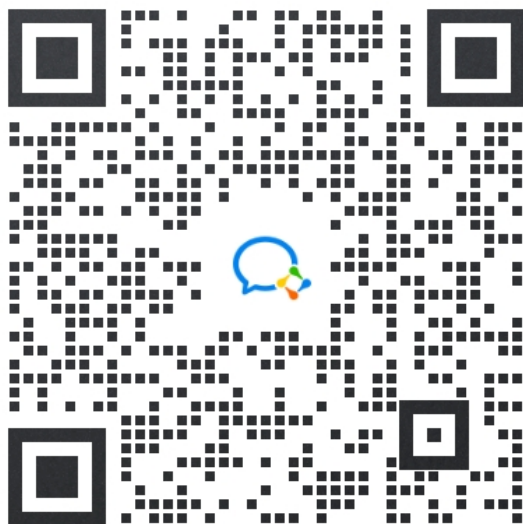


## 4. RabbitMQ高级特性

### 版权说明

本“比特就业课”课程（以下简称“本课程”）的所有内容，包括但不限于文字、图片、音频、视频、软件、程序、数据库、设计、布局、界面等，均由本课程的开发者或授权方拥有版权。我们鼓励个人学习者使用本课程进行学习和研究。在遵守相关法律法规的前提下，个人学习者可以下载、浏览、学习本课程的内容，并为了个人学习、研究或教学目的而使用其中的材料。但请注意，未经我们明确授权，个人学习者不得将本课程的内容用于任何商业目的，包括但不限于销售、转让、许可或以其他方式从中获利。此外，个人学习者也不得擅自修改、复制、传播、展示、表演或制作本课程内容的衍生作品。任何未经授权的使用均属侵权行为，我们将依法追究法律责任。如果您希望以其他方式使用本课程的内容，包括但不限于引用、转载、摘录、改编等，请事先与我们联系，获取书面授权。感谢您对“比特就业课”课程的关注与支持，我们将持续努力，为您提供更好的学习体验。特此说明。比特就业课版权所有方。

对比特课程感兴趣，可以联系这个微信。



前面更多讲了RabbitMQ的概念和应用，RabbitMQ实现了AMQP 0-9-1规范的许多扩展，在[RabbitMQ官网](#)上，也给大家介绍了RabbitMQ的一些特性，我们挑一些重要的且常用的给大家讲一下

# Protocol Extensions

## Overview

The RabbitMQ implements a number of extensions of the [AMQP 0-9-1 specification](#), which we document here.

Some extensions introduce new protocol methods (operations); others rely on existing extension points such as [optional queue arguments](#).

## Publishing

- [Publisher Confirms](#) (aka Publisher Acknowledgements) are a lightweight way to know when RabbitMQ has taken responsibility for messages.
- [Blocked Connection Notifications](#) allows clients to be notified when a connection is blocked and unblocked.

## Consuming

- [Consumer Cancellation Notifications](#) let a consumer know if it has been cancelled by the server.
- `basic.nack` extends `basic.reject` to support rejecting multiple messages at once.
- [Consumer Priorities](#) allow you to send messages to higher priority consumers first.
- [Direct reply-to](#) allows RPC clients to receive replies to their queries without needing to declare a temporary queue.

## Message Routing

- [Exchange to Exchange Bindings](#) allow messages to pass through multiple exchanges for more flexible routing.
- [Alternate Exchanges](#) route messages that were otherwise unroutable.

### In This Section

[Server Documentation](#)  
[Client Documentation](#)  
[Plugins](#)  
[News](#)  
[Protocol](#)

#### Our Extensions

[Confirms](#)  
[Consumer Cancel](#)  
[Consumer Prefetch](#)  
[Consumer Priorities](#)  
[Direct reply-to](#)  
[Blocked Connections](#)  
[basic.nack](#)  
[eze bindings](#)  
[Alternate Exchanges](#)  
[Sender Routing](#)  
[TTL](#)  
[Dead Lettering](#)  
[Length Limit](#)  
[Priority Queues](#)  
[Validated User ID](#)  
[Auth Failure](#)  
[Spec Differences](#)

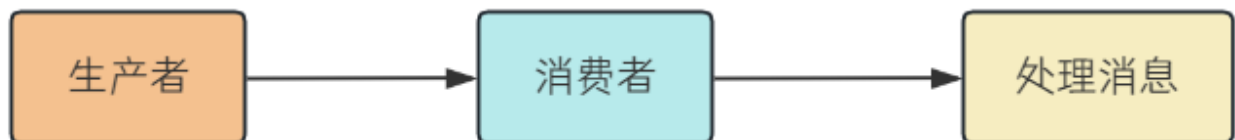
[Building](#)

## 1. 消息确认

### 1.1 消息确认机制

生产者发送消息之后, 到达消费端之后, 可能会有以下情况:

- a. 消息处理成功
- b. 消息处理异常



RabbitMQ向消费者发送消息之后, 就会把这条消息删掉, 那么第两种情况, 就会造成消息丢失.

那么如何确保消费端已经成功接收了, 并正确处理了呢?

为了保证消息从队列可靠地到达消费者, RabbitMQ提供了消息确认机制 (message acknowledgement)。

消费者在订阅队列时, 可以指定 `autoAck` 参数, 根据这个参数设置, 消息确认机制分为以下两种:

- **自动确认:** 当`autoAck` 等于`true`时, `RabbitMQ` 会自动把发送出去的消息置为确认, 然后从内存(或者磁盘)中删除, 而不管消费者是否真正地消费到了这些消息. 自动确认模式适合对于消息可靠性要求不高的场景.
- **手动确认:** 当`autoAck`等于`false`时, `RabbitMQ`会等待消费者显式地调用`Basic.Ack`命令, 回复确认信号后才从内存(或者磁盘) 中移去消息. 这种模式适合对消息可靠性要求比较高的场景.

```

1 /**
2  * Start a non-nolocal, non-exclusive consumer, with
3  * a server-generated consumerTag.
4  * @param queue the name of the queue
5  * @param autoAck true if the server should consider messages
6  * acknowledged once delivered; false if the server should expect
7  * explicit acknowledgements
8  * @param callback an interface to the consumer object
9  * @return the consumerTag generated by the server
10 * @throws java.io.IOException if an error is encountered
11 * @see com.rabbitmq.client.AMQP.Basic.Consume
12 * @see com.rabbitmq.client.AMQP.Basic.ConsumeOk
13 * @see #basicConsume(String, boolean, String, boolean, boolean, Map, Consumer)
14 */
15 String basicConsume(String queue, boolean autoAck, Consumer callback) throws
    IOException;

```

## 代码示例:

```

1 DefaultConsumer consumer = new DefaultConsumer(channel) {
2     @Override
3     public void handleDelivery(String consumerTag, Envelope envelope,
4         AMQP.BasicProperties properties, byte[] body) throws IOException {
5         System.out.println("接收到消息: " + new String(body));
6     }
7 };
8 channel.basicConsume(Constants.TOPIC_QUEUE_NAME1, true, consumer);

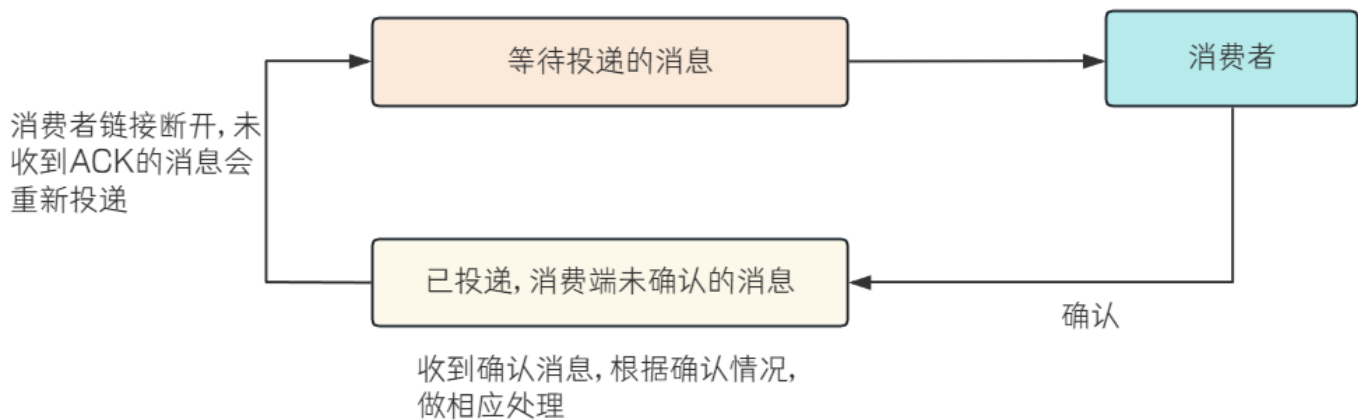
```

当autoAck参数置为false, 对于RabbitMQ服务端而言, 队列中的消息分成了两个部分:

一是等待投递给消费者的消息.

二是已经投递给消费者, 但是还没有收到消费者确认信号的消息.

如果RabbitMQ一直没有收到消费者的确认信号, 并且消费此消息的消费者已经断开连接, 则RabbitMQ会安排该消息重新进入队列, 等待投递给下一个消费者, 当然也有可能还是原来的那个消费者.



从RabbitMQ的Web管理平台上, 也可以看到当前队列中Ready状态和Unacked状态的消息数

Overview   Connections   Channels   Exchanges   **Queues**   Admin

Cluster rabbit@VM-8-12-centos  
User study   Log out

Queues

▼ All queues (8)

Pagination

Page 1 of 1 - Filter:   ☐ Regex ?   Displaying 8 items , page size up to: 100

Overview						Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer capacity	State	Ready	Unacked	Total	incoming	deliver / get	ack
bite	confirm_queue	classic	D	0	0%	idle	13	0	13	0.00/s	0.00/s	0.00/s
bite	fanout_queue_1	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s
bite	fanout_queue_2	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s
bite	simple_queue	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s
bite	topic_queue	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s
bite	topic_queue_error	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s
bite	topic_queue_info	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s
bite	topic_queue_warn	classic	D	0	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s

► Add a new queue

Ready: 等待投递给消费者的消息数

Unacked: 已经投递给消费者, 但是未收到消费者确认信号的消息数

## 1.2 手动确认方法

消费者在收到消息之后, 可以选择确认, 也可以选择直接拒绝或者跳过, RabbitMQ也提供了不同的确认应答的方式, 消费者客户端可以调用与其对应的channel的相关方法, 共有以下三种

### 1. 肯定确认: Channel.basicAck(long deliveryTag, boolean multiple)

RabbitMQ 已知道该消息并且成功的处理消息. 可以将其丢弃了.

参数说明:

- 1) deliveryTag: 消息的唯一标识, 它是一个单调递增的64 位的长整型值. deliveryTag 是每个通道 (Channel) 独立维护的, 所以在每个通道上都是唯一的. 当消费者确认(ack)一条消息时, 必须使用对应的通道上进行确认.
- 2) multiple: 是否批量确认. 在某些情况下, 为了减少网络流量, 可以对一系列连续的 deliveryTag 进行批量确认. 值为 true 则会一次性 ack所有小于或等于指定 deliveryTag 的消息. 值为false, 则只确认当前指定deliveryTag 的消息.

未确认消息



`deliveryTag=8`

`multiple = true`: 4,5,6,7,8 都会被确认

`multiple = false`: 只确认8

`deliveryTag` 是RabbitMQ中消息确认机制的一个重要组成部分, 它确保了消息传递的可靠性和顺序性。

## 2. 否定确认: `Channel.basicReject(long deliveryTag, boolean requeue)`

RabbitMQ在2.0.0版本开始引入了 `Basic.Reject` 这个命令, 消费者客户端可以调用 `channel.basicReject` 方法来告诉RabbitMQ拒绝这个消息。

参数说明:

1) `deliveryTag`: 参考`channel.basicAck`

2) `requeue`: 表示拒绝后, 这条消息如何处理. 如果`requeue` 参数设置为`true`, 则RabbitMQ会重新将这条消息存入队列, 以便可以发送给下一个订阅的消费者. 如果`requeue`参数设置为`false`, 则RabbitMQ会把消息从队列中移除, 而不会把它发送给新的消费者。

## 3. 否定确认: `Channel.basicNack(long deliveryTag, boolean multiple, boolean requeue)`

`Basic.Reject`命令一次只能拒绝一条消息, 如果想要批量拒绝消息, 则可以使用`Basic.Nack`这个命令. 消费者客户端可以调用 `channel.basicNack` 方法来实现。

参数介绍参考上面两个方法。

`multiple`参数设置为`true`则表示拒绝`deliveryTag`编号之前所有未被当前消费者确认的消息。

## 1.3 代码示例

我们基于SpringBoot来演示消息的确认机制, 使用方式和使用RabbitMQ Java Client 库有一定差异。

Spring-AMQP 对消息确认机制提供了三种策略。

```
1 public enum AcknowledgeMode {
```

```
2     NONE,  
3     MANUAL,  
4     AUTO;  
5 }
```

## 1. AcknowledgeMode.NONE

- 这种模式下, 消息一旦投递给消费者, 不管消费者是否成功处理了消息, RabbitMQ 就会自动确认消息, 从RabbitMQ队列中移除消息. 如果消费者处理消息失败, 消息可能会丢失.

## 2. AcknowledgeMode.AUTO(默认)

- 这种模式下, 消费者在消息处理成功时会自动确认消息, 但如果处理过程中抛出了异常, 则不会确认消息.

## 3. AcknowledgeMode.MANUAL

- 手动确认模式下, 消费者必须在成功处理消息后显式调用 `basicAck` 方法来确认消息. 如果消息未被确认, RabbitMQ 会认为消息尚未被成功处理, 并且会在消费者可用时重新投递该消息, 这种模式提高了消息处理的可靠性, 因为即使消费者处理消息后失败, 消息也不会丢失, 而是可以被重新处理.

主要流程:

1. 配置确认机制(自动确认/手动机制)
2. 生产者发送消息
3. 消费端逻辑
4. 测试

### 1.3.1 AcknowledgeMode.NONE

#### 1. 配置确认机制

```
1 spring:  
2   rabbitmq:  
3     addresses: amqp://study:study@110.41.51.65:15673/bite  
4     listener:  
5       simple:  
6         acknowledge-mode: none
```

#### 2. 发送消息

队列,交换机配置

```

1 public class Constant {
2     public static final String ACK_EXCHANGE_NAME = "ack_exchange";
3     public static final String ACK_QUEUE = "ack_queue";
4 }

```

```

1 /*
2  以下为消费端手动应答代码示例配置
3  */
4 @Bean("ackExchange")
5 public Exchange ackExchange(){
6     return
7     ExchangeBuilder.topicExchange(Constant.ACK_EXCHANGE_NAME).durable(true).build()
8     ;
9 }
10 //2. 队列
11 @Bean("ackQueue")
12 public Queue ackQueue() {
13     return QueueBuilder.durable(Constant.ACK_QUEUE).build();
14 }
15 //3. 队列和交换机绑定 Binding
16 @Bean("ackBinding")
17 public Binding ackBinding(@Qualifier("ackExchange") Exchange exchange,
18     @Qualifier("ackQueue") Queue queue) {
19     return BindingBuilder.bind(queue).to(exchange).with("ack").noargs();
20 }

```

通过接口发送消息:

```

1 import com.bite.rabbitmq.constant.Constant;
2 import org.springframework.amqp.rabbit.core.RabbitTemplate;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RestController;
6
7 @RestController
8 @RequestMapping("/producer")
9 public class ProductController {
10     @Autowired
11     private RabbitTemplate rabbitTemplate;
12
13     @RequestMapping("/ack")
14     public String ack(){

```

```

15         rabbitTemplate.convertAndSend(Constant.ACK_EXCHANGE_NAME, "ack",
        "consumer ack test...");
16         return "发送成功!";
17     }
18 }

```

### 3. 写消费端逻辑

```

1 import com.bite.rabbitmq.constant.Constant;
2 import com.rabbitmq.client.Channel;
3 import org.springframework.amqp.core.Message;
4 import org.springframework.amqp.rabbit.annotation.RabbitListener;
5 import org.springframework.stereotype.Component;
6
7 @Component
8 public class AckQueueListener {
9     //指定监听队列的名称
10    @RabbitListener(queues = Constant.ACK_QUEUE)
11    public void ListenerQueue(Message message, Channel channel) throws
    Exception {
12        System.out.printf("接收到消息: %s, deliveryTag: %d\n", new
        String(message.getBody(), "UTF-8"),
        message.getMessageProperties().getDeliveryTag());
13        //模拟处理失败
14        int num = 3/0;
15        System.out.println("处理完成");
16    }

```

这个代码运行的结果是正常的, 运行后消息会被签收: Ready为0, unacked为0

### 4. 运行程序

调用接口, 发送消息

可以看到队列中有一条消息, unacked的为0(需要先把消费者注掉)

#### Queues

► All queues (1)

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	ack_queue	classic	D	0	0%	idle	1	0	1	0.20/s	0.00/s	0.00/s		

开启消费者, 控制台输出:

```

1 接收到消息: consumer ack test..., deliveryTag: 1

```



```

2 2024-04-29T17:03:57.797+08:00 WARN 16952 --- [ntContainer#0-1]
  s.a.r.l.ConditionalRejectingErrorHandler : Execution of Rabbit message
  listener failed.
3
4 org.springframework.amqp.rabbit.support.ListenerExecutionFailedException:
  Listener method 'public void
  com.bite.rabbitmq.listener.AckQueueListener.ListenerQueue(org.springframework.a
  mpq.core.Message,com.rabbitmq.client.Channel) throws java.lang.Exception'
  threw exception
5 //....

```

## 管理界面:

### Queues

► All queues (1)

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		

可以看到, 消费者处理失败, 但是消息已经从RabbitMQ中移除.

## 1.3.2 AcknowledgeMode.AUTO

### 1. 配置确认机制

```

1 配置确认机制
2 spring:
3   rabbitmq:
4     addresses: amqp://study:study@110.41.51.65:15673/bite
5     listener:
6       simple:
7         acknowledge-mode: auto

```

### 2. 重新运行程序

#### 调用接口, 发送消息

可以看到队列中有一条消息, unacked的为0(需要先把消费者注掉)

### Queues

► All queues (1)

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	ack_queue	classic	D	0	0%	idle	1	0	1	0.20/s	0.00/s	0.00/s		

开启消费者, 控制台不断输出错误信息

```
1 接收到消息: consumer ack test..., deliveryTag: 1
2 2024-04-29T17:07:06.114+08:00 WARN 16488 --- [ntContainer#0-1]
   s.a.r.l.ConditionalRejectingErrorHandler : Execution of Rabbit message
   listener failed.
3
4 org.springframework.amqp.rabbit.support.ListenerExecutionFailedException:
   Listener method 'public void
   com.bite.rabbitmq.listener.AckQueueListener.ListenerQueue(org.springframework.a
   mpq.core.Message,com.rabbitmq.client.Channel) throws java.lang.Exception'
   threw exception
5
6
7 接收到消息: consumer ack test..., deliveryTag: 2
8 2024-04-29T17:07:07.161+08:00 WARN 16488 --- [ntContainer#0-1]
   s.a.r.l.ConditionalRejectingErrorHandler : Execution of Rabbit message
   listener failed.
9
10 org.springframework.amqp.rabbit.support.ListenerExecutionFailedException:
   Listener method 'public void
   com.bite.rabbitmq.listener.AckQueueListener.ListenerQueue(org.springframework.a
   mpq.core.Message,com.rabbitmq.client.Channel) throws java.lang.Exception'
   threw exception
11
12
13 接收到消息: consumer ack test..., deliveryTag: 3
14 2024-04-29T17:07:08.208+08:00 WARN 16488 --- [ntContainer#0-1]
   s.a.r.l.ConditionalRejectingErrorHandler : Execution of Rabbit message
   listener failed.
15
16 org.springframework.amqp.rabbit.support.ListenerExecutionFailedException:
   Listener method 'public void
   com.bite.rabbitmq.listener.AckQueueListener.ListenerQueue(org.springframework.a
   mpq.core.Message,com.rabbitmq.client.Channel) throws java.lang.Exception'
   threw exception
17
18
19 接收到消息: consumer ack test..., deliveryTag: 4
20 2024-04-29T17:07:09.254+08:00 WARN 16488 --- [ntContainer#0-1]
   s.a.r.l.ConditionalRejectingErrorHandler : Execution of Rabbit message
   listener failed.
21
22 org.springframework.amqp.rabbit.support.ListenerExecutionFailedException:
   Listener method 'public void
   com.bite.rabbitmq.listener.AckQueueListener.ListenerQueue(org.springframework.a
   mpq.core.Message,com.rabbitmq.client.Channel) throws java.lang.Exception'
   threw exception
23
```

## 管理界面

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	ack_queue	classic	D	1	100%	running	0	1	1	0.00/s	1.0/s	0.00/s	

从日志上可以看出, 当消费者出现异常时, RabbitMQ会不断的重发. 由于异常, 多次重试还是失败, 消息没被确认, 也无法nack, 就一直是unacked状态, 导致消息积压.

### 1.3.3 AcknowledgeMode.MANUAL

#### 1. 配置确认机制

```
1 spring:
2   rabbitmq:
3     addresses: amqp://study:study@110.41.51.65:15673/bite
4     listener:
5       simple:
6         acknowledge-mode: manual
```

#### 2. 消费端手动确认逻辑

```
1 import com.bite.rabbitmq.constant.Constant;
2 import com.rabbitmq.client.Channel;
3 import org.springframework.amqp.core.Message;
4 import org.springframework.amqp.rabbit.annotation.RabbitListener;
5 import org.springframework.stereotype.Component;
6
7 @Component
8 public class AckQueueListener {
9     //指定监听队列的名称
10    @RabbitListener(queues = Constant.ACK_QUEUE)
11    public void ListenerQueue(Message message, Channel channel) throws
12    Exception {
13        long deliveryTag = message.getMessageProperties().getDeliveryTag();
14        try {
15            //1. 接收消息
16            System.out.printf("接收到消息: %s, deliveryTag: %d\n", new
17            String(message.getBody(), "UTF-8"),
18            message.getMessageProperties().getDeliveryTag());
19            //2. 处理业务逻辑
20            System.out.println("处理业务逻辑");
21            //手动设置一个异常, 来测试异常拒绝机制
22            int num = 3/0;
```

```

20          //3. 手动签收
21          channel.basicAck(deliveryTag, true);
22      } catch (Exception e) {
23          //4. 异常了就拒绝签收
24          //第三个参数requeue, 是否重新发送, 如果为true, 则会重新发送,,若为false,
          则直接丢弃
25          channel.basicNack(deliveryTag, true,true);
26      }
27  }
28 }

```

这个代码运行的结果是正常的, 运行后消息会被签收: Ready为0, unacked为0

### 控制台输出:

- 1 接收到消息: consumer ack test..., deliveryTag: 1
- 2 处理业务逻辑

### 管理界面:

#### Queues

► All queues (1)

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	Incoming	deliver / get	ack	
bite	ack_queue	classic		1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	

### 3. 异常时拒绝签收

#### 主动设置异常

```

1 @Component
2 public class AckQueueListener {
3     //指定监听队列的名称
4     @RabbitListener(queues = Constant.ACK_QUEUE)
5     public void ListenerQueue(Message message, Channel channel) throws
        Exception {
6         long deliveryTag = message.getMessageProperties().getDeliveryTag();
7         try {
8             // Thread.sleep(1000);
9             //1. 接收消息
10            System.out.printf("接收到消息: %s, deliveryTag: %d\n", new
                String(message.getBody(), "UTF-8"),
                message.getMessageProperties().getDeliveryTag());
11            //2. 处理业务逻辑
12            System.out.println("处理业务逻辑");

```

```

13          //手动设置一个异常，来测试异常拒绝机制
14          int num = 3/0;
15          //3. 手动签收
16          channel.basicAck(deliveryTag, true);
17      } catch (Exception e) {
18          //4. 异常了就拒绝签收
19          //第三个参数requeue，是否重新发送，如果为true，则会重新发送，若为false，
      则直接丢弃
20          channel.basicNack(deliveryTag, true, true);
21      }
22  }
23 }

```

运行结果: 消费异常时不断重试, deliveryTag 从1递增

控制台日志:

```

1  接收到消息: consumer ack test..., deliveryTag: 1
2  处理业务逻辑
3  接收到消息: consumer ack test..., deliveryTag: 2
4  处理业务逻辑
5  接收到消息: consumer ack test..., deliveryTag: 3
6  处理业务逻辑
7  接收到消息: consumer ack test..., deliveryTag: 4
8  处理业务逻辑
9  接收到消息: consumer ack test..., deliveryTag: 5
10 处理业务逻辑
11 接收到消息: consumer ack test..., deliveryTag: 6
12 处理业务逻辑

```

管理界面上unacked也变成了1

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	ack_queue	classic	D	0	0%	idle	0	1	1	0.00/s	1.0/s	0.00/s		

Unacked的状态变化很快, 为方便观察, 消费消息前增加一下休眠时间

```
Thread.sleep(10000);
```

## 2. 持久性

我们在前面讲了消费端处理消息时, 消息如何不丢失, 但是如何保证当RabbitMQ服务停掉以后, 生产者发送的消息不丢失呢. 默认情况下, RabbitMQ 退出或者由于某种原因崩溃时, 会忽视队列和消息, 除非告知他不要这么做.

RabbitMQ的持久化分为三个部分:交换器的持久化、队列的持久化和消息的持久化.

## 2.1 交换机持久化

交换器的持久化是通过在声明交换机时是将durable参数置为true实现的.相当于将交换机的属性在服务器内部保存,当MQ的服务器发生意外或关闭之后,重启 RabbitMQ 时不需要重新去建立交换机,交换机会自动建立,相当于一直存在.

如果交换机不设置持久化,那么在 RabbitMQ 服务重启之后,相关的交换机元数据会丢失,对一个长期使用的交换机来说,建议将其置为持久化的.

```
1 ExchangeBuilder.topicExchange(Constant.ACK_EXCHANGE_NAME).durable(true).build()
```

## 2.2 队列持久化

队列的持久化是通过在声明队列时将 durable 参数置为 true实现的.

如果队列不设置持久化,那么在RabbitMQ服务重启之后,该队列就会被删掉,此时数据也会丢失.(队列没有了,消息也无处可存了)

队列的持久化能保证该队列本身的元数据不会因异常情况而丢失,但是并不能保证内部所存储的消息不会丢失.要确保消息不会丢失,需要将消息设置为持久化.

咱们前面用的创建队列的方式都是持久化的

```
1 QueueBuilder.durable(Constant.ACK_QUEUE).build();
```

点进去看源码会发现,该方法默认durable是true

```
1 public static QueueBuilder durable(String name) {  
2     return (new QueueBuilder(name)).setDurable();  
3 }
```

```
1 private QueueBuilder setDurable() {  
2     this.durable = true;  
3     return this;  
4 }
```

通过下面代码,可以创建非持久化的队列

```
1 QueueBuilder.nonDurable(Constant.ACK_QUEUE).build();
```

## 2.3 消息持久化

消息实现持久化, 需要把消息的投递模式( `MessageProperties` 中的 `deliveryMode` )设置为2, 也就是 `MessageDeliveryMode.PERSISTENT`

```
1 public enum MessageDeliveryMode {  
2     NON_PERSISTENT, //非持久化  
3     PERSISTENT; //持久化
```

设置了队列和消息的持久化, 当 `RabbitMQ` 服务重启之后, 消息依旧存在. 如果只设置队列持久化, 重启之后消息会丢失. 如果只设置消息的持久化, 重启之后队列消失, 继而消息也丢失. 所以单单设置消息持久化而不设置队列的持久化显得毫无意义.

```
1 //非持久化信息  
2 channel.basicPublish("", QUEUE_NAME, null, msg.getBytes());  
3  
4 //持久化信息  
5 channel.basicPublish("", QUEUE_NAME,  
6     MessageProperties.PERSISTENT_TEXT_PLAIN, msg.getBytes());
```

`MessageProperties.PERSISTENT_TEXT_PLAIN` 实际就是封装了这个属性

```
1 public static final BasicProperties PERSISTENT_TEXT_PLAIN =  
2     new BasicProperties("text/plain",  
3         null,  
4         null,  
5         2, //deliveryMode  
6         0, null, null, null,  
7         null, null, null, null,  
8         null, null);
```

如果使用 `RabbitTemplate` 发送持久化消息, 代码如下:

```
1 // 要发送的消息内容  
2 String message = "This is a persistent message";
```

```

3
4 // 创建一个Message对象, 设置为持久化
5 Message messageObject = new Message(message.getBytes(), new
  MessageProperties());
6 messageObject.getMessageProperties().setDeliveryMode(MessageDeliveryMode.PERSIST
  TENT);
7
8 // 使用RabbitTemplate发送消息
9 rabbitTemplate.convertAndSend(Constant.ACK_EXCHANGE_NAME, "ack",
  messageObject);

```


RabbitMQ默认情况下会将消息视为持久化的, 除非队列被声明为非持久化, 或者消息在发送时被标记为非持久化

我们也可以通过打印Message这个对象, 来观察消息是否持久化

```

1 (Body:'consumer ack test...' MessageProperties [headers={},
  contentType=text/plain, contentEncoding=UTF-8, contentLength=0,
  receivedDeliveryMode=PERSISTENT, priority=0, redelivered=true,
  receivedExchange=ack_exchange, receivedRoutingKey=ack, deliveryTag=2,
  consumerTag=amq.ctag-mtd-2Mec9zH2fXizRqVAqg, consumerQueue=ack_queue])

```

 将所有的消息都设置为持久化, 会严重影响RabbitMQ的性能(随机). 写入磁盘的速度比写入内存的速度慢得不只一点点. 对于可靠性不是那么高的消息可以不采用持久化处理以提高整体的吞吐量. 在选择是否要将消息持久化时, 需要在可靠性和吞吐量之间做一个权衡.

将交换器、队列、消息都设置了持久化之后就能百分之百保证数据不丢失了吗? 答案是否定的.

1. 从消费者来说, 如果在订阅消费队列时将autoAck参数设置为true, 那么当消费者接收到相关消息之后, 还没来得及处理就宕机了, 这样也算数据丢失. 这种情况很好解决, 将autoAck参数设置为false, 并进行手动确认, 详细可以参考[消息确认]章节.
2. 在持久化的消息正确存入RabbitMQ之后, 还需要有一段时间(虽然很短, 但是不可忽视)才能存入磁盘中. RabbitMQ并不会为每条消息都进行同步存盘(调用内核的fsync方法)的处理, 可能仅仅保存到操作系统缓存之中而不是物理磁盘之中. 如果在这段时间内RabbitMQ服务节点发生了宕机、重启等异常情况, 消息保存还没来得及落盘, 那么这些消息将会丢失.

这个问题怎么解决呢?



1. 引入RabbitMQ的仲裁队列(后面再讲), 如果主节点(master)在此特殊时间内挂掉, 可以自动切换到从节点(slave),这样有效地保证了高可用性, 除非整个集群都挂掉(此方法也不能保证100%可靠, 但是配置了仲裁队列要比没有配置仲裁队列的可靠性要高很多, 实际生产环境中的关键业务队列一般都会设置仲裁队列).
2. 还可以在发送端引入事务机制或者发送方确认机制来保证消息已经正确地发送并存储至RabbitMQ中, 详细参考下一个章节内容介绍--"发送方确认"

## 3. 发送方确认

在使用 RabbitMQ的时候, 可以通过消息持久化来解决因为服务器的异常崩溃而导致的消息丢失, 但是还有一个问题, 当消息的生产者将消息发送出去之后, 消息到底有没有正确地到达服务器呢? 如果在消息到达服务器之前已经丢失(比如RabbitMQ重启, 那么RabbitMQ重启期间生产者消息投递失败), 持久化操作也解决不了这个问题, 因为消息根本没有到达服务器, 何谈持久化?

RabbitMQ为我们提供了两种解决方案:

- a. 通过事务机制实现
- b. 通过发送方确认(publisher confirm) 机制实现

事务机制比较消耗性能, 在实际工作中使用也不多, 咱们主要介绍confirm机制来实现发送方的确认.

RabbitMQ为我们提供了两个方式来控制消息的可靠性投递

1. **confirm确认模式**
2. **return退回模式**

### 3.1 confirm确认模式

Producer 在发送消息的时候, 对发送端设置一个ConfirmCallback的监听, 无论消息是否到达Exchange, 这个监听都会被执行, 如果Exchange成功收到, ACK( `Acknowledge character` , 确认字符)为true, 如果没收到消息, ACK就为false.

步骤如下:

1. 配置RabbitMQ
2. 设置确认回调逻辑并发送消息
3. 测试

接下来看实现步骤

1. **配置RabbitMQ**

```

1 spring:
2   rabbitmq:
3     addresses: amqp://study:study@110.41.51.65:15673/bite
4     listener:
5       simple:
6         acknowledge-mode: manual #消息接收确认
7         publisher-confirm-type: correlated #消息发送确认

```

## 2. 设置确认回调逻辑并发送消息

无论消息确认成功还是失败, 都会调用ConfirmCallback的confirm方法. 如果消息成功发送到Broker, ack为true.

如果消息发送失败, ack为false, 并且cause提供失败的原因.

```

1 @Bean("confirmRabbitTemplate")
2 public RabbitTemplate confirmRabbitTemplate(ConnectionFactory
   connectionFactory){
3     RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
4     rabbitTemplate.setConfirmCallback(new RabbitTemplate.ConfirmCallback() {
5         @Override
6         public void confirm(CorrelationData correlationData, boolean ack,
   String cause) {
7             System.out.printf("");
8             if (ack){
9                 System.out.printf("消息接收成功, id:%s \n",
   correlationData.getId());
10            }else {
11                System.out.printf("消息接收失败, id:%s, cause: %s",
   correlationData.getId(), cause);
12            }
13        }
14    });
15    return rabbitTemplate;
16 }

```

```

1
2 @Resource(name = "confirmRabbitTemplate")
3 private RabbitTemplate confirmRabbitTemplate;
4
5 @RequestMapping("/confirm")
6 public String confirm() throws InterruptedException
7     CorrelationData correlationData1 = new CorrelationData("1");

```

```

8     confirmRabbitTemplate.convertAndSend(Constant.CONFIRM_EXCHANGE_NAME,
    "confirm", "confirm test...", correlationData1);
9     return "确认成功";
10 }

```

方法说明:

```

1 public interface ConfirmCallback {
2
3     /**
4      * 确认回调
5      * @param correlationData: 发送消息时的附加信息, 通常用于在确认回调中识别特定的消
    息
6      * @param ack: 交换机是否收到消息, 收到为true, 未收到为false
7      * @param cause: 当消息确认失败时, 这个字符串参数将提供失败的原因. 这个原因可以用于调
    试和错误处理.
8      *
    成功时, cause为null
9      */
10    void confirm(@Nullable CorrelationData correlationData, boolean ack,
    @Nullable String cause);
11
12 }

```

### RabbitTemplate.ConfirmCallback 和 ConfirmListener 区别

在RabbitMQ中, ConfirmListener和ConfirmCallback都是用来处理消息确认的机制, 但它们属于不同的客户端库, 并且使用的场景和方式有所不同.

1. ConfirmListener 是 RabbitMQ Java Client 库中的接口. 这个库是 RabbitMQ 官方提供的一个直接与RabbitMQ服务器交互的客户端库. ConfirmListener 接口提供了两个方法: handleAck 和 handleNack, 用于处理消息确认和否定确认的事件.
2. ConfirmCallback 是 Spring AMQP 框架中的一个接口. 专门为Spring环境设计. 用于简化与RabbitMQ交互的过程. 它只包含一个 confirm 方法, 用于处理消息确认的回调.

在 Spring Boot 应用中, 通常会使用 ConfirmCallback, 因为它与 Spring 框架的其他部分更加整合, 可以利用 Spring 的配置和依赖注入功能. 而在使用 RabbitMQ Java Client 库时, 则可能会直接实现 ConfirmListener 接口, 更直接的与RabbitMQ的Channel交互

### 3. 测试

运行程序, 调用接口 <http://127.0.0.1:8080/product/confirm>

观察控制台, 消息确认成功

- 1 confirm 方法被执行了
- 2 消息发送成功, id:1

接下来把交换机名称改下, 重新运行, 会触发另一个结果

- 1 //发送失败
- 2 rabbitTemplate.convertAndSend("confirm\_exchange1", "confirm", "confirm test...", correlationData1);

运行结果:

- 1 confirm 方法被执行了
- 2 消息发送失败, cause:channel error; protocol method: #method<channel.close>(reply-code=404, reply-text=NOT\_FOUND - no exchange 'confirm\_exchange1' in vhost 'bite', class-id=60, method-id=40)

原因中, 明确显示"no exchange 'confirm\_exchange1' in vhost 'bite'" 也就是说, bite这个虚拟机, 没有名字为confirm\_exchange1的交换机

完整代码:

```
1 public class Constant {
2     public static final String CONFIRM_EXCHANGE_NAME = "confirm_exchange";
3     public static final String CONFIRM_QUEUE = "confirm_queue";
4 }
```

```
1 import org.springframework.amqp.rabbit.connection.ConnectionFactory;
2 import org.springframework.amqp.rabbit.connection.CorrelationData;
3 import org.springframework.amqp.rabbit.core.RabbitTemplate;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6
7 @Configuration
8 public class RabbitTemplateConfig {
9
10     @Bean("rabbitTemplate")
11     public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory){
12         RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
13         return rabbitTemplate;
14     }
15 }
```

```

14     }
15
16     @Bean("confirmRabbitTemplate")
17     public RabbitTemplate confirmRabbitTemplate(ConnectionFactory
connectionFactory){
18         RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
19         rabbitTemplate.setConfirmCallback(new RabbitTemplate.ConfirmCallback()
{
20             @Override
21             public void confirm(CorrelationData correlationData, boolean ack,
String cause) {
22                 System.out.printf("");
23                 if (ack){
24                     System.out.printf("消息接收成功, id:%s \n",
correlationData.getId());
25                 }else {
26                     System.out.printf("消息接收失败, id:%s, cause: %s",
correlationData.getId(), cause);
27                 }
28             }
29         });
30         return rabbitTemplate;
31     }
32 }

```

```

1
2 import com.bite.rabbitmq.constant.Constant;
3 import org.springframework.amqp.core.*;
4 import org.springframework.beans.factory.annotation.Qualifier;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7
8 @Configuration
9 public class RabbitmqConfig {
10
11     //publish-confirm模式
12     //1. 交换机
13     @Bean("confirmExchange")
14     public Exchange confirmExchange() {
15         return
ExchangeBuilder.topicExchange(Constant.CONFIRM_EXCHANGE_NAME).durable(true).bui
ld();
16     }
17
18     //2. 队列

```

```

19     @Bean("confirmQueue")
20     public Queue confirmQueue() {
21         return QueueBuilder.durable(Constant.CONFIRM_QUEUE).build();
22     }
23
24     //3. 队列和交换机绑定 Binding
25     @Bean("confirmBinding")
26     public Binding confirmBinding(@Qualifier("confirmExchange") Exchange
exchange, @Qualifier("confirmQueue") Queue queue) {
27         return
BindingBuilder.bind(queue).to(exchange).with("confirm").noargs();
28     }
29 }

```

```

1 package com.bite.rabbitmq.controller;
2
3 import com.bite.rabbitmq.constant.Constant;
4 import org.springframework.amqp.core.Message;
5 import org.springframework.amqp.core.MessageDeliveryMode;
6 import org.springframework.amqp.core.MessageProperties;
7 import org.springframework.amqp.core.ReturnedMessage;
8 import org.springframework.amqp.rabbit.connection.CorrelationData;
9 import org.springframework.amqp.rabbit.core.RabbitTemplate;
10 import org.springframework.beans.factory.annotation.Autowired;
11 import org.springframework.web.bind.annotation.RequestMapping;
12 import org.springframework.web.bind.annotation.RestController;
13
14 @RestController
15 @RequestMapping("/product")
16 public class ProductController {
17     @Resource(name = "confirmRabbitTemplate")
18     private RabbitTemplate confirmRabbitTemplate;
19
20     @RequestMapping("/confirm")
21     public String confirm() throws InterruptedException {
22
23         CorrelationData correlationData1 = new CorrelationData("1");
24         confirmRabbitTemplate.convertAndSend(Constant.CONFIRM_EXCHANGE_NAME,
"confirm1", "confirm test...", correlationData1);
25         //发送失败
26         // confirmRabbitTemplate.convertAndSend("confirm_exchange1", "confirm",
"confirm test...", correlationData1);
27
28         Thread.sleep(2000);
29         return "确认成功";

```

```
30     }  
31  
32 }
```

## 3.2 return返回模式

消息到达Exchange之后, 会根据路由规则匹配, 把消息放入Queue中. Exchange到Queue的过程, 如果一条消息无法被任何队列消费(即没有队列与消息的路由键匹配或队列不存在等), 可以选择把消息退回给发送者. 消息退回给发送者时, 我们可以设置一个返回回调方法, 对消息进行处理.

步骤如下:

1. 配置RabbitMQ
2. 设置返回回调逻辑并发送消息
3. 测试

接下来看实现步骤

### 1. 配置RabbitMQ

```
1 spring:  
2   rabbitmq:  
3     addresses: amqp://study:study@110.41.51.65:15673/bite  
4     listener:  
5       simple:  
6         acknowledge-mode: manual #消息接收确认  
7         publisher-confirm-type: correlated #消息发送确认
```

### 2. 设置返回回调逻辑并发送消息

消息无法被路由到任何队列, 它将返回给发送者, 这时setReturnCallback设置的回调将被触发

```
1 @Bean("confirmRabbitTemplate")  
2 public RabbitTemplate confirmRabbitTemplate(CachingConnectionFactory  
   connectionFactory){  
3     RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);  
4     rabbitTemplate.setMandatory(true);  
5     rabbitTemplate.setReturnsCallback(new RabbitTemplate.ReturnsCallback() {  
6         @Override  
7         public void returnedMessage(ReturnedMessage returned) {  
8             System.out.printf("消息被退回: %s", returned);  
9         }  
    }
```

```

10     });
11     return rabbitTemplate;
12 }

```

```

1 @RequestMapping("/msgReturn")
2 public String msgReturn(){
3
4     CorrelationData correlationData = new CorrelationData("2");
5     confirmRabbitTemplate.convertAndSend(Constants.CONFIRM_EXCHANGE,
6     "confirm11", "message return test...", correlationData);
7     return "消息发送成功";
8 }

```

使用RabbitTemplate的setMandatory方法设置消息的mandatory属性为true(默认为false). 这个属性的作用是告诉RabbitMQ, 如果一条消息无法被任何队列消费, RabbitMQ应该将消息返回给发送者, 此时 `ReturnCallback` 就会被触发.

回调函数中有一个参数: ReturnedMessage, 包含以下属性:

```

1 public class ReturnedMessage {
2     //返回的消息对象, 包含了消息体和消息属性
3     private final Message message;
4     //由Broker提供的回复码, 表示消息无法路由的原因. 通常是一个数字代码, 每个数字代表不同的含义.
5     private final int replyCode;
6     //一个文本字符串, 提供了无法路由消息的额外信息或错误描述.
7     private final String replyText;
8     //消息被发送到的交换机名称
9     private final String exchange;
10    //消息的路由键, 即发送消息时指定的键
11    private final String routingKey;
12 }

```

### 3. 测试

运行程序, 调用接口 <http://127.0.0.1:8080/product/msgReturn>

观察控制台, 消息被退回

```

1 消息被退回: ReturnedMessage [message=(Body:'confirm test...' MessageProperties
    [headers={spring_returned_message_correlation=2}, contentType=text/plain,

```

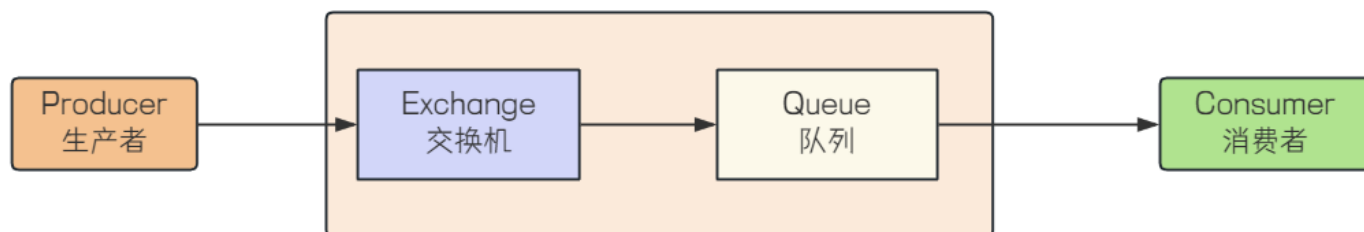


```
contentEncoding=UTF-8, contentType=application/json, receivedDeliveryMode=PERSISTENT, priority=0, deliveryTag=0)), replyCode=312, replyText=NO_ROUTE, exchange=confirm_exchange, routingKey=confirm11]
```

### 3.3 常见面试题

#### 如何保证RabbitMQ消息的可靠传输？

先放一张RabbitMQ消息传递图



从这个图中, 可以看出, 消息可能丢失的场景以及解决方案:

1. 生产者将消息发送到 RabbitMQ 失败
  - a. 可能原因: 网络问题等
  - b. 解决办法: 参考本章节[发送方确认-confirm确认模式]
2. 消息在交换机中无法路由到指定队列:
  - a. 可能原因: 代码或者配置层面错误, 导致消息路由失败
  - b. 解决办法: 参考本章节[发送方确认-return模式]
3. 消息队列自身数据丢失
  - a. 可能原因: 消息到达RabbitMQ之后, RabbitMQ Server 宕机导致消息丢失.
  - b. 解决办法: 参考本章节[持久性]. 开启 RabbitMQ持久化, 就是消息写入之后会持久化到磁盘, 如果 RabbitMQ 挂了, 恢复之后会自动读取之前存储的数据. (极端情况下, RabbitMQ还未持久化就挂了, 可能导致少量数据丢失, 这个概率极低, 也可以通过集群的方式提高可靠性)
4. 消费者异常, 导致消息丢失
  - a. 可能原因: 消息到达消费者, 还没来得及消费, 消费者宕机. 消费者逻辑有问题.
  - b. 解决办法: 参考本章节[消息确认]. RabbitMQ 提供了 消费者应答机制 来使 RabbitMQ 能够感知到消费者是否消费成功消息. 默认情况下消费者应答机制是自动应答的, 可以开启手动确认, 当消费者确认消费成功后才会删除消息, 从而避免消息丢失. 除此之外, 也可以配置重试机制(参考下一章节), 当消息消费异常时, 通过消息重试确保消息的可靠性

## 4. 重试机制

在消息传递过程中,可能会遇到各种问题,如网络故障,服务不可用,资源不足等,这些问题可能导致消息处理失败.为了解决这些问题,RabbitMQ 提供了重试机制,允许消息在处理失败后重新发送.

但如果是程序逻辑引起的错误,那么多次重试也是没有用的,可以设置重试次数

## 4.1 重试配置

```
1 spring:
2   rabbitmq:
3     addresses: amqp://study:study@110.41.51.65:15673/bite
4     listener:
5       simple:
6         acknowledge-mode: auto #消息接收确认
7         retry:
8           enabled: true # 开启消费者失败重试
9           initial-interval: 5000ms # 初始失败等待时长为5秒
10          max-attempts: 5 # 最大重试次数(包括自身消费的一次)
```

## 4.2 配置交换机&队列

```
1 //重试机制
2 public static final String RETRY_QUEUE = "retry_queue";
3 public static final String RETRY_EXCHANGE_NAME = "retry_exchange";
```

```
1 //重试机制 发布订阅模式
2 //1. 交换机
3 @Bean("retryExchange")
4 public Exchange retryExchange() {
5     return
6     ExchangeBuilder.fanoutExchange(Constant.RETRY_EXCHANGE_NAME).durable(true).build();
7 }
8 //2. 队列
9 @Bean("retryQueue")
10 public Queue retryQueue() {
11     return QueueBuilder.durable(Constant.RETRY_QUEUE).build();
12 }
13 //3. 队列和交换机绑定 Binding
14 @Bean("retryBinding")
```

```

15 public Binding retryBinding(@Qualifier("retryExchange") FanoutExchange
    exchange, @Qualifier("retryQueue") Queue queue) {
16     return BindingBuilder.bind(queue).to(exchange);
17 }

```

## 4.3 发送消息

```

1 @RequestMapping("/retry")
2 public String retry(){
3     rabbitTemplate.convertAndSend(Constant.RETRY_EXCHANGE_NAME, "", "retry
    test...");
4     return "发送成功!";
5 }

```

## 4.4 消费消息

```

1 import com.bite.rabbitmq.constant.Constant;
2 import org.springframework.amqp.core.Message;
3 import org.springframework.amqp.rabbit.annotation.RabbitListener;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class RetryQueueListener {
8     //指定监听队列的名称
9     @RabbitListener(queues = Constant.RETRY_QUEUE)
10    public void ListenerQueue(Message message) throws Exception {
11        System.out.printf("接收到消息: %s, deliveryTag: %d\n", new
    String(message.getBody(), "UTF-8"),
    message.getMessageProperties().getDeliveryTag());
12        //模拟处理失败
13        int num = 3/0;
14        System.out.println("处理完成");
15    }
16 }
17 }

```

## 4.5 运行程序, 观察结果

运行程序, 调用接口, 发送消息

<http://127.0.0.1:8080/product/retry>

```

1 接收到消息: retry test..., deliveryTag: 1
2 接收到消息: retry test..., deliveryTag: 1
3 接收到消息: retry test..., deliveryTag: 1
4 接收到消息: retry test..., deliveryTag: 1
5 接收到消息: retry test..., deliveryTag: 1
6 2024-04-29T17:17:21.819+08:00 WARN 32172 --- [ntContainer#0-1]
   o.s.a.r.r.RejectAndDontRequeueRecoverer : Retries exhausted for message
   (Body:'consumer ack test...' MessageProperties [headers={},
   contentType=text/plain, contentEncoding=UTF-8, contentLength=0,
   receivedDeliveryMode=PERSISTENT, priority=0, redelivered=false,
   receivedExchange=ack_exchange, receivedRoutingKey=ack, deliveryTag=1,
   consumerTag=amq.ctag-vYckQBt9_0-5v2oG9oBnFw, consumerQueue=ack_queue])
7
8 org.springframework.amqp.rabbit.support.ListenerExecutionFailedException:
   Listener method 'public void
   com.bite.rabbitmq.listener.AckQueueListener.ListenerQueue(org.springframework.a
   mpq.core.Message,com.rabbitmq.client.Channel) throws java.lang.Exception'
   threw exception

```

如果对异常进行捕获, 那么就不会进行重试

代码修改如下:

```

1 System.out.printf("接收到消息: %s, deliveryTag: %d\n", new
   String(message.getBody(),"UTF-8"),
   message.getMessageProperties().getDeliveryTag());
2 //模拟处理失败
3 try {
4     int num = 3/0;
5     System.out.println("处理完成");
6 }catch (Exception e){
7     System.out.println("处理失败");
8 }

```

重新运行程序, 结果如下:

```

1 接收到消息: consumer ack test..., deliveryTag: 1
2 处理失败

```

## 4.6 手动确认

改为手动确认

```

1 @RabbitListener(queues = Constant.RETRY_QUEUE)
2 public void ListenerQueue(Message message, Channel channel) throws Exception {
3     long deliveryTag = message.getMessageProperties().getDeliveryTag();
4     try {
5         System.out.printf("接收到消息: %s, deliveryTag: %d\n", new
String(message.getBody(), "UTF-8"),
message.getMessageProperties().getDeliveryTag());
6         //模拟处理失败
7         int num = 3/0;
8         System.out.println("处理完成");
9         //3. 手动签收
10        channel.basicAck(deliveryTag, true);
11
12    }catch (Exception e){
13        //4. 异常了就拒绝签收
14        Thread.sleep(1000);
15        //第三个参数requeue, 是否重新发送, 如果为true, 则会重新发送,,若为false, 则直接
        丢弃
16        channel.basicNack(deliveryTag, true,true);
17    }
18 }

```

## 运行结果

```

1 接收到消息: retry test..., deliveryTag: 1
2 接收到消息: retry test..., deliveryTag: 2
3 接收到消息: retry test..., deliveryTag: 3
4 接收到消息: retry test..., deliveryTag: 4
5 接收到消息: retry test..., deliveryTag: 5
6 接收到消息: retry test..., deliveryTag: 6
7 接收到消息: retry test..., deliveryTag: 7
8 接收到消息: retry test..., deliveryTag: 8
9 接收到消息: retry test..., deliveryTag: 9
10 接收到消息: retry test..., deliveryTag: 10
11 接收到消息: retry test..., deliveryTag: 11


```

可以看到, 手动确认模式时, 重试次数的限制不会像在自动确认模式下那样直接生效, 因为是否重试以及何时重试更多地取决于应用程序的逻辑和消费者的实现.

自动确认模式下, RabbitMQ 会在消息被投递给消费者后自动确认消息. 如果消费者处理消息时抛出异常, RabbitMQ 根据配置的重试参数自动将消息重新入队, 从而实现重试. 重试次数和重试间隔等参数可以直接在RabbitMQ的配置中设定, 并且RabbitMQ会负责执行这些重试策略.

手动确认模式下, 消费者需要显式地对消息进行确认. 如果消费者在处理消息时遇到异常, 可以选择不确认消息使消息可以重新入队. 重试的控制权在于应用程序本身, 而不是RabbitMQ的内部机制. 应用程序可以通过自己的逻辑和利用RabbitMQ的高级特性来实现有效的重试策略

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	ack_queue	classic	D	1	0%	Idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	confirm_queue	classic	D	0	0%	Idle	0	0	0				
bite	retry_queue	classic	D	1	100%	running	0	1	1	0.00/s	1.0/s	0.00/s	

-  使用重试机制时需要注意:
1. 自动确认模式下: 程序逻辑异常, 多次重试还是失败, 消息就会被自动确认, 那么消息就丢失了

2. 手动确认模式下: 程序逻辑异常, 多次重试消息依然处理失败, 无法被确认, 就一直是 unacked 的状态, 导致消息积压

## 5. TTL

TTL(Time to Live, 过期时间), 即过期时间. RabbitMQ可以对消息和队列设置TTL.

当消息到达存活时间之后, 还没有被消费, 就会被自动清除

咱们在网上购物, 经常会遇到一个场景, 当下单超过24小时还未付款, 订单会被自动取消  
还有类似的, 申请退款之后, 超过7天未被处理, 则自动退款



### 5.1 设置消息的TTL

目前有两种方法可以设置消息的TTL.

一是设置队列的TTL, 队列中所有消息都有相同的过期时间. 二是对消息本身进行单独设置, 每条消息的TTL可以不同. 如果两种方法一起使用, 则消息的TTL以两者之间较小的那个数值为准.

先看针对每条消息设置TTL

针对每条消息设置TTL的方法是在发送消息的方法中加入expiration的属性参数，单位为毫秒。

## 配置交换机&队列

```
1 //TTL
2 public static final String TTL_QUEUE = "ttl_queue";
3 public static final String TTL_EXCHANGE_NAME = "ttl_exchange";
```

```
1 //ttl
2 //1. 交换机
3 @Bean("ttlExchange")
4 public Exchange ttlExchange() {
5     return
        ExchangeBuilder.fanoutExchange(Constant.TTL_EXCHANGE_NAME).durable(true).build(
        );
6 }
7 //2. 队列
8 @Bean("ttlQueue")
9 public Queue ttlQueue() {
10     return QueueBuilder.durable(Constant.TTL_QUEUE).build();
11 }
12
13 //3. 队列和交换机绑定 Binding
14 @Bean("ttlBinding")
15 public Binding ttlBinding(@Qualifier("ttlExchange") FanoutExchange exchange,
16     @Qualifier("ttlQueue") Queue queue) {
17     return BindingBuilder.bind(queue).to(exchange);
18 }
```

## 发送消息

```
1 @RequestMapping("/ttl")
2 public String ttl(){
3     String ttlTime = "10000";//10s
4     rabbitTemplate.convertAndSend(Constant.TTL_EXCHANGE_NAME, "", "ttl
        test...", messagePostProcessor -> {
5         messagePostProcessor.getMessageProperties().setExpiration(ttlTime);
6         return messagePostProcessor;
7     });
8     return "发送成功!";
9 }
```

运行程序, 观察结果

调用接口, 发送消息

<http://127.0.0.1:8080/product/ttl>

1. 发送消息后, 可以看到, Ready消息为1

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	confirm_queue	classic	D	0	0%	idle	0	0	0					
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	ttl_queue	classic	D	0	0%	idle	1	0	1	0.00/s				

2. 10秒钟之后, 刷新页面, 发现消息已被删除

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	confirm_queue	classic	D	0	0%	idle	0	0	0					
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	ttl_queue	classic	D	0	0%	idle	0	0	0	0.00/s				

如果不设置TTL，则表示此消息不会过期;如果将TTL设置为0，则表示除非此时可以直接将消息投递到消费者，否则该消息会被立即丢弃.

5.2 设置队列的TTL

设置队列TTL的方法是在创建队列时, 加入 `x-message-ttl` 参数实现的, 单位是毫秒.

配置队列和绑定关系

```
1 public static final String TTL_QUEUE2 = "ttl_queue2";
```

```
1 //设置ttl
2 @Bean("ttlQueue2")
3 public Queue ttlQueue2() {
4     //设置20秒过期
5     return QueueBuilder.durable(Constant.TTL_QUEUE2).ttl(20*1000).build();
6 }
7 //3. 队列和交换机绑定 Binding
8 @Bean("ttlBinding2")
9 public Binding ttlBinding2(@Qualifier("ttlExchange") FanoutExchange exchange,
10    @Qualifier("ttlQueue2") Queue queue) {
11     return BindingBuilder.bind(queue).to(exchange);
12 }
```



设置过期时间, 也可以采用以下方式:

```
1 @Bean("ttlQueue2")
2 public Queue ttlQueue2() {
3     Map<String, Object> arguments = new HashMap<>();
4     arguments.put("x-message-ttl",20000);//20秒过期
5     return
6     QueueBuilder.durable(Constant.TTL_QUEUE2).withArguments(arguments).build();
7 }
```

发送消息

```
1 @RequestMapping("/ttl")
2 public String ttl() {
3     // String ttlTime = "30000";//10s
4     // //发送带ttl的消息
5     // rabbitTemplate.convertAndSend(Constant.TTL_EXCHANGE_NAME, "", "ttl
6     // test...", messagePostProcessor -> {
7     //     messagePostProcessor.getMessageProperties().setExpiration(ttlTime);
8     //     return messagePostProcessor;
9     // });
10    // //发送不带ttl的消息
11    rabbitTemplate.convertAndSend(Constant.TTL_EXCHANGE_NAME, "", "ttl
12    test...");
13    return "发送成功!";
14 }
```

运行程序, 观察结果

运行之后发现,新增了一个队列, 队列Features有一个TTL标识

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	Incoming	deliver / get	ack		
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	confirm_queue	classic	D	0	0%	idle	0	0	0					
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	ttl_queue	classic	D	0	0%	idle	0	0	0	0.00/s				
bite	ttl_queue2	classic	D TTL	0	0%	idle	0	0	0	0.00/s				

调用接口, 发送消息:

<http://127.0.0.1:8080/product/ttl>

1. 发送消息后, 可以看到, Ready消息为1

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	confirm_queue	classic	D	0	0%	idle	0	0	0				
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	ttl_queue	classic	D	0	0%	idle	1	0	1	0.20/s			
bite	ttl_queue2	classic	D TTL	0	0%	idle	1	0	1	0.20/s			

采用发布订阅模式, 所有与该交换机绑定的队列(ttl\_queue和ttl\_queue2)都会收到消息

2. 20秒钟之后, 刷新页面, 发现消息已被删除

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	confirm_queue	classic	D	0	0%	idle	0	0	0				
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	ttl_queue	classic	D	0	0%	idle	1	0	1	0.00/s			
bite	ttl_queue2	classic	D TTL	0	0%	idle	0	0	0	0.00/s			

由于ttl\_queue队列, 未设置过期时间, 所以ttl\_queue的消息未删除

5.3 两者区别

设置队列TTL属性的方法, 一旦消息过期, 就会从队列中删除

设置消息TTL的方法, 即使消息过期, 也不会马上从队列中删除, 而是在即将投递到消费者之前进行判定的.

为什么这两种方法处理的方式不一样?

因为设置队列过期时间, 队列中已过期的消息肯定在队列头部, RabbitMQ只要定期从队头开始扫描是否有过期的消息即可.

而设置消息TTL的方式, 每条消息的过期时间不同, 如果要删除所有过期消息需要扫描整个队列, 所以不如等到此消息即将被消费时再判定是否过期, 如果过期再进行删除即可.

5.4 完整代码

配置项

```
1 //TTL
2 public static final String TTL_QUEUE = "ttl_queue";
3 public static final String TTL_EXCHANGE_NAME = "ttl_exchange";
4
5 public static final String TTL_QUEUE2 = "ttl_queue2";
```

## 队列, 交换机, 绑定关系

```
1 //ttl 发布订阅模式
2 //1. 交换机
3 @Bean("ttlExchange")
4 public FanoutExchange ttlExchange() {
5     return
6     ExchangeBuilder.fanoutExchange(Constant.TTL_EXCHANGE_NAME).durable(true).build(
7 );
8 }
9 //2. 队列
10 @Bean("ttlQueue")
11 public Queue ttlQueue() {
12     return QueueBuilder.durable(Constant.TTL_QUEUE).build();
13 }
14 //设置ttl
15 @Bean("ttlQueue2")
16 public Queue ttlQueue2() {
17     Map<String, Object> arguments = new HashMap<>();
18     arguments.put("x-message-ttl", 20000); //20秒过期
19     return
20     QueueBuilder.durable(Constant.TTL_QUEUE2).withArguments(arguments).build();
21 }
22 //3. 队列和交换机绑定 Binding
23 @Bean("ttlBinding")
24 public Binding ttlBinding(@Qualifier("ttlExchange") FanoutExchange exchange,
25 @Qualifier("ttlQueue") Queue queue) {
26     return BindingBuilder.bind(queue).to(exchange);
27 }
28 //队列和交换机绑定 Binding
29 @Bean("ttlBinding2")
30 public Binding ttlBinding2(@Qualifier("ttlExchange") FanoutExchange exchange,
31 @Qualifier("ttlQueue2") Queue queue) {
32     return BindingBuilder.bind(queue).to(exchange);
33 }
```

## 发送消息

```
1 @RequestMapping("/ttl")
2 public String ttl() {
3     // String ttlTime = "10000"; //10s
```

```

4 //          //发送带ttl的消息
5 //          rabbitTemplate.convertAndSend(Constant.TTL_EXCHANGE_NAME, "", "ttl
  test...", messagePostProcessor -> {
6 //
  messagePostProcessor.getMessageProperties().setExpiration(ttlTime);
7 //          return messagePostProcessor;
8 //      });
9
10         //发送不带ttl的消息
11         rabbitTemplate.convertAndSend(Constant.TTL_EXCHANGE_NAME, "", "ttl
  test...");
12         return "发送成功!";
13     }

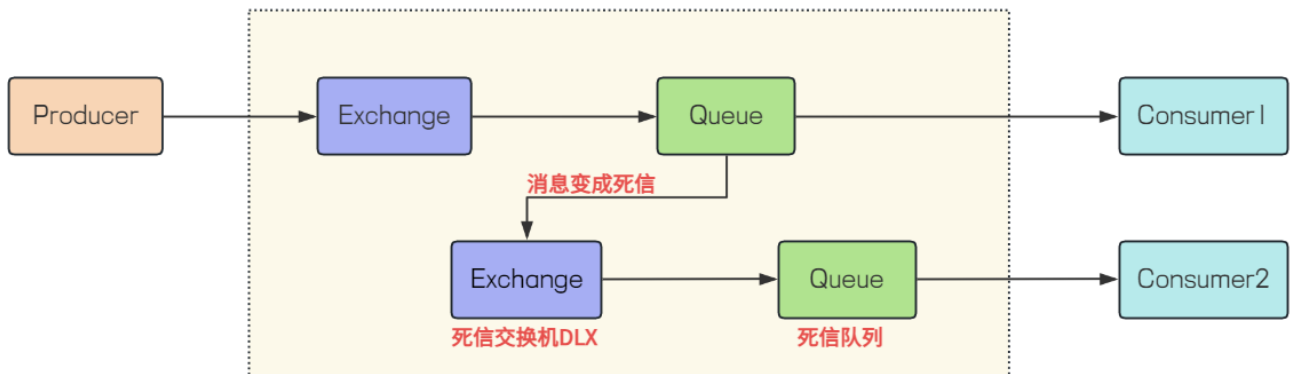
```

## 6. 死信队列

### 6.1 死信的概念

死信(dead message) 简单理解就是因为种种原因, 无法被消费的信息, 就是死信。

有死信, 自然就有死信队列. 当消息在一个队列中变成死信之后, 它会被重新被发送到另一个交换器中, 这个交换器就是DLX(Dead Letter Exchange), 绑定DLX的队列, 就称为死信队列(Dead Letter Queue, 简称DLQ).



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.

### 6.2 代码示例

#### 6.2.1 声明队列和交换机

包含两部分:

- 声明正常的队列和正常的交换机
- 声明死信队列和死信交换机

死信交换机和死信队列和普通的交换机, 队列没有区别

```
1 //死信队列
2 public static final String DLX_EXCHANGE_NAME = "dlx_exchange";
3 public static final String DLX_QUEUE = "dlx_queue";
4 public static final String NORMAL_EXCHANGE_NAME = "normal_exchange";
5 public static final String NORMAL_QUEUE = "normal_queue";
```

```
1
2 import org.springframework.amqp.core.*;
3 import org.springframework.beans.factory.annotation.Qualifier;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6 import rabbitmq.Constant;
7
8 /**
9  * 死信队列相关配置
10 */
11 @Configuration
12 public class DLXConfig {
13     //死信交换机
14     @Bean("dlxExchange")
15     public Exchange dlxExchange(){
16         return
17         ExchangeBuilder.topicExchange(Constant.DLX_EXCHANGE_NAME).durable(true).build()
18     };
19     //2. 死信队列
20     @Bean("dlxQueue")
21     public Queue dlxQueue() {
22         return QueueBuilder.durable(Constant.DLX_QUEUE).build();
23     }
24
25     //3. 死信队列和交换机绑定 Binding
26     @Bean("dlxBinding")
27     public Binding dlxBinding(@Qualifier("dlxExchange") Exchange exchange,
28                               @Qualifier("dlxQueue") Queue queue) {
29         return BindingBuilder.bind(queue).to(exchange).with("dlx").noargs();
30     }
31 }
```

```

29     }
30     //正常交换机
31     @Bean("normalExchange")
32     public Exchange normalExchange(){
33         return
ExchangeBuilder.topicExchange(Constant.NORMAL_EXCHANGE_NAME).durable(true).build();
34     }
35     //正常队列
36     @Bean("normalQueue")
37     public Queue normalQueue() {
38         return QueueBuilder.durable(Constant.NORMAL_QUEUE).build();
39     }
40
41     //正常队列和交换机绑定 Binding
42     @Bean("normalBinding")
43     public Binding normalBinding(@Qualifier("normalExchange") Exchange
exchange, @Qualifier("normalQueue") Queue queue) {
44         return BindingBuilder.bind(queue).to(exchange).with("normal").noargs();
45     }
46 }

```

## 6.2.2 正常队列绑定死信交换机

当这个队列中存在死信时, RabbitMQ会自动的把这个消息重新发布到设置的DLX上, 进而被路由到另一个队列, 即死信队列. 可以监听这个死信队列中的消息以进行相应的处理

```

1 @Bean("normalQueue")
2 public Queue normalQueue() {
3     Map<String, Object> arguments = new HashMap<>();
4     arguments.put("x-dead-letter-exchange", Constant.DLX_EXCHANGE_NAME); //绑定死
信队列
5     arguments.put("x-dead-letter-routing-key", "dlx"); //设置发送给死信队列的
RoutingKey
6     return
QueueBuilder.durable(Constant.NORMAL_QUEUE).withArguments(arguments).build();
7 }

```

简写为:

```

1 return QueueBuilder.durable(Constant.NORMAL_QUEUE)
2     .deadLetterExchange(Constant.DLX_EXCHANGE_NAME)
3     .deadLetterRoutingKey("dlx").build();

```

### 6.2.3 制造死信产生的条件

```
1 @Bean("normalQueue")
2 public Queue normalQueue() {
3     Map<String, Object> arguments = new HashMap<>();
4     arguments.put("x-dead-letter-exchange", Constant.DLX_EXCHANGE_NAME); //绑定死
    信队列
5     arguments.put("x-dead-letter-routing-key", "dlx"); //设置发送给死信队列的
    RoutingKey
6     //制造死信产生的条件
7     arguments.put("x-message-ttl", 10000); //10秒过期
8     arguments.put("x-max-length", 10); //队列长度
9     return
    QueueBuilder.durable(Constant.NORMAL_QUEUE).withArguments(arguments).build();
10 }
```

简写为:

```
1 return QueueBuilder.durable(Constant.NORMAL_QUEUE)
2     .deadLetterExchange(Constant.DLX_EXCHANGE_NAME)
3     .deadLetterRoutingKey("dlx")
4     .ttl(10*1000)
5     .maxLength(10L)
6     .build();
```

### 6.2.4 发送消息

```
1 @RequestMapping("/dlx")
2 public void dlx() {
3     //测试过期时间，当时间达到TTL，消息自动进入到死信队列
4     rabbitTemplate.convertAndSend(Constant.NORMAL_EXCHANGE_NAME, "normal",
    "dlx test...");
5
6     //测试队列长度
7     // for (int i = 0; i < 20; i++) {
8     //     rabbitTemplate.convertAndSend(Constant.NORMAL_EXCHANGE_NAME,
    "normal", "dlx test...");
9     // }
10 //     //测试消息拒收
```

```

11 //      rabbitTemplate.convertAndSend(Constant.NORMAL_EXCHANGE_NAME,
    "normal", "dlx test...");
12     }

```

## 6.2.5 测试死信

### 1. 程序启动之后, 观察队列

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	confirm_queue	classic	D	0	0%	idle	0	0	0				
bite	dlx_queue	classic	D	0	0%	idle	0	0	0				
bite	normal_queue	classic	D TTL Lim DLX DLK	0	0%	idle	0	0	0	0.00/s			
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	ttl_queue	classic	D	0	0%	idle	0	0	0	0.00/s			
bite	ttl_queue2	classic	D TTL	0	0%	idle	0	0	0	0.00/s			

队列Features说明:

D: durable的缩写, 设置持久化

TTL: Time to Live, 队列设置了TTL

Lim: 队列设置了长度(x-max-length)

DLX: 队列设置了死信交换机(x-dead-letter-exchange)

DLK: 队列设置了死信RoutingKey(x-dead-letter-routing-key)

### 2. 测试过期时间, 到达过期时间之后, 进入死信队列

调用接口, 发送消息: <http://127.0.0.1:8080/product/dlx>

发送之后:

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	confirm_queue	classic	D	0	0%	idle	0	0	0				
bite	dlx_queue	classic	D	0	0%	idle	0	0	0				
bite	normal_queue	classic	D TTL Lim DLX DLK	0	0%	idle	1	0	1	0.20/s			
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	ttl_queue	classic	D	0	0%	idle	0	0	0	0.00/s			
bite	ttl_queue2	classic	D TTL	0	0%	idle	0	0	0	0.00/s			

10秒后, 消息进入到死信队列



Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	confirm_queue	classic	D	0	0%	idle	0	0	0					
bite	dlx_queue	classic	D	0	0%	idle	1	0	1					
bite	normal_queue	classic	D TTL Lim DLX DLK	0	0%	idle	0	0	0	0.00/s				
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	ttr_queue	classic	D	0	0%	idle	0	0	0	0.00/s				
bite	ttr_queue2	classic	D TTL	0	0%	idle	0	0	0	0.00/s				

生产者首先发送一条消息,然后经过交换器(normal\_exchange)顺利地存储到队列(normal\_queue)中. 由于队列normal\_queue设置了过期时间为10s, 在这10s内没有消费者消费这条消息, 那么判定这条消息过期. 由于设置了DLX, 过期之时, 消息会被丢给交换器(dlx\_exchange)中, 这时根据RoutingKey匹配, 找到匹配的队列(dlx\_queue), 最后消息被存储在queue.dlx这个死信队列中.

### 3. 测试达到队列长度, 消息进入死信队列

队列长度设置为10, 我们发送20条数据, 会有10条数据直接进入到死信队列

发送前, 死信队列只有一条数据

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	confirm_queue	classic	D	0	0%	idle	0	0	0					
bite	dlx_queue	classic	D	0	0%	idle	1	0	1					
bite	normal_queue	classic	D TTL Lim DLX DLK	0	0%	idle	0	0	0	0.00/s				
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	ttr_queue	classic	D	0	0%	idle	0	0	0	0.00/s				
bite	ttr_queue2	classic	D TTL	0	0%	idle	0	0	0	0.00/s				

发送20条消息

```

1 //测试队列长度
2 for (int i = 0; i < 20; i++) {
3     rabbitTemplate.convertAndSend(Constant.NORMAL_EXCHANGE_NAME, "normal",
4         "dlx test...");
5 }

```

运行后, 可以看到死信队列变成了11条

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	confirm_queue	classic	D	0	0%	idle	0	0	0					
bite	dlx_queue	classic	D	0	0%	idle	11	0	11					
bite	normal_queue	classic	D TTL Lim DLX DLK	0	0%	idle	10	0	10	4.0/s				
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	ttr_queue	classic	D	0	0%	idle	0	0	0	0.00/s				
bite	ttr_queue2	classic	D TTL	0	0%	idle	0	0	0	0.00/s				

过期之后, 正常队列的10条也会进入到死信队列

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	confirm_queue	classic	D	0	0%	idle	0	0	0					
bite	dlx_queue	classic	D	0	0%	idle	21	0	21					
bite	normal_queue	classic	D TTL Lim DLX DLK	0	0%	idle	0	0	0	0.00/s				
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	ttl_queue	classic	D	0	0%	idle	0	0	0	0.00/s				
bite	ttl_queue2	classic	D TTL	0	0%	idle	0	0	0	0.00/s				

## 4. 测试消息拒收

写消费者代码, 并强制异常, 测试拒绝签收

```

1  import com.bite.rabbitmq.constant.Constant;
2  import com.rabbitmq.client.Channel;
3  import org.springframework.amqp.core.Message;
4  import org.springframework.amqp.rabbit.annotation.RabbitListener;
5  import org.springframework.stereotype.Component;
6
7  @Component
8  public class DlxQueueListener {
9      //指定监听队列的名称
10     @RabbitListener(queues = Constant.NORMAL_QUEUE)
11     public void ListenerQueue(Message message, Channel channel) throws
Exception {
12         long deliveryTag = message.getMessageProperties().getDeliveryTag();
13         try {
14             System.out.printf("接收到消息: %s, deliveryTag: %d\n", new
String(message.getBody(), "UTF-8"),
message.getMessageProperties().getDeliveryTag());
15             //模拟处理失败
16             int num = 3/0;
17             System.out.println("处理完成");
18             //3. 手动签收
19             channel.basicAck(deliveryTag, true);
20
21         } catch (Exception e){
22             //4. 异常了就拒绝签收
23             Thread.sleep(1000);
24             //第三个参数requeue, 是否重新发送, 如果为true, 则会重新发送,,若为false,
则直接丢弃, 若设置死信, 会进入到死信队列
25             channel.basicNack(deliveryTag, true, false);
26         }
27     }
28
29     //指定监听队列的名称
30     @RabbitListener(queues = Constant.DLX_QUEUE)

```

```
31     public void ListenerDLXQueue(Message message, Channel channel) throws
        Exception {
32         long deliveryTag = message.getMessageProperties().getDeliveryTag();
33         System.out.printf("死信队列接收到消息: %s, deliveryTag: %d\n", new
            String(message.getBody(), "UTF-8"),
            message.getMessageProperties().getDeliveryTag());
34     }
35
36 }
```

发送消息, 观察运行结果

- 1 接收到消息: dlx test..., deliveryTag: 1
- 2 死信队列接收到消息: dlx test..., deliveryTag: 1

## 6.3 常见面试题

死信队列作为RabbitMQ的高级特性,也是面试的一大重点.

### 1. 死信队列的概念

死信 (Dead Letter) 是消息队列中的一种特殊消息, 它指的是那些无法被正常消费或处理的消息. 在消息队列系统中, 如RabbitMQ, 死信队列用于存储这些死信消息

### 2. 死信的来源

- 1) 消息过期: 消息在队列中存活的时间超过了设定的TTL
- 2) 消息被拒绝: 消费者在处理消息时, 可能因为消息内容错误, 处理逻辑异常等原因拒绝处理该消息. 如果拒绝时指定不重新入队(requeue=false), 消息也会成为死信.
- 3) 队列满了: 当队列达到最大长度, 无法再容纳新的消息时, 新来的消息会被处理为死信.

### 3. 死信队列的应用场景

对于RabbitMQ来说, 死信队列是一个非常有用的特性. 它可以处理异常情况下, 消息不能够被消费者正确消费而被置入死信队列中的情况, 应用程序可以通过消费这个死信队列中的内容来分析当时所遇到的异常情况, 进而可以改善和优化系统.

比如: 用户支付订单之后, 支付系统会给订单系统返回当前订单的支付状态

为了保证支付信息不丢失, 需要使用到死信队列机制. 当消息消费异常时, 将消息投入到死信队列中, 由订单系统的其他消费者来监听这个队列, 并对数据进行处理(比如发送工单等, 进行人工确认).

场景的应用场景还有:

- 消息重试: 将死信消息重新发送到原队列或另一个队列进行重试处理.
- 消息丢弃: 直接丢弃这些无法处理的消息, 以避免它们占用系统资源.

- 日志收集：将死信消息作为日志收集起来，用于后续分析和问题定位。

## 7. 延迟队列

### 7.1 概念

延迟队列(Delayed Queue)，即消息被发送以后，并不想让消费者立刻拿到消息，而是等待特定时间后，消费者才能拿到这个消息进行消费。

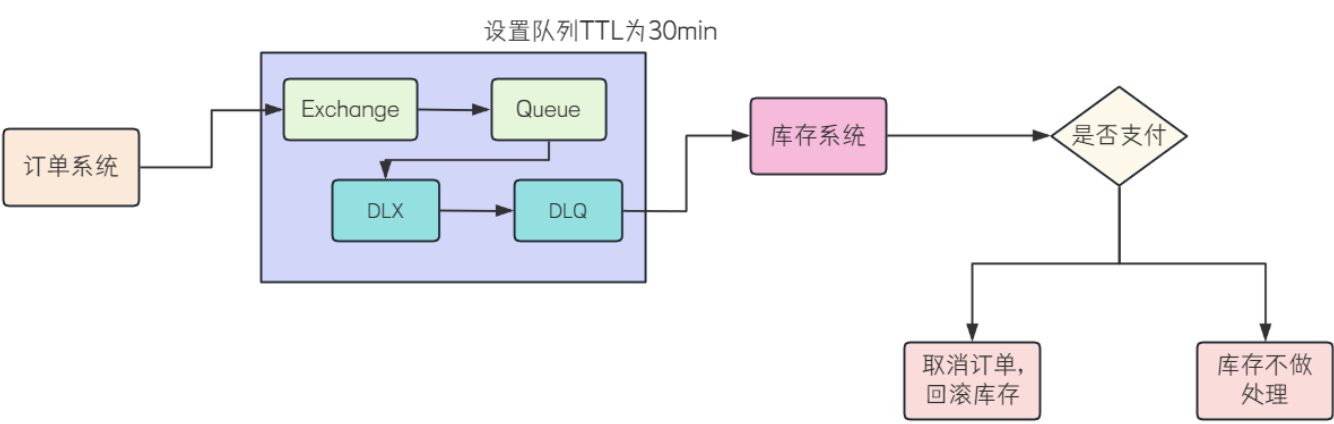
### 7.2 应用场景

延迟队列的使用场景有很多，比如：

1. 智能家居：用户希望通过手机远程遥控家里的智能设备在指定的时间进行工作。这时候就可以将用户指令发送到延迟队列，当指令设定的时间到了再将指令推送到智能设备。
2. 日常管理：预定会议后，需要在会议开始前十五分钟提醒参会人参加会议
3. 用户注册成功后，7天后发送短信，提高用户活跃度等
4. ....

RabbitMQ本身没有直接支持延迟队列的功能，但是可以通过前面所介绍的TTL+死信队列的方式组合模拟出延迟队列的功能。

假设一个应用中需要将每条消息都设置为10秒的延迟，生产者通过 `normal_exchange` 这个交换器将发送的消息存储在 `normal_queue` 这个队列中。消费者订阅的并非是 `normal_queue` 这个队列，而是 `dlx_queue` 这个队列。当消息从 `normal_queue` 这个队列中过期之后被存入 `dlx_queue` 这个队列中，消费者就恰巧消费到了延迟10秒的这条消息。



所以死信队列章节展示的也是延迟队列的使用。

### 7.3 TTL+死信队列实现

#### 代码实现

先看TTL+死信队列实现延迟队列

继续沿用死信队列的代码即可

声明队列:

```
1 //正常队列
2 @Bean("normalQueue")
3 public Queue normalQueue() {
4     Map<String, Object> arguments = new HashMap<>();
5     arguments.put("x-dead-letter-exchange", Constant.DLX_EXCHANGE_NAME); //绑定死
    信队列
6     arguments.put("x-dead-letter-routing-key", "dlx"); //设置发送给死信队列的
    RoutingKey
7     return
    QueueBuilder.durable(Constant.NORMAL_QUEUE).withArguments(arguments).build();
8
9 }
```

生产者:

发送两条消息, 一条消息10s后过期, 第二条20s后过期

```
1 @RequestMapping("/delay")
2 public String delay() {
3
4     //发送带ttl的消息
5     rabbitTemplate.convertAndSend(Constant.NORMAL_EXCHANGE_NAME, "normal",
    "ttl test 10s..." + new Date(), messagePostProcessor -> {
6         messagePostProcessor.getMessageProperties().setExpiration("10000");
7         //10s过期
8         return messagePostProcessor;
9     });
10    rabbitTemplate.convertAndSend(Constant.NORMAL_EXCHANGE_NAME, "normal",
    "ttl test 20s..." + new Date(), messagePostProcessor -> {
11        messagePostProcessor.getMessageProperties().setExpiration("20000");
12        //20s过期
13        return messagePostProcessor;
14    });
15 }
```

消费者:

```

1 //指定监听队列的名称
2 @RabbitListener(queues = Constant.DLX_QUEUE)
3 public void ListenerDLXQueue(Message message, Channel channel) throws
    Exception {
4     long deliveryTag = message.getMessageProperties().getDeliveryTag();
5     System.out.printf("%tc 死信队列接收到消息: %s, deliveryTag: %d%n", new
        Date(), new String(message.getBody(), "UTF-8"),
        message.getMessageProperties().getDeliveryTag());
6 }

```

## 运行程序:

调用接口, 发送数据: <http://127.0.0.1:8080/product/delay>

通过控制台观察死信队列消费情况:

```

1 周三 5月 22 11:59:00 CST 2024 死信队列接收到消息: ttl test 10s...Wed May 22
  11:58:50 CST 2024, deliveryTag: 1
2 周三 5月 22 11:59:10 CST 2024 死信队列接收到消息: ttl test 20s...Wed May 22
  11:58:50 CST 2024, deliveryTag: 2

```

可以看到, 两条消息按照过期时间依次进入了死信队列.

延迟队列, 就是希望等待特定的时间之后, 消费者才能拿到这个消息. TTL刚好可以让消息延迟一段时间成为死信, 成为死信的消息会被投递到死信队列里, 这样消费者一直消费死信队列里的消息就可以了.

## 存在问题

接下来把生产消息的顺序修改一下

先发送20s过期数据, 再发送10s过期数据

```

1 @RequestMapping("/delay")
2 public String delay() {
3
4     //发送带ttl的消息
5     rabbitTemplate.convertAndSend(Constant.NORMAL_EXCHANGE_NAME, "normal",
        "ttl test 20s..." + new Date(), messagePostProcessor -> {
6         messagePostProcessor.getMessageProperties().setExpiration("20000");
        //20s过期
7         return messagePostProcessor;
8     });
9     rabbitTemplate.convertAndSend(Constant.NORMAL_EXCHANGE_NAME, "normal",
        "ttl test 10s..." + new Date(), messagePostProcessor -> {

```

```
10         messagePostProcessor.getMessageProperties().setExpiration("10000");  
           //10s过期  
11         return messagePostProcessor;  
12     });  
13     return "发送成功!";  
14 }
```

通过控制台观察死信队列消费情况:

```
1 周三 5月 22 12:14:22 CST 2024 死信队列接收到消息: ttl test 20s...Wed May 22  
   12:14:02 CST 2024, deliveryTag: 3  
2 周三 5月 22 12:14:22 CST 2024 死信队列接收到消息: ttl test 10s...Wed May 22  
   12:14:02 CST 2024, deliveryTag: 4
```

这时会发现: 10s过期的消息, 也是在20s后才进入到死信队列.

消息过期之后, 不一定会被马上丢弃. 因为RabbitMQ只会检查队首消息是否过期, 如果过期则丢到死信队列. 此时就会造成一个问题, 如果第一个消息的延时时间很长, 第二个消息的延时时间很短, 那第二个消息并不会优先得到执行.

所以在考虑使用TTL+死信队列实现延迟任务队列的时候, 需要确认业务上每个任务的延迟时间是一致的, 如果遇到不同的任务类型需要不同的延迟的话, 需要为每一种不同延迟时间的消息建立单独的消息队列.

## 7.4 延迟队列插件

RabbitMQ官方也提供了一个延迟的插件来实现延迟的功能

参考: <https://www.rabbitmq.com/blog/2015/04/16/scheduling-messages-with-rabbitmq>

接下来看具体操作:




### 安装延迟队列插件

#### 1. 下载并上传插件

插件下载地址:

<https://github.com/rabbitmq/rabbitmq-delayed-message-exchange/releases>

## ▼ Assets 3

 rabbitmq_delayed_message_exchange-3.13.0.ez	44.2 KB	Mar 12
 Source code (zip)		Mar 12
 Source code (tar.gz)		Mar 12

 1  1  8  1  1 8 people reacted

根据自己的RabbitMQ版本选择相应版本的延迟插件, 下载后上传到服务器

插件上传目录参考: [installing Additional Plugins | RabbitMQ](#)

`/usr/lib/rabbitmq/plugins` 是一个附加目录, RabbitMQ包本身不会在此安装任何内容, 如果没有这个路径, 可以自己进行创建

如果为docker操作

使用以下命令复制文件到docker容器

```
1 docker cp 宿主机文件 容器名称或ID:容器目录
```

```
1 root@iZ2vc7a1n9gvhfp589oav7Z:~# docker cp
  /root/temp/rabbitmq_delayed_message_exchange-3.13.0.ez 0de863077982:/plugins
2 Successfully copied 47.1kB to 0de863077982:/opt/rabbitmq/plugins
3 root@iZ2vc7a1n9gvhfp589oav7Z:~#
```

## 2. 启动插件

```
1 #查看插件列表
2 rabbitmq-plugins list
3
4 #启动插件
5 rabbitmq-plugins enable rabbitmq_delayed_message_exchange
6
7 #重启服务
8 service rabbitmq-server restart
```

执行结果:

```
1 root@iZ2vc7a1n9gvhfp589oav7Z:/usr/lib/rabbitmq/plugins# rabbitmq-plugins list
  #查看插件列表
```



```

2 Listing plugins with pattern ".*" ...
3 Configured: E = explicitly enabled; e = implicitly enabled
4 | Status: * = running on rabbit@iZ2vc7a1n9gvhfp589oav7Z
5 | /
6 [ ] rabbitmq_amqp1_0          3.9.13
7 [ ] rabbitmq_auth_backend_cache 3.9.13
8 [ ] rabbitmq_auth_backend_http 3.9.13
9 [ ] rabbitmq_auth_backend_ldap 3.9.13
10 [ ] rabbitmq_auth_backend_oauth2 3.9.13
11 [ ] rabbitmq_auth_mechanism_ssl 3.9.13
12 [ ] rabbitmq_consistent_hash_exchange 3.9.13
13 [ ] rabbitmq_delayed_message_exchange 3.9.0
14 [ ] rabbitmq_event_exchange 3.9.13
15 [ ] rabbitmq_federation 3.9.13
16 [ ] rabbitmq_federation_management 3.9.13
17 [ ] rabbitmq_jms_topic_exchange 3.9.13
18 //...
19 root@iZ2vc7a1n9gvhfp589oav7Z:/usr/lib/rabbitmq/plugins# rabbitmq-plugins
enable rabbitmq_delayed_message_exchange #启动插件
20 Enabling plugins on node rabbit@iZ2vc7a1n9gvhfp589oav7Z:
21 rabbitmq_delayed_message_exchange
22 The following plugins have been configured:
23 rabbitmq_delayed_message_exchange
24 rabbitmq_management
25 rabbitmq_management_agent
26 rabbitmq_web_dispatch
27 Applying plugin configuration to rabbit@iZ2vc7a1n9gvhfp589oav7Z...
28 The following plugins have been enabled:
29 rabbitmq_delayed_message_exchange
30
31 started 1 plugins.
32 root@iZ2vc7a1n9gvhfp589oav7Z:/usr/lib/rabbitmq/plugins# service rabbitmq-
server restart #重启RabbitMQ服务
33 root@iZ2vc7a1n9gvhfp589oav7Z:/usr/lib/rabbitmq/plugins#
34

```

如果为docker操作

查看插件中是否包含延迟队列插件

```

1 #进入容器内容
2 docker exec -it <container_id_or_name> /bin/bash
3
4 #查看插件列表
5 rabbitmq-plugins list
6

```

```
7 #启动插件
8 rabbitmq-plugins enable rabbitmq_delayed_message_exchange
9
10 #重启docker
11 docker restart <container_id_or_name>
```

```
1 root@iZ2vc7a1n9gvhfp589oav7Z:~# docker exec -it 0de863077982 /bin/bash #进入容器内部
2 root@0de863077982:/# rabbitmq-plugins list #查看插件列表
3 Listing plugins with pattern ".*" ...
4 Configured: E = explicitly enabled; e = implicitly enabled
5 | Status: * = running on rabbit@0de863077982
6 | /
7 [e*] oauth2_client 3.13.2
8 [ ] rabbitmq_amqp1_0 (pending upgrade to 3.13.2)
9 [ ] rabbitmq_auth_backend_cache (pending upgrade to 3.13.2)
10 [ ] rabbitmq_auth_backend_http (pending upgrade to 3.13.2)
11 [ ] rabbitmq_auth_backend_ldap (pending upgrade to 3.13.2)
12 [ ] rabbitmq_auth_backend_oauth2 (pending upgrade to 3.13.2)
13 [ ] rabbitmq_auth_mechanism_ssl (pending upgrade to 3.13.2)
14 [ ] rabbitmq_consistent_hash_exchange (pending upgrade to 3.13.2)
15 [ ] rabbitmq_delayed_message_exchange (pending upgrade to 3.13.0)
16 [ ] rabbitmq_event_exchange (pending upgrade to 3.13.2)
17 [e*] rabbitmq_federation 3.13.2
18 root@0de863077982:/# rabbitmq-plugins enable rabbitmq_delayed_message_exchange #启动插件
19 Enabling plugins on node rabbit@0de863077982:
20 rabbitmq_delayed_message_exchange
21 The following plugins have been configured:
22   oauth2_client
23   rabbitmq_delayed_message_exchange
24   rabbitmq_federation
25   rabbitmq_management
26   rabbitmq_management_agent
27   rabbitmq_prometheus
28   rabbitmq_web_dispatch
29 Applying plugin configuration to rabbit@0de863077982...
30 The following plugins have been enabled:
31   rabbitmq_delayed_message_exchange
32
33 started 1 plugins.
34 root@0de863077982:/# exit #退出当前容器会话
35 exit
36 root@iZ2vc7a1n9gvhfp589oav7Z:~# docker restart 0de863077982 #重启指定容器
37 0de863077982
```

```
38 root@iZ2vc7a1n9gvhfp589oav7Z:~#
39
```

### 3. 验证插件

在 RabbitMQ 管理平台查看, 新建交换机时是否有延迟消息选项, 如果有就说明延迟消息插件已经正常运行了

Overview

Connections

Channels

Exchanges

Queues

Admin

## Exchanges

► All exchanges (14)

Virtual host	Name	Type	Features	Message rate in	Message rate out	+/-
/	(AMQP default)	direct	D			
/	amq.direct	direct	D			
/	amq.fanout	fanout	D			
/	amq.headers	headers	D			
/	amq.match	headers	D			
/	amq.rabbitmq.trace	topic	D I			
/	amq.topic	topic	D			
bite	(AMQP default)	direct	D			
bite	amq.direct	direct	D			
bite	amq.fanout	fanout	D			
bite	amq.headers	headers	D			
bite	amq.match	headers	D			
bite	amq.rabbitmq.trace	topic	D I			
bite	amq.topic	topic	D			

▼ Add a new exchange

Virtual host:

/ ▼

Name:

Type:

direct ▼

direct

fanout

headers

topic

x-delayed-message

Durability:

Auto delete:

?

Internal:

?

## 基于插件延迟队列实现

### 1. 声明交换机, 队列, 绑定关系

```
1 import com.bite.rabbitmq.constant.Constant;
2 import org.springframework.amqp.core.*;
3 import org.springframework.beans.factory.annotation.Qualifier;
4 import org.springframework.context.annotation.Bean;
```

```

5 import org.springframework.context.annotation.Configuration;
6
7 @Configuration
8 public class DelayedConfig {
9     @Bean("delayedExchange")
10    public Exchange delayedExchange(){
11        return
12        ExchangeBuilder.directExchange(Constant.DELAYED_EXCHANGE_NAME).durable(true).de
13        layed().build();
14    }
15    //2. 队列
16    @Bean("delayedQueue")
17    public Queue delayedQueue() {
18        return QueueBuilder.durable(Constant.DELAYED_QUEUE).build();
19    }
20    //3. 队列和交换机绑定 Binding
21    @Bean("delayedBinding")
22    public Binding delayedBinding(@Qualifier("delayedExchange") Exchange
23    exchange, @Qualifier("delayedQueue") Queue queue) {
24        return
25        BindingBuilder.bind(queue).to(exchange).with("delayed").noargs();
26    }
27 }

```

## 2. 生产者

发送两条消息, 并设置延迟时间

```

1 @RequestMapping("/delay2")
2 public String delay2() {
3     //发送带ttl的消息
4     rabbitTemplate.convertAndSend(Constant.DELAYED_EXCHANGE_NAME, "delayed",
5     "delayed test 20s..." + new Date(), messagePostProcessor -> {
6         messagePostProcessor.getMessageProperties().setDelayLong(20000L);
7         //20s过期
8         return messagePostProcessor;
9     });
10    rabbitTemplate.convertAndSend(Constant.DELAYED_EXCHANGE_NAME, "delayed",
11    "delayed test 10s..." + new Date(), messagePostProcessor -> {
12        messagePostProcessor.getMessageProperties().setDelayLong(10000L);
13        //10s过期
14        return messagePostProcessor;
15    });
16    return "发送成功!";
17 }

```

### 3. 消费者

```
1 import com.bite.rabbitmq.constant.Constant;
2 import com.rabbitmq.client.Channel;
3 import org.springframework.amqp.core.Message;
4 import org.springframework.amqp.rabbit.annotation.RabbitListener;
5 import org.springframework.stereotype.Component;
6
7 import java.util.Date;
8
9 @Component
10 public class DelayedQueueListener {
11     //指定监听队列的名称
12     @RabbitListener(queues = Constant.DELAYED_QUEUE)
13     public void ListenerDLXQueue(Message message, Channel channel) throws
14         Exception {
15         System.out.printf("%tc 死信队列接收到消息: %s\n", new Date(), new
16             String(message.getBody(),"UTF-8"));
17     }
18 }
```

### 4. 运行程序, 并测试

程序启动后, 观察交换机

bite	amq.fanout	fanout	D		
bite	amq.headers	headers	D		
bite	amq.match	headers	D		
bite	amq.rabbitmq.trace	topic	D I		
bite	amq.topic	topic	D		
bite	confirm_exchange	topic	D		
bite	delayed_exchange	x-delayed-message	D DM Args	0.00/s	

调用接口, 发送消息

<http://127.0.0.1:8080/product/delay2>

观察控制台

```
1 周三 5月 22 15:42:02 CST 2024 死信队列接收到消息: delayed test 10s...Wed May 22
   15:41:52 CST 2024
```

从结果可以看出, 使用延迟队列, 可以保证消息按照延迟时间到达消费者.

## 7.5 常见面试题

延迟队列作为RabbitMQ的高级特性,也是面试的一大重点.

### 介绍下RabbitMQ的延迟队列

延迟队列是一个特殊的队列, 消息发送之后, 并不立即给消费者, 而是等待特定的时间, 才发送给消费者.

延迟队列的应用场景有很多, 比如:

1. 订单在十分钟内未支付自动取消
2. 用户注册成功后, 3天后发调查问卷
3. 用户发起退款, 24小时后商家未处理, 则默认同意, 自动退款
4. ....

但RabbitMQ本身并没直接实现延迟队列, 通常有两种方法:

1. TTL+死信队列组合的方式
2. 使用官方提供的延迟插件实现延迟功能

### 二者对比:

1. 基于死信实现的延迟队列
  - a. 优点: 1) 灵活不需要额外的插件支持
  - b. 缺点: 1) 存在消息顺序问题 2) 需要额外的逻辑来处理死信队列的消息, 增加了系统的复杂性
2. 基于插件实现的延迟队列
  - a. 优点: 1) 通过插件可以直接创建延迟队列, 简化延迟消息的实现. 2) 避免了DLX的时序问题
  - b. 缺点: 1) 需要依赖特定的插件, 有运维工作 2) 只适用特定版本

## 8. 事务

RabbitMQ是基于AMQP协议实现的, 该协议实现了事务机制, 因此RabbitMQ也支持事务机制. Spring AMQP也提供了对事务相关的操作. RabbitMQ事务允许开发者确保消息的发送和接收是原子性的, 要么全部成功, 要么全部失败.

### 8.1 配置事务管理器

```

1 import org.springframework.amqp.core.Queue;
2 import org.springframework.amqp.core.QueueBuilder;
3 import org.springframework.amqp.rabbit.connection.CachingConnectionFactory;
4 import org.springframework.amqp.rabbit.core.RabbitTemplate;
5 import org.springframework.amqp.rabbit.transaction.RabbitTransactionManager;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.context.annotation.Configuration;
8
9 @Configuration
10 public class TransactionConfig {
11     @Bean
12     public RabbitTransactionManager
13     transactionManager(CachingConnectionFactory connectionFactory){
14         return new RabbitTransactionManager(connectionFactory);
15     }
16     @Bean
17     public RabbitTemplate rabbitTemplate(CachingConnectionFactory
18     connectionFactory){
19         RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
20         rabbitTemplate.setChannelTransacted(true);
21         return rabbitTemplate;
22     }
23 }

```

## 8.2 声明队列

```

1 @Bean("transQueue")
2 public Queue transQueue() {
3     return QueueBuilder.durable("trans_queue").build();
4 }

```

## 8.3 生产者

```

1 import org.springframework.amqp.rabbit.core.RabbitTemplate;
2 import org.springframework.beans.factory.annotation.Autowired;
3 import org.springframework.transaction.annotation.Transactional;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RestController;
6
7 @RequestMapping("/trans")
8 @RestController
9 public class TransactionProducer {

```

```

10
11     @Autowired
12     private RabbitTemplate rabbitTemplate;
13
14     @Transactional
15     @RequestMapping("/send")
16     public String send(){
17         rabbitTemplate.convertAndSend("", "trans_queue", "trans test 1...");
18         int a = 5/0;
19         rabbitTemplate.convertAndSend("", "trans_queue", "trans test 2...");
20         return "发送成功";
21     }
22 }

```

## 8.4 测试

1. 不加 `@Transactional` , 会发现消息1发送成功
2. 添加 `@Transactional` , 消息1和消息2全部发送失败

## 9. 消息分发

### 9.1 概念

RabbitMQ队列拥有多个消费者时, 队列会把收到的消息分派给不同的消费者. 每条消息只会发送给订阅列表里的一个消费者. 这种方式非常适合扩展, 如果现在负载加重, 那么只需要创建更多的消费者来消费处理消息即可.

默认情况下, RabbitMQ是以轮询的方法进行分发的, 而不管消费者是否已经消费并已经确认了消息. 这种方式是不太合理的, 试想一下, 如果某些消费者消费速度慢, 而某些消费者消费速度快, 就可能会导致某些消费者消息积压, 某些消费者空闲, 进而应用整体的吞吐量下降.

如何处理呢? 我们可以使用前面章节讲到的`channel.basicQos(int prefetchCount)` 方法, 来限制当前信道上的消费者所能保持的最大未确认消息的数量

比如: 消费端调用了 `channel.basicQos(5)` , RabbitMQ会为该消费者计数, 发送一条消息计数+1, 消费一条消息计数-1, 当达到了设定的上限, RabbitMQ就不会再向它发送消息了, 直到消费者确认了某条消息. 类似TCP/IP中的"滑动窗口".

prefetchCount 设置为0时表示没有上限.

basicQos 对拉模式的消费无效(后面再讲)

### 9.2 应用场景

消息分发的常见应用场景有如下:



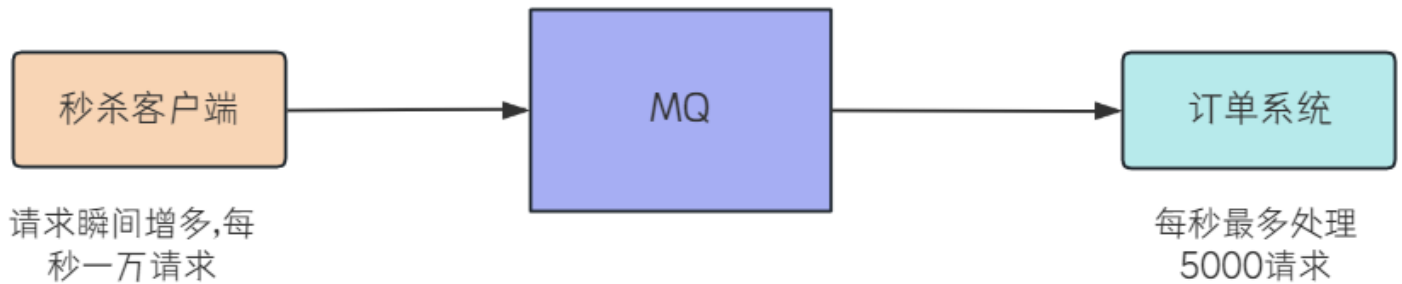
1. 限流
2. 非公平分发

### 9.2.1 限流

如下使用场景:

订单系统每秒最多处理5000请求, 正常情况下, 订单系统可以正常满足需求

但是在秒杀时间点, 请求瞬间增多, 每秒1万个请求, 如果这些请求全部通过MQ发送到订单系统, 无疑会把订单系统压垮.



RabbitMQ提供了限流机制, 可以控制消费端一次只拉取N个请求

通过设置prefetchCount参数, 同时也必须要设置消息应答方式为手动应答

prefetchCount: 控制消费者从队列中预取(prefetch)消息的数量, 以此来实现流控制和负载均衡.

代码示例:

#### 1. 配置prefetch参数, 设置应答方式为手动应答

```
1 #ack 确认方式:开启ack
2 listener:
3   simple:
4     acknowledge-mode: manual #手动确认
5     prefetch: 5
```

#### 2. 配置交换机, 队列

```
1 import com.bite.rabbitmq.constant.Constant;
2 import org.springframework.amqp.core.*;
3 import org.springframework.beans.factory.annotation.Qualifier;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6
```

```

7 @Configuration
8 public class QosConfig {
9     //限流
10    @Bean("qosExchange")
11    public Exchange qosExchange(){
12        return
13        ExchangeBuilder.directExchange(Constant.QOS_EXCHANGE_NAME).durable(true).build(
14    );
15    }
16    //2. 队列
17    @Bean("qosQueue")
18    public Queue qosQueue() {
19        return QueueBuilder.durable(Constant.QOS_QUEUE).build();
20    }
21    //3. 队列和交换机绑定 Binding
22    @Bean("qosBinding")
23    public Binding qosBinding(@Qualifier("qosExchange") Exchange exchange,
24    @Qualifier("qosQueue") Queue queue) {
25        return BindingBuilder.bind(queue).to(exchange).with("qos").noargs();
26    }
27 }

```

### 3. 发送消息, 一次发送20条消息

```

1 @RequestMapping("/qos")
2 public String qos() {
3     //发送消息
4     for (int i = 0; i < 20; i++) {
5         rabbitTemplate.convertAndSend(Constant.QOS_EXCHANGE_NAME, "qos", "qos
6         test..." + i);
7     }
8     return "发送成功!";
9 }

```

### 4. 消费者监听

```

1 @Component
2 public class QosQueueListener {
3     //指定监听队列的名称
4     @RabbitListener(queues = Constant.QOS_QUEUE)
5     public void ListenerQueue(Message message, Channel channel) throws
6     Exception {
7
8     }
9 }

```

```

6         long deliveryTag = message.getMessageProperties().getDeliveryTag();
7         System.out.printf("接收到消息: %s, deliveryTag: %d\n", new
    String(message.getBody(),"UTF-8"), deliveryTag);
8         //3. 手动签收
9         //         channel.basicAck(deliveryTag, true);
10    }
11 }

```

## 5. 测试

调用接口, 发送消息

发送消息时, 需要先把手动确认注掉, 不然会直接消费掉

可以看到, 控制台只打印了5条消息

```

1 接收到消息: qos test...0, deliveryTag: 1
2 接收到消息: qos test...1, deliveryTag: 2
3 接收到消息: qos test...2, deliveryTag: 3
4 接收到消息: qos test...3, deliveryTag: 4
5 接收到消息: qos test...4, deliveryTag: 5

```

我们观察管理平台

Overview							Messages			Message rates				+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack		
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	confirm_queue	classic	D	0	0%	idle	0	0	0					
bite	dlx_queue	classic	D	1	0%	idle	0	0	0		0.00/s	0.00/s		
bite	normal_queue	classic	D TTL Lim DLX DLK	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	qos_queue	classic	D	1	0%	idle	15	5	20	0.00/s	0.00/s	0.00/s		
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s		
bite	ttr_queue	classic	D	0	0%	idle	0	0	0	0.00/s				
bite	ttr_queue2	classic	D TTL	0	0%	idle	0	0	0	0.00/s				

可以看到, ready, 也就是待发送15条, 未确认的5条(因为代码未手动ack)

把 `prefetch: 5` 注掉, 再观察运行结果

从日志和控制台上可以看到: 消费者会一次性把20条消息全部收到

日志:

```

1 接收到消息: qos test...0, deliveryTag: 1
2 接收到消息: qos test...1, deliveryTag: 2
3 接收到消息: qos test...2, deliveryTag: 3
4 接收到消息: qos test...3, deliveryTag: 4

```

```

5  接收到消息: qos test...4, deliveryTag: 5
6  接收到消息: qos test...5, deliveryTag: 6
7  接收到消息: qos test...6, deliveryTag: 7
8  接收到消息: qos test...7, deliveryTag: 8
9  接收到消息: qos test...8, deliveryTag: 9
10 接收到消息: qos test...9, deliveryTag: 10
11 接收到消息: qos test...10, deliveryTag: 11
12 接收到消息: qos test...11, deliveryTag: 12
13 接收到消息: qos test...12, deliveryTag: 13
14 接收到消息: qos test...13, deliveryTag: 14
15 接收到消息: qos test...14, deliveryTag: 15
16 接收到消息: qos test...15, deliveryTag: 16
17 接收到消息: qos test...16, deliveryTag: 17
18 接收到消息: qos test...17, deliveryTag: 18
19 接收到消息: qos test...18, deliveryTag: 19
20 接收到消息: qos test...19, deliveryTag: 20

```

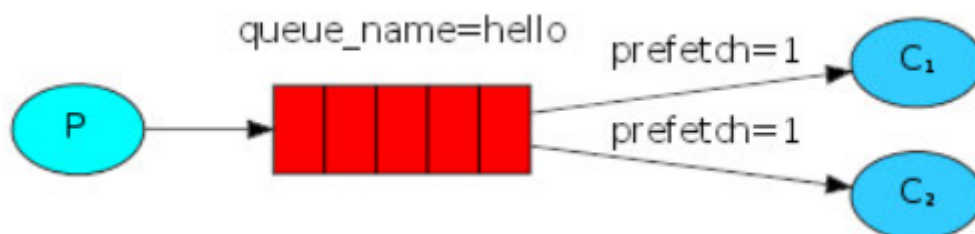
管理平台:

Overview							Messages			Message rates			+/-
Virtual host	Name	Type	Features	Consumers	Consumer utilisation	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	ack_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	confirm_queue	classic	D	0	0%	idle	0	0	0				
bite	dlx_queue	classic	D	1	0%	idle	0	0	0		0.00/s	0.00/s	
bite	normal_queue	classic	D TTL Lim DLX DLK	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	qos_queue	classic	D	1	0%	idle	0	20	20	0.00/s	0.00/s	0.00/s	
bite	retry_queue	classic	D	1	0%	idle	0	0	0	0.00/s	0.00/s	0.00/s	
bite	ttl_queue	classic	D	0	0%	idle	0	0	0	0.00/s			
bite	ttl_queue2	classic	D TTL	0	0%	idle	0	0	0	0.00/s			

## 9.2.2 负载均衡

我们也可以使用此配置,来实现"负载均衡"

如下图所示, 在有两个消费者的情况下, 一个消费者处理任务非常快, 另一个非常慢, 就会造成一个消费者会一直很忙, 而另一个消费者很闲. 这是因为 RabbitMQ 只是在消息进入队列时分派消息. 它不考虑消费者未确认消息的数量.



我们可以使用设置`prefetch=1`的方式, 告诉 RabbitMQ 一次只给一个消费者一条消息, 也就是说, 在处理并确认前一条消息之前, 不要向该消费者发送新消息. 相反, 它会将它分派给下一个不忙的消费者.

## 代码示例:

### 1. 配置`prefetch`参数, 设置应答方式为手动应答

```
1 #ack 确认方式:开启ack
2 listener:
3   simple:
4     acknowledged-mode: manual #手动确认
5     prefetch: 1
```

### 2. 启动两个消费者

使用 `Thread.sleep(100)` 模拟消费慢

```
1 import com.bite.rabbitmq.constant.Constant;
2 import com.rabbitmq.client.Channel;
3 import org.springframework.amqp.core.Message;
4 import org.springframework.amqp.rabbit.annotation.RabbitListener;
5 import org.springframework.stereotype.Component;
6
7 @Component
8 public class QosQueueListener {
9     //指定监听队列的名称
10    @RabbitListener(queues = Constant.QOS_QUEUE)
11    public void ListenerQosQueue(Message message, Channel channel) throws
12    Exception {
13        long deliveryTag = message.getMessageProperties().getDeliveryTag();
14        System.out.printf("接收到消息: %s, deliveryTag: %d\n", new
15    String(message.getBody(), "UTF-8"), deliveryTag);
16        //3. 手动签收
17        channel.basicAck(deliveryTag, true);
18    }
19
20    //指定监听队列的名称
21    @RabbitListener(queues = Constant.QOS_QUEUE)
22    public void ListenerQueue2(Message message, Channel channel) throws
23    Exception {
24        long deliveryTag = message.getMessageProperties().getDeliveryTag();
25        System.out.printf("消费者2接收到消息: %s, deliveryTag: %d\n", new
26    String(message.getBody(), "UTF-8"), deliveryTag);
27        //模拟处理流程慢
```

```
24         Thread.sleep(100);
25         //3. 手动签收
26         channel.basicAck(deliveryTag, true);
27     }
28
29 }
```

### 3. 测试

调用接口, 发送消息

通过日志观察两个消费者消费的消息

```
1  接收到消息: qos test...1, deliveryTag: 1
2  消费者2接收到消息: qos test...0, deliveryTag: 1
3  接收到消息: qos test...2, deliveryTag: 2
4  接收到消息: qos test...3, deliveryTag: 3
5  接收到消息: qos test...4, deliveryTag: 4
6  接收到消息: qos test...5, deliveryTag: 5
7  消费者2接收到消息: qos test...6, deliveryTag: 2
8  接收到消息: qos test...7, deliveryTag: 6
9  接收到消息: qos test...8, deliveryTag: 7
10 接收到消息: qos test...9, deliveryTag: 8
11 接收到消息: qos test...10, deliveryTag: 9
12 消费者2接收到消息: qos test...11, deliveryTag: 3
13 接收到消息: qos test...12, deliveryTag: 10
14 接收到消息: qos test...13, deliveryTag: 11
15 接收到消息: qos test...14, deliveryTag: 12
16 接收到消息: qos test...15, deliveryTag: 13
17 消费者2接收到消息: qos test...16, deliveryTag: 4
18 接收到消息: qos test...17, deliveryTag: 14
19 接收到消息: qos test...18, deliveryTag: 15
20 接收到消息: qos test...19, deliveryTag: 16
21
```

deliveryTag 有重复是因为两个消费者使用的是不同的Channel, 每个 Channel 上的 deliveryTag 是独立计数的.