设计模式

创建型	单例模式	确保类的唯一实例
创建型	工厂模式	提供创建对象的工厂方法
结构型	适配器模式	让不同接口兼容
结构型	装饰器模式	动态扩展对象功能
行为型	观察者模式	一对多事件通知
行为型	策略模式	运行时动态切换算法
行为型	责任链模式	多个处理器顺序处理请求

单例模式

单例模式的关键在于把new的功能（构造器） 私有化，导致无法new。

单例模式基本会被反射破解　除非用枚举

饿汉式

```
public class Window {
private static Window window = new Window();

    private Window(){

    }

    public static Window getWindow(){
        return window;
    }
}
// 使用getwindow返回单个实例
```

懒汉式单例+双重检查锁

该模式在类加载时不创建对象, 只有在使用时才创建对象, 这时生成对象的数量需要我们来控制, 所以会存在线程安全问题。 说白了就是调用**getinstance**的时候再**new**一个

当有多个线程同时进入到第一个if中时, 第一个线程去创建对象, 后来获得锁的线程就不会再创建对象, 所以这里利用了两个if, 也就是双重检索+synchronized

```
public class Window {
    private static volatile Window window; 防止指令重排

    private Window(){

    }

    //懒汉式单例,在类加载的时候不创建对象,在使用时创建对象
    //这时,生成的对象的数量需要我们自己来控制
    //懒汉式单例会出现线程安全问题:
    //    在多线程访问时,可能会出现多个线程同时进入到if,就会创建出多个对象
    //如何解决?
    //  1.给方法加锁,但是效率太低,一次只能有一个线程进入
    //  2.给代码块加锁,双重检索+synchronized
    public static Window getWindow(){
        if(window==null){
            synchronized (Window.class){  //锁类模板（所有该类的实例）
                if(window==null){
                    window=new Window();
                }
```

```
                }
            }
            return window;
        }
}
```

静态内部类

```
public class Singleton {
    // 私有构造方法，防止外部实例化
    private Singleton() {}

    // 静态内部类，持有 Singleton 实例
    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    // 获取单例对象
    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}程安全
Java 类的加载是线程安全的，JVM 确保了 SingletonHolder.INSTANCE 只会初始化一次。
避免了手动同步 synchronized 带来的性能开销，相比双重检查锁（DCL）更优雅。
☑ 延迟加载（Lazy Loading）
SingletonHolder 只有在 getInstance() 第一次被调用时才会被加载，避免了类加载时就实例化（节省资源）。
☑ 实现简单
不需要 synchronized 关键字，也不需要使用 volatile，代码清晰简洁，没有双重检查锁（DCL）那样复杂。
让 SingletonHolder 直接管理 INSTANCE
```

枚举

枚举的单例意义不大 因为只有枚举常量是单例 自己实现内部类其实意义不大。 实现自己类的单例还是得用上面几种方法

工厂模式

工厂模式有几个变种，最常见的包括：

1. 简单工厂模式（Simple Factory Pattern）
2. 工厂方法模式（Factory Method Pattern）
3. 抽象工厂模式（Abstract Factory Pattern）

就是接口嘛 会涉及到IOC和AOP

简单工厂模式（Simple Factory Pattern）

```
// 产品接口
public interface Product {
    void doSomething();
}

// 具体产品
public class ConcreteProductA implements Product {
    public void doSomething() {
        System.out.println("Product A");
    }
}

public class ConcreteProductB implements Product {
    public void doSomething() {
        System.out.println("Product B");
    }
}

// 工厂类
public class ProductFactory {
    public static Product createProduct(String type) {
        if ("A".equals(type)) {
            return new ConcreteProductA();
        } else if ("B".equals(type)) {
            return new ConcreteProductB();
        }
```

```
        return null;
    }
}

// 客户端
public class Client {
    public static void main(String[] args) {
        Product productA = ProductFactory.createProduct("A");
        productA.doSomething(); // Output: Product A

        Product productB = ProductFactory.createProduct("B");
        productB.doSomething(); // Output: Product B
    }
}
```

工厂方法模式（Factory Method Pattern）

```
// 产品接口
public interface Product {
    void doSomething();
}

// 具体产品
public class ConcreteProductA implements Product {
    public void doSomething() {
        System.out.println("Product A");
    }
}

public class ConcreteProductB implements Product {
    public void doSomething() {
        System.out.println("Product B");
    }
}

// 抽象工厂
public abstract class Creator {
    public abstract Product createProduct();
}

// 具体工厂
```

```
public class ConcreteCreatorA extends Creator {
    public Product createProduct() {
        return new ConcreteProductA();
    }
}

public class ConcreteCreatorB extends Creator {
    public Product createProduct() {
        return new ConcreteProductB();
    }
}

// 客户端
public class Client {
    public static void main(String[] args) {
        Creator creatorA = new ConcreteCreatorA();
        Product productA = creatorA.createProduct();
        productA.doSomething(); // Output: Product A

        Creator creatorB = new ConcreteCreatorB();
        Product productB = creatorB.createProduct();
        productB.doSomething(); // Output: Product B
    }
}
```

缺点：

- 增加了类的数量（每种产品需要一个具体工厂）。

  可以通过传入要生产的类参数来生成对应产品

  函数式接口

抽象工厂模式

```
// 抽象产品
public interface ProductA {
    void doSomethingA();
}

public interface ProductB {
    void doSomethingB();
```

```
}

// 具体产品
public class ConcreteProductA1 implements ProductA {
    public void doSomethingA() {
        System.out.println("Product A1");
    }
}

public class ConcreteProductA2 implements ProductA {
    public void doSomethingA() {
        System.out.println("Product A2");
    }
}

public class ConcreteProductB1 implements ProductB {
    public void doSomethingB() {
        System.out.println("Product B1");
    }
}

public class ConcreteProductB2 implements ProductB {
    public void doSomethingB() {
        System.out.println("Product B2");
    }
}

// 抽象工厂
public interface AbstractFactory {
    ProductA createProductA();
    ProductB createProductB();
}

// 具体工厂
public class ConcreteFactory1 implements AbstractFactory {
    public ProductA createProductA() {
        return new ConcreteProductA1();
    }

    public ProductB createProductB() {
        return new ConcreteProductB1();
    }
}
```

```
public class ConcreteFactory2 implements AbstractFactory {
    public ProductA createProductA() {
        return new ConcreteProductA2();
    }

    public ProductB createProductB() {
        return new ConcreteProductB2();
    }
}

// 客户端
public class Client {
    public static void main(String[] args) {
        AbstractFactory factory1 = new ConcreteFactory1();
        ProductA productA1 = factory1.createProductA();
        ProductB productB1 = factory1.createProductB();
        productA1.doSomethingA(); // Output: Product A1
        productB1.doSomethingB(); // Output: Product B1

        AbstractFactory factory2 = new ConcreteFactory2();
        ProductA productA2 = factory2.createProductA();
        ProductB productB2 = factory2.createProductB();
        productA2.doSomethingA(); // Output: Product A2
        productB2.doSomethingB(); // Output: Product B2
    }
}
```

定义

抽象工厂模式提供了一个接口，用来创建一系列相关或相互依赖的对象，而不需要指定具体的类。每个具体工厂类都可以生产不同种类的产品，每种产品都遵循相同的接口。抽象工厂可以有多个工厂方法，分别对应创建不同的产品。

结构

- 抽象产品**A**、**B**（AbstractProductA, AbstractProductB）：一组相关的产品接口。
- 具体产品**A**、**B**（ConcreteProductA, ConcreteProductB）：具体的产品实现。
- 抽象工厂（AbstractFactory）：声明一组用于创建产品A、B的抽象方法。

- 具体工厂**A**、**B**（ConcreteFactoryA, ConcreteFactoryB）：实现抽象工厂，创建具体的产品。

简单工厂模式：一个工厂类根据输入决定创建哪个具体类，适用于创建产品种类较少的场景。

工厂方法模式：工厂类提供一个抽象的接口，允许子类决定产品的实例化，适用于产品种类较多或需要扩展的场景。

抽象工厂模式：创建一系列相关的产品，适用于产品族的创建，确保创建的产品是相关联的。

抽象工厂模式相比工厂方法模式的核心区别在于“产品族”**。

.

适配器模式

适配器模式是一种结构型设计模式，用于将一个接口转换为客户端期望的另一个接口，以便两个不兼容的接口能够一起工作。

适配器模式通常有类适配器和对象适配器两种实现方式：

类适配器（基于继承）

- 通过继承的方式实现适配器，适用于目标类和适配者（被适配类）存在父子关系的情况。
- 缺点：由于使用继承，适配器不能适配多个不同的类。

```
// 目标接口：客户期望的接口
interface Target {
    void request();
}

// 被适配者：已有的类，方法名不同
class Adaptee {
    public void specialRequest() {
        System.out.println("特殊请求");
    }
}

// 适配器：继承 Adaptee，并实现 Target
```

```
class ClassAdapter extends Adaptee implements Target {
    public void request() {
        specialRequest(); // 适配
    }
}

// 客户端代码
public class AdapterPatternDemo {
    public static void main(String[] args) {
        Target target = new ClassAdapter();
        target.request(); // 输出：特殊请求
    }
}
```

特点：

ClassAdapter 继承 Adaptee，并实现 Target，相当于“翻译”了接口。

缺点：由于是继承，ClassAdapter 只能适配一个被适配者，扩展性不强。

对象适配器（基于组合）

- 通过组合的方式，在适配器中持有被适配类的对象，使其实现目标接口。
- 优点：可以适配多个不同的类，更加灵活，符合面向对象设计原则

```
// 目标接口
interface Target {
    void request();
}

// 被适配者
class Adaptee {
    public void specialRequest() {
        System.out.println("特殊请求");
    }
}

// 适配器：持有被适配者的实例，进行适配
class ObjectAdapter implements Target {
    private Adaptee adaptee;   //在这里

    public ObjectAdapter(Adaptee adaptee) {
        this.adaptee = adaptee;
```

```
    }

    public void request() {
        adaptee.specialRequest(); // 适配
    }
}

// 客户端代码
public class AdapterPatternDemo {
    public static void main(String[] args) {
        Adaptee adaptee = new Adaptee();
        Target target = new ObjectAdapter(adaptee);
        target.request(); // 输出：特殊请求
    }
}
```

装饰器模式

装饰器模式是一种结构型设计模式，用于动态地给对象添加额外的功能，而不改变其原有的代码。

```
interface Coffee {
    String getDescription();
    double cost();
}

class SimpleCoffee implements Coffee {
    public String getDescription() { return "Simple Coffee"; }
    public double cost() { return 5.0; }
}


abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;
    public CoffeeDecorator(Coffee coffee) { this.coffee = coffee; }
    public String getDescription() { return
coffee.getDescription(); }
    public double cost() { return coffee.cost(); }
}
```

```
class Milk extends CoffeeDecorator {
    public Milk(Coffee coffee) { super(coffee); } //调用了
CoffeeDecorator 的 构造函数，用于存储被装饰的 coffee 对象。在 Milk 这个类
中，虽然没有显式声明 coffee 变量，但它继承了 CoffeeDecorator，而
CoffeeDecorator 已经定义了 coffee 变量，所以 Milk 可以直接使用它。

    public String getDescription() { return coffee.getDescription()
+ ", Milk"; }
    public double cost() { return coffee.cost() + 2.0; }
}

class Sugar extends CoffeeDecorator {
    public Sugar(Coffee coffee) { super(coffee); }
    public String getDescription() { return coffee.getDescription()
+ ", Sugar"; }
    public double cost() { return coffee.cost() + 1.0; }
}
```

观察者模式是一种行为型设计模式，用于定义对象之间的一对多依赖关系。
当一个对象（被观察者，Subject）发生变化时，它会通知所有依赖它的对象（观察者，**Observers**），使它们自动更新。

简单讲就是我这个类持有你这个类（观察者类） 然后我调用方法的时候顺便调用你的类的方法（通知）

```
interface Observer {
    void update(String message);  // 当被观察者状态发生变化时调用
}


import java.util.ArrayList;
import java.util.List;

interface Subject {
    void attach(Observer observer);   // 添加观察者
    void detach(Observer observer);   // 移除观察者
    void notifyObservers(String message); // 通知所有观察者
```

```
}



class NewsPublisher implements Subject {
    private List<Observer> observers = new ArrayList<>();

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

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

    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

    // 业务方法：发布新闻
    public void publishNews(String news) {
        System.out.println("News Published: " + news);
        notifyObservers(news);
    }
}


class User implements Observer {
    private String name;

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

    @Override
    public void update(String message) {
        System.out.println(name + " received update: " + message);
    }
```

```
}
```

策略模式（**Strategy Pattern**）

这个倒没啥 很常用 通用的接口 而不是用extend

策略模式使用 接口 + 组合（`Composition`） 代替继承：

1. 概念

策略模式是一种行为型设计模式，它允许将一组算法封装成独立的类，并使它们可以互相替换，而不会影响使用这些算法的代码。

核心思想：定义一系列算法（策略），并将它们封装在不同的类中，客户端在运行时可以自由选择需要的算法，而不需要修改原来的代码。

---

2. 适用场景

- ☑ 多个算法可以互相替换（如不同的排序算法）。
- ☑ 希望代码具有良好的扩展性，避免 `if-else` 或 `switch-case` 语句。
- ☑ 不同环境下需要使用不同的业务逻辑（如支付方式选择：支付宝/微信/银行卡）。

---

3. 结构

策略模式主要由三部分组成：

1. 策略接口（**Strategy**）：定义一组可互换的算法或行为。
2. 具体策略（**Concrete Strategy**）：实现具体的算法。
3. 上下文类（**Context**）：持有一个策略对象，并在运行时调用具体策略。

```
// 策略接口
interface PaymentStrategy {
```

```
    void pay(int amount);
}

// 具体策略：使用支付宝支付
class AlipayStrategy implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("使用支付宝支付：" + amount + " 元");
    }
}

// 具体策略：使用微信支付
class WeChatPayStrategy implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("使用微信支付：" + amount + " 元");
    }
}

// 具体策略：使用银行卡支付
class BankCardStrategy implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("使用银行卡支付：" + amount + " 元");
    }
}



// 上下文类，持有一个支付策略
class PaymentContext {
    private PaymentStrategy paymentStrategy;

    // 通过构造函数注入策略
    public PaymentContext(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    // 运行时调用具体的支付策略
    public void executePayment(int amount) {
        paymentStrategy.pay(amount);
    }
}
```

可不可以这样理解 简单工厂重点在提供类 策略模式重点在不同类的方法？

GPT：简单工厂 关注对象的创建，它让客户端不需要关心具体创建哪种类的实例，而是交给工厂来创建。

策略模式 关注不同算法的执行方式，它让客户端可以在运行时切换不同的策略，而不影响主逻辑。

简单工厂是“选菜品”，策略模式是“选做法”。

责任链模式

主要目的是让多个处理对象形成一个链，每个对象依次处理请求，直到某个对象处理完毕。客户端不需要知道是哪一个对象处理了请求，而是将请求传递给链中的对象进行处理。

springmvc和boot 常用拦截器和过滤器

```
abstract class Handler {
    protected Handler nextHandler;

    // 设置下一个处理者
    public void setNextHandler(Handler nextHandler) {
        this.nextHandler = nextHandler;
    }

    // 处理请求
    public abstract void handleRequest(int request);
}



// 具体处理者1：处理 1-5 的请求
class ConcreteHandler1 extends Handler {
    @Override
    public void handleRequest(int request) {
        if (request >= 1 && request <= 5) {
```

```
            System.out.println("ConcreteHandler1 处理请求: " +
request);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request); // 传递给下一个处理者
        }
    }
}

// 具体处理者2：处理 6-10 的请求
class ConcreteHandler2 extends Handler {
    @Override
    public void handleRequest(int request) {
        if (request >= 6 && request <= 10) {
            System.out.println("ConcreteHandler2 处理请求: " +
request);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}

// 具体处理者3：处理 11-15 的请求
class ConcreteHandler3 extends Handler {
    @Override
    public void handleRequest(int request) {
        if (request >= 11 && request <= 15) {
            System.out.println("ConcreteHandler3 处理请求: " +
request);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}



public class ChainOfResponsibilityDemo {
    public static void main(String[] args) {
        // 创建处理者
        Handler handler1 = new ConcreteHandler1();
        Handler handler2 = new ConcreteHandler2();
        Handler handler3 = new ConcreteHandler3();
```

```
        // 设置责任链
        handler1.setNextHandler(handler2);
        handler2.setNextHandler(handler3);

        // 向链条发送请求
        System.out.println("请求 3:");
        handler1.handleRequest(3);

        System.out.println("\n请求 8:");
        handler1.handleRequest(8);

        System.out.println("\n请求 12:");
        handler1.handleRequest(12);

        System.out.println("\n请求 20:");
        handler1.handleRequest(20);
    }
}
```

其他杂谈

是的，责任链模式和单例模式是可以结合使用的。结合这两种模式的一个典型场景是：多个请求处理器（责任链模式）作为单例对象存在（单例模式），以确保系统中只会有一个处理链的实例，并且能够复用。

以静态内部类举例

```
public abstract class LogHandler {
    protected LogHandler nextHandler;

    public void setNextHandler(LogHandler nextHandler) {
        this.nextHandler = nextHandler;
```

```
    }

    public abstract void handleRequest(String message);
}


public class InfoLogHandler extends LogHandler {
    @Override
    public void handleRequest(String message) {
        if (message.contains("INFO")) {
            System.out.println("INFO: " + message);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(message);
        }
    }
}

public class ErrorLogHandler extends LogHandler {
    @Override
    public void handleRequest(String message) {
        if (message.contains("ERROR")) {
            System.out.println("ERROR: " + message);
        } else if (nextHandler != null) {
            nextHandler.handleRequest(message);
        }
    }
}


public class LogHandlerChain {
    private static class LogHandlerChainHolder {
        private static final LogHandlerChain INSTANCE = new
LogHandlerChain();
    }

    private LogHandler head;

    private LogHandlerChain() {
        // 初始化责任链
        head = new InfoLogHandler();
        LogHandler errorHandler = new ErrorLogHandler();
        head.setNextHandler(errorHandler); // 链接处理者
    }
```

```
    public static LogHandlerChain getInstance() {
        return LogHandlerChainHolder.INSTANCE;
    }

    public void handleLog(String message) {
        head.handleRequest(message); // 从链头开始处理
    }
}
```