1. **UML**

**1.1 类图**

在UML的类图中，常见的有以下几种关系: 泛化（Generalization）, 实现（Realization）, 关联（Association), 聚合（Aggregation）, 组合(Composition), 依赖(Dependency)

**1.1.1 泛化（Generalization）**

【泛化关系】：是一种继承关系, 表示一般与特殊的关系, 它指定了子类如何特化父类的所有特征和行为. 例如：老虎是动物的一种, 即有老虎的特性也有动物的共性.

【箭头指向】：带三角箭头的实线，箭头指向父类



**1.1.2 实现（Realization）**

【实现关系】：是一种类与接口的关系, 表示类是接口所有特征和行为的实现.

【箭头指向】：带三角箭头的虚线，箭头指向接口



**1.1.3. 关联（Association)**

【关联关系】：是一种拥有的关系, 它使一个类知道另一个类的属性和方法；如：老师与学生，丈夫与妻子

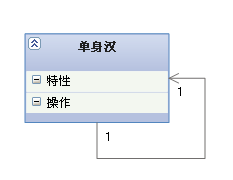
关联可以是双向的，也可以是单向的。双向的关联可以有两个箭头或者没有箭头，单向的关联有一个箭头。

【代码体现】：成员变量

【箭头及指向】：带普通箭头（或实心三角形箭头）的实心线，指向被拥有者



上图中，老师与学生是双向关联，老师有多名学生，学生也可能有多名老师。但学生与某课程间的关系为单向关联，一名学生可能要上多门课程，课程是个抽象的东西他不拥有学生。



上图为自身关联：

**1.1.4. 聚合（Aggregation）**

【聚合关系】：是整体与部分的关系, 且部分可离开整体而单独存在. 如车和轮胎是整体和部分的关系, 轮胎离开车仍然可以存在.

聚合关系是关联关系的一种，是强的关联关系；关联和聚合在语法上无法区分，必须考察具体的逻辑关系。

【代码体现】：成员变量

【箭头及指向】：带空心菱形的实心线，菱形指向整体



**1.1.5. 组合(Composition)**

【组合关系】：整体与部分关系, 但部分不能离开整体而单独存在.

如公司和部门是整体和部分的关系, 没有公司就不存在部门.

是关联关系的一种，是比聚合关系还要强的关系，要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期

【代码体现】：成员变量

【箭头及指向】：带实心菱形的实线，菱形指向整体



**1.1.6. 依赖(Dependency)**

【依赖关系】：是一种使用的关系, 即一个类的实现需要另一个类的协助, 所以要尽量不使用双向的互相依赖.

【代码表现】：局部变量、方法的参数或者对静态方法的调用

【箭头及指向】：带箭头的虚线，指向被使用者

****

各种关系的强弱顺序：

泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖

下面这张UML图，比较形象地展示了各种类图关系：

****

**1.2 序列图**

主要用于展示对象之间交互的顺序。

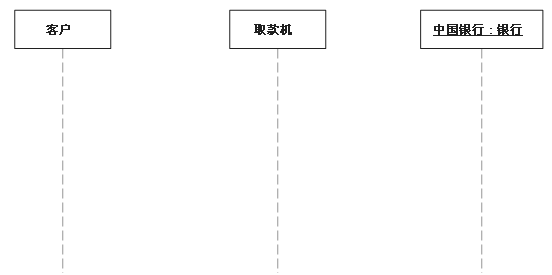
序列图将交互关系表示为一个二维图。纵向是时间轴，时间沿竖线向下延伸。横向轴代表了在协作中各独立对象的类元角色。类元角色用生命线表示。当对象存在时，角色用一条虚线表示，当对象的过程处于激活状态时，生命线是一个双道线。

消息用从一个对象的生命线到另一个对象生命线的箭头表示。箭头以时间顺序在图中从上到下排列。

序列图中涉及的元素：

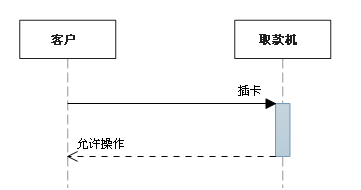
1. 生命线：

生命线名称可带下划线。当使用下划线时，意味着序列图中的生命线代表一个类的特定实体。

****

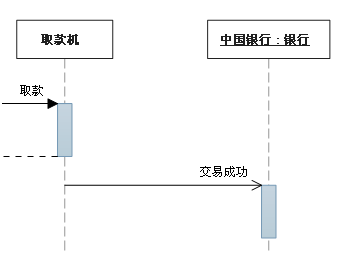
2. 同步消息

发送人在它继续之前，将等待同步消息响应

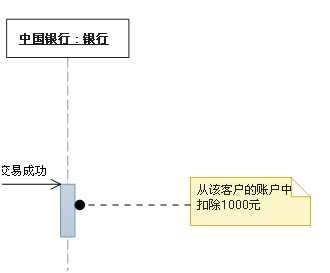
****

3.异步消息

在发送方继续之前，无需等待响应的消息

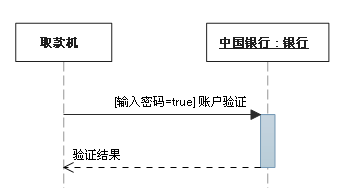
****

4. 注释

****

5.约束

约束的符号很简单；格式是: [Boolean Test]

****

6. 组合片段

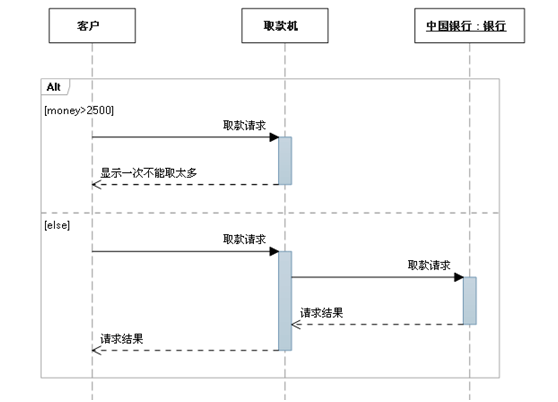
组合片段 用来解决交互执行的条件及方式。 它允许在序列图中直接表示逻辑组件，用于通过指定条件或子进程的应用区域，为任何生命线的任何部分定义特殊条件和子进程。

常用的组合片段有：

a. 抉择（Alt）

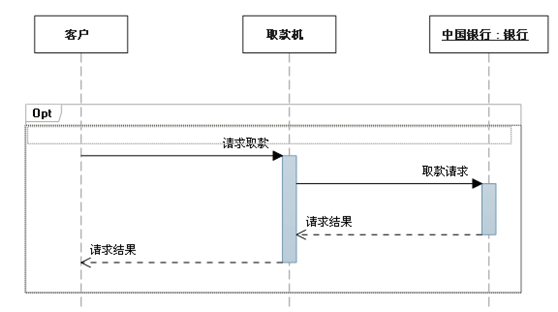
抉择用来指明在两个或更多的消息序列之间的互斥的选择，相当于经典的if..else..。

抉择在任何场合下只发生一个序列。 可以在每个片段中设置一个临界来指示该片段可以运行的条件。 else 的临界指示其他任何临界都不为 True 时应运行的片段。 如果所有临界都为 False 并且没有 else，则不执行任何片段。

****

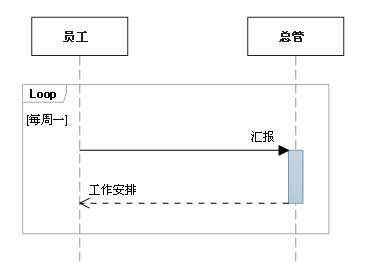
b. 选项（Opt）

包含一个可能发生或不发生的序列

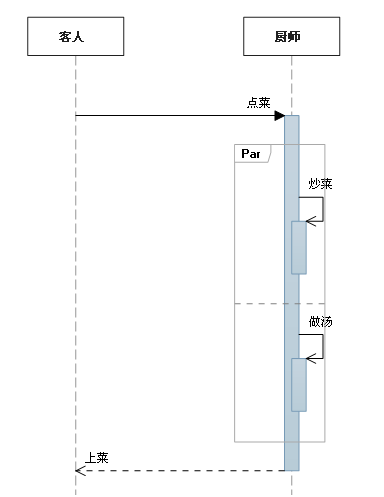
****

c. 循环（Loop）

片段重复一定次数。 可以在临界中指示片段重复的条件

****

d. 并行（Par）

****

下表列出了常用的组合片段：

片段类型

名称

说明

Opt

选项

包含一个可能发生或可能不发生的序列。 可以在临界中指定序列发生的条件。

Alt

抉择

包含一个片段列表，这些片段包含备选消息序列。 在任何场合下只发生一个序列。

可以在每个片段中设置一个临界来指示该片段可以运行的条件。 else 的临界指示其他任何临界都不为 True 时应运行的片段。 如果所有临界都为 False 并且没有 else，则不执行任何片段。

Loop

循环

片段重复一定次数。 可以在临界中指示片段重复的条件。

Loop 组合片段具有“Min”和“Max”属性，它们指示片段可以重复的最小和最大次数。 默认值是无限制。

Break

中断

如果执行此片段，则放弃序列的其余部分。 可以使用临界来指示发生中断的条件。

Par

并行

并行处理。 片段中的事件可以交错。

Critical

关键

用在 Par 或 Seq 片段中。 指示此片段中的消息不得与其他消息交错。

Seq弱顺序

有两个或更多操作数片段。 涉及同一生命线的消息必须以片段的顺序发生。 如果消息涉及的生命线不同，来自不同片段的消息可能会并行交错。

Strict强顺序

有两个或更多操作数片段。 这些片段必须按给定顺序发生。

有关如何解释序列的片段

默认情况下，序列图表明可能发生的一系列消息。 在运行的系统中，可能会出现您未选择显示在关系图上的其他消息。

以下片段类型可用于更改此释义：

片段类型 名称 说明

Consider考虑 指定此片段描述的消息列表。 其他消息可发生在运行的系统中，

但对此描述来说意义不大。在“Messages”属性中键入该列表。

Ignore 忽略 此片段未描述的消息列表。 这些消息可发生在运行的系统中，

但对此描述来说意义不大。在“Messages”属性中键入该列表。

Assert断言 操作数片段指定唯一有效的序列。 通常用在 Consider 或 Ignore 片段中。

Neg 否定 此片段中显示的序列不得发生。通常用在 Consider 或 Ignore 片段中。

===================================================

用例图主要用来描述 用户、需求、系统功能单元 之间的关系。它展示了一个外部用户能够观察到的系统功能模型图。

【用途】：帮助开发团队以一种可视化的方式理解系统的功能需求。

用例图所包含的元素如下：

1. 参与者(Actor)

表示与您的应用程序或系统进行交互的用户、组织或外部系统。用一个小人表示。



2. 用例(Use Case)

用例就是外部可见的系统功能，对系统提供的服务进行描述。 用椭圆表示



3. 子系统(Subsystem)

用来展示系统的一部分功能，这部分功能联系紧密。

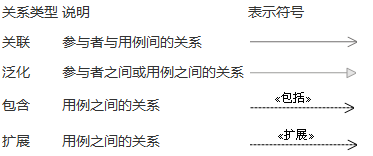




4. 关系

用例图中涉及的关系有：关联、泛化、包含、扩展；

如下表所示：



a. 关联(Association)

表示参与者与用例之间的通信，任何一方都可发送或接受消息。

【箭头指向】：指向消息接收方



b. 泛化(Inheritance)

就是通常理解的继承关系，子用例和父用例相似，但表现出更特别的行为；子用例将继承父用例的所有结构、行为和关系。子用例可以使用父用例的一段行为，也可以重载它。父用例通常是抽象的。

【箭头指向】：指向父用例



c. 包含(Include)

包含关系用来把一个较复杂用例所表示的功能分解成较小的步骤；

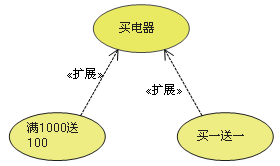
【箭头指向】：指向分解出来的功能用例



d. 扩展(Extend)

扩展关系是指 用例功能的延伸，相当于为基础用例提供一个附加功能。

【箭头指向】：指向基础用例

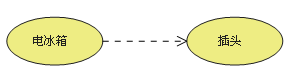


e. 依赖(Dependency)

以上4中关系，是UML定义的标准关系。 但VS2010的用例模型图中，添加了依赖关系，用带箭头的虚线表示

表示源用例依赖于目标用例；

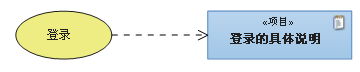
【箭头指向】：指向被依赖项



5. 项目(Artifact)

用例图虽然是用来帮助人们形象地理解功能需求，但却没多少人能够通看懂它。很多时候跟用户交流甚至用Excel都比用例图强，VS2010中引入了“项目”这样一个元素，以便让开发人员能够在用例图中链接一个普通文档。

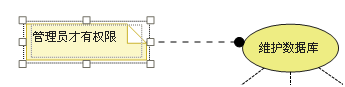
用依赖关系把某个用例依赖到项目上



然后把项目-》属性 的Hyperlink 设置到你的文档上

这样当你在用例图上 双击项目时，就会打开相关联的文档

6. 注释(Comment)



包含(include)、扩展(extend)、泛化(Inheritance) 的区别：

条件性：泛化中的子用例和include中的被包含的用例会无条件发生，而extend中的延伸用例的发生是有条件的；

直接性：泛化中的子用例和extend中的延伸用例为参与者提供直接服务，而include中被包含的用例为参与者提供间接服务。

对extend而言，延伸用例并不包含基础用例的内容，基础用例也不包含延伸用例的内容。

对Inheritance而言，子用例包含基础用例的所有内容及其和其他用例或参与者之间的关系；

一个用例图示例：



牢骚：

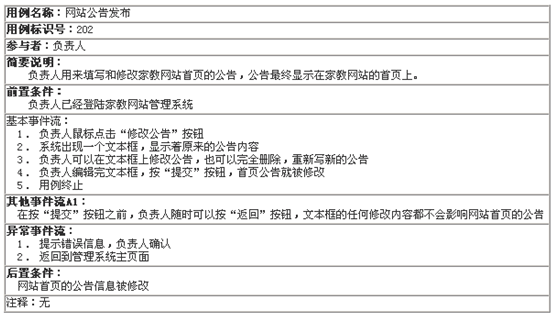
感觉用例图还不成熟，并不能很好地表达系统的需求， 没有UML背景的用户几乎不知道画的些什么。

其次，包含关系、扩展关系 的箭头符号竟然是同样的箭头，仅靠上方写个文字来加以区别，翻译成其他语言的话，几乎就不知道代表什么意思。 扩展关系的箭头朝向也很难理解，为何要指向基用例，而不指向扩展用例

VS2010添加的“项目”元素，是个很好的创新，能够在用例图中关联word,excel这些文档。但为什么不把这些功能直接集成到用例里面，双击用例就弹出一份文档岂不更容易理解，非要画蛇添足地加一个元件，仅仅为了提供个链接功能。

用例描述表：

鉴于用列图并不能清楚地表达功能需求，开发中大家通常用描述表来补充某些不易表达的用例，下图的表给大家提供一个参考：



1. **设计模式概述**

设计模式（Design Patterns）——可复用面向对象软件的基础

是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。

使用是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。

设计模式于己于他人于系统都是多赢的，使代码编制工程化，是软件工程的基石，如同大厦的一块块砖石一样。

项目中合理的运用设计模式可完美的解决很多问题，每种模式在现实中都有相应的原理与之对应，每一个模式描述了一个不断重复发生的问题，及该问题的核心解决方案，也是它能被广泛应用的原因。

**1.2 spring用到的设计模式**

1.工厂模式，这个很明显，在各种BeanFactory以及ApplicationContext创建中都用到了；

2.模版模式，这个也很明显，在各种BeanFactory以及ApplicationContext实现中也都用到了；

3.代理模式，在Aop实现中用到了JDK的动态代理；

4.策略模式，第一个地方，加载资源文件的方式，使用了不同的方法，比如：ClassPathResourece，FileSystemResource，ServletContextResource，UrlResource但他们都有共同的借口Resource；第二个地方就是在Aop的实现中，采用了两种不同的方式，JDK动态代理和CGLIB代理；

5.单例模式，这个比如在创建bean的时候。

http://ju.outofmemory.cn/entry/78206

**1.3 java JDK用到的设计模式**

**1.2.1 抽象工厂**

（通过创造性的方法来识别工厂本身，这又可以用于创建另一个抽象/接口类型）

|  |
| --- |
| javax.xml.parsers.DocumentBuilderFactory#newInstance()  javax.xml.transform.TransformerFactory#newInstance()  javax.xml.xpath.XPathFactory#newInstance() |

**1.2.2 建造者模式**

（通过创建方法识别返回实例本身）

|  |
| --- |
| java.lang.StringBuilder#append() （非线程安全）  java.lang.StringBuffer#append() （线程安全）  java.nio.ByteBuffer#put()（还CharBuffer，ShortBuffer，IntBuffer，LongBuffer  FloatBuffer和DoubleBuffer）  javax.swing.GroupLayout.Group#addComponent()  所有的实现 java.lang.Appendable |

**1.2.3工厂模式**

（可通过创建方法识别返回抽象/接口类型的实现）

|  |
| --- |
| java.util.Calendar#getInstance()  java.util.ResourceBundle#getBundle()  java.text.NumberFormat#getInstance()  java.nio.charset.Charset#forName()  java.net.URLStreamHandlerFactory#createURLStreamHandler(String) （每个协议返回单例对象）  java.util.EnumSet#of()  javax.xml.bind.JAXBContext#createMarshaller() 和其他类似的方法 |

**原型模式**

（通过创建方法识别，返回具有相同属性的其他实例）

|  |
| --- |
| java.lang.Object#clone()（班必须实施java.lang.Cloneable） |

**单例模式**

（通过创造性方法识别，每次返回相同的实例（通常是自己））

|  |
| --- |
| java.lang.Runtime#getRuntime()  java.awt.Desktop#getDesktop()  java.lang.System#getSecurityManager() |

**结构模式**

**适配器模式**

（可通过创建方法识别采用不同抽象/接口类型的实例，并返回自己/另一个抽象/接口类型的实现，其装饰/覆盖给定实例）

|  |
| --- |
| java.util.Arrays#asList()  java.util.Collections#list()  java.util.Collections#enumeration()java.io.InputStreamReader(InputStream)（返回a Reader）  java.io.OutputStreamWriter(OutputStream)（返回a Writer）  javax.xml.bind.annotation.adapters.XmlAdapter#marshal() 和 #unmarshal() |

**桥接模式**

（可以通过创建方法识别采用不同抽象/接口类型的实例，并返回自己的使用给定实例的抽象/接口类型的实现）

一个虚构的例子将会new LinkedHashMap(LinkedHashSet< K>, List< V>)返回一个不可修改的链接映射，它不会克隆，而是使用它们。该java.util.Collections#newSetFromMap()和singletonXXX()方法却接近。

**组合模式**

（通过将具有相同抽象/接口类型的实例的行为方法识别为树结构）

|  |
| --- |
| java.awt.Container#add(Component) （几乎全部摆动）  javax.faces.component.UIComponent#getChildren() |

**装饰器模式**

（通过创作方法识别采用相同抽象/接口类型的实例，添加额外的行为）

所有子类java.io.InputStream，OutputStream，Reader并Writer有一个构造函数取相同类型的实例。

java.util.Collections的checkedXXX()，synchronizedXXX()和unmodifiableXXX()方法。

javax.servlet.http.HttpServletRequestWrapper 和 HttpServletResponseWrapper

**门面模式**

（可通过内部使用不同独立抽象/接口类型实例的行为方法识别）

|  |
| --- |
| javax.faces.context.FacesContext，它在内部等使用抽象/接口类型LifeCycle，ViewHandler，NavigationHandler等等而没有终端用户具有担心它（它们然而通过注射覆写投放）。  javax.faces.context.ExternalContext，其在内部使用ServletContext，HttpSession，HttpServletRequest，HttpServletResponse，等。 |

**享元模式**

（使用缓存来加速大量小对象的访问时间）

|  |
| --- |
| java.lang.Integer#valueOf(int)（还Boolean，Byte，Character，Short，Long和BigDecimal） |

**代理模式**

（可通过创建方法识别，该方法返回给定的抽象/接口类型的实现，该类型依次代表/使用给定抽象/接口类型的不同实现）

|  |
| --- |
| java.lang.reflect.Proxy  java.rmi.\*  javax.ejb.EJB  javax.inject.Inject  javax.persistence.PersistenceContext |

**行为模式**

**责任链模式**

（通过行为方法识别（间接地）在队列中的相同抽象/接口类型的另一个实现中调用相同的方法）

|  |
| --- |
| java.util.logging.Logger#log()  javax.servlet.Filter#doFilter() |

**命令模式**

（可以通过抽象/接口类型中的行为方法识别，该方法在创建时由命令实现封装的不同抽象/接口类型的实现中调用方法）

|  |
| --- |
| 所有的实现 java.lang.Runnable  所有的实现 javax.swing.Action |

**解释器模式**

（通过行为方法识别，返回结构不同的实例/给定实例/类型的类型;请注意，解析/格式化不是模式的一部分，确定模式以及如何应用它）

|  |
| --- |
| java.util.Pattern  java.text.Normalizer  所有子类 java.text.Format  所有子类 javax.el.ELResolver |

**迭代器模式**

（可通过行为方法识别，从队列中顺序返回不同类型的实例）

|  |
| --- |
| 所有的实现java.util.Iterator（因此还有java.util.Scanner！）。  所有的实现 java.util.Enumeration |

**中介者模式**

（通过采用不同的抽象/接口类型（通常使用命令模式）实例的行为方法来识别给定实例）

|  |
| --- |
| java.util.Timer（所有scheduleXXX()方法）  java.util.concurrent.Executor#execute()  java.util.concurrent.ExecutorService（invokeXXX()和submit()方法）  java.util.concurrent.ScheduledExecutorService（所有scheduleXXX()方法）  java.lang.reflect.Method#invoke() |

**备忘录模式**

（可以通过内部改变整个实例的状态的行为方法来识别）

|  |
| --- |
| java.util.Date（setter方法这样做，Date内部由一个long值表示）  所有的实现 java.io.Serializable  所有的实现 javax.faces.component.StateHolder |

**观察者模式（或发布/订阅）**

（可以通过行为方法识别，根据自己的状态调用另一个抽象/接口类型的实例上的方法）

|  |
| --- |
| java.util.Observer/ java.util.Observable（很少在现实世界中使用）  所有实现java.util.EventListener（因此实际上各地的Swing）  javax.servlet.http.HttpSessionBindingListener  javax.servlet.http.HttpSessionAttributeListener  javax.faces.event.PhaseListener |

**状态模式**

（可以通过行为方法识别，根据可以从外部控制的实例的状态改变其行为）

|  |
| --- |
| javax.faces.lifecycle.LifeCycle#execute()（FacesServlet由此控制，行为取决于JSF生命周期的当前阶段（状态）） |

**策略**

（可以通过抽象/接口类型中的行为方法识别，该方法在已经作为方法参数传递到策略实现中的不同抽象/接口类型的实现中调用方法）

|  |
| --- |
| java.util.Comparator#compare()，由其他人执行Collections#sort()。  javax.servlet.http.HttpServlet，service()所有的doXXX()方法HttpServletRequest  HttpServletResponse实现者必须处理它们（而不是把它们保持为实例变量！）。  javax.servlet.Filter#doFilter() |

**模板方法**

（可以由已经具有抽象类型定义的“默认”行为的行为方法识别）

|  |
| --- |
| 所有非抽象方法java.io.InputStream，java.io.OutputStream，java.io.Reader和java.io.Writer。  所有非抽象方法java.util.AbstractList，java.util.AbstractSet和java.util.AbstractMap。  javax.servlet.http.HttpServlet，doXXX()默认情况下，所有方法都会向响应发送HTTP 405“方法不允许”错误。你可以自由地执行任何一个或任何它们。 |

**访问者**

（可以通过两种不同的抽象/接口类型识别，它们的方法定义为采用每个其他抽象/接口类型;实际上调用另一个抽象/接口类型的方法，另一个执行所需的策略）

|  |
| --- |
| javax.lang.model.element.AnnotationValue 和 AnnotationValueVisitor  javax.lang.model.element.Element 和 ElementVisitor  javax.lang.model.type.TypeMirror 和 TypeVisitor  java.nio.file.FileVisitor 和 SimpleFileVisitor |

**1.4 JDBC-桥接模式**

http://blog.csdn.net/mark2when/article/details/53895696

**1.5 mybatis设计模式**

http://blog.csdn.net/u012387062/article/details/54719114

1. **设计模式分类**

三大类：

* **创建型**模式(5)

单例、工厂方法、抽象工厂、建造者、原型

* 结构型模式(7)

装饰器、代理、适配器、外观、桥接、组合、享元。

* 行为型模式(11)

策略、观察者、模板方法、迭代子模式、责任链、命令、备忘录、状态、访问者、中介者、解释器。

其实还有两类：并发型模式和线程池模式。图片来整体描述一下：

****

1. **设计模式六大原则**

单一职责原则

里氏替换原则

依赖倒置原则

接口隔离原则

迪米特法则

开闭原则

* 1. **单一职责原则**

定义：不要存在多于一个导致类变更的原因。即一个类只负责一项职责。

问题由来：类T负责两个不同职责：职责P1，职责P2。当由于职责P1需求发生改变而需要修改类T时，有可能会导致原本运行正常的职责P2功能发生故障。

解决方案：遵循单一职责原则。分别建立两个类T1、T2，使T1完成职责P1功能，T2完成职责P2功能。这样，当修改类T1时，不会使职责P2发生故障风险；同理，当修改T2时，也不会使职责P1发生故障风险。

* 1. **依赖倒置原则**

定义：高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

|  |
| --- |
| 是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。 |

问题由来：类A直接依赖类B，假如要将类A改为依赖类C，则必须通过修改类A的代码来达成。这种场景下，类A一般是高层模块，负责复杂的业务逻辑；类B和类C是低层模块，负责基本的原子操作；假如修改类A，会给程序带来不必要的风险。

解决方案：将类A修改为依赖接口I，类B和类C各自实现接口I，类A通过接口I间接与类B或者类C发生联系，则会大大降低修改类A的几率。

* 1. **开闭原则**

定义：软件实体如类、模块和函数,应对扩展开放，对修改关闭。

问题由来：在软件的生命周期内，因为变化、升级和维护等原因需要对软件原有代码进行修改时，可能会给旧代码中引入错误，也可能会使我们不得不对整个功能进行重构，并且需要原有代码经过重新测试。

解决方案：程序需拓展时，不能去修改原有代码，实现一个热插拔效果。

一句话概括：为使程序扩展性好，易于维护和升级。需使用接口和抽象类

* 1. **里氏替换原则**(Liskov Substitution Principle LSP)

定义：所有引用基类的地方必须能透明地使用其子类的对象。

|  |
| --- |
| 里氏代换原则中说，任何基类可出现的地方，子类一定可出现。  LSP是继承复用的基石，只有当衍生类可替换掉基类，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。  LSP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。 |

问题由来：有一功能P1，由类A完成。现需要将功能P1进行扩展，扩展后的功能为P，其中P由原有功能P1与新功能P2组成。新功能P由类A的子类B来完成，则子类B在完成新功能P2的同时，有可能会导致原有功能P1发生故障。

解决方案：当使用继承时，遵循里氏替换原则。类B继承类A时，除添加新的方法完成新增功能P2外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法。

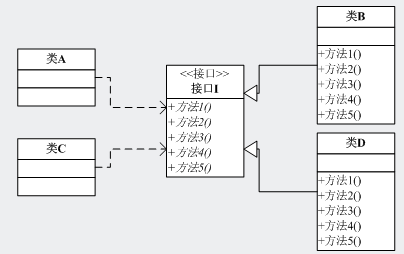
* 1. **接口隔离原则**

定义：客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。

|  |
| --- |
| 意思是：使用多个隔离的接口，比使用单个接口要好。  还是一个降低类之间的耦合度的意思  从这儿看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。 |

问题由来：类A通过接口I依赖类B，类C通过接口I依赖类D，如果接口I对于类A和类B来说不是最小接口，则类B和类D必须去实现他们不需要的方法。

解决方案：将臃肿的接口I拆分为独立的几个接口，类A和类C分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。



* 1. **迪米特法则（最少知道原则）**

定义：一个对象应该对其他对象保持最少的了解。

|  |
| --- |
| 叫最少知道原则，就是说：一个实体应尽量少的与其他实体间发生相互作用，使得系统功能模块相对独立。 |

问题由来：类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

解决方案：尽量降低类与类之间的耦合。

1. **单例模式**

**5.1 立即加载/"饿汉模式"**

|  |
| --- |
| package designmode.\_01singleton.\_01hungrytype;  /\*\*  \* 立即加载/饿汉式:  \* 使用类时已将对象创建完毕，即调用方法前，实例已被创建  \*/  public class Myobject {  private static Myobject myobject = new Myobject();    private Myobject() {  }    //此版本缺点是不能有其它实例变量，若有，因getInstence未同步，可能出现非线程安全问题  public static Myobject getInstence() {  return myobject;  }  } |

可通过打印出对象的hashcode证实是同一对象。

**5.2 延迟加载/"懒汉模式"**

在调用get()方法时实例才被创建

* 首先，测试代码:

|  |
| --- |
| package designmode.\_01singleton.\_01hungrytype;  import designmode.\_01singleton.\_02lazytype.Myobject;  public class Mythread extends Thread{  @Override  public void run() {  System.out.println(Myobject.getInstence().hashCode());  }  } |

|  |
| --- |
| package designmode.\_01singleton.\_01hungrytype;  public class Test {  public static void main(String[] args) {  Mythread t1 = new Mythread();  Mythread t2 = new Mythread();  Mythread t3 = new Mythread();  t1.start();  t2.start();  t3.start();  }  } |

* Synchronized保证线程安全

|  |
| --- |
| package designmode.\_01singleton.\_02lazytype;  public class Myobject {  private static Myobject myobject;    private Myobject() {  }    synchronized public static Myobject getInstence() {  if(myobject!=null) {  } else {  myobject = new Myobject();  }  return myobject;  }  } |

* **同步代码块**

|  |
| --- |
| package designmode.\_01singleton.\_02lazytype;  public class Myobject {  private static Myobject myobject;    private Myobject() {  }    public static Myobject getInstence() {  //等同于synchronized public static Myobject getInstence()  //全部代码被上锁，效率太低  synchronized (Myobject.class) {  if(myobject!=null) {  } else {  myobject = new Myobject();  }  }    return myobject;  }  } |

* **DCL双检查锁机制(最优)**

|  |
| --- |
| package designmode.\_01singleton.\_02lazytype;  public class Myobject {  private volatile static Myobject myobject;    private Myobject() {  }    public static Myobject getInstence() {  if(myobject==null) {  synchronized (Myobject.class) {  if(myobject==null) {  myobject = new Myobject();  }  }  }  return myobject;  }  } |

**5.3 使用静态内置类实现单例模式**

|  |
| --- |
| public class MyobjectInnerclass {  private static class MyobjectHandler{  private static MyobjectInnerclass myobject = new MyobjectInnerclass();  }    private MyobjectInnerclass() {  }    public static MyobjectInnerclass getInstence() {  return MyobjectHandler.myobject;  }  } |

**5.4 【难】序列化与反序列化的单例模式实现**

|  |
| --- |
| package designmode.\_01singleton.\_04serialize;  import java.io.ObjectStreamException;  import java.io.Serializable;  public class Myobject implements Serializable{  private static final long serialVersionUID = 1L;  private static class MyobjectHandler {  private static final Myobject myobject= new Myobject();  }    private Myobject() {  }    public static Myobject getInstence() {  return MyobjectHandler.myobject;  }    protected Object readResolve() throws ObjectStreamException{  System.out.println("调用了readResolve方法!");  return MyobjectHandler.myobject;  }  } |

|  |
| --- |
| package designmode.\_01singleton.\_04serialize;  import java.io.File;  import java.io.FileInputStream;  import java.io.FileNotFoundException;  import java.io.FileOutputStream;  import java.io.IOException;  import java.io.ObjectInputStream;  import java.io.ObjectOutputStream;  public class SaveAndRead {  public static void main(String[] args) {  Myobject myobject = Myobject.getInstence();  try {  FileOutputStream fileOutputStream = new FileOutputStream(new File("a.txt"));  ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);  objectOutputStream.writeObject(myobject);  objectOutputStream.close();  fileOutputStream.close();  System.out.println("存储:" + myobject.hashCode());  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }    try {  FileInputStream fileInputStream = new FileInputStream(new File("a.txt"));  ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);  Myobject myobject2 = (Myobject)objectInputStream.readObject();  objectInputStream.close();  fileInputStream.close();  System.out.println("读取:" + myobject2.hashCode());  } catch (IOException | ClassNotFoundException e) {  e.printStackTrace();  } finally {  }  }  } |

**结果:**

|  |
| --- |
| 存储:5699121  调用了readResolve方法!  读取:5699121 |

**5.5 使用static代码块实现单例模式**

静态代码块在使用类时就已执行。

|  |
| --- |
| package designmode.\_01singleton.\_05staticcodearea;  public class Myobject {  private static Myobject instence = null;    private Myobject() {  }    static {  instence = new Myobject();  }    public static Myobject getInstence() {  return instence;  }  } |

**5.6 【补】使用enum枚举数据类型实现单例模式**

**5.7 【补】完善使用enum枚举实现单例模式**

1. **【F】工厂模式**

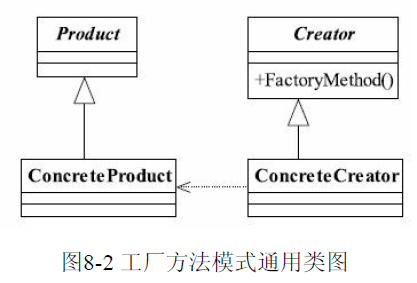
**8.2 @【F】工厂方法模式的定义**

定义为：

Define an interface for creating an object,but let subclasses decide which class to

instantiate.Factory Method lets a class defer instantiation to subclasses.（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。工厂方法使一个类的实例化延迟到其子类。）

* 图8-2 工厂方法模式通用类图



抽象产品类Product:定义产品共性，实现对事物最抽象的定义；

Creator:抽象创建类，抽象工厂，具体如何创建产品类是由具体的实现工厂

ConcreteCreator完成的。

* 代码清单8-8 抽象产品类

|  |
| --- |
| public abstract class Product {  //产品类的公共方法  public void method1(){  //业务逻辑处理  }  //抽象方法  public abstract void method2();  } |

具体产品类可有多个，都继承于抽象产品类

* 代码清单8-9 具体产品类

|  |
| --- |
| public class ConcreteProduct1 extends Product {  public void method2() {  //业务逻辑处理  }  }  public class ConcreteProduct2 extends Product {  public void method2() {  //业务逻辑处理  }  } |

抽象工厂类负责定义产品对象的产生。

* 代码清单8-10 抽象工厂类

|  |
| --- |
| public abstract class Creator {  /\*  \* 创建一个产品对象，其输入参数类型可以自行设置  \* 通常为String、Enum、Class等，当然也可以为空  \*/  public abstract <T extends Product> T createProduct(Class<T> c);  } |

具体如何产生一个产品对象，由具体工厂类实现。

* 代码清单8-11 具体工厂类

|  |
| --- |
| public class ConcreteCreator extends Creator {  public <T extends Product> T createProduct(Class<T> c){  Product product=null;  try {  product = (Product)Class.forName(c.getName()).newInstance();  } catch (Exception e) {  //异常处理  }  return (T)product;  }  } |

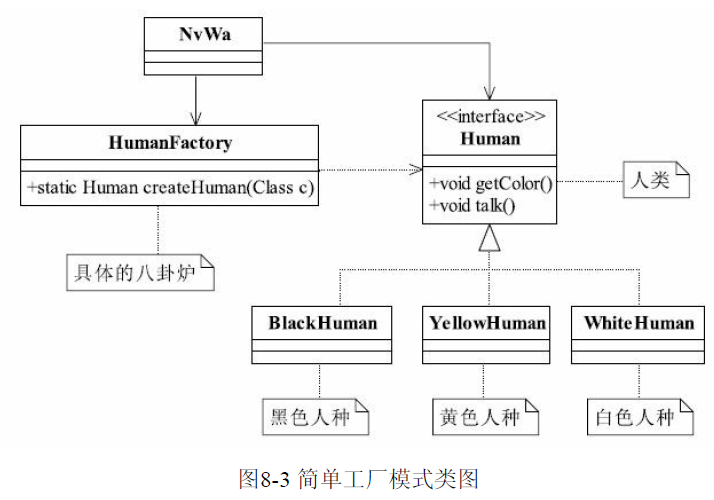
场景类的调用方法。

* 代码清单8-12 场景类

|  |
| --- |
| public class Client {  public static void main(String[] args) {  Creator creator = new ConcreteCreator();  Product product = creator.createProduct(ConcreteProduct1.class);  /\*  \* 继续业务处理  \*/  }  } |

**8.4.1. @@缩小为简单(**静态**)工厂模式**

* 【例子1】  
  考虑：一个模块仅需一个工厂类，没必要把它产生出来，用静态的方法就可，把上例中的AbstarctHumanFactory修改一下，



类图中去掉了AbstractHumanFactory抽象类，同时把createHuman方法设置为静态类型，简化了类的创建过程，变更的源码仅仅是HumanFactory和NvWa类，

* 代码清单8-13 简单工厂模式中的工厂类HumanFactory

|  |
| --- |
| public class HumanFactory {  public static <T extends Human> T createHuman(Class<T> c){  //定义一个生产出的人种  Human human=null;  try {  //产生一个人种  human = (Human)Class.forName(c.getName()).newInstance();  } catch (Exception e) {  System.out.println("人种生成错误！");  }  return (T)human;  }  } |

HumanFactory类仅有两个地方变化：去掉继承抽象类，并在createHuman前增加static关键字；

工厂类发生变化，也同时引起了调用者NvWa的变化。

代码清单8-14 简单工厂模式中的场景类

|  |
| --- |
| public class NvWa {  public static void main(String[] args) {  //女娲第一次造人，火候不足，于是白色人种产生了  System.out.println("--造出的第一批人是白色人种--");  Human whiteHuman = HumanFactory.createHuman(WhiteHuman.class);  whiteHuman.getColor();  whiteHuman.talk();  //女娲第二次造人，火候过足，于是黑色人种产生了  System.out.println("\n--造出的第二批人是黑色人种--");  Human blackHuman = HumanFactory.createHuman(BlackHuman.class);  blackHuman.getColor();  blackHuman.talk();  //第三次造人，火候刚刚好，于是黄色人种产生了  System.out.println("\n--造出的第三批人是黄色人种--");  Human yellowHuman = HumanFactory.createHuman(YellowHuman.class);  yellowHuman.getColor();  yellowHuman.talk();  }  } |

工厂方法模式的弱化，称为简单(静态)工厂模式。在实际项目中用的较多，缺点是工厂类的扩展较困难，不符合开闭原则，但它仍然是一个非常实用的设计模式。

* 【例子2】

例如:汽车销售商场

该模式将创建对象的过程放在了一个静态方法中来实现.

在实际编程中,如果需要大量的创建对象,该模式是比较理想的.

|  |
| --- |
| **public** **class** Demo1 {  **public** **static** **void** main(String[] args) {  System.*out*.println("买宝马");  Car bmw = *CarFactory*("BMW");  bmw.run();  System.*out*.println("买大奔");  Car benz = *CarFactory*("Benz");  benz.run();  }  **public** **static** Car CarFactory(String carName) {  **if** ("BMW".equals(carName)) {  **return** **new** BMW();  } **else** **if** ("Benz".equals(carName)) {  **return** **new** Benz();  } **else** {  **throw** **new** RuntimeException("车型有误");  }  }  }  **abstract** **class** Car {  **public** **abstract** **void** run();  }  **class** BMW **extends** Car {  @Override  **public** **void** run() {  System.*out*.println("BMW跑跑");  }  }  **class** Benz **extends** Car {  @Override  **public** **void** run() {  System.*out*.println("Benz跑跑");  }  } |

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~(后面没仔细看)

1.工厂设计模式：专门用于产生对象的方法。

|  |
| --- |
| package cn.itcast.factory;  import java.io.BufferedReader;  import java.io.File;  import java.io.FileNotFoundException;  import java.io.FileReader;  /\*  需求： 定义一个工厂方法可以产生任何类型的对象，产生的对象是根据配置文件而定。  \*/  import java.io.IOException;  import java.lang.reflect.Constructor;  import java.lang.reflect.Field;  import cn.itcast.reflect.Person;  public class Demo1 {  public static void main(String[] args) throws Exception {  Person p =(Person) newInstance();  System.out.println("人对象："+ p);  }  public static Object newInstance() throws Exception{  BufferedReader bufferedReader = new BufferedReader(new FileReader("obj.txt"));  //读取到完整的类名  String className = bufferedReader.readLine();  //根据类名获取Class对象  Class clazz = Class.forName(className);  //获取无参的构造函数  Constructor constructor = clazz.getConstructor(null);  //创建对象。  Object o = constructor.newInstance(null);    //读取配置文件，然后对象的属性数据 封装到对象中。  String line = null;  while((line = bufferedReader.readLine())!=null){  String[] datas = line.split("=");  Field field = clazz.getDeclaredField(datas[0]);  field.setAccessible(true);  if(field.getType()==int.class){  field.set(o, Integer.parseInt(datas[1]));  }else{  field.set(o, datas[1]);  }  }  return o;  }  } |

3.模拟spring工厂：

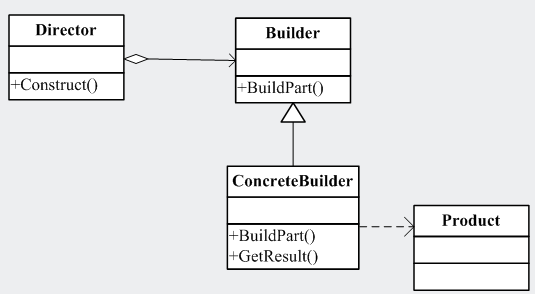
|  |
| --- |
| **import** java.io.BufferedReader;  **import** java.io.FileReader;  **import** java.io.IOException;  **import** java.lang.reflect.Constructor;  **import** java.lang.reflect.Field;  **class** Student{  **private** **int** id;  **private** String name;  **public** Student(**int** id , String name){  **this**.id = id;  **this**.name = name;  }  **public** Student(){  }  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** **this**.id +"-"+**this**.name;  }  }  **class** Person{  **private** **int** age;  **public** Person(){  }  @Override  **public** String toString() {  **return** **this**.age+"";  }  }  **public** **class** Demo1 {  **public** **static** **void** main(String[] args) **throws** Exception {  Object o = *getInstance*();  System.*out*.println(o);  }    **public** **static** Object getInstance() **throws** Exception{  FileReader fileReader = **new** FileReader("src/info.txt");  BufferedReader bufferedReader = **new** BufferedReader(fileReader);  String line = bufferedReader.readLine();  Class clazz = Class.*forName*(line);  Constructor c = clazz.getConstructor(**null**);  Object c1 = c.newInstance(**null**);  **while**((line=bufferedReader.readLine())!=**null**){  String[] datas = line.split("=");  Field f = clazz.getDeclaredField(datas[0]);  f.setAccessible(**true**);  **if**(f.getType()==**int**.**class**){  f.set(c1, Integer.*parseInt*(datas[1]));  }**else**{  //f.setAccessible(true);  f.set(c1,datas[1]);  }  }  **return** c1;  }  } |

1. **建造者模式**

**7.1 简介**

* 将复杂对象的构建与它的表示分离，使得同样的构建过程可创建不同的表示。
* 角色：

|  |
| --- |
| * **Product**：要创建的复杂对象。 * **builder**：给出抽象接口，以规范产品对象的各个组成成分的建造。该接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。 * **ConcreteBuilder**：实现Builder接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。 在建造过程完成后，提供产品的实例。 * **Director**：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。 |



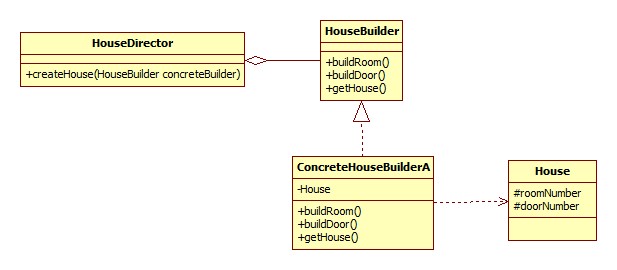
* 使用的好处：

|  |
| --- |
| 1.可使客户端不必知道产品内部组成细节。  2.具体的建造者类间相互独立，对系统的扩展非常有利。  3.由于具体建造者独立，因此可对建造过程逐步细化，而不影响其他的模块。 |

* 使用场合：

|  |
| --- |
| 1. 创建一些复杂对象时，对象的内部组成构件间的建造顺序稳定，但对象的内部组成构件面临着复杂的变化。   2.要创建的复杂对象的算法，独立于该对象的组成部分，也独立于组成部分的装配方法时。 |

**7.2 例子**



1. House.java

|  |
| --- |
| package designmode.\_08builder;  /\*\*  \* 最终需生成的产品:房屋  \*/  public class House {  int roomNumber;  int doorNumber;    public House() {  roomNumber = 0;  doorNumber = 0;  }  public int getRoomNumber() {  return roomNumber;  }  public int getDoorNumber() {  return doorNumber;  }  } |

1. HouseBuilder.java

|  |
| --- |
| package designmode.\_08builder;  /\*\*  \* 房屋的真正构建者:生成器的接口  \*/  public interface HouseBuilder {  public void buildRoom(int roomNo);    public void buildDoor(int room1, int room2);    public House getHouse();  } |

3. ConcreteHouseBuilderA.java

|  |
| --- |
| package designmode.\_08builder;  /\*\*  \* 房屋的真正构建者:生成器的实现  \*/  public class ConcreteHouseBuilderA implements HouseBuilder{  private House house;    public ConcreteHouseBuilderA() {  house = new House();  }  @Override  public void buildRoom(int roomNo) {  house.roomNumber = house.roomNumber + roomNo;  }  @Override  public void buildDoor(int room1, int room2) {  house.doorNumber = house.doorNumber + room1 + room2;  }  @Override  public House getHouse() {  return house;  }  } |

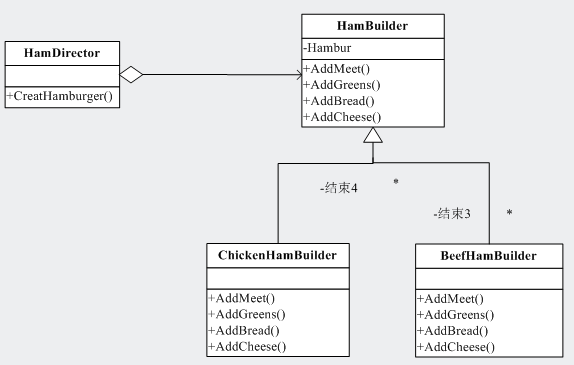
4. HouseDirector.java

|  |
| --- |
| package designmode.\_08builder;  /\*\*  \* Director:最终构建房屋的'表示者',对同一种构建采用不同的表示方式  \* 需给其提供'生成器'(HouseBuilder),然后由它来构建房屋  \*/  public class HouseDirector {  public static House createHouse(HouseBuilder concreteBuilder) {  concreteBuilder.buildRoom(1);  concreteBuilder.buildRoom(2);  concreteBuilder.buildDoor(1, 2);  concreteBuilder.buildDoor(2, 1);    return concreteBuilder.getHouse();  }  } |

5.测试

TestBuilder.java

|  |
| --- |
| package designmode.\_08builder;  /\*\*  \* 建造者模式：  \* 1.创建生成器对象  \* 2.表示者使用该生成器对象构建最终产品  \* 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创建不同的表示  \*/  public class TestBuilder {  public static void main(String[] args) {  ConcreteHouseBuilderA myHouseBuilder = new ConcreteHouseBuilderA();    //此处运用多态  House myHouse = HouseDirector.createHouse(myHouseBuilder);  System.out.println("My house has room:" + myHouse.getRoomNumber());  //My house has room:3  System.out.println("My house has door:" + myHouse.getDoorNumber());  //My house has door:6  }  } |



1. **【F】装饰模式**

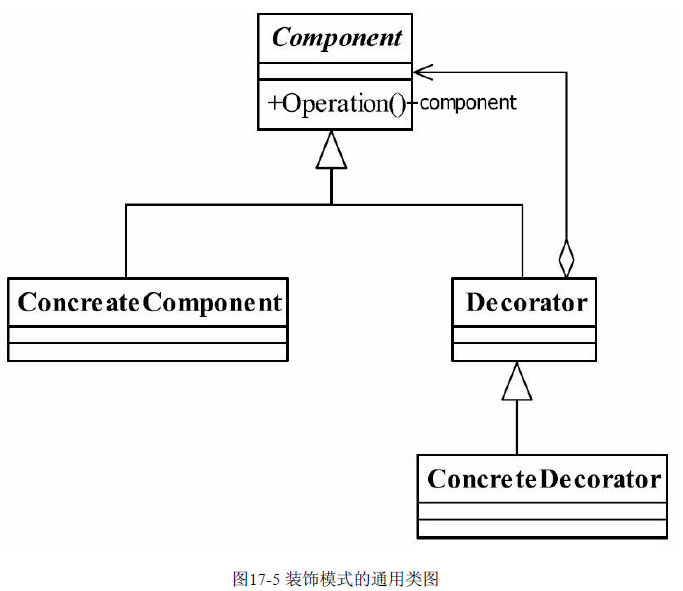
17.2 @装饰模式的定义

装饰模式定义：

Attach additionalres ponsibilities to an object dynamically keeping the same interface.Decorators provide a flexible alternative to subclassing for extending functionality.

（动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更灵活。）

图17-5 装饰模式的通用类图



在类图中，有四个角色需要说明：

|  |
| --- |
| ● Component抽象构件(必然有)  接口或抽象类，定义最基本、最核心、最原始的对象，如上面的成绩单。  ● ConcreteComponent 具体构件  最原始、最核心、最基本的接口或抽象类的实现，要装饰的就是它。  ● Decorator 装饰角色  抽象类.实现接口或者抽象方法，不一定有抽象方法，属性里必有一个private变量指向Component抽象构件。  ● 具体装饰角色  ConcreteDecoratorA和ConcreteDecoratorB是两个具体的装饰类，要把最核心的、最原始的、最基本的东西装饰成其他东西 |

* 代码清单17-10 抽象构件

|  |
| --- |
| public abstract class Component {  //抽象的方法  public abstract void operate();  } |

* 代码清单17-11 具体构件

|  |
| --- |
| public class ConcreteComponent extends Component {  //具体实现  @Override  public void operate() {  System.out.println("do Something");  }  } |

* 代码清单17-12 抽象装饰者

|  |
| --- |
| public abstract class Decorator extends Component {  private Component component = null;  //通过构造函数传递被修饰者  public Decorator(Component \_component){  this.component = \_component;  }  //委托给被修饰者执行  @Override  public void operate() {  this.component.operate();  }  } |

若只有一个装饰类，则可没有抽象装饰角色，直接实现具体的装饰角色即可。

* 代码清单17-13 具体的装饰类

|  |
| --- |
| public class ConcreteDecorator1 extends Decorator {  //定义被修饰者  public ConcreteDecorator1(Component \_component){  super(\_component);  }  //定义自己的修饰方法  private void method1(){  System.out.println("method1 修饰");  }  //重写父类的Operation方法  public void operate(){  this.method1();  super.operate();  }  }  } |

|  |
| --- |
| public class ConcreteDecorator2 extends Decorator {  //定义被修饰者  public ConcreteDecorator2(Component \_component){  super(\_component);  }  //定义自己的修饰方法  private void method2(){  System.out.println("method2修饰");  }  //重写父类的Operation方法  public void operate(){  super.operate();  this.method2();  } |

注意　原始方法和装饰方法的执行顺序在具体的装饰类是固定的，可通过方法重载实现多种执行顺序。

通过Client类模拟高层模块的耦合关系，看看装饰模式如何运行

* 代码清单17-14 场景类

|  |
| --- |
| public class Client {  public static void main(String[] args) {  Component component = new ConcreteComponent();  //第一次修饰  component = new ConcreteDecorator1(component);  //第二次修饰  component = new ConcreteDecorator2(component);  //修饰后运行  component.operate();  }  } |

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~(后面没仔细看)

**7.2 步骤：**

|  |
| --- |
| 1 让增强与被增强类有一个共同父类或父接口.  2. 在增强类的内部维护一个被增强类的引用。 |

好处：利用了多态达到了类与类间可互相装饰，较灵活。

缺点：代码结构不清晰，难以理解。

装饰模式(Decorator)，动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更加灵活。

装饰模式分为3个部分：

1. 抽象组件 -- 对应Coffee类

2. 具体组件 -- 对应具体的咖啡，如：Decaf，Espresso

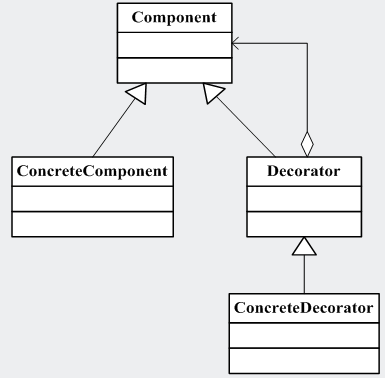
3. 装饰者 -- 对应调味品，如：Mocha，Whip

装饰模式有3个特点：

1. 具体组件和装饰者都继承自抽象组件(Decaf、Espresson、Mocha和Whip都继承自Coffee)，并且装饰者持有抽象组件的引用

2. 可以使用装饰者组合具体组件创造出新的类(Mocha组合Decaf创造出MochaDecaf)

3. 过程2可以重复，直到创造出需要的类



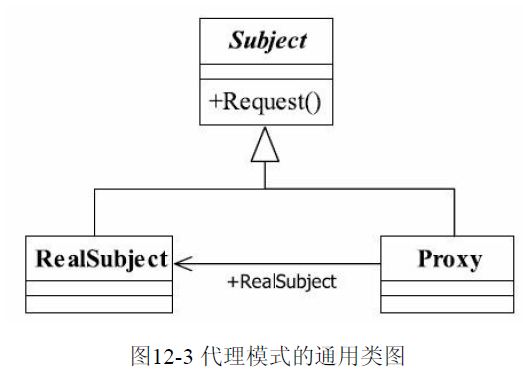
1. **【F】代理模式【详细见设计模式之禅】**

**12.2 @代理(委托)模式的定义**

定义：Provide a surrogate or placeholder for another object to control access to it.

（为其他对象提供一种代理以控制对这个对象的访问。）

* 许多其他模式，如状态、策略、访问者模式,本质上是在更特殊的场合采用了委托模式.
* 日常应用中，代理模式可提供非常好的访问控制。如Struts2的Form元素映射用了动态代理模式。



|  |
| --- |
| ● Subject 抽象主题  抽象类或接口  ● RealSubject 具体主题  被委托角色、被代理角色。业务逻辑的具体执行者。  ● Proxy 代理主题  委托类、代理类。  负责对真实角色的应用，把所有抽象主题类定义的方法限制委托给真实主题角色实现，且在真实主题角色处理完毕前后做预处理和善后处理工作。 |

* 代码清单12-6 抽象主题类

|  |
| --- |
| public interface Subject {  //定义一个方法  public void request();  } |

* 代码清单12-7 真实主题类

|  |
| --- |
| public class RealSubject implements Subject {//业务实现类  //实现方法  public void request() {  //业务逻辑处理  }  } |

* 代码清单12-8 代理类(核心)

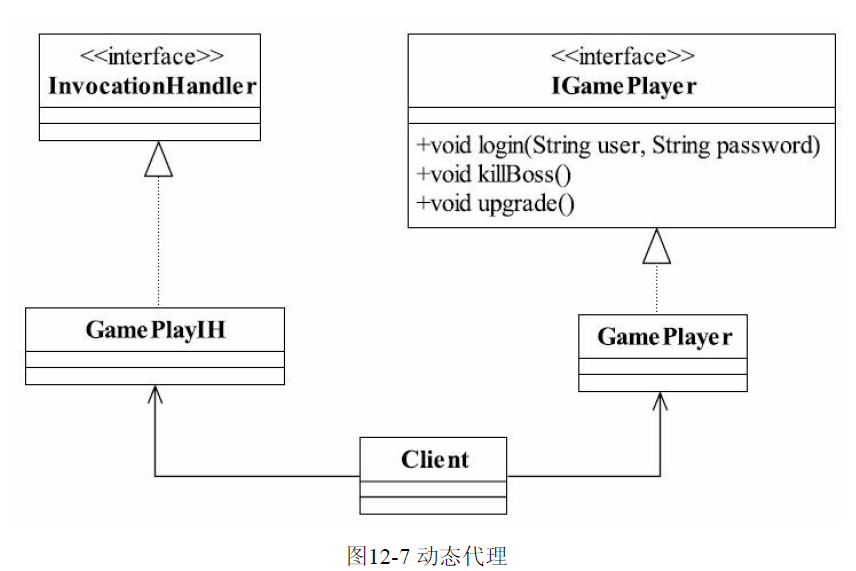
|  |
| --- |
| public class Proxy implements Subject {  //要代理哪个实现类  private Subject subject = null;  //默认被代理者  public Proxy(){  ~~this.subject = new Proxy();~~  }  //通过构造函数传递代理者  public Proxy(Subject \_subject){  this.subject = \_subject;  }  //实现接口中定义的方法  public void request() {  this.before();  this.subject.request();  this.after();  }  //预处理  private void before(){  //do something  }  //善后处理  private void after(){  //do something  }  } |

一个代理类可代理多个被代理者，具体代理哪个真实主题角色，由场景类决定。通常一个接口只需一个代理类就可，具体代理哪个实现类,在代理类的构造函数中传递被代理者的实例

**12.4.4 @动态代理**

实现阶段不用关心代理谁，运行阶段才指定代理哪个对象。相对来说，自己写代理类的方式就是静态代理。

类图修改一下以实现动态代理，如图12-7所示。



增加了InvocationHandler接口和GamePlayIH类，作用是产生一个对象的代理对象InvocationHandler是JDK提供的动态代理接口，对被代理类的方法进行代理。

请参考代码清单12-1和代码清单12-2所示。

* 代码清单12-21 动态代理类

|  |
| --- |
| public class GamePlayIH implements InvocationHandler {  //被代理者  Class cls =null;  //被代理的实例  Object obj = null;  //我要代理谁  public GamePlayIH(Object \_obj){  this.obj = \_obj;  }  //调用被代理的方法  public Object invoke(Object proxy, Method method, Object[] args)  throws Throwable {  Object result = method.invoke(this.obj, args);  return result;  }  } |

InvocationHandler接口，动态代理是根据被代理的接口生成所有的方法，也就是说给定一个接口，动态代理会宣称“我已经实现该接口下的所有方法了”，那各位读者想想看，动态代理怎么才能实现被代理接口中的方法呢？默认情况下所有的方法返回值都是空的，是的，代理已经实现它了，但是没有任何的逻辑含义，那怎么办？好办，通过InvocationHandler接口，所有方法都由该Handler来进行处理，即所有被代理的方法都由InvocationHandler接管实际的处理任务。

* 代码清单12-22 动态代理的场景类

|  |
| --- |
| public class Client {  public static void main(String[] args) throws Throwable {  //定义一个痴迷的玩家  IGamePlayer player = new GamePlayer("张三");  //定义一个handler  InvocationHandler handler = new GamePlayIH(player);  //开始打游戏，记下时间戳  System.out.println("开始时间是：2009-8-25 10:45");  //获得类的class loader  ClassLoader cl = player.getClass().getClassLoader();  Class< GamePlayer >[] interfaces = subject.getClass().getInterfaces()  //动态产生一个代理者  IGamePlayer proxy = (IGamePlayer)Proxy.newProxyInstance(cl, interfaces ,handler)  {//登录  proxy.login("zhangSan", "password");  //开始杀怪  proxy.killBoss();  //升级  proxy.upgrade();  //记录结束游戏时间  System.out.println("结束时间是：2009-8-26 03:40");  }  } |

|  |
| --- |
| public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) throws IllegalArgumentException |

其运行结果如下：

|  |
| --- |
| 开始时间是：2009-8-25 10:45  登录名为zhangSan 的用户 张三登录成功！  张三在打怪！  张三 又升了一级！  结束时间是：2009-8-26 03:40 |

还是让代练者帮我们打游戏，但我们既没有创建代理类，也没实现IGamePlayer接口，这就是动态代理。别急，动态代理可不仅就这么多内容，还有更重要的，如果想让游戏登录后发一个信息给我们，防止账号被人盗用嘛，该怎么处理？直接修改被代理类GamePlayer？这不是一个好办法，好办法如代码清单12-23所示。

* 代码清单12-23 修正后的动态代理

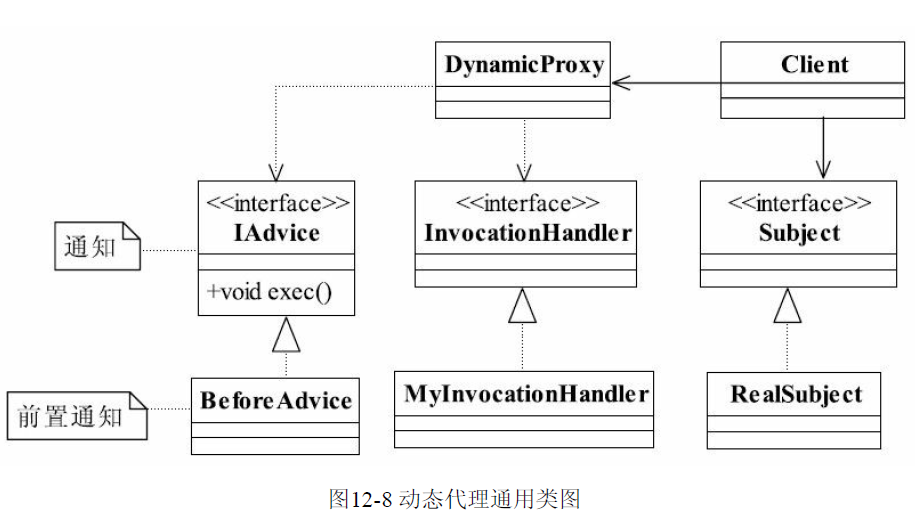
|  |
| --- |
| public class GamePlayIH implements InvocationHandler {  //被代理者  Class cls =null;  //被代理的实例  Object obj = null;  //我要代理谁  public GamePlayIH(Object \_obj){  this.obj = \_obj;  }  //调用被代理的方法  public Object invoke(Object proxy, Method method, Object[] args)  throws Throwable {  Object result = method.invoke(this.obj, args);  //如果是登录方法，则发送信息  if(method.getName().equalsIgnoreCase("login")){  System.out.println("有人在用我的账号登录！");  }  return result;  }  } |

只要在代理中增加一个判断就可决定是否要发送信息，运行结果如下：

|  |
| --- |
| 开始时间是：2009-8-25 10:45  登录名为zhangSan的用户　张三登录成功！  有人在用我的账号登录！  张三在打怪！  张三 又升了一级！  结束时间是：2009-8-26 03:40 |

太棒了！有人用我的账号就发送一个信息，然后看看自己的账号是不是被人盗了，非常好的方法，这就是AOP编程。AOP编程没用什么新技术，但对我们的设计、编码有非常大的影响，对于日志、事务、权限等都可以在系统设计阶段不用考虑，而在设计后通过AOP的方式切过去。

通用动态代理模型，类图如图12-8所示。



|  |
| --- |
| 两条独立发展的线路:   * 动态代理实现代理的职责 * 业务逻辑Subject实现相关的逻辑功能，两者间没有必然的耦合关系。   通知Advice从另一个切面切入，最终在高层模块也就是Client进行耦合，完成逻辑的封装任务。 |

* 代码清单12-24 抽象主题

|  |
| --- |
| public interface Subject {  //业务操作  public void doSomething(String str);  } |

代码清单12-25 真实主题

|  |
| --- |
| public class RealSubject implements Subject {  //业务操作  public void doSomething(String str) {  System.out.println("do something!---->" + str);  }  } |

代码清单12-26 动态代理的Handler类 重点

|  |
| --- |
| public class MyInvocationHandler implements InvocationHandler {  //被代理的对象  private Object target = null;  //通过构造函数传递一个对象  public MyInvocationHandler(Object \_obj){  this.target = \_obj;  }  //代理方法  public Object invoke(Object proxy, Method method, Object[] args)  throws Throwable {  //执行被代理的方法  return method.invoke(this.target, args);  }  } |

* 代码清单12-27 动态代理类(JDK)

|  |
| --- |
| public class DynamicProxy<T> {  public static <T> T newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) {  //寻找JoinPoint连接点，AOP框架使用元数据定义  if(true){  //执行一个前置通知  (new BeforeAdvice()).exec();  }  //执行目标，并返回结果  return (T)Proxy.newProxyInstance(loader,interfaces, h);  }  } |

* 通知Advice，要切入的类

代码清单12-28 通知接口及实现

|  |
| --- |
| public interface IAdvice {  //通知只有一个方法，执行即可  public void exec();  } |

|  |
| --- |
| public class BeforeAdvice implements IAdvice{  public void exec(){  System.out.println("我是前置通知，我被执行了！");  }  } |

代码清单12-29 动态代理的场景类(调用)

|  |
| --- |
| public class Client {  public static void main(String[] args) {  //定义一个主题  Subject subject = new RealSubject();  //定义一个Handler  InvocationHandler handler = new MyInvocationHandler(subject);  //定义主题的代理  Subject proxy = DynamicProxy.newProxyInstance(subject.getClass().  getClassLoader(), subject.getClass().getInterfaces(),handler);  //代理的行为  proxy.doSomething("Finish");  }  } |

运行结果如下所示：

|  |
| --- |
| 我是前置通知，我被执行了！  do something!---->Finish |

DynamicProxy类中方法：

|  |
| --- |
| this.obj=Proxy.newProxyInstance(c.getClassLoader(),c.getInterfaces(),new MyInvocationHandler(\_obj)); |

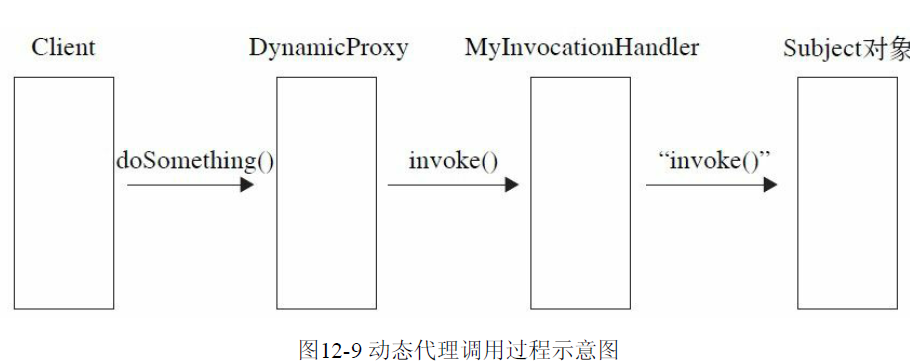
重新生成了一个对象,因为要使用代理

c.getInterfaces()这句话，是说查找到该类的所有接口，然后实现接口的所有方法。

当然了，方法都是空的，由谁具体负责接管呢？是new MyInvocationHandler(\_Obj)这个对象。

由InvocationHandler实现类实现所有方法，由其invoke方法接管所有方法的实现

* 图12-9 动态代理调用过程示意图



优化:DynamicProxy类，是一个通用类，不具有业务意义，如再产生一个实现类

* 代码清单12-30 具体业务的动态代理(main方法中代码移至此处)

|  |
| --- |
| public class SubjectDynamicProxy extends DynamicProxy{  public static <T> T newProxyInstance(Subject subject){  //获得ClassLoader  ClassLoader loader = subject.getClass().getClassLoader();  //获得接口数组  Class<?>[] classes = subject.getClass().getInterfaces();  //获得handler  InvocationHandler handler = new MyInvocationHandler(subject);  return newProxyInstance(loader, classes, handler);  }  } |

扩展后，高层模块对代理的访问会更加简单。

* 代码清单12-31 场景类

|  |
| --- |
| public class Client {  public static void main(String[] args) {  //定义一个主题  Subject subject = new RealSubject();  //定义主题的代理  Subject proxy = SubjectDynamicProxy.newProxyInstance(subject);  //代理的行为  proxy.doSomething("Finish");  }  } |

与静态代理还有什么区别？都是需要实现一个代理类，有区别，注意看父类，动态代理的主要意图就是解决我们常说的“审计”问题，也就是横切面编程，在不改变已有代码结构的情况下增强或控制对象的行为。

注意要实现动态代理的首要条件是：被代理类必须实现一个接口

CGLIB可实现不需要接口也可以实现动态代理的方式。

1. **桥接模式**

桥接模式(Bridge)，将抽象部分与它的实现部分分离，使它们都可以独立地变化

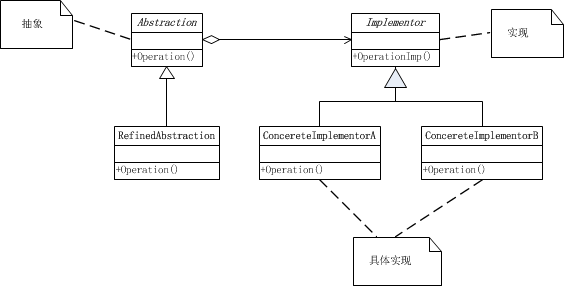
4个角色：

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

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

3. 实现化角色（Implementor）：这个角色给出实现化角色的接口，但不给出具体的实现。

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



优点：

分离抽象接口及其实现部分。提高了比继承更好的解决方案。

桥接模式提高了系统的可扩充性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统。

实现细节对客户透明，可以对用

缺点：

桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程

桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围具有一定的局限性。

使用桥接模式的场合：

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继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。

对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。

一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。

1. **[补]@策略模式**

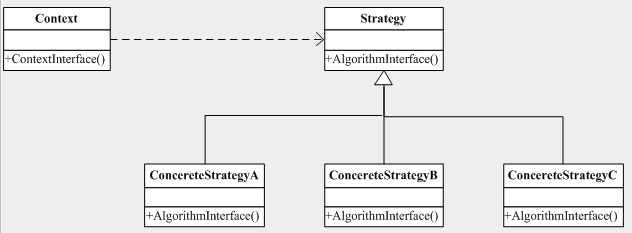
它定义了算法家族，分别封装起来，让它们之间可以互相替换，此模式算法的变化，不会影响到使用算法的客户。

策略模式通常包括下面几个角色：

1. Strategy：抽象策略类，通常由一个接口或者抽象类实现。

2. ConcereteStrategy：具体策略角色，包装了相关的算法和行为。

3. Context：环境角色，持有一个策略类的引用，最终给客户端调用。



**2.3 使用场景**

策略模式的好处：

相关算法系列 Strategy类层次为Context定义了一系列的可供重用的算法或行为。 继承有助于析取出这些算法中的公共功能。

提供了可以替换继承关系的办法

可以避免使用多重条件转移语句

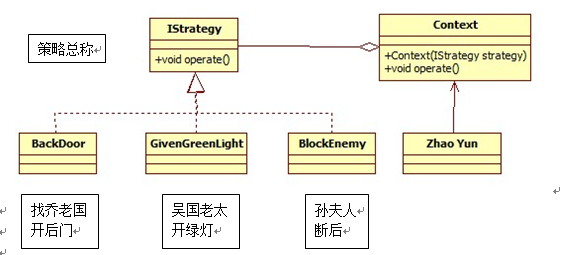
使用策略模式的场合：

多个类只区别在表现行为不同，可以使用Strategy模式，在运行时动态选择具 体要执行的行为。

需要在不同情况下使用不同的策略(算法)，或者策略还可能在未来用其它方式来实现。

对客户隐藏具体策略(算法)的实现细节，彼此完全独立。

**7.2 例子**



1. IStrategy.java

|  |
| --- |
| package designmode.\_09strategy;  /\*\*  \* 妙计接口  \*/  public interface IStrategy {  //每个锦囊妙计都是一个可执行的算法  public void operate();  } |

2.

* **BackDoor.java**

|  |
| --- |
| package designmode.\_09strategy;  public class BackDoor implements IStrategy{  @Override  public void operate() {  System.out.println("找乔老国帮忙，让吴国老太给孙权施加压力");  }  } |

* **GivenGreenLight.java**

|  |
| --- |
| package designmode.\_09strategy;  public class GivenGreenLight implements IStrategy{  @Override  public void operate() {  System.out.println("求吴国老太开绿灯，放行！");  }  } |

* **BlockEnemy.java**

|  |
| --- |
| package designmode.\_09strategy;  public class BlockEnemy implements IStrategy{  @Override  public void operate() {  System.out.println("孙夫人断后，挡住追兵");  }  } |

* **Context.java**

|  |
| --- |
| package designmode.\_09strategy;  /\*\*  \* 封装类(锦囊):承装三个策略，方便赵云使用  \* 通过构造函数把策略传递进来，然后用operate()方法执行相关策略方法  \*/  public class Context {  private IStrategy strategy;    //构造函数:你要使用哪个妙计  public Context(IStrategy strategy) {  this.strategy = strategy;  }    //使用计谋了，看我出招了  public void operate() {  this.strategy.operate();  }  } |

* **ZhaoYun.java**

|  |
| --- |
| package designmode.\_09strategy;  /\*\*  \* 使用计谋  \*/  public class ZhaoYun {  //赵云出场了，根据诸葛亮的交代，一次拆开妙计  public static void main(String[] args) {  Context context;    System.out.println("--刚到吴国时拆第一个--");  context = new Context(new BackDoor());  context.operate();  System.out.println("\n");    System.out.println("--刘备乐不思蜀了,拆第二个--");  context = new Context(new GivenGreenLight());  context.operate();//执行了第二个锦囊  System.out.println("\n");    System.out.println("---孙权的小兵追来了，咋办？拆第三个---");    context = new Context(new BlockEnemy());  context.operate();  System.out.println("\n");  }  } |

1. **【F】@观察者模式**

**7.1 概述**

当一个事物发生了指定动作，要通知另外一个事物做出相应的处理。

* 有时又被称为：

|  |
| --- |
| **发布-订阅**<Publish/Subscribe>模式、  **模型-视图**<Model/View>模式、  源-收听者<Source/Listener>模式  或从属者<Dependents>模式） |

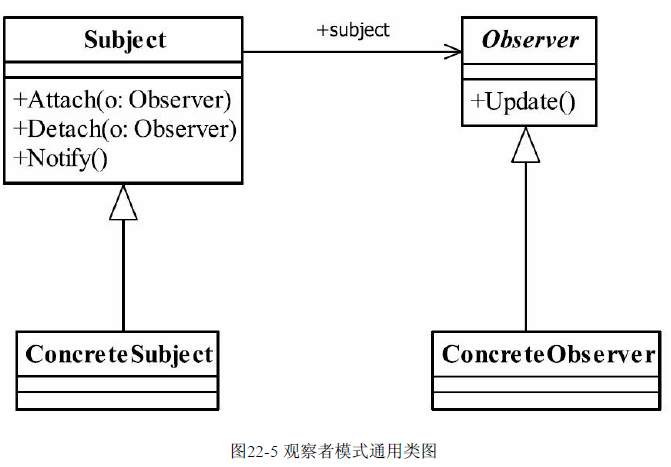
4. http://www.cnblogs.com/java-my-life/archive/2012/05/16/2502279.html

22.2 @ 定义

也叫发布订阅模式（Publish/subscribe）,是一个在项目中常用的模式，定义如下：

Define a one-to-many dependency between objects so that when one object changes state,all itsdependents are notified and updated automatically.（定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新）

观察者模式的通用类图，如图22-5所示。



解释一下几个角色名称：

● Subject被观察者

定义被观察者须实现的职责，须能动态增加、取消观察者。一般是抽象类或实现类，仅完成作为被观察者必须实现的职责：管理观察者并通知观察者。

● Observer观察者

观察者接收消息后，即进行update（更新方法）操作，对接收到的信息进行处理。

● ConcreteSubject具体的被观察者

定义被观察者自己的业务逻辑，同时定义对哪些事件进行通知。

● ConcreteObserver具体的观察者

每个观察者接收到消息后的都有自己的处理逻辑。

**通用代码**

* 被观察者

|  |
| --- |
| public abstract class Subject {  //定义一个观察者数组  private Vector<Observer> obsVector = new Vector<Observer>();  //增加一个观察者  public void addObserver(Observer o){  this.obsVector.add(o);  }  //删除一个观察者  public void delObserver(Observer o){  this.obsVector.remove(o);  }  //通知所有观察者  public void notifyObservers(){  for(Observer o:this.obsVector){  o.update();  }  }  } |

被观察者职责简单，就是定义谁能够观察，谁不能观察

用ArrayList和Vector没太大差别：

ArrayList是线程异步，不安全；

Vector是线程同步，安全。

* 具体被观察者

|  |
| --- |
| public class ConcreteSubject extends Subject {  //具体的业务  public void doSomething(){  //do something  super.notifyObservers();  }  } |

现在是一个纯净的观察者，具体项目中该类有很多变种，在22.4一节中介绍。

* **观察者**

|  |
| --- |
| public interface Observer {  //更新方法  public void update();  }  一般是接口，每一个实现该接口的实现类都是具体观察者 |

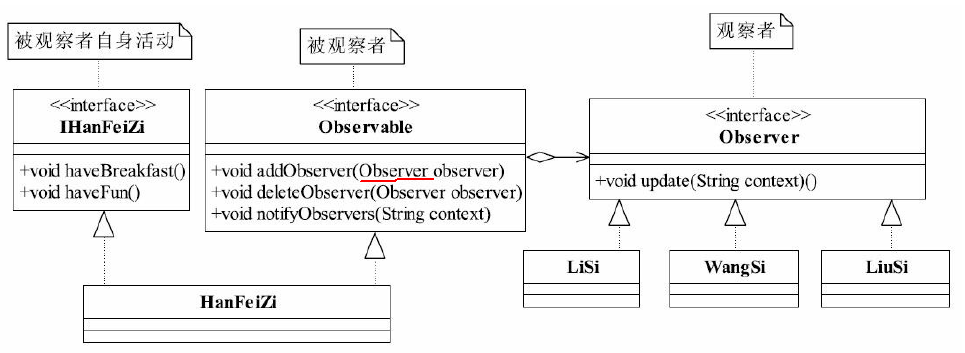
* **具体观察者**

|  |
| --- |
| public class ConcreteObserver implements Observer {  //实现更新方法  public void update() {  System.out.println("接收到信息，并进行处理！");  }  } |

* **场景类**

|  |
| --- |
| public class Client {  public static void main(String[] args) {  //创建一个被观察者  ConcreteSubject subject = new ConcreteSubject();  //定义一个观察者  Observer obs= new ConcreteObserver();  //观察者观察被观察者  subject.addObserver(obs);  //观察者开始活动了  subject.doSomething();  }  } |

**7.2 例2【设计模式之禅】**



* **IHanFeiZi.java**

|  |
| --- |
| package designmode.\_10observer;  /\*\*  \* 被观察者接口  \*/  public interface IHanFeiZi {  //韩非子吃早饭  public void haveBreakfast();    //韩非子娱乐  public void haveFun();  } |

* **Observable.java**

|  |
| --- |
| package designmode.\_10observer;  /\*\*  \* 被观察者接口  \*/  public interface Observable {  //增加一个观察者  public void addObserver(Observer observer);    //删除一个观察者  public void deleteObserver(Observer observer);    //通知观察者  public void notifyObserver(String context);  } |

* **HanFeiZi.java**

|  |
| --- |
| package designmode.\_10observer;  import java.util.ArrayList;  /\*\*  \* 被观察者实现类  \*/  public class HanFeiZi implements Observable, IHanFeiZi{  //定义一个变长数组，存放所有观察者  private ArrayList<Observer> observerList = new ArrayList<Observer>();    //增加观察者  @Override  public void addObserver(Observer observer) {  this.observerList.add(observer);  }  @Override  public void deleteObserver(Observer observer) {  this.observerList.remove(observer);  }  @Override  public void notifyObserver(String context) {  for(Observer observer : observerList) {  observer.update(context);  }  }    //韩非子要吃饭了  @Override  public void haveBreakfast() {  System.out.println("韩非子:开始吃饭了...");  //通知所有观察者  this.notifyObserver("韩非子在吃饭");  }    //韩非子开始娱乐了  @Override  public void haveFun() {  System.out.println("韩非子:开始娱乐了...");  this.notifyObserver("韩非子在娱乐");  }  } |

* **Observer.java**

|  |
| --- |
| package designmode.\_10observer;  /\*\*  \* 观察者接口  \*/  public interface Observer {  //一发现别人有动静，自己要行动起来  public void update(String context);  } |

* **Lisi.java**

|  |
| --- |
| package designmode.\_10observer;  public class Lisi implements Observer{  //首先李斯是一个观察者，一旦韩非子有活动，他就知道，就向老板报告  @Override  public void update(String context) {  System.out.println("李斯：观察到韩非子活动，开始向老板汇报了...");  this.reportToQingshihuang(context);  System.out.println("李斯:汇报完毕...\n");    }    private void reportToQingshihuang(String reportContent) {  System.out.println("李斯:报告，秦老板!韩非子有活动了-->" + reportContent);    }  } |

WangSi.java

|  |
| --- |
| package designmode.\_10observer;  /\*\*  \* 杜撰的观察者:王斯  \*/  public class WangSi implements Observer{  //王斯:看到韩非子有活动  @Override  public void update(String context) {  System.out.println("王斯：观察到韩非子活动自己也开始活动了...");  this.cry(context);  System.out.println("王斯，哭死了...\n");  }    private void cry(String context) {  System.out.println("王斯:因为" + context + ", -- 所以我悲伤呀!");  }  } |

* 运行结果，打印:

|  |
| --- |
| 韩非子:开始吃饭了...  李斯：观察到韩非子活动，开始向老板汇报了...  李斯:报告，秦老板!韩非子有活动了-->韩非子在吃饭  李斯:汇报完毕...  王斯：观察到韩非子活动自己也开始活动了...  王斯:因为韩非子在吃饭, -- 所以我悲伤呀!  王斯，哭死了...  刘斯：观察到韩非子活动，开始动作了...  刘斯:因为韩非子在吃饭,--所以我快乐呀!  刘斯:乐死了... |

**7.4 其它部分见设计模式之禅**

1. **命令模式**

命令模式(Bridge)，将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化：对请求队列或记录请求日志，以及支持可撤销的操作。

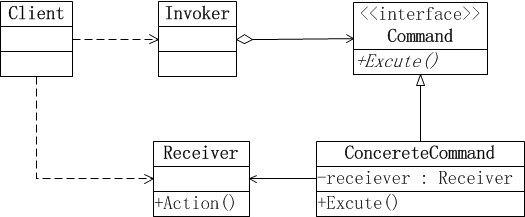
命令模式4个角色：

1、 Command ：定义命令的统一接口。

2、ConcreteCommand ： Command接口的实现者，用来执行具体的命令，某些情况下可以直接用来充当Receiver。

3、 Receiver：命令的实际执行者。

4、Invoker：命令的请求者，是命令模式中最重要的角色。这个角色用来对各个命令进行控制。



使用命令模式的场合：

命令的发送者和命令执行者有不同的生命周期。命令发送了并不是立即执行。

命令需要进行各种管理逻辑。

需要支持撤消\重做操作。