# RabbitMQ

**By Brother.Lee 20171211 Shanghai**

# （一）RabbitMQ消息队列-RabbitMQ的优劣势及产生背景

本篇并没有直接讲到技术，例如没有先写个Helloword。我想在选择了解或者学习一门技术之前先要明白为什么要现在这个技术而不是其他的，以免到最后发现自己学错了。同时如果已经确定就是他，最好先要了解下技术产生的背景等因素，以便对技术有更深刻全面的了解（那句话怎么讲的“你不了解过去的我，又怎么理解现在的我”）。

## 为什么使用RabbitMQ

**为什么我开始选择学习RabbitMQ：**

1. 安装部署简单，上手门槛低，功能丰富，符合AMQP标准；
2. 企业级消息队列，经过大量实践考验的高可靠；
3. 集群易扩展，可以轻松的增减集群节点；
4. 有强大的WEB管理页面。

**企业为什么将RabbitMQ作为消息队列系统：**

在我学习RabbitMQ的初期我在网上查到一些这方面的资料，我认为比较好的是下面这封<http://www.doc88.com/p-5826232080382.html> 一方面是他对于RabbitMQ的评价本身，更重要的是对于技术选型评价的两个维度：

**十万米高空看技术和显微镜看技术**。

* 1. 有商业化的运营，不会轻易死掉；
  2. 遵循AMQP协议，不会被绑架；
  3. 强大的社区支持，为技术进步提供动力；
  4. 大量成功的应用案例，例如阿里、网易等互联网巨头都有使用。

**显微镜看RabbitMQ：**

* 1. Erlang开发，AMQP的最佳搭档，在支持持久化的消息队列中性能算很优秀的；
  2. 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；
  3. 可靠性高；
  4. 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；
  5. WEB管理和监控，有些技术癌更喜欢命令行界面，但WEB管理为后期运维提供很大的便利。

**RabbitMQ劣势：**  
在kafka和zero面前性能被虐成渣，（持久化消息和ACK确认的情况下生产和消费消息单机大约在1-2万左右）

结论：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吧，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。

## RabbitMQ产生的背景

1、消息队列系统最在可以追溯到上个世纪（是不是感觉很久远，其实是1983年，那时候我还没用出生）。1983年最早的消息队列软件Teknekron诞生，当时紧用于一些金融交易等系统。

2、上世纪九十年代，诞生了多家消息队列系统，例如IBM MQ、微软的MSMQ、TIBCO MQ等消息队列在企业中的应用也愈加广泛。显然这些商用的消息队列系统如果企业要使用需要付出高昂的成本，并且各个消息队列之间使用不同的API不同的协议。

3、2004年，AMQP（Advanced Message Queuing Protocol,高级消息队列协议）开始开发。通过这一标准可以和任意AMQP供应商提供的MQ服务进行交互。

4、2006年，光阴荏苒时光如梭，一转眼就说到了重点。我们的主角使用Erlang语言实现的AMQP开源版本，RabbitMQ诞生了，同年AMQP协议首次发布。

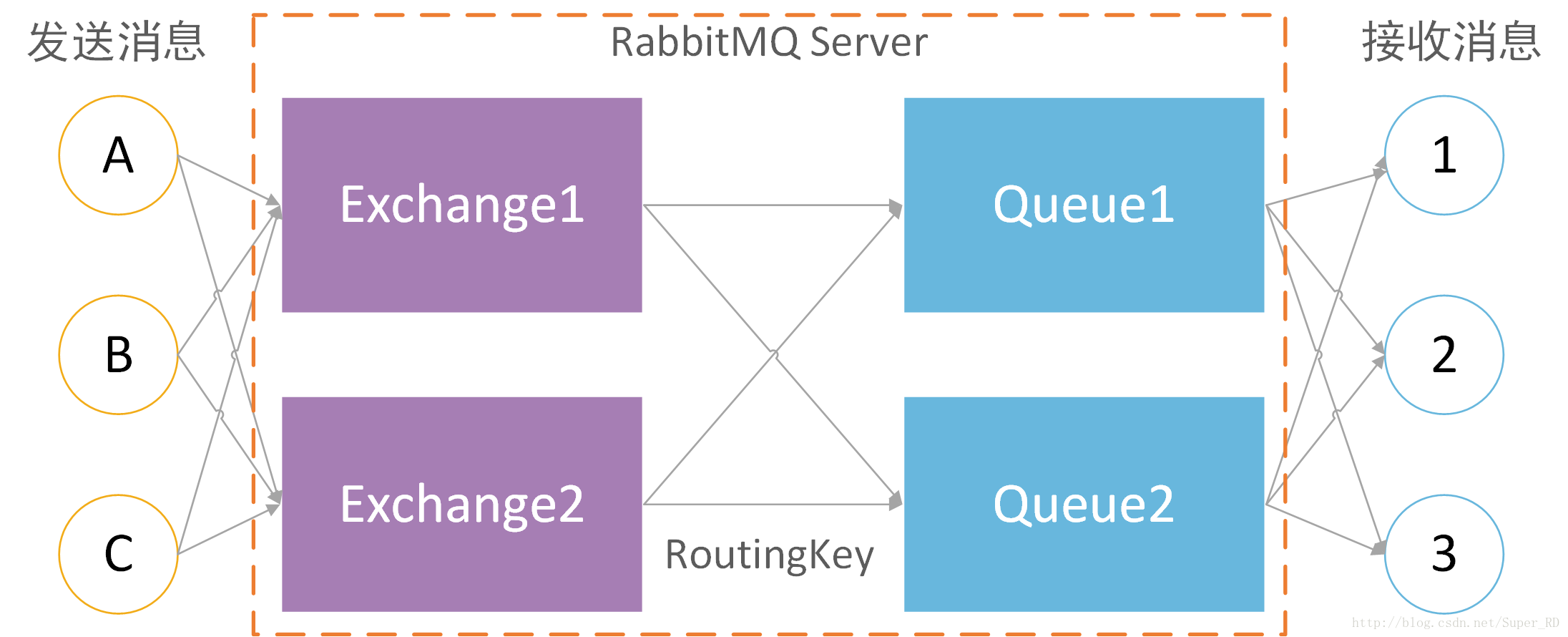
**为什么叫RabbitMQ？**   
很多人估计和我一样也有这个疑问，我在《RabbitMQ实战》这本书中找到了答案：兔子行动非常迅速而且繁殖起来也非常疯狂，所以就把Rabbit用作这个分布式软件的命名（就是真么简单）。

# （二）RabbitMQ消息队列-RabbitMQ消息队列架构与基本概念

没错我还是没有讲怎么安装和写一个HelloWord，不过快了，这一章我们先了解下RabbitMQ的基本概念。

## RabbitMQ架构

说是架构其实更像是应用场景下的架构（自己画的有点丑，勿嫌弃）



## RabbitMQ消息队列基本概念

**RabbitMQ Server：**也叫broker server，它是一种传输服务。 他的角色就是维护一条从Producer到Consumer的路线，保证数据能够按照指定的方式进行传输。但是这个保证也不是100%的保证，但是对于普通的应用来说这已经足够了。当然对于商业系统来说，可以再做一层数据一致性的guard，就可以彻底保证系统的一致性了。

**Producer：** 消息生产者，如图A、B、C，数据的发送方。消息生产者连接RabbitMQ服务器然后将消息投递到Exchange。

**Consumer：**消息消费者，如图1、2、3，数据的接收方。消息消费者订阅队列，RabbitMQ将Queue中的消息发送到消息消费者。

**Exchange：**生产者将消息发送到Exchange（交换器），由Exchange将消息路由到一个或多个Queue中（或者丢弃）。Exchange并不存储消息。RabbitMQ中的Exchange有fanout、direct、topic、headers四种类型，每种类型对应不同的路由规则，后面详细介绍这四种类型。

**Queue：**（队列）是RabbitMQ的内部对象，用于存储消息。消息消费者就是通过订阅队列来获取消息的，RabbitMQ中的消息都只能存储在Queue中，生产者生产消息并最终投递到Queue中，消费者可以从Queue中获取消息并消费。多个消费者可以订阅同一个Queue，这时Queue中的消息会被平均分摊给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理。

**RoutingKey：**生产者在将消息发送给Exchange的时候，一般会指定一个routing key，来指定这个消息的路由规则，而这个routing key需要与Exchange Type及binding key联合使用才能最终生效。在Exchange Type与binding key固定的情况下（在正常使用时一般这些内容都是固定配置好的），我们的生产者就可以在发送消息给Exchange时，通过指定routing key来决定消息流向哪里。RabbitMQ为routing key设定的长度限制为255 bytes。

**Connection：** （连接）。Producer和Consumer都是通过TCP连接到RabbitMQ Server的。以后我们可以看到，程序的起始处就是建立这个TCP连接。

**Channels：** （信道）。它建立在上述的TCP连接中。数据流动都是在Channel中进行的。也就是说，一般情况是程序起始建立TCP连接，第二步就是建立这个Channel。

**Exchange Types：**

* fanout

fanout类型的Exchange路由规则非常简单，它会把所有发送到该Exchange的消息路由到所有与它绑定的Queue中。 生产者发送到Exchange的所有消息都会路由到绑定的Queue，并最终被两个消费者消费。

* direct

direct类型的Exchange路由规则也很简单，它会把消息路由到那些binding key与routing key完全匹配的Queue中。（在实际使用RabbitMQ的过程中并没有binding key这个参数，只有routing key，为了区分我们把交换机和队列绑定时传的参数叫binding key，把发送消息时带的这个参数叫routing key）

* topic

前面讲到direct类型的Exchange路由规则是完全匹配binding key与routing key，但这种严格的匹配方式在很多情况下不能满足实际业务需求。topic类型的Exchange在匹配规则上进行了扩展，它与direct类型的Exchage相似，也是将消息路由到binding key与routing key相匹配的Queue中，但direct是完全匹配，而通过topic可以进行模糊匹配

routing key为一个句点号“. ”分隔的字符串（我们将被句点号“. ”分隔开的每一段独立的字符串称为一个单词），如“stock.usd.nyse”、“nyse.vmw”、“quick.orange.rabbit”   
binding key与routing key一样也是句点号“. ”分隔的字符串   
binding key中可以存在两种特殊字符“”与“#”，用于做模糊匹配，其中“”用于匹配一个单词，“#”用于匹配多个单词（可以是零个）

* headers

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

# （三）RabbitMQ消息队列-Centos7下安装RabbitMQ3.6.1

## 第一步:安装Erlang

  因为rabbitMQ是Erlang语言编写的，所以我们首先需要安装Erlang

**rpm** -Uvh http://www.rabbitmq.com/releases/erlang/erlang-18.1-1.el7.centos.x86\_64.rpm

检测：

**erl** -version

若出现对应的Erlang的版本号，则erlang安装成功，否则失败。

完成后进入/opt/erlang查看执行结果

**[root@iZ25e3bt9a6Z** rabbitmq]# cd /opt/erlang/  
**[root@iZ25e3bt9a6Z** erlang]# erl  
**Erlang/OTP** 18 [erts-7.3] [source] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-poll:false]  
**Eshell** V7.3 (abort with ^G)  
**1>**

当出现以上信息时表示安装完成。然后输入

**halt().**

退出即可。

## 第二步:安装rabbitMQ-server

本人安装成功的方式：

**rpm** -Uvh http://www.rabbitmq.com/releases/rabbitmq-server/v3.5.6/rabbitmq-server-3.5.6-1.noarch.rpm

## 第三步:查看rabbitmq-server是否已经安装好了，能查到说明已经安装完成了。

**rpm** -qa|grep rabbitmq



## 第四步:开启rabbit-server

**service** rabbitmq-server start



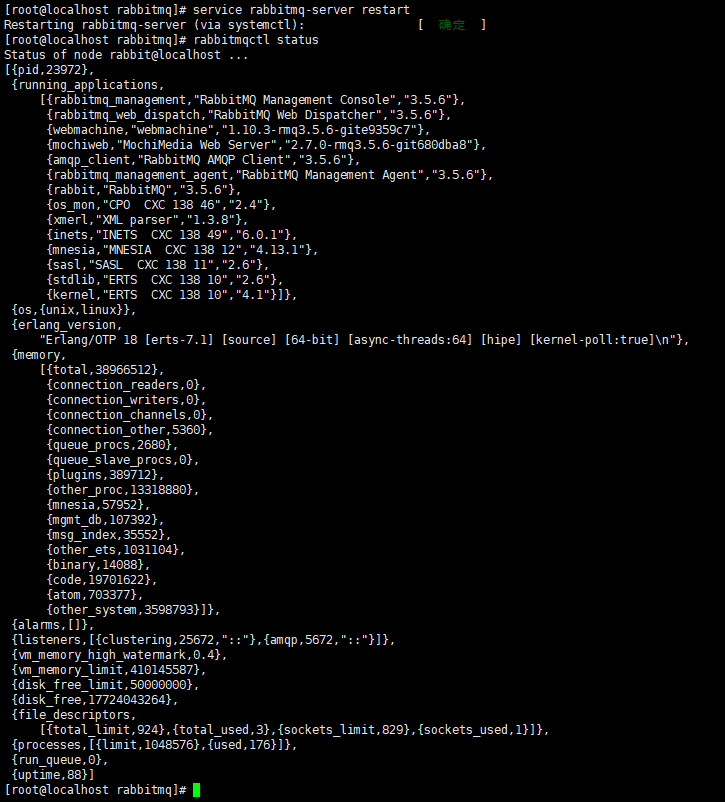
## 第五步:关闭rabbit-server（验证命令）

**service** rabbitmq-server stop



## 第六步:查看状态

**rabbitmqctl** status



这样虽然我们已经将rabbitmq的服务正常启动了，但是我们在物理机的浏览器中输入ip:15672时，并不能连接，因为我们还没有配置维护插件和开启远程连接

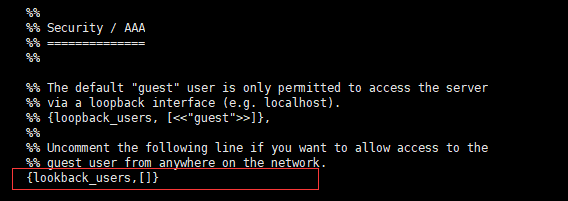
## 第七步、安装命令维护插件

**rabbitmq-plugins** enable rabbitmq\_management

## 第八步、设置配置文件，并开启用户远程访问

#cd /etc/rabbitmq   
#cp /usr/share/doc/rabbitmq-server-3.5.6/rabbitmq.config.example /etc/rabbitmq/   
  
#mv rabbitmq.config.example rabbitmq.config

**vi** /etc/rabbitmq/rabbitmq.config



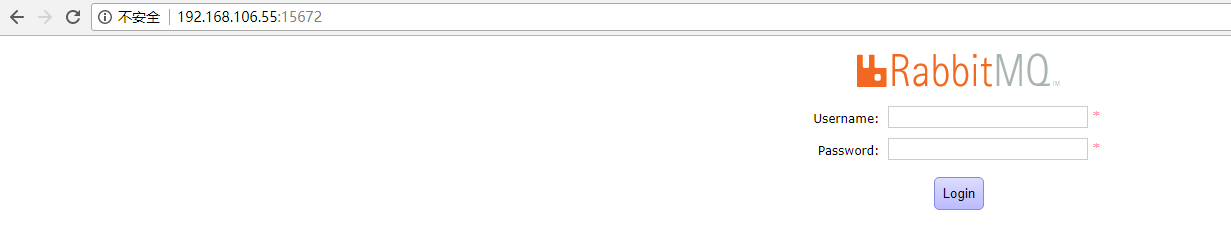
**注意去掉逗号**

## 第八步:重启rabbit-server服务

**service** rabbitmq-server restart

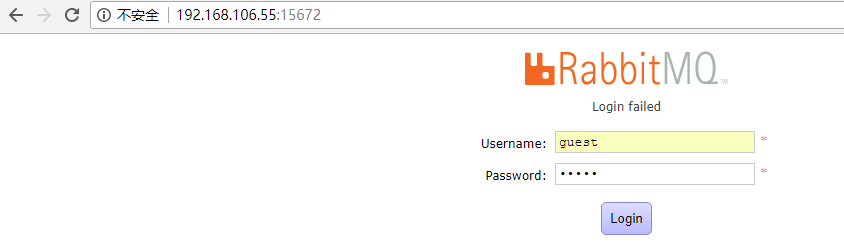
## 第九步、登录访问

  在浏览器中输入ip:15672即可出现登录页面，用户名和密码都是guest



## 错误:

1:rabbitMQ guest账号登录总是提示失败



解决方法：

vi /etc/rabbitmq/rabbitmq.config

设置文件内容如下：  
[{rabbit, [{loopback\_users, []}]}].

重启rabbitMQ：

   service rabbitmq-server stop

   service rabbitmq-server start

   #启用监控台web，通过http://ip:port访问，如：http://192.168.16.43:15672/#/queues

   rabbitmq-plugins enable rabbitmq\_management

**重试了之后发现还是不行!**

2:远程访问配置

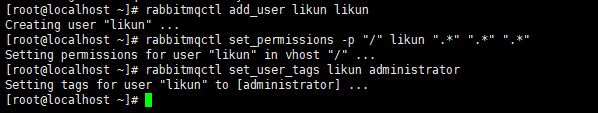
　　默认网页是不允许访问的，需要增加一个用户修改一下权限，代码如下：

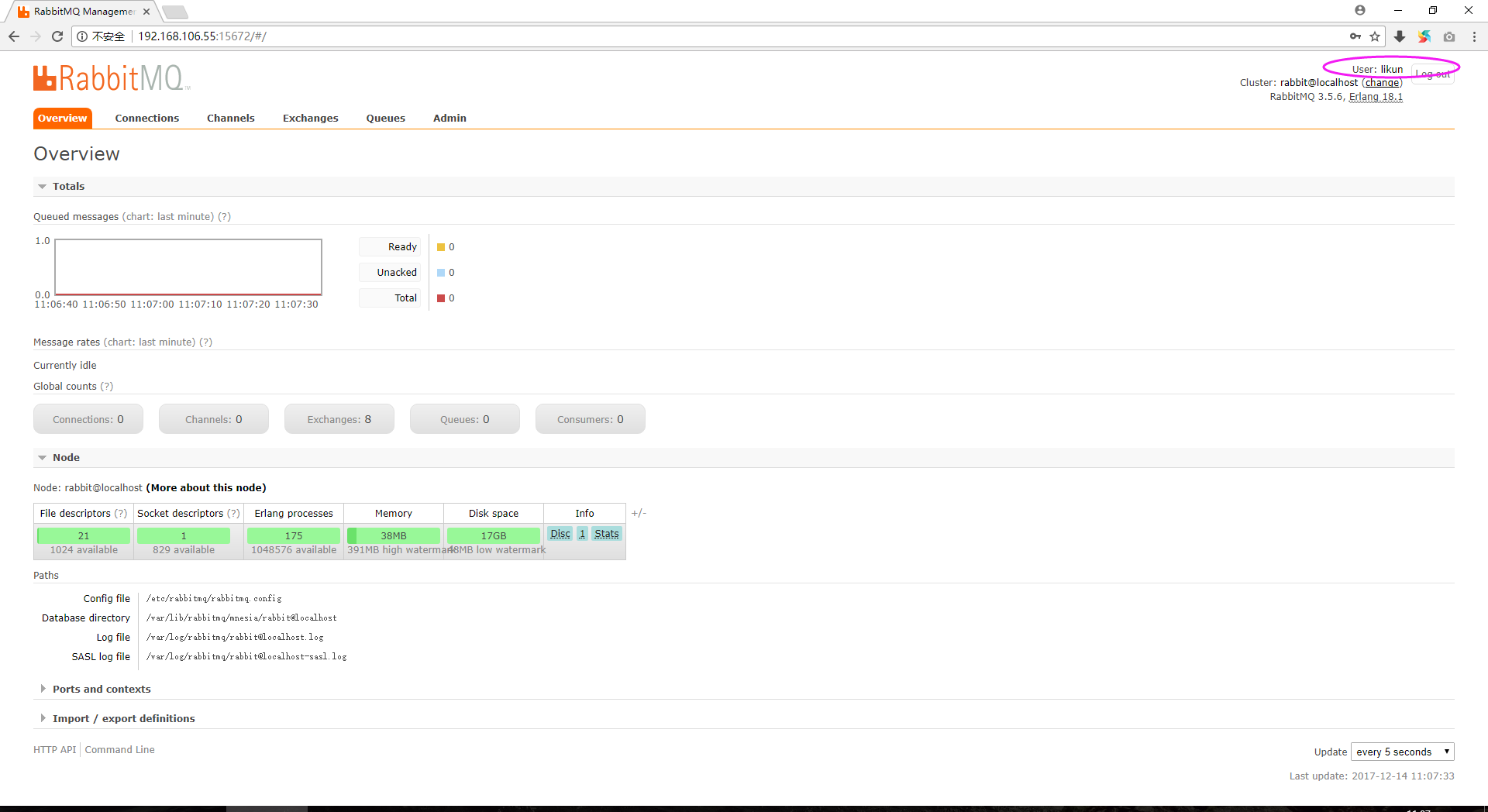
　　添加用户:rabbitmqctl add\_user hxb hxb

　　添加权限:rabbitmqctl set\_permissions -p "/" hxb ".\*" ".\*" ".\*"

修改用户角色rabbitmqctl set\_user\_tags hxb administrator

然后就可以远程访问了，然后可直接配置用户权限等信息。





# （四）RabbitMQ消息队列-服务详细配置与日常监控管理

## RabbitMQ服务管理

**启动服务：rabbitmq-server** -detached【 /usr/local/rabbitmq/sbin/rabbitmq-server -detached 】  
**查看状态：rabbitmqctl** status  
**关闭服务：rabbitmqctl** stop  
**列出角色：rabbitmqctl** list\_users  
**开启某个插件：rabbitmq-pluginsenable** xxx  
**关闭某个插件：rabbitmq-pluginsdisablexxx  
注意：重启服务器后生效。**

## RabbitMQ服务详细配置rabbitmq.config

RabbitMQ 提供了三种方式来定制服务器:

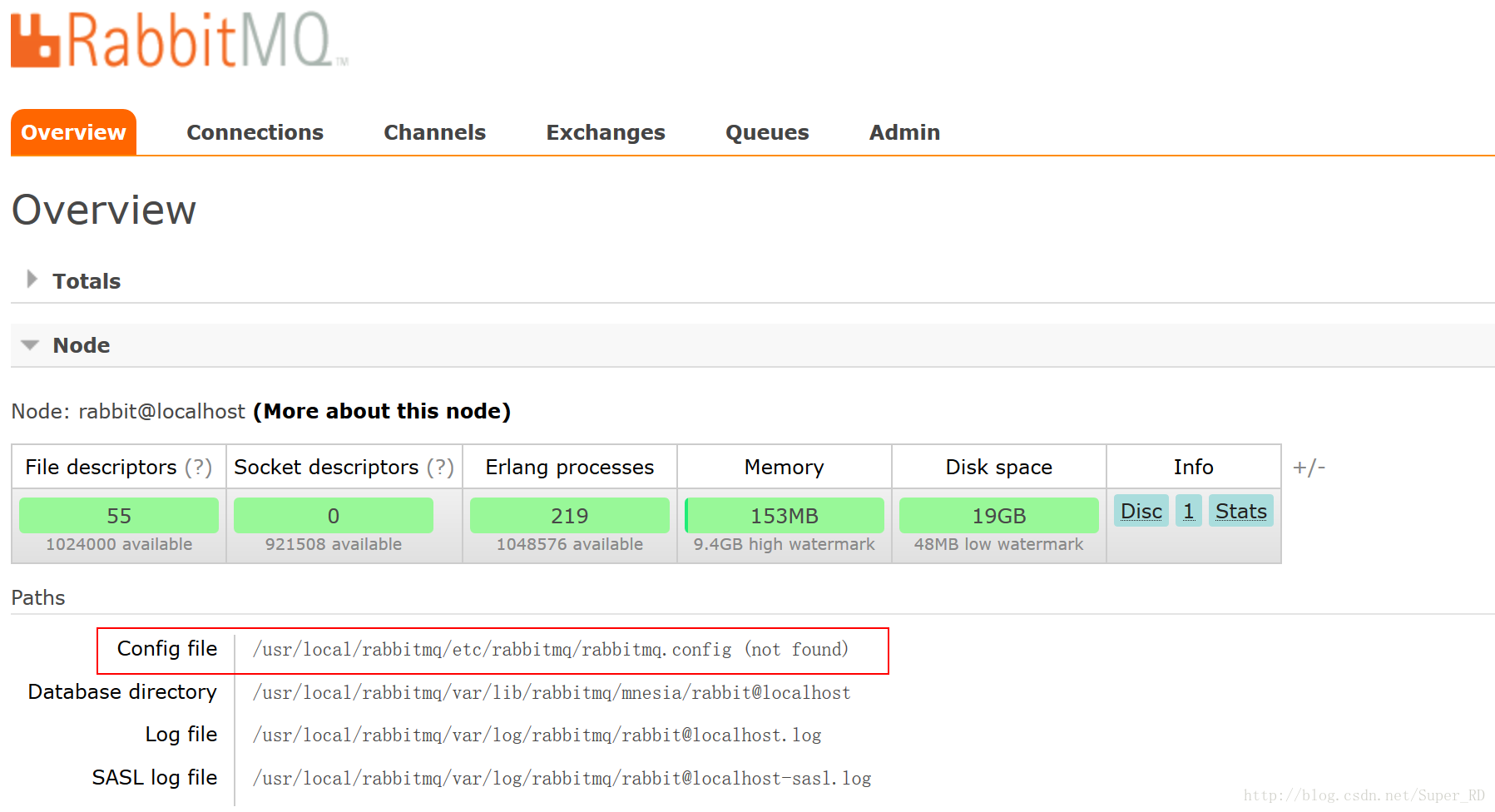
**环境变量**   
定义端口，文件位置和名称(接受shell输入,或者在环境配置文件（rabbitmq-env.conf）中设置)。   
**配置文件**   
为服务器组件设置权限,限制和集群，也可以定义插件设置（rabbitmq.config）。   
**运行时参数和策略**   
可在运行时进行修改集群设置。

下面我会介绍通过配置文件rabbitmq.config来配置rabbitmq。

* **rabbitmq.config 的位置**   
  这些文件的位置分布特定的. 默认情况下，这些文件是没有创建的,但每个平台上期望的位置如下：

**Generic** UNIX - $RABBITMQ\_HOME/etc/rabbitmq/  
**Debian** - /etc/rabbitmq/  
**RPM** - /etc/rabbitmq/  
**Mac** OS X (Homebrew) - ${install\_prefix}/etc/rabbitmq/, the Homebrew prefix is usually/usr/local  
**Windows** - %APPDATA%\RabbitMQ\

通过WEB管理插件我们也可以看到该配置文件的地址：



* **rabbitmq.config 配置**   
  我的是在/usr/local/rabbitmq/etc/rabbitmq/rabbitmq.config，rabbitmq.config配置文件允许配置RabbitMQ 核心程序， Erlang 服务和RabbitMQ 插件。它是标准的Erlang 配置文件。RabbitMQ在找不到配置文件的情况下会按照默认的配置运行。在系统提示的位置新建这个文件：

**touch** /usr/local/rabbitmq/etc/rabbitmq/rabbitmq.config

如下是我常用到的rabbitmq.config配置文件的信息，意思是RabbitMQ内存阈值最大可用使用我系统40%的内存，超过40%开始拒绝生产消息，当内存使用率达到阈值的40%时开始持久化到磁盘。0.4也是官方建议的值，我曾经为了性能把该值设置成0.8，内存稍一波动整个服务就崩溃了。

**[  
{rabbit,** [{vm\_memory\_high\_watermark\_paging\_ratio, 0.4},  
**{vm\_memory\_high\_watermark,** 0.4}]}  
**].**

更多可配置参数请查看：<http://blog.csdn.net/super_rd/article/details/70327712>

## 通过rabbitmqctl管理RabbitMQ

virtual\_host管理

**新建virtual\_host**: rabbitmqctl add\_vhost xxx  
**撤销virtual\_host**:rabbitmqctl delete\_vhost xxx

用户管理

**新建用户：rabbitmqctl** add\_user usernamexxx pwdxxx  
**删除用户：rabbitmqctl** delete\_user usernamexxx  
**改密码**: rabbimqctl change\_password {username} {newpassword}  
**设置用户角色：rabbitmqctl** set\_user\_tags {username} {tag ...}  
**Tag可以为** administrator,monitoring, management

权限管理

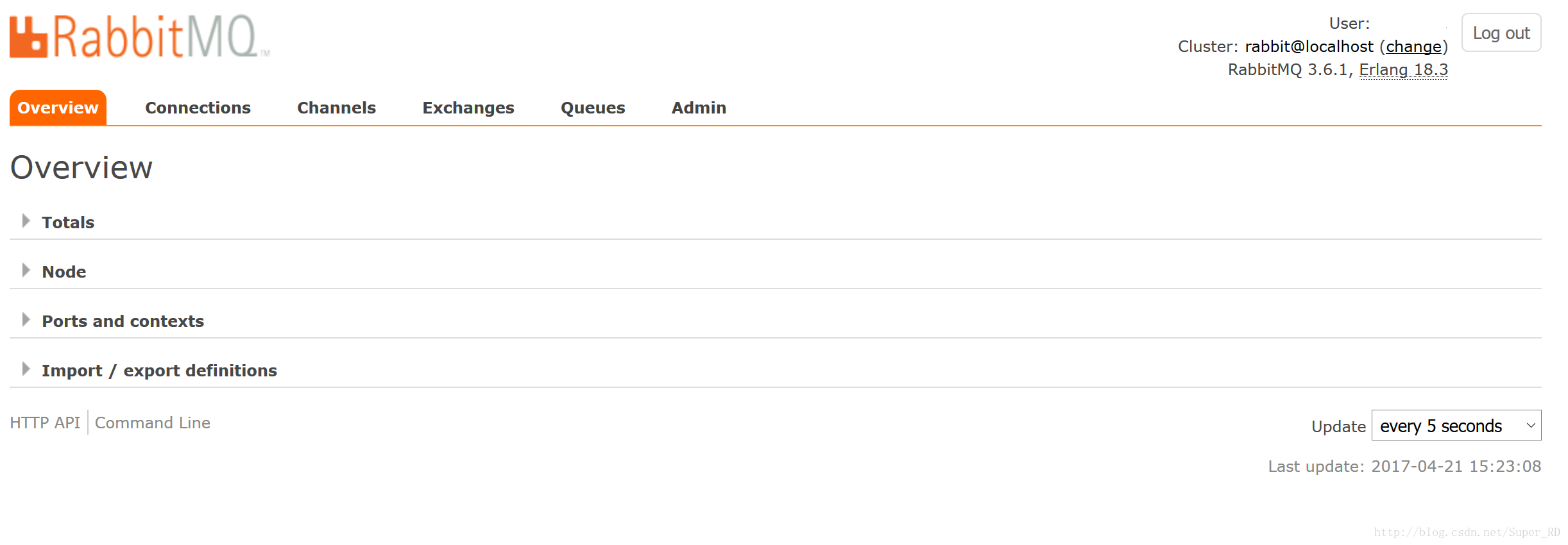
**rabbitmqctl** set\_permissions -p / username ".\*" ".\*" ".\*" //添加权限  
**rabbitmqctl** set\_user\_tags username administrator //修改用户角色  
  
**权限设置说明：rabbitmqctl** set\_permissions [-pvhostpath] {user} {conf} {write} {read}  
**Vhostpath：Vhost路径。  
user：用户名。  
Conf：一个正则表达式match哪些配置资源能够被该用户访问。  
Write：一个正则表达式match哪些配置资源能够被该用户读。  
Read：一个正则表达式match哪些配置资源能够被该用户访问。**

获取服务器状态信息

**获取服务器状态：rabbitmqctl** status  
  
**关闭应用：rabbitmqctl** stop\_app  
**启动应用，和上述关闭命令配合使用，达到清空队列的目的：rabbitmqctl** start\_app  
**清除所有队列：rabbitmqctl** reset  
  
**查看所有队列信息：rabbitmqctl** list\_queues  
  
**获取队列信息：rabbitmqctl** list\_queues[-p vhostpath] [queueinfoitem ...]  
**Queueinfoitem可以为：name，durable，auto\_delete，arguments，messages\_ready，messages\_unacknowledged，messages，consumers，memory。  
  
获取Exchange信息：rabbitmqctllist\_exchanges[-p** vhostpath] [exchangeinfoitem ...]  
**Exchangeinfoitem有：name，type，durable，auto\_delete，internal，arguments。  
  
获取Binding信息：rabbitmqctllist\_bindings[-p** vhostpath] [bindinginfoitem ...]   
**Bindinginfoitem有：source\_name，source\_kind，destination\_name，destination\_kind，routing\_key，arguments。  
  
获取Connection信息：rabbitmqctllist\_connections** [connectioninfoitem ...]  
**Connectioninfoitem有：recv\_oct，recv\_cnt，send\_oct，send\_cnt，send\_pend等。  
  
获取Channel信息：rabbitmqctl** list\_channels[channelinfoitem ...]  
**Channelinfoitem有consumer\_count，messages\_unacknowledged，messages\_uncommitted，acks\_uncommitted，messages\_unconfirmed，prefetch\_count，client\_flow\_blocked。**

## RabbitMQ消息队列WEB管理工具

目前处于技术癌早期，所以一般能图形化还是要尽量图形化的。RabbitMQ自带的消息队列管理插件就非常不错(上一章已经讲过安装方法<http://blog.csdn.net/super_rd/article/details/70241007>)。   
浏览器打开[HTTP://IP:15672](http://ip:15672/)



最上侧的导航以此是：《概览》、《连接》、《信道》、《交换机》、《队列》、《用户管理》   
无论是服务器监控还是日常的管理基本都可以解决，很方便。既然都图形化了我就不赘述了，建议都点开看下，如果有什么参数不明白可以加最下方的QQ群探讨。

## RabbitMQ日志文件

通过日志文件方便我们定位很多问题，同样在WEB管理插件中可以看到日志文件的路径，我的是在/usr/local/rabbitmq/var/log/rabbitmq/，在该路径下有两个日志文件：rabbit@localhost.log和rabbit@localhost-sasl.log。saal（System Application Support libraries 系统应用支持库）用来记录Erlang相关的信息，举例来说可以通过这个文件看到Erlang的崩溃报告。

另外也可以通过绑定监听amq.rabbitmq.log交换机来订阅日志信息来实现更多功能。

# （五）RabbitMQ消息队列-安装amqp扩展并订阅/发布Demo（PHP版）

本文将介绍在PHP中如何使用RabbitMQ来实现消息的订阅和发布。我使用的系统依然是Centos7，为了方便，应用服务器我使用Docker进行部署，容器环境：centos7+nginx+php5.6。

## 运行环境，安装AMQP扩展：

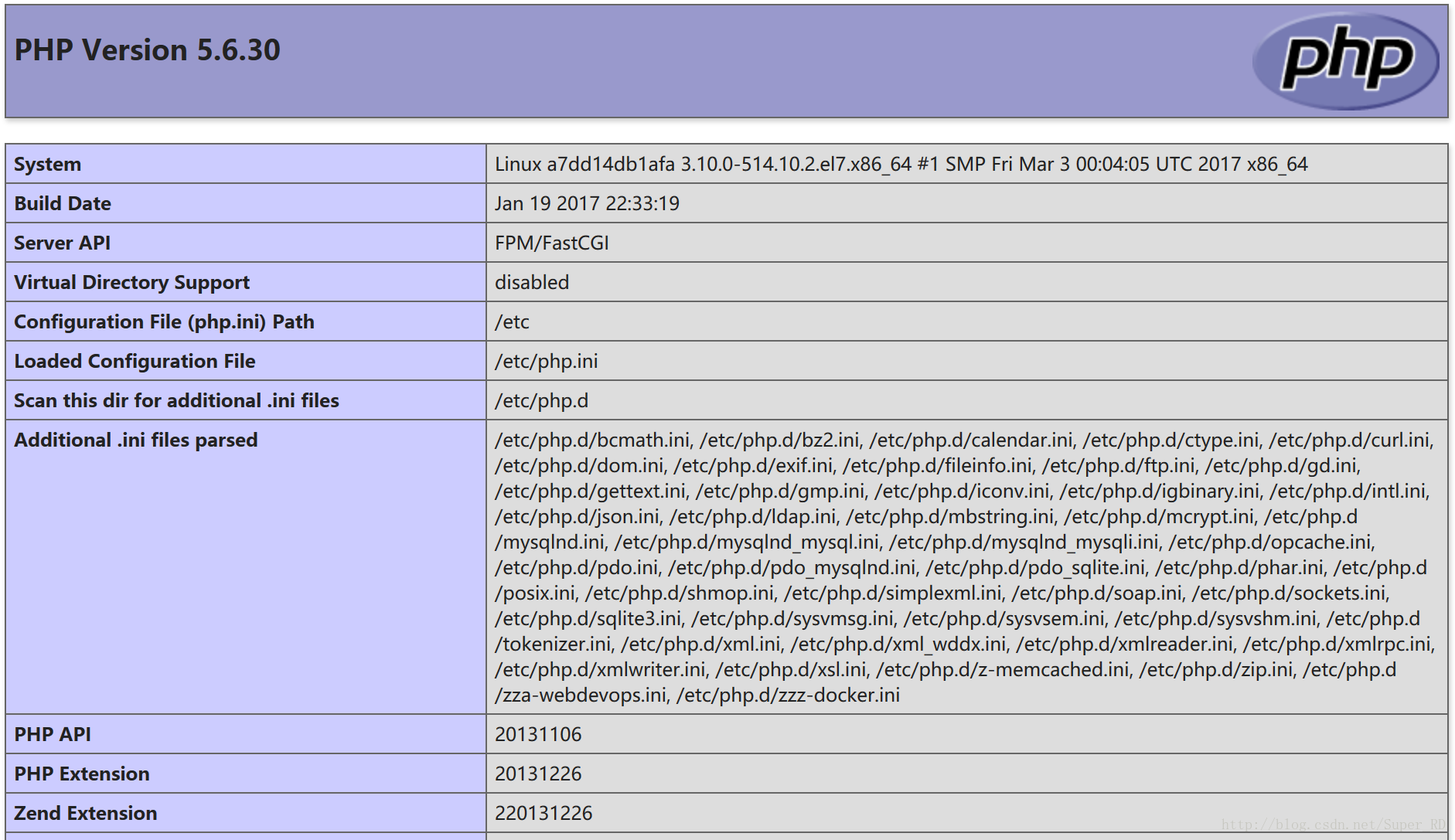
如何安装Docker我就不说了，网上很多教程非常简单，如果有现成的php环境可以直接使用。Docker中我使用的镜像名为webdevops/php-nginx，tag为：centos-7-php56。下载镜像：   
（国际带宽出口不稳定，可能会下载失败，重试记次就好了）

**docker** pull webdevops/php-nginx:centos-7-php56 //下载镜像  
**docker** run -d -p 80:80 --name rabbitmq webdevops/php-nginx:centos-7-php56 //运行容器  
**docker** exec -ti rabbitmq /bin/bash //进入容器

进入到容器后检测下环境是否有相应扩展

**cd** app  
**vi** index.php  
**<?php  
phpinfo();**

刚刚我们在运行容器的时候使用80端口，在浏览器中输入[http://ip](http://ip/)



搜索下没有amqp相关的信息。下面开始安装amqp扩展。

**yum** install gcc librabbitmq-devel.x86\_64 php56w-devel  
**-y  
wget** http://pecl.php.net/get/amqp-1.4.0.tgz  
**tar** -zxvf amqp-1.4.0.tgz  
**cd** amqp-1.4.0  
**phpize  
./configure** --with-amqp  
**make** && make install

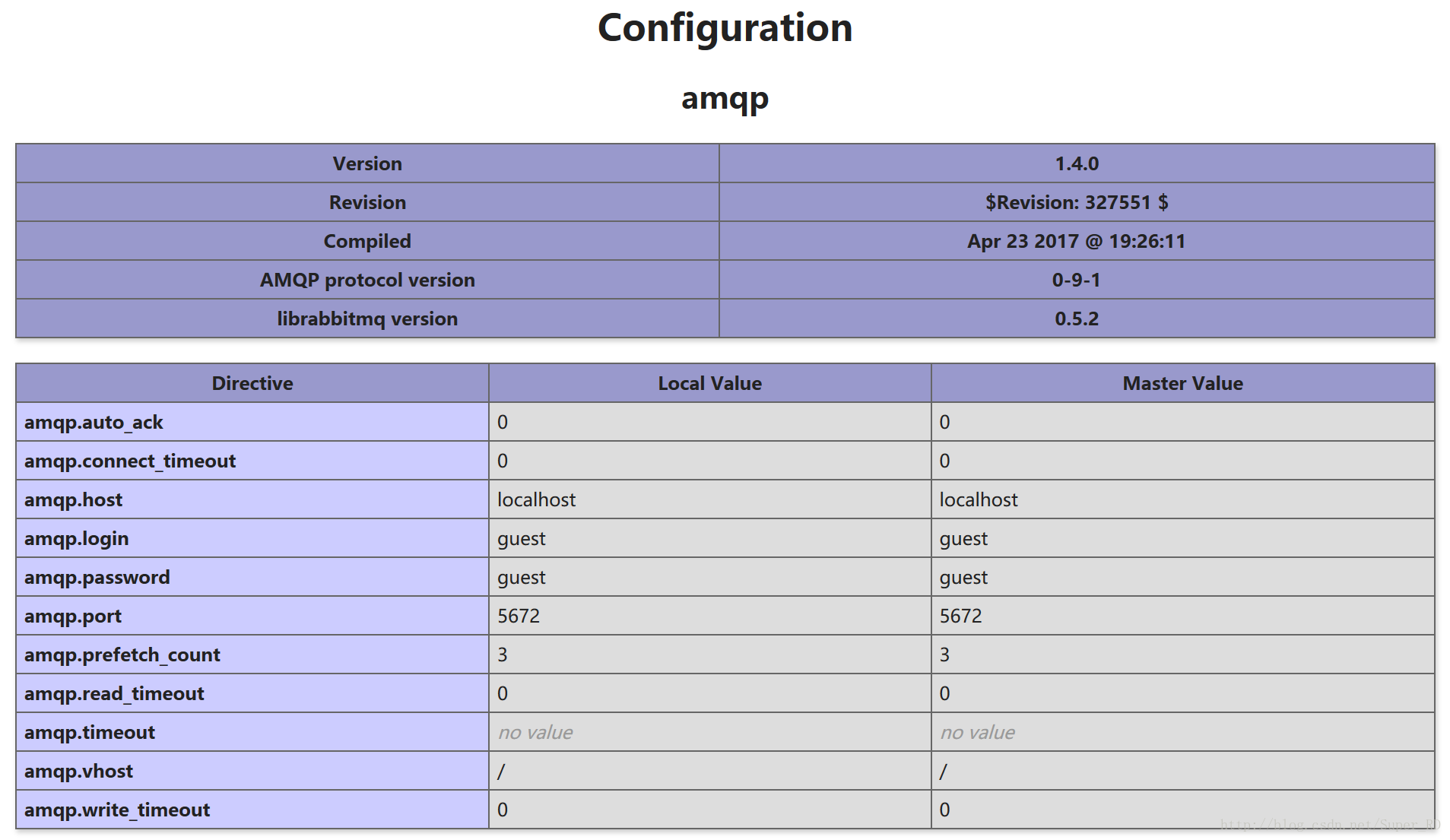
在php.ini中开启extension=amqp.so 接着重启php-fpm 或 Web服务器

**vi** /etc/php.ini  
**extension**=amqp.so

我这里就直接重启容器了，如果是宿主机直接安装php环境直接重启环境。

**exit** //退出容器  
**docker** restart rabbitmq //重启容器

再查看phpinfo，amqp扩展已经安装好了：



## publish发布消息

在/app路径下新建一个publish.php的文件

**touch** publish.php  
**vi** publish.php

以下是PHP代码，我们先定义好用来发消息的交换机、队列、RoutingKey、消息等变量。

**$queueName** = 'superrd';  
**$exchangeName** = 'superrd';  
**$routeKey** = 'superrd';  
**$message** = 'Hello World!';

按照我们第二章讲到的首先建立一个连接。

**$connection** = new AMQPConnection(array('host' => '10.99.121.137', 'port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");

新建一个信道。

**$channel** = new AMQPChannel($connection);

新建一个交换机Exchange，并定义属性，第二章我们讲过有四种类型的交换机，这里使用直连型DIRECT。AMQP\_DURABLE代表这是一个持久化的交换机，不会以为服务器异常等因素丢失。

**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_DIRECT);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();**

新建一个队列Queue，前面也讲过生产者将消息发送到Exchange中，Exchange会根据绑定关系投递到队列，也就是如果生产者在生产消息时没有队列与之绑定消息就会丢失。为了保证系统更加健硕，一般无论是消息的生产者还是消费者都会新建一遍Exchange和Queue，新建后属性不会改变。同样AMQP\_DURABLE代表这是一个持久化的队列，队列会被写入磁盘。需要注意的是虽然消息是缓存在队列中，但是并不是队列是持久化的队列队列中的消息就是持久化的，消息的持久化需要单独设置。

**$queue** = new AMQPQueue($channel);  
**$queue->setName($queueName);  
$queue->setFlags(AMQP\_DURABLE);  
$queue->declareQueue();**

通过routeKey绑定交换机和队列。

**$queue->bind($exchangeName,** $routeKey);

好了，下面可以发送消息了

**$exchange->publish($message,$routeKey);**

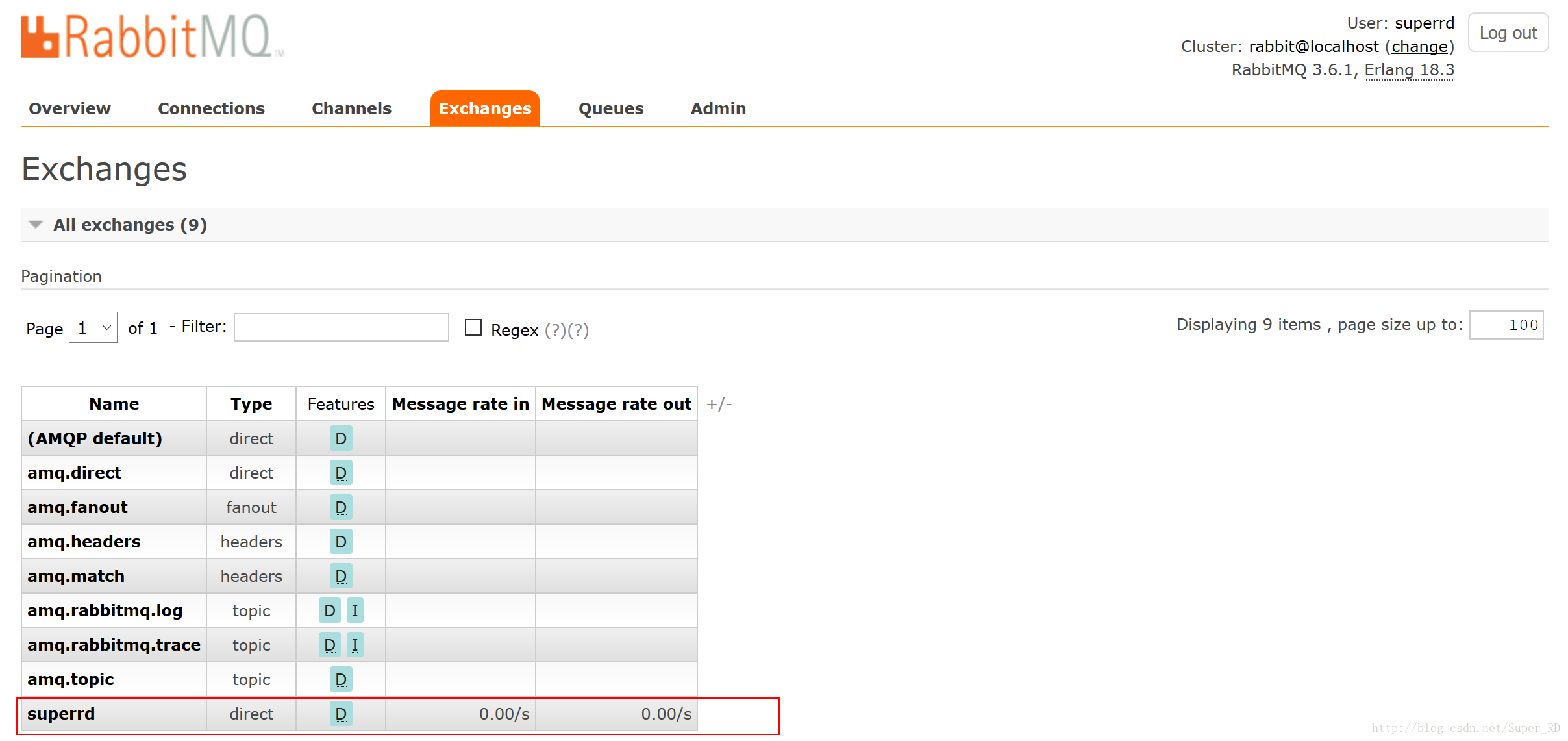
如果你希望消息也是持久化的可以使用如下的代码，实际测试结果在持久化消息后消息发布的性能下降一倍，我的磁盘是pcie的固态硬盘，如果你是机械磁盘这个性能下降估计会更明显，24核心CPU，48GB内存，pcie固态硬盘，单线程的情况下每秒可以发布2.5万左右的非持久化消息，持久化之后变为变为1.2万左右。

**$exchange->publish($message,$routeKey,AMQP\_NOPARAM,** array('delivery\_mode'=>2));

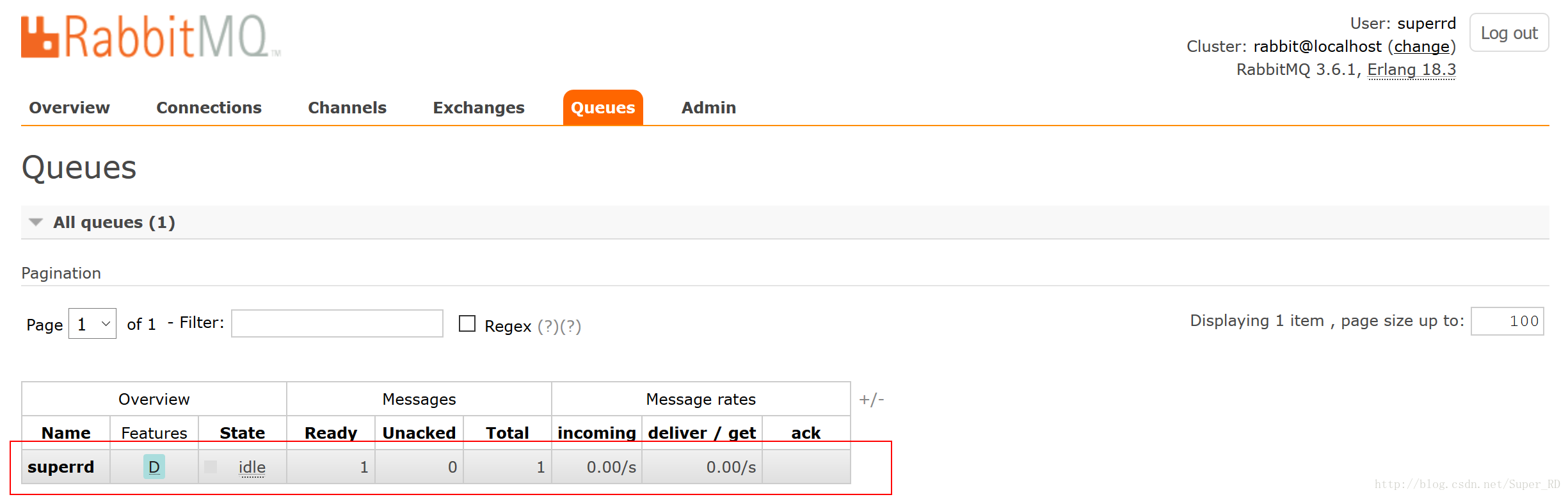
断开连接。

**$connection->disconnect();**

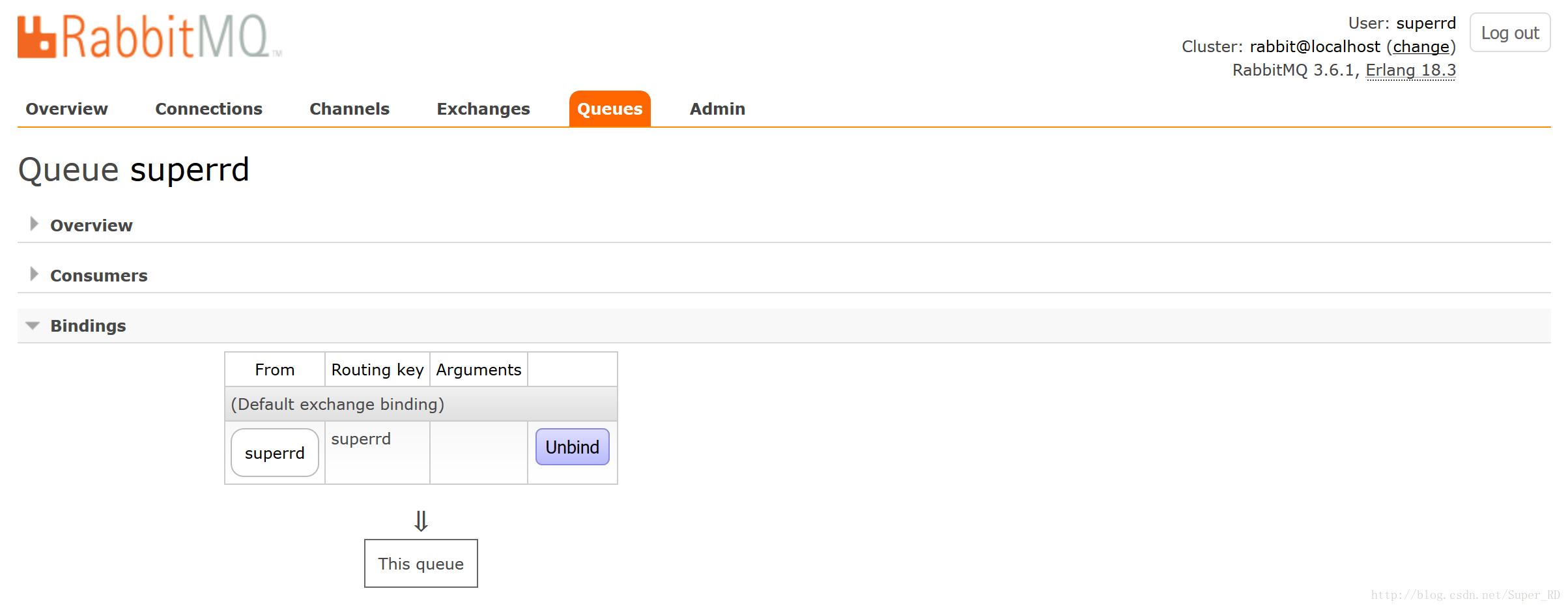
同样在发布消息之后可以通过WEB工具来查看是否发布成功，   
查看交换机多了一个superid交换机。



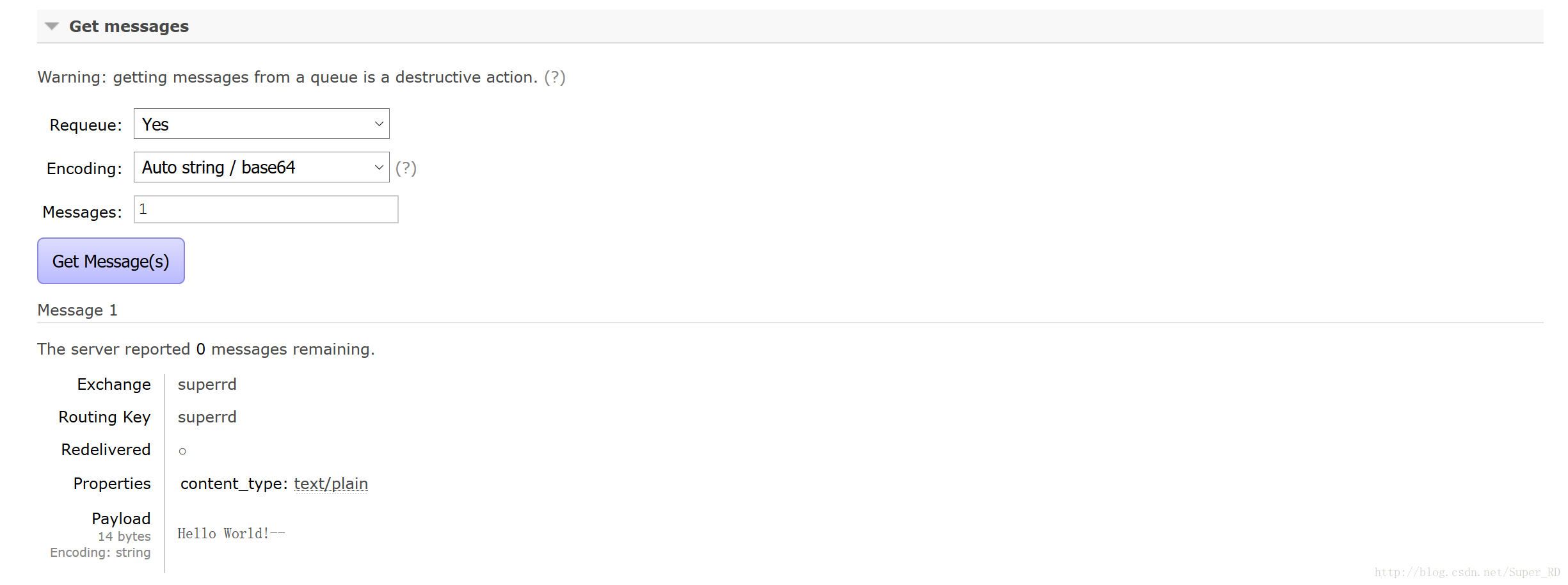
查看交换机已经有superrd队列。



点击队列查看队列详情。Bindings标签可以看到交换机和队列的绑定关系。



点击Get messages标签Get message（s）按钮可以看到队列中的消息。



到此说明我们已经将一个消息发布到了消息队列中。完整的PHP代码如下。

**<?php  
  
/\*  
\*** 发布-订阅  
**\*** create by superrd  
**\*/  
  
$queueName** = 'superrd';  
**$exchangeName** = 'superrd';  
**$routeKey** = 'superrd';  
**$message** = 'Hello World!';  
**$connection** = new AMQPConnection(array('host' => '10.99.121.137', 'port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
**try** {  
**$channel** = new AMQPChannel($connection);  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_DIRECT);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
$queue** = new AMQPQueue($channel);  
**$queue->setName($queueName);  
$queue->setFlags(AMQP\_DURABLE);  
$queue->declareQueue();  
  
$queue->bind($exchangeName,** $routeKey);  
  
**$exchange->publish($message,$routeKey);  
  
var\_dump("[x]** Sent 'Hello World!'");  
  
**}** catch (AMQPConnectionException $e) {  
**var\_dump($e);  
exit();  
}  
$connection->disconnect();**

## Subscribe订阅消息

在/app路径下新建一个subscribe.php的文件

**touch** subscribe.php  
**vi** subscribe.php

以下是PHP代码，和发布消息一样我们先定义好用交换机、队列、RoutingKey等变量。

**$queueName** = 'superrd';  
**$exchangeName** = 'superrd';  
**$routeKey** = 'superrd';

按照我们第二章讲到的首先建立一个连接。

**$connection** = new AMQPConnection(array('host' => '10.99.121.137', 'port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");

新建一个信道。

**$channel** = new AMQPChannel($connection);

与发布消息一样新建交换机。

**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_DIRECT);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();**

新建一个队列Queue。

**$queue** = new AMQPQueue($channel);  
**$queue->setName($queueName);  
$queue->setFlags(AMQP\_DURABLE);  
$queue->declareQueue();**

通过routeKey绑定交换机和队列。

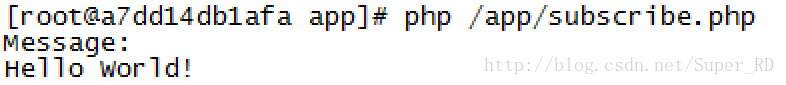
**$queue->bind($exchangeName,** $routeKey);

重点来了，阻塞订阅消息。

**//阻塞模式接收消息  
  
echo** "Message:**\n**";  
**while(True){  
$queue->consume('processMessage');  
//自动ACK应答  
//$queue->consume('processMessage',** AMQP\_AUTOACK);   
**}  
  
$conn->disconnect();  
/\*  
\*** 消费回调函数  
**\*** 处理消息  
**\*/  
function** processMessage($envelope, $q) {  
**$msg** = $envelope->getBody();  
**echo** $msg."**\n**"; //处理消息  
**$q->ack($envelope->getDeliveryTag());** //手动发送ACK应答  
**}**

注意因为是阻塞监听，因为输出缓冲区的原因用浏览器访问该文件是看不到输出的。使用脚本访问。

**php** /app/subscribe.php



通过WEB工具查看队列。superrd队列中的消息数已经为0。



完整的PHP代码如下。

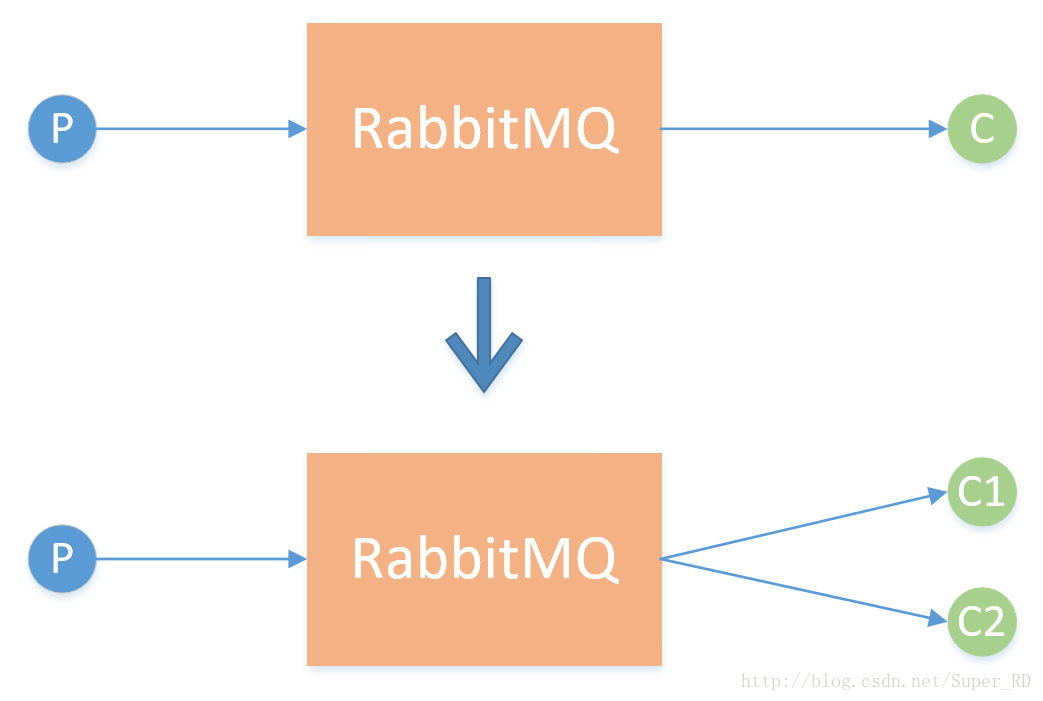
**<?php  
/\*  
\*** 发布-订阅  
**\*** create by superrd  
**\*/  
$queueName** = 'superrd';  
**$exchangeName** = 'superrd';  
**$routeKey** = 'superrd';  
**$connection** = new AMQPConnection(array('host' => '10.99.121.137', 'port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
  
**$channel** = new AMQPChannel($connection);  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_DIRECT);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
$queue** = new AMQPQueue($channel);  
**$queue->setName($queueName);  
$queue->setFlags(AMQP\_DURABLE);  
$queue->declareQueue();  
  
$queue->bind($exchangeName,** $routeKey);  
  
**//阻塞模式接收消息  
  
echo** "Message:**\n**";  
**while(True){  
$queue->consume('processMessage');  
//自动ACK应答  
//$queue->consume('processMessage',** AMQP\_AUTOACK);  
**}  
  
$conn->disconnect();  
  
/\*  
\*** 消费回调函数  
**\*** 处理消息  
**\*/  
function** processMessage($envelope, $q) {  
**$msg** = $envelope->getBody();  
**echo** $msg."**\n**"; //处理消息  
**$q->ack($envelope->getDeliveryTag());** //手动发送ACK应答  
**}**

# （六）RabbitMQ消息队列-消息任务分发与消息ACK确认机制（PHP版）

在前面一章介绍了在PHP中如何使用RabbitMQ，至此入门的的部分就完成了，我们内心中一定还有很多疑问：如果多个消费者消费同一个队列怎么办？如果这几个消费者分任务的权重不同怎么办？怎么把同一个队列不同级别的任务分发给不同的消费者？如果消费者异常离线怎么办？不要着急，后面将慢慢解开面纱。我们将结合实际的应用场景来讲解更多的高级用法。

## 任务分发机制

设想如果把每个消息当做一个任务，生产者把任务发布到RabbitMQ，然后Consumer接收消息处理任务，如果我们发现一个Consumer不能完成任务处理怎么办呢，我们会增加Consumer的数量。由一个Consumer增加到两个Consumer，如图由C变为C1和C2共同来分单工作。如果C1和C2是完全一样的，那RabbitMQ会将任务平均分发到两个消费者。



如下我们新建c1.php和c2.php来订阅同一个队列在接收到消息后sleep1秒模拟任务处理的时间。

p.php代码，生产100条带编号的消息：

**<?php  
  
/\*  
\*** 发布-订阅-P  
**\*** create by superrd  
**\*/  
  
$queueName** = 'superrd';  
**$exchangeName** = 'superrd';  
**$routeKey** = 'superrd';  
**$message** = 'task--';  
**$connection** = new AMQPConnection(array('host' => '10.99.121.137', 'port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
**try** {  
**$channel** = new AMQPChannel($connection);  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_DIRECT);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
$queue** = new AMQPQueue($channel);  
**$queue->setName($queueName);  
$queue->setFlags(AMQP\_DURABLE);  
$queue->declareQueue();  
  
$queue->bind($exchangeName,** $routeKey);  
  
**for($i**=0 ; $i<100;$i++){  
**$exchange->publish($message.$i,$routeKey);  
var\_dump("[x]** Sent $message $i");  
**}  
}** catch (AMQPConnectionException $e) {  
**var\_dump($e);  
exit();  
}  
$connection->disconnect();**

c1.php和c2.php代码完全一样：

**<?php  
  
/\*  
\*** 发布-订阅-c1c2  
**\*** create by superrd  
**\*/  
  
$queueName** = 'superrd';  
**$exchangeName** = 'superrd';  
  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
**$channel** = new AMQPChannel($connection);  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_DIRECT);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
$queue** = new AMQPQueue($channel);  
**$queue->setName($queueName);  
$queue->setFlags(AMQP\_DURABLE);  
$queue->declareQueue();  
  
$queue->bind($exchangeName,** $routeKey);  
**//阻塞模式接收消息  
  
echo** "Message:**\n**";  
**while(True){  
$queue->consume('processMessage');  
//自动ACK应答  
//$queue->consume('processMessage',** AMQP\_AUTOACK);  
**}  
  
$conn->disconnect();  
/\*  
\*** 消费回调函数  
**\*** 处理消息  
**\*/  
function** processMessage($envelope, $q) {  
**$msg** = $envelope->getBody();  
**sleep(1);** //sleep1秒模拟任务处理  
**echo** $msg."**\n**"; //处理消息  
**$q->ack($envelope->getDeliveryTag());** //手动发送ACK应答  
**}**

打开两个中断窗口分别执行c1.php和c2.php脚本。确定两个脚本处于订阅状态，然后执行p.php脚本。





看到上面两幅图结果就一目了然了。因为两个脚本sleep的时间相同所以任务是完全平均分发到两个消费者的。我们修改下c2.php脚本的sleep时间为2秒，看下结果会怎么样。





可以看到c1.php脚本共收到66条消息，c2.php脚本收到34条消息，基本是按照2:1来分配。那RabbitMQ是如何来保证这样的分发机制呢，下面看RabbitMQ是如何通过ACK确认机制来实现任务分发的。

## ACK消息确认机制

首先RabbitMQ支持消息确认机制来本证消息被consumer正常处理，当然也可以通过no-ack不使用确认机制。RabbitMQ默认是使用ACK确认机制的。当Consumer接收到RabbitMQ发布的消息时需要在适当的时机发送一个ACK确认的包来告知RabbitMQ，自己接收到了消息并成功处理。所以前面讲到适当的时机建议是在处理完消息任务后发送。正如我们之前的代码。

**$msg** = $envelope->getBody();  
**sleep(1);** //sleep1秒模拟任务处理  
**echo** $msg."**\n**"; //处理消息  
**$q->ack($envelope->getDeliveryTag());** //手动发送ACK应答

那如果不发送会怎样呢？

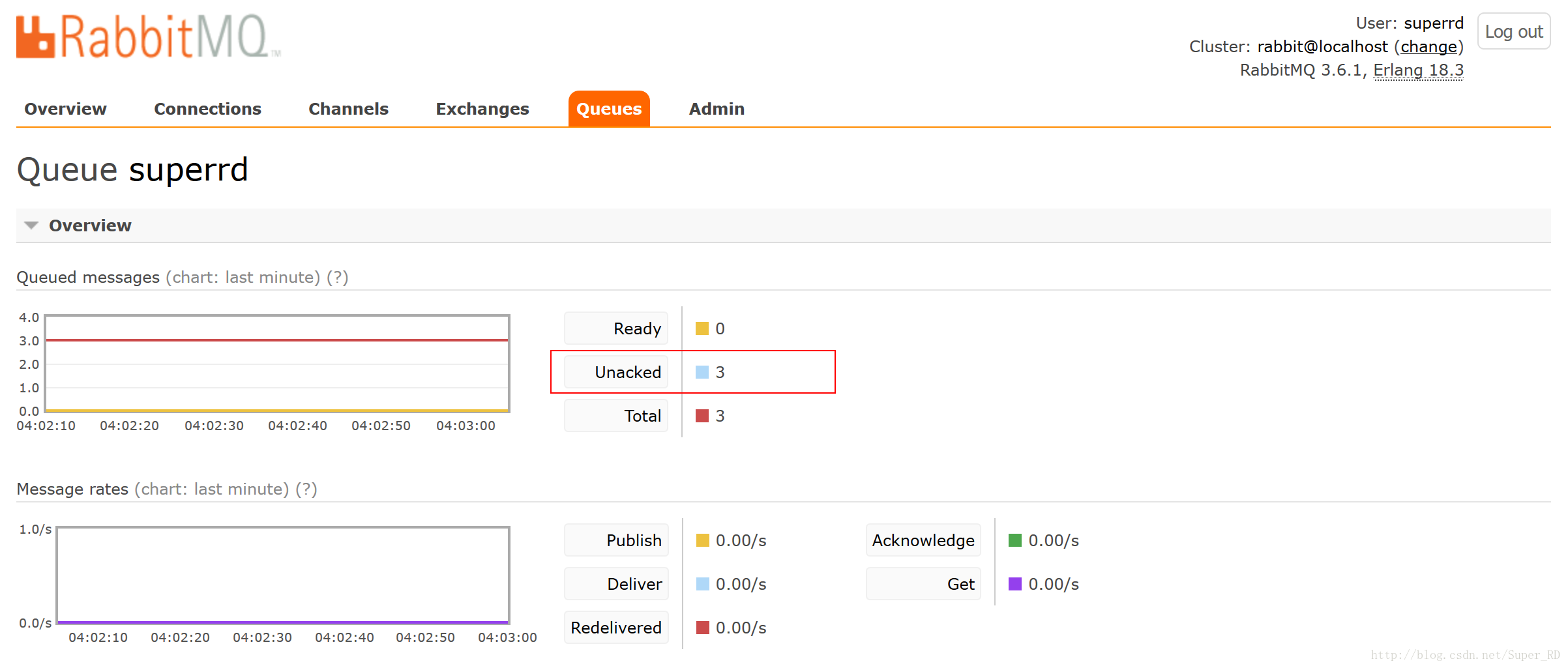
在RabbitMQ中有一个prefetch\_count的概念，这个参数的意思是允许Consumer最多同时处理几个任务。我的版本的RabbitMQ默认这个参数是3，也就是说如果某一个Consumer在收到消息后没有发送ACK确认包，RabbitMQ就会任务Consumer还在处理任务，当有3个消息都没有发送ACK确认包时，RabbitMQ就不会再发送消息给该Consumer。   
我们把c2.php的sleep时间改回1秒，并且注释掉ACK确认。

**$msg** = $envelope->getBody();  
**sleep(1);** //sleep1秒模拟任务处理  
**echo** $msg."**\n**"; //处理消息  
**//$q->ack($envelope->getDeliveryTag());** //手动发送ACK应答





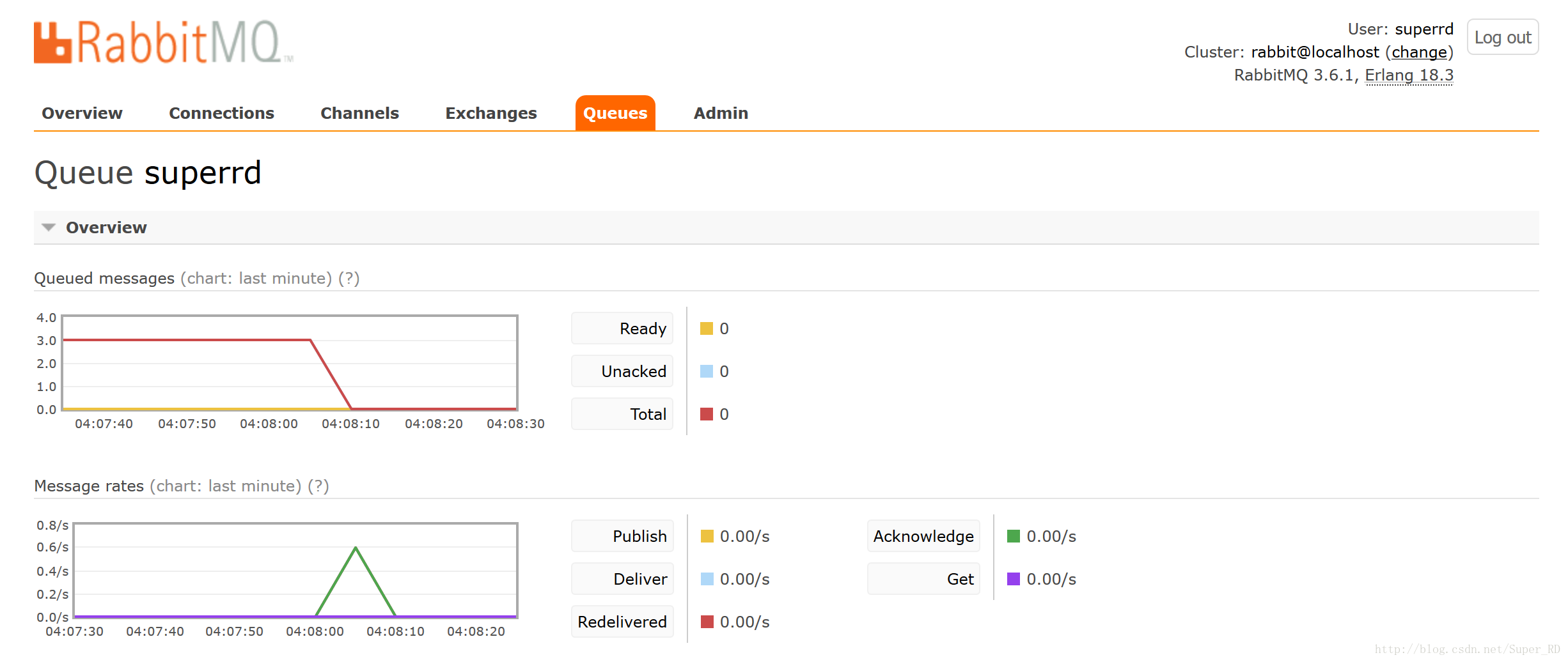
发现c2脚本只收到三条消息。通过WEB管理工具也可以看到有三条消息是没有被ACK确认的。



当然任务并不会一直卡在这里，在这是RabbitMQ任务c2在处理这三个任务。如果c2忽然终止RabbitMQ会重新分发任务。如下我终止c2脚本。



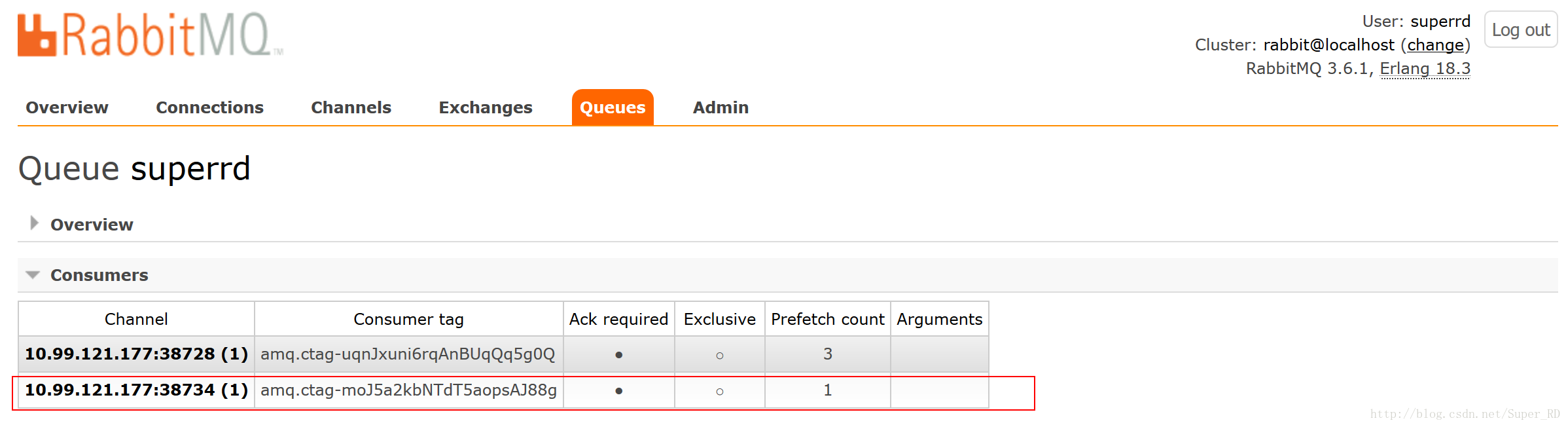
三条任务被重新分发到了c1。再查看下WEB管理工具，unackd已经为0



如果Consumer数量很多或者希望每个Consumer同时只处理一个任务可以通过在Consumer中设置PrefetchCount来实现更加均匀的任务分发。

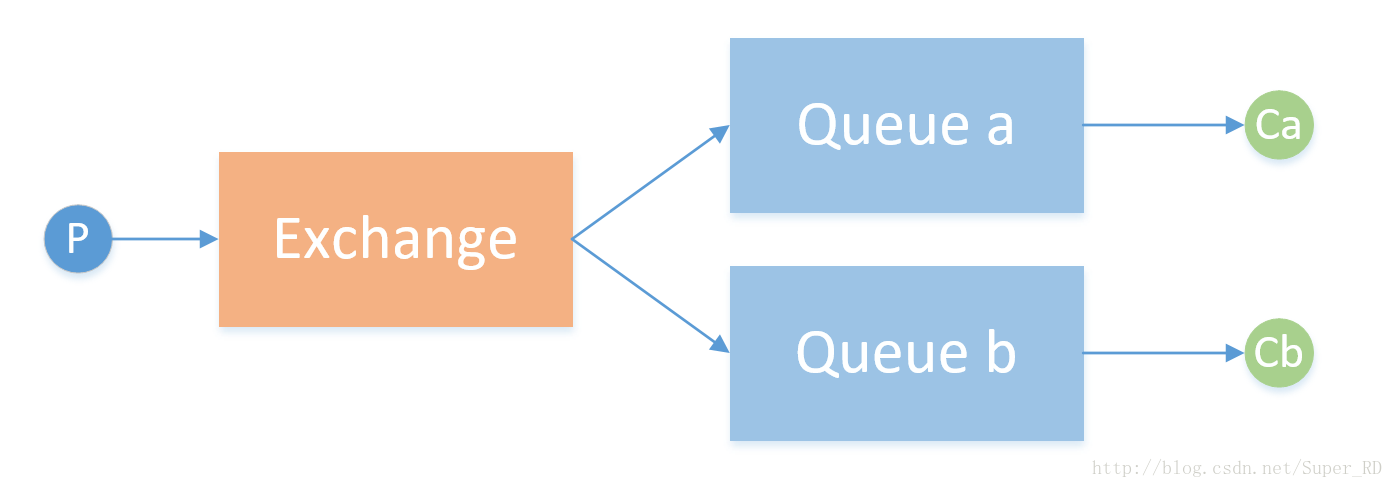
**$channel** = new AMQPChannel($connection);  
**$channel->setPrefetchCount(1);**

如下我修改了c2的PrefetchCount为1。在WEB管理插件中可以看到已经有一个Consumer的PrefetchCount为1了。



# （七）RabbitMQ消息队列-通过fanout模式将消息推送到多个Queue中

前面第六章我们使用的是direct直连模式来进行消息投递和分发。本章将介绍如何使用fanout模式将消息推送到多个队列。   
有时我们会遇到这样的情况，多个功能模块都希望得到完整的消息数据。例如一个log的消息，一个我们希望输出在屏幕上实时监控，另外一个用户持久化日志。这时就可以使用fanout模式。fanout模式模式不像direct模式通过routingkey来进行匹配，而是会把消息发送到所以的已经绑定的队列中。



新建fanout.php用来发布消息。ca.php和cb.php用来订阅不同队列消费消息。

fanout.php

**<?php  
  
/\*  
\*** RabbitMQ fanout  
**\*** create by superrd  
**\*/  
  
$exchangeName** = 'log';  
**$message** = 'log--';  
**$connection** = new AMQPConnection(array('host' => '10.99.121.137', 'port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
**try** {  
**$channel** = new AMQPChannel($connection);  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_FANOUT);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
for($i**=0 ; $i<100;$i++){  
**$exchange->publish($message.$i,"");  
  
var\_dump("[x]** Sent $message $i");  
**}  
}** catch (AMQPConnectionException $e) {  
**var\_dump($e);  
exit();  
}  
$connection->disconnect();**

ca.php

**<?php  
/\*  
\*** RabbitMQ fanout 模式  
**\*** create by superrd  
**\*/  
  
$exchangeName** = 'log';  
**$queueName** = 'queuea';  
**$routeKey** = '';  
  
**$connection** = new AMQPConnection(array('host' => '10.99.121.137','port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
  
**$channel** = new AMQPChannel($connection);  
  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_FANOUT);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
$queue** = new AMQPQueue($channel);  
**$queue->setName($queueName);  
$queue->setFlags(AMQP\_DURABLE);  
$queue->declareQueue();  
$queue->bind($exchangeName,** $routeKey);  
  
**//阻塞模式接收消息  
echo** "Message:**\n**";  
**while(True){  
$queue->consume('processMessage');  
//$queue->consume('processMessage',** AMQP\_AUTOACK); //自动ACK应答  
**}  
  
$conn->disconnect();  
/\*\*  
\*** 消费回调函数  
**\*** 处理消息  
**\*/  
function** processMessage($envelope, $q) {  
**$msg** = $envelope->getBody();  
**sleep(1);  
echo** $msg."**\n**"; //处理消息  
**$q->ack($envelope->getDeliveryTag());** //手动发送ACK应答  
**}**

cb.php

**/\*  
\*** RabbitMQ fanout 模式  
**\*** create by superrd  
**\*/  
  
$exchangeName** = 'log';  
**$queueName** = 'queueb';  
**$routeKey** = '';  
  
**$connection** = new AMQPConnection(array('host' => '10.99.121.137','port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
  
**$channel** = new AMQPChannel($connection);  
  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_FANOUT);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
$queue** = new AMQPQueue($channel);  
**$queue->setName($queueName);  
$queue->setFlags(AMQP\_DURABLE);  
$queue->declareQueue();  
$queue->bind($exchangeName,** $routeKey);  
  
**//阻塞模式接收消息  
echo** "Message:**\n**";  
**while(True){  
$queue->consume('processMessage');  
//$queue->consume('processMessage',** AMQP\_AUTOACK); //自动ACK应答  
**}  
  
$conn->disconnect();  
/\*\*  
\*** 消费回调函数  
**\*** 处理消息  
**\*/  
function** processMessage($envelope, $q) {  
**$msg** = $envelope->getBody();  
**sleep(1);  
echo** $msg."**\n**"; //处理消息  
**$q->ack($envelope->getDeliveryTag());** //手动发送ACK应答  
**}**





可以看到ca和cb收到的消息完全一致。注意以上代码fanout.php中并没有新建队列，所以先运行ca.php和cb.php的脚本，如果先运行fanout.php因为找不到绑定的队列数据就会丢失。   
还有一种情况我们有可能随时增加一项处理机制，如果在声明queue时不指定名字，那么RabbitMQ会随机为我们生成一个名字，如果不指定queue为持久化队列那在消息为空并且订阅者为0时自动删除该队列。这样Queue挥之即来呼之即去。

**<?php  
/\*  
\*** RabbitMQ fanout 模式  
**\*** create by superrd  
**\*/  
  
$exchangeName** = 'log';  
  
**$connection** = new AMQPConnection(array('host' => '10.99.121.137','port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
  
**$channel** = new AMQPChannel($connection);  
  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_FANOUT);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
$queue** = new AMQPQueue($channel);  
**$queue->declareQueue();  
$queue->bind($exchangeName,** $routeKey);  
  
**//阻塞模式接收消息  
echo** "Message:**\n**";  
**while(True){  
$queue->consume('processMessage');  
//$queue->consume('processMessage',** AMQP\_AUTOACK); //自动ACK应答  
**}  
  
$conn->disconnect();  
/\*\*  
\*** 消费回调函数  
**\*** 处理消息  
**\*/  
function** processMessage($envelope, $q) {  
**$msg** = $envelope->getBody();  
**sleep(1);  
echo** $msg."**\n**"; //处理消息  
**$q->ack($envelope->getDeliveryTag());** //手动发送ACK应答  
**}**

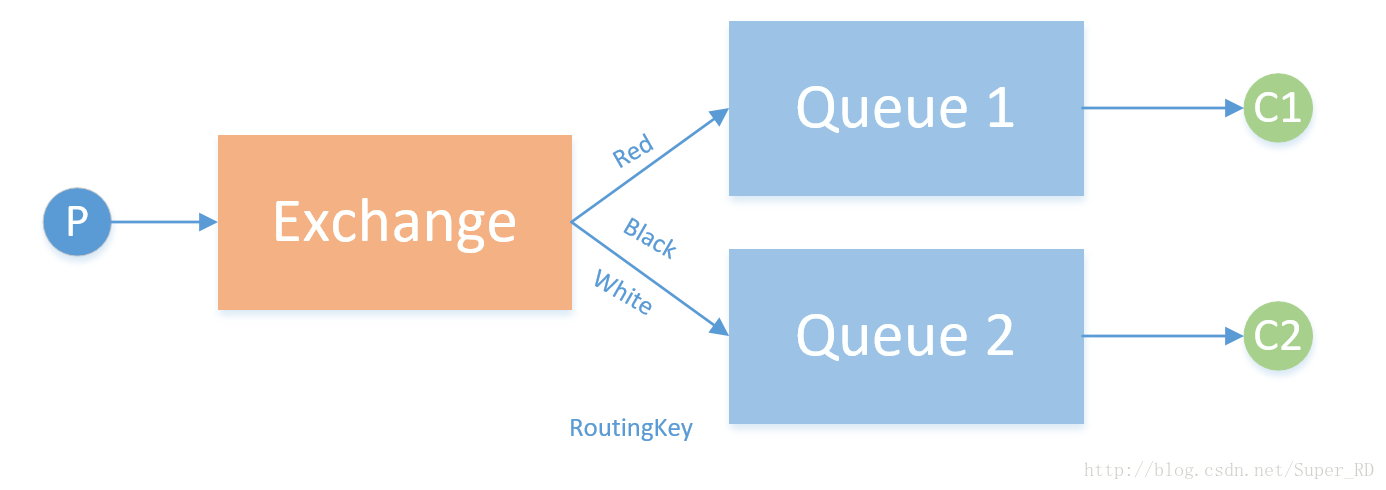
# （八）RabbitMQ消息队列-通过Topic主题模式分发消息

前两章我们讲了RabbitMQ的direct模式和fanout模式，本章介绍topic主题模式的应用。如果对direct模式下通过routingkey来匹配消息的模式已经有一定了解那fanout也很好理解。简单的可以理解成direct是通过routingkey精准匹配的，而topic是通过routingkey来模糊匹配。   
在topic模式下支持两个特殊字符的匹配。

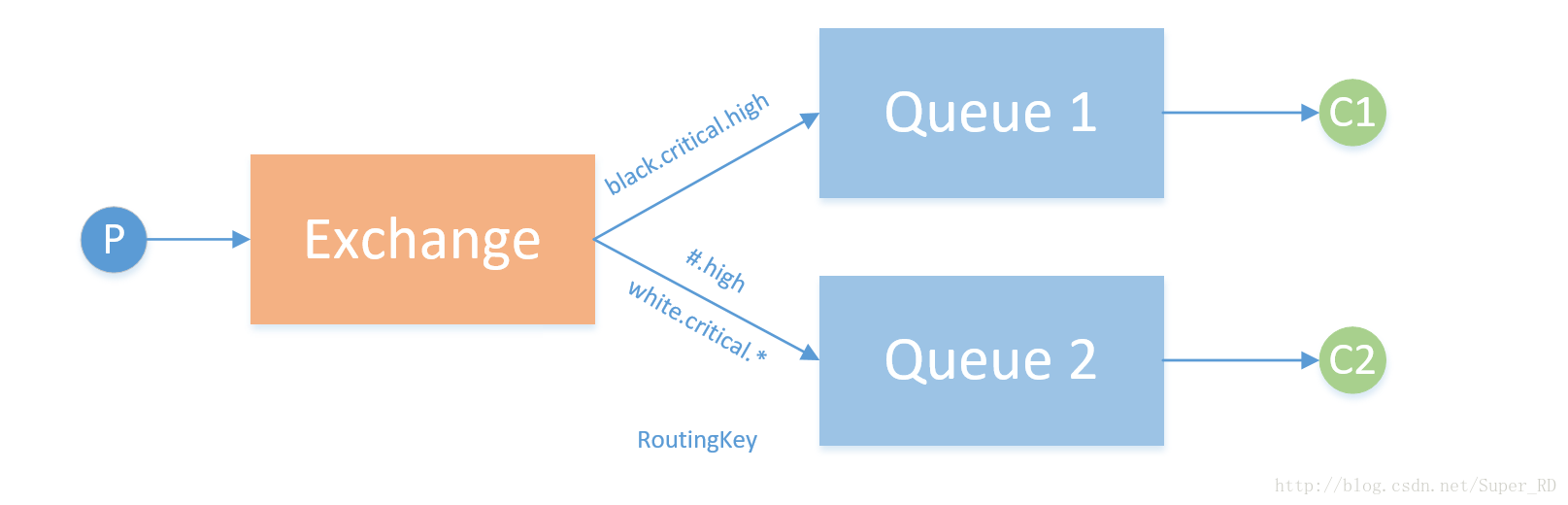
**\*** (星号) 代表任意 一个单词  
# (井号) 0个或者多个单词

注意：上面说的是单词不是字符。

如下图所示，RabbitMQ direct模式通过RoutingKey来精准匹配，RoutingKey为red的投递到Queue1，RoutingKey为black和white的投递到Queue2。



我们可以假设一个场景，我们要做一个日志模块来收集处理不同的日志，日志区分包含三个维度的标准：模块、日志紧急程度、日志重要程度。模块分为：red、black、white；紧急程度分为：critical、normal；把重要程度分为：medium、low、high在RoutingKey字段中我们把这三个维度通过两个“.“连接起来。   
现在我们需要对black模块，紧急程度为critical，重要程度为high的日志分配到队列1打印到屏幕；对所以模块重要程度为high的日志和white紧急程度为critical的日志发送到队列2持久化到硬盘。如下示例：



* RoutingKey为“black.critical.high”的日志会投递到queue1和queue2,。
* RoutingKey为“red.critical.high”的日志会只投递到queue2。
* RoutingKey为“white.critical.high”的日志会投递到queue2，并且虽然queue2的两个匹配规则都符合但只会向queue2投递一份。

新建topic.php用来发布三种routingkey的消息。

**<?php  
  
/\*  
\*** topic 模式  
**\*** create by superrd  
**\*/  
  
$exchangeName** = 'extopic';  
**$routeKey1** = "black.critical.high";  
**$routeKey2** = "red.critical.high";  
**$routeKey3** = "white.critical.high";  
  
**$message1** = 'black-critical-high!';  
**$message2** = 'red-critical-high!';  
**$message3** = 'white-critical-high!';  
  
**$connection** = new AMQPConnection(array('host' => '10.99.121.137', 'port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
**try** {  
**$channel** = new AMQPChannel($connection);  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_TOPIC);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
$exchange->publish($message1,$routeKey1);  
var\_dump("[x]** Sent ".$message1);  
**$exchange->publish($message2,$routeKey2);  
var\_dump("[x]** Sent ".$message2);  
**$exchange->publish($message3,$routeKey3);  
var\_dump("[x]** Sent ".$message3);  
  
**}** catch (AMQPConnectionException $e) {  
**var\_dump($e);  
exit();  
}  
$connection->disconnect();**

q1.php用来监听queue1队列：

**<?php  
  
/\*  
\*** topic 模式  
**\*** create by superrd  
**\*/  
  
$queueName** = 'queue1';  
**$exchangeName** = 'extopic';  
**$routeKey** = "black.critical.high";  
  
**$connection** = new AMQPConnection(array('host' => '10.99.121.137', 'port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
  
**$channel** = new AMQPChannel($connection);  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_TOPIC);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
$queue** = new AMQPQueue($channel);  
**$queue->setName($queueName);  
$queue->setFlags(AMQP\_DURABLE);  
$queue->declareQueue();  
  
$queue->bind($exchangeName,** $routeKey);  
  
**//阻塞模式接收消息  
  
echo** "Message:**\n**";  
**while(True){  
$queue->consume('processMessage');  
//自动ACK应答  
//$queue->consume('processMessage',** AMQP\_AUTOACK);  
**}  
  
$conn->disconnect();  
/\*  
\*** 消费回调函数  
**\*** 处理消息  
**\*/  
function** processMessage($envelope, $q) {  
**$msg** = $envelope->getBody();  
**echo** $msg."**\n**"; //处理消息  
**$q->ack($envelope->getDeliveryTag());** //手动发送ACK应答  
**}**

q2.php用来监听queue2队列：

**<?php  
  
/\*  
\*** topic 模式  
**\*** create by superrd  
**\*/  
  
$queueName** = 'queue2';  
**$exchangeName** = 'extopic';  
**$routeKey1** = "#.high";  
**$routeKey2** = "white.critical.\*";  
  
**$connection** = new AMQPConnection(array('host' => '10.99.121.137', 'port' => '5672', 'vhost' => '/', 'login' => 'superrd', 'password' => 'superrd'));  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
  
**$channel** = new AMQPChannel($connection);  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_TOPIC);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
$queue** = new AMQPQueue($channel);  
**$queue->setName($queueName);  
$queue->setFlags(AMQP\_DURABLE);  
$queue->declareQueue();  
  
$queue->bind($exchangeName,** $routeKey1);  
**$queue->bind($exchangeName,** $routeKey2);  
  
  
**//阻塞模式接收消息  
  
echo** "Message:**\n**";  
**while(True){  
$queue->consume('processMessage');  
//自动ACK应答  
//$queue->consume('processMessage',** AMQP\_AUTOACK);  
**}  
  
$conn->disconnect();  
/\*  
\*** 消费回调函数  
**\*** 处理消息  
**\*/  
function** processMessage($envelope, $q) {  
**$msg** = $envelope->getBody();  
**echo** $msg."**\n**"; //处理消息  
**$q->ack($envelope->getDeliveryTag());** //手动发送ACK应答  
**}**

先运行q1.php和q2.php脚本保持订阅状态。然后执行topic.php脚本发布消息。q1和q2收到的消息如下：





如上截图，验证了我们之前的结论。

另外还有一些特殊情况例如：

* 如果binding\_key 是 “#” - 它会接收所有的Message，不管routing\_key是什么，就像是fanout exchange。
* 如果 “\*” and “#” 没有被使用，那么topic exchange就变成了direct exchange。

# （九）RabbitMQ消息队列-通过Headers模式分发消息

Headers类型的exchange使用的比较少，以至于官方文档貌似都没提到，它是忽略routingKey的一种路由方式。是使用Headers来匹配的。Headers是一个键值对，可以定义成Hashtable。发送者在发送的时候定义一些键值对，接收者也可以再绑定时候传入一些键值对，两者匹配的话，则对应的队列就可以收到消息。   
匹配有两种方式all和any。这两种方式是在接收端必须要用键值”x-mactch”来定义。all代表定义的多个键值对都要满足，any代表只要满足一个就可以。fanout，direct，topic exchange的routingKey都需要要字符串形式的，而headers exchange则没有这个要求，因为键值对的值可以是任何类型。   
那在PHP中怎么应用呢？

# （十）RabbitMQ消息队列-高可用集群部署实战

前几章讲到RabbitMQ单主机模式的搭建和使用，我们在实际生产环境中出于对性能还有可用性的考虑会采用集群的模式来部署RabbitMQ。

## RabbitMQ集群基本概念

Rabbit模式大概分为以下三种：单主机模式、普通集群模式、镜像集群模式。

* 单主机模式：

RabbitMQ服务运行在单独的一台主机中，通常生产环境不使用该模式，性能有限，并且如果服务器宕机服务将完全不可用。

* 普通集群模式

一说到集群问题瞬间变得复杂多了。首先对于Queue来说消息实体只存在于其中一个节点，集群中其他节点仅有相同的元数据，即队列结构。

当消息进入A节点的Queue中后，Consumer从B节点拉取消息时，RabbitMQ会临时在两个节点间进行消息传输，把A中的消息实体取出并经过B发送给Consumer。所以Consumer应尽量连接每一个节点，从中取消息。即对于同一个逻辑队列，要在多个节点建立Queue。否则Consumer如果只连接一个节点区消息会造成该节点的性能瓶颈。

该模式存在一个问题就是当其中一个节点故障后，其他节点无法取到故障节点中还未消费的消息。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，那就杯具了！

* 镜像集群模式

前面讲到RabbitMQ的普通集群模式不同节点间只同步队列结构不同步消息。镜像模式会把队列结构和消息都存在于多个节点，属于RabbitMQ的HA方案。其实质和普通模式不同之处在于，消息实体会主动在镜像节点间同步。该模式带来的副作用也很明显，除了降低系统性能外，如果镜像队列数量过多，加之大量的消息进入，集群内部的网络带宽将会被这种同步通讯大量消耗。所以这种模式应用于可靠性要求较高的场合中。

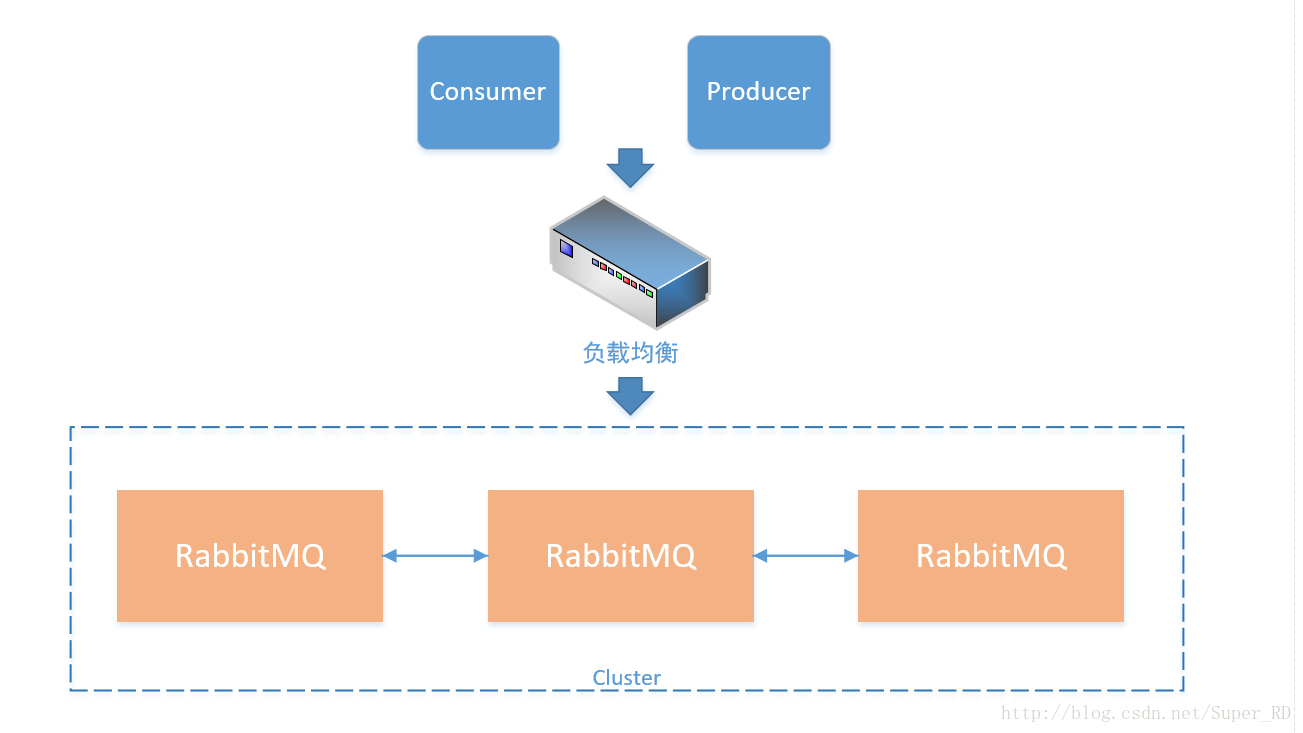
* 内存节点与磁盘节点

RabbitMQ的集群节点包括内存节点、磁盘节点。顾名思义内存节点就是将所有数据放在内存，磁盘节点将数据放在磁盘。不过，如前文所述，如果在投递消息时，打开了消息的持久化，那么即使是内存节点，数据还是会放在磁盘。原则上一个集群至少有一个磁盘节点。在实际使用中会发现所谓的磁盘节点是只用来存储集群的配置信息，也就是说如果集群中没有磁盘节点，当所有节点关机后集群的配置信息就会丢失。在进行性能测试时两个模式的节点订阅发布消息的性能没有太大差距。

* 多节点负载分发

RabbitMQ集群模式是没有中心节点的，并且在连接集群的时候实际上Consumer是连接其中某一台节点，连接方法和单主机模式一致。那就遇到一个尴尬的问题，怎么保证Consumer均匀的连接到多个节点。以下是我的一些思考，提供了两个思路。

1、通过负载均衡设备来实现流量分发。可以使用F5硬件负载均衡，如果没有F5的硬件负载均衡设备也可以使用想LVS等服务，当Consumer连接集群时实际是先经过负载均衡。



2、虽然负载均衡设备通常都很稳定，但这样一来RabbitMQ的集群就有了中心节点。我们在使用的时候是这样的，首先将集群中所以节点的IP放在一个数组中，app在连接RabbitMQ的时候会从数组中随机选择一个IP来连接，然后把连接的节点的IP缓存到服务器，如果连接超时则重新随机选择其他节点来连接。通过这种方式来实现app流量的分发。

现在对集群的基本概念都有了了解，下面我们一起来搭建一个普通模式的集群。

## RabbitMQ集群部署

我用5台服务器来搭建一个5个节点的集群，其中10.99.121.150为磁盘节点，其他服务器为内存节点。对服务器的命名如下：

10.99.121.150 RMQ\_D\_150   
10.99.121.151 RMQ\_M\_151   
10.99.121.152 RMQ\_M\_152   
10.99.121.153 RMQ\_M\_153   
10.99.121.154 RMQ\_M\_154

在前面第三章讲过单主机的RabbitMQ如何安装<http://blog.csdn.net/super_rd/article/details/70241007>   
先安装好5台单主机的RabbitMQ。

修改每一台主机的host文件：

**vi** /etc/hosts  
**10.99.121.150** RMQ\_D\_150  
**10.99.121.151** RMQ\_M\_151  
**10.99.121.152** RMQ\_M\_152  
**10.99.121.153** RMQ\_M\_153  
**10.99.121.154** RMQ\_M\_154

修改每一台主机的主机名：（我没有一一列出）

**vi** /etc/sysconfig/network  
**NETWORKING**=yes  
**NETWORKING\_IPV6**=no  
**HOSTNAME**= RMQ-M-154

打开每一台主机的相应端口：

**firewall-cmd** --permanent --add-port=25672/tcp  
**firewall-cmd** --permanent --add-port=15672/tcp  
**firewall-cmd** --permanent --add-port=5672/tcp  
**firewall-cmd** --permanent --add-port=4369/tcp  
**systemctl** restart firewalld.service

同步每个节点Cookie（在150执行）   
Rabbitmq的集群是依赖于erlang的集群来工作的，所以必须先构建起erlang的集群环境。Erlang的集群中各节点是通过一个magic cookie来实现的，这个cookie存放在 /var/lib/rabbitmq/.erlang.cookie 中，文件是400的权限。所以必须保证各节点cookie保持一致，否则节点之间就无法通信。

**scp** /root/.erlang.cookie root@10.99.121.151:/root/   
**scp** /root/.erlang.cookie root@10.99.121.152:/root/  
**scp** /root/.erlang.cookie root@10.99.121.153:/root/  
**scp** /root/.erlang.cookie root@10.99.121.154:/root/

所以节点重启

**rabbitmqctl** stop  
**rabbitmq-server** -detached

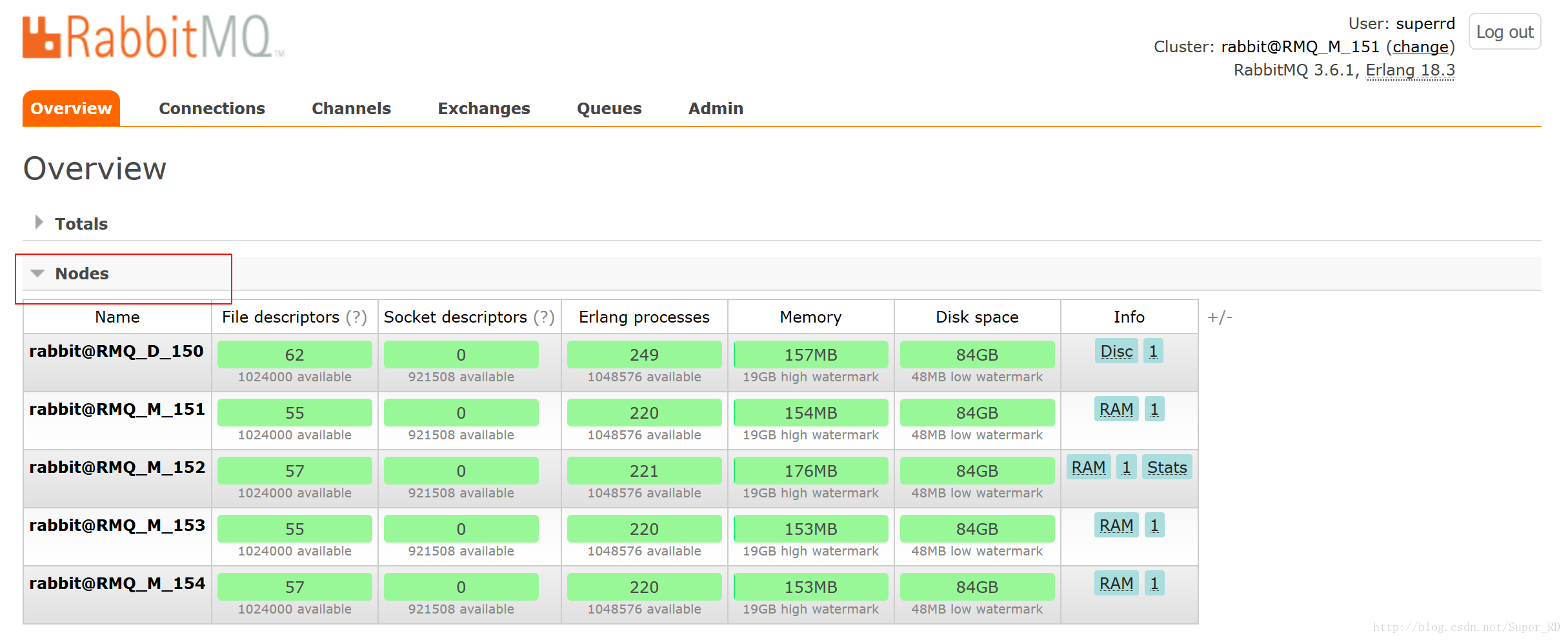
连接集群

**rabbitmqctl** stop\_app(注意硬盘节点先不要执行)  
**rabbitmqctl** join\_cluster --ram rabbit@RMQ\_D\_150（连接到任意一个已经加入集群的节点均可）  
**rabbitmqctl** start\_app  
**rabbitmqctl** cluster\_status //查看集群状态  
**//磁盘节点，join\_cluster** 命令去掉--ram参数即可。  
**//在RabbitMQ集群里，必须至少有一个磁盘节点存在（磁盘节点用来存储集群状态）。**

远程访问配置   
配置集群之后需要重新添加账号   
默认网页是不允许访问的，需要增加一个用户修改一下权限，代码如下：

**rabbitmqctl** add\_user superrd superrd //添加用户  
**rabbitmqctl** set\_permissions -p / superrd ".\*" ".\*" ".\*" //添加权限  
**rabbitmqctl** set\_user\_tags superrd administrator //修改用户角色

在浏览器输入任意一个节点的WEB管理插件[http://ip:15672](http://ip:15672/)，在Overviem视图找到Nodes选项卡可以看到所有的节点。



**删除节点：**   
修改host和主机名，同之前的步骤。   
在要脱离集群的节点执行：

**rabbitmqctl** stop\_app  
**rabbitmqctl** rest  
**rabbitmqctl** start\_app

或者在其他节点执行：（例如删除RMQ\_M\_154节点）

**rabbitmqctl** stop\_app  
**rabbitmqctl** forget\_cluster\_node rabbit@RMQ\_M\_154

**增加节点：**   
在已有节点复制cookies到新的节点

**scp** /root/.erlang.cookie root@10.99.121.155:/root/ //在已有节点执行。  
**//以下在要新增的节点执行。  
rabbitmqctl** stop\_app  
**rabbitmqctl** join\_cluster --ram rabbit@RMQ\_M\_154  
**rabbitmqctl** start\_app

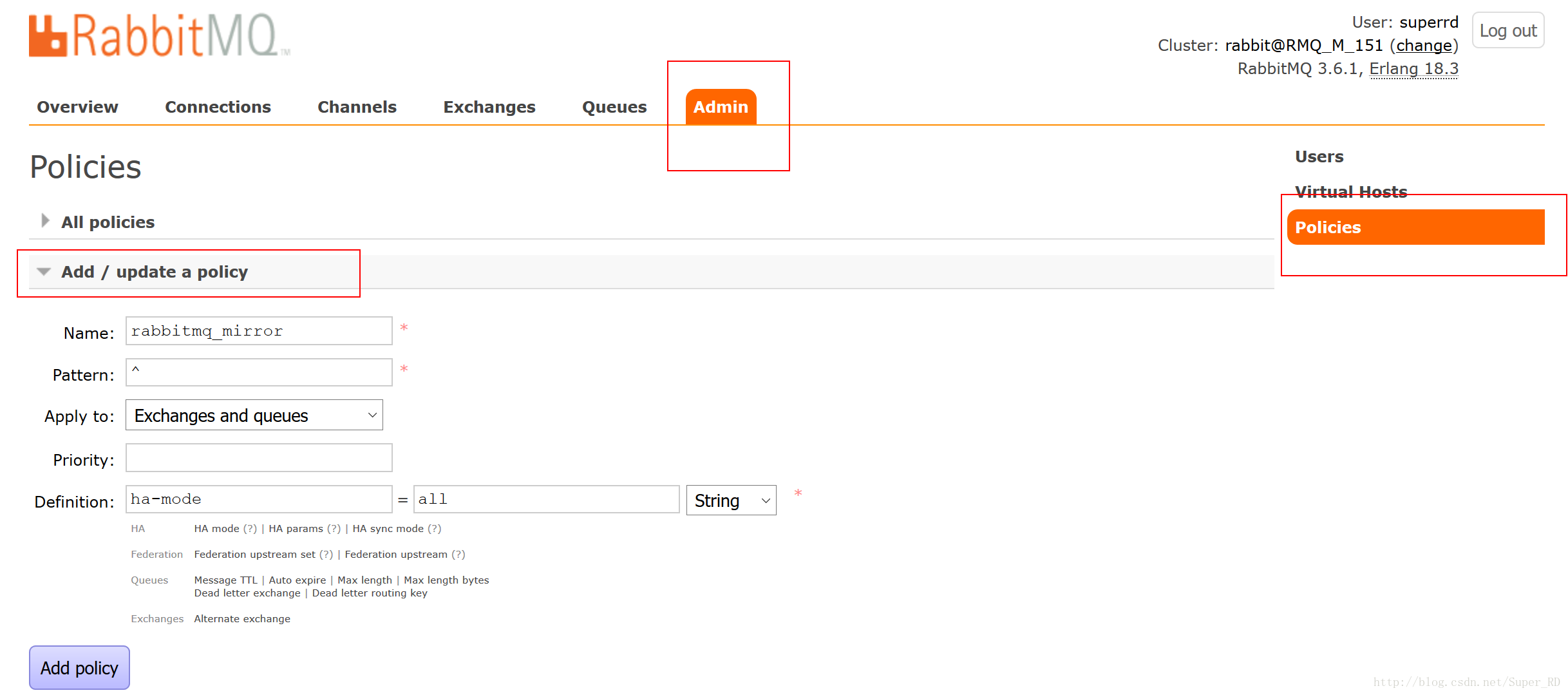
查看集群状态：

**rabbitmqctl** cluster\_status

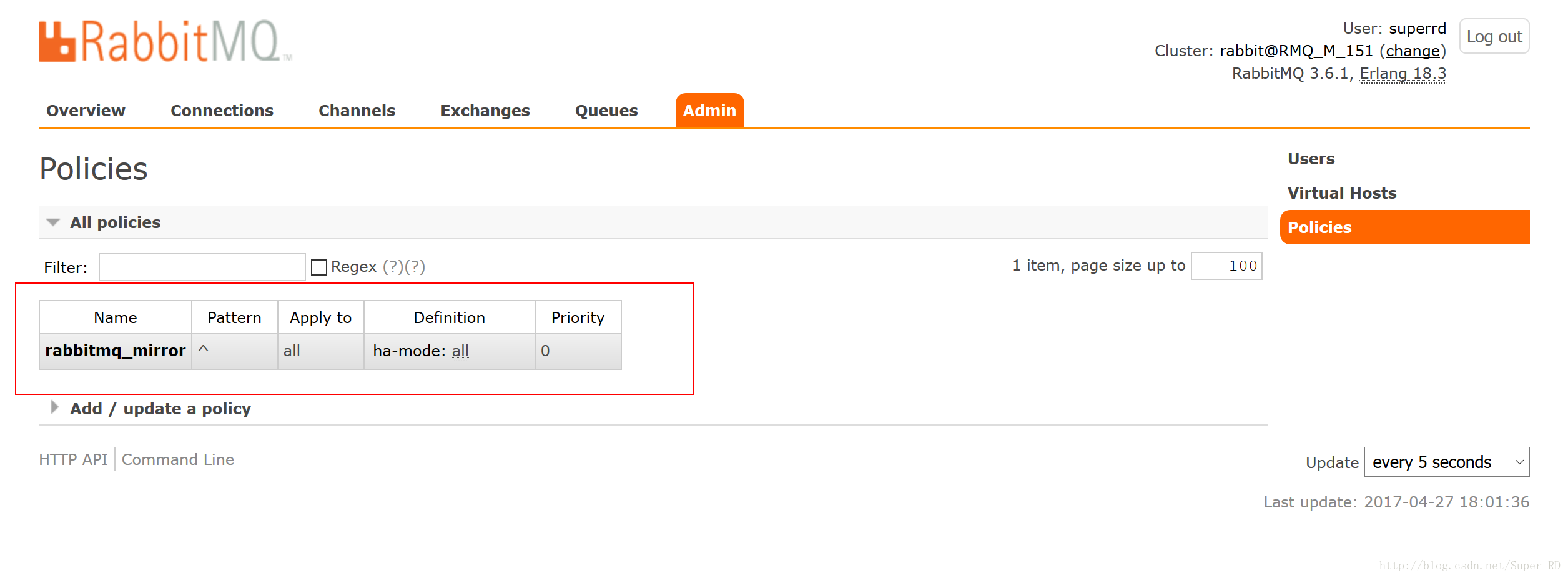
## RabbitMQ镜像集群配置

尽管我们部署好了普通模式的集群，但因为节点间只同步队列结构并不进行消息的同步，对于一些可靠性要求较高的场景需要对队列中的消息也同步到所以节点。   
使用Rabbit镜像功能，需要基于rabbitmq策略来实现，政策是用来控制和修改群集范围的某个vhost队列行为和Exchange行为，在cluster中任意节点启用策略，策略会自动同步到集群节点。

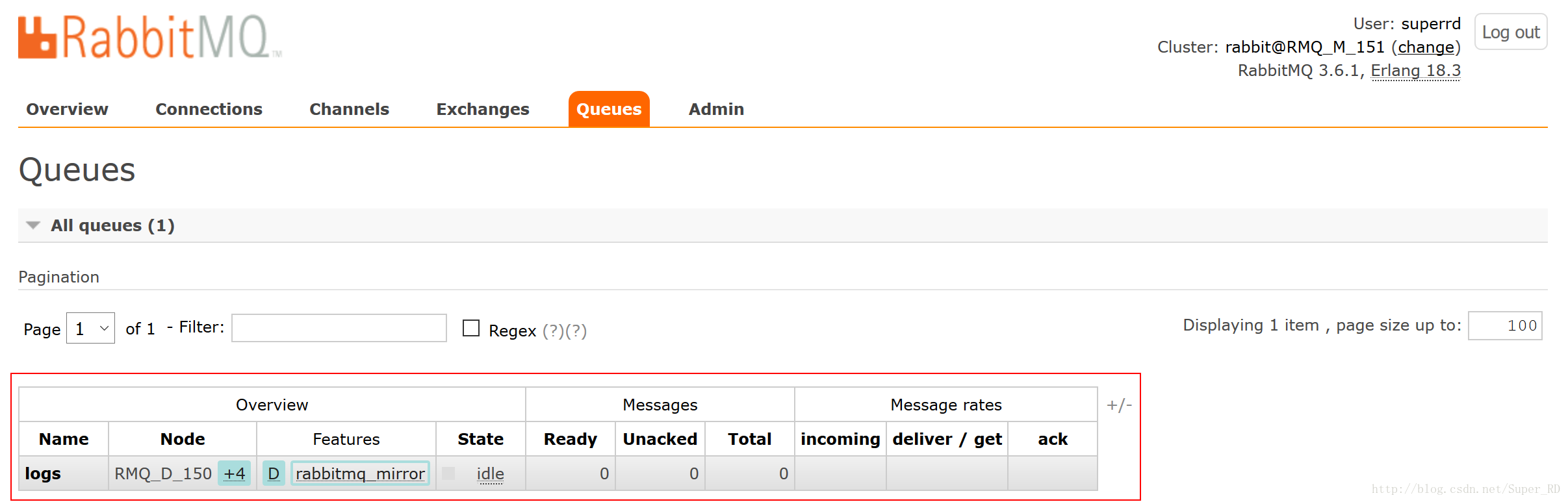
策略的修改可以通过命令也可以通过WEB，如果我是通过WEB来修改的，非常简单。   
Pattern：“^” 表示所有匹配所有队列名称。”^log” 是指同步”log”开头的队列名称。   
ha-mode：“all”代表同步到所以节点。



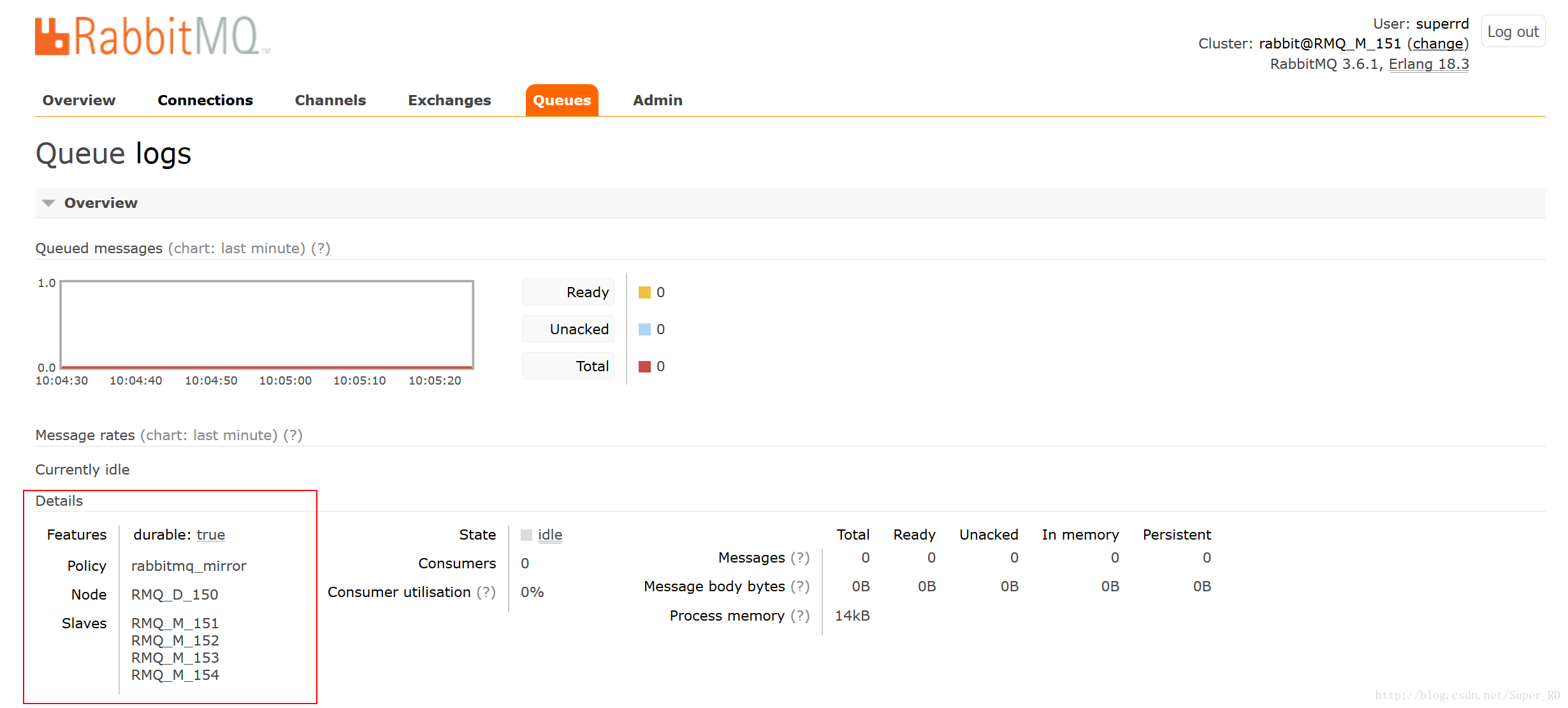
填写好后点击“Add policy”应用配置策略。可以看到已经新建的策略。



新建一个队列，然后查看队列列表。可以看到一个“+4”说明数据被保存了四份。



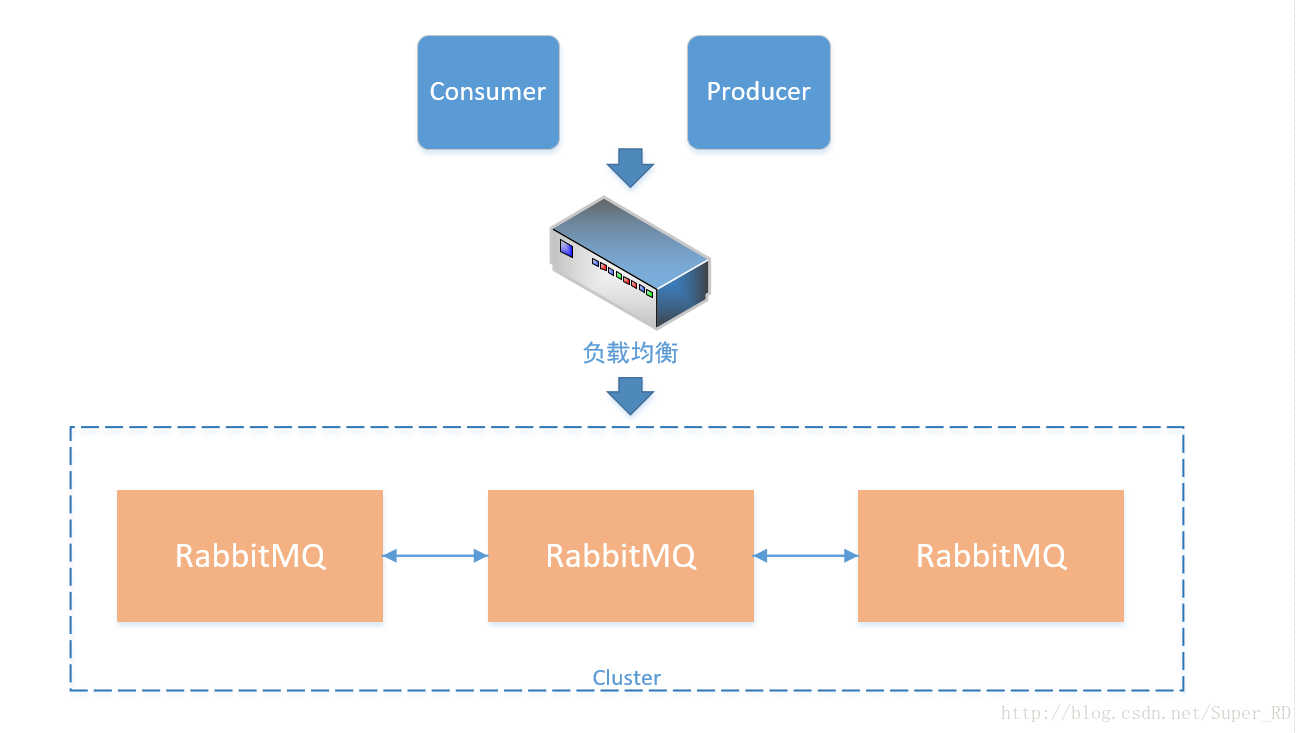
点击队列查看队列详情。可以看到队列是在150节点创建的，但是同步到了其余四个节点。



# （十一）RabbitMQ消息队列-如何实现高可用

在前面讲到了RabbitMQ高可用集群的搭建，但是我们知道只是集群的高可用并不能保证应用在使用消息队列时完全没有问题，例如如果应用连接的RabbitMQ集群突然宕机了，虽然这个集群时可以使用的，但是应用订阅的连接就断开了，如果有个机房外网出口带宽被挖掘机弄断了，那集群依然是不可用的。所以我们后面会介绍应用APP如何与连接集群来保证两者配合默契，以及如何实现跨机房的集群复制。

## 应用连接集群高可用



前面讲到应用服务器通过一个负载均衡服务将连接的流量分发到指定服务器，如果连接的节点宕机怎么办呢。应用服务器连接集群主要做两件事，订阅和发布，所以如果是发布消息每次都会重新初始化连接所以连接节点的切换对整个系统的可用性影响不大。如果是订阅消息就没有真么简单了。首先我们要做到如果连接出现问题应该是抛出异常而不是终止脚本，并且这时应该重新连接连接。   
好了不废话了，代码如下：

**$queueName** = 'superrd';  
**$exchangeName** = 'superrd';  
  
**while(True){  
  
try** {  
  
**$connection->connect()** or die("Cannot connect to the broker!**\n**");  
**$channel** = new AMQPChannel($connection);  
  
**$exchange** = new AMQPExchange($channel);  
**$exchange->setName($exchangeName);  
$exchange->setType(AMQP\_EX\_TYPE\_DIRECT);  
$exchange->setFlags(AMQP\_DURABLE);  
$exchange->declareExchange();  
  
$queue** = new AMQPQueue($channel);  
**$queue->setName($queueName);  
$queue->setFlags(AMQP\_DURABLE);  
$queue->declareQueue();  
$queue->bind($exchangeName,** $routeKey);  
  
**//阻塞模式接收消息  
  
echo** "Message:**\n**";  
**while(True){  
$queue->consume('processMessage');  
//自动ACK应答  
//$queue->consume('processMessage',** AMQP\_AUTOACK);  
**}  
  
}** catch (AMQPConnectionException $e) {  
**var\_dump($e);  
//** sleep(1);  
**}  
  
//$conn->disconnect();  
  
}  
  
/\*  
\*** 消费回调函数  
**\*** 处理消息  
**\*/  
function** processMessage($envelope, $q) {  
**$msg** = $envelope->getBody();  
**echo** $msg."**\n**"; //处理消息  
**$q->ack($envelope->getDeliveryTag());** //手动发送ACK应答  
**}**

所以通过以上的代码就可以保证服务器某节点宕机后订阅的连接自动重连切换。

## RabbitMQ集群异地复制

**基于warren的共享存储模式**

这种方式其实并不是跨地区的远程复制，并且需要共享存储，如果感兴趣的同学可以百度下。

**基于Shovel的远程复制**

如果直接基于WAN来组建异地的集群的话，集群间大量的数据通讯会产生高昂的费用，另外Erlang也不允许这么高延迟的通讯。   
Shovel是RabbitMQ自带插件（2.7.0后），自带插件的好处就是可以在RabbitMQ服务启动时自动启动Shovel和自定义复制关系。   
Shovel运行的原理其实非常简单。通过定义RabbitMQ上一个队列和另外一个RabbitMQ上的交换机之间的复制关系来实现远程复制。也就是说它会在主服务上建立一个队列来监听交换机，所以这是到交换机所以的消息会投递到该队列，并且在从服务中订阅这个队列，使队列中的消息复制到从服务的交换机中。RabbitMQ是一个比较全面的消息队列解决方案，我们公司并没有用到该功能，只是在这提下，感兴趣的同学可以搜下。

# （十二）RabbitMQ消息队列-性能测试

## 硬件配置

宿主机用的联想3850X6的服务器四颗E7-4850v3的处理器，DDR4内存，两块1.25TB的pcie固态。在宿主机上使用的事esxi5.5的虚拟化平台，在子系统中安装RabbitMQ和测试脚本，RabbitMQ配置如下：   
CPU：24核   
内存：24GB   
硬盘：24GB固态

确定哪些参数可能影响性能：进程数、是否持久化、是否ack确认以及交换机模式。其实交换机模式更多应该影响发布消息的性能，因为在消费消息时消息已经在队列所以影响不大。为了避免之前的一些配置影响测试我新建了一台RabbitMQ服务。

## 模式对性能的影响

首先我们看下不同交换机模式对新建交换机、新建队列、绑定、发布消息等性能的影响。在测试脚本中我是for循环建立10万交换机、循环建立10万队列、循环绑定10万、发送10万消息。有个疑问：消息发布的速度比我之前使用单独发布消息的速度快很多，单机模式之前从来没有超过5万过。



以上测试数据都是持久化的情况下的测试结果，可以看出**不同的模式对于新建交换机、新建队列、绑定等操作性能影响不大，但是在direct模式下明显消息发布的性能比其他模式强很多，并且消息发送到相同队列比发送到不同队列性能稍好。**

## 持久化对消息性能的影响

实际生产中大部分的操作一般是消息的订阅和发布，下面对消息订阅和发布的性能进行详细测试。发布100万消息。

在消息持久化模式下：   
发布：13888msg/s   
订阅：15384msg/s

在消息非持久化模式下：   
发布：18867msg/s   
订阅：26315msg/s

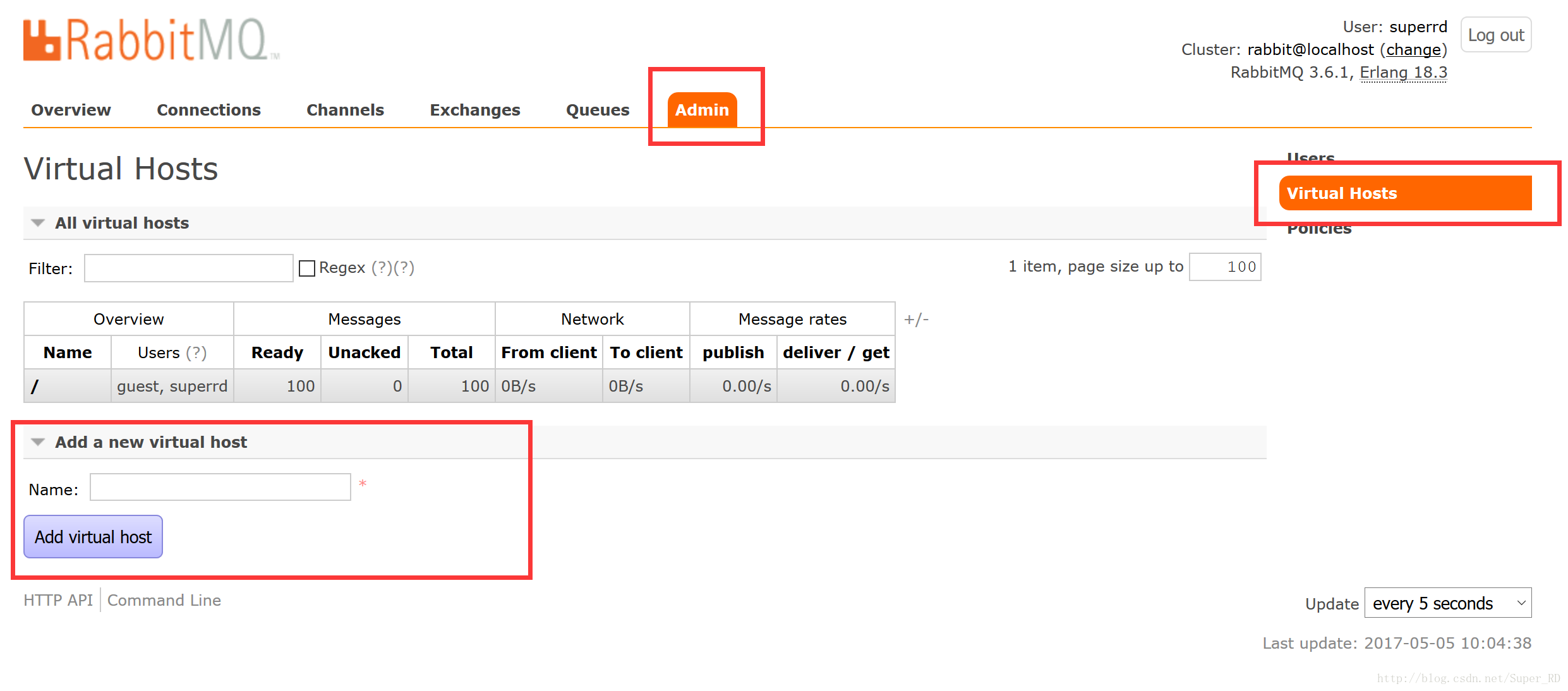
# （十三）RabbitMQ消息队列-VirtualHost与权限管理

## VirtualHost

像mysql有数据库的概念并且可以指定用户对库和表等操作的权限。那RabbitMQ呢？RabbitMQ也有类似的权限管理。在RabbitMQ中可以虚拟消息服务器VirtualHost，每个VirtualHost相当月一个相对独立的RabbitMQ服务器，每个VirtualHost之间是相互隔离的。exchange、queue、message不能互通。   
在RabbitMQ中无法通过AMQP创建VirtualHost，可以通过以下命令来创建。

**rabbitmqctl** add\_vhost [vhostname]

当然也可以通过WEB管理插件来创建。



如上图在创建完vhost后可以在All Virtual Host标签看到新建的VirtualHost。

## 用户权限管理

通常在权限管理中主要包含三步：

1. 新建用户
2. 配置权限
3. 配置角色

**新建用户**

**rabbitmqctl** add\_user superrd superrd

**配置权限**

**set\_permissions** [-p <vhostpath>] <user> <conf> <write> <read>

其中， 的位置分别用正则表达式来匹配特定的资源，如

**‘^(amq.gen.\*|amq.default)$’**

可以匹配server生成的和默认的exchange，’^$’不匹配任何资源

* exchange和queue的declare与delete分别需要exchange和queue上的配置权限
* exchange的bind与unbind需要exchange的读写权限
* queue的bind与unbind需要queue写权限exchange的读权限 发消息(publish)需exchange的写权限
* 获取或清除(get、consume、purge)消息需queue的读权限

示例：我们赋予superrd在“/”下面的全部资源的配置和读写权限。

**rabbitmqctl** set\_permissions -p / superrd ".\*" ".\*" ".\*"

注意”/”代表virtual host为“/”这个“/”和linux里的根目录是有区别的并不是virtual host为“/”可以访问所以的virtual host，把这个“/”理解成字符串就行。

**配置角色**

**rabbitmqctl** set\_user\_tags [user] [role]

RabbitMQ中的角色分为如下五类：none、management、policymaker、monitoring、administrator

官方解释如下：

**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.

* none   
  不能访问 management plugin
* management   
  用户可以通过AMQP做的任何事外加：   
  列出自己可以通过AMQP登入的virtual hosts   
  查看自己的virtual hosts中的queues, exchanges 和 bindings   
  查看和关闭自己的channels 和 connections   
  查看有关自己的virtual hosts的“全局”的统计信息，包含其他用户在这些virtual hosts中的活动。
* policymaker   
  management可以做的任何事外加：   
  查看、创建和删除自己的virtual hosts所属的policies和parameters
* monitoring   
  management可以做的任何事外加：   
  列出所有virtual hosts，包括他们不能登录的virtual hosts   
  查看其他用户的connections和channels   
  查看节点级别的数据如clustering和memory使用情况   
  查看真正的关于所有virtual hosts的全局的统计信息
* administrator   
  policymaker和monitoring可以做的任何事外加:   
  创建和删除virtual hosts   
  查看、创建和删除users   
  查看创建和删除permissions   
  关闭其他用户的connections

如下示例将superrd设置成administrator角色。

**rabbitmqctl** set\_user\_tags superrd administrator

# （十四）RabbitMQ消息队列-启用SSL安全通讯

如果RabbitMQ服务在内网中，只有内网的应用连接，我们认为这些连接都是安全的，但是个别情况我们需要让RabbitMQ对外提供服务。这种情况有两种解决方案：

* 在RabbitMQ外层在封装一层应用，应用对外提供服务，本质来说RabbitMQ还是只对内网提供服务。相对更安全，但灵活性差。
* RabbitMQ直接对外提供服务。这时除了服务本身的安全性还要考虑数据在互联网传输过程中是否可能被拦截破解。业界标准的解决方案就是SSL。

## 生成证书

首先确保已经安装好openssl，通常在安装WEB运行环境时都会自动安装。我就找了我之前的一个WEB服务器。   
检测方法：

**openssl** version

我在github上发现了一个可以自动生成证书的项目，通过这些shell脚本方便生成证书，项目github地址为：   
<https://github.com/Berico-Technologies/CMF-AMQP-Configuration>   
下载之后上传到安装openssl的服务器中   
切换到CMF-AMQP-Configuration/ssl 文件夹，并运行

**sh** setup\_ca.sh MyRabbitMQCA

名称定义为”MyRabbitMQCA”，这个名字可以自行指定，用于在证书中显示证书颁发机构名。

* 生成服务器证书

**sh** make\_server\_cert.sh rabbitmq-server rabbit

一个参数是服务器名，第二个参数是密码。

* 生成客户端证书

**sh** create\_client\_cert.sh rabbit-client rabbit

第一个参数是客户端名称，第二个参数是密码。

执行完以上步骤之后，会在ssl目录下生成：ca、server、client三个文件夹。

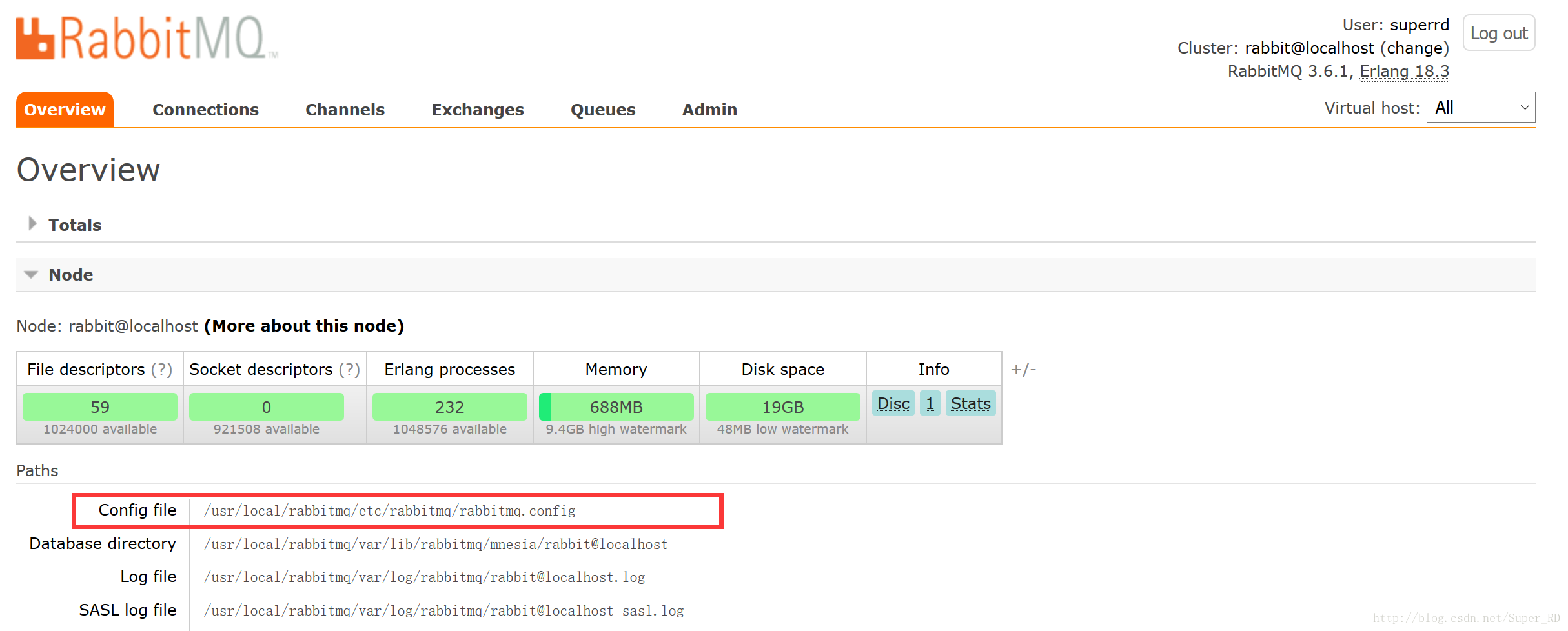
配置RabbitMQ Server用到的三个文件如下：   
配置RabbitMQ SSL只会用到以下3个文件：

**ca/cacert.pem  
server/rabbitmq-server.cert.pem  
server/rabbitmq-server.key.pem**

将这三个文件拷贝到RabbitMQ的/usr/local/rabbitmq/ssl/目录中。

## 配置RabbitMQ开启SSL

在WEB控制台中可以看到rabbitmq.config的地址，如果没有就新建这个文件



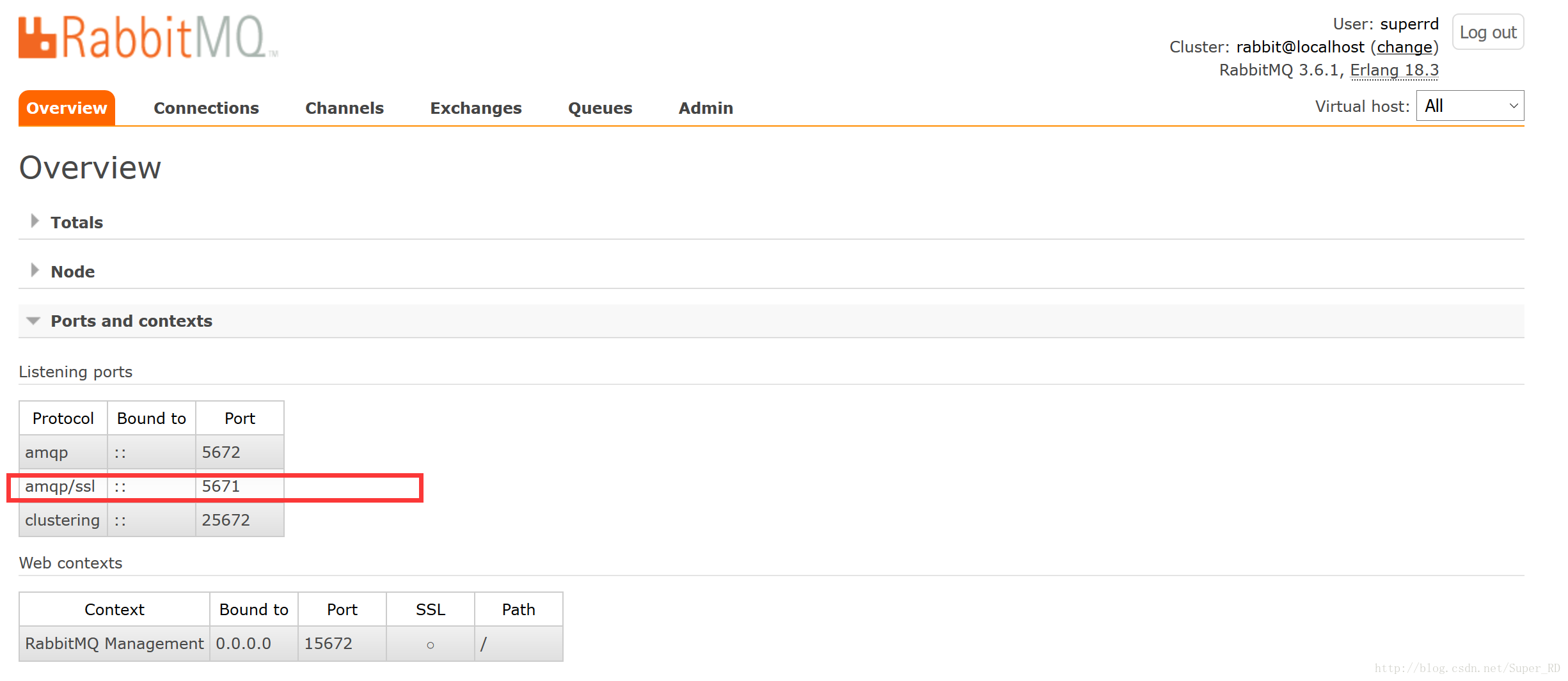
**vi** /usr/local/rabbitmq/etc/rabbitmq/rabbitmq.config  
  
**[  
{rabbit,** [  
**{tcp\_listeners,** [5672]},  
**{ssl\_listeners,** [5671]},  
**{ssl\_options,** [{cacertfile,"/usr/local/rabbitmq/ssl/cacert.pem"},  
**{certfile,"/usr/local/rabbitmq/ssl/rabbitmq-server.cert.pem"},  
{keyfile,"/usr/local/rabbitmq/ssl/rabbitmq-server.key.pem"},  
{verify,** verify\_peer},  
**{fail\_if\_no\_peer\_cert,** true}  
**]}  
]}  
].**

ssl\_listeners 指定SSL监听5671端口。   
fail\_if\_no\_peer\_cert 意思是是否强制验证证书。

* 重启RabbitMQ

**rabbitmqctl** stop  
**rabbitmq-server** -detached

* 查看是否开启



# Rabbit for Java

http://blog.csdn.net/chwshuang/article/details/50521708

http://blog.csdn.net/jacman/article/details/50261915