java的设计模式大体上分为三大类：

创建型模式（5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（7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（11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。

设计模式遵循的原则有6个：

1、开闭原则（Open Close Principle）

对扩展开放，对修改关闭。

2、里氏代换原则（Liskov Substitution Principle）

只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。

3、依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，对接口编程，依赖于抽象而不依赖于具体。

4、接口隔离原则（Interface Segregation Principle）

使用多个隔离的借口来降低耦合度。

5、迪米特法则（最少知道原则）（Demeter Principle）

一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

6、合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。继承实际上破坏了类的封装性，超类的方法可能会被子类修改。

1.单例模式：确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

（1）懒汉式

```
public class Singleton {
```

```
    /* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载
*/
    private static Singleton instance = null;

    /* 私有构造方法，防止被实例化 */
    private Singleton() {
    }

    /* 1:懒汉式，静态工程方法，创建实例 */
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

（2）饿汉式

```
public class Singleton {

    /* 持有私有静态实例，防止被引用 */
    private static Singleton instance = new Singleton();

    /* 私有构造方法，防止被实例化 */
    private Singleton() {
    }

    /* 1:懒汉式，静态工程方法，创建实例 */
    public static Singleton getInstance() {
        return instance;
    }
}
```

使用场景：
1.要求生成唯一序列号的环境；
2.在整个项目中需要一个共享访问点或共享数据，例如一个Web页面上的计数器，可以不用把每次刷新都记录到数据库中，使用单例模式保持计数器的值，并确保是线程安全的；
3.创建一个对象需要消耗的资源过多，如要访问IO和数据库等资源；
4.需要定义大量的静态常量和静态方法（如工具类）的环境，可以采用单例模式（当然，也可以直接声明为static的方式）。

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

```
public interface Fruit {
    public void print();
}
```

2个实现类

```
public class Apple implements Fruit{

    @Override
    public void print() {
        System.out.println("我是一个苹果");
    }

}
public class Orange implements Fruit{

    @Override
    public void print() {
        System.out.println("我是一个橘子");
```

```
    }

}
```

工厂类

```
public class FruitFactory {
    public Fruit produce(String type){
        if(type.equals("apple")){
            return new Apple();
        }else if(type.equals("orange")){
            return new Orange();
        }else{
            System.out.println("请输入正确的类型!");
            return null;
        }
    }
}
```

使用场景：jdbc连接数据库，硬件访问，降低对象的产生和销毁

3.抽象工厂模式：为创建一组相关或相互依赖的对象提供一个接口，而且无须指定它们的具体类。

相对于工厂模式，我们可以新增产品类（只需要实现产品接口），只需要同时新增一个工厂类，客户端就可以轻松调用新产品的代码。

```
interface food{}
class A implements food{}
class B implements food{}
interface produce{ food get();}
class FactoryForA implements produce{
    @Override
    public food get() {
        return new A();
```

```
    }
}
class FactoryForB implements produce{
    @Override
    public food get() {
        return new B();
    }
}
public class AbstractFactory {
    public void ClientCode(String name){
        food x= new FactoryForA().get();
        x = new FactoryForB().get();
    }
}
```

使用场景：一个对象族（或是一组没有任何关系的对象）都有相同的约束。

涉及不同操作系统的时候，都可以考虑使用抽象工厂模式

4.建造者模式：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。

```
public class Build {
    static class Student{
    String name = null ;
        int number = -1 ;
        String sex = null ;
    public Student(Builder builder)
      {
    this.name=builder.name;
    this.number=builder.number;
    this.sex=builder.sex;
    }
        static class Builder{
```

```
            String name = null ;
            int number = -1 ;
            String sex = null ;
            public Builder setName(String name){
                this.name=name;
                return this;
            }
            public Builder setNumber(int number){
                this.number=number;
                return this;
            }
            public Builder setSex(String sex){
                this.sex=sex;
                return this;
            }
            public Student build(){
                return new Student(this);
            }

        }
    }
    public static void main(String[] args) {
    Student A=new Student.Builder().setName("张
三").setNumber(1).build();
    Student B=new Student.Builder().setSex("男").setName("李四").build();
        System.out.println(A.name+" "+A.number+" "+A.sex);
        System.out.println(B.name+" "+B.number+" "+B.sex);
    }
}
```

使用场景：

1. 相同的方法，不同的执行顺序，产生不同的事件结果时，可以采用建造者模式。
2.多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时，则可以使用该模式。
3.产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效能，这个时候使用建造者模式非常合适。
5.原型模式：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

```
public class Prototype implements Cloneable{
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    protected Object clone()   {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }finally {
            return null;
        }
    }
    public static void main ( String[] args){
        Prototype pro = new Prototype();
        Prototype pro1 = (Prototype)pro.clone();
    }
```

}

原型模式实际上就是实现Cloneable接口，重写clone（）方法。

使用原型模式的优点：

1.性能优良

原型模式是在内存二进制流的拷贝，要比直接new一个对象性能好很多，特别是要在一个循环体内产生大量的对象时，原型模式可以更好地体现其优点。

2.逃避构造函数的约束

这既是它的优点也是缺点，直接在内存中拷贝，构造函数是不会执行的（参见13.4节）。

使用场景：

1.资源优化场景

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。

2.性能和安全要求的场景

通过new产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。

3. 一个对象多个修改者的场景

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。

浅拷贝和深拷贝：

浅拷贝：Object类提供的方法clone只是拷贝本对象，其对象内部的数组、引用对象等都不拷贝，还是指向原生对象的内部元素地址，这种拷贝就叫做浅拷贝，其他的原始类型比如int、long、char、string（当做是原始类型）等都会被拷贝。

注意：　使用原型模式时，引用的成员变量必须满足两个条件才不会被拷贝：一是类的成员变量，而不是方法内变量；二是必须是一个可变的引用对象，而不是一个原始类型或不可变对象。

深拷贝：对私有的类变量进行独立的拷贝

如：this.arrayList = (ArrayList)this.arrayList.clone();

6.适配器模式：将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

主要可分为3种：

1.类适配：创建新类，继承源类，并实现新接口，例如

class adapter extends oldClass implements newFunc{}

2.对象适配：创建新类持源类的实例，并实现新接口，例如

class adapter implements newFunc { private oldClass oldInstance ;}

3.接口适配：创建新的抽象类实现旧接口方法。例如

abstract class adapter implements oldClassFunc { void newFunc();}

使用场景：

你有动机修改一个已经投产中的接口时，适配器模式可能是最适合你的模式。比如系统扩展了，需要使用一个已有或新建立的类，但这个类又不符合系统的接口，怎么办？使用适配器模式，这也是我们例子中提到的。

7.装饰器模式：动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活

。

```
interface Source{ void method();}
public class Decorator implements Source{
    private Source source ;
    public void decotate1(){
        System.out.println("decorate");
    }
    @Override
    public void method() {
        decotate1();
        source.method();
    }
}
```

使用场景：

1. 需要扩展一个类的功能，或给一个类增加附加功能。

2. 需要动态地给一个对象增加功能，这些功能可以再动态地撤销。
3. 需要为一批的兄弟类进行改装或加装功能，当然是首选装饰模式。

8.代理模式：为其他对象提供一种代理以控制对这个对象的访问。

```
interface Source{ void method();}
class OldClass implements Source{
    @Override
    public void method() {
    }
}
class Proxy implements Source{
    private Source source = new OldClass();
    void doSomething(){}
    @Override
    public void method() {
        new Class1().Func1();
        source.method();
        new Class2().Func2();
        doSomething();
    }
}
```

9.中介者模式：用一个中介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，从而使其耦合松散，而且可以独立地改变它们之间的交互。

```
public abstract class Mediator {
    //定义同事类
    protected ConcreteColleague1 c1;
    protected ConcreteColleague2 c2;
    //通过getter/setter方法把同事类注入进来
    public ConcreteColleague1 getC1() {
            return c1;
    }
```

```
    public void setC1(ConcreteColleague1 c1) {
        this.c1 = c1;
    }
    public ConcreteColleague2 getC2() {
        return c2;
}
    public void setC2(ConcreteColleague2 c2) {
        this.c2 = c2;
    }
    //中介者模式的业务逻辑
    public abstract void doSomething1();
    public abstract void doSomething2();
}
```

使用场景：

中介者模式适用于多个对象之间紧密耦合的情况，紧密耦合的标准是：在类图中出现了蜘蛛网状结构，即每个类都与其他的类有直接的联系。

10.命令模式：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。

Receiver接受者角色：该角色就是干活的角色，命令传递到这里是应该被执行的

Command命令角色：需要执行的所有命令都在这里声明

Invoker调用者角色：接收到命令，并执行命令

```
//通用Receiver类
public abstract class Receiver {
    public abstract void doSomething();
}
//具体Receiver类
public class ConcreteReciver1 extends Receiver{
    //每个接收者都必须处理一定的业务逻辑
    public void doSomething(){ }
}
```

```
public class ConcreteReciver2 extends Receiver{
    //每个接收者都必须处理一定的业务逻辑
    public void doSomething(){ }
}
//抽象Command类
public abstract class Command {
    public abstract void execute();
}
//具体的Command类
public class ConcreteCommand1 extends Command {
    //对哪个Receiver类进行命令处理
    private Receiver receiver;
    //构造函数传递接收者
    public ConcreteCommand1(Receiver _receiver){
        this.receiver = _receiver;
    }
    //必须实现一个命令
    public void execute() {
    //业务处理
        this.receiver.doSomething();
    }
}
public class ConcreteCommand2 extends Command {
    //哪个Receiver类进行命令处理
    private Receiver receiver;
    //构造函数传递接收者
    public ConcreteCommand2(Receiver _receiver){
        this.receiver = _receiver;
    }
    //必须实现一个命令
```

```
    public void execute() {
        //业务处理
        this.receiver.doSomething();
    }
}
//调用者Invoker类
public class Invoker {
    private Command command;

    public void setCommand(Command _command){
        this.command = _command;
    }

    public void action() {
        this.command.execute();
    }
}
//场景类
public class Client {
    public static void main(String[] args){
        Invoker invoker = new Invoker();
        Receiver receiver = new ConcreteReceiver1();

        Command command = new ConcreteCommand1(receiver);
        invoker.setCommand(command);
        invoker.action();
    }
}
```

使用场景：

认为是命令的地方就可以采用命令模式，例如，在GUI开发中，一个按钮的点击是一个命令，可以采用命令模式；模拟DOS命令的时候，当然也要采用命令模

式；触发 - 反馈机制的处理等。

11.责任链模式：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受者之间的耦合关系。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。

```
public abstract class Handler {
    private Handler nextHandler;
    //每个处理者都必须对请求做出处理
    public final Response handleMessage(Request request){
        Response response = null;
        //判断是否是自己的处理级别
        if(this.getHandlerLevel().equals(request.getRequestLevel())){
            response = this.echo(request);
        }else{  //不属于自己的处理级别
            //判断是否有下一个处理者
            if(this.nextHandler != null){
                response = this.nextHandler.handleMessage(request);
            }else{
                //没有适当的处理者，业务自行处理
            }
        }
        return response;
    }
    //设置下一个处理者是谁
    public void setNext(Handler _handler){
        this.nextHandler = _handler;
    }
    //每个处理者都有一个处理级别
    protected abstract Level getHandlerLevel();
    //每个处理者都必须实现处理任务
    protected abstract Response echo(Request request);
```

}

12.策略模式：定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。

使用场景：

1. 多个类只有在算法或行为上稍有不同的场景。
2. 算法需要自由切换的场景。
3. 需要屏蔽算法规则的场景。

13.迭代器模式：它提供一种方法访问一个容器对象中各个元素，而又不需暴露该对象的内部细节。

迭代器模式已经被淘汰，java中已经把迭代器运用到各个聚集类（collection）中了，使用java自带的迭代器就已经满足我们的需求了。

14.组合模式：将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

```
public class Composite extends Component {
    //构件容器
    private ArrayList componentArrayList = new ArrayList();
    //增加一个叶子构件或树枝构件
    public void add(Component component){
        this.componentArrayList.add(component);
    }
    //删除一个叶子构件或树枝构件
    public void remove(Component component){
this.componentArrayList.remove(component);
    }
    //获得分支下的所有叶子构件和树枝构件
    public ArrayList getChildren(){
        return this.componentArrayList;
    }
}
```

使用场景：

1. 维护和展示部分-整体关系的场景，如树形菜单、文件和文件夹管理。
2. 从一个整体中能够独立出部分模块或功能的场景。

15.观察者模式：定义对象间一种一对多的依赖关系，使得每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被自动更新。

```
public abstract class Subject {
    //定义一个观察者数组
    private Vector obsVector = new Vector();
    //增加一个观察者
    public void addObserver(Observer o){
        this.obsVector.add(o);
    }
    //删除一个观察者
    public void delObserver(Observer o){
        this.obsVector.remove(o);
    }
    //通知所有观察者
    public void notifyObservers(){
        for(Observer o:this.obsVector){
            o.update();
}
    }
}
```

使用场景：

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。
2. 事件多级触发场景。
3. 跨系统的消息交换场景，如消息队列的处理机制

16.门面模式：要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。

```
public class Facade {
```

```
    private subSystem1 subSystem1 = new subSystem1();
    private subSystem2 subSystem2 = new subSystem2();
    private subSystem3 subSystem3 = new subSystem3();

    public void startSystem(){
        subSystem1.start();
        subSystem2.start();
        subSystem3.start();
    }

    public void stopSystem(){
        subSystem1.stop();
        subSystem2.stop();
        subSystem3.stop();
    }
}
```

使用场景：

1. 为一个复杂的模块或子系统提供一个供外界访问的接口
2. 子系统相对独立——外界对子系统的访问只要黑箱操作即可
3. 预防低水平人员带来的风险扩散

17.备忘录模式：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。这样以后就可将该对象恢复到原先保存的状态。

```
public class Originator {
    private String state;
    /**
     * 工厂方法，返回一个新的备忘录对象
     */
    public Memento createMemento(){
        return new Memento(state);
    }
    /**
```

```
     * 将发起人恢复到备忘录对象所记载的状态
     */
    public void restoreMemento(Memento memento){
        this.state = memento.getState();
    }
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
        System.out.println("当前状态：" + this.state);
    }
}
```

使用场景：

1. 需要保存和恢复数据的相关状态场景。
2. 提供一个可回滚（rollback）的操作。
3. 需要监控的副本场景中。
4. 数据库连接的事务管理就是用的备忘录模式。

18.访问者模式：封装一些作用于某种数据结构中的各元素的操作，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。

使用场景：

1. 一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作，也就说是用迭代器模式已经不能胜任的情景。
2. 需要对一个对象结构中的对象进行很多不同并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。

19.状态模式：当一个对象内在状态改变时允许其改变行为，这个对象看起来像改变了其类。

使用场景：

1. 行为随状态改变而改变的场景

这也是状态模式的根本出发点，例如权限设计，人员的状态不同即使执行相同的行为结果也会不同，在这种情况下需要考虑使用状态模式。

2. 条件、分支判断语句的替代者

20.解释器模式：给定一门语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言中的句子。

使用场景：

1. 重复发生的问题可以使用解释器模式
2. 一个简单语法需要解释的场景

21.享元模式：使用共享对象的方法，用来尽可能减少内存使用量以及分享资讯。

```
abstract class flywei{ }
public class Flyweight extends flywei{
    Object obj ;
    public Flyweight(Object obj){
        this.obj = obj;
    }
}
class  FlyweightFactory{
    private HashMap data;
    public FlyweightFactory(){ data = new HashMap<>();}
    public Flyweight getFlyweight(Object object){
        if ( data.containsKey(object)){
            return data.get(object);
        }else {
            Flyweight flyweight = new Flyweight(object);
            data.put(object,flyweight);
            return flyweight;
        }
    }
}
```

使用场景：
1. 系统中存在大量的相似对象。
2. 细粒度的对象都具备较接近的外部状态，而且内部状态与环境无关，也就是说对象没有特定身份。
3. 需要缓冲池的场景。

22.桥梁模式：将抽象和实现解耦，使得两者可以独立地变化。

Circle类将DrwaApi与Shape类进行了桥接，

```
interface DrawAPI {
    public void drawCircle(int radius, int x, int y);
}
class RedCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: red, radius: "
                + radius +", x: " +x+", "+ y +"]");
    }
}
class GreenCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: green, radius: "
                + radius +", x: " +x+", "+ y +"]");
    }
}
abstract class Shape {
    protected DrawAPI drawAPI;
    protected Shape(DrawAPI drawAPI){
        this.drawAPI = drawAPI;
    }
    public abstract void draw();
```

```
}
class Circle extends Shape {
    private int x, y, radius;
    public Circle(int x, int y, int radius, DrawAPI drawAPI) {
        super(drawAPI);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
    public void draw() {
        drawAPI.drawCircle(radius,x,y);
    }
}
//客户端使用代码
Shape redCircle = new Circle(100,100, 10, new RedCircle());
Shape greenCircle = new Circle(100,100, 10, new GreenCircle());
redCircle.draw();
greenCircle.draw();
```

使用场景：

1. 不希望或不适用使用继承的场景
2. 接口或抽象类不稳定的场景
3. 重用性要求较高的场景

23.模板方法模式：定义一个操作中的算法的框架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

使用场景：

1. 多个子类有公有的方法，并且逻辑基本相同时。
2. 重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现。

3. 重构时，模板方法模式是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子函数（见“模板方法模式的扩展”）约束其行为。