GNU/Linux-Rabbitmq

Rabbitmq 配置与应用

GNU/Linux-Rabbitmq

RabbitMQ

MQ 全称为 Message Queue, 消息队列（ MQ ）是一种应用程序对应用程序的通信方法。应用程序通过读写出入队列的消息（针对应用程序的数据）来通信，而无需专用连接来链接它们。

消息传递指的是程序之间通过在消息中发送数据进行通信，而不是通过直接调用彼此来通信，直接调用通常是用于诸如远程过程调用的技术。

GNU/Linux-Rabbitmq

Rabbitmq

排队指的是应用程序通过 队列来通信。队列的使用除去了接收和发送应用程序同时执行的要求。

其中较为成熟的 MQ 产品有 IBM WEBSPHERE MQ 。

GNU/Linux-Rabbitmq

Rabbitmq 简介

AMQP，即 Advanced Message Queuing Protocol，高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。

GNU/Linux-Rabbitmq

Rabbitmq 简介

AMQP 的主要特征是面向消息、队列、路由（包括点对点和发布 / 订阅）、可靠性、安全。

RabbitMQ 是一个开源的 AMQP 实现，服务器端用 Erlang 语言编写，支持多种客户端，如：Python 、 Ruby 、 .NET 、 Java 、 JMS 、 C 、 PHP、 ActionScript 、 XMPP 、 STOMP 等，支持 AJAX 。

GNU/Linux-Rabbitmq

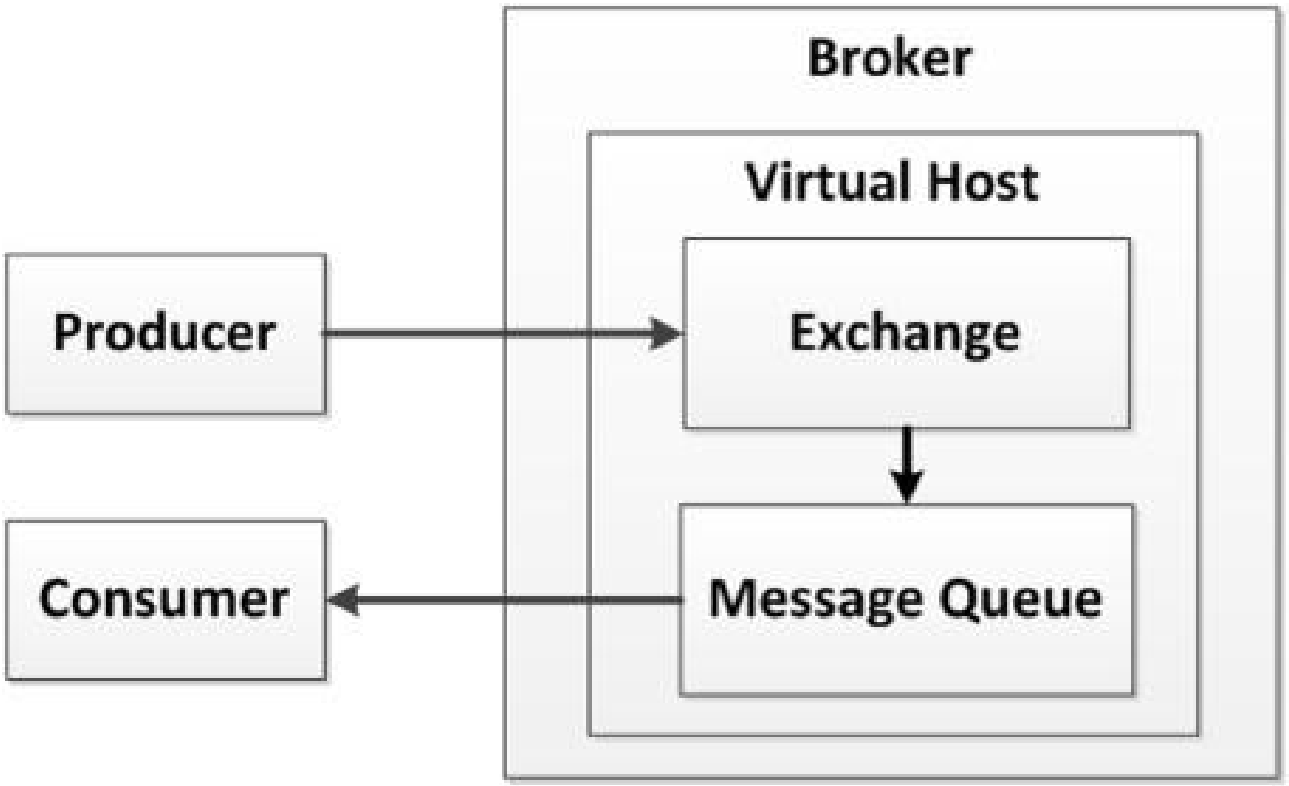
RabbtMQ简介

RabbitMQ用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。

下面将重点介绍RabbitMQ中的一些基础概念，了解了这些概念，是使用好RabbitMQ的基础。

GNU/Linux-Rabbitmq

AMQP中的核心概念图



GNU/Linux-Rabbitmq

Broker：简单来说就是消息队列服务器实体。

Exchange：消息交换机，它指定消息按什么规则，路由到哪个队列。

Queue：消息队列载体，用来保存消息直到发送给消费者。

Binding：绑定器，它的作用就是把exchange和queue按照路由规则绑定起来。

Routing Key：路由关键字，exchange根据这个关键字进行消息投递。

GNU/Linux-Rabbitmq

vhost：虚拟主机，一个broker里可以开设多个vhost，用作不同用户的权限分离。

producer：消息生产者，就是投递消息的程序。

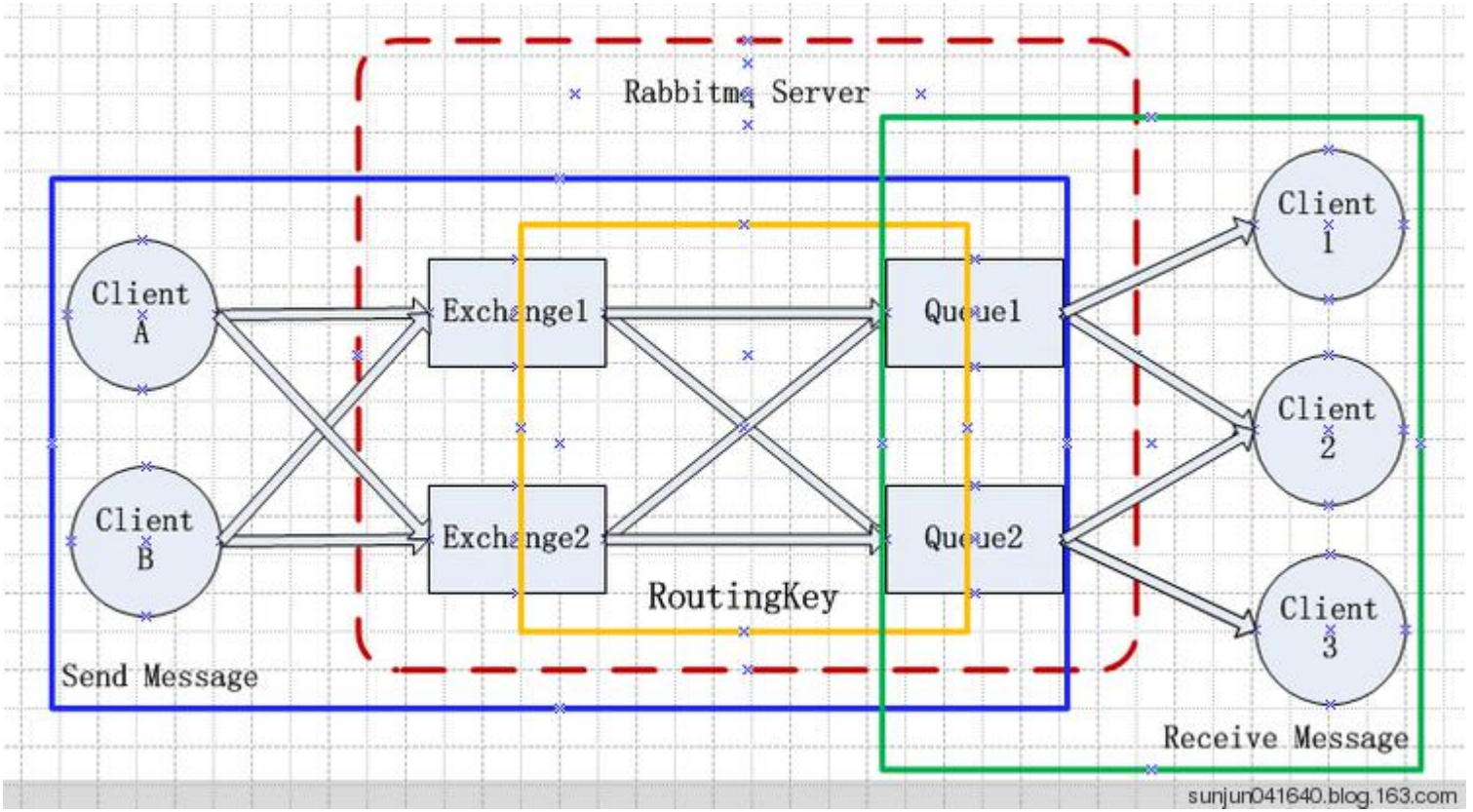
consumer：消息消费者，就是接受消息的程序。

channel：消息通道，在客户端的每个连接里，可建立多个channel，每个channel代表一个会话任务。

由Exchange，Queue，RoutingKey三个才能决定一个从Exchange到Queue的唯一的线路。

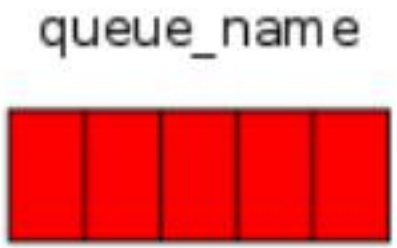
GNU/Linux-Rabbitmq

RabbitMQ工作原理



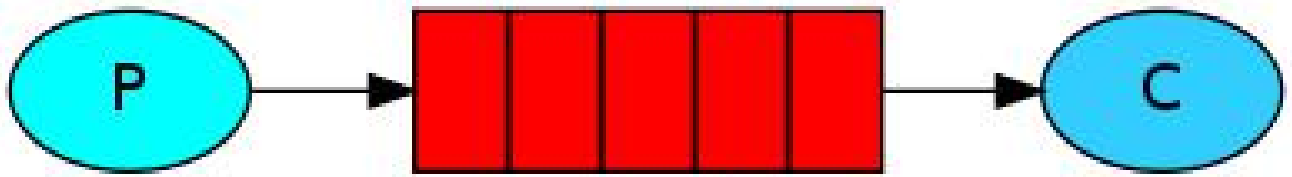
GNU/Linux-Rabbitmq

Queue（队列）是RabbitMQ的内部对象，用于存储消息，用下图表示



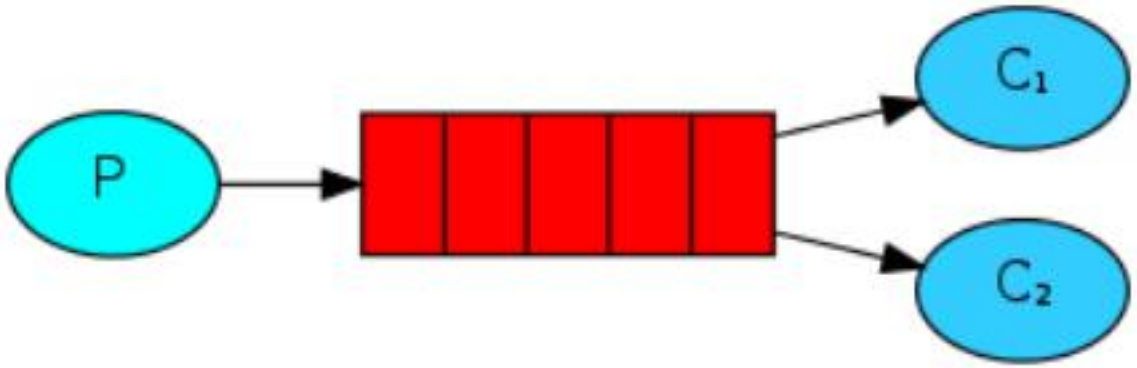
GNU/Linux-Rabbitmq

RabbitMQ中的消息都只能存储在Queue中，生产者（下图中的P）生产消息并最终投递到Queue中，消费者（下图中的C）可以从Queue中获取消息并消费。



GNU/Linux-Rabbitmq

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。

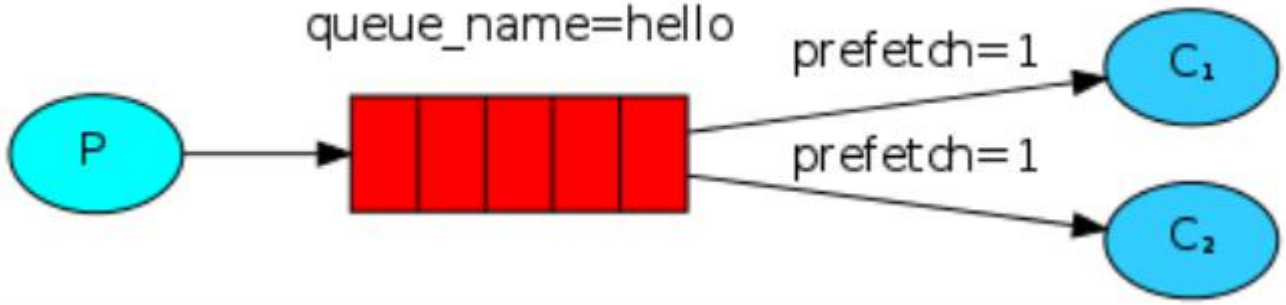


GNU/Linux-Rabbitmq

如果有多个消费者同时订阅同一个Queue中的消息，Queue中的消息会被平摊给多个消费者。这时如果每个消息的处理时间不同，就有可能会导致某些消费者一直在忙，而另外一些消费者很快就处理完手头工作并一直空闲的情况。

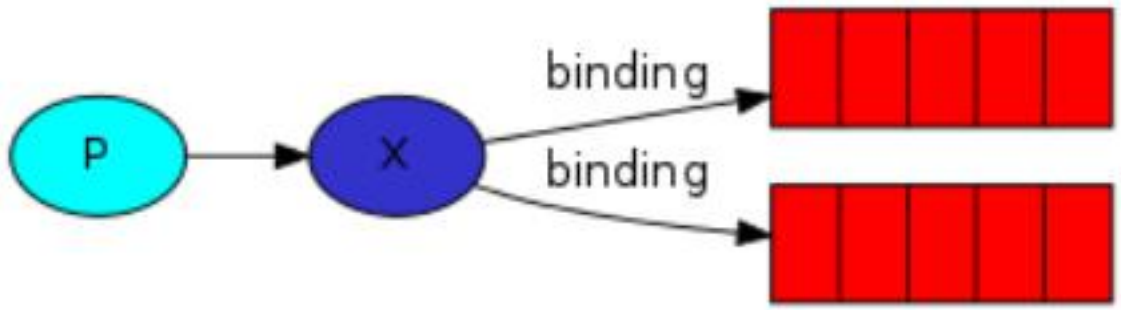
GNU/Linux-Rabbitmq

这时我们可以通过设置prefetchCount来限制Queue每次发送给每个消费者的消息数，比如我们设置prefetchCount=1，则Queue每次给每个消费者发送一条消息；消费者处理完这条消息后Queue会再给该消费者发送一条消息。



GNU/Linux-Rabbitmq

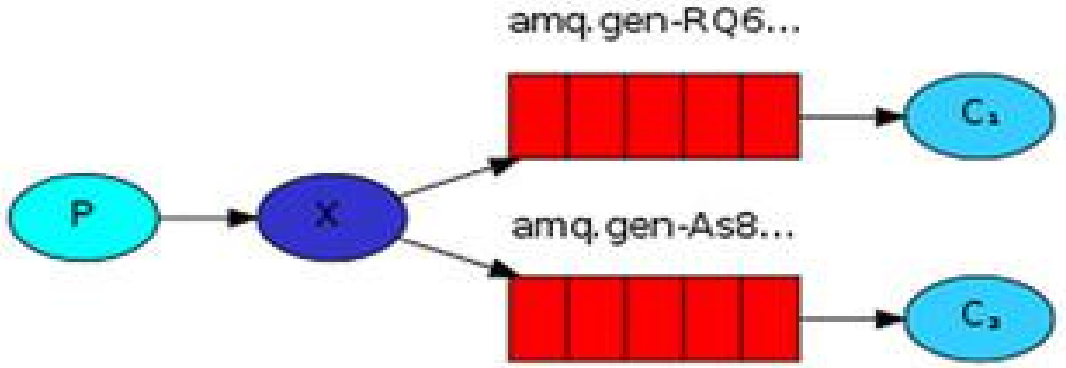
RabbitMQ中通过Binding将Exchange与Queue关联起来，这样RabbitMQ就知道如何正确地将消息路由到指定的Queue了。



GNU/Linux-Rabbitmq

队列可以保证每条消息发送给其中一个消费者，即每个消息只被处理一次。在实际应用中，经常会有这样的需求，每条消息要同时发送给多个消费者或者更复杂的情况。也就是说消息需要根据一定的规则发送给不同的消费者。

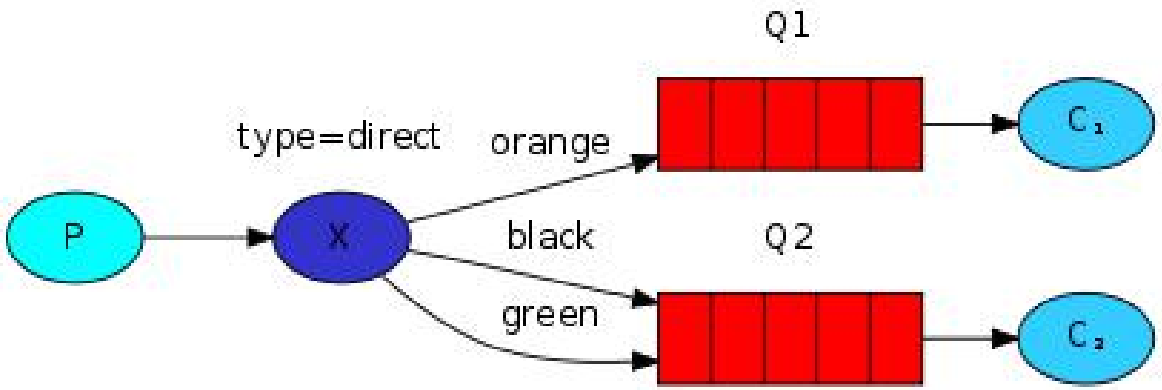
为实现消息路由，需要引入Exchange，图中用X表示。生产者不再直接发送消息给队列，而是先发送到Exchange。然后Exchange与队列绑定。这样消息会根据不同规则发送给不同队列，最终到达不同的消费者。



GNU/Linux-Rabbitmq

Routing

其实这个机制是建立在 Exchanges 上的，有了Route，我们就可以实现根据类别，让Exchange来选择性的分发任务给匹配的队列。要做到这点，只需要在为Exchange绑定queue时设置一个 routingKey 即可。



GNU/Linux-Rabbitmq

Binding Key

在绑定（Binding）Exchange与Queue的同时，一般会指定一个binding key；消费者将消息发送给Exchange时，一般会指定一个routing key；当binding key与routing key相匹配时，消息将会被路由到对应的Queue中。在绑定多个Queue到同一个Exchange的时候，这些Binding允许使用相同的binding key。

binding key 并不是在所有情况下都生效，它依赖于Exchange Type，比如fanout类型的Exchange就会无视binding key，而是将消息路由到所有绑定到该Exchange的Queue。

GNU/Linux-Rabbitmq

Exchange Types

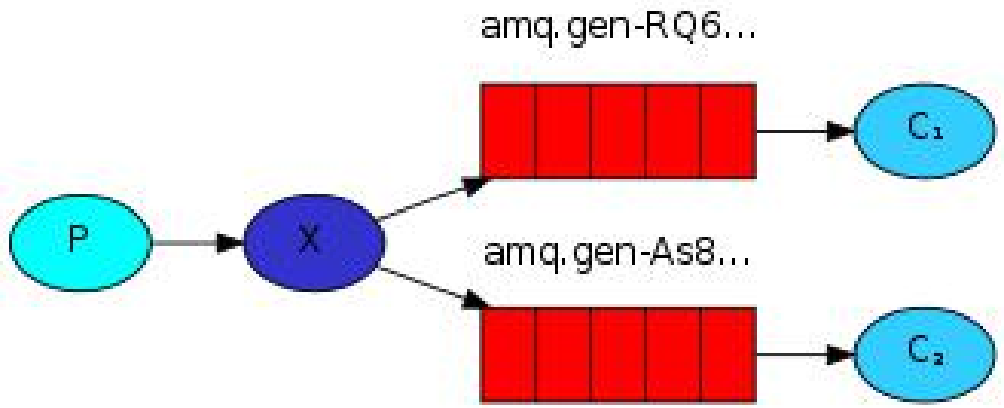
RabbitMQ常用的Exchange Type有fanout、direct、topic、headers这四种。

GNU/Linux-Rabbitmq

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。

下图中，生产者（P）发送到Exchange（X）的所有消息都会路由到图中的两个Queue，并最终被两个消费者（C1与C2）消费。

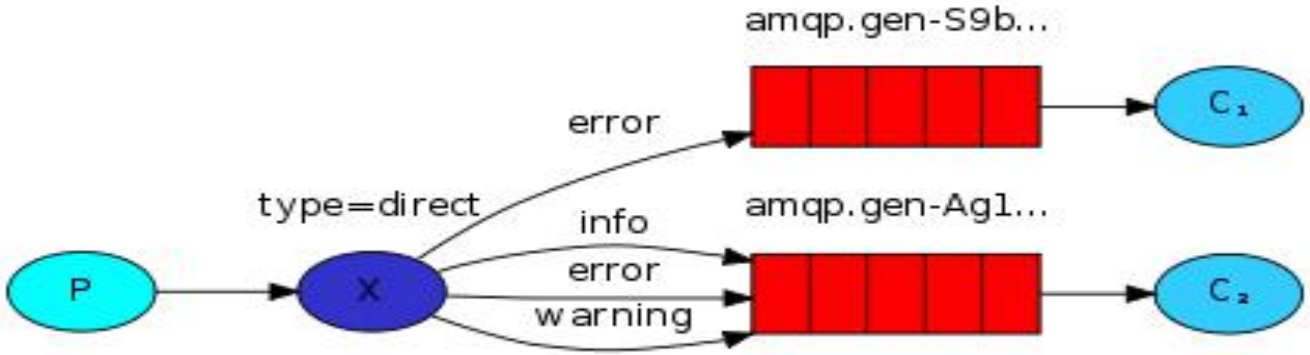


GNU/Linux-Rabbitmq

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。

以下图的配置为例，我们以routingKey=“error”发送消息到Exchange，则消息会路由到Queue1和Queue2，如果我们以routingKey=“info”或routingKey=“warning”来发送消息，则消息只会路由到Queue2。如果我们以其他routingKey发送消息，则消息不会路由到这两个Queue中。



GNU/Linux-Rabbitmq

topic

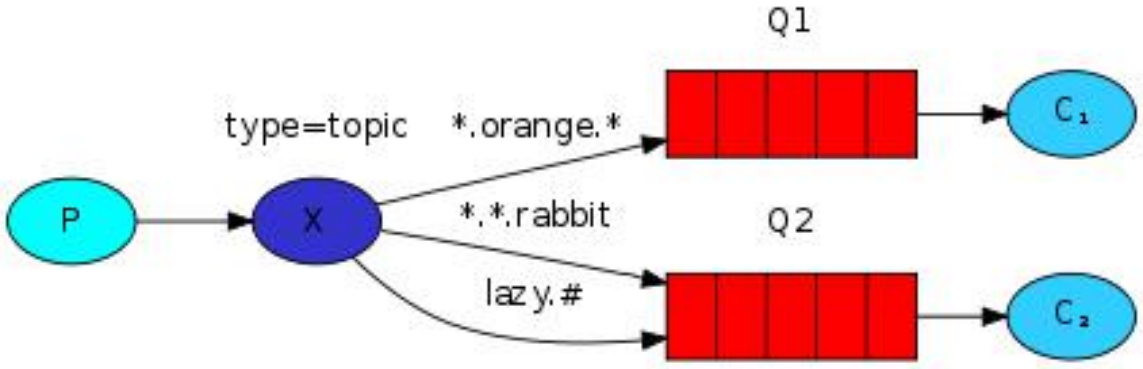
前面讲到direct类型的Exchange路由规则是完全匹配binding key与routing key，但这种严格的匹配方式在很多情况下不能满足实际业务需求。topic类型的Exchange在匹配规则上进行了扩展，它与direct类型的Exchage相似，也是将消息路由到binding key与routing key相匹配的Queue中，但这里的匹配规则有些不同，它约定：

1.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”

2.binding key与routing key一样也是句点号“.”分隔的字符串

3.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）

GNU/Linux-Rabbitmq



以上图中的配置为例，routingKey=“quick.orange.rabbit”的消息会同时路由到Q1与Q2，routingKey=“lazy.orange.fox”的消息会路由到Q1，routingKey=“lazy.brown.fox”的消息会路由到Q2，routingKey=“lazy.pink.rabbit”的消息会路由到Q2（只会投递给Q2一次，虽然这个routingKey与Q2的两个bindingKey都匹配）routingKey=“quick.brown.fox”、routingKey=“orange”、routingKey=“quick.orange.male.rabbit”的消息将会被丢弃，因为它们没有匹配任何bindingKey。

GNU/Linux-Rabbitmq

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。

GNU/Linux-Rabbitmq

Rabbitmq 特点

MQ 是消费 - 生产者模型的一个典型的代表，一端（生产者）往消息队列中不断写入消息，而另一端（消费者）则可以读取或者订阅队列中的消息。 MQ 则是遵循了 AMQP协议的具体实现和产品。

GNU/Linux-Rabbitmq

Rabbitmq 应用环境

在项目中，将一些无需即时返回且耗时的操作提取出来，进行了异步处理，而这种异步处理的方式大大的节省了服务器的请求响应时间，从而提高了系统的吞吐量。

GNU/Linux-Rabbitmq

Rabbitmq 安装

RabbitMQ 是基于 Erlang 的，所以必须具备 Erlang 环境。

GNU/Linux-Rabbitmq

Rabbitmq 安装

```
#yum --enablerepo=epel install rabbitmq-server
-y
```

GNU/Linux-Rabbitmq

#systemctl start rabbitmq-server

#systemctl enable rabbitmq-server

#netstat -lantp | grep 5672

GNU/Linux-Rabbitmq

让应用程序支持 Rabbitmq, 修改 guest 密码 (默认 guest 密码为 guest)

#rabbitmqctl change_password guest password

GNU/Linux-Rabbitmq

Rabbitmq 配置文件

#vi /etc/rabbitmq/rabbitmq.config

参考： http://www.rabbitmq.com/configure.html#confi-items

GNU/Linux-Rabbitmq

开启 Rabbitmq 之 WEB 管理

#rabbitmq-plugins enable rabbitmq_management

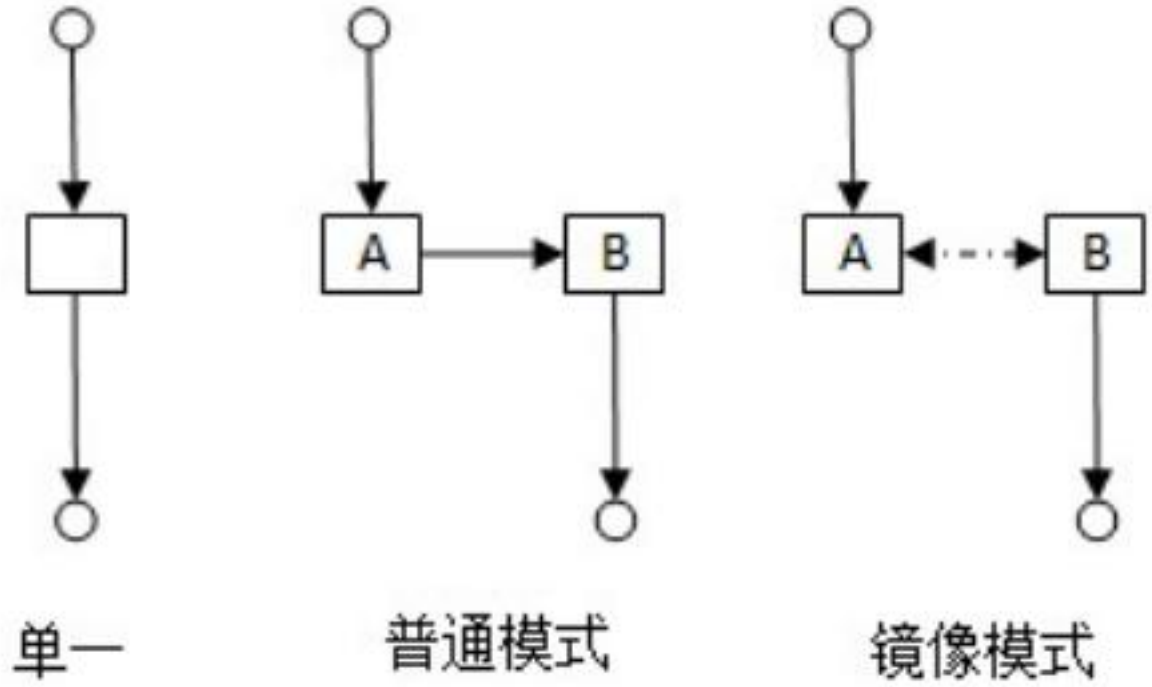
#systemctl restart rabbitmq-server

#netstat -lantp | grep 15672

使用浏览器访问 http://rabbitmq-server-ip:15672

GNU/Linux-Rabbitmq

Rabbitmq 之集群模式



GNU/Linux-Rabbitmq

Rabbitmq 之集群模式

单一模式：最简单的情况，非集群模式

普通模式：默认的集群模式。

镜像模式：把需要的队列做成镜像队列，存在于多个节点，属于RabbitMQ的HA方案。

GNU/Linux-Rabbitmq

普通模式：

对于Queue来说，消息实体只存在于其中一个节点，A、B两个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，消费者从B节点拉取时，RabbitMQ会临时在A、B间进行消息传输，把A中的消息实体取出并经过B发送给消费者。

所以消费者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，要在多个节点建立物理Queue。否则无论消费者连A或B，出口总在A，会产生瓶颈。

该模式存在一个问题就是当A节点故障后，B节点无法取到A节点中还未消费的消息实体。

如果做了消息持久化，那么得等A节点恢复，然后才可被消费；如果没有持久化的话，然后就没有然后了……

GNU/Linux-Rabbitmq

镜像模式：
该模式解决了上述问题，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，而不是在consumer取数据时临时拉取。

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大消耗掉。

所以在对可靠性要求较高的场合中适用(后面会详细介绍这种模式，目前我们搭建的环境属于该模式)

GNU/Linux-Rabbitmq

了解集群中的基本概念：

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，数据还是安全的放在磁盘。

GNU/Linux-Rabbitmq

Rabbitmq 的 metadata

元数据可以持久化在 RAM 或 Disc. 从这个角度看，也可以发现 RabbitMQ 集群中的节点分成两种 :RAM Node和 Disk Node. RAM Node 只会将元数据存放在 RAM,Disc node 会将元数据持久化到磁盘

单节点系统就没有什么选择了，只允许 disk node, 否则由于没有数据冗余一旦重启就会丢掉所有的配置信息 . 但在节点环境中可以选择哪些节点是 RAM node.

GNU/Linux-Rabbitmq

RAM Node 的性能优势

在集群中声明 (declare) 创建 exchange queue binding, 这类操作要等到所有的节点都完成创建才会返回， 如果是内存节点就要修改内存数据 , 如果是 disk node 就要等待写磁盘 , 节点过多这里的速度就会被大大的拖慢

GNU/Linux-Rabbitmq

RAM Node 的性能优势

有些场景 exchang queue 相当固定 , 变动很少 , 那即使全都是 disc node, 也没有什么影响 . 之前提到过使用 Rabbitmq 做 RPC, 如果是 RPC 或者类似 RPC 的场景这个问题就严重了 , 频繁创建销毁临时队列 , 磁盘读写能力就很快成为性能瓶颈了

GNU/Linux-Rabbitmq

RAM Node 的性能优势

所以，大多数情况下，我们尽量把 Node 创建为 RAM Node. 这里就有一个问题了，要想集群重启后元数据可以恢复就需要把集群元数据持久化到磁盘，那需要规划 RabbitMQ 集群中的 RAM Node 和 Disc Node 。

GNU/Linux-Rabbitmq

RAM Node or Disc Node

只要有一个节点是 Disc Node 就能提供条件把集群元数据写到磁盘 ,RabbitMQ 的确也是这样要求的 : 集群中只要有一个 disk node 就可以 , 其它的都可以是 RAM node. 节点加入或退出集群一定至少要通知集群中的一个 disk node 。

如果集群中 disk node 都当掉 , 就不要变动集群的元数据 . 声明 exchange queue 修改用户权限 , 添加用户等等这些变动在节点重启之后无法恢复 .

GNU/Linux-Rabbitmq

有一种情况要求所有的 disk node 都要在线情况在才能操作 , 那就是增加或者移除节点 .RAM node 启动的时候会连接到预设的 disk node 下载最新的集群元数据 . 如果你有两个 disk node(d1 d2), 一个 RAM node 加入的时候你只告诉 d1, 而恰好这个 RAM node 重启的时候 d1 并没有启动 , 重启就会失败 . 所以加入 RAM 节点的时候 , 把所有的disk node 信息都告诉它 ,RAM node 会把 disknode 的信息持久化到磁盘以便后续启动可以按图索骥 .

GNU/Linux-Rabbitmq

思路：

那么具体如何实现RabbitMQ高可用？下面，我们先搭建一个普通集群模式，在这个模式基础上再配置镜像模式实现高可用

GNU/Linux-Rabbitmq

Rabbitmq 之集群

0. 前提要求

1) 三台主机
2) 安装 rabbitmq 后不需要任何操作
3)NTP 启动
4)DNS 能够解析此三台主机

GNU/Linux-Rabbitmq

Rabbitmq 之集群

1. 修改配置文件 (所有节点)

#cd /etc/rabbitmq

#mv rabbitmq.config rabbitmq.config.bak

GNU/Linux-Rabbitmq

Rabbitmq 之集群

1. 修改配置文件 (所有节点)

```
#vi rabbitmq.config
[
{rabbit,[
{default_user,<<"guest">>},
{default_pass,<<"guest">>}
]},
{kernel,[
]}
].
```

GNU/Linux-Rabbitmq

Rabbitmq 之集群

2. 将管理节点 cookie 复制给其他节点（其他节点需要修改属主属组）

#cat /var/lib/rabbitmq/.erlang.cookie

3. 复制完成后所有节点重新启动 rabbitmq-server 服务

GNU/Linux-Rabbitmq

Rabbitmq 之集群

4. 主节点配置

```
# rabbitmqctl stop_app
# rabbitmqctl reset
#/usr/lib/rabbitmq/bin/rabbitmq-plugins enable
rabbitmq_management
#/usr/lib/rabbitmq/bin/rabbitmq-plugins enable
rabbitmq_management_agent
#rabbitmqctl start_app
```

GNU/Linux-Rabbitmq

Rabbitmq 之集群

5. 查看各节点中的 RabbitMQ brokers

```
# rabbitmqctl cluster_status
```

6. 其他节点操作

```
#rabbitmqctl stop_app
#rabbitmqctl join_cluster --ram rabbit@master_node
#rabbitmqctl start_app
```

GNU/Linux-Rabbitmq

Rabbitmq 之集群

7. 确认各节点中的 RabbitMQ brokers
rabbitmqctl cluster_status

8. 浏览器访问
http://rabbitmq_srv_ip:15672

GNU/Linux-Rabbitmq

Rabbitmq 之集群管理理

1. 查看集群状态 (可分别在集群中各个节点执行)
#rabbitmqctl cluster_status

2. 更改节点类型（内存型或磁盘型）
#rabbitmqctl stop_app
#rabbitmqctl change_cluster_node_type disc
或
#rabbitmqctl change_cluster_node_type ram
#rabbitmqctl start_app

GNU/Linux-Rabbitmq

Rabbitmq 之集群管理

3. 重启 cluster 中的节点 (停止某个节点或者节点 down 掉剩余节点不受影响)

#rabbitmqctl stop

4. 待节点重启后自动追上其他节点

#systemctl start rabbitmq-server

GNU/Linux-Rabbitmq

Rabbitmq 之集群管理

1. 保证集群中至少有一个磁盘类型的节点以防数据丢失，在更改节点类型时尤其要注意。

2. 若整个集群被停掉了，应保证最后一个 down 掉的节点被最先启动，若不能则要使用 forget_cluster_node 命令将其移出集群

3. 若集群中节点几乎同时以不可控的方式 down 了此时在其中一个节点使用 force_boot 命令重启节点

GNU/Linux-Rabbitmq

Rabbitmq 之集群管理理

5. 从集群移除节点

#rabbitmqctl stop_app

#sudo rabbitmqctl reset

#rabbitmqctl start_app

#rabbitmqctl cluster_status

GNU/Linux-Rabbitmq

Rabbitmq 之集群管理

6. 从某个节点移除集群中其他节点

```
#rabbitmqctl forget_cluster_node rabbit@other_node
#rabbitmqctl reset
#rabbitmqctl start_app
#rabbitmqctl cluster_status
```

GNU/Linux-Rabbitmq

Rabbitmq 之集群管理

7. 自动配置 cluster

1) 重置各节点
#rabbitmqctl stop_app
#rabbitmqctl reset

2) 其次各节点调整配置文件
[{rabbit,
 [{cluster_nodes,
{['rabbit@master_node','rabbit@node1','rabbit@node2'], disc}}]}].

GNU/Linux-Rabbitmq

Rabbitmq 之集群管理

7. 自动配置 cluster

3) 启动各节点
#systemctl start rabbitmqctl

4) 查看集群状态
#rabbitmqctl cluster_status

GNU/Linux-Rabbitmq

Rabbitmq 之集群管理

1. 无论通过命令行还是通过配置文件配置，请确保各节点上 Erlang 和 RabbitMQ 版本一致

2. 配置文件仅对新鲜节点有效，也即被 reset 或者第一次启动的节点。因此在重启节点后自动化集群过程并不会发生。也以为这通过 rabbitmq 进行的改变优先于自动化集群配置。

GNU/Linux-Rabbitmq

RabbitMQ镜像模式配置

上面配置RabbitMQ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。

GNU/Linux-Rabbitmq

虚拟主机

1.创建虚拟主机
rabbitmqctl add_vhost vhostpath

2.删除虚拟主机
rabbitmqctl delete_vhost vhostpath

3.列出所有虚拟主机
rabbitmqctl list_vhosts

GNU/Linux-Rabbitmq

在cluster中任意节点启用策略，策略会自动同步到集群节点

```
# rabbitmqctl set_policy -p hrsystem ha-allqueue"^" '{"ha-mode":"all"}'
```

这行命令在vhost名称为hrsystem创建了一个策略，策略名称为ha-allqueue, 策略模式为 all 即复制到所有节点，包含新增节点，策略正则表达式为 “^” 表示所有匹配所有队列名称。 例如rabbitmqctl set_policy -p hrsystem ha-allqueue "^message" '{"ha-mode":"all"}' 注意："^message" 这个规则要根据自己修改，这个是指同步"message"开头的队列名称，我们配置时使用的应用于所有队列，所以表达式为"^"

GNU/Linux-Rabbitmq

RabbitMQ策略

在RabbitMQ的Web管理界面（RabbitMQ_srv_ip:15672 ） 可以查看到以下策略信息

Virtual Host	Name	Pattern	Definition	Priority
server	**ha-allqueue**	^	ha-mode: all	0

GNU/Linux-Rabbitmq

镜像模式：把需要的队列做成镜像队列，存在于多个节点。

该模式解决了上述问题，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，而不是在消费者取数据时临时拉取。

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大消耗掉。所以在对可靠性要求较高的场合中适用。

GNU/Linux-Rabbitmq

RabbitMQ的HAproxy方案

集群的配置已经搭建好了，成功做到了高可用，但是我们的程序连接节点并不会管哪个服务器在忙、哪个服务器空闲，完全看心情想连谁就连谁。而且代码中要把每个ip的节点都手动的写出来，既然是手动的就很有可能发现写错这种情况，同样WEB UI 通常也不知道打开哪个好，因为每个服务器都有一个 WEB UI，可能有人说，既然哪个都行，你随便打开一个就是了。但是如果不巧这个服务器后面崩了呢。

GNU/Linux-Rabbitmq

RabbitMQ的HAproxy方案

基于前面的问题，我们就需要一个强大的负载均衡服务器来帮助我们完成这些事情。这里我们选择haproxy。

GNU/Linux-Rabbitmq

RabbitMQ的HAproxy方案

准备工作就是安装haproxy
#yum install haproxy -y

安装完成以后，编辑配置文件
vim /etc/haproxy/haproxy.conf

GNU/Linux-Rabbitmq

RabbitMQ的HAproxy方案

这里主要是针对WEB UI与集群负载均衡的配置
在配置文件底部添加：

```
listen rabbitmq_cluster 0.0.0.0:25672
        mode tcp #配置TCP模式
        mode http #配置http模式
        option tcplog
        balance roundrobin  #简单的轮询
```

GNU/Linux-Rabbitmq

RabbitMQ的HAproxy方案

#RabbitMQ集群节点配置

server rqmaster IP1:5672 check inter 5000 rise 2 fall 2
 server rqslave1 IP2:5672 check inter 5000 rise 2 fall 2
 server rqslave2 IP3:5672 check inter 5000 rise 2 fall 2

#服务器定义(check指健康状况检查，inter 5000指检测频率;rise 2指从离线状态转换至正常状态需要成功检查的次数；fall 2指失败2次即认为服务器不可用)

GNU/Linux-Rabbitmq

RabbitMQ的HAproxy方案

#配置haproxy web监控，查看统计信息

```
listen private_monitoring HAproxy_srv_ip:8100
    mode http
    option httplog
    stats enable
     stats uri /stats
    stats refresh 30s
    stats auth admin:1234  #管理员账户密码
```

GNU/Linux-Rabbitmq

RabbitMQ的HAproxy方案

配置文件写完以后，运行指令

haproxy -f /etc/haproxy/haproxy.conf

这时，配置完毕，可以通过访问HAproxy的Web管理界面查看所做的负载均衡了

访问：http://HAproxy_srv_ip:8100/stats

GNU/Linux-Rabbitmq

RabbitMQ的HAproxy方案

HAProxy version 1.5.14, released 2015/07/02

Statistics Report for pid 4230

> General process information

pid = 4230 (process #1, nbproc = 1)
uptime = 0d 0h00m14s
system limits: memmax = unlimited; ulimit-n = 8035
maxsock = 8035; **maxconn** = 4000; **maxpipes** = 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec
Running tasks: 1/8; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

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

External resources:
- Primary site
- Updates (v1.5)
- Online manual

rabbitmq_cluster

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	0	-	0	0	3 000	0			0	0	0	0	0					OPEN								
rqmaster	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	14s UP	L4OK in 0ms	1	Y	-	0	0	0s	-
rqslave1	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	14s UP	L4OK in 0ms	1	Y	-	0	0	0s	-
rqslave2	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	14s UP	L4OK in 0ms	1	Y	-	0	0	0s	-
Backend	0	0		0	0		0	0	300	0	0	?	0	0	0	0		0	0	0	0	14s UP		3	3	0		0	0s	

private_monitoring

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				1	1	-	1	1	3 000	1			0	0	0	0	0					OPEN								
Backend	0	0		0	0		0	0	300	0	0	0s	0	0	0	0		0	0	0	0	14s UP		0	0	0		0		