设计模式

> 设计经验的总结 解决特定问题的一系列套路 面向对象的设计原则的实际运用

UML

设计模式中要使用的是类图 类图显示模型的静态结构，定义类的名称，属性，方法，以及与其他类之间的关系

软件设计原则

> **开闭原则**：对扩展开放，对修改关闭，类似于电脑热插拔，使用抽象类和接口，提高灵活性和适应性 **里氏替换原则**：任何基类出现的地方，子类一定可以出现，子类可以扩展父类的功能，但不能改变父类原有的功能（使用接口抽象类，不要直接使用继承重写父类的方法）子类一般不要重写父类的方法，尽量直接使用父类的方法即可。 **依赖倒转原则**：高层模块不应该依赖底层模块，面对抽象接口来实现依赖，而不是直接面向类实现依赖 **接口隔离原则**：客户端不应该被迫依赖它不使用的方法，依赖应该建立在最小的接口上（将复杂的接口进行拆分，避免依赖于不必要的方法） **迪米特法则**： 最少知道原则，两个软件实体无需直接通信，则不应该发生直接的相互调用，可以通过第三方转发调用（模块之间应该尽可能的减少依赖，降低代码的耦合度） **合成复用原则**：尽量使用组合或聚合关系来实现，其次才考虑使用继承关系来实现 **单一职责原则**：一个类、接口或方法只负责一个职责，降低代码的复杂度，避免代码变更带来的风险

创建型模式

对象的创建与使用分离，使用者不需要关注对象的创建细节

1. 单例设计模式

确保全局只有一个实例存在，避免资源的浪费，提高系统的性能和一致性。 创建一个类的单一实例

> 饿汉式：类加载就会导致实例的创建，类的实例即使不使用也会在内存中

```
//1.直接初始化属静态性,
public class Singleton {
    //直接初始化属静态性
    private static Singleton singleton = new Singleton();

    private  Singleton() {
        System.out.println("生成一个实例");
    }
    public static Singleton getInstance(){
        return singleton;
    }
}
//2.利用静态代码块创建
public class Singleton2 {

    private static Singleton2 singleton2;
```

```
    //利用静态代码块创建
    static {
        singleton2 = new Singleton2();
    }

    private Singleton2() {
        System.out.println("生成一个实例");
    }
    public static Singleton2 getInstance(){
        return singleton2;
    }
}
```

> 懒汉式：实例的创建会推迟到类使用时

```
//3.线程的不安全方式
public class Singleton3 {
    private Singleton3(){

    }
    private static Singleton3 instance;

    public static Singleton3 getInstance(){
        if(instance == null){
            instance = new Singleton3();
        }
        return instance;
    }
}
//4.普通加锁方式
public class Singleton3 {
    private Singleton3(){

    }
    private static Singleton3 instance;

    public static synchronized Singleton3 getInstance(){
        if(instance == null){
            instance = new Singleton3();
        }
        return instance;
    }
}

//5.DCL方式,多线程情况下可能出现空指针，由于JVM的优化可能导致指令重排序，导致空指针，这里需要添加volatile
public class Singleton5 {
    private Singleton5() {
    }

    private static volatile Singleton5 instance;
```

```
    public static Singleton5 getInstance(){
        //第一次判断，如果instance不为null
        if(instance == null){
            synchronized (Singleton5.class){
                if(instance == null){
                    instance = new Singleton5();
                }
            }
        }
        return instance;
    }
}
//6.静态内部类方式 （最推荐），静态内部类只有在使用时才会去加载
public class Singleton6 {

    private Singleton6() {
    }

    private static class Singleton6Holder{
        //内部类持有外部类
        private final static Singleton6 instance = new Singleton6();
    }
    public static Singleton6 getInstance(){
        //在使用Singleton6Holder时才会加载Singleton6Holder并实例化Singleton6
        return Singleton6Holder.instance;
    }
}
```

静态内部类的加载时间： 静态内部类的加载是在首次使用时才进行的，而不是在外部类加载时就立即加载。静态内部类的加加载是延迟的，只有在程序中引用到该内部类时才会被加载进内存。

> 使用枚举类实现单例模式（利用枚举的安全性，枚举不存在安全性问题）饿汉模式

```
    public enum SingletonEnum {
    //单例枚举
    INSTANCE;
}
```

使用枚举的好处：

- 使用静态成员变量，限制实例个数
- 不会出现并发问题
- 不会被反射破坏单例
- 不会被反序列化破坏单例
- 属于饿汉式
- 可以通过构造方法加入初始化逻辑

> 单例模式的破坏

- 通过序列化和反序列化可以破坏单例模式 （Object io+Serilizable）
- 利用反射开启Construct的访问检查也可以多次创建不同的实例

> 解决单例模式被破坏的问题

```
//序列化破坏问题解决
public class Singleton6 implements Serializable {

    private Singleton6() {
    }

    private static class Singleton6Holder{
        //内部类持有外部类
        private final static Singleton6 instance = new Singleton6();
    }
    public static Singleton6 getInstance(){
        //在使用Singleton6Holder时才会加载Singleton6Holder并实例化Singleton6
        return Singleton6Holder.instance;
    }

    /**
     * 反序列化时，ObjectInputStream会自动调用该方法，解决反序列化时单例失效问题
     * @return
     */
    public Object readResolve(){
        return Singleton6Holder.instance;
    }
}
//反射破坏问题解决
public class Singleton6 implements Serializable {

    private static boolean flag = false;

    private Singleton6() {
        //判断flag的值是否为true，如果是则不是第一个访问，抛出异常,解决反射导致的单例
失效问题
        synchronized (Singleton6.class){
            if(flag){
                throw new RuntimeException("不能创建多个对象");
            }
            flag = true;
        }
    }

    private static class Singleton6Holder{
        //内部类持有外部类
        private final static Singleton6 instance = new Singleton6();
    }
    public static Singleton6 getInstance(){
        //在使用Singleton6Holder时才会加载Singleton6Holder并实例化Singleton6
        return Singleton6Holder.instance;
    }
```

```
    /**
     * 反序列化时，ObjectInputStream会自动调用该方法，解决反序列化时单例失效问题
     * @return
     */
    public Object readResolve(){
        return Singleton6Holder.instance;
    }

}
```

> jdk单例举例：Runtime（java调用系统命令的工具类） spring中创建的bean默认就是单例的

2. 工厂方法模式（生产同种类的产品，只产生同一接口或抽象类的子类）

普通设计方式，添加新的品种咖啡时需要修改工厂类，工厂类是具体的类

```
public abstract class Coffee {

    public abstract String getName();

    public void addSugar(){
        System.out.println("add sugar");
    }

    public void addWater(){
        System.out.println("add water");
    }
}
public class LatteCoffee extends Coffee{
    @Override
    public String getName() {
        return "latte coffee";
    }
}
public class AmerCoffee extends Coffee{
    @Override
    public String getName() {
        return "american coffee";
    }
}
//工厂类
public class CoffeeStore {
    public Coffee orderCoffee(String type){
        Coffee coffee  = null;
        //添加其他总类的类目需要修改这里的代码
        if("american".equals(type)){
            coffee = new AmerCoffee();
        }else if("latte".equals(type)){
            coffee = new LatteCoffee();
        }else{
```

```
            throw new RuntimeException("cannot support this type: "+ type);
        }
        coffee.addSugar();
        coffee.addWater();
        return coffee;
    }
}
```

简单工厂（不属于23设计模式），解耦工厂与实际产品之间的关系

```
public abstract class Coffee {

    public abstract String getName();

    public void addSugar(){
        System.out.println("add sugar");
    }

    public void addWater(){
        System.out.println("add water");
    }
}
public class LatteCoffee extends Coffee{
    @Override
    public String getName() {
        return "latte coffee";
    }
}
public class AmerCoffee extends Coffee{
    @Override
    public String getName() {
        return "american coffee";
    }
}
//添加一个简单工厂类
public class SimpleCoffeeFactory {

    public Coffee createCoffee(String type){
        Coffee coffee  = null;
        if("american".equals(type)){
            coffee = new AmerCoffee();
        }else if("latte".equals(type)){
            coffee = new LatteCoffee();
        }else{
            throw new RuntimeException("cannot support this type: "+ type);
        }
        coffee.addSugar();
        coffee.addWater();
        return coffee;
    }
}
```

```
//最终的输出工厂依赖于简单工厂，不依赖于具体的被生产类
public class CoffeeStore {
    public Coffee orderCoffee(String type){
        SimpleCoffeeFactory factory  = new SimpleCoffeeFactory();
        Coffee coffee = factory.createCoffee(type);
        coffee.addSugar();
        coffee.addWater();
        return coffee;
    }
}
```

静态工厂(非23种设计模式)

```
public class SimpleCoffeeFactory {

    public static Coffee createCoffee(String type){
        Coffee coffee  = null;
        if("american".equals(type)){
            coffee = new AmerCoffee();
        }else if("latte".equals(type)){
            coffee = new LatteCoffee();
        }else{
            throw new RuntimeException("cannot support this type: "+ type);
        }
        coffee.addSugar();
        coffee.addWater();
        return coffee;
    }
}
public class CoffeeStore {
    public Coffee orderCoffee(String type){
        Coffee coffee = SimpleCoffeeFactory.createCoffee(type);
        coffee.addSugar();
        coffee.addWater();
        return coffee;
    }
}
```

以上的几种实现都违背了开闭原则

工厂方法模式（解耦） 定义创建对象的接口，将实际创建工作延迟到子类中，在以下的场景中可以使用：

1. 对象的创建逻辑复杂
2. 类要在运行时决定创建哪个对象
3. 累需要由子类来指定创建对象的具体方式
4. 避免客户端和具体类之间的耦合
5. 动态加载类

> 定一个创建对象的接口，让子类决定实例化哪个产品类对象 角色 抽象工厂，提供接口，提供创建对象的抽象方法 具体工厂 抽象产品，定义产品（对象）的规范 具体产品

- 用户只需要具体工厂名称，不需要知道产品的生产过程
- 系统中添加新的品种时只需要添加具体的产品类和具体的工厂类
- 系统的复杂类会增加

```
public abstract class Coffee {

    public abstract String getName();

    public void addSugar(){
        System.out.println("add sugar");
    }

    public void addWater(){
        System.out.println("add water");
    }
}
public class LatteCoffee extends Coffee{
    @Override
    public String getName() {
        return "latte coffee";
    }
}
public class AmerCoffee extends Coffee{
    @Override
    public String getName() {
        return "american coffee";
    }
}
//抽象工厂类
public interface CoffeeFactory {

    Coffee createCoffee();
}

public class AmerCoffeeFactory implements CoffeeFactory{
    @Override
    public Coffee createCoffee() {
        return new AmerCoffee();
    }
}
public class LatteCoffeeFactory implements CoffeeFactory{
    @Override
    public Coffee createCoffee() {
        return new LatteCoffee();
    }
}
public class CoffeeStore {

    private CoffeeFactory coffeeFactory;

    //具体生产哪种coffee由客户端传入的工厂对象来决定
    public void setCoffeeFactory(CoffeeFactory coffeeFactory) {
```

```
        this.coffeeFactory = coffeeFactory;
    }

    public Coffee orderCoffee(){
        Coffee coffee = coffeeFactory.createCoffee();
        coffee.addSugar();
        coffee.addWater();
        return coffee;
    }
}
```

3. 抽象工厂模式

生产不同产品族，不同产品等级的产品，例如既做裤子，又做上衣，将不同风格的裤子、上衣组合起来组合成运动风、商务风、学生风等 角色

> 抽象工厂 包含多个创建不同产品的方法 具体工厂 具体产品的工厂 抽象产品 定义产品规范 具体产品

- 当一个产品族中的多个对象被设计成一起工作时，能保证客户端始终只使用同一个产品族中的对象
- 当产品族中需要新增一个产品时，所有的工厂类都需要添加相应的产品创建方法
- 一系列对象需要进行相互依赖时，或需要进行相互搭配使用时，可以采用抽象工厂

```
public abstract class Coffee {

    public abstract String getName();

    public void addSugar(){
        System.out.println("add sugar");
    }

    public void addWater(){
        System.out.println("add water");
    }
}
public class LatteCoffee extends Coffee{
    @Override
    public String getName() {
        return "latte coffee";
    }
}
public class AmerCoffee extends Coffee{
    @Override
    public String getName() {
        return "american coffee";
    }
}
public abstract class Dessert {
    public abstract void show();
}
public class MatchMousse extends Dessert{
    @Override
```

```
    public void show() {
        System.out.println("matchMousse");
    }
}
public class Trimisu extends Dessert{
    @Override
    public void show() {
        System.out.println("Trimisu");
    }
}
public interface DessertFactory {

    public Coffee createCoffee();

    public Dessert createDessert();
}
//具体的工厂中组装不同的产品族
public class ItalyDessertFactory implements DessertFactory{
    @Override
    public Coffee createCoffee() {
        return new LatteCoffee();
    }

    @Override
    public Dessert createDessert() {
        return new Trimisu();
    }
}
public class AmerDessertFactory implements DessertFactory{
    @Override
    public Coffee createCoffee() {
        return new AmerCoffee();
    }

    @Override
    public Dessert createDessert() {
        return new MatchMousse();
    }
}
```

简单工厂+配置文件解除耦合(spring的容器实现方式) 通过工厂模式+配置文件的方式解除工厂对象和产品对象的耦合,模仿spring 的IOC容器的实现

- 如果需要扩展新的bean实例，只需要在配置文件中添加一个class的键值对即可。 bean.properties配置文件

```
amer=steven.lee.com.factoryextends.AmerCoffee
latte=steven.lee.com.factoryextends.LatteCoffee
```

简单工厂类

```
public class CoffeeFactory {
    //读取配置文件
    //加载具体的实例
    //放在map中


    private static HashMap<String,Coffee> map = new HashMap<>();
    static{
        Properties properties = new Properties();
        final InputStream resourceAsStream =
CoffeeFactory.class.getClassLoader().getResourceAsStream("bean.properties"
);
        try{
            properties.load(resourceAsStream);
            Set<Object> keys = properties.keySet();
            keys.forEach(key ->{
                String classname = properties.getProperty((String) key);
                //反射创建对象
                Class clazz = null;
                try {
                    clazz = Class.forName(classname);
                    Coffee coffee = (Coffee) clazz.newInstance();
                    map.put((String) key,coffee);
                } catch (InstantiationException | IllegalAccessException |
ClassNotFoundException e) {
                    e.printStackTrace();
                }
            });

        } catch (IOException e) {
            e.printStackTrace();
        }

    }

    public static Coffee createCoffee(String name){
        return map.get(name);
    }
}
```

> jdk 中的Collection使用了工厂方法模式 ,DateFormat、Calendar 都用了工厂模式

- Collection 抽象工厂
- ArrayList 具体工厂
- Iterator 抽象产品
- ArrayList.Itr implements Iterator 具体产品

4. 原型模式

通过一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型对象相同的对象 角色 使用场景：

- 对象的创建成本较高
- 对象的类型不确定
- 对象的构造过程复杂
- 保护对象的状态
- 动态备至对象
- 保持对象的一致性

> 抽象原型类 Cloneable 具体原型类 访问类

- 对象的创建比较复杂，可以使用原型模式快捷的创建对象
- 性能和安全性比较高的场景使用
- 浅克隆： 新对象的属性和原来对象的属性完全相同（引用对象一样）
- 深克隆 : 新对象的引用对象也会被克隆

最简单情况

```
public class RealizeType implements Cloneable{

    public RealizeType() {
    }

    @Override
    protected RealizeType clone() throws CloneNotSupportedException {
        System.out.println("原型对象复制成功");
        return (RealizeType) super.clone();
    }
}
public class RealizeTypeMain {
    public static void main(String[] args) throws
CloneNotSupportedException {
        RealizeType realizeType1 = new RealizeType();
        RealizeType realizeTypeClone = realizeType1.clone();
        System.out.println(realizeType1 ==realizeTypeClone );
    }
}
```

浅克隆--属性不进行克隆

```
package steven.lee.com.prototype.shallow;

/**
 * @Description:
 * @CreateDate: Created in 2023/3/2 20:19
 * @Author: lijie3
 */
public class Citation implements Cloneable{
```

```
    private String name;

    public String getName() {
        return name;
    }

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

    public void show(){
        System.out.println(name + "is a citation");
    }

    @Override
    protected Citation clone() throws CloneNotSupportedException {
        return (Citation) super.clone();
    }
}
public class CitationMain {

    public static void main(String[] args) throws
CloneNotSupportedException {
        Citation citation = new Citation();
        //先克隆后赋值
        Citation citation1 = citation.clone();

        citation.setName("zhangsan");

        citation1.setName("lisi");
    }
}
```

深克隆 //对引用属性也要进行克隆

```
public class Citation implements Cloneable{

    private Student student;

    public Student getStudent() {
        return student;
    }

    public void setStudent(Student student) {
        this.student = student;
    }

    public void show(){
        System.out.println(student.getName() + "is a citation");
    }

    @Override
```

```
    protected Citation clone() throws CloneNotSupportedException {

        Citation citation =  (Citation) super.clone();
        //克隆时要将引用对象也要进行克隆
        citation.setStudent(student.clone());
        return citation;
    }
}

public class Student implements Cloneable{
    private String name;

    public String getName() {
        return name;
    }

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

    @Override
    protected Student clone() throws CloneNotSupportedException {
        return (Student) super.clone();
    }
}
public class DeepMain {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Student student = new Student();
        student.setName("steven ");
        Citation citation = new Citation();
        citation.setStudent(student);
        Citation citation1 = citation.clone();
        citation.show();
        citation1.show();
        student.setName("steven1 ");
        //citation1的student未被修改，已被克隆
        citation1.show();
        System.out.println(citation == citation1);
        System.out.println(citation.getStudent() ==
citation1.getStudent());
    }
}
//另外使用ObjectInputStream/ObjectOutputStream的序列化/反序列化也可以实现原型模式
```

5.构建者模式

将一个复杂的对象的构建与表示分离，使得用相同的构建过程可是创建不同的表示 分离了的部件的构造由builder来负责，由Director来装配

- 适用于某个对象构建过程复杂的情况
- 实现构建和装配的解耦，一步一步创建一个复杂的对象

角色

> 抽象建造者buidler，规定要实现复杂对象的哪些部分的创建 具体建造者 实现builder接口 产品类 要创建的复杂对象 指挥者类 director ，调用具体的建造者来实现类的装配

- 封装性好，将主要组装过程逻辑封装在指挥者中，当建造过程发生变更，只需要在指挥者中添加相应的建造逻辑
- 更加精细的控制创建的过程
- 产品本身与产品的创建过程解耦，使用相同的创建过程可以创建不同的产品对象
- 容易扩展，有新的需求，通过实现一个新的建造者类就可以完成
- 当产品之间的差异过大不适合使用建造者模式

```
//产品信息
public class Bike {

    private String frame;

    private String seat;

    public String getFrame() {
        return frame;
    }

    public void setFrame(String frame) {
        this.frame = frame;
    }

    public String getSeat() {
        return seat;
    }

    public void setSeat(String seat) {
        this.seat = seat;
    }
}
//建造者
public abstract class Builder {

    protected  Bike bike = new Bike();

    public abstract Builder builderFrame();
    public abstract Builder builderSeat();
    public abstract Bike createBike();
}
public class MobikeBuilder extends Builder{


    @Override
    public Builder builderFrame() {
        bike.setFrame("mobike frame");

        return this;
```

```
    }

    @Override
    public Builder builderSeat() {
        bike.setSeat("mobike seat");
        return this;
    }

    @Override
    public Bike createBike() {
        return new Bike();
    }
}
public class OfoBuilder extends Builder{
    @Override
    public Builder builderFrame() {
        bike.setFrame("ofo frame");

        return this;
    }

    @Override
    public Builder builderSeat() {
        bike.setSeat("ofo seat");
        return this;
    }

    @Override
    public Bike createBike() {
        return bike;
    }
}
//指挥者，控制调用的先后顺序并生成最终的产品
public class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder = builder;
    }

    public Bike construct(){
        return builder.builderFrame().builderSeat().createBike();
    }
}
```

- 可以合并指挥者和建造者

```
public class OfoBuilder extends Builder{
    @Override
    public Builder builderFrame() {
        bike.setFrame("ofo frame");
```

```
            return this;
        }

        @Override
        public Builder builderSeat() {
            bike.setSeat("ofo seat");
            return this;
        }

        @Override
        public Bike createBike() {
    //        return bike;
            return this.builderFrame().builderSeat().createBike();
        }
    }
```

扩展 当一个类创建时需要传入很多参数时，如果使用全参数的构造器，可读性会很差 可以使用建造者模式来实现

```
public class LombokBuilder {

    private int id;

    private String name;

    public LombokBuilder() {
    }

    public LombokBuilder(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public static Builder builder(){
        return new Builder();
    }

    public static class Builder {
        private int id;

        private String name;

        public Builder id(int id){
            this.id = id;
            return this;
        }

        public Builder name(String name){
            this.name = name;
            return this;
        }
```

```
        public LombokBuilder build(){
            return new LombokBuilder(this.id,this.name);
        }
    }

    @Override
    public String toString() {
        return "LombokBuilder{" +
                "id=" + id +
                ", name='" + name + '\'' +
                '}';
    }

    public static void main(String[] args) {
        LombokBuilder lombokBuilder =
LombokBuilder.builder().id(1).name("dxy").build();
        System.out.println(lombokBuilder);
    }
}
```

结构型模式

用于描述如何将类或对象按照某种布局组成更大的结构,分为类结构型模式和对象结构型模式 组合和聚合关系比继承关系耦合度更低，满足合成复用原则，对象结构型模式比类结构型模式具有更大的灵活性

6.代理模式

由于某些原因需要给对象提供一个代理提供对被代理对象的控制 使用场景： 控制对象访问 为对象提供额外的功能和逻辑，提供系统的安全性和性能

- java 中代理按照代理类的生成时机不同又分为静态代理和动态代理

角色

> 抽象主题类Subject 真是主题类Real Subject 代理类 Proxy 提供与真实主题相同的接口，提供对真实访问主题的访问控制和功能扩充

- 动态代理相比静态代理可以将所有的被代理类或接口的方法都放在代理类中的同一个方法中进行集中处理
- 被代理类增加一个方法，动态代理类中不需要添加额外的代理方法，而静态代理必须添加一个新的代理方法来支持代理
- 代理模式和客户端与目标独享之间起到了一个终结作用和保护目标对象的作用
- 代理对象可以扩展目标的功能
- 将客户端与目标对象分离，在一定程度上降低系统耦合度
- 增加了系统的复杂度
- 使用场景：远程代理（rpc请求），防火墙代理，保护代理

静态代理

```
public interface SellTickets {

    void sell();
}
//具体实现类
public class TrainStation implements SellTickets{
    @Override
    public void sell() {
        System.out.println("Train station sell tickets");
    }
}
//静态代理类
public class ProxyPoint implements SellTickets{
    private TrainStation trainStation = new TrainStation();
    @Override
    public void sell() {
        System.out.println("proxy get tips");
        trainStation.sell();
    }
}
//客户端
public class Main {

    public static void main(String[] args) {
        ProxyPoint proxyPoint = new ProxyPoint();
        proxyPoint.sell();
    }
}
```

jdk动态代理

> Proxy 生成动态代理的工具 newProxyInstance方法参数

- ClassLoader:类加载器
- Class<>[] 被代理类实现的接口class类型
- Invocationhandler

> InvocationHandler中的invoke参数

- proxy 生成的代理对象本身
- Method method 被代理执行的方法
- Object[] args 被代理对象方法执行时的传参

> 使用arthas分析生成的代码，可以看到动态生成了一个`$Proxy`的类 `$Proxy`实现了SellTickets接口，并且继承了Proxy类（这是jdk动态代理只能代理接口的原因） `$Proxy`中持有了一个h参数（从Proxy继承过来，h的即在newProxyInstance方法中指定的InvocationHandler） 客户端拿到的实际上就是$Proxy对象，执行该对象的sell方法会调用h的invoke方法进而执行station的sell方法

```
public class ProxyFactory {

    private TrainStation station = new TrainStation();
    public SellTickets getProxyObject(){
        //返回代理对象
        SellTickets sellTickets = (SellTickets)
Proxy.newProxyInstance(station.getClass().getClassLoader(),
                station.getClass().getInterfaces(),
                new InvocationHandler() {
                    /**
                     *  代理执行方法
                     */
                    @Override
                    public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
                        System.out.println("jdk动态代理执行");
                        return method.invoke(station,args);
                    }
                });
        return sellTickets;

    }
}
```

gpt -- Java 的动态代理要求被代理的类必须实现接口，主要是由于以下原因：

继承关系限制： Java 的动态代理是基于接口实现的，它利用了代理类和被代理类之间的继承关系。代理类实现了被代理接口，可以将代理类视为被代理接口的子类。如果被代理类没有实现接口，就无法建立这种继承关系，动态代理的机制就无法实现。

方法调度限制： 动态代理的核心机制是通过代理类的方法调用来拦截、增强或重写被代理对象的方法。由于 Java 是单继承语言，一个类只能继承一个父类，但是可以实现多个接口。因此，通过接口实现动态代理可以避免单继承带来的限制，让代理类能够同时拦截多个接口中的方法。

类型兼容性： 接口可以看作是一种规范或契约，实现了某个接口的类保证了其具有该接口定义的方法。代理类实现了被代理接口，因此可以完全替代被代理类，保持类型兼容性，使得代码在使用代理时不需要修改。

总结起来，Java 动态代理要求被代理的类实现接口，主要是为了建立代理类和被代理类之间的继承关系、支持多接口的方法调度以及保持类型兼容性。如果不需要实现接口来实现代理，可以考虑使用字节码操作库来生成代理类，但相对于 Java 标准的动态代理机制来说，会更加复杂和底层。

cglib动态代理

> cglib可以在内存中动态生成类的代理

```
public class Transtation {
    public void sell(){
        System.out.println("Transtation sell tickets");
    }
}
```

```
public class ProxyFactory implements MethodInterceptor {
    private Transtation transtation = new Transtation();
    public Transtation getProxyObject(){
        //创建enhancer对象，与jdk的proxy类类似
        Enhancer enhancer = new Enhancer();
        //设置父类
        enhancer.setSuperclass(Transtation.class);
        //设置回调函数
        enhancer.setCallback(this);
        //创建代理对象
        Transtation o = (Transtation) enhancer.create();
        return o;
    }

    @Override
    public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
        System.out.println("cglib invoke");
        return method.invoke(transtation,objects);
    }
}
public class Main {

    public static void main(String[] args) {
       ProxyFactory proxyFactory = new ProxyFactory();
        Transtation transtation = proxyFactory.getProxyObject();
        transtation.sell();
    }
}
```

7 适配器模式

> 将一个类的接口转换成客户端希望的另一个接口，使得原本由于接口不兼容而不能一起工作的类能一起工作 分为类适配器（使用类继承）和对象适配器（组合、聚合），前者耦合度高，后者实际使用中会多一点

使用场景： 系统接口升级，当系统的接口发生变化或升级时，为了保持旧代码的兼容性，使用适配器来衔接新旧接口之间的差异 类库使用，当需要使用某个类库中的组件，库中的接口与其他系统部分呢不兼容时，可以使用适配器 不同协议转换：在不同的通信协议之间进行转换 功能扩展：需要对现有功能进行扩展定制，不希望修改现有代码时使用适配器

角色

- 目标 当前系统业务所期待的接口，可以是抽象类或接口
- 适配者 是被访问和适配的现存组件库中的组件接口
- 适配器类 转换器，通过继承或引用适配者对象，把适配者接口转换成目标接口

基于继承关系实现的适配器

> 由于java的单继承特性，当目标是一个具体的类不是接口时，无法使用这种方式 违背了合成复用原则

```
//原有接口
public interface TFCard {

    String readTf();

    void writeTf(String content);
}

public class TFCardImpl implements TFCard{
    @Override
    public String readTf() {
        return "tfcard read hello world";
    }

    @Override
    public void writeTf(String content) {
        System.out.println("tf card write:" + content);
    }
}
//目标接口
public interface SDCard {

    String readSd();

    void writeSd(String content);
}
//目标实现类，无法满足现有系统对适配者接口的要求
public class SDCardImpl implements SDCard{

    @Override
    public String readSd() {
        return "sd card read hello world";
    }

    @Override
    public void writeSd(String content) {
        System.out.println("sd card write: "+ content);
    }
}
//基于类继承的适配器
public class SDCardAdapter extends TFCardImpl implements SDCard{


    @Override
    public String readSd() {
        return readTf();
    }

    @Override
    public void writeSd(String content) {
        writeTf(content);
    }
}
```

对象适配器模式 对象适配器模式采用聚合放方式，将适配者接口聚合到适配器中 解决了类适配器存在的两个问题

```
//对象聚合方式实现适配器
public class SDCardAdapter2 implements SDCard{
    private TFCard tfCard;

    @Override
    public String readSd() {
        return tfCard.readTf();
    }

    @Override
    public void writeSd(String content) {
        tfCard.writeTf(content);
    }
}
//client
public class ComputerMain {

    //从sd card中读取数据

    public String readSd(SDCard sdCard){
        if(sdCard == null){
            throw  new NullPointerException("sd card should not null");
        }
        return sdCard.readSd();
    }

    public void write(SDCard sdCard,String content){
        if(sdCard == null){
            throw  new NullPointerException("sd card should not null");
        }
        sdCard.writeSd(content);
    }

    public static void main(String[] args) {
        ComputerMain computerMain = new ComputerMain();
        //不使用适配器情况
        String message = computerMain.readSd(new SDCardImpl());
        System.out.println(message);

        //类适配器（少用）
        message = computerMain.readSd(new SDCardAdapter());
        System.out.println(message);
        //对象适配器
        message = computerMain.readSd(new SDCardAdapter2(new
TFCardImpl()));
        System.out.println(message);
    }
}
```

使用场景

- 以前开发的系统存在满足新系统功能需求的类，但旧接口同新系统的接口不一致
- 使用第三方的组件，但组件接口定义和自己要求的接口定义不同 jdk中的使用

> Reader、inputStream 的适配使用的是InputStreamReader,在底层使用的适配器是StreamDecoder，使用的是对象适配器

8 装饰者模式

在不改变现有对象结构的情况下，动态的给该对象增加一些职责（增加额外的功能）

角色

> 抽象构件 定义一个抽象接口以规范准备接收附加责任的对象 具体构建 实现抽象构件，通过装饰角色为其添加一些职责 抽象装饰 继承或实现抽象构件，并包含具体的构件的实例，通过其子类扩展构建的功能 具体装饰 实现抽象装饰的相关方法，并给具体构件对象添加附加的责任

好处

- 装饰者带来比继承更好的灵活性，扩展更加方便，通过组合不同的装饰者对象来获取具有不同行为状态的多样化结果，完全准许开闭原则
- 装饰类和呗装饰类可以独立扩展，不会发生耦合

场景 当继承方式对系统进行扩展或采用继承不利于系统宽展和维护时使用 final类的定义不能继承，这时可以使用装饰者模式 功能的动态扩展：当需要对对象的功能进行动态扩展或增强时使用 功能组合：对对象的功能进行组合或串联时 避免子类化 动态添加或删除功能 保持接口一致性

jdk中的装饰者示例 在IO中大量使用如BufferedWriter 使用装饰者模式提供咖啡

```
// 抽象组件
interface Coffee {
    String getDescription();
    double cost();
}

// 具体组件 - 浓缩咖啡
class Espresso implements Coffee {
    @Override
    public String getDescription() {
        return "Espresso";
    }

    @Override
    public double cost() {
        return 1.99;
    }
}

// 装饰器抽象类
```

```
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double cost() {
        return decoratedCoffee.cost();
    }
}

// 具体装饰器 - 牛奶
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    @Override
    public double cost() {
        return decoratedCoffee.cost() + 0.5;
    }
}

// 具体装饰器 - 摩卡
class MochaDecorator extends CoffeeDecorator {
    public MochaDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Mocha";
    }

    @Override
    public double cost() {
        return decoratedCoffee.cost() + 1.0;
    }
}

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

```
        // 创建一个浓缩咖啡
        Coffee espresso = new Espresso();
        System.out.println("Description: " + espresso.getDescription());
        System.out.println("Cost: $" + espresso.cost());

        // 用牛奶装饰浓缩咖啡
        Coffee milkEspresso = new MilkDecorator(espresso);
        System.out.println("Description: " +
milkEspresso.getDescription());
        System.out.println("Cost: $" + milkEspresso.cost());

        // 用摩卡和牛奶装饰浓缩咖啡
        Coffee mochaMilkEspresso = new MochaDecorator(milkEspresso);
        System.out.println("Description: " +
mochaMilkEspresso.getDescription());
        System.out.println("Cost: $" + mochaMilkEspresso.cost());
    }
}
```

```
//抽象构建角色
public abstract class FriedFood {

    private float price;

    private String desc;

    public FriedFood() {
    }

    public FriedFood(float price, String desc) {
        this.price = price;
        this.desc = desc;
    }

    public float getPrice() {
        return price;
    }

    public void setPrice(float price) {
        this.price = price;
    }


    public String getDesc() {
        return desc;
    }

    public void setDesc(String desc) {
        this.desc = desc;
    }
```

```
    public abstract  float cost();

}
//具体构建角色1
public class FriedRice extends FriedFood{


    public FriedRice() {
        this(10,"炒饭");
    }

    public FriedRice(float price, String desc) {
        super(price, desc);
    }



    @Override
    public float cost() {
        return getPrice();
    }
}

//具体构建角色2
public class FriedNoodles extends FriedFood{
    public FriedNoodles() {
        this(10,"炒面");
    }

    public FriedNoodles(float price, String desc) {
        super(price, desc);
    }

    @Override
    public float cost() {
        return getPrice();
    }
}

//装饰者类，抽象装饰角色
public abstract class Garnish  extends FriedFood{
    //声明快餐类变量

    private FriedFood friedFood;

    public FriedFood getFriedFood() {
        return friedFood;
    }

    public Garnish() {
    }

    public void setFriedFood(FriedFood friedFood) {
```

```
        this.friedFood = friedFood;
    }

    public Garnish(float price, String desc, FriedFood friedFood) {
        super(price, desc);
        this.friedFood = friedFood;
    }
}
//具体装饰者角色1
public class Egg extends Garnish{
    public Egg(FriedFood friedFood) {
        super(1,"egg", friedFood);
    }

    @Override
    public float cost() {
        return getPrice() + getFriedFood().cost();
    }

    @Override
    public String getDesc() {
        return getFriedFood().getDesc()+"+"+super.getDesc()  ;
    }
}
//具体装饰者角色2
public class Bacon extends Garnish{
    public Bacon(FriedFood friedFood) {
        super(2,"bacon",friedFood);
    }

    @Override
    public float cost() {
        return getPrice()+ getFriedFood().cost();
    }
    @Override
    public String getDesc() {
        return getFriedFood().getDesc()+"+"+super.getDesc()  ;
    }
}
//客户算使用时可以进行多次反复的装饰
public class Main {
    public static void main(String[] args) {
        FriedFood friedRice = new FriedRice();
        System.out.println(friedRice.getDesc()+",price:"+
friedRice.cost());
        friedRice = new Egg(friedRice);
        System.out.println(friedRice.getDesc()+",price:"+
friedRice.cost());
        //再次装饰，再加一个蛋
        friedRice = new Egg(friedRice);
        System.out.println(friedRice.getDesc()+",price:"+
friedRice.cost());
        friedRice = new Bacon(friedRice);
        System.out.println(friedRice.getDesc()+",price:"+
```

```
friedRice.cost());
    }
}
```

代理模式（静态）与装饰者模式的区别联系

- 都要实现与目标类相同的业务接口
- 两个类都需要声明目标对象
- 都可以在不修改目标类的前提下增强目标方法
- 使用目的不同，装饰者是增强目标对象
- 静态代理是保护和隐藏目标对象
- 聚合的对象不同，静态代理代理的是具体的实现类（在类定义时就已经确定了使用哪个具体的被代理对象，隐藏了目标独享），而装饰者装饰者接口（在使用时才能决定传入哪个具体的实现类） **gpt的回答：** 目的不同：代理模式的主要目的是控制对对象的访问，而装饰者模式的主要目的是在不改变原有对象结构的情况下，给对象添加新的行为或责任。 职责不同：代理模式主要关注与代理对象和被代理对象的关系，代理对象可以对被代理对象的访问进行控制，可以在访问被代理对象之前或之后进行一些额外的操作。而装饰者模式主要关注于给对象添加新的行为或责任，装饰者对象在执行自己的行为之前或之后可以调用被装饰对象的行为。 关注点不同：代理模式关注于两个对象之间的关系，被代理对象对于代理对象是透明的；而装饰者模式关注于给对象动态地添加行为，对于被装饰对象来说，装饰者对象是透明的。

实现方式不同：代理模式通常通过创建一个代理对象来控制对被代理对象的访问，代理对象通常持有被代理对象的引用；而装饰者模式通常通过创建一个装饰者对象来包裹被装饰对象，装饰者对象通常持有被装饰对象的引用。

9. 桥接模式

抽象与实现分离，使他们可以独立变化，使用组合关系代理继承关系，降低抽象和耦合的维度 两个不同层面的功能进行桥接 桥接模式的核心思想是将一个类的抽象部分与其具体实现部分分离，使它们可以独立地变化而不会相互影响。（GPT）

> 类功能层次与类实现层次的之间搭建桥梁 类功能层次：使用继承的方式，扩展基类的功能，比如A有a()方法，B继承A同时B又扩展了b()方法，C继承B又扩展了c（）方法 类实现层次：重写基类方法但不添加新的方法 **在画图的过程中，图形又颜色和形状两个维度，通过桥接模式，我们可以在不影响彼此的情况下组合不同的形状和颜色。这样，我们可以轻松地添加新的形状或颜色，而不需要修改已有的代码。 角色** 抽象部分（Abstraction）：抽象部分定义了高层的接口，并维护一个指向实现部分的引用。它可以是抽象类或接口。

实现部分（Implementor）：实现部分定义了低层的接口，它与抽象部分具有独立的继承关系。通常，实现部分也是一个抽象类或接口。

具体抽象类（Refined Abstraction）：具体抽象类是抽象部分的子类，它通过扩展抽象部分，可以添加更多的功能。

具体实现类（Concrete Implementor）：具体实现类是实现部分的子类，它实现了实现部分的接口，提供了具体的功能实现。

```
//功能层次类，扩展了display方法
public class Display {
    //持有类实现层次的基类
    private Displaybase displaybase;

    public Display(Displaybase displaybase) {
        this.displaybase = displaybase;
    }

    public void open(){
        displaybase.rawOpen();
    }

    public void print(){
        displaybase.rawPrint();
    }

    public void close(){
        displaybase.rawClose();
    }

    public final void display(){
        open();
        print();
        close();
    }

}
//类功能层次，扩展了multiDisplay方法
public class CountDisplay extends Display{
    public CountDisplay(Displaybase displaybase) {
        super(displaybase);
    }
    public void multiDisplay(int times){
        open();
        for (int i = 0; i < times; i++) {
            print();
        }
        close();
    }
}
//类实现层次基类
public abstract class Displaybase {

    public abstract void rawOpen();
    public abstract  void rawPrint();
    public abstract  void rawClose();
}
//类实现层次
```

```
public class DisplaybaseImpl extends Displaybase {

    private String string;
    private int width;

    public DisplaybaseImpl(String string) {
        this.string = string;
        this.width = string.length();
    }

    @Override
    public void rawOpen() {
        printLine();
    }

    @Override
    public void rawPrint() {
        System.out.println("|" + string + "|");
    }

    @Override
    public void rawClose() {
        printLine();
    }

    private void printLine(){
        System.out.print("+");
        for (int i = 0; i < width; i++) {
            System.out.print("-");
        }
        System.out.println("+");
    }
}
```

画图桥接

```
public interface Shape {
    void draw();
}
public interface Color {
    void applyColor();
}
public class Circle implements Shape {
    private Color color;

    public Circle(Color color) {
        this.color = color;
    }

    @Override
    public void draw() {
        System.out.print("Drawing a circle ");
```

```
            color.applyColor();
        }
    }

    public class Rectangle implements Shape {
        private Color color;

        public Rectangle(Color color) {
            this.color = color;
        }

        @Override
        public void draw() {
            System.out.print("Drawing a rectangle ");
            color.applyColor();
        }
    }
    public class RedColor implements Color {
        @Override
        public void applyColor() {
            System.out.println("with red color.");
        }
    }

    public class BlueColor implements Color {
        @Override
        public void applyColor() {
            System.out.println("with blue color.");
        }
    }
    public class BridgePatternDemo {
        public static void main(String[] args) {
            Shape redCircle = new Circle(new RedColor());
            Shape blueRectangle = new Rectangle(new BlueColor());

            redCircle.draw(); // Output: Drawing a circle with red color.
            blueRectangle.draw(); // Output: Drawing a rectangle with blue
    color.
        }
    }
```

好处：

> 提供系统的可扩展性，当两个变化为度中任意一个维度进行扩展，都不需要修改原有系统，只需要添加类实现层次或是类的功能层次结构中继续添加实现类即可 实现细节对客户端透明

使用场景: 当系统中不希望使用基层或因为多层次基层导致系统的类大量增加 当类存在两个维度独立变化，且这两个维度都需要扩展 实现部分有多个变化为度的场景 举例：在spring的BeanFactory设计中，扩展类WebApplicationContext持有了BeanFactory的一个层次实现类，实现了两个不同维度容器的桥接

10. 外观（门面（facade））模式

通过为多个复杂的子系统提供一个一致的接口，使得子系统更容易被访问的模式 ---迪米特法则 为相互关联的在一起的错综复杂的类整理出一个高层接口 角色： 外观角色：为多个子系统提供一个统一简单的接口 子系统角色：实现系统的部分功能，客户通过外观角色访问它

```
//子系统1
public class Light {
    public void on(){
        System.out.println("on light");
    }

    public void off(){
        System.out.println("off light");
    }
}
//子系统2
public class TV {

    public void on(){
        System.out.println("on tv");
    }

    public void off(){
        System.out.println("off tv");
    }
}
//子系统3
public class AirCondition {
    public void on(){
        System.out.println("on AirCondition");
    }

    public void off(){
        System.out.println("off AirCondition");
    }
}
//门面类，外观类，用户与该类对象交互
public class SmartApplianceFacade {

    private Light light;

    private TV tv;

    private AirCondition airCondition;

    public SmartApplianceFacade() {
        light = new Light();
        tv =new TV();
        airCondition = new AirCondition();
    }

    public void say(String message){
        if(message.contains("on")){
```

```
            on();
        } else if(message.contains("off")){
            off();
        }else{
            System.out.println("error");
        }

    }
    public  void on(){
        light.on();
        tv.on();
        airCondition.on();
    }
    public void off(){
        light.off();
        tv.off();
        airCondition.off();
    }
}
public class Main {
    public static void main(String[] args) {
        SmartApplianceFacade facade = new SmartApplianceFacade();
        facade.say("on");
        facade.say("off");
    }
}
```

优缺点：

- 降低子系统与客户端之间的耦合度
- 降低客户访问各个子系统的难度
- 不符合开闭原则，修改麻烦

实例：

- ServletRequest使用HttpServletRequest作为门面类
- slf4j 作为log4j的门面，在编程时只需要使用slf4j提供的接口即可

11.组合模式

在树形结构中，要保持节点之间的一致性，可以将节点和叶子节点定义保持一致可以很方便的操作各个节点把一组相似的对象作为单一的对象，依据树型结构来组合对象

角色

> 组件（Component）：定义组合中对象的共同接口，可以包括叶子对象和组合对象的操作。 叶子（Leaf）：实现组件接口，表示树的末端对象，没有子对象。 组合（Composite）：实现组件接口，表示包含子对象的对象。它可以包含叶子对象和其他组合对象，形成一个递归结构。

```
public abstract class MenuComponent {
```

```
    protected String name;

    protected int level;


    public MenuComponent(String name, int level) {
        this.name = name;
        this.level = level;
    }

    public  void add(MenuComponent component){
        throw new UnsupportedOperationException("not support add");
    }

    public void remove(MenuComponent component){
        throw new UnsupportedOperationException("not support remove");
    }

    public MenuComponent getChild(int index){
        throw new UnsupportedOperationException("not support get");
    }


    public String getName() {
        return name;
    }

    public int getLevel() {
        return level;
    }

    //打印
    public abstract void print();
}
//树枝节点
public class Menu extends MenuComponent{

    private List<MenuComponent> menuComponentList = new ArrayList<>();
    public Menu(String name, int level) {
        super(name, level);
    }

    @Override
    public void add(MenuComponent component) {
        this.menuComponentList.add(component);
    }

    @Override
    public void remove(MenuComponent component) {
        this.menuComponentList.remove(component);
    }

    @Override
    public MenuComponent getChild(int index) {
```

```
            return this.menuComponentList.get(index);
        }

        @Override
        public void print() {
            System.out.println(name);
            for (int i = 0; i < menuComponentList.size(); i++) {
                menuComponentList.get(i).print();
            }
        }
    }
    //叶子结点
    public class MenuItem extends MenuComponent {

        public MenuItem(String name, int level) {
            super(name, level);
        }

        @Override
        public void print() {
            System.out.println("name:" + name + ",level:" + level);
        }
    }
    //保持访问的一致性
    public class Main {
        public static void main(String[] args) {
            MenuComponent menuComponent = new Menu("菜单管理",2);
            menuComponent.add(new MenuItem("用户页面",3));
            menuComponent.add(new MenuItem("配置页面",3));
            menuComponent.add(new MenuItem("文档页面",3));
            MenuComponent menuComponent2 = new Menu("用户管理",2);
            menuComponent2.add(new MenuItem("添加用户",3));
            menuComponent2.add(new MenuItem("修改用户",3));
            menuComponent2.add(new MenuItem("删除用户",3));
            MenuComponent menuComponent3 = new Menu("角色管理",2);
            menuComponent3.add(new MenuItem("添加角色",3));
            menuComponent3.add(new MenuItem("修改角色",3));
            menuComponent3.add(new MenuItem("删除角色",3));
            MenuComponent root = new Menu("后台管理",1);
            root.add(menuComponent);
            root.add(menuComponent2);
            root.add(menuComponent3);
            root.print();
        }
    }
```

优缺点

- 清晰定义分层次的复杂对象
- 客户端可以一致的使用其中的单个对象，不需要关系是单个对象还是整个组合结构
- 添加子节点时不需要修改节点类

12.享元模式

运用共享技术来有效的支持大量细粒度的对象复用，提高代码的复用 区分外部状态（非享元部分）和内部（享元）部分 角色

> 抽象享元角色： 定义了具有共享状态的对象的接口。通过这个接口，可以在不同的上下文中共享对象。 具体享元角色：实现了享元接口，表示具体的享元对象。它包含了内部状态，并且可以被共享。 享元工厂：负责创建和管理享元对象。它维护一个享元池（或缓存），用于存储已经创建的享元对象，以便于复用。

```
//抽象享元
public abstract class AbstractBox {

    public abstract  String getShape();

    public void display(String color){
        System.out.println("方块形状:"+ getShape() +",颜色:"+color);
    }
}
public class IBox extends AbstractBox{
    @Override
    public String getShape() {
        return "I";
    }
}
public class LBox extends AbstractBox{
    @Override
    public String getShape() {
        return "L";
    }
}
public class OBox extends AbstractBox{
    @Override
    public String getShape() {
        return "O";
    }
}
//享元工厂
public class BoxFactory {

    private static BoxFactory factory = new BoxFactory();

    private HashMap<String,AbstractBox> map;

    private BoxFactory() {
        this.map = new HashMap<>();
        map.put("I",new IBox());
        map.put("O",new OBox());
        map.put("L",new LBox());
    }

    public static BoxFactory getInstance(){
```

```
            return factory;
        }

        public AbstractBox getShape(String name){
            return map.get(name);
        }
    }
    //对可以共用的部分提供享元，对不可共用的部分提供修改的接口
    public class Main {

        public static void main(String[] args) {
            BoxFactory factory = BoxFactory.getInstance();
            AbstractBox box1 = factory.getShape("I");
            box1.display("green");
            //两次用到的box是同一个对象
            AbstractBox box2 = factory.getShape("O");
            box2.display("red");
            AbstractBox box3 = factory.getShape("O");
            box3.display("pink");
            System.out.println(box2 == box3);

        }
    }
```

优缺点：

- 极大的减少内存中相似或相同的对象数量，节省系统资源
- 外部状态相对独立，不影响内部状态
- 分离内部状态和外部状态导致系统的复杂度增加 使用场景
- 当一个系统中有大量相同或相似的对象，造成内存的大量消耗
- 对象的大部分状态都可以外部化，将这些外部状态传入对象中
- 在使用享元模式需要维护一个存储享元对象的享元池，需要消耗一定的资源，需要频繁的访问享元时才使用享元

举例 jdk中的Integer（基本数据类型）、 使用了享元模式

```
    Integer i1 = 127;
        Integer i2 = 127;
        System.out.println(i1==i2); //true,Integer.valueof()中使用的
IntegerCache将-128~127这个范围内进行了缓存

        Integer i3 = 128;

        Integer i4= 128;
        System.out.println(i3==i4);//false,IntegerCache将-128~127这个范围内进
行了缓存,大于127部进行缓存
```

行为型模式

用于描述类或对象之间怎样写作共同完成单个对象无法单独完成的任务，以及怎样分配职责

13.模版方法模式

在一个算法设计中，已经确定了算法执行的所有关键步骤，而且确定了所有步骤执行的顺序，这时可以使用一个包含模版方法的抽象类，让子类实现具体的步骤 角色

> 抽象类，包含已实现的模版方法和基本方法（可以时抽象或具体方法） 具体类 实现具体的步骤处理方法

```
//抽象模版类
public abstract class AbstractCook {

    public abstract void pourOil();

    public abstract  void heatOil();

    public abstract void fry();


    public void cookProcess(){
        pourOil();
        heatOil();
        fry();
    }
}
//实现具体步骤方法的类
public class BaocaiCook extends AbstractCook{
    @Override
    public void pourOil() {
        System.out.println("pourOil");
    }

    @Override
    public void heatOil() {
        System.out.println("heatOil");
    }

    @Override
    public void fry() {
        System.out.println("fry baocai");
    }
}
public class Main {
    public static void main(String[] args) {
        AbstractCook baocaiCook = new BaocaiCook();
        baocaiCook.cookProcess();
    }
}
```

优缺点

- 提高代码复用性
- 将部分的代码放在抽象父类中，不需要放入不同的子类中
- 实现了反转控制
- 通过父类调用其子类的操作，扩展了子类的具体行为，复合开闭原则
- 对每个不同的实现都要定义一个子类，系统更加庞大
- 父类的抽象方法由子类实现，代码的降低

场景

- 实现整体步骤固定，各个步骤有区别的流程
- 需要通过子类来决定父类中的某个步骤是否需要执行

InputStream类中的read（）都是在父类中定义的抽象方法，在子类中实现具体的read逻辑

14. 策略模式

定义一系列算法，并将算法封装起来，可以相互替换，不影响使用算法的客户 角色

> 抽象策略类 定义给出所有具体策略所需的接口 具体策略类 实现算或行为 环境类 Context ， 策略引用上下文，给客户端调用

```
//策略接口
public abstract class Strategy {
    public abstract void show();
}
//具体策略1
public class StrategyA extends Strategy{
    @Override
    public void show() {
        System.out.println("use Strategy A");
    }
}
//具体策略2
public class StrategyB extends Strategy{
    @Override
    public void show() {
        System.out.println("use Strategy B");
    }
}
//策略上下文
public class StrategyContext {
    private Strategy strategy;

    public StrategyContext(Strategy strategy) {
        this.strategy = strategy;
    }
```

```
    public void show(){
        strategy.show();
    }
}
//通过上下文使用策略
public class Main {

    public static void main(String[] args) {
        StrategyContext context = new StrategyContext(new StrategyA());
        context.show();
    }
}
```

策略模式举例2

```
// 策略接口
interface PaymentStrategy {
    void pay(int amount);
}

// 具体策略类 - 支付宝支付
class AlipayStrategy implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " via Alipay.");
    }
}

// 具体策略类 - 微信支付
class WeChatPayStrategy implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " via WeChat Pay.");
    }
}

// 环境类
class PaymentContext {
    private PaymentStrategy paymentStrategy;

    public PaymentContext(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void processPayment(int amount) {
        paymentStrategy.pay(amount);
    }
}

// 客户端代码
public class StrategyPatternExample {
    public static void main(String[] args) {
        PaymentStrategy alipay = new AlipayStrategy();
        PaymentContext paymentContext = new PaymentContext(alipay);
```

```
        paymentContext.processPayment(100);

        PaymentStrategy weChatPay = new WeChatPayStrategy();
        paymentContext = new PaymentContext(weChatPay);
        paymentContext.processPayment(200);
    }
}
```

优缺点

- 策略类之间可以自由切换
- 易于扩展
- 避免多重条件判断
- 复合开遍则
- 客户端必须知道所有的策略类
- 通过享元模式可以减少相同策略类的创建

使用场景 系统动态选择某种算法 出现大量if else 可以考虑使用策略模式

jdk中的策略模式 Comparator接口的排序功能

15. 命令模式

将命令封装成类进行传递，使发出命令和执行命令的责任分开，对命令进行管理 角色

> 命令（Command）： 定义了一个接口，通常包含一个执行（execute）方法，用于封装某个特定的操作。 具体命令（Concrete Command）： 实现了命令接口，将一个具体的操作绑定到一个接收者上。 接收者（Receiver）： 执行具体操作的对象。 调用者（Invoker）： 调用命令对象并触发命令的执行。 客户端（Client）： 创建命令对象、接收者对象和调用者对象，并将它们组装起来。

```
//抽象命令类
public interface Command {

    void execute();
}

//命令接收者
public class SeniorChef {

    public void makeFood(String name,int num){
        System.out.println("做"+num +"份"+name);
    }
}
//具体命令类
public class OrderCommand implements Command {

    //持有接收者对象
```

```
    private SeniorChef reciver;

    //持有订单对象
    private Order order;

    public OrderCommand(SeniorChef reciver, Order order) {
        this.reciver = reciver;
        this.order = order;
    }

    @Override
    public void execute() {
        System.out.println(order.getDiningTable() + "桌的订单:");
        Map<String, Integer> foodDir = order.getFoodDir();
        foodDir.forEach((key, value) -> {
            reciver.makeFood(key, value);
        });
        System.out.println(order.getDiningTable() +"桌的饭准备好了");

    }
}
//其他附加信息类
public class Order {
    //桌号
    private int diningTable;

    //餐品
    private Map<String,Integer> foodDir = new HashMap<>();

    public int getDiningTable() {
        return diningTable;
    }

    public void setDiningTable(int diningTable) {
        this.diningTable = diningTable;
    }

    public Map<String, Integer> getFoodDir() {
        return foodDir;
    }

    public void setFoodDir(Map<String, Integer> foodDir) {
        this.foodDir = foodDir;
    }
    public void setFood(String name,int num){
        foodDir.put(name,num);
    }
}
//命令发起者
public class Waitor {
    //持有命令对象
    private List<Command> commandList = new ArrayList<>();

    public void setCommand(Command command){
```

```
            commandList.add(command);
        }

        //发起命令
        public void sendCommand(){
            System.out.println("服务员说，新订单来了....");
            for (Command command : commandList) {
                if(command!=null){
                    command.execute();
                }
            }
        }
    }
    public class Consumer {

        public static void main(String[] args) {

            Order order1 = new Order();
            order1.setDiningTable(1);
            order1.setFood("西红柿鸡蛋饭",1);
            order1.setFood("可乐",1);
            Order order2 = new Order();
            order2.setDiningTable(2);
            order2.setFood("汉堡",1);
            order2.setFood("百事",1);
            //创建接收着
            SeniorChef reciver = new SeniorChef();
            Command command1 = new OrderCommand(reciver,order1);
            Command command2 = new OrderCommand(reciver,order2);
            //创建调用者
            Waitor waitor = new Waitor();
            waitor.setCommand(command1);
            waitor.setCommand(command2);
            waitor.sendCommand();
        }
    }
```

命令模式举例2

```
// 命令接口
interface Command {
    void execute();
}

// 具体命令类 - 打开电视
class TVOnCommand implements Command {
    private TV tv;

    public TVOnCommand(TV tv) {
        this.tv = tv;
    }
```

```
    public void execute() {
        tv.turnOn();
    }
}

// 具体命令类 - 打开音响
class StereoOnCommand implements Command {
    private Stereo stereo;

    public StereoOnCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.turnOn();
    }
}
// 接收者类 - 电视
class TV {
    public void turnOn() {
        System.out.println("电视已打开");
    }
}

// 接收者类 - 音响
class Stereo {
    public void turnOn() {
        System.out.println("音响已打开");
    }
}
// 调用者类
class RemoteControl {
    private Command command;

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

    public void pressButton() {
        if (command != null) {
            command.execute();
        } else {
            System.out.println("未设置命令");
        }
    }
}
public class Main {
    public static void main(String[] args) {
        TV tv = new TV();
        Stereo stereo = new Stereo();

        Command tvOnCommand = new TVOnCommand(tv);
        Command stereoOnCommand = new StereoOnCommand(stereo);
```

```
        RemoteControl remote = new RemoteControl();

        remote.setCommand(tvOnCommand);
        remote.pressButton(); // 输出：电视已打开

        remote.setCommand(stereoOnCommand);
        remote.pressButton(); // 输出：音响已打开
    }
}
```

优缺点

- 降低系统耦合度
- 增加或删除命令很方便
- 可以使用宏命令
- 方便实现undo和redo
- 系统结构更加复杂
- 可能导致系统存在过多的命令类

使用场景

- 系统需要将请求调用者和接收着解除耦合可以使用命令模式
- 实现命令的撤销和回复操作

示例： jdk中的Runnable相当于是抽象的命令

16.责任链模式

避免请求发送者与多个请求处理者耦合在一起，将请求的处理者通过前一对象记住其下一个对象的引用而连成一条链，当有请求发生时，可沿着链传递请求，直到有对象处理它为止 角色

> 抽象处理者：定义一个处理请求的接口，包含抽象处理方法和一个后继处理者的引用 具体处理者：抽象处理者的实现类 类户端：创建处理链

```
//抽象处理者类
public abstract class Handler {
    protected final static int num_one =1;
    protected  final static int num_three = 3;
    protected  final static int num_seven = 7;
    //责任范围
    private int numStart;

    private int numEnd;

    //后继处理者
    private Handler nextHandler;
```

```
    public Handler(int numStart, int numEnd) {
        this.numStart = numStart;
        this.numEnd = numEnd;
    }

    public void setNextHandler(Handler nextHandler) {
        this.nextHandler = nextHandler;
    }

    //处理方法
    public abstract void handle(LeaveRequest request);

    public final void submit(LeaveRequest request){
        if(request.getNum() <= numEnd){
            this.handle(request);
        }else if(this.nextHandler!=null && request.getNum() > this.numEnd)
{
            this.nextHandler.submit(request);
        }else{
            System.out.println("流程结束");
        }
    }
}
//处理人1
public class GroupLeader extends Handler{

    public GroupLeader() {
        super(0, Handler.num_three);
    }

    @Override
    public void handle(LeaveRequest request) {
        System.out.println(request.getName() + "请假" + request.getNum() +
"天,"+request.getContent());
        System.out.println("小组长处理完毕");
    }
}
//处理人2
public class Manager extends Handler{
    public Manager() {
        super(Handler.num_one, Handler.num_three);
    }

    @Override
    public void handle(LeaveRequest request) {
        System.out.println(request.getName() + "请假" + request.getNum() +
"天,"+request.getContent());
        System.out.println("经理处理完毕");
    }
}
//处理人3
public class CEO extends Handler{
    public CEO() {
        super(Handler.num_three, Handler.num_seven);
```

```
    }

    @Override
    public void handle(LeaveRequest request) {
        System.out.println(request.getName() + "请假" + request.getNum() +
"天,"+request.getContent());
        System.out.println("ceo处理完毕");
    }
}
//请求
public class LeaveRequest {
    private String name;

    private int num;

    private String content;

    public String getName() {
        return name;
    }

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

    public int getNum() {
        return num;
    }

    public void setNum(int num) {
        this.num = num;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public LeaveRequest(String name, int num, String content) {
        this.name = name;
        this.num = num;
        this.content = content;
    }
}
public class Client {
    public static void main(String[] args) {
        LeaveRequest request = new LeaveRequest("steven",4,"cold");
        Handler gleader = new GroupLeader();
        Handler manager = new Manager();
        Handler ceo = new CEO();
        //设置处理链
```

```
            gleader.setNextHandler(manager);
            manager.setNextHandler(ceo);
            gleader.submit(request);
        }
    }
```

处理流程过长时，可以将处理流程分配到每个处理类中 javaweb中的filterChain是使用的责任链模式

17.状态模式

将类的状态封装成独立的状态类，允许状态对象在其内部发生改变时改变类的行为，避免在类中编写大量的状态判断逻辑 角色

- Context（上下文）：它是拥有状态的对象，它会根据当前的状态来执行不同的行为。它会维护一个对当前状态的引用，并将具体的行为委托给当前状态对象。
- State（状态接口）：定义了一个接口，用于封装与特定状态相关的行为。具体的状态类需要实现这个接口，来定义不同状态下的行为。
- ConcreteState（具体状态类）：实现了状态接口，具体描述了对象在某个特定状态下的行为。

```
//抽象状态类
public abstract class LiftState {
    //上下文对象，这里叫电梯
    protected Context context;

    public void setContext(Context context) {
        this.context = context;
    }

    public abstract void close();
    public abstract void run();
    public abstract void stop();
    public abstract void open();

}
//具体状态类1
public class RunState extends LiftState{
    @Override
    public void close() {
        //nothing
    }

    @Override
    public void run() {
        System.out.println("电梯已经在运行...");
    }

    @Override
    public void stop() {
        this.context.setCurrentState(Context.STOP_STATE);
        super.context.stop();
    }
```

```
    @Override
    public void open() {
        //nothing
    }
}

public class StopState extends LiftState{
    @Override
    public void close() {
        this.context.setCurrentState(Context.CLOSE_STATE);
        super.context.close();
    }

    @Override
    public void run() {
        this.context.setCurrentState(Context.RUN_STATE);
        super.context.run();
    }

    @Override
    public void stop() {
        System.out.println("电梯已经停止...");
    }

    @Override
    public void open() {
        this.context.setCurrentState(Context.OPEN_STATE);
        super.context.open();
    }
}
public class CloseState extends LiftState{
    @Override
    public void close() {
        System.out.println("电梯已关闭");
    }

    @Override
    public void run() {
        this.context.setCurrentState(Context.RUN_STATE);
        super.context.run();
    }

    @Override
    public void stop() {
        this.context.setCurrentState(Context.STOP_STATE);
        super.context.stop();
    }

    @Override
    public void open() {
        this.context.setCurrentState(Context.OPEN_STATE);
        super.context.open();
    }
```

```
}
public class OpenState extends LiftState{
    @Override
    public void close() {
        super.context.setCurrentState(Context.CLOSE_STATE);
        super.context.close();
    }

    @Override
    public void run() {
        //nothing
    }

    @Override
    public void stop() {
        //nothing
    }

    @Override
    public void open() {
        System.out.println("电梯开启...");
    }
}
//上下文对象
public class Context {

    public final static OpenState OPEN_STATE = new OpenState();
    public final static CloseState CLOSE_STATE = new CloseState();

    public final static  RunState RUN_STATE = new RunState();

    public final static  StopState STOP_STATE = new StopState();

    private LiftState currentState;

    public LiftState getCurrentState() {
        return currentState;
    }

    public void setCurrentState(LiftState currentState) {
        this.currentState = currentState;
        this.currentState.setContext(this);
    }

    public void close() {
        this.currentState.close();
    }

    public void run() {
        this.currentState.run();
    }

    public void stop() {
        this.currentState.stop();
```

```
    }

    public void open() {
        this.currentState.open();
    }
}
public class Main {

    public static void main(String[] args) {
        Context context =new Context();
        context.setCurrentState(Context.STOP_STATE);
        context.run();
        context.stop();
        context.open();
        context.close();
    }
}
```

开关灯状态代码示例

```
// State 接口
interface State {
    void turnOn();
    void turnOff();
}

// 具体状态类
class OnState implements State {

    @Override
    public void turnOn() {
        System.out.println("灯已经打开了");
    }

    @Override
    public void turnOff() {
        System.out.println("灯关闭");
    }
}

class OffState implements State {
    @Override
    public void turnOn() {
        System.out.println("灯打开");
    }

    @Override
    public void turnOff() {
        System.out.println("灯已经关闭了");
    }
}
```

```
// 上下文类
class Light {
    private State currentState;

    public Light() {
        currentState = new OffState();
    }

    public void setState(State state) {
        currentState = state;
    }

    public void turnOn() {
        currentState.turnOn();
    }

    public void turnOff() {
        currentState.turnOff();
    }
}

// 客户端代码
public class Client {
    public static void main(String[] args) {
        Light light = new Light();
        light.turnOn();
        light.turnOff();
    }
}
```

优缺点

- 将所有与状态相关的行为都放在一个类中，方便增加新的状态，只需要改变对象状态即可改变对象的行为
- 允许状态转换逻辑与对象合成一体，不需要在上下文类中进行大量的状态判断逻辑
- 系统中增加了大量的类
- 实现复杂
- 对开闭原则支持不好

18.观察者模式（事件发布订阅模式）

发布订阅模式，定义一种一对多的依赖关系，让观察者同时监听某个主题对象，主题对象发生变化时，通知所有观察者 角色

> 抽象主题 具体主题 抽象观察者 具体观察者

```
//被观察对象（主题）
public interface  Subject {
```

```
    void detach(Observer observer);

    void attach(Observer observer);


    void notify(String message);
}
//具体对象
public class SubSubject implements Subject{
    private List<Observer> observerList = new ArrayList<>();

    @Override
    public void detach(Observer observer) {
        observerList.remove(observer);
    }

    @Override
    public void attach(Observer observer) {
        observerList.add(observer);
    }

    @Override
    public void notify(String message) {
        for (Observer observer : observerList) {
            observer.update(message);
        }
    }
}
//抽象观察者
public interface Observer {
     void update(String message);
}
//具体观察者
public class Observer1 implements Observer{
    private String name;

    public Observer1(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        System.out.println(name + " get message"+message);
    }
}
//具体观察者
public class Observer2 implements Observer{
    private String name;
    public Observer2(String name) {
        this.name = name;
    }
    @Override
    public void update(String message) {
```

```
            System.out.println(name+ " get message :"+message);
        }
    }
    public class Main {
        public static void main(String[] args) {
            Subject subject = new SubSubject();
            subject.attach(new Observer1("jobs"));
            subject.attach(new Observer2("tom"));
            subject.notify("hello world");

        }
    }
```

优缺点

- 降低耦合
- 可实现广播机制

jdk中的观察者 Observable/Observer **在redis的订阅发布功能中，就使用了观察者模式，多个客户端同时订阅一个channel，当channel中有新的消息时，所有客户端会同时收到消息**

19. 中介者模式

- 存在多个同事类相互关联，呈复杂的网状结构，可以使用中介者对象来解耦对象之间的关联关系，当依赖关系出现变化时只需要修改中介者对象进行修改
- 定义一个中介角色来封装各个对象之间的交互，使原有对象之间的耦合解除
- 用于减少对象之间的直接耦合，通过引入一个中介者对象来管理多个对象之间的交互。中介者模式有助于将复杂的交互逻辑集中处理，从而使对象之间的通信更加简单和可维护。

角色： 抽象中介者： 提供同事对象注册与转发同事对象信息的抽象方法 具体中介者角色：实现中介者接口，定义一个List来管理同事对象，协调各个同事角色之间的交互关系，依赖于同事角色 抽象同事类角色：定义同事类，提供同事对象的交互方法 具体同事类：实现交互方法，由中介者来负责后续的交互 代码示例：使用中介者模式实现两个同事之间的数据发送与接收

```
// 定义中介者接口
interface Mediator {
    void sendMessage(String message, Colleague colleague);
}

// 实现中介者接口
class ConcreteMediator implements Mediator {
    private Colleague colleague1;
    private Colleague colleague2;

    public void setColleague1(Colleague colleague1) {
        this.colleague1 = colleague1;
    }

    public void setColleague2(Colleague colleague2) {
        this.colleague2 = colleague2;
```

```
    }

    @Override
    public void sendMessage(String message, Colleague colleague) {
        if (colleague == colleague1) {
            colleague2.receiveMessage(message);
        } else if (colleague == colleague2) {
            colleague1.receiveMessage(message);
        }
    }
}

// 定义抽象同事类
abstract class Colleague {
    protected Mediator mediator;

    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }

    public abstract void sendMessage(String message);
    public abstract void receiveMessage(String message);
}

// 实现具体同事类
class ConcreteColleague1 extends Colleague {
    public ConcreteColleague1(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void sendMessage(String message) {
        mediator.sendMessage(message, this);
    }

    @Override
    public void receiveMessage(String message) {
        System.out.println("ConcreteColleague1 received message: " +
message);
    }
}

// 实现具体同事类
class ConcreteColleague2 extends Colleague {
    public ConcreteColleague2(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void sendMessage(String message) {
        mediator.sendMessage(message, this);
    }

    @Override
```

```
    public void receiveMessage(String message) {
        System.out.println("ConcreteColleague2 received message: " +
message);
    }
}

// 测试中介者模式
public class MediatorPatternExample {
    public static void main(String[] args) {
        ConcreteMediator mediator = new ConcreteMediator();

        ConcreteColleague1 colleague1 = new ConcreteColleague1(mediator);
        ConcreteColleague2 colleague2 = new ConcreteColleague2(mediator);

        mediator.setColleague1(colleague1);
        mediator.setColleague2(colleague2);

        colleague1.sendMessage("Hello from Colleague1!");
        colleague2.sendMessage("Hi from Colleague2!");
    }
}
```

优点： 松散耦合 集中控制交互 一对多关联转变为一对一的关联 缺点： 当同事类太多，中介者的责任很大，会变得复杂庞大，导致系统难以维护

中介者模式和代理模式的区别（gpt）

- 主要目的：中介者模式的主要目的是解耦多个对象之间的复杂交互关系，将其转移到中介者对象中进行协调管理；代理模式的主要目的是为其他对象提供一种代理，以控制对该对象的访问。
- 参与对象：中介者模式的参与对象是多个对象之间的交互关系；代理模式的参与对象是客户端和被代理对象。
- 侧重点：中介者模式侧重于对象之间的交互逻辑的解耦和简化；代理模式侧重于对被代理对象的访问和控制。
- 关系建立：在中介者模式中，参与对象直接持有中介者对象的引用；在代理模式中，代理对象持有被代理对象的引用。

目的不同：中介者模式的目的是将多个对象之间的复杂交互逻辑集中到一个中介者对象中，从而简化对象之间的关系；代理模式的目的是为其他对象提供一种访问控制的方式，可以在访问对象之前进行一些额外的操作。

20. 迭代器模式

提供一个对象来顺序访问聚合对象中的一系列对象，而不暴露聚合对象的内部表示 角色： 抽象聚合角色：定义存储添加删除聚合元素及创建迭代器对象的接口 具体聚合角色： 实现抽象聚合类，返回一个具体的迭代器实例 抽象迭代器角色： 定义访问和遍历聚合元素的接口，通常包含hasNext（），next（） 具体迭代器角色： 实现访问和遍历聚合元素的方法 使用迭代器模式遍历书架上的书

```
//书本对象，被聚合元素
public class Book {
    private String name;

    public Book(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

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

    @Override
    public String toString() {
        return "Book{" +
                "name='" + name + '\'' +
                '}';
    }
}
//抽象聚合对象
public interface Aggregate {
    /**
     * 获取迭代器对象
     * @return
     */
    Iterator iterator();

    Object get(int index);

    boolean add(Object obj);

    int getLength();
}
//具体聚合对象
public class BookShelf implements Aggregate{

    private Book[] books;
    private int last = 0;

    public BookShelf(int maxsize){
        this.books = new Book[maxsize];
    }

    @Override
    public Object get(int index){
        return books[index];
    }

    @Override
```

```
    public boolean add(Object obj) {
        if(last < books.length){
            books[last++] = (Book) obj;
            return true;
        }else{
            return false;
        }
    }


    @Override
    public int getLength() {
        return last;
    }

    @Override
    public Iterator iterator() {
        return new BookShelfIterator(this);
    }
}
//具体迭代器实现
public class BookShelfIterator implements Iterator{

    private BookShelf bookShelf;

    private int index;

    public BookShelfIterator(BookShelf bookShelf) {
        this.bookShelf = bookShelf;
        this.index = 0;
    }

    @Override
    public boolean hasNext() {
        if(index < bookShelf.getLength()){
            return true;
        }
        return false;
    }

    @Override
    public Object next() {
        Book book = (Book) bookShelf.get(index++);
        return book;
    }
}
//客户端
public class Main {
    public static void main(String[] args) {

        BookShelf bookShelf = new BookShelf(10);
        bookShelf.add(new Book("Around the world in 80 Days"));
        bookShelf.add(new Book("Bible"));
        bookShelf.add(new Book("Three Body"));
```

```
        bookShelf.add(new Book("Daddy-Long-Legs"));
        bookShelf.add(new Book("thinking in java"));
        Iterator iterator = bookShelf.iterator();
        while (iterator.hasNext()){
            Book book = (Book) iterator.next();
            System.out.println(book);
        }
    }
}
```

在java中容器以及foreach语法都是用了迭代器接口，容器的迭代器一般以内部类的方式实现

优点： 支持以不同的遍历方式遍历一个聚合对象，可以聚合对象上可以定义多种遍历方式。引入抽象的迭代器接口，新增聚合类和迭代器类都很方便，不需要修改原有代码 缺点：增加了类的个数，增加了系统的复杂性

21. 访问者模式

封装一些用于某种数据结构中的各个元素的操作 在不改变数据结构的前提下定义作用于这些元素的新操作 引入一个访问者（Visitor）类，该类包含一系列访问操作，用于处理数据结构中不同类的对象。被访问的元素类提供一个接受（accept）方法，该方法接受一个访问者对象作为参数，从而使访问者能够访问并操作这个元素。

角色：

- 抽象访问者角色：定义对每个元素访问的行为
- 具体访问者角色：给出每个元素类访问时所产生的具体行为
- 抽象元素角色：定义一个接受访问方法，每个元素都可以被访问者访问
- 具体元素角色：提供接受访问方法的具体实现
- 对象结构：含有一组元素，可以迭代元素，提供访问者访问

示例实现： 遍历访问文件树结构的节点，包含文件夹和文件 抽象访问者：

```
public abstract class Visitor {
    public abstract  void visit(File file);
    public abstract  void visit(Directory directory);
}
```

抽象元素：

```
public interface Element {
    void accept(Visitor v);
}
```

扩展层（非必要）：

```
public abstract class Entry implements Element{
    public abstract String getName();
    public abstract int getSize();
    public Entry add(Entry entry) throws FileTreatmentException{
        throw new FileTreatmentException();
    }

    @Override
    public String toString() {
        return getName() + " (" + getSize() +")";
    }
}
public class FileTreatmentException extends RuntimeException{
    public FileTreatmentException() {
    }

    public FileTreatmentException(String message) {
        super(message);
    }
}
```

具体元素1（目录）：

```
public class Directory extends Entry{
    private String name;
    private ArrayList<Entry> dir = new ArrayList<>();

    public Directory(String name) {
        this.name = name;
    }

    @Override
    public void accept(Visitor v) {
        v.visit(this);
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public int getSize() {
        int size = 0;
        Iterator<Entry> it = iterator();
        while (it.hasNext()) {
            Entry entry = it.next();
            size += entry.getSize();
        }
        return size;
    }
```

```
    public Iterator<Entry> iterator(){
        return dir.iterator();
    }

    @Override
    public Entry add(Entry entry) throws FileTreatmentException {
        dir.add(entry);
        return this;
    }
}
```

具体元素（文件）：

```
public class File extends Entry{
    private String name;
    private int size;
    public File(String name,int size){
        this.name = name;
        this.size = size;
    }
    @Override
    public void accept(Visitor v) {
        v.visit(this);
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public int getSize() {
        return size;
    }
}
```

具体访问者角色

```
public class ListVisitor extends Visitor{
    private String currentDir = "";

    @Override
    public void visit(File file) {
        System.out.println(currentDir + "/" + file);
    }

    @Override
    public void visit(Directory directory) {
```

```
        System.out.println(currentDir + "/" + directory);
        String saveDir = currentDir;
        currentDir = currentDir + "/" + directory.getName();
        Iterator<Entry> it = directory.iterator();
        while (it.hasNext()){
            Entry entry = it.next();
            entry.accept(this);
        }
        currentDir = saveDir;
    }
}
```

客户端程序：

```
public class Main {
    public static void main(String[] args) {
        System.out.println("visitor file system");
        Directory rootDir = new Directory("root");
        Directory bindDir = new Directory("bin");
        Directory tmpDir = new Directory("tmp");
        Directory usrDir = new Directory("usr");
        rootDir.add(bindDir);
        rootDir.add(tmpDir);
        rootDir.add(usrDir);
        bindDir.add(new File("vi",10000));
        bindDir.add(new File("letex",20000));
        rootDir.accept(new ListVisitor());
    }
}
```

创建一个简单的图形形状示例，其中包含不同类型的形状（如圆形和矩形），并且我们将实现一个访问者模式来计算每种形状的面积。

```
// 访问者接口
interface ShapeVisitor {
    double visitCircle(Circle circle);
    double visitRectangle(Rectangle rectangle);
}

// 具体访问者类
class AreaCalculator implements ShapeVisitor {
    @Override
    public double visitCircle(Circle circle) {
        return Math.PI * circle.getRadius() * circle.getRadius();
    }

    @Override
    public double visitRectangle(Rectangle rectangle) {
        return rectangle.getWidth() * rectangle.getHeight();
```

```
    }
}
// 元素接口
interface Shape {
    double accept(ShapeVisitor visitor);
}

// 具体元素类 - 圆形
class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    @Override
    public double accept(ShapeVisitor visitor) {
        return visitor.visitCircle(this);
    }
}

// 具体元素类 - 矩形
class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double getWidth() {
        return width;
    }

    public double getHeight() {
        return height;
    }

    @Override
    public double accept(ShapeVisitor visitor) {
        return visitor.visitRectangle(this);
    }
}
//客户端
public class VisitorPatternExample {
    public static void main(String[] args) {
        AreaCalculator areaCalculator = new AreaCalculator();

        Circle circle = new Circle(5);
```

```
        Rectangle rectangle = new Rectangle(4, 6);

        double circleArea = circle.accept(areaCalculator);
        double rectangleArea = rectangle.accept(areaCalculator)
    }
}
```

适用场景 访问者模式在一些需要对复杂对象结构进行操作且希望将操作与数据结构分离的场景中非常有用，例如编译器设计、图形处理等领域。

22. 备忘录模式 Memento

提供一种状态恢复的实现机制，使得用户可以方便的回退到一个特定历史步骤 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，当需要时将该对象恢复到原先你保存的状态

角色 发起人角色：记录当前时刻的内部状态信息，提供备忘录恢复和创建的功能 备忘录角色：负责存储发起人的内部状态，在需要的时候提供这些内部状态给发起人 管理者角色：对备忘录进行管理，提供保存和获取备忘录的功能，但其不能对备忘录的内容进行访问和修改

示例： 发起人角色+管理者

```
public class Gamer {
    private int money;
    private List<String> fruits = new ArrayList<>();
    private Random random = new Random();
    private static String[] fruitsname = {
            "apple","orange","banana"
    };
    public Gamer(int money){
        this.money = money;
    }

    public int getMoney() {
        return money;
    }

    public void bet(){
        int dice = random.nextInt(6) + 1;
        if(dice == 1){
            money += 100;
            System.out.println("money add");
        }else if(dice == 2){
            money /=2;
            System.out.println("money half");
        }else if(dice == 6){
            String f = getFruit();
            System.out.println("get fruit ("+ f + ")");
            fruits.add(f);
        }else{
            System.out.println("nothing");
```

```
        }

    }


    public Memento createMemento(){
        Memento m = new Memento(money);
        Iterator it = fruits.iterator();
        while (it.hasNext()){
            String f= (String) it.next();
            if(f.startsWith("good fruit ")){
                m.addFruit(f);
            }
        }
        return m;
    }

    public void restoreMemento(Memento memento){
        this.money = memento.money;
        this.fruits = memento.getFruits();
    }

    private String getFruit() {
        String prefix = "";
        if(random.nextBoolean()){
            prefix = "good fruit ";
        }
        return prefix + fruitsname[random.nextInt(fruitsname.length)];
    }

    @Override
    public String toString() {
        return "Gamer{" +
                "money=" + money +
                ", fruits=" + fruits +
                ", random=" + random +
                '}';
    }
}
```

备忘录角色

```
public class Memento {
    int money;
    ArrayList<String> fruits;
    public Memento(int money) {
        this.money = money;
        fruits = new ArrayList<>();
    }

    public int getMoney(){
        return money;
```

```
    }
    void addFruit(String fruit){
        fruits.add(fruit);
    }
    List<String> getFruits(){
        return (List<String>) fruits.clone();
    }
}
```

客户端

```
public class Main {

    public static void main(String[] args) {
        Gamer gamer = new Gamer(100);
        Memento memento = gamer.createMemento();
        for (int i = 0; i < 100; i++) {
            System.out.println("========" + i);
            System.out.println("current status :" + gamer);
            gamer.bet();
            System.out.println("money : "+ gamer.getMoney());
            if(gamer.getMoney() > memento.getMoney()){
                System.out.println("money add,store status");
                memento = gamer.createMemento();
            }else if(gamer.getMoney() < memento.getMoney()/2){
                System.out.println("money desc,restore the status");
                gamer.restoreMemento(memento);
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

23. 解释器模式

给定一个语言，定义它的文法标识，定一个一个解释器，这个解释器用来解析文法标识 将语法规则封装成类，以模拟控制机器人为例,通过对语法的解释封装，形成在java平台上的专门的控制语言 角色：

- 抽象表达式角色：定义解释器接口，约定解释器的解释操作，包含解释方法
- 终结符表达式角色：是抽象表达式的子类，用来实现文法中与终结符相关的操作
- 非终结符表达式角色：抽象表达式子类，用来实现文法中与非终结符相关的操作，文法中的每条规则都对应一个非终结表符达式
- 环境上下文角色：包含各个解释器需要的数据或公共的功能，一般用来传递被所有解释器共享的数据，后面的解释器可以从这里获取这些值，实际上是存储指令执行的位置 客户端角色

示例：机器人根据指令进行移动操作 指令文件：program.txt

```
program end
program go end
program go right go right go right go right go right end
program repeat 4 go right end end
program repeat 4 repeat 3 go right go left end right end end
```

抽象表达式角色：

```
public abstract class Node {
    public abstract void parse(Context context) throws ParseException;
}
```

中间对象，包含了终结表达式角色的功能

```
public class ProgramNode extends Node {
    private Node commandListNode;

    @Override
    public void parse(Context context) throws ParseException {
        context.skipToken("program");
        commandListNode = new CommandListNode();
        commandListNode.parse(context);
    }

    @Override
    public String toString(){
        return "[program "+ commandListNode + "]";
    }
}
public class CommandListNode extends Node {

    private ArrayList<Node> list = new ArrayList<>();

    @Override
    public void parse(Context context) throws ParseException {
        while (true){
            if(context.currentToken() == null){
                throw new ParseException("Missing end");
            }else if(context.currentToken().equals("end")){
                context.skipToken("end");
                break;
            }else{
                Node commandNode = new CommandNode();
                commandNode.parse(context);
                list.add(commandNode);
            }
```

```
        }
    }

    @Override
    public String toString() {
        return list.toString();
    }
}
```

非终结表达式（三个）

```
public class PrimitiveCommandNode extends Node{
    private String name;
    @Override
    public void parse(Context context) throws ParseException {
        name = context.currentToken();
        context.skipToken(name);
        if(!name.equals("go") && !name.equals("right") &&
!name.equals("left")){
            throw new ParseException(name + " is undefined");
        }
    }

    @Override
    public String toString() {
        return name;
    }
}
public class RepeatCommandNode extends Node{
    private int number;
    private Node commandListNode;
    @Override
    public void parse(Context context) throws ParseException {
        context.skipToken("repeat");
        number = context.currentNumber();
        context.nextToken();
        commandListNode = new CommandListNode();
        commandListNode.parse(context);
    }

    @Override
    public String toString() {
        return "[" +
                "repeat " + number +
                " " + commandListNode +
                ']';
    }
}
public class CommandNode extends Node{
    private Node node;
    @Override
    public void parse(Context context) throws ParseException {
```

```
        if(context.currentToken().equals("repeat")){
            node = new RepeatCommandNode();
            node.parse(context);
        }else{
            node = new PrimitiveCommandNode();
            node.parse(context);
        }
    }

    @Override
    public String toString() {
        return node.toString();
    }
}
```

上下文角色

```
public class Context {

    private StringTokenizer tokenizer;

    private String currentToken;

    public Context(String text) {
        tokenizer = new StringTokenizer(text);
        nextToken();
    }

    public String nextToken(){
        if(tokenizer.hasMoreTokens()){
            currentToken = tokenizer.nextToken();
        }else{
            currentToken = null;
        }
        return currentToken;
    }

    public String currentToken(){
        return currentToken;
    }

    public void skipToken(String token) throws ParseException{
        if(!token.equals(currentToken)){
            throw new ParseException("warning: " + token + " is expected,
but "+ currentToken + " is found.");
        }
        nextToken();
    }
    public int currentNumber() throws ParseException{
        int number = 0;
        try {
            number = Integer.parseInt(currentToken);
```

```
        }catch (NumberFormatException e){
            throw new ParseException("warning: "+ e);
        }
        return number;
    }

}
```

客户端

```
public class Main {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(new
FileReader("program.txt"));
            String text;
            while((text =reader.readLine())!=null){
                System.out.println("text = \""+ text + "\"");
                Node node = new ProgramNode();
                node.parse(new Context(text));
                System.out.println("node= "+ node);
            }
            reader.close();
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

优点： 易于改变和扩展文法 实现文法容易 增加新的解释表达式比较方便 缺点： 对于复杂文法难以维护 执行效率较低 代码调试比较麻烦