Kafka水位(high watermark)与leader epoch的讨论

~~~这是一篇有点长的文章，希望不会令你昏昏欲睡~~~

本文主要讨论0.11版本之前Kafka的副本备份机制的设计问题以及0.11是如何解决的。简单来说，0.11之前副本备份机制主要依赖水位(或水印)的概念，而0.11采用了leader epoch来标识备份进度。后面我们会详细讨论两种机制的差异。不过首先先做一些基本的名词含义解析。

水位或水印（watermark）一词，也可称为高水位(high watermark)，通常被用在流式处理领域（比如Apache Flink、Apache Spark等），以表征元素或事件在基于时间层面上的进度。一个比较经典的表述为：流式系统保证在水位t时刻，创建时间（event time） = t'且t' ⩽ t的所有事件都已经到达或被观测到。在Kafka中，水位的概念反而与时间无关，而是与位置信息相关。严格来说，它表示的就是位置信息，即位移（offset）。网上有一些关于Kafka watermark的介绍，本不应再赘述，但鉴于本文想要重点强调的leader epoch与watermark息息相关，故这里再费些篇幅阐述一下watermark。注意：由于Kafka源码中使用的名字是高水位，故本文将始终使用high watermaker或干脆简称为HW。

Kafka分区下有可能有很多个副本(replica)用于实现冗余，从而进一步实现高可用。副本根据角色的不同可分为3类：

- leader副本：响应clients端读写请求的副本
- follower副本：被动地备份leader副本中的数据，不能响应clients端读写请求。

垃圾回收器(1)
Paxos(1)

随笔档案
2020年11月(1)
2020年9月(1)
2020年8月(2)
2020年7月(3)
2020年5月(1)
2020年4月(1)
2020年3月(3)
2020年2月(1)
2020年1月(1)
2019年12月(1)
2019年11月(1)
2019年10月(1)
2019年9月(2)
2019年8月(1)
2019年7月(1)
更多

相册
Common(1)
kafka(4)

阅读排行榜
1. Kafka消费组(consumer group)(157313)
2. Kafka 如何读取offset topic内容 (__consumer_offsets)(84409)
3. Kafka如何创建topic？(83566)
4. 【原创】如何确定Kafka的分区数、key和consumer线程数(67183)
5. Kafka与Flink集成(53170)

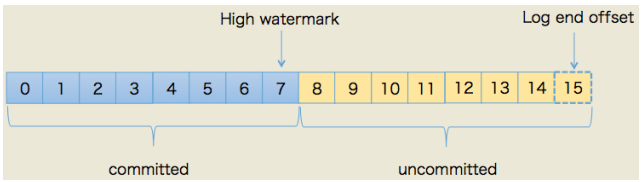
评论排行榜
1. Kafka消费组(consumer group)(53)
2. 【原创】Kafka Consumer多线程实例(53)
3. Kafka无消息丢失配置(45)
4. Kafka水位(high watermark)与leader epoch的讨论(44)
5. Kafka 如何读取offset topic内容 (__consumer_offsets)(41)

- ISR副本：包含了leader副本和所有与leader副本保持同步的follower副本——如何判定是否与leader同步后面会提到

每个Kafka副本对象都有两个重要的属性：LEO和HW。**注意是所有的副本，而不只是leader副本。**

- LEO：即日志末端位移(log end offset)，记录了该副本底层日志(log)中下一条消息的位移值。注意是下一条消息！也就是说，如果LEO=10，那么表示该副本保存了10条消息，位移值范围是[0, 9]。另外，leader LEO和follower LEO的更新是有区别的。我们后面会详细说
- HW：即上面提到的水位值。对于同一个副本对象而言，其HW值不会大于LEO值。小于等于HW值的所有消息都被认为是“已备份”的（replicated）。同理，leader副本和follower副本的HW更新是有区别的，我们后面详谈。

我们使用下图来形象化地说明两者的关系：



上图中，HW值是7，表示位移是0~7的所有消息都已经处于“已备份状态”（committed），而LEO值是15，那么8~14的消息就是尚未完全备份（fully replicated）——为什么没有15？因为刚才说过了，LEO指向的是下一条消息到来时的位移，故上图使用虚线框表示。我们总说consumer无法消费未提交消息。这句话如果用以上名词来解读的话，应该表述为：consumer无法消费分区下leader副本中位移值大于**分区HW的任何消息。这里需要特别注意分区HW就是leader副本的HW值**。

既然副本分为leader副本和follower副本，而每个副本又都有HW和LEO，那么它们是怎么被更

新的呢？它们更新的机制又有什么区别呢？我们一一来分析下：

一、follower副本何时更新LEO?

如前所述，follower副本只是被动地向leader副本请求数据，具体表现为follower副本不停地向leader副本所在的broker发送FETCH请求，一旦获取消息后写入自己的日志中进行备份。那么follower副本的LEO是何时更新的呢？首先我必须言明，Kafka有两套follower副本LEO(**明白这个是搞懂后面内容的关键，因此请多花一点时间来思考**)：1. 一套LEO保存在follower副本所在broker的副本管理机中；2. 另一套LEO保存在leader副本所在broker的副本管理机中——换句话说，leader副本机器上保存了所有的follower副本的LEO。

为什么要保存两套？这是因为Kafka使用前者帮助follower副本更新其HW值；而利用后者帮助leader副本更新其HW使用。下面我们分别看下它们被更新的时机。

1 follower副本端的follower副本LEO何时更新? （原谅我有点拗口~~~~~）

follower副本端的LEO值就是其底层日志的LEO值，也就是说每当新写入一条消息，其LEO值就会被更新(类似于LEO += 1)。当follower发送FETCH请求后，leader将数据返回给follower，此时follower开始向底层log写数据，从而自动地更新LEO值

2 leader副本端的follower副本LEO何时更新?

leader副本端的follower副本LEO的更新发生在leader在处理follower FETCH请求时。一旦leader接收到follower发送的FETCH请求，它首先会从自己的log中读取相应的数据，但是在给follower返回数据之前它先去更新follower的LEO(即上面所说的第二套LEO)

二、follower副本何时更新HW?

follower更新HW发生在其更新LEO之后，一旦follower向log写完数据，它会尝试更新它自己的HW值。具体算法就是比较当前LEO值与FETCH响应中leader的HW值，取两者的小者作为新的HW值。这告诉我们一个事实：如果follower的LEO值超过了leader的HW值，那么follower HW值是不会越过leader HW值的。

三、leader副本何时更新LEO?

和follower更新LEO道理相同，leader写log时就会自动地更新它自己的LEO值。

四、leader副本何时更新HW值?

前面说过了，leader的HW值就是分区HW值，因此何时更新这个值是我们最关心的，因为它直接影响了分区数据对于consumer的可见性 。以下4种情况下leader会尝试去更新分区HW——切记是尝试，有可能因为不满足条件而不做任何更新：

- 副本成为leader副本时：当某个副本成为了分区的leader副本，Kafka会尝试去更新分区HW。这是显而易见的道理，毕竟分区leader发生了变更，这个副本的状态是一定要检查的！不过，本文讨论的是当系统稳定后且正常工作时备份机制可能出现的问题，故这个条件不在我们的讨论之列。
- broker出现崩溃导致副本被踢出ISR时：若有broker崩溃则必须查看下是否会波及此分区，因此检查下分区HW值是否需要更新是有必要的。本文不对这种情况做深入讨论
- producer向leader副本写入消息时：因为写入消息会更新leader的LEO，故有必要再查看下HW值是否也需要修改
- leader处理follower FETCH请求时：当leader处理follower的FETCH请求时首先会从底层的log读取数据，之后会尝试更新分区HW值

特别注意上面4个条件中的最后两个。它揭示了一个事实——当Kafka broker都正常工作时，分区HW值的更新时机有两个：leader处理PRODUCE请求时和leader处理FETCH请求时。另外，leader是如何更新它的HW值的呢？前面说过了，leader broker上保存了一套follower副本的LEO以及它自己的LEO。当尝试确定分区HW时，它会选出所有满足条件的副本，比较它们的LEO(当然也包括leader自己的LEO)，并选择最小的LEO值作为HW值。这里的满足条件主要是指副本要满足以下两个条件之一：

- 处于ISR中
- 副本LEO落后于leader LEO的时长不大于replica.lag.time.max.ms参数值(默认是10s)

乍看上去好像这两个条件说得是一回事，毕竟ISR的定义就是第二个条件描述的那样。但某些情况下Kafka的确可能出现副本已经“追上”了leader的进度，但却不在ISR中——比如某个从failure中恢复的副本。如果Kafka只判断第一个条件的话，确定分区HW值时就不会考虑这些未在ISR中的副本，但这些副本已经具备了“立刻进入ISR”的资格，因此就可能出现分区HW值越过ISR中副本LEO的情况——这肯定是不允许的，因为分区HW实际上就是ISR中所有副本LEO的最小值。

好了，理论部分我觉得说的差不多了，下面举个实际的例子。我们假设有一个topic，单分区，副本因子是2，即一个leader副本和一个follower副本。我们看下当producer发送一条消息时，broker端的副本到底会发生什么事情以及分区HW是如何被更新的。

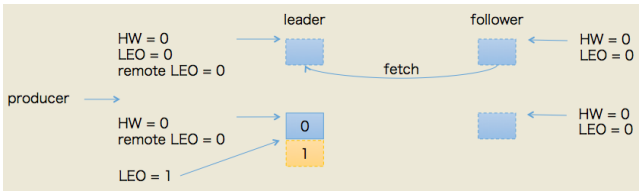
下图是初始状态，我们稍微解释一下：初始时leader和follower的HW和LEO都是0(严格来说源代码会初始化LEO为-1，不过这不影响之后的讨论)。leader中的remote LEO指的就是leader端保存的follower LEO，也被初始化成0。此时，producer没有发送任何消息给leader，而

follower已经开始不断地给leader发送FETCH请求了，但因为没有数据因此什么都不会发生。值得一提的是，follower发送过来的FETCH请求因为无数据而暂时会被寄存到leader端的purgatory中，待500ms(replica.fetch.wait.max.ms参数)超时后会强制完成。倘若在寄存期间producer端发送过来数据，那么会Kafka会自动唤醒该FETCH请求，让leader继续处理之。

虽然purgatory不是本文的重点，但FETCH请求发送和PRODUCE请求处理的时机会影响我们的讨论。因此后续我们也将分两种情况来讨论分区HW的更新。

第一种情况：follower发送FETCH请求在leader处理完PRODUCE请求之后

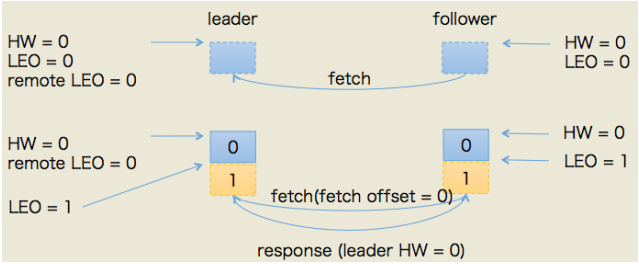
producer给该topic分区发送了一条消息。此时的状态如下图所示：



如图所示，leader接收到PRODUCE请求主要做两件事情：

1. 把消息写入写底层log（同时也就自动地更新了leader的LEO）
2. 尝试更新leader HW值（前面**leader副本何时更新HW值**一节中的第三个条件触发）。我们已经假设此时follower尚未发送FETCH请求，那么leader端保存的remote LEO依然是0，因此leader会比较它自己的LEO值和remote LEO值，发现最小值是0，与当前HW值相同，故不会更新分区HW值

所以，PRODUCE请求处理完成后leader端的HW值依然是0，而LEO是1，remote LEO是1。假设此时follower发送了FETCH请求(或者说follower早已发送了FETCH请求，只不过在broker的请求队列中排队)，那么状态变更如下图所示：



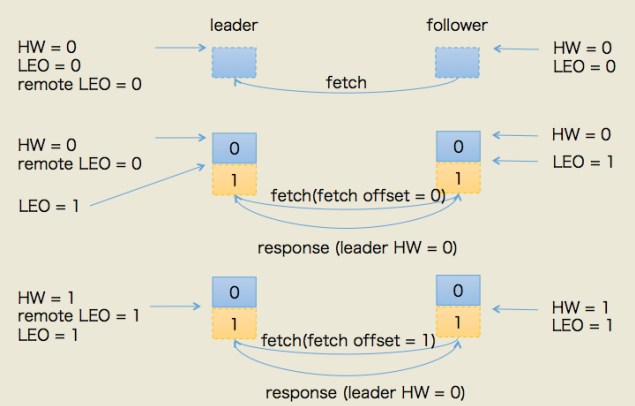
本例中当follower发送FETCH请求时，leader端的处理依次是：

1. 读取底层log数据
2. 更新remote LEO = 0（为什么是0？ 因为此时follower还没有写入这条消息。leader如何确认follower还未写入呢？这是通过follower发来的FETCH请求中的fetch offset来确定的）
3. 尝试更新分区HW——此时leader LEO = 1，remote LEO = 0，故分区HW值= min(leader LEO, follower remote LEO) = 0
4. 把数据和当前分区HW值（依然是0）发送给follower副本

而follower副本接收到FETCH response后依次执行下列操作：

1. 写入本地log（同时更新follower LEO）
2. 更新follower HW——比较本地LEO和当前leader HW取小者，故follower HW = 0

此时，第一轮FETCH RPC结束，我们会发现虽然leader和follower都已经在log中保存了这条消息，但分区HW值尚未被更新。实际上，它是在第二轮FETCH RPC中被更新的，如下图所示：



上图中，follower发来了第二轮FETCH请求，leader端接收到后仍然会依次执行下列操作：

1. 读取底层log数据
2. 更新remote LEO = 1（这次为什么是1了？ 因为这轮FETCH RPC携带的fetch offset是1，那么为什么这轮携带的就是1了呢，因为上一轮结束后follower LEO被更新为1了）
3. 尝试更新分区HW——此时leader LEO = 1，remote LEO = 1，故分区HW值= min(leader LEO, follower remote LEO) = 1。**注意分区HW值此时被更新了！！！**
4. 把数据（实际上没有数据）和当前分区HW值（已更新为1）发送给follower副本

同样地，follower副本接收到FETCH response后依次执行下列操作：

1. **写入本地log，当然没东西可写，故follower LEO也不会变化，依然是1**
2. **更新follower HW——比较本地LEO和当前leader LEO取小者。由于此时两者都是1，故更新follower HW = 1 （*注意：我特意用了两种颜色来描述这两步，后续会谈到原因！*）**

Okay，producer端发送消息后broker端完整的处理流程就讲完了。此时消息已经成功地被复制到leader和follower的log中且分区HW是1，表明consumer能够消费offset = 0的这条消息。下面

我们来分析下PRODUCE和FETCH请求交互的第二种情况。

第二种情况：FETCH请求保存在purgatory中PRODUCE请求到来

这种情况实际上和第一种情况差不多。前面说过了，当leader无法立即满足FECTH返回要求的时候(比如没有数据)，那么该FETCH请求会被暂存到leader端的purgatory中，待时机成熟时会尝试再次处理它。不过Kafka不会无限期地将其缓存着，默认有个超时时间（500ms），一旦超时时间已过，则这个请求会被强制完成。不过我们要讨论的场景是在寄存期间，producer发送PRODUCE请求从而使之满足了条件从而被唤醒。此时，leader端处理流程如下：

1. leader写入本地log（同时自动更新leader LEO）
2. 尝试唤醒在purgatory中寄存的FETCH请求
3. 尝试更新分区HW

至于唤醒后的FETCH请求的处理与第一种情况完全一致，故这里不做详细展开了。

以上所有的东西其实就想说明一件事情：**Kafka使用HW值来决定副本备份的进度，而HW值的更新通常需要额外一轮FETCH RPC才能完成，故而这种设计是有问题的。**它们可能引起的问题包括：

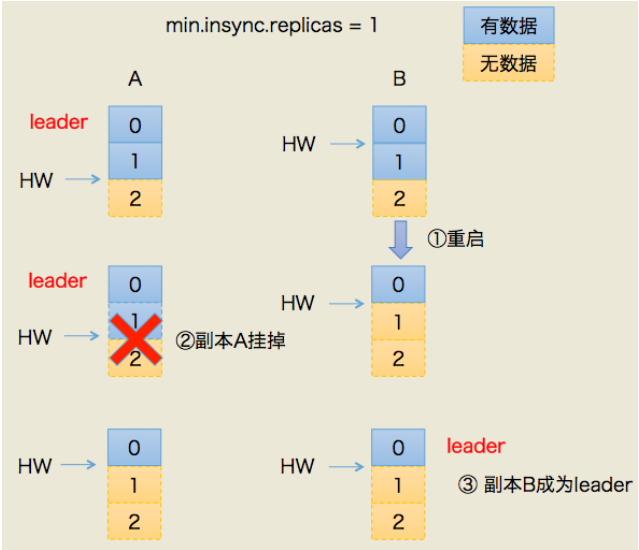
- 备份数据丢失
- 备份数据不一致

我们一一分析下：

一、数据丢失

如前所述，使用HW值来确定备份进度时其值的更新是在下一轮RPC中完成的。现在翻到上面使用两种不同颜色标记的步骤处思考下， 如果follower副本在蓝色标记的第一步与紫色标记的

第二步之间发生崩溃，那么就有可能造成数据的丢失。我们举个例子来看下。



上图中有两个副本：A和B。开始状态是A是leader。我们假设producer端min.insync.replicas设置为1，那么当producer发送两条消息给A后，A写入到底层log，此时Kafka会通知producer说这两条消息写入成功。

但是在broker端，leader和follower底层的log虽都写入了2条消息且分区HW已经被更新到2，但follower HW尚未被更新（也就是上面紫色颜色标记的第二步尚未执行）。倘若此时副本B所在的broker宕机，那么重启回来后B会自动把LEO调整到之前的HW值，故副本B会做日志截断(log truncation)，将offset = 1的那条消息从log中删除，并调整LEO = 1，此时follower副本底层log中就只有一条消息，即offset = 0的消息。

B重启之后需要给A发FETCH请求，但若A所在broker机器在此时宕机，那么Kafka会令B成为新的leader，而当A重启回来后也会执行日志截断，将HW调整回1。这样，位移=1的消息就从两个副本的log中被删除，即永远地丢失了。

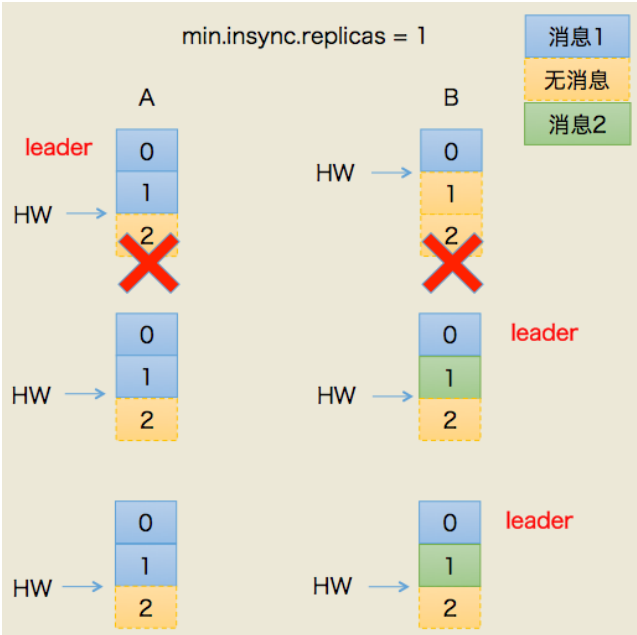
这个场景丢失数据的前提是在min.insync.replicas=1时，一旦消息被写入leader端log即被认为是“已提交”，而延迟一轮FETCH RPC更新HW值的设计使得follower HW值是异步延迟更新的，倘若在这个过程中leader

发生变更，那么成为新leader的follower的HW值就有可能是过期的，使得clients端认为是成功提交的消息被删除。

二、leader/follower数据离散

除了可能造成的数据丢失以外，这种设计还有一个潜在的问题，即造成leader端log和follower端log的数据不一致。比如leader端保存的记录序列是r1,r2,r3,r4,r5,....；而follower端保存的序列可能是r1,r3,r4,r5,r6...。这也是非法的场景，因为顾名思义，follower必须追随leader，完整地备份leader端的数据。

我们依然使用一张图来说明这种场景是如何发生的：



这种情况的初始状态与情况1有一些不同的：A依然是leader，A的log写入了2条消息，但B的log只写入了1条消息。分区HW更新到2，但B的HW还是1，同时producer端的min.insync.replicas = 1。

这次我们让A和B所在机器同时挂掉，然后假设B先重启回来，因此成为leader，分区HW = 1。假设此时producer发送了第3条消息(绿色框表示)给B，于是B的log中offset = 1的消息变成了绿色框表示的消息，同时分区HW更新到2（A还

没有回来，就B一个副本，故可以直接更新HW而不用理会A）之后A重启回来，需要执行日志截断，但发现此时分区HW=2而A之前的HW值也是2，故不做任何调整。此后A和B将以这种状态继续正常工作。

显然，这种场景下，A和B底层log中保存在offset = 1的消息是不同的记录，从而引发不一致的情形出现。

Kafka 0.11.0.0.版本解决方案

造成上述两个问题的根本原因在于HW值被用于衡量副本备份的成功与否以及在出现failture时作为日志截断的依据，但HW值的更新是异步延迟的，特别是需要额外的FETCH请求处理流程才能更新，故这中间发生的任何崩溃都可能导致HW值的过期。鉴于这些原因，Kafka 0.11引入了leader epoch来取代HW值。Leader端多开辟一段内存区域专门保存leader的epoch信息，这样即使出现上面的两个场景也能很好地规避这些问题。

所谓leader epoch实际上是一对值：（epoch，offset）。epoch表示leader的版本号，从0开始，当leader变更过1次时epoch就会+1，而offset则对应于该epoch版本的leader写入第一条消息的位移。因此假设有两对值：

(0, 0)

(1, 120)

则表示第一个leader从位移0开始写入消息；共写了120条[0, 119]；而第二个leader版本号是1，从位移120处开始写入消息。

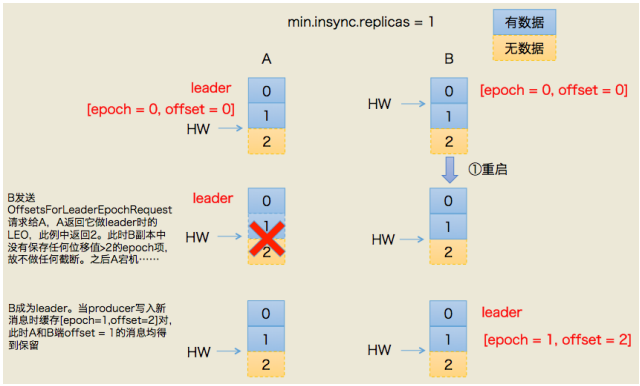
leader broker中会保存这样的一个缓存，并定期地写入到一个checkpoint文件中。

当leader写底层log时它会尝试更新整个缓存——如果这个leader首次写消息，则会在缓存中增加一个条目；否则就不做更新。而每次副本重

新成为leader时会查询这部分缓存，获取出对应leader版本的位移，这就不会发生数据不一致和丢失的情况。

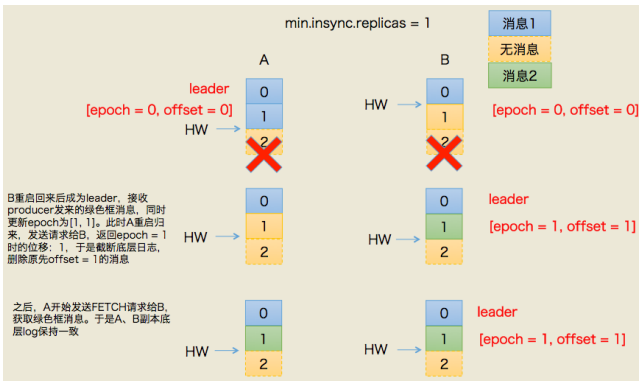
下面我们依然使用图的方式来说明下利用leader epoch如何规避上述两种情况

一、规避数据丢失



上图左半边已经给出了简要的流程描述，这里不详细展开具体的leader epoch实现细节（比如OffsetsForLeaderEpochRequest的实现），我们只需要知道每个副本都引入了新的状态来保存自己当leader时开始写入的第一条消息的offset以及leader版本。这样在恢复的时候完全使用这些信息而非水位来判断是否需要截断日志。

二、规避数据不一致



同样的道理，依靠leader epoch的信息可以有效地规避数据不一致的问题。

总结

0.11.0.0版本的Kafka通过引入leader epoch解决了原先依赖水位表示副本进度可能造成的数据丢

失/数据不一致问题。有兴趣的读者可以阅读源代码进一步地了解其中的工作原理。

源代码位置：kafka.server.epoch.LeaderEpochCache.scala（leader epoch数据结构）、kafka.server.checkpoints.LeaderEpochCheckpointFile（checkpoint检查点文件操作类）还有分布在Log中的CRUD操作。

标签：Kafka

huxihx
粉丝 - 563 关注 - 0
+加关注

« 上一篇： 【译】Apache Flink Kafka consumer
» 下一篇： 关于Kafka幂等producer的讨论

posted @ 2017-09-21 15:40 huxihx 阅读(34174) 评论(44) 编辑 收藏 举报

编辑推荐：
· .NET IoT 入门指南：基于 GPS 的 NTP 时间同步服务器
· ASP.NET Core 6框架揭秘实例演示[31]：路由高阶用法
· 使用前端技术实现静态图片局部流动效果
· 前端构建效率优化之路
· .NET性能优化-使用SourceGenerator-Logger记录日志

最新新闻：
· 优酷Q1日均付费用户同比增长15%，来自内容和88VIP的推动 | 看财报
· 当不小心磕到“发霉”瓜子……
· 下一代数据中心安全：Web3之基础架构
· 星巴克将在下个月宣布基于 Web3 的积分奖励计划
· 独家：快手组织架构再调整，原商业化负责人马宏彬转任国际化负责人
» 更多新闻...