

Java 设计模式和原则

23 种设计模式

1、单例模式

定义

单例模式，是一种常用的软件设计模式。在它的核心结构中只包含一个被称为单例的特殊类。通过单例模式可以保证系统中一个类只有一个实例。即一个类只有一个对象实例。

特点

- 1、单例类只能有一个实例。
- 2、单例类必须自己创建自己的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例

单例模式的要点：

- 1，私有的构造方法
- 2，指向自己实例的私有静态引用
- 3，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：

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;  
    }  
}
```

单例模式还有一种比较常见的形式：双重锁的形式

```
public class Singleton{
    private static volatile Singleton instance=null;
    private Singleton(){
        //do something
    }
    public static Singleton getInstance(){
        if(instance==null){
            synchronized(Singleton.class){
                if(instance==null){
                    instance=new Singleton();
                }
            }
        }
        return instance;
    }
}
```

这个模式将同步内容下方到 if 内部，提高了执行的效率，不必每次获取对象时都进行同步，只有第一次才同步，创建了以后就没必要了。

这种模式中双重判断加同步的方式，比第一个例子中的效率大大提升，因为如果单层 if 判断，在服务器允许的情况下，假设有一百个线程，耗费的时间为 $100 \times (\text{同步判断时间} + \text{if 判断时间})$ ，而如果双重 if 判断，100 的线程可以同时 if 判断，理论消耗的时间只有一个 if 判断的时间。

所以如果面对高并发的情况，而且采用的是懒汉模式，最好的选择就是双重判断加同步的方式。

单例模式的优点：

- 1，在内存中只有一个对象，节省内存空间。

- 2，避免频繁的创建销毁对象，可以提高性能。
- 3，避免对共享资源的多重占用。
- 4，可以全局访问。

单例模式的缺点

- 1，扩展困难，由于 getInstance 静态函数没有办法生成子类的实例。如果要拓展，只有重写那个类。
- 2，隐式使用引起类结构不清晰。
- 3，导致程序内存泄露的问题。

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。以下为使用单例模式的场景：

- 1，需要频繁实例化然后销毁的对象。
- 2，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。
- 3，资源共享的情况下，避免由于资源操作时导致的性能或损耗等
- 4，控制资源的情况下，方便资源之间的互相通信。

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 中单例模式的一些常见问题

单例模式的对象长时间不用会被 jvm 垃圾收集器收集吗

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。

jvm 卸载类的判定条件如下：

- 1，该类所有的实例都已经被回收，也就是 java 堆中不存在该类的任何实例。
- 2，加载该类的 ClassLoader 已经被回收。
- 3，该类对应的 java.lang.Class 对象没有任何地方被引用，无法在任何地方通过反射访问该类的方法。

只有三个条件都满足，jvm 才会在垃圾收集的时候卸载类。显然，单例的类不满足条件一，因此单例类也不会被回收。

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，一定要注意不要使用反射产生新的单例对象。

在 `getInstance()` 方法上同步有优势还是仅同步必要的块更优优势？

因为锁定仅仅在创建实例时才有意义，然后其他时候实例仅仅是只读访问的，因此只同步必要的块的性能更优，并且是更好的选择。

缺点：只有在第一次调用的时候，才会出现生成 2 个对象，才必须要求同步。而一旦 `singleton` 不为 `null`，系统依旧花费同步锁开销，有点得不偿失。

单例类可以被继承吗

根据单例实例构造的时机和方式不同，单例模式还可以分成几种。但对于这种通过私有化构造函数，静态方法提供实例的单例类而言，是不支持继承的。

这种模式的单例实现要求每个具体的单例类自身来维护单例实例和限制多个实例的生成。但可以采用另外一种实现单例的思路：登记式单例，来使得单例对继承开放。

2、工厂模式

定义

工厂模式是 Java 中最常用的设计模式之一。这种类型的设计模

式属于创建型模式，它提供了一种创建对象的最佳方式。

工厂模式主要是为创建对象提供过度接口，以便将创建对象的具体过程屏蔽隔离起来，达到提高灵活性的目的。

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）

工厂方法模式（也叫多形性工厂）

抽象工厂模式（也叫工具箱）

简单工厂模式

实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类（这些产品类继承自一个父类或接口）的实例。简单工厂模式的创建目标，所有创建的对象都是充当这个角色的某个具体类的实例。

工厂方法模式

工厂方法是粒度很小的设计模式，因为模式的表现只是一个抽象的方法。提前定义用于创建对象的接口，让子类决定实例化具体的某一个类，即在工厂和产品中间增加接口，工厂不再负责产品的创建，由接口针对不同条件返回具体的类实例，由具体类实例去实现。

抽象工厂模式

当有多个抽象角色时使用的工厂模式。抽象工厂模式可以向客户

端提供一个接口，使客户端在不必指定产品的具体的情况下，创建多个产品对象。它有多个抽象产品类，每个抽象产品类可以派生出多个具体产品类，一个抽象工厂类，可以派生出多个具体工厂类，每个具体工厂类可以创建多个具体产品类的实例。

工厂方法模式应该在实际中用的较多，我们以工厂方法模式举例。

抽象的产品类：定义 car 交通工具类

```
public interface Car {  
    void gotowork();  
}
```

定义实际的产品类，总共定义两个，bike 和 bus 分别表示不同的交通工具类

```
public class Bike implements Car {  
    @Override  
    public void gotowork() {  
        System.out.println("骑自行车去上班！");  
    }  
}
```

```
public class Bus implements Car {  
    @Override  
    public void gotowork() {  
        System.out.println("坐公交车去上班！");  
    }  
}
```

定义抽象的工厂接口

```
public interface ICarFactory {  
    Car getCar();  
}
```

具体的工厂子类，分别为每个具体的产品类创建不同的工厂子类


```
public class BikeFactory implements ICarFactory {  
    @Override  
    public Car getCar() {  
        return new Bike();  
    }  
}
```

```
public class BusFactory implements ICarFactory {  
    @Override  
    public Car getCar() {  
        return new Bus();  
    }  
}
```

简单的测试类，来验证不同的工厂能够产生不同的产品对象

```
public class TestFactory {  
    @Test  
    public void test() {  
        ICarFactory factory = null;  
        // bike  
        factory = new BikeFactory();  
        Car bike = factory.getCar();  
        bike.gotowork();  
  
        // bus  
        factory = new BusFactory();  
        Car bus = factory.getCar();  
        bus.gotowork();  
    }  
}
```

工厂模式的优点

- 1、一个调用者想创建一个对象，只要知道其名称就可以了，降低了耦合度。
- 2、扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。使得代码结构更加清晰。
- 3、屏蔽产品的具体实现，调用者只关心产品的接口。

工厂模式的缺点

每次增加一个产品时，都需要增加一个具体类和对象实现工厂（这里可以使用反射机制来避免），使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。所以对于简单对象来说，使用工厂模式反而增加了复杂度。

工厂模式的适用场景

- 1， 一个对象拥有很多子类。
- 2， 创建某个对象时需要进行许多额外的操作。
- 3， 系统后期需要经常扩展，它把对象实例化的任务交由实现类完成，扩展性好。

关于 Java 中的工厂模式的一些常见问题

利用父类的向下转型（使用父类类型的引用指向子类的对象）是可以达到类似于工厂模式的效果的，那为什么还要用工厂模式呢？

把指向子类对象的父类引用赋给子类引用叫做向下转型，如：

```
Class Student extends Person
Person s = new Student();
s = (Student)person ;
```

使用向下转型在客户端实例化子类的时候，严重依赖具体的子类的名字。当我们需要更改子类的构造方法的时候，比如增加一个参数，或者更改了子类的类名，所有的 new 出来的子类都需要跟着更改。

但如果我们使用工厂模式，我们仅仅需要在工厂中修改一下 new

的代码，其余项目中用到此实例的都会跟着改，而不需要我们手动去操作。

总结

无论是简单工厂模式、工厂模式还是抽象工厂模式，它们本质上都是将不变的部分提取出来，将可变的部分留作接口，以达到最大程度上的复用。究竟用哪种设计模式更适合，这要根据具体的业务需求来决定。

3、原型模式

定义

通过复制现有的对象实例来创建新的对象实例。

实现

实现 Cloneable 接口

Cloneable 接口的作用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，所有的类父类都是 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，原型类需要将 clone 方法的作用域修改为 public 类型。

示例：

例如，对于发邮件发邀请函，邮件类大部分内容都是一样的：邀请原由、相邀地点，相聚时间等等，但对于被邀请者的名称和发送的邮件地址是不同的。

定义 Mail 类：

```
public class Mail implements Cloneable {
    private String receiver;
    private String subject;
    private String content;
    private String tail;
    public Mail(EventTemplate et) {
        this.tail = et.geteventContent();
        this.subject = et.geteventSubject();
    }
    @Override
    public Mail clone() {
        Mail mail = null;
        try {
            mail = (Mail) super.clone();
        } catch (CloneNotSupportedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return mail;
    }
    //get、set.....
}
```

测试方法：

```
public static void main(String[] args) {  
    int i = 0;  
    int MAX_COUNT = 10;  
    EventTemplate et =  
new EventTemplate("邀请函（不变）", "婚嫁生日啥的....（不变部分）");  
    Mail mail = new Mail(et);  
    while (i < MAX_COUNT) {  
        Mail cloneMail = mail.clone();  
        cloneMail.setContent("XXX先生（女士）（变化部分）"  
+ mail.getTail());  
        cloneMail.setReceiver("每个人的邮箱地址...com（变化部分）");  
        sendMail(cloneMail);  
        i++;  
    }  
}
```

优点

- 1、使用原型模型创建一个对象比直接 new 一个对象更有效率，因为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。
- 2、隐藏了制造新实例的复杂性，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。

缺点

- 1、由于使用原型模式复制对象时不会调用类的构造方法，所以原型模式无法和单例模式组合使用，因为原型类需要将 clone 方法的作用域修改为 public 类型，那么单例模式的条件就无法满足了。
- 2、使用原型模式时不能有 final 对象。
- 3、Object 类的 clone 方法只会拷贝对象中的基本数据类型，对于数组，引用对象等只能另行拷贝。这里涉及到深拷贝和浅拷贝的概念。

深拷贝与浅拷贝

浅拷贝

将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的（这样不安全）。

深拷贝

将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。

那么深拷贝如何具体实现呢？

继续上面的例子，增加了一个 ArrayList 属性。

```
private String receiver;  
private String subject;  
private String content;  
private String tail;  
private ArrayList<String> ars;
```

此时，单 `mail = (Mail) super.clone();` 无法将 `ars` 指向的地址区域改变，必须另行拷贝：

```
try {  
    mail = (Mail) super.clone();  
    mail.ars = (ArrayList<String>)this.ars.clone();  
} catch (CloneNotSupportedException e) {  
    e.printStackTrace();  
}
```

适用场景

- 1、复制对象的结构和数据。
- 2、希望对目标对象的修改不影响既有的原型对象。

3、创建一个对象的成本比较大。

4、生成器模式

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。生成器模式利用一个导演者对象和具体建造者对象一个一个地建造出所有的零件，从而建造出完整的对象。

四个要素

Builder：生成器接口，定义创建一个 Product 对象所需要的各个部件的操作。

ConcreteBuilder：具体的生成器实现，实现各个部件的创建，并负责组装 Product 对象的各个部件，同时还提供一个让用户获取组装完成后的产品对象的方法。

Director：指导者，也被称导向者，主要用来使用 Builder 接口，以一个统一的过程来构建所需要的 Product 对象。

Product：产品，表示被生成器构建的复杂对象，包含多个部件。

示例

网上有用 KFC 的例子来描述生成器模式，比较通俗易懂。

假设 KFC 推出两种套餐：奥尔良鸡腿堡套餐和香辣鸡腿堡套餐。

奥尔良套餐包括：一个奥尔良鸡腿堡、一个炸鸡翅、一杯雪碧。

鸡腿堡套餐包括：一个香辣鸡腿堡、一份薯条、一杯可乐。

每份套餐都是：主食、副食、饮料。

KFC 服务员要根据顾客的要求来提供套餐，那这个需求里面什么是固定的，什么是变化的呢？很明显顾客都是要的套餐，顾客的目的是一样的。套餐里面都是主食、副食、饮料，这也是固定的。至于主食是什么、副食是什么、饮料是什么，这个是变化的。

在实际的软件开发过程中，有时候面临着“一个复杂对象”的创建工作，其通常由各个部分的子对象采用一定的组合构成，由于需求的变化，这个复杂对象的各个部分或者其子对象经常要变化（例如，鸡腿堡套餐的顾客不喜欢可乐，要换奶茶），但是他们的结构却相对稳定（套餐都得是一份主食，副食及饮料）。当遇到这种场景时，使用生成器模式比较合适。

定义一个产品类：

```
public class Entity1{...}  
public class Entity2{...}  
public class Entity3{...}  
public class Product{  
    Entity1 entity1;  
    Entity2 entity2;  
    Entity3 entity3;  
}
```

产品类中的各个小模块是不一样的，由他们建造组成产品。

根据具体场景要求，定义 n 个生成器类：


```

public interface IBuild{
    public void createEntity1();
    public void createEntity2();
    public void createEntity3();
    public Product composite();
    public Product create();
}

public class BuildProduct implements IBuild{
    Product p = new Product();
    public void createEntity1(){
        //p.entity1 = ...
    }
    public Product create(){
        return composite();
    }
    .....
}

public class BuildProduct1 implements IBuild{
    Product p = new Product();
    public void createEntity1(){
        //p.entity1 = ...
    }
    .....
}

```

定义一个指挥者类，统一调度 project:

```

public class Director{
    private IBuild build;
    public Director(IBuild build){
        this.build = build;
    }
    public Product build(){
        build.create();
    }
    public static void main(){
        IBuild build = new BuildProduct();
        Director director = new Director(build);
        Product p = director.build();
    }
}

```

优点

- 1、使用生成器模式可以使客户端不必知道产品内部组成的细节。

2、具体的建造者类之间是相互独立的，对系统的扩展非常有利。

3、由于具体的建造者是独立的，因此可以对建造过程逐步细化，而不对其他的模块产生任何影响。

缺点

建造者模式的“加工工艺”是暴露的，这样使得建造者模式更加灵活，也使得工艺变得对客户不透明。（待考证，笔者这里不是很理解，欢迎说自己的见解）

应用场景

1、需要生成一个产品对象有复杂的内部结构。每一个内部成分本身可以是对象，也可以使一个对象的一个组成部分。

2、需要生成的产品对象的属性相互依赖。建造模式可以强制实行一种分步骤进行的建造过程。

3、在对象创建过程中会使用到系统中的其他一些对象，这些对象在产品对象的创建过程中不易得到。

5、适配器模式

定义

将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

角色

目标（Target）角色：这就是所期待得到的接口，也就是这类的接口是符合我们要求的。

源（Adapee）角色：我们要使用的接口，但是这个接口不符合我们的要求，也就是现在需要适配的接口。

适配器（Adaper）角色：适配器类是适配器模式的核心。适配器把源接口转换成目标接口。显然，这一角色不可以是接口，而必须是具体类。

分类

1、类适配器模式

```
class Adaptee {  
    public void specificRequest() {  
        System.out.println("特殊请求，这个是源角色");  
    }  
}  
/*这个是目标角色，所期待的接口*/  
  
interface Target {  
    public void request();  
}
```

现在想要实现这个 Target 接口，但是不想重构，想要用上已有的 Adaptee 类，这时可以定义一个适配器类，继承想要使用的类，并且实现期待的接口。

```
class Adapter extends Adaptee implements Target {  
    public void request() {  
        super.specificRequest();  
    }  
}
```

这样，使用适配器类和实现目标接口就完成了计划，测试：

```
public class Test{
    publicstatic void main(String[] args) {
        //使用特殊功能类，即适配类
        Targetadapter = new Adapter();
        adapter.request();
    }
}
```

2、对象适配器模式

适配器类关联已有的 Adaptee 类，并且实现标准接口，这样做的好处是不再需要继承。

```
class Adapter implements Target{
    private Adaptee adaptee;

    public Adapter (Adaptee adaptee) {
        this.adaptee= adaptee;
    }

    public void request() {
        this.adaptee.specificRequest();
    }
}
```

我们可以想到，此时输出结果和类适配器模式是相同的，测试：

```
public class Test{
    publicstatic void main(String[] args) {
        Targetadapter = new Adapter(new Adaptee());
        adapter.request();
    }
}
```

区别：

对象的适配器模式不是使用继承关系连接到 Adaptee 类，而是使用委派关系连接到 Adaptee 类。

优点

复用性

系统需要使用现有的类，而此类的接口不符合系统的需要。那么通过适配器模式就可以让这些功能得到更好的复用。

扩展性

在实现适配器功能的时候，可以自由调用自己开发的功能，从而自然地扩展系统的功能。

缺点

过多的使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现。所以适配器模式不适合在详细设计阶段使用它，它是一种补偿模式，专用来在系统后期扩展、修改时所用。

适用场景

- 1、已经存在的类的接口不符合我们的需求；
- 2、创建一个可以复用的类，使得该类可以与其他不相关的类或不可预见的类协同工作；
- 3、使用一些已经存在的子类而不需要对其进行子类化来匹配接口。
- 4、旧的系统开发的类已经实现了一些功能，但是客户端却只能以另外接口的形式访问，但我们不希望手动更改原有类的时候。

小结

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

6、装饰者模式

定义

在不必改变原类文件和原类使用的继承的情况下，动态地扩展一个对象的功能。

它是通过创建一个包装对象，也就是用装饰来包裹真实的对象来实现。

角色

抽象构件角色 (Project): 给出一个接口，以规范准备接收附加责任的对象。

具体构件角色 (Employee): 定义一个将要接收附加责任的类。

装饰角色 (Manager): 持有一个构件对象的实例，并定义一个抽象构件接口一致的接口。

具体装饰角色 (ManagerA、ManagerB): 负责给构件对象“贴上”附加的责任。

示例

公共接口：

```
public interface Person {  
    void eat();  
}
```

被装饰对象：

```
public class OldPerson implements Person {  
    @Override  
    public void eat() {  
        System.out.println("吃饭");  
    }  
}
```

装饰对象：

```
public class NewPerson implements Person {  
    private OldPerson p;  
  
    NewPerson(OldPerson p) {  
        this.p = p;  
    }  
  
    @Override  
    public void eat() {  
        System.out.println("生火");  
        System.out.println("做饭");  
        p.eat();  
        System.out.println("刷碗");  
    }  
}
```

测试：

```
public class PersonDemo {  
    public static void main(String[] args) {  
        OldPerson old = new OldPerson();  
        //old.eat();  
        NewPerson np = new NewPerson(old);  
        np.eat();  
    }  
}
```

通过例子可以看到，没有改变原来的 OldPerson 类，同时也没有定义他的子类而实现了 Person 的扩展，这就是装饰者模式的作用。

优点

1、使用装饰者模式比使用继承更加灵活，因为它选择通过一种动态的方式来扩展一个对象的功能，在运行时可以选择不同的装饰器，从而实现不同的行为。

2、通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合。可以使用多个具体装饰类来装饰同一对象，得到功能更为强大的对象。

3、具体构件类与具体装饰类可以独立变化，他能在低耦合的。用户可以根据需要来增加新的具体构件类和具体装饰类，在使用时再对其进行各种组合，原有代码无须改变，符合“开闭原则”。

缺点

1、会产生很多的小对象，增加了系统的复杂性

2、这种比继承更加灵活机动的特性，也同时意味着装饰模式比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻

找错误可能需要逐级排查，较为烦琐。

装饰者与适配器模式的区别

1、适配器模式主要用来兼容那些不能在一起工作的类，使他们转化为可以兼容目标接口，虽然也可以实现和装饰者一样的增加新职责，但目的不在此。装饰者模式主要是给被装饰者增加新职责的。

2、适配器模式是用新接口来调用原接口，原接口对新系统是不可见或者说不可用的。装饰者模式原封不动的使用原接口，系统对装饰的对象也通过原接口来完成使用。

3、适配器是知道被适配者的详细情况的（就是那个类或那个接口）。装饰者只知道其接口是什么，至于其具体类型（是基类还是其他派生类）只有在运行期间才知道。

装饰者和继承的区别

继承：

优点：代码结构清晰，而且实现简单。

缺点：对于每一个的需要增强的类都要创建具体的子类来帮助其增强，这样会导致继承体系过于庞大。

装饰者：

优点：内部可以通过多态技术对多个需要增强的类进行增强。

缺点：需要内部通过多态技术维护需要增强的类的实例。进而使得代码稍微复杂。

适用场景

- 1、需要扩展一个类的功能，或给一个类添加附加职责。
- 2、需要动态的给一个对象添加功能，这些功能可能不明确或者暂时的，可以随时很方便的动态撤销掉。
- 3、需要增加由一些基本功能的排列组合而产生的非常大量的功能，从而使继承关系变的不现实。
- 4、当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

7、代理模式

定义

为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

角色

- 1、**抽象角色**：声明真实对象和代理对象的共同接口。
- 2、**代理角色**：代理对象角色内部含有对真实对象的引用，从而可以操作真实对象，同时代理对象提供与真实对象相同的接口以便在

任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

3、真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。

分类

静态代理

静态代理也就是在程序运行前就已经存在代理类的字节码文件，代理类和委托类的关系在运行前就确定了。

示例

抽象角色，真实对象和代理对象共同的接口

```
public interface UserInfo{  
    public void queryUser ();  
    public void updateUser ();  
}
```

真是角色

```
public class UserImpl implements UserInfo{  
  
    @Override  
    public void queryUser() {  
        //查询方法略...  
    }  
  
    @Override  
    public void updateUser() {  
        //修改方法略...  
    }  
}
```

代理角色

```

public class UserProxy implements UserInfo {
    private UserInfo userImpl;

    public AccountProxy(UserInfo userImpl) {
        this.userImpl = userImpl;
    }

    @Override
    public void queryUser() {
        //这里可以扩展，增加一些查询之前需要执行的方法
        //查询方法略...
        //这里可以扩展，增加一些查询之后需要执行的方法
    }

    @Override
    public void updateUser() {
        //这里可以扩展，增加一些修改之前需要执行的方法
        //修改方法略...
        //这里可以扩展，增加一些修改之后需要执行的方法
    }
}

```

使用代理之后如何调用他的方法？

```

public class Test {
    public static void main(String[] args) {
        UserInfo userImpl = new UserImpl();
        UserInfo userProxy = new UserProxy(userImpl);
        userProxy.queryUser();
        userProxy.updateUser();
    }
}

```

动态代理

动态代理类的源码是程序在运行期间由 JVM 根据反射等机制动态生成的，所以不存在代理类的字节码文件。代理角色和真实角色的联系在程序运行时确定。

示例

抽象角色，真实对象和代理对象共同的接口

```
public interface UserInfo{
    public void queryUser ();
    public void updateUser ();
}
```

真实角色

```
public class UserImpl implements UserInfo{

    @Override
    public void queryUser() {
        //查询方法略...
    }

    @Override
    public void updateUser() {
        //修改方法略...
    }

}
```

代理角色处理器：

```
public class UserHandler implements InvocationHandler{

    private UserInfo userImpl;
    public UserHandler(UserInfo userImpl2){
        this.userImpl = userImpl2;
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        Object object = null;
        //方法开始前做一些事情
        if (method.getName().equals("queryUser")) {
            object = method.invoke(userImpl, args);
            //激活调用的方法
        }
        //方法结束后做一些事情
        return object;
    }
}
```

如何调用（和静态代理略有不同）

```
public class Test {  
    public static void main(String[] args) {  
        UserInfouserImpl =new UserImpl();  
        UserHandlerhandler = new UserHandler(userImpl);  
        UserInfouserProxy = (UserInfo)Proxy.newProxyInstance  
            (ClassLoader.getSystemClassLoader(),  
             newClass[]{UserInfo.class}, handler);  
        userProxy.queryUser();  
    }  
}
```

优点

业务类只需要关注业务逻辑本身，保证了业务类的重用性。这是代理的共有优点。

能够协调调用者和被调用者，在一定程度上降低了系统的耦合度。

缺点

由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢，例如保护代理。

实现代理模式需要额外的工作，而且有些代理模式的实现过程较为复杂，例如远程代理。

8、外观模式

定义

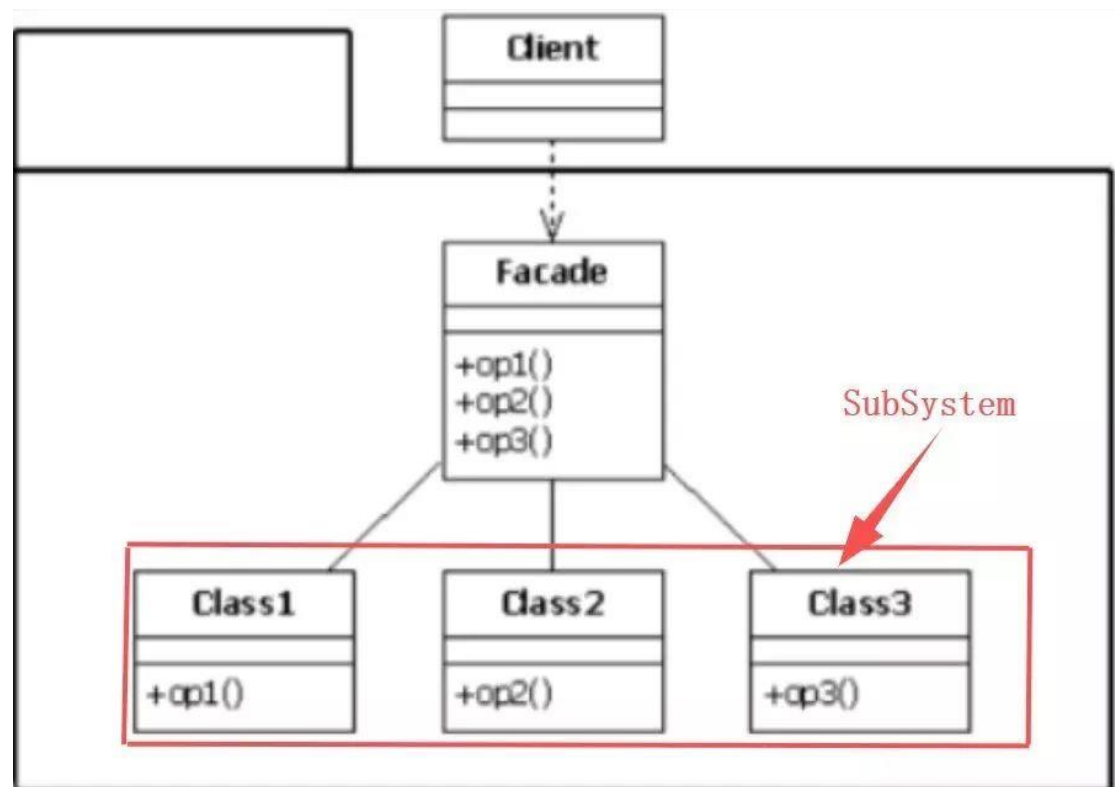
为子系统的一组接口提供一个一致的界面，Facade 模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

角色

1、外观(Facade)角色：客户端可以调用这个方法。此角色知晓相关子系统的功能和责任。在正常情况下，本角色会将所有从客户端发来的请求委派到相应的子系统去。

2、子系统(SubSystem)角色：可以同时有一个或者多个子系统。每个子系统都不是一个单独的类，而是一个类的集合。每个子系统都可以被客户端直接调用，或者被外观角色调用。子系统并不知道外观角色的存在，对于子系统而言，外观角色仅仅是另外一个客户端而已。

示意图



示例

1、子系统角色，由若干类组成

```
public class SubClass1 {  
    public void method1(){  
        System.out.println("这是子系统类1中的方法1");  
    }  
    public void method2(){  
        System.out.println("这是子系统类1中的方法2");  
    }  
}  
public class SubClass2 {  
    public void method1(){  
        System.out.println("这是子系统类2中的方法1");  
    }  
    public void method2(){  
        System.out.println("这是子系统类2中的方法2");  
    }  
}  
public class SubClass3 {  
    public void method1(){  
        System.out.println("这是子系统类3中的方法1");  
    }  
    public void method2(){  
        System.out.println("这是子系统类3中的方法2");  
    }  
}
```

2、外观角色类

```
public class FacadeClass {  
    public void FacadeMethod(){  
        SubClass1 s1 = new SubClass1();  
        s1.method1();  
        SubClass2 s2 = new SubClass2();  
        s2.method1();  
        SubClass3 s3 = new SubClass3();  
        s3.method1();  
    }  
}
```

3、客户端测试方法


```
public class ClientClass {  
    public static void main(String[] args) {  
        FacadeClass fc = new FacadeClass();  
        fc.FacadeMethod();  
    }  
}
```

Facade 类其实相当于子系统中 SubClass 类的外观界面，有了这个 Facade 类，那么客户端就不需要亲自调用子系统的那些具体实现的子类了，也不需要知道系统内部的实现细节，甚至都不需要知道这些子类的存在，客户端只需要跟 Facade 类交互就好了，从而更好地实现了客户端和子系统中具体类的解耦，让客户端更容易地使用系统。

同时，这样定义一个 Facade 类可以有效地屏蔽内部的细节，免得客户端去调用 Module 类时，发现一些不需要它知道的方法。如上代码，我的所有子类中的方法二都是方法一调用的方法，是配合方法一的，他们不需要被客户端调用，而且具有一定的保密性，这样使用外观模式时就可以不被客户端知道。

优点

实现了子系统与客户端之间的松耦合关系。

客户端屏蔽了子系统组件，减少了客户端所需处理的对象数目，并使得子系统使用起来更加容易。

适用场景

设计初期阶段，应该有意识的将不同层分离，层与层之间建立外

观模式。

开发阶段，子系统越来越复杂，增加外观模式提供一个简单的调用接口。

维护一个大型遗留系统的时候，可能这个系统已经非常难以维护和扩展，但又包含非常重要的功能，为其开发一个外观类，以便新系统与其交互。

外观模式总结

1、外观模式为复杂子系统提供了一个简单接口，并不为子系统添加新的功能和行为。

2、外观模式实现了子系统与客户端之间的松耦合关系。

3、外观模式没有封装子系统的类，只是提供了简单的接口。如果应用需要，它并不限制客户使用子系统类。因此可以灵活的在系统易用性与通用性之间选择。

4、外观模式注重的是简化接口，它更多的时候是从架构的层次去看整个系统，而并非单个类的层次。

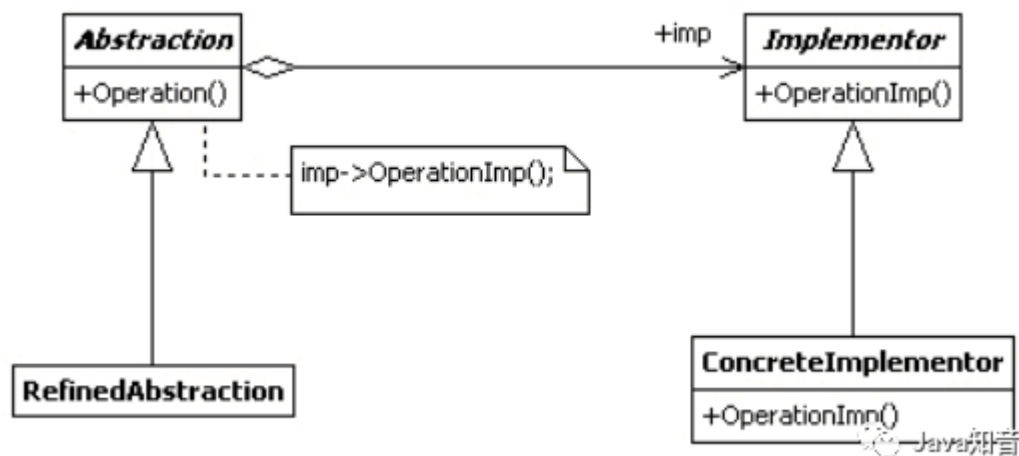
9、桥接模式

介绍

桥接模式 (Bridge)将抽象部分与实现部分分离，使它们都可以独立的变化。

桥接模式是一种结构式模式。

结构



代码实现

Implementor：定义实现接口。

```
interface Implementor {
    // 实现抽象部分需要的某些具体功能
    public void operationImpl();
}
```

Abstraction：定义抽象接口。

```
abstract class Abstraction {
    // 持有一个 Implementor 对象，形成聚合关系
    protected Implementor implementor;

    public Abstraction(Implementor implementor) {
        this.implementor = implementor;
    }

    // 可能需要转调实现部分的具体实现
    public void operation() {
        implementor.operationImpl();
    }
}
```

ConcreteImplementor：实现 Implementor 中定义的接口。

```
class ConcreteImplementorA implements Implementor {
    @Override
    public void operationImpl() {
        // 真正的实现
        System.out.println("具体实现A");
    }
}

class ConcreteImplementorB implements Implementor {
    @Override
    public void operationImpl() {
        // 真正的实现
        System.out.println("具体实现B");
    }
}
```

RefinedAbstraction：扩展 Abstraction 类。

```
class RefinedAbstraction extends Abstraction {

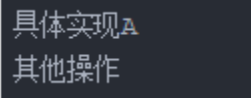
    public RefinedAbstraction(Implementor implementor) {
        super(implementor);
    }

    public void otherOperation() {
        // 实现一定的功能，可能会使用具体实现部分的实现方法，
        // 但是本方法更大的可能是使用 Abstraction 中定义的方法，
        // 通过组合使用 Abstraction 中定义的方法来完成更多的功能。
    }
}
```

测试代码

```
public class BridgePattern {
    public static void main(String[] args) {
        Implementor implementor = new ConcreteImplementorA();
        RefinedAbstraction abstraction = new RefinedAbstraction(implementor);
        abstraction.operation();
        abstraction.otherOperation();
    }
}
```

运行结果



具体实现A
其他操作

应用场景

1、如果你不希望在抽象和实现部分采用固定的绑定关系，可以采用桥接模式，来把抽象和实现部分分开，然后在程序运行期间来动态的设置抽象部分需要用到的具体的实现，还可以动态切换具体的实现。

2、如果出现抽象部分和实现部分都应该可以扩展的情况，可以采用桥接模式，让抽象部分和实现部分可以独立的变化，从而可以灵活的进行单独扩展，而不是搅在一起，扩展一边会影响到另一边。

3、如果希望实现部分的修改，不会对客户产生影响，可以采用桥接模式，客户是面向抽象的接口在运行，实现部分的修改，可以独立于抽象部分，也就不会对客户产生影响了，也可以说对客户是透明的。

4、如果采用继承的实现方案，会导致产生很多子类，对于这种情况，可以考虑采用桥接模式，分析功能变化的原因，看看是否能分离成不同的纬度，然后通过桥接模式来分离它们，从而减少子类的数目。

要点

如果一个系统需要在构件的抽象化角色和具体化角色之间增加

更多的灵活性，避免在两个层次之间建立静态的联系。

抽象化角色和具体化角色都应该可以被子类扩展。在这种情况下，桥接模式可以灵活地组合不同的抽象化角色和具体化角色，并独立化地扩展。

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

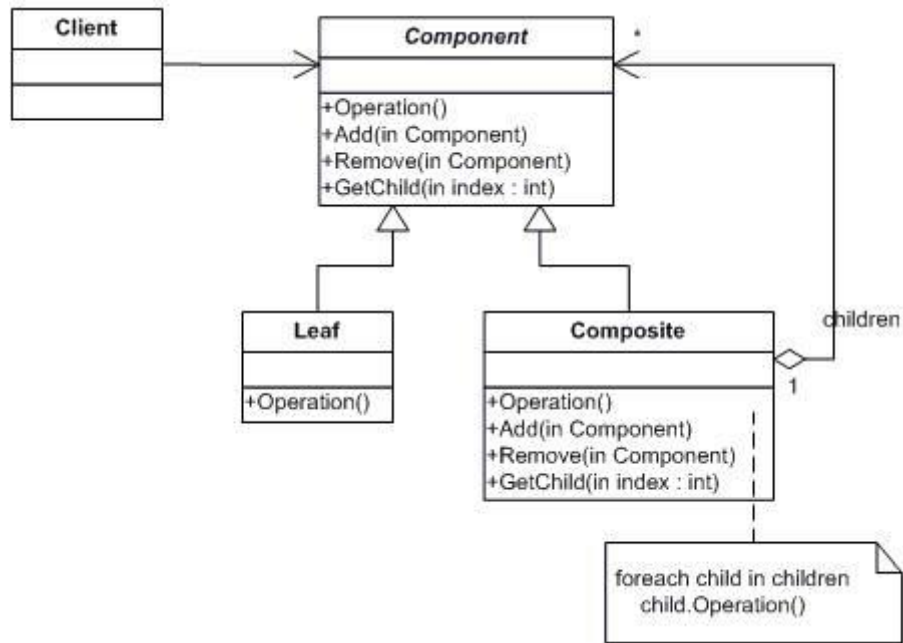
10、组合模式

介绍

组合模式又叫做部分-整体模式,它使我们树型结构的问题中,模糊了简单元素和复杂元素的概念,客户程序可以向处理简单元素一样来处理复杂元素,从而使得客户程序与复杂元素的内部结构解藕。

组合模式可以优化处理递归或分级数据结构.有许多关于分级数据结构的例子,使得组合模式非常有用武之地。

类图



组成部分

Component: 为参加组合的对象声明一个公共接口, 不管是组合还是叶结点.

Leaf: 在组合中表示叶子结点对象, 叶子结点没有子结点.

Composite: 表示参加组合的有子对象的对象, 并给出树枝构件的行为.

实例

FolderComponent

```
public abstract class FolderComponent {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(final String name) {
        this.name = name;
    }

    public FolderComponent() {
    }

    public FolderComponent(final String name) {
        this.name = name;
    }

    public abstract void add(FolderComponent component);

    public abstract void remove(FolderComponent component);

    public abstract void display();
}
```

FileLeaf


```
public class FileLeaf extends FolderComponent {
    public FileLeaf(final String name) {
        super(name);
    }

    @Override
    public void add(final FolderComponent component) {
        // ...
    }

    @Override
    public void remove(final FolderComponent component) {
        // ...
    }

    @Override
    public void display() {
        System.out.println("FileLeaf:" + this.getName());
    }
}
```

FolderComposite

```

public class FolderComposite extends FolderComponent {
    private final List<FolderComponent> components;

    public FolderComposite(final String name) {
        super(name);
        this.components = new ArrayList<FolderComponent>();
    }

    public FolderComposite() {
        this.components = new ArrayList<FolderComponent>();
    }

    @Override
    public void add(final FolderComponent component) {
        this.components.add(component);
    }

    @Override
    public void remove(final FolderComponent component) {
        this.components.remove(component);
    }

    @Override
    public void display() {
        System.out.println("FolderComposite---name:" + this.getName());
        for (final FolderComponent component : components) {
            System.out.println("FolderComposite---component-name:" + component.getName());
        }
    }
}

```

Client

```

public class Client
{
    public static void main(final String[] args)
    {
        final FolderComponent leaf = new FileLeaf("runnable file");
        leaf.display();

        final FolderComponent folder = new FolderComposite("new folder");
        folder.add(new FileLeaf("content1 in new folder"));
        folder.add(new FileLeaf("content2 in new folder"));
        folder.display();
    }
}

```

输出结果:

```
FileLeaf:runnable file
FolderComposite---name:new folder
FolderComposite---component-name:content1 in new folder
FolderComposite---component-name:content2 in new folder
```

适用场景

以下情况下适用 Composite 模式：

- 1、你想表示对象的部分-整体层次结构
- 2、你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

总结

组合模式解耦了客户程序与复杂元素内部结构，从而使客户程序可以向处理简单元素一样来处理复杂元素。

如果你想要创建层次结构，并可以在其中以相同的方式对待所有元素，那么组合模式就是最理想的选择。本章使用了一个文件

系统的例子来举例说明了组合模式的用途。在这个例子中，文件和目录都执行相同的接口，这是组合模式的关键。通过执行相同的接口，你就可以用相同的方式对待文件和目录，从而实现将文件或者目录储存为目录的子级元素。

11、策略模式

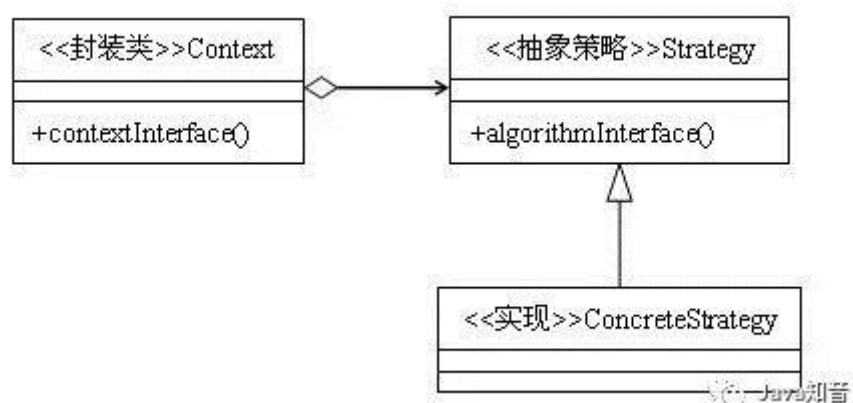
定义

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



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。

12、模板方法模式

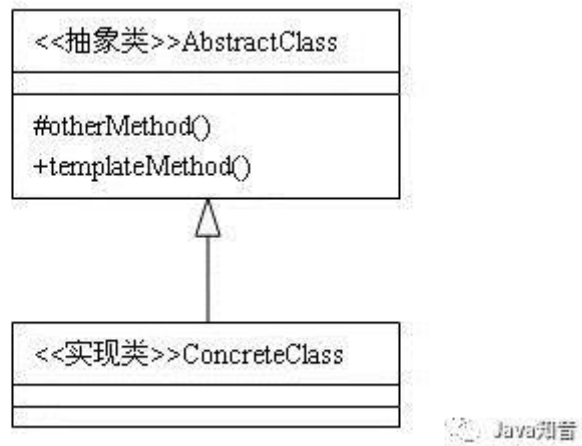
定义

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



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<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 < array.length; i++) {
            if (array[i] < 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：

```
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？

模板方法模式的结构

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。

13、观察者模式

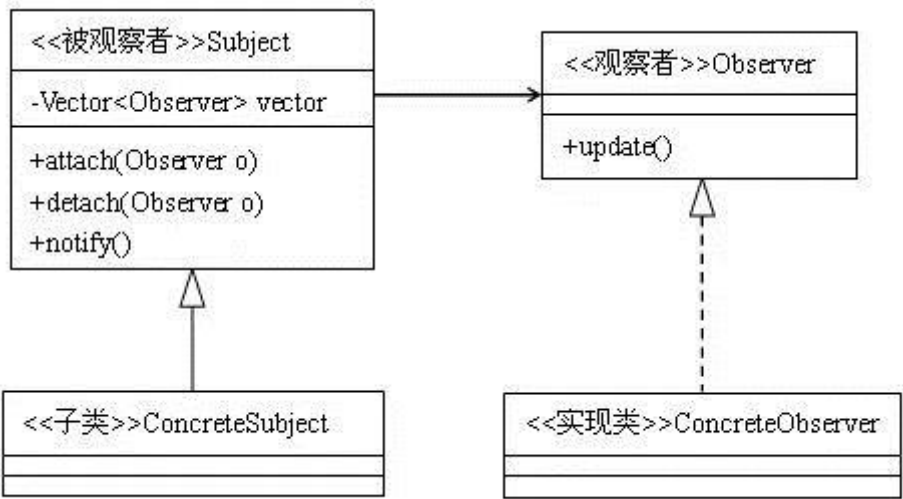
定义

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



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 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

```
public abstract class Subject {
    private Vector<Observer> obs = new Vector<Observer>();

    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();
}
```

```
public class ConcreteSubject extends Subject {
    public void doSomething(){
        System.out.println("被观察者事件反生");
        this.notifyObserver();
    }
}
```

```
public interface Observer {
    public void update();
}
```

```
public class ConcreteObserver1 implements Observer {
    public void update() {
        System.out.println("观察者1收到信息，并进行处理。");
    }
}
public 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，当观察者比较多时，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。

14、迭代器模式

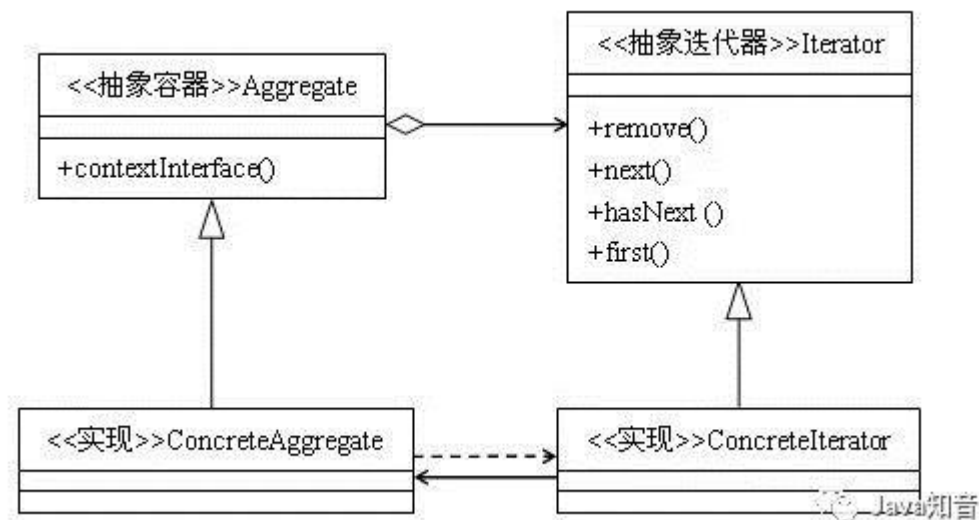
定义

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



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 `LinkedList`，`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();
}
```

续

```
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。

迭代器模式的优点

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。

15、解释器模式

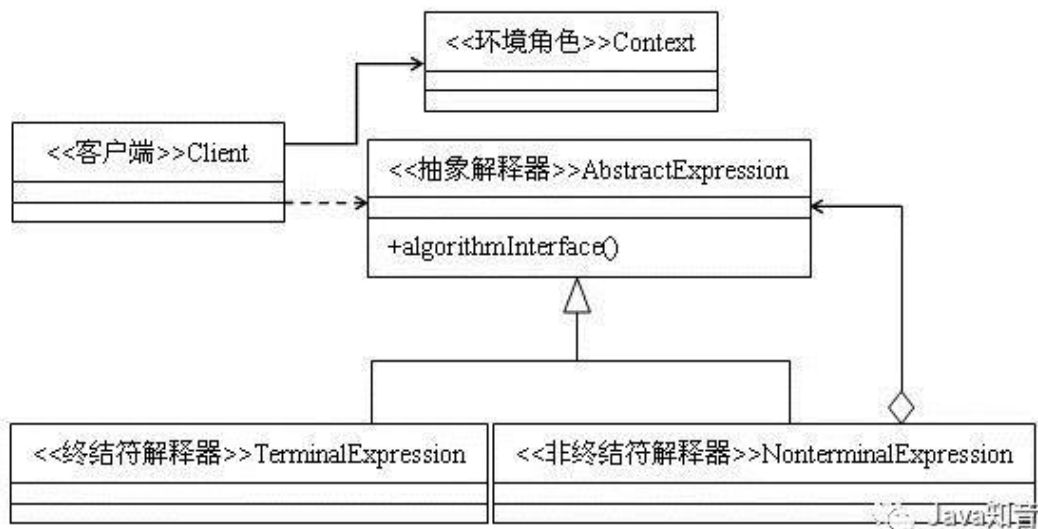
定义

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



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<Expression> stack = new Stack<Expression>();  
        for(int i=0;i<charArray.length;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。

16、访问者模式

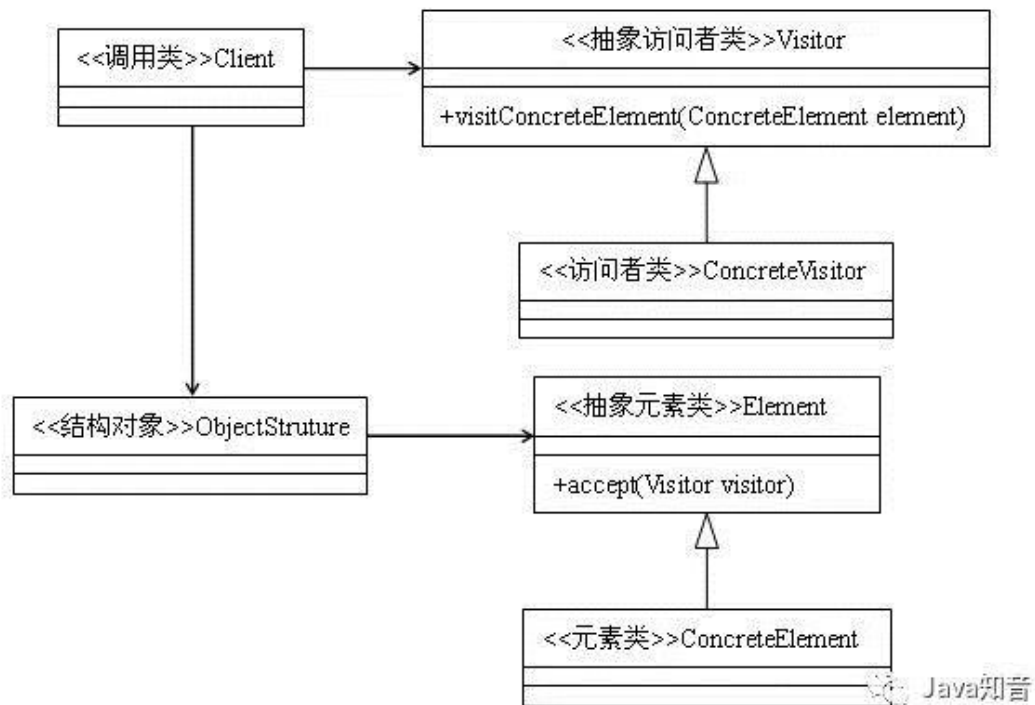
定义

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



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();
    }
}

```

续

```

class ObjectStruture {
    public static List<Element> getList() {
        List<Element> list = new ArrayList<Element>();
        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<Element> 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，你并不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。

17、命令模式

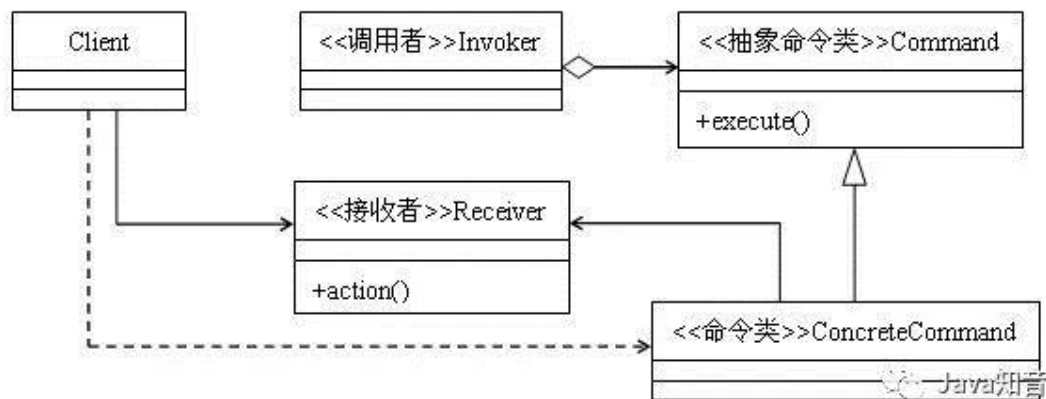
定义

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



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算是比较好理解的，下面我们重点说一下 **Invoker** 类和 **Receiver** 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，但这个预见的请求偏偏没有，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。

18、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

行为类模式

类图



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 javaBean，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 容器来存储所有的备份。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<String, Object> stateMap;

    public Memento(Map<String, Object> map) {
        this.stateMap = map;
    }
}
```

```

        this.stateMap = map;
    }

    public Map<String, Object> getStateMap() {
        return stateMap;
    }

    public void setStateMap(Map<String, Object> stateMap) {
        this.stateMap = stateMap;
    }
}

class BeanUtils {
    public static Map<String, Object> backupProp(Object bean) {
        Map<String, Object> result = new HashMap<String, Object>();
        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<String, Object> 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)});
                }
            }
        }
    }
}

```

```

        setter.invoke(bean, new Object[]{propMap.get(fieldName)});
    }
}
} catch (Exception e) {
    e.printStackTrace();
}
}
}
}
class Caretaker {
    private Map<String, Memento> memMap = new HashMap<String, Memento>();
    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。

备忘录模式的缺点有：

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。

19、责任链模式

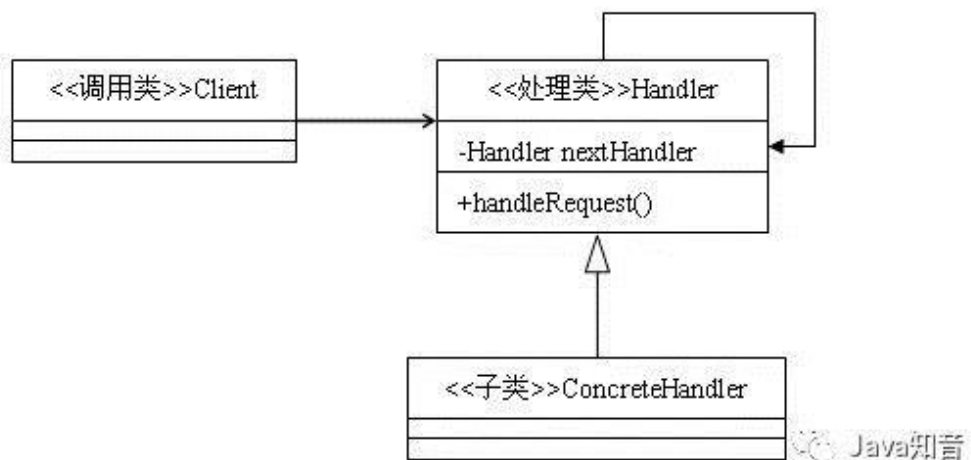
定义

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



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，这就是责任链模式的原理。

责任链模式的结构

责任链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 方法的主要思想是，如果满足处理的条件，则由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。

了解了责任链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，这就是责任链模式，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。

20、中介者模式

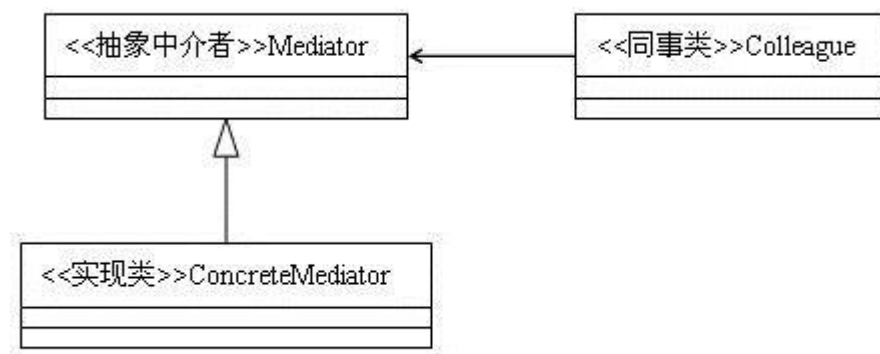
定义

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



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：

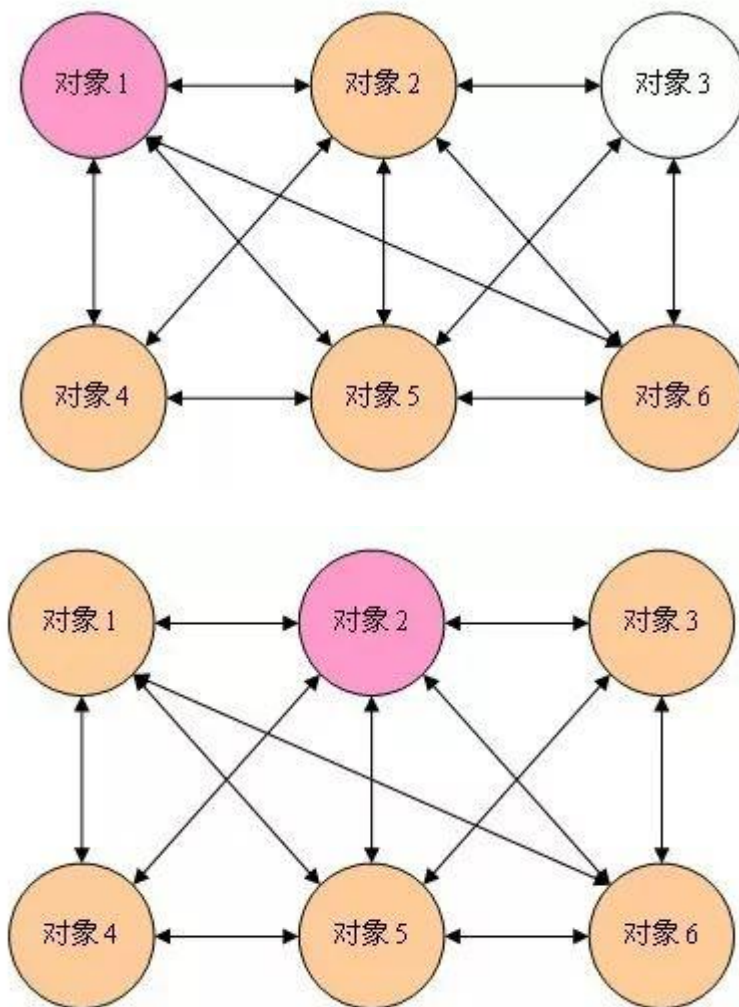
抽象中介者：定义好同事类对象到中介者对象的接口，用于各个同事类之间的通信。一般包括一个或几个抽象的事件方法，并由子类去实现。

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。

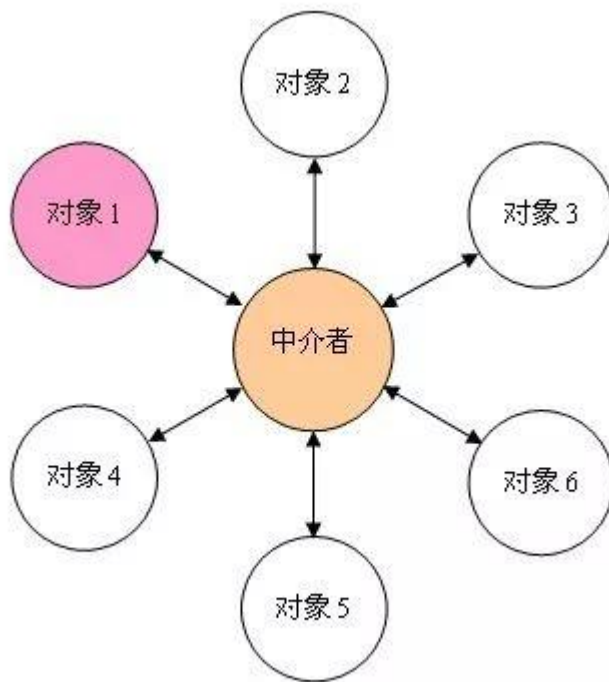
为什么要使用中介者模式

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。



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。



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;  
    }  
}
```

续

```

    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 BffectA();
}
class Mediator extends AbstractMediator {

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

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

    //处理B对A的影响
    public void BffectA() {
        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

- 1、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。
- 2、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。
- 3、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。

21、享元模式

在阎宏博士的《JAVA 与模式》一书中开头是这样描述享元（Flyweight）模式的：Flyweight 在拳击比赛中指最轻量级，即“蝇量级”或“雨量级”，这里选择使用“享元模式”的意译，是因为这样更能反映模式的用意。享元模式是对象的结构模式。享元模式以共享的方式高效地支持大量的细粒度对象。

Java 中的 String 类型

在 JAVA 语言中，String 类型就是使用了享元模式。String 对象是 final 类型，对象一旦创建就不可改变。在 JAVA 中字符串常量都是存在常量池中的，JAVA 会确保一个字符串常量在常量池中只有一个拷贝。String a="abc"，其中"abc"就是一个字符串常量。

```
public class Test {  
    public static void main(String[] args) {  
        String a = "abc";  
        String b = "abc";  
        System.out.println(a==b);  
    }  
}
```

上面的例子中结果为：true，这就说明 a 和 b 两个引用都指向了常量池中的同一个字符串常量"abc"。这样的设计避免了在创建 N 多相同对象时所产生的不必要的大量的资源消耗。

享元模式的结构

享元模式采用一个共享来避免大量拥有相同内容对象的开销。这

种开销最常见、最直观的就是内存的损耗。享元对象能做到共享的关键是区分**内蕴状态(Internal State)**和**外蕴状态(External State)**。

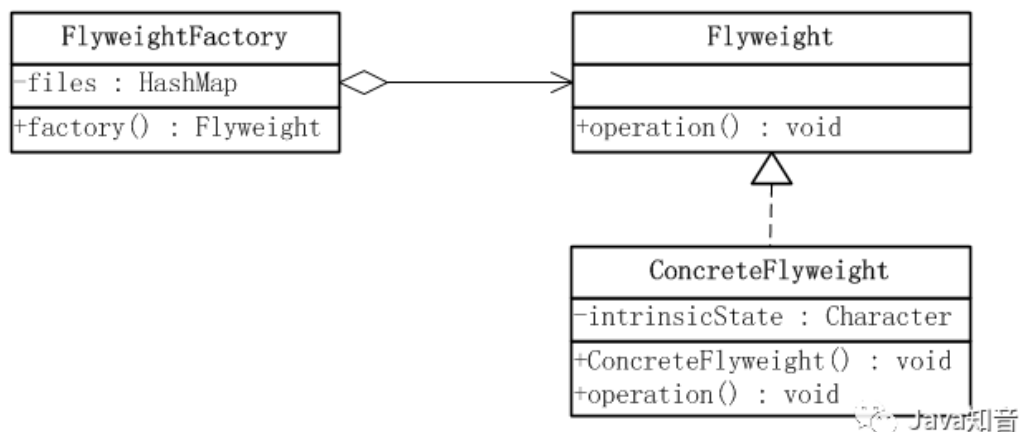
一个内蕴状态是存储在享元对象内部的，并且是不会随环境的改变而有所不同。因此，一个享元可以具有内蕴状态并可以共享。

一个外蕴状态是随环境的改变而改变的、不可以共享的。享元对象的外蕴状态必须由客户端保存，并在享元对象被创建之后，在需要使用的时候再传入到享元对象内部。外蕴状态不可以影响享元对象的内蕴状态，它们是相互独立的。

享元模式可以分成**单纯享元模式**和**复合享元模式**两种形式。

单纯享元模式

在单纯的享元模式中，所有的享元对象都是可以共享的。



单纯享元模式所涉及到的角色如下：

抽象享元(Flyweight)角色：给出一个抽象接口，以规定出所有具体享元角色需要实现的方法。

具体享元(ConcreteFlyweight)角色：实现抽象享元角色所规定出

的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。

享元工厂(FlyweightFactory)角色：本角色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象的时候，享元工厂角色会检查系统中是否已经有一个符合要求的享元对象。如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个合适的享元对象。

源代码

抽象享元角色类

```
public interface Flyweight {  
    //一个示意性方法，参数state是外蕴状态  
    public void operation(String state);  
}
```

具体享元角色类 ConcreteFlyweight 有一个内蕴状态，在本例中一个 Character 类型的 intrinsicState 属性代表，它的值应当在享元对象被创建时赋予。所有的内蕴状态在对象创建之后，就不会再改变了。

如果一个享元对象有外蕴状态的话，所有的外部状态都必须存储在客户端，在使用享元对象时，再由客户端传入享元对象。这里只有一个外蕴状态，operation()方法的参数 state 就是由外部传入的外蕴状态。


```

public class ConcreteFlyweight implements Flyweight {
    private Character intrinsicState = null;
    /**
     * 构造函数，内蕴状态作为参数传入
     * @param state
     */
    public ConcreteFlyweight(Character state){
        this.intrinsicState = state;
    }

    /**
     * 外蕴状态作为参数传入方法中，改变方法的行为，
     * 但是并不改变对象的内蕴状态。
     */
    @Override
    public void operation(String state) {
        // TODO Auto-generated method stub
        System.out.println("Intrinsic State = " + this.intrinsicState);
        System.out.println("Extrinsic State = " + state);
    }
}

```

享元工厂角色类，必须指出的是，客户端不可以直接将具体享元类实例化，而必须通过一个工厂对象，利用一个 factory() 方法得到享元对象。一般而言，享元工厂对象在整个系统中只有一个，因此也可以使用单例模式。

当客户端需要单纯享元对象的时候，需要调用享元工厂的 factory() 方法，并传入所需的单纯享元对象的内蕴状态，由工厂方法产生所需要的享元对象。

```

public class FlyweightFactory {
    private Map<Character, Flyweight> files = new HashMap<Character, Flyweight>();

    public Flyweight factory(Character state){
        // 先从缓存中查找对象
        Flyweight fly = files.get(state);
        if(fly == null){
            // 如果对象不存在则创建一个新的Flyweight对象
            fly = new ConcreteFlyweight(state);
            // 把这个新的Flyweight对象添加到缓存中
            files.put(state, fly);
        }
        return fly;
    }
}

```

客户端类

```

public class Client {

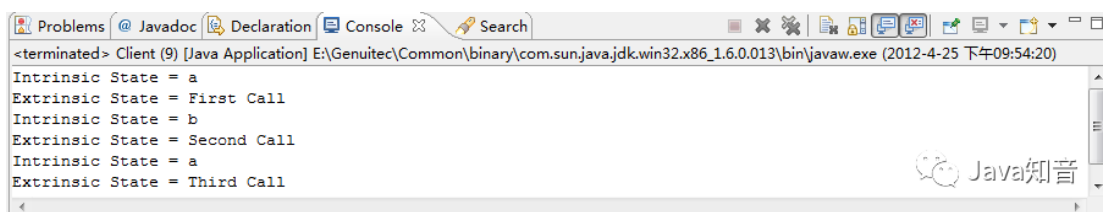
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        FlyweightFactory factory = new FlyweightFactory();
        Flyweight fly = factory.factory(new Character('a'));
        fly.operation("First Call");

        fly = factory.factory(new Character('b'));
        fly.operation("Second Call");

        fly = factory.factory(new Character('a'));
        fly.operation("Third Call");
    }
}

```

虽然客户端申请了三个享元对象，但是实际创建的享元对象只有两个，这就是共享的含义。运行结果如下：



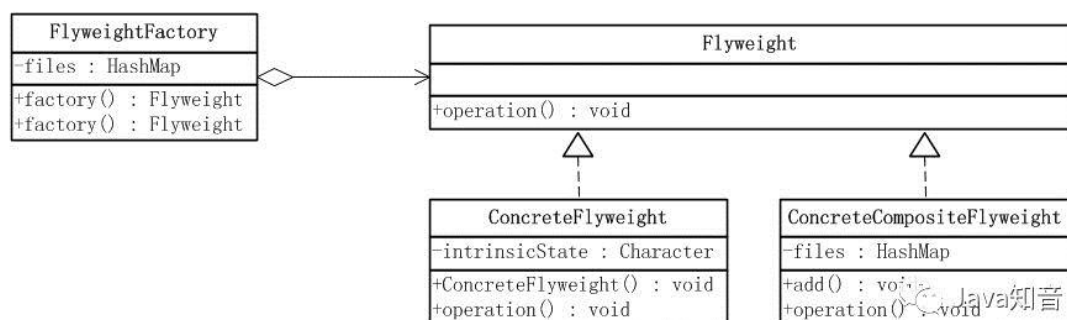
```

<terminated> Client (9) [Java Application] E:\Genuitec\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin\javaw.exe (2012-4-25 下午09:54:20)
Intrinsic State = a
Extrinsic State = First Call
Intrinsic State = b
Extrinsic State = Second Call
Intrinsic State = a
Extrinsic State = Third Call

```

复合享元模式

在单纯享元模式中，所有的享元对象都是单纯享元对象，也就是说都是可以直接共享的。还有一种较为复杂的情况，将一些单纯享元使用合成模式加以复合，形成复合享元对象。这样的复合享元对象本身不能共享，但是它们可以分解成单纯享元对象，而后者则可以共享。



复合享元角色所涉及到的角色如下：

抽象享元(Flyweight)角色： 给出一个抽象接口，以规定出所有具体享元角色需要实现的方法。

具体享元(ConcreteFlyweight)角色： 实现抽象享元角色所规定出的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。

复合享元(ConcreteCompositeFlyweight)角色： 复合享元角色所代表的对象是不可以共享的，但是一个复合享元对象可以分解成为多个本身是单纯享元对象的组合。复合享元角色又称作不可共享的享元对象。

享元工厂(FlyweightFactory)角色： 本角色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象的时候，享元工厂角色会检查系统中是否

已经有一个符合要求的享元对象。如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个合适的享元对象。

源代码

抽象享元角色类

```
public interface Flyweight {  
    //一个示意性方法，参数state是外蕴状态  
    public void operation(String state);  
}
```

具体享元角色类

```
public class ConcreteFlyweight implements Flyweight {  
    private Character intrinsicState = null;  
    /**  
     * 构造函数，内蕴状态作为参数传入  
     * @param state  
     */  
    public ConcreteFlyweight(Character state) {  
        this.intrinsicState = state;  
    }  
  
    /**  
     * 外蕴状态作为参数传入方法中，改变方法的行为，  
     * 但是并不改变对象的内蕴状态。  
     */  
    @Override  
    public void operation(String state) {  
        // TODO Auto-generated method stub  
        System.out.println("Intrinsic State = " + this.intrinsicState);  
        System.out.println("Extrinsic State = " + state);  
    }  
}
```

复合享元对象是由单纯享元对象通过复合而成的，因此它提供了

add()这样的聚集管理方法。由于一个复合享元对象具有不同的聚集元素，这些聚集元素在复合享元对象被创建之后加入，这本身就意味着复合享元对象的状态是会改变的，因此复合享元对象是不能共享的。

复合享元角色实现了抽象享元角色所规定的接口，也就是operation()方法，这个方法有一个参数，代表复合享元对象的外蕴状态。一个复合享元对象的所有单纯享元对象元素的外蕴状态都是与复合享元对象的外蕴状态相等的；而一个复合享元对象所含有的单纯享元对象的内蕴状态一般是不相等的，不然就没有使用价值了。

```
public class ConcreteCompositeFlyweight implements Flyweight {

    private Map<Character, Flyweight> files = new HashMap<Character, Flyweight>();
    /**
     * 增加一个新的单纯享元对象到聚集中
     */
    public void add(Character key , Flyweight fly){
        files.put(key, fly);
    }
    /**
     * 外蕴状态作为参数传入到方法中
     */
    @Override
    public void operation(String state) {
        Flyweight fly = null;
        for(Object o : files.keySet()){
            fly = files.get(o);
            fly.operation(state);
        }
    }
}
```

享元工厂角色提供两种不同的方法，一种用于提供单纯享元对象，另一种用于提供复合享元对象。

```

public class FlyweightFactory {
    private Map<Character, Flyweight> files = new HashMap<Character, Flyweight>();
    /**
     * 复合享元工厂方法
     */
    public Flyweight factory(List<Character> compositeState){
        ConcreteCompositeFlyweight compositeFly = new ConcreteCompositeFlyweight();

        for(Character state : compositeState){
            compositeFly.add(state, this.factory(state));
        }

        return compositeFly;
    }
    /**
     * 单纯享元工厂方法
     */
    public Flyweight factory(Character state){
        //先从缓存中查找对象
        Flyweight fly = files.get(state);
        if(fly == null){
            //如果对象不存在则创建一个新的Flyweight对象
            fly = new ConcreteFlyweight(state);
            //把这个新的Flyweight对象添加到缓存中
            files.put(state, fly);
        }
        return fly;
    }
}

```

客户端角色

```

public class Client {

    public static void main(String[] args) {
        List<Character> compositeState = new ArrayList<Character>();
        compositeState.add('a');
        compositeState.add('b');
        compositeState.add('c');
        compositeState.add('a');
        compositeState.add('b');

        FlyweightFactory flyFactory = new FlyweightFactory();
        Flyweight compositeFly1 = flyFactory.factory(compositeState);
        Flyweight compositeFly2 = flyFactory.factory(compositeState);
        compositeFly1.operation("Composite Call");

        System.out.println("-----");
        System.out.println("复合享元模式是否可以共享对象: " + (compositeFly1 == compositeFly2));

        Character state = 'a';
        Flyweight fly1 = flyFactory.factory(state);
        Flyweight fly2 = flyFactory.factory(state);
        System.out.println("单纯享元模式是否可以共享对象: " + (fly1 == fly2));
    }
}

```

```

class Client {

    static void main(String[] args) {
        List<Character> compositeState = new ArrayList<Character>();
        compositeState.add('a');
        compositeState.add('b');
        compositeState.add('c');
        compositeState.add('a');
        compositeState.add('b');

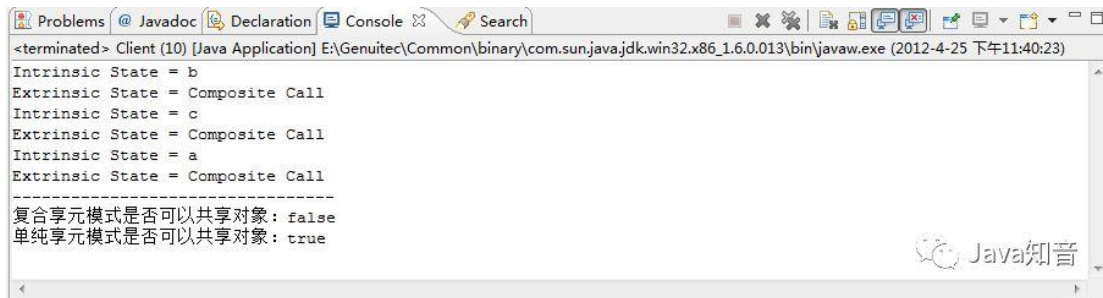
        FlyweightFactory flyFactory = new FlyweightFactory();
        Flyweight compositeFly1 = flyFactory.factory(compositeState);
        Flyweight compositeFly2 = flyFactory.factory(compositeState);
        compositeFly1.operation("Composite Call");

        System.out.println("-----");
        System.out.println("复合享元模式是否可以共享对象: " + (compositeFly1 == compositeFly2));

        Character state = 'a';
        Flyweight fly1 = flyFactory.factory(state);
        Flyweight fly2 = flyFactory.factory(state);
        System.out.println("单纯享元模式是否可以共享对象: " + (fly1 == fly2));
    }
}

```

运行结果如下：



```
<terminated> Client (10) [Java Application] E:\Genuitec\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin\javaw.exe (2012-4-25 下午11:40:23)
Intrinsic State = b
Extrinsic State = Composite Call
Intrinsic State = c
Extrinsic State = Composite Call
Intrinsic State = a
Extrinsic State = Composite Call
-----
复合享元模式是否可以共享对象: false
单纯享元模式是否可以共享对象: true
```

- 从运行结果可以看出，一个复合享元对象的所有单纯享元对象元素的外蕴状态都是与复合享元对象的外蕴状态相等的。即外运状态都等于 Composite Call。
- 从运行结果可以看出，一个复合享元对象所含有的单纯享元对象的内蕴状态一般是不相等的。即内蕴状态分别为 b、c、a。
- 从运行结果可以看出，复合享元对象是不能共享的。即使用相同的对象 compositeState 通过工厂分别两次创建出的对象不是同一个对象。
- 从运行结果可以看出，单纯享元对象是可以共享的。即使用相同的对象 state 通过工厂分别两次创建出的对象是同一个对象。

享元模式的优缺点

享元模式的优点在于它大幅度地降低内存中对象的数量。但是，它做到这一点所付出的代价也是很高的：

- 享元模式使得系统更加复杂。为了使对象可以共享，需要将

一些状态外部化，这使得程序的逻辑复杂化。

- 享元模式将享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。

22、状态模式

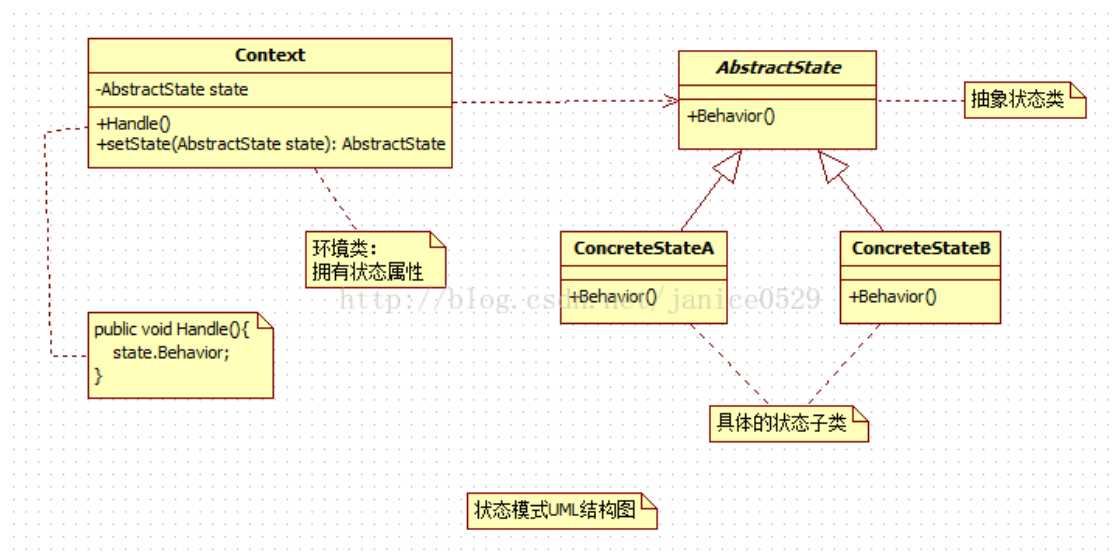
一、概述

当系统中某个对象存在多个状态，这些状态之间可以进行转换，而且对象在不同状态下行为不相同时可以使用状态模式。状态模式将一个对象的状态从该对象中分离出来，封装到专门的状态类中，使得对象状态可以灵活变化。状态模式是一种对象行为型模式。

二、适用场景

用于解决系统中复杂对象的多种状态转换以及不同状态下行为的封装问题。简单说就是处理对象的多种状态及其相互转换。

三、UML 类图



四、参与者

1>、AbstractState(抽象状态类):

在抽象状态类中定义申明了不同状态下的行为抽象方法，而由子类(不同的状态子类)中实现不同的行为操作。

2>、ConcreteState(实现具体状态下行为的状态子类):

抽象状态类的子类，每一个子类实现一个与环境类(Context)的一个状态相关的行为，每一个具体的状态类对应环境的一种具体状态，不同的具体状态其行为有所不同。

3>、Context(拥有状态对象的环境类):

拥有状态属性，因环境的多样性，它可拥有不同的状态，且在不同状态下行为也不一样。在环境类中维护一个抽象的状态实例，这个实例定义当前环境的状态(setState()方法)，而将具体的状态行为分离出来由不同的状态子类去完成。

五、用例学习

1、抽象状态类：State.java

```
/**
 * JAVA设计模式之 状态模式
 * 抽象状态类
 * @author lvzb.software@qq.com
 *
 */
public abstract class State {
    /**
     * 状态行为抽象方法,由具体的状态子类去实现不同的行为逻辑
     */
    public abstract void Behavior();
}
```

2、具体状态子类 A: ConcreteStateA.java

```
/**
 * 具体的状态子类A
 * @author lvzb.software@qq.com
 */
public class ConcreteStateA extends State {

    @Override
    public void Behavior() {
        // 状态A 的业务行为, 及当为该状态下时, 能干什么
        // 如: 手机在未欠费停机状态下, 能正常拨打电话
        System.out.println("手机在未欠费停机状态下, 能正常拨打电话");
    }
}
```

3、具体状态子类 B: ConcreteStateB.java

```

/**
 * 具体的状态子类B
 * @author lvzb.software@qq.com
 *
 */
public class ConcreteStateB extends State {

    @Override
    public void Behavior() {
        // 状态B 的业务行为，及当为该状态下时，能干什么
        // 如：手机在欠费停机状态下，不 能拨打电话
        System.out.println("手机在欠费停机状态下，不能拨打电话");
    }

}

```

4、拥有状态对象的环境类：Context.java

```

/**
 * 环境/上下文类<br/>
 * 拥有状态对象，且可以完成状态间的转换 [状态的改变/切换 在环境类中实现]
 * @author lvzb.software@qq.com
 *
 */
public class Context {
    // 维护一个抽象状态对象的引用
    private State state;

    /**
     * 模拟手机的话费属性<br/>
     * 环境状态如下：
     * 1>、当 bill >= 0.00$ : 状态正常 还能拨打电话
     * 2>、当 bill < 0.00$ : 手机欠费 不能拨打电话
     */
    private double bill;

    /**
     * 环境处理函数，调用状态实例行为 完成业务逻辑<br/>
     * 根据不同的状态实例引用 在不同状态下处理不同的行为
     */
    public void Handle(){
        checkState();
    }
}

```

```

        state.Behavior();
    }

    /**
     * 检查环境状态:状态的改变/切换 在环境类中实现
     */
    private void checkState(){
        if(bill >= 0.00){
            setState(new ConcreteStateA());
        } else {
            setState(new ConcreteStateB());
        }
    }

    /**
     * 设置环境状态<br/>
     * 私有方法，目的是 让环境的状态由系统环境自身来控制/切换,外部使用者无需关心环境内部的状态
     * @param state
     */
    private void setState(State state){
        this.state = state;
    }

    public double getBill() {
        return bill;
    }

    public void setBill(double bill) {
        this.bill = bill;
    }
}

```

5、测试客户端调用类：Client.java

```

public class Client {

    public static void main(String[] args) {
        Context context = new Context();
        context.setBill(5.50);
        System.out.println("当前话费余额: " + context.getBill() + "$");
        context.Handle();

        context.setBill(-1.50);
        System.out.println("当前话费余额: " + context.getBill() + "$");
        context.Handle();

        context.setBill(50.00);
        System.out.println("当前话费余额: " + context.getBill() + "$");
        context.Handle();
    }
}

```

6、程序运行结果：

```

当前话费余额: 5.5$
手机在未欠费停机状态下, 能正常拨打电话
当前话费余额: -1.5$
手机在欠费停机状态下, 不能拨打电话
当前话费余额: 50.0$
手机在未欠费停机状态下, 能正常拨打电话

```

六、扩展

状态模式中 关于状态的切换有两种不同的实现方式

方式一：状态的改变/切换 在环境类中实现。 如上面的用例代码 Context 类中的 checkState()方法。

```

/**
 * 检查环境状态:状态的改变/切换 在环境类中实现
 */
private void checkState(){
    if(bill >= 0.00){
        setState(new ConcreteStateA());
    } else {
        setState(new ConcreteStateB());
    }
}

```

方式二：状态的改变/切换 在具体的状态子类中实现。

实现步骤如下：

1> 在环境类 Context 类中 初始化一个状态实例对象，并将环境 Context 对象作为子类状态的构造参数传递到具体的状态子类实例中。

如在 Context.java 类中

```

// 设置初始状态
this.state = new ConcreteStateA(this);

```

2> 在具体的子类状态类中根据构造进来的 context 对象，通过调用 context 对象的属性值进行业务逻辑判断 进行状态的检查和切换。

如在 具体的状态子类 ConcreteStateA.java 类中：

```

/**
 * 具体的状态子类A
 * @author lvzb.software@qq.com
 */
public class ConcreteStateA extends State {
    private Context ctx;

    public ConcreteStateA(Context context){
        ctx = context;
    }

    @Override
    public void Behavior() {
        // 状态A 的业务行为，及当为该状态下时，能干什么
        // 如：手机在未欠费停机状态下，能正常拨打电话
        System.out.println("手机在未欠费停机状态下，能正常拨打电话");
        checkState();
    }

    /**
     * 检查状态 是否需要进行状态的转换<br/>
     * 状态的切换由具体状态子类中实现
     */
    private void checkState(){
        if (ctx.getBill() < 0.00) {
            ctx.setState(new ConcreteStateB(ctx));
        }
    }
}

```

设计模式六大原则

1、单一职责原则

定义

不要存在多于一个导致类变更的原因。通俗的说，即一个类只负

责一项职责。

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

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

说到单一职责原则，很多人都会不屑一顾。因为它太简单了。稍有经验的程序员即使从来没有读过设计模式、从来没有听说过单一职责原则，在设计软件时也会自觉的遵守这一重要原则，因为这是常识。在软件编程中，谁也不希望因为修改了一个功能导致其他的功能发生故障。而避免出现这一问题的方法便是遵循单一职责原则。虽然单一职责原则如此简单，并且被认为是常识，但是即便是经验丰富的程序员写出的程序，也会有违背这一原则的代码存在。为什么会出现这种现象呢？因为有职责扩散。**所谓职责扩散，就是因为某种原因，职责 P 被分化为粒度更细的职责 P1 和 P2。**

比如：类 T 只负责一个职责 P，这样设计是符合单一职责原则的。后来由于某种原因，也许是需求变更了，也许是程序的设计者境界提高了，需要将职责 P 细分为粒度更细的职责 P1，P2，这时如果要使程序遵循单一职责原则，需要将类 T 也分解为两个类 T1 和 T2，分别负责 P1、P2 两个职责。但是在程序已经写好的情况下，这样做简直

太费时间了。所以，简单的修改类 T，用它来负责两个职责是一个比较不错的选择，虽然这样做有悖于单一职责原则。（这样做的风险在于职责扩散的不确定性，因为我们不会想到这个职责 P，在未来可能会扩散为 P1，P2，P3，P4……Pn。所以记住，在职责扩散到我们无法控制的程度之前，立刻对代码进行重构。）

举例说明，用一个类描述动物呼吸这个场景：

```
class Animal{
    public void breathe(String animal){
        System.out.println(animal+"呼吸空气");
    }
}
public class Client{
    public static void main(String[] args){
        Animal animal = new Animal();
        animal.breathe("牛");
        animal.breathe("羊");
        animal.breathe("猪");
    }
}
```

运行结果：

牛呼吸空气

羊呼吸空气

猪呼吸空气

程序上线后，发现问题了，并不是所有的动物都呼吸空气的，比如鱼就是呼吸水的。修改时如果遵循单一职责原则，需要将 Animal 类细分为陆生动物类 Terrestrial，水生动物 Aquatic，代码如下：

```
class Terrestrial{
    public void breathe(String animal){
        System.out.println(animal+"呼吸空气");
    }
}
class Aquatic{
    public void breathe(String animal){
        System.out.println(animal+"呼吸水");
    }
}

public class Client{
    public static void main(String[] args){
        Terrestrial terrestrial = new Terrestrial();
        terrestrial.breathe("牛");
        terrestrial.breathe("羊");
        terrestrial.breathe("猪");

        Aquatic aquatic = new Aquatic();
        aquatic.breathe("鱼");
    }
}
```

运行结果：

牛呼吸空气

羊呼吸空气

猪呼吸空气

鱼呼吸水

我们会发现如果这样修改花销是很大的，除了将原来的类分解之外，还需要修改客户端。而直接修改类 `Animal` 来达成目的虽然违背了单一职责原则，但花销却小的多，代码如下：

```

class Animal{
    public void breathe(String animal){
        if("鱼".equals(animal)){
            System.out.println(animal+"呼吸水");
        }else{
            System.out.println(animal+"呼吸空气");
        }
    }
}

public class Client{
    public static void main(String[] args){
        Animal animal = new Animal();
        animal.breathe("牛");
        animal.breathe("羊");
        animal.breathe("猪");
        animal.breathe("鱼");
    }
}

```

可以看到，这种修改方式要简单的多。但是却存在着隐患：有一天需要将鱼分为呼吸淡水的鱼和呼吸海水的鱼，则又需要修改 Animal 类的 breathe 方法，而对原有代码的修改会对调用“猪”“牛”“羊”等相关功能带来风险，也许某一天你会发现程序运行的结果变为“牛呼吸水”了。这种修改方式直接在代码级别上违背了单一职责原则，虽然修改起来最简单，但隐患却是最大的。还有一种修改方式：

```
class Animal{
    public void breathe(String animal){
        System.out.println(animal+"呼吸空气");
    }

    public void breathe2(String animal){
        System.out.println(animal+"呼吸水");
    }
}

public class Client{
    public static void main(String[] args){
        Animal animal = new Animal();
        animal.breathe("牛");
        animal.breathe("羊");
        animal.breathe("猪");
        animal.breathe2("鱼");
    }
}
```

可以看到，这种修改方式没有改动原来的方法，而是在类中新加了一个方法，这样虽然也违背了单一职责原则，但在方法级别上却是符合单一职责原则的，因为它并没有动原来方法的代码。这三种方式各有优缺点，那么在实际编程中，采用哪一种呢？其实这真的比较难说，需要根据实际情况来确定。我的原则是：只有逻辑足够简单，才可以在代码级别上违反单一职责原则；只有类中方法数量足够少，才可以在方法级别上违反单一职责原则；

例如本文所举的这个例子，它太简单了，它只有一个方法，所以，无论是在代码级别上违反单一职责原则，还是在方法级别上违反，都不会造成太大的影响。实际应用中的类都要复杂的多，一旦发生职责扩散而需要修改类时，除非这个类本身非常简单，否则还是遵循单一职责原则的好。

遵循单一职责原则的优点

1、可以降低类的复杂度，一个类只负责一项职责，其逻辑肯定要比负责多项职责简单的多；

2、提高类的可读性，提高系统的可维护性；

3、变更引起的风险降低，变更是必然的，如果单一职责原则遵守的好，当修改一个功能时，可以显著降低对其他功能的影响。

需要说明的一点是单一职责原则不只是面向对象编程思想所特有的，只要是模块化的程序设计，都适用单一职责原则。

2、里氏替换原则

肯定有不少人跟我刚看到这项原则的时候一样，对这个原则的名字充满疑惑。其实原因就是这项原则最早是在 1988 年，由麻省理工学院的一位姓里的女士（Barbara Liskov）提出来的。

定义 1

如果对每一个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2，使得以 T1 定义的所有程序 P 在所有的对象 o1 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。

定义 2

所有引用基类的地方必须能透明地使用其子类的对象。

问题由来：有一功能 P1，由类 A 完成。现需要将功能 P1 进行扩

展，扩展后的功能为 P，其中 P 由原有功能 P1 与新功能 P2 组成。新功能 P 由类 A 的子类 B 来完成，则子类 B 在完成新功能 P2 的同时，有可能会产生原有功能 P1 发生故障。

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

继承包含这样一层含义：父类中凡是已经实现好的方法（相对于抽象方法而言），实际上是在设定一系列的规范和契约，虽然它不强制要求所有的子类必须遵从这些契约，但是如果子类对这些非抽象方法任意修改，就会对整个继承体系造成破坏。而里氏替换原则就是表达了这一层含义。

继承作为面向对象三大特性之一，在给程序设计带来巨大便利的同时，也带来了弊端。比如使用继承会给程序带来侵入性，程序的可移植性降低，增加了对象间的耦合性，如果一个类被其他的类所继承，则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，所有涉及到子类的功能都有可能产生故障。

举例说明继承的风险，我们需要完成一个两数相减的功能，由类 A 来负责。

```
class A{
    public int func1(int a, int b){
        return a-b;
    }
}

public class Client{
    public static void main(String[] args){
        A a = new A();
        System.out.println("100-50="+a.func1(100, 50));
        System.out.println("100-80="+a.func1(100, 80));
    }
}
```

运行结果：

100-50=50

100-80=20

后来，我们需要增加一个新的功能：完成两数相加，然后再与 100 求和，由类 B 来负责。即类 B 需要完成两个功能：

两数相减。

两数相加，然后再加 100。

由于类 A 已经实现了第一个功能，所以类 B 继承类 A 后，只需要再完成第二个功能就可以了，代码如下：


```

class B extends A{
    public int func1(int a, int b){
        return a+b;
    }

    public int func2(int a, int b){
        return func1(a,b)+100;
    }
}

public class Client{
    public static void main(String[] args){
        B b = new B();
        System.out.println("100-50="+b.func1(100, 50));
        System.out.println("100-80="+b.func1(100, 80));
        System.out.println("100+20+100="+b.func2(100, 20));
    }
}

```

类 B 完成后，运行结果：

100-50=150

100-80=180

100+20+100=220

我们发现原本运行正常的相减功能发生了错误。原因就是类 B 在给方法起名时无意中重写了父类的方法，造成所有运行相减功能的代码全部调用了类 B 重写后的方法，造成原本运行正常的功能出现了错误。在本例中，引用基类 A 完成的功能，换成子类 B 之后，发生了异常。在实际编程中，我们常常会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，程序运行出错的几率非常大。如果非要重写父类的方法，比较通用的做法是：原来的父类和子类都继承一个更通

俗的基类，原有的继承关系去掉，采用依赖、聚合，组合等关系代替。

里氏替换原则通俗的来讲就是：**子类可以扩展父类的功能，但不能改变父类原有的功能**。它包含以下 4 层含义：

1、子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。

2、子类中可以增加自己特有的方法。

3、当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。

4、当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

看上去很不可思议，因为我们会发现在自己编程中常常会违反里氏替换原则，程序照样跑的好好的。所以大家都会产生这样的疑问，假如我非要遵循里氏替换原则会有什么后果？

后果就是：**你写的代码出问题的几率将会大大增加。**

3、依赖倒置原则

定义

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

问题由来

类 A 直接依赖类 B，假如要将类 A 改为依赖类 C，则必须通过修

改类 A 的代码来达成。这种场景下，类 A 一般是高层模块，负责复杂的业务逻辑；类 B 和类 C 是低层模块，负责基本的原子操作；假如修改类 A，会给程序带来不必要的风险。

解决方案

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

依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。在 java 中，抽象指的是接口或者抽象类，细节就是具体的实现类，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他们的实现类去完成。

依赖倒置原则的核心思想是**面向接口编程**，我们依旧用一个例子来说明面向接口编程比相对于面向实现编程好在什么地方。场景是这样的，母亲给孩子讲故事，只要给她一本书，她就可以照着书给孩子讲故事了。代码如下：

```

class Book{
    public String getContent(){
        return "很久很久以前有一个阿拉伯的故事.....";
    }
}

class Mother{
    public void narrate(Book book){
        System.out.println("妈妈开始讲故事");
        System.out.println(book.getContent());
    }
}

public class Client{
    public static void main(String[] args){
        Mother mother = new Mother();
        mother.narrate(new Book());
    }
}

```

运行结果:

妈妈开始讲故事

很久很久以前有一个阿拉伯的故事……

运行良好，假如有一天，需求变成这样：不是给书而是给一份报纸，让这位母亲讲一下报纸上的故事，报纸的代码如下：

```

class Newspaper{
    public String getContent(){
        return "林书豪38+7领导尼克斯击败湖人.....";
    }
}

```

这位母亲却办不到，因为她居然不会读报纸上的故事，这太荒唐了，只是将书换成报纸，居然必须要修改 Mother 才能读。假如以后需求换成杂志呢？换成网页呢？还要不断地修改 Mother，这显然不是好的设计。原因就是 Mother 与 Book 之间的耦合性太高了，必须

降低他们之间的耦合度才行。

我们引入一个抽象的接口 IReader。读物，只要是带字的都属于读物：

```
interface IReader{  
    public String getContent();  
}
```

Mother 类与接口 IReader 发生依赖关系，而 Book 和 Newspaper 都属于读物的范畴，他们各自都去实现 IReader 接口，这样就符合依赖倒置原则了，代码修改为：

```
class Newspaper implements IReader {  
    public String getContent(){  
        return "林书豪17+9助尼克斯击败老鹰.....";  
    }  
}  
  
class Book implements IReader{  
    public String getContent(){  
        return "很久很久以前有一个阿拉伯的故事.....";  
    }  
}  
  
class Mother{  
    public void narrate(IReader reader){  
        System.out.println("妈妈开始讲故事");  
        System.out.println(reader.getContent());  
    }  
}  
  
public class Client{  
    public static void main(String[] args){  
        Mother mother = new Mother();  
        mother.narrate(new Book());  
        mother.narrate(new Newspaper());  
    }  
}
```

运行结果：

妈妈开始讲故事

很久很久以前有一个阿拉伯的故事……

妈妈开始讲故事

林书豪 17+9 助尼克斯击败老鹰……

这样修改后，无论以后怎样扩展 Client 类，都不需要再修改 Mother 类了。这只是一个简单的例子，实际情况中，代表高层模块的 Mother 类将负责完成主要的业务逻辑，一旦需要对它进行修改，引入错误的风险极大。所以遵循依赖倒置原则可以降低类之间的耦合性，提高系统的稳定性，降低修改程序造成的风险。

采用依赖倒置原则给多人并行开发带来了极大的便利，比如上例中，原本 Mother 类与 Book 类直接耦合时，Mother 类必须等 Book 类编码完成后才可以进行编码，因为 Mother 类依赖于 Book 类。修改后的程序则可以同时开工，互不影响，因为 Mother 与 Book 类一点关系也没有。参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。现在很流行的 TDD 开发模式就是依赖倒置原则最成功的应用。

传递依赖关系有三种方式，以上的例子中使用的方法是接口传递，另外还有两种传递方式：构造方法传递和 setter 方法传递，相信用过 Spring 框架的，对依赖的传递方式一定不会陌生。

在实际编程中，我们一般需要做到如下 3 点：

- 低层模块尽量都要有抽象类或接口，或者两者都有。
- 变量的声明类型尽量是抽象类或接口。
- 使用继承时遵循里氏替换原则。

依赖倒置原则的核心就是要我们面向接口编程，理解了面向接口编程，也就理解了依赖倒置。

4、接口隔离原则

定义

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

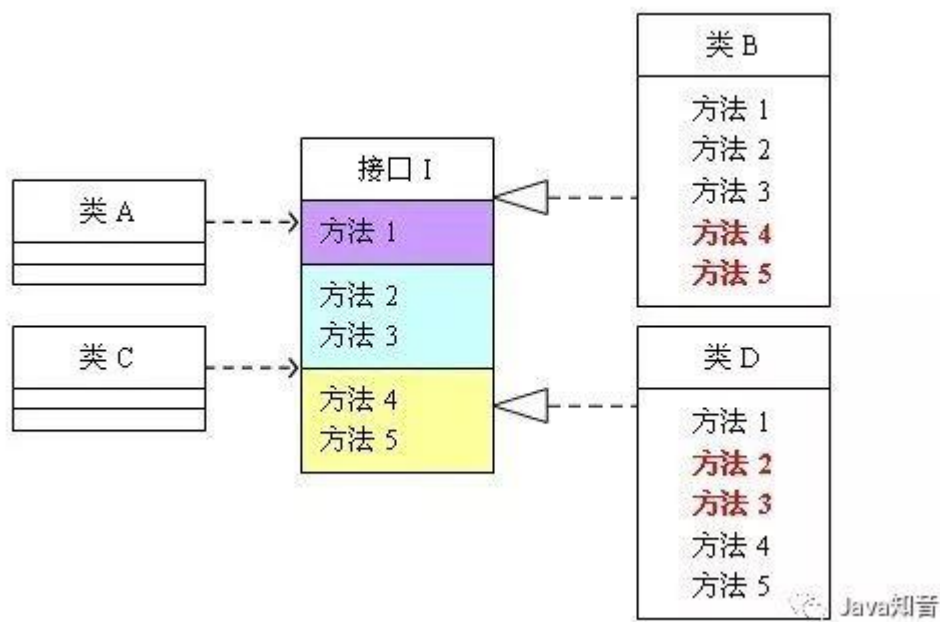
问题由来

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

解决方案

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

举例来说明接口隔离原则：



这个图的意思是：类 A 依赖接口 I 中的方法 1、方法 2、方法 3，类 B 是对类 A 依赖的实现。类 C 依赖接口 I 中的方法 1、方法 4、方法 5，类 D 是对类 C 依赖的实现。对于类 B 和类 D 来说，虽然他们都存在着用不到的方法（也就是图中红色字体标记的方法），但由于实现了接口 I，所以也必须要实现这些用不到的方法。对类图不熟悉的可以参照程序代码来理解，代码如下：

```
interface I {  
    public void method1();  
    public void method2();  
    public void method3();  
    public void method4();  
    public void method5();  
}
```



```
class A{
    public void depend1(I i){
        i.method1();
    }
    public void depend2(I i){
        i.method2();
    }
    public void depend3(I i){
        i.method3();
    }
}

class B implements I{
    public void method1() {
        System.out.println("类B实现接口I的方法1");
    }
    public void method2() {
        System.out.println("类B实现接口I的方法2");
    }
    public void method3() {
        System.out.println("类B实现接口I的方法3");
    }
    //对于类B来说，method4和method5不是必需的，但是由于接口A中有这两个方法，
    //所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进行实现。
    public void method4() {}
    public void method5() {}
}

class C{
    public void depend1(I i){
        i.method1();
    }
    public void depend2(I i){
        i.method4();
    }
    public void depend3(I i){
        i.method5();
    }
}
```

```

class D implements I{
    public void method1() {
        System.out.println("类D实现接口I的方法1");
    }
    //对于类D来说，method2和method3不是必需的，但是由于接口A中有这两个方法，
    //所以在实现过程中即使这两个方法的方法体为空，也要将这两个没有作用的方法进行实现。
    public void method2() {}
    public void method3() {}

    public void method4() {
        System.out.println("类D实现接口I的方法4");
    }
    public void method5() {
        System.out.println("类D实现接口I的方法5");
    }
}

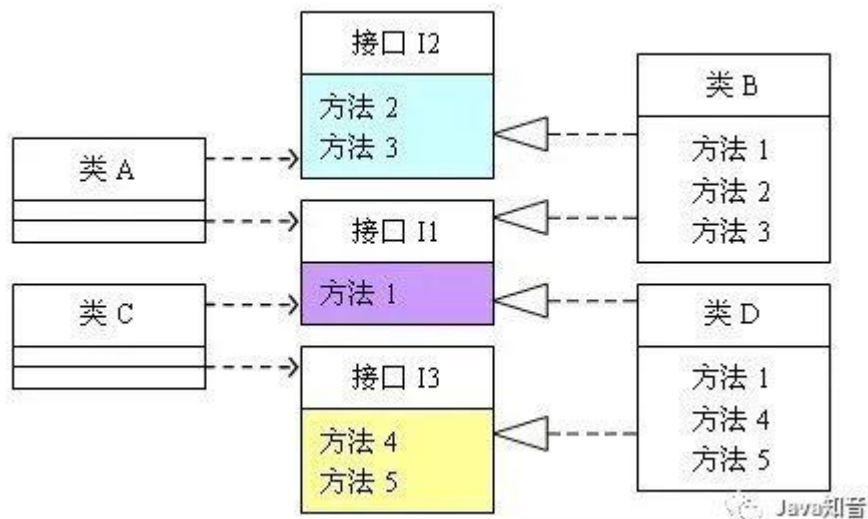
public class Client{
    public static void main(String[] args){
        A a = new A();
        a.depend1(new B());
        a.depend2(new B());
        a.depend3(new B());

        C c = new C();
        c.depend1(new D());
        c.depend2(new D());
        c.depend3(new D());
    }
}

```

可以看到，如果接口过于臃肿，只要接口中出现的方法，不管对依赖于它的类有没有用处，实现类中都必须去实现这些方法，这显然不是好的设计。如果将这个设计修改为符合接口隔离原则，就必须对接口 I 进行拆分。

在这里我们将原有的接口 I 拆分为三个接口，拆分后的设计如图 2 所示：



照例贴出程序的代码，供不熟悉类图的朋友参考：

```
interface I1 {  
    public void method1();  
}  
  
interface I2 {  
    public void method2();  
    public void method3();  
}  
  
interface I3 {  
    public void method4();  
    public void method5();  
}  
  
class A{  
    public void depend1(I1 i){  
        i.method1();  
    }  
    public void depend2(I2 i){  
        i.method2();  
    }  
    public void depend3(I2 i){  
        i.method3();  
    }  
}
```

```
class B implements I1, I2{
    public void method1() {
        System.out.println("类B实现接口I1的方法1");
    }
    public void method2() {
        System.out.println("类B实现接口I2的方法2");
    }
    public void method3() {
        System.out.println("类B实现接口I2的方法3");
    }
}

class C{
    public void depend1(I1 i){
        i.method1();
    }
    public void depend2(I3 i){
        i.method4();
    }
    public void depend3(I3 i){
        i.method5();
    }
}

class D implements I1, I3{
    public void method1() {
        System.out.println("类D实现接口I1的方法1");
    }
    public void method4() {
        System.out.println("类D实现接口I3的方法4");
    }
    public void method5() {
        System.out.println("类D实现接口I3的方法5");
    }
}
```

接口隔离原则的含义是：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，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。

本文例子中，将一个庞大的接口变更为 3 个专用的接口所采用的就是接口隔离原则。在程序设计中，依赖几个专用的接口要比依赖一个综合的接口更灵活。接口是设计时对外部设定的“契约”，通过分散定义多个接口，可以预防外来变更的扩散，提高系统的灵活性和可维护性。

说到这里，很多人会觉得的接口隔离原则跟之前的单一职责原则很相似，其实不然。其一，单一职责原则原注重的是职责；而接口隔离原则注重对接口依赖的隔离。其二，单一职责原则主要是约束类，其次才是接口和方法，它针对的是程序中的实现和细节；而接口隔离原则主要约束接口接口，主要针对抽象，针对程序整体框架的构建。

采用接口隔离原则对接口进行约束时，要注意以下几点：

1、接口尽量小，但是要有限度。对接口进行细化可以提高程序设计灵活性是不争的事实，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。

2、为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。

3、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。

4、运用接口隔离原则，一定要适度，接口设计的过大或过小都不好。设计接口的时候，只有多花些时间去思考和筹划，才能准确地

实践这一原则。

5、迪米特法则

定义

一个对象应该对其他对象保持最少的了解。

问题由来

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

解决方案

尽量降低类与类之间的耦合。

自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

迪米特法则又叫最少知道原则，最早是在 1987 年由美国 Northeastern University 的 Ian Holland 提出。通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部，对外除了提供的 public 方法，不对外泄漏任何信息。迪米特法则还有一个更简单的定义：**只与直接的朋友通信**。首先来解释一下什么是直接的朋友：

每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖、关联、组合、聚合等。其中，我们称出现成员变量、方法参数、方法返回值中的类为**直接的朋友**，而出现在局部变量中的类则不是直接的朋友。也就是说，陌生的类最好不要作为局部变量的形式出现在类的内部。

举一个例子：有一个集团公司，下属单位有分公司和直属部门，现在要求打印出所有下属单位的员工 ID。先来看一下违反迪米特法则的设计。

```
//总公司员工
class Employee{
    private String id;
    public void setId(String id){
        this.id = id;
    }
    public String getId(){
        return id;
    }
}

//分公司员工
class SubEmployee{
    private String id;
    public void setId(String id){
        this.id = id;
    }
    public String getId(){
        return id;
    }
}
```

```
class SubCompanyManager{
    public List<SubEmployee> getAllEmployee() {
        List<SubEmployee> list = new ArrayList<SubEmployee>();
        for(int i=0; i<100; i++){
            SubEmployee emp = new SubEmployee();
            //为分公司人员按顺序分配一个ID
            emp.setId("分公司"+i);
            list.add(emp);
        }
        return list;
    }
}

class CompanyManager{

    public List<Employee> getAllEmployee() {
        List<Employee> list = new ArrayList<Employee>();
        for(int i=0; i<30; i++){
            Employee emp = new Employee();
            //为总公司人员按顺序分配一个ID
            emp.setId("总公司"+i);
            list.add(emp);
        }
        return list;
    }

    public void printAllEmployee(SubCompanyManager sub){
        List<SubEmployee> list1 = sub.getAllEmployee();
        for(SubEmployee e:list1){
            System.out.println(e.getId());
        }

        List<Employee> list2 = this.getAllEmployee();
        for(Employee e:list2){
            System.out.println(e.getId());
        }
    }
}
```



```
public class Client{  
    public static void main(String[] args){  
        CompanyManager e = new CompanyManager();  
        e.printAllEmployee(new SubCompanyManager());  
    }  
}
```

现在这个设计的主要问题出在 CompanyManager 中，根据迪米特法则，只与直接的朋友发生通信，而 SubEmployee 类并不是 CompanyManager 类的直接朋友（以局部变量出现的耦合不属于直接朋友），从逻辑上讲总公司只与他的分公司耦合就行了，与分公司的员工并没有任何联系，这样设计显然是增加了不必要的耦合。按照迪米特法则，应该避免类中出现这样非直接朋友关系的耦合。修改后的代码如下：

```

class SubCompanyManager{
    public List<SubEmployee> getAllEmployee(){
        List<SubEmployee> list = new ArrayList<SubEmployee>();
        for(int i=0; i<100; i++){
            SubEmployee emp = new SubEmployee();
            //为分公司人员按顺序分配一个ID
            emp.setId("分公司"+i);
            list.add(emp);
        }
        return list;
    }
    public void printEmployee(){
        List<SubEmployee> list = this.getAllEmployee();
        for(SubEmployee e:list){
            System.out.println(e.getId());
        }
    }
}

class CompanyManager{
    public List<Employee> getAllEmployee(){
        List<Employee> list = new ArrayList<Employee>();
        for(int i=0; i<30; i++){
            Employee emp = new Employee();
            //为总公司人员按顺序分配一个ID
            emp.setId("总公司"+i);
            list.add(emp);
        }
        return list;
    }

    public void printAllEmployee(SubCompanyManager sub){
        sub.printEmployee();
        List<Employee> list2 = this.getAllEmployee();
        for(Employee e:list2){
            System.out.println(e.getId());
        }
    }
}

```

修改后，为分公司增加了打印人员 ID 的方法，总公司直接调用
来打印，从而避免了与分公司的员工发生耦合。

迪米特法则的初衷是降低类之间的耦合，由于每个类都减少了不必要的依赖，因此的确可以降低耦合关系。但是凡事都有度，虽然可以避免与非直接的类通信，但是要通信，必然会通过一个“中介”来发生联系，例如本例中，总公司就是通过分公司这个“中介”来与分公司的员工发生联系的。过分的使用迪米特原则，会产生大量这样的中介和传递类，导致系统复杂度变大。所以在采用迪米特法则时要反复权衡，既做到结构清晰，又要高内聚低耦合。

6、开闭原则

定义

一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

问题由来

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

解决方案

当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化。

开闭原则是面向对象设计中最基础的设计原则，它指导我们如何建立稳定灵活的系统。开闭原则可能是设计模式六项原则中定义最模

糊的一个了，它只告诉我们对扩展开放，对修改关闭，可是到底如何才能做到对扩展开放，对修改关闭，并没有明确的告诉我们。以前，如果有人告诉我“你进行设计的时候一定要遵守开闭原则”，我会觉的他什么都没说，但貌似又什么都说了。因为开闭原则真的太虚了。

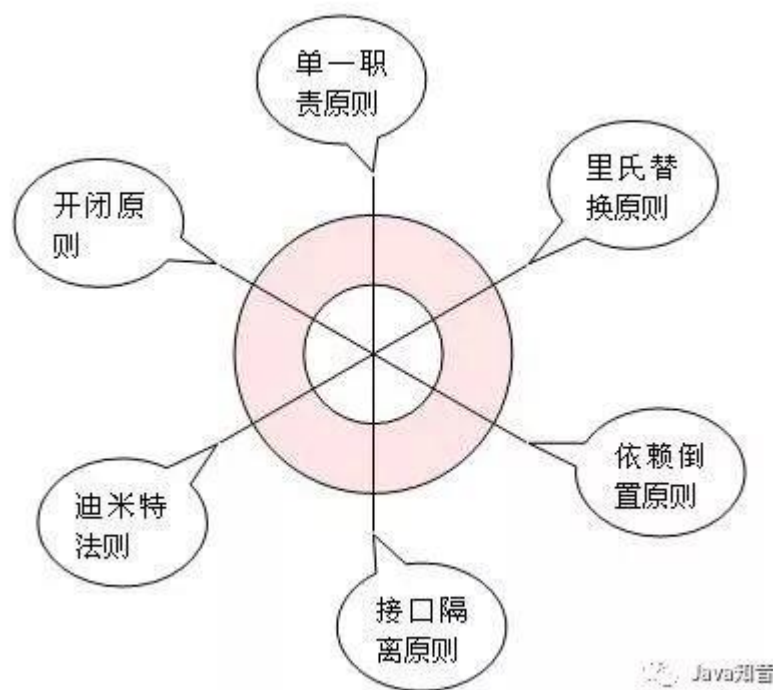
在仔细思考以及仔细阅读很多设计模式的文章后，终于对开闭原则有了一点认识。其实，我们遵循设计模式前面 5 大原则，以及使用 23 种设计模式的目的就是遵循开闭原则。也就是说，只要我们对前面 5 项原则遵守的好了，设计出的软件自然是符合开闭原则的，这个开闭原则更像是前面五项原则遵守程度的“平均得分”，前面 5 项原则遵守的好，平均分自然就高，说明软件设计开闭原则遵守的好；如果前面 5 项原则遵守的不好，则说明开闭原则遵守的不好。

其实笔者认为，开闭原则无非就是想表达这样一层意思：**用抽象构建框架，用实现扩展细节**。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，我们用从抽象派生的实现类来进行扩展，当软件需要发生变化时，我们只需要根据需求重新派生一个实现类来扩展就可以了。当然前提是我们的抽象要合理，要对需求的变更有前瞻性和预见性才行。

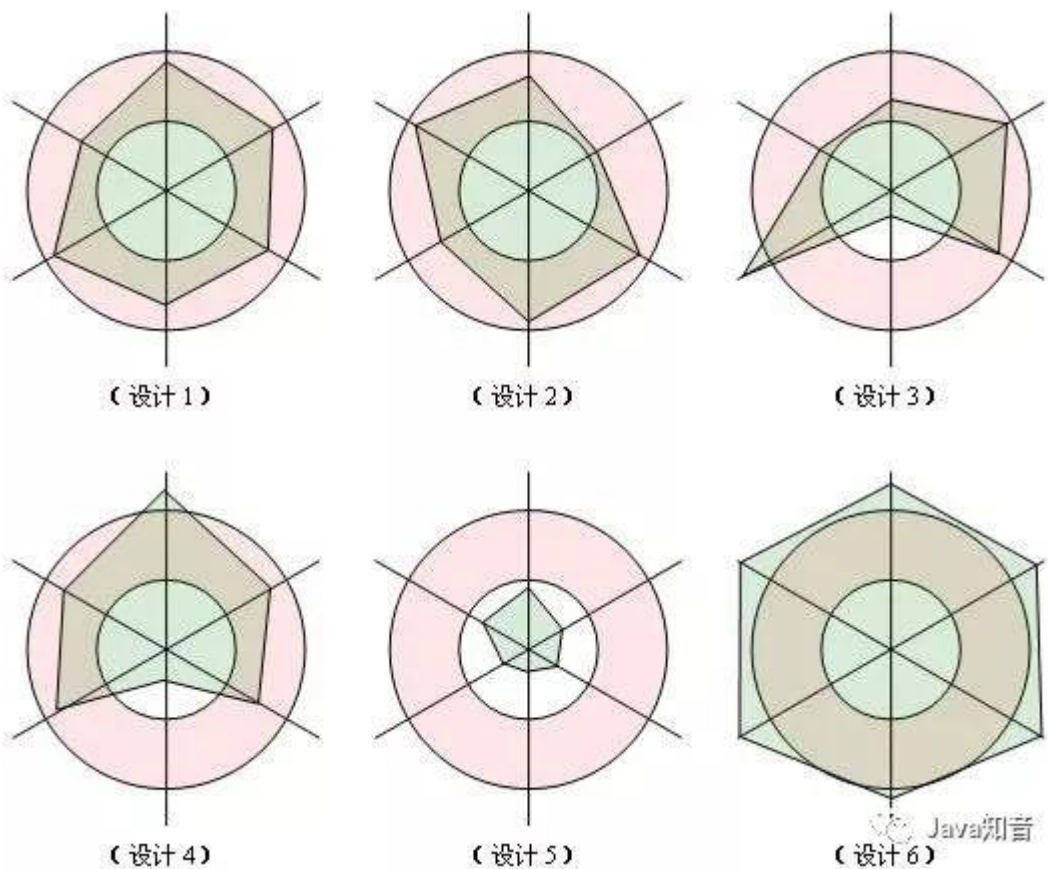
说到这里，再回想一下前面说的 5 项原则，恰恰是告诉我们**用抽象构建框架，用实现扩展细节**的注意事项而已：单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦合。而开闭原

则是总纲，他告诉我们要对扩展开放，对修改关闭。

最后说明一下如何去遵守这六个原则。对这六个原则的遵守并不是是与否的问题，而是多和少的问题，也就是说，我们一般会说有没有遵守，而是说遵守程度的多少。任何事都是过犹不及，设计模式的六个设计原则也是一样，制定这六个原则的目的并不是要我们刻板的遵守他们，而需要根据实际情况灵活运用。对他们的遵守程度只要在一个合理的范围内，就算是良好的设计。我们用一幅图来说明一下。



图中的每一条维度各代表一项原则，我们依据对这项原则的遵守程度在维度上画一个点，则如果对这项原则遵守的合理的话，这个点应该落在红色的同心圆内部；如果遵守的差，点将会在小圆内部；如果过度遵守，点将会落在大圆外部。一个良好的设计体现在图中，应该是六个顶点都在同心圆中的六边形。



在上图中，设计 1、设计 2 属于良好的设计，他们对六项原则的遵守程度都在合理的范围内；设计 3、设计 4 设计虽然有些不足，但也基本可以接受；设计 5 则严重不足，对各项原则都没有很好的遵守；而设计 6 则遵守过度了，设计 5 和设计 6 都是迫切需要重构的设计。