链滴

《Head First 设计模式》：剩下的模式

作者：jingqueyimu
原文链接：https://ld246.com/article/1603635735739
来源网站：链滴

正文

一、桥接模式

1、定义

桥接模式通过将实现和抽象分离开来，放在两个不同的类层次中，从而使得它们可以独立改变。

要点：

- 当一个类存在两个独立变化的维度，而且都需要进行扩展时，可以将其中一个维度抽象化，另一个度实现化。
- 抽象化就是通过抽象类来实现多态，实现化则是通过接口来实现多态。
- 桥接模式通过在抽象类中持有实现类接口，来将两个维度“桥接”起来。

2、实现步骤

（1）创建实现化角色接口

```
/**
 * 实现化角色接口
 */
public interface Implementor {

    void action();
}
```

（2）创建具体实现化角色

```
/**
 * 具体实现化角色A
 */
public class ConcreteImplementorA implements Implementor {

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

/**
 * 具体实现化角色B
 */
public class ConcreteImplementorB implements Implementor {

    @Override
    public void action() {
        System.out.println("ConcreteImplementorB action");
    }
}
```

（3）创建抽象化角色抽象类，并持有实现化角色接口

```
/**
 * 抽象化角色抽象类
 */
public abstract class Abstraction {

    /**
     * 实现化角色接口
     */
    Implementor implementor;

    public Abstraction(Implementor implementor) {
        this.implementor = implementor;
    }

    public abstract void action();
}
```

（4）创建具体抽象化角色

```
/**
 * 具体抽象化角色A
 */
public class ConcreteAbstractionA extends Abstraction {

    public ConcreteAbstractionA(Implementor implementor) {
        super(implementor);
    }
```

```
    @Override
    public void action() {
        System.out.print("ConcreteAbstractionA action --> ");
        // 调用实现化角色的方法
        implementor.action();
    }
}

/**
 * 具体抽象化角色B
 */
public class ConcreteAbstractionB extends Abstraction {

    public ConcreteAbstractionB(Implementor implementor) {
        super(implementor);
    }

    @Override
    public void action() {
        System.out.print("ConcreteAbstractionB action --> ");
        // 调用实现化角色的方法
        implementor.action();
    }
}
```

（5）组合抽象化角色与实现化角色

通过组合抽象化角色与实现化角色，来实现更多的功能。

```
public class Test {

    public static void main(String[] args) {
        // 实现化角色
        Implementor implementorA = new ConcreteImplementorA();
        Implementor implementorB = new ConcreteImplementorB();
        // 抽象化角色
        Abstraction abstractionAA = new ConcreteAbstractionA(implementorA);
        Abstraction abstractionAB = new ConcreteAbstractionA(implementorB);
        Abstraction abstractionBA = new ConcreteAbstractionB(implementorA);
        Abstraction abstractionBB = new ConcreteAbstractionB(implementorB);
        // 请求动作
        abstractionAA.action();
        abstractionAB.action();
        abstractionBA.action();
        abstractionBB.action();
    }
}
```

二、生成器模式（建造者模式）

1、定义

生成器模式封装一个产品的构造过程，并允许按步骤构造。

要点：

- 将一个复杂对象的创建过程封装起来。
- 允许对象通过多个步骤来创建，并且可以改变过程（这和只有一个步骤的工厂模式不同）。

2、实现步骤

（1）创建产品类

```
/**
 * 产品
 */
public class Product {

    /**
     * 产品部件1
     */
    private String part1;

    /**
     * 产品部件2
     */
    private String part2;

    /**
     * 产品部件3
     */
    private String part3;

    public String getPart1() {
        return part1;
    }

    public void setPart1(String part1) {
        this.part1 = part1;
    }

    public String getPart2() {
        return part2;
    }

    public void setPart2(String part2) {
        this.part2 = part2;
    }

    public String getPart3() {
        return part3;
    }

    public void setPart3(String part3) {
```

```
        this.part3 = part3;
    }

    @Override
    public String toString() {
        return "Product [part1=" + part1 + ", part2=" + part2 + ", part3=" + part3 + "]";
    }
}
```

（2）创建生成器抽象类

```
/**
 * 生成器抽象类
 */
public abstract class Builder {

    protected Product product = new Product();

    public abstract void buildPart1();

    public abstract void buildPart2();

    public abstract void buildPart3();

    /**
     * 获取产品
     */
    public Product getProduct() {
        return product;
    }
}
```

（3）创建具体生成器

```
/**
 * 具体生成器
 */
public class ConcreteBuilder extends Builder {

    @Override
    public void buildPart1() {
        product.setPart1("product part 1");
    }

    @Override
    public void buildPart2() {
        product.setPart2("product part 2");
    }

    @Override
    public void buildPart3() {
        product.setPart3("product part 3");
    }
```

```
}
```

（4）使用生成器生成产品

```
public class Test {

    public static void main(String[] args) {
        // 生成器
        Builder builder = new ConcreteBuilder();
        // 生成产品
        builder.buildPart1();
        builder.buildPart2();
        builder.buildPart3();
        // 获取产品
        Product product = builder.getProduct();
        System.out.println(product);
    }
}
```

三、责任链模式

1、定义

责任链模式为某个请求创建一个对象链。每个对象依序检查此请求，并对其进行处理，或者将它传给中的下一个对象。

要点：

- 将请求的发送者和接受者解耦。
- 通过改变链内成员或调动它们的次序，允许你动态地新增或删除责任。

2、实现步骤

（1）创建请求数据包类

```
/**
 * 请求数据包
 */
public class Request {

    /**
     * 级别
     */
    private int level;

    /**
     * 数据
     */
    private String data;
```

```
    public Request(int level, String data) {
        this.level = level;
        this.data = data;
    }

    public int getLevel() {
        return level;
    }

    public void setLevel(int level) {
        this.level = level;
    }

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }
}
```

（2）创建处理器抽象类

```
/**
 * 处理器抽象类
 */
public abstract class Handler {

    /**
     * 下一个处理器
     */
    protected Handler nextHandler;

    public Handler getNextHandler() {
        return nextHandler;
    }

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

    /**
     * 处理请求
     */
    protected abstract void handleRequest(Request request);
}
```

（3）创建具体处理器

```
/**
 * 具体处理器A
 */
```

```
public class ConcreteHandlerA extends Handler {

    @Override
    protected void handleRequest(Request request) {
        if (request.getLevel() <= 1) {
            System.out.println("ConcreteHandlerA is handling the request, data: " + request.getDa
a());
        } else {
            getNextHandler().handleRequest(request);
        }
    }
}

/**
 * 具体处理器B
 */
public class ConcreteHandlerB extends Handler {

    @Override
    protected void handleRequest(Request request) {
        if (request.getLevel() <= 2) {
            System.out.println("ConcreteHandlerB is handling the request, data: " + request.getDa
a());
        } else {
            getNextHandler().handleRequest(request);
        }
    }
}

/**
 * 具体处理器C
 */
public class ConcreteHandlerC extends Handler {

    @Override
    protected void handleRequest(Request request) {
        if (request.getLevel() <= 3) {
            System.out.println("ConcreteHandlerC is handling the request, data: " + request.getDa
a());
        } else {
            System.out.println("No handler can handle the request...");
        }
    }
}
```

（4）使用处理器链处理请求

```
public class Test {

    public static void main(String[] args) {
        // 创建责任链（处理器链）
        Handler handlerA = new ConcreteHandlerA();
        Handler handlerB = new ConcreteHandlerB();
        Handler handlerC = new ConcreteHandlerC();
```

```
        handlerA.setNextHandler(handlerB);
        handlerB.setNextHandler(handlerC);
        // 使用责任链处理请求
        handlerA.handleRequest(new Request(1, "请求1"));
        handlerA.handleRequest(new Request(2, "请求2"));
        handlerA.handleRequest(new Request(3, "请求3"));
        handlerA.handleRequest(new Request(4, "请求4"));
    }
}
```

四、蝇量模式（享元模式）

1、定义

蝇量模式能让某个类的一个实例能用来提供许多“虚拟实例”。

要点：

- 运用共享技术，减少运行时对象实例的个数，节省内存。
- 当一个类有许多实例，而这些实例能被同一个方法控制时，可以使用蝇量模式。

2、实现步骤

（1）创建抽象蝇量类

```
/**
 * 抽象蝇量类
 */
public abstract class Flyweight {

    /**
     * 共享状态（所有实例共有的、一致的状态）
     */
    public String sharedState;

    /**
     * 非共享状态（不同实例间不共有、或者不一致的状态）
     */
    public final String unsharedState;

    public Flyweight(String unsharedState) {
        this.unsharedState = unsharedState;
    }

    public abstract void operate();
}
```

（2）创建具体蝇量类

```
/**
```

```
 * 具体蝇量类
 */
public class ConcreteFlyweight extends Flyweight {

    public ConcreteFlyweight(String unsharedState) {
        super(unsharedState);
        sharedState = "Shared State";
    }

    @Override
    public void operate() {
        System.out.println("ConcreteFlyweight is operating. [sharedState: " + sharedState + ", un
haredState: " + unsharedState + "]");
    }
}
```

（3）创建蝇量类工厂

```
/**
 * 蝇量类工厂
 */
public class FlyweightFactory {

    /**
     * 池容器
     */
    private static HashMap<String, Flyweight> pool = new HashMap<>();

    /**
     * 获取蝇量类实例
     */
    public static Flyweight getFlyweight(String unsharedState) {
        // 从池中取出蝇量类实例
        Flyweight flyweight = pool.get(unsharedState);
        if (flyweight == null) {
            // 创建蝇量类实例,并放入池中
            System.out.println("Create flyweight instance, and put into the pool：" + unsharedSta
e);
            flyweight = new ConcreteFlyweight(unsharedState);
            pool.put(unsharedState, flyweight);
        } else {
            System.out.println("Get flyweight instance from the pool：" + unsharedState);
        }
        return flyweight;
    }
}
```

（4）使用蝇量类工厂创建蝇量类

```
public class Test {

    public static void main(String[] args) {
        // 从工厂获取蝇量类实例，并执行操作
```

```
        Flyweight flyweight1 = FlyweightFactory.getFlyweight("Unshared State A");
        flyweight1.operate();
        System.out.println();
        Flyweight flyweight2 = FlyweightFactory.getFlyweight("Unshared State B");
        flyweight2.operate();
        System.out.println();
        Flyweight flyweight3 = FlyweightFactory.getFlyweight("Unshared State A");
        flyweight3.operate();
    }
}
```

五、解释器模式

1、定义

解释器模式将每一个语法规则表示成一个类。

要点：

- 当你需要实现一个简单的语言时，就使用解释器。
- 解释器模式的每一个语法规则对应一个表达式类，表达式包含终结符表达式和非终结符表达式。
- 终结符表达式对应的语法规则不可再分解，因此终结符表达式的解释方法不会调用其他表达式的解方法。
- 非终结符表达式对应的语法规则可以分解为其他语法规则，因此非终结符表达式的解释方法会调用其他表达式的解释方法。

2、实现步骤

（1）创建上下文环境类

```
/**
 * 上下文环境（运行环境）
 * 用于管理全局信息
 */
public class Context {

    // TODO 处理全局信息的相关方法

    /**
     * 运行
     */
    public void run(String data) {
        // 调用相关表达式的解释方法
        Expression terminal1 = new TerminalExpression(data);
        Expression terminal2 = new TerminalExpression(data);
        Expression nonterminal = new NonterminalExpression(terminal1, terminal2);
        nonterminal.interpret(this);
    }
}
```

（2）创建表达式接口

```
/**
 * 表达式接口
 */
public interface Expression {

    /**
     * 执行解释
     */
    public void interpret(Context context);
}
```

（3）创建具体表达式

```
/**
 * 终结符表达式
 */
public class TerminalExpression implements Expression {

    private String data;

    public TerminalExpression(String data) {
        this.data = data;
    }

    @Override
    public void interpret(Context context) {
        System.out.println("TerminalExpression is interpreting data: " + data);
        // TODO 进行解释操作，终结符表达式不会调用其他表达式的解释方法
    }
}

/**
 * 非终结符表达式
 */
public class NonterminalExpression implements Expression {

    private Expression exp1;
    private Expression exp2;

    public NonterminalExpression(Expression exp1, Expression exp2) {
        this.exp1 = exp1;
        this.exp2 = exp2;
    }

    @Override
    public void interpret(Context context) {
        System.out.println("NonterminalExpression is interpreting...");
        // 调用其他表达式的解释方法
        exp1.interpret(context);
        exp2.interpret(context);
    }
```

```
}
```

（4）使用表达式解释数据

```
public class Test {

    public static void main(String[] args) {
        Context context = new Context();
        context.run("I like cat");
    }
}
```

3、举个栗子

创建一个解释“二元运算代码”的解释器。

代码格式：算术表达式; 变量赋值1; 变量赋值2。

代码例子：a + b; a = 1; b = 2。

（1）创建上下文环境类

```
/**
 * 上下文环境（运行环境）
 * 用于管理全局信息
 */
public class Context {

    /**
     * 数据池
     */
    private static Map<Expression, Integer> dataPool =  new HashMap<Expression, Integer>();

    /**
     * 赋值
     */
    public void assign(Expression var, int value) {
        dataPool.put(var, value);
    }

    /**
     * 取值
     */
    public int lookup(Expression var) {
        Integer value = dataPool.get(var);
        return value == null ? 0 : value;
    }

    /**
     * 运行代码
     */
    public int run(String code) {
```

```
        return new CodeExpression(code).interpret(this);
    }
}
```

（2）创建抽象表达式

```
/**
 * 抽象表达式
 */
public abstract class Expression {

    protected String code;

    public Expression(String code) {
        this.code = code;
    }

    /**
     * 执行解释
     */
    public abstract int interpret(Context context);

    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (this == obj) {
            return true;
        }
        if (obj instanceof Expression) {
            return this.code.equals(((Expression) obj).code);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return code.hashCode();
    }
}
```

（3）创建解释“二元运算代码”的具体表达式

```
/**
 * 代码表达式
 */
public class CodeExpression extends Expression {

    public CodeExpression(String code) {
        super(code);
    }
```

```
    @Override
    public int interpret(Context context) {
        // 代码格式: 算术表达式; 变量赋值1; 变量赋值2
        // 代码例子: a + b; a = 1; b = 2
        String[] codes = code.split("; ");
        // 算术表达式
        ArithExpression arith = new ArithExpression(codes[0]);
        // 赋值表达式
        AssignExpression assign = null;
        for (int i = 1; i < codes.length; i++) {
            assign = new AssignExpression(codes[i]);
            assign.interpret(context);
        }
        return arith.interpret(context);
    }
}

/**
 * 算术表达式
 */
public class ArithExpression extends Expression {

    public ArithExpression(String code) {
        super(code);
    }

    @Override
    public int interpret(Context context) {
        // a + b
        // 以"空格"分隔变量与运算符
        String[] codes = code.split(" ");
        // 变量表达式
        VarExpression var1 = new VarExpression(codes[0]);
        VarExpression var2 = new VarExpression(codes[2]);
        // 运算符表达式
        OperatorExpression operator = new OperatorExpression(var1, codes[1], var2);
        return operator.interpret(context);
    }
}

/**
 * 赋值表达式
 */
public class AssignExpression extends Expression {

    public AssignExpression(String code) {
        super(code);
    }

    @Override
    public int interpret(Context context) {
        // a = 1
        // 以"空格等号空格"分隔变量与数值
        String[] codes = code.split(" = ");
```

```
        // 变量表达式
        VarExpression var = new VarExpression(codes[0]);
        // 变量赋值
        context.assign(var, Integer.parseInt(codes[1]));
        return 0;
    }
}

/**
 * 变量表达式
 */
public class VarExpression extends Expression {

    public VarExpression(String code) {
        super(code);
    }

    @Override
    public int interpret(Context context) {
        return context.lookup(this);
    }
}

/**
 * 运算符表达式
 */
public class OperatorExpression extends Expression {

    Expression var1;
    Expression var2;

    public OperatorExpression(Expression var1, String code, Expression var2) {
        super(code);
        this.var1 = var1;
        this.var2 = var2;
    }

    @Override
    public int interpret(Context context) {
        OperatorExpression operator = null;
        switch (code) {
        case "+":
            operator = new AddExpression(var1, var2);
            break;
        case "-":
            operator = new SubExpression(var1, var2);
            break;
        default:
            throw new RuntimeException("暂不支持该运算");
        }
        return operator.interpret(context);
    }
}
```

```
/**
 * 加法表达式
 */
public class AddExpression extends OperatorExpression {

    public AddExpression(Expression var1, Expression var2) {
        super(var1, "+", var2);
    }

    @Override
    public int interpret(Context context) {
        return var1.interpret(context) + var2.interpret(context);
    }
}

/**
 * 减法表达式
 */
public class SubExpression extends OperatorExpression {

    public SubExpression(Expression var1, Expression var2) {
        super(var1, "-", var2);
    }

    @Override
    public int interpret(Context context) {
        return var1.interpret(context) - var2.interpret(context);
    }
}
```

（4）使用表达式解释“二元运算代码”

```
public class Test {

    public static void main(String[] args) {
        // 上下文环境
        Context context = new Context();
        // 运行代码
        int result = context.run("a + b; a = 1; b = 2");
        System.out.println("结果1： " + result);
        result = context.run("a - b; a = 7; b = 2");
        System.out.println("结果2： " + result);
    }
}
```

六、中介者模式

1、定义

中介者模式用于集中相关对象之间复杂的沟通和控制方式。

要点：

- 通过将对象彼此解耦，可以增加对象的复用性。
- 每个对象都会在自己状态改变时，告诉中介者。
- 每个对象都会对中介者所发出的请求做出回应。

2、实现步骤

（1）创建交互对象抽象类

```
/**
 * 交互对象抽象类
 */
public abstract class InteractiveObject {

    protected Mediator mediator;

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

    /**
     * 发送信息
     */
    public abstract void send(String msg);

    /**
     * 接收信息
     */
    public abstract void receive(String msg);
}
```

（2）创建具体交互对象

```
/**
 * 具体交互对象A
 */
public class ConcreteInteractiveObjectA extends InteractiveObject {

    public ConcreteInteractiveObjectA(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String msg) {
        System.out.println("ConcreteInteractiveObjectA has sended message: " + msg);
        mediator.forward(this, msg);
    }

    @Override
    public void receive(String msg) {
        System.out.println("ConcreteInteractiveObjectA has received message: " + msg);
    }
```

```
}

/**
 * 具体交互对象B
 */
public class ConcreteInteractiveObjectB extends InteractiveObject {

    public ConcreteInteractiveObjectB(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String msg) {
        System.out.println("ConcreteInteractiveObjectB has sended message: " + msg);
        mediator.forward(this, msg);
    }

    @Override
    public void receive(String msg) {
        System.out.println("ConcreteInteractiveObjectB has received message: " + msg);
    }
}

/**
 * 具体交互对象C
 */
public class ConcreteInteractiveObjectC extends InteractiveObject {

    public ConcreteInteractiveObjectC(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String msg) {
        System.out.println("ConcreteInteractiveObjectC has sended message: " + msg);
        mediator.forward(this, msg);
    }

    @Override
    public void receive(String msg) {
        System.out.println("ConcreteInteractiveObjectC has received message: " + msg);
    }
}
```

（3）创建中介者抽象类

```
/**
 * 中介者抽象类
 */
public abstract class Mediator {

    /**
     * 注册交互对象
     */
```

```
    public abstract void register(InteractiveObject obj);

    /**
     * 转发信息
     */
    public abstract void forward(InteractiveObject obj, String msg);
}
```

（4）创建具体中介者

```
/**
 * 具体中介者
 */
public class ConcreteMediator extends Mediator {

    /**
     * 交互对象集合
     */
    private List<InteractiveObject> interactiveObjs = new ArrayList<>();

    @Override
    public void register(InteractiveObject obj) {
        interactiveObjs.add(obj);
    }

    @Override
    public void forward(InteractiveObject obj, String msg) {
        for (InteractiveObject interactiveObj : interactiveObjs) {
            if (!interactiveObj.equals(obj)) {
                interactiveObj.receive(msg);
            }
        }
    }
}
```

（5）使用中介者管理交互对象之间的交互

```
public class Test {

    public static void main(String[] args) {
        // 中介者
        Mediator mediator = new ConcreteMediator();
        // 交互对象
        InteractiveObject objA = new ConcreteInteractiveObjectA(mediator);
        InteractiveObject objB = new ConcreteInteractiveObjectB(mediator);
        InteractiveObject objC = new ConcreteInteractiveObjectC(mediator);
        // 注册交互对象到中介者
        mediator.register(objA);
        mediator.register(objB);
        mediator.register(objC);
        // 发送信息
        objA.send("hello");
    }
```

```
}
```

七、备忘录模式

1、定义

备忘录模式通过将状态存储在对象外部，使得对象可以返回之前的状态。

2、实现步骤

（1）创建备忘录

```
/**
 * 备忘录
 */
public class Memento {

    private String state;

    public Memento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
}
```

（2）创建备忘录管理者

```
/**
 * 备忘录管理者
 */
public class MementoCaretaker {

    private Memento memento;

    public Memento getMemento() {
        return memento;
    }

    public void setMemento(Memento memento) {
        this.memento = memento;
    }
}
```

（3）创建备忘录发起人

```java
/**
 * 备忘录发起人
 */
public class MementoOriginator {

    private String state;

    public String getState() {
        return state;
    }

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

    /**
     * 创建备忘录
     */
    public Memento createMemento() {
        return new Memento(state);
    }

    /**
     * 从备忘录中恢复状态
     */
    public void restoreFromMemento(Memento memento) {
        this.setState(memento.getState());
    }
}
```

（4）使用备忘录存储、恢复状态

```java
public class Test {

    public static void main(String[] args) {
        // 备忘录管理者
        MementoCaretaker caretaker = new MementoCaretaker();
        // 备忘录发起人
        MementoOriginator originator = new MementoOriginator();
        originator.setState("状态1");
        System.out.println("初始状态：" + originator.getState());
        // 备忘录发起人创建备忘录
        Memento memento = originator.createMemento();
        // 备忘录管理者保存备忘录
        caretaker.setMemento(memento);
        // 备忘录发起人改变状态
        originator.setState("状态2");
        System.out.println("新状态：" + originator.getState());
        // 从备忘录管理者中取出备忘录，并通过备忘录恢复状态
        originator.restoreFromMemento(caretaker.getMemento());
        System.out.println("恢复状态：" + originator.getState());
    }
}
```

八、原型模式

1、定义

原型模式允许你通过复制现有的实例来创建新的实例。

要点：

- 在 Java 中，这通常意味着使用 clone() 方法，或者反序列化。

2、实现步骤

（1）创建原型类，并实现 Cloneable 接口

```
/**
 * 原型类（实现Cloneable接口）
 */
public class Prototype implements Cloneable {

    public String type;

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

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    /**
     * 实现clone方法
     */
    @Override
    public Prototype clone() {
        try {
            return (Prototype) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return null;
    }

    @Override
    public String toString() {
        return "Prototype [type=" + type + "]";
    }
}
```

（2）通过复制现有实例，来创建新的实例

```
public class Test {

    public static void main(String[] args) {
        Prototype prototype1 = new Prototype("A");
        System.out.println(prototype1);
        // 复制现有实例来创建新的实例
        Prototype prototype2 = prototype1.clone();
        System.out.println(prototype2);
    }
}
```

九、访问者模式

1、定义

访问者模式通过访问数据结构（比如组合结构）中的每个元素，来对元素进行各种操作。

要点：

- 通过将数据结构与数据操作分离，使得无需改变结构本身，就可以添加作用于结构内的元素的新的作。

2、实现步骤

（1）创建元素接口

元素接口中定义了接受访问者访问的方法。

```
/**
 * 元素接口
 */
public interface Element {

    /**
     * 接受访问者访问
     */
    public void accept(Visitor visitor);
}
```

（2）创建具体元素

```
/**
 * 具体元素A
 */
public class ConcreteElementA implements Element {

    @Override
    public void accept(Visitor visitor) {
        // 具体元素接受访问 -> 访问者访问具体元素
```

```
        visitor.visit(this);
    }

    public void operate() {
        System.out.println("  ConcreteElementA operate");
    }
}

/**
 * 具体元素B
 */
public class ConcreteElementB implements Element {

    @Override
    public void accept(Visitor visitor) {
        // 具体元素接受访问 -> 访问者访问具体元素
        visitor.visit(this);
    }

    public void operate1() {
        System.out.println("  ConcreteElementB operate1");
    }

    public void operate2() {
        System.out.println("  ConcreteElementB operate2");
    }
}
```

（3）创建数据结构

```
/**
 * 数据结构
 */
public class DataStructure {

    private List<Element> elements = new ArrayList<>();

    public void add(Element element) {
        elements.add(element);
    }

    public void remove(Element element) {
        elements.remove(element);
    }

    /**
     * 接受访问者访问
     */
    public void accept(Visitor visitor) {
        for (Element element : elements) {
            element.accept(visitor);
        }
    }
}
```

（4）创建访问者接口

```
/**
 * 访问者接口
 */
public interface Visitor {

    /**
     * 访问具体元素A
     */
    public void visit(ConcreteElementA element);

    /**
     * 访问具体元素B
     */
    public void visit(ConcreteElementB element);
}
```

（5）创建具体访问者

```
/**
 * 具体访问者
 */
public class ConcreteVisitor implements Visitor {

    @Override
    public void visit(ConcreteElementA element) {
        System.out.println("ConcreteVisitor visit ConcreteElementA:");
        // 访问者操作元素
        element.operate();
    }

    @Override
    public void visit(ConcreteElementB element) {
        System.out.println("ConcreteVisitor visit ConcreteElementB:");
        // 访问者操作元素
        element.operate1();
        element.operate2();
    }
}
```

（6）使用访问者操作数据结构中的元素

```
public class Test {

    public static void main(String[] args) {
        // 数据结构
        DataStructure dataStructure = new DataStructure();
        dataStructure.add(new ConcreteElementA());
        dataStructure.add(new ConcreteElementB());
        // 访问者
        Visitor visitor = new ConcreteVisitor();
        // 数据结构接受访问者访问
```

```
        dataStructure.accept(visitor);
    }
}
```