设计模式六大原则

1. 单一职责原则
2. 迪米特法则
3. 接口隔离原则
4. 里氏替换原则
5. 依赖倒置原则
6. 而开闭原则

工厂模式

1.什么是工厂模式

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

- 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.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。

4. 工厂模式分类

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）

不使用工厂模式

```
public class BMW320 {
    public BMW320(){
        System.out.println("制造-->BMW320");
    }
}

public class BMW523 {
    public BMW523(){
        System.out.println("制造-->BMW523");
    }
}

public class Customer {
    public static void main(String[] args) {
        BMW320 bmw320 = new BMW320();
        BMW523 bmw523 = new BMW523();
    }
}
```

使用简单工厂模式

产品类

```
abstract class BMW {
    public BMW(){

    }
}

public class BMW320 extends BMW {
    public BMW320() {
        System.out.println("制造-->BMW320");
    }
}
public class BMW523 extends BMW{
    public BMW523(){
        System.out.println("制造-->BMW523");
    }
}
```

工厂类

```
public class Factory {
    public BMW createBMW(int type) {
        switch (type) {

        case 320:
```

```
            return new BMW320();

        case 523:
            return new BMW523();

        default:
            break;
        }
        return null;
    }
}
```

客户类

```
public class Customer {
    public static void main(String[] args) {
        Factory factory = new Factory();
        BMW bmw320 = factory.createBMW(320);
        BMW bmw523 = factory.createBMW(523);
    }
}
```

1) 工厂类角色：这是本模式的核心，含有一定的商业逻辑和判断逻辑，用来创建产品

2) 抽象产品角色：它一般是具体产品继承的父类或者实现的接口。

3) 具体产品角色：工厂类所创建的对象就是此角色的实例。在java中由一个具体类实现。

工厂方法模式

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

工厂方法模式组成：

1)抽象工厂角色： 这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在java中它由抽象类或者接口来实现。

2)具体工厂角色：它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。

3)抽象产品角色：它是具体产品继承的父类或者是实现的接口。在java中一般有抽象类或者接口来实现。

4)具体产品角色：具体工厂角色所创建的对象就是此角色的实例。在java中由具体的类来实现。

(开闭原则)当有新的产品产生时，只要按照抽象产品角色、抽象工厂角色提供的合同来生成，那么就可以被客户使用，而不必去修改任何已有 的代码。

产品类

```
 abstract class BMW {
    public BMW(){

    }
}
public class BMW320 extends BMW {
    public BMW320() {
        System.out.println("制造-->BMW320");
    }
}
public class BMW523 extends BMW{
    public BMW523(){
        System.out.println("制造-->BMW523");
    }
}
```

创建工厂类:

```
interface FactoryBMW {
    BMW createBMW();
}

public class FactoryBMW320 implements FactoryBMW{

    @Override
    public BMW320 createBMW() {

        return new BMW320();
    }

}
public class FactoryBMW523 implements FactoryBMW {
    @Override
    public BMW523 createBMW() {

        return new BMW523();
    }
}
```

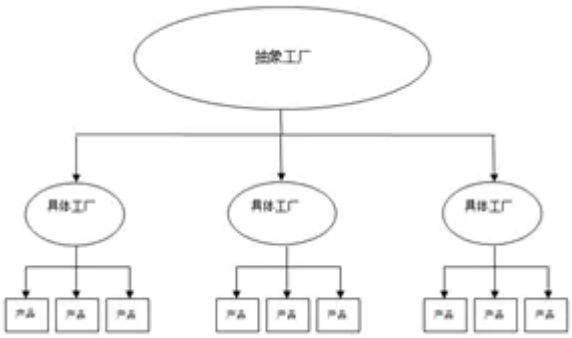
客户类

```
public class Customer {
    public static void main(String[] args) {
        FactoryBMW320 factoryBMW320 = new FactoryBMW320();
        BMW320 bmw320 = factoryBMW320.createBMW();

        FactoryBMW523 factoryBMW523 = new FactoryBMW523();
        BMW523 bmw523 = factoryBMW523.createBMW();
    }
}
```

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。



策略模式

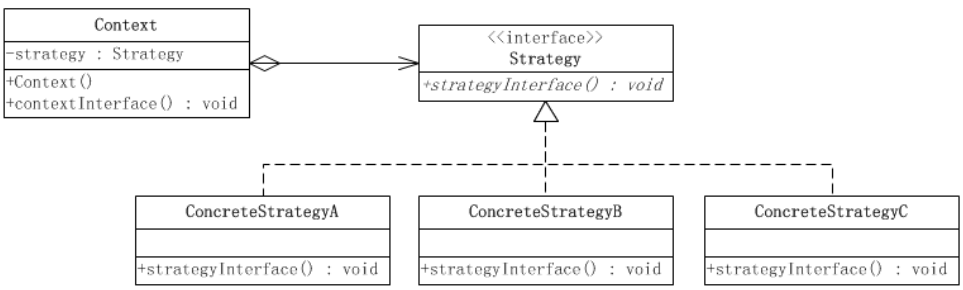
定义

定义一组算法，将每一个算法封装起来，从而使它们可以相互切换。

特点

1）一组算法，那就是不同的策略。

2）这组算法都实现了相同的接口或者继承相同的抽象类，所以可以相互切换。



策略模式涉及到的角色有三个：

- 封装角色：上层访问策略的入口，它持有抽象策略角色的引用。

- 抽象策略角色：提供接口或者抽象类，定义策略组必须拥有的方法和属性。

- 具体策略角色：实现抽象策略，定义具体的算法逻辑。

```
// Context持有Strategy的引用，并且提供了调用策略的方法，
    public class Context {

    private Strategy strategy;

    /**
     * 传进的是一个具体的策略实例
     * @param strategy
     */
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    /**
     * 调用策略
     */
    public void contextInterface() {
        strategy.algorithmLogic();
    }

}
```

```
// 抽象策略角色，定义了策略组的方法
public interface Strategy {

    public void algorithmLogic();

}
```

```
// 具体策略角色类
public class ConcreteStrategyA implements Strategy{

    @Override
    public void algorithmLogic() {
        // 具体的算法逻辑（）
    }
}
```

```
// 客户端
public class Client {

    public static void main(String[] args) {
        // 操控比赛，这场要输
        Context context = new Context(new ConcreteStrategyA());
        context.contextInterface();
    }
}
```

- 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

工厂结合策略实战

课堂代码

单例设计模式(面试重点)

概念：单例对象的类必须保证只有一个实例存在

适用场景： 单例模式只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，如多个模块使用同一个数据源连接对象等等。如：

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. 有状态的工具类对象。
4. 频繁访问数据库或文件的对象。

饿汉式

```
public class Singleton {
     /**
     * 优点：没有线程安全问题，简单
     * 缺点：提前初始化会延长类加载器加载类的时间；如果不使用会浪费内存空间；  不能传递参数
     */
     private static final Singleton instance = new Singleton();
     private Singleton(){};

     public static Singleton getInstance(){
         return instance;
     }
 }
```

懒汉式

```
    public class Singleton{
    /**
     * 优点:解决线程安全，延迟初始化( Effective Java推荐写法)
     */
    private Singleton(){}

    public static Singleton getInstance () {
        return Holder.SINGLE_TON;
    }

    private static class Holder{
        private static final Singleton SINGLE_TON = new Singleton();
    }
}
```

Java中的静态变量和静态代码块是在类加载的时候就执行的

成员变量随着对象的创建而存在，随着对象的被回收而释放;

静态变量随着类的加载而存在，随着类的消失而消失;

双重检查锁（double checked locking)

```
public class Singleton {
    private volatile static Singleton uniqueSingleton;

    private Singleton() {
    }

    public Singleton getInstance() {
        if (null == uniqueSingleton) {
            synchronized (Singleton.class) {
                if (null == uniqueSingleton) {
                    uniqueSingleton = new Singleton();
                }
            }
        }
        return uniqueSingleton;
    }
}
```

为什么要使用volatile指令重排序

volatile指令重排序，在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

1. 在单线程环境下不能改变程序运行的结果;

2. 存在数据依赖关系的不允许重排序

uniqueSingleton = new Singleton();

分配内存空间、初始化对象、将对象指向刚分配的内存空间

但是有些编译器为了性能的原因，可能会将第二步和第三步进行重排序，顺序就成了：

分配内存空间、将对象指向刚分配的内存空间、初始化对象

现在考虑重排序后，两个线程发生了以下调用：

Time	Thread A	Thread B
T1	检查到uniqueSingleton为空	
T2	获取锁	
T3	再次检查到uniqueSingleton为空	
T4	为uniqueSingleton分配内存空间	
T5	将uniqueSingleton指向内存空间	
T6		检查到uniqueSingleton不为空
T7		访问uniqueSingleton（此时对象还未完成初始化）
T8	初始化uniqueSingleton	

在这种情况下，T7时刻线程B对uniqueSingleton的访问，访问的是一个初始化未完成的对象。

使用了volatile关键字后，重排序被禁止，所有的写（write）操作都将发生在读（read）操作之前。

为什么要双重检查null

考虑这样一种情况，就是有两个线程同时到达，即同时调用getInstance() 方法，

此时由于singleTon== null ，所以很明显，两个线程都可以通过第一重的 singleTon== null ，

进入第一重 if语句后，由于存在锁机制，所以会有一个线程进入 lock 语句并进入第二重 singleTon== null ，

而另外的一个线程则会在lock 语句的外面等待。

而当第一个线程执行完new SingleTon（）语句后，便会退出锁定区域，此时，第二个线程便可以进入 lock 语句块，

此时，如果没有第二重singleTon== null 的话，那么第二个线程还是可以调用 new SingleTon（）语句，

这样第二个线程也会创建一个SingleTon实例，这样也还是违背了单例模式的初衷的，

所以这里必须要使用双重检查锁定。

细心的朋友一定会发现，如果我去掉第一重singleton == null ，程序还是可以在多线程下完好的运行的，

考虑在没有第一重singleton == null 的情况下，

当有两个线程同时到达，此时，由于lock 机制的存在，第一个线程会进入 lock 语句块，并且可以顺利执行 new SingleTon（），

当第一个线程退出lock 语句块时， singleTon 这个静态变量已不为 null 了，所以当第二个线程进入 lock 时，

还是会被第二重singleton == null 挡在外面，而无法执行 new Singleton（），

所以在没有第一重singleton == null 的情况下，也是可以实现单例模式的？那么为什么需要第一重singleton == null呢?

这里就涉及一个性能问题了，因为对于单例模式的话，newSingleTon（）只需要执行一次就 OK 了，

而如果没有第一重singleTon == null 的话，每一次有线程进入getInstance（）时，均会执行锁定操作来实现线程同步，

这是非常耗费性能的，而如果我加上第一重singleTon == null 的话，

那么就只有在第一次，也就是singleTton ==null 成立时的情况下执行一次锁定以实现线程同步，

而以后的话，便只要直接返回Singleton 实例就 OK 了而根本无需再进入 lock语句块了，这样就可以解决由线程同步带来的性能问题了。

单例模式的破坏

```
Singleton sc1 = Singleton.getInstance();
Singleton sc2 = Singleton.getInstance();
System.out.println(sc1); // sc1，sc2是同一个对象
System.out.println(sc2);
/*通过反射的方式直接调用私有构造器*/
Class<Singleton> clazz = (Class<Singleton>)
Class.forName("com.learn.example.Singleton");
Constructor<Singleton> c = clazz.getDeclaredConstructor(null);
c.setAccessible(true); // 跳过权限检查
Singleton sc3 = c.newInstance();
Singleton sc4 = c.newInstance();
System.out.println("通过反射的方式获取的对象sc3：" + sc3);  // sc3，sc4不是同一个对象
System.out.println("通过反射的方式获取的对象sc4：" + sc4);
public class Singleton implements Serializable {
    private volatile static Singleton uniqueSingleton;

    private Singleton() {
    }

    public Singleton getInstance() {
        if (null == uniqueSingleton) {
            synchronized (Singleton.class) {
                if (null == uniqueSingleton) {
                    uniqueSingleton = new Singleton();
                }
            }
        }
        return uniqueSingleton;
```

```
    }

      private static class SingletonInstance {
        private static final Singleton INSTANCE = new Singleton();
   }
    public static Singleton getInstance() {
        return SingletonInstance.INSTANCE;
   }
    private Object readResolve() {
        return SingletonInstance.INSTANCE;
   }
}
```

spring中bean的单例

1. 当多个用户同时请求一个服务时，容器会给每一个请求分配一个线程，这时多个线程会并发执行该请求对应的业务逻辑（成员方法），此时就要注意了，如果该处理逻辑中有对单例状态的修改（体现为该单例的成员属性），则必须考虑线程同步问题。
   有状态就是有数据存储功能。有状态对象(Stateful Bean)，就是有实例变量的对象，可以保存数据，是非线程安全的。在不同方法调用间不保留任何状态。
   无状态就是一次操作，不能保存数据。无状态对象(Stateless Bean)，就是没有实例变量的对象 .不能保存数据，是不变类，是线程安全的。
2. 实现

```
public abstract class AbstractBeanFactory implements ConfigurableBeanFactory{
/**
* 充当了Bean实例的缓存，实现方式和单例注册表相同
*/
private final Map singletonCache=new HashMap();
public Object getBean(String name)throws BeansException{
   return getBean(name,null,null);
}
...
public Object getBean(String name,Class requiredType,Object[] args)throws
BeansException{
  //对传入的Bean name稍做处理，防止传入的Bean name名有非法字符(或则做转码)
  String beanName=transformedBeanName(name);
  Object bean=null;
  //手工检测单例注册表
  Object sharedInstance=null;
  //使用了代码锁定同步块，原理和同步方法相似，但是这种写法效率更高
  synchronized(this.singletonCache){
     sharedInstance=this.singletonCache.get(beanName);
   }
  if(sharedInstance!=null){
     ...
     //返回合适的缓存Bean实例
     bean=getObjectForSharedInstance(name,sharedInstance);
  }else{
    ...
    //取得Bean的定义
    RootBeanDefinition
mergedBeanDefinition=getMergedBeanDefinition(beanName,false);
     ...
    //根据Bean定义判断，此判断依据通常来自于组件配置文件的单例属性开关
    //<bean id="date" class="java.util.Date" scope="singleton"/>
```

```
    //如果是单例，做如下处理
    if(mergedBeanDefinition.isSingleton()){
       synchronized(this.singletonCache){
        //再次检测单例注册表
         sharedInstance=this.singletonCache.get(beanName);
         if(sharedInstance==null){
            ...
           try {
              //真正创建Bean实例
              sharedInstance=createBean(beanName,mergedBeanDefinition,args);
              //向单例注册表注册Bean实例
               addSingleton(beanName,sharedInstance);
           }catch (Exception ex) {
              ...
           }finally{
              ...
          }
         }
       }
      bean=getObjectForSharedInstance(name,sharedInstance);
    }
   //如果是非单例，即prototpye，每次都要新创建一个Bean实例
   //<bean id="date" class="java.util.Date" scope="prototype"/>
   else{
      bean=createBean(beanName,mergedBeanDefinition,args);
   }
}
...
return bean;
}
}
```

controller默认是单例的

不要使用非静态的成员变量，否则会发生数据逻辑混乱。

```
@Controller
public class ScopeTestController {

    private int num = 0;

    @RequestMapping("/testScope")
    public void testScope() {
        System.out.println(++num);
    }

    @RequestMapping("/testScope2")
    public void testScope2() {
        System.out.println(++num);
    }

}
//我们首先访问 http://localhost:8080/testScope，得到的答案是1;
//然后我们再访问 http://localhost:8080/testScope2，得到的答案是 2。
```

单例对象生命周期

- 出生：容器创建时对象出生。(立即创建)
- 活着：只要容器在，对象一直活着。
- 死亡：容器销毁，对象消亡。
- `总结：单例对象与容器共存亡`。

多例对象生命周期

- 出生：当我们使用对象时，Spring框架为我们创建对象。(延迟创建)
- 活着：对象只要在使用过程中就一直活着。
- 死亡：当对象长时间不用且没有别的对象引用时，由Java的垃圾回收器回收。

工具类用单例模式还是静态方法

- 如果没有配置信息的工具类，当然是静态类好，随处调用，不需引用
- 如果有配置信息的工具类，最好还是使用单例模式吧，这样以后如果有多个数据源

命令模式

1. 背景:当需要向某些对象发送请求，但是并不知道请求的接收者是谁，也不知道被请求的操作是哪个，使得请求发送者与请求接收者消解耦

(1)我们须要Client和Receiver同时开发,而且在开发过程中分别须要不停重购，改名

(2)如果我们要求Redo ,Undo等功能

(3)我们须要命令不按照调用执行，而是按照执行时的情况排序，执行

(4)在上边的情况下，我们的接受者有很多，不止一个

1. 模式定义

命令模式(Command Pattern)：将一个请求封装为一个对象，从而使我们可用不同的请求对客户进行参数化；对请求排队或者记录请求日志，以及支持可撤销的操作。

1. 模式结构

命令模式包含如下角色：

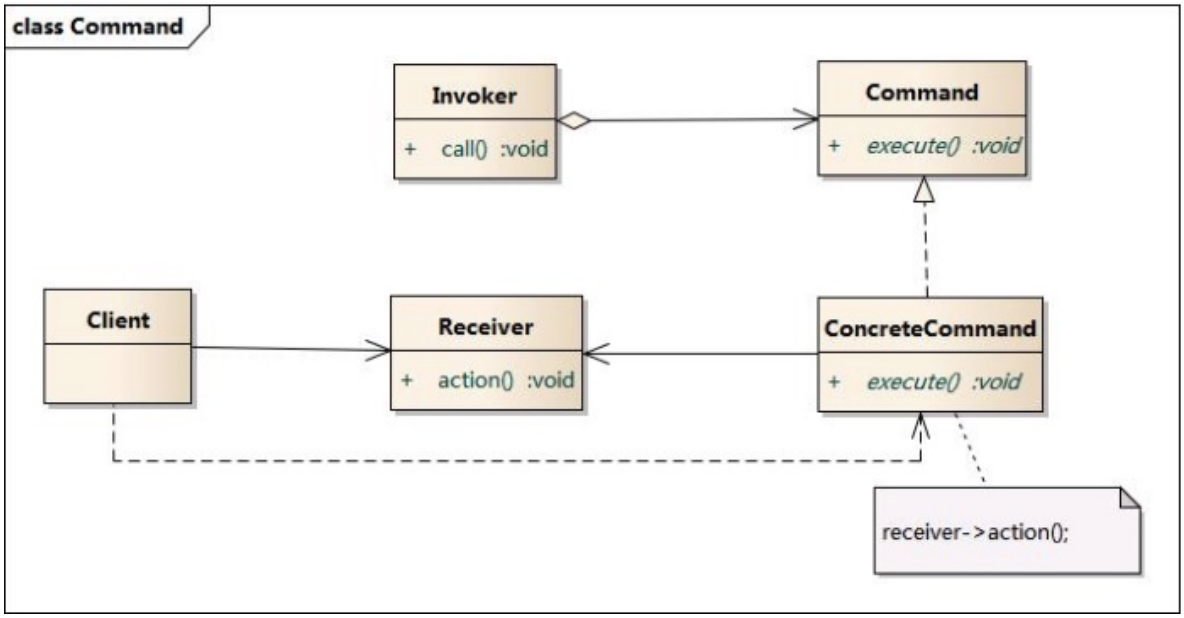
Command: 抽象命令类

ConcreteCommand: 具体命令类

Invoker: 调用者

Receiver: 接收者

Client:客户类



1. 命令模式的优点

降低系统的耦合度。

新的命令可以很容易地加入到系统中。

可以比较容易地设计一个命令队列和宏命令（组合命令）。

可以方便地实现对请求的Undo和Redo。

1. 命令模式的缺点

使用命令模式可能会导致某些系统有过多的具体命令类。因为针对每一个命令都需要设计一个具体命令类，　因此某些系统可能需要大量具体命令类，这将影响命令模式的使用。

1. 样例代码

```
// 客户端，请求者，命令接口，命令实现，接受者，
public class Client {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command commandOne = new ConcreteCommandOne(receiver);
        Command commandTwo = new ConcreteCommandTwo(receiver);
        Invoker invoker = new Invoker(commandOne, commandTwo);
        invoker.actionOne();
        invoker.actionTwo();
    }
}

public class Invoker {
    private Command commandOne;
    private Command commandTwo;

    public Invoker(Command commandOne, Command commandTwo) {
        this.commandOne = commandOne;
        this.commandTwo = commandTwo;

    }
```

```
    public void actionOne() {
        commandOne.execute();

    }

    public void actionTwo() {
        commandTwo.execute();

    }
}

public interface Command {
    void execute();
}

public class ConcreteCommandOne implements Command {
    private Receiver receiver

    public ConcreteCommandOne(Receiver receiver) {
        this.receiver = receiver;

    }

    public void execute() {
        receiver.actionOne();

    }
}

public class ConcreteCommandTwo implements Command {
    private Receiver receiver

    public ConcreteCommandTwo(Receiver receiver) {
        this.receiver = receiver;

    }

    public void execute() {
        receiver.actionTwo();

    }
}

public class Receiver {
    public Receiver() {
        //

    }

    public void actionOne() {
        System.out.println("ActionOne has been taken.");

    }

    public void actionTwo() {
        System.out.println("ActionTwo has been taken.");

    }
```

```
}
```

redo undo

```
public class ConcreteCommandOne implements Command {
    private Receiver receiver
    private Receiver lastReceiver;

    public ConcreteCommandOne(Receiver receiver) {
        this.receiver = receiver;
    }

    public void execute() {
        record();
        receiver.actionOne();
    }

    public void undo() {
        //恢复状态
    }

    public void redo() {
        lastReceiver.actionOne();
        //
    }

    public record() {
        //记录状态
    }
}
```

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

代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。

代理模式一般涉及到的角色有：

抽象角色：声明真实对象和代理对象的共同接口；

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。

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。

```
/**
 * 抽象角色
 */
public abstract class Subject {
    public abstract void request();
```

```
/**
 * 真实的角色
 */
public class RealSubject extends Subject {

    @Override
    public void request() {
        // TODO Auto-generated method stub

    }

}
```

```
/**
 * 静态代理，对具体真实对象直接引用
 * 代理角色，代理角色需要有对真实角色的引用，
 * 代理做真实角色想做的事情
 */
public class ProxySubject extends Subject {

    private RealSubject realSubject = null;

    /**
     * 除了代理真实角色做该做的事情，代理角色也可以提供附加操作，
     * 如：preRequest()和postRequest()
     */
    @Override
    public void request() {
        preRequest();  //真实角色操作前的附加操作

        if(realSubject == null){
            realSubject =  new RealSubject();
        }
```

```
        realSubject.request();

        postRequest();  //真实角色操作后的附加操作
    }

    /**
     *  真实角色操作前的附加操作
     */
    private void postRequest() {
        // TODO Auto-generated method stub

    }

    /**
     *  真实角色操作后的附加操作
     */
    private void preRequest() {
        // TODO Auto-generated method stub

    }

}
```

```
/**
 *  客户端调用
 */
public class Main {
    public static void main(String[] args) {
        Subject subject = new ProxySubject();
        subject.request();  //代理者代替真实者做事情
    }
}
```

优点：可以做到在不修改目标对象的功能前提下,对目标功能扩展.

缺点：每一个代理类都必须实现一遍委托类（也就是realsubject）的接口，如果接口增加方法，则代理类也必须跟着修改。其次，代理类每一个接口对象对应一个委托对象，如果委托对象非常多，则静态代理类就非常臃肿，难以胜任。

jdk动态代理

动态代理解决静态代理中代理类接口过多的问题，通过反射来实现的，借助Java自带的java.lang.reflect.Proxy,通过固定的规则生成。

步骤如下：

1. 编写一个委托类的接口，即静态代理的（Subject接口）
2. 实现一个真正的委托类，即静态代理的（RealSubject类）
3. 创建一个动态代理类，实现InvocationHandler接口，并重写该invoke方法
4. 在测试类中，生成动态代理的对象。

第一二步骤，和静态代理一样,第三步：

```
public class DynamicProxy implements InvocationHandler {
    private Object object;
    public DynamicProxy(Object object) {
        this.object = object;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        Object result = method.invoke(object, args);
        return result;
    }
}
```

创建动态代理的对象

```
Subject realSubject = new RealSubject();
DynamicProxy proxy = new DynamicProxy(realSubject);
ClassLoader classLoader = realSubject.getClass().getClassLoader();
Subject subject = (Subject) Proxy.newProxyInstance(classLoader, new  Class[]
{Subject.class}, proxy);
subject.visit();
```

上述代码的关键是Proxy.newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler handler)方法，该方法会根据指定的参数动态创建代理对象。三个参数的意义如下：

1. loader，指定代理对象的类加载器；
2. interfaces，代理对象需要实现的接口，可以同时指定多个接口；
3. handler，方法调用的实际处理者，代理对象的方法调用都会转发到这里（*注意1）。

cglib动态代理

假设我们有一个没有实现任何接口的类HelloConcrete

```
public class HelloConcrete {
    public String sayHello(String str) {
        return "HelloConcrete: " + str;
    }
}
```

```
// CGLIB动态代理
// 1. 首先实现一个MethodInterceptor，方法调用会被转发到该类的intercept()方法。
class MyMethodInterceptor implements MethodInterceptor{
  ...
    @Override
```

```
    public Object intercept(Object obj, Method method, Object[] args,
MethodProxy proxy) throws Throwable {
        logger.info("You said: " + Arrays.toString(args));
        return proxy.invokeSuper(obj, args);
    }
}
// 2. 然后在需要使用HelloConcrete的时候，通过CGLIB动态代理获取代理对象。
Enhancer enhancer = new Enhancer();
enhancer.setSuperclass(HelloConcrete.class);
enhancer.setCallback(new MyMethodInterceptor());

HelloConcrete hello = (HelloConcrete)enhancer.create();
System.out.println(hello.sayHello("I love you!"));
```

通过CGLIB的Enhancer来指定要代理的目标对象、实际处理代理逻辑的对象，最终通过调用create()方法得到代理对象，对这个对象所有非final方法的调用都会转发给MethodInterceptor.intercept()方法，在intercept()方法里我们可以加入任何逻辑，比如修改方法参数，加入日志功能、安全检查功能等；通过调用MethodProxy.invokeSuper()方法，我们将调用转发给原始对象，具体到本例，就是HelloConcrete的具体方法。

对于从Object中继承的方法，CGLIB代理也会进行代理，如hashCode()、equals()、toString()等，但是getClass()、wait()等方法不会，因为它是final方法，CGLIB无法代理。

原理：

CGLIB是一个强大的高性能的代码生成包，底层是通过使用一个小而快的字节码处理框架ASM，它可以在运行期扩展Java类与实现Java接口，Enhancer是CGLIB的字节码增强器，可以很方便的对类进行拓展

创建代理对象的几个步骤:

1、生成代理类的二进制字节码文件

2、加载二进制字节码，生成Class对象(例如使用Class.forName()方法)

3、通过反射机制获得实例构造，并创建代理类对象

区别

1. jdk动态代理：利用拦截器(拦截器必须实现InvocationHanlder)加上反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。只能对实现了接口的类生成代理只能对实现了接口的类生成代理
2. cglib：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。主要是对指定的类生成一个子类，覆盖其中的方法，并覆盖其中方法实现增强，但是因为采用的是继承，对于final类或方法，是无法继承的。
3. 选择
4.
    1. 如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP。
    2. 如果目标对象实现了接口，可以强制使用CGLIB实现AOP。
    3. 如果目标对象没有实现了接口，必须采用CGLIB库，Spring会自动在JDK动态代理和CGLIB之间转换。

模板方法模式

意图：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。

优点： 1、封装不变部分，扩展可变部分。 2、提取公共代码，便于维护。 3、行为由父类控制，子类实现。

缺点：每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。

```
//创建一个抽象类，它的模板方法被设置为 final
public abstract class Game {
   abstract void initialize();
   abstract void startPlay();
   abstract void endPlay();

   //模板
   public final void play(){

      //初始化游戏
      initialize();

      //开始游戏
      startPlay();

      //结束游戏
      endPlay();
   }
}
//创建扩展了上述类的实体类
public class Football extends Game {

   @Override
   void endPlay() {
      System.out.println("Football Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Football Game Initialized! Start playing.");
   }

   @Override
   void startPlay() {
      System.out.println("Football Game Started. Enjoy the game!");
   }
}
```

适配器模式

意图：将一个类的接口转换成客户希望的另外一个接口

如何解决：继承或依赖（推荐）

何时使用： 1、系统需要使用现有的类，而此类的接口不符合系统的需要。 2、想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口。 3、通过接口转换，将一个类插入另一个类系中。 （比如老虎和飞禽，现在多了一个飞虎，在不增加实体的需求下，增加一个适配器，在里面包容一个虎对象，实现飞的接口。）

优点： 1、可以让任何两个没有关联的类一起运行。 2、提高了类的复用。 3、增加了类的透明度。 4、灵活性好。

缺点： 1、过多地使用适配器，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，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。 2.由于 JAVA 至多继承一个类，所以至多只能适配一个适配者类，而且目标类必须是抽象类。

```
// 为媒体播放器和更高级的媒体播放器创建接口
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
// 创建实现了 AdvancedMediaPlayer 接口的实体类
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }

   @Override
   public void playMp4(String fileName) {
      //什么也不做
   }
}
public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {
      //什么也不做
   }

   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);
   }
}
//创建实现了 MediaPlayer 接口的适配器类。
public class MediaAdapter implements MediaPlayer {

   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){
      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();
      } else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }

   @Override
   public void play(String audioType, String fileName) {
      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
```

```
    }
}
//创建实现了 MediaPlayer 接口的实体类
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @Override
   public void play(String audioType, String fileName) {

      //播放 mp3 音乐文件的内置支持
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: "+ fileName);
      }
      //mediaAdapter 提供了播放其他文件格式的支持
      else if(audioType.equalsIgnoreCase("vlc")
         || audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }
      else{
         System.out.println("Invalid media. "+
            audioType + " format not supported");
      }
   }
}
```

装饰器模式

允许向一个现有的对象添加新的功能，同时又不改变其结构

优点：装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式，装饰模式可以动态扩展一个实现类的功能。

缺点：多层装饰比较复杂。

使用场景： 1、扩展一个类的功能。 2、动态增加功能，动态撤销。

```
public interface Shape {
   void draw();
}
// 创建实现接口的实体类
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Circle");
   }
}
// 创建实现了 Shape 接口的抽象装饰类。
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
```

```
      decoratedShape.draw();
   }
}
// 创建扩展了 ShapeDecorator 类的实体装饰类。
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

观察者模式

意图：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。

关键代码：在抽象类里有一个 ArrayList 存放观察者们。

优点： 1、观察者和被观察者是抽象耦合的。 2、建立一套触发机制。

缺点： 1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。 2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。 3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

```
// 创建 Subject 类
import java.util.ArrayList;
import java.util.List;

public class Subject {

   private List<Observer> observers
      = new ArrayList<Observer>();
   private int state;

   public int getState() {
      return state;
   }

   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }

   public void attach(Observer observer){
      observers.add(observer);
   }
```

```
   public void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}
//创建 Observer 类
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
//创建实体观察者类。
public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: "
      + Integer.toBinaryString( subject.getState() ) );
   }
}
//使用 Subject 和实体观察者对象。
public class ObserverPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```