# 设计模式说明文档

## 设计模式须知

### 产生背景

“设计模式”这个术语最初并不是出现在软件设计中，而是被用于建筑领域的设计中。

1977 年，美国著名建筑大师、加利福尼亚大学伯克利分校环境结构中心主任克里斯托夫·亚历山大（Christopher Alexander）在他的著作《建筑模式语言：城镇、建筑、构造（A Pattern Language: Towns Building Construction）中描述了一些常见的建筑设计问题，并提出了 253 种关于对城镇、邻里、住宅、花园和房间等进行设计的基本模式。

1979 年他的另一部经典著作《建筑的永恒之道》（The Timeless Way of Building）进一步强化了设计模式的思想，为后来的建筑设计指明了方向。

1987 年，肯特·贝克（Kent Beck）和沃德·坎宁安（Ward Cunningham）首先将克里斯托夫·亚历山大的模式思想应用在 Smalltalk 中的图形用户接口的生成中，但没有引起软件界的关注。

直到 1990 年，软件工程界才开始研讨设计模式的话题，后来召开了多次关于设计模式的研讨会。

1995 年，艾瑞克·伽马（ErichGamma）、理査德·海尔姆（Richard Helm）、拉尔夫·约翰森（Ralph Johnson）、约翰·威利斯迪斯（John Vlissides）等 4 位作者合作出版了《设计模式：可复用面向对象软件的基础》（Design Patterns: Elements of Reusable Object-Oriented Software）一书，在本教程中收录了 23 个设计模式，这是设计模式领域里程碑的事件，导致了软件设计模式的突破。这 4 位作者在软件开发领域里也以他们的“四人组”（Gang of Four，GoF）匿名著称。

直到今天，狭义的设计模式还是本教程中所介绍的 23 种经典设计模式。

### 概念与意义

有关软件设计模式的定义很多，有些从模式的特点来说明，有些从模式的作用来说明。本文给出的定义是大多数学者公认的，从以下两个方面来说明：

#### 1. 软件设计模式的概念

软件设计模式（Software Design Pattern），又称设计模式，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。

它描述了在软件设计过程中的一些不断重复发生的问题，以及该问题的解决方案。

也就是说，它是解决特定问题的一系列套路，是前辈们的代码设计经验的总结，具有一定的普遍性，可以反复使用。

其目的是为了提高代码的可重用性、代码的可读性和代码的可靠性。

#### 2. 学习设计模式的意义

设计模式的本质是面向对象设计原则的实际运用，是对类的封装性、继承性和多态性以及类的关联关系和组合关系的充分理解。正确使用设计模式具有以下优点：

1. 可以提高程序员的思维能力、编程能力和设计能力
2. 使程序设计更加标准化、代码编制更加工程化，使软件开发效率大大提高，从而缩短软件的开发周期
3. 使设计的代码可重用性高、可读性强、可靠性高、灵活性好、可维护性强

当然，软件设计模式只是一个引导。在具体的软件幵发中，必须根据设计的应用系统的特点和要求来恰当选择。

对于简单的程序开发，可能写一个简单的算法要比引入某种设计模式更加容易。

但对大项目的开发或者框架设计，用设计模式来组织代码显然更好。

### 基本要素

软件设计模式使人们可以更加简单方便地复用成功的设计和体系结构，它通常包含以下几个基本要素：模式名称、别名、动机、问题、解决方案、效果、结构、模式角色、合作关系、实现方法、适用性、已知应用、例程、模式扩展和相关模式等，其中最关键的元素包括以下4个主要部分：

#### 1.模式名称

每一个模式都有自己的名字，通常用一两个词来描述，可以根据模式的问题、特点、解决方案、功能和效果来命名。

模式名称（PatternName）有助于我们理解和记忆该模式，也方便我们来讨论自己的设计。

#### 2.问题

问题（Problem）描述了该模式的应用环境，即何时使用该模式。它解释了设计问题和问题存在的前因后果，以及必须满足的一系

列先决条件。

#### 3.解决方案

模式问题的解决方案（Solution）包括设计的组成成分、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（类或对象的组合）来解决这个问题。

#### 4.效果

描述了模式的应用效果以及使用该模式应该权衡的问题，即模式的优缺点。

主要是对时间和空间的衡量，以及该模式对系统的灵活性、扩充性、可移植性的影响，也考虑其实现问题。

显式地列出这些效果（Consequence）对理解和评价这些模式有很大的帮助。

### 分类及功能

设计模式有两种分类方法，即根据模式的目的来分和根据模式的作用的范围来分。

#### 1.根据目的来分

根据模式是用来完成什么工作来划分，这种方式可分为创建型模式、结构型模式和行为型模式 3 种。

##### 创建型模式

用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”。

GoF 中提供了单例、原型、工厂方法、抽象工厂、建造者等 5 种创建型模式。

##### 结构型模式

用于描述如何将类或对象按某种布局组成更大的结构。

GoF 中提供了代理、适配器、桥接、装饰、外观、享元、组合等 7 种结构型模式。

##### 行为型模式

用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责。

GoF 中提供了模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器等 11 种行为型模式。

#### 2.根据作用范围来分

根据模式是主要用于类上还是主要用于对象上来分，这种方式可分为类模式和对象模式两种。

##### 类模式

用于处理类与子类之间的关系，这些关系通过继承来建立，是静态的，在编译时刻便确定下来了。

GoF中的工厂方法、（类）适配器、模板方法、解释器属于该模式。

##### 对象模式

用于处理对象之间的关系，这些关系可以通过组合或聚合来实现，在运行时刻是可以变化的，更具动态性。

GoF 中除了以上 4 种，其他的都是对象模式。

#### 3.GoF的23种设计模式的功能

前面说明了 GoF 的 23 种设计模式的分类，现在对各个模式的功能进行介绍。

##### 单例（Singleton）模式

某个类只能生成一个实例，该类提供了一个全局访问点供外部获取该实例，其拓展是有限多例模式。

##### 原型（Prototype）模式

将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。

##### 工厂方法（Factory Method）模式

定义一个用于创建产品的接口，由子类决定生产什么产品。

##### 抽象工厂（AbstractFactory）模式

提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。

##### 建造者（Builder）模式

将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。

##### 代理（Proxy）模式

为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。

##### 适配器（Adapter）模式

将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

##### 桥接（Bridge）模式

将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。

##### 装饰（Decorator）模式

动态的给对象增加一些职责，即增加其额外的功能。

##### 外观（Facade）模式

为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。

##### 享元（Flyweight）模式

运用共享技术来有效地支持大量细粒度对象的复用。

##### 组合（Composite）模式

将对象组合成树状层次结构，使用户对单个对象和组合对象具有一致的访问性。

##### 模板方法（TemplateMethod）模式

定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。

##### 策略（Strategy）模式

定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。

##### 命令（Command）模式

将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。

##### 职责链（Chain of Responsibility）模式

把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。

##### 状态（State）模式

允许一个对象在其内部状态发生改变时改变其行为能力。

##### 观察者（Observer）模式

多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。

##### 中介者（Mediator）模式

定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。

##### 迭代器（Iterator）模式

提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

##### 访问者（Visitor）模式

在不改变集合元素的前提下，为一个集合中的每个元素提供多种访问方式，即每个元素有多个访问者对象访问。

##### 备忘录（Memento）模式

在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。

##### 解释器（Interpreter）模式

提供如何定义语言的文法，以及对语言句子的解释方法，即解释器。

#### 特别说明

以上23 种设计模式并不是孤立存在的，很多模式之间存在一定的关联关系，在大的系统开发中常常同时使用多种设计模式。

### 设计原则

在软件开发中，为了提高软件系统的可维护性和可复用性，增加软件的可扩展性和灵活性，程序员要尽量根据7条原则来开发程序，从而提高软件开发效率、节约软件开发成本和维护成本。

我们将在下面依次来介绍这7条原则（开闭原则、里氏替换原则、依赖倒置原则、单一职责原则、接口隔离原则、迪米特原则、合成复用原则）。

#### 开闭原则

##### 定义

开闭原则（Open Closed Principle，OCP）由勃兰特·梅耶（Bertrand Meyer）提出，他在 1988 年的著作《面向对象软件构造》（Object Oriented Software Construction）中提出：软件实体应当对扩展开放，对修改关闭（Software entities should be open for extension，but closed for modification），这就是开闭原则的经典定义。

这里的软件实体包括以下几个部分：

1.项目中划分出的模块

2.类与接口

3.方法

开闭原则的含义：当应用的需求改变时，在不修改软件实体的源代码或者二进制代码的前提下，可以扩展模块的功能，使其满足新的需求。

##### 作用

开闭原则是面向对象程序设计的终极目标，它使软件实体拥有一定的适应性和灵活性的同时具备稳定性和延续性。具体来说，其作用如下：

###### 1.对软件测试的影响

软件遵守开闭原则的话，软件测试时只需要对扩展的代码进行测试就可以了，因为原有的测试代码仍然能够正常运行。

###### 2.可以提高代码的可复用性

粒度越小，被复用的可能性就越大；在面向对象的程序设计中，根据原子和抽象编程可以提高代码的可复用性。

###### 3.可以提高软件的可维护性

遵守开闭原则的软件，其稳定性高和延续性强，从而易于扩展和维护。

##### 实现方法

可以通过“抽象约束、封装变化”来实现开闭原则，即通过接口或者抽象类为软件实体定义一个相对稳定的抽象层，而将相同的可变因素封装在相同的具体实现类中。

因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件架构的稳定。而软件中易变的细节可以从抽象派生来的实现类来进行扩展，当软件需要发生变化时，只需要根据需求重新派生一个实现类来扩展就可以了。

##### 说明示例

下面以 Windows 的桌面主题为例介绍开闭原则的应用。

分析：Windows 的主题是桌面背景图片、窗口颜色和声音等元素的组合。用户可以根据自己的喜爱更换自己的桌面主题，也可以从网上下载新的主题。

这些主题有共同的特点，可以为其定义一个抽象类（Abstract Subject），而每个具体的主题（Specific Subject）是其子类。

用户窗体可以根据需要选择或者增加新的主题，而不需要修改原代码，所以它是满足开闭原则的，其类图如下：



#### 里氏替换原则

##### 定义

里氏替换原则（Liskov Substitution Principle，LSP）由麻省理工学院计算机科学实验室的里斯科夫（Liskov）女士在 1987 年的“面向对象技术的高峰会议”（OOPSLA）上发表的一篇文章《数据抽象和层次》（Data Abstraction and Hierarchy）里提出来的，她提出：继承必须确保超类所拥有的性质在子类中仍然成立（Inheritance should ensure that any property proved about supertype objects also holds for subtype objects）。

里氏替换原则主要阐述了有关继承的一些原则，也就是什么时候应该使用继承，什么时候不应该使用继承，以及其中蕴含的原理。

里氏替换原是继承复用的基础，它反映了基类与子类之间的关系，是对开闭原则的补充，是对实现抽象化的具体步骤的规范。

##### 作用

里氏替换原则的主要作用如下:

1.里氏替换原则是实现开闭原则的重要方式之一。

2.它克服了继承中重写父类造成的可复用性变差的缺点。

3.它是动作正确性的保证。即类的扩展不会给已有的系统引入新的错误，降低了代码出错的可能性。

##### 实现方法

里氏替换原则通俗来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。

也就是说：子类继承父类时，除添加新的方法完成新增功能外，尽量不要重写父类的方法。

如果通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的概率会非常大。

如果程序违背了里氏替换原则，则继承类的对象在基类出现的地方会出现运行错误。这时其修正方法是：取消原来的继承关系，重新设计它们之间的关系。

关于里氏替换原则的例子，最有名的是“正方形不是长方形”。当然，生活中也有很多类似的例子，例如，企鹅、鸵鸟和几维鸟从生物学的角度来划分，它们属于鸟类；但从类的继承关系来看，由于它们不能继承“鸟”会飞的功能，所以它们不能定义成“鸟”的子类。同样，由于“气球鱼”不会游泳，所以不能定义成“鱼”的子类；“玩具炮”炸不了敌人，所以不能定义成“炮”的子类等。

##### 说明示例

下面以“几维鸟不是鸟”为例来说明里氏替换原则。

分析：鸟一般都会飞行，如燕子的飞行速度大概是每小时 120 千米。但是新西兰的几维鸟由于翅膀退化无法飞行。假如要设计一个实例，计算这两种鸟飞行 300 千米要花费的时间。显然，拿燕子来测试这段代码，结果正确，能计算出所需要的时间；但拿几维鸟来测试，结果会发生“除零异常”或是“无穷大”，明显不符合预期，其类图如下图所示：



程序代码如下：

package principle;

public class LSPtest

{

public static void main(String[] args)

{

Bird bird1=new Swallow();

Bird bird2=new BrownKiwi();

bird1.setSpeed(120);

bird2.setSpeed(120);

System.out.println("如果飞行300公里：");

try

{

System.out.println("燕子将飞行"+bird1.getFlyTime(300)+"小时.");

System.out.println("几维鸟将飞行"+bird2.getFlyTime(300)+"小时。");

}

catch(Exception err)

{

System.out.println("发生错误了!");

}

}

}

//鸟类

class Bird

{

double flySpeed;

public void setSpeed(double speed)

{

flySpeed=speed;

}

public double getFlyTime(double distance)

{

return(distance/flySpeed);

}

}

//燕子类

class Swallow extends Bird{}

//几维鸟类

class BrownKiwi extends Bird

{

public void setSpeed(double speed)

{

flySpeed=0;

}

}

程序的运行结果如下：

如果飞行300公里：

燕子将飞行2.5小时.

几维鸟将飞行Infinity小时。

程序运行错误的原因：几维鸟类重写了鸟类的 setSpeed(double speed) 方法，这违背了里氏替换原则。正确的做法是：取消几维鸟原来的继承关系，定义鸟和几维鸟的更一般的父类，如动物类，它们都有奔跑的能力。几维鸟的飞行速度虽然为 0，但奔跑速度不为 0，可以计算出其奔跑 300 千米所要花费的时间。其类图如下图所示：



#### 依赖倒置原则

##### 定义

依赖倒置原则（Dependence Inversion Principle，DIP）是 Object Mentor 公司总裁罗伯特·马丁（Robert C.Martin）于 1996 年在 [C++](http://c.biancheng.net/cplus/) Report 上发表的文章。

依赖倒置原则的原始定义：高层模块不应该依赖低层模块，两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（High level modules shouldnot depend upon low level modules.Both should depend upon abstractions.Abstractions should not depend upon details. Details should depend upon abstractions）。其核心思想是：要面向接口编程，不要面向实现编程。

依赖倒置原则是实现开闭原则的重要途径之一，它降低了客户与实现模块之间的耦合。

由于在软件设计中，细节具有多变性，而抽象层则相对稳定，因此以抽象为基础搭建起来的架构要比以细节为基础搭建起来的架构要稳定得多。这里的抽象指的是接口或者抽象类，而细节是指具体的实现类。

使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给它们的实现类去完成。

##### 作用

依赖倒置原则的主要作用如下：

1.依赖倒置原则可以降低类间的耦合性

2.依赖倒置原则可以提高系统的稳定性

3.依赖倒置原则可以减少并行开发引起的风险

4.依赖倒置原则可以提高代码的可读性和可维护性。

##### 实现方法

依赖倒置原则的目的是通过要面向接口的编程来降低类间的耦合性，所以我们在实际编程中只要遵循以下4点，就能在项目中满足这个规则。

1.每个类尽量提供接口或抽象类，或者两者都具备

2.变量的声明类型尽量是接口或者是抽象类

3.任何类都不应该从具体类派生

4.使用继承时尽量遵循里氏替换原则。

##### 说明示例

下面以“顾客购物程序”为例来说明依赖倒置原则的应用。

分析：本程序反映了 “顾客类”与“商店类”的关系。商店类中有 sell() 方法，顾客类通过该方法购物以下代码定义了顾客类通过韶关网店 ShaoguanShop 购物：

class Customer

{

public void shopping(ShaoguanShop shop)

{

//购物

System.out.println（shop.sell());

}

}

但是，这种设计存在缺点，如果该顾客想从另外一家商店（如婺源网店 WuyuanShop）购物，就要将该顾客的代码修改如下：

class Customer

{

public void shopping(WuyuanShop shop)

{

//购物

System.out.println(shop.sell());

}

}

顾客每更换一家商店，都要修改一次代码，这明显违背了开闭原则。存在以上缺点的原因是：顾客类设计时同具体的商店类绑定了，这违背了依赖倒置原则。解决方法是：定义“婺源网店”和“韶关网店”的共同接口 Shop，顾客类面向该接口编程，其代码修改如下：

class Customer

{

public void shopping(Shop shop)

{

//购物

System.out.println(shop.sell());

}

}

这样，不管顾客类 Customer 访问什么商店，或者增加新的商店，都不需要修改原有代码了，其类图如下图所示：



程序代码如下：

package principle;

public class DIPtest

{

public static void main(String[] args)

{

Customer wang=new Customer();

System.out.println("顾客购买以下商品：");

wang.shopping(new ShaoguanShop());

wang.shopping(new WuyuanShop());

}

}

//商店

interface Shop

{

public String sell(); //卖

}

//韶关网店

class ShaoguanShop implements Shop

{

public String sell()

{

return "韶关土特产：香菇、木耳……";

}

}

//婺源网店

class WuyuanShop implements Shop

{

public String sell()

{

return "婺源土特产：绿茶、酒糟鱼……";

}

}

//顾客

class Customer

{

public void shopping(Shop shop)

{

//购物

System.out.println(shop.sell());

}

}

程序的运行结果如下：

顾客购买以下商品：

韶关土特产：香菇、木耳……

婺源土特产：绿茶、酒糟鱼……

#### 单一职责原则

##### 定义

单一职责原则（Single Responsibility Principle，SRP）又称单一功能原则，由罗伯特·C.马丁（Robert C. Martin）于《敏捷软件开发：原则、模式和实践》一书中提出的。

这里的职责是指类变化的原因，单一职责原则规定一个类应该有且仅有一个引起它变化的原因，否则类应该被拆分（There should never be more than one reason for a class to change）。

该原则提出对象不应该承担太多职责，如果一个对象承担了太多的职责，至少存在以下两个缺点：

1.一个职责的变化可能会削弱或者抑制这个类实现其他职责的能力；

2.当客户端需要该对象的某一个职责时，不得不将其他不需要的职责全都包含进来，从而造成冗余代码或代码的浪费。

##### 作用

单一职责原则的核心就是控制类的粒度大小、将对象解耦、提高其内聚性。

如果遵循单一职责原则将有以下优点。

###### 1.降低类的复杂度

一个类只负责一项职责，其逻辑肯定要比负责多项职责简单得多。

###### 2.提高类的可读性

复杂性降低，自然其可读性会提高。

###### 3.提高系统的可维护性

可读性提高，那自然更容易维护了。

###### 4.变更引起的风险降低

变更是必然的，如果单一职责原则遵守得好，当修改一个功能时，可以显著降低对其他功能的影响。

##### 实现方法

单一职责原则是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，再封装到不同的类或模块中。

而发现类的多重职责需要设计人员具有较强的分析设计能力和相关重构经验。

##### 说明示例

下面以大学学生工作管理程序为例介绍单一职责原则的应用。

分析：大学学生工作主要包括学生生活辅导和学生学业指导两个方面的工作，其中生活辅导主要包括班委建设、出勤统计、心理辅导、费用催缴、班级管理等工作，学业指导主要包括专业引导、学习辅导、科研指导、学习总结等工作。

如果将这些工作交给一位老师负责显然不合理，正确的做 法是生活辅导由辅导员负责，学业指导由学业导师负责，其类图如下所示：



注意：单一职责同样也适用于方法。一个方法应该尽可能做好一件事情。如果一个方法处理的事情太多，其颗粒度会变得很粗，不利于重用。

#### 接口隔离原则

##### 定义

接口隔离原则（Interface Segregation Principle，ISP）要求程序员尽量将臃肿庞大的接口拆分成更小的和更具体的接口，让接口中只包含客户感兴趣的方法。

2002 年罗伯特·C.马丁给“接口隔离原则”的定义是：客户端不应该被迫依赖于它不使用的方法（Clients should not be forced to depend on methods they do not use）。

该原则还有另外一个定义：一个类对另一个类的依赖应该建立在最小的接口上（The dependency of one class to another one should depend on the smallest possible interface）。

以上两个定义的含义是：要为各个类建立它们需要的专用接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。

接口隔离原则和单一职责都是为了提高类的内聚性、降低它们之间的耦合性，体现了封装的思想，但两者是不同的：

1.单一职责原则注重的是职责，而接口隔离原则注重的是对接口依赖的隔离

2.单一职责原则主要是约束类，它针对的是程序中的实现和细节；接口隔离原则主要约束接口，主要针对抽象和程序整体框架的构建。

##### 作用

接口隔离原则是为了约束接口、降低类对接口的依赖性，遵循接口隔离原则有以下 5 个优点：

1.将臃肿庞大的接口分解为多个粒度小的接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。

2.接口隔离提高了系统的内聚性，减少了对外交互，降低了系统的耦合性。

3.如果接口的粒度大小定义合理，能够保证系统的稳定性；但是，如果定义过小，则会造成接口数量过多，使设计复杂化；如果定义太大，灵活性降低，无法提供定制服务，给整体项目带来无法预料的风险。

4.使用多个专门的接口还能够体现对象的层次，因为可以通过接口的继承，实现对总接口的定义。

5.能减少项目工程中的代码冗余。过大的大接口里面通常放置许多不用的方法，当实现这个接口的时候，被迫设计冗余的代码。

##### 实现方法

在具体应用接口隔离原则时，应该根据以下几个规则来衡量：

###### 1.接口尽量小，但是要有限度

一个接口只服务于一个子模块或业务逻辑。

###### 2.为依赖接口的类定制服务

只提供调用者需要的方法，屏蔽不需要的方法。

###### 3.了解环境，拒绝盲从

每个项目或产品都有选定的环境因素，环境不同，接口拆分的标准就不同深入了解业务逻辑。

###### 4.提高内聚，减少对外交互

使接口用最少的方法去完成最多的事情。

##### 说明示例

下面以学生成绩管理程序为例介绍接口隔离原则的应用。

分析：学生成绩管理程序一般包含插入成绩、删除成绩、修改成绩、计算总分、计算均分、打印成绩信息、査询成绩信息等功能，如果将这些功能全部放到一个接口中显然不太合理，正确的做法是将它们分别放在输入模块、统计模块和打印模块等 3 个模块中，其类图如下所示：



程序代码如下：

package principle;

public class ISPtest

{

    public static void main(String[] args)

    {

        InputModule input =StuScoreList.getInputModule();

        CountModule count =StuScoreList.getCountModule();

        PrintModule print =StuScoreList.getPrintModule();

        input.insert();

        count.countTotalScore();

        print.printStuInfo();

        //print.delete();

    }

}

//输入模块接口

interface InputModule

{

    void insert();

    void delete();

    void modify();

}

//统计模块接口

interface CountModule

{

    void countTotalScore();

    void countAverage();

}

//打印模块接口

interface PrintModule

{

    void printStuInfo();

    void queryStuInfo();

}

//实现类

class StuScoreList implements InputModule,CountModule,PrintModule

{

    private StuScoreList(){}

    public static InputModule getInputModule()

    {

        return (InputModule)new StuScoreList();

    }

    public static CountModule getCountModule()

    {

        return (CountModule)new StuScoreList();

    }

    public static PrintModule getPrintModule()

    {

        return (PrintModule)new StuScoreList();

    }

    public void insert()

    {

        System.out.println("输入模块的insert()方法被调用！");

    }

    public void delete()

    {

        System.out.println("输入模块的delete()方法被调用！");

    }

    public void modify()

    {

        System.out.println("输入模块的modify()方法被调用！");

    }

    public void countTotalScore()

    {

        System.out.println("统计模块的countTotalScore()方法被调用！");

    }

    public void countAverage()

    {

        System.out.println("统计模块的countAverage()方法被调用！");

    }

    public void printStuInfo()

    {

        System.out.println("打印模块的printStuInfo()方法被调用！");

    }

    public void queryStuInfo()

    {

        System.out.println("打印模块的queryStuInfo()方法被调用！");

    }

}

程序的运行结果如下：

输入模块的insert()方法被调用！

统计模块的countTotalScore()方法被调用！

打印模块的printStuInfo()方法被调用！

#### 迪米特原则

##### 定义

迪米特法则（Law of Demeter，LoD）又叫作最少知识原则（Least Knowledge Principle，LKP)，产生于 1987 年美国东北大学（Northeastern University）的一个名为迪米特（Demeter）的研究项目，由伊恩·荷兰（Ian Holland）提出，被 UML 创始者之一的布奇（Booch）普及，后来又因为在经典著作《程序员修炼之道》（The Pragmatic Programmer）提及而广为人知。

迪米特法则的定义：只与你的直接朋友交谈，不跟“陌生人”说话（Talk only to your immediate friends and not to strangers）。其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

迪米特法则中的“朋友”：当前对象本身、当前对象的成员对象、当前对象所创建的对象、当前对象的方法参数等，这些对象同当前对象存在关联、聚合或组合关系，可以直接访问这些对象的方法。

##### 作用

迪米特法则要求限制软件实体之间通信的宽度和深度，正确使用迪米特法则将有以下两个优点：

1.降低了类之间的耦合度，提高了模块的相对独立性

2.由于亲合度降低，从而提高了类的可复用率和系统的扩展性。

但是，过度使用迪米特法则会使系统产生大量的中介类，从而增加系统的复杂性，使模块之间的通信效率降低。

所以，在釆用迪米特法则时需要反复权衡，确保高内聚和低耦合的同时，保证系统的结构清晰。

##### 实现方法

从迪米特法则的定义和特点可知，它强调以下两点：

1.从依赖者的角度来说，只依赖应该依赖的对象

2.从被依赖者的角度说，只暴露应该暴露的方法。

所以，在运用迪米特法则时要注意以下 6点：

1.在类的划分上，应该创建弱耦合的类。类与类之间的耦合越弱，就越有利于实现可复用的目标

2.在类的结构设计上，尽量降低类成员的访问权限

3.在类的设计上，优先考虑将一个类设置成不变类

4.在对其他类的引用上，将引用其他对象的次数降到最低

5.不暴露类的属性成员，而应该提供相应的访问器（set 和 get 方法）

6.谨慎使用序列化（Serializable）功能。

##### 说明示例

下面以明星与经纪人的关系为例介绍迪米特法则的应用。

分析：明星由于全身心投入艺术，所以许多日常事务由经纪人负责处理，如与粉丝的见面会，与媒体公司的业务洽淡等。

这里的经纪人是明星的朋友，而粉丝和媒体公司是陌生人，所以适合使用迪米特法则，其类图如下所示：



程序代码如下：

package principle;

public class LoDtest

{

public static void main(String[] args)

{

Agent agent=new Agent();

agent.setStar(new Star("林心如"));

agent.setFans(new Fans("粉丝韩丞"));

agent.setCompany(new Company("中国传媒有限公司"));

agent.meeting();

agent.business();

}

}

//经纪人

class Agent

{

private Star myStar;

private Fans myFans;

private Company myCompany;

public void setStar(Star myStar)

{

this.myStar=myStar;

}

public void setFans(Fans myFans)

{

this.myFans=myFans;

}

public void setCompany(Company myCompany)

{

this.myCompany=myCompany;

}

public void meeting()

{

System.out.println(myFans.getName()+"与明星"+myStar.getName()+"见面了。");

}

public void business()

{

System.out.println(myCompany.getName()+"与明星"+myStar.getName()+"洽淡业务。");

}

}

//明星

class Star

{

private String name;

Star(String name)

{

this.name=name;

}

public String getName()

{

return name;

}

}

//粉丝

class Fans

{

private String name;

Fans(String name)

{

this.name=name;

}

public String getName()

{

return name;

}

}

//媒体公司

class Company

{

private String name;

Company(String name)

{

this.name=name;

}

public String getName()

{

return name;

}

}

程序的运行结果如下：

粉丝韩丞与明星林心如见面了。

中国传媒有限公司与明星林心如洽淡业务。

#### 合成复用原则

##### 定义

合成复用原则（Composite Reuse Principle，CRP）又叫组合/聚合复用原则（Composition/Aggregate Reuse Principle，CARP）。

它要求在软件复用时，要尽量先使用组合或者聚合等关联关系来实现，其次才考虑使用继承关系来实现。

如果要使用继承关系，则必须严格遵循里氏替换原则。

合成复用原则同里氏替换原则相辅相成的，两者都是开闭原则的具体实现规范。

##### 作用

通常类的复用分为继承复用和合成复用两种，继承复用虽然有简单和易实现的优点，但它也存在以下缺点：

1. 继承复用破坏了类的封装性

因为继承会将父类的实现细节暴露给子类，父类对子类是透明的，所以这种复用又称为“白箱”复用。

1. 子类与父类的耦合度高

父类的实现的任何改变都会导致子类的实现发生变化，这不利于类的扩展与维护。

3.它限制了复用的灵活性

从父类继承而来的实现是静态的，在编译时已经定义，所以在运行时不可能发生变化。

采用组合或聚合复用时，可以将已有对象纳入新对象中，使之成为新对象的一部分，新对象可以调用已有对象的功能，它有以下优点：

1. 它维持了类的封装性

因为成分对象的内部细节是新对象看不见的，所以这种复用又称为“黑箱”复用。

2.新旧类之间的耦合度低

这种复用所需的依赖较少，新对象存取成分对象的唯一方法是通过成分对象的接口。

3.复用的灵活性高

这种复用可以在运行时动态进行，新对象可以动态地引用与成分对象类型相同的对象。

##### 实现方法

合成复用原则是通过将已有的对象纳入新对象中，作为新对象的成员对象来实现的，新对象可以调用已有对象的功能，从而达到复用。

##### 说明示例

下面以汽车分类管理程序为例来介绍合成复用原则的应用。

分析：汽车按“动力源”划分可分为汽油汽车、电动汽车等；按“颜色”划分可分为白色汽车、黑色汽车和红色汽车等。

如果同时考虑这两种分类，其组合就很多。下图所示是用继承关系实现的汽车分类的类图：



从上图可以看出用继承关系实现会产生很多子类，而且增加新的“动力源”或者增加新的“颜色”都要修改源代码，这违背了开闭原则，显然不可取。

但如果改用组合关系实现就能很好地解决以上问题，其类图如下所示：



#### 设计原则总结

7 种设计原则是软件设计模式必须尽量遵循的原则，各种原则要求的侧重点不同。具体如下：

1. 开闭原则是总纲，它告诉我们要对扩展开放，对修改关闭
2. 里氏替换原则告诉我们不要破坏继承体系
3. 依赖倒置原则告诉我们要面向接口编程
4. 单一职责原则告诉我们实现类要职责单一
5. 接口隔离原则告诉我们在设计接口的时候要精简单一
6. 迪米特法则告诉我们要降低耦合度
7. 合成复用原则告诉我们要优先使用组合或者聚合关系复用，少用继承关系复用。

## 设计模式详解

### 单例模式详解

在有些系统中，为了节省内存资源、保证数据内容的一致性，对某些类要求只能创建一个实例，这就是所谓的单例模式。

在有些系统中，为了节省内存资源、保证数据内容的一致性，对某些类要求只能创建一个实例，这就是所谓的单例模式。

#### 定义与特点

##### 定义

一个类只有一个实例，且该类能自行创建这个实例的一种模式。例如，Windows 中只能打开一个任务管理器，这样可以避免因打开多个任务管理器窗口而造成内存资源的浪费，或出现各个窗口显示内容的不一致等错误。

在计算机系统中，还有 Windows 的回收站、操作系统中的文件系统、多线程中的线程池、显卡的驱动程序对象、打印机的后台处理服务、应用程序的日志对象、数据库的连接池、网站的计数器、Web 应用的配置对象、应用程序中的对话框、系统中的缓存等常常被设计成单例。

##### 特点

单例模式有 3 个特点：

1.单例类只有一个实例对象

2.该单例对象必须由单例类自行创建

3.单例类对外提供一个访问该单例的全局访问点。

#### 结构与实现

单例模式是设计模式中最简单的模式之一。

通常，普通类的构造函数是公有的，外部类可以通过“new 构造函数()”来生成多个实例。

但是，如果将类的构造函数设为私有的，外部类就无法调用该构造函数，也就无法生成多个实例。

这时该类自身必须定义一个静态私有实例，并向外提供一个静态的公有函数用于创建或获取该静态私有实例。

下面来分析其基本结构和实现方法。

##### 结构

单例模式的主要角色如下：

单例类：包含一个实例且能自行创建这个实例的类。

访问类：使用单例的类。

其结构如下图所示：



##### 实现

Singleton 模式通常有两种实现形式。

###### 懒汉式单例

该模式的特点是类加载时没有生成单例，只有当第一次调用 getlnstance 方法时才去创建这个单例。

代码如下：

public class LazySingleton

{

private static volatile LazySingleton instance=null; //保证 instance 在所有线程中同步

private LazySingleton(){} //private 避免类在外部被实例化

public static synchronized LazySingleton getInstance()

{

//getInstance 方法前加同步

if(instance==null)

{

instance=new LazySingleton();

}

return instance;

}

}

注意：如果编写的是多线程程序，则不要删除上例代码中的关键字 volatile 和 synchronized，否则将存在线程非安全的问题。如果不删除这两个关键字就能保证线程安全，但是每次访问时都要同步，会影响性能，且消耗更多的资源，这是懒汉式单例的缺点。

###### 饿汉式单例

该模式的特点是类一旦加载就创建一个单例，保证在调用 getInstance 方法之前单例已经存在了。

代码如下：

public class HungrySingleton

{

private static final HungrySingleton instance=new HungrySingleton();

private HungrySingleton(){}

public static HungrySingleton getInstance()

{

return instance;

}

}

饿汉式单例在类创建的同时就已经创建好一个静态的对象供系统使用，以后不再改变，所以是线程安全的，可以直接用于多线程而不会出现问题。

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1338.html

#### 应用场景

前面分析了单例模式的结构与特点，以下是它通常适用的场景的特点：

1.在应用场景中，某类只要求生成一个对象的时候，如一个班中的班长、每个人的身份证号等

2.当对象需要被共享的场合。由于单例模式只允许创建一个对象，共享该对象可以节省内存，并加快对象访问速度。如 Web 中的配置对象、数据库的连接池等。

3.当某类需要频繁实例化，而创建的对象又频繁被销毁的时候，如多线程的线程池、网络连接池等。

#### 扩展

单例模式可扩展为有限的多例（Multitcm）模式，这种模式可生成有限个实例并保存在 ArmyList 中，客户需要时可随机获取，其结构图如下图所示：



### 原型模式详解

在有些系统中，存在大量相同或相似对象的创建问题，如果用传统的构造函数来创建对象，会比较复杂且耗时耗资源，用原型模式生成对象就很高效，就像孙悟空拔下猴毛轻轻一吹就变出很多孙悟空一样简单。

#### 定义与特点

##### 定义

用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型相同或相似的新对象。在这里，原型实例指定了要创建的对象的种类。用这种方式创建对象非常高效，根本无须知道对象创建的细节。

例如，Windows 操作系统的安装通常较耗时，如果复制就快了很多。在生活中复制的例子非常多，这里不一一列举了。

#### 结构与实现

由于 Java 提供了对象的 clone() 方法，所以用 Java 实现原型模式很简单。

##### 模式的结构

原型模式包含以下主要角色:

###### 抽象原型类

规定了具体原型对象必须实现的接口。

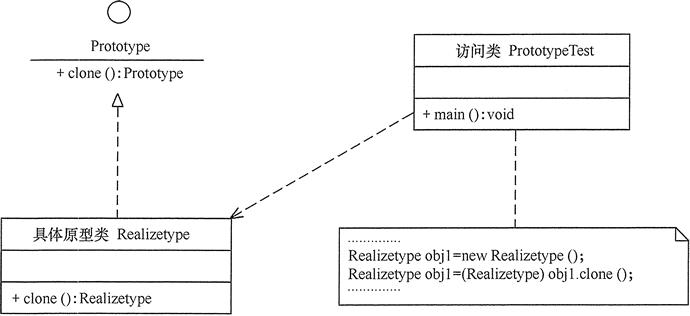
具体原型类

实现抽象原型类的 clone() 方法，它是可被复制的对象。

访问类

使用具体原型类中的 clone() 方法来复制新的对象。

其结构图如下图所示:



##### 模式的实现

原型模式的克隆分为浅克隆和深克隆，Java 中的 Object 类提供了浅克隆的 clone() 方法，具体原型类只要实现 Cloneable 接口就可实现对象的浅克隆，这里的 Cloneable 接口就是抽象原型类。其代码如下：

//具体原型类

class Realizetype implements Cloneable

{

Realizetype()

{

System.out.println("具体原型创建成功！");

}

public Object clone() throws CloneNotSupportedException

{

System.out.println("具体原型复制成功！");

return (Realizetype)super.clone();

}

}

//原型模式的测试类

public class PrototypeTest

{

public static void main(String[] args)throws CloneNotSupportedException

{

Realizetype obj1=new Realizetype();

Realizetype obj2=(Realizetype)obj1.clone();

System.out.println("obj1==obj2?"+(obj1==obj2));

}

}

程序的运行结果如下：

具体原型创建成功！

具体原型复制成功！

obj1==obj2?false

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1343.html

#### 应用场景

原型模式通常适用于以下场景：

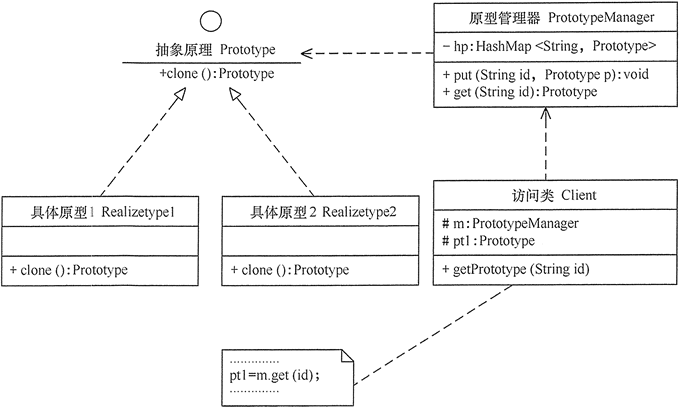
1.对象之间相同或相似，即只是个别的几个属性不同的时候

2.对象的创建过程比较麻烦，但复制比较简单的时候。

#### 扩展

原型模式可扩展为带原型管理器的原型模式，它在原型模式的基础上增加了一个原型管理器 PrototypeManager 类。

该类用 HashMap 保存多个复制的原型，Client 类可以通过管理器的 get(String id) 方法从中获取复制的原型。其结构图如下图所示：



### 工厂方法模式详解

#### 定义与特点

##### 定义

定义一个创建产品对象的工厂接口，将产品对象的实际创建工作推迟到具体子工厂类当中。这满足创建型模式中所要求的“创建与使用相分离”的特点。

我们把被创建的对象称为“产品”，把创建产品的对象称为“工厂”。如果要创建的产品不多，只要一个工厂类就可以完成，这种模式叫“简单工厂模式”，它不属于 GoF 的 23 种经典设计模式，它的缺点是增加新产品时会违背“开闭原则”。

本节介绍的“工厂方法模式”是对简单工厂模式的进一步抽象化，其好处是可以使系统在不修改原来代码的情况下引进新的产品，即满足开闭原则。

##### 主要优点

1.用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程

2.在系统增加新的产品时只需要添加具体产品类和对应的具体工厂类，无须对原工厂进行任何修改，满足开闭原则。

##### 缺点

每增加一个产品就要增加一个具体产品类和一个对应的具体工厂类，这增加了系统的复杂度。

#### 结构与实现

工厂方法模式由抽象工厂、具体工厂、抽象产品和具体产品等4个要素构成。

本节来分析其基本结构和实现方法。

##### 模式的结构

工厂方法模式的主要角色如下。

###### 抽象工厂（Abstract Factory）

提供了创建产品的接口，调用者通过它访问具体工厂的工厂方法 newProduct() 来创建产品。

###### 具体工厂（ConcreteFactory）

主要是实现抽象工厂中的抽象方法，完成具体产品的创建。

###### 抽象产品（Product）

定义了产品的规范，描述了产品的主要特性和功能。

###### 具体产品（ConcreteProduct）

实现了抽象产品角色所定义的接口，由具体工厂来创建，它同具体工厂之间一一对应。

其结构图如下图所示：



##### 模式的实现

根据上图写出该模式的代码如下：

package FactoryMethod;

public class AbstractFactoryTest

{

public static void main(String[] args)

{

try

{

Product a;

AbstractFactory af;

af=(AbstractFactory) ReadXML1.getObject();

a=af.newProduct();

a.show();

}

catch(Exception e)

{

System.out.println(e.getMessage());

}

}

}

//抽象产品：提供了产品的接口

interface Product

{

public void show();

}

//具体产品1：实现抽象产品中的抽象方法

class ConcreteProduct1 implements Product

{

public void show()

{

System.out.println("具体产品1显示...");

}

}

//具体产品2：实现抽象产品中的抽象方法

class ConcreteProduct2 implements Product

{

public void show()

{

System.out.println("具体产品2显示...");

}

}

//抽象工厂：提供了厂品的生成方法

interface AbstractFactory

{

public Product newProduct();

}

//具体工厂1：实现了厂品的生成方法

class ConcreteFactory1 implements AbstractFactory

{

public Product newProduct()

{

System.out.println("具体工厂1生成-->具体产品1...");

return new ConcreteProduct1();

}

}

//具体工厂2：实现了厂品的生成方法

class ConcreteFactory2 implements AbstractFactory

{

public Product newProduct()

{

System.out.println("具体工厂2生成-->具体产品2...");

return new ConcreteProduct2();

}

}

package FactoryMethod;

import javax.xml.parsers.\*;

import org.w3c.dom.\*;

import java.io.\*;

class ReadXML1

{

//该方法用于从XML配置文件中提取具体类类名，并返回一个实例对象

public static Object getObject()

{

try

{

//创建文档对象

DocumentBuilderFactory dFactory=DocumentBuilderFactory.newInstance();

DocumentBuilder builder=dFactory.newDocumentBuilder();

Document doc;

doc=builder.parse(new File("src/FactoryMethod/config1.xml"));

//获取包含类名的文本节点

NodeList nl=doc.getElementsByTagName("className");

Node classNode=nl.item(0).getFirstChild();

String cName="FactoryMethod."+classNode.getNodeValue();

//System.out.println("新类名："+cName);

//通过类名生成实例对象并将其返回

Class<?> c=Class.forName(cName);

Object obj=c.newInstance();

return obj;

}

catch(Exception e)

{

e.printStackTrace();

return null;

}

}

}

程序运行结果如下：

具体工厂1生成-->具体产品1...

具体产品1显示...

如果将XM配置文件中的ConcreteFactory1改为ConcreteFactory2，则程序运行结果如下：

具体工厂2生成-->具体产品2...

具体产品2显示...

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1348.html

#### 应用场景

工厂方法模式通常适用于以下场景：

1.客户只知道创建产品的工厂名，而不知道具体的产品名。如 TCL 电视工厂、海信电视工厂等

2.创建对象的任务由多个具体子工厂中的某一个完成，而抽象工厂只提供创建产品的接口

3.客户不关心创建产品的细节，只关心产品的品牌。

#### 扩展

当需要生成的产品不多且不会增加，一个具体工厂类就可以完成任务时，可删除抽象工厂类。

这时工厂方法模式将退化到简单工厂模式，其结构图如下图所示：

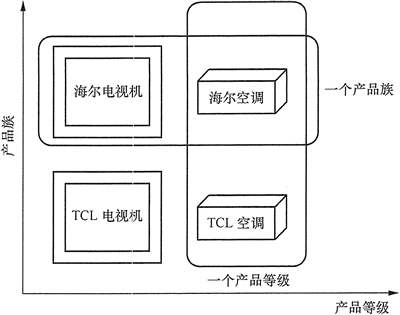


### 抽象工厂模式详解

前面介绍的工厂方法模式中考虑的是一类产品的生产，如畜牧场只养动物、电视机厂只生产电视机、计算机软件学院只培养计算机软件专业的学生等。

同种类称为同等级，也就是说：工厂方法模式只考虑生产同等级的产品，但是在现实生活中许多工厂是综合型的工厂，能生产多等级（种类） 的产品，如农场里既养动物又种植物，电器厂既生产电视机又生产洗衣机或空调，大学既有软件专业又有生物专业等。

本节要介绍的抽象工厂模式将考虑多等级产品的生产，将同一个具体工厂所生产的位于不同等级的一组产品称为一个产品族，下图所示的是海尔工厂和 TCL 工厂所生产的电视机与空调对应的关系图：



#### 定义与特点

##### 定义

一种为访问类提供一个创建一组相关或相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构。

抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。

使用抽象工厂模式一般要满足以下条件：

1.系统中有多个产品族，每个具体工厂创建同一族但属于不同等级结构的产品

2.系统一次只可能消费其中某一族产品，即同族的产品一起使用。

##### 主要优点

抽象工厂模式除了具有工厂方法模式的优点外，其他主要优点如下：

1.可以在类的内部对产品族中相关联的多等级产品共同管理，而不必专门引入多个新的类来进行管理

2.当增加一个新的产品族时不需要修改原代码，满足开闭原则。

##### 主要缺点

当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。

#### 结构与实现

抽象工厂模式同工厂方法模式一样，也是由抽象工厂、具体工厂、抽象产品和具体产品等 4 个要素构成，但抽象工厂中方法个数不同，抽象产品的个数也不同。

现在我们来分析其基本结构和实现方法。

##### 模式的结构

抽象工厂模式的主要角色如下:

###### 抽象工厂（Abstract Factory）

提供了创建产品的接口，它包含多个创建产品的方法 newProduct()，可以创建多个不同等级的产品。

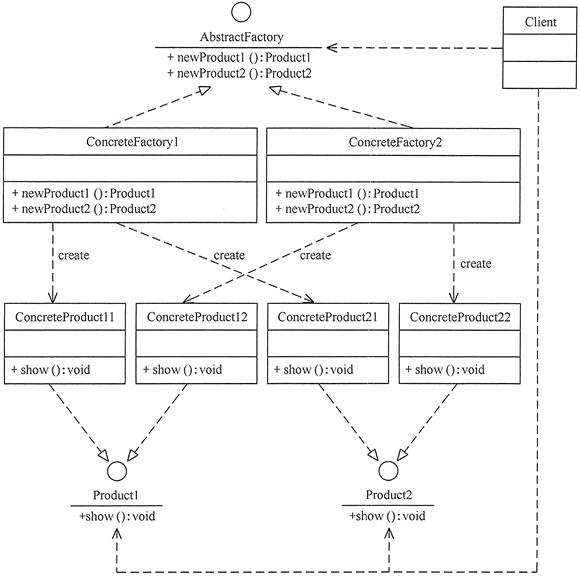
###### 具体工厂（Concrete Factory）

主要是实现抽象工厂中的多个抽象方法，完成具体产品的创建。

抽象产品（Product）：定义了产品的规范，描述了产品的主要特性和功能，抽象工厂模式有多个抽象产品。

###### 具体产品（ConcreteProduct）

实现了抽象产品角色所定义的接口，由具体工厂来创建，它 同具体工厂之间是多对一的关系。

抽象工厂模式的结构图下图所示：  


##### 模式的实现

从上图可以看出抽象工厂模式的结构同工厂方法模式的结构相似，不同的是其产品的种类不止一个，所以创建产品的方法也不止一个。

下面给出抽象工厂和具体工厂的代码：

1.抽象工厂：提供了产品的生成方法。

interface AbstractFactory

{

public Product1 newProduct1();

public Product2 newProduct2();

}

2. 具体工厂：实现了产品的生成方法。

class ConcreteFactory1 implements AbstractFactory

{

public Product1 newProduct1()

{

System.out.println("具体工厂 1 生成-->具体产品 11...");

return new ConcreteProduct11();

}

public Product2 newProduct2()

{

System.out.println("具体工厂 1 生成-->具体产品 21...");

return new ConcreteProduct21();

}

}

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1351.html

#### 应用场景

抽象工厂模式通常适用于以下场景：

1.当需要创建的对象是一系列相互关联或相互依赖的产品族时，如电器工厂中的电视机、洗衣机、空调等

2.系统中有多个产品族，但每次只使用其中的某一族产品。如有人只喜欢穿某一个品牌的衣服和鞋

3.系统中提供了产品的类库，且所有产品的接口相同，客户端不依赖产品实例的创建细节和内部结构。

#### 扩展

抽象工厂模式的扩展有一定的“开闭原则”倾斜性：

1.当增加一个新的产品族时只需增加一个新的具体工厂，不需要修改原代码，满足开闭原则

2.当产品族中需要增加一个新种类的产品时，则所有的工厂类都需要进行修改，不满足开闭原则。

另一方面，当系统中只存在一个等级结构的产品时，抽象工厂模式将退化到工厂方法模式。

### 建造者模式详解

在软件开发过程中有时需要创建一个复杂的对象，这个复杂对象通常由多个子部件按一定的步骤组合而成。例如，计算机是由CPU、主板、内存、硬盘、显卡、机箱、显示器、键盘、鼠标等部件组装而成的，采购员不可能自己去组装计算机，而是将计算机的配置要求告诉计算机销售公司，计算机销售公司安排技术人员去组装计算机，然后再交给要买计算机的采购员。

生活中这样的例子很多，如游戏中的不同角色，其性别、个性、能力、脸型、体型、服装、发型等特性都有所差异；还有汽车中的方向盘、发动机、车架、轮胎等部件也多种多样；每封电子邮件的发件人、收件人、主题、内容、附件等内容也各不相同。

以上所有这些产品都是由多个部件构成的，各个部件可以灵活选择，但其创建步骤都大同小异。这类产品的创建无法用前面介绍的工厂模式描述，只有建造者模式可以很好地描述该类产品的创建。

#### 定义与特点

##### 定义

将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表示，这样的设计模式被称为建造者模式。它是将一个复杂的对象分解为多个简单的对象，然后一步一步构建而成。它将变与不变相分离，即产品的组成部分是不变的，但每一部分是可以灵活选择的。

##### 主要优点

1.各个具体的建造者相互独立，有利于系统的扩展

2.客户端不必知道产品内部组成的细节，便于控制细节风险。

##### 缺点

1.产品的组成部分必须相同，这限制了其使用范围

2.如果产品的内部变化复杂，该模式会增加很多的建造者类。

建造者（Builder）模式和工厂模式的关注点不同：建造者模式注重零部件的组装过程，而工厂方法模式更注重零部件的创建过程，但两者可以结合使用。

#### 结构与实现

建造者（Builder）模式由产品、抽象建造者、具体建造者、指挥者等 4 个要素构成，现在我们来分析其基本结构和实现方法。

##### 模式的结构

建造者（Builder）模式的主要角色如下：

###### 产品角色（Product）

它是包含多个组成部件的复杂对象，由具体建造者来创建其各个零部件。

###### 抽象建造者（Builder）

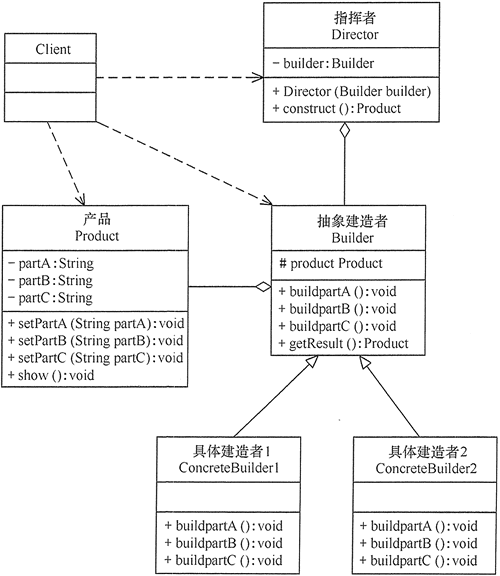
它是一个包含创建产品各个子部件的抽象方法的接口，通常还包含一个返回复杂产品的方法 getResult()。

###### 具体建造者(Concrete Builder）

实现 Builder 接口，完成复杂产品的各个部件的具体创建方法。

指挥者（Director）：它调用建造者对象中的部件构造与装配方法完成复杂对象的创建，在指挥者中不涉及具体产品的信息。

其结构图如下图所示：



##### 模式的实现

上图给出了建造者（Builder）模式的主要结构，其相关类的代码如下。

1.产品角色：包含多个组成部件的复杂对象。

class Product

{

private String partA;

private String partB;

private String partC;

public void setPartA(String partA)

{

this.partA=partA;

}

public void setPartB(String partB)

{

this.partB=partB;

}

public void setPartC(String partC)

{

this.partC=partC;

}

public void show()

{

//显示产品的特性

}

}

1. 抽象建造者：包含创建产品各个子部件的抽象方法。

abstract class Builder

{

//创建产品对象

protected Product product=new Product();

public abstract void buildPartA();

public abstract void buildPartB();

public abstract void buildPartC();

//返回产品对象

public Product getResult()

{

return product;

}

}

1. 具体建造者：实现了抽象建造者接口。

public class ConcreteBuilder extends Builder

{

public void buildPartA()

{

product.setPartA("建造 PartA");

}

public void buildPartB()

{

product.setPartA("建造 PartB");

}

public void buildPartC()

{

product.setPartA("建造 PartC");

}

}

1. 指挥者：调用建造者中的方法完成复杂对象的创建。

class Director

{

private Builder builder;

public Director(Builder builder)

{

this.builder=builder;

}

//产品构建与组装方法

public Product construct()

{

builder.buildPartA();

builder.buildPartB();

builder.buildPartC();

return builder.getResult();

}

}

1. 客户类。

public class Client

{

public static void main(String[] args)

{

Builder builder=new ConcreteBuilder();

Director director=new Director(builder);

Product product=director.construct();

product.show();

}

}

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1354.html

#### 应用场景

建造者（Builder）模式创建的是复杂对象，其产品的各个部分经常面临着剧烈的变化，但将它们组合在一起的算法却相对稳定，所以它通常在以下场合使用：

1.创建的对象较复杂，由多个部件构成，各部件面临着复杂的变化，但构件间的建造顺序是稳定的

2.创建复杂对象的算法独立于该对象的组成部分以及它们的装配方式，即产品的构建过程和最终的表示是独立的。

#### 扩展

建造者（Builder）模式在应用过程中可以根据需要改变，如果创建的产品种类只有一种，只需要一个具体建造者，这时可以省略掉抽象建造者，甚至可以省略掉指挥者角色。

### 代理模式详解

在有些情况下，一个客户不能或者不想直接访问另一个对象，这时需要找一个中介帮忙完成某项任务，这个中介就是代理对象。例如，购买火车票不一定要去火车站买，可以通过 12306 网站或者去火车票代售点买。又如找女朋友、找保姆、找工作等都可以通过找中介完成。

在软件设计中，使用代理模式的例子也很多，例如，要访问的远程对象比较大（如视频或大图像等），其下载要花很多时间。还有因为安全原因需要屏蔽客户端直接访问真实对象，如某单位的内部数据库等。

#### 定义与特点

##### 定义

由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。

##### 主要优点

1.代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用

2.代理对象可以扩展目标对象的功能

3.代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度。

##### 主要缺点

1.在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢

2.增加了系统的复杂度。

#### 结构与实现

代理模式的结构比较简单，主要是通过定义一个继承抽象主题的代理来包含真实主题，从而实现对真实主题的访问，下面来分析其基本结构和实现方法。

##### 模式的结构

代理模式的主要角色如下：

###### 抽象主题（Subject）类

通过接口或抽象类声明真实主题和代理对象实现的业务方法。

###### 真实主题（Real Subject）类

实现了抽象主题中的具体业务，是代理对象所代表的真实对象，是最终要引用的对象。

###### 代理（Proxy）类

提供了与真实主题相同的接口，其内部含有对真实主题的引用，它可以访问、控制或扩展真实主题的功能。

其结构图如下图所示：



##### 模式的实现

代理模式的实现代码如下：

package proxy;

public class ProxyTest

{

public static void main(String[] args)

{

Proxy proxy=new Proxy();

proxy.Request();

}

}

//抽象主题

interface Subject

{

void Request();

}

//真实主题

class RealSubject implements Subject

{

public void Request()

{

System.out.println("访问真实主题方法...");

}

}

//代理

class Proxy implements Subject

{

private RealSubject realSubject;

public void Request()

{

if (realSubject==null)

{

realSubject=new RealSubject();

}

preRequest();

realSubject.Request();

postRequest();

}

public void preRequest()

{

System.out.println("访问真实主题之前的预处理。");

}

public void postRequest()

{

System.out.println("访问真实主题之后的后续处理。");

}

}

程序运行的结果如下：

访问真实主题之前的预处理。

访问真实主题方法...

访问真实主题之后的后续处理。

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1359.html

#### 应用场景

前面分析了代理模式的结构与特点，现在来分析以下的应用场景：

1.远程代理，这种方式通常是为了隐藏目标对象存在于不同地址空间的事实，方便客户端访问。例如，用户申请某些网盘空间时，会在用户的文件系统中建立一个虚拟的硬盘，用户访问虚拟硬盘时实际访问的是网盘空间。

2.虚拟代理，这种方式通常用于要创建的目标对象开销很大时。例如，下载一幅很大的图像需要很长时间，因某种计算比较复杂而短时间无法完成，这时可以先用小比例的虚拟代理替换真实的对象，消除用户对服务器慢的感觉。

3.安全代理，这种方式通常用于控制不同种类客户对真实对象的访问权限。

4.智能指引，主要用于调用目标对象时，代理附加一些额外的处理功能。例如，增加计算真实对象的引用次数的功能，这样当该对象没有被引用时，就可以自动释放它。

5.延迟加载，指为了提高系统的性能，延迟对目标的加载。例如，Hibernate 中就存在属性的延迟加载和关联表的延时加载。

#### 扩展

在前面介绍的代理模式中，代理类中包含了对真实主题的引用，这种方式存在两个缺点。

1.真实主题与代理主题一一对应，增加真实主题也要增加代理

2.设计代理以前真实主题必须事先存在，不太灵活。采用动态代理模式可以解决以上问题，如 SpringAOP，其结构图如下图所示：



### 适配器模式详解

在现实生活中，经常出现两个对象因接口不兼容而不能在一起工作的实例，这时需要第三者进行适配。

例如，讲中文的人同讲英文的人对话时需要一个翻译，用直流电的笔记本电脑接交流电源时需要一个电源适配器，用计算机访问照相机的 SD 内存卡时需要一个读卡器等。

在软件设计中也可能出现：需要开发的具有某种业务功能的组件在现有的组件库中已经存在，但它们与当前系统的接口规范不兼容，如果重新开发这些组件成本又很高，这时用适配器模式能很好地解决这些问题。

#### 定义与特点

##### 定义

将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

适配器模式分为类结构型模式和对象结构型模式两种，前者类之间的耦合度比后者高，且要求程序员了解现有组件库中的相关组件的内部结构，所以应用相对较少些。

##### 主要优点

1.客户端通过适配器可以透明地调用目标接口

2.复用了现存的类，程序员不需要修改原有代码而重用现有的适配者类

3.将目标类和适配者类解耦，解决了目标类和适配者类接口不一致的问题。

##### 缺点

对类适配器来说，更换适配器的实现过程比较复杂。

#### 结构与实现

对类适配器来说，更换适配器的实现过程比较复杂。类适配器模式可采用多重继承方式实现，如 C++ 可定义一个适配器类来同时继承当前系统的业务接口和现有组件库中已经存在的组件接口；Java 不支持多继承，但可以定义一个适配器类来实现当前系统的业务接口，同时又继承现有组件库中已经存在的组件。

对象适配器模式可釆用将现有组件库中已经实现的组件引入适配器类中，该类同时实现当前系统的业务接口。

现在来介绍它们的基本结构。

##### 模式的结构

适配器模式（Adapter）包含以下主要角色：

1.目标（Target）接口

当前系统业务所期待的接口，它可以是抽象类或接口。

1. 适配者（Adaptee）类

它是被访问和适配的现存组件库中的组件接口。

3.适配器（Adapter）类

它是一个转换器，通过继承或引用适配者的对象，把适配者接口转换成目标接口，让客户按目标接口的格式访问适配者。

类适配器模式的结构图如下图所示：



对象适配器模式的结构图如下图所示：



##### 模式的实现

类适配器模式的代码如下：

package adapter;

//目标接口

interface Target

{

public void request();

}

//适配者接口

class Adaptee

{

public void specificRequest()

{

System.out.println("适配者中的业务代码被调用！");

}

}

//类适配器类

class ClassAdapter extends Adaptee implements Target

{

public void request()

{

specificRequest();

}

}

//客户端代码

public class ClassAdapterTest

{

public static void main(String[] args)

{

System.out.println("类适配器模式测试：");

Target target = new ClassAdapter();

target.request();

}

}

程序的运行结果如下：

类适配器模式测试：

适配者中的业务代码被调用！

对象适配器模式的代码如下：

package adapter;

//对象适配器类

class ObjectAdapter implements Target

{

private Adaptee adaptee;

public ObjectAdapter(Adaptee adaptee)

{

this.adaptee=adaptee;

}

public void request()

{

adaptee.specificRequest();

}

}

//客户端代码

public class ObjectAdapterTest

{

public static void main(String[] args)

{

System.out.println("对象适配器模式测试：");

Adaptee adaptee = new Adaptee();

Target target = new ObjectAdapter(adaptee);

target.request();

}

}

程序的运行结果如下：

对象适配器模式测试：

适配者中的业务代码被调用！

特别说明：对象适配器模式中的“目标接口”和“适配者类”的代码同类适配器模式一样，只要修改适配器类和客户端的代码即可。

#### 应用实例

详细应用实例请到原文链接查看：<http://c.biancheng.net/view/1361.html>

#### 应用场景

适配器模式（Adapter）通常适用于以下场景：

1.以前开发的系统存在满足新系统功能需求的类，但其接口同新系统的接口不一致

2.使用第三方提供的组件，但组件接口定义和自己要求的接口定义不同。

#### 扩展

适配器模式（Adapter）可扩展为双向适配器模式，双向适配器类既可以把适配者接口转换成目标接口，也可以把目标接口转换成适配者接口，其结构图如下图所示：



程序代码如下：

package adapter;

//目标接口

interface TwoWayTarget

{

public void request();

}

//适配者接口

interface TwoWayAdaptee

{

public void specificRequest();

}

//目标实现

class TargetRealize implements TwoWayTarget

{

public void request()

{

System.out.println("目标代码被调用！");

}

}

//适配者实现

class AdapteeRealize implements TwoWayAdaptee

{

public void specificRequest()

{

System.out.println("适配者代码被调用！");

}

}

//双向适配器

class TwoWayAdapter implements TwoWayTarget,TwoWayAdaptee

{

private TwoWayTarget target;

private TwoWayAdaptee adaptee;

public TwoWayAdapter(TwoWayTarget target)

{

this.target=target;

}

public TwoWayAdapter(TwoWayAdaptee adaptee)

{

this.adaptee=adaptee;

}

public void request()

{

adaptee.specificRequest();

}

public void specificRequest()

{

target.request();

}

}

//客户端代码

public class TwoWayAdapterTest

{

public static void main(String[] args)

{

System.out.println("目标通过双向适配器访问适配者：");

TwoWayAdaptee adaptee=new AdapteeRealize();

TwoWayTarget target=new TwoWayAdapter(adaptee);

target.request();

System.out.println("-------------------");

System.out.println("适配者通过双向适配器访问目标：");

target=new TargetRealize();

adaptee=new TwoWayAdapter(target);

adaptee.specificRequest();

}

}

程序的运行结果如下：

目标通过双向适配器访问适配者：

适配者代码被调用！

-------------------

适配者通过双向适配器访问目标：

目标代码被调用！

### 桥接模式详解

在现实生活中，某些类具有两个或多个维度的变化，如图形既可按形状分，又可按颜色分。如何设计类似于 Photoshop 这样的软件，能画不同形状和不同颜色的图形呢？如果用继承方式，m 种形状和 n 种颜色的图形就有 m×n 种，不但对应的子类很多，而且扩展困难。

当然，这样的例子还有很多，如不同颜色和字体的文字、不同品牌和功率的汽车、不同性别和职业的男女、支持不同平台和不同文件格式的媒体播放器等。如果用桥接模式就能很好地解决这些问题。

#### 定义与特点

##### 定义

将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。

##### 优点

1.由于抽象与实现分离，所以扩展能力强

2.其实现细节对客户透明。

##### 缺点

由于聚合关系建立在抽象层，要求开发者针对抽象化进行设计与编程，这增加了系统的理解与设计难度。

#### 结构与实现

可以将抽象化部分与实现化部分分开，取消二者的继承关系，改用组合关系。

##### 模式的结构

桥接（Bridge）模式包含以下主要角色：

###### 抽象化（Abstraction）角色

定义抽象类，并包含一个对实现化对象的引用。

###### 扩展抽象化（Refined Abstraction）角色

是抽象化角色的子类，实现父类中的业务方法，并通过组合关系调用实现化角色中的业务方法。

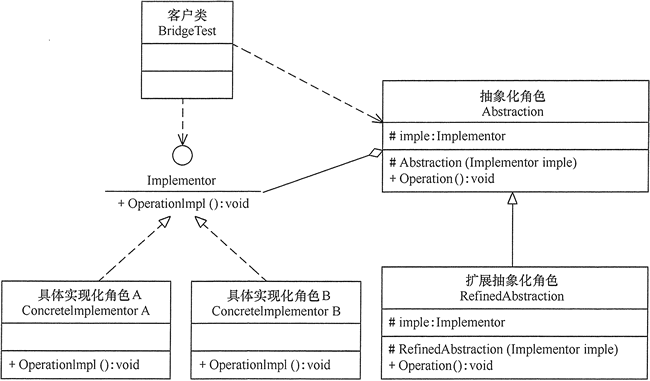
###### 实现化（Implementor）角色

定义实现化角色的接口，供扩展抽象化角色调用。

###### 具体实现化（Concrete Implementor）角色

给出实现化角色接口的具体实现。

其结构图如下图所示：



##### 模式的实现

桥接模式的代码如下：

package bridge;

public class BridgeTest

{

public static void main(String[] args)

{

Implementor imple=new ConcreteImplementorA();

Abstraction abs=new RefinedAbstraction(imple);

abs.Operation();

}

}

//实现化角色

interface Implementor

{

public void OperationImpl();

}

//具体实现化角色

class ConcreteImplementorA implements Implementor

{

public void OperationImpl()

{

System.out.println("具体实现化(Concrete Implementor)角色被访问" );

}

}

//抽象化角色

abstract class Abstraction

{

protected Implementor imple;

protected Abstraction(Implementor imple)

{

this.imple=imple;

}

public abstract void Operation();

}

//扩展抽象化角色

class RefinedAbstraction extends Abstraction

{

protected RefinedAbstraction(Implementor imple)

{

super(imple);

}

public void Operation()

{

System.out.println("扩展抽象化(Refined Abstraction)角色被访问" );

imple.OperationImpl();

}

}

程序的运行结果如下：

扩展抽象化(Refined Abstraction)角色被访问

具体实现化(Concrete Implementor)角色被访问

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1364.html

#### 应用场景

桥接模式通常适用于以下场景：

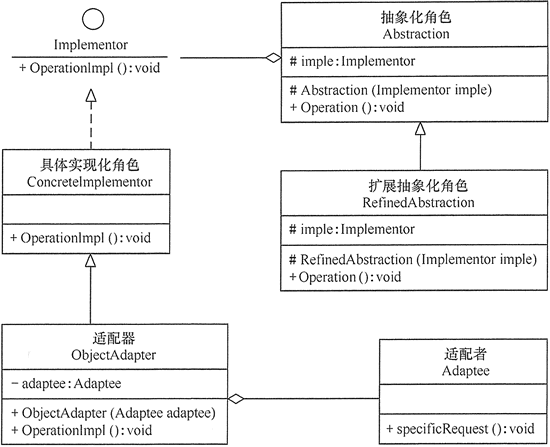
1.当一个类存在两个独立变化的维度，且这两个维度都需要进行扩展时

2.当一个系统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时

3.当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时。

#### 扩展

在软件开发中，有时桥接（Bridge）模式可与适配器模式联合使用。当桥接（Bridge）模式的实现化角色的接口与现有类的接口不一致时，可以在二者中间定义一个适配器将二者连接起来，其具体结构图如下图所示：



### 装饰模式详解

在现实生活中，常常需要对现有产品增加新的功能或美化其外观，如房子装修、相片加相框等。

在软件开发过程中，有时想用一些现存的组件。这些组件可能只是完成了一些核心功能。但在不改变其结构的情况下，可以动态地扩展其功能。

所有这些都可以釆用装饰模式来实现。.

#### 定义与特点

##### 定义

在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即增加其额外功能）的模式，它属于对象结构型模式。

##### 主要优点

1.采用装饰模式扩展对象的功能比采用继承方式更加灵活

2.可以设计出多个不同的具体装饰类，创造出多个不同行为的组合。

##### 主要缺点

装饰模式增加了许多子类，如果过度使用会使程序变得很复杂。

#### 结构与实现

通常情况下，扩展一个类的功能会使用继承方式来实现。

但继承具有静态特征，耦合度高，并且随着扩展功能的增多，子类会很膨胀。

如果使用组合关系来创建一个包装对象（即装饰对象）来包裹真实对象，并在保持真实对象的类结构不变的前提下，为其提供额外的功能，这就是装饰模式的目标。下面来分析其基本结构和实现方法。

##### 模式的结构

装饰模式主要包含以下角色

1.抽象构件（Component）角色

定义一个抽象接口以规范准备接收附加责任的对象。

2.具体构件（Concrete Component）角色

实现抽象构件，通过装饰角色为其添加一些职责。

3.抽象装饰（Decorator）角色

继承抽象构件，并包含具体构件的实例，可以通过其子类扩展具体构件的功能。

4.具体装饰（ConcreteDecorator）角色

实现抽象装饰的相关方法，并给具体构件对象添加附加的责任。

装饰模式的结构图下图 所示：



##### 模式的实现

装饰模式的实现代码如下：

package decorator;

public class DecoratorPattern

{

public static void main(String[] args)

{

Component p=new ConcreteComponent();

p.operation();

System.out.println("---------------------------------");

Component d=new ConcreteDecorator(p);

d.operation();

}

}

//抽象构件角色

interface Component

{

public void operation();

}

//具体构件角色

class ConcreteComponent implements Component

{

public ConcreteComponent()

{

System.out.println("创建具体构件角色");

}

public void operation()

{

System.out.println("调用具体构件角色的方法operation()");

}

}

//抽象装饰角色

class Decorator implements Component

{

private Component component;

public Decorator(Component component)

{

this.component=component;

}

public void operation()

{

component.operation();

}

}

//具体装饰角色

class ConcreteDecorator extends Decorator

{

public ConcreteDecorator(Component component)

{

super(component);

}

public void operation()

{

super.operation();

addedFunction();

}

public void addedFunction()

{

System.out.println("为具体构件角色增加额外的功能addedFunction()");

}

}

程序运行结果如下：

创建具体构件角色

调用具体构件角色的方法operation()

---------------------------------

调用具体构件角色的方法operation()

为具体构件角色增加额外的功能addedFunction()

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1366.html

#### 应用场景

前面讲解了关于装饰模式的结构与特点，下面介绍其适用的应用场景，装饰模式通常在以下几种情况使用：

1.当需要给一个现有类添加附加职责，而又不能采用生成子类的方法进行扩充时。例如，该类被隐藏或者该类是终极类或者采用继承方式会产生大量的子类

2.当需要通过对现有的一组基本功能进行排列组合而产生非常多的功能时，采用继承关系很难实现，而采用装饰模式却很好实现

3.当对象的功能要求可以动态地添加，也可以再动态地撤销时。

装饰模式在 Java 语言中的最著名的应用莫过于 Java I/O 标准库的设计了。

例如，InputStream 的子类 FilterInputStream，OutputStream 的子类 FilterOutputStream，Reader 的子类 BufferedReader 以及 FilterReader，还有 Writer 的子类 BufferedWriter、FilterWriter 以及 PrintWriter 等，它们都是抽象装饰类。

下面代码是为 FileReader 增加缓冲区而采用的装饰类 BufferedReader 的例子：

BufferedReader in=new BufferedReader(new FileReader("filename.txtn));

String s=in.readLine();

#### 扩展

装饰模式所包含的 4 个角色不是任何时候都要存在的，在有些应用环境下模式是可以简化的，如以下两种情况：

1.如果只有一个具体构件而没有抽象构件时，可以让抽象装饰继承具体构件，其结构图如下图所示：



2.如果只有一个具体装饰时，可以将抽象装饰和具体装饰合并，其结构图如下图所示：

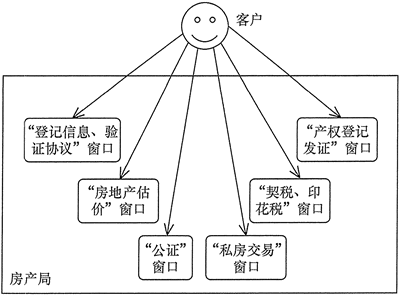


### 外观模式详解

在现实生活中，常常存在办事较复杂的例子，如办房产证或注册一家公司，有时要同多个部门联系，这时要是有一个综合部门能解决一切手续问题就好了。

软件设计也是这样，当一个系统的功能越来越强，子系统会越来越多，客户对系统的访问也变得越来越复杂。这时如果系统内部发生改变，客户端也要跟着改变，这违背了“开闭原则”，也违背了“迪米特法则”，所以有必要为多个子系统提供一个统一的接口，从而降低系统的耦合度，这就是外观模式的目标。

下图给出了客户去当地房产局办理房产证过户要遇到的相关部门：



#### 定义与特点

##### 定义

一种通过为多个复杂的子系统提供一个一致的接口，而使这些子系统更加容易被访问的模式。

该模式对外有一个统一接口，外部应用程序不用关心内部子系统的具体的细节，这样会大大降低应用程序的复杂度，提高了程序的可维护性。

##### 主要优点

1.降低了子系统与客户端之间的耦合度，使得子系统的变化不会影响调用它的客户类

2.对客户屏蔽了子系统组件，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

3.降低了大型软件系统中的编译依赖性，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不会影响其他的子系统，也不会影响外观对象。

##### 主要缺点

1.不能很好地限制客户使用子系统类

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”。

#### 结构与实现

外观（Facade）模式的结构比较简单，主要是定义了一个高层接口。

它包含了对各个子系统的引用，客户端可以通过它访问各个子系统的功能。现在来分析其基本结构和实现方法。

##### 模式的结构

外观（Facade）模式包含以下主要角色：

###### 外观（Facade）角色

为多个子系统对外提供一个共同的接口。

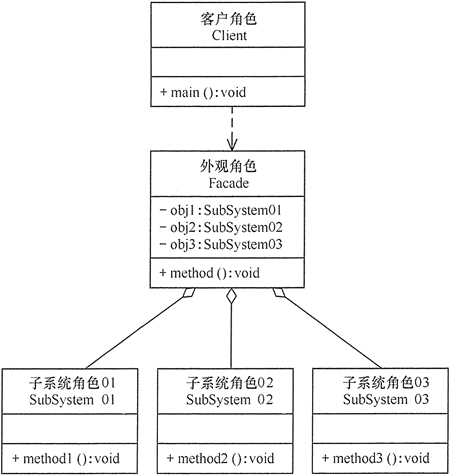
###### 子系统（Sub System）角色

实现系统的部分功能，客户可以通过外观角色访问它。

###### 客户（Client）角色

通过一个外观角色访问各个子系统的功能。

其结构图如下图所示：



##### 模式的实现

外观模式的实现代码如下：

package facade;

public class FacadePattern

{

public static void main(String[] args)

{

Facade f=new Facade();

f.method();

}

}

//外观角色

class Facade

{

private SubSystem01 obj1=new SubSystem01();

private SubSystem02 obj2=new SubSystem02();

private SubSystem03 obj3=new SubSystem03();

public void method()

{

obj1.method1();

obj2.method2();

obj3.method3();

}

}

//子系统角色

class SubSystem01

{

public void method1()

{

System.out.println("子系统01的method1()被调用！");

}

}

//子系统角色

class SubSystem02

{

public void method2()

{

System.out.println("子系统02的method2()被调用！");

}

}

//子系统角色

class SubSystem03

{

public void method3()

{

System.out.println("子系统03的method3()被调用！");

}

}

程序运行结果如下：

子系统01的method1()被调用！

子系统02的method2()被调用！

子系统03的method3()被调用！

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1369.html

#### 应用场景

通常在以下情况下可以考虑使用外观模式：

1.对分层结构系统构建时，使用外观模式定义子系统中每层的入口点可以简化子系统之间的依赖关系

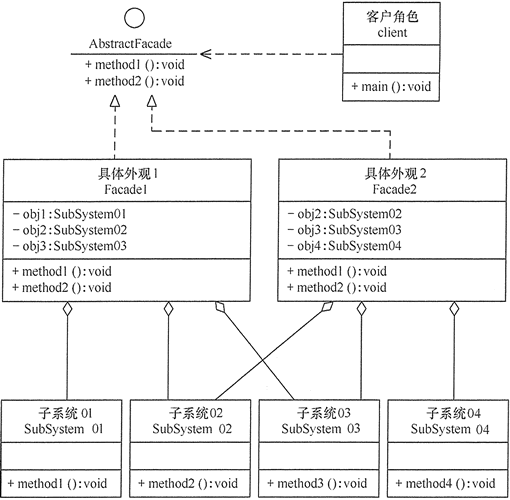
2.当一个复杂系统的子系统很多时，外观模式可以为系统设计一个简单的接口供外界访问

3.当客户端与多个子系统之间存在很大的联系时，引入外观模式可将它们分离，从而提高子系统的独立性和可移植性。

#### 扩展

在外观模式中，当增加或移除子系统时需要修改外观类，这违背了“开闭原则”。

如果引入抽象外观类，则在一定程度上解决了该问题，其结构图如下图所示：



### 享元模式详解

在面向对象程序设计过程中，有时会面临要创建大量相同或相似对象实例的问题。创建那么多的对象将会耗费很多的系统资源，它是系统性能提高的一个瓶颈。

例如，围棋和五子棋中的黑白棋子，图像中的坐标点或颜色，局域网中的路由器、交换机和集线器，教室里的桌子和凳子等。

这些对象有很多相似的地方，如果能把它们相同的部分提取出来共享，则能节省大量的系统资源，这就是享元模式的产生背景。

#### 定义与特点

##### 定义

运用共享技术来有効地支持大量细粒度对象的复用。它通过共享已经存在的又橡来大幅度减少需要创建的对象数量、避免大量相似类的开销，从而提高系统资源的利用率。

##### 主要优点

相同对象只要保存一份，这降低了系统中对象的数量，从而降低了系统中细粒度对象给内存带来的压力。

##### 主要缺点

1.为了使对象可以共享，需要将一些不能共享的状态外部化，这将增加程序的复杂性

2.读取享元模式的外部状态会使得运行时间稍微变长。

#### 结构与实现

享元模式中存在以下两种状态：

1.内部状态，即不会随着环境的改变而改变的可共享部分

2.外部状态，指随环境改变而改变的不可以共享的部分。享元模式的实现要领就是区分应用中的这两种状态，并将外部状态外部化。下面来分析其基本结构和实现方法。

##### 模式的结构

享元模式的主要角色有如下：

###### 抽象享元角色（Flyweight）

所有的具体享元类的基类，为具体享元规范需要实现的公共接口，非享元的外部状态以参数的形式通过方法传入。

###### 具体享元（Concrete Flyweight）角色

实现抽象享元角色中所规定的接口。

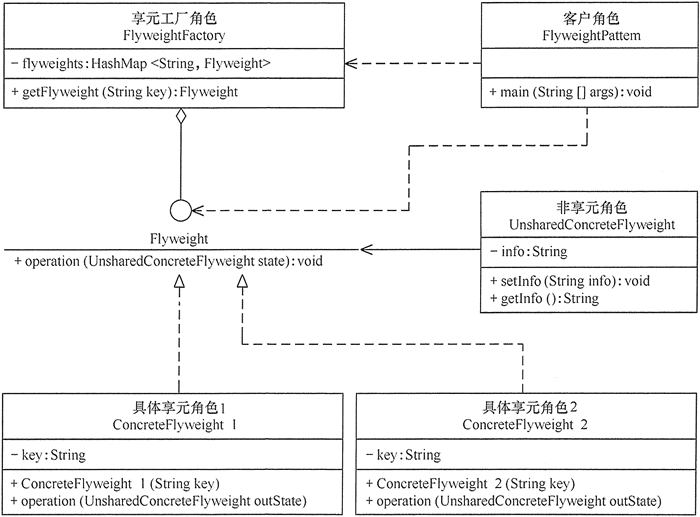
###### 非享元（Unsharable Flyweight)角色

不可以共享的外部状态，它以参数的形式注入具体享元的相关方法中。

###### 享元工厂（Flyweight Factory）角色

负责创建和管理享元角色。当客户对象请求一个享元对象时，享元工厂检査系统中是否存在符合要求的享元对象，如果存在则提供给客户；如果不存在的话，则创建一个新的享元对象。

下图是享元模式的结构图：



图中的UnsharedConcreteFlyweight是与淳元角色，里面包含了非共享的外部状态信息info；而Flyweight是抽象享元角色，里面包含了享元方法operation(UnsharedConcreteFlyweight state)，非享元的外部状态以参数的形式通过该方法传入；ConcreteFlyweight 是具体享元角色，包含了关键字 key，它实现了抽象享元接口；FlyweightFactory 是享元工厂角色，它逝关键字 key 来管理具体享元；客户角色通过享元工厂获取具体享元，并访问具体享元的相关方法。

##### 模式的实现

享元模式的实现代码如下：

package flyweight;

import java.util.HashMap;

public class FlyweightPattern

{

public static void main(String[] args)

{

FlyweightFactory factory=new FlyweightFactory();

Flyweight f01=factory.getFlyweight("a");

Flyweight f02=factory.getFlyweight("a");

Flyweight f03=factory.getFlyweight("a");

Flyweight f11=factory.getFlyweight("b");

Flyweight f12=factory.getFlyweight("b");

f01.operation(new UnsharedConcreteFlyweight("第1次调用a。"));

f02.operation(new UnsharedConcreteFlyweight("第2次调用a。"));

f03.operation(new UnsharedConcreteFlyweight("第3次调用a。"));

f11.operation(new UnsharedConcreteFlyweight("第1次调用b。"));

f12.operation(new UnsharedConcreteFlyweight("第2次调用b。"));

}

}

//非享元角色

class UnsharedConcreteFlyweight

{

private String info;

UnsharedConcreteFlyweight(String info)

{

this.info=info;

}

public String getInfo()

{

return info;

}

public void setInfo(String info)

{

this.info=info;

}

}

//抽象享元角色

interface Flyweight

{

public void operation(UnsharedConcreteFlyweight state);

}

//具体享元角色

class ConcreteFlyweight implements Flyweight

{

private String key;

ConcreteFlyweight(String key)

{

this.key=key;

System.out.println("具体享元"+key+"被创建！");

}

public void operation(UnsharedConcreteFlyweight outState)

{

System.out.print("具体享元"+key+"被调用，");

System.out.println("非享元信息是:"+outState.getInfo());

}

}

//享元工厂角色

class FlyweightFactory

{

private HashMap<String, Flyweight> flyweights=new HashMap<String, Flyweight>();

public Flyweight getFlyweight(String key)

{

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

if(flyweight!=null)

{

System.out.println("具体享元"+key+"已经存在，被成功获取！");

}

else

{

flyweight=new ConcreteFlyweight(key);

flyweights.put(key, flyweight);

}

return flyweight;

}

}

程序运行结果如下：

具体享元a被创建！

具体享元a已经存在，被成功获取！

具体享元a已经存在，被成功获取！

具体享元b被创建！

具体享元b已经存在，被成功获取！

具体享元a被调用，非享元信息是:第1次调用a。

具体享元a被调用，非享元信息是:第2次调用a。

具体享元a被调用，非享元信息是:第3次调用a。

具体享元b被调用，非享元信息是:第1次调用b。

具体享元b被调用，非享元信息是:第2次调用b。

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1371.html

#### 应用场景

享元模式是通过减少内存中对象的数量来节省内存空间的，所以以下几种情形适合采用享元模式：

1.系统中存在大量相同或相似的对象，这些对象耗费大量的内存资源

2.大部分的对象可以按照内部状态进行分组，且可将不同部分外部化，这样每一个组只需保存一个内部状态

3.由于享元模式需要额外维护一个保存享元的数据结构，所以应当在有足够多的享元实例时才值得使用享元模式。

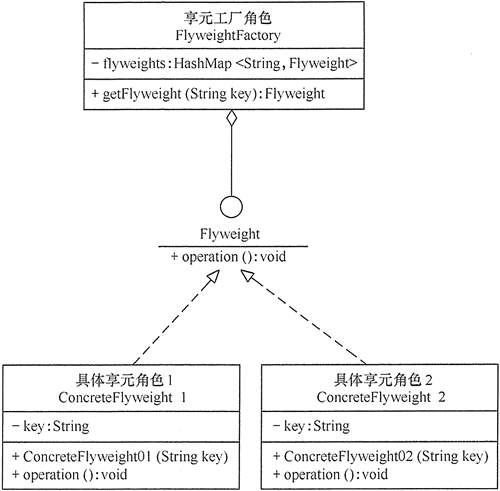
#### 扩展

在前面介绍的享元模式中，其结构图通常包含可以共享的部分和不可以共享的部分。

在实际使用过程中，有时候会稍加改变，即存在两种特殊的享元模式：单纯享元模式和复合享元模式，下面分别对它们进行简单介绍。

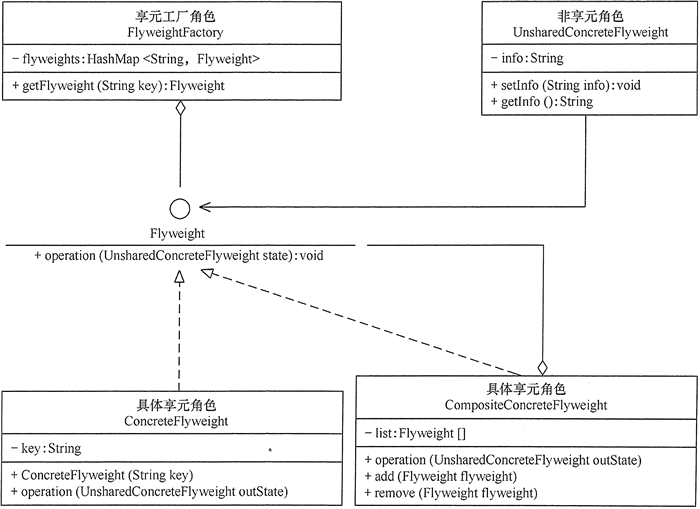
##### 单纯享元模式

这种享元模式中的所有的具体享元类都是可以共享的，不存在非共享的具体享元类，其结构图如下图所示：



##### 复合享元模式

这种享元模式中的有些享元对象是由一些单纯享元对象组合而成的，它们就是复合享元对象。虽然复合享元对象本身不能共享，但它们可以分解成单纯享元对象再被共享，其结构图如下图所示：



### 组合模式详解

在现实生活中，存在很多“部分-整体”的关系，例如，大学中的部门与学院、总公司中的部门与分公司、学习用品中的书与书包、厨房中的锅碗瓢盆等。

在软件开发中也是这样，例如，文件系统中的文件与文件夹、窗体程序中的简单控件与容器控件等。对这些简单对象与复合对象的处理，如果用组合模式来实现会很方便。

#### 定义与特点

##### 定义

有时又叫作部分-整体模式，它是一种将对象组合成树状的层次结构的模式，用来表示“部分-整体”的关系，使用户对单个对象和组合对象具有一致的访问性。

##### 主要优点

1.组合模式使得客户端代码可以一致地处理单个对象和组合对象，无须关心自己处理的是单个对象，还是组合对象，这简化了客户端代码

2.更容易在组合体内加入新的对象，客户端不会因为加入了新的对象而更改源代码，满足“开闭原则”。

##### 主要缺点

1.设计较复杂，客户端需要花更多时间理清类之间的层次关系

2.不容易限制容器中的构件

3.不容易用继承的方法来增加构件的新功能。

#### 结构与实现

组合模式的结构不是很复杂，下面对它的结构和实现进行分析。

##### 模式的结构

组合模式包含以下主要角色：

###### 抽象构件（Component）角色

它的主要作用是为树叶构件和树枝构件声明公共接口，并实现它们的默认行为。

在透明式的组合模式中抽象构件还声明访问和管理子类的接口；在安全式的组合模式中不声明访问和管理子类的接口，管理工作由树枝构件完成。

###### 树叶构件（Leaf）角色

是组合中的叶节点对象，它没有子节点，用于实现抽象构件角色中 声明的公共接口。

###### 树枝构件（Composite）角色

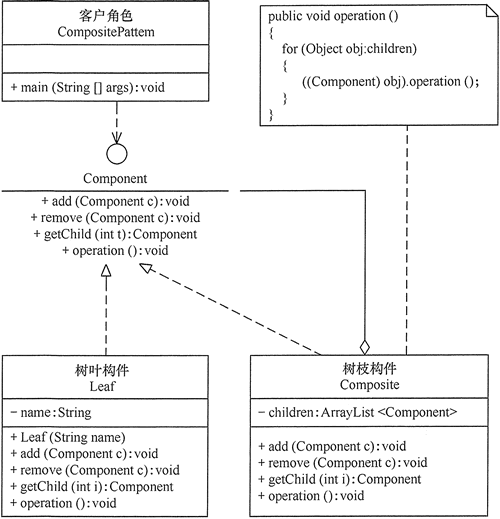
组合中的分支节点对象，它有子节点。它实现了抽象构件角色中声明的接口，它的主要作用是存储和管理子部件，通常包含 Add()、Remove()、GetChild() 等方法。

组合模式分为透明式的组合模式和安全式的组合模式。

###### 透明方式

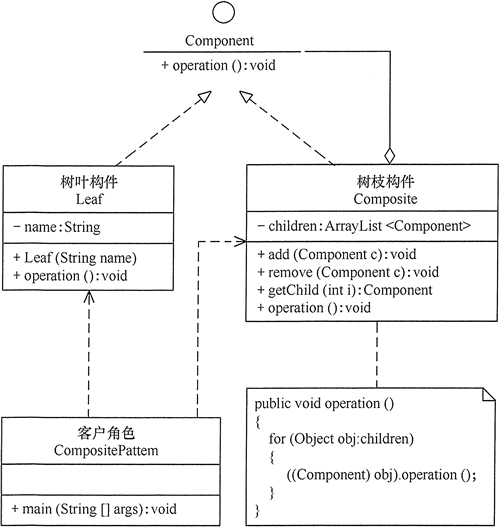
在该方式中，由于抽象构件声明了所有子类中的全部方法，所以客户端无须区别树叶对象和树枝对象，对客户端来说是透明的。

但其缺点是：树叶构件本来没有 Add()、Remove() 及 GetChild() 方法，却要实现它们（空实现或抛异常），这样会带来一些安全性问题。其结构图如下图所示：



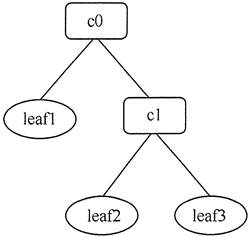
###### 安全方式

在该方式中，将管理子构件的方法移到树枝构件中，抽象构件和树叶构件没有对子对象的管理方法，这样就避免了上一种方式的安全性问题，但由于叶子和分支有不同的接口，客户端在调用时要知道树叶对象和树枝对象的存在，所以失去了透明性。其结构图如下图所示：



##### 模式的实现

假如要访问集合 c0={leaf1,{leaf2,leaf3}} 中的元素，其对应的树状图如下图所示：



下面给出透明式的组合模式的实现代码，与安全式的组合模式的实现代码类似，只要对其做简单修改就可以了。

package composite;

import java.util.ArrayList;

public class CompositePattern

{

public static void main(String[] args)

{

Component c0=new Composite();

Component c1=new Composite();

Component leaf1=new Leaf("1");

Component leaf2=new Leaf("2");

Component leaf3=new Leaf("3");

c0.add(leaf1);

c0.add(c1);

c1.add(leaf2);

c1.add(leaf3);

c0.operation();

}

}

//抽象构件

interface Component

{

public void add(Component c);

public void remove(Component c);

public Component getChild(int i);

public void operation();

}

//树叶构件

class Leaf implements Component

{

private String name;

public Leaf(String name)

{

this.name=name;

}

public void add(Component c){ }

public void remove(Component c){ }

public Component getChild(int i)

{

return null;

}

public void operation()

{

System.out.println("树叶"+name+"：被访问！");

}

}

//树枝构件

class Composite implements Component

{

private ArrayList<Component> children=new ArrayList<Component>();

public void add(Component c)

{

children.add(c);

}

public void remove(Component c)

{

children.remove(c);

}

public Component getChild(int i)

{

return children.get(i);

}

public void operation()

{

for(Object obj:children)

{

((Component)obj).operation();

}

}

}

程序运行结果如下：

树叶1：被访问！

树叶2：被访问！

树叶3：被访问！

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1373.html

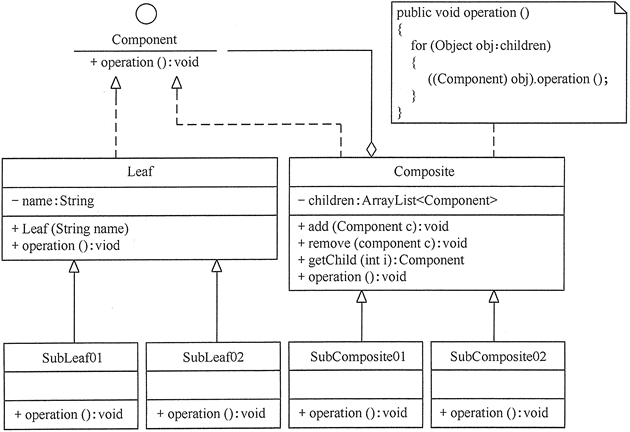
#### 应用场景

1.在需要表示一个对象整体与部分的层次结构的场合

2.要求对用户隐藏组合对象与单个对象的不同，用户可以用统一的接口使用组合结构中的所有对象的场合。

#### 扩展

如果对前面介绍的组合模式中的树叶节点和树枝节点进行抽象，也就是说树叶节点和树枝节点还有子节点，这时组合模式就扩展成复杂的组合模式了，如 Java AWT/Swing 中的简单组件 JTextComponent 有子类 JTextField、JTextArea，容器组件 Container 也有子类 Window、Panel。复杂的组合模式的结构图如下图所示：



### 模板方法模式详解

在面向对象程序设计过程中，程序员常常会遇到这种情况：设计一个系统时知道了算法所需的关键步骤，而且确定了这些步骤的执行顺序，但某些步骤的具体实现还未知，或者说某些步骤的实现与具体的环境相关。

例如，去银行办理业务一般要经过以下4个流程：取号、排队、办理具体业务、对银行工作人员进行评分等，其中取号、排队和对银行工作人员进行评分的业务对每个客户是一样的，可以在父类中实现，但是办理具体业务却因人而异，它可能是存款、取款或者转账等，可以延迟到子类中实现。

这样的例子在生活中还有很多，例如，一个人每天会起床、吃饭、做事、睡觉等，其中“做事”的内容每天可能不同。我们把这些规定了流程或格式的实例定义成模板，允许使用者根据自己的需求去更新它，例如，简历模板、论文模板、Word 中模板文件等。

以下介绍的模板方法模式将解决以上类似的问题。

#### 定义与特点

##### 定义

定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。它是一种类行为型模式。

##### 主要优点

1.它封装了不变部分，扩展可变部分。它把认为是不变部分的算法封装到父类中实现，而把可变部分算法由子类继承实现，便于子类继续扩展

2.它在父类中提取了公共的部分代码，便于代码复用

3.部分方法是由子类实现的，因此子类可以通过扩展方式增加相应的功能，符合开闭原则。

##### 主要缺点

1.对每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象

2.父类中的抽象方法由子类实现，子类执行的结果会影响父类的结果，这导致一种反向的控制结构，它提高了代码阅读的难度。

#### 结构与实现

模板方法模式需要注意抽象类与具体子类之间的协作。

它用到了虚函数的多态性技术以及“不用调用我，让我来调用你”的反向控制技术。现在来介绍它们的基本结构。

##### 模式的结构

模板方法模式包含以下主要角色：

###### 抽象类（Abstract Class）

负责给出一个算法的轮廓和骨架。它由一个模板方法和若干个基本方法构成。这些方法的定义如下：

1.模板方法：定义了算法的骨架，按某种顺序调用其包含的基本方法。

2.基本方法：是整个算法中的一个步骤，包含以下几种类型：

抽象方法：在抽象类中申明，由具体子类实现。

具体方法：在抽象类中已经实现，在具体子类中可以继承或重写它。

钩子方法：在抽象类中已经实现，包括用于判断的逻辑方法和需要子类重写的空方法两种。

###### 具体子类（Concrete Class）

实现抽象类中所定义的抽象方法和钩子方法，它们是一个顶级逻辑的一个组成步骤。

模板方法模式的结构图下图所示：



##### 模式的实现

模板方法模式的代码如下：

package templateMethod;

public class TemplateMethodPattern

{

public static void main(String[] args)

{

AbstractClass tm=new ConcreteClass();

tm.TemplateMethod();

}

}

//抽象类

abstract class AbstractClass

{

public void TemplateMethod() //模板方法

{

SpecificMethod();

abstractMethod1();

abstractMethod2();

}

public void SpecificMethod() //具体方法

{

System.out.println("抽象类中的具体方法被调用...");

}

public abstract void abstractMethod1(); //抽象方法1

public abstract void abstractMethod2(); //抽象方法2

}

//具体子类

class ConcreteClass extends AbstractClass

{

public void abstractMethod1()

{

System.out.println("抽象方法1的实现被调用...");

}

public void abstractMethod2()

{

System.out.println("抽象方法2的实现被调用...");

}

}

程序的运行结果如下：

抽象类中的具体方法被调用...

抽象方法1的实现被调用...

抽象方法2的实现被调用...

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1376.html

#### 应用场景

模板方法模式通常适用于以下场景：

1.算法的整体步骤很固定，但其中个别部分易变时，这时候可以使用模板方法模式，将容易变的部分抽象出来，供子类实现。

2.当多个子类存在公共的行为时，可以将其提取出来并集中到一个公共父类中以避免代码重复。首先，要识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

3.当需要控制子类的扩展时，模板方法只在特定点调用钩子操作，这样就只允许在这些点进行扩展。

#### 扩展

在模板方法模式中，基本方法包含：抽象方法、具体方法和钩子方法，正确使用“钩子方法”可以使得子类控制父类的行为。

如下面例子中，可以通过在具体子类中重写钩子方法 HookMethod1() 和 HookMethod2() 来改变抽象父类中的运行结果，其结构图如下图 所示：



程序代码如下：

package templateMethod;

public class HookTemplateMethod

{

public static void main(String[] args)

{

HookAbstractClass tm=new HookConcreteClass();

tm.TemplateMethod();

}

}

//含钩子方法的抽象类

abstract class HookAbstractClass

{

public void TemplateMethod() //模板方法

{

abstractMethod1();

HookMethod1();

if(HookMethod2())

{

SpecificMethod();

}

abstractMethod2();

}

public void SpecificMethod() //具体方法

{

System.out.println("抽象类中的具体方法被调用...");

}

public void HookMethod1(){} //钩子方法1

public boolean HookMethod2() //钩子方法2

{

return true;

}

public abstract void abstractMethod1(); //抽象方法1

public abstract void abstractMethod2(); //抽象方法2

}

//含钩子方法的具体子类

class HookConcreteClass extends HookAbstractClass

{

public void abstractMethod1()

{

System.out.println("抽象方法1的实现被调用...");

}

public void abstractMethod2()

{

System.out.println("抽象方法2的实现被调用...");

}

public void HookMethod1()

{

System.out.println("钩子方法1被重写...");

}

public boolean HookMethod2()

{

return false;

}

}

程序的运行结果如下：

抽象方法1的实现被调用...

钩子方法1被重写...

抽象方法2的实现被调用...

如果钩子方法 HookMethod1() 和钩子方法 HookMethod2() 的代码改变，则程序的运行结果也会改变。

### 策略模式详解

在现实生活中常常遇到实现某种目标存在多种策略可供选择的情况，例如，出行旅游可以乘坐飞机、乘坐火车、骑自行车或自己开私家车等，超市促销可以釆用打折、送商品、送积分等方法。

在软件开发中也常常遇到类似的情况，当实现某一个功能存在多种算法或者策略，我们可以根据环境或者条件的不同选择不同的算法或者策略来完成该功能，如数据排序策略有冒泡排序、选择排序、插入排序、二叉树排序等。

如果使用多重条件转移语句实现（即硬编码），不但使条件语句变得很复杂，而且增加、删除或更换算法要修改原代码，不易维护，违背开闭原则。如果采用策略模式就能很好解决该问题。

#### 定义与特点

##### 定义

该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

##### 主要优点

1.多重条件语句不易维护，而使用策略模式可以避免使用多重条件语句

2.策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而避免重复的代码

3.策略模式可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同的

4.策略模式提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算法

5.策略模式把算法的使用放到环境类中，而算法的实现移到具体策略类中，实现了二者的分离。

##### 主要缺点

1.客户端必须理解所有策略算法的区别，以便适时选择恰当的算法类

2.策略模式造成很多的策略类。

#### 结构与实现

策略模式是准备一组算法，并将这组算法封装到一系列的策略类里面，作为一个抽象策略类的子类。

策略模式的重心不是如何实现算法，而是如何组织这些算法，从而让程序结构更加灵活，具有更好的维护性和扩展性，现在我们来分析其基本结构和实现方法。

##### 模式的结构

策略模式的主要角色如下：

1. 抽象策略（Strategy）类

定义了一个公共接口，各种不同的算法以不同的方式实现这个接口，环境角色使用这个接口调用不同的算法，一般使用接口或抽象类实现。

1. 具体策略（Concrete Strategy）类

实现了抽象策略定义的接口，提供具体的算法实现。

3.环境（Context）类

持有一个策略类的引用，最终给客户端调用。

其结构图如下图所示：



##### 模式的实现

策略模式的实现代码如下：

package strategy;

public class StrategyPattern

{

public static void main(String[] args)

{

Context c=new Context();

Strategy s=new ConcreteStrategyA();

c.setStrategy(s);

c.strategyMethod();

System.out.println("-----------------");

s=new ConcreteStrategyB();

c.setStrategy(s);

c.strategyMethod();

}

}

//抽象策略类

interface Strategy

{

public void strategyMethod(); //策略方法

}

//具体策略类A

class ConcreteStrategyA implements Strategy

{

public void strategyMethod()

{

System.out.println("具体策略A的策略方法被访问！");

}

}

//具体策略类B

class ConcreteStrategyB implements Strategy

{

public void strategyMethod()

{

System.out.println("具体策略B的策略方法被访问！");

}

}

//环境类

class Context

{

private Strategy strategy;

public Strategy getStrategy()

{

return strategy;

}

public void setStrategy(Strategy strategy)

{

this.strategy=strategy;

}

public void strategyMethod()

{

strategy.strategyMethod();

}

}

程序运行结果如下：

具体策略A的策略方法被访问！

-----------------

具体策略B的策略方法被访问！

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1378.html

#### 应用场景

策略模式在很多地方用到，如 Java SE 中的容器布局管理就是一个典型的实例，Java SE 中的每个容器都存在多种布局供用户选择。

在程序设计中，通常在以下几种情况中使用策略模式较多：

1.一个系统需要动态地在几种算法中选择一种时，可将每个算法封装到策略类中

2.一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现，可将每个条件分支移入它们各自的策略类中以代替这些条件语句

3.系统中各算法彼此完全独立，且要求对客户隐藏具体算法的实现细节时

4.系统要求使用算法的客户不应该知道其操作的数据时，可使用策略模式来隐藏与算法相关的数据结构

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

#### 扩展

在一个使用策略模式的系统中，当存在的策略很多时，客户端管理所有策略算法将变得很复杂，如果在环境类中使用策略工厂模式来管理这些策略类将大大减少客户端的工作复杂度，其结构图如下图所示：



### 命令模式详解

在软件开发系统中，常常出现“方法的请求者”与“方法的实现者”之间存在紧密的耦合关系。这不利于软件功能的扩展与维护。

例如，想对行为进行“撤销、重做、记录”等处理都很不方便，因此“如何将方法的请求者与方法的实现者解耦？”变得很重要，命令模式能很好地解决这个问题。

在现实生活中，这样的例子也很多，例如，电视机遥控器（命令发送者）通过按钮（具体命令）来遥控电视机（命令接收者），还有计算机键盘上的“功能键”等。

#### 定义与特点

##### 定义

将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。

这样两者之间通过命令对象进行沟通，这样方便将命令对象进行储存、传递、调用、增加与管理。

##### 主要优点

1.降低系统的耦合度。命令模式能将调用操作的对象与实现该操作的对象解耦

2.增加或删除命令非常方便。采用命令模式增加与删除命令不会影响其他类，它满足“开闭原则”，对扩展比较灵活

3.可以实现宏命令。命令模式可以与组合模式结合，将多个命令装配成一个组合命令，即宏命令

4.方便实现 Undo 和 Redo 操作。命令模式可以与后面介绍的备忘录模式结合，实现命令的撤销与恢复。

##### 主要缺点

可能产生大量具体命令类。因为计对每一个具体操作都需要设计一个具体命令类，这将增加系统的复杂性。

#### 结构与实现

可以将系统中的相关操作抽象成命令，使调用者与实现者相关分离，其结构如下：

##### 模式的结构

命令模式包含以下主要角色：

###### 抽象命令类（Command）角色

声明执行命令的接口，拥有执行命令的抽象方法 execute()。

###### 具体命令角色（Concrete Command）角色

抽象命令类的具体实现类，它拥有接收者对象，并通过调用接收者的功能来完成命令要执行的操作。

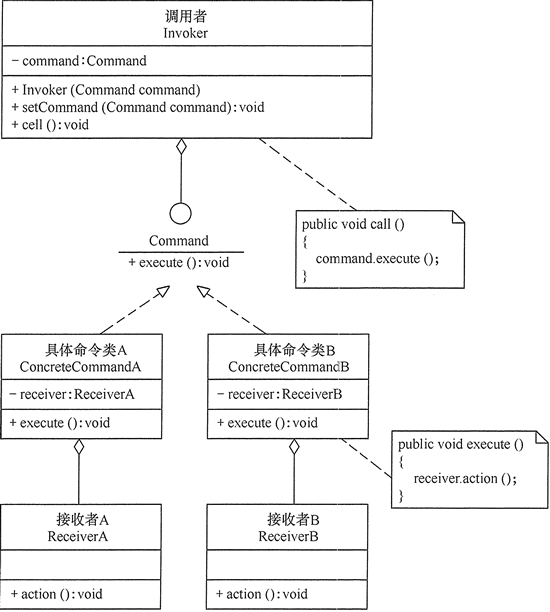
###### 实现者/接收者（Receiver）角色

执行命令功能的相关操作，是具体命令对象业务的真正实现者。

###### 调用者/请求者（Invoker）角色

请求的发送者，它通常拥有很多的命令对象，并通过访问命令对象来执行相关请求，它不直接访问接收者。

其结构图如下图所示：



##### 模式的实现

命令模式的代码如下：

package command;

public class CommandPattern

{

public static void main(String[] args)

{

Command cmd=new ConcreteCommand();

Invoker ir=new Invoker(cmd);

System.out.println("客户访问调用者的call()方法...");

ir.call();

}

}

//调用者

class Invoker

{

private Command command;

public Invoker(Command command)

{

this.command=command;

}

public void setCommand(Command command)

{

this.command=command;

}

public void call()

{

System.out.println("调用者执行命令command...");

command.execute();

}

}

//抽象命令

interface Command

{

public abstract void execute();

}

//具体命令

class ConcreteCommand implements Command

{

private Receiver receiver;

ConcreteCommand()

{

receiver=new Receiver();

}

public void execute()

{

receiver.action();

}

}

//接收者

class Receiver

{

public void action()

{

System.out.println("接收者的action()方法被调用...");

}

}

程序的运行结果如下：

客户访问调用者的call()方法...

调用者执行命令command...

接收者的action()方法被调用...

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1380.html

#### 应用场景

1.当系统需要将请求调用者与请求接收者解耦时，命令模式使得调用者和接收者不直接交互

2.当系统需要随机请求命令或经常增加或删除命令时，命令模式比较方便实现这些功能

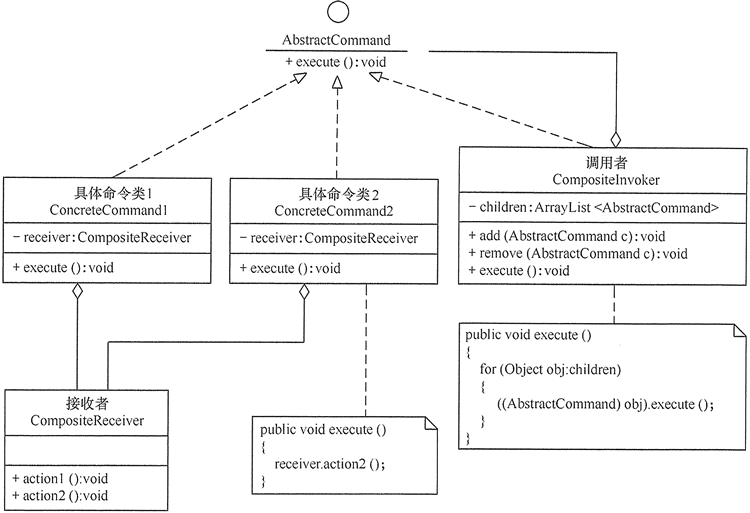
3.当系统需要执行一组操作时，命令模式可以定义宏命令来实现该功能

4.当系统需要支持命令的撤销（Undo）操作和恢复（Redo）操作时，可以将命令对象存储起来，采用备忘录模式来实现。

#### 扩展

在软件开发中，有时将命令模式与前面学的组合模式联合使用，这就构成了宏命令模式，也叫组合命令模式。

宏命令包含了一组命令，它充当了具体命令与调用者的双重角色，执行它时将递归调用它所包含的所有命令，其具体结构图如下图所示：



程序代码如下：

package command;

import java.util.ArrayList;

public class CompositeCommandPattern

{

public static void main(String[] args)

{

AbstractCommand cmd1=new ConcreteCommand1();

AbstractCommand cmd2=new ConcreteCommand2();

CompositeInvoker ir=new CompositeInvoker();

ir.add(cmd1);

ir.add(cmd2);

System.out.println("客户访问调用者的execute()方法...");

ir.execute();

}

}

//抽象命令

interface AbstractCommand

{

public abstract void execute();

}

//树叶构件: 具体命令1

class ConcreteCommand1 implements AbstractCommand

{

private CompositeReceiver receiver;

ConcreteCommand1()

{

receiver=new CompositeReceiver();

}

public void execute()

{

receiver.action1();

}

}

//树叶构件: 具体命令2

class ConcreteCommand2 implements AbstractCommand

{

private CompositeReceiver receiver;

ConcreteCommand2()

{

receiver=new CompositeReceiver();

}

public void execute()

{

receiver.action2();

}

}

//树枝构件: 调用者

class CompositeInvoker implements AbstractCommand

{

private ArrayList<AbstractCommand> children = new ArrayList<AbstractCommand>();

public void add(AbstractCommand c)

{

children.add(c);

}

public void remove(AbstractCommand c)

{

children.remove(c);

}

public AbstractCommand getChild(int i)

{

return children.get(i);

}

public void execute()

{

for(Object obj:children)

{

((AbstractCommand)obj).execute();

}

}

}

//接收者

class CompositeReceiver

{

public void action1()

{

System.out.println("接收者的action1()方法被调用...");

}

public void action2()

{

System.out.println("接收者的action2()方法被调用...");

}

}

程序的运行结果如下：

客户访问调用者的execute()方法...

接收者的action1()方法被调用...

接收者的action2()方法被调用...

当然，命令模式还可以同备忘录（Memento）模式组合使用，这样就变成了可撤销的命令模式，这将在下面有介绍。

### 责任链模式详解

在现实生活中，常常会出现这样的事例：一个请求有多个对象可以处理，但每个对象的处理条件或权限不同。

例如，公司员工请假，可批假的领导有部门负责人、副总经理、总经理等，但每个领导能批准的天数不同，员工必须根据自己要请假的天数去找不同的领导签名，也就是说员工必须记住每个领导的姓名、电话和地址等信息，这增加了难度。

这样的例子还有很多，如找领导出差报销、生活中的“击鼓传花”游戏等。

在计算机软硬件中也有相关例子，如总线网中数据报传送，每台计算机根据目标地址是否同自己的地址相同来决定是否接收；还有异常处理中，处理程序根据异常的类型决定自己是否处理该异常；还有 Struts2 的拦截器、JSP 和 Servlet 的 Filter 等，所有这些，如果用责任链模式都能很好解决。

#### 定义与特点

##### 定义

为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止。

###### 注意

责任链模式也叫职责链模式。

在责任链模式中，客户只需要将请求发送到责任链上即可，无须关心请求的处理细节和请求的传递过程，所以责任链将请求的发送者和请求的处理者解耦了。

##### 主要优点

1.降低了对象之间的耦合度。该模式使得一个对象无须知道到底是哪一个对象处理其请求以及链的结构，发送者和接收者也无须拥有对方的明确信息

2.增强了系统的可扩展性。可以根据需要增加新的请求处理类，满足开闭原则

3.增强了给对象指派职责的灵活性。当工作流程发生变化，可以动态地改变链内的成员或者调动它们的次序，也可动态地新增或者删除责任。

4.责任链简化了对象之间的连接。每个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if···else 语句。

5.责任分担。每个类只需要处理自己该处理的工作，不该处理的传递给下一个对象完成，明确各类的责任范围，符合类的单一职责原则。

##### 主要缺点

1.不能保证每个请求一定被处理。由于一个请求没有明确的接收者，所以不能保证它一定会被处理，该请求可能一直传到链的末端都得不到处理

2.对比较长的职责链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响

3.职责链建立的合理性要靠客户端来保证，增加了客户端的复杂性，可能会由于职责链的错误设置而导致系统出错，如可能会造成循环调用。

#### 结构与实现

通常情况下，可以通过数据链表来实现职责链模式的数据结构。

##### 模式的结构

职责链模式主要包含以下角色：

###### 抽象处理者（Handler）角色

定义一个处理请求的接口，包含抽象处理方法和一个后继连接。

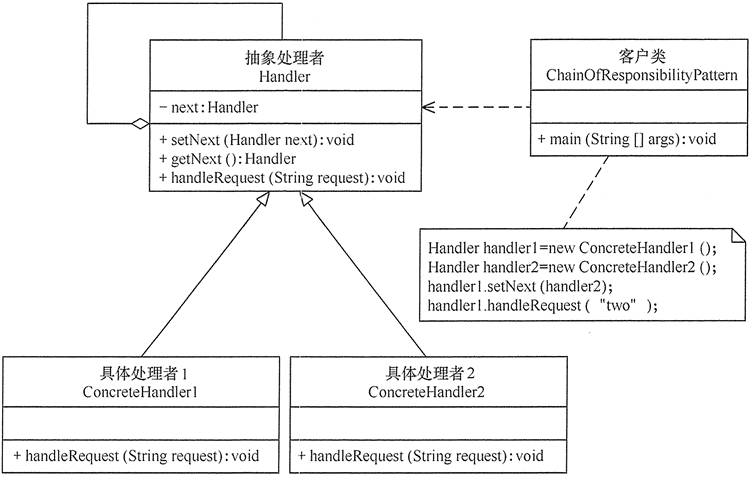
###### 具体处理者（Concrete Handler）角色

实现抽象处理者的处理方法，判断能否处理本次请求，如果可以处理请求则处理，否则将该请求转给它的后继者。

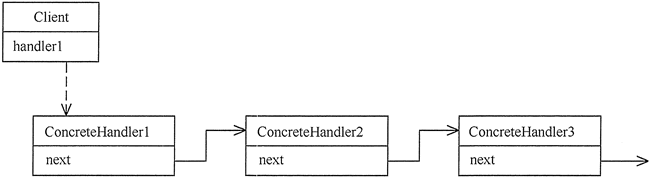
###### 客户类（Client）角色

创建处理链，并向链头的具体处理者对象提交请求，它不关心处理细节和请求的传递过程。

其结构图如下图所示：



客户端可按下图所示设置责任链：



##### 模式的实现

职责链模式的实现代码如下：

package chainOfResponsibility;

public class ChainOfResponsibilityPattern

{

public static void main(String[] args)

{

//组装责任链

Handler handler1=new ConcreteHandler1();

Handler handler2=new ConcreteHandler2();

handler1.setNext(handler2);

//提交请求

handler1.handleRequest("two");

}

}

//抽象处理者角色

abstract class Handler

{

private Handler next;

public void setNext(Handler next)

{

this.next=next;

}

public Handler getNext()

{

return next;

}

//处理请求的方法

public abstract void handleRequest(String request);

}

//具体处理者角色1

class ConcreteHandler1 extends Handler

{

public void handleRequest(String request)

{

if(request.equals("one"))

{

System.out.println("具体处理者1负责处理该请求！");

}

else

{

if(getNext()!=null)

{

getNext().handleRequest(request);

}

else

{

System.out.println("没有人处理该请求！");

}

}

}

}

//具体处理者角色2

class ConcreteHandler2 extends Handler

{

public void handleRequest(String request)

{

if(request.equals("two"))

{

System.out.println("具体处理者2负责处理该请求！");

}

else

{

if(getNext()!=null)

{

getNext().handleRequest(request);

}

else

{

System.out.println("没有人处理该请求！");

}

}

}

}

程序运行结果如下：

具体处理者2负责处理该请求！

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1383.html

#### 应用场景

1.有多个对象可以处理一个请求，哪个对象处理该请求由运行时刻自动确定

2.可动态指定一组对象处理请求，或添加新的处理者

3.在不明确指定请求处理者的情况下，向多个处理者中的一个提交请求。

#### 扩展

职责链模式存在以下两种情况：

纯的职责链模式

一个请求必须被某一个处理者对象所接收，且一个具体处理者对某个请求的处理只能采用以下两种行为之一：自己处理（承担责任）；把责任推给下家处理。

###### 不纯的职责链模式

允许出现某一个具体处理者对象在承担了请求的一部分责任后又将剩余的责任传给下家的情况，且一个请求可以最终不被任何接收端对象所接收。

### 状态模式详解

在软件开发过程中，应用程序中的有些对象可能会根据不同的情况做出不同的行为，我们把这种对象称为有状态的对象，而把影响对象行为的一个或多个动态变化的属性称为状态。

当有状态的对象与外部事件产生互动时，其内部状态会发生改变，从而使得其行为也随之发生改变。如人的情绪有高兴的时候和伤心的时候，不同的情绪有不同的行为，当然外界也会影响其情绪变化。

对这种有状态的对象编程，传统的解决方案是：将这些所有可能发生的情况全都考虑到，然后使用 if-else 语句来做状态判断，再进行不同情况的处理。但当对象的状态很多时，程序会变得很复杂。而且增加新的状态要添加新的 if-else 语句，这违背了“开闭原则”，不利于程序的扩展。

以上问题如果采用“状态模式”就能很好地得到解决。状态模式的解决思想是：当控制一个对象状态转换的条件表达式过于复杂时，把相关“判断逻辑”提取出来，放到一系列的状态类当中，这样可以把原来复杂的逻辑判断简单化。

#### 定义与特点

##### 定义

对有状态的对象，把复杂的“判断逻辑”提取到不同的状态对象中，允许状态对象在其内部状态发生改变时改变其行为。

##### 主要优点

1.状态模式将与特定状态相关的行为局部化到一个状态中，并且将不同状态的行为分割开来，满足“单一职责原则”

2.减少对象间的相互依赖。将不同的状态引入独立的对象中会使得状态转换变得更加明确，且减少对象间的相互依赖

3.有利于程序的扩展。通过定义新的子类很容易地增加新的状态和转换。

##### 主要缺点

1.状态模式的使用必然会增加系统的类与对象的个数

2.状态模式的结构与实现都较为复杂，如果使用不当会导致程序结构和代码的混乱。

#### 结构与实现

状态模式把受环境改变的对象行为包装在不同的状态对象里，其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。现在我们来分析其基本结构和实现方法。

##### 模式的结构

状态模式包含以下主要角色：

###### 环境（Context）角色

也称为上下文，它定义了客户感兴趣的接口，维护一个当前状态，并将与状态相关的操作委托给当前状态对象来处理。

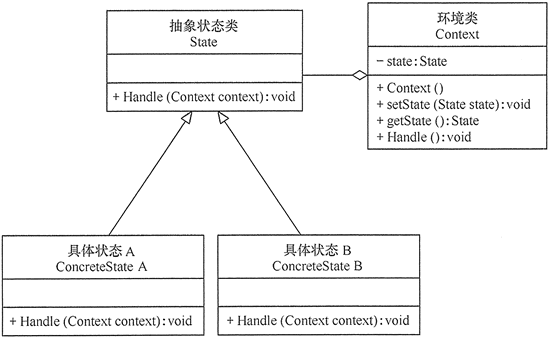
###### 抽象状态（State）角色

定义一个接口，用以封装环境对象中的特定状态所对应的行为。

###### 具体状态（Concrete State）角色

实现抽象状态所对应的行为。

其结构图如下图所示：



##### 模式的实现

状态模式的实现代码如下：

package state;

public class StatePatternClient

{

public static void main(String[] args)

{

Context context=new Context(); //创建环境

context.Handle(); //处理请求

context.Handle();

context.Handle();

context.Handle();

}

}

//环境类

class Context

{

private State state;

//定义环境类的初始状态

public Context()

{

this.state=new ConcreteStateA();

}

//设置新状态

public void setState(State state)

{

this.state=state;

}

//读取状态

public State getState()

{

return(state);

}

//对请求做处理

public void Handle()

{

state.Handle(this);

}

}

//抽象状态类

abstract class State

{

public abstract void Handle(Context context);

}

//具体状态A类

class ConcreteStateA extends State

{

public void Handle(Context context)

{

System.out.println("当前状态是 A.");

context.setState(new ConcreteStateB());

}

}

//具体状态B类

class ConcreteStateB extends State

{

public void Handle(Context context)

{

System.out.println("当前状态是 B.");

context.setState(new ConcreteStateA());

}

}

程序运行结果如下：

当前状态是 A.

当前状态是 B.

当前状态是 A.

当前状态是 B.

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1388.html

#### 应用场景

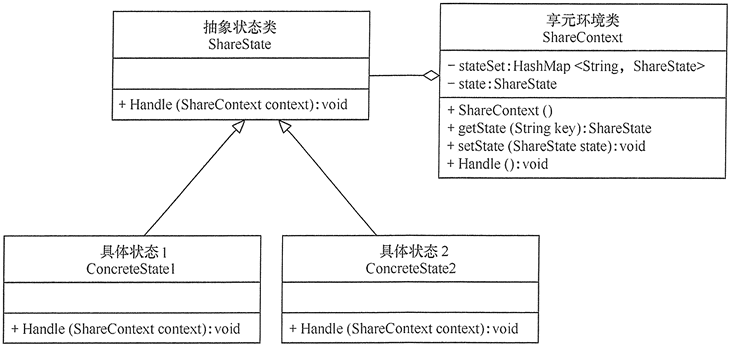
通常在以下情况下可以考虑使用状态模式：

1.当一个对象的行为取决于它的状态，并且它必须在运行时根据状态改变它的行为时，就可以考虑使用状态模式

2.一个操作中含有庞大的分支结构，并且这些分支决定于对象的状态时。

#### 扩展

在有些情况下，可能有多个环境对象需要共享一组状态，这时需要引入享元模式，将这些具体状态对象放在集合中供程序共享，其结构图如下图所示：



分析：共享状态模式的不同之处是在环境类中增加了一个 HashMap 来保存相关状态，当需要某种状态时可以从中获取，其程序代码如下：

package state;

import java.util.HashMap;

public class FlyweightStatePattern

{

public static void main(String[] args)

{

ShareContext context=new ShareContext(); //创建环境

context.Handle(); //处理请求

context.Handle();

context.Handle();

context.Handle();

}

}

//环境类

class ShareContext

{

private ShareState state;

private HashMap<String, ShareState> stateSet=new HashMap<String, ShareState>();

public ShareContext()

{

state=new ConcreteState1();

stateSet.put("1", state);

state=new ConcreteState2();

stateSet.put("2", state);

state=getState("1");

}

//设置新状态

public void setState(ShareState state)

{

this.state=state;

}

//读取状态

public ShareState getState(String key)

{

ShareState s=(ShareState)stateSet.get(key);

return s;

}

//对请求做处理

public void Handle()

{

state.Handle(this);

}

}

//抽象状态类

abstract class ShareState

{

public abstract void Handle(ShareContext context);

}

//具体状态1类

class ConcreteState1 extends ShareState

{

public void Handle(ShareContext context)

{

System.out.println("当前状态是： 状态1");

context.setState(context.getState("2"));

}

}

//具体状态2类

class ConcreteState2 extends ShareState

{

public void Handle(ShareContext context)

{

System.out.println("当前状态是： 状态2");

context.setState(context.getState("1"));

}

}

程序运行结果如下：

当前状态是： 状态1

当前状态是： 状态2

当前状态是： 状态1

当前状态是： 状态2

### 观察者模式详解

在现实世界中，许多对象并不是独立存在的，其中一个对象的行为发生改变可能会导致一个或者多个其他对象的行为也发生改变。例如，某种商品的物价上涨时会导致部分商家高兴，而消费者伤心；还有，当我们开车到交叉路口时，遇到红灯会停，遇到绿灯会行。

这样的例子还有很多，例如，股票价格与股民、微信公众号与微信用户、气象局的天气预报与听众、小偷与警察等。

在软件世界也是这样，例如，Excel 中的数据与折线图、饼状图、柱状图之间的关系；MVC 模式中的模型与视图的关系；事件模型中的事件源与事件处理者。所有这些，如果用观察者模式来实现就非常方便。

#### 定义与特点

##### 定义

多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

这种模式有时又称作发布-订阅模式、模型-视图模式，它是对象行为型模式。

##### 主要优点

1.降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系

2.目标与观察者之间建立了一套触发机制。

##### 主要缺点

1.目标与观察者之间的依赖关系并没有完全解除，而且有可能出现循环引用

2.当观察者对象很多时，通知的发布会花费很多时间，影响程序的效率。

#### 结构与实现

实现观察者模式时要注意具体目标对象和具体观察者对象之间不能直接调用，否则将使两者之间紧密耦合起来，这违反了面向对象的设计原则。

##### 模式的结构

观察者模式的主要角色如下:

###### 抽象主题（Subject）角色

也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。

###### 具体主题（Concrete Subject）角色

也叫具体目标类，它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。

###### 抽象观察者（Observer）角色

它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。

###### 具体观察者（Concrete Observer）角色

实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。

观察者模式的结构图如下图所示:



##### 模式的实现

观察者模式的实现代码如下：

package observer;

import java.util.\*;

public class ObserverPattern

{

public static void main(String[] args)

{

Subject subject=new ConcreteSubject();

Observer obs1=new ConcreteObserver1();

Observer obs2=new ConcreteObserver2();

subject.add(obs1);

subject.add(obs2);

subject.notifyObserver();

}

}

//抽象目标

abstract class Subject

{

protected List<Observer> observers=new ArrayList<Observer>();

//增加观察者方法

public void add(Observer observer)

{

observers.add(observer);

}

//删除观察者方法

public void remove(Observer observer)

{

observers.remove(observer);

}

public abstract void notifyObserver(); //通知观察者方法

}

//具体目标

class ConcreteSubject extends Subject

{

public void notifyObserver()

{

System.out.println("具体目标发生改变...");

System.out.println("--------------");

for(Object obs:observers)

{

((Observer)obs).response();

}

}

}

//抽象观察者

interface Observer

{

void response(); //反应

}

//具体观察者1

class ConcreteObserver1 implements Observer

{

public void response()

{

System.out.println("具体观察者1作出反应！");

}

}

//具体观察者1

class ConcreteObserver2 implements Observer

{

public void response()

{

System.out.println("具体观察者2作出反应！");

}

}

程序运行结果如下：

具体目标发生改变...

--------------

具体观察者1作出反应！

具体观察者2作出反应！

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1390.html

#### 应用场景

通过前面的分析与应用实例可知观察者模式适合以下几种情形：

1.对象间存在一对多关系，一个对象的状态发生改变会影响其他对象

2.当一个抽象模型有两个方面，其中一个方面依赖于另一方面时，可将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

#### 扩展

在Java中，通过java.util.Observable类和java.util.Observer接口定义了观察者模式，只要实现它们的子类就可以编写观察者模式实例。

##### Observable类

Observable 类是抽象目标类，它有一个 Vector 向量，用于保存所有要通知的观察者对象，下面来介绍它最重要的 3 个方法：

void addObserver(Observer o) 方法：用于将新的观察者对象添加到向量中。

void notifyObservers(Object arg) 方法：调用向量中的所有观察者对象的update方法，通知它们数据发生改变。通常越晚加入向量的观察者越先得到通知。

void setChange() 方法：用来设置一个 boolean 类型的内部标志位，注明目标对象发生了变化。当它为真时，notifyObservers() 才会通知观察者。

##### Observer 接口

Observer 接口是抽象观察者，它监视目标对象的变化，当目标对象发生变化时，观察者得到通知，并调用void update(Observable o,Object arg) 方法，进行相应的工作。

##### 示例

利用 Observable 类和 Observer 接口实现原油期货的观察者模式实例。

分析：当原油价格上涨时，空方伤心，多方高兴；当油价下跌时，空方高兴，多方伤心。本实例中的抽象目标（Observable)类在 Java 中已经定义，可以直接定义其子类，即原油期货（OilFutures)类，它是具体目标类，该类中定义一个 SetPriCe(float price) 方法，当原油数据发生变化时调用其父类的 notifyObservers(Object arg) 方法来通知所有观察者；另外，本实例中的抽象观察者接口（Observer）在 Java 中已经定义，只要定义其子类，即具体观察者类（包括多方类 Bull 和空方类 Bear），并实现 update(Observable o,Object arg) 方法即可。其结构图如下图所示：



程序代码如下：

package observer;

import java.util.Observer;

import java.util.Observable;

public class CrudeOilFutures

{

public static void main(String[] args)

{

OilFutures oil=new OilFutures();

Observer bull=new Bull(); //多方

Observer bear=new Bear(); //空方

oil.addObserver(bull);

oil.addObserver(bear);

oil.setPrice(10);

oil.setPrice(-8);

}

}

//具体目标类：原油期货

class OilFutures extends Observable

{

private float price;

public float getPrice()

{

return this.price;

}

public void setPrice(float price)

{

super.setChanged() ; //设置内部标志位，注明数据发生变化

super.notifyObservers(price); //通知观察者价格改变了

this.price=price ;

}

}

//具体观察者类：多方

class Bull implements Observer

{

public void update(Observable o,Object arg)

{

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

if(price>0)

{

System.out.println("油价上涨"+price+"元，多方高兴了！");

}

else

{

System.out.println("油价下跌"+(-price)+"元，多方伤心了！");

}

}

}

//具体观察者类：空方

class Bear implements Observer

{

public void update(Observable o,Object arg)

{

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

if(price>0)

{

System.out.println("油价上涨"+price+"元，空方伤心了！");

}

else

{

System.out.println("油价下跌"+(-price)+"元，空方高兴了！");

}

}

}

程序运行结果如下：

油价上涨10.0元，空方伤心了！

油价上涨10.0元，多方高兴了！

油价下跌8.0元，空方高兴了！

油价下跌8.0元，多方伤心了！

### 中介者模式详解

在现实生活中，常常会出现好多对象之间存在复杂的交互关系，这种交互关系常常是“网状结构”，它要求每个对象都必须知道它需要交互的对象。例如，每个人必须记住他（她）所有朋友的电话；而且，朋友中如果有人的电话修改了，他（她）必须告诉其他所有的朋友修改，这叫作“牵一发而动全身”，非常复杂。

如果把这种“网状结构”改为“星形结构”的话，将大大降低它们之间的“耦合性”，这时只要找一个“中介者”就可以了。如前面所说的“每个人必须记住所有朋友电话”的问题，只要在网上建立一个每个朋友都可以访问的“通信录”就解决了。这样的例子还有很多，例如，你刚刚参加工作想租房，可以找“房屋中介”；或者，自己刚刚到一个陌生城市找工作，可以找“人才交流中心”帮忙。

在软件的开发过程中，这样的例子也很多，例如，在 MVC 框架中，控制器（C）就是模型（M）和视图（V）的中介者；还有大家常用的 QQ 聊天程序的“中介者”是 QQ 服务器。所有这些，都可以采用“中介者模式”来实现，它将大大降低对象之间的耦合性，提高系统的灵活性。

#### 定义与特点

##### 定义

定义一个中介对象来封装一系列对象之间的交互，使原有对象之间的耦合松散，且可以独立地改变它们之间的交互。中介者模式又叫调停模式，它是迪米特法则的典型应用。

##### 主要优点

1.降低了对象之间的耦合性，使得对象易于独立地被复用

2.将对象间的一对多关联转变为一对一的关联，提高系统的灵活性，使得系统易于维护和扩展。

##### 主要缺点

当同事类太多时，中介者的职责将很大，它会变得复杂而庞大，以至于系统难以维护。

#### 结构与实现

中介者模式实现的关键是找出“中介者”，下面对它的结构和实现进行分析。

##### 模式的结构

中介者模式包含以下主要角色：

###### 抽象中介者（Mediator）角色

它是中介者的接口，提供了同事对象注册与转发同事对象信息的抽象方法。

###### 具体中介者（ConcreteMediator）角色

实现中介者接口，定义一个 List 来管理同事对象，协调各个同事角色之间的交互关系，因此它依赖于同事角色。

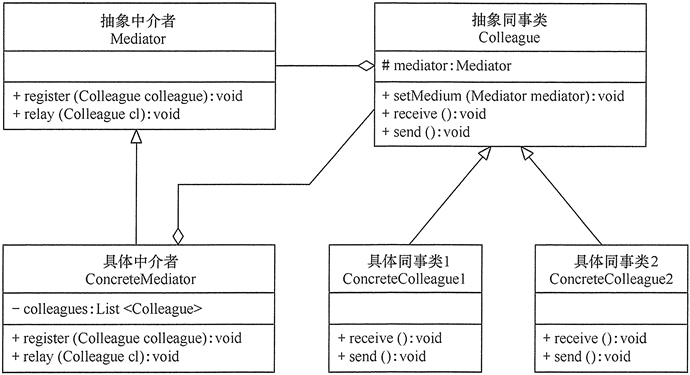
###### 抽象同事类（Colleague）角色

定义同事类的接口，保存中介者对象，提供同事对象交互的抽象方法，实现所有相互影响的同事类的公共功能。

###### 具体同事类（Concrete Colleague）角色

抽象同事类的实现者，当需要与其他同事对象交互时，由中介者对象负责后续的交互。

中介者模式的结构图如下图所示：



##### 模式的实现

中介者模式的实现代码如下：

package mediator;

import java.util.\*;

public class MediatorPattern

{

public static void main(String[] args)

{

Mediator md=new ConcreteMediator();

Colleague c1,c2;

c1=new ConcreteColleague1();

c2=new ConcreteColleague2();

md.register(c1);

md.register(c2);

c1.send();

System.out.println("-------------");

c2.send();

}

}

//抽象中介者

abstract class Mediator

{

public abstract void register(Colleague colleague);

public abstract void relay(Colleague cl); //转发

}

//具体中介者

class ConcreteMediator extends Mediator

{

private List<Colleague> colleagues=new ArrayList<Colleague>();

public void register(Colleague colleague)

{

if(!colleagues.contains(colleague))

{

colleagues.add(colleague);

colleague.setMedium(this);

}

}

public void relay(Colleague cl)

{

for(Colleague ob:colleagues)

{

if(!ob.equals(cl))

{

((Colleague)ob).receive();

}

}

}

}

//抽象同事类

abstract class Colleague

{

protected Mediator mediator;

public void setMedium(Mediator mediator)

{

this.mediator=mediator;

}

public abstract void receive();

public abstract void send();

}

//具体同事类

class ConcreteColleague1 extends Colleague

{

public void receive()

{

System.out.println("具体同事类1收到请求。");

}

public void send()

{

System.out.println("具体同事类1发出请求。");

mediator.relay(this); //请中介者转发

}

}

//具体同事类

class ConcreteColleague2 extends Colleague

{

public void receive()

{

System.out.println("具体同事类2收到请求。");

}

public void send()

{

System.out.println("具体同事类2发出请求。");

mediator.relay(this); //请中介者转发

}

}

程序的运行结果如下：

具体同事类1发出请求。

具体同事类2收到请求。

-------------

具体同事类2发出请求。

具体同事类1收到请求。

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1393.html

#### 应用场景

1.当对象之间存在复杂的网状结构关系而导致依赖关系混乱且难以复用时

2.当想创建一个运行于多个类之间的对象，又不想生成新的子类时。

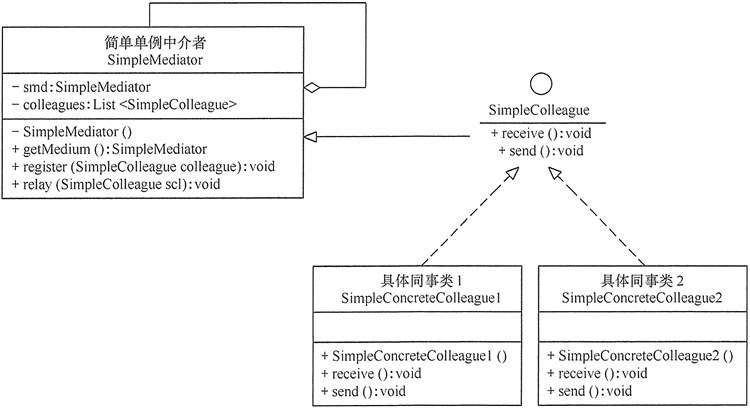
#### 扩展

在实际开发中，通常采用以下两种方法来简化中介者模式，使开发变得更简单。

1.不定义中介者接口，把具体中介者对象实现成为单例

2.同事对象不持有中介者，而是在需要的时f矣直接获取中介者对象并调用。

下图所示是简化中介者模式的结构图：



程序代码如下：

package mediator;

import java.util.\*;

public class SimpleMediatorPattern

{

public static void main(String[] args)

{

SimpleColleague c1,c2;

c1=new SimpleConcreteColleague1();

c2=new SimpleConcreteColleague2();

c1.send();

System.out.println("-----------------");

c2.send();

}

}

//简单单例中介者

class SimpleMediator

{

private static SimpleMediator smd=new SimpleMediator();

private List<SimpleColleague> colleagues=new ArrayList<SimpleColleague>();

private SimpleMediator(){}

public static SimpleMediator getMedium()

{ return(smd); }

public void register(SimpleColleague colleague)

{

if(!colleagues.contains(colleague))

{

colleagues.add(colleague);

}

}

public void relay(SimpleColleague scl)

{

for(SimpleColleague ob:colleagues)

{

if(!ob.equals(scl))

{

((SimpleColleague)ob).receive();

}

}

}

}

//抽象同事类

interface SimpleColleague

{

void receive();

void send();

}

//具体同事类

class SimpleConcreteColleague1 implements SimpleColleague

{

SimpleConcreteColleague1(){

SimpleMediator smd=SimpleMediator.getMedium();

smd.register(this);

}

public void receive()

{ System.out.println("具体同事类1：收到请求。"); }

public void send()

{

SimpleMediator smd=SimpleMediator.getMedium();

System.out.println("具体同事类1：发出请求...");

smd.relay(this); //请中介者转发

}

}

//具体同事类

class SimpleConcreteColleague2 implements SimpleColleague

{

SimpleConcreteColleague2(){

SimpleMediator smd=SimpleMediator.getMedium();

smd.register(this);

}

public void receive()

{ System.out.println("具体同事类2：收到请求。"); }

public void send()

{

SimpleMediator smd=SimpleMediator.getMedium();

System.out.println("具体同事类2：发出请求...");

smd.relay(this); //请中介者转发

}

}

程序运行结果如下：

具体同事类1：发出请求...

具体同事类2：收到请求。

-----------------

具体同事类2：发出请求...

具体同事类1：收到请求。

### 迭代器模式详解

在现实生活以及程序设计中，经常要访问一个聚合对象中的各个元素，如“数据结构”中的链表遍历，通常的做法是将链表的创建和遍历都放在同一个类中，但这种方式不利于程序的扩展，如果要更换遍历方法就必须修改程序源代码，这违背了 “开闭原则”。

既然将遍历方法封装在聚合类中不可取，那么聚合类中不提供遍历方法，将遍历方法由用户自己实现是否可行呢？答案是同样不可取，因为这种方式会存在两个缺点：

1.暴露了聚合类的内部表示，使其数据不安全

2.增加了客户的负担。

“迭代器模式”能较好地克服以上缺点，它在客户访问类与聚合类之间插入一个迭代器，这分离了聚合对象与其遍历行为，对客户也隐藏了其内部细节，且满足“单一职责原则”和“开闭原则”，如 Java 中的 Collection、List、Set、Map 等都包含了迭代器。

#### 定义与特点

##### 定义

提供一个对象来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。

##### 主要优点

1.访问一个聚合对象的内容而无须暴露它的内部表示

2.遍历任务交由迭代器完成，这简化了聚合类

3.它支持以不同方式遍历一个聚合，甚至可以自定义迭代器的子类以支持新的遍历

4.增加新的聚合类和迭代器类都很方便，无须修改原有代码

5.封装性良好，为遍历不同的聚合结构提供一个统一的接口。

##### 主要缺点

增加了类的个数，这在一定程度上增加了系统的复杂性。

#### 结构与实现

迭代器模式是通过将聚合对象的遍历行为分离出来，抽象成迭代器类来实现的，其目的是在不暴露聚合对象的内部结构的情况下，让外部代码透明地访问聚合的内部数据。现在我们来分析其基本结构与实现方法。

##### 模式的结构

迭代器模式主要包含以下角色：

###### 抽象聚合（Aggregate）角色

定义存储、添加、删除聚合对象以及创建迭代器对象的接口。

###### 具体聚合（ConcreteAggregate）角色

实现抽象聚合类，返回一个具体迭代器的实例。

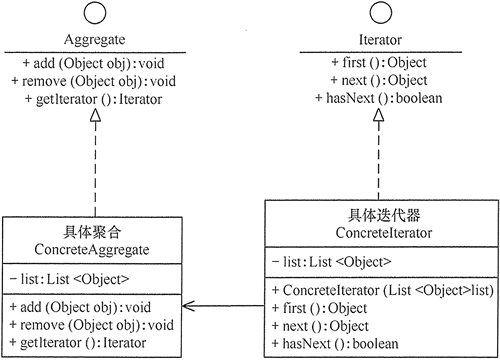
###### 抽象迭代器（Iterator）角色

定义访问和遍历聚合元素的接口，通常包含 hasNext()、first()、next() 等方法。

###### 具体迭代器（Concretelterator）角色

实现抽象迭代器接口中所定义的方法，完成对聚合对象的遍历，记录遍历的当前位置。

其结构图如下图所示：



##### 模式的实现

迭代器模式的实现代码如下：

package iterator;

import java.util.\*;

public class IteratorPattern

{

public static void main(String[] args)

{

Aggregate ag=new ConcreteAggregate();

ag.add("中山大学");

ag.add("华南理工");

ag.add("韶关学院");

System.out.print("聚合的内容有：");

Iterator it=ag.getIterator();

while(it.hasNext())

{

Object ob=it.next();

System.out.print(ob.toString()+"\t");

}

Object ob=it.first();

System.out.println("\nFirst："+ob.toString());

}

}

//抽象聚合

interface Aggregate

{

public void add(Object obj);

public void remove(Object obj);

public Iterator getIterator();

}

//具体聚合

class ConcreteAggregate implements Aggregate

{

private List<Object> list=new ArrayList<Object>();

public void add(Object obj)

{

list.add(obj);

}

public void remove(Object obj)

{

list.remove(obj);

}

public Iterator getIterator()

{

return(new ConcreteIterator(list));

}

}

//抽象迭代器

interface Iterator

{

Object first();

Object next();

boolean hasNext();

}

//具体迭代器

class ConcreteIterator implements Iterator

{

private List<Object> list=null;

private int index=-1;

public ConcreteIterator(List<Object> list)

{

this.list=list;

}

public boolean hasNext()

{

if(index<list.size()-1)

{

return true;

}

else

{

return false;

}

}

public Object first()

{

index=0;

Object obj=list.get(index);;

return obj;

}

public Object next()

{

Object obj=null;

if(this.hasNext())

{

obj=list.get(++index);

}

return obj;

}

}

程序运行结果如下：

聚合的内容有：中山大学 华南理工 韶关学院

First：中山大学

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1395.html

#### 应用场景

迭代器模式通常在以下几种情况使用：

1.当需要为聚合对象提供多种遍历方式时

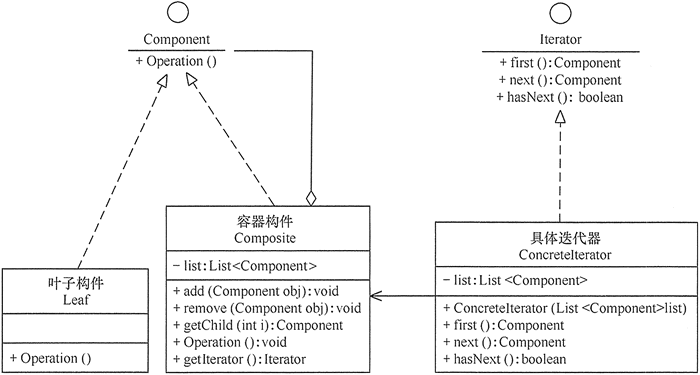
2.当需要为遍历不同的聚合结构提供一个统一的接口时

3.当访问一个聚合对象的内容而无须暴露其内部细节的表示时。

由于聚合与迭代器的关系非常密切，所以大多数语言在实现聚合类时都提供了迭代器类，因此大数情况下使用语言中已有的聚合类的迭代器就已经够了。

#### 扩展

迭代器模式常常与组合模式结合起来使用，在对组合模式中的容器构件进行访问时，经常将迭代器潜藏在组合模式的容器构成类中。当然，也可以构造一个外部迭代器来对容器构件进行访问，其结构图如下图所示：



### 访问者模式详解

在现实生活中，有些集合对象中存在多种不同的元素，且每种元素也存在多种不同的访问者和处理方式。例如，公园中存在多个景点，也存在多个游客，不同的游客对同一个景点的评价可能不同；医院医生开的处方单中包含多种药元素，査看它的划价员和药房工作人员对它的处理方式也不同，划价员根据处方单上面的药品名和数量进行划价，药房工作人员根据处方单的内容进行抓药。

这样的例子还有很多，例如，电影或电视剧中的人物角色，不同的观众对他们的评价也不同；还有顾客在商场购物时放在“购物车”中的商品，顾客主要关心所选商品的性价比，而收银员关心的是商品的价格和数量。

这些被处理的数据元素相对稳定而访问方式多种多样的数据结构，如果用“访问者模式”来处理比较方便。访问者模式能把处理方法从数据结构中分离出来，并可以根据需要增加新的处理方法，且不用修改原来的程序代码与数据结构，这提高了程序的扩展性和灵活性。

#### 定义与特点

##### 定义

将作用于某种数据结构中的各元素的操作分离出来封装成独立的类，使其在不改变数据结构的前提下可以添加作用于这些元素的新的操作，为数据结构中的每个元素提供多种访问方式。它将对数据的操作与数据结构进行分离，是行为类模式中最复杂的一种模式。

##### 主要优点

###### 扩展性好

能够在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能。

###### 复用性好

可以通过访问者来定义整个对象结构通用的功能，从而提高系统的复用程度。

###### 灵活性好

访问者模式将数据结构与作用于结构上的操作解耦，使得操作集合可相对自由地演化而不影响系统的数据结构。

###### 符合单一职责原则

访问者模式把相关的行为封装在一起，构成一个访问者，使每一个访问者的功能都比较单一。

##### 主要缺点

###### 增加新的元素类很困难

在访问者模式中，每增加一个新的元素类，都要在每一个具体访问者类中增加相应的具体操作，这违背了“开闭原则”。

###### 破坏封装

访问者模式中具体元素对访问者公布细节，这破坏了对象的封装性。

###### 违反了依赖倒置原则

访问者模式依赖了具体类，而没有依赖抽象类。

#### 结构与实现

访问者（Visitor）模式实现的关键是如何将作用于元素的操作分离出来封装成独立的类，其基本结构与实现方法如下。

##### 模式的结构

访问者模式包含以下主要角色：

###### 抽象访问者（Visitor）角色

定义一个访问具体元素的接口，为每个具体元素类对应一个访问操作 visit() ，该操作中的参数类型标识了被访问的具体元素。

###### 具体访问者（ConcreteVisitor）角色

实现抽象访问者角色中声明的各个访问操作，确定访问者访问一个元素时该做什么。

###### 抽象元素（Element）角色

声明一个包含接受操作 accept() 的接口，被接受的访问者对象作为 accept() 方法的参数。

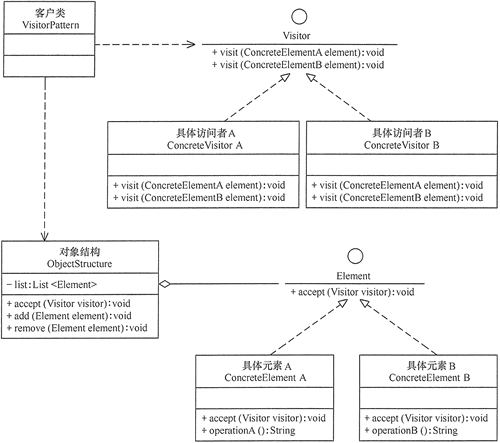
###### 具体元素（ConcreteElement）角色

实现抽象元素角色提供的 accept() 操作，其方法体通常都是 visitor.visit(this) ，另外具体元素中可能还包含本身业务逻辑的相关操作。

###### 对象结构（Object Structure）角色

一个包含元素角色的容器，提供让访问者对象遍历容器中的所有元素的方法，通常由 List、Set、Map 等聚合类实现。

其结构图如下图所示：



##### 模式的实现

访问者模式的实现代码如下：

package visitor;

import java.util.\*;

public class VisitorPattern

{

public static void main(String[] args)

{

ObjectStructure os=new ObjectStructure();

os.add(new ConcreteElementA());

os.add(new ConcreteElementB());

Visitor visitor=new ConcreteVisitorA();

os.accept(visitor);

System.out.println("------------------------");

visitor=new ConcreteVisitorB();

os.accept(visitor);

}

}

//抽象访问者

interface Visitor

{

void visit(ConcreteElementA element);

void visit(ConcreteElementB element);

}

//具体访问者A类

class ConcreteVisitorA implements Visitor

{

public void visit(ConcreteElementA element)

{

System.out.println("具体访问者A访问-->"+element.operationA());

}

public void visit(ConcreteElementB element)

{

System.out.println("具体访问者A访问-->"+element.operationB());

}

}

//具体访问者B类

class ConcreteVisitorB implements Visitor

{

public void visit(ConcreteElementA element)

{

System.out.println("具体访问者B访问-->"+element.operationA());

}

public void visit(ConcreteElementB element)

{

System.out.println("具体访问者B访问-->"+element.operationB());

}

}

//抽象元素类

interface Element

{

void accept(Visitor visitor);

}

//具体元素A类

class ConcreteElementA implements Element

{

public void accept(Visitor visitor)

{

visitor.visit(this);

}

public String operationA()

{

return "具体元素A的操作。";

}

}

//具体元素B类

class ConcreteElementB implements Element

{

public void accept(Visitor visitor)

{

visitor.visit(this);

}

public String operationB()

{

return "具体元素B的操作。";

}

}

//对象结构角色

class ObjectStructure

{

private List<Element> list=new ArrayList<Element>();

public void accept(Visitor visitor)

{

Iterator<Element> i=list.iterator();

while(i.hasNext())

{

((Element) i.next()).accept(visitor);

}

}

public void add(Element element)

{

list.add(element);

}

public void remove(Element element)

{

list.remove(element);

}

}

程序的运行结果如下：

具体访问者A访问-->具体元素A的操作。

具体访问者A访问-->具体元素B的操作。

------------------------

具体访问者B访问-->具体元素A的操作。

具体访问者B访问-->具体元素B的操作。

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1397.html

#### 应用场景

通常在以下情况可以考虑使用访问者（Visitor）模式：

1.对象结构相对稳定，但其操作算法经常变化的程序

2.对象结构中的对象需要提供多种不同且不相关的操作，而且要避免让这些操作的变化影响对象的结构

3.对象结构包含很多类型的对象，希望对这些对象实施一些依赖于其具体类型的操作。

#### 扩展

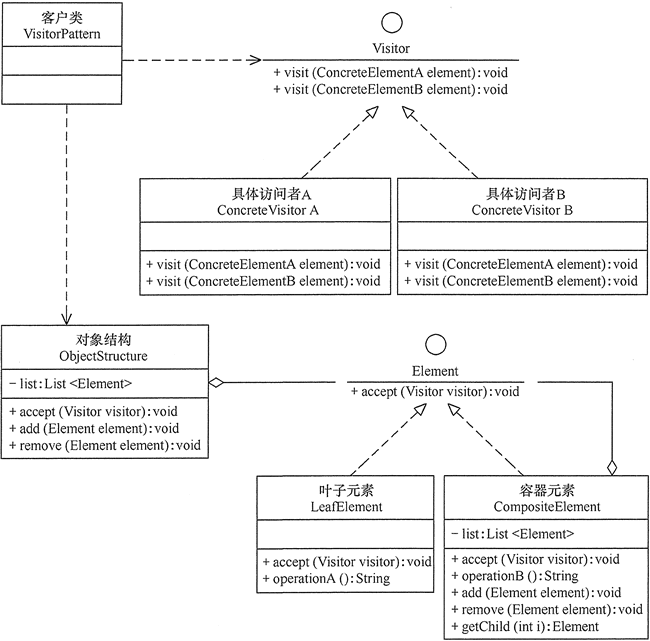
访问者（Visitor）模式是使用频率较高的一种设计模式，它常常同以下两种设计模式联用。

###### 与“迭代器模式”联用

因为访问者模式中的“对象结构”是一个包含元素角色的容器，当访问者遍历容器中的所有元素时，常常要用迭代器。如【例1】中的对象结构是用 List 实现的，它通过 List 对象的 Itemtor() 方法获取迭代器。如果对象结构中的聚合类没有提供迭代器，也可以用迭代器模式自定义一个。

###### 访问者（Visitor）模式同“组合模式”联用

因为访问者（Visitor）模式中的“元素对象”可能是叶子对象或者是容器对象，如果元素对象包含容器对象，就必须用到组合模式，其结构图如下图所示：



### 备忘录模式详解

每个人都有犯错误的时候，都希望有种“后悔药”能弥补自己的过失，让自己重新开始，但现实是残酷的。在计算机应用中，客户同样会常常犯错误，能否提供“后悔药”给他们呢？当然是可以的，而且是有必要的。这个功能由“备忘录模式”来实现。

其实很多应用软件都提供了这项功能，如 Word、记事本、Photoshop、Eclipse 等软件在编辑时按 Ctrl+Z 组合键时能撤销当前操作，使文档恢复到之前的状态；还有在 IE 中的后退键、数据库事务管理中的回滚操作、玩游戏时的中间结果存档功能、数据库与操作系统的备份操作、棋类游戏中的悔棋功能等都属于这类。

备忘录模式能记录一个对象的内部状态，当用户后悔时能撤销当前操作，使数据恢复到它原先的状态。

#### 定义与特点

##### 定义

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态。该模式又叫快照模式。

##### 主要优点

###### 提供了一种可以恢复状态的机制

当用户需要时能够比较方便地将数据恢复到某个历史的状态。

###### 实现了内部状态的封装

除了创建它的发起人之外，其他对象都不能够访问这些状态信息。

###### 简化了发起人类

发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合单一职责原则。

##### 主要缺点

###### 资源消耗大

如果要保存的内部状态信息过多或者特别频繁，将会占用比较大的内存资源。

#### 结构与实现

备忘录模式的核心是设计备忘录类以及用于管理备忘录的管理者类，现在我们来学习其结构与实现。

##### 模式的结构

备忘录模式的主要角色如下：

###### 发起人（Originator）角色

记录当前时刻的内部状态信息，提供创建备忘录和恢复备忘录数据的功能，实现其他业务功能，它可以访问备忘录里的所有信息。

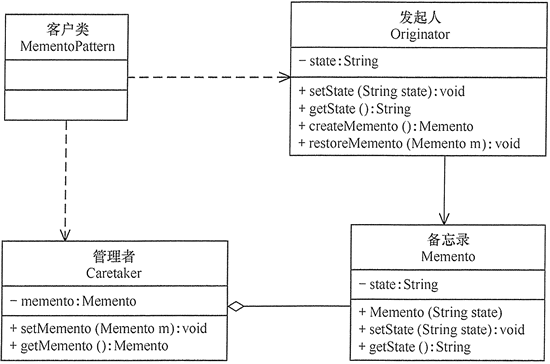
###### 备忘录（Memento）角色

负责存储发起人的内部状态，在需要的时候提供这些内部状态给发起人。

###### 管理者（Caretaker）角色

对备忘录进行管理，提供保存与获取备忘录的功能，但其不能对备忘录的内容进行访问与修改。

备忘录模式的结构图如下图所示：



##### 模式的实现

备忘录模式的实现代码如下：

package memento;

public class MementoPattern

{

public static void main(String[] args)

{

Originator or=new Originator();

Caretaker cr=new Caretaker();

or.setState("S0");

System.out.println("初始状态:"+or.getState());

cr.setMemento(or.createMemento()); //保存状态

or.setState("S1");

System.out.println("新的状态:"+or.getState());

or.restoreMemento(cr.getMemento()); //恢复状态

System.out.println("恢复状态:"+or.getState());

}

}

//备忘录

class Memento

{

private String state;

public Memento(String state)

{

this.state=state;

}

public void setState(String state)

{

this.state=state;

}

public String getState()

{

return state;

}

}

//发起人

class Originator

{

private String state;

public void setState(String state)

{

this.state=state;

}

public String getState()

{

return state;

}

public Memento createMemento()

{

return new Memento(state);

}

public void restoreMemento(Memento m)

{

this.setState(m.getState());

}

}

//管理者

class Caretaker

{

private Memento memento;

public void setMemento(Memento m)

{

memento=m;

}

public Memento getMemento()

{

return memento;

}

}

程序运行的结果如下：

初始状态:S0

新的状态:S1

恢复状态:S0

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1400.html

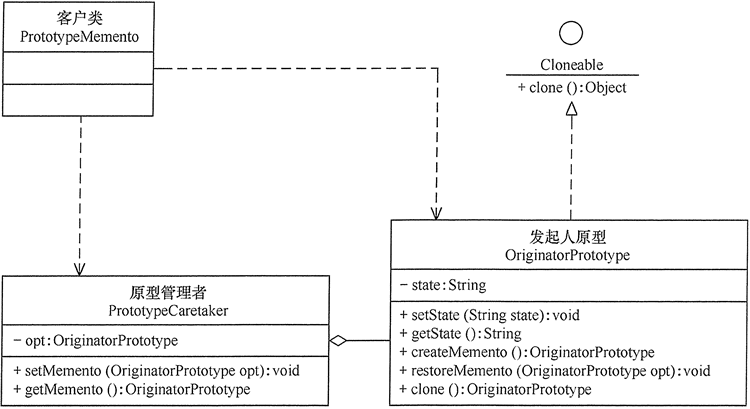
#### 应用场景

1.需要保存与恢复数据的场景，如玩游戏时的中间结果的存档功能

2.需要提供一个可回滚操作的场景，如 Word、记事本、Photoshop，Eclipse 等软件在编辑时按 Ctrl+Z 组合键，还有数据库中事务操作。

#### 扩展

在前面介绍的备忘录模式中，有单状态备份的例子，也有多状态备份的例子。下面介绍备忘录模式如何同原型模式混合使用。在备忘录模式中，通过定义“备忘录”来备份“发起人”的信息，而原型模式的 clone() 方法具有自备份功能，所以，如果让发起人实现 Cloneable 接口就有备份自己的功能，这时可以删除备忘录类，其结构图如下图所示：



实现代码如下：

package memento;

public class PrototypeMemento

{

public static void main(String[] args)

{

OriginatorPrototype or=new OriginatorPrototype();

PrototypeCaretaker cr=new PrototypeCaretaker();

or.setState("S0");

System.out.println("初始状态:"+or.getState());

cr.setMemento(or.createMemento()); //保存状态

or.setState("S1");

System.out.println("新的状态:"+or.getState());

or.restoreMemento(cr.getMemento()); //恢复状态

System.out.println("恢复状态:"+or.getState());

}

}

//发起人原型

class OriginatorPrototype implements Cloneable

{

private String state;

public void setState(String state)

{

this.state=state;

}

public String getState()

{

return state;

}

public OriginatorPrototype createMemento()

{

return this.clone();

}

public void restoreMemento(OriginatorPrototype opt)

{

this.setState(opt.getState());

}

public OriginatorPrototype clone()

{

try

{

return (OriginatorPrototype) super.clone();

}

catch(CloneNotSupportedException e)

{

e.printStackTrace();

}

return null;

}

}

//原型管理者

class PrototypeCaretaker

{

private OriginatorPrototype opt;

public void setMemento(OriginatorPrototype opt)

{

this.opt=opt;

}

public OriginatorPrototype getMemento()

{

return opt;

}

}

程序的运行结果如下：

初始状态:S0

新的状态:S1

恢复状态:S0

### 解释器模式详解

在软件开发中，会遇到有些问题多次重复出现，而且有一定的相似性和规律性。如果将它们归纳成一种简单的语言，那么这些问题实例将是该语言的一些句子，这样就可以用“编译原理”中的解释器模式来实现了。

虽然使用解释器模式的实例不是很多，但对于满足以上特点，且对运行效率要求不是很高的应用实例，如果用解释器模式来实现，其效果是非常好的，本文将介绍其工作原理与使用方法。

#### 定义与特点

##### 定义

给分析对象定义一个语言，并定义该语言的文法表示，再设计一个解析器来解释语言中的句子。也就是说，用编译语言的方式来分析应用中的实例。这种模式实现了文法表达式处理的接口，该接口解释一个特定的上下文。

这里提到的文法和句子的概念同编译原理中的描述相同，“文法”指语言的语法规则，而“句子”是语言集中的元素。例如，汉语中的句子有很多，“我是中国人”是其中的一个句子，可以用一棵语法树来直观地描述语言中的句子。

##### 主要优点

###### 扩展性好

由于在解释器模式中使用类来表示语言的文法规则，因此可以通过继承等机制来改变或扩展文法。

###### 容易实现

在语法树中的每个表达式节点类都是相似的，所以实现其文法较为容易。

##### 主要缺点

###### 执行效率较低

解释器模式中通常使用大量的循环和递归调用，当要解释的句子较复杂时，其运行速度很慢，且代码的调试过程也比较麻烦。

###### 会引起类膨胀

解释器模式中的每条规则至少需要定义一个类，当包含的文法规则很多时，类的个数将急剧增加，导致系统难以管理与维护。

###### 可应用的场景比较少

在软件开发中，需要定义语言文法的应用实例非常少，所以这种模式很少被使用到。

#### 结构与实现

解释器模式常用于对简单语言的编译或分析实例中，为了掌握好它的结构与实现，必须先了解编译原理中的“文法、句子、语法树”等相关概念。

##### 文法

文法是用于描述语言的语法结构的形式规则。没有规矩不成方圆，例如，有些人认为完美爱情的准则是“相互吸引、感情专一、任何一方都没有恋爱经历”，虽然最后一条准则较苛刻，但任何事情都要有规则，语言也一样，不管它是机器语言还是自然语言，都有它自己的文法规则。例如，中文中的“句子”的文法如下：

〈句子〉::=〈主语〉〈谓语〉〈宾语〉

〈主语〉::=〈代词〉|〈名词〉

〈谓语〉::=〈动词〉

〈宾语〉::=〈代词〉|〈名词〉

〈代词〉你|我|他

〈名词〉7大学生I筱霞I英语

〈动词〉::=是|学习

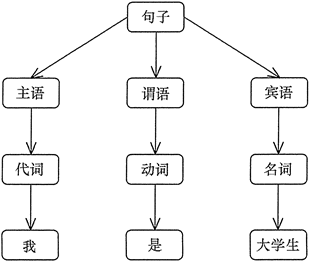
注：这里的符号“::=”表示“定义为”的意思，用“〈”和“〉”括住的是非终结符，没有括住的是终结符。

##### 句子

句子是语言的基本单位，是语言集中的一个元素，它由终结符构成，能由“文法”推导出。例如，上述文法可以推出“我是大学生”，所以它是句子。

##### 语法树

语法树是句子结构的一种树型表示，它代表了句子的推导结果，它有利于理解句子语法结构的层次。下图所示是“我是大学生”的语法树：



有了以上基础知识，现在来介绍解释器模式的结构就简单了。解释器模式的结构与组合模式相似，不过其包含的组成元素比组合模式多，而且组合模式是对象结构型模式，而解释器模式是类行为型模式。

##### 模式的结构

解释器模式包含以下主要角色：

###### 抽象表达式（Abstract Expression）角色

定义解释器的接口，约定解释器的解释操作，主要包含解释方法 interpret()。

###### 终结符表达式（Terminal Expression）角色

抽象表达式的子类，用来实现文法中与终结符相关的操作，文法中的每一个终结符都有一个具体终结表达式与之相对应。

###### 非终结符表达式（Nonterminal Expression）角色

也是抽象表达式的子类，用来实现文法中与非终结符相关的操作，文法中的每条规则都对应于一个非终结符表达式。

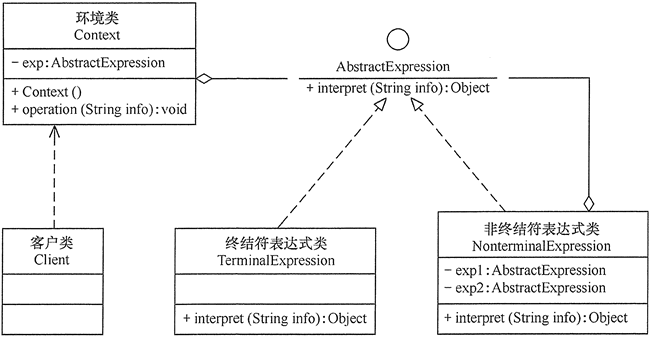
###### 环境（Context）角色

通常包含各个解释器需要的数据或是公共的功能，一般用来传递被所有解释器共享的数据，后面的解释器可以从这里获取这些值。

###### 客户端（Client）

主要任务是将需要分析的句子或表达式转换成使用解释器对象描述的抽象语法树，然后调用解释器的解释方法，当然也可以通过环境角色间接访问解释器的解释方法。

解释器模式的结构图如下图所示：



##### 模式的实现

解释器模式实现的关键是定义文法规则、设计终结符类与非终结符类、画出结构图，必要时构建语法树，其代码结构如下：

//抽象表达式类

interface AbstractExpression

{

public Object interpret(String info); //解释方法

}

//终结符表达式类

class TerminalExpression implements AbstractExpression

{

public Object interpret(String info)

{

//对终结符表达式的处理

}

}

//非终结符表达式类

class NonterminalExpression implements AbstractExpression

{

private AbstractExpression exp1;

private AbstractExpression exp2;

public Object interpret(String info)

{

//非对终结符表达式的处理

}

}

//环境类

class Context

{

private AbstractExpression exp;

public Context()

{

//数据初始化

}

public void operation(String info)

{

//调用相关表达式类的解释方法

}

}

#### 应用实例

详细应用实例请到原文链接查看：http://c.biancheng.net/view/1402.html

#### 应用场景

1.当语言的文法较为简单，且执行效率不是关键问题时

2.当问题重复出现，且可以用一种简单的语言来进行表达时

3.当一个语言需要解释执行，并且语言中的句子可以表示为一个抽象语法树的时候，如 XML 文档解释。

##### 注意

解释器模式在实际的软件开发中使用比较少，因为它会引起效率、性能以及维护等问题。如果碰到对表达式的解释，在 Java 中可以用 Expression4J 或 Jep 等来设计。

#### 扩展

在项目开发中，如果要对数据表达式进行分析与计算，无须再用解释器模式进行设计了，Java 提供了以下强大的数学公式解析器：Expression4J、MESP(Math Expression String Parser) 和 Jep 等，它们可以解释一些复杂的文法，功能强大，使用简单。

现在以 Jep 为例来介绍该工具包的使用方法。Jep 是 Java expression parser 的简称，即 Java 表达式分析器，它是一个用来转换和计算数学表达式的 Java 库。通过这个程序库，用户可以以字符串的形式输入一个任意的公式，然后快速地计算出其结果。而且 Jep 支持用户自定义变量、常量和函数，它包括许多常用的数学函数和常量。

使用前先下载 Jep 压缩包，解压后，将 jep-x.x.x.jar 文件移到选择的目录中，在 Eclipse 的“Java 构建路径”对话框的“库”选项卡中选择“添加外部 JAR(X)...”，将该 Jep 包添加项目中后即可使用其中的类库。

下面以计算存款利息为例来介绍。存款利息的计算公式是：本金x利率x时间=利息，其相关代码如下：

package interpreterPattern;

import com.singularsys.jep.\*;

public class JepDemo

{

public static void main(String[] args) throws JepException

{

Jep jep=new Jep();

//定义要计算的数据表达式

String 存款利息="本金\*利率\*时间";

//给相关变量赋值

jep.addVariable("本金",10000);

jep.addVariable("利率",0.038);

jep.addVariable("时间",2);

jep.parse(存款利息); //解析表达式

Object accrual=jep.evaluate(); //计算

System.out.println("存款利息："+accrual);

}

}

程序运行结果如下：

存款利息：760.0

## 设计模式总结

### 单例模式

#### 使用场景

1.要求生成唯一序列号的环境

2.在整个项目中需要一个共享访问点或共享数据，例如一个Web页面上的计数器，可以不用把每次刷新都记录到数据库中，使用单例模式保持计数器的值，并确保是线程安全的

3.创建一个对象需要消耗的资源过多，如要访问IO和数据库等资源

4.需要定义大量的静态常量和静态方法（如工具类）的环境，可以采用单例模式（当然，也可以直接声明为static的方式）。

#### 使用示例

1. 一个党只能有一个主席

2、Windows 是多进程多线程的，在操作一个文件的时候，就不可避免地出现多个进程或线程同时操作一个文件的现象，所以所有文件的处理必须通过唯一的实例来进行

3、一些设备管理器常常设计为单例模式，比如一个电脑有两台打印机，在输出的时候就要处理不能两台打印机打印同一个文件。

###### 注意事项

getInstance() 方法中需要使用同步锁 synchronized (Singleton.class) 防止多线程同时进入造成 instance 被多次实例化。

### 原型模式

#### 使用场景

1.资源优化场景

类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。

2.性能和安全要求的场景

通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。

3.一个对象多个修改者的场景

一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。

#### 使用示例

1、细胞分裂

2、JAVA 中的 Object clone() 方法。

###### 注意事项

与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的。

浅拷贝实现 Cloneable，重写，深拷贝是通过实现 Serializable 读取二进制流。

### 工厂模式

#### 使用场景

1.消费者不关心它所要创建对象的类(产品类)的时候

2.消费者知道它所要创建对象的类(产品类)，但不关心如何创建的时候。

例如：hibernate里通过sessionFactory创建session、通过代理方式生成ws客户端时，通过工厂构建报文中格式化数据的对象。

#### 使用示例

1、日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方

2、数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时

3、设计一个连接服务器的框架，需要三个协议，”POP3”、”IMAP”、”HTTP”，可以把这三个作为产品类，共同实现一个接口。

###### 注意事项

作为一种创建类模式，在任何需要生成复杂对象的地方，都可以使用工厂方法模式。

有一点需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过 new 就可以完成创建的对象，无需使用工厂模式。

如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。

### 抽象工厂模式

#### 使用场景

1.一个对象族（或是一组没有任何关系的对象）都有相同的约束

2.涉及不同操作系统的时候，都可以考虑使用抽象工厂模式。

#### 使用示例

1、QQ 换皮肤，一整套一起换

2、生成不同操作系统的程序。

###### 注意事项

产品族难扩展，产品等级易扩展。

### 建造者模式

#### 使用场景

1.相同的方法，不同的执行顺序，产生不同的事件结果时，可以采用建造者模式

2.多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时，则可以使用该模式

3.产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效能，这个时候使用建造者模式非常合适。

#### 使用示例

1、去肯德基，汉堡、可乐、薯条、炸鸡翅等是不变的，而其组合是经常变化的，生成出所谓的”套餐”

2、JAVA中的StringBuilder。

###### 注意事项

与工厂模式的区别如下：

1.建造者模式最主要的功能是基本方法的调用顺序安排，这些基本方法已经实现了，顺序不同产生的对象也不同

2.工厂方法则重点是创建，创建零件是它的主要职责，组装顺序则不是它关心的。

### 代理模式

#### 使用场景

代理模式的类型较多，不同类型的代理模式有不同的优缺点，它们应用于不同的场合：

1.当客户端对象需要访问远程主机中的对象时，可以使用远程代理

2.当需要用一个消耗资源较少的对象来代表一个消耗资源较多的对象，从而降低系统开销、缩短运行时间时，可以使用虚拟代理，例如一个对象需要很长时间才能完成加载时

3.当需要为某一个被频繁访问的操作结果提供一个临时存储空间，以供多个客户端共享访问这些结果时，可以使用缓冲代理。通过使用缓冲代理，系统无须在客户端每一次访问时都重新执行操作，只需直接从临时缓冲区获取操作结果即可

4.当需要控制对一个对象的访问，为不同用户提供不同级别的访问权限时，可以使用保护代理

5.当需要为一个对象的访问（引用）提供一些额外的操作时，可以使用智能引用代理。

#### 使用示例

1. Spring中AOP底层动态代理实现
2. 日志记录
3. 权限控制
4. 过滤器
5. RPC远程调用

### 适配器模式

#### 使用场景

你有动机修改一个已经投产中的接口时，适配器模式可能是最适合你的模式。

比如系统扩展了，需要使用一个已有或新建立的类，但这个类又不符合系统的接口，怎么办？使用适配器模式。

#### 使用示例

1、美国电器 110V，中国 220V，就要有一个适配器将110V转化为 220V

2、JAVA JDK 1.1 提供了Enumeration 接口，而在1.2中提供了Iterator 接口，想要使用1.2的JDK，则要将以前系统的Enumeration接口转化为Iterator接口，这时就需要适配器模式

3、在LINUX上运行WINDOWS 程序

4、JAVA中的jdbc。

###### 注意事项

适配器不是在详细设计时添加的，而是解决正在服役的项目的问题。.

### 桥接模式

#### 使用场景

1.不希望或不适用使用继承的场景

2.接口或抽象类不稳定的场景

3.重用性要求较高的场景。

##### 注意

发现类的继承有N层时，可以考虑使用桥梁模式。桥梁模式主要考虑如何拆分抽象和实现。

#### 使用示例

1.女士皮包的选购

女士皮包有很多种，可以按用途分、按皮质分、按品牌分、按颜色分、按大小分等，存在多个维度的变化，所以采用桥接模式来实现女士皮包的选购比较合适。

2. 穿衣搭配

以男人为例，上班时会着装正式（领带、衬衫、外套、西裤、皮鞋、手表等），睡觉时会穿舒适（睡衣、睡裤等），运动时会穿运动服（T恤衫、运动裤、运动鞋等），存在多个维度的变化，可以采用桥接模式实现。

### 装饰模式

#### 使用场景

1.需要扩展一个类的功能，或给一个类增加附加功能

2.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

3.需要为一批的兄弟类进行改装或加装功能，当然是首选装饰模式。

#### 使用示例

1.游戏角色“莫莉卡·安斯兰”的变身

在《恶魔战士》中，游戏角色“莫莉卡·安斯兰”的原身是一个可爱少女，但当她变身时，会变成头顶及背部延伸出蝙蝠状飞翼的女妖，当然她还可以变为穿着漂亮外衣的少女。这些都可用装饰模式来实现。

2.穿衣搭配

以男人为例，上班时会着装正式（领带、衬衫、外套、西裤、皮鞋、手表等），睡觉时会穿舒适（睡衣、睡裤等），运动时会穿运动服（T恤衫、运动裤、运动鞋等），存在多个维度的变化，可以采用桥接模式实现。

### 外观模式

#### 使用场景

1.对分层结构系统构建时，使用外观模式定义子系统中每层的入口点可以简化子系统之间的依赖关系

2.当一个复杂系统的子系统很多时，外观模式可以为系统设计一个简单的接口供外界访问

3.当客户端与多个子系统之间存在很大的联系时，引入外观模式可将它们分离，从而提高子系统的独立性和可移植性。

#### 使用示例

1. spring ApplicationContext

它实现了Factory、ResourceLoader等接口，并通过引用这些接口的实例，对外统一提供：加载配置、解析资源、创建Bean、提供环境、启动流程等功能;客户代码只需要操作context就可以获取spring的提供的功能，而无需关心内部的细节。

2. 启动电脑（按一下电源键）

启动CPU、启动内存、启动硬盘

1. 电影发行公司

电影发行公司，他去跟影院谈影片排片时，他会说自己能拍出哪些类型的电影，希望这几部电影排片率高一些，那么影院真正找电影发行公司要电影发行时，电影发行公司提供电影，但是他的电影都是从那些导演那边拿过来给影院放映的。

这个小模型中，电影发行公司其实就是导演或者说众多电影的门面，各大影院需要放映找发行公司就可以，而不是一个一个找导演。

### 享元模式

#### 使用场景

1.系统中存在大量的相似对象

2.细粒度的对象都具备较接近的外部状态，而且内部状态与环境无关，也就是说对象没有特定身份

3.需要缓冲池的场景。

##### 注意

1. 享元模式是线程不安全的

只有依靠经验，在需要的地方考虑一下线程安全，在大部分场景下不用考虑。对象池中的享元对象尽量多，多到足够满足为止。

1. 性能安全

外部状态最好以java的基本类型作为标志，如String，int，可以提高效率。

#### 使用示例

1. 五子棋游戏
2. 体育馆（进行足球赛、开演唱会等）

### 组合模式

#### 使用场景

1.维护和展示部分-整体关系的场景，如树形菜单、文件和文件夹管理

2.从一个整体中能够独立出部分模块或功能的场景。

##### 注意

只要是树形结构，就考虑使用组合模式。

#### 使用示例

1.商场购物

当用户在商店购物后，显示其所选商品信息，并计算所选商品总价的功能。

2.韦博优惠券

假设存在韦博优惠券，其代表韦博英语优惠券、嗨英语优惠券和开心豆优惠券，只要是韦博所属中心都可以使用。

### 模板方法模式

#### 使用场景

1.多个子类有公有的方法，并且逻辑基本相同时

2.重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现

3.重构时，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子函数约束其行为。

#### 使用示例

1.出国留学手续

出国留学手续一般经过以下流程：索取学校资料，提出入学申请，办理因私出国护照、出境卡和公证，申请签证，体检、订机票、准备行装，抵达目标学校等，其中有些业务对各个学校是一样的，但有些业务因学校不同而不同，所以比较适合用模板方法模式来实现。

2.去银行办理业务

通常步骤的次序是：取号 –> 填单 –> 等待叫号–>办理业务。这几个步骤中，有的是不变的，比如取号，每个人都要取，但有的是要变的，比如都有填单，但是根据不同的业务，填写的单据不同。

### 策略模式

#### 使用场景

1.多个类只有在算法或行为上稍有不同的场景

2.算法需要自由切换的场景

3.需要屏蔽算法规则的场景。

#### 使用示例

1. 烹饪大闸蟹

关于大闸蟹的做法有很多种，可以清蒸大闸蟹、红烧大闸蟹、香辣大闸蟹等。

1. 旅行的出游方式

选择骑自行车、坐汽车，每一种旅行方式都是一个策略。

### 命令模式

#### 使用场景

命令模式通常适用于以下场景：

1.当系统需要将请求调用者与请求接收者解耦时，命令模式使得调用者和接收者不直接交互

2.当系统需要随机请求命令或经常增加或删除命令时，命令模式比较方便实现这些功能

3.当系统需要执行一组操作时，命令模式可以定义宏命令来实现该功能

4.当系统需要支持命令的撤销（Undo）操作和恢复（Redo）操作时，可以将命令对象存储起来，采用备忘录模式来实现。

#### 使用示例

1. 客户去餐馆吃早餐

客户去餐馆可选择的早餐有肠粉、河粉和馄饨等，客户可向服务员选择以上早餐中的若干种，服务员将客户的请求交给相关的厨师去做。

这里的点早餐相当于“命令”，服务员相当于“调用者”，厨师相当于“接收者”，所以用命令模式实现比较合适。

2.遥控器控制电器开关

### 责任链模式

#### 使用场景

#### 使用示例

1.请假条审批

假如规定学请假小于或等于 2 天，班主任可以批准；小于或等于 7 天，系主任可以批准；小于或等于 10 天，院长可以批准；其他情况不予批准；这个实例适合使用职责链模式实现。

2. 需求开发

假设现在有个需求来了，首先是实习生拿到这个需求。

如果实习生能够实现，直接实现。如果不行，他把这个需求交给初级工程师。

如果初级工程师能够实现，直接实现。如果不行，交给中级工程师。

如果中级工程师能够实现，直接实现。如果不行，交给高级工程师。

如果高级工程师能够实现，直接实现。如果不行，交给 CTO。

如果 CTO能够实现，直接实现。如果不行，直接跟产品说，需求不做。

### 状态模式

#### 使用场景

通常在以下情况下可以考虑使用状态模式：

1.当一个对象的行为取决于它的状态，并且它必须在运行时根据状态改变它的行为时

2.一个操作中含有庞大的分支结构，并且这些分支决定于对象的状态时。

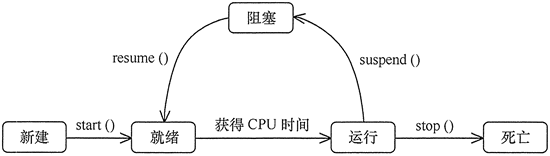
#### 使用示例

1. 学生成绩的状态转换

假设当前学员成绩存在“不及格”、“中等”和“优秀”3种状态，当学生的分数小于60分时为“不及格”状态，当分数大于等于60分且小于90分时为“中等”状态，当分数大于等于90分时为“优秀”状态。

1. 多线程的状态转换

多线程存在 5 种状态，分别为新建状态、就绪状态、运行状态、阻塞状态和死亡状态，各个状态当遇到相关方法调用或事件触发时会转换到其他状态，其状态转换规律如下图所示：



### 观察者模式

#### 使用场景

1. 关联行为场景

需要注意的是，关联行为是可拆分的，而不是“组合”关系

2.事件多级触发场景

3.跨系统的消息交换场景，如消息队列的处理机制。

##### 注意

1.广播链的问题

在一个观察者模式中最多出现一个对象既是观察者也是被观察者，也就是说消息最多转发一次（传递两次）。

2.异步处理问题

观察者比较多，而且处理时间比较长，采用异步处理来考虑线程安全和队列的问题。

#### 使用示例

1. “人民币汇率”变更产生的影响

当“人民币汇率”升值时，进口公司的进口产品成本降低且利润率提升，出口公司的出口产品收入降低且利润率降低；当“人民币汇率”贬值时，进口公司的进口产品成本提升且利润率降低，出口公司的出口产品收入提升且利润率提升。

1. 学校铃声的事件处理

学校的“铃”是事件源和目标，“老师”和“学生”是事件监听器和具体观察者，“铃声”是事件类。学生和老师来到学校的教学区，都会注意学校的铃，这叫事件绑定；当上课时间或下课时间到，会触发铃发声，这时会生成“铃声”事件；学生和老师听到铃声会开始上课或下课，这叫事件处理。

### 中介者模式

#### 使用场景

中介者模式适用于多个对象之间紧密耦合的情况，紧密耦合的标准是：在类图中出现了蜘蛛网状结构，即每个类都与其他的类有直接的联系。总的来说，使用场景如下：

1.当对象之间存在复杂的网状结构关系而导致依赖关系混乱且难以复用时

2.当想创建一个运行于多个类之间的对象，又不想生成新的子类时。

#### 使用示例

1.QQ群、QQ游戏平台

不论是QQ游戏还是QQ群，它们都是充当一个中间平台，QQ用户可以登录这个中间平台与其他QQ用户进行交流，如果没有这些中间平台，我们如果想与朋友进行聊天的话，可能就需要当面才可以了。

2.联合国

中国加入 WTO 之前是各个国家相互贸易，结构复杂，现在是各个国家通过 WTO 来互相贸易。

1. 租房的中介系统

房主跟租房者不需要知道彼此只需要，只要在中介系统发布消息；

如此房主跟租房者之间不需要建立复杂关系，他们都只需要跟中介建立关系。

### 迭代器模式

#### 使用场景

迭代器模式通常在以下几种情况使用：

1.当需要为聚合对象提供多种遍历方式时

2.当需要为遍历不同的聚合结构提供一个统一的接口时

3.当访问一个聚合对象的内容而无须暴露其内部细节的表示时。

由于聚合与迭代器的关系非常密切，所以大多数语言在实现聚合类时都提供了迭代器类，因此大数情况下使用语言中已有的聚合类的迭代器就已经够了。

#### 使用示例

1.Java集合框架

List, Set, Map 都支持迭代

2.查看书柜书籍

有一个书柜，书柜上放有一堆的书籍，我们在可以对这个书柜存放书籍或移除书籍，也可以查看所有书籍的信息，这里的目的是遍历获取到所有的书籍信息，这种情况下可以使用迭代模式。

### 访问者模式

#### 使用场景

通常在以下情况可以考虑使用访问者（Visitor）模式：

1.对象结构相对稳定，但其操作算法经常变化的程序

2.对象结构中的对象需要提供多种不同且不相关的操作，而且要避免让这些操作的变化影响对象的结构

3.对象结构包含很多类型的对象，希望对这些对象实施一些依赖于其具体类型的操作。

##### 注意

访问者模式是一种集中规整模式,特别使用于大规模重构的项目, 在这一个阶段需求已经非常清晰, 原系统的功能点也已经明确, 通过访问者模式可以很容易把一些功能进行梳理, 达到最终目的--功能集中化, 如一个统一的报表运算、UI展现等, 我们还可以与其它模式混编建立一套自己的过滤器或者拦截器。

#### 使用示例

1.模拟艺术公司与造币公司的功能

艺术公司利用“铜”可以设计出铜像，利用“纸”可以画出图画；造币公司利用“铜”可以印出铜币，利用“纸”可以印出纸币。对“铜”和“纸”这两种元素，两个公司的处理方法不同，所以该实例用访问者模式来实现比较适合。

2. 统计功能

对不同的具体元素进行统计, 针对不同的具体元素针对性统计。

比如统计员工工资, 经理和员工的工资情况不同, 使用访问者进行分别计算, 然后可以计算总额等。

### 备忘录模式

#### 使用场景

1.需要保存和恢复数据的相关状态场景

2.提供一个可回滚（rollback）的操作

3.需要监控的副本场景中

4.数据库连接的事务管理就是用的备忘录模式。

##### 注意

1.备忘录的生命期

2.备忘录的性能。

不要在频繁建立备份的场景中使用备忘录模式（比如一个for循环中）。

#### 使用示例

1. 相亲游戏

假如有西施、王昭君、貂蝉、杨玉环四大美女同你相亲，你可以选择其中一位作为你的爱人；当然，如果你对前面的选择不满意，还可以重新选择；这个游戏提供后悔功能，用“备忘录模式”设计比较合适。

2.打游戏时的存档

3.数据库的事务管理

### 解释器模式

#### 使用场景

1.当语言的文法较为简单，且执行效率不是关键问题时

2.当问题重复出现，且可以用一种简单的语言来进行表达时

3.当一个语言需要解释执行，并且语言中的句子可以表示为一个抽象语法树的时候，如 XML 文档解释。

##### 注意

尽量不要在重要的模块中使用解释器模式，否则维护会是一个很大的问题。在项目中可以使用shell、JRuby、Groovy等脚本语言来代替解释器模式，弥补Java编译型语言的不足。

#### 使用示例

1. 公交车卡的读卡器

假如XXX公交车读卡器可以判断乘客的身份，如果是“老人”、“妇女”、“儿童”中任意一个就可以免费乘车，其他人员乘车一次扣2元。

1. 机器人控制程序

开发机器人控制程序，通过英文控制，每个表达式包括移动方向、方式、距离，表达式之间通过and连接，例如： down run 10 and left move 20。

### 模式对比

#### 外观模式、代理模式和中介者模式对比

##### 外观模式（Facade Pattern）

定义一个外观类，外观类隐藏系统的复杂性，为客户端提供简化的方法和对现有系统类方法的委托调用。

例如：二手房交易的中介，属于外观模式。买房者通过中介可以简单地买到二手房，中介自己把联系房东看房砍价、过户、交税这些复杂的事情都搞定了。

##### 代理模式（Proxy Pattern）

用一个代理类代表另一个类的功能，但是不改变被代理类的功能。目的是控制对被代理类的访问。

##### 中介者模式（Mediator Pattern）

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。将各对象之间的网状结构分离为星型结构。

例如：MVC 框架，其中C（控制器）就是 M（模型）和 V（视图）的中介者。微信群是各群员之间的中介者。

#### 桥接模式、装饰模式和适配器模式对比

##### 桥接模式

重点强调的是多维度的变化。

关键点在于：

1.主体类依赖抽象A

2.主体类具有多个不同的实现类

3.抽象A具有多个不同的实现类。

最终的效果就是，主体类的实现类和抽象的实现类分别可以在两个维度上进行各自的变化。如果主体类依赖多个抽象，则维度进行增加，方便扩展。

##### 装饰器模式

重点强调的是装饰功能。

关键点在于：

1.抽象A具有多个具体子类

2.装饰器类依赖抽象A

3.装饰器类实现抽象A

4.装饰器类存在不同子类。

最终的效果就是，（装饰器实现类）对（原抽象的子类）进行某些方法的功能加强。

##### 适配器模式

重点强调的是适配的功能。

关键点在于：

1.主体类和适配器类实现相同的接口A

2.主体类依赖适配器类

3.适配器类依赖抽象接口B

4.被适配的类实现抽象接口B。

最终的效果就是，主体类可以使用之前不相关的被适配类中的某些功能。

#### 工厂方法模式和建造者模式对比

工厂方法模式注重的是整体对象的创建方法，而建造者模式注重的是部件构建的过程，旨在通过一步一步地精确构造创建出一个复杂的对象。

工厂方法模式和建造者模式都属于对象创建类模式，都用来创建类的对象。但它们之间的区别还是比较明显的：

##### 意图不同

在工厂方法模式里，我们关注的是一个产品整体；但在建造者模式中，一个具体产品的产生是依赖各个部件的产生以及装配顺序，它关注的是“由零件一步一步地组装出产品对象”。简单地说，工厂模式是一个对象创建的粗线条应用，建造者模式则是通过细线条勾勒出一个复杂对象，关注的是产品组成部分的创建过程。

##### 产品的复杂度不同

工厂方法模式创建的产品一般都是单一性质产品，都是一个模样，而建造者模式创建的则是一个复合产品，它由各个部件复合而成，部件不同产品对象当然不同。这不是说工厂方法模式创建的对象简单，而是指它们的粒度大小不同。一般来说，工厂方法模式的对象粒度比较粗，建造者模式的产品对象粒度比较细。

#### 抽象工厂模式和建造者模式对比

抽象工厂模式实现对产品家族的创建，一个产品家族是这样的一系列产品：具有不同分类维度的产品组合，采用抽象工厂模式则是不需要关心构建过程，只关心什么产品由什么工厂生产即可。

而建造者模式则是要求按照指定的蓝图建造产品，它的主要目的是通过组装零配件而产生一个新产品。

#### 代理模式和装饰模式对比

装饰模式就是代理模式的一个特殊应用，两者的共同点是都具有相同的接口，不同点则是代理模式着重对代理过程的控制，而装饰模式则是对类的功能进行加强或减弱，它着重类的功能变化。

##### 代理模式

是把当前的行为或功能委托给其他对象执行，代理类负责接口限定：是否可以调用真实角色，以及是否对发送到真实角色的消息进行变形处理，它不对被主题角色（也就是被代理类）的功能做任何处理，保证原汁原味的调用。

代理模式使用到极致开发就是AOP，是Spring架构开发必然要使用到的技术，它就是使用了代理和反射的技术。

##### 装饰模式

是在要保证接口不变的情况下加强类的功能，它保证的是被修饰的对象功能比原始对象丰富（当然，也可以减弱），但不做准入条件判断和准入参数过滤，如是否可以执行类的功能，过滤输入参数是否合规等，这不是装饰模式关心的。

#### 命令模式和策略模式对比

命令模式和策略模式的类图确实很相似，只是命令模式多了一个接收者（Receiver）角色。它们虽然同为行为类模式，但是两者的区别还是很明显的。

策略模式的意图是封装算法，它认为“算法”已经是一个完整的、不可拆分的原子业务（注意这里是原子业务，而不是原子对象），即其意图是让这些算法独立，并且可以相互替换，让行为的变化独立于拥有行为的客户。

而命令模式则是对动作的解耦，把一个动作的执行分为执行对象（接收者角色）、执行行为（命令角色），让两者相互独立而不相互影响。

策略模式和命令模式相似，特别是命令模式退化时，比如无接收者（接收者非常简单或者接收者是一个Java的基础操作，无需专门编写一个接收者），在这种情况下，命令模式和策略模式的类图完全一样。

##### 关注点不同

策略模式关注的是算法替换的问题，一个新的算法投产，旧算法退休，或者提供多种算法由调用者自己选择使用，算法的自由更替是它实现的要点。换句话说，策略模式关注的是算法的完整性、封装性，只有具备了这两个条件才能保证其可以自由切换。

命令模式则关注的是解耦问题，如何让请求者和执行者解耦是它需要首先解决的，解耦的要求就是把请求的内容封装为一个一个的命令，由接收者执行。由于封装成了命令，就同时可以对命令进行多种处理，例如撤销、记录等。

##### 角色功能不同

策略模式中的抽象算法和具体算法与命令模式的接收者非常相似，但是它们的职责不同。策略模式中的具体算法是负责一个完整算法逻辑，它是不可再拆分的原子业务单元，一旦变更就是对算法整体的变更。

而命令模式则不同，它关注命令的实现，也就是功能的实现。例如我们在分支中也提到接收者的变更问题，它只影响到命令族的变更，对请求者没有任何影响，从这方面来说，接收者对命令负责，而与请求者无关。命令模式中的接收者只要符合六大设计原则，完全不用关心它是否完成了一个具体逻辑，它的影响范围也仅仅是抽象命令和具体命令，对它的修改不会扩散到模式外的模块。

当然，如果在命令模式中需要指定接收者，则需要考虑接收者的变化和封装，例如一个老顾客每次吃饭都点同一个厨师的饭菜，那就必须考虑接收者的抽象化问题。

##### 使用场景不同

策略模式适用于算法要求变换的场景，而命令模式适用于解耦两个有紧耦合关系的对象场合或者多命令多撤销的场景。

#### 策略模式和桥梁模式对比

策略模式是一个行为模式，旨在封装一系列的行为。而桥梁模式则是解决在不破坏封装的情况下如何抽取出它的抽象部分和实现部分，它的前提是不破坏封装，让抽象部分和实现部分都可以独立地变化。

简单来说，策略模式是使用继承和多态建立一套可以自由切换算法的模式，桥梁模式是在不破坏封装的前提下解决抽象和实现都可以独立扩展的模式。桥梁模式必然有两个“桥墩”——抽象化角色和实现化角色，只要桥墩搭建好，桥就有了，而策略模式只有一个抽象角色，可以没有实现，也可以有很多实现。

还是很难区分，是吧？多想想两者的意图，就可以理解为什么要建立两个相似的模式了。我们在做系统设计时，可以不考虑到底使用的是策略模式还是桥梁模式，只要好用，能够解决问题就成，“不管黑猫白猫，抓住老鼠的就是好猫”。

## 文档参考

大话设计模式文档下载

<https://blog.csdn.net/hemingyang97/article/details/82025608>

设计模式-在线手册教程

<http://www.php.cn/course/58.html>

ProcessOn设计模式

<https://www.processon.com/view/5b503678e4b0f8477d8a22fa#map>

ProcessOn设计模式之间的关系

<https://www.processon.com/view/5a371fa8e4b0dce99f02f005>

Java设计模式：23种设计模式全面解析（超级详细）

<http://c.biancheng.net/design_pattern>

【设计模式】GoF设计模式学习总结

<https://www.cnblogs.com/chenpi/p/5222597.html>

24种设计模式及案例

<https://blog.csdn.net/WiKi_Su/article/details/80263967>

23种设计模式总结

<https://www.cnblogs.com/tongkey/p/7170826.html>

设计模式：（五）相似模式比较

<https://www.jianshu.com/p/7475a65e261e>