|  |
| --- |
|  |
| Massq内部实现细节 |
| 1.0.0 |
|  |
| **兰生** |
| **2014/1/5** |

l

目录

[1. 前言 4](#_Toc378163479)

[2. 存储结构对比 4](#_Toc378163480)

[3. 差异 6](#_Toc378163481)

[3.1 数据写入 6](#_Toc378163482)

[3.2 数据拉取 7](#_Toc378163483)

[3.3 数据删除 7](#_Toc378163484)

[3.4 数据恢复 13](#_Toc378163485)

[4. 面临的问题 15](#_Toc378163486)

[4.1 小包优化 15](#_Toc378163487)

[4.2 内存回收 16](#_Toc378163488)

[4.3 锁瓶颈 16](#_Toc378163489)

[4.4 数据堆积 16](#_Toc378163490)

[4.5 CPU利用率 16](#_Toc378163491)

[4.6 操作异常处理 16](#_Toc378163492)

[4.7 数据复制HA 16](#_Toc378163493)

[4.8 编码 16](#_Toc378163494)

[4.9 双向链表特性挖掘 16](#_Toc378163495)

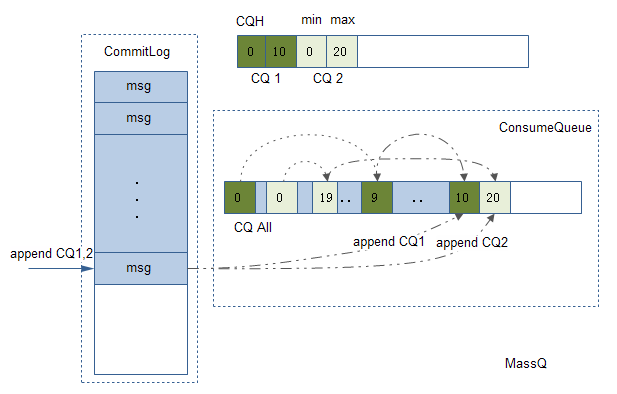
[5. MassQ适合的场景 17](#_Toc378163496)

# 前言

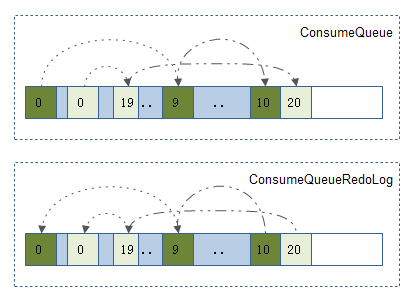
MassQ的目标是支持海量的消费队列，所以在索引文件的组织上和MetaQ相比有较大的变化；不在按照Topic的维度组织真实的数据，减少文件映射；通过链表的方式将各个消费队列组织起来。

# 存储结构对比

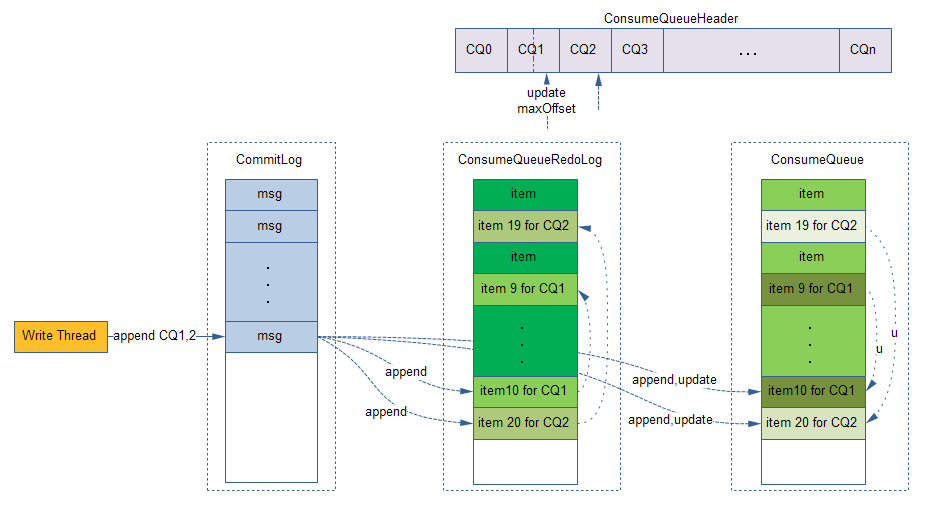
**MassQ存储结构：**



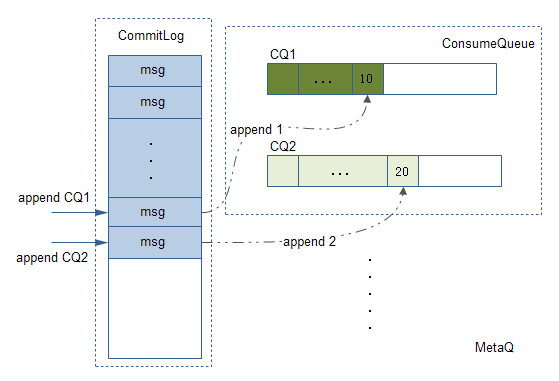
CL（CommitLog）顺序写，CQ（ConsumeQueue）随机写并且没法记录一个确切的刷盘位点；所以CQ没有定时刷盘；只有broker正常结束的时候才全部刷盘。为了异常恢复更加快速引入CQR（ConsumeQueueRedoLog），是CQ的反向链表。



CQ，CQR的组织类似于链表。需要知道前一个节点的位置才能知道后面的位置。所以引入CQH（ConsumeQueueHeader）记录每个CQ的起始、结束位置。CQ是老的数据指向新的数据，所以每次写入新数据需要更新前一个节点的next指向。CQR是新的索引项指向老的；所以只需往后追加即可不用更新前一个消息的next指向，因而可以有一个确切的刷盘点，数据恢复根据CL和CQR中记录的最小时间点做恢复。一个数据写入的完整流程如下：



**MetaQ存储结构：**



CL顺序写入，CQ是随机写入；同一个队列的索引在同一个文件内，拉取消息的时候单个队列内是顺序读。CQ组织形式类似于数据，所以可以很容易知道下一个写入的位置，写入位置是递增的，因为无需CQH记录队列的起始，结束位置。CQ只有追加操作，没有更新操作，所以每个CQ有一个确切的刷盘位置。因为不同的CQ是存放在不同的文件中，所以刷盘需要同时刷多个文件。

MassQ和MetaQ存储结构的差异导致数据删除和数据恢复上有较大的区别。同时数据写入和读取也有一些需要特别留意的地方。

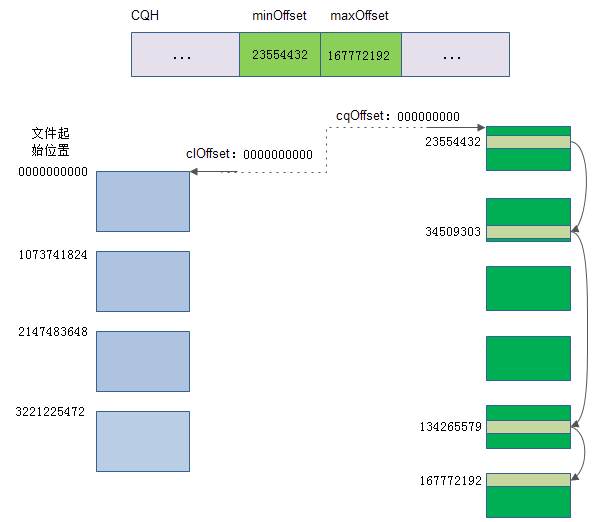
# 差异

## 数据写入

## 数据拉取

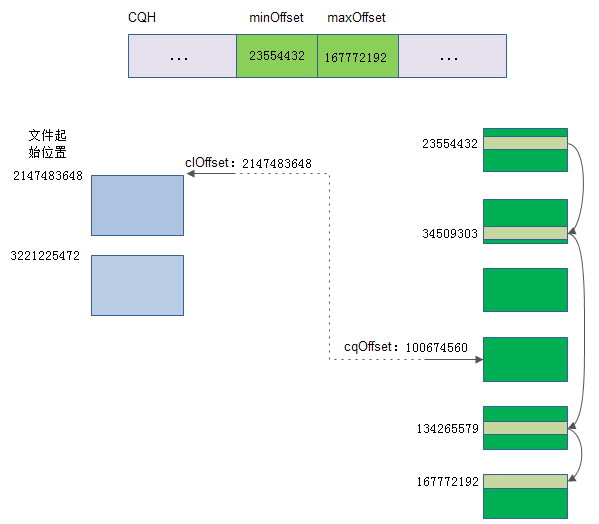
## 数据删除

删除算法：根据文件有效期，删除失效的CL文件；然后根据删除后CL中最小的Offset去纠正CQH，并删除对应无效CQ文件。流程如下：

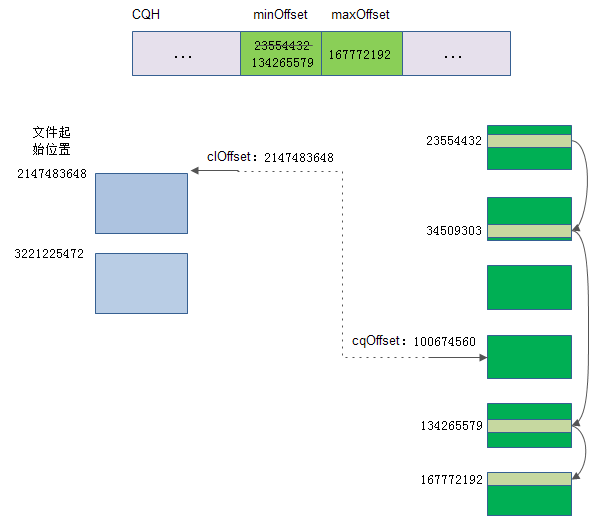


删除之前CL有4个有效文件，起始位置分别是0000000000, 1073741824, 2147483648, 3221225472；对应有效索引文件有6个；其中有一个消费队列有四条索引项（位置分别为23554432, 34509303, 134265579, 167772192）分散在其中的4个索引文件中，CQH中记录着该队列的起始，结束位置。

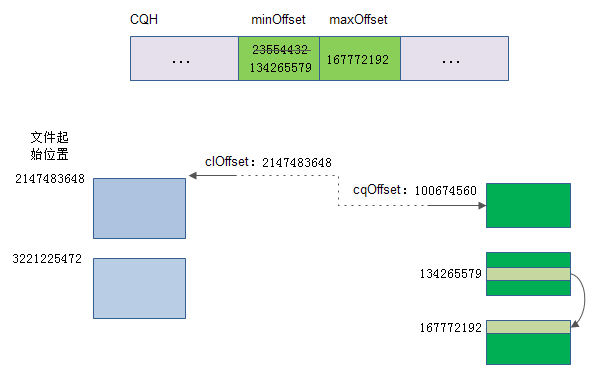
假设此时触发删除，并且CL的前面两个文件（0000000000， 1073741824）已过有效期。首先删除前两个物理文件。



此时物理文件的最小有效位置从0000000000变成2147483648；对应的有效逻辑位置也变到了100674560；可以看出前面两条索引（23554432, 34509303）已经无效，紧接着会做CQH的清理。



纠正CQH中记录minOffset从23554432变到134265579；最后清除无效的索引文件



注意清除无效索引文件只会删除文件里面全是失效索引的文件；有可能CL第一个有效位置对应着索引文件的中间某个位置，此时该索引文件不会被删除，只会等到该索引文件全部失效才会删除。

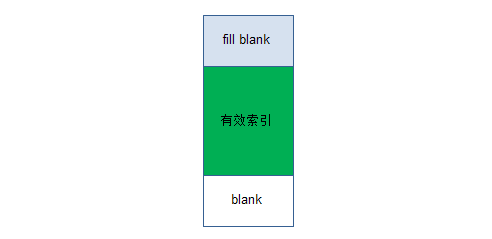
***删除正确性***，取保数据删除后剩余数据能正确的消费，无效数据及时删除释放空间

**如何保证CQ和CQR删除后长度一致？**

异常恢复的时候会根据CQR反转CQ，所以必须保证数据删除后CQR和CQ的长度一致；删除过程中有可能删除文件失败（比如文件还被引用），所以每次在删除数据的时候需保证CQR和CQ的长度一致才能进行新一轮的数据删除操作，及保证上一轮的删除完整。

CQR的删除比较简单无需做数据的纠正，直接删除就好。

**如何保证删除后CQH不会记录无效的索引位置？**



数据异常重构后可能会出现上面图示的数据分布；开始部分（fill blank）是异常恢复时填充的无效数据；中间部分是数据索引；结尾部分是空白，新的数据会写到这里直到当前文件写满分配新的文件。所以在纠正CQH的时候需要跳过“填空数据”的部分。

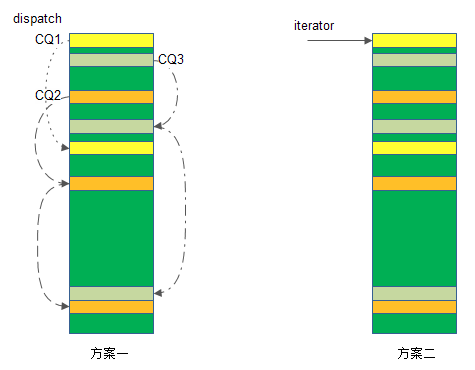
**如何避免写入和删除并行操作的数据竞争？**

如果数据的起始，结束位置都失效，及该队列的所有数据都失效的时候，需要同时将CQH中记录的minOffet和maxOffset设置为无效，如果此时该队列正好有数据写入就有可能产生竞争；此时就需要加锁，但为了效率又不能简单的在数据操作上面直接加锁，这里做了一些优化，只有当发现可能存在竞争的时候才加锁，并且是一把细粒度的锁，大致思想和ConcurrentHashMap锁一致（不支持锁的重入）。

***删除效率***，数据删除，索引的纠正能够尽量的快

**如何尽可能快的纠正CQH记录的队列起始结束位置？**

MassQ索引的组织是将所有消费队列的索引混到一个文件中通过连接的方式将各队列内的索引串起来；所以只有知道前一个节点，通过next指针，才能找到下一个节点的位置，所以纠正CQH的时候需要遍历所有需要删除的索引文件，纠正所有的消费队列的起始、结束位置。因为涉及到IO操作，而磁盘又是一个串行设备，所以磁盘操作效率将决定删除操作的性能。同时和MetaQ索引组织方式的不同，不能根据二分等方式快速定位到有效的索引，所以全部遍历的时候会产生很多的临时对象，加速GC。有两种纠正的方案：



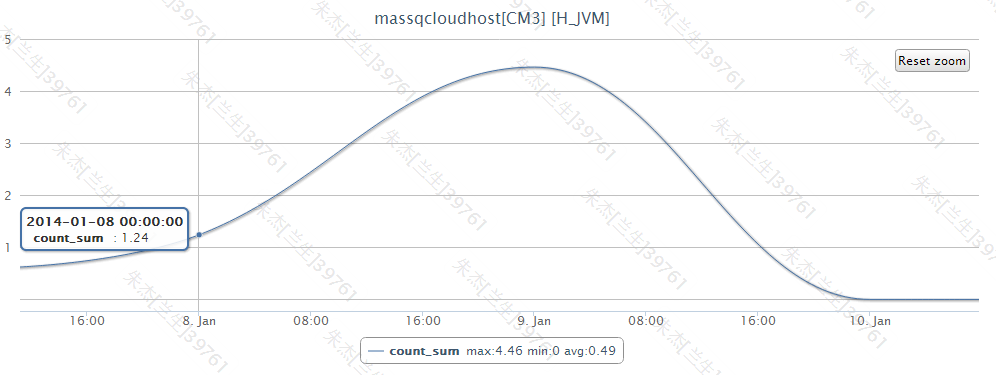
1. 遍历单个CQ文件，单个CQ文件范围内并行的做纠正。

纠正按文件为单位，一个文件一个文件的进行纠正；单个文件内按照队列为维度进行任务派发（dispatch）并行的纠正。按文件为单位进行纠正是为了减少随机读，一个文件期望只产生一次的IO操作，同时在文件内做并行希望利用CPU多核提高执行的速度。

1. 遍历单个CQ文件，单个文件内串行的做纠正。

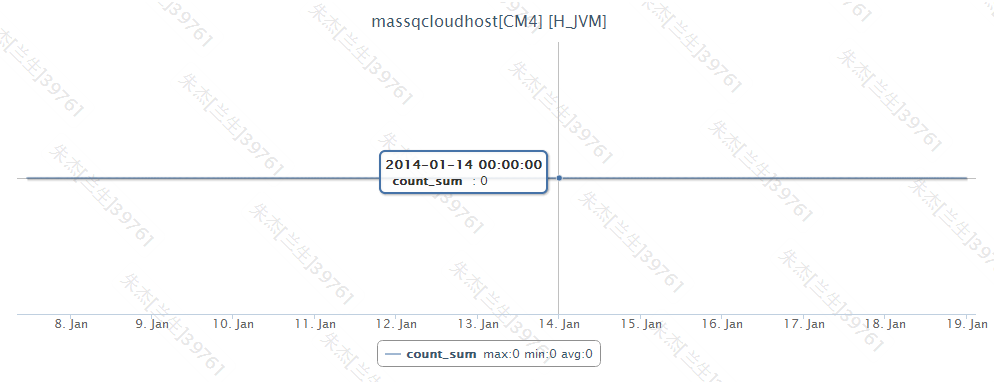
方案2是在方案1上面做了一个变化，将CQH的纠正串行化，因为如果纠正的操作如果足够简单串行（iterator）的操作就能满足能减少线程上下文的切换。同时纠正算法上的优化，减少了对象的产生。

实际测试数据在方案1里面一个单个CQ文件的纠正耗时在6s左右，同时如果一次清除的数据量特别多会触发Full GC。



而方案2里面一个CQ文件的纠正耗时在1s左右，同时如果一次清理的数据多也是成线性的增长。





MassQ实际采用的方案2。

**如何尽快的释放磁盘空间？**

物理文件删除和索引文件删除异步化；物理文件只需要根据文件的有效期将失效的文件直接删除即可，无需其他额外操作；而索引文件需要先纠正所有消费队列的有效位置后才能删除索引文件，而纠正操作会慢很多所以将物理文件的删除和索引文件的删除分开，这样当磁盘快写满的时候能够开速释放空间。

MetaQ数据删除基本思想也是一致，物理文件根据数据有效期进行删除，逻辑文件根据最小物理位置进行删除；相比之下MetaQ会更加直观，MetaQ物理文件和索引文件都是顺序追加，无需引入额外的存储结构（CQR，CQH），同时删除也无需做纠正。

## 数据恢复

**正常恢复**

服务器正常结束，数据完整持久化，执行步骤如下：

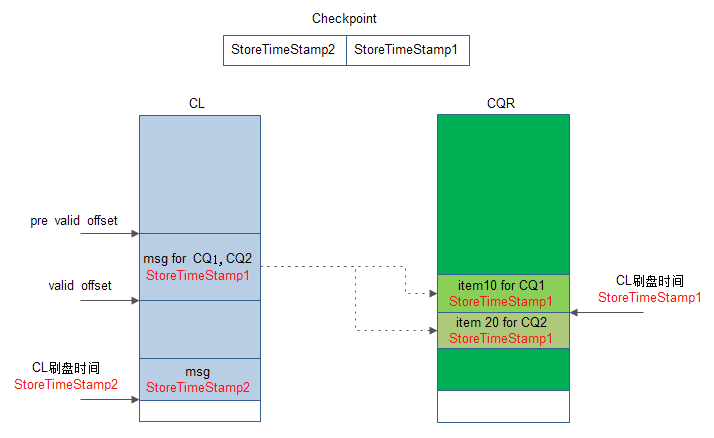
1. 加载CL，CQ，CQR
2. 找到下一次写入的位置

**异常恢复**

服务器异常结束，部分数据可能没有及时持久化，执行步骤如下：

1. 根据Checkpoint中记录的刷盘时间点找到CL的一个有效offset
2. 根据CL的有效offset对CQR进行纠正，truncate掉大于offset的记录
3. 反转CQR得到CQ
4. 重构CL有效offset之后的索引数据

步骤一中找到的是Checkpoint中记录的有效刷盘时间对应的偏移量的后一个有效offset；因为MassQ的还有一个设计目标是支持一次写入多份投递，所以索引文件有可能存在多条索引对应一条真实的消息；同时checkpoint中记录的是消息的StoreTimeStamp；有可能出现以下情况：

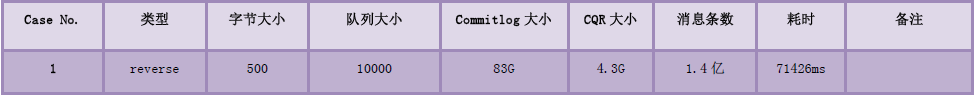


CL刷盘时间是StoreTimeStamp2，CQR中刷盘时间是StoreTimeStamp1；StoreTimeStamp2>StoreTimeStamp1；Checkpoint里面记录的时间对应的有效位置是valid offset但是CQR中的索引item10和item20对应的时间都是StoreTimeStamp1所以当CQR刷盘的位置正好是两条索引的中间位置的时候就不能保证所有StoreTimeStamp1对应的记录都是有效或者无效；所以是从最小刷盘时间的上一条有效位置开始（pre valid offset）。

步骤三CQR反转算法：

1. 串行的遍历文件序列，文件内部按照不同的QueueId并行处理

文件串行遍历，期望一个文件不会产生多次的磁盘IO；文件内部按照QueueId并行处理的好处减少更新CQH的频率，只需要在结束的位置更新CQH记录的有效位置即可。反转4.3G的CQR耗时71426ms：



1. 完全串行的进行文件反转

单线程串行操作，虽然增多了CQH的更新（因为相邻两条索引并不一定是同一队列的索引，所以需要每次遍历操作更新CQH记录的有效位置），但却减少了对象的产生。反转4.7G的CQR耗时26490ms：



MassQ采用的是第二种方式。

**恢复效率**

**为什么要通过CQR反转得到CQ？**

CL里面包含了索引需要的全部数据，通过CQR反转得到CQ而非通过CL重构CQ主要出于执行效率考虑；一条数据比一条索引更大，根据CL重构CQ需要遍历更多的数据，同时需要解析CL的内容，执行时间远大于CQR反转的时间；这也是为什么引入CQR的原因。耗时对比如下：



**数据恢复和数据删除提升效率的指导思想基本一致：**

1. 尽可能顺序读，离散读最大限度的利用文件缓存
2. 减少对象产生，缩短对象生命周期，尽量降低gc开销

MetaQ数据恢复的基本思想也是通过Checkpoint中记录的物理文件和逻辑文件有效刷盘时间，找到两者都有效的位置，从该位置之后进行恢复。由于MetaQ每个队列都是单独组织存储，而非MassQ链表的方式进行组织，所以恢复会更直观些。

# 面临的问题

## 小包优化

资源循环利用，pooled的方式进行资源的循环利用。

## 内存回收

GC回收效率；堆内堆外数据回收

## 锁瓶颈

细化锁，降低锁的范围；锁的执行效率

消息数据写入存储层保证数据的串行写入，以及数据文件和索引文件严格顺序需要加锁处理，这里可能成为制约系统吞吐量的瓶颈。大体两方面的思路：

1. 程序内部优化，降低锁块中处理耗时，将耗时的操作做到状态无关避免在锁块中执行。（如MsgId的生成）
2. 单机多实力部署，弱化锁的瓶颈。（或者虚拟化技术）

## 数据堆积

MassQ消费队列通过链表的方式组织，每次数据的写入都会更新前一条索引的next指向；每次读取都通过next指针找到下一条索引的位置，逻辑上相邻的两条索引分布在不相连的存储位置。所以当堆积的情形会产生大量的随机IO。

对比SSD vs SAS：





SSD上性能要优于SAS上执行效果，QPS，磁盘利用率，await都相对稳定。

## CPU利用率

降低CPU利用热点；MsgId生成，字符编解码

## 操作异常处理

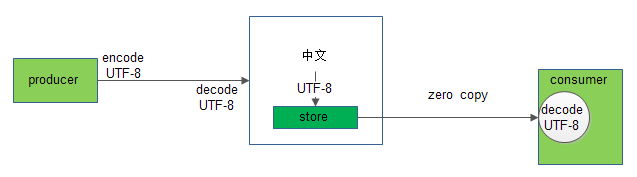
避免核心路劲异常结束，减少对象存活周期

## 数据复制HA

除了主备的方式外，是否还有其他方式保障系统的高可用？

## 编码

DMA方式避免数据拷贝，Broker和客户端编码保持一致



## 双向链表特性挖掘

如何实现高效查询

# MassQ适合的场景