RabbitMQ快速指南

# 安装部署：

下载安装Erlang，地址：<http://www.erlang.org/download>/otp\_win\*\_\*exe；

下载安装RabbitMQ，地址：<http://www.rabbitmq.com/releases/rabbitmq-server>/v\*/rabbitmq-server-\*exe；

下载rabbit-client.jar，MQ连接依赖，开发时使用，地址：[http://www.rabbitmq.com/releases/rabbitmq-java-client/v\*/rabbitmq-java-client-bin-\*.zip](http://www.rabbitmq.com/releases/rabbitmq-java-client/v*/rabbitmq-java-client-bin-*.zip)；

安装完成后，RabbitMQ会自动安装自启动的Windows服务，可运行services.msc查看是否存在并开启（如图Figure 1），也可以手动启动RabbitMQ安装目录sbin下的rabbitmq-server.bat来启动MQ服务（如图Figure 2）。默认安装的Rabbit MQ 监听端口是5672；

注：

1. 以上工具均下载官网对应的最新正式版即可，也可使用Inteligence-ATM\9.运行环境下已经下载好的环境软件；
2. 上述部署过程为了方便测试采用的是Windows平台解决方案，具体的生产环境将会切换到Linux平台，其环境配置过程可参考[集群配置](#_集群配置：)；

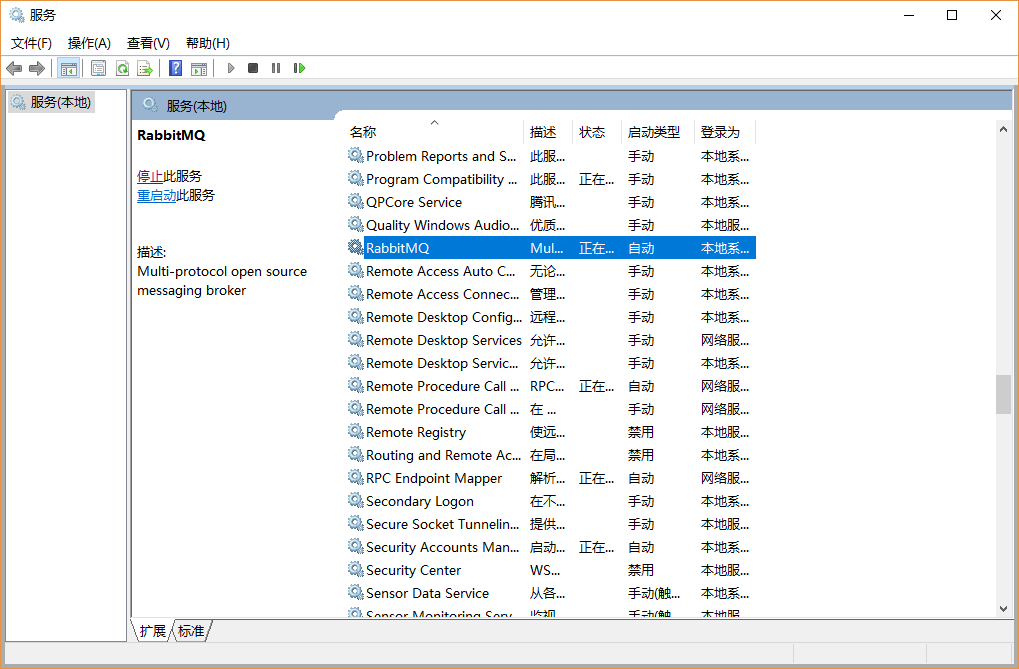


Figure 1

# 监控管理：

使用Rabbit MQ 管理插件，可以更好的可视化方式查看Rabbit MQ 服务器实例的状态和管理用户与MQ消息，可以在RabbitMQ安装目录sbin下使用如下命令激活：

rabbitmq-plugins.bat enable rabbitmq\_management

更改需要重启服务生效，可以执行net stop RabbitMQ && net start RabbitMQ；

Web管理端基本能满足日常的使用要求，比如MQ信息查看和用户管理，使用浏览器打开http://localhost:15672 访问Rabbit Mq的管理控制台：（默认用户为guest/guest）

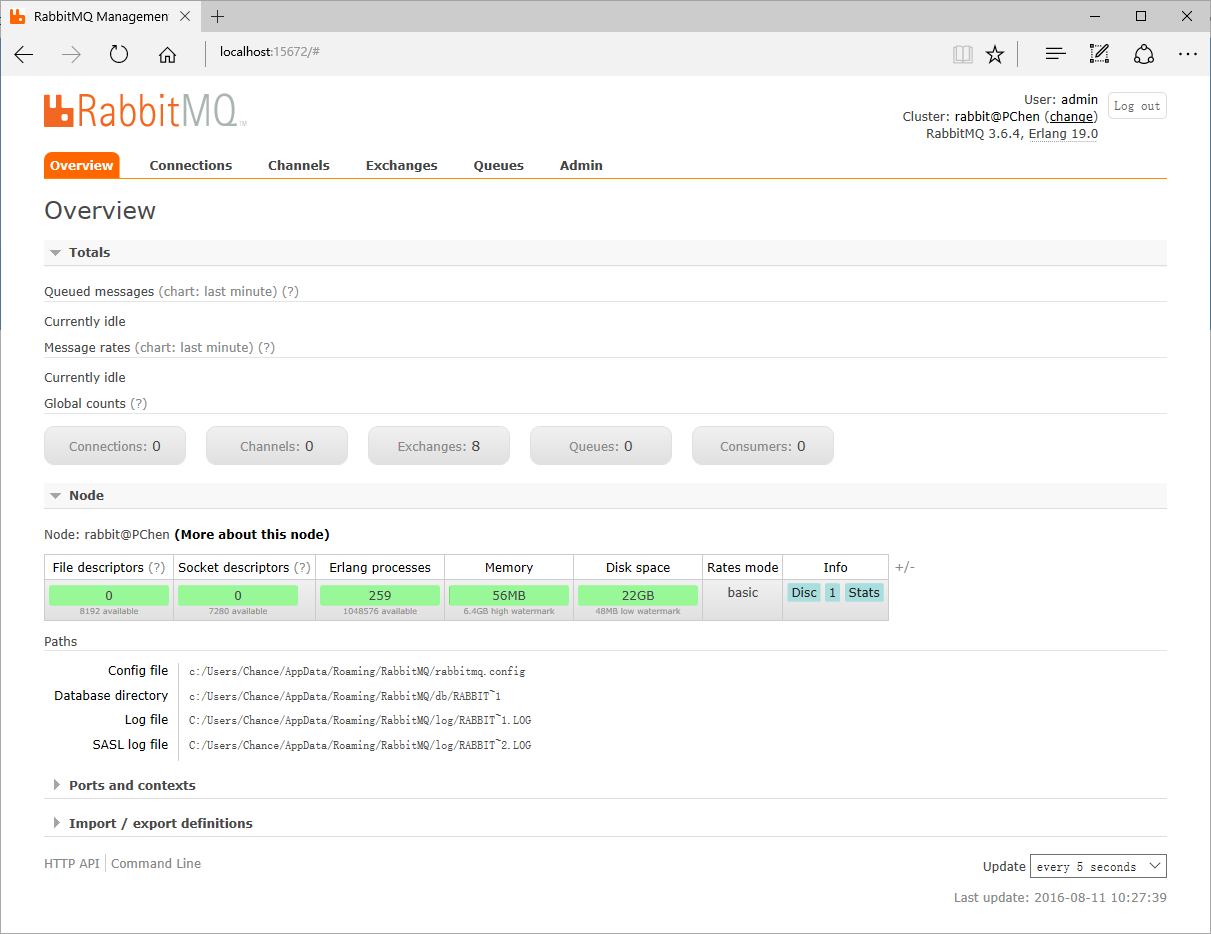


Figure 3

管理台说明：

* Overview可以查看MQ数量、MQ集群基本信息以及配置文件和日志文件路径等；
* Connections、Channels、Exchanges以及Queues可以查看连接和消息详情；
* Admin界面可以进行用户的管理，添加删除或者修改权限，右侧的Policies标签点击之后还能进行策略的配置，详情可查看官方策略配置说明；

# 基本概念：

AMQP(高级消息队列协议) 是一个异步消息传递所使用的应用层协议规范，作为线路层协议，而不是API（例如JMS），AMQP 客户端能够无视消息的来源任意发送和接受信息。AMQP的原始用途只是为金融界提供一个可以彼此协作的消息协议，而现在的目标则是为通用消息队列架构提供通用构建工具。因此，面向消息的中间件 （MOM）系统，例如发布/订阅队列，没有作为基本元素实现。反而通过发送简化的AMQ实体，用户被赋予了构建例如这些实体的能力。这些实体也是规范的一 部分，形成了在线路层协议顶端的一个层级：AMQP模型。这个模型统一了消息模式，诸如之前提到的发布/订阅，队列，事务以及流数据，并且添加了额外的特性，例如更易于扩展，基于内容的路由。

AMQP当中有四个概念非常重要：virtual host/虚拟主机，exchange/交换机，queue/队列，binding/绑定。

一个虚拟主机持有一组交换机、队列和绑定。为什么需要多个虚拟主机呢？因为RabbitMQ当中，用户只能在虚拟主机的粒度进行权限控制。因此，如果需要禁止A组访问B组的交换机/队列/绑定，必须为A和B分别创建一个虚拟主机。每一个RabbitMQ服务器都有一个默认的虚拟主机。

队列（Queues）是你的消息（messages）的终点，可以理解成装消息的容器。消息就一直在里面，直到有客户端（也就是消费者，Consumer）连接到这个队列并且将其取走为止。不过，也可以将一个队列配置成这样的：一旦消息进入这个队列，此消息就被删除。

队列是由消费者（Consumer）通过程序建立的，不是通过配置文件或者命令行工具。这没什么问题，如果一个消费者试图创建一个已经存在的队列，RabbitMQ会直接忽略这个请求。因此我们可以将消息队列的配置写在应用程序的代码里面。

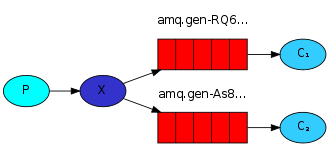
而要把一个消息放进队列前，需要有一个交换机（Exchange）。交换机（Exchange）可以理解成具有路由表的路由程序。每个消息都有一个称为路由键（routing key）的属性，就是一个简单的字符串。交换机当中有一系列的绑定（binding），即路由规则（routes）。（例如，指明具有路由键 “X” 的消息要到名为timbuku的队列当中去。）

消费者程序（Consumer）要负责创建你的交换机。交换机可以存在多个，每个交换机在自己独立的进程当中执行，因此增加多个交换机就是增加多个进程，可以充分利用服务器上的CPU核以便达到更高的效率。例如，在一个8核的服务器上，可以创建5个交换机来用5个核，另外3个核留下来做消息处理。类似的，在RabbitMQ的集群当中，你可以用类似的思路来扩展交换机一边获取更高的吞吐量。

交换机如何判断要把消息送到哪个队列？你需要路由规则，即绑定（binding）。一个绑定就是一个类似这样的规则：将交换机“desert（沙漠）”当中具有路由键“阿里巴巴”的消息送到队列“hideout（山洞）”里面去。换句话说，一个绑定就是一个基于路由键将交换机和队列连接起来的路由规则。例如，具有路由键“audit”的消息需要被送到两个队列，“log-forever”和“alert-the-big-dude”。要做到这个，就需要创建两个绑定，每个都连接一个交换机和一个队列，两者都是由“audit”路由键触发。在这种情况下，交换机会复制一份消息并且把它们分别发送到两个队列当中。交换机不过就是一个由绑定构成的路由表。

交换机有多种类型。他们都是做路由的，但是它们接受不同类型的绑定。为什么不创建一种交换机来处理所有类型的路由规则呢？因为每种规则用来做匹配分子的CPU开销是不同的。例如，一个“topic”类型的交换机试图将消息的路由键与类似“dogs.\*”的模式进行匹配。匹配这种末端的通配符比直接将路由键与“dogs”比较（“direct”类型的交换机）要消耗更多的CPU。如果你不需要“topic”类型的交换机带来的灵活性，你可以通过使用“direct”类型的交换机获取更高的处理效率。在[接口调用](#_接口调用：)部分将详细说明几种基本类型交换机的使用。

MQ的基本结构可简单描述如下，其中X即代表交换机。



MQ消息流程说明：

* Producer：连接验证MQ Server，创建一个频道，声明EXCHANGE\_NAME或者QUEUE\_NAME，（指定RoutingKey），然后使用MQ的basicPublish接口发送消息。需要说明的是，basicPublish第一个参数为交换机名，第二个为队列名或者路由规则，当使用的交换机类型不为“”时，将会自动忽略QUEUE\_NAME而将第二个参数作为路由规则使用；
* Consumer：连接验证MQ Server，创建一个频道，声明EXCHANGE\_NAME或者QUEUE\_NAME，（指定RoutingKey），创建消费者并与QUEUE\_NAME绑定，然后使用MQ的nextDelivery阻塞方法获取消息。需要说明的是，如果声明的是EXCHANGE\_NAME，声明队列时就需要使用Channel的queueDeclare().getQueue()方法来创建一个系统自动维护的随机队列，然后将其与交换机以及指定的RoutingKey绑定；

# 接口调用：

注：以下代码均可在项目LearnRabbit中找到。

## ****1基本消息：****

(P) -> [|||] -> (C)

消息发送：

|  |
| --- |
| ConnectionFactory factory = **new** ConnectionFactory(); *//设置MabbitMQ所在主机ip或者主机名* factory.setHost(**"10.34.10.245"**);  factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); *//创建一个连接* Connection connection = factory.newConnection(); *//创建一个频道* Channel channel = connection.createChannel(); *//指定一个队列* channel.queueDeclare(***QUEUE\_NAME***, true, **false**, **false**, **null**); //参数2声明队列持久化 *//发送的消息* String message = **"hello world!"**; *//往队列中发出一条消息* channel.basicPublish(**""**, ***QUEUE\_NAME***, **MessageProperties.PERSISTENT\_TEXT\_PLAIN**, message.getBytes()); //参数3声明消息持久化 *//关闭频道和连接* channel.close(); connection.close(); |

消息接收：

|  |
| --- |
| *//打开连接和创建频道，与发送端一样* ConnectionFactory factory = **new** ConnectionFactory(); factory.setHost(**"10.34.10.245"**);  factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); Connection connection = factory.newConnection(); Channel channel = connection.createChannel(); *//声明队列，主要为了防止消息接收者先运行此程序，队列还不存在时创建队列。* channel.queueDeclare(***QUEUE\_NAME***, **true**, **false**, **false**, **null**);  *//创建队列消费者* QueueingConsumer consumer = **new** QueueingConsumer(channel); *//指定消费队列* channel.basicConsume(***QUEUE\_NAME***, **true**, consumer); **while** (**true**) {  *//nextDelivery是一个阻塞方法（内部实现其实是阻塞队列的take方法）* QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = **new** String(delivery.getBody());  System.***out***.println(**" [x] Received '"** + message + **"'"**); } |

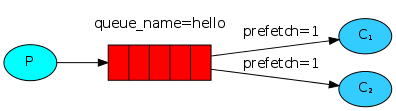
## ****2工作队列：****

工作队列的主要任务是：避免立刻执行资源密集型任务，然后必须等待其完成。相反地，我们进行任务调度：我们把任务封装为消息发送给队列。工作进行在后台运行并不断的从队列中取出任务然后执行。当你运行了多个工作进程时，任务队列中的任务将会被工作进程共享执行。

默认的，RabbitMQ会一个一个的发送信息给下一个消费者(consumer)，而不考虑每个任务的时长等等，且是一次性分配，并非一个一个分配。平均的每个消费者将会获得相等数量的消息。这样分发消息的方式叫做round-robin。

我们不希望丢失任何任务（信息）。当某个工作者（接收者）被杀死时，我们希望将任务传递给另一个工作者。为了保证消息永远不会丢失，RabbitMQ支持消息应答（message acknowledgments）。消费者发送应答给RabbitMQ，告诉它信息已经被接收和处理，然后RabbitMQ可以自由的进行信息删除。

如果消费者被杀死而没有发送应答，RabbitMQ会认为该信息没有被完全的处理，然后将会重新转发给别的消费者。通过这种方式，你可以确认信息不会被丢失，即使消者偶尔被杀死。这种机制并没有超时时间这么一说，RabbitMQ只有在消费者连接断开是重新转发此信息。如果消费者处理一个信息需要耗费特别特别长的时间是允许的。消息应答默认是打开的。



发送消息：

|  |
| --- |
| *// 创建连接和频道* ConnectionFactory factory = **new** ConnectionFactory(); factory.setHost(**"10.34.10.245"**);  factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); Connection connection = factory.newConnection(); Channel channel = connection.createChannel(); *// 声明队列* **boolean** durable = **true**;*// 1、设置队列持久化* channel.queueDeclare(***QUEUE\_NAME***, durable, **false**, **false**, **null**); *// 发送10条消息，依次在消息后面附加1-10个点* **for** (**int** i = 10; i > 0; i--) {  String dots = **""**;  **for** (**int** j = 0; j <= i; j++)  {  dots += **"."**;  }  String message = **"helloworld"** + dots + dots.length();  *// MessageProperties 2、设置消息持久化* channel.basicPublish(**""**, ***QUEUE\_NAME***,  MessageProperties.***PERSISTENT\_TEXT\_PLAIN***, message.getBytes());  System.***out***.println(**" [x] Sent '"** + message + **"'"**); } channel.close(); connection.close(); |

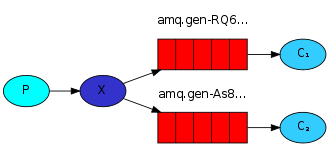
接收消息：

|  |
| --- |
| *// 区分不同工作进程的输出* **int** hashCode = Work3.**class**.hashCode(); *// 创建连接和频道* ConnectionFactory factory = **new** ConnectionFactory(); factory.setHost(**"10.34.10.245"**);  factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); Connection connection = factory.newConnection(); Channel channel = connection.createChannel(); *// 声明队列* **boolean** durable = **true**; channel.queueDeclare(***QUEUE\_NAME***, durable, **false**, **false**, **null**); System.***out***.println(hashCode  + **" [\*] Waiting for messages. To exit press CTRL+C"**); *//设置最大服务转发消息数量* **int** prefetchCount = 1; channel.basicQos(prefetchCount); QueueingConsumer consumer = **new** QueueingConsumer(channel); *// 指定消费队列* **boolean** ack = **false**; *// 打开应答机制* channel.basicConsume(***QUEUE\_NAME***, ack, consumer); **while** (**true**) {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = **new** String(delivery.getBody());   System.***out***.println(hashCode + **" [x] Received '"** + message + **"'"**);  *doWork*(message);  System.***out***.println(hashCode + **" [x] Done"**);channel.basicAck(delivery.getEnvelope().getDeliveryTag(), **false**); } |

## ****3发布订阅：****

发布/订阅有点类似观察者模式，使用这种模式可以把一个消息发给多个消费者。

RabbitMQ消息模型的核心理念是生产者永远不会直接发送任何消息给队列，一般的情况生产者甚至不知道消息应该发送到哪些队列。相反的，生产者只能发送消息给转发器（Exchange）。转发器是非常简单的，一边接收从生产者发来的消息，另一边把消息推送到队列中。转发器必须清楚的知道消息如何处理它收到的每一条消息。是否应该追加到一个指定的队列？是否应该追加到多个队列？或者是否应该丢弃？这些规则通过转发器的类型进行定义。（Direct，Topic，Headers，Fanout）



发送消息：

|  |
| --- |
| *// 创建连接和频道* ConnectionFactory factory = **new** ConnectionFactory(); factory.setHost(**"10.34.10.245"**);  factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); Connection connection = factory.newConnection(); Channel channel = connection.createChannel(); *// 声明转发器和类型* channel.exchangeDeclare(***EXCHANGE\_NAME***, **"fanout"** ); String message = **new** Date().toLocaleString()+**" : log something"**; *// 往转发器上发送消息* channel.basicPublish(***EXCHANGE\_NAME***, **""**, **null**, message.getBytes()); channel.close(); connection.close(); |

接收消息1：

|  |
| --- |
| *// 创建连接和频道* ConnectionFactory factory = **new** ConnectionFactory(); factory.setHost(**"10.34.10.245"**);  factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); Connection connection = factory.newConnection(); Channel channel = connection.createChannel(); channel.exchangeDeclare(***EXCHANGE\_NAME***, **"fanout"**); *// 创建一个非持久的、唯一的且自动删除的队列* String queueName = channel.queueDeclare().getQueue(); *// 为转发器指定队列，设置binding* channel.queueBind(queueName, ***EXCHANGE\_NAME***, **""**); QueueingConsumer consumer = **new** QueueingConsumer(channel); *// 指定接收者，第二个参数为自动应答，无需手动应答* channel.basicConsume(queueName, **true**, consumer); **while** (**true**) {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = **new** String(delivery.getBody());  System.***out***.println(**" [x] Received '"** + message + **"'"**); } |

接收消息2：

|  |
| --- |
| *// 创建连接和频道* ConnectionFactory factory = **new** ConnectionFactory(); factory.setHost(**"10.34.10.245"**);  factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); Connection connection = factory.newConnection(); Channel channel = connection.createChannel(); channel.exchangeDeclare(***EXCHANGE\_NAME***, **"fanout"**); *// 创建一个非持久的、唯一的且自动删除的队列* String queueName = channel.queueDeclare().getQueue(); *// 为转发器指定队列，设置binding* channel.queueBind(queueName, ***EXCHANGE\_NAME***, **""**); QueueingConsumer consumer = **new** QueueingConsumer(channel); *// 指定接收者，第二个参数为自动应答，无需手动应答* channel.basicConsume(queueName, **true**, consumer); **while** (**true**) {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = **new** String(delivery.getBody());  *print2File*(message); //*写文件操作* } |

## ****4路由选择：****

绑定表示转发器与队列之间的关系。我们也可以简单的认为：队列对该转发器上的消息感兴趣。绑定可以附带一个额外的参数routingKey。下面展示如何使用绑定键（binding key）来创建一个绑定：channel.queueBind(queueName, EXCHANGE\_NAME, "black");

绑定键的意义依赖于转发器的类型。对于fanout类型，忽略此参数。

上一节基于发布订阅的日志系统广播所有的消息给所有的消费者。我们希望可以对其扩展，来允许根据日志的严重性进行过滤日志。这节将会使用direct类型的转发器进行替代。direct类型的转发器背后的路由转发算法很简单：消息会被推送至绑定键（binding key）和消息发布附带的选择键（routing key）完全匹配的队列。



发送消息：

|  |
| --- |
| *// 创建连接和频道* ConnectionFactory factory = **new** ConnectionFactory(); factory.setHost(**"10.34.10.245"**);  factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); Connection connection = factory.newConnection(); Channel channel = connection.createChannel(); *// 声明转发器的类型* channel.exchangeDeclare(***EXCHANGE\_NAME***, **"direct"**); *//发送6条消息* **for** (**int** i = 0; i < 6; i++) {  String severity = *getSeverity*();  String message = severity + **"\_log :"** + UUID.*randomUUID*().toString();  *// 发布消息至转发器，指定routingkey* channel.basicPublish(***EXCHANGE\_NAME***, severity, **null**, message  .getBytes());  System.***out***.println(**" [x] Sent '"** + message + **"'"**); } channel.close(); connection.close(); |

接收消息：

|  |
| --- |
| *// 创建连接和频道* ConnectionFactory factory = **new** ConnectionFactory(); factory.setHost(**"10.34.10.245"**);  factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); Connection connection = factory.newConnection(); Channel channel = connection.createChannel(); *// 声明direct类型转发器* channel.exchangeDeclare(***EXCHANGE\_NAME***, **"direct"**); String queueName = channel.queueDeclare().getQueue(); String severity = *getSeverity*(); *// 指定binding\_key* channel.queueBind(queueName, ***EXCHANGE\_NAME***, severity); System.***out***.println(**" [\*] Waiting for "**+severity+**" logs. To exit press CTRL+C"**); QueueingConsumer consumer = **new** QueueingConsumer(channel); channel.basicConsume(queueName, **true**, consumer); **while** (**true**) {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = **new** String(delivery.getBody());  System.***out***.println(**" [x] Received '"** + message + **"'"**); } |

## ****5主题转发：****

虽然使用direct类型改良了我们的系统，但是仍然存在一些局限性：它不能够基于多重条件进行路由选择。在我们的日志系统中，我们有可能希望不仅根据日志的级别而且想根据日志的来源进行订阅。为了在我们的系统中实现上述的需求，我们需要学习稍微复杂的主题类型的转发器（topic exchange）。

发往主题类型的转发器的消息不能随意的设置选择键（routing\_key），必须是由点隔开的一系列的标识符组成。标识符可以是任何东西，但是一般都与消息的某些特性相关。

绑定键和选择键的形式一样。主题类型的转发器背后的逻辑和直接类型的转发器很类似：一个附带特殊的选择键将会被转发到绑定键与之匹配的队列中。需要注意的是：关于绑定键有两种特殊的情况。（\*可以匹配一个标识符；#可以匹配0个或多个标识符。）



发送消息：

|  |
| --- |
| *// 创建连接和频道* ConnectionFactory factory = **new** ConnectionFactory(); factory.setHost(**"10.34.10.245"**); factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); Connection connection = factory.newConnection(); Channel channel = connection.createChannel(); channel.exchangeDeclare(***EXCHANGE\_NAME***, **"topic"**); String[] routing\_keys = **new** String[]{**"kernal.info"**, **"cron.warning"**,  **"auth.info"**, **"kernel.critical"**}; **for** (String routing\_key : routing\_keys) {  String msg = UUID.*randomUUID*().toString();  channel.basicPublish(***EXCHANGE\_NAME***, routing\_key, **null**, msg  .getBytes());  System.***out***.println(**" [x] Sent routingKey = "** + routing\_key + **" ,msg = "** + msg + **"."**); } channel.close(); connection.close(); |

接收消息1：

|  |
| --- |
| *// 创建连接和频道* ConnectionFactory factory = **new** ConnectionFactory(); factory.setHost(**"10.34.10.245"**); factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); Connection connection = factory.newConnection(); Channel channel = connection.createChannel(); *// 声明转发器* channel.exchangeDeclare(***EXCHANGE\_NAME***, **"topic"**); *// 随机生成一个队列* String queueName = channel.queueDeclare().getQueue(); *// 接收所有与kernel相关的消息* channel.queueBind(queueName, ***EXCHANGE\_NAME***, **"\*.critical"**); System.***out***.println(**" [\*] Waiting for critical messages. To exit press CTRL+C"**); QueueingConsumer consumer = **new** QueueingConsumer(channel); channel.basicConsume(queueName, **true**, consumer); **while** (**true**) {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = **new** String(delivery.getBody());  String routingKey = delivery.getEnvelope().getRoutingKey();   System.***out***.println(**" [x] Received routingKey = "** + routingKey  + **",msg = "** + message + **"."**); } |

接收消息2：

|  |
| --- |
| *// 创建连接和频道* ConnectionFactory factory = **new** ConnectionFactory(); factory.setHost(**"10.34.10.245"**); factory.setUsername(**"admin"**); factory.setPassword(**"admin"**); Connection connection = factory.newConnection(); Channel channel = connection.createChannel(); *// 声明转发器* channel.exchangeDeclare(***EXCHANGE\_NAME***, **"topic"**); *// 随机生成一个队列* String queueName = channel.queueDeclare().getQueue(); *//接收所有与kernel相关的消息* channel.queueBind(queueName, ***EXCHANGE\_NAME***, **"kernel.\*"**); System.***out***.println(**" [\*] Waiting for messages about kernel. To exit press CTRL+C"**); QueueingConsumer consumer = **new** QueueingConsumer(channel); channel.basicConsume(queueName, **true**, consumer); **while** (**true**) {  QueueingConsumer.Delivery delivery = consumer.nextDelivery();  String message = **new** String(delivery.getBody());  String routingKey = delivery.getEnvelope().getRoutingKey();  System.***out***.println(**" [x] Received routingKey = "** + routingKey  + **",msg = "** + message + **"."**); } |

# 扩展功能：

## ****1队列长度：****

消息长度可以用对控制超过设定长度或者数量的消息将会被抛弃。消息长度的设置有两种方式：第一种是在客户端加入队列参数设置，第二种是使用服务端策略。

* 客户端：

在队列声明的参数中使用“x-max-length”参数设置队列消息最大数量，使用“x-max-length-bytes”设置队列消息最大字节量，示例如下：

|  |
| --- |
| Map<String, Object> settings = **new** HashMap<String, Object>(); settings.put(**"x-max-length"**, 10); *//设置消息队列最大长度为10* settings.put(**"x-max-length-bytes"**, 1000000); *//设置消息队列最大不超过1000000字节* channel.queueDeclare(*QUEUE\_NAME*, **false**, **false**, **false**, settings); |

* 服务端：

使用“rabbitmqctl”脚本命令来控制策略，队列消息最大长度和队列消息字节上限分别使用“max-length”和“max-length-bytes”，示例如下：

|  |
| --- |
| //Windows版本：  rabbitmqctl set\_policy Ten "^**QUEUE\_NAME**$" "{""max-length "":10}" --apply-to queues  rabbitmqctl set\_policy Ten "^**QUEUE\_NAME**$" "{""max-length-bytes"":1000000}" --apply-to queues  //Linux版本  rabbitmqctl set\_policy Ten "^**QUEUE\_NAME** $" '{"max-length ":10}' --apply-to queues  rabbitmqctl set\_policy Ten "^**QUEUE\_NAME** $" '{"max-length-bytes":1000000}' --apply-to queues |

## ****2消息超时：****

RabbitMQ使用TTL（Time To Live）来控制超时，对Message和Queue都可以设置，超过设定好的TTL之后的Message或者Queue将会被抛弃。

* 特定队列的消息TTL：

客户端使用“x-message-ttl”参数，服务端使用“message-ttl”策略，示例如下：

|  |
| --- |
| //客户端  Map<String, Object> settings = **new** HashMap<String, Object>(); settings.put(**"x-message-ttl"**, 60000); *//在QUEUE\_NAME超过60秒的消息被丢弃* channel.queueDeclare(*QUEUE\_NAME*, **false**, **false**, **false**, settings); |
| //服务端windows（".\*"表示设置到所有队列）  rabbitmqctl set\_policy TTL ".\*" "{""message-ttl"":60000}" --apply-to queues  //服务端Linux  rabbitmqctl set\_policy TTL ".\*" '{"message-ttl":60000}' --apply-to queues |

* 单条消息TTL：

单条消息超时在客户端发布消息时设置，示例如下

|  |
| --- |
| **byte**[] messageBodyBytes = **"Hello, world!"**.getBytes(); *// AMQP.BasicProperties properties = new AMQP.BasicProperties(); // properties.setExpiration("60000");//setExpiration无法使用？* AMQP.BasicProperties properties = **new** AMQP.BasicProperties(**null**, **null**, **null**, **null**, **null**,  **null**, **null**, **"60000"**, **null**, **null**, **null**, **null**, **null**, **null**); channel.basicPublish(**""**, *QUEUE\_NAME*, properties, messageBodyBytes); |

* 队列TTL：

客户端加入“x-expires”参数，服务端使用“expires”策略，示例如下：

|  |
| --- |
| //客户端  Map<String, Object> settings = **new** HashMap<String, Object>(); settings.put(**"x-expires"**, 1800000);*//抛弃超过30分钟的没有使用的队列* channel.queueDeclare(*QUEUE\_NAME*, **false**, **false**, **false**, settings); |
| //服务端Windows （".\*"表示设置到所有队列）  rabbitmqctl set\_policy expiry ".\*" "{""expires"":1800000}" --apply-to queues  //服务端Linux  rabbitmqctl set\_policy expiry ".\*" '{"expires":1800000}' --apply-to queues |

## ****3消息失效：****

队列的消息可能会变成失效信息，在以下情况会重新通过交换机进行转发：

* 消息被拒绝（requeue=false）；
* 消息TTL已超时；
* 队列长度限制溢出；

使用DLXs(Dead letter exchanges) 设定这些'dead-lettered'也有两种方式：客户端加入“x-dead-letter-exchange”或者“x-dead-letter-routing-key”参数，服务端使用“dead-letter-exchange”策略。示例如下：

|  |
| --- |
| //客户端  channel.exchangeDeclare(***EXCHANGE\_NAME***, **"direct"**);*//声明交换机为“direct”类型* Map<String, Object> settings = **new** HashMap<String, Object>(); settings.put(**"x-dead-letter-exchange"**, ***EXCHANGE\_NAME***);*//给交换机添加DLXs* settings.put(**"x-dead-letter-routing-key"**, **"some-routing-key"**);*//给特定路由添加DLXs* channel.queueDeclare(***QUEUE\_NAME***, **false**, **false**, **false**, settings); |
| //服务端Windows（".\*"表示设置到所有队列）  rabbitmqctl set\_policy DLX ".\*" "{""dead-letter-exchange"":""my-dlx""}" --apply-to queues  //服务端Linux  rabbitmqctl set\_policy DLX ".\*" '{"dead-letter-exchange":"my-dlx"}' --apply-to queues |

## ****4监控报警：****

通过加入消息监控，可以提前知道内存、磁盘和流程控制导致的MQ异常。

* 内存超出警告：

当执行了“rabbitmqctl set\_vm\_memory\_high\_watermark fraction”时，如果RabbitMQ服务器发现超出被安装机器的某个内存使用率，将会发出内存超出警告。默认情况下MQ服务器内存使用率达到总量的40%之后，就会发出警告并且阻断所有的MQ连接，警告清除后，服务器才会重新工作。

使用MQ配置文件设置示例如下：

设置MQ内存使用率超过40%报警：

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

设置MQ内存使用量超过1G报警：

[{rabbit, [{vm\_memory\_high\_watermark, {absolute, "1024MiB"}}]}].

在MQ服务器启动后，内存告警设置进会追加到“RABBITMQ\_NODENAME.log”文件中：

=INFO REPORT==== 29-Oct-2009::15:43:27 ===

Memory limit set to 2048MB.

如果是32位系统（建议不用），MQ启动后将会提示可寻址限制：

=WARNING REPORT==== 19-Dec-2013::11:27:13 ===

Only 2048MB of 12037MB memory usable due to limited address space.

Crashes due to memory exhaustion are possible - see

http://www.rabbitmq.com/memory.html#address-space

设置MQ虚拟内存使用率超过75%报警：

[{rabbit, [{vm\_memory\_high\_watermark\_paging\_ratio, 0.75},

{vm\_memory\_high\_watermark, 0.4}]}].

也可以通过服务端对内存告警进行设置：

rabbitmqctl set\_vm\_memory\_high\_watermark fraction

rabbitmqctl set\_vm\_memory\_high\_watermark absolute memory\_limit

* 磁盘超出警告：

如果剩余磁盘空间低于某个限制，RabbitMQ将会禁止生产者产生消息。

设置MQ最低剩余磁盘空间为1G：

[{rabbit, [{disk\_free\_limit, "1GB"}]}].

设置MQ最低剩余磁盘空间和主机的RAM相同：

[{rabbit, [{disk\_free\_limit, {mem\_relative, 1.0}}]}].

也可以通过服务端策略对磁盘告警进行设置：

rabbitmqctl set\_disk\_free\_limit disk\_limit

rabbitmqctl set\_disk\_free\_limit mem\_relative fraction

# 集群配置：

## ****1集群准备(必须)：****

由于分配的Ubuntu云主机基于同一镜像创建，hostname都一样，而集群中MQ服务器要求id唯一，所以如果不在安装RabbitMQ之前修改，可能在集群过程中会出现各种意想不到的问题。

1.修改ubuntu的hostname文件：

sudo vim /etc/hostname

这里我们分别将我们的MQ服务器主机名改为ubuntu-mq0、ubuntu-mq1、ubuntu-mq2……

更改完成之后重启服务器。

2.修改hosts文件：

Sudo vim /etc/hosts

在每个MQ服务器hosts文件都加上如下字段并保存（IP按照实际的，域名分别与hostname对应起来）：

10.34.10.121 ubuntu-mq1

10.34.10.159 ubuntu-mq2

10.34.10.169 ubuntu-mq0

完成之后刷新DNS：

sudo /etc/init.d/networking restart

若rabbitmq-server第一次启动后hostname不能被解析或者发生了更改则会导致启动失败，重装前执行如下操作：

sudo rm -rf /var/lib/rabbitmq/mnesia（相关信息会记录在此数据库）

## ****2Linux环境安装：****

安装之前请再确认是否完成集群准备操作！

1.安装Erlang依赖：

sudo apt-get install erlang

2.配置更新软件源：（默认仓库提供3.2.4版有bug）

sudo cp /etc/apt/sources.list /etc/apt/sources.list.bak（备份源）

sudo vim /etc/apt/sources.list

在源最后添加“deb http://www.rabbitmq.com/debian/ testing main”保存

wget -O- https://www.rabbitmq.com/rabbitmq-release-signing-key.asc | sudo apt-key add –（添加RabbitMQ公钥）

sudo apt-get update（更新软件源）

3.安装RabbitMQ服务器：

sudo apt-get install rabbitmq-server

4.启动管理组件：

sudo rabbitmq-plugins enable rabbitmq\_management

重启MQ管理组件生效

5.添加管理用户：

sudo rabbitmqctl add\_user admin admin

sudo rabbitmqctl set\_user\_tags admin administrator

sudo rabbitmqctl set\_permissions -p VHostPath admin ConfP WriteP ReadP

注：

* RabbitMQ尽量使用最新版本，并且确保要集群的各个MQ服务器的软件版本一致，经测试发现默认库提供的3.2.4确实存在集群节点无法发现的bug；
* 本次使用的Linux具体版本为Ubuntu-Server14.04（测试发现在12.04下基本MQ功能正常，但是管理组件无法使用，官方也有低版本的解决方案但不推荐），其它平台Linux请使用通用或者自平台软件版本；
* 相关依赖和软件包也可以通过离线包自行编译安装（~/运行环境/Linux），如果出现依赖问题可在安装完成后使用sudo apt-get -f install解决；
* 账号guest具有所有的操作权限，并且又是默认账号，出于安全因素的考虑，guest用户只能通过localhost登陆使用，所以要进行Web管理和消息通信必须执行用户操作；

## ****3常用MQ命令：****

Linux版的MQ指令稍微和Windows有些差别，所有的MQ操作都需要root权限。

开启服务：(三选一)

sudo invoke-rc.d rabbitmq-server start

sudo service rabbitmq-server start

sudo rabbitmq-server start

停止服务：

sudo invoke-rc.d rabbitmq-server stop

sudo service rabbitmq-server start

sudo rabbitmq-server stop

重启服务：

sudo invoke-rc.d rabbitmq-server restart

sudo service rabbitmq-server restart

sudo rabbitmq-server restart

查看状态：

sudo service rabbitmq-server status

sudo invoke-rc.d rabbitmq-server status

查看队列状态：

sudo rabbitmqctl list\_queues

新增一个用户：

sudo rabbitmqctl add\_user Username Password

删除一个用户

sudo rabbitmqctl delete\_user Username

修改用户的密码

sudo rabbitmqctl change\_password Username Newpassword

查看当前用户列表

sudo rabbitmqctl list\_users

设置用户角色的命令为：

sudo rabbitmqctl set\_user\_tags User Tag（User为用户名， Tag为角色名(administrator，monitoring，policymaker，management)，可参考[用户说明](#_2RabbitMQ用户：)。）

设置用户权限

sudo rabbitmqctl set\_permissions -p VHostPath User ConfP WriteP ReadP

查看(指定hostpath)所有用户的权限信息

sudo rabbitmqctl list\_permissions [-p VHostPath]

查看指定用户的权限信息

sudo rabbitmqctl list\_user\_permissions User

清除用户的权限信息

sudo rabbitmqctl clear\_permissions [-p VHostPath] User

停止节点服务：

sudo rabbitmqctl stop\_app

开启节点服务：

sudo rabbitmqctl start\_app

查看集群状态：

sudo rabbitmqctl cluster\_status

加入集群：

sudo rabbitmqctl join\_cluster --ram rabbit@ubuntu-mq1

更改节点类型：

sudo rabbitmqctl change\_cluster\_node\_type disc

## ****4集群环境搭建：****

集群配置说明：

* RabbitMQ集群通信是通过Erlang消息传递，所以所有运行在集群中的节点服务器必须有相同的Erlang cookie，“.erlang.cookie”文件实际是集群节点进行通信的验证密钥，而且他们的RabbitMQ和Erlang的版本也都必须相同；
* 每个MQ服务器上的“.erlang.cookie”文件有两个，分别在/var/lib/rabbitmq/.erlang.cookie和$HOME/.erlang.cookie目录下，自己测试时需要保证两个目录的cookie文件的值、哈希值、权限以及用户组都相同集群节点才能相互发现并连接；
* 集群后的所有节点构成一个逻辑的MQ 服务器，它们共享所有的用户、虚拟主机、消息队列、交换机、路由绑定以及运行时参数和数据。（默认情况下消息队列只存在于某个节点，但是对所有节点可见并且可以访问）；

集群配置步骤如下：

以下操作将ubuntu-mq1作为主节点，其它两个作为辅节点

1.备份“.erlang.cookie”：

（所有MQ节点）

sudo cp .erlang.cookie .erlang.cookie.bak

sudo cp /var/lib/rabbitmq/.erlang.cookie /var/lib/rabbitmq/.erlang.cookie.bak

2.检查home目录下的cookie权限、用户组以及md5值：

（121上执行）

ls -l .erlang.cookie（-r-------- 1 root root \*\*\*）

md5sum .erlang.cookie（输出32位字符串）

3.复制主节点（ubuntu-mq1）“.erlang.cookie”到辅节点：（注意对于远程服务器只能操作tmp目录）

（121上执行）

sudo scp .erlang.cookie [ubuntu@10.34.10.159:/tmp](mailto:ubuntu@10.34.10.159:/tmp)

sudo scp .erlang.cookie [ubuntu@10.34.10.169:/tmp](mailto:ubuntu@10.34.10.169:/tmp)

4.将cookie移动到home目录：

（159,169上执行）

sudo mv /tmp/.erlang.cookie ./

5.检查cookie的权限、用户组和md5值是否和第一步的相同：

（159,169上执行）

ls –l .erlang.cookie

md5sum .erlang.cookie

确保和第二步的操作输出一致（权限、用户组、哈希值）

sudo chmod 400 .erlang.cookie(更改cookie权限为只读)

sudo chown root:root .erlang.cookie（更改为root用户组的root）

同理，对/var/lib/rabbitmq目录下的.erlang.cookie执行相似操作（root权限）：

（121上执行）

sudo ls –l /var/lib/rabbitmq/.erlang.cookie

（这里的用户组和上面的不同，-r-------- 1 rabbitmq rabbitmq \*\*\*）

sudo md5sum /var/lib/rabbitmq/.erlang.cookie

sudo scp /var/lib/rabbitmq/.erlang.cookie [ubuntu@10.34.10.159:/tmp](mailto:ubuntu@10.34.10.159:/tmp)

sudo scp /var/lib/rabbitmq/.erlang.cookie [ubuntu@10.34.10.169:/tmp](mailto:ubuntu@10.34.10.169:/tmp)

（159,169上执行）

sudo mv /tmp/.erlang.cookie /var/lib/rabbitmq/.erlang.cookie

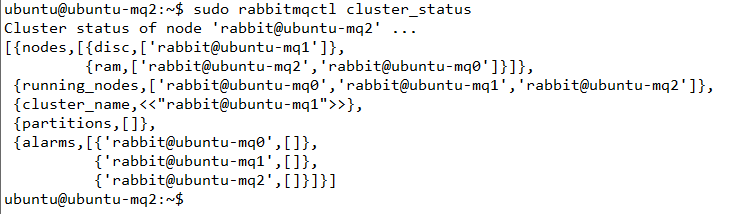
sudo ls -l /var/lib/rabbitmq/.erlang.cookie

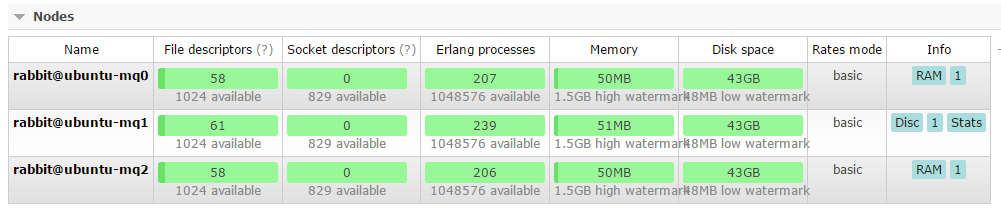
sudo md5sum /var/lib/rabbitmq/.erlang.cookie

sudo chmod 400 /var/lib/rabbitmq/.erlang.cookie

sudo chown rabbitmq:rabbitmq /var/lib/rabbitmq/.erlang.cookie(#)

可以通过命令查看集群状态如下，也可以登录Web管理端查看：





## ****5镜像模式配置：****

第四步配置RabbitMQ为默认集群模式，但并不保证队列的高可用性，尽管交换机、绑定这些可以复制到集群里的任何一个节点，但是队列内容不会复制，虽然该模式解决一部分节点压力，但队列节点宕机直接导致该队列无法使用，只能等待重启，所以要想在队列节点宕机或故障也能正常使用，就要复制队列内容到集群里的每个节点，需要创建镜像队列。

1.在10.34.10.248上安装负载均衡器haproxy：

sudo apt-get update

sudo apt-get install haproxy

2.配置haproxy：

sudo vim /etc/haproxy/haproxy.cfg

在后面添加以下字段：

listen rabbitmq\_cluster 0.0.0.0:5672

mode tcp

balance roundrobin

server ubuntu-mq2 10.34.10.159:5672 check inter 2000 rise 2 fall 3

server ubuntu-mq0 10.34.10.169:5672 check inter 2000 rise 2 fall 3

#server ubuntu-mq1 10.34.10.121:5672 check inter 2000 rise 2 fall 3

3.在集群的任意节点中使用策略开启RabbitMQ镜像功能

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

之后所有队列会被设置为镜像队列，即队列会被复制到各个节点，各个节点状态保持一直。

4.客户端连接负载均衡服务器进行消息收发时，创建队列需要指定“x-ha-policy”，这样队列将会自动在集群节点间复制。

负载均衡器会监听5672端口，轮询我们的两个内存节点10.34.10.159、10.34.10.169的5672端口，10.34.10.121为磁盘节点，只做备份不提供给生产者、消费者使用，当然如果我们服务器资源充足情况也可以配置多个磁盘节点，这样磁盘节点除了故障也不会影响，除非同时出故障。

## ****6MQ集群架构图：****



# 其它说明：

## ****1MQEmulator：****

MQEmulator（~/测试）是对RabbitMQ的部分API封装与UI展示，可以用来进行MQ收发报文模拟操作，基本说明如下：



MQEmulator界面主要分为 **接收区**（禁用编辑）、 **控制区**和 **发送区**（可编辑），其中控制区功能如下：

* **设置**：设置RabbitMQ服务器的IP和监听的MQ队列，设置完成会自动进行重连；
* **浏览**：选择加载要发送的报文文件（不区分后缀的json文本文件）；
* **连接**：连接RabbitMQ服务器（默认连接10.34.10.245 Windows服务器，可换10.34.10.121/10.34.10.159/10.34.10.169）；
* **发送**：发送ＭＱ消息（发送的消息为下半窗口内填写的json格式内容）；
* **断开**：断开与ＭＱ连接；
* **保存**：保存收到的ＭＱ消息（保存上半窗口接收到的消息记录到文本文件，不区分后缀）；
* **历史**：查看保存的ＭＱ消息文件（查看 **保存**操作生成的历史消息记录）；

## ****2RabbitMQ用户：****

* 用户角色

RabbitMQ用户角色可分为五类，超级管理员, 监控者, 策略制定者, 普通管理者以及其他。

(1) 超级管理员(administrator)

可登陆管理控制台(启用management plugin的情况下)，可查看所有的信息，并且可以对用户，策略(policy)进行操作。

(2) 监控者(monitoring)

可登陆管理控制台(启用management plugin的情况下)，同时可以查看rabbitmq节点的相关信息(进程数，内存使用情况，磁盘使用情况等)

(3) 策略制定者(policymaker)

可登陆管理控制台(启用management plugin的情况下), 同时可以对policy进行管理。但无法查看节点的相关信息(上图红框标识的部分)。

与administrator的对比，administrator能看到这些内容

(4) 普通管理者(management)

仅可登陆管理控制台(启用management plugin的情况下)，无法看到节点信息，也无法对策略进行管理。

(5) 其他

无法登陆管理控制台，通常就是普通的生产者和消费者。

* 用户权限

用户权限指的是用户对exchange，queue的操作权限，包括配置权限，读写权限。配置权限会影响到exchange，queue的声明和删除。读写权限影响到从queue里取消息，向exchange发送消息以及queue和exchange的绑定(bind)操作。

例如：将queue绑定到某exchange上，需要具有queue的可写权限，以及exchange的可读权限；向exchange发送消息需要具有exchange的可写权限；从queue里取数据需要具有queue的可读权限。详细请参考官方文档中"How permissions work"部分。