第二版：精选设计模式 10 道

目录

我们的网站：https://tech.souyunku.com

关注我们的公众号：搜云库技术团队，回复以下关键字

回复:【进群】邀请您进「技术架构分享群」

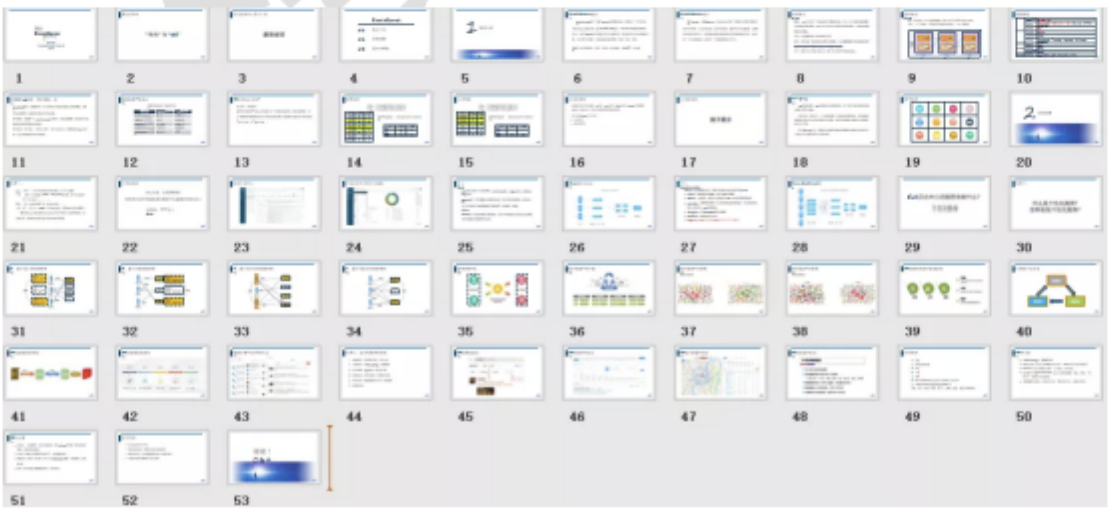
回复:【内推】即可进：北京，上海，广周，深圳，杭州，成都，武汉，南京，郑州，西安，长沙「程序员工作内推群」

回复 【1024】送 4000G 最新架构师视频

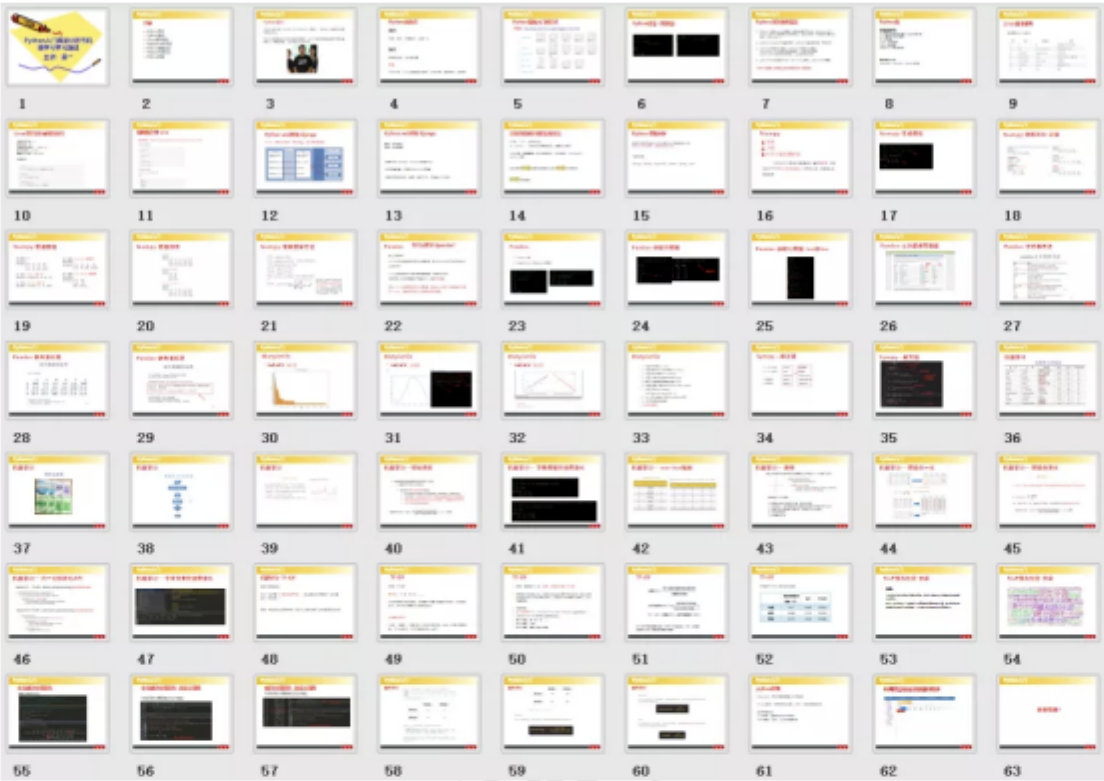
回复 【PPT】即可无套路获取，以下最新整理调优 PPT!

46 页《JVM 深度调优，演讲 PPT》

53 页《Elasticsearch 调优演讲 PPT》



63 页《Python 数据分析入门 PPT》



什么是设计模式

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

为什么要学习设计模式

- 看懂源代码：如果你不懂设计模式去看 Jdk、Spring、SpringMVC、IO 等等等等的源码，你会很迷茫，你会寸步难行
- 看看前辈的代码：你去个公司难道都是新项目让你接手？很有可能是接盘的，前辈的开发难道不用设计模式？
- 编写自己的理想中的好代码：我个人反正是这样的，对于我自己开发的项目我会很认真，我对他比对我女朋友还好，把项目当成自己的儿子一样

设计模式分类

- 创建型模式，共五种：**工厂方法模式、抽象工厂模式、单例模式**、建造者模式、**原型模式。**
- 结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
- 行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

设计模式的六大原则

开放封闭原则（Open Close Principle）

- 原则思想：尽量通过扩展软件实体来解决需求变化，而不是通过修改已有的代码来完成变化
- 描述：一个软件产品在生命周期内，都会发生变化，既然变化是一个既定的事实，我们就应该在设计的时候尽量适应这些变化，以提高项目的稳定性和灵活性。
- 优点：单一原则告诉我们，每个类都有自己负责的职责，里氏替换原则不能破坏继承关系的体系。

里氏代换原则（Liskov Substitution Principle）

- 原则思想：使用的基类可以在任何地方使用继承的子类，完美的替换基类。
- 大概意思是：子类可以扩展父类的功能，但不能改变父类原有的功能。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，子类中可以增加自己特有的方法。
- 优点：增加程序的健壮性，即使增加了子类，原有的子类还可以继续运行，互不影响。

依赖倒转原则（Dependence Inversion Principle）

- 依赖倒置原则的核心思想是面向接口编程.
- 依赖倒转原则要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，

- 这个是开放封闭原则的基础，具体内容是：对接口编程，依赖于抽象而不依赖于具体。

接口隔离原则（Interface Segregation Principle）

- 这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。
- 例如：支付类的接口和订单类的接口，需要把这俩个类别的接口变成俩个隔离的接口

迪米特法则（最少知道原则）（Demeter Principle）

- 原则思想：一个对象应当对其他对象有尽可能少地了解，简称类间解耦
- 大概意思就是一个类尽量减少自己对其他对象的依赖，原则是低耦合，高内聚，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。
- 优点：低耦合，高内聚。

单一职责原则（Principle of single responsibility）

- 原则思想：一个方法只负责一件事情。
- 描述：单一职责原则很简单，一个方法 一个类只负责一个职责，各个职责的程序改动，不影响其它程序。 这是常识，几乎所有程序员都会遵循这个原则。
- 优点：降低类和类的耦合，提高可读性，增加可维护性和可拓展性，降低可变性的风险。

单例模式

1.什么是单例

- 保证一个类只有一个实例，并且提供一个访问该全局访问点

2.那些地方用到了单例模式

1、 网站的计数器，一般也是采用单例模式实现，否则难以同步。

2、 应用程序的日志应用，一般都是单例模式实现，只有一个实例去操作才好，否则内容不好追加显示。

3、 多线程的线程池的设计一般也是采用单例模式，因为线程池要方便对池中的线程进行控制

4、 Windows 的（任务管理器）就是很典型的单例模式，他不能打开俩个

5、 windows 的（回收站）也是典型的单例应用。在整个系统运行过程中，回收站只维护一个实例。

3.单例优缺点

优点：

1、 在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就防止其它对象对自己的实例化，确保所有的对象都访问一个实例

2、 单例模式具有一定的伸缩性，类自己来控制实例化进程，类就在改变实例化进程上有相应的伸缩性。

3、 提供了对唯一实例的受控访问。

4、 由于在系统内存中只存在一个对象，因此可以节约系统资源，当需要频繁创建和销毁的对象时单例模式无疑可以提高系统的性能。

5、 允许可变数目的实例。

6、 避免对共享资源的多重占用。

缺点：

1、 不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。

2、 由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。

3、 单例类的职责过重，在一定程度上违背了“单一职责原则”。

4、 滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

4.单例模式使用注意事项：

1、 使用时不能用反射模式创建单例，否则会实例化一个新的对象

2、 使用懒单例模式时注意线程安全问题

3、 饿单例模式和懒单例模式构造方法都是私有的，因而是不能被继承的，有些单例模式可以被继承（如登记式模式）

5.单例防止反射漏洞攻击

```
private static boolean flag = false;

private Singleton() {
```

```
        if (flag == false) {
            flag = !flag;
        } else {
            throw new RuntimeException("单例模式被侵犯！");
        }
    }

    public static void main(String[] args) {

    }
```

6.如何选择单例创建方式

- 如果不需要延迟加载单例，可以使用枚举或者饿汉式，相对来说枚举性好于饿汉式。 如果需要延迟加载，可以使用静态内部类或者懒汉式，相对来说静态内部类好于懒韩式。 最好使用饿汉式

7.单例创建方式

（主要使用懒汉和懒汉式）

1、 饿汉式:类初始化时,会立即加载该对象，线程天生安全,调用效率高。

2、 懒汉式：类初始化时,不会初始化该对象,真正需要使用的时候才会创建该对象,具备懒加载功能。

3、 静态内部方式:结合了懒汉式和饿汉式各自的优点，真正需要对象的时候才会加载，加载类是线程安全的。

4、 枚举单例：使用枚举实现单例模式 优点:实现简单、调用效率高，枚举本身就是单例，由 jvm 从根本上提供保障!避免通过反射和反序列化的漏洞， 缺点没有延迟加载。

5、 双重检测锁方式（因为 JVM 本质重排序的原因，可能会初始化多次，不推荐使用）

1.饿汉式

1、 饿汉式:类初始化时,会立即加载该对象，线程天生安全,调用效率高。

```
package com.lijie;

//饿汉式
public class Demo1 {

    // 类初始化时,会立即加载该对象，线程安全,调用效率高
    private static Demo1 demo1 = new Demo1();

    private Demo1() {
        System.out.println("私有 Demo1 构造参数初始化");
    }

    public static Demo1 getInstance() {
        return demo1;
    }

    public static void main(String[] args) {
        Demo1 s1 = Demo1.getInstance();
        Demo1 s2 = Demo1.getInstance();
        System.out.println(s1 == s2);
    }
```

```
}
```

2.懒汉式

1、 懒汉式：类初始化时,不会初始化该对象,真正需要使用的时候才会创建该对象,具备懒加载功能。

```
package com.lijie;

//懒汉式
public class Demo2 {

    //类初始化时，不会初始化该对象，真正需要使用的时候才会创建该对象。
    private static Demo2 demo2;

    private Demo2() {
        System.out.println("私有 Demo2 构造参数初始化");
    }

    public synchronized static Demo2 getInstance() {
        if (demo2 == null) {
            demo2 = new Demo2();
        }
        return demo2;
    }

    public static void main(String[] args) {
        Demo2 s1 = Demo2.getInstance();
        Demo2 s2 = Demo2.getInstance();
        System.out.println(s1 == s2);
    }
```

```
}
```

3.静态内部类

1、 静态内部方式:结合了懒汉式和饿汉式各自的优点，真正需要对象的时候才会加载，加载类是线程安全的。

```
package com.lijie;

// 静态内部类方式
public class Demo3 {

    private Demo3() {
        System.out.println("私有 Demo3 构造参数初始化");
    }

    public static class SingletonClassInstance {
        private static final Demo3 DEMO_3 = new Demo3();
    }

    // 方法没有同步
    public static Demo3 getInstance() {
        return SingletonClassInstance.DEMO_3;
    }

    public static void main(String[] args) {
        Demo3 s1 = Demo3.getInstance();
        Demo3 s2 = Demo3.getInstance();
        System.out.println(s1 == s2);
    }
}
```

4.枚举单例式

1、 枚举单例：使用枚举实现单例模式 优点:实现简单、调用效率高，枚举本身就是单例，由 jvm 从根本上提供保障!避免通过反射和反序列化的漏洞， 缺点没有延迟加载。

```
package com.lijie;

//使用枚举实现单例模式 优点:实现简单、枚举本身就是单例，由 jvm 从根本上提供保障!避免通过反射和反序列化的漏洞 缺点没有延迟加载
public class Demo4 {

    public static Demo4 getInstance() {
        return Demo.INSTANCE.getInstance();
    }

    public static void main(String[] args) {
        Demo4 s1 = Demo4.getInstance();
        Demo4 s2 = Demo4.getInstance();
        System.out.println(s1 == s2);
    }

    //定义枚举
    private static enum Demo {
        INSTANCE;
        // 枚举元素为单例
        private Demo4 demo4;

        private Demo() {
            System.out.println("枚举 Demo 私有构造参数");
```

```
            demo4 = new Demo4();
        }

        public Demo4 getInstance() {
            return demo4;
        }
    }
}
```

5.双重检测锁方式

1、 双重检测锁方式（因为 JVM 本质重排序的原因，可能会初始化多次，不推荐使用）

```
package com.lijie;

//双重检测锁方式
public class Demo5 {

    private static Demo5 demo5;

    private Demo5() {
        System.out.println("私有 Demo4 构造参数初始化");
    }

    public static Demo5 getInstance() {
        if (demo5 == null) {
            synchronized (Demo5.class) {
                if (demo5 == null) {
                    demo5 = new Demo5();
                }
```

```
            }
        }
        return demo5;
    }

    public static void main(String[] args) {
        Demo5 s1 = Demo5.getInstance();
        Demo5 s2 = Demo5.getInstance();
        System.out.println(s1 == s2);
    }
}
```

工厂模式

1.什么是工厂模式

- 它提供了一种创建对象的最佳方式。在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。实现了创建者和调用者分离，工厂模式分为简单工厂、工厂方法、抽象工厂模式

2.工厂模式好处

- 工厂模式是我们最常用的实例化对象模式了，是用工厂方法代替 new 操作的一种模式。
- 利用工厂模式可以降低程序的耦合性，为后期的维护修改提供了很大的便利。
- 将选择实现类、创建对象统一管理和控制。从而将调用者跟我们的实现类解耦。

3.为什么要学习工厂设计模式

- 不知道你们面试题问到过源码没有，你知道 Spring 的源码吗，MyBatis 的源码吗，等等等 如果你想学习很多框架的源码，或者你想自己开发自己的框架，就必须先掌握设计模式（工厂设计模式用的是非常非常广泛的）

4.Spring 开发中的工厂设计模式

1、Spring IOC

- 看过 Spring 源码就知道，在 Spring IOC 容器创建 bean 的过程是使用了工厂设计模式
- Spring 中无论是通过 xml 配置还是通过配置类还是注解进行创建 bean，大部分都是通过简单工厂来进行创建的。
- 当容器拿到了 beanName 和 class 类型后，动态的通过反射创建具体的某个对象，最后将创建的对象放到 Map 中。

2、为什么 Spring IOC 要使用工厂设计模式创建 Bean 呢

- 在实际开发中，如果我们 A 对象调用 B，B 调用 C，C 调用 D 的话我们程序的耦合性就会变高。（耦合大致分为类与类之间的依赖，方法与方法之间的依赖。）
- 在很久以前的三层架构编程时，都是控制层调用业务层，业务层调用数据访问层时，都是是直接 new 对象，耦合性大大提升，代码重复量很高，对象满天飞
- 为了避免这种情况，Spring 使用工厂模式编程，写一个工厂，由工厂创建 Bean，以后我们如果要对象就直接管工厂要就可以，剩下的事情不归我们管了。Spring

IOC 容器的工厂中有个静态的 Map 集合，是为了让工厂符合单例设计模式，即每个对象只生产一次，生产出对象后就存入到 Map 集合中，保证了实例不会重复影响程序效率。

5.工厂模式分类

- 工厂模式分为简单工厂、工厂方法、抽象工厂模式

> 简单工厂 ： 用来生产同一等级结构中的任意产品。（不支持拓展增加产品）
> 工厂方法 ： 用来生产同一等级结构中的固定产品。（支持拓展增加产品）
> 抽象工厂 ： 用来生产不同产品族的全部产品。（不支持拓展增加产品；支持增加产品族）

我下面来使用代码演示一下：

5.1 简单工厂模式

什么是简单工厂模式

- 简单工厂模式相当于是一个工厂中有各种产品，创建在一个类中，客户无需知道具体产品的名称，只需要知道产品类所对应的参数即可。但是工厂的职责过重，而且当类型过多时不利于系统的扩展维护。

代码演示：

1、 创建工厂

```
package com.lijie;

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

1、 创建工厂的产品（宝马）

```
package com.lijie;

public class Bmw implements Car {
    public void run() {
        System.out.println("我是宝马汽车...");
    }
}
```

1、 创建工另外一种产品（奥迪）

```
package com.lijie;

public class AoDi implements Car {
    public void run() {
        System.out.println("我是奥迪汽车..");
    }
}
```

1、 创建核心工厂类，由他决定具体调用哪产品

```
package com.lijie;

public class CarFactory {
```

```
    public static Car createCar(String name) {
        if ("".equals(name)) {
            return null;
        }
        if(name.equals("奥迪")){
            return new AoDi();
        }
        if(name.equals("宝马")){
            return new Bmw();
        }
        return null;
    }
}
```

1、 演示创建工厂的具体实例

```
package com.lijie;

public class Client01 {

    public static void main(String[] args) {
        Car aodi =CarFactory.createCar("奥迪");
        Car bmw =CarFactory.createCar("宝马");
        aodi.run();
        bmw.run();
    }
}
```

单工厂的优点/缺点

- 优点：简单工厂模式能够根据外界给定的信息，决定究竟应该创建哪个具体类的对象。明确区分了各自的职责和权力，有利于整个软件体系结构的优化。
- 缺点：很明显工厂类集中了所有实例的创建逻辑，容易违反 GRASPR 的高内聚的责任分配原则

5.2 工厂方法模式

什么是工厂方法模式

- 工厂方法模式 Factory Method，又称多态性工厂模式。在工厂方法模式中，核心的工厂类不再负责所有的产品的创建，而是将具体创建的工作交给子类去做。该核心类成为一个抽象工厂角色，仅负责给出具体工厂子类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节

代码演示：

1、 创建工厂

```
package com.lijie;

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

1、 创建工厂方法调用接口（所有的产品需要 new 出来必须继承他来实现方法）

```
package com.lijie;

public interface CarFactory {
```

```
    Car createCar();

}
```

1、 创建工厂的产品（奥迪）

```
package com.lijie;

public class AoDi implements Car {
    public void run() {
        System.out.println("我是奥迪汽车..");
    }
}
```

1、 创建工厂另外一种产品（宝马）

```
package com.lijie;

public class Bmw implements Car {
    public void run() {
        System.out.println("我是宝马汽车...");
    }
}
```

1、 创建工厂方法调用接口的实例（奥迪）

```
package com.lijie;

public class AoDiFactory implements CarFactory {
```

```
    public Car createCar() {

        return new AoDi();
    }
}
```

1、 创建工厂方法调用接口的实例（宝马）

```
package com.lijie;

public class BmwFactory implements CarFactory {

    public Car createCar() {

        return new Bmw();
    }

}
```

1、 演示创建工厂的具体实例

```
package com.lijie;

public class Client {

    public static void main(String[] args) {
        Car aodi = new AoDiFactory().createCar();
        Car jili = new BmwFactory().createCar();
        aodi.run();
        jili.run();
    }
```

```
}
```

5.3 抽象工厂模式

什么是抽象工厂模式

- 抽象工厂简单地说是工厂的工厂，抽象工厂可以创建具体工厂，由具体工厂来产生具体产品。 **代码演示：**

1、 创建第一个子工厂，及实现类

```
package com.lijie;

//汽车
public interface Car {
        void run();
}

 class CarA implements Car{

    public void run() {
        System.out.println("宝马");
    }

}
 class CarB implements Car{

    public void run() {
        System.out.println("摩拜");
    }
```

```
}
```

1、 创建第二个子工厂，及实现类

```
package com.lijie;

//发动机
public interface Engine {

    void run();

}

class EngineA implements Engine {

    public void run() {
        System.out.println("转的快!");
    }

}

class EngineB implements Engine {

    public void run() {
        System.out.println("转的慢!");
    }

}
```

1、 创建一个总工厂，及实现类（由总工厂的实现类决定调用那个工厂的那个实例）

```
package com.lijie;

public interface TotalFactory {
    // 创建汽车
    Car createChair();
    // 创建发动机
    Engine createEngine();
}

//总工厂实现类，由他决定调用哪个工厂的那个实例
class TotalFactoryReally implements TotalFactory {

    public Engine createEngine() {

        return new EngineA();
    }

    public Car createChair() {

        return new CarA();
    }
}
```

1、 运行测试

```
package com.lijie;

public class Test {
```

```
    public static void main(String[] args) {
        TotalFactory totalFactory2 = new TotalFactoryReally();
        Car car = totalFactory2.createChair();
        car.run();

        TotalFactory totalFactory = new TotalFactoryReally();
        Engine engine = totalFactory.createEngine();
        engine.run();
    }
}
```

代理模式

1.什么是代理模式

- 通过代理控制对象的访问，可以在这个对象调用方法之前、调用方法之后去处理/添加新的功能。(也就是 AO 的 P 微实现)
- 代理在原有代码乃至原业务流程都不修改的情况下，直接在业务流程中切入新代码，增加新功能，这也和 Spring 的（面向切面编程）很相似

2.代理模式应用场景

- Spring AOP、日志打印、异常处理、事务控制、权限控制等

3.代理的分类

- 静态代理(静态定义代理类)
- 动态代理(动态生成代理类，也称为 Jdk 自带动态代理)
- Cglib 、javaassist（字节码操作库）

4.三种代理的区别

1、 静态代理：简单代理模式，是动态代理的理论基础。常见使用在代理模式

2、 jdk 动态代理：使用反射完成代理。需要有顶层接口才能使用，常见是 mybatis 的 mapper 文件是代理。

3、 cglib 动态代理：也是使用反射完成代理，可以直接代理类（jdk 动态代理不行），使用字节码技术，不能对 final 类进行继承。（需要导入 jar 包）

5.用代码演示三种代理

5.1.静态代理

什么是静态代理

- 由程序员创建或工具生成代理类的源码，再编译代理类。所谓静态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。

代码演示：

- 我有一段这样的代码：（如何能在不修改 UserDao 接口类的情况下开事务和关闭事务呢）

```
package com.lijie;

//接口类
public class UserDao{
    public void save() {
        System.out.println("保存数据方法");
    }
}
```

```
package com.lijie;

//运行测试类
public   class Test{
    public static void main(String[] args) {
        UserDao userDao = new UserDao();
        userDao.save();
    }
}
```

修改代码，添加代理类

```
package com.lijie;

//代理类
public class UserDaoProxy extends UserDao {
    private UserDao userDao;
```

```
    public UserDaoProxy(UserDao userDao) {
        this.userDao = userDao;
    }

    public void save() {
        System.out.println("开启事物...");
        userDao.save();
        System.out.println("关闭事物...");
    }

}
```

```
//添加完静态代理的测试类
public class Test{
    public static void main(String[] args) {
        UserDao userDao = new UserDao();
        UserDaoProxy userDaoProxy = new UserDaoProxy(userDao);
        userDaoProxy.save();
    }
}
```

- 缺点：每个需要代理的对象都需要自己重复编写代理，很不舒服，
- 优点：但是可以面相实际对象或者是接口的方式实现代理

2.2.动态代理

什么是动态代理

- 动态代理也叫做，JDK 代理、接口代理。

- 动态代理的对象，是利用 JDK 的 API，动态的在内存中构建代理对象（是根据被代理的接口来动态生成代理类的 class 文件，并加载运行的过程），这就叫动态代理

```
package com.lijie;

//接口
public interface UserDao {
    void save();
}
```

```
package com.lijie;

//接口实现类
public class UserDaoImpl implements UserDao {
    public void save() {
        System.out.println("保存数据方法");
    }
}
```

- //下面是代理类，可重复使用，不像静态代理那样要自己重复编写代理

```
package com.lijie;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

// 每次生成动态代理类对象时,实现了 InvocationHandler 接口的调用处理器对象
```

```
public class InvocationHandlerImpl implements InvocationHandler {

    // 这其实业务实现类对象，用来调用具体的业务方法
    private Object target;

    // 通过构造函数传入目标对象
    public InvocationHandlerImpl(Object target) {
        this.target = target;
    }

    //动态代理实际运行的代理方法
    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
        System.out.println("调用开始处理");
        //下面 invoke()方法是以反射的方式来创建对象，第一个参数是要创建
的对象，第二个是构成方法的参数，由第二个参数来决定创建对象使用哪个构造
方法
        Object result = method.invoke(target, args);
        System.out.println("调用结束处理");
        return result;
    }
}
```

- //利用动态代理使用代理方法

```
package com.lijie;

import java.lang.reflect.Proxy;

public class Test {
```

```
    public static void main(String[] args) {
        // 被代理对象
        UserDao userDaoImpl = new UserDaoImpl();
        InvocationHandlerImpl invocationHandlerImpl = new
InvocationHandlerImpl(userDaoImpl);

        //类加载器
        ClassLoader loader = userDaoImpl.getClass().getClassLoader();
        Class<?>[] interfaces = userDaoImpl.getClass().getInterfaces();

        // 主要装载器、一组接口及调用处理动态代理实例
        UserDao newProxyInstance = (UserDao)
Proxy.newProxyInstance(loader, interfaces, invocationHandlerImpl);
        newProxyInstance.save();
    }
}
```

- 缺点：必须是面向接口，目标业务类必须实现接口
- 优点：不用关心代理类，只需要在运行阶段才指定代理哪一个对象

5.3.CGLIB 动态代理

CGLIB 动态代理原理：

- 利用 asm 开源包，对代理对象类的 class 文件加载进来，通过修改其字节码生成子类来处理。

什么是 CGLIB 动态代理

- CGLIB 动态代理和 jdk 代理一样，使用反射完成代理，不同的是他可以直接代理类（jdk 动态代理不行，他必须目标业务类必须实现接口），CGLIB 动态代理底层使用字节码技术，CGLIB 动态代理不能对 final 类进行继承。（CGLIB 动态代理需要导入 jar 包）

代码演示：

```
package com.lijie;

//接口
public interface UserDao {
    void save();
}
```

```
package com.lijie;

//接口实现类
public class UserDaoImpl implements UserDao {
    public void save() {
        System.out.println("保存数据方法");
    }
}
```

```
package com.lijie;

import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;
```

```
//代理主要类
public class CglibProxy implements MethodInterceptor {
    private Object targetObject;
    // 这里的目标类型为 Object，则可以接受任意一种参数作为被代理类，实现了动态代理
    public Object getInstance(Object target) {
        // 设置需要创建子类的类
        this.targetObject = target;
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(target.getClass());
        enhancer.setCallback(this);
        return enhancer.create();
    }

    //代理实际方法
    public Object intercept(Object obj, Method method, Object[] args,
MethodProxy proxy) throws Throwable {
        System.out.println("开启事物");
        Object result = proxy.invoke(targetObject, args);
        System.out.println("关闭事物");
        // 返回代理对象
        return result;
    }
}
```

```
package com.lijie;

//测试 CGLIB 动态代理
public class Test {
    public static void main(String[] args) {
        CglibProxy cglibProxy = new CglibProxy();
```

```
        UserDao userDao = (UserDao) cglibProxy.getInstance(new
UserDaoImpl());
        userDao.save();
    }
}
```

建造者模式

1.什么是建造者模式

- 建造者模式：是将一个复杂的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方式进行创建。
- 工厂类模式是提供的是创建单个类的产品
- 而建造者模式则是将各种产品集中起来进行管理，用来具有不同的属性的产品

建造者模式通常包括下面几个角色：

1、 uilder：给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。
2、 ConcreteBuilder：实现 Builder 接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。 在建造过程完成后，提供产品的实例。
3、 Director：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。
4、 Product：要创建的复杂对象。

2.建造者模式的使用场景

使用场景：

1、 需要生成的对象具有复杂的内部结构。

2、 需要生成的对象内部属性本身相互依赖。

- 与工厂模式的区别是：建造者模式更加关注与零件装配的顺序。
- JAVA 中的 StringBuilder 就是建造者模式创建的，他把一个单个字符的 char 数组组合起来
- Spring 不是建造者模式，它提供的操作应该是对于字符串本身的一些操作，而不是创建或改变一个字符串。

3.代码案例

1、 建立一个装备对象 Arms

```
package com.lijie;

//装备类
public class Arms {
    //头盔
    private String helmet;
    //铠甲
    private String armor;
    //武器
    private String weapon;

    //省略 Git 和 Set 方法.....
}
```

1、 创建 Builder 接口（给出一个抽象接口，以规范产品对象的各个组成成分的建造，这个接口只是规范）

```
package com.lijie;

public interface PersonBuilder {

    void builderHelmetMurder();

    void builderArmorMurder();

    void builderWeaponMurder();

    void builderHelmetYanLong();

    void builderArmorYanLong();

    void builderWeaponYanLong();

    Arms BuilderArms(); //组装
}
```

1、 创建 Builder 实现类（这个类主要实现复杂对象创建的哪些部分需要什么属性）

```
package com.lijie;

public class ArmsBuilder implements PersonBuilder {
    private Arms arms;

    //创建一个 Arms 实例,用于调用 set 方法
```

```
public ArmsBuilder() {
    arms = new Arms();
}

public void builderHelmetMurder() {
    arms.setHelmet("夺命头盔");
}

public void builderArmorMurder() {
    arms.setArmor("夺命铠甲");
}

public void builderWeaponMurder() {
    arms.setWeapon("夺命宝刀");
}

public void builderHelmetYanLong() {
    arms.setHelmet("炎龙头盔");
}

public void builderArmorYanLong() {
    arms.setArmor("炎龙铠甲");
}

public void builderWeaponYanLong() {
    arms.setWeapon("炎龙宝刀");
}

public Arms BuilderArms() {
    return arms;
}
```

```
}
```

1、 Director（调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建）

```
package com.lijie;

public class PersonDirector {

    //组装
    public Arms constructPerson(PersonBuilder pb) {
        pb.builderHelmetYanLong();
        pb.builderArmorMurder();
        pb.builderWeaponMurder();
        return pb.BuilderArms();
    }

    //这里进行测试
    public static void main(String[] args) {
        PersonDirector pb = new PersonDirector();
        Arms arms = pb.constructPerson(new ArmsBuilder());
        System.out.println(arms.getHelmet());
        System.out.println(arms.getArmor());
        System.out.println(arms.getWeapon());
    }
}
```

模板方法模式

1.什么是模板方法

- 模板方法模式：定义一个操作中的算法骨架（父类），而将一些步骤延迟到子类中。 模板方法使得子类可以不改变一个算法的结构来重定义该算法的

2.什么时候使用模板方法

- 实现一些操作时，整体步骤很固定，但是呢。就是其中一小部分需要改变，这时候可以使用模板方法模式，将容易变的部分抽象出来，供子类实现。

3.实际开发中应用场景哪里用到了模板方法

- 其实很多框架中都有用到了模板方法模式
- 例如：数据库访问的封装、Junit 单元测试、servlet 中关于 doGet/doPost 方法的调用等等

4.现实生活中的模板方法

例如：

1、 去餐厅吃饭，餐厅给我们提供了一个模板就是：看菜单，点菜，吃饭，付款，走人 （这里 “**点菜和付款**” 是不确定的由子类来完成的，其他的则是一个模板。）

5.代码实现模板方法模式

1、 先定义一个模板。把模板中的点菜和付款，让子类来实现。

```
package com.lijie;

//模板方法
public abstract class RestaurantTemplate {

    // 1.看菜单
    public void menu() {
        System.out.println("看菜单");
    }

    // 2.点菜业务
    abstract void spotMenu();

    // 3.吃饭业务
    public void havingDinner(){ System.out.println("吃饭"); }

    // 3.付款业务
    abstract void payment();

    // 3.走人
    public void GoR() { System.out.println("走人"); }

    //模板通用结构
    public void process(){
        menu();
        spotMenu();
```

```
        havingDinner();
        payment();
        GoR();
    }
}
```

1、 具体的模板方法子类 1

```
package com.lijie;

public class RestaurantGinsengImpl extends RestaurantTemplate {

    void spotMenu() {
        System.out.println("人参");
    }

    void payment() {
        System.out.println("5 快");
    }
}
```

1、 具体的模板方法子类 2

```
package com.lijie;

public class RestaurantLobsterImpl  extends RestaurantTemplate  {

    void spotMenu() {
        System.out.println("龙虾");
    }
```

```
    void payment() {
        System.out.println("50 块");
    }
}
```

1、 客户端测试

```
package com.lijie;

public class Client {

    public static void main(String[] args) {
        //调用第一个模板实例
        RestaurantTemplate restaurantTemplate = new
RestaurantGinsengImpl();
        restaurantTemplate.process();
    }
}
```

外观模式

1.什么是外观模式

- 外观模式：也叫门面模式，隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。
- 它向现有的系统添加一个接口，用这一个接口来隐藏实际的系统的复杂性。
- 使用外观模式，他外部看起来就是一个接口，其实他的内部有很多复杂的接口已经被实现

2.外观模式例子

- 用户注册完之后，需要调用阿里短信接口、邮件接口、微信推送接口。

1、 创建阿里短信接口

```
package com.lijie;

//阿里短信消息
public interface AliSmsService {
    void sendSms();
}
```

```
package com.lijie;

public class AliSmsServiceImpl implements AliSmsService {

    public void sendSms() {
        System.out.println("阿里短信消息");
    }

}
```

1、 创建邮件接口

```
package com.lijie;

//发送邮件消息
```

```
public interface EamilSmsService {
    void sendSms();
}
```

```
package com.lijie;

public class EamilSmsServiceImpl implements    EamilSmsService{
    public void sendSms() {
        System.out.println("发送邮件消息");
    }
}
```

1、 创建微信推送接口

```
package com.lijie;

//微信消息推送
public interface WeiXinSmsService {
    void sendSms();
}
```

```
package com.lijie;

public class WeiXinSmsServiceImpl implements   WeiXinSmsService {
    public void sendSms() {
        System.out.println("发送微信消息推送");
    }
}
```

1、 创建门面（门面看起来很简单使用，复杂的东西以及被门面给封装好了）

```
package com.lijie;

public class Computer {
    AliSmsService aliSmsService;
    EamilSmsService eamilSmsService;
    WeiXinSmsService weiXinSmsService;

    public Computer() {
        aliSmsService = new AliSmsServiceImpl();
        eamilSmsService = new EamilSmsServiceImpl();
        weiXinSmsService = new WeiXinSmsServiceImpl();
    }

    //只需要调用它
    public void sendMsg() {
        aliSmsService.sendSms();
        eamilSmsService.sendSms();
        weiXinSmsService.sendSms();
    }
}
```

1、 启动测试

```
package com.lijie;

public class Client {

    public static void main(String[] args) {
        //普通模式需要这样
        AliSmsService aliSmsService = new AliSmsServiceImpl();
        EamilSmsService eamilSmsService = new EamilSmsServiceImpl();
```

```
        WeiXinSmsService weiXinSmsService = new
WeiXinSmsServiceImpl();
        aliSmsService.sendSms();
        eamilSmsService.sendSms();
        weiXinSmsService.sendSms();

        //利用外观模式简化方法
        new Computer().sendMsg();
    }
}
```

原型模式

1.什么是原型模式

- 原型设计模式简单来说就是克隆
- 原型表明了有一个样板实例，这个原型是可定制的。原型模式多用于创建复杂的或者构造耗时的实例，因为这种情况下，复制一个已经存在的实例可使程序运行更高效。

2.原型模式的应用场景

1、 类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。这时我们就可以通过原型拷贝避免这些消耗。

2、 通过 new 产生的一个对象需要非常繁琐的数据准备或者权限，这时可以使用原型模式。

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

我们 Spring 框架中的多例就是使用原型。

3.原型模式的使用方式

1、 实现 Cloneable 接口。在 java 语言有一个 Cloneable 接口，它的作用只有一个，就是在运行时通知虚拟机可以安全地在实现了此接口的类上使用 clone 方法。在 java 虚拟机中，只有实现了这个接口的类才可以被拷贝，否则在运行时会抛出 CloneNotSupportedException 异常。

2、 重写 Object 类中的 clone 方法。Java 中，所有类的父类都是 Object 类，Object 类中有一个 clone 方法，作用是返回对象的一个拷贝，但是其作用域 protected 类型的，一般的类无法调用，因此 Prototype 类需要将 clone 方法的作用域修改为 public 类型。

3.1 原型模式分为浅复制和深复制

1、 （浅复制）只是拷贝了基本类型的数据，而引用类型数据，只是拷贝了一份引用地址。

2、 （深复制）在计算机中开辟了一块新的内存地址用于存放复制的对象。

4.代码演示

1、 创建 User 类

```
package com.lijie;

import java.util.ArrayList;
```

```
public class User implements Cloneable {
    private String name;
    private String password;
    private ArrayList<String> phones;

    protected User clone() {
        try {
            User user = (User) super.clone();
            //重点，如果要连带引用类型一起复制，需要添加底下一条代码，如果不加就对于是复制了引用地址
            user.phones = (ArrayList<String>) this.phones.clone();//设置深复制
            return user;
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return null;
    }

    //省略所有属性 Git Set 方法.....
}
```

1、 测试复制

```
package com.lijie;

import java.util.ArrayList;

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

```
        //创建 User 原型对象
        User user = new User();
        user.setName("李三");
        user.setPassword("123456");
        ArrayList<String> phones = new ArrayList<>();
        phones.add("17674553302");
        user.setPhones(phones);

        //copy 一个 user 对象,并且对象的属性
        User user2 = user.clone();
        user2.setPassword("654321");

        //查看俩个对象是否是一个
        System.out.println(user == user2);

        //查看属性内容
        System.out.println(user.getName() + " | " + user2.getName());
        System.out.println(user.getPassword() + " | " +
user2.getPassword());
        //查看对于引用类型拷贝
        System.out.println(user.getPhones() == user2.getPhones());
    }
}
```

1、 如果不需要深复制，需要删除 User 中的

```
//默认引用类型为浅复制，这是设置了深复制
user.phones = (ArrayList<String>) this.phones.clone();
```

策略模式

1.什么是策略模式

- 定义了一系列的算法 或 逻辑 或 相同意义的操作，并将每一个算法、逻辑、操作封装起来，而且使它们还可以相互替换。（其实策略模式 Java 中用的非常非常广泛）
- 我觉得主要是为了 简化 if...else 所带来的复杂和难以维护。

2.策略模式应用场景

- 策略模式的用意是针对一组算法或逻辑，将每一个算法或逻辑封装到具有共同接口的独立的类中，从而使得它们之间可以相互替换。

1、 例如：我要做一个不同会员打折力度不同的三种策略，初级会员，中级会员，高级会员（三种不同的计算）。

2、 例如：我要一个支付模块，我要有微信支付、支付宝支付、银联支付等

3.策略模式的优点和缺点

- 优点： 1、算法可以自由切换。 2、避免使用多重条件判断。 3、扩展性非常良好。
- 缺点： 1、策略类会增多。 2、所有策略类都需要对外暴露。

4.代码演示

- 模拟支付模块有微信支付、支付宝支付、银联支付

1、 定义抽象的公共方法

```
package com.lijie;

//策略模式 定义抽象方法 所有支持公共接口
abstract class PayStrategy {

    // 支付逻辑方法
    abstract void algorithmInterface();

}
```

1、 定义实现微信支付

```
package com.lijie;

class PayStrategyA extends PayStrategy {

    void algorithmInterface() {
        System.out.println("微信支付");
    }
}
```

1、 定义实现支付宝支付

```
package com.lijie;
```

```
class PayStrategyB extends PayStrategy {

    void algorithmInterface() {
        System.out.println("支付宝支付");
    }
}
```

1、 定义实现银联支付

```
package com.lijie;

class PayStrategyC extends PayStrategy {

    void algorithmInterface() {
        System.out.println("银联支付");
    }
}
```

1、 定义下文维护算法策略

```
package com.lijie;// 使用上下文维护算法策略

class Context {

    PayStrategy strategy;

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

    public void algorithmInterface() {
```

```
            strategy.algorithmInterface();
        }

}
```

1、 运行测试

```
package com.lijie;

class ClientTestStrategy {
    public static void main(String[] args) {
        Context context;
        //使用支付逻辑 A
        context = new Context(new PayStrategyA());
        context.algorithmInterface();
        //使用支付逻辑 B
        context = new Context(new PayStrategyB());
        context.algorithmInterface();
        //使用支付逻辑 C
        context = new Context(new PayStrategyC());
        context.algorithmInterface();
    }
}
```

观察者模式

1.什么是观察者模式

- 先讲什么是行为性模型，行为型模式关注的是系统中对象之间的相互交互，解决系统在运行时对象之间的相互通信和协作，进一步明确对象的职责。
- 观察者模式，是一种行为性模型，又叫发布-订阅模式，他定义对象之间一种一对多的依赖关系，使得当一个对象改变状态，则所有依赖于它的对象都会得到通知并自动更新。

2.模式的职责

- 观察者模式主要用于 1 对 N 的通知。当一个对象的状态变化时，他需要及时告知一系列对象，令他们做出相应。

实现有两种方式：

1、 推：每次都会把通知以广播的方式发送给所有观察者，所有的观察者只能被动接收。

2、 拉：观察者只要知道有情况即可，至于什么时候获取内容，获取什么内容，都可以自主决定。

3.观察者模式应用场景

1、 关联行为场景，需要注意的是，关联行为是可拆分的，而不是“组合”关系。事件多级触发场景。

2、 跨系统的消息交换场景，如消息队列、事件总线的处理机制。

4.代码实现观察者模式

1、 定义抽象观察者，每一个实现该接口的实现类都是具体观察者。

```
package com.lijie;

//观察者的接口，用来存放观察者共有方法
public interface Observer {
    // 观察者方法
    void update(int state);
}
```

1、 定义具体观察者

```
package com.lijie;

// 具体观察者
public class ObserverImpl implements Observer {

    // 具体观察者的属性
    private int myState;

    public void update(int state) {
        myState=state;
        System.out.println("收到消息,myState 值改为：  "+state);
    }

    public int getMyState() {
        return myState;
    }
}
```

1、 定义主题。主题定义观察者数组，并实现增、删及通知操作。

```
package com.lijie;

import java.util.Vector;

//定义主题，以及定义观察者数组，并实现增、删及通知操作。
public class Subjecct {
    //观察者的存储集合，不推荐 ArrayList，线程不安全，
    private Vector<Observer> list = new Vector<>();

    // 注册观察者方法
    public void registerObserver(Observer obs) {
        list.add(obs);
    }
    // 删除观察者方法
    public void removeObserver(Observer obs) {
        list.remove(obs);
    }

    // 通知所有的观察者更新
    public void notifyAllObserver(int state) {
        for (Observer observer : list) {
            observer.update(state);
        }
    }
}
```

1、 定义具体的，他继承继承 Subject 类，在这里实现具体业务，在具体项目中，该类会有很多。

```
package com.lijie;
```

```
//具体主题
public class RealObserver extends Subjecct {
    //被观察对象的属性
     private int state;
     public int getState(){
          return state;
     }
     public void   setState(int state){
          this.state=state;
          //主题对象(目标对象)值发生改变
          this.notifyAllObserver(state);
     }
}
```

1、 运行测试

```
package com.lijie;

public class Client {

    public static void main(String[] args) {
        // 目标对象
        RealObserver subject = new RealObserver();
        // 创建多个观察者
        ObserverImpl obs1 = new ObserverImpl();
        ObserverImpl obs2 = new ObserverImpl();
        ObserverImpl obs3 = new ObserverImpl();
        // 注册到观察队列中
        subject.registerObserver(obs1);
        subject.registerObserver(obs2);
```

```
        subject.registerObserver(obs3);
        // 改变 State 状态
        subject.setState(300);
        System.out.println("obs1 观察者的 MyState 状态值为:
"+obs1.getMyState());
        System.out.println("obs2 观察者的 MyState 状态值为:
"+obs2.getMyState());
        System.out.println("obs3 观察者的 MyState 状态值为:
"+obs3.getMyState());
        // 改变 State 状态
        subject.setState(400);
        System.out.println("obs1 观察者的 MyState 状态值为:
"+obs1.getMyState());
        System.out.println("obs2 观察者的 MyState 状态值为:
"+obs2.getMyState());
        System.out.println("obs3 观察者的 MyState 状态值为:
"+obs3.getMyState());
    }
}
```

文章就到这了，没错，没了

察者方法 public void removeObserver(Observer obs) { list.remove(obs); }

```
// 通知所有的观察者更新
public void notifyAllObserver(int state) {
    for (Observer observer : list) {
        observer.update(state);
    }
}
```

}

4. 定义具体的，他继承继承 Subject 类，在这里实现具体业务，在具体项目中，该类会有很多。

```java
package com.lijie;

//具体主题
public class RealObserver extends Subjecct {
    //被观察对象的属性
     private int state;
     public int getState(){
          return state;
     }
     public void   setState(int state){
          this.state=state;
          //主题对象(目标对象)值发生改变
          this.notifyAllObserver(state);
     }
}
```

1、 运行测试

```
package com.lijie;

public class Client {

    public static void main(String[] args) {
        // 目标对象
        RealObserver subject = new RealObserver();
```

```
        // 创建多个观察者
        ObserverImpl obs1 = new ObserverImpl();
        ObserverImpl obs2 = new ObserverImpl();
        ObserverImpl obs3 = new ObserverImpl();
        // 注册到观察队列中
        subject.registerObserver(obs1);
        subject.registerObserver(obs2);
        subject.registerObserver(obs3);
        // 改变 State 状态
        subject.setState(300);
        System.out.println("obs1 观察者的 MyState 状态值为:
"+obs1.getMyState());
        System.out.println("obs2 观察者的 MyState 状态值为:
"+obs2.getMyState());
        System.out.println("obs3 观察者的 MyState 状态值为:
"+obs3.getMyState());
        // 改变 State 状态
        subject.setState(400);
        System.out.println("obs1 观察者的 MyState 状态值为:
"+obs1.getMyState());
        System.out.println("obs2 观察者的 MyState 状态值为:
"+obs2.getMyState());
        System.out.println("obs3 观察者的 MyState 状态值为:
"+obs3.getMyState());
    }
}
```