# Zab协议

* **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。

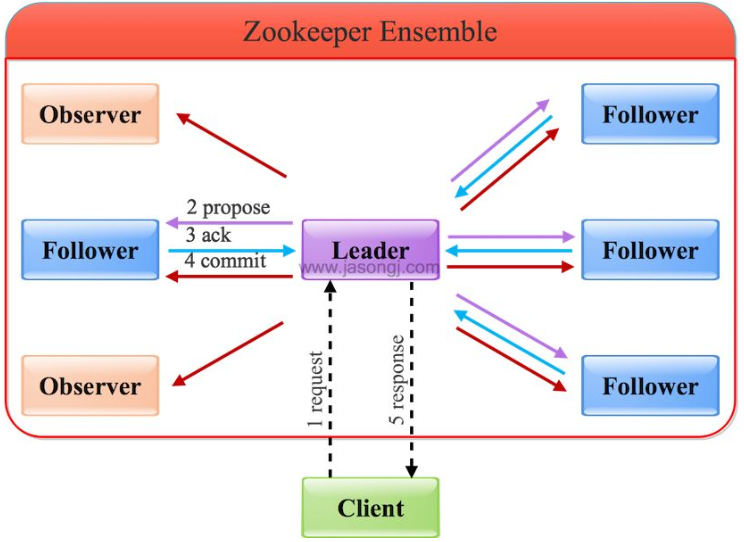
## ****原子广播（ZAB）****

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。

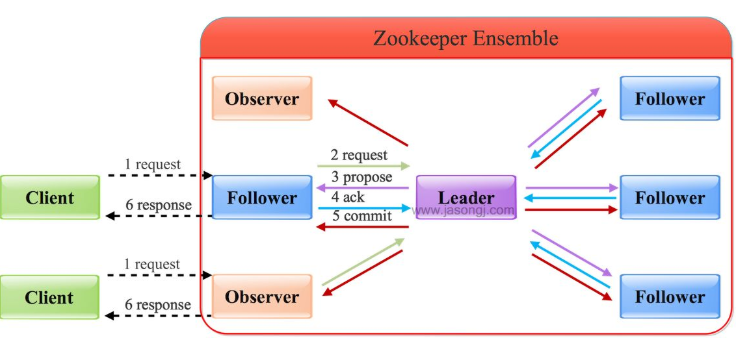
一旦Leader节点无法工作，ZAB协议能够自动从Follower节点中重新选出一个合适的替代者，即新的Leader，该过程即为领导选举。该领导选举过程，是ZAB协议中最为重要和复杂的过程。

### ****写Leader****



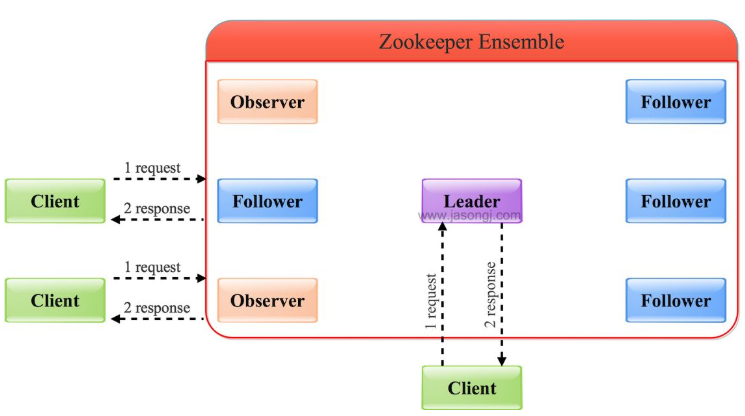
1. 客户端向Leader发起写请求
2. Leader将写请求以Proposal的形式发给所有Follower并等待ACK
3. Follower收到Leader的Proposal后返回ACK
4. Leader得到过半数的ACK（Leader对自己默认有一个ACK）后向所有的Follower和Observer发送Commmit
5. Leader将处理结果返回给客户端

### ****写Follower/Observer****



* 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

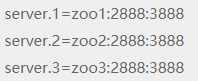
### ****读操作****



## 选主

**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



**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。

**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

### 大致流程

**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：

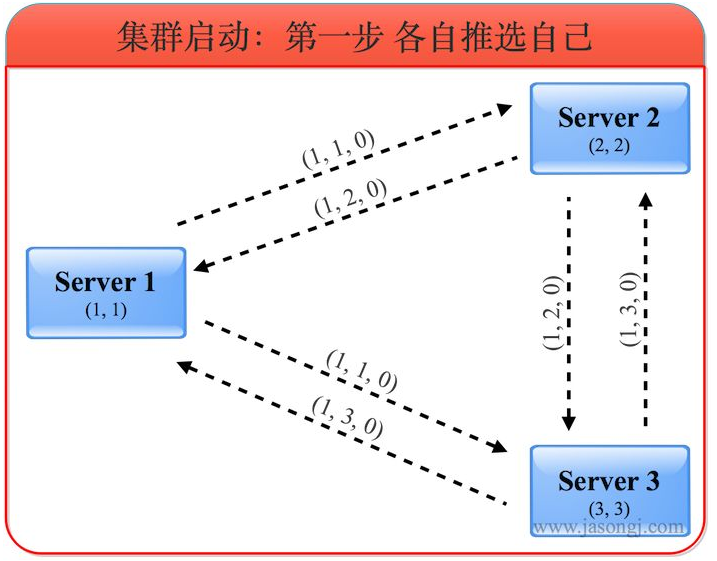
* 外部投票的logicClock大于自己的logicClock，则将自己的logicClock及自己的选票的logicClock变更为收到的logicClock
* 若logicClock一致，则对比二者的vote\_zxid，若外部投票的vote\_zxid比较大，则将自己的票中的vote\_zxid与vote\_myid更新为收到的票中的vote\_zxid与vote\_myid并广播出去，另外将收到的票及自己更新后的票放入自己的票箱。如果票箱内已存在(self\_myid, self\_zxid)相同的选票，则直接覆盖
* 若二者vote\_zxid一致，则比较二者的vote\_myid，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

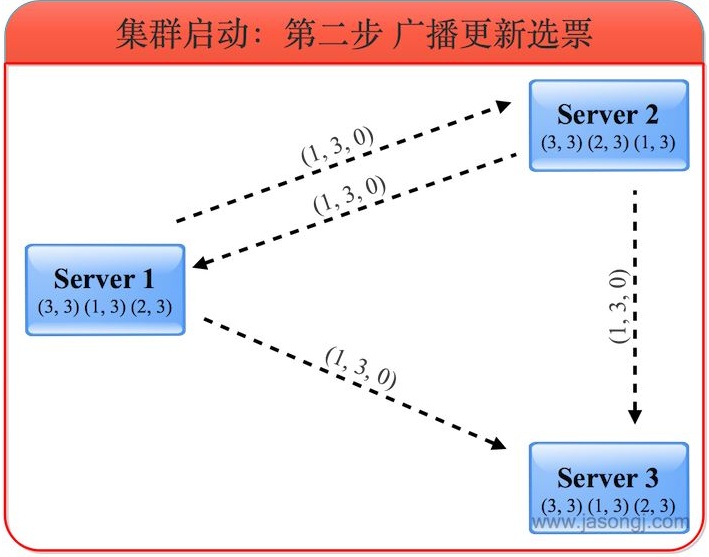
如果已经确定有过半服务器认可了自己的投票（可能是更新后的投票），则终止投票。否则继续接收其它服务器的投票。

### ****例子****

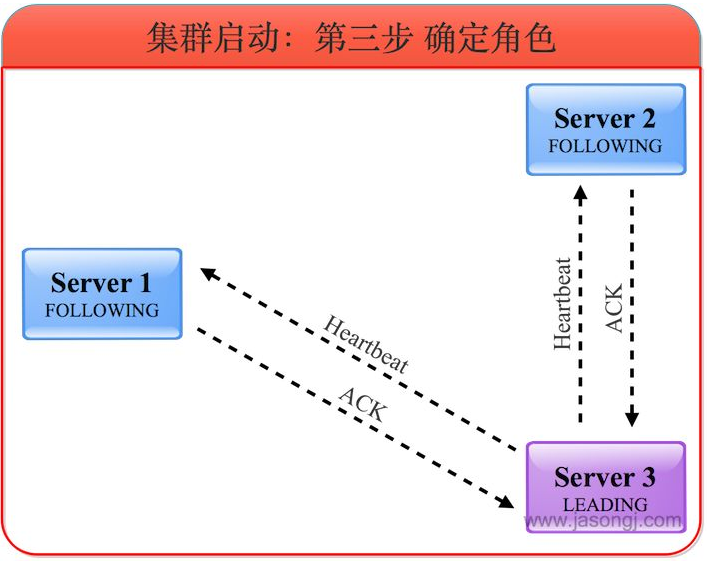
#### 初始选举



(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。

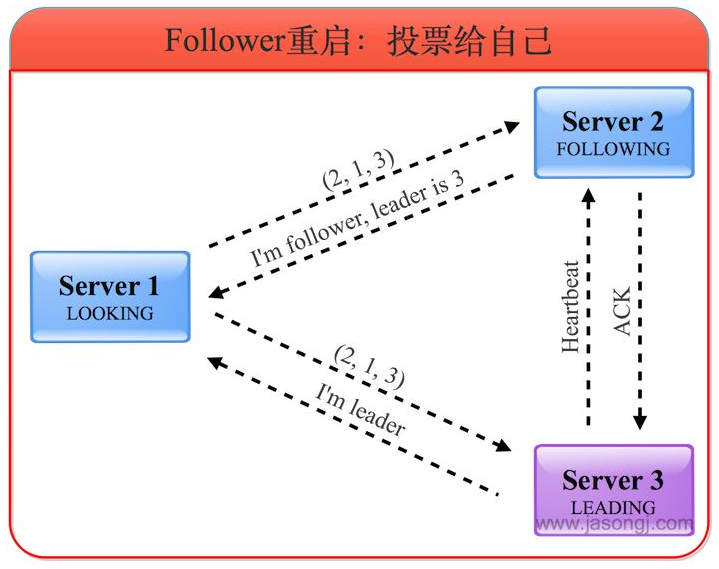


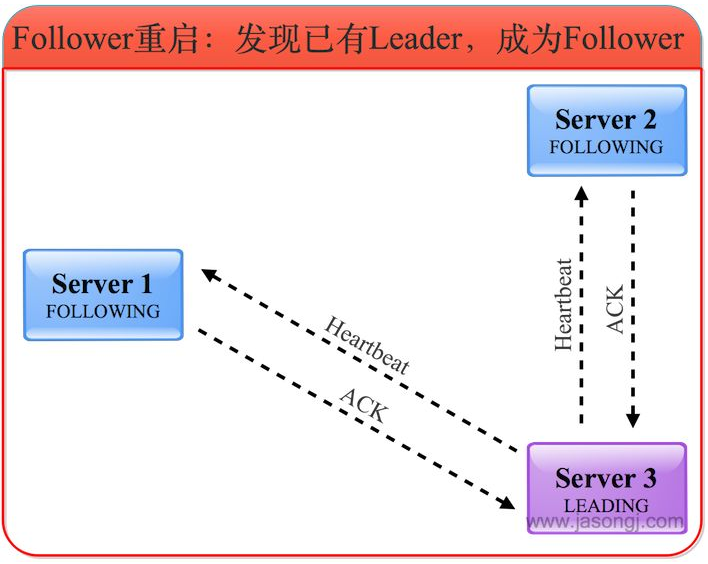
服务器1收到服务器2的选票（1, 2, 0）和服务器3的选票（1, 3, 0）后，由于所有的logicClock都相等，所有的zxid都相等，因此根据myid判断应该将自己的选票按照服务器3的选票更新为（1, 3, 0），并将自己的票箱全部清空，再将服务器3的选票与自己的选票存入自己的票箱，接着将自己更新后的选票广播出去。此时服务器1票箱内的选票为(1, 3)，(3, 3)。  
 同理，服务器2收到服务器3的选票后也将自己的选票更新为（1, 3, 0）并存入票箱然后广播。此时服务器2票箱内的选票为(2, 3)，(3, ,3)。服务器3根据上述规则，无须更新选票，自身的票箱内选票仍为（3, 3）。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。



#### 重新选举（失败）

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。

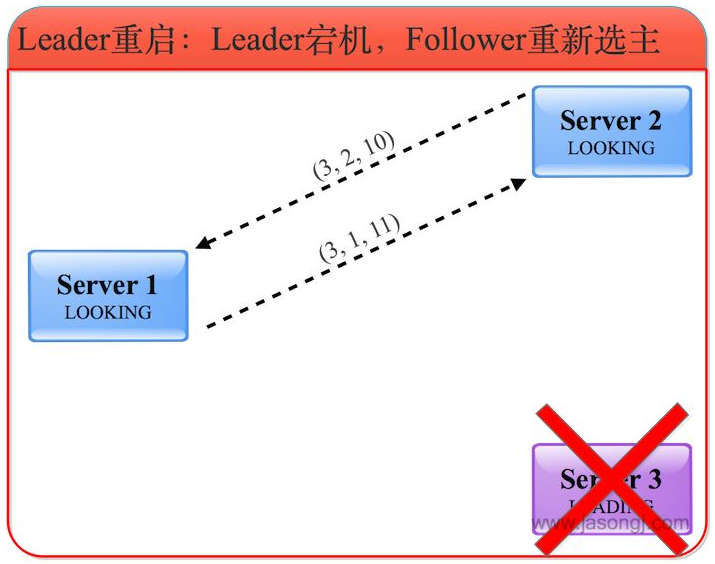




服务器3收到服务器1的投票后，将自己的状态LEADING以及选票返回给服务器1。服务器2收到服务器1的投票后，将自己的状态FOLLOWING及选票返回给服务器1。此时服务器1知道服务器3是Leader，并且通过服务器2与服务器3的选票可以确定服务器3确实得到了超过半数的选票。因此服务器1进入FOLLOWING状态。

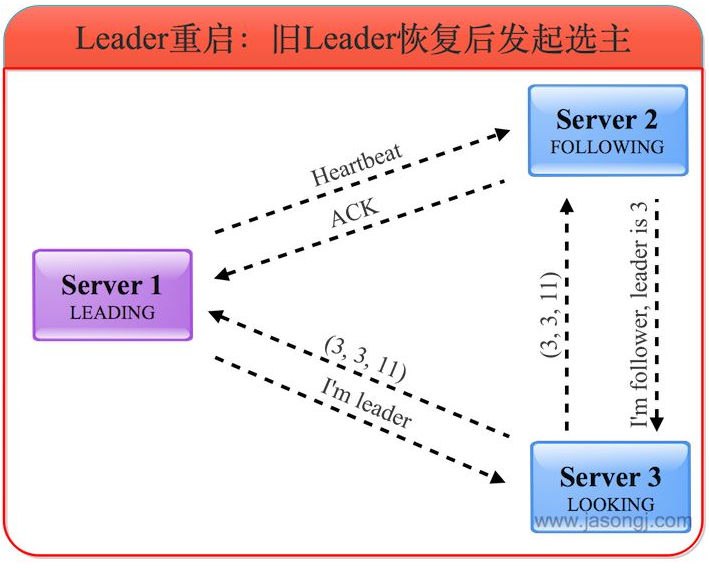
#### 重新选举（成功）

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。



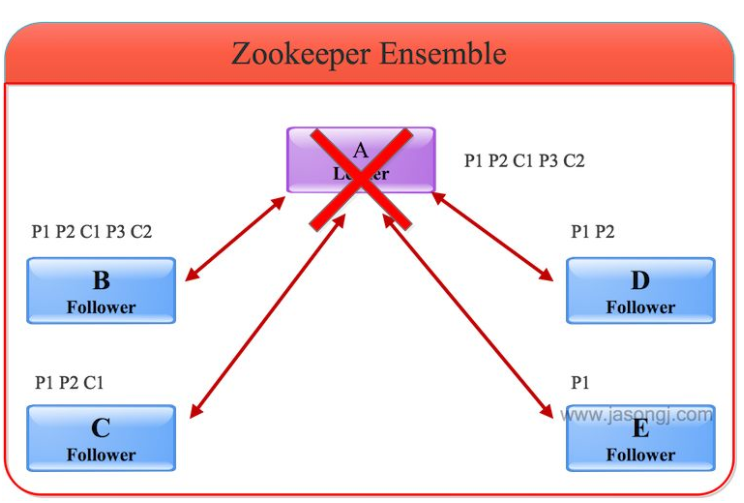
服务器1和2根据外部投票确定是否要更新自身的选票。这里有两种情况：

* 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

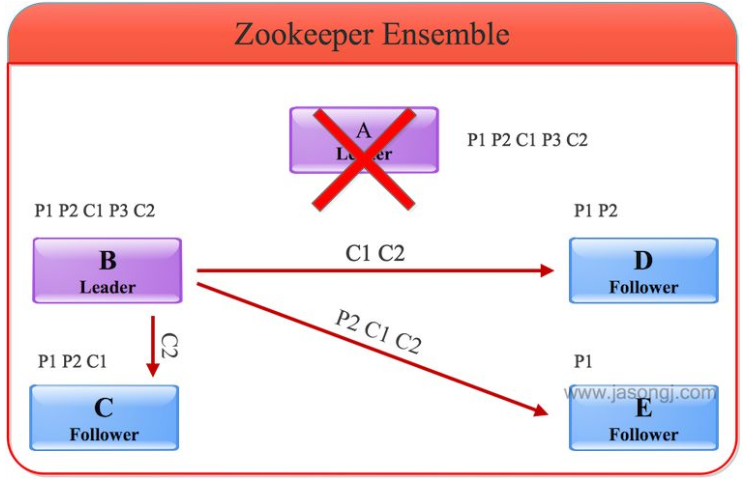


## Zab的优势

### ****Commit过的数据不丢失****



* 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



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。

### ****未Commit过的消息对客户端不可见****

Zab 通过巧妙的设计 zxid 来实现这一目的。一个 zxid 是64位，高 32 是纪元（epoch）编号，每经过一次 leader 选举产生一个新的 leader，新 leader 会将 epoch 号 +1。低 32 位是消息计数器，每接收到一条消息这个值 +1，新 leader 选举后这个值重置为 0。这样设计的好处是旧的 leader 挂了后重启，它不会被选举为 leader，因为此时它的 zxid 肯定小于当前的新 leader。当旧的 leader 作为 follower 接入新的 leader 后，新的 leader 会让它将所有的拥有旧的 epoch 号的未被 COMMIT 的 proposal 清除。