RabbitMQ系统中消息队列的实现原理

笔记本： RabbitMq源代码阅读系统原理分析
创建时间： 2015/7/6 18:47 **更新时间：** 2015/11/17 16:04

当rabbit无法直接将消息投递到消费者时，需要暂时将消息入队列，以便重新投递。但是，将消息入队列并不意味着，就是将消息扔在一个队列中就不管了，rabbit提供了一套状态转换的机制来管理消息。

在rabbit中，队列中的消息可能会处理以下四种状态：
alpha：消息内容以及消息在队列中的位置（消息索引）都保存在内存中；
beta：消息内容只在磁盘上，消息索引只在内存中；
gamma：消息内容只在磁盘上，消息索引在内存和磁盘上都存在；
delta：消息的内容和索引都在磁盘上。
注意：对于持久化消息，消息内容和消息索引都必须先保存在磁盘上，然后才处于上述状态中的一种。其中gamma很少被用到，只有持久化的消息才会出现这种状态。

rabbit提供这种分类的主要作用是满足不同的内存和CPU需求。alpha最耗内存，但很少消耗CPU；delta基本不消耗内存，但是要消耗更多的CPU以及磁盘I/O操作（delta需要两次I/O操作，一次读索引，一次读消息内容；beta及gamma只需要一次I/O操作来读取消息内容）。

rabbit在运行时会根据统计的消息传送速度定期计算一个当前内存中能够保存的最大消息数量（target_ram_count），如果内存中的消息数量大于这个数量，就会引起消息的状态转换，转换主要两种：alpha -> beta， beta -> delta。alpha -> beta的转换会将消息的内容写到磁盘（如果是持久化消息，在这一步转换后，消息将会处于gamma状态），beta -> delta的转换会更进一步减少内存消耗，将消息索引也写到磁盘。

运行时怎么体现消息的各种状态呢？在队列的状态vqstate（参见[$RABBIT_SRC/src/rabbit_variable_queue.erl]）结构中，存在q1，q2，delta，q3，q4五个队列（使用q1~q4使用Erlang的queue模块，delta存储结构依赖于消息索引的实现），其中q1，q4只包含alpha状态的消息，q2，q3只包含beta和gamma状态的消息，delta包含delta状态的消息。

一般情况下消息会以q1->q2->delta->q3->q4的顺序进行状态变换：消息进入队列时，处于alpha状态并保存在内存中（q1或q4），然后某个时刻发现内存不足，被转换到beta状态（q2，q3）（这时候其实有两个转换q1->q2，q4->q3），如果还是内存不足，被转换到delta状态（delta）（q2->delta，q3->delta）；当从队列中消费消息时，会先从处于alpha状态的内存队列（q4）中获取消息，如果q4为空，则从beta状态的队列（q3）中获取消息，如果q3也为空，则会从delta状态的消息队列中读取消息，并将之转移到q3。

上述步骤只是个一般步骤，实际运行时，中间的步骤都是可以跳过的，比如消息可能在一开始被放在q4队列中、q1队列中的元素会直接跳到q4队列中（这两种情况下，q3，delta，q2队列都为空）；当delta队列为空时，q2队列可以直接跳到q3队列，q1队列也可以直接跳到q3。

上述这些状态转换主要发生的两个地方为：从队列中获取消息时（[$RABBIT_SRC/src/rabbit_variable_queue.erl -> queue_out/1]）以及减少内存使用量时

（[$RABBIT_SRC/src/rabbit_variable_queue.erl -> reduce_memery_use/1]）。下面详细说明下这两个操作的逻辑。

Erlang代码

```
$RABBIT_SRC/src/rabbit_variable_queue.erl -> queue_out/1
queue_out(State = #vqstate { q4 = Q4 }) ->
    case ?QUEUE:out(Q4) of
        {empty, _Q4} ->
            case fetch_from_q3(State) of
                {empty, _State1} = Result     -> Result;
                {loaded, {MsgStatus, State1}} -> {{value, MsgStatus}, State1}
            end;
        {{value, MsgStatus}, Q4a} ->
            {{value, MsgStatus}, State #vqstate { q4 = Q4a }}
    end.
```

逻辑很简单，尝试从q4队列中获取一个消息，如果成功，则返回获取到的消息，如果失败，则尝试通过试用fetch_from_q3/1从q3队列获取消息，成功则返回，如果为空则返回空。

Erlang代码

```
$RABBIT_SRC/src/rabbit_variable_queue.erl -> fetch_from_q3/1
fetch_from_q3(State = #vqstate { q1    = Q1,
                                 q2    = Q2,
                                 delta = #delta { count = DeltaCount },
                                 q3    = Q3,
                                 q4    = Q4 }) ->
    case ?QUEUE:out(Q3) of
        {empty, _Q3} ->
            {empty, State};
        {{value, MsgStatus}, Q3a} ->
            State1 = State #vqstate { q3 = Q3a },
            State2 = case {?QUEUE:is_empty(Q3a), 0 == DeltaCount} of
                         {true, true} ->
                             true = ?QUEUE:is_empty(Q2), %% ASSERTION
                             true = ?QUEUE:is_empty(Q4), %% ASSERTION
                             State1 #vqstate { q1 = ?QUEUE:new(), q4 = Q1 }; // A
                         {true, false} ->
                             maybe_deltas_to_betas(State1); // B
                         {false, _} ->
                             State1
                     end,
            {loaded, {MsgStatus, State2}}
    end.
```

从case语句的第一个分支可以看到，当发现q3为空时，就直接返回了空。也就说当q3和q4为空时，整个队列为空（后续解释）。再看case的第二个分支，当q3非空，并且成功的从q3中取出一条消息后，需要检查取出消息后的队列状态。内嵌的case检查取出消息后q3的队列长度和当前delta队列的长度：1）当这两个队列都为空时，可以确认q2也为空（后续解释），也就是这时候，q2，q3，delta，q4都为空，那么，q1队列的消息可以直接转移到q4，下次获取消息时就可以直接从q4获取；2）q3空，delta非空，这时候就需要从delta队列（内容与索引都在磁盘上，通过maybe_deltas_to_betas/1调用）读取消息，并转移到q3队列；3）q3非空，直接返回，下次获取消息还可以从q3获取。

Erlang代码

```
$RABBIT_SRC/src/rabbit_variable_queue.erl -> maybe_deltas_to_betas/1
maybe_deltas_to_betas(State = #vqstate {
```

```
                        q2                  = Q2,
                        delta               = Delta,
                        q3                  = Q3,
                        index_state         = IndexState,
                        pending_ack         = PA,
                        transient_threshold  = TransientThreshold }) ->
    #delta { start_seq_id = DeltaSeqId,
             count        = DeltaCount,
             end_seq_id   = DeltaSeqIdEnd } = Delta,
    DeltaSeqId1 =
        lists:min([rabbit_queue_index:next_segment_boundary(DeltaSeqId),
                   DeltaSeqIdEnd]),
    {List, IndexState1} =
        rabbit_queue_index:read(DeltaSeqId, DeltaSeqId1, IndexState),
    {Q3a, IndexState2} =
        betas_from_index_entries(List, TransientThreshold, PA, IndexState1),
    State1 = State #vqstate { index_state = IndexState2 },
    case ?QUEUE:len(Q3a) of
        0 ->
            maybe_deltas_to_betas(
              State1 #vqstate {
                delta = d(Delta #delta { start_seq_id = DeltaSeqId1 })});
        Q3aLen ->
            Q3b = ?QUEUE:join(Q3, Q3a),
            case DeltaCount - Q3aLen of
                0 ->
                    State1 #vqstate { q2    = ?QUEUE:new(),
                                      delta = ?BLANK_DELTA,
                                      q3    = ?QUEUE:join(Q3b, Q2) };   // C
                N when N > 0 ->
                    Delta1 = d(#delta { start_seq_id = DeltaSeqId1,
                                        count        = N,
                                        end_seq_id   = DeltaSeqIdEnd }),
                    State1 #vqstate { delta = Delta1,
                                      q3    = Q3b }
            end
    end.
```

在case语句之前，主要的逻辑就是从delta队列中读取消息，Q3a为读取到的消息（按索引分段读取，后面讲索引的时候会详细说明），直到读取的消息不为空为止（在执行这步时，可以确认delta中肯定有消息存在）。当读取到的消息个数与delta中保存的数量一样时（这时delta中所有消息都被读出），delta被重置（消息个数为0），q2中的消息被合并到q3中；不一样的话，就将读出的消息转移到q3。

Erlang代码

```
$RABBIT_SRC/src/rabbit_variable_queue.erl -> reduce_memory_use/1
reduce_memory_use(State) ->
    {_, State1} = reduce_memory_use(fun push_alphas_to_betas/2,
                                    fun push_betas_to_deltas/2,
                                    fun limit_ram_acks/2,
                                    State),
    State1
reduce_memory_use(AlphaBetaFun, BetaDeltaFun, AckFun,
                  State = #vqstate {
                    ram_ack_index    = RamAckIndex,
                    ram_msg_count    = RamMsgCount,
                    target_ram_count = TargetRamCount,
                    rates            = #rates { avg_ingress = AvgIngress,
                                                avg_egress  = AvgEgress },
```

```
                    ack_rates        = #rates { avg_ingress = AvgAckIngress,
                                                avg_egress  = AvgAckEgress }
                  }) ->

    {Reduce, State1 = #vqstate { q2 = Q2, q3 = Q3 }} =
        case chunk_size(RamMsgCount + gb_trees:size(RamAckIndex),
                        TargetRamCount) of
            0  -> {false, State};
            S1 -> {_, State2} =
                      lists:foldl(fun (ReduceFun, {QuotaN, StateN}) ->
                                          ReduceFun(QuotaN, StateN)
                                  end,
                                  {S1, State},
                                  case (AvgAckIngress - AvgAckEgress) >
                                      (AvgIngress - AvgEgress) of
                                      true  -> [AckFun, AlphaBetaFun];
                                      false -> [AlphaBetaFun, AckFun]
                                  end),
                  {true, State2}
        end,

    case chunk_size(?QUEUE:len(Q2) + ?QUEUE:len(Q3),
                    permitted_beta_count(State1)) of
        ?IO_BATCH_SIZE = S2 -> {true, BetaDeltaFun(S2, State1)};
        _                   -> {Reduce, State1}
    end
chunk_size(Current, Permitted)
  when Permitted =:= infinity orelse Permitted >= Current ->
    0;
chunk_size(Current, Permitted) ->
    lists:min([Current - Permitted, ?IO_BATCH_SIZE]).
```

也就是说当内存中的消息数量（RamMsgCount）及内存中的等待ack的消息数量（RamAckIndex）的和大于允许的内存消息数量（TargetRamCount）时，多余数量的消息内容会被写到磁盘中，调用AckFun（limit_ram_acks/2）或者AlphaBetaFun（push_alphas_to_betas/2）来完成（保证每次的写操作个数不超过IO_BATCH_SIZE），先调用哪个由当前的消息速度来决定：如果当前ack的速度大于消息进入队列的速度，则调用AckFun，否则调用AlphaBetaFun。当beta状态的消息（q2，q3）多于允许的数量时，会调用BetaDeltaFun（push_betas_to_deltas/2）来将多余的消息索引写入磁盘，同时可以看到，只有这个多余的数量达到IO_BATCH_SIZE（值为64），才会引用写操作（文档对此的解释是：这个值不能太小也不能太大：太大，一次写操作会将整个队列阻塞更长的时间，太小，频繁的写操作，也会对q2、q3队列产生不利影响）。

$RABBIT_SRC/src/rabbit_variable_queue.erl -> limit_ram_acks /2

逻辑很简单，从pending_ack中拿到最大序号（seq_id，序号越大，等待的时间越短）的消息，并将消息内容写入磁盘，直到pending_ack中的消息数为0，或者要写入的消息数量已经完成。

如果处理完成正在等待ack的内存消息后，要写入磁盘的消息数量还没有完成（pending_ack中的消息数量已为0），则会调用push_alphas_to_betas/2。

Erlang代码

```
$RABBIT_SRC/src/rabbit_variable_queue.erl -> push_alphas_to_betas/2
push_alphas_to_betas(Quota, State) ->
    {Quota1, State1} =
        push_alphas_to_betas(
          fun ?QUEUE:out/1,
          fun (MsgStatus, Q1a,
               State0 = #vqstate { q3 = Q3, delta = #delta { count = 0 } }) ->
```

```
 8.                     State0 #vqstate { q1 = Q1a, q3 = ?QUEUE:in(MsgStatus, Q3) };
 9.                 (MsgStatus, Q1a, State0 = #vqstate { q2 = Q2 }) ->
10.                     State0 #vqstate { q1 = Q1a, q2 = ?QUEUE:in(MsgStatus, Q2) }
11.             end, Quota, State #vqstate.q1, State),
12.      {Quota2, State2} =
13.          push_alphas_to_betas(
14.            fun ?QUEUE:out_r/1,
15.            fun (MsgStatus, Q4a, State0 = #vqstate { q3 = Q3 }) ->
16.                    State0 #vqstate { q3 = ?QUEUE:in_r(MsgStatus, Q3), q4 = Q4a }
17.            end, Quota1, State1 #vqstate.q4, State1),
18.      {Quota2, State2}.
```

逻辑也很简单，q1向q2或者q3转移（取决于delta队列是否为空：如果为空，则向q3转移，否则向q2转移），q4向q3转移。push_alphas_to_betas/4的实现基本上就是把消息内容写到磁盘上，然后更新相关状态（消息的，队列的）。

现在来解释下前面的疑问：

q4和q3为空时，为什么整个队列为空：如果delta非空，则在q3变为空之前（取q3中最后一条消息时），B处代码最终引起delta中的消息转移到q3，并最终导致q3非空，与前提矛盾，所以delta肯定为空；同理，q2也必空，不然的话，通过B处的调用，最终将使q2的消息转移到q3（C），从而导致q3非空；如果q1非空，则A处的代码将使q1的消息转移到q4，从而导致q4非空，矛盾，所以q1为空。

q4为空，q3变空，delta已为空的情况下，为什么q2必空：首先可以肯定，在q3变空前，delta已为空。那么，在q3非空，delta变空的时刻，B处的代码调用通过C会将q2中的消息转移到q3，q2变为空（当然在q2变为空与q3变为空之间有一个时间段，这段时间内，唯一有机会使q2非空的操作是push_alphas_to_betas/2，但是因为delta队列为空，如果有转移，q1也只会转移到q3）。

RabbitMQ系统中队列进程中消息流动逻辑

1.publish(消息的插入)

如果Q3队列为空，则将消息放入Q4队列，如果Q3队列不为空，则将消息放入Q1队列

2.alpha类型的消息转化为beta类型的消息(将消息的内容存入磁盘文件)(push_alphas_to_betas函数)

(1).Q1队列的转化，如果磁盘文件中的消息数量为0，则将Q1中的消息写入Q3队列，如果磁盘文件中的消息数量不为0，则将Q1中的消息写入Q2队列

(2).Q4队列的转化，从Q4队列末端将消息放入到Q3队列的末端

3.beta类型的消息delta类型的消息(push_betas_to_deltas函数)

(1).从Q3队列末端将消息的队列索引存入磁盘文件

(2).将Q2队列中的消息的队列索引存入磁盘文件

4.如果Q4，Q3中的消息为空，且磁盘文件中有消息，则将从磁盘中读取消息，会将delta类型的消息转化为beta类型的消息

(1).如果读取一个队列索引磁盘文件后磁盘文件中已经没有消息，则先将读取到的消息存入Q3队列末尾，然后将Q2队列中的消息存入Q3队列的末尾

(2).如果读取一个队列索引磁盘文件后的磁盘文件中还有消息，则将读取到的消息存入Q3队列末尾

得到最终的结果(消息在这4个队列和磁盘中的顺序)：Q4 <- Q3 <- delta <- Q2 <- Q1