# 深入浅出Zookeeper（一） Zookeeper架构及FastLeaderElection机制

本文介绍了Zookeeper的架构，并组合实例分析了原子广播(ZAB)协议的原理，包括但不限于Zookeeper的读写流程，FastLeaderElection算法的原理，ZAB如何保证Leader Failover过程中的数据一致性。

原创文章，转载请务必将下面这段话置于文章开头处。  
本文转发自**[技术世界](http://www.jasongj.com/)**，[原文链接](http://www.jasongj.com/zookeeper/fastleaderelection)　[http://www.jasongj.com/zookeeper/fastleaderelection/](http://www.jasongj.com/zookeeper/fastleaderelection)

# **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。

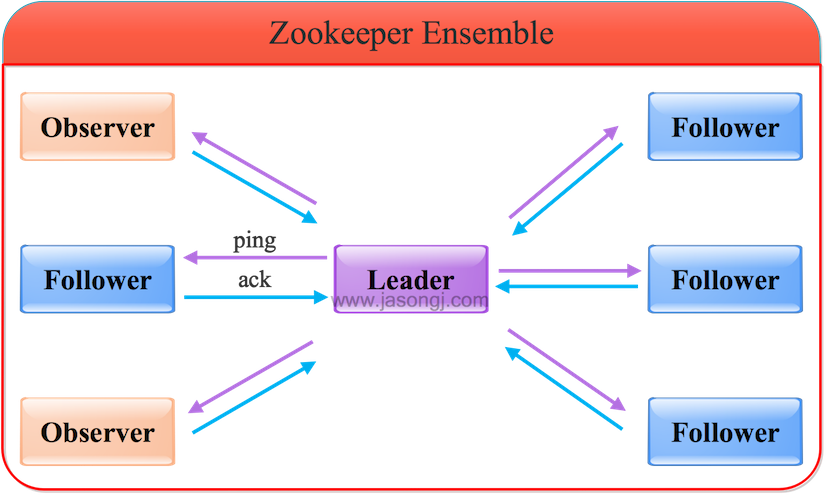
既然是一个文件系统，就不得不提Zookeeper是如何保证数据的一致性的。本文将介绍Zookeeper如何保证数据一致性，如何进行领导选举，以及数据监控/通知机制的语义保证。

# **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。

[](http://www.jasongj.com/img/zookeeper/1_architecture/architecture.png)

## **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。

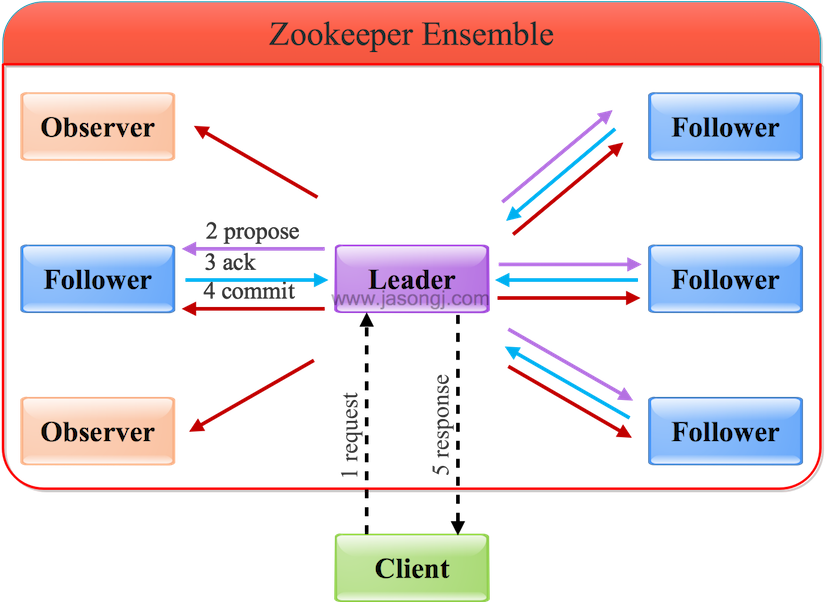
根据ZAB协议，所有的写操作都必须通过Leader完成，Leader写入本地日志后再复制到所有的Follower节点。

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

[](http://www.jasongj.com/img/zookeeper/1_architecture/writeleader.png)

由上图可见，通过Leader进行写操作，主要分为五步：

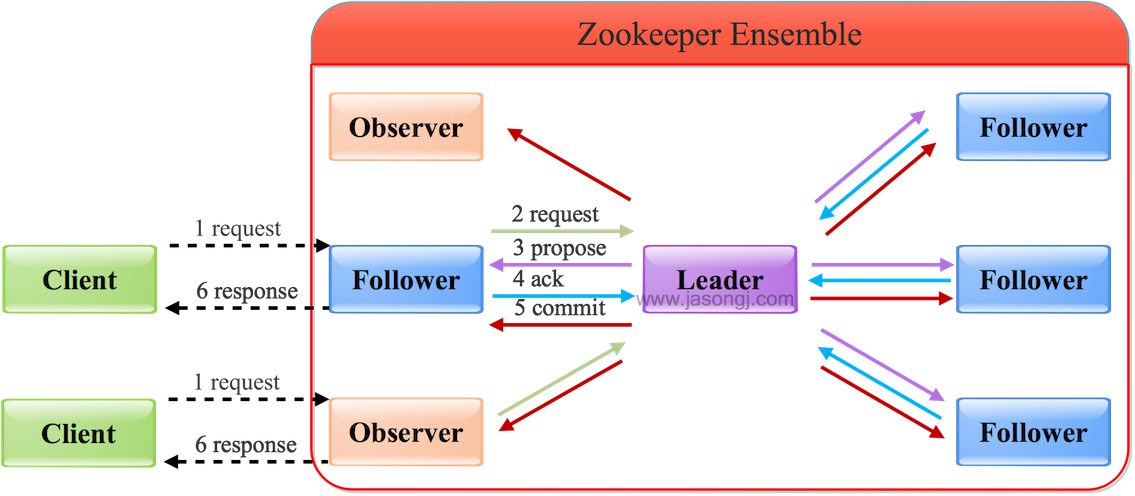
1. 客户端向Leader发起写请求
2. Leader将写请求以Proposal的形式发给所有Follower并等待ACK
3. Follower收到Leader的Proposal后返回ACK
4. Leader得到过半数的ACK（Leader对自己默认有一个ACK）后向所有的Follower和Observer发送Commmit
5. Leader将处理结果返回给客户端

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：

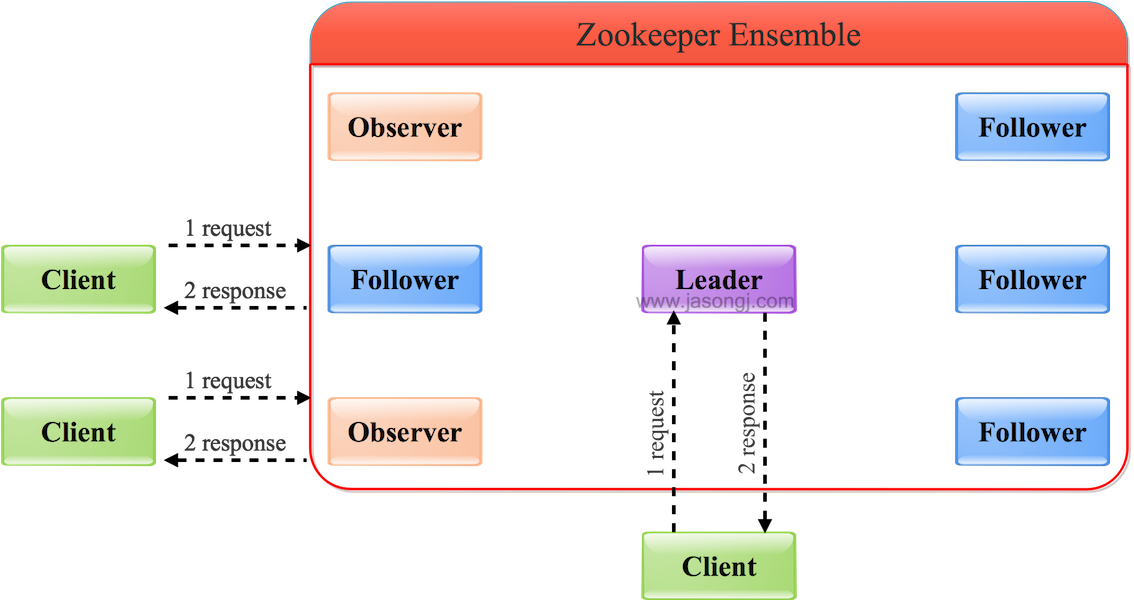
[](http://www.jasongj.com/img/zookeeper/1_architecture/writefollower.png)

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。

[](http://www.jasongj.com/img/zookeeper/1_architecture/read.png)

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

|  |  |
| --- | --- |
| 1 2 3 | server.1=zoo1:2888:3888 server.2=zoo2:2888:3888 server.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。

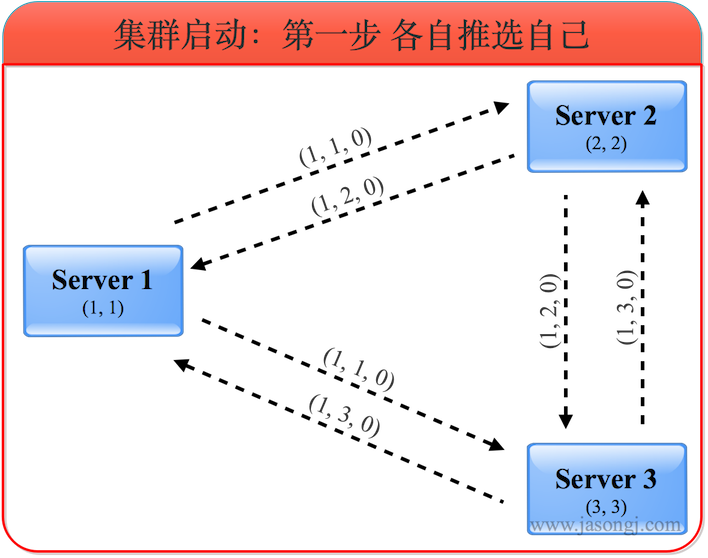
**更新服务器状态**  
投票终止后，服务器开始更新自身状态。若过半的票投给了自己，则将自己的服务器状态更新为LEADING，否则将自己的状态更新为FOLLOWING

## **几种领导选举场景**

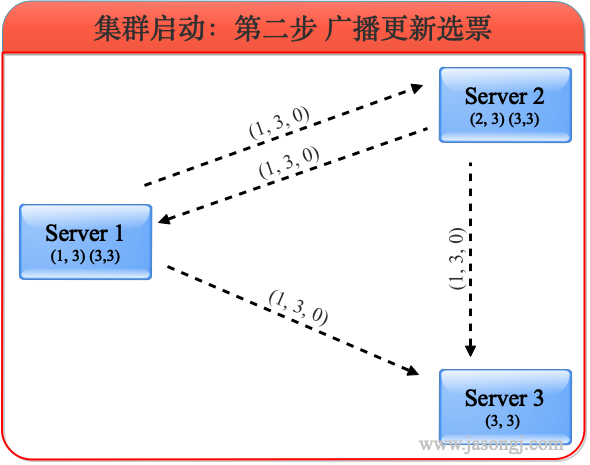
### **集群启动领导选举**

**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。

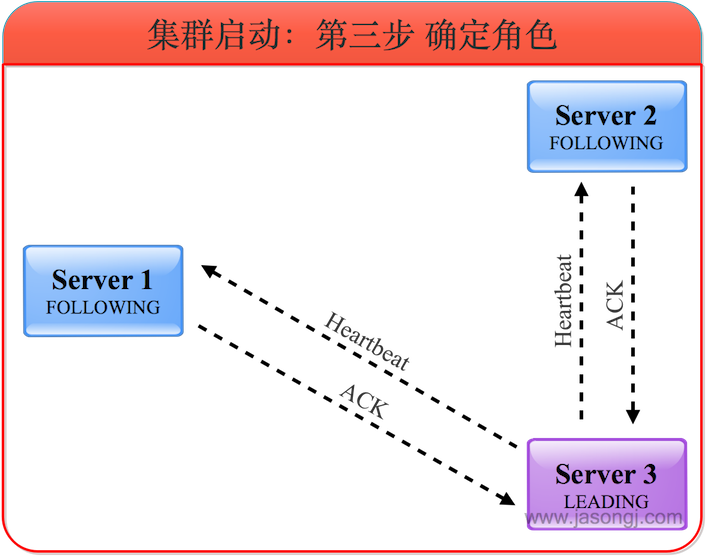
各服务器初始化后，都投票给自己，并将自己的一票存入自己的票箱，如下图所示。

[](http://www.jasongj.com/img/zookeeper/1_architecture/start_election_1.png)

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。

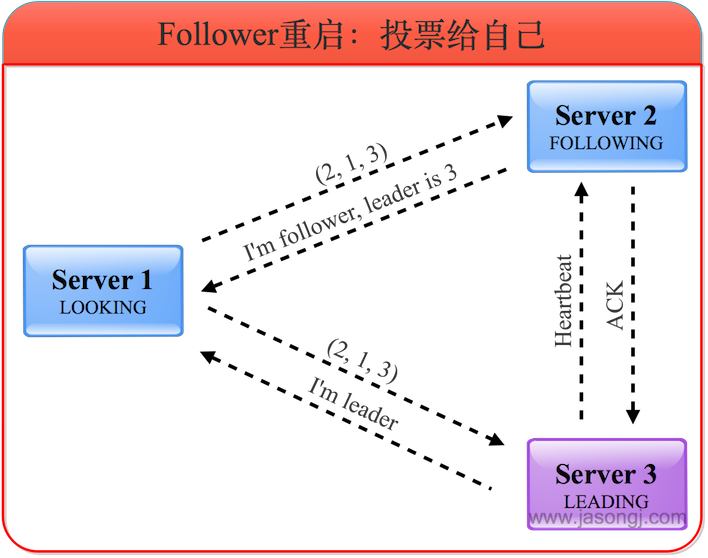
[](http://www.jasongj.com/img/zookeeper/1_architecture/start_election_2.png)

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。

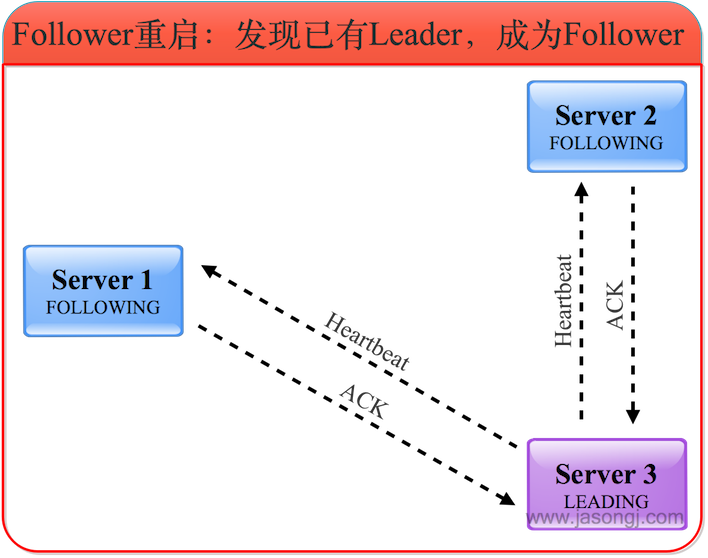
[](http://www.jasongj.com/img/zookeeper/1_architecture/start_election_3.png)

### **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。

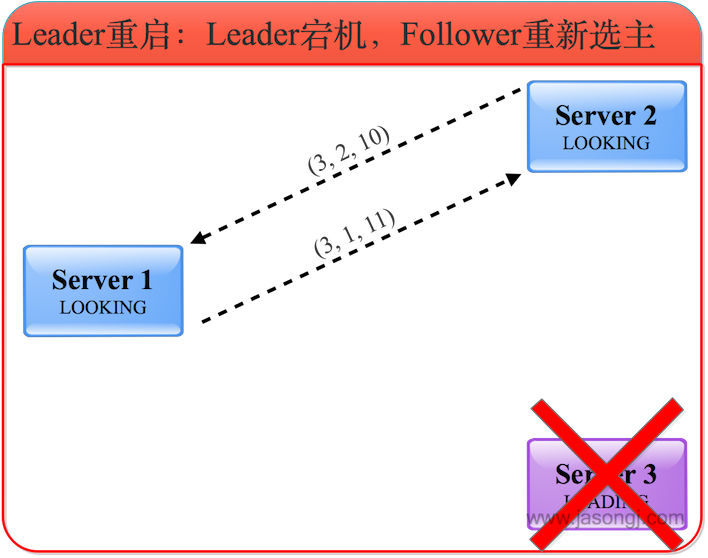
[](http://www.jasongj.com/img/zookeeper/1_architecture/follower_restart_election_1.png)

**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。

[](http://www.jasongj.com/img/zookeeper/1_architecture/follower_restart_election_2.png)

### **Leader重启**

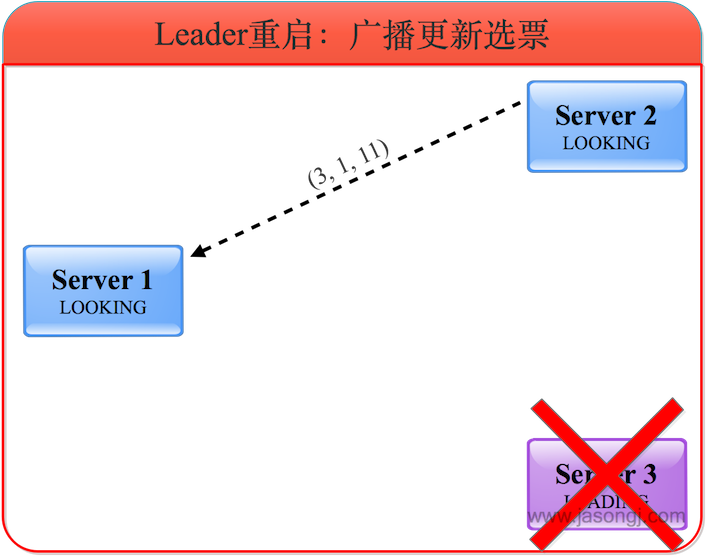
**Follower发起新投票**  
Leader（服务器3）宕机后，Follower（服务器1和2）发现Leader不工作了，因此进入LOOKING状态并发起新的一轮投票，并且都将票投给自己。

[](http://www.jasongj.com/img/zookeeper/1_architecture/leader_restart_election_1.png)

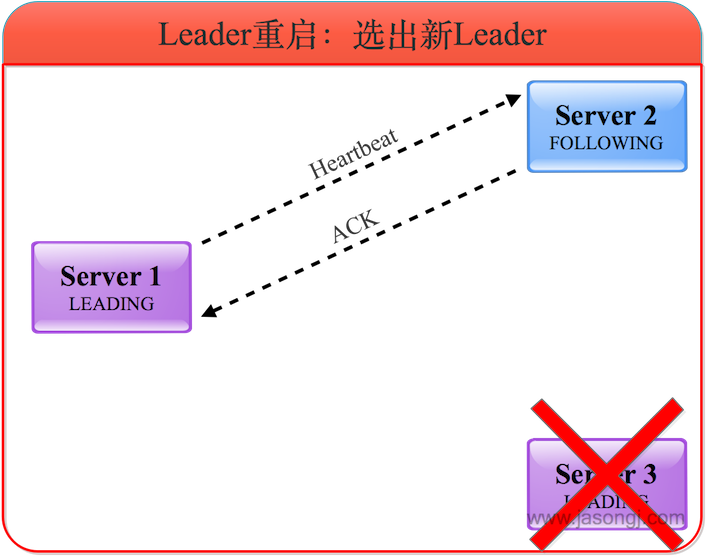
**广播更新选票**  
服务器1和2根据外部投票确定是否要更新自身的选票。这里有两种情况

* 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

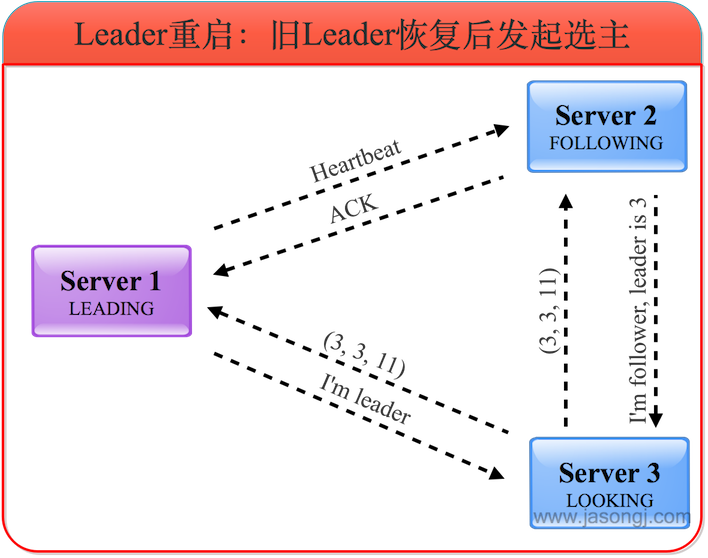
在上图中，服务器1的zxid为11，而服务器2的zxid为10，因此服务器2将自身选票更新为（3, 1, 11），如下图所示。

[](http://www.jasongj.com/img/zookeeper/1_architecture/leader_restart_election_2.png)

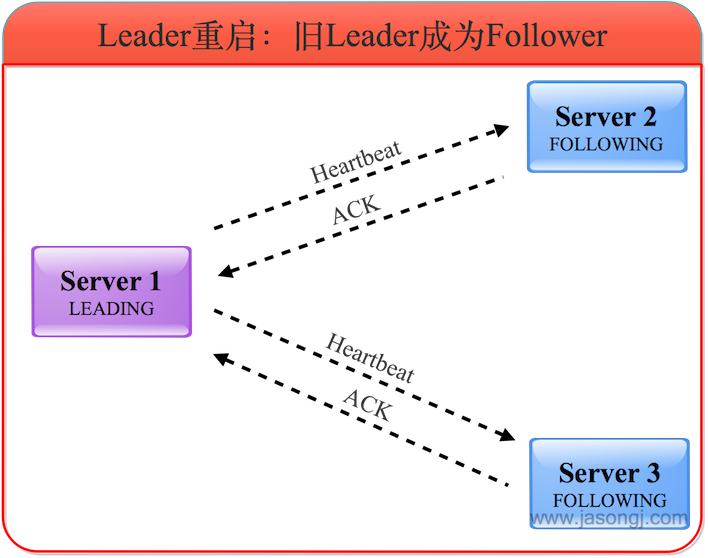
**选出新Leader**  
经过上一步选票更新后，服务器1与服务器2均将选票投给服务器1，因此服务器2成为Follower，而服务器1成为新的Leader并维护与服务器2的心跳。

[](http://www.jasongj.com/img/zookeeper/1_architecture/leader_restart_election_3.png)

**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。

[](http://www.jasongj.com/img/zookeeper/1_architecture/leader_restart_election_4.png)

**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。

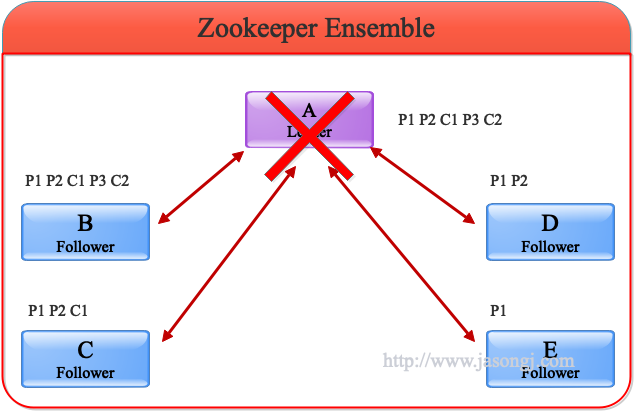
[](http://www.jasongj.com/img/zookeeper/1_architecture/leader_restart_election_5.png)

# **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**

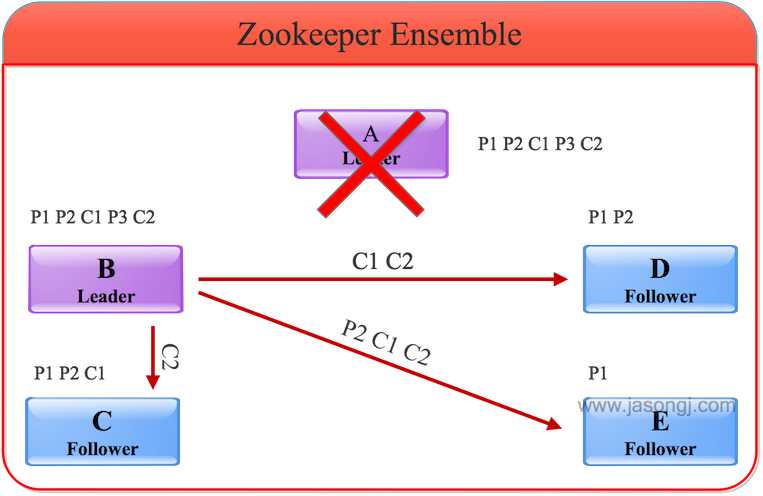
**Failover前状态**  
为更好演示Leader Failover过程，本例中共使用5个Zookeeper服务器。A作为Leader，共收到P1、P2、P3三条消息，并且Commit了1和2，且总体顺序为P1、P2、C1、P3、C2。根据顺序性原则，其它Follower收到的消息的顺序肯定与之相同。其中B与A完全同步，C收到P1、P2、C1，D收到P1、P2，E收到P1，如下图所示。

[](http://www.jasongj.com/img/zookeeper/1_architecture/recovery_1.png)

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

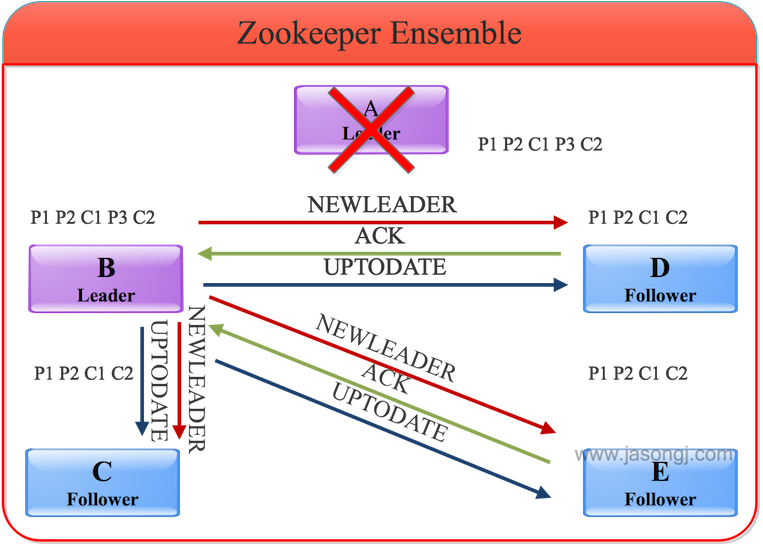
**选出新Leader**  
旧Leader也即A宕机后，其它服务器根据上述FastLeaderElection算法选出B作为新的Leader。C、D和E成为Follower且以B为Leader后，会主动将自己最大的zxid发送给B，B会将Follower的zxid与自身zxid间的所有被Commit过的消息同步给Follower，如下图所示。

[](http://www.jasongj.com/img/zookeeper/1_architecture/recovery_2.png)

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。

[](http://www.jasongj.com/img/zookeeper/1_architecture/recovery_3.png)

## **未Commit过的消息对客户端不可见**

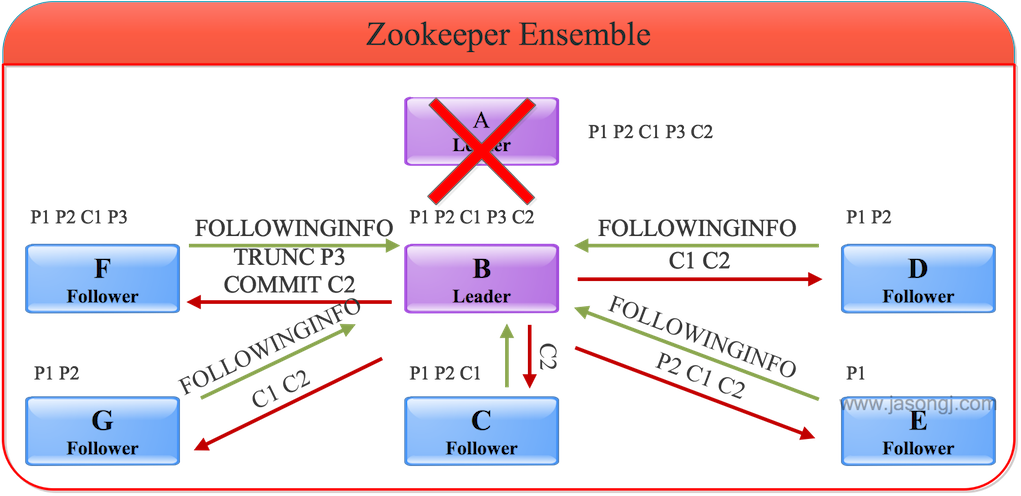
在上例中，P3未被A Commit过，同时因为没有过半的服务器收到P3，因此B也未Commit P3（如果有过半服务器收到P3，即使A未Commit P3，B会主动Commit P3，即C3），所以它不会将P3广播出去。

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。

[](http://www.jasongj.com/img/zookeeper/1_architecture/recovery_4.png)

# **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