# 设计模式－可复用的面向对象软件元素

## 面向对象程序设计

### 程序设计语言的发展回顾

#### 机器指令和汇编语言

依赖于特定机器、CPU。。。。优化到字节和机器指令时间

Y=y+3

Add ax,3

Inc ax

Inc ax

Inc ax

#### **高级程序设计语言**

FORTRAN, ALGO60, PASCAL, BASIC, C。。。。

goto

结构化程序设计是高级语言程序设计的主流，结构化程序设计又称为[面向过程](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E8%BF%87%E7%A8%8B" \t "https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1/_blank)的程序设计。在面向过程程序设计中，问题被看作一系列需要完成的任务，函数（在此泛指例程、函数、过程）用于完成这些任务，解决问题的焦点集中于函数。其中函数是面向过程的，即它关注如何根据规定的条件完成指定的任务。

#### 面向对象程序设计语言

越来越多的大型系统应用开发，涉及处理复杂事物，在软件设计时需要首先考虑的是事物、事物之间的关系及合作机制，而不是具体事务处理过程。需要在面向“对象”层次上构建系统。C++, JAVA。。。。

在大型软件系统开发过程中，面向对象设计已经是主流。而有两种技术是软件工程师追求：重构和设计模式。

软件重构(Refactoring)：软件工程师对软件内容（代码）完美化的追求

设计模式(Design Patterns)：软件工程师对软件架构完美化的追求

有人把以下两本书<<重构>>与<<设计模式>>并列为“软件工程师的圣经（葵花宝典）”

可复用技术是面向对象软件开发人员梦想，期望充分利用原有的代码，将原有代码应用到升级的或者新的系统中。

软件架构层面的设计模式的引入，可以让项目代码的骨架更加的合理、灵活，方便扩展、修改、重用。

#### **脚本类语言，新一代语言**

Python?

### 面向对象程序设计中的基本概念

**C++的例子：**

#include <iostream>

using namespace std;

class Shape

{

public:

virtual double getArea()=0;

void setWidth(double w)

{

width = w;

}

void setHeight(double h)

{

height = h;

}

protected:

double width;

double height;

};

class Rectangle: public Shape

{

public:

double getArea()

{

return(width \* height);

}

};

class Triangle: public Shape

{

public:

double getArea()

{

return(width \* height \* 0.5);

}

};

int main()

{

Rectangle Rect;

Triangle Tri;

Rect.setWidth(5);

Rect.setHeight(7);

cout << "Total Rectangle area:" << Rect.getArea()<<endl;

Tri.setWidth(5);

Tri.setHeight(7);

cout << "Total Triangle area:" << Tri.getArea() << endl;

return 0;

}

/////换一种主函数

int main()

{

Shape \* pShape;

pShape= new Rectangle();

pShape->setWidth(5);

pShape->setHeight(7);

cout << "Total Rectangle area:" << pShape->getArea()<<endl;

delete pShape;

pShape= new Triangle();

pShape->setWidth(5);

pShape->setHeight(7);

cout << "Total Triangle area:" << pShape->getArea() << endl;

delete pShape;

return 0;

}

**Java的例子：**

**/////Shape.Java**

package Shapes;

public abstract class Shape {

public abstract double getArea();

public void setWidth(double w)

{

width = w;

}

public void setHeight(double h)

{

height = h;

}

protected double width;

protected double height;

}

**/////Rectangle.Java**

package Shapes;

public class Rectangle extends Shape

{

@Override

public double getArea()

{

return(width \* height);

}

}

**/////Triangle.Java**

package Shapes;

public class Triangle extends Shape

{

@Override

public double getArea()

{

return(width \* height \* 0.5);

}

}

**/////DesignPatternsDemo.Java**

package DesignPatternsDemo;

import Shapes.\*;

public class DesignPatternsDemo{

public static void main(String[] args) {

Rectangle Rect=new Rectangle();

Triangle Tri=new Triangle();

Rect.setWidth(5);

Rect.setHeight(7);

System.out.println("Total Rectangle area:"+Rect.getArea());

Tri.setWidth(5);

Tri.setHeight(7);

System.out.println("Total Triangle area:"+Tri.getArea());

}

}

**/////换一种主函数DesignPatternsDemo.Java**

package DesignPatternsDemo;

import Shapes.\*;

public class DesignPatternsDemo {

public static void main() {

Shape theShape;

theShape=new Rectangle();

theShape.setWidth(5);

theShape.setHeight(7);

System.out.println("Total Rectangle area:"+theShape.getArea());

theShape=new Triangle();

theShape.setWidth(5);

theShape.setHeight(7);

System.out.println("Total Triangle area:"+theShape.getArea());

}

}

运行结果输出：

Total Rectangle area:35.0

Total Triangle area:17.5

#### 对象（Object)

可以对其做事情的一些东西。一个对象有状态、行为和标识三种属性。

#### 类（class)

一个共享相同结构和行为的对象的集合。类（Class）定义了一件事物的抽象特点。通常来说，类定义了事物的属性和它可以做到的（它的行为）。举例来说，“狗”这个类会包含狗的一切基础特征，例如它的孕育、毛皮颜色和吠叫的能力。类可以为程序提供模版和结构。一个类的方法和属性被称为“成员”。

#### 封装（encapsulation)：

第一层意思：将数据和操作捆绑在一起，创造出一个新的类型的过程。第二层意思：将接口与实现分离的过程。

#### **继承（inherit）**

类之间的关系，在这种关系中，一个类共享了一个或多个其他类定义的结构和行为。继承描述了类之间的“是一种”关系。子类可以对基类的行为进行扩展、覆盖、重定义。

#### **多态（Polymorphic）**

类型理论中的一个概念，一个名称可以表示很多不同类的对象，这些类和一个共同超类有关。因此，这个名称表示的任何对象可以以不同的方式响应一些共同的操作集合。

#### **组合和**聚合

既是类之间的关系也是对象之间的关系。在这种关系中一个对象或者类包含了其他的对象和类。组合和聚合描述了“有”关系，它们的不同之处在于前者是一个对象有另一些对象的实体，后者是一个对象有另一些对象的引用（指针、名字）。**举例**学院、班级各学生之间的关系

#### **消息传递**

指的是一个对象调用了另一个对象的方法（或者称为成员函数）。

#### **方法**

也称为成员函数，是指对象上的操作，作为类声明的一部分来定义。方法定义了可以对一个对象执行那些操作。

#### **接口和抽象类**

设计抽象类（通常称为 ABC）的目的，是为了给其他类提供一个可以继承的适当的基类。抽象类不能被用于实例化对象，如果一个 ABC 的子类需要被实例化，则必须实现每个虚函数。

接口：接口是一套规范。一个比抽象类更抽象的类。

接口中只能写抽象方法。

接口中没有构造函数

C++中没有接口的概念，通过设计只包含纯虚函数的抽象类来实现接口的定义。

如果一个类实现了一个接口，那么我们要重写里面所有的方法

类可以实现多个接口，但是要重写实现的接口中的所有方法

接口可以多继承用逗号进行隔开

如果我们实现的接口是多继承的，那么就要实现接口中所有的功能（接口里面写的就是一个一个的功能）

如果两个接口里面有两个方法名相同的抽象方法，那么在继承的时候只需要重写一次

接口的作用：

1、增加的代码的可拓展性

2、降低代码的耦合程度（紧密程度）

接口在设计模式中经常用到，是一个非常重要的概念。

**Java的例子：**

**/////Shapes.Java**

package Shapes;

public interface Shape {

public double getArea();

public void setWidth(double w);

public void setHeight(double h);

}

**/////Rectangle .Java**

package Shapes;

public class Rectangle implements Shape

{

protected double width;

protected double height;

@Override

public double getArea()

{

return(width \* height);

}

@Override

public void setWidth(double w)

{

width = w;

}

@Override

public void setHeight(double h)

{

height = h;

}

}

**/////Triangle .Java**

package Shapes;

public class Triangle implements Shape

{

protected double width;

protected double height;

@Override

public double getArea()

{

return(width \* height \* 0.5);

}

@Override

public void setWidth(double w)

{

width = w;

}

@Override

public void setHeight(double h)

{

height = h;

}

}

**/////DesignPatternsDemo.Java**

package DesignPatternsDemo;

import Shapes.\*;

public class DesignPatternsDemo {

public static void main() {

Shape theShape;

theShape=new Rectangle();

theShape.setWidth(5);

theShape.setHeight(7);

System.out.println("Total Rectangle area:"+theShape.getArea());

theShape=new Triangle();

theShape.setWidth(5);

theShape.setHeight(7);

System.out.println("Total Triangle area:"+theShape.getArea());

}

}

### 面向对象程序设计的特征

1、封装性：

封装是指将一个计算机系统中的数据以及与这个数据相关的一切操作语言（即描述每一个对象的属性以及其行为的程序代码）组装到一起，一并封装在一个有机的实体中，把它们封装在一个“模块”中，也就是一个类中，为软件结构的相关部件所具有的模块性提供良好的基础。在面向对象技术的相关原理以及程序语言中，封装的最基本单位是对象，而使得软件结构的相关部件的实现“高内聚、低耦合”的“最佳状态”便是面向对象技术的封装性所需要实现的最基本的目标。对于用户来说，对象是如何对各种行为进行操作、运行、实现等细节是不需要刨根问底了解清楚的，用户只需要通过封装外的通道对计算机进行相关方面的操作即可。大大地简化了操作的步骤，使用户使用起计算机来更加高效、更加得心应手。

2、继承性：

继承性是面向对象技术中的另外一个重要特点，其主要指的是两种或者两种以上的类之间的联系与区别。继承，顾名思义，是后者延续前者的某些方面的特点，而在面向对象技术则是指一个对象针对于另一个对象的某些独有的特点、能力进行复制或者延续。如果按照继承源进行划分，则可以分为单继承（一个对象仅仅从另外一个对象中继承其相应的特点）与多继承（一个对象可以同时从另外两个或者两个以上的对象中继承所需要的特点与能力，并且不会发生冲突等现象）；如果从继承中包含的内容进行划分，则继承可以分为四类，分别为取代继承（一个对象在继承另一个对象的能力与特点之后将父对象进行取代）、包含继承（一个对象在将另一个对象的能力与特点进行完全的继承之后，又继承了其他对象所包含的相应内容，结果导致这个对象所具有的能力与特点大于等于父对象，实现了对于父对象的包含）、受限继承、特化继承。

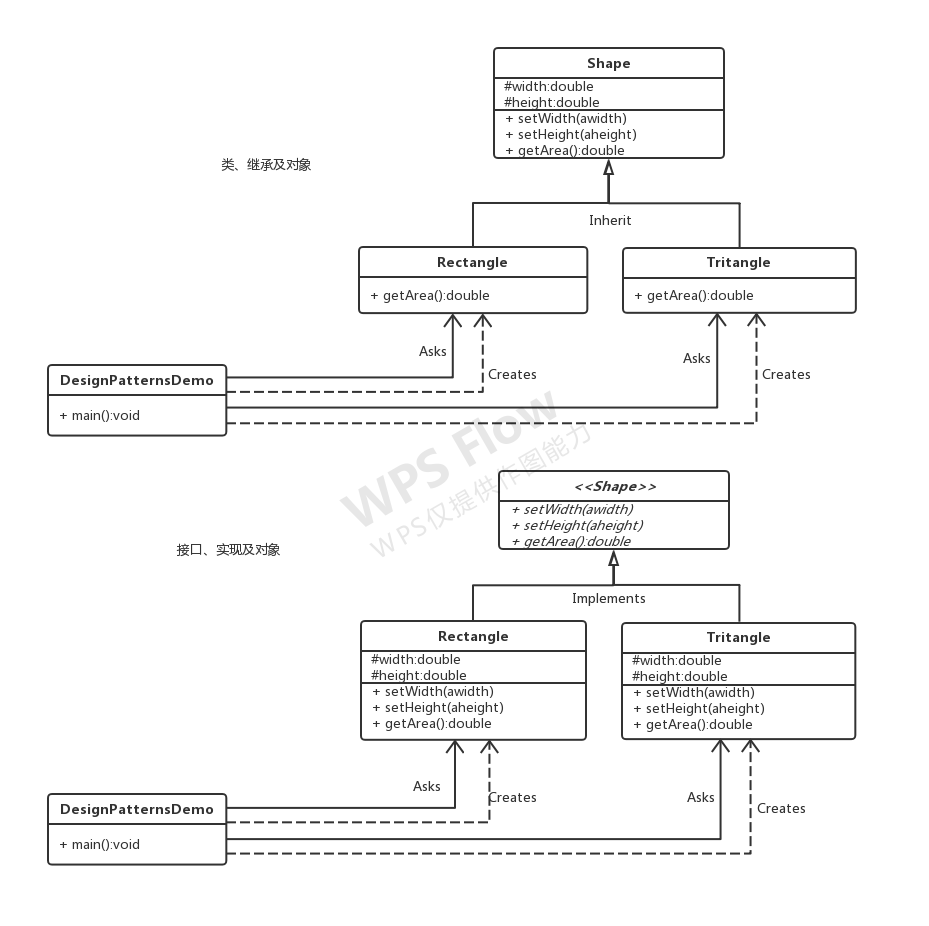
3、多态性：

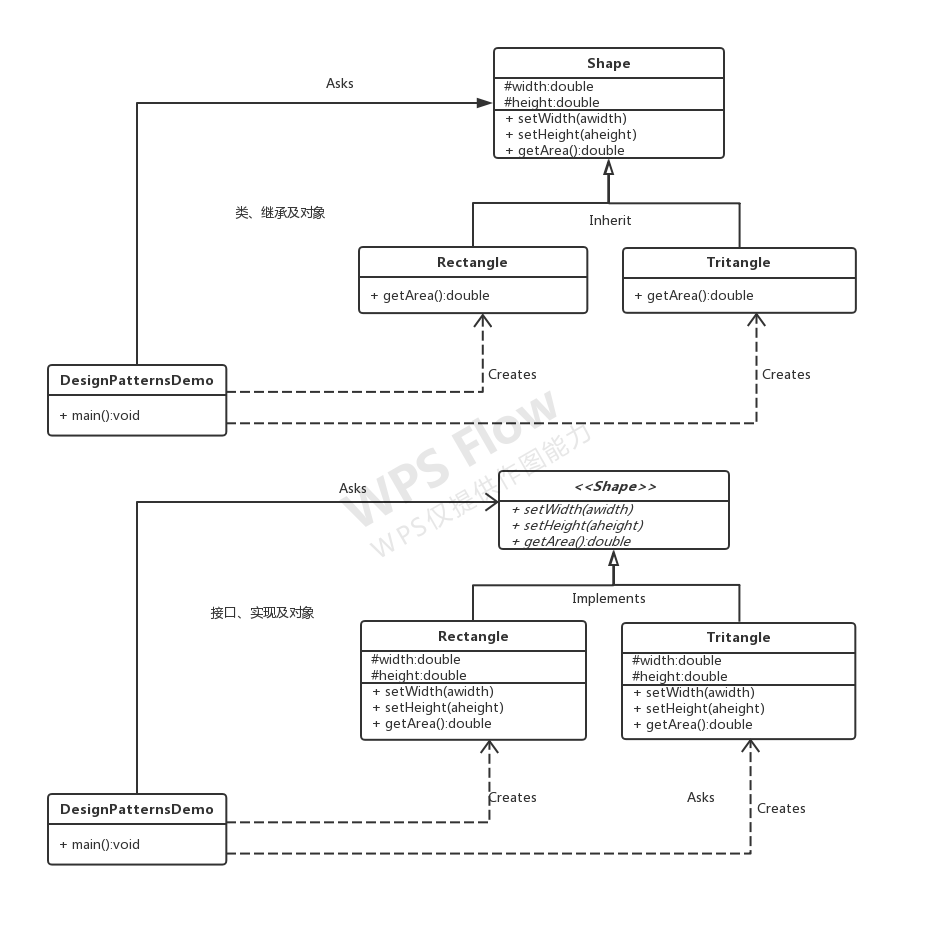
从宏观的角度来讲，多态性是指在面向对象技术中，当不同的多个对象同时接收到同一个完全相同的消息之后，所表现出来的动作是各不相同的，具有多种形态；从微观的角度来讲，多态性是指在一组对象的一个类中，面向对象技术可以使用相同的调用方式来对相同的函数名进行调用，即便这若干个具有相同函数名的函数所表示的函数是不同的。

### 图形化设计工具

UML

[UML](https://baike.baidu.com/item/UML%E5%9B%BE/6963758?fr=aladdin)[统一建模语言](https://baike.baidu.com/item/%E7%BB%9F%E4%B8%80%E5%BB%BA%E6%A8%A1%E8%AF%AD%E8%A8%80/3160571" \t "https://baike.baidu.com/item/UML%E5%9B%BE/_blank)





### **面向对象程序设计的原则**

#### **1、开闭原则（Open Close Principle）**

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：

项目中划分出的模块

类与接口

方法

**开闭原则的含义是：**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。

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。

1. 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。

1. 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。

**开闭原则的实现方法:**

*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。

#### **2、里氏代换原则（Liskov Substitution Principle）**

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。里氏代换原则是面向对象设计的基本原则之一。

里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。

LSP 是继承复用的基石，它反映了基类与子类之间的关系，只有当派生类（子类）可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

**里氏替换原则的主要作用如下。**

里氏替换原则是实现开闭原则的重要方式之一。

它克服了继承中重写父类造成的可复用性变差的缺点。

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。

**里氏替换原则的实现方法**

*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，例如设计一个更一般的基类。

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。

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

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++ Report 上发表的文章。

依赖倒置原则的原始定义为：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 should not 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。

**依赖倒置原则的主要作用如下：**

依赖倒置原则可以降低类间的耦合性。

依赖倒置原则可以提高系统的稳定性。

依赖倒置原则可以减少并行开发引起的风险。

依赖倒置原则可以提高代码的可读性和可维护性。

**依赖倒置原则的实现方法：**

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。

每个类尽量提供接口或抽象类，或者两者都具备。

变量的声明类型尽量是接口或者是抽象类。

任何类都不应该从具体类派生。

使用继承时尽量遵循里氏替换原则。

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

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。

**接口隔离原则的优点：**

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。

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。

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。

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。

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。

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。

**接口隔离原则的实现方法：**

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。

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。

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。

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。要深入了解业务逻辑。

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。

#### **5、迪米特法则，又称最少知识原则（Demeter Principle）**

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。

迪米特法则的定义是：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”是指：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。

**迪米特法则的优点：**

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。

降低了类之间的耦合度，提高了模块的相对独立性。

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。

**迪米特法则的实现方法：**

从迪米特法则的定义和特点可知，它强调以下两点：

从依赖者的角度来说，只依赖应该依赖的对象。

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。

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。

在类的结构设计上，尽量降低类成员的访问权限。

在类的设计上，优先考虑将一个类设置成不变类。

在对其他类的引用上，将引用其他对象的次数降到最低。

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）。

谨慎使用序列化（Serializable）功能。

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

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。

**合成复用原则的重要性：**

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。

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。

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。

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。

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。

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。

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。

合成复用原则的实现方法：

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。

### 设计模式的类型

１、创建型模式－创建一个或一组对象，who, what,when, how

２、结构型模式－描述类、对象间的关系

３、行为型模式－描述类、对象及对象组的协同合作行为

４、其它类型模式－特殊应用场景下的常见问题

## 创建型模式

创建对象是面向对象程序设计最基本的操作。在C++中，通过定义类变量就可直接创建对象。也可以定义类的指针，然后由指针指向由new运算符所创建的对象。在Java中，定义类变量时只是定义了一个引用（给出一个名字）。之后必须使用 new 运算符实例化对象。

在大型系统中，往往不是由使用者创建对象，也不是什么时候使用什么时候创建对或者程序加载运行时就创建对象，有时对象创建者可能都不知道所创建的对象具体是什么类的对象。

创建型模式解决对象创建时所遇到的问题。这些问题包括由谁创建对象(who)、创建什么类的对象(what)、什么时候创建对象(when)及如何创建对象(how)。

这些创建型模式提供了一种在创建对象的同时隐藏创建逻辑的方式，而不是使用 new 运算符直接实例化对象。这使得程序在判断针对某个给定实例需要创建哪些对象时更加灵活。

### **工厂方法模式（Factory Method Pattern）**

在现实生活中，产品的使用者往往不是生产者，在软件系统中也是一样，这里的产品即对象。一方面是一个对象的存在往往不是只为一个客户（也是对象）服务的，可能有多个客户需要使用该同一个对象，因而不是所有使用对象的客户都需要创建（生产）对象；另一方面，对象的使用者可能并不知道对象应该如何创建（生产），甚至不知道对象的具体类型（确切地说是使用者不必知道这些，只须知道如何使用即可。有时一个对象的生产过程不是简单了用new运算符就能完成的，可能还有更特殊的要求或生产环节，这对于仅仅需要知道如何使用的客户来说是不必要的）；第三方面，随着大型系统的升级换代，或者软件系统开发过程的推进，往往会引进新的产品供原客户使用，而且希望这些新产品的引入不对原代码产生影响，即：不改变原系统代码。

在软件开发中能否做到软件对象的生产和使用相分离呢？能否在满足“开闭原则”的前提下，客户随意增删或改变对软件相关对象的使用呢？这就是本节要讨论的问题。

**工厂方法（FactoryMethod）模式的定义：**又称工厂模式、多态工厂模式和虚拟构造器模式。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交到具体子工厂类当中，即由子工厂类来决定应该实例化（创建）哪一个类（产品对象），使系统在不修改原来代码的情况下引进新的产品。

**模式的结构：**

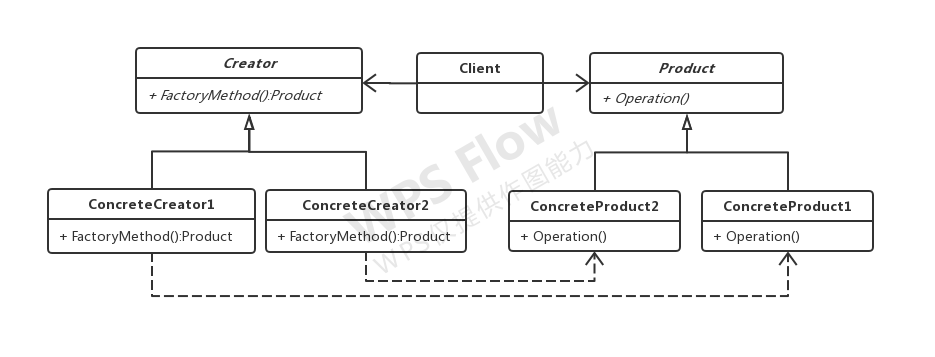
工厂方法模式由抽象工厂、具体工厂、抽象产品和具体产品等4个要素构成。

抽象工厂（Creator）：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 FactoryMethod() 来创建产品。

具体工厂（ConcreteCreator）：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。



**工厂方法模式的主要优点有：**

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；

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；

**其缺点是：**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。

**模式的应用场景：**

客户只知道创建产品的工厂名，而不知道具体的产品名。

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。

客户不关心创建产品的细节，只关心产品的品牌。

**应用实例：** 您需要一辆汽车，可以直接从工厂里面提货，而不用去管这辆汽车是怎么做出来的，以及这个汽车里面的具体实现。

**实现：**

创建一个接口:

//////Vehicle.java

package Vehicles;

public interface Vehicle{

void drive();

}

步骤 2

创建实现接口的实体类。

//////Truck.java

package Vehicles;

public class Truck implements Vehicle{

@Override

public void drive() {

System.out.println("Inside Truck::drive() method.");

}

}

//////Car.java

package Vehicles;

public class Car implements Vehicle{

@Override

public void drive() {

System.out.println("Inside Car::drive() method.");

}

}

//////Tractor.java

package Vehicles;

public class Tractor implements Vehicle{

@Override

public void drive() {

System.out.println("Inside Tractor::drive() method.");

}

}

步骤 3

创建一个工厂接口，生成一个实现Vehicle接口的实体类的对象。

//////VehicleFactory.java

package VehicleFactories;

import Vehicles.\*;

import Vehicles.Vehicle;public interface VehicleFactory {

//使用 getVehicle 方法获取Vehicle类型的对象

Vehicle getVehicle();

}

//////TruckFactory.java

package VehicleFactories;

import Vehicles.\*;

public class TruckFactory implements VehicleFactory {

@Override

public Vehicle getVehicle(){

return new Truck();

}

}

//////CarFactory.java

package VehicleFactories;

import Vehicles.\*;

public class CarFactory implements VehicleFactory {

@Override

public Vehicle getVehicle(){

return new Car();

}

}

//////TractorFactory.java

package VehicleFactories;

import Vehicles.\*;

public class TractorFactory implements VehicleFactory {

@Override

public Vehicle getVehicle(){

return new Tractor();

}

}

步骤 4

使用上述工厂，获取实体类的对象。

FactoryMethodPattern.java

import VehicleFactories.\*;

import Vehicles.Vehicle;

public class FactoryMethodPattern {

public static void main() {

VehicleFactory theTruckFactory=new TruckFactory();

VehicleFactory theCarFactory=new CarFactory();

VehicleFactory theTractorFactory=new TractorFactory();

//获取 Truck的对象，并调用它的 drive 方法

Vehicle vehicle1 = theTruckFactory.getVehicle();

vehicle1 .drive();

//获取 Car的对象，并调用它的 drive 方法

Vehicle vehicle2 = theCarFactory.getVehicle();

vehicle2 .drive();

//获取 Tractor的对象，并调用它的 drive 方法

Vehicle vehicle3 = theTractorFactory .getVehicle();

vehicle3 .drive();

}

}

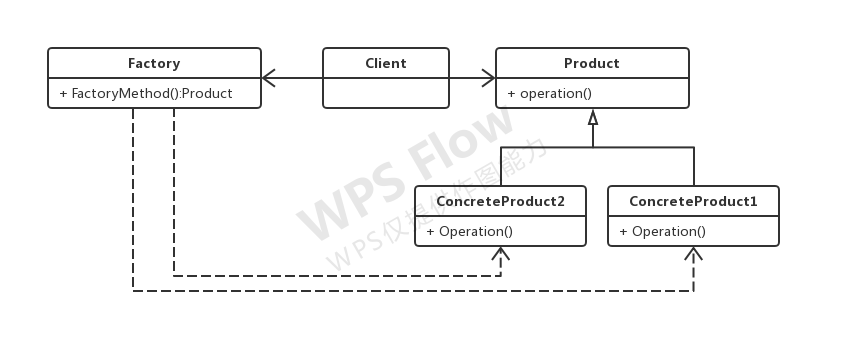
执行程序，输出结果：

Inside Truck::drive() method.

Inside Car::drive() method.

Inside Tractor::drive() method.

**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。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。



例如，如果只有TRUCK、CAR和TRACTOR三种车辆，不会有其它车辆时，修改上述实现的步骤3、4。

步骤 3

创建一个简单工厂，生成一个实现Vehicle接口的实体类的对象。

//////SimpleVehicleFactory.java

package VehicleFactories;

import Vehicles.\*;

public class SimpleVehicleFactory {

//使用 getVehicle 方法获取形状类型的对象

public Vehicle getVehicle(String vehicleType){

if(vehicleType== null){

return null;

}

if(vehicleType.equalsIgnoreCase("TRUCK")){

return new Truck();

} else if(vehicleType.equalsIgnoreCase("CAR")){

return new Car();

} else if(vehicleType.equalsIgnoreCase("TRACTOR")){

return new Tractor();

}

return null;

}

}

步骤 4

使用上述工厂，获取实体类的对象。

FactoryMethodPattern.java

package factorymethodpattern;

import VehicleFactories.\*;

import Vehicles.Vehicle;

public class FactoryMethodPattern{

public static void main() {

SimpleVehicleFactory theVehicleFactory= new SimpleVehicleFactory();

//获取 Truck的对象，并调用它的 drive 方法

Vehicle vehicle1 = theVehicleFactory .getVehicle("TRUCK");

vehicle1 .drive();

//获取 Car的对象，并调用它的 drive 方法

Vehicle vehicle2 = theVehicleFactory .getVehicle("CAR");

vehicle2 .drive();

//获取 Tractor的对象，并调用它的 drive 方法

Vehicle vehicle3 = theVehicleFactory .getVehicle("TRACTOR");

vehicle3 .drive();

}

}

### **抽象工厂模式（Abstract Factory Pattern）**

前面介绍的工厂方法模式中考虑的是一类产品的生产，如汽车厂只生产汽车。

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，如电脑公司既生产台式机又笔记本电脑，家具厂既生产桌子又生产椅子。

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，如台式机和笔记本电脑。

**抽象工厂（AbstractFactory）模式的定义：**是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。

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。

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。

**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:**

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。

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。

**其缺点是：**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。

**模式的结构**

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。

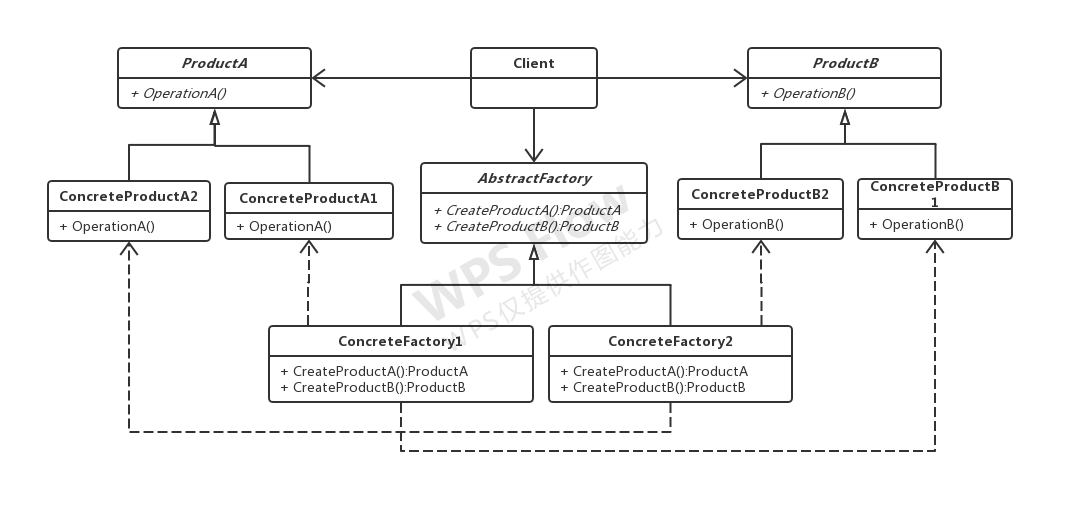
抽象工厂模式的主要角色如下。

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 CreateProduct()，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。



**模式的应用场景**

抽象工厂模式通常适用于以下场景：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，如智能设备公司生产的平板电脑、手机、智能手表等。

系统中有多个产品族，但每次只使用其中的某一族产品。如有人只喜欢使用苹果公司的系列产品，有人喜欢使用华为公司的系列产品。

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。

**应用实例：** 汽车生产厂家在生产汽车时，需要向其它厂商订购汽车配套设备，如车载电器。汽车生产厂家提出需求标准（或者采用已有公共标准），向某个车载电器厂商订购车载电器，包括汽车音响、汽车导航仪等。而能提供这些设备的厂商有多家，汽车生产厂选择其中一家。

**实现：**

现编！！！

一、产品接口

//MobilePhone .java

public interface MobilePhone {

void ShowPhone();

}

//Pad.java

public interface Pad {

void ShowPad();

}

二、工厂接口

//ITFactory.java

public interface ITFactory {

MobilePhone getPhone();

Pad getPad();

}

三、产品

１、Apple产品

//iPhone .java

public class iPhone implements MobilePhone{

@Override

public void ShowPhone() {

System.out.println("This is an iPhone!");

}

}

//iPad .java

public class iPad implements Pad{

@Override

public void ShowPad() {

System.out.println("This ia an iPad!");

}

}

２、华为产品

//Mate.java

public class Mate implements MobilePhone{

@Override

public void ShowPhone() {

System.out.println("This is a Mate!");

}

}

//MatePad .java

public class MatePad implements Pad{

@Override

public void ShowPad() {

System.out.println("This is a MatePad !");

}

}

四、具体工厂

//Foxcom.java

public class Foxcom implements ITFactory{

@Override

public MobilePhone getPhone() {

return new iPhone();

}

@Override

public Pad getPad() {

return new iPad();

}

}

//HuaWei .java

public class HuaWei implements ITFactory{

@Override

public MobilePhone getPhone() {

return new Mate();

}

@Override

public Pad getPad() {

return new MatePad();

}

}

//AbstractFractoryPattern .java

public class AbstractFractoryPattern {

public static void main(String[] args) {

ITFactory aFactory = new HuaWei();

MobilePhone myPhone;

Pad myPad;

myPhone=aFactory.getPhone();

myPad=aFactory.getPad();

myPhone.ShowPhone();

myPad.ShowPad();

aFactory = new Foxcom);

myPhone=aFactory.getPhone();

myPad=aFactory.getPad();

myPhone.ShowPhone();

myPad.ShowPad();

}

}

**注意事项：**

抽象工厂模式的扩展有一定的“开闭原则”倾斜性：

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**。

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**。

### **单例模式（Singleton Pattern）**

在有些系统中，为了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。大多数情况下，资源管理类事物都需要只创建一个实例。

**单例（Singleton）模式的定义：**又称为单件模式，指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。

**单例模式有 3 个特点：**

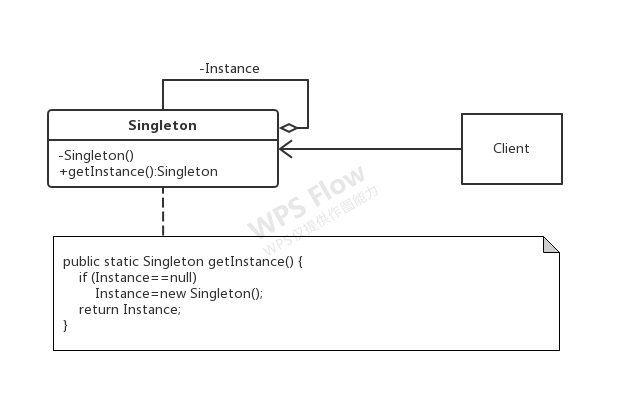
单例类只有一个实例对象；

该单例对象必须由单例类自行创建，任何外部事物都不能创建该类对象；

单例类对外提供一个访问该单例的全局访问点；

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。

**单例模式的结构：**

****

**单例模式的应用场景:**

在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;

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;

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。

**注意：**如果编写的是多线程程序，则Instance 要修饰成static volatile，getInstance() 要修饰成static synchronized，否则将存在线程非安全的问题。加上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，这是此单例模式的缺点。

**变成饿汉单例模式：**

public class HungrySingleton

{

private static final HungrySingleton instance=new HungrySingleton();

private HungrySingleton(){}

public static HungrySingleton getInstance()

{

return instance;

}

}

**具体实例：**

//Singleton.java

public class Singleton {

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

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

public static synchronized Singleton getInstance()

{

if(instance==null)

instance=new Singleton();

return instance;

}

public void SayHello() {

System.out.println("Hello! It's me, the Singleton! My varible x="+x);

}

public int x=1;

}

//SingletonPattern .java

public class SingletonPattern {

public static void main(String[] args) {

Singleton Ob1=Singleton.getInstance();

Ob1.x=2;

System.out.print("Ob1 Says: ");

Ob1.SayHello();

Singleton Ob2=Singleton.getInstance();

System.out.print("Ob2 Says: ");

Ob2.SayHello();

Ob2.x=5;

System.out.print("Ob1 Says: ");

Ob1.SayHello();

System.out.print("Ob2 Says: ");

Ob2.SayHello();

}

}

运行结果：

Ob1 Says: Hello! It's me, the Singleton! My varible x=2

Ob2 Says: Hello! It's me, the Singleton! My varible x=2

Ob1 Says: Hello! It's me, the Singleton! My varible x=5

Ob2 Says: Hello! It's me, the Singleton! My varible x=5

### **建造者模式（Builder Pattern）**

前面几个模式讨论的是由谁（ＷＨＯ）创建对象，而没有讨论如何（ＨＯＷ）创建对象。当一个被创建的对象较为复杂时，单纯用new 运算符就解决不了问题了，需要用专门的工厂负责，这也是我们前面讨论工厂方法模式、抽象工厂模式中工厂的作用。

比如，汽车的创建。一台汽车由多个部件组成，车架、发动机、方向盘、轮胎等等，如何选取不同的组件？如何将这些组件组装成最终车辆？这不是也不应是对象的使用者需要考虑的问题。

一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，而建造者模式可以很好地描述该类产品的创建。

**建造者（Builder）模式的定义：**指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。

**该模式的主要优点如下：**

各个具体的建造者相互独立，有利于系统的扩展。

客户端不必知道产品内部组成的细节，便于控制细节风险。

**其缺点如下：**

产品的组成部分必须相同，这限制了其使用范围。

如果产品的内部变化复杂，该模式会增加很多的建造者类。

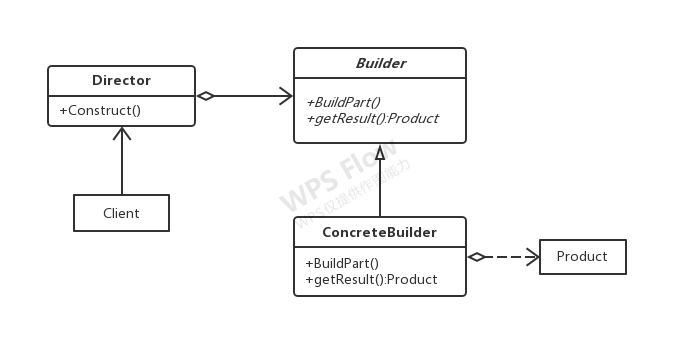
**建造者（Builder）模式的结构：**

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。



**模式的应用场景**

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：

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。

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（形态）是独立的。

**模式的实现:**

/////Product.java

public class Product {

private int PartA;

private int PartB;

private int PartC;

public void SetPartA(int a) {

PartA=a;

}

public void SetPartB(int b) {

PartB=b;

}

public void SetPartC(int c) {

PartC=c;

}

public void Show() {

System.out.println("PartA="+PartA+"; PartB="+PartB+"; PartA="+PartC);

}

}

/////Builder.java

public interface Builder {

void BuildPartA();

void BuildPartB();

void BuildPartC();

Product getResult();

}

/////ConcreteBuilder1.java

public class ConcreteBuilder1 implements Builder {

private Product product=new Product();

@Override

public void BuildPartA() {

product.SetPartA(1);

}

@Override

public void BuildPartB() {

product.SetPartB(2);

}

@Override

public void BuildPartC() {

product.SetPartC(3);

}

@Override

public Product getResult() {

return product;

}

}

/////ConcreteBuilder2.java

public class ConcreteBuilder2 implements Builder {

private Product product=new Product();

@Override

public void BuildPartA() {

product.SetPartA(10);

}

@Override

public void BuildPartB() {

product.SetPartB(20);

}

@Override

public void BuildPartC() {

product.SetPartC(30);

}

@Override

public Product getResult() {

return product;

}

}

/////Director.java

public class Director {

private Builder builder;

public Director(Builder builder)

{

this.builder=builder;

}

public void SetDirector(Builder builder)

{

this.builder=builder;

}

public Product construct()

{

builder.BuildPartA();

builder.BuildPartB();

builder.BuildPartC();

return builder.getResult();

}

}

/////ProductPattern.java

public class BuilerPattern {

public static void main(String[] args) {

Builder builder=new ConcreteBuilder1();

Director director=new Director(builder);

Product product=director.construct();

product.Show();

builder=new ConcreteBuilder2();

director.SetDirector(builder);

product=director.construct();

product.Show();

}

}

**运行结果：**

PartA=1; PartB=2; PartA=3

PartA=10; PartB=20; PartA=30

**注意事项：**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。

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。建造者可以使用工厂方法模式或抽象工厂模式创建部件；工厂方法模式或抽象工厂模式也可以使用建造者模式生产产品。

### **原型模式（Prototype Pattern）**

工厂方法模式和抽象工厂模式描述了由工厂在工厂方法中创建对象的形式；

建造者模式描述了由指定的建造者进行对象搭建的方法；

单例模式描述了由类自身创建对象保证只有唯一一个对象可以存在。

原型模式（Prototype Pattern）描述了由类对象复制自身而生成对象的方法。

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。

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。这种模式实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。

另外，有时会向已完成的系统中添加新的成员，该成员实现了已知接口。我们希望在不改变原系统的情况下自由使用该成员。如一些游戏的插件，高级游戏玩家喜欢自己编写插件加入到游戏当中。这些插件符合原系统接口规范，但并不需要原系统知道这些新插件的类名字。可以用原型模式实现。

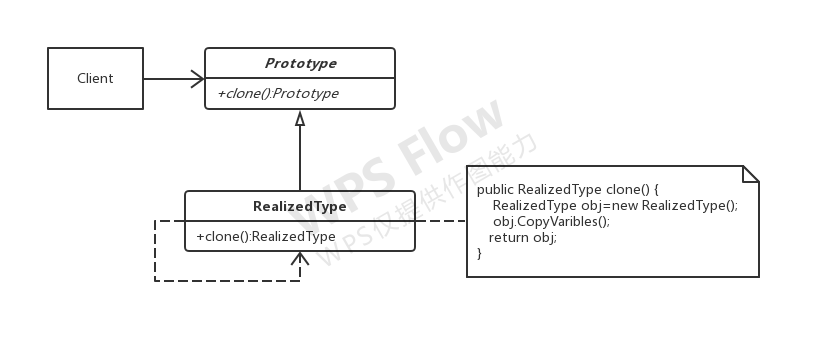
**原型（Prototype）模式的定义：**用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型相同或相似的新对象。在这里，原型实例指定了要创建的对象的种类。用这种方式创建对象非常高效，根本无须知道对象创建的细节。

**模式的结构**

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。



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。Object对象有个clone()方法，实现了对象中各个属性的复制，但它的可见范围是protected的，所以实体类使用克隆的前提是：

① 实现Cloneable接口，这是一个标记接口，自身没有方法。

② 覆盖clone()方法，可见性提升为public。

**//具体原型类**

class Realizetype implements Cloneable

{

public Realizetype 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));

}

}

但这种简单的super.clone();属于浅克隆，即只是进行了值复制，对于只包含基本类型属性的类，这种克隆没有问题。但如果一个对象除了包含基本类型属性，还包含其它实体类对象的引用，这种浅克隆只是复制了对象的引用地址，被克隆的对象中的引用对象与原引用对象共享相同的引用。

要实现将克隆对象内容（包含其它实体类对象的引用）完全复制，就需进行深克隆，将各引用的对象也专门复制。

public Object clone() {

RealizeType obj=(RealizedType)super.clone();

//填加自身向obj复制各引用对象的代码

return obj;

}

## 结构型模式

回忆、重新学习[合成复用原则](#_6、合成复用原则（Composite Reuse Principle）)

在本章学习的各种结构型设计模式中，主要关注类和对象的组合关系。将复杂、多样的类的继承关系用组合来替代，而原本的继承只被用来组合接口和定义组合对象以获得新功能。

结构型模式描述如何将类或对象按某种布局组成更大的结构。它分为类结构型模式和对象结构型模式，前者采用继承机制来组织接口和类，后者釆用组合或聚合来组合对象。

由于组合关系或聚合关系比继承关系耦合度低，满足“合成复用原则”，所以对象结构型模式比类结构型模式具有更大的灵活性。

结构型模式分为以下 7 种：

适配器（Adapter）模式：将一个类的接口转换成满足新标准的另外一个接口，使得原本由于接口不兼容类能一起工作。

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的，从而降低了抽象和实现这两个可变维度的耦合度。

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。

装饰（Decorator）模式：动态地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、简化的接口，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。

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。

以上 7 种结构型模式，除了适配器模式分为类结构型模式和对象结构型模式两种，其他的全部属于对象结构型模式，下面我们会分别、详细地介绍它们的特点、结构与应用。

### **适配器模式（Adapter Pattern）**

在应用程序中我们可能需要将两个不同接口的类之间进行通信，在不修改这两个的前提下我们可能会需要某个中间件来完成这个衔接的过程。这个中间件就是适配器。所谓适配器模式就是将一个类的接口，转换成客户期望的另一个接口。它可以让原本两个不兼容的接口能够无缝完成对接。

作为中间件的适配器将目标类和适配者解耦，增加了类的透明性和可复用性。

**适配器模式（Adapter）的定义**：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。

**该模式的主要优点：**

客户端通过适配器可以透明地调用目标接口。

复用了现存的类，程序员不需要修改原有代码就能使用现有的适配者类。

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。

其缺点是：对类适配器来说，更换适配器的实现过程比较复杂。

**模式的应用场景：**

适配器模式（Adapter）通常适用于以下场景。

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。

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

**模式的结构与实现**

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。

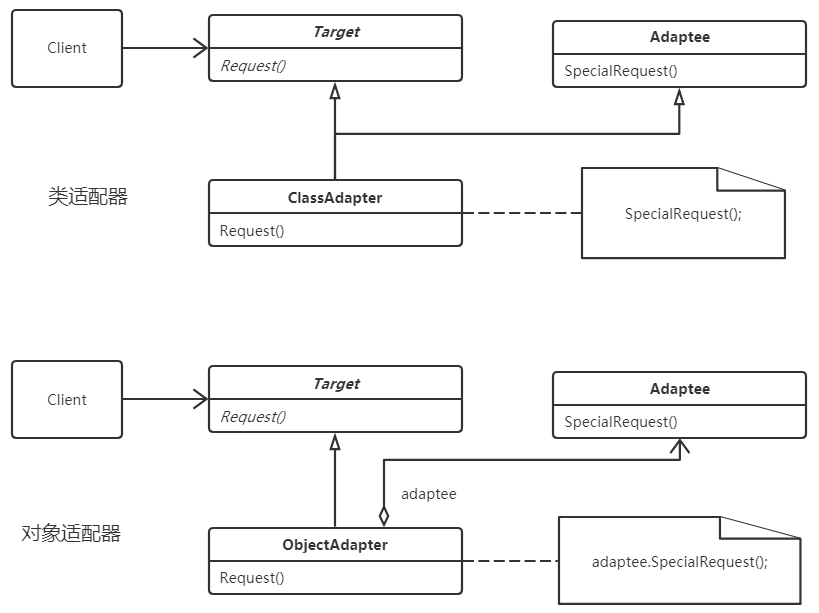
**模式的结构**

适配器模式（Adapter）包含以下主要角色。

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。

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。

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。



模式的实现

**（１）类适配器**

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 TwowayObjectAdapter implements Adaptee, implements Target　{

Target target;

Adaptee adaptee;

public TwowayjObectAdapter (Adaptee adaptee, Target target) {

target = new Target();

adaptee= new Adaptee();

}

public void request() {

specificRequest();

}

public void specialRequest() {

target .request();

}

}

//对象适配器类

class ObjectAdapter implements Target　{

private Adaptee adaptee;

public ObjectAdapter()　{

this.adaptee= new Adaptee();

}

public void request() {

adaptee.specificRequest();

}

}

//客户端代码

public class AdapterTest　{

public static void main(String[] args) {

System.out.println("类适配器模式测试：");

Target target = new ClassAdapter();

target.request();

System.out.println("对象适配器模式测试：");

Adaptee adaptee = new Adaptee();

Target target = new ObjectAdapter(adaptee);

target.request();

}

}

### **桥接模式（Bridge Pattern）**

如果说某个系统能够从多个角度来进行分类，且每一种分类都可能会变化。如果将m个角度分类，每个分类平均有n种变化，就会得到m\*n种不同种类的事物。完全以继承方式实现这种分类和变化是不可取的。我们需要做的就是将这多个角度分离出来，使得他们能独立变化，减少他们之间的耦合，这个分离过程就使用了桥接模式。所谓桥接模式就是讲抽象部分和实现部分隔离开来，使得他们能够独立变化。这种方式只需m+n种实现即可。

桥接模式将继承关系转化成关联关系，封装了变化，完成了解耦，减少了系统中类的数量，也减少了代码量。

**桥接（Bridge）模式的定义**：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。

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。

**桥接（Bridge）模式的优点：**

由于抽象与实现分离，所以扩展能力强；

其实现细节对客户透明。

**缺点是：**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。

**桥接模式的应用场景：**

桥接模式通常适用于以下场景。

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。

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。

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。

**模式的结构**

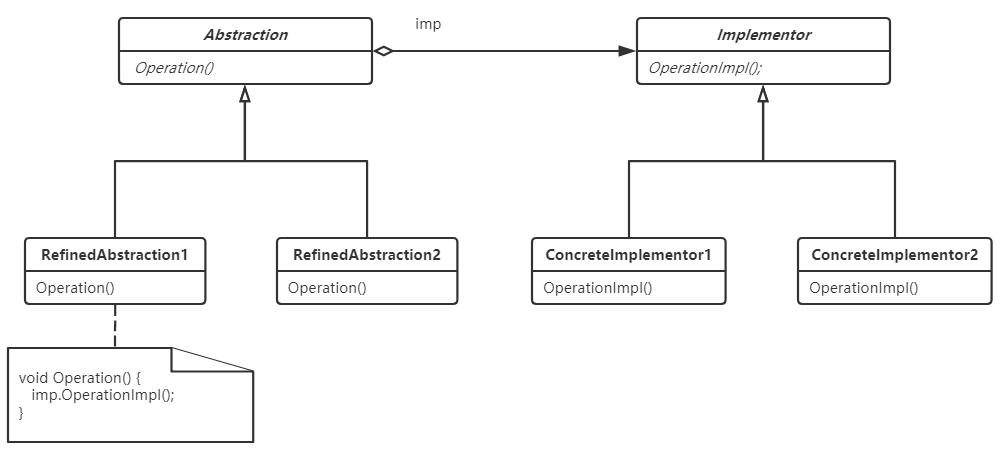
桥接（Bridge）模式包含以下主要角色。

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。



**模式的实现**

package bridge;

//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 ImplementorA)角色被访问" );

}

}

class ConcreteImplementorB implements Implementor

{

public void OperationImpl()

{

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

}

}

//抽象化角色

abstract class Abstraction

{

protected Implementor imple;

protected Abstraction(Implementor imple)

{

this.imple=imple;

}

public void setImplementor(Implementor imple)

{

this.imple=imple;

}

public abstract void Operation();

}

//扩展抽象化角色

class RefinedAbstraction1 extends Abstraction

{

protected RefinedAbstraction1(Implementor imple)

{

super(imple);

}

public void Operation()

{

System.out.println("扩展抽象化(Refined Abstraction1)角色被访问" );

imple.OperationImpl();

}

}

class RefinedAbstraction2 extends Abstraction

{

protected RefinedAbstraction2(Implementor imple)

{

super(imple);

}

public void Operation()

{

System.out.println("扩展抽象化(Refined Abstraction2)角色被访问" );

imple.OperationImpl();

}

}

//客户端

public class BridgeTest

{

public static void main(String[] args)

{

Implementor imple=new ConcreteImplementorA();

Abstraction abs=new RefinedAbstraction1(imple);

abs.Operation();

imple=new ConcreteImplementorB();

abs=new RefinedAbstraction2(imple);

abs.Operation();

}

}

扩展抽象化(Refined Abstraction1)角色被访问

具体实现化(Concrete ImplementorA)角色被访问

扩展抽象化(Refined Abstraction2)角色被访问

具体实现化(Concrete ImplementorB)角色被访问

**桥接模式模式的扩展**

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。

【例】用桥接（Bridge）模式模拟女士皮包的选购。

分析：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。

本实例按用途分可选钱包（Wallet）和挎包（HandBag），按颜色分可选黄色（Yellow）和红色（Red）。可以按两个维度定义为颜色类和包类。

颜色类（Color）是一个维度，定义为实现化角色，它有两个具体实现化角色：黄色和红色，通过 getColor() 方法可以选择颜色；包类（Bag）是另一个维度，定义为抽象化角色，它有两个扩展抽象化角色：挎包和钱包，它包含了颜色类对象，通过 getName() 方法可以选择相关颜色的挎包和钱包。

package bridge;

//实现化角色：颜色

interface Color

{

String getColor();

}

//具体实现化角色：黄色

class Yellow implements Color

{

public String getColor()

{

return "yellow";

}

}

//具体实现化角色：红色

class Red implements Color

{

public String getColor()

{

return "red";

}

}

//抽象化角色：包

abstract class Bag

{

protected Color color;

public void setColor(Color color)

{

this.color=color;

}

public abstract String getName();

}

//扩展抽象化角色：挎包

class HandBag extends Bag

{

public String getName()

{

return color.getColor()+" HandBag";

}

}

//扩展抽象化角色：钱包

class Wallet extends Bag

{

public String getName()

{

return color.getColor()+" Wallet";

}

}

//用户程序

public class BagManage

{

public static void main(String[] args)

{

Color yellowcolor= new Yellow() ;

Color redcolor= new Red() ;

Bag handbag= new HandBag();

Bag wallet = new Wallet();

handbag.setColor(yellowcolor);

wallet .setColor(redcolor);

System.out.println(handbag.getName());

System.out.println(wallet .getName());

handbag.setColor(redcolor);

wallet .setColor(yellowcolor);

System.out.println(“更换颜色：”);

System.out.println(handbag.getName());

System.out.println(wallet .getName());

}

}

输出结果：

yellow HandBag

red Wallet

更换颜色：

red HandBag

Yellow Wallet

### **组合模式（Composite Pattern）**

在现实生活及软件开发过程中，存在很多“部分-整体”的关系，例如，大学中学院与专业和班级、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、绘图中的图形组合等。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。

**组合（Composite）模式的定义**：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。

**组合模式的主要优点：**

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；

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”；

**组合模式主要缺点：**

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；

不容易限制容器中的构件；

不容易用继承的方法来增加构件的新功能；

**组合模式的应用场景:**

前面分析了组合模式的结构与特点，下面分析它适用的以下应用场景。

在需要表示一个对象整体与部分的层次结构的场合。

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。

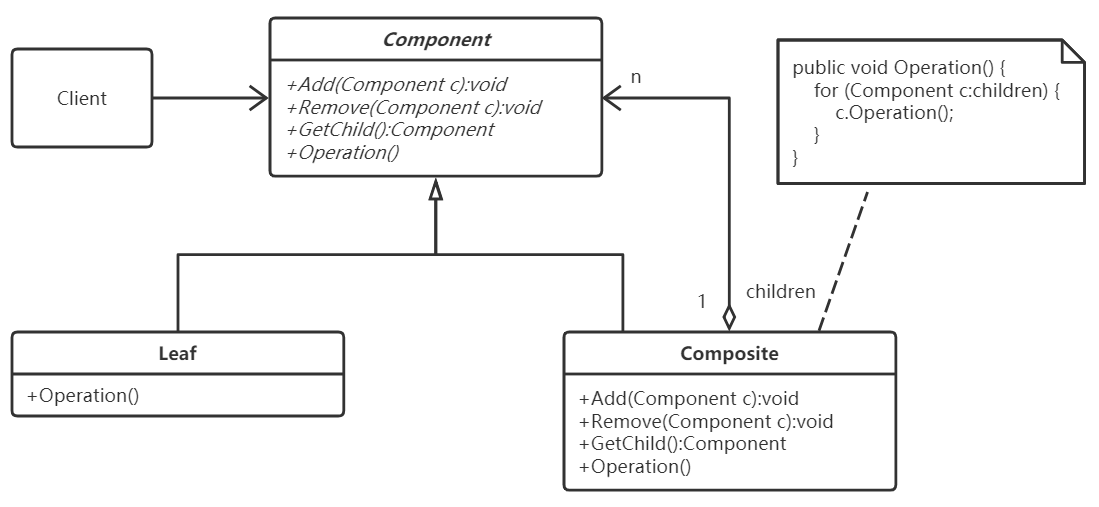
**组合模式结构**

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。

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：是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。



**组合模式的实现**

package composite;

import java.util.ArrayList;

//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 Leaf1 implements Component

{

private String name;

public Leaf1(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1类--"+name+"！");

}

}

class Leaf2 implements Component

{

private String name;

public Leaf2(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2类--"+name+"！");

}

}

//树枝构件

class Composite1 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()

{

System.out.println("组件类1");

for(Component c: children)

{

c.operation();

}

}

}

class Composite2 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()

{

System.out.println("组件类2");

for(Component c: children)

{

c.operation();

}

}

}

public class CompositePattern

{

public static void main(String[] args)

{

Component c0=new Composite1();

Component c1=new Composite1();

Component c2=new Composite2();

Component c3=new Composite2();

Component leaf1=new Leaf1("1");

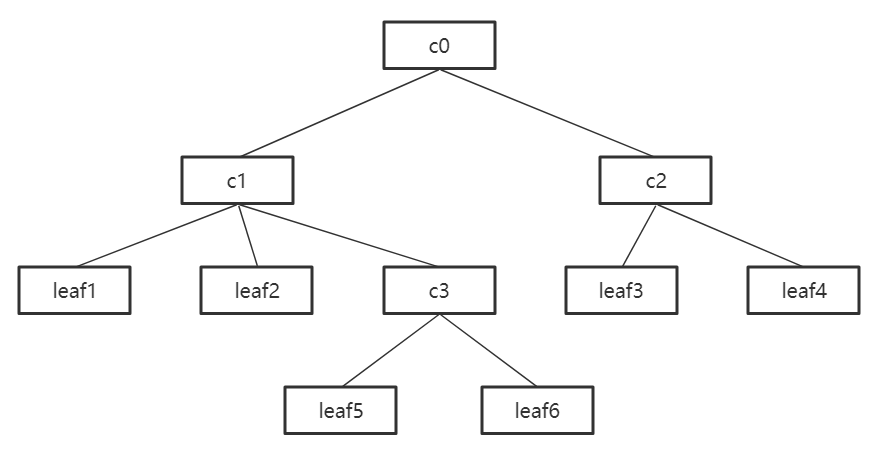
Component leaf2=new Leaf1("2");

Component leaf3=new Leaf2("A");

Component leaf4=new Leaf2("B");

Component leaf5=new Leaf2("C");

Component leaf6=new Leaf2("D");

 c0.add(c1);

c0.add(c2);

c1.add(leaf1);

c1.add(leaf2);

c1.add(c3);

c2.add(leaf3);

c2.add(leaf4);

c3.add(leaf5);

c3.add(leaf6);

c0.operation();

}

}

结果输出：

组件类1

组件类1

树叶1类--1！

树叶1类--2！

组件类2

树叶2类--C！

树叶2类--D！

组件类2

树叶2类--A！

树叶2类--B！

【例】用组合模式实现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。

假如李先生到韶关“天街e角”生活用品店购物，用 1 个红色小袋子装了 2 包婺源特产（单价 7.9 元）、1 张婺源地图（单价 9.9 元）；用 1 个白色小袋子装了 2 包韶关香藉（单价 68 元）和 3 包韶关红茶（单价 180 元）；用 1 个中袋子装了前面的红色小袋子和 1 个景德镇瓷器（单价 380 元）；用 1 个大袋子装了前面的中袋子、白色小袋子和 1 双李宁牌运动鞋（单价 198 元）。

最后“大袋子”中的内容有：{1 双李宁牌运动鞋（单价 198 元）、白色小袋子{2 包韶关香菇（单价 68 元）、3 包韶关红茶（单价 180 元）}、中袋子{1 个景德镇瓷器（单价 380 元）、红色小袋子{2 包婺源特产（单价 7.9 元）、1 张婺源地图（单价 9.9 元）}}}，现在要求编程显示李先生放在大袋子中的所有商品信息并计算要支付的总价。

package composite;

import java.util.ArrayList;

//抽象构件：物品

interface Articles {

public float calculation(); //计算

public void show();

}

//树叶构件：商品

class Goods implements Articles {

private String name; //名字

private int quantity; //数量

private float unitPrice; //单价

public Goods(String name,int quantity,float unitPrice) {

this.name=name;

this.quantity=quantity;

this.unitPrice=unitPrice;

}

public float calculation() {

return quantity\*unitPrice;

}

public void show() {

System.out.println(name+"(数量："+quantity+"，单价："+unitPrice+"元)");

}

}

//树枝构件：袋子

class Bags implements Articles {

private String name; //名字

private ArrayList<Articles> bags=new ArrayList<Articles>();

public Bags(String name) {

this.name=name;

}

public void add(Articles c) {

bags.add(c);

}

public void remove(Articles c) {

bags.remove(c);

}

public Articles getChild(int i) {

return bags.get(i);

}

public float calculation() {

float s=0;

for(Object obj:bags) {

s+=((Articles)obj).calculation();

}

return s;

}

public void show() {

for(Object obj:bags) {

((Articles)obj).show();

}

}

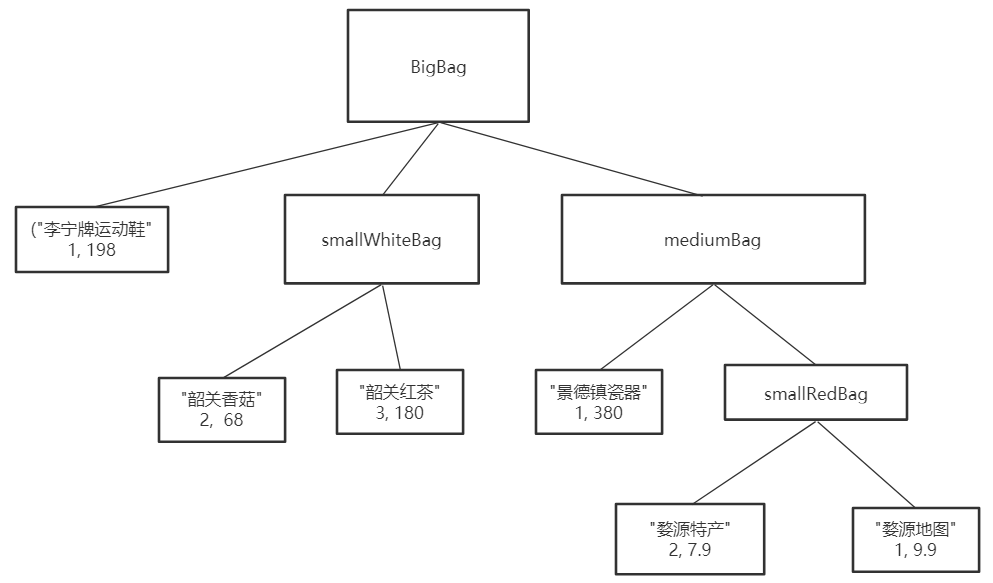
}

public class ShoppingTest {

public static void main(String[] args) {

float s=0;

Bags BigBag,mediumBag,smallRedBag,smallWhiteBag;

 Goods sp;

BigBag=new Bags("大袋子");

mediumBag=new Bags("中袋子");

smallRedBag=new Bags("红色小袋子");

smallWhiteBag=new Bags("白色小袋子");

sp=new Goods("婺源特产",2,7.9f);

smallRedBag.add(sp);

sp=new Goods("婺源地图",1,9.9f);

smallRedBag.add(sp);

sp=new Goods("韶关香菇",2,68);

smallWhiteBag.add(sp);

sp=new Goods("韶关红茶",3,180);

smallWhiteBag.add(sp);

sp=new Goods("景德镇瓷器",1,380);

mediumBag.add(sp);

mediumBag.add(smallRedBag);

 sp=new Goods("李宁牌运动鞋",1,198);

BigBag.add(sp);

BigBag.add(smallWhiteBag);

BigBag.add(mediumBag);

System.out.println("您选购的商品有：");

BigBag.show();

s=BigBag.calculation();

System.out.println("要支付的总价是："+s+"元");

}

}

### **装饰器模式（Decorator Pattern）**

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。在现实生活及软件开发过程中，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。

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。

**装饰（Decorator）模式定义：**指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）和属性的模式，它属于对象结构型模式。

**装饰模式的主要优点：**

采用装饰模式扩展对象的功能比采用继承方式更加灵活。

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。

**装饰模式的主要缺点是：**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。

**装饰模式的应用场景**

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。

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。

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。

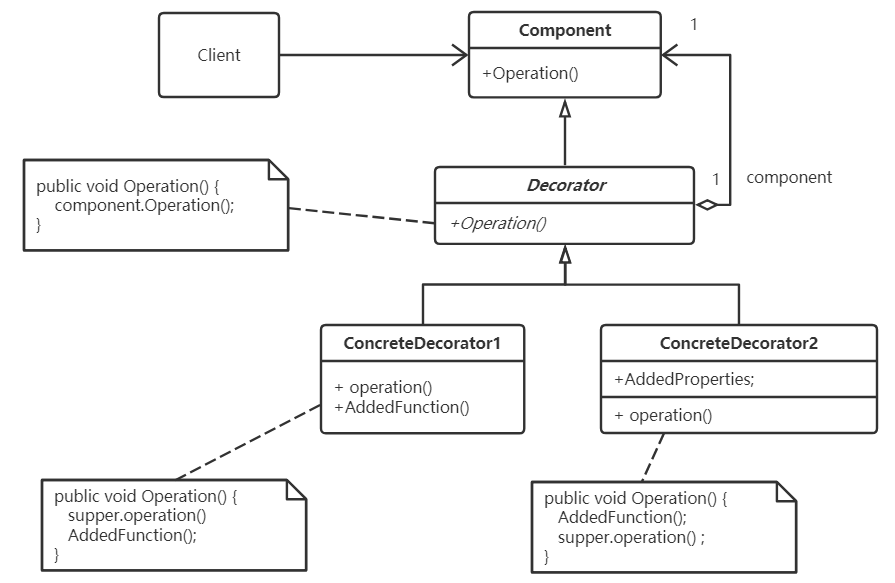
**装饰模式结构**

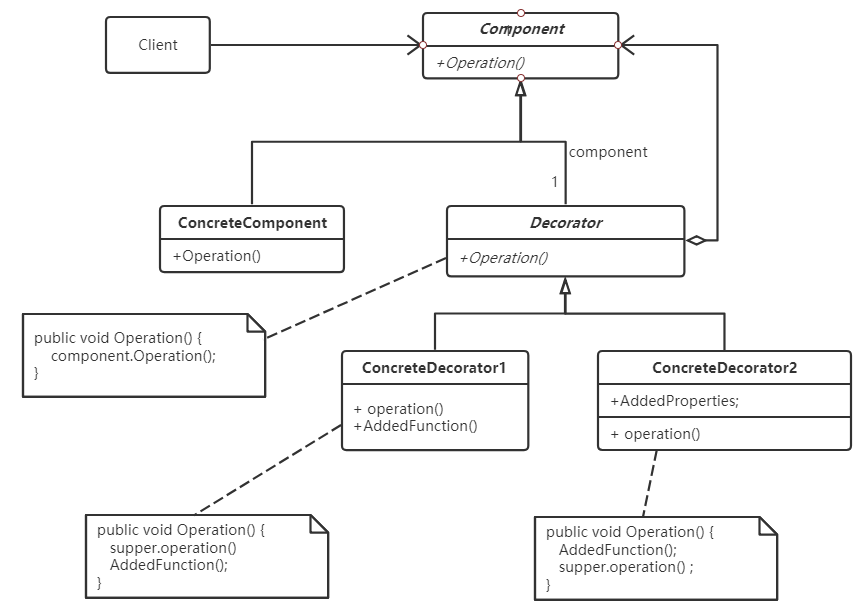
抽象构件（Component）角色：定义一个抽象接口以规范准备接收附加责任的对象。

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。

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。

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。





**装饰模式实现**

package decorator;

//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 ConcreteDecorator1 extends Decorator {

public ConcreteDecorator1(Component component) {

super(component);

}

public void operation() {

super.operation();

addedFunction1();

}

public void addedFunction1() {

System.out.println("在具体构件角色功能后增加额外的功能addedFunction1()");

}

}

class ConcreteDecorator2 extends Decorator {

private string s=”new propertiy”;

public ConcreteDecorator2(Component component) {

super(component);

}

public void operation() {

addedFunction2();

super.operation();

}

public void addedFunction2() {

System.out.println("在具体构件角色功能前增加额外的功能addedFunction2()");

}

}

//装饰模式

public class DecoratorPattern {

public static void main(String[] args) {

Component p=new ConcreteComponent();

p.operation();

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

Component d=new ConcreteDecorator1(p);

d.operation();

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

Component e=new ConcreteDecorator2(p);

e.operation();

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

Component f=new ConcreteDecorator1(e);

f.operation();

}

}

**运行结果：**

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()

在具体构件角色功能后增加额外的功能addedFunction1()

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

在具体构件角色功能前增加额外的功能addedFunction2()

调用具体构件角色的方法operation()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

在具体构件角色功能前增加额外的功能addedFunction2()

调用具体构件角色的方法operation()

在具体构件角色功能后增加额外的功能addedFunction1()

### **外观模式（Facade Pattern）**

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。外观模式是为了解决使用者类与被使用者类之间的依赖关系的，将各子系统关系放在一个Facade类中，降低了类类之间的耦合度。

**外观（Facade）模式的定义：**是一种通过为多个复杂的子系统提供一个统一的接口，而使这些子系统更加容易被访问的模式。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。  
  
**外观（Facade）模式是“迪米特法则”的典型应用，它有以下主要优点：**

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。

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。

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。

**外观（Facade）模式的主要缺点：**

不能很好地限制客户使用子系统类。

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”。

**外观模式的应用场景：**

通常在以下情况下可以考虑使用外观模式。

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。

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。

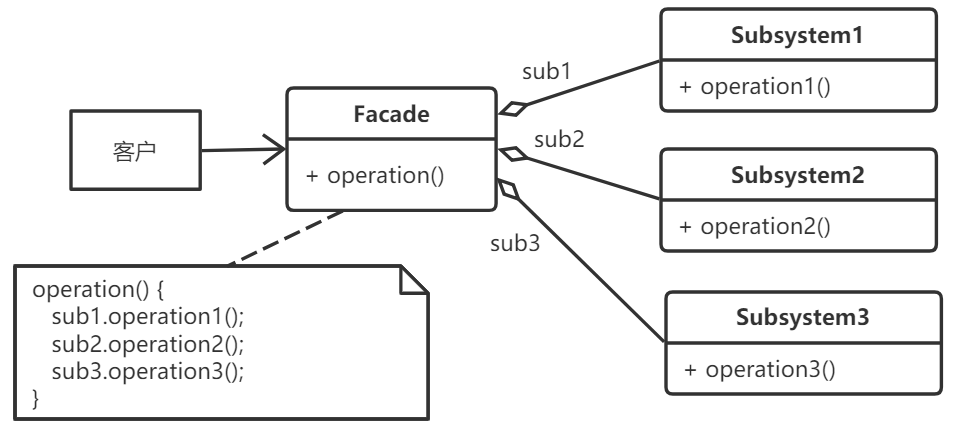
当客户端与多个子系统之间存在很大的联系时，引入外观模式可将它们分离，从而提高子系统的独立性和可移植性。

**外观（Facade）模式的结构**

外观（Facade）角色：为多个子系统对外提供一个统一的外部访问的接口。

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。



例如手动档汽车的换档车是很需要技巧的，而自动档汽车将这些工作替驾驶员自动来做了，驾驶员只需要油门控制即可。

**这是在网上查到的一个“史上最全的手动挡汽车换挡技巧”：**

1、牢记手动挡挡位图

手动挡车的挡位分布就那几种，当你坐在驾驶座上，第一眼就应该扫描变速杆手把上的挡位图。切记开车时不能低头看变速杆换挡，特别是在市区和高速时，低头换挡容易造成车辆跑偏，往往事故随之发生。

2、要熟练离合器的踩踏

多练习，踩离合器要稳。养成好的习惯，一脚踩到底，缓慢释放。

3、要掌握换档的时机

当发动机转速在2000转左右换档，先减油门然后再踩下离合器，这样离合器就不会被强制脱开，然后换挡。进档顺序一定要是一档--二档--三档--四档--五档，不推荐跳档，减档顺序按车速，根据自己车子的情况，一般发动机转速低于1500转左右就要减档。

4、车子行驶过程中要勤换档

根据车速和路况选择合适的档位，如果换档不勤，很容易使车子拖档或发动机疯鸣，增加汽车的磨损和费油，所以一定要养成勤换档的习惯。

5、换档动作要连贯

离合器要一脚到底，挂挡也要一步到位，要多熟悉档位，多练习手感，如果遇到无法上档的情况(现在的汽车很少遇见)，挂空档松一下离合器，然后重新踩下离合器，就可以上档了，如果档位没有到位，不要送开离合器，那样很容易磨损变速齿轮。

**驾驶手动挡车型如何快速起步**

1、启动车辆时务必同时踩下刹车和离合器踏板

很多没有接触过手动挡车型的人很容易出现这种情况：上车拧钥匙却发现打不着火，也不明白为什么会这样。其实解决办法很简单：在拧钥匙之前，先踩下刹车和离合器踏板，然后再打火，车辆自然就能启动了。这种保护措施主要目的是为了防止变速箱不在空挡时启动车辆导致失控。

上车后不要急于拧钥匙点火，一旦此时不是空挡，车辆很容易在点火瞬间猛蹿出去。正确做法应该在拧钥匙之前先踩下刹车和离合器踏板，同时确认挡位是否在空挡，防止车辆在点火瞬间失控。

2、找准离合器衔接点同时不要长时间半联动

每个驾校想必都会告诉学员一个词语：压离合。所谓压离合，就是当踩下离合器后慢慢抬起时，发动机与变速箱动力开始衔接，此时会感到车辆有向前走的趋势。这个动力衔接点每辆车都有所不同，所以需要新手适应。

压离合是驾驶手动挡车型最重要、最基础的技巧之一。

随着离合器慢慢抬起，发动机转速开始降低，同时车辆有向前走的趋势时，证明你找到了离合器半联动的点。

如果此时脚下离合器控制不稳，车辆很可能被憋熄火。不要慌张，快速踩下离合器并摘到空挡，同时踩下刹车再次启动车辆即可。

3、起步油离配合好

当熟悉了离合器半联动后，如何快速起步又是另外一件比较重要的事情了。身处大城市，生活节奏非常快，如果没有掌握快速起步的要领，很可能会招来后面司机急迫的催促。

快速起步方法很简单，在离合器还未到达半联动点时，轻轻点下并稳住油门，而后再慢慢松开离合器。这样不仅不会熄火，同时还能很平顺地起步。

起步前稍微给点油门，是快速起步的法宝。

### **享元模式（Flyweight Pattern）**

面向对象技术可以很好地解决一些灵活性或可扩展性问题，但在很多情况下需要在系统中增加类和对象的个数。当对象数量太多时，将导致运行代价过高，带来性能下降等问题。享元模式通过共享技术实现相同或相似对象的重用提高系统资源的利用率。

**享元（Flyweight）模式的定义：**运用共享技术有效地支持大量细粒度的对象的复用。系统只使用少量的对象，而这些对象都很相似，状态变化很小，可以实现对象的多次复用。由于享元模式要求能够共享的对象必须是细粒度对象，因此它又称为轻量级模式。

在享元模式中可以共享的相同内容称为内部状态(Intrinsic State)，而那些需要外部环境来设置的不能共享的内容称为外部状态(Extrinsic State)，其中外部状态和内部状态是相互独立的，外部状态的变化不会引起内部状态的变化。由于区分了内部状态和外部状态，因此可以通过设置不同的外部状态使得相同的对象可以具有一些不同的特征，而相同的内部状态是可以共享的。也就是说，享元模式的本质是分离与共享 ： 分离变与不变，并且共享不变。把一个对象的状态分成内部状态和外部状态，内部状态即是不变的，外部状态是变化的；然后通过共享不变的部分，达到减少对象数量并节约内存的目的。

例如我们在计算机上编辑的各种文档中使用的文字，每个文字就是一个对象，其写法就是共享的内部状态，而其在文档中的位置、字号、颜色等应是非共享的外部状态。（在某些时候，又可把一个连续的具有相同属性的部分如颜色、字号等也会按共享属性处理）。而字库就是存储文字共享信息的地方。

在享元模式中通常会出现工厂模式，需要创建一个享元工厂来负责维护一个享元池(Flyweight Pool)，用于存储具有相同内部状态的享元对象。在享元模式中，共享的是享元对象的内部状态，外部状态需要通过环境来设置。在实际使用中，能够共享的内部状态是有限的，因此享元对象一般都设计为较小的对象，它所包含的内部状态较少，这种对象也称为细粒度对象。

**享元模式的主要优点：**

相同对象只要保存一份，降低了系统中对象的数量，从而降低了给内存带来的压力。

**其主要缺点：**

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。

读取享元模式的外部状态会使得运行时间稍微变长。

**享元模式的应用场景：**

1、系统中有大量对象。 2、这些对象消耗大量内存。 3、这些对象的状态大部分可以外部化。 4、这些对象可以按照内蕴状态分为很多组，当把外蕴对象从对象中剔除出来时，每一组对象都可以用一个对象来代替。 5、系统不依赖于这些对象身份，这些对象是不可分辨的。

注意事项： 1、注意划分外部状态和内部状态，否则可能会引起线程安全问题。 2、这些类必须有一个工厂对象加以控制。

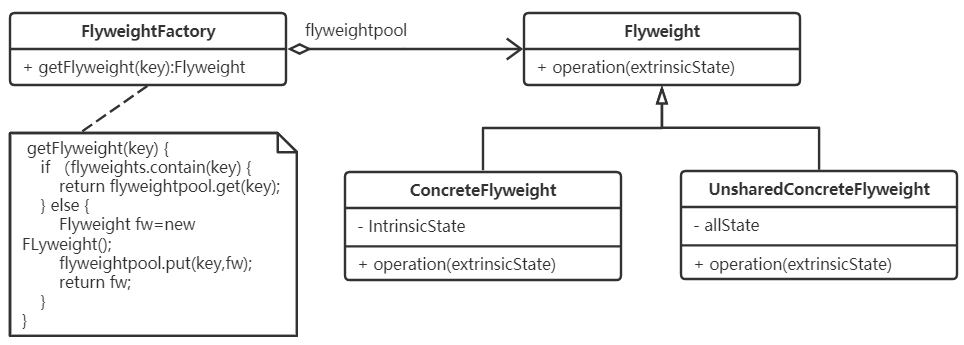
**享元模式Flyweight的结构：**

抽象享元角色（Flyweight）:是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：是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。



两种 FlyweightFactory：

(1) 一般形态

public class FlyweightFactory

{

private HashMap flyweights = new HashMap();

public Flyweight getFlyweight(String key)

{

if(flyweights.containsKey(key))

{

return (Flyweight)flyweights.get(key);

}

else

{

Flyweight fw = new ConcreteFlyweight();

flyweights.put(key,fw);

return fw;

}

}

}

(2) 事先创建形态

public class FlyweightFactory

{

private HashMap flyweights = new HashMap();

public Flyweight getFlyweight(String key)

{

if(flyweights.containsKey(key))

{

return (Flyweight)flyweights.get(key);

}

else return null;

}

public void putFlyweight(String key, Flyweight fw)

{

If (!flyweights.containsKey(key))

{

flyweights.put(key,fw);

}

}

}

例子：

//Word .java, 一个Flyweight

package FlyweightPattern;

public class Word {

private String EStr;

private String CStr;

public Word(String eStr, String cStr) {

EStr=eStr;

CStr=cStr;

}

public void Print() {

Print(0);

}

public void Print(int lang) {

if (lang==0) System.out.print(CStr);

else System.out.print(EStr+" ");

}

}

//FlyweightFactory.java

package FlyweightPattern;

import java.util.HashMap;

public class FlyweightFactory {

private static final HashMap<String, Word> wordMap = new HashMap<>();

public static Word getWord(String key) {

return (Word)wordMap.get(key);

}

public static Word putWord(String key,Word wd) {

if (wordMap.get(key)!=null) return;

return wordMap.put(key, wd);

}

}

//FlyweightPattern .java

package FlyweightPattern;

public class FlyweightPattern {

public static void main(String[] args) {

// TODO code application logic here

Word wd;

wd =new Word("seagull","海鸥");

FlyweightFactory.putWord("a", wd);

wd =new Word("our","我们的");

FlyweightFactory.putWord("b", wd);

wd =new Word("friend","朋友");

FlyweightFactory.putWord("c", wd);

wd =new Word("you are","你是");

FlyweightFactory.putWord("d", wd);

wd =new Word("good","好");

FlyweightFactory.putWord("e", wd);

String str="aabc,dbec!\n";

printString(str,0);

printString(str,1);

}

private static void printString(String str, int lang) {

for (int i=0; i<str.length(); i++) {

String key=str.substring(i, i+1);

Word wd=FlyweightFactory.getWord(key);

if (wd==null) {

System.out.print(key);

} else

wd.Print(lang);

}

}

}

**运行结果：**

海鸥海鸥我们的朋友,你是我们的好朋友!

seagull seagull our friend ,you are our good friend !

### **代理模式（Proxy Pattern）**

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。

**代理模式的定义：**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。

**代理模式的主要优点：**

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；

代理对象可以扩展目标对象的功能；

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；

**其主要缺点：**

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；

增加了系统的复杂度；

**代理模式的应用场景:**

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。

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。

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。

智能指针: 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。

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。

**代理模式的结构**

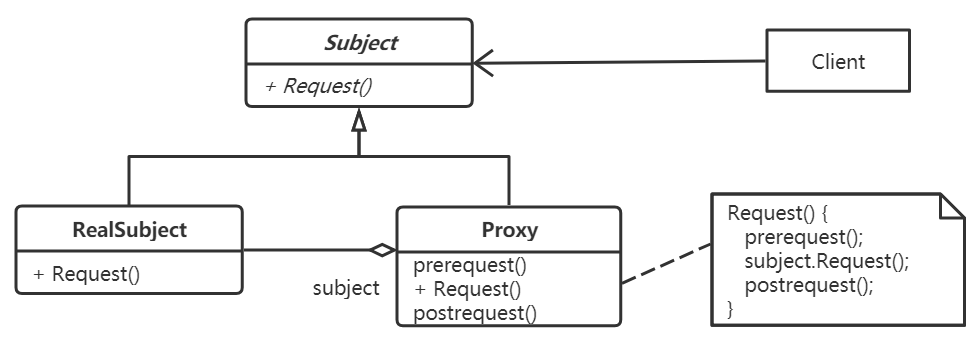
代理模式的结构比较简单，主要是通过定义一个继承抽象主题的代理来包含真实主题，从而实现对真实主题的访问，下面来分析其基本结构和实现方法。

代理模式的主要角色如下。

抽象主题（Subject）：声明了目标对象和代理对象的共同接口，这样一来在任何可以使用目标对象的地方都可以使用代理对象。

具体主题（RealSubject)：也称为委托角色或者被代理角色。定义了代理对象所代表的目标对象。

代理主题（Proxy)：也叫委托类、代理类。代理对象内部含有目标对象的引用，从而可以在任何时候操作目标对象；代理对象提供一个与目标对象相同的接口，以便可以在任何时候替代目标对象。代理对象通常在客户端调用传递给目标对象之前或之后，执行某个操作，而不是单纯地将调用传递给目标对象。



**代理模式的实现**

package proxy;

//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 Proxy (RealSubject asubject)

{

realSubject = asubject;

}

public void Request()

{

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。");

}

}

//ProxyPattern

public class ProxyPattern

{

public static void main(String[] args)

{

RealSubject subject=new RealSubject();

Proxy proxy=new Proxy(subject);

proxy.Request();

}

}

程序执行结果：

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。

## 行为型模式

行为型模式用于描述对象之间的通信、协同合作行为机制。这些通信、协同合作是通过过程（方法）调用来实现的，用以完成单个对象都无法单独完成的任务，它涉及算法与对象间职责的分配。

行为型模式分为**类行为模式**和**对象行为模式**，前者采用继承机制来在类间分派行为，后者采用组合或聚合在对象间分配行为。由于组合关系或聚合关系比继承关系耦合度低，满足“合成复用原则”，所以对象行为模式比类行为模式具有更大的灵活性。

行为型模式包含以下 11 种模式。

责任链（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。

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。

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。

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。

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。

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。

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。

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。

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。

模板方法（Template Method）模式：定义一个操作中的算法骨架，将算法的一些步骤延迟到子类中，使得子类在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。

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。

以上 11 种行为型模式，除了模板方法模式和解释器模式是类行为型模式，其他的全部属于对象行为型模式，下面我们将详细介绍它们的特点、结构与应用。

### **责任链模式（Chain of Responsibility Pattern）**

责任链模式

顾名思义，责任链模式（Chain of Responsibility Pattern）为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。

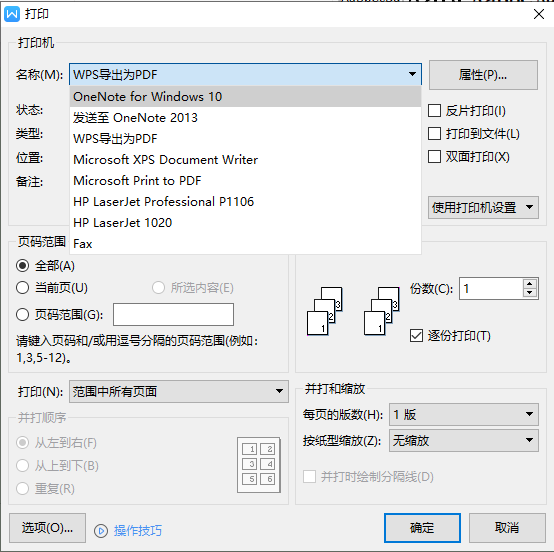
在责任链模式中，通常每个接收者都包含对另一个接收者的引用（有一个指向另一个接收者的指针）。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

所谓接收者就是通信的接收者、被调用对象。一般地，所有接收者组成一个队列链表。

经常存在这样的情况：可能有多个不同的对象完成同一类工作，例如文档打印输出，可以选择不同的打印机名称。然而文档编辑器（或者是系统）事先并不知道有多少种打印机可以使用，不知道机器上安装了哪些打印设备。这些设备有些是系统安装时自带的，有些是后续安装的。如何让系统能够访问这些事先不知道是哪一个类型的打印机呢？

操作系统规定了打印机驱动程序标准接口，所有打印机都要提供支持这种接口的驱动程序。当打印机安装时，相应的驱动程序就被安装到系统中，被链入到打印驱动链表。当文档编辑器（或者是系统）要打印输出文档时，首先要选择打印机（指定打印机名称），通过该打印机名称（实际上是打印机ID）将打印命令请求发给打印驱动链表。打印驱动链表中的打印驱动程序根据请求指定的打印机ID确定是否与自身的名字ID所匹配，匹配则执行请求，不匹配则将请求传递给链表中的下一驱动程序。

这种机制的好外是：系统不必事先知道所有打印机的名字、安装所有打印机的驱动程序；打印时也不必面向所有的打印机驱动程序发出调用请求或查询，只需和打印机驱动链首发出请求即可。



**责任链（Chain of Responsibility）模式的定义：**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。

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。

**责任链模式是一种对象行为型模式，其主要优点：**

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。

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。

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。

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。

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：**

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。

对比较长的责任链，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。

责任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，可能会由于责任链的错误设置而导致系统出错，如可能会造成循环调用。

**模式的应用场景：**

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。

可动态指定一组对象处理请求，或添加新的处理者。

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。

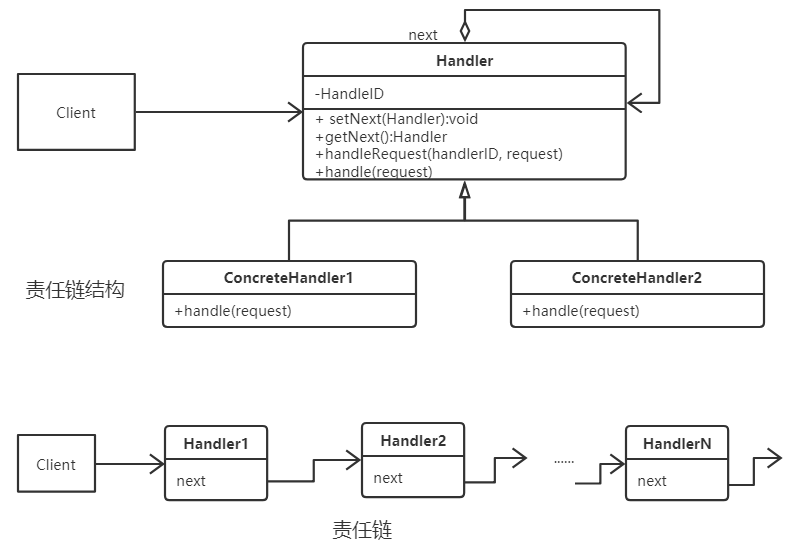
**责任链模式的结构：**

责任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。



**责任链模式的实现：**

package ChainOfResposibilityPettern;

public class ChainOfResponsibilityPattern

{

public static void main(String[] args)

{

//组装责任链

Handler handler1=new ConcreteHandler1("handle1");

Handler handler2=new ConcreteHandler2("handle2");

handler1.setNext(handler2);

//提交请求 handle2

handler1.handleRequest("handle2","我的请求");

}

}

//抽象处理者角色

abstract class Handler

{

protected String HandlerID;

private Handler next;

public Handler(String handlerId) {

HandlerID=handlerId;

}

public void setNext(Handler next) {

this.next=next;

}

public Handler getNext() {

return next;

}

//处理请求的方法

public void handleRequest(String handlerID, String request) {

if (HandlerID.equals(handlerID)) {

handle(request);

} else {

Handler handler=getNext();

if (handler!=null) handler.handleRequest(handlerID, request);

else {

System.out.println("没有人处理请求--"+request+" ！");

}

}

}

protected abstract void handle(String request);

}

//具体处理者角色1

class ConcreteHandler1 extends Handler

{

public ConcreteHandler1(String handlerId) {

super(handlerId);

}

@Override

public void handle(String request) {

System.out.println(HandlerID+"负责处理请求--"+request+" ！");

}

}

//具体处理者角色2

class ConcreteHandler2 extends Handler

{

public ConcreteHandler2(String handlerId) {

super(handlerId);

}

@Override

public void handle(String request) {

System.out.println(HandlerID+"负责处理请求--"+request+" ！");

}

}

**责任链模式的扩展：**

责任链模式存在以下两种情况。

纯的责任链模式：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。

不纯的责任链模式：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。

另外，责任链模式可以用于获取接收者地址，通过责任链获取接收者引用后，客户程序可以在后续的调用中直接访问接收者。当请求者与接收者之间有相关一系列操作时，这样可以提高访问效率。

代码如下：

package ChainOfResposibilityPettern;

public class ChainOfResponsibilityPattern

{

public static void main(String[] args)

{

//组装责任链

Handler handler1=new ConcreteHandler1("handle1");

Handler handler2=new ConcreteHandler2("handle2");

Handler handler3=new ConcreteHandler1("handle3");

handler1.setNext(handler2);

handler2.setNext(handler3);

//提交请求 handle2

Handler hd=handler1.Gethandler("handle3");

if (hd!=null)hd.handle("我的请求");

}

}

//抽象处理者角色

abstract class Handler

{

public String HandlerID;

private Handler next;

public Handler(String handlerId) {

HandlerID=handlerId;

}

public void setNext(Handler next) {

this.next=next;

}

public Handler getNext() {

return next;

}

//处理请求的方法

public Handler Gethandler(String handlerID) {

if (HandlerID.equals(handlerID)) {

return this;

} else {

Handler hd=getNext();

if (hd!=null) return hd.Gethandler(handlerID);

else return null;

}

}

public abstract void handle(String request);

}

//具体处理者角色1

class ConcreteHandler1 extends Handler

{

public ConcreteHandler1(String handlerId) {

super(handlerId);

}

@Override

public void handle(String request) {

System.out.println(HandlerID+"负责处理请求--"+request+" ！");

}

}

//具体处理者角色2

class ConcreteHandler2 extends Handler

{

public ConcreteHandler2(String handlerId) {

super(handlerId);

}

@Override

public void handle(String request) {

System.out.println(HandlerID+"负责处理请求--"+request+" ！");

}

}

### **命令模式（Command Pattern）**

在软件系统中，“方法的请求者”与“方法的实现者”之间通常存在紧密的耦合关系--调用关系。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。

**命令（Command）模式的定义：**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。请求对象和执行对象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。

**命令模式的主要优点：**

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。

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。

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。

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。

**主要其缺点：**可能产生大量具体命令类。因为针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。

**命令模式的应用场景：**

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。

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。

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。

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。

**命令模式的结构：**

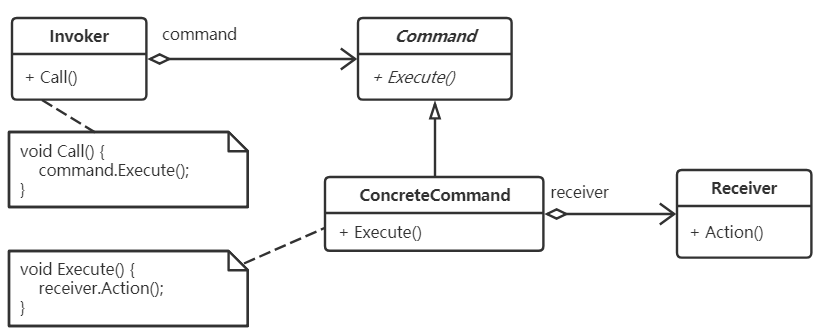
命令模式包含以下主要角色。

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：是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：是请求的发送者，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。



**实现：**

package commandpattern;

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 setReceiver(Receiver areceiver) {

receiver = areceiver;

}

@Override

public void Execute() {

System.out.println("命令执行前...");

receiver.Action();

System.out.println("命令执行后...");

}

}

//接收者

class Receiver

{

public void Action() {

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

}

}

**运行结果：**

客户访问调用者的call()方法...

调用者执行命令command...

命令执行前...

接收者的action()方法被调用...

命令执行后...

### **解释器模式（Interpreter Pattern）**

解释器这个名词想必大家都不会陌生，比如编译原理中，一个算术表达式通过词法分析器形成词法单元，而后这些词法单元再通过语法分析器构建语法分析树，最终形成一颗抽象的语法分析树。诸如此类的例子也有很多，比如编译器、正则表达式等等。

如果一种特定类型的问题发生的频率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子，这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。

就比如正则表达式，它就是解释器模型的一种应用，解释器为正则表达式定义了一个文法，如何表示一个特定的正则表达式，以及如何解释这个正则表达式。

解释器模式常用于对简单语言的编译或分析实例中。首先复习（了解）编译原理中的“文法、句子、语法树”等相关概念。

1) 文法

文法是用于描述语言的语法结构的形式规则。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〉::=大学生I英语

〈动词〉::=是|学习

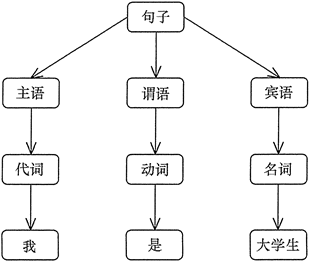
注：这里的符号“::=”表示“定义为”的意思，用“〈”和“〉”括住的是非终结符，没有括住的是终结符。

2) 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。

3) 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。

  
图1 句子“我是大学生”的语法树

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。

****解释器模式（Interpreter）****，Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language。给定一个语言, 定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。

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。  
  
 **解释器模式是一种类行为型模式，其主要优点如下：**

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。

**解释器模式的主要缺点如下：**

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。

**模式的应用场景：**

当语言的文法较为简单，且执行效率不是关键问题时；

当问题重复出现，且可以用一种简单的语言来进行表达时；

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时；

**解释器模式的结构：**

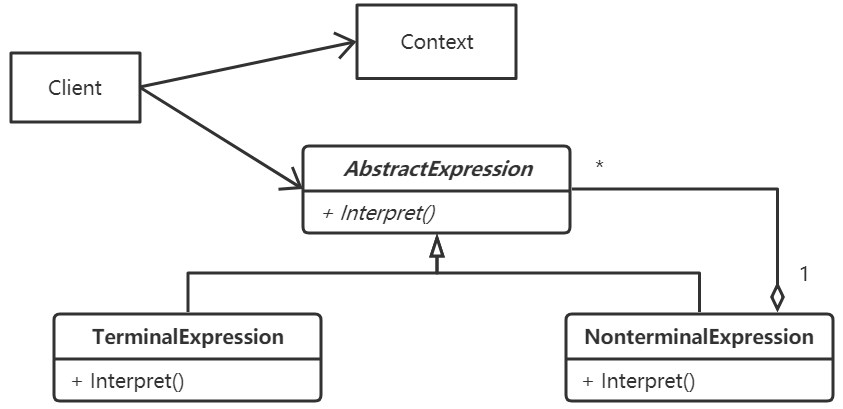
抽象表达式（Abstract Expression）角色：定义解释器的接口（抽象类），约定解释器的解释操作，主要包含解释方法 interpret()。

终结符表达式（Terminal    Expression）角色：是抽象表达式的子类，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。

上下文（Context，又称环境）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（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。



解释器模式的结构与[组合模式](http://c.biancheng.net/view/1373.html" \t "http://c.biancheng.net/view/_blank)相似，一个非终结符表达式由多个表达式（终结符或非终结符）组成，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**。

为什么说解释器是行为模式呢？主要区别在于每个表达式（终结符或非终结符）的Interpret方法都要有不同的实现，要对其包含各子表达式进行相应运算（而组合模式中的对象操作基本都是类似的，组合对象要对其包含的成员对象进行组合操作）。

**模式实现：**

//抽象表达式类

//抽象表达式是生成语法集合（也叫做语法树）的关键，每个语法集合完成指定语法解析任务，它是通过递归调用的方式，最终由最小的语法单元进行解析完成。

public Abstract calss AbstractExpression{

//每个表达式必须有一个解析任务

public Object interpret(Context context) { //解释方法

return 解释结果;

}

}

//终结符表达式类。 通常，终结符表达式比较简单，主要是处理场景元素和数据的转换。

class TerminalExpression extends AbstractExpression

{

//通常终结符表达式只有一个，但是有多个对象

public Object interpret(Context context)

{

//对终结符表达式的处理

return 解释结果;

}

}

//非终结符表达式类。

//每个非终结符表达式都代表了一个文法规则，并且每个文法规则都只关心自己周边的文法规则的结果（注意是结果），因此这就产生了每个非终结符表达式调用自己周边的非终结符表达式，然后最终、最小的文法规则就是终结符表达式，终结符表达式的概念就是如此，不能够再参与比自己更小的文法运算了。

class NonterminalExpression extends AbstractExpression

{

//每个非终结符表达式都会对其他表达式产生依赖，可能包括1个或多个表达式

private AbstractExpression exp1;

private AbstractExpression exp2;

public Object interpret(Context context)

{

//进行文法处理

return 解释结果;

}

}

//上下文（环境）类

class Context

{

private String info; //需要被解释的语言文本

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

}

}

//客户类

class Client

{

private Context context;

private AbstractExpression exp;

public static void main(String[] args) {

//负责输入上下文，并调用表达式对象进行解释。

}

}

**应用举例：整数算术表达式**

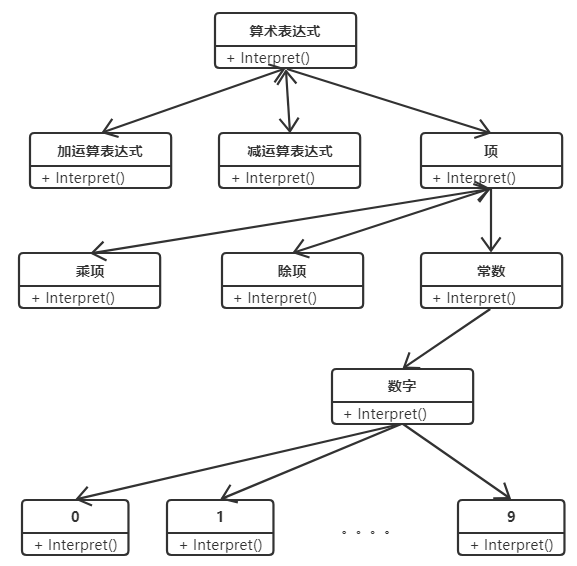
**12+6\*3-7-2+7/2**

〈算术表达式〉::=〈项〉|〈算术表达式〉+〈项〉|〈算术表达式〉-〈项〉

〈项〉::=〈常数〉|〈项〉\*〈常数〉|〈项〉/〈常数〉

〈常数〉::=〈数字〉|〈常数〉〈数字〉

〈数字〉::=0|1|2|3|4|5|6|7|8|9



**程序代码：**

package inpterpreterpattern;

//上下文

class Context {

private String ExpString;

private int CurrentPosition=0;

private Integer ItemValue= null; //记录当前项值

private Integer ResultValue= null; //记录当前表达式值

public void setexpStr( String exp) {

ExpString=exp;

}

public String getexpStr() {

return ExpString.substring(CurrentPosition);

}

public void NextPosition(int offset) {

CurrentPosition+=offset;

}

public void setItemValue(Integer intval) {

ItemValue=intval;

}

public Integer getItemVale() {

return ItemValue;

}

public void setResultValue(Integer intval) {

ResultValue=intval;

}

public Integer getResultValue() {

return ResultValue;

}

}

//算术表达式

class NumericExpression {

AddExpression addExp;

SubExpression subExp;

NumericItem NItem;

public Integer interpret(Context context) {

if (context.getexpStr().length()==0) return null;

Integer dval;

NItem = new NumericItem();

dval=NItem.interpret(context);

if (dval==null) return null;

context.setResultValue(dval);

if (context.getexpStr().length()==0) return context.getResultValue();

String c=context.getexpStr().substring(0, 1);

while (c.equals("+")||c.equals("-")) {

context.NextPosition(1);

switch (c) {

case "+":

addExp = new AddExpression();

addExp.interpret(context);

break;

case "-":

subExp = new SubExpression();

subExp.interpret(context);

break;

}

if (context.getexpStr().length()==0) return context.getResultValue();

c=context.getexpStr().substring(0, 1);

}

return context.getResultValue();

}

}

//加运算表达式

class AddExpression extends NumericExpression{

NumericItem NItem;

@Override

public Integer interpret(Context context) {

if (context.getexpStr().length()==0) return null;

NItem = new NumericItem();

Integer dval;

dval=NItem.interpret(context);

if (dval==null) return null;

context.setResultValue(context.getResultValue()+dval);

return context.getResultValue();

}

}

//减运算表达式

class SubExpression extends NumericExpression{

NumericItem NItem = new NumericItem();

@Override

public Integer interpret(Context context) {

if (context.getexpStr().length()==0) return null;

Integer dval;

dval=NItem.interpret(context);

if (dval==null) return null;

context.setResultValue(context.getResultValue()-dval);

return context.getResultValue();

}

}

//项

class NumericItem extends NumericExpression{

MultipleItem MItem;

DivideItem DItem;

NumericValue NVale;

@Override

public Integer interpret(Context context) {

if (context.getexpStr().length()==0) return null;

Integer dval;

NVale = new NumericValue();

dval=NVale.interpret(context);

if (dval==null) return null;

context.setItemValue(dval);

if (context.getexpStr().length()==0) return context.getItemVale();

String c=context.getexpStr().substring(0, 1);

while (c.equals("\*")||c.equals("/")) {

context.NextPosition(1);

switch (c) {

case "\*":

MItem = new MultipleItem();

MItem.interpret(context);

break;

case "/":

DItem = new DivideItem();

DItem.interpret(context);

break;

}

if (context.getexpStr().length()==0) return context.getItemVale();

c=context.getexpStr().substring(0, 1);

}

return context.getItemVale();

}

}

//乘项

class MultipleItem extends NumericExpression{

NumericValue NVale;

@Override

public Integer interpret(Context context) {

if (context.getexpStr().length()==0) return null;

Integer dval;

NVale = new NumericValue();

dval=NVale.interpret(context);

if (dval==null) return null;

context.setItemValue(context.getItemVale()\*dval);

return context.getItemVale();

}

}

//除项

class DivideItem extends NumericExpression{

NumericValue NVale;

@Override

public Integer interpret(Context context) {

if (context.getexpStr().length()==0) return null;

Integer dval;

NVale = new NumericValue();

dval=NVale.interpret(context);

if (dval==null) return null;

context.setItemValue(context.getItemVale()/dval);

return context.getItemVale();

}

}

//常数

class NumericValue extends NumericExpression{

Digit digit = new Digit();

@Override

public Integer interpret(Context context) {

if (context.getexpStr().length()==0) return null;

Integer dval;

dval=digit.interpret(context);

if (dval==null) return null;

Integer numericval=0;

while (dval!=null) {

numericval=numericval\*10+dval;

dval=digit.interpret(context);

}

return numericval;

}

}

//数字

class Digit extends NumericExpression{

@Override

public Integer interpret(Context context) {

if (context.getexpStr().length()==0) return null;

String c=context.getexpStr().substring(0, 1);

if (c.compareTo("0")>=0&&c.compareTo("9")<=0) {

context.NextPosition(1);

return Integer.parseInt(c);

} else return null;

}

}

public class InpterpreterPattern {

public static void main(String[] args) {

Context context = new Context();

NumericExpression nexp=new NumericExpression();

String expStr="12+6\*3-7-7/2";

context.setexpStr(expStr);

int x= nexp.interpret(context);

System.out.printf("表达式 %s = %d\n",expStr ,x);

}

}

**运行结果：**

表达式 12+6\*3-7 -7/2= 20

### **迭代器模式（Iterator Pattern）**

迭代器模式是针对集合对象而生的，对于集合对象而言，肯定会涉及到对集合的添加和删除操作，同时也肯定支持遍历集合元素的操作，我们此时可以把遍历操作放在集合对象中，但这样的话，集合对象既承担太多的责任了，面向对象设计原则中有一条就是单一职责原则，所以我们要尽可能地分离这些职责，用不同的类承担不同的责任，迭代器模式就是用迭代器类来承担遍历集合的职责。而且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（集合对象中）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。

**迭代器（Iterator）模式的定义：**迭代器模式提供了一种方法顺序访问一个聚合对象中的各个元素，而又无需暴露该对象的内部实现，这样既可以做到不暴露集合的内部结构，又可让外部代码透明地访问集合内部的数据。

**迭代器模式是一种对象行为型模式，其主要优点如下:**

使得访问一个聚合对象的内容而无需暴露它的内部表示，即迭代抽象。

为遍历不同的集合结构提供了一个统一的接口，从而支持同样的算法在不同的集合结构上进行操作

**其主要缺点是：**增加了类的个数，这在一定程度上增加了系统的复杂性。

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。

**迭代器模式的使用场景：**

访问一个集合对象的内容而无需暴露它的内部表示

为遍历不同的集合结构提供一个统一的接口

就目前来说，充分利用相关语言提供的工具即可，如 Java 的 Collection、List、Set、Map等。

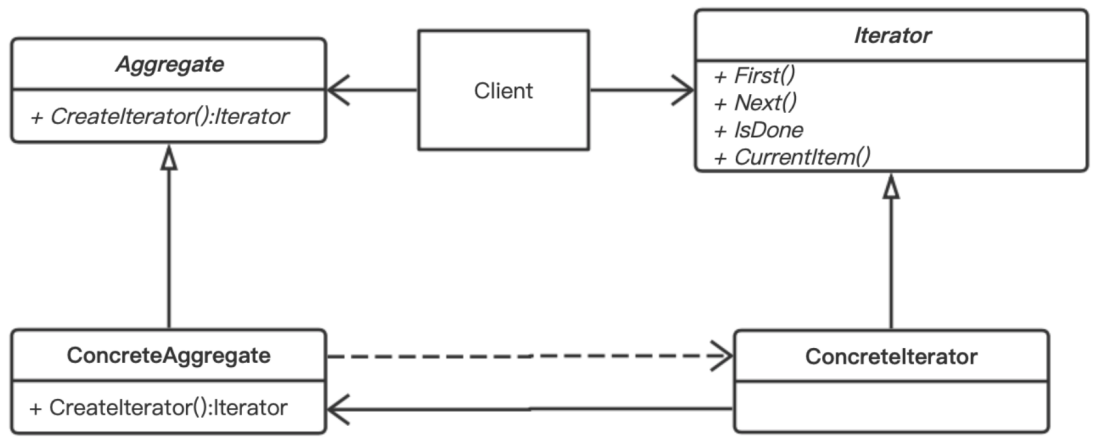
**迭代器模式的结构:**

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。例如java中的Collection接口，List接口，Set接口等。

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。比如ava中List接口的有序列表实现ArrayList，List接口的链表实现LinkedList,Set接口的哈希列表的实现HashSet等。

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 Next()、First()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。



### **中介者模式（Mediator Pattern）**

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。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”。这样的例子还有很多，例如，“房屋中介”、“人才交流中心”等。

**中介者（Mediator）模式的定义：**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。

中介者作为一种行为设计模式，它公开一个统一的接口，系统的不同对象或组件可以通过该接口进行通信。增加一个中介者对象后，所有的相关对象通过中介者对象来通信，而不是互相引用，所以当一个对象发生改变时，只需要通知中介者对象即可。

**中介者模式是一种对象行为型模式，其主要优点如下：**

降低了对象之间的耦合性，使得对象易于独立地被复用。

将对象间的多对多关联转变为一对多的关联，提高系统的灵活性，使得系统易于维护和扩展。

**中介者模式主要缺点：**

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。

**中介者模式使用场景：**

系统中对象之间存在比较复杂的引用关系，导致它们之间的依赖关系结构混乱而且难以复用该对象;

想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

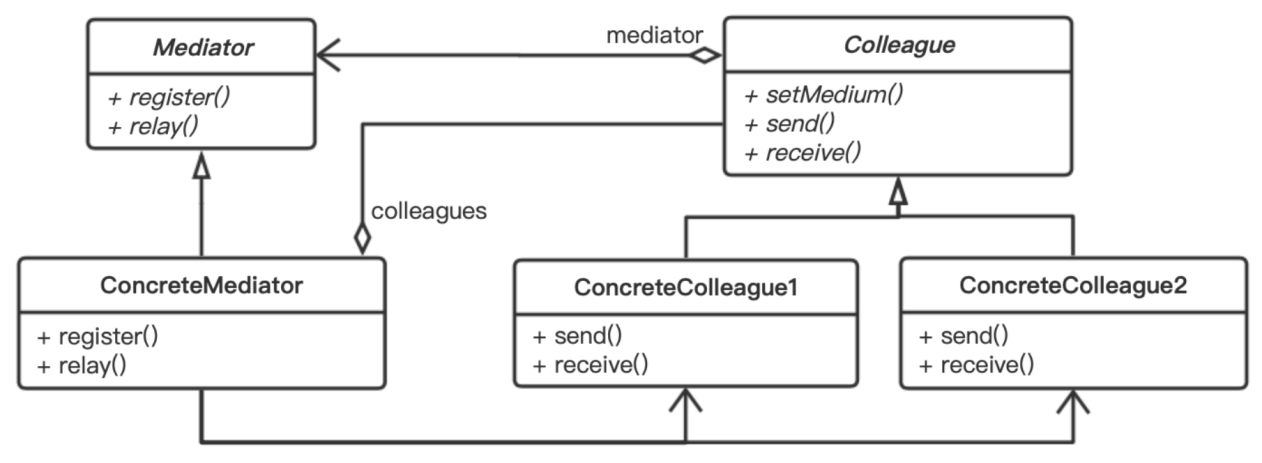
**中介者模式的结构：**

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。

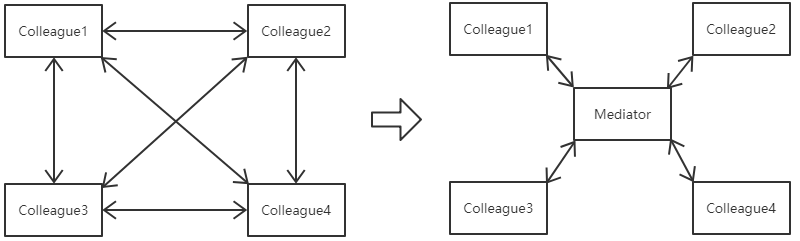
具体中介者（ConcreteMediator）角色：实现中介者接口，协调各个同事角色之间的交互关系，因此它依赖于同事角色。

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：是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**



**中介者模式将多方交互通信（网状结构）改成多方与一方的通信（星状结构）**

**实现：**

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

}

}

**运行结果：**

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。

**扩展实现：**

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.Broadcasat("Message 1");

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

c2.send(c1,"Message 2");

}

}

//抽象中介者

abstract class Mediator

{

public abstract void register(Colleague colleague);

public abstract void BroadcastRelay(Colleague cl，String Message); //转发广播

public abstract void relay(Colleague cl，String Message); //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 BroadcastRelay(Colleague cl，String Message) {

for(Colleague ob:colleagues) {

if(!ob.equals(cl)) {

((Colleague)ob).receive(Message);

}

}

}

public void relay(Colleague c2，String Message) {

if(colleagues.contains(c2)) {

C2.receive(Message);

}

}

}

//抽象同事类

abstract class Colleague

{

protected Mediator mediator;

public void setMedium(Mediator mediator) {

this.mediator=mediator;

}

public abstract void receive(String message);

public abstract void BroadCast(String message);

public abstract void send(ConcreteColleague c, String message);

}

//具体同事类

class ConcreteColleague1 extends Colleague

{

public void receive(String message) {

System.out.println("具体同事类1收到请求。");

}

public abstract void BroadCast(String message) {

System.out.println("具体同事类1发出广播请求。");

mediator.BroadcastRelay(this，message); //请中介者转发

}

public void send(ConcreteColleague c, String message) {

System.out.println("具体同事类1发出请求。");

mediator.relay(c，message); //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 abstract void BroadCast(String message) {

System.out.println("具体同事类2发出广播请求。");

mediator.BroadcastRelay(this，message); //请中介者转发

}

public void send(ConcreteColleague c, String message) {

System.out.println("具体同事类2发出请求。");

mediator.relay(c，message); //请中介者转发

}

}

### **备忘录模式（Memento Pattern）**

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。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。

**备忘录（Memento）模式的定义：**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。

**备忘录模式是一种对象行为型模式，其主要优点是：**

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。

**备忘录模式主要缺点：**

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。

**备忘录模式的应用场景：**

需要保存与恢复数据的场景。

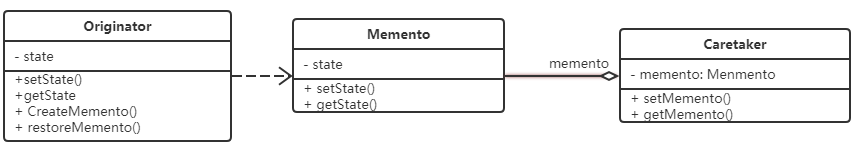
需要提供一个可回滚操作的场景。

**模式的结构：**

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。



**实现：**

package mementopattern;

public class MementoPattern

{

public static void main(String[] args) {

Originator originator =new Originator();

Caretaker caretaker=new Caretaker();

originator.setState("S0");

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

caretaker.setMemento(originator.createMemento()); //保存状态

originator.setState("S1");

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

originator.restoreMemento(caretaker.getMemento()); //恢复状态

System.out.println("恢复状态:"+originator.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 writeMemento(Memento m) {

m.setState(state);

}

public void restoreMemento(Memento m) {

this.setState(m.getState());

}

}

//管理者

class Caretaker

{

private Memento memento; //也可以是一个列表。若是列表，则下述方法需要给出标识码

public void setMemento(Memento m) {

memento=m;

}

public Memento getMemento() {

return memento;

}

}

**运行结果：**

初始状态:S0

新的状态:S1

恢复状态:S0

**应用举例：**将状态保存在文件中，可以在程序下次运行时根据备份赋初值，即恢复到上次的状态。

package mementopattern;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class MementoPattern

{

public static void main(String[] args)

{

Originator originator =new Originator();

Caretaker caretaker=new Caretaker();

originator.setState(0);

caretaker.setMemento(originator.createMemento()); //创建或打开备忘录

originator.restoreMemento(caretaker.getMemento()); //恢复状态

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

originator.setState(originator.getState()+1);

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

originator.writeMemento(caretaker.getMemento());

}

}

//备忘录

class Memento

{

private String state;

String MementoFile="memento.txt";

File af;

FileWriter wf;

BufferedReader bReader;//new一个BufferedReader对象，将文件内容读取到缓存

public Memento(String state)

{

af = new File(MementoFile);

if (!af.exists()){

try {

af.createNewFile();

wf = new FileWriter(af);

bReader = new BufferedReader(new FileReader(af));

wf.write(state);

wf.close();

} catch (IOException e) {

}

}

}

public void setState(String state)

{

try {

wf = new FileWriter(af);

wf.write(state);

wf.close();

} catch (IOException e) {

}

}

public String getState()

{

String inLine=null;

try {

bReader = new BufferedReader(new FileReader(af));

inLine = bReader.readLine();

} catch (IOException e) {

}

return inLine;

}

}

//发起人

class Originator

{

private int state;

public void setState(int state)

{

this.state=state;

}

public int getState()

{

return state;

}

public Memento createMemento()

{

return new Memento(String.valueOf(state));

}

public void writeMemento(Memento m)

{

m.setState(String.valueOf(state));

}

public void restoreMemento(Memento m)

{

String state=m.getState();

if (state==null) state="0";

this.setState(Integer.parseInt(state));

}

}

//管理者

class Caretaker

{

private Memento memento; //也可以是一个列表。若是列表，则下述方法需要给出标识码

public void setMemento(Memento m)

{

memento=m;

}

public Memento getMemento()

{

return memento;

}

}

### **观察者模式（Observer Pattern）**

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。

**观察者（Observer）模式的定义：**指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（Pub-Sub Pattern）**。

观察者模式是一种对象行为型模式，**其主要优点**:

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;

目标与观察者之间建立了一套触发机制。

**观察者模式的主要缺点:**

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。

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。

**观察者模式的应用场景：**

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。

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。

实现观察者模式时要注意具体主题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。

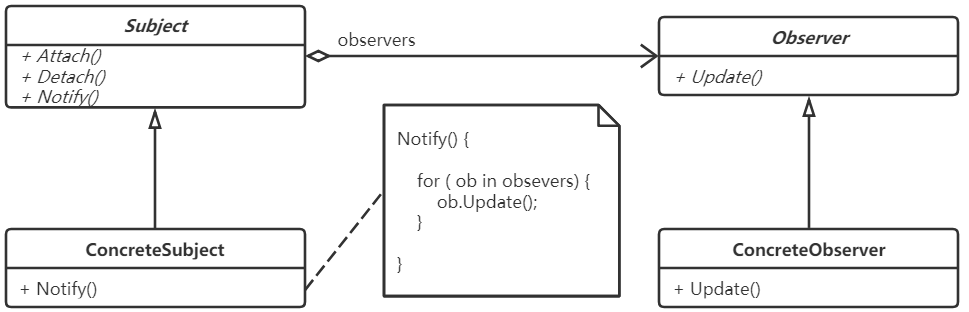
**观察者模式的结构：**

抽象主题（Subject）角色：也叫抽象目标类、抽象发布者，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、具体发布者，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：也叫抽象订阅者，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：也叫具体订阅者，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。



**实现：**

package observer;

public class ObserverPattern

{

public static void main(String[] args) {

Subject subject=new ConcreteSubject();

Observer obs1=new ConcreteObserver1();

Observer obs2=new ConcreteObserver2();

subject.Attach(obs1);

subject.Attach(obs2);

subject.Notify();

}

}

//抽象目标

abstract class Subject

{

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

public void Attach(Observer observer) { //增加观察者方法

observers.add(observer);

}

public void Detach(Observer observer) { //删除观察者方法

observers.remove(observer);

}

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

}

//具体目标

class ConcreteSubject extends Subject

{

public void Notify() {

System.out.println("具体目标发生改变...");

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

for(Object ob:observers) {

((Observer)ob).Update();

}

}

}

//抽象观察者

interface Observer

{

void Update(); //反应

}

//具体观察者1

class ConcreteObserver1 implements Observer

{

public void Update() {

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 Update() {

System.out.println("具体观察者2作出反应！");

}

}

**运行结果：**

具体目标发生改变...

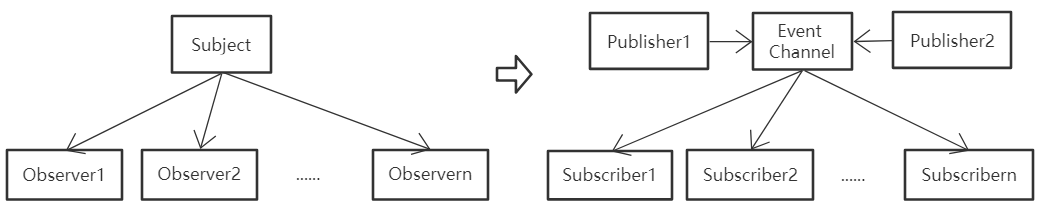
--------------

具体观察者1作出反应！

具体观察者2作出反应！

**观察者模式的扩展：**

上述观察者模式可以进一步改成扩展的发布订阅模式，在主题与观察者之间加入一个信息中转中心，以降低信息发布与订阅者之间的耦合度。请同学们根据下图， 参照观察者模式和中介者模式，自行设计出相应的模式结构及实现代码。



### **状态模式（State Pattern）**

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。

**状态（State）模式的定义：**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。

**状态模式是一种对象行为型模式，其主要优点：**

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

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。

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。

**状态模式的主要缺点：**

状态模式的使用必然会增加系统的类与对象的个数。

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。

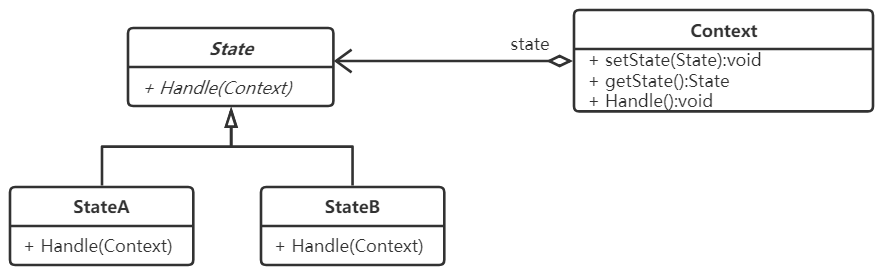
状态模式把受环境改变的对象行为包装在不同的状态对象里，其意图是让一个对象在其内部状态改变的时候，其行为也随之改变。

**状态模式的结构：**

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。



**实现：**

package statepattern;

public class StatePattern

{

public static void main(String[] args)

{

Context context=new Context(); //创建环境

context.Handle(); //处理请求

}

}

//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()

{

while (state!=null) //通常要有结束状态

state.Handle(this);

}

}

//抽象状态类

abstract class State

{

public abstract void Handle(Context context); //一般需要对Context的状态进行修改

}

//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(null);

}

}

**运行结果：**

当前状态是 A.

当前状态是 B.

**简单应用，**对一个整数在不同数值下的不同处理方式**：**

package statepattern;

public class StatePattern

{

public static void main(String[] args)

{

Context context=new Context(); //创建环境

context.Handle(); //处理请求

}

}

//环境（上下文）类

class Context

{

public int CountTest;

int countstate;

private State state;

//定义环境类的初始状态

public Context()

{

CountTest=0;

Countstate=0;

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()

{

/\*

while (CountTest>=0)

{

if (Countstate==0) {

CountTest+=10;

if (CountTest>=100) Countstate=1;

} else if (Countstate==1) {

CountTest\*=10;

if (CountTest>=100000) Countstate=2;

} else if (Countstate==2) {

CountTest/=5;

if (CountTest<=100) Countstate=3;

} else if (Countstate==3) {

CountTest-=4;

if (CountTest<=0) break;

}

\*’/

while (state!=null) //通常要有结束状态

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)

{

context.CountTest += 10;

System.out.println("当前状态是 A, count="+context.CountTest);

if (context.CountTest>=100)

context.setState(new ConcreteStateB());

}

}

//具体状态B类

class ConcreteStateB extends State

{

public void Handle(Context context)

{

context.CountTest \*= 10;

System.out.println("当前状态是 B, count="+context.CountTest);

if (context.CountTest>=100000)

context.setState(new ConcreteStateC());

}

}

//具体状态C类

class ConcreteStateC extends State

{

public void Handle(Context context)

{

context.CountTest /= 5;

System.out.println("当前状态是 C, count="+context.CountTest);

if (context.CountTest<=100)

context.setState(new ConcreteStateD());

}

}

//具体状态D类

class ConcreteStateD extends State

{

public void Handle(Context context)

{

context.CountTest -= 4;

System.out.println("当前状态是 D, count="+context.CountTest);

if (context.CountTest<=0)

context.setState(null);

}

}

**运行结果：**

当前状态是 A, count=10

当前状态是 A, count=20

当前状态是 A, count=30

当前状态是 A, count=40

当前状态是 A, count=50

当前状态是 A, count=60

当前状态是 A, count=70

当前状态是 A, count=80

当前状态是 A, count=90

当前状态是 A, count=100

当前状态是 B, count=1000

当前状态是 B, count=10000

当前状态是 B, count=100000

当前状态是 C, count=20000

当前状态是 C, count=4000

当前状态是 C, count=800

当前状态是 C, count=160

当前状态是 C, count=32

当前状态是 D, count=28

当前状态是 D, count=24

当前状态是 D, count=20

当前状态是 D, count=16

当前状态是 D, count=12

当前状态是 D, count=8

当前状态是 D, count=4

当前状态是 D, count=0

### **策略模式（Strategy Pattern）**

在软件开发中常常遇到类似这样的情况：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。

**策略（Strategy）模式的定义：**定义一系列算法，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。

**策略模式的主要优点：**

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。

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。

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。

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。

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。

**策略模式的主要缺点：**

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。

策略模式造成很多的策略类。

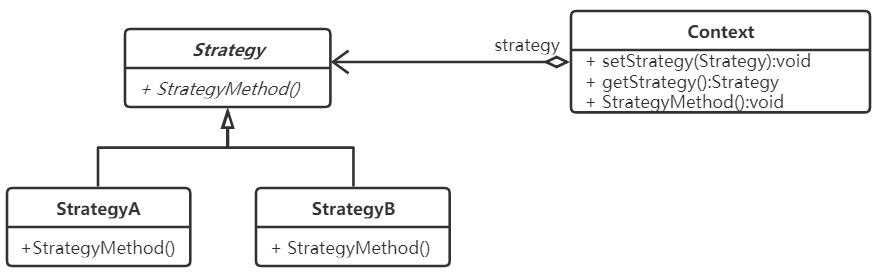
策略模式是准备一组算法，并将这组算法封装到一系列的策略类里面，作为一个抽象策略类的子类。策略模式的重心不是如何实现算法，而是如何组织这些算法，从而让程序结构更加灵活，具有更好的维护性和扩展性。

**策略模式的结构：**

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。

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。

环境（Context）类：又称上下文，持有一个策略类的引用，最终给客户端调用。



**注意：**

策略模式与状态模式的结构相同，但它们的应用方法却是不同的。

状态对用户是透明的，在一次运算过程中，状态是变化的，是根据上下文内容确定的，用户不能控制状态变化。

策略的选择是由用户根据处理需求事先确定的，在一次运算过程中是不变的。

**实现：**

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！");

}

}

//上下文（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(); //注意此处与状态模式的区别

}

}

**运行结果：**

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！

### **模板模式（Template Pattern）**

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。

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。

模板方法模式将解决以上类似的问题。

**模板方法（Template Method）模式的定义：**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。**

**模板方法模式的主要优点：**

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。

它在父类中提取了公共的部分代码，便于代码复用。

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。

**模板方法模式的主要缺点：**

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。

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。

**模板方法模式的应用场景：**

一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现；

各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复；

控制子类的扩展。

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。

**模板方法模式的结构“**

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TemplateMethod：定义了算法的骨架，按某种顺序调用其包含的基本方法。

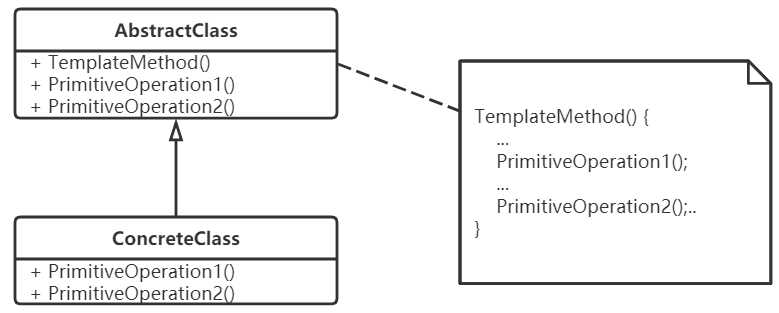
② 基本方法PrimitiveOperation：是整个算法中的一个步骤，包含以下几种类型。

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。



**实现：**

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

{

System.out.println("模板方法内部分操作1...");

PrimitiveOperation1();

System.out.println("模板方法内部分操作2...");

PrimitiveOperation2();

System.out.println("模板方法内部分操作3...");

}

public void PrimitiveOperation1() //具体方法

{

System.out.println("抽象类中的PrimitiveOperation1具体方法被调用...");

}

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

}

//具体子类

class ConcreteClass extends AbstractClass

{

@Override

public void PrimitiveOperation1()

{

System.out.println("具体方法PrimitiveOperation1的覆盖被调用...");

}

@Override

public void PrimitiveOperation2()

{

System.out.println("抽象方法PrimitiveOperation2的实现被调用...");

}

}

运行结果：

模板方法内部分操作1...  
具体方法PrimitiveOperation1的覆盖被调用...

模板方法内部分操作2...

抽象方法PrimitiveOperation2的实现被调用...

模板方法内部分操作3...

### **访问者模式（Visitor Pattern）**

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。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。

**访问者（Visitor）模式的定义：**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。

访问者模式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，是一种对象行为型模式。

**访问者模式主要优点：**

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。

**访问者模式的主要缺点：**

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。

**访问者模式的应用场景：**

通常在以下情况可以考虑使用访问者（Visitor）模式。

对象结构相对稳定，但其操作算法经常变化的程序。

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。

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。

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。

**访问者模式的结构：**

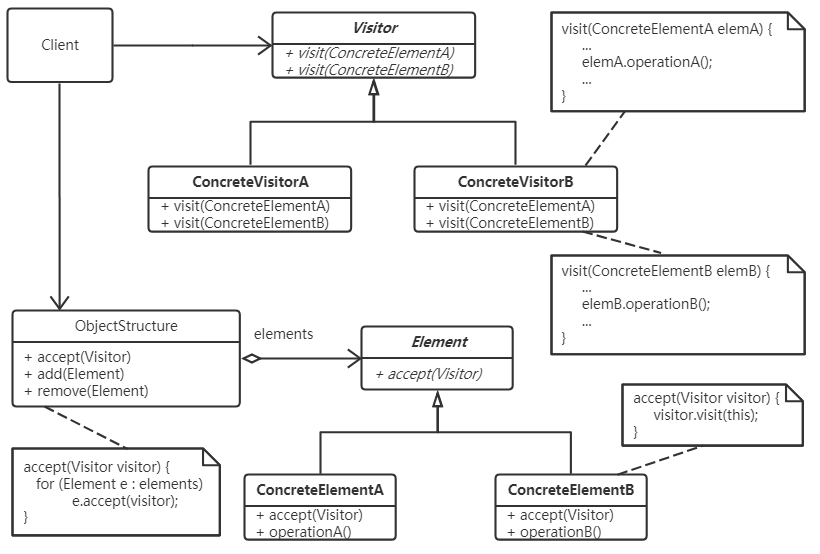
抽象访问者（Visitor）角色：定义一个访问具体元素的接口，为每个具体元素类对应一个访问操作 visit() ，该操作中的参数类型标识了被访问的具体元素。

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：是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。



**访问者模式的扩展：**

访问者（Visitor）模式常常同“组合模式”联用。因为访问者模式中的“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。

**实现：**

package visitorpattern;

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 elemA);

void visit(ConcreteElementB elemB);

}

//具体访问者A类

class ConcreteVisitorA implements Visitor

{

@Override

public void visit(ConcreteElementA elemA)

{

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

}

@Override

public void visit(ConcreteElementB elemB)

{

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

}

}

//具体访问者B类

class ConcreteVisitorB implements Visitor

{

@Override

public void visit(ConcreteElementA elemA)

{

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

}

@Override

public void visit(ConcreteElementB elemB)

{

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

}

}

//抽象元素类

interface Element

{

void accept(Visitor visitor);

}

//具体元素A类

class ConcreteElementA implements Element

{

@Override

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

{

@Override

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> elements=new ArrayList<>();

public void accept(Visitor visitor)

{

//此处也可以使用迭代器模式对列表elements中元素进行访问

for (Element element : elements) {

element.accept(visitor);

}

}

public void add(Element element)

{

elements.add(element);

}

public void remove(Element element)

{

elements.remove(element);

}

}

**运行结果：**

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。

## 设计模式总结

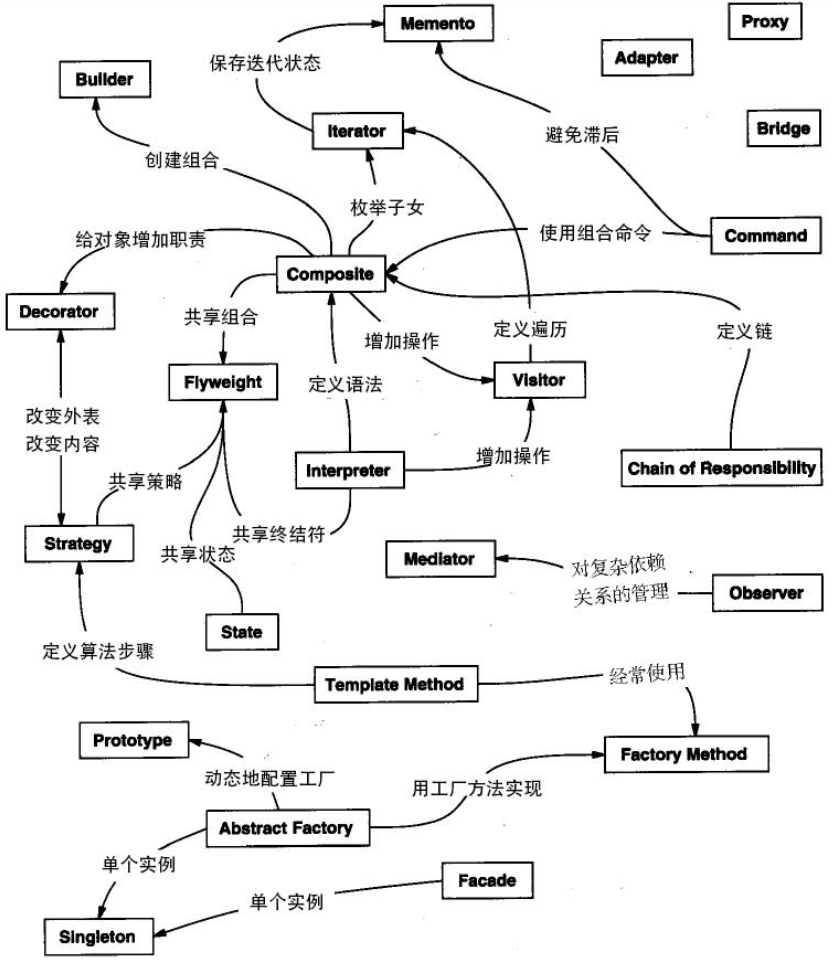
### **设计模式类型间的关系**

创建型模式→结构型模式→行为型模式

行为型模式**使用**结构型模式**使用**创建型模式

行为模式要有结构，在行为模式和结构模式中要创建对象。

**各种模式的组合使用：**



### **设计模式应用问题测试**

以下问题可以个人独自完成，也可以多人合作完成，并在完成的基础上形成报告，作为期终考核成绩的一部分。多人合作时，要有一份总体报告，阐述每个问题整体设计思想、程序结构、模式选用及每人在该问题设计工作中的分工。除此之外，无论是个人独立完成还是多人合作完成，每个人都要有一份个人报告，在简述个人对本门课程的理解之外，重点阐述个人所负责部分的设计思想、程序结构、模式选用及设计心得。

注意：多人合作时，要有明确的分工，组织者不一定获得高分。每个人的最后成绩依据整体任务完成情况、分工性质、个人工作量、个人完成度、个人完成质量、报告内容而定。

以下问题的相关概念：

**汉字：**GB2312中16区之后（含16区）所规定的中文字符

**英文字：**由ASCII码表中英文字符构成的连续字符串

**段：**由换行符’\n’分隔的字符串，连续多个换行符’\n’应当成一段的分隔

**词：**由多个字（汉字或英文字）构成的连续字串。词一般应该多次出现，其出现的次数叫做词频。

**问题一：**设计一个程序，对于给定的一个文本文件，统计在该文本文件中包含的所有字（汉字或英文字）出现的次数（字频），结果输出到指定的文本文件。

**问题二：**设计一个程序，对于给定的一个文本文件，在问题一的基础上，统计在该文本文件中所包含的词频大于给定数值p（p缺省值为3）的词的词频及每个词所出现的位置（段号及在段中的偏移量），结果输出到指定的文本文件。**提示：**如果词A和B构成一个新词AB，则AB词频不可能大于A或B的词频。

**问题三：**设计一个程序，对于给定的两篇WORD文档，在问题二的基础上，根据词频结果判断这两篇WORD文档内容主题的相似度（高频词的相似度。可以自行定义相似度的概念），形成相似度比较报告（WORD文档，自定义内容格式）。

**问题四：**在上述问题基础上，提出可能的新的应用，并给出相应设计方案。

以上前三个问题，在最终报告后，要附有相关程序代码文件及测试案例（案例最好多样，有一定的代表性）和测试结果。