设计模式课程安排

第一天 设计原则、工厂、单例模式

第二天 代理模式

第三天 原型、模板、委派、策略、适配器模式以及设计模式总结

设计模式第一天

何为设计模式

在设计模式出来之前，大家虽然是面向对象编程，但是大家都是逻辑从头写到尾！！会出现1个方法，1个类中代码的繁杂冗余，导致可读性很差很差！！然后需要加新功能的时候，会把之前的代码都可能会被改动！！

所以才出来了设计模式！！能不能把一些冗余的代码复用，看起来更加优雅！！同时能不能让 添加新功能的时候，不去改动之前的代码！！

所以设计模式的初衷：

1.代码更加的优雅复用

2.让代码更加易于扩展

但是也会带来一些问题：

比如：加大代码的理解难度、有时间会需要添加很多类，让代码量更大！！

所以，设计模式是有利有弊，大家要根据具体的业务场景来决定需不需要来使用设计模式！！不要强行装X使用不该使用的设计模式！！

那么讲到设计模式，肯定离不开什么时候用设计模式，就有相对于的设计原则，但是设计原则也是有利有弊，不可能完全满足，所以，七大设计原则更加注重的是在思想层面！！！

跟数据库设计的三范式是一样的。虽然有，但是我不一定遵守！

七大设计原则

开闭原则

对扩展开放，对修改关闭；减少对已有代码的修改（比如已有方法、逻辑），提升扩展性；同时减少对已有代码的影响。

可以采用继承、重写等方式。

还可以 需要更多的依赖接口与抽象，不要依赖具体实现！！ 因为你依赖实现，改动只能改动具体实现。

这也是我们的第二个原则：依赖倒置原则

依赖倒置原则

依赖倒置原则（Dependence Inversion Principle,DIP）是指设计代码结构时，抽象不依赖细节，细节应该依赖抽象。

这也是去开闭原则的基础！！

举例

场景：我现在要去旅游，去旅游必须选择交通工具！！比如最开始我选择骑自行车去旅游！！我们现在去实现！

在没用依赖倒置之前的

1.去旅行类

```
public class Tour {
    public String tour(Bicycle  bicycle)
    {
        return bicycle.name()+"去旅游！";
    }
}
```

2.自行车对象

```
public class Bicycle {
    public String name()
    {
        return "自行车";
    }
}
```

3.调用方

```
Bicycle bicycle=new Bicycle();
Tour tour=new Tour();
System.out.println(tour.tour(bicycle));
```

我们发现，我现在是依赖自行车，自行车是个具体的实现，那么假如哪天我想换小汽车，那么必须改动tour类的tour方法，这样就不符合开闭原则，任何改动我都必须改动我不应该改动的点！

所以，改进它！我不再依赖自行车，而是依赖一个抽象！

```
public interface ITraffic {
    String name();
}
```

具体类

```
public class Bicycle implements ITraffic {
    public String name()
    {
        return "自行车";
    }
}
```

实现类不依赖具体

```
public class Tour {
    public String tour(ITraffic  traffic)
    {
        return traffic.name()+"去旅游！";
    }
}
```

调用

```
ITraffic traffic=new Car();
Tour tour=new Tour();
System.out.println(tour.tour(traffic));
```

加新的交通工具，我不需要再更改具体的实现！！

单一职责原则

单一职责（Simple Responsibility Pinciple，SRP）是指不要存在多于一个导致类变更的原因。假设我们有一个 Class
负责两个职责，一旦发生需求变更，修改其中一个职责的逻辑代码，有可能会导致另一个职责的功能发生故障。这样一来，这个 Class 存在两个导致类变更的原因。如何解决这个问题呢？我们就要给两个职责分别用两个 Class
来实现，进行解耦。后期需求变更维护互不影响。这样的设计可以降低类

的复杂度，提高类的可读性，提高系统的可维护性，降低变更引起的风险。总体来说就是一个 Class/Interface/Method 只负责一项职责。

尽量的原子化，但是我们实际情况很多时候是不符合单一职责的，因为会加大代码量！！以及让代码过于分散！

接口隔离原则

接口隔离原则（Interface Segregation Principle, ISP）是指用多个专门的接口，而不使用单一的总接口，客户端不应该依赖它不需要的接口

IAnimal 接口：

```
public interface IAnimal
{
 void eat();
 void fly();
 void swim();
}
```

bird类实现

```
public class Bird implements IAnimal
{
    @Override
    public void eat() {}
    @Override
    public void fly() {}
    @Override
    public void swim() {}
}
```

Dog类实现：

```
public class Dog implements IAnimal
{
    @Override public void eat() {}
    @Override public void fly() {}
    @Override public void swim() {}
}
```

可以看出，Bird 的 swim()方法可能只能空着，Dog 的 fly()方法显然不可能的。所以我们可以分别设计 IEatAnimal，IFlyAnimal 和 ISwimAnimal 接口

迪米特法则

迪米特原则（Law of Demeter LoD）是指一个对象应该对其他对象保持最少的了解，又叫最少知道原则（Least Knowledge Principle,LKP），尽量降低类与类之间的耦合。迪米特原则主要强调只和朋友交流，不和陌生人说话。出现在成员变量、方法的输入、输出参数中的类都可以称之为成员朋友类， 而出现在方法体内部的类不属于朋友类。

实体类之间尽量减少相互作用！！ 简单就是一句话，减少类之间的耦合，这也是为什么我们很多方法定义成private或者protect的原因！

里式替换原则

当使用继承的时候，子类继承父类时，除了新加方法完成新的功能，尽量不要重写父类的方法，也尽量不要重载父类的方法！

里氏替换原则（Liskov Substitution Principle,LSP）

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。

比如：

父类

```
public class Parent {
    public int decr(int a, int b) {
        return a - b;
    }
}
```

子类：

```
public class Children extends Parent{
    public int decr(int a, int b) {
        return a + b;
    }
}
```

但是如果父类是抽象类，则子类可以实现其抽象方法

2、子类中可以增加自己特有的方法。

比如：我在子类可以添加父类没有的方法

```
public class Children extends Parent{
//    public int decr(int a, int b) {
//        return a + b;
//    }

    public int incr(int a,int b)
    {
        return a+b;
    }
}
```

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

举例

父类

```
public class Parent {
    public int decr(int a, int b) {
        return a - b;
    }

    public void fun(HashMap  m)
    {
        System.out.println("执行父类方法！");
    }
}
```

子类

```
public class Children extends Parent{
//    public int decr(int a, int b) {
//        return a + b;
//    }

    public int incr(int a,int b)
    {
        return a+b;
    }

    public void fun(Map m)
    {
        System.out.println("执行子类方法！");
    }
}
```

调用

```
Children children=new Children();
HashMap m=new HashMap();
children.fun(m);
```

我们发现HashMap是Map的实现类，所以子类的前置条件（即方法的输入/入参）要比父类方法的输入参数更宽松！所以调用的是父类的方法。

反之：如果参数互换，则调用的是子类方法，就不符合

4、当子类的方法实现父类的方法时（重写/重载或实现抽象方法），方法的后置条件（即方法的输出/返回值）要比父类更严格或相等。

父类

```
public abstract class AbstractParent {
    public  abstract Map fun1();
}
```

实现类

```
public class Children extends AbstractParent{

    @Override
    public HashMap fun1() {
        System.out.println("执行子类方法！");
        return null;
    }
}
```

合成复用原则

合成复用原则（Composite/Aggregate Reuse Principle,CARP）是指尽量使用对象组合(has-a)/ 聚合(contanis-a)

继承is-a，而不是继承关系达到软件复用的目的。可以使系统更加灵活，降低类与类之间的耦合度，一个类的变化对其他类造成的影响相对较少

is-a　鸟是动物 鸟类继承动物

设计模式分类

设计模式我们一般分为3大类：创建型、结构型、行为型

设计模式的的3大类型其实是一个递进的过程，主要是基于它这个设计模式的目的

首先，你要实现功能，肯定要创建对象是不是，所以像单例、工厂、建造者、原型都属于怎么去很好的创建对象

创建完对象后是不是就应该考虑对象之间怎么合理的存在，如何更好的继承、依赖、组合，那么就衍生了很多结构型的模式、比如门面、适配器、代理、装饰、组合、享元

前面2步做完，就是到了具体的实现了。怎么更好的达到目的，那么为了行为更清晰，效率更高就是行为型牧师，比如我们的委派，后面的策略，责任链、迭代器等等

工厂模式

工厂，顾名思义，创建对象的实例的工厂，属于创建型模式，使用在创建不同的相关联的对象实例的场景，工厂也分为3种，分别为简单工厂、工厂方法、抽象工厂！！

比如，我现在有个需求，生产手机，但是，我可以有2个不同的公司可以去生产。小米与华为，为了满足依赖倒置原则，为了以后加公司的时候减少改动，我们定义一个抽象类

简单工厂模式

抽象手机：

```
public interface IPhone {
    String name();
}
```

具体手机实现Huawei：

```
public class HuaweiPhone implements IPhone {
    @Override
    public String name() {
        return "华为手机";
    }
}
```

具体手机实现Xiaomi:

```
public class XiaomiPhone implements IPhone {
    @Override
    public String name() {
        return "小米手机";
    }
}
```

工厂类: 根据名称来得到相关手机并且实例化

```
public class PhoneFactory {

    public IPhone createPhone(String phoneName) {
        try {
            if (!(null == phoneName ||
"".equals(phoneName))) {
                return (IPhone)
Class.forName(phoneName).newInstance();
            }

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

}
```

工厂方法模式

核心思想：将对象创建的逻辑下推到子类，就是创建多个子工厂

华为工厂：

```
public class HuaweiFactory {
    public IPhone createPhone()
    {
        return new HuaweiPhone();
    }
}
```

小米工厂：

```
public class XiaomiFactory {
    public IPhone createPhone()
    {
        return new XiaomiPhone();
    }
}
```

调用：

```
public class FactoryMethodTest {
    public static void main(String[] args) {
        HuaweiFactory factory=new HuaweiFactory();
        IPhone phone=factory.createPhone();
        System.out.println(phone.name());

        XiaomiFactory xiaomiFactory=new XiaomiFactory();
        IPhone xiaomiPhone=xiaomiFactory.createPhone();
        System.out.println(xiaomiPhone.name());
    }
}
```

抽象工厂模式

刚刚我们讲的只是一个工厂里面去生成手机，但是哪一天我需要再生产电脑，我是不是也是直接可以再我工厂里面添加，这个其实就是产品族的概念!

再添加电脑的抽象实现:

```
public interface IComputer {
     String name();
}
```

电脑具体实现

华为电脑:

```
public class HuaweiComputer implements IComputer{

    @Override
    public String name() {
        return "华为电脑";
    }
}
```

小米电脑:

```
public class XiaomiComputer implements IComputer{

    @Override
    public String name() {
        return "小米电脑";
    }
}
```

抽象工厂，顾名思义，把工厂抽象出去，所以，我们先建立一个抽象工厂，里面有多个产品了

```
public abstract class CompanyFactory {
    abstract IPhone createPhone();

    abstract IComputer createComputer();
}
```

接下来就是具体的工厂实现

```
public class HuaweiFactory extends CompanyFactory {
    public IPhone createPhone()
    {
        return new HuaweiPhone();
    }

    @Override
    IComputer createComputer() {
        return new HuaweiComputer();
    }
}
```

```
public class XiaomiFactory extends CompanyFactory {
    public IPhone createPhone()
    {
        return new XiaomiPhone();
    }

    @Override
    IComputer createComputer() {
        return new XiaomiComputer();
    }
}
```

调用:

```
public class AbstractFactoryTest {
    public static void main(String[] args) {
        CompanyFactory huaweiFactory=new HuaweiFactory();

 System.out.println(huaweiFactory.createPhone().name());
    }
}
```

单例模式

单例模式 任何情况下，确保类只能有1个实例，并提供一个全局访问点 属于创建型模式！！

所以，单例的主要思想就是只提供对外的一个获取实例，其他方法私有化

饿汉式单例 首次加载的时候创建，加上static ,类变量随着类的加载而加载，即使此类还未new过对象，这个类变量也存在，而且仅一份。

饿汉式单例

在类加载的时候立即初始化，并且创建单例对象，绝对的线程安全，在线程还没出现以前就开始实例化了，不存在访问安全的问题

static修饰，跟类一起初始化， final修饰，只能被赋值一次，赋值后不允许再被改变

优点：线程安全，执行效率比较高

缺点：所有对象类加载的时候就实例化，那么可能会有大量的内存浪费，因为不确定这个对象会不会使用

```
public class HungrySingleton {

    private static final HungrySingleton hungrySingleton =
new HungrySingleton();

    private HungrySingleton(){}

    public static HungrySingleton getInstance(){
        return  hungrySingleton;
    }
}
```

静态代码块机制

```
public class HungryStaticSingleton
{
    private static final HungryStaticSingleton
hungrySingleton;

    static {
        hungrySingleton = new HungryStaticSingleton();
    }

    private HungryStaticSingleton(){}

    public static HungryStaticSingleton getInstance(){
        return  hungrySingleton;
    }
}
```

懒汉式单例

双重检查锁单例

为了解决饿汉式单例带来的内存浪费，出来了懒汉单例，只有在对象在使用的时候才会被初始化

```
public class LazySimpleSingletion {
    private static LazySimpleSingletion instance;
    private LazySimpleSingletion(){}

    public static LazySimpleSingletion getInstance(){
        if(instance == null){
            instance = new LazySimpleSingletion();
        }
        return instance;
    }
}
```

有线程安全问题,可以使用synchronized锁，让getInstance变成同步方法，但是不能出现并行访问的情况，严重的拖慢了处理性能，所以我们只锁住代码块，使用双重检查锁

```
public class LazyDoubleCheckSingleton {
    private volatile static LazyDoubleCheckSingleton
instance;
    private LazyDoubleCheckSingleton(){}

    public static LazyDoubleCheckSingleton getInstance(){
        //检查是否要阻塞,如果已经创建过，就不需要再进入加锁代码块拉
低性能，大大的提升性能，如果没有该检查，每次都会去竞争锁
        if (instance == null) {
            synchronized (LazyDoubleCheckSingleton.class)
{
                //检查是否要重新创建实例
                //如果没有该判断，2个线程在第一个if都判断为null,
那么就会创建2个实例
                if (instance == null) {
                    instance = new
LazyDoubleCheckSingleton();
```

```
                }
            }
        }
        return instance;
    }
}
```

为什么要加volatile

new对象在JVM底层的指令

```
public class Test {
    public static void main(String[] args) {
        Children children=new Children();
        HashMap map=new HashMap();
        children.fun(map);
    }
}
```

将java文件编译成class文件

javac AtomicTest.java

然后我们通过javap -v AtomicTest.class 可以看到相关指令会有3条

```
 stack=2, locals=2, args_size=1
         0: new           #2                  // class java/util/HashMap  //创建对象
         3: dup
         4: invokespecial #3                  // Method java/util/HashMap."<init>":()V  //初始化
         7: astore_1    //将map指向实例对象
         8: return
```

底层cpu优化，在不影响最终结果的时候，会进行指令重排，也就是 map 引用会在初始化之前执行。

那么当另外的线程来判断第一个instance == null 不满足条件，但是对象还是一个没有初始化的对象。

所以需要加volatile来禁止指令重排。

静态内部类

还是有一定性能问题，那么我们还可以采用静态内部类的方式

同样也是利用了类的加载机制，它与饿汉模式不同的是，它是在内部类里面去创建对象实例。这样的话，只要应用中不使用内部类，JVM就不会去加载这个单例类，也就不会创建单例对象，从而实现懒汉式的延迟加载。也就是说这种方式可以同时保证延迟加载和线程安全。

```
public class LazyStaticInnerClassSingleton {

    private LazyStaticInnerClassSingleton(){
        //解决反射破坏，因为反射可以调用私有的构造器
        if(LazyHolder.INSTANCE != null){
            throw new RuntimeException("不允许非法访问");
        }
    }

    public static LazyStaticInnerClassSingleton
getInstance(){
        return LazyHolder.INSTANCE;
    }

    private static class LazyHolder{
        private static final LazyStaticInnerClassSingleton
INSTANCE = new LazyStaticInnerClassSingleton();
    }

}
```

注册式单例

枚举式单例模式

```
public enum EnumSingleton {
    INSTANCE;

    private Object data;

    public Object getData() {
        return data;
    }

    public void setData(Object data) {
        this.data = data;
    }

    public static EnumSingleton getInstance(){return
INSTANCE;}
}
```

优点：解决了反射与序列化破坏，但是也在类加载时将所有的对象初始化放在内存中去了，不适合大量创建单例对象的场景

容器式单例

```
private ContainerSingleton(){}

    private static  Map<String,Object> ioc = new HashMap<>
(); //put不会被覆盖

    //有线程安全问题，需要加锁
    public static Object getInstance(String className) {
        Object instance = null;
        if (!ioc.containsKey(className)) {
            synchronized (ContainerSingleton.class) {
                if (!ioc.containsKey(className)) {
                    try {
                        instance =
Class.forName(className).newInstance();
                        ioc.put(className, instance);
```

```
                } catch (Exception e) {
                    e.printStackTrace();
                }
            } else {
                instance = ioc.get(className);
            }
        }
        return instance;
    } else {
        return ioc.get(className);
    }
}
```

单例模式总结

1. 饿汉式单例：在类加载就实例化，线程安全，执行效率比较高，但是不确定是否需要使用，造成内存浪费
2. 懒汉式单例：第一次调用进行初始化、线程不安全，需要通过synchronized或双重检查锁来实现线程安全
3. 反射破坏单例原理：虽然构造器设置为私有，但是可以通过设置强制访问来调用其构造函数，具体为：`c.setAccessible(true);` 这样就会实例化一个新的对象 反射是通过public的无参数来实例化一个对象！！
4. 序列化破坏单例原理：反序列化后的对象会重新分配内存，即重新创建

   readObject0 中为object的时候，调用readOrdinaryObject，里面如果有hasReadResolveMethod，会调用ResolveMethod重写

   其实序列化底层也是调用了newInstance

   readOrdinaryObject方法里

   ```
   obj = desc.isInstantiable() ? desc.newInstance() :
   null;
   ```

5. readResolve()方法防止反序列化破坏单例原理：在反序列化调用readObject()
   方法中，会先反序列化一个实例，再进行判断是否定义了该方法，如果定义了该方法，则将刚才反序列化生成的对象进行覆盖。其实实际上实例化了两次，只不过新创建的对象没有被返回

6.枚举式单例模式：枚举式单例模式，无法通过反射及反序列化来破坏单例。无法通过反射破坏单例是因为jdk底层做了限制，当发现反射调用的是枚举的构造器时，会抛出“”异常；无法反序列化来破坏单例是因为反序列化时如果该Enum类已被实例化则通过类名及类对象找到该枚举类并返回，所以不会产生多实例。

源码:

```
Enum<?> en = Enum.valueOf((Class)cl, name);
进入
public static <T extends Enum<T>> T valueOf(Class<T>
enumType,
                                            String
name) {
        T result =
enumType.enumConstantDirectory().get(name);
        if (result != null)
            return result;
        if (name == null)
            throw new NullPointerException("Name is
null");
        throw new IllegalArgumentException(
            "No enum constant " +
enumType.getCanonicalName() + "." + name);
}
```

7.容器式单例模式：方便于管理众多的单例对象，但会出现线程安全问题，也会出现反射和反序列化破坏其单例的现象，不过spring中的对象管理通过该方式