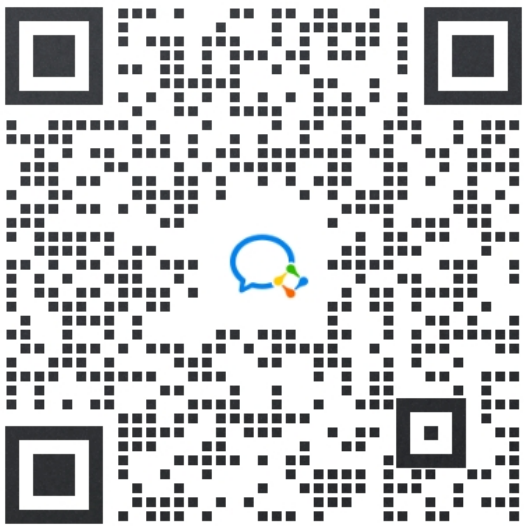
6. RabbitMQ运维

版权说明



上面的章节介绍了RabbitMQ的安装及运行, 但这都是单机版的, 无法满足目前真实应用的要求. 试想一下, 如果RabbitMQ服务器遇到内存崩溃, 断电或者主板故障等情况, 该怎么办?

假如单台RabbitMQ服务器可以满足每秒1000条消息的吞吐量, 应用需要 RabbitMQ服务满足每秒10万条消息的吞吐量, 购买昂贵的服务器来增强单机RabbitMQ服务的性能显得捉襟见肘, 这时候就需要通过搭建RabbitMQ集群来解决问题了.

RabbitMQ集群允许消费者和生产者在RabbitMQ单个节点崩溃的情况下继续运行, 它可以通过添加更多的节点来线性地扩展消息通信的吞吐量. 当失去一个RabbitMQ节点时, 客户端能够重新连接到集群中的

任何其他节点并继续生产或者消费.

> 不过RabbitMQ集群不能保证消息的万无一失, 即使把消息, 队列, 交换器等都设置为可持久化, 生产端和消费端都正确地使用了确认方式. 当集群中一个RabbitMQ节点崩溃时,该节点上的所有队列中的消息也会丢失.
>
> RabbitMQ集群中的所有节点都会备份所有的元数据信息(队列, 交换机的名称及属性, 以及他们之间的绑定关系,还有vhost等相关信息), 但是不会备份消息, 当然这可以通过一些配置解决这个问题.

接下来我们来讲下, 如何正确有效的搭建一个RabbitMQ集群

1. 多机多节点

RabbitMQ集群对延迟非常敏感, 所以搭建RabbitMQ集群时, 多个节点应当在同一个局域网内.

接下来我们采用三台局域网的云服务器来搭建RabbitMQ集群.

> 🎁 为避免兼容性问题, 所有节点上的RabbitMQ版本应该相同

搭建步骤参考**[多机多节点搭建集群]**

2. 单机多节点

我们搭建集群, 需要多个服务器, 并且要求是局域网的, 如果只是想要学习验证集群的某些特性, 也可以选择单机多节点的方式来搭建.

也就是只有一个虚拟机, 我们启动N个节点, 用端口号来区分

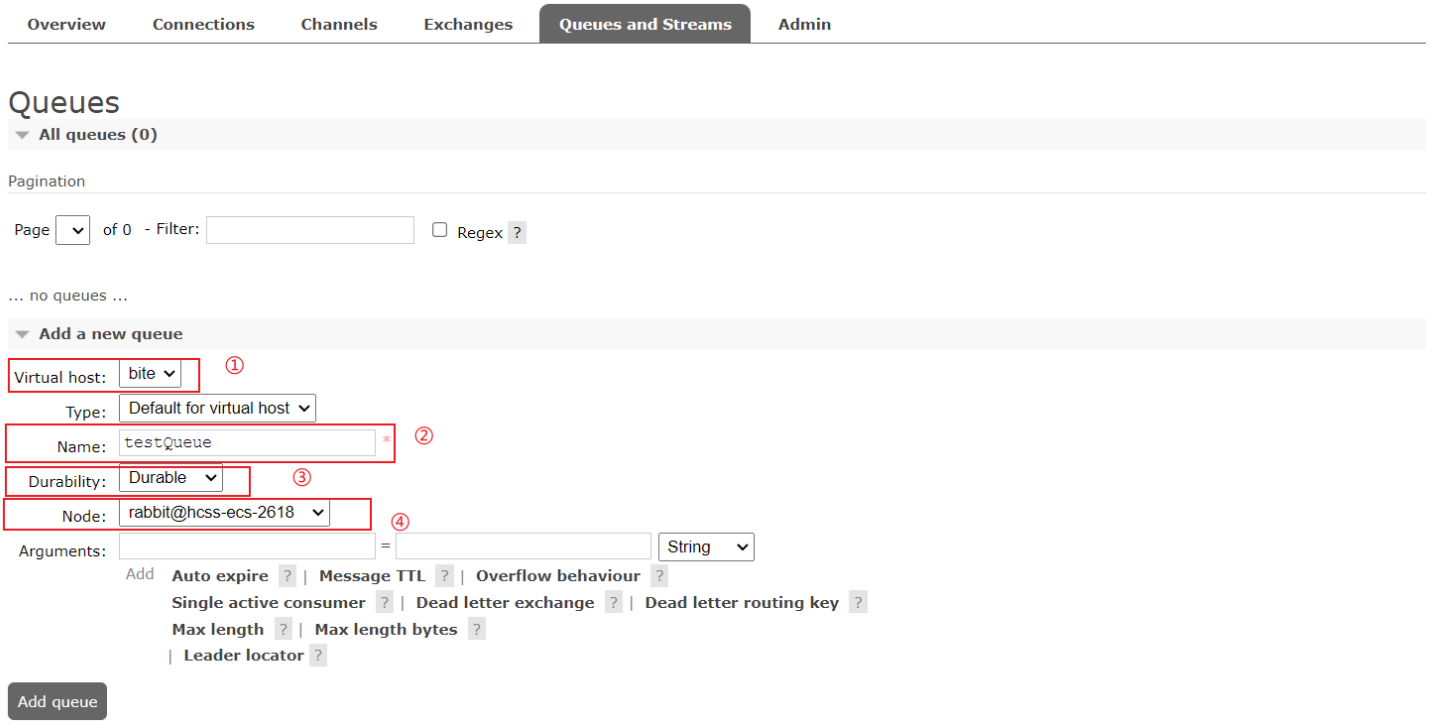
> 单机多节点的方式, 也称为"伪集群".
>
> 搭建集群是为了提高RabbitMQ服务的可用性, 采用单机多节点的方式, 如果这台机器挂掉了, 整个集群也会挂掉, 所以在实际生产环境中并不使用这种方式, 此处为演示学习使用.

搭建步骤参考 **[单机多节点搭建集群]**

3. 宕机演示

安装之后, 是有问题的, 也就是数据不同步. 我们可以来看下是什么问题

1. 添加队列



①: 选择虚拟机(需要保证操作用户对当前虚拟机有操作权限)

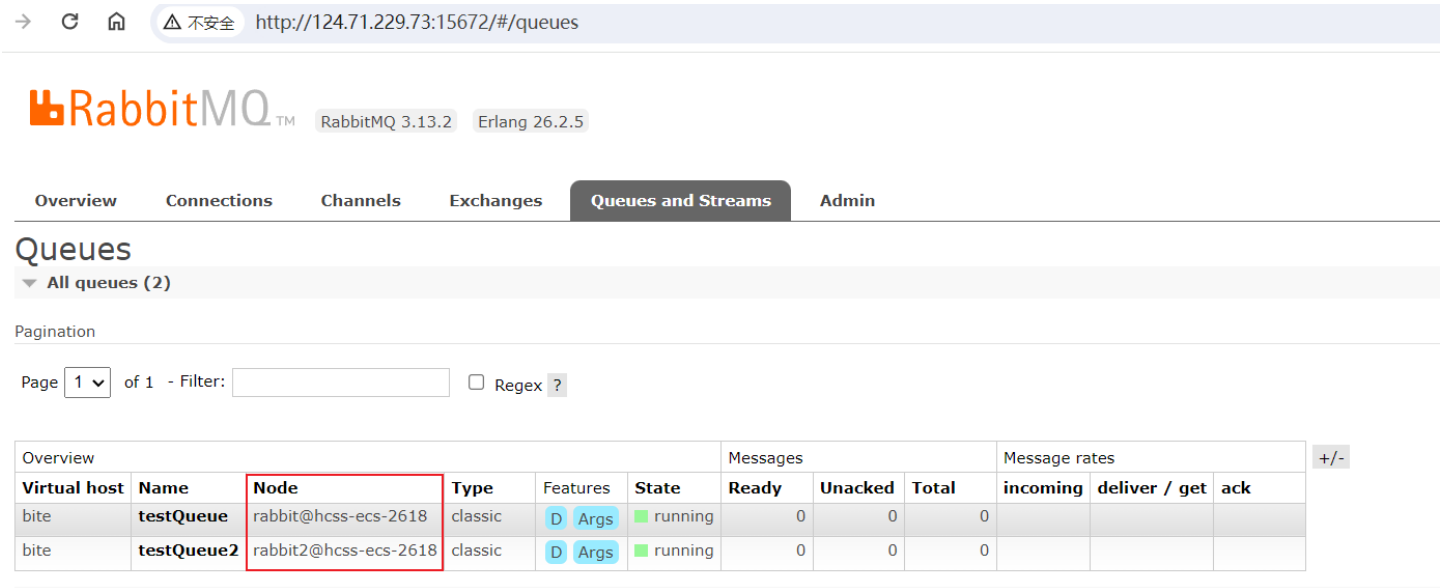
②: 设置队列名称

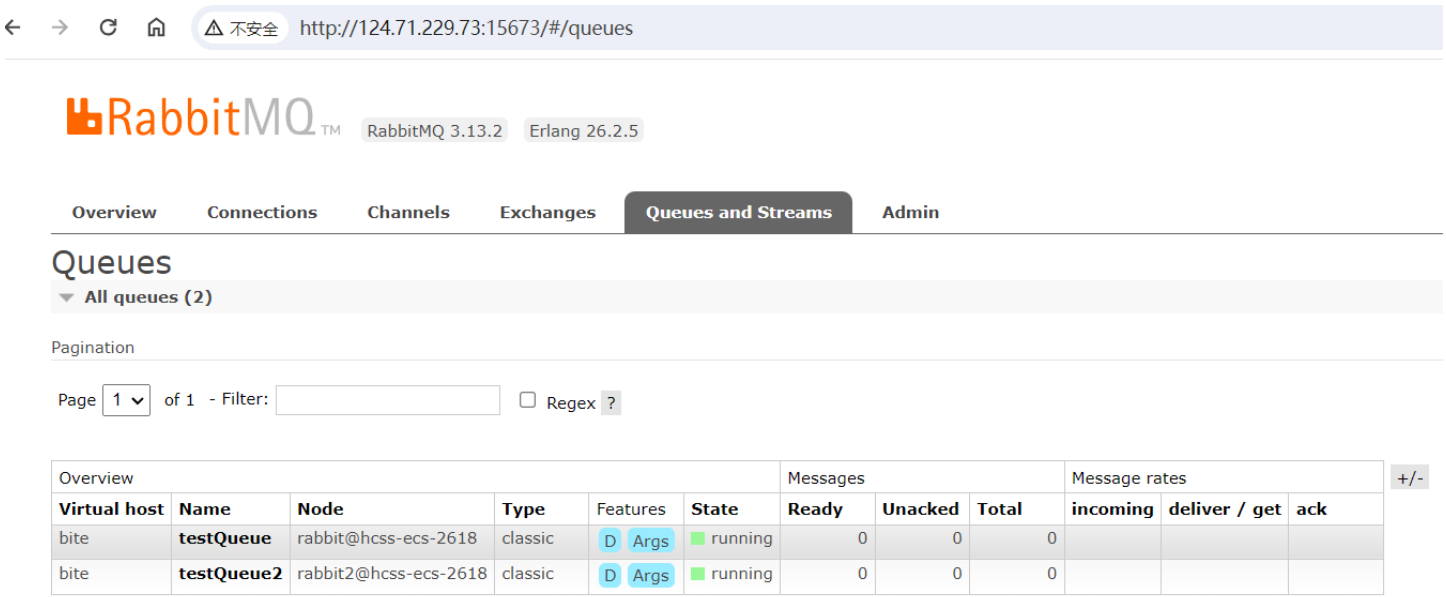
③: 是都持久化

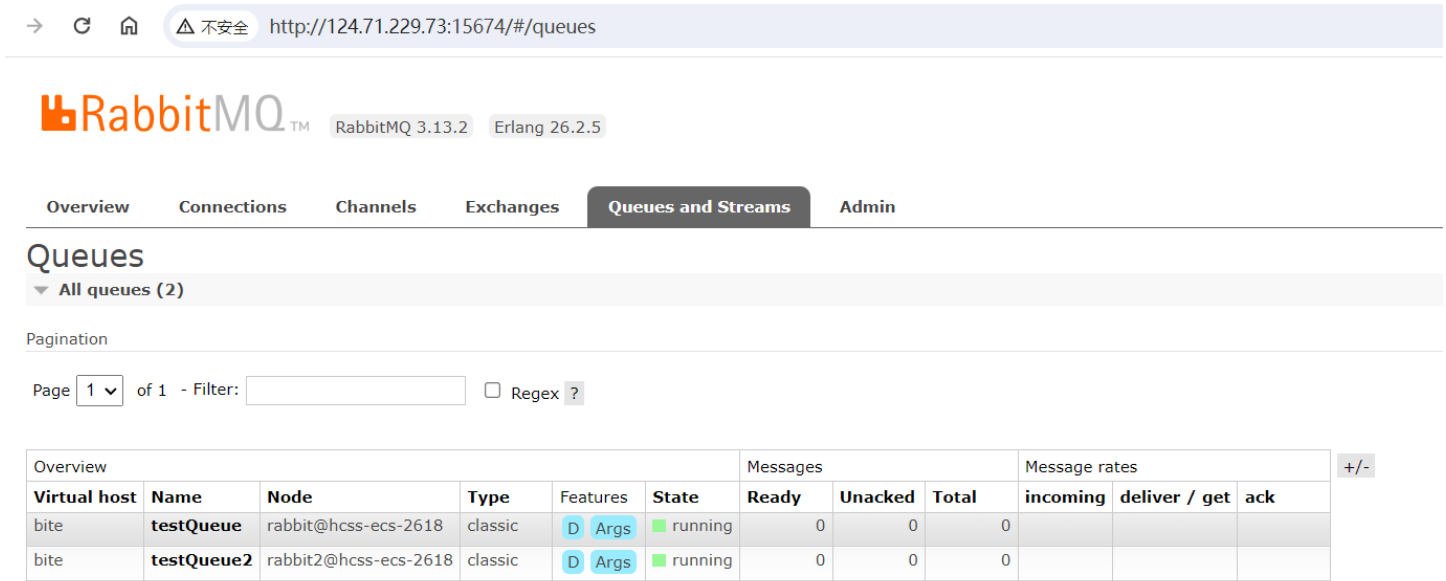
④: 指定主节点, 其他为从节点

分别以rabbit节点和rabbit2节点添加两个队列

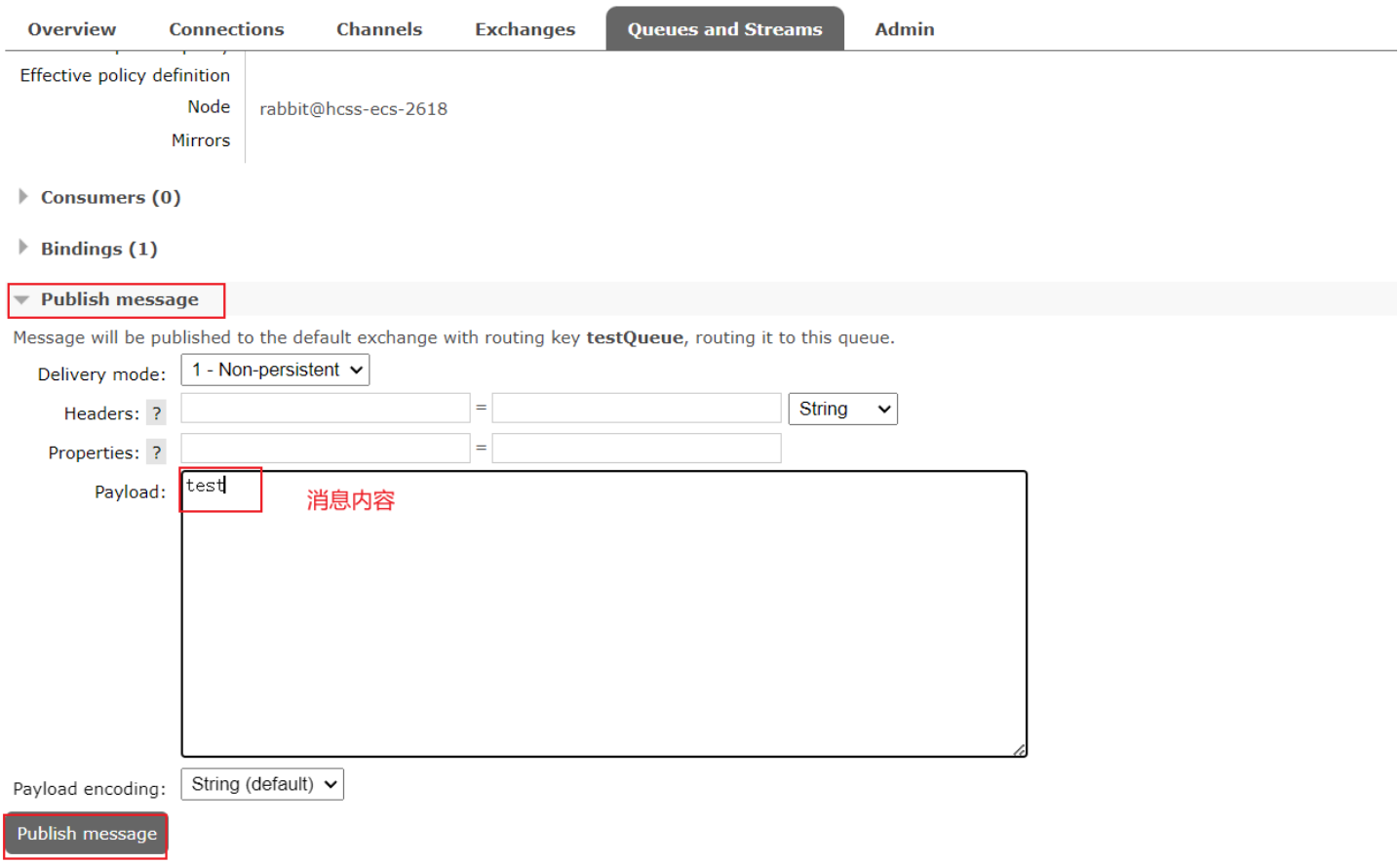
2. 添加之后,可以看到三个节点都有队列了



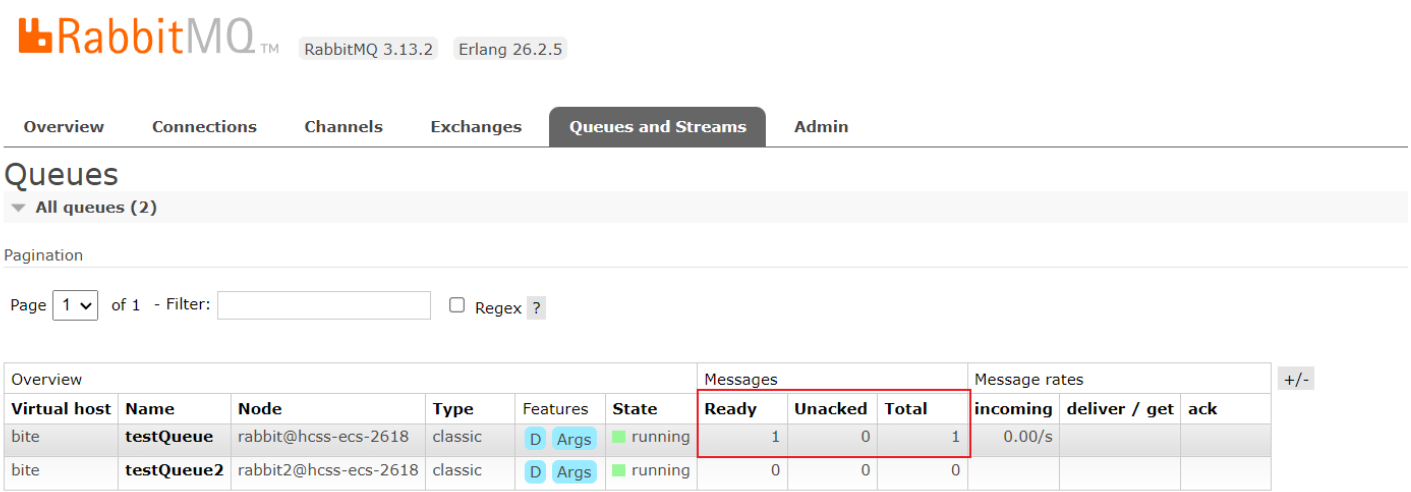




3. 往testQueue队列中发送一条数据(从任一节点都可以)



发送之后, 观察3个节点的队列中均有消息

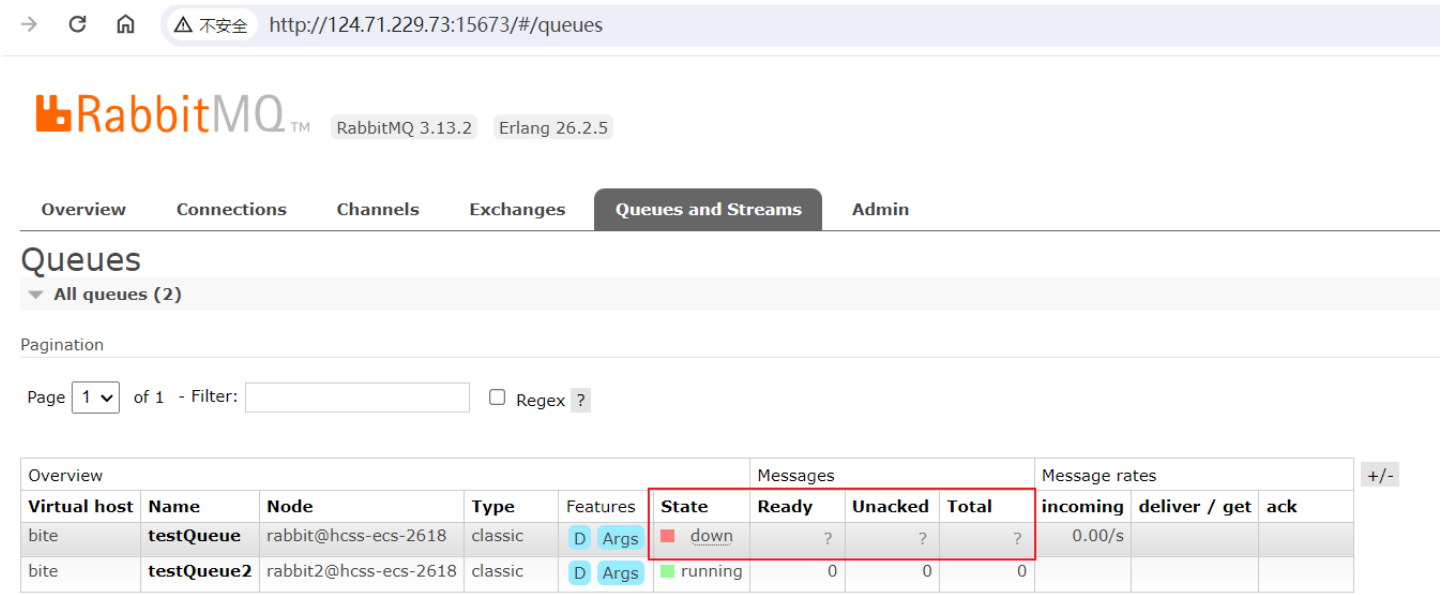


4. 关闭主节点

```
root@hcss-ecs-2618:~# rabbitmqctl -n rabbit stop_app   #rabbit为节点名称
Stopping rabbit application on node rabbit@hcss-ecs-2618 ...
root@hcss-ecs-2618:~#

```

关闭后可以看到rabbit2和rabbit3 没有该队列的数据了, 但是另一个队列不受影响



> RabbitMQ版本不同, 界面显示会有些差异

也就是说, 这个数据只在主节点存在, 从节点没有

如果关闭的是rabbit2, 那么testQueue2的数据会消失

如何解决这个问题呢, 就需要引入咱们的"仲裁队列"

4. 仲裁队列(Quorum Queues)

RabbitMQ 的仲裁队列是一种基于 Raft 一致性算法实现的持久化、复制的 FIFO 队列. 仲裁队列提供队列复制的能力, 保障数据的高可用和安全性. 使用仲裁队列可以在 RabbitMQ 节点间进行队列数据的复制, 从而达到在一个节点宕机时, 队列仍然可以提供服务的效果.

官方文档: Quorum Queues | RabbitMQ

仲裁队列是RabbitMQ 3.8版本最重要的改动. 他是镜像队列的替代方案. 在 RabbitMQ 3.8 版本问世之前, 镜像队列是实现数据高可用的唯一手段, 但是它有一些设计上的缺陷, 这也是 RabbitMQ 提供仲裁队列的原因. 经典镜像队列已被弃用，并计划在将来版本中移除, 如果当前使用经典镜像队列的 RabbitMQ 安装需要迁移, 可以参考官方提供的迁移指南 Migrate your RabbitMQ Mirrored Classic Queues to Quorum Queues | RabbitMQ

Raft协议介绍

什么是Raft?

```
1 Raft is a consensus algorithm for managing a replicated log. It produces a
  result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its
  structure is different from Paxos; this makes Raft more understandable than
```

```
Paxos and also provides a better foundation for building practical systems. In
order to enhance understandability, Raft separates the key elements of
consensus, such as leader election, log replication, and safety, and it
enforces a stronger degree of coherency to reduce the number of states that
must be considered. Results from a user study demonstrate that Raft is easier
for students to learn than Paxos. Raft also includes a new mechanism for
changing the cluster membership, which uses overlapping majorities to
guarantee safety.
```

> 摘自: [**In Search of an Understandable Consensus Algorithm**] , 更建议直接阅读原文.

Raft 是一种用于管理和维护分布式系统一致性的协议, 它是一种共识算法, 旨在实现高可用性和数据的持久性. Raft 通过在节点间复制数据来保证分布式系统中的一致性，即使在节点故障的情况下也能保证数据不会丢失.

在分布式系统中, 为了消除单点提高系统可用性, 通常会使用副本来进行容错, 但这会带来另一个问题, 即如何保证多个副本之间的一致性?

共识算法（Consensus Algorithm）就是做这个事情的, 它允许多个分布式节点就某个值或一系列值达成一致性协议. 即使在一些节点发生故障, 网络分区或其他问题的情况下, 共识算法也能保证系统的一致性和数据的可靠性.

常见的共识算法有: Paxos, Raft, Zab等

- **Paxos:** 一种经典的共识算法, 用于解决分布式系统中的一致性问题.
- **Raft**：一种较新的共识算法, Paxos 不易实现, raft是对Paxos算法的简化和改进, 旨在易于理解和实现.
- **Zab**：ZooKeeper 使用的共识算法, 基于 Paxos 算法., 大部分和Raft相同, 主要区别是对于Leader的任期, Raft叫做term, Zab叫做epoch, 状态复制的过程中, raft的心跳从Leader向Follower发送, 而ZAB则相反.
- **Gossip**: Gossip算法每个节点都是对等的, 即没有角色之分. Gossip算法中的每个节点都会将数据改动告诉其他节点（类似传八卦）

Raft 基本概念

Raft 使用 Quorum 机制来实现共识和容错, 我们将对 Raft 集群的操作必须得到大多数(> N/2)节点的同意才能提交.

> 节点指的是分布式系统中的一个独立成员

当我们向 Raft 集群发起一系列读写操作时, 集群内部究竟发生了什么呢? 我们先来简单了解一下.

Raft 集群必须存在一个主节点(Leader), 客户端向集群发起的所有操作都必须经由主节点处理. 所以 Raft 核心算法中的第一部分就是选主(Leader election). 没有主节点集群就无法工作, 先选出一个主节点, 再考虑其它事情.

主节点会负责接收客户端发过来的操作请求, 将操作包装为日志同步给其它节点, 在保证大部分节点都同步了本次操作后, 就可以安全地给客户端回应响应了. 这一部分工作在 Raft 核心算法中叫日志复制 (Log replication).

因为主节点的责任非常大, 所以只有**符合条件**的节点才可以当选主节点. 为了保证集群对外展现的一致性, 主节点在处理操作日志时, 也一定要谨慎, 这部分在Raft核心算法中叫 安全性(**Safety**).

Raft算法将一致性问题分解为三个子问题: Leader选举, 日志复制和安全性.

下面我们详细介绍下Raft的选主过程.

选主(Leader election)

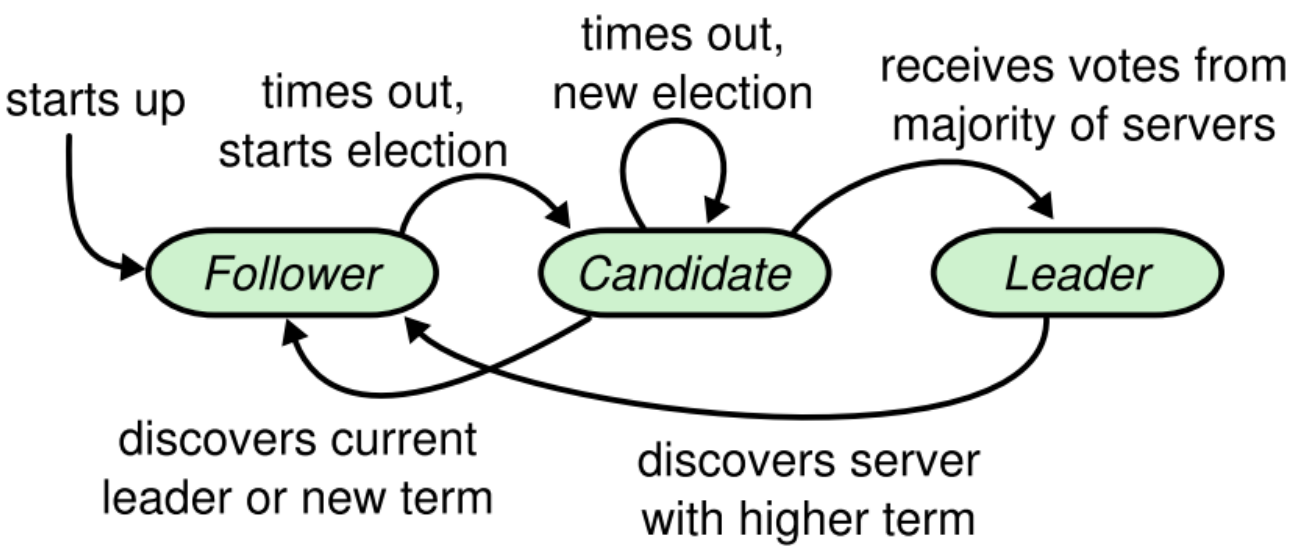
选主(Leader election)就是在集群中抉择出一个主节点来负责一些特定的工作. 在执行了选主过程后, 集群中每个节点都会识别出一个特定的, 唯一的节点作为 leader.

节点角色

在 Raft 算法中，每个节点都处于以下三种角色之一

- **Leader(领导者)**: 负责处理所有客户请求，并将这些请求作为日志项复制到所有 Follower. Leader 定期向所有 Follower 发送心跳消息, 以维持其领导者地位, 防止 Follower 进入选举过程.
- **Follower(跟随者)**: 接收来自 Leader 的日志条目, 并在本地应用这些条目. 跟随者不直接处理客户请求.
- **Candidate(候选者)**: 当跟随者在一段时间内没有收到来自 Leader 的心跳消息时，它会变得不确定 Leader 是否仍然可用. 在这种情况下, 跟随者会转变角色成为 Candidate, 并开始尝试通过投票过程成为新的 Leader.

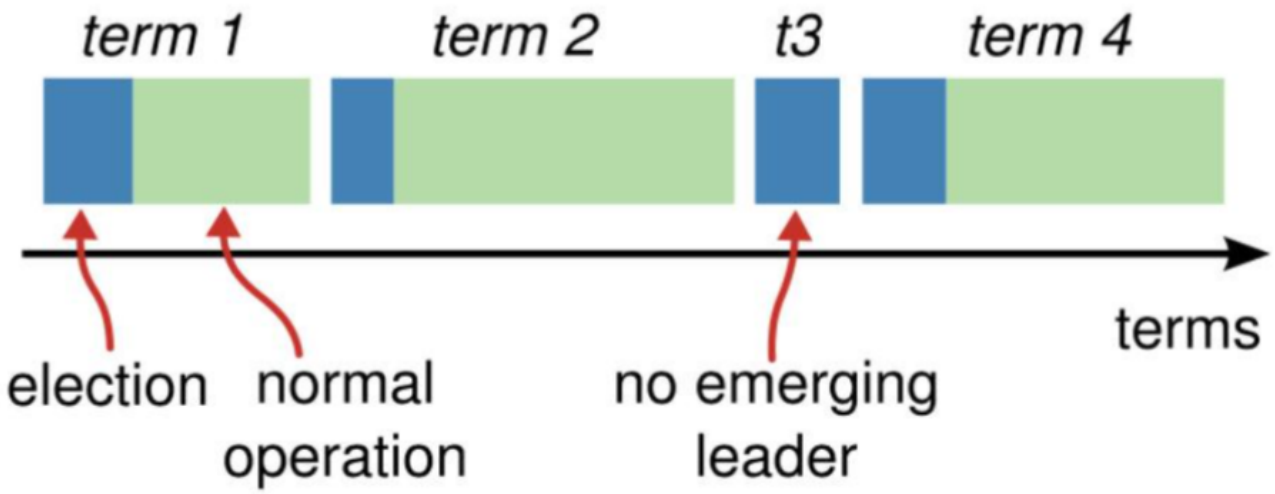
在正常情况下, 集群中只有一个 Leader, 剩下的节点都是 follower, 下图展示了这些状态和它们之间的转换关系



可以看出所有节点在启动时, 都是follow状态, 在一段时间内如果没有收到来自leader的心跳, 从follower切换到candidate, 发起选举. 如果收到多数派(majority)的投票(含自己的一票) 则切换到leader状态. Leader一般会一直工作直到它发生异常为止.

任期

Raft 将时间划分成任意长度的任期(term). 每一段任期从一次选举开始, 在这个时候会有一个或者多个 candidate 尝试去成为 leader. 在成功完成一次leaderelection之后,一个leader就会一直节管理集群直到任期结束. 在某些情况下, 一次选举无法选出 leader, 这个时候这个任期会以没有 leader 而结束(如下图t3). 同时一个新的任期(包含一次新的选举) 会很快重新开始



Term 更像是一个逻辑时钟(logic clock)的作用, 有了它，就可以发现哪些节点的状态已经过期. 每一个节点都保存一个 current term, 在通信时带上这个 term的值.

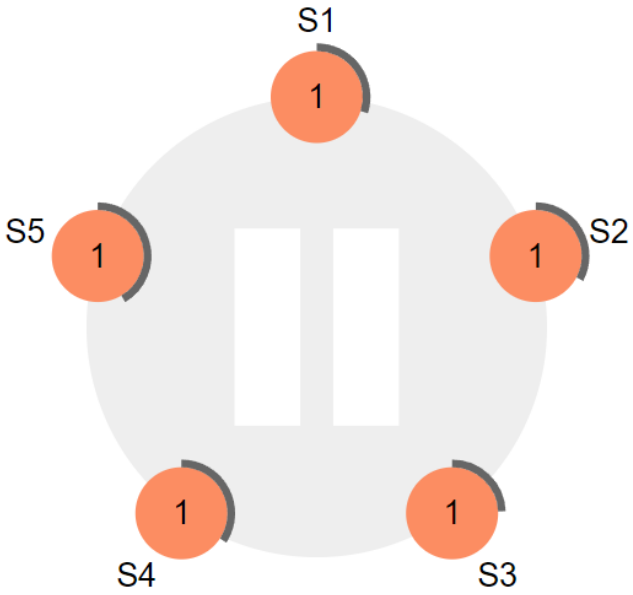
每一个节点都存储着一个当前任期号(current term number). 该任期号会随着时间单调递增. 节点之间通信的时候会交换当前任期号, 如果一个节点的当前任期号比其他节点小, 那么它就将自己的任期号更新为较大的那个值. 如果一个 candidate 或者 leader 发现自己的任期号过期了, 它就会立刻回到 follower 状态. 如果一个节点接收了一个带着过期的任期号的请求, 那么它会拒绝这次请求.

Raft 算法中服务器节点之间采用 RPC 进行通信, 主要有两类 RPC 请求:

- RequestVote RPCs: 请求投票, 由 candidate 在选举过程中发出
- AppendEntries RPCs: 追加条目, 由 leader 发出, 用来做日志复制和提供心跳机制

选举过程

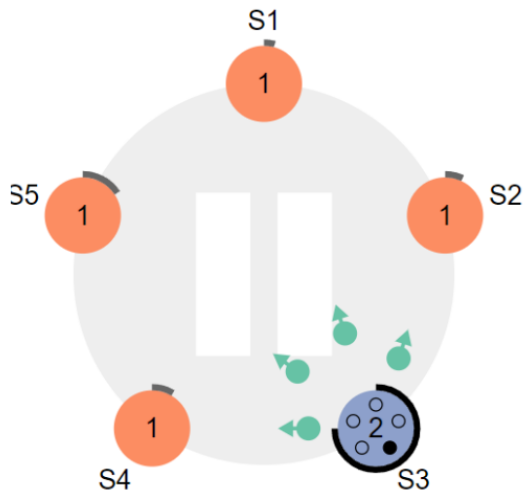
Raft 采用一种心跳机制来触发 leader 选举, 当服务器启动的时候, 都是follow状态. 如果follower在election timeout内没有收到来自leader的心跳(可能没有选出leader, 也可能leader挂了, 或者leader与follower之间网络故障), 则会主动发起选举.



所有节点均为follow状态

步骤如下:

1. 率先超时的节点, 自增当前任期号然后切换为 candidate 状态, 并投自己一票
2. 以并行的方式发送一个 RequestVote RPCs 给集群中的其他服务器节点（企图得到它们的投票）
3. 等待其他节点的回复



S3节点率先超时, 把任期号改为2, 切换为candidate状态, 发起投票请求, 并给自己投一票

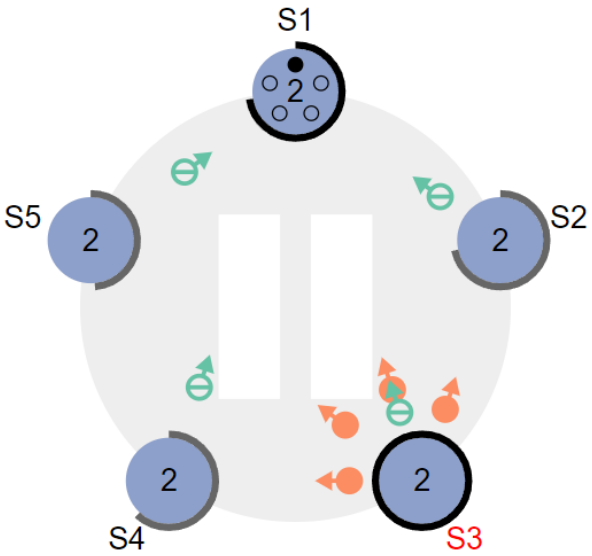
在这个过程中, 可能出现三种结果

a. 赢得选举, 成为Leader(包括自己的一票)
b. 其他节点赢得了选举, 它自行切换到follower
c. 一段时间内没有收到majority投票, 保持candidate状态, 重新发出选举

> 投票要求:
>
> - 每一个服务器节点会按照 先来先服务原则(first-come-first-served)只投给一个 candidate.
> - 候选人知道的信息不能比自己的少

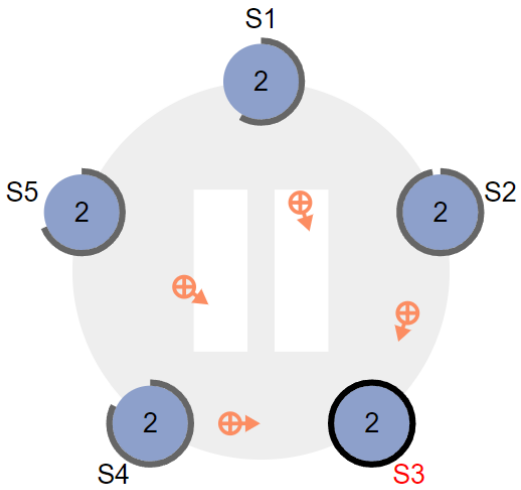
接下来对这三种情况进行说明:

第一种情况: 赢得了选举之后, 新的leader会立刻给所有节点发消息, 广而告之, 避免其余节点触发新的选举.



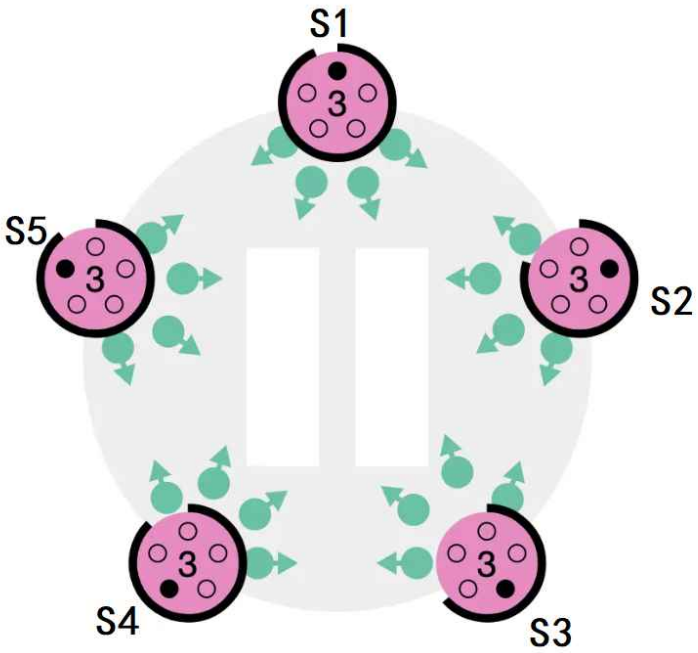
S3节点赢得多数投票, 广而告之

第二种情况: 比如有三个节点A B C, A B同时发起选举, 而A的选举消息先到达C, C给A投了一票, 当B的消息到达C时, 已经不能满足上面提到的第一个约束, 即C不会给B投票, 这时候A就胜出了. A胜出之后, 会给B,C发心跳消息, 节点B发现节点A的term不低于自己的term, 知道有已经有Leader了, 于是把自己转换成follower.



S1 收到 S3的心跳消息, 发现S3的term不低于自己, 知道有leader了, 把自己切换为follower

第三种情况: 没有任何节点获得majority投票. 比如所有的 follower 同时变成 candidate, 然后它们都将票投给自己, 那这样就没有 candidate 能得到超过半数的投票了. 当这种情况发生的时候, 每个 candidate 都会进行一次超时响应, 然后通过自增任期号来开启一轮新的选举, 并启动另一轮的 RequestVote RPCs. 如果没有额外的措施, 这种无结果的投票可能会无限重复下去.



S1-S5 都在参与选举
每个节点都投给自己
如果超时, 会重复发起新一轮的选举

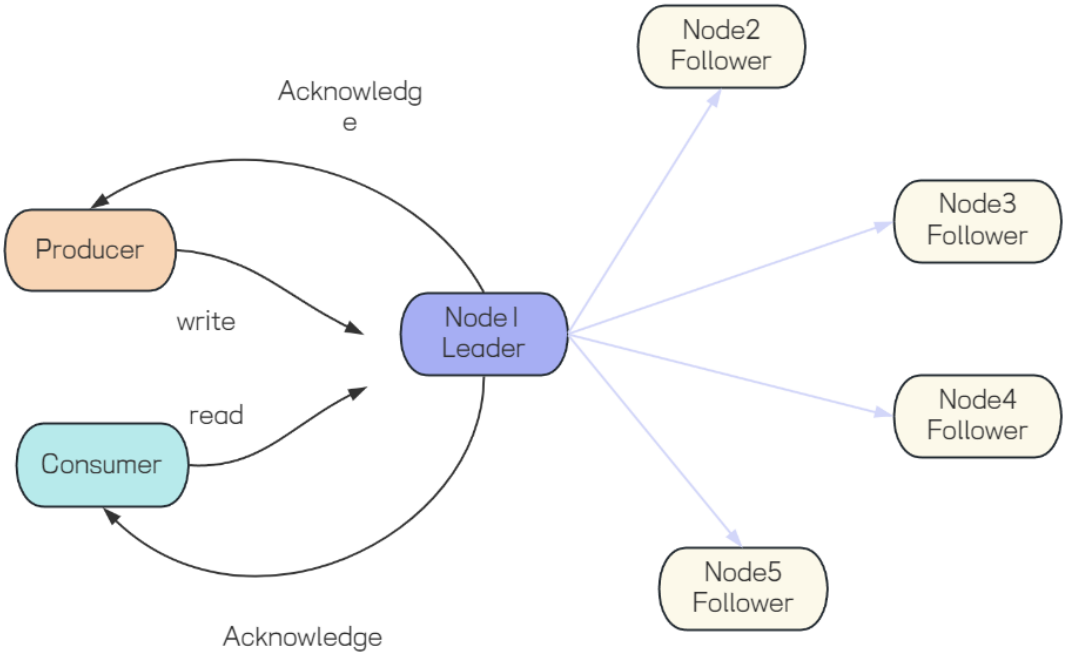
为了解决上述问题，Raft 采用 随机选举超时时间（randomized election timeouts） 来确保很少产生无结果的投票，并且就算发生了也能很快地解决。为了防止选票一开始就被瓜分，选举超时时间是从一个固定的区间（比如，150-300ms）中随机选择。这样可以把服务器分散开来以确保在大多数情况下会只有一个服务器率先结束超时，那么这个时候，它就可以赢得选举并在其他服务器结束超时之前发送心跳。

Raft 动画演示在线地址: https://raft.github.io/

Raft 协议下的消息复制

每个仲裁队列都有多个副本, 它包含一个主和多个从副本. replication factor 为 5的仲裁队列将会有 1个主副本和 4 个从副本. 每个副本都在不同的 RabbitMQ 节点上

客户端(生产者和消费者)只会与主副本进行交互, 主副本再将这些命令复制到从副本. 当主副本所在的节点下线, 其中一个从副本会被选举成为主副本, 继续提供服务.



消息复制和主副本选举的操作, 需要超过半数的副本同意. 当生产者发送一条消息, 需要超过半数的队列副本都将消息写入磁盘以后才会向生产者进行确认, 这意味着少部分比较慢的副本不会影响整个队列的性能.

仲裁队列的使用

1. **创建仲裁队列**

下面讲述三种创建方式

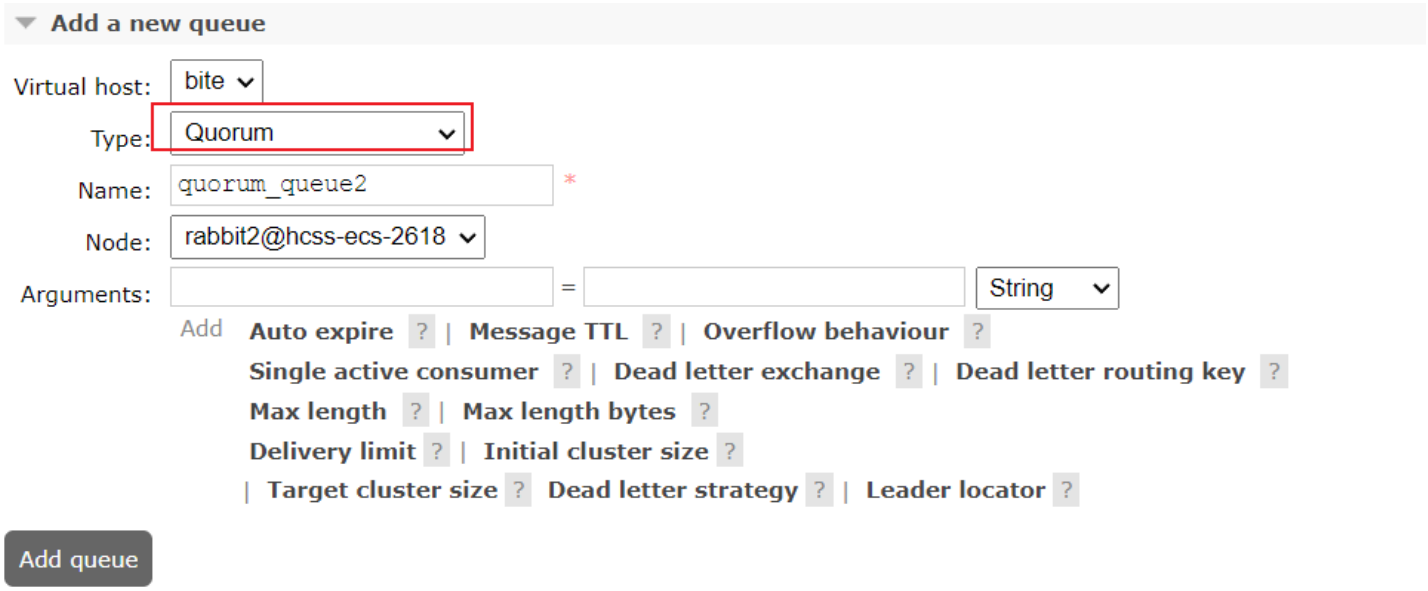
1) 使用Spring框架代码创建

```
@Bean("quorumQueue")
public Queue quorumQueue() {
    return QueueBuilder.durable("quorum_queue").quorum().build();
}
```

2) 使用amqp-client创建

```
Map<String, Object> param = new HashMap<>();
param.put("x-queue-type", "quorum");
channel.queueDeclare("quorum_queue",true,false,false,param);
```

3) 使用管理平台创建



创建时选择Type为Quorum, 指定主副本

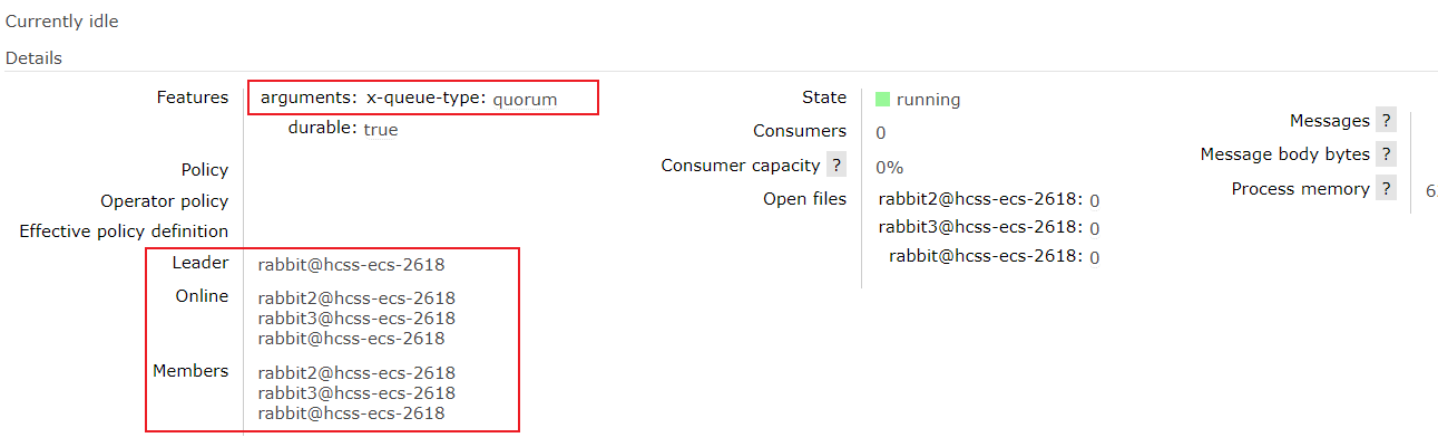
2. 创建后观察管理平台

Overview						Messages			Message rates		
Virtual host	**Name**	**Node**	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**
bite	**ack_queue**	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0			
bite	**confirm_queue**	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0			
bite	**dlx_queue**	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0			
bite	**normal_queue**	rabbit@hcss-ecs-2618	classic	D TTL Lim DLX DLK Args	running	0	0	0			
bite	**qos_queue**	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0			
bite	**quorum_queue**	rabbit@hcss-ecs-2618 +2	quorum	D Args	running	0	0	0			
bite	**quorum_queue2**	rabbit2@hcss-ecs-2618 +2	quorum	D Args	running	0	0	0			
bite	**retry_queue**	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0			
bite	**ttl_queue**	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0			
bite	**ttl_queue2**	rabbit@hcss-ecs-2618	classic	D TTL Args	running	0	0	0			

可以看到, 仲裁队列后面有一个 `+ 2` 字样, 代表这个队列有2个镜像节点.

> 仲裁队列默认的镜像数为5, 即一个主节点, 4个从副本节点.
>
> 如果集群中节点数量少于 5, 比如我们搭建了3个节点的集群, 那么创建的仲裁队列就是 1 主 2 从.
>
> 如果集群中的节点数大于 5 个的话, 那么就只会在 5 个节点中创建出 1主 4 从.

点进去, 可以看到队列详情



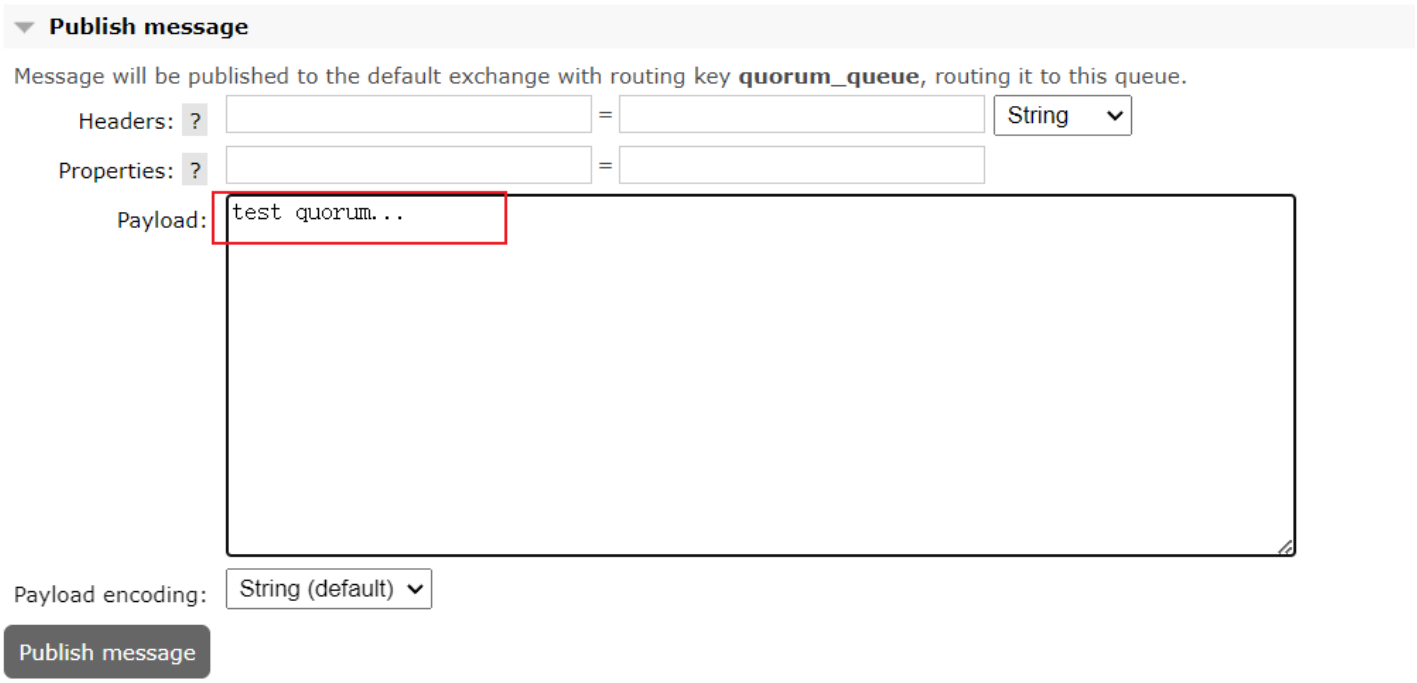
可以看到: 当有多个仲裁队列时, 主副本和从副本会分布在集群的不同节点上, 每个节点可以承载多个主副本和从副本.

3. **接收/发送消息**

仲裁队列发送接收消息和普通队列操作一样

宕机演示

1. 给仲裁队列 `quorum_queue` 发送消息



发送消息后:

Overview						Messages			Message rates			+/-
Virtual host	Name	Node	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
bite	ack_queue	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0				
bite	confirm_queue	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0				
bite	dlx_queue	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0				
bite	normal_queue	rabbit@hcss-ecs-2618	classic	D TTL Lim DLX DLK Args	running	0	0	0				
bite	qos_queue	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0				
bite	quorum_queue	rabbit@hcss-ecs-2618 +2	quorum	D Args	running	1	0	1	0.00/s			
bite	quorum_queue2	rabbit2@hcss-ecs-2618 +2	quorum	D Args	running	0	0	0				
bite	retry_queue	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0				
bite	ttl_queue	rabbit@hcss-ecs-2618	classic	D Args	running	0	0	0				
bite	ttl_queue2	rabbit@hcss-ecs-2618	classic	D TTL Args	running	0	0	0				

2. 停掉队列主副本所在的节点

quorum_queue 队列主副本所在的节点在 rabbit@hcss-ecs-2618, 停掉这台机器

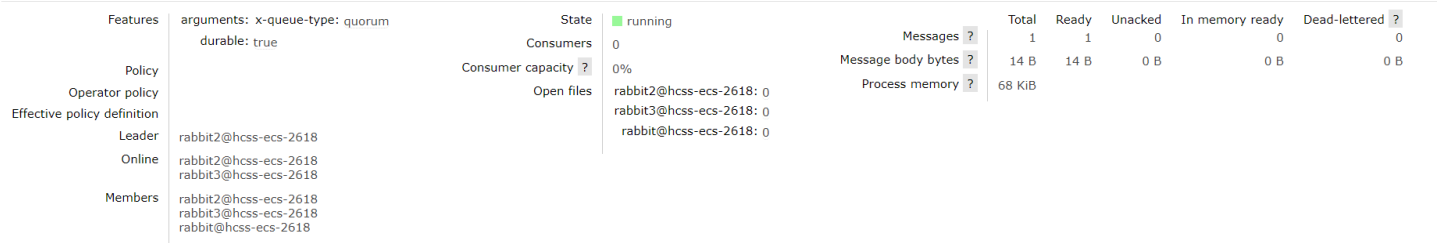
```
root@hcss-ecs-2618:~# rabbitmqctl -n rabbit stop_app   #rabbit为节点名称
Stopping rabbit application on node rabbit@hcss-ecs-2618 ...
root@hcss-ecs-2618:~#
```

观察其他节点, 可以看到quorum_queue 队列的内容依然存在.

并且, 因为主副本所在节点宕机了, quorum_queue 主副本从rabbit@hcss-ecs-2618 转移到了 rabbit2@hcss-ecs-2618

Overview						Messages			Message rates			+/-
Virtual host	**Name**	**Node**	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**	
bite	**ack_queue**	rabbit@hcss-ecs-2618	classic	D Args	down	?	?	?				
bite	**confirm_queue**	rabbit@hcss-ecs-2618	classic	D Args	down	?	?	?				
bite	**dlx_queue**	rabbit@hcss-ecs-2618	classic	D Args	down	?	?	?				
bite	**normal_queue**	rabbit@hcss-ecs-2618	classic	D TTL Lim DLX DLK Args	down	?	?	?				
bite	**qos_queue**	rabbit@hcss-ecs-2618	classic	D Args	down	?	?	?				
bite	**quorum_queue**	rabbit2@hcss-ecs-2618 +1	quorum	D Args	running	1	0	1				
bite	**quorum_queue2**	rabbit2@hcss-ecs-2618 +1	quorum	D Args	running	0	0	0				
bite	**retry_queue**	rabbit@hcss-ecs-2618	classic	D Args	down	?	?	?				
bite	**ttl_queue**	rabbit@hcss-ecs-2618	classic	D Args	down	?	?	?				
bite	**ttl_queue2**	rabbit@hcss-ecs-2618	classic	D TTL Args	down	?	?	?				

队列详细信息: 只剩下两个成员了



仲裁队列, 是RabbitMQ从3.8.0版本引入的新的队列类型, Quorum相比Classic在分布式环境下对消息的可靠性保障更高. 普通队列只会存放在集群中的一个节点上, 虽然通过其它节点也可以访问普通队列, 但是其它节点只是把请求转发到队列所在的节点进行操作. 一旦队列所在节点宕机, 队列中的消息就会丢失, 因此普通集群只是提高了并发能力, 并未实现高可用. 仲裁队列可以极大的保障 RabbitMQ 集群对接的高可用.

5. HAProxy 负载均衡

面对大量业务访问、高并发请求,可以使用高性能的服务器来提升RabbitMQ服务的负载能力. 当单机容量达到极限时, 可以采取集群的策略来对负载能力做进一步的提升, 但这里还存在一些问题.

试想如果一个集群中有3个节点, 我们在写代码时, 访问哪个节点呢?

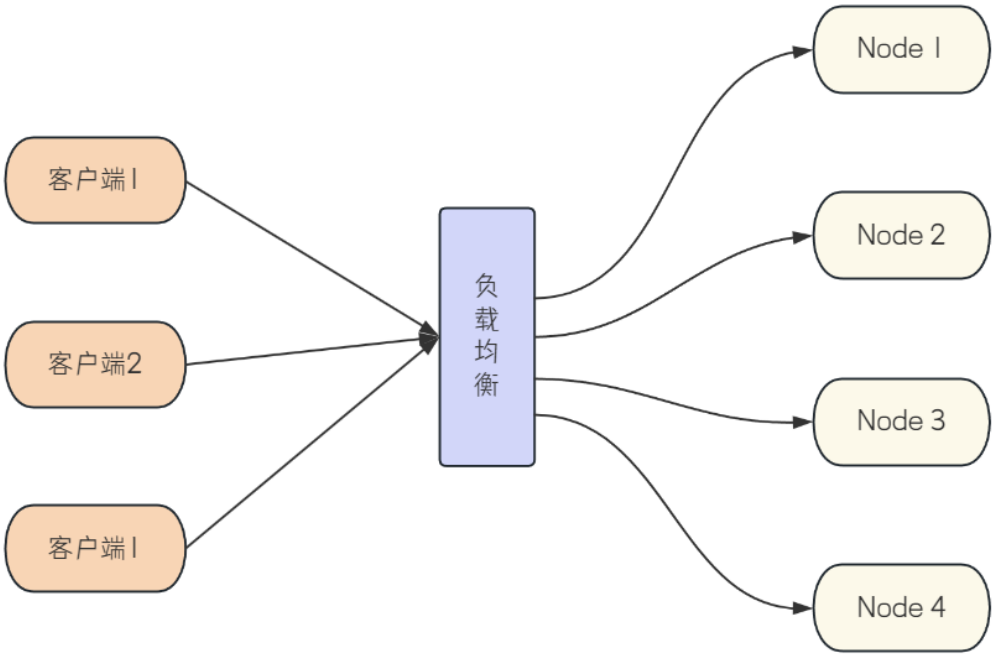
答案是访问任何一个节点都可以.

这时候就存在两个问题:

1. 如果我们访问的是node1, 但是node1挂了, 咱们的程序也会出现问题, 所以最好是有一个统一的入口, 一个节点故障时, 流量可以及时转移到其他节点.
2. 如果所有的客户端都与node1建议连接, 那么node1的网络负载必然会大大增加, 而其他节点又由于没有那么多的负载而造成硬件资源的浪费.

这时候负载均衡显得尤其重要.

引入负载均衡之后, 各个客户端的连接可以通过负载均衡分摊到集群的各个节点之中, 从而避免前面的问题.



这里主要讨论的是如何有效地对RabbitMQ集群使用软件负载均衡技术,目前主流的方式有在客户端内部实现负载均衡,或者使用HAProxy、LVS等负载均衡有软件来实现. 咱们这里讲一下使用HAProxy来实现负载均衡.

5.1 安装

接下来我们来安装HAProxy, 实现负载均衡.

HAProxy（High Availability Proxy）是一个开源的负载均衡器和TCP/HTTP应用程序的代理服务器, 它被设计用来提供高可用性, 负载均衡和代理功能. HAProxy主要用于分发网络流量到多个后端服务器, 以提高网络的可靠性和性能.

安装参考: [**HAProxy安装.pdf**]

> 如果HAProxy主机突然宕机或者网卡失效,那么虽然RabbitMQ集群没有任何故障, 但是对于外界的客户端来说所有的连接都会被断开,结果将是灾难性的. 确保负载均衡服务的可靠性同样显得十分重要. 通常情况下, 会使用Keepalived 等高可用解决方案对haproxy做主备, 在HAProxy主节点故障时自动将流量转移到备用节点

5.2 使用

引入HAProxy之后, RabbitMQ的集群使用和单机使用方式一样, 只不过需要把RabbitMQ的IP和port改为HAProxy的IP和port

1. **修改配置文件**

```
spring:
  rabbitmq:
    addresses: amqp://study:study@124.71.229.73:5670/bite
```

2. **声明队列 test_cluster**

```
public static final String CLUSTER_QUEUE = "cluster_queue";
```

```
@Configuration
public class ClusterConfig {
    @Bean("clusterQueue")
    public Queue clusterQueue() {
        return QueueBuilder.durable(Constant.CLUSTER_QUEUE).quorum().build();
    }
}
```

3. **发送消息**

```
@RequestMapping("/cluster")
public String cluster() {
    rabbitTemplate.convertAndSend("", Constant.CLUSTER_QUEUE, "quorum test...");
    return "发送成功!";
}
```

> 或者使用amqp客户端发消息

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

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.TimeoutException;

public class ClusterProducer {
    private static final String QUEUE_NAME = "hello_world";

```

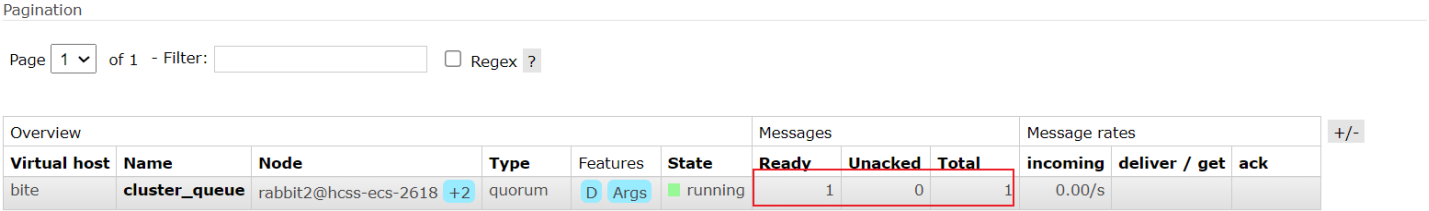
```
13      public static void main(String[] args) throws IOException,
    TimeoutException {
14          // 1. 创建连接工厂
15          ConnectionFactory factory = new ConnectionFactory();
16          //2. 设置参数
17          factory.setHost("124.71.229.73");//HAProxy
18          factory.setPort(5670);//HAProxy port
19          factory.setVirtualHost("bite");//虚拟机名称，默认/
20          factory.setUsername("study");//用户名,默认guest
21          factory.setPassword("study");//密码，默认guest
22          //3. 创建连接Connection
23          Connection connection = factory.newConnection();
24          //4. 创建channel通道
25          Channel channel = connection.createChannel();
26          //5. 声明队列
27          Map<String, Object> param = new HashMap<>();
28          param.put("x-queue-type", "quorum");
29          channel.queueDeclare("test_cluster",true,false,false,param);
30          //6. 通过channel发送消息到队列中
31          String msg = "hello cluster~~";
32          //简单模式下，使用的是默认交换机，使用默认交换机时，routingKey要个队列名称一
    样，才可以路由到对应的队列上去
33          channel.basicPublish("","test_cluster",null,msg.getBytes());
34          //7.释放资源
35          System.out.println(msg+"消息发送成功");
36          channel.close();
37          connection.close();
38      }
39  }
```

4. 测试

发送消息到 cluster_queue

发送成功!

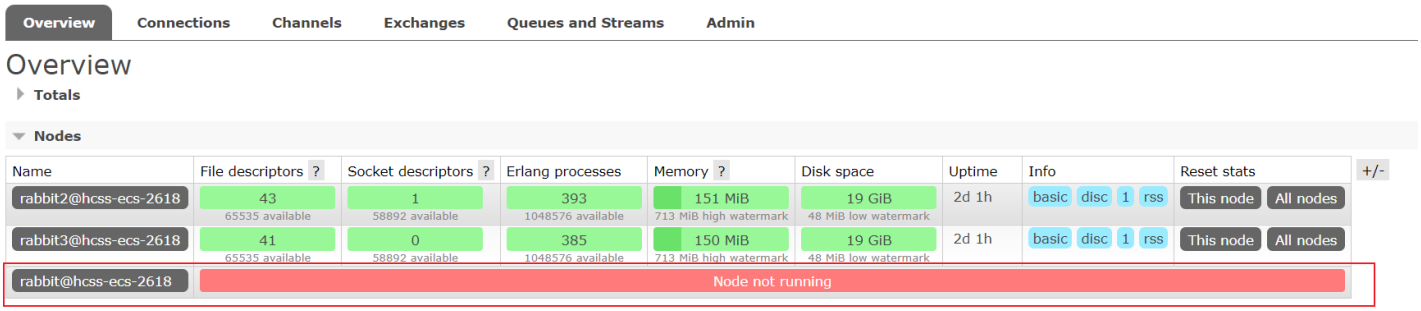
可以看到IP和端口号改成HAProxy的IP和端口之后, 消息依然可以发送成功



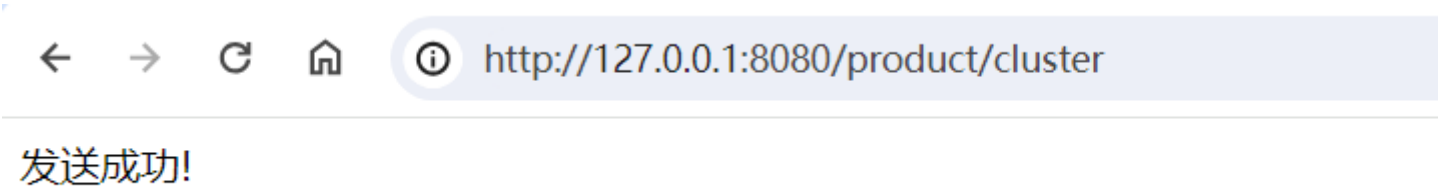
5. 宕机演示

我们停止其中一个节点，继续测试步骤2的代码

```
1 rabbitmqctl -n rabbit stop_app
```



在节点rabbit宕机的情况下，继续发送消息



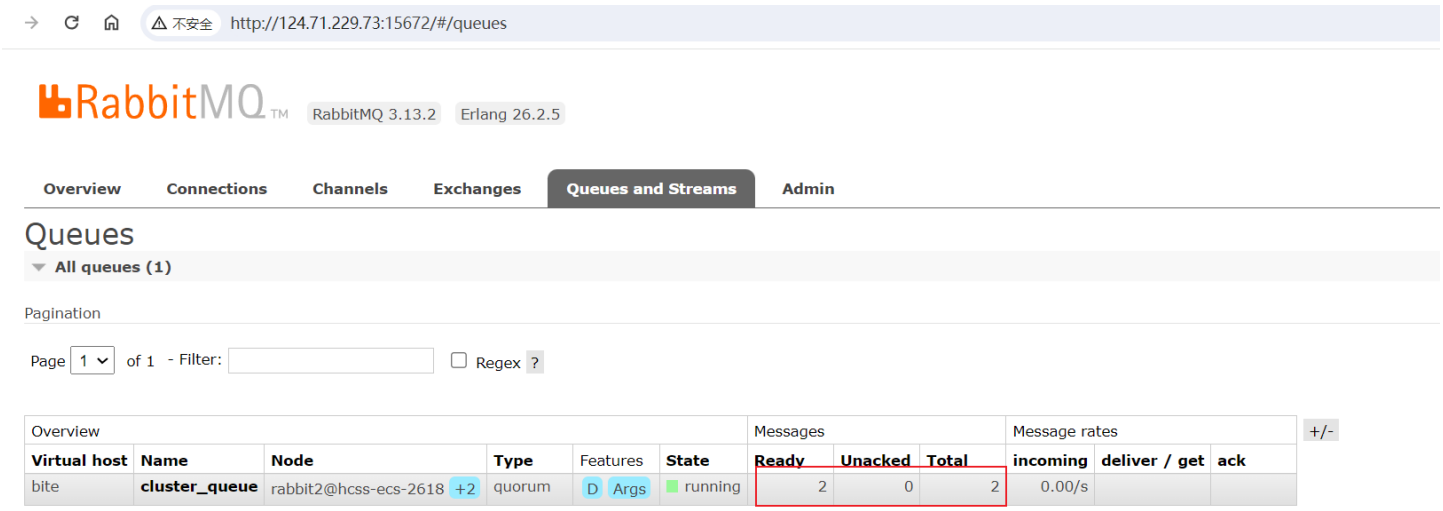
可以看到消息发送成功了，我们看看界面上，显示队列中有两条数据

6. 集群恢复

恢复上述节点

```
1 rabbitmqctl -n rabbit start_app
```

观察消息也同步到当前节点了


不安全
http://124.71.229.73:15672/#/queues
RabbitMQ
RabbitMQ 3.13.2
Erlang 26.2.5
Overview
Connections
Channels
Exchanges
Queues and Streams
Admin
Queues
All queues (1)
Pagination
Page 1 of 1 - Filter:
Regex ?
Overview
Messages
Message rates
+/-
Virtual host
Name
Node
Type
Features
State
Ready
Unacked
Total
incoming
deliver / get
ack
bite
cluster_queue
rabbit2@hcss-ecs-2618 +2
quorum
D Args
running
2
0
2
0.00/s