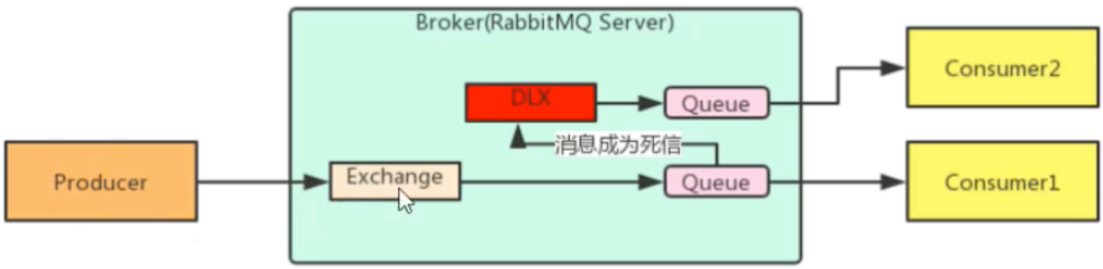
死信队列，英文缩写：DLX 。Dead Letter Exchange（死信交换机），当消息成为Dead message后，可以被重新发送到另一个交换机，这个交换机就是DLX。

死信队列

1.5 死信队列

死信队列，英文缩写：DLX 。Dead Letter Exchange（死信交换机），当消息成为Dead message后，可以被重新发送到另一个交换机，这个交换机就是DLX。

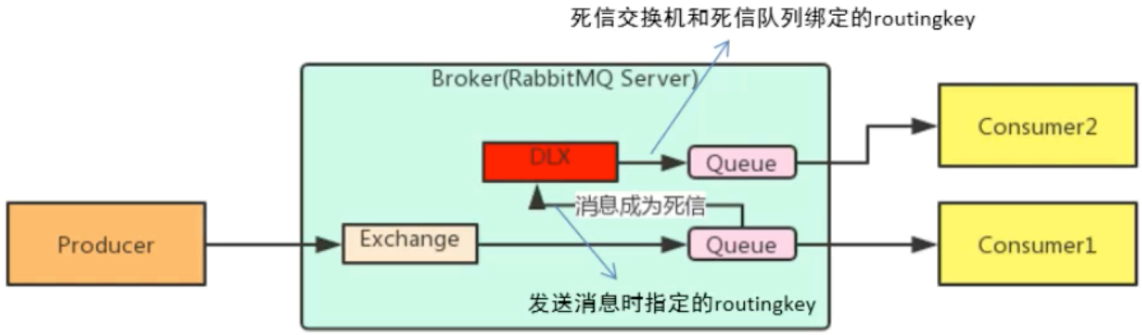


消息成为死信的三种情况：

1. 队列消息长度到达限制；
2. 消费者拒接消费消息，basicNack/basicReject,并且不把消息重新放入原目标队列,requeue=false；
3. 原队列存在消息过期设置，消息到达超时时间未被消费；

队列绑定死信交换机：

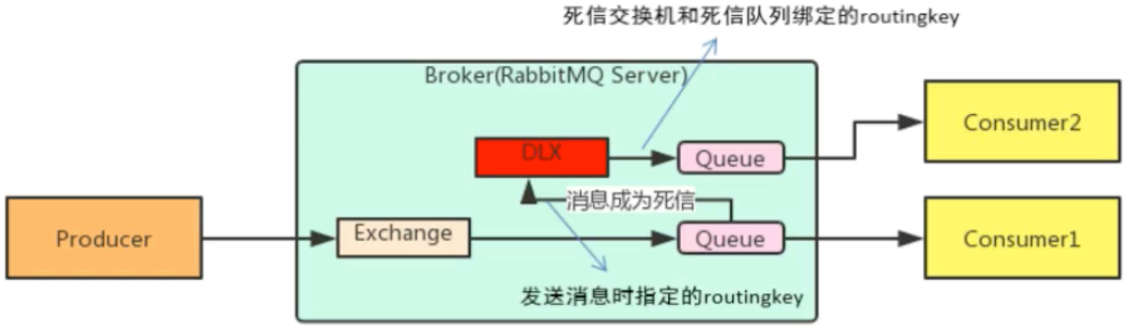
给队列设置参数： x-dead-letter-exchange 和 x-dead-letter-routing-key



1.5 死信队列

队列绑定死信交换机：

给队列设置参数： x-dead-letter-exchange 和 x-dead-letter-routing-key



1.5 死信队列小结

1. 死信交换机和死信队列和普通的没有区别

2. 当消息成为死信后，如果该队列绑定了死信交换机，则消息会被死信交换机重新路由到死信队列

3. 消息成为死信的三种情况：
    1. 队列消息长度到达限制；
    2. 消费者拒接消费消息，并且不重回队列；
    3. 原队列存在消息过期设置，消息到达超时时间未被消费；

springboot版

参考 https://www.jianshu.com/p/54eb90353534

前言

目的主要是学习RabbitMQ的TTL+DLX实现延迟队列，大概会简单介绍学习为主：毕竟还是要来演示Springboot整合RabbitMQ注解的方式来使用。

一. TTL

> TTL，Time to Live的简称，即过期时间。RabbitMQ 可以对消息和队列设置TTL。
> 消息的 TTL 指的是消息的存活时间，RabbitMQ 支持消息、队列两种方式设置 TTL，分别如下：
> **消息设置 TTL**：对消息的设置是在发送时进行 TTL 设置，通过 x-message-ttl 或expiration 字段设置，单位为毫秒，代表消息的过期时间，每条消息的 TTL 可不同。
> **队列设置 TTL**：对队列的设置是在消息入队列时计算，通过 x-expires 设置，队列中的所有消息都有相同的过期时间，当超过了队列的超时设置，消息会自动的清除。
> 注意：如果以上两种方式都做了设置，消息的 TTL 则以两者之中最小的那个为准。

二.死信队列

> DLX，全称为Dead-Letter-Exchange，可以称之为死信交换器，也有人称之为死信邮箱。当消息在一个队列中变成死信(dead message)之后，它能被重新被发送到另一个交换器中，这个交换器就是DLX，绑定DLX的队列就称之为死信队列。
> 消息变成死信一般是由于以下几种情况:
> 1). 消息被拒绝(Basic. Reject/Basic.Nack)，并且设置requeue参数为false;
> 2). 消息过期;
> 3). 队列达到最大长度。

2.1 DLX 交换机

DLX也是一个正常的交换器，和一般的交换器没有区别，它能在任何的队列上被指定，实际上就是设置某个队列的属性。当这个队列中存在死信时，RabbitMQ就会自动地将这个消息重新发布到设置的DLX上去，进而被路由到另一个队列，即死信队列。可以监听这个队列中的消息以进行相应的处理

####### 2.2 DLX声明

声明步骤如下：

第一步：声明正常的交换机，队列，路由键

第二步：在正常的队列里设置过期时间，绑定死信队列交换机名称以及设置发送给死信队列交换机路由键

第三步：声明死信交换机，队列，路由键

上面的做法是为了设置一个过期的正常队列路由到死信队列上面，让任意的队列去监听消费它。

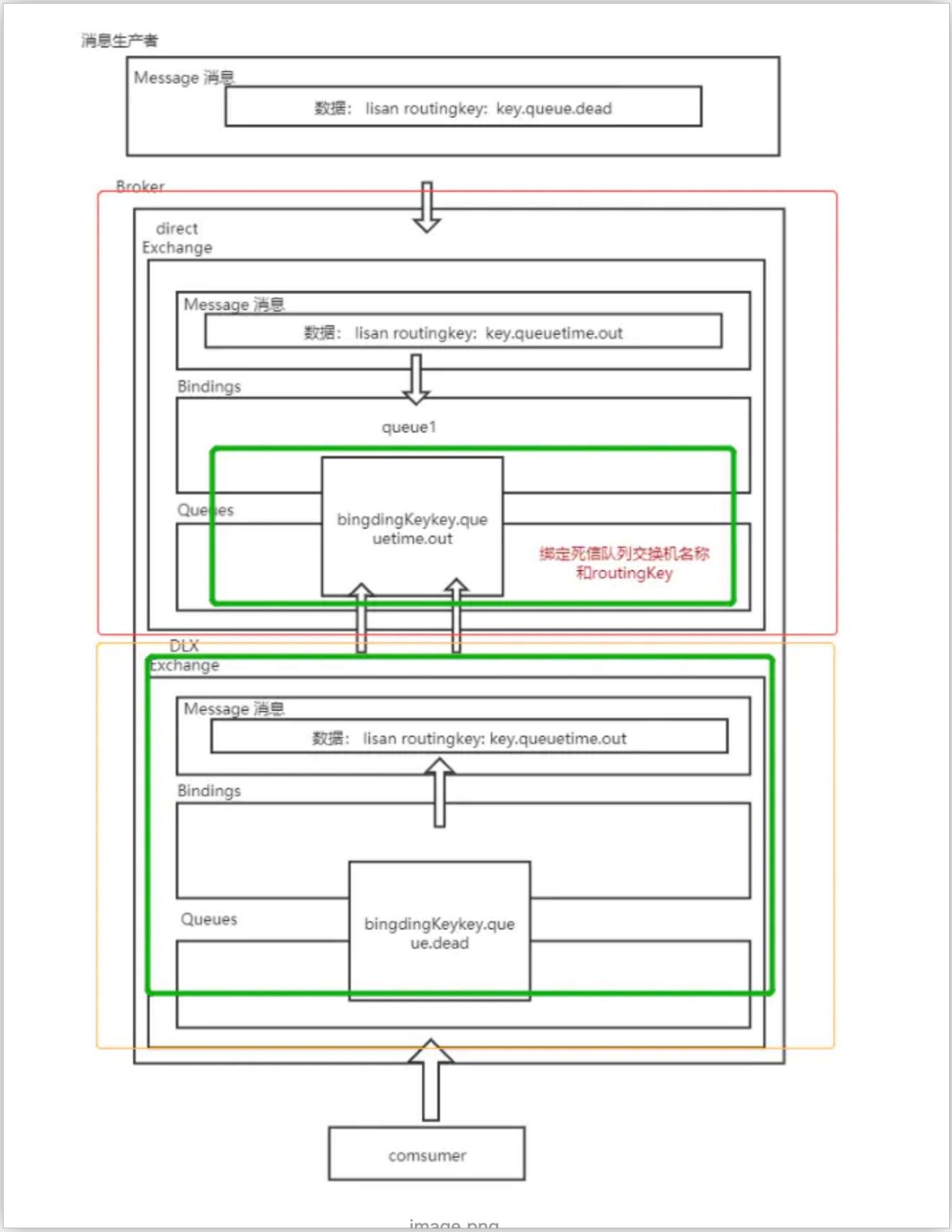
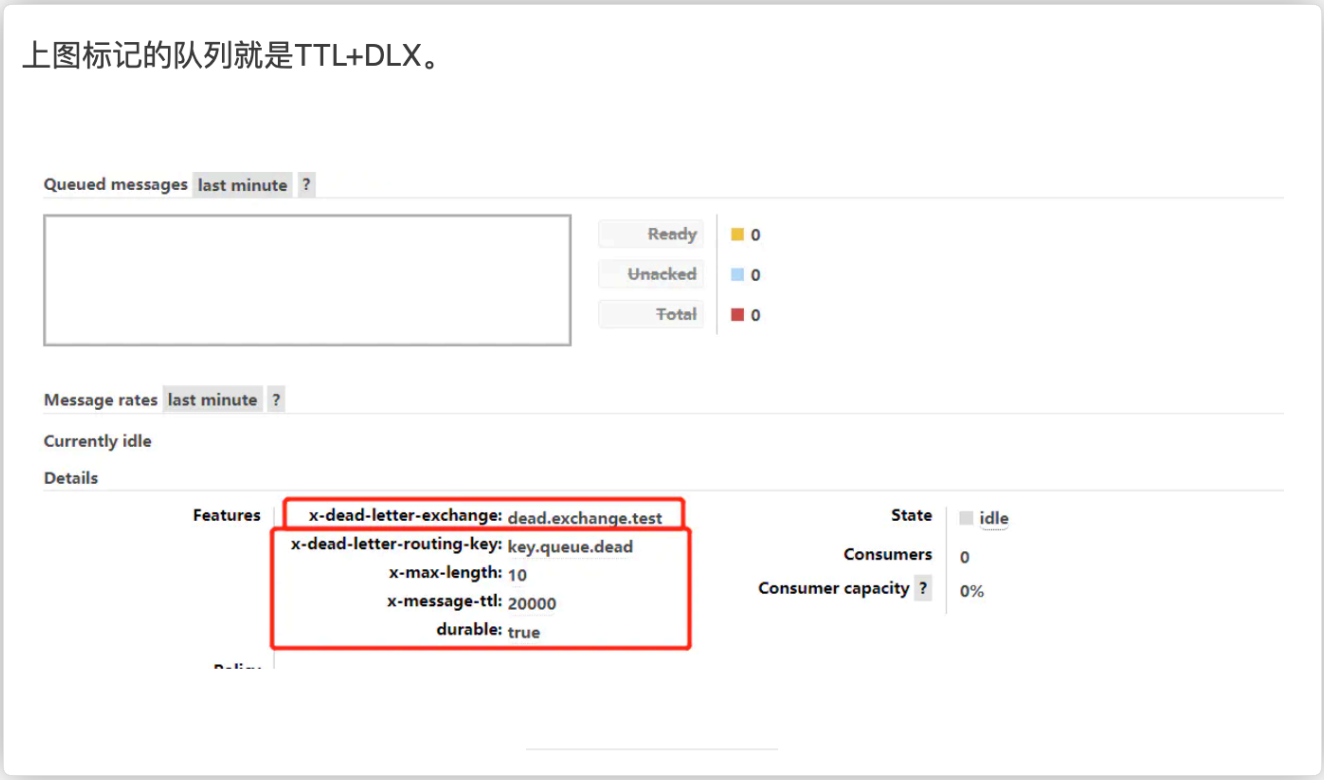


image.png

```
/* 设置队列过期时间*/
@Bean
public Exchange queueTimeOutExchange(){
```

```
        return new DirectExchange("queueTimeOut.exchange.test",true,false);
    }

    // 声明过期的队列并定义队列名称
    @Bean
    public Queue queueTimeOutQueue(){
        // 消息过期 特殊的args
        Map<String,Object> args  = new HashMap<>(16);
        args.put("x-message-ttl",20000);
        // 绑定死信交换机名称
        args.put("x-dead-letter-exchange", "dead.exchange.test");
        // 置发送给死信交换机的routingKey
        args.put("x-dead-letter-routing-key", "key.queue.dead");
        // 设置队列可以存储的最大消息数量
        args.put("x-max-length", 10);
        return new Queue("queueTimeOut.queue.test"
                ,true,false,false,args);
    }

    @Bean
    public Binding queueTimeOutBinding(){
        return new Binding("queueTimeOut.queue.test",
                Binding.DestinationType.QUEUE,
                "queueTimeOut.exchange.test",
                "key.queuetime.out",null);
    }

    /* 死信队列*/
    @Bean
    public Exchange deadExchange(){
        return new DirectExchange("dead.exchange.test",true,false);
    }

    // 声明过期的队列并定义队列名称
    @Bean
    public Queue deadQueue(){

        return new Queue("dead.queue.test"
                ,true,false,false);
    }

    @Bean
    public Binding deadBinding(){
        return new Binding("dead.queue.test",
                Binding.DestinationType.QUEUE,
                "dead.exchange.test",
                "key.queue.dead",null);
    }
```

上图标记的队列就是TTL+DLX。



上面标记之前为这个队列设置的特性

三，实现简单实例

3.1 生产端

service

```
    // 设置 死信队列
    public void deadMessag() throws JsonProcessingException;
```

imp

```
    /**
     * 測試死信隊列
     */
    @Override
    public void deadMessag() throws JsonProcessingException {
        /* 使用MessageProperties传递的对象转换成message*/
        MessageProperties messageProperties = new MessageProperties();
        OrderMessageDTO orderMessageDTO = new OrderMessageDTO();
        orderMessageDTO.setProductId(100);
        orderMessageDTO.setPrice(new BigDecimal("20"));
        orderMessageDTO.setOrderId(1);
        String messageToSend = objectMapper.writeValueAsString(orderMessageDTO);
```

```
        Message message = new Message(messageToSend.getBytes(),messageProperties);
        // 发送端确认是否确认消费
        CorrelationData correlationData = new CorrelationData();
        // 唯一ID
        correlationData.setId(orderMessageDTO.getOrderId().toString());

 rabbitTemplate.convertAndSend("dead.exchange.test","key.queue.dead",message,correlationData);
    }
```

controller

```
@RestController
@Slf4j
@RequestMapping("/api")
public class SendController {
    @GetMapping("/deadQueue")
    public void deadQueue() throws JsonProcessingException {
        directService.deadMessag();
    }
}
```

3.2 消费端

service

```
    // 监听死信队列，  死信队列可以在任何的队列上被指定，实际上就是设置某个队列的属性
    public void deadQueueListenter(Message message, Channel channel) throws IOException;
```

impl

```
   /**
     * 监听死信队列
     */
    @RabbitListener(
            containerFactory = "rabbitListenerContainerFactory",
            bindings = {
                    @QueueBinding(
                            value = @Queue(name = "dead.queue.test"),
                            exchange = @Exchange(name = "dead.exchange.test",
                                    type = ExchangeTypes.DIRECT),
                            key = "key.queue.dead"
                    )
            }
    )
    @Override
    public void deadQueueListenter(@Payload Message message, Channel channel) throws IOException
{
        log.info("========direct死信队列,业务场景超时==========");
        DefaultConsumer consumer = new DefaultConsumer(channel) {
```

```
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                String message = new String(body, "UTF-8");
                log.info("[x] Received '" + message + "'");
                // 手动 false ack
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        };
        //. 设置 Channel 消费者绑定队列
        channel.basicConsume("dead.queue.test",false,consumer);
    }
```

拒收消息放到死信队列

参考： https://www.cnblogs.com/lori/p/12074299.html

```
  rabbitmq:
    host: xxx
    port: xxx
    username: xxx
    password: xxx
    virtual-host: xxx
    ###开启消息确认机制 confirms
    publisher-confirms: true
    publisher-returns: true
    listener:
      simple:
        acknowledge-mode: manual #设置确认方式
        prefetch: 1 #每次处理1条消息
        retry.max-attempts: 3 # 最大重试次数
        retry.enabled: true #是否开启消费者重试（为false时关闭消费者重试，这时消费端代码异常会一直重复收到消息）
        retry.initial-interval: 2000 #重试间隔时间（单位毫秒）
        default-requeue-rejected: true #该配置项是决定由于监听器抛出异常而拒绝的消息是否被重新放回队列。默认值为true,需要手动basicNack时这些参数谅失效了
```

定义队列的相关配置

```
/**
    * 创建普通交换机.
    */
   @Bean
   public TopicExchange lindExchange() {
     //消息持久化
     return (TopicExchange) ExchangeBuilder.topicExchange(EXCHANGE).durable(true).build();
   }

   @Bean
   public TopicExchange deadExchange() {
     return (TopicExchange)
ExchangeBuilder.topicExchange(LIND_DL_EXCHANGE).durable(true).build();
   }

   /**
    * 基于消息事务的处理方式，当消费失败进行重试，有时间间隔，当达到超时时间，就发到死信队列，等待人工处理.
    * @return
    */
   @Bean
   public Queue testQueue() {
     //设置死信交换机
     return QueueBuilder.durable(QUEUE).withArgument("x-dead-letter-exchange", LIND_DL_EXCHANGE)
         //毫秒
         .withArgument("x-message-ttl", CONSUMER_EXPIRE)
         //设置死信routingKey
         .withArgument("x-dead-letter-routing-key", LIND_DEAD_QUEUE).build();
   }
   @Bean
   public Queue deadQueue() {
     return new Queue(LIND_DEAD_QUEUE);
   }
   @Bean
   public Binding bindBuildersRouteKey() {
     return BindingBuilder.bind(testQueue()).to(lindExchange()).with(ROUTER);
   }

   @Bean
   public Binding bindDeadBuildersRouteKey() {
     return BindingBuilder.bind(deadQueue()).to(deadExchange()).with(LIND_DEAD_QUEUE);
   }
```

消费者实现的代码

```
/**
    * 延时队列：不应该有RabbitListener订阅者，应该让它自己达到超时时间后自动转到死信里去消费
    * 消息异常处理：消费出现异常后，延时几秒，然后从新入队列消费，直到达到TTL超时时间，再转到死信，证明这个信息有
问题需要人工干预
    *
    * @param message
    */
```

```
  @RabbitListener(queues = MqConfig.QUEUE)
  public void testSubscribe(Message message, Channel channel) throws IOException,
InterruptedException {
    try {
      System.out.println(LocalDateTime.now() + ":Subscriber:" + new String(message.getBody(),
"UTF-8"));
      //当程序处理出现问题时，消息使用basicReject上报
      int a = 0;
      int b = 1 / a;
      channel.basicAck(message.getMessageProperties().getDeliveryTag(), true);
    } catch (Exception ex) {
      //出现异常手动放回队列
      Thread.sleep(2000);
      channel.basicNack(message.getMessageProperties().getDeliveryTag(), false, true);
    }

  }
 /**
   * 死信队列.
   *
   * @param message
   */
  @RabbitListener(queues = MqConfig.LIND_DEAD_QUEUE)
  public void dealSubscribe(Message message, Channel channel) throws IOException {
    System.out.println("Dead Subscriber:" + new String(message.getBody(), "UTF-8"));
    channel.basicAck(message.getMessageProperties().getDeliveryTag(), true);
  }
```

消费者这块,也可以直接声明队列和绑定交换机,直接在注解上添加 QueueBinding即可.

```
@RabbitListener(bindings = {@QueueBinding(value = @Queue(
      name = MqConfig.QUEUE,
      durable = "true",arguments = {@Argument(name = "x-dead-letter-exchange", value =
MqConfig.LIND_DL_EXCHANGE),
      @Argument(name = "x-message-ttl", value = MqConfig.CONSUMER_EXPIRE,type="java.lang.Long"),
      @Argument(name = "x-dead-letter-routing-key", value = MqConfig.LIND_DEAD_QUEUE)}),
      exchange = @Exchange(value = MqConfig.EXCHANGE, durable = "true",type="topic")

      )})
  public void testSubscribe(Message message, Channel channel) throws IOException,
InterruptedException {

  }
```

这边尝试让消费者执行出错,然后走到catch里使用basicNack方法把消息从新放里队列里,并让线程让休息2秒,以避免频繁操作,之后就是我们希望看到的代码

```
2019-12-20T17:21:31.190:Subscriber:send a message to mq
2019-12-20T17:21:33.200:Subscriber:send a message to mq
2019-12-20T17:21:35.206:Subscriber:send a message to mq
2019-12-20T17:21:37.213:Subscriber:send a message to mq
2019-12-20T17:21:39.221:Subscriber:send a message to mq
Dead Subscriber:send a message to mq
```