设计模式之独孤九剑

<http://www.runoob.com/design-pattern/design-pattern-tutorial.html>

# 基本知识

**什么是设计模式（四个基本要素）：**

**1. 模式名称（pattern name）**： 一个助记名，它用一两个词来描述模式的问题、解决方案和效果。命名一个新的模式增加了我们的设计词汇。设计模式允许我们在较高的抽象层次上进行设计。基于一个模式词汇表，我们自己以及同事之间就可以讨论模式并在编写文档时使用它们。模式名可以帮助我们思考，便于我们与其他人交流设计思想及设计结果。找到恰当的模式名也是我们设计模式编目工作的难点之一。

**2. 问题（problem）：** 描述了应该在何时使用模式。它解释了设计问题和问题存在的前因后果，它可能描述了特定的设计问题，如怎样用对象表示算法等。也可能描述了导致不灵活设计的类或对象结构。有时候，问题部分会包括使用模式必须满足的一系列先决条件。

**3. 解决方案（solution）：** 描述了设计的组成成分，它们之间的相互关系及各自的职责和协作方式。因为模式就像一个模板，可应用于多种不同场合，所以解决方案并不描述一个特定而具体的设计或实现，而是提供设计问题的抽象描述和怎样用一个具有一般意义的元素组合（类或对象组合）来解决这个问题。

**4. 效果（consequences）：** 描述了模式应用的效果及使用模式应权衡的问题。尽管我们描述设计决策时，并不总提到模式效果，但它们对于评价设计选择和理解使用模式的代价及好处具有重要意义。软件效果大多关注对时间和空间的衡量，它们也表述了语言和实现问题。因为复用是面向对象设计的要素之一，所以模式效果包括它对系统的灵活性、扩充性或可移植性的影响，显式地列出这些效果对理解和评价这些模式很有帮助。

**描述设计模式（独孤九剑）：**

**1. 模式概述：**模式名简洁地描述了模式的本质。一个好的名字非常重要，因为它将成为你的设计词汇表中的一部分。设计模式是做什么的？它的基本原理和意图是什么？它解决的是什么样的特定设计问题？

**2. 模式结构：**用以说明一个设计问题以及如何用模式中的类、对象来解决该问题的特定情景。该情景会帮助你理解随后对模式更抽象的描述。采用基于对象建模技术（ O M T）的表示法对模式中的类进行图形描述。我们也使用了交互图来说明对象之间的请求序列和协作关系。指设计模式中的类和/或对象以及它们各自的职责。

**3. 角色协作：**模式的参与者怎样协作以实现它们的职责。

**4. 作用效果：**模式怎样支持它的目标？使用模式的效果和所需做的权衡取舍？系统结构的哪些方面可以独立改变？

**5. 注意事项：**实现模式时需要知道的一些提示、技术要点及应避免的缺陷，以及是否存在某些特定于实现语言的问题。

**6. 使用场景：**什么情况下可以使用该设计模式？该模式可用来改进哪些不良设计？你怎样识别这些情况？

**7. 应用示例：**应用到此模式的典型例子

**8. 相关模式：**与这个模式紧密相关的模式有哪些？其间重要的不同之处是什么？这个模式应与哪些其他模式一起使用？

**9. 模式总结：** 模式延伸和优化等

**设计思想：**

OOA(Object-Oriented Analysis)面向对象分析—第一步

OOD(Object-Oriented Design)面向对象设计—第二步

需求（第一步）决定设计（第二步）（例如：可扩展性和过度设计）

主语（名词）—对象

状态（名称、形容词）—属性（封装：private）

功能（动词）—方法

事件（功能触发事件）—对象

事件监听者（监听事件）—对象

注：首先确定实体对象🡪实实在在存在的

主动—>被动（小孩睡觉/醒来）

当程序中有一段代码重复出现多次时就要考虑封装了

继承关系耦合性很高，谨慎使用

将两个角色之间的继承关系改为[聚合关系](http://baike.baidu.com/view/338574.htm" \t "_blank)，就是将它们之间的强关联改换成为弱关联。

一般的应用里，最顶级的是接口，然后是抽象类实现接口，最后才到具体类实现

模糊的事物设计成抽象类，事物的特性设计成接口：abstract class表示的是"is-a"关系，interface表示的是"like-a"关系。

多态—>可扩展性（Extendibility）：继承、重写、父类引用指向子类对象

面向接口编程

高内聚低耦合原则

先把功能完成，再考虑重构

**说明：**

涉及没有绝对的对与错

Over Design也是一种罪过

没有任何实际中的设计会一步到位

**设计原则：**

**总原则：开闭原则（Open Close Principle）**

开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，而是要扩展原有代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类等，后面的具体设计中我们会提到这点。

**1、单一职责原则**

不要存在多于一个导致类变更的原因，也就是说**每个类应该实现单一的职责**，如若不然，就应该把类拆分。

干自己改干的事情，别什么事都插一脚。

**2、里氏替换原则（Liskov Substitution Principle）**

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类**可以替换掉基类**，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科

历史替换原则中，**子类对父类的方法尽量不要重写和重载**。因为父类代表了定义好的结构，通过这个规范的接口与外界交互，子类不应该随便破坏它。

**3、依赖倒转原则（Dependence Inversion Principle）**

这个是开闭原则的基础，具体内容：**面向接口编程，依赖于抽象而不依赖于具体**。写代码时用到具体类时，不与具体类交互，而与具体类的上层接口交互。

针对接口编程，实现开闭原则的基础。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：每个接口中不存在子类用不到却必须实现的方法，如果不然，就要将接口拆分。使用多个隔离的接口，比使用单个接口（多个接口方法集合到一个的接口）要好。**<去部队领挂号信，不需要知道武器库在哪里>—接口定义的越细（但别过）越好**。

降低耦合度，接口单独设计，互相隔离。

**5、****迪米特法则（最少知道原则）（Demeter Principle）**

迪米特法则的[初衷](http://baike.baidu.com/view/406303.htm" \t "_blank)在于降低类之间的[耦合](http://baike.baidu.com/subview/156245/156245.htm" \t "_blank)。由于**每个类尽量减少对其他类的依赖**，因此，很容易使得系统的功能模块功能独立，相互之间不存在（或很少有）依赖关系。

就是说：一个类对自己依赖的类知道的越少越好。也就是说无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过public方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。

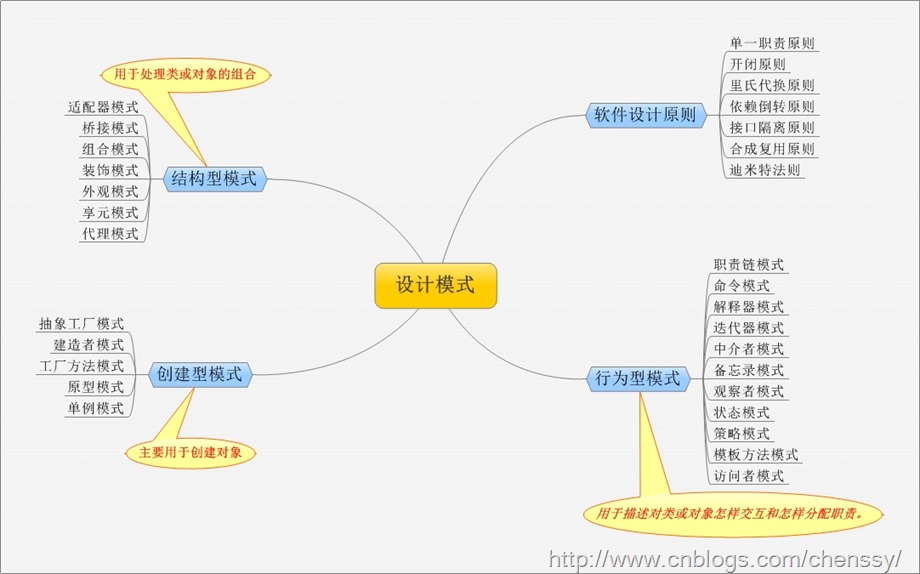
最少知道原则的另一个表达方式是：只与直接的朋友通信。类之间只要有耦合关系，就叫朋友关系。耦合分为依赖、关联、聚合、组合等。我们称出现为成员变量、方法参数、方法返回值中的类为直接朋友。局部变量、临时变量则不是直接的朋友。我们要求陌生的类不要作为局部变量出现在类中。

又称不知道原则：功能模块尽量独立。

**6、合成复用原则（Composite Reuse Principle）**

原则是尽量**首先使用合成/聚合的方式，而不是使用继承**。





**追美眉与23种设计模式：**

**创建型模式**

1、FACTORY—追MM少不了请吃饭了，麦当劳的鸡翅和肯德基的鸡翅都是MM爱吃的东西，虽然口味有所不同，但不管你带MM去麦当劳或肯德基，只管向服务员说“来四个鸡翅”就行了。麦当劳和肯德基就是生产鸡翅的Factory

**工厂模式：**客户类和工厂类分开。消费者任何时候需要某种产品，只需向工厂请求即可。消费者无须修改就可以接纳新产品。缺点是当产品修改时，工厂类也要做相应的修改。如：如何创建及如何向客户端提供。

　　2、BUILDER—MM最爱听的就是“我爱你”这句话了，见到不同地方的MM,要能够用她们的方言跟她说这句话哦，我有一个多种语言翻译机，上面每种语言都有一个按键，见到MM我只要按对应的键，它就能够用相应的语言说出“我爱你”这句话了，国外的MM也可以轻松搞掂，这就是我的“我爱你”builder。（这一定比美军在伊拉克用的翻译机好卖）

**建造模式：**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。

　　3、FACTORY METHOD—请MM去麦当劳吃汉堡，不同的MM有不同的口味，要每个都记住是一件烦人的事情，我一般采用Factory Method模式，带着MM到服务员那儿，说“要一个汉堡”，具体要什么样的汉堡呢，让MM直接跟服务员说就行了。

**工厂方法模式：**核心工厂类不再负责所有产品的创建，而是将具体创建的工作交给子类去做，成为一个抽象工厂角色，仅负责给出具体工厂类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。

　　4、PROTOTYPE—跟MM用QQ聊天，一定要说些深情的话语了，我搜集了好多肉麻的情话，需要时只要copy出来放到QQ里面就行了，这就是我的情话prototype了。（100块钱一份，你要不要）

**原始模型模式：**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。

　　5、SINGLETON—俺有6个漂亮的老婆，她们的老公都是我，我就是我们家里的老公Sigleton，她们只要说道“老公”，都是指的同一个人，那就是我(刚才做了个梦啦，哪有这么好的事)

**单例模式：**单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例单例模式。单例模式只应在有真正的“单一实例”的需求时才可使用。

**结构型模式**

　　6、ADAPTER—在朋友聚会上碰到了一个美女Sarah，从香港来的，可我不会说粤语，她不会说普通话，只好求助于我的朋友kent了，他作为我和Sarah之间的Adapter，让我和Sarah可以相互交谈了(也不知道他会不会耍我)

**适配器（变压器）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。

　　7、BRIDGE—早上碰到MM，要说早上好，晚上碰到MM，要说晚上好；碰到MM穿了件新衣服，要说你的衣服好漂亮哦，碰到MM新做的发型，要说你的头发好漂亮哦。不要问我“早上碰到MM新做了个发型怎么说”这种问题，自己用BRIDGE组合一下不就行了

**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。

　　8、COMPOSITE—Mary今天过生日。“我过生日，你要送我一件礼物。”“嗯，好吧，去商店，你自己挑。”“这件T恤挺漂亮，买，这条裙子好看，买，这个包也不错，买。”“喂，买了三件了呀，我只答应送一件礼物的哦。”“什么呀，T恤加裙子加包包，正好配成一套呀，小姐，麻烦你包起来。”“……”，MM都会用Composite模式了，你会了没有？

**合成模式：**合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。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。

　　9、DECORATOR—Mary过完轮到Sarly过生日，还是不要叫她自己挑了，不然这个月伙食费肯定玩完，拿出我去年在华山顶上照的照片，在背面写上“最好的的礼物，就是爱你的Fita”，再到街上礼品店买了个像框（卖礼品的MM也很漂亮哦），再找隔壁搞美术设计的Mike设计了一个漂亮的盒子装起来……，我们都是Decorator，最终都在修饰我这个人呀，怎么样，看懂了吗？

**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。

　　10、FACADE—我有一个专业的Nikon相机，我就喜欢自己手动调光圈、快门，这样照出来的照片才专业，但MM可不懂这些，教了半天也不会。幸好相机有Facade设计模式，把相机调整到自动档，只要对准目标按快门就行了，一切由相机自动调整，这样MM也可以用这个相机给我拍张照片了。

**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。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。

　　11、FLYWEIGHT—每天跟MM发短信，手指都累死了，最近买了个新手机，可以把一些常用的句子存在手机里，要用的时候，直接拿出来，在前面加上MM的名字就可以发送了，再不用一个字一个字敲了。共享的句子就是Flyweight，MM的名字就是提取出来的外部特征，根据上下文情况使用。

**享元模式：**FLYWEIGHT在拳击比赛中指最轻量级。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。

　　12、PROXY—跟MM在网上聊天，一开头总是“hi,你好”,“你从哪儿来呀？”“你多大了？”“身高多少呀？”这些话，真烦人，写个程序做为我的Proxy吧，凡是接收到这些话都设置好了自动的回答，接收到其他的话时再通知我回答，怎么样，酷吧。

**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。

**行为模式**

　　13、CHAIN OF RESPONSIBLEITY—晚上去上英语课，为了好开溜坐到了最后一排，哇，前面坐了好几个漂亮的MM哎，找张纸条，写上“Hi,可以做我的女朋友吗？如果不愿意请向前传”，纸条就一个接一个的传上去了，糟糕，传到第一排的MM把纸条传给老师了，听说是个老处女呀，快跑!

**责任链模式：**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。

　　14、COMMAND—俺有一个MM家里管得特别严，没法见面，只好借助于她弟弟在我们俩之间传送信息，她对我有什么指示，就写一张纸条让她弟弟带给我。这不，她弟弟又传送过来一个COMMAND，为了感谢他，我请他吃了碗杂酱面，哪知道他说：“我同时给我姐姐三个男朋友送COMMAND，就数你最小气，才请我吃面。”，

**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。

　　15、INTERPRETER—俺有一个《泡MM真经》，上面有各种泡MM的攻略，比如说去吃西餐的步骤、去看电影的方法等等，跟MM约会时，只要做一个Interpreter，照着上面的脚本执行就可以了。

**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。

　　16、ITERATOR—我爱上了Mary，不顾一切的向她求婚。

　　Mary：“想要我跟你结婚，得答应我的条件”

　　我：“什么条件我都答应，你说吧”

　　Mary：“我看上了那个一克拉的钻石”

　　我：“我买，我买，还有吗？”

　　Mary：“我看上了湖边的那栋别墅”

　　我：“我买，我买，还有吗？”

　　Mary：“我看上那辆法拉利跑车”

　　我脑袋嗡的一声，坐在椅子上，一咬牙：“我买，我买，还有吗？”

　　……

**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。

　　17、MEDIATOR—四个MM打麻将，相互之间谁应该给谁多少钱算不清楚了，幸亏当时我在旁边，按照各自的筹码数算钱，赚了钱的从我这里拿，赔了钱的也付给我，一切就OK啦，俺得到了四个MM的电话。

**调停者模式：**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。

　　18、MEMENTO—同时跟几个MM聊天时，一定要记清楚刚才跟MM说了些什么话，不然MM发现了会不高兴的哦，幸亏我有个备忘录，刚才与哪个MM说了什么话我都拷贝一份放到备忘录里面保存，这样可以随时察看以前的记录啦。

**备忘录模式：**备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捉住，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。

　　19、OBSERVER—想知道咱们公司最新MM情报吗？加入公司的MM情报邮件组就行了，tom负责搜集情报，他发现的新情报不用一个一个通知我们，直接发布给邮件组，我们作为订阅者（观察者）就可以及时收到情报啦

**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。

　　20、STATE—跟MM交往时，一定要注意她的状态哦，在不同的状态时她的行为会有不同，比如你约她今天晚上去看电影，对你没兴趣的MM就会说“有事情啦”，对你不讨厌但还没喜欢上的MM就会说“好啊，不过可以带上我同事么？”，已经喜欢上你的MM就会说“几点钟？看完电影再去泡吧怎么样？”，当然你看电影过程中表现良好的话，也可以把MM的状态从不讨厌不喜欢变成喜欢哦。

**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。

　　21、STRATEGY—跟不同类型的MM约会，要用不同的策略，有的请电影比较好，有的则去吃小吃效果不错，有的去海边浪漫最合适，单目的都是为了得到MM的芳心，我的追MM锦囊中有好多Strategy哦。

**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。

　　22、TEMPLATE METHOD——看过《如何说服女生上床》这部经典文章吗？女生从认识到上床的不变的步骤分为巧遇、打破僵局、展开追求、接吻、前戏、动手、爱抚、进去八大步骤(Template method)，但每个步骤针对不同的情况，都有不一样的做法，这就要看你随机应变啦(具体实现)；

**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。

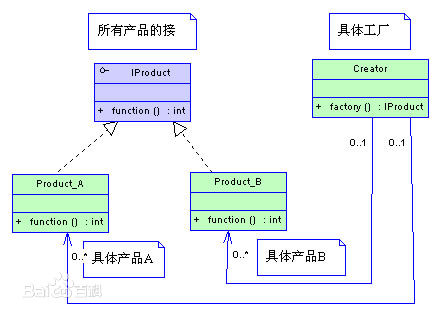
　　23、VISITOR—情人节到了，要给每个MM送一束鲜花和一张卡片，可是每个MM送的花都要针对她个人的特点，每张卡片也要根据个人的特点来挑，我一个人哪搞得清楚，还是找花店老板和礼品店老板做一下Visitor，让花店老板根据MM的特点选一束花，让礼品店老板也根据每个人特点选一张卡，这样就轻松多了；

**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。

## 创建模式(5)

### Factory (工厂模式) – 创建模式(3.1,3.3)

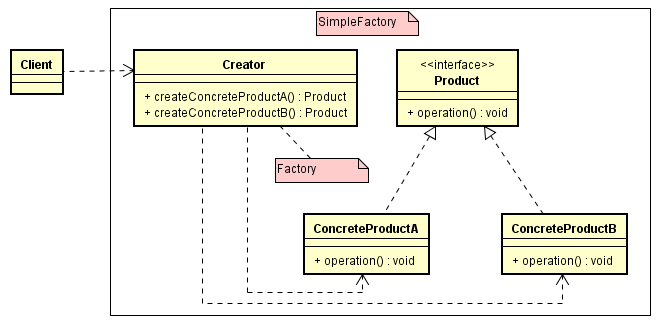
工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替new操作的一种模式。著名的Jive论坛 ,就大量使用了工厂模式，工厂模式在Java程序系统可以说是随处可见。因为工厂模式就相当于创建实例对象的new，我们经常要根据类Class生成实例对象，如A a=new A() 工厂模式也是用来创建实例对象的，所以以后new时就要多个心眼，是否可以考虑使用工厂模式，虽然这样做，可能多做一些工作，但会给你系统带来更大的可扩展性和尽量少的修改量。



#### Simple Factory(简单工厂模式)

##### 模式概述

简单工厂模式又称静态工厂模式。Simple Factory模式根据提供给它的数据，返回几个可能类中的一个类的实例。通常它返回的类都有一个公共的父类和公共的方法。



##### 模式结构

* 工厂角色Creator：工厂类在客户端的直接控制下（Create方法）创建产品对象。
* 抽象产品角色Product：定义简单工厂创建的对象的父类或它们共同拥有的接口。可以是一个类、抽象类或接口。
* 具体产品角色ConcreteProduct：定义工厂具体加工出的对象。

##### 角色协作

客户端调用工厂角色中创建具体产品的方法即可得到响应的产品对象。另外客户端还可以传给工厂角色信息，有工厂决定调用创建具体产品的方法得到具体的产品对象。

##### 作用效果

工厂类是整个模式的关键.包含了必要的逻辑判断,根据外界给定的信息,决定究竟应该创建哪个具体类的对象.通过使用工厂类,外界可以从直接创建具体产品对象的尴尬局面摆脱出来,仅仅需要负责“消费”对象就可以了。而不必管这些对象究竟如何创建及如何组织的．明确了各自的职责和权利，有利于整个[软件体系结构](http://baike.baidu.com/view/1317046.htm" \t "_blank)的优化。

##### 注意事项

由于工厂类集中了所有实例的创建逻辑，违反了[高内聚](http://baike.baidu.com/view/292136.htm" \t "_blank)责任分配原则，将全部创建逻辑集中到了一个工厂类中；它所能创建的类只能是事先考虑到的，如果需要添加新的类，则就需要改变工厂类了，违反了OCP原则。

当系统中的具体产品类不断增多时候，可能会出现要求工厂类根据不同条件创建不同实例的需求．这种对条件的判断和对具体产品类型的判断交错在一起，很难避免模块功能的蔓延，对系统的维护和扩展非常不利；

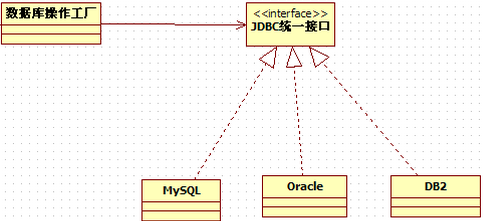
这些缺点在[工厂方法模式](http://baike.baidu.com/view/1580263.htm" \t "_blank)中得到了一定的克服。

##### 使用场景

* 工厂类负责创建的对象比较少；
* 客户只知道传入工厂类的参数，对于如何创建对象（逻辑）不关心；
* 由于简单工厂很容易违反高内聚责任分配原则，因此一般只在很简单的情况下应用。

##### 应用示例

JDK：java.text.DateFormat, java.lang.Integer, java.lang.Class<T>等



##### 相关模式

##### 模式扩展

#### Factory Method(工厂方法模式)

##### 模式描述

别名：多态性工厂模式 -> 简单工厂中的方法升级为工厂。

工厂方法模式之所以有一个别名叫多态性工厂模式是因为具体工厂类都有共同的接口，或者有共同的抽象父类。

工厂方法模式是一种常用的对象创建型设计模式,此模式的核心精神是封装类中不变的部分，提取其中个性化善变的部分为独立类，通过依赖注入以达到解耦、复用和方便后期维护拓展的目的。它的核心结构有四个角色，分别是抽象工厂；具体工厂；抽象产品；具体产品。

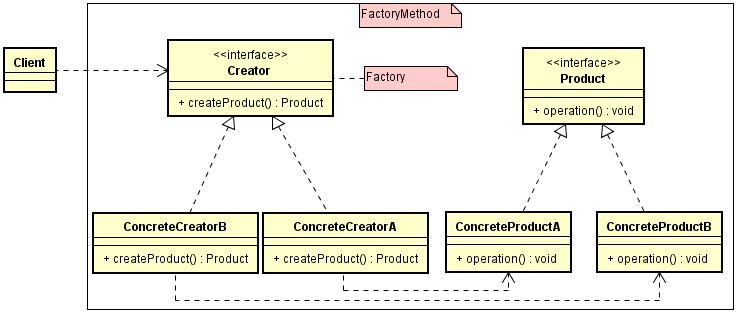
工厂方法模式的意义是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。核心工厂类不再负责产品的创建，这样核心类成为一个抽象工厂角色，仅负责具体工厂子类必须实现的接口，这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品。抽象工厂角色可以规定出多于一个的工厂方法，从而使具体工厂角色实现这些不同的工厂方法，这些方法可以提供不同的商业逻辑，以满足提供不同的产品对象的任务。

工厂方法模式是[简单工厂模式](http://baike.baidu.com/view/1227908.htm" \t "_blank)的衍生，由于使用了多态性，其解决了许多简单工厂模式的问题。首先完全实现‘开－闭 原则’，实现了可扩展。其次更复杂的层次结构，可以应用于产品结果复杂的场合。当系统扩展需要添加新的产品对象时，仅仅需要添加一个具体对象以及一个具体工厂对象，原有工厂对象不需要进行任何修改，也不需要修改客户端，很好的符合了"开放－封闭"原则。

工厂方法模式对简单工厂模式进行了抽象。有一个抽象的Factory类（可以是抽象类和接口），这个类将不再负责具体的产品生产，而是只制定一些规范，具体的生产工作由其子类去完成。在这个模式中，工厂类和产品类往往可以依次对应。即一个抽象工厂对应一个抽象产品，一个具体工厂对应一个具体产品，这个具体的工厂就负责生产对应的产品。

工厂方法模式是最典型的模板方法模式(Template Method pattern)应用。

在工厂方法模式中，工厂类与产品类往往具有平行的等级结构，它们之间一一对应。



##### 模式结构

* 抽象工厂（Creator）角色：是工厂方法模式的核心，与应用程序无关。任何在模式中创建的对象的工厂类必须实现这个接口。
* 具体工厂（ConcreteCreator）角色：这是实现抽象工厂接口的具体工厂类，包含与应用程序密切相关的逻辑，并且受到应用程序调用以创建产品对象。
* 抽象产品（Product）角色：工厂方法模式所创建的对象的超类型，也就是产品对象的共同父类或共同拥有的接口。
* 具体产品（Concrete Product）角色：这个角色实现了抽象产品角色所定义的接口。某具体产品有专门的具体工厂创建，它们之间往往一一对应。

##### 角色协作

客户端调用具体的工厂即可得到具体产品的对象。内部是，具体工厂来创建具体的产品对象并返回给客户端。

##### 作用效果

在工厂方法中，用户只需要知道所要产品的具体工厂，无须关系具体的创建过程，甚至不需要具体产品类的类名。

在系统增加新的产品时，我们只需要添加一个具体产品类和对应的实现工厂，无需对原工厂进行任何修改，很好地符合了“开闭原则”。

##### 注意事项

每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。

##### 使用场景

* 对于某个产品，调用者清楚地知道应该使用哪个具体工厂服务，实例化该具体工厂，生产出具体的产品来。Java Collection中的iterator() 方法即属于这种情况。
* 只是需要一种产品，而不想知道也不需要知道究竟是哪个工厂为生产的，即最终选用哪个具体工厂的决定权在生产者一方，它们根据当前系统的情况来实例化一个具体的工厂返回给使用者，而这个决策过程这对于使用者来说是透明的。

##### 应用示例

简单模拟SpringIOC

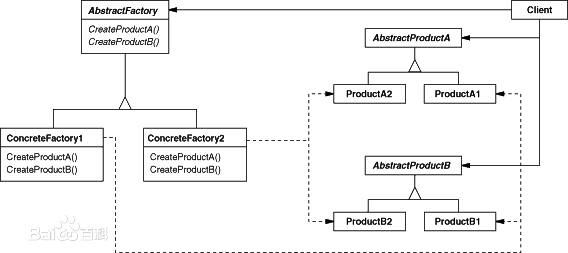
##### 相关模式

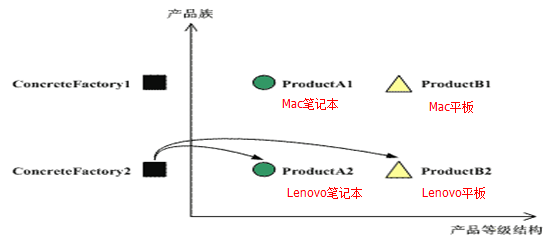
##### 模式总结

#### Abstract Factory(抽象工厂模式)

##### 模式概述

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。





##### 模式结构

* 抽象工厂（Abstract Factory）角色：担任这个角色的是工厂方法模式的核心，它是与应用系统商业逻辑无关的。
* 具体工厂（Concrete Factory）角色：这个角色直接在客户端的调用下创建产品的实例。这个角色含有选择合适的产品对象的逻辑，而这个逻辑是与应用系统的商业逻辑紧密相关的。
* 抽象产品（Abstract Product）角色：担任这个角色的类是工厂方法模式所创建的对象的父类，或它们共同拥有的接口。
* 具体产品（Concrete Product）角色：抽象工厂模式所创建的任何产品对象都是某一个具体产品类的实例。这是客户端最终需要的东西，其内部一定充满了应用系统的商业逻辑。

##### 角色协作

客户端只需要调用具体的产品族工厂即可得到具体的产品对象。产品族中的具体产品由工厂创建。

##### 作用效果

* 它分离了具体的类；
* 它使得易于交换产品系列；
* 它有利于产品的一致性。

##### 注意事项

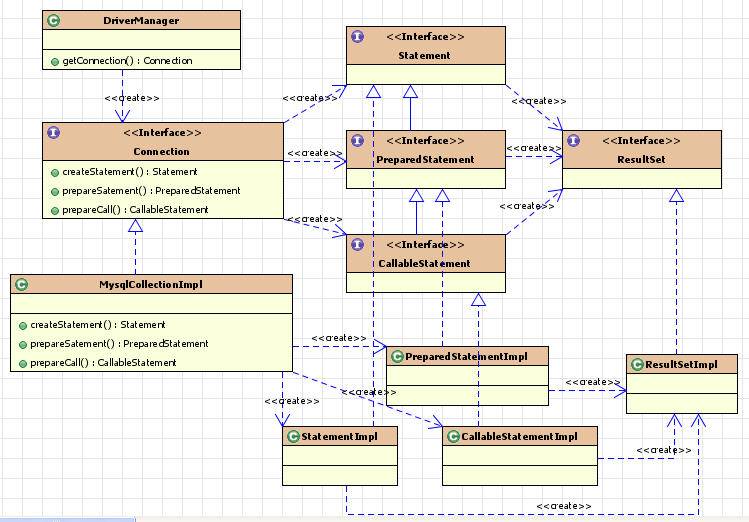
难以支持新种类的产品。抽象工厂模式以一种倾斜的方式支持增加新的产品，它为新产品族的增加提供方便，而不能为新的产品等级结构的增加提供这样的方便。

##### 使用场景

* 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的。
* 这个系统有多于一个的产品族，而系统只消费其中某一产品族。（抽象工厂模式的原始用意Unix&Windows）。
* 同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来。
* 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现。

##### 应用示例

* 抽象工厂模式的原始用意Unix&Windows
* java.sql.Connection



##### 相关模式

##### 模式总结

一般而言，有多少个产品等级结构，就会在工厂角色中发现多少个工厂方法。每一个产品等级结构中有多少个具体的产品，就有多少个产品族，也就会在工厂等级结构中发现多少个具体工厂。

在不改变产品等级结构的情况下，增加产品族就是意味着向每一个产品等级结构中增加一个或者多个新的具体产品角色，这时只需要向工厂等级结构中增加新的元素就可以了，具体的说，只需要增加新的具体工厂类就可以了。

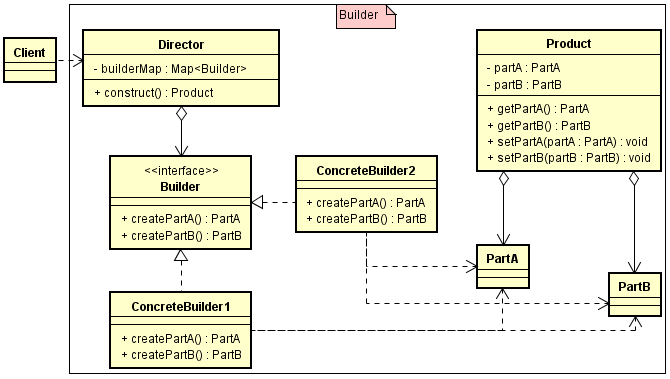
在产品族数目不变的情况下，增加产品等级结构，相当于增加一个与现有产品等级结构平行的一个新的产品等级结构，这时需要向修改所有的工厂角色，增加一个新的工厂方法,这是不支持OCP(开放-封闭原则)的。

### Builder(建造者模式) – 创建模式(3.2)

#### 模式概述

建造者模式可以将一个产品的内部架结构与产品的生成过程分割开来，从而可以使一个建造过程生成具有不同的内部结构的产品对象。

之所以使用"建造者"而没有用"生成器"就是因为用零件生产产品，"建造"更为合适，"创建"或"生成"不太恰当。



#### 模式结构

* 建造者（Builder）角色：给出一个抽象接口，以规范产品对象的各个组成成分的建造。
* 具体建造者（ConcreteBuilder）角色：实现产品对象中各个组成部分的建造。
* 指导者（Director）角色：调用具体建造者角色以创建复杂产品对象的各个组成部分；实现复杂产品对象的建造过程。
* 产品（Product）角色：产品便是建造中的复杂对象。内部定义各个组成部分，并提供getter/setter方法。

#### 角色协作

客户端调用指导者，指导者new一个复杂产品对象，并调用具体的建造者创建复杂对象的各个组成部分，最后调用建造过程完成复杂对象的组装，并将结果返回。

#### 作用效果

建造模式的使用使得产品的内部结构可以独立的变化。使用建造者模式可以使客户端不必知道产品内部组成的细节。

#### 注意事项

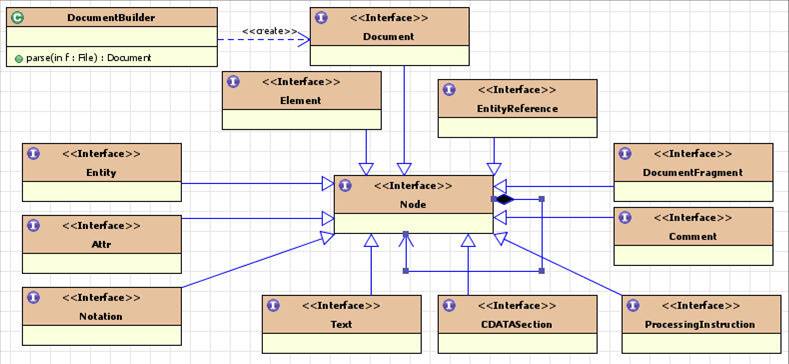
注意防止类泛滥。

#### 使用场景

* 需要生成的产品对象有复杂的内部结构。
* 需要生成的产品对象的属性相互依赖，建造者模式可以强迫生成顺序。
* 对象创建过程中会用到其它对象，这些对象在产品对象的创建过程中不易得到。

#### 应用示例

StringBuilder、DocumentBuilder



#### 相关模式

抽象工厂模式

#### 模式总结

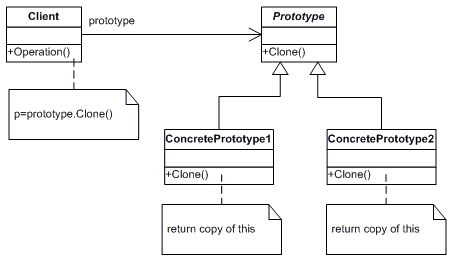
### Prototype(原型模式) – 创建模式(3.4)

#### 模式概述

别名：克隆模式。

通过给出一个原型对象来指明所要创建的对象类型，然后用复制这个原型对象的办法创建出更多的同类型对象。

我们通过原型模式可以快速的创建一个对象而不需要提供专门的new()操作就可以快速完成对象的创建，这无疑是一种非常有效的方式，快速的创建一个新的对象。**基于内存**。



#### 模式结构

* 客户（Client）角色：让一个具体原型克隆自身从而创建一个新的对象。
* 抽象原型（Prototype）角色：声明一个克隆自身的接口。例如：Cloneable
* 具体原型（Concrete Prototype）角色：被复制的对象。此角色需要实现抽象原型角色所要求的接口。实现一个克隆自身的操作。

#### 角色协作

客户端创建一个具体原型对象，并调用其clone()创建一个新的原型对象。具体原型角色实现克隆自身的操作，即实现clone()方法。

#### 作用效果

* 运行时刻增加和删除产品
* 改变值以指定新对象
* 改变结构以指定新对象
* 减少子类的构造
* 用类动态配置应用

#### 注意事项

Prototype的主要缺陷是每一个Prototype的子类都必须实现clone操作，这可能很困难。例如，当所考虑的类已经存在时就难以新增clone操作。当内部包括一些不支持拷贝或有循环引用的对象时，实现克隆可能也会很困难的。

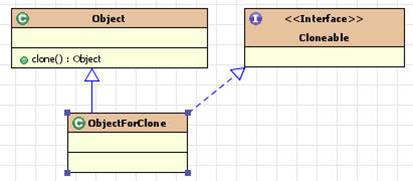
#### 使用场景

* 当一个系统应该独立于它的产品创建、构成和表示时，要使用P r o t o t y p e模式；
* 当要实例化的类是在运行时刻指定时，例如，通过动态装载；
* 当为了避免创建一个与产品类层次平行的工厂类层次时；
* 当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们

可能比每次用合适的状态手工实例化该类更方便一些。

#### 应用示例

JDK：Object.clone；Cloneable



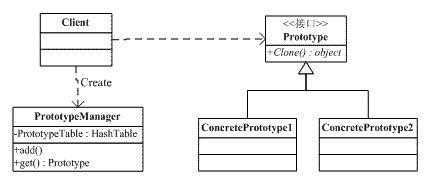
#### 相关模式

Prototype有许多和Abstract Factory和Builder一样的效果：它对客户隐藏了具体的产品类，因此减少了客户知道的名字的数目。此外，这些模式使客户无需改变即可使用与特定应用相关的类。

大量使用Composite和Decorator模式的设计通常也可从Prototype模式处获益。

#### 模式总结

带原型管理器的原型模式



客户（Client）角色：让一个具体原型克隆自身从而创建一个新的对象。

抽象原型（Prototype）角色：声明一个克隆自身的接口。例如：Cloneable。

具体原型（Concrete Prototype）角色：被复制的对象。此角色需要实现抽象原型角色所要求的接口。实现一个克隆自身的操作。

原型管理器（Prototype Manager）角色：创建具体原型类的对象，并记录每一个被创建的对象。

### Singleton(单例模式) – 创建模式(3.5)

#### 模式概述

单例模式最初的定义出现于《设计模式》（艾迪生维斯理, 1994）：“保证一个类仅有一个实例，并提供一个访问它的全局访问点。”

Java中单例模式定义：“一个类有且仅有一个实例，并且自行实例化向整个系统提供。”



#### 模式结构

Singleton模式包含的角色只有一个，就是Singleton。Singleton拥有一个私有构造函数，确保用户无法通过new直接实例它。除此之外，该模式中包含一个静态私有成员变量instance与静态公有方法Instance()。Instance方法负责检验并实例化自己，然后存储在静态成员变量中，以确保只有一个实例被创建。

#### 角色协作

#### 作用效果

实例控制：单例模式会阻止其他对象实例化其自己的单例对象的副本，从而确保所有对象都访问唯一实例。

灵活性：因为类控制了实例化过程，所以类可以灵活更改实例化过程。

#### 注意事项

开销：虽然数量很少，但如果每次对象请求引用时都要检查是否存在类的实例，将仍然需要一些开销。可以通过使用静态初始化解决此问题。

可能的开发混淆：使用单例对象（尤其在类库中定义的对象）时，开发人员必须记住自己不能使用new关键字实例化对象。因为可能无法访问库源代码，因此应用程序开发人员可能会意外发现自己无法直接实例化此类。

对象的生存期：不能解决删除单个对象的问题。在提供内存管理的语言中（例如基于.NET Framework的语言），只有单例类能够导致实例被取消分配，因为它包含对该实例的私有引用。在某些语言中（如 C++），其他类可以删除对象实例，但这样会导致单例类中出现悬浮引用。

不要使用单例模式存取全局变量。这违背了单例模式的用意，最好放到对应类的静态成员中。

不要将数据库连接做成单例，因为一个系统可能会与数据库有多个连接，并且在有连接池的情况下，应当尽可能及时释放连接。Singleton模式由于使用静态成员存储类实例，所以可能会造成资源无法及时释放，带来问题。（多例模式：数据库连接池）

#### 使用场景

使用Singleton模式有一个必要条件：在一个系统要求一个类只有一个实例时才应当使用单例模式。

当一个类的对象只需要或者只可能有一个时，应该考虑单例模式。

#### 应用示例

JDK之Runtime，Toolkit

#### 相关模式

#### 模式总结

单例类只有一个实例。

单例类自己创建自己的唯一实例。

单例类给所有其它对象提供这一实例。

如果一个类的实例应该在初始化时被创建出来，应该考虑使用饿汉式单例。

如果一个类的实例不需要预先被创建，也许这个类的实例并不一定能用得上，也许这个类的实例创建过程比较耗费时间，也许就是真的没必须提前创建。那么应该考虑懒汉式单例。

在使用懒汉式单例的时候，应该考虑到线程的安全性问题🡪用静态内部类方式。

## 结构模式(7)

### Adapter([适配器模式](http://baike.baidu.com/view/3371585.htm" \t "_blank)) – 结构模式(4.1)

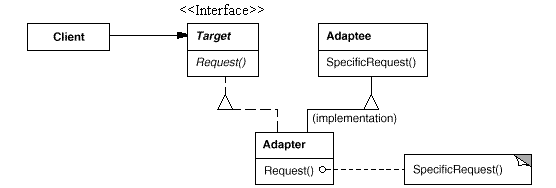
#### 模式概述

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，从而使原本接口不匹配而无法在一起工作的两个类能够在一起工作。

适配器模式有类的适配器模式和对象的适配器模式两种。

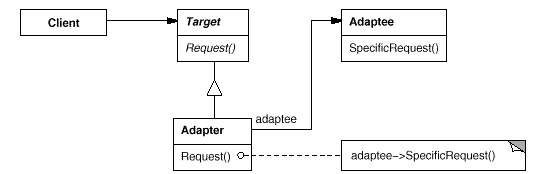
#### 模式结构

类的适配器模式（继承）：



* 目标（Target）角色：这是客户所期待的接口。
* 源（Adaptee）角色：需要适配的类。
* 适配器（Adapter）角色：把源接口转换成目标接口。这一角色必须是类。

对象的适配器模式（聚合）：



* 目标（Target）角色：这是客户所期待的接口。目标可以是具体的或抽象的类，也可以是接口。
* 源（Adaptee）角色：需要适配的类。
* 适配器（Adapter）角色：通过在内部包装（Wrap）一个Adaptee对象，把源接口转换成目标接口。

#### 角色协作

Client在Adapter实例上调用一些操作。接着适配器调用Adaptee的操作实现这个请求。

#### 作用效果

转换类的功能

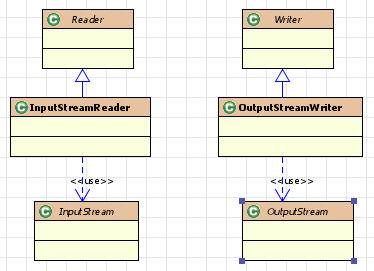
#### 注意事项

#### 使用场景

* 系统需要使用现有的类，而此类的接口不符合系统的需要。
* 你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。
* （仅适用于对象Adapter）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

#### 应用示例

JDK之InputStreamReader(InputStream) and OutputStreamWriter(OutputStream)



#### 相关模式

适配器（**转换功能**）、装饰（**增加功能**）、代理（**处理功能**）相通，很多模式重点是语义的区别而非语法的区别。

#### 模式总结

目标接口可以省略，模式发生退化。但这种做法看似平庸而并不平庸，它可以使Adaptee不必实现不需要的方法（可以参考Default Adapter模式）。其表现形式就是父类实现缺省方法，而子类只需实现自己独特的方法。这有些像模板（Template）模式。

适配器类可以是抽象类。

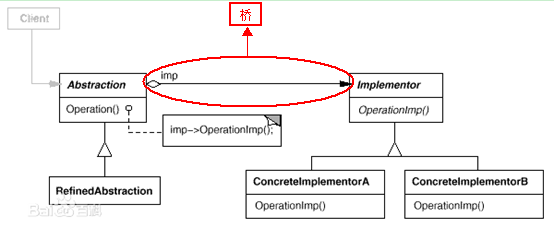
源可以进一步抽象 -> 父类引用指向子类对象（多态）。

带参数的适配器模式。使用这种办法，适配器类可以根据参数返还一个合适的实例给客户端。

适配器模式不适合在详细设计阶段使用它，它是一种补偿模式，专用来在系统后期扩展、修改时所用。

### Bridge(桥接模式) – 结构模式(4.2)

桥接模式的用意是"将抽象化(Abstraction)与实现化([Implementation](http://baike.baidu.com/subview/1147244/1147244.htm" \t "_blank))[脱耦](http://baike.baidu.com/view/1877376.htm" \t "_blank)（将两个实体的行为的某种强关联关系去掉或者弱化），使得二者可以独立地变化"。（将抽象部分与实现部分分离，使它们都可以独立的变化）



**抽象化(Abstraction)角色：**抽象化给出的定义，并保存一个对实现化对象的引用。

**修正抽象化(Refined Abstraction)角色：**扩展抽象化角色，改变和修正父类对抽象化的定义。

**实现化(Implementor)角色：**这个角色给出实现化角色的接口，但不给出具体的实现。必须指出的是，这个接口不一定和抽象化角色的接口定义相同，实际上，这两个接口可以非常不一样。实现化角色应当只给出底层操作，而抽象化角色应当只给出基于底层操作的更高一层的操作。

**具体实现化(Concrete Implementor)角色：**这个角色给出实现化角色接口的具体实现。

**使用场景：**

1.如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的联系。

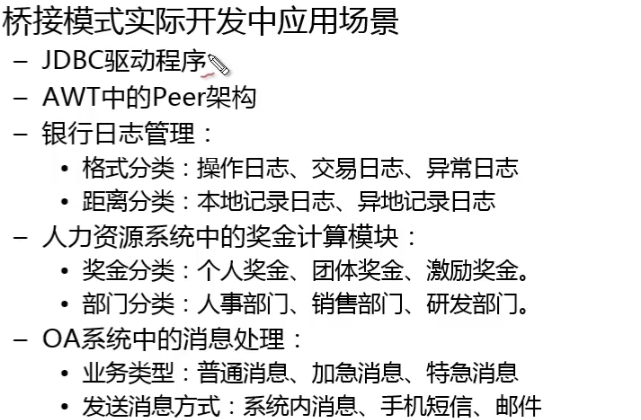
2.设计要求实现化角色的任何改变不应当影响客户端，或者说实现化角色的改变对客户端是完全透明的。

3.一个构件有多于一个的抽象化角色和实现化角色，系统需要它们之间进行动态耦合。

4.虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。

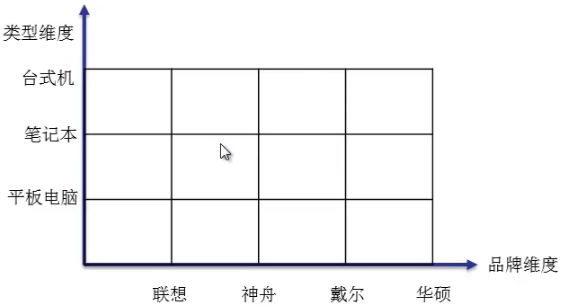
总结：

Bridge模式是一个非常有用的模式，也非常复杂，它很好的符合了开放-封闭原则和优先使用聚合，而不是继承这两个面向对象原则。



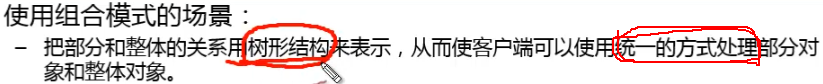
**示例：**模拟路和交通工具

**模式总结：两个纬度，独立扩展，聚合桥接！**

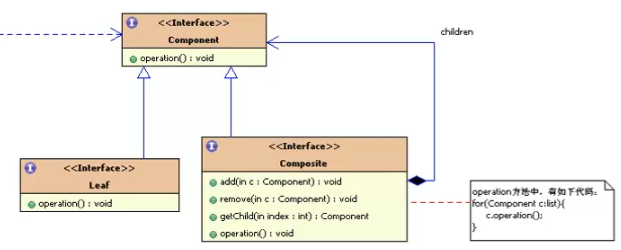


### Composite(组合模式) – 结构模式(4.3)

#### 模式概述



#### 模式结构



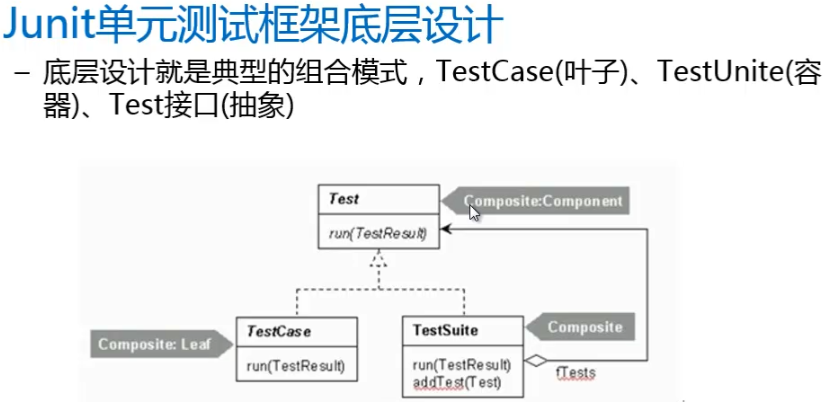
#### 角色协作

#### 作用效果

#### 注意事项

#### 使用场景

#### 应用示例



#### 相关模式

#### 模式总结

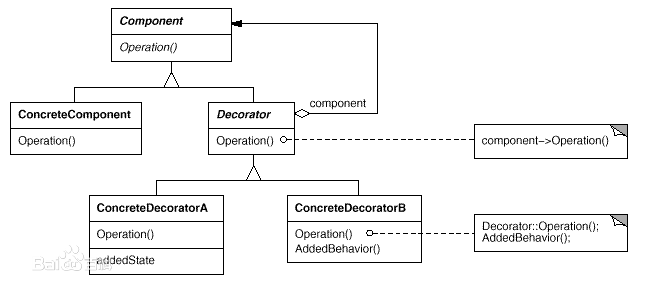
### Decorator(装饰模式) – 结构模式(4.4)

#### 模式概述

别名：包装（Wrapper）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。

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任。换言之，客户端并不会觉得对象在装饰前和装饰后有什么不同。装饰模式可以在不使用创造更多子类的情况下，将对象的功能加以扩展。



#### 模式结构

* 抽象构件（Component）角色：给出一个抽象接口，以规范准备接收附加责任的对象。
* 具体构件（Concrete Component）角色：定义一个将要接收附加责任的类。
* 装饰（Decorator）角色：持有一个构件（Component）对象的实例，并定义一个与抽象构件接口一致的接口。
* 具体装饰（Concrete Decorator）角色：负责给构件对象"贴上"附加的责任。

#### 角色协作

Decorator将请求转发给它的Component对象，并有可能在转发请求前后执行一些附加的动作。

#### 作用效果

* 比静态继承更灵活
* 避免在层次结构高层的类有太多的特征

#### 注意事项

由于使用装饰模式，可以比使用继承关系需要较少数目的类。使用较少的类，当然使设计比较易于进行。但是，在另一方面，使用装饰模式会产生比使用继承关系更多的对象。更多的对象会使得查错变得困难，特别是这些对象看上去都很相像。

#### 使用场景

* 需要扩展一个类的功能，或给一个类增加附加责任。
* 需要动态地给一个对象增加功能，这些功能可以再动态地撤销。
* 需要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，从而使继承关系变得不现实。

#### 应用示例

JDK之BufferedReader。

#### 相关模式

Adapter模式：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。

Strategy模式：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。

#### 模式总结

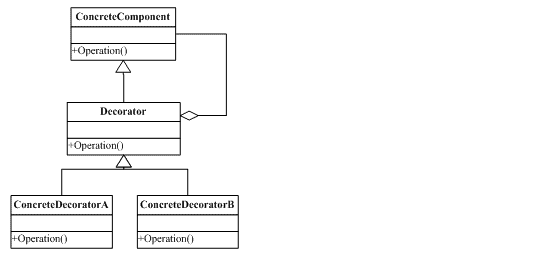
**总结：**

大多数情况下，装饰模式的实现都比上面定义中给出的示意性实现要简单。对模式进行简化时需要注意以下的情况：

1.一个装饰类的接口必须与被装饰类的接口相容。

2.尽量保持Component作为一个"轻"类，不要把太多的逻辑和状态放在Component类里。

3.如果只有一个ConcreteComponent类而没有抽象的Component类（接口），那么Decorator类经常可以是ConcreteComponent的一个子类。如下图所示：



4.如果只有一个ConcreteDecorator类，那么就没有必要建立一个单独的Decorator类，而可以把Decorator和ConcreteDecorator的责任合并成一个类。

5.就增加功能来说，Decorator模式相比生成子类更为灵活。

### Facade(外观模式) – 结构模式(4.5)

#### 模式概述

别名：门面模式。

为子系统中的一组接口提供一个一致的界面，门面模式模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。引入外观角色之后，门面类将客户端与子系统的内部复杂性分隔开，用户只需要直接与外观角色交互，用户与子系统之间的复杂关系由外观角色来实现，从而降低了系统的耦合度。

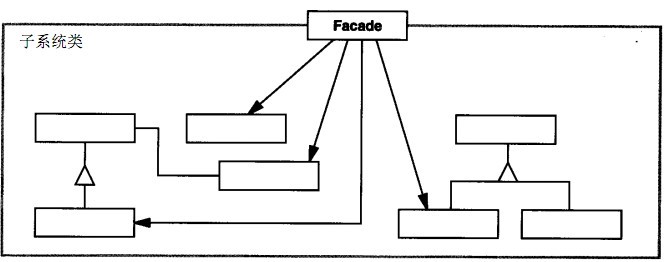
1. 将一个系统划分成为若干个子系统有利于降低系统的复杂性。一个常见的设计目标是使子系统间的通信和相互依赖关系达到最小。达到该目标的途径之一是就是引入一个外观（facade）对象，它为子系统及个模块提供一个简单而单一的入口。符合单一职责原则。

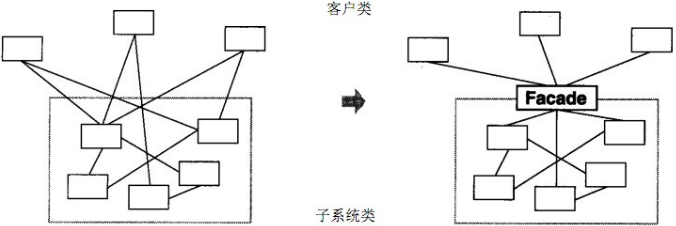
2. 通过引入一个新的外观类可以降低原有系统的复杂度，外观类充当了客户类与子系统类之间的“中间者”，同时降低客户类与子系统类的耦合度。外观模式就是实现代码重构以便达到“迪米特法则”要求的一个强有力的武器。

3. 在门面模式中，通常只需要一个门面类，并且此门面类只有一个实例，换言之它是一个单例类。当然这并不意味着在整个系统里只能有一个门面类，而仅仅是说对每一个子系统只有一个门面类。或者说，如果一个系统有好几个子系统的话，每一个子系统有一个门面类，整个系统可以有数个门面类。

4. 外观模式的用意是为客户端和子系统提供一个集中化和简化的沟通渠道，而不是向子系统加入新的功能，新功能的增加应该通过修改原有子系统类或增加新的子系统类来实现，不能通过外观类来实现。

#### 模式结构





* 门面(Facade)角色：

1. 是模式的核心，它被客户调用，知道各个子系统的功能，知道哪些子系统类负责处理请求。

2. 将客户的请求代理给适当的子系统对象。

* 子系统(Subsystem)角色：

1. 实现子系统的功能。

2. 处理由Facade对象指派的任务。

3. 没有Facade的任何相关信息；即没有指向Facade的引用。

#### 角色协作

1. 客户程序通过发送请求给Facade的方式与子系统通讯，Facade将这些消息转发给适当的子系统对象。尽管是子系统中的有关对象在做实际工作，但Facade模式本身也必须将它的接口转换成子系统的接口。

2. 使用Facade的客户程序不需要直接访问子系统对象。

#### 作用效果

1. 对客户屏蔽子系统组件，减少了客户处理的对象数目并使得子系统使用起来更加容易。通过引入外观模式，客户代码将变得很简单，与之关联的对象也很少。

2. 实现了子系统与客户之间的松耦合关系，这使得子系统的组件变化不会影响到调用它的客户类，只需要调整外观类即可。

3. 降低了大型软件系统中的编译依赖性，并简化了系统在不同平台之间的移植过程，因为编译一个子系统一般不需要编译所有其他的子系统。一个子系统的修改对其他子系统没有任何影响，而且子系统内部变化也不会影响到外观对象。（移植还好，其他有点费解）

4. 只是提供了一个访问子系统的统一入口，并不影响用户直接使用子系统类。可以在系统易用性和通用性之间加以选择。

#### 注意事项

1. 不能很好地限制客户使用子系统类，如果对客户访问子系统类做太多的限制则减少了可变性和灵活性。

2.  增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”。办法是引入抽象外观类或接口。

#### 使用场景

1. 当你要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。Facade可以提供一个简单的缺省视图，这一视图对大多数用户来说已经足够，而那些需要更多的可定制性的用户可以越过Facade层。

2. 客户程序与抽象类的实现部分之间存在着很大的依赖性。引入Facade将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。

3. 当你需要构建一个层次结构的子系统时，使用门面模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过Facade进行通讯，从而简化了它们之间的依赖关系。

#### 应用示例

1. JDK之java.util.logging.Logger简单模拟。日志系统Façade🡪Logger，当然还有log4j。

2. JDK之java.lang.Class。无论一个类被设计的如何复杂，如果只需要获取这个类的名称的话，那么只需要为这个对象生成一个Class对象，然后调用相应的方法即可，成功地对客户端隐藏了很多实现细节，运用了Facade模式。

#### 相关模式

1. [抽象工厂模式](http://blog.csdn.net/hguisu/article/details/7505909)：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。

2. [中介模式](http://blog.csdn.net/hguisu/article/details/7554612)：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。

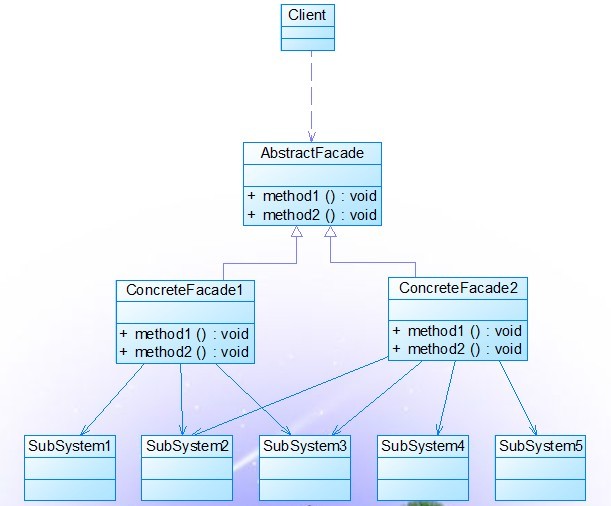
3. 通常来讲，仅需要一个Facade对象，因此Facade对象通常属于Singleton模式。

[4. 适配器模式](http://blog.csdn.net/hguisu/article/details/blog.csdn.net/hguisu/article/details/7527842)：Adapter模式是将一个接口通过适配来间接转换为另一个接口。外观模式的话，其主要是提供一个整洁的一致的接口给客户端。

#### 模式总结

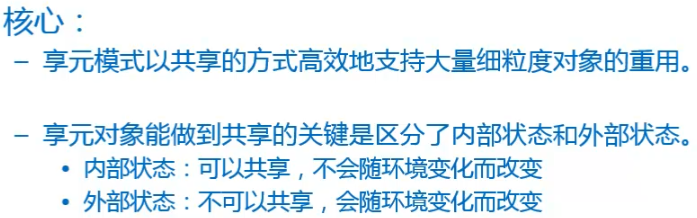
1. 一个系统有多个外观类：在外观模式中，通常只需要一个外观类，并且此外观类只有一个实例，换言之它是一个单例类。在很多情况下为了节约系统资源，一般将外观类设计为单例类。当然这并不意味着在整个系统里只能有一个外观类，在一个系统中可以设计多个外观类，每个外观类都负责和一些特定的子系统交互，向用户提供相应的业务功能。

2. 抽象外观类的引入：外观模式最大的缺点在于违背了[“开闭原则”](http://blog.csdn.net/hguisu/article/details/7571617)，当增加新的子系统或者移除子系统时需要修改外观类，可以通过引入抽象外观类在一定程度上解决该问题，客户端针对抽象外观类进行编程。对于新的业务需求，不修改原有外观类，而对应增加一个新的具体外观类，由新的具体外观类来关联新的子系统对象，同时通过修改配置文件来达到不修改源代码并更换外观类的目的。

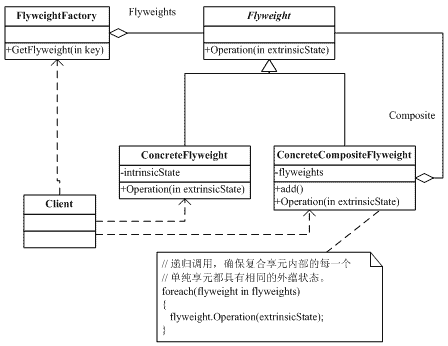


Tomcat

### Flyweight(享元模式) – 结构模式(4.6)



#### 模式概述



#### 模式结构

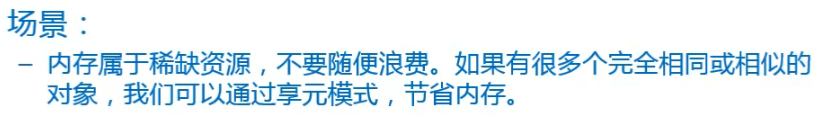
* 抽象享元(Flyweight)角色：此角色是所有的具体享元类的超类，为这些类规定出需要实现的公共接口。那些需要外蕴状态(External State)的操作可以通过调用商业方法以参数形式传入。
* 具体享元(ConcreteFlyweight)角色：实现抽象享元角色所规定的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。享元对象的内蕴状态必须与对象所处的周围环境无关，从而使得享元对象可以在系统内共享的。
* 享元工厂(FlyweightFactory)角色：本角色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象的时候，享元工厂角色会检查系统中是否已经有一个复合要求的享元对象。如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个合适的享元对象。
* 客户端(Client)角色：本角色需要维护一个对所有享元对象的引用。本角色需要自行存储所有享元对象的外蕴状态。

#### 角色协作

#### 作用效果

#### 注意事项

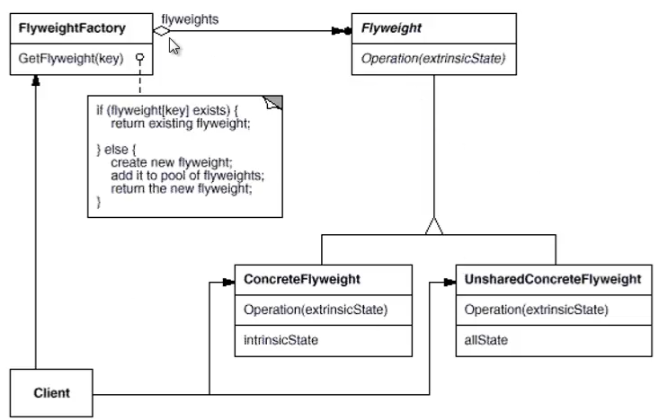
#### 使用场景

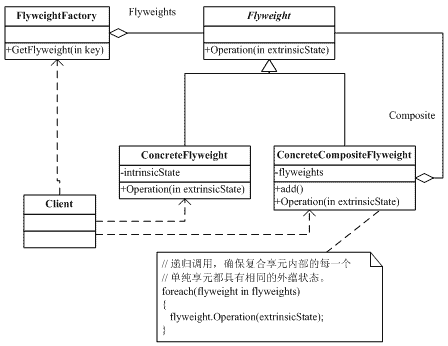


#### 应用示例

#### 相关模式

#### 模式总结





### Proxy(代理模式) – 结构模式(4.7)

为其他对象提供一种[代理](http://baike.baidu.com/view/24001.htm" \t "_blank)以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

1.一个是真正的你要访问的对象(目标类)，一个是代理对象,真正对象与代理。

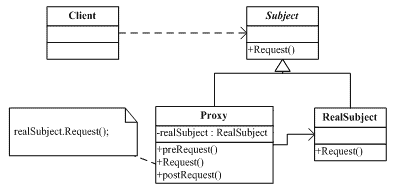
2.对象实现同一个接口,先访问代理类再访问真正要访问的对象。

**静态代理：**

静态代理是由程序员创建或工具生成代理类的源码，再编译代理类。所谓静态也就是在程序运行前就已经存在代理类的字节码文件，代理类和委托类的关系在运行前就确定了。

**动态代理：**

动态代理是在实现阶段不用关心代理类，而在运行阶段才指定哪一个对象。



**抽象主题角色（Subject）：**声明了真实主题和代理主题的共同接口，这样一来在任何使用真实主题的地方都可以使用代理主题。

**代理主题（Proxy）角色：**代理主题角色内部含有对真是主题的引用，从而可以在任何时候操作真实主题对象；代理主题角色提供一个与真实主题角色相同的接口，以便可以在任何时候都可以替代真实主体；控制真实主题的应用，负责在需要的时候创建真实主题对象（和删除真实主题对象）；代理角色通常在将客户端调用传递给真实的主题之前或之后，都要执行某个操作，而不是单纯的将调用传递给真实主题对象。

**真实主题角色（RealSubject）角色：**定义了代理角色所代表的真实对象。

**使用场景：**

1.远程代理（Remote Proxy）：为一个对象在不同的地址空间提供局部代表。这个不同的地址空间可以是在本机器中，也可是在另一台机器中。远程代理又叫做大使（Ambassador）。

可以将网络的细节隐藏起来，使得客户端不必考虑网络的存在。客户完全可以认为被代理的对象是局域的而不是远程的，而代理对象承担了大部分的网络通信工作。

2.虚代理（Virtual Proxy）：根据需要创建一个资源消耗较大的对象，使得此对象只在需要时才会被真正创建。

代理对象可以在必要的时候才将被代理的对象加载。代理可以对加载的过程加以必要的优化。当一个模块的加载十分耗费资源的时候，虚拟代理的优点就非常明显。

3.保护代理（Protection Proxy）：控制对一个对象的访问，如果需要，可以给不同的用户提供不同级别的使用权限。

可以在运行时间对用户的有关权限进行检查，然后在核实后决定将调用传递给被代理的对象。

4. 智能引用代理（Smart Reference Proxy）：当一个对象被引用时，提供一些额外的操作，比如将对此对象调用的次数记录下来等。

**代理模式总结：**

代理模式在很多情况下都非常有用,特别是你想强行控制一个对象的时候,比如:延迟加载,监视状态变更的方法等等

 1、“增加一层间接层”是软件系统中对许多负责问题的一种常见解决方法。在面向对象系统中，直接使用某些对象会带来很多问题，作为间接层的proxy对象便是解决这一问题的常用手段。

2、具体proxy设计模式的实现方法、实现粒度都相差很大，有些可能对单个对象作细粒度的控制，有些可能对组件模块提供抽象代理层，在架构层次对对象作proxy。

3、proxy并不一定要求保持接口的一致性，只要能够实现间接控制，有时候损及一些透明性是可以接受的。

**优点：**

1.职责清晰，真实的角色就是实现实际的[业务逻辑](http://baike.baidu.com/view/1642754.htm" \t "_blank)，不用关心其他非本职责的事务，通过后期的代理完成一件完成事务，附带的结果就是编程简洁清晰。

2.代理对象可以在客户端和目标对象之间起到中介的作用，这样起到了中介的作用和保护了目标对象的作用。

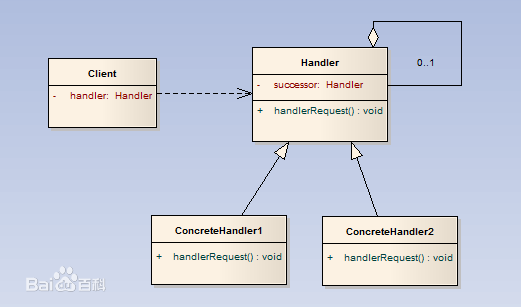
3.高扩展性

**注：**实现代理有聚合别用继承，以防类爆炸

**示例：**模拟JDK之Proxy

## 行为模式(11)

### Chain of Responsibility(责任链模式) – 行为模式(5.1)

责任链模式是一种[设计模式](http://baike.baidu.com/view/66964.htm" \t "_blank)。在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。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。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。 

**抽象处理者(Handler)角色：**定义出一个处理请求的接口。如果需要，接口可以定义出一个方法，以设定和返回对下家的引用。这个角色通常由一个抽象类或接口实现。

**具体处理者(ConcreteHandler)角色：**具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家。由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。

**使用场景：**

1.有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定；

2.在不明确指定接收者的情况下，向多个对象中的一个提交一个请求；

3.处理一个请求的对象集合应被动态指定。

**优点：**

1.降低耦合度 ：该模式使得一个对象无需知道是其他哪一个对象处理其请求。对象仅需知道该请求会被“正确”地处理。接收者和发送者都没有对方的明确的信息，且链中的对象不需知道链的结构。

2.职责链可简化对象的相互连接 :    结果是，职责链可简化对象的相互连接。它们仅需保持一个指向其后继者的引用，而不需保持它所有的候选接受者的引用。

3.增强了给对象指派职责( R e s p o n s i b i l i t y )的灵活性 ：当在对象中分派职责时，职责链给你更多的灵活性。你可以通过在运行时刻对该链进行动态的增加或修改来增加或改变处理一个请求的那些职责。你可以将这种机制与静态的特例化处理对象的继承机制结合起来使用。

4.增加新的请求处理类很方便

**缺点：**

1. 不能保证请求一定被接收。既然一个请求没有明确的接收者，那么就不能保证它一定会被处理 —该请求可能一直到链的末端都得不到处理。一个请求也可能因该链没有被正确配置而得不到处理。

2. 系统性能将受到一定影响，而且在进行代码调试时不太方便；可能会造成循环调用。

**示例：**模拟Filter，Tomcat

### Command(命令模式) – 行为模式(5.2)

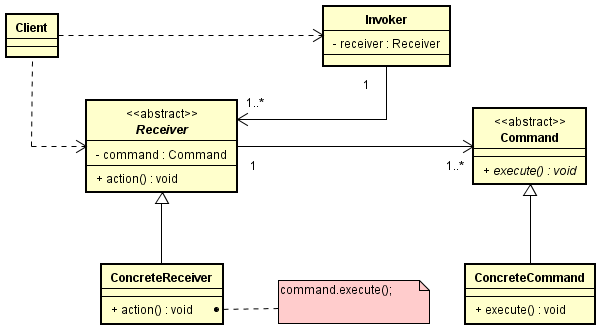
**别名：**行动（Action）模式或交易（Transaction）模式

命令模式把一个请求或者操作封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

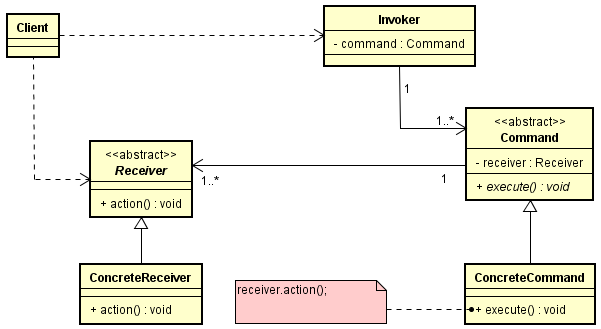
命令模式是对命令的封装。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。

每一个命令都是一个操作：请求的一方发出请求要求执行一个操作；接收的一方收到请求，并执行操作。命令模式允许请求的一方和接收的一方独立开来，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，以及操作是否被执行、何时被执行，以及是怎么被执行的。

**一个Receiver对应多个Command：**



**一个Command对应多个Receiver：（下面的解释就是针对这种的）**



**客户（Client）角色：**创建具体命令(ConcreteCommand)对象并确定其接收者。

**命令（Command）角色：**声明了一个给所有具体命令类的抽象接口。这是一个抽象角色。

**具体命令（ConcreteCommand）角色：**定义一个接受者和行为之间的弱耦合；实现execute()方法，负责调用接收者的相应操作。execute()方法通常叫做执方法。

**请求者（Invoker）角色：**负责调用命令对象执行请求，相关的方法叫做行动方法。

**接收者（Receiver）角色：**负责具体实施和执行一个请求。任何一个类都可以成为接收者，实施和执行请求的方法叫做行动方法。

**使用场景：**

1.可用过程语言中的回调（callback）函数表达这种参数化机制。使用命令模式作为"CallBack"在面向对象系统中的替代。"CallBack"讲的便是先将一个函数登记上，然后在以后调用此函数。

2.需要在不同的时间指定请求、将请求排队。一个命令对象和原先的请求发出者可以有不同的生命期。换言之，原先的请求发出者可能已经不在了，而命令对象本身仍然是活动的。这时命令的接收者可以是在本地，也可以在网络的另外一个地址。命令对象可以在串形化之后传送到另外一台机器上去。

3.系统需要支持命令的撤消(undo)。命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用undo()方法，把命令所产生的效果撤销掉。命令对象还可以提供redo()方法，以供客户端在需要时，再重新实施命令效果。

4.如果一个系统要将系统中所有的数据更新到日志里，以便在系统崩溃时，可以根据日志里读回所有的数据更新命令，重新调用execute()方法一条一条执行这些命令，从而恢复系统在崩溃前所做的数据更新。

5.一个系统需要支持交易(Transaction)。一个交易结构封装了一组数据更新命令。使用命令模式来实现交易结构可以使系统增加新的交易类型。

6.用构建在原语操作上的高层操作构造一个系统。这样一种结构在支持事务（Transaction）的信息系统中很常见。一个事务封装了对数据的一组变动。Command模式提供了对事务进行建模的方法。Command有一个公共的接口，使得你可以用同一种方式调用所有的事务。同时使用该模式也易于添加新事务以扩展系统。

**优点：**

1.命令模式使新的命令很容易地被加入到系统里。

2.允许接收请求的一方决定是否要否决（Veto）请求。

3.能较容易地设计一个命令队列。

4.可以容易地实现对请求的Undo和Redo。

5.在需要的情况下，可以较容易地将命令记入日志。

6.命令模式把请求一个操作的对象与知道怎么执行一个操作的对象分割开。

7.命令类与其他任何别的类一样，可以修改和推广。

8.你可以把命令对象聚合在一起，合成为合成命令。比如宏命令便是合成命令的例子。合成命令是合成模式的应用。

9.由于加进新的具体命令类不影响其他的类，因此增加新的具体命令类很容易。

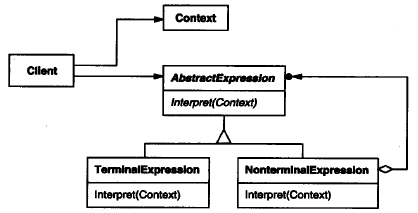
**缺点：**

使用命令模式会导致某些系统有过多的具体命令类。某些系统可能需要几十个，几百个甚至几千个具体命令类，这会使命令模式在这样的系统里变得不实际。

**示例：**MM之命令，Tomcat

### Interpreter(解释器模式) – 行为模式(5.3)

#### 模式概述



**注：在编写解释器的时候用，如velocity用到，正则表达式内部解析用等！工作中很少用！**

#### 模式结构

#### 角色协作

#### 作用效果

#### 注意事项

#### 使用场景

#### 应用示例

#### 相关模式

#### 模式总结

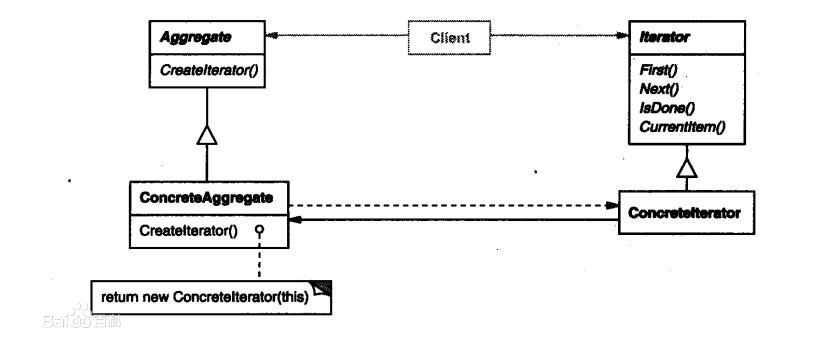
### Iterator(迭代器模式) – 行为模式(5.4)

#### 模式概述

别名：游标（Cursor）模式

迭代器提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。

当你需要访问一个聚合对象，而且不管这些对象是什么都需要遍历的时候，就应该考虑使用迭代器模式。另外，当需要对聚集有多种方式遍历时，可以考虑去使用迭代器模式。迭代器模式为遍历不同的聚集结构提供如开始、下一个、是否结束、当前哪一项等统一的接口。



#### 模式结构

* 迭代器角色（Iterator）：迭代器角色负责定义访问和遍历元素的接口。
* 具体迭代器角色（Concrete Iterator）：具体迭代器角色要实现迭代器接口，并要记录遍历中的当前位置。
* 容器角色（Aggregate）：容器角色负责提供创建具体迭代器角色的接口。
* 具体容器角色（Concrete Aggregate）：具体容器角色实现创建具体迭代器角色的接口——这个具体迭代器角色于该容器的结构相关。

#### 角色协作

客户端操作迭代器来遍历集合，迭代器由具体容器角色产生。

#### 作用效果

简化了遍历方式，对于对象集合的遍历，还是比较麻烦的，对于数组或者有序列表，我们尚可以通过游标来取得，但用户需要在对集合了解很清楚的前提下，自行遍历对象，但是对于hash表来说，用户遍历起来就比较麻烦了。而引入了迭代器方法后，用户用起来就简单的多了。

可以提供多种遍历方式，比如说对有序列表，我们可以根据需要提供正序遍历，倒序遍历两种迭代器，用户用起来只需要得到我们实现好的迭代器，就可以方便的对集合进行遍历了。

封装性良好，用户只需要得到迭代器就可以遍历，而对于遍历算法则不用去关心。

#### 注意事项

对于比较简单的遍历（像数组或者有序列表），使用迭代器方式遍历较为繁琐，大家可能都有感觉，像ArrayList，我们宁可愿意使用for循环和get方法来遍历集合。

#### 使用场景

迭代器模式是与集合共生共死的，一般来说，我们只要实现一个集合，就需要同时提供这个集合的迭代器，就像java中的Collection，List、Set、Map等，这些集合都有自己的迭代器。假如我们要实现一个这样的新的容器，当然也需要引入迭代器模式，给我们的容器实现一个迭代器。

       但是，由于容器与迭代器的关系太密切了，所以大多数语言在实现容器的时候都给提供了迭代器，并且这些语言提供的容器和迭代器在绝大多数情况下就可以满足我们的需要，所以现在需要**我们自己去实践迭代器模式的场景还是比较少见的**，我们只需要使用语言中已有的容器和迭代器就可以了。

#### 应用示例

模拟JDK之Iterator

#### 相关模式

#### 模式总结

      迭代器模式（Iterator）就是分离了聚合对象的遍历行为，抽象出一个迭代器来负责，这样既可以做到不暴露集合的内部结构，又可让外部代码透明的访问集合内部数据。

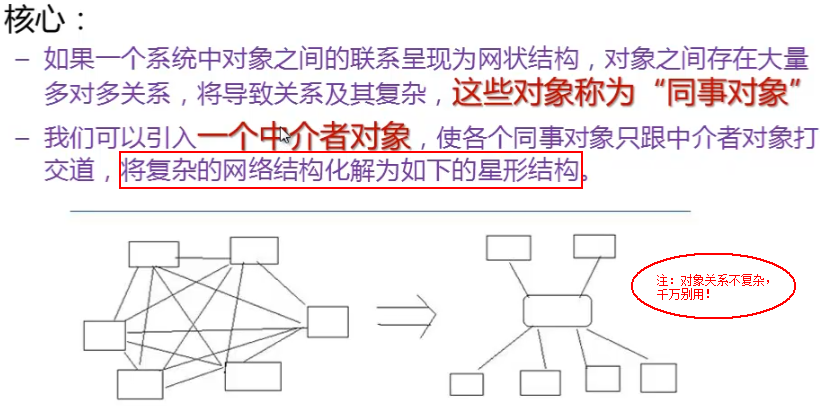
### Mediator(中介模式) – 行为模式(5.5)

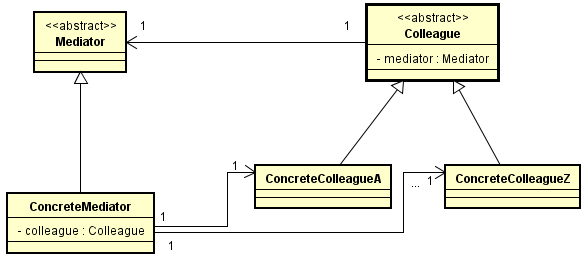
**将多对多关系解耦成一对多关系！！！**

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

通过定义我们可以看出中介者主要是通过中介对象来封装对象之间的关系，使之各个对象在不需要知道其他对象的具体信息情况下通过中介者对象来与之通信。同时通过引用中介者对象来减少系统对象之间关系，提高了对象的可复用和系统的可扩展性。

但是就是因为中介者对象封装了对象之间的关联关系，导致中介者对象变得比较庞大，所承担的责任也比较多。它需要知道每个对象和他们之间的交互细节，如果它出问题，将会导致整个系统都会出问题。所以它比较容易应用也很容易误用。故当系统中出现了“多对多”交互复杂的关系群时，千万别急着使用中介者模式，你首先需要做的就是反思你的系统在设计上是不是合理。





**中介者角色（Mediator）：**抽象中介者。定义了同事对象到中介者对象之间的接口。

**具体中介者角色（ConcreteMediator）：** 具体中介者。实现抽象中介者的方法，它需要知道所有的具体同事类，同时需要从具体的同事类那里接收信息，并且向具体的同事类发送信息。

**同事角色（Colleague）：** 抽象同事类。

**具体同事角色（ConcreteColleague:）：**具体同事类。每个具体同事类都只需要知道自己的行为即可，但是他们都需要认识中介者。

**使用场景：**

1.一组定义良好的对象，现在要进行复杂的通信。产生的相互依赖关系结构混乱且难以理解。

2.一个对象引用其他很多对象并且直接与这些对象通信,导致难以复用该对象。

3.想定制一个分布在多个类中的行为，而又不想生成太多的子类。

可以看出，中介对象主要是用来封装行为的，行为的参与者就是那些对象，但是通过中介者，这些对象不用相互知道。

**中介模式总结：**

在中介者模式中中介者对象处于核心地位，因为它定义了整个系统中所有具体同事类之间的关系。在整个系统中它主要承担两个方面的责任。

1.结构上起到**中转**作用。通过中介者对象对关系的封装，使得具体的同事类不再需要显示的引用其他对象，它只需要通过中介者就可以完成与其他同事类之间的通信。

2.行为上起到**协作**作用。中介者对同事类之间的关系进行封装，同事类在不需要知道其他对象的情况下通过中介者与其他对象完成通信。在这个过程中同事类是不需要指明中介者该如何做，中介者可以根据自身的逻辑来进行协调，对同事的请求进一步处理，将同事成员之间的关系行为进行分离和封装。

在中介者模式中通过引用中介者对象，将系统中有关的对象所引用的其他对象数目减少到最少。它简化了系统的结构，将系统由负责的网状结构转变成简单的星形结构，中介者对象在这里起到中转和协调作用。

中介者类是中介者模式的核心，它对整个系统进行控制和协调，简化了对象之间的交互，还可以对对象间的交互进行进一步的控制。

通过使用中介者模式，具体的同事类可以独立变化，通过引用中介者可以简化同事类的设计和实现。

由于中介者对对象的关系进行了封装，使得各个同事类之间的耦合减少了，使得他们可以独立改变和复用。

就是由于中介者对象需要知道所有的具体同事类，封装具体同事类之间相互关系，导致中介者对象变得非常复杂，系统维护起来较为困难。

**优点：**

1.降低了系统对象之间的耦合性，使得对象易于独立的被复用。

2.提高系统的灵活性，使得系统易于扩展和维护。

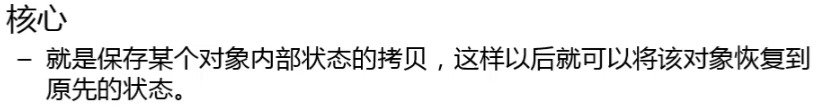
**缺点：**

中介者对象封装了系统中对象之间的相互关系，导致其变得非常复杂，使得系统维护比较困难。一旦这个中介对象出现了问题，那么整个系统就会受到重大的影响。

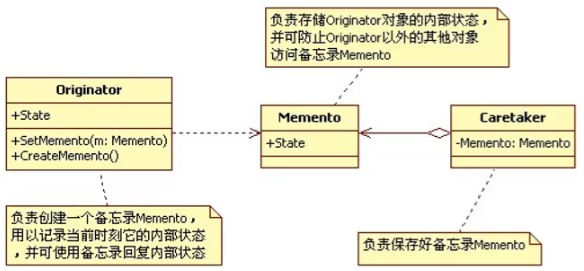
**注：**和外观模式相通，很多模式重点是语义的区别而非语法的区别。

**示例：**模拟房屋中介，MVC模式

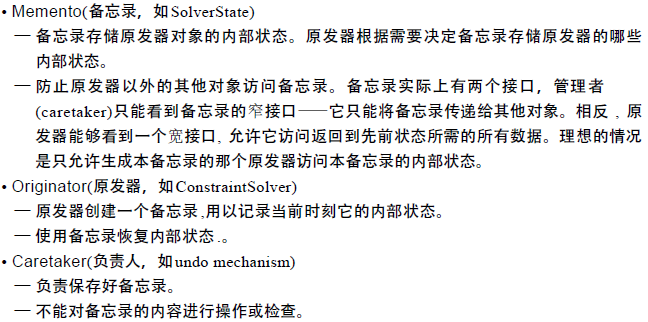
### Memento(备忘录模式)- 行为模式(5.6)



#### 模式概述



#### 模式结构



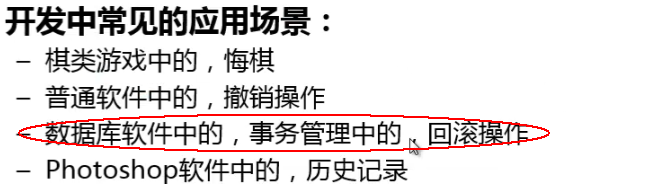
#### 角色协作

#### 作用效果

#### 注意事项

#### 使用场景

#### 应用示例



#### 相关模式

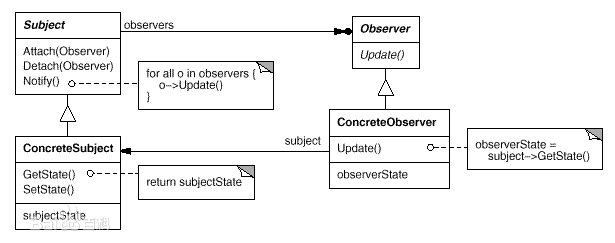
#### 模式总结

### Observer(观察者设计模式) – 行为模式(5.7)

**别名：**源-监听器（Source/Listener）模式

观察者模式定义了一种**一对多**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，使它们能够自动更新自己。

一个软件系统常常要求在某一个对象的状态发生变化的时候，某些其它的对象做出相应的改变。做到这一点的设计方案有很多，但是为了使系统能够易于复用，应该选择低耦合度的设计方案。减少对象之间的耦合有利于系统的复用，但是同时设计师需要使这些低耦合度的对象之间能够维持行动的协调一致，保证高度的协作（Collaboration）。观察者模式是满足这一要求的各种设计方案中最重要的一种。



**抽象主题（Subject）角色**：主题角色把所有的观察者对象的引用保存在一个列表里；每个主题都可以有任何数量的观察者。主题提供一个接口可以加上或撤销观察者对象；

抽象主题角色，有时又叫做抽象被观察者角色，可以用一个抽象类或者一个接口实现；在具体的情况下也不排除使用具体类实现。

**抽象观察者（Observer）角色**：为所有的具体观察者定义一个接口，在得到通知时更新自己；

抽象观察者角色，可以用一个抽象类或者一个接口实现；在具体的情况下也不排除使用具体类实现。

**具体主题（ConcreteSubject）角色**：保存对具体观察者对象有用的内部状态；在这种内部状态改变时给其观察者发出一个通知；

具体主题角色，通常用一个具体子类实现。

**具体观察者（ConcreteObserver）角色**：保存一个指向具体主题对象的引用；和一个与主题的状态相符的状态。具体观察者角色实现抽象观察者角色所要求的更新自己的接口，以便使本身的状态与主题的状态自恰。

具体观察者角色，通常用一个具体子类实现。

**观察者模式的总结:**

**优点**：

第一、观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者列表，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体观察者，它只知道它们都有一个共同的接口。

第二、观察者模式支持广播通讯。被观察者会向所有的登记过的观察者发出通知

**缺点：**

第一、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

　　第二、如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式是要特别注意这一点。

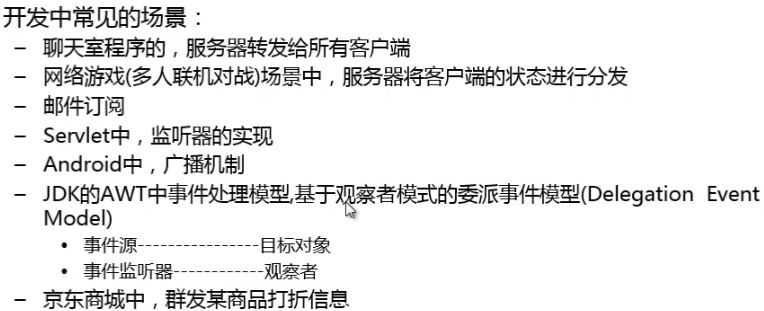
　　第三、如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以自恰的方式进行的。

第四、虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察的对象是怎么发生变化的。

**观察者模式与其它模式的关系：**

观察者模式使用了备忘录模式(Memento Pattern)暂时将观察者对象存储在被观察者对象里面。

**示例：**模拟GUI组件的事件监听，Tomcat



### State(状态模式) – 行为模式(5.8)

#### 模式概述

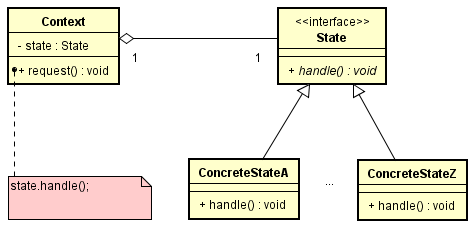
当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。

状态模式主要解决的是当控制一个对象状态的条件表达式过于复杂时的情况。把状态的判断逻辑转移到表示不同状态的一系列类中，可以把复杂的判断逻辑简化。

在很多情况下，一个对象的行为取决于一个或多个动态变化的属性，这样的属性叫做状态，这样的对象叫做有状态的(stateful)对象，这样的对象状态是从事先定义好的一系列值中取出的。当一个这样的对象与外部事件产生互动时，其内部状态就会改变，从而使得系统的行为也随之发生变化。

将主对象和其状态分离，状态对象负责主对象的状态转换，使主对象功能减轻

#### 模式结构



**环境类（Context）:**  维护一个ConcreteState子类的实例，这个实例定义当前状态。

**抽象状态类（State）:**  定义一个接口以封装与Context的一个特定状态相关的行为。

**具体状态类（ConcreteState）:** 每一子类实现一个与Context的一个状态相关的行为。

#### 角色协作

1. 首先，定义一个State接口。在这个接口内，Context的每个动作都有一个对应的方法；
2. 然后，为Context中的每个状态构建对应的状态类。这些状态类负责在对应的转态下进行Context的行为；
3. 最后，摆脱旧的条件代码，将动作委托到状态类。

#### 作用效果

1. 它将与特定状态相关的行为局部化，并且将不同状态的行为分割开来: State模式将所有与一个特定的状态相关的行为都放入一个对象中。因为所有与状态相关的代码都存在于某一个State子类中, 所以通过定义新的子类可以很容易的增加新的状态和转换。另一个方法是使用数据值定义内部状态并且让 Context操作来显式地检查这些数据。但这样将会使整个Context的实现中遍布看起来很相似的条件if else语句或switch case语句。增加一个新的状态可能需要改变若干个操作, 这就使得维护变得复杂了。State模式避免了这个问题, 但可能会引入另一个问题, 因为该模式将不同状态的行为分布在多个State子类中。这就增加了子类的数目，相对于单个类的实现来说不够紧凑。但是如果有许多状态时这样的分布实际上更好一些, 否则需要使用巨大的条件语句。正如很长的过程一样，巨大的条件语句是不受欢迎的。它们形成一大整块并且使得代码不够清晰，这又使得它们难以修改和扩展。 State模式提供了一个更好的方法来组织与特定状态相关的代码。决定状态转移的逻辑不在单块的 i f或s w i t c h语句中, 而是分布在State子类之间。将每一个状态转换和动作封装到一个类中，就把着眼点从执行状态提高到整个对象的状态。这将使代码结构化并使其意图更加清晰。
2. 它使得状态转换显式化: 当一个对象仅以内部数据值来定义当前状态时 , 其状态仅表现为对一些变量的赋值，这不够明确。为不同的状态引入独立的对象使得转换变得更加明确。而且, State对象可保证Context不会发生内部状态不一致的情况，因为从 Context的角度看，状态转换是原子的—只需重新绑定一个变量(即Context的State对象变量)，而无需为多个变量赋值
3. State对象可被共享 如果State对象没有实例变量—即它们表示的状态完全以它们的类型来编码—那么各Context对象可以共享一个State对象。当状态以这种方式被共享时, 它们必然是没有内部状态, 只有行为的轻量级对象。

#### 注意事项

1. 状态模式的使用必然会增加系统类和对象的个数。
2. 状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱。

#### 使用场景

1. 一个对象的行为取决于它的状态, 并且它必须在运行时刻根据状态改变它的行为。
2. 代码中包含大量与对象状态有关的条件语句:一个操作中含有庞大的多分支的条件（if else(或switch case)语句，且这些分支依赖于该对象的状态。这个状态通常用一个或多个枚举常量表示。通常 , 有多个操作包含这一相同的条件结构。 State模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

#### 应用示例

边缘模拟计算器

#### 相关模式

和策略模式相通，很多模式重点是语义的区别而非语法的区别

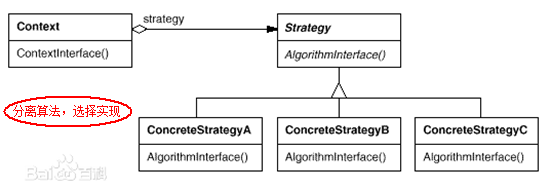
#### 模式总结

### Strategy(策略模式) – 行为模式(5.9)

策略模式的用意是针对一组算法，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。

策略模式是对算法的包装，是把使用算法的责任和算法本身分割开，委派给不同的对象管理。策略模式通常把一个系列的算法包装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是："准备一组算法，并将每一个算法封装起来，使得它们可以互换。"

软件开发过程中，通常要设计相应的算法，比如电子网站中经常有折扣活动，并且用户等级不同，享受的折扣也不一样。通过使用策略模式将折扣算法封装起来，不仅可以用于网商平台，还可以用于手持设备中折扣活动的实现。封装的特性确保了扩展算法和增加新算法的简便性。



**环境（Context）角色：**持有一个Strategy类的引用。

**抽象策略（Strategy）角色：**这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

**具体策略（ConcreteStrategy）角色：**包装了相关的算法或行为。

**策略模式总结：**

**如何使用策略模式：**

策略模式仅仅封装算法，提供新算法插入到已有系统中，以及老算法从系统中"退休"的方便，策略模式并不决定在何时使用何种算法。

在下面的情况下应当考虑使用策略模式：

1. 如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。

2. 一个系统需要动态地在几种算法中选择一种。那么这些算法可以包装到一个个的具体算法类里面，而这些具体算法类都是一个抽象算法类的子类。换言之，这些具体算法类均有统一的接口，由于多态性原则，客户端可以选择使用任何一个具体算法类，并只持有一个数据类型是抽象算法类的对象。

3. 一个系统的算法使用的数据不可以让客户端知道。策略模式可以避免让客户端涉及到不必要接触到的复杂的和只与算法有关的数据。

4. 如果一个对象有很多的行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。此时，使用策略模式，把这些行为转移到相应的具体策略类里面，就可以避免使用难以维护的多重条件选择语句，并体现面向对象设计的概念。

**优点：**

1. 策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把公共的代码移到父类里面，从而避免重复的代码。

2. 策略模式提供了可以替换继承关系的办法。继承可以处理多种算法或行为。如果不是用策略模式，那么使用算法或行为的环境类就可能会有一些子类，每一个子类提供一个不同的算法或行为。但是，这样一来算法或行为的使用者就和算法或行为本身混在一起。决定使用哪一种算法或采取哪一种行为的逻辑就和算法或行为的逻辑混合在一起，从而不可能再独立演化。继承使得动态改变算法或行为变得不可能。

3. 使用策略模式可以避免使用多重条件转移语句。多重转移语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重转移语句里面，比使用继承的办法还要原始和落后。

**缺点：**

1. 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类。换言之，策略模式只适用于客户端知道所有的算法或行为的情况。

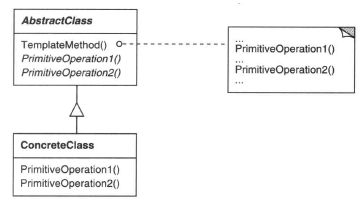
2. 策略模式造成很多的策略类。有时候可以通过把依赖于环境的状态保存到客户端里面，而将策略类设计成可共享的，这样策略类实例可以被不同客户端使用。换言之，可以使用享元模式来减少对象的数量。

**示例：**模拟JDK之Comparable和Comparator

### [Template Method](http://baike.baidu.com/view/1191246.htm" \t "_blank)(模版方法模式) – 行为模式(5.10)

SpringMVC用到

#### 模式概述



#### 模式结构

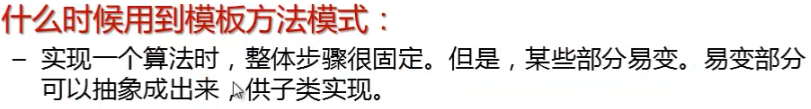
* 抽象模版（AbstractClass）角色：定义了一个或多个抽象操作，以便让子类实现。这些抽象操作叫做基本操作，它们是一个顶级逻辑的组成步骤。定义并实现了一个模版方法。这个模版方法一般是一个具体方法，它给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中，推迟到子类实现。顶级逻辑也有可能调用一些具体方法。
* 具体模版（ConcreteClass）角色：实现父类所定义的一个或多个抽象方法，它们是一个顶级逻辑的组成步骤。每一个抽象模版角色都可以有任意多个具体模版角色与之对应，而每一个具体模版角色都可以给出这些抽象方法（也就是顶级逻辑的组成步骤）的不同实现，从而使得顶级逻辑的实现各不相同。

#### 角色协作

#### 作用效果

#### 注意事项

#### 使用场景



#### 应用示例



#### 相关模式

#### 模式总结

模版方法模式中的方法可以分为两大类：模版方法（Template Method）和基本方法（Primitive Method)。

**模版方法**

一个模版方法是定义在抽象类中的，把基本操作方法组合在一起形成一个总算法或一个总行为的方法。这个模版方法一般会在抽象类中定义，并由子类不加以修改地完全继承下来。

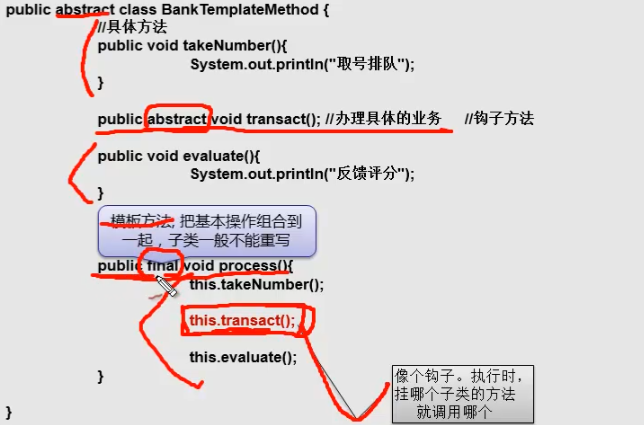
**基本方法**

基本方法又可以分为三种：抽象方法（Abstract Method）、具体方法（Concrete Method）和钩子方法（Hook Method）。

抽象方法：一个抽象方法由抽象类声明，由具体子类实现。

具体方法：一个具体方法由抽象类声明并实现，而子类并不实现或置换。

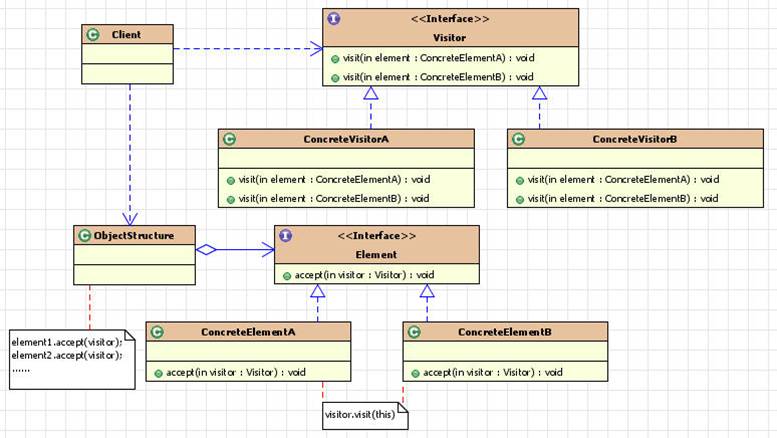
钩子方法：一个钩子方法由抽象类声明并实现，而子类会加以扩展。通常抽象类给出的实现是一个空实现，作为方法的默认实现。



### Visitor(访问者模式) – 行为模式(5.11)

#### 模式概述

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则可以保持不变。



#### 模式结构

* 抽象访问者（Visitor）角色：为该对象结构中ConcreteElement的每一个类声明一个Visit操作。该操作的名字和特征标识了发送Visit请求给该访问者的那个类。这使得访问者可以确定正被访问元素的具体的类。这样访问者就可以通过该元素的特定接口直接访问它。
* 具体访问者（ConcreteVisitor）角色：实现每个由Visitor声明的操作。每个操作实现本算法的一部分，而该算法片断乃是对应于结构中对象的类。ConcreteVsitor为该算法提供了上下文并存储它的局部状态。这一状态常常在遍历该结构的过程中累积结果。
* 抽象元素（Element）角色：定义一个Accept操作，它以一个访问者为参数。
* 具体元素（ConcreteElement）角色：实现Accept操作，该操作以一个访问者为参数。
* 结构对象（ObiectStructure）角色：可以遍历结构中的所有元素；如果需要，提供一个高层次的接口让访问者对象可以访问每一个元素；如果需要，可以设计成一个复合对象或者一个聚集，如列（List）或集合（Set）。

#### 角色协作

一个使用Visitor模式的客户必须创建一个ConcreteVisitor对象，然后遍历该对象结构，并用该访问者访问每一个元素。

当一个元素被访问时，它调用对应于它的类的Visitor操作。如果必要，该元素将自身作为这个操作的一个参数以便该访问者访问它的状态。

#### 作用效果

* 访问者模式使得易于增加新的操作
* 访问者集中相关的操作而分离无关的操作

#### 注意事项

* 增加新的ConcreteElement类很困难

#### 使用场景

1. 一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。
2. 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。Visitor使得你可以将相关的操作集中起来定义在一个类中。
3. 当该对象结构被很多应用共享时，用Vi s i t o r模式让每个应用仅包含需要用到的操作。
4. 定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。

#### 应用示例

Javac

#### 相关模式

* Composite：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。
* Interpreter：访问者可以用于解释。

#### 模式总结

可以将Visitor聚合到ObiectStructure中。

# 实战补充

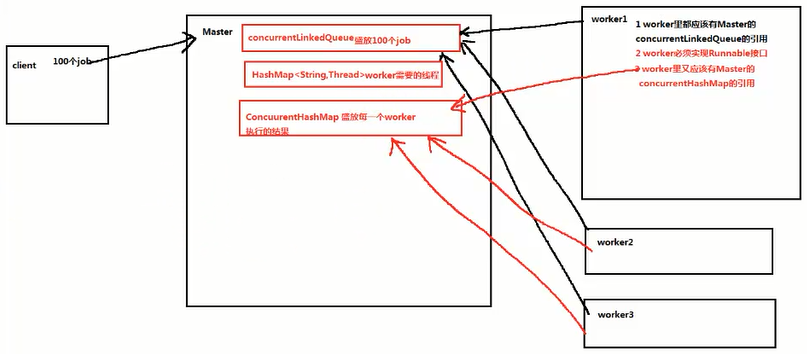
## Future模式

FutureTask Callable

减小响应时间，提高吞吐量

## Master-Worker模式（并行计算模式）

Future+Callable+ThreadPool+CountDownLatch



## 生产者-消费者模式