

Rabbit MQ学习

Rabbit MQ是一个消息中间件，它接受并转发消息。可以把它当做一个快递站点，当需要发送一个包裹时，把包裹放在快递站，快递员最终会把包裹送到收件人处。按照这个逻辑，Rabbit MQ是一个快递站，一个快递员帮助传递快件。Rabbit MQ与快递站的区别是，它不处理快件而是接收、存储和转发消息数据。

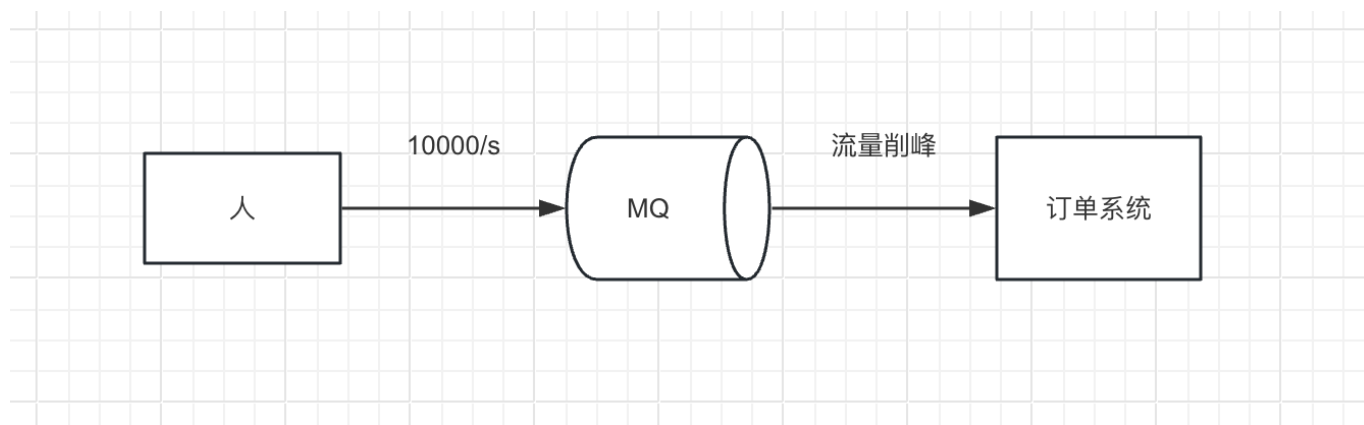
MQ基本概念

为什么要使用MQ

流量削峰

如果订单系统最多能处理一万次订单，这个处理能力正常时绰绰有余。但是高峰时两万次下单就无法处理，只能限制订单超过一万后不允许用户下单，使用消息队列做缓冲（排队），我们可以取消这个限制，把一秒内下单分散成一段时间处理，这时候有些用户可能需要下单后十几秒才有成功反馈，但总比不能下单的体验要好。

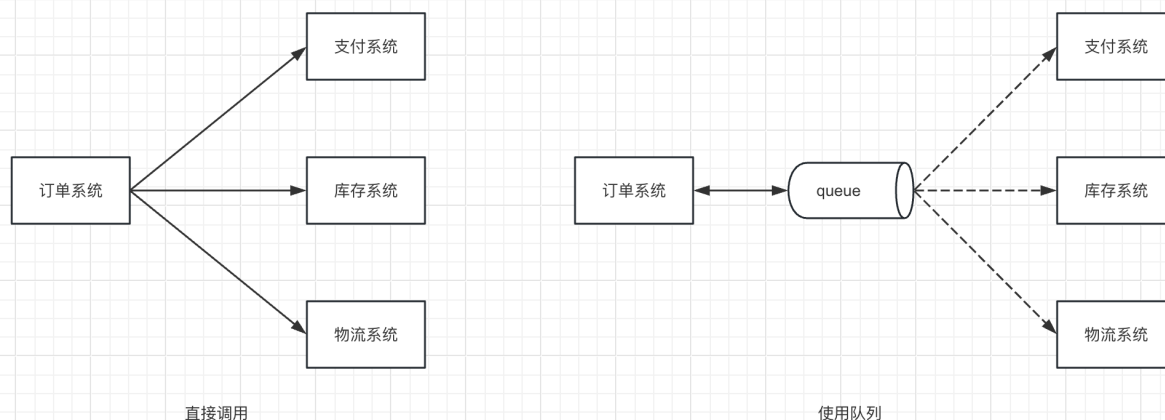
图：img/mq1-llx.png



降低耦合调用给用户带来的不流畅体验

假如一个系统有订单系统、库存系统、物流系统、支付系统。用户创建订单后如果耦合调用库存系统、物流系统、支付系统，任何一个系统出故障订单都会异常。当转变为基于队列的方式后，系统间调用的问题会少很多。假如物流系统发生故障需要几分钟修复，在这几分钟里物流系统要处理的内存被缓存在消息队列中，用户的下单可以正常完成。当物流系统恢复后，继续处理订单信息即可，中单用户不会感受到系统故障，提升系统的可用性。

图：img/mq2-yyjo.png



异步处理

有些服务间调用是异步的，例如A调用B，B需要花很长时间执行，但是A需要知道B什么时候可以完成执行。

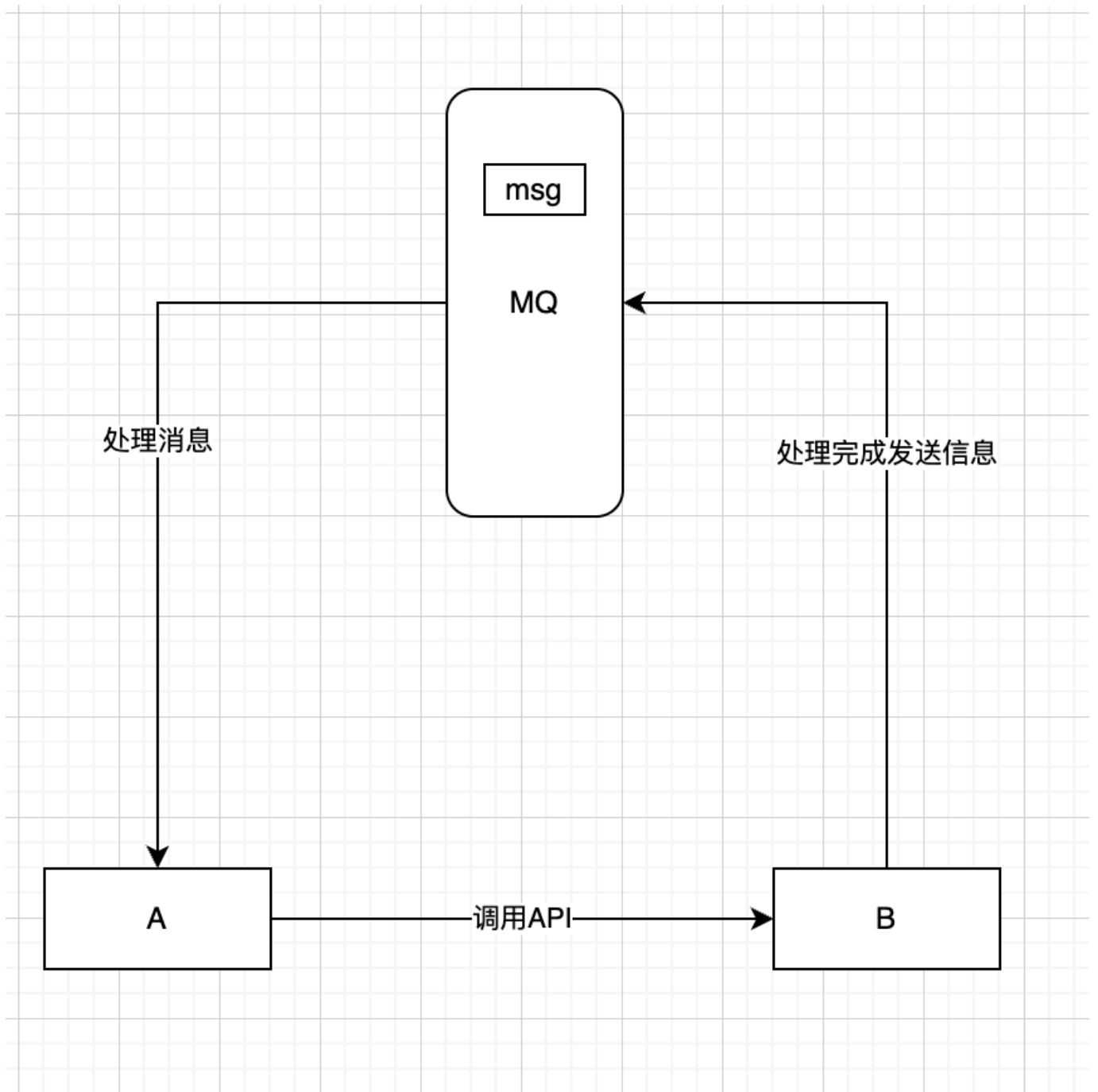
以前一般有两种形式：

- A过一段时间调用B的API查询
- A提供一个CALL_BACK的API，B执行接收函调用API通知A服务

使用MQ可以使用消息总线很方便的解决这个问题：

A调用B服务后，只需要监听B处理完成的信息，当B处理完成后，会发送一条消息给MQ，MQ会将此消息转发给A服务。这样，A服务既不用循环调用B的查询API，也不用提供CALL_BACK API。同样B服务也不需要这些操作。A服务还能及时得到异步处理成功的消息。

图： *img/mq3-ybcl.png*



MQ分类与对比

MQ类型	优点	缺点
ActiveMQ	单机吞吐量万级，时效性ms级，可用性高，基于主从框架实现高可用性，消息可靠性较低的概率丢失数据	官方社区现在对ActiveMQ 5.x的维护越来越少，高吞吐量场景较少使用
Kafka	性能卓越，单机写入TPS约在百万条/秒，最大的优点就是吞吐量高。时效性ms级可用性非常高。Kafka是分布式的，一个数据多个副本，少数机器宕机不会丢失数据，不会导致不可用。消费者采用Pull方式获取消息，消息有序，通过控制能够保证所有消息被消费且仅被消费一次，有优秀的第三方Kafka Web管理界面Kafka-Manager；在日志领域比较成熟，被多家公司和多个开源项目使用。功能支持：功能较为简单，主要支持简单的MQ功能，在大数据领域的实时计算以及日志采集被大规模使用	Kafka单机超过64个队列/分区，Load会发生明显的飙升现象，队列越多，load越高，发送消息响应时间变长，使用短轮询方式，实时性取决于轮询时间间隔，消费失败不支持重试；支持消息顺序，但是一台代理宕机后，就会产生消息乱序，社区更新较慢
RocketMQ	单机吞吐量十万级，可用性非常高，分布式架构，消息可以做到0丢失，MQ功能较为晚上，还是分布式的，扩展性好，支持10亿级别的消息堆积，不会因为堆积导致性能下降，源码是Java，因此便于Java开发者阅读源码，定制自己的MQ	支持的客户端语言不多，目前是Java、C++，其中C++不成熟；社区活跃度一般，没有在MQ核心中去实现JMS等接口，有些系统要迁移需要修改大量代码
RabbitMQ	由于erlang语言的高并发特征，性能较好；吞吐量打到万级，MQ功能比较完备，健壮、文档、易用、跨平台、支持多语言（如Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等），支持AJAX文档齐全；开源提供的管理界面简洁实用，社区剪跃度高；更新频率相当高	商业版需要收费，学习成本较高

tips1: Kafka被称为大数据的杀手锏，谈到大数据领域内的消息传输则绕不开Kafka，这款为大数据而生的消息中间件以其百万级TPS的吞吐量名声大噪，迅速成为大数据领域的宠儿，在数据采集、传输、存储的过程中发挥着举足轻重的作用。目前已经被LinkedIn、Uber、Twitter、Netflix等大公司采纳。

tips2: RocketMQ出自阿里巴巴的开源产品，用Java语言实现，在设计时参考了Kafka，并做出了自己的一些改进。被阿里巴巴广泛应用在订单、交易、充值、流计算、消息推送、日志流式处理、binglog分发等场景。

tips3: RabbitMQ在2007年发布，是一个在AMQP（高级消息队列协议）基础上完成的，可复用的企业消息系统，是当前最主流的消息中间件之一。

Rabbit MQ四大核心概念

生产者

产生数据发送信息的程序是生产者

交换机

交换机是Rabbit MQ非常重要的一个部件，一方面它接收来自生产者的消息，另一方面它将信息推送到队列中。交换机必须确切知道如何处理它接收到的消息，是将这些消息推送到特定队列还是推送到多个队列，亦或是把消息丢弃，这个得交换机决定

队列

队列是RabbitMQ内部使用的一种数据结构，尽管消息流经Rabbit MQ和应用程序，但它们只能存储在队列中。队列仅受主机的内存和磁盘限制的约束，本质上是一个打的消息缓冲区。许多生产者可以将消息发送到一个队列，许多消费者可以尝试从一个队列接收数据。这就是我们使用队列的方式


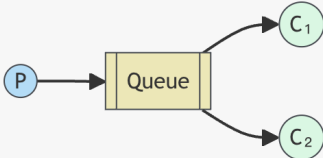
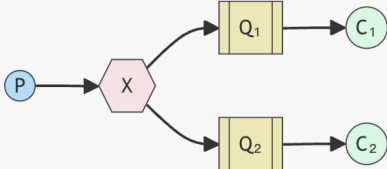
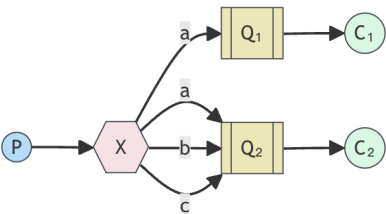
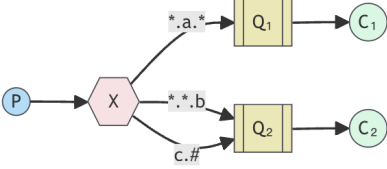
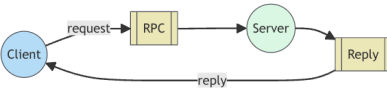
（一个消费者对应一个队列；但是一个生产者可以对应多个队列）

消费者

消费与接收举要相似的含义。消费者大多时候是一个等待接收消息的程序。请注意，生产者、消费者和消息中间件很多事实并不在同一机器上。同一个应用程序既可以是生产者，又可以是消费者

Rabbit MQ核心部分（六大模式）

图： [img/mq4-hxms.png](#)

<p>1. "Hello World!"</p> <p>The simplest thing that does <i>something</i></p> 	<p>2. Work Queues</p> <p>Distributing tasks among workers (the <i>competing consumers pattern</i>)</p> 	<p>3. Publish/Subscribe</p> <p>Sending messages to many consumers at once</p> 
<p>4. Routing</p> <p>Receiving messages selectively</p> 	<p>5. Topics</p> <p>Receiving messages based on a pattern (topics)</p> 	<p>6. RPC</p> <p><i>Request/reply pattern</i> example</p> 
<p>7. Publisher Confirms</p> <p>Reliable publishing with publisher confirms</p>		

简单模式（Hello World）

工作模式（Work queues）

发布-订阅模式（Publish/Subscribe）

路由模式（Routing）

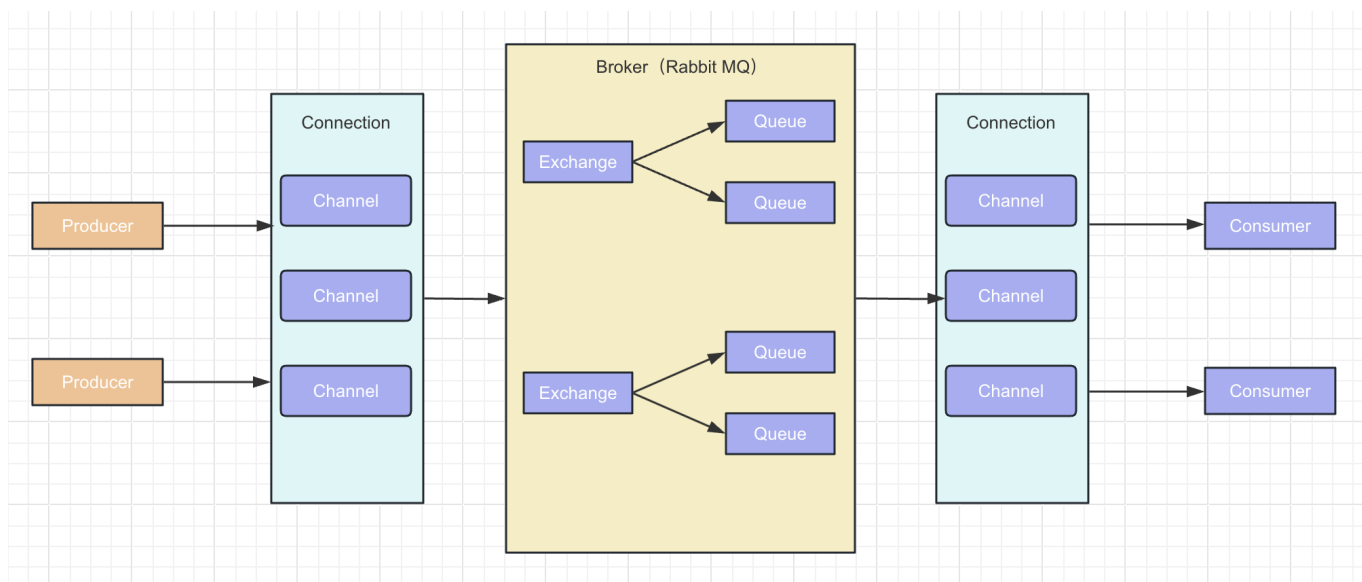
主题模式（Topics）

发布确认模式（Publisher Confirms）

~~RPC模式~~

RabbitMQ工作原理

图：img/mq5-gzyl.png



Broker: 接收和分发消息的应用，Rabbit MQ Server就是Message Broker

Virtual host: 出于多租户和安全因素设计的，把AMQP的基本组件划分到一个虚拟的分组中，类似于网络中的namespace概念。当多个不同的用户使用同一个Rabbit MQ Server提供的服务时，可以划分出多个vhost，每个用户在自己的vhost创建Exchange/Queue等（等同于SQL中的数据库database，在管理界面中的Admin-右侧可以看到）

Connection: publisher/consumer和Broker之间的TPC连接

Channel: 如果每一次访问Rabbit MQ都建立一个Connection，在消息量打的时候建立TPC Connection的开销将是巨大的，效率也低。Channel是在Connection内部建立的逻辑连接，如果应用程序支持多线程，通常每个thread创建单独的Channel进行通讯，AMQP method包含了Channel id帮助客户端和Message Broker识别Channel，所以Channel之间是完全隔离的。Channel作为轻量级的Connection极大减少了操作系统建立TPC Connection的开销

Exchange: Message到达Broker的第一站，根据分发规则，匹配查询表中的routing key，分发消息到Queue中去。常用的类型有：direct（point-to-point）、topic（Publish-subscribe）、fanout（multicast）

Queue: 消息最终被送到这里等待Consumer取走

Binding: Exchange和Queue之间的虚拟链接，binding中可以包含routing key，binding信息被保存到Exchange中的查询表中，用于Message的分发依据

Rabbit MQ核心部分

封装Channel部分代码

直接将信道封装，往下每次调用不需要再重新连接信道，直接调用方法即可

```
/**
 * 直接获取信道
 * @return 返回MQ设置好的的信道
 */
public static Channel getChannel() throws IOException, TimeoutException {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost(RabbitMQConfigDiction.MQ_HOST);
    factory.setVirtualHost(RabbitMQConfigDiction.VIRTUAL_HOST);
    factory.setUsername(RabbitMQConfigDiction.USER_NAME);
    factory.setPassword(RabbitMQConfigDiction.PASSWORD);
    return factory.newConnection().createChannel();
}
```

所需依赖

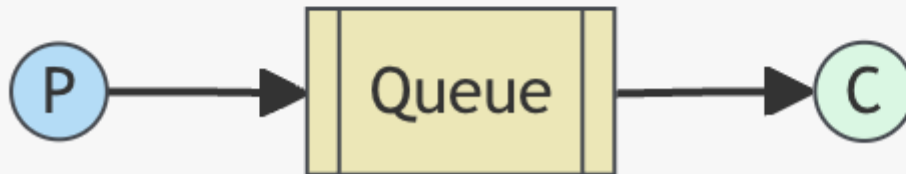
```
<dependencies>
  <!-- rabbitmq客户端依赖 -->
  <!-- https://mvnrepository.com/artifact/com.rabbitmq/amqp-client -->
  <dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.20.0</version>
  </dependency>
  <!-- 操作文件流 -->
  <!-- https://mvnrepository.com/artifact/commons-io/commons-io -->
  <dependency>
    <groupId>commons-io</groupId>
    <artifactId>commons-io</artifactId>
    <version>2.11.0</version>
  </dependency>
</dependencies>
```

Hello World

图: *img/mq6-hello.png*

1. "Hello World!"

The simplest thing that does *something*



代码实现

简单的生产者 (hello/Producer)

```

/**
 * HelloWorld-简单模式，发送方（生产者）
 */
private void testHelloWorldSent() {
    // 生产者创建连接-获取信道（Channel）
    RabbitMQTestUtils rabbitMQTestUtils = new RabbitMQTestUtils();
    try (Connection connection =
rabbitMQTestUtils.getConnectionFactory().newConnection();
        Channel channel = connection.createChannel()){
        /**
         * 生成一个队列
         * 1. 队列名称
         * 2. 队列里消息是否需要持久化（写入磁盘），默认存在内存中，即false
         * 3. 该队列是否只供一个消费者进行消费共享，true为可以多个消费者消费，false只能一个
消费者消费
         * 4. 是否自动删除，最后一个消费者断开连接后该队列是否执行自动删除-true自动删除，
false不自动删除
         * 5. 其他参数
         */
        channel.queueDeclare(QUEUE_NAME, false, false, false, null);
        String message = "Hello Rabbit MQ";
        /**
         * 进行消息发送
         * 1. 目标交换机
         * 2. 队列名称（routingKey）
         * 3. 其他参数信息
         * 4. 发送消息的消息体
         */
        channel.basicPublish("", QUEUE_NAME, null,
message.getBytes(StandardCharsets.UTF_8));
        System.out.println("[X] Sent '" + message + "'");
    } catch (IOException e) {
        e.printStackTrace();
        System.err.println(e.getMessage());
    } catch (TimeoutException e) {
        e.printStackTrace();
        System.err.println(e.getMessage());
    }
}
}

```

简单的消费者（hello/Consumer）

```

/**
 * HelloWorld-简单模式，接收方（消费者）
 * @throws IOException
 * @throws TimeoutException
 */
private void testHelloWorldReceived() throws IOException, TimeoutException {
    RabbitMQTestUtils rabbitMQTestUtils = new RabbitMQTestUtils();
    Connection connection =
rabbitMQTestUtils.getConnectionFactory().newConnection();
    Channel channel = connection.createChannel();
    channel.queueDeclare(QueueName, false, false, false, null);
    System.out.println("[*] Waiting for messages. To exit press CTRL+C");
    // 额外的 DeliverCallback 接口，用于缓冲服务器推送给我们的信息。
    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        String message = new String(delivery.getBody(), "UTF-8");
        System.out.println("[X] Received '" + message + "'");
    };
}
/**
 * 消费者消费信息
 * 1. 消费队列名称
 * 2. 消费成功后是否自动应答，true表示自动，false手动
 * 3. 消费者成功消费的回调
 * 4. 消费者取消消息的回调（消费被中断、消费失败）
 */
channel.basicConsume(QueueName, true, deliverCallback, consumerTag -> {
    System.out.println("Received is stop!");
});
}

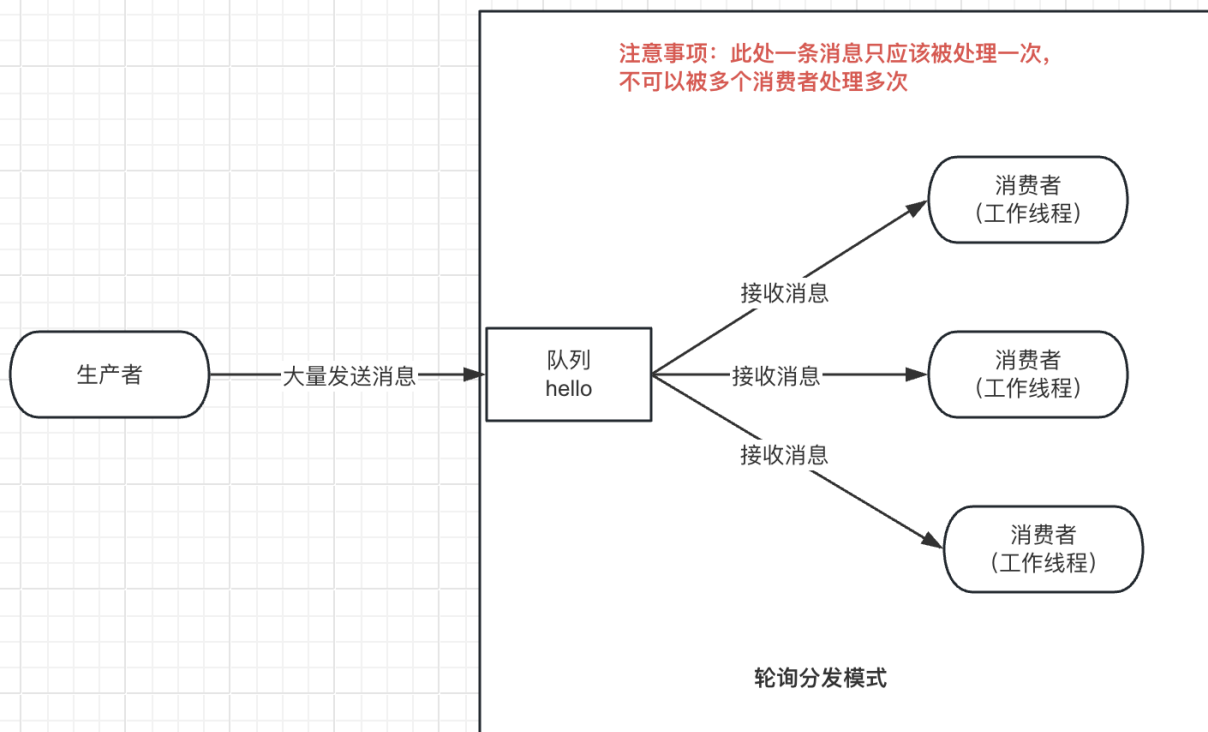
```

Work Queues

工作队列（任务队列），主要思想是避免立即执行资源密集型任务，而不得不等待它完成。工作队列（又名：任务队列）背后的主要理念是避免立即执行资源密集型任务并等待其完成。相反，我们将任务安排在稍后完成。我们将任务封装为消息，并将其发送到队列。在后台运行的 Worker 进程会弹出任务并最终执行作业。当运行多个 Worker 时，任务将在它们之间共享。

这一概念在网络应用程序中尤其有用，因为在短时间的 HTTP 请求窗口中不可能处理复杂的任务。安排任务在之后执行。

图：



消费者声明队列

- 消费者与生产者都应当使用 `channel.queueDeclare` 方法声明队列
- 消费者如果不声明队列，那么如果首次创建对象，生产者未声明队列前消费者就启动，会触发 `NOT_FOUND` 异常
- 有一个疑问点：优先创建了消费者，并且声明了 `QueueDeclare`，为什么此时的消费者无法消费后起的生产者生产的消息？

轮训模式

当一个生产者生产消息时，多个消费者同时消费，此时消费者会轮流消费消息。例如生产者生产了 `AA/BB/CC/DD` 四条消息，有两个消费者 `Worker01` 和 `Worker02` 此时01就会消费 `AA/CC`，02就会消费 `BB/DD` 如此轮流依次消费

代码实现

生产者Task (`workqueues/Task01`)

```

/**
 * 简单的工作队列发送方
 */
private void testWorkQueueSent() {
    try (Channel channel = RabbitMQTestUtils.getChannel()) {
        // 声明队列
        channel.queueDeclare(RabbitMQConfigDiction.TASK_QUEUE, false, false,
false, null);
        // 从控制台接收信息
        System.out.println("please input the message");
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) {
            String message = scanner.next();
            channel.basicPublish("", RabbitMQConfigDiction.TASK_QUEUE, null,
message.getBytes(StandardCharsets.UTF_8));
            System.out.println("be sent successfully! the message is : [" +
message + "]");
        }
    } catch (IOException e) {
        e.printStackTrace();
    } catch (TimeoutException e) {
        e.printStackTrace();
    }
}
}

```

用于轮训的消费者Worker（启动时启动多个即可，代码使用同一套）（workqueues/Worker01）

```

/**
 * 简单的工作队列模式接收方
 * @throws IOException
 * @throws TimeoutException
 */
private void testWorkQueuesReceived() throws IOException, TimeoutException {
    Channel channel = RabbitMQTestUtils.getChannel();
    // 声明队列-消费者如果不声明队列，那么如果首次创建对象，生产者未声明队列前消费者就启动，
    会触发NOT_FOUND异常
    channel.queueDeclare(RabbitMQConfigDiction.TASK_QUEUE, false, false, false,
    null);
    // 额外的 DeliverCallback 接口，接收消息成功时执行。
    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        String message = new String(delivery.getBody(), "UTF-8");
        System.out.println("[X] Received '" + message + "'");
    };
    // 消息接收取消后的接口
    CancelCallback cancelCallback = consumerTag ->
    System.out.println(consumerTag + "Received is cancel!");
    System.out.println("Work01 is waiting");
    // 消息接收
    channel.basicConsume(RabbitMQConfigDiction.TASK_QUEUE, true,
    deliverCallback, cancelCallback);
}

```

消息应答

消费者完成一个任务可能需要一段时间，如果其中一个消费者处理一个长的任务并仅完成了部分突然就挂掉，那么该消息就无法被接收。**Rabbit MQ**一旦向消费者传递了一条消息，便立即将该消息标记为删除。在这种情况下突然消费者挂掉，丢失了正在处理的消息以及后续轮训发送给该消费者的消息，因为消费者无法接收。

为了保证消息在发送过程中不丢失，**Rabbit MQ**引入了消息应答机制。即，消费者在接收到消息并处理该消息后，返回告知**Rabbit MQ**已经处理成功，**Rabbit MQ**此时可以删除消息。

自动应答

消息发送后立即被认为已经传送成功，这种模式需要在高吞吐量和数据传输安全性方面做权衡，因为这种模式如果消息在接收到之前，消费者就出现连接或者Channel关闭，那么消息就丢失了，当然另一方面这种模式消费者也可以传递过载的消息，没有对传递的消息数量进行限制，当然这样有可能使得消费者由于接收太多还来不及处理的消息导致消息积压，最终使得内存耗尽，这些消费者线程被系统杀死。

因此该模式仅适用于消费者可以高效并以某种速率能够处理这些消息情况下使用。

手动应答

手动应答的方法

```
Channel.basicAck()
```

确定消息，适用于消息肯定被处理后的放回，告知Rabbit MQ消息已经被接收并且处理，可以丢弃该消息

```
Channel.basicNack()
```

不确定消息，表示不处理该消息直接拒绝，提示Rabbit MQ可以将消息丢弃。

```
Channel.basicReject()
```

不确定消息，表示不处理该消息直接拒绝，提示Rabbit MQ可以将消息丢弃。

与上述的 `Channel.basicAck()/Channel.basicNack()` 相比少了一个参数 `Multiple`（批量处理）。

Multiple批量处理

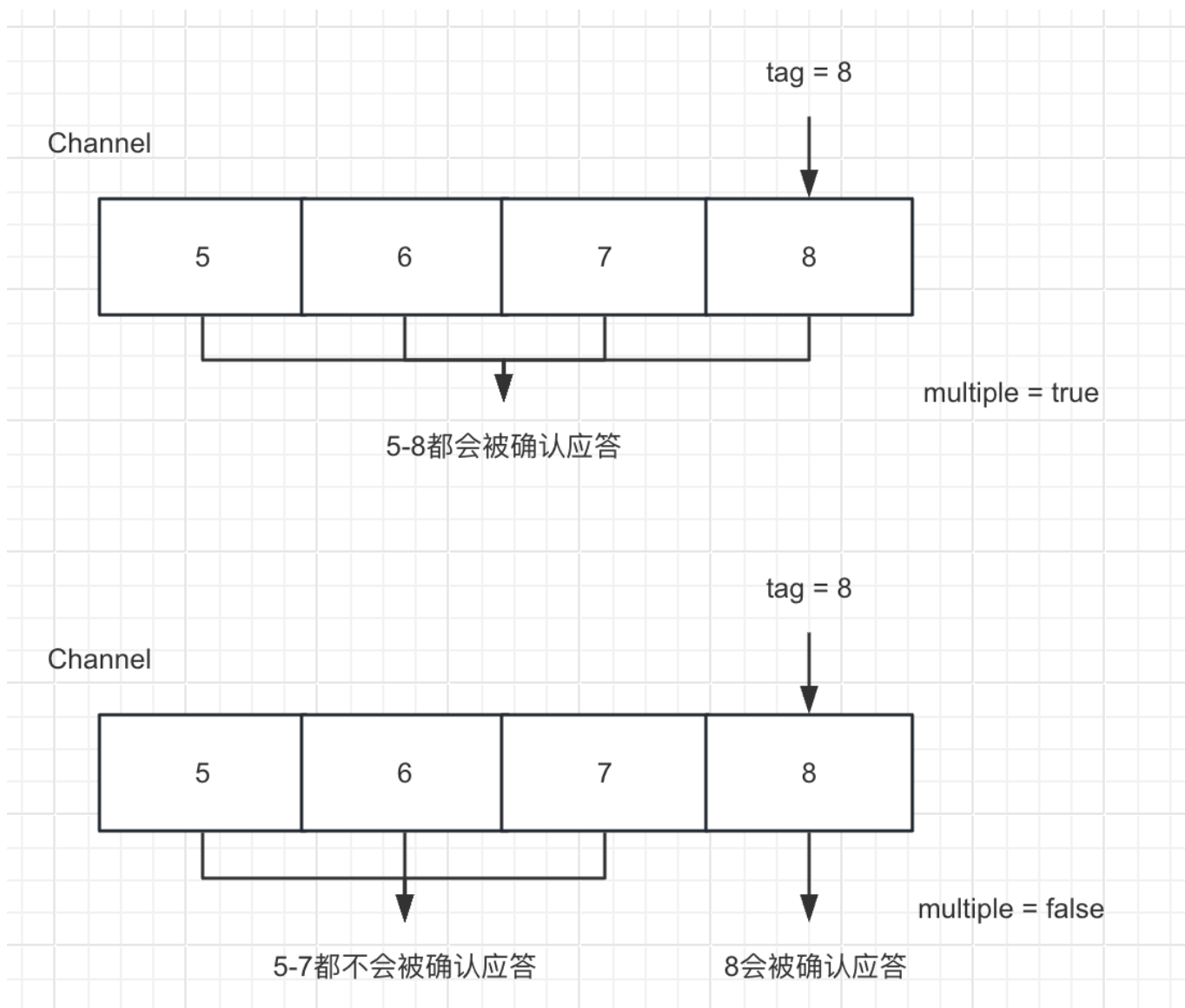
手动应答的好处就是可以批量应答并且减少网络拥堵（如下方例子，第二个参数“true”即为multiple参数）

```
channel.basicAck(deliceryTag, true);
```

multiple参数的 `true` 和 `false`

- `true`代表批量应答Channel上未应答的消息
 - 比如Channel上有传送tag的消息5、6、7、8，当前tag是8，那么此时5-8这些还未应答的消息都会被确收到消息应答
- `false`只会应答最后一条消息（日常开发建议使用）
 - 只会应答tag为8的消息（之应答当前），5、6、7这三个消息依然不会被确认收到消息应答

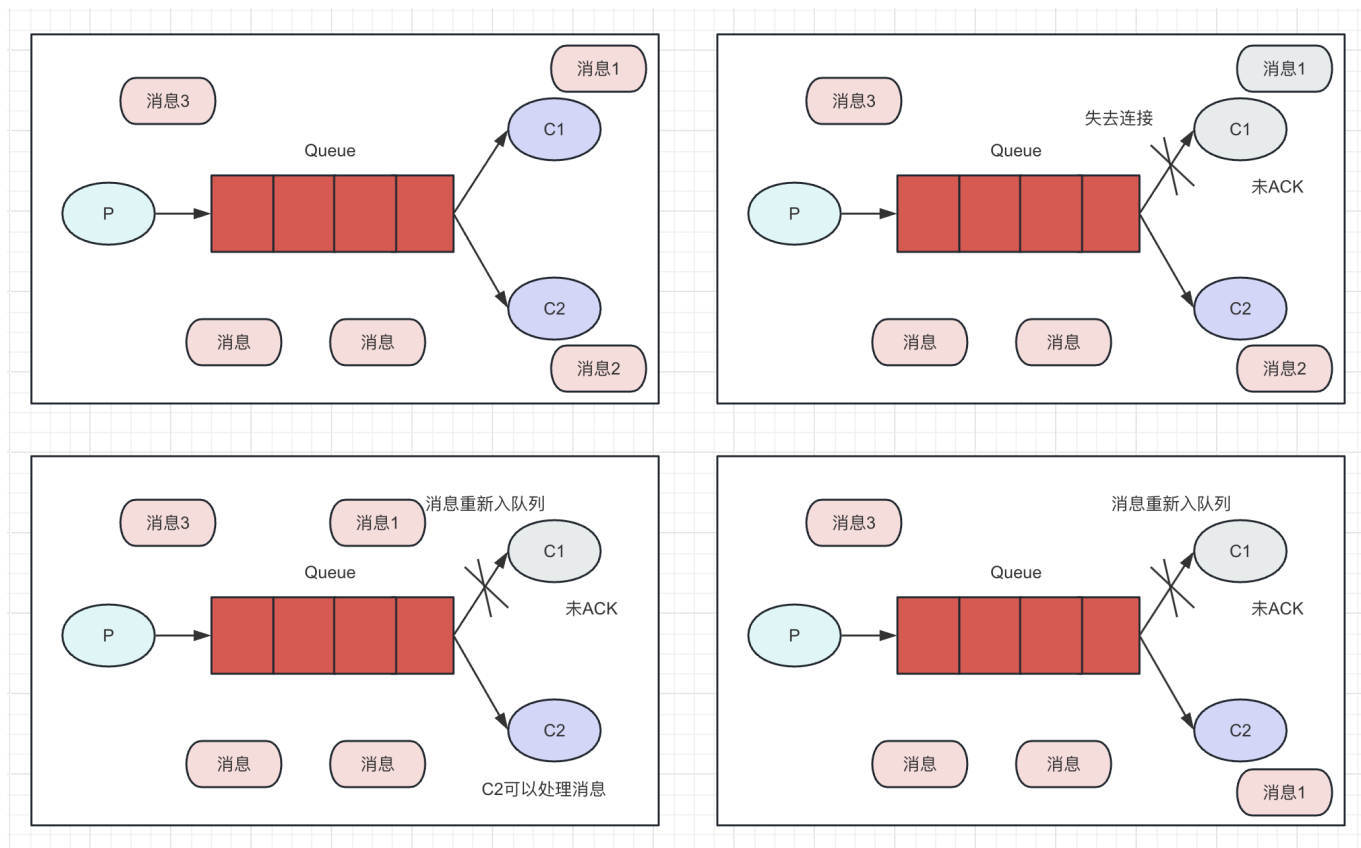
图： *img/mq8-ack.png*



消息自动重新入队

如果消费者由于某些原因失去连接（其通道已关闭，连接已关闭或TCP连接丢失），导致消息未发送ACK确认，Rabbit MQ将了解到消息未处理，并对其重新排队。如果此时其他消费者可以处理，它将很快将其重新分发给另一个消费者。这样即使某个消费者偶尔死亡，也可以确保消息不会丢失。

图：img/mq9-ack.png



由上图可见消息手动应答有两个特性：

- 消息在手动应答时是不丢失的
- 消息在未得到应答时，应当重新放回队列被重新消费

代码实现

生产者部分（workqueues/Task01）

```

/**
 * 消息手动应答发送方
 * 本次主要测试消费者，因此生产者代码与上一致，只是使用了不同的消息队列名称
 */
private void testWorkQueueAckSent() {
    try (Channel channel = RabbitMQTestUtils.getChannel()) {
        // 声明队列
        channel.queueDeclare(RabbitMQConfigDiction.TASK_ACK_QUEUE, false,
false, false, null);
        // 从控制台接收信息
        System.out.println("please input the message");
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) {
            String message = scanner.next();
            channel.basicPublish("", RabbitMQConfigDiction.TASK_ACK_QUEUE,
null, message.getBytes(StandardCharsets.UTF_8));
            System.out.println("be sent successfully! the message is : [" +
message + "]");
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        catch (TimeoutException e) {
            e.printStackTrace();
        }
    }
}

```

消费者部分 (workqueues/Worker01)

```

/**
 * 手动应答时不丢失，放回队列重新消费
 * @throws IOException
 * @throws TimeoutException
 */
private void testWorkQueuesAckReceived01() throws IOException, TimeoutException
{
    final Channel channel = RabbitMQTestUtils.getChannel();
    channel.queueDeclare(RabbitMQConfigDiction.TASK_ACK_QUEUE, false, false,
false, null);
    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        // 模拟事务处理，本例为较快，因此沉睡1秒
        RabbitMQTestUtils.getSleep(1);
        String message = new String(delivery.getBody(), "UTF-8");
        System.out.println("[X] Received '" + message + "'");
        /**
         * 手动应答
         * 1. 消息的标记，long类型参数 (tag)
         * 2. 是否批量应答 (multiple)
         */
        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
    };
    // 消息接收取消后的接口
    CancelCallback cancelCallback = consumerTag ->
System.out.println(consumerTag + "Received is cancel!");
    System.out.println("Work01 is waiting...fast");
    // 采用手动应答
    boolean autoAck = false;
    channel.basicConsume(RabbitMQConfigDiction.TASK_ACK_QUEUE, autoAck,
deliverCallback, cancelCallback);
}

```

消费者部分 (workqueues/Worker02)

上述代码中 `RabbitMQTestUtils.getSleep(30);` 模拟工作时间改为30秒

- 经过测试，两台接受者服务器均正常的情况下，消息遵循轮训原则被消费，只是Worker02处理较慢。
- 但是在Worker02漫长的等待过程中，消息未被消费到就切断服务器程序，此时Worker01则会消费丢失的信息。
- 如果再次启动Worker02服务器，会再次遵循轮训原则，并且Worker02不会去重复消费之前的丢失消息。

消息的持久化

队列持久化

我们已经学习了消息应答来保证即使消费者死亡，消息也不会丢失。但是，如果 RabbitMQ 服务器停止，我们的任务仍然会丢失。

当 RabbitMQ 退出或崩溃时，它会忘记队列和消息，除非您告诉它不要这样做。要确保消息不会丢失，需要做两件事：我们需要将队列和消息都标记为持久化。

队列持久化实现

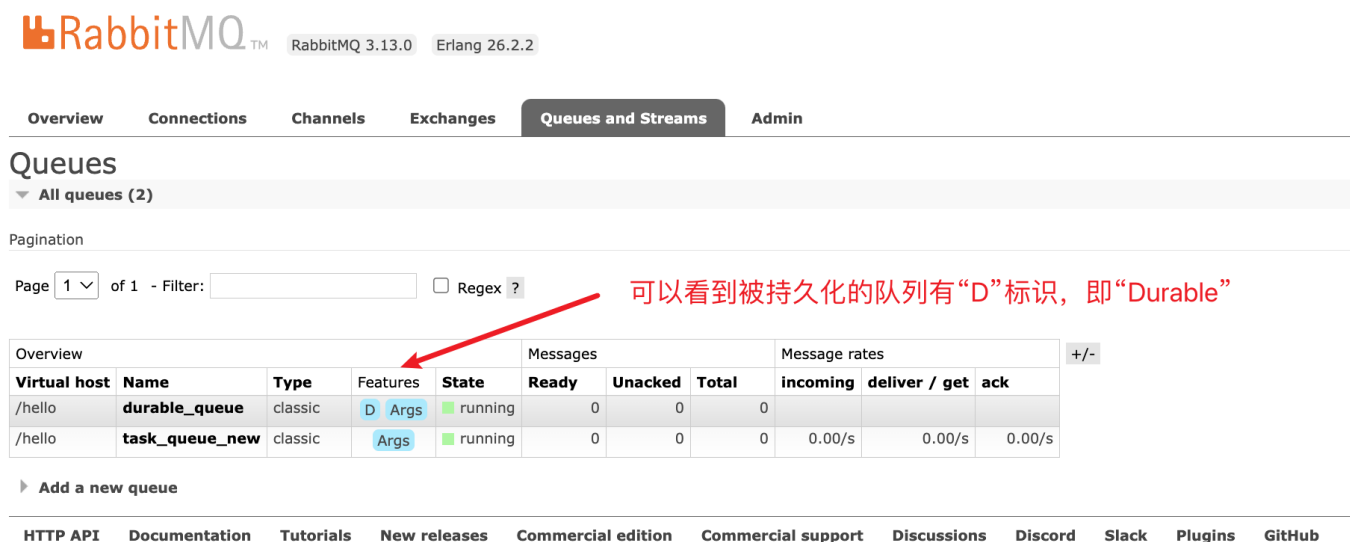
之前我们创建的队列都是非持久化的，Rabbit MQ一旦重启就会被删除。如果需要进行队列持久化，我们需要把参数 `durable` 设为true。当做了该项设置后，及时Rabbit MQ重启，队列也将存在，而不会被删除。

官网说明 *This `queueDeclare` change needs to be applied to both the producer and consumer code.* ——对 `queueDeclare` 的更改需要同时应用于生产者和消费者代码。

经过测试，下述代码段如果消费者没有，则不会消费消息。

```
boolean durable = true;
channel.queueDeclare("queueName", durable, false, false, null);
```

图： `img/mq10-Durable.png`



The screenshot shows the RabbitMQ Admin interface. The 'Queues and Streams' tab is selected. Under 'All queues (2)', there are two queues listed: 'durable_queue' and 'task_queue_new'. The 'durable_queue' has a 'D' icon in the 'Features' column, which is highlighted by a red arrow. A red text annotation says '可以看到被持久化的队列有“D”标识，即“Durable”' (You can see that queues with the 'D' identifier are durable, i.e., 'Durable').

Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/hello	durable_queue	classic	D Args	running	0	0	0			
/hello	task_queue_new	classic	Args	running	0	0	0	0.00/s	0.00/s	0.00/s

Tips: 当一个队列名称在对应的Virtual host已经存在，并且为非持久化时，创建同名的持久化队列会报创建错误。

消息持久化

如果仅仅按照上述进行队列持久化，那么，重启后队列会存在，但是未被消费的消息将会丢失，因此需要进行消息持久化。

消息持久化实现

队列的持久化应该是有生产者生产出消息时，就应当注明消息是否持久化，即

`channel.basicPublish()` 中的第三个参数，规定为 `PERSISTENT_TEXT_PLAIN` 持久性文本。

该方法不能完全保证不会丢失消息；会尽量要求Rabbit MQ将消息保存到磁盘，因为在Rabbit MQ将消息准备存储到设备时，未存储完毕这个时间节点未写入，则会导致持久化失败。这种情况则需要使用**发布确认**这种更加强有力的持久化策略。

```
import com.rabbitmq.client.MessageProperties;

channel.basicPublish("", "queueName",
    MessageProperties.PERSISTENT_TEXT_PLAIN,
    message.getBytes());
```

代码实例

生产者部分（workqueues/Task01）

```

/**
 * 队列持久化
 * 消息持久化
 */
private void testWorkQueueDurable() {
    try (Channel channel = RabbitMQTestUtils.getChannel();) {
        // 持久化队列Queue的参数-durable
        boolean durable = true;
        channel.queueDeclare(RabbitMQConfigDiction.TASK_DURABLE_QUEUE, durable,
false, false, null);
        // 从控制台接收信息
        System.out.println("please input the message");
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) {
            String message = scanner.next();
            channel.basicPublish("", RabbitMQConfigDiction.TASK_DURABLE_QUEUE,
                MessageProperties.PERSISTENT_TEXT_PLAIN, // 消息持久化
                message.getBytes(StandardCharsets.UTF_8));
            System.out.println("be sent successfully! the message is : [" +
message + "]);
        }
    } catch (IOException | TimeoutException e) {
        e.printStackTrace();
    }
}

```

消费者部分 (workqueues/Worker01)

```

/**
 * 持久化测试接收方
 * @throws IOException
 * @throws TimeoutException
 */
private void testWorkQueuesDurableReceived() throws IOException,
TimeoutException {
    Channel channel = RabbitMQTestUtils.getChannel();
    // 同样声明持久化队列
    channel.queueDeclare(RabbitMQConfigDiction.TASK_DURABLE_QUEUE, true, false,
false, null);
    // 额外的 DeliverCallback 接口，接收消息成功时执行。
    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        String message = new String(delivery.getBody(), "UTF-8");
        System.out.println("[X] Received '" + message + "'");
    };
    // 消息接收取消后的接口
    CancelCallback cancelCallback = consumerTag ->
System.out.println(consumerTag + "Received is cancel!");
    System.out.println("Work01 is waiting");
    // 消息接收
    channel.basicConsume(RabbitMQConfigDiction.TASK_DURABLE_QUEUE, true,
deliverCallback, cancelCallback);
}

```

公平调度

其实是非轮训分发，能者多劳。

调度工作仍然不能完全按照我们的要求进行。例如，在有两个 Worker 的情况下，当所有奇数消息都很重，而偶数消息都很轻时，一个 Worker 将一直处于忙碌状态，而另一个 Worker 几乎不做任何工作。但是，RabbitMQ 对此一无所知，它仍然会均匀地分派消息。

出现这种情况是因为 RabbitMQ 只会在消息进入队列时分派消息。它不会查看消费者未确认消息的数量。它只是盲目地将每 n 条消息分派给第 n 个消费者。

为了解决这个问题，我们可以在消费者使用带有 `prefetchCount = 1` 设置的 `basicQos` 方法。这样，RabbitMQ 就不会一次向 Worker 发送多于一条消息。或者换句话说，在处理并确认前一条消息之前，不要向 Worker 发送新消息。相反，它会将消息分派给下一个不忙的 Worker。

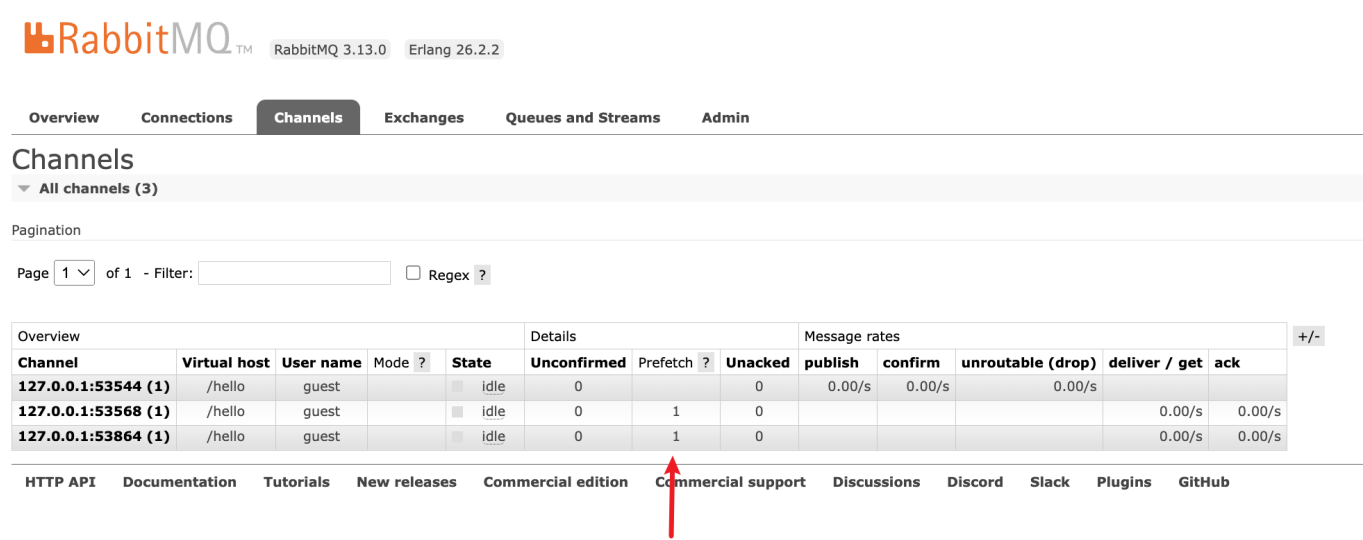
消费者模块代码

```

int prefetchCount = 1; // 轮训分发默认为0
channel.basicQos(prefetchCount);

```

图：img/mq11-FairDispatch.png



公平调度basicQos(1)中的参数1

代码实例

消费者部分（workqueues/Worker01）


```

/**
 * 公平调度
 * @throws IOException
 * @throws TimeoutException
 */
private void testWorkQueuesFairDispatchReceived01() throws IOException,
TimeoutException {
    final Channel channel = RabbitMQTestUtils.getChannel();
    channel.queueDeclare(RabbitMQConfigDiction.TASK_DURABLE_QUEUE, true, false,
false, null);
    // 公平调度
    channel.basicQos(1);
    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        // 模拟事务处理，本例为较慢，因此沉睡1秒
        RabbitMQTestUtils.getSleep(1);
        String message = new String(delivery.getBody(),
StandardCharsets.UTF_8);
        System.out.println("[X] Received '" + message + "'");
        /**
         * 手动应答
         * 1.消息的标记，long类型参数 (tag)
         * 2.是否批量应答 (multiple)
         */
        channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
    };
    // 消息接收取消后的接口
    CancelCallback cancelCallback = consumerTag ->
System.out.println(consumerTag + "Received is cancel!");
    System.out.println("Work01 is waiting...fast");
    // 采用手动应答
    boolean autoAck = false;
    channel.basicConsume(RabbitMQConfigDiction.TASK_DURABLE_QUEUE, autoAck,
deliverCallback, cancelCallback);
}

```

消费者部分 (workqueues/Worker02)

上述代码中 `RabbitMQTestUtils.getSleep(2);` 模拟工作时间改为2秒

其实公平调度实现原理就是让每个消费者的信道当中消息量为1

预期值调度

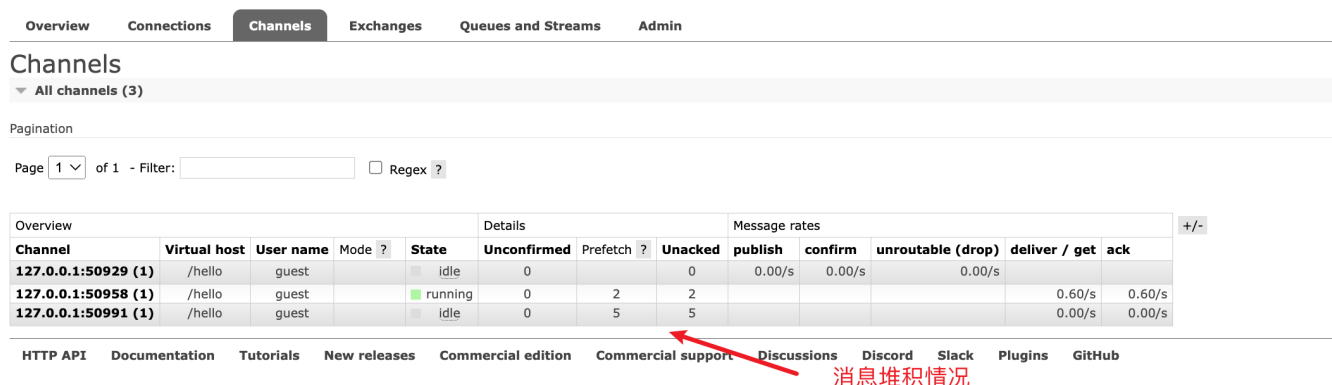
根据上一章公平调度的 `prefetchCount` 参数属性，默认为0，公平调度为1。但是当我们给到设定指定值时，则该值为欲取值。

欲取值与公平调度不同，不会在乎你的机器工作效率，而是会十分暴力的根据预设值来进行分配，例如Worker01分配2条消息要消费、Worker02要分配5条消息要消费，那么前消息会不在乎机器性能直接强行按照这个分配额进行分配。

也可以将这个值当做信道Channel可以存储等待的最大值。

此时，消息只能看信道中消息存储量，如果未到达预设值，即使消息未被消费，也会被分配消息。

图： *img/mq12-basicQos.png*



Channels											
All channels (3)											
Pagination											
Page 1 of 1 - Filter: <input type="text"/> <input type="checkbox"/> Regex ?											
Overview				Details				Message rates			
Channel	Virtual host	User name	Mode ?	State	Unconfirmed	Prefetch ?	Unacked	publish	confirm	unroutable (drop)	deliver / get ack
127.0.0.1:50929 (1)	/hello	guest		idle	0		0	0.00/s	0.00/s	0.00/s	
127.0.0.1:50958 (1)	/hello	guest		running	0	2	2				0.60/s 0.60/s
127.0.0.1:50991 (1)	/hello	guest		idle	0	5	5				0.00/s 0.00/s

HTTP API Documentation Tutorials New releases Commercial edition Commercial support Discussions Discord Slack Plugins GitHub

消息堆积情况

发布确认

上面讲到，在写入磁盘的间隙，假如MQ服务器宕机，将可能会导致持久化失败，因此就需要一个发布确认机制，发布确认需要达到3个条件才能保证消息持久化：

- 要求队列必须持久化
- 要求队列中的消息必须持久化
- 发布确认（即生产者发布消息后，Rabbit MQ返回一个确认信息）

开启发布确认单方法

发布确认默认不开启，如果需要开启则需要在Channel信道上调用confirmSelect方法

```
channel.confirmSelect();
```

单个发布确认

这是一种简单的确认方式，它是一种同步确认发布的方式，也就是发布一个消息之后只有它被确认发布，后续的消息才能继续发布。 `waitForConfirms()` 这个方法只有在消息被确认时才返

回，如果在指定时间范围内这个消息没有被确认，那么将抛出异常。

此方法最大缺点是**发布速度极慢**，因为如果没有确认发布的消息就会阻塞后续所有消息，这种方式最多提供每秒不超过数百条发布消息的吞吐量。

代码实现

所在目录（confirmselect/ConfirmSelectTask）

```

/**
 * 单个发布确认
 * Total execution time is [548ms]
 * 队列持久化 - channel.queueDeclare 中的 durable = true
 * 消息持久化 - channel.basicPublish 中的 MessageProperties.PERSISTENT_TEXT_PLAIN
参数
 * 发布确认 - channel.confirmSelect()
 * 单个消息发布确认 - channel.waitForConfirms()
 */
private void testWorkQueueConfirmSelectIndividuallySent() {
    try (Channel channel = RabbitMQTestUtils.getChannel()) {
        // 持久化队列Queue的参数-durable
        boolean durable = true;
        // 随机获取队列名称
        final String queueName = UUID.randomUUID().toString();
        channel.queueDeclare(queueName, durable, false, false, null);
        // 开启发布确认
        channel.confirmSelect();
        // 开始时间
        final long beginTime = System.currentTimeMillis();
        // 批量发消息，用于测试耗时
        for (int i = 0; i < RabbitMQConfigDicttion.MESSAGE_COUNT; i++) {
            String message = String.valueOf(i);
            channel.basicPublish("", queueName,
MessageProperties.PERSISTENT_TEXT_PLAIN,
message.getBytes(StandardCharsets.UTF_8));
            // 单个消息发布确认
            if (channel.waitForConfirms()) {
                System.out.println("message is confirm, success, the message is
[" + message + "]);
            }
        }
        final long endTime = System.currentTimeMillis();
        System.out.println("Total execution time is [" + (endTime - beginTime)
+ "ms"]);
    } catch (IOException | TimeoutException | InterruptedException e) {
        e.printStackTrace();
    }
}

```

批量发布确认

单个发布确认执行极慢，如果使用批量发布确认可以极大地提高吞吐量，但是缺点是，当故障导致发布出现问题时，不知道哪个消息会出现问题，需要将整个批次保存在内存中，以记录重要的信息而后重新发布消息。

而且这种形式也是同步确认，也会阻塞消息发布。

代码实现

所在目录 (confirmselect/ConfirmSelectTask)

```

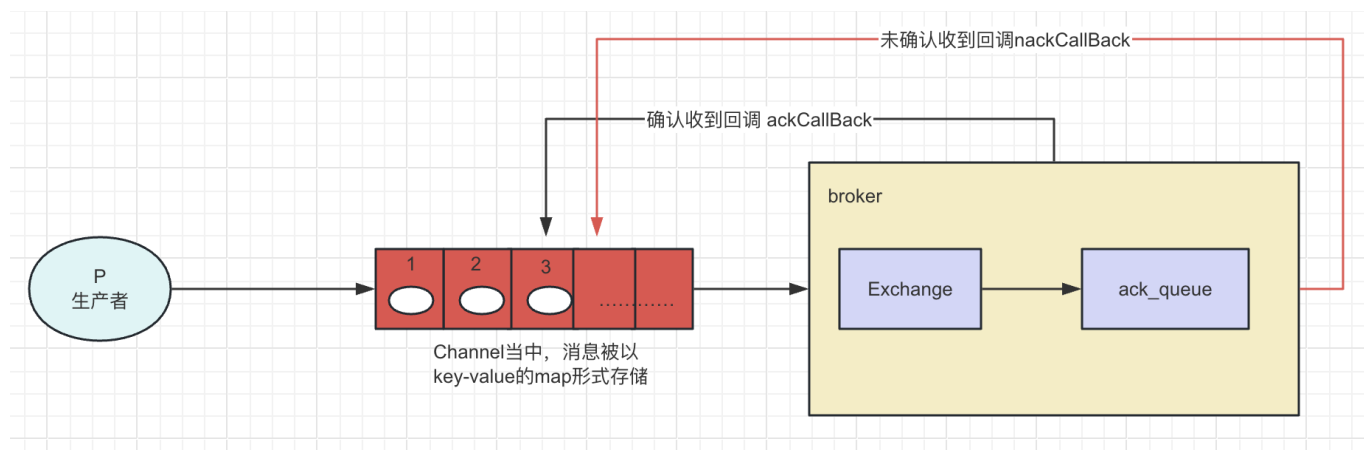
/**
 * 批量发布确认
 * Total execution time is [66ms]
 * 队列持久化 - channel.queueDeclare 中的 durable = true
 * 消息持久化 - channel.basicPublish 中的 MessageProperties.PERSISTENT_TEXT_PLAIN
参数
 * 发布确认 - channel.confirmSelect()
 * 批量消息发布确认 - 定义批量确认长度: final int confirmCount = 100;/
channel.waitForConfirms()
 */
private void testWorkQueueConfirmSelectBatchSent() {
    try (Channel channel = RabbitMQTestUtils.getChannel()) {
        // 持久化队列Queue的参数-durable
        boolean durable = true;
        // 随机获取队列名称
        final String queueName = UUID.randomUUID().toString();
        channel.queueDeclare(queueName, durable, false, false, null);
        // 开启发布确认
        channel.confirmSelect();
        // 定义批量确认长度
        final int confirmCount = 100;
        // 开始时间
        final long beginTime = System.currentTimeMillis();
        // 批量发消息, 用于测试耗时
        for (int i = 0; i < RabbitMQConfigDiction.MESSAGE_COUNT; i++) {
            String message = String.valueOf(i);
            channel.basicPublish("", queueName,
MessageProperties.PERSISTENT_TEXT_PLAIN,
message.getBytes(StandardCharsets.UTF_8));
            // 达到100条批量确认一次
            if ((i + 1) % confirmCount == 0) {
                if (channel.waitForConfirms()) {
                    System.out.println("message is confirm, success, now count
is [" + message + "]");
                }
            }
        }
        final long endTime = System.currentTimeMillis();
        System.out.println("Total execution time is [" + (endTime - beginTime)
+ "ms]");
    } catch (IOException | TimeoutException | InterruptedException e) {
        e.printStackTrace();
    }
}

```

异步发布确认

异步发布确认虽然逻辑上比上述两种都复杂，但是性价比最高。可靠性强、异步执行不阻塞效率高。它是利用回调函数来达到消息可靠性传递的，这个中间件也是通过函数回调来保证是否投递成功。

图：img/mq13-Confirm.png



创建一个 `channel.addConfirmListener` 为监听器，监听回调函数。该方法有两个构造函数：

- 一个为单参数的 `channel.addConfirmListener(ConfirmListener)` 静态返回
- 一个为双参数的 `channel.addConfirmListener(ConfirmCallback, ConfirmCallback)` 返回值为 `ConfirmListener`，一个监听器。该方法第一个参数为成功回调，即 `ackCallBack`；第二个为失败回调，即 `nackCallBack`

该监听为异步的，并且应当在消息发送前提前创建

创建监听器

```
// 成功回调函数（消息标识，是否为批量确认）
ConfirmCallback ackCallBack = (deliveryTag, multiple) -> {
    System.out.println("the success message's key is [" + deliveryTag + "]");
};
// 失败回调函数（消息标识，是否为批量确认）
ConfirmCallback nackCallBack = (deliveryTag, multiple) -> {
    System.out.println("the loss message's key is [" + deliveryTag + "]");
};
channel.addConfirmListener(ackCallBack, nackCallBack);
```

代码实例

所在目录（confirmselect/ConfirmSelectTask）

```

/**
 * 异步发布确认
 * Total execution time is [25ms]
 * 队列持久化 - channel.queueDeclare 中的 durable = true
 * 消息持久化 - channel.basicPublish 中的 MessageProperties.PERSISTENT_TEXT_PLAIN
参数
 * 发布确认 - channel.confirmSelect()
 * 监听器 - channel.addConfirmListener(ackCallBack, nackCallBack);
 *      - 成功回调函数: ackCallBack
 *      - 失败回调函数: nackCallBack
 */
private void testWorkQueueConfirmAsyncSent() {
    try (Channel channel = RabbitMQTestUtils.getChannel()) {
        // 持久化队列Queue的参数-durable
        boolean durable = true;
        // 随机获取队列名称
        final String queueName = UUID.randomUUID().toString();
        channel.queueDeclare(queueName, durable, false, false, null);
        // 开启发布确认
        channel.confirmSelect();
        // 开始时间
        final long beginTime = System.currentTimeMillis();
        // 消息监听器, 监听消息发送情况; 要在发送消息前优先准备, 否则可能监听不到broker返回
        // 成功回调函数 (消息标识, 是否为批量确认)
        ConfirmCallback ackCallBack = (deliveryTag, multiple) -> {
            System.out.println("the success message's key is [" + deliveryTag +
            "]);
        };
        // 失败回调函数 (消息标识, 是否为批量确认)
        ConfirmCallback nackCallBack = (deliveryTag, multiple) -> {
            System.out.println("the loss message's key is [" + deliveryTag +
            "]);
        };
        channel.addConfirmListener(ackCallBack, nackCallBack);
        // 批量发消息, 用于测试耗时
        for (int i = 0; i < RabbitMQConfigDiction.MESSAGE_COUNT; i++) {
            String message = String.valueOf(i);
            channel.basicPublish("", queueName, null,
message.getBytes(StandardCharsets.UTF_8));
        }
        final long endTime = System.currentTimeMillis();
        System.out.println("Total execution time is [" + (endTime - beginTime)
+ "ms]");
    } catch (IOException | TimeoutException e) {
        e.printStackTrace();
    }
}

```



```
}  
}
```

如何处理异步未确认消息

由于监听器是异步的，可能导致未完成监听但是消息已经发送完毕。因此会导致未确认的消息未被处理的情况。最好的解决方案就是把未确认的消息放到一个基于内存的，能被发布线程访问的队列，比如说ConcurrentLinkedQueue这个队列在confirm callbacks与发布线程之间进行消息的传递。

解决方案

- 记录下 `channel.basicPublish` 执行之后发布消息的总和
- 删除已经被监听器确认过的数据，其余就是未确认数据

代码逻辑

在开启发布确认后，先准备出一个线程安全有序的hash表，适用于高并发的情况

```
// 开启发布确认  
channel.confirmSelect();  
/**  
 * 在开启发布确认后，先准备出一个线程安全有序的hash表，适用于高并发的情况  
 * 1. 可以轻松地让消息与消息关联  
 * 2. 轻松批量删除条目，只需要key值  
 * 3. 支持高并发（多线程）  
 */  
ConcurrentSkipListMap<Long, Object> outstandingConfirms = new  
ConcurrentSkipListMap<>();
```

记录下所有消息的总和

`channel.getNextPublishSeqNo()` 方法获取信道中发送信息的序号

```
channel.basicPublish("", queueName, null,  
message.getBytes(StandardCharsets.UTF_8));  
// 记录下所有消息的总和  
outstandingConfirms.put(channel.getNextPublishSeqNo(), message);
```

在回调函数中删除已经处理的map中的数据

```

// 成功回调函数（消息标识，是否为批量确认）
ConfirmCallback ackCallBack = (deliveryTag, multiple) -> {
    // 如果是批量确认，那么则获取到批量信息后批量删除
    if (multiple) {
        ConcurrentNavigableMap<Long, Object> confirmed =
outstandingConfirms.headMap(deliveryTag);
        confirmed.clear();
    } else {
        // 单个确认时调用单个删除
        outstandingConfirms.remove(deliveryTag);
    }
    System.out.println("the success message's key is [" + deliveryTag + "]");
};
// 失败回调函数（消息标识，是否为批量确认）
ConfirmCallback nackCallBack = (deliveryTag, multiple) -> {
    String nackMessage = (String) outstandingConfirms.get(deliveryTag);
    System.out.println("the loss message's key is [" + deliveryTag + "]");
    System.out.println("the loss message is [" + nackMessage + "]");
};

```

代码实现

所在目录（confirmselect/ConfirmSelectTask）

```

/**
 * 异步发布确认
 * Total execution time is [25ms]
 * 队列持久化 - channel.queueDeclare 中的 durable = true
 * 消息持久化 - channel.basicPublish 中的 MessageProperties.PERSISTENT_TEXT_PLAIN
参数
 * 发布确认 - channel.confirmSelect()
 * 监听器 - channel.addConfirmListener(ackCallBack, nackCallBack);
 *      - 成功回调函数: ackCallBack
 *      - 失败回调函数: nackCallBack
 */
private void testWorkQueueConfirmAsyncSent() {
    try (Channel channel = RabbitMQTestUtils.getChannel()) {
        // 持久化队列Queue的参数-durable
        boolean durable = true;
        // 随机获取队列名称
        final String queueName = UUID.randomUUID().toString();
        channel.queueDeclare(queueName, durable, false, false, null);
        // 开启发布确认
        channel.confirmSelect();
        /**
         * 在开启发布确认后, 先准备出一个线程安全有序的hash表, 适用于高并发的情况
         * 1. 可以轻松地让消息与消息关联
         * 2. 轻松批量删除条目, 只需要key值
         * 3. 支持高并发 (多线程)
         */
        ConcurrentSkipListMap<Long, Object> outstandingConfirms = new
ConcurrentSkipListMap<>();
        // 消息监听器, 监听消息发送情况; 要在发送消息前优先准备, 否则可能监听不到broker返回
的信息
        // 成功回调函数 (消息标识, 是否为批量确认)
        ConfirmCallback ackCallBack = (deliveryTag, multiple) -> {
            // 如果是批量确认, 那么则获取到批量信息后批量删除
            if (multiple) {
                ConcurrentNavigableMap<Long, Object> confirmed =
outstandingConfirms.headMap(deliveryTag);
                confirmed.clear();
            } else {
                // 单个确认时调用单个删除
                outstandingConfirms.remove(deliveryTag);
            }
            System.out.println("the success message's key is [" + deliveryTag +
"]");
        };
        // 失败回调函数 (消息标识, 是否为批量确认)
        ConfirmCallback nackCallBack = (deliveryTag, multiple) -> {
            String nackMessage = (String) outstandingConfirms.get(deliveryTag);

```

```

        System.out.println("the loss message's key is [" + deliveryTag +
    "]"");
        System.out.println("the loss message is [" + nackMessage + "]"");
    };
    channel.addConfirmListener(ackCallBack, nackCallBack);
    // 开始时间
    final long beginTime = System.currentTimeMillis();
    // 批量发消息，用于测试耗时
    for (int i = 0; i < RabbitMQConfigDiction.MESSAGE_COUNT; i++) {
        String message = String.valueOf(i);
        channel.basicPublish("", queueName, null,
message.getBytes(StandardCharsets.UTF_8));
        // 1.记录下所有消息的总和
        outstandingConfirms.put(channel.getNextPublishSeqNo(), message);
    }
    final long endTime = System.currentTimeMillis();
    System.out.println("Total execution time is [" + (endTime - beginTime)
+ "ms]");
    } catch (IOException | TimeoutException e) {
        e.printStackTrace();
    }
}

```

发布确认形式对比

发布确认类型	等待确认	难易程度	吞吐量	错误判断
单个发布确认	同步	较为简单	有限	容易判断
批量发布确认	同步	及其简单	高	很难判断
异步发布确认	异步	代码实现难	极高	容易判断

- 单个发布确认：同步等待确认，简单，吞吐量非常有限
- 批量发布确认：批量同步等待确认，简单，合理吞吐量，一旦出现问题很难判断出是哪条消息出了问题
- 异步发布确认：最佳性能和资源使用，在出现错误的情况下可以很好的控制，但是实现较难

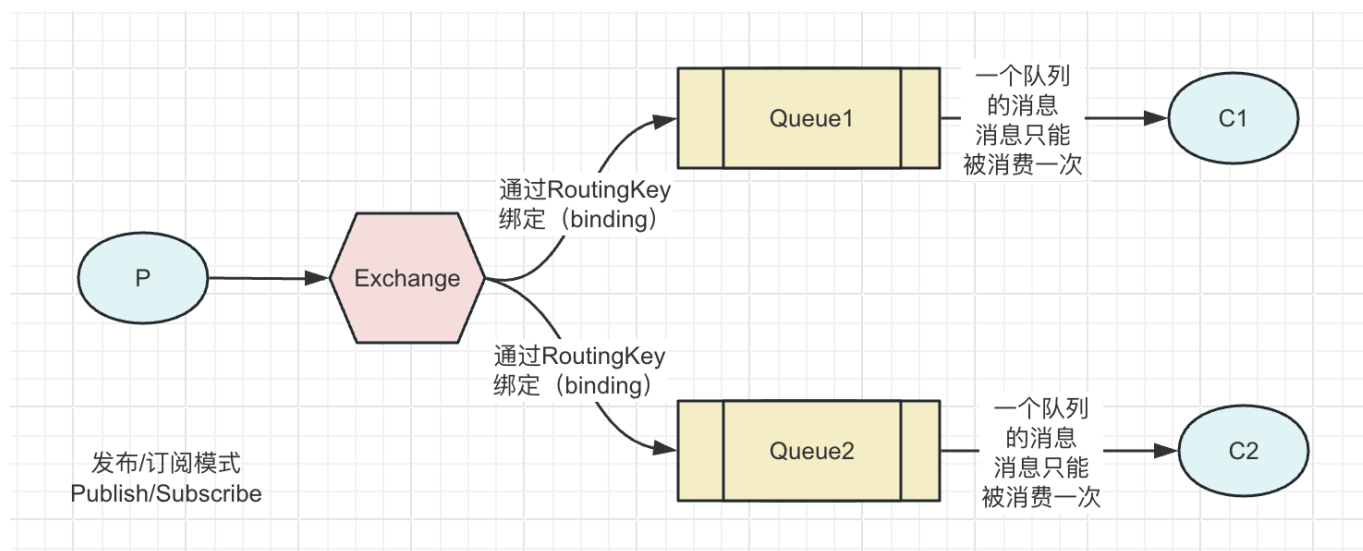
交换机

生产者发送消息的用户应用程序、队列是存储消息的缓冲区、消费者是接收消息的用户应用程序。而有了交换机我们可以将消息发送给多个消费者。

有几种可用的交换类型：直接交换（`direct`）、主题交换（`topic`）、标题交换（`headers`）和扇出交换（`fanout`）。

假设工作队列背后，每个任务都恰好交付给一个消费者（工作进程）。在这一部分中，将消息传递给多个消费者，这种模式称为“发布/订阅（`Publish/Subscribe`）”模式。

图：img/mq14-Publish/Subscribe.png



tips: 在前面没有交换机介入时，一条消息只能被消费一次，因为一个队列只能对应一个消费者

Exchanges

Exchanges概念

RabbitMQ 消息传递模型的核心思想是，生产者从不直接向队列发送任何消息（简单模式走的是默认交换机）。实际上，很多时候生产者甚至不知道消息是否会被传送到任何队列。

相反，生产者只能将消息发送到交换中心。交换机的工作内容非常简单。它的一端接收来自生产者的信息，另一端将信息推送到队列。交易所必须清楚地知道如何处理收到的信息。它应该被附加到某个队列吗？是否应将其添加到多个队列中？还是丢弃？这些规则由交换类型定义。

交换机的类型

有几种可用的交换类型：

- 直接交换 [即路由类型]（`direct`）
- 主题交换（`topic`）
- 标题交换（`headers`）

- 扇出交换 [即广播模式] (fanout)

无名Exchange

无名交换机又为默认交换机，前面的实例中，没有指定交换机就可以进行消息收发就是使用了默认交换机，一般用空字符串 "" 进行识别。

```
channel.basicPublish("", queueName, null,
message.getBytes(StandardCharsets.UTF_8));
```

第一个参数是交换名称。空字符串表示默认交换或无名交换：如果队列中存在 routingKey (bindingKey) 指定的名称，则报文将被路由到该队列。

临时队列

为队列命名对我们来说至关重要--我们需要将工人指向同一个队列。在生产者和消费者之间共享队列时，给队列命名非常重要。

但我们的日志记录器并非如此。我们希望听到所有日志信息，而不仅仅是其中的一个子集。此外，我们只对当前流动的信息感兴趣，而不是旧信息。要解决这个问题，我们需要两样东西。

首先，每当我们连接 Rabbit 时，我们都需要一个新的、空的队列。为此，我们可以创建一个具有随机名称的队列，或者，更好的办法是让服务器为我们选择一个随机的队列名称。

其次，一旦我们断开消费者连接，队列应自动删除。

在 Java 客户端中，当我们不向 queueDeclare() 提供任何参数时，我们会创建一个非持久、独占、自动删除的队列，并生成一个名称：

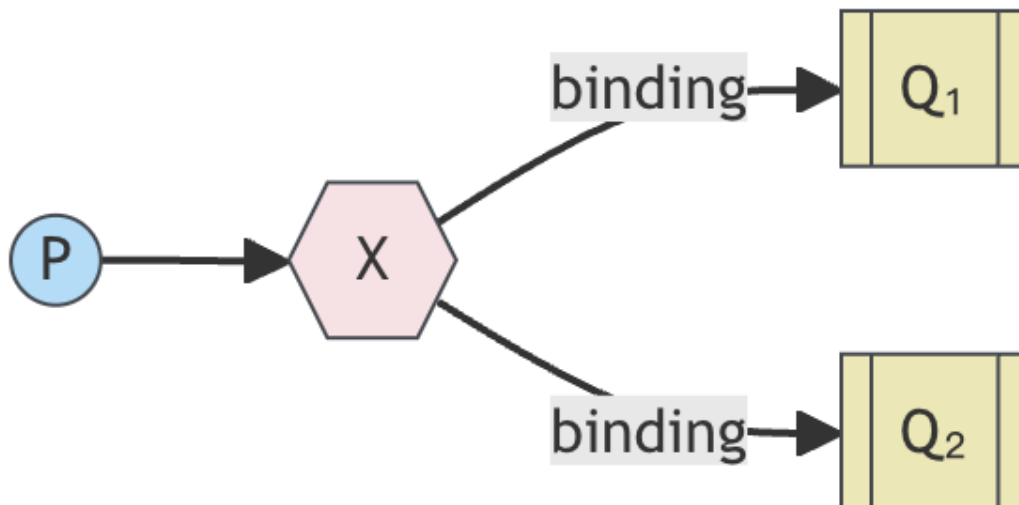
```
String queueName = channel.queueDeclare().getQueue();
```

运行后会生成一个类似于 amq.gen-F8uiBVXMAJSg4TZyVWDBow 的随机队列；到管理界面可以看到，队列属性为 【AD Excl】，其中“AD”为临时队列。

绑定 (Bindings)

我们已经创建了一个 fanout 交换机和一个队列。现在，我们需要告诉交易所向队列发送消息。交换机和队列之间的这种关系称为绑定。

图：img/mq15-binding.png



实例代码

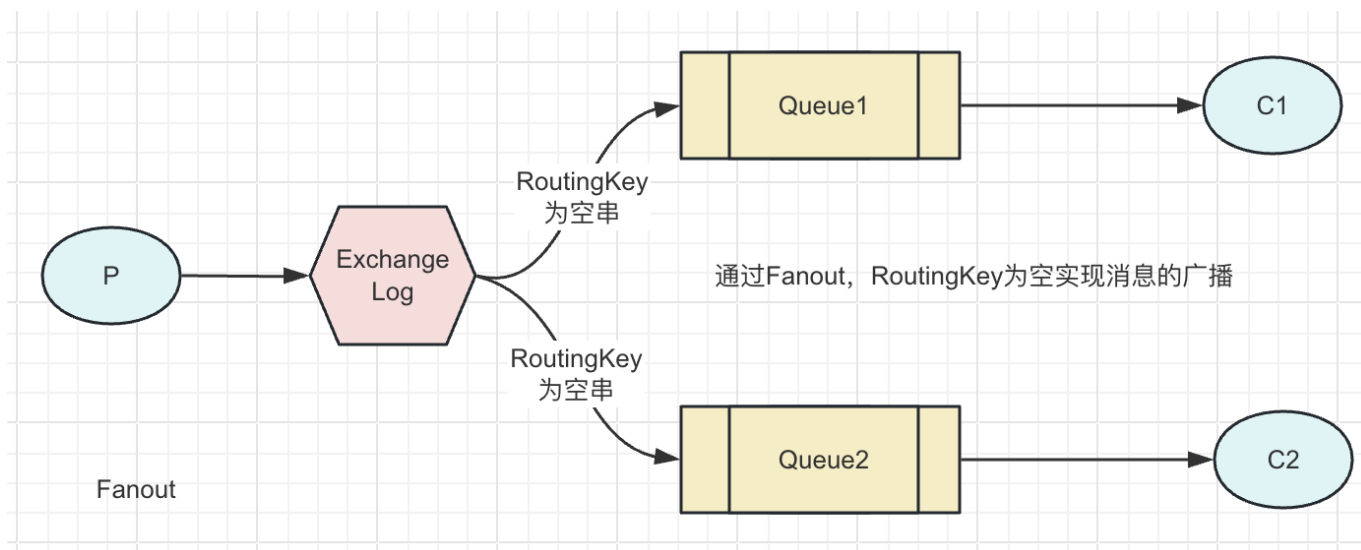
```
channel.queueBind(queueName, "logs", "");
```

如上，交换机会将“log”连接到对应队列名的队列中。

Fanout

发布日志信息的生产者程序与之前的教程并无太大区别。最重要的变化是，我们现在要将消息发布到我们的日志交换中心，而不是无名交换中心。我们需要在发送时提供路由键（**routingKey**），但其值在**Fanout**（扇出模式、广播模式）时交换机会忽略。

图： *img/mq16-Fanout.png*



生产者代码实例

所在目录 (fanout/SentLog.java)

```

private void testFanoutSent() {
    try (Channel channel = RabbitMQTestUtils.getChannel()) {
        /**
         * 声明交换机
         * 1.交换机名称
         * 2.交换机类型
         */
        channel.exchangeDeclare(RabbitMQConfigDiction.FANOUT_EXCHANGE_NAME,
            BuiltinExchangeType.FANOUT);
        System.out.println("Please input the message...");
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) {
            String message = scanner.next();
            // 这回是直接发送给交换机，而不是发送给具体的队列，因此队列为空；
            // 其实在发布订阅模式下，第二个参数也为"RoutingKey"
            channel.basicPublish(RabbitMQConfigDiction.FANOUT_EXCHANGE_NAME,
                "", null, message.getBytes(StandardCharsets.UTF_8));
            System.out.println("message: [" + message + "] is sent success!");
        }
    } catch (IOException | TimeoutException e) {
        e.printStackTrace();
    }
}

```

消费者代码实例

所在目录 (fanout/Received01.java&Received02.java)


```

private void testFanoutReceived() throws IOException, TimeoutException {
    Channel channel = RabbitMQTestUtils.getChannel();
    /**
     * 声明交换机
     * 1.交换机名称
     * 2.交换机类型
     */
    channel.exchangeDeclare(FANOUT_EXCHANGE_NAME, BuiltinExchangeType.FANOUT);
    /**
     * 获取临时队列，队列名称随即
     * 当消费者断开连接，队列自动删除
     */
    String queueName = channel.queueDeclare().getQueue();
    /**
     * 绑定交换机与信道
     * 1.队列名
     * 2.交换机名称
     * 3.RoutingKey-为空则为Fanout广播模式
     */
    channel.queueBind(queueName, FANOUT_EXCHANGE_NAME, "");
    System.out.println("Received01 Wait for received and log the message...");
    channel.basicConsume(queueName, true, RECEIVED_SUCCESS_CALL_BACK,
        RECEIVED_CANCEL_CALL_BACK);
}

```

Direct

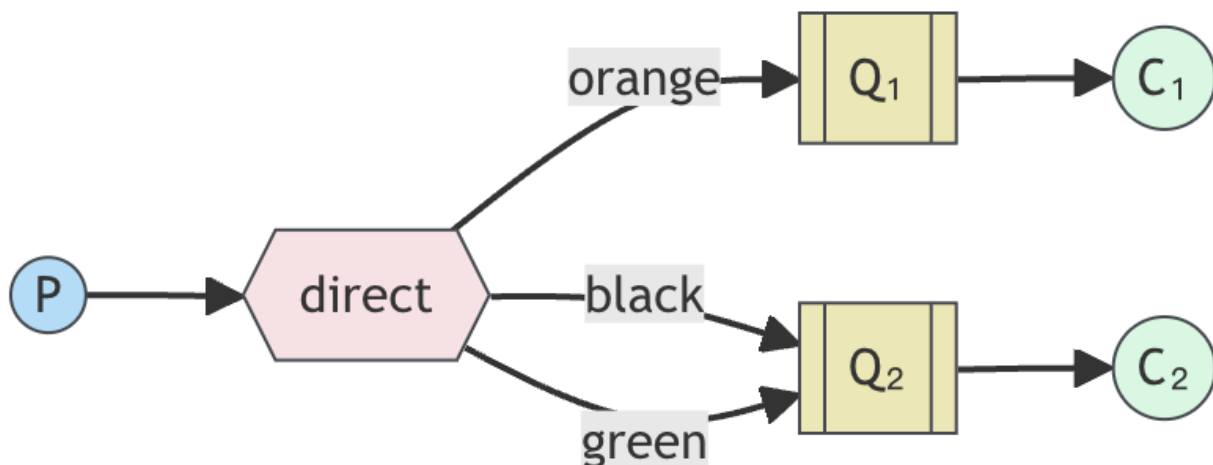
路由模式、直接交换机

上一教程中的系统会向所有用户广播所有信息。我们希望对其进行扩展，允许根据严重程度过滤信息。例如，我们可能希望将日志信息写入磁盘的程序只接收严重错误，而不浪费磁盘空间在警告或信息日志信息上。

Fanout我们使用的是扇出交换器，它没有给我们提供太多的灵活性——它只能进行无意识的广播。

我们将改用直接（Direct）交换。直接交换背后的路由算法很简单——消息会进入绑定密钥与消息路由密钥完全匹配的队列。

图： *img/mq17-direct.png*



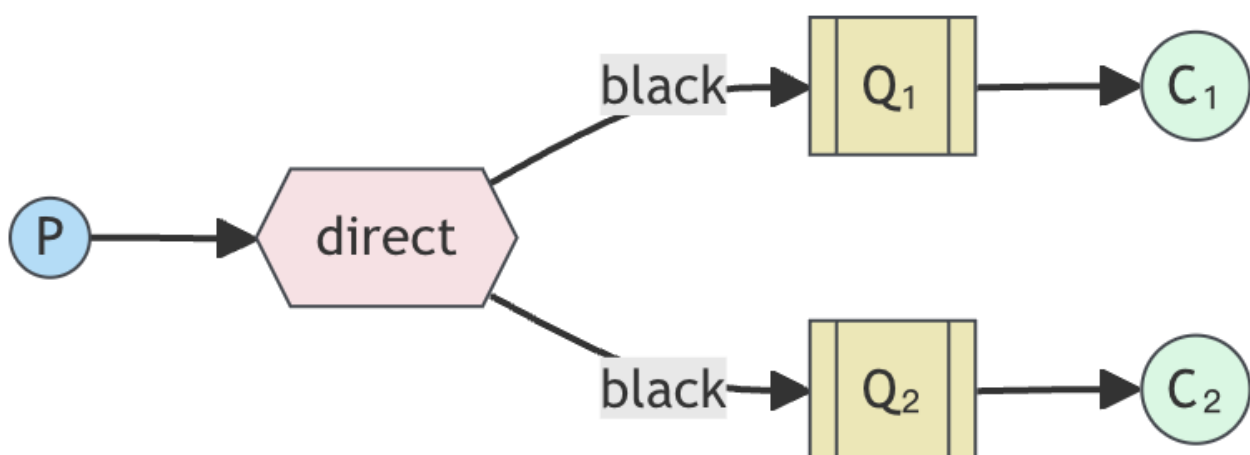
在此设置中，我们可以看到 `direct` 交换机 X 绑定了两个队列。第一个队列绑定了 `orange` 绑定密钥，第二个队列有两个绑定密钥，一个是 `black` 绑定密钥，另一个是 `green` 绑定密钥。

在这种情况下，发布到交易所的路由密钥（`Routing Key`）为 `orange` 的报文将被路由到 `Q1` 队列。路由密钥为 `black` 或 `green` 的报文将进入 `Q2`。所有其他报文都将被丢弃。

多重绑定

用同一个绑定密钥绑定多个队列是完全合法的。在我们的例子中，我们可以用绑定密钥 `black` 在 X 和 `Q1` 之间添加一个绑定。在这种情况下，直接交换将像扇出一样，向所有匹配队列广播报文。路由密钥为 `black` 的报文将同时发送到 `Q1` 和 `Q2`。

图： `img/mq18-MultipleBindings.png`



队列绑定

当一个队列需要消费多个RoutingKey的消息时，可以同时绑定多个RoutingKey

```
channel.queueBind(DIRECT_CONSOLE_QUEUE, DIRECT_EXCHANGE_NAME, "info");  
channel.queueBind(DIRECT_CONSOLE_QUEUE, DIRECT_EXCHANGE_NAME, "warning");
```

生产者代码实例

所在目录（direct/EmitLog.java）

```

private void testDirectSent() {
    try (Channel channel = RabbitMQTestUtils.getChannel()) {
        String routingKey = "";
        int count = 0;
        /**
         * 声明交换机
         * 1.交换机名称
         * 2.交换机类型
         */
        channel.exchangeDeclare(RabbitMQConfigDiction.DIRECT_EXCHANGE_NAME,
            BuiltinExchangeType.DIRECT);
        System.out.println("Please input the message...");
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) {
            String message = scanner.next();
            switch (count) {
                case 0:
                    routingKey = "info";
                    count++;
                    break;
                case 1:
                    routingKey = "error";
                    count++;
                    break;
                default:
                    routingKey = "warning";
                    count = 0;
            }
            // 这回是直接发送给交换机，而不是发送给具体的队列，因此队列为空；
            // 其实在发布订阅模式下，第二个参数也为"RoutingKey"
            channel.basicPublish(RabbitMQConfigDiction.DIRECT_EXCHANGE_NAME,
                routingKey, null, message.getBytes(StandardCharsets.UTF_8));
            System.out.println("message: [" + message + "], routingKey: [" +
                routingKey + "] is sent success!");
        }
    } catch (IOException | TimeoutException e) {
        e.printStackTrace();
    }
}

```

消费者代码实例

所在目录 (direct/DirectReceived01.java&DirectReceived02.java)

```

private void testDirectReceived() throws IOException, TimeoutException {
    Channel channel = RabbitMQTestUtils.getChannel();
    channel.queueDeclare(DIRECT_CONSOLE_QUEUE, false, false, false, null);
    /**
     * 声明交换机
     * 1. 交换机名称
     * 2. 交换机类型
     */
    channel.exchangeDeclare(DIRECT_EXCHANGE_NAME, BuiltinExchangeType.DIRECT);
    /**
     * 绑定交换机与信道
     * 1. 队列名
     * 2. 交换机名称
     * 3. RoutingKey
     */
    channel.queueBind(DIRECT_CONSOLE_QUEUE, DIRECT_EXCHANGE_NAME, "info");
    channel.queueBind(DIRECT_CONSOLE_QUEUE, DIRECT_EXCHANGE_NAME, "warning");
    // 绑定两个Routing key
    System.out.println("DirectReceived01 Wait for received and log the
message...");
    channel.basicConsume(DIRECT_CONSOLE_QUEUE, true,
RECEIVED_SUCCESS_CALL_BACK, RECEIVED_CANCEL_CALL_BACK);
}

```

Topics

主题模式

在上一教程中，我们改进了日志系统。我们不再使用只能进行虚假广播的Fanout交换机，而是使用了Direct交换机，并获得了选择性接收日志的可能性。

虽然使用直接交换机改进了我们的系统，但它仍有局限性--不能根据多个条件进行路由选择。

在我们的日志系统中，我们可能不仅想根据严重性订阅日志，还想根据日志的来源订阅日志。您可能从 syslog unix 工具中了解到这一概念，它可以根据严重性（info/warn/crit.....）和设施（auth/cron/kern.....）路由日志。

这给了我们很大的灵活性--我们可能只想监听来自 "cron "的严重错误，但也想监听来自 "kern "的所有日志。

要在日志系统中实现这一点，我们需要了解更复杂的 Topics 主题交换。

Topics的要求

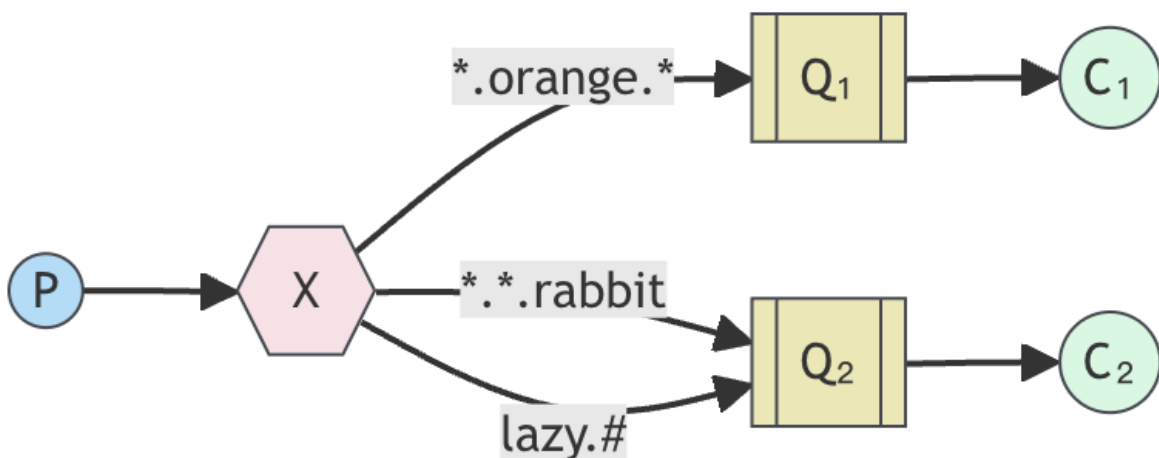
发送到主题交换中心的报文不能任意拼写 **routing_key**——它必须是一个用点分隔的单词列表。这些单词可以是任何内容，但通常会指定与信息相关的一些特征。以下是几个有效的路由密钥示例 `stock.usd.nyse`、`nyse.vyw`、`quick.orange.rabbit`。路由密钥的字数不限，最多 255 字节。

绑定密钥也必须采用相同的形式。主题交换背后的逻辑与直接交换类似--使用特定路由密钥发送的信息将被传送到所有使用匹配绑定密钥绑定的队列。不过，绑定密钥有两种重要的特殊情况：

- `*`（星号）可以完全替代一个单词。
- `#`（散列）可以替代零个或多个单词。

Topics匹配实例

图： *img/mq19-topics.png*



在本例中，我们要发送的信息都是描述动物的。这些信息将使用由三个单词（两个点）组成的路由密钥发送。路由键中的第一个单词描述速度，第二个单词描述颜色，第三个单词描述物种：

`<speed>.<colour>.<species>`

我们创建了三个绑定：Q1 与绑定键 `*.orange.*` 绑定，Q2 与 `*.*.rabbit` 和 `lazy.#` 绑定。

这些绑定可以概括为

- Q1 对所有橙色动物都感兴趣。
- Q2 希望听到所有关于兔子和懒惰动物的信息。

路由关键字设置为 `quick.orange.rabbit` 的报文将同时传送到这两个队列。信息

`lazy.orange.elephant` 也会同时发送到这两个队列。另一方面，`quick.orange.fox` 只会发送

到第一个队列，而 `lazy.brown.fox` 只会发送到第二个队列。尽管 `lazy.pink.rabbit` 匹配了两个绑定，但它只会被送到第二个队列一次。`quick.brown.fox` 不匹配任何绑定，因此会被丢弃。

如果我们违反约定，发送包含一个或四个单词的信息，如 `orange` 或 `quick.orange.new.rabbit`，会发生什么情况呢？这些信息与任何绑定都不匹配，因此会丢失。

另一方面，`lazy.orange.new.rabbit` 虽然有四个单词，但它会匹配最后一个绑定，并被传送到第二个队列。

tips: 当同一个队列被匹配两次，那么消息只会被消费一次。

主题交换功能强大，可以像其他交换一样运行。

当队列使用 "#"（哈希）绑定密钥绑定时，无论路由密钥如何，它都会接收所有信息，就像在Fanout扇出交换中一样。

如果绑定中没有使用特殊字符 "*"（星号）和 "#"（散列），则主题交换的行为与Direct直接交换无异。

代码实例

生产者代码实例

所在位置 (`topics/TopicsEmitLog.java`)

```

private void testTopicsSent() {
    try (Channel channel = RabbitMQTestUtils.getChannel()) {
        channel.exchangeDeclare(RabbitMQConfigDiction.TOPICS_EXCHANGE_NAME,
            BuiltinExchangeType.TOPIC);
        Map<String, String> bindingKeyMap = new HashMap<>();
        bindingKeyMap.put("quick.orange.rabbit", "R1&&R2");
        bindingKeyMap.put("lazy.orange.elephant", "R1&R2");
        bindingKeyMap.put("quick.orange.fox", "R1");
        bindingKeyMap.put("lazy.brown.fox", "R2");
        bindingKeyMap.put("lazy.pink.rabbit", "R2");
        bindingKeyMap.put("quick.brown.fox", "NULL");
        bindingKeyMap.put("quick.orange.male.rabbit", "NULL");
        bindingKeyMap.put("lazy.orange.male.rabbit", "R2");
        bindingKeyMap.forEach((routingKey, message) -> {
            System.out.println("message: [" + message + "], routingKey: [" +
routingKey + "] is sent success!");
            try {

channel.basicPublish(RabbitMQConfigDiction.TOPICS_EXCHANGE_NAME, routingKey,
                    null, message.getBytes(StandardCharsets.UTF_8));
                } catch (IOException e) {
                    e.printStackTrace();
                }
            });
        } catch (IOException | TimeoutException e) {
            e.printStackTrace();
        }
    }
}

```

消费者代码实例

所在位置 (topics/TopicsReceived01.java&TopicsReceived02.java)


```

private void testTopicsReceived() throws IOException, TimeoutException {
    Channel channel = RabbitMQTestUtils.getChannel();
    String queueName = channel.queueDeclare().getQueue();
    channel.exchangeDeclare(RabbitMQConfigDiction.TOPICS_EXCHANGE_NAME,
        BuiltinExchangeType.TOPIC);
    /**
     * 绑定队列、交换机、规则
     * 1. 队列名
     * 2. 交换机
     * 3. 规则
     */
    channel.queueBind(queueName, RabbitMQConfigDiction.TOPICS_EXCHANGE_NAME,
        TOPICS_RULE_02_ONLY_RABBIT);
    channel.queueBind(queueName, RabbitMQConfigDiction.TOPICS_EXCHANGE_NAME,
        TOPICS_RULE_03_LAZY_ALL); // 绑定两个规则
    System.out.println("Received02 Wait for received and log the message...");
    channel.basicConsume(queueName, true,
        RabbitMQConfigDiction.TOPICS_RECEIVED_SUCCESS_CALL_BACK,
        RabbitMQConfigDiction.RECEIVED_CANCEL_CALL_BACK);
}

```

死信队列

死信队列的概念

死信，顾名思义就是无法被消费的消息。一般来说，producer将消息投递到broker或者queue里了，consumer从queue取出消息进行消费，但是某些时候由于特定的原因导致queue中的某些消息无法被消费，这样的消息如果没有后续的处理，就变成了死信，有死信自然就有死信队列。

应用场景

- 为了保证订单业务的消息数据不丢失，需要使用到Rabbit MQ的死信队列机制，当消息消费发生异常时，将消息投入到死信队列中（防止消息丢失）
- 用户在商城下单成功并点击去支付后在指定的时间未支付时自动失效

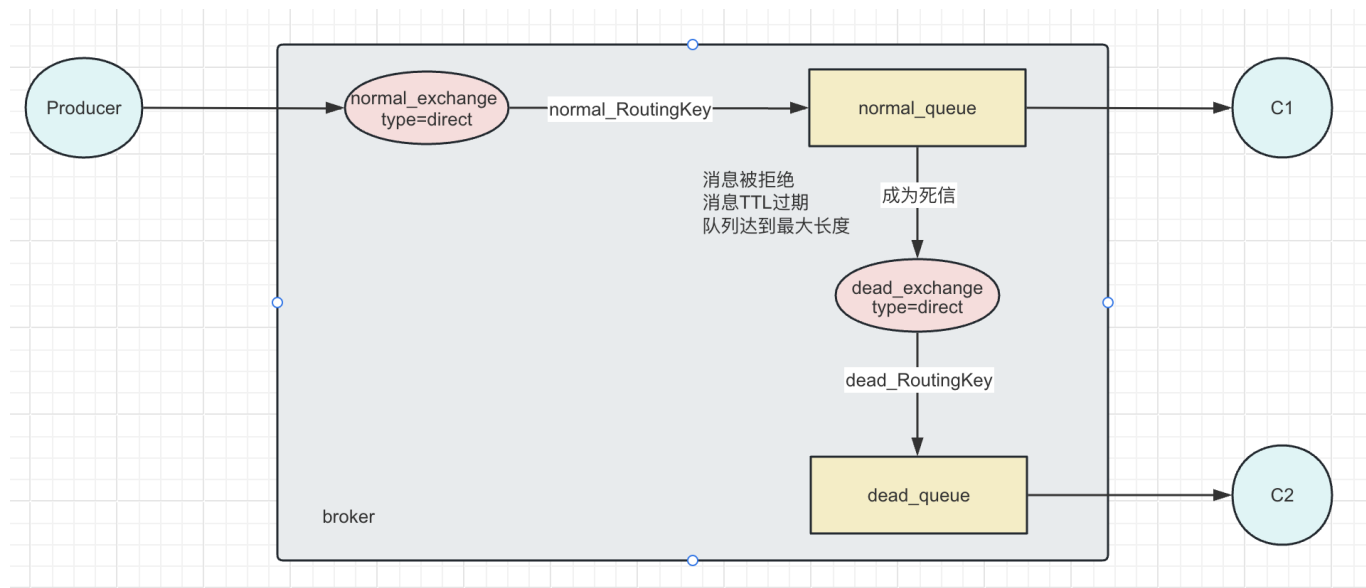
死信的来源

- 消息TTL（存活时间）过期
- 队列达到最大长度（队列已满，无法继续添加数据到MQ当中）
- 消息被拒绝（`basic.reject` 或 `basic.nack`）并且 `requeue=false`（不放回队列）

死信实战

代码架构图

图：img/mq20-deadQueue.png



tips: C1和C2不会同时消费，只有C1不消费的情况下才会被C2消费；并且C2的消息是由C1接受失败后转发给死信交换机并转发给C2的，而不是生产者生产后直接放入死信交换机的

设置死信参数

消费者模块

在声明队列的 `Channel.queueDeclare(String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object> arguments)` 方法中，`arguments` 参数可以存储诸如死信队列交换机、死信队列Routing Key、TTL超时时间、最大队列长度等信息。

```

/**
 * 正常队列声明
 * 在此处，正常的交换机声明需要用到最后的map参数指定消息成为死信后要转发至死信交换机
 * 队列声明中的arguments用于承接转发规则等信息
 */
Map<String, Object> arguments = new HashMap<>();
// 正常队列声明设置死信交换机
arguments.put("x-dead-letter-exchange",
RabbitMQConfigDiction.DEAD_EXCHANGE_NAME);
// 设置死信RoutingKey
arguments.put("x-dead-letter-routing-key", DEAD_ROUTING_KEY);
// 过期时间 - 10s = 10000ms，也可以在生产者进行声明
arguments.put("x-message-ttl", 10000);
// 设置队列最大长度 - 一旦超过此长度则会交由死信队列消费
arguments.put("x-max-length", 6);
channel.queueDeclare(RabbitMQConfigDiction.NORMAL_QUEUE, false, false, false,
arguments);

```

生产者模块

在生产者当中，可以通过 `basicPublish(String exchange, String routingKey, BasicProperties props, byte[] body)` 中的 `props` 参数构建过程添加诸如过期时间等参数。

```

// 设置死信参数 TTL时间
AMQP.BasicProperties props = new AMQP.BasicProperties()
    .builder()
    .expiration("10000") // 过期时间, ms
    .build();
channel.basicPublish(RabbitMQConfigDiction.NORMAL_EXCHANGE_NAME,
NORMAL_ROUTING_KEY,
    props,
    message.getBytes(StandardCharsets.UTF_8));

```

TTL过期

生产者模块

所在位置 (deadmsgqueue/DeadLetterProducer.java)

```

/**
 * 生产者 -
 * 生产者只需要
 * 因为死信转发
 */private void testMaxTTLSent() {
    try (Channel channel = RabbitMQTestUtils.getChannel()) {
        // 设置死信参数 TTL时间
        AMQP.BasicProperties props = new AMQP.BasicProperties()
            .builder()
            .expiration("10000") // 过期时间, ms
            .build();
        for (int i = 0; i < 10; i++) {
            String message = "info " + i;
            System.out.println("Producer sent message is : [" + message + "]");
            channel.basicPublish(RabbitMQConfigDiction.NORMAL_EXCHANGE_NAME,
                NORMAL_ROUTING_KEY, props,
                    message.getBytes(StandardCharsets.UTF_8));
        }
    } catch (IOException | TimeoutException e) {
        e.printStackTrace();
    }
}

```

正常消费者模块

所在位置 (deadmsgqueue/NormalConsumer.java)

```

/**
 * 正常消费方法
 * 正常消费失败后，由正常交换机转发给死信交换机，之后由后者对应消费者进行消费
 * 而不是直接由生产者直接发送给死信交换机
 */
private void testNormalReceived() throws IOException, TimeoutException {
    Channel channel = RabbitMQTestUtils.getChannel();
    // 独自生成队列与交换机，用作正常消费的队列与交换机
    /**
     * 正常队列声明
     * 在此处，正常的交换机声明需要用到最后的map参数指定消息成为死信后要转发至死信交换机
     * 队列声明中的arguments用于承接转发规则等信息
     */
    Map<String, Object> arguments = new HashMap<>();
    // 正常队列声明设置死信交换机
    arguments.put("x-dead-letter-exchange",
RabbitMQConfigDiction.DEAD_EXCHANGE_NAME);
    // 设置死信RoutingKey
    arguments.put("x-dead-letter-routing-key", DEAD_ROUTING_KEY);
    channel.queueDeclare(RabbitMQConfigDiction.NORMAL_QUEUE, false, false,
false, arguments);
    // 死信队列声明
    channel.queueDeclare(RabbitMQConfigDiction.DEAD_QUEUE, false, false, false,
null);
    // 声明交换机，类型为direct
    channel.exchangeDeclare(RabbitMQConfigDiction.NORMAL_EXCHANGE_NAME,
BuiltinExchangeType.DIRECT);
    channel.exchangeDeclare(RabbitMQConfigDiction.DEAD_EXCHANGE_NAME,
BuiltinExchangeType.DIRECT);
    // 关系绑定
    channel.queueBind(RabbitMQConfigDiction.NORMAL_QUEUE,
RabbitMQConfigDiction.NORMAL_EXCHANGE_NAME, NORMAL_ROUTING_KEY);
    channel.queueBind(RabbitMQConfigDiction.DEAD_QUEUE,
RabbitMQConfigDiction.DEAD_EXCHANGE_NAME, DEAD_ROUTING_KEY);
    // 交换机与队列绑定
    System.out.println("Normal is Received ...");
    channel.basicConsume(RabbitMQConfigDiction.NORMAL_QUEUE, true,
        RabbitMQConfigDiction.TOPICS_RECEIVED_SUCCESS_CALL_BACK,
        RabbitMQConfigDiction.RECEIVED_CANCEL_CALL_BACK);
}

```

死信消费者

所在位置（deadmsgqueue/DeadLetterConsumer.java）

```

/**
 * 正常消费方法
 * 正常消费失败后，由正常交换机转发给死信交换机，之后由后者对应消费者进行消费
 * 而不是直接由生产者直接发送给死信交换机
 */
private void testDeadLetterReceived() throws IOException, TimeoutException {
    Channel channel = RabbitMQTestUtils.getChannel();
    // 死信队列声明
    channel.queueDeclare(RabbitMQConfigDiction.DEAD_QUEUE, false, false, false,
null);
    // 声明交换机，类型为direct
    channel.exchangeDeclare(RabbitMQConfigDiction.DEAD_EXCHANGE_NAME,
BuiltinExchangeType.DIRECT);
    // 关系绑定
    channel.queueBind(RabbitMQConfigDiction.DEAD_QUEUE,
RabbitMQConfigDiction.DEAD_EXCHANGE_NAME, DEAD_ROUTING_KEY);
    // 交换机与队列绑定
    System.out.println("DeadLetterConsumer is Received ...");
    channel.basicConsume(RabbitMQConfigDiction.DEAD_QUEUE, true,
        RabbitMQConfigDiction.TOPICS_RECEIVED_SUCCESS_CALL_BACK,
        RabbitMQConfigDiction.RECEIVED_CANCEL_CALL_BACK);
}

```

最大队列长度

只需要将上方生产者超时参数部分屏蔽，并且在正常消费者中的队列声明参数加入以下属性即可

```

// 设置队列最大长度 - 一旦超过此长度则会交由死信队列消费
arguments.put("x-max-length", 6);

```

此时，如果生产者发送的信息，正常消费者有超过6条未消费，就会转发到死信交换机中。

消息被拒

- 将消费者的最大长度、超时设定部分代码屏蔽，并且重写消息接收成功后的回调函数
- 消费成功回调函数中需要声明拒收消息、是否返回队列等属性
- 回调函数中除了加入拒收方法 `basicReject(long deliveryTag, boolean requeue)` 还要加入接收应答 `basicAck(long deliveryTag, boolean multiple)` 方法
- 需要开启手动应答模式

```

/**
 * 声明队列拒收
 * 1. 队列标签
 * 2. 是否返回队列 - 指当前正常normal队列，而不是死信队列
 */
channel.basicReject(message.getEnvelope().getDeliveryTag(), false);

```

重写的消费成功回调方法

```

DeliverCallback deliverCallback = (consumerTag, message) -> {
    String msg = new String(message.getBody(), StandardCharsets.UTF_8);
    if (msg.contains("5")) {
        System.out.println("[!] This message is dont received, and sent to dead
exchange: [" + msg + "]);
        /**
         * 声明队列拒收
         * 1. 队列标签
         * 2. 是否返回队列 - 指当前正常normal队列，而不是死信队列
         */
        channel.basicReject(message.getEnvelope().getDeliveryTag(), false);
    } else {
        System.out.println("[X] Received ... The message is [" + msg
            + "] And the RoutingKey type is [" +
message.getEnvelope().getRoutingKey() + "]);
        channel.basicAck(message.getEnvelope().getDeliveryTag(), false);
    }
};
// 需要开启手动应答
channel.basicConsume(RabbitMQConfigDiction.NORMAL_QUEUE, false,
    deliverCallback,
    RabbitMQConfigDiction.RECEIVED_CANCEL_CALL_BACK);

```

Spring AMQP

使用Spring Boot整合Spring AMQP实现Rabbit MQ

关键依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-amqp</artifactId>
  <version>3.1.5</version>
</dependency>
```

延迟队列

其实是上一节死信队列中 TTL过期 的一个延伸；相当于C1被关闭，经过TTL的超时时间后由normal交换机转到dead交换机，随后直接被C2消费。

当然，使用插件后可以不依赖死信队列模式进行处理。

延迟队列概念

延迟队列内部是有序的，最重要的特性就体现在它的延迟属性上。延迟队列中的元素是希望在指定时间到了以后或之前取出和处理，简单来说，延迟队列就是用来存放需要在指定时间被处理的元素的队列。

延迟队列使用场景

- 订单在十分钟内未支付自动取消
- 新创建的店铺如果在10天内未上传商品，则自动发送信息提醒
- 用户注册成功后，如果三天内未登录则进行短信提醒
- 用户发起退款，如果三天内没有得到处理则通知相关运营人员
- 预定会议后，需要在预定的时间点前10分钟通知各个与会人员参加会议

以上场景都有一个特点，需要在某个事件发生之后或者之前的指定时间点前完成某一项任务，如发生订单事件，在十分钟后检查该订单支付状态，然后将未支付的订单进行关闭；看起来似乎使用定时任务，一直轮询查询，每秒查一次，但是如果短期内订单量很大，活动期间达到百万甚至千万级别，那么使用轮询是不可群的，会给数据库造成很大压力，无法满足业务要求且性能低下。

整合Spring Boot

项目所需依赖


```

<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.6.10</version>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-
boot-starter-amqp -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
    <version>3.1.5</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.springframework.amqp/spring-
rabbit-test -->
  <dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit-test</artifactId>
    <version>3.1.5</version>
    <scope>test</scope>
  </dependency>
  <!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->
  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.83</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger2 -->
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger-ui -
->

```

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.9.2</version>
</dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>2.6.10</version>
      <!-- 如果使用了lombok -->
      <configuration>
        <excludes>
          <exclude>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
          </exclude>
        </excludes>
      </configuration>
      <!-- 如果使用了lombok -->
    </plugin>
  </plugins>
</build>
```

添加SwaggerConfig

添加SwaggerConfig方便使用Swagger进行测试（由于项目启动异常已经屏蔽该部分代码，改用postman进测试）

```

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.Contact;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConfig {
    @Bean
    public Docket webApiConfig() {
        return new Docket(DocumentationType.SWAGGER_2)
            .groupName("webApi")
            .apiInfo(webApiInfo())
            .select().build();
    }

    private ApiInfo webApiInfo() {
        return new ApiInfoBuilder()
            .title("rabbitmq接口文档")
            .description("本文档描述了Rabbit MQ微服务接口定义")
            .version("v1.0.0")
            .contact(new Contact("Jayce", "http://127.0.0.1",
                "jayce404@foxmail.com")) // 一些个人信息随意填写
            .build();
    }
}

```

配置文件application.yml

```

server:
  port: 8999

# mq
spring:
  rabbitmq:
    host: 127.0.0.1
    port: 5672
    username: guest
    password: guest
    virtual-host: /hello

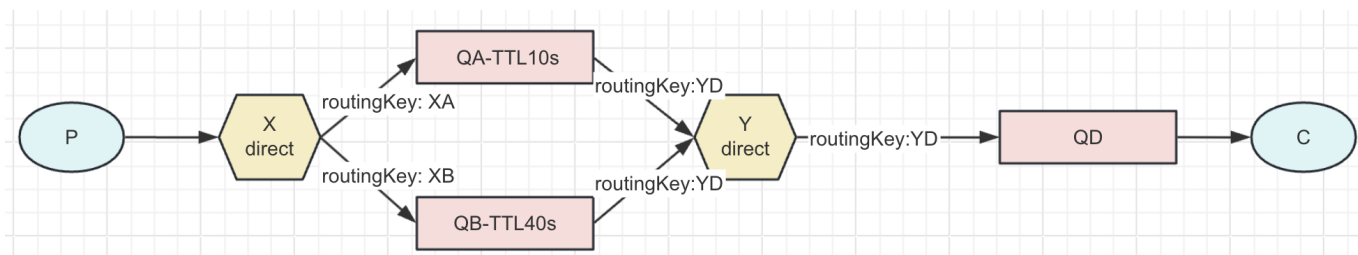
```

队列TTL

代码架构图

创建两个队列QA和QB，两者队列 TTL分贝设置为10S和40S，然后再创建一个交换机X和死信交换机Y，它们的类型都是 `direct`，创建一个死信队列QD，它们绑定关系如下：

图： *img/mq21-TTL.png*



配置绑定关系代码

在之前未整合Spring AMQP时，我们的队列创建、声明，交换机创建、声明均放在生产者消费者中，现在我们可以单独创建一个配置类来存储队列、交换机的声明及绑定信息。

由于此处代码重复率较高，因此只做几个举例，详见springboot-rabbitmq的 `config/TtlQueueConfig.java`

交换机声明方法

```
@Bean("yDlExchange")
public DirectExchange yDlExchange() {
    return new DirectExchange(DEAD_LETTER_EXCHANGE_Y);
}
```

队列声明方法

```

@Bean("queueQA")
public Queue queueQA() {
    Map<String, Object> arguments = new HashMap<>(3);
    // 正常队列声明设置死信交换机
    arguments.put("x-dead-letter-exchange", DEAD_LETTER_EXCHANGE_Y);
    // 设置死信RoutingKey
    arguments.put("x-dead-letter-routing-key", ROUTING_KEY_YD);
    arguments.put("x-message-ttl", 10000);
    return QueueBuilder // 构建一个队列
        .durable(QUEUE_A) // 队列名称
        .withArguments(arguments) // 相当于Java声明中的参数map
        .build();
}
@Bean("queueDLQD")
public Queue queueDLQD() {
    return QueueBuilder.durable(DEAD_LETTER_QUEUE_D).build();
}

```

绑定方法

此处采用了两种绑定形式，一种是直接使用方法体内的队列、交换机进行声明；一种是使用Spring依赖注入进行声明

此处使用到了 `@Qualifier(BeanName)` 注解，该注解有以下特性：

在Spring Boot中，`@Qualifier` 注解是一个用于解决bean的歧义问题的工具。当你有多个相同类型的bean在Spring容器中时，Spring默认自动装配机制可能无法决定应该注入哪一个bean。这时，`@Qualifier` 注解可以帮助你指定注入哪一个具体的bean。

```

/**
 * 队列QA与交换机X绑定
 * @return
 */
@Bean
public Binding queueQABindingExchangeX() {
    return BindingBuilder.bind(queueQA())
        .to(xExchange())
        .with(ROUTING_KEY_XA);
}
/**
 * 队列QB与交换机X绑定
 * @return
 */
@Bean
public Binding queueQBBindingExchangeX(@Qualifier("queueQB")Queue queueQB,
                                         @Qualifier("xExchange") DirectExchange
xExchange) {
    return BindingBuilder.bind(queueQB).to(xExchange).with(ROUTING_KEY_XB);
}

```

生产者与消费者

与之前Java中的 `Channel.basicPublish()` 不同，在Spring AMQP中可以使用依赖注入将 `RabbitTemplate` 注入到模块中，并在生产者和消费者使用：

- 生产者使用的是 `RabbitTemplate.convertAndSend()` 进行生产者消息发送

生产者

所在位置（`controller/SentMsgController.java`）

```
@Resource
private RabbitTemplate rabbitTemplate;
/**
 * 队列TTL的消息发送
 * @param message 发送消息
 */
@PostMapping("sendMsg01")
public void sendMsg01(String message) {
    log.info("[X]The message is : [{}], that is send success!", message);
    rabbitTemplate.convertAndSend(EXCHANGE_X, ROUTING_KEY_XA,
    message.getBytes(StandardCharsets.UTF_8));
    rabbitTemplate.convertAndSend(EXCHANGE_X, ROUTING_KEY_XB,
    message.getBytes(StandardCharsets.UTF_8));
}
```

消费者

与生产者不同，消费者是通过监听来实现消费的，监听注解 `@RabbitListener(queues = QUEUE_NAME)`

所在位置 (consumer/DeadLetterQueueConsumer.java)

```

import com.rabbitmq.client.Channel;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

import java.nio.charset.StandardCharsets;

import static org.example.util.TtlCommonDiction.DEAD_LETTER_QUEUE_D;

/**
 * 队列TTL - 消费者
 *
 */
@Component
@Slf4j
public class DeadLetterQueueConsumer {
    @RabbitListener(queues = DEAD_LETTER_QUEUE_D)
    public void receivedQDMsg(Message message, Channel channel) {
        log.info("[X]Received message success, the message is [{}], and routing Key is [{}]",
            new String(message.getBody(), StandardCharsets.UTF_8),
            message.getMessageProperties().getReceivedRoutingKey());
    }
}

```

延迟队列优化

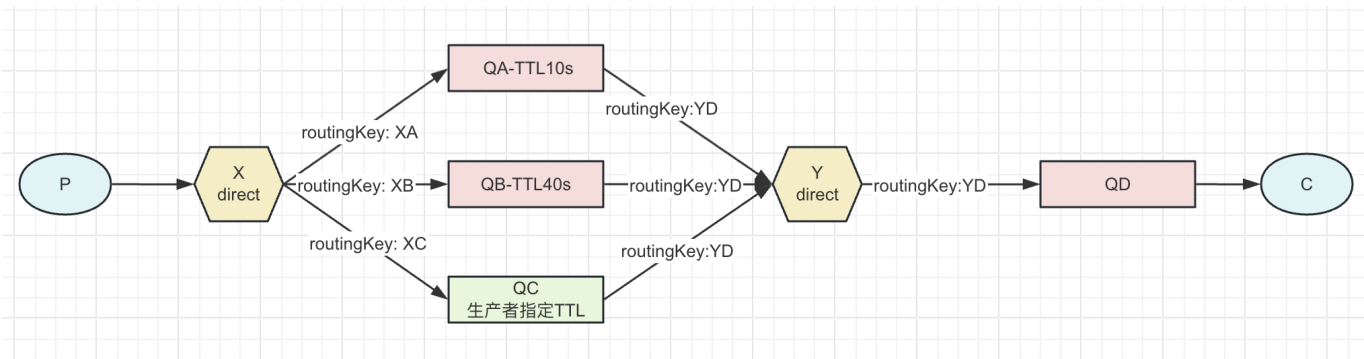
如上述例子中，第一条消息10s后变成了死信消息然后被消费者消费；第二条为40s。

如果这样使用，会造成每增加一个新的时间需求就要增加一个新的队列。这里只有10s和40s，如果需要一个小时后处理，就要增加一个TTL为一小时的队列，如果是预定会议室然后通知的场景，需要增加无数队列才能满足需求。

代码架构图

此处创建队列QC，与QA、QB不同，**QC**本身不指定TTL时长，而是交由生产者进行指定。

图：img/mq22-TTLMakeSuperior.png



配置绑定关系代码

进行相应的绑定关系，并且不再设置TTL时长

```
/**
 * 不设置TTL，交由生产者进行设置
 * @return 无TTL的队列
 */
@Bean("queueQC")
public Queue queueQC() {
    Map<String, Object> arguments = new HashMap<>(2);
    arguments.put("x-dead-letter-exchange", DEAD_LETTER_EXCHANGE_Y);
    arguments.put("x-dead-letter-routing-key", ROUTING_KEY_YD);
    return QueueBuilder.durable(QUEUE_C).withArguments(arguments)
        .build();
}

/**
 * 队列QC与交换机X绑定
 * @param queue 队列QC
 * @param exchange 交换机X
 * @return 返回绑定关系
 */
@Bean
public Binding queueQCBindingExchangeX(@Qualifier("queueQC") Queue queue,
    @Qualifier("xExchange") DirectExchange
exchange) {
    return BindingBuilder.bind(queue).to(exchange).with(ROUTING_KEY_XC);
}
```

tips: `@Qualifier` 已经指定了Bean名字，因此方法内参数命名可以随意命名

生产者与消费者

在生产者中，使用 `convertAndSend()` 中的属性进行TTL设置，
`message.getMessageProperties().setExpiration(String TTL)` 为设置超时时间

```

/**
 * 由生产者指定TTL的方法
 * @param sendMsgAndTimeDTO 存储消息及TTL时间
 */
@PostMapping("sendMsg02")
public void sendExpirationMsg02(SendMsgAndTimeDTO sendMsgAndTimeDTO) {
    log.info(JSONObject.toJSONString(sendMsgAndTimeDTO));
    log.info("[X]The message is send, message is [{}], and TTL is [{}]",
        sendMsgAndTimeDTO.getMessage(),
        sendMsgAndTimeDTO.getTtlTime());
    rabbitTemplate.convertAndSend(EXCHANGE_X, ROUTING_KEY_XC,
        sendMsgAndTimeDTO.getMessage().getBytes(StandardCharsets.UTF_8),
        message -> {
            // 设置超时时间

message.getMessageProperties().setExpiration(String.valueOf(sendMsgAndTimeDTO.g
etTtlTime()));
            return message;
        });
}

```

消费者由于与上一个例子中同为监听 `DEAD_LETTER_QUEUE_D` 队列的方法，因此不做修改，直接沿用上一例中的消费者即可。

基于死信存在的问题

我们上述的延迟队列是基于死信进行处理的，这存在一个问题，就是先进先出，这会导致如下问题：

- 向队列发送一条20S的消息1，随后再向队列发送一条2S的消息2
- 按照预想结果，消费者应当先消费消息2，再消费消息1
- 但是查看控制台发现，消费者会等待消息1达到20s消费后，才会消费消息2
- 这是队列先进先出导致的

测试日志

```
11:15:27.446 : {"message":"测试信息1","ttlTime":20000}
11:15:27.446 : [X]The message is send, message is [测试信息1], and TTL is [20000]
11:15:32.739 : {"message":"测试信息2","ttlTime":2000}
11:15:32.739 : [X]The message is send, message is [测试信息2], and TTL is [2000]
11:15:47.472 : [X]Received message success, the message is [测试信息1], and routing Key is [yd]
11:15:47.473 : [X]Received message success, the message is [测试信息2], and routing Key is [yd]
```

在消息属性上设置TTL的方式（即生产者设置），消息可能不会按时“死亡”，因为Rabbit MQ只会检查第一个消息是否过期，如果过期则丢到死信队列。如果第一个消息延时时长很长，而第二个消息的延时时长很短，第二个消息并不会得到优先消费。

Rabbit MQ插件实现队列优化

按照上面所讲到的，队列为先进先出的形式，如果需要解决上述问题则需要使用Rabbit MQ的 `rabbitmq_delayed_message_exchange` 插件来进行处理。

插件安装

- 需要下载插件，需要到Rabbit MQ官网中的 <https://www.rabbitmq.com/community-plugins> 寻找到对应插件，并选择 Releases 模块下载 `.ez` 为文件后缀的文件
- 下载插件后放入Rabbit MQ安装目录中的 `plugins` 目录
- 执行 `rabbitmq-plugins enable rabbitmq_delayed_message_exchange` 启用插件

插件安装成功后可以使用 `rabbitmq-plugins list` 查看插件安装状态；也可以在管理界面的交换机新增模块中查看“Type”属性是否新增了一个 `x-delayed-messahe` 类型

插件处理原理

原本上述例子中，我们是基于死信队列中的队列或者生产者进行TTL设定的，优缺点如下

TTL 设置 方式	优点	缺点
在队列中设置	不会存在先进先出原则，而是正常依照设定时间消费	延时时间固定，每次设定一个不同的延时时间就要重新创建一个队列
在生产者中设置	不需要设置多个定时队列，应用灵活	存在先进先出原则，一旦前一条消息的延时时间远大于后一条消息，后一条则需要等待前一条时间才能被消费；存在消费延迟

当启用 `rabbitmq_delayed_message_exchange` 插件后，可以在交换机中进行TTL设置，摆脱了生产者设置中先进先出原则带来的消费延迟。

运行流程

P ==> `delayed.exchange;type: direct` = `delay.routingkey` ==> `delayed.queue` ==> C

配置绑定关系代码

所在位置（`config/DelayedQueueConfig.java`）

```

/**
 * 配置队列
 * @return
 */
@Bean
public Queue delayedQueue() {
    return new Queue(DELAYED_QUEUE_NAME);
}
/**
 * 基于插件配置自定义交换机
 * @return CustomExchange 返回一个自定义类型的交换机
 */
@Bean
public CustomExchange delayedExchange() {
    Map<String, Object> arguments = new HashMap<>();
    // 延迟类型, 直接类型, 因为RoutingKey是个固定值
    arguments.put("x-delayed-type", "direct");
    /**
     * 自定义交换机
     * 1. 交换机名称
     * 2. 交换机类型
     * 3. 是否持久化
     * 4. 是否自动删除
     * 5. 其他参数
     */
    return new CustomExchange(DELAYED_EXCHANGE_NAME,
        X_DELAYED_MESSAGE_EXCHANGE_TYPE, // x-delayed-message
        true,
        false,
        arguments); arguments.put("x-delayed-type", "direct");
}
/**
 * 队列与交换机绑定
 * @return
 */
@Bean
public Binding delayedQueueBindingDelayedExchange(Queue delayedQueue, Exchange
delayedExchange) {
    return BindingBuilder.bind(delayedQueue)
        .to(delayedExchange)
        .with(DELAYED_ROUTING_KEY)
        .noargs();
}

```

生产者与消费者

在生产者消息中指定延迟时间，时间传递到交换机中，交换机就会进行延迟时间设定，这样消息就不会堆积在队列中，而由交换机进行调配。

生产者

```
/**
 * 发送延迟消息
 * @param sendMsgAndTimeDTO
 */
@PostMapping("sendMsg03")
public void sendDelayedMsg(SendMsgAndTimeDTO sendMsgAndTimeDTO) {
    log.info(JSONObject.toJSONString(sendMsgAndTimeDTO));
    log.info("[X]The message is send, message is [{}], and delayed time is [{}]",
            sendMsgAndTimeDTO.getMessage(),
            sendMsgAndTimeDTO.getTtlTime());
    rabbitTemplate.convertAndSend(DELAYED_EXCHANGE_NAME, DELAYED_ROUTING_KEY,
            sendMsgAndTimeDTO.getMessage().getBytes(StandardCharsets.UTF_8),
            message -> {
                // 设置延迟时间

            message.getMessageProperties().setDelay(sendMsgAndTimeDTO.getTtlTime());
            return message;
        });
}
```

消费者

所在位置 (consumer/DelayedQueueConsumer.java)

```
@Component
@Slf4j
public class DelayedQueueConsumer {
    @RabbitListener(queues = TtlCommonDiction.DELAYED_QUEUE_NAME)
    public void receivedDelayedMsg(Message message, Channel channel) {
        String msg = new String(message.getBody(), StandardCharsets.UTF_8);
        log.info("[X] The message is received, body is [{}]", msg);
    }
}
```

运行结果

```
17:12:33.849: {"message":"测试信息1","ttlTime":20000}
17:12:33.849: [X]The message is send, message is [测试信息1], and delayed time
is [20000]
17:12:38.702: {"message":"测试信息2","ttlTime":2000}
17:12:38.702: [X]The message is send, message is [测试信息2], and delayed time
is [2000]
17:12:40.716: [X] The message is received, body is [测试信息2]
17:12:45.461: {"message":"测试信息3","ttlTime":200}
17:12:45.461: [X]The message is send, message is [测试信息3], and delayed time
is [200]
17:12:45.665: [X] The message is received, body is [测试信息3]
17:12:53.856: [X] The message is received, body is [测试信息1]
```

总结

延迟队列在需要延迟处理的场景下非常有用，使用RabbitMQ来实现延迟队列可以很好的利用RabbitMQ的特性，如：消息可靠发送、消息可靠投递、死信队列来保障消息至少被消费一次以及未被正确消费的消息不会被丢弃。另外，通过RabbitMQ集群的特性，可以很好的解决单点故障问题，不会因为单个节点挂掉导致延时队列不可用或者消息丢失。

当然，延时队列还有其他很多选择，例如利用Java的DelayQueue、Redis的zset、Quartz或者Kafka的时间轮，这些方式各有特点，看需要到的使用场景。

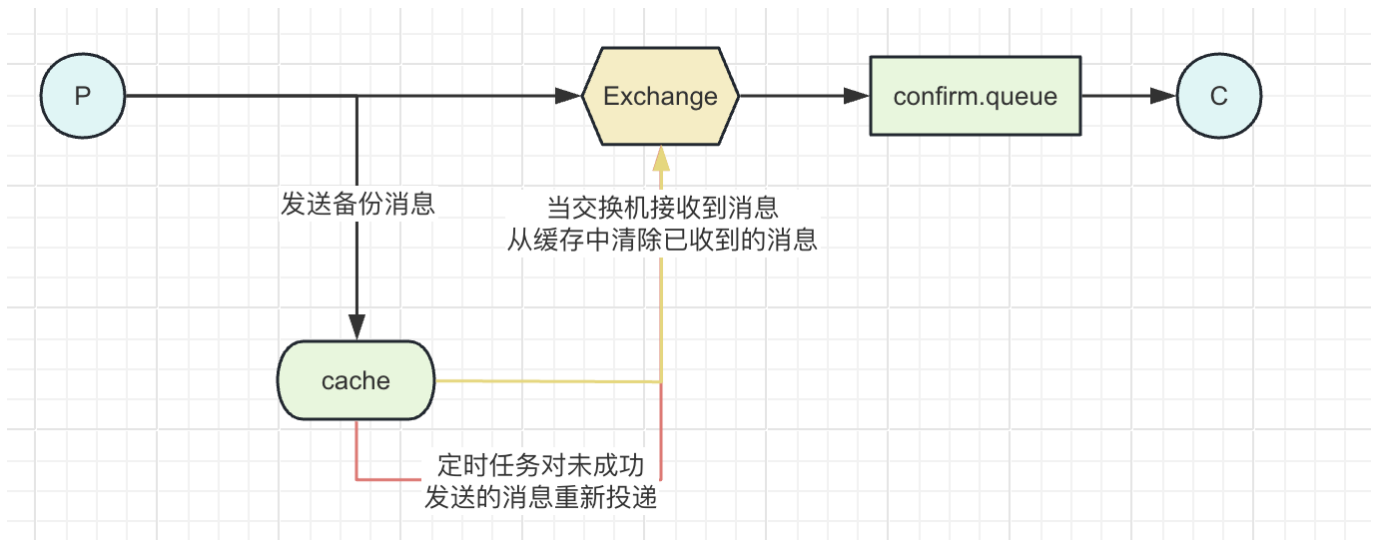
发布确认高级

在生产环境中由于一些不明原因导致RabbitMQ重启，在RabbitMQ重启期间生产者消息投递失败，导致消息丢失，需要手动处理和恢复。如何才能进行RabbitMQ的消息可靠投递？特别在比较极端的情况下，RabbitMQ集群不可用的时候，无法投递的消息应该如何处理？

发布确认SpringBoot版本

确认机制方案

图：*img/mq23-confirm.png*



代码架构

```
P ==> confirm.exchange; type: direct =key1=> confirm.queue ==> confirm consumer
```

配置文件

首先，配置文件 `application.yml` 中加入以下配置

```
spring:
  rabbitmq:
    publisher-confirm-type: CORRELATED
```

参数解析

- `NONE` 禁用发布确认模式，默认值
- `CORRELATED` 发布消息成功到交换机后会触发回调方法
- `SIMPLE` 类似于发布确认的单条确认，经测有两种效果
 - 第一种效果：和 `CORRELATED` 一样会触发回调方法
 - 第二种效果：发布消息成功后使用 `rabbitTemplate.waitForConfirms(long timeout)` 或 `rabbitTemplate.waitForConfirmsOrDie(long timeout)` 方法等待broker节点返回发送结果，根据返回结果来判断下一步的逻辑。值得注意的是，`rabbitTemplate.waitForConfirmsOrDie(long timeout)` 方法如果返回 `false` 则会关闭 `Channel`，接下来无法发送消息到broker

配置绑定关系代码

所在位置 (`config/ConfirmConfig.java`)


```

@Bean
public DirectExchange confirmExchange() {
    return new DirectExchange(CONFIRM_EXCHANGE_NAME);
}
@Bean
public Queue confirmQueue() {
    return new Queue(CONFIRM_QUEUE_NAME);
}
@Bean
public Binding confirmQueueBindingConfirmExchange(Queue confirmQueue,
                                                    DirectExchange
confirmExchange) {
    return BindingBuilder.bind(confirmQueue)
        .to(confirmExchange)
        .with(CONFIRM_ROUTING_KEY);
}

```

回调接口

发布确认是通过监听来实现的，监听需要实现 `RabbitTemplate.ConfirmCallback` 接口

接口源码

- `var1` 存储发送内容
- `var2` 存储的boolean值参数在接收成功时会返回 `true`，接收失败返回 `false`
- `var3` 存储失败原因

```

@FunctionalInterface
public interface ConfirmCallback {
    void confirm(@Nullable CorrelationData var1, boolean var2, @Nullable String
var3);
}

```

接口实现

定义一个 `MyConfirmCallback` 配置类，用于实现 `RabbitTemplate.ConfirmCallback` 接口

值得注意的是，在实现接口后，其实 `RabbitTemplate` 并没有识别到自己的接口被实现了；因此需要使用 `init` 自定义一个注入，将本身注入到 `RabbitTemplate` 中，并且开启 `@PostConstruct`，防止在 `RabbitTemplate` 之前这个方法被加载。如果不进行注入，方法则无效。

所在位置（`config/msgconfirm/MyExchangeConfirmCallback.java`）

```

package org.example.config.msgconfirm;

import com.alibaba.fastjson.JSONObject;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.rabbit.connection.CorrelationData;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.context.annotation.Configuration;
import org.springframework.util.StringUtils;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;

@Configuration
@Slf4j
public class MyExchangeConfirmCallback implements
RabbitTemplate.ConfirmCallback {
    @Resource
    private RabbitTemplate rabbitTemplate;

    /**
     * 注入
     * 将接口的实现注入到RabbitTemplate中
     * @ PostConstruct - 加载后构造
     */
    @PostConstruct
    public void init() {
        rabbitTemplate.setConfirmCallback(this);
    }

    /**
     * 交换机确认回调方法
     * @param correlationData 消息内容，存储消息id及相关信息
     * @param ack 接收确认成功true；失败false
     * @param cause 存储失败原因，成功为null；失败为false
     */
    @Override
    public void confirm(CorrelationData correlationData, boolean ack, String
cause) {
        String messageId = "";
        if (correlationData != null) {
            messageId = StringUtils.hasText(correlationData.getId()) ?
correlationData.getId(): "";
        }
        if (ack) {
            log.info("[Ex]Exchange is received success, the message id is [{}],
and correlationData is [{}]",
                messageId, JSONObject.toJSONString(correlationData));
        } else {

```

```

        log.error("[Ex Err]Exchange is received loss, the message id is
[{}], and cause is [{}], and correlationData is [{}]",
            messageId, cause,
            JSONObject.toJSONString(correlationData));
    }
}
}

```

生产者与消费者

生产者

所在目录（controller/ProducerController.java）

```

@PostMapping("sendMsg02")
public void testSendMsgConfirm(String message) {
    log.info("[S]Send message success! The message is : [{}]", message);
    // 交换机发布确认的所需参数；用于传递信息给到RabbitTemplate.ConfirmCallback
    CorrelationData correlationData = new
    CorrelationData(UUID.randomUUID().toString());
    rabbitTemplate.convertAndSend(CONFIRM_EXCHANGE_NAME,
        CONFIRM_ROUTING_KEY,
        message.getBytes(StandardCharsets.UTF_8),
        correlationData);
}

```

消费者

所在目录（consumer/ConfirmConsumer.java）

```

@RabbitListener(queues = CONFIRM_QUEUE_NAME)
public void receivedTestMsg(Message message) {
    log.info("[R]Received success! The message is [{}], and routing key is
[{}]",
        new String(message.getBody(), StandardCharsets.UTF_8),
        message.getMessageProperties().getReceivedRoutingKey());
}

```

调用结果

```
[S]Send message success! The message is : [测试信息3]
[R]Received success! The message is [测试信息3], and routing key is
[confirm.key]
[Ex]Exchange is received success, the message id is [a6c51dc7-a7bb-4357-8dad-
26a6343741f3],
and correlationData is [{"future":
{"cancelled":false,"done":true},"id":"a6c51dc7-a7bb-4357-8dad-26a6343741f3"}]
```

交换机确认

经过测试，我们如果将上述生产者部分的交换机名称换位错误名称（即找不到的交换机，模拟宕机），此时我们实现的回调接口会告知我们消息接收失败

报错明细

```
* 执行结果：
* Rabbit MQ报NOT_FOUND异常
* Shutdown Signal: channel error; protocol method: #method<channel.close>
(reply-code=404,
* reply-text=NOT_FOUND - no exchange 'confirm.exchange123' in vhost '/hello',
class-id=60, method-id=40)
*
* 实现的接口监听异常：
* [Ex Err]Exchange is received loss, the message id is [150f092d-c4a5-40fa-
a4e0-e3432c36d883],
* and cause is [channel error; protocol method: #method<channel.close>(reply-
code=404,
* reply-text=NOT_FOUND - no exchange 'confirm.exchange123' in vhost '/hello',
class-id=60, method-id=40)] ,
* and correlationData is [{"future":
{"cancelled":false,"done":true},"id":"150f092d-c4a5-40fa-a4e0-e3432c36d883"}]
```

由此可见，我们实现接口并在生产者发送消息时加入 `CorrelationData` 参数是有效的。但是用同样的方法测试，将交换机还原（即模拟交换机正常）；并且将Routing Key修改为错误的值（即模拟队列宕机或异常），此时发现消息正常发送，并且被确认接收了，只是消费者未消费而已。由此可见，上述方式只做了交换机确认，未做队列确认。因此，我们需要到下面所述的回退消息来进行队列确认。

回退消息

Mandatory 参数

在仅开启了生产者确认机制的情况下，交换机接收到消息后会直接给消息生产者发送确认信息，如果发现该消息不可路由（即交换机无法发送给队列，**routing**），那么消息会直接被丢弃，此时生产者是不知道消息被丢弃这个事情的。通过设置 `mandatory` 参数可以在当消息传递过程中不可达目的地时将消息返回给生产者。

回退实现

配置文件

队列确认，需要在配置文件中加入如下配置

```
# publisher-returns - 消息送达队列确认
spring:
  rabbitmq:
    publisher-returns: true
```

实现ReturnsCallback

Queue确认是通过实现 `RabbitTemplate.ReturnsCallback` 接口实现的，与前面的交换机确认一样，也需要在实现后将方法注入到RabbitTemplate中

所在位置（`config/msgconfirm/MyQueueReturnCallback.java`）

```

package org.example.config.msgconfirm;

import com.alibaba.fastjson.JSONObject;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.core.ReturnedMessage;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.context.annotation.Configuration;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;

/**
 * 用于队列确认的工具类
 * 重写了队列确认的方法 ConfirmCallback
 */
@Configuration
@Slf4j
public class MyQueueReturnCallback implements
RabbitTemplate.ReturnsCallback {
    @Resource
    private RabbitTemplate rabbitTemplate;

    @PostConstruct
    public void init() {
        rabbitTemplate.setReturnsCallback(this);
    }

    @Override
    public void returnedMessage(ReturnedMessage returnedMessage) {
        log.error("return message : {}",
JSONObject.toJSONString(returnedMessage));
        // 对应的correlationData存储的消息id，可以在此处获取，
        // 可以理解为correlationData就是headers，只是以map进行存储了
        // 其中，消息的id被存储在了"spring_returned_message_correlation"当中
        String springReturnedMessageCorrelation = (String)
returnedMessage.getMessage()
                .getMessageProperties()
                .getHeaders()
                .get("spring_returned_message_correlation");
        log.error("[Queue Err] Message is returned! The message exchange is :
[{}], and routing key is [{}], " +
                "and the message correlationData id is [{}], reply text
is [{}]!",
                returnedMessage.getExchange(),
                returnedMessage.getRoutingKey(),
                springReturnedMessageCorrelation,
                returnedMessage.getReplyText()); // 原因
    }
}

```

```

    }
  }
}

```

与 `ConfirmCallback` 接口不同，`ReturnsCallback` 接口虽然也是在监听消息，但是在队列接收到消息后，接口不会有任何动作，只有在队列接收失败后才会触发该接口的实现类。

该接口只有一个 `ReturnedMessage` 参数，一旦队列接收消息失败（也就是无队列接收到该消息），接口被触发，随即就会打印该参数。

可以看到 `ReturnedMessage` 存储了包括交换机、`RoutingKey` 等信息，如果在发送消息时，生产者携带了 `correlationData` 的id，那么将会被以Map的形式存储在 `headers` 的 `spring_returned_message_correlation` 当中，也就是上面例子中的 `returnedMessage.getMessage().getMessageProperties().getHeaders().get("spring_returned_message_correlation");` 获取对应的id

ReturnedMessage信息

```

{
  "exchange": "confirm.exchange",
  "message": {
    "body": "5rWL6K+V5L+h5oGvMQ==",
    "messageProperties": {
      "contentType": "application/octet-stream",
      "deliveryTag": 0,
      "finalRetryForMessageWithNoId": false,
      "headers": {
        "spring_returned_message_correlation": "47b8a2cb-3220-42ce-9a97-061e172ccd00"
      },
      "lastInBatch": false,
      "priority": 0,
      "projectionUsed": false,
      "publishSequenceNumber": 0,
      "receivedDeliveryMode": "PERSISTENT"
    },
    "replyCode": 312,
    "replyText": "NO_ROUTE",
    "routingKey": "confirm.key123"
  }
}

```

测试详情

- 当交换机错误时（也就是交换机宕机等消息未被交换机接收的情况），队列确认不会被触发
- 当消息正常被传递到队列时，队列确认不会被触发
- 当交换机正常、交换机错误（未被创建、宕机、routingKey错误等情况），队列确认方法实现会被触发，ReturnedMessage返回如上述的参数

实际上，两个确认的配置类（即 `ConfirmCallback` 和 `ReturnedCallback` 的实现类）都可以写在同一个类中，在注入时同事将它们set进 `RabbitTemplate` 即可。

所在位置（`config/msgconfirm/MyConfirmCallback.java`）


```

package org.example.config.msgconfirm;

import com.alibaba.fastjson.JSONObject;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.core.ReturnedMessage;
import org.springframework.amqp.rabbit.connection.CorrelationData;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.context.annotation.Configuration;
import org.springframework.util.StringUtils;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;

@Configuration
@Slf4j
public class MyConfirmCallback implements RabbitTemplate.ConfirmCallback,
RabbitTemplate.ReturnsCallback {
    @Resource
    private RabbitTemplate rabbitTemplate;

    @PostConstruct
    public void init() {
        rabbitTemplate.setConfirmCallback(this);
        rabbitTemplate.setReturnsCallback(this);
    }

    /**
     * 实现 RabbitTemplate.ConfirmCallback 接口
     * 交换机确认回调方法
     * @param correlationData 消息内容，存储消息id及相关信息
     * @param ack 接收确认成功true；失败false
     * @param cause 存储失败原因，成功为null；失败为false
     */
    @Override
    public void confirm(CorrelationData correlationData, boolean ack, String
cause) {
        String messageId = "";
        if (correlationData != null) {
            messageId = StringUtils.hasText(correlationData.getId()) ?
correlationData.getId(): "";
        }
        if (ack) {
            log.info("[Ex]Exchange is received success, the message id is [{}],
and correlationData is [{}]",
                messageId, JSONObject.toJSONString(correlationData));
        } else {
            log.error("[Ex Err]Exchange is received loss, the message id is

```

```

[{}], and cause is [{}], and correlationData is [{}]",
        messageId, cause,
        JSONObject.toJSONString(correlationData));
    }
}

/**
 * 只有但消息不可到达目的地时才回退给生产者
 * 实现 RabbitTemplate.ReturnsCallback 接口
 * 队列确认回调方法
 * @param returnedMessage 存储队列未接收到消息的信息，如RoutingKey、交换机等
 */
@Override
public void returnedMessage(ReturnedMessage returnedMessage) {
    // 对应的correlationData存储的消息id，可以在此处获取，
    // 可以理解为correlationData就是headers，只是以map进行存储了
    // 其中，消息的id被存储在了"spring_returned_message_correlation"当中
    String springReturnedMessageCorrelation = (String)
returnedMessage.getMessage()
        .getMessageProperties()
        .getHeaders()
        .get("spring_returned_message_correlation");
    log.error("[Queue Err] Message is returned! The message exchange is :
[{}], and routing key is [{}], " +
        "and the message correlationData id is [{}], reply text
is [{}]",
        returnedMessage.getExchange(),
        returnedMessage.getRoutingKey(),
        springReturnedMessageCorrelation,
        returnedMessage.getReplyText()); // 原因
}
}

```

备份交换机

有了mandatory参数和回退消息，我们获得了对无法投递消息的感知能力，有机会在生产者的消息无法被投递时发现并处理。但有时候，我们并不知道该如何处理这些无法路由的消息，最多打个日志，然后触发警报手动处理。而通过日子来处理这些无法路由的消息不是很优雅的做法，特别是当生产者所在的服务器有多台的时候，手动复制日子更加麻烦且容易出错。而且设置mandatory参数会增加生产者的复杂性，需要添加处理这些被退回的消息的逻辑。如果即不想丢失消息，又不想增加生产者的复杂性该如何处理？

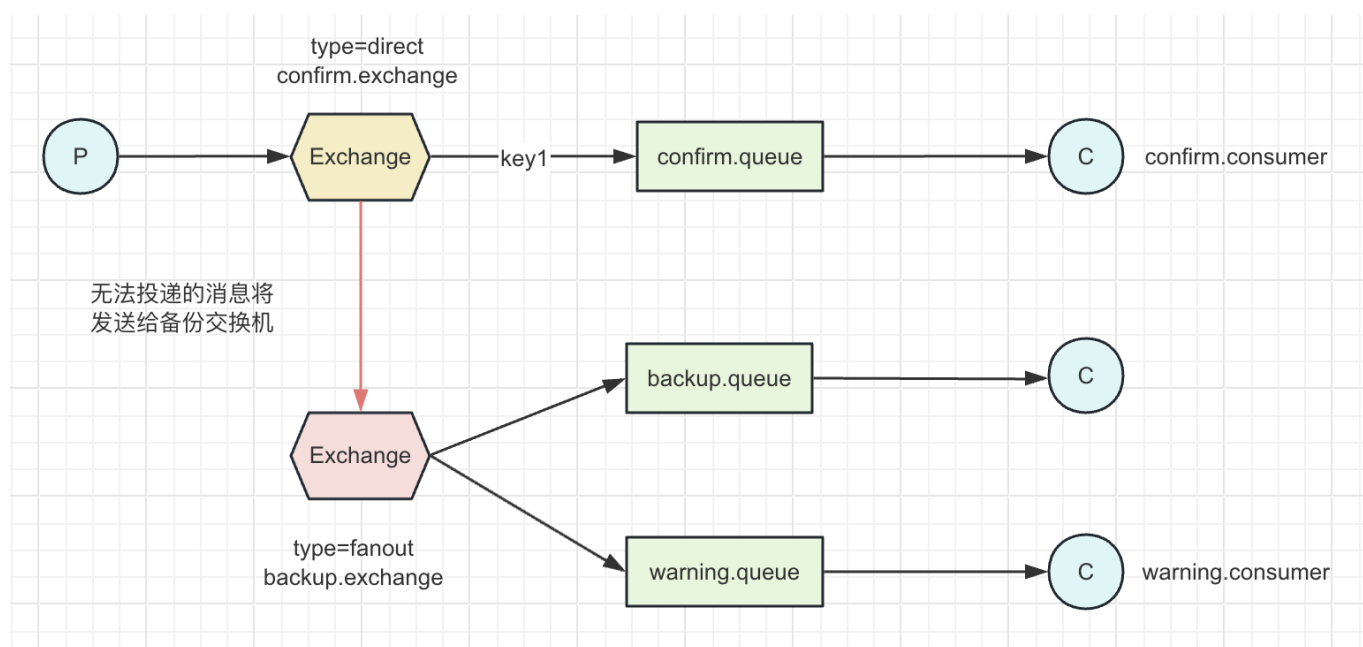
前面在设置死信队列的课程中有提到，可以为队列设置死信交换机来保存消息。在Rabbit MQ中，有一种备份交换机的机制存在，可以很好的应对这个问题。

备份交换机

备份交换机可以理解为RabbitMQ交换机中的“备胎”，当我们为某一个交换机声明一个对应的备份交换机时，就是为它创建一个备胎，但交换机接受到一条不可路由的消息时，将会把这条消息转发到备份交换机中，由备份交换机来进行转发和处理，通常备份交换机的类型为 `Fanout`，这样就能把所有消息都投递到与其绑定的队列中，然后我们在备份交换机下绑定一个队列，这样所有那些原交换机无法被路由的消息，就会都进入这个队列了。当然还可以建立一个报警队列，用独立的消费者来进行检测和报警。

代码架构图

图: `img/mq24-backupexchange.png`



配置绑定关系代码

在上述原来的 `ConfirmConfig` 基础上进行修改

修改原来交换机的策略

```

/**
 * 发布确认交换机
 * @return 发布确认交换机
 */
@Bean
public DirectExchange confirmExchange() {
    // 由于需要做交换机备份，因此需要改造
    // 如果使用上面的new DirectExchange(CONFIRM_EXCHANGE_NAME)，那么durable默认为
    true，因此此处需要声明
    // withArgument(key, value)为单个键值对，withArguments(map)为map存储多个键值对
    return ExchangeBuilder.directExchange(CONFIRM_EXCHANGE_NAME)
        .durable(true)
        .withArgument("alternate-exchange", BACKUP_EXCHANGE_NAME) //
alternate-exchange 备份交换机
        .build();
}

```

交换机与队列绑定关系

```

/**
 * 备份交换机
 * 以Fanout扇出形式将消息投递给backupQueue和warningQueue
 * @return 备份交换机
 */
@Bean
public FanoutExchange backupExchange() {
    return new FanoutExchange(BACKUP_EXCHANGE_NAME);
}
/**
 * 备份队列
 * @return 接收备份交换机传递的消息
 */
@Bean
public Queue backupQueue() {
    return new Queue(BACKUP_QUEUE_NAME);
}
/**
 * 报警队列
 * @return 接收备份交换机传递的消息
 */
@Bean
public Queue warningQueue() {
    return new Queue(BACKUP_WARNING_QUEUE_NAME);
}
/**
 * 绑定备份队列与备份交换机的关系
 * @param backupQueue 备份队列
 * @param backupExchange 备份交换机
 * @return 绑定关系
 */
@Bean
Binding backupQueueBindingBackupExchange(Queue backupQueue,
                                           FanoutExchange backupExchange) {
    return BindingBuilder.bind(backupQueue).to(backupExchange);
}
/**
 * 绑定报警队列与备份交换机的关系
 * @param warningQueue 报警队列
 * @param backupExchange 备份交换机
 * @return 绑定关系
 */
@Bean
Binding warningQueueBindingBackupExchange(Queue warningQueue,
                                           FanoutExchange backupExchange) {
    return BindingBuilder.bind(warningQueue).to(backupExchange);
}

```

消费者

所在位置 (consumer/WarningConsumer.java)

```
package org.example.consumer;

import com.alibaba.fastjson.JSONObject;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

import java.nio.charset.StandardCharsets;

import static org.example.util.CommonDiction.BACKUP_WARNING_QUEUE_NAME;

@Component
@Slf4j
public class WarningConsumer {
    @RabbitListener(queues = BACKUP_WARNING_QUEUE_NAME)
    public void receivedBackupMsg(Message message) {
        log.warn("[Warn] Warning! The message is not routing! Message : {}; the detailed message is {}",
            new String(message.getBody(), StandardCharsets.UTF_8),
            JSONObject.toJSONString(message));
    }
}
```

所在位置 (consumer/BackupConsumer.java)

```

package org.example.consumer;

import com.alibaba.fastjson.JSONObject;
import lombok.extern.slf4j.Slf4j;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

import java.nio.charset.StandardCharsets;

import static org.example.util.CommonDictioN.BACKUP_QUEUE_NAME;

@Component
@Slf4j
public class BackupConsumer {
    @RabbitListener(queues = BACKUP_QUEUE_NAME)
    public void receivedBackupMsg(Message message) {
        log.error("[Backup] Backup success! The message is not routing! Message
: {}; the detailed message is {}",
            new String(message.getBody(), StandardCharsets.UTF_8),
            JSONObject.toJSONString(message));
    }
}

```

测试结果

- 当模拟交换机失效时，交换机不会将消息转发到备份交换机
- 当交换机成功接收且消息未被路由时，消息被备份交换机截取，并且广播至 BackupConsumer 与 WarningConsumer 当中；并且原本开启的 RabbitTemplate.returnedMessage 并未被执行

```
INFO : [S]Send message success! The message is : [测试信息1]
ERROR : [Backup] Backup success! The message is not routing! Message : 测试信息1; the detailed message is {"body":"5rWL6K+V5L+h5oGvMQ==","messageProperties":{"consumerQueue":"backup.queue","consumerTag":"amq.ctag-TxVoEUXCfm-Sr2WkgS4vdg","contentLength":0,"contentType":"application/octet-stream","deliveryTag":2,"finalRetryForMessageWithNoId":false,"headers":{"spring_listener_return_correlation":"19f22d18-a703-4b8b-8352-ee07f8dd5107","spring_returned_message_correlation":"3c5fcdf0-5621-4e84-8ff7-1bc5f3371ffa"},"lastInBatch":false,"priority":0,"projectionUsed":false,"publishSequenceNumber":0,"receivedDeliveryMode":"PERSISTENT","receivedExchange":"confirm.exchange","receivedRoutingKey":"confirm.key123","redelivered":false}}
WARN : [Warn] Warning! The message is not routing! Message : 测试信息1; the detailed message is {"body":"5rWL6K+V5L+h5oGvMQ==","messageProperties":{"consumerQueue":"warning.queue","consumerTag":"amq.ctag-9Cb0wvxolCXT0HRH1AKLIA","contentLength":0,"contentType":"application/octet-stream","deliveryTag":2,"finalRetryForMessageWithNoId":false,"headers":{"spring_listener_return_correlation":"19f22d18-a703-4b8b-8352-ee07f8dd5107","spring_returned_message_correlation":"3c5fcdf0-5621-4e84-8ff7-1bc5f3371ffa"},"lastInBatch":false,"priority":0,"projectionUsed":false,"publishSequenceNumber":0,"receivedDeliveryMode":"PERSISTENT","receivedExchange":"confirm.exchange","receivedRoutingKey":"confirm.key123","redelivered":false}}
INFO : [Ex]Exchange is received success, the message id is [3c5fcdf0-5621-4e84-8ff7-1bc5f3371ffa], and correlationData is [{"future":{"cancelled":false,"done":true},"id":"3c5fcdf0-5621-4e84-8ff7-1bc5f3371ffa"}]
```

当 mandatory 参数与备份交换机一起使用时，两者同时开启，备份交换机优先级较高，也就是以备份交换机为准，当不可到达时，不再打印 returnedMessage（回退消息）的实现，即未被执行。

RabbitMQ 其他知识点

幂等性

概念

用户对于同一操作做发起的一次或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。举个最简单的例子——支付——用户购买商品后支付，支付扣款成功，但是返回结果的时候网络异常，此时已经扣款，用户再次点击按钮就会二次扣款，返回结果成功，用户查询余额发现多扣款了，流水记录也变成了两条。在以前的单应用系统中，只需要把数据操作放入事务中即可，发生错误立即回滚，但是在响应客户端的时候也有可能出现网络中断或者异常等。

消息重复消费

消费者在消费 MQ 中的消息时，MQ 已经把消息发送给消费者，消费者在给 MQ 返回 ack 时网络中断，故 MQ 未接收到确认信息，该条消息会重新发给其他消费者，或者在网络重连后再次发送给消费者，但实际上该消费者已经成功消费了这条消息，造成消费者消费了重复的消息。

解决思路

- MQ 消费者的幂等性的解决一般使用全局 ID 或者写个唯一标识，比如时间戳、UUID
- 订单消费者消费 MQ 中的消息也可利用 MQ 的该 id 来判断
- 可以按自己的规则生成一个全局唯一 id，每次消费消息时用该 id 先判断该消息是否已经消费过

消费端的幂等性保障

在海量订单生成的业务高峰期，生产端可能会重复发生了消息，这时候消费端要实现幂等性，就意味着消息永远不会被消费多次，即使接收到一样的消息。业界主流的幂等性操作有两种：

- 唯一 ID+指纹码机制，利用数据库主键去重
- 利用 Redis 的原子性实现

唯一 ID+指纹码机制

指纹码：我们的一些规则或者时间戳加别的服务给到的唯一信息码，它并不一定是系统生成的，基本都是由业务规则拼接而来，但是一定要保证唯一性，然后利用查询语句判断这个 id 是否存在数据库中，优势就是实现简单就一个拼接，然后查询判断是否重复；劣势就是在高并发时，如果是单个数据库就会有写入性能瓶颈，当然也可以采用分库分表提升性能，但是不是最推荐的方式。

Redis 原子性

利用 Redis 执行 `setnx` 命令，天然具有幂等性，从而实现不重复消费。

优先级队列

使用场景

在系统中有一个“订单催付”的场景，客户在天猫下的订单，淘宝会及时将订单推送给我们，如果用户在设定的时间内未付款那么就会给用户推送一条短信提醒。

但是商家我们需要进行排列优先级，例如 Apple、小米这种一年能创造很大利润的商家理所当然订单要进行优先级处理，而曾经我们的后端是使用 Redis 来存放的定时轮询，但是 Redis 只能用

List 做一个简单的消息队列，并不能实现优先级场景，所以订单量大了后采用 RabbitMQ 进行改造和优化，如果发现时大客户的订单给一个相对较高的优先级，否则就是默认优先级。

优先级队列区间

优先级队列的优先级设定区间为 0-255，越大越优先被消费。

tips: 优先级的数值越大，对电脑性能损耗越高。

优先级队列的使用

在队列设置优先级

在 RabbitMQ 的管理界面创建

在 Add a new queue 模块下，Arguments 参数设定 `x-max-priority = [0, 255]` 即可

在代码中创建

其实是在创建队列时，arguments 的 map 中加入优先级队列设置参数，创建成功后可以在界面看到 Pri 标识

```
@Bean
public Queue priorityQueue() {
    Map<String, Object> arguments = new HashMap<>();
    arguments.put("x-max-priority", Integer.valueOf(5));
    return QueueBuilder.durable(PRIORITY_QUEUE_NAME)
        .withArguments(arguments)
        .build();
}
```

在消息设置优先级

要让队列实现优先级需要做的除了队列需要设置优先级还需要为消息设置优先级，消费者需要等待消息已经发送到队列中才去消费，这样才会有机会对消息进行排序。

在Spring AMQP中，可以通过 RabbitTemplate 类发送携带 BasicProperties 的消息。

BasicProperties 是一个提供丰富消息属性设置的类，包括消息的contentType、headers、priority、correlationId等。

在原生 Java 中

```
AMQP.BasicProperties properties = new AMQP.BasicProperties.builder()  
    .priority(5) // 优先级为 5  
    .build();  
channel.basicPublish("", QUEUE_NAME, properties,  
message.getBytes(StandardCharsets.UTF_8));
```

在 Spring AMQP 中

其实实质就是调用 `RabbitTemplate.send()` 方法，并且将配置好的参数写入到 `new Message()` 当中

```
// 设置配置类  
MessageProperties messageProperties = new MessageProperties();  
// 配置生产者发送消息优先级  
messageProperties.setPriority(priority);  
rabbitTemplate.send(exchange, routingKey, new  
Message(message.getBytes(StandardCharsets.UTF_8), messageProperties));
```

生产者

所在位置 (controller/OtherController.java)

```

/**
 * 测试优先级队列
 */
@GetMapping("testPriority")
public void testPrioritySend() {
    for (int i = 0; i <= 15; i++) {
        String message = "info" + i;
        // 添加消息配置
        MessageProperties messageProperties = new MessageProperties();
        // 设置优先级
        messageProperties.setPriority(5);
        if (i % 5 == 0) {
            log.info("[S-P] Send message is success , the message is {}, and this is priority!", i);
            // 使用 rabbitTemplate.send可以承接上面设置好优先级的参数
            rabbitTemplate.send("", PRIORITY_QUEUE_NAME,
                new Message(message.getBytes(StandardCharsets.UTF_8),
                    messageProperties));
        } else {
            log.info("[S-N] Send message is success , the message is {}", i);
            rabbitTemplate.convertAndSend(PRIORITY_QUEUE_NAME,
                message.getBytes(StandardCharsets.UTF_8));
        }
    }
}

```

惰性队列

使用场景

RabbitMQ 从 3.6.0 版本开始引入了惰性队列的概念。惰性队列会尽可能的将消息存入磁盘中，而在消费者消费到相应的消息时才会被加载到内存中，它的一个重要的设计目标是能够支持更长的队列，即支持更多的消息存储。当消费者由于各种各样的原因（比如消费者下线、宕机亦或是由于维护而关闭等）而致使长时间内不能消费消息造成堆积时，惰性队列就很有必要。

默认情况下，当生产者将消息发送到 RabbitMQ 的时候，队列中的消息会尽可能的存储在内存之中，这样可以更快速的将消息发送给消费者。即使是持久化的消息，在被写入磁盘的同时也会在内存中驻留一份备份。当 RabbitMQ 需要释放内存的时候，会将内存中的消息换页至磁盘中，这个操作会耗费较长的时间，也会阻塞队列的操作，进而无法接受新的消息。虽然 RabbitMQ 的开发者们一直在升级相关的算法，但是始终效果不太理想，尤其是在消费信息量特别大的时候。

两种模式

队列具备两种模式：

- default
- lazy

默认为 default 模式，在 3.6.0 之前的版本无需做任何变更。layz 模式即为惰性队列模式

- 通过声明队列时使用 arguments 参数声明
- 通过 Policy 的方式设置

如果一个队列同时拥有两个方式设置，那么 Policy 具备更高的优先级。如果要通过声明的方式改变已有队列模式，那么只能先删除队列，然后重新声明一个新的队列。

在声明队列时，可以通过 x-queue-mode 参数设置队列的模式，取值为 default 和 lazy 。

声明实例

```
Map<String, Object> arguments = new HashMap<>();
// 声明惰性队列
arguments.put("x-queue-mode", "lazy");
return QueueBuilder // 构建一个队列
    .durable(Queue.A) // 队列名称
    .withArguments(arguments) // 相当于Java声明中的参数map
    .build();
```

队列类型	内存消耗	消费速度
default	大（消息直接在内存读写）	快（直接读取内存）
lazy	小（因为只存放索引）	慢（还需要通过内存的索引去读取磁盘上的数据）

RabbitMQ集群

clustering

使用集群的原因

前面的学习中，都是 Rabbit MQ 单机服务，无法满足真实应用需求。如果 RabbitMQ 服务器遇到内存崩溃、机器掉电或者主板故障等情况；单台 RabbitMQ 服务器可以满足每秒 1000 条消息的吞吐量，如果需要满足每秒 10 万条消息吞吐量等极端问题需要如何处理？

购买昂贵的服务器来增强单机 RabbitMQ 服务的性能显得捉襟见肘，搭建一个 RabbitMQ 集群才是解决实际问题的关键。

搭建步骤

