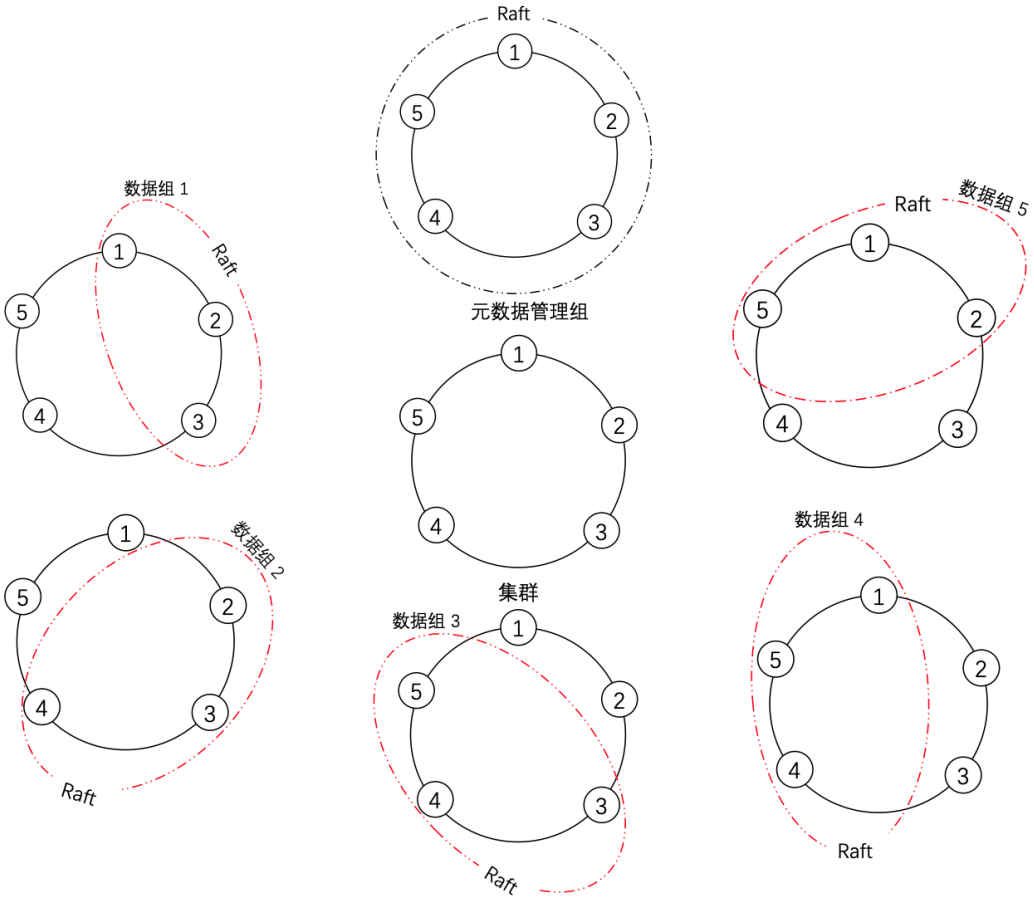
Apache IoTDB集群扩展设计文档

概述

在当前Apche IoTDB的分布式框架中，缺少集群扩展功能。但是集群的可扩展性是提升分布式系统高可用性和性能的必要功能，因此需要提供支持该功能。

Apache IoTDB当前分布式框架概述

双层元数据管理



集群分组(5节点3副本)

以5节点3副本为例，将5个集群节点按照哈希值放在哈希环中，所有节点组成一个元数据组；以每个节点为起始点，在哈希环上顺时针寻找3个节点组成一个数据组。如上图所示，该集群中包括一个元数据组和5个数据组，图中虚线框中代表一个分组。

双层元数据管理方法中包含元数据持有者和数据分区持有者，元数据组的每个节点称为元数据持有者，数据组中的每个节点称为数据分区持有者，采用 raft 协议来保证每个持有者与同组的其他持有者的副本一致性。该方法将元数据按存储组和时间序列两层

粒度分别在元数据持有者和数据分区持有者中管理，由于将时间序列元数据和数据一同在数据组内同步，因此每次数据写入不需要进行元数据的检查与同步操作，仅需要在修改时间序列元数据时进行存储组元数据的检查与同步操作。

数据分区

Apache IoTDB的分布式框架基于一致性哈希和环上查找算法对时序数据按**存储组加上时间片**。时间片指的是一段时间，具体的时间间隔可以由用户指定，可以设定为一小时、一天或者一周等。系统进行读写数据请求的路由时，首先根据一致性哈希算法计算<存储组，时间片>的哈希值，然后放置到哈希环上，顺时针寻找到第一个节点，该节点作为首节点的数据组负责管理该时间序列。节点在处理时间序列元数据的相关请求时，由于请求中不包含时间戳，因此创建/删除时间序列时是以存储组为粒度进行分区路由。

元数据组

集群中所有节点构成元数据组，创建和删除存储组的操作都交由元数据组进行管理。

数据组

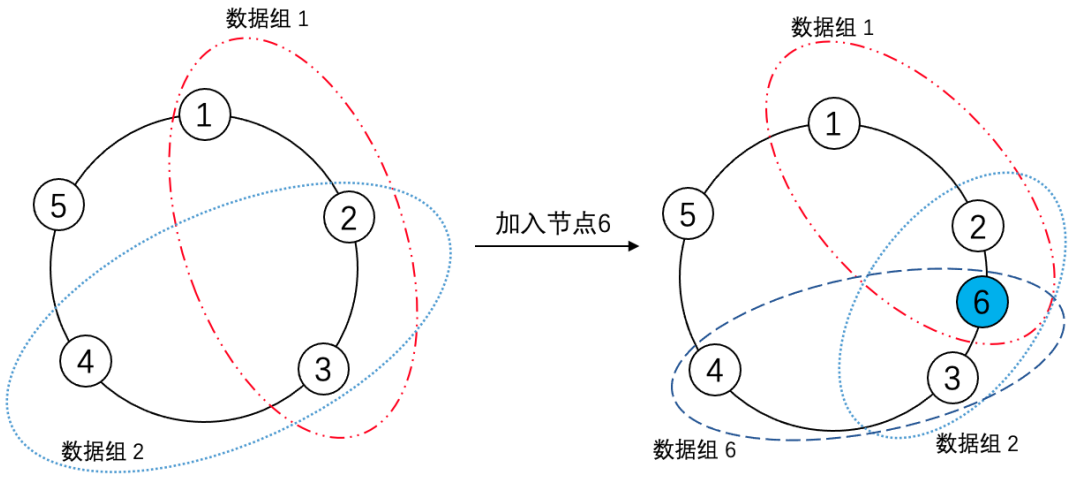
集群中每个节点作为起始点，在哈希环上顺时针找到副本数个节点即可组成一个数据组，因此N个节点的集群共有N个数据组。各个数据组之间的数据不重叠，每个数据组内使用raft协议来保证副本的一致性。

设计方案

两阶段方法

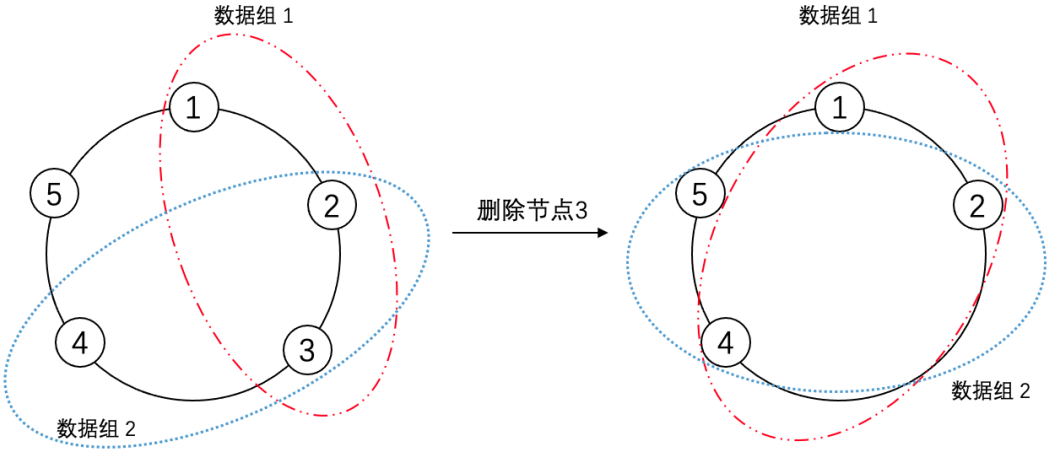
设计思路

在上述Apache IoTDB的分布式框架中，基于一致性哈希的数据组成员分配方法在集群扩展时会造成多个数据组的成员替换，使集群扩展变得更复杂。集群中包含多个数据组，每个数据组是由一致性哈希环中连续的副本数个节点构成。这种方式下在集群增删节点时不仅会存在数据组的增加或者删除，还会存在一些数据组进行了成员替换。



加入节点后的成员变更

如图所示，在5节点3副本的集群中加入节点6后除了增加一个数据组6（数据组成员{6，3，4}）外，数据组1和数据组2发生了成员替换。数据组1中新加入的节点6替换了节点3，数据组成员由{1，2，3}变成了{1，2，6}；数据组2中新加入的节点6替换了节点4，数据组成员由{2，3，4}变成了{2，6，3}。



删除节点后的成员变更

如图所示，在5节点3副本的集群中移除节点3后除了删除了一个数据组3（数据组成员{3，4，5}）外，数据组1和数据组2发生了成员替换。数据组1中节点4替换了节点3，数据组成员由{1，2，3}变成了{1，2，4}；数据组2中节点5替换了节点3，数据组成员由{2，3，4}变成了{2，4，5}。

在集群扩展时，由于会发生数据组成员的替换，而raft组直接进行成员替换可能会产生多个leader，导致脑裂，因此需要拆分成两个步骤分别进行单节点增删变更，后文描述时统一采用先增后删的顺序。此外，为了保证数据的安全性，在进行数据迁移前要保证数据不会再写入旧的数据组中，因此集群扩展期间要有个明确的时间点来标识所有的数据组已应用新的数据分区信息。

方法内容

基于以上思路设计了两阶段方法来保证安全性，具体内容如下：

1. **第一阶段**

用户的增删节点操作转发给元数据组leader来处理，处理过程分为三步：

1）元数据组leader首先进行安全性检查，确保之前的集群扩展已经完成。检查满足要求后，元数据组leader对数据进行重分区并将结果加入增删节点的日志中，然后将这条日志分发给所有元数据组的follower，等到超过半数的follower收到这条日志执行第二步。

2）元数据组leader将这条日志发送给所有的raft数据组。每个数据组收到日志后按照raft流程进行处理，不同点是所有数据组成员在收到集群增删节点的日志时直接提交这条日志（包括leader），即更新本地的集群成员信息和数据分区信息并进行**预成员变更**（*预成员变更*是对发生成员替换的数据组执行第一步增加节点的变更操作）。每个数据组leader按照*预成员变更*后的组内成员信息进行日志分发。

3）当所有的数据组完成增删节点请求后，第一阶段结束并返回给用户集群增加/删除节点成功的消息。若集群扩展操作为增加节点，新加入的节点会正式启动并提供服务。

2. **第二阶段**

元数据组提交增删节点日志，处理过程分为四步：

1） 对本地所有数据组进行**正式成员变更**（*正式成员变更*是对发生成员替换的数据组执行第二步删除节点的变更操作）。

2） 当由于成员替换导致本节点成为某个数据组的新成员时，本地创建并启动该数据组的实例提供服务；当由于成员替换导致某个数据组成员不包含本节点时，本地关闭并删除该数据组的实例。

3）对所有的数据组按照数据重分区情况启动异步的数据迁移任务。当所有数据组的数据迁移完成后，用户就可以进行下一个集群扩展请求。

4）如果是删除节点操作，元数据组leader需要通知被删除的节点执行删除流程。因为第二阶段时元数据组的成员已经不包含被删除的节点，不会和这个节点进行心跳，因此需要主动通知它执行删除流程。

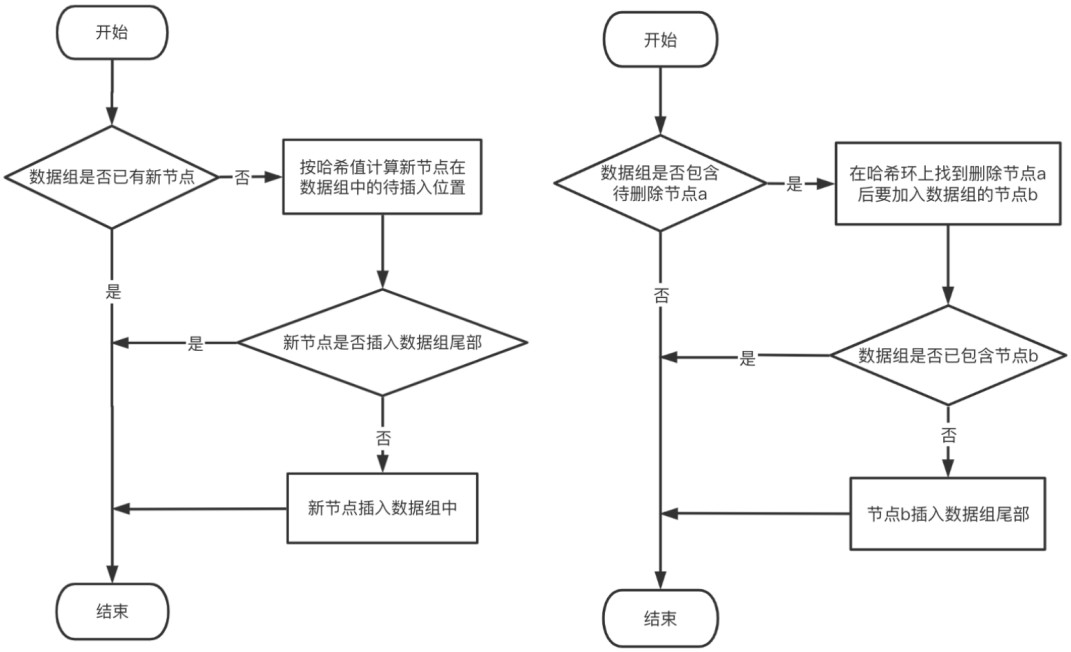
在第一阶段中数据组收到集群增删节点的日志时直接提交日志。只有当raft组内超过半数节点已经提交了上一次成员变更，该raft组才可以进行下一次成员变更操作。如果对于增删节点的日志也像其他类型的日志一样，即数据组成员只有在leader提交了这条日志后才会提交日志， 那么数据组leader将很难知道什么时候数据组中超过半数节点已经采用了新的集群成员信息和数据分区信息，这种情况下就需要添加一些其他的机制来保证；反之，如果在收到增删节点的日志后直接提交日志，那么数据组leader返回执行结果时就可以安全地进行下一次成员变更操作。此外，在集群扩展前检查上次集群扩展是否完成的步骤也保证了下一次*预成员变更*之前上一次的*正式成员变更*已完成，因此该方法**保证了raft数据组成员变更的安全性**。

在两阶段方法中，仅当第一阶段结束时集群中所有的数据组都已经采用了新的数据分区信息后，第二阶段中第三步所有数据组才会启动数据迁移，因此该方法严格限制了变更数据分区信息和数据迁移的执行顺序，确保了第一阶段结束后所有数据写入请求都只会由新的数据组来处理，不会写入旧的数据组，从而解决了数据可能丢失的问题，**保证了数据的安全性**。当用户在第二阶段查询某个分区的数据时，只要将管理该分区的旧数据组的查询结果和新数据组的查询结果合并，就可以返回正确且完整的查询请求结果。

方法实现

两阶段方法中最主要的操作步骤是每个数据组的*预成员变更*和*正式成员变更*。在实现这两个功能时，需要考虑并解决*预成员变更*和*正式成员变更*操作的顺序问题：在系统运行过程中集群大部分节点会先进行*预成员变更*再进行*正式成员变更*，但是存在一些数据组节点先进行了*正式成员变更*再进行*预成员变更*。这是因为*正式成员变更*是在第二阶段元数据组提交增删节点日志时触发的，第一阶段结束时保证了所有数据组超过半数节点进行了*预成员变更*，而那些没有进行*预成员变更*的数据组可能会出现先进行*正式成员变更*的情况。

为了解决上述弊端，在实现时直接将*预成员变更*设计成幂等操作，这样在进行*正式成员变更*操作时不管数据组有没有进行*预成员变更*操作，都再执行一遍*预成员变更*操作，并且保证在*正式成员变更*操作后再进行*预成员变更*操作的正确性。

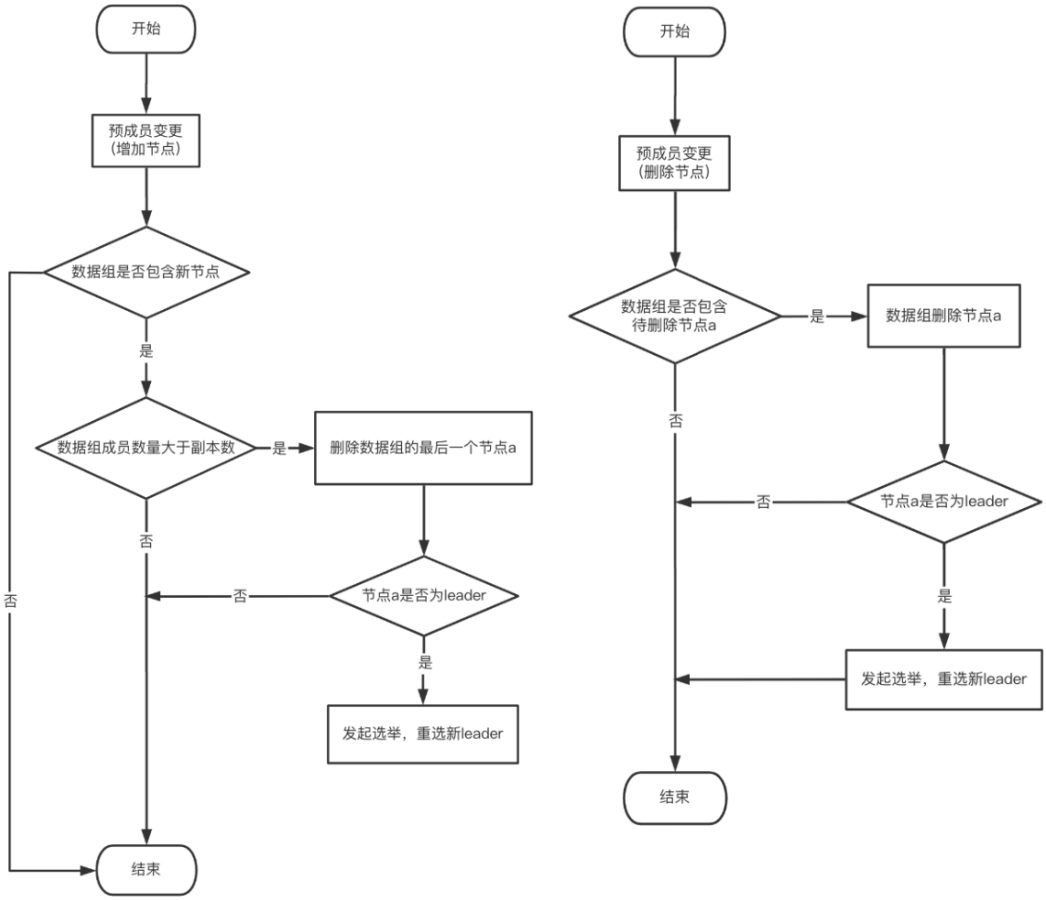


(a) 增加节点　　(b) 删除节点

数据组*预成员变更*流程图

上图描述了*预成员变更*的流程：

- 增加节点时，首先判断数据组是否已经包括了新节点。若数据组包括新节点，则说明数据组已经执行过*预成员变更*，这种情况下应直接返回；若数据组不包括新节点，则根据哈希值找到新节点在哈希环上的位置。如果新节点位于数据组最后一个节点沿着哈希环顺时针到数据组首节点的区间，说明新节点的加入不会引起本数据组的成员变更，否则将节点插入数据组中并返回。
- 删除节点时，首先判断数据组是否包含待删除节点。如果数据组不包含待删除节点，则会有两种情况：一是这个节点的删除本身不会引起本数据组的成员变更，二是数据组已经执行过*正式成员变更*并已经将这个节点删除，这种情况下应直接返回。然后要判断数据组是否已经包括了替换节点，如果包括，则说明数据组之前已经执行过*预成员变更*，直接返回；如果不包括，就将替换的节点加入数据组并返回。



(a) 增加节点　　(b) 删除节点

数据组*正式成员变更*流程图

集群成员变更时，在一些特殊情况下增删节点操作不成功，用户可能会多次触发同一个增删节点操作，因此*正式成员变更*也要保证幂等性，上图描述了*正式成员变更*的流程：

- 增加节点时，首先进行*预成员变更*，然后判断是否包括新节点，如果不包括，则说明新节点的加入不会引起本数据组的成员变更，直接返回。其次，判断数据组成员数量是否大于副本数，如果不是，则说明数据组已经执行过一次*正式成员变更*操作，直接返回。最后删除数据组的最后一个节点，如果该节点是leader，那么应该立刻发起选举选出新leader。
- 删除节点时，首先进行*预成员变更*，然后判断数据组是否包含待删除节点，如果数据组不包含待删除节点，则有两种情况：一是这个节点的删除本身不会引起本数据组的成员变更，二是数据组已经执行过*正式成员变更*并将这个节点删除，这种情况下应直接返回。最后删除这个节点，如果该节点是leader，那么应该立刻发起选举选出新leader。

异常处理

在两阶段处理的过程中每个阶段都有可能产生异常，下面对各种异常情况下的处理进行介绍：

- 如果第一阶段第一步中元数据组leader一直没有收到超过半数节点的回复，超过一定时间后用户会收到超时的响应。此时用户会重新发送集群扩展请求，系统可能会执行多次*预成员变更*和多次*正式成员变更*，由于在实现中保证了操作的幂等性，因此这种情况下不会出现问题。

- 如果第一阶段第二步中某些数据组处理增删节点日志超时，元数据leader会不断地重试直至完成，因此会出现数据组执行多次*预成员变更*操作，同样由于在实现中保证了操作的幂等性，因此提供了正确性的保障。
- 如果第一阶段第一步完成后，元数据组更换了leader，新leader直接进行第二阶段会产生问题。因此在实现时将第一阶段第二步放在元数据组日志提交逻辑中。元数据组成员日志提交时，如果本节点是元数据组leader，那么先将日志分发给所有的数据组。这样保证了即使更换了元数据组leader，第一阶段第二步还是会正常完成。

数据重分区

方法内容

在两阶段方法中，集群成员变更的请求由元数据组leader来处理，因此可以由元数据leader先进行数据分区的修改，然后将修改后的数据分区信息（即分区表）放入增删节点日志中同步给其他节点，这些节点提交这条日志时加载新的数据分区信息，这样可以实现数据重分区的统一管理。

分区表需要版本控制。当集群已经进行过多次集群扩展时，raft组的日志列表中会存在多个集群扩展日志，每个集群扩展日志中都会包含一个分区表。当新加入raft组的节点和leader进行日志同步时会依次执行这些日志并加载历史分区表，这时通过版本可以在加载分区表时判断是否为历史分区表，如果是则直接跳过加载阶段。

数据的重分区可以有多种策略来执行，最基础的策略是将哈希槽的数量均分到所有的数据组中，这种方式适合数据比较均匀的情况，每个哈希槽的数据量都大致相同；但当出现了热点数据时，有些哈希槽的数据量会远多于其他槽的数据，此时的数据重分区策略更适合按照各个节点的实际负载指标来进行分配，需要搭配负载均衡器使用，比如可以按照数据量、写入速率等，使得选定的负载能尽量均分到各个数据组中。

方法实现

SlotPartitionTable属性说明

属性名	说明
List<Node> **allNodes**	记录集群中的所有节点，按哈希值顺序排列
Node[] **slotNodes**	记录管理每个哈希槽的数据组首节点
long **version**	分区表的版本号
SlotBalancer **slotBalancer**	槽分配器，用于在集群增删节点时对数据进行重分区
Map<Node, Map<Integer, Node>> **previousHolder**	记录数据组获得的所有新 slot 的原有持有组，外层的 key 是新分区管理组的首节点，内层的 key 是 slot，value 是旧分区管理组的首节点

集群扩展后由元数据组leader进行数据重分区，并将修改后的分区表随着增删节点日志一起同步给其他节点实现统一管理。为了实现以上功能，分区表类SlotPartitionTable包含的属性如上表所示。

在分区表中，数据组都用首节点来表示，完整的数据组成员可以直接在哈希环上找到首节点后顺序查找即可。在分区表的属性中，版本号用于区分不同的集群扩展操作导

致的数据重分区结果。当一个新节点加入集群时，作为元数据组的新成员，follower要和元数据组的leader进行日志同步并依次提交所有的历史集群增删节点操作，此时通过版本号可以直接将这些旧操作跳过。具体实现中采用**元数据组的日志编号**作为版本号。

SlotPartitionTable 接口说明

接口名	说明
List<Node> **route**(String *sgName*, long *time*)	给定存储组和时间戳，返回管理该存储组下的给定时间片的数据组
boolean **judgeHoldSlot**(Node *node*, int *slot*)	判断节点 node 是否管理槽 slot 的数据
void **addNode**(Node *node*)	集群增加节点时，更新分区表的相关属性并用槽分配器进行数据重分区
void **removeNode**(Node *node*)	集群删除节点时，更新分区表的相关属性并用槽分配器槽进行数据重分区
ByteBuffer **serialize**()	序列化分区表，当元数据组 leader 进行分区表更新结果同步和将分区表持久化到磁盘时使用
void **deserialize**(ByteBuffer *buffer*)	反序列化分区表，当节点提交分区表更新结果和节点初始化时使用

分区表的作用主要是数据路由和增删节点时对数据进行重分区，为了支持这些功能，分区表类SlotPartitionTable包含的接口上表所示。增删节点的操作主要通过SlotBalancer进行数据重分区，槽分配器主要包含两个接口，如下表所示。当前实现中采用的是哈希槽均匀分配策略：当增加节点时，首先用总槽数除以节点数算出各个节点应当管理的哈希槽数，然后从各个节点将多余的槽交由新节点管理；当删除节点时，待删除节点的槽均匀地分配给其他节点。

LoadBalancer 接口说明

接口名	说明
void **moveSlotsToNew**(Node *node*)	当加入节点时，根据一定策略从其他节点接管一些槽，当前实现采用均分哈希槽策略
void **retrieveSlots**(Node *node*)	当删除节点时，根据一定策略将待删除节点管理的槽分配给其他节点，当前实现采用均分哈希槽策略

数据迁移

分区方法改进

采用哈希槽分区方式。将一致性哈希环划分为一定数量相等大小的槽（默认为10000），然后将这些哈希槽分配给数据组。

在Apache IoTDB的存储引擎中，每个存储组按照时间片分开进行数据的管理。这样存储组的每个时间片内所有TsFile文件的数据都属于同一个槽，每个槽管理多个存储组的多个时间片数据，因此在进行数据迁移时可以以时间片为单位进行整体迁移。

方法内容

数据迁移的内容分为两部分，一部分是时序数据，即一系列TsFile数据文件；另一部分是时间序列的元数据。旧数据组节点和新数据组节点之间的数据迁移流程如下：

1）每个新数据组节点选择一个旧数据组节点，并向旧数据组节点请求接管的哈希槽数据。

2）旧数据组节点对所有需要迁移的存储组时间片数据执行flush操作，然后收集需要迁移的TsFile文件列表和属于该分区的时间序列元信息，最后向新数据组节点返回结果。

3）收到结果后，新数据组节点加载时间序列元信息，并将文件列表中的所有文件从旧数据组节点拉取到本地并加载。所有文件加载完毕后，通知旧数据组的所有节点一个副本的数据迁移完成。

4）当旧数据组的各个节点收到副本数个通知后，说明新数据组的所有副本都完成了数据迁移，此时可以将本地已完成迁移的数据进行删除，数据迁移结束。

整个数据迁移流程中有以下几点设计细节：

1）由于系统中每个TsFile文件中的数据都属于同一个哈希槽，因此旧数据组节点在进行flush操作时仅flush需要迁移的存储组时间片的MemTable即可，减少了不必要的flush操作。

2) 由于集群中每个数据组都是由n个节点组成的n副本raft组，因此新数据组的n个节点都要进行数据迁移，如果全都和旧数据组的leader执行数据迁移流程，那么旧数据组leader的资源消耗会很大。为了均摊数据迁移的负载，新数据组节点和旧数据组节点按照一对一的方式进行数据迁移，可以最大化数据迁移效率。因此在第一步选择旧数据组节点时，新数据组节点按照自己在组内的位置去找旧数据组中相同位置的节点，如新数据组成员为{1，2，3}且旧数据组成员为{4，5，6}，那么节点1选择节点4、节点2选择节点5、节点3选择节点6进行数据迁移。这种方法下旧数据组follower需要和leader同步保证数据是最新的。

3) 在进行数据迁移时，新数据组的节点有可能同时也是旧数据组的成员，以新数据组成员是{1，2，3}且旧数据组成员是{3，4，5}为例，由于节点3本地已有相应的数据，这种情况下不需要进行数据迁移流程，直接通知旧数据组的所有节点一个副本的数据迁移完成；此外，执行第四步骤时节点3作为旧数据组成员由于同时也是新数据组成员，因此本地不能将相应的数据删除。

方法实现

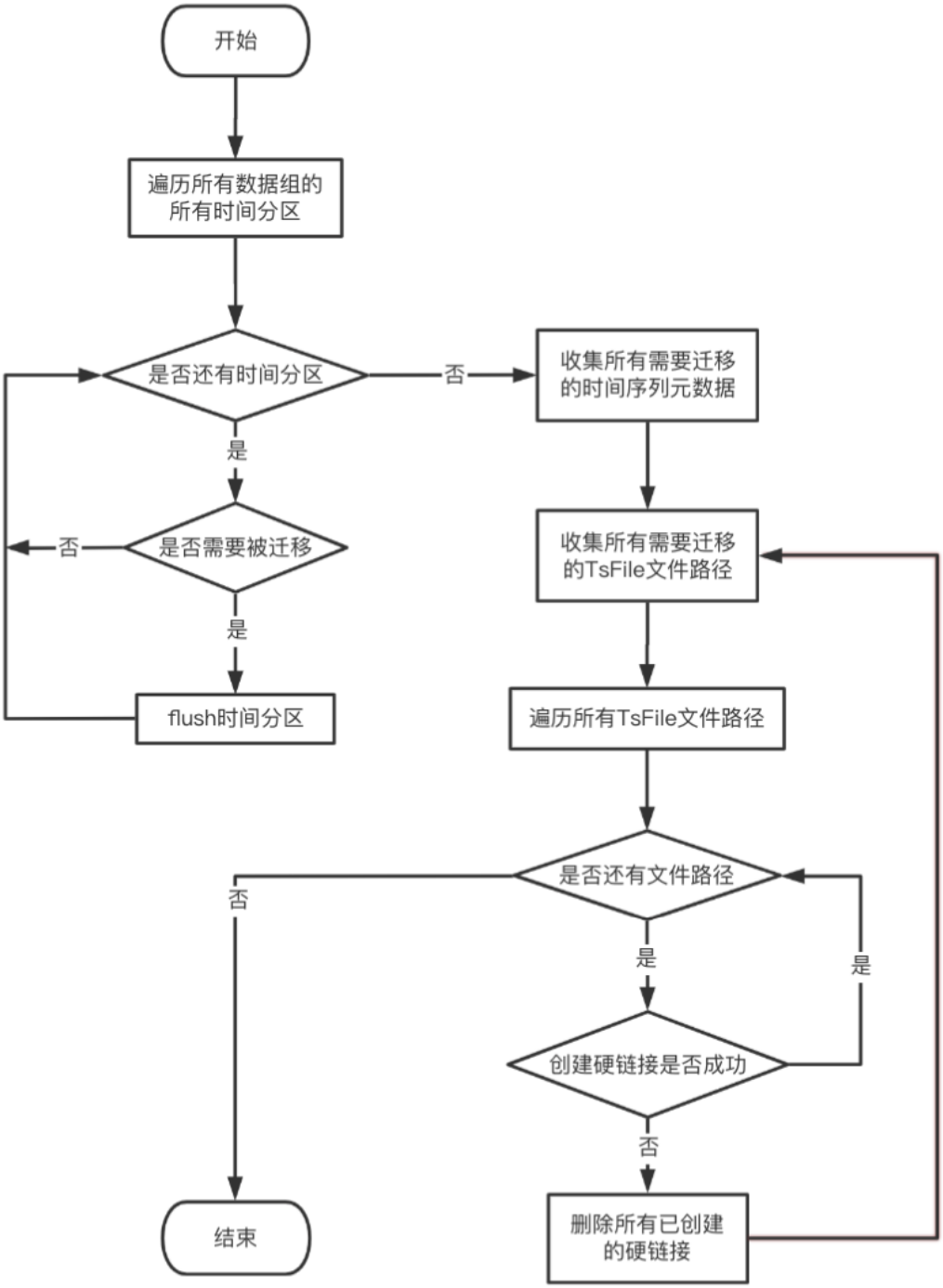
获取快照

旧分区持有者收到新分区持有者获取某些分区的数据快照请求后，首先会检查本地数据是否完整，因为数据组的非leader节点的日志是有滞后性的，数据迁移时需要保证数据的完整性。如果本地数据完整，则执行获取快照流程。

获取快照首先要对所有需要迁移的存储组时间分区执行flush操作实现内存中的数据落盘，由于在此前已经保证了本地数据完整，即数据组日志最新并且本地的分区表信息最新，因此数据组不会再处理新的属于该分区的数据写入请求，保证了所有需要传输的数据都在磁盘上的TsFile文件中。

旧分区持有者需要返回的内容分为两部分：一部分是时间序列元数据，另一部分是所有待迁移文件的路径。由于存储引擎存在merge操作，新分区持有者拉取文件时，可能会由于merge操作已经将文件删除，从而导致数据迁移的不完整性。为了解决这个问

题，可以采用对所有文件创建硬链接的方式来保证快照请求者拉取文件时本地文件存在。



数据迁移获取快照流程图

如上图所示，获取文件数据快照首先收集所有需要迁移的TsFile文件路径，然后全部创建硬链接，当且仅当所有文件的硬链接创建成功才算完成；否则，说明存在文件已经因为merge操作而被删除，这种情况下需要删除所有已创建的硬链接并重新执行流程。当新分区持有者拉取完所有文件后再将硬链接全部移除。

安装快照

新分区持有者收到快照后，首先直接加载快照中的元数据注册时间序列，然后依次拉取远程TsFile文件，最后调用加载外部tsfile模块进行处理。