Sping StateMachine

有限状态机（英语：finite-state machine，缩写：FSM），简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。应用FSM模型可以帮助对象生命周期的状态的顺序以及导致状态变化的事件进行管理。将状态和事件控制从不同的业务Service方法的if else中抽离出来。FSM的应用范围很广，对于有复杂状态流，扩展性要求比较高的场景都可以使用该模型。下面是状态机模型中的4个要素，即现态、条件、动作、次态：

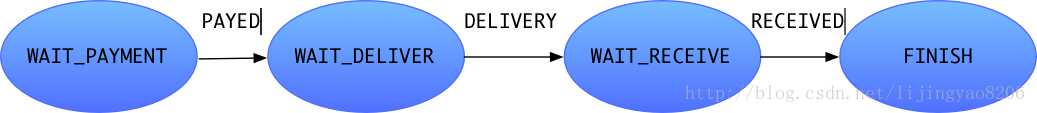
* 现态，当前所处的状态
* 条件，成为事件，当一个条件被满足，将会触发一个动作或者执行一次状态的迁移
* 动作，条件满足时执行的动作，执行完毕后可以迁移到新的状态，也可以保持原状态。动作不是必须的，当条件满足时也可以不执行任何动作，直接迁移到新状态
* 次态，条件满足后要迁往的新状态，次态是相对于现态而言的。次态一旦被激活，就转变成新的现态了。

下图是有限的状态集：opened及closed，如果现态是opened，当条件为close时，执行的动作是close door，次态则是closed，状态机逻辑执行完毕后cloed则变成现态：

|  |  |
| --- | --- |
| 68747470733a2f2f75706c6f61642e77696b696d656469612e6f72672f77696b6970656469612f636f6d6d6f6e732f7468756d622f632f63662f46696e6974655f73746174655f6d616368696e655f6578616d706c655f776974685f636f6d6d656e74732e7376672f34353070 | 68747470733a2f2f75706c6f61642e77696b696d656469612e6f72672f77696b6970656469612f636f6d6d6f6e732f7468756d622f362f36342f46696e6974655f53746174655f4d616368696e655f4c6f6769632e7376672f34303070782d46696e6974655f53746174655f4d |

FSM的执行逻辑可以理解为右图，即FSM的下一个状态和输出是由输入和当前状态决定的。

Spring StateMachine让状态更加层次化，可以帮助开发者简化状态机的开发过程，下面介绍一个订单流程的FSM实现，其订单流程如下图：



Spring State Machine的Maven依赖配置如下：

*<dependency>*

*<groupId>org.springframework.statemachine</groupId>*

*<artifactId>spring-statemachine-core</artifactId>*

*<version>2.0.2.RELEASE</version>*

*</dependency>*

1. 定义订单状态模型

* OrderStatus，状态枚举类

*public enum OrderStatus {*

*// 待支付，待发货，待收货，订单结束*

*WAIT\_PAYMENT, WAIT\_DELIVER, WAIT\_RECEIVE, FINISH;*

*}*

* OrderStatusChangeEvent，事件枚举类

*public enum OrderStatusChangeEvent {*

*// 支付，发货，确认收货*

*PAYED, DELIVERY, RECEIVED*

*}*

* 订单Entity Order

*@Entity*

*@Table(name = "order\_test")*

*@Getter*

*@Setter*

*@ToString*

*@NoArgsConstructor*

*@AllArgsConstructor*

*public class Order {*

*@Id*

*@GeneratedValue(strategy = GenerationType.AUTO)*

*@Column(name = "id")*

*private Integer id;*

*@NotNull*

*@Column(name = "order\_id")*

*private Integer orderId;*

*@NotNull*

*@Enumerated(EnumType.ORDINAL)*

*@Column(name = "status")*

*private OrderStatus status;*

*}*

定义Repository类OrderRepo

*public interface OrderRepo extends JpaRepository<Order, Integer> {*

*Order findByOrderId(Integer order);*

*}*

1. 初始化订单的状态集合以及状态转移事件

在启动Spring Boot时，需要注入状态机的状态及事件配置，主要涉及两个类：

* StateMachineStateConfigurer<S,E>，配置状态集合以及初始状态，泛型参数S代表状态，E代表事件
* StateMachineTransitionConfigurer<S,E>，配置状态流的转移，可以定义状态转换接收的事件

*@Configuration*

*@EnableStateMachine*

*static class StateMachineConfig*

*extends StateMachineConfigurerAdapter<OrderStatus,OrderStatusChangeEvent> {*

*@Override*

*public void configure(StateMachineStateConfigurer<OrderStatus,*

*OrderStatusChangeEvent> states) hrows Exception {*

*states.withStates()*

*.initial(OrderStatus.WAIT\_PAYMENT)*

*.states(EnumSet.allOf(OrderStatus.class));*

*}*

*@Override //描述状态转移及对应事件*

*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) .and().withExternal()*

*.source(OrderStatus.WAIT\_RECEIVE).target(OrderStatus.CLOSED)*

*.event(OrderStatusChangeEvent.REFUND);*

*}*

*}*

1. 状态转移监听器，状态转移过程中通过监听器（Listener）来处理一些持久化或者业务监控的任务，在需要持久化的场景中，可以在状态机模式中的监听器添加持久化处理，其中涉及到：

* StateMachineListener，事件监听器，通过Spring的Event机制实现，监听stateEntered, 离开状态，事件无法响应，transition等，借助listener跟踪状态转移
* StateChangeInterceptor，拦截器接口，其可以根据状态转移链的变化，主要在preEvent

,preStateChanged,postStateChange,preTransition，postTransition等执行点生效

* StateMachine状态机示例

3.1）监听器的Handler及接口定义PersistStateMachineHandler

*public class PersistStateMachineHandler extends LifecycleObjectSupport {*

*private final StateMachine<OrderStatus, OrderStatusChangeEvent> stateMachine;*

*private final PersistingStateChangeInterceptor interceptor =*

*new PersistingStateChangeInterceptor();*

*private final CompositePersistStateChangeListener listeners =*

*new CompositePersistStateChangeListener();*

*/\*\**

*\* 实例化一个新的持久化状态机Handler*

*\**

*\* @param stateMachine 状态机实例*

*\*/*

*public PersistStateMachineHandler(StateMachine<OrderStatus,*

*OrderStatusChangeEvent> stateMachine) {*

*Assert.notNull(stateMachine, "State machine must be set");*

*this.stateMachine = stateMachine;*

*}*

*@Override*

*protected void onInit() throws Exception {*

*stateMachine.getStateMachineAccessor().doWithAllRegions(function -> function.addStateMachineInterceptor(interceptor));*

*}*

*/\*\**

*\* 处理entity的事件*

*\**

*\* @param event*

*\* @param state*

*\* @return 如果事件被接受处理，返回true*

*\*/*

*public boolean handleEventWithState(Message<OrderStatusChangeEvent> event, OrderStatus state) {*

*stateMachine.stop();*

*List<StateMachineAccess<OrderStatus, OrderStatusChangeEvent>> withAllRegions = stateMachine.getStateMachineAccessor()*

*.withAllRegions();*

*for (StateMachineAccess<OrderStatus, OrderStatusChangeEvent> a : withAllRegions) {*

*a.resetStateMachine(new DefaultStateMachineContext<>(state, null, null, null));*

*}*

*stateMachine.start();*

*return stateMachine.sendEvent(event);*

*}*

*/\*\**

*\* 添加listener*

*\**

*\* @param listener the listener*

*\*/*

*public void addPersistStateChangeListener(PersistStateChangeListener listener) {*

*listeners.register(listener);*

*}*

*/\*\**

*\* 可以通过 addPersistStateChangeListener，增加当前Handler的PersistStateChangeListener。*

*\* 在状态变化的持久化触发时，会调用相应的实现了PersistStateChangeListener的Listener实例。*

*\*/*

*public interface PersistStateChangeListener {*

*/\*\**

*\* 当状态被持久化，调用此方法*

*\**

*\* @param state*

*\* @param message*

*\* @param transition*

*\* @param stateMachine 状态机实例*

*\*/*

*void onPersist(State<OrderStatus, OrderStatusChangeEvent> state, Message<OrderStatusChangeEvent> message, Transition<OrderStatus,*

*OrderStatusChangeEvent> transition,*

*StateMachine<OrderStatus, OrderStatusChangeEvent> stateMachine);*

*}*

*private class PersistingStateChangeInterceptor extends StateMachineInterceptorAdapter<OrderStatus, OrderStatusChangeEvent> {*

*// 状态预处理的拦截器方法*

*@Override*

*public void preStateChange(State<OrderStatus, OrderStatusChangeEvent> state, Message<OrderStatusChangeEvent> message,*

*Transition<OrderStatus, OrderStatusChangeEvent> transition, StateMachine<OrderStatus,*

*OrderStatusChangeEvent> stateMachine) {*

*listeners.onPersist(state, message, transition, stateMachine);*

*}*

*}*

*private class CompositePersistStateChangeListener extends AbstractCompositeListener<PersistStateChangeListener> implements*

*PersistStateChangeListener {*

*@Override*

*public void onPersist(State<OrderStatus, OrderStatusChangeEvent> state, Message<OrderStatusChangeEvent> message,*

*Transition<OrderStatus, OrderStatusChangeEvent> transition, StateMachine<OrderStatus,*

*OrderStatusChangeEvent> stateMachine) {*

*for (Iterator<PersistStateChangeListener> iterator = getListeners().reverse(); iterator.hasNext(); ) {*

*PersistStateChangeListener listener = iterator.next();*

*listener.onPersist(state, message, transition, stateMachine);*

*}*

*}*

*}*

*}*

3.2) 持久化状态变化的订单实体OrderPersistStateChangeListener

*public class OrderPersistStateChangeListener implements PersistStateMachineHandler.PersistStateChangeListener {*

*@Autowired*

*private OrderRepo repo;*

*@Override*

*public void onPersist(State<OrderStatus, OrderStatusChangeEvent> state, Message<OrderStatusChangeEvent> message,*

*Transition<OrderStatus, OrderStatusChangeEvent> transition, StateMachine<OrderStatus, OrderStatusChangeEvent> stateMachine) {*

*if (message != null && message.getHeaders().containsKey("order")) {*

*Integer order = message.getHeaders().get("order", Integer.class);*

*Order o = repo.findByOrderId(order);*

*OrderStatus status = state.getId();*

*o.setStatus(status);*

*repo.save(o);*

*}*

*}*

*}*

1. 测试，Controller及Service，查看订单状态及改变订单的状态

*@RestController*

*@RequestMapping("/orders")*

*public class OrderController {*

*@Autowired*

*private OrderStateService orderStateService*

*/\*\**

*\* 列出所有的订单列表*

*\**

*\* @return*

*\*/*

*@RequestMapping(method = {RequestMethod.GET})*

*public ResponseEntity orders() {*

*String orders = orderStateService.listDbEntries();*

*return new ResponseEntity(orders, HttpStatus.OK);*

*}*

*/\*\**

*\* 通过触发一个事件，改变一个订单的状态*

*\* @param orderId*

*\* @param event*

*\* @return*

*\*/*

*@RequestMapping(value = "/{orderId}", method = {RequestMethod.POST})*

*public ResponseEntity processOrderState(@PathVariable("orderId") Integer orderId, @RequestParam("event") OrderStatusChangeEvent event) {*

*Boolean result = orderStateService.change(orderId, event);*

*return new ResponseEntity(result, HttpStatus.OK);*

*}*

*}*

订单服务类：OrderStateService

*@Component*

*public class OrderStateService {*

*private PersistStateMachineHandler handler;*

*public OrderStateService(PersistStateMachineHandler handler) {*

*this.handler = handler;*

*}*

*@Autowired*

*private OrderRepo repo;*

*public String listDbEntries() {*

*List<Order> orders = repo.findAll();*

*StringJoiner sj = new StringJoiner(",");*

*for (Order order : orders) {*

*sj.add(order.toString());*

*}*

*return sj.toString();*

*}*

*public boolean change(int order, OrderStatusChangeEvent event) {*

*Order o = repo.findByOrderId(order);*

*return handler.handleEventWithState(MessageBuilder.withPayload(event).setHeader("order", order).build(), o.getStatus());*

*}*

*}*

通过命令写入4个Order订单，变更状态的请求如下：

*# curl -X POST "http://192.168.239.152:8089/orders/1?event=PAYED"*

*# curl -X POST "http://192.168.239.152:8089/orders/2?event=PAYED"*

*# curl -X POST "http://192.168.239.152:8089/orders/2?event=DELIVERY"*

https://github.com/lijingyao/state-machine

https://blog.csdn.net/lijingyao8206/article/details/78639552

http://projects.spring.io/spring-statemachine/