Zookeeper 架构及 FastLeaderElection 机制

Zookeeper 是什么

Zookeeper 是一个分布式协调服务，可用于服务发现，分布式锁，分布式领导选举，配置管理等。

这一切的基础，都是 Zookeeper 提供了一个类似于 Linux 文件系统的树形结构（可认为是轻量级的内存文件系统，但只适合存少量信息，完全不适合存储大量文件或者大文件），同时提供了对于每个节点的监控与通知机制。

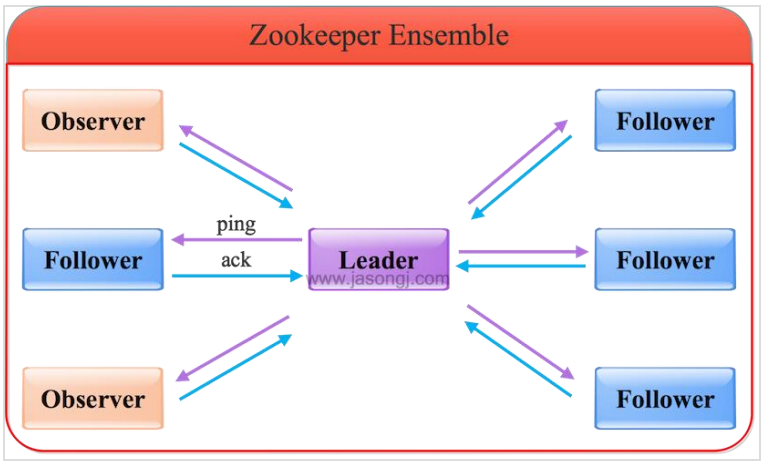
既然是一个文件系统，就不得不提 Zookeeper 是如何保证数据的一致性的。本文将介绍 Zookeeper 如何保证数据一致性，如何进行领导选举，以及数据监控/通知机制的语义保证。

Zookeeper 架构

角色

Zookeeper 集群是一个基于主从复制的高可用集群，每个服务器承担如下三种角色中的一种

- Leader 一个 Zookeeper 集群同一时间只会有一个实际工作的 Leader，它会发起并维护与各 Follwer 及 Observer 间的心跳。所有的写操作必须要通过 Leader 完成再由 Leader 将写操作广播给其它服务器。
- Follower 一个 Zookeeper 集群可能同时存在多个 Follower，它会响应 Leader 的心跳。Follower 可直接处理并返回客户端的读请求，同时会将写请求转发给 Leader 处理，并且负责在 Leader 处理写请求时对请求进行投票。
- Observer 角色与 Follower 类似，但是无投票权。



原子广播（ZAB）

为了保证写操作的一致性与可用性，Zookeeper 专门设计了一种名为原子广播（ZAB）的支持崩溃恢复的一致性协议。基于该协议，Zookeeper 实现了一种主从模式的系统架构来保持集群中各个副本之间的数据一致性。

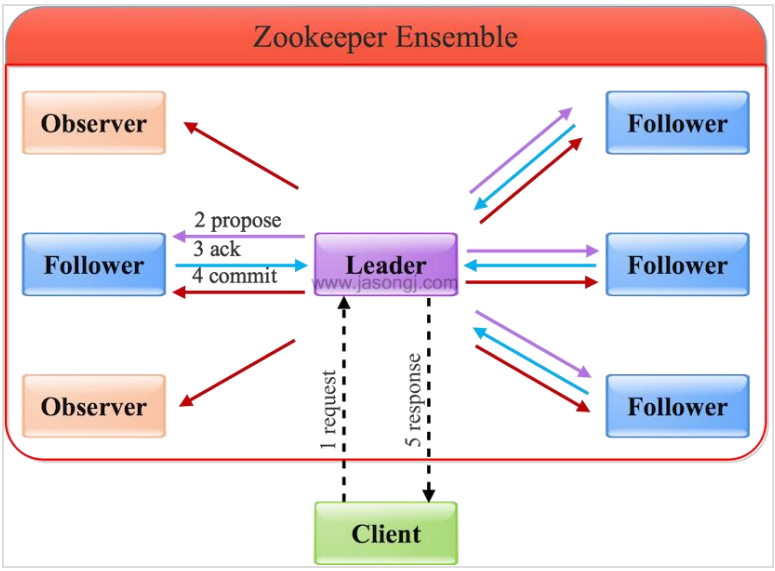
根据 ZAB 协议，所有的写操作都必须通过 Leader 完成，Leader 写入本地日志后再复制到所有的 Follower 节点。

一旦 Leader 节点无法工作，ZAB 协议能够自动从 Follower 节点中重新选出一个合适的替代者，即新的 Leader，该过程即为领导选举。该领导选举过程，是 ZAB 协议中最为重要和复杂的过程。

写操作

写 Leader

通过 Leader 进行写操作流程如下图所示



由上图可见，通过 Leader 进行写操作，主要分为五步：

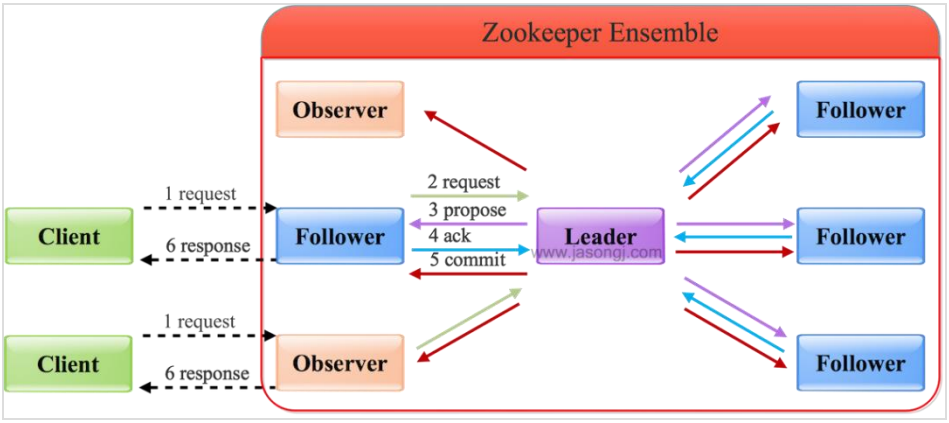
1. 客户端向 Leader 发起写请求
2. Leader 将写请求以 Proposal 的形式发给所有 Follower 并等待 ACK
3. Follower 收到 Leader 的 Proposal 后返回 ACK
4. Leader 得到过半数的 ACK（Leader 对自己默认有一个 ACK）后向所有的 Follower 和 Observer 发送 Commmit
5. Leader 将处理结果返回给客户端

这里要注意

- Leader 并不需要得到 Observer 的 ACK，即 Observer 无投票权
- Leader 不需要得到所有 Follower 的 ACK，只要收到过半的 ACK 即可，同时 Leader 本身对自己有一个 ACK。上图中有 4 个 Follower，只需其中两个返回 ACK 即可，因为(2+1) / (4+1) > 1/2
- Observer 虽然无投票权，但仍须同步 Leader 的数据从而在处理读请求时可以返回尽可能新的数据

写 Follower/Observer

通过 Follower/Observer 进行写操作流程如下图所示：

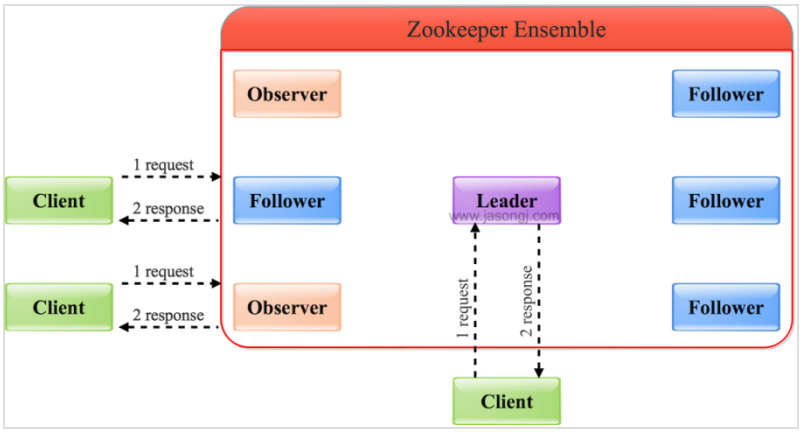


从上图可见

- Follower/Observer 均可接受写请求，但不能直接处理，而需要将写请求转发给 Leader 处理
- 除了多了一步请求转发，其它流程与直接写 Leader 无任何区别

读操作

Leader/Follower/Observer 都可直接处理读请求，从本地内存中读取数据并返回给客户端即可。



由于处理读请求不需要服务器之间的交互，Follower/Observer 越多，整体可处理的读请求量越大，也即读性能越好。

FastLeaderElection 原理

术语介绍

myid

每个 Zookeeper 服务器，都需要在数据文件夹下创建一个名为 myid 的文件，该文件包含整个 Zookeeper 集群唯一的 ID（整数）。例如某 Zookeeper 集群包含三台服务器，hostname 分别为 zoo1、zoo2 和 zoo3，其 myid 分别为 1、2 和 3，则在配置文件中其 ID 与 hostname 必须一一对应，如下所示。在该配置文件中，`server.`后面的数据即为 myid

```
1server.1=zoo1:2888:3888
2server.2=zoo2:2888:3888
3server.3=zoo3:2888:3888
```

zxid

类似于 RDBMS 中的事务 ID，用于标识一次更新操作的 Proposal ID。为了保证顺序性，该 zkid 必须单调递增。因此 Zookeeper 使用一个 64 位的数来表示，高 32 位是 Leader 的 epoch，从 1 开始，每次选出新的 Leader，epoch 加一。低 32 位为该 epoch 内的序号，每次 epoch 变化，都将低 32 位的序号重置。这样保证了 zkid 的全局递增性。

支持的领导选举算法

可通过 `electionAlg` 配置项设置 Zookeeper 用于领导选举的算法。

到 3.4.10 版本为止，可选项有

- `0` 基于 UDP 的 LeaderElection
- `1` 基于 UDP 的 FastLeaderElection
- `2` 基于 UDP 和认证的 FastLeaderElection
- `3` 基于 TCP 的 FastLeaderElection

在 3.4.10 版本中，默认值为 3，也即基于 TCP 的 FastLeaderElection。另外三种算法已经被弃用，并且有计划在之后的版本中将它们彻底删除而不再支持。

FastLeaderElection

FastLeaderElection 选举算法是标准的 Fast Paxos 算法实现，可解决 LeaderElection 选举算法收敛速度慢的问题。

服务器状态

- LOOKING 不确定 Leader 状态。该状态下的服务器认为当前集群中没有 Leader，会发起 Leader 选举
- FOLLOWING 跟随者状态。表明当前服务器角色是 Follower，并且它知道 Leader 是谁
- LEADING 领导者状态。表明当前服务器角色是 Leader，它会维护与 Follower 间的心跳
- OBSERVING 观察者状态。表明当前服务器角色是 Observer，与 Folower 唯一的不同在于不参与选举，也不参与集群写操作时的投票

选票数据结构

每个服务器在进行领导选举时，会发送如下关键信息

- *logicClock* 每个服务器会维护一个自增的整数，名为 *logicClock*，它表示这是该服务器发起的第多少轮投票
- *state* 当前服务器的状态
- *self_id* 当前服务器的 *myid*
- *self_zxid* 当前服务器上所保存的数据的最大 *zxid*
- *vote_id* 被推举的服务器的 *myid*
- *vote_zxid* 被推举的服务器上所保存的数据的最大 *zxid*

投票流程

自增选举轮次

Zookeeper 规定所有有效的投票都必须在同一轮次中。每个服务器在开始新一轮投票时，会先对自己维护的 *logicClock* 进行自增操作。

初始化选票

每个服务器在广播自己的选票前，会将自己的投票箱清空。该投票箱记录了所收到的选票。例：服务器 2 投票给服务器 3，服务器 3 投票给服务器 1，则服务器 1 的投票箱为(2, 3), (3, 1), (1, 1)。票箱中只会记录每一投票者的最后一票，如投票者更新自己的选票，则其它服务器收到该新选票后会在自己票箱中更新该服务器的选票。

发送初始化选票

每个服务器最开始都是通过广播把票投给自己。

接收外部投票

服务器会尝试从其它服务器获取投票，并记入自己的投票箱内。如果无法获取任何外部投票，则会确认自己是否与集群中其它服务器保持着有效连接。如果是，则再次发送自己的投票；如果否，则马上与之建立连接。

判断选举轮次

收到外部投票后，首先会根据投票信息中所包含的 *logicClock* 来进行不同处理

- 外部投票的 *logicClock* 大于自己的 *logicClock*。说明该服务器的选举轮次落后于其它服务器的选举轮次，立即清空自己的投票箱并将自己的 *logicClock* 更新为收到的 *logicClock*，然后再对比自己之前的投票与收到的投票以确定是否需要变更自己的投票，最终再次将自己的投票广播出去。
- 外部投票的 *logicClock* 小于自己的 *logicClock*。当前服务器直接忽略该投票，继续处理下一个投票。
- 外部投票的 *logickClock* 与自己的相等。当时进行选票 *PK*。

选票 *PK*

选票 *PK* 是基于(*self_id*, *self_zxid*)与(*vote_id*, *vote_zxid*)的对比

- 外部投票的 *logicClock* 大于自己的 *logicClock*，则将自己的 *logicClock* 及自己的选票的 *logicClock* 变更为收到的 *logicClock*

- 若 logicClock 一致，则对比二者的 vote_zxid，若外部投票的 vote_zxid 比较大，则将自己的票中的 vote_zxid 与 vote_myid 更新为收到的票中的 vote_zxid 与 vote_myid 并广播出去，另外将收到的票及自己更新后的票放入自己的票箱。如果票箱内已存在(self_myid, self_zxid)相同的选票，则直接覆盖
- 若二者 vote_zxid 一致，则比较二者的 vote_myid，若外部投票的 vote_myid 比较大，则将自己的票中的 vote_myid 更新为收到的票中的 vote_myid 并广播出去，另外将收到的票及自己更新后的票放入自己的票箱

统计选票

如果已经确定有过半服务器认可了自己的投票（可能是更新后的投票），则终止投票。否则继续接收其它服务器的投票。

更新服务器状态

投票终止后，服务器开始更新自身状态。若过半的票投给了自己，则将自己的服务器状态更新为 LEADING，否则将自己的状态更新为 FOLLOWING

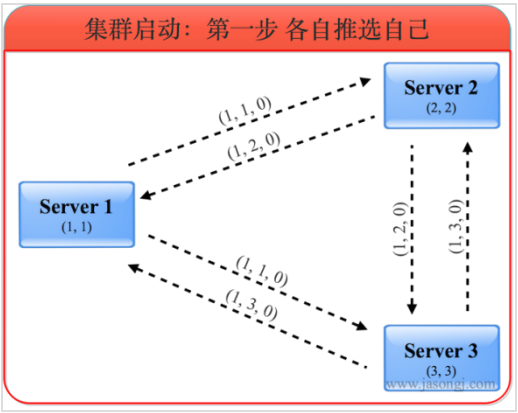
几种领导选举场景

集群启动领导选举

初始投票给自己

集群刚启动时，所有服务器的 logicClock 都为 1，zxid 都为 0。

各服务器初始化后，都投票给自己，并将自己的一票存入自己的票箱，如下图所示。

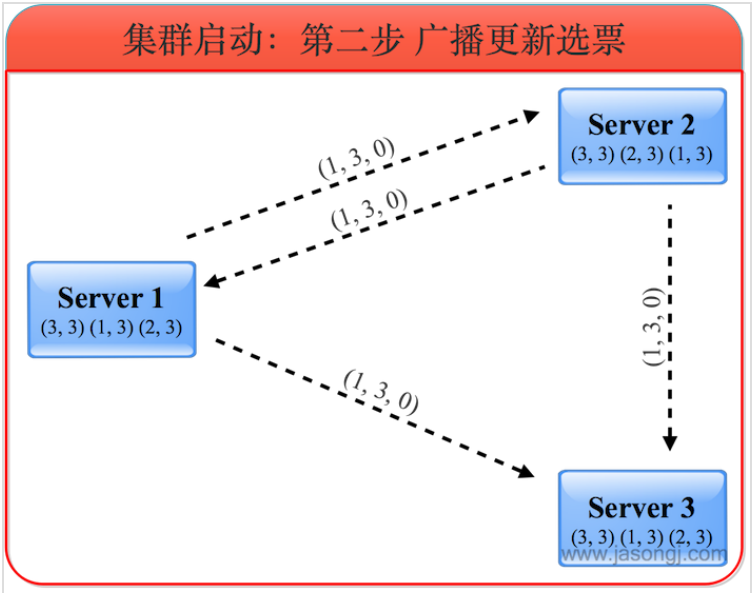


在上图中，(1, 1, 0)第一位数代表投出该选票的服务器的 logicClock，第二位数代表被推荐的服务器的 myid，第三位代表被推荐的服务器的最大的 zxid。由于该步骤中所有选票都投给自己，所以第二位的 myid 即是自己的 myid，第三位的 zxid 即是自己的 zxid。

此时各自的票箱中只有自己投给自己的一票。

更新选票

服务器收到外部投票后，进行选票 PK，相应更新自己的选票并广播出去，并将合适的选票存入自己的票箱，如下图所示。



服务器 1 收到服务器 2 的选票（1，2，0）和服务器 3 的选票（1，3，0）后，由于所有的 logicClock 都相等，所有的 zxid 都相等，因此根据 myid 判断应该将自己的选票按照服务器 3 的选票更新为（1，3，0），并将自己的票箱全部清空，再将服务器 3 的选票与自己的选票存入自己的票箱，接着将自己更新后的选票广播出去。此时服务器 1 票箱内的选票为(1，3)，(3，3)。

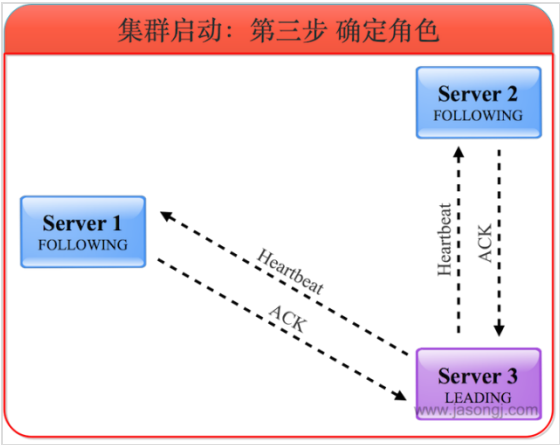
同理，服务器 2 收到服务器 3 的选票后也将自己的选票更新为（1，3，0）并存入票箱然后广播。此时服务器 2 票箱内的选票为(2，3)，(3，,3)。

服务器 3 根据上述规则，无须更新选票，自身的票箱内选票仍为（3，3）。

服务器 1 与服务器 2 更新后的选票广播出去后，由于三个服务器最新选票都相同，最后三者的票箱内都包含三张投给服务器 3 的选票。

根据选票确定角色

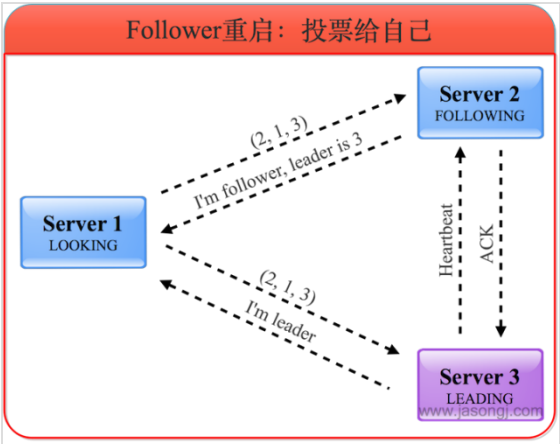
根据上述选票，三个服务器一致认为此时服务器 3 应该是 Leader。因此服务器 1 和 2 都进入 FOLLOWING 状态，而服务器 3 进入 LEADING 状态。之后 Leader 发起并维护与 Follower 间的心跳。



Follower 重启

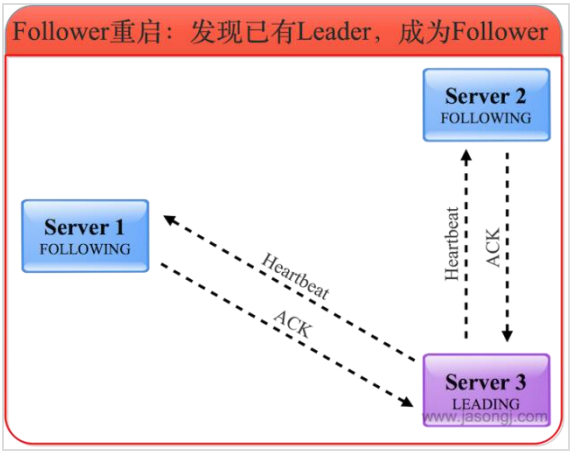
Follower 重启投票给自己

Follower 重启，或者发生网络分区后找不到 Leader，会进入 LOOKING 状态并发起新的一轮投票。



发现已有 Leader 后成为 Follower

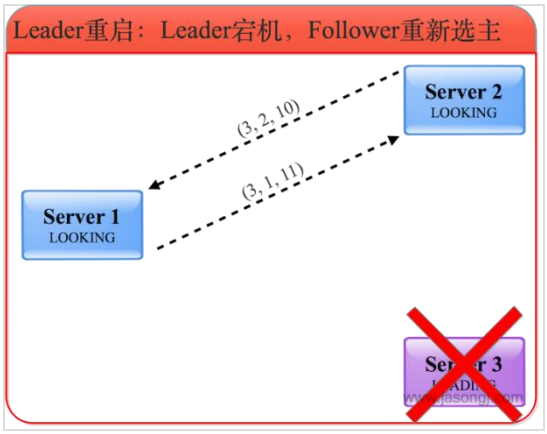
服务器 3 收到服务器 1 的投票后，将自己的状态 LEADING 以及选票返回给服务器 1。服务器 2 收到服务器 1 的投票后，将自己的状态 FOLLOWING 及选票返回给服务器 1。此时服务器 1 知道服务器 3 是 Leader，并且通过服务器 2 与服务器 3 的选票可以确定服务器 3 确实得到了超过半数的选票。因此服务器 1 进入 FOLLOWING 状态。



Leader 重启

Follower 发起新投票

Leader（服务器 3）宕机后，Follower（服务器 1 和 2）发现 Leader 不工作了，因此进入 LOOKING 状态并发起新的一轮投票，并且都将票投给自己。

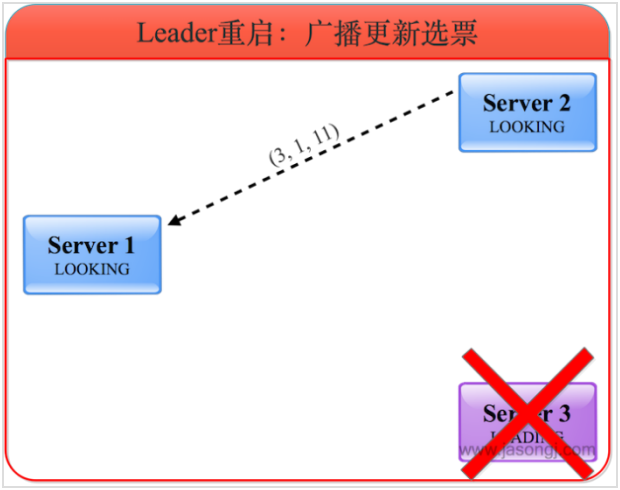


广播更新选票

服务器 1 和 2 根据外部投票确定是否要更新自身的选票。这里有两种情况

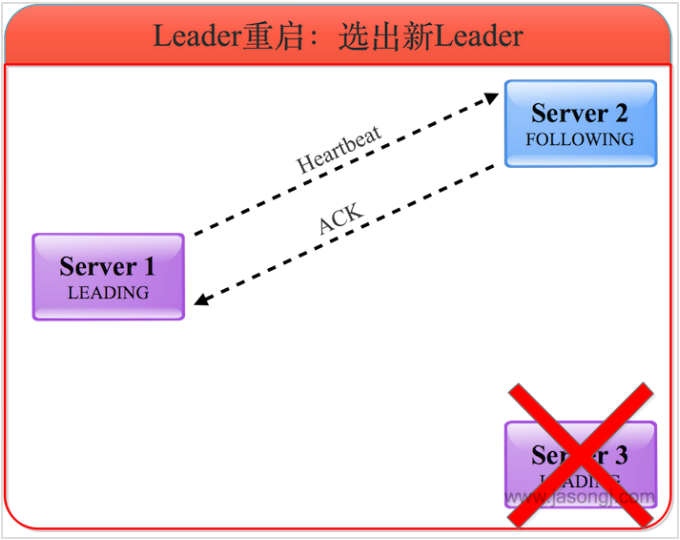
- 服务器 1 和 2 的 zxid 相同。例如在服务器 3 宕机前服务器 1 与 2 完全与之同步。此时选票的更新主要取决于 myid 的大小
- 服务器 1 和 2 的 zxid 不同。在旧 Leader 宕机之前，其所主导的写操作，只需过半服务器确认即可，而不需所有服务器确认。换句话说，服务器 1 和 2 可能一个与旧 Leader 同步（即 zxid 与之相同）另一个不同步（即 zxid 比之小）。此时选票的更新主要取决于谁的 zxid 较大

在上图中，服务器 1 的 zxid 为 11，而服务器 2 的 zxid 为 10，因此服务器 2 将自身选票更新为（3, 1, 11），如下图所示。



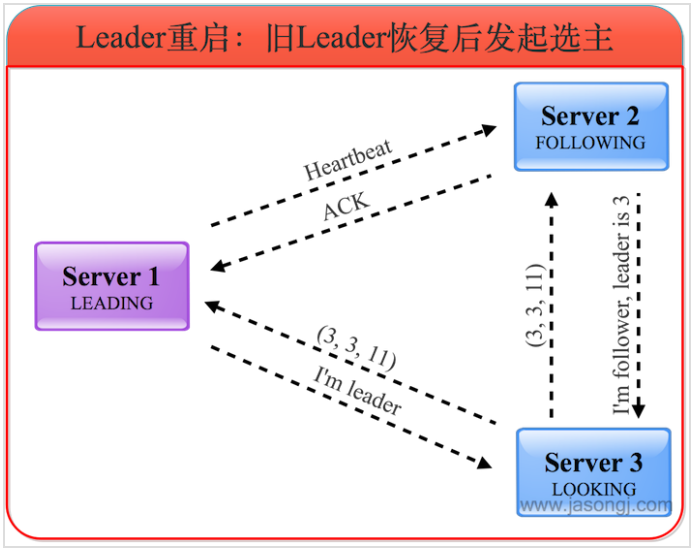
选出新 Leader

经过上一步选票更新后，服务器 1 与服务器 2 均将选票投给服务器 1，因此服务器 2 成为 Follower，而服务器 1 成为新的 Leader 并维护与服务器 2 的心跳。



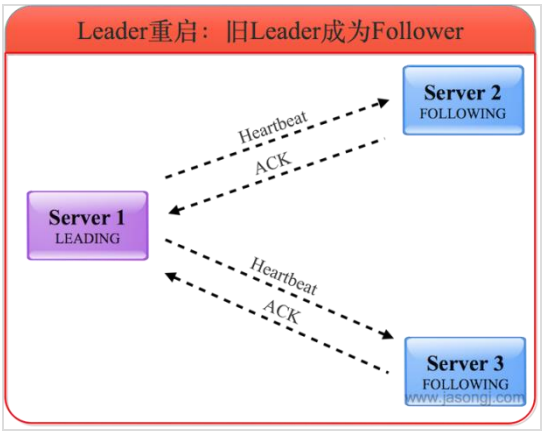
旧 Leader 恢复后发起选举

旧的 Leader 恢复后，进入 LOOKING 状态并发起新一轮领导选举，并将选票投给自己。此时服务器 1 会将自己的 LEADING 状态及选票（3，1，11）返回给服务器 3，而服务器 2 将自己的 FOLLOWING 状态及选票（3，1，11）返回给服务器 3。如下图所示。



旧 Leader 成为 Follower

服务器 3 了解到 Leader 为服务器 1，且根据选票了解到服务器 1 确实得到过半服务器的选票，因此自己进入 FOLLOWING 状态。



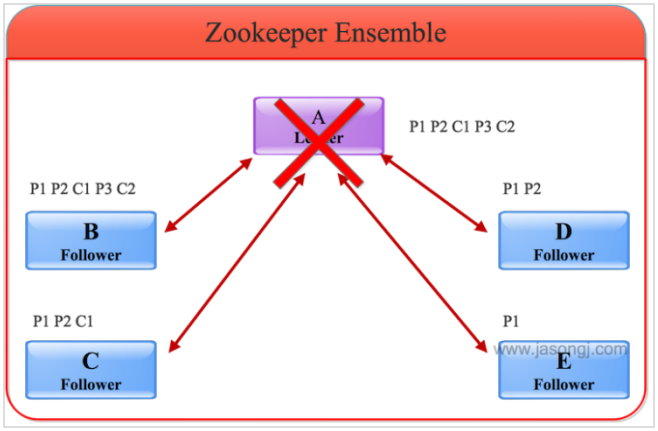
一致性保证

ZAB 协议保证了在 Leader 选举的过程中，已经被 Commit 的数据不会丢失，未被 Commit 的数据对客户端不可见。

Commit 过的数据不丢失

Failover 前状态

为更好演示 Leader Failover 过程，本例中共使用 5 个 Zookeeper 服务器。A 作为 Leader，共收到 P1、P2、P3 三条消息，并且 Commit 了 1 和 2，且总体顺序为 P1、P2、C1、P3、C2。根据顺序性原则，其它 Follower 收到的消息的顺序肯定与之相同。其中 B 与 A 完全同步，C 收到 P1、P2、C1，D 收到 P1、P2，E 收到 P1，如下图所示。

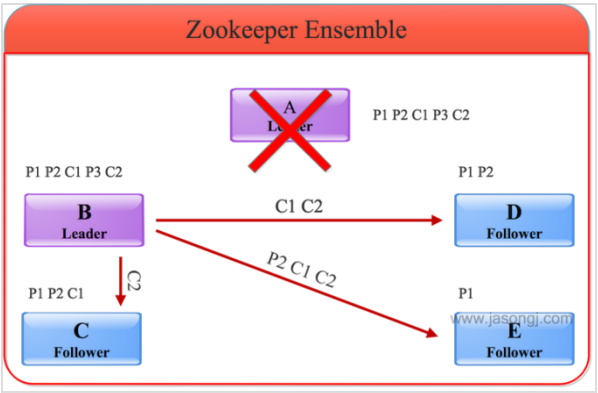


这里要注意

- 由于 A 没有 C3，意味着收到 P3 的服务器的总个数不会超过一半，也即包含 A 在内最多只有两台服务器收到 P3。在这里 A 和 B 收到 P3，其它服务器均未收到 P3
- 由于 A 已写入 C1、C2，说明它已经 Commit 了 P1、P2，因此整个集群有超过一半的服务器，即最少三个服务器收到 P1、P2。在这里所有服务器都收到了 P1，除 E 外其它服务器也都收到了 P2

选出新 Leader

旧 Leader 也即 A 宕机后，其它服务器根据上述 FastLeaderElection 算法选出 B 作为新的 Leader。C、D 和 E 成为 Follower 且以 B 为 Leader 后，会主动将自己最大的 zxid 发送给 B，B 会将 Follower 的 zxid 与自身 zxid 间的所有被 Commit 过的消息同步给 Follower，如下图所示。

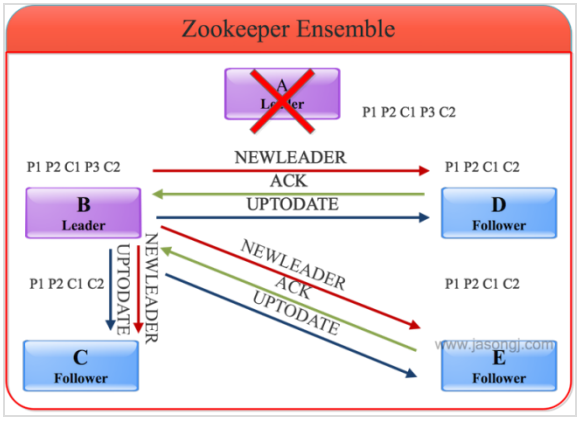


在上图中

- P1 和 P2 都被 A Commit，因此 B 会通过同步保证 P1、P2、C1 与 C2 都存在于 C、D 和 E 中
- P3 由于未被 A Commit，同时幸存的所有服务器中 P3 未存在于大多数据服务器中，因此它不会被同步到其它 Follower

通知 Follower 可对外服务

同步完数据后，B 会向 D、C 和 E 发送 NEWLEADER 命令并等待大多数服务器的 ACK（下图中 D 和 E 已返回 ACK，加上 B 自身，已经占集群的大多数），然后向所有服务器广播 UPTODATE 命令。收到该命令后的服务器即可对外提供服务。



未 Commit 过的消息对客户端不可见

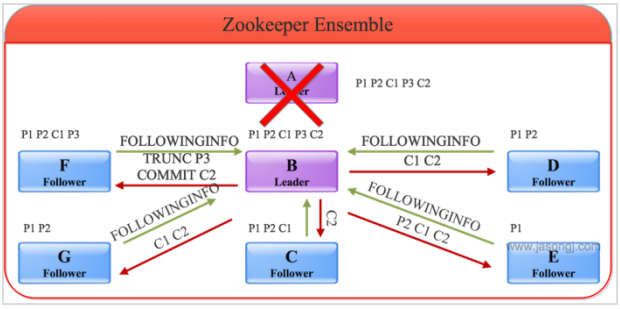
在上例中，P3 未被 A Commit 过，同时因为没有过半的服务器收到 P3，因此 B 也未 Commit P3（如果有过半服务器收到 P3，即使 A 未 Commit P3，B 会主动 Commit P3，即 C3），所以它不会将 P3 广播出去。

具体做法是，B 在成为 Leader 后，先判断自身未 Commit 的消息（本例中即 P3）是否存在于大多数服务器中从而决定是否要将其 Commit。然后 B 可得出自身所包含的被 Commit 过的消息中的最小 zxid（记为 min_zxid）与最大 zxid（记为 max_zxid）。C、D 和 E 向 B 发送自身 Commit 过的最大消息 zxid（记为 max_zxid）以及未被 Commit 过的所有消息（记为 zxid_set）。B 根据这些信息作出如下操作

- 如果 Follower 的 max_zxid 与 Leader 的 max_zxid 相等，说明该 Follower 与 Leader 完全同步，无须同步任何数据
- 如果 Follower 的 max_zxid 在 Leader 的(min_zxid，max_zxid)范围内，Leader 会通过 TRUNC 命令通知 Follower 将其 zxid_set 中大于 Follower 的 max_zxid（如果有）的所有消息全部删除

上述操作保证了未被 Commit 过的消息不会被 Commit 从而对外不可见。

上述例子中 Follower 上并不存在未被 Commit 的消息。但可考虑这种情况，如果将上述例子中的服务器数量从五增加到七，服务器 F 包含 P1、P2、C1、P3，服务器 G 包含 P1、P2。此时服务器 F、A 和 B 都包含 P3，但是因为票数未过半，因此 B 作为 Leader 不会 Commit P3，而会通过 TRUNC 命令通知 F 删除 P3。如下图所示。



总结

- 由于使用主从复制模式，所有的写操作都要由 *Leader* 主导完成，而读操作可通过任意节点完成，因此 *Zookeeper* 读性能远好于写性能，更适合读多写少的场景
- 虽然使用主从复制模式，同一时间只有一个 *Leader*，但是 *Failover* 机制保证了集群不存在单点失败（*SPOF*）的问题
- *ZAB* 协议保证了 *Failover* 过程中的数据一致性
- 服务器收到数据后先写本地文件再进行处理，保证了数据的持久性