**畅购电商系统开发**

**第14天**

传智播客.黑马程序员.深圳

# 学习目标

* 防止秒杀重复排队

重复排队：一个人抢购商品，如果没有支付，不允许重复排队抢购

* 并发超卖问题解决
* 秒杀订单支付

秒杀支付：支付流程需要调整

* 超时支付订单库存回滚

1.RabbitMQ延时队列

2.利用延时队列实现支付订单的监听，根据订单支付状况进行订单数据库回滚

# 防止秒杀重复排队

用户每次抢单的时候，一旦排队，我们设置一个自增值，让该值的初始值为1，每次进入抢单的时候，对它进行递增，如果值>1，则表明已经排队,不允许重复排队,如果重复排队，则对外抛出异常，并抛出异常信息20005表示已经正在排队。

## 后台排队记录

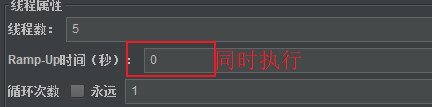
修改SeckillOrderServiceImpl的add方法，新增递增值判断是否排队中，代码如下：

@Override  
**public boolean** add(Long id, String time, String username){  
 *//System.out.println("进入了抢购业务方法，开始异步调用...");  
  
 //使用Redis的自增统计用户下单次数，第一次是1，每次增1，支付后，要重置为0* Long count = **redisTemplate**.boundHashOps(**"UserQueueCount"**).increment(username, 1);  
 **if**(count > 1){  
 **throw new** RuntimeException(StatusCode.***REPERROR*** + **""**);  
 }  
 *//排队信息封装* SeckillStatus seckillStatus = **new** SeckillStatus(username, **new** Date(),1, id,time);  
 *//将秒杀抢单信息存入到Redis中,这里采用List方式存储,List本身是一个分布式队列* **redisTemplate**.boundListOps(**"SeckillOrderQueue"**).leftPush(seckillStatus);  
 *//记录用户排队信息* **redisTemplate**.boundHashOps(**"UserQueueStatus"**).put(username,seckillStatus);  
 *//调用异步测试* **multiThreadingCreateOrder**.createOrder();  
 *//System.out.println("进入了抢购业务方法，结束异步调用...");* **return true**;  
}

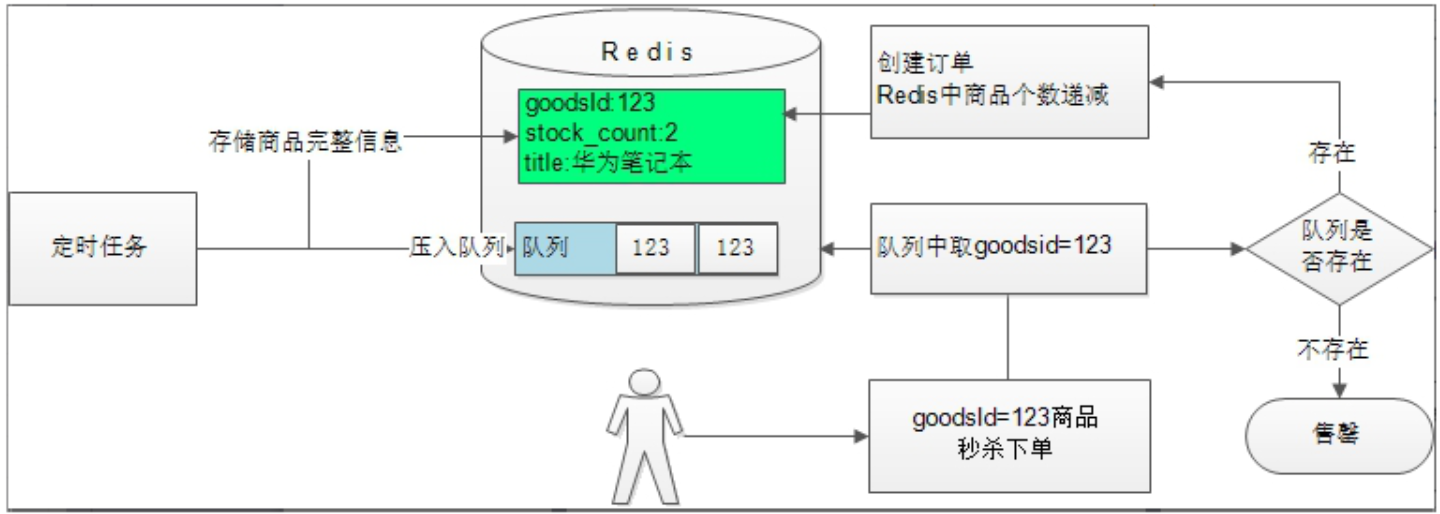
# 并发超卖问题解决

超卖问题，这里是指多人抢购同一商品的时候，多人同时判断是否有库存，如果只剩一个，则都会判断有库存，此时会导致超卖现象产生，也就是一个商品下了多个订单的现象。

超卖测试：测试前先把上面写好的限制用户能只下一个订单功能先注释掉🡪通过jmeter发起多个并发下单请求，大家看看，最终商品是不是会超卖或者库存计算不正确！



## 思路分析



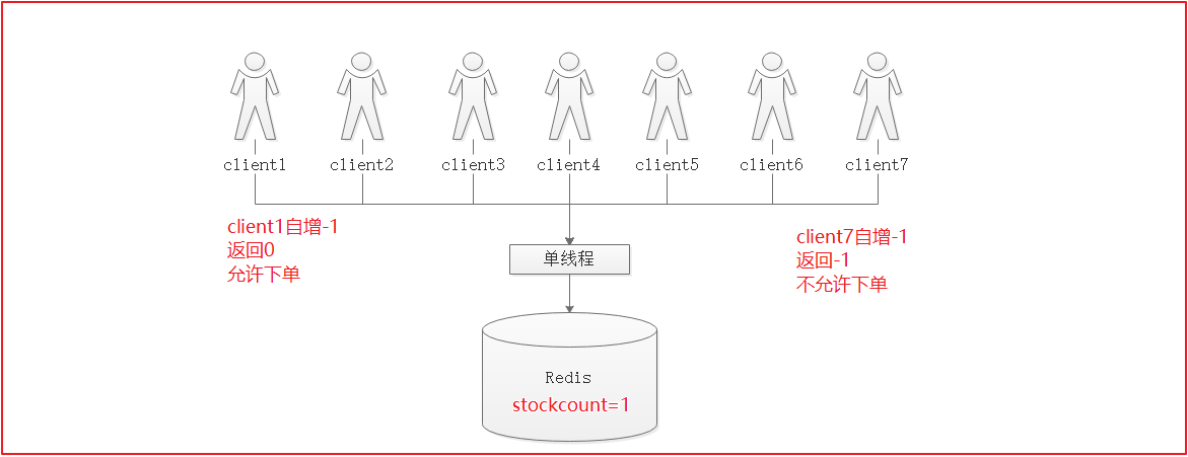
**方式一：**

解决超卖问题，可以利用Redis队列实现，给每件商品创建一个独立的商品个数队列，例如：A商品有2个，A商品的ID为1001，则可以创建一个队列,key=SeckillGoodsCountList\_1001,往该队列中塞2次该商品ID。

每次给用户下单的时候，先从队列中取数据，如果能取到数据，则表明有库存，如果取不到，则表明没有库存，这样就可以防止超卖问题产生了。

**方式二(推荐)：**

在我们队Redis进行操作的时候，很多时候，都是先将数据查询出来，在内存中修改，然后存入到Redis，在并发场景，会出现数据错乱问题，为了控制数量准确，我们单独将商品数量整一个自增键，自增键是线程安全的，所以不担心并发场景的问题。



## 方式一实现

每次将商品压入Redis缓存的时候，另外多创建一个商品的队列。

### 修改SeckillGoodsPushTask

添加一个pushIds方法，并将商品信息存入队列，代码如下：

*/\*\*  
 \* 30秒执行一次  
 \*/*@Scheduled(cron = **"0/5 \* \* \* \* \*"**)  
**public void** loadGoodsPushRedis() {  
 *//System.out.println("定时任务被调度了...");  
 //获取时间段集合* List<Date> dateMenus = DateUtil.*getDateMenus*();  
 *//循环时间段* **for** (Date startTime : dateMenus) {  
 *// namespace = SeckillGoods\_20195712* String extName = DateUtil.*data2str*(startTime, DateUtil.***PATTERN\_YYYYMMDDHH***);  
  
 *//根据时间段数据查询对应的秒杀商品数据* Example example = **new** Example(SeckillGoods.**class**);  
 Example.Criteria criteria = example.createCriteria();  
 *// 1)商品必须审核通过 status=1* criteria.andEqualTo(**"status"**, **"1"**);  
 *// 2)库存>0* criteria.andGreaterThan(**"stockCount"**, 0);  
 *// 3)开始时间<=活动开始时间* criteria.andGreaterThanOrEqualTo(**"startTime"**, startTime);  
 *// 4)活动结束时间<开始时间+2小时* criteria.andLessThan(**"endTime"**, DateUtil.*addDateHour*(startTime, 2));  
 *// 5)排除之前已经加载到Redis缓存中的商品数据* Set keys = **redisTemplate**.boundHashOps(**"SeckillGoods\_"** + extName).keys();  
 **if** (keys != **null** && keys.size() > 0) {  
 criteria.andNotIn(**"id"**, keys);  
 }  
 *//查询数据* List<SeckillGoods> seckillGoods = **seckillGoodsMapper**.selectByExample(example);  
  
 *//System.out.println(extName + "时段导入商品个数为：" + seckillGoods.size());  
 //将秒杀商品数据存入到Redis缓存* **for** (SeckillGoods seckillGood : seckillGoods) {  
 *//商品基本信息* **redisTemplate**.boundHashOps(**"SeckillGoods\_"** + extName).put(seckillGood.getId(), seckillGood);  
  
 *//方式一：商品数据队列存储,防止高并发超卖* Long[] ids = pushIds(seckillGood.getStockCount(), seckillGood.getId());  
 **redisTemplate**.boundListOps(**"SeckillGoodsCountList\_"**+seckillGood.getId()).leftPushAll(ids);  
 }  
 }  
}  
*/\*\*\*  
 \* 将商品ID存入到数组中  
 \** ***@param len****:长度  
 \** ***@param id*** *:值  
 \*/***public** Long[] pushIds(**int** len,Long id){  
 Long[] ids = **new** Long[len];  
 **for** (**int** i = 0; i <ids.**length** ; i++) {  
 ids[i]=id;  
 }  
 **return** ids;  
}

### 修改MultiThreadingCreateOrder.createOrder

**public void** createOrder(){  
 *//从队列中获取排队信息-左进右出* SeckillStatus seckillStatus = (SeckillStatus) **redisTemplate**.boundListOps(**"SeckillOrderQueue"**).rightPop();  
  
 *//超卖方式一：队列-只能处理超卖问题，不能解决正确库存数显示问题* Object sgoods = **redisTemplate**.boundListOps(**"SeckillGoodsCountList\_"** + seckillStatus.getGoodsId()).rightPop();  
 *//如果队列没数据，说明没有库存了* **if**(sgoods == **null**){  
 *//清理排队标示* **redisTemplate**.boundHashOps(**"UserQueueCount"**).delete(seckillStatus.getUsername());  
 *//清理抢单标示* **redisTemplate**.boundHashOps(**"UserQueueStatus"**).delete(seckillStatus.getUsername());  
 **return**;  
 }  
 *//省略其它代码………………….*   
 }  
}

## 方式二实现

### 修改SeckillGOodsPushTask

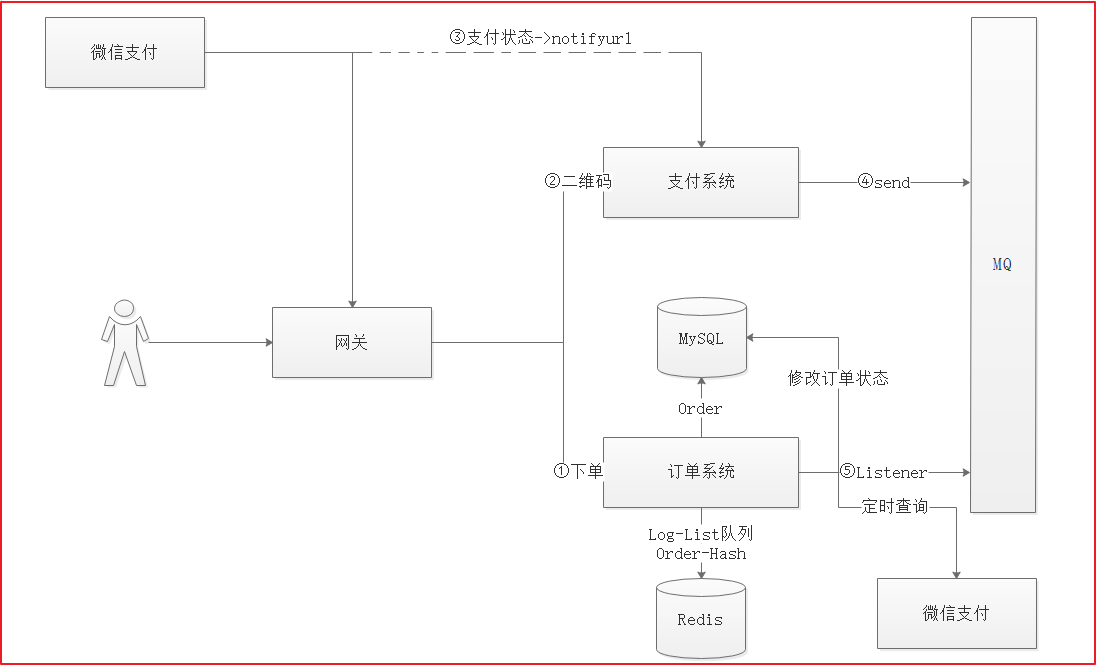
@Scheduled(cron = **"0/5 \* \* \* \* \*"**)  
**public void** loadGoodsPushRedis() {  
 *//System.out.println("定时任务被调度了...");  
 //获取时间段集合* List<Date> dateMenus = DateUtil.*getDateMenus*();  
 *//循环时间段* **for** (Date startTime : dateMenus) {  
 *// namespace = SeckillGoods\_20195712* String extName = DateUtil.*data2str*(startTime, DateUtil.***PATTERN\_YYYYMMDDHH***);  
  
 *//根据时间段数据查询对应的秒杀商品数据* Example example = **new** Example(SeckillGoods.**class**);  
 Example.Criteria criteria = example.createCriteria();  
 *// 1)商品必须审核通过 status=1* criteria.andEqualTo(**"status"**, **"1"**);  
 *// 2)库存>0* criteria.andGreaterThan(**"stockCount"**, 0);  
 *// 3)开始时间<=活动开始时间* criteria.andGreaterThanOrEqualTo(**"startTime"**, startTime);  
 *// 4)活动结束时间<开始时间+2小时* criteria.andLessThan(**"endTime"**, DateUtil.*addDateHour*(startTime, 2));  
 *// 5)排除之前已经加载到Redis缓存中的商品数据* Set keys = **redisTemplate**.boundHashOps(**"SeckillGoods\_"** + extName).keys();  
 **if** (keys != **null** && keys.size() > 0) {  
 criteria.andNotIn(**"id"**, keys);  
 }  
 *//查询数据* List<SeckillGoods> seckillGoods = **seckillGoodsMapper**.selectByExample(example);  
  
 *//System.out.println(extName + "时段导入商品个数为：" + seckillGoods.size());  
 //将秒杀商品数据存入到Redis缓存* **for** (SeckillGoods seckillGood : seckillGoods) {  
 *//商品基本信息* **redisTemplate**.boundHashOps(**"SeckillGoods\_"** + extName).put(seckillGood.getId(), seckillGood);  
  
 *//方式一：商品数据队列存储,防止高并发超卖* Long[] ids = pushIds(seckillGood.getStockCount(), seckillGood.getId());  
 **redisTemplate**.boundListOps(**"SeckillGoodsCountList\_"** + seckillGood.getId()).leftPushAll(ids);  
  
 *//方式二：使用加redis的自减，先把库存存到Redis* **redisTemplate**.boundHashOps(**"SeckillGoodsCount"**).increment(seckillGood.getId(),seckillGood.getStockCount());  
 }  
 }  
}

### 修改MultiThreadingCreateOrder.createOrder

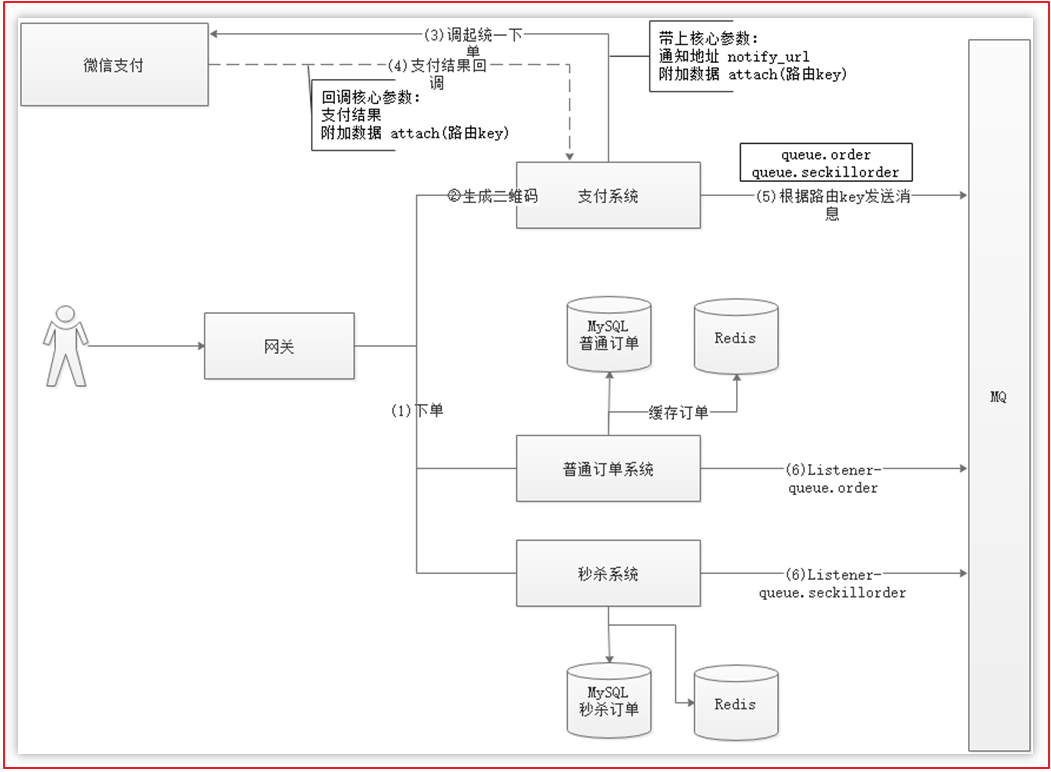
**public void** createOrder(){  
 *//从队列中获取排队信息-左进右出* SeckillStatus seckillStatus = (SeckillStatus) **redisTemplate**.boundListOps(**"SeckillOrderQueue"**).rightPop();  
  
 *//超卖方式一：队列  
 /\*Object sgoods = redisTemplate.boundListOps("SeckillGoodsCountList\_" + seckillStatus.getGoodsId()).rightPop();  
 //如果队列没数据，说明没有库存了  
 if(sgoods == null){  
 //清理排队标示  
 redisTemplate.boundHashOps("UserQueueCount").delete(seckillStatus.getUsername());  
 //清理抢单标示  
 redisTemplate.boundHashOps("UserQueueStatus").delete(seckillStatus.getUsername());  
 return;  
 }\*/  
  
 //超卖方式二：库存精确显示-自减* Long count = **redisTemplate**.boundHashOps(**"SeckillGoodsCount"**).increment(seckillStatus.getGoodsId(), -1);  
  
 *//如果有排队信息* **if**(seckillStatus != **null**){  
 *//时间区间* String time = seckillStatus.getTime();  
 *//用户登录名* String username=seckillStatus.getUsername();  
 *//用户抢购商品* Long id = seckillStatus.getGoodsId();  
  
 *//1.获取商品数据* SeckillGoods goods = (SeckillGoods) **redisTemplate**.boundHashOps(**"SeckillGoods\_"** + time).get(id);  
  
 *//2.如果没有库存，则直接抛出异常* **if** (goods == **null** || count <= 0) {  
 **throw new** RuntimeException(**"你来晚了一步，商品已抢购一空!"**);  
 }  
 *//3.如果有库存，则创建秒杀商品订单* SeckillOrder seckillOrder = **new** SeckillOrder();  
 seckillOrder.setId(**idWorker**.nextId());  
 seckillOrder.setSeckillId(id);  
 seckillOrder.setMoney(goods.getCostPrice());  
 seckillOrder.setUserId(username);  
 seckillOrder.setCreateTime(**new** Date());  
 seckillOrder.setStatus(**"0"**);  
 *//将秒杀订单存入到Redis中* **redisTemplate**.boundHashOps(**"SeckillOrder"**).put(username,seckillOrder);  
  
 *//4.扣减库存* goods.setStockCount(count.intValue());  
  
 *//5.判断当前商品是否还有库存* **if** (count <= 0) {  
 *//并且将商品数据同步到MySQL中* **seckillGoodsMapper**.updateByPrimaryKeySelective(goods);  
 *//如果没有库存,则清空Redis缓存中该商品  
 //redisTemplate.boundHashOps("SeckillGoods\_" + time).delete(id);* } **else** {  
 *//如果有库存，则将扣减库存后的goods重新放入redis* **redisTemplate**.boundHashOps(**"SeckillGoods\_"** + time).put(id, goods);  
 }  
  
 *//抢单成功，更新抢单状态,排队->等待支付* seckillStatus.setStatus(2); *//1:排队中，2:秒杀等待支付,3:支付超时，4:秒杀失败,5:支付完成* seckillStatus.setOrderId(seckillOrder.getId()); *//更新订单id* seckillStatus.setMoney(**new** Float(seckillOrder.getMoney())); *//记录金额  
 //更新用户订单排队信息为等待支付* **redisTemplate**.boundHashOps(**"UserQueueStatus"**).put(username,seckillStatus);  
 }  
}

# 订单支付

在第12天的课程中我们已经实现了支付功能，流程如下：

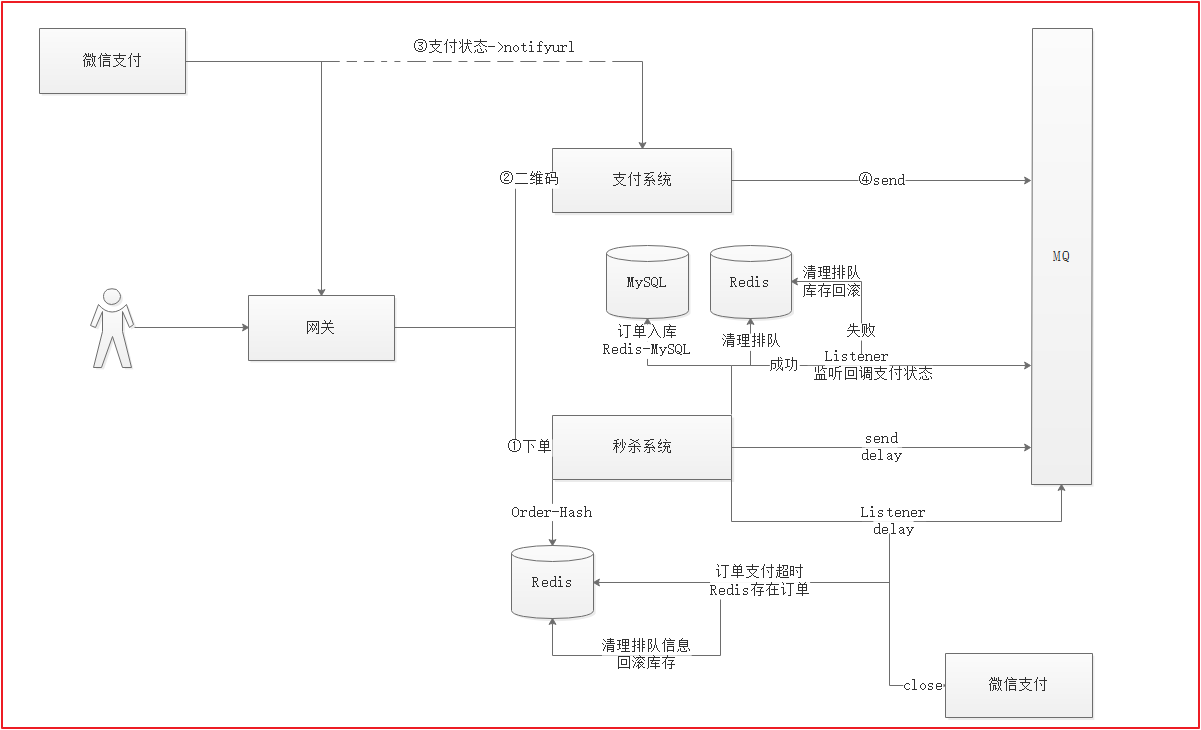


但之前实现的是订单信息的回调数据发送给MQ，指定了普通订单的队列，不过现在需要实现的是秒杀信息发送给秒杀队列，所以之前的代码那块需要动态指定队列，解决思路调整如下：



## 支付流程分析

细化上面的设计流程后，逻辑如下：



如上图，步骤分析如下：

1.用户抢单，经过秒杀系统实现抢单，下单后会将向MQ发送一个延时队列消息，包含抢单信息，延时半小时后才能监听到

2.秒杀系统同时启用延时消息监听，一旦监听到订单抢单信息，判断Redis缓存中是否存在订单信息，如果存在，则回滚

3.秒杀系统还启动支付回调信息监听，如果支付完成，则将订单吃句话到MySQL，如果没完成，清理排队信息回滚库存

4.每次秒杀下单后调用支付系统，创建二维码，如果用户支付成功了，微信系统会将支付信息发送给支付系统指定的回调地址，支付系统收到信息后，将信息发送给MQ，第3个步骤就可以监听到消息了。

## 支付回调改造

1.创建支付生成二维码需要指定路由key

2.回调地址回调的时候，获取支付二维码指定的路由key，将支付信息发送到指定路由key中

在微信支付统一下单API中，有一个附加参数,如下：

attach:附加数据,String(127)，在查询API和支付通知中原样返回，可作为自定义参数使用。

我们可以在创建二维码的时候，指定该参数，该参数用于指定回调支付信息的对应队列，每次回调的时候，会获取该参数，然后将回调信息发送到该参数对应的队列去。

### 修改支付系统application.yml

*#位置支付交换机和队列***mq**:  
 **pay**:  
 **exchange**:  
 **order**: exchange.order  
 **queue**:  
 **order**: queue.order  
 **seckillorder**: queue.seckillorder  
 **routing**:  
 **key**: queue.order  
 **seckillkey**: queue.seckillorder

### 修改WeixinPayApplication

修改WeixinPayApplication，添加对应队列以及对应交换机绑定，代码如下：

@SpringBootApplication(exclude={DataSourceAutoConfiguration.**class**})  
@EnableEurekaClient  
**public class** WeixinPayApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(WeixinPayApplication.**class**,args);  
 }  
 @Autowired  
 **private** Environment **env**;  
  
 *//创建交换机* @Bean  
 **public** DirectExchange orderExchange(){  
 **return new** DirectExchange(**env**.getProperty(**"mq.pay.exchange.order"**), **true**, **false**);  
 }  
  
 */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*普通订单队列与路由key\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 //创建队列* @Bean  
 **public** Queue orderQueue(){  
 **return new** Queue(**env**.getProperty(**"mq.pay.queue.order"**));  
 }  
 */\*\*  
 \* 队列绑定交换机  
 \** ***@Qualifier*** *可以指定spring bean的id,默认情况下id就是方法名  
 \** ***@return*** *\*/* @Bean  
 **public** Binding bindingOrder(@Qualifier(**"orderQueue"**) Queue orderQueue, DirectExchange orderExchange){  
 **return** BindingBuilder.*bind*(orderQueue).to(orderExchange).with(**env**.getProperty(**"mq.pay.routing.key"**));  
 }  
  
 */\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*秒杀订单队列与路由key\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 //创建队列* @Bean  
 **public** Queue seckillQueue(){  
 **return new** Queue(**env**.getProperty(**"mq.pay.queue.seckillorder"**));  
 }  
 */\*\*  
 \* 队列绑定交换机  
 \** ***@Qualifier*** *可以指定spring bean的id,默认情况下id就是方法名  
 \** ***@return*** *\*/* @Bean  
 **public** Binding bindingSeckillOrder(@Qualifier(**"seckillQueue"**) Queue seckillQueue, DirectExchange orderExchange){  
 **return** BindingBuilder.*bind*(seckillQueue).to(orderExchange).with(**env**.getProperty(**"mq.pay.routing.seckillkey"**));  
 }  
}

### 创建队列测试

在com.changgou.pay.controller.WeixinPayController，添加测试方法，用于创建队列

@RequestMapping(**"createSeckillQueue"**)  
**public** String createSeckillQueue(){  
 *//发送MQ消息-用于创建秒杀队列* **rabbitTemplate**.convertAndSend(**"exchange.order"**,**"queue.seckillorder"**, **"{'flag':'ok'}"**);  
 **return "ok"**;  
}

访问：<http://localhost:18090/weixin/pay/createSeckillQueue>

### 修改生成二维码service

修改支付微服务的WeixinPayService的createNative方法，代码如下：

*/\*\*  
 \* 生成微信支付二维码  
 \** ***@param paramMap*** *{  
 \* out\_trade\_no 订单号,  
 \* total\_fee 金额(分),  
 \* exchange 交换机,  
 \* routingKey 路由Key  
 \* }  
 \** ***@return*** *\*/***public** Map createNative(Map<String,String> paramMap);

修改支付微服务的WeixinPayServiceImpl的createNative方法，代码如下：

@Override  
**public** Map createNative(Map<String,String> paramMap) {  
 Map map = **new** HashMap();  
 **try** {  
 *//1、包装微信接口需要的参数* Map param = **new** HashMap();  
 param.put(**"appid"**, **appid**); *//公众号ID* param.put(**"mch\_id"**, **partner**); *//商户号* param.put(**"nonce\_str"**, WXPayUtil.*generateNonceStr*()); *//随机字符串* param.put(**"body"**, **"畅购"**); *//商品描述，扫码后用户看到的商品信息* param.put(**"out\_trade\_no"**, paramMap.get(**"out\_trade\_no"**)); *//订单号* param.put(**"total\_fee"**, paramMap.get(**"total\_fee"**)); *//订单总金额，单位为分* param.put(**"spbill\_create\_ip"**, **"127.0.0.1"**); *//终端IP，只要附合ip地址规范，可以随意写* param.put(**"notify\_url"**, **notifyurl**); *//回调地址* param.put(**"trade\_type"**, **"NATIVE"**); *//交易类型，NATIVE 扫码支付  
   
 //附加参数* Map<String, String> attachMap = **new** HashMap<String,String>();  
 attachMap.put(**"exchange"**, paramMap.get(**"exchange"**));  
 attachMap.put(**"routingKey"**, paramMap.get(**"routingKey"**));  
 param.put(**"attach"**, JSON.*toJSONString*(attachMap));  
  
 *//2、生成xml，通过httpClient发送请求得到数据* String xmlParam = WXPayUtil.*generateSignedXml*(param, **partnerkey**);  
 System.***out***.println(**"正在调起统一下单接口，请求参数:"** + xmlParam);  
 HttpClient httpClient = **new** HttpClient(**"https://api.mch.weixin.qq.com/pay/unifiedorder"**);  
 httpClient.setHttps(**true**);  
 httpClient.setXmlParam(xmlParam);  
 httpClient.post();  
 *//3、解析结果* String xmlResult = httpClient.getContent();  
 System.***out***.println(**"调起统一下单接口成功，返回结果："** + xmlResult);  
 Map<String, String> resultMap = WXPayUtil.*xmlToMap*(xmlResult);  
 map.put(**"code\_url"**, resultMap.get(**"code\_url"**));*//支付地址* map.put(**"total\_fee"**, paramMap.get(**"total\_fee"**));*//总金额* map.put(**"out\_trade\_no"**,paramMap.get(**"out\_trade\_no"**));*//订单号* } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return** map;  
}

### 修改生成二维码controller

修改支付微服务的WeixinPayController的createNative方法，代码如下：

*/\*\*\*  
 \* 创建二维码  
 \** ***@param paramMap*** *{  
 \* out\_trade\_no 订单号,  
 \* total\_fee 金额(分),  
 \* exchange 交换机,  
 \* routingKey 路由Key  
 \* }  
 \** ***@return*** *\*/*@RequestMapping(value = **"/create/native"**)  
**public** Result createNative(@RequestParam Map<String,String> paramMap){  
 Map<String,String> resultMap = **weixinPayService**.createNative(paramMap);  
 **return new** Result(**true**, StatusCode.***OK***,**"创建二维码预付订单成功！"**,resultMap);  
}

### 改造支付回调方法

修改com.changgou.pay.controller.WeixinPayController的notifyUrl方法，获取自定义参数，并转成Map，获取交换机与路由key

*/\*\*\*  
 \* 支付回调  
 \* 支付完成后，微信会把相关支付结果及用户信息通过数据流的形式发送给商户，  
 \* 商户需要接收处理，并按文档规范返回应答  
 \** ***@param request*** *\** ***@return*** *\*/*@RequestMapping(value = **"/notify/url"**)  
**public** String notifyUrl(HttpServletRequest request){  
 **try** {  
 *//1.读取支付回调数据* InputStream inStream = request.getInputStream();  
 *//2.使用Apache IOUtils把输入转换成字符* String result = IOUtils.*toString*(inStream,**"UTF-8"**);  
 *//将xml字符串转换成Map结构* Map<String, String> map = WXPayUtil.*xmlToMap*(result);  
 System.***out***.println(map);  
  
 *//读取附加消息-交换机与队列* Map<String,String> attachMap = JSON.*parseObject*(map.get(**"attach"**), Map.**class**);  
 **exchange** = attachMap.get(**"exchange"**);  
 **routing** = attachMap.get(**"routingKey"**);  
  
 *//发送MQ消息* **rabbitTemplate**.convertAndSend(**exchange**,**routing**, JSON.*toJSONString*(map));  
  
 *//3.包装响应数据设置* Map respMap = **new** HashMap();  
 respMap.put(**"return\_code"**,**"SUCCESS"**);  
 respMap.put(**"return\_msg"**,**"OK"**);  
 **return** WXPayUtil.*mapToXml*(respMap);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 *//记录错误日志* }  
 **return null**;  
}

## 支付状态监听

支付状态通过回调地址发送给MQ之后，我们需要在秒杀系统中监听支付信息，如果用户已支付，则修改用户订单状态，如果支付失败，则直接删除订单，回滚库存。

### 监听器实现

在秒杀工程中创建com.changgou.seckill.consumer.SeckillOrderPayMessageListener,实现监听消息，代码如下:

此处代码与我们之前在订单模块写的逻辑基本一致，所以可以把那代码复制过来改改。

*/\*\*  
 \* 秒杀支付消息监听器  
 \** ***@author*** *Steven  
 \** ***@description*** *com.changgou.seckill.consumer  
 \*/*@Component  
**public class** SeckillOrderPayMessageListener {  
  
 @RabbitListener(queues = **"${mq.pay.queue.seckillorder}"**)  
 **public void** payListener(String msg) {  
 *//将数据转成Map* Map<String, String> result = JSON.*parseObject*(msg, Map.**class**);  
 System.***out***.println(**"收到消息，参数为："** + result);  
 *//return\_code=SUCCESS* String return\_code = result.get(**"return\_code"**);  
  
 *//返回状态码 result\_code=SUCCESS/FAIL，修改订单状态* **if** (**"success"**.equalsIgnoreCase(return\_code)) {  
 *//业务结果* String result\_code = result.get(**"result\_code"**);  
 *//获取订单号* String out\_trade\_no = result.get(**"out\_trade\_no"**);  
 *//交易流水号* String transaction\_id = result.get(**"transaction\_id"**);  
 *//业务结果-SUCCESS/FAIL，为success时，支付成功* **if** (**"success"**.equalsIgnoreCase(result\_code)) {  
 *//修改订单状态 out\_trade\_no* } **else** {  
 *//订单删除* }  
 }  
 }  
}

### 修改秒杀系统的application.yml文件，添加如下配置：

*#位置支付交换机和队列***mq**:  
 **pay**:  
 **exchange**:  
 **order**: exchange.order  
 **queue**:  
 **order**: queue.order  
 **seckillorder**: queue.seckillorder  
 **routing**:  
 **key**: queue.order  
 **seckillkey**: queue.seckillorder

### 添加amqp依赖

*<!--加入amqp-->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-amqp</**artifactId**>  
</**dependency**>

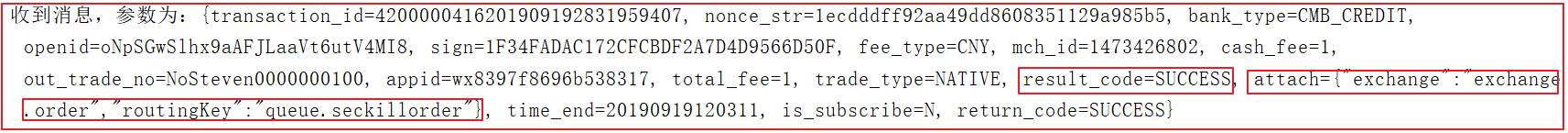
## 秒杀支付测试

**生成二维码访问：**<http://localhost:18090/weixin/pay/create/native?out_trade_no=NoSteven0000000100&total_fee=1&exchange=exchange.order&routingKey=queue.seckillorder>

**二维码展示，并扫码支付：**



**在秒杀支付监听器中，可以收到消息：**



## 修改订单状态

监听到支付信息后，根据支付信息判断，如果用户支付成功，则修改订单信息，并将订单入库，删除用户排队信息，如果用户支付失败，则删除订单信息，回滚库存，删除用户排队信息。

### 监听器获取用户名

由于我们秒杀订单存在Redis中并且key为用户名，我们在监听器是无法获取当前登录名的，只能从生成二维码那一刻包装到附加参数中，再带到监听器。

修改支付微服务的WeixinPayServiceImpl的createNative方法，代码如下：

@Override  
**public** Map createNative(Map<String,String> paramMap) {  
 Map map = **new** HashMap();  
 **try** {  
 *//1、包装微信接口需要的参数* Map param = **new** HashMap();  
 param.put(**"appid"**, **appid**); *//公众号ID* param.put(**"mch\_id"**, **partner**); *//商户号* param.put(**"nonce\_str"**, WXPayUtil.*generateNonceStr*()); *//随机字符串* param.put(**"body"**, **"畅购"**); *//商品描述，扫码后用户看到的商品信息* param.put(**"out\_trade\_no"**, paramMap.get(**"out\_trade\_no"**)); *//订单号* param.put(**"total\_fee"**, paramMap.get(**"total\_fee"**)); *//订单总金额，单位为分* param.put(**"spbill\_create\_ip"**, **"127.0.0.1"**); *//终端IP，只要附合ip地址规范，可以随意写* param.put(**"notify\_url"**, **notifyurl**); *//回调地址* param.put(**"trade\_type"**, **"NATIVE"**); *//交易类型，NATIVE 扫码支付  
  
 //附加参数* Map<String, String> attachMap = **new** HashMap<String,String>();  
 attachMap.put(**"exchange"**, paramMap.get(**"exchange"**));  
 attachMap.put(**"routingKey"**, paramMap.get(**"routingKey"**));  
 attachMap.put(**"username"**, paramMap.get(**"username"**));  
 param.put(**"attach"**, JSON.*toJSONString*(attachMap));  
  
 *//2、省略其它代码……………../*

**return** map;  
}

修改支付微服务的WeixinPayController的createNative方法，代码如下：

@RequestMapping(value = **"/create/native"**)  
**public** Result createNative(@RequestParam Map<String,String> paramMap){  
 *//paramMap.put("username", TokenDecode.getUserInfo().get("username"));* paramMap.put(**"username"**, **"zhangsan"**);  
 Map<String,String> resultMap = **weixinPayService**.createNative(paramMap);  
 **return new** Result(**true**, StatusCode.***OK***,**"创建二维码预付订单成功！"**,resultMap);  
}

修改秒杀微服务的支付监听器com.changgou.seckill.consumer.SeckillOrderPayMessageListener，获取用户名

@Component  
**public class** SeckillOrderPayMessageListener {  
  
 @RabbitListener(queues = **"${mq.pay.queue.seckillorder}"**)  
 **public void** payListener(String msg) {  
 *//将数据转成Map* Map<String, String> result = JSON.*parseObject*(msg, Map.**class**);  
 System.***out***.println(**"收到消息，参数为："** + result);  
 *//return\_code=SUCCESS* String return\_code = result.get(**"return\_code"**);  
  
 *//返回状态码 result\_code=SUCCESS/FAIL，修改订单状态* **if** (**"success"**.equalsIgnoreCase(return\_code)) {  
 *//业务结果* String result\_code = result.get(**"result\_code"**);  
 *//获取订单号* String out\_trade\_no = result.get(**"out\_trade\_no"**);  
 *//交易流水号* String transaction\_id = result.get(**"transaction\_id"**);  
  
 *//附加参数* Map<String, String> attachMap = JSON.*parseObject*(result.get(**"attach"**), Map.**class**);  
 *//用户名* String username = attachMap.get(**"username"**);  
  
 *//业务结果-SUCCESS/FAIL，为success时，支付成功* **if** (**"success"**.equalsIgnoreCase(result\_code)) {  
 *//修改订单状态 out\_trade\_no* } **else** {  
 *//订单删除* }  
 }  
 }  
}

### 业务层

修改SeckillOrderService，添加修改订单方法，代码如下

*/\*\*\*  
 \* 更新订单状态  
 \** ***@param out\_trade\_no*** *\** ***@param transaction\_id*** *\** ***@param username*** *\*/***void** updatePayStatus(String out\_trade\_no, String transaction\_id,String username);

修改SeckillOrderServiceImpl，添加修改订单方法实现，代码如下：

*/\*\*\*  
 \* 更新订单状态  
 \** ***@param out\_trade\_no*** *\** ***@param transaction\_id*** *\** ***@param username*** *\*/*@Override  
**public void** updatePayStatus(String out\_trade\_no, String transaction\_id,String username) {  
 *//订单数据从Redis数据库查询出来* SeckillOrder seckillOrder = (SeckillOrder) **redisTemplate**.boundHashOps(**"SeckillOrder"**).get(username);  
 *//修改状态* seckillOrder.setStatus(**"1"**);  
 *//流水号* seckillOrder.setTransactionId(transaction\_id);  
 *//支付时间* seckillOrder.setPayTime(**new** Date());  
 *//同步到MySQL中* **seckillOrderMapper**.insertSelective(seckillOrder);  
  
 *//删除订单Redis缓存* **redisTemplate**.boundHashOps(**"SeckillOrder"**).delete(username);  
 *//删除用户下单记录数据* **redisTemplate**.boundHashOps(**"UserQueueCount"**).delete(username);  
 *//删除抢购状态信息* **redisTemplate**.boundHashOps(**"UserQueueStatus"**).delete(username);  
}

## 删除订单回滚库存

如果用户支付失败，我们需要删除用户订单数据，并回滚库存。

修改SeckillOrderService，创建一个关闭订单方法，代码如下：

*/\*\*\*  
 \* 关闭订单，回滚库存  
 \*/***void** closeOrder(String username);

修改SeckillOrderServiceImpl，创建一个关闭订单实现方法：

**步骤：**

//1.读取排队状态-redis.UserQueueStatus

//2.获取Redis中订单信息-SeckillOrder

//3.如果Redis中有订单信息，说明用户未支付

//3.1删除订单-SeckillOrder.delete

//3.2回滚库存

//3.2.1从Redis中获取该商品-SeckillGoods\_time.get(id)

//3.2.2如果Redis中没有，则从数据库中加载

//3.2.3数量+1 (递增数量+1，队列数量+1)

//3.3.数据同步到Redis中-SeckillGoods\_time,UserQueueCount,UserQueueStatus

**代码：**

@Autowired  
**private** SeckillGoodsMapper **seckillGoodsMapper**;  
*/\*\*\*  
 \* 关闭订单，回滚库存  
 \** ***@param username*** *\*/*@Override  
**public void** closeOrder(String username) {  
 *//1.读取排队状态-redis.UserQueueStatus* SeckillStatus seckillStatus = (SeckillStatus) **redisTemplate**.boundHashOps(**"UserQueueStatus"**).get(username);  
 *//2.获取Redis中订单信息-SeckillOrder* SeckillOrder seckillOrder = (SeckillOrder) **redisTemplate**.boundHashOps(**"SeckillOrder"**).get(username);  
 *//3.如果Redis中有订单信息，说明用户未支付* **if**(seckillStatus!=**null** && seckillOrder!=**null**){  
 *//3.1删除订单-SeckillOrder.delete* **redisTemplate**.boundHashOps(**"SeckillOrder"**).delete(username);  
 *//3.2回滚库存  
 //3.2.1从Redis中获取该商品-SeckillGoods\_time.get(id)* SeckillGoods seckillGoods = (SeckillGoods) **redisTemplate**.boundHashOps(**"SeckillGoods\_"**+seckillStatus.getTime()).get(seckillStatus.getGoodsId());  
 *//3.2.2如果Redis中没有，则从数据库中加载* **if**(seckillGoods==**null**){  
 seckillGoods = **seckillGoodsMapper**.selectByPrimaryKey(seckillStatus.getGoodsId());  
 }  
 *//3.2.3数量+1 (递增数量+1，队列数量+1)* Long surplusCount = **redisTemplate**.boundHashOps(**"SeckillGoodsCount"**).increment(seckillStatus.getGoodsId(), 1);  
 seckillGoods.setStockCount(surplusCount.intValue());  
 **redisTemplate**.boundListOps(**"SeckillGoodsCountList\_"** + seckillStatus.getGoodsId()).leftPush(seckillStatus.getGoodsId());  
 *//3.3.数据同步到Redis中-SeckillGoods\_time,UserQueueCount,UserQueueStatus* **redisTemplate**.boundHashOps(**"SeckillGoods\_"**+seckillStatus.getTime()).put(seckillStatus.getGoodsId(),seckillGoods);  
 *//清理排队标识* **redisTemplate**.boundHashOps(**"UserQueueCount"**).delete(seckillStatus.getUsername());  
 *//清理抢单标识* **redisTemplate**.boundHashOps(**"UserQueueStatus"**).delete(seckillStatus.getUsername());  
 }  
}

## 监听器调用修改与删除逻辑

@Autowired  
**private** SeckillOrderService **seckillOrderService**;  
  
@RabbitListener(queues = **"${mq.pay.queue.seckillorder}"**)  
**public void** payListener(String msg) {  
 *//将数据转成Map* Map<String, String> result = JSON.*parseObject*(msg, Map.**class**);  
 System.***out***.println(**"收到消息，参数为："** + result);  
 *//return\_code=SUCCESS* String return\_code = result.get(**"return\_code"**);  
  
 *//返回状态码 result\_code=SUCCESS/FAIL，修改订单状态* **if** (**"success"**.equalsIgnoreCase(return\_code)) {  
 *//业务结果* String result\_code = result.get(**"result\_code"**);  
 *//获取订单号* String out\_trade\_no = result.get(**"out\_trade\_no"**);  
 *//交易流水号* String transaction\_id = result.get(**"transaction\_id"**);  
  
 *//附加参数* Map<String, String> attachMap = JSON.*parseObject*(result.get(**"attach"**), Map.**class**);  
 *//用户名* String username = attachMap.get(**"username"**);  
  
 *//业务结果-SUCCESS/FAIL，为success时，支付成功* **if** (**"success"**.equalsIgnoreCase(result\_code)) {  
 *//修改订单状态 out\_trade\_no* **seckillOrderService**.updatePayStatus(out\_trade\_no,transaction\_id,username);  
 } **else** {  
 *//订单删除* **seckillOrderService**.closeOrder(username);  
 }  
 }  
}

## 整体测试流程

1. **先查看当前时间段的商品列表**

<http://localhost:18091/seckillGoods/list?time=2019091914>

1. **录入当前时间段与商品id开始抢购**

<http://localhost:18091/seckillOrder/add?time=2019091914&id=1131815455707762688>

1. **查询当前用户的抢购订单**

<http://localhost:18091/seckillOrder/query>

1. **调起微信统一下单，生成二维码**

<http://localhost:18090/weixin/pay/create/native?out_trade_no=1174554207802621952&total_fee=1&exchange=exchange.order&routingKey=queue.seckillorder>

1. **展示二维码并支付**



1. **支付成功后，可以在数据库查看到新订单信息，库存也已经扣减**



1. **可以通过查询当前时间段，再次查看库存扣减情况**

<http://localhost:18091/seckillGoods/list?time=2019091914>

# RabbitMQ延时消息队列

## 延时队列介绍

延时队列即放置在该队列里面的消息是不需要立即消费的，而是等待一段时间之后取出消费。

那么，为什么需要延迟消费呢？我们来看以下的场景

网上商城下订单后30分钟后没有完成支付，取消订单(如：淘宝、去哪儿网)

系统创建了预约之后，需要在预约时间到达前一小时提醒被预约的双方参会

系统中的业务失败之后，需要重试

这些场景都非常常见，我们可以思考，比如第二个需求，系统创建了预约之后，需要在预约时间到达前一小时提醒被预约的双方参会。那么一天之中肯定是会有很多个预约的，时间也是不一定的，假设现在有1点 2点 3点 三个预约，如何让系统知道在当前时间等于0点 1点 2点给用户发送信息呢，是不是需要一个轮询，一直去查看所有的预约，比对当前的系统时间和预约提前一小时的时间是否相等呢？这样做非常浪费资源而且轮询的时间间隔不好控制。如果我们使用延时消息队列呢，我们在创建时把需要通知的预约放入消息中间件中，并且设置该消息的过期时间，等过期时间到达时再取出消费即可。

Rabbitmq实现延时队列一般而言有两种形式：

第一种方式：利用两个特性： Time To Live(TTL)、Dead Letter Exchanges（DLX）[A队列过期转发给B队列]

第二种方式：利用rabbitmq中的插件x-delay-message

## TTL DLX实现延时队列

### TTL DLX介绍

**TTL**

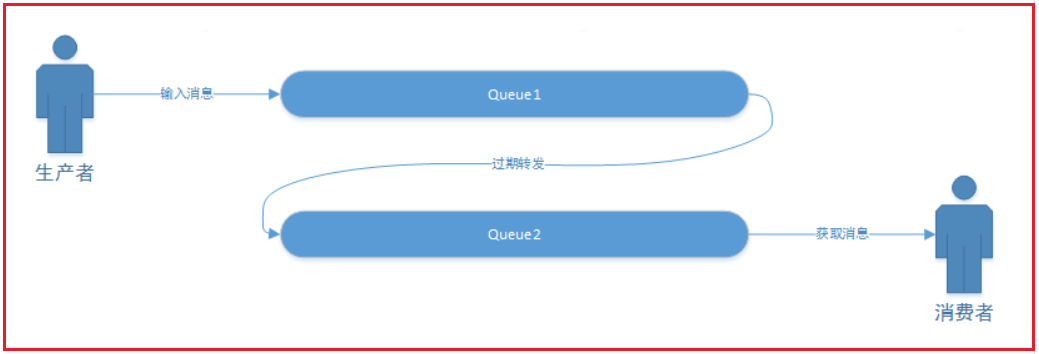
RabbitMQ可以针对队列设置x-expires(则队列中所有的消息都有相同的过期时间)或者针对Message设置x-message-ttl(对消息进行单独设置，每条消息TTL可以不同)，来控制消息的生存时间，如果超时(两者同时设置以最先到期的时间为准)，则消息变为dead letter(死信)

**Dead Letter Exchanges（DLX）**

RabbitMQ的Queue可以配置x-dead-letter-exchange和x-dead-letter-routing-key（可选）两个参数，如果队列内出现了dead letter，则按照这两个参数重新路由转发到指定的队列。

x-dead-letter-exchange：出现dead letter之后将dead letter重新发送到指定exchange

x-dead-letter-routing-key：出现dead letter之后将dead letter重新按照指定的routing-key发送



### DLX延时队列实现

#### 创建工程

创建springboot\_rabbitmq\_delay工程，并引入相关依赖

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**groupId**>com.itheima</**groupId**>  
 <**artifactId**>springboot\_rabbitmq\_delay</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
  
 <**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.1.7.RELEASE</**version**>  
 </**parent**>  
  
 <**dependencies**>  
 *<!--starter-web-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 </**dependency**>  
 *<!--加入ampq-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-amqp</**artifactId**>  
 </**dependency**>  
 *<!--测试-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

application.yml配置

**spring**:  
 **application**:  
 **name**: springboot-rabbitmq-delay  
 **rabbitmq**:  
 **host**: 192.168.211.132  
 **port**: 5672  
 **password**: guest  
 **username**: guest

创建启动引导类

@SpringBootApplication  
@EnableRabbit  
**public class** SpringRabbitMQApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(SpringRabbitMQApplication.**class**,args);  
 }  
}

#### 队列创建

创建2个队列，用于接收消息的叫延时队列queue.message.delay，用于转发消息的队列叫queue.message，同时创建一个交换机，代码如下：

@Configuration  
**public class** QueueConfig {  
  
 */\*\* 短信接收队列 \*/* **public static final** String ***QUEUE\_MESSAGE*** = **"queue.message"**;  
  
 */\*\* 交换机 \*/* **public static final** String ***DLX\_EXCHANGE*** = **"dlx.exchange"**;  
  
 */\*\* 短信发送队列 延迟缓冲（收消息） \*/* **public static final** String ***QUEUE\_MESSAGE\_DELAY*** = **"queue.message.delay"**;  
  
 *//------------------------------接收消息队列---------------------------------  
 /\*\*  
 \* 短信接收队列  
 \** ***@return*** *\*/* @Bean  
 **public** Queue messageQueue() {  
 **return new** Queue(***QUEUE\_MESSAGE***, **true**);  
 }  
  
 *//--------------------------发送消息队列----------------------------  
 /\*\*  
 \* 短信发送队列  
 \** ***@return*** *\*/* @Bean  
 **public** Queue delayMessageQueue() {  
 *//durable(死信队列)* **return** QueueBuilder.*durable*(***QUEUE\_MESSAGE\_DELAY***)  
 *// 消息超时进入死信队列，绑定死信队列交换机* .withArgument(**"x-dead-letter-exchange"**, ***DLX\_EXCHANGE***)  
 *// 绑定指定转发的routing-key* .withArgument(**"x-dead-letter-routing-key"**, ***QUEUE\_MESSAGE***)  
 .build();  
 }  
  
 *//---------------绑定交换机与队列----------------  
 /\*\*\*  
 \* 创建交换机  
 \** ***@return*** *\*/* @Bean  
 **public** DirectExchange directExchange(){  
 **return new** DirectExchange(***DLX\_EXCHANGE***);  
 }  
 */\*\*\*  
 \* 交换机与队列绑定  
 \** ***@param messageQueue*** *\** ***@param directExchange*** *\** ***@return*** *\*/* @Bean  
 **public** Binding basicBinding(Queue messageQueue, DirectExchange directExchange) {  
 **return** BindingBuilder.*bind*(messageQueue)  
 .to(directExchange)  
 .with(***QUEUE\_MESSAGE***);  
 }  
}

#### 消息监听

创建MessageListener用于监听消息，代码如下：

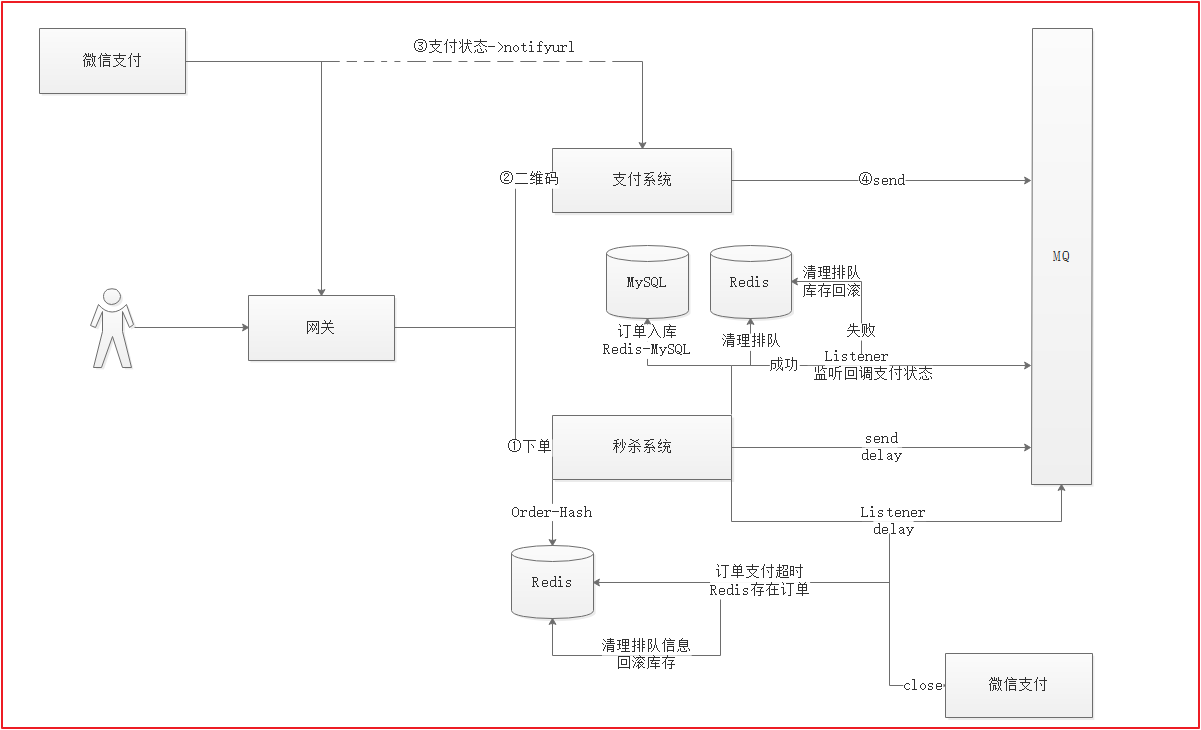
@Component  
**public class** MessageListener {  
  
 */\*\*\*  
 \* 监听消息  
 \** ***@param msg*** *\*/* @RabbitListener(queues = QueueConfig.***QUEUE\_MESSAGE***)  
 **public void** msg(@Payload Object msg) {  
 SimpleDateFormat dateFormat = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  
 System.***out***.println(**"当前时间:"** + dateFormat.format(**new** Date()));  
 System.***out***.println(**"收到信息:"** + msg);  
 }  
}

#### 测试

@SpringBootTest  
@RunWith(SpringRunner.**class**)  
**public class** RabbitMQTest {  
  
 @Autowired  
 **private** RabbitTemplate **rabbitTemplate**;  
  
 */\*\*\*  
 \* 发送消息  
 \*/* @Test  
 **public void** sendMessage() **throws** InterruptedException, IOException {  
 SimpleDateFormat dateFormat = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  
 System.***out***.println(**"发送当前时间:"** + dateFormat.format(**new** Date()));  
 Map<String, String> message = **new** HashMap<>();  
 message.put(**"name"**, **"szitheima"**);  
 **rabbitTemplate**.convertAndSend(QueueConfig.***QUEUE\_MESSAGE\_DELAY***, message, **new** MessagePostProcessor() {  
 @Override  
 **public** Message postProcessMessage(Message message) **throws** AmqpException {  
 *//延时10发送消息* message.getMessageProperties().setExpiration(**"10000"**);  
 **return** message;  
 }  
 });  
 System.***in***.read();  
 }  
}

# 库存回滚(作业)

## 秒杀流程回顾



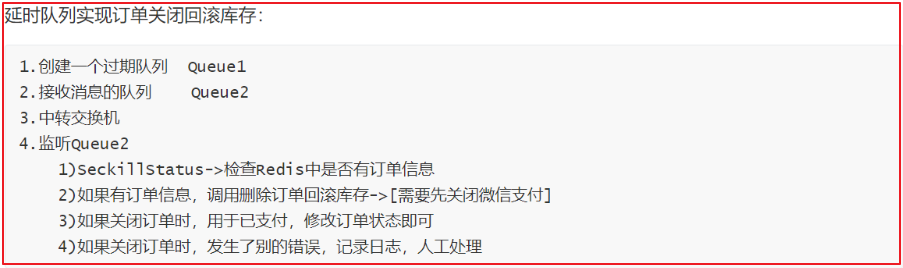
如上图，步骤分析如下：

1.用户抢单，经过秒杀系统实现抢单，下单后会将向MQ发送一个延时队列消息，包含抢单信息，延时半小时后才能监听到

2.秒杀系统同时启用延时消息监听，一旦监听到订单抢单信息，判断Redis缓存中是否存在订单信息，如果存在，则回滚

3.秒杀系统还启动支付回调信息监听，如果支付完成，则将订单吃句话到MySQL，如果没完成，清理排队信息回滚库存

4.每次秒杀下单后调用支付系统，创建二维码，如果用户支付成功了，微信系统会将支付信息发送给支付系统指定的回调地址，支付系统收到信息后，将信息发送给MQ，第3个步骤就可以监听到消息了。



## 关闭支付

用户如果半个小时没有支付，我们会关闭支付订单，但在关闭之前，需要先关闭微信支付，防止中途用户支付。

修改支付微服务的WeixinPayService，添加关闭支付方法，代码如下：

/\*\*\*

\* 关闭支付

\* @param orderId

\* @return

\*/

Map<String,String> closePay(Long orderId) **throws** Exception;

修改WeixinPayServiceImpl，实现关闭微信支付方法，代码如下：

/\*\*\*

\* 关闭微信支付

\* @param orderId

\* @return

\* @throws Exception

\*/

@Override

**public** Map<String, String> closePay(Long orderId) **throws** Exception {

//参数设置

Map<String,String> paramMap = **new** HashMap<String,String>();

paramMap.put("appid",appid); //应用ID

paramMap.put("mch\_id",partner); //商户编号

paramMap.put("nonce\_str",WXPayUtil.generateNonceStr());//随机字符

paramMap.put("out\_trade\_no",String.valueOf(orderId)); //商家的唯一编号

//将Map数据转成XML字符

String xmlParam = WXPayUtil.generateSignedXml(paramMap,partnerkey);

//确定url

String url = "https://api.mch.weixin.qq.com/pay/closeorder";

//发送请求

HttpClient httpClient = **new** HttpClient(url);

//https

httpClient.setHttps(**true**);

//提交参数

httpClient.setXmlParam(xmlParam);

//提交

httpClient.post();

//获取返回数据

String content = httpClient.getContent();

//将返回数据解析成Map

**return** WXPayUtil.xmlToMap(content);

}

## 关闭订单回滚库存

### 配置延时队列

在application.yml文件中引入队列信息配置，如下：

#位置支付交换机和队列

mq:

pay:

exchange:

order: exchange.order

queue:

order: queue.order

seckillorder: queue.seckillorder

seckillordertimer: queue.seckillordertimer

seckillordertimerdelay: queue.seckillordertimerdelay

routing:

orderkey: queue.order

seckillorderkey: queue.seckillorder

配置队列与交换机,在SeckillApplication中添加如下方法

/\*\*

\* 到期数据队列

\* @return

\*/

@Bean

**public** Queue **seckillOrderTimerQueue**() {

**return** **new** **Queue**(env.getProperty("mq.pay.queue.seckillordertimer"), **true**);

}

/\*\*

\* 超时数据队列

\* @return

\*/

@Bean

**public** Queue **delaySeckillOrderTimerQueue**() {

**return** QueueBuilder.durable(env.getProperty("mq.pay.queue.seckillordertimerdelay"))

.withArgument("x-dead-letter-exchange", env.getProperty("mq.pay.exchange.order")) // 消息超时进入死信队列，绑定死信队列交换机

.withArgument("x-dead-letter-routing-key", env.getProperty("mq.pay.queue.seckillordertimer")) // 绑定指定的routing-key

.build();

}

/\*\*\*

\* 交换机与队列绑定

\* @return

\*/

@Bean

**public** Binding **basicBinding**() {

**return** BindingBuilder.bind(seckillOrderTimerQueue())

.to(basicExchange())

.with(env.getProperty("mq.pay.queue.seckillordertimer"));

}

### 发送延时消息

修改MultiThreadingCreateOrder，添加如下方法：

/\*\*\*

\* 发送延时消息到RabbitMQ中

\* @param seckillStatus

\*/

**public** **void** **sendTimerMessage**(SeckillStatus seckillStatus){

rabbitTemplate.convertAndSend(env.getProperty("mq.pay.queue.seckillordertimerdelay"), (Object) JSON.toJSONString(seckillStatus), **new** MessagePostProcessor() {

@Override

**public** Message **postProcessMessage**(Message message) **throws** AmqpException {

message.getMessageProperties().setExpiration("10000");

**return** message;

}

});

}

在createOrder方法中调用上面方法，如下代码：

**//发**送延时消息到MQ中

sendTimerMessage(seckillStatus);

### 库存回滚

创建SeckillOrderDelayMessageListener实现监听消息，并回滚库存，代码如下：

@Component

@RabbitListener(queues = "${mq.pay.queue.seckillordertimer}")

**public** **class** **SeckillOrderDelayMessageListener** {

@Autowired

**private** RedisTemplate redisTemplate;

@Autowired

**private** SeckillOrderService seckillOrderService;

@Autowired

**private** WeixinPayFeign weixinPayFeign;

/\*\*\*

\* 读取消息

\* 判断Redis中是否存在对应的订单

\* 如果存在，则关闭支付，再关闭订单

\* @param message

\*/

@RabbitHandler

**public** **void** **consumeMessage**(@Payload String message){

//读取消息

SeckillStatus seckillStatus = JSON.parseObject(message,SeckillStatus.class);

//获取Redis中订单信息

String username = seckillStatus.getUsername();

SeckillOrder seckillOrder = (SeckillOrder) redisTemplate.boundHashOps("SeckillOrder").get(username);

//如果Redis中有订单信息，说明用户未支付

**if**(seckillOrder!=**null**){

System.out.println("准备回滚---"+seckillStatus);

//关闭支付

Result closeResult = weixinPayFeign.closePay(seckillStatus.getOrderId());

Map<String,String> closeMap = (Map<String, String>) closeResult.getData();

**if**(closeMap!=**null** && closeMap.get("return\_code").equalsIgnoreCase("success") &&

closeMap.get("result\_code").equalsIgnoreCase("success") ){

//关闭订单

seckillOrderService.closeOrder(username);

}

}

}

}