

# RabbitMq

## 消息中间件介绍&为什么要使用消息中间件&什么时候使用消息中间件

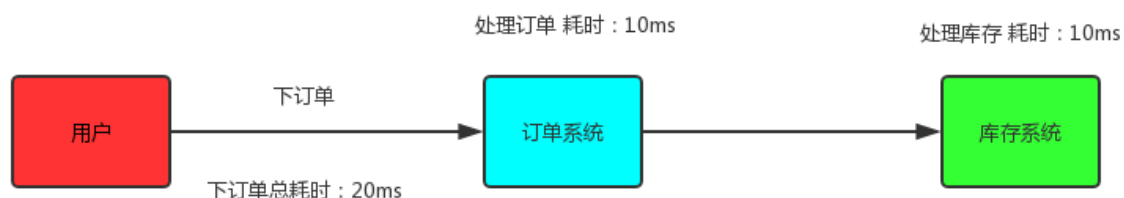
我们用java来举例子，打个比方 我们客户端发送一个下单请求给订单系统（order）订单系统发送了

一个请求给我们的库存系统告诉他需要更改库存了，我已经下单了，这里，每一个请求我们都可以看作一条消息，

但是 我们客户端需要等待订单系统告诉我这条消息的处理结果（我到底有没有下单成功）但是 订单系统不需要知道库存系统这条消息的处理情况 因为无论你库存有没有改动成功，我订单还是下了，因为是先下完了订单（下成功了）才去更改库存，库存如果更改出BUG了 那是库存系统的问题，这个BUG不会影响订单系统。如果这里你能理解的话，那么我们就能发现 **我们用户发送的这条消息（下订单），是需要同步的（我需要知道结果），订单发送给库存的消息，是可以异步的（我不想知道你库存到底改了没，我只是通知你我这边成功下了一个订单）**

那么如果我们还按原来的方式去实现这个需求的话，那么结果会是这样：

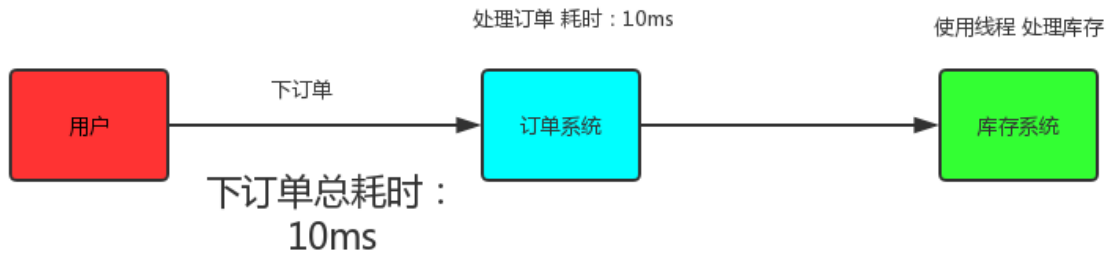
原来的实现方案:用户下订单 订单处理完之后更改库存 更改完库存之后返回处理结果



问题：笔记中有说到，用户实际上只需要得到订单系统的返回信息，但是这里其实是在等待订单和库存的处理结果，所以是没必要的，而且库存系统出现问题会影响用户所看到的结果（报错）其实对于用户来讲 不关心库存 只关心订单

那可能有同学说了，我们订单系统开辟线程去访问库存系统不就好了吗？

使用线程的解决方案：用户下订单 订单系统开辟线程去调用库存 同时返回下订单的信息

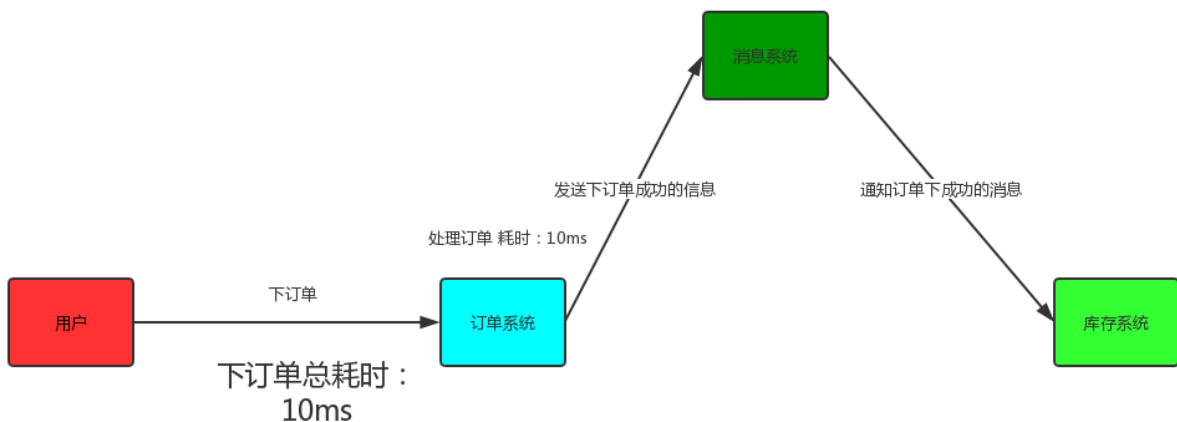


问题：我们能发现，现在用户不用等库存的处理结果了，而且库存也不会影响用户下订单了，但是 我们需要管理线程池，代码耦合度严重。

##

使用线程池解决 确实可以，但是也有他的缺点，那么 到底怎么来完美解决这个问题呢？

最终方案：使用一个独立的系统来处理他们之间的消息

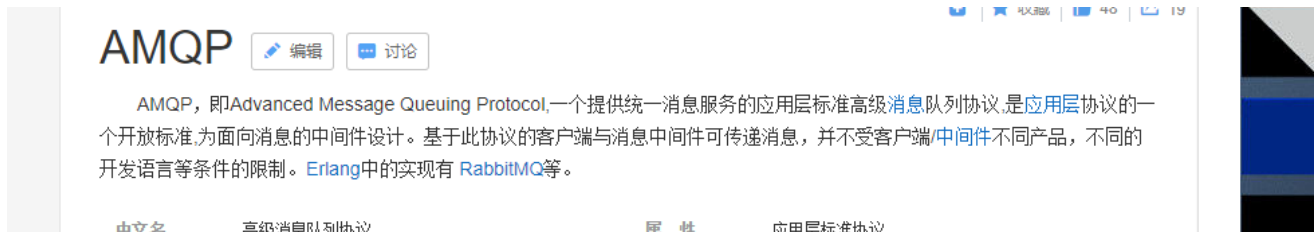


如果这张图能理解的话，那么 这个消息系统，就是我们的消息中间件。

# RabbitMq介绍&AMQP介绍

导语:我们刚刚介绍了什么是消息中间件, 那么RabbitMq就是对于消息中间件的一种实现, 市面上还有很多很多实现, 比如RabbitMq、ActiveMq、ZeroMq、kafka, 以及阿里开源的RocketMQ等等 我们这节主要讲RabbitMq

## AMQP



这里引用百度的一句话 再加以我的理解: AMQP 其实和Http一样 都是一种协议, 只不过 Http是针对网络传输的, 而AMQP是基于消息队列的

AMQP 协议中的基本概念:

- **Broker:** 接收和分发消息的应用, 我们在介绍消息中间件的时候所说的消息系统就是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RabbitMQ 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。
- **Connection:** publisher / consumer和broker之间的TCP连接。断开连接的操作只会在client端进行, Broker不会断开连接, 除非出现网络故障或broker服务出现问题。
- **Channel:** 如果每一次访问RabbitMQ都建立一个Connection, 在消息量大的时候建立TCP Connection的开销将是巨大的, 效率也较低。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TCP 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) and fanout (multicast)。
- **Queue:** 消息最终被送到这里等待consumer取走。一个message可以被同时拷贝到多个queue中。
- **Binding:** exchange和queue之间的虚拟连接, 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。

## Exchange的类型:

### direct :

这种类型的交换机的路由规则是根据一个routingKey的标识, 交换机通过一个routingKey与队列绑定, 在生产者生产消息的时候 指定一个routingKey 当绑定的队列的routingKey 与生产者发送的一样 那么交换机会把这个消息发送给对应的队列。

### fanout:

这种类型的交换机路由规则很简单，只要与他绑定了的队列，他就会把消息发送给对应队列（与routingKey没关系）

**topic:(因为\*在这个笔记软件里面是关键字，所以下面就用星替换掉了)**

这种类型的交换机路由规则也是和routingKey有关 只不过 topic他可以根据:星,#（星号代表过滤一单词，#代表过滤后面所有单词，用.隔开）来识别routingKey 我打个比方 假设 我绑定的routingKey 有队列A和B A的routingKey是: 星.user B的routingKey是: #.user

那么我生产一条消息routingKey 为: error.user 那么此时 2个队列都能接受到，如果改为 topic.error.user 那么这时候 只有B能接受到了

**headers:**

这个类型的交换机很少用到，他的路由规则 与routingKey无关 而是通过判断header参数来识别的，基本上没有应用场景，因为上面的三种类型已经能应付了。

## RabbitMQ

MQ: message Queue 顾名思义 消息队列，队列大家都知道，存放内容的一个东西，存放的内容先进先出，消息队列，只是里面存放的内容是消息而已。

RabbitMq 是一个开源的 基于AMQP协议实现的一个完整的企业级消息中间件，服务端语言由Erlang（面向并发编程）语言编写 对于高并发的处理有着天然的优势，客户端支持非常多的语言：

- Python
- Java
- Ruby
- PHP
- C#
- JavaScript
- Go
- Elixir
- Objective-C
- Swift

## 主流MQ对比

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，比RocketMQ、Kafka 低一个数量级	同ActiveMQ	10 万级，支撑高吞吐	10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic 数量对吞吐量的影响			topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic	topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量不要过多，如果要支撑大规模的 topic，需要增加更多的机器资源
时效性	毫秒级	微秒级，这是RabbitMQ 的一大特点，延迟最低	毫秒级	毫秒级
可用性	高，基于主从架构实现高可用	同ActiveMQ	非常高，分布式架构	非常高，分布式，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
消息可靠性	有较低的概率丢失数据		经过参数优化配置，可以做到 0 丢失	同 RocketMQ
功能支持	MQ 领域的功能极其完备	基于erlang 开发，并发能力很强，性能极好，延时很低	MQ 功能较为完善，还是分布式的，扩展性好	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

## RabbitMQ服务端部署

在介绍消息中间件的时候所提到的“消息系统”便是我们这节的主题：RabbitMq 如同redis一样 他也是采用c/s架构由服务端 与客户端组成， 我们现在我们计算机上部署他的服务端

由于我们刚刚介绍过了RabbitMQ服务端是由Erlang语言编写所以我们这里先下载Erlang语言的环境

注意：如果是在官网下的RabbitMQ服务端的话 Erlang语言的版本不能太低， 不然要卸载掉旧的去装新的， 我们这里下载OTP21.0版本直接从外网下载会很慢， 我这里直接贴上百度网盘的地址(因为这个东西还是有点大的)

<https://pan.baidu.com/s/1pZJ8l2f3omrgnuCm9a8DVA>

我们再去官网下载 他的服务端安装包

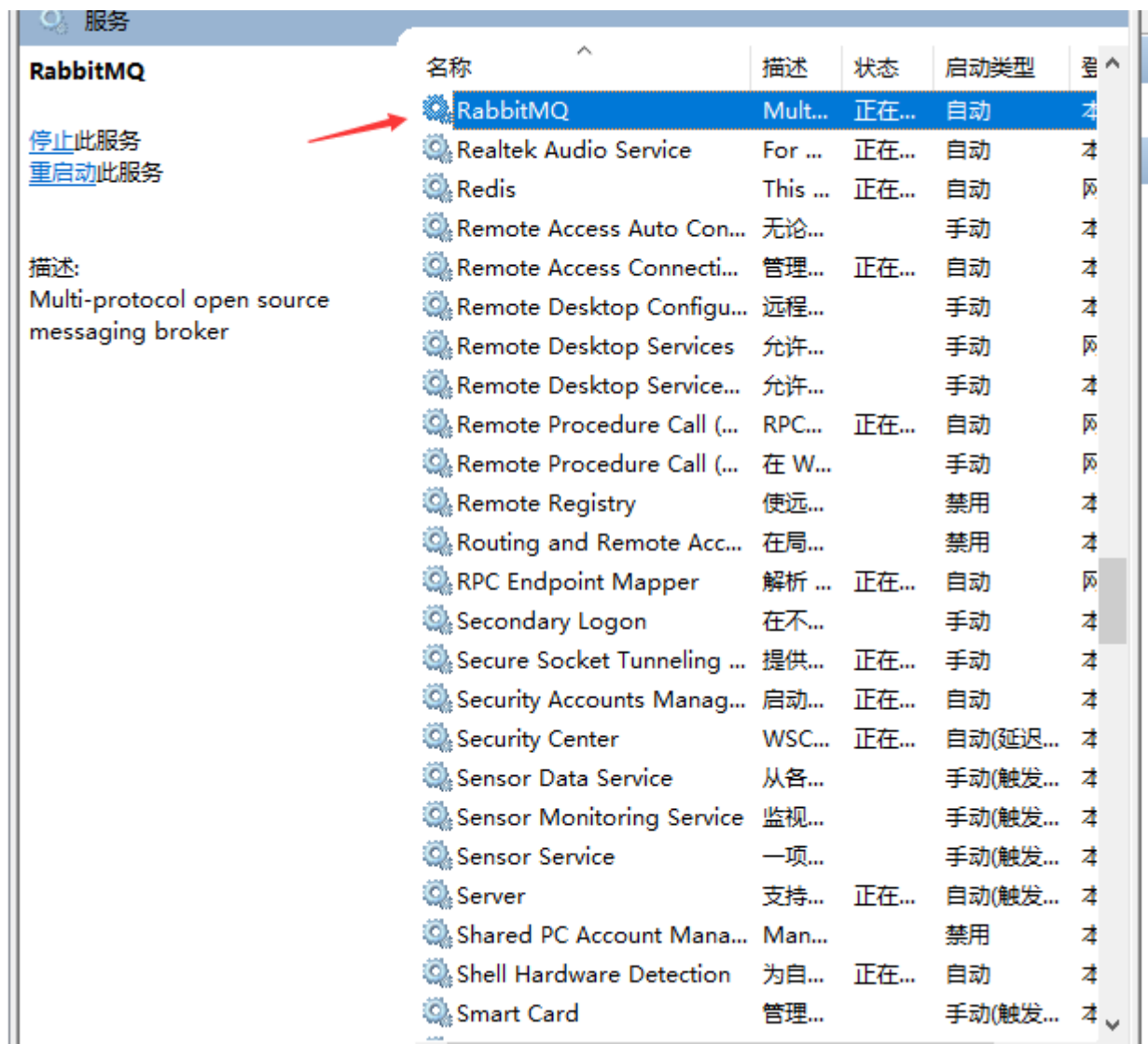
<http://www.rabbitmq.com/download.html>

根据自己的系统选择下载即可

**注意！** 需要先下载Erlang再下载安装包安装，不然安装RabbitMQ服务端的时候会提示你本地没有Erlang环境

**安装的话，基本上就是默认选项不用改**

如何看RabbitMq安装完成了？在系统-服务中找到如下即可：



包括启动 停止 重启 服务等

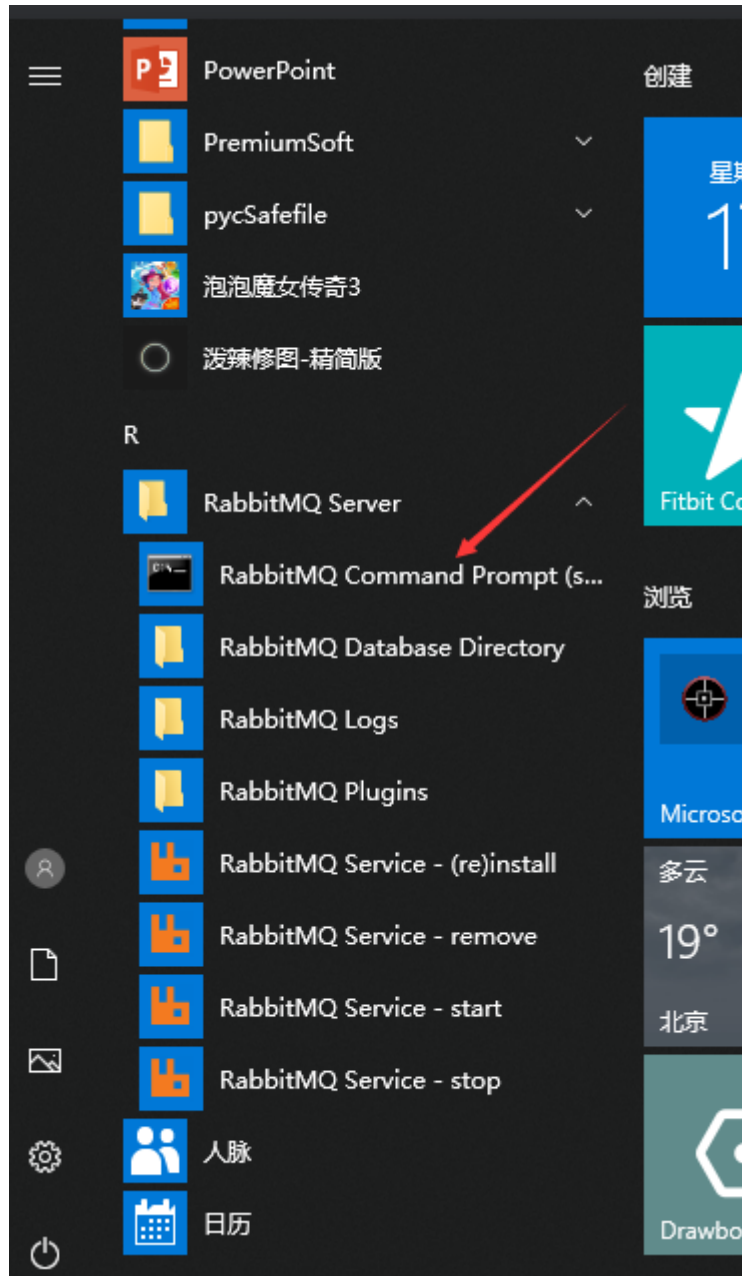
RabbitMQ安装会附带一个管理工具（方便我们能直观的查看整个RabbitMQ的运行状态和详细数据等，有点像Navicat 对应Mysql的关系）值得一提的是，管理工具和RabbitMQ是两码事 希望同学们不要混稀了。

管理工具启动方式：

到你们安装的 RabbitMQ Server\rabbitmq\_server-3.7.12\sbin 目录下面 执行一条cmd命令：

## rabbitmq-plugins enable rabbitmq\_management

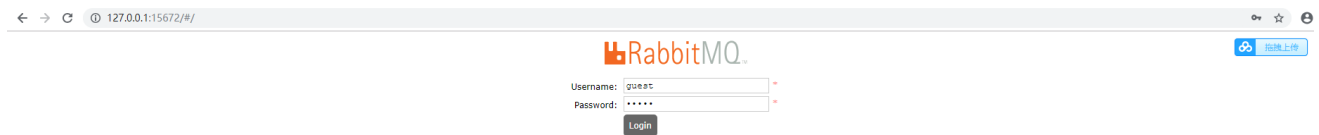
直接复制这条命令即可，当然嫌每次都要去目录中去执行的麻烦的话，可以配置一个环境变量 或者在我们的开始菜单栏中找到这个：



输入完启动命令后 稍微等一下会有结果返回 然后可以打开浏览器 输入

<http://127.0.0.1:15672>

访问管理页面：



默认账号密码都是

guest 即

username : guest

password: guest

登录进去之后会看到如下界面（因为我不小心装了2次RabbitMq 所以这里能看到都重复了，你们自己那不会重复，然后我们刚刚说了 管理工具和rabbitmq 是两码事 所以端口也就不一样）

Refreshed 2019-03-18 14:56:38 Refresh every 5 seconds

Virtual host All

Cluster rabbit@DESKTOP-2J0UGJ1

User guest Log out

Overview Connections Channels Exchanges Queues Admin

Message rates last minute ?

1.0/s

0.0/s

14:55:40 14:55:50 14:56:00 14:56:10 14:56:20 14:56:30

Disk read 0.00/s

Disk write 0.00/s

Global counts ?

Connections: 0 Channels: 0 Exchanges: 16 Queues: 2 Consumers: 0

Nodes

Churn statistics

Ports and contexts

Listening ports

Protocol	Bound to	Port
amqp	0.0.0.0	5672
amqp	::	5672
clustering	::	25672
http	0.0.0.0	15672
http	::	15672

Web contexts

Context	Bound to	Port	SSL	Path
RabbitMQ Management	0.0.0.0	15672	o	/

Export definitions

Import definitions

HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

这个页面在笔记里面介绍起来可能比较复杂，就不一一介绍了，我这里讲个重点，就是线上环境下一定要吧 guest用户（当然 guest这个用户只能本机才能登陆）删掉并且新加一个用户，这里就演示一下这个功能

首先 点击admin页签，在下面找到Add User



## Users

All users

Filter:  ☐ Regexp ?

2 items, page size up to 100

Name	Tags	Can access virtual hosts	Has password
admin	administrator	testhost	*
guest	administrator	/, testhost	*

?

Add a user

[HTTP API](#) [Server Docs](#) [Tutorials](#) [Community Support](#) [Community Slack](#) [Commercial Support](#) [Plugins](#) [GitHub](#) [Changelog](#)

## Users

Virtual Hosts

Policies

Limits

Cluster

然后输入账号 密码 确认密码 这个Tags其实是一个用户权限标签，关于他的介绍可以看官方介绍（点旁边那个小问号就好了，我这里直接翻译他的介绍）

## Add a user

Username: Password: 

(confirm)

Tags: 

Set

[Admin](#) | [Monitoring](#) | [Policymaker](#)  
[Management](#) | [Impersonator](#) | [None](#)

Add user

3 items,

Comma-separated list of tags to apply to the user. Currently **supported by the management plugin**:

**management**

User can access the management plugin

**policymaker**

User can access the management plugin and manage policies and parameters for the vhosts they have access to.

**monitoring**

User can access the management plugin and see all connections and channels as well as node-related information.

**administrator**

User can do everything monitoring can do, manage users, vhosts and permissions, close other user's connections, and manage policies and parameters for all vhosts.

Note that you can set any tag here; the links for the above four tags are just for convenience.

Close

以逗号分隔的标签列表，应用于用户。目前 由管理插件支持：

**管理**

用户可以访问管理插件

**政策制定者**

用户可以访问管理插件并管理他们有权访问的vhost的策略和参数。

**监控**

用户可以访问管理插件并查看所有连接和通道以及与节点相关的信息。

**管理员**

用户可以执行监控可以执行的所有操作，管理用户，虚拟主机和权限，关闭其他用户的连接以及管理所有虚拟主机的策略和参数。

请注意，您可以在此处设置任何标记；以上四个标签的链接只是为了方便。

关

填写完之后点击AddUser 就可以添加一个用户了， 添加完用户之后还要给这个用户添加对应的权限（注：Tag不等于权限）

比如说 我刚刚添加了一个jojo角色

## Users

▼ All users

Filter:  ☐ Regex ?

Name	Tags	Can access virtual hosts	Has password
admin	administrator	testhost	•
guest	administrator	/, testhost	•
jojo	administrator	testhost	•

?

点击这个jojo可以进去给他添加权限 这个权限可以是 Virtual host 级别的 也可以是交换机级别的 甚至是细化到某一个读写操作 我这里就给他添加一个Virtual host权限

## User: jojo

▼ Overview

Tags

administrator

Can log in with password

•

▼ Permissions

Current permissions

Virtual host	Configure regexp	Write regexp	Read regexp	
testhost	.*	.*	.*	Clear

Set permission

Virtual Host:

testhost ▼

Configure regexp:

.\*

Write regexp:

.\*

Read regexp:

.\*

Set permission

▼ Topic permissions

Current topic permissions

... no topic permissions ...

Set topic permission

Virtual Host:

/ ▼

Exchange:

(AMQP default) ▼

Write regexp:

.\*

Read regexp:

.\*

这里 我们给了他 testhost这个Virtual host的权限 正则匹配都是\* 也就是所有权限

然后点击set添加完毕

那么管理页面 我们就讲到这里

## RabbitMq快速开始

因为我们这里是用java来作为客户端， 我们首先引入maven依赖：

```
<dependency>
  <groupId>com.rabbitmq</groupId>
  <artifactId>amqp-client</artifactId>
  <version>5.1.2</version>
</dependency>
```

（注意的是， 我这里引入的是5.x的rabbitmq客户端版本， 那么我们jdk的版本最好在8以上， 反之， 这里就建议使用4.x的版本， 这里仅仅讨论jdk8 其他的版本不做讨论）

首先 我们编写一个连接的工具类：

```

package com.luban.util;

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

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * 需要咨询java高级VIP课程的同学可以木兰老师的QQ: 2746251334
 * 需要往期视频的同学可以加加安其拉老师的QQ: 3164703201
 * author: 鲁班学院-商鞅老师
 */
public class ConnectionUtil {

    public static final String QUEUE_NAME = "testQueue";

    public static final String EXCHANGE_NAME = "exchange";

    public static Connection getConnection() throws Exception{
        //创建一个连接工厂
        ConnectionFactory connectionFactory = new ConnectionFactory();
        //设置rabbitmq 服务端所在地址 我这里在本地就是本地
        connectionFactory.setHost("127.0.0.1");
        //设置端口号, 连接用户名, 虚拟地址等
        connectionFactory.setPort(5672);
        connectionFactory.setUsername("jojo");
        connectionFactory.setPassword("jojo");
        connectionFactory.setVirtualHost("testhost");
        return connectionFactory.newConnection();
    }

}

```

然后我们编写一个消费者 (producer) , 和一个生产者 (consumer) :

生产者:

```

public class Consumer {

    public static void sendByExchange(String message) throws Exception {

        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();
    }
}

```

```

//声明队列
channel.queueDeclare(ConnectionUtil.QUEUE_NAME,true,false,false,null);
// 声明exchange
channel.exchangeDeclare(ConnectionUtil.EXCHANGE_NAME, "fanout");
//交换机和队列绑定
channel.queueBind(ConnectionUtil.QUEUE_NAME, ConnectionUtil.EXCHANGE_NAME, "");
channel.basicPublish(ConnectionUtil.EXCHANGE_NAME, "", null,
message.getBytes());
System.out.println("发送的信息:" + message);
channel.close();
connection.close();
}

}

```

消费者:

```

public class Producer {

    public static void getMessage() throws Exception {
        Connection connection = ConnectionUtil.getConnection();
        Channel channel = connection.createChannel();
//        channel.queueDeclare(ConnectionUtil.QUEUE_NAME,true,false,false,null);
        DefaultConsumer deliverCallback = new DefaultConsumer(channel) {
            @Override
            public void handleDelivery(String consumerTag, Envelope envelope,
AMQP.BasicProperties properties, byte[] body) throws IOException {
                System.out.println(new String(body, "UTF-8"));
            }
        };
        channel.basicConsume(ConnectionUtil.QUEUE_NAME, deliverCallback);
    }

}

```

这里，我们演示绑定fanout的类型的交换机，所以不需要routingKey 就可以路由只需要绑定即可

（可能有同学要问了，如果没有绑定交换机怎么办呢？没有绑定交换机的话，消息会发给rabbitmq默认的交换机里面 默认的交换机隐式的绑定了所有的队列，默认的交换机类型是direct 路由建就是队列的名字）

基本上这样的话就已经进行一个快速入门了，由于我们现在做项目基本上都是用spring boot（就算没用spring boot也用spring 吧）所以后面我们直接基于spring boot来讲解（rabbitmq的特性，实战等）

## spring boot 整合rabbitmq

spring boot的环境怎么搭建这边就不提了，这里引入spring boot -AMQP的依赖：

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

## 配置连接,创建交换机, 队列

首先 我和上面一样 先要配置连接信息这里 可以选用yml的方式 也可以选用javaConfig的方式 这里两种方式我都贴出来 你们自己选

yml: 参数什么意思刚刚介绍过了 这里吧你自己的参数填进去就好了

```
spring:
  rabbitmq:
    host:
    port:
    username:
    password:
    virtual-host:
```

这样 spring boot 会帮你把rabbitmq其他相关的东西都自动装备好,

javaConfig:

```
@Bean
public ConnectionFactory connectionFactory() {
    CachingConnectionFactory connectionFactory = new
    CachingConnectionFactory("localhost", 5672);
    //我这里直接在构造方法传入了
    //      connectionFactory.setHost();
    //      connectionFactory.setPort();
    connectionFactory.setUsername("admin");
    connectionFactory.setPassword("admin");
    connectionFactory.setVirtualHost("testhost");
    //是否开启消息确认机制
    //connectionFactory.setPublisherConfirms(true);
    return connectionFactory;
}
```

配置完连接之后 我们就可以开始发送消息和接收消息了 (因为我们上面刚刚测试rabbitmq的时候创建过队列和交换机等等这种东西了 当然 spring boot也可以创建)

spring boot创建交换机 队列 并绑定:

```
@Bean
public DirectExchange defaultExchange() {
```

```

        return new DirectExchange("directExchange");
    }

    @Bean
    public Queue queue() {
        //名字 是否持久化
        return new Queue("testQueue", true);
    }

    @Bean
    public Binding binding() {
        //绑定一个队列 to: 绑定到哪个交换机上面 with: 绑定的路由建 (routingKey)
        return BindingBuilder.bind(queue()).to(defaultExchange()).with("direct.key");
    }
}

```

发送消息:

发送消息比较简单，spring 提供了一个RabbitTemplate 来帮助我们完成发送消息的操作

如果想对于RabbitTemplate 这个类进行一些配置（至于有哪些配置我们后面会讲到） 我们可以在config类中 把他作为Bean new出来并配置

```

@Bean
public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
    //注意 这个ConnectionFactory 是使用javaconfig方式配置连接的时候才需要传入的 如果是yml配置
    //的连接的话是不需要的
    RabbitTemplate template = new RabbitTemplate(connectionFactory);
    return template;
}

```

```

@Component
public class TestSend {

    @Autowired
    RabbitTemplate rabbitTemplate;

    public void testSend() {
        //至于为什么调用这个API 后面会解释
        //参数介绍: 交换机名字, 路由建, 消息内容
        rabbitTemplate.convertAndSend("directExchange", "direct.key", "hello");
    }
}

```

我们只需要写一个类 然后交给spring 管理 在类里面注入RabbitTemplate 就可以直接调用api来发送消息了

接受消息：

这里我新建了一个项目（最好新建一个 当然 不新建也没关系） 来接收信息（之前的配置这里就不贴出来了 和上面基本一样），

```
@Component
public class TestListener {

    @RabbitListener(queues = "testQueue")
    public void get(String message) throws Exception{
        System.out.println(message);
    }

}
```

不出意外的话运行起来能看见效果， 这里就不贴效果图了

那么至此rabbitmq的一个快速开始 以及和spring boot整合 就完毕了， 下面开始会讲一些rabbitmq的一些高级特性 以及原理等

## RabbitMq特性

### 如何确保消息一定发送到Rabbitmq了？

我们刚刚所讲过的例子 在正常情况下 是没问题的， 但是 实际开发中 我们往往要考虑一些非正常的情况， 我们从消息的发送开始：

默认情况下，我们不知道我们的消息到底有没有发送到rabbitmq当中， 这肯定是不可取的， 假设我们是一个电商项目的话 用户下了订单 订单发送消息给库存 结果这个消息没发送到rabbitmq当中 但是订单还是下了，这时候 因为没有消息 库存不会去减少库存， 这种问题是非常严重的， 所以 接下来就讲一种解决方案： 失败回调

失败回调， 顾名思义 就是消息发送失败的时候会调用我们事先准备好的回调函数， 并且把失败的消息 和失败原因等 返回过来。

具体操作：

**注意 使用失败回调也需要开启发送方确认模式 开启方式在下文**

更改RabbitmqTemplate:



```

@Bean
public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
    RabbitTemplate template = new RabbitTemplate(connectionFactory);
    //开启mandatory模式 (开启失败回调)
    template.setMandatory(true);
    //指定失败回调接口的实现类
    template.setReturnCallback(new MyReturnCallback());
    return template;
}

```

回调接口的实现类：

实现RabbitTemplate.ReturnCallback里面的returnedMessage方法即可 他会把相关的参数都传给你

```

public class MyReturnCallback implements RabbitTemplate.ReturnCallback {

    @Override
    public void returnedMessage(Message message, int replyCode, String replyText,
    String exchange, String routingKey) {
        System.out.println(message);
        System.out.println(replyCode);
        System.out.println(replyText);
        System.out.println(exchange);
        System.out.println(routingKey);
    }
}

```

这里模拟一个失败的发送：当指定的交换机不能把消息路由到队列时（没有指定路由键或者指定的路由键没有绑定对应的队列 或者压根就没有绑定队列都会失败）消息就会发送失败 效果：

```

312
NO_ROUTE
directExchange
direct.key123123

```

分别打印的是错误状态码，错误原因（这里的原因是不能路由）交换机名字 和路由键（还有个参数是你发送出去的消息 因为太长了就没截图了。）

可能有些同学想到了一个答案-----事物

没错事物的确能解决这个问题，而且 恰巧rabbitmq刚好也支持事物，**但是！事物非常影响rabbitmq的性能** 有多严重？据我所查到的资料（当然 只是我所了解的 同学们也可以自己去尝试测试结果）开启rabbitmq事物的话**对性能的影响超过100倍之多** 也就是说 开启事物后处理一条消息的时间 不开事物能处理100条（姑且这样认为吧），那么 这样是非常不合理的，因为消息中间件的性能其实非常关键的（参考双11）如果这样子做的话 虽然能确保消息100%投递成功 但是代价太大了！

那么除了事物还有什么解决方案吗？

rabbitmq其实还提供了一种解决方案，叫：**发送方确认模式** 这种方式 对性能的影响非常小 而且也能确定消息是否发送成功

而且 发送方确认模式一般也会和失败回调一起使用 这样 就能确保消息100%投递了

发送方确认开启:

其实代码在上面配置连接的时候已经放出来了 就是在连接工厂那被注释的一行代码：

```
connectionFactory.setPublisherConfirms(true);
```

如果是yml配置的话:

```
spring:
  rabbitmq:
    publisher-confirms: true
```

和失败回调一样 实现一个接口:

```
public class MyConfirmCallback implements RabbitTemplate.ConfirmCallback{

    @Override
    public void confirm(CorrelationData correlationData, boolean ack, String cause) {
        System.out.println(correlationData);
        System.out.println(ack);
        System.out.println(cause);
    }
}
```

在RabbitmqTemplate 设置一下

```
template.setConfirmCallback(new MyConfirmCallback());
```

而且我们可以在发送消息的时候附带一个CorrelationData参数 这个对象可以设置一个id, 可以是你的业务id 方便进行对应的操作

```
CorrelationData correlationData = new CorrelationData(UUID.randomUUID().toString());
rabbitTemplate.convertAndSend("directExchange", "direct.key123123",
    "hello", correlationData);
```

效果:

```
CorrelationData [id=8ff37382-5202-4245-a5e8-ed6eb08a3328]
true
null
```

这里会把我们传入的那个业务id 以及ack（是否发送成功） 以及原因 返回回来

但是 要注意的是 confirm模式的发送成功 的意思是发送到RabbitMq（Broker）成功 而不是发送到队列成功

所以才有了上面我所说的那句 要和失败回调结合使用 这样才能确认消息投递成功了

可能这里有点绕，简单的总结一下就是 confirm机制是确认我们的消息是否投递到了 RabbitMq（Broker）上面 而mandatory是在我们的消息进入队列失败时候不会被遗弃（让我们自己进行处理）

那么上面 就是rabbitmq在发送消息时我们可以做的一些处理，接下来我们会讲到rabbitmq在接收（消费）消息时的一些特性

## 消费者如何确认消费？

为什么要确认消费？默认情况下 消费者在拿到rabbitmq的消息时 已经自动确认这条消息已经消费了，讲白话就是rabbitmq的队列里就会删除这条消息了，但是我们实际开发中 难免会遇到这种情况，比如说 拿到这条消息 发现我处理不了 比如说 参数不对，又比如说 我当前这个系统出问题了，暂时不能处理这个消息，但是 这个消息已经被你消费掉了 rabbitmq的队列里也删除掉了，你自己这边又处理不了，那么，这个消息就被遗弃了。这种情况在实际开发中是不合理的，rabbitmq提供了解决这个问题的方案，也就是我们上面所说的confirm模式 只是我们刚刚讲的是发送方的 这次我们来讲消费方的。

首先 我们在消费者这边（再强调一遍 我这里建议大家消费者和生产者分两个项目来做，包括我自己就是这样的，虽然一个项目也可以，我觉得分开的话容易理解一点）

设置一下消息确认为手动确认：

当然 我们要对我们的消费者监听器进行一定的配置的话，我们需要先实例一个监听器的Container 也就是容器，那么我们的监听器（一个消费者里面可以实例多个监听器）可以指定这个容器 那么我们只需要对这个 Container（容器）进行配置就可以了

首先得声明一个容器并且在容器里面指定消息确认为手动确认：

```
@Bean
public SimpleRabbitListenerContainerFactory
simpleRabbitListenerContainerFactory(ConnectionFactory connectionFactory){
    SimpleRabbitListenerContainerFactory simpleRabbitListenerContainerFactory =
        new SimpleRabbitListenerContainerFactory();
    //这个connectionFactory就是我们自己配置的连接工厂直接注入进来
    simpleRabbitListenerContainerFactory.setConnectionFactory(connectionFactory);
    //这边设置消息确认方式由自动确认变为手动确认
    simpleRabbitListenerContainerFactory.setAcknowledgeMode(AcknowledgeMode.MANUAL);
    return simpleRabbitListenerContainerFactory;
}
```

AcknowledgeMode关于这个类 就是一个简单的枚举类 我们来看看：

```

public enum AcknowledgeMode {
    NONE,
    MANUAL,
    AUTO;

    private AcknowledgeMode() {
    }

    public boolean isTransactionAllowed() { return this == AUTO || this == MANUAL; }

    public boolean isAutoAck() { return this == NONE; }

    public boolean isManual() { return this == MANUAL; }
}

```

3个状态 不确认 手动确认 自动确认

我们刚刚配置的就是中间那个 手动确认

既然现在是手动确认了 那么我们在处理完这条消息之后 得使这条消息确认：

```

@Component
public class TestListener {

    //containerFactory:指定我们刚刚配置的容器
    @RabbitListener(queues = "testQueue", containerFactory =
"simpleRabbitListenerContainerFactory")
    public void getMessage(Message message, Channel channel) throws Exception{
        System.out.println(new String(message.getBody(), "UTF-8"));
        System.out.println(message.getBody());
        //这里我们调用了下单方法 如果下单成功了 那么这条消息就可以确认被消费了
        boolean f = placeAnOrder();
        if (f){
            //传入这条消息的标识, 这个标识由rabbitmq来维护 我们只需要从message中拿出来就可以
            //第二个boolean参数指定是不是批量处理的 什么是批量处理我们待会儿会讲到
            channel.basicAck(message.getMessageProperties().getDeliveryTag(), false);
        }else {
            //当然 如果这个订单处理失败了 我们也需要告诉rabbitmq 告诉他这条消息处理失败了 可以退回
            //也可以遗弃 要注意的是 无论这条消息成功与否 一定要通知 就算失败了 如果不通知的话 rabbitmq端会显示这条
            //消息一直处于未确认状态, 那么这条消息就会一直堆积在rabbitmq端 除非与rabbitmq断开连接 那么他就会把这条
            //消息重新发给别人 所以 一定要记得通知!
            //前两个参数 和上面的意义一样, 最后一个参数 就是这条消息是返回到原队列 还是这条消息作废
            //就是不退回了。

            channel.basicNack(message.getMessageProperties().getDeliveryTag(), false, true);
            //其实 这个API也可以去告诉rabbitmq这条消息失败了 与basicNack不同之处 就是 他不能批量
            //处理消息结果 只能处理单条消息 其实basicNack作为basicReject的扩展开发出来的

            //channel.basicReject(message.getMessageProperties().getDeliveryTag(), true);
        }
    }
}

```

```
}
```

正常情况下的效果， 我就不演示给大家看了， 这里给大家看一个如果忘记退回消息的效果:

这里 我把消息确认的代码注释掉:

```
// channel.basicAck(message.getMessageProperties().getDeliveryTag(),false);
```

然后调用生产者发送一条消息 我们来看管理页面:

Overview				Messages			Message rates				+/-
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack		
testhost	testQueue	D	idle	1	0	1	0.00/s	0.00/s	0.00/s		

这里能看到 有一条消息在rabbitmq当中 而且状态是ready

然后我们使用消费者来消费掉他 注意 这里我们故意没有告诉rabbitmq我们消费成功了 来看看效果

这里 消费的结果打印就不截图了 还是来看管理页面:

Overview				Messages			Message rates				+/-
Virtual host	Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack		
testhost	testQueue	D	idle	0	1	1	0.00/s	0.00/s	0.00/s		

就算我们消费端消费了下次 但是能看到 这条消息还是会在rabbitmq当中 只是他的状态为 unacked 就是未确认

这就是我们刚刚说的那种情况 无论消费成功与否 一定要通知rabbitmq 不然就会这样 一直囤积在rabbitmq当中 直到连接断开为止.

## 消息预取

扯完消息确认 我们来讲一下刚刚所说的批量处理的问题

什么情况下会遇到批量处理的问题呢?

在这里 就要先扯一下rabbitmq的消息发放机制了

rabbitmq 默认 他会最快 以轮询的机制吧队列所有的消息发送给所有客户端 (如果消息没确认的话 他会添加一个 Unacked的标识上图已经看过了)

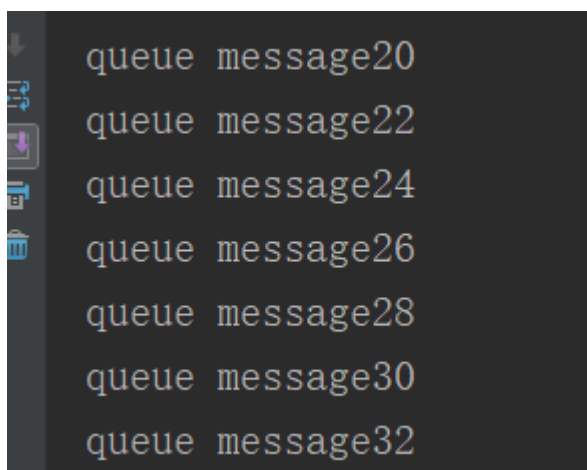
那么 这种机制会有什么问题呢, 对于Rabbitmq来讲 这样子能最快速的使自己不会囤积消息而对性能造成影响, 但是 对于我们整个系统来讲, 这种机制会带来很多问题, 比如说 我一个队列有2个人同时在消费, 而且他们处理能力不同, 我打个最简单的比方 有100个订单消息需要处理(消费) 现在有消费者A 和消费者B, 消费者A消费一条消息的速度是 10ms 消费者B 消费一条消息的速度是15ms (当然 这里只是打比方) 那么 rabbitmq 会默认给消费者A B 一人50条消息让他们消费 但是 消费者A 他500ms 就可以消费完所有的消息 并且处于空闲状态 而 消费

者B需要750ms 才能消费完 如果从性能上来考虑的话 这100条消息消费完的时间一共是750ms（因为2个人同时在消费） 但是如果 在消费者A消费完的时候 能把这个空闲的性能用来和B一起消费剩下的信息的话， 那么这处理速度就会快非常多。

这个例子可能有点抽象， 我们通过代码来演示一下

我往Rabbitmq生产100条消息 由2个消费者来消费 其中我们让一个消费者在消费的时候休眠0.5秒（模拟处理业务的延迟） 另外一个消费者正常消费 我们来看看效果：

正常的那个消费者会一瞬间吧所有消息（50条）全部消费完（因为我们计算机处理速度非常快） 下图是加了延迟的消费者：



可能我笔记里面你看不出效果， 这个你自己测试就会发现 其中一个消费者很快就处理完自己的消息了 另外一个消费者还在慢慢的处理 其实 这样严重影响了我们的性能了。

其实讲了这么多 那如何解决这个问题呢？

我刚刚解释过了 造成这个原因的根本就是rabbitmq消息的发放机制导致的， 那么我们现在来讲一下解决方案: **消息预取**

什么是消息预取？ 讲白了以前是rabbitmq一股脑吧所有消息都均发给所有的消费者（不管你受不受得了） 而现在是在我消费者消费之前 先告诉rabbitmq **我一次能消费多少数据 等我消费完了之后告诉rabbitmq** rabbitmq再给我发送数据

在代码中如何体现？

在使用消息预取前 要注意一定要设置为手动确认消息， 原因参考上面划重点的那句话。

因为我们刚刚设置过了 这里就不贴代码了， 完了之后设置一下我们预取消息的数量 一样 是在容器（Container）里面设置：

```

@Bean
public SimpleRabbitListenerContainerFactory
simpleRabbitListenerContainerFactory(ConnectionFactory connectionFactory){
    SimpleRabbitListenerContainerFactory simpleRabbitListenerContainerFactory =
        new SimpleRabbitListenerContainerFactory();
    simpleRabbitListenerContainerFactory.setConnectionFactory(connectionFactory);
    //手动确认消息
    simpleRabbitListenerContainerFactory.setAcknowledgeMode(AcknowledgeMode.MANUAL);
    //设置消息预取的数量
    simpleRabbitListenerContainerFactory.setPrefetchCount(1);
    return simpleRabbitListenerContainerFactory;
}

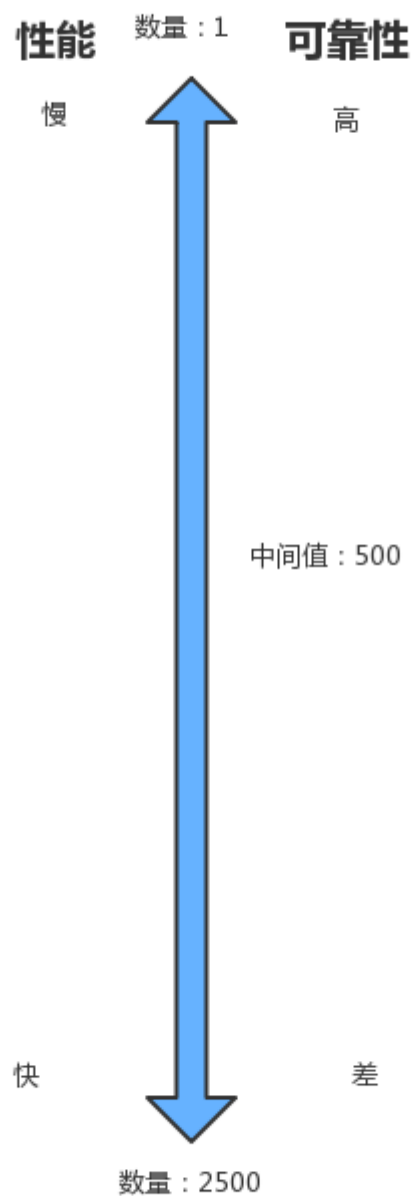
```

那么设置完之后是什么效果呢？还是刚刚那个例子 还是2个消费者 因为会在消费者返回消息的确认之后 rabbitmq 才会继续发送消息给客户端 而且客户端的消息累计量不会超过我们刚刚设置预取的数量，所以我们再运行同样的例子的话 会发现 A消费者消费完99条消息了 B消费者才消费1条 （因为B消费者休眠了0.5秒才消费完{返回消息确认} 但是0.5秒之内A消费者就已经把所有消息消费完毕了 当然 如果计算机处理速度较慢这个结果可能会有差异,效果大概就是A消费者会处理大量消息）

我这里的 effect 就是B消费者只消费一条消息 A消费者就消费完了，效果图就不发了 这里同学们尽量自己测试一下 或者改变一下参数看看效果。

关于这个预取的数量如何设置呢？我们发现 如果设置为1 能极大的利用客户端的性能（我消费完了就可以赶紧消费下一条 不会导致忙的很忙 闲的很闲）但是，我们每消费一条消息 就要通知一次rabbitmq 然后再取出新的消息，这样对于rabbitmq的性能来讲 是非常不合理的 所以这个参数要根据业务情况设置

我根据我查阅到的资料然后加以测试，这个数值的大小与性能成正比 但是有上限，与数据可靠性，以及我们刚刚所说的客户端的利用率成反比 大概如下图：



那么批量确认，就是对于我们预取的消息，进行统一的确认。

## 死信交换机

我们来看一段代码:

```
channel.basicNack(message.getMessageProperties().getDeliveryTag(), false, true);
```



我们上面解释过 这个代码是消息处理失败的确认 然后第三个参数我有解释过是消息是否返回到原队列， 那么问题来了， 如果 没有返回给原队列 那么这条消息就被作废了？

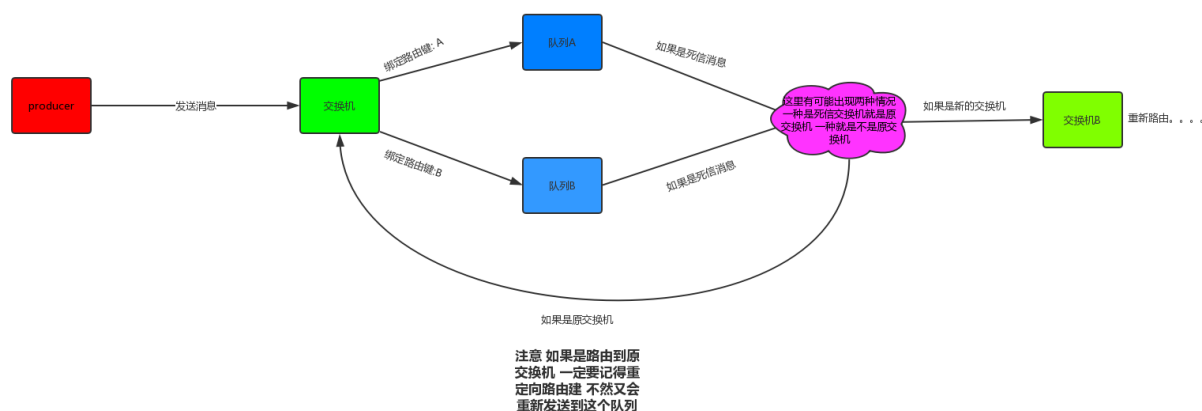
rabbitmq考虑到了这个问题提供了解决方案：**死信交换机**(有些人可能叫作垃圾回收器,垃圾交换机等)

死信交换机有什么用呢？ 在创建队列的时候 可以给这个队列附带一个交换机， 那么这个队列作废的消息就会被重新发到附带的交换机， 然后让这个交换机重新路由这条消息

理论是这样， 代码如下：

```
@Bean
public Queue queue() {
    Map<String, Object> map = new HashMap<>();
    //设置消息的过期时间 单位毫秒
    map.put("x-message-ttl", 10000);
    //设置附带的死信交换机
    map.put("x-dead-letter-exchange", "exchange.dlx");
    //指定重定向的路由建 消息作废之后可以决定需不需要更改他的路由建 如果需要 就在这里指定
    map.put("x-dead-letter-routing-key", "dead.order");
    return new Queue("testQueue", true, false, false, map);
}
```

大概是这样的一个效果：



其实我们刚刚发现 所谓死信交换机， 只是对应的队列设置了对应的交换机是死信交换机， 对于交换机来讲， 他还是一个普通的交换机。

下面会列出rabbitmq的常用配置：

队列配置：

参数名	配置作用
x-dead-letter-exchange	死信交换机
x-dead-letter-routing-key	死信消息重定向路由键
x-expires	队列在指定毫秒数后被删除
x-ha-policy	创建HA队列
x-ha-nodes	HA队列的分布节点
x-max-length	队列的最大消息数
x-message-ttl	毫秒为单位的消息过期时间，队列级别
x-max-priority	最大优先值为255的队列优先排序功能

消息配置：

参数名	配置作用
content-type	消息体的MIME类型，如application/json
content-encoding	消息的编码类型
message-id	消息的唯一性标识，由应用进行设置
correlation-id	一般用做关联消息的message-id，常用于消息的响应
timestamp	消息的创建时刻，整形，精确到秒
expiration	消息的过期时刻，字符串，但是呈现格式为整形，精确到秒
delivery-mode	消息的持久化类型，1为非持久化，2为持久化，性能影响巨大
app-id	应用程序的类型和版本号
user-id	标识已登录用户，极少使用
type	消息类型名称，完全由应用决定如何使用该字段
reply-to	构建回复消息的私有响应队列
headers	键/值对表，用户自定义任意的键和值
priority	指定队列中消息的优先级

# Rabbitmq linux安装&集群高可用

## rabbitmq linux下安装

这里考虑到可能有同学没了解过linux 或者不太熟悉linux 所以下载地址之类的东西我这里直接贴现成的，也就是说 只要按照我的步骤走下去基本上都没问题.

在安装(搭建集群)之前 确定两个点 **1:防火墙关掉 2:打开网络**

关闭防火墙 `systemctl stop firewalld.service` 禁止开机自启 `systemctl disable firewalld.service`

首先 还是安装erlang

下载erlang wget [http://www.rabbitmq.com/releases/erlang/erlang-18.2-1.el6.x86\\_64.rpm](http://www.rabbitmq.com/releases/erlang/erlang-18.2-1.el6.x86_64.rpm) 安装erlang `rpm -ihv http://www.rabbitmq.com/releases/erlang/erlang-18.2-1.el6.x86\_64.rpm`

安装完erlang之后 开始rabbitmq 还是再提醒一下 先装erlang 再装rabbitmq

装Rabbitmq之前 先装一个公钥：

`rpm --import https://dl.bintray.com/rabbitmq/Keys/rabbitmq-release-signing-key.asc`

装好公钥之后 下载Rabbitmq:

wget <http://www.rabbitmq.com/releases/rabbitmq-server/v3.6.6/rabbitmq-server-3.6.6-1.el7.noarch.rpm>

安装:

`rpm -ihv rabbitmq-server-3.6.6-1.el7.noarch.rpm`

安装中途可能会提示你需要一个叫socat的插件

如果提示了 就先安装socat 再装rabbitmq

安装socat:

`yum install socat`

至此 就装好了Rabbitmq了 可以执行以下命令启动Rabbitmq:

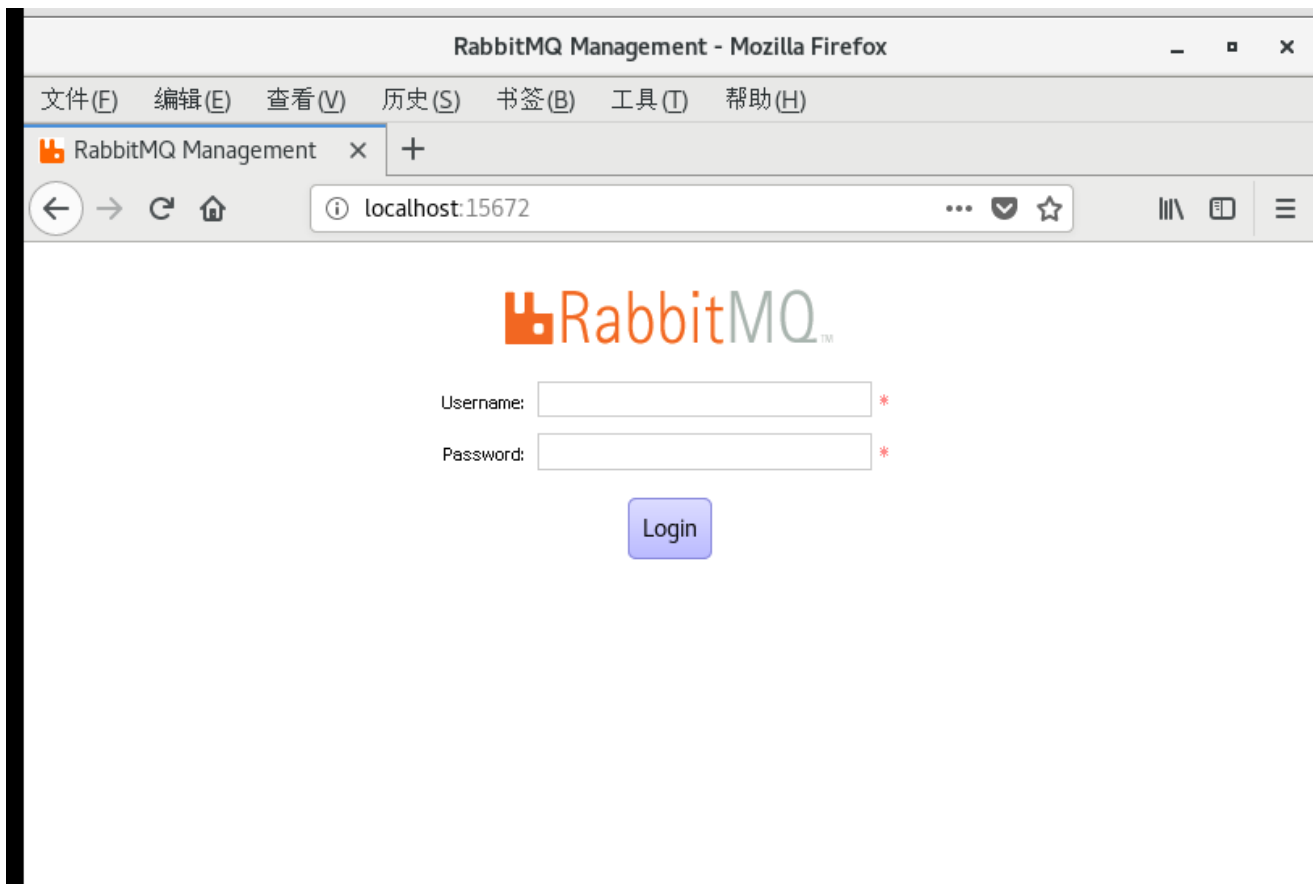
`service rabbitmq-server start`

和windows环境下一样 rabbitmq对于linux也提供了他的管理插件

安装rabbitmq管理插件:

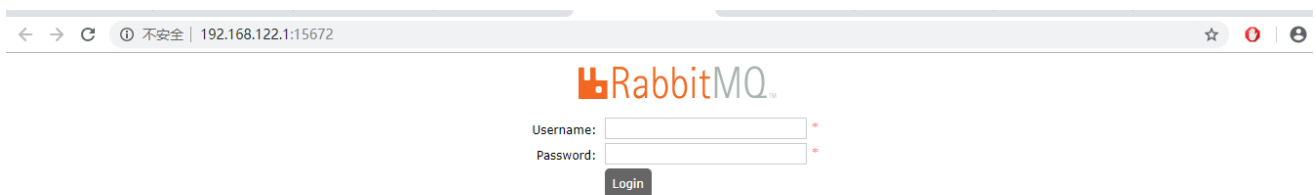
`rabbitmq-plugins enable rabbitmq_management`

安装完管理插件之后 如果有装了浏览器的话 比如火狐 可以和windows一样 访问 一下 localhost:15672 可以看到一个熟悉的页面:



当然 如果你和我一样 是用虚拟机搭建的linux的话 可以用主机访问一下也是没问题的 比如说我这里虚拟机的地址为:

```
ix errors 0 dropped 0 overruns 0 carrier 0 collisions 0
virbr0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 192.168.122.1 netmask 255.255.255.0 broadcast 192.168.122.255
    ether 52:54:00:3f:8b:99 txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```



那么 安装 我们就讲到这 后面我们来讲集群环境的搭建 以及一些问题

## rabbitmq集群搭建，配置

rabbitmq由于是由erlang语言开发的 天生就支持分布式

rabbitmq 的集群分两种模式 一种是默认模式 一种是镜像模式

当然 所谓的镜像模式是基于默认模式加上一定的配置来的

在rabbitmq集群当中 所有的节点（一个rabbitmq服务器） 会被归为两类 一类是磁盘节点 一类是内存节点

磁盘节点会把集群的所有信息(比如交换机,队列等信息)持久化到磁盘当中，而内存节点只会将这些信息保存到内存当中 讲白了 重启一遍就没了。

为了可用性考虑 rabbitmq官方强调集群环境至少需要有一个磁盘节点，而且为了高可用的话，必须至少要有2个磁盘节点，因为如果只有一个磁盘节点 而刚好这唯一的磁盘节点宕机了的话，集群虽然还是可以运作，但是不能对集群进行任何的修改操作（比如 队列添加，交换机添加，增加/移除 新的节点等）

具体想让rabbitmq实现集群，我们首先需要改一下系统的hostname (因为rabbitmq集群节点名称是读取hostname的)

这里 我们模拟3个节点：

rabbitmq1

rabbitmq2

rabbitmq3

linux修改hostname命令: hostnamectl set-hostname [name]

### 修改后重启一下 让rabbitmq重新读取节点名字

然后 我们需要让每个节点通过hostname能ping通（记得关闭防火墙）这里 我们可以修改修改一下hosts文件

关闭防火墙:

关闭防火墙 systemctl stop firewalld.service 禁止开机自启 systemctl disable firewalld.service

接下来,我们需要将各个节点的.erlang.cookie文件内容保持一致(文件路径/var/lib/rabbitmq/.erlang.cookie)

因为我是采用虚拟机的方式来模拟集群环境，所以如果像我一样是克隆的虚拟机的话 同步.erlang.cookie文件这个操作在克隆的时候就已经完成了。

上面这些步骤完成之后 我们就可以开始来构建集群 了

我们先让rabbitmq2 加入 rabbitmq1与他构建为一个集群


执行命令( ram:使rabbitmq2成为一个内存节点 默认为:disk 磁盘节点):

rabbitmqctl stop\_app rabbitmqctl join\_cluster rabbit@rabbitmq1 --ram rabbitmqctl start\_app

在构建的时候 我们需要先停掉rabbitmqctl服务才能构建 等构建完毕之后再启动

我们吧rabbitmq2添加完之后在rabbitmq3节点上也执行同样的代码 使他也加入进去 当然 我们也可以让rabbitmq3也作为一个磁盘节点

当执行完操作以后我们来看看效果:



Name	File descriptors (?)	Socket descriptors (?)	Erlang processes	Memory	Disk sp
rabbit@rabbitmq1	57 1024 available	0 829 available	197 1048576 available	49MB 389MB high watermark	13GB 48MB low wa
rabbit@rabbitmq2	56 1024 available	0 829 available	227 1048576 available	49MB 389MB high watermark	13GB 48MB low wa
rabbit@rabiitmq3	57 1024 available	0 829 available	231 1048576 available	50MB 389MB high watermark	13GB 48MB low wa

Ports and contexts

随便在哪个节点打开管理页面都能看到集群环境各节点的信息;

有关集群的其他命令:

rabbitmq-server -detached 启动RabbitMQ节点 rabbitmqctl start\_app 启动RabbitMQ应用, 而不是节点  
rabbitmqctl stop\_app 停止 rabbitmqctl status 查看状态 rabbitmqctl add\_user mq 123456 rabbitmqctl  
set\_user\_tags mq administrator 新增账户 rabbitmq-plugins enable rabbitmq\_management 启用  
RabbitMQ\_Management rabbitmqctl cluster\_status 集群状态 rabbitmqctl forget\_cluster\_node  
rabbit@[nodeName] 节点摘除 rabbitmqctl reset application 重置

普通模式的rabbitmq集群搭建好后, 我们来说一下镜像模式

在普通模式下的rabbitmq集群 他会吧所有节点的交换机信息 和队列的元数据(队列数据分为两种 一种为队列里面的消息, 另外一种为队列本身的信息 比如队列的最大容量, 队列的名称, 等等配置信息, 后者称之为元数据) 进行复制 确保所有节点都有一份。

而镜像模式, 则是吧所有的队列数据完全同步 (当然 对性能肯定会有一定影响) 当对数据可靠性要求高时 可以使用镜像模式

实现镜像模式也非常简单 有2种方式 一种是直接在管理台控制, 另外一种是在声明队列的时候控制

声明队列的时候可以加入镜像队列参数 在上方的参数列表当中有解释 我们来讲一下管理台控制

镜像队列配置命令解释:

**rabbitmqctl set\_policy [-p Vhost] Name Pattern Definition [Priority]**

**-p Vhost:** 可选参数, 针对指定vhost下的queue进行设置 **Name:** policy的名称 **Pattern:** queue的匹配模式(正则表达式) **Definition:** 镜像定义, 包括三个部分ha-mode, ha-params, ha-sync-mode **ha-mode:**指明镜像队列的模式, 有效值为 all/exactly/nodes **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

这里举个例子 如果想配置所有名字开头为 policy的队列进行镜像 镜像数量为1那么命令如下:

```
rabbitmqctl set_policy ha_policy "^policy_" '{"ha-mode":"exactly","ha-params":1,"ha-sync-mode":"automatic"}
```