RabbitMQ 镜像队列

笔记本： RabbitMq源代码阅读系统原理分析
创建时… 2015/8/13 15:35 **更新时…** 2016/1/18 18:27

一．镜像队列的设置(通过RabbitMQ系统的rabbit_policy模块实现)

镜像队列的配置通过添加policy完成，policy添加的命令为：

rabbitmqctl set_policy [-p Vhost] Name Pattern Definition [Priority]

-p Vhost: 可选参数，针对指定vhost下的queue进行设置

Name: policy的名称

Pattern: queue的匹配模式（正则表达式）

Definition: 镜像定义，包括三个部分 ha-mode，ha-params，ha-sync-mode

ha-mode: 指明镜像队列的模式，有效值为 all/exactly/nodes

all表示在集群所有的节点上进行镜像

exactly表示在指定个数的节点上进行镜像，节点的个数由ha-params指定

nodes表示在指定的节点上进行镜像，节点名称通过ha-params指定

ha-params: ha-mode模式需要用到的参数

ha-sync-mode: 镜像队列中消息的同步方式，有效值为automatic，manually

Priority: 可选参数， policy的优先级

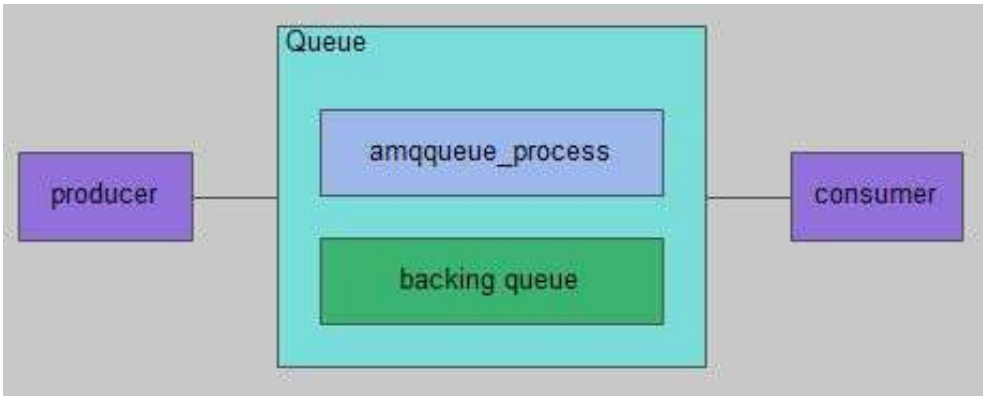
例如，对队列名称以hello开头的所有队列进行镜像，并在集群的两个节点上完成镜像，policy的设置命令为：

```
rabbitmqctl set_policy hello-ha "^hello" '{"ha-mode":"exactly","ha-params":2,"ha-sync-mode":"automatic"}'
```

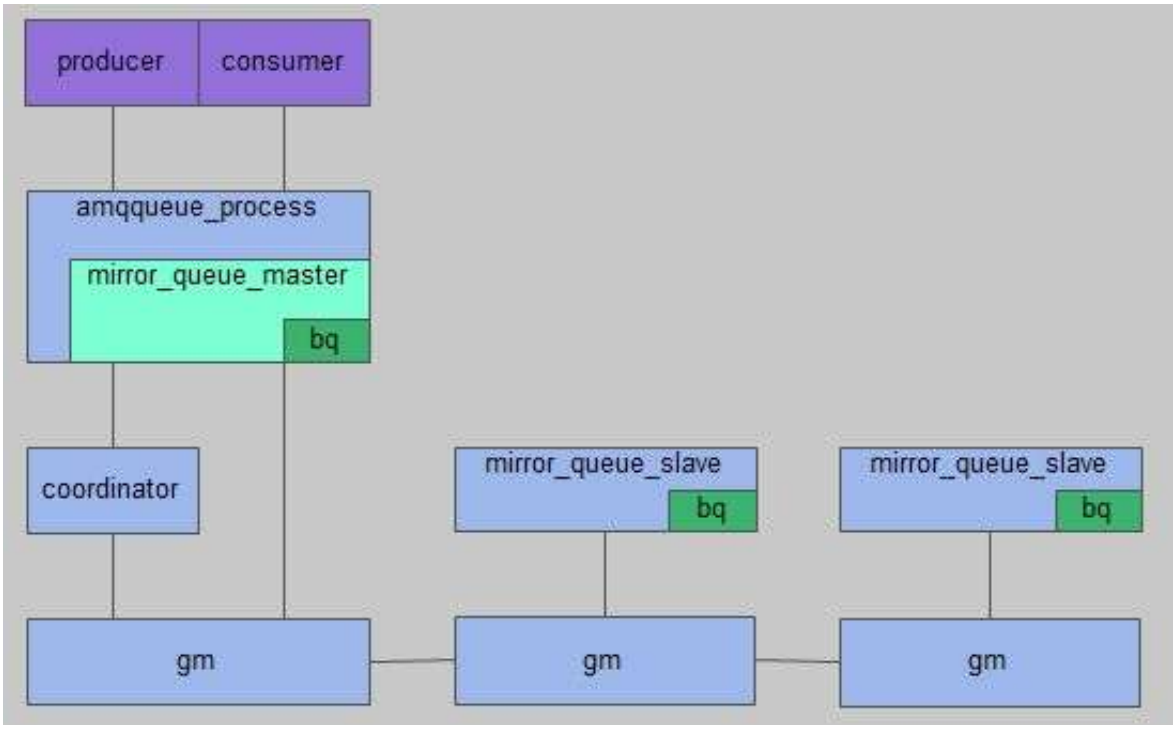
二．镜像队列的实现原理

(1) 整体介绍

通常队列由两部分组成：一部分是amqqueue_process，负责协议相关的消息处理，即接收生产者发布的消息、向消费者投递消息、处理消息confirm、acknowledge等等；另一部分是backing_queue，它提供了相关的接口供amqqueue_process调用，完成消息的存储以及可能的持久化工作等。



镜像队列同样由这两部分组成，amqqueue_process仍旧进行协议相关的消息处理，backing_queue则是由master节点和slave节点组成的一个特殊的backing_queue。master节点和slave节点都由一组进程组成，一个负责消息广播的gm，一个负责对gm收到的广播消息进行回调处理。在master节点上回调处理是coordinator(协调员)，在slave节点上则是mirror_queue_slave。mirror_queue_slave中包含了普通的backing_queue进行消息的存储，master节点中backing_queue包含在mirror_queue_master中由amqqueue_process进行调用。



注意：消息的发布与消费都是通过master节点完成。master节点对消息进行处理的同时将消息的处理动作通过gm广播给所有的slave节点，slave节点的gm收到消息后，通过回调交由mirror_queue_slave进行实际的处理。

(2) gm(Guaranteed Multicast)

传统的主从复制方式：由master节点负责向所有slave节点发送需要复制的消息，在复制过程中，如果有slave节点出现异常，master节点需要作出相应的处理；如果是master节点本身出现问题，那么slave节点间可能会进行通信决定本次复制是否继续。当然为了处理各种异常情况，整个过程中的日志记录是免不了的。

然而rabbitmq中并没有采用这种方式，而是将所有的节点形成一个循环链表，每个节点都会监控位于自己左右两边的节点，当有节点新增时，相邻的节点保证当前广播的消息会复制到新的节点上；当有节点失效时，相邻的节点会接管保证本次广播的消息会复制到所有节点。

在master节点和slave节点上的这些gm形成一个group，group的信息会记录在mnesia中。不同的镜像队列形成不同的group。

消息从master节点对应的gm发出后，顺着链表依次传送到所有节点，由于所有节点组成一个循环链表，master节点对应的gm最终会收到自己发送的消息，这个时候master节点就知道消息已经复制到所有slave节点了。

(3) 重要的表结构

rabbit_queue表记录队列的相关信息：

```
-record(q, {
                  q,                              %% 队列信息数据结构amqqueue
                  exclusive_consumer,             %% 当前队列的独有消费者
                  has_had_consumers,              %% 当前队列中是否有消费者的标识
                  backing_queue,                  %% backing_queue对应的模块名字
                  backing_queue_state,            %% backing_queue对应的状态结构
                  consumers,                      %% 消费者存储的优先级队列
                  expires,                        %% 当前队列未使用就删除自己的时
间
                  sync_timer_ref,                 %% 同步confirm的定时器，当前队
列大部分接收一次消息就要确保当前定时器的存在(200ms的定时器)
                  rate_timer_ref,                 %% 队列中消息进入和出去的速率定
时器
                  expiry_timer_ref,               %% 队列中未使用就删除自己的定时
器
                  stats_timer,                    %% 向rabbit_event发布信息的数据
结构状态字段
                  msg_id_to_channel,              %% 当前队列进程中等待confirm的
消息gb_trees结构，里面的结构是Key:MsgId Value:{SenderPid, MsgSeqNo}
                  ttl,                            %% 队列中设置的消息存在的时间
                  ttl_timer_ref,                  %% 队列中消息存在的定时器
                  ttl_timer_expiry,               %% 当前队列头部消息的过期时间点
                  senders,                        %% 向当前队列发送消息的rabbit_c
hannel进程列表
                  dlx,                            %% 死亡消息要发送的exchange交换
机(通过队列声明的参数或者policy接口来设置)
                  dlx_routing_key,                %% 死亡消息要发送的路由规则(通
过队列声明的参数或者policy接口来设置)
                  max_length,                     %% 当前队列中消息的最大上限(通
过队列声明的参数或者policy接口来设置)
                  max_bytes,                      %% 队列中消息内容占的最大空间
                  args_policy_version,            %% 当前队列中参数设置对应的版本
号，每设置一次都会将版本号加一
                  status                          %% 当前队列的状态
              }).
```

注意：slave_pids的存储是按照slave加入的时间来排序的，以便master节点失效时，提升

为新的master。

gm_group表记录gm形成的group的相关信息：

```
%% 整个镜像队列群组的信息，该信息会存储到Mnesia数据库
-record(gm_group, { name,                    %% group的名称,与queue的名称一致
                    version,                 %% group的版本号，新增节点/节点失效时会递增
                    members                  %% group的成员列表，按照节点组成的链表顺序进行排序
                  }).
```

view_member记录当前GM群组中真实的视图的数据结构：

```
%% 镜像队列群组视图成员数据结构
-record(view_member, { id,                   %% 单个镜像队列(结构是{版本号，该镜像队列的Pid})
                       aliases,              %% 记录id对应的左侧死亡的GM进程列表
                       left,                 %% 当前镜像队列左边的镜像队列(结构是{版本号，该镜像队列的Pid})
                       right                 %% 当前镜像队列右边的镜像队列(结构是{版本号，该镜像队列的Pid})
                     }).
```

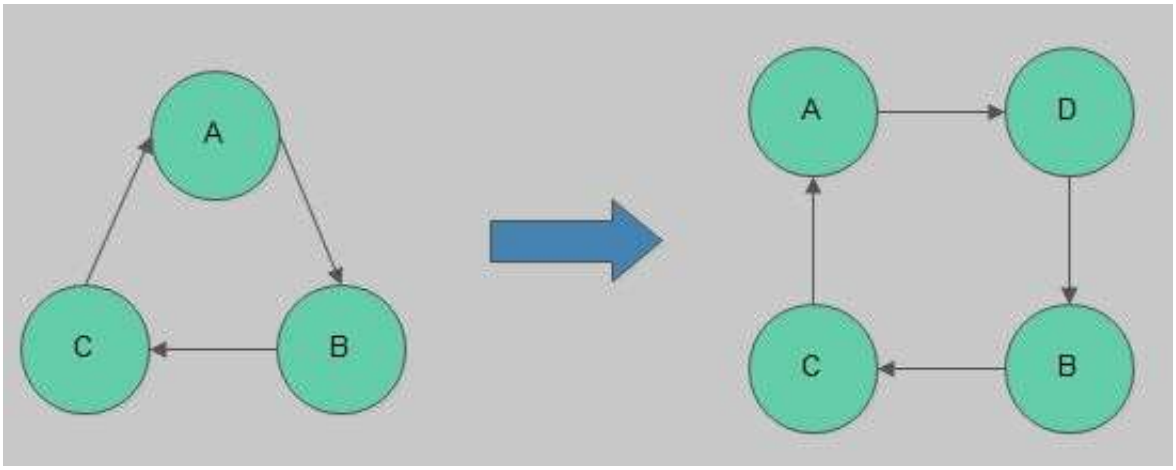
记录单个GM进程中信息的数据结构：

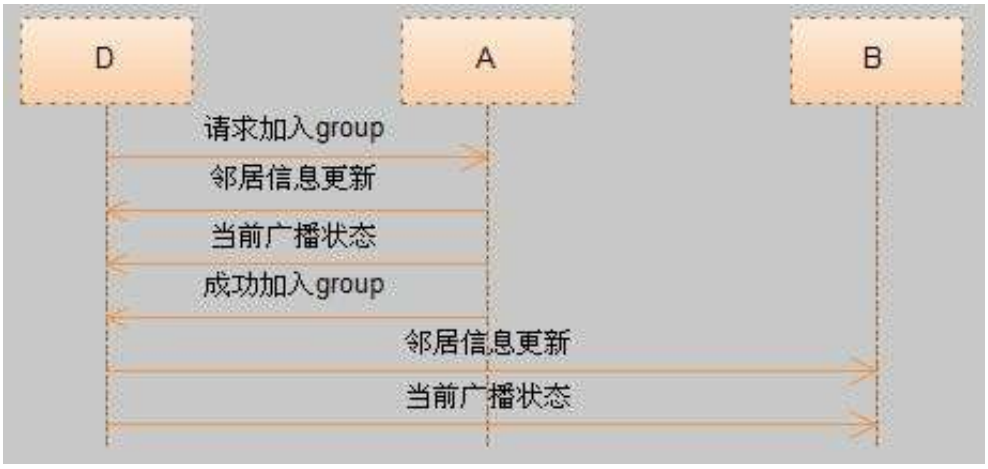
```
%% 记录单个GM进程中信息的数据结构
-record(member, { pending_ack,               %% 待确认的消息,也就是已发布的消息缓存的地方
                  last_pub,                  %% 最后一次发布的消息计数
                  last_ack                   %% 最后一次确认的消息计数
                }).
```

三．镜像队列群组增加删除相关操作

（1）新增节点

slave节点先从gm_group中获取对应group的所有成员信息，然后随机选择一个节点并向这个节点发送请求，这个节点收到请求后，更新gm_group对应的信息，同时通知左右节点更新邻居信息（调整对左右节点的监控）及当前正在广播的消息，然后回复通知请求节点成功加入group。请求加入group的节点收到回复后再更新rabbit_queue中的相关信息，并根据需要进行消息的同步。





D的GM进程请求加入群组的代码：

```
%% 处理将自己加入到镜像队列的群组中的消息
handle_cast(join, State = #state { self          = Self,
                                   group_name    = GroupName,
                                   members_state = undefined,
                                   module        = Module,
                                   callback_args = Args,
                                   txn_executor  = TxnFun }) ->
    %% 将当前镜像队列加入到镜像队列的循环队列视图中，返回最新的镜像队列循环队列视图
    %% 如果当前GM进程不是群组的第一个GM进程，则会call到自己要加入的位置左侧的GM进程，直到左侧成功的将当前GM加入群组，然后返回给
    %% 当前GM进程最新的gm_group数据结构信息
    View = join_group(Self, GroupName, TxnFun),
    MembersState =
        %% 获取镜像队列视图的所有key列表
        case alive_view_members(View) of
            %% 如果是第一个GM进程的启动则初始化成员状态数据结构
            [Self] -> blank_member_state();
            %% 如果是第一个以后的GM进程加入到Group中，则成员状态先不做初始化，让自己左侧的GM进程发送过来的信息进行初始化
```

```
            _       -> undefined
        end,
    %% 检查当前镜像队列的邻居信息(根据消息镜像队列的群组循环视图更新自己最新的
左右两边的镜像队列)
    State1 = check_neighbours(State #state { view          = View,
                                             members_state = MembersState }),
    %% 向启动该GM进程的进程通知他们已经加入镜像队列群组成功(rabbit_mirror_queu
e_coordinator和rabbit_mirror_queue_slave模块回调)
    handle_callback_result(
      {Module:joined(Args, get_pids(all_known_members(View))), State1});
```

A的GM进程处理新的GM进程增加到自己右侧的处理过程代码：

```
%% 同步处理将新的镜像队列加入到本镜像队列的右侧的消息
handle_call({add_on_right, NewMember}, _From,
            State = #state { self          = Self,
                             group_name    = GroupName,
                             members_state = MembersState,
                             txn_executor  = TxnFun }) ->
    %% 记录将新的镜像队列成员加入到镜像队列组中，将新加入的镜像队列写入gm_grou
p结构中的members字段中(有新成员加入群组的时候，则将版本号增加一)
    Group = record_new_member_in_group(NewMember, Self, GroupName, TxnFun),
    %% 根据组成员信息生成新的镜像队列视图数据结构
    View1 = group_to_view(Group),
    %% 删除擦除的成员
    MembersState1 = remove_erased_members(MembersState, View1),
    %% 向新加入的成员即右边成员发送加入成功的消息
    ok = send_right(NewMember, View1,
                    {catchup, Self, prepare_members_state(MembersState1)}),
    %% 根据新的镜像队列循环队列视图和老的视图修改视图，同时根据镜像队列循环视
图更新自己左右邻居信息
    {Result, State1} = change_view(View1, State #state {
                                                        members_state = Member
sState1 }),
    %% 向请求加入的镜像队列发送最新的当前镜像队列的群组信息
    handle_callback_result({Result, {ok, Group}, State1}).
```

D处的GM进程处理A处的GM进程发送过来的A处GM进程当前尚未处理完的消息：

```
%% 左侧的GM进程通知右侧的GM进程最新的成员状态(此情况是本GM进程是新加入Group的
，等待左侧GM进程发送过来的消息进行初始化成员状态)
handle_msg({catchup, Left, MembersStateLeft},
           State = #state { self          = Self,
                            left          = {Left, _MRefL},
                            right         = {Right, _MRefR},
                            view          = View,
                            %% 新加入的GM进程在加入后是没有初始化成员状态，是
等待左侧玩家发送消息来进行初始化
                            members_state = undefined }) ->
    %% 异步向自己右侧的镜像队列发送最新的所有成员信息，让Group中的所有成员更新
成员信息
    ok = send_right(Right, View, {catchup, Self, MembersStateLeft}),
    %% 将成员信息转化成字典数据结构
```

```
    MembersStateLeft1 = build_members_state(MembersStateLeft),
    %% 新增加的GM进程更新最新的成员信息
    {ok, State #state { members_state = MembersStateLeft1 }};
```

D处的GM进程再接收到的消息广播给自己右侧的GM进程，让它处理最新的消息：

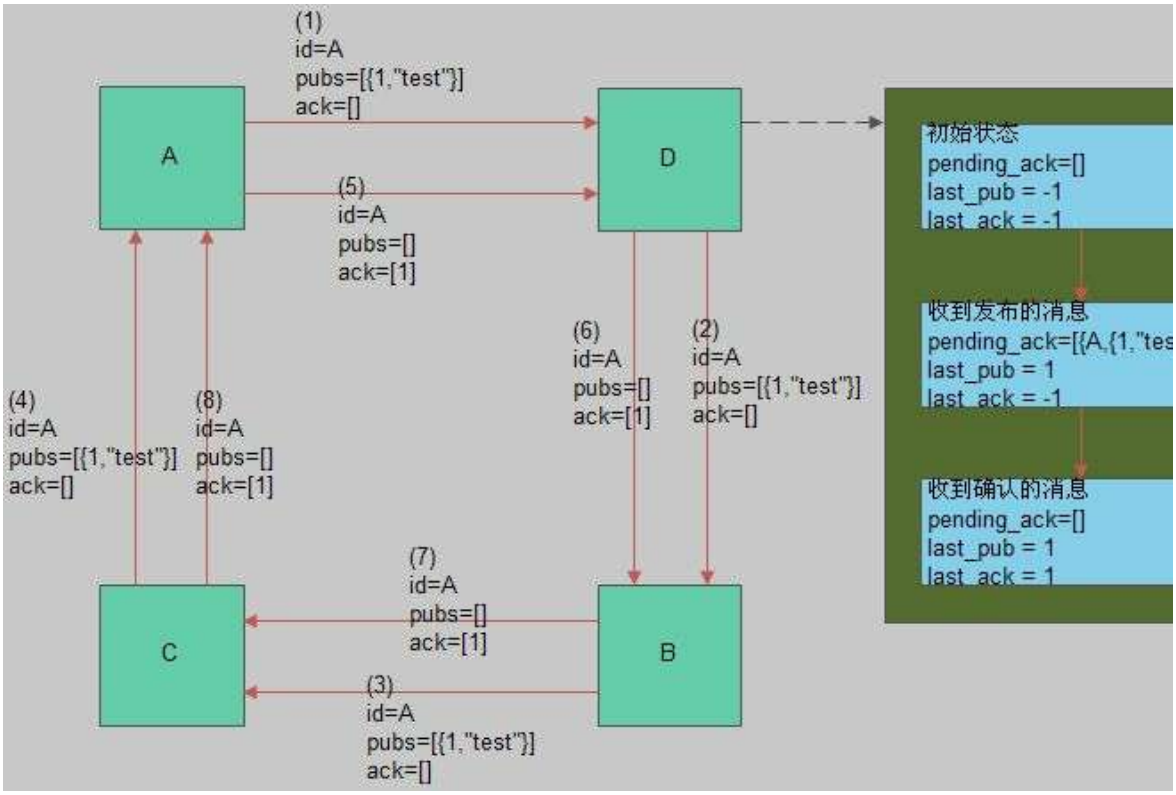
```
%% 左侧的GM进程通知右侧的GM进程最新的成员状态(此情况是有新GM进程加入Group，但是自己不是最新加入的GM进程，但是自己仍然需要更新成员信息)
handle_msg({catchup, Left, MembersStateLeft},
           State = #state { self = Self,
                            left = {Left, _MRefL},
                            view = View,
                            members_state = MembersState })
  when MembersState =/= undefined ->
    %% 将最新的成员信息转化成字典数据结构
    MembersStateLeft1 = build_members_state(MembersStateLeft),
    %% 拿到左侧镜像队列传入过来的成员信息和自己进程存储的成员信息的ID的去重
    AllMembers = lists:usort(?DICT:fetch_keys(MembersState) ++
                                 ?DICT:fetch_keys(MembersStateLeft1)),
    %% 根据左侧GM进程发送过来的成员状态和自己GM进程里的成员状态得到需要广播给后续GM进程的信息
    {MembersState1, Activity} =
        lists:foldl(
          fun (Id, MembersStateActivity) ->
                  %% 拿到左侧镜像队列传入过来的Id对应的镜像队列成员信息
                  #member { pending_ack = PALeft, last_ack = LA } =
                              find_member_or_blank(Id, MembersStateLeft1),
                  with_member_acc(
                    %% 函数的第一个参数是Id对应的自己进程存储的镜像队列成员信息
                    fun (#member { pending_ack = PA } = Member, Activity1) ->
                            %% 发送者和自己是一个人则表示消息已经发送回来，或者判断发送者是否在死亡列表中
                            case is_member_alias(Id, Self, View) of
                                %% 此情况是发送者和自己是同一个人或者发送者已经死亡
                                true ->
                                    %% 根据左侧GM进程发送过来的ID最新的成员信息和本GM进程ID对应的成员信息得到已经发布的信息
                                    {_AcksInFlight, Pubs, _PA1} =
                                        find_prefix_common_suffix(PALeft, PA),
                                    {Member #member { last_ack = LA },
                                                %% 组装发送的发布和ack消息结构
                                                activity_cons(Id, pubs_from_queue(Pubs),
                                                              [], Activity1)};
                                false ->
                                    %% 根据左侧GM进程发送过来的ID最新的成员信息和本GM进程ID对应的成员信息得到Ack和Pub列表
```

```
                                        {Acks, _Common, Pubs} =
                                            find_prefix_common_suffix(PA, PALeft
),
                                        {Member,
                                         %% 组装发送的发布和ack消息结构
                                         activity_cons(Id, pubs_from_queue(Pubs)
,
                                                       acks_from_queue(Acks),
                                                       Activity1)}
                                end
                        end, Id, MembersStateActivity)
                end, {MembersState, activity_nil()}, AllMembers),
            handle_msg({activity, Left, activity_finalise(Activity)},
                       State #state { members_state = MembersState1 });
```

（2）消息的广播

消息从master节点发出，顺着节点链表发送。在这期间，所有的slave节点都会对消息进行缓存，当master节点收到自己发送的消息后，会再次广播ack消息，同样ack消息会顺着节点链表经过所有的slave节点，其作用是通知slave节点可以清除缓存的消息，当ack消息回到master节点时对应广播消息的生命周期结束。

下图为一个简单的示意图，A节点为master节点，广播一条内容为"test"的消息。"1"表示消息为广播的第一条消息；"id=A"表示消息的发送者为节点A。右边是slave节点记录的状态信息。



为什么所有的节点都需要缓存一份发布的消息呢？

master发布的消息是依次经过所有slave节点，在这期间的任何时刻，有可能有节点失效，那么相邻的节点可能需要重新发送给新的节点。例如，A->B->C->D->A形成的循环链表，A为master节点，广播消息发送给节点B，B再发送给C，如果节点C收到B发送的消息还未发送给D时异常结束了，那么节点B感知后节点C失效后需要重新将消息发送给D。同样，如果B节点将消息发送给C后，B,C节点中新增了E节点，那么B节点需要再将消息发送给新增的E节点。

GM群组中消息广播代码：

```
%% 节点挂掉的情况或者新增节点发送给自己右侧GM进程的信息
%% 左侧的GM进程通知右侧的GM进程最新的成员状态(此情况是有新GM进程加入Group，但是自己不是最新加入的GM进程，但是自己仍然需要更新成员信息)
handle_msg({catchup, Left, MembersStateLeft},
           State = #state { self = Self,
                            left = {Left, _MRefL},
                            view = View,
                            members_state = MembersState })
  when MembersState =/= undefined ->
    %% 将最新的成员信息转化成字典数据结构
    MembersStateLeft1 = build_members_state(MembersStateLeft),
    %% 拿到左侧镜像队列传入过来的成员信息和自己进程存储的成员信息的ID的去重
    AllMembers = lists:usort(?DICT:fetch_keys(MembersState) ++
                               ?DICT:fetch_keys(MembersStateLeft1)),
    %% 根据左侧GM进程发送过来的成员状态和自己GM进程里的成员状态得到需要广播给后续GM进程的信息
    {MembersState1, Activity} =
        lists:foldl(
          fun (Id, MembersStateActivity) ->
                  %% 拿到左侧镜像队列传入过来的Id对应的镜像队列成员信息
                  #member { pending_ack = PALeft, last_ack = LA } =
                              find_member_or_blank(Id, MembersStateLeft1),
                  with_member_acc(
                    %% 函数的第一个参数是Id对应的自己进程存储的镜像队列成员信息
                    fun (#member { pending_ack = PA } = Member, Activity1) ->
                            %% 发送者和自己是一个人则表示消息已经发送回来，或者判断发送者是否在死亡列表中
                            case is_member_alias(Id, Self, View) of
                                %% 此情况是发送者和自己是同一个人或者发送者已经死亡
                                true ->
                                    %% 根据左侧GM进程发送过来的ID最新的成员信息和本GM进程ID对应的成员信息得到已经发布的信息
                                    {_AcksInFlight, Pubs, _PA1} =
                                        find_prefix_common_suffix(PALeft, PA),
                                    %% 重新将自己的消息发布
                                    {Member #member { last_ack = LA },
                                                  %% 组装发送的发布和ack消息结构
                                                  activity_cons(Id, pubs_from_queue(Pubs),
```

```
                                                                  [], Activi
ty1)};
                                    false ->
                                        %% 根据左侧GM进程发送过来的ID最新的成员
信息和本GM进程ID对应的成员信息得到Ack和Pub列表
                                        %% 上一个节点少的消息就是已经得到确认的
消息，多出来的是新发布的消息
                                        {Acks, _Common, Pubs} =
                                            find_prefix_common_suffix(PA, PALeft
),
                                        {Member,
                                         %% 组装发送的发布和ack消息结构
                                         activity_cons(Id, pubs_from_queue(Pubs)
,
                                                       acks_from_queue(Acks),
                                                       Activity1)}
                                end
                        end, Id, MembersStateActivity)
            end, {MembersState, activity_nil()}, AllMembers),
    handle_msg({activity, Left, activity_finalise(Activity)},
               State #state { members_state = MembersState1 });
```

GM主进程内消息在群组中发送的时候的发送原理：处理完消息后，先将消息内容写入缓存中：

```
%% GM进程内部广播的接口(先调用本GM进程的回调进程进行处理消息，然后将广播数据放
入广播缓存中)
internal_broadcast(Msg, SizeHint,
                   State = #state { self                = Self,
                                    pub_count           = PubCount,
                                    module              = Module,
                                    callback_args       = Args,
                                    broadcast_buffer    = Buffer,
                                    broadcast_buffer_sz = BufferSize }) ->
    %% 将发布次数加一
    PubCount1 = PubCount + 1,
    {%% 先将消息调用回调模块进行处理
     Module:handle_msg(Args, get_pid(Self), Msg),
     %% 然后将广播消息放入广播缓存
     State #state { pub_count           = PubCount1,
                    broadcast_buffer    = [{PubCount1, Msg} | Buffer],
                    broadcast_buffer_sz = BufferSize + SizeHint}}.
```

当处理完当条消息后，如果缓存中有数据则启动发送消息的定时器：

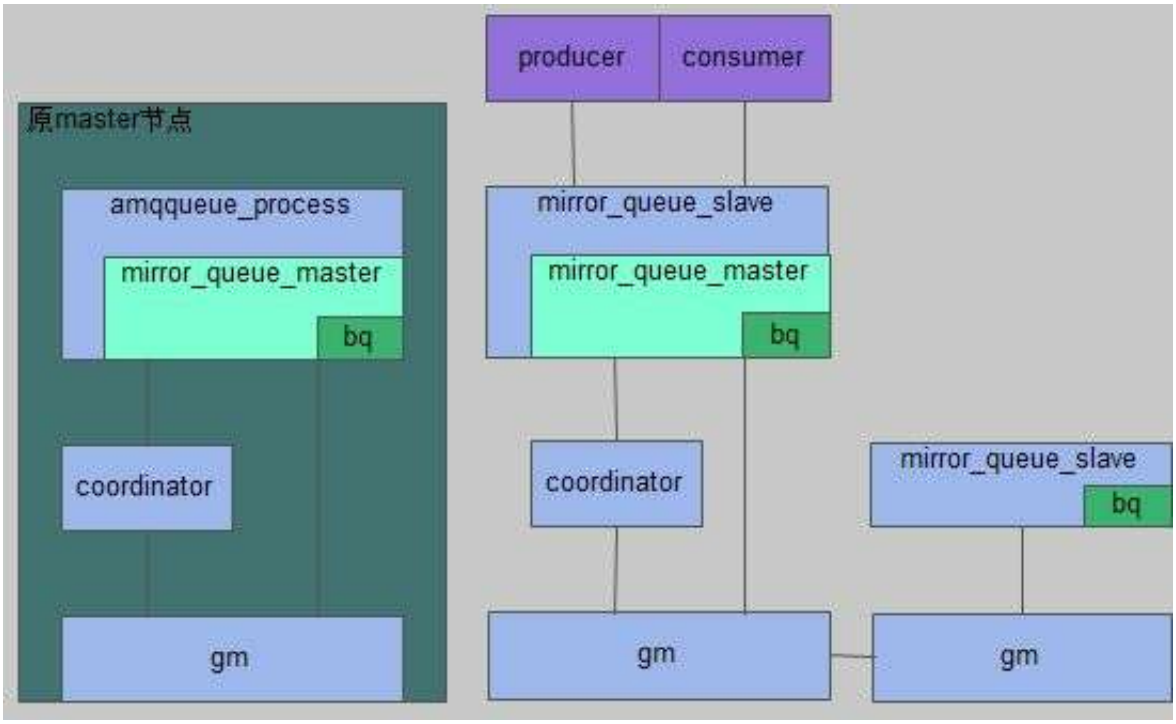
```
%% 确保广播定时器的关闭和开启，当广播缓存中有数据则启动定时器，当广播缓存中没
有数据则停止定时器
%% 广播缓存中没有数据，同时广播定时器不存在的情况
ensure_broadcast_timer(State = #state { broadcast_buffer = [],
                                        broadcast_timer  = undefined }) ->
    State;
%% 广播缓存中没有数据，同时广播定时器存在，则直接将定时器删除掉
ensure_broadcast_timer(State = #state { broadcast_buffer = [],
```

```
                                        broadcast_timer  = TRef }) ->
    erlang:cancel_timer(TRef),
    State #state { broadcast_timer = undefined };
%% 广播缓存中有数据且没有定时器的情况
ensure_broadcast_timer(State = #state { broadcast_timer = undefined }) ->
    TRef = erlang:send_after(?BROADCAST_TIMER, self(), flush),
    State #state { broadcast_timer = TRef };
ensure_broadcast_timer(State) ->
    State.
```

当处理消息的时候，缓存中的内容大小超过100M则不会等待刷新定时器，会立刻将消息发给自己右侧的GM进程

(3) 节点的失效

当slave节点失效时，仅仅是相邻节点感知，然后重新调整邻居节点信息、更新rabbit_queue、gm_group的记录等。如果是master节点失效，"资格最老"的slave节点被提升为master节点，slave节点会创建出新的coordinator，并告知gm修改回调处理为coordinator，原来的mirror_queue_slave充当amqqueue_process处理生产者发布的消息，向消费者投递消息等。



上面提到如果是slave节点失效，只有相邻的节点能感知到，那么master节点失效是不是也是只有相邻的节点能感知到？假如是这样的话，如果相邻的节点不是"资格最老"的节点，怎么通知"资格最老"的节点提升为新的master节点呢？

实际上，所有的slave节点在加入group时，mirror_queue_slave进程会对master节点的amqqueue_process进程（也可能是mirror_queue_slave进程）进行监控，如果master节点失效的话，mirror_queue_slave会感知，然后再通过gm进行广播，这样所有的节点最终都会知道master节点失效。当然，只有"资格最老"的节点会提升自己为新的master。

另外，在slave提升为master时，mirror_queue_slave内部来了一次"偷梁换柱"，即原本需要回调mirror_queue_slave的handle_call/handle_info/handle_cast等接口进行处理的消息，全部改为调用amqqueue_process的handle_call/handle_info/handle_cast等接口，从而可以解释上面说的，mirror_queue_slave进程充当了amqqueue_process完成协议相关的消息的处理。

GM进程挂掉的处理代码：

```
%% 接收到自己左右两边的镜像队列GM进程挂掉的消息
handle_info({'DOWN', MRef, process, _Pid, Reason},
            State = #state { self          = Self,
                             left          = Left,
                             right         = Right,
                             group_name    = GroupName,
                             confirms      = Confirms,
                             txn_executor  = TxnFun }) ->
    %% 得到挂掉的GM进程
    Member = case {Left, Right} of
                 %% 左侧的镜像队列GM进程挂掉的情况
                 {{Member1, MRef}, _} -> Member1;
                 %% 右侧的镜像队列GM进程挂掉的情况
                 {_, {Member1, MRef}} -> Member1;
                 _                    -> undefined
             end,
    case {Member, Reason} of
        {undefined, _} ->
            noreply(State);
        {_, {shutdown, ring_shutdown}} ->
            noreply(State);
        _ ->
            %% In the event of a partial partition we could see another member
            %% go down and then remove them from Mnesia. While they can
            %% recover from this they'd have to restart the queue - not
            %% ideal. So let's sleep here briefly just in case this was caused
            %% by a partial partition; in which case by the time we record the
            %% member death in Mnesia we will probably be in a full
            %% partition and will not be assassinating another member.
            timer:sleep(100),
            %% 先记录有镜像队列成员死亡的信息，然后将所有存活的镜像队列组装镜像队列群组循环队列视图
            %% 有成员死亡的时候会将版本号增加一，record_dead_member_in_group函数是更新gm_group数据库表中的数据，将死亡信息写入数据库表
            View1 = group_to_view(record_dead_member_in_group(
                                    Member, GroupName, TxnFun)),
            handle_callback_result(
              case alive_view_members(View1) of
                  %% 当存活的镜像队列GM进程只剩自己的情况
                  [Self] -> maybe_erase_aliases(
                              State #state {
                                            members_state = blank_member_state
(),
                                            confirms      = purge_confirms(Con
```

```
firms) },
                                                View1);
                    %% 当存活的镜像队列GM进程不止自己(根据新的镜像队列循环队列视
图和老的视图修改视图，同时根据镜像队列循环视图更新自己左右邻居信息)
                    %% 同时将当前自己节点的消息信息发布到自己右侧的GM进程
                    _       -> change_view(View1, State)
                end)
    end.
```

主镜像队列回调模块rabbit_mirror_queue_coordinator处理GM进程挂掉的情况代码：

%% 处理循环镜像队列中有死亡的镜像队列(主镜像队列接收到死亡的镜像队列不可能是主镜像队列死亡的消息，它监视的左右两侧的副镜像队列进程)

```
handle_cast({gm_deaths, DeadGMPids},
            State = #state { q  = #amqqueue { name = QueueName, pid = MPid } }
)
  when node(MPid) =:= node() ->
    %% 返回新的主镜像队列进程，死亡的镜像队列进程列表，需要新增加镜像队列的节
点列表
    case rabbit_mirror_queue_misc:remove_from_queue(
           QueueName, MPid, DeadGMPids) of
        {ok, MPid, DeadPids, ExtraNodes} ->
            %% 打印镜像队列死亡的日志
            rabbit_mirror_queue_misc:report_deaths(MPid, true, QueueName,
                                                   DeadPids),
            %% 异步在ExtraNodes的所有节点上增加QName队列的副镜像队列
            rabbit_mirror_queue_misc:add_mirrors(QueueName, ExtraNodes, async)
,
            noreply(State);
        {error, not_found} ->
            {stop, normal, State}
    end;
```

副镜像队列回调模块rabbit_mirror_queue_slave处理GM进程挂掉的情况代码：

```
%% 副镜像队列处理有镜像队列成员死亡的消息(副镜像队列接收到主镜像队列死亡的消息
)
handle_call({gm_deaths, DeadGMPids}, From,
            State = #state { gm = GM, q = Q = #amqqueue {
                                                        name = QName, pid = M
Pid }}) ->
    Self = self(),
    %% 返回新的主镜像队列进程，死亡的镜像队列进程列表，需要新增加镜像队列的节
点列表
    case rabbit_mirror_queue_misc:remove_from_queue(QName, Self, DeadGMPids) o
f
        {error, not_found} ->
            gen_server2:reply(From, ok),
            {stop, normal, State};
        {ok, Pid, DeadPids, ExtraNodes} ->
            %% 打印镜像队列死亡的日志(Self是副镜像队列)
            rabbit_mirror_queue_misc:report_deaths(Self, false, QName,
```

```
                                                DeadPids),
            case Pid of
                %% 此情况是主镜像队列没有变化
                MPid ->
                    %% master hasn't changed
                    gen_server2:reply(From, ok),
                    %% 异步在ExtraNodes的所有节点上增加QName队列的副镜像队列
                    rabbit_mirror_queue_misc:add_mirrors(
                      QName, ExtraNodes, async),
                    noreply(State);
                %% 此情况是本副镜像队列成为主镜像队列
                Self ->
                    %% we've become master
                    %% 将自己这个副镜像队列提升为主镜像队列
                    QueueState = promote_me(From, State),
                    %% 异步在ExtraNodes的所有节点上增加QName队列的副镜像队列
                    rabbit_mirror_queue_misc:add_mirrors(
                      QName, ExtraNodes, async),
                    %% 返回消息，告知自己这个副镜像队列成为主镜像队列
                    {become, rabbit_amqqueue_process, QueueState, hibernate};
                _ ->
                    %% 主镜像队列已经发生变化
                    %% master has changed to not us
                    gen_server2:reply(From, ok),
                    %% assertion, we don't need to add_mirrors/2 in this
                    %% branch, see last clause in remove_from_queue/2
                    [] = ExtraNodes,
                    %% Since GM is by nature lazy we need to make sure
                    %% there is some traffic when a master dies, to
                    %% make sure all slaves get informed of the
                    %% death. That is all process_death does, create
                    %% some traffic.
                    ok = gm:broadcast(GM, process_death),
                    noreply(State #state { q = Q #amqqueue { pid = Pid } })
            end
    end;
```

主镜像队列挂掉后选取新的主镜像队列的代码：

```
%% Returns {ok, NewMPid, DeadPids, ExtraNodes}
%% 返回新的主镜像队列进程，死亡的镜像队列进程列表，需要新增加镜像队列的节点列表
remove_from_queue(QueueName, Self, DeadGMPids) ->
    rabbit_misc:execute_mnesia_transaction(
      fun () ->
              %% Someone else could have deleted the queue before we
              %% get here.
              case mnesia:read({rabbit_queue, QueueName}) of
                  [] -> {error, not_found};
                  [Q = #amqqueue { pid        = QPid,
                                   slave_pids = SPids,
                                   gm_pids    = GMPids }] ->
```

```
                        %% 获得死亡的GM列表和存活的GM列表
                        {DeadGM, AliveGM} = lists:partition(
                                              fun ({GM, _}) ->
                                                      lists:member(GM, DeadGMP
ids)
                                              end, GMPids),
                        %% 获得死亡的实际进程的Pid列表
                        DeadPids  = [Pid || {_GM, Pid} <- DeadGM],
                        %% 获得存活的实际进程的Pid列表
                        AlivePids = [Pid || {_GM, Pid} <- AliveGM],
                        %% 获得slave_pids字段中存活的队列进程Pid列表
                        Alive     = [Pid || Pid <- [QPid | SPids],
                                            lists:member(Pid, AlivePids)],
                        %% 从存活的镜像队列提取出第一个镜像队列进程Pid，它是最
老的镜像队列，它将作为新的主镜像队列进程
                        {QPid1, SPids1} = promote_slave(Alive),
                        Extra =
                            case {{QPid, SPids}, {QPid1, SPids1}} of
                                {Same, Same} ->
                                    [];
                                %% 此处的情况是主镜像队列没有变化，或者调用此接
口的副镜像队列成为新的主镜像队列
                                _ when QPid =:= QPid1 orelse QPid1 =:= Self ->
                                    %% Either master hasn't changed, so
                                    %% we're ok to update mnesia; or we have
                                    %% become the master.
                                    Q1 = Q#amqqueue{pid        = QPid1,
                                                    slave_pids = SPids1,
                                                    gm_pids    = AliveGM},
                                    %% 存储更新队列的副镜像队列信息
                                    store_updated_slaves(Q1),
                                    %% If we add and remove nodes at the
                                    %% same time we might tell the old
                                    %% master we need to sync and then
                                    %% shut it down. So let's check if
                                    %% the new master needs to sync.
                                    %% 根据队列的策略如果启动的副镜像队列需要自
动同步，则进行同步操作
                                    maybe_auto_sync(Q1),
                                    %% 根据当前集群节点和副镜像队列进程所在的节
点得到新增加的节点列表
                                    slaves_to_start_on_failure(Q1, DeadGMPids);
                                %% 此处的情况是主镜像队列已经发生变化，且调用此
接口的副镜像队列没有成为新的主镜像队列
                                _ ->
                                    %% Master has changed, and we're not it.
                                    %% [1].
                                    %% 更新最新的存活的副镜像队列进程Pid列表和
存活的GM进程列表
                                    Q1 = Q#amqqueue{slave_pids = Alive,
                                                    gm_pids    = AliveGM},
```

```
                                %% 存储更新队列的副镜像队列信息
                                store_updated_slaves(Q1),
                                []
                        end,
                    {ok, QPid1, DeadPids, Extra}
            end
    end).
```

（4）消息的同步

配置镜像队列的时候有个ha-sync-mode属性，这个有什么用呢？

新节点加入到group后，最多能从左边节点获取到当前正在广播的消息内容，加入group之前已经广播的消息则无法获取到。如果此时master节点不幸失效，而新节点有恰好成为了新的master，那么加入group之前已经广播的消息则会全部丢失。

注意：这里的消息具体是指新节点加入前已经发布并复制到所有slave节点的消息，并且这些消息还未被消费者消费或者未被消费者确认。如果新节点加入前，所有广播的消息被消费者消费并确认了，master节点删除消息的同时会通知slave节点完成相应动作。这种情况等同于新节点加入前没有发布任何消息。

避免这种问题的解决办法就是对新的slave节点进行消息同步。当ha-sync-mode配置为自动同步（automatic）时，新节点加入group时会自动进行消息的同步；如果配置为manually则需要手动操作完成同步。