# 第十章 撮合系统的实现

# 撮合系统的简介

撮合引擎是所有撮合交易系统的核心组件，不管是股票交易系统——包括现货交易、期货交易、期权交易等，还是数字货币交易系统——包括币币交易、合约交易、杠杆交易等，以及各种不同的贵金属交易系统、大宗商品交易系统等，虽然各种不同交易系统的交易标的不同，但只要都是采用撮合交易模式，都离不开撮合引擎。



撮合引擎是可以具有通用性的，一套具有通用性的撮合引擎实现理论上可以应用到任何撮合交易系统中，而无需做任何代码上的调整。即是说，同一套撮合引擎实现，既可以应用在股票交易系统，也可以应用在数字货币交易系统，可以用于现货交易，也可以用于合约交易等。

那么，一套具有通用性的撮合引擎应该具备哪些功能呢？确定该问题的答案之前，我们先简单梳理一下一个完整的交易流程是怎样的？一般会包括以下步骤：

1.系统开放某个交易标的的交易功能。

2.用户提交该交易标的的买卖申报，即委托单。

3.系统验证委托单是否有效，包括交易标的是否处于可交易的状态、订单的价格和数量是否符合要求等。

4.确定该委托单的挂单(Maker)费率和吃单(Taker)费率。

5.检查用户的资产账户情况，包括账户状态是否交易受限，是否有足够资金用于下单等。

6.将详细的委托单数据持久化到数据库，并冻结用户账户中相应数量的资金。

7.将委托单进行撮合处理，即在交易委托账本(OrderBook)中寻找能与该委托单匹配成交的订单，匹配的结果可能是：全部成交、部分成交或无匹配。全部成交或部分成交时，可能在交易委托账本中存在一个或多个匹配的订单，即会产生一条或多条成交记录。当无匹配或部分成交时，委托单的部分数据包括剩余未成交的数量会暂时保存到交易委托账本中，等待与后续的委托单匹配撮合。

8.将撮合产生的成交记录持久化到数据库，并根据历史成交记录生成市场数据，如K线数据、今日涨跌幅等。

9.更新数据库中所有成交订单的委托单数据，以及更新订单用户的资产账户余额。

10.将更新的订单数据、市场数据等发送给到前台。

整个交易流程中涉及到多个服务，包括用户服务、账户服务、订单服务、撮合服务、市场数据服务等。其中，只有第7步是撮合引擎处理的。从单一职责原则来说，撮合引擎就应该只做一件事，那就是负责撮合订单。撮合之前的委托单持久化、冻结资金等，以及撮合之后生成K线数据等，都不应该属于撮合引擎的职责。

# 撮合竞价的方式

撮合竞价方式一般有两种，一是集合竞价，二是连续竞价。股票交易系统一般会在不同交易时间段采用不同的竞价方式，比如在开盘或收盘时采用集合竞价，从而产生开盘价或收盘价，其余时间采用连续竞价。而大多数字货币交易系统则没有集合竞价，只有连续竞价，开盘价一般是在开始交易之前就设定好的。

## 2.1 集合竞价

常见在开盘或收盘时采用集合竞价。

所谓集合竞价，是指对一段时间内接收的买卖委托单一次性集中撮合的竞价方式。以深沪的股票交易系统为例，在每个交易日的 9:15~9:25 期间是集合竞价时间。在该时间段内，系统陆续接收到的委托单不会即时成交，而是先将所有委托单按照价格优先、时间优先的原则排序，并在此基础上，找出一个基准价格，使它能同时满足以下三个条件：

* 可实现**最大成交量**的价格；
* 高于该价格的买单与低于该价格的卖单能全部成交的价格；
* 与该价格相同的买方或卖方至少有一方全部成交的价格。

在 9:25 分结束的时候，该基准价格就被确定为成交价格，所有高于该价格的买单与低于该价格的卖单都将以该价格成交。未能成交的委托单，则自动转入连续竞价。

不过，如果满足以上三个条件的价格存在两个或两个以上呢？对此，深交所和上交所的处理方案有所不同，深交所会取距前**收盘价最近的价格为成交价**，而上交所则取使未成交量**最小的价格为成交价**，如果未成交量最小的价格仍不止一个，则取中间价为成交价。

我们可以看出，集合竞价的主要目的就是为了**确定开盘价或收盘价**。

## 2.2 限价竞价

所谓连续竞价，也是我们所熟悉的竞价方式，是指对买卖委托单逐笔连续撮合的竞价方式。用户的挂单，只要满足成交条件，就能即时成交。而集合竞价，则要等到最后一刻才会成交。

连续竞价时，依然要满足价格优先、时间优先的成交原则：

1. 价格优先：买单则价格较高者能优先成交，卖单则是价格较低者能优先成交。
2. 时间优先：买卖方向和价格相同的委托单，先申报的委托单会比后申报的委托单优先成交。

另外，买入价必须大于或等于卖出价才能撮合成交。当买入价等于卖出价时，成交价就是买入价或卖出价。当买入价大于卖出价时，则还要参考前一笔成交价来确定最新成交价。假设买入价为 B，卖出价为 S，前一笔成交价为 P，最新成交价为 N，那么：

* 如果 P >= B，则 N = B
* 如果 P <= S，则 N = S
* 如果 B > P > S，则 N = P

一套通用的撮合引擎应该两种竞价方式都支持，但对于同一交易标的来说，两种竞价方式不能同时进行，因此设计上需要考虑如何在两种竞价方式之间切换，具体的实现思路在后续章节我们再展开来讲。

# 质量要求

我们的撮合引擎除了要满足以上所说的功能需求，还应该满足一些质量需求，尤其对可用性、可伸缩性和性能的要求较高。另外，为了达到通用，也要满足可复用性的需求。

先说下可复用性，我们期望的是该撮合引擎既能用于股票交易系统，也能用于数字货币交易系统，既能用于币币交易，也能用于合约交易。因此，该撮合引擎要避免引入与具体系统强相关的业务逻辑，以加强它的可复用性。

再看看性能，要衡量一个撮合引擎的性能，就看它处理每个交易对的 TPS 有多高，即每秒钟能处理多少笔相同交易对的委托单。以前，基于数据库的撮合技术，TPS 一般只有10笔/秒。而现在基本都是采用内存撮合技术，TPS 很容易就能达到1000笔/秒，如果使用独占的高性能服务器，1万笔/秒甚至更高的 TPS 都不难达到。

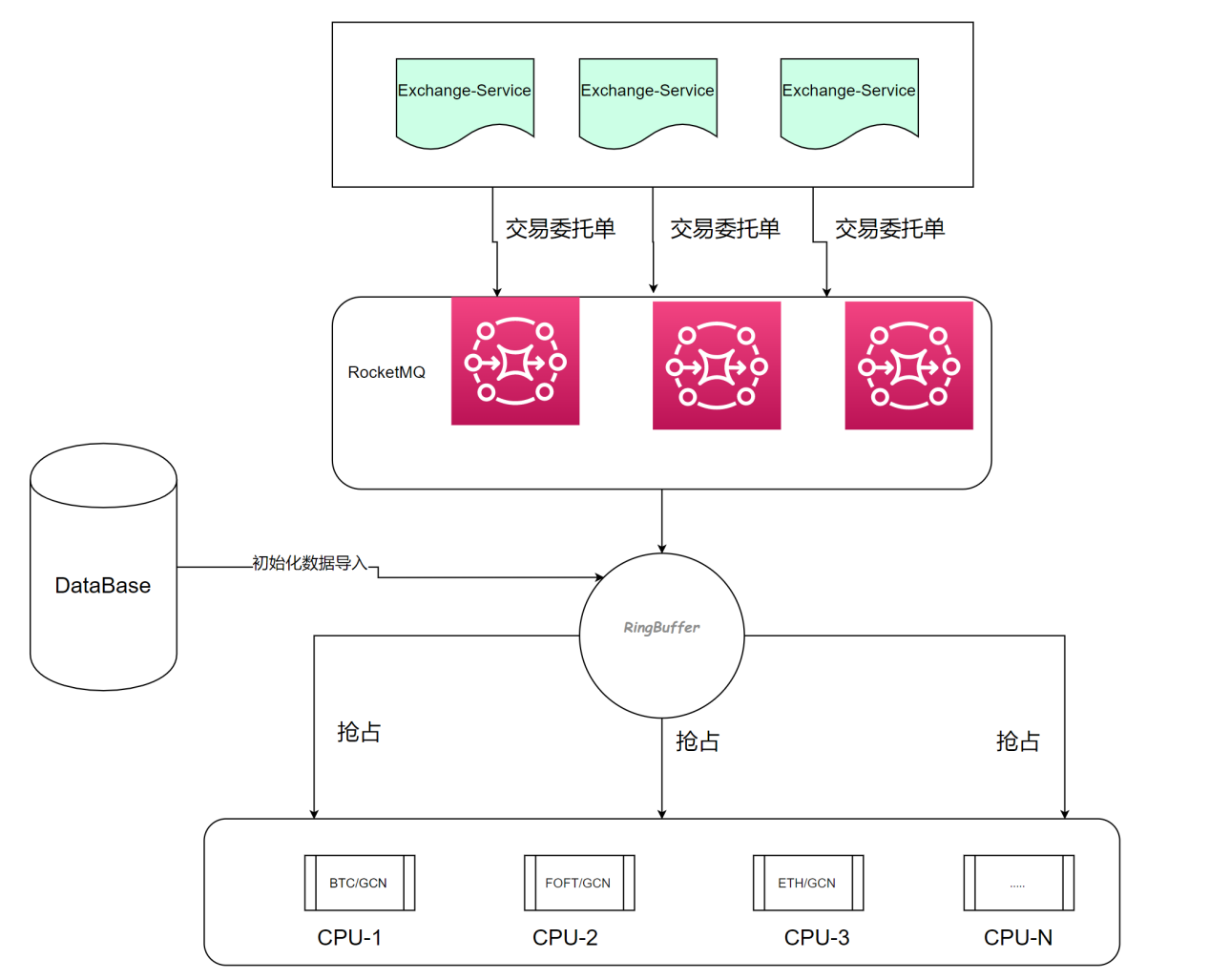
接着谈谈可伸缩性，我们的每一个撮合引擎既可以同时处理多个交易标的，也可以只处理单个交易标的。当交易标的和并发量增多的时候，可以增加服务器，部署成撮合引擎集群，分别用来处理不同的交易标的，从而能够实现负载均衡。

最后聊聊可用性，高可用主要体现在两点，一是故障率要低，二是对故障维修的时间要短。要降低故障率，那撮合引擎就需要有较高的健壮性，对于可能导致引擎出故障的各种异常情况要考虑好并设计好解决方案。另外，还可以采用多机热备份技术来提高可用性，而且要保证互备服务器之间的数据一致，那就需要引入内存状态机复制方案，实现上会复杂很多。

不过，我们并非一下子就要达到很高的质量要求，因为要求越高，其架构和实现会越复杂。我们可以先从简单的版本开始，然后不断升级迭代。

# 数据的流转方式

## 4.1 数据的流转方向



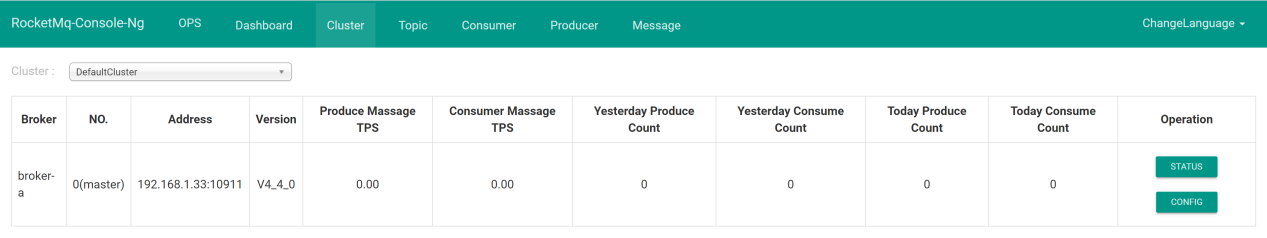
## 4.2 代码的实现

### 4.2.1 准备RocketMQ

参考:02\_环境的搭建\_文档\_五\_RocketMQ的Docker安装

访问:

|  |
| --- |
| http://rocket-server:8080/ |



若不能访问,看看自己的host文件是否正确:

|  |
| --- |
| 192.168.1.33 nacos-server  192.168.1.33 sentinel-server  192.168.1.33 seata-server  192.168.1.33 mysql-server  192.168.1.33 mongo-server  192.168.1.33 redis-server  192.168.1.33 rocket-server |

### 4.2.2 项目添加依赖(供数据的流转)

打开match-service里面的pom.xml文件,添加以下依赖:

|  |
| --- |
| *<!-- disruptor 高速队列-->* <dependency>  <groupId>com.lmax</groupId>  <artifactId>disruptor</artifactId>  </dependency>  *<!-- cpu亲和锁-->* <dependency>  <groupId>net.openhft</groupId>  <artifactId>affinity</artifactId>  <version>${affinity.version}</version>  </dependency>   *<!-- spring-cloud-stream-rocketmq-->* <dependency>  <groupId>com.alibaba.cloud</groupId>  <artifactId>spring-cloud-stream-binder-rocketmq</artifactId>  </dependency> |

### 4.2.3 准备启动类

|  |
| --- |
| @SpringBootApplication @EnableDiscoveryClient public class MatchServiceApplication {   public static void main(String[] args) {  SpringApplication.*run*(MatchServiceApplication.class ,args) ;  } } |

### 4.2.4 准备配置文件

|  |
| --- |
| spring:  application:  name: match-service  profiles:  active: dev  cloud:  nacos:  server-addr: nacos-server:8848  config:  file-extension: yaml *# 去 nacos-server 里面拉取 match-service-dev.yaml* |

## 4.3 Disruptor和SpringBoot的整合

### 4.3.1 DisruptorProperties

|  |
| --- |
| @Data public class DisruptorProperties {   public static final String *PREFIX* = "spring.disruptor";    *// RingBuffer缓冲区大小, 默认 1024 \* 1024* private int ringBufferSize = 1024 \* 1024;  */\*\*  \* 是否为多生产者，如果是则通过 RingBuffer.createMultiProducer创建一个多生产者的RingBuffer，否则通过RingBuffer.createSingleProducer创建一个单生产者的RingBuffer  \*/* private boolean multiProducer = false; } |

### 4.3.2 DisruptorHandlerException

|  |
| --- |
| */\*\*  \* 异常处理  \*/* public class DisruptorHandlerException implements ExceptionHandler<Object> {   private Logger logger = LoggerFactory.*getLogger*(DisruptorHandlerException.class);   @Override  public void handleEventException(Throwable ex, long sequence, Object event) {  ex.printStackTrace();  logger.error("process data error sequence ==[{}] event==[{}] ,ex ==[{}]", sequence, event.toString(), ex.getMessage());  }    @Override  public void handleOnStartException(Throwable ex) {  logger.error("start disruptor error ==[{}]!", ex.getMessage());  }    @Override  public void handleOnShutdownException(Throwable ex) {  logger.error("shutdown disruptor error ==[{}]!", ex.getMessage());  }  } |

### 4.3.3 OrderEvent

|  |
| --- |
| @Data public class OrderEvent implements Serializable {   private static final long *serialVersionUID* = 5516075349620653480L;   */\*\*  \* 时间戳  \*/* private final long timestamp;   */\*\*  \* 事件携带的对象  \*/* protected transient Object source;    public OrderEvent(Object source) {  timestamp = System.*currentTimeMillis*();  this.source = source;  }   public OrderEvent() {  timestamp = System.*currentTimeMillis*();  }  } |
| public class OrderEvent implements Serializable {   private static final long *serialVersionUID* = 5516075349620653480L;   */\*\*  \* 时间戳  \*/* private final long timestamp;   */\*\*  \* 事件携带的对象  \*/* protected transient Object source;    public OrderEvent(Object source) {  timestamp = System.*currentTimeMillis*();  this.source = source;  }   public OrderEvent() {  timestamp = System.*currentTimeMillis*();  }   */\*\*  \* Clearing Objects From the Ring Buffer  \*/* public void clear() {  this.source = null;  }   public long getTimestamp() {  return timestamp;  }   public Object getSource() {  return source;  }   public void setSource(Object source) {  this.source = source;  }  } |

### 4.3.4 Order

|  |
| --- |
| */\*\*  \* 委托单  \*/* @Data @NoArgsConstructor public class Order implements Serializable {   */\*\*  \* 本次订单的Id  \*/* private String orderId;  */\*\*  \* 用户/会员Id  \*/* private Long userId;   */\*\*  \* 支持的币币交易对  \*/* private String symbol;   */\*\*  \* 买入或卖出量  \*/* private BigDecimal amount = BigDecimal.*ZERO*;   */\*\*  \* 成交量  \*/* private BigDecimal tradedAmount = BigDecimal.*ZERO*;   */\*\*  \* 成交额  \*/* private BigDecimal turnover = BigDecimal.*ZERO*;   */\*\*  \* 币单位  \*/* private String coinSymbol;   */\*\*  \* 结算单位  \*/* private String baseSymbol;   */\*\*  \* 订单状态  \*/* private Integer orderStatus;   */\*\*  \* 订单的方向  \*/* private Integer orderDirection;   */\*\*  \* 挂单的价格  \*/* private BigDecimal price = BigDecimal.*ZERO*;   */\*\*  \* 挂单时间  \*/* private Long time;   */\*\*  \* 交易完成时间  \*/* private Long completedTime;   */\*\*  \* 交易取消时间  \*/* private Long cancelTime;   */\*\*  \* 已经成功的水平订单  \*/* private List<OrderDetail> details;   public boolean isCompleted() {  *//TODO* return false ;  } } |

### 4.3.5 OrderDetail

|  |
| --- |
| @Data public class OrderDetail {  */\*\*  \* 订单Id  \*/* private String orderId;   */\*\*  \* 成交价格  \*/* private BigDecimal price;   */\*\*  \* 成交数量  \*/* private BigDecimal amount;   */\*\*  \* 成交额  \*/* private BigDecimal turnover;   */\*\*  \* 费率  \*/* private BigDecimal fee;   */\*\*  \* 成功时间  \*/* private Long dealTime;  } |

### 4.3.6 DisruptorTemplate

|  |
| --- |
| */\*\*  \* 在boot里面使用它发送消息  \*/* public class DisruptorTemplate {   private static final EventTranslatorOneArg<OrderEvent, Order> *TRANSLATOR* = new EventTranslatorOneArg<OrderEvent, Order>() {   public void translateTo(OrderEvent event, long sequence, Order input) {  event.setSource(input);  }  };  private final RingBuffer<OrderEvent> ringBuffer;   public DisruptorTemplate(RingBuffer<OrderEvent> ringBuffer) {  this.ringBuffer = ringBuffer;  }   public void onData(Order input) {  ringBuffer.publishEvent(*TRANSLATOR*, input);  } } |

### 4.3.7 DisruptorAutoConfiguration

|  |
| --- |
| @Configuration @EnableConfigurationProperties(value = DisruptorProperties.class) public class DisruptorAutoConfiguration {   public DisruptorProperties disruptorProperties ;   public DisruptorAutoConfiguration(DisruptorProperties disruptorProperties){  this.disruptorProperties = disruptorProperties ;  }    @Bean  public EventFactory<OrderEvent> eventEventFactory(){   EventFactory<OrderEvent> orderEventEventFactory = new EventFactory<OrderEvent>() {  @Override  public OrderEvent newInstance() {  return new OrderEvent() ;  }  };  return orderEventEventFactory;  }    @Bean  public ThreadFactory threadFactory(){  return new AffinityThreadFactory("Match-Handler:") ;  }    */\*\*  \* 无锁高效的等待策略  \* @return  \*/* @Bean  public WaitStrategy waitStrategy(){  return new YieldingWaitStrategy();  }  */\*\*  \* 创建一个RingBuffer  \* eventFactory: 事件工厂  \* threadFactory: 我们执行者(消费者)的线程该怎么创建  \* waitStrategy : 等待策略: 当我们ringBuffer 没有数据时,我们怎么等待  \*/* @Bean  public RingBuffer<OrderEvent> ringBuffer(  EventFactory<OrderEvent> eventFactory ,  ThreadFactory threadFactory ,  WaitStrategy waitStrategy,  EventHandler<OrderEvent> []eventHandlers  ){   */\*\*  \* 构建disruptor  \*/* Disruptor<OrderEvent> disruptor = null ;   ProducerType producerType = ProducerType.*SINGLE* ;   if (disruptorProperties.isMultiProducer()){  producerType = ProducerType.*MULTI* ;  }   disruptor = new Disruptor<OrderEvent>(eventFactory,disruptorProperties.getRingBufferSize(),threadFactory,producerType,waitStrategy) ;  disruptor.setDefaultExceptionHandler(new DisruptorHandlerException());   *// 设置消费者---我们的每个消费者代表我们的一个交易对,有多少个交易对,我们就有多少个eventHandlers ,事件来了后,多个eventHandlers 是并发执行的* disruptor.handleEventsWith(eventHandlers) ;   RingBuffer<OrderEvent> ringBuffer = disruptor.getRingBuffer();   disruptor.start() ;*// 开始监听* final Disruptor<OrderEvent> disruptorShutdown = disruptor ;   *// 使用优雅的停机* Runtime.*getRuntime*().addShutdownHook(new Thread(  ()->{  disruptorShutdown.shutdown();  },"DisruptorShutdownThread"  ));  return ringBuffer ;  }    */\*\*  \* 创建DisruptorTemplate  \* @param ringBuffer  \* @return  \*/* @Bean  public DisruptorTemplate disruptorTemplate(RingBuffer<OrderEvent> ringBuffer){  return new DisruptorTemplate(ringBuffer) ;  } } |

### 4.3.8 OrderMatchDisruptorHandler

|  |
| --- |
| public class OrderMatchDisruptorHandler implements EventHandler<OrderEvent> {   private static final Logger *LOG* = LoggerFactory.*getLogger*(OrderMatchDisruptorHandler.class);    */\*\*  \* 交给引擎去处理事件  \*  \* @param event  \* @throws Exception  \*/* @Override  public void onEvent(OrderEvent event, long sequence, boolean endOfBatch) throws Exception {  Order order = (Order) event.getSource();  *LOG*.info("====> 开始接受委托单{} <======", JSON.*toJSONString*(order));  try {  } catch (Exception e) {  *LOG*.error(String.*format*("处理委托单：%s 发生了错误", JSON.*toJSON*(order)), e);  } finally {   }  } } |

# 使用SpringCloudStreamRocketMQ

## 5.1 创建一个Sink

|  |
| --- |
| public interface Sink {   @Input("order.in")  public MessageChannel messageChannel() ; } |

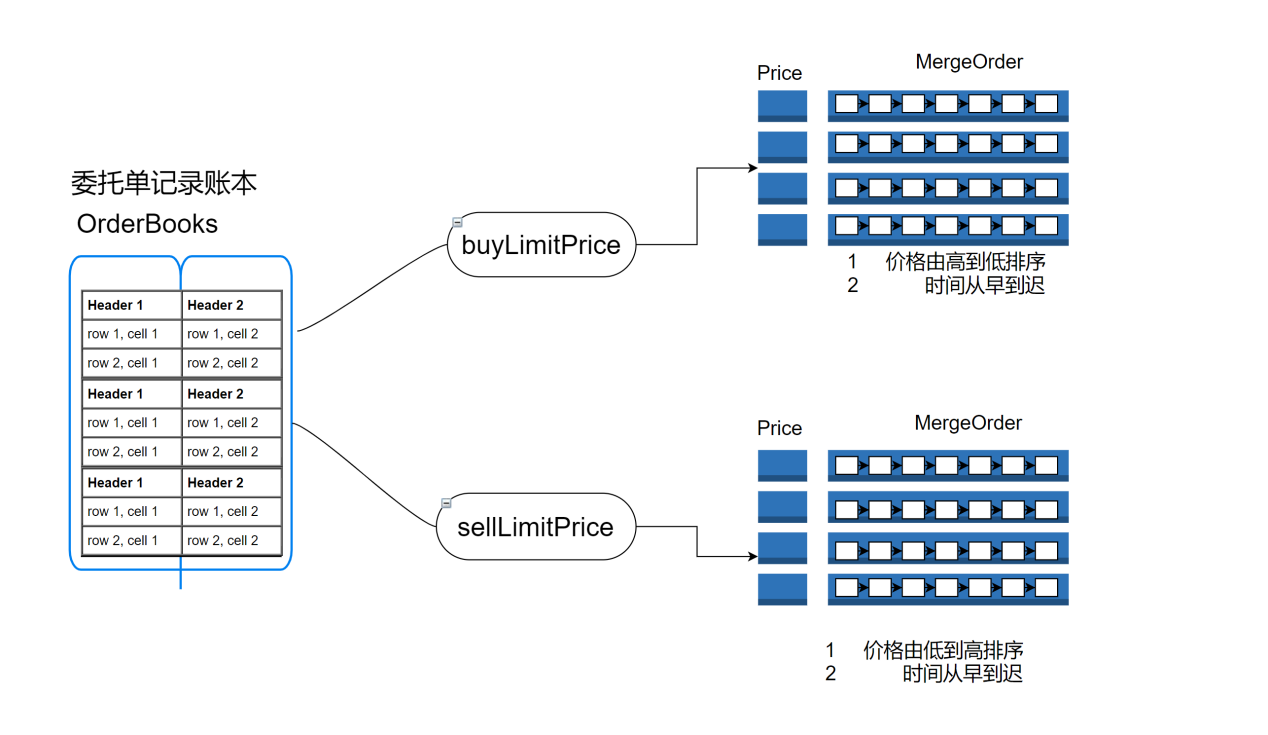
## 5.2 创建监听器

|  |
| --- |
| @Service @Slf4j public class MessageConsumerListener {   @Autowired  private DisruptorTemplate disruptorTemplate ;   @StreamListener("order.in")  public void handleMessage(Order order){  *log*.info("接收到了委托单:{}",order);  disruptorTemplate.onData(order);  } } |

## 5.3 配置类

|  |
| --- |
| */\*\*  \* 开启我们的Stream的开发  \*/* @Configuration @EnableBinding(Sink.class) *//* public class RocketStreamConfig { } |

# 委托单账本数据结构设计



## 6.1 委托单的数据结构

|  |
| --- |
| @Data @Slf4j public class OrderBooks {   */\*\*  \* 买入的限价交易 价格从高到底  \* eg: 价格越高，越容易买到  \* Key: 价格  \* MergeOrder 同价格的订单，订单按照时间排序  \*/* private TreeMap<BigDecimal, MergeOrder> buyLimitPrice;   */\*\*  \* 卖出的限价交易，价格从低到高  \* eg: 价格越低，卖出的越容易  \*/* private TreeMap<BigDecimal, MergeOrder> sellLimitPrice;  */\*\*  \* 交易的币种  \*/* private String symbol;   */\*\*  \* 交易币种的精度  \*/* private int coinScale;   */\*\*  \* 基币的精度  \*/* private int baseCoinScale;    */\*\*  \* 日期格式器  \*/* private SimpleDateFormat dateTimeFormat;  } |

## 6.2 合并订单的数据结构

|  |
| --- |
| */\*\*  \* 订单的合并  \*/* public class MergeOrder {   private LinkedList<Order> orders = new LinkedList<>();   *//最后位置添加一个* public void add(Order order) {  orders.addLast(order);  }   // 从头获取数据  public Order get() {  return orders.get(0);  }   public int size() {  return orders.size();  }   public BigDecimal getPrice() {  return orders.get(0).getPrice();  }   public Iterator<Order> iterator() {  return orders.iterator();  }   public BigDecimal getTotalAmount() {  BigDecimal total = new BigDecimal(0);  for (Order item : orders) {  total = total.add(item.getAmount());  }  return total;  } } |

## 6.3 订单的方向枚举

|  |
| --- |
| public enum OrderDirection {   *BUY*(1,"买入") ,  *SELL*(2,"卖出") ;   private int code ;   private String desc ;    OrderDirection(int code,String desc){  this.code = code ;  this.desc = desc ;  }   public int getCode() {  return code;  }   public void setCode(int code) {  this.code = code;  }   public String getDesc() {  return desc;  }   public void setDesc(String desc) {  this.desc = desc;  }   public static OrderDirection getOrderDirection(int code){  OrderDirection[] values = OrderDirection.*values*();  for (OrderDirection value : values) {  if(value.getCode()==code){  return value ;  }  }  return null ;  }  } |

## 6.4 需要实现的方法

### 6.4.1 初始化订单队列

|  |
| --- |
| */\*\*  \* 初始化订单队列  \*/* public void initialize() {  *log*.info("init CoinTrader for symbol {}", symbol);  *// 载入比较器* buyLimitPrice = new TreeMap<>(Comparator.*reverseOrder*()); *//价格从大到小* sellLimitPrice = new TreeMap<>(Comparator.*naturalOrder*()); *// 价格从小到大* dateTimeFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss"); } |

### 6.4.2 获取当前需要的交易队列

|  |
| --- |
| */\*\*  \* 获取当前的交易队列  \* @param orderDirection 交易方向  \* @return  \*/* public TreeMap<BigDecimal,MergeOrder> getCurrentOrders(OrderDirection orderDirection){  return orderDirection == OrderDirection.*BUY* ? this.buyLimitPrice : this.sellLimitPrice ; } |

### 6.4.3 获取当前需要的交易队列的迭代器

|  |
| --- |
| */\*\*  \* 获取当前交易队列的迭代器  \* @param orderDirection  \* @return  \*/* public Iterator<Map.Entry<BigDecimal,MergeOrder>> getCurrentOrderIterator(OrderDirection orderDirection){  return getCurrentOrders(orderDirection).entrySet().iterator(); } |

### 6.4.4 添加到交易队列里面

|  |
| --- |
| */\*\*  \* 将订单添加到限价队列里面，限价队列的数据是使用价格和时间排序的  \*  \* @param order  \*/* public void addOrder(Order order) {   TreeMap<BigDecimal, MergeOrder> limitPriceMap = getCurrentOrders(OrderDirection.*getOrderDirection*(order.getOrderDirection()));   MergeOrder mergeOrder = buyLimitPrice.get(order.getPrice());  *// 注意，此处均为单线程操作，无需考虑并发问题，当为集群或多线程时， 需要添加锁/分布式锁* if (mergeOrder == null) { *// 之前不存在* mergeOrder = new MergeOrder();  *// 之前的二叉树里面不存在该节点，插入进去* limitPriceMap.put(order.getPrice(), mergeOrder);  }  *// 添加到水平的订单里面* mergeOrder.add(order); } |

### 6.4.5 添加到交易队列里面

|  |
| --- |
| */\*\*  \* 从交易队列里面移除  \* @param order  \*/* public void cancelOrder(Order order) {  */\*\*  \* 获取当前要操作的数据容器  \*/* TreeMap<BigDecimal, MergeOrder> limitPriceMap =  getCurrentOrders(OrderDirection.*getOrderDirection*(order.getOrderDirection())) ;   MergeOrder mergeOrder = limitPriceMap.get(order.getPrice());  if (mergeOrder == null || mergeOrder.size() <= 0) {  return;  }  Iterator<Order> iterator = mergeOrder.iterator();  while (iterator.hasNext()) {  Order each = iterator.next();  if (each.getOrderId().equals(order.getOrderId())) {  iterator.remove(); *// 移除该订单* }  }  */\*\*  \* 订单移除成功后，需要判断，移除此订单后，是否要移除之前的二叉树结点  \*/* if (mergeOrder.size() <= 0) { *//O(1)* limitPriceMap.remove(order.getPrice());  } } |

### 6.4.6 获取排在队列里面的第一个数据

|  |
| --- |
| */\*\*  \* 获取排在队列里面的第一个数据  \* @param orderDirection  \* @return  \*/* public Map.Entry<BigDecimal ,MergeOrder> getBestSuitPriceMergeOrder(OrderDirection orderDirection){  return getCurrentOrders(orderDirection).firstEntry(); } |

### 6.4.7 构造器

|  |
| --- |
| public OrderBooks(String symbol){  this(symbol,4,4) ; }  public OrderBooks(String symbol,int coinScale,int baseCoinScale){  this.symbol = symbol ;  this.coinScale = coinScale ;  this.coinScale = baseCoinScale ;  this.initialize(); } |

## 6.3 OrderBooks的所有代码

|  |
| --- |
| @Data @Slf4j public class OrderBooks {   */\*\*  \* 买入的限价交易 价格从高到底  \* eg: 价格越高，越容易买到  \* Key: 价格  \* MergeOrder 同价格的订单，订单按照时间排序  \*/* private TreeMap<BigDecimal, MergeOrder> buyLimitPrice;   */\*\*  \* 卖出的限价交易，价格从低到高  \* eg: 价格越低，卖出的越容易  \*/* private TreeMap<BigDecimal, MergeOrder> sellLimitPrice;  */\*\*  \* 交易的币种  \*/* private String symbol;   */\*\*  \* 交易币种的精度  \*/* private int coinScale;   */\*\*  \* 基币的精度  \*/* private int baseCoinScale;    */\*\*  \* 日期格式器  \*/* private SimpleDateFormat dateTimeFormat;    public OrderBooks(String symbol){  this(symbol,4,4) ;  }   public OrderBooks(String symbol,int coinScale,int baseCoinScale){  this.symbol = symbol ;  this.coinScale = coinScale ;  this.coinScale = baseCoinScale ;  this.initialize();  }  */\*\*  \* 初始化订单队列  \*/* private void initialize() {  *log*.info("init CoinTrader for symbol {}", symbol);  *// 载入比较器* buyLimitPrice = new TreeMap<>(Comparator.*reverseOrder*()); *//价格从大到小* sellLimitPrice = new TreeMap<>(Comparator.*naturalOrder*()); *// 价格从小到大* dateTimeFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  }    */\*\*  \* 获取当前的交易队列  \* @param orderDirection 交易方向  \* @return  \*/* public TreeMap<BigDecimal,MergeOrder> getCurrentOrders(OrderDirection orderDirection){  return orderDirection == OrderDirection.*BUY* ? this.buyLimitPrice : this.sellLimitPrice ;  }    */\*\*  \* 获取当前交易队列的迭代器  \* @param orderDirection  \* @return  \*/* public Iterator<Map.Entry<BigDecimal,MergeOrder>> getCurrentOrderIterator(OrderDirection orderDirection){  return getCurrentOrders(orderDirection).entrySet().iterator();  }      */\*\*  \* 将订单添加到限价队列里面，限价队列的数据是使用价格和时间排序的  \*  \* @param order  \*/* public void addOrder(Order order) {   TreeMap<BigDecimal, MergeOrder> limitPriceMap = getCurrentOrders(OrderDirection.*getOrderDirection*(order.getOrderDirection()));   MergeOrder mergeOrder = buyLimitPrice.get(order.getPrice());  *// 注意，此处均为单线程操作，无需考虑并发问题，当为集群或多线程时， 需要添加锁/分布式锁* if (mergeOrder == null) { *// 之前不存在* mergeOrder = new MergeOrder();  *// 之前的二叉树里面不存在该节点，插入进去* limitPriceMap.put(order.getPrice(), mergeOrder);  }  *// 添加到水平的订单里面* mergeOrder.add(order);  }    */\*\*  \* 从交易队列里面移除  \* @param order  \*/* public void cancelOrder(Order order) {  */\*\*  \* 获取当前要操作的数据容器  \*/* TreeMap<BigDecimal, MergeOrder> limitPriceMap =  getCurrentOrders(OrderDirection.*getOrderDirection*(order.getOrderDirection())) ;   MergeOrder mergeOrder = limitPriceMap.get(order.getPrice());  if (mergeOrder == null || mergeOrder.size() <= 0) {  return;  }  Iterator<Order> iterator = mergeOrder.iterator();  while (iterator.hasNext()) {  Order each = iterator.next();  if (each.getOrderId().equals(order.getOrderId())) {  iterator.remove(); *// 移除该订单* }  }  */\*\*  \* 订单移除成功后，需要判断，移除此订单后，是否要移除之前的二叉树结点  \*/* if (mergeOrder.size() <= 0) { *//O(1)* limitPriceMap.remove(order.getPrice());  }  } } |

# 撮合引擎的实现

## 7.1 MatchEngineProperties

|  |
| --- |
| @Data @ConfigurationProperties(prefix = "spring.match") public class MatchEngineProperties {   */\*\*  \* 交易对的信息  \*/* private Map<String,CoinScale> symbols ;   @Data  public static class CoinScale {  */\*\*  \* 交易币种的精度  \*/* private int coinScale;   */\*\*  \* 基币的精度  \*/* private int baseCoinScale;  } } |

## 7.2 MatchEnginAutoConfiguration

|  |
| --- |
| @Configuration @EnableConfigurationProperties(value = MatchEngineProperties.class) public class MatchEngineAutoConfiguration {   private MatchEngineProperties matchEngineProperties;    public MatchEngineAutoConfiguration(MatchEngineProperties matchEngineProperties) {  this.matchEngineProperties = matchEngineProperties;  }    @Bean("eventHandlers")  public EventHandler[] eventHandlers() {  Map<String, MatchEngineProperties.CoinScale> symbols = matchEngineProperties.getSymbols();  Set<Map.Entry<String, MatchEngineProperties.CoinScale>> entries = symbols.entrySet();  EventHandler[] eventHandlers = new EventHandler[symbols.size()];  int i = 0;  for (Map.Entry<String, MatchEngineProperties.CoinScale> entry : entries) {  String symbol = entry.getKey();  MatchEngineProperties.CoinScale value = entry.getValue();  OrderBooks orderBooks = null;  if (value != null) {  orderBooks = new OrderBooks(symbol, value.getCoinScale(), value.getBaseCoinScale());  } else {  orderBooks = new OrderBooks(symbol);  }  eventHandlers[i++] = new OrderEventHandler(orderBooks);  }  return eventHandlers;   }  } |

## 7.3 交易策略枚举MatchStrategy

|  |
| --- |
| public enum MatchStrategy {   */\*\*  \* 限价交易  \*/  LIMIT\_PRICE* ,   */\*\*  \* 市场交易  \*/  MARKER\_PRICE* ; } |

## 7.4 交易策略定义

|  |
| --- |
| public interface MatchService {   */\*\*  \* 执行撮合交易  \* @param order  \*/* void match(Order order) ; } |

|  |
| --- |
| @Service @Slf4j public class LimitPriceMatchServiceImpl implements MatchService, InitializingBean {   */\*\*  \* 进行订单的撮合交易  \*  \* @param order  \*/* @Override  public void match(Order order) {  *log*.info("开始撮合");  }    @Override  public void afterPropertiesSet() throws Exception {   MatchServiceFactory.*addMatchService*(MatchStrategy.*LIMIT\_PRICE*,this);  } } |

## 7.5 交易策略工厂StrategyFactory

|  |
| --- |
| public class MatchServiceFactory {   private static Map<MatchStrategy,MatchService> *matchServiceMap* = new HashMap<>() ;    */\*\*  \* 给我们的策略工厂里面添加一个交易的实现类型  \* @param matchStrategy  \* @param matchService  \*/* public static void addMatchService(MatchStrategy matchStrategy,MatchService matchService){  *matchServiceMap*.put(matchStrategy ,matchService ) ;  }    */\*\*  \* 使用策略的名称获取具体的实现类  \* @param matchStrategy  \* @return  \*/* public static MatchService getMatchService(MatchStrategy matchStrategy){  return *matchServiceMap*.get(matchStrategy) ;  } } |

## 7.6 在OrderEventHandler调用某种交易策略



## 7.7 并发的情况思考

1 OrderEventHandler

每个Symbol 都有一个和它对应,它是个多例对象,但是我们在这个地方:



2 OrderBooks 和一个交易对绑定,没有线程安全问题

3 MatchService 只有一个,若我们将OrderBooks 当成它的一个属性,就会有并发问题,因此,我们的这个参数使用方法传递过去:

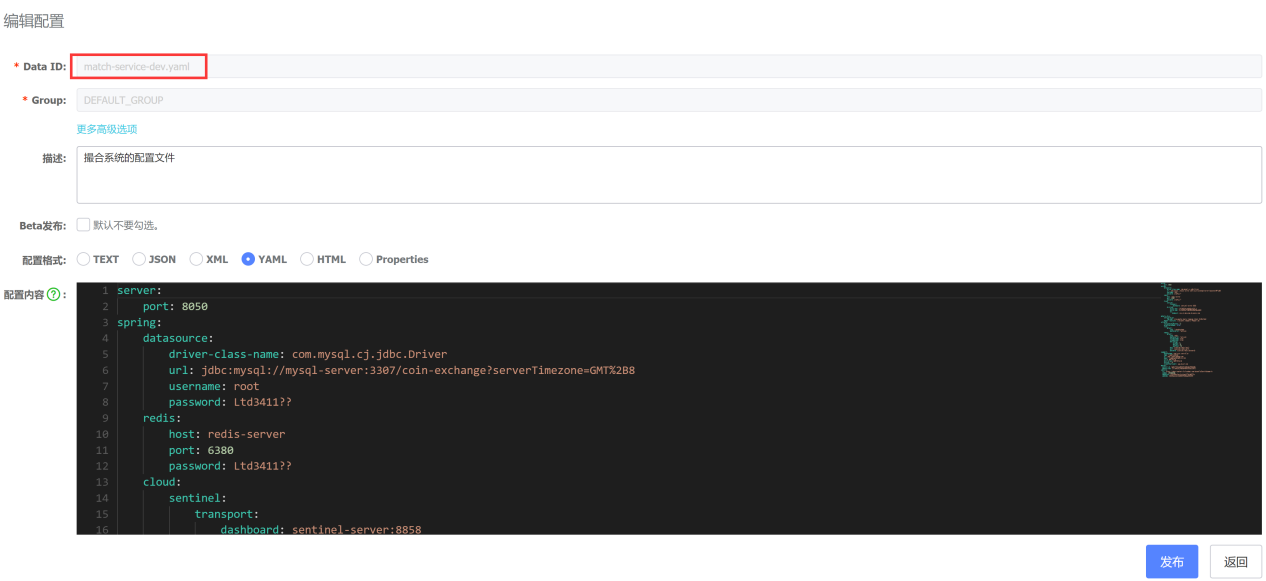


## 7.8 初始数据的加载

|  |
| --- |
| @Component public class DataLoaderCmdLine implements CommandLineRunner {   @Autowired  private DisruptorTemplate disruptorTemplate;   @Autowired  private EntrustOrderMapper entrustOrderMapper;    @Override  public void run(String... args) throws Exception {  List<EntrustOrder> entrustOrders =  entrustOrderMapper.selectList(  new LambdaQueryWrapper<EntrustOrder>()  .eq(EntrustOrder::getStatus, 0)  .orderByAsc(EntrustOrder::getCreated)  );  if (!CollectionUtils.*isEmpty*(entrustOrders)) {  for (EntrustOrder entrustOrder : entrustOrders) {  disruptorTemplate.onData(entrustOrder2Order(entrustOrder));  }  }   }   private Order entrustOrder2Order(EntrustOrder entrustOrder) {  Order order = new Order();  order.setSymbol(entrustOrder.getSymbol()); *// 设置交易对* order.setAmount(entrustOrder.getVolume().add(entrustOrder.getDeal().negate())); *// 设置交易额* order.setPrice(entrustOrder.getPrice()); *// 设置交易价格* order.setTime(entrustOrder.getCreated().getTime()); *// 设置交易时间* order.setOrderId(entrustOrder.getId().toString()); *// 设置交易的id* order.setOrderDirection(entrustOrder.getType().intValue()); *// 设置交易的方向* return order;  } } |

# 测试数据的流转

## 8.1 添加配置文件



内容为:

|  |
| --- |
| server:  port: 8050  spring:  datasource:  driver-class-name: com.mysql.cj.jdbc.Driver  url: jdbc:mysql://mysql-server:3307/coin-exchange?serverTimezone=GMT%2B8  username: root  password: Ltd3411??  redis:  host: redis-server  port: 6380  password: Ltd3411??  cloud:  sentinel:  transport:  dashboard: sentinel-server:8858  alicloud:  access-key: LTAI4GEXnLoaQg6UraLEJL1v  secret-key: txr1DH29usTcQo5MUSsDGoVBusQQHX  oss:  endpoint: oss-cn-beijing.aliyuncs.com  mybatis-plus:  configuration:  log-impl: org.apache.ibatis.logging.stdout.StdOutImpl  mapper-locations: classpath:/mappers/\*Mapper.xml  jetcache:  statIntervalMinutes: 15  areaInCacheName: false  local:  default:  type: linkedhashmap  keyConvertor: fastjson  remote:  default:  type: redis  keyConvertor: fastjson  valueEncoder: kryo  valueDecoder: kryo  poolConfig:  minIdle: 5  maxIdle: 20  maxTotal: 50  host: ${spring.redis.host}  port: ${spring.redis.port}  password: ${spring.redis.password}  swagger2:  basePackage: com.bjsxt.controller  name: liangtiandong  url: www.liangtiandong.com  email: liangtiandong@live.com  title: 交易系统API接口  description: 交易系统API接口演示  version: 1.0  termsOfServiceUrl: www.bjsxt.com  geetest:  geetest-id: 3a01ffc01c1d63b37c3dbe8ee9555290  geetest-key: 27c7b4a18124d5d649b9c58ca1830871  identify:  url: https://idcert.market.alicloudapi.com/idcard?idCard=%s&name=%s  appKey: 203866940  appSecret: KF6LRDKejSejx2vfyLqiLP7bVvWOZYCp  appCode: 39e89acacec14a9d80782d8aa2893295 |

## 8.2 RocketMq的配置文件

|  |
| --- |
| stream:  rocketmq:  binder:  name-server: rocketmq-server:9876  binders:  order-in: {consumer.orderly: true}  bindings:  order-in: {destination: order-in, content-type: application/plain, group: order-group, consumer.maxAttempts: 1} |

## 8.3 新建MatchController获取数据

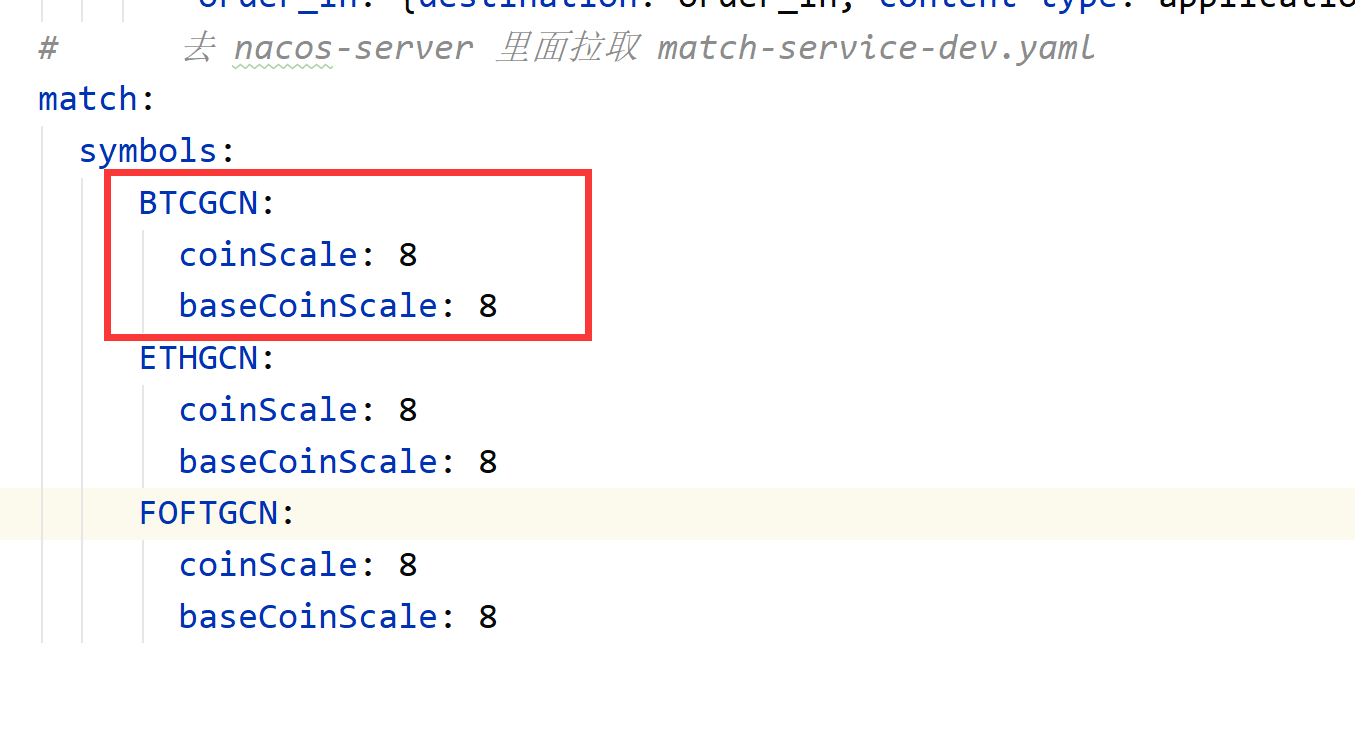
|  |
| --- |
| @RestController public class MatchController {   @Autowired  private EventHandler<OrderEvent>[] eventHandlers;    @GetMapping("/match/order")  public TreeMap<BigDecimal, MergeOrder> getTradeData(@RequestParam(required = true) String symbol, @RequestParam(required = true)Integer orderDirection) {  for (EventHandler<OrderEvent> eventHandler : eventHandlers) {  OrderEventHandler orderEventHandler = (OrderEventHandler) eventHandler;  if (orderEventHandler.getSymbol().equals(symbol)) {  OrderBooks orderBooks = orderEventHandler.getOrderBooks();  return orderBooks.getCurrentLimitPrices(OrderDirection.*getOrderDirection*(orderDirection));  }  }  return null;  } } |

发现了2 处Bug:

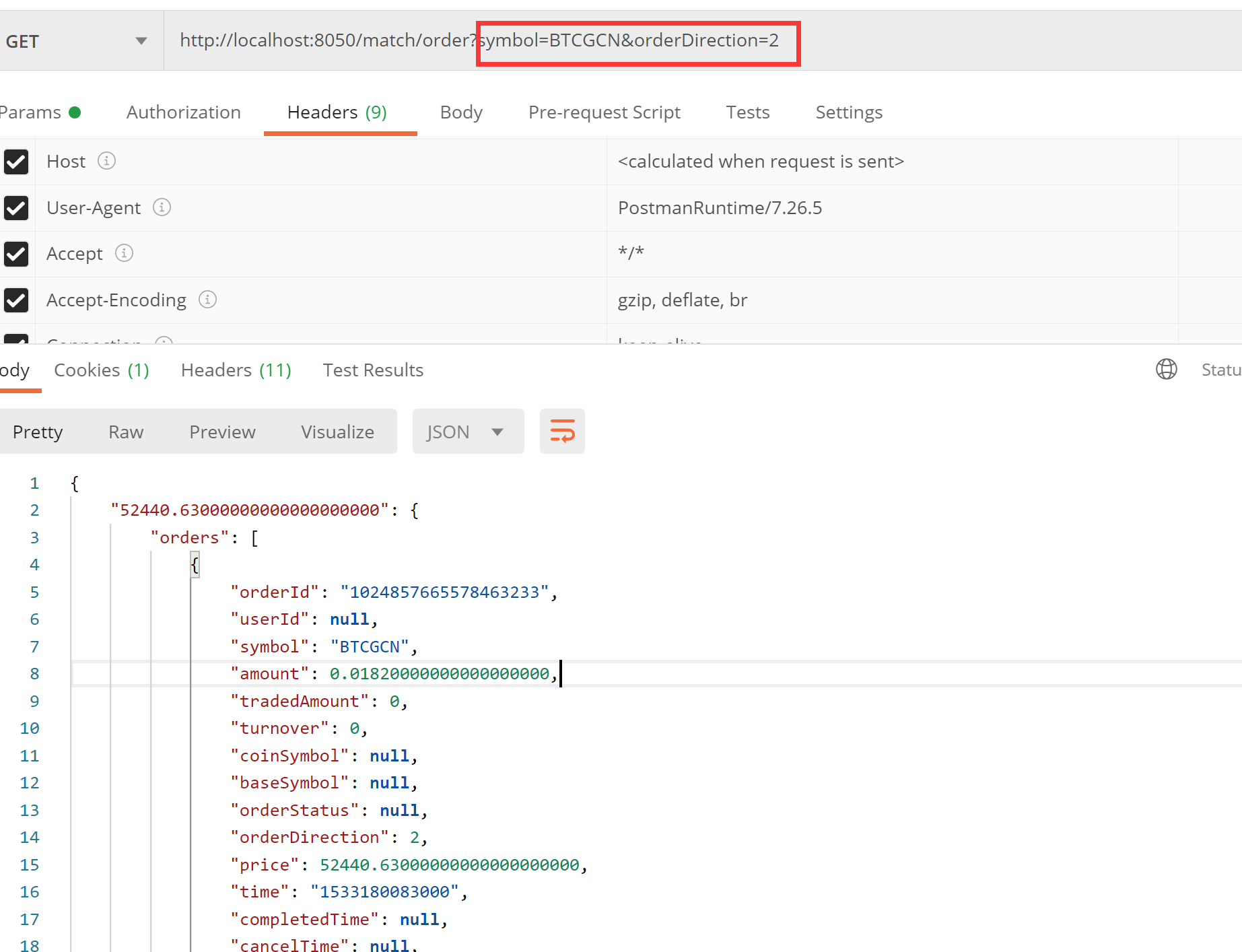
OrderBooks:



配置文件:



测试成功的效果:



## 8.4 在Exchange-Service 里面发送消息到消息队列测试

### 8.4.1 添加依赖Exchange-Service的pom.xml

|  |
| --- |
| *<!-- spring-cloud-stream-rocketmq-->* <dependency>  <groupId>com.alibaba.cloud</groupId>  <artifactId>spring-cloud-stream-binder-rocketmq</artifactId> </dependency> |

### 8.4.2 配置文件

|  |
| --- |
| stream:  bindings:  order\_in: {destination: order\_in, content-type: application/plain, group: order-group, consumer.maxAttempts: 1}  rocketmq:  binder:  name-server: rocket-server:9876 |

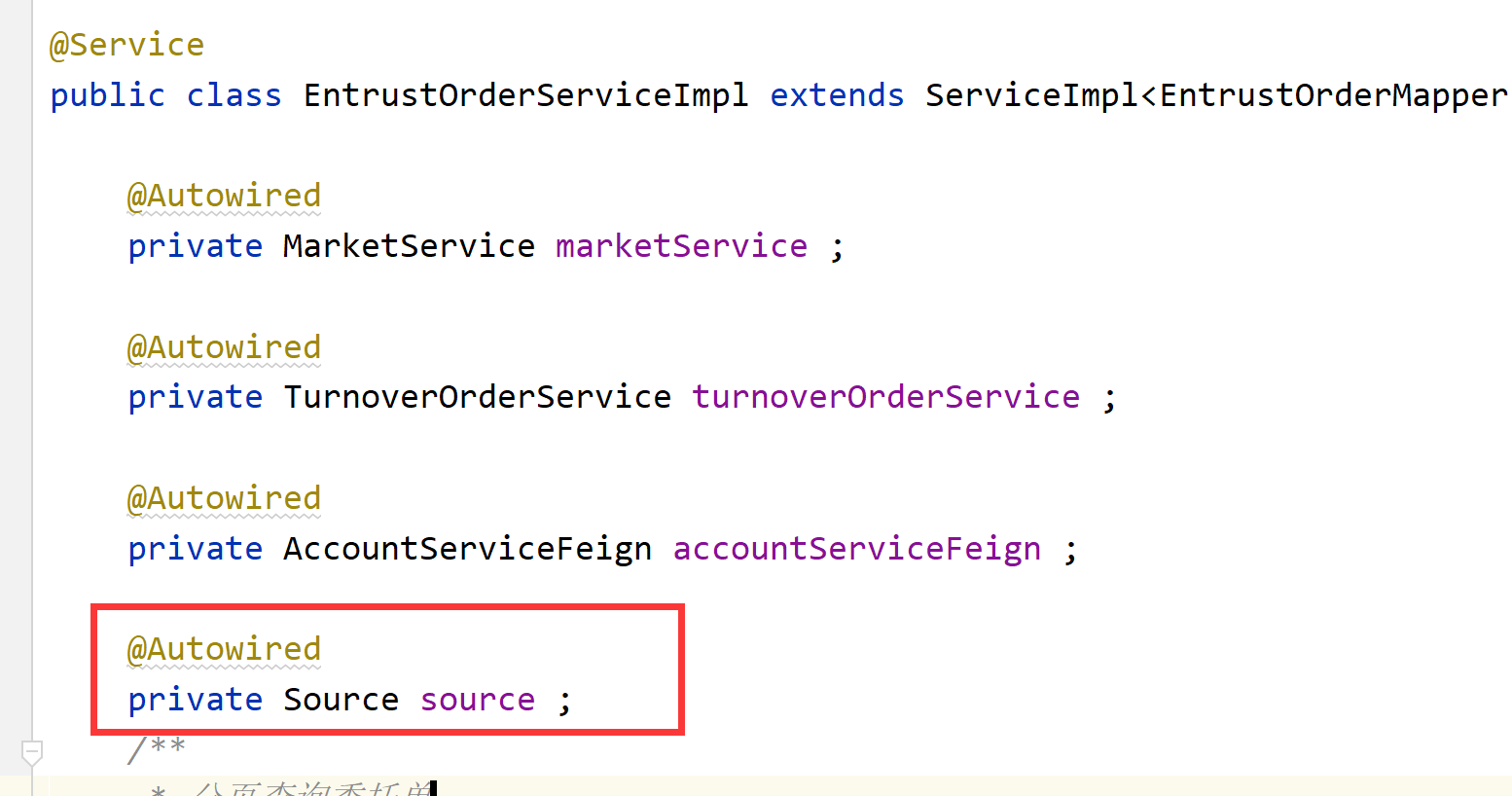
### 8.4.3 Source

|  |
| --- |
| public interface Source {   @Output("order\_out")  MessageChannel outputMessage() ; } |

### 8.4.4 RocketMQConfig

|  |
| --- |
| @Configuration @EnableBinding(value = Source.class) public class RocketMQConfig { } |

### 8.4.5 在EntrustOrderServiceImpl发送消息



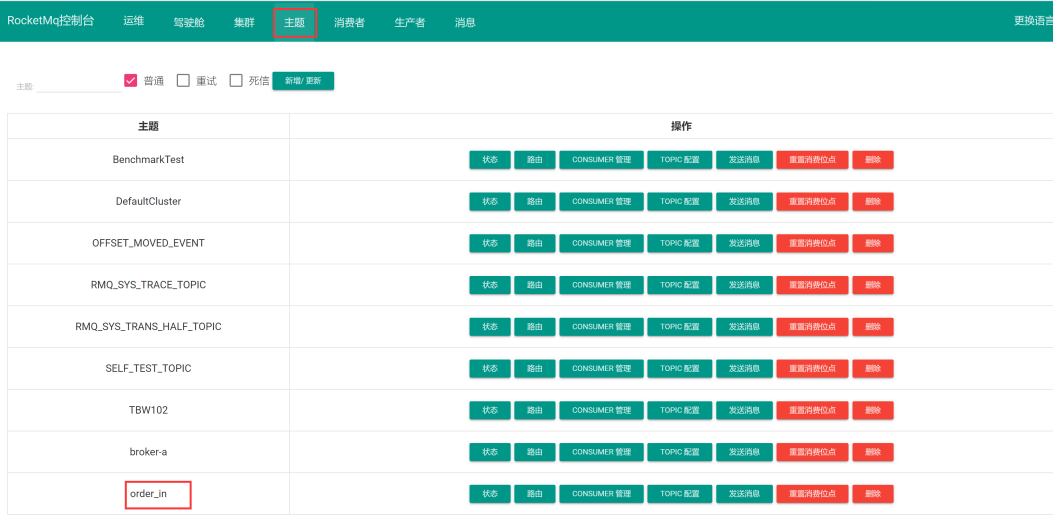
|  |
| --- |
| */\*\*  \* 创建一个新的委托单  \*  \* @param userId 用户的id  \* @param orderParam 委托单的数据  \* @return  \*/* @Override public Boolean createEntrustOrder(Long userId, OrderParam orderParam) {  *// 1 层层校验* @NotBlank String symbol = orderParam.getSymbol();  Market markerBySymbol = marketService.getMarkerBySymbol(symbol);  if(markerBySymbol==null){  throw new IllegalArgumentException("您购买的交易对不存在") ;  }   BigDecimal price = orderParam.getPrice().setScale(markerBySymbol.getPriceScale(), RoundingMode.*HALF\_UP*);  BigDecimal volume = orderParam.getVolume().setScale(markerBySymbol.getNumScale(), RoundingMode.*HALF\_UP*);   *// 计算成交额度* BigDecimal mum = price.multiply(volume) ;   *// 交易数量的交易* @NotNull BigDecimal numMax = markerBySymbol.getNumMax();  @NotNull BigDecimal numMin = markerBySymbol.getNumMin();  if(volume.compareTo(numMax)>0 || volume.compareTo(numMin)<0 ){  throw new IllegalArgumentException("交易的数量不在范围内") ;  }   *// 校验交易额* BigDecimal tradeMin = markerBySymbol.getTradeMin();  BigDecimal tradeMax = markerBySymbol.getTradeMax();   if(mum.compareTo(tradeMin)<0 || mum.compareTo(tradeMax)>0){  throw new IllegalArgumentException("交易的额度不在范围内") ;  }  *// 计算手续费* BigDecimal fee = BigDecimal.*ZERO* ;  BigDecimal feeRate = BigDecimal.*ZERO* ;  @NotNull Integer type = orderParam.getType();  if(type==1){ *// 买入 buy* feeRate = markerBySymbol.getFeeBuy() ;  fee = mum.multiply(markerBySymbol.getFeeBuy()) ;  }else{ *// 卖出 sell* feeRate = markerBySymbol.getFeeSell() ;  fee = mum.multiply(markerBySymbol.getFeeSell()) ;  }  EntrustOrder entrustOrder = new EntrustOrder();  entrustOrder.setUserId(userId);  entrustOrder.setAmount(mum);  entrustOrder.setType(orderParam.getType().byteValue());  entrustOrder.setPrice(price);  entrustOrder.setVolume(volume);  entrustOrder.setFee(fee);  entrustOrder.setCreated(new Date());  entrustOrder.setStatus((byte)0);  entrustOrder.setMarketId(markerBySymbol.getId());  entrustOrder.setMarketName(markerBySymbol.getName());  entrustOrder.setMarketType(markerBySymbol.getType());  entrustOrder.setSymbol(markerBySymbol.getSymbol());  entrustOrder.setFeeRate(feeRate);  entrustOrder.setDeal(BigDecimal.*ZERO*);  entrustOrder.setFreeze(entrustOrder.getAmount().add(entrustOrder.getFee())); *// 冻结余额* boolean save = save(entrustOrder);  if(save){  *// 用户余额的扣减* @NotNull Long coinId = null ;  if(type==1){ *// 购买操作* coinId = markerBySymbol.getBuyCoinId();   }else{  coinId = markerBySymbol.getSellCoinId() ;  }  accountServiceFeign.lockUserAmount(userId,coinId,entrustOrder.getFreeze(),"trade\_create",entrustOrder.getId(),fee);  *// 发送到撮合系统里面* **MessageBuilder<EntrustOrder> entrustOrderMessageBuilder = MessageBuilder.*withPayload*(entrustOrder).setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*);**  **source.outputMessage().send(entrustOrderMessageBuilder.build());**  }  return save ; } |

## 8.5 启动项目创建委托单

进行买入和卖出都是可以的.



## 8.6 发送成功后



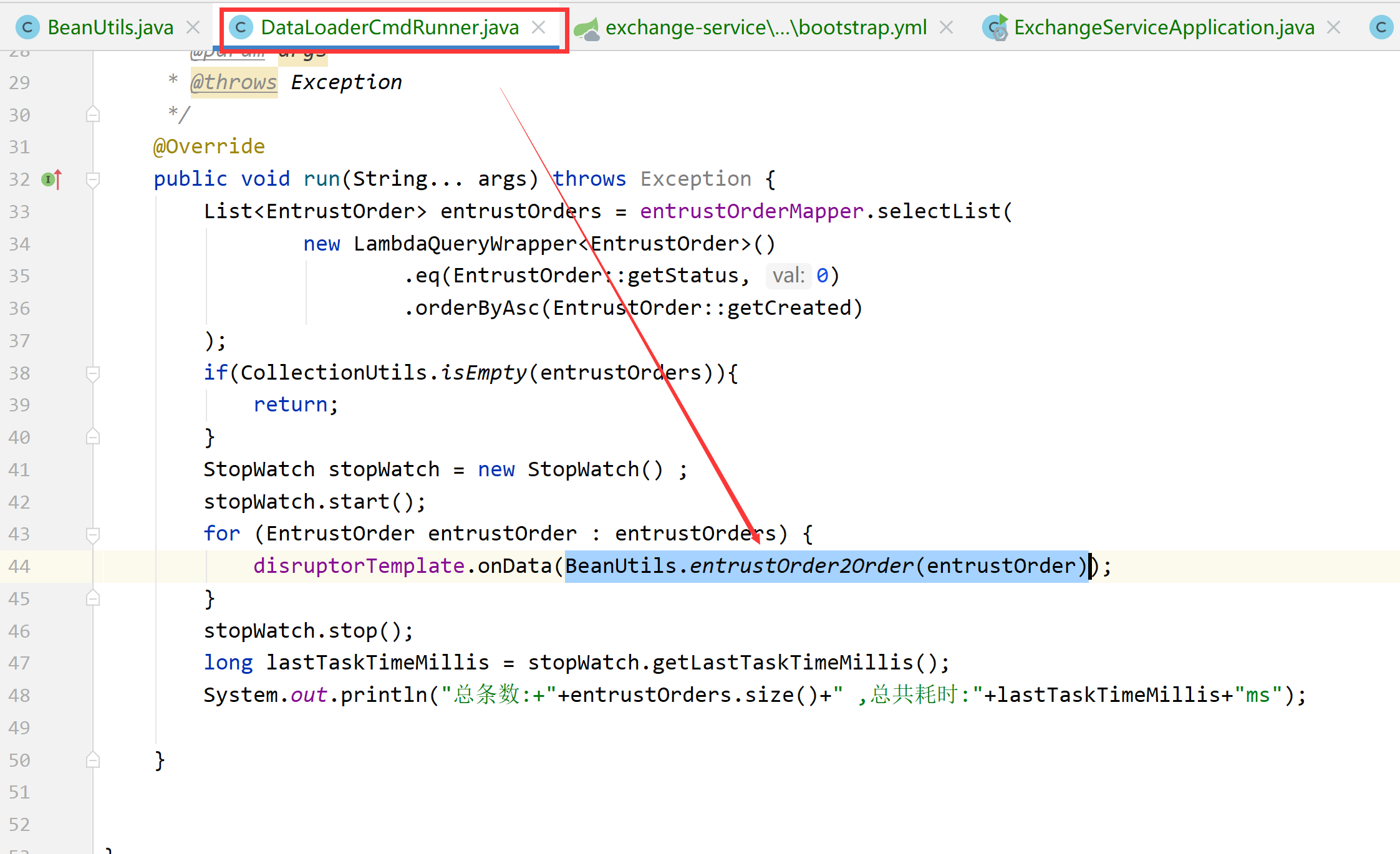
## 8.7 MatchService 代码改造

### 8.7.1 提取工具类

|  |
| --- |
| public class BeanUtils {    */\*\*  \* 将EntrustOrder 转化为我们的Order  \* @param entrustOrder  \* @return  \*/* public static Order entrustOrder2Order(EntrustOrder entrustOrder) {  Order order = new Order();  order.setOrderId(entrustOrder.getId().toString());   order.setPrice(entrustOrder.getPrice());  order.setAmount(entrustOrder.getVolume().add(entrustOrder.getDeal().negate())); *// 交易的数量= 总数量- 已经成交的数量* order.setSymbol(entrustOrder.getSymbol());  order.setOrderDirection(entrustOrder.getType().intValue());  order.setTime(entrustOrder.getCreated().getTime());  return order ;   } } |

### 8.7.2 改造代码

#### DataLoaderCmdRunner:

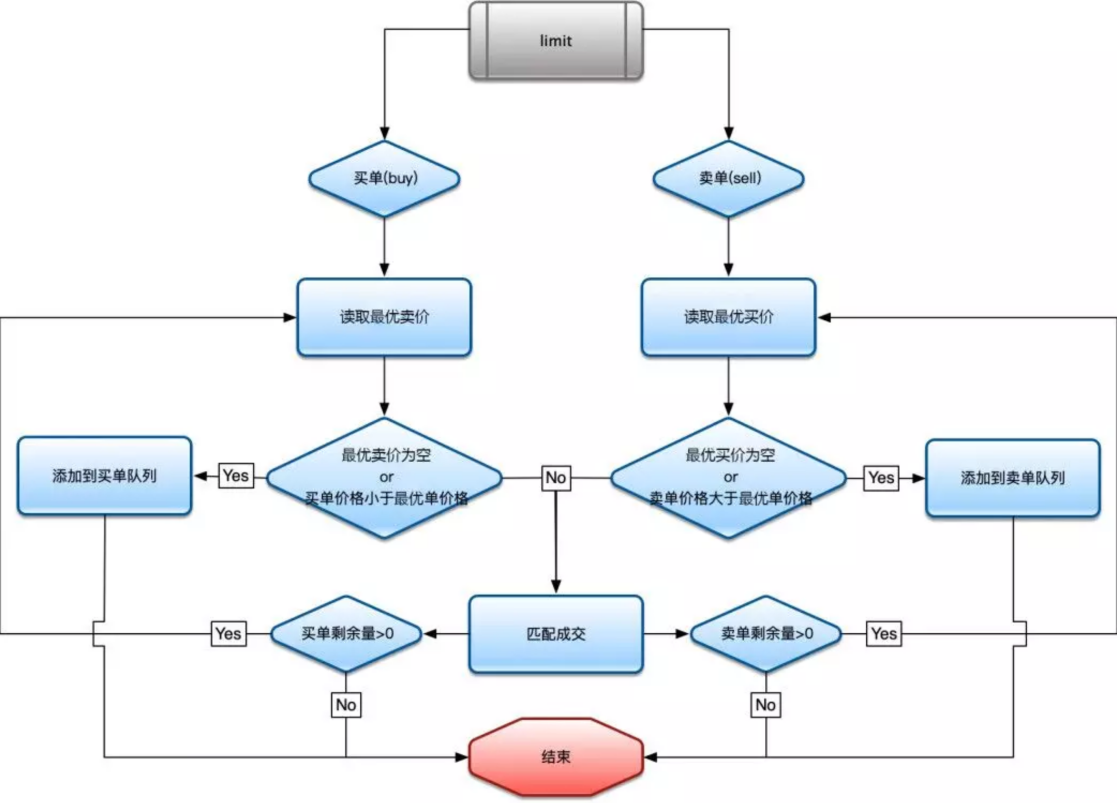


#### MessageConsumerListener:



# 完成限价撮合交易

## 9.1 交易流程



## 9.2 添加模型

### 9.2.1 成交记录模型(ExchangeTrade)



|  |
| --- |
| */\*\*  \* 成交记录 -- 只要成交一次,就产生一个记录  \*/* @Data public class ExchangeTrade {   */\*\*  \* 交易对  \*/* private String symbol;  */\*\*  \* 订单的方向  \*/* private OrderDirection direction;  */\*\*  \* 本次交易的价格  \*/* private BigDecimal price;   */\*\*  \* 本次交易的数量  \*/* private BigDecimal amount;   */\*\*  \* 本次买方的Id  \*/* private String buyOrderId;  */\*\*  \* 本次出售方的id  \*/* private String sellOrderId;  */\*\*  \* 买方的成交额  \*/* private BigDecimal buyTurnover;  */\*\*  \* 出售方的成交额  \*/* private BigDecimal sellTurnover;   */\*\*  \* 成交时间  \*/* private Long time; } |

**每次撮合完成后,都会产生一个成交记录!**

### 9.2.2 盘口数据模型/交易数据模型(TradePlate)

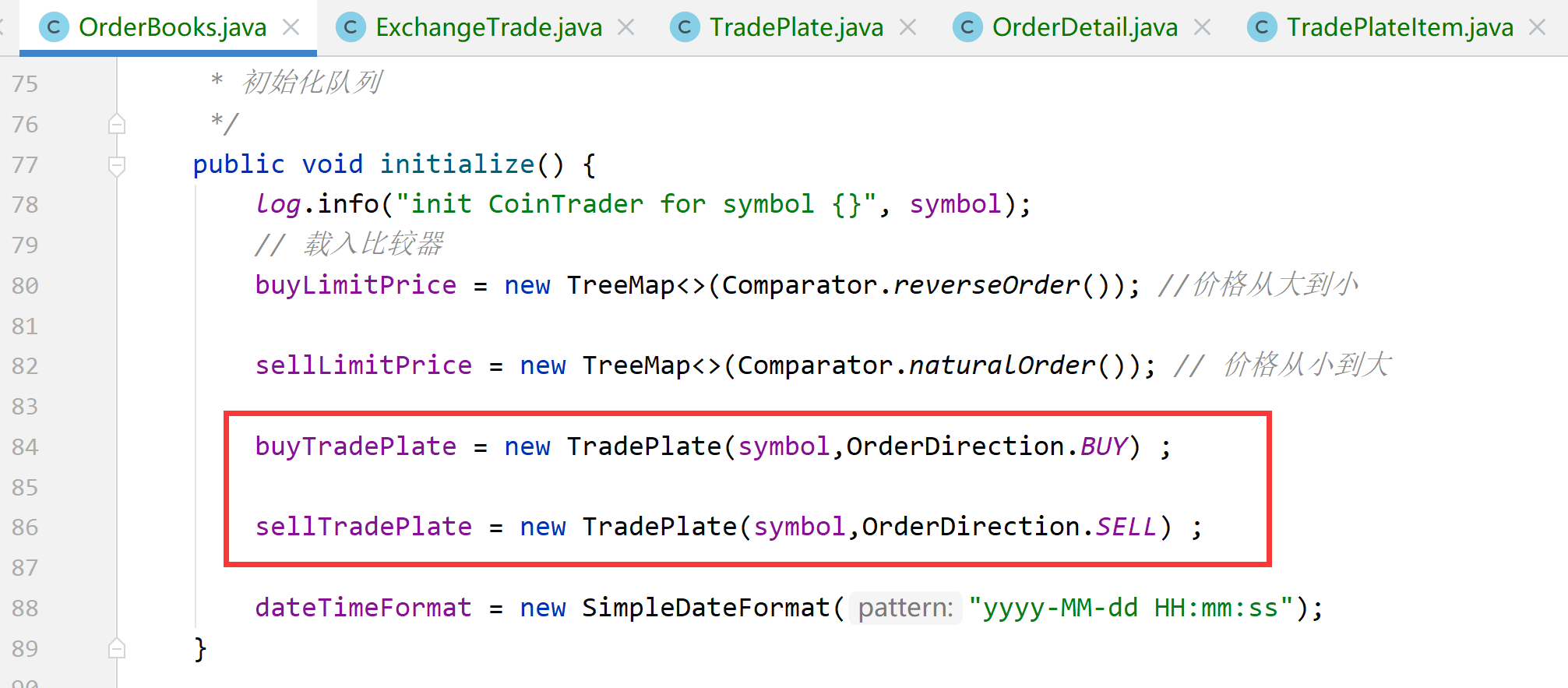


|  |
| --- |
| */\*\*  \* 交易的盘口数据,以后前端可以查询该数据  \*/* @Data public class TradePlate {   */\*\*  \* 判断数据的详情  \*/* private LinkedList<TradePlateItem> items;  */\*\*  \* 最大支持的深度  \*/* private int maxDepth = 100;   */\*\*  \* 订单的方向  \*/* private OrderDirection direction;   */\*\*  \* 交易对  \*/* private String symbol;    public void remove(Order marker, BigDecimal tradedAmount) {  } } |

|  |
| --- |
| */\*\*  \* 盘口数据的详情  \*/* @Data public class TradePlateItem {   */\*\*  \* 交易的价格  \*/* private BigDecimal price;   */\*\*  \* 交易的数量  \*/* private BigDecimal amount; } |

### 9.2.3 将盘口数据和委托单账本绑定





## 9.3 开始撮合

|  |
| --- |
| @Service @Slf4j public class LimitPriceMatchServiceImpl implements MatchService, InitializingBean {   */\*\*  \* 进行订单的撮合交易  \*  \* @param orderBooks  \* @param order  \*/* @Override  public void match(OrderBooks orderBooks, Order order) {  *log*.info("开始撮合");  *// 1 进行数据的校验* if (order.getPrice().compareTo(BigDecimal.*ZERO*) <= 0) {  return;  }  *// 2 获取一个挂单队列* Iterator<Map.Entry<BigDecimal, MergeOrder>> markerQueueIterator = null;  if (order.getOrderDirection() == OrderDirection.*BUY*) {  markerQueueIterator = orderBooks.getCurrentLimitPriceIterator(OrderDirection.*SELL*);  } else {  markerQueueIterator = orderBooks.getCurrentLimitPriceIterator(OrderDirection.*BUY*);  }    *// 是否退出循环* boolean exitLoop = false;   *// 已经完成的订单* List<Order> completedOrders = new ArrayList<>();  *// 产生的交易记录* List<ExchangeTrade> exchangeTrades = new ArrayList<>();   *// 3 循环我们的队列* while (markerQueueIterator.hasNext() && exitLoop) {  Map.Entry<BigDecimal, MergeOrder> markerOrderEntry = markerQueueIterator.next();  BigDecimal markerPrice = markerOrderEntry.getKey();  MergeOrder markerMergeOrder = markerOrderEntry.getValue();  *// 我花10 块钱买东西 ,别人的东西如果大于10 块 ,我就买不了* if (order.getOrderDirection() == OrderDirection.*BUY* && order.getPrice().compareTo(markerPrice) < 0) {  break;  }   *// 我出售一个东西 10 ,结果有个人花5块钱* if (order.getOrderDirection() == OrderDirection.*SELL* && order.getPrice().compareTo(markerPrice) > 0) {  break;  }  Iterator<Order> markerIterator = markerMergeOrder.iterator();  while (markerIterator.hasNext()) {  Order marker = markerIterator.next();   ExchangeTrade exchangeTrade = processMath(order, marker, orderBooks);  exchangeTrades.add(exchangeTrade);  if (order.isCompleted()) { *// 经过一圈的吃单,我吃饱了* completedOrders.add(order);  exitLoop = true; *// 退出最外层的循环* break; *// 退出当前的MergeOrder的循环* }   if (marker.isCompleted()) {*// MergeOrder 的一个小的订单完成了* completedOrders.add(marker);  markerIterator.remove();  }   }   if (markerMergeOrder.size() == 0) { *// MergeOrder 已经吃完了* markerQueueIterator.remove(); *// 将该MergeOrder 从树上移除掉* }   }   *// 4 若我们的订单没有完成* if (order.getAmount().compareTo(order.getTradedAmount()) > 0) {  orderBooks.addOrder(order);  }   *// 5 发送交易记录* handlerExchangeTrades(exchangeTrades);  if (completedOrders.size() > 0) {   *// 6 发送已经成交的交易记录* completedOrders(completedOrders);  *// 发送盘口数据,更新盘口* TradePlate tradePlate = order.getOrderDirection() == OrderDirection.*BUY* ?  orderBooks.getBuyTradePlate() : orderBooks.getSellTradePlate();  *// 7 发送盘口数据* sendTradePlateData(tradePlate);  }    }   */\*\*  \* 进行委托单的匹配撮合交易  \*  \* @param taker 吃单  \* @param marker 挂单  \* @return ExchangeTrade 交易记录  \*/* private ExchangeTrade processMath(Order taker, Order marker, OrderBooks orderBooks) {  *// 1 定义交易的变量  // 成交的价格* BigDecimal dealPrice = marker.getPrice();  *// 成交的数量* BigDecimal turnoverAmount = BigDecimal.*ZERO*;  *// 本次需要的数量* BigDecimal needAmount = calcTradeAmount(taker); *// 10 20  // 本次提供给你的数量* BigDecimal providerAmount = calcTradeAmount(marker); *// 20 10* turnoverAmount = needAmount.compareTo(providerAmount) <= 0 ? needAmount : providerAmount;   if (turnoverAmount.compareTo(BigDecimal.*ZERO*) == 0) {  return null; *// 无法成交* }   *// 设置本次吃单的成交数据* taker.setTradedAmount(taker.getTradedAmount().add(turnoverAmount));  BigDecimal turnoverTaker = turnoverAmount.multiply(dealPrice).setScale(orderBooks.getCoinScale(), RoundingMode.*HALF\_UP*);  taker.setTurnover(turnoverTaker);   *// 设置本次挂单的成交数据* marker.setTradedAmount(marker.getTradedAmount().add(turnoverAmount));  BigDecimal markerTurnover = turnoverAmount.multiply(dealPrice).setScale(orderBooks.getBaseCoinScale(), RoundingMode.*HALF\_UP*);  marker.setTurnover(markerTurnover);   ExchangeTrade exchangeTrade = new ExchangeTrade();   exchangeTrade.setAmount(turnoverAmount); *// 设置购买的数量* exchangeTrade.setPrice(dealPrice); *// 设置购买的价格* exchangeTrade.setTime(System.*currentTimeMillis*()); *// 设置成交的时间* exchangeTrade.setSymbol(orderBooks.getSymbol()); *// 设置成交的交易对* exchangeTrade.setDirection(taker.getOrderDirection()); *// 设置交易的方法* exchangeTrade.setSellOrderId(marker.getOrderId()); *// 设置出售方的id* exchangeTrade.setBuyOrderId(taker.getOrderId()); *// 设置买方的id* exchangeTrade.setBuyTurnover(taker.getTurnover()); *// 设置买方的交易额* exchangeTrade.setSellTurnover(marker.getTurnover()); *// 设置卖方的交易额* if (taker.getOrderDirection() == OrderDirection.*BUY*) {  orderBooks.getTradePlate().remove(marker, turnoverAmount);  } else {  orderBooks.getSellTradePlate().remove(marker, turnoverAmount);  }   return exchangeTrade;   }   */\*\*  \* 计算本次的交易额  \*  \* @param order  \* @return  \*/* private BigDecimal calcTradeAmount(Order order) {   return order.getAmount().subtract(order.getTradedAmount());   }   */\*\*  \* 发送盘口数据,供以后我们前端的数据更新  \*  \* @param tradePlate  \*/* private void sendTradePlateData(TradePlate tradePlate) {  }   */\*\*\*  \* 订单的完成  \* @param completedOrders  \*/* private void completedOrders(List<Order> completedOrders) {  System.*out*.println("本次完成的订单文");  }   */\*\*  \* 处理订单的记录  \*  \* @param exchangeTrades  \*/* private void handlerExchangeTrades(List<ExchangeTrade> exchangeTrades) {   System.*out*.println("交易的订单为:"+exchangeTrades);  }    @Override  public void afterPropertiesSet() throws Exception {   MatchServiceFactory.*addMatchService*(MatchStrategy.*LIMIT\_PRICE*, this);  } } |

## 9.4 订单是否完成的判断依据



|  |
| --- |
| */\*\*  \* 委托单  \*/* @Data @NoArgsConstructor public class Order implements Serializable {   */\*\*  \* 本次订单的Id  \*/* private String orderId;   */\*\*  \* 用户/会员Id  \*/* private Long userId;   */\*\*  \* 支持的币币交易对  \*/* private String symbol;   */\*\*  \* 买入或卖出量  \*/* private BigDecimal amount = BigDecimal.*ZERO*;   */\*\*  \* 成交量  \*/* private BigDecimal tradedAmount = BigDecimal.*ZERO*;   */\*\*  \* 成交额  \*/* private BigDecimal turnover = BigDecimal.*ZERO*;   */\*\*  \* 币单位  \*/* private String coinSymbol;   */\*\*  \* 结算单位  \*/* private String baseSymbol;   */\*\*  \* 订单状态  \*/* private Integer orderStatus;   */\*\*  \* 订单的方向  \*/* private OrderDirection orderDirection;   */\*\*  \* 挂单的价格  \*/* private BigDecimal price = BigDecimal.*ZERO*;   */\*\*  \* 挂单时间  \*/* private Long time;   */\*\*  \* 交易完成时间  \*/* private Long completedTime;   */\*\*  \* 交易取消时间  \*/* private Long cancelTime;  *// /\*\* // \* 已经成功的水平订单 // \*/ // private List<OrderDetail> details;    /\*\*  \* 订单是否完成的判断依据  \*  \* @return  \*/* public boolean isCompleted() {  return amount.compareTo(tradedAmount) <= 0;  } } |

## 9.5 委托单新增数据或删除数据时更新盘口

在OrderBooks 里面:

|  |
| --- |
| */\*\*  \* 怎么添加一个订单进入我们的队列里面  \*/* public void addOrder(Order order) {  TreeMap<BigDecimal, MergeOrder> currentLimitPrices = getCurrentLimitPrices(order.getOrderDirection());  MergeOrder mergeOrder = currentLimitPrices.get(order.getPrice());  *//* if (mergeOrder == null) { *// 之前在红黑树里面没有这个价格的key* mergeOrder = new MergeOrder();  currentLimitPrices.put(order.getPrice(), mergeOrder);  }  *//* mergeOrder.add(order);    ***// 添加到盘点里面* if (order.getOrderDirection() == OrderDirection.*BUY*) {  buyTradePlate.add(order);  } else {  sellTradePlate.add(order);  }**  }  */\*\*  \* 怎么取消一个订单  \*/* public void cancelOrder(Order order) {  TreeMap<BigDecimal, MergeOrder> currentLimitPrices = getCurrentLimitPrices(order.getOrderDirection());  MergeOrder mergeOrder = currentLimitPrices.get(order);  if (mergeOrder == null) {  return;  }  Iterator<Order> iterator = mergeOrder.iterator();  while (iterator.hasNext()) {  Order next = iterator.next();  *// 找到之前的我们的订单记录* if (next.getOrderId().equals(order.getOrderId())) {  iterator.remove();  }  }  int size = mergeOrder.size(); *// 删除之前,我们看合并订单的大小* if (size == 0) { *// 若我们的红黑树里面的合并订单的数据为空,我们摘除* currentLimitPrices.remove(order.getPrice());  }   ***// 添加到盘点里面* if (order.getOrderDirection() == OrderDirection.*BUY*) {  buyTradePlate.remove(order);  } else {  sellTradePlate.remove(order);  }** } |

## 9.6 盘口数据的添加和删除

|  |
| --- |
| */\*\*  \* 添加订单数据到盘口数据里面  \* 当我们新增一个委托单时: 它没有全部的成交 ,因此把它展示在盘口里面  \*  \* @param order  \*/* public void add(Order order) {  if (order.getOrderDirection() != direction) {  return;  }  int i = 0;  for (i = 0; i < items.size(); i++) {  *// 1 我们的sell 队列是: 从小到大  // 2 我们的buy 队列是: 从大到小* DepthItemVo depthItemVo = items.get(i);  if (  (direction == OrderDirection.*BUY* && order.getPrice().compareTo(depthItemVo.getPrice()) == -1)  ||  (direction == OrderDirection.*SELL* && order.getPrice().compareTo(depthItemVo.getPrice()) == 1)   ) {  *// 还不能插入,往前走一步* continue;  } else if (depthItemVo.getPrice().compareTo(order.getPrice()) == 0) {  depthItemVo.setVolume(depthItemVo.getVolume().add(order.getAmount().subtract(order.getTradedAmount())));  return;  } else {  break; *// 我就想插入 当前我就在第 i* }  }   if (i < maxDepth) {  DepthItemVo depthItemVo = new DepthItemVo();  depthItemVo.setPrice(order.getPrice());  depthItemVo.setVolume(order.getAmount().subtract(order.getTradedAmount()));  items.add(i, depthItemVo);  } }   */\*\*  \* 从盘口里面移除订单  \*  \* @param order  \*/* public void remove(Order order) {  *// order.getAmount().subtract(order.getTradedAmount() 成交的数量* remove(order, order.getAmount().subtract(order.getTradedAmount())); }  */\*\*\*  \* 从盘口里面移除数据  \* @param order  \* @param amount  \*/* public void remove(Order order, BigDecimal amount) {   if (items.size() == 0) {  return;  }  if (order.getOrderDirection() != direction) {  return;  }  Iterator<DepthItemVo> iterator = items.iterator();  while (iterator.hasNext()) {  DepthItemVo next = iterator.next();  if (order.getPrice().compareTo(next.getPrice()) == 0) { *// 价格相同* next.setVolume(next.getVolume().subtract(amount));  if (next.getVolume().compareTo(BigDecimal.*ZERO*) <= 0) {  iterator.remove(); *// 若价格为 0 后,我们直接可以摘掉它* }  }   }  } |

## 9.7 处理数据的发送

### 9.7.1 添加Source

|  |
| --- |
| public interface Source {    */\*\*  \* 盘口数据的输出  \* @return  \*/* @Output("trade\_plate\_out")  MessageChannel plateOut() ;    */\*\*  \* 完成订单数据的输出  \* @return  \*/* @Output("completed\_orders\_out")  MessageChannel completedOrdersOut() ;    */\*\*  \* 交易记录的输入  \* @return  \*/* @Output("exchange\_trades\_out")  MessageChannel exchangeTradesOut() ; } |

### 9.7.2 添加配置文件

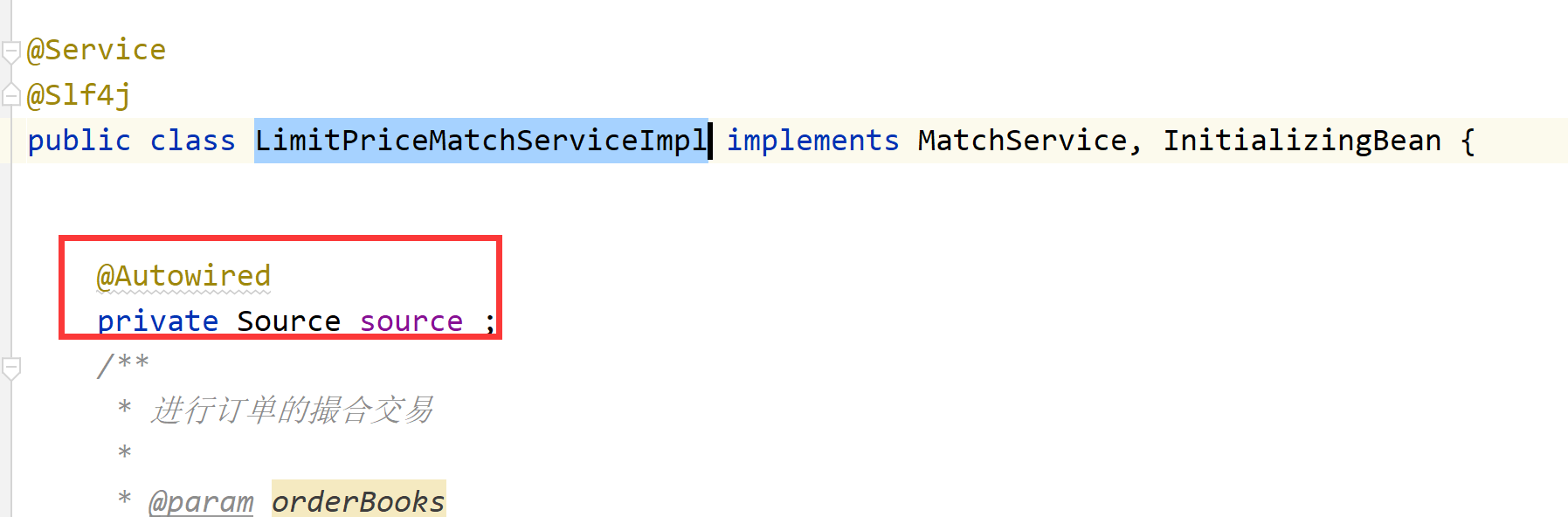
|  |
| --- |
|  |

### 9.7.3 Source注入

|  |
| --- |
| */\*\*  \* 开启我们的Stream的开发  \*/* @Configuration @EnableBinding(value = {Sink.class,Source.class}) *//* public class RocketStreamConfig { } |

### 9.7.3 数据的发送

在LimitPriceMatchServiceImpl 里面:



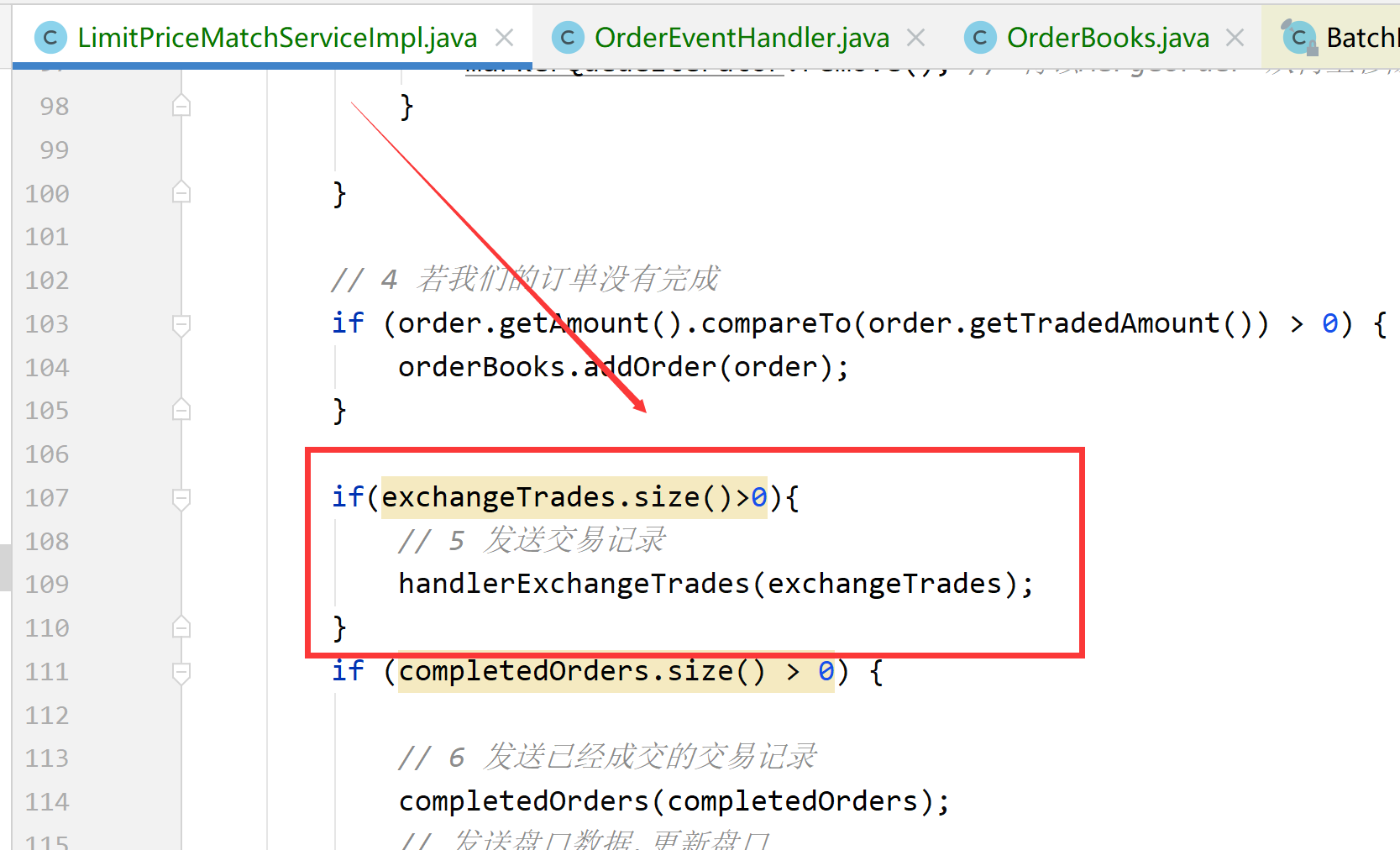
|  |
| --- |
| */\*\*  \* 发送盘口数据,供以后我们前端的数据更新  \*  \* @param tradePlate  \*/* private void sendTradePlateData(TradePlate tradePlate) {  Message<TradePlate> message = MessageBuilder  .*withPayload*(tradePlate)  .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*)  .build();  source.plateOut().send(message) ; }  */\*\*\*  \* 订单的完成  \* @param completedOrders  \*/* private void completedOrders(List<Order> completedOrders) {   Message<List<Order>> message = MessageBuilder  .*withPayload*(completedOrders)  .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*)  .build();  source.completedOrdersOut().send(message) ; }  */\*\*  \* 处理订单的记录  \*  \* @param exchangeTrades  \*/* private void handlerExchangeTrades(List<ExchangeTrade> exchangeTrades) {   Message<List<ExchangeTrade>> message = MessageBuilder  .*withPayload*(exchangeTrades)  .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*)  .build();  source.exchangeTradesOut().send(message) ;  } |

# Bug 解决

## 10.1 盘口数据的初始化



## 10.2 发送一个Null的集合



# 完成盘口数据的查询操作

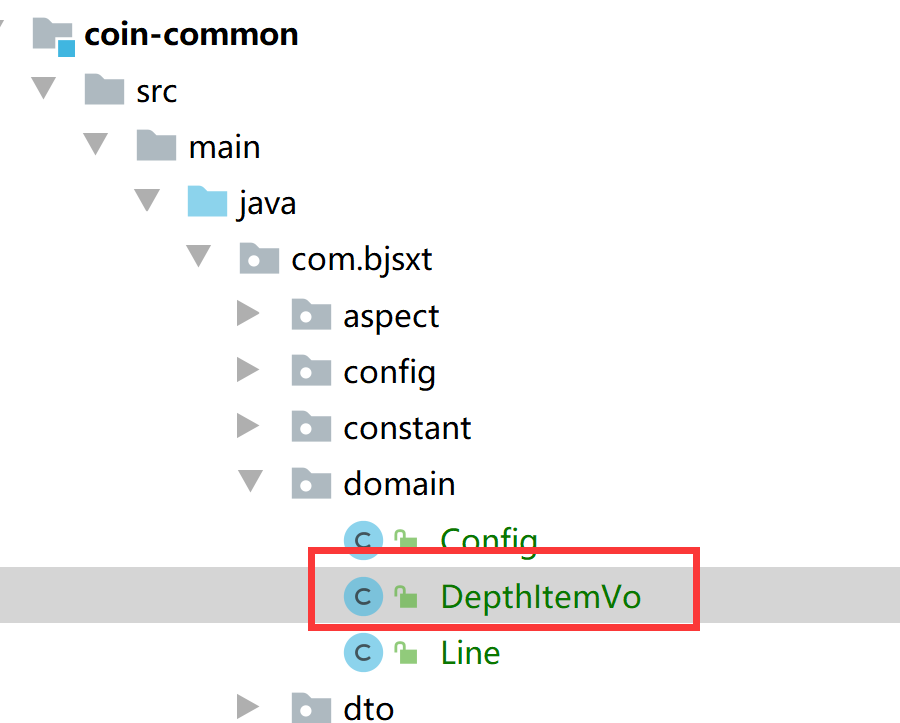
## 10.1 修改盘口数据的模型

将之前的:TradePlateItem 修改为我们之前在exchange-service 里面构造的

DepthItemVo 对象



将DepthItemVo 提取到common 包里面:

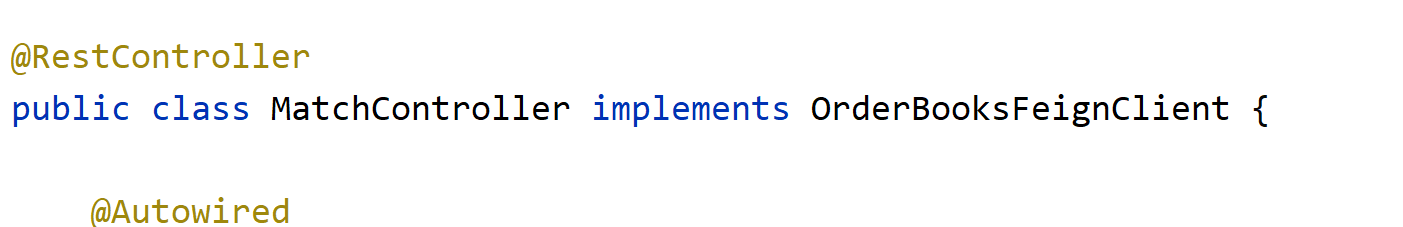


这样,他们就能共享了这个公共的类了.

## 10.2 在match-api 里面添加feign客户端

|  |
| --- |
| @FeignClient(name = "match-service", contextId = "orderBooksFeignClient", configuration = OAuth2FeignConfig.class) public interface OrderBooksFeignClient {    */\*\*  \* 远程调用深度数据  \* @param symbol  \* @return  \*/* @GetMapping("/match/depth")  Map<String, List<DepthItemVo>> getDepth(@RequestParam(required = true) String symbol) ; } |

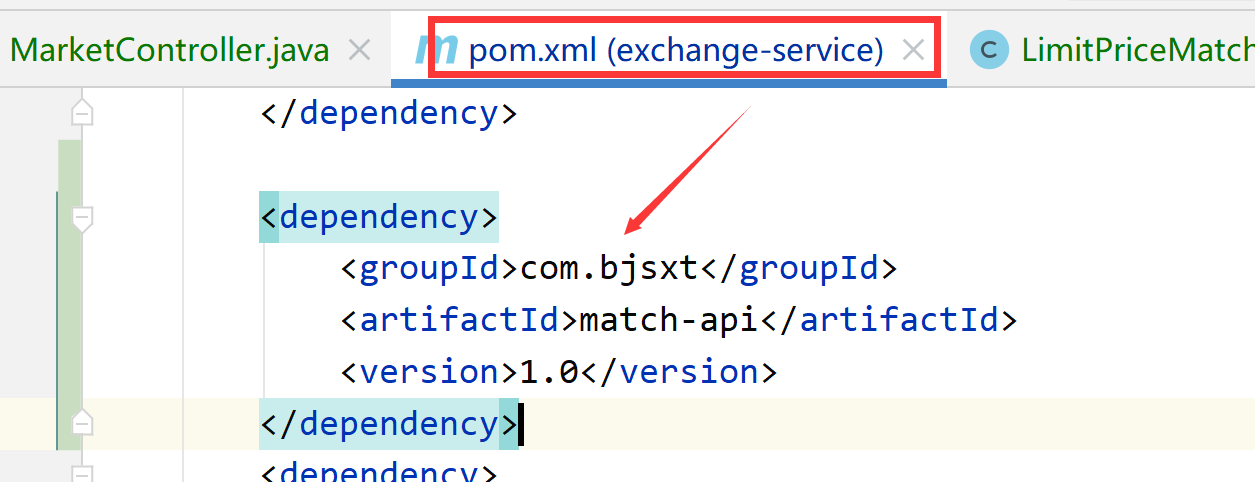
## 10.3 在match-service的MatchController里面实现它



|  |
| --- |
| public Map<String, List<DepthItemVo>> getDepth(@RequestParam(required = true) String symbol) {  Map<String, List<DepthItemVo>> depths = new HashMap<>();  for (EventHandler<OrderEvent> eventHandler : eventHandlers) {  OrderEventHandler orderEventHandler = (OrderEventHandler) eventHandler;  *// 找到对应的深度数据* if (orderEventHandler.getSymbol().equals(symbol)) {  OrderBooks orderBooks = orderEventHandler.getOrderBooks();  TradePlate buyTradePlate = orderBooks.getBuyTradePlate();  TradePlate sellTradePlate = orderBooks.getSellTradePlate();  depths.put("bids", buyTradePlate.getItems());  depths.put("asks", sellTradePlate.getItems());  return depths;  }  }  return null; } |

## 10.4 在MarketController修改findDeptVosSymbol方法

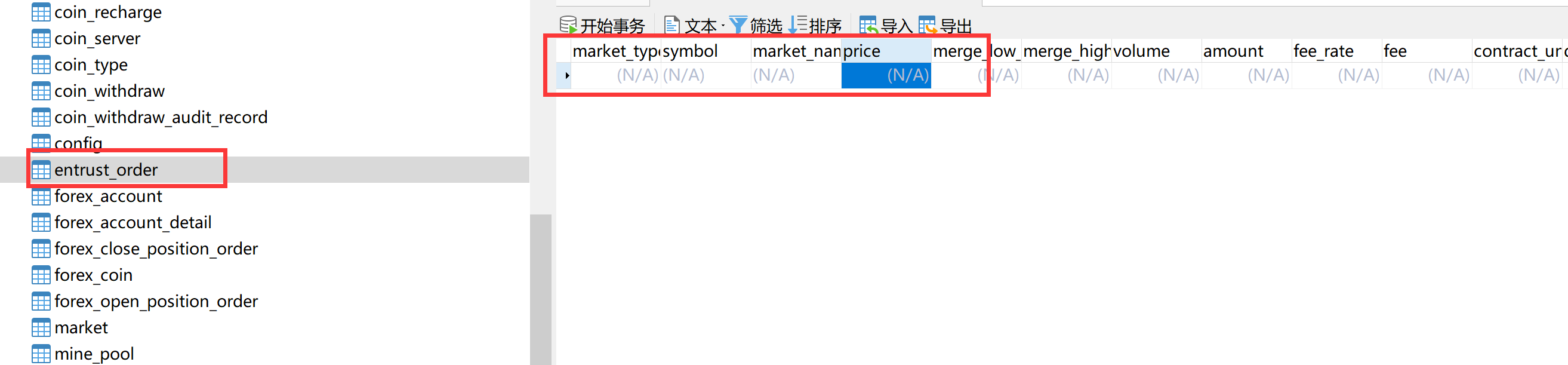
Tip: 在exchange-service 里面添加match-api的依赖:



|  |
| --- |
| @ApiOperation(value = "通过的交易对以及深度查询当前的市场的深度数据") @GetMapping("/depth/{symbol}/{dept}") @ApiImplicitParams({  @ApiImplicitParam(name = "symbol", value = "交易对"),  @ApiImplicitParam(name = "dept", value = "深度类型"), }) public R<DepthsVo> findDeptVosSymbol(@PathVariable("symbol") String symbol, String dept) {  *// 交易市场* Market market = marketService.getMarkerBySymbol(symbol);   DepthsVo depthsVo = new DepthsVo();  depthsVo.setCnyPrice(market.getOpenPrice()); *// CNY的价格* depthsVo.setPrice(market.getOpenPrice()); *// GCN的价格* Map<String, List<DepthItemVo>> depthMap = orderBooksFeignClient.getDepth(symbol);  if (!CollectionUtils.*isEmpty*(depthMap)) {  depthsVo.setAsks(depthMap.get("asks"));  depthsVo.setBids(depthMap.get("bids"));  }  return R.*ok*(depthsVo);  } |

## 10.5 添加委托单演示

测试之前,先清除数据库的的这个数据:

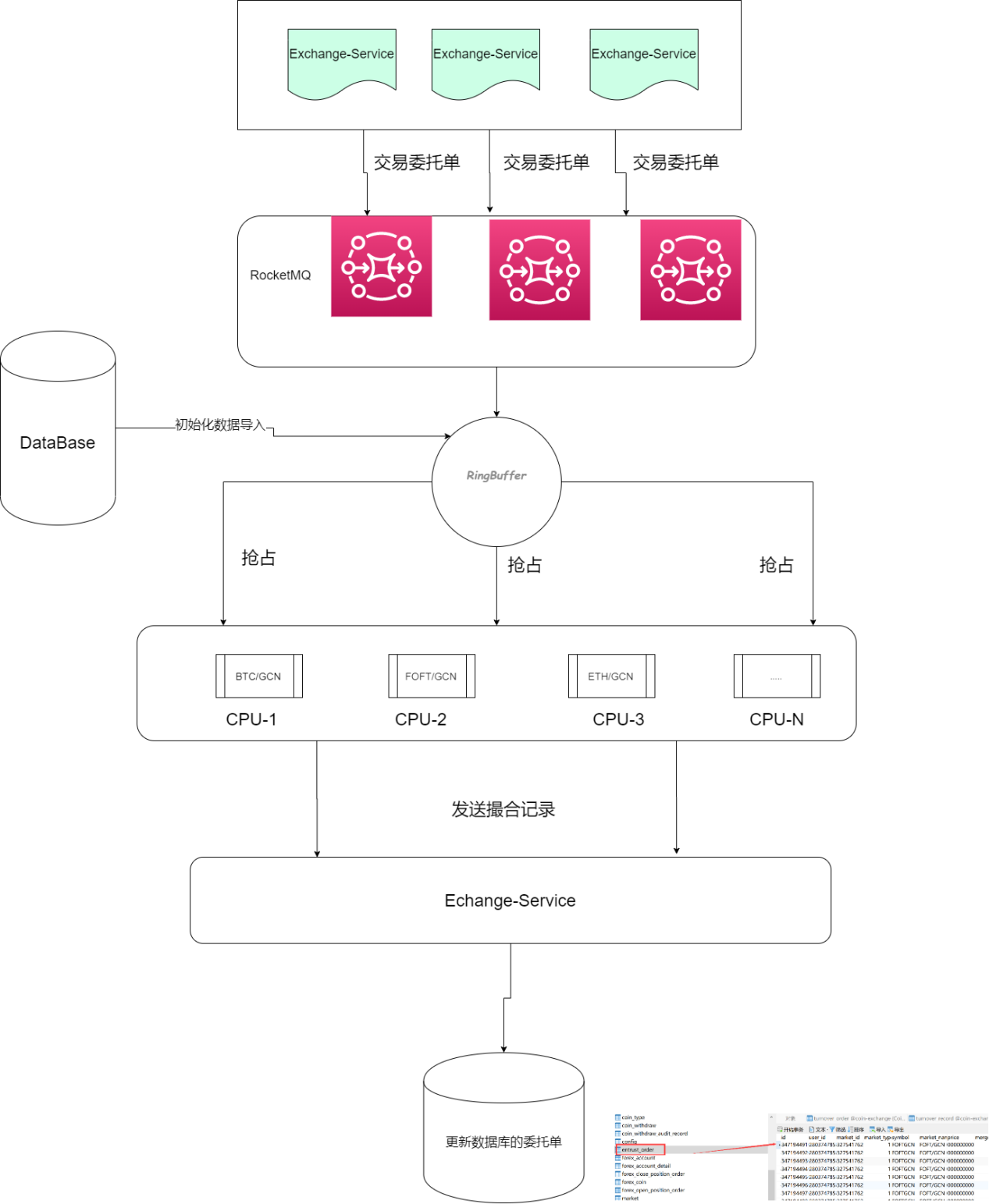


可以自己添加和删除测试:



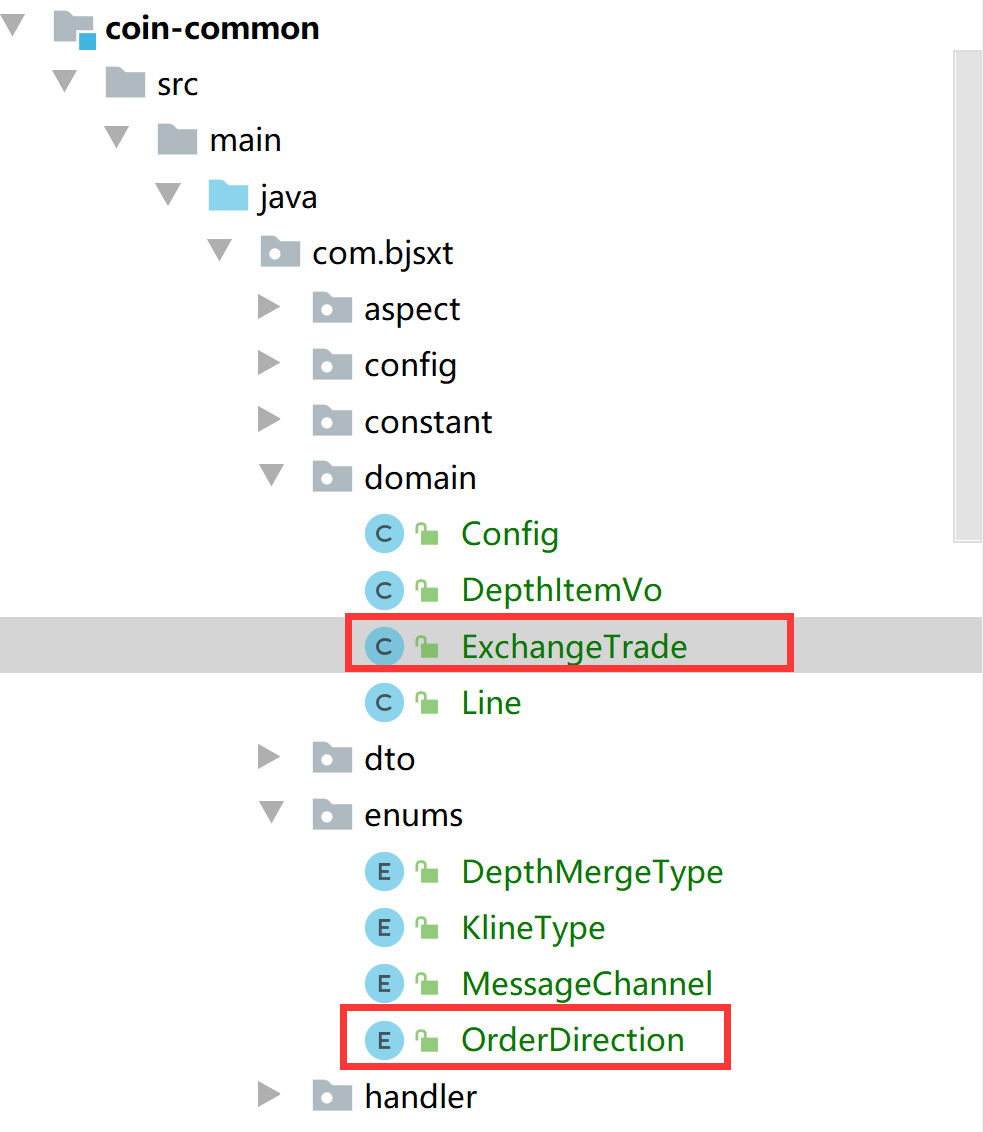
# 撮合完成

## 12.1 撮合完成需要更新数据库的值



## 12.2 公共类的提取

将ExchangeTrade 和OrderDirection 提取到common模块



## 12.3 使用Rocket接收撮合数据

### 12.3.1 Sink

|  |
| --- |
| */\*\*  \* 数据的接收  \*/* public interface Sink {   */\*\*  \* 交易数据的输入  \* @return  \*/* @Input("exchange\_trade\_in")  MessageChannel exchangeTradeIn() ; } |

### 12.3.2 监听器

|  |
| --- |
| */\*\*\*  \* 交易数据的监听  \*/* @Component @Slf4j public class ExchangeTradeListener {   @Autowired  private EntrustOrderService entrustOrderService ;   @Transactional  @StreamListener("exchange\_trade\_in")  public void receiveExchangeTrade(List<ExchangeTrade> exchangeTrades){  if (CollectionUtils.*isEmpty*(exchangeTrades)){  return;  }  for (ExchangeTrade exchangeTrade : exchangeTrades) {  if(exchangeTrade!=null){  *// 交易完成后,去更新我们的数据库* entrustOrderService.doMatch(exchangeTrade) ;  }  }   } } |

### 12.3.3 EntrustOrderService接口

|  |
| --- |
| */\*\*  \* 更新我们的委托单的数据  \* @param exchangeTrade  \*/* void doMatch(ExchangeTrade exchangeTrade); |

### 12.3.4 EntrustOrderServiceImpl

|  |
| --- |
| */\*\*  \* 更新我们的委托单的数据  \*  \* @param exchangeTrade  \*/* @Override @Transactional public void doMatch(ExchangeTrade exchangeTrade) {  String sellOrderId = exchangeTrade.getSellOrderId();  String buyOrderId = exchangeTrade.getBuyOrderId();  EntrustOrder sellOrder = getById(sellOrderId);  EntrustOrder buyOrder = getById(buyOrderId);  Long marketId = sellOrder.getMarketId();  Market market = marketService.getById(marketId);    *// 1 新增成交记录* addTurnOverOrderRecord(sellOrder, buyOrder, market, exchangeTrade);  *// 2 更新委托单* updateEntrustOrder(sellOrder, buyOrder, exchangeTrade);  *// 3 余额的返还* rollBackAccount(sellOrder,buyOrder,exchangeTrade,market); }   */\*\*  \* 添加成交记录  \*  \* @param  \*/* private void addTurnOverOrderRecord(EntrustOrder sellOrder, EntrustOrder buyOrder, Market market, ExchangeTrade exchangeTrade) {    *// 出售订单的成交记录* TurnoverOrder sellTurnoverOrder = new TurnoverOrder();  sellTurnoverOrder.setSellOrderId(sellOrder.getId());  sellTurnoverOrder.setBuyCoinId(buyOrder.getId());  sellTurnoverOrder.setBuyVolume(exchangeTrade.getAmount());  sellTurnoverOrder.setAmount(exchangeTrade.getSellTurnover());   sellTurnoverOrder.setBuyCoinId(market.getBuyCoinId());  sellTurnoverOrder.setSellCoinId(market.getSellCoinId());  sellTurnoverOrder.setCreated(new Date());  sellTurnoverOrder.setBuyUserId(buyOrder.getUserId());  sellTurnoverOrder.setSellUserId(sellOrder.getUserId());  sellTurnoverOrder.setPrice(exchangeTrade.getPrice()) ;  sellTurnoverOrder.setBuyPrice(buyOrder.getPrice()) ;  sellTurnoverOrder.setTradeType(2);  turnoverOrderService.save(sellTurnoverOrder);   *// 买方数据的成交记录* TurnoverOrder buyTurnoverOrder = new TurnoverOrder();  buyTurnoverOrder.setBuyOrderId(buyOrder.getId());  buyTurnoverOrder.setSellOrderId(sellOrder.getId());  buyTurnoverOrder.setAmount(exchangeTrade.getBuyTurnover());  buyTurnoverOrder.setBuyVolume(exchangeTrade.getAmount());  buyTurnoverOrder.setSellUserId(sellOrder.getUserId());  buyTurnoverOrder.setBuyUserId(buyOrder.getUserId());  buyTurnoverOrder.setSellCoinId(market.getSellCoinId());  buyTurnoverOrder.setBuyCoinId(market.getBuyCoinId());  buyTurnoverOrder.setCreated(new Date());  sellTurnoverOrder.setTradeType(1);  turnoverOrderService.save(sellTurnoverOrder); }  */\*\*  \* 更新委托单记录  \*  \* @param exchangeTrade  \*/* private void updateEntrustOrder(EntrustOrder sellOrder, EntrustOrder buyOrder, ExchangeTrade exchangeTrade) {   */\*\*  \* 已经成交的数量  \*/* sellOrder.setDeal(exchangeTrade.getAmount());  buyOrder.setDeal(exchangeTrade.getAmount());  BigDecimal volume = sellOrder.getVolume(); *// 总的数量* BigDecimal amount = exchangeTrade.getAmount(); *// 本次成交的数量* if (amount.compareTo(volume) == 0) { *// 交易完成  // 状态(已经完成)* sellOrder.setStatus((byte) 1);  }  BigDecimal buyOrderVolume = buyOrder.getVolume();  if (buyOrderVolume.compareTo(volume) == 0) { *// 交易完成  // 状态(已经完成)* buyOrder.setStatus((byte) 1);  }   *// 更新委托单* updateById(sellOrder);  updateById(buyOrder); }   */\*\*  \* 返回账户的余额  \*  \* @param exchangeTrade  \*/* private void rollBackAccount(EntrustOrder sellOrder, EntrustOrder buyOrder, ExchangeTrade exchangeTrade, Market market) {   *// 买单需要返还用户的余额,之前扣减的余额* accountServiceFeign.transferBuyAmount(buyOrder.getUserId(), *// 买单用户ID* sellOrder.getUserId(), *// 卖单用户ID* market.getBuyCoinId(), *// 买单支付币种* exchangeTrade.getBuyTurnover(), *// 买单成交金额* "币币交易",  Long.*valueOf*(exchangeTrade.getBuyOrderId()));   *// 出售单需要* accountServiceFeign.transferSellAmount(sellOrder.getUserId(), *// 卖单用户ID* sellOrder.getUserId(), *// 买单用户ID* market.getSellCoinId(), *// 卖单支付币种* exchangeTrade.getSellTurnover(), *// 卖单成交数量* "币币交易", *// 业务类型：币币交易撮合成交* Long.*valueOf*( exchangeTrade.getSellOrderId())); *// 成交订单ID* } |

### 12.3.5 AccountServiceFeign

|  |
| --- |
| */\*\*  \* 划转买入的账户余额  \* @param fromUserId  \* @param toUserId  \* @param coinId  \* @param amount  \* @param businessType  \* @param orderId  \*/* @PostMapping("/transferBuyAmount") void transferBuyAmount(@RequestParam("fromUserId") Long fromUserId, @RequestParam("toUserId")Long toUserId, @RequestParam("coinId")Long coinId, @RequestParam("amount")BigDecimal amount,@RequestParam("businessType")String businessType, @RequestParam("orderId")Long orderId);  */\*\*  \* 划转出售的成功的账户余额  \* @param fromUserId  \* @param toUserId  \* @param coinId  \* @param amount  \* @param businessType  \* @param orderId  \*/* @PostMapping("/transferSellAmount") void transferSellAmount(@RequestParam("fromUserId") Long fromUserId, @RequestParam("toUserId")Long toUserId, @RequestParam("coinId")Long coinId, @RequestParam("amount")BigDecimal amount,@RequestParam("businessType")String businessType, @RequestParam("orderId")Long orderId); |

### 12.3.6 AccountController

|  |
| --- |
| */\*\*  \* 划转买入的账户余额  \*  \* @param fromUserId  \* @param toUserId  \* @param coinId  \* @param amount  \* @param businessType  \* @param orderId  \*/* @Override public void transferBuyAmount(Long fromUserId, Long toUserId, Long coinId, BigDecimal amount, String businessType, Long orderId) {  accountService.transferBuyAmount(fromUserId, toUserId, coinId, amount, businessType, orderId); }  */\*\*  \* 划转出售的成功的账户余额  \*  \* @param fromUserId  \* @param toUserId  \* @param coinId  \* @param amount  \* @param businessType  \* @param orderId  \*/* @Override public void transferSellAmount(Long fromUserId, Long toUserId, Long coinId, BigDecimal amount, String businessType, Long orderId) {  accountService.transferSellAmount(fromUserId, toUserId, coinId, amount, businessType, orderId); } |

### 12.3.7 AccountService

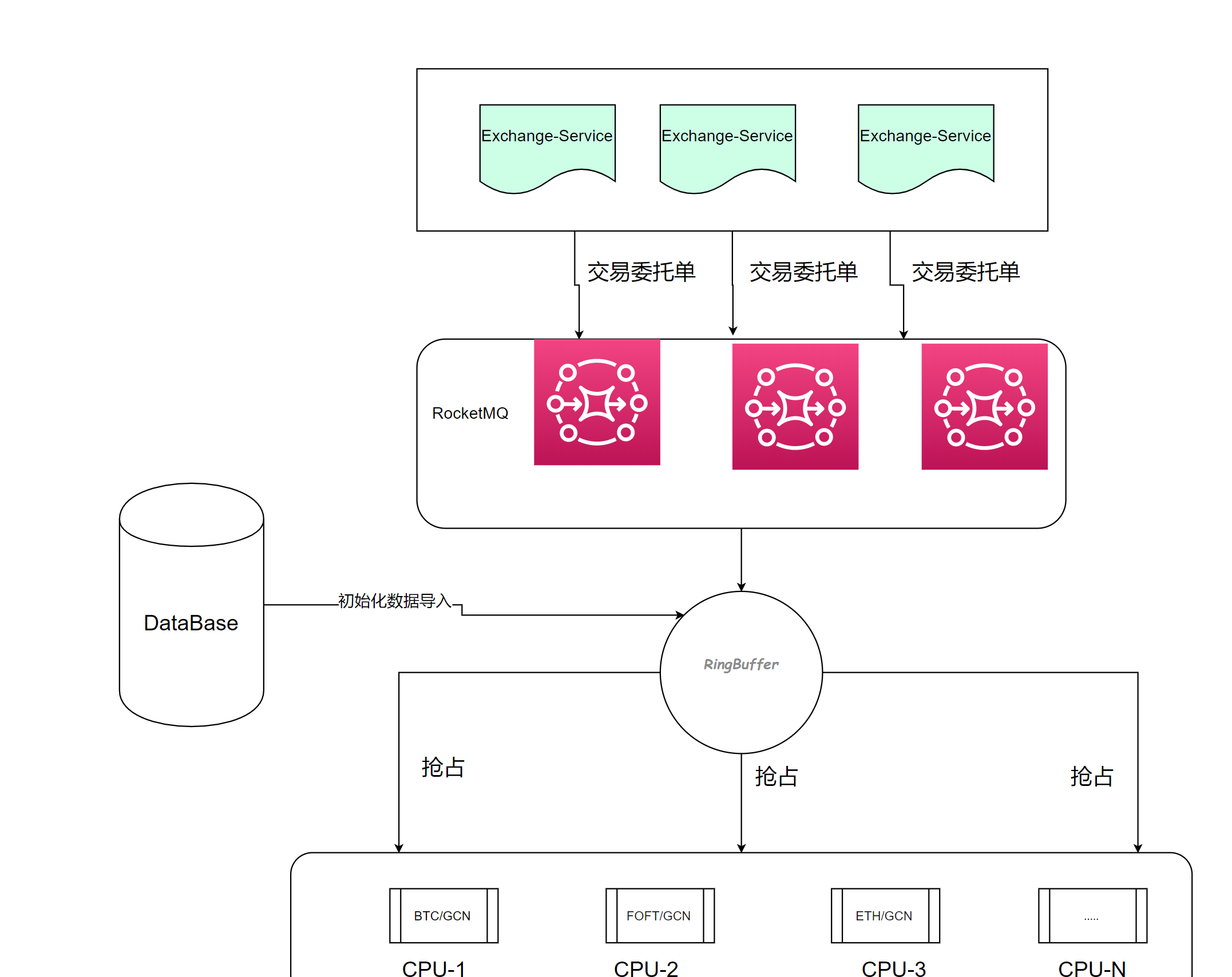
|  |
| --- |
| */\*\*  \*  \* @param fromUserId  \* @param toUserId  \* @param coinId  \* @param amount  \* @param businessType  \* @param orderId  \*/* void transferBuyAmount(Long fromUserId, Long toUserId, Long coinId, BigDecimal amount, String businessType, Long orderId);  */\*\*  \*  \* @param fromUserId  \* @param toUserId  \* @param coinId  \* @param amount  \* @param businessType  \* @param orderId  \*/* void transferSellAmount(Long fromUserId, Long toUserId, Long coinId, BigDecimal amount, String businessType, Long orderId); |

### 12.3.8 AccountServiceImpl

|  |
| --- |
| @Override public void transferBuyAmount(Long fromUserId, Long toUserId, Long coinId, BigDecimal amount, String businessType, Long orderId) {  Account fromAccount = getCoinAccount(coinId, fromUserId);  if (fromAccount == null) {  *log*.error("资金划转-资金账户异常，userId:{}, coinId:{}", fromUserId, coinId);  throw new IllegalArgumentException("资金账户异常");  } else {  Account toAccount = getCoinAccount(toUserId, coinId);  if (toAccount == null) {  throw new IllegalArgumentException("资金账户异常");  } else {  boolean count1 = decreaseAmount(fromAccount, amount);  boolean count2 = addAmount(toAccount, amount);  if (count1 && count2) {  List<AccountDetail> accountDetails = new ArrayList(2);  AccountDetail fromAccountDetail = new AccountDetail(fromUserId, coinId, fromAccount.getId(), toAccount.getId(), orderId, 2, businessType, amount, BigDecimal.*ZERO*, businessType);  AccountDetail toAccountDetail = new AccountDetail(toUserId, coinId, toAccount.getId(), fromAccount.getId(), orderId, 1, businessType, amount, BigDecimal.*ZERO*, businessType);  accountDetails.add(fromAccountDetail);  accountDetails.add(toAccountDetail);   accountDetails.addAll(accountDetails);  } else {  throw new RuntimeException("资金划转失败");  }  }  } }  private boolean addAmount(Account account, BigDecimal amount) {  account.setBalanceAmount(account.getBalanceAmount().add(amount));  return updateById(account); }  private boolean decreaseAmount(Account account, BigDecimal amount) {  account.setBalanceAmount(account.getBalanceAmount().subtract(amount));  return updateById(account); }  @Override public void transferSellAmount(Long fromUserId, Long toUserId, Long coinId, BigDecimal amount, String businessType, Long orderId) {  Account fromAccount = getCoinAccount(coinId, fromUserId);  if (fromAccount == null) {  *log*.error("资金划转-资金账户异常，userId:{}, coinId:{}", fromUserId, coinId);  throw new IllegalArgumentException("资金账户异常");  } else {  Account toAccount = getCoinAccount(toUserId, coinId);  if (toAccount == null) {  throw new IllegalArgumentException("资金账户异常");  } else {  boolean count1 = addAmount(fromAccount, amount);  boolean count2 = decreaseAmount(toAccount, amount);  if (count1 && count2) {  List<AccountDetail> accountDetails = new ArrayList(2);  AccountDetail fromAccountDetail = new AccountDetail(fromUserId, coinId, fromAccount.getId(), toAccount.getId(), orderId, 2, businessType, amount, BigDecimal.*ZERO*, businessType);  AccountDetail toAccountDetail = new AccountDetail(toUserId, coinId, toAccount.getId(), fromAccount.getId(), orderId, 1, businessType, amount, BigDecimal.*ZERO*, businessType);  accountDetails.add(fromAccountDetail);  accountDetails.add(toAccountDetail);   accountDetails.addAll(accountDetails);  } else {  throw new RuntimeException("资金划转失败");  }  }  } } |

# 订单的取消

## 13.1 取消思路



以上的过程都是单线程的,但是因为添加取消后,情况可能会发生变化.

## 13.2 取消的实现(思路)

### 13.2.1 接收取消的请求

**EntrustOrderController:**

|  |
| --- |
| @ApiOperation(value = "委托单的取消操作") @ApiImplicitParams({  @ApiImplicitParam(name = "id" ,value = "委托单的id") }) @DeleteMapping("/{id}") public R deleteEntrustOrder(@PathVariable("id") Long orderId){  entrustOrderService.cancleEntrustOrder(orderId) ;  return R.*ok*("取消成功") ; } |

**EntrustOrderService:**

|  |
| --- |
|  |

**EntrustOrderServiceImpl:**

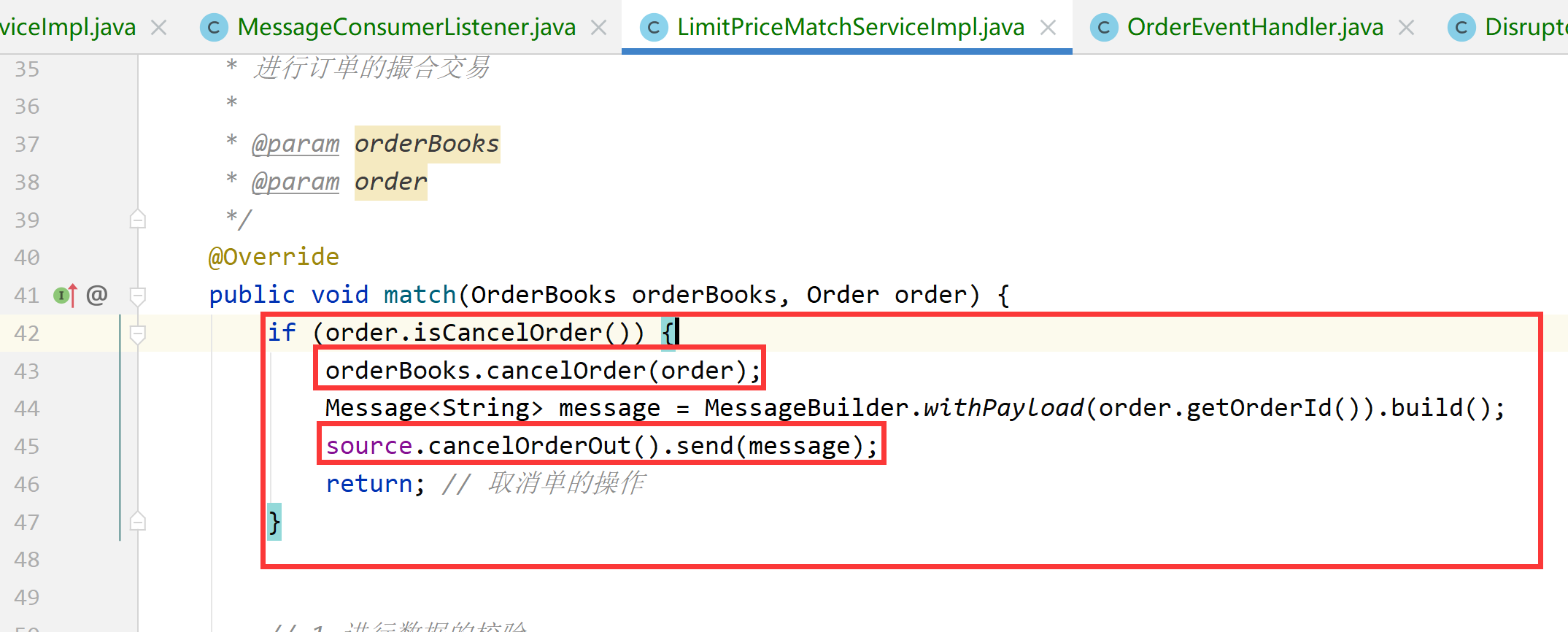
|  |
| --- |
| @Override public void cancleEntrustOrder(Long orderId) {  *// 取消委托单  // 1 将该委托单从撮合引擎里面的委托单账本里面移除* EntrustOrder entrustOrder = new EntrustOrder();  entrustOrder.setStatus((byte) 2);  entrustOrder.setId(orderId);  Message<EntrustOrder> message = MessageBuilder.*withPayload*(entrustOrder).setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*).build();  source.outputMessage().send(message);  } |

### 13.2.2 从委托单账本里面移除

在撮合系统里面的MessageConsumerListener处理该消息:



从账单本上取消:



### 13.2.3 移除成功后,更新数据库



监听取消队列的消息:ExchangeTradeListener:

|  |
| --- |
| @Transactional @StreamListener("cancel\_order\_in") public void receiveCancelOrder(String orderId){  entrustOrderService.cancleEntrustOrderToDb(orderId); } |

### 13.2.4 执行数据库的取消操作

**EntrustOrderService:**

|  |
| --- |
| */\*\*  \* 数据库里面委托单的取消  \* @param orderId  \*/* void cancleEntrustOrderToDb(String orderId); |

**EntrustOrderServiceImpl:**

|  |
| --- |
| */\*\*  \* 数据库里面委托单的取消  \*  \* @param orderId  \*/* @Override public void cancleEntrustOrderToDb(String orderId) {  *// 2 数据库的操作* if (StringUtils.*hasText*(orderId)) {  Long orderIdVal = Long.*valueOf*(orderId);  EntrustOrder entrustOrder = getById(orderId);  entrustOrder.setStatus((byte) 2);  updateById(entrustOrder);  } } |

## 13.3 配置文件

### 13.3.1 交易引擎的配置文件

|  |
| --- |
| spring:  application:  name: exchange-service  cloud: *# nacos地址* nacos:  server-addr: nacos-server:8848  config:  file-extension: yaml  stream:  rocketmq:  binder:  name-server: rocket-server:9876 *#/RocketMQ Message hasn't been sent* binders:  order\_out: {consumer.orderly: true}  bindings:  order\_out: {destination: order\_in, content-type: application/plain}  cancel\_order\_in: {destination: cancel\_order\_out, content-type: application/plain, group: order-group, consumer.maxAttempts: 1}  exchange\_trade\_in: {destination: exchange\_trades\_out, content-type: application/plain, group: order-group, consumer.maxAttempts: 1} *# 拉取的配置文件的dataID = exchange-service-dev.yaml* profiles:  active: dev |

### 13.3.2 撮合引擎的配置文件

|  |
| --- |
| spring:  application:  name: match-service  profiles:  active: dev  cloud:  nacos:  server-addr: nacos-server:8848  config:  file-extension: yaml  stream:  bindings:  order\_in: {destination: order\_in, content-type: application/plain, group: order-group, consumer.maxAttempts: 1}  trade\_plate\_out: {destination: trade\_plate\_out, content-type: application/plain}  completed\_orders\_out: {destination: completed\_orders\_out, content-type: application/plain}  exchange\_trades\_out: {destination: exchange\_trades\_out, content-type: application/plain}  cancel\_order\_out: {destination: cancel\_order\_out, content-type: application/plain}  rocketmq:  binder:  name-server: rocket-server:9876  *# 去 nacos-server 里面拉取 match-service-dev.yaml* match:  symbols:  BTCGCN:  coinScale: 8  baseCoinScale: 8  ETHGCN:  coinScale: 8  baseCoinScale: 8  FOFTGCN:  coinScale: 8  baseCoinScale: 8 |