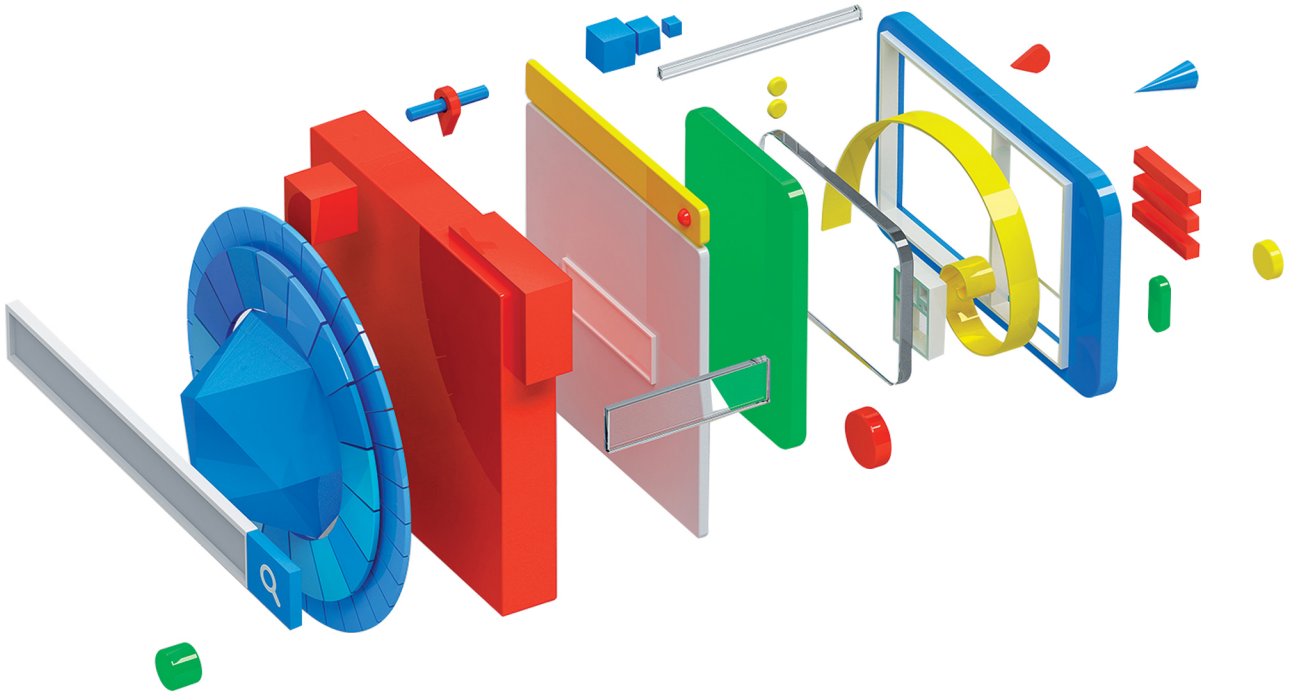
完整介绍 Java 23 种设计模式

23 种设计模式



极客学院出版

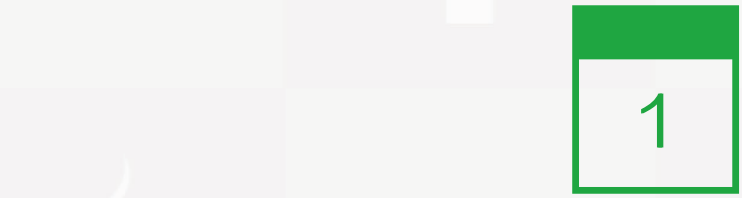
前言

设计模式是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。本文将介绍23种设计模式。

本教程内容来源于 卡奴达摩专栏（持续更新中）

更新日期	更新内容
2015-04-21	23 种设计模式

目录



23 种设计模式

单例模式

定义：确保一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

类型：创建类模式

类图：

图片 1.1 singleton

类图知识点：

1.类图分为三部分，依次是类名、属性、方法

2.以<<开头和以>>结尾的为注释信息

3.修饰符+代表public，-代表private，#代表protected，什么都没有代表包可见。

4.带下划线的属性或方法代表是静态的。

5.对类图中对象的关系不熟悉的朋友可以参考文章：设计模式中类的关系。

单例模式应该是23种设计模式中最简单的一种模式了。它有以下几个要素：

- 私有的构造方法
- 指向自己实例的私有静态引用
- 以自己实例为返回值的静态的公有的方法

单例模式根据实例化对象时机的不同分为两种：一种是饿汉式单例，一种是懒汉式单例。饿汉式单例在单例类被加载时候，就实例化一个对象交给自己的引用；而懒汉式在调用取得实例方法的时候才会实例化对象。代码如下：

饿汉式单例

```
public class Singleton {
    private static Singleton singleton = new Singleton();
    private Singleton(){}
    public static Singleton getInstance(){
        return singleton;
    }
}
```

懒汉式单例

```
public class Singleton {
    private static Singleton singleton;
    private Singleton(){}

    public static synchronized Singleton getInstance(){
        if(singleton==null){
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

单例模式的优点：

- 在内存中只有一个对象，节省内存空间。
- 避免频繁的创建销毁对象，可以提高性能。
- 避免对共享资源的多重占用。
- 可以全局访问。

适用场景：由于单例模式的以上优点，所以是编程中用的比较多的一种设计模式。我总结了一下我所知道的适合使用单例模式的场景：

- 需要频繁实例化然后销毁的对象。
- 创建对象时耗时过多或者耗资源过多，但又经常用到的对象。
- 有状态的工具类对象。
- 频繁访问数据库或文件的对象。
- 以及其他我没用过的所有要求只有一个对象的场景。

单例模式注意事项：

- 只能使用单例类提供的方法得到单例对象，不要使用反射，否则将会实例化一个新对象。

- 不要做断开单例类对象与类中静态引用的危险操作。
- 多线程使用单例使用共享资源时，注意线程安全问题。

关于java中单例模式的一些争议：

单例模式的对象长时间不用会被jvm垃圾收集器收集吗

看到不少资料中说：如果一个单例对象在内存中长久不用，会被jvm认为是一个垃圾，在执行垃圾收集的时候会被清理掉。对此这个说法，笔者持怀疑态度，笔者本人的观点是：**在hotspot虚拟机1.6版本中，除非人为地断开单例中静态引用到单例对象的联接，否则jvm垃圾收集器是不会回收单例对象的。**

对于这个争议，笔者单独写了一篇文章进行讨论，如果您有不同的观点或者有过这方面的经历请进入文章单例模式讨论篇：单例模式与垃圾收集参与讨论。

在一个jvm中会出现多个单例吗

在分布式系统、多个类加载器、以及序列化的的情况下，会产生多个单例，这一点是无庸置疑的。那么在同一个jvm中，会不会产生单例呢？使用单例提供的getInstance()方法只能得到同一个单例，除非是使用反射方式，将会得到新的单例。代码如下

```
Class c = Class.forName(Singleton.class.getName());
Constructor ct = c.getDeclaredConstructor();
ct.setAccessible(true);
Singleton singleton = (Singleton)ct.newInstance();
```

这样，每次运行都会产生新的单例对象。所以运用单例模式时，一定注意不要使用反射产生新的单例对象。

懒汉式单例线程安全吗

主要是网上的一些说法，懒汉式的单例模式是线程不安全的，即使是在实例化对象的方法上加synchronized关键字，也依然是危险的，但是笔者经过编码测试，发现加synchronized关键字修饰后，虽然对性能有部分影响，但是却是线程安全的，并不会产生实例化多个对象的情况。

单例模式只有饿汉式和懒汉式两种吗

饿汉式单例和懒汉式单例只是两种比较主流和常用的单例模式方法，从理论上讲，任何可以实现一个类只有一个实例的设计模式，都可以称为单例模式。

单例类可以被继承吗

饿汉式单例和懒汉式单例由于构造方法是private的，所以他们都是不可继承的，但是其他很多单例模式是可以继承的，例如登记式单例。

饿汉式单例好还是懒汉式单例好

在java中，饿汉式单例要优于懒汉式单例。C++中则一般使用懒汉式单例。

单例模式比较简单，在此就不举例代码演示了。

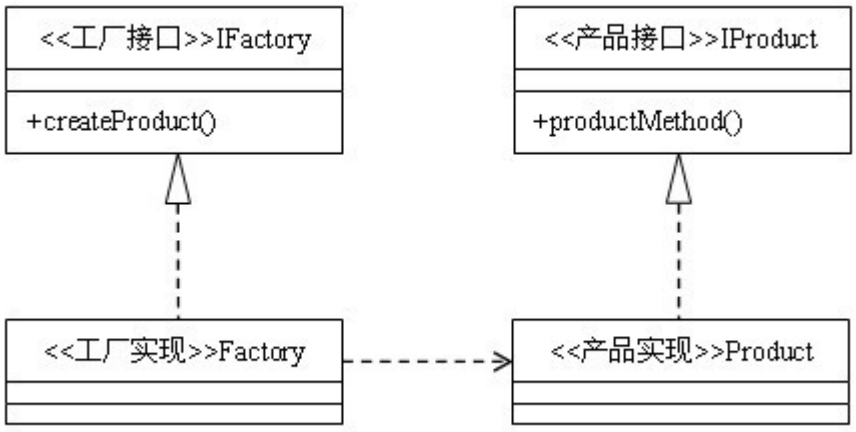
工厂方法模式

定义：定义一个用于创建对象的接口，让子类决定实例化哪一个类，工厂方法使一个类的实例化延迟到其子类。

类型：创建类模式

类图：

!



图片 1.2 factory

工厂方法模式代码

```
interface IProduct {
    public void productMethod();
}

class Product implements IProduct {
    public void productMethod() {
        System.out.println("产品");
    }
}

interface IFactory {
    public IProduct createProduct();
}

class Factory implements IFactory {
    public IProduct createProduct() {
        return new Product();
```

```
    }
}

public class Client {
    public static void main(String[] args) {
        IFactory factory = new Factory();
        IProduct prodect = factory.createProduct();
        prodect.productMethod();
    }
}
```

工厂模式：

首先需要说一下工厂模式。工厂模式根据抽象程度的不同分为三种：简单工厂模式（也叫静态工厂模式）、本文所讲述的**工厂方法模式**、以及抽象工厂模式。工厂模式是编程中经常用到的一种模式。它的主要优点有：

- 可以使代码结构清晰，有效地封装变化。在编程中，产品类的实例化有时候是比较复杂和多变的，通过工厂模式，将产品的实例化封装起来，使得调用者根本无需关心产品的实例化过程，只需依赖工厂即可得到自己想要的产品。
- 对调用者屏蔽具体的产品类。如果使用工厂模式，调用者只关心产品的接口就可以了，至于具体的实现，调用者根本无需关心。即使变更了具体的实现，对调用者来说没有任何影响。
- 降低耦合度。产品类的实例化通常来说是很复杂的，它需要依赖很多的类，而这些类对于调用者来说根本无需知道，如果使用了工厂方法，我们需要做的仅仅是实例化好产品类，然后交给调用者使用。对调用者来说，产品所依赖的类都是透明的。

工厂方法模式：

通过工厂方法模式的类图可以看到，工厂方法模式有四个要素：

- 工厂接口。工厂接口是工厂方法模式的核心，与调用者直接交互用来提供产品。在实际编程中，有时候也会使用一个抽象类来作为与调用者交互的接口，其本质上是一样的。
- 工厂实现。在编程中，工厂实现决定如何实例化产品，是实现扩展的途径，需要有多少种产品，就需要有多少个具体的工厂实现。
- 产品接口。产品接口的主要目的是定义产品的规范，所有的产品实现都必须遵循产品接口定义的规范。产品接口是调用者最为关心的，产品接口定义的优劣直接决定了调用者代码的稳定性。同样，产品接口也可以用抽象类来代替，但要注意最好不要违反里氏替换原则。
- 产品实现。实现产品接口的具体类，决定了产品在客户端中的具体行为。

前文提到的简单工厂模式跟工厂方法模式极为相似，区别是：简单工厂只有三个要素，他没有工厂接口，并且得到产品的方法一般是静态的。因为没有工厂接口，所以在工厂实现的扩展性方面稍弱，可以算所工厂方法模式的简化版，关于简单工厂模式，在此一笔带过。

适用场景：

不管是简单工厂模式，工厂方法模式还是抽象工厂模式，他们具有类似的特性，所以他们的适用场景也是类似的。

首先，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。

其次，工厂模式是一种典型的解耦模式，迪米特法则在工厂模式中表现的尤为明显。假如调用者自己组装产品需要增加依赖关系时，可以考虑使用工厂模式。将会大大降低对象之间的耦合度。

再次，由于工厂模式是依靠抽象架构的，它把实例化产品的任务交由实现类完成，扩展性比较好。也就是说，当需要系统有比较好的扩展性时，可以考虑工厂模式，不同的产品用不同的实现工厂来组装。

典型应用

要说明工厂模式的优点，可能没有比组装汽车更合适的例子了。场景是这样的：汽车由发动机、轮、底盘组成，现在需要组装一辆车交给调用者。假如不使用工厂模式，代码如下：

```
class Engine {
    public void getStyle(){
        System.out.println("这是汽车的发动机");
    }
}
class Underpan {
    public void getStyle(){
        System.out.println("这是汽车的底盘");
    }
}
class Wheel {
    public void getStyle(){
        System.out.println("这是汽车的轮胎");
    }
}
public class Client {
    public static void main(String[] args) {
        Engine engine = new Engine();
        Underpan underpan = new Underpan();
```

```
        Wheel wheel = new Wheel();
        ICar car = new Car(underpan, wheel, engine);
        car.show();
    }
}
```

可以看到，调用者为了组装汽车还需要另外实例化发动机、底盘和轮胎，而这些汽车的组件是与调用者无关的，严重违反了迪米特法则，耦合度太高。并且非常不利于扩展。另外，本例中发动机、底盘和轮胎还是比较具体的，在实际应用中，可能这些产品的组件也都是抽象的，调用者根本不知道怎样组装产品。假如使用工厂方法的话，整个架构就显得清晰了许多。

```
interface IFactory {
    public ICar createCar();
}
class Factory implements IFactory {
    public ICar createCar() {
        Engine engine = new Engine();
        Underpan underpan = new Underpan();
        Wheel wheel = new Wheel();
        ICar car = new Car(underpan, wheel, engine);
        return car;
    }
}
public class Client {
    public static void main(String[] args) {
        IFactory factory = new Factory();
        ICar car = factory.createCar();
        car.show();
    }
}
```

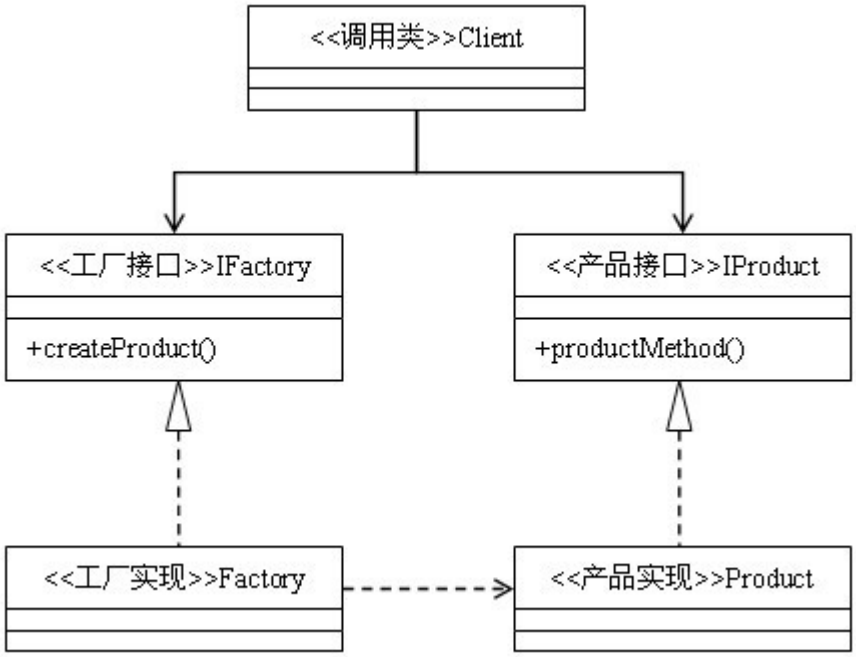
使用工厂方法后，调用端的耦合度大大降低了。并且对于工厂来说，是可以扩展的，以后如果想组装其他的汽车，只需要再增加一个工厂类的实现就可以。无论是灵活性还是稳定性都得到了极大的提高。

抽象工厂模式

定义：为创建一组相关或相互依赖的对象提供一个接口，而且无需指定他们的具体类。

类型：创建类模式

类图：

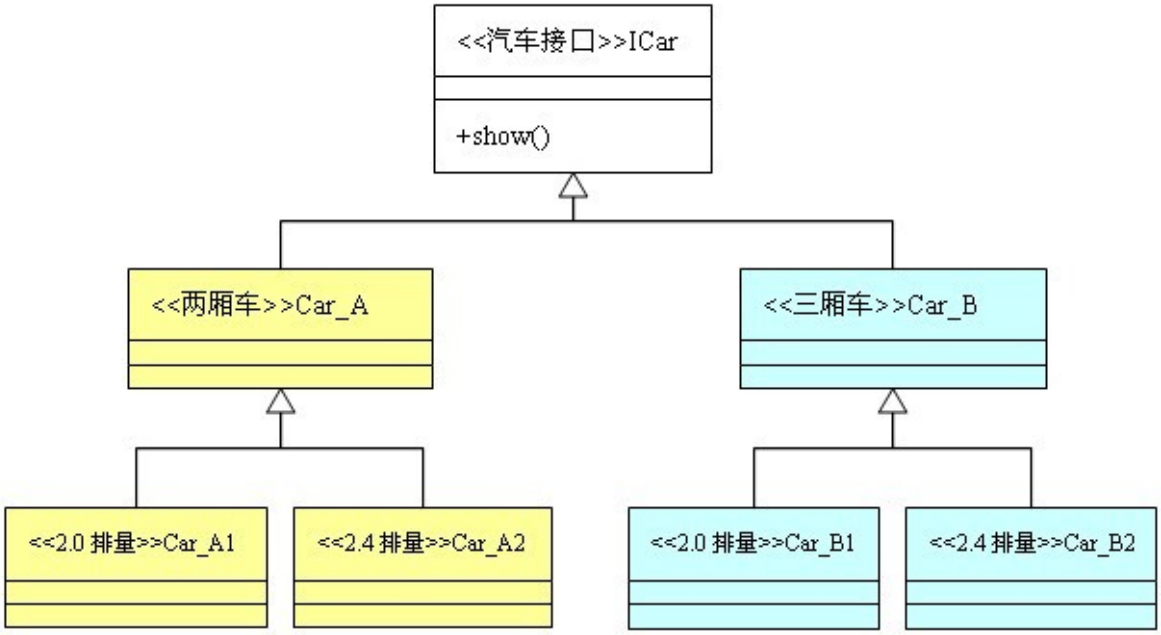


图片 1.3 abstract-factory-pattern

抽象工厂模式与工厂方法模式的区别

抽象工厂模式是工厂方法模式的升级版本，他用来创建一组相关或者相互依赖的对象。他与工厂方法模式的区别就在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则是针对的多个产品等级结构。在编程中，通常一个产品结构，表现为一个接口或者抽象类，也就是说，工厂方法模式提供的所有产品都是衍生自同一个接口或抽象类，而抽象工厂模式所提供的产品则是衍生自不同的接口或抽象类。

在抽象工厂模式中，有一个**产品族**的概念：所谓的产品族，是指**位于不同产品等级结构中功能相关联的产品组成的家族**。抽象工厂模式所提供的一系列产品就组成一个产品族；而工厂方法提供的一系列产品称为一个等级结构。我们依然拿生产汽车的例子来说明他们之间的区别。



图片 1.4 abstract-factory-pattern

在上面的类图中，两厢车和三厢车称为两个不同的等级结构；而2.0排量车和2.4排量车则称为两个不同的产品族。再具体一点，2.0排量两厢车和2.4排量两厢车属于同一个等级结构，2.0排量三厢车和2.4排量三厢车属于另一个等级结构；而2.0排量两厢车和2.0排量三厢车属于同一个产品族，2.4排量两厢车和2.4排量三厢车属于另一个产品族。

明白了等级结构和产品族的概念，就理解工厂方法模式和抽象工厂模式的区别了，如果工厂的产品全部属于同一个等级结构，则属于工厂方法模式；如果工厂的产品来自多个等级结构，则属于抽象工厂模式。在本例中，如果一个工厂模式提供2.0排量两厢车和2.4排量两厢车，那么他属于工厂方法模式；如果一个工厂模式是提供2.4排量两厢车和2.4排量三厢车两个产品，那么这个工厂模式就是抽象工厂模式，因为他提供的产品是分属两个不同的等级结构。当然，如果一个工厂提供全部四种车型的产品，因为产品分属两个等级结构，他当然也属于抽象工厂模式了。

抽象工厂模式代码

```
interface IProduct1 {
    public void show();
}
interface IProduct2 {
    public void show();
}

class Product1 implements IProduct1 {
    public void show() {
```

```
        System.out.println("这是1型产品");
    }
}
class Product2 implements IProduct2 {
    public void show() {
        System.out.println("这是2型产品");
    }
}

interface IFactory {
    public IProduct1 createProduct1();
    public IProduct2 createProduct2();
}
class Factory implements IFactory{
    public IProduct1 createProduct1() {
        return new Product1();
    }
    public IProduct2 createProduct2() {
        return new Product2();
    }
}

public class Client {
    public static void main(String[] args){
        IFactory factory = new Factory();
        factory.createProduct1().show();
        factory.createProduct2().show();
    }
}
```

抽象工厂模式的优点

抽象工厂模式除了具有工厂方法模式的优点外，最主要的优点就是可以在类的内部对产品族进行约束。所谓的产品族，一般或多或少的都存在一定的关联，抽象工厂模式就可以在类内部对产品族的关联关系进行定义和描述，而不必专门引入一个新的类来进行管理。

抽象工厂模式的缺点

产品族的扩展将是一件十分费力的事情，假如产品族中需要增加一个新的产品，则几乎所有的工厂类都需要进行修改。所以使用抽象工厂模式时，对产品等级结构的划分是非常重要的。

适用场景

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，便可以使用抽象工厂模式。说的更明白一点，就是一个继承体系中，如果存在着多个等级结构（即存在着多个抽象类），并且分属各个等级结构中的实现类之间存

在着一定的关联或者约束，就可以使用抽象工厂模式。假如各个等级结构中的实现类之间不存在关联或约束，则使用多个独立的工厂来对产品进行创建，则更合适一点。

总结

无论是简单工厂模式，工厂方法模式，还是抽象工厂模式，他们都属于工厂模式，在形式和特点上也是极为相似的，他们的最终目的都是为了解耦。在使用时，我们不必去在意这个模式到底工厂方法模式还是抽象工厂模式，因为他们之间的演变常常是令人琢磨不透的。经常你会发现，明明使用的工厂方法模式，当新需求来临，稍加修改，加入了一个新方法后，由于类中的产品构成了不同等级结构中的产品族，它就变成抽象工厂模式了；而对于抽象工厂模式，当减少一个方法使的提供的产品不再构成产品族之后，它就演变成了工厂方法模式。

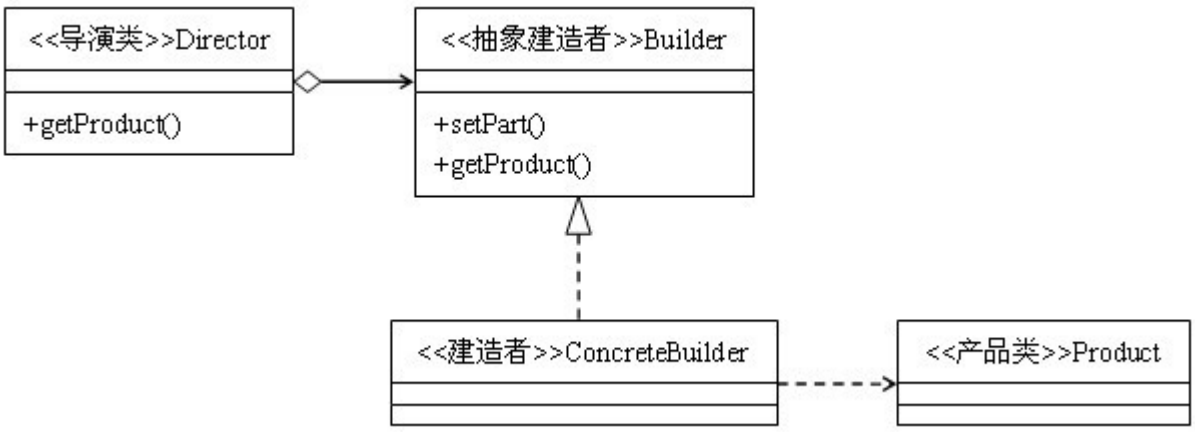
所以，在使用工厂模式时，只需要关心降低耦合度的目的是否达到了。

建造者模式

定义：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

类型：创建类模式

类图：



图片 1.5 builder-pattern

四个要素

- **产品类：**一般是一个较为复杂的对象，也就是说创建对象的过程比较复杂，一般会有比较多的代码量。在本类图中，产品类是一个具体的类，而非抽象类。实际编程中，产品类可以是由一个抽象类与它的不同实现组成，也可以是由多个抽象类与他们的实现组成。
- **抽象建造者：**引入抽象建造者的目的，是为了将建造的具体过程交与它的子类来实现。这样更容易扩展。一般至少会有两个抽象方法，一个用来建造产品，一个是用来返回产品。
- **建造者：**实现抽象类的所有未实现的方法，具体来说一般是两项任务：组建产品；返回组建好的产品。
- **导演类：**负责调用适当的建造者来组建产品，导演类一般不与产品类发生依赖关系，与导演类直接交互的是建造者类。一般来说，导演类被用来封装程序中易变的部分。

代码实现

```
class Product {
    private String name;
    private String type;
    public void showProduct(){
        System.out.println("名称："+name);
        System.out.println("型号："+type);
```

```
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setType(String type) {
        this.type = type;
    }
}

abstract class Builder {
    public abstract void setPart(String arg1, String arg2);
    public abstract Product getProduct();
}
class ConcreteBuilder extends Builder {
    private Product product = new Product();

    public Product getProduct() {
        return product;
    }

    public void setPart(String arg1, String arg2) {
        product.setName(arg1);
        product.setType(arg2);
    }
}

public class Director {
    private Builder builder = new ConcreteBuilder();
    public Product getAProduct(){
        builder.setPart("宝马汽车","X7");
        return builder.getProduct();
    }
    public Product getBProduct(){
        builder.setPart("奥迪汽车","Q5");
        return builder.getProduct();
    }
}
public class Client {
    public static void main(String[] args){
        Director director = new Director();
        Product product1 = director.getAProduct();
        product1.showProduct();

        Product product2 = director.getBProduct();
        product2.showProduct();
```

```
    }
  }
```

建造者模式的优点

首先，建造者模式的封装性很好。使用建造者模式可以有效的封装变化，在使用建造者模式的场景中，一般产品类和建造者类是比较稳定的，因此，将主要的业务逻辑封装在导演类中对整体而言可以取得比较好的稳定性。

其次，建造者模式很容易进行扩展。如果有新的需求，通过实现一个新的建造者类就可以完成，基本上不用修改之前已经测试通过的代码，因此也就不会对原有功能引入风险。

建造者模式与工厂模式的区别

我们可以看到，建造者模式与工厂模式是极为相似的，总体上，建造者模式仅仅只比工厂模式多了一个"导演类"的角色。在建造者模式的类图中，假如把这个导演类看做是最终调用的客户端，那么图中剩余的部分就可以看作是一个简单的工厂模式了。

与工厂模式相比，建造者模式一般用来创建**更为复杂的对象**，因为对象的创建过程更为复杂，因此将对象的创建过程独立出来组成一个新的类——导演类。也就是说，工厂模式是将对象的全部创建过程封装在工厂类中，由工厂类向客户端提供最终的产品；而建造者模式中，建造者类一般只提供产品类中各个组件的建造，而将具体建造过程交付给导演类。由导演类负责将各个组件按照特定的规则组建为产品，然后将组建好的产品交付给客户端。

总结

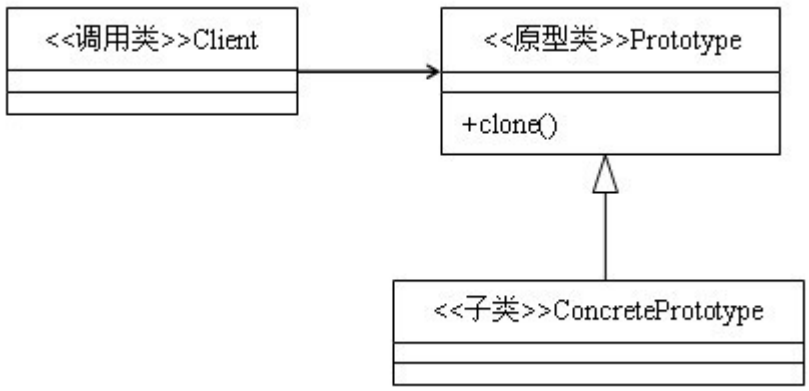
建造者模式与工厂模式类似，他们都是建造者模式，适用的场景也很相似。一般来说，如果产品的建造**很复杂**，那么请用工厂模式；如果产品的建造**更复杂**，那么请用建造者模式。

原型模式

定义：用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。

类型：创建类模式

类图：



图片 1.6 builder-pattern

原型模式主要用于对象的复制，它的核心是就是类图中的原型类Prototype。Prototype类需要具备以下两个条件：

- 实现Cloneable接口。在java语言有一个Cloneable接口，它的作用只有一个，就是在运行时通知虚拟机可以安全地在实现了此接口的类上使用clone方法。在java虚拟机中，只有实现了这个接口的类才可以被拷贝，否则在运行时会抛出CloneNotSupportedException异常。
- 重写Object类中的clone方法。Java中，所有类的父类都是Object类，Object类中有一个clone方法，作用是返回对象的一个拷贝，但是其作用域protected类型的，一般的类无法调用，因此，Prototype类需要将clone方法的作用域修改为public类型。

原型模式是一种比较简单的模式，也非常容易理解，实现一个接口，重写一个方法即完成了原型模式。在实际应用中，原型模式很少单独出现。经常与其他模式混用，他的原型类Prototype也常用抽象类来替代。

实现代码：

```
class Prototype implements Cloneable {
    public Prototype clone(){
        Prototype prototype = null;
        try{
            prototype = (Prototype)super.clone();
        }catch(CloneNotSupportedException e){
```

```
            e.printStackTrace();
        }
        return prototype;
    }
}

class ConcretePrototype extends Prototype{
    public void show(){
        System.out.println("原型模式实现类");
    }
}

public class Client {
    public static void main(String[] args){
        ConcretePrototype cp = new ConcretePrototype();
        for(int i=0; i< 10; i++){
            ConcretePrototype clonecp = (ConcretePrototype)cp.clone();
            clonecp.show();
        }
    }
}
```

原型模式的优点及适用场景

使用原型模式创建对象比直接new一个对象在性能上要好的多，因为Object类的clone方法是一个本地方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显。

使用原型模式的另一个好处是简化对象的创建，使得创建对象就像我们在编辑文档时的复制粘贴一样简单。

因为以上优点，所以在需要重复地创建相似对象时可以考虑使用原型模式。比如需要在一个循环体内创建对象，假如对象创建过程比较复杂或者循环次数很多的话，使用原型模式不但可以简化创建过程，而且可以使系统的整体性能提高很多。

原型模式的注意事项

- 使用原型模式复制对象不会调用类的构造方法。因为对象的复制是通过调用Object类的clone方法来完成的，它直接在内存中复制数据，因此不会调用到类的构造方法。不但构造方法中的代码不会执行，甚至连访问权限都对原型模式无效。还记得单例模式吗？单例模式中，只要将构造方法的访问权限设置为private型，就可以实现单例。但是clone方法直接无视构造方法的权限，所以，单例模式与原型模式是冲突的，在使用时要特别注意。
- 深拷贝与浅拷贝。Object类的clone方法只会拷贝对象中的基本的数据类型，对于数组、容器对象、引用对象等都不会拷贝，这就是浅拷贝。如果要实现深拷贝，必须将原型模式中的数组、容器对象、引用对象等另行拷贝。例如：

```
public class Prototype implements Cloneable {
    private ArrayList list = new ArrayList();
    public Prototype clone(){
        Prototype prototype = null;
        try{
            prototype = (Prototype)super.clone();
            prototype.list = (ArrayList) this.list.clone();
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        return prototype;
    }
}
```

由于ArrayList不是基本类型，所以成员变量list，不会被拷贝，需要我们自己实现深拷贝，幸运的是java提供的大部分的容器类都实现了Cloneable接口。所以实现深拷贝并不是特别困难。

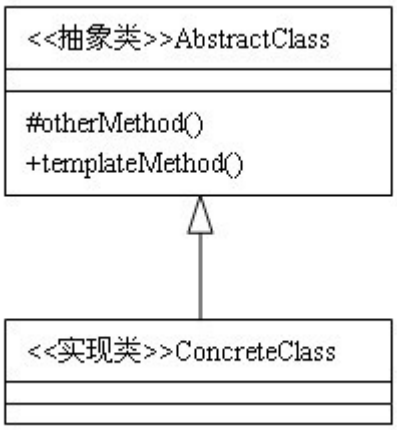
PS：深拷贝与浅拷贝问题中，会发生深拷贝的有java中的8中基本类型以及他们的封装类型，另外还有String类型。其余的都是浅拷贝。

模版方法模式

定义：定义一个操作中算法的框架，而将一些步骤延迟到子类中，使得子类可以不改变算法的结构即可重定义该算法中的某些特定步骤。

类型：行为类模式

类图：



图片 1.7 template-method-pattern

事实上，模版方法是编程中一个经常用到的模式。先来看一个例子，某日，程序员A拿到一个任务：给定一个整数数组，把数组中的数由小到大排序，然后把排序之后的结果打印出来。经过分析之后，这个任务大体上可分为两部分，排序和打印，打印功能好实现，排序就有点麻烦了。但是A有办法，先把打印功能完成，排序功能另找人做。

```
abstract class AbstractSort {

    /**
     * 将数组array由小到大排序
     * @param array
     */
    protected abstract void sort(int[] array);

    public void showSortResult(int[] array){
        this.sort(array);
        System.out.print("排序结果：");
        for (int i = 0; i < array.length; i++){
            System.out.printf("%3s", array[i]);
        }
```

```
    }
  }
```

写完后，A找到刚毕业入职不久的同事B说：有个任务，主要逻辑我已经写好了，你把剩下的逻辑实现一下吧。于是把AbstractSort类给B，让B写实现。B拿过来一看，太简单了，10分钟搞定，代码如下：

```
class ConcreteSort extends AbstractSort {
  @Override
  protected void sort(int[] array){
    for(int i=0; i&lt;array.length-1; i++){
      selectSort(array, i);
    }
  }

  private void selectSort(int[] array, int index) {
    int MinValue = 32767; // 最小值变量
    int indexMin = 0; // 最小值索引变量
    int Temp; // 暂存变量
    for (int i = index; i &lt; array.length; i++) {
      if (array[i] &lt; MinValue){ // 找到最小值
        MinValue = array[i]; // 储存最小值
        indexMin = i;
      }
    }
    Temp = array[index]; // 交换两数值
    array[index] = array[indexMin];
    array[indexMin] = Temp;
  }
}
```

写好后交给A，A拿来一运行：

```
  <pre class="java" name="code">public class Client {
    public static int[] a = { 10, 32, 1, 9, 5, 7, 12, 0, 4, 3 }; // 预设数据数组
    public static void main(String[] args){
      AbstractSort s = new ConcreteSort();
      s.showSortResult(a);
    }
  }
```

运行结果：

排序结果： 0 1 3 4 5 7 9 10 12 32

运行正常。行了，任务完成。没错，这就是模版方法模式。大部分刚步入职场的毕业生应该都有类似B的经历。一个复杂的任务，由公司中的牛人们将主要的逻辑写好，然后把那些看上去比较简单的方法写成抽象的，交给其他

的同事去开发。这种分工方式在编程人员水平层次比较明显的公司中经常用到。比如一个项目组，有架构师，高级工程师，初级工程师，则一般由架构师使用大量的接口、抽象类将整个系统的逻辑串起来，实现的编码则根据难度的不同分别交给高级工程师和初级工程师来完成。怎么样，是不是用到过模版方法模式?

模版方法模式的结构

模版方法模式由一个抽象类和一个（或一组）实现类通过继承结构组成，抽象类中的方法分为三种：

- **抽象方法**：父类中只声明但不加以实现，而是定义好规范，然后由它的子类去实现。
- **模版方法**：由抽象类声明并加以实现。一般来说，模版方法调用抽象方法来完成主要的逻辑功能，并且，模版方法大多会定义为final类型，指明主要的逻辑功能在子类中不能被重写。
- **钩子方法**：由抽象类声明并加以实现。但是子类可以去扩展，子类可以通过扩展钩子方法来影响模版方法的逻辑。
- 抽象类的任务是搭建逻辑的框架，通常由经验丰富的人员编写，因为抽象类的好坏直接决定了程序是否稳定性。

  实现类用来实现细节。抽象类中的模版方法正是通过实现类扩展的方法来完成业务逻辑。只要实现类中的扩展方法通过了单元测试，在模版方法正确的前提下，整体功能一般不会出现大的错误。

模版方法的优点及适用场景

容易扩展。一般来说，抽象类中的模版方法是不易反生改变的部分，而抽象方法是容易反生变化的部分，因此通过增加实现类一般可以很容易实现功能的扩展，符合开闭原则。

便于维护。对于模版方法模式来说，正是由于他们的主要逻辑相同，才使用了模版方法，假如不使用模版方法，任由这些相同的代码散乱的分布在不同的类中，维护起来是非常不方便的。

比较灵活。因为有钩子方法，因此，子类的实现也可以影响父类中主逻辑的运行。但是，在灵活的同时，由于子类影响到了父类，违反了里氏替换原则，也会给程序带来风险。这就对抽象类的设计有了更高的要求。

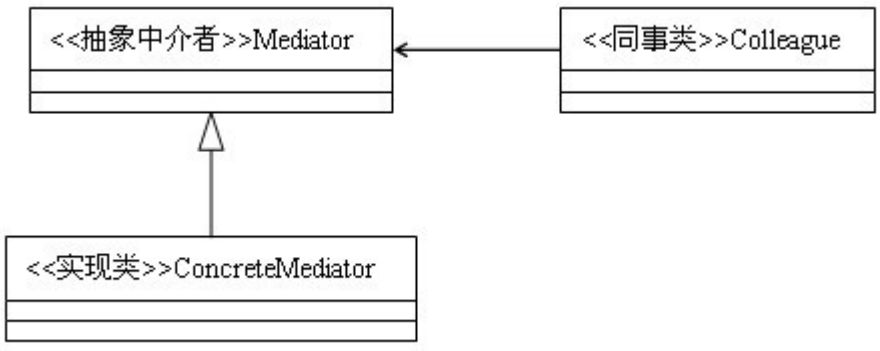
在多个子类拥有相同的方法，并且这些方法逻辑相同时，可以考虑使用模版方法模式。在程序的主框架相同，细节不同的场合下，也比较适合使用这种模式。

中介者模式

定义：用一个中介者对象封装一系列的对象交互，中介者使各对象不需要显示地相互作用，从而使耦合松散，而且可以独立地改变它们之间的交互。

类型：行为类模式

类图：



图片 1.8 mediator-pattern

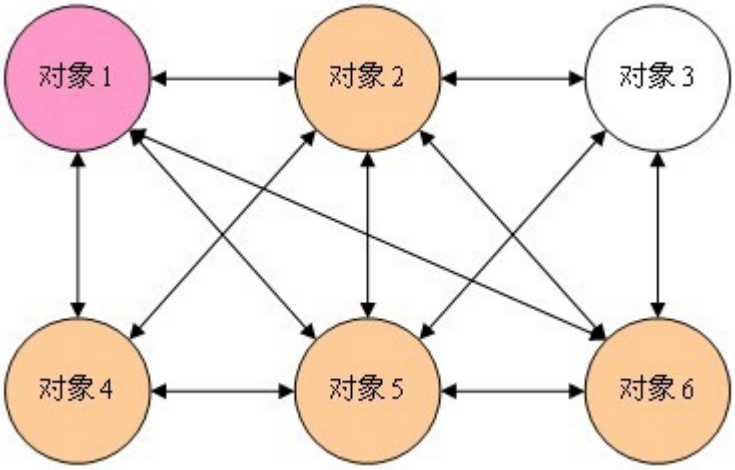
中介者模式的结构

中介者模式又称为调停者模式，从类图中看，共分为3部分：

- 抽象中介者：定义好同事类对象到中介者对象的接口，用于各个同事类之间的通信。一般包括一个或几个抽象的事件方法，并由子类去实现。
- 中介者实现类：从抽象中介者继承而来，实现抽象中介者中定义的事件方法。从一个同事类接收消息，然后通过消息影响其他同时类。
- 同事类：如果一个对象会影响其他的对象，同时也会被其他对象影响，那么这两个对象称为同事类。在类图中，同事类只有一个，这其实是现实的省略，在实际应用中，同事类一般由多个组成，他们之间相互影响，相互依赖。同事类越多，关系越复杂。并且，同事类也可以表现为继承了同一个抽象类的一组实现组成。在中介者模式中，同事类之间必须通过中介者才能进行消息传递。

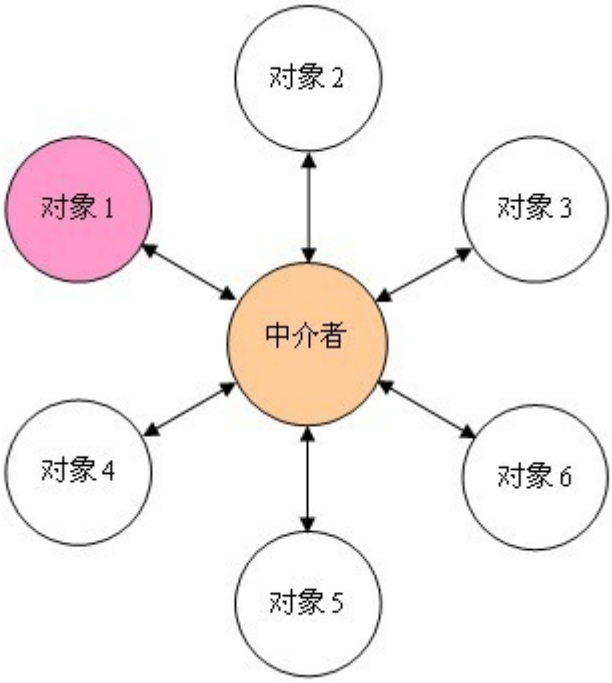
为什么要使用中介者模式

一般来说，同事类之间的关系是比较复杂的，多个同事类之间互相关联时，他们之间的关系会呈现为复杂的网状结构，这是一种过度耦合的架构，即不利于类的复用，也不稳定。例如在下图中，有六个同事类对象，假如对象1发生变化，那么将会有4个对象受到影响。如果对象2发生变化，那么将会有5个对象受到影响。也就是说，同事类之间直接关联的设计是不好的。



图片 1.9 mediator-pattern

如果引入中介者模式，那么同事类之间的关系将变为星型结构，从图中可以看到，任何一个类的变动，只会影响的类本身，以及中介者，这样就减小了系统的耦合。一个好的设计，必定不会把所有的对象关系处理逻辑封装在本类中，而是使用一个专门的类来管理那些不属于自己的行为。



图片 1.10 mediator-pattern

我们使用一个例子来说明一下什么是同事类：有两个类A和B，类中各有一个数字，并且要保证类B中的数字永远是类A中数字的100倍。也就是说，当修改类A的数时，将这个数字乘以100赋给类B，而修改类B时，要将数除以100赋给类A。类A类B互相影响，就称为同事类。代码如下：

```
abstract class AbstractColleague {
    protected int number;
```

```
    public int getNumber() {
        return number;
    }

    public void setNumber(int number){
        this.number = number;
    }
    //抽象方法，修改数字时同时修改关联对象
    public abstract void setNumber(int number, AbstractColleague coll);
}

class ColleagueA extends AbstractColleague{
    public void setNumber(int number, AbstractColleague coll) {
        this.number = number;
        coll.setNumber(number*100);
    }
}

class ColleagueB extends AbstractColleague{

    public void setNumber(int number, AbstractColleague coll) {
        this.number = number;
        coll.setNumber(number/100);
    }
}

public class Client {
    public static void main(String[] args){

        AbstractColleague collA = new ColleagueA();
        AbstractColleague collB = new ColleagueB();

        System.out.println("==========设置A影响B==========");
        collA.setNumber(1288, collB);
        System.out.println("collA的number值："+collA.getNumber());
        System.out.println("collB的number值："+collB.getNumber());

        System.out.println("==========设置B影响A==========");
        collB.setNumber(87635, collA);
        System.out.println("collB的number值："+collB.getNumber());
        System.out.println("collA的number值："+collA.getNumber());
    }
}
```

上面的代码中，类A类B通过直接的关联发生关系，假如我们要使用中介者模式，类A类B之间则不可以直接关联，他们之间必须要通过一个中介者来达到关联的目的。

```
abstract class AbstractColleague {
    protected int number;

    public int getNumber() {
        return number;
    }

    public void setNumber(int number){
        this.number = number;
    }
    //注意这里的参数不再是同事类，而是一个中介者
    public abstract void setNumber(int number, AbstractMediator am);
}

class ColleagueA extends AbstractColleague{

    public void setNumber(int number, AbstractMediator am) {
        this.number = number;
        am.AaffectB();
    }
}

class ColleagueB extends AbstractColleague{

    @Override
    public void setNumber(int number, AbstractMediator am) {
        this.number = number;
        am.BaffectA();
    }
}

abstract class AbstractMediator {
    protected AbstractColleague A;
    protected AbstractColleague B;

    public AbstractMediator(AbstractColleague a, AbstractColleague b) {
        A = a;
        B = b;
    }

    public abstract void AaffectB();
```

```
    public abstract void BaffectA();

}
class Mediator extends AbstractMediator {

    public Mediator(AbstractColleague a, AbstractColleague b) {
        super(a, b);
    }

    //处理A对B的影响
    public void AaffectB() {
        int number = A.getNumber();
        B.setNumber(number*100);
    }

    //处理B对A的影响
    public void BaffectA() {
        int number = B.getNumber();
        A.setNumber(number/100);
    }
}

public class Client {
    public static void main(String[] args){
        AbstractColleague collA = new ColleagueA();
        AbstractColleague collB = new ColleagueB();

        AbstractMediator am = new Mediator(collA, collB);

        System.out.println("==========通过设置A影响B==========");
        collA.setNumber(1000, am);
        System.out.println("collA的number值为："+collA.getNumber());
        System.out.println("collB的number值为A的10倍："+collB.getNumber());

        System.out.println("==========通过设置B影响A==========");
        collB.setNumber(1000, am);
        System.out.println("collB的number值为："+collB.getNumber());
        System.out.println("collA的number值为B的0.1倍："+collA.getNumber());

    }
}
```

虽然代码比较长，但是还是比较容易理解的，其实就是把原来处理对象关系的代码重新封装到一个中介类中，通过这个中介类来处理对象间的关系。

中介者模式的优点

- 适当地使用中介者模式可以避免同事类之间的过度耦合，使得各同事类之间可以相对独立地使用。
- 使用中介者模式可以将对象间一对多的关联转变为一对一的关联，使对象间的关系易于理解和维护。
- 使用中介者模式可以将对象的行为和协作进行抽象，能够比较灵活的处理对象间的相互作用。

适用场景

在面向对象编程中，一个类必然会与其他的类发生依赖关系，完全独立的类是没有意义的。一个类同时依赖多个类的情况也相当普遍，既然存在这样的情况，说明，一对多的依赖关系有它的合理性，适当的使用中介者模式可以使原本凌乱的对象关系清晰，但是如果滥用，则可能会带来反的效果。一般来说，只有对于那种同事类之间是网状结构的关系，才会考虑使用中介者模式。可以将网状结构变为星状结构，使同事类之间的关系变的清晰一些。

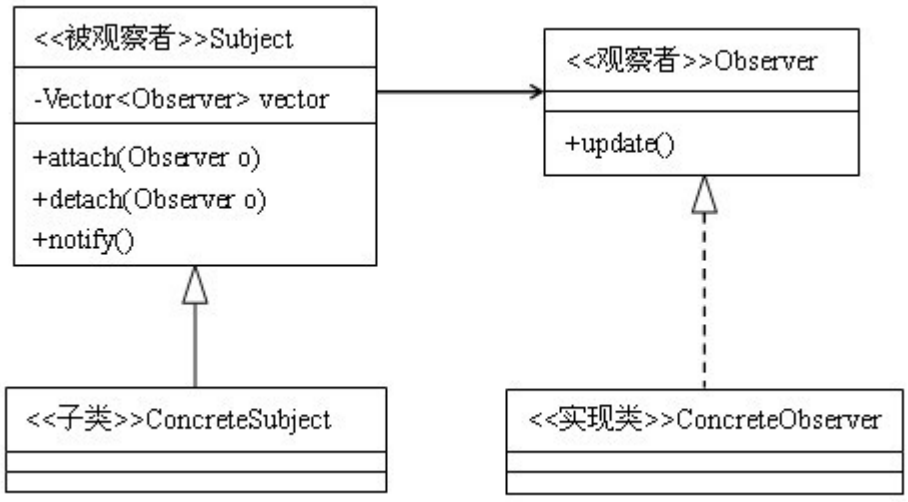
中介者模式是一种比较常用的模式，也是一种比较容易被滥用的模式。对于大多数的情况，同事类之间的关系不会复杂到混乱不堪的网状结构，因此，大多数情况下，将对象间的依赖关系封装的同事类内部就可以的，没有必要非引入中介者模式。滥用中介者模式，只会让事情变的更复杂。

观察者模式

定义：定义对象间一种一对多的依赖关系，使得当每一个对象改变状态，则所有依赖于它的对象都会得到通知并自动更新。

类型：行为类模式

类图：



图片 1.11 observer-pattern

在软件系统中经常会有这样的需求：如果一个对象的状态发生改变，某些与它相关的对象也要随之做出相应的变化。比如，我们要设计一个右键菜单的功能，只要在软件的有效区域内点击鼠标右键，就会弹出一个菜单；再比如，我们要设计一个自动部署的功能，就像eclipse开发时，只要修改了文件，eclipse就会自动将修改的文件部署到服务器中。这两个功能有一个相似的地方，那就是一个对象要时刻监听着另一个对象，只要它的状态一发生改变，自己随之要做出相应的行动。其实，能够实现这一点的方案很多，但是，无疑使用观察者模式是一个主流的选择。

观察者模式的结构

在最基础的观察者模式中，包括以下四个角色：

- **被观察者**：从类图中可以看到，类中有一个用来存放观察者对象的Vector容器（之所以使用Vector而不使用List，是因为多线程操作时，Vector在是安全的，而List则是不安全的），这个Vector容器是被观察者类的核心，另外还有三个方法：attach方法是向这个容器中添加观察者对象；detach方法是从容器中移除观察者对象；notify方法是依次调用观察者对象的对应方法。这个角色可以是接口，也可以是抽象类或者具体的类，因为很多情况下会与其他的模式混用，所以使用抽象类的情况比较多。

- **观察者：**观察者角色一般是一个接口，它只有一个update方法，在被观察者状态发生变化时，这个方法就会被触发调用。
- **具体的被观察者：**使用这个角色是为了便于扩展，可以在此角色中定义具体的业务逻辑。
- **具体的观察者：**观察者接口的具体实现，在这个角色中，将定义被观察者对象状态发生变化时所要处理的逻辑。

观察者模式代码实现

```
abstract class Subject {
    private Vector obs = new Vector();

    public void addObserver(Observer obs){
        this.obs.add(obs);
    }
    public void delObserver(Observer obs){
        this.obs.remove(obs);
    }
    protected void notifyObserver(){
        for(Observer o: obs){
            o.update();
        }
    }
    public abstract void doSomething();
}

class ConcreteSubject extends Subject {
    public void doSomething(){
        System.out.println("被观察者事件反生");
        this.notifyObserver();
    }
}
interface Observer {
    public void update();
}
class ConcreteObserver1 implements Observer {
    public void update() {
        System.out.println("观察者1收到信息，并进行处理。");
    }
}
class ConcreteObserver2 implements Observer {
    public void update() {
        System.out.println("观察者2收到信息，并进行处理。");
    }
```

```
}

public class Client {
    public static void main(String[] args){
        Subject sub = new ConcreteSubject();
        sub.addObserver(new ConcreteObserver1()); //添加观察者1
        sub.addObserver(new ConcreteObserver2()); //添加观察者2
        sub.doSomething();
    }
}
```

运行结果

被观察者事件反生

观察者1收到信息，并进行处理。

观察者2收到信息，并进行处理。

通过运行结果可以看到，我们只调用了Subject的方法，但同时两个观察者的相关方法都被同时调用了。仔细看一下代码，其实很简单，无非就是在Subject类中关联一下Observer类，并且在doSomething方法中遍历一下Observer的update方法就行了。

观察者模式的优点

观察者与被观察者之间是属于轻度的关联关系，并且是抽象耦合的，这样，对于两者来说都比较容易进行扩展。

观察者模式是一种常用的触发机制，它形成一条触发链，依次对各个观察者的方法进行处理。但同时，这也算是观察者模式一个缺点，由于是链式触发，当观察者比较多的时候，性能问题是比较令人担忧的。并且，在链式结构中，比较容易出现循环引用的错误，造成系统假死。

总结

java语言中，有一个接口Observer，以及它的实现类Observable，对观察者角色常进行了实现。我们可以在jdk的api文档具体查看这两个类的使用方法。

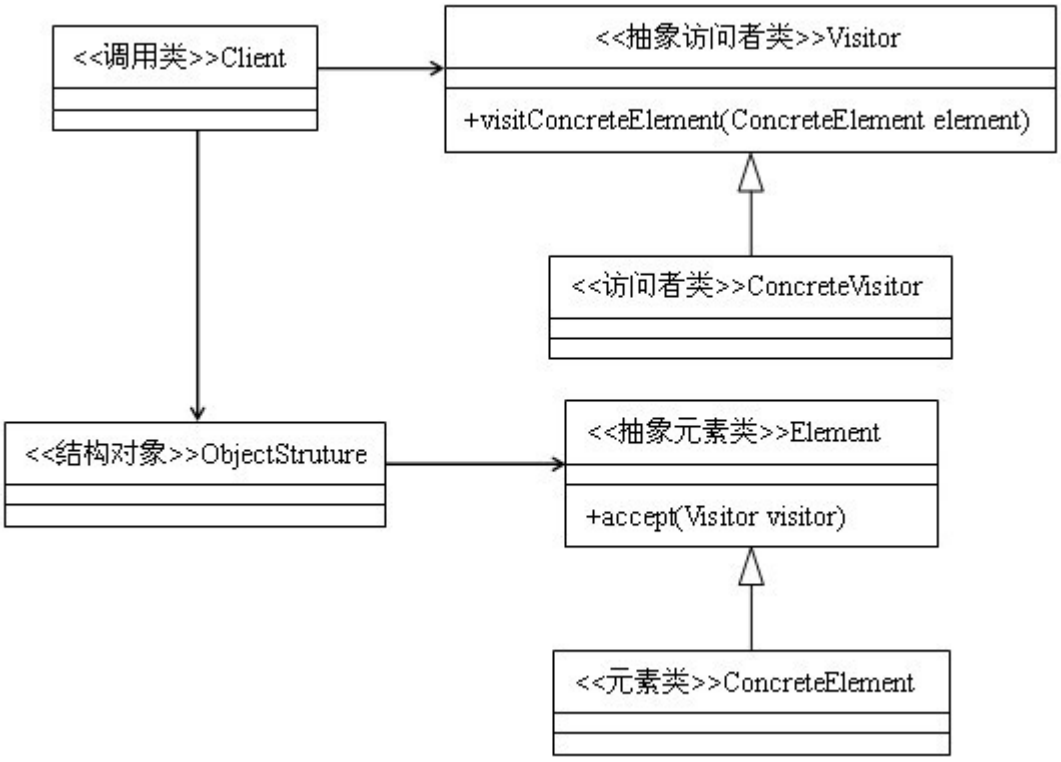
做过VC++、javascript DOM或者AWT开发的朋友都对它们的事件处理感到神奇，了解了观察者模式，就对事件处理机制的原理有了一定的了解了。如果要设计一个事件触发处理机制的功能，使用观察者模式是一个不错的选择，AWT中的事件处理DEM（委派事件模型Delegation Event Model）就是使用观察者模式实现的。

访问者模式

定义：封装某些作用于某种数据结构中各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。

类型：行为类模式

类图：



图片 1.12 visitor-pattern

访问者模式可能是行为类模式中最复杂的一种模式了，但是这不能成为我们不去掌握它的理由。我们首先来看一个简单的例子，代码如下：

```
class A {
    public void method1(){
        System.out.println("我是A");
    }

    public void method2(B b){
        b.showA(this);
    }
```

```
}

class B {
    public void showA(A a){
        a.method1();
    }
}
```

我们主要来看一下在类A中，方法method1和方法method2的区别在哪里，方法method1很简单，就是打印出一句"我是A"；方法method2稍微复杂一点，使用类B作为参数，并调用类B的showA方法。再来看一下类B的showA方法，showA方法使用类A作为参数，然后调用类A的method1方法，可以看到，method2方法绕来绕去，无非就是调用了一下自己的method1方法而已，它的运行结果应该也是"我是A"，分析完之后，我们来运行一下这两个方法，并看一下运行结果：

```
public class Test {
    public static void main(String[] args){
        A a = new A();
        a.method1();
        a.method2(new B());
    }
}
```

运行结果为：

我是A
我是A

看懂了这个例子，就理解了访问者模式的90%，在例子中，对于类A来说，类B就是一个访问者。但是这个例子并不是访问者模式的全部，虽然直观，但是它的可扩展性比较差，下面我们就来说一下访问者模式的通用实现，通过类图可以看到，在访问者模式中，主要包括下面几个角色：

- ** 抽象访问者：**抽象类或者接口，声明访问者可以访问哪些元素，具体到程序中就是visit方法中的参数定义哪些对象是可以被访问的。
- **访问者：**实现抽象访问者所声明的方法，它影响到访问者访问到一个类后该干什么，要做什么事情。
- **抽象元素类：**接口或者抽象类，声明接受哪一类访问者访问，程序上是通过accept方法中的参数来定义的。抽象元素一般有两类方法，一部分是本身的业务逻辑，另外就是允许接收哪类访问者来访问。
- **元素类：**实现抽象元素类所声明的accept方法，通常都是visitor.visit(this)，基本上已经形成一种定式了。
- **结构对象：**一个元素的容器，一般包含一个容纳多个不同类、不同接口的容器，如List、Set、Map等，在项目中一般很少抽象出这个角色。

  访问者模式的通用代码实现

```
abstract class Element {
    public abstract void accept(IVisitor visitor);
    public abstract void doSomething();
}

interface IVisitor {
    public void visit(ConcreteElement1 el1);
    public void visit(ConcreteElement2 el2);
}

class ConcreteElement1 extends Element {
    public void doSomething(){
        System.out.println("这是元素1");
    }

    public void accept(IVisitor visitor) {
        visitor.visit(this);
    }
}

class ConcreteElement2 extends Element {
    public void doSomething(){
        System.out.println("这是元素2");
    }

    public void accept(IVisitor visitor) {
        visitor.visit(this);
    }
}
class Visitor implements IVisitor {

    public void visit(ConcreteElement1 el1) {
        el1.doSomething();
    }

    public void visit(ConcreteElement2 el2) {
        el2.doSomething();
    }
}

class ObjectStruture {
    public static List getList(){
        List list = new ArrayList();
        Random ran = new Random();
        for(int i=0; i<10; i++){
```

```
            int a = ran.nextInt(100);
            if(a>50){
                list.add(new ConcreteElement1());
            }else{
                list.add(new ConcreteElement2());
            }
        }
        return list;
    }
}

public class Client {
    public static void main(String[] args){
        List list = ObjectStruture.getList();
        for(Element e: list){
            e.accept(new Visitor());
        }
    }
}
```

访问者模式的优点

- **符合单一职责原则：**凡是适用访问者模式的场景中，元素类中需要封装在访问者中的操作必定是与元素类本身关系不大且是易变的操作，使用访问者模式一方面符合单一职责原则，另一方面，因为被封装的操作通常来说都是易变的，所以当发生变化时，就可以在不改变元素类本身的前提下，实现对变化部分的扩展。
- **扩展性良好：**元素类可以通过接受不同的访问者来实现对不同操作的扩展。

** 访问者模式的适用场景**

假如一个对象中存在着一些与本对象不相干（或者关系较弱）的操作，为了避免这些操作污染这个对象，则可以使用访问者模式来把这些操作封装到访问者中去。

假如一组对象中，存在着相似的操作，为了避免出现大量重复的代码，也可以将这些重复的操作封装到访问者中去。

但是，访问者模式并不是那么完美，它也有着致命的缺陷：增加新的元素类比较困难。通过访问者模式的代码可以看到，在访问者类中，每一个元素类都有它对应的处理方法，也就是说，每增加一个元素类都需要修改访问者类（也包括访问者类的子类或者实现类），修改起来相当麻烦。也就是说，在元素类数目不确定的情况下，应该慎用访问者模式。所以，访问者模式比较适用于对已有功能的重构，比如说，一个项目的基本功能已经确定下来，元素类的数据已经基本确定下来不会变了，会变的只是这些元素内的相关操作，这时候，我们可以使用访问者模式对原有的代码进行重构一遍，这样一来，就可以在不修改各个元素类的情况下，对原有功能进行修改。

总结

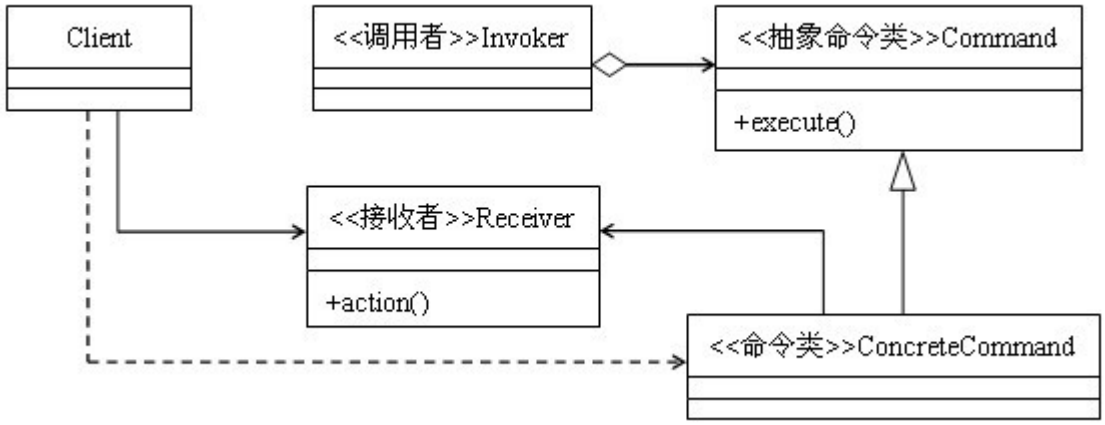
正如《 设计模式 》的作者GoF对访问者模式的描述：大多数情况下，你并需要使用访问者模式，但是当你一旦需要使用它时，那你就是真的需要它了。当然这只是针对真正的大牛而言。在现实情况下（ 至少是我所处的环境当中 ），很多人往往沉迷于设计模式，他们使用一种设计模式时，从来不去认真考虑所使用的模式是否适合这种场景，而往往只是想展示一下自己对面向对象设计的驾驭能力。编程时有这种心理，往往会发生滥用设计模式的情况。所以，在学习设计模式时，一定要理解模式的适用性。必须做到使用一种模式是因为了解它的优点，不使用一种模式是因为了解它的弊端；而不是使用一种模式是因为不了解它的弊端，不使用一种模式是因为不了解它的优点。

命令模式

定义：将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

类型：行为类模式

类图：



图片 1.13 command-pattern

命令模式的结构

顾名思义，命令模式就是对命令的封装，首先来看一下命令模式类图中的基本结构：

- **Command类**：是一个抽象类，类中对需要执行的命令进行声明，一般来说要对外公布一个execute方法用来执行命令。
- **ConcreteCommand类**：Command类的实现类，对抽象类中声明的方法进行实现。
- **Client类**：最终的客户端调用类。

以上三个类的作用应该是比较好理解的，下面我们重点说一下Invoker类和Recevier类。

- **Invoker类**：调用者，负责调用命令。
- **Receiver类**：接收者，负责接收命令并且执行命令。

所谓对命令的封装，说白了，无非就是把一系列的操作写到一个方法中，然后供客户端调用就行了，反映到类图上，只需要一个ConcreteCommand类和Client类就可以完成对命令的封装，即使再进一步，为了增加灵活性，可以再增加一个Command类进行适当地抽象，这个调用者和接收者到底是什么作用呢?

其实大家可以换一个角度去想：假如仅仅是简单地把一些操作封装起来作为一条命令供别人调用，怎么能称为一种模式呢？命令模式作为一种行为类模式，首先要做到低耦合，耦合度低了才能提高灵活性，而加入调用者和接收者两个角色的目的也正是为此。命令模式的通用代码如下：

```
class Invoker {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void action(){
        this.command.execute();
    }
}

abstract class Command {
    public abstract void execute();
}

class ConcreteCommand extends Command {
    private Receiver receiver;
    public ConcreteCommand(Receiver receiver){
        this.receiver = receiver;
    }
    public void execute() {
        this.receiver.doSomething();
    }
}

class Receiver {
    public void doSomething(){
        System.out.println("接受者-业务逻辑处理");
    }
}

public class Client {
    public static void main(String[] args){
        Receiver receiver = new Receiver();
        Command command = new ConcreteCommand(receiver);
        //客户端直接执行具体命令方式（此方式与类图相符）
        command.execute();

        //客户端通过调用者来执行命令
        Invoker invoker = new Invoker();
        invoker.setCommand(command);
```

```
        invoker.action();
    }
}
```

通过代码我们可以看到，当我们调用时，执行的时序首先是调用者类，然后是命令类，最后是接收者类。也就是说一条命令的执行被分成了三步，它的耦合度要比把所有的操作都封装到一个类中要低的多，而这也正是命令模式的精髓所在：把命令的调用者与执行者分开，使双方不必关心对方是如何操作的。

命令模式的优缺点

首先，命令模式的封装性很好：每个命令都被封装起来，对于客户端来说，需要什么功能就去调用相应的命令，而无需知道命令具体是怎么执行的。比如有一组文件操作的命令：新建文件、复制文件、删除文件。如果把这三个操作都封装成一个命令类，客户端只需要知道有这三个命令类即可，至于命令类中封装好的逻辑，客户端则无需知道。

其次，命令模式的扩展性很好，在命令模式中，在接收者类中一般会对操作进行最基本的封装，命令类则通过对这些基本的操作进行二次封装，当增加新命令的时候，对命令类的编写一般不是从零开始的，有大量的接收者类可供调用，也有大量的命令类可供调用，代码的复用性很好。比如，文件的操作中，我们需要增加一个剪切文件的命令，则只需要把复制文件和删除文件这两个命令组合一下就行了，非常方便。

最后说一下命令模式的缺点，那就是命令如果很多，开发起来就要头疼了。特别是很多简单的命令，实现起来就几行代码的事，而使用命令模式的话，不用管命令多简单，都需要写一个命令类来封装。

命令模式的适用场景

对于大多数请求-响应模式的功能，比较适合使用命令模式，正如命令模式定义说的那样，命令模式对实现记录日志、撤销操作等功能比较方便。

** 总结**

对于一个场合到底用不用模式，这对所有的开发人员来说都是一个很纠结的问题。有时候，因为预见到需求上会发生的某些变化，为了系统的灵活性和可扩展性而使用了某种设计模式，但这个预见的需求偏偏没有，相反，没预见到的需求倒是来了不少，导致在修改代码的时候，使用的设计模式反而起了相反的作用，以至于整个项目组怨声载道。这样的例子，我相信每个程序设计者都遇到过。所以，基于敏捷开发的原则，我们在设计程序的时候，如果按照目前的需求，不使用某种模式也能很好地解决，那么我们就不要引入它，因为要引入一种设计模式并不困难，我们大可以在真正需要用到的时候再对系统进行一下，引入这个设计模式。

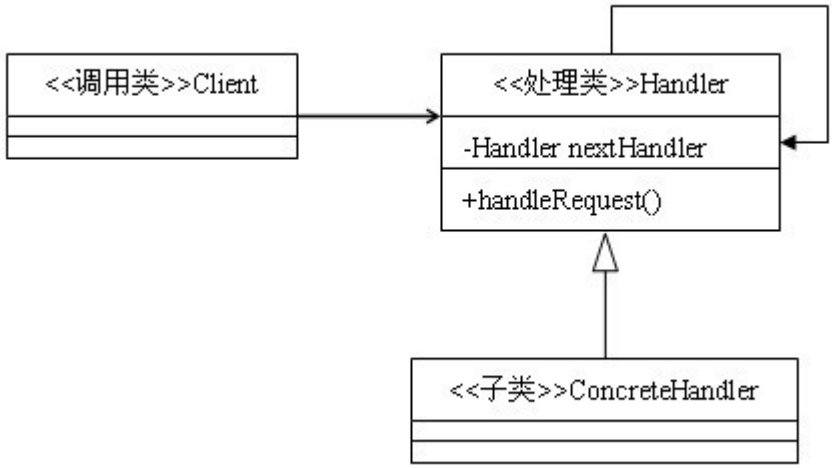
拿命令模式来说吧，我们开发中，请求-响应模式的功能非常常见，一般来说，我们会把对请求的响应操作封装到一个方法中，这个封装的方法可以称之为命令，但不是命令模式。到底要不要把这种设计上升到模式的高度就要另行考虑了，因为，如果使用命令模式，就要引入调用者、接收者两个角色，原本放在一处的逻辑分散到了三个类中，设计时，必须考虑这样的代价是否值得。

责任连模式

定义：使多个对象都有机会处理请求，从而避免了请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。

类型：行为类模式

类图：



图片 1.14 command-pattern

首先来看一段代码：

```
public void test(int i, Request request){
    if(i==1){
        Handler1.response(request);
    }else if(i == 2){
        Handler2.response(request);
    }else if(i == 3){
        Handler3.response(request);
    }else if(i == 4){
        Handler4.response(request);
    }else{
        Handler5.response(request);
    }
}
```

代码的业务逻辑是这样的，方法有两个参数：整数i和一个请求request，根据i的值来决定由谁来处理request，如果i==1，由Handler1来处理，如果i==2，由Handler2来处理，以此类推。在编程中，这种处理业务的方法

非常常见，所有处理请求的类有if…else…条件判断语句连成一条责任链来对请求进行处理，相信大家都经常用到。这种方法的优点是非常直观，简单明了，并且比较容易维护，但是这种方法也存在着几个比较令人头疼的问题：

- **代码臃肿**：实际应用中的判定条件通常不是这么简单地判断是否为1或者是否为2，也许需要复杂的计算，也许需要查询数据库等等，这就会有很多额外的代码，如果判断条件再比较多，那么这个if…else…语句基本上就没法看了。
- **耦合度高**：如果我们想继续添加处理请求的类，那么就要继续添加else if判定条件；另外，这个条件判定的顺序也是写死的，如果想改变顺序，那么也只能修改这个条件语句。

既然缺点我们已经清楚了，就要想办法来解决。这个场景的业务逻辑很简单：如果满足条件1，则由Handler1来处理，不满足则向下传递；如果满足条件2，则由Handler2来处理，不满足则继续向下传递，以此类推，直到条件结束。其实改进的方法也很简单，就是把判定条件的部分放到处理类中，这就是责任连模式的原理。

责任连模式的结构

责任连模式的类图非常简单，它由一个抽象地处理类和它的一组实现类组成：

- **抽象处理类**：抽象处理类中主要包含一个指向下一处理类的成员变量nextHandler和一个处理请求的方法handRequest，handRequest方法的主要主要思想是，如果满足处理的条件，则有本处理类来进行处理，否则由nextHandler来处理。
- **具体处理类**：具体处理类主要是对具体的处理逻辑和处理的适用条件进行实现。

了解了责任连模式的大体思想之后，再看代码就比较好理解了：

```
class Level {
    private int level = 0;
    public Level(int level){
        this.level = level;
    };

    public boolean above(Level level){
        if(this.level >= level.level){
            return true;
        }
        return false;
    }
}

class Request {
    Level level;
    public Request(Level level){
```

```
        this.level = level;
    }

    public Level getLevel(){
        return level;
    }
}

class Response {

}

abstract class Handler {
    private Handler nextHandler;
    public final Response handleRequest(Request request){
        Response response = null;

        if(this.getHandlerLevel().above(request.getLevel())){
            response = this.response(request);
        }else{
            if(this.nextHandler != null){
                this.nextHandler.handleRequest(request);
            }else{
                System.out.println("-----没有合适的处理器-----");
            }
        }
        return response;
    }
    public void setNextHandler(Handler handler){
        this.nextHandler = handler;
    }
    protected abstract Level getHandlerLevel();
    public abstract Response response(Request request);
}

class ConcreteHandler1 extends Handler {
    protected Level getHandlerLevel() {
        return new Level(1);
    }
    public Response response(Request request) {
        System.out.println("-----请求由处理器1进行处理-----");
        return null;
    }
}
```

```
class ConcreteHandler2 extends Handler {
    protected Level getHandlerLevel() {
        return new Level(3);
    }
    public Response response(Request request) {
        System.out.println("-----请求由处理器2进行处理-----");
        return null;
    }
}

class ConcreteHandler3 extends Handler {
    protected Level getHandlerLevel() {
        return new Level(5);
    }
    public Response response(Request request) {
        System.out.println("-----请求由处理器3进行处理-----");
        return null;
    }
}

public class Client {
    public static void main(String[] args){
        Handler handler1 = new ConcreteHandler1();
        Handler handler2 = new ConcreteHandler2();
        Handler handler3 = new ConcreteHandler3();

        handler1.setNextHandler(handler2);
        handler2.setNextHandler(handler3);

        Response response = handler1.handleRequest(new Request(new Level(4)));
    }
}
```

代码中Level类是模拟判定条件；Request，Response分别对应请求和响应；抽象类Handler中主要进行条件的判断，这里模拟一个处理等级，只有处理类的处理等级高于Request的等级才能处理，否则交给下一个处理者处理。在Client类中设置好链的前后执行关系，执行时将请求交给第一个处理类，这就是责任连模式，它完成的功能与前文中的if…else…语句是一样的。

责任链模式的优缺点

责任链模式与if…else…相比，他的耦合性要低一些，因为它把条件判定都分散到了各个处理类中，并且这些处理类的优先处理顺序可以随意设定。责任链模式也有缺点，这与if…else…语句的缺点是一样的，那就是在找到正确的处理类之前，所有的判定条件都要被执行一遍，当责任链比较长时，性能问题比较严重。

责任链模式的适用场景

就像开始的例子那样，假如使用if…else…语句来组织一个责任链时感到力不从心，代码看上去很糟糕时，就可以使用责任链模式来进行重构。

总结

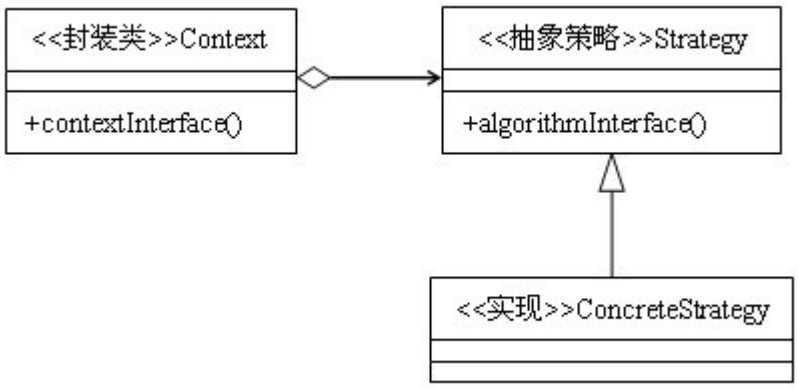
责任链模式其实就是一个灵活版的if…else…语句，它就是将这些判定条件的语句放到了各个处理类中，这样做的优点是比较灵活了，但同样也带来了风险，比如设置处理类前后关系时，一定要特别仔细，搞对处理类前后逻辑的条件判断关系，并且注意不要在链中出现循环引用的问题。

策略模式

定义：定义一组算法，将每个算法都封装起来，并且使他们之间可以互换。

类型：行为类模式

类图：



图片 1.15 strategy-pattern

策略模式是对算法的封装，把一系列的算法分别封装到对应的类中，并且这些类实现相同的接口，相互之间可以替换。在前面说过的行为类模式中，有一种模式也是关注对算法的封装——模版方法模式，对照类图可以看到，策略模式与模版方法模式的区别仅仅是多了一个单独的封装类Context，它与模版方法模式的区别在于：在模版方法模式中，调用算法的主体在抽象的父类中，而在策略模式中，调用算法的主体则是封装到了封装类Context中，抽象策略Strategy一般是一个接口，目的只是为了定义规范，里面一般不包含逻辑。其实，这只是通用实现，而在实际编程中，因为各个具体策略实现类之间难免存在一些相同的逻辑，为了避免重复的代码，我们常常使用抽象类来担任Strategy的角色，在里面封装公共的代码，因此，在很多应用的场景中，在策略模式中一般会看到模版方法模式的影子。

策略模式的结构

- **封装类**：也叫上下文，对策略进行二次封装，目的是避免高层模块对策略的直接调用。
- **抽象策略**：通常情况下为一个接口，当各个实现类中存在着重复的逻辑时，则使用抽象类来封装这部分公共的代码，此时，策略模式看上去更像是模版方法模式。
- **具体策略**：具体策略角色通常由一组封装了算法的类来担任，这些类之间可以根据需要自由替换。

策略模式代码实现

```
interface IStrategy {
    public void doSomething();
}
class ConcreteStrategy1 implements IStrategy {
    public void doSomething() {
        System.out.println("具体策略1");
    }
}
class ConcreteStrategy2 implements IStrategy {
    public void doSomething() {
        System.out.println("具体策略2");
    }
}
class Context {
    private IStrategy strategy;

    public Context(IStrategy strategy){
        this.strategy = strategy;
    }

    public void execute(){
        strategy.doSomething();
    }
}

public class Client {
    public static void main(String[] args){
        Context context;
        System.out.println("-----执行策略1-----");
        context = new Context(new ConcreteStrategy1());
        context.execute();

        System.out.println("-----执行策略2-----");
        context = new Context(new ConcreteStrategy2());
        context.execute();
    }
}
```

策略模式的优缺点

策略模式的主要优点有：

- 策略类之间可以自由切换，由于策略类实现自同一个抽象，所以他们之间可以自由切换。
- 易于扩展，增加一个新的策略对策略模式来说非常容易，基本上可以在不改变原有代码的基础上进行扩展。

- 避免使用多重条件，如果不使用策略模式，对于所有的算法，必须使用条件语句进行连接，通过条件判断来决定使用哪一种算法，在上一篇文章中我们已经提到，使用多重条件判断是非常不容易维护的。

策略模式的缺点主要有两个：

- 维护各个策略类会给开发带来额外开销，可能大家在这方面都有经验：一般来说，策略类的数量超过5个，就比较令人头疼了。
- 必须对客户端（调用者）暴露所有的策略类，因为使用哪种策略是由客户端来决定的，因此，客户端应该知道有什么策略，并且了解各种策略之间的区别，否则，后果很严重。例如，有一个排序算法的策略模式，提供了快速排序、冒泡排序、选择排序这三种算法，客户端在使用这些算法之前，是不是先要明白这三种算法的适用情况？再比如，客户端要使用一个容器，有链表实现的，也有数组实现的，客户端是不是也要明白链表和数组有什么区别？就这一点来说是有悖于迪米特法则的。

适用场景

做面向对象设计的，对策略模式一定很熟悉，因为它实质上就是面向对象中的继承和多态，在看完策略模式的通用代码后，我想，即使之前从来没有听说过策略模式，在开发过程中也一定使用过它吧？至少在在以下两种情况下，大家可以考虑使用策略模式，

- 几个类的主要逻辑相同，只在部分逻辑的算法和行为上稍有区别的情况。
- 有几种相似的行为，或者说算法，客户端需要动态地决定使用哪一种，那么可以使用策略模式，将这些算法封装起来供客户端调用。

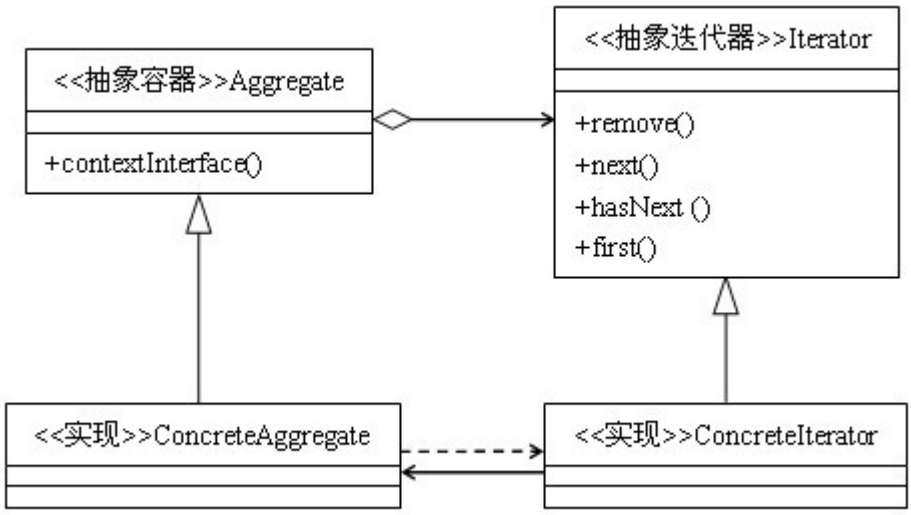
策略模式是一种简单常用的模式，我们在进行开发的时候，会经常有意无意地使用它，一般来说，策略模式不会单独使用，跟模版方法模式、工厂模式等混合使用的情况比较多。

迭代器模式

定义：提供一种方法访问一个容器对象中各个元素，而又不暴露该对象的内部细节。

类型：行为类模式

类图：



图片 1.16 iterator-pattern

如果要问java中使用最多的一种模式，答案不是单例模式，也不是工厂模式，更不是策略模式，而是迭代器模式，先来看一段代码吧：

```
public static void print(Collection coll){
    Iterator it = coll.iterator();
    while(it.hasNext()){
        String str = (String)it.next();
        System.out.println(str);
    }
}
```

这个方法的作用是循环打印一个字符串集合，里面就用到了迭代器模式，java语言已经完整地实现了迭代器模式，Iterator翻译成汉语就是迭代器的意思。提到迭代器，首先它是与集合相关的，集合也叫聚集、容器等，我们可以将集合看成是一个可以包容对象的容器，例如List，Set，Map，甚至数组都可以叫做集合，而迭代器的作用就是把容器中的对象一个一个地遍历出来。

迭代器模式的结构

- 抽象容器：一般是一个接口，提供一个iterator()方法，例如java中的Collection接口，List接口，Set接口等。
- 具体容器：就是抽象容器的具体实现类，比如List接口的有序列表实现ArrayList，List接口的链表实现LinkList，Set接口的哈希列表的实现HashSet等。
- 抽象迭代器：定义遍历元素所需要的方法，一般来说会有这么三个方法：取得第一个元素的方法first()，取得下一个元素的方法next()，判断是否遍历结束的方法isDone()（或者叫hasNext()），移出当前对象的方法remove(),
- 迭代器实现：实现迭代器接口中定义的方法，完成集合的迭代。

代码实现

```
interface Iterator {
    public Object next();
    public boolean hasNext();
}
class ConcreteIterator implements Iterator{
    private List list = new ArrayList();
    private int cursor =0;
    public ConcreteIterator(List list){
        this.list = list;
    }
    public boolean hasNext() {
        if(cursor==list.size()){
            return false;
        }
        return true;
    }
    public Object next() {
        Object obj = null;
        if(this.hasNext()){
            obj = this.list.get(cursor++);
        }
        return obj;
    }
}
interface Aggregate {
    public void add(Object obj);
    public void remove(Object obj);
    public Iterator iterator();
}
class ConcreteAggregate implements Aggregate {
    private List list = new ArrayList();
```

```
    public void add(Object obj) {
        list.add(obj);
    }

    public Iterator iterator() {
        return new ConcreteIterator(list);
    }

    public void remove(Object obj) {
        list.remove(obj);
    }
}
public class Client {
    public static void main(String[] args){
        Aggregate ag = new ConcreteAggregate();
        ag.add("小明");
        ag.add("小红");
        ag.add("小刚");
        Iterator it = ag.iterator();
        while(it.hasNext()){
            String str = (String)it.next();
            System.out.println(str);
        }
    }
}
```

上面的代码中，Aggregate是容器类接口，大家可以想象一下Collection，List，Set等，Aggregate就是他们的简化版，容器类接口中主要有三个方法：添加对象方法add、删除对象方法remove、取得迭代器方法iterator。Iterator是迭代器接口，主要有两个方法：取得迭代对象方法next，判断是否迭代完成方法hasNext，大家可以对比java.util.List和java.util.Iterator两个接口自行思考。

迭代器模式的优缺点

迭代器模式的优点有：

- 简化了遍历方式，对于对象集合的遍历，还是比较麻烦的，对于数组或者有序列表，我们尚可以通过游标来取得，但用户需要在对集合了解很清楚的前提下，自行遍历对象，但是对于hash表来说，用户遍历起来就比较麻烦了。而引入了迭代器方法后，用户用起来就简单的多了。
- 可以提供多种遍历方式，比如说对有序列表，我们可以根据需要提供正序遍历，倒序遍历两种迭代器，用户用起来只需要得到我们实现好的迭代器，就可以方便的对集合进行遍历了。
- 封装性良好，用户只需要得到迭代器就可以遍历，而对于遍历算法则不用去关心。

迭代器模式的缺点：

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

迭代器模式的适用场景

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

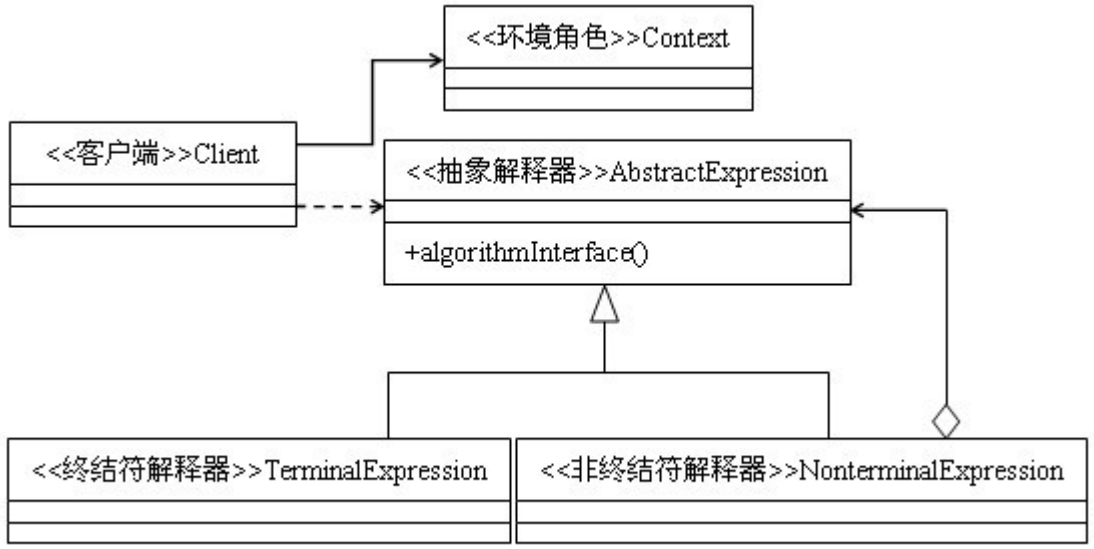
但是，由于容器与迭代器的关系太密切了，所以大多数语言在实现容器的时候都给提供了迭代器，并且这些语言提供的容器和迭代器在绝大多数情况下就可以满足我们的需要，所以现在需要我们自己去实践迭代器模式的场景还是比较少见的，我们只需要使用语言中已有的容器和迭代器就可以了。

解释器模式

定义：给定一种语言，定义他的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中句子。

类型：行为类模式

类图：



图片 1.17 interpreter-pattern

解释器模式是一个比较少用的模式，本人之前也没有用过这个模式。下面我们就来一起看一下解释器模式。

解释器模式的结构

- 抽象解释器：声明一个所有具体表达式都要实现的抽象接口（或者抽象类），接口中主要是一个interpret()方法，称为解释操作。具体解释任务由它的各个实现类来完成，具体的解释器分别由终结符解释器TerminalExpression和非终结符解释器NonterminalExpression完成。
- 终结符表达式：实现与文法中的元素相关联的解释操作，通常一个解释器模式中只有一个终结符表达式，但有多个实例，对应不同的终结符。终结符一半是文法中的运算单元，比如有一个简单的公式R=R1+R2，在里面R1和R2就是终结符，对应的解析R1和R2的解释器就是终结符表达式。
- 非终结符表达式：文法中的每条规则对应于一个非终结符表达式，非终结符表达式一般是文法中的运算符或者其他关键字，比如公式R=R1+R2中，+就是非终结符，解析+的解释器就是一个非终结符表达式。非终结符表达式根据逻辑的复杂程度而增加，原则上每个文法规则都对应一个非终结符表达式。

- 环境角色：这个角色的任务一般是用来存放文法中各个终结符所对应的具体值，比如R=R1+R2，我们给R1赋值100，给R2赋值200。这些信息需要存放到环境角色中，很多情况下我们使用Map来充当环境角色就足够了。

代码实现

```
class Context {}
abstract class Expression {
    public abstract Object interpreter(Context ctx);
}
class TerminalExpression extends Expression {
    public Object interpreter(Context ctx){
        return null;
    }
}
class NonterminalExpression extends Expression {
    public NonterminalExpression(Expression...expressions){

    }
    public Object interpreter(Context ctx){
        return null;
    }
}
public class Client {
    public static void main(String[] args){
        String expression = "";
        char[] charArray = expression.toCharArray();
        Context ctx = new Context();
        Stack stack = new Stack();
        for(int i=0;i
         //进行语法判断，递归调用
    }
    Expression exp = stack.pop();
    exp.interpreter(ctx);
  }
}
```

文法递归的代码部分需要根据具体的情况来实现，因此在代码中没有体现。抽象表达式是生成语法集合的关键，每个非终结符表达式解释一个最小的语法单元，然后通过递归的方式将这些语法单元组合成完整的文法，这就是解释器模式。

解释器模式的优缺点

解释器是一个简单的语法分析工具，它最显著的优点就是扩展性，修改语法规则只需要修改相应的非终结符就可以了，若扩展语法，只需要增加非终结符类就可以了。

但是，解释器模式会引起类的膨胀，每个语法都需要产生一个非终结符表达式，语法规则比较复杂时，就可能产生大量的类文件，为维护带来非常多的麻烦。同时，由于采用递归调用方法，每个非终结符表达式只关心与自己相关的表达式，每个表达式需要知道最终的结果，必须通过递归方式，无论是面向对象的语言还是面向过程的语言，递归都是一个不推荐的方式。由于使用了大量的循环和递归，效率是一个不容忽视的问题。特别是用于解释一个解析复杂、冗长的语法时，效率是难以忍受的。

解释器模式的适用场景

在以下情况下可以使用解释器模式：

- 有一个简单的语法规则，比如一个sql语句，如果我们需要根据sql语句进行rm转换，就可以使用解释器模式来对语句进行解释。
- 一些重复发生的问题，比如加减乘除四则运算，但是公式每次都不同，有时是a+b−c*d，有时是a*b+c−d，等等等等个，公式千变万化，但是都是由加减乘除四个非终结符来连接的，这时我们就可以使用解释器模式。

注意事项

解释器模式真的是一个比较少用的模式，因为对它的维护实在是太麻烦了，想象一下，一坨一坨的非终结符解释器，假如不是事先对文法的规则了如指掌，或者是文法特别简单，则很难读懂它的逻辑。解释器模式在实际的系统开发中使用的很少，因为他会引起效率、性能以及维护等问题。

备忘录模式

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。这样就可以将该对象恢复到原先保存的状态

类型：行为类

类图：

图片 1.18 memento-pattern

我们在编程的时候，经常需要保存对象的中间状态，当需要的时候，可以恢复到这个状态。比如，我们使用Eclipse进行编程时，假如编写失误（例如不小心误删除了几行代码），我们希望返回删除前的状态，便可以使用Ctrl+Z来进行返回。这时我们便可以使用备忘录模式来实现。

备忘录模式的结构

- **发起人：**记录当前时刻的内部状态，负责定义哪些属于备份范围的状态，负责创建和恢复备忘录数据。
- **备忘录：**负责存储发起人对象的内部状态，在需要的时候提供发起人需要的内部状态。
- **管理角色：**对备忘录进行管理，保存和提供备忘录。

通用代码实现

```
class Originator {
    private String state = "";

    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
    public Memento createMemento(){
        return new Memento(this.state);
    }
```

```
    public void restoreMemento(Memento memento){
        this.setState(memento.getState());
    }
}

class Memento {
    private String state = "";
    public Memento(String state){
        this.state = state;
    }
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
}
class Caretaker {
    private Memento memento;
    public Memento getMemento(){
        return memento;
    }
    public void setMemento(Memento memento){
        this.memento = memento;
    }
}
public class Client {
    public static void main(String[] args){
        Originator originator = new Originator();
        originator.setState("状态1");
        System.out.println("初始状态:"+originator.getState());
        Caretaker caretaker = new Caretaker();
        caretaker.setMemento(originator.createMemento());
        originator.setState("状态2");
        System.out.println("改变后状态:"+originator.getState());
        originator.restoreMemento(caretaker.getMemento());
        System.out.println("恢复后状态:"+originator.getState());
    }
}
```

代码演示了一个单状态单备份的例子，逻辑非常简单：Originator类中的state变量需要备份，以便在需要的时候恢复；Memento类中，也有一个state变量，用来存储Originator类中state变量的临时状态；而Caretaker类就是用来管理备忘录类的，用来向备忘录对象中写入状态或者取回状态。

多状态多备份备忘录

通用代码演示的例子中，Originator类只有一个state变量需要备份，而通常情况下，发起人角色通常是一个java Bean，对象中需要备份的变量不止一个，需要备份的状态也不止一个，这就是多状态多备份备忘录。实现备忘录的方法很多，备忘录模式有很多变形和处理方式，像通用代码那样的方式一般不会用到，多数情况下的备忘录模式，是多状态多备份的。其实实现多状态多备份也很简单，最常用的方法是，我们在Memento中增加一个Map容器来存储所有的状态，在Caretaker类中同样使用一个Map容器才存储所有的备份。下面我们给出一个多状态多备份的例子：

```
class Originator {
    private String state1 = "";
    private String state2 = "";
    private String state3 = "";

    public String getState1() {
        return state1;
    }
    public void setState1(String state1) {
        this.state1 = state1;
    }
    public String getState2() {
        return state2;
    }
    public void setState2(String state2) {
        this.state2 = state2;
    }
    public String getState3() {
        return state3;
    }
    public void setState3(String state3) {
        this.state3 = state3;
    }
    public Memento createMemento(){
        return new Memento(BeanUtils.backupProp(this));
    }

    public void restoreMemento(Memento memento){
        BeanUtils.restoreProp(this, memento.getStateMap());
    }
    public String toString(){
        return "state1="+state1+"state2="+state2+"state3="+state3;
    }
}
class Memento {
```

```
    private Map stateMap;

    public Memento(Map map){
        this.stateMap = map;
    }

    public Map getStateMap() {
        return stateMap;
    }

    public void setStateMap(Map stateMap) {
        this.stateMap = stateMap;
    }
}
class BeanUtils {
    public static Map backupProp(Object bean){
        Map result = new HashMap();
        try{
            BeanInfo beanInfo = Introspector.getBeanInfo(bean.getClass());
            PropertyDescriptor[] descriptors = beanInfo.getPropertyDescriptors();
            for(PropertyDescriptor des: descriptors){
                String fieldName = des.getName();
                Method getter = des.getReadMethod();
                Object fieldValue = getter.invoke(bean, new Object[]{});
                if(!fieldName.equalsIgnoreCase("class")){
                    result.put(fieldName, fieldValue);
                }
            }

        }catch(Exception e){
            e.printStackTrace();
        }
        return result;
    }

    public static void restoreProp(Object bean, Map propMap){
        try {
            BeanInfo beanInfo = Introspector.getBeanInfo(bean.getClass());
            PropertyDescriptor[] descriptors = beanInfo.getPropertyDescriptors();
            for(PropertyDescriptor des: descriptors){
                String fieldName = des.getName();
                if(propMap.containsKey(fieldName)){
                    Method setter = des.getWriteMethod();
                    setter.invoke(bean, new Object[]{propMap.get(fieldName)});
                }
```

```
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
class Caretaker {
    private Map memMap = new HashMap();
    public Memento getMemento(String index){
        return memMap.get(index);
    }

    public void setMemento(String index, Memento memento){
        this.memMap.put(index, memento);
    }
}
class Client {
    public static void main(String[] args){
        Originator ori = new Originator();
        Caretaker caretaker = new Caretaker();
        ori.setState1("中国");
        ori.setState2("强盛");
        ori.setState3("繁荣");
        System.out.println("===初始化状态===n"+ori);

        caretaker.setMemento("001",ori.createMemento());
        ori.setState1("软件");
        ori.setState2("架构");
        ori.setState3("优秀");
        System.out.println("===修改后状态===n"+ori);

        ori.restoreMemento(caretaker.getMemento("001"));
        System.out.println("===恢复后状态===n"+ori);
    }
}
```

备忘录模式的优缺点和适用场景

备忘录模式的优点有：

- 当发起人角色中的状态改变时，有可能这是个错误的改变，我们使用备忘录模式就可以把这个错误的改变还原。
- 备份的状态是保存在发起人角色之外的，这样，发起人角色就不需要对各个备份的状态进行管理。

备忘录模式的缺点：

- 在实际应用中，备忘录模式都是多状态和多备份的，发起人角色的状态需要存储到备忘录对象中，对资源的消耗是比较严重的。

如果有需要提供回滚操作的需求，使用备忘录模式非常适合，比如jdbc的事务操作，文本编辑器的Ctrl+Z恢复等。