咕泡学院 VIP 课 :分布式消息通信 kafka 原理分析

课程目标

1. 消息的存储原理
2. Partition 的副本机制原理
3. 副本数据的同步原理

消息的文件存储机制

前面我们知道了一个 topic 的多个 partition 在物理磁盘上的保存路径，那么我们再来分析日志的存储方式。通过如下命令找到对应 partition 下的日志内容

```
[root@localhost ~]# ls /tmp/kafka-logs/firstTopic-1/
00000000000000000000.index
```

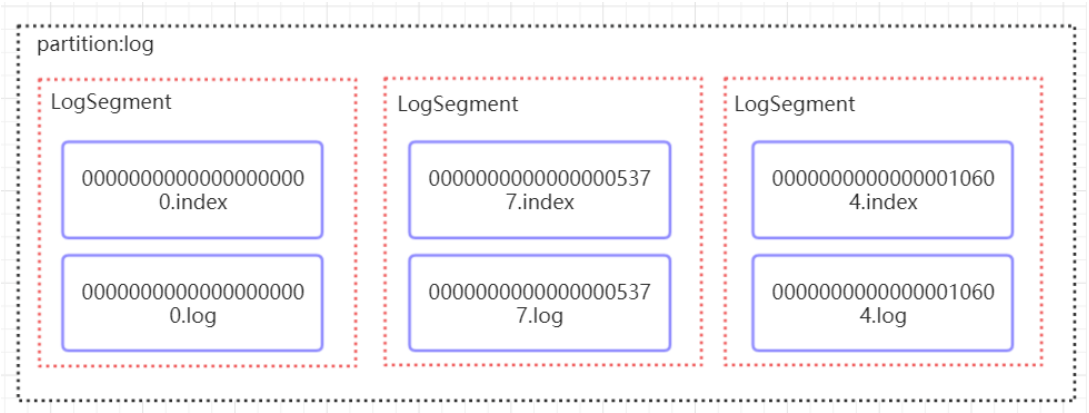
00000000000000000000.log 00000000000000000000.timeindex leader-epoch-checkpoint

kafka 是通过分段的方式将 Log 分为多个 LogSegment，LogSegment 是一个逻辑上的概念，一个 LogSegment 对应磁盘上的一个日志文件和一个索引文件，其中日志文件是用来记录消息的。索引文件是用来保存消息的索引。那么这个 LogSegment 是什么呢?

LogSegment

假设 kafka 以 partition 为最小存储单位，那么我们可以想象当 kafka producer 不断发送消息，必然会引起 partition 文件的无线扩张，这样对于消息文件的维护以及被消费的消息的清理带来非常大的挑战，所以 kafka 以 segment 为单位又把 partition 进行细分。每个 partition 相当于一个巨型文件被平均分配到多个大小相等的 segment 数据文件中（每个 segment 文件中的消息不一定相等），这种特性方便已经被消费的消息的清理，提高磁盘的利用率。

- log.segment.bytes=107370 （设置分段大小),默认是 1gb，我们把这个值调小以后，可以看到日志分段的效果
- 抽取其中 3 个分段来进行分析



segment file 由 2 大部分组成，分别为 index file 和 data file，此 2 个文件一一对应，成对出现，后缀".index"和".log"分别表示为 segment 索引文件、数据文件.

segment 文件命名规则：partion 全局的第一个 segment 从 0 开始，后续每个 segment 文件名为上一个 segment 文件最后一条消息的 offset 值进行递增。数值最大为 64 位 long 大小，20 位数字字符长度，没有数字用 0 填充

查看 segment 文件命名规则

➢ 通过下面这条命令可以看到 kafka 消息日志的内容

sh kafka-run-class.sh kafka.tools.DumpLogSegments --files /tmp/kafka-logs/test-0/00000000000000000000.log --print-data-log

输出结果为:

```
offset: 5376 position: 102124 CreateTime: 1531477349287 isvalid: true keysize: -1 valuesize: 12 magic: 2 compresscodec: NONE producerId: -1 producerEpoch: -
```

1 sequence: -1 isTransactional: false headerKeys: [] payload: message_5376

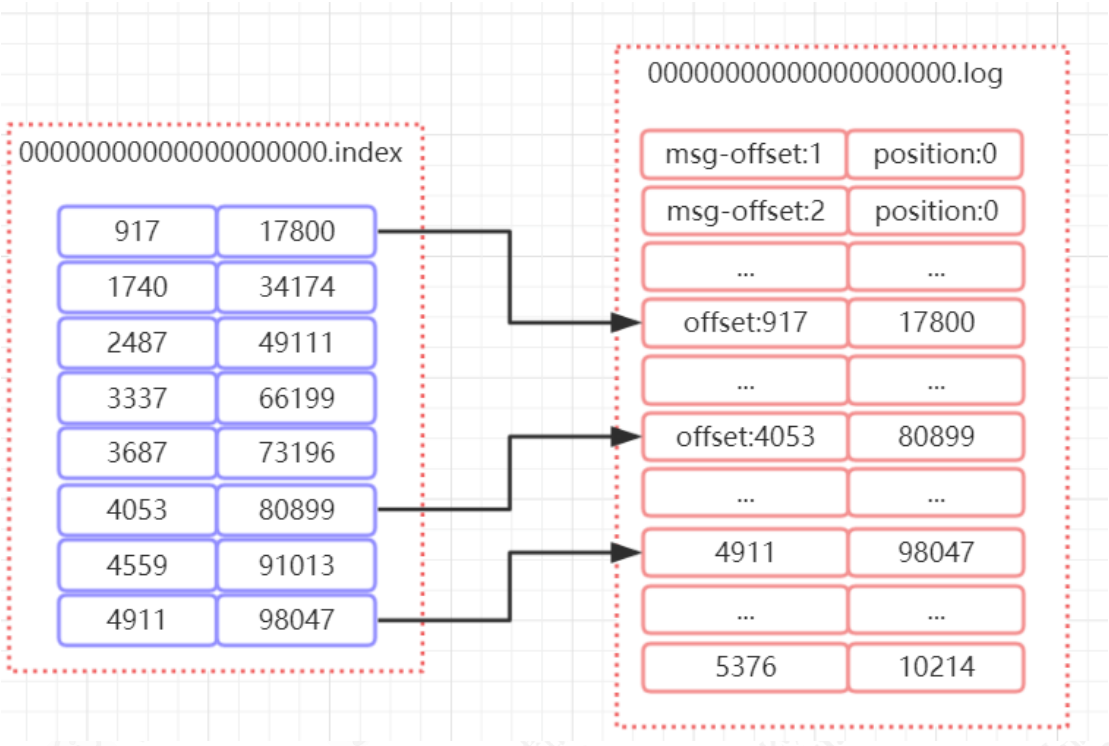
第一个 log 文件的最后一个 offset 为:5376,所以下一个 segment 的文件命名为: 00000000000000005376.log。对应的 index 为 00000000000000005376.index

segment 中 index 和 log 的对应关系

从所有分段中，找一个分段进行分析

为了提高查找消息的性能，为每一个日志文件添加 2 个索引索引文件：OffsetIndex 和 TimeIndex，分别对应*.index 以及*.timeindex, TimeIndex 索引文件格式：它是映射时间戳和相对 offset

查看索引内容：sh kafka-run-class.sh kafka.tools.DumpLogSegments --files /tmp/kafka-logs/test-0/00000000000000000000.index --print-data-log



如图所示，index 中存储了索引以及物理偏移量。 log 存储了消息的内容。索引文件的元数据执行对应数据文件中 message 的物理偏移地址。举个简单的案例来说，以[4053,80899]为例，在 log 文件中，对应的是第 4053 条记录，物理偏移量（position）为 80899. position 是 ByteBuffer 的指针位置

在 partition 中如何通过 offset 查找 message

查找的算法是

1. 根据 offset 的值，查找 segment 段中的 index 索引文件。由于索引文件命名是以上一个文件的最后一个 offset 进行命名的，所以，使用二分查找算法能够根据 offset 快速定位到指定的索引文件。

2. 找到索引文件后，根据 offset 进行定位，找到索引文件中的符合范围的索引。（kafka 采用稀疏索引的方式来提高查找性能）
3. 得到 position 以后，再到对应的 log 文件中，从 position 出开始查找 offset 对应的消息，将每条消息的 offset 与目标 offset 进行比较，直到找到消息

比如说，我们要查找 offset=2490 这条消息，那么先找到 00000000000000000000.index，然后找到[2487,49111]这个索引，再到 log 文件中，根据 49111 这个 position 开始查找，比较每条消息的 offset 是否大于等于 2490。最后查找到对应的消息以后返回

Log 文件的消息内容分析

前面我们通过 kafka 提供的命令，可以查看二进制的日志文件信息，一条消息，会包含很多的字段。

offset: 5371 position: 102124 CreateTime: 1531477349286 isvalid: true keysize: -1 valuesize: 12 magic: 2 compresscodec: NONE producerId: -1 producerEpoch: -1 sequence: -1 isTransactional: false headerKeys: [] payload: message_5371

offset 和 position 这两个前面已经讲过了、 createTime 表示创建时间、keysize 和 valuesize 表示 key 和 value 的大小、 compresscodec 表示压缩编码、payload:表示消息的具体内容

日志的清除策略以及压缩策略

日志清除策略

前面提到过，日志的分段存储，一方面能够减少单个文件内容的大小，另一方面，方便 kafka 进行日志清理。日志的清理策略有两个

1. 根据消息的保留时间，当消息在 kafka 中保存的时间超过了指定的时间，就会触发清理过程
2. 根据 topic 存储的数据大小，当 topic 所占的日志文件大小大于一定的阀值，则可以开始删除最旧的消息。kafka 会启动一个后台线程，定期检查是否存在可以删除的消息

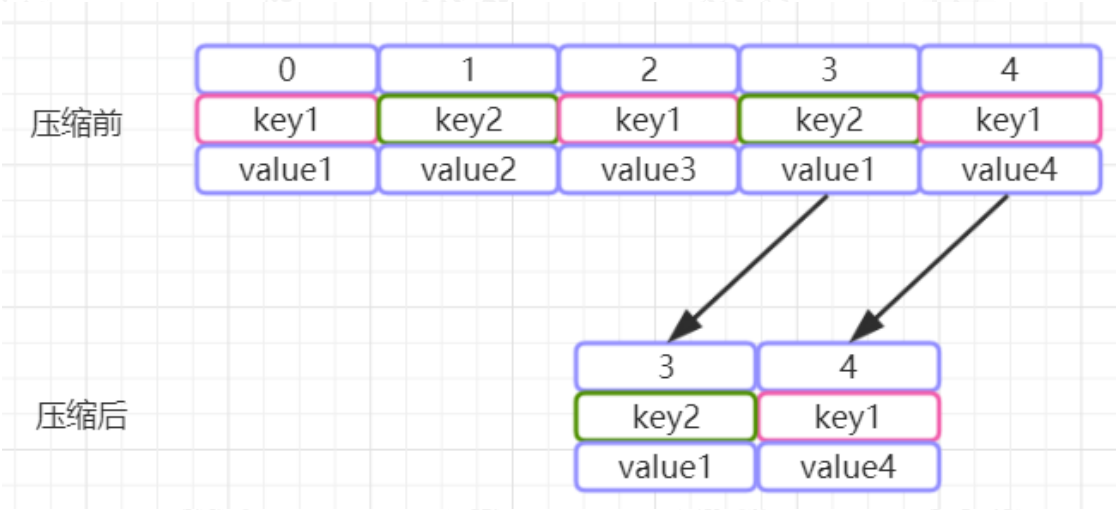
通过 log.retention.bytes 和 log.retention.hours 这两个参数来设置，当其中任意一个达到要求，都会执行删除。

默认的保留时间是：7 天

日志压缩策略

Kafka 还提供了“日志压缩（Log Compaction）”功能，通

过这个功能可以有效的减少日志文件的大小，缓解磁盘紧张的情况，在很多实际场景中，消息的 key 和 value 的值之间的对应关系是不断变化的，就像数据库中的数据会不断被修改一样，消费者只关心 key 对应的最新的 value。因此，我们可以开启 kafka 的日志压缩功能，服务端会在后台启动启动 Cleaner 线程池，定期将相同的 key 进行合并，只保留最新的 value 值。日志的压缩原理是



partition 的高可用副本机制

我们已经知道 Kafka 的每个 topic 都可以分为多个 Partition，并且多个 partition 会均匀分布在集群的各个节点下。虽然这种方式能够有效的对数据进行分片，但是对于每个 partition 来说，都是单点的，当其中一个 partition 不可用的时候，那么这部分消息就没办法消费。所以 kafka 为了

提高 partition 的可靠性而提供了副本的概念（Replica），通过副本机制来实现冗余备份。

每个分区可以有多个副本，并且在副本集合中会存在一个 leader 的副本，所有的读写请求都是由 leader 副本来进行处理。剩余的其他副本都做为 follower 副本，follower 副本会从 leader 副本同步消息日志。这个有点类似 zookeeper 中 leader 和 follower 的概念，但是具体的时间方式还是有比较大的差异。所以我们可以认为，副本集会存在一主多从的关系。

一般情况下，同一个分区的多个副本会被均匀分配到集群中的不同 broker 上，当 leader 副本所在的 broker 出现故障后，可以重新选举新的 leader 副本继续对外提供服务。通过这样的副本机制来提高 kafka 集群的可用性。

副本分配算法

将所有 N Broker 和待分配的 i 个 Partition 排序.

将第 i 个 Partition 分配到第(i mod n)个 Broker 上.

将第 i 个 Partition 的第 j 个副本分配到第((i + j) mod n)个 Broker 上.

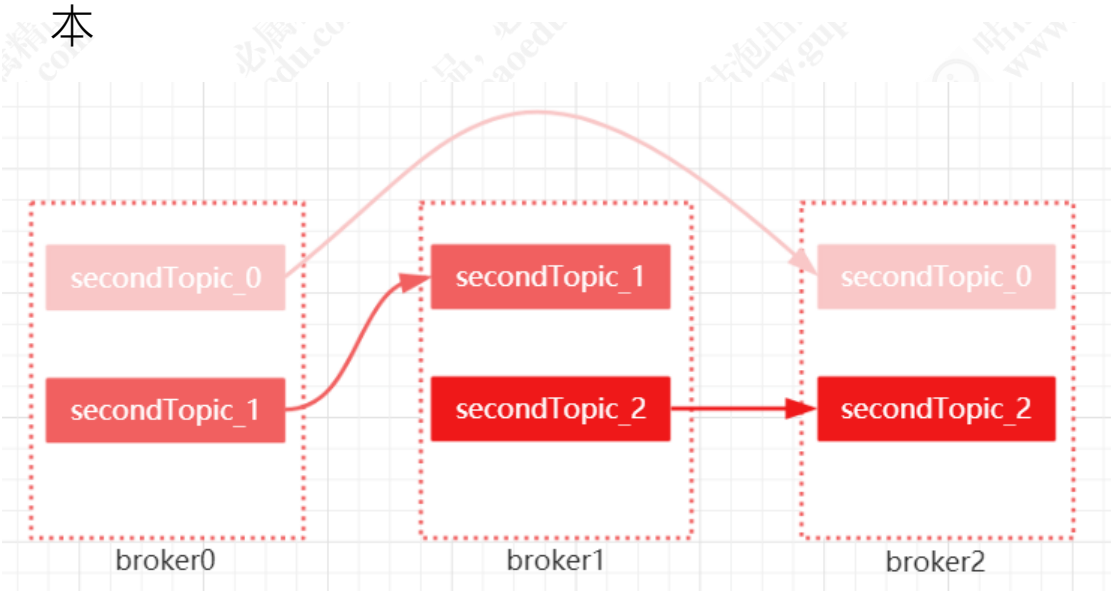
创建一个带副本机制的 topic

通过下面的命令去创建带 2 个副本的 topic

./kafka-topics.sh --create --zookeeper 192.168.11.156:2181 --replication-factor 2 --partitions 3 --topic secondTopic

然后我们可以在/tmp/kafka-log 路径下看到对应 topic 的副本信息了。我们通过一个图形的方式来表达。

- 针对 secondTopic 这个 topic 的 3 个分区对应的 3 个副本



如何知道那个各个分区中对应的 leader 是谁呢?

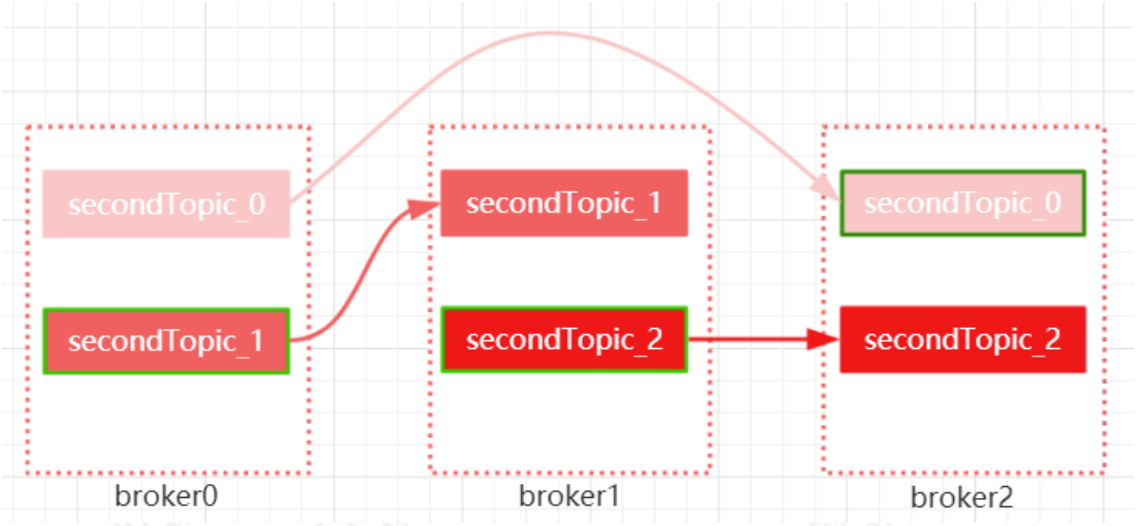
在 zookeeper 服务器上，通过如下命令去获取对应分区的信息，比如下面这个是获取 secondTopic 第 1 个分区的状态信息。

get /brokers/topics/secondTopic/partitions/1/state

- {"controller_epoch":12,"leader":0,"version":1,"leader_epoch":0,"isr":[0,1]}

leader 表示当前分区的 leader 是那个 broker-id。下图中。

绿色线条的表示该分区中的 leader 节点。其他节点就为 follower



Kafka 提供了数据复制算法保证，如果 leader 发生故障或挂掉，一个新 leader 被选举并被接受客户端的消息成功写入。Kafka 确保从同步副本列表中选举一个副本为 leader；leader 负责维护和跟踪 ISR(in-Sync replicas ， 副本同步队列)中所有 follower 滞后的状态。当 producer 发送一条消息到 broker 后，leader 写入消息并复制到所有 follower。消息提交之后才被成功复制到所有的同步副本。

- 既然有副本机制，就一定涉及到数据同步的概念，那接下来分析下数据是如何同步的?

需要注意的是，大家不要把 zookeeper 的 leader 和 follower 的同步机制和 kafka 副本的同步机制搞混了。虽然从思想层面来说是一样的，但是原理层面的实现是完全不同的。

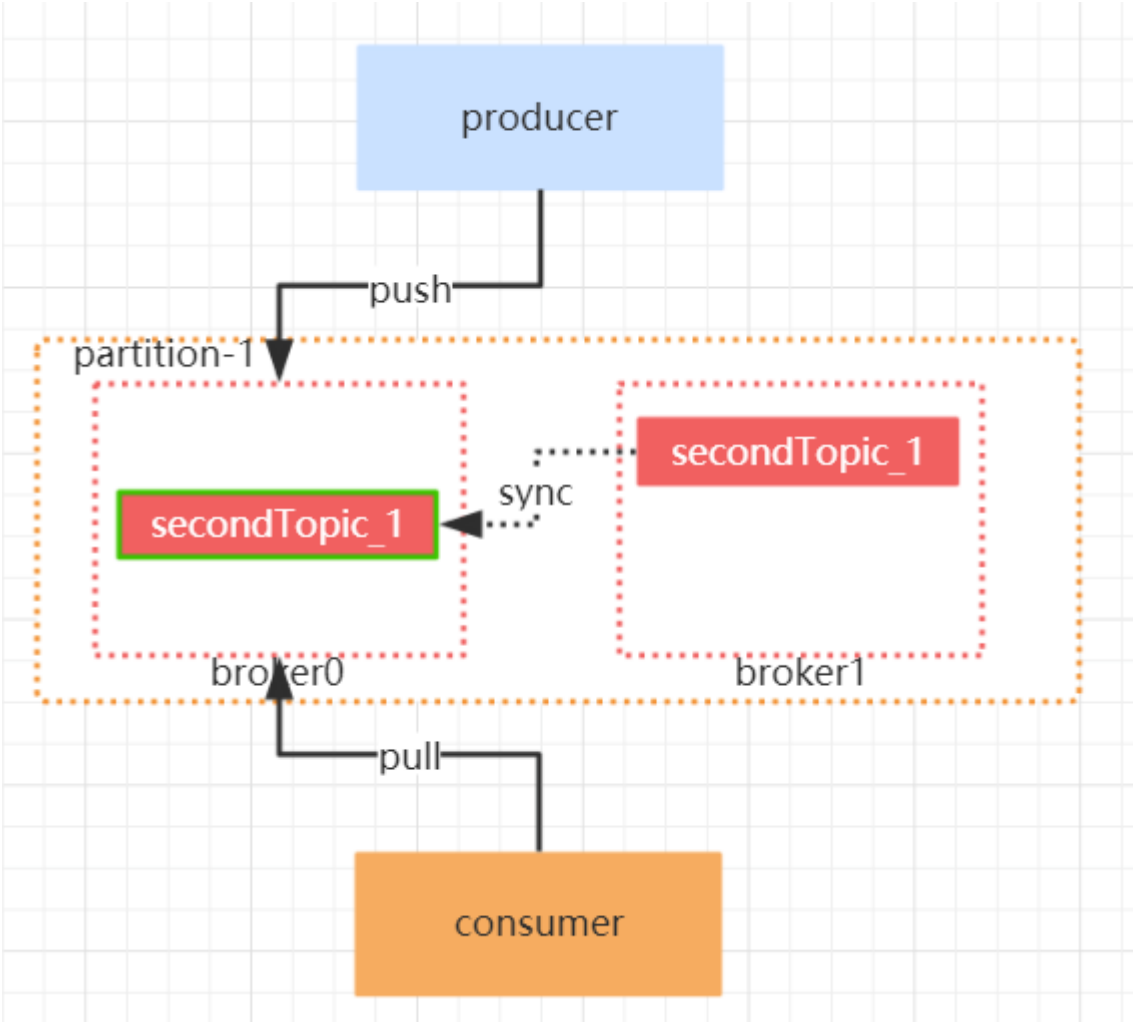
kafka 副本机制中的几个概念

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。
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

副本协同机制

刚刚提到了，消息的读写操作都只会由 leader 节点来接收和处理。follower 副本只负责同步数据以及当 leader 副本所在的 broker 挂了以后，会从 follower 副本中选取新的 leader。



写请求首先由 Leader 副本处理，之后 follower 副本会从 leader 上拉取写入的消息，这个过程会有一定的延迟，导致 follower 副本中保存的消息略少于 leader 副本，但是只要没有超出阈值都可以容忍。但是如果一个 follower 副本

出现异常，比如宕机、网络断开等原因长时间没有同步到消息，那这个时候，leader 就会把它踢出去。kafka 通过 ISR 集合来维护一个分区副本信息

ISR

ISR 表示目前"可用且消息量与 leader 相差不多的副本集合，这是整个副本集合的一个子集"。怎么去理解可用和相差不多这两个词呢？具体来说，ISR 集合中的副本必须满足两个条件

1. 副本所在节点必须维持着与 zookeeper 的连接
2. 副本最后一条消息的 offset 与 leader 副本的最后一条消息的 offset 之间的差值不能超过指定的阈值(replica.lag.time.max.ms)

   replica.lag.time.max.ms：如果该 follower 在此时间间隔内一直没有追上过 leader 的所有消息，则该 follower 就会被剔除 isr 列表

- ISR 数据保存在 Zookeeper 的/brokers/topics/<topic>/partitions/<partitionId>/state 节点中

HW&LEO

关于 follower 副本同步的过程中，还有两个关键的概念，HW(HighWatermark)和 LEO(Log End Offset). 这两个参

数跟 ISR 集合紧密关联。HW 标记了一个特殊的 offset，当消费者处理消息的时候，只能拉去到 HW 之前的消息，HW 之后的消息对消费者来说是不可见的。也就是说，取 partition 对应 ISR 中最小的 LEO 作为 HW，consumer 最多只能消费到 HW 所在的位置。每个 replica 都有 HW，leader 和 follower 各自维护更新自己的 HW 的状态。一条消息只有被 ISR 里的所有 Follower 都从 Leader 复制过去才会被认为已提交。这样就避免了部分数据被写进了 Leader，还没来得及被任何 Follower 复制就宕机了，而造成数据丢失（Consumer 无法消费这些数据）。而对于 Producer 而言，它可以选择是否等待消息 commit，这可以通过 acks 来设置。这种机制确保了只要 ISR 有一个或以上的 Follower，一条被 commit 的消息就不会丢失。

数据的同步过程

了解了副本的协同过程以后，还有一个最重要的机制，就是数据的同步过程。它需要解决

1. 怎么传播消息
2. 在向消息发送端返回 ack 之前需要保证多少个 Replica 已经接收到这个消息

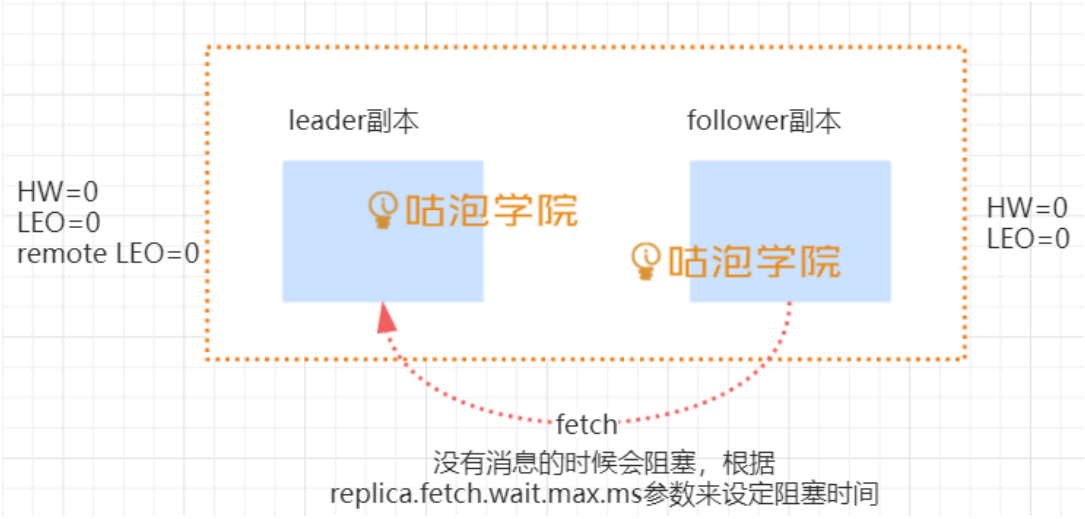
数据的处理过程是

Producer 在发布消息到某个 Partition 时，先通过

ZooKeeper 找到该 Partition 的 Leader【get /brokers/topics/<topic>/partitions/2/state】，然后无论该 Topic 的 Replication Factor 为多少（也即该 Partition 有多少个 Replica），Producer 只将该消息发送到该 Partition 的 Leader。Leader 会将该消息写入其本地 Log。每个 Follower 都从 Leader pull 数据。这种方式上，Follower 存储的数据顺序与 Leader 保持一致。Follower 在收到该消息并写入其 Log 后，向 Leader 发送 ACK。一旦 Leader 收到了 ISR 中的所有 Replica 的 ACK，该消息就被认为已经 commit 了，Leader 将增加 HW(HighWatermark)并且向 Producer 发送 ACK。

初始状态

初始状态下，leader 和 follower 的 HW 和 LEO 都是 0，leader 副本会保存 remote LEO，表示所有 follower LEO，也会被初始化为 0，这个时候，producer 没有发送消息。follower 会不断地个 leader 发送 FETCH 请求，但是因为没有数据，这个请求会被 leader 寄存，当在指定的时间之后会强制完成请求，这个时间配置是(replica.fetch.wait.max.ms)，如果在指定时间内 producer 有消息发送过来，那么 kafka 会唤醒 fetch 请求，让 leader 继续处理

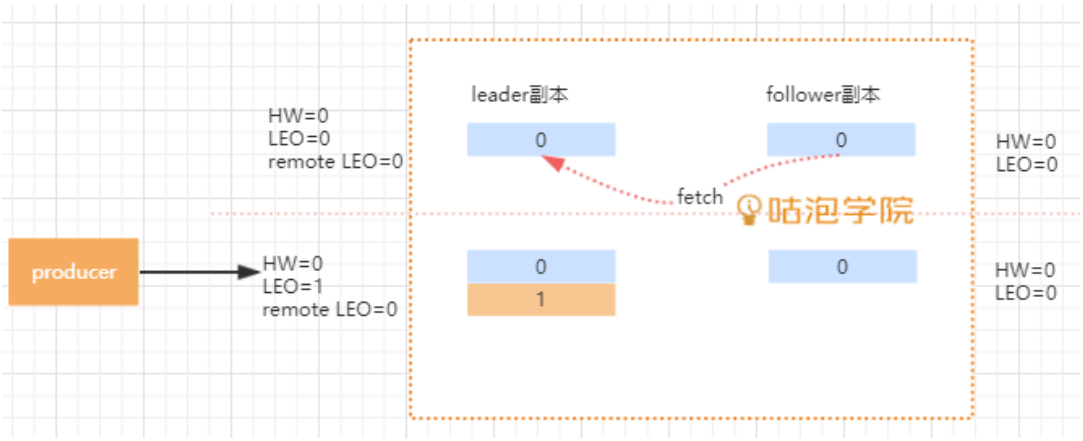


这里会分两种情况，第一种是 leader 处理完 producer 请求之后，follower 发送一个 fetch 请求过来、第二种是 follower 阻塞在 leader 指定时间之内，leader 副本收到 producer 的请求。这两种情况下处理方式是不一样的。先来看第一种情况

follower 的 fetch 请求是当 leader 处理消息以后执行的

生产者发送一条消息

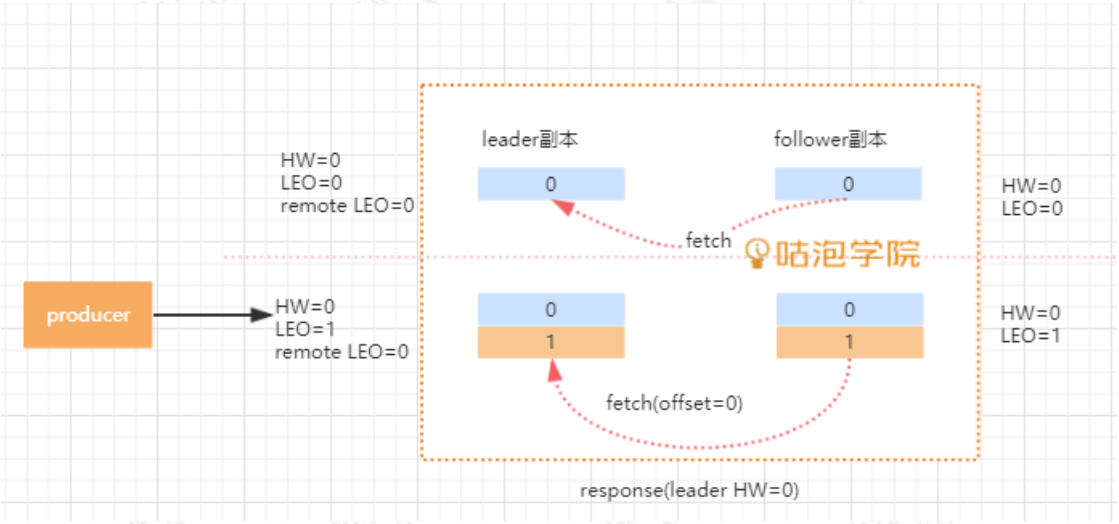
- leader 处理完 producer 请求之后，follower 发送一个 fetch 请求过来 。状态图如下



leader 副本收到请求以后，会做几件事情

1. 把消息追加到 log 文件，同时更新 leader 副本的 LEO
2. 尝试更新 leader HW 值。这个时候由于 follower 副本还没有发送 fetch 请求，那么 leader 的 remote LEO 仍然是 0。leader 会比较自己的 LEO 以及 remote LEO 的值发现最小值是 0，与 HW 的值相同，所以不会更新 HW

follower fetch 消息



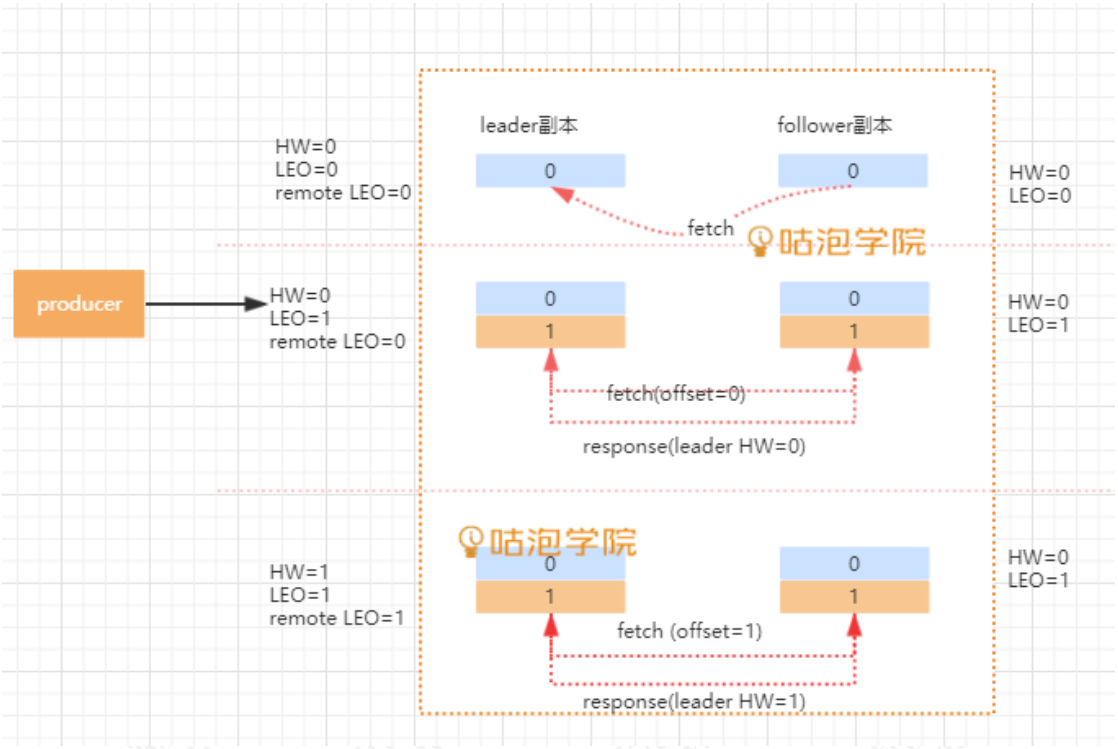
follower 发送 fetch 请求，leader 副本的处理逻辑是:

1. 读取 log 数据、更新 remote LEO=0(follower 还没有写入这条消息，这个值是根据 follower 的 fetch 请求中的 offset 来确定的)
2. 尝试更新 HW，因为这个时候 LEO 和 remoteLEO 还是不一致，所以仍然是 HW=0
3. 把消息内容和当前分区的 HW 值发送给 follower 副本

follower 副本收到 response 以后

1. 将消息写入到本地 log，同时更新 follower 的 LEO
2. 更新 follower HW，本地的 LEO 和 leader 返回的 HW 进行比较取小的值，所以仍然是 0

第一次交互结束以后，HW 仍然还是 0，这个值会在下一次 follower 发起 fetch 请求时被更新



follower 发第二次 fetch 请求，leader 收到请求以后

1. 读取 log 数据
2. 更新 remote LEO=1， 因为这次 fetch 携带的 offset 是 1.
3. 更新当前分区的 HW，这个时候 leader LEO 和 remote LEO 都是 1，所以 HW 的值也更新为 1
4. 把数据和当前分区的 HW 值返回给 follower 副本，这个时候如果没有数据，则返回为空

follower 副本收到 response 以后

1. 如果有数据则写本地日志，并且更新 LEO
2. 更新 follower 的 HW 值

到目前为止，数据的同步就完成了，意味着消费端能够消费 offset=0 这条消息。

follower 的 fetch 请求是直接从阻塞过程中触发

前面说过，由于 leader 副本暂时没有数据过来，所以 follower 的 fetch 会被阻塞，直到等待超时或者 leader 接收到新的数据。当 leader 收到请求以后会唤醒处于阻塞的 fetch 请求。处理过程基本上和前面说的一直

1. leader 将消息写入本地日志，更新 Leader 的 LEO
2. 唤醒 follower 的 fetch 请求
3. 更新 HW

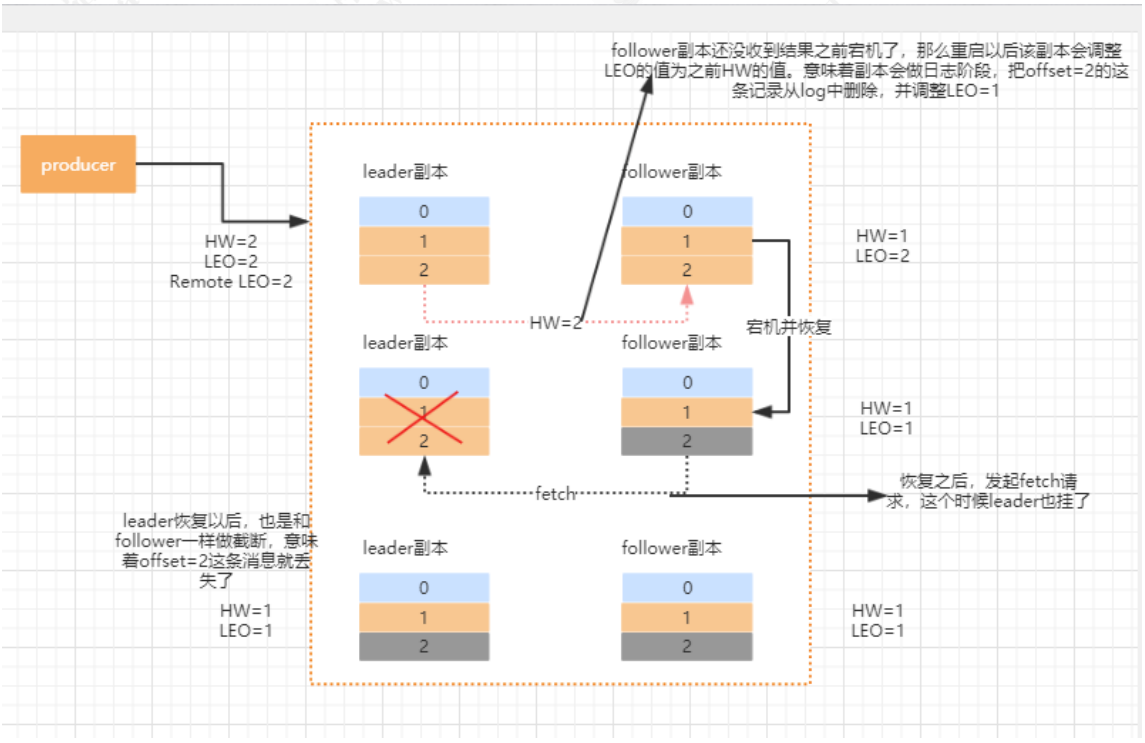
kafka 使用 HW 和 LEO 的方式来实现副本数据的同步，本身是一个好的设计，但是在这个地方会存在一个数据丢失的问题，当然这个丢失只出现在特定的背景下。我们回想一下，HW 的值是在新的一轮 FETCH 中才会被更新。我们分析下这个过程为什么会出现数据丢失

数据丢失的问题

前提：min.insync.replicas=1 的时候。->设定 ISR 中的最小副本数是多少，默认值为 1，当且仅当 acks 参数设置为-1（表示需要所有副本确认）时，此参数才生效. 表达的含义是，至少需要多少个副本同步才能表示消息是提交的

所以，当 min.insync.replicas=1 的时候

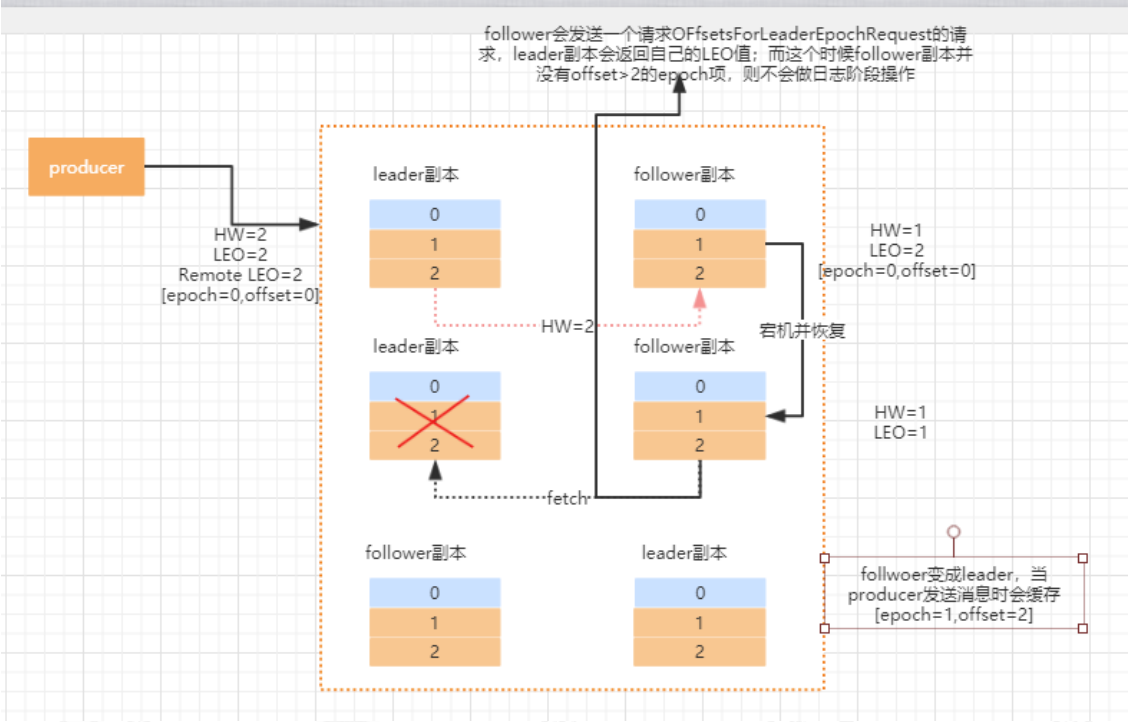
一旦消息被写入 leader 端 log 即被认为是“已提交”，而延迟一轮 FETCH RPC 更新 HW 值的设计使得 follower HW 值是异步延迟更新的，倘若在这个过程中 leader 发生变更，那么成为新 leader 的 follower 的 HW 值就有可能是过期的，使得 clients 端认为是成功提交的消息被删除。



数据丢失的解决方案

在 kafka0.11.0.0 版本以后，提供了一个新的解决方案，使用 leader epoch 来解决这个问题，leader epoch 实际上是一对之(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。
比如说
(0,0) ; (1,50); 表示第一个 leader 从 offset=0 开始写消息，一共写了 50 条，第二个 leader 版本号是 1，从 50 条处开始写消息。这个信息保存在对应分区的本地磁盘文件中，文件名为：/tml/kafka-log/topic/leader-epoch-checkpoint

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 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 offset



如何处理所有的 Replica 不工作的情况

在 ISR 中至少有一个 follower 时，Kafka 可以确保已经 commit 的数据不丢失，但如果某个 Partition 的所有 Replica 都宕机了，就无法保证数据不丢失了

1. 等待 ISR 中的任一个 Replica“活”过来，并且选它作为 Leader
2. 选择第一个“活”过来的 Replica（不一定是 ISR 中的）作为 Leader

这就需要在可用性和一致性当中作出一个简单的折衷。

如果一定要等待 ISR 中的 Replica“活”过来，那不可用的时间就可能会相对较长。而且如果 ISR 中的所有 Replica 都无法“活”过来了，或者数据都丢失了，这个 Partition 将永远

不可用。

选择第一个“活”过来的 Replica 作为 Leader，而这个 Replica 不是 ISR 中的 Replica，那即使它并不保证已经包含了所有已 commit 的消息，它也会成为 Leader 而作为 consumer 的数据源（前文有说明，所有读写都由 Leader 完成）。在我们课堂讲的版本中，使用的是第一种策略。

ISR 的设计原理

在所有的分布式存储中，冗余备份是一种常见的设计方式，而常用的模式有同步复制和异步复制，按照 kafka 这个副本模型来说

如果采用同步复制，那么需要要求所有能工作的 Follower 副本都复制完，这条消息才会被认为提交成功，一旦有一个 follower 副本出现故障，就会导致 HW 无法完成递增，消息就无法提交，消费者就获取不到消息。这种情况下，故障的 Follower 副本会拖慢整个系统的性能，设置导致系统不可用

如果采用异步复制，leader 副本收到生产者推送的消息后，就认为次消息提交成功。follower 副本则异步从 leader 副本同步。这种设计虽然避免了同步复制的问题，但是假设所有 follower 副本的同步速度都比较慢他们保存的消息量远远落后于 leader 副本。而此时 leader 副本所在的 broker 突然宕机，则会重新选举新的 leader 副本，而新的 leader 副本中没有原

来 leader 副本的消息。这就出现了消息的丢失。

kafka 权衡了同步和异步的两种策略，采用 ISR 集合，巧妙解决了两种方案的缺陷：当 follower 副本延迟过高，leader 副本则会把该 follower 副本提出 ISR 集合，消息依然可以快速提交。当 leader 副本所在的 broker 突然宕机，会优先将 ISR 集合中 follower 副本选举为 leader，新 leader 副本包含了 HW 之前的全部消息，这样就避免了消息的丢失。