

# RabbitMQ详解

--- 老孙

## 课程目标

- 1、什么是RabbitMQ
- 2、怎么用RabbitMQ

## 1.什么是RabbitMQ

### 1.1 MQ ( Message Queue ) 消息队列

- 消息队列中间件，是分布式系统中的重要组件
- 主要解决，异步处理，应用解耦，流量削峰等问题
- 从而实现高性能，高可用，可伸缩和最终一致性的架构
- 使用较多的消息队列产品：RabbitMQ，RocketMQ，ActiveMQ，ZeroMQ，Kafka等

#### 1.1.1 异步处理

- 用户注册后，需要发送验证邮箱和手机验证码；
- 将注册信息写入数据库，发送验证邮件，发送手机，三个步骤全部完成后，返回给客户端



#### 1.1.2 应用解耦

- 场景：订单系统需要通知库存系统
- 如果库存系统异常，则订单调用库存失败，导致下单失败
  - 原因：订单系统和库存系统耦合度太高



- 订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户，下单成功；

- 库存系统：订阅下单的消息，获取下单信息，库存系统根据下单信息，再进行库存操作；
- 假如：下单的时候，库存系统不能正常运行，也不会影响下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了，实现了订单系统和库存系统的应用解耦；
- 所以说，消息队列是典型的：**生产者消费者模型**
- 生产者不断的向消息队列中生产消息，消费者不断的从队列中获取消息
- 因为消息的生产和消费都是异步的，而且只关心消息的发送和接收，没有业务逻辑的入侵，这样就实现了生产者和消费者的解耦

### 1.1.3 流量削峰

- 抢购，秒杀等业务，针对高并发的场景
- 因为流量过大，暴增会导致应用挂掉，为解决这个问题，在前端加入消息队列



- 用户的请求，服务器接收后，首先写入消息队列，如果超过队列的长度，就抛弃，甩一个秒杀结束的页面！
- 说白了，秒杀成功的就是进入队列的用户；

## 1.2 背景知识介绍

### 1.2.1 AMQP高级消息队列协议

- 即**A**dvanced **M**essage **Q**ueuing **P**rotocol，一个提供统一消息服务的应用层标准高级消息队列协议
- 协议：数据在传输的过程中必须要遵守的规则
- 基于此协议的客户端可以与消息中间件传递消息
- 并不受产品、开发语言等条件的限制

### 1.2.2 JMS

- **J**ava **M**essage **S**erver，Java消息服务应用程序接口，一种规范，和JDBC担任的角色类似
- 是一个Java平台中关于面向消息中间件的API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信

### 1.2.3 二者的联系

- JMS是定义了统一接口，**统一消息操作**；AMQP通过协议**统一数据交互格式**
- JMS必须是java语言；AMQP只是协议，与语言无关

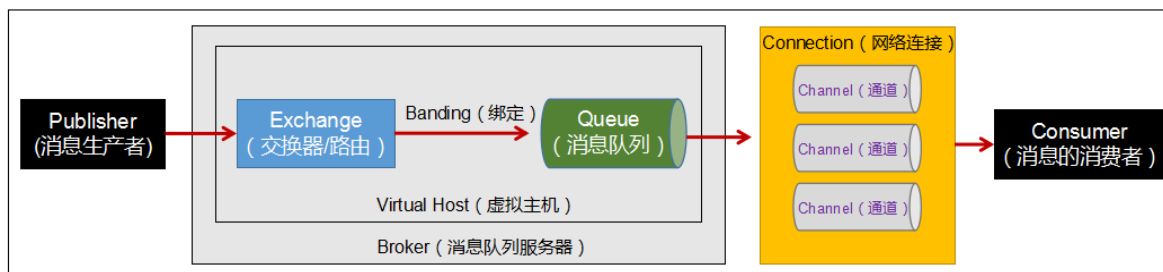
### 1.2.4 Erlang语言

- Erlang ( ['ə:læŋ] ) 是一种通用的面向**并发**的编程语言，它由瑞典电信设备制造商爱立信所辖的CS-Lab开发，目的是创造一种可以应对大规模并发活动的编程语言和运行环境
- 最初是由爱立信专门为**通信应用**设计的，比如控制交换机或者变换协议等，因此非常适合构建分布式，实时软并行计算系统
- Erlang运行时环境是一个**虚拟机**，有点像Java的虚拟机，这样代码一经编译，同样可以随处运行

## 1.3 为什么选择RabbitMQ

- 我们开篇说消息队列产品那么多，为什么偏偏选择RabbitMQ呢？
- 先看命名：兔子行动非常迅速而且繁殖起来也非常疯狂，所以就把Rabbit用作这个分布式软件的命名（就是这么简单）
- Erlang开发，AMQP的最佳搭档，安装部署简单，上手门槛低
- 企业级消息队列，经过大量实践考验的高可靠，大量成功的应用案例，例如阿里、网易等一线大厂都有使用
- 有强大的WEB管理页面
- 强大的社区支持，为技术进步提供动力
- 支持消息持久化、支持消息确认机制、灵活的任务分发机制等，支持功能非常丰富
- 集群扩展很容易，并且可以通过增加节点实现成倍的性能提升
- **总结：如果你希望使用一个可靠性高、功能强大、易于管理的消息队列系统那么就选择RabbitMQ，如果你想用一个性能高，但偶尔丢点数据不是很在乎可以使用kafka或者zeroMQ**
- kafka和zeroMQ的性能爆表，绝对可以压RabbitMQ一头！

## 1.4 RabbitMQ各组件功能



- **Broker**：消息队列服务器实体
- **Virtual Host**：虚拟主机
  - 标识一批交换机、消息队列和相关对象，形成的整体
  - 虚拟主机是共享相同的身份认证和加密环境的独立服务器域
  - 每个vhost本质上就是一个mini版的RabbitMQ服务器，拥有自己的队列、交换机、绑定和权限机制
  - vhost是AMQP概念的基础，RabbitMQ默认的vhost是 / ，必须在链接时指定
- **Exchange**：交换器（路由）
  - 用来接收生产者发送的消息并将这些消息路由给服务器中的队列
- **Queue**：消息队列
  - 用来保存消息直到发送给消费者。
  - 它是消息的容器，也是消息的终点。
  - 一个消息可投入一个或多个队列。
  - 消息一直在队列里面，等待消费者连接到这个队列将其取走。
- **Banding**：绑定，用于消息队列和交换机之间的关联。
- **Channel**：通道（信道）

- 多路复用连接中的一条独立的双向数据流通道。
- 信道是建立在真实的TCP连接内的 **虚拟链接**
- AMQP命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，都是通过信道完成的
- 因为对于操作系统来说，建立和销毁TCP连接都是非常昂贵的开销，所以引入了信道的概念，用来复用TCP连接。
- **Connection**：网络连接，比如一个TCP连接。
- **Publisher**：消息的生产者，也是一个向交换器发布消息的客户端应用程序。
- **Consumer**：消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。
- **Message**：消息
  - 消息是不具名的，它是由消息头和消息体组成。
  - 消息体是不透明的，而消息头则是由一系列的可选属性组成，这些属性包括routing-key(路由键)、priority(优先级)、delivery-mode(消息可能需要持久性存储[消息的路由模式])等。

## 2.怎么用RabbitMQ

- 想要安装RabbitMQ，必须先安装erlang语言环境，类似安装tomcat，必须先安装JDK
- 查看匹配的版本：<https://www.rabbitmq.com/which-erlang.html>

<a href="#">RabbitMQ version</a>	Minimum required Erlang/OTP	Maximum supported Erlang/OTP	Notes
3.8.6 3.8.5 3.8.4	21.3	23.x	<ul style="list-style-type: none"> <li>• <a href="#">Erlang/OTP 23 compatibility notes</a></li> <li>• Erlang 22.x or 23.x is recommended</li> <li>• Erlang 22.x dropped support for HiPE</li> </ul>
3.8.3 3.8.2 3.8.1 3.8.0	21.3	22.x	<ul style="list-style-type: none"> <li>• Erlang 22.x is recommended.</li> <li>• Erlang 22.x dropped support for HiPE</li> </ul>

## 2.1 RabbitMQ安装启动

erlang下载：<https://dl.bintray.com/rabbitmq-erlang/rpm/erlang>

socat下载：[http://repo.iotti.biz/CentOS/7/x86\\_64/socat-1.7.3.2-5.el7.linux.x86\\_64.rpm](http://repo.iotti.biz/CentOS/7/x86_64/socat-1.7.3.2-5.el7.linux.x86_64.rpm)

RabbitMQ下载：<https://www.rabbitmq.com/install-rpm.html#downloads>

### 2.1.1 安装

```
[root@localhost opt]# rpm -ivh erlang-21.3.8.16-1.el7.x86_64.rpm
```

```
[root@localhost opt]# rpm -ivh socat-1.7.3.2-5.el7.linux.x86_64.rpm
```

```
[root@localhost opt]# rpm -ivh rabbitmq-server-3.8.6-1.el7.noarch.rpm
```

## 2.1.2 启动后台管理插件

```
[root@localhost opt]# rabbitmq-plugins enable rabbitmq_management
```

## 2.1.3 启动RabbitMQ

```
[root@localhost opt]# systemctl start rabbitmq-server.service  
[root@localhost opt]# systemctl status rabbitmq-server.service  
[root@localhost opt]# systemctl restart rabbitmq-server.service  
[root@localhost opt]# systemctl stop rabbitmq-server.service
```

## 2.1.4 查看进程

```
[root@localhost opt]# ps -ef | grep rabbitmq
```

## 2.1.5 测试

1. 关闭防火墙：`systemctl stop firewalld`
2. 浏览器输入：<http://ip:15672>
3. 默认帐号密码：guest，guest用户默认不允许远程连接

1. 创建账号

```
[root@localhost opt]# rabbitmqctl add_user laosun 123456
```

2. 设置用户角色

```
[root@localhost opt]# rabbitmqctl set_user_tags laosun administrator
```

3. 设置用户权限

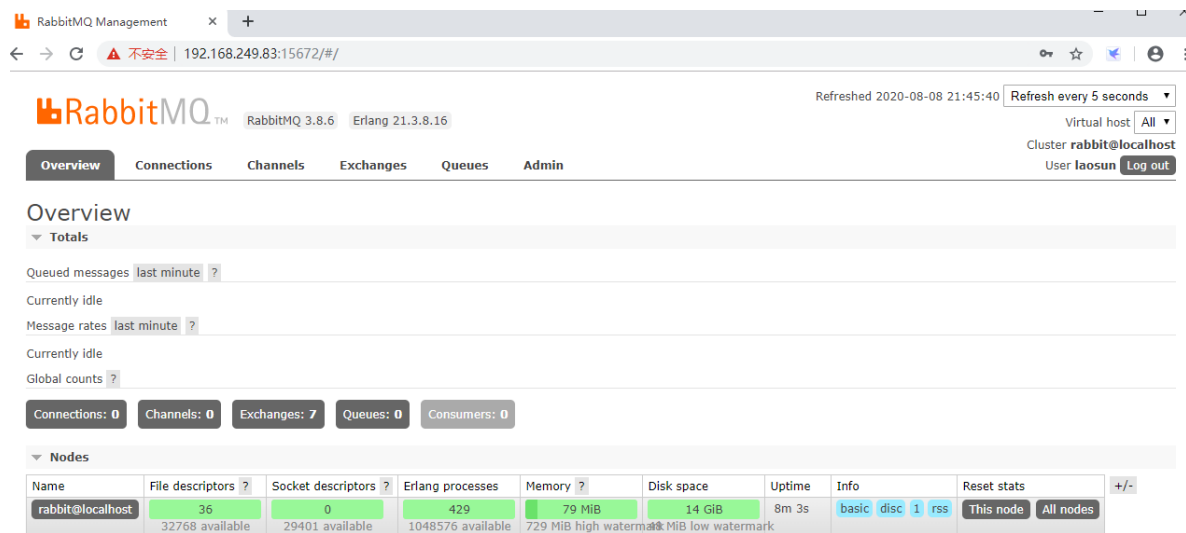
```
[root@localhost opt]# rabbitmqctl set_permissions -p "/" laosun ".*"  
".*" ".*"
```

4. 查看当前用户和角色

```
[root@localhost opt]# rabbitmqctl list_users
```

5. 查看当前用户和角色

```
[root@localhost opt]# rabbitmqctl change_password laosun 123123
```



## 管理界面介绍

- overview：概览
- connections：查看链接情况
- channels：信道（通道）情况
- Exchanges：交换机（路由）情况，默认4类7个
- Queues：消息队列情况
- Admin：管理员列表
- 端口：
  - **5672**：RabbitMQ提供给编程语言客户端链接的端口
  - **15672**：RabbitMQ管理界面的端口
  - **25672**：RabbitMQ集群的端口

## 2.2 RabbitMQ快速入门

### 2.2.1 依赖

```
<dependencies>
  <dependency>
    <groupId>com.rabbitmq</groupId>
    <artifactId>amqp-client</artifactId>
    <version>5.7.3</version>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.25</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.9</version>
  </dependency>
</dependencies>
```

```
</dependencies>
```

## 2.2.2 日志依赖log4j (可选项)

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %m%n

log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.File=rebitmq.log
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %l %m%n

log4j.rootLogger=debug, stdout, file
```

## 2.2.2 创建连接

- 先创建好虚拟主机

The screenshot shows the RabbitMQ Admin interface. At the top, it says 'RabbitMQ 3.8.6 Erlang 21.3.8.16'. The 'Admin' tab is selected. On the right, it shows 'Virtual host: All', 'Cluster: rabbit@localhost', and 'User: laosun'. The main section is 'Virtual Hosts', with a sub-section 'All virtual hosts'. A table lists virtual hosts, with one entry: '/ guest, laosun running'. Below the table, there is a form to 'Add a new virtual host'. The form fields are: Name: '/lagou', Description: '专注于物联网人', and Tags: 'administrator'. A red box highlights the 'Virtual Hosts' tab on the right, and a red arrow points to the 'Add a new virtual host' form.

```
package util;

import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;

/**
 * @BelongsProject: lagou-rabbitmq
 * @Author: GuoAn.Sun
 * @Description: 专门与RabbitMQ获得连接
 */
public class ConnectionUtil {
    public static Connection getConnection() throws Exception{
        //1.创建连接工厂
        ConnectionFactory factory = new ConnectionFactory();
        //2.在工厂对象中设置MQ的连接信息 (ip,port,vhost,username,password)
        factory.setHost("192.168.204.141");
        factory.setPort(5672);
    }
}
```

```

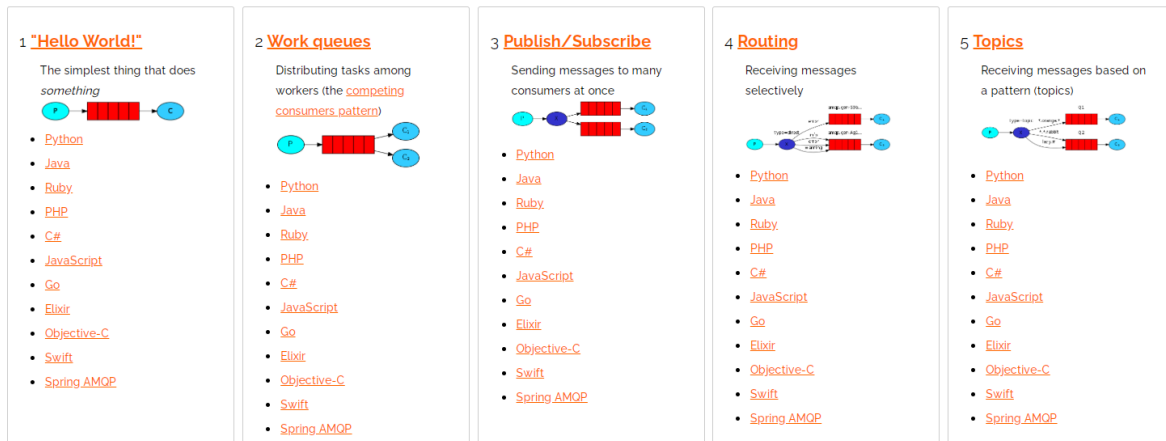
        factory.setVirtualHost("/lagou");
        factory.setUsername("laosun");
        factory.setPassword("123123");
        //3.通过工厂获得与MQ的连接
        Connection connection = factory.newConnection();
        return connection;
    }

    public static void main(String[] args) throws Exception{
        Connection connection = getConnection();
        System.out.println("connection = " + connection);
        connection.close();
    }
}

```

## 2.3 RabbitMQ模式

- RabbitMQ提供了6种消息模型，但是第6种其实是RPC，并不是MQ，因此我们只学习前5种
- 在线手册：<https://www.rabbitmq.com/getstarted.html>



- 5种消息模型，大体分为两类：
  - 1和2属于点对点
  - 3、4、5属于发布订阅模式（一对多）
- **点对点模式**：P2P（point to point）模式包含三个角色：
  - 消息队列（queue），发送者（sender），接收者（receiver）
  - 每个消息发送到一个特定的队列中，接收者从中获得消息
  - 队列中保留这些消息，直到他们被消费或超时
  - 特点：
    1. 每个消息只有一个消费者，一旦消费，消息就不在队列中了
    2. 发送者和接收者之间没有依赖性，发送者发送完成，不管接收者是否运行，都不会影响消息发送到队列中（我给你发微信，不管你看不看手机，反正我发完了）
    3. 接收者成功接收消息之后需向对象应答成功（确认）
  - 如果希望发送的每个消息都会被成功处理，那需要P2P
- **发布订阅模式**：publish（Pub）/subscribe（Sub）
  - pub/sub模式包含三个角色：交换机（exchange），发布者（publisher），订阅者（subscriber）
  - 多个发布者将消息发送交换机，系统将这些消息传递给多个订阅者



- 特点：
  1. 每个消息可以有多个订阅者
  2. 发布者和订阅者之间在时间上有依赖，对于某个交换机的订阅者，必须创建一个订阅后，才能消费发布者的消息
  3. 为了消费消息，订阅者必须保持运行状态；类似于，看电视直播。
- 如果希望发送的消息被多个消费者处理，可采用本模式

## 2.3.1 简单模式

下面引用官网的一段介绍：

### Introduction

RabbitMQ is a message broker: it accepts and forwards messages. You can think about it as a post office: when you put the mail that you want posting in a post box, you can be sure that Mr. or Ms. Mailperson will eventually deliver the mail to your recipient. In this analogy, RabbitMQ is a post box, a post office and a postman.

译文：RabbitMQ是一个消息代理:它接收和转发消息。你可以把它想象成一个邮局:当你把你想要寄的邮件放到一个邮箱里，你可以确定邮递员先生或女士最终会把邮件送到你的收件人那里。在这个类比中，RabbitMQ是一个邮箱、一个邮局和一个邮递员。

RabbitMQ本身只是接收，存储和转发消息，并不会对信息进行处理！

类似邮局，处理信件的应该是收件人而不是邮局！



### 2.3.1.1 生产者P

```
package simplest;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import util.ConnectionUtil;

/**
 * @BelongsProject: lagou-rabbitmq
 * @Author: GuoAn.Sun
 * @Description: 消息生产者
 */
public class Sender {
    public static void main(String[] args) throws Exception {
        String msg = "老孙: Hello,RabbitMQ!";

        // 1.获得连接
        Connection connection = ConnectionUtil.getConnection();
        // 2.在连接中创建通道（信道）
        Channel channel = connection.createChannel();
        // 3.创建消息队列(1,2,3,4,5)
```

```

    /*
    参数1:队列的名称
    参数2:队列中的数据是否持久化
    参数3:是否排外（是否支持扩展，当前队列只能自己用，不能给别人用）
    参数4:是否自动删除（当队列的连接数为0时，队列会销毁，不管队列是否还存保存数据）
    参数5:队列参数（没有参数为null）
    */
    channel.queueDeclare("queue1", false, false, false, null);
    // 4.向指定的队列发送消息(1,2,3,4)
    /*
    参数1:交换机名称，当前是简单模式，也就是P2P模式，没有交换机，所以名称为""
    参数2:目标队列的名称
    参数3:设置消息的属性（没有属性则为null）
    参数4:消息的内容(只接收字节数组)
    */
    channel.basicPublish("", "queue1", null, msg.getBytes());
    System.out.println("发送: " + msg);
    // 5.释放资源
    channel.close();
    connection.close();
}
}

```

启动生产者，即可前往管理端查看队列中的信息，会有一条信息没有处理和确认

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/lagou	queue1	classic		idle	1	0	1	0.00/s	0.00/s	0.00/s

### 2.3.1.2 消费者C

```

package simplest;

import com.rabbitmq.client.*;
import util.ConnectionUtil;

import java.io.IOException;

/**
 * @BelongsProject: lagou-rabbitmq
 * @Author: GuoAn.Sun
 * @Description: 消息接收者
 */
public class Recer {
    public static void main(String[] args) throws Exception {
        // 1.获得连接
        Connection connection = ConnectionUtil.getConnection();
        // 2.获得通道（信道）
        Channel channel = connection.createChannel();
        // 3.从信道中获得消息
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override //交付处理（收件人信息，包裹上的快递标签，协议的配置，消息）
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                // body就是从队列中获取的消息
                String s = new String(body);
            }
        };
    }
}

```

```

        System.out.println("接收 = " + s);
    }
};
// 4.监听队列 true:自动消息确认
channel.basicConsume("queue1", true, consumer);
}
}

```

启动消费者，前往管理端查看队列中的信息，所有信息都已经处理和确认，显示0

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/lagou	queue1	classic		idle	0	0	0	0.00/s	0.00/s	0.00/s

### 2.3.1.3 消息确认机制ACK

- 通过刚才的案例可以看出，消息一旦被消费，消息就会立刻从队列中移除
- RabbitMQ如何得知消息被消费者接收？
  - 如果消费者接收消息后，还没执行操作就抛异常宕机导致消费失败，但是RabbitMQ无从得知，这样消息就丢失了
  - 因此，RabbitMQ有一个**ACK机制**，当消费者获取消息后，会向RabbitMQ发送**回执ACK**，告知消息已经被接收
  - ACK：(Acknowledge character) 即是确认字符，在数据通信中，接收站发给发送站的一种传输类控制字符。表示发来的数据已确认接收无误我们在使用http请求时，http的状态码200就是告诉我们服务器执行成功
  - 整个过程就想快递员将包裹送到你手里，并且需要你的签字，并拍照回执
  - 不过这种回执ACK分为两种情况：
    - 自动ACK**：消息接收后，消费者立刻自动发送ACK（快递放在快递柜）
    - 手动ACK**：消息接收后，不会发送ACK，需要手动调用（快递必须本人签收）
  - 两种情况如何选择，需要看消息的重要性：
    - 如果消息不太重要，丢失也没有影响，自动ACK会比较方便
    - 如果消息非常重要，最好消费完成手动ACK，如果自动ACK消费后，RabbitMQ就会把消息从队列中删除，如果此时消费者抛异常宕机，那么消息就永久丢失了
- 修改手动消息确认

```

// false: 手动消息确认
channel.basicConsume("queue1", false, consumer);

```

- 结果如下：

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/lagou	queue1	classic		idle	0	1	1	0.00/s	0.00/s	0.00/s

- 解决问题

```

public class RecerByACK {
    public static void main(String[] args) throws Exception {
        // 1.获得连接
        Connection connection = ConnectionUtil.getConnection();
        // 2.获得通道（信道）
    }
}

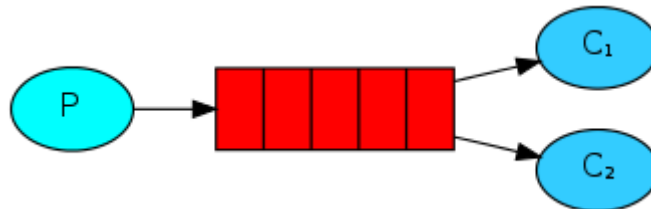
```

```

final Channel channel = connection.createChannel();
// 3.从信道中获得消息
DefaultConsumer consumer = new DefaultConsumer(channel){
    @Override //交付处理（收件人信息，包裹上的快递标签，协议的配置，消息）
    public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
        // body就是从队列中获取的消息
        String s = new String(body);
        System.out.println("接收 = " + s);
        // 手动确认（收件人信息，是否同时确认多个消息）
        channel.basicAck(envelope.getDeliveryTag(),false);
    }
};
// 4.监听队列，false:手动消息确认
channel.basicConsume("queue1", false,consumer);
}
}

```

## 2.3.2 工作队列模式



- 之前我们学习的简单模式，一个消费者来处理消息，如果生产者生产消息过快过多，而消费者的能力有限，就会产生消息在队列中堆积（生活中的滞销）
- 一个烧烤师傅，一次烤50支羊肉串，就一个人吃的话，烤好的肉串会越来越多，怎么处理？
- 多招揽客人进行消费即可。当我们运行许多消费者程序时，消息队列中的任务会被众多消费者共享，但其中某一个消息只会被一个消费者获取（100支肉串20个人吃，但是其中的某支肉串只能被一个人吃）

### 2.3.2.1 生产者P

```

public class MessageSender {
    public static void main(String[] args) throws Exception{
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();
        // 声明队列（此处为生产者，创建队列）注明出餐口位置，通知大家来排队
        channel.queueDeclare("test_work_queue",false,false,false,null);
        for(int i = 1;i<=100;i++) {
            String msg = "羊肉串 --> "+i;
            channel.basicPublish("", "test_work_queue", null, msg.getBytes());
            System.out.println("师傅烤好: " + msg);
        }
        channel.close();
        connection.close();
    }
}

```

### 2.3.2.2 消费者1

```
public class MessageReceiver1 {
    static int i = 1; // 记录执行次数
    public static void main(String[] args) throws IOException, TimeoutException
    {
        Connection connection = ConnectionUtil.getConnection();
        final Channel channel = connection.createChannel();
        // 声明队列（此处为消费者，不是声明创建队列，而且获取，二者代码相同）出餐口排队
        channel.queueDeclare("test_work_queue", false, false, false, null);
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                     AMQP.BasicProperties properties, byte[]
body) throws IOException {
                String msg = new String(body);
                System.out.println("【顾客1】吃掉 " + msg + " ! 共吃【"+i+++"】串");
                // 撸一会，有延迟
                try {
                    Thread.sleep(200);
                } catch (InterruptedException e){
                }
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        };
        channel.basicConsume("test_work_queue", false, consumer);
    }
}
```

### 2.3.2.3 消费者2

```
public class MessageReceiver2 {
    static int i = 1; // 记录执行次数
    public static void main(String[] args) throws IOException, TimeoutException
    {
        Connection connection = ConnectionUtil.getConnection();
        final Channel channel = connection.createChannel();
        // 声明队列（此处为消费者，不是声明创建队列，而且获取，二者代码相同）出餐口排队
        channel.queueDeclare("test_work_queue", false, false, false, null);
        Consumer consumer = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                                     AMQP.BasicProperties properties, byte[]
body) throws IOException {
                String msg = new String(body);
                System.out.println("【顾客2】吃掉 " + msg + " ! 共吃【"+i+++"】串");
                // 撸一会，有延迟
                try {
                    Thread.sleep(200);
                } catch (InterruptedException e){
                }
                channel.basicAck(envelope.getDeliveryTag(), false);
            }
        };
    }
}
```

```

    channel.basicConsume("test_work_queue", false, consumer);
}
}

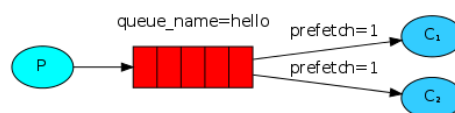
```

- 先运行2个消费者，排队等候消费（取餐），再运行生产者开始生产消息（烤肉串）
- 虽然两个消费者的消费速度不一致（线程休眠时间），但是消费的数量却是一致的，各消费50个消息
  - 例如：工作中，A同学编码速率高，B同学编码速率低，两个人同时开发一个项目，A10天完成，B30天完成，A完成自己的编码部分，就无所事事了，等着B完成就可以了，这样是不可以的，应该遵循“能者多劳”
  - 效率高的多干点，效率低的少干点
  - 看下面官网是如何给出解决思路的：

#### Fair dispatch

You might have noticed that the dispatching still doesn't work exactly as we want. For example in a situation with two workers, when all odd messages are heavy and even messages are light, one worker will be constantly busy and the other one will do hardly any work. Well, RabbitMQ doesn't know anything about that and will still dispatch messages evenly.

This happens because RabbitMQ just dispatches a message when the message enters the queue. It doesn't look at the number of unacknowledged messages for a consumer. It just blindly dispatches every n-th message to the n-th consumer.



In order to defeat that we can use the `basicQos` method with the `prefetchCount = 1` setting. This tells RabbitMQ not to give more than one message to a worker at a time. Or, in other words, don't dispatch a new message to a worker until it has processed and acknowledged the previous one. Instead, it will dispatch it to the next worker that is not still busy.

```

int prefetchCount = 1;
channel.basicQos(prefetchCount);

```

#### 公平的分配

您可能已经注意到分派仍然不能完全按照我们的要求工作。例如，如果有两个员工，当所有奇怪的消息都很重，甚至消息都很轻时，一个员工会一直很忙，而另一个人几乎什么工作都不做。好吧，RabbitMQ对此一无所知，它仍然会均匀地分派消息。

这是因为RabbitMQ只在消息进入队列时发送消息。它不查看用户未确认消息的数量。它只是盲目地将每条第n个消息分派给第n个消费者。

为了克服这个问题，我们可以使用设置为`prefetchCount = 1`的`basicQos`方法。这告诉RabbitMQ一次不要给一个worker发送一条以上的消息。或者，换句话说，在worker处理并确认前一个消息之前，不要向它发送新消息。相反，它将把它分派到下一个不繁忙的worker。

```

// 声明队列（此处为消费者，不是声明创建队列，而且获取，二者代码相同）出餐口排队
channel.queueDeclare("test_work_queue", false, false, false, null);
// 可以理解为：快递一个一个送，送完一个再送下一个，速度快的送件就多
channel.basicQos(1);

```

- 能者多劳必须要配合手动的ACK机制才生效

### 2.3.2.4 面试题：避免消息堆积？

1. workqueue，多个消费者监听同一个队列
2. 接收到消息后，通过线程池，异步消费

## 2.3.3 发布订阅模式

看官网：

### Publish/Subscribe

In the [previous tutorial](#) we created a work queue. The assumption behind a work queue is that each task is delivered to exactly one worker. In this part we'll do something completely different -- we'll deliver a message to multiple consumers. This pattern is known as "publish/subscribe".

To illustrate the pattern, we're going to build a simple logging system. It will consist of two programs -- the first will emit log messages and the second will receive and print them.

In our logging system every running copy of the receiver program will get the messages. That way we'll be able to run one receiver and direct the logs to disk; and at the same time we'll be able to run another receiver and see the logs on the screen.

Essentially, published log messages are going to be broadcast to all the receivers.

发布-订阅

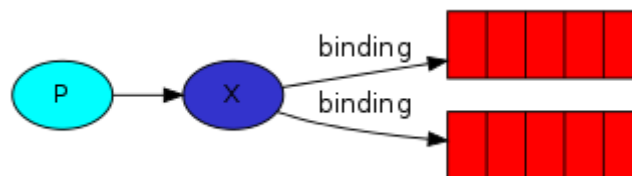
在上一篇教程中，我们创建了一个工作队列。工作队列背后的假设是，每个任务都被准确地交付给一个工作者。在这一部分中，我们将做一些完全不同的事情——将消息传递给多个消费者。此模式称为“发布/订阅”。

为了演示这个模式，我们将构建一个简单的日志记录系统。它将由两个程序组成——第一个将发送日志消息，第二个将接收和打印它们。

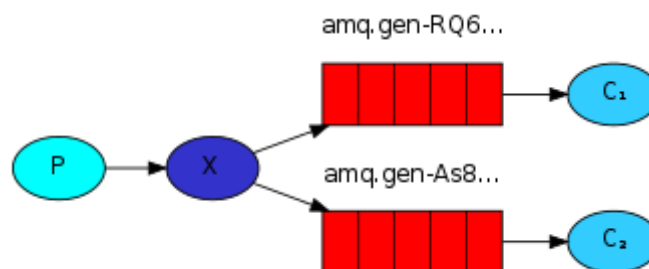
在我们的日志系统中，接收程序的每一个正在运行的副本都将获得消息。这样我们就可以运行一个接收器并将日志指向磁盘；与此同时，我们可以运行另一个接收器并在屏幕上看到日志。

基本上，发布的日志消息将广播到所有接收方。

**生活中的案例：就是玩抖音快手，众多粉丝关注一个视频主，视频主发布视频，所有粉丝都可以得到视频通知**



- 上图中，X就是视频主，红色的队列就是粉丝。binding是绑定的意思（关注）
- P生产者发送信息给X路由，X将信息转发给绑定X的队列



- X队列将信息通过信道发送给消费者，从而进行消费
- 整个过程，必须先创建路由

- 路由在生产者程序中创建
- 因为路由没有存储消息的能力，当生产者将信息发送给路由后，消费者还没有运行，所以没有队列，路由并不知道将信息发送给谁
- 运行程序的顺序：
  1. MessageSender
  2. MessageReceiver1和MessageReceiver2
  3. MessageSender

### 2.3.3.1 生产者

```
public class Sender {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();

        // 声明路由(路由名，路由类型)
        // fanout: 不处理路由键（只需要将队列绑定到路由上，发送到路由的消息都会被转发到与该
        // 路由绑定的所有队列上）
        channel.exchangeDeclare("test_exchange_fanout", "fanout");
        String msg = "hello, 大家好! ";
        channel.basicPublish("test_exchange_fanout", "", null, msg.getBytes());
        System.out.println("生产者: " + msg);
        channel.close();
        connection.close();
    }
}
```

### 2.3.3.2 消费者1

```
public class Recer1 {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();
        // 声明队列

        channel.queueDeclare("test_exchange_fanout_queue_1", false, false, false, null);
        // 绑定路由（关注）
        /*
        参数1: 队列名
        参数2: 交换器名称
        参数3: 路由key（暂时无用，""即可）
        */
        channel.queueBind("test_exchange_fanout_queue_1",
            "test_exchange_fanout", "");
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                String s = new String(body);
                System.out.println("【消费者1】 = " + s);
            }
        };
        // 4.监听队列 true:自动消息确认
    }
}
```



```

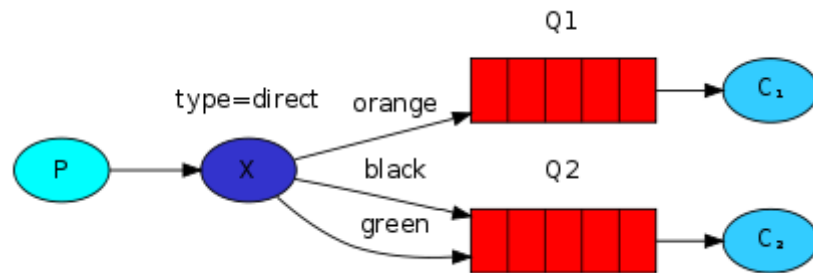
        channel.basicConsume("test_exchange_fanout_queue_1", true, consumer);
    }
}

```

### 2.3.3.3 消费者2

- 将消费者1代码中的1修改为2即可，具体代码略

## 2.3.4 路由模式



- 路由会根据类型进行**定向分发**消息给不同的队列，如图所示
- 可以理解为是快递公司的分拣中心，整个小区，东面的楼小张送货，西面的楼小王送货

### 2.3.4.1 生产者

```

public class Sender {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();
        // 声明路由(路由名, 路由类型)
        // direct: 根据路由键进行定向分发消息
        channel.exchangeDeclare("test_exchange_direct", "direct");
        String msg = "用户注册, 【userid=S101】";
        channel.basicPublish("test_exchange_direct", "insert", null,
msg.getBytes());
        System.out.println("[用户系统]: " + msg);
        channel.close();
        connection.close();
    }
}

```

### 2.3.4.2 消费者1

```

public class Recer1 {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();
        // 声明队列

channel.queueDeclare("test_exchange_direct_queue_1", false, false, false, null);
        // 绑定路由(如果路由键的类型是 添加, 删除, 修改 的话, 绑定到这个队列1上)

```

```

        channel.queueBind("test_exchange_direct_queue_1",
            "test_exchange_direct", "insert");
        channel.queueBind("test_exchange_direct_queue_1",
            "test_exchange_direct", "update");
        channel.queueBind("test_exchange_direct_queue_1",
            "test_exchange_direct", "delete");
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                String s = new String(body);
                System.out.println("【消费者1】 = " + s);
            }
        };
        // 4.监听队列 true:自动消息确认
        channel.basicConsume("test_exchange_direct_queue_1", true, consumer);
    }
}

```

### 2.3.4.3 消费者2

```

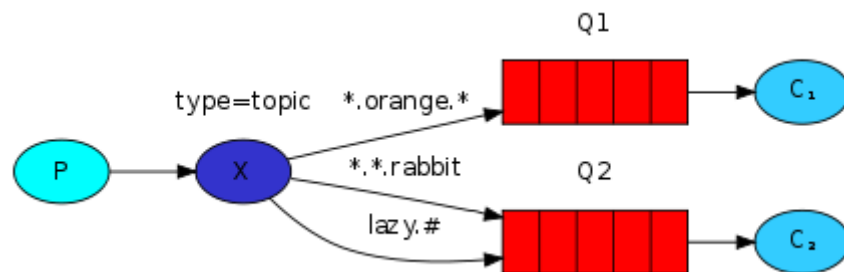
public class Recer2 {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();
        // 声明队列

        channel.queueDeclare("test_exchange_direct_queue_2", false, false, false, null);
        // 绑定路由（如果路由键的类型是 查询 的话，绑定到这个队列2上）
        channel.queueBind("test_exchange_direct_queue_2",
            "test_exchange_direct", "select");
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                String s = new String(body);
                System.out.println("【消费者2】 = " + s);
            }
        };
        // 4.监听队列 true:自动消息确认
        channel.basicConsume("test_exchange_direct_queue_2", true, consumer);
    }
}

```

1. 记住运行程序的顺序，先运行一次sender（创建路由器），
2. 有了路由器之后，在创建两个Recer1和Recer2，进行队列绑定
3. 再次运行sender，发出消息

### 2.3.5 通配符模式



- 和路由模式90%是一样的。
- 唯独的区别就是路由键支持模糊匹配
- 匹配符号
  - \*：只能匹配一个词（正好一个词，多一个不行，少一个也不行）
  - #：匹配0个或更多个词
- 看一下官网案例：
  - Q1绑定了路由键 \*.orange.\*      Q2绑定了路由键 \*.\*.rabbit 和 lazy.#
  - 下面生产者的消息会被发送给哪个队列？

quick.orange.rabbit	# Q1	Q2
lazy.orange.elephant	# Q1	Q2
quick.orange.fox	# Q1	
lazy.brown.fox	# Q2	
lazy.pink.rabbit	# Q2	
quick.brown.fox	# 无	
orange	# 无	
quick.orange.male.rabbit	# 无	

### 2.3.5.1 生产者

```
public class Sender {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();

        // 声明路由(路由名，路由类型)
        // topic: 模糊匹配的定向分发
        channel.exchangeDeclare("test_exchange_topic", "topic");
        String msg = "商品降价";
        channel.basicPublish("test_exchange_topic", "product.price", null,
msg.getBytes());
        System.out.println("[用户系统]: " + msg);
        channel.close();
        connection.close();
    }
}
```

### 2.3.5.2 消费者1

```

public class Recer1 {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();
        // 声明队列

        channel.queueDeclare("test_exchange_topic_queue_1", false, false, false, null);
        // 绑定路由（绑定 用户相关 的消息）
        channel.queueBind("test_exchange_topic_queue_1", "test_exchange_topic",
            "user.#");
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                String s = new String(body);
                System.out.println("【消费者1】 = " + s);
            }
        };
        // 4.监听队列 true:自动消息确认
        channel.basicConsume("test_exchange_topic_queue_1", true, consumer);
    }
}

```

### 2.3.5.3 消费者2

```

public class Recer2 {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();
        // 声明队列

        channel.queueDeclare("test_exchange_topic_queue_2", false, false, false, null);
        // 绑定路由（绑定 商品和订单相关 的消息）
        channel.queueBind("test_exchange_topic_queue_2", "test_exchange_topic",
            "product.#");
        channel.queueBind("test_exchange_topic_queue_2", "test_exchange_topic",
            "order.#");
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                String s = new String(body);
                System.out.println("【消费者2】 = " + s);
            }
        };
        // 4.监听队列 true:自动消息确认
        channel.basicConsume("test_exchange_topic_queue_2", true, consumer);
    }
}

```

## 2.4 持久化

- 消息的可靠性是RabbitMQ的一大特色，那么RabbitMQ是如何避免消息丢失？
  - 消费者的ACK确认机制，可以防止消费者丢失消息
  - 万一在消费者消费之前，RabbitMQ服务器宕机了，那消息也会丢失
- 想要将消息持久化，那么 路由和队列都要持久化 才可以

## 2.4.1 生产者

```
public class Sender {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();

        // 声明路由(路由名, 路由类型, 持久化)
        channel.exchangeDeclare("test_exchange_topic", "topic", true);
        String msg = "商品降价";
        // 发送消息(第三个参数作用是让消息持久化)
        channel.basicPublish("test_exchange_topic", "product.price",
            MessageProperties.PERSISTENT_TEXT_PLAIN, msg.getBytes());
        System.out.println("[用户系统]: " + msg);
        channel.close();
        connection.close();
    }
}
```

## 2.4.2 消费者

```
public class Recer1 {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();
        // 声明队列( 第二个参数为true: 支持持久化)

        channel.queueDeclare("test_exchange_topic_queue_1", true, false, false, null);
        channel.queueBind("test_exchange_topic_queue_1", "test_exchange_topic",
            "user.#");
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
                AMQP.BasicProperties properties, byte[] body) throws IOException {
                String s = new String(body);
                System.out.println("【消费者1】 = " + s);
            }
        };
        channel.basicConsume("test_exchange_topic_queue_1", true, consumer);
    }
}
```

## 2.4 Spring整合RabbitMQ

- 五种消息模型，在企业中应用最广泛的就是最后一种：定向匹配topic
- Spring AMQP 是基于 Spring 框架的AMQP消息解决方案，提供模板化的发送和接收消息的抽象层，提供基于消息驱动的 POJO的消息监听等，简化了我们对于RabbitMQ相关程序的开发。

## 2.4.1 生产端工程

- 依赖

```
<dependency>
  <groupId>org.springframework.amqp</groupId>
  <artifactId>spring-rabbit</artifactId>
  <version>2.0.1.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.25</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.9</version>
</dependency>
```

- spring-rabbitmq-producer.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:rabbit="http://www.springframework.org/schema/rabbit"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/rabbit
    http://www.springframework.org/schema/rabbit/spring-rabbit.xsd">

  <!-- 1.配置连接 -->
  <rabbit:connection-factory
    id="connectionFactory"
    host="192.168.204.141"
    port="5672"
    username="laosun"
    password="123123"
    virtual-host="/lagou"
  />

  <!-- 2.配置队列 -->
  <rabbit:queue name="test_spring_queue_1"/>

  <!-- 3.配置rabbitAdmin: 主要用于在Java代码中对理队和队列进行管理，用于创建、绑定、删除队列与交换机，发送消息等 -->
  <rabbit:admin connection-factory="connectionFactory"/>

  <!-- 4.配置topic类型exchange; 队列绑定到交换机 -->
  <rabbit:topic-exchange name="spring_topic_exchange">
    <rabbit:bindings>
      <rabbit:binding queue="test_spring_queue_1" pattern="msg.#"/>
    </rabbit:bindings>
  </rabbit:topic-exchange>
```

```

</rabbit:topic-exchange>
<!-- 5. 配置消息对象json转换类 -->
<bean id="jsonMessageConverter"
class="org.springframework.amqp.support.converter.Jackson2JsonMessageConverter"/
>

<!-- 6. 配置RabbitTemplate（消息生产者） -->
<rabbit:template id="rabbitTemplate"
    connection-factory="connectionFactory"
    exchange="spring_topic_exchange"
    message-converter="jsonMessageConverter"

/>

</beans>

```

- 发消息

```

public class Sender {
    public static void main(String[] args) {
        // 1.创建spring容器
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("spring/spring-rabbitmq-producer.xml");
        // 2.从容器中获取对象
        RabbitTemplate template = context.getBean(RabbitTemplate.class);
        // 3.发送消息
        Map<String, String> map = new HashMap();
        map.put("name", "大佬孙");
        map.put("email", "19998539@qq.com");
        template.convertAndSend("msg.user", map);
        context.close();
    }
}

```

## 2.4.2 消费端工程

- 依赖与生产者一致
- spring-rabbitmq-consumer.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:rabbit="http://www.springframework.org/schema/rabbit"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/rabbit
        http://www.springframework.org/schema/rabbit/spring-rabbit.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <!-- 1. 配置连接 -->
    <rabbit:connection-factory
        id="connectionFactory"
        host="192.168.204.141"

```

```

        port="5672"
        username="laosun"
        password="123123"
        virtual-host="/lagou"
    />

    <!-- 2. 配置队列 -->
    <rabbit:queue name="test_spring_queue_1"/>
    <!-- 3.配置rabbitAdmin -->
    <rabbit:admin connection-factory="connectionFactory"/>
    <!-- 4.springIOC注解扫描包-->
    <context:component-scan base-package="listener"/>
    <!-- 5.配置监听 -->
    <rabbit:listener-container connection-factory="connectionFactory">
        <rabbit:listener ref="consumerListener" queue-
names="test_spring_queue_1" />
    </rabbit:listener-container>
</beans>

```

- 消费者

- MessageListener接口用于spring容器接收到消息后处理消息
- 如果需要使用自己定义的类型 来实现 处理消息时，必须实现该接口，并重写onMessage()方法
- 当spring容器接收消息后，会自动交由onMessage进行处理

```

@Component
public class ConsumerListener implements MessageListener {

    // jackson提供序列化和反序列中使用最多的类，用来转换json的
    private static final ObjectMapper MAPPER = new ObjectMapper();

    public void onMessage(Message message) {
        try {
            // 将message对象转换成json
            JsonNode jsonNode = MAPPER.readTree(message.getBody());
            String name = jsonNode.get("name").asText();
            String email = jsonNode.get("email").asText();
            System.out.println("从队列中获取：【"+name+"的邮箱是:"+email+"】");
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}

```

- 启动项目

```

public class TestRunner {
    public static void main(String[] args) throws Exception {
        // 获得容器
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("spring/spring-rabbitmq-consumer.xml");
        // 让程序一直运行，别终止
        System.in.read();
    }
}

```



## 2.5 消息成功确认机制

在实际场景下，有的生产者发送的消息是必须保证成功发送到消息队列中，那么如何保证成功投递呢？

- 事务机制
- 发布确认机制

### 2.5.1 事务机制

- AMQP协议提供了一种保证消息成功投递的方式，通过信道开启 transactional 模式
- 并利用信道 的三个方法来实现以事务方式 发送消息，若发送失败，通过异常处理回滚事务，确保消息成功投递
  - channel.txSelect()：开启事务
  - channel.txCommit()：提交事务
  - channel.txRollback()：回滚事务
- Spring已经对上面三个方法进行了封装，所以我们只能使用原始的代码演示

#### 2.5.1.1 生产者

```
public class Sender {
    public static void main(String[] args) throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();

        channel.exchangeDeclare("test_transaction", "topic");
        channel.txSelect(); // 开启事务
        try {
            channel.basicPublish("test_transaction", "product.price", null, "商品
降价1".getBytes());
            // System.out.println(1/0); // 模拟异常!
            channel.basicPublish("test_transaction", "product.price", null, "商品
降价2".getBytes());
            System.out.println("消息全部发出!");
            channel.txCommit(); // 事务提交
        } catch (Exception e) {
            System.out.println("由于系统异常，消息全部撤回!");
            channel.txRollback(); // 事务回滚
            e.printStackTrace();
        } finally {
            channel.close();
            connection.close();
        }
    }
}
```

#### 2.5.1.2 消费者

```
public class Recer {
    public static void main(String[] args) throws Exception {
```

```

        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();
        channel.queueDeclare("test_transaction_queue", false, false, false, null);
        channel.queueBind("test_transaction_queue", "test_transaction",
"product.#");
        DefaultConsumer consumer = new DefaultConsumer(channel){
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                String s = new String(body);
                System.out.println("【消费者】 = " + s);
            }
        };
        channel.basicConsume("test_transaction_queue", true, consumer);
    }
}

```

## 2.5.2 Confirm发布确认机制

- RabbitMQ为了保证消息的成功投递，采用通过AMQP协议层面为我们提供事务机制的方案，但是采用事务会大大降低消息的吞吐量
- 老孙我本机SSD硬盘测试结果10w条消息未开启事务，大约8s发送完毕；而开启了事务后，需要将近310s，差了30多倍。
- 接着老孙翻阅官网，发现官网中已标注

Using standard AMQP 0-9-1, the only way to guarantee that a message isn't lost is by using transactions – make the channel transactional then for each message or set of messages publish, commit. In this case, transactions are unnecessarily heavyweight and decrease throughput by a factor of 250. To remedy this, a confirmation mechanism was introduced. It mimics the consumer acknowledgements mechanism already present in the protocol.

关键性译文：开启事务性能最大损失超过250倍

- 那么有没有更加高效的解决方式呢？答案就是采用Confirm模式。
- 事务效率为什么会这么低呢？试想一下：10条消息，前9条成功，如果第10条失败，那么9条消息要全部撤销回滚。太太太浪费
- 而confirm模式则采用**补发**第10条的措施来完成10条消息的送达

### 2.5.2.1 在spring中应用

- spring-rabbitmq-producer.xml

```

<!--1.配置连接，启动生产者确认机制: publisher-confirms="true"-->
<rabbit:connection-factory id="connectionFactory"
    host="192.168.204.141"
    port="5672"
    username="laosun"
    password="123123"
    virtual-host="/lagou"
    publisher-confirms="true"
/>

```

```

<!--6.配置rabbitmq的模版,添加确认回调处理类:confirm-
callback="msgSendConfirmCallback"-->
<rabbit:template id="rabbitTemplate"
    connection-factory="connectionFactory"
    exchange="spring_topic_exchange"
    message-converter="jsonMessageConverter"
    confirm-callback="msgSendConfirmCallback"/>

<!--7.确认机制处理类-->
<bean id="msgSendConfirmCallback" class="confirm.MsgSendConfirmCallback"/>

```

- 消息确认处理类

```

package confirm;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.rabbit.support.CorrelationData;
import org.springframework.stereotype.Component;

import java.io.IOException;

/**
 * @BelongsProject: spring-rabbitmq-producer
 * @Author: GuoAn.Sun
 * @Description: 确认机制
 */
@Component
public class MsgSendConfirmCallback implements
    RabbitTemplate.ConfirmCallback {
    public void confirm(CorrelationData correlationData, boolean b, String
s) {
        if (b){
            System.out.println("消息确认成功!!");
        } else {
            System.out.println("消息确认失败。。。");
            // 如果本条消息一定要发送到队列中, 例如如下订单消息, 我们可以采用消息补发
            // 采用递归(固定次数, 不可无限)或 redis+定时任务
        }
    }
}

```

- log4j.properties

```

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %m%n

log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.File=rabbitmq.log
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %l
%m%n

log4j.rootLogger=debug, stdout,file

```

- 发送消息





```
public class Sender {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new
        ClassPathXmlApplicationContext("spring/spring-rabbitmq-producer.xml");
        RabbitTemplate rabbitTemplate =
        context.getBean(RabbitTemplate.class);
        Map<String,String> map = new HashMap<String, String>();
        map.put("name", "吕布");
        map.put("email", "666@qq.com");
        // 第一个参数是路由名称,
        // 不写, 则使用spring容器中创建的路由
        // 乱写一个, 因为路由名错误导致报错, 则进入消息确认失败流程
        rabbitTemplate.convertAndSend("x", "msg.user", map);
        System.out.println("ok");
        context.close();
    }
}
```

## 2.6 消费端限流

- 在沙漠中行走, 3天不喝水, 突然喝水, 如果使劲喝, 容易猝死, 要一口一口慢慢喝
- 我们 Rabbitmq 服务器积压了成千上万条未处理的消息, 然后随便打开一个消费者客户端, 就会出现这样的情况: 巨量的消息瞬间全部喷涌推送过来, 但是单个客户端无法同时处理这么多数据, 就会被压垮崩溃
- 所以, 当数据量特别大的时候, 我们对**生产端限流**肯定是不科学的, 因为有时候并发量就是特别大, 有时候并发量又特别少, 这是用户的行为, 我们是无法约束的
- 所以我们应该对**消费端限流**, 用于保持消费端的稳定
- 例如: 汽车企业不停的生产汽车, 4S店有好多库存车卖不出去, 但是也不会降价处理, 就是要保证市值的稳定, 如果生产多少台, 就卖多少台, 不管价格的话, 市场就乱了, 所以我们要用不变的价格来稳住消费者购车, 才能平稳发展
- RabbitMQ 提供了一种 Qos ( Quality of Service, 服务质量 ) 服务质量保证功能
  - 即在非自动确认消息的前提下, 如果一定数目的消息未被确认前, 不再进行消费新的消息
- 生产者使用循环发出多条消息

```
for(int i = 1;i<=10;i++) {
    rabbitTemplate.convertAndSend("msg.user", map);
    system.out.println("消息已发出...");
}
```

- 生产10条堆积未处理的消息

Overview					Messages			Message rates			
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
/lagou	test_spring_queue_1	classic		 idle	10	0	10	0.00/s	0.00/s	0.00/s	
/lagou	test_exchange_topic_queue_2	classic		 idle	0	0	0				

- 消费者进行限流处理

```

<!--5.配置监听-->
<!-- prefetch="3" 一次性消费的消息数量。会告诉 RabbitMQ 不要同时给一个消费者推送多于
N 个消息，一旦有 N 个消息还没有ack，则该 consumer 将阻塞，直到消息被ack-->
<!-- acknowledge-mode: manual 手动确认-->
<rabbit:listener-container connection-factory="connectionFactory"
prefetch="3" acknowledge="manual">
    <rabbit:listener ref="consumerListener" queue-
names="test_spring_queue_1" />
</rabbit:listener-container>

```

```

// AbstractAdaptableMessageListener用于在spring容器接收到消息后用于处理消息的抽象
基类
@Component
public class ConsumerListener extends AbstractAdaptableMessageListener {
    // jackson提供序列化和反序列中使用最多的类，用来转换json的
    private static final ObjectMapper MAPPER = new ObjectMapper();

    public void onMessage(Message message, Channel channel) throws Exception
    {
        try {
            // String str = new String(message.getBody());
            // 将message对象转换成json
            JsonNode jsonNode = MAPPER.readTree(message.getBody());
            String name = jsonNode.get("name").asText();
            String email = jsonNode.get("email").asText();
            System.out.println("从队列中获取：【"+name+"的邮箱是:"+email+"】");

            long deliveryTag =
message.getMessageProperties().getDeliveryTag();
            //确认收到(参数1,参数2)
            /*
            参数1: RabbitMQ 向该 Channel 投递的这条消息的唯一标识 ID，是一个单调递
            增的正整数，delivery_tag 的范围仅限于 Channel
            参数2: 为了减少网络流量，手动确认可以被批处理，当该参数为 true 时，则可以
            一次性确认 delivery_tag 小于等于传入值的所有消息
            */
            channel.basicAck(deliveryTag , true);
            Thread.sleep(3000);
            System.out.println("休息三秒然后再接收消息");
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}

```

- 每次确认接收3条消息

Overview					Messages			Message rates			
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
/lagou	test_spring_queue_1	classic	D	idle	7	3	10	0.00/s	0.00/s	0.00/s	
/lagou	test_exchange_topic_queue_2	classic	D	idle	0	0	0				

## 2.7 过期时间TTL

- Time To Live：生存时间、还能活多久，单位毫秒

- 在这个周期内，消息可以被消费者正常消费，超过这个时间，则自动删除（其实是被称为dead message并投入到死信队列，无法消费该消息）
- RabbitMQ可以对消息和队列设置TTL
  - 通过队列设置，队列中所有消息都有相同的过期时间
  - 对消息单独设置，每条消息的TTL可以不同（更颗粒化）

## 2.7.1 设置队列TTL

- spring-rabbitmq-producer.xml

```
<!--2.重新配置一个队列，同时，对队列中的消息设置过期时间-->
<rabbit:queue name="test_spring_queue_ttl" auto-declare="true">
  <rabbit:queue-arguments>
    <entry key="x-message-ttl" value-type="long" value="5000"></entry>
  </rabbit:queue-arguments>
</rabbit:queue>
```

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/lagou	test_spring_queue_ttl	classic	D TTL	idle	1	0	1	0.20/s		

- 5秒之后，消息自动删除

Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/lagou	test_spring_queue_ttl	classic	D TTL	idle	0	0	0	0.20/s		

## 2.7.1 设置消息TTL

- 设置某条消息的ttl，只需要在创建发送消息时指定即可

```
<!--2.配置队列-->
<rabbit:queue name="test_spring_queue_ttl_2">
```

```
package test;

import org.springframework.amqp.core.Message;
import org.springframework.amqp.core.MessageProperties;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.HashMap;
import java.util.Map;

/**
 * @BelongsProject: spring-rabbitmq-producer
 * @Author: GuoAn.Sun
 * @Description: 生产者
 */
public class Sender2 {
    public static void main(String[] args) {
```

```

ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("spring/spring-rabbitmq-producer.xml");
RabbitTemplate rabbitTemplate = context.getBean(RabbitTemplate.class);
// 创建消息配置对象
MessageProperties messageProperties = new MessageProperties();
// 设置消息过期时间
messageProperties.setExpiration("6000");
// 创建消息
Message message = new Message("6秒后自动删除".getBytes(),
messageProperties);
// 发送消息
rabbitTemplate.convertAndSend("msg.user", message);
System.out.println("消息已发出...");

context.close();
}
}

```

- 如果同时设置了queue和message的TTL值，则二者中较小的才会起作用

## 2.8 死信队列

- DLX ( Dead Letter Exchanges ) 死信交换机/死信邮箱，当消息在队列中由于某些原因没有被及时消费而变成死信 ( dead message ) 后，这些消息就会被分发到DLX交换机中，而绑定DLX交换机的队列，称之为：“死信队列”
- 消息没有被及时消费的原因：
  - 消息被拒绝 ( basic.reject/ basic.nack ) 并且不再重新投递 requeue=false
  - 消息超时未消费
  - 达到最大队列长度



- spring-rabbitmq-producer-dlx.xml

```

<!--1.配置连接-->
<rabbit:connection-factory id="connectionFactory"
    host="192.168.204.141"
    port="5672"
    username="laosun"
    password="123123"
    virtual-host="/lagou"/>

<!--3.配置rabbitAdmin:主要用于在java代码中对队列的管理，用来创建，绑定，删除队列与交换机，发送消息等-->
<rabbit:admin connection-factory="connectionFactory"/>

<!--6.配置rabbitmq的模版-->
<rabbit:template id="rabbitTemplate"

```

```

        connection-factory="connectionFactory"
        exchange="my_exchange"/>

<!--
#####
#####-->

<!--死信队列-->
<rabbit:queue name="dlx_queue"/>
<!--定向死信交换机-->
<rabbit:direct-exchange name="dlx_exchange" >
    <rabbit:bindings>
        <rabbit:binding key="dlx_ttl" queue="dlx_queue"></rabbit:binding>
        <rabbit:binding key="dlx_max" queue="dlx_queue"></rabbit:binding>
    </rabbit:bindings>
</rabbit:direct-exchange>

<!--测试超时的队列-->
<rabbit:queue name="test_ttl_queue">
    <rabbit:queue-arguments>
        <!--队列ttl为6秒-->
        <entry key="x-message-ttl" value-type="long" value="6000"/>
        <!--超时 消息 投递给 死信交换机-->
        <entry key="x-dead-letter-exchange" value="dlx_exchange"/>
    </rabbit:queue-arguments>
</rabbit:queue>

<!--测试超长度的队列-->
<rabbit:queue name="test_max_queue">
    <rabbit:queue-arguments>
        <!--队列ttl为6秒-->
        <entry key="x-max-length" value-type="long" value="2"/>
        <!--超时 消息 投递给 死信交换机-->
        <entry key="x-dead-letter-exchange" value="dlx_exchange"/>
    </rabbit:queue-arguments>
</rabbit:queue>

<!--定向测试交换机-->
<rabbit:direct-exchange name="my_exchange" >
    <rabbit:bindings>
        <rabbit:binding key="dlx_ttl" queue="test_ttl_queue">
</rabbit:binding>
        <rabbit:binding key="dlx_max" queue="test_max_queue">
</rabbit:binding>
    </rabbit:bindings>
</rabbit:direct-exchange>

```

- 发消息进行测试

```

public class SenderDLX {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("spring/spring-rabbitmq-producer-dlx.xml");
        RabbitTemplate rabbitTemplate =
context.getBean(RabbitTemplate.class);
        // rabbitTemplate.convertAndSend("dlx_ttl", "测试超时".getBytes());
    }
}

```



```

        rabbitTemplate.convertAndSend("dlx_max", "测试长度1".getBytes());
        rabbitTemplate.convertAndSend("dlx_max", "测试长度2".getBytes());
        rabbitTemplate.convertAndSend("dlx_max", "测试长度3".getBytes());

        System.out.println("消息已发出...");
        context.close();
    }
}

```

## 2.9 延迟队列

- 延迟队列：TTL + 死信队列的合体
- 死信队列只是一种特殊的队列，里面的消息仍然可以消费
- 在电商开发部分中，都会涉及到延时关闭订单，此时延迟队列正好可以解决这个问题

### 2.9.1 生产者

沿用上面死信队列案例的超时测试，超时时间改为订单关闭时间即可

### 2.9.2 消费者

- spring-rabbitmq-consumer.xml

```

<!-- 监听死信队列 -->
<rabbit:listener-container connection-factory="connectionFactory" prefetch="3"
acknowledge="manual">
    <rabbit:listener ref="consumerListener" queue-names="dlx_queue" />
</rabbit:listener-container>

```

## 3. RabbitMQ 集群

rabbitmq有3种模式，但集群模式是2种。具体如下：

- 单一模式：即单机情况不做集群，就单独运行一个rabbitmq而已。之前我们一直在用
- 普通模式：默认模式，以两个节点（A、B）为例来进行说明
  - 当消息进入A节点的Queue后，consumer从B节点消费时，RabbitMQ会在A和B之间创建临时通道进行消息传输，把A中的消息实体取出并经过通过交给B发送给consumer
  - 当A故障后，B就无法取到A节点中未消费的消息实体
    - 如果做了消息持久化，那么得等A节点恢复，然后才可被消费
    - 如果没有持久化的话，就会产生消息丢失的现象
- 镜像模式：非常经典的 mirror 镜像模式，保证 100% 数据不丢失。
  - 高可靠性解决方案，主要就是实现数据的同步，一般来讲是 2 - 3 个节点实现数据同步
  - 对于 100% 数据可靠性解决方案，一般是采用 3 个节点。
  - 在实际工作中也是用得最多的，并且实现非常的简单，一般互联网大厂都会构建这种镜像集群模式
- 还有主备模式，远程模式，多活模式等，本次课程不作为重点，可自行查阅资料了解

## 3.1 集群搭建

前置条件：准备两台linux，并安装好rabbitmq

- 集群步骤如下：

### 1. 修改 /etc/hosts 映射文件

1号服务器：

```
127.0.0.1 A localhost localhost.localdomain localhost4
localhost4.localdomain4
::1      A localhost localhost.localdomain localhost6
localhost6.localdomain6

192.168.204.141 A
192.168.204.142 B
```

2号服务器：

```
127.0.0.1 B localhost localhost.localdomain localhost4
localhost4.localdomain4
::1      B localhost localhost.localdomain localhost6
localhost6.localdomain6

192.168.204.141 A
192.168.204.142 B
```

### 2. 相互通信，cookie必须保持一致，同步 rabbitmq的cookie 文件：跨服务器拷贝 .erlang.cookie（隐藏文件，使用 ls -all 显示）

```
[root@A opt]# scp /var/lib/rabbitmq/.erlang.cookie
192.168.204.142:/var/lib/rabbitmq
```

**修改cookie文件，要重启服务器，reboot**

### 3. 停止防火墙，启动rabbitmq服务

```
[root@A ~]# systemctl stop firewalld
[root@A ~]# systemctl start rabbitmq-server
```

### 4. 加入集群节点

```
[root@B ~]# rabbitmqctl stop_app
[root@B ~]# rabbitmqctl join_cluster rabbit@A
[root@B ~]# rabbitmqctl start_app
```

### 5. 查看节点状态

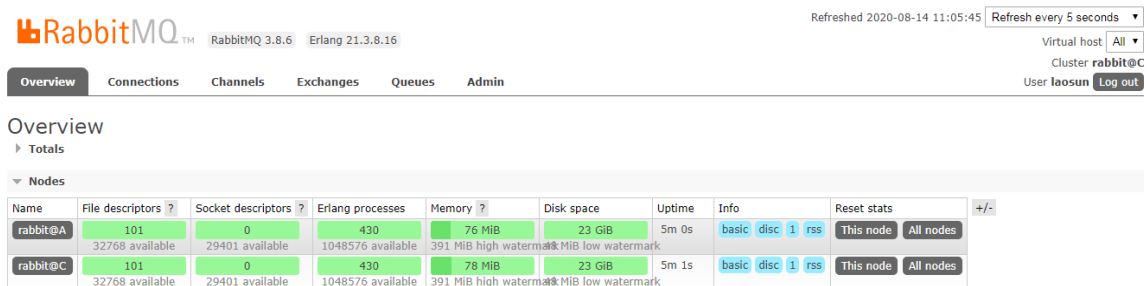
```
[root@B ~]# rabbitmqctl cluster_status
```

## 6. 查看管理端

- 搭建集群结构之后，之前创建的交换机、队列、用户都属于单一结构，在新的集群环境中是不能用的
- 所以在新的集群中**重新手动添加用户**即可（任意节点添加，所有节点共享）

```
[root@A ~]# rabbitmqctl add_user laosun 123123
[root@A ~]# rabbitmqctl set_user_tags laosun administrator
[root@A ~]# rabbitmqctl set_permissions -p "/" laosun ".*" ".*" ".*"
```

- 注意：当节点脱离集群还原成单一结构后，交换机，队列和用户等数据都会重新回来



The screenshot shows the RabbitMQ Management UI. At the top, it says 'RabbitMQ 3.8.6 Erlang 21.3.8.16'. The 'Overview' tab is selected. Below the navigation bar, there's a table titled 'Nodes' showing the status of two nodes: 'rabbit@A' and 'rabbit@C'. Both nodes are in a 'running' state. The table includes columns for Name, File descriptors, Socket descriptors, Erlang processes, Memory, Disk space, Uptime, Info, and Reset stats. For 'rabbit@A', File descriptors are 101 (32768 available), Socket descriptors are 0 (29401 available), Erlang processes are 430 (1048576 available), Memory is 76 MiB (391 MiB high watermark, 23 GiB low watermark), Disk space is 23 GiB, Uptime is 5m 0s, and Info shows 'basic disc 1 rss'. For 'rabbit@C', File descriptors are 101 (32768 available), Socket descriptors are 0 (29401 available), Erlang processes are 430 (1048576 available), Memory is 78 MiB (391 MiB high watermark, 23 GiB low watermark), Disk space is 23 GiB, Uptime is 5m 1s, and Info shows 'basic disc 1 rss'. There are buttons for 'This node' and 'All nodes' for each node's info.

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats
rabbit@A	101 32768 available	0 29401 available	430 1048576 available	76 MiB 391 MiB high watermark 23 GiB low watermark	23 GiB	5m 0s	basic disc 1 rss	This node All nodes
rabbit@C	101 32768 available	0 29401 available	430 1048576 available	78 MiB 391 MiB high watermark 23 GiB low watermark	23 GiB	5m 1s	basic disc 1 rss	This node All nodes

- 此时，集群搭建完毕，但是默认采用的模式“普通模式”，可靠性不高

## 3.2 镜像模式

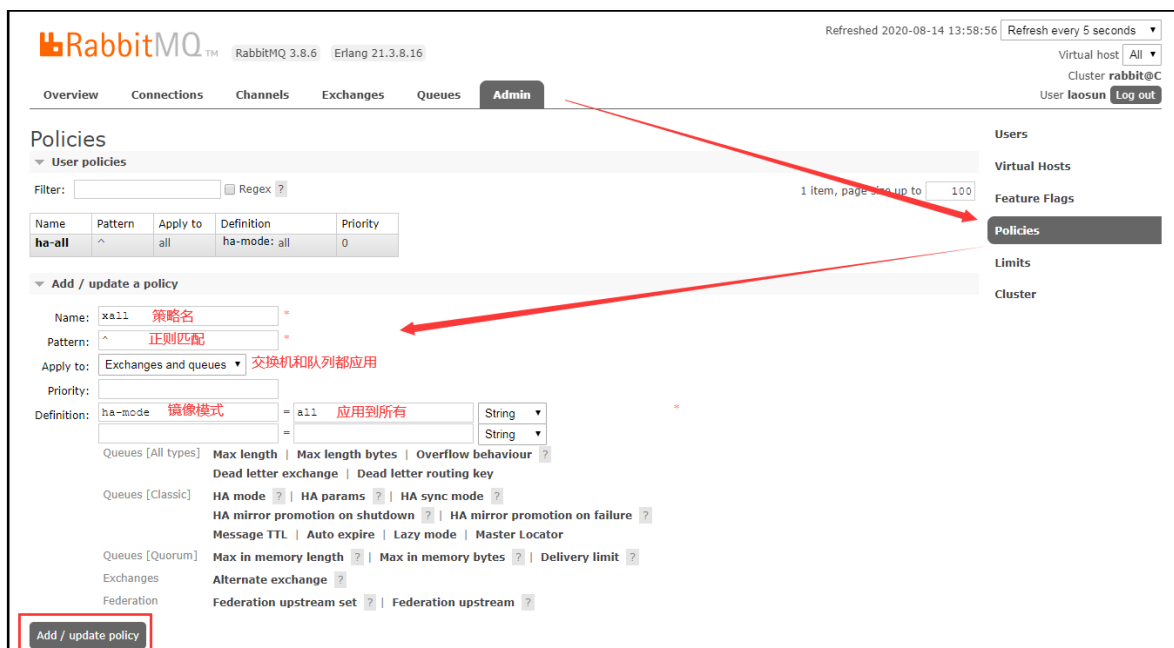
- 将所有队列设置为镜像队列，即队列会被复制到各个节点，各个节点状态一致
  - 语法：set\_policy {name} {pattern} {definition}
    - name：策略名，可自定义
    - pattern：队列的匹配模式（正则表达式）
      - "^" 可以使用正则表达式，比如"^queue\_" 表示对队列名称以“queue\_”开头的队列进行镜像，而"@"表示匹配所有的队列
    - definition：镜像定义，包括三个部分ha-mode, ha-params, ha-sync-mode
      - ha-mode：（**H**igh **A**vailable，高可用）模式，指明镜像队列的模式，有效值为all/exactly/nodes，当前策略模式为 all，即复制到所有节点，包含新增节点

all：表示在集群中所有的节点上进行镜像  
exactly：表示在指定个数的节点上进行镜像，节点的个数由ha-params指定  
nodes：表示在指定的节点上进行镜像，节点名称通过ha-params指定

- ha-params：ha-mode模式需要用到的参数
- ha-sync-mode：进行队列中消息的同步方式，有效值为automatic和manual

```
[root@A ~]# rabbitmqctl set_policy xall "^" '{"ha-mode":"all"}'
```

- 通过管理端设置镜像策略

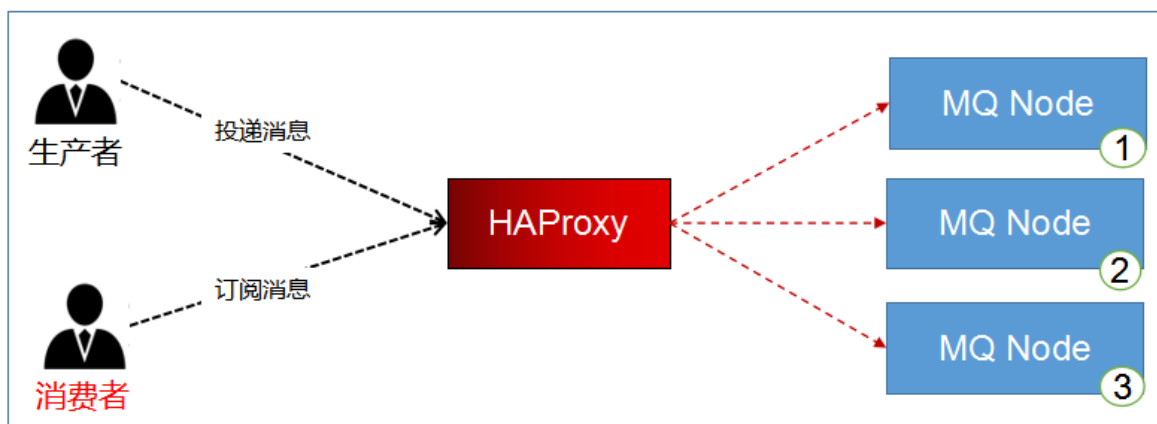


## 3.3 HAProxy实现镜像队列的负载均衡

- 虽然我们在程序中访问A服务器，可以实现消息的同步，虽然在同步，但都是A服务器在接收消息，A太累
- 是否可以想Nginx一样，做负载均衡，A和B轮流接收消息，再镜像同步

### 3.3.1 HAProxy简介

- HA ( High Available, 高可用 ) , Proxy ( 代理 )
- HAProxy是一款提供高可用性，负载均衡，并且基于TCP和HTTP应用的代理软件
- HAProxy完全免费
- HAProxy可以支持数以万计的并发连接
- HAProxy可以简单又安全的整合进架构中，同时还保护web服务器不被暴露到网络上



### 3.3.2 HAProxy与Nginx

OSI : ( Open System Interconnection : 开放式系统互联 是把网络通信的工作分为7层,分别是物理层,数据链路层,网络层,传输层,会话层,表示层和应用层 )

- Nginx的优点 :

- 工作在OS第7层，可以针对http应用做一些分流的策略
  - Nginx对网络的依赖非常小，理论上能ping通就能进行负载功能，屹立至今的绝对优势
  - Nginx安装和配置比较简单，测试起来比较方便；
  - Nginx不仅仅是一款优秀的负载均衡器/反向代理软件，它同时也是功能强大的Web应用服务器
- HAProxy的优点：
  - 工作在网络4层和7层，支持TCP与Http协议，
  - 它仅仅就只是一款负载均衡软件；单纯从效率上来讲HAProxy更会比Nginx有更出色的负载均衡速度，在并发处理上也是优于Nginx的
  - 支持8种负载均衡策略，支持心跳检测
- 性能上HA胜，功能性和便利性上Nginx胜
- 对于Http协议，Haproxy处理效率比Nginx高。所以，没有特殊要求的时候或者一般场景，建议使用Haproxy来做Http协议负载
- 但如果是Web应用，那么建议使用Nginx！
- 总之，大家可以结合各自使用场景的特点来进行合理地选择

### 3.3.3 安装和配置

HAProxy下载：<http://www.haproxy.org/download/1.8/src/haproxy-1.8.12.tar.gz>

- 解压

```
[root@localhost opt]# tar -zxvf haproxy-1.8.12.tar.gz
```

- make时需要使用 TARGET 指定内核及版本

```
[root@localhost opt]# uname -r
3.10.0-514.6.2.el7.x86_64
```

根据内核版本选择编译参数：

To build haproxy, you have to choose your target OS amongst the following ones and assign it to the TARGET variable :

- linux22	for Linux 2.2
- linux24	for Linux 2.4 and above (default)
- linux24e	for Linux 2.4 with support for a working epoll (> 0.21)
- linux26	for Linux 2.6 and above
- linux2628	for Linux 2.6.28, 3.x and above (enables splice and tproxy)
- solaris	for Solaris 8 or 10 (others untested)
- freebsd	for FreeBSD 5 to 10 (others untested)
- netbsd	for NetBSD
- osx	for Mac OS/X
- openbsd	for OpenBSD 5.7 and above
- aix51	for AIX 5.1
- aix52	for AIX 5.2
- cygwin	for Cygwin
- haiku	for Haiku

- 进入目录，编译和安装

```
[root@localhost opt]# cd haproxy-1.8.12
[root@localhost haproxy-1.8.12]# make TARGET=linux2628
PREFIX=/usr/local/haproxy
[root@localhost haproxy-1.8.12]# make install PREFIX=/usr/local/haproxy
```

- 安装成功后，查看版本

```
[root@localhost haproxy-1.8.12]# /usr/local/haproxy/sbin/haproxy -v
```

- 配置启动文件，复制haproxy文件到/usr/sbin下，复制haproxy脚本，到/etc/init.d下

```
[root@localhost haproxy-1.8.12]# cp /usr/local/haproxy/sbin/haproxy
/usr/sbin/
[root@localhost haproxy-1.8.12]# cp ./examples/haproxy.init
/etc/init.d/haproxy
[root@localhost haproxy-1.8.12]# chmod 755 /etc/init.d/haproxy
```

- 创建系统账号

```
[root@localhost haproxy-1.8.12]# useradd -r haproxy
```

- haproxy.cfg 配置文件需要自行创建

```
[root@localhost haproxy-1.8.12]# mkdir /etc/haproxy
[root@localhost haproxy-1.8.12]# vim /etc/haproxy/haproxy.cfg
```

- 添加配置信息到haproxy.cfg

```
#全局配置
global
    #设置日志
    log 127.0.0.1 local0 info
    #当前工作目录
    chroot /usr/local/haproxy
    #用户与用户组
    user haproxy
    group haproxy
    #运行进程ID
    uid 99
    gid 99
    #守护进程启动
    daemon
    #最大连接数
    maxconn 4096

#默认配置
defaults
    #应用全局的日志配置
    log global
    #默认的模式mode {tcp|http|health}，TCP是4层，HTTP是7层，health只返回OK
    mode tcp
    #日志类别tcplog
    option tcplog
    #不记录健康检查日志信息
```

```

option dontlognull
#3次失败则认为服务不可用
retries 3
#每个进程可用的最大连接数
maxconn 2000
#连接超时
timeout connect 5s
#客户端超时30秒，ha就会发起重新连接
timeout client 30s
#服务端超时15秒，ha就会发起重新连接
timeout server 15s

#绑定配置
listen rabbitmq_cluster
    bind 192.168.204.143:5672
    #配置TCP模式
    mode tcp
    #简单的轮询
    balance roundrobin
    #RabbitMQ集群节点配置，每隔5秒对mq集群做检查，2次正确证明服务可用，3次失败证明服务不可用
    server A 192.168.204.141:5672 check inter 5000 rise 2 fall 3
    server B 192.168.204.142:5672 check inter 5000 rise 2 fall 3

#haproxy监控页面地址
listen monitor
    bind 192.168.204.143:8100
    mode http
    option httplog
    stats enable
    # 监控页面地址 http://192.168.204.143:8100/monitor
    stats uri /monitor
    stats refresh 5s

```

- 启动HAProxy

```
[root@localhost haproxy]# service haproxy start
```

- 访问监控中心：<http://192.168.204.143:8100/monitor>

记得关闭防火墙：`systemctl stop firewallld`

**HAProxy version 1.8.12-8a200c7, released 2018/06/27**

**Statistics Report for pid 1698**

**> General process information**

pid = 1698 (process #1, nproc = 1, nbthread = 1)  
 uptime = 0s 0m0s4m47s  
 system limits: memmax = unlimited; ulimit-n = 8206  
 maxsock = 8206; maxconn = 4096; maxpipes = 0  
 current conns = 2; current pipes = 0/0; conn rate = 1/sec  
 Running tasks: 1/6; idle = 100 %

Legend:  
 active UP  
 active UP, going down  
 active DOWN, going up  
 active or backup DOWN  
 active or backup DOWN for maintenance (MAINT)  
 active or backup SOFT STOPPED for maintenance  
 backup UP  
 backup UP, going down  
 backup DOWN, going up  
 not checked

Display option:  
 Scope:   
 Hide DOWN servers  
 Disable refresh  
 Refresh now  
 CSV export

External resources:  
 Primary site  
 Updates (v1.8)  
 Online manual

rabbitmq_cluster		Queue			Session rate			Sessions			Bytes			Denied			Errors			Warnings			Server									
		Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle	
Frontend		0	1	-	0	1	-	0	1	2 000	50	1	2m42s	1 163	750	0	0	0	0	0	0	0	OPEN									
A		0	0	-	0	1	-	0	1	-	1	1	2m42s	1 163	750	0	0	0	0	1	0	0	4m47s UP	L4OK in 0ms	1	Y	-	0	0	0s	-	
B		0	0	-	0	0	-	0	0	-	0	0	?	0	0	0	0	0	0	0	0	4m47s UP	L4OK in 0ms	1	Y	-	0	0	0s	-		
Backend		0	0	-	0	1	-	0	1	200	1	1	2m42s	1 163	750	0	0	0	0	1	0	0	4m47s UP		2	2	0	0	0	0s		

monitor		Queue			Session rate			Sessions			Bytes			Denied			Errors			Warnings			Server									
		Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle	
Frontend		1	2	-	2	2	-	2	2	2 000	50			45 960	792 336	0	0	0	0	0	0	0	OPEN									
Backend		0	0	-	0	1	-	0	1	200	48	0	0s	45 960	792 336	0	0	0	48	0	0	0	4m47s UP		0	0	0	0	0	0		

- 项目发消息，只需要将服务器地址修改为143即可，其余不变

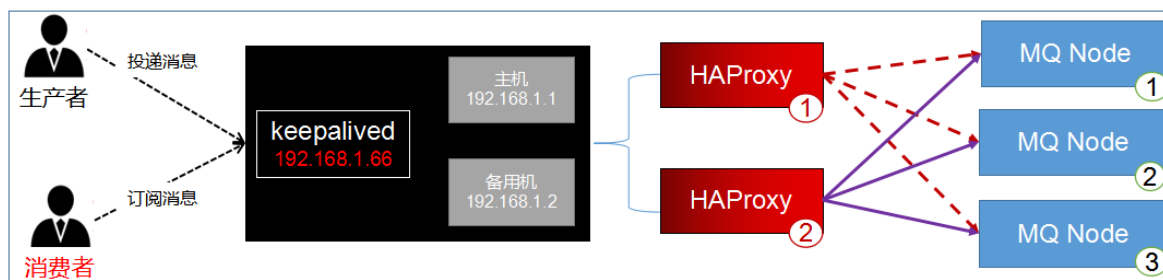
- 所有的请求都会交给HAProxy，其负载均衡给每个rabbitmq服务器

## 3.4 KeepAlived搭建高可用的HAProxy集群

现在的最后一个问题暴露出来了，如果HAProxy服务器宕机，rabbitmq服务器就不可用了。所以我们需要对HAProxy也要做高可用的集群

### 3.4.1 概述

- Keepalived是Linux下一个轻量级别的高可用热备解决方案
- Keepalived的作用是检测服务器的状态，它根据TCP/IP参考模型的第三、第四层、第五层交换机检测每个服务节点的状态，如果有一台web服务器宕机，或工作出现故障，Keepalived将检测到，并将有故障的服务器从系统中剔除，同时使用其他服务器代替该服务器的工作，当服务器工作正常后Keepalived自动将服务器加入到服务器群中，这些工作全部自动完成，不需要人工干涉，需要人工做的只是修复故障的服务器。
- keepalived基于vrrp ( **V**irtual **R**outer **R**edundancy **P**rotocol，虚拟路由冗余协议 ) 协议，vrrp它是一种 **主备** ( 主机和备用机 ) 模式的协议，通过VRRP可以在网络发生故障时**透明的进行设备切换**而不影响主机之间的数据通信
- 两台主机之间生成一个虚拟的ip，我们称漂移ip，漂移ip由主服务器承担，一但主服务器宕机，备份服务器就会抢夺漂移ip，继续工作，有效的解决了群集中的单点故障
- 说白了，将多台路由器设备**虚拟**成一个设备，对外提供统一ip ( VIP )



### 3.4.2 安装KeepAlived

- 修改hosts文件的地址映射

ip	用途	主机名
192.168.204.143	KeepAlived HAProxy	C
192.168.204.144	KeepAlived HAProxy	D

- 安装 keepalived

```
[root@C ~]# yum install -y keepalived
```

- 修改配置文件 ( 内容大改，不如删掉，重新创建 )



```
[root@C ~]# rm -rf /etc/keepalived/keepalived.conf
[root@C ~]# vim /etc/keepalived/keepalived.conf
```

```
! Configuration File for keepalived

global_defs {
    router_id C          ## 非常重要，标识本机的hostname
}

vrrp_script chk_haproxy{
    script "/etc/keepalived/haproxy_check.sh"      ## 执行的脚本位置
    interval 2          ## 检测时间间隔
    weight -20          ## 如果条件成立则权重减20
}

vrrp_instance VI_1 {
    state MASTER          ## 非常重要，标识主机，备用机143改为 BACKUP
    interface ens33       ## 非常重要，网卡名（ifconfig查看）
    virtual_router_id 66  ## 非常重要，自定义，虚拟路由ID号（主备节点要相同）
    priority 100          ## 优先级（0-254），一般主机的大于备机
    advert_int 1          ## 主备信息发送间隔，两个节点必须一致，默认1秒
    authentication {     ## 认证匹配，设置认证类型和密码，MASTER和BACKUP必须使
        用相同的密码才能正常通信
        auth_type PASS
        auth_pass 1111
    }
    track_script {
        chk_haproxy      ## 检查haproxy健康状况的脚本
    }
    virtual_ipaddress {   ## 简称“VIP”
        192.168.204.66/24 ## 非常重要，虚拟ip，可以指定多个，以后连接mq就用这个虚
        拟ip
    }
}

virtual_server 192.168.204.66 5672 {    ## 虚拟ip的详细配置
    delay_loop 6          # 健康检查间隔，单位为秒
    lb_algo rr            # lvs调度算法rr|wrr|lc|wlc|lb|c|sh|dh
    lb_kind NAT           # 负载均衡转发规则。一般包括DR,NAT,TUN 3种
    protocol TCP          # 转发协议，有TCP和UDP两种，一般用TCP
    real_server 192.168.204.143 5672 {  ## 本机的真实ip
        weight 1          # 默认为1,0为失效
    }
}
```

- 创建执行脚本 /etc/keepalived/haproxy\_check.sh

```
#!/bin/bash
COUNT=`ps -C haproxy --no-header |wc -l`
if [ $COUNT -eq 0 ];then
    /usr/local/haproxy/sbin/haproxy -f /etc/haproxy/haproxy.cfg
    sleep 2
    if [ `ps -C haproxy --no-header |wc -l` -eq 0 ];then
        killall keepalived
    fi
fi
```

keepalived 组之间的心跳检查并不能察觉到 HAProxy 负载是否正常，所以需要使用此脚本。  
在 Keepalived 主机上，开启此脚本检测 HAProxy 是否正常工作，如正常工作，记录日志。  
如进程不存在，则尝试重启 HAProxy，2秒后检测，如果还没有，则关掉主 keepalived，此时备 keepalived 检测到主 keepalived 挂掉，接管VIP，继续服务

- 授权，否则不能执行

```
[root@C etc]# chmod +x /etc/keepalived/haproxy_check.sh
```

- 启动keepalived（两台都启动）

```
[root@C etc]# systemctl stop firewalld
[root@C etc]# service keepalived start | stop | status | restart
```

- 查看状态

```
[root@C etc]# ps -ef | grep haproxy
[root@C etc]# ps -ef | grep keepalived
```

- 查看ip情况 ip addr 或 ip a

```
[root@C etc]# ip a
```

```
[root@C haproxy-1.8.12]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:bf:8a:6e brd ff:ff:ff:ff:ff:ff
    ip: inet 192.168.204.143/24 brd 192.168.204.255 scope global ens33
        valid_lft forever preferred_lft forever
    vip: inet 192.168.204.66/24 scope global secondary ens33
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:febf:8a6e/64 scope link
        valid_lft forever preferred_lft forever
```

- 此时，安装完毕，按照上面的步骤就可以安装第二台了（服务器hostname和ip注意要修改）

- 常见的网络错误：子网掩码、网关等信息要一致

```
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 00:0c:29:bf:8a:6e brd ff:ff:ff:ff:ff:ff
    inet 192.168.204.143/24 brd 192.168.204.255 scope global ens33
        valid_lft forever preferred_lft forever
    inet 192.168.204.66/24 scope global secondary ens33
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:febf:8a6e/64 scope link
        valid_lft forever preferred_lft forever
```

### 3.4.3 测试ip漂移的规则

- 查看虚拟ip `ip addr` 或 `ip a`
- 目前，C节点是主机，所以虚拟ip在C节点

<pre>[root@C haproxy-1.8.12]# ip a 1: lo: &lt;LOOPBACK,UP,LOWER_UP&gt; mtu 65536 qdisc noqu     link/loopback 00:00:00:00:00:00 brd 00:00:00:00     inet 127.0.0.1/8 scope host lo         valid_lft forever preferred_lft forever     inet6 ::1/128 scope host         valid_lft forever preferred_lft forever 2: ens33: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 15     link/ether 00:0c:29:bf:8a:6e brd ff:ff:ff:ff:ff:f     inet 192.168.204.143/24 brd 192.168.204.255 sc         valid_lft forever preferred_lft forever     inet 192.168.204.66/24 scope global secondary         valid_lft forever preferred_lft forever     inet6 fe80::20c:29ff:febf:8a6e/64 scope link         valid_lft forever preferred_lft forever</pre> <p style="text-align: center;"><b>主机</b></p>	<pre>[root@D keepalived-1.2.18]# service keepalived stop Stopping keepalived (via systemctl): [root@D keepalived-1.2.18]# ip a 1: lo: &lt;LOOPBACK,UP,LOWER_UP&gt; mtu 65536 qdisc noqueu     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:     inet 127.0.0.1/8 scope host lo         valid_lft forever preferred_lft forever     inet6 ::1/128 scope host         valid_lft forever preferred_lft forever 2: ens33: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 1500     link/ether 00:0c:29:e4:20:e9 brd ff:ff:ff:ff:ff:ff     inet 192.168.204.144/23 brd 192.168.205.255 scop         valid_lft forever preferred_lft forever     inet6 fe80::20c:29ff:fee4:20e9/64 scope link         valid_lft forever preferred_lft forever</pre> <p style="text-align: center;"><b>备机</b></p>
---	---

- 停止C的keepalived，虚拟ip漂移到D节点

<pre>[root@C haproxy-1.8.12]# service keepalived stop Redirecting to /bin/systemctl stop keepalived.serv [root@C haproxy-1.8.12]# ip a 1: lo: &lt;LOOPBACK,UP,LOWER_UP&gt; mtu 65536 qdisc noqu     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:     inet 127.0.0.1/8 scope host lo         valid_lft forever preferred_lft forever     inet6 ::1/128 scope host         valid_lft forever preferred_lft forever 2: ens33: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 15     link/ether 00:0c:29:bf:8a:6e brd ff:ff:ff:ff:ff:f     inet 192.168.204.143/24 brd 192.168.204.255 s         valid_lft forever preferred_lft forever     inet6 fe80::20c:29ff:febf:8a6e/64 scope link         valid_lft forever preferred_lft forever</pre> <p style="text-align: center;"><b>主机</b></p>	<pre>[root@D ~]# ip a 1: lo: &lt;LOOPBACK,UP,LOWER_UP&gt; mtu 65536 qdisc noc     link/loopback 00:00:00:00:00:00 brd 00:00:00:     inet 127.0.0.1/8 scope host lo         valid_lft forever preferred_lft forever     inet6 ::1/128 scope host         valid_lft forever preferred_lft forever 2: ens33: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu :     link/ether 00:0c:29:e4:20:e9 brd ff:ff:ff:ff:ff:ff     inet 192.168.204.144/24 brd 192.168.204.255 s         valid_lft forever preferred_lft forever     inet 192.168.204.66/24 scope global secondary         valid_lft forever preferred_lft forever     inet6 fe80::20c:29ff:fee4:20e9/64 scope link         valid_lft forever preferred_lft forever</pre> <p style="text-align: center;"><b>备机</b></p>
---	--

- 重新启动C节点keepalived，虚拟ip依旧在D节点，并不会由于C的回归而回归

<pre>[root@C haproxy-1.8.12]# service keepalived start Redirecting to /bin/systemctl start keepalived.serv [root@C haproxy-1.8.12]# ip a 1: lo: &lt;LOOPBACK,UP,LOWER_UP&gt; mtu 65536 qdisc noque     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:     inet 127.0.0.1/8 scope host lo         valid_lft forever preferred_lft forever     inet6 ::1/128 scope host         valid_lft forever preferred_lft forever 2: ens33: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 150     link/ether 00:0c:29:bf:8a:6e brd ff:ff:ff:ff:ff:ff     inet 192.168.204.143/24 brd 192.168.204.255 scc         valid_lft forever preferred_lft forever     inet6 fe80::20c:29ff:febf:8a6e/64 scope link         valid_lft forever preferred_lft forever</pre> <p style="text-align: center;"><b>主机</b></p>	<pre>[root@D ~]# ip a 1: lo: &lt;LOOPBACK,UP,LOWER_UP&gt; mtu 65536 qdisc noc     link/loopback 00:00:00:00:00:00 brd 00:00:00:     inet 127.0.0.1/8 scope host lo         valid_lft forever preferred_lft forever     inet6 ::1/128 scope host         valid_lft forever preferred_lft forever 2: ens33: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu :     link/ether 00:0c:29:e4:20:e9 brd ff:ff:ff:ff:ff:ff     inet 192.168.204.144/24 brd 192.168.204.255 s         valid_lft forever preferred_lft forever     inet 192.168.204.66/24 scope global secondary         valid_lft forever preferred_lft forever     inet6 fe80::20c:29ff:fee4:20e9/64 scope link         valid_lft forever preferred_lft forever</pre> <p style="text-align: center;"><b>备机</b></p>
--	--

- 停止D的keepalived，虚拟ip再漂移回C节点

<pre>[root@C haproxy-1.8.12]# ip a 1: lo: &lt;LOOPBACK,UP,LOWER_UP&gt; mtu 65536 qdisc noqueu    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00    inet 127.0.0.1/8 scope host lo        valid_lft forever preferred_lft forever    inet6 ::1/128 scope host        valid_lft forever preferred_lft forever 2: ens33: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 15    link/ether 00:0c:29:bf:8a:6e brd ff:ff:ff:ff:ff:ff    inet 192.168.204.143/24 brd 192.168.204.255 sc        valid_lft forever preferred_lft forever    inet 192.168.204.66/24 scope global secondary        valid_lft forever preferred_lft forever    inet6 fe80::20c:29ff:febf:8a6e/64 scope link        valid_lft forever preferred_lft forever</pre>	<pre>[root@D keepalived-1.2.18]# service keepalived stop Stopping keepalived (via systemctl): [root@D keepalived-1.2.18]# ip a 1: lo: &lt;LOOPBACK,UP,LOWER_UP&gt; mtu 65536 qdisc noqueu    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00    inet 127.0.0.1/8 scope host lo        valid_lft forever preferred_lft forever    inet6 ::1/128 scope host        valid_lft forever preferred_lft forever 2: ens33: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 1500    link/ether 00:0c:29:e4:20:e9 brd ff:ff:ff:ff:ff:ff    inet 192.168.204.144/23 brd 192.168.205.255 scop        valid_lft forever preferred_lft forever    inet6 fe80::20c:29ff:fee4:20e9/64 scope link        valid_lft forever preferred_lft forever</pre>
主机	备机

- 测试vip+端口是否提供服务（在141，A服务器上测试）

```
[root@A ~]# curl 192.168.204.66:5672
AMQP      ## 正常提供AMQP服务，表示通过vip访问mq服务正常
```