# RubbitMQ

## Docker搭建集群环境

创建容器

前提：集群节点之间需要通讯，需要本地创建hosts文件描述ip和主机名映射，映射到容器的hosts文件上

主节点：docker run -d --net host --hostname rabbit1 --name rabbit1 -v /home/docker/RabbitMQ/hosts:/etc/hosts -v /home/docker/RabbitMQ/data:/var/lib/rabbitmq/mnesia -e RABBITMQ\_ERLANG\_COOKIE='rabbitcookie' --restart=always --privileged=true rabbitmq:3.7.7-management

从节点：docker run -d --net host --hostname rabbit2 --name rabbit2 -v /home/docker/rabbitmq/hosts:/etc/hosts -v /home/docker/rabbitmq/data:/var/lib/rabbitmq/mnesia -e RABBITMQ\_ERLANG\_COOKIE='rabbitcookie' --restart=always --privileged=true rabbitmq:3.7.7-management

加入集群

docker exec -it myrabbit2 bash # 进入镜像命令行

rabbitmqctl stop\_app

rabbitmqctl reset

rabbitmqctl join\_cluster --ram rabbit@rabbit1 #rabbit1为主节点的hostname

rabbitmqctl start\_app

启动rabbitmqrabbitmqctl start\_app

停止rabbitmqrabbitmqctl stop\_app

rabbitmq 查看所有队列信息rabbitmqctl list\_queues

还原 rabbitmq 清除所有队列rabbitmqctl reset

rabbitmq 查看用户rabbitmqctl list\_users

rabbitmq添加用户rabbitmqctl add\_user root root

rabbitmq设置权限rabbitmqctl set\_permissions -p / root ".\*" ".\*" ".\*"

Rabbitmq 设置管理员rabbitmqctl set\_user\_tags djs administrator

rabbitmq查看集群状态信息rabbitmqctl cluster\_status

移除某个集群节点：一般情况当节点故障时，在正常的节点中，移除该故障节点rabbitmqctl -n rabbit@rabbit1 forget\_cluster\_node rabbit@rabbit3

镜像队列：

会在集群内部同步数据耗费性能

给集群任意节点增加策略，生产者连接节点的数据会同步到其他节点

rabbitmqctl set\_policy ha-all "^" '{"ha-mode":"all"}'

rabbitmqctl set\_policy -p hrsystem ha-allqueue "^message" '{"ha-mode":"all"}'注意："^message" 这个规则要根据自己修改，这个是指同步"message"开头的队列名称，我们配置时使用的应用于所有队列，所以表达式为"^"

## 作用

使得web服务组件间的同步调用变为异步调用（1.流量消峰 2.系统解耦（目标应用挂掉消息也不会丢失））

## 基本概念

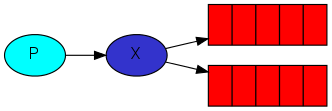
### **ConnectionFactory、Connection、Channel**

ConnectionFactory、Connection、Channel都是RabbitMQ对外提供的API中最基本的对象。Connection是RabbitMQ的socket链接，它封装了socket协议相关部分逻辑。ConnectionFactory为Connection的制造工厂。  
Channel是我们与RabbitMQ打交道的最重要的一个接口，我们大部分的业务操作是在Channel这个接口中完成的，包括定义Queue、定义Exchange、绑定Queue与Exchange、发布消息等。

### **Exchange**

生产者将消息投递到Queue中，实际上这在RabbitMQ中这种事情永远都不会发生。实际的情况是，生产者将消息发送到Exchange（交换器，下图中的X），由Exchange将消息根据对应的路由策略匹配routing key路由到一个或多个Queue中（或者未匹配到路由key时丢弃）。

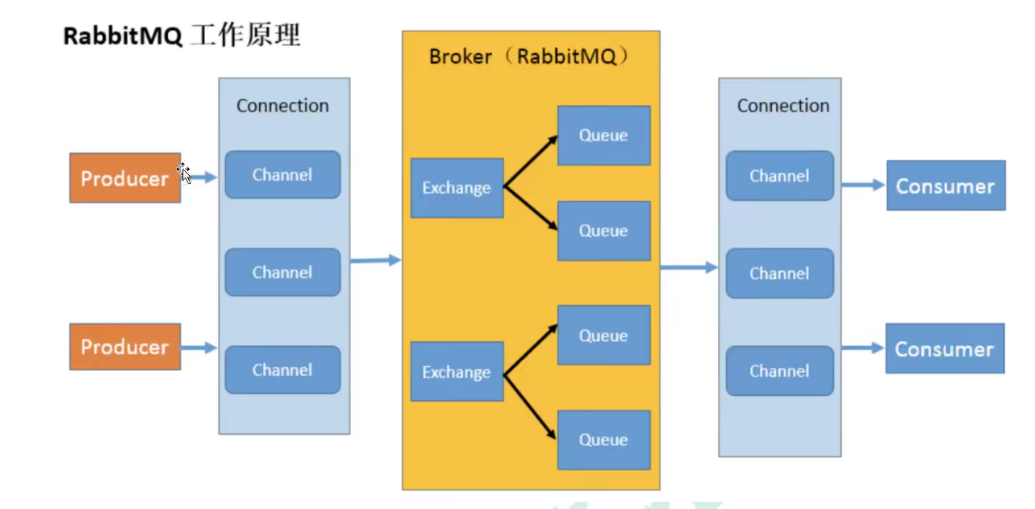
并且交换机不光可以绑定队列，也可以绑定交换机。



### Queue：

存储消息的容器

## 工作原理



一个队列中的消息只能一个消费者消费，但是可以把一个消息通过交换机分发到不同的队列中。

## 队列持久化：

在声明队列时设置持久化，可防止服务器宕机时丢列丢失（队列持久化不代表消息持久化）

代码实现：

设置第二个参数是否持久化为true

channel.queueDeclare("myQueue", true, false, false,null);

## 消息持久化：

发布消息时可设置消息持久化设置，重启服务消息不回丢失

代码实现：

设置：MessageProperties.PERSISTENT\_TEXT\_PLAIN

channel.basicPublish("", "myQueue", MessageProperties.PERSISTENT\_TEXT\_PLAIN, message.getBytes());

## 消息拉取和推送：

消息拉取实现：

消费者主动获取消息，只能获取一次消息

channel.basicGet("myQueue", false)

消息推送实现：

消息中间件服务器主动推送消息，采取异步回调方式，可以无限推送

channel.basicConsume("myQueue", false, deliverCallback, cancelCallback);

## 消息消费分发模式：

默认情况：当一个队列有多个消费者时，默认情况下消息为轮询分发，即使消费能力差别比较大每个消费者一次给一个，导致消费能力强的消费者会空等。（此种消费策略不合理），在消费者端设置不公平分发，能者多劳。

消息预取值：可以给信道设置预取值（前提是消息必须是手动确认模式才能生效），此值得含义为：服务器最多为设置的信道推送不多于此值得消息，在消息未被确认前，会导致处理快的消费者消费更多消息，并且是非公平的（处理快的消费者分发的多）。

代码实现：

channel.basicQos(5);

## 发布确认：

消息生产者可以保证消息发布不丢失

1. 单个确认

每发一次消息，则阻塞的去等待rocker返回确认消息，根据返回值是否成功进行下一步操作

代码实现：

批量确认//开启消息发布确认  
channel.confirmSelect();

//确认消息发送是否成功：阻塞的去等待broker返回,可以增加时间参数，

超时则抛出异常  
boolean sendSuccess = channel.waitForConfirms(3000);

和单个确认类似，只不过生产者计算固定数量后通过消息确认api去确认（此方式不可靠，消息发送失败无法知道那些失败）

1. 异步确认

此方式给信道增加消息确认监听器，发送消息时将消息编号（channel.getNextPublishSeqNo()

获取编号）和消息保存到map中，rocker异步的去调用我们的监听器去告诉生产者消息发布的结果，并且传入消息编号，我们通过编号去获取失败消息，再次重发。

也可以增加投递队列失败时消息回退监听器，交换机获取到消息后未成功投递到队列中，则会调用。

代码实现：

//开启消息发布确认  
channel.confirmSelect();

*/\*\*  
 \* 交换机获取到消息后未送达队列时调用  
 \*/*channel.addReturnListener(new ReturnCallback()  
{  
 @Override  
 public void handle(Return aReturn)  
 {  
 System.*out*.println("消息未被送达队列");  
 }  
});  
//异步消息发布确认监听器(交换机是否获取到消息都会调用)  
channel.addConfirmListener(new ConfirmListener()  
{  
 @Override  
 public void handleAck(long deliveryTag, boolean b) throws IOException  
 {  
 //删除消息缓存  
 messageCache.remove(deliveryTag);  
 //l为消息标识，可以确定那个消息成功那个失败  
 System.*out*.println(String.*format*("交换机获取到了消息,消息标识：%s", deliveryTag));  
  
 }  
  
 @Override  
 public void handleNack(long deliveryTag, boolean b) throws IOException  
 {  
 //删除消息缓存  
 messageCache.put(channel.getNextPublishSeqNo(), messageCache.get(deliveryTag));  
 messageCache.remove(deliveryTag);  
 //重新发送消息  
 channel.basicPublish("", "myQueue", MessageProperties.*PERSISTENT\_TEXT\_PLAIN*, messageCache.get(deliveryTag).getBytes());  
 System.*out*.println(String.*format*("消息发布失败,已重发，消息标识：%s", deliveryTag));  
 }  
});

## 消费确认：

同一个消息只能消费一次，当消费者确认消息后rocker才会删除掉队列中的消息。默认为自动确认。手动确认时需要设置确认模式为手动确认。

代码实现：

//第2个参数为设置消息手动确认

channel.basicConsume("myQueue", false, deliverCallback, cancelCallback);

//手动确认消息

channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);

## 六种模式：

### 简单模式：

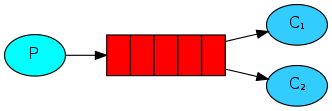
一个生产者一个消费者

应用场景：

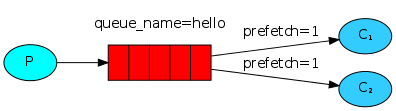
邮件、手机短信这种一对一关系的业务。

### 工作模式：

如下图：当多个消费者订阅了同一个队列，则服务器会将队列中的消息依次轮流分发给消费者（一个消息只发给一个消费者，等待消费者确认消息后才会把队列中对应的消息删除）



前面我们讲到如果有多个消费者同时订阅同一个Queue中的消息，Queue中的消息会被平摊给多个消费者。这时如果每个消息的处理时间不同，就有可能会导致某些消费者一直在忙，而另外一些消费者很快就处理完手头工作并一直空闲的情况。我们可以通过设置prefetchCount来限制Queue每次发送给每个消费者的消息数，比如我们设置prefetchCount=1，则Queue每次给每个消费者发送一条消息；消费者处理完这条消息后Queue会再给该消费者发送一条消息。

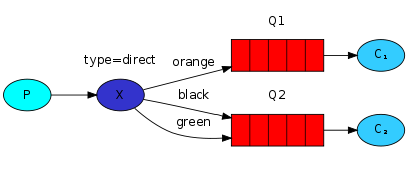


应用场景：

抢红包、资源分配等业务

### 路由模式（direct）：

direct类型的Exchange路由规则也很简单，它会把消息路由到那些binding key与routing key完全匹配的Queue中。



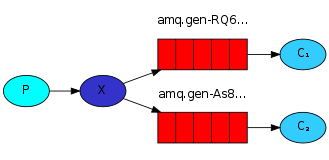
代码实现：

channel.exchangeDeclare(exchangeName,BuiltinExchangeType.*DIRECT*);  
//队列绑定交换机并设置路由key  
channel.queueBind(queueName1,exchangeName,"q1");  
channel.queueBind(queueName2,exchangeName,"q2");

应用场景：

短信、聊天工具等

### 广播模式（fanout）

它会把所有发送到该Exchange的消息路由到所有与它绑定的Queue中，不计算路由。  


 代码实现：

channel.exchangeDeclare(exchangeName,BuiltinExchangeType.*DIRECT*);  
//队列绑定交换机并设置路由key  
channel.queueBind(queueName1,exchangeName,"q1");  
channel.queueBind(queueName2,exchangeName,"q2");

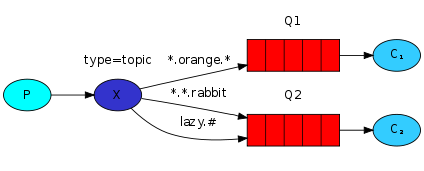
应用场景：

广播

### 主题模式（topic）

将消息路由到binding key与routing key相匹配的Queue中，但这里的匹配规则有些不同，它约定：

* routing key为一个句点号“. ”分隔的字符串（我们将被句点号“. ”分隔开的每一段独立的字符串称为一个单词），如“stock.usd.nyse”
* binding key中可以存在两种特殊字符“\*”与“#”，用于做模糊匹配，其中“\*”用于匹配一个单词，“#”用于匹配多个单词（可以是零个）



代码实现：

*/\*\*  
 \* 声明交换机  
 \* 参数1：交换机名称  
 \* 参数2：路由策略  
 \*/*channel.exchangeDeclare(exchangeName,BuiltinExchangeType.*TOPIC*);  
//队列绑定交换机  
channel.queueBind(queueName1,exchangeName,"陕西.渭南");  
channel.queueBind(queueName2,exchangeName,"陕西.#");  
channel.queueBind(queueName3,exchangeName,"甘肃.#");  
channel.queueBind(queueName4,exchangeName,"#");

应用场景：

做物流分拣的多级传递

### **headers**

headers类型的Exchange不依赖于routing key与binding key的匹配规则来路由消息，而是根据发送的消息内容中的headers属性进行匹配。  
在绑定Queue与Exchange时指定一组键值对；当消息发送到Exchange时，RabbitMQ会取到该消息的headers（也是一个键值对的形式），对比其中的键值对是否完全匹配Queue与Exchange绑定时指定的键值对；如果完全匹配则消息会路由到该Queue，否则不会路由到该Queue。

代码实现：

Map<String,Object> map1 = new HashMap<>();  
map1.put("k1","v1");  
map1.put("k2","v2");  
Map<String,Object> map2 = new HashMap<>();  
map1.put("kk1","vv1");  
map1.put("kk2","vv2");

channel.exchangeDeclare(exchangeName,BuiltinExchangeType.*HEADERS*);  
//队列绑定交换机  
channel.queueBind(queueName1,exchangeName,"",map1);  
channel.queueBind(queueName2,exchangeName,"",map2);

//消息发送

AMQP.BasicProperties properties= MessageProperties.*PERSISTENT\_TEXT\_PLAIN*.builder().headers(map).build();  
channel.basicPublish(exchangeName, routingKey, properties, message.getBytes());

## 死信队列：

死信队列和正常的队列一样只不过被设置为另外一个正常队列的死信队列。当正常队列中的消息未被正常的消费时会将消息通过死信队列的交换机路由到死信队列中。我们可以通过单独的消费者再去处理死信队列中的消息（保证了业务消息处理异常时不丢失）。

死信消息产生原因：

1. 消息超时未被处理（发送消息时设置超时时间或者直接给队列设置过期时间）
2. 消息被消费者拒绝（调用信道的basicReject()方法即可）
3. 队列达到了最大长度无法添加新的数据 （声明队列是参数中需要指定最大队列长度）

代码实现：

在声明正常队列时参数中增加死信队列绑定的交换机即可

Map<String,Object> param = new HashMap<>();//正常队列的参数  
param.put("x-dead-letter-exchange", diedExchangeName);//指定该队列的死信交换机  
channel.queueDeclare(normalQueue, false, false, false, param);

## 延迟队列：

应用场景：

业务操作需要在固定时间后进行处理。

解决方案：

1.可以使用定时任务处理，但是定时任务（但是对失效要求比较高的业务，定时任务不能支持）

1. 使用消息队列的延迟队列进行处理（当业务的消息延迟时间一样时可以使用，不一样时不能使用）。

队列设置消息过期时间，并且设置同一个死信队列

原理：

延迟队列实质上为死信队列，以超时进入死信队列的方式，实现延迟执行的效果。

问题：

队列为先进先出，两条延迟消息前一条延迟时间长，后一条延迟时间久，则只有第一条过期，则第二条才能过期进入死信队列中。

问题解决方案：

安装rabbitmq\_delayed\_message\_exchange插件实现延迟消息，声明自定义交换机，发送消息时设置每个消息的延迟时间。

@Bean  
public CustomExchange delayExchange(){  
 Map<String,Object> argument = new HashMap<>();  
 argument.put("x-delayed-type","direct");  
 return new CustomExchange("delayExchange","x-delayed-message",false,true,argument);

## 备份交换机：

当交换机未能成功把消息投递给队列时会直接把消息推送给备份交换机，我们可以给备份交换机绑定相应的队列，在指定消费者，对此类消息进行处理。

代码实现：

Map<String,Object> arg = new HashMap<>();  
arg.put("alternate-exchange","backupExchange");//备份交换机   
channel.exchangeDeclare(exchangeName,BuiltinExchangeType.*DIRECT*);

## 使用rabbitmq时常见问题：

1. 消息被重复消费

原因：消费者消费完消息后给与mq服务ack消费确认消息，但是此消息在发送过程中，由于网络问题未成功送达mq服务器，导致消息未被删除，再次被消费。

解决办法：

1.可以在消费者数据库中保存已消费消息的唯一id（可以发送消息时在消息中自定义，或者使用mq自带的消息唯一id），拿到消息后访问数据库判断是否已经消费过。

2.使用redis的setnx()根据返回值轻松解决问题

## 优先级队列：

应用场景：订单催付（大客户的催付消息优先处理）

代码实现：

argument.put("x-max-priority",10);//创建队列时增加优先级参数最大为255，必须是所有要排序的消息还没有被消费才可以排序

amqpTemplate.convertAndSend("priorityExchange","priority-routing-key","消息"+i, msg -> {msg.getMessageProperties().setPriority(5);return msg;});

//发送消息时设置优先级参数，参数越大优先级越高

## 惰性队列：

一般队列的消息会在设置持久化durable参数时会在内存和磁盘中各保存一份。惰性队列会将消息放在磁盘上，在内存中保存一份索引，减少内存的消耗，但是因为要从内存中加载因此效率低。

代码实现：

队列声明时增加参数即可

argument.put("x-queue-mode","lazy"); //惰性队列

## 集群：

集群有普通集群和镜像队列集群两种模式

### 普通集群：

没有设置消息同步策略，数据只在单个节点存在，节点之间不存在主从关系。消费者访问的节点没有数据则从对应队列保存的节点拉取数据。

优点：高吞吐率

缺点：集群非高可用（消息只保存了一个节点，一旦宕机数据无法使用）

### 镜像队列集群：

设置消息同步策略，节点存在主从关系。为了综合下方的优缺点可以在设置策略时仅将可靠性要求高的消息队列进行同步。

优点：高可用

缺点：不能提高吞吐率，并且数据需要同步影响性能