



## 前言

---

最近花了我几个月的业余时间，对EQueue做了一个重大的改造，消息持久化采用本地写文件的方式。到现在为止，总算完成了，所以第一时间写文章分享给大家这段时间我所积累的一些成果。

1. EQueue开源地址: <https://github.com/tangxuehua/equeue>
2. EQueue相关文档: <http://www.cnblogs.com/netfocus/category/598000.html>
3. EQueue Nuget地址: <http://www.nuget.org/packages/equeue>

昨天，我写过一篇关于EQueue 2.0性能测试结果的文章，有兴趣的可以看看。

文章地址: <http://www.cnblogs.com/netfocus/p/4926305.html>

## 为什么要改为文件存储？

---

### SQL Server的问题

---

之前EQueue的消息持久化是采用SQL Server的。一开始我觉得没什么问题，采用的是异步定时批量持久化，使用SqlBulkCopy的方法，这个方法测试下来，批量插入消息的性能还不错，就决定使用了。一开始我并没有在使用到EQueue后做集成的性能测试。在功能上确实没什么问题了。而且使用DB持久化也有很多好处，比如消息查询很简单，DB天生支持各种方式的查询。删除消息也非常简单，一条DELETE语句即可。所以功能实现比较顺利。但后来当我对EQueue做性能测试时，发现一些问题。当数据库服务器和Broker本身部署在不同的服务器上时，持久化消息也会走网卡，消耗带宽，影响消息的发送和消费的TPS。而如果数据库服务器部署在Broker同一台服务器上，则因为SQLServer本身也会消耗CPU以及内存，也会影响Broker的消息发送和消费的TPS。另外SqlBulkCopy的速度，再本身机器正在接收大量的发送消息和拉取消息的请求时，会不太稳定。经过一些测试，发现整个EQueue Broker的性能不太理想。然后又想想，Broker服务器有一个硬件一直没有好好利用起来，那就是硬盘。假设我们的消息是持久化到本地硬盘的，顺序写文件，就应该能解决SQL Server的问题了。所以，开始调研如何实现文件持久化消息的方案了。

### 消息缓存在托管内存的GC的问题

---

之前消息存储在SQL Server，如果消费者每次读取消息时，总是从数据库去读取，那对数据库就是不断的写入和读取，性能不太理想。所以当初的思路是，尽量把最近可能要被消费的消息缓存在本地内存中。当初的做法是设计了一个很大的ConcurrentDictionary<long, Message>，这个字典就是存放了所有可能会被消费的消息。如果要消费的消息当前不在这个字典里，就批量从DB拉取一批出来消费。这个设计可以尽可能的避免读取DB的情况。但是带来了另一个问题。就是我们对这个字典在高并发不断的写入和读取。且这个字典里缓存的消息又很多，到达几百上千万时，GC的压力过大，导致很多线程都会被阻塞。严重影响Broker的TPS。

所以，基于上面的两个主要原因，我想到了两个思路来解决：1）采用写文件的方式来持久化消息；2）使用非托管内存来缓存将要被消费的消息；下面我们来看看这两个设计的一些关键问题的设计思路。

### 文件存储的关键问题设计

---

之前一直无法驾驭写文件的设计。因为精细化的将数据写入文件，并能要精确的读取想要的的数据，实在没什么经验。之前虽然也知道阿里的RocketMQ的消息持久化也是采用顺序写文件的方式的，但是看了代码，发现设计很复杂，一下子也比较难懂。尝试看了多次也无法完全理解。所以一直无法掌握这种方式。有一天不经意间想到之前看过的EventStore这个开源项目中，也有写文件的设计。这个项目是CQRS架构之父greg young所主导的开源项目，是一个专门为ES（Event Sourcing）设计模式中提供保存事件流支持的事件流存储系统。于是下定决心专研其源码，看C#代码肯定还是比Java容易，呵呵。经过一段时间的摸索之后，基本学到了它是如何写文件以及如何读文件的。了解了很多设计思路。然后，在看懂了EventStore的文件存储设计之后，再去看RocketMQ的文件持久化的设计，发现惊人的相似。原来看不懂的代码现在也能看懂了，因为思路差不多的。所以，这给我开始动手提供了很大的信心。经过自己的一些准备（文件读写的性能验证）和设计思路整理后，终于开始动手了。

### 如何写消息到文件？

其实说出来也很简单。之前一直以为写文件就是一个消息一行呗。这样当我们要找哪个消息时，只需要知道行号即可。确实，理论上这样也挺好。但上面这两个开源项目都不是这样做的，而是都是采用更精细化的直接写二进制的方式。搞清楚写入的格式之后，还要考虑一个文件写不下的时候怎么办？因为一个文件总是有大小的，比如1G，那超过1G后，必然要创建新的文件，然后把消息写入新的文件。所以，我们就又有了Chunk的概念。一个Chunk就是一个文件，假设我们现在实现了一个FileMessageStore，表示对文件持久化的封装，那这个FileMessageStore肯定维护了一堆的Chunk。然后我们也很容易想到一点，就是Chunk有3种状态：1）New，表示刚创建的Chunk，这种Chunk我们可以写入新消息进去；2）Completed，已写入完成的Chunk，这种Chunk是只读的；3）OnGoing的Chunk，就是当FileMessageStore初始化时，要从磁盘的某个chunk的目录下加载所有的Chunk文件，那不难理解，最后一个文件之前的Chunk文件应该都是Completed的；最后一个Chunk文件可能写入了一半，就是之前没完全用完的。所以，本质上New和Ongoing的Chunk其实是一样的，只是初始化的方式不同。

至此，我们知道了写文件的两个关键思路：1）按二进制写；2）拆分为Chunk文件，且每个Chunk文件有状态；按二进制写主要的思路是，假如我们当前要写入的消息的二进制数组大小为100个字节，也就是说消息的长度为100，那我们可以先把消息的长度写入文件，再接着写入消息本身。这样我们读取消息时，只要知道了写入消息长度时的那个Position，就能先读取到消息的长度，然后就能知道接下来要读取多少字节为消息内容。从而能正确读取消息出来。

另外再分享一点，EventStore中，写入一个事件到文件中时，还会在写入消息内容后再写入这个消息的长度到文件里。也就是说，写入一个数据到文件时，会在头尾都写入该数据的长度。这样做的好处是什么呢？就是当我们想从后往前读数据时，也能方便的做到，因为每个数据的前后都记录了该数据的长度。这点应该不难理解吧？而EventStore是一个面向流的存储系统，我们对事件流确实可能从前往后读，也可能是从后往前读。另外这个设计还有一个好处，就是起到了校验数据合法性的目的。当我们根据长度读取数据后，再数据之后再读取一个长度，如果这两个长度一致，那数据应该就没问题的。在RocketMQ中，是通过CRC校验的方式来保证读取的数据没有问题。我个人还是比较喜欢EventStore的做法。所以EQueue里现在写入数据就是这样做的。

上面我介绍了一种写入不定长数据到文件的设计思路，这种设计是为了解决写入消息到文件的情况，因为消息的长度是不定的。在EQueue中，我们还有一另一种写文件的场景。就是队列信息的持久化。EQueue的架构是一个Topic下有多个Queue，每个Queue里有很多消息，消费者负载均衡是通过给消费者分配均匀数量的Queue的方式来达到的。这样我们只要确保写入Queue的消息是均匀的，那每个Consumer消费到的消息数就是均匀的。那

一个Queue里记录的是什么呢？就是一个消息和其在队列的位置的对应关系。假设消息写入在文件的物理位置为10000，然后这个消息在Queue里的索引是100，那这个队列就会把这两个位置对应起来。这样当我们要消费这个Queue中索引为100的消息时，就能找到这个消息在文件中的物理位置为10000，就能根据这个位置找到消息的内容了。如果是托管内存，我们只需要弄一个Dictionary，key是消息在队列中的Offset，value是消息在文件中的物理Offset即可。这样我们有了这个dict，就能轻松建立起对应关系了。但上面我说过，这种巨大的dict是要占用内存的，会有GC的问题。所以更好的办法是，把这个对应关系也写入文件，那怎么做呢？这时就又需要更精细化的设计了。想到了其实也很简单，这个设计我是从RocketMQ中学到的。就是我们设计一种固定长度的结构体，这个结构体里就存放一个数据，就是消息在文件的物理位置（为了后面好表达，我命名为MessagePosition），一个Long值，一个Long的长度是8个字节。也就是说，这个文件中，每个写入的数据的长度都是8个字节。假设我们一个文件要保存100W个MessagePosition。那这个文件的长度就是100W \* 8这么多字节，大概为7.8MB。那么这样做有什么好处呢？好处就是，假如我们现在要消费这个Queue里的第一个消息，那这个消息的MessagePosition在这个文件中的位置0，第二个消息在这个文件中的位置是8，第三个就是16，以此类推，第N个消息就是(N-1) \* 8。也就是说，我们无须显式的把消息在队列中的位置信息也写入到文件，而是通过这样的固定算法，就能精确的算出Queue中某个消息的MessagePosition是写入在文件的哪个位置。然后拿到了MessagePosition之后，就能从Message的Chunk文件中读取到这个消息了。

通过上面的分析，我们知道了，Producer发送一个消息到Broker时，Broker会写两次磁盘。一次是现将消息本身写入磁盘（Message Chunk里），另一次是将消息的写入位置写入到磁盘（Queue Chunk里）。细心的朋友可能会问，假如我第一次写入成功，但第二次写入时失败，比如正好机器断电或者当前Broker服务器正好出啥问题了，没有写入成功。那怎么办呢？这个没有什么大的影响。因为首先这种情况会被认为是消息发送失败。所以Producer还会重新发送该消息，然后Broker收到消息后还会再做一次这两个写入操作。也就是说，第一次写入的消息内容永远也不会用到了，因为那个写入位置永远也不会在Queue Chunk里有记录。

下面的代码展示了写消息到文件的核心代码：



```
//消息写文件需要加锁，确保顺序写文件
MessageStoreResult result = null;
lock (_syncObj)
{
    var queueOffset = queue.NextOffset;
    var messageRecord = _messageStore.StoreMessage(queueId, queueOffset, message);
    queue.AddMessage(messageRecord.LogPosition, message.Tag);
    queue.IncrementNextOffset();
    result = new MessageStoreResult(messageRecord.MessageId, message.Code,
    message.Topic, queueId, queueOffset, message.Tag);
}
```



StoreMessage方法内部实现：



```
public MessageLogRecord StoreMessage(int queueId, long queueOffset, Message message)
{
    var record = new MessageLogRecord(
        message.Topic,
```

```

        message.Code,
        message.Body,
        queueId,
        queueOffset,
        message.CreatedTime,
        DateTime.Now,
        message.Tag);
    _chunkWriter.Write(record);
    return record;
}

```



queue.AddMessage方法的内部实现:

```

public void AddMessage(long messagePosition, string messageTag)
{
    _chunkWriter.Write(new QueueLogRecord(messagePosition + 1,
        messageTag.GetHashCode2()));
}

```

ChunkWriter的内部实现:



```

public long Write(ILogRecord record)
{
    lock (_lockObj)
    {
        if (_isClosed)
        {
            throw new ChunkWriteException(_currentChunk.ToString(), "Chunk writer is
closed.");
        }

        //如果当前文件已经写完,则需要新建文件
        if (_currentChunk.IsCompleted)
        {
            _currentChunk = _chunkManager.AddNewChunk();
        }

        //先尝试写文件
        var result = _currentChunk.TryAppend(record);

        //如果当前文件已满
        if (!result.Success)
        {
            //结束当前文件
            _currentChunk.Complete();

            //新建新的文件
            _currentChunk = _chunkManager.AddNewChunk();
        }
    }
}

```

```

        //再尝试写入新的文件
        result = _currentChunk.TryAppend(record);

        //如果还是不成功，则报错
        if (!result.Success)
        {
            throw new ChunkWriteException(_currentChunk.ToString(), "Write record to
chunk failed.");
        }
    }

    //如果需要同步刷盘，则立即同步刷盘
    if (_chunkManager.Config.SyncFlush)
    {
        _currentChunk.Flush();
    }

    //返回数据写入位置
    return result.Position;
}
}

```



当然，我上面为了简化问题的复杂度。所以没有引入关于如何根据某个全局的**MessagePosition**找到其在哪个**Message Chunk**的问题。这个其实也很好做，我们首先固定好每个**Message Chunk**文件的大小。比如大小为**256MB**，然后我们为每个**Chunk**文件设计一个**ChunkHeader**，每个**Chunk**文件总是先把这个**ChunkHeader**写入文件，这个**Header**里记录了这个文件的起始位置和结束位置，以及文件的大小。这样我们根据某个**MessagePosition**计算其在哪个**Chunk**文件时，只需要把这个**MessagePositon**对**Chunk**的大小做取模操作即可。根据数据的位置找其在哪个**Chunk**的代码看起来如下面这样这样：

```

public Chunk GetChunkFor(long dataPosition)
{
    var chunkNum = (int)(dataPosition / _config.GetChunkDataSize());
    return GetChunk(chunkNum);
}

public Chunk GetChunk(int chunkNum)
{
    if (_chunks.ContainsKey(chunkNum))
    {
        return _chunks[chunkNum];
    }

    return null;
}

```



代码很简单，就不多讲了。拿到了**Chunk**对象后，我们就可以把**dataPosition**传给**Chunk**，然后**Chunk**内部把这



个全局的`dataPosition`转换为本地的一个位置，就能准确的定位到这个数据在当前`Chunk`文件的实际位置了。将全局位置转换为本地的位置的算法也很简单直接：



```
public int GetLocalDataPosition(long globalDataPosition)
{
    if (globalDataPosition < ChunkDataStartPosition || globalDataPosition >
        ChunkDataEndPosition)
    {
        throw new Exception(string.Format("globalDataPosition {0} is out of chunk data
positions [{1}, {2}].",
            globalDataPosition, ChunkDataStartPosition, ChunkDataEndPosition));
    }
    return (int)(globalDataPosition - ChunkDataStartPosition);
}
```



只需要把这个全局的位置减去当前`Chunk`的数据开始位置，就能知道这个全局位置相对于当前`Chunk`的本地位置了。

好了，上面介绍了消息如何写入的主要思路以及如何读取数据的思路。

另外一点还想提一下，就是关于刷盘的策略。一般我们写数据到文件后，是需要调用文件流的`Flush`方法的，确保数据最终刷入到了磁盘上。否则数据就还是在缓冲区里。当然，我们需要注意到，即便调用了`Flush`方法，数据可能也还没真正逻辑到磁盘，而只是在操作系统内部的缓冲区里。这个我们就无法控制了，我们能做到的是调用了`Flush`方法即可。那当我们每次写入一个数据到文件都要调用`Flush`方法的话，无疑性能是低下的，所以就有了所谓的异步刷盘的设计。就是我们写入消息后不立即调用`Flush`方法，而是采用一个独立的线程，定时调用`Flush`方法来实现刷盘。目前`EQueue`支持同步刷盘和异步刷盘，开发者可以自己配置决定采用哪一种。异步刷盘的间隔默认是`100ms`。当我们在追求高吞吐量时，应该考虑异步刷盘，但要求数据可靠性更高但对吞吐量可以低一点时，则可以使用同步刷盘。如果又要高吞吐又要数据高可靠，那就只有一个办法了，呵呵。就是多增加一些`Broker`机器，通过集群来弥补单台`Broker`写入数据的瓶颈。

## 如何从文件读取消息？

假设我们现在要从一个文件读取数据，且是多线程并发的读取，要怎么设计？一个办法是，每次读取时，创建文件流，然后创建`StreamReader`，然后读取文件，读取完成后释放`StreamReader`并关闭文件流。但每次要读取文件的一个数据都要这样做的话性能不是太好，因为我们反复的创建这样的对象。所以，这里我们可以使用对象池的概念。就是`Chunk`内部，预先创建好一些`Reader`，当需要读文件时，获取一个可用的`Reader`，读取完成后，再把`Reader`归还到对象池里。基于这个思路，我设计了一个简单的对象池：



```
private readonly ConcurrentQueue<ReaderWorkItem> _readerWorkItemQueue = new
ConcurrentQueue<ReaderWorkItem>();
private void InitializeReaderWorkItems()
{
    for (var i = 0; i < _chunkConfig.ChunkReaderCount; i++)
```

```

    {
        _readerWorkItemQueue.Enqueue(CreateReaderWorkItem());
    }

    _isReadersInitialized = true;
}

private ReaderWorkItem CreateReaderWorkItem()
{
    var stream = default(Stream);
    if (_isMemoryChunk)
    {
        stream = new UnmanagedMemoryStream((byte*)_cachedData, _cachedLength);
    }
    else
    {
        stream = new FileStream(_filename, FileMode.Open, FileAccess.Read,
        FileShare.ReadWrite, _chunkConfig.ChunkReadBuffer, FileOptions.None);
    }

    return new ReaderWorkItem(stream, new BinaryReader(stream));
}

private ReaderWorkItem GetReaderWorkItem()
{
    ReaderWorkItem readerWorkItem;
    while (!_readerWorkItemQueue.TryDequeue(out readerWorkItem))
    {
        Thread.Sleep(1);
    }
    return readerWorkItem;
}

private void ReturnReaderWorkItem(ReaderWorkItem readerWorkItem)
{
    _readerWorkItemQueue.Enqueue(readerWorkItem);
}

```



当一个Chunk初始化时，我们预先初始化好固定数量（可配置）的Reader对象，并把这些对象放入一个ConcurrentQueue里（对象池的作用），然后要读取数据时，从ConcurrentQueue里拿一个可用的Reader即可，如果当前并发太高拿不到怎么办，就等待直到拿到为止，目前我是等待1ms后继续尝试拿，直到最后拿到为止。然后ReturnReaderWorkItem就是数据读取完之后归还Reader到对象池。就是不是很简单哦。这样的设计，可以避免不断的创建文件流和Reader对象，可以避免GC的副作用。

## Broker重启时如何做？

大家知道，当Broker重启时，我们是需要扫描磁盘上Chunk目录下的所有Chunk文件的。那怎么扫描呢？上面其实我也简单提到过。首先，我们可以对每个Chunk文件的文件名的命名定义一个规则，第一个Chunk文件的文件名比如为：message-chunk-000000000，第二个为：message-chunk-000000001，以此类推。那我们扫描时，只要先把所有的文件名获取到，然后对文件名升序排序。那最后一个文件之前的文件肯定都是写入完全了的，即上面我说的Completed状态的，而最后一个文件是还没有写入完的，还可以接着写。所以我们初始化时，只需要先初始化最后一个之前的所有Chunk文件，最后再初始化最后一个文件即可。这里我所说的初始化不是要把整个Chunk文件的内容都加载到内存，而是只是读取这个文件的ChunkHeader的信息维护在内存即可。有了

Header信息，我们就可以为后续的数据读取提供位置计算了。所以，整个加载过程是很快的，读取100个Chunk文件的ChunkHeader也不过一两秒的时间，完全不影响Broker的启动时间。对于初始化Completed的Chunk比较简单，只需要读取ChunkHeader信息即可。但是初始化最后一个文件比较麻烦，因为我们还要知道这个文件当前写入到哪里了？从而我们可以从这个位置的下一个位置接着往下写。那怎么知道这个文件当前写入到哪里了呢？其实比较复杂。有很多技术，我看到RocketMQ和EventStore这两个开源项目都采用了Checkpoint的技术。就是当我们每次写入一个数据到文件后，都会更新一下Checkpoint，即表示当前写入到这个文件的哪里了。然后这个Checkpoint值我们也是定时异步保存到某个独立的小文件里，这个文件里只保存了这个Checkpoint。这样的设计有一个问题，就是假如数据写入了，但由于Checkpoint的保存不是实时的，所以理论上会出现Checkpoint值会小于实际文件写入的位置的情况。一般我们忽略这种情况即可，即可能会存在初始化时，下次写入可能会覆盖一定的之前已经写入的数据，因为Checkpoint可能是稍微老一点的。

而我在设计时，希望能再严谨一点，取消Checkpoint的设计，而是采用在初始化Ongoing状态的Chunk文件时，从文件的头开始不断往下读，当最后无法往下读时，我们就知道这个文件我们当前写入到哪里了。那怎么知道无法往下读了呢？也就是说怎么确定后续的文件内容不是我们写入的？也很简单。对于不固定数据长度的Chunk来说，由于我们每次写入一个数据时都是同时在前后写入这个数据的长度；所以我们再初始化读取这个文件时，可以借助这一点来校验，但出现不符合这个规则的数据时，就认为后续不是正常的数据了。对于固定长度的Chunk来说，我们只要保证每次写入的数据的数据是非0了。而对于EQueue的场景，固定数据的Chunk里存储的都是消息在Message Chunk中的全局位置，一个Long值；但这个Long值我们正常是从0开始的，怎么办呢？很简单，我们写入MessagePosition时，总是加1即可。即假如当前的MessagePosition为0，那我们实际写入1，如果为100，则实际写入的值是101。这样我们就能确保这个固定长度的Chunk文件里每个数据都是非0的。然后我们在初始化这样的Chunk文件时，只要不断读取固定长度（8个字节）的数据，当出现读取到的数据为0时，就认为已经到头了，即后续的不是我们写入的数据了。然后我们就能知道接下来要从哪里开始读取了哦。

## 如何尽量避免读文件？

上面我介绍了如何读文件的思路。我们也知道了，我们是在消费者要消费消息时，从文件读取消息的。但对从文件读取消息总是没有比从内存读取消息来的快。我们前面的设计都没有把内存好好利用起来。所以我们能否考虑把未来可能要消费的Chunk文件的内容直接缓存在内存呢？这样我们就可以避免对文件的读取了。肯定可以的。那怎么做呢？前面我提得多，曾经我们用托管内存中的ConcurrentDictionary<long, Message>这样的字典来缓存消息。我也提到这会带来垃圾回收而影响性能的问题。所以我们不能直接这样简单的设计。经过我的一些尝试，以及从EventStore中的源码中学到的，我们可以使用非托管内存来缓存Chunk文件。我们可以使用Marshal.AllocHGlobal来申请一块完整的非托管内存，然后再需要释放时，通过Marshal.FreeHGlobal来释放。然后，我们可以通过UnmanagedMemoryStream来访问这个非托管内存。这个是核心思路。那么怎样把一个Chunk文件缓存到非托管内存呢？很简单了，就是扫描这个文件的所有内容，把内容都写入内存即可。代码如下：



```
private void LoadFileChunkToMemory()
{
    using (var fileStream = new FileStream(_filename, FileMode.Open, FileAccess.Read,
        FileShare.ReadWrite, 8192, FileOptions.None))
    {
        var cachedLength = (int)fileStream.Length;
        var cachedData = Marshal.AllocHGlobal(cachedLength);
```



```

        try
        {
            using (var unmanagedStream = new UnmanagedMemoryStream((byte*) cachedData,
cachedLength, cachedLength, FileAccess.ReadWrite))
            {
                fileStream.Seek(0, SeekOrigin.Begin);
                var buffer = new byte[65536];
                int toRead = cachedLength;
                while (toRead > 0)
                {
                    int read = fileStream.Read(buffer, 0, Math.Min(toRead,
buffer.Length));

                    if (read == 0)
                    {
                        break;
                    }

                    toRead -= read;
                    unmanagedStream.Write(buffer, 0, read);
                }
            }
        }
        catch
        {
            Marshal.FreeHGlobal(cachedData);
            throw;
        }

        _cachedData = cachedData;
        _cachedLength = cachedLength;
    }
}

```




代码很简单，不用多解释了。需要注意的是，上面这个方法针对的是**Completed**状态的**Chunk**，即已经写入完成的**Chunk**的。已经写入完全的**Chunk**是只读的，不会再发生更改，所以我们可以随便缓存在内存中。

那对于新创建出来的**Chunk**文件呢？正常情况下，消费者来得及消费时，我们总是在不断的写入最新的**Chunk**文件，也在不断的从这个最新的**Chunk**文件读取消息。那我们怎么确保消费最新的消息时，也不需要从文件读取呢？也很简单，就是在新建一个**Chunk**文件时，如果内存足够，也同时创建一个一样大小的基于非托管内存的**Chunk**。然后我们再写入消息到文件**Chunk**成功后，再同时写入这个消息到非托管内存的**Chunk**。这样，我们在消费消息，读取消息时总是首先判断当前**Chunk**是否关联了一个非托管内存的**Chunk**，如果有，就优先从内存读取即可。如果没有才从文件**Chunk**读取。


但是从文件读取时，可能会遇到一个问题。就是我们刚写入到文件的数据可能无法立即读取到。因为写入的数据没有立即刷盘，所以无法通过**Reader**读取到。所以，我们不能仅通过判断当前写入的位置来判断当前是否还有数据可以被读取，而是考虑当前的最后一次刷盘的位置。理论上只能读取刷盘之前的数据。但即便这样设计了，在如果当前硬盘不是**SSD**的情况下，好像也会出现读不到数据的问题。偶尔会报错，有朋友在测试时已经遇到了这样的问题。那怎么办呢？我想了一个办法。因为这种情况归根接地还是因为我们逻辑上认为已经写入到文件的数据由于

未及时刷盘或者操作系统本身的内部缓存的问题，导致数据未能及时写入磁盘。出现这种情况一定是最近的一些数据。所以我们如果能够把比如最近写入的**10000**（可配置）个数据都缓存在本地托管内存中，然后读取时先看本地缓存的托管内存中有没有这个位置的数据，如果有，就不需要读文件了。这样就能很好的解决这个问题了。那怎么确保我们只缓存了最新的**10000**个数据且不会超出**10000**个呢？答案是环形队列。这个名字听起来很高大上，其实就是一个数组，数组的长度为**10000**，然后我们在写入数据时，我们肯定知道这个数据在文件中的位置的，我们可以把这个位置（一个long值）对**10000**取摸，就能知道该把这个数据缓存在这个数组的哪个位置了。通过这个设计确保缓存的数据不会超过**10000**个，且确保一定只缓存最新的数据，如果新的数据保存到数组的某个下标时，该下标已经存在以前已经保存过的数据了，就自动覆盖掉即可。由于这个数组的长度不是很长，所以每什么GC的问题。

但是光这样还不够，我们这个数组中的每个元素至少要记录这个元素对应的数据在文件中的位置。这个是为了我们在从数组中获取到数据后，进一步校验这个数据是否是我想要的那个位置的数据。这点大家应该可以理解的吧。下面这段代码展示了如何从环形数组中读取想要的数据：



```
if (_cacheItems != null)
{
    var index = dataPosition % _chunkConfig.ChunkLocalCacheSize;
    var cacheItem = _cacheItems[index];
    if (cacheItem != null && cacheItem.RecordPosition == dataPosition)
    {
        var record = readRecordFunc(cacheItem.RecordBuffer);
        if (record == null)
        {
            throw new ChunkReadException(
                string.Format("Cannot read a record from data position {0}. Something is seriously wrong in chunk {1}.",
                    dataPosition, this));
        }
        if (_chunkConfig.EnableChunkReadStatistic)
        {
            _chunkStatisticService.AddCachedReadCount(ChunkHeader.ChunkNumber);
        }
        return record;
    }
}
```



`_cacheItems`是当前Chunk内的一个环形数组，然后假如当前我们要读取的数据的位置是`dataPosition`，那我们只需要先对环形数据的长度取摸，得到一个下标，即上面代码中的`index`。然后就能从数组中拿到对应的数据了，然后如果这个数据存在，就进一步判断这个数据`dataPosition`是否和要求的`dataPosition`，如果一致，我们就能确定这个数据确实是我们想要的数据了。就可以返回了。

所以，通过上面的两种缓存（非托管内存+托管内存环形数组）的设计，我们可以确保几乎不用再从文件读取消息了。那什么时候还是会从文件读取呢？就是在1）内存不够用了；2）当前要读取的数据不是最近的**10000**个；这两个前提下，才会从文件读取。一般我们线上服务器，肯定会保证内存是可用的。`EQueue`现在有两个内存使用的水位。一个是当物理内存使用到多少百分比（默认值为**40%**）时，开始清理已经不再活跃的Chunk文件的非托管

内存Chunk；那什么是不活跃呢？就是在最近5s内没有发生过读写的Chunk。这个设计我觉得是非常有效的，因为假如一个Chunk有5s没有发生过读写，那一般肯定是没有消费者在消费它了。另一个水位是指，最多EQueue Broker最多使用物理内存的多少百分比（默认值为75%），这个应该好理解。这个水位是为了保证EQueue不会把所有物理内存都吃光，是为了确保服务器不会因为内存耗尽而宕机或导致服务不可用。

那什么时候会出现大量使用服务器内存的情况呢？我们可以推导出来的。正常情况下，消息写入第一个Chunk，我们就在读取第一个Chunk；写入第二个Chunk我们也会跟着读取第二个Chunk；假设当前写入到了第10个Chunk，那理论上前面的9个Chunk之前缓存的非托管内存都可以释放了。因为肯定超过5s没有发生读写了。但是假如现在消费者有很多，且每个消费者的消费进度都不同，有些很快，有些很慢，当所有的消费者的消费进度正好覆盖到所有的Chunk文件时，就意味着每个Chunk文件都在发生读取。也就是说，每个Chunk都是活跃的。那此时就无法释放任何一个Chunk的非托管内存了。这样就会导致占用大量非托管内存了。但由于75%的水位的设计，Broker内存的使用是不会超过物理内存75%的。在创建新的Chunk或者尝试缓存一个Completed的Chunk时，总是会判断当前使用的物理内存是否已经超过75%，如果已经超过，就不会分配对应的非托管内存了。

## 如何删除消息？

---

删除消息的设计比较简单。主要的思路是，当我们的消息已经被所有的消费者都消费过了，且满足我们的删除策略了，就可以删除了。RocketMQ删除消息的策略比较粗暴，没有考虑消息是否经被消费，而是直接到了一定的时间就删除了，比如最多只保留2天。这个是RocketMQ的设计。EQueue中，会确保消息一定是被所有的消费者都消费了才会考虑删除。然后目前我设计的删除策略有两种：

1. 按Chunk文件数；即设计一个阈值，表示磁盘上最多保存多少个Chunk文件。目前默认值为100，每个Chunk文件的大小为256MB。也就是大概总磁盘占用25G。一般我们的硬盘肯定有25G的。当我们不关心消息保存多久而只从文件数的角度来决定消息是否要删除时，可以使用这个策略；
2. 按时间来删除，默认是7天，即当某个Chunk是7天前创建的，那我们就可以创建了。这种策略是不关心Chunk总共有多少，完全根据时间的维度来判断。

实际上，应该可能还有一些需求希望能把两个策略合起来考虑的。这个目前我没有做，我觉得这两种应该够了。如果大家想做，可以自己扩展的。

另外，上面我说过EQueue中目前有两种Chunk文件，一种是存储消息本身的，我叫做Message Chunk；一种是存储队列信息的，我叫做Queue Chunk；而Queue Chunk的数据是依赖于Message Chunk的。上面我说的两种删除策略是针对Message Chunk而言的。而Queue Chunk，由于这个依赖性，我觉得比较合理的方式是，只需要判断当前Queue Chunk中的所有的消息对应的Message Chunk是否已经都删除了，如果是，难说明这个Queue Chunk也已经没意义了，就可以删除了。但只要这个Queue Chunk中至少还有一个消息的Chunk文件没删除，那这个Queue Chunk就不会删除。

上面这个只是思路哦，真实的代码肯定比这个复杂，呵呵。有兴趣的朋友还是要看代码的。

## 如何查消息？

---

之前用SQL Server的方式，由于DB很容易查消息，所以查询消息不是大问题。但是现在的消息是放在文件里的，那要怎么查询呢？目前支持根据消息ID来查询。当Producer发送一个消息到Broker，Broker返回结果里会包含消息的ID。Producer的正确做法应该是要用日志或其他方式记录这个ID，并最好和自己的当前业务消息的某个业务ID一起记录，比如CommandId或者EventId，这样我们就能根据我们的业务ID找到消息ID，然后根据消息ID找到消息内容了。那消息ID现在是怎么设计的呢？也是受到RocketMQ的启发，消息ID由两部分组成：

1) Broker的IP; 2) 消息在Broker的文件中的全局位置; 这样, 当我们要根据某个消息ID查询时, 就可以先定位到这个消息在哪个Broker上, 也同时知道了消息在文件的哪个位置了, 这样就能最终读取这个消息的内容了。

为什么要这样设计呢? 如果我们的Broker没有集群, 那其实不需要包含Broker的IP; 这个设计是为了未来EQueue Broker会支持集群的, 那个时候, 我们就必须要知道某个消息ID对应的Broker是哪个了。

## 如何保存队列消费进度?

EQueue中, 每个Queue, 都会有一个对应的Consumer。消费进度就是这个Queue当前被消费到哪里了, 一个Offset值。比如Offset为100, 就表示当前这个Queue已经消费到第99 (因为是从0开始的) 个位置的消息了。因为一个Broker上有很多的Queue, 比如有100个。而我们现在是使用文件的方式来存储信息了。所以自然消费进度也是用文件了。但由于消费进度的信息很少, 也不是递增的形式。所以我们可以简单设计, 目前EQueue采用一个文件的方式来保存所有Queue的消费进度, 文件内容为JSON, JSON里记录了每个Queue的消费进度。文件内容看起来像下面这样:

```
{"SampleGroup":{"topic1-3":89055,"topic1-2":89599,"topic1-1":89471,"topic1-0":89695}}
```

上面的JSON标识一个名为SampleGroup的ConsumerGroup, 他消费了一个名为topic1的topic, 然后这个topic下的每个Queue的消费进度记录了下来。如果有另一个ConsumerGroup, 也消费了这个topic, 那消费进度是隔离的。如果还不清楚ConsumerGroup的同学, 要去看一下我之前写的EQueue的文章了。

## 还有没有可以优化的地方?

到目前为止, 还有没有其他可优化的大的地方呢? 有。之前我做EQueue时, 总是把消息从数据库读取出来, 然后构造出消息对象, 再把消息对象序列化为二进制, 再返回给Consumer。这里涉及到从DB拿出来, 再序列化为二进制。学习了RocketMQ的代码后, 我们可以做的更聪明一点。因为其实基于文件存储时, 我们从文件里拿出来已经是二进制了。所以可以直接把二进制返回给消费者即可。不需要先转换为对象再做序列化了。通过这个设计的改进, 我们现在的消费者消费消息, 可以说无任何瓶颈了, 非常快。

## 如何统计消息读写情况?

在测试写文件的这个版本时, 我们很希望知道每个Chunk的读写情况的统计, 从而确定设计是正确的。所以, 我给EQueue的Chunk增加了实时统计Chunk读写的统计服务。目前我们在运行EQueue自带的例子时, Broker会每个一秒打印出所有Chunk的读写情况, 这个特性极大的方便我们判断消息的发送和消费是否正常, 消费是否有延迟等。

## 其他新增功能

### 更完善和安全的队列扩容和缩容设计

这次我给EQueue的Web后台管理控制台也完善了一下队列的增加和减少的设计。增加队列 (即队列的扩容) 比较简单, 直接新增即可。但是当我们要删除一个队列时, 怎样安全的删除呢? 主要是要确保删除这个队列时, 已经没有Producer或Consumer在使用这个队列了。要怎么做到呢? 我的思路是, 为每个Queue对象设计两个属性, 表示对Producer是否可见, 对Consumer是否可见。当我们要删除某个Queue时, 可以: 1) 先让其对Producer不可见, 这样Producer后续就不会再发送新的消息到这个队列了; 然后等待, 直到这个队列里的消息都被所有的消费者消费掉了; 然后再设置为对Consumer不可见。然后再过几秒, 确保每个消费者都不会再向这

个队列发出拉取消息的请求了。这样我们就能安全的删除这个队列了。删除队列的逻辑大概如如下：



```
public void DeleteQueue(string topic, int queueId)
{
    lock (this)
    {
        var key = QueueKeyUtil.CreateQueueKey(topic, queueId);
        Queue queue;
        if (!_queueDict.TryGetValue(key, out queue))
        {
            return;
        }

        //检查队列对Producer或Consumer是否可见，如果可见是不允许删除的
        if (queue.Setting.ProducerVisible || queue.Setting.ConsumerVisible)
        {
            throw new Exception("Queue is visible to producer or consumer, cannot be delete.");
        }
        //检查是否有未消费完的消息
        var minConsumedOffset = _consumeOffsetStore.GetMinConsumedOffset(topic, queueId);
        var queueCurrentOffset = queue.NextOffset - 1;
        if (minConsumedOffset < queueCurrentOffset)
        {
            throw new Exception(string.Format("Queue is not allowed to delete, there are not consumed messages: {0}", queueCurrentOffset - minConsumedOffset));
        }

        //删除队列的消费进度信息
        _consumeOffsetStore.DeleteConsumeOffset(queue.Key);

        //删除队列本身，包括所有的文件
        queue.Delete();

        //最后将队列从字典中移除
        _queueDict.Remove(key);
    }
}
```



代码应该很简单直接，不多解释了。队列的动态新增和删除，可以方便我们线上应付在线活动时，随时为消费者提供更高的并行消费能力，以及活动结束后去掉多余的队列。是非常实用的功能。



## 支持Tag功能

---

这个功能，也是非常实用的。这个版本我加了上去。以前EQueue只有Topic的概念，没有Tag的概念。Tag是对Topic的二级过滤。比如当某个Producer发送了3个消息，Topic都是topic，然后tag分别是01,02,03。然后Consumer订阅了这个Topic，但是订阅这个Topic时同时制定了Tag，比如指定为02，那这个Consumer就只会收到一个消息。Tag为01,03的消息是不会收到的。这个就是Tag的功能。我觉得Tag对我们是非常有用的，它可以极大的减少我们定义Topic。本来我们必须定义一个新的Topic时，现在可能只需要定义一个Tag即可。关于Tag的实现，我就不展开了。

## 支持消息堆积报警

---

终于到最后一点了，终于坚持快写完了，呵呵。EQueue Web后台管理控制台现在支持消息堆积的报警了。当EQueue Broker上当前所有未消费的消息数达到一定的阈值时，就会发送邮件进行报警。我们可以把我们的邮件和我们的手机短信进行绑定，比如移动的139邮箱我记得就有这个功能。这样我们就能第一时间知道Broker上是否有大量消息堆积了，可以让我们第一时间处理问题。

## 结束语

---

这篇文章感觉是我有史以来写过的最有干货的一篇了，呵呵。一气呵成，也是对我前面几个月的所有积累知识经验的一次性释放吧。希望能给大家一些帮助。我写文章比较喜欢写思路，不太喜欢介绍如何用。我觉得一个程序员，最重要的是要学会如何思考去解决自己想解决的问题。而不是别人直接告诉你如何去解决。通过做EQueue这个分布式消息队列，也算是我自己的一个实践过程。我非常鼓励大家写开源项目哦，当你专注于实现某个你感兴趣的开源项目时，你就会有目标性的去学习相关的知识，你的学习就不会迷茫，不会为了学技术而学技术了。我在做EQueue时，要考虑各种东西，比如通信层的设计、消息持久化、整个架构设计，等等。我觉得是非常锻炼人的。一个人时间短暂，如果能用有限的时间做出好的东西可以造福后人，那我们来到这个世上也算没白来了，你说对吗？所以，我们千万不要放弃我们的理想，虽然坚持理想很难，但也要坚持。

分类: [EQueue](#)