课程目标

1、掌握状态模式的应用场景。

2、了解状态机实现订单状态流转控制的过程

3、掌握状态模式和策略模式的区别。

4、掌握状态模式和责任链模式的区别。

内容定位

1、如果参与电商订单业务开发的人群，可以重点关注状态模式。

状态模式

状态模式（ State Pattern ）也称为状态机模式(State Machine Pattern)，是允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类，属于行为型模式。

```
原文：Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
解释：允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。
```

状态模式中类的行为是由状态决定的，不同的状态下有不同的行为。其意图是让一个对象在其内部改变的时候，其行为也随之改变。状态模式核心是状态与行为绑定，不同的状态对应不同的行为。

状态模式的应用场景

状态模式在生活场景中也还比较常见。例如：我们平时网购的订单状态变化。另外，我们平时坐电梯，电梯的状态变化。

订单状态变化

电梯状态的变化

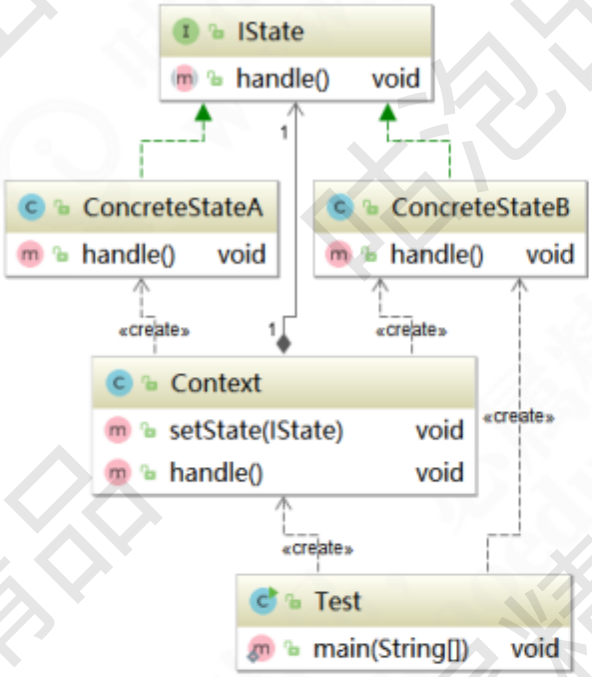
在软件开发过程中，对于某一项操作，可能存在不同的情况。通常处理多情况问题最直接的方式就是使用 if...else 或 switch...case 条件语句进行枚举。但是这种做法对于复杂状态的判断天然存在弊端：条件判断语句过于臃肿，可读性差，且不具备扩展性，维护难度也大。而如果转换思维，将这些不同状态独立起来用各个不同的类进行表示，系统处于哪种情况，直接使用相应的状态类对象进行处理，消除了 if...else，switch...case 等冗余语句，代码更有层次性且具备良好扩展力。

状态模式主要解决的就是当控制一个对象状态的条件表达式过于复杂时的情况。通过把状态的判断逻辑转移到表示不同状态的一系列类中，可以把复杂的判断逻辑简化。对象的行为依赖于它的状态（属性），并且会根据它的状态改变而改变它的相关行为。状态模式适用于以下场景：

1、行为随状态改变而改变的场景；

2、一个操作中含有庞大的多分支结构，并且这些分支取决于对象的状态。

首先来看下状态模式的通用 UML 类图：



从 UML 类图中，我们可以看到，状态模式主要包含三种角色：

1、环境类角色（Context）：定义客户端需要的接口，内部维护一个当前状态实例，并负责具体状态的切换；

2、抽象状态角色（State）：定义该状态下的行为，可以有一个或多个行为；

3、具体状态角色（ConcreteState）：具体实现该状态对应的行为，并且在需要的情况下进行状态切换。

状态模式在业务场景中的应用

我们在 GPer 社区阅读文章时，如果觉得文章写的很好，我们就会评论、收藏两连发。如果处于登录情况下，我们就可以直接做评论，收藏这些行为。否则，跳转到登录界面，登录后再继续执行先前的动作。这里涉及的状态有两种：登录与未登录，行为有两种：评论，收藏。下面我们使状态模式来实现一下这个逻辑，代码如下。

首先创建抽象状态角色 UserState 类：

```
public abstract class UserState {
    protected AppContext context;

    public void setContext(AppContext context) {
        this.context = context;
```

```
    }

    public abstract void favorite();

    public abstract void comment(String comment);
}
```

然后，创建登录状态 LogInState 类：

```
public class LoginInState extends UserState {
    @Override
    public void favorite() {
        System.out.println("收藏成功！");
    }

    @Override
    public void comment(String comment) {
        System.out.println(comment);
    }
}
```

创建未登录状态 UnloginState 类：

```
public class UnLoginState extends UserState {
    @Override
    public void favorite() {
        this.switch2Login();
        this.context.getState().favorite();
    }

    @Override
    public void comment(String comment) {
        this.switch2Login();
        this.context.getState().comment(comment);
    }

    private void switch2Login() {
        System.out.println("跳转到登录页面！");
        this.context.setState(this.context.STATE_LOGIN);
    }
}
```

创建上下文角色 AppContext 类：

```
public class AppContext {
    public static final UserState STATE_LOGIN = new LoginInState();
    public static final UserState STATE_UNLOGIN = new UnLoginState();
    private UserState currentState = STATE_UNLOGIN;
    {
        STATE_LOGIN.setContext(this);
        STATE_UNLOGIN.setContext(this);
    }

    public void setState(UserState state) {
        this.currentState = state;
        this.currentState.setContext(this);
    }

    public UserState getState() {
        return this.currentState;
    }

    public void favorite() {
```

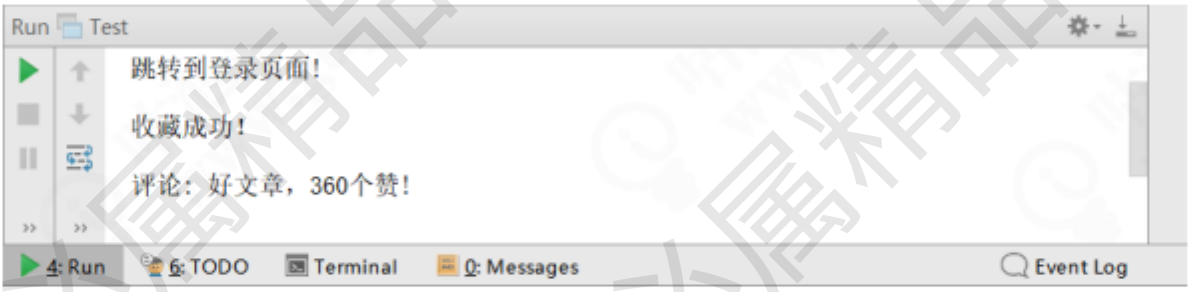
```
            this.currentState.favorite();
        }

        public void comment(String comment) {
            this.currentState.comment(comment);
        }
    }
```

编写客户端测试代码：

```
    public static void main(String[] args) {
        AppContext context = new AppContext();
        context.favorite();
        context.comment("评论：好文章，360个赞!");
    }
```

运行结果如下：



利用状态机实现订单状态流转控制

状态机是状态模式的一种应用，相当于上下文角色的一个升级版。在工作流或游戏等各种系统中有大量使用，如各种工作流引擎，它几乎是状态机的子集和实现，封装状态的变化规则。Spring 也提供给了我们一个很好的解决方案。Spring 中的组件名称就叫 StateMachine(状态机)。状态机帮助开发者简化状态控制的开发过程，让状态机结构更加层次化。下面，我们用 Spring 状态机模拟一个订单状态流转的过程。

1、添加依赖

```
<dependency>
    <groupId>org.springframework.statemachine</groupId>
    <artifactId>spring-statemachine-core</artifactId>
    <version>2.0.1.RELEASE</version>
</dependency>
```

5、创建订单实体 Order 类

```
public class Order {
    private int id;
    private OrderStatus status;
    public void setStatus(OrderStatus status) {
        this.status = status;
    }

    public OrderStatus getStatus() {
```

```
        return status;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    @Override
    public String toString() {
        return "订单号：" + id + ", 订单状态：" + status;
    }
}
```

2、创建订单状态枚举类和状态转换枚举类

```
/**
 * 订单状态
 */
public enum OrderStatus {
    // 待支付，待发货，待收货，订单结束
    WAIT_PAYMENT, WAIT_DELIVER, WAIT_RECEIVE, FINISH;
}
```

```
/**
 * 订单状态改变事件
 */
public enum OrderStatusChangeEvent {
    // 支付，发货，确认收货
    PAYED, DELIVERY, RECEIVED;
}
```

3、添加状态流转配置

```
/**
 * 订单状态机配置
 */
@Configuration
@EnableStateMachine(name = "orderStateMachine")
public class OrderStateMachineConfig extends StateMachineConfigurerAdapter<OrderStatus, OrderStatusChangeEvent> {

    /**
     * 配置状态
     * @param states
     * @throws Exception
     */
    public void configure(StateMachineStateConfigurer<OrderStatus, OrderStatusChangeEvent> states) throws Exception {
        states
                .withStates()
                .initial(OrderStatus.WAIT_PAYMENT)
                .states(EnumSet.allOf(OrderStatus.class));
    }

    /**
     * 配置状态转换事件关系
     * @param transitions
     * @throws Exception
     */
```

```
    public void configure(StateMachineTransitionConfigurer<OrderStatus, OrderStatusChangeEvent> transitions) throws
Exception {
        transitions
                .withExternal().source(OrderStatus.WAIT_PAYMENT).target(OrderStatus.WAIT_DELIVER)
                .event(OrderStatusChangeEvent.PAYED)
                .and()
                .withExternal().source(OrderStatus.WAIT_DELIVER).target(OrderStatus.WAIT_RECEIVE)
                .event(OrderStatusChangeEvent.DELIVERY)
                .and()
                .withExternal().source(OrderStatus.WAIT_RECEIVE).target(OrderStatus.FINISH)
                .event(OrderStatusChangeEvent.RECEIVED);
    }

    /**
     * 持久化配置
     * 实际使用中，可以配合 redis 等，进行持久化操作
     * @return
     */
    @Bean
    public DefaultStateMachinePersister persister(){
        return new DefaultStateMachinePersister<>(new StateMachinePersist<Object, Object, Order>() {
            @Override
            public void write(StateMachineContext<Object, Object> context, Order order) throws Exception {
                //此处并没有进行持久化操作
            }

            @Override
            public StateMachineContext<Object, Object> read(Order order) throws Exception {
                //此处直接获取 order 中的状态，其实并没有进行持久化读取操作
                return new DefaultStateMachineContext(order.getStatus(), null, null, null);
            }
        });
    }
}
```

4、添加订单状态监听器

```
@Component("orderStateListener")
@WithStateMachine(name = "orderStateMachine")
public class OrderStateListenerImpl{

    @OnTransition(source = "WAIT_PAYMENT", target = "WAIT_DELIVER")
    public boolean payTransition(Message<OrderStatusChangeEvent> message) {
        Order order = (Order) message.getHeaders().get("order");
        order.setStatus(OrderStatus.WAIT_DELIVER);
        System.out.println("支付，状态机反馈信息：" + message.getHeaders().toString());
        return true;
    }

    @OnTransition(source = "WAIT_DELIVER", target = "WAIT_RECEIVE")
    public boolean deliverTransition(Message<OrderStatusChangeEvent> message) {
        Order order = (Order) message.getHeaders().get("order");
        order.setStatus(OrderStatus.WAIT_RECEIVE);
        System.out.println("发货，状态机反馈信息：" + message.getHeaders().toString());
        return true;
    }

    @OnTransition(source = "WAIT_RECEIVE", target = "FINISH")
    public boolean receiveTransition(Message<OrderStatusChangeEvent> message){
        Order order = (Order) message.getHeaders().get("order");
        order.setStatus(OrderStatus.FINISH);
        System.out.println("收货，状态机反馈信息：" + message.getHeaders().toString());
        return true;
```

```
    }
}
```

5、创建 IOrderService 接口

```
public interface IOrderService {
    //创建新订单
    Order create();
    //发起支付
    Order pay(int id);
    //订单发货
    Order deliver(int id);
    //订单收货
    Order receive(int id);
    //获取所有订单信息
    Map<Integer, Order> getOrders();
}
```

6、在 Service 业务逻辑中应用

```
@Service("orderService")
public class OrderServiceImpl implements IOrderService {

    @Autowired
    private StateMachine<OrderStatus, OrderStatusChangeEvent> orderStateMachine;

    @Autowired
    private StateMachinePersister<OrderStatus, OrderStatusChangeEvent, Order> persister;

    private int id = 1;
    private Map<Integer, Order> orders = new HashMap<>();

    public Order create() {
        Order order = new Order();
        order.setStatus(OrderStatus.WAIT_PAYMENT);
        order.setId(id++);
        orders.put(order.getId(), order);
        return order;
    }

    public Order pay(int id) {
        Order order = orders.get(id);
        System.out.println("线程名称：" + Thread.currentThread().getName() + " 尝试支付，订单号：" + id);
        Message message = MessageBuilder.withPayload(OrderStatusChangeEvent.PAYED).setHeader("order", order).build();
        if (!sendEvent(message, order)) {
            System.out.println("线程名称：" + Thread.currentThread().getName() + " 支付失败，状态异常，订单号：" + id);
        }
        return orders.get(id);
    }

    public Order deliver(int id) {
        Order order = orders.get(id);
        System.out.println("线程名称：" + Thread.currentThread().getName() + " 尝试发货，订单号：" + id);
        if (!sendEvent(MessageBuilder.withPayload(OrderStatusChangeEvent.DELIVERY).setHeader("order", order).build(),
orders.get(id))) {
            System.out.println("线程名称：" + Thread.currentThread().getName() + " 发货失败，状态异常，订单号：" + id);
        }
        return orders.get(id);
    }

    public Order receive(int id) {
        Order order = orders.get(id);
        System.out.println("线程名称：" + Thread.currentThread().getName() + " 尝试收货，订单号：" + id);
```

```
        if (!sendEvent(MessageBuilder.withPayload(OrderStatusChangeEvent.RECEIVED).setHeader("order", order).build(),
orders.get(id))) {
            System.out.println("线程名称：" + Thread.currentThread().getName() + " 收货失败，状态异常，订单号：" + id);
        }
        return orders.get(id);
    }


    public Map<Integer, Order> getOrders() {
        return orders;
    }


    /**
     * 发送订单状态转换事件
     *
     * @param message
     * @param order
     * @return
     */
    private synchronized boolean sendEvent(Message<OrderStatusChangeEvent> message, Order order) {
        boolean result = false;
        try {
            orderStateMachine.start();
            //尝试恢复状态机状态
            persister.restore(orderStateMachine, order);
            //添加延迟用于线程安全测试
            Thread.sleep(1000);
            result = orderStateMachine.sendEvent(message);
            //持久化状态机状态
            persister.persist(orderStateMachine, order);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            orderStateMachine.stop();
        }
        return result;
    }
}
```

7、编写客户端测试代码

```
@SpringBootApplication
public class Test {
    public static void main(String[] args) {

        Thread.currentThread().setName("主线程");

        ConfigurableApplicationContext context = SpringApplication.run(Test.class,args);

        IOrderService orderService = (IOrderService)context.getBean("orderService");

        orderService.create();
        orderService.create();

        orderService.pay(1);

        new Thread("客户线程"){
            @Override
            public void run() {
                orderService.deliver(1);
                orderService.receive(1);
            }
```

```
        }.start();

        orderService.pay(2);
        orderService.deliver(2);
        orderService.receive(2);

        System.out.println("全部订单状态：" + orderService.getOrders());

    }
}
```

相信小伙伴们，通过这个真实的业务案例，对状态模式已经有了一个非常深刻的理解。

状态模式在源码中的体现

状态模式的具体应用在源码中非常少见，在源码中一般只是提供一种通用的解决方案。如果一定要找，当然也是能找到的。经历千辛万苦，持续烧脑，下面我们来看一个在 JSF 源码中的 Lifecycle 类。JSF 也算是一个比较经典的前端框架，那么没用过的小伙伴也没关系，我们这是只是分析一下其设计思想。在 JSF 中它所有页面的处理分为 6 个阶段，被定义在了 PhaseId 类中，用不同的常量来表示生命周期阶段，源码如下：

```
public class PhaseId implements Comparable {
   ...
    private static final PhaseId[] values =
    {
        ANY_PHASE, //任意一个生命周期阶段
        RESTORE_VIEW, //恢复视图阶段
        APPLY_REQUEST_VALUES,//应用请求值阶段
        PROCESS_VALIDATIONS, //处理输入校验阶段
        UPDATE_MODEL_VALUES, //更新模型的值阶段
        INVOKE_APPLICATION, //调用应用阶段
        RENDER_RESPONSE //显示响应阶段
    };
    ...
}
```

那么这些状态的切换都在 Lifecycle 的 execute()方法中进行。其中会传一个参数 FacesContext 对象，最终所有的状态都被 FacesContext 保存。在此呢，我们就不做继续深入的分析。

状态模式相关的设计模式

1、状态模式与责任链模式

状态模式和责任链模式都能消除 if 分支过多的问题。但某些情况下，状态模式中的状态可以理解为责任，那么这种情况下，两种模式都可以使用。

从定义来看，状态模式强调的是一个对象内在状态的改变，而责任链模式强调的是外部节点对象间的改变。

从其代码实现上来看，他们间最大的区别就是状态模式各个状态对象知道自己下一个要进入的状态对象；而责任链模式并不清楚其下一个节点处理对象，因为链式组装由客户端负责。

2、状态模式与策略模式

状态模式和策略模式的 UML 类图架构几乎完全一样，但他们的应用场景是不一样的。策略模式多种算法行为择其一都能满足，彼此之间是独立的，用户可自行更换策略算法；而状态模式各个状态间是存在相互关系的，彼此之间在一定条件下存在自动切换状态效果，且用户无法指定状态，只能设置初始状态。

状态模式的优缺点

优点：

1、结构清晰 :将状态独立为类 ,消除了冗余的 if...else 或 switch...case 语句 ,使代码更加简洁，提高系统可维护性；

2、将状态转换显示化：通常的对象内部都是使用数值类型来定义状态，状态的切换是通过赋值进行表现，不够直观；而使用状态类，在切换状态时，是以不同的类进行表示，转换目的更加明确；

3、状态类职责明确且具备扩展性。

缺点：

1、类膨胀：如果一个事物具备很多状态，则会造成状态类太多；

2、状态模式的结构与实现都较为复杂，如果使用不当将导致程序结构和代码的混乱；

3、状态模式对开闭原则的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源代码，否则无法切换到新增状态，而且修改某个状态类的行为也需

修改对应类的源代码。