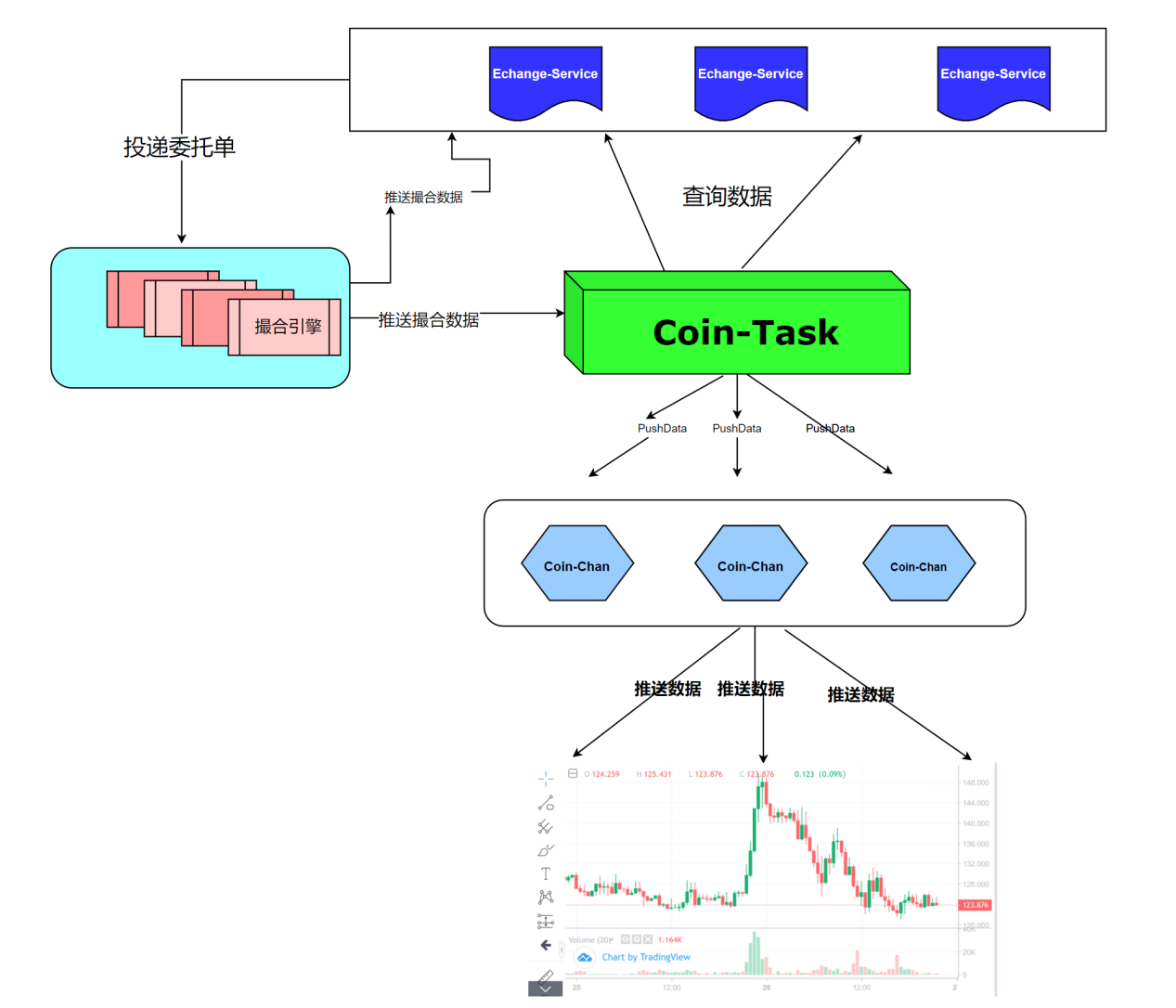
# 推送系统的实现

# 任务系统的简介

从上游(Match-engine\Exchange-engin)会获取数据,对数据做一定的加工后,将该数据推送给下游服务里面.



# 任务系统的启动

## 2.1 事件

事件是某种动作的载体,我们认为发送了某种事件后,肯定会触发该动作。

## 2.2 定时任务

使用定时任务触发事件,从而引发事件里面的处理器处理它

## 2.3 添加依赖

|  |
| --- |
| <dependency>  <groupId>com.bjsxt</groupId>  <artifactId>coin-common</artifactId>  <version>1.0</version> </dependency> <dependency>  <groupId>joda-time</groupId>  <artifactId>joda-time</artifactId> </dependency> <dependency>  <groupId>com.bjsxt</groupId>  <artifactId>exchange-api</artifactId>  <version>1.0</version> </dependency> |

## 2.4 添加配置文件

|  |
| --- |
| spring:  application:  name: task-service  cloud: *# nacos地址* nacos:  server-addr: nacos-server:8848  config:  file-extension: yaml  stream:  bindings:  subscribe\_event\_out: {destination: tio\_group, content-type: application/plain, group: order-group, consumer.maxAttempts: 1}  rocketmq:  binder:  name-server: rocket-server:9876  profiles:  active: dev *# 拉取的配置文件的dataID = task-service-dev.yaml* |

## 2.5 添加启动类

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

# 事件定义

## 3.1 定义事件的接口

|  |
| --- |
| */\*\*  \* 事件的接口  \*/* public interface Event {   */\*\*  \* 事件触发处理机制  \*/* void handle(); } |

## 3.2 盘口深度事件

|  |
| --- |
| */\*\*  \* 深度盘口数据事件  \*/* @Component @Slf4j public class DepthEvent implements Event {  */\*\*  \* 推送市场合并深度  \*/* @Override  public void handle() {    } } |

## 3.3 K线推送事件

|  |
| --- |
| */\*\*  \* K 线推送事件  \*/* @Component @Slf4j public class KlineEvent implements Runnable,Event {   */\*\*  \* 交易对标识符  \*/* private String symbol;   */\*\*  \* 通道  \*/* private String channel;   */\*\*  \* redis key 前缀  \*/* private String keyPrefix;   public KlineEvent() {  }   public KlineEvent(String symbol, String channel, String keyPrefix) {  this.symbol = symbol;  this.channel = channel;  this.keyPrefix = keyPrefix;  }   */\*\*  \* 事件触发处理机制  \*/* @Override  public void handle() {    }   */\*\*  \* 让线程池调度  \*/* @Override  public void run() {  handle();  }  } |

## 3.4 实时成交订单事件

|  |
| --- |
| */\*\*  \* 成交记录事件  \*/* @Component @Slf4j public class TradeEvent implements Event {   @Override  public void handle() {    } |

## 3.5 币币交易数据事件

|  |
| --- |
| */\*\*  \* 行情数据的K线  \*/* @Component @Slf4j public class TradeKLineEvent implements Event {   @Override  public void handle() {  }  } |

## 3.6 行情市场的事件

|  |
| --- |
| */\*\*  \* 行情市场的K 线  \*/* @Component @Slf4j public class MarketEvent implements Event {  @Override  public void handle() {  }    } |

# 事件触发

## 4.1 关于行情事件的触发

|  |
| --- |
| */\*\*  \* 行情的任务触发  \*/* @Component public class MarketTickerTask {   @Autowired  private MarketEvent marketEvent;   @Autowired  private TradeEvent tradeEvent;   @Autowired  private DepthEvent depthEvent;    */\*\*  \* 推送交易对信息  \*/* @Scheduled(fixedRate = 1000)  public void pushMarkets() {  marketEvent.handle();  }   */\*\*  \* 推送市场深度  \*/* @Scheduled(fixedRate = 500)  public void pushDepths() {  depthEvent.handle();  }   */\*\*  \* 推送实时成交订单数据  \*/* @Scheduled(fixedRate = 500)  public void pushTrades() {  tradeEvent.handle();  }  } |

## 4.2 关于K线事件的触发

|  |
| --- |
| */\*\*  \* K 线数据的推送  \*/* @Component @Slf4j public class TopicKLineTask{    private ExecutorService executor = null;   {  executor = new ThreadPoolExecutor(  5,  10,  100L, TimeUnit.*MILLISECONDS*,  new LinkedBlockingQueue<>(30),  new ThreadPoolExecutor.CallerRunsPolicy());  }   */\*\*  \* 每3秒推送K线数据  \*/* @Scheduled(fixedRate = 3000)  public void pushKline() {   String channel = "market.%s.kline.%s";  executor.submit(new KlineEvent())  } } |

## 4.3 关于交易事件的触发

|  |
| --- |
| @Component @Slf4j public class TradeKLineTask {   @Autowired  private TradeKLineEvent tradeKLineEvent;   */\*\*  \* 币币交易生成一次K线  \*/* @Scheduled(fixedRate = 25000)  public void generateKLine() {  tradeKLineEvent.handle();  } } |

# 整合MQ 向下游发送数据

## 5.1 添加依赖

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

## 5.2 添加Source

|  |
| --- |
| public interface Source {    */\*\*  \* 向指定的output 里面输出信息  \* @return  \*/* @Output("subscribe\_event\_out")  MessageChannel subscribeEventOutput() ; } |

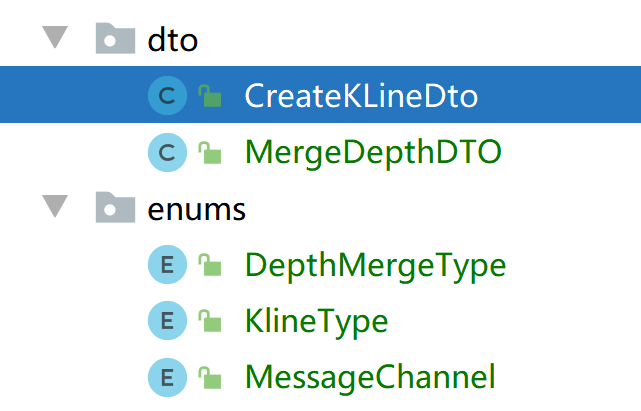
## 5.3 开启Stream

|  |
| --- |
| */\*\*  \* 开启Stream  \*/* @Configuration @EnableBinding(value = Source.class) public class RocketMqConfig { } |

## 5.3 添加配置文件

|  |
| --- |
| spring:  application:  name: task-service  cloud: *# nacos地址* nacos:  server-addr: nacos-server:8848  config:  file-extension: yaml  stream:  bindings:  subscribe\_event\_out: {destination: tio\_group, content-type: application/plain, group: order-group, consumer.maxAttempts: 1}  rocketmq:  binder:  name-server: rocket-server:9876  profiles:  active: dev *# 拉取的配置文件的dataID = task-service-dev.yaml* |

## 5.4 添加常用的工具类/枚举类



### CreateKLineDto : common里面

|  |
| --- |
| @Data @EqualsAndHashCode public class CreateKLineDto {   */\*\*  \* 交易对名称  \*/* private String symbol;   */\*\*  \* 交易的价格  \*/* private BigDecimal price;   */\*\*  \* 交易的数量  \*/* private BigDecimal volume;  } |

### MergeDepthDTO :

|  |
| --- |
| @Data @NoArgsConstructor @Accessors(chain = true) public class MergeDepthDTO {   */\*\*  \* 合并类型  \*/* private String mergeType;   */\*\*  \* 合并精度  \*/* private BigDecimal value; } |

### DepthMergeType :

|  |
| --- |
| public enum DepthMergeType {   *DEFAULT*("step0", 0),  *LOW*("step1", 1),  *HIGH*("step2", 2);   */\*\*  \* 代码  \*/* private String code;   */\*\*  \* 值  \*/* private int value;   public String getCode() {  return code;  }   public void setCode(String code) {  this.code = code;  }   public int getValue() {  return value;  }   public void setValue(int value) {  this.value = value;  }   DepthMergeType(String code, int value) {  this.code = code;  this.value = value;  }   public static DepthMergeType getByCode(String code) {  if (StringUtils.*isEmpty*(code)) {  return null;  }  for (DepthMergeType depthMergeType : DepthMergeType.*values*()) {  if (depthMergeType.getCode().equals(code)) {  return depthMergeType;  }  }  return null;  }   public static DepthMergeType getByValue(int value) {  for (DepthMergeType depthMergeType : DepthMergeType.*values*()) {  if (depthMergeType.getValue() == value) {  return depthMergeType;  }  }  return null;  } } |

### KlineType:

|  |
| --- |
| */\*\*  \* K线类型  \*/* public enum KlineType {    *ONE\_MINUTES*("1min"),  *FIVE\_MINUTES*("5min"),  *FIFTEEN\_MINUTES*("15min"),  *THIRTY\_MINUTES*("30min"),  *ONE\_HOUR*("1hour"),  *TWO\_HOURS*("2hour"),  *FOUR\_HOURS*("4hour"),  *SIX\_HOURS*("6hour"),  *TWELVE\_HOURS*("12hour"),  *ONE\_DAY*("1day"),  *ONE\_WEEK*("1week"),  *ONE\_MONTH*("1mon"),  *ONE\_YEAR*("1year");   private String value;   KlineType(final String value) {  this.value = value;  }   public String getValue(){  return this.value;  }   public static KlineType getByValue(String value) {  for (KlineType klineType : KlineType.*values*()) {  if (klineType.getValue().equals(value)) {  return klineType;  }  }  return null;  } } |

### MessageChannel :

|  |
| --- |
| */\*\*  \* 订阅的频道名称  \*/* public enum MessageChannel {    *ORDER\_TX*("order.tx"),  *ORDER\_CANCEL*("order.cancel"),  *ORDER\_IN*("order.in"),  *MATCH\_IN*("match.in"),  *ORDER\_DELAY\_NOTIFY*("order.delay.notify"),  *SYNC\_ACCOUNT*("sync.account"),  *MARKET\_REFRESH*("market.refresh"),  *BONUS*("bonus"),  *REGISTER\_REWARD*("register.reward"),  *FINANCE\_RECHARGE\_SUCCESS*("finance.recharge.success"),  *FINANCE\_WITHDRAW\_RESULT*("finance.withdraw.result"),  *RECHARGE\_ADDRESS*("plant.user.address");   private String channel;   MessageChannel(String channel) {  this.channel = channel;  }   public String getChannel() {  return channel;  }   public void setChannel(String channel) {  this.channel = channel;  } } |

### 常量Constants :

|  |
| --- |
| */\*\*  \* 常用的常量  \*/* public interface Constants {  */\*\*  \* UTF-8 字符集  \*/* public static final String *UTF8* = "UTF-8";   */\*\*  \* GBK 字符集  \*/* public static final String *GBK* = "GBK";   */\*\*  \* http请求  \*/* public static final String *HTTP* = "http://";   */\*\*  \* https请求  \*/* public static final String *HTTPS* = "https://";   */\*\*  \* 成功标记  \*/* public static final Integer *SUCCESS* = 200;   */\*\*  \* 失败标记  \*/* public static final Integer *FAIL* = 500;    */\*\*  \* 验证码 redis key  \*/* public static final String *CAPTCHA\_CODE\_KEY* = "captcha\_codes:";   */\*\*  \* 验证码有效期（分钟）  \*/* public static final long *CAPTCHA\_EXPIRATION* = 2;    */\*\*  \* 数值类型精度  \*/* public static final int *SCALE* = 8;   public static final int *AGENT\_TYPE* = 2;   public static final String *BCRYPT\_SALT* = "!waihui!\_+waihui+-";   */\*\*  \* 首页Banner图  \*/* public static final String *CONFIG\_WEB\_BANNER* = "WEB\_BANNER";   public static final String *TOKEN* = "token";   */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MATCH \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* public static final String *MATCH\_RESULT\_DEAL\_ORDER* = "dealOrder";  public static final String *MATCH\_RESULT\_BUY\_ORDER* = "buyOrder";  public static final String *MATCH\_RESULT\_SELL\_ORDER* = "sellOrder";  public static final String *MATCH\_RESULT\_NEW\_BUY\_LOCK* = "buyLock";  public static final String *MATCH\_RESULT\_NEW\_SELL\_LOCK* = "sellLock";  public static final String *MATCH\_RESULT\_PART\_MATCH* = "partMatch";  */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MATCH \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CONFIG TABLE CODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* public static final String *CONFIG\_TYPE\_SYSTEM* = "SYSTEM";  public static final String *CONFIG\_TYPE\_CNY* = "CNY";  public static final String *CONFIG\_TYPE\_SMS* = "SMS";  public static final String *CONFIG\_TYPE\_REGISTER* = "REGISTER\_REWARD";  public static final String *CONFIG\_TYPE\_REWARD* = "INVITE\_REWARD";   */\*\*  \* 最小提现额度（USDT）  \*/* public static final String *CONFIG\_WITHDRAW\_MIN\_AMOUNT* = "WITHDRAW\_MIN\_AMOUNT";   */\*\*  \* 最大提现额度（USDT）  \*/* public static final String *CONFIG\_WITHDRAW\_MAX\_AMOUNT* = "WITHDRAW\_MAX\_AMOUNT";   */\*\*  \* 最小取现手续费（USDT）  \*/* public static final String *CONFIG\_WITHDRAW\_MIN\_POUNDAGE* = "WITHDRAW\_MIN\_POUNDAGE";   */\*\*  \* 取现手续费率（USDT）  \*/* public static final String *CONFIG\_WITHDRAW\_POUNDAGE\_RATE* = "WITHDRAW\_POUNDAGE\_RATE";   */\*\*  \* 取现基数（USDT），取现值必须是基数的倍数，基数如果是100，那么取现值只能是100的倍数，例如：200, 300等  \*/* public static final String *CONFIG\_WITHDRAW\_BASEAMOUNT* = "CONFIG\_WITHDRAW\_BASEAMOUNT";   */\*\*  \* 每日最大提现额（USDT）  \*/* public static final String *CONFIG\_WITHDRAW\_DAY\_MAX\_AMOUNT* = "WITHDRAW\_DAY\_MAX\_AMOUNT";   */\*\*  \* 提现状态（USDT）  \*/* String *CONFIG\_WITHDRAW\_STATUS* = "WITHDRAW\_STATUS";   */\*\*  \* 币币交易状态  \*/* String *CONFIG\_TRADE\_STATUS* = "TRADE\_STATUS";   */\*\*  \* 人民币充值USDT换算费率  \*/* String *CONFIG\_CNY2USDT* = "CNY2USDT";   */\*\*  \* 人民币提现USDT换算费率  \*/* String *CONFIG\_USDT2CNY* = "USDT2CNY";   */\*\*  \* 提现审核级数  \*/* String *CONFIG\_CASH\_WITHDRAW\_AUDIT\_STEPS* = "CASH\_WITHDRAW\_AUDIT\_STEPS";   */\*\*  \* 充值审核级数  \*/* String *CONFIG\_CASH\_RECHARGE\_AUDIT\_STEPS* = "CASH\_RECHARGE\_AUDIT\_STEPS";   */\*\*  \* 提币审核级数  \*/* String *CONFIG\_COIN\_WITHDRAW\_AUDIT\_STEPS* = "COIN\_WITHDRAW\_AUDIT\_STEPS";   String *C2C\_ADMIN\_USER\_ID* = "C2C\_ADMIN\_USER\_ID";  */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CONFIG TABLE CODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/    /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CACHE KEY \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*  \* 交易区域  \*/* String *CACHE\_KEY\_TRADE\_AREA* = "tradeArea:";   */\*\*  \* 交易区域  \*/* String *CACHE\_KEY\_TRADE\_AREAS* = "tradeAreas:";   */\*\*  \* 币种  \*/* String *CACHE\_KEY\_COIN* = "coin:";   */\*\*  \* 交易对列表  \*/* String *CACHE\_KEY\_MARKETS* = "markets:";   */\*\*  \* 交易对列表  \*/* String *CACHE\_KEY\_AREA\_MARKETS* = "areaMarkets:";   */\*\*  \* 交易对  \*/* String *CACHE\_KEY\_MARKET* = "market:";   */\*\*  \* 基础配置  \*/* String *CACHE\_KEY\_CONFIG* = "config:";   */\*\*  \* 系统配置列表  \*/* String *CACHE\_KEY\_CONFIG\_LIST* = "config\_list:";   */\*\*  \* 管理员  \*/* String *CACHE\_KEY\_ADMIN\_USER* = "user\_admin";  String *CACHE\_KEY\_C2C\_ADMIN\_USER* = "user\_c2c\_admin";  String *CACHE\_KEY\_AGENT\_ADMIN\_USER* = "user\_agent\_admin";  */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CACHE KEY \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/    /\*\*  \* 币币市场行情Socket推送  \*/* String *CH\_MARKETS\_TICKER* = "market.ticker";  */\*\*  \* 未成交委托订阅通道  \*/* String *CH\_ORDER\_PENDING* = "order.pending.update";  */\*\*  \* 历史委托订阅通道  \*/* String *CH\_ORDER\_FINISHED* = "order.finished.update";  */\*\*  \* 持仓汇总订阅通道  \*/* String *CH\_POSITION\_SUMMARY\_UPDATE* = "position.summary.update";  */\*\*  \* 今日平仓订阅通道  \*/* String *CH\_CLOSEPOSITION\_ORDERS\_UPDATE* = "closeposition.orders.update";  */\*\*  \* 今日委托订阅通道  \*/* String *CH\_ENTRUST\_ORDERS\_UPDATE* = "entrust.orders.update";  */\*\*  \* 今日成交订阅通道  \*/* String *CH\_TURNOVER\_ORDERS\_UPDATE* = "turnover.orders.update";    */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* REDIS KEY \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*  \* 验证码redis存储Key  \*/* String *REDIS\_KEY\_CAPTCHA\_KEY* = "CAPTCHA:";   */\*\*  \* 短信验证码redis存储Key  \*/* String *REDIS\_KEY\_SMS\_CODE\_KEY* = "SMSCODE:";   */\*\*  \* 登录设备存储key  \*/* String *REDIS\_KEY\_DEVICES\_KEY* = "DEVICES";   */\*\*  \* 最大缓存数据量  \*/* long *REDIS\_MAX\_CACHE\_KLINE\_SIZE* = 10000L;   */\*\*  \* 币币交易K线 redis存储Key  \*/* String *REDIS\_KEY\_TRADE\_KLINE* = "TRADE\_KLINE:";    */\*\*  \* 币币交易撮合队列 redis存储Key  \*/* String *REDIS\_KEY\_TRADE\_MATCH* = "TRADE\_MATCH:";   */\*\*  \* 币币交易对  \*/* String *REDIS\_KEY\_TRADE\_MARKET* = "TRADE\_MARKET";   */\*\*  \* 法币充值审核锁  \*/* String *REDIS\_KEY\_CASH\_RECHARGE\_AUDIT\_LOCK* = "CASH\_RECHARGE\_AUDIT\_LOCK:";   */\*\*  \* 法币提现审核锁  \*/* String *REDIS\_KEY\_CASH\_WITHDRAW\_AUDIT\_LOCK* = "CASH\_WITHDRAW\_AUDIT\_LOCK:";   */\*\*  \* 数字货币提现审核锁  \*/* String *REDIS\_KEY\_COIN\_WITHDRAW\_AUDIT\_LOCK* = "COIN\_WITHDRAW\_AUDIT\_LOCK:";   */\*\*  \* 币币交易撤单锁  \*/* String *REDIS\_KEY\_TRADE\_ORDER\_CANCEL\_LOCK* = "TRADE\_ORDER\_CANCEL\_LOCK:";   */\*\*  \* 币币交易撮合锁  \*/* String *REDIS\_KEY\_TRADE\_ORDER\_MATCH\_LOCK* = "TRADE\_ORDER\_MATCH\_LOCK:";   */\*\*  \* 创新交易资金账户锁  \*/* String *REDIS\_KEY\_TRADE\_ACCOUNT\_LOCK* = "TRADE\_ACCOUNT\_LOCK:";  */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* REDIS KEY \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/    /\*\*\*\*\*\*\*\*\*\*\*\*OAUTH\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* String *AUTHORIZATION\_HEADER* = "Authorization";   String *BEARER* = "Bearer ";  */\*\*\*\*\*\*\*\*\*\*\*\*OAUTH\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/   /\*\*\*\*\*\*\*\*\*\*\*\*STREAM\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/* String *CHANNEL\_SENDTO\_USER* = "user";  String *CHANNEL\_SENDTO\_GROUP* = "group";  String *CHANNEL\_TICKER\_UPDATE* = "ticker";      */\*\*  \* 钱包币  \*/* String *COIN\_TYPE\_QBB* = "qbb";  */\*\*  \* 认购币  \*/* String *COIN\_TYPE\_RGB* = "rgb";  } |

# 盘口深度事件触发机制的处理

## 6.1 远程调用市场数据

在 **MarketServiceFeign** 里面添加:

|  |
| --- |
| */\*\*  \* 获取所有的市场  \*  \* @return  \*/* **@GetMapping(value = "/all")  List<MarketDto> tradeMarkets();** |

## 6.2 远程调用该市场里面的盘口深度数据

在 **MarketServiceFeign** 里面添加:

|  |
| --- |
| */\*\*  \* 币币交易市场深度  \*  \* @param symbol 交易对标识符  \* @param mergeType 合并深度类型  \* @return  \*/* **@GetMapping("/depth/{symbol}/{mergeType}") String depth(@PathVariable("symbol") String symbol,@PathVariable("mergeType") String mergeType);** |

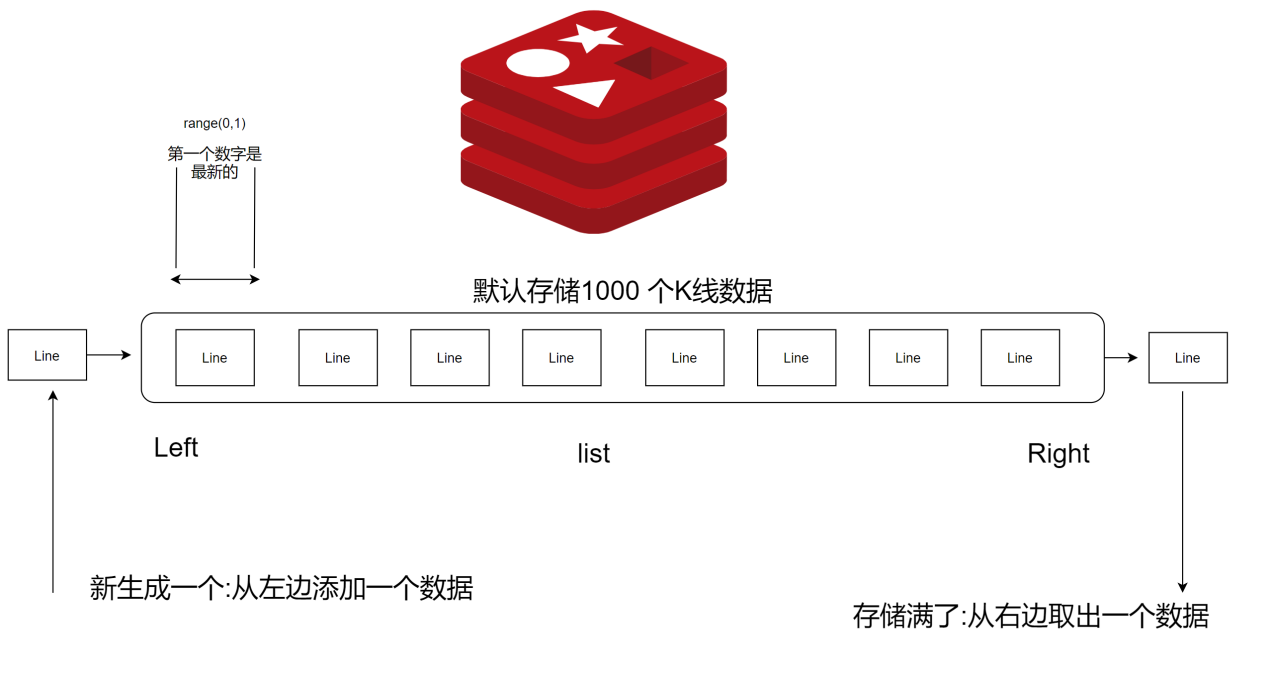
## 6.3 向下游订阅的队列里面推送该数据

订阅组: **market.%s.depth.step%s**

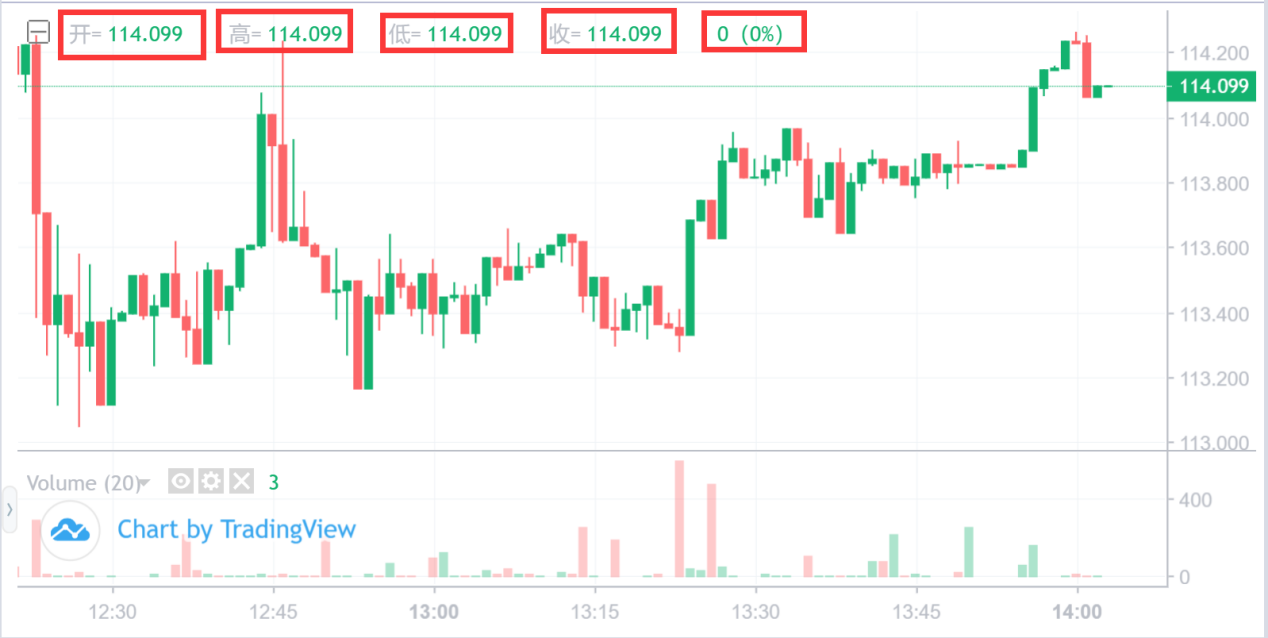
|  |
| --- |
| */\*\*  \* 深度盘口数据事件  \*/* @Component @Slf4j public class DepthEvent implements Event {    @Autowired  private ObjectMapper objectMapper;   @Autowired  private MarketServiceFeign marketServiceClient ;   @Autowired  private Source source ;   */\*\*  \* 推送市场合并深度  \*/* @Override  public void handle() {  marketServiceClient.tradeMarkets().forEach(market -> {  try {  for (DepthMergeType mergeType : DepthMergeType.*values*()) {  String groupName = String.*format*("market.%s.depth.step%s", market.getSymbol(), mergeType.getValue()).toLowerCase();  JSONObject body = new JSONObject();  String data = marketServiceClient.depth(market.getSymbol(), mergeType.getCode());  body.put("tick", data);   MessagePayload messagePayload = new MessagePayload();  messagePayload.setChannel(groupName);  messagePayload.setBody(objectMapper.writeValueAsString(body));  source.subscribeEventOutput().send(  MessageBuilder  .*withPayload*(messagePayload)  .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*)  .build()  );  }  } catch (JsonProcessingException e) {  e.printStackTrace();  }  });  } } |

# K线事件的处理

## 7.1 数据流向图



## 7.2 K线的模型



|  |
| --- |
| @Data public class Line {   */\*\*  \* 时间  \*/* private DateTime time;   */\*\*  \* 开盘价  \*/* private BigDecimal open;   */\*\*  \* 最高价  \*/* private BigDecimal high;   */\*\*  \* 最低价  \*/* private BigDecimal low;   */\*\*  \* 收盘价  \*/* private BigDecimal close;   */\*\*  \* 总交易量  \*/* private BigDecimal volume;   public Line() {}   */\*\*  \* 通过价格构造  \* @param time 时间  \* @param price 成交价  \* @param volume 成交量  \*/* public Line(DateTime time, BigDecimal price, BigDecimal volume) {  this.time = time;  this.open = price;  this.high = price;  this.low = price;  this.close = price;  this.volume = volume;  }   */\*\*  \* 通过K线构造  \* @param line k线  \*/* public Line(String line) {  }   */\*\*  \* 格式化称kline  \* @return  \*/* public String toKline() {  *// 时间，开，高，低，收，量* JSONArray array = new JSONArray();  array.add(time.getMillis());  array.add(open);  array.add(high);  array.add(low);  array.add(close);  array.add(volume);  return array.toJSONString();  } } |

## 7.3 K线的推送

|  |
| --- |
| */\*\*  \* K 线推送事件  \*/* @Component @Slf4j public class KlineEvent implements Runnable, Event {   */\*\*  \* 交易对标识符  \*/* private String symbol;   */\*\*  \* 通道  \*/* private String channel;   */\*\*  \* redis key 前缀  \*/* private String keyPrefix;   @Autowired  private static RedisTemplate<String, Object> *redisTemplate*;    @Autowired  private Source source;   public KlineEvent() {  }   public KlineEvent(String symbol, String channel, String keyPrefix) {  this.symbol = symbol;  this.channel = channel;  this.keyPrefix = keyPrefix;  }   */\*\*  \* 事件触发处理机制  \*/* @Override  public void handle() {  for (KlineType klineType : KlineType.*values*()) {  String groupName = String.*format*(channel, symbol.toLowerCase(), klineType.getValue()).toLowerCase();  String key = new StringBuffer(keyPrefix).append(symbol).append(":").append(klineType.getValue()).toString();  List<Object> klines = *redisTemplate*.opsForList().range(key, 0, 1);  if (!CollectionUtils.*isEmpty*(klines)) {  MessagePayload messagePayload = new MessagePayload();  messagePayload.setChannel(groupName);  JSONObject body = new JSONObject();  body.put("tick", klines.get(0));  messagePayload.setBody(body.toJSONString());  source.subscribeEventOutput().send(  MessageBuilder  .*withPayload*(messagePayload)  .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*)  .build()  );  }  }  }   @Override  public void run() {  handle();  } } |

## 7.4 K 线事件的触发

|  |
| --- |
| */\*\*  \* K 线推送事件  \*/* @Component @Slf4j public class KlineEvent implements Runnable,Event {   */\*\*  \* 交易对标识符  \*/* private String symbol;   */\*\*  \* 通道  \*/* private String channel;   */\*\*  \* redis key 前缀  \*/* private String keyPrefix;   private static final String *KLINE\_GROUP* = "market.%s.kline.%s" ; *// 1%s: 交易对 2%s: k 线类型* @Autowired  private RedisTemplate<String,Object> redisTemplate ;   @Autowired  private Source source ;   public KlineEvent() {  }   public KlineEvent(String symbol, String channel, String keyPrefix) {  this.symbol = symbol;  this.channel = channel;  this.keyPrefix = keyPrefix;  }   */\*\*  \* 事件触发处理机制  \*/* @Override  public void handle() {  *// 1 循环所有的K 线类型* for (KlineType klineType : KlineType.*values*()) {  *// 2 获取特定的K 线类型 keyPrefix:etcgcn:1min* String key = new StringBuilder(keyPrefix).append(symbol).append(":").append(klineType.getValue().toLowerCase()).toString() ;  List<Object> lines = redisTemplate.opsForList().range(key, 0, 1);  if(!CollectionUtils.*isEmpty*(lines)){  Object lineData = lines.get(0);  MessagePayload messagePayload = new MessagePayload();  JSONObject jsonObject = new JSONObject();  jsonObject.put("tick",lineData) ;  *// market.%s.kline.%s* messagePayload.setChannel(String.*format*(*KLINE\_GROUP*,symbol,klineType.getValue().toLowerCase()));  messagePayload.setBody(jsonObject.toString());  }   }   }   */\*\*  \* 让线程池调度  \*/* @Override  public void run() {  handle();  } } |

## 7.5 K 线的生成

|  |
| --- |
| @Component public class TradeKlineService implements CommandLineRunner {   *// 当我们的交易完成(撮合)之后,就会触发k 线的生成* public static BlockingQueue<CreateKLineDto> *queue* = new LinkedBlockingDeque<>();   @Autowired  private StringRedisTemplate redisTemplate;   @Override  public void run(String... args) throws Exception {   while (true) {  CreateKLineDto createKLineDto = *queue*.poll(); *// 从里面获取一个数据,若没有数据,则会阻塞* if (createKLineDto != null) {  for (KlineType klineType : KlineType.*values*()) {  this.generateKLine(createKLineDto, klineType);  }  }  }   }   */\*\*  \* 为当前的交易数据生成K 线  \*  \* @param klineData  \* @param klineType  \*/* private void generateKLine(CreateKLineDto klineData, KlineType klineType) {  *// 1 获取之前该K 线的数据* String redisKey = new StringBuilder(Constants.*REDIS\_KEY\_TRADE\_KLINE*)  .append(klineData.getSymbol().toLowerCase())  .append(":")  .append(klineType.getValue().toLowerCase()).toString();  Long size = redisTemplate.opsForList().size(redisKey);  DateTime dateTime = KlineTimeUtil.*getKLineTime*(klineType);  *// 2 之前没有该K 线的数据* if (size == 0) {  Line line = new Line(dateTime, klineData.getPrice(), klineData.getVolume());  *// 放在Redis 里面* redisTemplate.opsForList().leftPush(redisKey, line.toKline());  return;  }   *// 3 之前有数据 ,获取最近的一个数据* String historyKlineData = redisTemplate.opsForList().range(redisKey, 0, 1).get(0);  Line historyKline = new Line(historyKlineData);  *// 4 若当前的时间: 是否还在上一个时间的区间内* if (dateTime.compareTo(historyKline.getTime()) == 1) {   *// redis的容量是否超* if (size > Constants.*REDIS\_MAX\_CACHE\_KLINE\_SIZE*) {  redisTemplate.opsForList().rightPop(redisKey);  }   Line line = new Line();  line.setTime(dateTime);  *// 如果我们当前的交易量为 0* if (klineData.getVolume().compareTo(BigDecimal.*ZERO*) == 0) {  line.setHigh(historyKline.getClose());  line.setLow(historyKline.getClose());  line.setOpen(historyKline.getClose());  line.setClose(historyKline.getClose());  line.setVolume(BigDecimal.*ZERO*) ;  *// 放在Redis 里面* redisTemplate.opsForList().leftPush(redisKey, line.toKline());  return;  }  line.setOpen(klineData.getPrice());  line.setClose(klineData.getPrice());   *// 最高价 ,最低价* if (klineData.getPrice().compareTo(historyKline.getHigh()) == 1) {  line.setHigh(klineData.getPrice());  line.setLow(historyKline.getClose());  }  if (klineData.getPrice().compareTo(historyKline.getLow()) == 1) {  line.setLow(klineData.getPrice());  line.setHigh(historyKline.getClose());  }   historyKline.setClose(klineData.getPrice());  redisTemplate.opsForList().set(redisKey, 0, historyKline.toKline());  *// 放最新的进入  // 放在Redis 里面* redisTemplate.opsForList().leftPush(redisKey, line.toKline());  return;  }   if (klineData.getVolume().compareTo(BigDecimal.*ZERO*) == 0) {  return;  }   historyKline.setClose(klineData.getPrice());   if (klineData.getPrice().compareTo(historyKline.getHigh()) == 1) {  historyKline.setHigh(klineData.getPrice());  }   if (klineData.getPrice().compareTo(historyKline.getLow()) == 1) {  historyKline.setLow(klineData.getPrice());  }  historyKline.setVolume(historyKline.getVolume().add(klineData.getVolume()));  redisTemplate.opsForList().set(redisKey, 0, historyKline.toKline());  }   } |

## 7.6 监听撮合的数据

### 7.6.1 添加Sink

|  |
| --- |
| public interface Sink {   */\*\*  \* 交易记录的输入  \* @return  \*/* @Input("exchange\_trades\_in")  MessageChannel exchangeTradesIn() ; } |

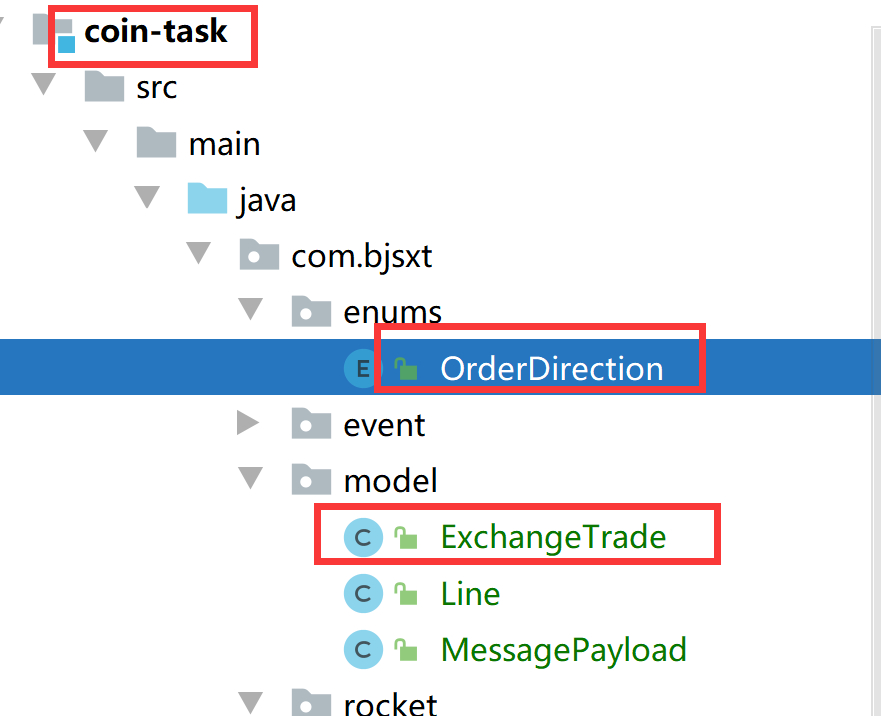
### 7.6.2 添加配置信息



### 7.6.3 监听器

|  |
| --- |
| @Component @Slf4j public class ExchangeTradesListener {    @StreamListener("exchange\_trades\_in")  public void handlerExchangeTrades(List<ExchangeTrade> exchangeTrades) {  *log*.info("接收到撮合引擎的数据===>{}", exchangeTrades);  if (!CollectionUtils.*isEmpty*(exchangeTrades)) {  for (ExchangeTrade exchangeTrade : exchangeTrades) {  CreateKLineDto createKLineDto = exchangeTrade2CreateKLineDto(exchangeTrade);  TradeKlineService.*queue*.offer(createKLineDto);  }  }  }   private CreateKLineDto exchangeTrade2CreateKLineDto(ExchangeTrade exchangeTrade) {  CreateKLineDto createKLineDto = new CreateKLineDto();  createKLineDto.setPrice(exchangeTrade.getPrice());  createKLineDto.setSymbol(exchangeTrade.getSymbol());  createKLineDto.setVolume(exchangeTrade.getAmount());  return createKLineDto;  }  } |

记得把撮合引擎的对象复制过来:



# 市场行情事件的处理

## 8.1 远程调用交易区域

### 8.1.1 创建TradeAreaDto对象

|  |
| --- |
| @Data public class TradeAreaDto implements java.io.Serializable {   */\*\*  \* 主键  \*/* private Long id;  */\*\*  \* 交易区名称  \*/* private String name;  */\*\*  \* 交易区代码  \*/* private String code;  */\*\*  \* 类型：1-数字货币交易；2-创新交易使用；  \*/* private Integer type;  */\*\*  \* 结算币种（仅创新交易需要使用）  \*/* private Long coinId;  */\*\*  \* 结算币种名称（仅创新交易需要使用）  \*/* private String coinName;  */\*\*  \* 排序  \*/* private int sort;  */\*\*  \* 状态  \*/* private Integer status;  */\*\*  \* 创建时间  \*/* private Date createTime;  */\*\*  \* 修改时间  \*/* private Date lastUpdateTime;   */\*\*  \* 交易区内所在市场Id(逗号分隔)  \*/* private String marketIds; } |

### 8.1.2 创建TradingAreaServiceClient

|  |
| --- |
| @FeignClient(name = "exchange-service",contextId = "tradingAreaServiceClient" ,configuration = OAuth2FeignConfig.class ,path = "/tradeArea" ) public interface TradingAreaServiceClient {   */\*\*  \* 查询所有的交易区域  \* @return  \*/* @GetMapping("/list")  List<TradeAreaDto> tradingAreaList(); } |

### 8.1.3 创建TradeMarketDto

|  |
| --- |
| @Data public class TradeMarketDto implements Comparable<TradeMarketDto> {   */\*\*  \* 交易对  \*/* private String symbol;   */\*\*  \* 市场名称  \*/* private String name;   */\*\*  \* 图片  \*/* private String image;   */\*\*  \* 买入手续费  \*/* private BigDecimal buyFeeRate;   */\*\*  \* 卖出手续费  \*/* private BigDecimal sellFeeRate;   */\*\*  \* 价格小数位数  \*/* private int priceScale;   */\*\*  \* 数量小数位数  \*/* private int numScale;   */\*\*  \* 最小委托数量  \*/* private BigDecimal numMin;   */\*\*  \* 最大小数位数  \*/* private BigDecimal numMax;   */\*\*  \* 最小成交额  \*/* private BigDecimal tradeMin;   */\*\*  \* 最大成交额  \*/* private BigDecimal tradeMax;   */\*\*  \* 当前价  \*/* private BigDecimal price;   */\*\*  \* 价格单位  \*/* private String priceUnit;   */\*\*  \* 当前价（人民币）  \*/* private BigDecimal cnyPrice;   */\*\*  \* 报价货币人民币价格  \*/* private BigDecimal coinCnyPrice;   */\*\*  \* 合并深度  \*/* private List<MergeDepthDto> mergeDepth;   */\*\*  \* 最高价  \*/* private BigDecimal high;   */\*\*  \* 最低价  \*/* private BigDecimal low;   */\*\*  \* 日总交易量  \*/* private BigDecimal volume = BigDecimal.*ZERO*;   */\*\*  \* 日总成交额  \*/* private BigDecimal amount = BigDecimal.*ZERO*;   */\*\*  \* 涨幅  \*/* private double change;   */\*\*  \* 排序  \*/* private int sort;   @Override  public int compareTo(TradeMarketDto object) {  if (this.sort >= object.getSort()) {  return 1;  } else {  return -1;  }  } } |

## 8.2 处理该事件

|  |
| --- |
| */\*\*  \* 行情市场的K 线  \*/* @Component @Slf4j public class MarketEvent implements Event {   @Autowired  private Source source;   @Autowired  private TradingAreaServiceClient tradingAreaServiceClient;    @Autowired  private MarketServiceFeign marketServiceFeign;   private static final String *MARKET\_GROUP* = "market.%s.ticker"; *// %s代表交易区域* private static final String *MARKET\_DETAIL\_GROUP* = "markets.%s.detail" ; *// 交易区域的详情交易数据* @Override  public void handle() {  *// 1 交易区域的查询* List<TradeAreaDto> tradeAreaDtoList = tradingAreaServiceClient.tradeAreas();  if (CollectionUtils.*isEmpty*(tradeAreaDtoList)) {  return;  }  for (TradeAreaDto tradeAreaDto : tradeAreaDtoList) {  *// 2 差该交易区域下的交易数据 , 使用的是交易区域里面市场的id("id1,id2")* List<TradeMarketDto> tradeMarketDtos = marketServiceFeign.queryMarkesByIds(tradeAreaDto.getMarketIds());  if (CollectionUtils.*isEmpty*(tradeMarketDtos)) {  return;  }   JSONObject jsonObject = new JSONObject();  jsonObject.put("markets", tradeMarketDtos);  MessagePayload messagePayload = new MessagePayload();  messagePayload.setChannel(String.*format*(*MARKET\_GROUP*, tradeAreaDto.getCode().toLowerCase()));  messagePayload.setBody(jsonObject.toJSONString());  source.subscribeGroupOutput()  .send(  MessageBuilder  .*withPayload*(messagePayload)  .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*).build()  );  }    *// 获取所有的交易市场* List<MarketDto> marketDtos = marketServiceFeign.tradeMarkets();  if(CollectionUtils.*isEmpty*(marketDtos)){  return;  }  for (MarketDto marketDto : marketDtos) {  List<TradeMarketDto> tradeMarketDtos = marketServiceFeign.queryMarkesByIds(marketDto.getId().toString());   JSONObject jsonObject = new JSONObject();  jsonObject.put("tick",tradeMarketDtos) ;   MessagePayload messagePayload = new MessagePayload();  messagePayload.setChannel(String.*format*(*MARKET\_DETAIL\_GROUP*,marketDto.getSymbol()));  messagePayload.setBody(jsonObject.toJSONString());  source.subscribeGroupOutput()  .send(  MessageBuilder  .*withPayload*(messagePayload)  .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*).build()  );  }   } } |

# 成交记录事件的处理

## 9.1 远程调用数据

在**MarketServiceFeign**里面添加:

|  |
| --- |
| */\*\*  \* 获取最新成交列表  \*  \* @param symbol 交易对标识符  \* @return  \*/* @GetMapping("/trades/{symbol}") String trades(@PathVariable("symbol") String symbol); |

## 9.2 处理该事件

|  |
| --- |
| */\*\*  \* 成交记录事件  \*/* @Component @Slf4j public class TradeEvent implements Event {   @Autowired  private MarketServiceFeign marketServiceClient;    @Autowired  private Source source;   @Override  public void handle() {  try {  marketServiceClient.tradeMarkets().forEach(market -> {  String groupName = String.*format*("market.%s.trade.detail", market.getSymbol()).toLowerCase();  String data = marketServiceClient.trades(market.getSymbol());  JSONObject orders = new JSONObject();  orders.put("data", data);   MessagePayload messagePayload = new MessagePayload();  messagePayload.setChannel(groupName);  messagePayload.setBody(orders.toJSONString());  source.subscribeEventOutput().send(  MessageBuilder  .*withPayload*(messagePayload)  .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*)  .build()  );  });  } catch (Exception e) {  *log*.error("推送最新成交订单异常");  e.printStackTrace();  }  } } |

# 行情数据事件

## 10.1 处理该事件

|  |
| --- |
| */\*\*  \* 行情数据的K线  \*/* @Component @Slf4j public class TradeKLineEvent implements Event {   @Autowired  private MarketServiceFeign marketServiceClient;   @Override  public void handle() {  List<MarketDto> markets = marketServiceClient.tradeMarkets();  if (!CollectionUtils.*isEmpty*(markets)) {  for (MarketDto market : markets) {  CreateKLineDto createKLineDTO = new CreateKLineDto();  createKLineDTO.setSymbol(market.getSymbol()) ;  createKLineDTO.setPrice(market.getOpenPrice());  createKLineDTO.setVolume(BigDecimal.*ZERO*);  TradeKLineService.*queue*.offer(createKLineDTO);  }  }  } } |

# 定时任务完善

## 11.1 刷新24小时的成交记录

### 11.1.1 远程调用

在**MarketServiceFeign**里面添加:

|  |
| --- |
| */\*\*  \* 刷新24小时成交数据  \*  \* @param symbol 交易对  \* @return  \*/* @GetMapping(value = "/refresh\_24hour") void refresh24hour(@RequestParam("symbol") String symbol); |

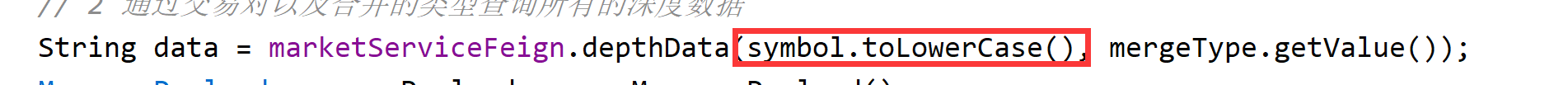
### 11.1.2 添加任务

在: MarketTickerTask 里面添加

|  |
| --- |
| */\*\*  \* 刷新24小时成交数据  \*/* @Scheduled(fixedRate = 1000) public void refresh24HDeal() {  marketServiceClient.tradeMarkets().forEach(market -> {  if (market.getStatus() == 1) {  marketServiceClient.refresh24hour(market.getSymbol());  }  }); } |

## 11.2 关于前端订阅的channel

前端订阅数据时:它的所有的交易对都是小写,因次,我们在使用source发送数据之前,需要将所有的symbol都转化为小写.



# 定时任务的启动

## 12.1 MarketServiceFeign 的处理

|  |
| --- |
| @FeignClient(name = "exchange-service", contextId = "marketServiceFeign", configuration = OAuth2FeignConfig.class, path = "/markets") public interface MarketServiceFeign {   */\*\*  \* 使用报价货币 以及 出售的货币的iD  \*  \* @param buyCoinId  \* @return  \*/* @GetMapping("/getMarket")  MarketDto findByCoinId(@RequestParam("buyCoinId") Long buyCoinId, @RequestParam("sellCoinId") Long sellCoinId);    @GetMapping("/getMarket/symbol")  MarketDto findBySymbol(@RequestParam("symbol") String symbol);   */\*\*  \* 查询所有的交易市场  \*  \* @return  \*/* @GetMapping("/tradeMarkets")  List<MarketDto> tradeMarkets();   */\*\*  \* 查询该交易对下的盘口数据  \*  \* @param symbol  \* @param value  \* @return  \*/* @GetMapping("/tradeMarkets/{symbol}/{type}")  String depthData(@PathVariable("symbol") String symbol, @PathVariable("type") int value);   */\*\*  \* 使用市场的ids 查询该市场的交易趋势  \*  \* @param marketIds  \* @return  \*/* @GetMapping("/queryMarketsByIds")  List<TradeMarketDto> queryMarkesByIds(@RequestParam("marketIds") String marketIds);   */\*\*  \* 通过交易对查询所有的交易数据  \*  \* @param symbol  \* @return  \*/* @GetMapping("/trades")  String trades(String symbol);    */\*\*  \* 刷新24小时成交数据  \*  \* @param symbol 交易对  \* @return  \*/* @GetMapping(value = "/refresh\_24hour")  void refresh24hour(@RequestParam("symbol") String symbol);   } |

## 12.2 TradingAreaServiceClient的处理

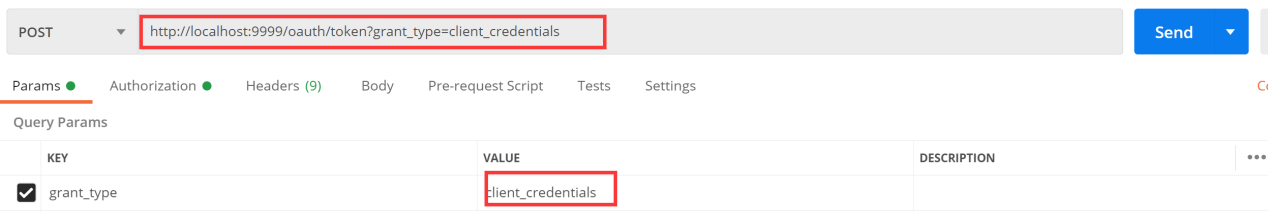
|  |
| --- |
| @FeignClient(name = "exchange-service",contextId = "tradingAreaServiceClient" ,configuration = OAuth2FeignConfig.class,path = "/tradeArea" ) public interface TradingAreaServiceClient {    */\*\*  \* 查询所有的交易区域  \* @return  \*/* @GetMapping("/all")  List<TradeAreaDto> tradeAreas(); } |

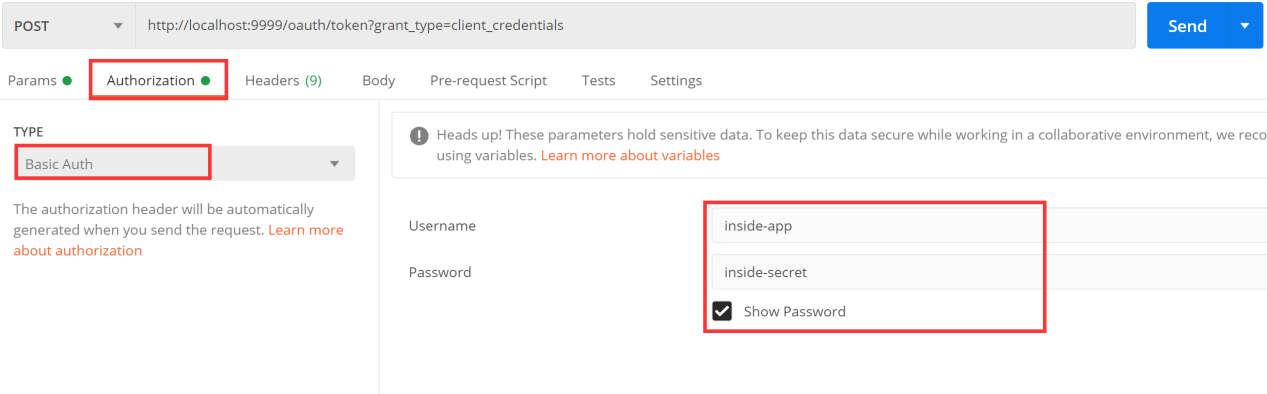
## 12.3 Token 传递问题

### 12.3.1 修改AuthorizationServerConfig

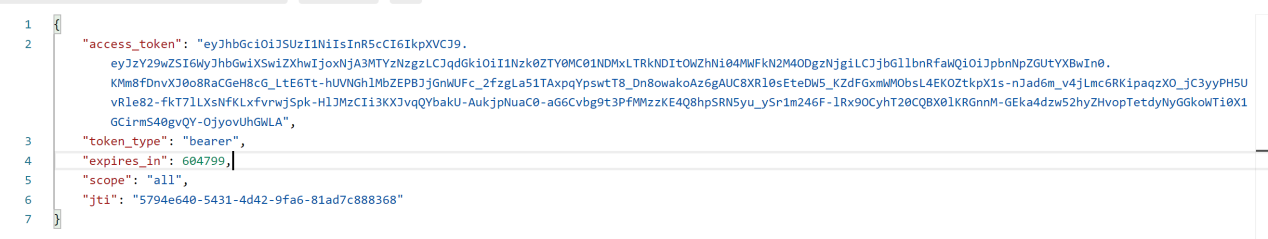


### 12.3.2 获取Token 并放在Constants 里面





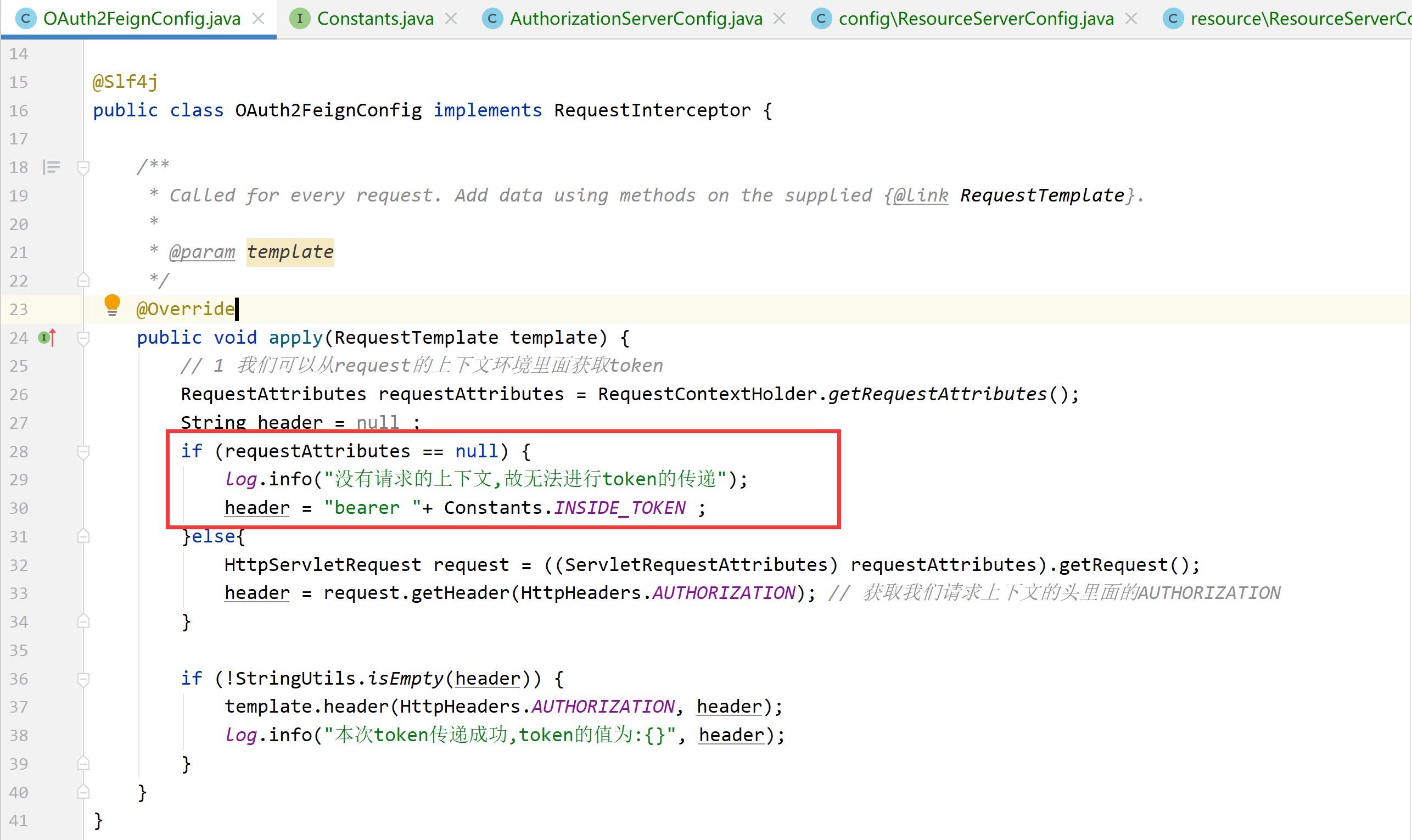
获取到Token:

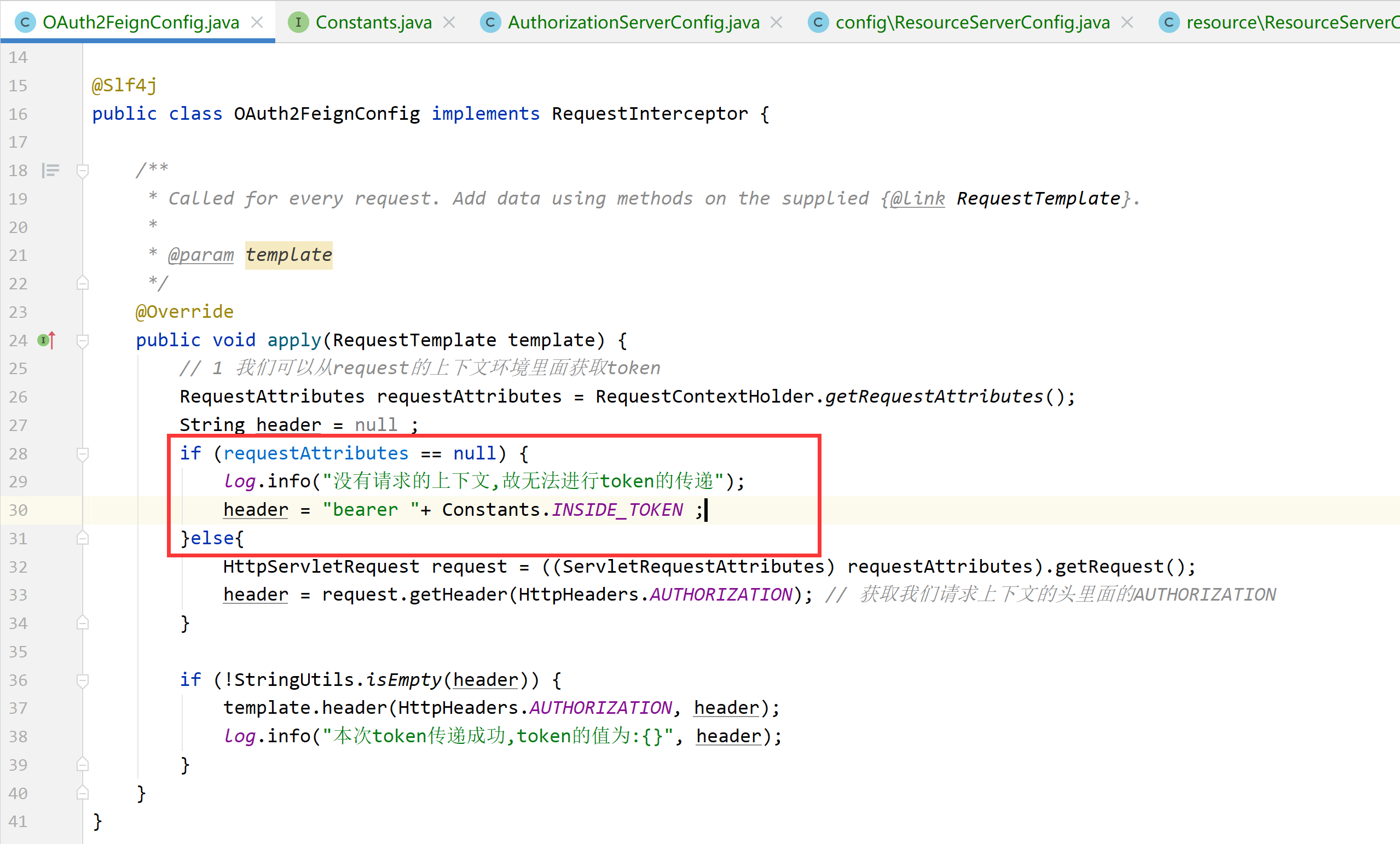


放在常量里面:



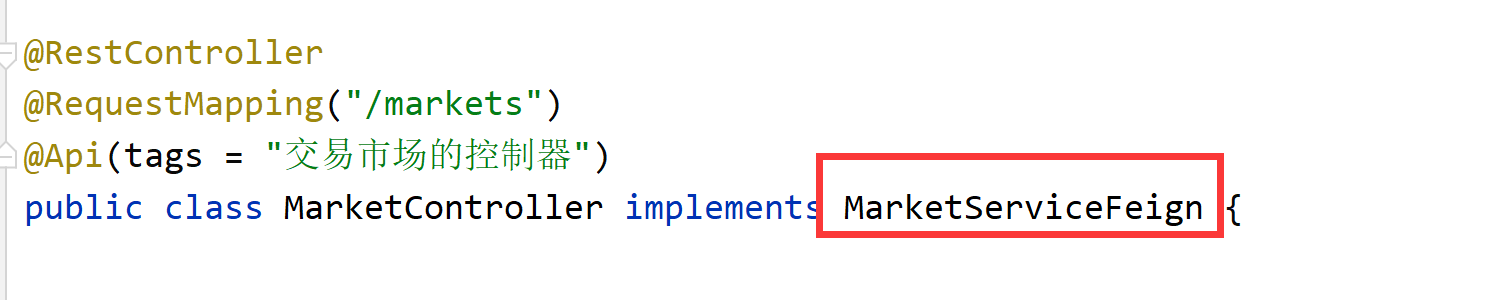
### 12.3.3 传递获取到的内置的Token





# 盘口数据的推送测试

## 13.1 实现MarketServiceFeign 里面的方法



### 13.1.1 MarketController

|  |
| --- |
| */\*\*  \* 查询所有的交易市场  \*  \* @return  \*/* @Override public List<MarketDto> tradeMarkets() {  return marketService.queryAllMarkets() ; }  */\*\*  \* 查询该交易对下的盘口数据  \*  \* @param symbol  \* @param value  \* @return  \*/* @Override public String depthData(String symbol, int value) {  R<DepthsVo> deptVosSymbol = findDeptVosSymbol(symbol, value + "");  DepthsVo data = deptVosSymbol.getData();  return JSON.*toJSONString*(data); } |

### 13.1.2 MarketService

|  |
| --- |
| */\*\*  \* 查询所有的市场数据  \* @return  \*/* List<MarketDto> queryAllMarkets(); |

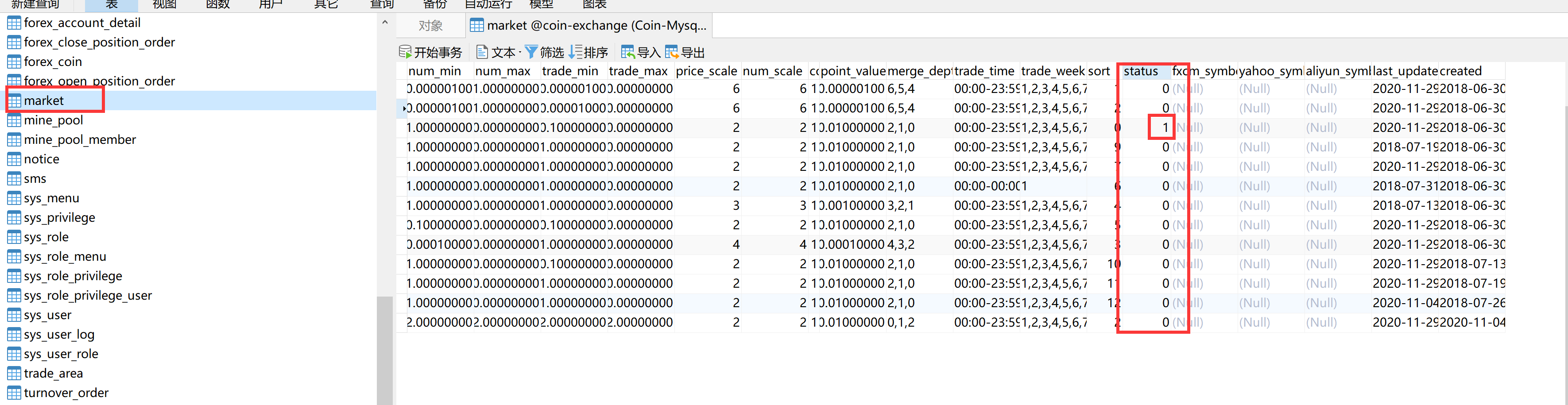
### 13.1.3 MarketServiceImpl

|  |
| --- |
| */\*\*  \* 查询所有的市场数据  \*  \* @return  \*/* @Override public List<MarketDto> queryAllMarkets() {  List<Market> list = list(new LambdaQueryWrapper<Market>().eq(Market::getStatus, 1));  return MarketDtoMappers.*INSTANCE*.toConvertDto(list); } |

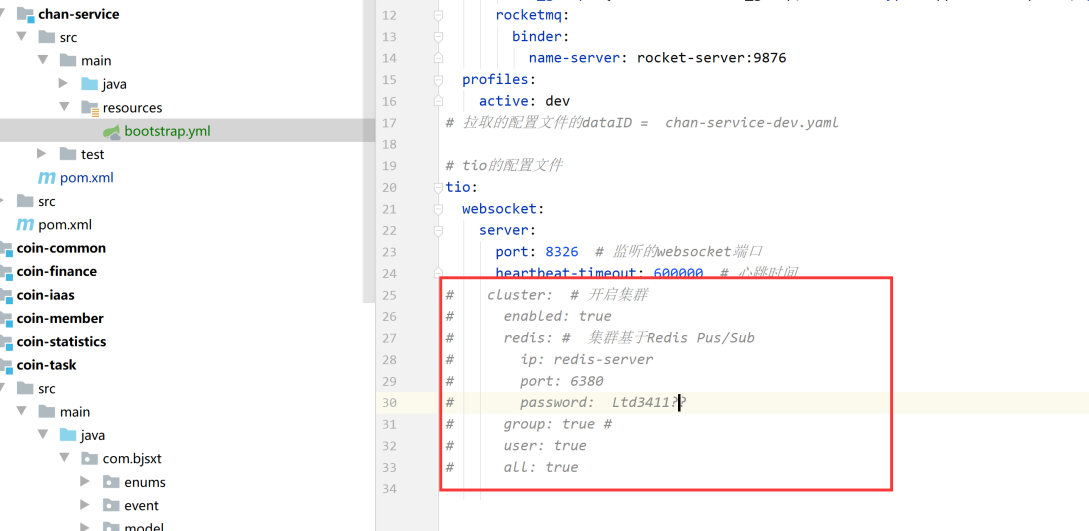
## 13.2 测试盘口数据的推送

### 13.2.1因为计算机性能较差,我们需要先将部分数据修改

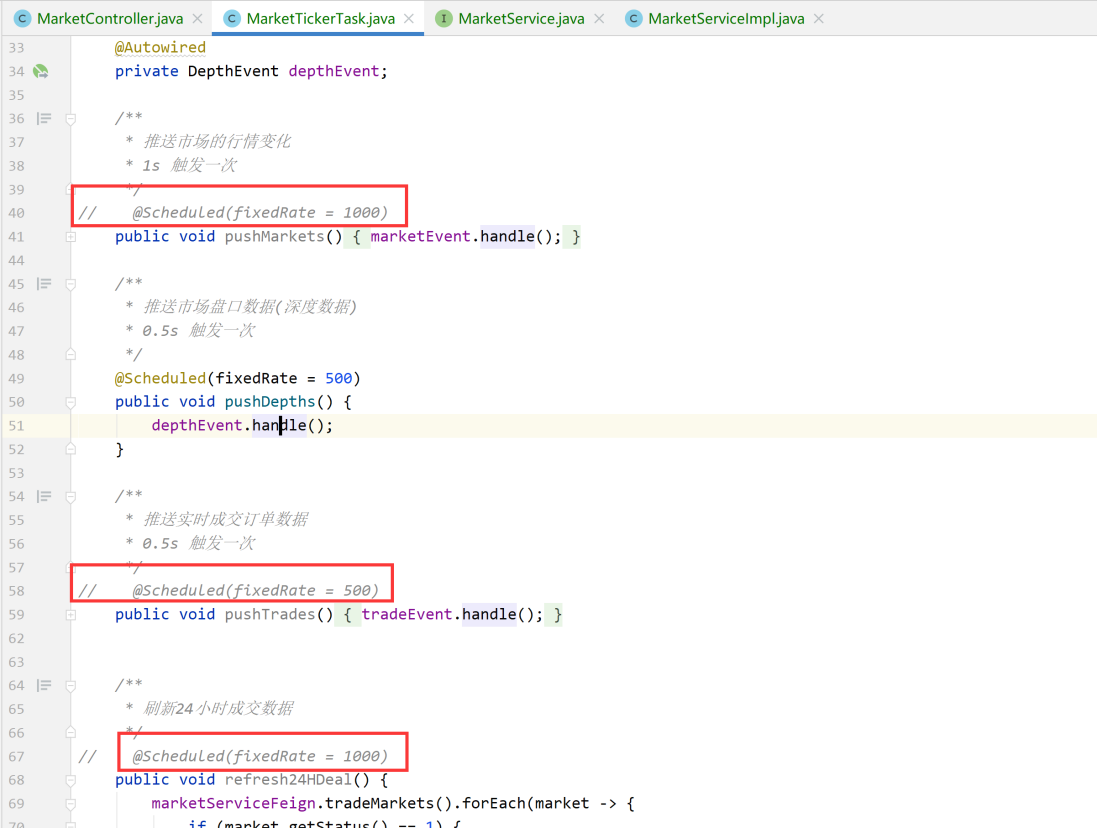
:将交易市场设置为 关闭,只剩第三个启用

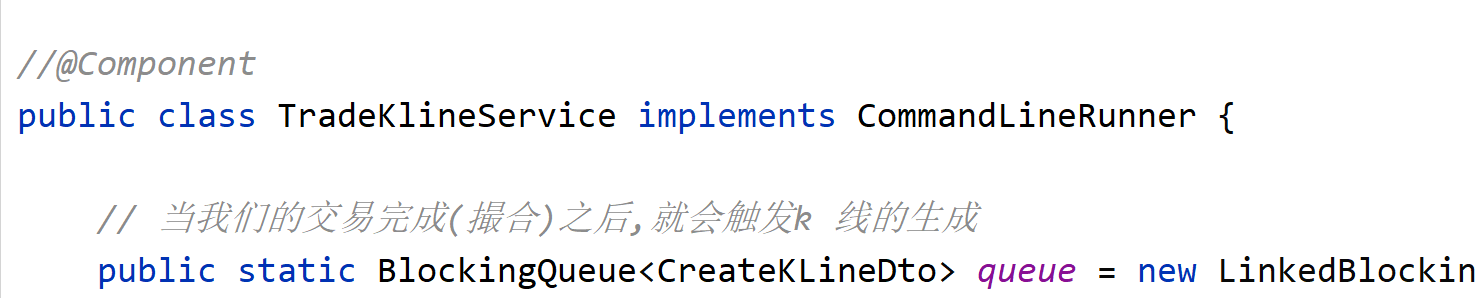


修改exchange-chan的配置文件,修改为单机推送:



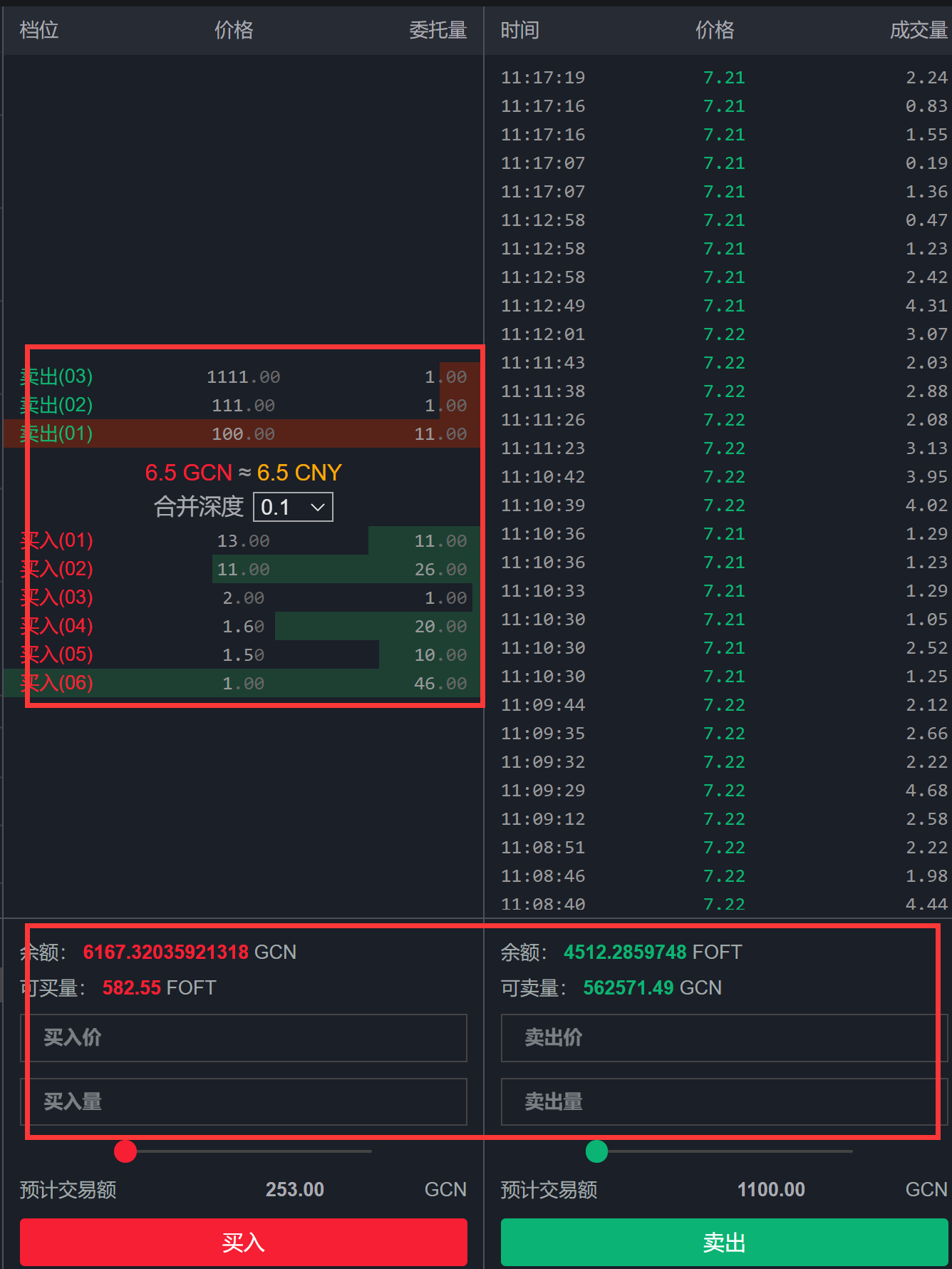
### 13.2.2 关闭多余的定时任务





### 13.2.3 测试推送

新增数据后,观察数据的变化就可以了



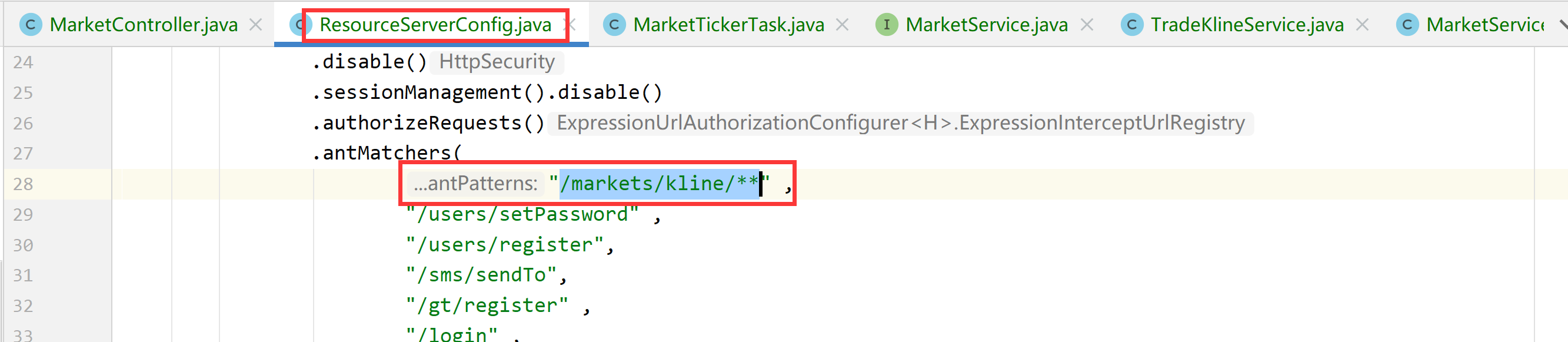
# KLine的推送

## 14.1 完成MarketController 里面的K线查询方法

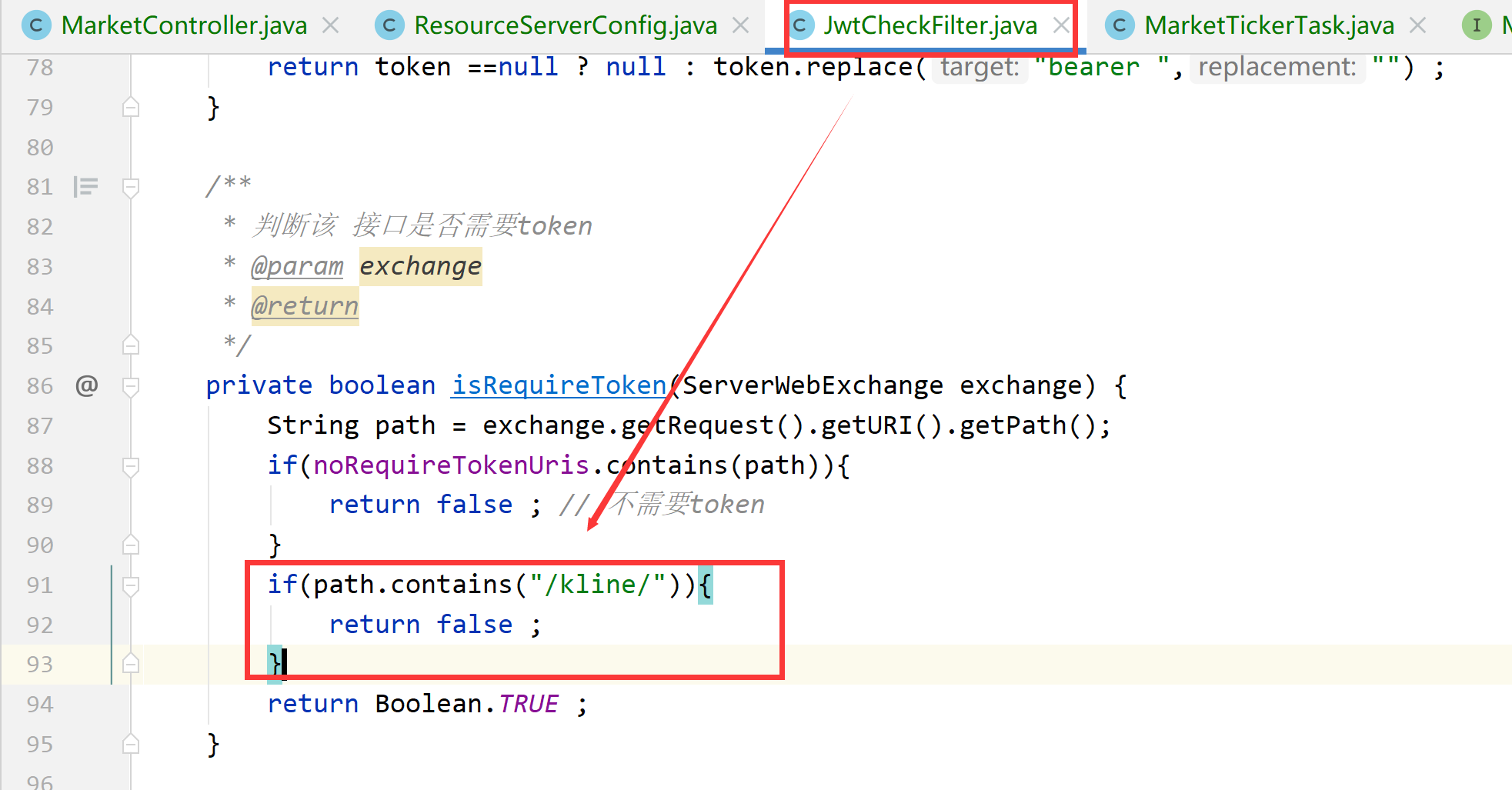
|  |
| --- |
| */\*\*  \* K 线的查询  \*  \* @param symbol 交易对  \* @param type K 线类型  \* @return  \*/* @GetMapping("/kline/{symbol}/{type}") public R<List<JSONArray>> queryKLine(@PathVariable("symbol") String symbol, @PathVariable("type") String type) {  *// 我们的K 线放在Redis 里面* String redisKey = new StringBuilder(Constants.*REDIS\_KEY\_TRADE\_KLINE*).append(symbol.toLowerCase()).append(":").append(type).toString();  List<String> klines = redisTemplate.opsForList().range(redisKey, 0, Constants.*REDIS\_MAX\_CACHE\_KLINE\_SIZE* - 1);  List<JSONArray> result = new ArrayList<>(klines.size()) ;  if (!CollectionUtils.*isEmpty*(klines)) {  for (String kline : klines) {  JSONArray objects = JSON.*parseArray*(kline);  result.add(objects) ;  }  return R.*ok*(result);  }   return null; } |

## 14.2 放行该请求

ResourceServerConfig 里面:



网关里面:JwtCheckFilter:



## 14.3 关闭其他的定时任务,只推K 线





## 14.4 打开 K 线推送

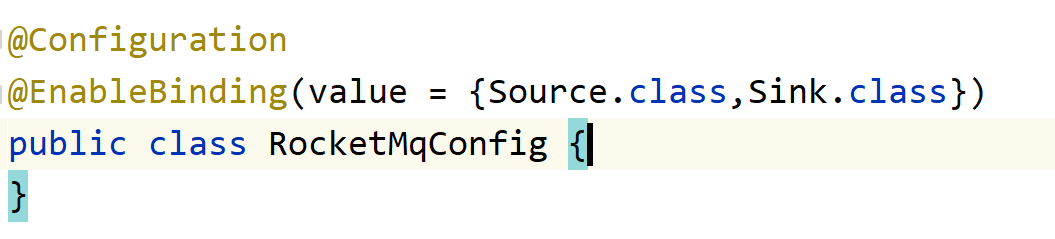












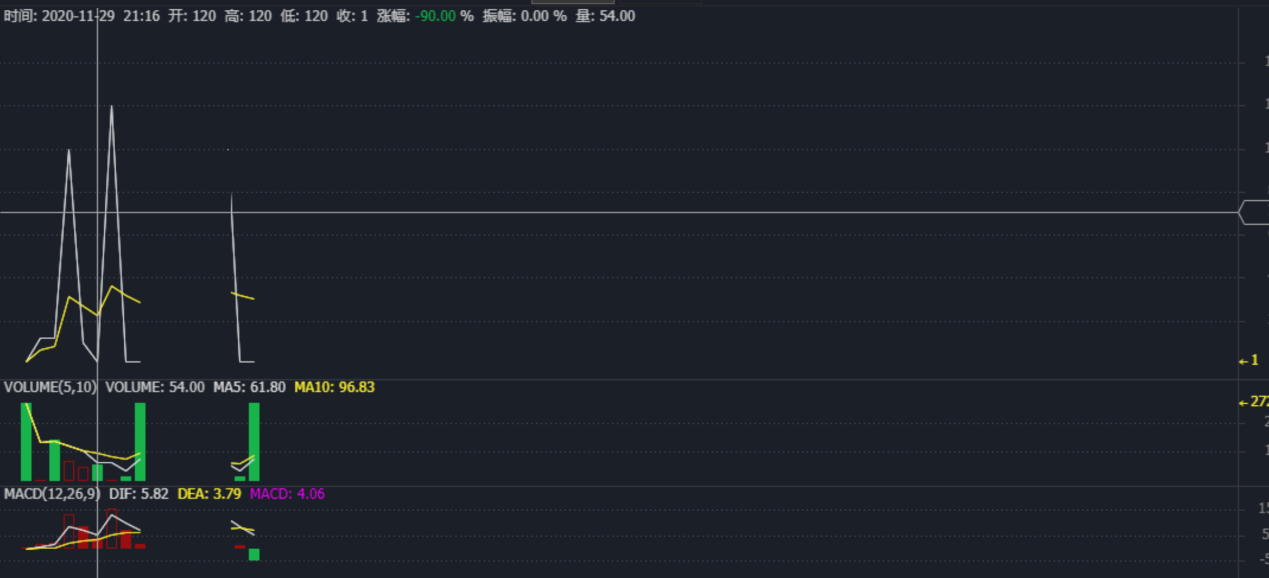
## 14.5 清除Redis 里面的所有K 线



## 14.6 撮合交易测试

在买入和售出选择合适的价格,即可进行撮合,撮合成功后,数据会推送过来

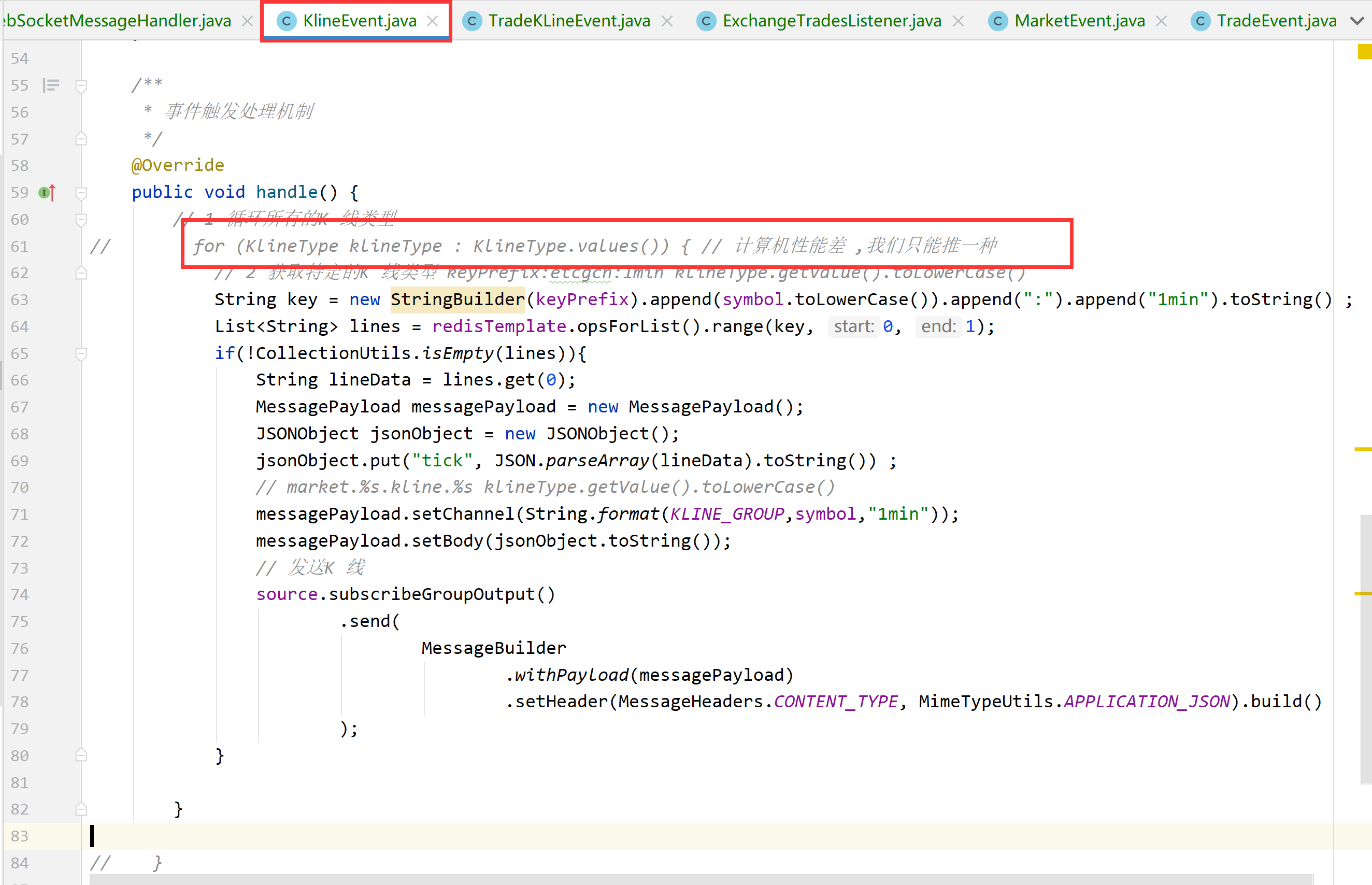




## 14.7 延迟太高的同学

因为运行了9个微服务,导致线程的调度非常慢,因此,K 线的推送速度会很慢.大家一定要关闭电脑上无用的软件,以及关闭多余的服务.

最后,如果实在推送不过来,或者是电脑死机了,可以在这个地方只推一种K 线类型:



# 关于其他的推送服务





## 15.1 市场行情K 线的推送

|  |
| --- |
| */\*\*  \* 行情市场的K 线  \*/* @Component @Slf4j public class MarketEvent implements Event {   @Autowired  private Source source;   @Autowired  private TradingAreaServiceClient tradingAreaServiceClient;    @Autowired  private MarketServiceFeign marketServiceFeign;   private static final String *MARKET\_GROUP* = "market.%s.ticker"; *// %s代表交易区域* private static final String *MARKET\_DETAIL\_GROUP* = "markets.%s.detail" ; *// 交易区域的详情交易数据* @Override  public void handle() {  *// 1 交易区域的查询* List<TradeAreaDto> tradeAreaDtoList = tradingAreaServiceClient.tradeAreas();  if (CollectionUtils.*isEmpty*(tradeAreaDtoList)) {  return;  }  for (TradeAreaDto tradeAreaDto : tradeAreaDtoList) {  *// 2 差该交易区域下的交易数据 , 使用的是交易区域里面市场的id("id1,id2")* List<TradeMarketDto> tradeMarketDtos = marketServiceFeign.queryMarkesByIds(tradeAreaDto.getMarketIds());  if (CollectionUtils.*isEmpty*(tradeMarketDtos)) {  return;  }   JSONObject jsonObject = new JSONObject();  jsonObject.put("markets", tradeMarketDtos);  MessagePayload messagePayload = new MessagePayload();  messagePayload.setChannel(String.*format*(*MARKET\_GROUP*, tradeAreaDto.getCode().toLowerCase()));  messagePayload.setBody(jsonObject.toJSONString());  source.subscribeGroupOutput()  .send(  MessageBuilder  .*withPayload*(messagePayload)  .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*).build()  );  }    *// 获取所有的交易市场* List<MarketDto> marketDtos = marketServiceFeign.tradeMarkets();  if(CollectionUtils.*isEmpty*(marketDtos)){  return;  }  for (MarketDto marketDto : marketDtos) {  List<TradeMarketDto> tradeMarketDtos = marketServiceFeign.queryMarkesByIds(marketDto.getId().toString());   JSONObject jsonObject = new JSONObject();  jsonObject.put("tick",tradeMarketDtos) ;   MessagePayload messagePayload = new MessagePayload();  messagePayload.setChannel(String.*format*(*MARKET\_DETAIL\_GROUP*,marketDto.getSymbol().toLowerCase()));  messagePayload.setBody(jsonObject.toJSONString());  source.subscribeGroupOutput()  .send(  MessageBuilder  .*withPayload*(messagePayload)  .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*).build()  );  }   } } |

## 15.2 成交记录的推送

|  |
| --- |
| */\*\*  \* 成交记录事件  \*/* @Component @Slf4j public class TradeEvent implements Event {   @Autowired  private Source source;   private static final String *TRADE\_GROUP* = "market.%s.trade.detail"; *// %s 代表交易对* @Autowired  private MarketServiceFeign marketServiceFeign;   @Override  public void handle() {  *// 1 获取所有的交易市场* List<MarketDto> marketDtos = marketServiceFeign.tradeMarkets();  if (CollectionUtils.*isEmpty*(marketDtos)) {  return;  }  for (MarketDto marketDto : marketDtos) {  *// 2 查询该交易对下的交易数据* String data = marketServiceFeign.trades(marketDto.getSymbol());   JSONObject jsonObject = new JSONObject();  jsonObject.put("data", data);  MessagePayload messagePayload = new MessagePayload();  messagePayload.setChannel(String.*format*(*TRADE\_GROUP*, marketDto.getSymbol().toLowerCase()));  messagePayload.setBody(jsonObject.toString());  source.subscribeGroupOutput()  .send(  MessageBuilder  .*withPayload*(messagePayload)  .setHeader(MessageHeaders.*CONTENT\_TYPE*, MimeTypeUtils.*APPLICATION\_JSON*).build()  );  }   } } |