设计模式

面向对象的特点：可维护、可复用、可拓展、灵活性好。随着业务变得越来越复杂，让面向对象保持结构良好的秘诀就是设计模式。

设计模式六原则

▼ 开闭原则

一个软件实体，如类、模块、函数应该对修改封闭， 对拓展开放

▼ 单一职责原则

一个类只做一件事， 一个类应该只有一个引起它修改的原因

▼ 里氏替换原则

子类应该可以完全替换父类。也就是说在使用继承时， 只拓展新功能，而不要破坏父类原有的功能

▼ 依赖倒置原则

细节应该依赖于抽象，抽象不应依赖于细节。把抽象层放在程序设计的高层，并保持稳定，程序的细节变化由底层的实现层来完成

▼ 迪米特法则（最少知道原则）

一个类不应该知道自己操作的类的细节，换言之，只和朋友说话， 不和朋友的朋友说话

▼ 接口隔离原则

客户端不应该依赖它不需要的接口。如果一个接口在实现时，部分方法由于冗余被客户端空实现，则应该将接口拆分，让实现类只需依赖自己需要的接口方法。

构造型模式

▼ **工厂模式 Factory**　隐藏构造细节

通过new的方式构建对象， 相当于调用者多知道了一个类，增加了类与类之间的联系。而工厂模式就是用来封装构建过程的

▼ **简单工厂模式**

工厂承担了构建所有对象的职责， 变成了超级类。每一个类的变更都需要修改工厂类，违背了单一职责原则。每新增一个类，都需要修改工厂类， 违背了开闭原则

▼ 代码实现

```
public class FruitFactory {
    public Fruit create(String type) {
        switch (type) {
            case "苹果":
                AppleSeed appleSeed = new AppleSeed();
                Sunlight sunlight = new Sunlight();
                Water water = new Water();
                return new Apple(appleSeed, sunlight, water);
            case "梨子":
                return new Pear();
            default:
                throw new IllegalArgumentException("暂时没有这种水果");
        }
    }
}

public class User {
    private void eat(){
        FruitFactory fruitFactory = new FruitFactory();
        Fruit apple = fruitFactory.create("苹果");
        Fruit pear = fruitFactory.create("梨子");
        apple.eat();
        pear.eat();
    }
}

作者：力扣 (LeetCode)
```

```
链接：https://leetcode-cn.com/leetbook/read/design-patterns/99gpi3/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **工厂方法模式**

每一个工厂都只负责单一的产品，工厂方法模式虽然同样暴露了多个工厂概念，但依然可以隐藏复杂的构造过程

▼ 代码实现

```
public class SurgicalMaskFactory{

    public Mask create() {
        return new SurgicalMask();
    }
}


public class N95MaskFactory {
    public Mask create() {
        return new N95Mask();
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/99gpi3/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **抽象工厂**

为每一类工厂提取出抽象接口， 使得新增工厂、替换工厂变得容易。将工厂方法模式下定义的多个单一职责的工厂， 统一继承一个工厂抽象类或接口。客户端使用抽象类来调用方法。抽象工厂模式很好的发挥了开闭原则，依赖倒置原则

▼ 代码实现

```
public interface IFactory {
    Mask create();
}


public class SurgicalMaskFactory implements IFactory{

    @Override
    public Mask create() {
        return new SurgicalMask();
    }
}


public class N95MaskFactory implements IFactory {
    @Override
    public Mask create() {
        return new N95Mask();
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/99zelm/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **单例模式 Singleton**　全局使用同一对象

当某个对象全局只需要一个实例时。能避免对象重复创建， 节约空间并提升效率。单例模式有两种实现： lazy（懒汉式）和 eager（饿汉式）

▼ 代码实现

```
//eager
public class Singleton {

    private static Singleton instance = new Singleton();

    private Singleton() {
```

```
    }

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

//lazy
//1、volatile + synchronized双重检查
public class Singleton {

    private static volatile Singleton instance = null;

    private Singleton() {
    }

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

//laze
//2、静态内部类托管
public class Singleton {

    private static class SingletonHolder {
        public static Singleton instance = new Singleton();
    }

    private Singleton() {
    }

    public static Singleton getInstance() {
        return SingletonHolder.instance;
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/99sx01/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **建造者模式 Builder**　用于创建构造过程稳定，但配置多变的对象

主要应用于构建过程稳定，可通过不同配置建造出不同对象的场景。

▼ 代码实现

```
public class MilkTea {
    private final String type;
    private final String size;
    private final boolean pearl;
    private final boolean ice;

  private MilkTea() {}

    private MilkTea(Builder builder) {
        this.type = builder.type;
        this.size = builder.size;
        this.pearl = builder.pearl;
        this.ice = builder.ice;
    }

    public String getType() {
        return type;
    }

    public String getSize() {
        return size;
    }

    public boolean isPearl() {
        return pearl;
    }
    public boolean isIce() {
```

```
        return ice;
    }

    public static class Builder {

        private final String type;
        private String size = "中杯";
        private boolean pearl = true;
        private boolean ice = false;

        public Builder(String type) {
            this.type = type;
        }

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

        public Builder pearl(boolean pearl) {
            this.pearl = pearl;
            return this;
        }

        public Builder ice(boolean cold) {
            this.ice = cold;
            return this;
        }

        public MilkTea build() {
            return new MilkTea(this);
        }
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/99sjd7/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **原型模式 Prototype**　用于克隆对象

用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。简单理解就是克隆。为一个类定义clone方法，使得创建相同的对象更方便。

▼ 代码实现

```
//java语法糖， 继承Cloneable接口， 但该接口只是浅拷贝
public class MilkTea implements Cloneable{
    public String type;
    public boolean ice;

    @NonNull
    @Override
    protected MilkTea clone() throws CloneNotSupportedException {
        return (MilkTea) super.clone();
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/994yw5/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

结构型模式

▼ **适配器模式 Adapter**　用于有相关性但不兼容的接口

适用于有相关性，但不兼容的结构。将一个类的接口转换成客户希望的另一个接口，使得原本由于接口不兼容而不能在一起工作的那些类能一起工作。

▼ 代码实现

```
class HomeBattery {
    int supply() {
        // 家用电源提供一个 220V 的输出电压
        return 220;
    }
}

class USBLine {
    void charge(int volt) {
        // 如果电压不是 5V，抛出异常
        if (volt != 5) throw new IllegalArgumentException("只能接收 5V 电压");
        // 如果电压是 5V，正常充电
        System.out.println("正常充电");
    }
}

class Adapter {
    int convert(int homeVolt) {
        // 适配过程：使用电阻、电容等器件将其降低为输出 5V
        int chargeVolt = homeVolt - 215;
        return chargeVolt;
    }
}

public class User {
    @Test
    public void chargeForPhone() {
        HomeBattery homeBattery = new HomeBattery();
        int homeVolt = homeBattery.supply();
        System.out.println("家庭电源提供的电压是 " + homeVolt + "V");

        Adapter adapter = new Adapter();
        int chargeVolt = adapter.convert(homeVolt);
        System.out.println("使用适配器将家庭电压转换成了 " + chargeVolt + "V");

        USBLine usbLine = new USBLine();
        usbLine.charge(chargeVolt);
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/99yk22/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **桥接模式 Bridge**　用于同等级的接口互相组合

桥接模式体现了【合成 / 聚合复用原则】， 有限使用合成 / 聚合， 而不是类继承。

将抽象部分与它的实现部分分离，使他们都可以独立地变化。它是一种对象结构型模式。

▼ 代码实现

```
public interface IColor {
    String getColor();
}

public class Red implements IColor {
    @Override
    public String getColor() {
        return "红";
    }
}

class Rectangle implements IShape {

    private IColor color;

    void setColor(IColor color) {
        this.color = color;
    }

    @Override
    public void draw() {
        System.out.println("绘制" + color.getColor() + "矩形");
    }
}
```

```
@Test
public void drawTest() {
    Rectangle rectangle = new Rectangle();
    rectangle.setColor(new Red());
    rectangle.draw();
}


作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/99c3mc/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **组合模式 Composite**　用于整体与部分的结构

组合模式用于整体与部分的结构，当整体与部分有相似的结构，在操作时可以被一直对待时，可以使用组合模式， 把一组相似的对象当作一个单一的对象。

组合模式中的安全方式与透明方式

透明方式：个别实体可能不需要某些属性，但因为透明方式， 需要进行空实现。这样的缺点是违背了接口隔离原则， 但好处是可以统一对待所有实体。而安全方式是把某些不共有的属性剥离出去， 使用的时候就需要特别对待。实践中， 更常使用透明方式。

▼ 代码实现

```
public abstract class Component {
    // 职位
    private String position;
    // 工作内容
    private String job;

    public Component(String position, String job) {
        this.position = position;
        this.job = job;
    }

    // 做自己的本职工作
    public void work() {
        System.out.println("我是" + position + "，我正在" + job);
    }

    abstract void addComponent(Component component);

    abstract void removeComponent(Component component);

    abstract void check();
}

public class Manager extends Component {
    // 管理的组件
    private List<Component> components = new ArrayList<>();

    public Manager(String position, String job) {
        super(position, job);
    }

    @Override
    public void addComponent(Component component) {
        components.add(component);
    }

    @Override
    void removeComponent(Component component) {
        components.remove(component);
    }

    // 检查下属
    @Override
    public void check() {
        work();
        for (Component component : components) {
            component.check();
        }
    }
}

public class Employee extends Component {

    public Employee(String position, String job) {
```

```
        super(position, job);
    }

    @Override
    void addComponent(Component component) {
        System.out.println("职员没有管理权限");
    }

    @Override
    void removeComponent(Component component) {
        System.out.println("职员没有管理权限");
    }

    @Override
    void check() {
        work();
    }
}

public class Client {

    @Test
    public void test(){
        Component boss = new Manager("老板", "唱怒放的生命");
        Component HR = new Employee("人力资源", "聊微信");
        Component PM = new Manager("产品经理", "不知道干啥");
        Component CFO = new Manager("财务主管", "看剧");
        Component CTO = new Manager("技术主管", "划水");
        Component UI = new Employee("设计师", "画画");
        Component operator = new Employee("运营人员", "兼职客服");
        Component webProgrammer = new Employee("程序员", "学习设计模式");
        Component backgroundProgrammer = new Employee("后台程序员", "CRUD");
        Component accountant = new Employee("会计", "背九九乘法表");
        Component clerk = new Employee("文员", "给老板递麦克风");
        boss.addComponent(HR);
        boss.addComponent(PM);
        boss.addComponent(CFO);
        PM.addComponent(UI);
        PM.addComponent(CTO);
        PM.addComponent(operator);
        CTO.addComponent(webProgrammer);
        CTO.addComponent(backgroundProgrammer);
        CFO.addComponent(accountant);
        CFO.addComponent(clerk);

        boss.check();
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/99bupi/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **装饰模式 Decorator**　用于增强功能或添加功能

动态地给一个对象增强原有功能或增加一些新功能。

增强功能的实现叫透明装饰模式，可以无限装饰。

增加功能的实现就半透明装饰模式，无法多次装饰。

装饰模式与适配器模式实现方式上相似， 但概念不同。

▼ 代码实现

```
//透明装饰模式， 增强原有功能
public interface IBeauty {
    int getBeautyValue();
}

public class Me implements IBeauty {

    @Override
    public int getBeautyValue() {
        return 100;
    }
}

public class RingDecorator implements IBeauty {
    private final IBeauty me;
```

```
    public RingDecorator(IBeauty me) {
        this.me = me;
    }

    @Override
    public int getBeautyValue() {
        return me.getBeautyValue() + 20;
    }
}

public class Client {
    @Test
    public void show() {
        IBeauty me = new Me();
        System.out.println("我原本的颜值：" + me.getBeautyValue());

        // 随意挑选装饰
        IBeauty meWithNecklace = new NecklaceDecorator(me);
        System.out.println("戴上了项链后，我的颜值：" + meWithNecklace.getBeautyValue());

        // 多次装饰
        IBeauty meWithManyDecorators = new NecklaceDecorator(new RingDecorator(new EarringDecorator(me)));
        System.out.println("戴上耳环、戒指、项链后，我的颜值：" + meWithManyDecorators.getBeautyValue());

        // 任意搭配装饰
        IBeauty meWithNecklaceAndRing = new NecklaceDecorator(new RingDecorator(me));
        System.out.println("戴上戒指、项链后，我的颜值：" + meWithNecklaceAndRing.getBeautyValue());
    }
}



//半透明装饰模式， 增加新功能

public interface IHouse {
    void live();
}

public class House implements IHouse{

    @Override
    public void live() {
        System.out.println("房屋原有的功能：居住功能");
    }
}

public interface IStickyHookHouse extends IHouse{
    void hangThings();
}

public class StickyHookDecorator implements IStickyHookHouse {
    private final IHouse house;

    public StickyHookDecorator(IHouse house) {
        this.house = house;
    }

    @Override
    public void live() {
        house.live();
    }

    @Override
    public void hangThings() {
        System.out.println("有了粘钩后，新增了挂东西功能");
    }
}

public class Client {
    @Test
    public void show() {
        IHouse house = new House();
        house.live();

        IStickyHookHouse stickyHookHouse = new StickyHookDecorator(house);
        stickyHookHouse.live();
        stickyHookHouse.hangThings();
    }
}
```

▼ **外观模式 Facade**　体现封装的思想

体现了面向对象封装的思想，将多个子系统封装起来，提供一个更简洁的接口供外部调用，外部与一个子系统的通信必须通过一个统一的外观对象进行。

▼ 代码实现

```
public class Browser {
    public static void open() {
        System.out.println("打开浏览器");
    }

    public static void close() {
        System.out.println("关闭浏览器");
    }
}

public class IDE {
    public static void open() {
        System.out.println("打开 IDE");
    }

    public static void close() {
        System.out.println("关闭 IDE");
    }
}

public class Wechat {
    public static void open() {
        System.out.println("打开微信");
    }

    public static void close() {
        System.out.println("关闭微信");
    }
}

public class Facade {
    public void open() {
        Browser.open();
        IDE.open();
        Wechat.open();
    }

    public void close() {
        Browser.close();
        IDE.close();
        Wechat.close();
    }
}

public class Client {
    @Test
    public void test() {
        Facade facade = new Facade();
        System.out.println("上班:");
        facade.open();

        System.out.println("下班:");
        facade.close();
    }
}


作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/99fweg/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **享元模式 Flyweight**　体现面向对象的可复用性

简单来说就是共享对象，提高复用性

运用共享技术，有效地支持大量细粒度对象的复用。系统只使用少量的对象，而这些对象都很相似，状态变化很小，可以实现对象的多次复用。

超级玛丽里的草和云更改一下颜色就实现了复用，乌龟换了一种颜色就变成了不同的怪。正是这些精湛的复用，使得这款游戏仅有40KB。

▼ **代理模式 Proxy**　主要用于对某个对象加以控制

给某一个对象提供一个代理，并由代理对象控制对原对象的引用

代理模式分为静态代理和动态代理。区别是动态代理编码量少

▼ 静态代理代码实现

```
public class HttpProxy implements IHttp {
    private final HttpUtil httpUtil;

    public HttpProxy(HttpUtil httpUtil) {
        this.httpUtil = httpUtil;
    }

    @Override
    public void request(String sendData) {
        httpUtil.request(sendData);
    }

    @Override
    public void onSuccess(String receivedData) {
        httpUtil.onSuccess(receivedData);
    }
}

public class HttpProxy implements IHttp {
    private final HttpUtil httpUtil;

    public HttpProxy(HttpUtil httpUtil) {
        this.httpUtil = httpUtil;
    }

    @Override
    public void request(String sendData) {
        System.out.println("发送数据:" + sendData);
        httpUtil.request(sendData);
    }

    @Override
    public void onSuccess(String receivedData) {
        System.out.println("收到数据:" + receivedData);
        httpUtil.onSuccess(receivedData);
    }
}

public class Client {
    @Test
    public void test() {
        HttpUtil httpUtil = new HttpUtil();
        HttpProxy proxy = new HttpProxy(httpUtil);
        proxy.request("request data");
        proxy.onSuccess("received result");
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/9e5eht/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ 动态代理代码实现

```
public class HttpProxy implements InvocationHandler {
    private HttpUtil httpUtil;

    public IHttp getInstance(HttpUtil httpUtil) {
        this.httpUtil = httpUtil;
        return (IHttp) Proxy.newProxyInstance(httpUtil.getClass().getClassLoader(), httpUtil.getClass().getInterfaces()
    }

    // 调用 httpUtil 的任意方法时，都要通过这个方法调用
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Object result = null;
        if (method.getName().equals("request")) {
            // 如果方法名是 request，打印日志，并调用 request 方法
            System.out.println("发送数据:" + args[0]);
```

```
            result = method.invoke(httpUtil, args);
        } else if (method.getName().equals("onSuccess")) {
            // 如果方法名是 onSuccess，打印日志，并调用 onSuccess 方法
            System.out.println("收到数据:" + args[0]);
            result = method.invoke(httpUtil, args);
        }
        return result;
    }
}
```

行为型模式

▼ **责任链模式 Chain of responsibility**　处理职责相同, 程度不同的对象, 使其在一条链上传递

主要用于处理职责相同, 程度不同的类

使多个对象都有机会处理请求, 将请求的发送者与接受者解耦. 将这些对象连城一条链, 并沿着这条链传递该请求, 直到有一个对象处理它为止

▼ 优点

1. 降低了对象之间的耦合度
2. 拓展性强, 满足开闭原则
3. 灵活性强, 可以改变责任链的成员次序来适应流程的变化
4. 责任分担, 符合类的单一职责原则

▼ 缺点

1. 不能保证每个请求一定被处理
2. 责任链过长会影响性能
3. 责任链建立的合理性要靠客户端来保证, 增加了客户端的复杂性. 可能会因为责任链拼接错误而导致系统出错

▼ 代码实现

```
public abstract class Programmer {
    protected Programmer next;

    public void setNext(Programmer next) {
        this.next = next;
    }

    abstract void handle(Bug bug);
}

public class NewbieProgrammer extends Programmer {

    @Override
    public void handle(Bug bug) {
        if (bug.value > 0 && bug.value <= 20) {
            solve(bug);
        } else if (next != null) {
            next.handle(bug);
        }
    }

    private void solve(Bug bug) {
        System.out.println("菜鸟程序员解决了一个难度为 " + bug.value + " 的 bug");
    }
}

public class NormalProgrammer extends Programmer {

    @Override
    public void handle(Bug bug) {
        if (bug.value > 20 && bug.value <= 50) {
            solve(bug);
        } else if (next != null) {
            next.handle(bug);
        }
    }

    private void solve(Bug bug) {
```

```
        System.out.println("普通程序员解决了一个难度为 " + bug.value + " 的 bug");
    }
}

public class GoodProgrammer extends Programmer {

    @Override
    public void handle(Bug bug) {
        if (bug.value > 50 && bug.value <= 100) {
            solve(bug);
        } else if (next != null) {
            next.handle(bug);
        }
    }

    private void solve(Bug bug) {
        System.out.println("优秀程序员解决了一个难度为 " + bug.value + " 的 bug");
    }
}

public class Client4 {
    @Test
    public void test() {
        NewbieProgrammer newbie = new NewbieProgrammer();
        NormalProgrammer normal = new NormalProgrammer();
        GoodProgrammer good = new GoodProgrammer();

        Bug easy = new Bug(20);
        Bug middle = new Bug(50);
        Bug hard = new Bug(100);

        // 组成责任链
        newbie.setNext(normal);
        normal.setNext(good);

        // 从菜鸟程序员开始，沿着责任链传递
        newbie.handle(easy);
        newbie.handle(middle);
        newbie.handle(hard);
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/9e9hfr/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **命令模式 Command**　封装方法调用, 将行为请求者和行为实现者解耦

将请求封装为一个对象, 把请求的实现封装在里面, 对外只暴露统一的执行方法, 实现了将行为请求者 与 行为实现者解耦

命令对象的唯一职责就是通过execute去调用一个方法, 它把方法调用这个步骤封装起来了, 使得我们可以对方法调用进行排队、撤销等处理

▼ 优点

1. 降低系统的耦合度, 将行为请求者与行为实现者解耦
2. 拓展性强. 增加或删除命令不影响其他命令
3. 封装方法调用, 方便实现Undo 和 Redo操作
4. 灵活性强, 可以实现宏指令

▼ 缺点

1. 会产生大量的命令类. 增加系统的复杂性

▼ 代码实现

```
public interface ICommand {

    void execute();

    void undo();
}

public class DoorOpenCommand implements ICommand {
    private Door door;
```

```
    public void setDoor(Door door) {
        this.door = door;
    }

    @Override
    public void execute() {
        door.openDoor();
    }

    @Override
    public void undo() {
        door.closeDoor();
    }
}

public class DoorCloseCommand implements ICommand {
    private Door door;

    public void setDoor(Door door) {
        this.door = door;
    }

    @Override
    public void execute() {
        door.closeDoor();
    }

    @Override
    public void undo() {
        door.openDoor();
    }
}

public class Client {

    // 所有的命令
    Stack<ICommand> commands = new Stack<>();

    @Test
    protected void test() {
        ...初始化

        // 大门开关遥控
        switchDoor.setOnCheckedChangeListener((view, isChecked) -> {
            handleCommand(isChecked, doorOpenCommand, doorCloseCommand);
        });
        // 电灯开关遥控
        switchLight.setOnCheckedChangeListener((view, isChecked) -> {
            handleCommand(isChecked, lightOnCommand, lightOffCommand);
        });

        // 撤销按钮
        btnUndo.setOnClickListener(view -> {
            if (commands.isEmpty()) return;
            // 撤销上一个命令
            ICommand lastCommand = commands.pop();
            lastCommand.undo();
        });
    }

    private void handleCommand(boolean isChecked, ICommand openCommand, ICommand closeCommand) {
        if (isChecked) {
            commands.push(openCommand);
            openCommand.execute();
        } else {
            commands.push(closeCommand);
            closeCommand.execute();
        }
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/9e9bbs/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **解释器模式 Interpreter**　定义自己的语法规则

给定一门语言, 定义它的文法的一种表示, 并定义一个解释器, 该解释器使用该文法来解释语言中的句子

在解释器模式中, 不可拆分的最小单元称为终结表达式, 可以被拆分的表达式称为非终结表达式.

正则表达式就是一个解释器

▼ 代码实现

```
interface Expression {
    int intercept();
}

public class Number implements Expression {
    int number;

    public Number(char word) {
        switch (word) {
            case '零':
                number = 0;
                break;
            case '一':
                number = 1;
                break;
            case '二':
                number = 2;
                break;
            case '三':
                number = 3;
                break;
            case '四':
                number = 4;
                break;
            case '五':
                number = 5;
                break;
            case '六':
                number = 6;
                break;
            case '七':
                number = 7;
                break;
            case '八':
                number = 8;
                break;
            case '九':
                number = 9;
                break;
            default:
                break;
        }
    }

    @Override
    public int intercept() {
        return number;
    }
}

abstract class Operator implements Expression {
    Expression left;
    Expression right;

    Operator(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }
}

class Add extends Operator {

    Add(Expression left, Expression right) {
        super(left, right);
    }

    @Override
    public int intercept() {
        return left.intercept() + right.intercept();
    }
}

class Sub extends Operator {

    Sub(Expression left, Expression right) {
        super(left, right);
    }

    @Override
    public int intercept() {
        return left.intercept() - right.intercept();
```

```
    }
}

class Calculator {
    int calculate(String expression) {
        Stack<Expression> stack = new Stack<>();
        for (int i = 0; i < expression.length(); i++) {
            char word = expression.charAt(i);
            switch (word) {
                case '加':
                    stack.push(new Add(stack.pop(), new Number(expression.charAt(++i))));
                    break;
                case '减':
                    stack.push(new Sub(stack.pop(), new Number(expression.charAt(++i))));
                    break;
                default:
                    stack.push(new Number(word));
                    break;
            }
        }
        return stack.pop().intercept();
    }
}

public class Client {
    @Test
    public void test() {
        Calculator calculator = new Calculator();
        String expression1 = "一加一";
        String expression2 = "一加一加一";
        String expression3 = "二加五减三";
        String expression4 = "七减五加四减一";
        String expression5 = "九减五加三减一";
        // 输出： 一加一 等于 2
        System.out.println(expression1 + " 等于 " + calculator.calculate(expression1));
        // 输出： 一加一加一 等于 3
        System.out.println(expression2 + " 等于 " + calculator.calculate(expression2));
        // 输出： 二加五减三 等于 4
        System.out.println(expression3 + " 等于 " + calculator.calculate(expression3));
        // 输出： 七减五加四减一 等于 5
        System.out.println(expression4 + " 等于 " + calculator.calculate(expression4));
        // 输出： 九减五加三减一 等于 6
        System.out.println(expression5 + " 等于 " + calculator.calculate(expression5));
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/9ee1cd/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **迭代器模式 Iterator**　让外部类通过next, hasNext来遍历列表, 隐藏内部细节

提供一种方法访问一个容器对象中的各个元素，而又不暴露该对象的内部结构

▼ 代码实现

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
}

public interface Iterable<T> {
    Iterator<T> iterator();
    ...
}

public class ArrayList<E> implements Iterable<E>{

    @NonNull
    @Override
    public Iterator<String> iterator() {
        // 每次生成一个新的迭代器，用于遍历列表
        return new Itr();
    }

    private class Itr implements Iterator<E> {
        protected int limit = ArrayList.this.size;
        int cursor;
```

```
        public boolean hasNext() {
            return cursor < limit;
        }

        public E next() {
            ...
        }
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/9eozyh/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **中介者模式 Mediator**　通过引入中介者, 将网状耦合结构变成星型结构

定义一个中介对象来封装一系列对象之间的交互，使原有对象之间的耦合松散， 且可以独立地改变它们之间的交互。

将类与类之间的多对多关系， 简化成多对一， 一对多关系。把网状结构变成星状结构。

▼ 代码实现

```
class Group {
    public int money;
}

class Player {
    public int money = 100;
    public Group group;

    public Player(Group group) {
        this.group = group;
    }

    public void change(int money) {
        // 输了钱将钱发到群里 或 在群里领取自己赢的钱
        group.money += money;
        // 自己的余额改变
        this.money += money;
    }
}

public class Client {
    @Test
    public void test(){
        Group group = new Group();
        Player player1 = new Player(group);
        Player player2 = new Player(group);
        Player player3 = new Player(group);
        Player player4 = new Player(group);
        // player1 赢了 5 元
        player1.change(5);
        // player2 赢了 20 元
        player2.change(20);
        // player3 输了 12 元
        player3.change(-12);
        // player4 输了 3 元
        player4.change(-3);

        // 输出：四人剩余的钱：105,120,88,97
        System.out.println("四人剩余的钱：" + player1.money + "," + player2.money + "," + player3.money + "," + player4.mo
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/9er643/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **备忘录模式 Memento**　存储对象的状态, 以便恢复

在不破坏封装的条件下， 通过备忘录对象存储另一个对象内部状态的快照，在将来合适的时候把这个对象还原到存储起来的状态。

▼ 代码实现

```
class Memento {
    int life;
    int magic;

    Memento(int life, int magic) {
        this.life = life;
        this.magic = magic;
    }
}

class Player {
    ...

    // 存档
    public Memento saveState() {
        return new Memento(life, magic);
    }

    // 读档
    public void restoreState(Memento memento) {
        this.life = memento.life;
        this.magic = memento.magic;
    }
}

public class Client {
    @Test
    public void test() {
        Player player = new Player();
        // 存档
        Memento memento = player.saveState();

        // 打 Boss，打不过，壮烈牺牲
        player.fightBoss();

        // 读档
        player.restoreState(memento);
    }
}
```


▼ **观察者模式 Observer**　处理一对多的依赖关系, 被观察的对象改变时, 多个观察者都能收到通知

定义对象间的一种一对多的依赖关系， 当一个对象的状态发生改变时， 所有依赖于它的对象都得到通知并被自动更新

被观察者中维护了一个观察者列表

▼ 代码实现

```
public interface Observer {
    void update(Observable o, Object arg);
}

public class Observable {
    private boolean changed = false;
    private Vector<Observer> obs;

    public Observable() {
        obs = new Vector<>();
    }

    public synchronized void addObserver(java.util.Observer o) {
        if (o == null)
            throw new NullPointerException();
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }

    public synchronized void deleteObserver(java.util.Observer o) {
        obs.removeElement(o);
    }

    public void notifyObservers() {
```

```
        notifyObservers(null);
    }

    public void notifyObservers(Object arg) {
        Object[] arrLocal;
        synchronized (this) {
            if (!hasChanged())
                return;
            arrLocal = obs.toArray();
            clearChanged();
        }
        for (int i = arrLocal.length - 1; i >= 0; i--)
            ((Observer) arrLocal[i]).update(this, arg);
    }

    public synchronized void deleteObservers() {
        obs.removeAllElements();
    }

    protected synchronized void setChanged() {
        changed = true;
    }

    protected synchronized void clearChanged() {
        changed = false;
    }

    public synchronized boolean hasChanged() {
        return changed;
    }

    public synchronized int countObservers() {
        return obs.size();
    }
}
```


▼ **状态模式 State**　关于多态的设计模式, 每个状态类处理对象的一种状态

当一个对象的内在状态改变时允许改变其行为, 这个对象看起来像是改变了其类

如果一个对象有多种状态, 并且每种状态下的行为不同, 一般的做法是在这个对象的各个行为中添加if-else或switch语句, 违反了开闭原则. 但更好但做法是为每种状态创建一个状态对象, 使用状态对象替换掉这些条件判断语句, 使得状态控制更加灵活, 拓展性也更好

开发中应该多使用多态取代条件表达式

▼ 优点

将与特定状态相关的行为封装到一个状态对象中, 使用多态代替if-else 或 switch-case 状态判断

▼ 缺点

导致类膨胀

▼ 代码实现

```
class Normal implements IUser {

    @Override
    public void mockInterview() {
        System.out.println("模拟面试是 Plus 会员专享功能");
    }
}

class Plus implements IUser {

    @Override
    public void mockInterview() {
        System.out.println("开始模拟面试");
    }
}

class User implements IUser, ISwitchState {

    IUser state = new Normal();
```

```
    @Override
    public void mockInterview() {
        state.mockInterview();
    }

    @Override
    public void purchasePlus() {
        state = new Plus();
    }

    @Override
    public void expire() {
        state = new Normal();
    }
}

public class Client {

    @Test
    public void test() {
        // 用户初始状态为普通用户
        User user = new User();
        // 输出：模拟面试是 Plus 会员专享功能
        user.mockInterview();

        // 用户购买 Plus 会员，状态改变
        user.purchasePlus();
        // 输出：开始模拟面试
        user.mockInterview();

        // Plus 会员过期，变成普通用户，状态改变
        user.expire();
        // 输出：模拟面试是 Plus 会员专享功能
        user.mockInterview();
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/9ea0k7/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **策略模式 Strategy**　殊途同归, 用多种方法做同一件事

定义一系列的算法, 并将每一个算法封装起来, 而且使他们还可以相互替换. 策略模式让算法独立于使用它的客户而独立变化

实践中, 一般策略模式会与工厂模式结合使用, 让使用者只了解配置, 而无需了解具体的策略类

策略模式与状态模式非常相似, 但策略模式强调的是殊途同归, 每一个策略都是完整的, 都能单独完成同一件事情. 而状态模式强调的是随势而动, 每个状态只能完成这件事的一部分, 需要所有的状态类组合起来才能完成整件事情

▼ 代码实现

```
enum SortStrategy {
    BUBBLE_SORT,
    SELECTION_SORT,
    INSERT_SORT
}

class Sort implements ISort {

    private ISort sort;

    Sort(SortStrategy strategy) {
        setStrategy(strategy);
    }

    @Override
    public void sort(int[] arr) {
        sort.sort(arr);
    }

    // 客户端通过此方法设置不同的策略
    public void setStrategy(SortStrategy strategy) {
        switch (strategy) {
            case BUBBLE_SORT:
                sort = new BubbleSort();
                break;
            case SELECTION_SORT:
                sort = new SelectionSort();
```

```
                break;
            case INSERT_SORT:
                sort = new InsertSort();
                break;
            default:
                throw new IllegalArgumentException("There's no such strategy yet.");
        }
    }
}

public class Client {
    @Test
    public void test() {
        int[] arr = new int[]{6, 1, 2, 3, 5, 4};
        Sort sort = new Sort(SortStrategy.BUBBLE_SORT);
        // 可以通过选择不同的策略完成排序
        // sort.setStrategy(SortStrategy.SELECTION_SORT);
        // sort.setStrategy(SortStrategy.INSERT_SORT);
        sort.sort(arr);
        // 输出 [1, 2, 3, 4, 5, 6]
        System.out.println(Arrays.toString(arr));
    }
}

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/9e2v65/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **模版方法 Template method**　关于继承的设计模式, 父类是子类的模版

定义一个操作中的算法的骨架, 将一些步骤延迟到子类中, 模版方法使得子类可以不改变一个算法的结构即可重新定义算法的某些特定步骤

在使用模版方法模式时, 我们可以为不同的模版方法设置不同的控制权限:

1. 如果不希望子类覆写模版中的某个方法, 使用final修饰此方法
2. 如果要求子类必须覆写模版中的方法, 使用abstract修饰此方法
3. 如果没有特殊要求, 可使用protected 或 public修饰此方法, 子类可以根据世纪情况考虑是否覆写

▼ 代码实现

```
abstract class LeaveRequest {

    void request() {
        System.out.print("本人");
        System.out.print(name());
        System.out.print("因");
        System.out.print(reason());
        System.out.print("需请假");
        System.out.print(duration());
        System.out.print("天，望批准");
    }

    abstract String name();

    abstract String reason();

    abstract String duration();
}

class MyLeaveRequest extends LeaveRequest {
    @Override
    String name() {
        return "阿笠";
    }

    @Override
    String reason() {
        return "参加力扣周赛";
    }

    @Override
    String duration() {
        return "0.5";
    }
}
```

```
// 输出：本人阿笠因参加力扣周赛需请假0.5天，望批准
new MyLeaveRequest().request();

作者：力扣 (LeetCode)
链接：https://leetcode-cn.com/leetbook/read/design-patterns/9ena36/
来源：力扣（LeetCode）
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

▼ **访问者模式 Visitor**　将数据的结构和对数据的操作分离

表示一个作用于某对象结构中的各元素的操作. 它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作. 简单说就是将 数据的结构 与 对数据的操作 分离

该模式的男单在于, 大多数语言都是单分派语言, 所以不得不模拟出一个双重分派, 也就是用重写方法的动态分派特性将重载方法也模拟成动态分派

▼ 代码实现

```
public abstract class Food {
    public abstract String name();

    // Food 中添加 accept 方法，接收访问者
    public abstract void accept(IVisitor visitor);
}

public class Lobster extends Food {
    @Override
    public String name() {
        return "lobster";
    }

    @Override
    public void accept(IVisitor visitor) {
        visitor.chooseFood(this);
    }
}

public class Aurora implements IVisitor {

    @Override
    public void chooseFood(Lobster lobster) {
        System.out.println("Aurora gets a " + lobster.name());
    }

    @Override
    public void chooseFood(Watermelon watermelon) {
        System.out.println("Aurora gets a " + watermelon.name());
    }

    @Override
    public void chooseFood(Steak steak) {
        System.out.println("Aurora doesn't like " + steak.name());
    }

    @Override
    public void chooseFood(Banana banana) {
        System.out.println("Aurora doesn't like " + banana.name());
    }
}

class Restaurant {

    // 准备当天的食物
    private List<Food> prepareFoods() {
        List<Food> foods = new ArrayList<>();
        // 简单模拟，每种食物添加 10 份
        for (int i = 0; i < 10; i++) {
            foods.add(new Lobster());
            foods.add(new Watermelon());
            foods.add(new Steak());
            foods.add(new Banana());
        }
        return foods;
    }

    // 欢迎顾客来访
    public void welcome(IVisitor visitor) {
        // 获取当天的食物
```

```
        List<Food> foods = prepareFoods();
        // 将食物依次提供给顾客选择
        for (Food food : foods) {
            // 由于重写方法是动态分派的，所以这里会调用具体子类的 accept 方法，
            food.accept(visitor);
        }
    }
}

public class Client {
    @Test
    public void test() {
        Restaurant restaurant = new Restaurant();
        IVisitor Aurora = new Aurora();
        IVisitor Kevin = new Kevin();
        restaurant.welcome(Aurora);
        restaurant.welcome(Kevin);
    }
}
```