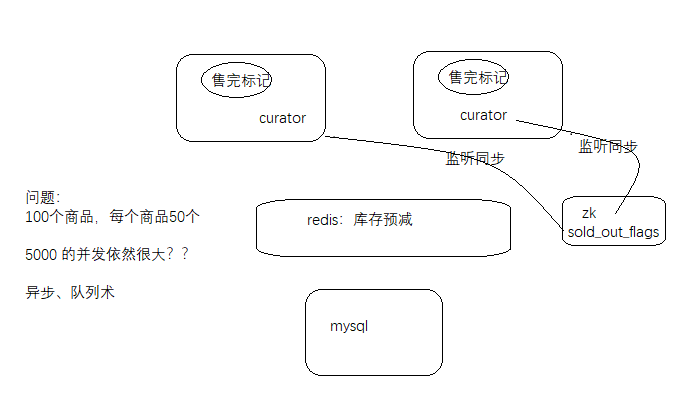
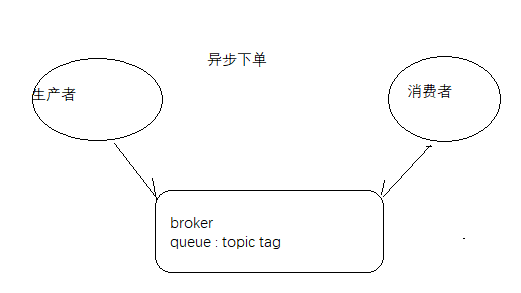
异步下单

# 问题



# MQ



# 业务

## 下单生产

|  |
| --- |
| package com.cjl.skill.mq;  import org.apache.rocketmq.client.producer.DefaultMQProducer;  import org.apache.rocketmq.common.message.Message;  public class MQProducer {  public void sendMessage(String group, String topic, String tag, byte[] message) throws Exception {  DefaultMQProducer producer = new DefaultMQProducer(group);  // Specify name server addresses.  producer.setNamesrvAddr("localhost:9876");  // Launch the instance.  producer.start();  producer.setRetryTimesWhenSendAsyncFailed(0);  Message msg = new Message(topic, tag, message);  producer.send(msg);  producer.shutdown();  }  } |

|  |
| --- |
| // 异步下单  **private** Object createAsyncOrder(Product p, Address address, User user) {  Order record = **new** Order();  record.setNote("秒杀下单测试");  record.setPrice(p.getPrice());  record.setProductId(p.getId());  record.setQuantity(1);  record.setUserId(user.getId());  record.setSum(p.getPrice());  record.setStatus("待付款");  **try** {  // 发送消息  MQProducer producer = **new** MQProducer();  producer.sendMessage("order\_group", "order\_topic", "order\_tag", JsonUtil.*obj2String*(record).getBytes());  System.***out***.println("send order message success.");  **return** **new** AckMessage<>(200,"排队中。。。。。。。。。。");  } **catch** (Exception e) {  // 异常情况，退库存  stringRedisTemplate.opsForValue().increment(ConstantPrefixUtil.***SKILL\_PRODUCT\_PREFIX*** + p.getId());  // 清除售完标记  LocalCache.***SOLD\_OUT\_FLAGS***.remove(String.*valueOf*(p.getId()));  // 删除节点标记  **try** {  client.delete().forPath(ConstantPrefixUtil.***ZK\_SOLD\_OUT\_PRODUCT\_ROOT\_PATH*** + "/" + p.getId());  } **catch** (Exception e1) {  e1.printStackTrace();  }  //清除订单排队标记  stringRedisTemplate.delete(ConstantPrefixUtil.***REDIS\_ORDER\_QUEUE\_FLAG\_PREFIX*** + user.getId() + ":" + p.getId());  **return** AckMessage.*error*("下单失败");  }  } |

## 下单消费

|  |
| --- |
| package com.cjl.skill.mq;  import java.util.List;  import org.apache.rocketmq.client.consumer.DefaultMQPushConsumer;  import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyContext;  import org.apache.rocketmq.client.consumer.listener.ConsumeConcurrentlyStatus;  import org.apache.rocketmq.client.consumer.listener.MessageListenerConcurrently;  import org.apache.rocketmq.common.message.MessageExt;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.CommandLineRunner;  import org.springframework.stereotype.Component;  import com.cjl.skill.pojo.Order;  import com.cjl.skill.service.OrderService;  import com.cjl.skill.util.JsonUtil;  @Component  public class MQConsumer implements CommandLineRunner {  @Autowired  private OrderService orderService;  @Override  public void run(String... args) throws Exception {  /\*\*  \* 启动消费者，然后消费消息，添加订单到数据库  \*/  try {  // Instantiate with specified consumer group name."order\_group", "order\_topic",  // "order\_tag",  DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("order\_group");  // Specify name server addresses.  consumer.setNamesrvAddr("localhost:9876");  // Subscribe one more more topics to consume.  consumer.subscribe("order\_topic", "order\_tag");  // Register callback to execute on arrival of messages fetched from brokers.  consumer.registerMessageListener(new MessageListenerConcurrently() {  @Override  public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs,  ConsumeConcurrentlyContext context) {  for (MessageExt mes : msgs) {  Order order = JsonUtil.string2Obj(new String(mes.getBody()), Order.class);  orderService.createSkillOrder(order);  System.out.printf("%s Receive New Order Messages: %s %n", Thread.currentThread().getName(),  msgs);  }  return ConsumeConcurrentlyStatus.CONSUME\_SUCCESS;  }  });  // Launch the consumer instance.  consumer.start();  System.out.printf("Consumer Started.%n");  } catch (Exception e) {  e.printStackTrace();  System.out.println("consumer started failed.");  }  }  } |

|  |
| --- |
| /\*\*  \* 压测核心业务逻辑：下单、减库存  \*/  @Transactional(isolation =Isolation.***DEFAULT***, propagation = Propagation.***REQUIRED***)  @Override  **public** Order createSkillOrder(Order order) {  **try** {  // 扣减库存时，同时做判断，需要修改sql语句    //syncoostock--; stock = 10; stock = 10-1; 赋值  **if**(productMapper.decreaseStock(order.getProductId())==0) {  order.setId(0);  **return** order;  }  //if ("1".equals("1")) throw new OrderFailException();  // 成功就下单  orderMapper.insertSelective(order);  //在redis缓存里设置一个下单成功标记  stringRedisTemplate.opsForValue().set(  ConstantPrefixUtil.***REDIS\_ORDER\_SUCCESS\_FLAG\_PREFIX***+order.getUserId()+":"+order.getProductId(), JsonUtil.*obj2String*(order));  **return** order;  } **catch** (Exception e) {  //e.printStackTrace();  //logger.debug("order fail with userid {} and productId {}",order.getUserId(),order.getProductId());  logger.error("order fail with userid "+order.getUserId()+" and productId "+order.getProductId()+"\_"+e.getMessage(), e);  **throw** **new** OrderFailException();  }**finally** {  //无论成功还是失败，都清除排队标记  //清除订单排队标记  stringRedisTemplate.delete(ConstantPrefixUtil.***REDIS\_ORDER\_QUEUE\_FLAG\_PREFIX*** + order.getUserId() + ":" + order.getProductId());  }  } |

# 问题

## 友好提示

前端轮询后端的查询订单结果的接口！！

## Bug??

用户发出多次请求，有的请求在队列中，后面的请求如何提示？？

成功？？失败？？

|  |
| --- |
| @PostMapping("/skill")  **public** @ResponseBody Object skill(**int** productId) {  // 做商品售完判断，拦截无效的请求  **if** (LocalCache.***SOLD\_OUT\_FLAGS***.get(String.*valueOf*(productId)) != **null**) {  **return** **new** AckMessage<>(603, "商品已经抢完了");  }  // 后台校验秒杀时间  **if** (!validSkillTime(productId)) {  **return** AckMessage.*error*("time is error");  }  // 验证参数  **if** (productId <= 0) {  **return** AckMessage.*illegalArgs*();  }  // 验证是否登陆  User user = getLoginUser();  **if** (user == **null**) {  **return** AckMessage.*unauthorized*();  }  // 是否有默认收货地址  Address address = getUserDefaultAddress(user);  **if** (address == **null**) {  **return** **new** AckMessage<>(601, "没有默认收货地址");  }  // 一个人一个商品只能下一次单  String json = stringRedisTemplate.opsForValue()  .get(ConstantPrefixUtil.***REDIS\_ORDER\_SUCCESS\_FLAG\_PREFIX*** + user.getId() + ":" + productId);  **if** (json != **null**) {  **return** **new** AckMessage<>(201, "亲，该商品已经秒杀过了");  }  //设置redis订单排队标记，分布式锁功能，保证同一用户同一商品只能秒杀成功一次，没有抢到锁的同学，说明前面自己已经在排队了  //设置超时时间防止死锁  **if** (!stringRedisTemplate.opsForValue().setIfAbsent(ConstantPrefixUtil.***REDIS\_ORDER\_QUEUE\_FLAG\_PREFIX*** + user.getId() + ":" + productId,  "queue", 60, TimeUnit.***SECONDS***)) {  **return** **new** AckMessage<>(200, "亲，您正在排队中，请耐心等待哦");  }  // redis来减轻数据库的压力：10w QPS，原子减，串行执行  Long result = stringRedisTemplate.opsForValue().decrement(ConstantPrefixUtil.***SKILL\_PRODUCT\_PREFIX*** + productId);  **if** (result < 0) {  // 还原负数  stringRedisTemplate.opsForValue().increment(ConstantPrefixUtil.***SKILL\_PRODUCT\_PREFIX*** + productId);  // 添加售完标记  LocalCache.***SOLD\_OUT\_FLAGS***.put(String.*valueOf*(productId), **true**);  // 创建zk售完标记节点  createZkNode(productId);  **return** **new** AckMessage<>(603, "商品已经抢完了");  }  // 商品是否存在  Product product = productService.getById(productId);  **if** (product == **null**) {  **return** **new** AckMessage<>(602, "商品不存在");  }  // 检查库存  **if** (product.getStock() < 1) {  **return** **new** AckMessage<>(603, "商品已经抢完了");  }  // 同步生成订单  // return createOrder(product, address, user);  // 异步下单，成功排队  **return** createAsyncOrder(product, address, user);  } |