# SpringBoot消息重试机制

### 消息重试机制幂等性

#### 如何合适选择重试机制

情况1:  消费者获取到消息后，调用第三方接口，但接口暂时无法访问，是否需要重试?      需要重试

情况2:  消费者获取到消息后，抛出数据转换异常，是否需要重试?                                      不需要重试

总结：对于情况2，如果消费者代码抛出异常是需要发布新版本才能解决的问题，那么不需要重试，重试也无济于事。应该采用日志记录+定时任务job健康检查+人工进行补偿

#### 消费者如果保证消息幂等性，不被重复消费

产生原因:网络延迟传输中，会造成进行MQ重试中，在重试过程中，可能会造成重复消费。

解决办法:

使用全局MessageID判断消费方使用同一个，解决幂等性。

#### 基于全局消息id区分消息，解决幂等性

##### 生产者:

请求头设置消息id（messageId）

|  |
| --- |
| @Component  **public** **class** FanoutProducer {  @Autowired  **private** AmqpTemplate amqpTemplate;  **public** **void** send(String queueName) {  String msg = "my\_fanout\_msg:" + System.*currentTimeMillis*();  Message message = MessageBuilder.*withBody*(msg.getBytes()).setContentType(MessageProperties.***CONTENT\_TYPE\_JSON***)  .setContentEncoding("utf-8").setMessageId(UUID.*randomUUID*() + "").build();  System.***out***.println(msg + ":" + msg);  amqpTemplate.convertAndSend(queueName, message);  }  } |

##### 消费者:

###### 核心代码

|  |
| --- |
| @Component  **public** **class** FanoutEamilConsumer {  @RabbitListener(queues = "fanout\_email\_queue")  **public** **void** process(Message message) **throws** Exception {  System.***out***  .println(Thread.*currentThread*().getName() + ",邮件消费者获取生产者消息msg:" + **new** String(message.getBody(), "UTF-8")  + ",messageId:" + message.getMessageProperties().getMessageId());  // int i = 1 / 0;  }  } |

###### application配置

|  |
| --- |
| spring:  rabbitmq:  ####连接地址  host: 127.0.0.1  ####端口号  port: 5672  ####账号  username: guest  ####密码  password: guest  ### 地址  virtual-host: /admin\_host  listener:  simple:  retry:  ####开启消费者重试  enabled: **true**  ####最大重试次数  max-attempts: 5  ####重试间隔次数  initial-interval: 3000    server:  port: 8081 |

#### RabbitMQ消费者重试调用接口

|  |
| --- |
| //邮件队列  @Component  **public** **class** FanoutEamilConsumer {  @RabbitListener(queues = "fanout\_email\_queue")  **public** **void** process(String msg) **throws** Exception {  System.***out***.println("邮件消费者获取生产者消息msg:" + msg);  JSONObject jsonObject = JSONObject.*parseObject*(msg);  // 获取email参数  String email = jsonObject.getString("email");  // 请求地址  String emailUrl = "http://127.0.0.1:8083/sendEmail?email=" + email;  JSONObject result = HttpClientUtils.*httpGet*(emailUrl);  **if** (result == **null**) {  // 因为网络原因,造成无法访问,继续重试  **throw** **new** Exception("调用接口失败!");  }  System.***out***.println("执行结束....");  }  } |

|  |
| --- |
| @RabbitListener(queues = "fanout\_email\_queue")  **public** **void** process(Message message) **throws** Exception {  // 获取消息Id  String messageId = message.getMessageProperties().getMessageId();  String msg = **new** String(message.getBody(), "UTF-8");  System.***out***.println("邮件消费者获取生产者消息" + "messageId:" + messageId + ",消息内容:" + msg);  JSONObject jsonObject = JSONObject.*parseObject*(msg);  // 获取email参数  String email = jsonObject.getString("email");  // 请求地址  String emailUrl = "http://127.0.0.1:8083/sendEmail?email=" + email;  JSONObject result = HttpClientUtils.*httpGet*(emailUrl);  **if** (result == **null**) {  // 因为网络原因,造成无法访问,继续重试  **throw** **new** Exception("调用接口失败!");  }  System.***out***.println("执行结束....");  } |

# RabbitMQ签收模式

|  |
| --- |
| //邮件队列  @Component  **public** **class** FanoutEamilConsumer {  @RabbitListener(queues = "fanout\_email\_queue")  **public** **void** process(Message message, @Headers Map<String, Object> headers, Channel channel) **throws** Exception {  System.***out***  .println(Thread.*currentThread*().getName() + ",邮件消费者获取生产者消息msg:" + **new** String(message.getBody(), "UTF-8")  + ",messageId:" + message.getMessageProperties().getMessageId());  // 手动ack  Long deliveryTag = (Long) headers.get(AmqpHeaders.***DELIVERY\_TAG***);  // 手动签收  channel.basicAck(deliveryTag, **false**);  }  } |

开启手动应答

|  |
| --- |
| spring:  rabbitmq:  ####连接地址  host: 127.0.0.1  ####端口号  port: 5672  ####账号  username: guest  ####密码  password: guest  ### 地址  virtual-host: /admin\_host  listener:  simple:  retry:  ####开启消费者异常重试  enabled: **true**  ####最大重试次数  max-attempts: 5  ####重试间隔次数  initial-interval: 2000  ####开启手动ack  acknowledge-mode: manual |

# RabbitMQ死信队列

死信队列 听上去像 消息“死”了     其实也有点这个意思，死信队列  是 当消息在一个队列 因为下列原因：

消息被拒绝（basic.reject/ basic.nack）并且不再重新投递 requeue=false

消息超期 (rabbitmq Time-To-Live -> messageProperties.setExpiration())

队列超载

变成了 “死信” 后    被重新投递（publish）到另一个Exchange   该Exchange 就是DLX     然后该Exchange 根据绑定规则 转发到对应的 队列上  监听该队列  就可以重新消费     说白了 就是  没有被消费的消息  换个地方重新被消费

生产者 --> 消息 --> 交换机 --> 队列 --> 变成死信 --> DLX交换机 -->队列 --> 消费者

## 什么是死信呢？什么样的消息会变成死信呢？

消息被拒绝（basic.reject或basic.nack）并且requeue=false.

消息TTL过期

队列达到最大长度（队列满了，无法再添加数据到mq中）

## 应用场景分析

在定义业务队列的时候，可以考虑指定一个死信交换机，并绑定一个死信队列，当消息变成死信时，该消息就会被发送到该死信队列上，这样就方便我们查看消息失败的原因了

## 如何使用死信交换机呢？

定义业务（普通）队列的时候指定参数

x-dead-letter-exchange: 用来设置死信后发送的交换机

x-dead-letter-routing-key：用来设置死信的routingKey

## 死信队列环境搭建

### 死信队列配置

生产者配置

|  |
| --- |
| @Component  **public** **class** FanoutConfig {  /\*\*  \* 定义死信队列相关信息  \*/  **public** **final** **static** String ***deadQueueName*** = "dead\_queue";  **public** **final** **static** String ***deadRoutingKey*** = "dead\_routing\_key";  **public** **final** **static** String ***deadExchangeName*** = "dead\_exchange";  /\*\*  \* 死信队列 交换机标识符  \*/  **public** **static** **final** String ***DEAD\_LETTER\_QUEUE\_KEY*** = "x-dead-letter-exchange";  /\*\*  \* 死信队列交换机绑定键标识符  \*/  **public** **static** **final** String ***DEAD\_LETTER\_ROUTING\_KEY*** = "x-dead-letter-routing-key";  // 邮件队列  **private** String FANOUT\_EMAIL\_QUEUE = "fanout\_email\_queue";  // 短信队列  **private** String FANOUT\_SMS\_QUEUE = "fanout\_sms\_queue";  // fanout 交换机  **private** String EXCHANGE\_NAME = "fanoutExchange";  // 1.定义邮件队列  @Bean  **public** Queue fanOutEamilQueue() {  // 将普通队列绑定到死信队列交换机上  Map<String, Object> args = **new** HashMap<>(2);  args.put(***DEAD\_LETTER\_QUEUE\_KEY***, ***deadExchangeName***);  args.put(***DEAD\_LETTER\_ROUTING\_KEY***, ***deadRoutingKey***);  Queue queue = **new** Queue(FANOUT\_EMAIL\_QUEUE, **true**, **false**, **false**, args);  **return** queue;  }  // 2.定义短信队列  @Bean  **public** Queue fanOutSmsQueue() {  **return** **new** Queue(FANOUT\_SMS\_QUEUE);  }  // 2.定义交换机  @Bean  FanoutExchange fanoutExchange() {  **return** **new** FanoutExchange(EXCHANGE\_NAME);  }  // 3.队列与交换机绑定邮件队列  @Bean  Binding bindingExchangeEamil(Queue fanOutEamilQueue, FanoutExchange fanoutExchange) {  **return** BindingBuilder.*bind*(fanOutEamilQueue).to(fanoutExchange);  }  // 4.队列与交换机绑定短信队列  @Bean  Binding bindingExchangeSms(Queue fanOutSmsQueue, FanoutExchange fanoutExchange) {  **return** BindingBuilder.*bind*(fanOutSmsQueue).to(fanoutExchange);  }  /\*\*  \* 配置死信队列  \*  \* **@return**  \*/  @Bean  **public** Queue deadQueue() {  Queue queue = **new** Queue(***deadQueueName***, **true**);  **return** queue;  }  @Bean  **public** DirectExchange deadExchange() {  **return** **new** DirectExchange(***deadExchangeName***);  }  @Bean  **public** Binding bindingDeadExchange(Queue deadQueue, DirectExchange deadExchange) {  **return** BindingBuilder.*bind*(deadQueue).to(deadExchange).with(***deadRoutingKey***);  }  } |

### 消费者配置

|  |
| --- |
| @RabbitListener(queues = "fanout\_email\_queue")  **public** **void** process(Message message, @Headers Map<String, Object> headers, Channel channel) **throws** Exception {  String messageId = message.getMessageProperties().getMessageId();  String msg = **new** String(message.getBody(), "UTF-8");  System.***out***.println("邮件消费者获取生产者消息msg:" + msg + ",消息id:" + messageId);  JSONObject jsonObject = JSONObject.*parseObject*(msg);  Integer timestamp = jsonObject.getInteger("timestamp");  **try** {  **int** result = 1 / timestamp;  System.***out***.println("result:" + result);  // 通知mq服务器删除该消息  channel.basicAck(message.getMessageProperties().getDeliveryTag(), **false**);  } **catch** (Exception e) {  e.printStackTrace();  // // 丢弃该消息  channel.basicNack(message.getMessageProperties().getDeliveryTag(), **false**, **false**);  }  } |

|  |
| --- |
| @Component  **public** **class** DeadConsumer {  @RabbitListener(queues = "dead\_queue")  **public** **void** process(Message message, @Headers Map<String, Object> headers, Channel channel) **throws** Exception {  String messageId = message.getMessageProperties().getMessageId();  String msg = **new** String(message.getBody(), "UTF-8");  System.***out***.println("死信邮件消费者获取生产者消息msg:" + msg + ",消息id:" + messageId);  channel.basicAck(message.getMessageProperties().getDeliveryTag(), **false**);  }  } |

MQ解决分布式事务三个重要概念

1. 确保生产者消息一定要投递到MQ服务器中 Confirm机制
2. 确保消费者能够正确的消费消息，采用手动ACK（注意幂等）
3. 如何保证第一个事务一定要创建成功（在创建一个补单的队列，绑定同一个交换机，检查订单数据是否已经创建在数据库中 实现补偿机制）

生产者 一定确保消息投递到MQ服务器（使用）

# RabbitMQ解决分布式事务问题

RabbitMQ解决分布式事务原理： 采用最终一致性原理。需要保证以下三要素1、确认生产者一定要将数据投递到MQ服务器中（采用MQ消息确认机制）2、MQ消费者消息能够正确消费消息，采用手动ACK模式（注意重试幂等性问题）3、如何保证第一个事务先执行，采用补偿机制，在创建一个补单消费者进行监听，如果订单没有创建成功，进行补单。

## 订单项目

### 生产者

|  |
| --- |
| @Service  **public** **class** OrderService **extends** BaseApiService **implements** RabbitTemplate.ConfirmCallback {  @Autowired  **private** OrderMapper orderMapper;  @Autowired  **private** RabbitTemplate rabbitTemplate;  **public** ResponseBase addOrderAndDispatch() {  OrderEntity orderEntity = **new** OrderEntity();  orderEntity.setName("蚂蚁课堂永久会员充值");  orderEntity.setOrderCreatetime(**new** Date());  // 价格是300元  orderEntity.setOrderMoney(300d);  // 状态为 未支付  orderEntity.setOrderState(0);  Long commodityId = 30l;  // 商品id  orderEntity.setCommodityId(commodityId);  String orderId = UUID.*randomUUID*().toString();  orderEntity.setOrderId(orderId);  // ##################################################  // 1.先下单，创建订单 (往订单数据库中插入一条数据)  **int** orderResult = orderMapper.addOrder(orderEntity);  System.***out***.println("orderResult:" + orderResult);  **if** (orderResult <= 0) {  **return** setResultError("下单失败!");  }  // 2.使用消息中间件将参数存在派单队列中  send(orderId);  **return** setResultSuccess();  }  **private** **void** send(String orderId) {  JSONObject jsonObect = **new** JSONObject();  jsonObect.put("orderId", orderId);  String msg = jsonObect.toJSONString();  System.***out***.println("msg:" + msg);  // 封装消息  Message message = MessageBuilder.*withBody*(msg.getBytes()).setContentType(MessageProperties.***CONTENT\_TYPE\_JSON***)  .setContentEncoding("utf-8").setMessageId(orderId).build();  // 构建回调返回的数据  CorrelationData correlationData = **new** CorrelationData(orderId);  // 发送消息  **this**.rabbitTemplate.setMandatory(**true**);  **this**.rabbitTemplate.setConfirmCallback(**this**);  rabbitTemplate.convertAndSend("order\_exchange\_name", "orderRoutingKey", message, correlationData);  }  // 生产消息确认机制  @Override  **public** **void** confirm(CorrelationData correlationData, **boolean** ack, String cause) {  String orderId = correlationData.getId();  System.***out***.println("消息id:" + correlationData.getId());  **if** (ack) {  System.***out***.println("消息发送确认成功");  } **else** {  send(orderId);  System.***out***.println("消息发送确认失败:" + cause);  }  }  } |

### 补单消费者

|  |
| --- |
| @Component  **public** **class** CreateOrderConsumer {  @Autowired  **private** OrderMapper orderMapper;  @RabbitListener(queues = "order\_create\_queue")  **public** **void** process(Message message, @Headers Map<String, Object> headers, Channel channel) **throws** Exception {  String messageId = message.getMessageProperties().getMessageId();  String msg = **new** String(message.getBody(), "UTF-8");  System.***out***.println("补单消费者" + msg + ",消息id:" + messageId);  JSONObject jsonObject = JSONObject.*parseObject*(msg);  String orderId = jsonObject.getString("orderId");  // 判断订单是否存在，如果不存在 实现自动补单机制  OrderEntity orderEntityResult = orderMapper.findOrderId(orderId);  **if** (orderEntityResult != **null**) {  System.***out***.println("订单已经存在 无需补单 orderId:" + orderId);  **return**;  }  // 订单不存在 ，则需要进行补单  OrderEntity orderEntity = **new** OrderEntity();  orderEntity.setName("蚂蚁课堂永久会员充值");  orderEntity.setOrderCreatetime(**new** Date());  // 价格是300元  orderEntity.setOrderMoney(300d);  // 状态为 未支付  orderEntity.setOrderState(0);  Long commodityId = 30l;  // 商品id  orderEntity.setCommodityId(commodityId);  orderEntity.setOrderId(orderId);  // ##################################################  // 1.先下单，创建订单 (往订单数据库中插入一条数据)  **try** {  **int** orderResult = orderMapper.addOrder(orderEntity);  System.***out***.println("orderResult:" + orderResult);  **if** (orderResult >= 0) {  // 手动签收消息,通知mq服务器端删除该消息  channel.basicAck(message.getMessageProperties().getDeliveryTag(), **false**);  }  } **catch** (Exception e) {  // 丢弃该消息  channel.basicNack(message.getMessageProperties().getDeliveryTag(), **false**, **false**);  }  }  } |

### RabbitmqConfig

|  |
| --- |
| @Component  **public** **class** RabbitmqConfig {  // 下单并且派单存队列  **public** **static** **final** String ***ORDER\_DIC\_QUEUE*** = "order\_dic\_queue";  // 补单队列，判断订单是否已经被创建  **public** **static** **final** String ***ORDER\_CREATE\_QUEUE*** = "order\_create\_queue";  // 下单并且派单交换机  **private** **static** **final** String ***ORDER\_EXCHANGE\_NAME*** = "order\_exchange\_name";  // 1.定义订单队列  @Bean  **public** Queue directOrderDicQueue() {  **return** **new** Queue(***ORDER\_DIC\_QUEUE***);  }  // 2.定义补订单队列  @Bean  **public** Queue directCreateOrderQueue() {  **return** **new** Queue(***ORDER\_CREATE\_QUEUE***);  }  // 2.定义交换机  @Bean  DirectExchange directOrderExchange() {  **return** **new** DirectExchange(***ORDER\_EXCHANGE\_NAME***);  }  // 3.订单队列与交换机绑定  @Bean  Binding bindingExchangeOrderDicQueue() {  **return** BindingBuilder.*bind*(directOrderDicQueue()).to(directOrderExchange()).with("orderRoutingKey");  }  // 3.补单队列与交换机绑定  @Bean  Binding bindingExchangeCreateOrder() {  **return** BindingBuilder.*bind*(directCreateOrderQueue()).to(directOrderExchange()).with("orderRoutingKey");  }  } |

## 派单服务

### 消费者

|  |
| --- |
| @Component  **public** **class** DispatchConsumer {  @Autowired  **private** DispatchMapper dispatchMapper;  @RabbitListener(queues = "order\_dic\_queue")  **public** **void** process(Message message, @Headers Map<String, Object> headers, Channel channel) **throws** Exception {  String messageId = message.getMessageProperties().getMessageId();  String msg = **new** String(message.getBody(), "UTF-8");  System.***out***.println("派单服务平台" + msg + ",消息id:" + messageId);  JSONObject jsonObject = JSONObject.*parseObject*(msg);  String orderId = jsonObject.getString("orderId");  **if** (StringUtils.*isEmpty*(orderId)) {  // 日志记录  **return**;  }  DispatchEntity dispatchEntity = **new** DispatchEntity();  // 订单id  dispatchEntity.setOrderId(orderId);  // 外卖员id  dispatchEntity.setTakeoutUserId(12l);  // 外卖路线  dispatchEntity.setDispatchRoute("40,40");  **try** {  **int** insertDistribute = dispatchMapper.insertDistribute(dispatchEntity);  **if** (insertDistribute > 0) {  // 手动签收消息,通知mq服务器端删除该消息  channel.basicAck(message.getMessageProperties().getDeliveryTag(), **false**);  }  } **catch** (Exception e) {  e.printStackTrace();  // // 丢弃该消息  channel.basicNack(message.getMessageProperties().getDeliveryTag(), **false**, **false**);  }  }  } |