zookeeper选举机制

ZooKeeper是一个开源分布式协调服务、分布式数据一致性解决方案。可基于ZooKeeper实现命名服务、集群管理、Master选举、分布式锁等功能。

本文主要分析的是zookeeper的选举机制，zookeeper提供三种方式进行选举leader，主要分析默认的`FastLeaderElection`算法。

主要角色

ZooKeeper没有使用Master/Slave的概念，每个服务器上的数据是相同的，每一个服务器可以对外提供读的服务。

而zookeeper将集群中的节点分为了3类角色：

Leader

在一个ZooKeeper集群中，只能存在一个Leader，这个Leader是集群中事务请求唯一的调度者和处理者，所谓事务请求是指会改变集群状态的请求。Leader根据事务ID可以保证事务处理的顺序性。
如果一个集群中存在多个Leader，这种现象称为「脑裂」。

试想一下，一个集群中存在多个Leader会产生什么影响？相当于原本一个大集群，裂出多个小集群，他们之间的数据是不会相互同步的。「脑裂」后集群中的数据会变得非常混乱。

Follower

Follower角色的ZooKeeper服务只能处理非事务请求，如果接收到客户端事务请求会将请求转发给Leader服务器。

参与Leader选举，参与Leader事务处理投票处理。Follower发现集群中Leader不可用时会变更自身状态，并发起Leader选举投票，最终集群中的某个Follower会被选为Leader。

Observer

Observer与Follower很像，可以处理非事务请求；将事务请求转发给Leader服务器。
与Follower不同的是，Observer不会参与Leader选举；不会参与Leader事务处理投票。
Observer用于不影响集群事务处理能力的前提下提升集群的非事务处理能力。

相关概念

Leader选举

Leader在集群中是非常重要的一个角色，负责了整个事务的处理和调度，保证分布式数据一致性的关键所在。
既然Leader在ZooKeeper集群中这么重要所以一定要保证集群在任何时候都有且仅有一个Leader存在。

如果集群中Leader不可用了，需要有一个机制来保证能从集群中找出一个最优的服务晋升为Leader继续处理事务和调度等一系列职责。这个过程称为Leader选举。

事务ID（zxid）

数据的新旧使用事务ID判定，事务ID越大认为节点数据约接近Leader的数据，自然应该成为Leader。

服务器ID（myid）

服务id也就是我们配置的`myid`。比如三台服务器，编号分别为1、2、3。

> 编号越大在选举算法中的权重越大。

如果每个参与竞选节点事务ID(zxid)一样，再使用server.id做比较。
server.id是节点在集群中唯一的id，myid文件中配置。

逻辑时钟（epoch）

或者集群投票的纪元，每次接受的投票epoch必须相同，不同则不记录投票，每次投完会接受其他Follower的epoch进行比较，
接受最大的值并且+1，作为新的epoch，并发送给其他的Follower。

过半原则

不管是在集群启动时选举Leader还是集群运行中重新选举Leader。集群中每个Follower角色服务都是以上面的条件作为基础推选出合适的Leader，一旦出现某个节点被过半推选，那么该节点晋升为Leader。

ZooKeeper集群会有很多类型投票。Leader选举投票；事务提议投票；这些投票依赖**过半原则**。就是说ZooKeeper认为投票结果超过了集群总数的一半，便可以安全的处理后续事务。

选举

全新集群初始化时选举

> 目前有3台服务器，每台服务器均没有数据，它们的编号分别是1,2,3按编号（myid）依次启动，它们的选择举过程如下：

1. server.1启动，给自己投票（1,0），然后发投票信息，由于其它机器还没有启动所以它收不到反馈信息，server.1的状态一直属于Looking。
2. server.2启动，给自己投票（2,0），同时与之前启动的server.1交换结果，由于server.2的编号大所以server.2胜出，但此时投票数正好大于半数，所以server.2成为领导者，server.1成为小弟。
3. server.3启动，给自己投票（3,0），同时与之前启动的server.1,server.2换信息，尽管server.3的编号大，但之前server.2已经胜出，所以server.3只能成为小弟。

当确定了Leader之后，每个Server更新自己的状态，Leader将状态更新为Leading，Follower将状态更新为Following。

集群运行阶段选举（非全新集群）

> 假设有3节点组成的集群，分别是server.1（Follower）、server.2（Leader）、server.3（Follower）。此时server.2不可用了。集群会产生以下变化：

集群不可用

因为Leader挂了，集群不可用于事务请求了。

状态变更

所有Follower节点变更自身状态为LOOKING，并且变更自身投票。投票内容就是自己节点的事务ID和myid。我们以(zxid, myid)表示。

假设`server.1`的`zxid`是10，变更的自身投票就是（10，1），`server.3`的`zxid`是8，变更的自身投票就是（8，3）。

首轮投票

将变更的投票发给集群中所有的Follower节点。`server.1`将（10，1）发给集群中所有Follower，包括它自己。`server.3`也一样，将（8，3）发给所有Follower。

所以`server.1`将收到（10，1）和（8，3）两个投票，server.3将收到（8，3）和（10，1）两个投票。

投票PK

每个Follower节点除了发起投票外，还接其他Follower发来的投票，并与自己的投票PK(比较两个提议的事务ID和myid)，PK结果决定是否要变更自身状态并再次投票。

对于`server.1`来说收到（10，1）和（8，3）两个投票，与自己变更的投票比较后没找到比自身投票（10，1）要求更大的，所以`server.1`维持自身投票不变。

对于`server.3`来说收到（8，3）和（10，1）两个投票，与自身变更的投票比较后认为`server.1`发来的投票要比自身的投票大，所以`server.3`会变更自身投票并将变更后的投票发给集群中所有Follower。

也就意味着第一轮**投票失败**，准备开始第二轮投票！

第二轮投票

`server.3`将自身投票变更为（10，1）后再次将投票发给集群中所有Follower。

对于server.1来说在第二轮收到了（10，1）投票，server.1经过PK后继续维持不变。

对于server.3来说在第二轮收到了（10，1）投票，因为server.3自身已变更为（10，3）投票，所以本次也维持不变。

此时server.1和server.3在投票上达成一致。**Leader产生！！**

投票接收桶

节点接收的投票存储在一个接收桶里，每个Follower的投票结果在桶内只记录一次。ZooKeeper源码中

接收桶用Map实现。

下面代码片段是ZooKeeper定义的接收桶，以及向桶内写入数据。Map.Key是Long类型，用来存储投票来源节点的server.id，Vote则是对应节点的投票信息。节点收到投票后会更新这个接收桶，也就是说桶里存储了所有Follower节点的投票并且仅存最后一次的投票结果。

```
1 HashMap<Long, Vote> recvset = new HashMap<Long, Vote>();
2 recvset.put(n.sid, new Vote(n.leader, n.zxid, n.electionEpoch, n.peerEpoch));
```

统计投票

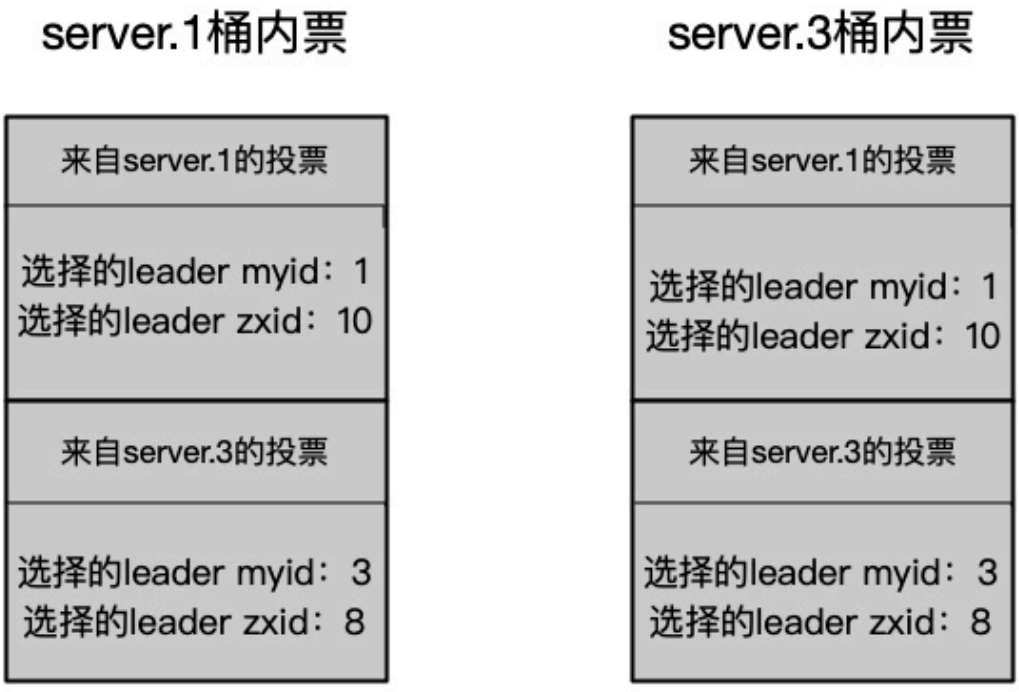
接收到投票后每次都会尝试统计投票，投票统计过半后选举成功。

投票统计的数据来源于投票接收桶里的投票数据，我们从头描述这个场景，来看一下接收桶里的数据变化情况。

server.2挂了后，server.1和server.3发起第一轮投票。

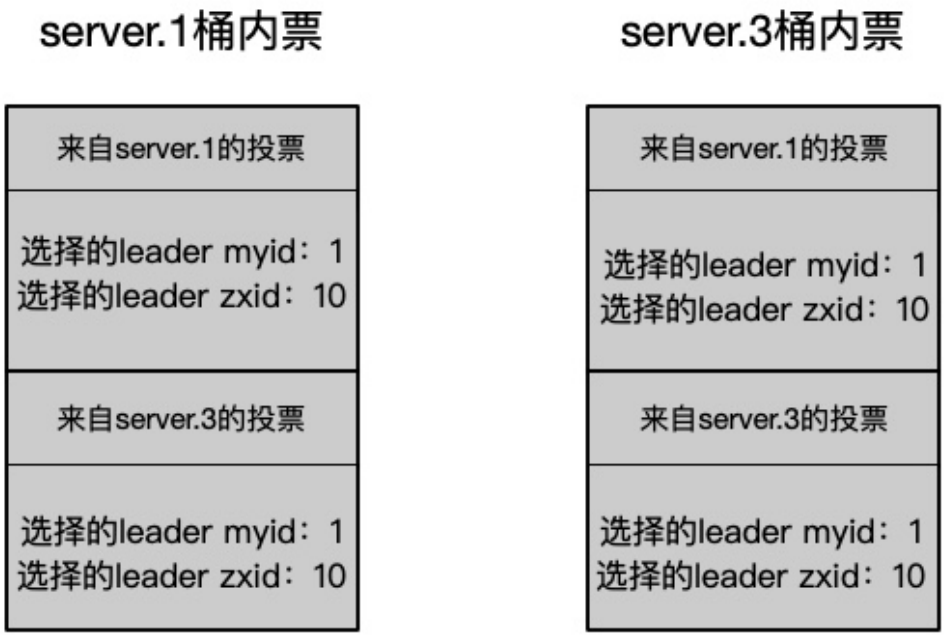
server.1接收到来自server.1的（10，1）投票和来自server.3的（8，3）投票。

server.3同样接收到来自server.1的（10，1）投票和来自server.3的（8，3）投票。此时server.1和server.3接收桶里的数据是这样的：



server.3经过PK后认为server.1的选票比自己要大，所以变更了自己的投票并重新发起投票。

server.1收到了来自server.3的（10，1）投票;server.3收到了来自sever.3的（10，1）投票。此时server.1和server.3接收桶里的数据变成了这样：



基于ZooKeeper过半原则：桶内投票选举server.1作为Leader出现2次，**满足了过半 2 > 3/2 即2>1**。

最后sever.1节点晋升为Leader，server.3变更为Follower，当server.2再次加入时，发现集群中已经存在Leader，他将自动变为Follower。