# RabbitMQ HA初步分析

## 普通MQ的结构

**X**

Amqqueue

Backing queue

binding

MQ

in

out

Rabbitmq大体上可以分为两部分（Exchange和MQ），所有发送给RabbitMQ的消息都会先交给Exchange， Exchange的功能类似于路由器，它会根据自身类型（fanout、direct、topic）以及binding信息决定一个消息该被放到哪一个MQ， 而MQ的功能在于暂时存储消息，并将MQ中的消息以订阅或者poll的方式交给接收方。

MQ内部大致又可以分为两部分:amqueue和backing queue, amqqueue负责实现amqp协议规定的mq的基本逻辑，backing queue则实现消息的存储，它会尽量为durable=true的消息做持久化的存储，而在内存不足时将一部分消息放入DISK换取更多的内存空间。Backing queu内部又细分为5各小Q，

Q1

Q2

delta

Q3

Q4

消息在这些Q中传递的“一般”过程q1->q2->delta->q3->q4，这几个Q实现的是

“RAM ->DISK->RAM”这一过程中对消息的分类管理。大多数情况下，一个消息并非需要走完每个小Q，通常大部分都可以略过。与这5各Q对应，在backing queue中消息的生命周期可以分为四个状态：

Alpha：该消息的位置信息和消息本身都在RAM中，这类消息排列在Q1和Q4。

Beta：消息的位置保存在RAM中，消息本身保存在DISK中，这类消息排列在Q2或Q3中。

Gamma: 消息的位置保存RAM和DISK中，消息本身保存在DISK中，这类消息排列在Q2或Q3中。

Delta：消息的位置和消息本身都保存在DISK中，这类消息排列在delta中。

从Q1->Q2->delta这一个过程是将消息逐步从RAM移动到DISK的过程，而delta->Q3->Q4是从DISK逐步移动到RAM的过程。

通常在负载正常时，一个消息不会经历每种状态，如果消息被消费的速度不小于接收新消息的速度，对于不需要保证可靠不丢的消息极可能只会有Alpha状态。对于durable=true的消息，它一定会进入gamma，若开启publish confirm，只有到了这个阶段才会确认该消息已经被接收，若消息消费的速度足够快，内存也充足，这些消息也不会继续走到下一状态。

从上述backing queue对消息的处理过程可以看出，消息若能尽早被消费掉即在不要走完这5个队列，尽量在q1或q2中就被消费掉，就能减少系统的开销。若走的“太深”则会有内存的换入换出增加系统开销。这样就存在一个问题：

通常在系统负载较高时，已接收到的消息若不能很快的被消费掉，这些消息就会进入到很深的队列中去，增加处理每个消息的平均开销。因为要花更多的时间和资源处理“积压”的消息，所以用于处理新来的消息的能力就会降低，使得后来的消息又被积压进入很深的队列，继续加大处理每个消息的平均开销，这样情况就会越来越恶化，使得系统的处理能力大大降低。

根据官方博客，应对这一问题，有三个措施：

1. 进行流量控制。
2. 增加prefetch的值，即一次发送多个消息给接收者，加快消息被消费掉的速度。
3. 才有mutli ack，降低处理ack带来的开销。

目前RabbitMQ已经有了很好的流量控制机制，通过前面多次的测试，在测试过程中观察到的现象是，MQ中堆积的消息数一直都很少（低于5个）。需要使用者做的就是2，3两点。

## 镜像MQ的结构

mirror queue基本上就是一个特殊的backing queue， 它内部包裹了一个普通的backing queue做本地的消息持久化处理，在此基础上增加了将消息和ack复制到所有镜像的功能。所有对rabbit\_mirror\_queue\_master的操作，会通过组播GM（Guarenteed Multicast）的方式同步到各slave节点。

### 新节点加入

允许新的slave节点中途加入到集群中，新加入的slave节点并不同步master节点的所有在该slave加入之前存在的消息，只对新来的消息保持同步，随着旧的消息被消费，经过一段时间后，slave节点就会与master节点完全同步。

### 节点失效

当master节点失效后，所有slave中消息队列最长者会成为新的master，因为这样的节点最有可能与原来的master节点完全同步。

### 节点重启

当一个节点无论master还是slave失效后重启，都会丢弃本地记录在disk中的所有消息，作为一个全新的slave节点加入到集群中去。

### GM

GM模块实现的一种可靠的组播通讯协议，该协议能够保证组播消息的原子性，即保证组中活着的节点要么都收到消息要么都收不到。它的实现大致如下：

一个组的所有成员组成一个ring，例如 A->B->C->D->A。假如A是master节点，A要发组播消息，A首先会将消息发送到A的后继节点B，B接收到消息后在传递给C然后是D，最后D再发给A。在此过程中若有节点失效，发送节点就会往失效的节点的后继节点发消息，若后继节点也失效就往后继的后继发消息。当A收到传回来的消息时，A就可以确认所有“活着的”节点都已收到该消息，但此时B、C、D并不能确认所有节点都收到了该消息，所以不能往上提交该消息。这时，A往B发Ack，当B收到ack后就能确认所有的节点都收到该消息，B将该ack继续传递给C，D最终又传回A，至此整个发送过程就完成了。若最终A没有收到ack，则说明此次发送失败。

## 使用RabbitMQ HA存在的问题及可能的解决方法

1. 在master节点失效时，slave节点可能并未与mater节点完全同步造成消息丢失。

首先定位到rabbit\_mirror\_queue\_master中消息的处理的入口函数

|  |
| --- |
| publish(Msg = #basic\_message { id = MsgId }, MsgProps, ChPid,  State = #state { gm = GM,  seen\_status = SS,  backing\_queue = BQ,  backing\_queue\_state = BQS }) ->  false = dict:is\_key(MsgId, SS), %% ASSERTION  **ok = gm:broadcast(GM, {publish, false, ChPid, MsgProps, Msg}),**  **BQS1 = BQ:publish(Msg, MsgProps, ChPid, BQS),**  ensure\_monitoring(ChPid, State #state { backing\_queue\_state = BQS1 }). |

第一句Gm:broadcast做的是将新消息组播到所有slave节点，第二句BQ:publish的功能是做本地持久化的处理，这里BQ是普通的BQ。对于durable=true的消息，只有当消息持久化成功后即进入gamma阶段才会向sender发确认。所以新消息的处理过程大致可以归纳为：

1. 向所有slave节点发送该消息
2. 持久化
3. 发送ack给sender

根据gm.erl中对broadcast的描述“This is a cast and the function call will return immediately. There is no guarantee that the message will reach any member of the group.”

该函数是异步的，所以就有可能出现 slave并未接收到该消息，此时又像sender发了确认，接着master失效，这条消息就丢失了。

解决这个问题的可以采用同步组播方式向所有slave节点发消息，即确认所有的slave都已收到了该消息才向sender确认。目前gm.erl代码中已经提供了同步发消息的函数gm:confirmed\_broadcast/2。从前面对gm算法的描述可以看出，这样做的坏处是会增加处理延时，slave越多，时延越大。

1. 在mater节点失效时， ack消息可能并未同步到所有节点，造成消息的重复发送。

为保证消息不丢，接收者在收到消息后，需要向MQ回ack，MQ收到ack后才会删除该消息。但由于ack的发送是异步的，将该ack消息组播到每个slave节点也是异步的，所以在mater节点失效时， ack消息可能并未同步到所有节点，造成消息的重复发送。

为解决这一问题，初步有两种思路：第一种，去重。第二种，更改接收者回ack的逻辑，改为同步的发送，保证所有的节点都以收到该ack，才算ack成功。