Decorator负责完成其他功能。Decorator模式适用于编译时不能（至少不方便）确定对象的全部功能的情况。这种开放性使

递归组合成为Decorator模式中一个必不可少的部分。而在Proxy模式中则不是这样，因为Proxy模式强调一种关系（Proxy与它的实体之间的关系），这种关系可以静态的表达。模式间的这些差异非常重要，因为它们针对了面向对象设计过程中一些特定的经常发生问题的解决方法。但这并不意味着这些模式不能结合使用。可以设想有一个Proxy - Decorator，它可以给Proxy添加功能，或是一个Proxy - Proxy用来修饰一个远程对象。尽管这种混合可能有用（我们手边还没有现成的例子），但它们可以分割成一些有用的模式。

行为模式的讨论

封装变化

封装变化是很多行为模式的主题。当一个程序的某个方面的特征经常发生改变时，这些模式就定义一个封装这个方面的对象。这样当该程序的其他部分依赖于这个方面时，它们都可以与此对象协作。这些模式通常定义一个抽象类来描述这些封装变化的对象，并且通常该模式依据这个对象来命名。例如，

- 一个Strategy对象封装一个算法
- 一个State对象封装一个与状态相关的行为
- 一个Mediator对象封装对象间的协议
- 一个Iterator对象封装访问和遍历一个聚集对象中的各个构件的方法。

这些模式描述了程序中很可能会改变的方面。大多数模式有两种对象：封装该方面特征的新对象，和使用这些新的对象的已有对

象。如果不使用这些模式的话，通常这些新对象的功能就会变成这些已有对象的难以分割的一部分。例如，一个Strategy的代码可能会被嵌入到其Context类中，而一个State的代码可能会在该状态的Context类中直接实现。但不是所有的对象行为模式都象这样分割功能。例如， Chain of Responsibility）可以处理任意数目的对象（即一个链），而所有这些对象可能已经存在于系统中了。职责链说明了行为模式间的另一个不同点：并非所有的行为模式都定义类之间的静态通信关系。职责链提供在数目可变的对象间进行通信的机制。其他模式涉及到一些作为参数传递的对象。

对象作为参数

一些模式引入总是被用作参数的对象。例如Visitor。一个Visitor对象是一个多态的Accept操作的参数，这个操作作用于该Visitor对象访问的对象。虽然以前通常代替Visitor模式的方法是将Visitor代码分布在一些对象结构的类中，但Visitor从来都不是它所访问的对象的一部分。

其他模式定义一些可作为令牌到处传递的对象，这些对象将在稍后被调用。Command和Memento都属于这一类。在Command中，令牌代表一个请求；而在Memento中，它代表在一个对象在某个特定时刻的内部状态。在这两种情况下，令牌都可以有一个复杂的内部表示，但客户并不会意识到这一点。但这里还有一些区别：在Command模式中多态这个主题也贯穿于其他种类的模式。AbstractFactory，Builder( 3 . 2 )和Prototype都封装了关于对象是如何创建的信息。Decorator封装了可以被加入一个对象的职责。Bridge将一个抽象与它的实现分离，使它们可以各自独立的变化。很重要，因为执行Command对象是一个多态的操作。相反，Memento接口非常小，以至于备忘录只能作为一个值传递。因此它很可能根本不给它的客户提供任何多态操作。

Mediator和Observer是相互竞争的模式。它们之间的差别是， Observer通过引入Observer和Subject对象来分布通信，而Mediatorr对象则封装了其他对象间的通信。在Observer模式中，不存在封装一个约束的单个对象，而必须是由Observer和Subject对象相互协作来维护这个约束。通信模式由观察者和目标连接的方式决定：一个目标通常有多个观察者，并且有时一个目标的观察者也是另一个观察者的目标。Mediator模式的目的是集中而不是分布。它将维护一个约束的职责直接放在一个中介者中。我们发现生成可复用的Observer和Subject比生成可复用的MMediator容易一些。Observer模式有利于Observer和Subject间的分割和松耦合，同时这将产生粒度更细,从而更易于复用的类。

另一方面，相对于Subject，Mediator中的通信流更容易理解。观察者和目标通常在它们被创建后很快即被连接起来，并且很难看出此后它们在程序中是如何连接的。如果你了解Observerr模式，你将知道观察者和目标间连接的方式是很重要的，并且你也知道寻找哪些连接。然而， Observer模式引入的间接性仍然会使得一个系统难以理解。

对发送者和接收者解耦

当合作的对象直接互相引用时，它们变得互相依赖，这可能会对一个系统的分层和重用性产生负面影响。命令、观察者、中介者，和职责链等模式都涉及如何对发送者和接收者解耦，但它们又各有不同的权衡考虑。

命令模式使用一个Command对象来定义一个发送者和一个接收者之间的绑定关系，从而支持解耦。

观察者模式通过定义一个接口来通知目标中发生的改变，从而将发送者（目标）与接收者（观察者）解耦。Observer定义了一个比Command更松的发送者－接收者绑定，因为一个目标可能有多个观察者，并且其数目可以在运行时变化，因此当对象间有数据依赖时，最好用观察者模式来对它们进行解耦。中介者模式让对象通过一个Mediator对象间接的互相引用，从而对它们解耦。因此各Colleague对象仅能通过Mediatorr接口相互交谈。因为这个接口是固定的，为增加灵活性Mediator可能不得不实现它自己的分发策略。可以用一定方式对请求编码并打包参数，使得Colleague对象可以请求的操作数目不限。中介者模式可以减少一个系统中的子类生成，因为它将通信行为集中到一个类中而不是将其分布在各个子类中。然而，特别的分发策略通常会降低类型安全性。最后，职责链模式通过沿一个潜在接收者链传递请求而将发送者与接收者解耦，因为发送者和接收者之间的接口是固定的，职责链可能也需要一个定制的分发策略。因此它与Mediator一样存在类型安全的问题。如果职责链已经是系统结构的一部分，同时在链上的多个对象中总有一个可以处理请求，那么职责链将是一个很好的将发送者和接收者解耦的方法。此外，因为链可以被简单的改变和扩展，从而该模式提供了更大的灵活性。

总结,除了少数例外情况，各个行为设计模式之间是相互补充和相互加强的关系。职责链可以使用Command模式将请求表示为对象。Interpreter可以使用State模式定义语法分析上下文。迭代器可以遍历一个聚合，而访问者可以对它的每一个元素进行一个操作。行为模式也与能其他模式很好地协同工作。例如，一个使用Composite模式的系统可以使用一个访问者对该复合的各成分进行一些操作。它可以使用职责链使得各成分可以通过它们的父类访问某些全局属性。它也可以使用Decorator对该复合的某些部分的这些属性进行改写。它可以使用Observer模式将一个对象结构与另一个对象结构联系起来，可以使用State模式使得一个构件在状态改变时可以改变自身的行为。复合本身可以使用Builder中的方法创建，并且它可以被系统中的其他部分当作一个Prototype。设计良好的面向对象式系统通常有多个模式镶嵌在其中，但其设计者却未必使用这些术语进行思考。然而，在模式级别而不是在类或对象级别上的进行系统组装可以使我们更方便地获取同等的协同性。

参考文献：

http://blog.csdn.net/airhand/

http://blog.csdn.net/bloom121/

http://blog.csdn.net/laurecn/

http://blog.csdn.net/legendinfo/

http://www-128.ibm.com/developerworks/cn/java/l-struts1-1/

《Design Patterns》

《Java与模式》

《设计模式：可复用面向对象软件的基础》





上一篇 这样的网站怎么没人做 一个账户全网通行

下一篇 UML概述与详解

## 我的同类文章

### 软件工程、原理（49）

- 计算机必读经典教材图书... 2012-03-11 阅读 2377
- 关系数据库及NoSql图书... 2011-09-26 阅读 3783
- 推荐一本敏捷开发图书 2010-11-16 阅读 1065
- 以前使用的SVN配置 2010-06-21 阅读 646
- svn配置 2009-06-11 阅读 824
- geekOS操作系统（1） 2009-03-22 阅读 1147
- Linux源码阅读推荐阅读图书 2012-03-11 阅读 2017
- JavaEE程序员必读图书大推 2011-09-18 阅读 25353
- svn基本配置 2010-07-20 阅读 582
- 同步方法 2010-03-14 阅读 495
- c语言课程设计常用功能 2009-04-19 阅读 690

更多文章



## 参考知识库

- 大型网站架构知识库 1281 关注 | 532 收录
- **Java EE**知识库 1249 关注 | 581 收录
- **MySQL**知识库 8499 关注 | 1396 收录
- **Java SE**知识库 9507 关注 | 454 收录
- **Java Web**知识库 9800 关注 | 1042 收录

## 猜你在找

- PHP面向对象设计模式
- 数据结构和算法
- 设计模式
- 数据结构基础系列(1)：数据结构和算法
- PHP魔鬼训练之设计模式篇
- 设计模式大全
- 设计模式大全
- 设计模式大全
- java设计模式大全
- 设计模式大全



查看评论

11楼 熊猫小牛牛 2015-12-21 17:18发表

总结的太好了，我转载下，留着看

10楼 qq_24622863 2015-10-11 21:58发表

很好

9楼 qq_30911127 2015-09-01 20:10发表

我也写了个设计模式，欢迎切磋
http://godeye.org/index.php?a=lesson&id=95

8楼 Superemwill 2015-06-23 09:23发表

写的很好，学习了，谢谢

7楼 chachaxw 2015-05-18 12:42发表

总结的很全面，学到了！

Re: qq_24622863 2015-10-11 21:58发表

回复chachaxw：sadsad

6楼 davygeek 2015-04-11 13:32发表

不错，感谢分享

5楼 daigaigai520 2015-03-16 11:55发表

真的是好啊！！！

4楼 ISS314 2015-01-22 10:33发表

总结的很好，很有用！多谢博主分享

3楼 GJYSK 2014-12-09 09:42发表

总结的不错！多谢博主分享。

2楼 dukong123 2013-11-14 14:31发表

非常不错，感谢分享

1楼 freedomdebug 2012-05-28 17:18发表

很酷，总结的很全，感谢你的分享

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack
FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide
Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase
Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap